HAL
open science

Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LinBox

Pascal Giorgi

To cite this version:

Pascal Giorgi. Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LinBox. Génie logiciel [cs.SE]. Ecole normale supérieure de lyon - ENS LYON, 2004. Français. NNT : . tel-00008951

HAL Id: tel-00008951

https://theses.hal.science/tel-00008951v1

Submitted on 4 Apr 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 303
N° attribué par la bibliothèque : 04ENSL303

ÉCOLE NORMALE SUPÉRIEURE DE LYON

Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'école doctorale MathIf

présentée et soutenue publiquement le 20 décembre 2004

par **Pascal Giorgi**

ARITHMÉTIQUE ET ALGORITHMIQUE EN ALGÈBRE LINÉAIRE EXACTE POUR LA BIBLIOTHÈQUE LINBOX

Après avis de : M. Bernard Mourrain
M. Jean-Louis Roch

Devant la commission d'examen formée de :

M. Bernard Mourrain	Membre/Rapporteur
M. Yves Robert	Membre
M. Jean-Louis Roch	Membre/Rapporteur
M. Bruno Salvy	Membre/Président du jury
M. Gilles Villard	Membre/Directeur de thèse

N° d'ordre : 303
N° attribué par la bibliothèque : 04ENSL303

ÉCOLE NORMALE SUPÉRIEURE DE LYON

Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'école doctorale MathIf

présentée et soutenue publiquement le 20 décembre 2004

par **Pascal Giorgi**

ARITHMÉTIQUE ET ALGORITHMIQUE EN ALGÈBRE LINÉAIRE EXACTE POUR LA BIBLIOTHÈQUE LINBOX

Après avis de : M. Bernard Mourrain
M. Jean-Louis Roch

Devant la commission d'examen formée de :

M. Bernard Mourrain	Membre/Rapporteur
M. Yves Robert	Membre
M. Jean-Louis Roch	Membre/Rapporteur
M. Bruno Salvy	Membre/Président du jury
M. Gilles Villard	Membre/Directeur de thèse

Remerciements

En premier lieu, je tiens à remercier mon directeur de thèse Gilles Villard sans qui cette thèse n'aurait jamais pu voir le jour. Je le remercie pour son encadrement, sa disponibilité, ses conseils, sa clairvoyance et pour sa générosité aussi bien dans le travail que dans la vie de tous les jours.

Je remercie Jean-Michel Muller et Jean-Claude Bajard qui sont à l'origine de ma candidature pour une thèse à l'ENS Lyon et qui m'ont permis de m'intéresser à la recherche scientifique.

Je remercie également Jean-Louis Roch et Bernard Mourrain qui ont accepté la lourde tâche de rapporter sur ce manuscrit et qui m'ont fortement encouragé pour son accomplissement ainsi que pour la poursuite de mes travaux de recherche. Un grand merci à tout les membres du jury pour l'attention portée à mon travail et pour avoir accepter une date de soutenance aussi proche de noël.

Je remercie fortement Claude-Pierre Jeannerod, Arnaud Tisserand et Nathalie Revol pour leur soutien au quotidien, leur encouragement et pour toutes les discussions sérieuses et moins sérieuses que nous avons pu partager autour d'un bon café.

Merci aux autres personnes de l'équipe Arénaire de m'avoir aussi bien acceuilli et supporté pendant ces trois années : David, Nicolas W, Marc, Florent, Jean-Luc, Catherine, Jéremy, Guillaume, Saurhab, Romain, Sylvie et Nicolas.

Je tiens également à remercier l'ensemble des membres du projet LinBox sans qui je n'aurai pu développer l'ensemble des codes disponibles à l'heure actuelle dans la bibliothèque. Plus particulièrement, merci à Jean-Guillaume et Clément pour notre collaboration autour du projet FFLAS-FFPACK ainsi que pour l'enthousiasme et la bonne humeur que vous m'avez apporté. Merci aussi à Erich Kaltofen, Dave Saunders et Mark Giesbrecht pour l'ensemble des visites en Amérique du nord et un grand merci a Zhendong pour l'ensemble des corrections de bugs.

Je remercie ma famille pour leur comprehension et leur soutien sans faille tout au long de ces trois annés. Je remercie également tous mes amis Alex, Mehdi, Chass, Cissou, Bruno, Sév pour les moment de détente que nous avons passé ensemble. Enfin, je remercie Séverine pour la patience et la gentillesse dont elle a fait preuve pendant tout ce temps passé loin d'elle.

Table des matières

Introduction **1**

1 Organisation et motivation du projet LinBox **7**

1.1 Choix du langage C++ 9

1.1.1 Classes : abstraction, accès et hiérarchisation 10

1.1.2 Polymorphisme et généricité 13

1.1.3 Spécialisations et caractérisations 16

1.2 Utilisation de bibliothèques spécialisées 18

1.2.1 GMP 18

1.2.2 Givaro 19

1.2.3 NTL 19

1.2.4 LiDIA 20

1.2.5 BLAS 20

1.3 Interfaces utilisateurs 21

1.3.1 Maple 21

1.3.2 GAP 22

1.3.3 Serveurs web 23

1.4 Organisation des codes 23

2 Arithmétique des corps finis **27**
2.1 Archétype de données 28
2.1.1 Modèle de base des corps finis 29
2.1.2 Interface compilable 31
2.1.3 Implantation 34
2.1.4 Performances *vs* généricités 38
2.2 Corps finis premiers 45
2.2.1 Modular 45
2.2.2 GivaroZpz standard 48
2.2.3 GivaroZpz : base logarithmique (*Zech's log*) 48
2.2.4 GivaroZpz : base de Montgomery 50
2.2.5 NTL 51
2.2.6 Performances et surcoût des *wrappers* 53
2.3 Extension algébrique $GF(p^k)$ 59
2.3.1 Givaro 59
2.3.2 NTL 60
2.3.3 LiDIA 61
2.3.4 Performances et surcoût des *wrappers* 63
2.4 Conclusion 65

3 Algèbre linéaire dense sur un corps fini **67**
3.1 Systèmes linéaires triangulaires 69
3.1.1 Algorithme récursif par blocs 70
3.1.2 Utilisation de la routine "dtrsm" des BLAS 71
3.1.3 Utilisation de réductions modulaires retardées 74
3.1.4 Comparaison des implantations 75
3.2 Triangularisations de matrices 78
3.2.1 Factorisation LSP 78
3.2.2 LUdivine 80
3.2.3 LQUP 81
3.2.4 Performances et comparaisons 81
3.3 Applications des triangularisations 83
3.3.1 Rang et déterminant 83
3.3.2 Inverse 84
3.3.3 Base du noyau 85
3.3.4 Alternative à la factorisation LQUP : Gauss-Jordan 85
3.4 Interfaces pour le calcul "exact/numérique" 89

3.4.1 Interface avec les BLAS 90
3.4.2 Connections avec MAPLE 95
3.4.3 Intégration et utilisation dans LinBox 100

4 Systèmes linéaires entiers 119
4.1 Solutions rationnelles 122
4.1.1 Développement p-adique de la solution rationnelle 122
4.1.2 Reconstruction de la solution rationnelle 123
4.1.3 Algorithme complet 124
4.2 Interface pour la résolution des systèmes linéaires entiers 125
4.2.1 RationalSolver 126
4.2.2 LiftingContainer et LiftingIterator 128
4.2.3 RationalReconstruction 130
4.3 Algorithme de Dixon 135
4.3.1 Cas non singulier 137
4.3.2 Cas singulier et certificat d'inconsistance 139
4.3.3 Solutions aléatoires 144
4.3.4 Optimisations et performances 145
4.4 Solutions diophantiennes 150
4.4.1 Approche proposée par Giesbrecht 151
4.4.2 Certificat de minimalité 153
4.4.3 Implantations et performances 156

Conclusion et perspectives 161

Annexes 173

A Code LinBox 173
A.1 Développements p-adiques de systèmes linéaires entiers 173
A.2 Reconstruction de la solution rationnelle 175
A.3 Résolution de systèmes linéaires entiers singuliers avec l'algorithme de Dixon 178
A.4 Produits matrice-vecteur et matrice-matrice en représentation q-adique . . . 183

Table des figures 189

Liste des tableaux 191

Introduction

À la différence du calcul numérique qui s'attache à calculer les meilleures approximations possibles d'un résultat, le calcul formel ou symbolique consiste à calculer les solutions de manière exacte. Dans le cadre de l'algèbre linéaire, le calcul numérique bénéficie de plusieurs années d'expérience, tant au niveau mathématique qu'informatique. Le développement de routines de calcul exploitant au maximum les caractéristiques des processeurs et des unités de calcul flottantes a permis de fournir une puissance de calcul incomparable pour les opérations de base en algèbre linéaire. On connaît notamment la hiérarchie des BLAS (*Basic Linear Algebra Subroutines*) qui propose, en particulier, des routines de multiplication de matrices permettant d'obtenir les meilleures performances pour cette opération. On s'efforce donc d'exprimer l'ensemble des algorithmes pour les problèmes majeurs en algèbre linéaire à partir de ces routines. C'est exactement ce qui est fait par la bibliothèque LAPACK [1] qui fournit aujourd'hui un véritable standard pour le calcul en algèbre linéaire numérique. La réutilisation de cette bibliothèque est devenue indispensable pour résoudre certains problèmes provenant d'applications concrètes.

Actuellement, il n'existe aucun équivalent de ces bibliothèques pour le calcul en algèbre linéaire exacte. La difficulté de conception d'un tel outil de calcul se situe à deux niveaux, d'abord sur un plan théorique, où certains gains en complexité ne sont que très récents dans le domaine. En effet, le problème central, qui est le grossissement de la taille des données durant les calculs, est encore difficile à maîtriser et entraîne un lourd tribut sur le nombre d'opérations arithmétiques nécessitées par les algorithmes. Ensuite, sur un plan pratique, la diversité des calculs et des arithmétiques utilisées ne permet pas de définir un véritable standard. Les calculs peuvent, par exemple, nécessiter l'utilisation d'arithmétiques basées sur des entiers, des polynômes, des corps de fractions, des corps finis ou des extensions algébriques.

L'une des clés pour proposer des solutions performantes en calcul exact réside donc dans la conception de supports logiciels portables et efficaces pour l'ensemble des arithmétiques nécessaires. Ces supports arithmétiques sont aujourd'hui disponibles à partir de bibliothèques de calcul spécialisées. Par exemple, la bibliothèque GMP[2] s'impose depuis quelques années comme le standard pour l'arithmétique des grands entiers. De même, des bibliothèques comme NTL[3] ou Givaro[4] proposent des ensembles d'arithmétiques diverses très convaincantes. Du fait de l'émergence de ces bibliothèques spécialisées très performantes, portables et facilement réutilisables, le logiciel en calcul formel n'est plus dominé par des systèmes généralistes comme MAPLE ou MATHEMATICA qui proposent des codes propriétaires peu réutilisables et peu satisfaisants pour des calculs qui demandent de très hautes performances.

La conception de logiciels fédérant l'ensemble des codes définis par des bibliothèques spécialisées au sein d'un environnement de calcul commun devient un réel besoin pour la résolution de problèmes en calcul formel qui nécessitent de plus en plus de puissance de calcul. Cette thèse s'inspire de cette tendance récente pour proposer une bibliothèque de calcul générique en algèbre linéaire exacte. L'un des intérêts de la généricité est de faciliter l'intégration de composants externes et d'offrir une réutilisation des codes simplifiée. Cette bibliothèque est la concrétisation logicielle du projet LinBox, issu d'une collaboration internationale sur le thème de l'algèbre linéaire exacte.

Contrairement au calcul numérique, où la précision des calculs est fixée par les types flottants des processeurs, la précision utilisée pour des calculs exacts sur des nombre entiers est poten-

[1] http://www.netlib.org/lapack/
[2] http://www.swox.com/gmp
[3] http://www.shoup.net/ntl
[4] http://www-lmc.imag.fr/Logiciels/givaro/

tiellement infinie. Toutefois, certaines approches classiques du calcul formel comme le théorème des restes chinois ou les développements p-adiques permettent d'effectuer l'essentiel des calculs en précision finie et d'utiliser une précision infinie uniquement pour la reconstruction de la solution exacte. L'utilisation des corps finis est alors un bon moyen pour fournir des implantations efficaces pour ces approches, en particulier, du fait que la théorie algorithmique sur ces objets mathématiques se rapproche fortement de celle développée pour le calcul numérique.

Les problèmes que nous étudions dans cette thèse concernent la conception et la validation de boîtes à outils génériques performantes pour l'implantation d'algorithmes de l'algèbre linéaire exacte et l'intégration de codes externes. En premier lieu, notre objectif est de développer et de valider une approche permettant le branchement à la demande (*plug and play*) de composants externes (*plugins*) ; pour cela, nous nous appuyons sur la notion d'interfaces abstraites et de fonctions virtuelles du langage C++. Dans ce cadre précis, et dans le but de fournir des implantations algorithmiques bénéficiant des meilleures arithmétiques de corps finis, nous étudions la faisabilité et l'efficacité de la réutilisation de bibliothèques spécialisées pour définir un noyau de corps finis facilement interchangeable dans la bibliothèque LinBox.

Un des problèmes majeurs que nous considérons concerne la possibilité du développement de routines de calcul sur les corps finis similaires à celles proposées par les bibliothèques numériques BLAS/LAPACK, à savoir des routines portables, optimisées en fonction des caractéristiques des processeurs et facilement réutilisables. Notre démarche pour aborder ce vaste problème, a consisté à utiliser des calculs hybrides "exact/numérique" permettant la réutilisation des routines numériques BLAS pour le calcul sur les corps finis. Cette approche se base sur l'utilisation d'algorithmes par blocs se réduisant au produit de matrices et permettant ainsi de bénéficier des très bonnes performances du niveau trois des BLAS. De cette façon, nous proposons une brique de base pour les problèmes fondamentaux d'algèbre linéaire sur les corps finis permettant d'approcher l'efficacité obtenue par la bibliothèque LAPACK .

Afin de valider l'ensemble des fonctionnalités que nous proposons au sein de la bibliothèque LinBox, nous étudions l'implantation d'une application d'algèbre linéaire sur les entiers permettant de solliciter une multitude de caractéristiques de calcul sur les corps finis et les entiers, ainsi que différentes méthodes algorithmiques selon que les matrices sont denses, creuses ou structurées ; nous nous intéressons à la résolution de systèmes linéaires diophantiens. L'utilisation de mécanismes d'abstraction de calcul et d'optimisations en fonction des instances traitées nous permettent de réutiliser efficacement les fonctionnalités de la bibliothèque LinBox. La conception d'une interface de calcul pour ces résolutions qui est facilement paramétrable et réutilisable nous permet de proposer un outil de calcul de haut niveau très performant.

Le premier chapitre de cette thèse présente le projet LinBox et ses objectifs. Nous introduisons en particulier les différents concepts du langage C++ que nous utilisons pour permettre l'interopérabilité de la bibliothèque avec des composants externes. De plus, nous faisons un tour d'horizon des différentes bibliothèques spécialisées et des plates-formes généralistes qu'utilise le projet LinBox. Enfin, nous donnons une description exhaustive des fonctionnalités déjà disponibles dans la bibliothèque LinBox.

Dans le deuxième chapitre, nous proposons d'étudier un concept de branchement à la demande proposé dans LinBox pour fournir une arithmétique de corps finis facilement interchangeable dans les codes et pouvant provenir de bibliothèques externes. Pour cela, nous définissons la notion d'archétype de données qui fixe un modèle de base permettant une définition purement abstraite de l'arithmétique utilisée dans les codes génériques. En particulier, nous montrons que grâce à cet archétype on peut facilement intégrer des codes externes à partir d'adaptateurs

(*wrapper*) qui permettent d'harmoniser les différentes implantations avec le formalisme défini par le modèle de base. La présentation de ces adaptateurs est l'occasion de faire un état de l'art des différentes implantations de corps finis actuellement disponibles et d'en évaluer les performances.

Le chapitre 3 développe l'approche de calcul hybride "exact/numérique" que nous utilisons pour intégrer les routines de multiplication de matrices numériques BLAS à l'intérieur de calculs d'algèbre linéaire sur les corps finis. En particulier, nous développons les différents algorithmes employés pour obtenir une réduction au produit de matrices. Grâce à ces algorithmes et à l'utilisation des routines numériques BLAS, nous proposons des implantations pour les problèmes classiques de l'algèbre linéaire sur un corps fini qui ont des performances très proches de celles obtenues pour le produit de matrice ; à savoir, le calcul de certaines triangularisations de matrices, le calcul du déterminant, le calcul du rang, la résolution de systèmes triangulaires matriciels et le calcul d'une base du noyau. En plus de ce travail général, nous avons eu l'occasion de travailler sur le problème particulier du calcul d'une base du noyau qui nous a permis de proposer une adaptation de l'algorithme de Gauss-Jordan par blocs améliorant le nombre d'opérations nécessaires par rapport au produit de matrices d'un facteur 1/6. La dernière partie de ce chapitre est consacrée à la définition d'une interface standard pour l'utilisation de ces approches hybrides "exact/numérique" ; en outre, nous montrons comment réutiliser ces approches à plus haut niveau, dans le logiciel MAPLE et la bibliothèque LinBox.

Enfin, la chapitre 4 concerne la validation des différentes briques de base développées dans la bibliothèque LinBox autour du problème de la résolution de systèmes linéaires diophantiens ; nous démontrons alors l'intérêt de la bibliothèque. Ce travail s'appuie sur les développements p-adiques des solutions rationnelles pour fournir une interface de résolution générique qui est facilement paramétrable en fonction des caractéristiques du système et du type de solution que l'on souhaite calculer. Nous montrons en particulier que la réutilisation des calculs "exacts/numériques" nous permet de proposer l'une des meilleures implantations actuelles pour les systèmes linéaires diophantiens denses.

Chapitre 1

Organisation et motivation du projet LinBox

Sommaire

1.1 Choix du langage C++ 9
1.1.1 Classes : abstraction, accès et hiérarchisation 10
1.1.2 Polymorphisme et généricité 13
1.1.3 Spécialisations et caractérisations 16
1.2 Utilisation de bibliothèques spécialisées 18
1.2.1 GMP 18
1.2.2 Givaro 19
1.2.3 NTL 19
1.2.4 LiDIA 20
1.2.5 BLAS 20
1.3 Interfaces utilisateurs 21
1.3.1 Maple 21
1.3.2 GAP 22
1.3.3 Serveurs web 23
1.4 Organisation des codes 23

Le projet LinBox est un projet international (Canada, France, USA) http://www.linalg.org autour du calcul exact et plus particulièrement de l'algèbre linéaire exacte. Le but de ce projet est le développement d'un outil logiciel générique pour les problèmes classiques de l'algèbre linéaire tels que la résolution de systèmes linéaires, le calcul du déterminant, le calcul du rang et le calcul des formes normales de matrices. L'ambition du projet s'insère dans une tendance récente de développement logiciel qui vise à fédérer des ensembles de codes spécifiques au sein d'un environnement de calcul commun. L'idée est de proposer un logiciel évolutif et configurable en autorisant le branchement à la demande de composants externes spécifiques. Cette approche est relativement nouvelle dans le calcul scientifique et trouve son intérêt dans l'existence d'outils de calcul symbolique déjà très performants et d'amélioration croissante des techniques algorithmiques du domaine. Cette approche commence à émerger de plus en plus dans le développement de logiciels scientifiques. On peut citer par exemple la bibliothèque SYNAPS [68] pour le calcul symbolique-numérique et la bibliothèque MTL (*Matrix Template Library*) [74] pour l'algèbre linéaire numérique.

La mise en place d'un tel logiciel doit permettre de bénéficier des performances de logiciels très spécialisés comme par exemple la bibliothèque GMP [41] pour l'arithmétique multiprécision ou la bibliothèque NTL [73] pour l'arithmétique des polynômes. À un plus haut niveau, ce logiciel doit aussi permettre de servir de *middleware* pour une mise en place de calculs efficaces dans des environnements conviviaux comme par exemple le logiciel Maple [58] ou GAP [81] ainsi que certains serveurs web.

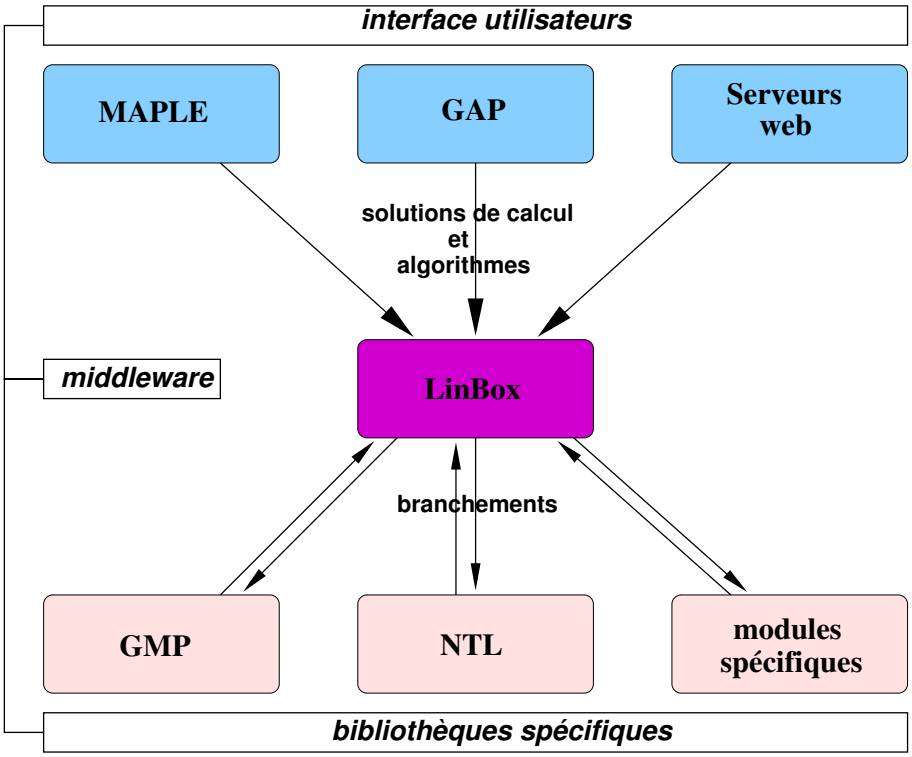

Fig. 1.1 – Physionomie du projet LinBox

La figure 1.1 illustre la philosophie du projet LinBox. Afin de développer ce logiciel, le pro-

jet LinBox s'est orienté vers la conception d'une bibliothèque C++ générique appelée LinBox. L'intérêt d'un langage orienté objet provient du fait que le calcul scientifique, et particulièrement l'algèbre linéaire, s'illustre parfaitement au travers des concepts définis par le modèle objet.

L'objectif de ce chapitre est de présenter les différents niveaux d'organisation de la bibliothèque LinBox illustrés par la figure 1.1. Avant de définir chacun de ces niveaux, nous présentons dans la partie 1.1 quelques concepts du langage C++ que nous utilisons pour définir la structure de *middleware* de la bibliothèque LinBox. Ensuite, dans la partie 1.2, nous décrivons les différentes bibliothèques de calcul spécialisées sur lesquelles nous nous appuyons pour effectuer des calculs spécifiques performants. Nous présentons dans la partie 1.3 quelques applications existantes permettant d'utiliser la bibliothèque LinBox. Enfin, nous terminerons ce chapitre en dressant le bilan de l'organisation des codes à l'intérieur de la bibliothèque LinBox ainsi que les différentes implantations algorithmiques disponibles.

1.1 Choix du langage C++

Depuis sa création dans les années 80, C++ est devenu un langage très utilisé dans le développement d'applications logicielles performantes. Au départ, C++ n'est qu'une simple surcouche orientée objet pour le langage C. A cette époque, ce langage s'appelait encore "*C with classes*". Après plusieurs changements de nom et une évolution des fonctionnalités, le langage C++ émerge enfin en 1984. Bjarne Stroustrup, le concepteur de ce langage [80], explique le choix de C comme langage de base pour définir C++ du fait que C était massivement utilisé et qu'il permettait la mise en œuvre de tout type de programme. Grâce à sa sémantique bas niveau ce langage était d'ailleurs l'un des plus performants à l'époque. Il figure toujours parmi les langages de programmation les plus performants à l'heure actuelle. Encore aujourd'hui, certain compilateurs C++ continuent de générer leurs codes résultats au travers de compilation C. L'apport majeur de C++ par rapport au langage C est l'organisation des programmes à l'aide de classes. Depuis 1991 et l'ajout de modèles paramétrables (i.e. *template*), le langage C++ suscite un intérêt particulier pour le développement d'applications génériques performantes.

Le choix du langage C++ pour développer la bibliothèque LinBox se justifie à la fois par les bonnes performances des programmes compilés et par les facilités de développement apportées par les mécanismes de généricité et de conception objet. De plus, ce langage autorise une bonne portabilité du fait qu'il existe des compilateurs C++ pour la plupart des architectures matérielles. L'utilisation d'autres langages orientés objet, comme Java par exemple, semble ne pas être encore assez compétitive au niveau des calculs pour s'intéresser au développement d'applications en calcul formel [5]. L'autre point important de notre choix pour C++ est la réutilisation de bibliothèques spécialisées. D'une part, ce langage autorise l'intégration d'autres langages comme Fortran qui est le langage de programmation utilisé généralement par les bibliothèques numériques. D'autre part, la majeure partie des bibliothèques faisant référence en calcul symbolique ont été développées en C ou en C++. La bibliothèque GMP [41], qui est déjà considérée comme un standard pour l'arithmétique multiprécision, est développée en C et propose même une interface C++ depuis sa version 4. De même, la bibliothèque NTL [73], qui propose les meilleures implantations d'arithmétique polynomiale, est aussi développée en C++. L'interaction entre ces bibliothèques et LinBox est donc grandement facilitée par l'utilisation d'un langage de programmation commun.

La notion de classe permet de regrouper les structures de données et les différentes fonctions afférentes à un objet à l'intérieur d'un unique modèle de données. Ce modèle sert d'interface aussi bien pour la définition que pour la manipulation des objets. Ainsi, la structure de données sous-jacentes à un objet est totalement décorrélée de l'objet. Seules les fonctions de manipulation de l'objet permettent d'accéder à sa structure. Ce n'est pas le cas dans un langage fonctionnel classique comme C. Nous présentons dans les paragraphes suivants les différents mécanismes de C++ permettant de définir et d'organiser les objets au travers de classes. Plus précisément, nous aborderons les différents concepts de programmation générique disponibles en C++ en présentant leur intérêt pour le développement de la bibliothèque LinBox.

1.1.1 Classes : abstraction, accès et hiérarchisation

Le principal but des classes est de pouvoir spécifier un concept ou un objet au travers d'une définition structurelle et fonctionnelle. La classe définie alors un modèle de données qui encapsule la structure de données de l'objet et les fonctionnalités qui lui sont afférentes. L'intérêt d'utiliser les classes est que l'on peut abstraire la structure d'un objet de sa manipulation. Par exemple, la manipulation d'un vecteur ne nécessite pas de connaître sa structure de données. Il suffit simplement de pouvoir accéder à ces éléments et à sa dimension. Le but de la conception d'une classe consiste à définir l'ensemble des données représentant l'objet (appelées attributs) et l'ensemble des fonctions permettant d'utiliser l'objet (appelées méthodes).

Le code 1.1 décrit un modèle de classe C++ permettant de définir un vecteur d'entiers. Les attributs de cet objet sont ses coefficients et sa dimension et les méthodes sont l'accès aux éléments, l'accès à la dimension et la lecture de coefficients. La création d'un vecteur est ici basée sur un constructeur en fonction de la dimension.

Code 1.1 – Exemple de classe

```
class VecteurInt
{
  // private data
  private:
     int    *_Rep;            // storage of vector
     int    _size;            // size of vector

     void read(int n)         // unsafe read function
     { for (int i=0;i<n;++i)
       cin>>_Rep[i];
     }

  // public data
  public:
     VecteurInt(int n): _size(n) { _Rep = new int[n];} // vector constructor

     ~VecteurInt() {delete[] _Rep;}                    // vector destructor

     int getCoeff(int i) {return _Rep[i];}             // elements accessor

     int size() {return _size;}                        // size accessor

     void readCoeff() {read(_size);}                   // safe read function
     ...
};
```

Une notion importante dans la mise en place de classes C++ est le *contrôle d'accès*. Ce mécanisme permet de protéger les attributs et les méthodes d'un objet des erreurs de manipulation. Par exemple, notre code 1.1 définit les attributs `_Rep`, `_size` et la méthode `read` de façon privée afin d'empêcher leur accès depuis une instance de l'objet `VecteurInt`. En effet, un accès public à ces données permettrait à un utilisateur de modifier l'objet de façon non maîtrisée. Toujours dans notre exemple, la fonction `read` permet la lecture d'un nombre de coefficients potentiellement supérieur à la zone mémoire allouée au vecteur. Voila pourquoi cette fonction doit être définie de façon privée. En contrepartie, il faut fournir des méthodes publiques garantissant qu'il n'y aura aucun effet de bord sur l'objet. La fonction `readCoeff` de notre modèle est définie de façon publique car elle permet d'éviter les effets de bord de la fonction `read`. Il existe trois niveaux de contrôle d'accès en C++ :

- accès privé (`private`)
- accès protégé (`protected`)
- accès public (`public`)

L'accès privé est le plus restrictif et n'autorise que l'accès à l'intérieur du modèle (par exemple `read` dans le code 1.1). L'accès public autorise l'accès à tous les niveaux. Enfin, l'accès protégé propose le même niveau de restriction que l'accès privé sauf pour les modèles et les fonctions qui sont déclarés explicitement comme amis (`friend`) dans la déclaration du modèle de l'objet.

Une autre caractéristique importante du C++ est la *hiérarchisation de classes*. Cette hiérarchisation s'exprime grâce au concept d'héritage, qui permet de modéliser l'inclusion des ensembles des objets. L'utilisation de cette notion permet donc d'organiser le code de façon plus intuitive en spécialisant les objets à partir d'un graphe de relation.

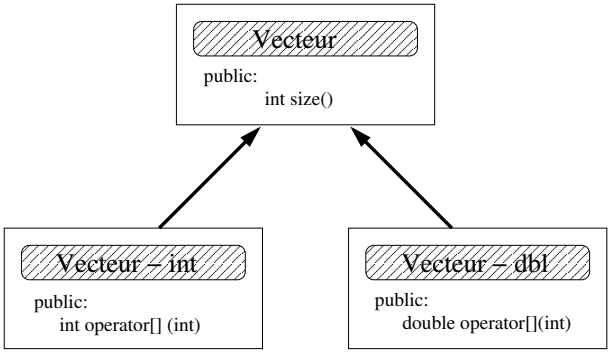

FIG. 1.2 – Exemple d'héritage : modèle de vecteur à deux niveaux

Les caractéristiques communes de plusieurs objets peuvent ainsi être définies dans un même modèle de base. La figure 1.2 illustre cette notion de modèle de base pour des vecteurs dont les coefficients ne sont pas définis pour le même type de données. La notion de dimension étant une caractéristique commune à nos deux modèles de vecteurs, on peut donc modéliser cette caractéristique à l'aide d'une classe de base `vecteur`.

Le contrôle d'accès aux classes de base joue ici un rôle très important. En effet, ce type de contrôle se propage dans les relations d'héritage. Le mécanisme d'héritage s'accompagne lui aussi d'une gestion de contrôle d'accès qui permet de modifier le contrôle d'accès des données

d'une classe de base pour ses classes dérivées.

		Héritage		
		`private`	`protected`	`public`
Contrôle d'accés	`private`	*inaccessible*	*inaccessible*	*inaccessible*
	`protected`	*private*	*protected*	*protected*
	`public`	*private*	*protected*	*public*

TAB. 1.1 – Contrôle d'accès des classes de base via l'héritage

À l'instar du contrôle d'accès des caractéristiques d'une classe (attributs+méthodes), il existe trois types d'héritage possibles : `private`, `protected`, `public`. La combinaison du contrôle d'héritage et du contrôle d'accès de la classe de base permet de générer un contrôle d'accès dans les classes dérivées. Par exemple, une caractéristique déclarée privée (`private`) dans une classe de base sera inaccessible dans une classe dérivée quel que soit le type d'héritage. De même, l'héritage privé rend toutes les caractéristiques héritées inaccessibles dans les classes dérivées. Le choix du type de contrôle d'accès et du type d'héritage est donc un facteur prépondérant pour la mise en place d'une hiérarchisation de classe cohérente. Nous résumons dans le tableau 1.1 les types de contrôles obtenus en fonction du type d'héritage et du type de contrôle d'accès initial. Le code 1.2 illustre la notion d'héritage à partir du schéma de dépendance établi dans la figure 1.2. La syntaxe du C++ pour définir une relation d'héritage entre une classe A et une classe B est :
`class A : public B  ...;` où `public` précise le type d'héritage.

Code 1.2 – Exemple d'héritage

```
class Vecteur{
  private:
     int _size;
  public:
     int size() {return _size;}
};

class VecteurInt : public Vecteur {
  private:
     int *_Rep;
  public:
     VecteurInt (int n) : Vecteur(n) { _Rep = new int[n];}
     ~VecteurInt() {delete[] _Rep;}
     int getCoeff(int i) {return _Rep[i];}
     ...
};

class VecteurDbl : public Vecteur {
  private:
     double *_Rep;
  public:
     VecteurDbl (int n) : Vecteur(n) { _Rep = new double[n];}
     ~VecteurDbl() {delete[] _Rep;}
     int getCoeff(int i) {return _Rep[i]};
     ...
```

```
};
```

1.1.2 Polymorphisme et généricité

La définition d'objets à partir de classes permet une description haut niveau facilitant l'écriture de programmes. La notion d'héritage permet d'organiser les codes par des relations d'inclusion comme nous avons vu dans le paragraphe précédent. Un autre aspect intéressant du C++ est la définition de modèles de données génériques. On parle de polymorphisme pour définir cette notion. On distingue deux types de polymorphisme en C++ : le polymorphisme statique et le polymorphisme dynamique. Ces deux polymorphismes diffèrent par leur gestion des paramètres génériques. Le polymorphisme statique fixe ces paramètres lors de la compilation (*compile time*) alors que le polymorphisme dynamique gère ces paramètres lors de l'exécution (*run-time*).

La notion de *polymorphisme statique* s'exprime en C++ par l'utilisation de la commande `template` [80, chap. 13]. Cette commande permet de spécifier qu'un ou plusieurs types de données ne sont pas fixés dans le modèle. On dit alors que le modèle est générique en fonction des types de données non spécifiés. Ces types sont spécifiés lors de l'instanciation d'un objet du modèle. En particulier, il faut que l'ensemble des paramètres génériques soit connu au moment de la compilation afin que des spécialisations du modèle soient utilisées. En reprenant notre exemple de classe vecteur du code 1.2, nous définissons un modèle de vecteur générique en fonction de la nature des coefficients. Le code 1.3 illustre cette modélisation.

Code 1.3 – Exemple de polymorphisme statique

```
template <class Coeff>
class Vecteur {
  private:
     Coeff *_Rep;
     int   _size;
  public:
      Vecteur(int n): _size(n) { _Rep= new Coeff[n];}
      ~Vecteur() {delete[] _Rep;}
      Coeff getCoeff(int i) {return _Rep[i]};
      int size() {return _size;}
      ...
};
```

Le principe des modèles génériques est de fournir une conception type pour un ensemble d'objets. Le code 1.3 nous permet par exemple de remplacer les classes `VecteurInt` et `VecteurDbl` données dans le code 1.2 par un seul modèle de classe. L'utilisation des classes `Vecteur<int>` et `Vecteur<double>` permet d'appréhender exactement le même type d'objets. C'est le compilateur qui se charge de dupliquer et de spécialiser le modèle générique sur les différents types d'instanciations.

En utilisant ce mécanisme, on peut donc facilement décrire des objets concrets à partir de modèles de données abstraits. La seule obligation est que le type d'instanciation du modèle générique respecte les prérequis du modèle. Typiquement, il faut que ces types proposent l'ensemble

des opérations utilisables dans le modèle.

La bibliothèque STL (*Standard Template Library*) [62, 35] utilise cette technique afin de fournir une bibliothèque générique pour la manipulation de vecteurs, de listes, de tables de hachage et différentes structures de *conteneurs* [10, page 188] de données. On peut ainsi utiliser cette bibliothèque pour définir des vecteurs pour n'importe quel type de coefficients (ex. : `std::vector<int>` et `std::vector<double>`). L'intérêt de la bibliothèque STL est qu'elle permet de manipuler n'importe quel de ces conteneurs à l'aide d'une interface générique. Cette interface est définie par la notion d'*itérateur de données* [10, page 193]. Chaque conteneur propose une structure d'itérateur permettant de parcourir l'ensemble des objets qu'il représente. Le nom des itérateurs étant standard (`iterator, const_iterator,...`) on peut alors les utiliser pour définir des codes génériques sur les conteneurs. On peut par exemple définir la fonction générique suivante qui affiche l'ensemble des éléments d'un conteneur.

```
  template <class container>
  print(const container &c){
    typename container::const_iterator it;
    for (it=c.begin(); it!=c.end(); ++it)
       cout<<*it<<endl
}
```

Chaque conteneur doit fournir des fonctions de création d'itérateur en début et en fin de structure, les fonctions `begin()` et `end()` dans notre exemple. L'opérateur `++` permet de déplacer l'itérateur sur les données du conteneur alors que l'opérateur `*` permet de récupérer l'élément pointé par l'itérateur. Cette approche par itérateur est standard en C++ pour des structures de données ayant un stockage linéaire. Nous réutilisons cette approche dans la bibliothèque LinBox pour la plupart des structures de données linéaires (vecteur, matrice, générateur aléatoire, développement p-adique, ...). Le but est de pouvoir fournir des implantations génériques pour l'ensemble des structures de données qui proposent les bons itérateurs. Nous verrons dans la partie 4.2 de ce document un exemple d'utilisation de cette approche pour la résolution de systèmes linéaires entiers.

Bien que le mécanisme de polymorphisme statique soit intéressant pour définir des modèles génériques, son utilisation entraîne la génération et la compilation de code de taille conséquente. En effet, dès lors qu'un modèle générique est instancié sur un nouveau type de données, le compilateur génère un modèle spécifique associé à ce type de données. La compilation de codes uniquement définis à partir de codes *templates* est donc susceptible de saturer la taille des codes compilés ou des exécutables générés. On parle alors d'explosion de code. Une alternative en C++ pour définir des fonctions génériques sans utiliser les mécanismes *templates* consiste à utiliser des *classes abstraites*. Le but d'une classe abstraite est de servir d'unique interface pour la manipulation d'objets définissant un concept commun. Par exemple, l'arithmétique définit un concept commun à l'ensemble des nombres. L'idée est de proposer une interface pour modéliser ce concept de façon abstraite et de pouvoir l'utiliser pour manipuler des implantations d'arithmétiques concrètes. Le principe des classes abstraites est de pouvoir déterminer lors de l'exécution quelle est l'implantation concrète à utiliser en fonction du type dynamique de l'objet. On parle alors de *polymorphisme dynamique*.

Ce polymorphisme dynamique est possible en pratique grâce à l'utilisation des *fonctions virtuelles* [10, page 101]. Les fonctions virtuelles sont un moyen de spécifier au compilateur que la fonction est résolue en fonction du type dynamique de l'objet et non pas en fonction de son type statique. Nous illustrons le principe des fonctions virtuelles avec l'exemple suivant.

```
class A {
  public:
     virtual void f() {cout<<"object A";}
};

class B : public A {
  public:
     virtual void f() {cout<<"object B";}
};

int main() {
  A a;
  B b;
  A& c=b;
  a.f();     // print object A
  b.f();     // print object B
  c.f();     // print object B
  return 0;
}
```

Cet exemple définit deux classes A et B dont l'une est dérivée de l'autre (B est dérivée de A). Ces deux classes encapsulent une fonction virtuelle `f()` qui affiche le type dynamique de l'objet. La syntaxe C++ pour définir une fonction virtuelle est `virtual`. Cet exemple illustre le fait que, même si le type statique de l'objet `c` est la classe `A`, la fonction `f()` dépend du type dynamique, ici la classe `B`.

En pratique, ce mécanisme est défini grâce à des tables de pointeurs encapsulées dans la structure de l'objet. On comprend mieux ce mécanisme en regardant à quoi ressemblent les objets en mémoire. La figure 1.3 illustre les objets `a, b, c` de notre exemple.

La classe A définit la fonction virtuelle `f()` à l'intérieur d'une table virtuelle. Cette table est associée à la structure de données des objets de type `A`. Un objet de la classe B étant aussi un objet de type A, il hérite dans sa structure de la table virtuelle de la classe A. En redéfinissant la fonction `f()` dans la classe B, on redéfinit en même temps la table virtuelle de la classe A associée aux objets B. Le transtypage d'un objet de type B vers un objet de type A permet de conserver la table virtuelle associée. Cela signifie que l'on peut manipuler des objets B à partir d'objets A. Lorsque la classe de base est définie de façon totalement abstraite (fonctions virtuelles pures), on ne peut créer d'instance du type de base. La classe de base est juste une interface (à partir des fonctions virtuelles).

Cette technique de classe abstraite est réutilisée dans LinBox pour définir des archétypes de données. Ce type d'utilisation est dû à Erich Kaltofen et s'applique particulièrement au concept de matrices boîtes noires. Ces matrices sont considérées uniquement comme des opérateurs linéaires où seule l'application de la matrice à un vecteur est disponible. Toutefois, l'utilisation des classes abstraites ne permet pas de fournir d'instance d'objet compilé. En effet, ces classes

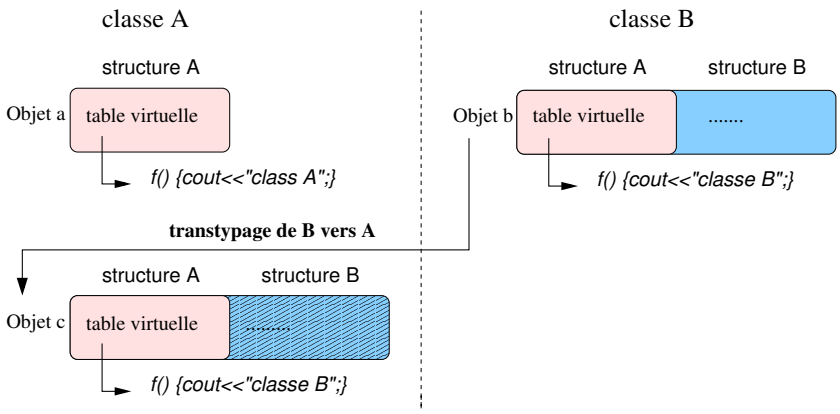

FIG. 1.3 – Mécanisme des fonctions virtuelles

ne peuvent être instanciées qu'à partir de références ou de pointeurs d'un objet dérivé. Nous nous intéresserons dans la section 2.1.2 à définir l'approche d'archétype compilable initiée par Erich Kaltofen dans le projet LinBox et qui consiste à utiliser des classes abstraites pour définir une interface de données compilable.

1.1.3 Spécialisations et caractérisations

Le développement d'archétypes de données et de fonctions génériques permet de définir des implantations génériques facilement paramétrables. Pour cela, il suffit de proposer des modèles de données qui soient compatibles avec les archétypes. L'autre étape importante dans la mise en place de la bibliothèque LinBox est l'optimisation des codes en fonction des implantations d'algorithmes disponibles. Le mécanisme d'archétype encapsule déjà un niveau d'optimisation grâce à la surcharge des fonctions virtuelles dans les classes dérivées. Cela permet pour chaque classe dérivée de proposer sa propre optimisation. Le même type de surcharge est possible dans les fonctions *templates* à partir de *spécialisation du modèle*. Lors de l'instanciation d'un modèle de fonction *template* sur un nouveau type de données, le compilateur duplique le code et le spécifie (voir section 1.1.2). La spécialisation explicite d'un modèle de fonction *template* permet de remplacer cette duplication de code par le chargement d'un modèle spécialisé. Cependant, ce type de spécialisation se base uniquement sur le type des objets utilisés. Dans certains cas, il est intéressant de pouvoir spécialiser ces fonctions à partir d'un caractère commun à plusieurs types de données sans avoir à redéfinir une nouvelle interface. Par exemple, on veut pouvoir distinguer les matrices denses des matrices creuses sans avoir à redéfinir une interface pour chacune d'elle. Une solution proposée par le C++ est l'utilisation des *caractérisations de type*.

Le principe de ces caractérisations (*traits* en anglais) est d'encapsuler une ou plusieurs caractéristiques à l'intérieur des modèles de données. Ces caractéristiques sont définies à partir d'un type de données (structure ou classe). Ensuite, on utilise ce type de données pour spécialiser une fonction générique en fonction du caractère associé. Par exemple, on peut définir les types de données `struct Dense ;` `struct Diagonal ;` pour caractériser la structure d'une matrice. Grâce à cette caractérisation, on peut spécialiser l'implantation de la résolution de systèmes linéaires uniquement en fonction du caractère de la matrice et non en fonction de son type de données. Cela permet d'éviter de spécialiser la fonction de résolution pour tous les types de matrices denses ou diagonales.

Nous utilisons ce type de caractérisation dans la bibliothèque LinBox afin de pouvoir utiliser différentes techniques algorithmiques au sein d'une même interface de résolution. On peut alors changer d'algorithme directement en spécifiant le "traits" associé à la méthode. Par exemple, on utilisera des résolutions de systèmes linéaires différentes selon que la matrice est dense ou diagonale. Afin de spécifier le type d'implantation à utiliser, on s'appuie sur une fonction générique qui pour un type de données retourne ses caractéristiques associées. Nous illustrons cette technique avec le code suivant, qui propose une interface de résolution de système spécialisée en fonction de la structure des matrices.

Code 1.4 – Exemple d'utilisation des *traits*

```
// Matrix category
struct MatrixCategory{
  struct Dense{};
  struct Diagonal{};
};

// classes of matrices
class DenseMatrix1    { typedef MatrixCategory Dense;};
class DenseMatrix2    { typedef MatrixCategory Dense;};
class DiagonalMatrix1 { typedef MatrixCategory Diagonal;};
class DiagonalMatrix2 { typedef MatrixCategory Diagonal;};

// accessor of matrix traits
template <class Matrix>
class Matrix_Traits {
  typedef typename Matrix::MatrixCategory MatrixCategory;
};

// interface of system solving
template <class Matrix, class Vector>
Vector  LinearSolve(const Matrix& A, const Vector& b) {
  return LinearSolveSpecialized (A,b, Matrix_Traits<Matrix>::MatrixCategory());
}

// specialization of system solving for Dense traits
template <class Matrix, class Vector>
Vector LinearSolveSpecialized(const Matrix& A,
                              const Vector& b,
                              MatrixCategory::Dense trait);

// specialization of system solving for Diagonal traits
template <class Matrix, class Vector>
Vector LinearSolveSpecialized(const Matrix& A,
                              const Vector& b,
                              MatrixCategory::Diagonal trait);
```

La fonction `LinearSolve` définit l'interface générique pour la résolution de systèmes linéaires. Les implantations de résolution de systèmes sont définies dans les différentes spécialisations de la fonction `LinearSolveSpecialized`. Si tous les types de matrices possèdent une caractérisation de type `MatrixCategory` alors le choix de la spécialisation dans l'interface se fera automatiquement. Dans notre exemple (code 1.4), on utilise les deux caractéristiques (Dense et Diagonal)

pour les types de matrices (Dense1, Dense2 , Diagonal1, Diagonal2). Suivant le caractère associé au type de la matrice, l'interface choisira l'implantation adéquate.

1.2 Utilisation de bibliothèques spécialisées

Dans cette partie, nous présentons les différentes bibliothèques spécialisées sur lesquelles nous nous appuyons pour développer la bibliothèque LinBox et fournir des implantations performantes. L'ensemble de ces bibliothèques est une liste représentative de ce qu'il existe à l'heure actuelle. Nous présentons ici les grandes lignes de ces bibliothèques en précisant leurs domaines d'application ainsi que nos motivations pour leur utilisation dans la bibliothèque LinBox.

1.2.1 GMP

La bibliothèque GMP[5] (*GNU Multiprecision Package*) [41] est une bibliothèque spécialisée pour l'arithmétique des nombres en multiprécision. Elle propose en particulier des implantations pour les nombres entiers signés, les nombres flottants et les nombres rationnels. L'efficacité de la bibliothèque repose à la fois sur des codes très performants (mots machines, pipelines logiciels, codes assembleurs) et sur l'utilisation de méthodes algorithmiques performantes (coût théorique *vs* coût pratique). Bien que certaines parties critiques pour les performances soient écrites en assembleur, la majorité de la bibliothèque est écrite en C. La portabilité de la bibliothèque est assurée grâce au développement de codes assembleur pour la plupart des architectures matérielles disponibles (x86, Itanium, UltraSparc, AMD,...). L'utilisation combinée de couches assembleur, de codes C optimisés et d'algorithmes adaptatifs en fonction de la taille des opérandes permet à GMP d'être la bibliothèque multiprécision la plus efficace à l'heure actuelle. De plus, cette bibliothèque est distribuée sous licence LGPL (*Lesser General Public Licence*), ce qui nous permet de réutiliser les codes facilement.

Nous utilisons essentiellement GMP dans la bibliothèque LinBox pour définir l'arithmétique d'entiers multiprécision qui est la base pour le calcul exact. L'un de nos intérêts d'utiliser la bibliothèque GMP provient du fait que les opérations critiques en algèbre linéaire sont les multiplications d'entiers. L'implantation de cette opération a donc un impact important sur les performances des codes de la bibliothèque LinBox. L'intérêt de la bibliothèque GMP est qu'elle propose une implantation de multiplication d'entiers multiprécisions extrêmement efficace. Pas moins de quatre algorithmes sont utilisés pour assurer les meilleures performances à tous les niveaux de précisions. En considérant que la taille des opérandes est de n mots machine, nous précisons le nombre d'opérations sur les mots machine des algorithmes utilisés en fonction de n. Nous utilisons la notation log pour exprimer le logarithme en base deux.

- $n < 10$: algorithme classique, $O(n^2)$ opérations ;
- $10 \leq n < 300$: algorithme de Karatsuba [52], $O(n^{\log 3})$ opérations ;
- $300 \leq n < 1000$: algorithme de Toom-Cook [83, 16], $O(n^{\frac{\log 5}{\log 3}})$ opérations ;
- $n \geq 1000$: algorithme de Schönhage-Strassen [72], $O(n \log n \log \log n)$ opérations.

L'intégration des entiers multiprécisions GMP au sein de la bibliothèque LinBox est faite en utilisant une classe C++ représentant le type `LinBox::integer`. Le développement de cette interface permet d'appréhender les entiers multiprécisions de GMP comme un type natif dans

[5] http://www.swox.com/gmp/

la bibliothèque. Une interface similaire est seulement disponible dans GMP depuis 2002 et la version 4.0 de la bibliothèque.

1.2.2 Givaro

La bibliothèque Givaro[6] [37] est une bibliothèque C++ pour le calcul formel. Elle propose en particulier des implantations efficaces pour les corps finis ayant une taille inférieure au mot machine ainsi que des structures de données pour la manipulation de matrices, de vecteurs, de polynômes et de nombres algébriques. Comme pour la bibliothèque LinBox, Givaro utilise GMP pour définir un type natif d'entiers multiprécision. L'interface utilisée est d'ailleurs commune avec celle la bibliothèque LinBox. À partir de ces entiers multiprécision, la bibliothèque Givaro propose un ensemble d'algorithmes pour la cryptographie tels que la factorisation d'entiers, le test de primalité ainsi qu'une implantation de l'algorithme RSA [69].

Notre intérêt pour la bibliothèque Givaro provient des implantations de corps finis qu'elle propose. En particulier, les implantations de corps premiers de type $\mathbb{Z}/p\mathbb{Z}$ sur des mots machine sont très performantes (voir §2.2.6). L'implantation d'algorithmes efficaces pour la recherche de racines primitives creuses d'une extension algébrique [25, §3.3] permet à la bibliothèque Givaro de proposer une représentation logarithmique des corps de Galois qui permet une arithmétique particulièrement performante comme l'a montré J.-G. Dumas dans sa thèse [25, §4.2]. L'intégration des codes Givaro au sein de la bibliothèque LinBox est directe du fait de nos travaux de synchronisation entre ces deux bibliothèques.

1.2.3 NTL

La bibliothèque NTL[7] (*Number Theory Library*) développée par V. Shoup est une bibliothèque C++ dédiée au calcul avec des polynômes, des vecteurs et des matrices à coefficients entiers ou dans un corps fini. L'arithmétique des entiers utilise la bibliothèque GMP pour obtenir les meilleures performances même si une implantation directe est proposée pour conserver une certaine indépendance. Un des points forts de la bibliothèque NTL est l'implantation d'un nombre important d'algorithmes classiques du calcul formel. Plus particulièrement, on trouve des algorithmes pour les polynômes (arithmétique, factorisation, test d'irréductibilité,...), pour les réseaux d'entiers et l'algèbre linéaire exacte sur des matrices denses.

L'intérêt de la bibliothèque NTL est double pour nous. D'une part, elle nous permet de comparer les performances de nos implantations d'algorithmes dans la bibliothèque LinBox. D'autre part, elle nous permet de fournir des implantations de corps finis et d'extensions algébriques performantes pour de grandes cardinalités. L'ensemble des implantations disponibles ont des interfaces standardisées qui permettent une utilisation simplifiée. Bien que la bibliothèque soit développée en C++, l'intégration des codes et des modèles de données dans LinBox n'est pas immédiate. Nous utilisons donc des *wrappers* permettant de synchroniser les codes NTL avec la bibliothèque LinBox. Le surcoût engendré par cette synchronisation est raisonnable et nous permet de bénéficier pleinement des performances de la bibliothèque NTL. Nous aborderons plus en détail la réutilisation des corps finis au chapitre 2.

[6] http://www-lmc.imag.fr/Logiciels/givaro/
[7] http://www.shoup.net/ntl/

1.2.4 LiDIA

La bibliothèque LiDIA[8] est une bibliothèque C++ spécialisée dans la théorie des nombres. Comme la bibliothèque NTL, elle propose des structures de données pour les corps finis, les matrices, les vecteurs et les polynômes. Des modules pour les corps de nombres et les courbes elliptiques sont aussi disponibles. Le développement de la bibliothèque LiDIA est similaire à celui de la bibliothèque LinBox dans le sens où le noyau de la bibliothèque est reconfigurable en fonction de modules externes. Ce noyau comprend l'arithmétique des entiers multiprécision et le ramasse-miettes (*garbage collector*). L'utilisation de ce noyau dans la bibliothèque se fait uniquement à partir des interfaces correspondantes : la classe `bigint` pour les entiers et la classe `gmm` pour le ramasse-miettes. Plusieurs intégrations sont disponibles, en particulier avec les bibliothèques GMP, cln[9], piologie[10] pour les entiers multiprécision et la bibliothèque Boehm-gc[11] pour la gestion mémoire. Cependant, la reconfiguration du noyau ne peut se faire que de façon statique lors de la compilation de la bibliothèque.

Notre intérêt pour la bibliothèque LiDIA provient essentiellement des implantations de corps finis. En particulier, elle propose une interface unique pour la manipulation des corps finis quelque soit le type de corps (corps premier, extension, tour d'extension). De ce fait, les optimisations sur les corps finis sont disponibles de façon adaptatives de telle sorte que pour une caractéristique et une cardinalité données, on utilise la meilleure implantation disponible. L'intégration dans LinBox des différentes implantations de corps finis de LiDIA est donc simplifiée grâce à l'utilisation de l'interface. Néanmoins, nous verrons que l'utilisation de cette interface pénalise fortement LiDIA par rapport aux performances obtenues avec la bibliothèque NTL (voir §2.3.4).

1.2.5 BLAS

BLAS (*Basic Linear Algebra Subroutines*) [24] est une collection de routines C et Fortran pour les opérations de base en algèbre linéaire numérique. Plus précisément, on parle de BLAS 1 pour les opérations scalaire-vecteur, BLAS 2 pour les opérations matrice-vecteur et BLAS 3 pour les opérations matrice-matrice. Ces routines sont standardisées au travers du BLAS *Technical Forum*[12] et s'appuient sur la représentation des nombres flottants IEEE-754 [1] en simple et double précision ainsi que sur des précisions étendues [55]. Les BLAS fournissent un ensemble de routines robustes, efficaces et portables permettant de développer facilement des algorithmes de plus haut niveau performants. La bibliothèque LAPACK [13] est l'un des meilleurs exemples d'utilisation des BLAS. Les performances des BLAS sont obtenues grâce à l'utilisation d'optimisation de cache, de hiérarchisation mémoire, de déroulage de boucles, de parallélisme et d'autres techniques permettant de spécialiser l'exécution des calculs sur les processeurs (classiques, superscalaires, parallèles).

Un des grands intérêts des BLAS est l'efficacité des routines de multiplication de matrices (i.e. `dgemm` sur les doubles). Les BLAS permettent en pratique d'obtenir des performances quasi optimales pour les unités flottantes des processeurs classiques. Pour des processeurs comme le Pentium 4, qui possède un parallélisme intrinsèque, il est même possible d'obtenir des performances pratiquement deux fois supérieures à la cadence du processeur. Ces performances sont possibles grâce à l'utilisation de BLAS spécifiques, généralement proposées par les constructeurs matériels.

[8] http://www.informatik.tu-darmstadt.de/TI/LiDIA/
[9] http://www.ginac.de/CLN/index_normal.html
[10] http://www.hipilib.de/piologie.htm
[11] http://www.hpl.hp.com/personal/Hans_Boehm/gc/
[12] http://www.netlib.org/blas/blast-forum/
[13] http://www.netlib.org/lapack/

Néanmoins, depuis 1998, il est possible d'obtenir des performances similaires en utilisant le logiciel d'optimisation automatique ATLAS[14] *Automatically Tuned Linear Algebra Software* [90]. Ce logiciel permet de créer automatiquement des versions des BLAS et de LAPACK optimisées en déterminant les critères optimaux de l'architecture suivant une série de tests.

L'utilisation des BLAS dans la bibliothèque LinBox se justifie par les performances exceptionnelles qu'elles permettent d'atteindre. Néanmoins, les calculs disponibles sont numériques et leur utilisation pour des opérations exactes ne semble pas naturelle. Nous verrons au chapitre 3 comment ces routines sont quand même utilisées par LinBox pour effectuer des calculs exacts tout en conservant les performances des bibliothèques numériques.

1.3 Interfaces utilisateurs

L'interopérabilité entre LinBox et d'autres logiciels de calcul est l'une des motivations du développement de la bibliothèque LinBox. À un plus haut niveau, l'intégration de LinBox dans des environnements plus conviviaux qu'une bibliothèque C++ devrait permettre d'atteindre un plus large spectre d'utilisateurs. En particulier, la disponibilité de calcul LinBox à l'intérieur d'un logiciel comme MAPLE est l'un des objectifs du projet car ce logiciel est un standard pour le calcul scientifique.

Ce type d'interaction entre des logiciels de calcul généralistes et des bibliothèques spécialisées est primordiale pour l'évolution du calcul scientifique. Depuis quelques années, l'association de calculs spécialisés et d'interface généraliste commence à émerger. On peut prendre comme exemple l'intégration de certains codes GMP dans la version 9 du logiciel MAPLE[15]. On peut aussi citer des logiciels comme TexMacs[16] ou Mathemagix[17] qui proposent des environnements de calculs facilitant l'intégration et l'interaction de bibliothèques spécialisées.

Dans cette partie, nous présentons les logiciels dans lesquels il est d'ores et déjà possible d'effectuer des calculs à partir de LinBox. Pour chacun d'eux, nous préciserons l'objectif principal de l'utilisation de la bibliothèque LinBox. Enfin, dans la dernière partie, nous indiquons quels sont les différents accès en ligne à des calculs LinBox à partir d'interfaces web.

1.3.1 MAPLE

MAPLE [58] est l'un des logiciels de calcul formel les plus connus et l'un des plus utilisés aussi bien en recherche que pour l'enseignement. L'intérêt majeur de ce logiciel est qu'il propose un langage interprété non typé permettant une description simple des calculs. Néanmoins, l'exécution de calculs nécessitant de très hautes performances n'est pas satisfaisante du fait de la densité du noyau de calcul et de l'utilisation d'algorithmes moins compétitifs que les algorithmes récents.

Le premier objectif du projet LinBox par rapport à MAPLE était de fournir des implantations spécifiques pour les matrices creuses définies sur un corps premier. En particulier, le but était de proposer des calculs basés sur des méthodes itératives exactes de type Krylov qui sont les plus performantes à l'heure actuelle pour ce type de matrices. Une première interface entre LinBox et MAPLE a été développée par R. Seagraves. L'idée est d'utiliser le format "NAG sparse" de matrices creuses pour communiquer les matrices entre LinBox et MAPLE. Ce format stocke les matrices dans trois vecteurs correspondants aux valeurs non nulles et à leurs indices de lignes et

[14] http://math-atlas.sourceforge.net/
[15] http://www.maplesoft.com
[16] http://www.texmacs.org
[17] http://www.mathemagix.org/mmxweb/web/welcome.en.html

de colonnes. La création d'une matrice LinBox basée sur ce format (i.e. `TriplesBB`) a permis la communication des matrices entre LinBox et MAPLE.

Grâce à ces conversions de données entre MAPLE et LinBox, on peut alors utiliser les fonctionnalités disponibles pour les matrices creuses dans LinBox. Plus précisément, cela concerne le calcul du rang, du déterminant et du polynôme minimal par l'approche de Wiedemann. L'utilisation de cette interface consiste à créer des instances d'objets LinBox à l'intérieur de MAPLE et d'exécuter des calculs LinBox sur ces objets. Par exemple, la création de matrice LinBox à partir de cette interface consiste à utiliser la commande LinBoxBB . Cette commande prend en paramètre :

- un nombre premier définissant le corps fini
- une matrice MAPLE (i.e. Matrix(...))
  ou
- ses dimensions et une fonction de remplissage

On peut par exemple créer deux matrices LinBox dans MAPLE par le code suivant

```
L1:= LinBoxBB(17,Matrix(4,4,[[0,3,0,1],[0,0,0,4],[1,0,1,0],[5,0,0,0]]));
L2:= LinBoxBB(17,4,4,proc(i,j) if (i+j) mod 2 =0 then 1 else 0 fi end);
```

La matrice L1 est définie à partir d'une matrice MAPLE alors que la matrice L2 est définie à partir d'une procédure MAPLE. À partir de ces deux matrices on peut définir des matrices MAPLE à stockage "NAG sparse" en utilisant la fonction getMatrix().

```
M1:= L1:-getMatrix();
M2:= L2:-getMatrix();
```

L'utilisation des algorithmes LinBox sur les matrices `L1` et `L2` est directe.

```
LinBox[LBrank](L2);
LinBox[LBdet](L2);
LinBox[LBminpoly](L2,x);
```

Cette interface est pour l'instant au stade d'expérimentation. Elle ne prend pas encore en compte la reconfiguration dynamique des codes de la bibliothèque LinBox. Le prochain objectif du projet LinBox est de fournir une interface MAPLE permettant d'utiliser le branchement à la demande de composantes externes (domaine de calcul, matrice, algorithme) proposé par la bibliothèque LinBox.

1.3.2 GAP

Le logiciel GAP [81] (*Groups, Algorithm and Programming*) offre des solutions pour des problèmes en algèbre discrète avec un intérêt particulier pour la théorie des groupes. Le noyau de GAP propose un langage de programmation avec des structures de contrôle de type Pascal, un ramasse-miettes, des types pré-construits pour les structures de la géométrie algébrique et un environnement interactif interprété pour son utilisation. Le logiciel propose des algorithmes soit écrits en GAP soit écrits en C disponibles directement ou à partir de paquetages. Il existe pas moins de 21 paquetages disponibles pour la version 3.4.4 du logiciel GAP. GAP propose des arithmétiques pour les entiers multiprécision, les rationnels, les corps cyclotimiques, les corps finis, les anneaux, les nombres p-adiques, les polynômes mais aussi des outils pour la définition de vecteurs, de matrices, de fonctions combinatoires et un ensemble de briques de base pour la théorie des nombres.

Certains calculs en géométrie algébrique et en combinatoire font appel à de l'algèbre linéaire sur des entiers. Par exemple, le calcul d'homologie de complexes simpliciaux peut être effectué à

partir de la forme de Smith de matrices entières. J.G. Dumas, F. Heckenbach, B.D. Saunder et V. Welker ont proposé un paquetage GAP pour le calcul d'homologie de complexes simpliciaux basé sur la bibliothèque GMP 1.2.1 pour l'arithmétique des entiers multiprécision, sur Givaro pour l'arithmétique des corps finis et sur LinBox pour le calcul de rang et de forme normale de matrices [31]. Ce paquetage est disponible à partir de la page du projet LinBox (*GAP homology package*[18]).

1.3.3 Serveurs web

Les calculs en algèbre linéaire exact au travers de la bibliothèque LinBox sont très performants et ont permis de trouver des solutions jusqu'à lors incalculables sur des ordinateurs classiques [25] (dépassement des ressources mémoire). La mise en place de serveurs de calcul directement utilisables à partir d'une interface web permet de résoudre certains problèmes sans avoir à installer la bibliothèque LinBox et à programmer la résolution du problème. Le fonctionnement et l'utilisation de ces serveurs sont très simples. Les entrées du problèmes à résoudre sont stockées dans un fichier disponible à partir d'une `url` (i.e. une matrice) et grâce à des formulaires internet on peut paramétrer les calculs. Le résultat est soit renvoyé directement si le calcul n'est pas trop long soit il est envoyé à une adresse mail configurée par l'utilisateur.

Il existe plusieurs types de serveur offrant des calculs basés sur la bibliothèque LinBox. Le premier propose une utilisation directe de la bibliothèque LinBox. Ce serveur est encore en cours de développement mais propose déjà des solutions pour le calcul du polynôme minimal, du déterminant et du rang pour des matrices creuses à coefficients dans un corps premier.

Le deuxième serveur est spécialisé pour le calcul du rang ou de la forme de Smith de matrices creuses à coefficients entiers. Il est néanmoins possible de paramétrer le calcul pour qu'il soit effectué modulo un nombre premier. Enfin, le dernier serveur est une interface web pour utiliser le module GAP de calcul d'homologie de complexes simpliciaux (voir §1.3.2). Ce serveur permet en particulier d'utiliser le logiciel GAP sans avoir à l'installer localement.

Le premier de ces serveurs est hébergé à l'université du Delaware (USA) alors que les deux autres se trouvent au *Laboratoire de Modélisation et Calcul* de l'université Joseph Fourier de Grenoble. L'ensemble de ces serveurs sont en accès libre et illimité et sont accessibles directement à partir de la page *online computing servers*[19] du projet LinBox [20].

1.4 Organisation des codes

Le développement de la bibliothèque LinBox se divise sur trois niveaux comme l'illustre la figure 1.4. Le premier niveau consiste à la définition des domaines de calculs pour l'arithmétique. Typiquement, on retrouve l'arithmétique des corps finis ou celle des anneaux d'entiers. Les implantations disponibles doivent répondre aux critères imposés par des modèles de base qui permettent la généricité des codes de la bibliothèque. Au dessus de ce niveau, on retrouve des modèles de données génériques pour la représentation d'objets nécessaire au calcul en algèbre linéaire tels que les matrices, les vecteurs et les matrices boîtes noires. Chacune de ces structures de données doit aussi répondre à des critères imposés par la bibliothèque. En outre, les vecteurs doivent satisfaire le modèle de conteneur/itérateur proposé par la bibliothèque standard STL alors que les matrices et les boîtes noires doivent satisfaire des modèles de base prédéfinis

[18] http://www.linalg.org/gap.html
[19] http://www.linalg.org/servers.html
[20] http://www.linalg.org

dans la bibliothèque LinBox. Enfin, le dernier niveau de l'organisation des codes concerne l'implantation d'algorithmes. Le principe pour ce dernier niveau est de définir des calculs qui soient génériques à la fois pour l'arithmétique et pour les structures de données utilisées pour modéliser les objets mathématiques. Du fait que les deux couches bas niveau de la bibliothèque sont régies par des modèles de base, l'utilisation de paramètres *template* pour abstraire les implantations utilisées permet de définir facilement des algorithmes génériques de haut niveau. Cependant, pour des raisons d'efficacité ou de faisabilité, les implantations d'algorithmes spécifiques à des instances particulières d'un problème sont effectuées directement sur la structure de données la plus appropriée.

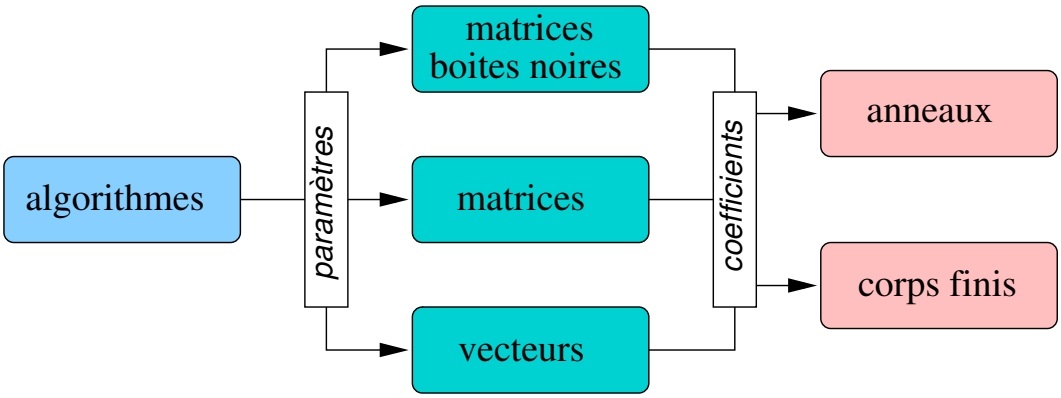

FIG. 1.4 – Hierarchie dans la bibliothèque LinBox

Nous détaillons maintenant l'ensemble des codes disponibles pour ces trois niveaux dans la bibliothèque LinBox. Pour chacun des codes, nous spécifions quelle est la part de notre travail pour la conception et le développement ; $(*) \rightarrow$ faible, $(**) \rightarrow$ moyenne, $(***) \rightarrow$ importante.

Arithmétiques (field/ring)

- `FieldArchetype` (**) : interface compilable et modèle de base (§2.1).
- `FieldTraits` : caractérisation des implantations de corps finis (taille maximale).
- `GivaroZpz, GivaroMong` (***) : adaptateurs des corps premiers de Givaro (§2.2).
- `GivaroGfq`(***) : adaptateurs des extensions algébriques de Givaro (§2.3).
- `GMPRationalField` : corps des rationnels basé sur les entiers GMP.
- `LidiaGfq` (***) : adaptateur des corps finis de LiDIA (§2.3).
- `Modular` (**) : adaptateur générique de corps premiers pour les types natifs (§2.2).
- `NTL_zz_p, NTL_ZZ_p` : adaptateurs des corps premiers de NTL (§2.2).
- `NTL_GF2E, NTL_zz_pE, NTL_ZZ_pE` (**) : adaptateurs des extensions algébriques de NTL (§2.3).
- `UnparametricField` (*) : adaptateur générique des corps finis non paramétré.
- `PID_double` (***) : anneau des entiers (basé sur une représentation flottante)
- `PID_integer` (***) : anneau des entiers (basé sur les entiers GMP).
- `NTL_ZZ, NTL_RR` (*) : adaptateur pour l'anneau des entiers et les nombres flottants en précision arbitraire de NTL.

Matrices boîtes noires (blackbox)

- `BlackboxArchetype` : interface et modèle de base.

- `ButterFly` : préconditionneur à base de réseaux de permutation.
- `ComposeBlackbox` : composition de deux matrices boîtes noires.
- `DenseMatrix` (*) : matrice dense (stockage linéaire)
- `Diagonal` : matrice boîte noire diagonale.
- `Dif` : différence de deux matrices boîtes noires.
- `DirectSum` : somme directe de deux matrices boîtes noires.
- `BlackboxFactory` : changement de domaine de calcul (homomorphisme).
- `Frobenius` : matrice boîte noire pour la forme de Frobenius.
- `Hilbert` : matrice boîte noire pour les matrices de Hilbert.
- `Inverse` : matrice boîte noire pour l'inverse (basée sur le polynôme minimal).
- `LambdaSparseMatrix` (***) : préconditionneur creux.
- `MatrixBlackbox` : adaptateur générique de matrices conformes avec `MatrixArchetype`.
- `MoorePenrose` : matrice boîte noire pour l'inverse généralisée de Moore-Penrose.
- `NAGSparse` : adaptateur générique pour les matrices creuses (format NAG).
- `Hankel` : matrice boîte noire de type Hankel (basée sur les polynômes de NTL).
- `Sylvester` : matrice boîte noire de type Sylvester (à partir de deux polynômes).
- `Toeplitz` : matrice boîte noire de type Toeplitz (basée sur les polynômes de NTL).
- `Permutation` : matrice boîte noire de permutations.
- `ScalarMatrix` : matrice boîte noire pour un scalaire.
- `SparseMatrix` (*) : matrice creuse (trois stockages : parallèle, séquentiel, associatif).
- `Submatrix` (*) : sous-matrice d'une matrice boîte noire.
- `Sum` : somme de deux matrices boîtes noires.
- `Transpose` : transposée d'une matrice boîte noire.
- `ZeroOne` : matrice boîte noire à coefficients zero ou un.

Matrices (matrix)
- `MatrixArchetype` : structure de données définissant l'interface (archétype).
- `BlasMatrix, TriangularBlasMatrix, TransposedBlasMatrix` (***) : interfaces des matrices pour l'utilisation des routines numériques BLAS (§3.4.3).
- `BlasPermutation`(***) : interface des permutations pour les routines de calculs hybrides "exact/numérique" (§3.4.3).
- `DenseMatrixBase` (**) : matrice dense (stockage linéaire à partir d'un vecteur STL).
- `DenseRowsMatrix` : matrice dense (stockage deux dimensions : vecteur de vecteur).
- `DenseSubmatrix` (**) : sous-matrice dense (utilisation de vues de matrices)
- `LQUPMatrix` (***) : matrice dense spécifique au calcul de la factorisation LQUP (§3.4.3).
- `SparseMatrixBase` (*) : matrice creuse (trois stockages : parallèle, séquence, associatif).
- `TransposeMatrix` : transposée d'une matrice.

Vecteurs (vector)
- `BitVector` : vecteur de bits (compression utilisant des entiers machines).
- `randomVector` : générateur de vecteurs aléatoires (creux et denses).
- `ReverseVector` : vecteur miroir.

Algorithmes (algorithms)
- `BlackboxContainer` : structure de conteneur/itérateur pour le calcul des projections par des vecteurs, des puissances d'une matrice boîte noire.
- `BlackboxContainerSymmetric` : structure de conteneur/itérateur pour le calcul des projections par des vecteurs, des puissances d'une matrice boîte noire symétrique.

- `BlackboxBlockContainer` (***) : structure de conteneur/itérateur pour le calcul des projections par des blocs de vecteurs, des puissances d'une matrice boîte noire.
- `BlasMatrixDomain` (***) : domaine de calcul à partir des BLAS pour les matrices denses sur les corps finis (§3.4.3) : produit, inversion, déterminant, rang, système linéaire, polynôme minimal, polynôme caractéristique.
- `LanczosSolver` : domaine de résolution pour les systèmes linéaires sur un corps fini ; méthode de Lanczos [54].
- `BlockLanczosSolver` : domaine de résolution pour les systèmes linéaires sur un corps fini ; méthode de blocs Lanczos [44].
- `MasseyDomain` : domaine de calcul pour le polynôme minimal scalaire d'une matrice boîte noire sur un corps fini ; algorithme de Berlekamp/Massey [56].
- `BlockMasseyDomain` (***) : domaine de calcul pour le polynôme minimal matriciel d'une matrice boîte noire sur un corps fini ; méthode d'approximant de Padé matricie [84, 40, 82].
- `cra` : reconstruction d'entiers par le théorème des restes chinois [36, §5.4, page 102].
- `DiophantineSolver` (***) : domaine de résolution pour les systèmes linéaires diophantiens ; méthode de Giesbrecht (§4.4).
- `GaussDomain` : domaine de calcul pour l'élimination de Gauss sur des matrices creuses sur un corps fini (calcul du rang) [25].
- `IliopoulosElimination` : domaine de calcul pour la diagonalisation de matrices denses à coefficients entiers ; algorithme d'Illiopoulos [47].
- `LiftingContainer` (***) : interface de conteneur/itérateur pour le calcul de développements p-adiques d'une solution rationnelle d'un système linéaire entier (§4.2).
- `RationalReconstruction` (***) : domaine de calcul pour la reconstruction de solutions rationnelles à partir de développements p-adiques (§4).
- `RationalSolver` (***) : interface de résolution de systèmes linéaires entiers (§4.2).
- `RationalSolverAdaptive` : domaine de calcul adaptatif pour la résolution de système linéaire dense.
- `SmithForm` : domaine de calcul pour la forme de Smith de matrices à coefficients entiers ; algorithme EGV [70].
- `SmithFormAdaptive` : domaine de calcul adaptatif pour la forme de Smith de matrices à coefficients entiers.
- `WiedemannSolver` : domaine de résolution pour les systèmes linéaires sur un corps fini ; méthode de Wiedemann [91].
- `BlockWiedemannSolver` (***) : domaine de résolution pour les systèmes linéaires sur un corps fini ; méthode de Wiedemann par blocs [17, 48, 85]

L'ensemble des arithmétiques, des structures de données et des méthodes de calcul fournis par la bibliothèque représente une brique de base importante pour la mise en place d'algorithme de haut niveau en calcul formel. Dans la suite de ce document, nous proposons d'étudier en détail comment certaines parties de la bibliothèque ont été développées. En particulier, nous présentons quels sont les mécanismes mis en place pour intégrer des arithmétiques de corps finis externes même à l'intérieur de codes déjà compilés. Nous nous intéressons ensuite à la réutilisation des routines numériques BLAS pour proposer une boîte à outils très performante pour les problèmes classiques en algèbre linéaire sur un corps fini. Enfin, nous montrons dans la dernière partie, l'intérêt de toutes ces briques de base pour le développement d'une application haut niveau, à savoir la résolution de systèmes linéaires diophantiens.

Chapitre 2

Arithmétique des corps finis

Sommaire

2.1 Archétype de données 28
2.1.1 Modèle de base des corps finis 29
2.1.2 Interface compilable 31
2.1.3 Implantation 34
2.1.4 Performances *vs* généricités 38
2.2 Corps finis premiers 45
2.2.1 Modular 45
2.2.2 GivaroZpz standard 48
2.2.3 GivaroZpz : base logarithmique (*Zech's log*) 48
2.2.4 GivaroZpz : base de Montgomery 50
2.2.5 NTL 51
2.2.6 Performances et surcoût des *wrappers* 53
2.3 Extension algébrique $GF(p^k)$ 59
2.3.1 Givaro 59
2.3.2 NTL 60
2.3.3 LiDIA 61
2.3.4 Performances et surcoût des *wrappers* 63
2.4 Conclusion 65

L'algorithmique en calcul exact et plus particulièrement en algèbre linéaire nécessite des calculs sur de très grands entiers. Les données calculées ont généralement un grossissement de l'ordre de la taille du problème à résoudre. Afin de limiter les conséquences de ce grossissement, une approche classique consiste à calculer la solution modulo plusieurs nombres premiers et reconstruire la solution entière à l'aide du théorème des restes chinois [36, section 5.4]. Le calcul de la solution modulo des nombres premiers p_i repose sur des calculs dans les corps finis $\mathbb{Z}/p_i\mathbb{Z}$. Les performances des opérations arithmétiques de ces corps finis sont donc un critère important de l'efficacité de cette méthode.

Dans certains cas, le calcul de solutions probabilistes permet d'obtenir des gains en complexité non négligeables. Sur un corps fini, les probabilités de réussite de ces méthodes sont directement reliées à la taille du corps finis. Afin d'augmenter les probabilités de réussite, une approche classique consiste à plonger les corps finis dans une extension algébrique. La validité de cette méthode s'appuie sur le fait que les extensions conservent les propriétés du corps de base.

Le développement d'arithmétiques de corps finis et d'extensions algébriques est donc une brique de base importante pour la mise en place de solutions pour l'algèbre linéaire exacte. Le choix d'une représentation ou d'une arithmétique particulière est somme toute dépendant des algorithmes et des problèmes à résoudre. L'objectif de la bibliothèque LinBox est de proposer un moyen simple et efficace pour configurer à la demande les implantations d'algorithmes en fonction des corps finis, aussi bien dans des codes compilés que dans des codes interprétés. L'intérêt est de pouvoir réutiliser des corps finis provenant de bibliothèques externes sans avoir à récrire l'ensemble de la bibliothèque, et à plus long terme, de bénéficier des évolutions futures en arithmétique des corps finis. L'utilisation de bibliothèques externes permet de bénéficier de corps finis performants sans avoir à réécrire leurs implantations.

Dans ce chapitre, nous étudions les mécanismes utilisés dans la bibliothèque LinBox pour fournir des codes configurables et nous nous intéressons aux différentes implantations de corps finis existantes dans des bibliothèques spécialisées. La première section présente les modèles de données utilisés pour définir une interface de données compilable au travers d'un archétype de données. Les deux sections suivantes illustrent les différentes implantations de corps finis disponibles au sein de la bibliothèque LinBox et peuvent être vues comme un survol de ce qui existe en matière d'implantation de corps finis. Notre objectif dans ce chapitre est de montrer que l'utilisation de mécanismes d'abstraction pour les corps finis dans LinBox influe très peu sur les performances intrinsèques des bibliothèques utilisées alors qu'elle permet de définir des codes évolutifs et robustes.

2.1 Archétype de données

Afin de fournir un ensemble d'arithmétiques de corps finis facilement interchangeables dans les implantations d'algorithmes et dans les classes d'objets génériques, la bibliothèque LinBox propose de définir les corps finis au travers d'un archétype de donnée. Cet archétype, comme les interfaces en Java, a pour but de fixer le nom des méthodes et la structure de données des corps finis. L'utilisation de cet archétype comme modèle de base pour l'ensemble des codes de corps finis permet de développer des codes indépendants des implantations en utilisant des paramètres *template* (voir §1.1.2). Néanmoins, l'utilisation de ces paramètres entraîne deux désavantages. Le premier est qu'ils ne permettent pas de fournir des instances de codes compilés. En effet, il faut que lors de la compilation l'ensemble des types d'instanciations soient connus.

L'autre désavantage est que tous les codes *template* sont dupliqués pour chaque type d'instanciation. De ce fait, le temps de compilation est assez coûteux et les codes compilés peuvent atteindre une taille conséquente.

Une alternative proposée par la bibliothèque LinBox est de définir un archétype de corps finis au travers d'un objet compilable. L'archétype est une implantation de corps fini abstraite qui définit des indirections sur des corps finis concrets au travers de pointeurs et de fonctions virtuelles. L'utilisation de cet archétype comme instance permet à la fois de limiter la taille des codes générés et de fournir une instance de codes compilables. La généricité des paramètres *template* est ici remplacée par du polymorphisme dynamique au travers de classes abstraites (voir §1.1.2). Un avantage de ce type d'archétype est qu'il permet de valider à la compilation l'ensemble des codes basés sur les corps finis sans avoir à fournir d'implantation réelle. Une autre alternative possible est de définir un archétype dans lequel toutes les fonctions sont définies au travers d'un pointeur et de gérer à la main les indirections. Cette approche est similaire à la nôtre mais nécessite une gestion plus complexe de l'interface et des objets à manipuler. De plus, elle complique considérablement la définition et la prise en main des codes.

Dans cette section, nous développons les différents mécanismes mis en place pour définir l'archétype des corps finis dans la bibliothèque LinBox. Nous abordons dans un premier temps le modèle de données utilisé pour représenter les corps finis et nous définissons ces spécifications. Ensuite, nous nous intéressons aux mécanismes utilisés pour définir un archétype de données compilable intégrant du polymorphisme dynamique. Enfin, nous étudierons les différents surcoûts apportés par les niveaux de conception et d'utilisation de cet archétype.

2.1.1 Modèle de base des corps finis

La définition d'un archétype de données permet de fixer le modèle de base pour tous les corps finis. Toutes les implantations de corps finis doivent respecter cet archétype afin de fournir des objets génériques pour la bibliothèque, c'est-à-dire qu'elles fournissent l'ensemble des méthodes requises par l'archétype.

La totalité des opérations du modèle de corps fini sont définies au travers d'un domaine de calcul. Ce domaine définit l'interface de calculs pour les éléments d'un corps fini. Soient `x,y` deux éléments du corps fini `F`, pour calculer au travers des éléments `x,y` il faut toujours utiliser le corps `F`. Par exemple l'opération `z=x+y` sur `F` s'effectue par l'appel de la fonction : `F.add(z,x,y)` où `z` est un troisième élément de `F`.

La manipulation des éléments au travers d'un domaine de calcul permet de séparer leur représentation de la façon dont on calcule avec. Cela permet entre autres de définir un type d'élément totalement indépendant du corps sur lequel il va pouvoir être utilisé. Toutes les informations relatives au corps sont stockées uniquement à l'intérieur du domaine. Les éléments n'ont aucune connaissance de leur corps d'appartenance. L'utilisation d'un domaine n'a aucune incidence sur les performances des appels de fonction du fait des méthodes d'*inlining* des compilateurs. Par contre, ce domaine permet de minimiser la taille mémoire des éléments et permet d'éviter l'exclusivité relative à l'utilisation d'un corps fini statique comme c'est le cas dans la bibliothèque NTL (voir §2.2.5).

Le modèle de base des éléments se limite donc à une structure de données et aux opérations standard d'une classe (constructeurs, affectation, destructeur). Le domaine quant à lui définit toutes les opérations relatives aux éléments (arithmétiques, tests d'égalités, entrées-sorties, initialisations, conversions) ainsi que des opérations d'information sur les calculs (caractéristique, cardinalité). Chaque domaine de calcul encapsule son propre type d'éléments. Afin de fournir

une définition générique de ces types de données dans les domaines, on utilise un alias pour les renommer (`typedef` en C++). Ainsi, pour n'importe quel domaine de calcul `Field` on peut accéder à son type d'éléments en utilisant le type `Field::Element`.

Une autre propriété du modèle de corps fini de la bibliothèque LinBox est qu'il encapsule un type de générateur aléatoire d'éléments. Comme pour les éléments, ce type est directement accessible au travers du corps par l'alias `Field::RandIter`. Le modèle de données de ces générateurs s'appuie sur le concept des itérateurs de la STL [62, 35]. L'idée est de considérer ces générateurs comme une suite infinie et homogène d'éléments d'un corps fini ou d'un de ses sous-ensembles. Il suffit alors de fournir un itérateur de cette structure qui soit positionné aléatoirement pour générer une suite aléatoire d'éléments. En fixant l'itérateur de façon déterministe, on peut alors générer une suite d'éléments spécifique du générateur. C'est lors de la construction du générateur que l'itérateur est positionné à partir d'une graine d'aléa et d'un corps fini, ou d'un sous-ensemble. Le parcours de la structure du générateur se fait au travers de la fonction `random(Element &a)` qui affecte la valeur de l'itérateur à `a` et incrémente l'itérateur.

La figure 2.1 présente le modèle de base des corps finis avec ses différentes encapsulations de types. Toutes les implantations de corps finis de la bibliothèque LinBox doivent intégrer ces trois types de données en respectant leurs archétypes respectifs. L'ensemble des méthodes et l'organisation de ces modèles de données sont disponibles au travers des codes 2.1, 2.2 et 2.3 ou de la documentation[21] en ligne de la bibliothèque LinBox.

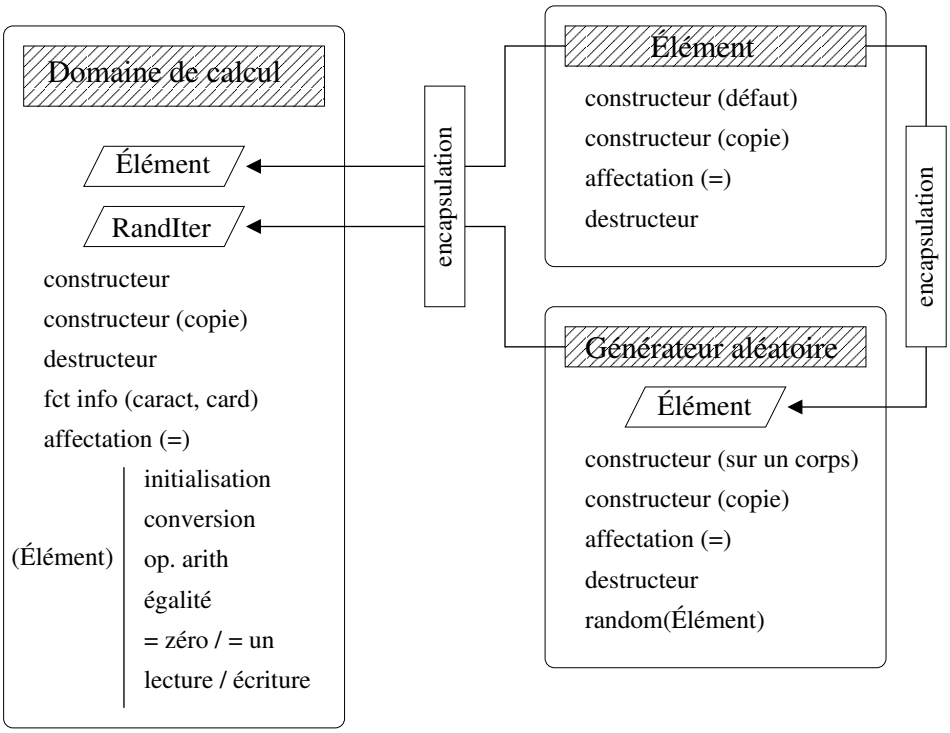

FIG. 2.1 – Spécifications du modèle de base des corps finis.

[21] http://www.linalg.org/docs.html

Soit le type `Field` qui définit une implantation de corps premiers de type $\mathbb{Z}/p\mathbb{Z}$ en accord avec le modèle de base de la bibliothèque LinBox. On définit alors l'exemple suivant :

```
1  Field F(19);
2  Field::Element a,b,c;
3  F.init(a,13);
4  F.init(b,3);
5  F.init(c);
6  F.mul(c,a,b);
```

Ce premier exemple construit tout d'abord à la ligne `1` le corps premier `F` de caractéristique 19. La ligne `2` déclare trois éléments `a,b,c` du type d'élément encapsulé par le type `Field`. Les lignes `3,4,5` permettent d'initialiser ces trois éléments sur le corps `F`. Cette étape est primordiale pour le bon fonctionnement des codes dans la bibliothèque. Nous verrons dans la section suivante que cette étape d'initialisation permet de fixer les allocations mémoire des éléments lorsque l'archétype est utilisé comme instance. Enfin, la ligne `6` permet d'appeler la fonction d'addition du corps `F` sur les éléments `a,b` et de stocker le résultat dans l'élément `c`. Le passage de la valeur de retour comme paramètre dans les opérations sur les éléments est ici obligatoire du fait qu'on peut manipuler les éléments uniquement à partir du domaine. En particulier, l'opération d'affectation des éléments doit se faire par l'appel de la fonction `assign(Element&, const Element&)` et non pas avec l'opérateur égal (`=`). L'implantation d'opérations avec paramètre de retour permet d'éviter l'utilisation de la fonction `assign` directement sur les valeurs de retour des fonctions (`F.assign(c,F.mul(a,b)) ;`). Une autre possibilité proposée dans le modèle des corps finis est d'effectuer les opérations en place dans l'un des deux opérandes. Les opérations sur un corps fini étant commutatives on peut donc fixer cette valeur de retour au premier opérande. Dans l'exemple précédent on peut donc remplacer la fonction `mul(c,a,b)` par la fonction `mulin(a,b)`.

Dans l'exemple suivant nous précisons l'utilisation des générateurs aléatoires.

```
1  Field F(19);
2  Field::RandIter G(F,19,123456789);
3  Field::Element a,b,c;
4  F.init(a);
5  F.init(b);
6  F.init(c);
7  G.random(a);
8  G.random(b);
9  F.mul(c,a,b);
```

Dans ce nouvel exemple, nous avons changé les valeurs des éléments par des valeurs aléatoires. Le générateur aléatoire `G` est initialisé sur le corps `F` et le positionnement de l'itérateur est défini à partir de la graine 123456789. Le paramètre 19 précise que l'on considère l'ensemble du corps pour générer les éléments aléatoires. Le reste des calculs est identique à l'exemple précédent.

2.1.2 Interface compilable

Comme nous l'avons vu dans la section précédente, le premier rôle de l'archétype est de fixer le modèle de base des corps finis. Nous avons aussi vu que ce modèle comportait trois types de données (domaines de calcul, éléments, générateurs aléatoires). Chacune de ces structures de données doit respecter son propre archétype afin d'offrir la généricité complète des implantations de corps finis.

Une autre caractéristique de l'archétype est de servir d'interface de données pour proposer des instances compilées des codes de la bibliothèque LinBox. L'intérêt de proposer une telle interface est de pouvoir fournir des codes compilés qui soient robustes et facilement reconfigurables. L'approche classique en C++ pour définir de telles interfaces est d'utiliser une hiérarchie de classes basée sur une classe de base abstraite [80, §12.4.2]. En effet, nous avons vu dans la partie 1.1.2 que les tables virtuelles permettent d'accéder aux fonctions à partir du type dynamique des objets (voir figure 1.3). Toutefois, la construction d'objets abstraits reste impossible. De ce fait, ces objets ne peuvent exister qu'au travers de pointeurs ou de références sur un objet dérivant de la classe abstraite. L'utilisation d'interfaces abstraites comme paramètres des fonctions nécessite de manipuler tous les objets de façon dynamique, ce qui est incompatible avec la définition de codes *template* du fait que l'allocation des éléments est faite de façon statique pour des raisons de performance.

Une alternative proposée par la bibliothèque LinBox est d'encapsuler ces interfaces abstraites à l'intérieur d'objets compilables. Le but de ces objets est de gérer l'ensemble des allocations mémoire des objets abstraits tout en conservant le rôle d'interface de données. Pour cela, la gestion mémoire est ici relayée au constructeur et au destructeur de la classe alors que le polymorphisme est assuré par des indirections sur les fonctions de l'interface abstraite. L'utilisation de cette interface comme instanciation des codes *template* permet de fournir une instance de code compilable sans avoir à modifier les codes génériques.

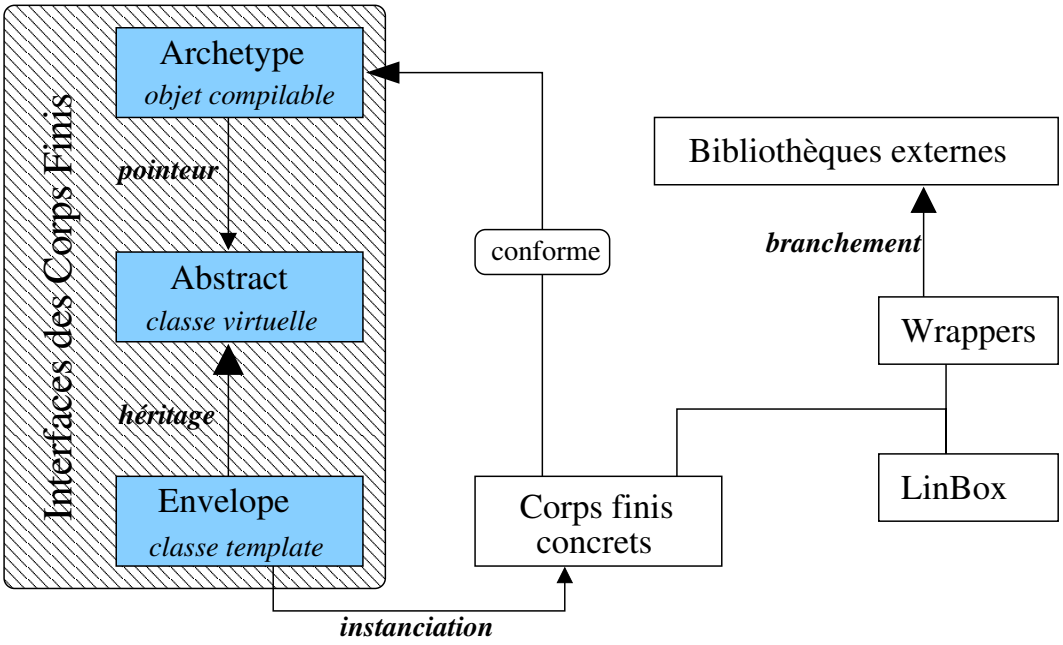

FIG. 2.2 – Structure de l'archétype des corps finis.

La figure 2.2 décrit la structure d'une interface de données compilable. Cette structure se découpe en trois modules. Tout d'abord on peut distinguer au plus bas niveau les classes **Abstract** et **Envelope**. La classe **Abstract** définit l'interface abstraite, en proposant un modèle de fonction virtuelle pure pour toutes les fonctions du modèle de base. La classe **Envelope** permet de synchroniser l'ensemble des corps finis conformes au modèle de base avec l'interface

abstraite. Plus précisément, cette classe permet d'intégrer de façon générique des fonctions d'allocations mémoire nécessaires à la manipulation d'objets abstraits. Cette classe est optionnelle si l'on utilise directement une implantation de corps finis qui dérive de l'interface abstraite et qui propose l'ensemble des fonctions d'allocations mémoire. La classe **Archetype** définit à la fois le modèle de base des corps finis et l'objet compilable intégrant l'interface abstraite. L'intégration de l'interface abstraite se fait au travers d'un pointeur de données. L'instanciation de ce pointeur par un objet dérivé de l'interface abstraite permet donc de fixer le polymorphisme dynamique de l'interface. L'utilisation de l'enveloppe pour instancier ce pointeur permet d'assurer le bon comportement des allocations mémoire pour toutes les implantations de corps finis.

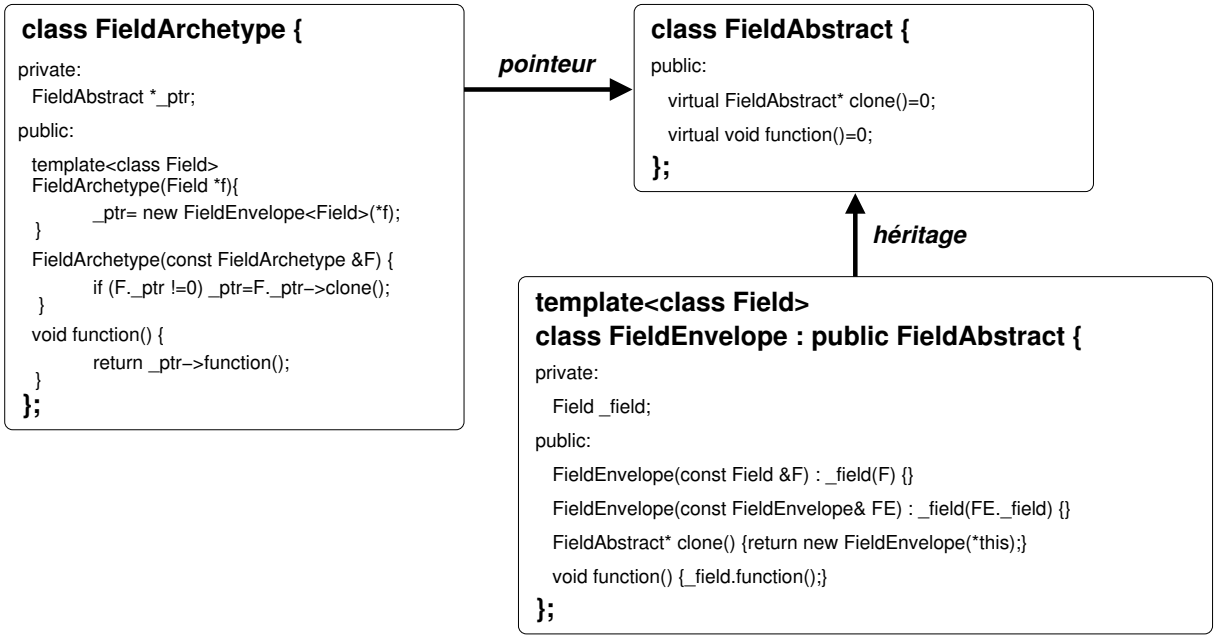

Fig. 2.3 – Code pour une interface abstraite compilable.

Nous nous intéressons maintenant aux détails d'implantation de l'archétype de données des corps finis. La figure 2.3 décrit les différents mécanismes C++ utilisés pour obtenir une interface de données compilable. La classe `FieldArchetype` intègre la classe `FieldAbstract` au travers du pointeur de données `_ptr`. Afin de conserver le polymorphisme de l'interface abstraite, le constructeur est générique sur le type de corps fini. La construction d'un archétype sur un corps fini concret entraîne l'allocation du pointeur `_ptr` au travers de l'enveloppe.

La redirection de toutes les fonctions de l'archétype vers les fonctions de l'interface abstraite au travers du pointeur `_ptr` permet de rendre l'archétype polymorphe. Ainsi, l'appel de la fonction `ptr->function()` dans la définition de la fonction `function()` de l'archétype entraîne l'appel de la fonction `_field.function` rattachée au corps d'instanciation de l'archétype. On remarque que ce type d'appel comporte trois niveaux d'indirection : pointeur `FieldAbstract`, tables virtuelles de `FieldAbstract`, `FieldEnvelope`. L'enveloppe peut ne pas être considérée comme une indirection car les méthodes d'*inlining* des compilateurs absorbent en partie les définitions par appel de fonctions.

La partie la plus importante de cet archétype concerne la gestion mémoire des objets abstraits. En dehors de sa construction, l'archétype n'a aucune connaissance du type de corps sur

lequel il agit. Il ne connaît en fait que l'interface abstraite. Afin de permettre à l'archétype de créer des copies de lui-même, une méthode classique des interfaces abstraites [80, §15.6.2] est de proposer une fonction `clone()` définie dans l'interface abstraite qui permet de construire des copies dynamiques des objets abstraits. Grâce à cette fonction, il est alors possible de créer des objets abstraits sans connaître leur type dynamique. Le constructeur par copie de l'archétype utilise donc cette fonction pour créer des copies d'objets abstraits.

2.1.3 Implantation

Nous venons de voir dans la section précédente comment l'archétype des corps finis définit une interface de données compilable. Néanmoins, cette implantation n'est pas suffisante pour décrire le modèle complet de l'archétype des corps finis. Dans la partie 2.1.1 nous avons vu que le modèle de base encapsule un type d'élément et un type de générateur aléatoire. Bien que chacun de ces types possède un archétype définissant une interface compilable, suivant le même modèle que celui décrit dans la figure 2.3, l'interaction de ces différents archétypes n'est pas immédiate. En effet, la construction d'éléments abstraits au sein de l'archétype des corps finis ne peut se faire que si l'archétype connaît le type concret de ces éléments, ce qui n'est pas le cas. Il en est de même pour les générateurs aléatoires.

Une solution proposée dans la bibliothèque LinBox est de stocker les types dynamiques des éléments abstraits et des générateurs abstraits. Cela se traduit en pratique par l'intégration dans l'archétype d'un pointeur sur un `ElementAbstract` et d'un pointeur sur un `RandIterAbstract`. Ces pointeurs sont instanciés lors de la construction de l'archétype à partir d'un corps fini concret (voir code 2.1 lignes 15, 16, 26 et 27). Grâce à ces pointeurs et aux fonctions `clone()`, l'archétype peut donc générer des éléments abstraits relatifs au corps fini sur lequel est instancié l'archétype. C'est d'ailleurs à partir de ce mécanisme que la fonction `init` est capable d'allouer de la mémoire pour des éléments abstraits (voir code 2.1 ligne 70). Toutefois, la fonction `clone` n'est pas suffisante pour allouer un archétype de générateur aléatoire. En plus du corps finis, le générateur nécessite une graine d'aléa et la taille de l'échantillon des éléments aléatoires. Dans ce cas précis, la fonction `clone` est remplacée par la fonction `construct` qui permet la construction d'un générateur abstrait paramétrable (voir code 2.3 ligne 14). Le pointeur `_randiter_ptr` de la classe `FieldArchetype` sert uniquement à récupérer la fonction `construct` à partir du type dynamique du générateur associé au corps fini instancié dans l'archétype.

L'utilisation de ces deux pointeurs pourrait cependant être remplacée par deux fonctions virtuelles `construct_elem` et `construct_randiter` définies dans la classe `FieldAbstract`. La construction dynamique des éléments et des générateurs serait alors directement attachée au bon type de corps finis. Cette solution semble raisonnable du fait que les types de ces derniers sont déjà encapsulés dans le modèle des corps finis. De plus, cela permettrait une gestion plus transparente de l'archétype. Néanmoins, cette solution entraîne une dépendance entre les archétypes qui peut s'avérer gênante

Nous avons vu dans la section précédente que l'utilisation de l'enveloppe pouvait être inutile si le corps utilisé dérive directement de l'interface abstraite. Cette option est prise en considération dans la construction de l'archétype. La solution proposée consiste à utiliser une fonction `construct` qui utilise l'enveloppe uniquement si le corps ne dérive pas de la classe abstraite. Cette fonction est définie en deux exemplaires et utilise une spécialisation d'un pointeur `void` pour récupérer la hiérarchie de classe du corps utilisé (voir code 2.1 lignes 11 et 23).

Enfin, le dernier point important de l'interaction entre les différents archétypes est la défi-

nition des opérations sur les éléments du corps. Ces fonctions sont définies dans l'archétype au travers de l'archétype des éléments. Afin d'implanter ces fonctions directement à partir d'appels de fonction de l'interface abstraite, il faut pouvoir accéder au type abstrait des éléments dans l'archétype des corps finis. Cela est possible en pratique grâce à la définition mutuelle des archétypes en tant qu'amis à partir du caractère `friend` (voir code 2.2 lignes 5 et 6).

Soient `Field1` et `Field2` deux modèles de corps finis conforment à l'archétype de la bibliothèque LinBox, on peut alors définir l'exemple suivant pour l'utilisation de l'archétype.

```
  std::vector<ElementArchetype> genvector (const FieldArchetype &F,
                                           size_t                n,
                                           size_t            sample,
                                           size_t              seed)
  {
    std::vector<ElementArchetype> u(n);
    std::vector<ElementArchetype>::iterator pu= u.begin();
    RandIterArchetype G(F,sample,seed);
    for(; pu != u.end(); ++pu){
      F.init(*pu);
      G.random(*pu);
    }
    return u;
  }

  int main(){

    Field1 F1(17);
    Field2 F2(101);
    FieldArchetype FA1(&F1), FA2(&F2);

    std::vector<ElementArchetype> u1,u2;
    u1=genvector(FA1,100,17,123456789);
    u2=genvector(FA2,100,101,987654321);
}
```

Cet exemple montre qu'on peut proposer une fonction `genvector` qui est à la fois générique et compilable. La fonction `genvector` est compilable car elle est définie à partir de l'archétype et de types de base qui sont connus à la compilation. Cette fonction est générique du fait des propriétés polymorphes de l'archétype. En pratique, il suffit d'instancier des archétypes à partir de corps finis concrets, ici les archétype `FA1` et `FA2` pour les corps concrets `F1` et `F2`. L'appel de la fonction `genvector` à partir de ces archétypes permet de générer des vecteurs aléatoires à la fois dans `F1` et dans `F2`.

Ce type de code en compilation séparée est impossible à la fois pour des paramètres *template* et pour des interfaces abstraites. Il est clair qu'en utilisant des paramètres *template* on ne pourrait pas compiler la fonction `genvector` sans préciser les types `Field1` et `Field2`. Par contre, à partir de classes abstraites on ne pourrait pas générer les vecteurs `u1` et u2 car la STL nécessite que les objets des conteneurs possèdent des constructeurs, ce qui n'est pas le cas des objets abstraits. Une alternative possible serait d'utiliser des vecteurs de pointeurs mais on voit tout de suite que la gestion mémoire des éléments alourdirait considérablement le code à définir.

Code 2.1 – Archétype des corps finis

```
class FieldArchetype {

  private:
    mutable FieldAbstract        *_field_ptr;
    mutable ElementAbstract      *_elem_ptr;
    mutable RandIterAbstract *_randiter_ptr;

    // constructor from a Field which inherits
    // from FieldAbstract (Envelope is not needed)
    template<class Field>
    void constructor (FieldAbstract *traits, Field *F) {
      _field_ptr     = new Field(*F);
      _elem_ptr      = new typename Field::Element();
      _randiter_ptr = new typename Field::RandIter(*F);
    }

    // constructor from a Field which does not inherit
    // from FieldAbstract (Envelope is needed)
    template<class Field>
    void constructor (void *traits, Field *F) {
      _field_ptr     = new FieldEnvelope<Field> (*F);
      _elem_ptr      = new ElementEnvelope<Field>();
      _randiter_ptr = new RandIterEnvelope<Field> (*F);
    }

  public:
    typedef ElementArchetype   Element;
    typedef RandIterArchetype RandIter;

    // constructor of field archetype
    template <class Field>
    FieldArchetype (Field *f){
      constructor(f,f);
    }

    // copy constructor
    FieldArchetype (const FieldArchetype &F) {
      if (F._field_ptr    != 0)   _field_ptr   = F._field_ptr->clone();
      if (F._elem_ptr     != 0)   _elem_ptr    = F._elem_ptr->clone();
      if (F_randiter_ptr != 0) _randiter_ptr = F._randiter_ptr->clone();
    }

    // destructor
    ~FieldArchetype(void) {
      if (_field_ptr     != 0)  delete _field_ptr;
      if (_elem_ptr      != 0)  delete _elem_ptr;
      if (_randiter_ptr != 0)  delete _randiter_ptr;
    }

    // assignement operator
    FieldArchetype &operator= (const FieldArchetype &F) {
      if (this != &F) {
        if (_field_ptr      != 0) delete _field_ptr;
        if (_elem_ptr       != 0) delete _elem_ptr;
        if (_randiter_ptr   != 0) delete _randiter_ptr;
```

```
        if (F._field_ptr    != 0) _field_ptr     = F._field_ptr->clone();
        if (F._elem_ptr     != 0) _elem_ptr      = F._elem_ptr->clone();
        if (F_randiter_ptr  != 0) _randiter_ptr  = F._randiter_ptr->clone();
      }
    }

    // initialization of elements archetype over the field archetype
    Element &init(Element &x, const integer &y) const {
      if (x._elem_ptr != 0) delete x._elem_ptr;
      x._elem_ptr = _elem_ptr->clone();
      _field_ptr->init(*x._elem_ptr,y);
      return x;
    }

    // addition of elements archetype
    Element& add(Element &x,
                 const Element &y,
                 const Element &z) const {
    _field_ptr->add(*x._elem_ptr, *y._elem_ptr, *z._elem_ptr);
    return x;
    }
};
```

Code 2.2 – Archétype des éléments

```
class ElementArchetype {

  private:
    friend class FieldArchetype;
    friend class RandIterAbstract;
    mutable ElementAbstract  *_elem_ptr;

  public:
    // default constructor
    ElementArchetype (void) {_elem_ptr=0;}

    // copy construcutor
    ElementArchetype (const ElementArchetype &a) {
      if (a._elem_ptr != 0) _elem_ptr = a._elem_ptr->clone();
      else _elem_ptr=0;
    }

    // destructor
    ~ElementArchetype (void) {
      if (_elem_ptr != 0) delete _elem_ptr;
    }

    // affectation operator
    ElementArchetype &operator= (const ElementArchetype &a) {
      if (this != &a) {
        if (_elem_ptr   != 0 ) delete _elem_ptr;
        if (a._elem_ptr != 0) _elem_ptr = a._elem_ptr->clone();
      }
      return *this;
    }
```

```
};
```

Code 2.3 – Archétype des générateurs aléatoires

```
class RandIterArchetype {

  private:
    mutable RandIterAbstract  *_randiter_ptr;

  public:
    typedef ElementArchetype Element;

    // constructor from a field archetype, a size of sampling, and a seed
    RandIterArchetype (const FieldArchetype &F,
                       const integer &size=0,
                       const integer &seed=0)
    {
      _randiter_ptr=F._randiter_ptr->construct(*F._field_ptr, size, seed);
    }

  // copy constructor
  RandIterArchetype (const RandIterArchetype &R)
  {
    if ( R._randiter_ptr != 0)
    _randiter_ptr = R._randiter_ptr->clone();
  }

  // destructor
  ~RandIterArchetype(void) {delete _randiter_ptr;}

  // assignement operator
  RandIterArchetype &operator= (const RandIterArchetype &R)
  {
    if (*this != &R) {
      if (_randiter_ptr   != 0) delete _randiter_ptr;
      if (R._randiter_ptr != 0) _randiter_ptr= R._randiter_ptr->clone();
    }
    return *this;
  }

  Element &random(Element &a) const
  {
    _randiter_ptr->random(*a._elem_ptr);
    return a;
  }
};
```

2.1.4 Performances *vs* généricités

L'utilisation de l'archétype pour l'instanciation d'un corps fini entraîne un surcoût dû à la manipulation abstraite des objets et des fonctions. Dans cette partie nous nous intéressons

à quantifier ces différents surcoûts et à observer les différents impacts sur les implantations concrètes. Nous effectuons nos tests à la fois pour une architecture 32 bits (Pentium III 1Ghz, 512 Mo RAM) et pour une architecture 64 bits (Itanium I 733 Mhz, 1Go RAM). Nous utilisons le compilateur gcc version 3.0 pour le PIII et gcc version 3.3 pour l'Itanium avec dans les deux cas l'option `-O3`. Afin d'évaluer les performances des archétypes en fonction de la structure des corps finis, nous utilisons à la fois des corps premiers et des extensions algébriques. Les implantations de corps finis que nous utilisons sont décrites dans les parties 2.2 et 2.3 de ce document.

Nous focalisons nos tests sur le produit scalaire de vecteurs denses et sur une élimination de Gauss de matrices denses qui sont deux opérations clés des algorithmes en algèbre linéaire. Concernant le produit scalaire, nos tests s'appuient sur des vecteurs aléatoires d'ordre 1000 et nous utilisons 100000 itérations pour les corps premiers et 10000 itérations pour les extensions afin d'observer des temps de calcul supérieurs à la seconde. Pour l'élimination de Gauss, nous utilisons une matrice carrée de plein rang d'ordre 500 et nous calculons son rang. Les temps de calcul relevés sont basés sur un chronométrage des ressources utilisateur. Dans la suite, nous donnons les performances du produit scalaire et de l'élimination de Gauss pour le corps premier $\mathbb{Z}/1009\mathbb{Z}$ et l'extension $\mathtt{GF}(3^7)$ en fonction du niveau de généricité. Par commodité, nous notons les corps premiers $\mathbb{Z}/p\mathbb{Z}$ par $\mathbb{Z}_p$. Nous désignons par "corps LinBox" les implantations utilisant directement les corps finis au travers de paramètres *template*. Les termes "envelope", "abstract" et "archetype" font référence à l'utilisation des corps finis aux différents niveaux d'abstraction de l'archétype.

Nous présentons les performances obtenues pour nos tests dans les tables 2.1 et 2.2 pour le corps premier $\mathbb{Z}_{1009}$ et dans les tables 2.3 et 2.4 pour l'extension algébrique $\mathtt{GF}(3^7)$. Les figures 2.4, 2.5, 2.6 et 2.7 illustrent le surcoût relatif de l'archétype par rapport à chaque implantation concrète de corps finis ("corps LinBox"). Ces figures n'ont pas vocation à comparer les différentes implantations. Une implantation de corps finis qui entraîne un grand surcoût dans l'archétype n'est pas pour autant l'implantation la moins performante.

Dans un premier temps, nous remarquons que l'encapsulation de l'interface abstraite à l'intérieur d'une couche compilable n'entraîne qu'un faible surcoût. Par exemple, l'élimination de Gauss sur le corps fini GivaroZpz<Std32> nécessite 6.96 secondes au travers de l'interface abstraite et 7.42 secondes au travers de l'archétype (voir table 2.1, Gauss), ce qui représente une différence de seulement 6%. En moyenne, le surcoût de l'archétype par rapport à l'interface abstraite est de moins de 10%. Contrairement à ce que nous pensions, on observe que l'utilisation de l'enveloppe entraîne un surcoût qui n'est pas négligeable. Il semble que le compilateur ne parvienne pas à complètement *inliner* les appels de fonctions effectués par l'enveloppe. En particulier, l'utilisation de l'enveloppe pour le corps fini GivaroZpz<Std32> entraîne un surcoût de $4.96 - 2.43 = 2.26$ secondes alors que l'archétype entraîne un surcoût de $7.42 - 2.43 = 4.99$ secondes (voir table 2.1). Ce qui signifie que l'enveloppe est responsable de 45% du surcoût total de l'utilisation de l'archétype. D'après l'ensemble de nos tests, le surcoût de l'enveloppe est de l'ordre de 50% du surcoût de l'archétype.

Dans un deuxième temps, nous observons une certaine corrélation entre le surcoût de l'archétype et la structure du corps finis sous-jacente. De manière générale, le surcoût de l'archétype est bien moins important pour les extensions algébriques que pour les corps premiers. On remarque qu'il y a en moyenne un facteur 10 entre ces deux types de corps. Cela provient du fait que les structures utilisées pour représenter les extensions algébriques sont plus complexes que celles généralement utilisées pour les corps premiers. Typiquement, on utilise des polynômes pour représenter les éléments des extensions algébriques alors que pour les corps premiers on utilise des

types natifs du langage (int, double) voire des entiers en précision arbitraire (GMP). Les calculs dans les extensions sont donc plus coûteux en pratique et de ce fait l'utilisation de l'archétype est moins pénalisante que pour des calculs directement effectués à partir des unités arithmétiques des processeurs. Par exemple, l'utilisation de l'archétype avec l'extension NTL_zz_pE, qui utilise une représentation polynomiale, entraîne un surcoût de seulement 8% pour l'élimination de Gauss (voir table 2.3) alors que pour l'extension GivaroGfq qui utilise une représentation logarithmique et des entiers 32 bits ce surcoût est de 48%. Nous observons les mêmes différences entre les corps premiers utilisant des précisions machine et ceux utilisant des précisions arbitraires. Notamment, nous remarquons que les corps premiers NTL_ZZ_p et Modular<integer> qui sont basés sur des entiers GMP entraînent un surcoût très faible au travers de l'archétype en comparaison des autres implantations. En particulier, ces deux implantations entraînent un surcoût respectif de 14% et 1% pour l'élimination de Gauss (voir table 2.2) alors que les autres implantations entraînent toutes un surcoût supérieur à 100%. Bien que le surcoût de l'archétype varie en fonction des implantations, on observe que l'utilisation de l'archétype conserve les performances des implantations utilisées à l'ordre de grandeur près. Néanmoins, l'archétype permet de donner une approximation assez précise de la meilleure implantation concrète. Dans les huit tables que nous présentons, seulement une seule ne préserve pas la meilleure implantation concrète au travers de l'archétype.

Enfin, on peut remarquer un meilleur comportement général de l'archétype dans le cadre du produit scalaire que pour l'élimination de Gauss. Nous pensons que ces différences proviennent essentiellement du jeu de données de nos tests. En particulier, les données du produit scalaire sont suffisamment petites pour tenir dans les caches alors que cela n'est pas le cas pour l'élimination de Gauss. Néanmoins, une autre explication possible pourrait provenir des différences de comportement mémoire des applications. Le produit scalaire n'utilise en effet qu'une seule fois les données alors que le pivot de Gauss effectue un ensemble de mise à jour des données. L'écriture de données étant plus critique que la lecture, cela expliquerait les différences de comportement. À partir de nos tests, nous ne pouvons émettre aucune corrélation sûre entre les archétypes et les applications mises en place.

En conclusion, nous pouvons dire que l'utilisation d'archétypes de données n'est pas plus coûteuse que l'utilisation d'interfaces abstraites seules. Les archétypes offrent plusieurs avantages facilitant le développement de codes génériques :

- gestion des allocations de données ;
- dissociation interface/modèle de données dans les implantations (enveloppe) ;
- réutilisation des codes génériques *template* ;
- conservation des performances intrinsèques des implantations sous-jacentes.

De plus, le développement de bibliothèques compilées au travers des archétypes permet de fournir des codes robustes et facilement configurables. Ainsi, un utilisateur pourra évaluer directement ces implantations au travers des algorithmes proposés par la bibliothèque ou tout simplement utiliser la bibliothèque sur sa propre implantation. La réutilisation de ces archétypes à plus haut niveau doit permettre à la bibliothèque LinBox d'être à la fois un outil performant pour les calculs en algèbre linéaire mais aussi une plate-forme d'évaluation pour les corps finis.

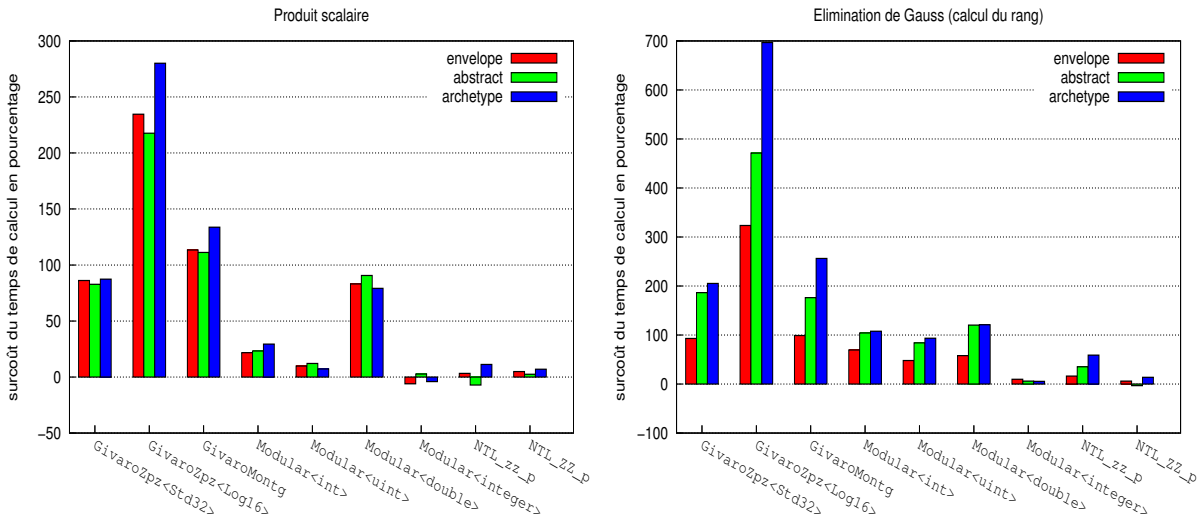

FIG. 2.4 – Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`).

Produit scalaire (ordre=1000, iterations=100000)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroZpz<Std32>	4.13	7.69	7.55	7.74
GivaroZpz<Log16>	1.36	4.55	4.32	5.17
GivaroMontg	2.76	5.89	5.83	6.45
Modular<int>	9.14	11.13	11.27	11.83
Modular<uint>	11.32	12.43	12.69	12.14
Modular<double>	3.31	6.06	6.31	5.93
Modular<integer>	279.07	262.36	286.51	267.77
NTL_zz_p	22.34	23.07	20.76	24.85
NTL_ZZ_p	83.80	87.90	85.88	89.62
Élimination de Gauss (ordre=500)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroZpz<Std32>	2.43	4.69	6.96	7.42
GivaroZpz<Log16>	0.94	3.98	5.37	7.49
GivaroMontg	2.19	4.36	6.05	7.80
Modular<int>	4.45	7.54	9.09	9.24
Modular<uint>	5.47	8.09	10.06	10.59
Modular<double>	3.78	5.97	8.33	8.35
Modular<integer>	123.34	135.46	130.76	130.06
NTL_zz_p	10.66	12.40	14.43	16.96
NTL_ZZ_p	48.87	51.78	47.27	55.58

TAB. 2.1 – Performances (secondes) de l'archétype des corps finis sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`).

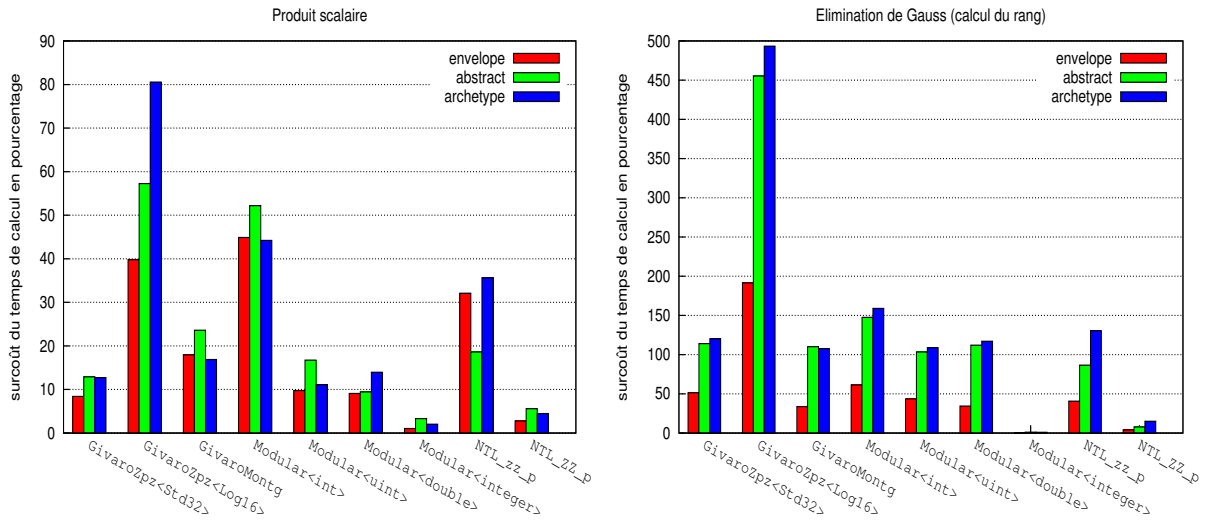

FIG. 2.5 – Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`).

Produit scalaire (ordre=1000, iterations=100000)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroZpz<Std32>	15.34	16.63	17.33	17.29
GivaroZpz<Log16>	3.92	5.49	6.17	7.09
GivaroMontg	11.02	13.00	13.62	12.88
Modular<int>	8.50	12.32	12.94	12.26
Modular<uint>	16.83	18.48	19.65	18.70
Modular<double>	12.81	13.98	14.03	14.6
Modular<integer>	524.58	529.96	541.98	535.15
NTL_zz_p	10.23	13.52	12.14	13.88
NTL_ZZ_p	151.85	156.05	160.37	158.63
Élimination de Gauss (ordre=500)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroZpz<Std32>	6.45	9.77	13.81	14.20
GivaroZpz<Log16>	1.70	4.96	9.46	10.10
GivaroMontg	5.40	7.22	11.35	11.21
Modular<int>	4.56	7.36	11.30	11.82
Modular<uint>	7.01	10.07	14.26	14.63
Modular<double>	5.76	7.74	12.21	12.50
Modular<integer>	243.66	243.89	246.19	246.09
NTL_zz_p	5.45	7.67	10.17	12.57
NTL_ZZ_p	97.72	101.71	105.28	112.32

TAB. 2.2 – Performances (secondes) de l'archétype des corps finis sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`).

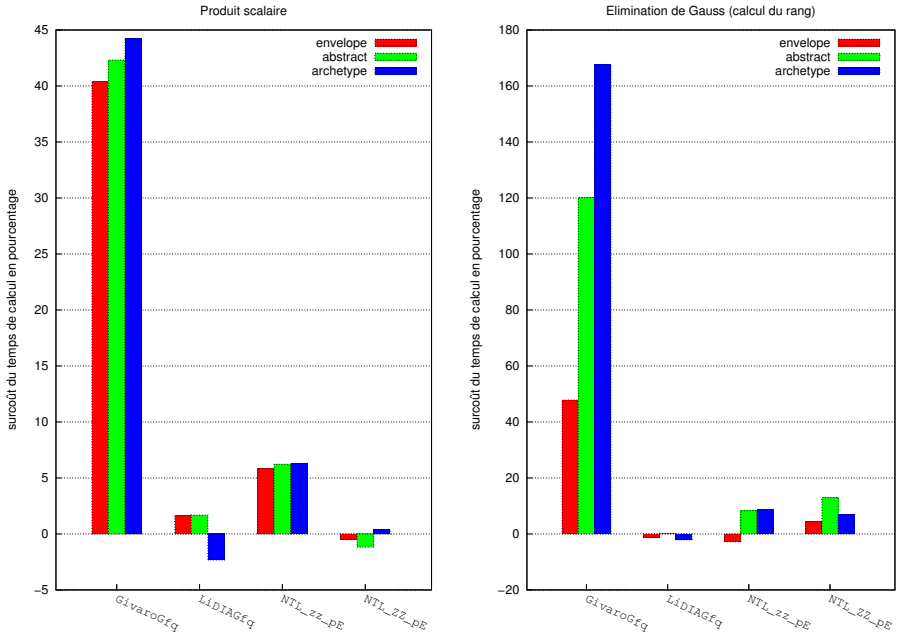

FIG. 2.6 – Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathtt{GF}(3^7)$ (`P3-1Ghz`).

Produit scalaire (ordre=1000, iterations=10000)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroGfq	0.52	0.73	0.74	0.75
LiDIAGfq	301.20	306.09	306.29	294.20
NTL_zz_pE	75.42	79.83	80.11	80.16
NTL_ZZ_pE	279.23	277.89	275.95	280.31
Élimination de Gauss (ordre=500)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroGfq	2.28	3.37	5.02	6.10
LiDIAGfq	1523.92	1505.11	1527.43	1492.29
NTL_zz_pE	40.89	39.74	44.30	44.43
NTL_ZZ_pE	87.57	91.49	98.91	93.58

TAB. 2.3 – Performances (secondes) de l'archétype des corps finis sur $\mathtt{GF}(3^7)$ (`P3-1Ghz`).

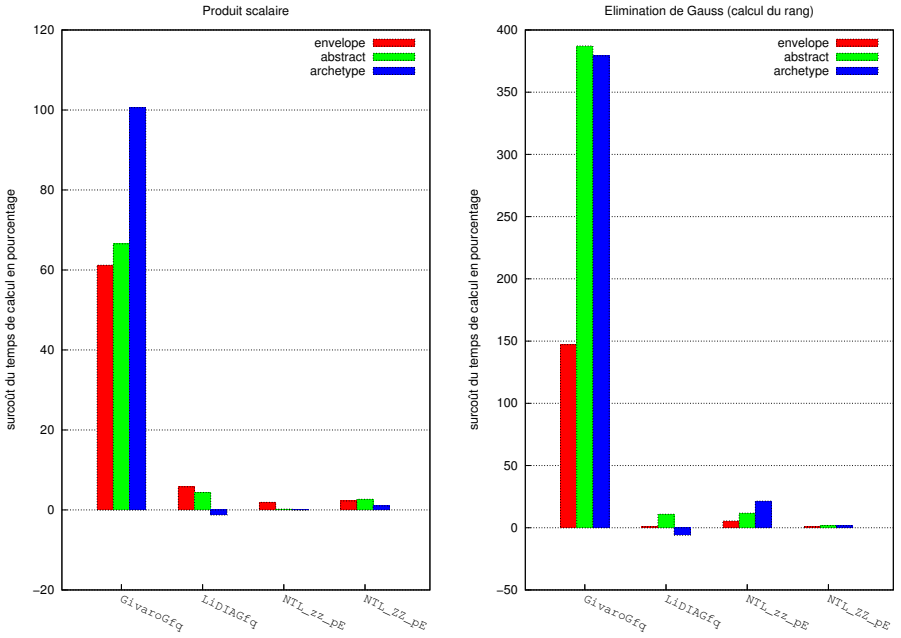

FIG. 2.7 – Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathtt{GF}(3^7)$ (`IA64-733Mhz`).

Produit scalaire (ordre=1000, iterations=10000)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroGfq	0.51	0.82	0.85	1.02
LiDIAGfq	490.25	518.63	511.69	484.05
NTL_zz_pE	100.41	102.31	100.62	100.37
NTL_ZZ_pE	475.56	486.29	488.20	480.41
Élimination de Gauss (ordre=500)				
	corps LinBox	**enveloppe**	**abstract**	**archetype**
GivaroGfq	1.39	3.44	6.78	6.67
LiDIAGfq	2659.99	2684.45	2948.50	2497.17
NTL_zz_pE	63.14	66.60	70.55	76.65
NTL_ZZ_pE	157.36	158.88	160.18	160.00

TAB. 2.4 – Performances (secondes) de l'archétype des corps finis sur $\mathtt{GF}(3^7)$ (`IA64-733Mhz`).

2.2 Corps finis premiers

Nous nous intéressons dans cette partie aux implantations de corps finis premiers. En particulier, nous présentons les différentes implantations disponibles au sein de la bibliothèque LinBox. Ces implantations proviennent essentiellement de bibliothèques externes telles que NTL[22], Givaro[23] et LiDIA[24]. Pour chacune des bibliothèques, nous décrivons les implantations de corps premiers utilisées ainsi que la façon dont elles sont définies. Nous présentons aussi les implantations de corps premiers développées directement dans la bibliothèque LinBox. Notre objectif est de fournir un ensemble d'implantations de corps premiers variées afin de proposer un large choix d'arithmétiques. Parmi toutes ces implantations, nous nous intéressons à déterminer laquelle est la plus performante et sous quelles conditions.

2.2.1 Modular

L'implantation `Modular` est une implantation propre à la bibliothèque LinBox. Le principe de cette implantation est d'utiliser les mécanismes templates pour fournir une arithmétique modulaire générique en fonction du type de représentation des éléments. Pour cela, il faut que les types d'instanciation de cette classe fournissent les opérateurs arithmétiques classiques (+, -, ×, /) ainsi que l'opération de réduction modulo (%). De plus, le modèle de ces éléments doit respecter l'archétype de la bibliothèque LinBox (voir code 2.2). En particulier, cette classe définit des implantations de corps premiers pour les types natifs entiers du langage (int, long). Le nombre premier définissant les caractéristiques de l'arithmétique est encapsulé dans la classe par un attribut protégé (`protected _modulus`). À partir de maintenant, nous utilisons l'alias `Element` encapsulé dans l'archétype pour spécifier le type générique des éléments dans l'implantation.

```
  template< class Type>
  class Modular {
    public:
       typedef Type Element;
       ...
};
```

À partir de cette classe, on peut définir l'ensemble des opérations de façon générique. Par exemple, la fonction d'addition est implantée par une addition classique suivie d'une phase de correction si la valeur calculée n'appartient plus à la représentation.

```
  Element &add(Element &a, const Element &b, const Element &c) const {
    a=b+c;
    if (a > _modulus) a-= _modulus;
    return a;
  }
```

[22]http://www.shoup.net/ntl
[23]http://www-apache.imag.fr/software/givaro
[24]http://www.informatik.tu-darmstadt.de/TI/LiDIA/

Pour la fonction de multiplication, la détection de correction ne peut se faire par de simples comparaisons car la valeur intermédiaire est trop grande. Dans ce cas, on utilise la fonction de réduction modulaire (%). Cette fonction est directement disponible dans la bibliothèque standard (libc) pour les entiers machines au travers d'algorithmes de division entière.

```
Element &mul(Element &a, const Element &b, const Element &c) const {
  return a = (b * c) % _modulus;
}
```

L'utilisation de ce type de réduction est coûteux en pratique car elle fait appel à une opération de division. Nous essayons donc de limiter au maximum son utilisation. Par exemple, l'opération `axpy` qui correspond à une multiplication et une addition combinées (i.e. `axpy`(r,a,x,y) => $r = ax + y \mod _modulus$) est implantée au travers d'une seule réduction modulaire après le calcul entier exact.

Toutefois, ce choix d'implantation restreint la taille des corps premiers possibles du fait que les calculs intermédiaires doivent rester exacts. Il faut donc assurer que la taille des corps premiers définis au travers de cette classe permette de toujours effectuer les calculs intermédiaires de façon exacte. Pour cela, il suffit de limiter la taille du modulo en fonction des valeurs intermédiaires maximales. L'opération la plus restrictive est la fonction `axpy`. Pour un type d'éléments ayant une précision de m bits, le modulo p doit satisfaire

$$(p-1)^2 + p - 1 < 2^m. \tag{2.1}$$

Cette restriction est somme toute importante car cela signifie que pour des entiers machine 32 bits non signés, le corps premier maximal sera défini pour $p = 65521$, soit 16 bits. Afin de fournir des implantations sur les entiers machine autorisant des corps premiers plus grands, nous proposons des spécialisations de la classe `Modular` pour les entiers machine signés et non signés 8, 16 et 32 bits.

Ces spécialisations utilisent essentiellement des conversions de types dans les opérations les plus restrictives : multiplication et `axpy`. L'implantation de ces fonctions est basée sur une conversion des opérandes dans un type autorisant une précision deux fois plus grande. Par exemple, l'implantation de la fonction `axpy` pour les entiers non signés 32 bits (uint32) se fait à partir d'une conversion vers des entiers non signés 64 bits (uint64), de calcul sur 64 bits et d'une conversion sur 32 bits.

```
uint32 & axpy(uint32 &r,
              const uint32 &a,
              const uint32 &x,
              const uint32 &y) const {

  return r= ((uint64) a * (uint64) x + (uint64) y) % (uint64) _modulus;
}
```

En utilisant cette implantation pour les uint32, on augmente la taille des corps possibles de $p < 2^{16}$ à $p < 2^{31}$ avec une faible perte d'efficacité. En effet, cette implantation tire parti du fait que la plupart des processeurs proposent une instruction de multiplication 32 bits $\times$ 32 bits qui donne le résultat sur 64 bits stockés dans deux registres 32 bits (instruction "imul" sur Intel [19]). Ainsi, le seul surcoût de cette méthode provient de l'utilisation d'une réduction modulo 64 bits à la place d'une réduction 32 bits qui est intrinséquement moins performante du fait des algorithmes de division implantés dans les bibliothèques standard. Les conversions sont ici gratuites car le compilateur considère uniquement les résultats de la multiplication et de la réduction comme des entiers 64 bits. Les entiers 64 bits étant en pratique deux registres 32 bits, la conversion vers des entiers 32 bits est immédiate. Ce sont maintenant les opérations d'addition et de soustraction qui sont limitantes car elles nécessitent un bit supplémentaire pour stocker les valeurs intermédiaires. L'utilisation de la même méthode pour ces opérations n'est pas intéressante car il n'existe pas d'instruction d'addition $(32 + 32) \rightarrow 64$ bits et de plus elle ne permettrait de gagner qu'un seul bit sur la taille des corps premiers.

Une autre possibilité proposée au travers d'une spécialisation de cette classe générique est d'utiliser les nombres flottants machine pour atteindre des corps premiers plus grands. Les nombres flottants double précision possèdent une mantisse codée sur 53 bits qui permet de représenter de façon exacte des entiers strictement inférieurs à 2^{53}. Bien que la norme IEEE-754 [1] spécifie qu'un des bits de la mantisse est implicite du fait d'une représentation fractionnaire, le positionnement de la virgule garantit les 53 bits de précision pour des valeurs entières. L'utilisation de ces nombres flottants permet donc de définir des corps premiers allant jusqu'à $p \leq 94906265$, c'est-à-dire de l'ordre de 26 bits. L'implantation des opérations arithmétiques est la même que dans le cas entier, sauf pour l'opération de réduction modulaire qui est ici effectuée en utilisant la fonction `fmod` des bibliothèques standard. Cette fonction retourne le reste entier de la division de deux nombres flottants dans un format flottant.

Enfin, une dernière alternative proposée dans la bibliothèque LinBox est d'utiliser des entiers en précision arbitraire. En particulier, nous utilisons l'interface `integer` (voir §1.2.1) des entiers multiprécisions GMP[25] pour instancier la classe `Modular`. L'utilisation d'entiers multiprécision permet de définir des corps premiers de très grande taille, les entiers GMP proposent une précision maximale de $m2^m$ où m est le nombre de bits d'un entier machine (32 ou 64 suivant l'architecture).

Nous avons vu que la plupart des opérations de la classe `Modular` sont effectuées à partir d'opérations sur les entiers suivies d'une réduction modulo. Néanmoins l'opération d'inversion des éléments du corps ne peut se faire en suivant ce schéma. En effet, l'inverse modulaire y d'un entier x est la solution de l'équation $yx \equiv 1 \mod p$. Une façon classique de résoudre cette équation est d'utiliser l'algorithme d'Euclide étendu [36, théoreme 4.1] qui calcule les coefficients de Bezout $s, t \in \mathbb{Z}$ tel que $sx + tp = 1$ et donc $1/x \equiv s \mod p$.

En pratique, on utilise une version partielle de l'algorithme d'Euclide étendu [21, §1.4.5] qui permet de calculer uniquement le coefficient s de l'équation de Bezout. Cette version permet d'éviter un tiers des opérations arithmétiques de l'algorithme traditionnel. La fonction d'inversion modulaire peut alors s'écrire

```
Element &inv(Element &x, const Element &y) const {
  Element x_int, y_int, q, tx, ty, tmp;
  x_int = _modulus;
```

[25] http ://www.swox.com/gmp/

```
  y_int = y;
  tx    = 0;
  ty    = 1;

  while (y_int != 0) {
    q      = x_int / y_int;
    tmp    = y_int;
    y_int  = x_int  - q*y_int;
    x_int  = temp;
    tmp    = ty;
    ty     = tx - q*ty;
    tx     = temp;
  }
  if (x < 0) x += _modulus;

  return x;
}
```

Une remarque intéressante par rapport à l'algorithme d'Euclide étendu classique est qu'en pratique on préférera utiliser la version partielle pour calculer un seul coefficient et déduire le deuxième à partir de l'équation $t = (sx - \mathrm{pgcd}(x, p))/p$. En particulier, ce calcul devient intéressant quand l'un des deux opérandes est une puissance de la base de représentation des entiers car ce calcul se ramène à une multiplication, une soustraction et un décalage.

2.2.2 GivaroZpz standard

La bibliothèque Givaro[26] propose une implantation de corps premiers basée sur une arithmétique modulaire standard. Ces arithmétiques sont implantées au travers d'un domaine de calcul basé sur l'archétype des corps finis de la bibliothèque LinBox. Plusieurs types de données sont disponibles : entier signés 16, 32 et 64 bits. Les opérations arithmétique sont implantées de la même manière que pour l'implantation `Modular` générique (calculs entiers puis réduction modulo) sans changement de précision. Cela signifie que l'ordre des corps premiers est borné respectivement par 2^8, 2^{16} et 2^{32}.

L'intégration de ces implantations au sein de la bibliothèque LinBox est directe du fait de l'utilisation de l'archétype des corps finis LinBox comme modèle de développement. Néanmoins, les fonctions d'initialisation et de conversion définies par l'archétype sont basées sur les entiers multiprécisions de la bibliothèque LinBox et ne peuvent être implantées qu'à l'intérieur de celle-ci. Ces fonctions sont donc définies à l'intérieur de classes qui héritent des implantations d'origine. Nous avons développé un *wrapper* standard dans LinBox pour intégrer ces implantations au travers d'un classe générique `GivaroZpz<Std>` où le paramètre template `Std` représente la précision du type des éléments. Ce *wrapper* standard nous sert aussi pour intégrer l'implantation logarithmique présentée dans la prochaine partie. On utilise alors le paramètre `Log16` pour définir ces implantations dans LinBox (`GivaroZpz<Log16>`).

2.2.3 GivaroZpz : base logarithmique (*Zech's log*)

Une autre implantation proposée par la bibliothèque Givaro est basée sur une représentation logarithmique des éléments du corps. L'intérêt d'un telle représentation est qu'elle permet

[26] http://www-lmc.imag.fr/Logiciels/givaro/

d'effectuer les opérations de multiplication à partir de simples additions. Ce qui en pratique permet d'obtenir de meilleures performances du fait que la latence des unités entière d'addition est plus faible que celle des unités entières de multiplication [20, table 3.1, page 42]. Toutefois, l'opération d'addition devient moins performante du fait qu'elle nécessite une lecture de table.

Une propriété mathématiques des corps finis est que l'ensemble des inversibles d'un corps fini définit un groupe multiplicatif cyclique. En d'autres termes, si l'on connaît un générateur de ce groupe, on peut exprimer tous les inversibles par une puissance de ce générateur. Si $\mathbb{Z}_p$ est un corps premier et $\mathbb{Z}_p^*$ est le sous-groupe des inversibles de $\mathbb{Z}_p$ alors il existe au moins un élément $g \in \mathbb{Z}_p^*$ tel que tous les éléments de $\mathbb{Z}_p^*$ sont représentables au travers d'une puissance de g. L'élément g est une racine primitive de p et il définit un générateur du groupe multiplicatif $\mathbb{Z}_p^*$. Le nombre des inversibles du corps premier $\mathbb{Z}_p$ est égal $p-1$ et $\mathbb{Z}_p^* = \mathbb{Z}_p \backslash \{0\}$. En codant le zéro du corps par une valeur spéciale on peut donc représenter tous les éléments d'un corps fini uniquement à partir des puissances de g [25].

En utilisant ce codage des éléments, les opérations du groupe multiplicatif peuvent s'effectuer uniquement par des opérations sur les exposants.

- multiplication : $g^i \times g^j = g^{i+j \mod p-1}$
- division : $g^i / g^j = g^{i-j \mod p-1}$

Par contre, les opérations de la loi additive du corps ne peuvent se faire directement à partir des exposants. Cependant, en précalculant certaines valeurs on ramène facilement ces opérations dans le groupe multiplicatif [25, §3.1]. En connaissant les successeurs de chaque élément et la valeur de l'exposant pour -1 (notée `_mone`), les opérations d'addition, de soustraction et de négation sont définies par

- négation : $-g^i = g^{i+_mone}$
- addition : $g^i + g^j = g^i \times (1 + g^{j-i \mod p-1})$
- soustraction : $g^i - g^j = g^i \times (1 - g^{j-i \mod p-1})$

Toutefois, les opérations faisant intervenir le zéro du corps doivent être à part du fait que le codage de cet élément n'est pas une puissance de g. En pratique, ce traitement alourdit les opérations arithmétiques par un test d'égalité des opérandes avec le codage du zéro du corps. Afin d'éviter ce traitement, la bibliothèque Givaro propose d'utiliser des lectures de table [25, §4.1.3]. L'idée consiste à lire le résultat des opérations dans le corps à partir d'opérations sur le codage des éléments, le zéro du corps compris. En codant le zéro du corps par la valeur $2p-2$, il faut alors stocker une table de $5(p-1)$ éléments pour retrouver l'ensemble des résultats possibles des opérations de la loi multilicative du corps. Les éléments inversibles du corps étant codés par des valeurs comprises entre 0 et $p-2$, les valeurs possibles pour les résultats d'addition et de soustraction des éléments du corps (le zéro compris) sont donc comprises entre $-2p+2$ et $3p-4$, soit $5(p-1)$ valeurs différentes. Soit i le résultat de l'addition ou de la soustraction du codage de deux éléments du corps, on définit la table de correspondance $t[i]$ pour le codage du résultat :

$$\begin{cases} t[i] = & 2p-2 & -p+1 & \leq & i < 0; \\ t[i] = & i & 0 & \leq & i < p-1; \\ t[i] = & i-(p-1) & p-1 & \leq & i < 2p-2; \\ t[i] = & 2p-2 & 2p-2 & \leq & i < 4p-5. \end{cases}$$

L'implantation finale nécessite de stocker un total de $15p$ éléments en mémoire : $5p$ pour les opérations arithmétiques, $4p$ pour la table des successeurs, $4p$ pour la table des prédécesseurs et $2p$ pour les conversions. Grâce à ces tables, les opérations dans le corps se ramènent uniquement à des opérations d'addition et de soustraction sur des entiers et à des lectures de tables, ce qui en

pratique permet d'obtenir de très bonnes performances (voir §2.2.6). Cependant, les ressources mémoires requises limitent fortement la taille des corps premiers possibles. Par exemple, l'utilisation de cette implantation pour un corps premier de taille 2^{20} nécessite 60Mo de ressources mémoire. Du fait de la limitation imposée par cette méthode, la bibliothèque Givaro propose une implantation basée sur des entiers signés 16 bits qui permettent de limiter la taille des corps premiers à 2^{15} et donc limite les ressources mémoire à 1Mo.

2.2.4 GivaroZpz : base de Montgomery

Une autre implantation de la bibliothèque Givaro est basée sur l'utilisation de la représentation de Montgomery. En 1985 Peter. L. Montgomery a proposé un algorithme de multiplication modulaire pour des moduli génériques sans aucune division [59]. Le principe de cet algorithme est de changer la représentation des entiers en les multipliant par une puissance de la base de représentation et d'utiliser l'équation de Bezout pour remplacer la division par un simple décalage. Soient $x, y \in \mathbb{Z}$ tels que $x = \sum_{i=0}^{k} x_i b^i$ et $y = \sum_{i=0}^{k} y_i b^i$, soit $r = b^s$ tel que $r > p > x, y$ alors la représentation de x et y dans la base de Montgomery est $\bar{x} = xr \mod p$ et $\bar{y} = yr \mod p$. Soit $z \equiv xy \mod p$ alors la représentation de z dans la base de Montgomery doit être équivalente au produit de $\bar{x}$ et $\bar{y}$. L'algorithme proposé par Montgomery permet de satisfaire cette équivalence.

Algorithme `Montgomery-Multmod(`$\bar{x}$`,`$\bar{y}$`,p,r)`
Entrées : $\bar{x} = xr \mod p$, $\bar{y} = yr \mod p$, p, r
Sortie : $\bar{x}\bar{y}r^{-1} \mod p = xyr \mod p$
Conditions : $x, y < p < r = b^s$, $\mathrm{pgcd}(p, r) = 1$
Précalculs : $p' = (-p)^{-1} \mod r$
  $q := (\bar{x}\bar{y} \mod r)p' \mod r$
  $t := (\bar{x}\bar{y} + pq)/r$
  si $(t \geq p)$ **alors**
    $t := t - p$
retourner t ;

Lemme 2.2.1. *Soient $x, y \in \mathbb{Z}_p$ et r une puissance de la base de représentation de x et y tel que $p < r$ et $\mathrm{pgcd}(p, r) = 1$. Si l'on connaît $p' = (-p)^{-1} \mod r$ alors la multiplication de x par y modulo p s'effectue à partir de 2 multiplications, 1 décalage et une addition grâce à l'algorithme* `Montgomery-Multmod`.

Preuve. Pour prouver que l'algorithme `Montgomery-Multmod` est correct il faut montrer que $\bar{x}\bar{y} + qp$ est divisible par r et que le résultat est borné par $2p$ car par construction $t \equiv \bar{x}\bar{y}r^{-1} \mod p$. En remplaçant la variable q par sa valeur dans l'expression $\bar{x}\bar{y} + qp$ on peut écrire l'équation suivante :

$$\begin{aligned} \bar{x}\bar{y} + qp &\equiv \bar{x}\bar{y} + \bar{x}\bar{y}p'p \mod r^2 \\ &\equiv \bar{x}\bar{y}(1 + pp') \mod r^2. \end{aligned}$$

En utilisant l'équation de Bezout $rr' - pp' = 1$ on obtient $\bar{x}\bar{y} + qp \equiv \bar{x}\bar{y}rr' \mod r^2$, ce qui signifie que $\bar{x}\bar{y} + qp$ est bien divisible par r. Par hypothèse, on sait que $\bar{x}, \bar{y} < p < b$ et par construction de q on a $q < r$. On peut donc borner la valeur de t par

$$t < (p^2 + pr)/r < (2pr)/r < 2p.$$

Les seules opérations effectuées durant cet algorithme sont 2 multiplications, 1 addition et 1 soustraction. Du fait que r est une puissance de la base de représentation, les opérations de réduction modulo r et de division par r s'effectuent respectivement à partir d'un masque bit à bit et d'un décalage. □

L'implantation proposée par la bibliothèque Givaro est donc basée sur une représentation des éléments dans la base de Montgomery, c'est-à-dire que pour un entier $x \in \mathbb{Z}_p$, sa représentation est $x.2^s \mod p$ tel que $p < 2^s$ avec 2^s la base de Montgomery. Les opérations d'addition et de soustraction sont effectuées de façon classique (opération puis réduction) du fait que la loi additive est convervée par la représentation ($x + y \mod p \rightarrow x2^s + y2^s \mod p$). L'opération de multiplication est implantée par l'algorithme `Montgomery-Multmod` qui conserve les propriétés de la représentation. Toutefois, l'opération d'inversion nécessite une implantation particulière car l'inverse modulaire classique par résolution de l'équation de Bezout ne conserve pas la représentation : $(x.2^s)^{-1} \mod p \not\equiv x^{-1}2^s \mod p$. L'implantation proposée par la bibliothèque Givaro consiste à corriger le résultat obtenu par l'inversion classique en le multipliant par une puissance de la base de Montgomery. En utilisant l'algorithme `Montgomery-Multmod` pour effectuer cette correction, il faut alors choisir le cube de la base de Montgomery.

$$\texttt{Montgomery} - \texttt{Multmod}(\texttt{2}^{\texttt{3s}} \mod \texttt{p}, (\texttt{x2}^{\texttt{s}})^{-1} \mod \texttt{p}, \texttt{p}, \texttt{2}^{\texttt{s}}) = \texttt{x}^{-1}\texttt{2}^{\texttt{s}} \mod \texttt{p}.$$

L'utilisation de la base de Montgomery pour représenter les éléments d'un corps fini permet de bénéficier d'une opération de multiplication modulaire sans aucune division. Néanmoins, l'algorithme `Montgomery-Multmod` nécessite de calculer des valeurs intermédiaires plus grandes que dans l'approche classique avec division. En effet, il faut calculer q et t qui sont respectivement bornés par $q \leq (r-1)^2$ et $t \leq (p-1)^2 + p(r-1)$. En considérant que les éléments ont une précision de m bits, la taille des corps finis et la base de Montgomery doivent satisfaire le système suivant :

$$\begin{cases} (r-1)^2 & < & 2^m, \\ (p-1)^2 + p(r-1) & < & 2^m. \end{cases} \tag{2.2}$$

L'implantation proposée par la bibliothèque Givaro s'appuie sur des entiers non signés 32 bits avec une base de Montgomery $r = 2^{16}$. La taille des corps finis possible est donc limitée par $p <= 40499$, soit 15 bits. En pratique, la fonction de multiplication-addition (`axpy`) n'est pas implantée à la manière de la multiplication car cela nécessiterait une multiplication supplémentaire pour synchroniser l'opérande de l'addition avec le résultat de la multiplication ($\bar{x}\bar{y} + \bar{z} = xyr^2 + zr$). L'implantation de cette opération par deux opérations successives est ici plus efficace.

Comme pour les autres implantations de corps finis de la bibliothèque Givaro, l'intégration de cette implantation à la bibliothèque LinBox est immédiate en redéfinissant les fonctions d'initialisation et de conversion sur le type d'entiers multiprécision de LinBox. Le *wrapper* LinBox intégrant cette implantation est la classe `GivaroMontg`.

2.2.5 NTL

L'implantation de l'arithmétique des corps premiers dans la bibliothèque NTL se base sur une arithmétique modulaire classique. La définition de corps premiers se fait par l'initialisation d'une variable globale fixant la caractéristique du corps. Ainsi les opérations sont directement implantées sur les éléments du corps premier sans avoir à définir de domaine de calcul. Cette

approche ne permet cependant que la manipulation d'un seul corps premier à la fois du fait du caractère statique des caractéristiques.

Deux types d'arithmétique de corps premiers sont disponibles. La première appelée `NTL::zz_p` est basée sur des entiers machine 32 bits ou 64 bits selon l'architecture utilisée. L'arithmétique utilisée est une arithmétique modulaire classique (opérations + réduction). Une des particularités de cette implantation est qu'elle propose des schémas particuliers de réduction modulaire.

La réduction modulaire d'entiers compris entre 0 et $2p$, typiquement le résultat d'une addition modulo p, peut être effectuée sans aucune comparaison. L'idée est de toujours calculer $z = x - p$ pour $0 \leq x < 2p$ et de se servir du bit de signe de z pour corriger le résultat. En effet, en récupérant le bit de signe de z par décalage on obtient soit 0 si le résultat est positif ou nul soit -1. Du fait que les nombres négatifs sont codés en complément à la base, le codage binaire de -1 est une suite de 1 ($-1 = 1111...1111$) alors que le codage de 0 est une suite de 0. En utilisant le codage binaire du signe de z comme masque bit à bit sur p, on obtient la valeur de la correction (0 ou p) qu'il faut ajouter à z pour obtenir le bon résultat.

Soit $x, y \in \mathbb{Z}_p$, codés sur des entiers signés ayant une précision de m bits l'addition modulaire $z \equiv x + y \mod p$ s'écrit

```
z= x+y-p+((x+y-p)>>m-1)&p
```

Une autre idée développée dans la bibliothèque NTL est d'utiliser une approximation du quotient de la division entière pour effectuer les réductions modulaires après multiplication des opérandes [4]. En pratique, l'idée est d'extraire la partie entière du quotient flottant. Ce quotient approche le quotient exact à 1 près à cause des arrondis flottants. Il suffit donc de calculer le résultat en soustrayant le produit du quotient approché et du modulo, et finir de corriger s'il le faut. On peut donc calculer le reste de la division entière en soustrayant le produit du quotient approché et du modulo, et finir de corriger s'il le faut.

Le modulo étant toujours le même pour un corps premier donné, le précalcul de l'inverse du modulo sur un nombre flottant double précision permet de remplacer la division par une multiplication. Cette méthode de réduction modulaire permet de remplacer la division entière par une multiplication flottante, une multiplication entière et quelques opérations d'addition/-soustraction. En pratique, cette réduction est plus efficace que la version par division entière car la plupart des processeurs ne possèdent pas d'unité arithmétique de division entière. Néanmoins, les conversions entre entiers et nombres flottants peuvent entraîner des effets de bord coûteux, en particulier au niveaux des *pipelines*. L'implantation de cette réduction modulaire s'écrit :

```
long NTL_mod(long r, long t, long modulus, double inv_modulus){
  long q = ((double) t) * inv_modulus;
  r = t - q*t;
  if (r > modulus) r-= modulus;
  if (r < 0)       r+= modulus;
  return r;
}
```

Cette réduction modulaire est essentiellement utilisée dans NTL pour l'opération de multiplication modulaire simple précision. Cependant, cette réduction pourrait être utilisée pour l'implantation de la fonction `axpy` mais cette dernière n'est pas disponible dans la bibliothèque NTL. Une partie intéressante de cette réduction est qu'elle autorise une taille de corps premier supérieure à la moitié de la précision des entiers. En effet, la taille des corps premiers est ici bornée par $p < 2^{30}$ pour des machines 32 bits et $p < 2^{52}$ pour des machines 64 bits. Cela provient du fait que le calcul du reste par soustraction ne fait intervenir que les bits de poids faible. La valeur absolue du reste obtenu par soustraction est ici inférieure à $2p$. Il suffit donc que $2p$ soit représentable pour utiliser cette réduction. Cette implantation nécessitant des entiers signés, la borne maximale est donc de $2^{m'-1}$, où m' définit le nombre de bits de précision des entiers non signés. Toutefois, cette borne n'est plus valable pour les entiers 64 bits du fait de l'utilisation de flottants double précision. En effet, pour que le calcul du quotient approché soit correct à 1 près, il faut que les opérandes soient représentables en double précision. Or, nous avons vu précédemment que le plus grand entier représentable à partir d'un flottant double précision est $2^{53} - 1$. Cela implique donc qu'il faut limiter la taille des corps premiers à $p < 2^{52}$ pour des machines 64 bits.

Une alternative possible qui n'est pas implantée dans la bibliothèque NTL serait d'utiliser une implantation des doubles étendus qui garantissent une mantisse d'au moins 64 bits sans aucun bit implicite [1]. En pratique, les `long double` en C propose une mantisse codée sur 64 bits. L'utilisation de ce type de données permettrait donc d'obtenir une limite pour les corps premiers de $p < 2^{63}$ sur n'importe quelle architecture.

L'autre implantation de corps premiers, appelée `NTL::ZZ_p`, est basée sur des entiers multiprécision et une arithmétique modulaire classique. Les entiers multiprécisions utilisés peuvent soit provenir de la bibliothèque GMP[27] soit provenir d'une implantation fournie par la bibliothèque NTL elle-même. L'arithmétique modulaire est effectuée par des calculs entiers suivis soit par une correction soit par une division entière multiprécision.

L'intégration de ces deux implantations dans la bibliothèque LinBox a été faite au travers d'un *wrapper* générique appelé `UnparametricField`. Ce *wrapper* est une classe générique non paramétrée permettant de synchroniser les types de données munis des opérateurs (+,-,×,/) avec l'archétype des corps finis de LinBox. Afin de proposer un domaine de calcul paramétrable pour ce *wrapper*, nous avons développé les classes `NTL_zz_p` et `NTL_ZZ_p` qui surchargent le *wrapper* générique pour les types `NTL::zz_p` et `NTL::ZZ_p`.

2.2.6 Performances et surcoût des *wrappers*

La plupart des implantations de la bibliothèque LinBox proviennent de bibliothèques externes et sont intégrées au travers de *wrappers*. Dans un premier temps, nous évaluons quel est l'impact de ces wrappers sur les performances des implantations externes. Pour cela, nous comparons les performances des corps finis à partir de codes définis à la fois sur les *wrappers* et sur les implantations directes des bibliothèques. À l'instar des tests de la section 2.1.4, nous utilisons le produit scalaire et l'élimination de Gauss pour évaluer le surcoût des *wrappers*. Nous nous appuyons sur 100000 itérations de produits scalaire d'ordre 1000 et sur le calcul du rang d'une matrice d'ordre 500.

Les tables 2.5 et 2.6 illustrent les temps de calcul de ces applications en fonction du type

[27] http ://www.swox.com/gmp/

de l'implantation (*wrapper*/directe). Pour chacune des applications, nous exprimons le surcoût relatif entre l'utilisation directe et l'utilisation à partir des wrappers.

Les résultats obtenus montrent que l'utilisation des *wrappers* ne pénalise pas les implantations des bibliothèques externes. Cela vient du fait que les fonctions de *wrappers* ne sont généralement que des appels de fonction des bibliothèques externes. De ce fait, les méthodes d'*inlining* des compilateurs permettent de supprimer ces appels. Typiquement, le *wrapper* `GivaroZpz<Log16>` de l'implantation logarithmique de la bibliothèque Givaro entraîne un surcoût nul sur l'Itanium. Par contre, ce même *wrapper* entraîne un surcoût entre -10% et 3% sur le Pentium III. En pratique, on observe un surcoût moyen quasiment nul bien que dans certains cas les *wrappers* permettent d'obtenir de meilleures performances que l'implantation d'origine. Ces améliorations de performances sont reproductibles mais nous ne savons déduire aucune explication concrète si ce n'est que le compilateur traite les codes de façon différente et que les effets de caches peuvent être différents.

	Produit scalaire			*Élimination de Gauss*		
	direct	**wrapper**	**surcoût**	**direct**	**wrapper**	**surcoût**
GivaroZpz<Std32>	4.18s	4.13s	-2%	2.33s	2.43s	4%
GivaroZpz<Log16>	1.51s	1.36s	-10%	1.04s	1.08s	3%
GivaroMontg	2.85s	2.75s	-4%	2.18s	2.19s	0%
NTL_zz_p	19.53s	21.44s	9%	9.20s	9.79s	6%
NTL_ZZ_p	82.61s	88.17s	6%	50.92s	49.87s	-3%

TAB. 2.5 – Surcoût des wrappers sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`).

	Produit scalaire			*Élimination de Gauss*		
	direct	**wrapper**	**surcoût**	**direct**	**wrapper**	**surcoût**
GivaroZpz<Std32>	14.16s	15.25s	7%	6.42s	6.41s	-1%
GivaroZpz<Log16>	3.71s	3.72s	0%	1.93s	1.93s	0%
GivaroMontg	11.28s	11.28s	0%	5.18s	4.96s	-5%
NTL_zz_p	10.30s	10.42s	1%	5.97s	6.24s	4%
NTL_ZZ_p	190.87s	190.34s	-1%	102.56s	100.30s	-3%

TAB. 2.6 – Surcoût des wrappers sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`).

Nous nous intéressons maintenant aux performances des implantations. Pour cela, nous étudions les performances des opérations arithmétiques de base pour chacune des implantations. Afin de mesurer les performances de ces opérations atomiques, nous appliquons les opérateurs sur deux vecteurs de données d'ordre 10000. Nous effectuons dix mille fois ce calcul pour atteindre des temps supérieurs à la seconde. Les performances sont exprimées en terme d'opérations arithmétiques du corps premier par seconde. Nous utilisons les termes Mops et Kops pour définir respectivement un million et un millier d'opérations du corps premier par seconde. Nous séparons les implantations ayant une précision finie de celles basées sur des entiers multiprécision car leur comparaison ne serait pas pertinente.

Les implantations proposant une précision finie sont toutes basées sur des opérations machine

flottantes ou entières. Les différences proviennent de la taille des corps premiers possibles et de la façon dont les calculs sont effectués. Afin de comparer l'ensemble de ces implantations, nous utilisons le corps premier $\mathbb{Z}_{32749}$. Dès lors qu'on utilise des corps plus grands, les performances obtenues passent à l'échelle à condition que les implantations autorisent la précision utilisée.

Les figures 2.8, 2.9 et 2.10 illustrent les performances obtenues sur trois architectures différentes, respectivement un Pentium III, un Itanium et un Xeon. Les programmes ont été compilés avec l'option de compilation `-O2` et le compilateur gcc version 3.0.4 pour le Pentium III et le Xeon et gcc version 3.3.4 pour l'Itanium. Les implantations de corps finis utilisées dans ces figures sont classées par précision décroissante.

- (a) `Modular<uint32>` : **31 bits**
- (b) `NTL_zz_p` : **30 bits**
- (c) `Modular<int32>` : **30 bits**
- (d) `Modular<double>` : **26 bits**
- (e) `GivaroZpz<Std32>` : **16 bits**
- (f) `GivaroZpzMontg` : **15 bits**
- (g) `GivaroZpz<Log16>` : **15 bits**

Dans un premier temps, nous voyons que l'utilisation de la représentation logarithmique (i.e. `GivaroZpz<log16>`) permet d'obtenir les meilleures performances pour la multiplication et la division sur les trois architectures. En effet, ces opérations sont effectuées par des additions/-soustractions et des lectures de table, ce qui est en pratique moins coûteux que la multiplication modulaire ou l'algorithme d'Euclide étendu. Du fait de la représentation logarithmique, les opérations d'addition et de soustraction sont moins performantes que pour les représentations classiques. Cela provient en particulier des lectures de table pour calculer le successeur ou le prédécesseur. On remarque d'ailleurs que la différence des performances est nettement plus marquée sur les architectures ayant peu de mémoire cache (voir figure 2.8 et 2.9). Toutefois, la combinaison d'une addition et d'une multiplication (opération `axpy`) reste efficace et permet d'obtenir globalement les meilleures performances.

On remarque aussi que l'algorithme de Montgomery utilisé pour la multiplication dans `GivaroZpzMontg` permet d'obtenir pour le Pentium III et le Xeon la meilleure performance après la représentation logarithmique. Pour le Pentium III, cela permet même d'obtenir les meilleures performances pour l'opération `axpy`. Néanmoins, les gains de taille sur les corps finis possibles n'est que minime en comparaisons de la différence de performances pour la multiplication.

Dès lors que l'on s'intéresse à une précision supérieure à 16 bits, c'est l'implantation à base de flottants double précisions, à savoir `Modular<double>`, qui s'avère être la plus performante pour l'opération de multiplication. Cela s'explique par les meilleures performances des multiplieurs flottants par rapport aux multiplieurs entiers [20]. Néanmoins, les performances obtenues sur l'Itanium, pour cette implantation, ne sont pas vraiment celles que nous attendions. Il semble que l'utilisation de la fonction `fmod` pour effectuer la réduction modulaire entraîne un surcoût très important pour cette opération.

Si l'on s'intéresse à une précisions un peu plus grande que 26 bits, on s'aperçoit que les implantations `Modular<uint32>` et `Modular<int32>` offrent de bien meilleures performances que l'implantation `NTL_zz_p`. Il n'y a que pour l'opération de multiplication sur l'Itanium que l'implantation `NTL_zz_p` tire parti des performances des multiplieurs flottants.

Les tables 2.8, 2.9 et 2.10 illustrent les performances des implantations de corps finis utilisant des entiers multiprécision. Nous utilisons le corps premier $\mathbb{Z}_p$ avec un nombre premier p codé sur 128 bits, $p = 340282366920938463463374607431768211507$. Les deux implantations

que nous comparons sont toutes les deux basées sur la bibliothèque GMP. On remarque que les performances de l'implantation provenant de la bibliothèque NTL, à savoir `NTL_ZZ_p`, est toujours plus performante que celle provenant de la bibliothèque LinBox à partir de la classe `Modular<integer>`. Ces différences de performances s'expliquent par le fait que la bibliothèque NTL définit ces entiers multiprécision sur la couche `mpn` de GMP, qui est la couche bas niveau, alors que l'implantation `Modular<integer>` utilise une interface C++ basée sur la couche `mpz` de GMP, qui est une surcouche de `mpn`. En particulier, ces différences sont non négligeables pour les opérations d'addition et de soustraction. En outre, cela représente en moyenne entre 30% et 100% de surcoût pour ces opérations. Par contre, l'opération de multiplication est moins pénalisée du fait que les performances sont plus reliées aux algorithmes sous-jacents qui sont identiques dans notre cas. Le surcoût moyen est ici inférieur à 20%. En revanche, l'opération de division de l'implantation `NTL_ZZ_p` est pratiquement 10 fois plus performante que celle de l'implantation `Modular<integer>`. Cette différence provient de l'implantation de l'inverse modulaire. Grâce à la routine GMP `mpn_gcdex`, ce calcul est fait à bas niveau pour l'implantation `NTL_ZZ_p`. Par contre, la classe `Modular<integer>` implante cette opération à partir de l'algorithme d'Euclide étendu partiel [21, algorithme 1.19, page 27] sur l'interface C++ des entiers GMP. Notre choix s'est porté sur une implantation haut niveau car il n'existe pas de version de l'algorithme d'Euclide étendu partiel (un seul des cofacteurs est calculé) pour la couche `mpz` de GMP.

En conclusion de cette partie, on peut dire que le choix de l'implantation de corps premier est vraiment dépendant de la taille du corps premier utilisée. En particulier, on utilisera l'implantation logarithmique `GivaroZpz<Log16>` pour des petits corps (p inférieur à 2^{15}). Si l'on a besoin de corps premier de l'ordre du mot machine, on utilisera une implantation classique à base d'entier ou de flottant (i.e. `Modular<uint32>` ou `Modular<double>`). En revanche, si la taille du corps dépasse le mot machine, on utilisera l'implantation `NTL_ZZ_p` qui permet de tirer parti des implantations bas niveaux de la bibliothèque GMP.

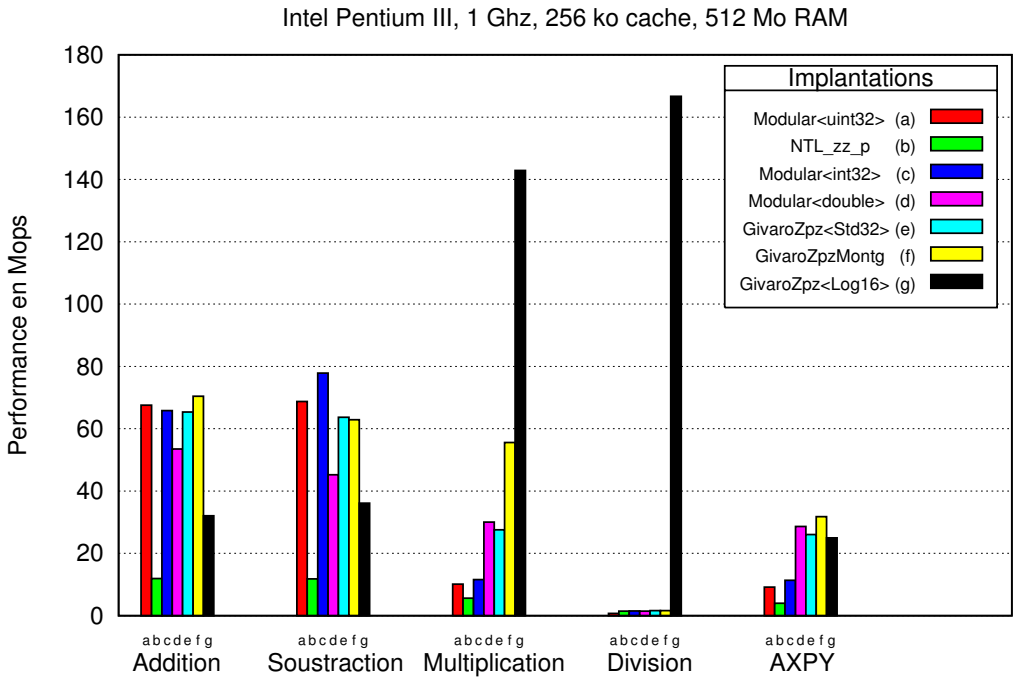

FIG. 2.8 – Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`P3-1Ghz`).

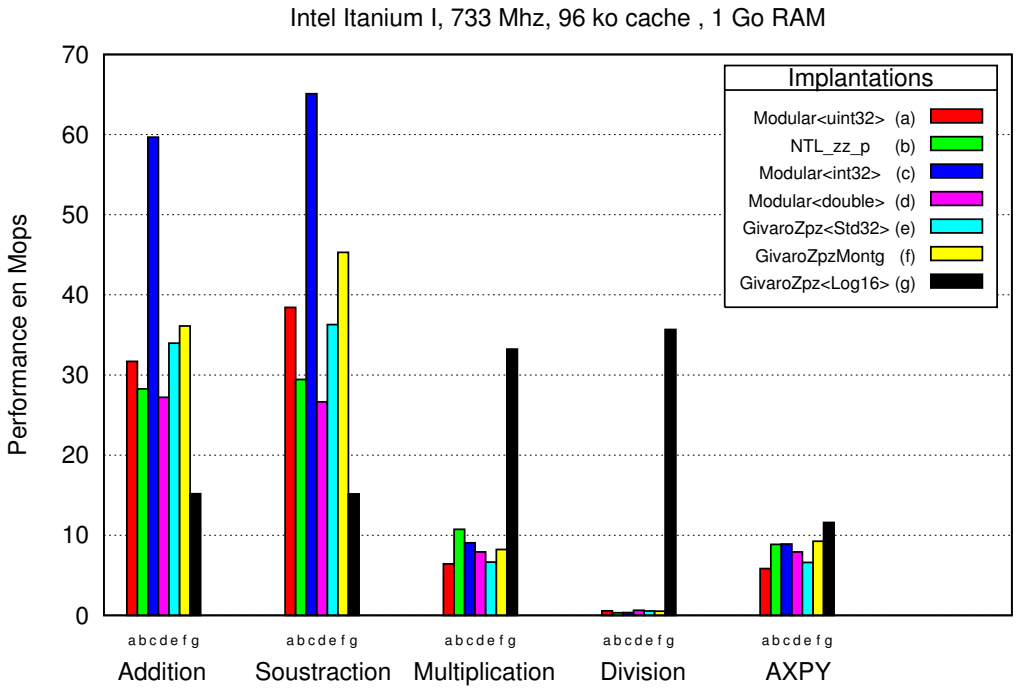

FIG. 2.9 – Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`IA64-733Mhz`).

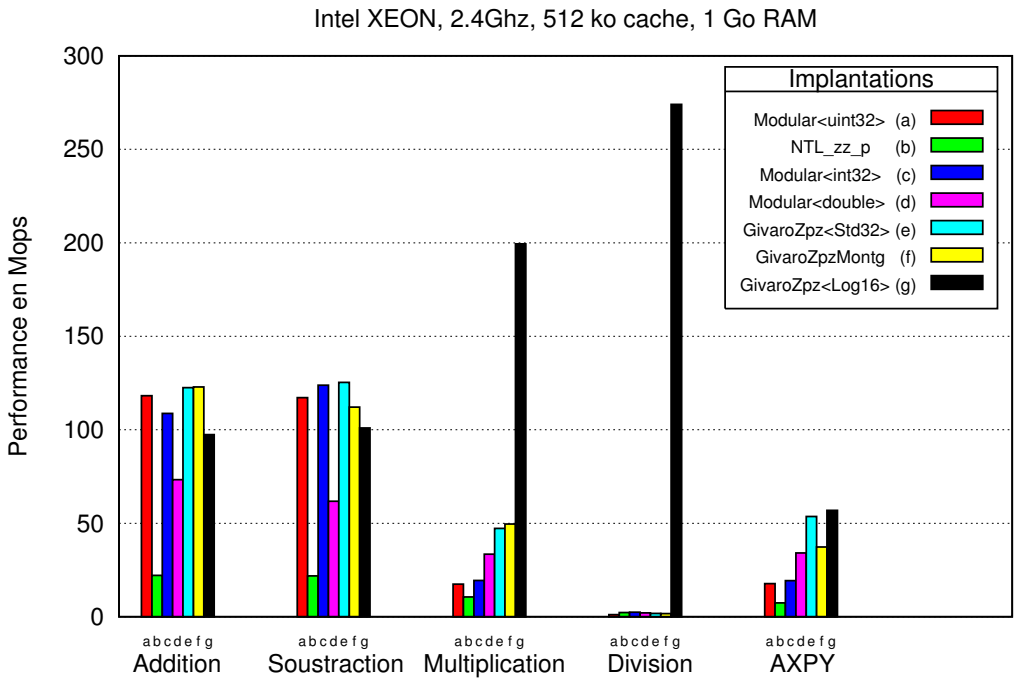

FIG. 2.10 – Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`Xeon-2.66Ghz`).

	Addition	Soustraction	Multiplication	Division	AXPY
Modular<integer>	862	833	529	7	290
NTL_ZZ_p	1754	1785	546	60	366

TAB. 2.7 – Performance (en Kops) pour un corps premier de taille 128 bits (`P3-1Ghz`).

	Addition	Soustraction	Multiplication	Division	AXPY
Modular<integer>	610	532	311	5	202
NTL_ZZ_p	953	907	380	40	269

TAB. 2.8 – Performance (en Kops) pour un corps premier de taille 128 bits (`IA64-733Mhz`).

	Addition	Soustraction	Multiplication	Division	AXPY
Modular<integer>	1805	1623	804	13	514
NTL_ZZ_p	2493	2710	938	103	700

TAB. 2.9 – Performance (en Kops) des corps premiers de taille 128 bits (`Xeon-2.66Ghz`).

2.3 Extension algébrique $GF(p^k)$

Une extension algébrique, ou corps de Galois, de type $GF(p^k)$ est une extension d'un corps premier d'ordre p^k. L'extension algébrique du corps premier Z_p de degré k définit une structure d'espace vectoriel sur $\mathbb{Z}_p$ de dimension k. En pratique, pour construire ce type d'espace vectoriel, on utilise le corps de fraction d'un anneau de polynômes [21]. Soit $\mathbb{Z}_p[x]$ l'anneau des polynômes à coefficients dans $\mathbb{Z}_p$ et $P \in \mathbb{Z}_p[x]$ un polynôme irréductible sur $\mathbb{Z}_p$ de degré k, alors $F_q = \mathbb{Z}_p[x]/P$ définit une structure d'extension algébrique de $\mathbb{Z}_p$ à l'ordre p^k.

Le calcul dans ce type d'extension se fait généralement au travers d'une arithmétique polynomiale modulo un polynôme irréductible $P \in \mathbb{Z}_p[x]$. De nombreuses études ont été menées pour trouver des structures de polynômes irréductibles particulières et des corps de base particuliers pour simplifier l'arithmétique modulaire des polynômes [3, 2]. Une alternative possible à l'arithmétique polynomiale est d'utiliser une représentation logarithmique des éléments au travers d'une racine primitive de l'extension. Les calculs se ramènent alors à des calculs entiers sur les puissances. Toutefois, cette méthode ne permet pas de manipuler de grandes extensions car elle nécessite d'importantes ressources mémoire.

Dans cette section nous présentons les différentes implantations d'extensions de corps finis disponibles au sein de la bibliothèques LinBox. Le but de nos travaux est de réutiliser le plus efficacement possible les implantations provenant d'autres bibliothèques et de les rendre disponibles au travers de l'archétype des corps finis.

2.3.1 Givaro

La bibliothèque Givaro[28] propose une implantation d'extension algébrique basée sur une représentation logarithmique des éléments au travers des puissances d'un générateur de l'extension. Comme pour les corps premiers, l'ensemble des éléments inversibles d'une extension algébrique définit un groupe multiplicatif cyclique. La notion de générateur est ici encore utilisée pour représenter les éléments au travers d'une puissance. A la différence des corps premier, pour calculer ce générateur il faut tout d'abord définir la structure de l'extension.

L'implantation développée dans la bibliothèque Givaro, appelée `GFqDom`, s'appuie sur la théorie des polynômes cyclotomiques [21, §8.2, page 200] , [36, §14.10, page 402] pour trouver des polynômes irréductibles. Cette implantation recherche un polynôme irréductible possédant une racine primitive simple pour définir le générateur du corps [25, §3.3], typiquement le monôme x. Grâce à ce générateur, la construction des tables de successeurs et de conversions entre les éléments et les puissances du générateur est très rapide, jusqu'à 8.5 fois plus rapide que pour un générateur classique [25, figure 3.4]. L'arithmétique utilisée dans cette implantation est similaire à celle présentée dans la section 2.2.2 pour l'implantation `GivaroZpz<Log16>`. L'utilisation de tables pour éviter la gestion des exceptions de calcul dues au zéro du corps est ici remplacée par des comparaisons car les ressources mémoire nécessaires limitent d'autant plus la taille des extensions possibles. Malgré tout, l'implantation `GivGfq` demande de stocker 4 tables de $p^k - 1$ valeurs pour une extension $GF(p^k)$: 2 tables de conversion (puissance, valeur), une table des successeurs et une table des prédécesseurs. Par exemple, la définition de l'extension $GF(3^{15})$ à partir de cette implantation entraînerait l'allocation de 230 Mo rien que pour les tables.

De la même manière que pour les implantations de corps premiers, l'intégration de l'implantation `GFqDom` dans LinBox est immédiate. La classe définissant le `wrapper` LinBox pour cette implantation est la classe `GivaroGfq`.

[28] http://www-lmc.imag.fr/Logiciels/givaro/

2.3.2 NTL

La bibliothèque NTL[29] propose des implantations d'extensions algébriques basées sur une représentation polynomiale et une arithmétique modulaire. À l'instar des implantations de corps premiers pour le modulo, le polynôme irréductible définissant la structure de l'extension est stocké dans une variable globale. Ainsi, les opérations sur les éléments de l'extension s'effectuent à partir d'opérations modulaires définies par cette variable globale. La bibliothèque propose trois implantations d'extensions algébriques. La première est spécifique aux extensions en caractéristique 2 et utilise des tableaux de mots machine pour représenter les éléments de l'extension. Les deux autres utilisent des polynômes à coefficients dans des corps finis à précision finie (NTL::zz_pE) ou à précision arbitraire (NTL::ZZ_pE). Le schéma standard de construction de ces extensions consiste dans un premier temps à définir le corps de base de l'extension puis de définir le polynôme irréductible qui définit la structure de l'extension. La bibliothèque NTL propose pour chaque type de corps fini un module de polynôme associé qui permet de construire des polynômes irréductibles.

Par exemple, pour définir l'extension $GF(7^{10})$ il faut :
- construire le corps premier $\mathbb{Z}_7$:
  `NTL::zz_p::init(7) ;`
- définir un polynôme irréductible de degré 10 sur $\mathbb{Z}_7$:
  `NTL::zz_pX irredPoly = BuildIrred_zz_pX (k) ;`
- construire l'extension $GF(7^{10})$:
  `NTL::zz_pE::init(irredPoly) ;`

La nomenclature des types de données et des fonctions dans la bibliothèque NTL est standardisée. De ce fait, il suffit de remplacer le corps premier à précision finie `zz_p` par le corps fini à précision arbitraire `ZZ_p` dans l'exemple précédent pour définir une extension algébrique autorisant des caractéristiques supérieures au mot machine. Les noms des fonctions doivent être ajustés en fonction du type de corps de base utilisé. Pour les extensions en caractéristique 2, le principe reste le même avec le corps `GF2`. Néanmoins, la construction de l'extension est un peu différente car le corps de base est connu à l'avance, à savoir $\mathbb{Z}_2$. De ce fait, l'utilisation d'un polynôme irréductible creux (trinômes, pentanômes) permet d'améliorer la réduction modulaire. La fonction `BuildSparseIrred_GF2X(long n)` permet de générer de tels polynômes. En particulier, pour des degrés inférieurs à 2048 le polynôme est récupéré dans une table alors que pour des degrés supérieurs, le polynôme est généré en récupérant le premier trinôme ou pentanôme irréductible à partir d'une recherche exhaustive.

Comme pour les corps finis (voir §2.2.5), l'intégration de ces extensions algébriques dans LinBox est effectuée à partir du *wrapper* générique `UnparametricField`. Néanmoins, l'utilisation directe de ce *wrapper* nécessite que l'utilisateur instancie lui même le polynôme irréductible définissant l'extension. Afin de proposer, une construction simplifiée, nous avons développé des classes spécialisant ce *wrapper* pour les types `NTL::zz_pE, NTL::ZZ_pE, NTL::GF2E` . Le code suivant illustre la classe utilisée pour les extensions en caractéristique 2.

```
class NTL_GF2E : public UnparametricField<NTL::GF2E>
{
```

[29] http::/www.shoup.net/ntl

```
  public:
    NTL_GF2E (const integer &p, const integer &k) {
    if(p != 2)
      throw PreconditionFailed(__FUNCTION__,__LINE__,"modulus must be 2");
    NTL::GF2X irredPoly = BuildSparseIrred_GF2X((long) k);
    NTL::GF2E::init(irredPoly);
 }
};
```

2.3.3 LiDIA

La bibliothèque LiDIA[30] propose des implantations de corps finis disponibles au travers d'une seule et même interface. Le but de cette interface est de faciliter la manipulation des corps finis à partir d'un unique type de données, à savoir `galois_field`. Plusieurs implantations de corps finis sont proposées en fonction de leur structure. On trouve des spécialisations pour les corps premiers, les extensions algébriques de caractéristique quelconque et les extensions en caractéristique 2, chacune utilisant sa propre arithmétique. L'arithmétique des corps premiers est implantée au travers d'une arithmétique modulaire sur les entiers multiprécision de LiDIA (`bigint`) qui peuvent être paramétrés dans le noyau de la bibliothèque (voir §1.2.4). L'arithmétique des extensions algébriques est basée sur une arithmétique polynomiale modulaire avec un polynôme irréductible unitaire. Les coefficients sont définis dans un corps fini qui peut être lui aussi une extension algébrique, ce qui permet de pouvoir facilement définir des tours d'extensions. Dans le cas des extensions en caractéristique 2, le choix du polynôme irréductible se fait à partir d'une base de polynômes irréductibles très creux (trinôme, quadrinôme, pentanôme) stockés dans un fichier. Ce type de construction est utilisé pour des extensions ayant un degrés inférieur ou égal à trois mille. La représentation des éléments est basée sur un tableau dynamique d'entiers machine non signé (`unsigned long`) permettant de faire les additions et les soustractions uniquement à partir de "ou exclusif" (`xor`).

La définition des corps finis se fait directement au travers de l'interface qui gère le choix de la représentation. Par exemple, on peut définir les trois corps finis $\mathbb{Z}_{17}$, $GF(2^{10})$ et $GF(7^7)$ de la façon suivante :

```
LiDIA::galois_field K1(17,1);
LiDIA::galois_field K2(2,10);
LiDIA::galois_field K3(7,7);
```

Chacun de ces corps possède une représentation interne différente. Afin de récupérer les fonctions correspondantes à la représentation sous-jacente, l'interface définit l'ensemble des fonctions du corps finis à partir de pointeurs. Lors de la création du corps fini, ces pointeurs sont initialisés sur les adresses des fonctions correspondantes à la représentation utilisée. Le même type d'interface est utilisée pour définir les éléments au travers d'un type de données, à savoir `gf_element`. Chaque élément stocke un pointeur sur l'interface des corps finis, ce qui lui permet de récupérer les fonctions correspondantes à son corps fini d'appartenance.

[30] http://www.informatik.tu-darmstadt.de/TI/LiDIA/

La représentation des éléments est définie dans un premier temps sur pointeur `void*` instancié uniquement lors du rattachement à un corps fini particulier. Ainsi, la construction des éléments `gf_element a1(K1) ; gf_element a2(K2) ; gf_element a3(K3) ;` permet de définir des éléments sur respectivement $\mathbb{Z}_{17}$, $GF(2^{10})$ et $GF(7^7)$. Le calcul sur des éléments n'étant rattachés à aucun corps entraînera la levée d'une exception.

L'utilisation de ces interfaces permet de définir des codes simples bénéficiant de la meilleure représentation possible en fonction de la caractéristique et de la cardinalité du corps fini utilisé. L'exemple suivant illustre l'utilisation de ces interfaces.

```
LiDIA::galois_field K(17,3);
LiDIA::gf_element a(K),b(K),c(K);
LiDIA::bigint tmp=10;
a.assign(tmp);
b.assign(tmp);
multiply(c,a,b);
```

Cet exemple calcule le produit de deux éléments dans `GF`(17^3). L'appel de la fonction multiplication au travers de l'interface entraîne les appels suivants :

- récupération de l'adresse de la fonction `multiply` à partir du corps fini attaché aux opérandes
- transtypage de la représentation `void*` de a, b, c sur la représentation dans K
- appel de la fonction `multiply` sur le bon type de corps et d'éléments.

L'arithmétique de corps finis proposé par LiDIA s'appuie directement sur des opérations au travers des éléments et ne propose pas de domaine de calcul. L'intégration de ces implantations à la bibliothèque LinBox consiste à développer un domaine de calcul autour des opérations proposées par LiDIA. En particulier, nous avons développé la classe `LidiaGfq` permettant de manipuler les corps finis de LiDIA au travers d'un domaine de calcul compatible avec l'archétype de LinBox. L'intégration des opérations arithmétiques sur les éléments est immédiate du fait qu'elles sont accédées directement à partir des éléments (via un pointeur sur l'interface des corps finis). Par exemple, la définition de la fonction de multiplication de ce wrapper consiste uniquement à appeler la fonction `multiply` de LiDIA.

```
gf_element &LinBox::LidiaGfq::mul(gf_element &x,
                                  const gf_element &y,
                                  const gf_element &z) const {
  LiDIA::multiply(x,y,z);
  return x;
}
```

De même, la construction du corps fini est faite directement par le constructeur de la classe `galois_field`. Nous utilisons la fonction `init` définie par l'archétype LinBox pour rattacher les éléments `gf_element` au type de corps finis instancié par le domaine de calcul `LidiaGfq`.

Bien que l'interface des corps finis proposée par la bibliothèque LiDIA soit intéressante, l'utilisation des pointeurs pénalise les performances des implantations. L'utilisation directe des implantations nous permettrait d'obtenir de meilleures performances mais les dépendances entre

les implantations et l'interface ne permettent pas une gestion simple des codes. Nous envisageons néanmoins de proposer une version de ces implantations dans LinBox sans utiliser l'interface.

2.3.4 Performances et surcoût des *wrappers*

L'ensemble des implantations d'extensions algébriques disponibles dans la bibliothèque LinBox proviennent de bibliothèques externes. Comme pour les corps premiers, nous nous intéressons dans un premier temps à évaluer le surcoût des wrappers utilisés pour intégrer ces codes au sein d'un modèle de données compatible avec l'archétype. Afin de calculer le surcoût des wrappers en fonction des implantations sous-jacentes, nous effectuons des tests similaires à ceux de la section 2.2.6. Plus précisément, nous comparons les performances des implantations utilisées directement et utilisées à partir des *wrappers*. Pour cela, nous utilisons ici 10000 itérations de produits scalaire d'ordre 1000 et nous calculons le rang d'une matrice d'ordre 500 à partir d'une élimination de Gauss.

Les tables 2.10 et 2.11 illustrent les résultats de ces tests. Ici encore, on remarque que les wrappers ne pénalisent pas ou très peu les performances des implantations des bibliothèques d'origine. On observe que les différents surcoûts oscillent entre -8% et 13%. Comme pour les corps finis, les méthode d'*inlining* des compilateurs permettent d'absorber les différents appels de fonction effectuées par les *wrappers*. On retrouve néanmoins les effets de bord qui font que les wrappers autorisent parfois de meilleurs temps de calcul que l'implantation initiale. Nous ne savons déduire aucune interprétation de ces effets de bord, certainement dûs au compilateur et au chargement des codes dans les caches instructions.

	Produit scalaire			*Élimination de Gauss*		
	direct	**wrapper**	**surcoût**	**direct**	**wrapper**	**surcoût**
GivaroGfq, `GF`(3^7)	0.46s	0.52s	13%	2.20s	2.28s	3%
LiDIAGfq, `GF`(3^7)	319.97s	323.52s	1%	1445.18s	1523.92s	5%
NTL_zz_pE, `GF`(3^7)	77.97s	80.3s	−5%	41.30s	40.89s	−1%
NTL_ZZ_pE, `GF`(3^7)	287.68s	288.19s	0%	92.55s	87.57s	−6%
NTL_GF2E, `GF`(2^{512})	133.80s	134.18s	0%	17.02s	17.61s	3%

TAB. 2.10 – Surcoût des wrappers des extensions algébriques (`P3-1Ghz`).

	Produit scalaire			*Élimination de Gauss*		
	direct	**wrapper**	**surcoût**	**direct**	**wrapper**	**surcoût**
GivaroGfq, `GF`(3^7)	0.51s	0.51s	−1%	1.51s	1.39s	−8%
LiDIAGfq, `GF`(3^7)	505.40s	490.25s	−3%	2455.33s	2659.99s	8%
NTL_zz_pE, `GF`(3^7)	101.61s	100.41s	−2%	63.78s	63.14s	−2%
NTL_ZZ_pE, `GF`(3^7)	471.78s	475.56s	1%	157.95s	157.36s	−1%
NTL_GF2E, `GF`(2^{512})	87.75s	88.81s	1%	27.75s	27.63s	−1%

TAB. 2.11 – Surcoût des wrappers des extensions algébriques (`IA64-733Mhz`).

Nous nous intéressons maintenant à évaluer les performances intrinsèques des implantations

d'extensions algébriques proposées par les différentes bibliothèques que nous utilisons. Dans un premier temps, nous comparons les extensions en caractéristiques 2. En particulier, nous utilisons l'extension `GF`(2^{512}). Cela concerne plus précisément les implantations à partir des *wrappers* `NTL_GF2E` et `LiDIAGfq`. Les tables 2.12 et 2.13 illustrent les performances obtenues pour chacune des opérations arithmétiques de base ainsi que pour l'opération `axpy`. Pour évaluer correctement les différents coûts atomiques de ces opérations, nous effectuons 1000 fois le calcul de ces opérations sur un vecteur de données d'ordre 1000.

On remarque tout d'abord que les performances pour l'addition et la soustraction sont toujours meilleures pour l'implantation `LiDIAGfq`. Cela s'explique par le fait que l'implantation développée dans la bibliothèque LiDIA pour ces extensions utilise une représentation des éléments en taille fixe, c'est-à-dire que chaque élément de l'extension est considéré comme un polynôme de degrés 512. Par contre, la bibliothèque NTL propose une implantation pour ces extensions qui utilise une représentation adaptative, au sens que les éléments de l'extension sont représentés par des polynômes à taille variable. Ce qui signifie que les éléments n'ont pas tous la même taille. De ce fait, il faut gérer cela dans les opérations, en comparant les éléments durant l'opération et en réduisant le résultat à partir du monôme non nul de plus haut degré. Toute cette gestion des éléments entraîne un surcoût non négligeable par rapport à l'implantation `LiDIAGfq` qui effectue les opérations sur tous les coefficients, même ceux non significatifs. Cette méthode est particulièrement plus efficace du fait que les opérations utilisées sont simplement des "ou exclusif" binaires.

En revanche, pour les opérations de multiplication et de division qui ne sont pas linéaires et de surcroît plus complexes, l'utilisation d'une représentation fixe ne permet pas d'obtenir les meilleures performances. En effet, plus les éléments sont petit moins les opérations sont coûteuses. L'approche utilisée par la bibliothèque `NTL` pour une gestion d'éléments à taille variable prend donc tout son intérêt. Par ailleurs, on peut voir que l'implantation `NTL_GF2E` reste de loin la plus performante pour l'opération `axpy` qui combine une addition et une multiplication.

	Addition	Soustraction	Multiplication	Division	AXPY
NTL_GF2E	813	862	25	2	25
LiDIAGfq	1439	1527	17	1	16

Tab. 2.12 – Performance (en Kops) pour l'extension algébrique `GF`(2^{512}) (`P3-1Ghz`).

	Addition	Soustraction	Multiplication	Division	AXPY
NTL_GF2E	2991	2973	56	4	56
LiDIAGfq	8269	9530	19	2	18

Tab. 2.13 – Performance (en Kops) pour l'extension algébrique `GF`(2^{512}) (`Xeon-2.66Ghz`).

Nous évaluons maintenant les performances des extensions en caractéristique quelconque. Nous ne comparons pas l'implantation `GivaroGfq` d'une part parce qu'elle ne permet pas d'obtenir des extensions de grande taille et d'autre part parce que les performances obtenues pour de petites extensions sont largement supérieures à celle obtenues à partir d'arithmétique polynomiale comme l'attestent les tables 2.10 et 2.11.

Les tables 2.14 et 2.15 illustrent les performances obtenues pour les opérations arithmétiques de base et l'opération `axpy` pour l'extension de corps fini `GF`(17^{100}).

On remarque d'après ces tables que l'implantation `NTL_zz_pE` est toujours la plus perfor-

mante alors que les deux autres restent bien en deçà. Cela s'explique essentiellement par le fait que l'implantation `NTL_zz_pE` utilise un représentation des éléments de l'extension à partir de polynôme à coefficients sur des entiers machine alors que les deux autres utilisent des coefficients à base d'entiers multiprécisions ; ce qui signifie que les opérations de base de l'arithmétique polynomiale sont d'emblée plus coûteuses, et donc les performances sont moins bonnes.

	Addition	Soustraction	Multiplication	Division	AXPY
LiDIAGfq	43Kops	50Kops	648ops	60ops	581ops
NTL_zz_pE	153Kops	181Kops	968ops	249ops	960ops
NTL_ZZ_pE	24Kops	26ops	549ops	78ops	615ops

TAB. 2.14 – Performance pour l'extension algébrique `GF`(17^{100}) (`P3-1Ghz`).

	Addition	Soustraction	Multiplication	Division	AXPY
LiDIAGfq	169Kops	215Kops	1038ops	114ops	922ops
NTL_zz_pE	488Kops	526Kops	1422ops	386ops	1420ops
NTL_ZZ_pE	65Kops	71Kops	882ops	141ops	873ops

TAB. 2.15 – Performance pour l'extension algébrique `GF`(17^{100}) (`Xeon-2.66Ghz`).

2.4 Conclusion

En conclusion de ce chapitre, nous avons montré que l'utilisation de mécanismes d'abstractions pour l'arithmétique des corps finis à partir d'un archétype de données permet le développement de codes robustes, grâce à l'utilisation de codes compilés, et facilite l'intégration de codes provenant de bibliothèques externes, notamment grâce à l'utilisation de *wrappers*.

Les différents surcoûts apportés par l'utilisation de ces mécanismes permettent de conserver les performances intrinsèques des implantations sous-jacentes. En particulier, l'utilisation de ce type de mécanisme doit permettre de comparer à priori, sans avoir à effectuer un nouveau développement, si les implantations d'arithmétiques sont satisfaisantes pour un problème donnée. De plus, nous avons pu mettre en avant les différences des implantations de corps fini déjà existantes. Ce qui nous a permis de proposer un choix d'arithmétique de corps finis efficace pour la bibliothèque LinBox.

Lorsque les opérations sont regroupées dans des calculs sur des blocs de données l'utilisation des opérations atomiques du corps fini ne permet pas d'obtenir les meilleures performances. En effet, l'utilisation d'opérations par blocs spécifiques permet une meilleure gestion des réductions dans le corps. En particulier, l'opération de multiplication de matrices peut être effectuer d'abord de façon exacte sur les entiers puis être suivie d'une réduction dans le corps fini. Grâce à cela, on arrive à diminuer le nombre de réductions dans le corps ce qui permet d'accroître considérablement les performances. Dans le chapitre suivant nous nous intéressons à utiliser de telles opérations pour améliorer les performances des algorithmes d'algèbre linéaire dense sur un corps fini. En particulier, nous nous intéressons à la réutilisation des routines numériques BLAS pour obtenir des implantations portables et très performantes.

Chapitre 3

Algèbre linéaire dense sur un corps fini

Sommaire

3.1 Systèmes linéaires triangulaires 69
3.1.1 Algorithme récursif par blocs 70
3.1.2 Utilisation de la routine "dtrsm" des BLAS 71
3.1.3 Utilisation de réductions modulaires retardées 74
3.1.4 Comparaison des implantations 75
3.2 Triangularisations de matrices 78
3.2.1 Factorisation LSP 78
3.2.2 LUdivine 80
3.2.3 LQUP 81
3.2.4 Performances et comparaisons 81
3.3 Applications des triangularisations 83
3.3.1 Rang et déterminant 83
3.3.2 Inverse 84
3.3.3 Base du noyau 85
3.3.4 Alternative à la factorisation LQUP : Gauss-Jordan 85
3.4 Interfaces pour le calcul "exact/numérique" 89
3.4.1 Interface avec les BLAS 90
3.4.2 Connections avec MAPLE 95
3.4.3 Intégration et utilisation dans LinBox 100

En calcul numérique, les bibliothèques BLAS (§1.2.5) permettent d'obtenir les meilleures performances pour multiplier deux matrices à coefficients flottants. Ces performances sont obtenues grâce à une programmation bas niveau qui permet une meilleure hiérarchisation mémoire des données durant le calcul. La réutilisation des BLAS dans des bibliothèques d'algèbre linéaire numérique comme LAPACK permet d'offrir une puissance de calcul exceptionnelle pour les algorithmes classiques de l'algèbre linéaire (déterminant, systèmes linéaires).

L'existence d'outils de calcul exact similaires sur les nombres entiers s'avère être aujourd'hui nécessaire pour résoudre des problèmes provenant d'applications concrètes. Cependant, il n'existe aucun équivalent aux BLAS pour le calcul exact sur des entiers. L'utilisation des BLAS pour le calcul exact est alors une alternative possible. En particulier, le projet FFLAS [28] a montré que cette approche pour la multiplication de matrices dans un corps fini permettait d'atteindre des performances très proches de celles obtenues par les BLAS. Ce projet a aussi mis en avant le fait que l'utilisation d'algorithmes rapides pour la multiplication de matrices comme celui de Strassen [79] (variante de Winograd [36, algorithme 12.1,page 328]) permettait d'obtenir sur les corps finis une amélioration du temps de calcul de 19% par rapport à l'algorithme classique pour des matrices d'ordre 3000 [66].

Les résultats obtenus par le projet FFLAS sont d'un grand intérêt pour le développement d'outils de calcul très performants en algèbre linéaire exacte sur les corps finis. En effet, dans un modèle algébrique l'ensemble des problèmes d'algèbre linéaire se réduisent au produit de matrices [9, chap. 16] et ces réductions sont effectives à partir d'algorithmes. L'utilisation de ces algorithmes permet d'obtenir les meilleurs coûts théoriques actuels pour les problèmes classiques d'algèbre linéaire dense sur un corps fini. C'est le cas par exemple pour le calcul du déterminant, le calcul du rang, le calcul de l'inverse, la résolution de systèmes linéaires, le calcul d'une base du noyau, le calcul du polynôme minimal et du polynôme caractéristique. Ces algorithmes sont aussi très importants car ils interviennent pour résoudre certains problèmes sur les entiers, notamment pour la résolution de systèmes linéaires diophantiens [61] ou certains problèmes en algèbre linéaire polynomiale comme le calcul d'approximants matriciels minimaux [40].

Le développement de routines d'algèbre linéaire basées sur la multiplication de matrices du projet FFLAS et sur les routines numériques BLAS devrait donc permettre de proposer des implantations très efficaces de ces algorithmes et ainsi fournir une boîte à outils performante pour l'implantation algorithmes de plus haut niveau. Nous nous intéressons dans ce chapitre à développer de telles routines.

La plupart des algorithmes se réduisant au produit de matrices sont basés sur une simplification du problème par une triangularisation de matrices. Le calcul de cette triangularisation est alors effectué à partir d'une élimination de Gauss par blocs, où les étapes d'élimination consistent en des résolutions de systèmes linéaires triangulaires matriciels.

Nous proposons dans la première partie de ce chapitre (§3.1) de développer des implantations pour ces résolutions de système qui bénéficient au maximum des performances des routines numériques BLAS. En particulier, nous verrons que l'utilisation de résolutions numériques n'est possible que pour des matrices ayant de faibles dimensions. L'utilisation d'un algorithme récursif par blocs nous permettra alors de profiter du produit de matrices FFLAS pour diminuer la dimension des matrices et ainsi permettre l'utilisation des résolutions numériques. Pour cela, nous définirons une notion de seuil permettant de changer la résolution utilisée (récursive ou numérique). Cependant, l'utilisation des routines numériques entraîne des conversions de données qui peuvent affecter les performances globales de l'implantation. Nous montrons alors que dans certains cas l'utilisation d'un algorithme de résolution "adhoc" et de produits scalaires avec une réduction modulaire retardée permet d'obtenir de meilleures performances qu'avec des

résolutions numériques.

Dans la deuxième partie de ce chapitre (§3.2), nous nous intéressons au calcul des triangularisations de matrices aussi bien dans le cas singulier que non singulier. En particulier, nous montrons que la gestion des calculs et de la mémoire est un facteur important pour obtenir les meilleures performances possibles. Pour cela, nous nous appuierons sur les algorithmes de factorisation LSP/LQUP [46] et nous utiliserons les implantations de résolution de systèmes triangulaires matriciels présentées dans la première partie de ce chapitre ainsi que le produit matriciel des FFLAS.

En utilisant notre implantation pour la triangularisation de matrices nous nous intéressons à définir des implantations pour calculer le rang, le déterminant, l'inverse et une base du noyau (§3.3). Nous montrons l'efficacité de ces implantations en les rapprochant de celles obtenues pour le produit de matrices FFLAS. En particulier, nous verrons que les ratios obtenus en pratique sont très proches de ceux théoriques en nous basant sur les constantes des coûts algorithmiques. Sur un plan plus théorique, nous nous intéressons au cas particulier du calcul d'une base du noyau et nous proposerons un nouvel algorithme permettant de diminuer de $\frac{1}{6}$ le nombre d'opérations arithmétiques nécessaires avec un produit de matrice classique.

Les travaux présentés dans ce chapitre ont été menés en collaboration avec J.-G. Dumas et C. Pernet et ont permis d'aboutir au développement du paquetage FFPACK[31] [29]. Ce paquetage implante de façon générique le calcul du déterminant, du rang, de l'inverse, du noyau, des polynômes minimal et caractéristique pour tout les types de corps premiers compatibles avec le modèle de base de la bibliothèque LinBox (voir §2.1.1) en utilisant au maximum le produit de matrices FFLAS et les routines numériques BLAS.

Les paquetages FFLAS-FFPACK ont été développés en C++ pour bénéficier de la généricité des paramètres *templates*. Néanmoins, ils s'appuient sur une représentation bas niveau des matrices et des vecteurs en utilisant des tableaux de données pour une compatibilité maximale avec les BLAS. Afin de réutiliser plus facilement toutes ces routines, nous proposons de les intégrer dans des domaines de calculs simplifiés, en particulier au sein de calculs interprétés avec le logiciel MAPLE et au sein d'une modélisation objet avec la bibliothèque LinBox. La dernière partie de ce chapitre (§3.4) concerne donc ces intégrations. Plus précisément, nous présentons en détail les implantations disponibles dans les paquetages FFLAS-FFPACK et nous montrons comment nous avons permis leurs utilisations dans des interfaces de plus haut niveau avec MAPLE et LinBox.

3.1 Systèmes linéaires triangulaires

Dans cette section, nous nous intéressons à la résolution de systèmes linéaires triangulaires où le second membre est une matrice. Cela correspond, plus précisément, à la résolution simultanée de plusieurs systèmes linéaires issus d'une même matrice. Soient $A \in \mathbb{Z}_p^{m\times m}$ une matrice triangulaire inversible et $B \in \mathbb{Z}_p^{m\times n}$, on cherche alors à calculer la matrice $X \in \mathbb{Z}_p^{m\times n}$ telle que $AX = B$. Nous considérons sans perte de généralité que $m \leq n$. Ce problème est classique en algèbre linéaire car c'est l'une des opérations de base de l'élimination de Gauss par blocs [79]. L'utilisation d'un algorithme de type "diviser pour régner" pour résoudre ces systèmes permet de ramener la complexité de ce problème à celui du produit de matrices de $\mathbb{Z}_p^{n\times n}$.

Notre objectif consiste à fournir une implantation pour ces résolutions qui s'appuie sur le produit de matrices. Nous présentons dans un premier temps l'algorithme de résolution récursif basé sur la multiplication de matrices et nous précisons son coût arithmétique en fonction du

[31] http ://www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/FFLAS/

produit de matrices. Nous considérons que le coût arithmétique de la multiplication de matrices d'ordre n sur $\mathbb{Z}_p$ est de $O(n^\omega)$ opérations arithmétiques, avec $\omega = 3$ pour l'algorithme classique et $\omega = \log 7$ pour l'algorithme de Strassen [79], où log représente le logarithme en base 2. La plus petite valeur connue pour w est de ≈ 2.37 en utilisant le résultat de Coppersmith-Winograd [18]. Nous définissons le coût des multiplications de matrices rectangulaires $m \times k$ par $k \times n$ à partir de la fonction $R(m, k, n)$. Cette dernière est bornée d'après [45, équation 2.5] par $R(m, k, n) \leq C_\omega \min(m, k, n)^{\omega-2} \max(mk, mn, kn)$; où C_ω représente la constante devant le terme dominant du coût de la multiplication de matrices carrées (i.e. $C_\omega = 2$ pour l'algorithme classique).

En utilisant le paquetage FFLAS pour le produit de matrices, l'implantation de l'algorithme de résolution récursif est direct et permet d'obtenir de très bonnes performances. Néanmoins, nous nous intéresserons à améliorer cette implantation en introduisant soit des résolutions numériques (§3.1.2) soit des produits matrice-vecteur avec une réduction modulaire retardée (§3.1.3) Enfin, dans la dernière partie (§3.1.4), nous illustrerons ces différentes améliorations à partir d'expérimentations utilisant les corps premiers `GivaroZpz<Std32>` et `Modular<double>` présentés dans le chapitre 2 (§2.2.2 et §2.2.1).

3.1.1 Algorithme récursif par blocs

Nous considérons ici sans perte de généralité le cas de matrices triangulaires supérieures avec une résolution de systèmes à gauche. En effet, l'algorithme s'étend facilement aux différentes combinaisons de résolutions (triangulaire supérieure/inférieure, résolution à droite/à gauche). Soit U une matrice triangulaire supérieure, L une matrice triangulaire inférieure et B une matrice rectangulaire ; nous définissons les fonctions de résolution en X suivantes :

$$\begin{array}{lcl} \texttt{ULeft-Trsm} & \rightarrow & UX = B \\ \texttt{URight-Trsm} & \rightarrow & XU = B \\ \texttt{LLeft-Trsm} & \rightarrow & LX = B \\ \texttt{LRight-Trsm} & \rightarrow & XL = B \end{array}$$

L'acronyme `trsm` s'inspire des noms des routines BLAS pour ce type de résolutions et signifie *TRiangular System with Matrix*. Nous utilisons les lettres `U,L` pour spécifier si la matrice du système est triangulaire supérieure ou inférieure et nous utilisons `Left` et `Right` pour préciser le côté de la résolution.

Algorithme `ULeft-Trsm`(A, B)
Entrées : $A = [a_{ij}] \in \mathbb{Z}_p^{m\times m}$, $B \in \mathbb{Z}_p^{m\times n}$
Sortie : $X \in \mathbb{Z}_p^{m\times n}$ tel que $AX = B$
Schéma
si m = 1 **alors**
$X := a_{11}^{-1} \times B$
sinon *(découper A en blocs de dimension $\lfloor \frac{m}{2} \rfloor$ et $\lceil \frac{m}{2} \rceil$)*

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

$X_2 :=$ `ULeft-Trsm`(A_3, B_2) ;
$B_1 := B_1 - A_2 X_2$;
$X_1 :=$ `ULeft-Trsm`(A_1, B_1) ;
retourner X ;

Lemme 3.1.1. *L'algorithme* `ULeft-Trsm` *est correct et son coût est borné par* $\frac{C_\omega}{2(2^{\omega-2}-1)} nm^{\omega-1}$ *opérations arithmétiques dans le corps premier* $\mathbb{Z}_p$ *pour* $m \leq n$ *et* $\omega > 2$.

Preuve. La preuve de cet algorithme se fait par induction sur les lignes du système. Pour cela, il suffit de remarquer que

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \text{ est solution} \Longleftrightarrow \begin{cases} A_3 X_2 = B_2 \\ A_1 X_1 + A_2 X_2 = B_1 \end{cases}$$

Le coût arithmétique de cet algorithme est donné par une expression récursive. Soient $C(m,n)$ le coût de l'algorithme `ULeft-Trsm` où m est la dimension de la matrice U et n le nombre de colonnes de B. D'après le schéma récursif de l'algorithme, on déduit que

$$\begin{aligned} C(m,n) &= 2C(\frac{m}{2},n) + R(\frac{m}{2},\frac{m}{2},n) \\ &= \sum_{i=1}^{\log m} 2^{i-1} R(\frac{m}{2^i},\frac{m}{2^i},n). \end{aligned}$$

Par hypothèse $m \leq n$ et donc, pour tout $i > 0$,

$$R(\frac{m}{2^i},\frac{m}{2^i},n) \leq C_\omega \left(\frac{m}{2^i}\right)^{\omega-1} n.$$

On peut donc borner $C(m,n)$ par

$$C(m,n) \leq \frac{C_\omega n m^{\omega-1}}{2} \sum_{i=1}^{\log m} \left(\frac{1}{2^i}\right)^{\omega-2}.$$

En considérant $\omega > 2$, on obtient la borne de $O(nm^{\omega-1})$ opérations sur $\mathbb{Z}_p$. □

En utilisant la routine de multiplication-addition de matrices `Fgemm` proposée par le paquetage FFLAS, l'implantation de cet algorithme est directe et satisfaisante. En particulier, cette implantation permet d'atteindre 2184 millions d'opérations sur un corps fini par seconde pour la résolution d'un système d'ordre 5000 sur $\mathbb{Z}_{32749}$ avec un Pentium 4, 2.4 Ghz (voir table 3.1). Néanmoins, nous allons voir dans les deux parties suivantes que l'on peut encore améliorer ces performances.

3.1.2 Utilisation de la routine "dtrsm" des BLAS

L'utilisation des routines numériques BLAS a permis de réduire considérablement les temps de calcul pour la multiplication de matrices sur les corps finis [28, 66]. L'idée consiste à convertir les matrices dans un format flottant double précision, calculer la multiplication avec les BLAS et convertir le résultat dans la représentation des éléments du corps fini. Cette méthode est possible du fait que la valeur maximale des données intervenant dans le calcul est linéaire en fonction de la dimension des matrices. Pour des matrices d'ordre n sur $\mathbb{Z}_p$ et des nombres flottants double précision, l'utilisation des BLAS est possible si l'équation suivante est satisfaite [28] :

$$n(p-1)^2 < 2^{53}.$$

Notre idée consiste à utiliser la même approche pour la résolution de systèmes linéaires triangulaires matriciels. Cependant, l'utilisation directe des résolutions numériques est moins

évidente. Premièrement, la valeur maximale des données durant le calcul est exponentielle en la dimension du système. Deuxièmement, la solution du système est une solution rationnelle.

Du fait de la taille des valeurs calculées, l'utilisation directe des BLAS est ici impossible. Par exemple, le résultat entier d'un système d'ordre 100 à coefficients entiers inférieurs à 1009 possède de l'ordre de 1000 bits. Afin de pouvoir utiliser les routines de résolution des BLAS, nous utilisons la récursivité de l'algorithme `ULeft-Trsm` pour faire diminuer la dimension des systèmes jusqu'à ce qu'ils soient suffisamment petits pour être résolus numériquement. Pour gérer les solutions rationnelles, nous décomposons le système de telle sorte que la solution rationnelle est un dénominateur égal à 1. Ainsi, cela nous permet d'éviter d'avoir une approximation du résultat.

Nous étudions dans un premier temps une borne sur la croissance des coefficients des résultats entiers nous permettant d'utiliser au maximum les routines de résolution numérique des BLAS (i.e. `dtrsm` en double précision et `strsm` en simple précision) à la place des derniers niveaux récursifs de l'algorithme `ULeft-Trsm`.

3.1.2.a Croissance des coefficients

La k-ième composante de la solution d'un système triangulaire d'ordre n étant une combinaison linéaire des $n-k$ composantes suivantes, on peut donc majorer la valeur maximale de la solution en fonction de la dimension du système et de la taille des entrées initiales. Il suffit donc de trouver la largeur maximale de blocs pour laquelle l'appel de la fonction `dtrsm` retournera un résultat exact. Ainsi en utilisant l'algorithme récursif par blocs et en remplaçant les derniers niveaux d'appel récursif par des appels à la fonction `dtrsm` on bénéficie au maximum des performances des BLAS. Dans la suite, nous définissons pour une matrice $M = [m_{ij}] \in \mathbb{Z}^{m\times n}$ ou un vecteur $v = [v_i] \in \mathbb{Z}^n$, les fonctions de magnitude $|M|$ et $|v|$ telles que $|M| = \max_{ij}(|m_{ij}|)$ et $|v| = \max_i(|v_i|)$. Nous définissons aussi la notion de matrice triangulaire unitaire pour parler des matrices triangulaires possédant uniquement des "1" sur la diagonale.

Lemme 3.1.2. *Soient $U \in \mathbb{Z}^{n\times n}$ une matrice triangulaire unitaire et $b \in \mathbb{Z}^n$ tels que $|T|, |b| < p$ et $p > 1$. Soit $x = [x_1, \ldots, x_n]^T \in \mathbb{Z}^n$ la solution entière du système $Tx = b$. Alors pour tout $k \in \{0, \ldots, n-1\}$,*

$$(p-2)^k - p^k \leq 2\tfrac{x_{n-k}}{p-1} \leq p^k + (p-2)^k \qquad \textit{si } k \textit{ est pair}$$

$$-p^k - (p-2)^k \leq 2\tfrac{x_{n-k}}{p-1} \leq p^k - (p-2)^k \quad \textit{si } k \textit{ est impair}$$

La preuve de ce théorème se fait à partir d'une induction sur les dépendances des x_i, en s'appuyant sur la relation $x_k = b_k - \sum_{i=k+1}^{n} T_{ki}x_i$. La preuve complète de ce théorème est proposée en annexe de [30].

Théorème 3.1.3. *La borne $|x| \leq \frac{p-1}{2}(p^{n-1} - (p-2)^{n-1})$ est la meilleure possible.*

Preuve. Considérons les séries $\{u_k\}_{k\geq 1}$ et $\{v_k\}_{k\geq 1}$ définies par les bornes du théorème 3.1.2 :

$$\begin{aligned} u_k &= \frac{p-1}{2}\left(p^k - (p-2)^k\right), \\ v_k &= \frac{p-1}{2}\left(p^k + (p-2)^k\right). \end{aligned}$$

Pour tout k, $u_k \leq v_k \leq v_{n-1}$. D'après le théorème 3.1.2, on peut borner les valeurs absolues des x_k par

$$\text{pour tout } k, \quad |x_k| \leq v_{n-1} = \frac{p-1}{2}\left(p^{n-1} + (p-2)^{n-1}\right).$$

Considérons le système $Tx = b$ suivant

$$\mathrm{T} = \begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ & 1 & p-1 & 0 & p-1 \\ & & 1 & p-1 & 0 \\ & & & 1 & p-1 \\ & & & & 1 \end{bmatrix}, \quad b = \begin{bmatrix} \vdots \\ 0 \\ p-1 \\ 0 \\ p-1 \end{bmatrix}.$$

La solution de ce système est $[v_n, v_{n-1}, \ldots, v_1]^T$ et la borne est donc atteinte ici. □

À partir de cette borne, on peut donc déterminer la taille maximale des systèmes pouvant être résolus à partir des routines BLAS. Pour un système de n équations et un corps de cardinalité p, il suffit que l'équation suivante soit vérifiée :

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

Ici, s représente la précision autorisée par les nombres flottants pour représenter des entiers. Par exemple, les nombres flottants double précision permettent une précision de 53 bits (voir §2.2.5), ce qui donne au plus des matrices 55×55 pour $p = 2$ et 4×4 pour $p = 9739$. Bien que cette borne limite l'utilisation des BLAS, nous verrons dans la section 3.1.4 que cette technique permet d'accélérer le temps de calcul par rapport à la version purement récursive.

Néanmoins, on peut pratiquement doubler la borne définie par l'équation (3.1) en utilisant une représentation centrée des éléments du corps finis (i.e. $-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}$). Ainsi, on obtient la borne suivante :

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

et on peut atteindre par exemple des matrices 93×93 pour $p = 2$.

3.1.2.b Gestion des divisions

Nous nous intéressons maintenant à éliminer les calculs approchées que peut entraîner la résolution numérique. En particulier, cette approximation provient du fait que la solution exacte du système est un nombre rationnel où le dénominateur est égal au déterminant de la matrice (règles de Cramer [36, théorème 25.6, page 706]). Le déterminant d'une matrice triangulaire étant égal au produit des éléments diagonaux, les divisions n'apparaisent que dans le dernier niveau récursif de l'algorithme `ULeft-Trsm`(i.e. $A_{11}^{-1} \times B$). On ne peut prédire si le résultat de ces divisions sera exacte ou non, cela dépend totalement du second membre B. Toutefois, si le système provient d'une matrice triangulaire unitaire alors ces divisions sont exactes (division par 1). L'idée est donc de transformer le système initial en un système unitaire de telle sorte que chaque appel récursif dans l'algorithme `ULeft-Trsm` soit unitaire. Soit le système $AX = B$, si $DA = U$, où D est une matrice diagonale et U est une matrice triangulaire unitaire. Alors, la résolution du système $UY = DB$ assure qu'aucune division n'est effectuée, et la solution du système inital est égale à Y. Pour déterminer un tel système, il faut donc calculer la matrice D qui correspond à l'inverse de la diagonale de A dans le corps fini et multiplier B par D. Le nombre d'opérations nécessaires pour réaliser cela est de :

- m inversions dans $\mathbb{Z}_p$ pour calculer D.
- $(m-1)\frac{m}{2} + mn$ multiplications dans $\mathbb{Z}_p$ pour calculer normaliser U et X.

Cependant, l'élimination des divisions n'est nécessaire que pour les résolutions à partir de la routine numérique `dtrsm`. On peut donc retarder l'utilisation des systèmes unitaires tant que l'on ne résout pas le système de façon numérique. Soit β la taille maximale des systèmes pouvant être résolus numériquement. Afin d'évaluer le coût relatif du calcul des systèmes unitaires, nous considérons que la dimension des matrices triangulaires est de l'ordre de $m = 2^i\beta$, où i le nombre d'appels récursifs dans l'algorithme `ULeft-Trsm`. Dans ce cas précis, il y a 2^i utilisations de systèmes unitaires de dimension β. Le coût total pour utiliser ces systèmes unitaires est donc de :

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

Cette implantation nous permet donc d'éviter $\left(\frac{1}{2} - \frac{1}{2^{i+1}}\right) m^2$ multiplications dans $\mathbb{Z}_p$ par rapport au passage à un système unitaire dès le début. En utilisant un seul niveau récursif, on économise $\frac{1}{4}m^2$ multiplications alors que le gain maximum est de $\frac{1}{2}(m^2 - m)$ multiplications pour $\log m$ niveaux récursifs.

3.1.3 Utilisation de réductions modulaires retardées

Nous nous intéressons maintenant à une autre alternative que l'utilisation de la routines de résolution numérique `dtrsm` pour remplacer les derniers niveaux récursifs de l'algorithme `ULeft-Trsm`. L'idée est d'utiliser l'algorithme de résolution de système linéaire "adhoc" par produits matrice-vecteur et d'incorporer une réduction modulaire retardée. Pour une matrice M, nous définissons les notations suivantes :

- $M_{i,*}$ et $M_{*,i}$ correspondent à la i-ème ligne (resp. colonne) de M
- $M_{i,[j,k]}$ et $M_{[j,k],i}$ correspondent aux composantes $\{j,\ldots,k\}$ de la i-ème ligne (resp. colonne) de M
- $M_{*,[j,k]}$ et $M_{[j,k],*}$ correspondent à la sous-matrice composée des lignes (resp. colonnes) $\{j,\ldots,k\}$ de M

L'algorithme de résolution "adhoc" se définit à partir de la relation suivante :
Soient $A = [a_{ij}] \in \mathbb{Z}_p^{m\times m}$ et $B, X \in \mathbb{Z}_p^{m\times n}$ tels que $AX = B$; on a la relation suivante :

$$\text{pour tout } i = \{1,\ldots,m\}, \quad X_{i,*} = \frac{1}{a_{ii}}(B_{i,*} - A_{i,[i+1,m]}X_{[i+1,m],*}) \tag{3.3}$$

L'algorithme "adhoc" calcule chacune des lignes de la solution l'une après l'autre en utilisant un produit matrice-vecteur sur les lignes de la solution dejà calculées.

Algorithme `adhoc-ULeft-Trsm`(A, B)
Entrées : $A = [a_{ij}] \in \mathbb{Z}_p^{m\times m}$, $B \in \mathbb{Z}_p^{m\times n}$
Sortie : $X \in \mathbb{Z}_p^{m\times n}$ tel que $AX = B$
Schéma
$x_m := a_{mm}^{-1} B_{m,*}$
pour i **de** $m-1$ **à** 1 **faire**

$$x_i := a_{ii}^{-1}(B_{i,*} - A_{i,[i+1,m]}X_{[i+1,m],*})$$

retourner $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$

Si l'on effectue les produits matrice-vecteur avec une réduction modulaire retardée (une seule réduction après le calcul entier) alors l'algorithme `adhoc-ULeft-Trsm` peut être utilisé si le système vérifie l'équation suivante :

$$t(p-1)^2 < 2^s \tag{3.4}$$

où t définit la dimension du systèmes et s représente la précision des calculs entiers.

Bien que la complexité de cet algorithme soit cubique en la taille des entrées, cela n'a pas d'incidence sur les performances en pratique, du fait que l'utilisation d'algorithmes de multiplication de matrices rapides n'est pas intéressante pour de petites matrices [28, §3.3.2]. Un paramètre important pour implanter cette méthode est de définir à partir de quelle dimension les systèmes doivent être résolus par l'algorithme `adhoc-ULeft-Trsm`. L'idée est de tirer parti des performances intrinsèques des processeurs pour les opérations d'addition et de multiplication lorsque toutes les données tiennent dans les caches. Le choix de ce seuil est donc fortement relié à la taille de la mémoire cache disponible. Si l'on effectue le produit matrice-vecteur à partir de produits scalaires, alors on peut espérer obtenir des performances quasi optimales pour certains processeurs [26]. La figure 3.1 tirée de [29] montre que pour des nombres premiers p suffisamment petits le produit scalaire d'ordre 512 par réduction moudlaire retardée s'avèrent être pratiquement au maximum des performances du processeur.

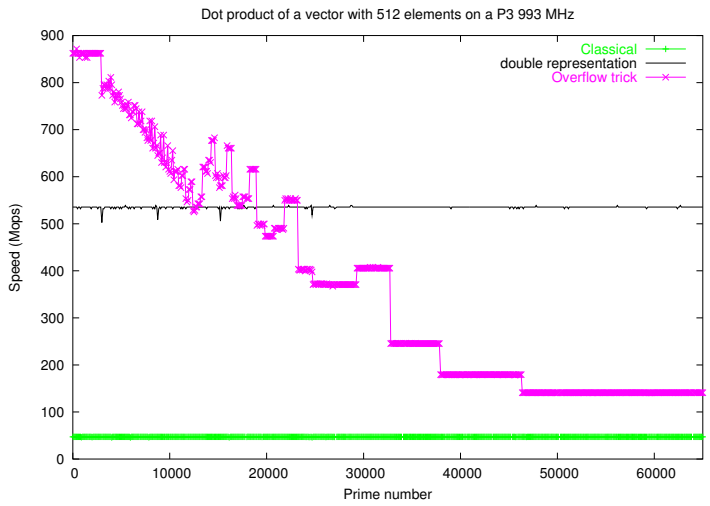

FIG. 3.1 – Produit scalaire avec réduction modulaire paresseuse (`P3-933Mhz`)

3.1.4 Comparaison des implantations

Nous nous intéressons maintenant à comparer les différentes variantes pour l'implantation de la résolution de systèmes linéaires triangulaires. L'algorithme `ULeft-Trsm` étant uniquement

basé sur l'utilisation du produit de matrices, cela nous permet de tirer parti des performances des routines développées par le projet FFLAS [28]. Nous comparons ici les différentes implantations que nous venons de présenter. Nous désignons "pure rec" l'implantation de l'algorithme `ULeft-Trsm` uniquement à partir de produit de matrices FFLAS . Ensuite nous désignons respectivement "BLAS " et "delayed" les deux variantes de l'algorithme `ULeft-Trsm` en remplaçant les derniers niveaux récursifs soit par des appels à la routines `dtrsm` des BLAS soit par l'algorithme `adhoc-ULeft-Trsm`. Afin de comparer plusieurs seuils pour l'utilisation de l'algorithme `adhoc-ULeft-Trsm`, nous notons celui-ci par "delayed$_t$" où t vérifie l'équation (3.4) et spécifie le seuil d'utilisation de l'agorithme `adhoc-ULeft-Trsm`. Nous comparons nos implantations en utilisant deux corps premiers présentés dans la section 2.2 (i.e. Modular<double>, Givaro-Zpz<Std32>). Nous utilisons une version des BLAS optimisée au travers du logiciel ATLAS[32][90] version 3.4.2. Nous testons ici des systèmes triangulaires denses carrés d'ordre n et nous exprimons les performances de nos routines en millions d'opérations arithmétiques sur un corps finis par seconde (`Mops`).

D'après la table 3.1, on peut voir que la version totalement récursive est toujours la moins performante. Cela s'explique par le fait que cette implantation effectue beaucoup plus de réduction modulaire que les deux autres car elle utilise des multiplications de matrices sur un corps fini jusqu'au dernier niveau récursif.

Les meilleures performances sont obtenues pour l'implantation utilisant la résolution numérique "BLAS " et le type de corps finis `Modular<double>`. En effet, cette représentation de corps finis permet d'éviter toutes les conversions entre les représentations flottantes et les éléments du corps. De plus, elle bénéficie des très bonnes performances de la résolution numérique des BLAS. Les routines BLAS sont ici appelées pour des systèmes d'ordre $n = 23$ pour $p = 5$. Toutefois, lorsque la taille du corps fini est plus importante (i.e. $p = 32749$), la dimension des systèmes descend à $n = 3$ et l'utilisation de l'algorithme `adhoc-ULeft-Trsm` et de réductions modulaires retardées avec le seuil $t = 50$ se révèle plus efficace pour des systèmes de dimension $n < 1000$. Cela s'explique par le bon comportement du produit scalaire avec les caches et par le fait qu'on effectue beaucoup moins de calculs sur les corps finis. Toutefois, le choix du seuil t doit être assez précis pour permettre un gain par rapport à la variante utilisant les BLAS. Afin de comparer exactement les variantes "BLAS " et "delayed", nous exprimons les performances pour des seuils identiques (i.e. 3, 23).

Dans la table 3.2, les performances obtenues par la variante "BLAS " ne sont pas toujours les meilleures. On remarque aussi que les performances obtenues sont globalement moins bonnes que celles obtenues avec le corps `Modular<double>` (table 3.1). En effet, l'utilisation principale du produit matriciel des FFLAS entraîne ici beaucoup de conversions de données entre le format flottant et les corps finis. Ces conversions deviennent d'ailleurs trop coûteuses dès lors que le corps fini est grand. En effet, pour $p = 32749$ la version "delayed$_{50}$" est toujours la meilleure. Lorsque le corps fini est petit, la variante "BLAS " s'avère plus performante pour des systèmes ayant une dimension supérieure à 2000.

Pour résumer, on préférera utiliser l'implantation basée sur la résolution de systèmes numériques avec le corps finis `Modular<double>` la plupart du temps. Toutefois, si un type de corps finis est imposé (à base d'entiers) l'utilisation de la version "delayed" peut se révéler la plus efficace si l'on utilise un seuil adapté au caractéristique de l'architecture utilisée. L'utilisation d'un logiciel d'optimisation automatique, similaire à ATLAS [90], pourrait permettre de déterminer un seuil optimal pour cette méthode en fonction d'une architecture déterminée.

[32] `http ://math-atlas.sourceforge.net`

$\mathbb{Z}/5\mathbb{Z}$						
n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	853	1216	1470	1891	2059	2184
BLAS	**1306**	**1715**	**1851**	**2312**	**2549**	**2660**
delayed$_{100}$	1163	1417	1538	1869	2042	2137
delayed$_{50}$	1163	1491	1639	1955	2067	2171
delayed$_{23}$	1015	1465	1612	2010	2158	2186
delayed$_{3}$	901	1261	1470	1937	2134	2166

$\mathbb{Z}/32749\mathbb{Z}$						
n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	810	1225	1449	1886	2037	2184
BLAS	1066	1504	**1639**	**2099**	**2321**	**2378**
delayed$_{100}$	1142	1383	1538	1860	2019	2143
delayed$_{50}$	**1163**	**1517**	**1639**	1955	2080	2172
delayed$_{23}$	1015	1478	1612	2020	2146	2184
delayed$_{3}$	914	1279	1449	1941	2139	2159

TAB. 3.1 – Performances (Mops) des routines `Trsm` pour `Modular<double>` (`P4-2.4Ghz`)

$\mathbb{Z}/5\mathbb{Z}$						
n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	571	853	999	1500	1708	1960
BLAS	688	1039	1190	**1684**	**1956**	**2245**
delayed$_{150}$	799	**1113**	909	1253	1658	2052
delayed$_{100}$	**831**	1092	**1265**	1571	1669	2046
delayed$_{23}$	646	991	1162	1584	1796	2086
delayed$_{3}$	528	755	917	1369	1639	1903

$\mathbb{Z}/32749\mathbb{Z}$						
n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	551	786	1010	1454	1694	1929
BLAS	547	828	990	1449	1731	1984
delayed$_{100}$	703	958	1162	1506	1570	1978
delayed$_{50}$	**842**	**1113**	**1282**	**1731**	**1890**	**2174**
delayed$_{23}$	653	952	1086	1556	1800	2054
delayed$_{3}$	528	769	900	1367	1664	1911

TAB. 3.2 – Performances (Mops) des routines `Trsm` pour `Givaro-ZpZ` (`P4-2.4Ghz`)

3.2 Triangularisations de matrices

Nous nous intéressons maintenant à l'un des problèmes majeurs de l'algèbre linéaire sur un corps fini pour des matrices denses non structurées, la triangularisation de matrice. Les triangularisations sont très importantes en algèbre linéaire car elles permettent de simplifier la plupart des problèmes (i.e. résolution de systèmes, inverse, rang, déterminant, ...). Nous focalisons ici notre attention sur les algorithmes de triangularisation basés sur la multiplication de matrices car ils autorisent les meilleures complexités théoriques.

Une de ces triangularisations est basée sur la factorisation sous forme LDU, où L, U et D sont respectivement un matrice triangulaire inférieure unitaire, une matrice triangulaire supérieure unitaire et une matrice diagonale. L'algorithme de Gauss par blocs permet de calculer une telle factorisation. Toutefois, cette factorisation n'est envisageable que sous certaines conditions de généricité (inversibilité, profil de rang générique). Un autre algorithme proposé en 1974 par Bunch et Hopcroft [8] permet d'éliminer la condition sur le profil de rang générique en utilisant la notion de pivotage et en introduisant la factorisation LUP, où L, U et P sont respectivement une matrice triangulaire inférieure unitaire, une matrice triangulaire supérieure et une matrice de permutation. Cependant, cette factorisation ne reste possible que pour des matrices inversibles. Enfin en 1982, Ibarra, Moran et Hui ont proposé une généralisation de la factorisation LUP au cas singulier en introduisant les factorisations LSP/LQUP. Nous nous intéressons ici à l'implantations d'un algorithme de factorisations LSP/LQUP en insistant sur des aspects pratiques tels que la gestion mémoire et la gestion du profil de rang. Nous proposons dans la suite plusieurs implantations récursives de cette algorithme basées sur la résolution de systèmes triangulaires et la multiplication de matrices. Tout d'abord, nous détaillons le schéma de l'algorithme original d'Ibarra *et al.* [46] en précisant sa complexité arithmétique. Ensuite, nous présentons une implantation de cet algorithme appelée `LUdivine` [67, 7] permettant de réduire l'espace mémoire en compressant L pour la stocker en place. Enfin, nous proposons une version totalement en place qui correspond à la factorisation LQUP d'Ibarra *et al.*. On peut noter qu'il est toujours possible à partir de ces différentes versions de retrouver la factorisation LSP simplement à partir d'extractions et de permutations.

3.2.1 Factorisation LSP

Soit A une matrice $m \times n$ définie sur un corps premier $\mathbb{Z}_p$. La factorisation LSP de A se définit par un triplet de matrices $< L, S, P >$ tel que $A = L \times S \times P$. Les matrices L et P sont définies de la même manière que pour la factorisation LUP alors que S se réduit à une matrice triangulaire supérieure non singulière quand toutes les lignes nulles ont été supprimées. L'algorithme permettant de se ramener au produit de matrices est un algorithme récursif utilisant l'approche "diviser pour régner". Dans la suite nous considérons sans perte de généralité que les dimensions des matrices sont des puissances de deux.

Algorithme `LSP`
Entrées : $A \in \mathbb{Z}_p^{m\times n}$, $m \leq n$
Sortie : $< L, S, P >$ tels que $A = LSP$: $L \in \mathsf{K}^{m\times m}$, $S \in \mathsf{K}^{m\times n}$, P une permutation
Schéma
si m=1 **alors**
$< L, S, P >:=< [\ 1\ ], AP^T, P >$; tel que $(AP^T)_{1,1} \neq 0$
sinon *Découper A en 2 blocs de* $\frac{m}{2}$ *lignes*

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$$

Soit $< L_1, S_1, P_1 >:= \texttt{LSP}(A_1)$ `(1 appel récursif)`

$$< L, S, P >:=< \begin{bmatrix} L_1 & \\ & I_{\frac{m}{2}} \end{bmatrix}, \begin{bmatrix} S_1 \\ A_2 P_1^T \end{bmatrix}, P_1 >$$

Découper S *en* r *et* $n-r$ *colonnes avec* $r = rang(S_1)$

$$S = \begin{bmatrix} S_1 \\ A_2 P_1^T \end{bmatrix} = \begin{array}{cc} \overbrace{}^{r} & \overbrace{}^{n-r} \\ \begin{bmatrix} T & Y \\ X & Z \end{bmatrix} \end{array}$$

Soit G *tel que* $GT = X$ *;* `(1 trsm)`

$$< L, S, P >:=< \begin{bmatrix} L_1 & \\ G & I_{\frac{m}{2}} \end{bmatrix}, \begin{bmatrix} T & Y \\ & Z - GY \end{bmatrix}, P_1 >$$ `(1 mat. mul.)`

Soit $< L_2, S_2, P_2 >:= \texttt{LSP}(Z - GY)$ `(1 appel récursif)`

$$< L, S, P >:=< \begin{bmatrix} L_1 & \\ G & L_2 \end{bmatrix}, \begin{bmatrix} T & Y \\ & S_2 \end{bmatrix}, \begin{bmatrix} I_r & \\ & P_2 \end{bmatrix} P_1 >$$

retourner $< L, S, P >$;

Lemme 3.2.1. *L'algorithme* `LSP` *est correct et le nombre d'opérations arithmétiques est borné par* $\frac{C_\omega}{2(2^{\omega-2}-1)} m^{\omega-1}(n - m\frac{2^{\omega-2}-1}{2^{\omega-1}-1})$ *opérations sur* K.

Preuve. La validité de l'algorithme `LSP` s'appuie sur une résolution du système $GT = X$. En effet, si une telle solution existe alors l'algorithme `LSP` est correct. Nous renvoyons le lecteur à [46] et [6, page 103].

Le coût de l'algorithme `LSP` se déduit à partir d'une expression récursive. Soient $C_{\texttt{LSP}}(m, n)$ le coût arithmétique de l'algorithme `LSP` pour une matrice $m \times n$, $C_{\texttt{Trsm}}(m, n)$ le coût de l'algorithme `URight-Trsm` (section 3.1.1) et $R(m, n, k)$ le coût de la multiplication de matrices rectangulaires. D'après l'algorithme `lsp`, on peut alors écrire la relation de récurrence suivante :

$$C_{\texttt{LSP}}(m, n) = C_{\texttt{LSP}}(\frac{m}{2}, n) + C_{\texttt{LSP}}(\frac{m}{2}, n - \frac{m}{2}) + R(\frac{m}{2}, \frac{m}{2}, n - \frac{m}{2}) + C_{\texttt{Trsm}}(\frac{m}{2}, \frac{m}{2}).$$

D'après la preuve de [67, théorème 1] on obtient la complexité attendue lorsque tous les blocs intermédiaires sont de plein rang. Il suffit de remarquer que $R(m, k, n-k) \leq R(m, m, n-m)$ quand $k \leq m$ pour obtenir la même complexité lorsque les blocs intermédiaires ne sont pas de plein rang (par exemple, rang $\leq \frac{m}{2^i}$). □

Le point important mis en évidence dans la figure 3.2 est le fait que la matrice L qui est $m \times m$ ne peut être stockée en place en dessous de S. L'implantation directe de cette factorisation

nécessite donc de stocker une matrice $m \times m$ en plus de la matrice A initiale en considérant que S soit stockée dans A. Afin de proposer des implantations nécessitant moins de ressources mémoire, nous nous intéressons dans les deux sections suivantes à compresser L pour qu'elle tienne en dessous de S et à effectuer l'ensemble des calculs en place.

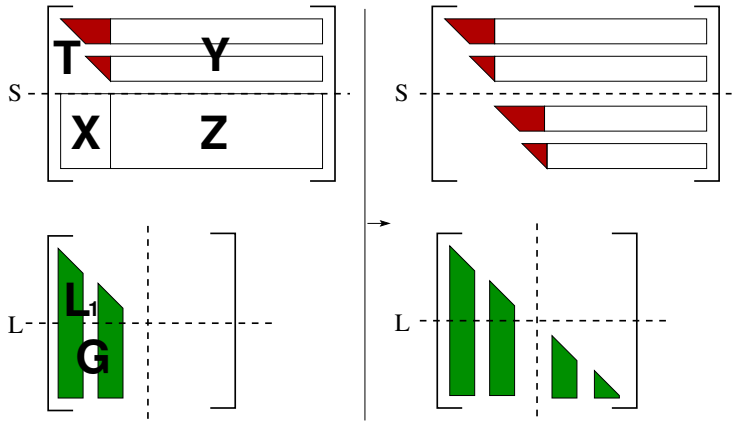

FIG. 3.2 – Principe de la factorisation `LSP`

3.2.2 LUdivine

On remarque d'après la figure 3.2 que bien que la matrice L ne peut être placée directement sous S, il y a assez de place sous S pour stocker tous les coefficients non nuls de L. Le principe de l'implantation `LUdivine` est de stocker uniquement les coefficients non nuls de L. D'après la figure 3.2 on voit que si la i-ème ligne de S est nulle alors la i-ème colonne correspondante dans L est égale au vecteur unité e_i ($e_i[i] = 1$ et $\forall k \neq i,\ e_i[k] = 0$). En ne stockant dans L que les colonnes différentes des e_i on remarque alors que l'on peut stocker L directement sous S comme décrit dans la figure 3.3. Soit $\tilde{L} = LQ$ la matrice contenant les colonnes non unitaires de L. Alors Q est telle que les premières lignes de $Q^T S$ forme une matrice triangulaire supérieure. La récupération de L est donc directe à partir de $\tilde{L}$ et S. On notera que la matrice $\tilde{L}$ correspond à la forme échelonnée décrite dans [49, 76] à une transposition près.

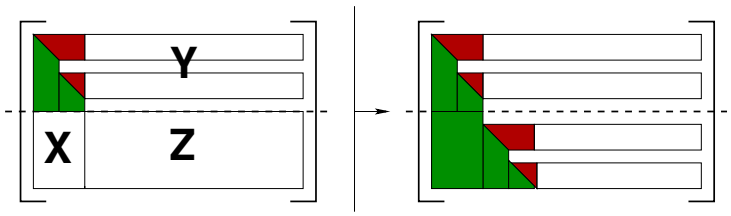

FIG. 3.3 – Principe de la factorisation `LUdivine`

Toutefois, cette implantation n'est pas encore satisfaisante car elle ne permet pas d'effectuer les calculs totalement en place. En effet, on remarque que pour résoudre le système $GT = X$ il faut compresser les lignes de T de telle sorte qu'elles engendrent un système triangulaire non singulier. De même pour le calcul du complément de Schur ($Z = Z - GY$) où il faut permuter les lignes de Y pour les faire correspondre aux colonnes non nulles de G. L'avantage de cette

implantation par rapport à l'implantation directe de la factorisation `LSP` est de tirer parti du rang des blocs intermédiaires, et donc d'obtenir un coût en $O(mnr^{\omega-2})$, où r est le rang de la matrice.

3.2.3 LQUP

Afin de proposer une version totalement en place, c'est-à-dire qui élimine aussi les lignes nulles dans S, on préférera utiliser la factorisation LQUP proposée dans [46]. La factorisation LQUP est reliée à la factorisation LSP par la relation $S = Q^T U$, où Q^T est une permutation telle que la matrice U soit triangulaire supérieure. En utilisant la même compression de L que pour l'implantation `LUdivine`, l'implantation `LQUP` est totalement en place (voir figure 3.4). En effet, du fait de la contiguïté des blocs T, Y et G ,l'utilisation de ressources mémoire supplémentaires pour résoudre $GT = X$ et calculer $Z = Z - GY$ n'est plus nécessaire.

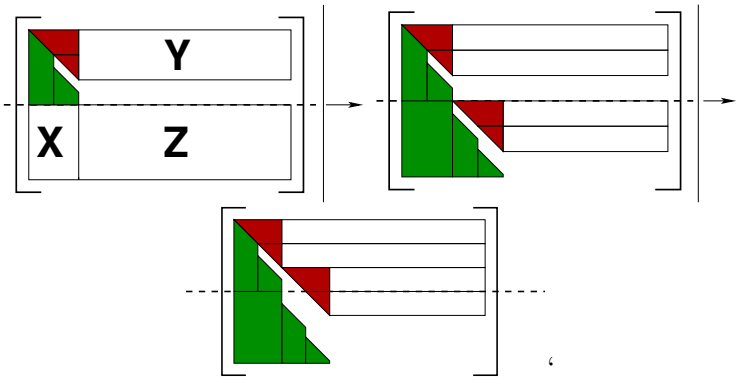

Fig. 3.4 – Principe de la factorisation `LQUP`

Toutefois, cette implantation nécessite encore de permuter les lignes de U et de L à chaque niveau récursif. Néanmoins, ces opérations sont quadratiques et ne sont pas primordiales dans le coût final de l'algorithme.

3.2.4 Performances et comparaisons

Nous comparons maintenant les performances de ces trois implantations. Les coûts arithmétiques de ces trois implantations se réduisent tous au produit de matrices ; seule la gestion de la mémoires et du rang diffère. Nos implantations reposent toutes sur le produit de matrices FFLAS [28] et sur la résolution de système triangulaire utilisant les BLAS présentée dans la section 3.1.2. Afin d'éviter les conversions de données entre les nombres flottants et les corps finis, nous utilisons l'implantation de corps finis `Modular<double>` qui nous permet d'obtenir les meilleures performances. Grâce à cette implantation, nous obtenons des résultats très satisfaisant, par exemple l'implantation `LQUP` permet de factoriser une matrice 3000×3000 dans $\mathbb{Z}_{101}$ en à peine 7.59 secondes sur un Pentium 4 cadencé à $2.4Ghz$ avec 512 Mo de RAM. En comparaison, le calcul de cette factorisation en numérique à partir de la bibliothèque `lapack`[33] s'effectue en 6 secondes.

Nous comparons maintenant le temps de calcul et l'espace mémoire effectif de nos trois routines. Nos relevés s'appuient sur le temps utilisateur et l'allocation mémoire des processus.

[33] http://www.netlib.org/lapck

Afin d'observer un comportement non générique de nos implantations, nous utilisons des matrices ayant une déficience de rang non nulle.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	0.05	0.48	8.01	32.54	404.8	1804
`LUdivine`	0.05	0.47	7.79	30.27	403.9	1691
`LQUP`	**0.05**	**0.45**	**7.59**	**29.90**	**201.7**	**1090**

TAB. 3.3 – Performances (secondes) `LSP`, `LUdivine`, `LQUP` pour $\mathbb{Z}_{101}$ (`P4-2.4Ghz`)

Nous observons dans la table 3.3 que les performances de nos trois implantations sont très proches sauf pour les matrices qui nécessitent des accès disque. Cela provient du fait que le coût dominant de ces implantations restent le produit de matrices. Toutefois, l'implantation `LSP` est la moins performante du fait que son coût arithmétique n'est pas dépendant du rang de la matrice et qu'elle effectue des opérations sur des lignes nulles. En effet, le calcul directe de $Z = Z - GY$ entraîne des opérations inutiles du fait de la présence des lignes nulles dans Y. En comparaison, la variante `LUdivine`, permet de meilleures performances du fait de la gestion des lignes nulles. Elle est cependant moins performante que l'implantation `LQUP` car elle nécessite encore des ressources mémoire supplémentaires pour compresser les lignes non nulles de Y. Bien que les temps de calculs de ces trois implantations sont très proches pour de petites matrices, on observe un saut de performance dès lors que les ressources mémoire sont supérieures à la mémoire RAM et nécessitent des accès disques. Typiquement, les versions `LSP` et `LQUP` nécessitent des accès disques pour des matrices d'ordre 8000 alors que la version `LQUP` tient encore dans la RAM (voir table 3.4), ce qui induit une chute de performances de quasiment 100%.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	2.83	17.85	160.4	444.2	1136.0	1779.0
`LUdivine`	1.60	10.00	89.98	249.9	639.6	999.5
`LQUP`	**1.28**	**8.01**	**72.02**	**200.0**	**512.1**	**800.0**

TAB. 3.4 – Ressource (Mo) `LSP`, `LUdivine`, `LQUP` pour $\mathbb{Z}_{101}$

En ce qui concerne les ressources mémoire de nos implantations, on remarque que l'implantation totalement en place `LQUP` permet d'économiser en moyenne 20% par rapport à l'implantation `LUdivine` et 55% par rapport à l'implantation `LSP`. Plus précisément, les ressources de `LSP` sont celles de `LUdivine` plus la mémoire pour stocker la matrice L, ce qui correspond exactement à la mémoire de `LQUP` : e.g. ($5000 \times 5000 \times 8ocets = 200Mo$). La mémoire temporaire utilisée par `LUdivine` permet de nous renseigner sur le profil de rang de la matrice. En outre, les ressources mémoire supplémentaires requises par `LUdivine` s'élèvent exactement à $r_1(n - r_1)$ éléments, où r_1 représente le nombre de lignes non nulles de Y au premier appel récursif. Ainsi, en comparant l'espace mémoire et la dimension des matrices, on peut connaître approximativement le défaut de rang de la matrice. Dans notre cas, les matrices utilisées ont un défaut de rang assez faible, ce qui se vérifie par le fait que les ressources mémoire supplémentaires sont très proches de $\frac{m^2}{4}$ éléments.

L'économie de ressources mémoire est un facteur important pour permettre de traiter de grandes matrices sans perte de performances due aux accès disque. Néanmoins, lorsqu'on se place en calcul *out of core* on préférera généralement utiliser des algorithmes qui nécessitent plus de ressource mémoire mais qui offrent une meilleure gestion de la localité des données. L'idée proposée dans [29] est d'utiliser l'algorithme de triangularisation `TURBO` [32] pour obtenir une

meilleure gestion de la localité dans le calcul du rang et du déterminant. Cet algorithme possède la même complexité arithmétique que l'algorithme `LQUP` mais permet d'offrir une meilleure localité en découpant les matrices en blocs carrés. Plus précisément, cet algorithme calcule une factorisation TU des matrices où T et U sont équivalentes à L et U à des permutations près. Cet algorithme utilise un découpage en 4 blocs carrés qui permet d'obtenir à la fois une meilleure parallélisation et une meilleure localité que l'algorithme `LQUP` [32]. L'utilisation d'un seul niveau récursif de cet algorithme permet d'ailleurs d'améliorer le calcul du rang en calcul *out of core* par rapport à la version `LQUP` seule [29, figure 6]. L'utilisation de tels algorithmes est donc importante pour améliorer les performances de nos routines pour des matrices de très grande dimension. L'implantation de l'algorithme `TURBO` sur plusieurs niveaux récursifs n'est pas encore effective du fait de sa complexité mais son utilisation nous permet déjà d'obtenir de meilleures performances que le schéma classique `LQUP`. L'utilisation de plus de niveaux récursifs de l'algorithme `TURBO` et l'utilisation de formats de blocs de données récursifs [42] sont donc indispensables pour la mise en place de routines en algèbre linéaire dense pour de très grandes données.

3.3 Applications des triangularisations

3.3.1 Rang et déterminant

Nous avons vu que la factorisation LQUP se réduit au produit de matrices. Le nombre d'opérations nécessaires pour la multiplication de matrices classique est de $2n^3 - n^2$ alors que la factorisation LQUP en requiert au plus $\frac{2}{3}n^3$. En théorie, le calcul de la factorisation LQUP est donc 3 fois plus rapide que celui du produit matriciel. Nous nous intéressons ici à comparer ce facteur théorique avec les facteurs obtenus par l'implantation `LQUP` et le produit matriciel `Fgemm` des FFLAS . Cette comparaison est pertinente du fait que nos routines sont toutes basées sur la multiplication de matrices classique `Fgemm`. Nous utilisons encore une fois le corps finis `Modular<double>` afin d'obtenir les meilleures performances possibles. Les tests ont été effectués sur un Pentium Xeon cadencé à 2.66Ghz avec 1Go RAM. Le produit matriciel `Fgemm` permet de de multiplier deux matrices d'ordre 5000 en seulement $69,19$ secondes, ce qui représente 3613 millions d'opérations par seconde. Ces performances exceptionnelles sont rendues possibles en pratique grâce aux BLAS qui exploitent le parallélisme et les pipelines des unités arithmétiques. Nous montrons dans la table 3.5 que notre implantation `LQUP` pour des matrices carrées n'est pas trop loin du ratio théorique de $\frac{1}{3}$ déduit des coûts arithmétiques des algorithmes.

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
`LQUP`	0.04s	0.20s	0.49s	2.85s	8.22s	34.13s
`Fgemm`	0.04s	0.24s	0.66s	4.44s	14.96s	69.19s
Ratio	**1**	**0.83**	**0.74**	**0.64**	**0.55**	**0.49**

TAB. 3.5 – Produit matriciel comparé à la factorisation LQUP sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`)

A partir de l'algorithme de factorisation $LQUP$ il est facile de dériver plusieurs autres algorithmes. Soit $A = LQUP$ alors :

- Le rang de la matrice A est égal au nombre de lignes non nulles de la matrice U.

- Le déterminant de A est égal au produit des éléments diagonaux de U. Notons qu'il est facile de proposer pour le déterminant une version adaptative de l'algorithme `LQUP` qui s'arrête

dès lors qu'une ligne nulle est trouvée dans U.

3.3.2 Inverse

L'inverse se déduit de la factorisation LQUP à partir d'une inverse triangulaire et d'une résolution de système. Toutefois, l'application directe de nos routines de résolution de système triangulaire permet de proposer une implantation immédiate pour l'inverse à partir de deux résolutions de système triangulaire matriciel. L'inverse triangulaire est alors remplacé par une résolution de système avec l'identité comme second membre.

Algorithme `Inverse`
Entrée : $A \in \mathbb{Z}_p^{m\times m}$, inversible.
Sortie : $A^{-1} \in \mathbb{Z}_p^{m\times m}$.
Schéma

$$\begin{array}{rcl} L, U, P & := & \texttt{LQUP}(A). \quad (Q = I_m \text{ car } \mathrm{rang}(A) = m) \\ X & := & \texttt{LLeft-Trsm}(L, I_m). \\ A^{-1} & := & P^T\ \texttt{ULeft-Trsm}(U, X). \end{array}$$

L'inverse est ici calculée avec $\frac{2}{3}n^3 + 2n^3$ opérations arithmétiques car la résolution de système nécessite n^3 opérations arithmétiques d'après le lemme 3.1.1. Ce qui nous donne un facteur théorique par rapport à la multiplication de matrices égal à 4/3. Nous pouvons encore voir dans la table 3.6 que cette implantation de l'inverse basée sur les routines `LQUP` et `Trsm` permet d'atteindre de bonnes performances par rapport à la multiplication de matrices.

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
INV	0.22s	0.81s	1.95s	11.28s	34.67s	147.3s
`Fgemm`	0.04s	0.24s	0.66s	4.44s	14.96s	69.19s
Ratio	**5.5**	**3.37**	**2.95**	**2.54**	**2.31**	**2.12**

TAB. 3.6 – Produit matriciel par rapport à l'inverse sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`)

Néanmoins, l'inverse se calcule normalement avec une inversion de matrice triangulaire. En théorie, l'utilisation de l'inverse triangulaire permet d'améliorer la constante pour le coût de l'inverse. L'inversion d'une matrice triangulaire se réduit aussi au produit de matrices et son coût arithmétique est de $\frac{1}{3}n^3$ opérations arithmétiques contre exactement n^3 pour la résolution de système. Soit $A \in \mathbb{Z}_p^{2m\times 2m}$ une matrice triangulaire supérieure, le schéma de l'algorithme de déduit alors de la relation :

$$A = \begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix} \Longrightarrow A^{-1} = \begin{bmatrix} {A_1}^{-1} & -{A_1}^{-1}A_2{A_3}^{-1} \\ & {A_3}^{-1} \end{bmatrix},$$

où $A_1, A_2, A_3 \in \mathbb{Z}_p^{m\times m}$. Le coût arithmétique se déduit à partir de ce schéma par la récurrence $C_{inv}(m) = 2C_{inv}(\frac{m}{2}) + 2T_{MM}(\frac{m}{2}, \frac{m}{2})$, où $T_{MM}(m, n)$ représente le coût du produit d'une matrice triangulaire $m \times m$ avec une matrice dense $m \times n$ (à savoir m^2n pour la multiplication classique). En utilisant cette inversion triangulaire à la place de la première résolution de système dans l'algorithme d'inversion, on obtient alors un coût de $\frac{2}{3}n^3 + \frac{1}{3}n^3 + n^3 = 2n^3$ opérations arithmétiques et donc un ratio de 1 par rapport à la multiplication de matrices.

La table 3.7 montre que cette nouvelle implantation permet d'améliorer à la fois les performances de l'inverse mais aussi le ratio avec le produit de matrices. Les performances exceptionnelles obtenues pour la multiplication de matrices rendent toutefois les facteurs de comparaison théorique pratiquement inaccessibles. En effet, nos implantations utilisent une approche "diviser pour régner" qui divise d'emblée la dimension des matrices par deux. Les performances des BLAS augmentant avec la dimension des matrices, on ne peut donc pas espérer obtenir 100% des performances du produit de matrices des BLAS contrairement à la routine `Fgemm`.

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
INV	0.14s	0.53s	1.34s	7.93s	22.94s	95.19s
`Fgemm`	0.04s	0.24s	0.66s	4.44s	14.96s	69.19s
Ratio	**3.5**	**2.2**	**2.03**	**1.78**	**1.53**	**1.37**

TAB. 3.7 – Produit matriciel par rapport à l'inverse sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`)

3.3.3 Base du noyau

Une base du noyau à gauche de la matrice A se déduit directement à partir de l'inverse de L :

Algorithme `left-nullspace`
Entrée : $A \in \mathbb{Z}_p^{m\times n}$.
Sortie : $B \in \mathbb{Z}_p^{m-r\times n}$, $r =$rang(A).
Schéma

$$\begin{aligned} L, Q, U, P &:= \texttt{LQUP}(A) \\ X &:= Q^T L^{-1} \\ B &:= X_{[r+1,m],*} \end{aligned}$$

Le coût arithmétique de cet algorithme est donc de $\frac{2}{3}n^3 + \frac{1}{3}n^3 = n^3$ opérations arithmétiques. Ce qui donne un facteur théorique par rapport à la multiplication égal à $\frac{1}{2}$. La table 3.8 montre encore une fois que notre implantation basée sur le produit de matrices `Fgemm` des FFLAS est vraiment efficace.

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
`left-nullspace`	0.07s	0.29s	0.69s	3.99s	11.6s	48.61s
`Fgemm`	0.04s	0.24s	0.66s	4.44s	14.96s	69.19s
Ratio	**1.75**	**1.21**	**1.04**	**0.89**	**0.77**	**0.7**

TAB. 3.8 – Produit matriciel par rapport à la base du noyau sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`)

3.3.4 Alternative à la factorisation LQUP : Gauss-Jordan

Une alternative à la factorisation $LQUP$ est d'utiliser l'approche de Gauss-Jordan par blocs [76, 11]. L'approche consiste à calculer la triangularisation en appliquant des transformations sur la matrice. Cette approche est similaire à la factorisation LQUP car elle permet de calculer les matrices L^{-1} et U si la matrice accepte une décomposition LU. A partir de cette triangularisation on peut donc déduire les mêmes types d'algorithmes qu'avec la factorisation LQUP pour le calcul du rang, du déterminant, de l'inverse et d'une base du noyau. En particulier, les

coûts arithmétiques de ces algorithmes sont identiques à ceux déduis de la factorisation LQUP et se réduisent tous au produit de matrices. Cette approche a été utilisée par A. Storjohann et Z. Chen [11] dans le logiciel MAPLE pour mettre en corrélation les ratios théoriques/pratiques par rapport au produit de matrices.

Dans cette partie, nous proposons une version incomplète de la triangularisation de Gauss-Jordan permettant de calculer une base du noyau avec $\frac{5}{6}n^3$ opérations arithmétiques. Notre approche se base sur le fait que l'algorithme de Gauss-Jordan par blocs permet de calculer directement L^{-1}. L'algorithme consiste à découper la matrice en deux blocs colonnes et d'effectuer des appels récursifs sur ces blocs. Ce type d'approche est appelé *slice and conquer*. Soit $A = [A_1 \ A_2] \in \mathbb{Z}_p^{2n\times 2n}$ une matrice inversible telle que $A_1, A_2 \in \mathbb{Z}_p^{2n\times n}$. Alors, l'algorithme de Gauss-Jordan fonctionne récursivement sur A_1, A_2 pour calculer les transformations $T_1, T_2 \in \mathbb{Z}_p^{2n\times 2n}$ telles que

$$T_1 A_1 = \begin{bmatrix} U_1 \\ 0 \end{bmatrix} \quad \text{et} \quad T_2(T_1 A_2) = \begin{bmatrix} * \\ U_2 \end{bmatrix}$$

avec $U_1, U_2 \in \mathbb{Z}_p^{n\times n}$ triangulaires supérieures. À partir de ces deux transformations on a donc $T_2 T_1 A = U \in \mathbb{Z}_p^{2n\times 2n}$ avec U triangulaire supérieure, ce qui signifie que $L^{-1} = T_2 T_1$.

L'idée que nous utilisons se base sur le fait que la connaissance totale de U n'est pas nécessaire pour calculer L^{-1} dans l'algorithme de Gauss-Jordan par blocs. En effet, à l'inverse de la factorisation LQUP, le complément de Schur est ici directement déduit de L^{-1} à chaque appel récursif. On peut donc se permettre dans la mise à jour de A_2 avec T_1 de ne pas calculer les n premières lignes car elles ne sont pas utilisées pour calculer T_2. Nous renvoyons le lecteur à [76, §2.2] pour une description plus détaillée de l'algorithme de Gauss-Jordan.

Sans perte de généralité, nous considérons le cas des matrices ayant un profil de rang générique et des dimensions égales à des puissances de deux. L'adaptation au cas de matrices non génériques se déduit en introduisant une matrice de permutation de lignes [76, §2.2]. Nous définissons dans un premier l'algorithme `GaussJordanTransform` qui pour une matrice $M \in \mathbb{Z}_p^{m\times n}$ et un entier $k \in \{1, \ldots, m\}$ calcule le rang de M et la transformation $T \in \mathbb{Z}_p^{m\times m}$ triangulaire inférieure unitaire telle que

$$TM = \begin{bmatrix} M_{[1,k-1],*} \\ U \end{bmatrix}$$

où U est une matrice triangulaire supérieure. Nous utilisons la notation $\tilde{U}_k$ pour définir la matrice obtenue en multipliant T et M.

Algorithme `GaussJordanTransform`(M, k)
Entrée : $M = [m_{ij}] \in \mathbb{Z}_p^{m\times n}$
Sortie : $< T, r >$ tel que $TM = \tilde{U}_k$ et $r =$rang(M)
Schéma
si $n = 1$ **alors**
  si $m_{k,1} = 0$ **alors**
    retourner $< I_m, 0 >$

  sinon

$$T := \begin{bmatrix} I_{k-1} & & \\ & 1 & \\ & -m_{k,1}^{-1} M_{[k+1,m]} & I_{m-k} \end{bmatrix}$$

retourner $< T, 1 >$

sinon

Découper M en deux blocs de $\frac{n}{2}$ colonnes $M = [M_1\ M_2]$

$< T_1, r_1 >:=$ `GaussJordanTransform`(M_1, k)

Découper T_1, M_1 et M_2 en trois blocs de $k-1$, r_1 et $m-k-r_1-1$ lignes

$$T_1 = \begin{bmatrix} I_{k-1} & & \\ & L_1 & \\ & G_1 & I_{m-k-r_1-1} \end{bmatrix}, \quad M_1 = \begin{bmatrix} A \\ B \\ C \end{bmatrix}, \quad M_2 = \begin{bmatrix} D \\ E \\ F \end{bmatrix}$$

$F := F + G_1 E$;

$< T_2, r_2 >:=$ `GaussJordanTransform`$(M_2, k + r_1)$

retourner $< T_2 T_1, r_1 + r_2 >$

Lemme 3.3.1. *L'algorithme* `GaussJordanTransform`(M, k) *est correct et son coût arithmétique est borné par $-\frac{7}{6}m^3 + 2m^2 n$ opérations sur $\mathbb{Z}_p$ pour une multiplication de matrices classique.*

Preuve. La preuve de cet algorithme se fait par récurrence sur les transformations T_1, T_2. Si T_1 et T_2 sont des transformations valides pour M_1 et M_2, alors il suffit de montrer que $T_2 T_1$ est une transformation valide pour $[M_1\ M_2]$. Il est facile de vérifier que les transformations calculées au dernier niveau de la récurrence sont correctes. Soient T_1 et T_2 les deux transformations calculées par le premier niveau récursif de l'algorithme `GaussJordanTransform` on a alors :

$$T_1 = \begin{bmatrix} I_{k-1} & & & \\ & L_1 & & \\ & \overline{G}_1 & I_{r_2} & \\ & \underline{G}_1 & & I_{m-k-r_1-r_2-1} \end{bmatrix}, \ T_2 = \begin{bmatrix} I_k & & & \\ & I_{r_1} & & \\ & & L_2 & \\ & & G_2 & I_{m-k-r_1-r_2-1} \end{bmatrix}$$

$$T_1 M_1 = \begin{bmatrix} I_{k-1} & & & \\ & L_1 & & \\ & \overline{G}_1 & I_{r_2} & \\ & \underline{G}_1 & & I_{m-k-r_1-r_2-1} \end{bmatrix} \begin{bmatrix} A \\ B \\ \overline{C} \\ \underline{C} \end{bmatrix} = \begin{bmatrix} A \\ U_1 \\ 0 \\ 0 \end{bmatrix}, \tag{3.5}$$

$$T_2 M_2^* = \begin{bmatrix} I_k & & & \\ & I_{r_1} & & \\ & & L_2 & \\ & & G_2 & I_{m-k-r_1-r_2} \end{bmatrix} \begin{bmatrix} D \\ E \\ \overline{F} + \overline{G}_1 E \\ \underline{F} + \underline{G}_1 E \end{bmatrix} = \begin{bmatrix} D \\ E \\ U_2 \\ 0 \end{bmatrix}, \tag{3.6}$$

avec M_2^* qui représente la matrice M_2 où l'on a effectué la mise à jour de F, à savoir $F := F + G_1 E$. Les matrices $\overline{G}_1$, $\underline{G}_1$, $\overline{C}$, $\underline{C}$, $\overline{F}$ et $\underline{F}$ représentent respectivement la partie haute et la partie basse des matrices G, C, F par rapport au rang r_2. En utilisant ce découpage, on peut alors écrire le produit des transformations $T_2 T_1$ par

$$T_2T_1 = \begin{bmatrix} I_k & & & \\ & L_1 & & \\ & L_2\overline{G}_1 & L_2 & \\ & \underline{G}_1 + G_2\overline{G}_1 & G_2 & I_{m-k-r_1-r_2} \end{bmatrix}. \tag{3.7}$$

Il faut maintenant vérifier que $T_2T_1[M_1\ M_2]$ a bien la forme souhaitée.

$$T_2T_1 \begin{bmatrix} A & D \\ B & E \\ \overline{C} & \overline{F} \\ \underline{C} & \overline{F} \end{bmatrix} = \begin{bmatrix} A & D \\ U_1 & L_1E \\ L_2\overline{G}_1B + L_2\overline{C} & L_2\overline{G}_1E + L_2\overline{F} \\ (\underline{G}_1 + G_2\overline{G}_1)B + G_2\overline{C} + \underline{C} & (\underline{G}_1 + G_2\overline{G}_1)E + G_2\overline{F} + \underline{F} \end{bmatrix}$$

D'après le système 3.5 on a :

$$\begin{array}{rcll} L_2\overline{G}_1B + L_2\overline{C} & = & L_2(\overline{G}_1B + \overline{C}) & = 0 \\ (\underline{G}_1 + G_2\overline{G}_1)B + G_2\overline{C} + \underline{C} & = & (\underline{G}_1B + \underline{C}) + G_2(\overline{G}_1B + \overline{C}) & = 0 \end{array}$$

D'après le système 3.6 on a :

$$\begin{array}{rcll} L_2\overline{G}_1E + L_2\overline{F} & = & L_2(\overline{F} + \overline{G}_1E) & = U_2 \\ (\underline{G}_1 + G_2\overline{G}_1)E + G_2\overline{F} + \underline{F} & = & \underline{F} + \underline{G}_1E + G_2(\overline{F} + \overline{G}_1E) & = 0 \end{array}$$

Ce qui donne

$$T_2T_1 \begin{bmatrix} A & D \\ B & E \\ \overline{C} & \overline{F} \\ \underline{C} & \overline{F} \end{bmatrix} = \begin{bmatrix} A & D \\ U_1 & L_1^{-1}E \\ & U_2 \\ & \end{bmatrix}$$

Nous considérons dans la suite que le coût du produit de matrices est $2n^3$. Soit $C_L(m,n,k)$ le coût de l'algorithme `GaussJordanTransform`(M,k) pour $M \in \mathbb{Z}_p^{m\times n}$. Ce coût s'exprime au travers de la relation de récurrence suivante :

$$\begin{array}{rcl} C_L(m,n,k) & = & C_L\left(m, \frac{n}{2}, k\right) + C_L\left(m, \frac{n}{2}, k + r_1\right) + R\left(m - k - r_1, r_1, \frac{n}{2}\right) \\ & & +T_{MM}(r_2, r_1) + R(m - k - r_1 - r_2, r_2, r_1). \end{array}$$

On voit que dans cette expression le paramètre m est toujours relié au paramètre k. On peut donc ramener cette récurrence à trois paramètres à une récurrence à seulement deux paramètres. Soit $G_L(m,n)$ cette nouvelle récurrence telle que

$$\begin{array}{rcl} G_L(m,n) & = & G_L\left(m, \frac{n}{2}\right) + G_L\left(m - r_1, \frac{n}{2}\right) + R\left(m - r_1, r_1, \frac{n}{2}\right) \\ & & +T_{MM}(r_2, r_1) + R(m - r_1 - r_2, r_2, r_1), \end{array}$$

on a alors l'égalité $C_L(m,n,k) = G_L(m-k,n)$. En considérant que les blocs colonnes sont toujours de plein rang, on peut donc remplacer r_1 et r_2 par $\frac{n}{2}$. Soit la récurrence $H_L(m,n)$ telle que

$$H_L(m,n) = H_L\left(m, \tfrac{n}{2}\right) + H_L\left(m - \tfrac{n}{2}, \tfrac{n}{2}\right) + R\left(m - \tfrac{n}{2}, \tfrac{n}{2}, \tfrac{n}{2}\right) + T_{MM}\left(\tfrac{n}{2}, \tfrac{n}{2}\right) + R\left(m - n, \tfrac{n}{2}, \tfrac{n}{2}\right).$$

En remplaçant les fonctions R et T_{MM} par leurs complexités respectives pour une multiplication classique et en résolvant la récurrence on obtient

$$\begin{aligned}
H_L(m,n) &= H_L\left(m,\tfrac{n}{2}\right) + H_L\left(m-\tfrac{n}{2},\tfrac{n}{2}\right) + 2\left(\tfrac{n}{2}\right)^2\left(m-\tfrac{n}{2}\right) + \left(\tfrac{n}{2}\right)^3 + 2(m-n)\left(\tfrac{n}{2}\right)^2 \\
&= H_L\left(m,\tfrac{n}{2}\right) + H_L\left(m-\tfrac{n}{2},\tfrac{n}{2}\right) + 4\left(\tfrac{n}{2}\right)^2 m - 5\left(\tfrac{n}{2}\right)^3 \\
&= 4\sum_{i=1}^{\log_2 n}\left[\left(\frac{n}{2^i}\right)^2 \sum_{j=0}^{2^{i-1}-1}\left(m - k\frac{n}{2^{i-1}}\right)\right] - 5\sum_{i=1}^{\log_2 n} 2^{i-1}\left(\frac{n}{2^i}\right)^3 \\
&= 4n^2 m \sum_{i=1}^{\log_2 n}\frac{2^{i-1}}{4^i} - 8n^3\sum_{i=1}^{\log_2 n} \\
&= 2n^2 m - \tfrac{7}{6}n^3 + O(n^2) + O(nm).
\end{aligned}$$

Ce qui donne l'inégalité $G_L(m,n) \leq 2n^2m - \frac{7}{6}n^3 + O(n^2) + O(nm)$. En considérant que $m = n$, on obtient bien le coût de $\frac{5}{6}m^3$ opérations. □

L'utilisation de cette algorithme pour calculer une base du noyau est directe. Il suffit de récupérer les lignes de T correspondantes aux lignes de M nulles quand on applique T. Nous n'avons pas encore développé l'implantation de cet algorithme, néanmoins nous pensons que les performances devraient être assez proches de celles de l'algorithme `left-nullspace` (§3.3.3). Toutefois, il y a un avantage à utiliser cette algorithme car il requiert moins de ressources mémoire. En effet l'inversion de matrices triangulaires ne peut se faire en place. Il faut donc utiliser une autre matrice pour calculer l'inverse ; ce qui représente un total de $2m^2$ éléments pour des matrices carrées d'ordre m. L'utilisation de l'algorithme `GaussJordanTransform` permet de définir une implantation qui nécessite moins de ressources mémoire. En particulier, le nombre d'élément à stocker est de $m^2 + \frac{m^2}{4}$. La plupart des opérations peuvent être effectuées en place. Seule la dernière multiplication T_2T_1 nécessite des ressources mémoire supplémentaires.

3.4 Interfaces pour le calcul "exact/numérique"

L'utilisation des routines numériques de type BLAS a permis d'obtenir des performances plus que satisfaisantes pour des calculs en algèbre linéaire sur les corps finis. La réutilisation de ces routines dans les logiciels de calcul formel est donc primordiale pour obtenir de meilleures performances. Dans ce but, nous avons développé plusieurs interfaces de calcul permettant une réutilisation simple et efficace des routines numériques BLAS. Pour cela, nous avons tout d'abord développé les paquetages FFLAS-FFPACK qui permettent l'utilisation des BLAS dans des calculs sur les corps finis. Ces paquetages s'appuient sur le formalisme des BLAS pour faciliter l'intégration des calculs numériques. Ces paquetages implantent des fonctions génériques sur les corps premiers en utilisant le mécanisme template de C++ et en utilisant le modèle de base de corps finis de la bibliothèque LinBox. Dans le but d'offrir une utilisation simplifiée et optimale de ces routines, nous avons développé une interface entre le logiciel MAPLE et les paquetages FFLAS-FFPACK. Cette interface se base sur la réutilisation des structures de données MAPLE compatibles avec le formalisme des BLAS. Enfin, nous avons intégré l'ensemble de ces routines à la

bibliothèque LinBox à partir d'une interface de calcul spécialisée. Cette intégration repose sur la définition d'un domaine de calcul qui permet d'utiliser les routines BLAS de façon transparente par des classes C++. Nous présentons maintenant l'ensemble des ces implantations.

3.4.1 Interface avec les BLAS

L'ensemble des algorithmes et des implantations présentés dans les sections 3.1, 3.2 et 3.3 sont disponibles dans les paquetages FFLAS-FFPACK[34]. Ces paquetages sont basés sur la représentation matricielle des BLAS, c'est-à-dire que les matrices sont stockées de façon linéaire dans un tableau de données, soit en ligne soit en colonne. Toutes les fonctions développées dans ces paquetages sont génériques par rapport au type de corps finis. Le modèle de corps finis utilisé est celui de la bibliothèque LinBox auquel nous avons rajouté des fonctions de conversion entre la représentation des éléments d'un corps fini et la représentation en nombres flottants double précision. Grâce à la représentation linéaire des matrices et aux fonctions de conversions, l'utilisation des routines BLAS est pratiquement immédiate. Pour cela, il suffit de convertir les matrices à coefficients sur un corps fini dans un format double précision, d'appeler les routines BLAS et de convertir le résultat dans la représentation du corps fini initial. On rappelle que les calculs effectués par les routines BLAS doivent retourner un résultat exact (pas de dépassement de capacité ou de divisions flottantes). Nous avons vu par exemple que la résolution de systèmes linéaires triangulaires nécessite l'utilisation de systèmes unitaires pour supprimer les divisions (voir §3.1.2).

Les routines que nous proposons dans les paquetages FFLAS sont standardisées en réutilisant la nomenclature définie par les BLAS. On retrouve ici les trois niveaux définis dans les BLAS :

BLAS **1** :
- `MatF2MatD` : conversion des matrices (corps finis → double)
- `MatD2MatF` : conversion des matrices (double → corps finis)
- `fscal` : mise à l'échelle par un scalaire
- `fcopy` : copie des données
- `fswap` : échange des données

BLAS **2** :
- `fgemv` : produit matrice-vecteur avec mise à l'échelle
- `ftrsv` : résolution de systèmes linéaires triangulaires

BLAS **3** :
- `ftrsm` : résolution de systèmes linéaires triangulaires matriciels
- `ftrmm` : produit de matrices triangulaire (un seul des opérandes suffit)
- `fgemm` : produit de matrices avec addition et mise à l'échelle
- `fsquare` : mise au carré de matrices avec addition et mise à l'échelle

Le paquetage FFPACK propose les routines suivantes :
- `applyP` : multiplication par une permutation
- `LUdivine` : factorisation LSP/LQUP
- `Rank` : calcul du rang
- `TURBO` : calcul du rang par l'algorithme "Turbo" [32]
- `IsSingular` : test pour la singularité
- `Det` : calcul du déterminant

[34] http://www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/FFLAS/

- `Invert` : calcul de l'inverse
- `CharPoly` : calcul du polynôme caractéristique
- `MinPoly` : calcul du polynôme minimal

Les deux dernières implantations (`CharPoly` et `MinPoly`) sont exclusivement dues au travaux de C. Pernet sur la réutilisation de la factorisation LQUP pour calculer le polynôme minimal et le polynôme caractéristique [67].

Afin de conserver les possibilités de calculs en place proposées par les BLAS, les routines que nous avons définies nécessitent plusieurs informations sur le stockage des matrices. On a besoin de connaître les dimensions, si la matrice est transposée ou non, si la matrice est unitaire et surtout quel est le *stride* des données. Le *stride* est un entier positif qui définit la distance mémoire entre deux éléments adjacents non consécutifs dans le stockage linéaire de la matrice. Par exemple, pour une matrice à m lignes et n colonnes stockée en ligne, le *stride* est égal à n. L'utilisation de ce paramètre est important car il permet de travailler sur des sous-matrices sans avoir à faire de copie. Les implantations que nous avons proposées dans les paquetages FFLAS-FFPACK tiennent compte de toutes ces caractéristiques. Nous utilisons des énumérations pour spécifier les caractères attribués aux matrices et aux calculs. Par exemple, l'implantation de la résolution de systèmes triangulaires matriciels est faite à partir d'une interface permettant d'appeler des routines spécialisées en fonction des caractères des matrices et des calculs. Les différents caractères que nous proposons sont :

- pour les matrices : *unitaire, non unitaire, triangulaire inférieure et supérieure*

```
enum FFLAS_UPLO      { FflasUpper=121, FflasLower=122 };
enum FFLAS_DIAG      { FflasNonUnit=131, FflasUnit=132 };
```

- pour les calculs : *matrice transposée, résolution à droite ou à gauche, type de factorisation*

```
enum FFLAS_TRANSPOSE { FflasNoTrans=111, FflasTrans=112};
enum FFLAS_SIDE      { FflasLeft=141, FflasRight = 142 };
enum FFLAPACK_LUDIVINE_TAG { FflapackLQUP=1,
                             FflapackSingular=2,
                             FflapackLSP=3,
                             FflapackTURBO=4};
```

Les caractéristiques de type `FFLAPACK_LUDIVINE_TAG` permettent de configurer le type des calculs effectués par la factorisation LQUP. Par exemple, l'utilisation de `FflapackSingular` permet d'arrêter le calcul de la factorisation dès qu'une ligne nulle est rencontrée. Cette caractéristique est utilisée par les routines `isSingular` et `Det`.

Nous détaillons maintenant l'interface de résolution des systèmes linéaires triangulaires qui permet, en fonction des caractéristiques des matrices et du calcul, d'utiliser des fonctions spécialisées. Les paramètres utilisés par cette interface sont les suivants :

- `Side` : coté de la résolution (FflasLeft→ $AX = B$ et FflasRight→ $XA = B$)
- `Uplo` : A est une matrice triangulaire de type `Uplo`
- `TransA` : résolution à partir de A ou A^T
- `Diag` : A est unitaire ou non
- `M,N` : $B \in F^{M\times N}$ et $A \in F^{M\times M}$ ou $A \in F^{N\times N}$ suivant `Side`
- `alpha` : coefficient de mise à l'échelle de la solution ($alphaX$)
- `A,lda` : un pointeur sur le début de la zone mémoire de A et la valeur du *stride* associé
- `B,ldd` : un pointeur sur le début de la zone mémoire de B et la valeur du *stride* associé

Code 3.1 – Interface de résolution de systèmes triangulaires des FFLAS (`ftrsm`)

```
template<class Field>
inline void
FFLAS::ftrsm(const Field                          &F,
             const enum FFLAS_SIDE                Side,
             const enum FFLAS_UPLO                Uplo,
             const enum FFLAS_TRANSPOSE           TransA,
             const enum FFLAS_DIAG                Diag,
             const size_t                         M,
             const size_t                         N,
             const typename Field::Element        alpha,
             typename Field::Element              *A,
             const size_t                         lda,
             typename Field::Element              *B,
             const size_t                         ldb)
{
    if (!M || !N ) return;
    integer pi;
    F.characteristic(pi);
    long long p = pi;
    size_t nmax = bound(p);

    if ( Side==FflasLeft ){
      if ( Uplo==FflasUpper){
        if (TransA == FflasNoTrans)
          ftrsmLeftUpNoTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
        else
          ftrsmLeftUpTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
      }
      else{
        if (TransA == FflasNoTrans)
          ftrsmLeftLowNoTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
        else
          ftrsmLeftLowTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
      }
    }
    else{
      if ( Uplo==FflasUpper){
        if (TransA == FflasNoTrans)
          ftrsmRightUpNoTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
        else
          ftrsmRightUpTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
      }
      else{
        if (TransA == FflasNoTrans)
```

```
45            ftrsmRightLowNoTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
          else
            ftrsmRightLowTrans(F,Diag,M,N,alpha,A,lda,B,ldb,nmax);
        }
      }
50  }
```

L'ensemble des fonctions de résolutions spécifiques est implanté de la même manière. Elles prennent un paramètre supplémentaire, appelé `nmax` dans le code 3.1, qui définit la dimension maximale des systèmes définis sur le corps F pouvant être résolus en utilisant la routine de résolution des BLAS `cblas_dtrsm`. Cette dimension est déterminée à partir de la fonction `bound` qui est basée sur l'équation 3.1 (§3.1.2.a). Comme décrit dans la section 3.1.2, ces implantations s'appuient sur l'algorithme récursif `Trsm` (§3.1.1) en remplaçant les derniers niveaux récursifs par des résolutions numériques BLAS dès que la taille du système est inférieure à `nmax`. Dans ces implantations, nous utilisons les fonctions `fscal`, `fgemm`, `MatF2MatD` et `MatD2MatF` définies dans le paquetage FFLAS. Dans le code suivant, nous proposons l'implantation de la résolution à gauche avec des matrices triangulaires inférieures non transposées :

Code 3.2 – Résolution de système linéaire triangulaire à partir des BLAS

```
  template<class Field>
  inline void
  FFLAS::ftrsmLeftLowNoTrans(const Field&                        F,
                             const enum FFLAS_DIAG              Diag,
5                            const size_t                          M,
                             const size_t                          N,
                             const typename Field::Element alpha,
                             typename Field::Element            *A,
                             const size_t                        lda,
10                           typename Field::Element            *B,
                             const size_t                        ldb,
                             const size_t                       nmax)
    {
      static typename Field::Element Mone;
15    static typename Field::Element one;
      F.init(Mone, -1);
      F.init(one, 1);
      if ( M <= nmax ){
        typename Field::Element inv;
20      if (Diag == FflasNonUnit ){
          //Normalization of A and B
          typename Field::Element * Ai = A;
          typename Field::Element * Bi = B;
          for (size_t i=0; i<M; ++i){
25          F.inv( inv, Ai[i] );
            fscal(F, i, inv, Ai, 1 );
            fscal(F, N, inv, Bi, 1 );
            Ai += lda; Bi+=ldb;
          }
30      }
        double alphad;
        if (F.areEqual(alpha, Mone))
          alphad = -1.0;
        else
```

```
35        F.convert( alphad, alpha );

        double *Ad = new double[M*M];
        double *Bd = new double[M*N];
        MatF2MatD( F, Ad, A, lda, M, M );
40      MatF2MatD( F, Bd, B, ldb, M, N );

        cblas_dtrsm( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans,
                     CblasUnit, M, N, alphad, Ad, M, Bd, N );
        delete[] Ad;
45      MatD2MatF( F, B, ldb, Bd, M, N );
        delete[] Bd;
        if (Diag == FflasNonUnit ){
          //Denormalization of A
          typename Field::Element *  Ai=A;
50        for (size_t i=0; i<M; ++i){
            fscal( F, i, *(Ai+i), Ai, 1 );
            Ai += lda;
          }
        }
55    }
      else{
        size_t Mup=M>>1;
        size_t Mdown = M-Mup;
        ftrsmLeftLowNoTrans( F, Diag, Mup, N, alpha,
60                             A, lda, B, ldb, nmax);

        fgemm( F, FflasNoTrans, FflasNoTrans, Mdown, N, Mup,
               Mone, A+Mup*lda, lda, B, ldb, alpha, B+Mup*ldb, ldb);

65      ftrsmLeftLowNoTrans( F, Diag, Mdown, N, one,
                             A+Mup*(lda+1), lda, B+Mup*ldb, ldb, nmax);
      }
    }
```

La ligne 18 permet de basculer entre les appels récursifs et les appels aux routines numériques BLAS. Si la dimension du système est suffisamment petite (`M <= nmax`) alors on utilise une résolution numérique. La ligne 20 détermine si la matrice A est unitaire et entraîne une normalisation si nécessaire. Cette phase de normalisation permet d'éviter les divisions dans la résolution numérique comme nous l'avons expliqué dans la section 3.1.2. Pour cela, on utilise la fonction `fscal` sur les ligne de A et de B. On convertit ensuite les données dans un format flottant double précision grâce à la fonction `MatF2MatD` et on appelle la fonction de résolution des BLAS `cblas_dtrsm`. Cette fonction calcule la solution en place dans le second membre de l'équation, à savoir Bd. Enfin, on convertit ce résultat dans B et on modifie A si nécessaire pour retrouver la matrice d'origine. Si le système possède une dimension supérieure à la borne de résolution des BLAS, alors on effectue deux appels récursifs sur des sous-matrices de A en mettant à jour le second membre par la routine `fgemm` comme décrit par l'algorithme `Trsm` (§3.1.1). La récupération des sous-matrices se fait par décalage du pointeur sur la zone mémoire contiguë de A en conservant le *stride* initial de A et en définissant de nouvelles dimensions.

L'utilisation de ces routines nécessite de définir les matrices à partir de tableaux de données linéaires et d'associer les caractéristiques de l'instance du problème et du type de résolution. Voici un exemple d'utilisation de ces routines dans le cas de la résolution du système $AX = B$ avec $A \in$

$\mathbb{Z}_{17}^{10\times 10}$ triangulaire supérieure et $X, B \in \mathbb{Z}_{17}^{10\times 20}$. Nous utilisons le corps fini `Modular<double>` présenté dans la section 2.2.1 et les matrices sont initialisées à l'aide de fichier qui stockent les coefficients de façon linéaire.

Code 3.3 – Exemple d'utilisation de la routine `ftrsm` des FFLAS

```
int main(){

  Modular<double> F(17);
  typedef Modular<Double>::Element Element;

  size_t m,n;
  m=10;
  n=20;

  Element *A=new Element[m*m];
  Element *B=new Element[m*n];
  Element one;
  F.init(one,1);

  ifstream readA("A.txt"), readB("B.txt");

  // lecture de A dans le fichier "A.txt"
  for (size_t i=0;i<m;++i)
     for (size_t j=i;j<m;++j)
        F.read(readA, A[j+i*m]);

  // lecture de B dans le fichier "B.txt"
  for (size_t i=0;i<m;++i)
     for (size_t j=0;j<n;++j)
        F.read(readB, B[j+i*n]);

  readA.close();
  readB.close();

  // B <-A^(-1).B
  FFLAS::ftrsm(F, FFLAS::FflasLeft, FFLAS::FlasUpper,
               FflasNoTrans, FflasNonUnit,
               m,n,one,A,m,B,n);
  delete[] A;
  delete[] B;
  return 0;
}
```

Ce code montre que l'utilisation de nos routines est somme toute assez simple. Néanmoins, l'utilisateur doit quand même gérer l'allocation et la désallocation des matrices ainsi que le paramétrage des calculs. Afin d'améliorer l'utilisation, nous proposons dans la section 3.4.3 d'intégrer ces routines dans la bibliothèque LinBox et d'utiliser des matrices définies à partir de classes C++ pour définir des environnements de calcul de plus haut niveau. Tout d'abord, nous nous intéressons dans la prochaine section à réutiliser ces routines dans le logiciel Maple.

3.4.2 Connections avec Maple

Afin de proposer l'utilisation de ces paquetages dans un environnement de calcul formel plus complet, nous avons développé une interface entre les routines des paquetages FFLAS/FFPACK

et le logiciel MAPLE. Le but de cette interface est de pouvoir utiliser nos routines à l'intérieur de code MAPLE et surtout de pouvoir effectuer le calcul sur les données MAPLE pour bénéficier des calculs en place. Pour cela notre travail est double. Il faut dans un premier temps rendre les données MAPLE accessibles à nos routines. Dans un deuxième temps il faut pouvoir intégrer nos routines à l'intérieur de l'environnement MAPLE.

La première partie de nos travaux a donc consisté à récupérer des données MAPLE, à les intégrer dans du code C et à effectuer des calculs sur ces données à partir de nos routines. Pour cela, nous avons utilisé l'*Application Programming Interface* (API) OpenMaple[35] développée par MAPLE et qui permet de réutiliser des routines et des structures de données MAPLE dans du code C, Java ou VisualBasic. En utilisant le fichier "maplec.h" fourni par cette API, nous avons pu convertir les données MAPLE dans un format typé pour nos routines et ainsi nous avons défini un *wrapper* de nos routines pour des données MAPLE. Cependant, certains paramètres de nos routines sont définis de façon générique et la réutilisation de ce *wrapper* à l'intérieur de MAPLE nécessite l'utilisation de bibliothèques dynamiques. Afin de proposer la version la plus efficace possible de nos routines pour ce *wrapper*, nous avons supprimé le caractère générique de nos routines en fixant le corps fini au type `Modular<double>` qui permet d'atteindre les meilleures performances. Toutefois, une évolution naturelle de ce wrapper serait d'intégrer nos routines au travers de l'archétype de données des corps finis de la bibliothèque LinBox et d'en proposer une version générique. Cette approche n'était pas notre première intention dans le développement de cette interface et n'est donc pas encore disponible. Nous présentons dans le code 3.4 les principes de notre *wrapper* sur l'exemple de l'inversion de matrice.

Code 3.4 – Wrapper MAPLE pour les routines des paquetages FFLAS-FFPACK

```
#include "maplec.h"
#include "linbox/field/modular.h"
#include "linbox/fflas/fflas.h"
#include "linbox/fflapack/fflapack.h"

using namespace LinBox;
typedef Modular<double> Field;
typedef Field::Element Element;


extern "C" {

  ALGEB inverse(MKernelVector kv, ALGEB* argv ) {
    /* expect arguments in following order:
    1- p (int prime)
    2- m (int)
    3- A (matrix)
    4- X (matrix)
    */

    int p,m;
    p=MapleToInteger32(kv,argv[1]);
    m=MapleToInteger32(kv,argv[2]);

    Field F(p);
    Element *A,*X;
```

[35] http://www.mapleapps.com/categories/maple9/html/OpenMaple.html

```
RTableSettings settings;
ALGEB Matrix;

Matrix= (ALGEB)argv[3];
RTableGetSettings(kv,&settings,Matrix);
if( settings.data_type != RTABLE_FLOAT64
    || settings.storage != RTABLE_RECT
    || !IsMapleNULL(kv,settings.index_functions) )
{
  MaplePrintf(kv,"Making a copy \n");
  settings.data_type        = RTABLE_FLOAT64;
  settings.storage          = RTABLE_RECT;
  settings.index_functions = ToMapleNULL(kv);
  settings.foreign          = FALSE;
  Matrix = RTableCopy(kv,&settings,Matrix);
}
A= (Element*) RTableDataBlock(kv,Matrix);

Matrix= (ALGEB)argv[4];
RTableGetSettings(kv,&settings,Matrix);
if( settings.data_type != RTABLE_FLOAT64
    || settings.storage != RTABLE_RECT
    || !IsMapleNULL(kv,settings.index_functions) )
{
  MaplePrintf(kv,"Making a copy \n");
  settings.data_type        = RTABLE_FLOAT64;
  settings.storage          = RTABLE_RECT;
  settings.index_functions = ToMapleNULL(kv);
  settings.foreign          = FALSE;
  Matrix = RTableCopy(kv,&settings,Matrix);
}
X= (Element*) RTableDataBlock(kv,Matrix);

FFLAPACK::Invert(F,m,A,m,X,m);

Matrix = (ALGEB)argv[4];
RTableGetSettings(kv,&settings,Matrix);
switch (settings.data_type) {

  case RTABLE_FLOAT64:
     break;

  case RTABLE_INTEGER32:
     MaplePrintf(kv,"Converting the result to int[4]\n");
     int* Ae;
     Ae= (int *)RTableDataBlock(kv,Matrix);
     for (int i=0;i<m*m;i++)
        *(Ae+i)= (int) *(X+i);
     break;

  case RTABLE_DAG:
     MaplePrintf(kv,"Converting the result to maple int\n");
     ALGEB* Cee;
     Cee= (ALGEB *)RTableDataBlock(kv,Matrix);
     for (int i=0;i<m*m;i++)
        *(Cee+i)= ToMapleInteger(kv,(long)*(X+i));
     break;
}
```

```
      return Matrix;
    }
  }
```

Le principe de la fonction `inverse` définie à la ligne 13 du code 3.4 se base sur des paramètres de type `MKernelVector` et de type `ALGEB`. Le type `MKernelVector` définit une instance de l'environnement MAPLE alors que le type `ALGEB` décrit l'ensemble des objets utilisés dans cet environnement. L'environnement correspond soit à l'environnement courant lorsqu'on travaille directement dans une session MAPLE soit à un environnement lancé à partir d'un code externe. Dans notre cas, nous considérons que nous sommes à l'intérieur d'une session MAPLE. Le paramètre `argv` de type `ALGEB` définit un tableau de paramètres passés comme instance de la fonction. Ce paramètre correspond au `char** argv` que l'on définit généralement en C comme paramètre du programme principal `main`. Les lignes 22 et 23 permettent de convertir les deux premiers paramètres de la fonction en deux entiers machines 32 bits. Ces paramètres sont du type `ALGEB` et représentent donc tous les types MAPLE. Néanmoins, nous supposons que les valeurs données en paramètre sont des nombres. Le code entre la ligne 31 et 43 permet de convertir le troisième paramètre de la fonction, qui doit être une matrice, au format de données de nos routines (un tableau linéaire de nombres flottants double précision). La ligne 59 permet de récupérer les données de la structure MAPLE au travers d'un pointeur de données. Nous réitérons ce procédé pour le quatrième argument entre la ligne 46 et 59 qui est la matrice permettant de stocker le résultat. Une fois que les structures de données MAPLE sont accessibles, il suffit d'appeler la routine d'inversion du paquetage FFPACK sur les pointeurs de données (cf ligne 61). Ensuite, il faut convertir le résultat dans la structure de données MAPLE prévue à cet effet.

La deuxième partie concerne l'intégration de ce *wrapper* à l'intérieur d'un environnement MAPLE. Pour cela nous utilisons le paquetage d'appels externes proposé par MAPLE qui permet d'appeler des fonctions C ou Fortran au travers d'objets dynamiques. En particulier, nous utilisons la procédure define_external qui permet de spécifier qu'une fonction est externe. Typiquement, l'utilisation de cette procédure pour connecter une fonction définie en C de type `int myCfunction(int, int)` dans MAPLE se fait en définissant une procédure du type :

```
myMaplefunction:=define_external('myCfunction',
                                 a::integer[4],
                                 b::integer[4],
                                 RETURN::integer[4],
                                 LIB="myCfunction.so");
```

La variable LIB spécifie la localisation de l'objet dynamique définissant la fonction "myCfunction(int,int)". Les paramètres a et b définissent le type des paramètres de la fonction C et RETURN spécifie le type de retour de la fonction. Le type integer[4] correspond aux entiers machines 32 bits. L'appel de la procédure myMaplefunction(4,5) ; dans un environnement MAPLE entraîne l'exécution du code C `myCfunction(4,5)` et récupère la valeur de retour. L'intégration de notre *wrapper* au travers de ce type de procédure est donc immédiate. Pour cela, il suffit de compiler notre *wrapper* à l'intérieur d'une bibliothèque dynamique "lbinverse.so" et de définir la procédure

```
lbinverse:=define_external('inverse', MAPLE, LIB="lbinverse.so");
```

On remarque que les types des paramètres de la fonction ne sont plus spécifiés. On utilise ici le paramètre MAPLE pour spécifier que les paramètres de la fonction externe sont de type MAPLE.

Afin de simplifier l'utilisation de notre *wrapper*, nous intégrons la définition de ces procédures MAPLE à l'intérieur d'un module. Le module que nous avons développé définit l'ensemble des procédures MAPLE qui permettent d'accéder aux fonctions des paquetages FFLAS-FFPACK disponibles au sein de notre *wrapper*. Plus précisément, cela comprend les routine pour la multiplication de matrices (`fgemm`), le calcul de rang (`Rank`), le calcul de déterminant (`Det`) la factorisation $LQUP$ (`LUdivine`) et d'inversion de matrice (`Invert`). Dans la suite, nous décrivons en détail l'intégration de la procédure d'inversion au sein de ce module.

Code 3.5 – Module MAPLE pour l'utilisation des routines FFLAS-FFPACK

```
FFPACK:=module()
    export LBInverse; `
    local lbinverse;
    option package, load=create, unload=gc;
    description "Maple interface with FFLAS-FFPACK package";

    create:= proc()
        libname:= "liblbmapleffpack.so"
        lbinverse:= define_external('inverse',MAPLE,LIB=libname);
    end proc

    LBInverse:=proc()
        # calls are:
        #           LBInverse(p,A)   return A^(-1) mod p
        #           LBInverse(p,A,X) return X:=A^(-1) mod p

        local m,n,X;
        m,n:= LinearAlgebra:-Dimension(args[2]);
        if (m<>n) then error "matrix should be square"; end if;

        if (nargs <3) then
            X:=LinearAlgebra:-Modular:-Create(args[1],m,m,float[8]);
            lbinverse(args[1],m,args[2],X);
        else
            lbinverse(args[1],m,args[2],args[3]);
        end if;
    end proc;
end module;
```

Dans ce module, on considère que les coefficients des matrices sont strictement inférieurs au nombre premier définissant le corps fini. L'utilisation de ce module peut être couplée à celle du module Modular proposé par MAPLE et qui permet de définir des matrices à coefficients dans un corps premier. Le calcul de l'inverse d'une matrice modulo p à partir de notre module s'écrit alors de la façon suivante :

```
with(LinearAlgebra:-Modular);
with(FFPACK);
p:=17;
n:=400;
```

```
A:=Create(p,n,n,random,float[8]);
Ainv:=LBInverse(p,A);
```

L'utilisation de la fonction inverse entraîne des calculs en place et donc des effets de bord qui peuvent être parfois gênants. Typiquement, la matrice A dans notre exemple est modifiée car la routine `Invert` du paquetage FFPACK effectue la factorisation LQUP en place. C'est malheureusement le prix à payer pour obtenir les meilleures performances. Ce module MAPLE est actuellement disponible dans la version de développement de la bibliothèque LinBox qui est accessible à partir d'un serveur CVS[36].

3.4.3 Intégration et utilisation dans LinBox

L'utilisation de routines rapides sur les corps finis est un facteur important pour les performances de la bibliothèque LinBox. Afin de simplifier l'utilisation des paquetages FFLAS-FFPACK, nous proposons une interface de calcul LinBox pour manipuler ces routines à partir de classes C++ et de domaines de calcul. Pour cela, nous définissons une interface sur trois niveaux. Le premier niveau fournit un ensemble de modèles de données pour représenter les matrices par FFLAS-FFPACK : matrices, sous-matrices, matrices triangulaires, matrices transposées, permutations. Le deuxième niveau propose une interface de calcul pour les routines de factorisation permettant une réutilisation efficace et sûre des résultats qui sont stockés dans une forme compressée (voir §3.2.2). Enfin, le dernier niveau de cette interface définit un domaine de calcul intégrant l'ensemble des routines des paquetages FFLAS-FFPACK de façon transparente pour les modèles de données proposés par notre interface. Les trois niveaux de notre interface correspondent à la gestion des données, à la réutilisation des résultats et à la gestion des calculs.

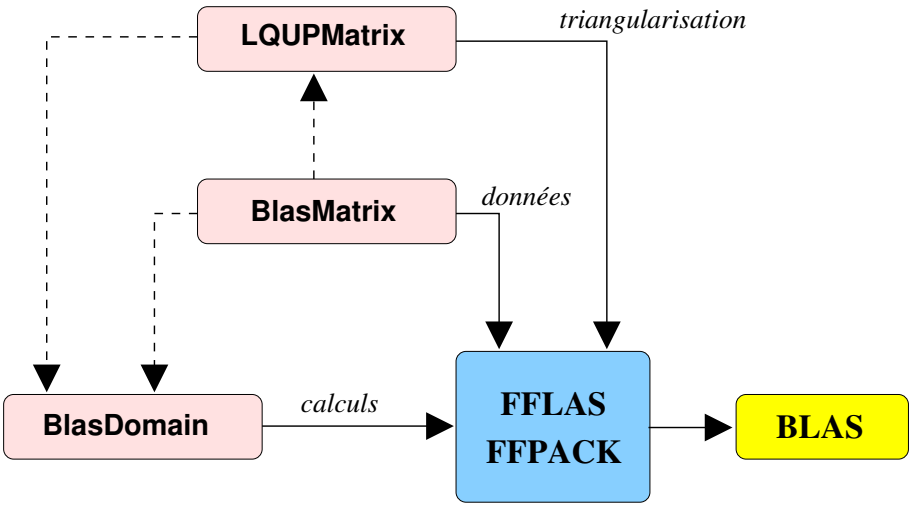

FIG. 3.5 – Interface LinBox pour l'utilisation des BLAS

3.4.3.a Gestion des données

Ces niveaux sont définis en pratique au travers de trois classes C++ comme le décrit la figure 3.5. La première est la classe `BlasMatrix` qui définit une structure de matrices compatible avec

[36] http://www.linalg.org/developer.html

le format linéaire des BLAS. Cette classe définit aussi l'ensemble des caractéristiques nécessaires à l'utilisation des routines FFLAS-FFPACK :

- dimension de la matrice ;
- pointeur sur la zone mémoire des données ;
- *stride* des données.

L'utilisation de ces caractéristiques permet de gérer la notion de sous-matrice de la même manière que les BLAS, à savoir à partir de vues de la matrice initiale. Il suffit pour cela de décaler le pointeur sur le début de la zone mémoire de la sous-matrice et de redéfinir les dimensions en conservant le *stride* initial. L'utilisation des sous-matrices est proposée dans cette classe au travers d'un constructeur de copie sur une matrice non constante. Néanmoins, lorsque la matrice est définie constante (`const`) la construction d'une sous-matrice revient à l'extraction et la copie. Ces constructeurs utilisent deux indices pour définir le premier élément de la sous-matrice et deux autres paramètres pour les dimensions. Une autre caractéristique de ce modèle de matrice est qu'il est compatible avec le modèle des données des matrices de la bibliothèque LinBox. Pour se faire, la classe `BlasMatrix` hérite de la classe `DenseSubmatrix` qui définit toutes les caractéristiques du modèle de base des matrices LinBox. Cette classe définit le stockage de données de façon linéaire en utilisant un vecteur STL (`std::vector`) [62, 35]. La récupération du pointeur sur la zone mémoire des données se fait en récupérant l'adresse mémoire du premier élément de ce vecteur.

Enfin, la classe `BlasMatrix` est accessible à partir de n'importe quelle matrice compatible avec le modèle de base LinBox. La construction d'une `BlasMatrix` à partir de ces matrices se fait grâce à une conversion générique définie par des copies de données vers un format linéaire. Néanmoins, il est possible de spécialiser ces conversions pour éviter la réallocation des données si la structure le permet.

Code 3.6 – Modèle de matrice LinBox pour les routines BLAS

```
template <class _Element>
class BlasMatrix : public DenseSubmatrix<_Element> {
  public:
    typedef _Element Element;
5
  protected:
    size_t    _stride;
    bool      _alloc;
    Element   *_ptr;
10
  public:

    BlasMatrix ();

15  BlasMatrix (size_t m, size_t n);

    template <class Matrix>
    BlasMatrix (const Matrix& A);

20  template <class Matrix>
    BlasMatrix (const Matrix& A,
                const size_t i0, const size_t j0,
                const size_t m, const size_t n);
```

```
25      BlasMatrix (const BlasMatrix<Element>& A);

        BlasMatrix(BlasMatrix<Element>& A,
                   const size_t i0, const size_t j0,
                   const size_t m,  const size_t n);
30
        ~BlasMatrix();

        BlasMatrix<Element>& operator= (const BlasMatrix<Element>& A);

35      Element* getPointer() const;
        Element* getWritePointer();
        size_t   getStride()  const;
    };
```

Le code 3.6 spécifie le modèle de données de la classe `BlasMatrix`. La classe est définie de façon générique sur le type des coefficients de la matrice (`Element`). On distingue un constructeur par défaut et un constructeur permettant d'allouer les données au travers des dimensions de la matrice (cf lignes 13 et 15). Les constructeurs génériques (lignes 18 et 21) permettent de définir les transtypages pour l'ensemble de matrices LinBox. Ces constructeurs entraînent la copie et la réorganisation des données de façon linéaire. L'extraction de sous-matrices est ici possible en donnant les coordonnées du premier élément et les dimensions de la sous-matrice. Le constructeur défini à la ligne 27 permet de définir des sous-matrices n'étant que des vues de la matrice initiale, aucune copie n'est effectuée. La gestion des multiplicités des références ne se fait pas à partir de compteur de références mais au travers du paramètre `_alloc`. Cela signifie que la destruction de la zone mémoire est effectuée uniquement par le propriétaire initial des données. Une vue ne peut en aucun cas désallouer la mémoire. La définition de vues étant principalement intéressante dans l'implantation d'algorithmes récursifs par blocs, nous n'avons pas jugé utile de mettre en place une structure plus complexe de comptage de références pour la gestion des sous-matrices.

Afin de fournir un type de matrice triangulaire compatible avec la structure des `BlasMatrix` nous proposons une spécialisation de cette classe au travers d'une relation d'héritage. Ainsi nous définissons la classe `TriangularBlasMatrix` qui spécialise la classe `BlasMatrix` en ajoutant le caractère triangulaire et le caractère unitaire de la matrice. L'utilisation de ces caractères permet de définir les matrices triangulaires uniquement au travers de vues d'une `BlasMatrix`. Les données qui ne sont pas prises en compte par cette vue ne sont jamais modifiées. Ainsi, on peut par exemple définir une matrice triangulaire inférieure unitaire et une matrice supérieure non unitaire au travers d'une seule structure de données de matrice.

Code 3.7 – Spécialisation de la classe `BlasMatrix` aux matrices triangulaires

```
  class BlasTag {
    public:
    typedef enum{low,up} uplo;
    typedef enum{unit,nonunit} diag;
 5 };

  template <class Element>
  class TriangularBlasMatrix: public BlasMatrix<Element> {
    protected:
10      BlasTag::uplo      _uplo;
        BlasTag::diag      _diag;
```

```
    public:
        TriangularBlasMatrix (const size_t m, const size_t n,
                              BlasTag::uplo x= BlasTag::up,
                              BlasTag::diag y= BlasTag::nonunit);

        TriangularBlasMatrix (const BlasMatrix<Element>& A,
                              BlasTag::uplo x= BlasTag::up,
                              BlasTag::diag y= BlasTag::nonunit);

        TriangularBlasMatrix (BlasMatrix<Element>& A,
                              BlasTag::uplo x= BlasTag::up,
                              BlasTag::diag y= BlasTag::nonunit);

        TriangularBlasMatrix (const TriangularBlasMatrix<Element>& A)

        BlasTag::uplo getUpLo() const { return _uplo;}

        BlasTag::diag getDiag() const { return _diag;}
    };
```

La classe `TriangularBlasMatrix` décrite dans le code 3.7 surcharge la classe `BlasMatrix` en définissant les paramètres `_uplo` et `_diag`. Ces paramètres précisent les caractéristiques que l'on souhaite attribuer à une structure de données. L'utilisation de matrices constantes et non constantes dans les différents constructeurs permet de gérer les copies et les vues de matrices.

Nous introduisons maintenant une autre classe permettant d'ajouter le caractère de transposition à la structure des `BlasMatrix`. Cette classe ne sert qu'à définir le caractère transposé de la matrice au travers d'un type de donnée. En effet, les routines FFLAS-FFPACK gèrent directement les calculs sur les matrices transposées. Le caractère transposé d'une matrice ne doit pas agir sur sa structure, il doit simplement intervenir lors des calculs. Pour cela, nous définissons la classe `TransposedBlasMatrix` qui encapsule les matrices au travers d'une référence et fournit un moyen d'accéder à ces données. Afin de manipuler efficacement les transposées de matrices transposées, nous proposons une spécialisation de cette classe pour des matrices transposées comme le montre le code 3.8.

Code 3.8 – Spécialisation de la classe `BlasMatrix` aux matrices transposées

```
template< class Matrix >
class TransposedBlasMatrix {

    public:
        TransposedBlasMatrix ( Matrix& M ) : _M(M) {}

        Matrix& getMatrix() const { return _M; }

    protected:
        Matrix& _M;
};

template<>
template< class Matrix >
class TransposedBlasMatrix< TransposedBlasMatrix< Matrix > > : public Matrix {
```

```
    public:
        TransposedBlasMatrix ( Matrix& M ) : Matrix(M){}
        TransposedBlasMatrix ( const Matrix& M ) : Matrix(M){}
20 };
```

3.4.3.b Gestion des triangularisations

Le deuxième niveau de notre interface concerne la gestion de la triangularisation de matrices du paquetage FFPACK. Comme nous l'avons expliqué dans les sections 3.2.2 et 3.2.3 la factorisation $LQUP$ des matrices est effectuée en place en utilisant une forme compressée. De plus, cette implantation nécessite le stockage de deux permutations définissant la compression des données et les permutations durant le pivotage. Afin de proposer une structure de données compatible avec les permutations utilisées par les routines de triangularisation FFPACK, nous définissons la classe `BlasPermutation`. Cette classe se base sur un vecteur STL pour stocker une permutation de type `lapack`[37]. La permutation est représentée par un vecteur d'échanges d'éléments deux à deux. Typiquement, la permutation échangeant la première et la troisième colonne d'une matrice d'ordre n est $[3\ 2\ 3\ ...\ n-1]$.

Code 3.9 – Modèle de permutation LinBox pour les routines FFLAS-FFPACK

```
class BlasPermutation {
  protected:
    std::vector<size_t>  _PP;
    size_t               _order;

  public:
    BlasPermutation() {};

    BlasPermutation(const size_t n ): _PP(n), _order( n ) {};

    BlasPermutation(const std::vector<size_t> &P): _PP(P), _order(P.size()) {};

    BlasPermutation( const BlasPermutation &P): _PP(P._PP), _order(P._order) {};

    BlasPermutation& operator=(const BlasPermutation& P){
      _PP = P._PP;
      _order = P._order;
      return *this;
    }

    const size_t* getPointer() const  { return &_PP[0]; }

    size_t* getWritePointer()  { return &_PP[0]; }

    const size_t  getOrder()  const { return _order; }

};
```

[37] http://www.netlib.org/lapack/

En utilisant cette structure de permutation et la classe `BlasMatrix` nous proposons la classe `LQUPMatrix` définissant l'interface de triangularisation avec le paquetage FFLAS-FFPACK. Le but de cette classe est d'encapsuler la factorisation *LQUP* d'une matrice de type `BlasMatrix` et de fournir des moyens simples d'accéder aux données ou de les réutiliser. Afin de conserver la généricité des routines FFLAS-FFPACK, cette classe est définie de façon générique sur les corps finis de la bibliothèque LinBox. Le calcul de la factorisation s'effectue lors de la construction d'une instance de la classe `LQUPMatrix` au travers de la fonction `FFLAPACK::LUdivine`. Nous proposons deux types de constructions. La première travaille directement sur la matrice que l'on souhaite factoriser et la deuxième travaille sur une copie. Comme pour la classe `BlasMatrix`, les copies sont gérées au travers du caractère constant des matrices en utilisant l'attribut `_alloc`. Le code 3.10 décrit les moyens utilisés pour définir cette interface. Nous ne précisons ici que le code des constructeurs, le reste des fonctions servant principalement à la récupération des attributs et à l'extraction des matrices représentant la factorisation.

Code 3.10 – Interface LinBox pour la factorisation de matrices denses sur un corps fini

```
template <class Field>
class LQUPMatrix {

  public:
 5 typedef typename Field::Element Element;

  protected:
      Field                    _F;
      BlasMatrix<Element>    &_LU;
10    BlasPermutation          _P;
      BlasPermutation          _Q;
      size_t                   _m;
      size_t                   _n;
      size_t                _rank;
15    bool                 _alloc;

  public:
      LQUPMatrix (const Field& F, const BlasMatrix<Element>& A)
        : _F(F), _LU(*(new BlasMatrix<Element> (A))) ,
20        _P(A.coldim()), _Q(A.rowdim()), _m(A.rowdim()),
          _n(A.coldim()), _alloc(true)
      {
        _rank= FFLAPACK::LUdivine( _F,
                                   FFLAS::FflasNonUnit,
25                                 _m, _n,
                                   _LU.getPointer(),
                                   _LU.getStride(),
                                   _P.getWritePointer(),
                                   FFLAPACK::FflapackLQUP,
30                                 _Q.getWritePointer() );
      }

      LQUPMatrix (const Field& F, BlasMatrix<Element>& A)
        : _F(F), _LU(A) , _P(A.coldim()), _Q(A.rowdim()),
35        _m(A.rowdim()), _n(A.coldim()), _alloc(false)
      {
        _rank= FFLAPACK::LUdivine( _F,
                                   FFLAS::FflasNonUnit,
                                   _m, _n,
40                                 _LU.getPointer(),
```

```
                                        _LU.getStride(),
                                        _P.getWritePointer(),
                                        FFLAPACK::FflapackLQUP,
                                        _Q.getWritePointer() );
45    }

      TriangularBlasMatrix<Element>& getL(TriangularBlasMatrix<Element>& L) const;

      TriangularBlasMatrix<Element>& getU(TriangularBlasMatrix<Element>& U) const;
50
      BlasMatrix<Element>& getS( BlasMatrix<Element>& S) const;
};
```

Nous avons vu dans la section 3.3 que l'utilisation des triangularisations est centrale dans la mise en place de solutions pour l'algèbre linéaire dense sur un corps finis. Une des applications des triangularisations est la résolution des systèmes linéaires. En effet, à partir d'une triangularisation de matrice on calcule la solution d'un système linéaire soit en résolvant deux systèmes triangulaires si le second membre est un vecteur, soit avec une inversion triangulaire et une résolution de système triangulaire matriciel si le second membre est une matrice. Dans le but de proposer des fonctions de résolution pour des seconds membres matriciels et vectoriels, nous avons développé dans la classe `LQUPMatrix` une interface de résolution de systèmes linéaires à partir de la factorisation d'une matrice. Ainsi, nous définissons les fonctions génériques `left_solve, right_solve, left_Lsolve, right_Lsolve, left_Usolve, right_Usolve` qui respectivement calculent une solution aux systèmes $LQUPX = B$, $XLQUP = B$, $LX = B$, $XL = B$, $UX = B$ et $XU = B$. Afin de définir une interface de résolution pour les différents types d'opérandes possibles, nous utilisons la notion de spécialisation de fonctions génériques. Malheureusement, l'utilisation de spécialisations de fonctions génériques ne nous permet pas de conserver la généricité sur les corps finis. En effet la spécialisation partielle de fonctions génériques n'est pas encore disponible en C++ bien que la question ait été soulevée dans les révisions de la norme[38]. Une alternative classique que nous employons est d'utiliser des fonctions objets génériques pour définir ces fonctions. On obtient ainsi le code suivant pour l'implantation de la fonction `left_solve`

```
template <class Operand>
Operand& LQUPMatrix::left_solve(Operand& X, const Operand& B) const {
  return FactorizedMatrixLeftSolve<Field,Operand>()( _F, *this, X, B );
}
```

La spécialisation partielle de ces fonctions objets sur des type d'opérandes compatibles avec le modèle de données des `BlasMatrix` permet de proposer une interface générique pour les routines de résolution des paquetages FFLAS-FFPACK. Nous décrivons dans le code 3.11 la classe de fonction objet `FactorizedMatrixLeftSolve` et des spécialisations pour les opérandes de type `BlasMatrix` et `std::vector`. On peut facilement utiliser le type `std::vector` dans les routines FFLAS-FFPACK du fait qu'il propose une structure de données linéaire, un accès à sa dimension `size()` et un accès au début de sa zone mémoire (adresse de la première composante).

[38] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#229

Code 3.11 – Implantation des systèmes linéaires au travers de la classe `LQUPMatrix`

```
template <class Field, class Operand>
class FactorizedMatrixLeftSolve {
  public:

  Operand& operator() ( const Field                  &F,
                        const LQUPMatrix<Field>      &A,
                        Operand                      &X,
                        const Operand                &B ) const;

  Operand& operator() ( const Field                  &F,
                        const LQUPMatrix<Field>      &A,
                        Operand                      &B ) const;
};

template <class Field>
class FactorizedMatrixLeftSolve<Field, BlasMatrix<typename Field::Element> > {
  public:

  BlasMatrix<typename Field::Element>&
  operator() ( const Field                                  &F,
               const LQUPMatrix<Field>                      &A,
               BlasMatrix<typename Field::Element>          &X,
               const BlasMatrix<typename Field::Element> &B ) const
  {
    linbox_check( A.coldim() == A.rowdim() );
    linbox_check( A.coldim() == B.rowdim() );
    linbox_check( A.getrank() == B.rowdim() );

    X = B;
    return (*this)( F, A, X );
  }

  BlasMatrix<typename Field::Element>&
  operator() ( const Field                              &F,
               const LQUPMatrix<Field>                  &A,
               BlasMatrix<typename Field::Element>      &B ) const
  {
    size_t m = B.rowdim();
    size_t n = B.coldim();
    linbox_check( A.coldim() == A.rowdim() );
    linbox_check( A.coldim() == m );
    linbox_check( A.getrank() == m );

    typename Field::Element one;
    F.init( one, 1UL );

    // B<- L^(-1)B
    FFLAS::ftrsm( F, FFLAS::FflasLeft, FFLAS::FflasLower,
                  FFLAS::FflasNoTrans, FFLAS::FflasUnit,
                  m, n, one,
                  A.getPointer(), A.getStride(),
                  B.getPointer(), B.getStride() );

    // B<- U^(-1)B
    FFLAS::ftrsm( F, FFLAS::FflasLeft, FFLAS::FflasUpper,
                  FFLAS::FflasNoTrans, FFLAS::FflasNonUnit,
                  m, n, one,
```

```
                    A.getPointer(), A.getStride(),
                    B.getPointer(), B.getStride() );

    // B<- P^(-1)B
    FFLAPACK::applyP( F, FFLAS::FflasLeft, FFLAS::FflasTrans,
                      n, 0, m,
                      B.getPointer(), B.getStride(),
                      A.getP().getPointer() );

    return B;
  }

  template <class Field>
  class FactorizedMatrixLeftSolve<Field,std::vector<typename Field::Element> > {
    public:

    std::vector<typename Field::Element>&
    operator() ( const Field                                  &F,
                 const LQUPMatrix<Field>                      &A,
                 std::vector<typename Field::Element>         &x,
                 const std::vector<typename Field::Element>   &b ) const
    {
      linbox_check( A.coldim() == A.rowdim() );
      linbox_check( A.coldim() == b.size() );
      linbox_check( A.getrank() == A.rowdim() );

      x = b;
      return (*this)( F, A, x );
    }

    std::vector<typename Field::Element>&
    operator() ( const Field                              &F,
                 const LQUPMatrix<Field>                  &A,
                 std::vector<typename Field::Element>     &b ) const
    {

      linbox_check( A.coldim() == A.rowdim() );
      linbox_check( A.coldim() == b.size() );
      linbox_check( A.getrank() == A.rowdim() );

      typename Field::Element one;
      F.init( one, 1UL );

      // b<-L^(-1)b
      FFLAS::ftrsv(F,FFLAS::FflasLower,FFLAS::FflasNoTrans,FFLAS::FflasUnit,
                   b.size(), A.getPointer(), A.getStride(), &b[0], 1 );

      // b<-U^(-1)b
      FFLAS::ftrsv(F,FFLAS::FflasUpper,FFLAS::FflasNoTrans,FFLAS::FflasNonUnit,
                   b.size(), A.getPointer(), A.getStride(), &b[0], 1 );

      // b<-P^(-1)b
      FFLAPACK::applyP(F,FFLAS::FflasLeft,FFLAS::FflasTrans,
                       1, 0, b.size(), &b[0], b.size(), A.getP().getPointer() );

      return b;
    }
  };
```

3.4.3.c Domaine de calcul

Enfin le dernier niveau de notre interface définit un domaine de calcul fournissant l'interface complète avec les routines FFLAS-FFPACK. Ce domaine s'appuie sur les mêmes concepts de spécialisation de fonctions objets que la classe `LQUPMatrix` pour fournir des implantations spécialisées qui restent génériques sur les corps finis. L'ensemble des fonctions disponibles sont le calcul du rang, le calcul du déterminant, la résolution de système linéaire, le calcul du polynôme minimal, le calcul du polynôme caractéristique et la multiplication-addition avec mise à l'échelle. Nous présentons dans le code 3.12 l'implantation de notre interface.

Code 3.12 – Interface LinBox pour les routines FFLAS-FFPACK

```
template <class Field>
class BlasMatrixDomain {

  private:
    Field  _F;
    Element _One;
    Element _Zero;
    Element _MOne;

  public:

    typedef typename Field::Element          Element;

    // Constructor of BlasDomain.
    BlasMatrixDomain (const Field& F ) : _F(F)
    { F.init(_One,1UL); F.init(_Zero,0UL);F.init(_MOne,-1L);}

    // Copy constructor
    BlasMatrixDomain (const BlasMatrixDomain<Field> & BMD)
     : _F(BMD._F), _One(BMD._One), _Zero(BMD._Zero), _MOne(BMD._MOne) {}

    // Field accessor
    Field& field() {return _F;}

    // C = A*B
    template <class Operand1, class Operand2, class Operand3>
    Operand1& mul(Operand1& C, const Operand2& A, const Operand3& B) const
    {return BlasMatrixDomainMul<Field,Operand1,Operand2,Operand3>()(_F,C,A,B);}

    // A = A*B
    template <class Operand1, class Operand2>
    Operand1& mulin_left(Operand1& A, const Operand2& B ) const
    {return BlasMatrixDomainMulin<Field,Operand1,Operand2>()(_F,A,B);}

    // B = A*B
    template <class Operand1, class Operand2>
    Operand2& mulin_right(const Operand1& A, Operand2& B ) const
    {return BlasMatrixDomainMulin<Field,Operand2,Operand1>()(_F,A,B);}

    // D= beta.C + alpha.A*B
    template <class Operand1, class Operand2, class Operand3>
```

```
Operand1& muladd(Operand1& D, const Element& beta, const Operand1& C,
const Element& alpha, const Operand2& A, const Operand3& B) const
{
  return
  BlasMatrixDomainMulAdd<Field,
                         Operand1,
                         Operand2,
                         Operand3> () (_F,D,beta,C,alpha,A,B);
}

// C= beta.C + alpha.A*B
template <class Operand1, class Operand2, class Operand3>
Operand1& muladdin(const Element& beta, Operand1& C,
const Element& alpha, const Operand2& A, const Operand3& B) const
{
  return
  BlasMatrixDomainMulAdd<Field,
                         Operand1,
                         Operand2,
                         Operand3> () (_F,beta,C,alpha,A,B);
}

// Ainv=A^(-1)
template <class Matrix>
Matrix& inv( Matrix &Ainv, const Matrix &A) const
{BlasMatrixDomainInv<Field,Matrix>()(_F,Ainv,A);
  return Ainv;
}

// Ainv=A^(-1) (A is modified)
template <class Matrix>
Matrix& invin( Matrix &Ainv, Matrix &A) const
{BlasMatrixDomainInv<Field,Matrix>()(_F,Ainv,A);
  return Ainv;
}

// rank(A)
template <class Matrix>
unsigned int rank(const Matrix &A) const
{return BlasMatrixDomainRank<Field,Matrix>()(_F,A);}

// rank(A) (A is modified)
template <class Matrix>
unsigned int rankin(Matrix &A) const
{return BlasMatrixDomainRank<Field, Matrix>()(_F,A); }

// det(A)
template <class Matrix>
Element det(const Matrix &A) const
{return BlasMatrixDomainDet<Field, Matrix>()(_F,A);}

// det(A) (A is modified)
template <class Matrix>
Element detin(Matrix &A) const
{return BlasMatrixDomainDet<Field, Matrix>()(_F,A);}

// AX=B
template <class Operand, class Matrix>
Operand& left_solve (Operand& X, const Matrix& A, const Operand& B) const
```

```
    {return BlasMatrixDomainLeftSolve<Field,Operand,Matrix>()(_F,X,A,B);}

    // AX=B , (B<-X)
    template <class Operand,class Matrix>
    Operand& left_solve (const Matrix& A, Operand& B) const
    {return BlasMatrixDomainLeftSolve<Field,Operand,Matrix>()(_F,A,B);}

    // XA=B
    template <class Operand, class Matrix>
    Operand& right_solve (Operand& X, const Matrix& A, const Operand& B) const
    {return BlasMatrixDomainRightSolve<Field,Operand,Matrix>()(_F,X,A,B);}

    // XA=B , (B<-X)
    template <class Operand, class Matrix>
    Operand& right_solve (const Matrix& A, Operand& B) const
    {return BlasMatrixDomainRightSolve<Field,Operand,Matrix>()(_F,A,B);}

    // minimal polynomial computation
    template <class Polynomial, class Matrix>
    Polynomial& minpoly( Polynomial& P, const Matrix& A ) const
    {return BlasMatrixDomainMinpoly<Field, Polynomial, Matrix>()(_F,P,A);}

    // characteristic polynomial computation
    template <class Polynomial,template<class> class List,class Matrix >
    List<Polynomial>& charpoly(List<Polynomial>& P,const Matrix& A) const
    {
      return BlasMatrixDomainCharpoly<Field,
                                      Polynomial,
                                      List,
                                      Matrix >  ()  (_F,P,A);
    }
  };
```

Les fonctions de calcul de rang, de déterminant et d'inverse ne sont définies que pour des matrices de type `BlasMatrix`. La définition de ces calculs sur d'autres type de matrices denses n'a aucun intérêt du fait que la classe `BlasMatrix` sert d'interface pour ces calculs. L'opération de multiplication et ses dérivées sont disponibles pour plusieurs types d'opérandes de façon symétrique (A×B et B×A) :

- `BlasMatrix` × `BlasMatrix` ⟹ `BlasMatrix`
- `BlasMatrix` × `TriangularBlasMatrix` ⟹ `BlasMatrix`
- `BlasMatrix` × `BlasPermutation` ⟹ `BlasMatrix`
- `BlasMatrix` × `TransposedBlasMatrix<BlasPermutation>` ⟹ `BlasMatrix`
- `BlasMatrix` × `std::vector` ⟹ `std::vector`
- `TriangularBlasMatrix` × `BlasPermutation` ⟹ `BlasMatrix`
- `BlasPermutation` × `std::vector` ⟹ `std::vector`
- `TransposedBlasMatrix<BlasPermutation>` × `std::vector` ⟹ `std::vector`

Les implantations de ces spécialisations sont basées sur les routines FFLAS correspondantes. On utilise la fonction `FFLAS::fgemm` pour définir l'ensemble des fonctions de multiplication et de multiplication-addition avec ou sans mise à l'échelle pour les matrices denses.

Code 3.13 – Spécialisations pour la multiplication de matrices denses `BlasMatrix`

```
template< class Field, class Operand1, class Operand2, class Operand3>
class BlasMatrixDomainMul {
  public:
     Operand1 &operator() (const Field      &F,
                           Operand1         &C,
                           const Operand2   &A,
                           const Operand3   &B) const
     {
       typename Field::Element zero, one;
       F.init( zero, 0UL );
       F.init( one, 1UL );
       return BlasMatrixDomainMulAdd<Field,
                                     Operand1,
                                     Operand2,
                                     Operand3> () ( F, zero, C, one, A, B );
                                   }
 };

// specialisation D=beta.C + alpha A.B for BlasMatrix
class BlasMatrixDomainMulAdd<Field,
                             BlasMatrix<typename Field::Element>,
                             BlasMatrix<typename Field::Element>,
                             BlasMatrix<typename Field::Element> > {
   public:
   typedef typename Field::Element Element;

   BlasMatrix<Element>& operator()(const Field                    &F,
                                   BlasMatrix<Element>            &D,
                                   const Element                 &beta,
                                   const BlasMatrix<Element>      &C,
                                   const Element                &alpha,
                                   const BlasMatrix<Element>      &A,
                                   const BlasMatrix<Element>      &B) const
   {
     linbox_check( A.coldim() == B.rowdim());
     linbox_check( C.rowdim() == A.rowdim());
     linbox_check( C.coldim() == B.coldim());
     linbox_check( D.rowdim() == C.rowdim());
     linbox_check( D.coldim() == C.coldim());

     D=C;

     FFLAS::fgemm( F, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans,
                   C.rowdim(), C.coldim(), A.coldim(),
                   alpha,
                   A.getPointer(), A.getStride(),
                   B.getPointer(), B.getStride(),
                   beta,
                   D.getPointer(), D.getStride());
     return D;
   }
 };
```

La spécialisation des fonctions de multiplication pour des matrices triangulaires est un peu différente de celle des matrices denses. En effet, l'implantation de la multiplication de matrices triangulaires peut se faire totalement en place. Pour cela, il suffit d'utiliser une implantation récursive. Comme on a

$$\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_3 & A_3B_4 \end{bmatrix}$$

il nous suffit de calculer, dans l'ordre,
$[B_1\ B_2] \leftarrow A_1 \times [B_1\ B_2]$
$[B_1\ B_2] \leftarrow [B_1\ B_2] + A_2 \times [B_3\ B_4]$
$[B_3\ B_4] \leftarrow A_3 \times [B_3\ B_4]$

La routine `FFLAS::ftrmm` utilise cette implantation récursive pour effectuer le produit d'une matrice triangulaire avec une matrice dense. Du fait de l'implantation en place de cette multiplication, notre interface propose une spécialisation pour la multiplication en place à partir de la fonction objet `BlasMatrixDomainMulin`. L'implantation de la multiplication avec variable de retour s'appuie sur cette version en place. Nous proposons ici les spécialisations pour la multiplication de matrices triangulaires par des matrices denses de type `BlasMatrix`. Nous définissons deux exemplaires de ces spécialisations afin de gérer la non commutativité de cette multiplication.

Code 3.14 – Spécialisation pour la multiplication de matrices triangulaires `BlasMatrix`

```
template<class Field>
class BlasMatrixDomainMulin<Field,
                            BlasMatrix<typename Field::Element>,
                            TriangularBlasMatrix<typename Field::Element> >{
 public:
 typedef typename Field::Element Element;

 BlasMatrix<Element>& operator()(const Field& F,
                                 BlasMatrix<Element>& A,
                                 const TriangularBlasMatrix<Element>& B) const
 {
   typename Field::Element one;
   F.init(one, 1UL);
   linbox_check( A.coldim() == B.rowdim() );

   FFLAS::ftrmm(F,
                FFLAS::FflasRight,
                (B.getUpLo()==BlasTag::up)?
                      FFLAS::FflasUpper : FFLAS::FflasLower,
                FFLAS::FflasNoTrans,
                (B.getDiag()==BlasTag::unit)?
                      FFLAS::FflasUnit : FFLAS::FflasNonUnit,
                A.rowdim(),
                A.coldim(),
                one,
                B.getPointer(), B.getStride(),
                A.getPointer(), A.getStride() );
   return A;
 }
```

```
  BlasMatrix<Element>& operator()(const Field& F,
                                  const TriangularBlasMatrix<Element>& B,
                                  BlasMatrix<Element>& A) const
  {
    linbox_check( B.coldim() == A.rowdim() );
    typename Field::Element one;
    F.init(one, 1UL);
    FFLAS::ftrmm(F,
                 FFLAS::FflasLeft,
                 (B.getUpLo()==BlasTag::up)?
                       FFLAS::FflasUpper : FFLAS::FflasLower,
                 FFLAS::FflasNoTrans,
                 (B.getDiag()==BlasTag::unit)?
                       FFLAS::FflasUnit : FFLAS::FflasNonUnit,
                 A.rowdim(),
                 A.coldim(),
                 one,
                 B.getPointer(), B.getStride(),
                 A.getPointer(), A.getStride() );
    return A;
  }
};
```

On retrouve dans cette implantation les paramètres `FflasLeft` et `FflasRight` qui permettent à la fonction `ftrmm` de savoir de quel coté se trouve la matrice triangulaire et donc quelle est la variable de retour.

Nous définissons maintenant la multiplication par une opérande de type BlasPermutation. Cette multiplication entraîne la permutation des lignes ou des colonnes suivant l'ordre des opérandes. Plusieurs types de spécialisation sont définis pour gérer les permutations de lignes, de colonnes et la multiplication par la permutation transposée. L'ensemble de ces spécialisations sont implantées en utilisant la fonction `applyP` proposée par le paquetage FFPACK. Comme pour la multiplication par une matrice triangulaire, cette opération se fait totalement en place. C'est donc à partir de la fonction objet `BlasMatrixDomainMulin` que nous proposons nos spécialisations.

Code 3.15 – Spécialisations pour appliquer une permutation à une matrice `BlasMatrix`

```
template<class Field>
class BlasMatrixDomainMulin<Field,
                            BlasMatrix<typename Field::Element>,
                            BlasPermutation > {
 public:
  typedef typename Field::Element Element;

  BlasMatrix<Element>& operator()(const Field& F,
                                  BlasMatrix<Element>& A,
                                  const BlasPermutation& B) const
  {
    linbox_check( A.coldim() == B.getOrder() );
    FFLAPACK::applyP(F,
                     FFLAS::FflasRight,
                     FFLAS::FflasNoTrans,
                     A.rowdim(), 0, A.coldim(),
```

```
                    A.getPointer(), A.getStride(), B.getPointer() );
    return A;
  }

  BlasMatrix<Element>& operator()(const Field& F,
                                  const BlasPermutation& B,
                                  BlasMatrix<Element>& A) const
  {
    linbox_check( A.rowdim() == B.getOrder() );
    FFLAPACK::applyP(F,
                    FFLAS::FflasLeft,
                    FFLAS::FflasNoTrans,
                    A.coldim(), 0, A.rowdim(),
                    A.getPointer(), A.getStride(), B.getPointer() );
    return A;
  }

};

template<class Field>
class BlasMatrixDomainMulin<Field,
                            BlasMatrix<typename Field::Element>,
                            TransposedBlasMatrix<BlasPermutation> > {
  public:
  typedef typename Field::Element Element;

  BlasMatrix<Element>& operator()
                      (const Field& F,
                       BlasMatrix<Element>& A,
                       const TransposedBlasMatrix<BlasPermutation>& B) const
  {
    linbox_check( A.coldim() == B.getMatrix().getOrder() );
    FFLAPACK::applyP(F,
                    FFLAS::FflasRight,
                    FFLAS::FflasTrans,
                    A.rowdim(), 0, A.coldim(),
                    A.getPointer(), A.getStride(), B.getMatrix().getPointer() );
    return A;
  }

  BlasMatrix<Element>& operator()(const Field& F,
                                  const TransposedBlasMatrix<BlasPermutation>& B,
                                  BlasMatrix<Element>& A) const
  {
    linbox_check( A.rowdim() == B.getMatrix().getOrder() );
    FFLAPACK::applyP(F,
                    FFLAS::FflasLeft,
                    FFLAS::FflasTrans,
                    A.coldim(), 0, A.rowdim(),
                    A.getPointer(), A.getStride(), B.getMatrix().getPointer() );
    return A;
  }
};
```

Comme pour la multiplication, les fonctions de résolution de systèmes linéaires sont spécialisées pour plusieurs types de matrices et de seconds membres :

- matrice : `BlasMatrix, TriangularBlasMatrix`
- second membre : `BlasMatrix, std::vector`

L'implantation des résolutions de systèmes linéaires pour des matrices denses s'appuie sur la classe `LQUPMatrix` alors que la résolution faisant intervenir des matrices triangulaires s'appuie sur la routine `FFLAS::ftrsm` pour des seconds membres matriciels et la routine `FFLAS::ftrsv` pour des seconds membres vectoriels. L'ensemble de ces résolutions sont effectuées en place dans le second membre. Les fonctions ayant une variable de retour différente des opérandes sont implantées aux travers d'une copie et de la résolution en place. Ainsi, on obtient les spécialisations suivantes pour les matrices denses et les matrices triangulaires.

Code 3.16 – Spécialisations pour la résolution de systèmes linéaires

```
template <class Field>
class BlasMatrixDomainLeftSolve<Field,
                                BlasMatrix<typename Field::Element>,
                                BlasMatrix<typename Field::Element> > {
  public:
  typedef typename Field::Element Element;

  BlasMatrix<Element>& operator() (const Field                  &F,
                                   BlasMatrix<Element>          &X,
                                   const BlasMatrix<Element>    &A,
                                   const BlasMatrix<Element>    &B) const
  {
    LQUPMatrix<Field> LQUP(F,A);
    LQUP.left_solve(X,B);
    return X;
  }
};

BlasMatrix<Element>& operator()(const Field                          &F,
                                const TriangularBlasMatrix<Element> &A,
                                BlasMatrix<Element>                  &B) const
  {
    linbox_check( A.rowdim() == A.coldim());
    linbox_check( A.coldim() == B.rowdim());
    typename Field::Element _One;
    F.init(_One,1UL);

    switch (A.getUpLo()) {
      case BlasTag::up:
        switch(A.getDiag()) {
          case BlasTag::unit:
            {FFLAS::ftrsm( F,
                           FFLAS::FflasLeft,FFLAS::FflasUpper,
                           FFLAS::FflasNoTrans,FFLAS::FflasUnit,
                           A.rowdim(), B.coldim(),_One,
                           A.getPointer(),A.getStride(),
                           B.getPointer(),B.getStride());
            break;}

          case BlasTag::nonunit:
            {FFLAS::ftrsm( F,
                            FFLAS::FflasLeft,FFLAS::FflasUpper,
```

```
                                FFLAS::FflasNoTrans,FFLAS::FflasNonUnit,
                                A.rowdim(), B.coldim(),_One,
                                A.getPointer(),A.getStride(),
                                B.getPointer(),B.getStride());
            break;}

          default:
            throw LinboxError ("Error in BlasMatrixDomain");
        }
        break;

      case BlasTag::low:
        switch(A.getDiag()) {
          case BlasTag::unit:
            {FFLAS::ftrsm( F,
                                FFLAS::FflasLeft,FFLAS::FflasLower,
                                FFLAS::FflasNoTrans,FFLAS::FflasUnit,
                                A.rowdim(), B.coldim(),_One,
                                A.getPointer(),A.getStride(),
                                B.getPointer(),B.getStride());
            break;}

          case BlasTag::nonunit:
            {FFLAS::ftrsm( F,
                                FFLAS::FflasLeft,FFLAS::FflasLower,
                                FFLAS::FflasNoTrans,FFLAS::FflasNonUnit,
                                A.rowdim(), B.coldim(),_One,
                                A.getPointer(),A.getStride(),
                                B.getPointer(),B.getStride());
            break;}

          default:
            throw LinboxError ("Error in BlasMatrixDomain");
        }
        break;

      default:
        throw LinboxError ("Error in BlasMatrixDomain");
    }
    return B;
  }
};
```

Grâce à toutes ces spécialisations, l'utilisation des routines numériques à partir du domaine de calcul `BlasMatrixDomain` permet de définir des codes de haut niveau très robustes qui sont basés sur des calculs très performants. De plus, comme notre interface s'appuie sur les routines des paquetages FFLAS-FFPACK, l'ensemble des codes définit à partir de cette interface pourront bénéficier des améliorations futures de ces paquetages. En particulier, comme l'ensemble des routines FFLAS-FFPACK sont développées autour du produit de matrice, si l'on améliore la routine de multiplication de matrices alors l'ensemble des routines et des codes définis à plus haut niveau bénéficieront aussi de ces améliorations. Nous verrons par exemple dans le chapitre suivant comment l'utilisation de cette interface nous a permis de définir des algorithmes haut niveau pour la résolution de systèmes linéaires entier bénéficiant des très bonnes performances des routines numériques BLAS.

En conclusion de ce chapitre, nous pouvons dire que l'approche qui consiste à réutiliser les routines numériques BLAS pour le calcul en algèbre linéaire sur les corps finis s'avère très satisfaisante et permet de fournir les meilleures implantations actuelles. Le développement des paquetages FFLAS-FFPACK a permis de proposer une couche bas niveau complète et performante pour les problèmes de base de l'algèbre linéaire. L'utilisation d'algorithmes basés sur le produit de matrices permet de bénéficier des améliorations futures des BLAS ou d'autres implantations pour ce problème. Notre interface avec le logiciel MAPLE permet de proposer des codes très performants pour des problèmes fondamentaux comme le déterminant ou l'inverse. L'utilisation de ces solutions dans la mise en place de codes MAPLE devrait permettre d'améliorer considérablement les performances pour des calculs conséquents, ce qui est généralement un facteur limitant dans l'utilisation de ce type de logiciel. Enfin, le développement d'un domaine de calcul haut niveau dans la bibliothèque LinBox permet à la fois de bénéficier des meilleures performances actuelles pour les opérations d'algèbre linéaire sur un corps finis mais aussi de définir des codes simples et robustes qui bénéficient de ces performances. Nous pensons en particulier aux implantations d'algorithme pour les problèmes d'algèbre linéaire sur les entiers et sur les polynômes.

Chapitre 4

Systèmes linéaires entiers

Sommaire

4.1 Solutions rationnelles 122
4.1.1 Développement p-adique de la solution rationnelle 122
4.1.2 Reconstruction de la solution rationnelle 123
4.1.3 Algorithme complet 124
4.2 Interface pour la résolution des systèmes linéaires entiers 125
4.2.1 RationalSolver 126
4.2.2 LiftingContainer et LiftingIterator 128
4.2.3 RationalReconstruction 130
4.3 Algorithme de Dixon 135
4.3.1 Cas non singulier 137
4.3.2 Cas singulier et certificat d'inconsistance 139
4.3.3 Solutions aléatoires 144
4.3.4 Optimisations et performances 145
4.4 Solutions diophantiennes 150
4.4.1 Approche proposée par Giesbrecht 151
4.4.2 Certificat de minimalité 153
4.4.3 Implantations et performances 156

Dans ce chapitre, nous nous intéressons à la résolution d'équations linéaires à coefficients entiers. Étant donnés une matrice $A \in \mathbb{Z}^{m\times n}$ et un vecteur $b \in \mathbb{Z}^m$, on cherche à calculer, si c'est possible, un vecteur $x \in \mathbb{Q}^n$ tel que $Ax = b$. Ce problème est classique en mathématiques et il intervient dans beaucoup d'algorithmes d'algèbre linéaire sur les entiers. Ce type de résolution intervient en particulier dans les calculs de la forme de Smith, ce qui permet d'obtenir la plupart des meilleurs algorithmes d'algèbre linéaire sur les entiers [77]. Nous renvoyons le lecteur à [34, 70, 33, 75] pour le calcul de la forme de Smith et à [86, 51] pour un survol des complexités binaires en algèbre linéaire exacte. Une approche classique pour résoudre ces systèmes linéaires est de se ramener à des calculs sur les corps finis grâce au théorème des restes chinois [36, §5.4, page 102]. Néanmoins, cette approche ne permet pas d'obtenir les meilleures complexités. En effet, en 1982 Dixon a proposé un algorithme permettant d'obtenir une complexité en $\tilde{O}(n^3)$ [22] alors que l'approche par restes chinois conduit à une complexité en $\approx n^4$ pour des systèmes à n équations. La notation $\tilde{O}(n^3)$ signifie ici $O(n^3)$ à des facteurs logarithmiques près. L'algorithme proposé par Dixon se base sur un développement p-adique de la solution rationnelle. Ces travaux sont une généralisation au cas entier des développements x-adiques utilisés en 1979 par Moenck et Carter pour la résolution de systèmes polynomiaux univariés [57]. Le principe de l'algorithme de Dixon est de calculer un développement p-adique de la solution puis de calculer la solution à partir de reconstruction de fractions. L'utilisation de techniques par blocs dans cet algorithme a permis à Storjohann de proposer un algorithme pour résoudre les systèmes linéaires entiers avec la même complexité que la multiplication de matrice entiére [77], c'est-à-dire de l'ordre de n^ω.

Un cas particulier de résolution de système linéaire entier est le calcul de solutions entières. On s'intéresse alors à calculer un vecteur $y \in \mathbb{Z}^n$ tel que $Ay = b$. Ce type de solutions est appelé solution diophantienne. Le calcul de telles solutions a des applications en théorie des groupes [63], théorie des nombres [15] et en programmation linéaire. Une approche classique pour calculer une solution diophantienne est d'utiliser la forme de Smith [63, chap. 2-21]. Néanmoins, cette technique est trop coûteuse car la décomposition sous forme de Smith entraîne l'utilisation de coefficients dont la taille est de l'ordre de $O(n)$ bits. Ceci entraîne un coût de $\tilde{O}(n^4)$ opérations binaires pour calculer une solution diophantienne du système linéaire entier. Une technique proposée en 1997 par Giesbrecht [39] permet d'améliorer cette complexité à $\tilde{O}(n^3)$ en utilisant des combinaisons de solutions rationnelles.

Notre objectif dans ce chapitre est de proposer une implantation générique de résolution de systèmes linéaires entiers qui soit facilement paramétrable et qui soit la plus performante possible. Pour cela, nous utilisons la méthode de résolution par approximations p-adiques et l'idée de combinaison des solutions de Giesbrecht qui autorisent les meilleures complexités théoriques. Nous verrons aussi que ces méthodes sont très performantes en pratique.

Notre idée est d'utiliser ces deux schémas algorithmiques pour proposer une interface de résolution de systèmes linéaires qui puisse être paramétrée en fonction de l'instance du problème : matrices denses, creuses ou structurées. Pour cela nous considérons plusieurs méthodes algorithmiques pour calculer les chiffres du développement p-adique de la solution. En particulier, cela concerne l'algorithme proposé par Dixon [22] pour les matrices denses et des méthodes itératives de type Wiedemann [91] pour les matrices creuses. L'approche que nous utilisons reste aussi valide pour le cas des matrices structurées de type Toeplitz/Hankel [64]. Nous verrons que l'une des difficultés liées à l'utilisation des développement p-adique concerne la gestion de la singularité des matrices, la gestion de l'inconsistance des systèmes et la certification des calculs qui font intervenir pour la plupart des algorithmes probabilistes.

Le but de notre travail est de fournir une boîte à outils pour résoudre le plus efficacement possible les systèmes linéaires entiers. Pour cela, nous utilisons la bibliothèque LinBox comme

base de développement. Le développement de telles implantations dans cette bibliothèque doit permettre de :

- proposer les meilleures solutions actuelles pour les problèmes d'algèbre linéaire sur les entiers ;
- valider la bibliothèque sur des algorithmes de plus haut niveau.

Dans un premier temps, nous présentons dans la section 4.1 l'approche générale de la résolution de système linéaire sur le corps des rationnels par les développements p-adiques. En particulier, nous présenterons le schéma itératif générique de la méthode pour calculer ces développements en fonction de la résolution sur les corps finis. Nous développerons ensuite les moyens que nous avons mis en place au sein de la bibliothèque LinBox pour implanter la résolution de systèmes linéaires par développement p-adique (§4.2). En particulier, nous proposerons une méthode qui se base sur l'utilisation d'itérateur de données et qui permet d'abstraire le calcul des chiffres du développement. La difficulté est ici la gestion des calculs probabilistes effectués par les algorithmes que nous utiliserons. Nous préciserons dans la section 4.3 comment nous utilisons l'algorithme de Dixon pour résoudre les systèmes linéaires denses. En outre, nous verrons que le problème pour résoudre ces systèmes est la gestion de la singularité et de l'inconsistance. Nous nous appuierons alors sur les travaux de Mulders et Storjohann [61] sur le sujet qui proposent une solution probabiliste très satisfaisante. Enfin, dans la dernière partie, nous étudierons le calcul des solutions entières en utilisant l'approche proposée par Giesbrecht. Plus précisément, nous essaierons de quantifier le nombre de solutions rationnelles nécessaires en pratique pour recouvrir une solution entière. Néanmoins, il se peut qu'il n'existe aucune solution entière au système. Dans ce cas la, nous utiliserons l'approche proposée par Mulders et Storjohann [61] qui consiste à certifier la minimalité des solutions calculées.

Notations

- Pour tout exposant e_1, $\tilde{O}(n^{e_1})$ est équivalent à $O(n^{e_1}\log^{e_2} n)$ pour tout exposant $e_2 > 0$.
- Étant donnée un vecteur $v = [v_i] \in \mathbb{Z}^n$, on définit sa norme Euclidienne par :
  $||v|| = (\sum_{1\leq i\leq n} v_i^2)^{1/2}$.
- Ètant donné un nombre réel β, on définit la partie entière inférieure de β par $\lfloor\beta\rfloor$ et sa partie entière supérieure par $\lceil\beta\rceil$.
- Ètant donnés une matrice $A \in \mathbb{Z}^{n\times n}$, un vecteur $b \in \mathbb{Z}^n$ et un vecteur $x \in \mathbb{Q}^n$ tel que $Ax = b$, les bornes sur le numérateur et le dénominateur de x sont définies en utilisant la borne d'Hadamard et les règles de Cramer [36, pages 706, 465]. Nous utilisons les fonctions $\texttt{numbound}(A, b)$ et $\texttt{denbound}(A, b)$ pour spécifier ces bornes, respectivement du numérateur et du dénominateur de x.

$$\texttt{denbound}(A, b) = \left\lceil \prod_{i=1}^{n} ||A_{*,i}|| \right\rceil,$$

$$\texttt{numbound}(A, b) = \left\lceil \frac{\texttt{denbound}(A, b) \times ||b||}{\min_{1\leq i\leq n}(||A_{*,i}||)} \right\rceil.$$

4.1 Solutions rationnelles

Nous nous intéressons dans cette section au calcul des solutions rationnelles d'un système linéaire entier. Pour cela, nous utilisons la méthode proposée par Moenck et Carter en 1979 [57] et adaptée en 1982 par Dixon [22] pour le cas entier. Cette méthode consiste à calculer les solutions des systèmes linéaires entiers en passant par un développement p-adique et en utilisant la théorie des fractions continues pour reconstruire la solution. Afin de pouvoir réutiliser les algorithmes d'algèbre linéaire sur les corps finis et les implantations que nous avons présentés dans le chapitre 3, nous considérons que la base p-adique des développements est définie à partir d'un nombre premier p.

Soient $A \in \mathbb{Z}^{n\times n}$ une matrice inversible et $b \in \mathbb{Z}^n$. On cherche à calculer $x \in \mathbb{Q}^n$ tel que $Ax = b$. Pour cela, la méthode consiste à choisir un nombre premier p tel que $p \nmid det(A)$ et à calculer l'unique vecteur $y \in \mathbb{Z}^n$ tel que $Ay \equiv b \mod p^k$ et $|y| < p^k$ pour k fixé. En choisissant k suffisamment grand, on sait d'après la théorie des fractions continues que l'on peut retrouver la solution $x \equiv y \mod p^k$ telle que $x \in \mathbb{Q}^n$ à partir de reconstructions de fractions sur p^k et les coefficients de y. On sait que k est suffisamment grand lorsque que $k \geq 2ND$ où N et D sont des majorants pour respectivement le numérateur et le dénominateur des coefficients de x.

En utilisant les règles de Cramer et la borne d'Hadamard [36, pages 706, 465], on peut alors définir précisément ces majorants. On a en particulier $D = \texttt{denbound}(A, b)$ et $N = \texttt{numbound}(A, b)$.

4.1.1 Développement p-adique de la solution rationnelle

La première phase de la méthode consiste à calculer la solution du système modulo p^k. La technique utilisée pour calculer cette solution se rapproche des méthodes numériques itératives avec correction du résidu. Soit $A\bar{y}^{[i]} \equiv b \mod p^i$ tel que $\bar{y}^{[i]} = y^{[0]} + y^{[1]}p + \ldots + y^{[i-1]}p^{i-1}$. On utilise le résidu $(b - A\bar{y}^{[i]})/p^i$ pour calculer le $i$ème chiffre $y^{[i]}$ tel que $A(\bar{y}^{[i]} + p^i y^{[i]}) \equiv b \mod p^{i+1}$.

$$\begin{aligned} Ay^{[0]} &\equiv b \mod p, \\ Ay^{[1]} &\equiv \frac{b - Ay^{[0]}}{p} \mod p, \\ &\ldots \\ Ay^{[k-1]} &\equiv \frac{b - A(y^{[0]} + y^{[1]}p + \ldots + y^{[k-2]}p^{k-2})}{p^{k-1}} \mod p. \end{aligned} \tag{4.1}$$

Les équations (4.1) illustrent le schéma itératif de la méthode de calcul. Néanmoins, afin de limiter le grossissement des seconds membres dans les équations (4.1), les résidus sont calculés successivement à l'aide de mises à jour. Ainsi, on définit le résidu $c_{i+1} = (b - A(y^{[0]} + \ldots + y^{(i)}p^i))/p^{i+1}$ à partir du résidu c_i par la relation

$$c_{i+1} = \frac{c_i - Ay^{(i)}}{p} \quad y^{(i+1)} = A^{-1}c_{i+1} \mod p$$

En utilisant ces mises à jour des résidus, on peut alors définir l'algorithme suivant pour calculer le développement p-adique d'une solution rationnelle à l'ordre k.

Algorithme `PadicLifting`(A,b,p,k)
Entrée : $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^n$, $p \in \mathbb{N}$, $k \in \mathbb{N}$.
Condition : $\det(A) \not\equiv 0 \mod p$

Sortie : $y = A^{-1}b \mod p^k$
Schéma
$c := b$;
$M := 1$;
pour i **de** 0 **à** k-1 **faire**
résoudre $Ax \equiv c \mod p$;
$c := (c - Ax/p$;
$y := y + Mx$;
$M := Mp$;
retourner y ;

Grâce à la mise à jour des résidus à chaque itération, on peut borner la taille des composantes de c tout au long des itérations par $\max_j(|c_j|) \leq n(\alpha+\beta)$ où $\alpha = \max_{i,j}(|A_{i,j}|)$ et $\beta = \max_j(|b_j|)$. Soit la fonction $L(k, n, p)$ définissant le nombre d'opérations binaires de k résolutions successives modulo p de systèmes linéaires de dimension n issus d'une même matrice. Soient $p, \alpha, \beta << n$. On peut alors définir le complexité de l'algorithme `PadicLifting` par le lemme suivant.

Lemme 4.1.1. *Soient $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^n$, $p, k \in \mathbb{N}$. Le nombre d'opérations binaires de l'algorithme* `PadicLifting(A,b,p,k)` *est en $O(kn^\lambda + kn \log n) + L(k, n, p)$, où λ est l'exposant dans la complexité du produit matrice-vecteur.*

La preuve de ce lemme est immédiate en utilisant le fait que les résidus sont de l'ordre de $O(\log n)$ bits et que le produit matrice-vecteur d'ordre n nécessite $O(n^\lambda)$ opérations. Plus précisément, le coût de cet algorithme est de $O(kn^2) + L(k, n, p)$ pour des systèmes denses et de $O(kn \log n) + L(k, n, p)$ pour des systèmes creux ayant de l'ordre de $O(n)$ éléments non nuls.

4.1.2 Reconstruction de la solution rationnelle

La deuxième phase de la résolution des systèmes linéaires entiers consiste à déterminer la solution rationnelle à partir de son développement p-adique. Une approche classique pour reconstruire une fraction a/b à partir d'une paire d'entiers (g, m) telle que $a/b \equiv g \mod m$ est d'appliquer l'algorithme d'Euclide étendu à g et m. En effet, cet algorithme calcule pour deux entiers g, m une suite de triplets (s_j, t_j, r_j) telle que $s_j g + t_j m = r_j$. Si $\text{pgcd}(g, m) = 1$ on a alors $r_j/s_j \equiv g \mod m$.

Pour identifier le triplet $(s_{j'}, t_{j'}, r_{j'})$ tel que $s_{j'} = b$ et $r_{j'} = a$ dans l'algorithme d'Euclide étendu il faut pouvoir assurer une condition d'unicité sur la fraction à reconstruire. Pour cela, Wang [88, lemme 2] propose d'utiliser une borne sur les valeurs absolues de a et de b, à savoir $|a|, |b| < \sqrt{m/2}$. Grâce à cette condition, il y a unicité de la fraction $r_{j'}/s_{j'}$ telle que $|r_j|, |s_j| < \sqrt{m/2}$. Pour récupérer cette fraction il suffit de récupérer le premier triplet (s_j, t_j, r_j) dans l'algorithme d'Euclide étendu qui vérifie $r_j < \sqrt{m/2}$.

Dans le cas de la reconstruction d'une solution rationnelle, on utilise les bornes fixées par les paramètres N et D donnés par les fonctions `numbound` et `denbound` du système linéaire. On peut alors définir l'algorithme suivant pour reconstruire la solution rationnelle à partir de son expression p-adique.

Algorithme `ReconstructSolution(`y`,`p^k`,`N`,`D`)`
Entrée : $y \in \mathbb{Z}^n$ tel que $|y| < p^k$, $N, D \in \mathbb{N}$
Sortie : $x \in \mathbb{Q}^n$ tel que $x_i \equiv y_i \mod p^k$ et $|x_i| < N$, $d_i < D$

Schéma
pour i **de** 1 **à** n **faire**
$a := y_i$; $b := p^k$;
$s_1 := 0$; $s_2 := 1$;
tant que $a \neq 0$ **et** $a > N$ **faire**
$q := b/a$;
$r := b \mod a$;
$a := b$;
$b := r$;
$s_0 := s_2$;
$s_2 := s_1 - qs_2$;
$s_1 := s_0$;
$x_i := a$;
$d_i := s_2$;
retourner $[x_1/d_1, x_2/d_2, \ldots, x_n/d_n]^T$;

Le coût de l'algorithme `ReconstructSolution` est directement lié au coût de l'algorithme d'Euclide étendu (ici la conditionnelle **tant que**) qui est quadratique en la taille des entrées. Les composantes du développement p-adique étant de l'ordre de $O(k)$ bits, on peut donc borner le coût de cet algorithme par $O(nk^2)$ opérations binaires. Néanmoins, cet algorithme ne permet pas d'obtenir la meilleure complexité théorique pour la reconstruction de solutions rationnelles. En effet, la meilleure complexité connue est celle obtenue en utilisant un algorithme de pgcd rapide à la Knuth/Schönhage [53, 71]. On obtient alors une complexité quasi linéaire en la taille des entrées pour reconstruire une composante, ce qui donne un coût pour la reconstruction de la solution rationnelle de $\tilde{O}(nk \log^2 k)$ opérations binaires [89].

4.1.3 Algorithme complet

Nous présentons maintenant le schéma algorithmique complet pour la résolution des systèmes linéaires entiers par développement p-adique. Le principe est de calculer un nombre suffisant de chiffres p-adiques pour retrouver la solution rationnelle à partir de son développement p-adique. Pour cela, nous utilisons les bornes données par les règle de Cramer et la borne d'Hadmard du système linéaire.

Algorithme `PadicSolver`(A,b)
Entrée : $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^n$
condition : $\det(A) \not\equiv 0 \mod p$
Sortie : $x = A^{-1}b$
Schéma
$N :=$ `numbound`(A, b) ;
$D :=$ `denbound`(A, b) ;
$k := \lceil \log(2ND) + \log p \rceil + 1$;
$y :=$`PadicLifting`(A,b,p,k) ;
$x :=$`ReconstructSolution`(y,p^k,N,D) ;
retourner x ;

Le nombre de chiffres du développement p-adique étant de l'ordre de $O(n \log n)$ bits, le nombre d'opérations binaires de l'algorithme `PadicSolver` est en $O(n^{\lambda+1} \log n + n^3 \log^2 n) +$

$L(k, n, p)$. Si on utilise un algorithme rapide pour la reconstruction rationnelle [89, algorithme 4.1] on obtient alors un coût de $\tilde{O}(n^{\lambda+1} \log n + n^2 \log^3 n) + L(k, n, p)$ opération binaire. Ce qui donne en particulier $O(n^3 \log n) + L(k, n, p)$ pour des systèmes linéaires denses et $\tilde{O}(n^2 \log^3 n) + L(k, n, p)$ pour des systèmes linéaires creux ayant $O(n)$ éléments non nuls.

On remarque que la validité de cet algorithme repose sur une base p-adique qui conserve le rang de la matrice A. En pratique le choix d'une telle base ne peut se faire de façon déterministe dans le coût de l'algorithme lui-même. Le choix d'une base aléatoire parmi un ensemble de nombre premier suffisamment grand permet alors de fournir un algorithme Monte Carlo pour calculer de telles solutions rationnelles.

4.2 Interface pour la résolution des systèmes linéaires entiers

Dans cette section nous nous intéressons à définir une interface C++ haut niveau dans LinBox pour la résolution des systèmes linéaires entiers par développement p-adique. Notre objectif est de fournir une implantation de l'algorithme `PadicSolver` qui permette de changer facilement la méthode de calcul des chiffres p-adiques sans avoir à modifier toute l'implantation. La configuration de ces calculs doit permettre de fournir des implantations spécialisées pour la résolution de système linéaire entiers suivant si la structure du système est denses, creuses ou structurées. La difficulté d'un tel développement se situe dans la spécification d'un modèle de calcul de haut niveau suffisamment robuste pour autoriser l'utilisation de calculs spécialisés.

L'une des opération clés des résolutions de systèmes linéaires entiers est le calcul du développement p-adique qui varie suivant la méthode de résolution de système linéaire dans la base p-adique. Notre approche pour permettre une configuration simple de l'algorithme `PadicLifting` est de fournir une implantation dans laquelle les calculs des chiffres du développement p-adique sont asbtraits. En particulier, le but est de fournir une structure de développement p-adique qui permette une telle abstraction. Pour cela, nous proposons de réutiliser l'approche "conteneur/itérateur" développée dans la bibliothèque STL [35, 62]. Ainsi, notre interface peut se diviser en trois modules correspondant aux différentes phases de la résolution. Nous utilisons alors les algorithmes `PadicLifting, ReconstructSolution` et `PadicSolver` pour implanter ces trois modules.

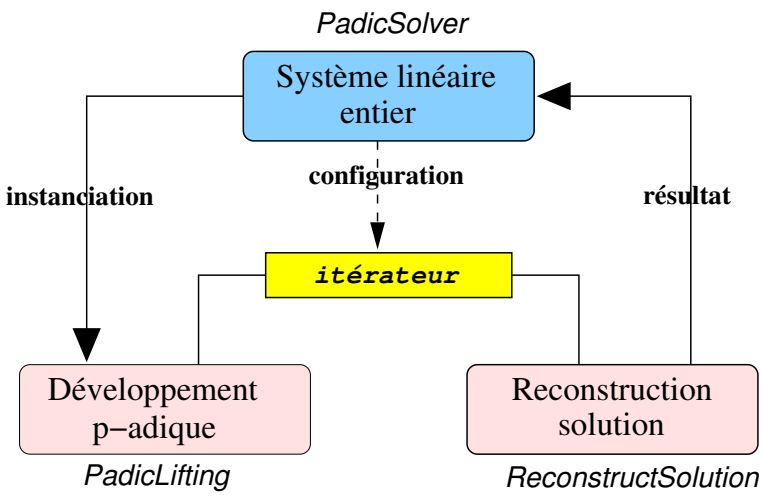

FIG. 4.1 – Interface LinBox pour la résolution des systèmes linéaires entiers

Le premier module correspond à l'algorithme `PadicLifting` et définit l'interface de calcul des développements p-adiques. L'approche "conteneur/itérateur" est ici utilisée pour faciliter la configuration des calculs. En particulier, nous utilisons l'itérateur pour calculer à la volée les chiffres du développement p-adique alors que le conteneur permet de fournir une spécialisation des calculs. Grâce à cette dissociation, la phase de reconstruction de la solution rationnelle est totalement indépendante de la structure des développements p-adiques. Seule la connaissance de l'itérateur est nécessaire pour calculer le développement p-adique. Grâce à la normalisation de l'itérateur on peut donc définir une interface abstraite pour calculer des solutions rationnelles quelque soit la méthode de calcul. Enfin, le dernier module correspond à l'algorithme `PadicSolver` et permet de spécifier les calculs effectués par l'itérateur grâce à des conteneurs spécialisés. Du fait qu'on utilise des algorithmes probabilistes pour calculer une solution, si toutefois il en existe une, ce module sert aussi à valider les résultats obtenus et à certifier l'inconsistance des systèmes.

L'interaction de ces trois modules et le fonctionnement de notre interface sont illustrés par la figure 4.1. Nous définissons notre interface pour la résolution de systèmes linéaires à partir de trois classes C++ : `RationalSolver`, `LiftingContainer` et `RationalReconstruction`. Nous nous intéressons dans les sections suivantes à spécifier le modèle de ces trois classes ainsi que leurs implantations.

4.2.1 RationalSolver

La classe `RationalSolver` définit la partie utilisateur de notre interface. Le principe de cette classe est de définir un domaine de résolution de systèmes linéaires entiers. Afin que notre implantation soit générique, nous définissons les type de données suivants à partir de paramètres *templates* :

- `Ring` : arithmétique des entiers,
- `Field` : arithmétique modulaire de la base p-adique,
- `RandomPrime` : générateur aléatoire de nombres premiers,
- `MethodTraits` : méthode de calcul des chiffres du développement p-adique.

Afin d'assurer la généricité de notre implantation, chacun de ces types de données doit respecter un modèle de base prédéfini. Nous utilisons la structure de corps finis de la section 2.1 pour spécifier le domaine de calcul défini par la base p-adique (`Field`). Pour caractériser le domaine de calcul sur les entiers (`Ring`), nous utilisons un modèle de données similaire à celui des corps finis en ajoutant quelques fonctions nécessaires au calcul dans des idéaux principaux (test de divisibilité, pgcd, ppcm, quotient, reste, reconstruction de fractions). Le modèle de données du générateur aléatoire de nombres premiers (`RandomPrime`) doit encapsuler un type d'entier défini par l'alias `Prime` et une fonction de génération aléatoire `void randomPrime(Prime &p)`. La méthode de calcul des chiffres du développement p-adique (`MethodTraits`) doit être définie à partir d'une structure de "traits" (voir §1.1.3) encapsulant les différents paramètres de calcul de la méthode.

Le code suivant illustre la spécification de la classe `RationalSolver` dans la bibliothèque LinBox.

Code 4.1 – Domaine de résolution de systèmes linéaires entiers

```
template<class Ring, class Field, class RandomPrime, class MethodTraits>
class RationalSolver {
```

```
  protected:
    Ring                          _R;
    RandomPrime                _genprime;
    mutable Prime                 _prime;
    MethodTraits                 _traits;

  public:
    RationalSolver(const Ring                 &r =Ring(),
                   const RandomPrime        &rp =RandomPrime(DEFAULT_PRIMESIZE),
                   const MethodTraits &traits =MethodTraits())
      : _R(r), _genprime(rp), _traits(traits)
    {
      _prime=_genprime.randomPrime();
    }

    RationalSolver(const Prime                &p,
                   const Ring                 &r =Ring(),
                   const RandomPrime        &rp =RandomPrime(DEFAULT_PRIMESIZE),
                   const MethodTraits &traits =MethodTraits())
      : _R(r), _genprime(rp), _prime(p), _traits(traits){}

    void chooseNewPrime() const {_prime = _genprime.randomPrime();}
};
```

La construction d'un domaine de calcul de type `RationalSolver` permet de fixer les arithmétiques utilisées, la méthode de calcul du développement p-adique ainsi que le générateur aléatoire de nombres premiers. Ce générateur permet de définir la base p-adique utilisée pour effectuer les calculs. Il est néanmoins possible de préciser une base spécifique lors de la construction. On remarque que l'attribut `_prime` est défini de façon `mutable`, c'est-à-dire qu'il peut être modifier même si le domaine de calcul est défini constant (`const RationalSolver`). Cela provient du fait qu'il faut pouvoir changer la base p-adique lorsque celle-ci ne vérifie par les conditions de validité de l'algorithme `PadicSolver`; à savoir que la base p-adique conserve le rang de la matrice. Ce changement de base est assuré dans la classe `RationalSolver` par la fonction `ChooseNewPrime()`.

Le but de la classe `RationalSolver` est de définir un domaine de résolution de systèmes linéaires offrant les calculs les plus adaptés à l'instance du système et à la solution recherchée. En particulier, les calculs différent suivant si le système est singulier ou non et suivant si l'on souhaite obtenir une solution aléatoire du système ou non. Afin de couvrir tous ces types de calculs spécifiques nous définissons plusieurs fonctions de résolution dans la classe `RationalSolver` :

- `solve`
- `solveNonsingular`
- `solveSingular`
- `findRandomSolution`

Les fonctions `solveSingular` et `solveNonSingular` doivent être utiliser lorsqu'on connaît à priori le caractère de singularité du système. Si ce caractère n'est pas connu à l'avance, la fonction `solve` sert alors d'interface. La fonction `findRandomSolution` permet quant à elle de calculer une solution aléatoire du système quand celui-ci est singulier. Toutes ces fonctions sont définies de façon générique en fonction des types de matrices et de vecteurs utilisés pour définir le système linéaire. Nous utilisons des paramètres *templates* pour spécifier ces types de données

dans les implantations. Les vecteurs doivent alors respecter les itérateurs de données de la STL tandis que les matrices sont considérées comme des boîtes noires et doivent respecter le modèle de base défini par l'archétype `BlackboxArchetype` [27].

La difficulté dans l'implantation de ces fonctions de résolution provient de la validité de la solution calculée. En particulier, il se peut que les systèmes soient inconsistants et donc qu'aucune solution n'existe. Les calculs étant probabilistes il se peut aussi que le résultat du calcul ne soit pas une solution du système. Afin de gérer tout ces cas de figure et permettre à l'utilisateur de connaître les caractéristiques du résultat, nous utilisons le paramètre de retour des fonctions pour fournir cette information. Pour cela, chacune des fonctions de résolution de la classe `RationalSolver` doit retourner le statut des calculs effectués. Nous utilisons l'énumération `SolverReturnStatus` pour spécifier l'ensemble des statuts possibles.

```
enum SolverReturnStatus {
  SS_OK, SS_FAILED, SS_SINGULAR, SS_INCONSISTENT
};
```

- `SS_OK` $\Longrightarrow$ calcul correct
- `SS_FAILED` $\Longrightarrow$ calcul incorrect
- `SS_SINGULAR` $\Longrightarrow$ système singulier
- `SS_INCONSISTENT` $\Longrightarrow$ système inconsistant

Le principe général de l'implantation des fonctions de résolution consiste à construire le développement p-adique adéquat à la méthode de calcul spécifiée et instancier le module de reconstruction de la solution rationnelle avec ce développement. Nous présenterons l'implantation de ces fonctions dans le cas des systèmes denses dans la suite de ce document (§4.3).

4.2.2 LiftingContainer et LiftingIterator

Les classes `LiftingContainer` et `LiftingIterator` définissent l'interface permettant de manipuler les développements p-adiques uniquement par un itérateur de données, le but étant d'abstraire la structure des développements (matrice, vecteur, base p-adique) de la manière dont on les calcule. L'idée est de fournir un itérateur spécialisé pour calculer les chiffres du développement à la volée. Pour cela nous définissons les deux structures de données suivantes.

- `LiftingContainer` : un conteneur encapsulant les caractéristiques du développement p-adique (matrice, vecteur, base p-adique, longueur, dimension)
- `LiftingIterator` : un itérateur encapsulant un résidu courant et qui calcule à la volée les chiffres du développement p-adique.

Le principe de cette dissociation décrite par la figure 4.2 est de permettre la spécialisation du calcul des chiffres p-adiques à partir de la méthode de calcul la plus adéquate à l'instance du système. Le calcul des résidus étant indépendant de la méthode utilisée, nous les séparons des calculs des chiffres du développement. Grâce à cette séparation, on peut ainsi différencier les calculs dans la base p-adique de ceux sur les entiers. Afin de manipuler les développements p-adiques uniquement à partir de la classe `LiftingIterator`, il nous faut fournir un moyen efficace et générique pour paramétrer les calculs effectués par cet itérateur. Notre approche consiste à utiliser une fonction virtuelle pure pour définir les calculs des chiffres p-adiques dans l'itérateur. Ainsi, en utilisant des conteneurs qui spécifient cette fonction virtuelle en fonction d'une

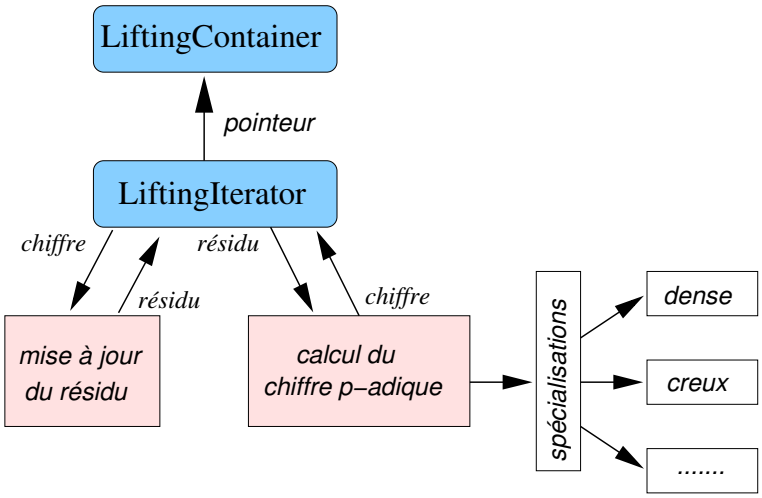

FIG. 4.2 – Principe de l'itérateur de développements p-adiques

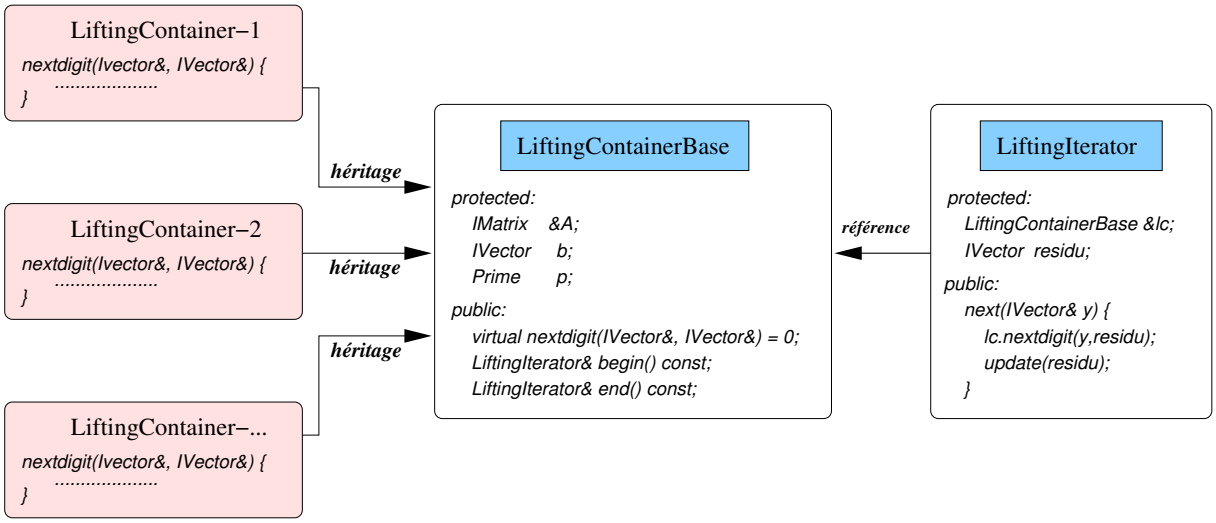

FIG. 4.3 – Interface des conteneurs de développements p-adiques

méthode de calcul on obtient un itérateur configurable. La figure 4.3 illustre plus précisément l'implantation de notre approche.

L'idée est d'utiliser la fonction `next` de l'itérateur pour calculer et récupérer un chiffre du développement. Cette fonction calcule le nouveau chiffre du développement à partir du résidu courant `residu` et de la fonction du conteneur `lc.nexdigit`. Une fois que le nouveau chiffre est calculé, le résidu est mis à jour à partir de la fonction `update` qui est générique à tous les conteneurs. Du fait que la fonction `nextdigit` est définie virtuelle pure dans la classe `LiftingContainerBase` c'est la fonction définie par les classes dérivées qui sera utilisée. Le but de cette classe est double. Elle permet à la fois de spécifier le modèle de base des conteneurs de développement p-adique et de définir l'ensemble des caractéristiques communes à tous les conteneurs. En particulier c'est elle qui définit les fonctions relatives à la création d'itérateurs de début et de fin de structure. De plus, elle permet de spécifier la longueur du développement ainsi que les bornes sur le numérateur et le dénominateur de la solution rationnelle qu'il faudra reconstruire à partir du développement. D'autre part, elle sert d'interface pour la définition de la fonction `nextdigit`. Ainsi, chaque conteneur doit hériter de cette classe afin de pouvoir définir un conteneur valide pour l'itérateur.

À l'instar de la STL, notre implantation est générique et permet de manipuler n'importe quelle structure de développement p-adique définie à partir d'un système linéaire et d'une base p-adique. De même, l'interface des itérateurs est définie de façon générique pour des vecteurs basés sur la STL. Nous proposons dans l'annexe A.1 les codes LinBox définissant l'interface et le modèle de données de ces développements p-adiques. Nous verrons dans 4.3 une spécialisations de ce modèle pour les systèmes denses.

4.2.3 RationalReconstruction

La classe `RationalReconstruction` définit un domaine de calcul pour la reconstruction de solutions rationnelles à partir de développements p-adiques. En particulier, ce domaine est générique pour l'ensemble des développements p-adiques respectant l'interface proposée par la classe `LiftingContainerBase`. Le but de cette classe est de fournir une fonction pour la reconstruction d'une solution rationnelle à partir d'un itérateur de développement p-adique. Le code 4.2 spécifie le modèle de cette classe.

Code 4.2 – Interface de reconstruction rationnelle

```
template< class LiftingContainer >
class RationalReconstruction {

  public:
    typedef typename LiftingContainer::Ring          Ring;
    typedef typename Ring::Element                 Integer;
    typedef typename LiftingContainer::IVector      Vector;
    typedef typename LiftingContainer::Field         Field;
    typedef typename Field::Element                Element;

  protected:
    const LiftingContainer  &_lcontainer;
    Ring                            _r;
    int                     _threshold;
```

```
  public:
     RationalReconstruction (const LiftingContainer &lcontainer,
                             const Ring& r = Ring(),
                             int THRESHOLD = DEF_THRESH)
        : _lcontainer(lcontainer), _r(r), _threshold(THRESHOLD) {}

     const LiftingContainer& getContainer() const {return _lcontainer;}

     template <class Vector>
     bool getRational(Vector& num, Integer& den, int switcher) const;
};
```

L'implantation de la fonction `getRational` se divise en deux parties. La première consiste à calculer le développement p-adique en utilisant l'itérateur `LiftingIterator`. La seconde consiste à reconstruire la solution rationnelle à partir du développement p-adique. Nous proposons d'étudier maintenant ces deux étapes en détail pour fournir l'implantation la plus performante possible.

4.2.3.a Construction du développement p-adique

L'utilisation de l'itérateur permet de calculer les chiffres du développement p-adique les uns après les autres. Afin de calculer complètement le développement p-adique il faut toutefois accumuler chacun de ces chiffres en les multipliant par la puissance de la base p-adique correspondante. Ce calcul revient précisément à l'évaluation d'un vecteur de polynômes dans la base p-adique. L'approche classique pour évaluer un polynôme en un point est d'utiliser la méthode de Horner. Dans un modèle de calcul algébrique cette méthode est optimale. Cependant, nous sommes dans un modèle de calcul binaire, à savoir que le coût des opérations arithmétiques dépend de la taille des opérandes. Dans ce cas précis, la méthode de Horner ne permet pas d'obtenir la meilleure complexité. En considérant que l'on utilise un algorithme de multiplication rapide d'entier de type FFT [72], le meilleur coût pour évaluer un polynôme de degré d à coefficients entiers sur k bits en un entier sur k bits est de $\tilde{O}(dk)$ opérations binaires. En comparaison la complexité obtenue avec la méthode de Horner est de $\tilde{O}(d^2k)$ opérations binaires. L'algorithme permettant d'obtenir ce coût quasiment linéaire en le degré du polynôme est récursif et il s'appuie sur une approche de type "diviser pour régner" [87]. Nous présentons maintenant cet algorithme et nous montrons qu'il est aussi le plus efficace en pratique.

Étant donnés $b = b_0 + b_1x + b_2x^2 + \ldots \in \mathbb{Z}[x]$ et $k \in \mathbb{N}$, nous définissons les fonctions de décalage et de troncature suivants :

- $\mathrm{Trunc}(b, k) = b_0 + b_1x + b_2x^2 + \ldots + b_{k-1}x^{k-1}$
- $\mathrm{Left}(b, k) = b_k + b_{k+1}x + b_{k+2}x^2 + \ldots$

Nous considérons ici sans perte de généralité que le degré du polynôme à évaluer est une puissance de deux.

Algorithme `recursiveEval`(P, a, d)
Entrées : $P \in \mathbb{Z}[x]$, $a \in \mathbb{Z}$ et $d = \deg(P)$
Sortie : $(P(a), a^d)$ tel que $P(a), a^d \in \mathbb{Z}$
Schéma :
  si $d = 1$ **alors**
    retourner $(P(0), a)$;
  sinon

$(s_1, x_1) :=$ `recursiveEval`$(\text{Trunc}(P, d/2), a, d/2)$;
$(s_2, x_2) :=$ `recursiveEval`$(\text{Left}(P, d/2), a, d/2)$;
retourner $(s_1 + x_1 s_2, x_1 x_2)$;

Soit $E(d, k)$ la coût binaire de l'algorithme `recursiveEval` où k représente la taille en bit de a et des coefficients de P. Soit $M(k)$ le coût binaire de la multiplication de deux entiers de k bits. On peut alors écrire la relation de récurrence suivante pour $E(d, k)$:

$$E(d, k) = 2E(d/2, k) + 2M(kd/2) + kd,$$

ce qui donne un coût total de $\tilde{O}(M(kd) + kd)$ opérations binaires. En particulier, on obtient $\tilde{O}(kd)$ en utilisant l'algorithme de multiplication d'entiers de Schönhage-Strassen [72].

Nous montrons maintenant que cet algorithme est aussi le plus efficace en pratique en utilisant l'arithmétique des entiers proposée par la bibliothèque GMP. Pour cela nous comparons les performances obtenues pour cet algorithme avec celle de la méthode de Horner et celle d'un algorithme basé sur l'approche "pas de bébé/pas de géant" [65]. Ce dernier algorithme offre en particulier un coût de $\tilde{O}(kd\sqrt{d})$ opérations binaires.

Le tableau 4.1 illustre les temps de calcul nécessaires pour évaluer un développement p-adique par ces trois méthodes. Nous exprimons ici le temps de calcul en seconde pour des vecteur de dimension 1000 et la base p-adique $p = 65521$. Le paramètre d dans ce tableau représente le nombre de chiffres du développement, à savoir le degré du polynôme à évaluer.

degrés	$d = 940$	$d = 1770$	$d = 3016$	$d = 4677$
Horner	$3.84s$	$13.61s$	$31.28s$	$67.73s$
"pas de bébé/pas de géant"	$2.56s$	$8.48s$	$21.99s$	$47.88s$
"diviser pour régner"	$2.08s$	$4.48s$	$5.24s$	$13.17s$

TAB. 4.1 – Comparaisons des méthodes d'évaluation de développement p-adique

D'après ce tableau on peut voir que l'approche "diviser pour régner" permet en pratique d'obtenir de meilleures performances que pour les deux autres méthodes. Toutefois, nous avons remarqué que dans certain cas la méthode par "pas de bébé/pas de géant" était plus rapide que la méthode "diviser pour régner". En particulier, cette remarque se vérifie quand la base p-adique est très petite, c'est-à-dire que les coefficients du polynôme et du point d'évaluation ont très peu de bits. Par exemple, en utilisant la base p-adique $p = 17$, un vecteur d'ordre 1000 et un nombre de coefficients $d = 3679$ nous observons les temps de calcul suivants :
Horner $\rightarrow 13.74s$; "pas de bébé/pas de géant" $\rightarrow 4.49s$; "diviser pour régner" $\rightarrow 6.5s$.

Ce résultat nous laisse penser que comme pour la multiplication d'entier il existe des seuils pratiques ou le choix d'un algorithme spécifique permet d'obtenir de meilleures performances. Dans notre cas, la recherche de ces seuils n'est pas nécessaire du fait qu'en pratique les bases p-adique utilisées sont assez grande (de l'ordre de 22 bits) et que le nombre de chiffre du développement est lui aussi important (de l'ordre de $n \log n$ avec n la dimension du système). L'utilisation de l'algorithme de type "diviser pour régner" permet ici d'obtenir la plupart du temps les meilleures performances. Nous utilisons donc cet algorithme pour implanter la fonction `getRational`. Nous proposons dans l'annexe A.2 l'implantation de cet algorithme.

Toutefois, la recherche de ces seuils est intéressante et permettrait certainement d'améliorer les temps de calcul pour l'évaluation de polynômes de faible degrés. Nous envisageons d'étudier ce sujet plus en détail dans de futur travaux.

4.2.3.b Calcul de la solution rationnelle

L'utilisation de l'algorithme `ReconstructSolution` (§4.1.2) permet de reconstruire une solution rationnelle à partir de son développement p-adique et des bornes sur le numérateur et le dénominateurs des fractions. Néanmoins, nous avons vu dans la section (§4.1.2) que le coût de cet algorithme n'est pas le meilleur possible. Toutefois, si en pratique on bénéficie d'un algorithme de multiplication d'entier rapide, ce qui est le cas avec GMP, on peut modifier cet algorithme pour qu'il autorise un meilleur coût en pratique. En particulier, nous nous appuyons sur un résultat probabiliste qui permet de remplacer les n reconstructions de fraction par quelques reconstructions et des multiplications modulaires. Du fait que les multiplications d'entiers sont quasiment linéaires alors que les reconstructions de fraction sont dans notre cas quadratiques, on arrive à diminuer le nombre d'opérations nécessaire par un facteur proche de n.

L'heuristique que nous utilisons se base sur le fait que si le système est suffisamment générique alors il y a une bonne probabilité pour que les dénominateurs de la solution soient tous identiques [33]. L'approche consiste donc à reconstruire uniquement la première composante de la solution par une reconstruction de fraction et calculer les autres par des multiplications modulaires [50, 73]. Si les dénominateurs ne sont pas identiques on conserve tout de même une forte probabilité pour que le ppcm de quelques dénominateurs soit égal au ppcm de tous les dénominateurs. En calculant un dénominateur commun au fur à mesure de la reconstruction des composantes de la solution on peut générer la plupart des solutions uniquement par des multiplications modulaires.

Nous considérons dans l'algorithme suivant que la fonction `ReconstructFraction` implante la reconstruction de fraction à partir de l'algorithme d'Euclide étendu. Cette fonction retourne dans l'ordre : le numérateur et le dénominateur.

Algorithme `FastReconstructSolution(`y`,`p^k`,`N`,`D`)`
Entrée : $y \in \mathbb{Z}^n$ tel que $|y_i| < p^k$, $N, D \in \mathbb{N}$
Sortie : (x, δ) avec $x \in \mathbb{Z}^n$, $\delta \in \mathbb{Z}$ tels que $x_i/\delta \equiv y_i \mod p^k$ et $\mathrm{pgcd}(|x_i|, \delta) < N$, $\delta < D$
Schéma

$n, d \in \mathbb{Z}^n$;
$\delta := 1$;
$q := p^k$;
pour i **de** 1 **à** n **faire**
  $y_i := \delta y_i \mod q$;
  $n_i, d_i :=$ `ReconstructFraction`$(y_i, q, N, D/d)$;
  si $d_i \neq 1$ **alors**
    $\delta := d_i \delta$;
    $q := p^l$ tel que $p^{l-1} < ND/\delta \leq p^l$;
pour j **de** 1 **à** n **faire**
  $x_j := x_j \prod_{k=j+1}^{n} d_k$;
retourner $([x_1, \ldots, x_n]^T, \delta)$;

Cet algorithme permet de remplacer les reconstructions de fractions par des multiplications

modulaires dès que le dénominateur commun δ représente le dénominateur commun de tous les dénominateurs de la solution. En fait, ce changement n'est pas visible directement dans l'algorithme. Il faut s'intéresser de plus près à la fonction `ReconstructFraction` qui implante la reconstruction de fraction par l'algorithme d'Euclide étendu. Le principe de cette fonction consiste à exécuter l'algorithme d'Euclide étendu jusqu'à ce que le reste partiel soit inférieur à la borne sur le numérateur (§4.1.2). Si le diviseur et le dividende utilisés pour initialiser l'algorithme d'Euclide étendu sont déjà inférieurs à cette borne cela signifie que la fraction à reconstruire n'est rien d'autre que le dividende. Dans ce cas précis, l'appel de la fonction `ReconstructFraction` n'effectue aucun calcul mis à part une comparaison. Les seuls calculs effectués dans l'algorithme `FastReconstructSolution` sont alors la multiplication modulaire de la composante du développement par le dénominateur commun ($\delta y_i \mod q$) et la mise à jour des fractions ayant un dénominateur différent du dénominateur commun ($x_j \prod_{k=j+1}^{n} d_k$).

En pratique, on observe sur des systèmes linéaires aléatoires qu'il suffit de $O(1)$ reconstructions de fractions pour que δ soit égal au dénominateur commun de la solution. L'algorithme `FastReconstructSolution` permet alors de bien meilleures performances que l'algorithme classique présenté dans la section 4.1.2. La figure 4.4 illustre les facteurs d'amélioration que nous obtenons en pratique en utilisant les entiers GMP. Afin de comparer cette amélioration par rapport à celle qu'on pourrait obtenir en utilisant une reconstruction utilisant un pgcd rapide à la Knuth/Schönhage, nous proposons dans cette même figure la courbe théorique du gain asymptotique apporté par l'algorithme de reconstruction rapide proposé par Pan et Wang [89], c'est-à-dire un facteur de $n/(\log n \log\log n)$. Nous utilisons ici une échelle logarithmique pour représenter les facteurs d'amélioration.

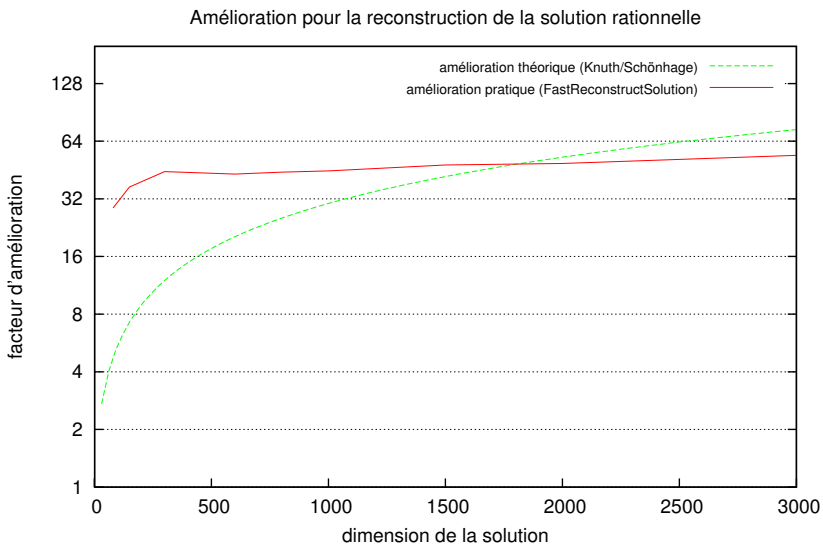

FIG. 4.4 – Amélioration pratique de l'algorithme `FastReconstructSolution` par rapport à une reconstruction classique

On peut voir dans cette figure que l'algorithme `FastReconstructSolution` permet d'améliorer en moyenne d'un facteur 40 les temps de calcul de l'algorithme de reconstruction classique. Les gains obtenus sont d'ailleurs très proche de ceux que l'on pourrait obtenir en utilisant un algorithme de reconstruction basé sur un pgcd rapide. Par exemple, pour un solution de dimen-

sion 2000 notre implantation permet d'obtenir une amélioration asymptotique équivalente à celle d'un algorithme de reconstruction rapide (i.e. un facteur 50). Toutefois, notre implantation se base sur les entiers GMP qui utilisent différents algorithmes de multiplication suivant la taille des opérandes (voir §1.2.1). De ce fait, notre implantation permet de conserver un facteur d'amélioration important même pour la reconstruction de solutions ayant peu de bits ; les solutions à reconstruire ont de l'ordre de $n \log n$ bits où n représente la dimension de la solution.

4.3 Algorithme de Dixon

Nous nous intéressons à spécialiser la méthode de résolution générique de la section 4.1 au cas des systèmes denses non structurés. Nous utilisons l'approche proposée par Dixon [22] qui utilise l'inverse de la matrice $A \mod p$ pour résoudre les systèmes linéaires sur Z_p. Ainsi, le calcul des chiffres du développement s'effectue uniquement à partir de produits matrice-vecteur. On obtient alors la spécialisation suivante pour l'algorithme `PadicLifting` de la section 4.1.1.

Algorithme `DixonPadicLifting(`A`,`b`,`p`,`k`)`
Entrée : $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^n$, $p, k \in \mathbb{N}$
Condition : $\det(A) \not\equiv 0 \mod p$
Sortie : $A^{-1}b \mod p^k$
Schéma
(1) $B := A^{-1} \mod p$;
(2) $c := b$;
(3) **pour** i **de** 0 **à** k **faire**
(4) $\bar{z} := Bc \mod p$;
(5) $c := (c - A\bar{z})/p$;
(6) $z := z + \bar{z}M$;
(7) $M := Mp$;
retourner z ;

Le coût de cet algorithme est de $O(kn^2 + n^\omega)$ opérations binaires. La preuve de ce coût se déduit en utilisant le lemme 4.1.1. Dans l'algorithme `DixonPadicLifting`, le coût des k résolutions successives dans la base p-adique est équivalent à une inversion de matrices sur un corps finis et à k produits matrice-vecteur, soit kn^2+n^ω où n^ω représente le coût du produit de matrice d'ordre n dans la base p-adique. L'inversion est ici effectuée par une reduction au produit de matrice (§3.3). En utilisant l'algorithme `DixonPadicLifting` dans l'algorithme `PadicSolver` on obtient alors un coût de $O(n^3 \log n)$ opérations binaires pour la résolution de systèmes linéaires entiers denses.

Nous nous intéressons maintenant à utiliser cette algorithme pour résoudre les systèmes linéaires entiers denses aussi bien dans le cas singulier que non singulier. Nous nous appuyons sur les travaux algorithmiques de Mulders et Storjohann [61] sur le sujet et nous utilisons notre interface (§4.2) pour fournir une implantation de haut niveau de ces résolutions. Pour cela, notre premier objectif est de fournir une implantation de l'algorithme `DixonPadicLifting` à partir d'un itérateur de développement p-adique spécialisé. L'implantation de cet algorithme est très importante car elle est la base des résolutions de systèmes linéaires entiers denses. Notre second objectif est donc de proposer une implantation de cet algorithme qui soit la plus performante possible.

Afin d'intégrer l'algorithme `DixonPadicLifting` dans notre interface, nous définissons un

conteneur spécialisé à la méthode de Dixon (`DixonLiftingContainer`). Ce conteneur hérite de l'interface `LiftingContainerBase` et spécialise la fonction virtuelle `nextdigit`. Pour que ce conteneur puisse implanter le calcul des chiffres p-adique uniquement par des produits matrice-vecteur, il encapsule l'inverse de la matrice dans la base p-adique. En utilisant cet inverse, la fonction `nextdigit` s'implante directement par des produits matrice-vecteur et des conversions de données entre l'anneau des entiers et la base p-adique.

Dans le but de permettre l'utilisation d'optimisations pour effectuer les produits matrice-vecteur dans la base p-adique, nous définissons le domaine de calcul `BlasApply` qui implante la fonction `applyV` de multiplication d'une matrice par un vecteur. Grâce à ce domaine, on peut alors spécialiser ces calculs pour qu'ils soient effectués à partir des routines numériques BLAS. Pour cela, nous réutilisons la fonction `fgemv` du paquetage FFLAS (§3.4).

Le code suivant illustre l'implantation du conteneur `DixonLiftingContainer` :

Code 4.3 – Spécialisation de la classe LiftingContainerBase à l'algorithme de Dixon

```
template <class Ring, class Field, class IMatrix, class FMatrix>
class DixonLiftingContainer : public LiftingContainerBase< Ring, IMatrix> {

  public:
     typedef typename Field::Element          Element;
     typedef typename Ring::Element           Integer;
     typedef std::vector<Integer>             IVector;
     typedef std::vector<Element>             FVector;

  protected:
     const FMatrix                  &_Ap;
     Field                            _F;
     mutable FVector              _res_p;
     mutable FVector            _digit_p;
     BlasApply<Field>                _BA;

   public:
      template <class Prime_Type, class VectorIn>
      DixonLiftingContainer (const Ring          &R,
                             const Field         &F,
                             const IMatrix       &A,
                             const FMatrix      &Ap,
                             const VectorIn&     &b,
                             const Prime_Type    &p)
         : LiftingContainerBase<Ring,IMatrix> (R,A,b,p),
         _Ap(Ap), _F(F), _res_p(b.size()),
         _digit_p(A.coldim()), _BA(F) {}

      virtual ~DixonLiftingContainer() {}

   protected:
      virtual IVector& nextdigit(IVector& digit, const IVector& residu) const
      {
        LinBox::integer tmp;

        // res_p =  residu mod p
        typename FVector::iterator iter_p = _res_p.begin();
        typename IVector::const_iterator iter = residu.begin();
        for ( ;iter != residu. end(); ++iter, ++iter_p)
           _F. init (*iter_p, _R.convert(tmp,*iter));
```

```
        // compute the solution by applying the inverse of A mod p
        _BA.applyV(_digit_p, _Ap, _res_p);

        // digit = digit_p
        typename FVector::const_iterator iter_p = _digit_p.begin();
        typename IVector::iterator iter = digit.begin();
        for ( ; iter_p!= _digit_p.end(); ++iter_p, ++iter)
           _R.init(*iter, _F.convert(tmp,*iter_p));

      return digit;
    }
  };
```

On peut voir dans ce code que cette spécialisation de conteneur ne sert qu'a définir la fonction virtuelle `nextdigit`. Le reste des calculs de l'algorithme `DixonPadicLifting` est implanté de façon générique dans l'interface `LiftingContainerBase` (§A.1).

À partir de ce conteneur spécifique, nous spécialisons les différentes fonctions de résolution de la classe `RationalSolver` à la méthode de Dixon. Nous proposons tout d'abord une spécialisation partielle de la classe `RationalSolver` à l'utilisation de ce conteneur. Pour cela, nous utilisons la structure de "trait" `struct DixonTraits ;` et nous définissons le domaine de calcul spécialisé suivant :

```
template <class Ring, class Field, class RandomPrime>
class RationalSolver<Ring,Field,RandomPrime,DixonTraits>;
```

Dans les sections suivantes nous nous intéressons à implanter le plus efficacement possible les fonctions `solveNonsingular, solveSingular` et `findRandomsolution` pour ce domaine de calcul spécialisé. En particulier, nous définissons les moyens algorithmiques utilisés pour résoudre les problèmes liés à la singularité des systèmes, à l'inconsistance des systèmes, au calcul de solutions aléatoires et à l'utilisation de méthodes probabilistes pour assurer la condition de validité de l'algorithme `DixonPadicLifting`. Nos contributions dans ces sections concernent essentiellement le développement d'implantations génériques performantes pour la résolution de systèmes linéaires denses. Nous proposons par exemple d'utiliser au maximum les routines numériques BLAS pour effectuer les produits matrice-vecteur aussi bien dans la base p-adique que pour les calculs des résidus. Pour cela, nous utilisons une écriture q-adique des matrices à coefficients entiers nous permettant de remplacer le produit matrice-vecteur en multiprécision par plusieurs produits matrice-vecteur numériques. Nous verrons qu'en particulier nos implantations sont plus performantes que celles proposées par la bibliothèque NTL [73] et qu'elles autorisent des performances équivalentes à celle de la bibliothèque IML [12].

4.3.1 Cas non singulier

L'implantation de l'algorithme de Dixon dans le cas de systèmes linéaires entiers *non singuliers* est directe. Toutefois, le calcul d'une base p-adique conservant le caractère non singulier de la matrice ne peut être effectué de façon déterministe. L'approche classique consiste à utiliser

un nombre premier aléatoire pour déterminer une base p-adique consistante. Si l'ensemble des nombres premiers aléatoires considéré est suffisamment grand par rapport au déterminant de la matrice alors la probabilité qu'un de ces nombres divise le déterminant est faible.

Notre implantation consiste donc à calculer l'inverse de la matrice modulo une base p-adique aléatoire. Nous réutilisons ici notre interface avec les paquetages FFLAS-FFPACK pour obtenir les meilleures performances possibles pour calculer cet inverse. Néanmoins, pour que notre routine d'inversion fonctionne il est nécessaire que la matrice soit inversible. Pour cela, nous calculons tout d'abord la factorisation LQUP de la matrice dans la base p-adique. Si le rang de la matrice est différent de sa dimension, on choisit alors une nouvelle base p-adique aléatoire. En pratique, ce cas n'arrive que très rarement. Si par contre la matrice est de plein rang, on calcule l'inverse à partir de sa factorisation LQUP (§3.3). L'utilisation des routines FFLAS-FFPACK n'étant possible que pour de petits nombres premiers (inférieur à 2^{26}) nous proposons une alternative basée sur une implantation générique de l'inverse par l'algorithme de Gauss classique. Cette dernière est cependant bien moins performante que l'approche hybride proposée par le paquetage FFPACK.

Une fois qu'une base p-adique consistante est déterminée et que l'inverse est calculé, il suffit d'instancier le conteneur `DixonLiftingContainer`. L'instanciation du domaine de reconstruction `RationalReconstruction` sur ce conteneur permet de calculer la solution directement à partir de la fonction `getRational`.

Le code suivant décrit l'implantation de la fonction `solveNonsingular` à partir de la méthode de Dixon :

Code 4.4 – Résolution de systèmes linéaires non singuliers par l'algorithme de Dixon

```
template <class IMatrix, class Vector1, class Vector2>
SolverReturnStatus solveNonsingular (Vector1          &num,
                                     Integer          &den,
                                     const IMatrix      &A,
                                     const Vector2      &b,
                                     bool          oldMatrix,
                                     int           maxPrimes) const
{
  typedef typename Field::Element Element;
  typedef typename Ring::Element Integer;
  int trials = 0, notfr;

  // checking size of system
  linbox_check(A.rowdim() == A.coldim());
  linbox_check(A.rowdim() == b.size());

  // history sensitive data for optimal reason
  static const IMatrix* IMP = 0;

  static BlasMatrix<Element>* FMP;
  Field *F=NULL;
  BlasMatrix<Element> *invA;

  do {
    if (trials == maxPrimes) return SS_SINGULAR;
    if (trials != 0) chooseNewPrime();
    trials++;

    LinBox::integer tmp;
```

```
      if (!oldMatrix) {
         delete FMP;
         IMP = &A;
         F= new Field (_prime);
         FMP = new BlasMatrix<Element>(A.rowdim(),A.coldim());

         typename BlasMatrix<Element>::RawIterator iter_p  = FMP->rawBegin();
         typename IMatrix::ConstRawIterator iter  = A.rawBegin();
         for (;iter != A.rawEnd();++iter,++iter_p)
            F->init(*iter_p, _R.convert(tmp,*iter));

         if ( _prime >  Prime(67108863) )
            notfr = MatrixInverse::matrixInverseIn(*F,*FMP);
         else {
            = new BlasMatrix<Element>(A.rowdim(),A.coldim());
            BlasMatrixDomain<Field> BMDF(*F);
            BMDF.inv(*invA, *FMP, notfr);
            delete FMP;
            FMP = invA;
          }
      }
      else {
        notfr = 0;
      }
   } while (notfr);

   typedef DixonLiftingContainer<Ring,
                                 Field,
                                 IMatrix,
                                 BlasMatrix<Element> > LiftingContainer;

   LiftingContainer lc( _R, *F, *IMP, *FMP, b, _prime);

   RationalReconstruction<LiftingContainer > re(lc);

   if (!re.getRational(num, den, 0))
      return SS_FAILED;
   else
      return SS_OK;
}
```

La boucle `do{}while;` dans cette implantation permet de calculer l'inverse de la matrice du système dans une base p-adique consistante. Afin d'assurer la terminaison de cette boucle, nous utilisons le paramètre `maxPrimes` pour déterminer quel doit être le nombre de base p-adique à essayer. Cela permet en pratique de fournir un moyen probabiliste pour déterminer la singularité du système et donc de changer de fonction de résolution. Nous utilisons ce procédé dans la fonction `solve` pour déterminer quelle fonction de résolution doit être employée si l'on ne connaît aucune caractéristique du système a priori.

4.3.2 Cas singulier et certificat d'inconsistance

Nous nous intéressons maintenant aux systèmes linéaires denses $Ax = b$ dans lesquel la matrice A est singulière. Cela correspond à des systèmes surdéterminés ou sous-déterminés. L'utilisation directe de l'algorithme de Dixon est ici impossible du fait qu'il n'existe aucune

base p-adique telle que A soit inversible modulo p. Une manière de résoudre ce problème est de calculer une solution à partir d'une sous-matrice associée à un mineur de A non nul d'ordre maximal. Ainsi, le calcul se ramène au cas non singulier et on peut alors utiliser l'algorithme de Dixon.

Étant donnée une matrice $A \in \mathbb{Z}^{m\times n}$ de rang r, on dit que la matrice A est de profil de rang générique si les r premiers mineurs principaux de A sont non nuls. Le lemme suivant établit la validité de l'utilisation d'un mineur non nul d'ordre maximal pour résoudre les systèmes linéaires singuliers.

Lemme 4.3.1. *Soient $A \in \mathbb{Z}^{m\times n}$ de profil de rang générique et de rang r, $b \in \mathbb{Z}^m$ et A_r la matrice associée au mineur principal de A d'ordre r. Si $y \in \mathbb{Q}^r$ est la solution du système $A_r y = b_{[1,r]}$ alors soit $x = [y^T|0]^T \in \mathbb{Q}^n$ est une solution du système $Ax = b$, soit le système $Ax = b$ est inconsistant.*

Preuve. La preuve de ce lemme est un résultat classique en algèbre linéaire. Pour cela il suffit d'utiliser le fait que si $[y^T|0]^T$ n'est pas une solution du système alors b n'appartient pas à l'espace vectoriel engendré par les lignes de la matrice A. Soient

$$A = \begin{bmatrix} A_r & A_1 \\ A_2 & A_3 \end{bmatrix}, b = \begin{bmatrix} b_r \\ b_2 \end{bmatrix} \text{ et } x = \begin{bmatrix} y \\ 0 \end{bmatrix} \text{ tels que } A_r y = b_r.$$

Comme A_r est inversible, il existe une matrice $T \in \mathbb{Q}^{(m-r)\times r}$ telle que $A_2 = TA_r$. Soit on a $b_2 = Tb_r$ et on en déduit que $A_2 y = b_2$ et de ce fait on a $Ax = b$. Soit on a $b_2 \neq Tb_r$ et on en déduit que rang$A \neq$rang$[A|b]$ et donc que le système $Ax = b$ est inconsistant. □

Néanmoins, pour déterminer une sous-matrice associée à un mineur maximal non nul il faut connaître le rang de la matrice, ce qu'on ne connaît pas a priori. Dans notre cas précis, il est impossible de calculer le rang de matrices car le coût des meilleurs algorithmes connus (Gauss-Bareiss, reste chinois) est de l'ordre de $O(n^4)$ opérations binaires alors que la résolution de système linéaire se fait en $O^{\sim}(n^3)$. Un moyen de résoudre ce problème est de calculer le rang de la matrice modulo un nombre premier p. En utilisant un ensemble de nombres premiers suffisamment grand par rapport au déterminant de la matrice A on obtient alors une forte probabilité que le rang calculé modulo p soit celui de la matrice sur $\mathbb{Z}$. Ainsi en calculant une solution du système à partir d'une sous-matrice de A inversible d'ordre r où r est le rang de la matrice A modulo p on obtient un algorithme probabiliste pour résoudre les systèmes singuliers. Pour cela, il suffit de déterminer un mineur d'ordre r non nul et d'utiliser l'algorithme de Dixon. Cet algorithme peut être rendu Las Vegas en vérifiant si la solution obtenue est bien une solution du système original. Néanmoins, si la vérification effectuée par cet algorithme échoue, on ne sait pas dire si c'est à cause d'un mauvais choix de nombre premier ou si c'est à cause d'un système inconsistant.

Dans le but de fournir une implantation complète de résolution de systèmes linéaires singuliers nous nous intéressons maintenant à certifier l'inconsistance des systèmes. Nous réutilisons ici les travaux effectués par Mulders et Storjohann [61] à ce sujet. L'approche classique pour déterminer cela est de vérifier que le rang de la matrice $[A|b]$ est égal au rang de la matrice A. Encore une fois, on ne peut pas utiliser les algorithmes de calcul de rang car il nous faut pouvoir certifier l'inconsistance avec la même complexité que la résolution de systèmes linéaires. Contrairement à la résolution de systèmes linéaires, on ne peut pas effectuer ces calculs modulo un nombre premier car l'inconsistance sur $\mathbb{Q}$ n'est pas conservée par homomorphisme dans un corps fini. Cela nous amène à la remarque suivante :

La consistance et l'inconsistance sur $\mathbb{Q}$ d'un système linéaire entier ne sont pas conservées par réduction modulaire. Plus précisément, un système linéaire peut être consistant sur les rationnels et ne plus l'être dans le corps finis $\mathbb{Z}_p$ et inversement. Cette remarque s'illustre par les deux systèmes linéaires suivants :

$$\begin{bmatrix} 1 & & \\ & 3 & \\ & & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix} \xrightarrow{\text{mod } 5} \begin{bmatrix} 1 & & \\ & 3 & \\ & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & & \\ & 3 & \\ & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix} \xrightarrow{\text{mod } 7} \begin{bmatrix} 1 & & \\ & 3 & \\ & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ \end{bmatrix}$$

L'inconsistance des systèmes peut être tout de même déterminé avec une forte probabilité en calculant le rang de la matrice modulo plusieurs nombres premiers. Toutefois, cette méthode n'est pas satisfaisante car elle ne permet pas de certifier a posteriori que le système est inconsistant. Une alternative proposée dans [38] et réutilisé dans [61] consiste à utiliser un vecteur sur les rationnels permettant de certifier que le système est inconsistant par un produit matrice-vecteur et un produit scalaire. Plus précisément, l'objectif est d'utiliser un vecteur du noyau gauche de la matrice A qui n'annule pas b. Le calcul déterministe d'un tel vecteur est cependant aussi coûteux que le calcul du rang de la matrice. L'approche utilisée consiste donc à déterminer de manière probabiliste un tel certificat.

Définition 4.3.2. *Soit le système linéaire $Ax = b$ tel que $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$. Alors $q \in \mathbb{Q}^m$ est un certificat d'inconsistance pour le système $Ax = b$ si $q^T A = 0$ et $q^T b \neq 0$.*

Il est clair que s'il existe un tel vecteur q alors il permet de certifier que le système est inconsistant car $Ax = b$ implique $q^T Ax = q^T b$. Pour déterminer un tel certificat nous utilisons le lemme suivant.

Lemme 4.3.3. *Soient $A \in \mathbb{Z}^{m\times n}$ de rang r et de profil de rang générique, et $b \in \mathbb{Z}^m$ tels que*

$$A = \begin{bmatrix} A_r & A_1 \\ A_2 & A_3 \end{bmatrix}, \; b = \begin{bmatrix} b_r \\ b_2 \end{bmatrix}$$

où $A_r \in \mathbb{Z}^{r\times r}$ est la matrice associé au mineur principal d'ordre r de A, et b_r correspond aux r premières composantes du vecteur b. Si le vecteur $c = b_2 - A_2 A_r^{-1} b_r$ est non nul alors le système $Ax = b$ est inconsistant. De plus, si i est l'indice d'une composante non nulle de c et si $v^T \in \mathbb{Z}^{1\times r}$ est la $i$ème ligne de la matrice A_2, alors le vecteur

$$q^T = [v^T A_r^{-1} | \underbrace{0, \ldots, 0}_{i-1}, -1, \underbrace{0, \ldots, 0}_{m-r-i}] \in \mathbb{Q}^{1\times m}$$

définit un certificat d'inconsistance pour le système $Ax = b$.

Preuve. La preuve de ce lemme provient du fait que le vecteur $c = b_2 - A_2 A_r^{-1} b_r$ définit le complément de Schur de la matrice $\begin{bmatrix} A_r & b_r \\ A_2 & b_2 \end{bmatrix}$. Si le vecteur c est différent du vecteur nul alors par définition du complément de Schur on sait que le rang de $[A|b]$ est différent du rang de A et donc que le système est inconsistant. □

À partir de ce lemme on peut déterminer un certificat de manière probabiliste en résolvant deux systèmes linéaires non singuliers issus d'un mineur d'ordre maximal modulo un nombre premier. Si le mineur utilisé est bien d'ordre maximal alors le vecteur q^T est déterminé en résolvant les systèmes $A_r x = b_r$ et $xA_r = v$.

Nous proposons maintenant une approche permettant de calculer un tel certificat avec une seule résolution de système. L'idée est de remplacer la résolution sur les rationnels du système $A_r x = b_r$ par une résolution sur un corps fini. Ainsi, on peut calculer l'indice de ligne i révélant l'inconsistance de manière probabiliste. Pour se faire, nous utilisons le fait que si le système est inconsistant modulo un nombre premier, alors il y a une forte probabilité pour qu'il le soit aussi sur $\mathbb{Q}$ et que les indides de ligne révélant l'inconsistance soient identiques.

On calcule alors le vecteur $c = b_2 - A_2 A_r^{-1} b_r$ modulo un nombre premier pour déterminer l'indice de ligne i de façon probabiliste. En pratique, cette approche permet d'économiser approximativement la moitié des opérations nécessaires lorsqu'un système est inconsistant.

En combinant cette approche avec l'approche Las Vegas de résolution de systèmes linéaires singuliers décrite au début de cette section nous définissons l'algorithme `DixonSingularSolve`. Cet algorithme est similaire à celui proposé par Mulders et Strojohann [61, algorithme *Certified-Solver*]. Afin d'assurer une probabilité de réussite supérieure à 1/2, on considére un ensemble de nombres premiers T tel que plus de la moitié de ces nombres vérifient rang(A)=rang($A \mod p$) et rang($[A|b]$)=rang($[A|b] \mod p$).

Algorithme `DixonSingularSolve(`A`,`b`)`
Entrée : $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$
Sortie : Soit (y) avec $y \in \mathbb{Q}^n$ tel que $Ay = b$ soit ("no solution", q^T) avec $q \in \mathbb{Q}^m$ tel que $q^T A = 0$ et $q^T b \neq 0$
Schéma
(1) Choisir $p \in T$ aléatoirement et uniformément ;
Calculer $A = LQUP \mod p$ et $r =$rang $A \mod p$;
$B := Q^T A P^T$;
$B_r := B[1..r, 1..r]$;
$c := Q^T L^{-1} b \mod p$;
$\Delta := \{i \ : \ c[i] \neq 0 \ et \ r < i \leq m\}$;
si $\Delta = \{\}$ **alors**
    $b_r := (Q^T b)[1..r]$;
    $N :=$ `numbound`(B_r, b_r) ;
    $D :=$ `denbound`(B_r, b_r) ;
    $x :=$ `DixonPadicLifting`(B_r, b_r, p, k) ;
    $z :=$ `RationalReconstruction`(x, p^k, N, D) ;
    $y := P^T[z^T, 0, \ldots, 0]^T$;
    si $Ay = b$;
        retourner (y) ;
    sinon
        aller à (1) ;
sinon
    Choisir n'importe quel $j \in \Delta$;
    $N :=$ `numbound`$(B_r^T, B^T[j, 1..r])$;
    $D :=$ `denbound`$(B_r^T, B^T[j, 1..r])$;
    $v :=$ `DixonPadicLifting`$(B_r^T, (B[j, 1..r])^T, p, k)$;
    $u :=$ `RationalReconstruction`(v, p^k, N, D) ;

$q^T := [u^T, 0^{1\times j-r-1}, -1, 0^{1\times m-j}]$;
si $q^T B = 0$ **alors**
  retourner ("no solution, $q^T Q^T$) ;
sinon
  aller à (1) ;

Lemme 4.3.4. *L'algorithme* `DixonSingularSolve` *est correct et sa complexité est bornée par* $O(m^{\omega-1}(m+n) + r^3 \log r)$ *opérations binaire où* r *est le rang de la matrice* A.

Preuve. La factorisation LQUP de la matrice A modulo p permet de déterminer le rang de A modulo p ainsi que la matrice B_r associée à un mineur non nul d'ordre maximal de $A \mod p$. En utilisant la matrice L on peut aussi déterminer si le vecteur $b \mod p$ appartient à l'espace vectoriel engendré par les colonnes de $A \mod p$. On distingue alors deux cas possibles :

Soit les $m-r$ dernières composantes du vecteur c sont nulles et donc le système est consistant modulo p. Dans ce cas, l'algorithme `DixonSingularSolve` résout le système issu du mineur non nul B_r d'ordre r. Si la solution obtenue est la solution du système initial alors l'algorithme termine et renvoit la solution. Si tel n'est pas le cas alors cela signifie que le mineur n'est pas d'ordre maximal et que le nombre premier p est un mauvais choix ; l'algorithme recommence alors le calcul avec un nouveau nombre premier.

Soit il existe au moins une composante non nulle parmi les $m-r$ dernières composantes du vecteur c et donc on considère que le système est à une forte probabilité d'être inconsistant car $\text{rang}([A|b] \mod p) \neq \text{rang}(A \mod p)$. L'algorithme `DixonSingularSolve` calcule alors un certificat à partir d'une ligne de $Q^T A P^T$ révélant l'inconsistance sur $\mathbb{Z}_p$; à savoir un des indices de lignes appartenant à Δ. Si le certificat obtenue vérifie $q^T B = 0$ alors l'algorithme termine et retourner ce certificat en spécifiant qu'aucune solution n'existe. Sinon, cela signifie que le mineur B_r^T n'est pas d'ordre maximal et donc que le nombre premier p est un mauvais choix ; l'algorithme recommence alors le calcul avec un nouveau nombre premier.

L'algorithme `DixonSolvesingular` effectue en général moins de deux fois l'étape (1) avant de terminer d'après [61, proposition 43].

Le coût de l'algorithme `DixonSingularSolve` se déduit alors directement du coût de la factorisation LQUP (lemme 3.2.1 §3.2), de l'algorithme `DixonPadicLifting` et de l'algorithme `RationalReconstruction`. □

Nous avons implanté cet algorithme dans la fonction `solveSingular` au sein de la spécialisation `RationalSolver` pour la méthode de Dixon. Nous réutilisons le conteneur de développement p-adique `DixonLiftingContainer` et le domaine de reconstruction `RationalReconstruction` pour calculer à la fois la solution du système et le certificat d'inconsistance. Comme pour le cas non singulier, nous utilisons le domaine de calcul `BlasMatrixDomain` pour calculer l'inverse et la factorisation $LQUP$ modulo p. Le code de notre implantation est somme toute similaire à celui du cas non singulier, hormis le fait qu'il faille vérifier le résultat a posteriori afin de savoir si la solution calculée est correcte ou si le certificat d'inconsistance est valide. Néanmoins, notre implantation offre la possibilité de relacher certaines vérifications et certains calculs afin d'obtenir un code plus performant pour des systèmes linéaires génériques. En particulier, nous proposons trois niveaux de calcul : Monte Carlo, Las Vegas, Las Vegas (Certifié). Pour cela nous utilisons un paramètre de type `SolverLevel` pour spécifier la stratégie de calcul utilisée. Ce paramètre est défini à partir de l'énumération suivante.

```
enum SolverLevel {
  SL_MONTECARLO, SL_LASVEGAS, SL_CERTIFIED
};
```

Les principes de ces trois niveaux de calcul sont les suivants :

- `SL_MONTECARLO` : resultats probabilistes non vérifiés (aucune validation)
  $\hookrightarrow$ retourne soit une solution, soit un statut inconsistant ;
- `SL_LASVEGAS` : résultats probabilistes vérifiés
  $\hookrightarrow$ retourne soit une solution, soit un statut inconsistant, soit un statut d'échec ;
- `SL_CERTIFIED` : résultats probabilistes vérifiés avec certificat d'inconsistance
  $\hookrightarrow$ retourne soit une solution, soit un certificat d'inconsistance, soit un statut d'échec.

Afin d'optimiser notre implantation, nous utilisons certaines astuces permettant de réduire le nombre d'opérations. En particulier, au lieu de calculer la factorisation LQUP de la matrice A puis calculer le vecteur $Q^T L^{-1} b$, nous calculons directement la factorisation LQUP de la matrice $[A^T|b^T]^T$. À partir de cette factorisation, on détermine à la fois la factorisation $LQUP$ de A et le vecteur $Q^T L^{-1} b$ à l'aide de permutations. Une autre astuce plus évidente consiste à ne calculer qu'une seule fois l'inverse de la sous-matrice B_r à partir de la factorisation LQUP. Pour calculer le certificat d'inconsistance, on peut alors réutiliser la transposée de cet inverse pour initialiser le développement p-adique. Le code de notre implantation pour l'algorithme `DixonSingularSolve` est disponible dans l'annexe A.3.

La recherche de solutions rationnelles diférentes pour un même système linéaire entier est l'une des opérations clés dans l'approche proposée par Giesbrecht (§4.4) pour trouver une solution diophantienne du système. On entend ici par *différente* des solutions qui n'ont pas le même dénominateur. En pratique, nous avons observé que pour un système linéaire donné l'algorithme `DixonSingularSolve` retourne souvent la même solution quelque soit le nombre premier utilisé. L'utilisation de cet algorithme pour implanter la résolution des systèmes diophantiens n'est pas intéressante. Nous nous intéressons dans la prochaine section à modifier cet algorithme pour qu'il permette le calcul de solutions différentes pour un même système linéaire entier.

4.3.3 Solutions aléatoires

Le dénominateur de la solution rationnelle calculée par l'algorithme `DixonSongularsolve` est déterminé par le mineur non nul d'ordre maximal utilisé. En particulier, le dénominateur est un diviseur de ce mineur. Si la sous-matrice inversible associée à ce mineur est toujours la même alors les solutions obtenues par l'algorithme `DixonSongularsolve` sont identiques. En pratique, on observe que pour différents nombres premiers p la sous-matrice inversible d'ordre maximal déduit à partir de la factorisation LQUP de $A \mod p$ est toujours le même. Afin de modifier cet algorithme pour que le mineur d'ordre maximal choisi soit différent d'une exécution à l'autre, il faut donc choisir aléatoirement la sous-matrice inversible de A, ce qui ne semble pas être le cas en utilisant la factorisation LQUP de la matrice A. Une approche utilisée par Mulders et Storjohann [61] pour obtenir une sous-matrice inversible aléatoire consiste à perturber la matrice A par une matrice aléatoire W.

Lemme 4.3.5 (voir p. 489 [61]). *Soient $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$ et $W \in \mathbb{Z}^{n\times m}$ une matrice aléatoire tels que rang(A)=rang(AW). Si $y \in \mathbb{Q}^m$ est une solution du système $AWy = b$ alors $x = Wy$*

est un solution du système $Ax = b$.

En utilisant les permutations Q et P issues de la factorisation LQUP de la matrice AW mod p pour déduire une sous-matrice inversible d'ordre maximal on obtient alors une méthode probabiliste pour déterminer des mineurs aléatoires non nuls d'ordre maximal. En outre, cela permet de calculer des solutions rationnelles ayant des dénominateurs aléatoires. Plus précisément, Mulders et Storjohann utilisent un sous-ensemble fini $U \subset \mathbb{Z}$ pour définir les coefficients de la matrice W et ils montrent qu'en considérant $U = \{0, 1\}$ les solutions calculées restent aléatoires.

Nous utilisons cette approche pour modifier l'algorithme `DixonSingularSolve` de telle sorte que le mineur d'ordre maximal soit défini de façon aléatoire en préconditionnant la matrice par un matrice W à coefficients dans $U = \{0, 1\}$. Afin de minimiser le coût de ce préconditionnement, nous considérons que la matrice A est de plein rang en ligne. Pour cela, nous utilisons les r premières lignes de la matrice $Q^T A$ où Q est définie par la factorisation LQUP de la matrice A mod p. La sous-matrice inversible est alors déterminée en considérant le mineur principal d'ordre maximal de la matrice AW. Mulders et Storjohann utilisent aussi cette méthode dans l'algorithme *SpecialMinimalSolution* [61, page 502] pour générer des solutions rationnelles aléatoires d'un système linéaire singulier.

Nous proposons l'implantation de cette version de l'algorithme `DixonSongularsolve` dans la fonction `findRandomSolution` définie par la l'interface `RationalSolver`. L'implantation de cette fonction est simplement une modification de la fonction `solvesingular` de telle sorte que le mineur utilisé pour calculer la solution soit égal à la sous-matrice principale d'ordre r de $Q^T AW$ où Q provient de la factorisation LQUP de $A \mod p$ et où la matrice W est définie aléatoirement à coefficients dans l'ensemble $\{0, 1\}$.

4.3.4 Optimisations et performances

Nous étudions maintenant certaines optimisations permettant d'améliorer en pratique les performances de l'algorithme de Dixon aussi bien dans le cas singulier que non singulier. Ces optimisations consistent à effectuer tous les produits matrice-vecteur dans l'algorithme de Dixon à partir des routines numériques BLAS. En particulier, nous proposons d'utiliser une représentation q-adique des matrices pour permettre cela même quand les données sont a priori très grandes. De même, nous réutilisons nos routines hybrides "exact/numériques" présentées dans le chapitre 3 pour effectuer toutes les opérations d'algèbre linéaire dans la base p-adique. Nous nous intéressons ensuite à comparer les performances de notre implantation par rapport à celle des bibliothèques NTL [73] et IML [12, 13].

L'optimisation majeure de notre implantation consiste à utiliser au maximum les routines numériques BLAS. On a pu voir dans le chapitre 3 que l'utilisation de ces routines permettait d'obtenir des performances très satisfaisantes pour les opérations en algèbre linéaire dense sur un corps fini. Nous réutilisons ici ces routines pour calculer l'ensemble des opérations dans la base p-adique. Plus précisément, cela concerne le calcul de l'inverse et le calcul des chiffres p-adique à partir de produits matrice-vecteur. En pratique, nous utilisons l'interface `BlasMatrixDomain` (§3.4.1) pour définir ces calculs.

Nous utilisons aussi les routines BLAS pour effectuer les produits matrice-vecteur $A\bar{z}$ dans le calcul des résidus (voir opération (4) de l'algorithme `DixonPadicLifting`). Ces calculs sont effectués sur les entiers et l'utilisation des BLAS est réalisable uniquement si les résultats de ces

opérations ne dépassent jamais les 53 bits de mantisse des nombres flottants double précision. En pratique, cette approche est possible uniquement pour des systèmes ayant de petits coefficients (i.e. de l'ordre de 16 bits).

Afin de bénéficier des performances des BLAS pour des systèmes ayant de plus grands coefficients, nous utilisons un changement de représentation de la matrice du système. Soit $A \in^{m\times n}$, on définit la représentation q-adique de A par

$$A = S_0 + qS_1 + q^2S_2 + \ldots + q^kS_k, \quad \text{où} \quad S_i \in \mathbb{Z}_q^{m\times n}, \quad 0 \leq i \leq k. \tag{4.2}$$

Grâce à cette représentation de A, le produit $A\bar{z}$ s'effectue via la relation $A\bar{z} = \sum_{j=0}^{k} S_j\bar{z}q^j$. En utilisant les nombres flottants double précision pour définir les chiffres q-adiques S_j de la matrice A, on peut alors réutiliser les BLAS pour effectuer les k produits matrice-vecteur pourvu que chaque résultat partiel $S_j\bar{z}$ tienne sur 53 bits. Ceci se vérifie en utilisant une base q-adique qui satisfait l'équation

$$(q-1)(p-1)n \leq 2^{53}. \tag{4.3}$$

A partir de ces produits partiels, on retrouve le produit complet en évaluant le polynôme $S(x) = S_0\bar{z} + S_1\bar{z}x + \ldots + S_k\bar{z}x^k$ en q. Nous avons vu dans la section 4.2.3 que cette évaluation pouvait bénéficier d'un algorithme rapide de type "diviser pour régner". Toutefois, le choix d'une base q-adique spécifique nous permet d'effectuer cette évaluation encore plus rapidement. En particulier, nous utilisons 2^{16} comme base q-adique. Cette base permet de vérifier l'équation 4.3 pour un ensemble de systèmes linéaires et de base p-adique suffisants. Par exemple, en définissant la base p-adique à partir d'un nombre premier sur 22 bits, on peut atteindre des systèmes linéaires ayant de l'ordre de 30000 inconnues ; ce qui est bien au dela de ce que les ordinateurs actuels sont capables de stocker en mémoire vive et en mémoire swap. L'avantage de cette base est qu'elle permet en pratique d'effectuer l'évaluation du polynôme $S(x)$ uniquement avec trois additions multiprécisions et des masques bits à bits. Pour cela, nous nous appuyons sur le fait que les entiers multiprécisions sont stockés au travers d'une chaîne d'octets et qu'il est possible d'instancier ces derniers directement à partir d'une chaîne de bits. Il suffit alors de remarquer que les quatres séries suivantes

$$\begin{array}{rl} & S_0\bar{z} + x^4S_4\bar{z} + x^8S_8\bar{z} + \ldots \\ x\times & (S_1\bar{z} + x^4S_5\bar{z} + x^8S_9\bar{z} + \ldots) \\ x^2\times & (S_2\bar{z} + x^4S_6\bar{z} + x^8S_{10}\bar{z} + \ldots) \\ x^3\times & (S_3\bar{z} + x^4S_7\bar{z} + x^8S_{11}\bar{z} + \ldots) \end{array}$$

ne possèdent aucun terme recouvrant si $x > 2^{14}$. Les coefficients de ces séries étant codés sur 53 bits et q étant égal à 2^{16}, cela signifie que l'écriture binaire du résultat de l'évaluation de ces séries en 2^{16} se déduit directement de l'écriture binaire de chacun des termes de ces séries. On peut donc évaluer chacune de ces séries en 2^{16} uniquement en utilisant des masques bits à bits. En pratique on utilise une chaîne de caractères pour stocker les valeurs binaires de ces séries et on utilise les fonctions d'initialisation des nombres multiprécisions à partir de tableaux de bits (fonction `mpz_import` dans GMP). Il suffit ensuite d'additionner ces entiers multiprécisions pour obtenir le résultat de l'évaluation de $S(x)$ en 2^{16}.

Toutefois, l'utilisation de la représentation q-adique définie par l'équation 4.2 nécessite que tous les coefficients $A_{i,j}$ soient positifs, ce qui n'est pas le cas en pratique. Afin d'autoriser cette technique pour des coefficients négatifs, nous utilisons un chiffre supplémentaire pour stocker le signe des coefficients et nous utilisons une écriture en complément à la base. Ainsi, pour chaque coefficient négatif $A_{i,j}$ on définit une écriture q-adique en complément à la base 2^{16}. Ceci nous donne le découpage suivant pour A :

$$A = S_0 + qS_1 + \ldots + q^k S_k - q^{k+1} S_{k+1},$$

où $S_{k+1}[i,j] = 1$ si le coefficient $A[i,j]$ est négatifs sinon $S_{k+1}[i,j] = 0$. L'utilisation d'une représentation en complément à la base permet d'étendre notre méthode aux nombres négatifs uniquement en rajoutant un produit matrice-vecteur $S_{k+1}\bar{z}$ et une soustraction multiprécision par rapport au cas positif. Ceci donne un coût total pour calculer le produit matrice vecteur $A\bar{z}$ sur les entiers de 3 additions multiprécisions, une soustraction multiprécision et $k+1$ produits matrice-vecteur BLAS avec k minimal tel que $16k > \log max_{i,j}|A_{i,j}|$.

En effet, le choix de la base $q = 2^{16}$ permet de convertir la matrice dans une représentation q-adique en complément à la base directement par des masques bits à bits et des décalages ; nous considérons que les entiers multiprécisions sont représentés par des tableaux de mots machines (32 ou 64 bits), ce qui est pratiquement toujours le cas. À partir de cette écriture, on peut donc effectuer le produit matrice-vecteur $A\bar{z}$ en calculant les produits partiels $S_j\bar{z}$, soit $k+1$ produits matrice-vecteur numériques, et en évaluant les quatres séries déduites de ces produits partiels en q, soit 3 additions, une soustraction multiprécision et des masques bits à bits. Notons toutefois que si les coefficients entiers vérifient l'équation 4.3 nous utilisons directement les BLAS sans passer par une représentation q-adique. Cette optimisation est intégrée dans le code définissant l'interface de calcul des développements p-adiques `LiftingContainerBase` à partir du domaine de calcul `MatrixApplyDomain`. Nous proposons en annexe A.11 l'implantation LinBox de ce domaine de calcul.

Nous nous intéressons maintenant à évaluer les performances de notre implantation de l'algorithme de Dixon dans le cas de matrices inversibles. Pour cela, nous utilisons la fonction `solveNonsingular` définie dans le code 4.4 (§4.3.1) qui prend en compte l'ensemble de nos optimisations.

Afin de comparer nos performances, nous effectuons les mêmes tests avec les bibliothèques NTL et IML. La bibliothèque NTL développée par Victor Shoup [73] propose une implantation de l'algorithme de Dixon en utilisant des bases p-adiques de l'ordre de 30 bits définies par des nombres premiers de type FFT ($p-1$ divisible par une grande puissance de 2). Si la taille des données est suffisamment petite, la plupart des calculs sont effectués soit à partir de nombres flottants doubles précisions soit à partir d'entiers 64 bits si l'architecture le permet. Dans le cas où les données sont trop grandes c'est l'arithmétique d'entiers multiprécisions de la bibliothèque qui est utilisée.

La bibliothèque IML [12] propose une implantation de l'algorithme de Dixon utilisant une base p-adique de type RNS (*residu number system*). Le but est d'utiliser une base définie par plusieurs moduli ayant de l'ordre de 22 bits afin de pouvoir effectuer l'ensemble des opérations dans la base à partir des routines numériques BLAS. Grâce à cette technique l'implantation de la bibliothèque IML bénéficie des performances des BLAS tout en utilisant une base p-adique suffisamment grande pour diminuer le nombre d'itérations dans l'algorithme de Dixon.

La table 4.2 illustre les performances de ces trois implantations pour des systèmes comportant des données sur 3, 32 et 100 bits. Les temps de calculs sont exprimés en secondes, minutes

et heures suivant l'ordre de grandeur. Ces tests ont été effectués à partir d'une architecture Intel Xeon 2.66ghz avec 1Go de RAM et 1Go de swap. La première colonne de ce tableau indique les dimensions des systèmes, les autres colonnes représentent les temps de calcul pour les implantations (LinBox , IML, NTL) en fonction de la taille des entrées. Lorsqu'aucun résultat n'est spécifié cela signifie que l'architecture que nous utilisons ne possède pas assez de ressources mémoire (RAM+swap) pour traiter le système.

	3 bits			32 bits			100 bits	
	LinBox	IML	NTL	LinBox	IML	NTL	LinBox	IML
200×200	$0.23s$	$0.18s$	$0.91s$	$1.66s$	$0.79s$	$4.89s$	$7.51s$	$5.53s$
500×500	$2.02s$	$1.95s$	$15.95s$	$14.74s$	$10.32s$	$77.17s$	$74.52s$	$48.90s$
800×800	$6.62s$	$6.71s$	$66.50s$	$48.81s$	$32.74s$	$328.33s$	$265.04s$	$164.63s$
1200×1200	$20.41s$	$20.90s$	$228.86s$	$167.41s$	$100.63s$	$21m$	$14m$	$6m$
2000×2000	$76.87s$	$86.96s$	$17m$	$628.78s$	$422.52s$	$3h$	$1h$	$30m$
3000×3000	$234.05s$	$297.23s$	$1h$	$39m$	$23m$	$11h$	-	-

TAB. 4.2 – Comparaisons des implantations de l'algorithme de Dixon (`Xeon-2.66Ghz`)

D'après ce tableau on remarque dans un premier temps que les performances de la bibliothèque NTL sont largement inférieures à celles de notre implantation et de la bibliothèque IML. Cela s'explique par le fait que la bibliothèque NTL n'utilise pas de technique pour éviter d'effectuer les calculs sur des entiers multiprécisions dans le cas où les données du systèmes sont grandes (i.e. 32 bits). Dans le cas de petites entrées (i.e. 3 bits), la bibliothèque NTL reste quand même moins performante du fait qu'elle n'utilise pas les bibliothèques BLAS pour calculer l'inverse de la matrice. Ce calcul représente pratiquement 75% du temps total pour l'implantation NTL alors que pour notre implantation celui-ci ne représente qu'à peine 10%. D'ailleurs, cette inversion nous permet d'obtenir de meilleures performances que la bibliothèque IML pour de petites entrées (i.e. 3 bits). En effet, l'utilisation d'une base RNS nécessite de calculer l'inverse de la matrice pour plusieurs moduli, ce qui dans le cas de petites entrées représente pratiquement 50% du temps de calcul total. Bien que cette technique permette de diminuer le nombre d'itérations, cela ne compense pas le temps de calcul passé à inverser la matrice. Par contre, lorsque les entrées sont plus grandes (i.e. 32 bits, 100 bits) cette technique s'avère judicieuse car le nombre d'itérations est tellement important que le calcul de l'inverse dans la base RNS ne représentes plus que 6% du temps de calcul total. Cette technique permet par exemple de n'effectuer que 996 itérations pour un système 800×800 sur 100 bits avec une base RNS de 8 moduli sur 22 bits alors que notre implantation nécessite 7872 itérations avec un seul modulo sur 22 bits. Bien que notre implantation effectue pratiquement 8 fois plus d'itérations, les performances restent tout de même dans un facteur inférieur à 2 par rapport à l'approche RNS. Une implantation idéale serait donc une méthode hybride qui en fonction de la taille des entrées utilise soit une base RNS soit un découpage q-adique.

Notre implantation dans le cas singulier s'appuie directement sur l'implantation du cas non singulier. Pour cela il suffit de déterminer un mineur non nul d'ordre maximal. Dans le cas de la fonction `solveSingular` nous utilisons les permutations calculées par la factorisation LQUP de la matrice dans la base p-adique pour déterminer un tel mineur. Dans le cas de la fonction `findRandomSolution` nous utilisons la permutation en ligne Q de cette même factorisation pour calculer une matrice de plein rang en ligne et nous utilisons une matrice dense aléatoire W à coefficients dans $U = \{0, 1\}$ pour obtenir un mineur non nul d'ordre maximal de façon aléatoire.

En pratique ce précondionnement est critique car il nécessite une multiplication de matrices sur des entiers qui peuvent être vraisemblablement grands. Néanmoins, si $max_{i,j}|A_{i,j}| < M$ est tel que la dimension n du système vérifie $nM < 2^{53}$ on peut alors utiliser la routine de multiplication `dgemm` des BLAS pour effectuer ce calcul. Dans le cas où la magnitude des entrées du système ne permet pas d'utiliser directement les routines BLAS, nous utilisons alors la même technique de découpage q-adique que nous avons décrite au début de cette section (fonction `applyM` code A.11 §A.4). De même, nous réutilisons cette technique pour calculer la solution $y = Wx$ à partir de la solution x du système $AWx = b$.

	`solveSingular`			
	3 bits	d	32 bits	d
200×250	$0.33s$	1059	$1.94s$	6857
400×800	$1.82s$	2326	$10.64s$	13919
600×1000	$4.51s$	3662	$28.98s$	21062
800×1000	$8.83s$	5042	$61.42s$	28244
800×1500	$9.41s$	5040	$61.98s$	28246
1200×1400	$24.56s$	7916	$188.71s$	42723
1200×2000	$24.76s$	7916	$187.94s$	42717

TAB. 4.3 – Performances de la fonction `solveSingular` (`Xeon-2.66Ghz`)

	`findRandomSolution`			
	3 bits	d	32 bits	d
200×250	$0.60s$	1574	$2.73s$	7371
400×800	$3.82s$	3763	$15.84s$	15364
600×1000	$9.68s$	5885	$42.99s$	23283
800×1000	$18.15s$	7888	$89.78s$	31092
800×1500	$20.86s$	8274	$95.66s$	31479
1200×1400	$51.97s$	12412	$264.67s$	47215
1200×2000	$58.73s$	12971	$282.69s$	47771

TAB. 4.4 – Performances de la fonction `findRandomSolution` (`Xeon-2.66Ghz`)

Nous présentons dans les tables 4.3 et 4.4 les performances de ces deux fonctions. Nous utilisons pour cela des systèmes singuliers aléatoires définis à partir de matrices de plein rang en ligne. Nous ne pouvons comparer nos implantations avec celles des bibliothèques NTL et IML du fait que ce type de résolutions n'est pas disponible dans leurs distributions. Les colonnes intitulées d dans ces tables correspondent à la taille en bits du dénominateur de la solution calculée. On peut voir par exemple dans ces tables que lorsque les données sont sur 3 bits la fonction `finRandomSolution` retourne des solutions rationnelles qui possèdent des dénominateurs pratiquement deux fois plus grands que celles obtenues à partir de la fonction `solveSingular`. Cela provient du fait que le préconditionnement par la matrice aléatoire dense W fait augmenter la taille des entrées du système. Les entrées du système étant beaucoup plus petites que la dimension des matrices, le préconditionnement par la matrice W fait pratiquement doubler le nombre de bits des entrées du système. À cause de ce grossissement le nombre d'itérations nécessaires pour calculer une solution est doublé. Par exemple pour le système 1200×2000 sur 3 bits le nombre d'itérations passe de 832 à 1582 en utilisant une base p-adique de 22 bits. De ce fait, on peut

observer que le temps de calcul de ces deux fonctions diffère approximativement d'un facteur deux. Toutefois, ceci ne se vérifie plus lorsque on regarde les résultats des systèmes sur 32 bits. En effet, les données du système sont maintenant beaucoup plus grandes que la dimension du système. Le préconditionnement n'affecte que très peu le nombre d'itérations nécessaires pour calculer une solution. Pour le système 1200×2000 on passe de 4123 itérations pour la fonction `solveSingular` à 4621 pour la fonction `finRandomSolution`, ce qui représente une augmentation d'à peine 12%. On peut voir que cette augmentation se répercute également sur la taille du dénominateur des solutions : $100 \times (47771/42717 - 1) = 11.8\%$. Bien que le nombre d'itérations n'augmente que très peu, les temps de calcul des solutions aléatoires sont une fois et demie plus importants que ceux obtenus avec la fonction `solveSingular`. Ce qui représente un surcoût de 50%. Ce surcoût est dû en partie au produit matriciel AW et au produit matrice-vecteur Wx qui représentent approximativement 15% d'augmentation mais surtout à cause du grossissement des données de la matrice AW qui alourdit le calcul des résidus d'approximativement 25%. Les 10% manquants sont dûs à la repercution de l'augmentation du nombre des itérations sur les autres opérations (calcul des chiffres p-adiques et reconstruction rationnelle).

Par rapport au cas non singulier, la fonction `solveSingular` conserve approximativement les même performances que celles obtenues avec la fonction `solveNonsingular`. Néanmoins, on observe un surcoût inférieur à 20% qui est dû aux permutations et aux manipulations mémoire effectuées pour récupérer un mineur non nul d'ordre maximal.

Dans ces expérimentations, nous avons montré que l'intégration des routines numériques BLAS dans l'algorithme de Dixon permettait d'améliorer les performances d'un facteur pratiquement 10 par rapport à l'implantation classique déjà très performante proposée par la bibliothèque NTL. Notre implantation permet dans le cas de systèmes linéaires non singuliers d'obtenir les meilleures performances pour de petites données (i.e 3 bits). L'approche RNS utilisée par la bibliothèque IML s'avère être la plus intéressante lorsque les données ne tiennent plus sur un mot machine. Néanmoins, notre implantation basée sur un développement q-adique de la matrice permet d'obtenir des performances très proches de celles de la bibliothèque IML (approximativement un facteur deux) bien que l'on effectue beaucoup plus d'itérations. Notre implantation haut niveau à partir d'un conteneur de développement p-adique et d'un module de reconstruction rationnelle basé sur un itérateur nous ont permis de définir facilement plusieurs implantations pour le cas singulier, qui ne sont pas disponibles dans les bibliothèques NTL et IML, et qui autorisent des performances très satisfaisantes.

4.4 Solutions diophantiennes

Dans cette section nous nous intéressons au calcul de solutions diophantiennes d'un système linéaire entier. Pour $A \in \mathbb{Z}^{m \times n}$ et $b \in \mathbb{Z}^m$ on cherche alors un vecteur solution $x \in \mathbb{Z}^n$ tel que $Ax = b$. Nous nous intéressons uniquement aux systèmes à la fois consistants et singuliers. En effet, si le système est non singulier cela signifie que la matrice est inversible et que l'on peut calculer l'unique solution du système en utilisant n'importe quel algorithme de résolution sur le corps des rationnels. Si le système est inconsistant sur $\mathbb{Q}$ alors cela signifie qu'il est aussi sur $\mathbb{Z}$ et donc qu'il n'existe aucune solution diophantienne. Le calcul de solutions diophantiennes revient en particulier à trouver une solution rationnelle possédant un dénominateur égal à 1 parmi l'infinité des solutions possibles. Cependant, il se peut qu'une telle solution n'existe pas. On parle alors d'inconsistance du système sur $\mathbb{Z}$ [38].

Historiquement le calcul de solutions diophantiennes se ramène au calcul de la forme de

Smith [63, chap. 2.21] ou d'Hermite [76]. Toutefois, ces approches ne sont pas satisfaisantes car elles nécessitent de l'ordre de n^4 opérations binaires pour calculer une solution diophantienne alors qu'on sait résoudre les systèmes linéaires sur $\mathbb{Q}$ avec de l'ordre de n^3 opérations binaires avec l'algorithme de Dixon ou dans le même coût que la multiplication de matrices entières avec l'algorithme de Storjohann [77]. Nous nous intéressons ici à une approche probabiliste proposée par Giesbrecht [39] permettant de calculer une solution diophantienne à partir de résolutions de systèmes linéaires sur $\mathbb{Q}$. Cette approche consiste à combiner différentes solutions rationnelles du système linéaire dans le but de faire diminuer le dénominateur de la solution jusqu'a ce qu'il soit égal à 1. En particulier, cette approche permet d'obtenir la même complexité que la résolution de systèmes linéaires entiers sur $\mathbb{Q}$ à des facteurs logarithmiques près. Nous montrons dans cette section que c'est effectivement le cas en pratique et qu'il est même possible d'obtenir des temps de calculs similaires à ceux obtenues pour la résolution de systèmes linéaires dense sur les rationnels. Pour cela, nous nous appuyons sur notre interface de calcul de solutions rationnelles `RationnalSolver` et l'implantation de l'algorithme de Dixon (§4.3). Nous verrons en particulier que pour des systèmes linéaires génériques il suffit de deux solutions rationnelles pour calculer une solution diophantienne. L'implantation que nous proposons ici permet d'obtenir les meilleures performances actuelles pour la résolution de systèmes linéaires diophantiens denses.

Dans un premier temps, nous détaillons la méthode probabiliste proposée par Giesbrecht [39] pour calculer une solution diophantienne (§4.4.1). Cette méthode est satisfaisante seulement si l'on sait a priori que le systèmes est consistants sur $\mathbb{Z}$. Afin de fournir une implantation de cette méthode dans le cas où l'on ne connaît a priori aucune information sur le système linéaire, nous réutilisons dans la section 4.4.2 une approche développée par Mulders et Storjohann [61] pour gérer cette inconsistance. Cette méthode consiste à utiliser un certificat établissant la minimalité du dénominateur des solutions rationnelles obtenues par combinaison. Grâce à ce certificat, nous verrons que l'on peut définir une implantation certifiée pour le calcul de solutions diophantiennes. Enfin, nous montrons dans la dernière section (§4.4.3) que notre implantation est très satisfaisante en nous comparant à l'implantation proposée par la bibliothèque IML [12] qui propose le même type de résolutions. En particulier, nous verrons que notre approche est beaucoup plus adaptative que celle proposée par la bibliothèque IML et qu'elle permet en général de trouver une solution diophantienne en combinant seulement deux solutions rationnelles.

4.4.1 Approche proposée par Giesbrecht

La méthode proposée par Giesbrecht dans [39] pour calculer une solution diophantienne consiste à utiliser des combinaisons de solutions rationnelles pour faire décroître le dénominateur des solutions. Nous considérons ici que les systèmes linéaires possèdent au moins une solution diophantienne.

Soit $y \in \mathbb{Q}^k$ tel que $y = [n_1/d_1, \ldots, n_k/d_k]^T$ où les fractions n_i/d_i sont irréductibles. Nous définissons les fonctions suivantes :

- $d(y) = \mathrm{ppcm}(d_1, \ldots, d_k)$
- $n(y) = d(y)y$

L'idée utilisée par Giesbrecht s'illustre à partir du lemme suivant.

Lemme 4.4.1 (voir p.488 [61]). *Soient $y_1, y_2 \in \mathbb{Q}^n$ deux solutions du système $Ax = b$. Soient*

$g, s_1, s_2 \in \mathbb{Z}$ tel que $g = \mathrm{pgcd}(d(y_1), d(y_2)) = s_1 d(y_1) + s_2 d(y_2)$, alors

$$y = \frac{s_1 d(y_1) y_1 + s_2 d(y_2) y_2}{g}$$

est une solution du système $Ax = b$ et $d(y)|g$.

En utilisant ce lemme, on peut définir un algorithme probabiliste pour calculer une solution diophantienne. Pour cela, il suffit de calculer des solutions rationnelles aléatoires du système et de les combiner jusqu'à ce que l'on obtienne un dénominateur $d(y)$ égal à 1. Le nombre de solutions nécessaires pour obtenir une solution diophantienne est alors dépendant des dénominateurs des solutions calculées. Pour permettre le calcul de solutions ayant des dénominateurs suffisamment différents, l'approche générale consiste à modifier le système en le préconditonnant par une ou plusieurs matrices aléatoires. Ainsi, pour un système $Ax = b$, on se ramène à calculer un solution du système $PAQx = Pb$ où P et Q sont des matrices aléatoires à coefficients entiers.

La première approche proposée par Giesbrecht [39, §2] pour borner le nombre de solutions rationnelles nécessaires consiste à utiliser des matrices triangulaires Toeplitz aléatoires pour préconditionner le système. Toutefois, cette méthode nécessite d'utiliser des extensions algébriques de $\mathbb{Q}$ pour prouver une convergence en $O(\log r)$ itérations avec $r = \mathrm{rang}(A)$. Une autre approche, proposée par Mulders et Storjohann [61], consiste à utiliser des matrices denses aléatoires à coefficients dans un sous-ensemble fini U de $\mathbb{Z}$ comme préconditionneurs. Cette approche est spécifique aux systèmes denses mais permet d'éliminer l'utilisation d'extensions algébriques de $\mathbb{Q}$. De plus, cette approche permet d'obtenir une convergence en $O(1)$ itérations lorsque $U = \{0, 1, \ldots, M\}$ avec $M = \max(24, \lceil \log r^{\frac{r}{2}} ||A||^r \rceil)$ [61, corollaire 28]. En utilisant le sous-ensemble $U = \{0, 1\}$ cette méthode permet d'obtenir une convergence de $O(\log r + \log\log ||A||)$ itérations [61, corollaire 27]. Dans le cas de systèmes linéaires singuliers, Mulders [60] réutilise cette approche en la combinant avec les résultats de Wiedemann [91] pour définir des préconditionneurs creux aléatoires et obtenir le même type de convergence.

L'utilisation de préconditionneurs aléatoires spécifiques permet de fournir une solution diophantienne dans le même coût que la résolution de systèmes rationnels, à des facteurs logarithmiques près. Néanmoins, l'utilisation directe du lemme 4.4.1 entraîne un grossissement des composantes de la solution au fur et à mesure des combinaisons. Afin de limiter ce grossissement, Mulders et Storjohann propose d'utiliser une combinaison différente. L'idée est d'utiliser une troisième solution y_0 permettant de définir la solution diophantienne comme une combinaison linéaire des solutions rationnelles sans avoir à utiliser les coefficients de Bezout.

Lemme 4.4.2 (voir p.501 [61]). *Soient $y_0, y_1, y_2 \in \mathbb{Q}^n$ des solutions du système $Ax = b$. Soit $a \in \mathbb{Z}$ tel que $\mathrm{pgcd}(d(y_0), d(y_1) + ad(y_2)) = \mathrm{pgcd}(d(y_0), d(y_1), d(y_2))$. Alors*

$$y = \frac{d(y_1) y_1 + a d(y_2) y_2}{d(y_1) + a d(y_2)}$$

vérifie $Ay = b$ et $\mathrm{pgcd}(d(y), d(y_0))$ est un diviseur de $\mathrm{pgcd}(d(y_0), d(y_1), d(y_2))$.

L'utilisation de ce lemme pour combiner les solutions permet de calculer une solution de la forme

$$\hat{y} = \frac{d(y_1) y_1 + a_1 d(y_2) y_2 + a_2 d(y_3) y_3 + \ldots + a_k d(y_{k+1}) y_{k+1}}{d(y_1) + a_1 d(y_2) + a_2 d(y_3) + \ldots + a_k d(y_{k+1})}$$

telle que $\mathrm{pgcd}(d(y_0), d(\hat{y})) = \mathrm{pgcd}(d(y_0), d(y_1), \ldots, d(y_k))$. Grâce à cette combinaison, la taille de la solution $\hat{y}$ reste du même ordre que la taille des solutions combinées [61, §6.1]. Pour calculer

un entier $a \in \mathbb{Z}$ satisfaisant $gdc(d(y_0), d(y_1)+ad(y_2)) = \text{pgcd}(d(y_0), d(y_1), d(y_2))$, on peut utiliser l'algorithme `Split` proposé par Mulders et Storjohann dans [78, §2]. Cet algorithme permet pour deux entiers D et a tels que $0 \leq a < D$ de séparer les facteurs de D qui ne sont pas commun avec ceux de a.

En considérant que l'on possède un algorithme `findRandomSolution`(A, b) qui retourne une solution rationnelle aléatoire du système $Ax = b$ on peut alors définir l'algorithme suivant pour calculer une solution diophantienne.

Algorithme `DiophantineSolve`(A,b)
Entrée : $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$
Sortie : $y \in \mathbb{Z}^n$ tel que $Ay = b$
Schéma
$y_0 :=$ `findRandomSolution`(A, b) ;
$y := y_0$;
tant que $\text{pgcd}(d(y), d(y_0)) \neq 1$ **faire**
$y_1 :=$ `findRandomSolution`(A, b) ;
si $\text{pgcd}(d(y_0), d(y), d(y_1)) < \text{pgcd}(d(y_0), d(y))$ **alors**
$a :=$ `Split`$(d(y_0), d(y))$;
$y := \dfrac{d(y)y + ad(y_1)y_1}{d(y) + ad(y_1)}$;
calculer $s, t \in \mathbb{Z}$ tel que $sd(y_0) + td(y) = 1$;
retourner $y = sd(y_0)y_0 + td(y)y$;

Pour que cette algorithme termine, il faut que le système $Ax = b$ soit consistant sur $\mathbb{Z}$. Une solution proposée par Giesbrecht, Lobo et Saunders [38] consiste à déterminer au préalable si le système est inconsistant sur Z. Comme pour le cas rationnel, le principe consiste à calculer un certificat établissant l'inconsistance du système sur $\mathbb{Z}$. L'idée utilisée par Giesbrecht, Lobo et Saunders consiste à dire que si il n'existe pas de solutions diophantiennes alors il existe des entiers d pour lesquels il n'existe pas de solution au système modulo d. Un certificat d'inconsistance sur Z est alors défini par un vecteur $u \in \mathbb{Z}^m$ tels que $u^T A \equiv 0 \mod d$ et $u^T b \not\equiv 0 \mod d$ où d est un diviseur du pgcd de tous les mineurs non nuls d'ordre maximal de A [38, lemme 3.2]. Une méthode probabiliste pour calculer un tel certificat est d'utiliser un vecteur aléatoire $v \in \{0, 1\}^r$ et de résoudre le système $u^T A = dv^T$ jusqu'à ce que $u^T A \equiv 0 \mod d$ et $u^T b \not\equiv 0 \mod$.

Toutefois, cette méthode permet de déterminer uniquement si le système est inconsistant sur $\mathbb{Z}$. Dans la prochaine section, nous nous intéressons à l'extension de ce certificat pour déterminer le dénominateur minimal de l'ensemble des solutions rationnelles et ainsi permettre à l'algorithme `DiophantineSolve` de calculer une solution rationnelle ayant un dénominateur minimal.

4.4.2 Certificat de minimalité

Le but d'un certificat de minimalité est de spécifier pour un système linéaire entier $Ax = b$ un entier positif d tel que d est égal au plus grand diviseur commun des dénominateurs de l'ensemble des solutions du système linéaire entier. En particulier, ce dénominateur minimal est un diviseur du plus grand coefficient de la forme de Smith de la matrice A. Si un tel certificat existe pour le système $Ax = b$ et que $d > 1$ alors il n'existe aucune solution diophantienne pour ce système. Pour un système linéaire $Ax = b$, nous utilisons la notation $d(A, b)$ pour représenter le dénominateur minimal de l'ensemble des solutions. À partir de ce dénominateur minimal on peut définir le certificat de minimalité suivant.

Définition 4.4.3. *Soient $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$ et $d(A,b)$ le dénominateur minimal du système $Ax = b$. Si $z \in \mathbb{Q}^m$ est tel que $z^T A \in \mathbb{Z}^{1\times n}$ et $d(z^T b) = d(A,b)$ alors z est un certificat de minimalité pour le système $Ax = b$.*

Pour calculer un tel certificat, Mulders et Storjohann proposent d'utiliser une approche probabiliste similaire à celle utilisée par Giesbrecht pour calculer une solution diophantienne. Le principe consiste à calculer aléatoirement plusieurs certificats validant différents diviseurs de $d(A,b)$ et à les combiner afin d'obtenir un certificat pour $d(A,b)$. Le lemme suivant établit la notion de certificat pour un diviseur du dénominateur minimal d'un système linéaire entier.

Lemme 4.4.4. *Si le système $Ax = b$ est consistant sur $\mathbb{Q}$ et qu'il existe $z \in \mathbb{Q}^m$ tel que $z^T A \in \mathbb{Z}$ alors z est un certificat du dénominateur $d(z^T b)$ pour l'ensemble des solutions de $Ax = b$ et $d(z^T b)$ divise $d(A,b)$.*

En calculant plusieurs certificats pour différents dénominateurs, on peut obtenir un certificat pour un dénominateur plus grand en les combinant à partir du lemme suivant.

Lemme 4.4.5. *Soient $z_1, z_2 \in \mathbb{Z}^m$ tels que $z_1^T A, z_2^T A \in \mathbb{Z}^{1\times n}$. Soient $n_1 = n(z_1^T b)$, $d_1 = d(z_1^T b)$, $n_2 = n(z_2^T b)$, $d_2 = d(z_2^T b)$ tels que $\mathrm{pgcd}(n_1, d_1) = \mathrm{pgcd}(n_2, d_2) = 1$. Soient $g = \mathrm{pgcd}(d_1, d_2)$, $l = \mathrm{ppcm}(d_1, d_2)$, s, $t \in \mathbb{Z}$ tels que*

$$\mathrm{pgcd}(n_1 \frac{d_2}{g}, n_2 \frac{d_1}{g}) = s n_1 \frac{d_2}{g} + t n_2 \frac{d_1}{g}.$$

Alors $z = s z_1 + t z_2$ vérifie $z^T A \in \mathbb{Z}^{1\times n}$ et $d(zb) = l$.

Les lemmes 4.4.4 et 4.4.5 correspondent aux lemmes 6 et 7 définis dans [61] où la preuve est spécifiée. À l'instar du calcul des solutions diophantiennes, il suffit maintenant de pouvoir calculer des certificats aléatoires afin que la combinaison de plusieurs de ces certificats permettent d'atteindre le dénominateur minimal. Pour calculer de tels certificats, l'approche consiste à résoudre un système linéaire rationnel défini à partir d'un second membre aléatoire à coefficients dans un sous-ensemble U de $\mathbb{Z}$. Nous considérons ici que le système est dense mais le même type d'approche est valide dans le cas creux [60].

Lemme 4.4.6. *Soient $U \subset \mathbb{Z}$, $P \in U^{n\times r}$ et $q \in U^{1\times m}$ deux matrices denses aléatoires. Si $z \in \mathbb{Q}^r$ tel que $z^T AP = q$ alors $\hat{z} = d(z^T A) z$ est un certificat du dénominateur $d(\hat{z}^T b)$ pour le système $Ax = b$ et $d(\hat{z}^T b)$ est un diviseur aléatoire de $d(A,b)$.*

Ce lemme s'appuie directement sur le lemme 8 et le corollaire 28 définis dans [61]. En utilisant ce lemme pour générer des certificats validant des dénominateurs de plus en plus grands, on peut espérer obtenir un certificat de minimalité du système $Ax = b$ à partir de $O(\log r + \log\log ||AP||)$ combinaison de certificats en prenant $U = \{0, 1\}$. Comme pour la combinaison de solutions rationnelles, cette convergence peut être ramenée à $O(1)$ en utilisant le sous-ensemble $U = \{0, 1, \ldots, M\}$ avec $M = \max(24, \lceil \log r^{\frac{r}{2}} ||A||^r \rceil)$ [61, corollaire 28]. Le coût pour calculer un certificat de minimalité du système $Ax = b$ est donc directement relié au coût de la résolution des systèmes linéaires entiers $z^T AP = q$ sur les rationnels, soit $O(r^3 \log r)$ dans le cas de systèmes linéaires denses où $r = \mathrm{rang}(A)$ (voir §4.3).

On retrouve ici le même problème que pour la combinaison de solutions rationelles. L'utilisation directe du lemme 4.4.5 entraîne un grossissement des certificats au fur et à mesure des combinaisons. Afin d'éviter ce grossissement, Mulders et Storjohann utilise alors une combinaison similaire à celle du lemme 4.4.2.

Lemme 4.4.7 (voir p.501 [61]). *Soient* $z_1, z_2 \in \mathbb{Z}^m$ *tels que* $z_1^T A, z_2^T A \in \mathbb{Z}^{1\times n}$. *Soient* $n_1 = n(z_1^T b)$, $d_1 = d(z_1^T b)$, $n_2 = n(z_2^T b)$, $d_2 = d(z_2^T b)$ *tels que* $\mathrm{pgcd}(n_1, d_1) = \mathrm{pgcd}(n_2, d_2) = 1$. *Soient* $g = \mathrm{pgcd}(d_1, d_2)$, $l = \mathrm{ppcm}(d_1, d_2)$ *et* $a \in \mathbb{Z}$ *tel que*

$$\mathrm{pgcd}(\frac{n_1 d_2}{g} + a\frac{n_2 d_1}{g}, l) = \mathrm{pgcd}(\frac{n_1 d_2}{g}, \frac{n_2 d_1}{g}, l) = 1.$$

Alors $z = z_1 + a z_2$ *vérifie* $z^T A \in \mathbb{Z}^{1\times n}$ *et* $d(z^T b) = l$.

Ce lemme permet donc de calculer un certificat de la forme $z = z_1 + a_2 z_2 + \ldots + a_k z_k$ et donc de limiter le grossissement des entrées du certificat [61, §6.1]. En considérant que l'on possède un algorithme `findRandomCertificate`(A, b, U) pour calculer un certificat minimal aléatoire du système $Ax = b$ à partir du sous-ensemble U de $\mathbb{Z}$, on peut alors définir l'algorithme suivant pour calculer le certificat de minimalité du dénominateur $d(A, b)$.

Algorithme `MinimalCertificate`(A,b,k,U)
Entrée : $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$, $k \in \mathbb{N}$, $U \subset \mathbb{Z}$
Sortie : $z \in \mathbb{Q}^m$ tel que $z^T A \in \mathbb{Z}^{1\times n}$ et $d(z^T b) = d(A, b)$
Schéma

```
z_0 := findRandomCertificate(A, b, U) ;
z := z_0 ;
Pour i de 1 à k faire
    z_1 := findRandomCertificate(A, b, U) ;
    l := ppcm(d(z^T b), d(z_1^T b)) ;
    si l > d(z^T b) alors
        g := pgcd(d(z^T b), d(z_1^T b)) ;
        d := d(z^T b) ;
        n := n(z_1^T b) ;
        a := Split(l, nd/g) ;
        z := z + a z_1 ;
retourner z ;
```

Dans cette algorithme le paramètre k spécifie le nombre d'itérations nécessaires pour obtenir un certificat du dénominateur minimal. Soit S l'ensemble des nombres premiers qui divisent le plus grand coefficient de la forme de Smith de A. La probabilité de réussite de l'algorithme `MinimalCertificate` est de

$$\begin{array}{rl} 1 - \#S(\frac{9}{10})^k & \text{pour } U = \{0, 1\}, \\ 1 - (\frac{9}{10})^k + \sum_{p\in S, p>2} (\frac{2}{p} + \frac{2}{\#U})^k & \text{pour } \#U \geq 25, \end{array}$$

en s'appuyant sur la proposition 24 définie dans [61].

En combinant l'algorithme `DiophantineSolve` et `MinimalCertificate`, on peut définir un algorithme de résolution de systèmes linéaires entiers certifiée. L'idée est de faire converger les deux algorithmes jusqu'à ce que la solution obtenue possède un dénominateur égal à celui validé par le certificat. On obtient ainsi une solution du système ayant un dénominateur minimal. Si

le but est de calculer une solution diophantienne il suffit alors d'itérer l'algorithme jusqu'à ce qu'une solution diophantienne soit calculée ou bien jusqu'à ce qu'un certificat détermine un dénominateur minimal différent de un et donc établit l'inconsistance du système sur $\mathbb{Z}$.

4.4.3 Implantations et performances

L'implantation de l'algorithme `DiophantineSolve` est immédiate en utilisant notre interface de résolution de système de la section 4.2. Le principe consiste à calculer des solutions rationnelles aléatoires du système en utilisant la fonction `findRandomSolution` et à utiliser le lemme 4.4.2 pour générer des solutions ayant des dénominateurs de plus en plus petits. Cette implantation est disponible dans la bibliothèque LinBox à partir de la classe `DiophantineSolver` qui définit un domaine de calcul de solutions diophantiennes. Cette classe est définie de façon générique pour tous les types de solveurs rationnels conformes à l'interface `RationalSolver`. La fonction `diophantineSolve` de cette classe permet de calculer une solution minimale d'un système linéaire entier. À l'instar des solveurs rationnels, cette fonction utilise les trois niveaux de calcul `SL_MONTECARLO, SL_LASVEGAS, SL_CERTIFIED` définis dans la section 4.2.1. La différence par rapport à la résolution rationnnelle est que les niveaux `SL_LASVEGAS` et `SL_CERTIFIED` assure que la solution calculée est bien de dénominateur minimal. La différence entre ces deux niveaux provient du fait que le résolution certifiée entraîne la construction du certificat de minimalité alors que le niveau LAS VEGAS calcule uniquement les certificats intermédiaires pour déterminer le dénominateur minimal.

Nous proposons dans le code suivant un exemple d'utilisation de la classe `DiophantineSolver` pour calculer une solution diophantienne.

Code 4.5 – Utilisation de LinBox pour la résolution de systèmes linéaires diophantiens

```
int main(int argc, char **argv) {
  typedef Modular<double>          Field;
  typedef PID_integer               Ring;
  typedef Field::Element         Element;
  typedef Ring::Element         Integer;
  typedef DenseMatrix<Ring>      Matrix;
  typedef std::vector<Integer> Vector;

  ifstream inA(argv[1]), inb(argv[2]);
  ofstream outx(argv[3]);
  Ring     Z;

  // creation and reading of A and b over Z
  Matrix  A(Z);
  A.read(inA);
  Vector b(A.rowdim());
  Vector::iterator it = b.begin();
  for (; it != b.end(); ++it)
     Z.read(inb,*it);

  // creation of Rational Solver with Dixon's method
  typedef RationalSolver<Ring,Field,RandomPrime,DixonTraits> RatSolve;
  RatSolve QSolver(Z);

  // creation of Diophantine Solver with Dixon's method
  DiophantineSolver<RatSolve> ZSolver(QSolver);
```

```
    // creation of solution numerators and common denominator
    Vector x(A.coldim());
    Integer denom;

    // number of prime to try
    size_t numPrimes = 5;

    // set the level of computation
    SolverLevel level= SL_CERTIFIED;

    // find a diophantine solution to the system Ax=b using certified computation
    SolverReturnStatus status;
    status = ZSolver.diophantineSolve(x, denom, A, b, numPrimes, level);

    //writing the solution if status is ok
    if (status == SS_OK){
      outx<<"numerator:\n";
      it = x.begin();
      for (; it != x.end(); ++it)
         Z.write(outx,*it)<<endl;
      outx<<"denominator:\n";
      Z.write(outx,denom);
    }
    return 0;
}
```

Ce code montre que l'utilisation de notre interface permet de résoudre des systèmes linéaires diophantiens à un haut niveau. En particulier, le paramètrage des calculs dans cette interface permet de définir des codes simples bénéficiant des algorithmes les plus adaptés. Par exemple, si l'on désire utiliser une méthode de calcul de type Monte Carlo pour permettre de meilleure performance il suffit simplement d'initialiser la variable `level` avec la valeur `SL_MONTECARLO`. De même, si l'on souhaite utiliser une autre méthode de calcul des développements p-adiques pour la résolution des systèmes linéaires diophantiens il suffit de changer le trait `DixonTraits` définissant la méthode dans le type du solveur rationnel `RationalSolver`.

En utilisant ce code pour la matrice A et le vecteur b suivant

$$A = \begin{bmatrix} 11 & 13 & 4 \\ 5 & 7 & 9 \end{bmatrix}, \quad b = \begin{bmatrix} 7 \\ 10 \end{bmatrix},$$

on obtient la solution diophantienne

$$x = \begin{bmatrix} -29 \\ 26 \\ -3 \end{bmatrix}$$

en combinant deux solutions rationnelles de la façon suivante :

$$y_1 = \begin{bmatrix} -27/4 \\ 25/4 \\ 0 \end{bmatrix}, \quad y_2 = \begin{bmatrix} 2/3 \\ -1/3 \\ 3/3 \end{bmatrix} \Longrightarrow x = n(y_1) - n(y_2) = \begin{bmatrix} -27 & -2 \\ 25 & +1 \\ 0 & -3 \end{bmatrix}.$$

Nous évaluons maintenant les performances obtenues avec notre implantation dans le cas de systèmes linéaires denses. Nous utilisons des systèmes linéaires aléatoires surdéterminés afin que

le probabilité d'inconsistance sur $\mathbb{Z}$ soit faible. Ainsi, nous pouvons utiliser un calcul de type `SL_MONTECARLO` qui permet d'obtenir les meilleures performances pour notre implantation. En comparaison, les deux autres niveaux de calcul (`SL_LASVEGAS, SL_CERTIFIED`) sont en moyenne deux fois plus lents. Cela provient du fait que le calcul du certificat de minimalité double le nombre de résolutions de système linéaire sur $\mathbb{Q}$ qui est le coût dominant de la méthode.

Afin d'évaluer ces performances, nous nous comparons avec la bibliothèque IML [12]. Cette bibliothèque propose l'implantation d'une approche récente [13] pour le calcul de solutions diophantiennes. Le principe consiste à déterminer une solution ayant un dénominateur minimal en utilisant la forme d'Hermite modulaire d'une solution rationnelle du système linéaire augmenté par 10 colonnes aléatoires. L'approche utilisée est probabiliste et s'appuie sur le fait que la compression à 10 colonnes conserve les informations sur le dénominateur minimal contenues dans la forme d'Hermite de la matrice $[A|b]$. Plus précisément, cette implantation permet de déterminer une solution diophantienne à partir d'une résolution de système linéaire matriciel (avec 11 colonnes) par l'algorithme de Dixon et en calculant la forme d'Hermite modulaire de la solution matricielle [23].

La table 4.5 illustre les temps de calcul pour notre implantation et celle de la biliothèque IML pour des systèmes aléatoires définis à partir de données sur 3 bits, 32 bits et 100 bits. L'architecture utilisée est un Pentium Xeon 2.66ghz possédant 1Go de RAM et 1Go de mémoire swap. Les résultats dans le tableau sont exprimés en secondes pour les deux implantations. Pour notre implantation, nous spécifions le nombre de solutions nécessaires pour construire la solution diophantienne. Ce nombre correspond à la valeur entre parenthèse, si celle-ci n'est pas spécifiée cela signifie que le nombre de solutions utilisées est de deux. Lorsque les temps de calcul ne sont pas spécifiés cela signifie que l'espace mémoire de la machine n'est pas suffisant pour calculer une solution du système.

	3 bits		32 bits		100 bits	
	LinBox	IML	LinBox	IML	LinBox	IML
200×250	$0.92s$	$1.24s$	(4) $10.26s$	$8.18s$	$16.57s$	$38.87s$
200×500	$1.23s$	$1.34s$	$5.18s$	$8.59s$	$17.03s$	$40.26s$
400×450	(3) $7.77s$	$6.93s$	$26.40s$	$44.00s$	(3) $139.69s$	$247.58s$
400×800	$5.72s$	$7.18s$	$27.20s$	$49.93s$	$95.23s$	$252.06s$
600×800	$13.33s$	$21.16s$	$72.72s$	$130.79s$	$267.10s$	$691.80s$
600×1000	$14.42s$	$20.59s$	$73.77s$	$131.53s$	(3) $402.92s$	$675.96s$
800×1000	$27.49s$	$42.49s$	$131.61s$	$278.99s$	(3) $886.98s$	$1387.24s$
800×1500	$29.88s$	$45.47s$	$131.59s$	$308.50s$	$644.91s$	$1378.29s$
1200×1400	$78.57s$	$124.79s$	$390.93s$	$814.68s$	-	-
1200×2000	$86.17s$	$128.66s$	(3) $634.87s$	$807.06s$	-	-
1800×2400	$215.77s$	$425.83s$	(3) $2020.88s$	$2533.30s$	-	-

TAB. 4.5 – Comparaisons des temps de calculs de solutions diophantiennes (`Xeon-2.66Ghz`)

D'après les résultats du tableau 4.5 on voit que notre approche est plus efficace en pratique que celle utilisée par la bibliothèque IML. On s'aperçoit que la plupart du temps il suffit de seulement deux solutions pour recouvrir une solution diophantienne. Cela signifie que l'approche de la bibliothèque IML est un peu pessimiste en utilisant d'emblée onze solutions. En pratique la bibliothèque IML est moins performante que notre implantation car elle calcule trop de solutions par rapport à ce dont elle a réellement besoin pour déterminer une solution diophantienne. Dans

le cas où le nombre de solutions nécessaires est supérieur à deux (seulement 15% de nos tests), on peut voir qu'en moyenne notre approche reste meilleure car le nombre de solutions utilisées reste petit par rapport à 11 (3 ou 4).

Afin de comparer plus précisément ces deux implantations, nous nous intéressons maintenant à la résolution de systèmes linéaires issus de matrices possédant des noyau de petite dimension. Pour obtenir de telles matrices avec une bonne probabilité, nous utilisons des matrices aléatoires rectangulaire ou la différence des dimensions est petite. Du fait que les noyaux sont plus petits, le nombre de solutions nécessaires pour calculer une solution diophantienne risque d'être plus important. Notons que dans le cas où la taille du noyau de la matrice est inférieure à dix, l'implantation proposée par la bibliothèque IML s'adapte et utilise un membre droit possédant un nombre de colonnes égal à la dimension du noyau plus un.

	LinBox		IML	
	temps	nbr solutions	temps	nbr solutions
400×401	$113.10s$	2 (2)	$57.57s$	2
400×405	$113.41s$	2 (2)	$126.94s$	6
400×410	$117.19s$	2 (2)	$214.88s$	11
400×450	$115.22s$	2 (2)	$243.94s$	11
800×801	$1829.36s$	5 (3)	$324.48s$	2
800×805	$1131.38s$	3 (3)	$724.57s$	6
800×810	$2949.75s$	8 (3)	$1247.84s$	11
800×850	$734.62s$	2 (2)	$1397.56s$	11

TAB. 4.6 – Solutions diophantiennes en fonction de la taille du noyau pour des systèmes linéaires à coefficients sur 100 bits (`Xeon-2.66Ghz`)

La table 4.6 illustre le résultat de nos tests pour des matrices rectangulaires de plein rang. La colonne "nbr solutions" dans cette table spécifie le nombre de solutions calculées par les deux implantations. Pour implantation "LinBox ", nous précisons entre parenthèses quel est le nombre de solutions réellement nécessaires à la construction de la solution diophantienne parmi l'ensemble des solutions rationnelles calculées. D'après ces résultats on s'aperçoit que la bibliothèque IML possède un meilleur comportement que notre implantation lorsque la taille du noyaux est inférieure à 10. Cela s'explique par le fait que plus le noyau est petit, plus il est difficile de trouver des solutions rationnelles aléatoires permettant d'aboutir à une solution diophantienne. On peut voir par exemple que pour le système d'ordre 800×810 notre approche a nécessité le calcul de 8 solutions rationnelles pour construire une solution diophantienne alors que seulement 3 étaient suffisantes. Lorsque le nombre de solutions rationnelles est exactement le même pour les deux implantations (i.e. 400×401) on s'aperçoit que la bibliothèque IML permet d'obtenir les meilleures performances. Cette différence provient du fait que le coût principal des implantations réside dans le calcul des développements p-adiques des solutions. En particulier, la bibliothèque IML calcule l'ensemble des solutions rationnelles à partir d'un seul développement p-adique matriciel alors que notre implantation utilise un développement p-adique vectoriel pour chaque solution. En remplaçant les produits matrice-vecteur dans l'algorithme de Dixon (§4.3) par des produits matriciel la bibliothèque IML peut tirer parti du niveau 3 des BLAS tandis que notre implantation ne bénéficie que du niveau 2.

En conclusion, on peut dire que notre implantation est en pratique plus adaptative que celle proposée par la bibliothèque IML. Le nombre de solutions rationnelles nécessaires en moyenne

pour calculer une solution diophantienne est de l'ordre de deux. En particulier, on préférera utiliser notre implantation lorsque les systèmes linéaires sont susceptibles de posséder un noyau suffisamment important (supérieur à 50 dans nos tests) alors que l'approche par blocs de la bibliothèque IML se révélera plus intéressante lorsque les noyaux seront petits.

Conclusion et perspectives

Nous avons développé dans ce document plusieurs outils logiciels de base pour implanter efficacement des calculs en algèbre linéaire exacte. Cela comprend l'arithmétique des corps finis et les algorithmes d'algèbre linéaire sur un corps fini utilisant une réduction au produit de matrices. Notamment, nous avons proposé et validé l'utilisation de mécanismes d'abstraction par des archétypes de données pour développer des codes facilitant le changement d'arithmétiques. Ce mécanisme nous a permis de fournir plusieurs implantations d'arithmétiques de corps finis facilement utilisables et interchangeables à l'intérieur des codes de la bibliothèque LinBox. En particulier, nous avons montré que le choix de l'arithmétique dépend fortement de la taille des corps finis. L'utilisation d'une représentation logarithmique, aussi bien pour les corps premiers que pour les extensions algébriques, s'avère être la représentation la plus performante pour calculer sur les corps finis de petite taille. Dès lors que l'on s'intéresse à des corps ne pouvant plus être représentés à partir de mots machines, l'utilisation des entiers multiprécisions de la bibliothèque GMP ou des polynômes de la bibliothèque NTL reste l'un des meilleurs moyens d'obtenir des performances satisfaisantes.

Un autre point important de notre travail a été le développement de routines de calculs sur les corps finis pour l'ensemble des problèmes classiques de l'algèbre linéaire dense. Notre approche de calcul hybride "exact/numérique" nous laisse penser qu'il est raisonnablement possible d'espérer un jour obtenir des bibliothèques de calcul sur les corps finis similaires à celles développées par les bibliothèques numériques BLAS/LAPACK. En outre, notre approche a déjà permis de proposer des performances jamais égalées sur les corps finis et qui se rapprochent de celles obtenues en calcul numérique. Ces résultats ont des conséquences importantes sur le développement de solutions haut niveau très performantes pour le calcul sur les entiers. En particulier, l'utilisation maximale du calcul numérique pour effectuer les opérations de base comme le produit matrice-vecteur ou le produit matriciel permet d'atteindre une meilleure efficacité qu'avec des calculs en multiprécision. Nous avons démontré que c'est le cas pour la résolution de systèmes linéaires diophantiens denses en proposant des implantations basées sur nos routines de calcul hybride "exact/numérique" qui permettent d'obtenir des performances très convaincantes ; de ce fait, nous avons pu valider nos briques de base autant au niveau efficacité que simplicité d'utilisation. Cette thèse offre donc un crédit important à l'utilisation et à la réutilisation de la bibliothèque LinBox pour les applications en calcul formel nécessitant de très hautes performances de calcul.

Toutefois, les solutions que nous avons apportées ne constituent pas l'aboutissement final du développement de la bibliothèque LinBox. Il reste certaines directions vers lesquelles il nous paraît important de s'orienter. En particulier, les résultats récents en complexité binaire pour la résolution de problème en algèbre linéaire sur les entiers proposent, comme pour les corps finis, des réductions au produit de matrices entières. La question qui nous intéresse est de savoir s'il est possible de développer des routines de multiplication de matrices sur de très grands entiers suffisamment performantes pour permettre l'utilisation efficace de ces algorithmes en pratique ; dans ce cas, quelles sont les stratégies à utiliser : algorithmes de multiplication de matrices rapides, multiplications d'entiers à base de FFT, réutilisation des BLAS, théorème des restes chinois, représentation q-adique ?

Un autre problème qui nous semble essentiel pour la réutilisation de la bibliothèque LinBox concerne l'interopération automatique avec des langages de plus haut niveau, généralement interprétés, comme MAPLE. L'utilisation des archétypes est ici un bon moyen pour fournir des briques de base abstraites. Néanmoins, un problème qui se pose encore est de savoir si l'on peut intégrer tous les composants du calcul formel à partir d'archétypes ou si au contraire il est préférable d'utiliser un protocole standard d'échange de données, et dans ce cas, quel doit être le niveau de description des objets.

Bibliographie

[1] American National Standards Institute and Institute of Electrical and Electronics Engineers : IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[2] Daniel V. Bailey et Christof Paar : Optimal extension fields for fast arithmetic in public-key algorithms. *Lecture Notes in Computer Science*, 1462:472–485, 1998.

[3] Daniel V. Bailey et Christof Paar : Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[4] Paul Barret : Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. *In* A. M. Odlyzko, éditeur : *Advances in Cryptology : Proceedings of Crypto '86*, Lecture Notes in Computer Science, pages 311–323, 1987.

[5] Laurent Bernardin, Bruce Char et Erich Kaltofen : Symbolic computation in Java : An appraisement. *In* Sam Dooley, éditeur : *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, Vancouver, Canada.* ACM Press, New York, juillet 1999.

[6] Dario Bini et Victor Pan : *Polynomial and Matrix Computations, Volume 1 : Fundamental Algorithms.* Birkhauser, Boston, 1994.

[7] Morgan Brassel et Clément Pernet : Ludivine : une divine factorisation lu. Rapport technique, Université Joseph Fourier, 2002.

[8] James R. Bunch et John E. Hopcroft : Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, pages 231–236, 1974.

[9] P. Bürgisser, M. Clausen et M.A. Shokrollahi : *Algebraic Complexity Theory*, volume 315 de *Grundlehren der mathematischen Wissenschaften.* Springer-Verlag, 1997.

[10] Jacquelin Charbonnel : *Langage C++ - les spécifications ANSI/ISO expliquées.* Dunod, deuxième édition, 1997.

[11] Zhuliang Chen et Arne Storjohann : Effective reductions to matrix multiplication, juillet 2003. ACA'2003, 9th International Conference on Applications of Computer Algebra, Raleigh, North Carolina State University, USA.

[12] Zhuliang Chen et Arne Storjohann : IML : Integer matrix library, 2004. `www.scg.uwaterloo.ca/ z4chen/iml.html`.

[13] Zhuliang Chen et Arne Storjohann : Implementation of certified linear system solving for integer matrices. Poster at ISSAC'04 : International Symposium on Symbolic and Algebraic Computation, Santander, Spain, juillet 2004.

[14] Arjeh M. Cohen, Xiao-Shan Gao et Nobuki Takayama, éditeurs. *ICMS'2002. Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, août 2002. World Scientific.

[15] Henri Cohen : *A course in computational algebraic number theory*, volume 138 de *Graduate Texts in Mathematics.* Springer-Verlag, 1993.

[16] Stephen A. Cook : On the minimum computation time of functions. Mémoire de D.E.A., Harvard University, mai 1966.

[17] Don Coppersmith : Solving homogeneous linear equations over GF[2] via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, janvier 1994.

[18] Don Coppersmith et Shmuel Winograd : Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[19] Intel Corporation : *Intel(R) Architecture Software Developer's Manual, Volume 2 : Instruction Set Reference Manual.* URL http://www.intel.com/design/intarch/manuals/243191.htm.

[20] David Defour : *Fonctions élémentaires : algorithmes et implantations efficaces pour l'arrondi correct en double précision.* Thèse de doctorat, École Normale Supérieure de Lyon, septembre 2003.

[21] Michel Demazure : *Cours d'algèbre. Primalité, Divisibilité, Codes*, volume XIII de *Nouvelle biliothèque Mathématique.* Cassini, Paris, 1997.

[22] John D. Dixon : Exact solution of linear equations using p-adic expansions. *Numerische Mathematik*, 40:137–141, 1982.

[23] P. D. Domich, R. Kannan et Jr. L. E. Trotter : Hermite normal form computation using modulo determinant arithmetic. *Mathematics of Operations Research*, 12(1):50–59, 1987.

[24] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling et Iain Duff : A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, mars 1990. `www.acm.org/pubs/toc/Abstracts/0098-3500/79170.html`.

[25] Jean-Guillaume Dumas : *Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, décembre 2000. `ftp ://ftp.imag.fr/pub/Mediatheque.IMAG/-theses/2000/Dumas.Jean-Guillaume`.

[26] Jean-Guillaume Dumas : Efficient dot product over word-size finite fields. *In Proc. CASC'2004 :Computer Algebra in Scientific Computing*, juillet 2004.

[27] Jean-Guillaume Dumas, Thierry Gautier, Mark Giesbrecht, Pascal Giorgi, Bradford Hovinen, Erich Kaltofen, B. David Saunders, Will J. Turner et Gilles Villard : LinBox : A generic library for exact linear algebra. *In* Cohen *et al.* [14], pages 40–50.

[28] Jean-Guillaume Dumas, Thierry Gautier et Clément Pernet : Finite field linear algebra subroutines. *In* Teo Mora, éditeur : *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, juillet 2002.

[29] Jean-Guillaume Dumas, Pascal Giorgi et Clément Pernet : FFPACK : Finite Field Linear Algebra Package. *In* Gutierrez [43], pages 119–126.

[30] Jean-Guillaume Dumas, Pascal Giorgi et Clément Pernet : FFPACK : Finite Field Linear Algebra Package. Research Report, LIP-RR2004-02, janvier 2004. preliminary version.

[31] Jean-Guillaume Dumas, Frank Heckenbach, B. David Saunders et Volkmar Welker : *Simplicial Homology, a share package for GAP*, mars 2000.

[32] Jean-Guillaume Dumas et Jean-Louis Roch : On parallel block algorithms for exact triangularizations. *Parallel Computing*, 28(11):1531–1548, novembre 2002.

[33] Jean Guillaume Dumas, B. David Saunders et Gilles Villard : On efficient sparse integer matrix Smith normal form computation. *JSC*, 32(1/2):71–99, 2001.

[34] Wayne Eberly, Mark Giesbrecht et Gilles Villard : On computing the determinant and Smith form of an integer matrix. *In Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 675. IEEE Computer Society, 2000.

[35] Alain Bernard Fontaine : *La bibliothèque standard STL du C++.* Interedition, 1997.

[36] Joachim von zur Gathen et Jürgen Gerhard : *Modern Computer Algebra.* Cambridge University Press, New York, USA, 1999.

[37] Thierry Gautier : *Calcul Formel et Parallélisme : Conception du Système* GIVARO *et Applications au Calcul dans les Extensions Algébriques.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 1996. http ://www-apache.imag.fr/software/givaro.

[38] Mark Giesbrecht, Austin Lobo et B. David Saunders : Certifying inconsistency of sparse linear systems. *In* Oliver Gloor, éditeur : *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, Rostock, Germany*, pages 113–119. ACM Press, New York, août 1998.

[39] Mark W. Giesbrecht : Efficient parallel solution of sparse systems of linear diophantine equations. *In Parallel Symbolic Computation (PASCO'97)*, pages 1–10, Maui, Hawaii, juillet 1997.

[40] Pascal Giorgi, Claude-Pierre Jeannerod et Gilles Villard : On the complexity of polynomial matrix computations. *In* Rafael Sendra, éditeur : *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, Philadelphia, Pennsylvania, USA*, pages 135–142. ACM Press, New York, août 2003.

[41] Torbjörn Granlund : *The GNU multiple precision arithmetic library*, 2002. Version 4.1, `www.swox.com/gmp/manual`.

[42] Fred G. Gustavson, Andre Henriksson, Isak Jonsson et Bo Kaagstroem : Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Lecture Notes in Computer Science*, 1541:195–206, 1998.

[43] Jaime Gutierrez, éditeur. *ISSAC'2004. Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*, juillet 2004. ACM Press, New York.

[44] Bradford Hovinen : Blocked lanczos-style algorithms over small finite fields. Mémoire de D.E.A., University of Waterloo, 2004.

[45] Xiaohan Huang et Victor Y. Pan : Fast rectangular matrix multiplications and improving parallel matrix computations. *Journal of Complexity*, 14(2):257–299, 1998.

[46] Oscar H. Ibarra, Shlomo Moran et Roger Hui : A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, mars 1982.

[47] 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 J. Comput.*, 18(4):658–669, 1989.

[48] Erich Kaltofen : Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, avril 1995.

[49] Erich Kaltofen, Mukkai S. Krishnamoorthy et B. David Saunders : Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, 136:189–208, 1990.

[50] Erich Kaltofen et B. David Saunders : On Wiedemann's method of solving sparse linear systems. *In Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, volume 539 de *LNCS*, pages 29–38, octobre 1991.

[51] Erich Kaltofen et Gilles Villard : On the complexity of computing determinants, 2003. URL http://www.ens-lyon.fr/ gvillard/BIBLIOGRAPHIE/PDF/KaVi03:2697263.pdf. Research Report 2003-36, Laboratoire LIP, ENS Lyon, France.

[52] Anatolii Karatsuba et Yu Ofman : Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.

[53] Donald E. Knuth : *Seminumerical Algorithms*, volume 2 de *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, troisième édition, 1997.

[54] Cornelius Lanczos : Solutions of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standard*, 49:33–53, 1952.

[55] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung et Daniel J. Yoo : Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions of Mathematical Software*, 28(2):152–205, 2002.

[56] James L. Massey : Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, IT-15:122–127, 1969.

[57] Robert T. Moenck et John H. Carter : Approximate algorithms to derive exact solutions to systems of linear equations. *In Proc. EUROSAM'79, volume 72 of Lecture Notes in Computer Science*, pages 65–72, Berlin-Heidelberg-New York, 1979. Springer-Verlag.

[58] Michael B. Monagan, Keith O. Geddes, K. M. Heal, George Labahn et S. M. Vorkoetter : *Maple V Programming Guide*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998. With the assistance of J. S. Devitt, M. L. Hansen, D. Redfern, and K. M. Rickard.

[59] Peter L. Montgomery : Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, avril 1985.

[60] Thom Mulders : Certified sparse linear system solving. Rapport technique 357, ETH Zurich, décembre 2000.

[61] Thom Mulders et Arne Storjohann : Certified dense linear system solving. *Journal of Symbolic Computation*, 37(4):485–510, 2004.

[62] David R. Musser et Atul Saini : *STL Tutorial and Reference Guide : C++ Programming with the Standard Template Library*. Addison-Wesley, Reading (MA), USA, 1996.

[63] Morris Newman : *Integral Matrices*, volume 45 de *Pure and Applied Mathematics, a Series of Monographs and Textbooks*. Academic Press, 1972.

[64] Victor Y. Pan : Superfast algorithms for singular toeplitz/hankel-like matrices. Rapport technique, Department of Mathematics and Computer Science Lehman College of CUNY, Lehman College of CUNY, Bronx, NY 10468 USA, mai 2003.

[65] Mike Paterson et Larry J. Stockmeyer : On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[66] Clément Pernet : Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Rapport technique, Laboratoire Informatique et Distribution, juillet 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.

[67] Clément Pernet : Calcul du polynôme caractéristique sur des corps finis. Mémoire de D.E.A., University of Delaware, juin 2003.

[68] Gabriel D. Reis, Bernard Mourrain, Fabrice Rouillier et Philippe Trébuchet : An environment for symbolic and numeric computation. *In* Cohen *et al.* [14], pages 239–249.

[69] Ronald L. Rivest, Adi Shamir et Leonard Adleman : A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM*, 21(2), 1978.

[70] B. David Saunders et Zhendong Wan : Smith normal form of dense integer matrices, fast algorithms into practice. *In* Gutierrez [43].

[71] Arnold Schönhage, A. F. W. Grotefeld et E. Vetter : *Fast Algorithms : A Multitape Turing Machine Implementation.* BI-Wissenschaftsverlag, Mannheim, 1994.

[72] Arnold Schönhage et Volker Strassen : Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[73] Victor Shoup : NTL 5.3 : A library for doing number theory, 2002. `www.shoup.net/ntl`.

[74] Jeremy G. Siek : *A modern framework for portable high performance numerical linear algebra.* Master's thesis, University of Notre Dame, mai 1999.

[75] Arne Storjohann : Near optimal algorithms for computing Smith normal forms of integer matrices. *In* Yagati N. Lakshman, éditeur : *ISSAC '96 : Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, Zurich, Switzerland*, pages 267–274, juillet 1996.

[76] Arne Storjohann : *Algorithms for Matrix Canonical Forms.* Thèse de doctorat, Institut für Wissenschaftliches Rechnen, ETH-Zentrum, Zürich, Switzerland, novembre 2000.

[77] Arne Storjohann : The shifted number system for fast linear algebra on integer matrices. Rapport technique CS-2004-18, School of Computer Science, University of Waterloo, 2004.

[78] Arne Storjohann et Thom Mulders : Fast algorithms for linear algebra modulo n. *Lecture Notes in Computer Science*, 1461:139–150, 1998.

[79] Volker Strassen : Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[80] Bjarne Stroustrup : *The C++ Programming Language.* Addison-Wesley, third édition, 1997.

[81] the GAP group : *The GAP reference manual*, 2004. release 4.4.2.

[82] Emmanuel Thomé : *Algorithmes de calcul de logarithme discret dans les corps finis.* Thèse de doctorat, École polytechnique, LIX, mai 2003.

[83] Andrei L. Toom : The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Math*, 3:714–716, 1963.

[84] William J. Turner : *Black Box Linear Algebra with Linbox Library.* Thèse de doctorat, North Carolina State University, mai 2002.

[85] Gilles Villard : A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport technique 975–IM, LMC/IMAG, avril 1997.

[86] Gilles Villard : *Algorithmique en algèbre linéaire exacte.* Mémoire d'habilitation à diriger des recherches, Université Claude Bernard Lyon I, mars 2003.

[87] Joachim von zur Gathen : Functional decomposition of polynomials : The wild case. *J. Symb. Comput.*, 10(5):437–452, 1990.

[88] Paul S. Wang : A p-adic algorithm for univariate partial fractions. *In Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 212–217. ACM Press, 1981.

[89] Xinmao Wang et Victor Y. Pan : Acceleration of euclidean algorithm and rational number reconstruction. *SIAM Journal on Computing*, 32(2):548–556, 2003.

[90] R. Clint Whaley, Antoine Petitet et Jack J. Dongarra : Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, janvier 2001.

[91] Douglas H. Wiedemann : Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, janvier 1986.

Annexe A

Code LinBox

A.1 Développements p-adiques de systèmes linéaires entiers

Code A.6 – Interface LinBox des conteneurs de développements p-adiques

```
template< class Ring, class IMatrix>
class LiftingContainerBase {

  public:
     typedef typename Ring::Element            Integer;
     typedef std::vector<Integer>              IVector;
     typedef LiftingIterator<IVector>  const_iterator;

  protected:
     const IMatrix              &_A;
     Ring                        _R;
     Integer                     _p;
     IVector                     _b;
     VectorDomain<Ring>        _VDR;
     size_t                  _length;
     Integer               _numbound;
     Integer               _denbound;
     MatrixApplyDomain<Ring,IMatrix>     _MAD;

     friend class LiftingIterator;

  public:
     template <class Prime_Type, class Vector1>
     LiftingContainerBase (const Ring        &R,
                           const IMatrix     &A,
                           const Vector1     &b,
```

```
                                const Prime_Type  &p);

      // function to solve Ax=residu mod p
      virtual IVector& nextdigit (IVector& x, const IVector& residu) const = 0;

      // return iterator on the beginning
      const_iterator begin() const {
        return const_iterator(*this);}

      // return iterator on the end
      const_iterator end() const {
        return const_iterator (*this, _length);}

      // return length of container
      const size_t length() const { return _length;}

      // return size of container
      const size_t size() const { return _A.coldim();}

      // return the ring
      const Ring& ring() const { return _R;}

      // return the prime
      const Integer& prime () const { return _p;}

      // return the bound for the numerator
      const Integer numbound() const { return _numbound;}

      // return the bound for the denominator
      const Integer denbound() const { return _denbound;}
};
```

Code A.7 – Interface LinBox des itérateurs de développements p-adiques

```
template <class IVector>
class const_iterator {
  private:
     IVector                          _res;
     const LiftingContainerBase      &_lc;
     size_t                      _position;

  public:
     const_iterator(const LiftingContainerBase &lc, size_t end=0)
       : _res(lc._b), _lc(lc), _position(end) {}

     bool next (IVector &digit)  {

       // Compute next digit
       _lc.nextdigit(digit,_res);

       // Update residu
       IVector v2 (_res.size());
       _lc._MAD.applyV(v2,digit);
       _lc._VDR.subin (_res, v2);
```

```
      typename IVector::iterator p0;
      for ( p0 = _res.begin(); p0 != _res.end(); ++ p0, )
         _lc._R.divin(*p0, _lc._p);

      // increase position of the iterator
      _position++;

      return true;
    }

    bool operator != (const const_iterator &iterator) const {
      if ( &_lc != &iterator._lc)
         assert("Trying to compare different LiftingIterator, abort\n");
      return _position != iterator._position;
    }

    bool operator == (const const_iterator &iterator) const {
      if ( &_lc != &iterator._lc)
         assert("Trying to compare different LiftingIterator, abort\n");
      return _position == iterator._position;
    }
  };
```

A.2 Reconstruction de la solution rationnelle

Code A.8 – Évaluation des développements p-adiques

```
void PolEval(Vector& y,
             std::vector<Vector>::const_iterator &Pol,
             size_t deg,
             Integer &x) const
{

  if (deg == 1){
    for (size_t i=0;i<y.size();++i)
       _r.assign(y[i],(*Pol)[i]);
  }
  else{
    size_t deg_low, deg_high;
    deg_high = deg/2;
    deg_low  = deg - deg_high;

    Integer zero;
    _r.init(zero,0);

    Vector y1(y.size(),zero), y2(y.size(),zero);

    Integer x1=x, x2=x;

    PolEval(y1, Pol, deg_low, x1);

    std::vector<Vector>::const_iterator Pol_high= Pol+deg_low;
```

```
      PolEval(y2, Pol_high, deg_high, x2);

      for (size_t i=0;i< y.size();++i)
         _r.axpy(y[i],x1,y2[i],y1[i]);

      _r.mul(x,x1,x2);
    }
  }
}
```

Code A.9 – Implantation de la fonction `getRational` de la classe `RationalReconstruction`

```
  template<class Vector1>
  bool getRational(Vector1& num, Integer& den) const {

    linbox_check(num.size() == (size_t)_lcontainer.size());

    // prime
    Integer prime = _lcontainer.prime();

    // length of whole approximation
    size_t length=_lcontainer.length();

    // size of solution
    size_t size= _lcontainer.size();


    Integer zero;
    _r.init(zero,0);
    Vector zero_digit(_lcontainer.size(),zero);

    // store approximation as a polynomial
    std::vector<Vector>  digit_approximation(length,zero_digit);

    // store real approximation
    Vector real_approximation(size,zero);


    // store modulus (intially set to 1)
    Integer modulus;
    _r.init(modulus, 1);

    // denominator upper bound
    Integer denbound;
    _r.assign(denbound,_lcontainer.denbound());

    // numerator  upper bound
    Integer numbound;
    _r.assign(numbound,_lcontainer.numbound());

    // Compute all the approximation using liftingcontainer
    typename LiftingContainer::const_iterator iter = _lcontainer.begin();
    for (size_t i=0 ;
         iter != _lcontainer.end() && iter.next(digit_approximation[i]);++i) {
      _r.mulin(modulus,prime);
```

```
}


// problem occured during lifting
if (iter!= _lcontainer.end()){
  cout << "ERROR in lifting container?" << endl;
  return false;
}

// Evaluate padic lifting using divide an conquer evaluation
Integer xeval=prime;
typename std::vector<Vector>::const_iterator poly_digit;
poly_digit = digit_approximation.begin();
PolEval(real_approximation, poly_digit, length, xeval);


/*
 * Rational Reconstruction of each coefficient
 * according to a common denominator
 */

Integer common_den, common_den_mod_prod, bound,two,tmp;
_r.init(common_den,1);
_r.init(common_den_mod_prod,1);
_r.init(two,2);

Vector denominator(num.size());

int counter=0;
typename Vector::iterator   iter_approx = real_approximation.begin();
typename Vector1::iterator  iter_num    = num.begin();
typename Vector::iterator   iter_denom  = denominator.begin();

//numbound=denbound;

for (size_t i=0;
     iter_approx != real_approximation.end();
     ++iter_approx, ++ iter_num, ++iter_denom, ++i)
{
  _r.mulin( *iter_approx , common_den_mod_prod);
  _r.remin( *iter_approx , modulus);
  if (!_r.reconstructRational(*iter_num, *iter_denom,
                              *iter_approx, modulus,
                              numbound, denbound))
  {
    cout << "ERROR in reconstruction ?\n" << endl;
    return false;
  }

  _r.mulin(common_den, *iter_denom);
  if (i != size-1){
    if (! _r.isUnit(*iter_denom)) {
      counter++;
      _r.quoin(denbound , *iter_denom);
      _r.mul(bound, denbound,numbound);
      _r.mulin(bound,two);
      _r.div(tmp,modulus,prime);
      while(tmp > bound) {
        _r.assign(modulus,tmp);
```

```
            _r.div(tmp,modulus,prime);
          }
          _r.rem(tmp , *iter_denom , modulus);
          _r.remin(common_den_mod_prod , modulus);
          _r.mulin(common_den_mod_prod , tmp);
          _r.remin(common_den_mod_prod , modulus);
        }
      }

    }

    typename Vector1::reverse_iterator rev_iter_num   = num.rbegin();
    typename Vector::reverse_iterator  rev_iter_denom = denominator.rbegin();
    _r.init(tmp,1);
    for (; rev_iter_num != num.rend(); ++rev_iter_num, ++rev_iter_denom){
      _r.mulin(*rev_iter_num,tmp);
      _r.mulin(tmp, *rev_iter_denom);
    }

    den = common_den;
    return true;
}
```

A.3 Résolution de systèmes linéaires entiers singuliers avec l'algorithme de Dixon

Code A.10 – Implantation de l'algorithme `DixonSingularSolve`

```
template <class Ring, class Field, class RandomPrime>
template <class IMatrix, class Vector1, class Vector2>
SolverReturnStatus
RationalSolver<Ring,Field,RandomPrime,DixonTraits>::SolveSingular
(Vector1                &num,
 Integer                &den,
 const IMatrix            &A,
 const Vector2            &b,
 int                maxPrimes,
 const SolverLevel     level) const {

  int trials = 0;
  while (trials < maxPrimes){
    if (trials != 0) chooseNewPrime();
    trials++;

    typedef typename Field::Element Element;
    typedef typename Ring::Element Integer;
    typedef DixonLiftingContainer<Ring,
                                  Field,
                                  BlasMatrix<Integer>,
                                  BlasMatrix<Element> > LiftingContainer;

    // Checking size of system.
```

```
linbox_check(A.rowdim() == b.size());

// Definition of useful value.
LinBox::integer tmp;
Integer _rone,_rzero;
_R.init(_rone,1);
_R.init(_rzero,0);

// Definition of computation domains.
Field                        F(_prime);
BlasMatrixDomain<Ring>       BMDI(_R);
BlasMatrixDomain<Field>      BMDF(F);
BlasApply<Ring>              BAR(_R);
MatrixDomain<Ring>           MD(_R);
VectorDomain<Ring>           VDR(_R);

BlasMatrix<Integer>    A_check(A);
BlasMatrix<Element> *TAS_factors;
// Computation of *TAS_factors = [A^T| b^T]^T mod p.
TAS_factors = new BlasMatrix<Element>(A.coldim()+1, A.rowdim());
for (size_t i=0;i<A.rowdim();++i)
   for (size_t j=0;j<A.coldim();++j)
      F.init(TAS_factors->refEntry(j,i),_R.convert(tmp,A.getEntry(i,j)));
for (size_t i=0;i<A.rowdim();++i)
   F.init(TAS_factors->refEntry(A.coldim(),i),_R.convert(tmp,b[i]));

// LQUP factorisation of *TAS_factors.
LQUPMatrix<Field>* TAS_LQUP = new LQUPMatrix<Field>(F, *TAS_factors);

// Get rank of [A|b] mod p from LQUP.
size_t TAS_rank = TAS_LQUP->getrank();

// Compute row index of inconsistency
BlasPermutation TAS_P  = TAS_LQUP->getP();
BlasPermutation TAS_Qt = TAS_LQUP->getQ();
std::vector<size_t> srcRow(A.rowdim()), srcCol(A.coldim()+1);
std::vector<size_t>::iterator sri = srcRow.begin(), sci = srcCol.begin();
for (size_t i=0; i<A.rowdim(); i++, sri++) *sri = i;
for (size_t i=0; i<A.coldim()+1; i++, sci++) *sci = i;
indexDomain iDom;
BlasMatrixDomain<indexDomain> BMDs(iDom);
BMDs.mulin_right(TAS_Qt, srcCol);
BMDs.mulin_right(TAS_P, srcRow);

bool appearsInconsistent = (srcCol[TAS_rank-1] == A.coldim());
size_t rank = TAS_rank - (appearsInconsistent ? 1 : 0);

// Prevent from null matrix
if (rank == 0) {
  delete TAS_LQUP;
  delete TAS_factors;
  bool aEmpty = true;
  if (level >= SL_LASVEGAS) {
    typename BlasMatrix<Integer>::RawIterator iter = A_check.rawBegin();
    for (; aEmpty && iter != A_check.rawEnd(); ++iter)
      aEmpty &= _R.isZero(*iter);
  }
  if (aEmpty) {
    for (size_t i=0; i<b.size(); i++)
```

```
        if (!_R.areEqual(b[i], _rzero)) {
          if (level >= SL_CERTIFIED) {
            lastCertificate.clearAndResize(b.size());
            _R.assign(lastCertificate.numer[i], _rone);
          }
          return SS_INCONSISTENT;
        }
      _R. assign (den, _rone);
      for (typename Vector1::iterator p = num. begin(); p != num. end(); ++ p)
        _R. assign (*p, _rzero);
      return SS_OK;
    }
    // so A was empty mod p but not over Z.
    continue; //try new prime
  }

  // Stop if the system seems to be inconsistent in a Monte Carlo fashion.
  if (appearsInconsistent && level <= SL_MONTECARLO)
    return SS_INCONSISTENT;

  // Compute the inverse mod p of the maximal leading minor.
  BlasMatrix<Element>* Atp_minor_inv = NULL;

  // take advantage of the (LQUP)t factorization to compute
  // an inverse to the leading minor of (TAS_P . (A|b) . TAS_Q)
  Atp_minor_inv = new BlasMatrix<Element>(rank, rank);
  typename BlasMatrix<Element>::RawIterator iter=Atp_minor_inv->rawBegin();
  for (; iter != Atp_minor_inv->rawEnd(); iter++)
    F.init(*iter, 0);
  FFLAPACK::LQUPtoInverseOfFullRankMinor(F, rank,
                                         TAS_factors->getPointer(),
                                         A.rowdim(),
                                         TAS_Qt.getPointer(),
                                         Atp_minor_inv->getPointer(),
                                         rank);

  delete TAS_LQUP;
  delete TAS_factors;

  if (appearsInconsistent) {
    // get the corresponding row of A from the inconsistent index.
    std::vector<Integer> zt(rank);
    for (size_t i=0; i<rank; i++)
       _R.assign(zt[i], A.getEntry(srcRow[rank], srcCol[i]));

    // Get the transposed of the maximal leading minor over the integers.
    BlasMatrix<Integer> At_minor(rank, rank);
    for (size_t i=0; i<rank; i++)
       for (size_t j=0; j<rank; j++)
         _R.assign(At_minor.refEntry(j,i), A.getEntry(srcRow[i],srcCol[j]));

    // Compute the certficate of inconsistency using DixonLiftingContainer.
    BlasMatrix<Integer>  BBAt_minor( At_minor);
    BlasMatrix<Element>  BBAtp_minor_inv( *Atp_minor_inv);
    LiftingContainer lc(_R, F, BBAt_minor, BBAtp_minor_inv, zt, _prime);
    RationalReconstruction<LiftingContainer > re(lc);
    Vector1 short_num(rank); Integer short_den;
    if (!re.getRational(short_num, short_den,0))
      return SS_FAILED;
```

```
    delete Atp_minor_inv;
    VectorFraction<Ring> cert(_R, short_num. size());
    cert. numer = short_num;
    cert. denom = short_den;
    cert.numer.resize(b.size());
    _R.subin(cert.numer[rank], cert.denom);
    _R.init(cert.denom, 1);
    BMDI.mulin_left(cert.numer, TAS_P);


    // Checking certificate
    bool certifies = true;
    std::vector<Integer> certnumer_A(A.coldim());
    BAR.applyVTrans(certnumer_A, A_check, cert.numer);
    typename std::vector<Integer>::iterator cai = certnumer_A.begin();
    for (size_t i=0; certifies && i<A.coldim(); i++, cai++)
        certifies &= _R.isZero(*cai);

    // Certificate is valide stop computation.
    if (certifies) {
      if (level == SL_CERTIFIED) lastCertificate.copy(cert);
        return SS_INCONSISTENT;
    }
    // Cerificate is not valide, choose new prime.
    continue;
  }

  // System is known as consistent mod p.
  // Try to solve it using DixonLiftingContainer.

  BlasMatrix<Integer> A_minor(rank, rank);
  BlasMatrix<Element>        *Ap_minor_inv;

  Element _rtmp;
  Ap_minor_inv = Atp_minor_inv;
  for (size_t i=0; i<rank; i++)
  for (size_t j=0; j<i; j++) {
    Ap_minor_inv->getEntry(_rtmp, i, j);
    Ap_minor_inv->setEntry(i, j, Ap_minor_inv->refEntry(j, i));
    Ap_minor_inv->setEntry(j, i, _rtmp);
  }

  // Permute original entries into A_minor.
  for (size_t i=0; i<rank; i++)
    for (size_t j=0; j<rank; j++)
      _R.assign(A_minor.refEntry(i,j), A_check.getEntry(srcRow[i],srcCol[j]));

  // Compute newb = (TAS_P.b)[0..(rank-1)].
  std::vector<Integer> newb(b);
  BMDI.mulin_right(TAS_P, newb);
  newb.resize(rank);

  // Compute the solution of the non singular system.
  BlasMatrix<Integer>     BBA_minor(A_minor);
  BlasMatrix<Element> BBA_inv(*Ap_minor_inv);
  LiftingContainer lc(_R, F, BBA_minor, BBA_inv, newb, _prime);
  RationalReconstruction<LiftingContainer > re(lc);

  Vector1 short_num(rank); Integer short_den;
```

```
      if (!re.getRational(short_num, short_den,0))
         return SS_FAILED;

      VectorFraction<Ring> answer_to_vf(_R, short_num. size());
      answer_to_vf. numer = short_num;
      answer_to_vf. denom = short_den;

      // short_answer = TAS_Q * short_answer.
      answer_to_vf.numer.resize(A.coldim()+1,_rzero);
      BMDI.mulin_left(answer_to_vf.numer, TAS_Qt);
      answer_to_vf.numer.resize(A.coldim());

      // Check the solution.
      if (level >= SL_LASVEGAS) {
        std::vector<Integer> A_times_xnumer(b.size());
        BAR.applyV(A_times_xnumer, A_check, answer_to_vf.numer);
        Integer tmpi;

        typename Vector2::const_iterator ib = b.begin();
        typename std::vector<Integer>::iterator iAx = A_times_xnumer.begin();
        int thisrow = 0;
        bool needNewPrime = false;

        for (; !needNewPrime && ib != b.end(); iAx++, ib++, thisrow++)
        if (!_R.areEqual(_R.mul(tmpi, *ib, answer_to_vf.denom), *iAx)) {
          needNewPrime = true;
        }

        // Solution is not a solution, choose new prime.
        if (needNewPrime) {
          delete Ap_minor_inv;
          continue;
        }
      }

      // return the right solution.
      num = answer_to_vf. numer;
      den = answer_to_vf. denom;

      delete Ap_minor_inv;
      delete B;

      // Computation succeeded.
      return SS_OK;
    }
    // All prime were bad, computation failed.
    return SS_FAILED;
  }
```

A.4 Produits matrice-vecteur et matrice-matrice en représentation q-adique

Code A.11 – Domaine de calcul `MatrixApplyDomain`

```
template <class Domain, class IMatrix>
class MatrixApplyDomain {

  public:
    typedef typename Domain::Element      Element;
    typedef std::vector<Element>           Vector;

    MatrixApplyDomain(const Domain& D, const IMatrix &M) : _D(D), _M(M) {}
    void setup(LinBox::integer prime){}
    Vector& applyV(Vector& y, Vector& x) const
      { return _M.apply(y,x);}
    Vector& applyVTrans(Vector& y, Vector& x) const
      {return _M.applyTranspose(y,x);}

  private:
    Domain          _D;
    const IMatrix  &_M;
};

template <class Domain, class IMatrix>
class BlasMatrixApplyDomain {

  protected:
    Domain                          _D;
    const IMatrix                  &_M;
    MatrixDomain<Domain>           _MD;
    size_t                          _m;
    size_t                          _n;
    bool                     use_chunks;
    bool                        use_neg;
    size_t                   chunk_size;
    size_t                   num_chunks;
    double *                     chunks;

  public:
    typedef typename Domain::Element      Element;
    typedef std::vector<Element>           Vector;
    typedef IMatrix                         Matrix;

    BlasMatrixApplyDomain(const Domain& D, const IMatrix &M)
      : _D(D), _M(M), _MD(D), _m(M.rowdim()), _n(M.coldim()) {}

    ~BlasMatrixApplyDomain () {
      if (use_chunks) delete[] chunks;
    }

    // Compute the q-adic representation of M with q=2^16.
    void setup(LinBox::integer prime){
      // Compute the maximum size of chunks
      LinBox::integer maxChunkVal = 1;
      maxChunkVal <<= 53;
```

```
maxChunkVal /= (prime-1) * _n;
chunk_size = -1;
while (maxChunkVal > 0) {
  maxChunkVal /= 2;
  chunk_size++;
}
use_chunks = (chunk_size >= 16);

if (use_chunks){
  chunk_size = 16;

  // Compute the magnitude in bit of the matrix M.
  // Check if we need negative representation.
  LinBox::integer tmp=0;
  size_t maxBitSize = 0;
  use_neg = false;
  typename Matrix::ConstRawIterator it = _M.rawBegin();
  for (size_t i=0; i<_m*_n; i++, ++it) {
    _D.convert(tmp, *it);
    maxBitSize = max(maxBitSize, tmp.bitsize());
    use_neg |= (tmp < 0);
  }

  // Compute the number of chunk.
  num_chunks=(maxBitSize/chunk_size);
  num_chunks+=(((maxBitSize % chunk_size) > 0)? 1:0);
  if (num_chunks ==1)
    use_neg= false;
  if (use_neg)
    num_chunks++; //the leading chunk will be negative.

  int n2 = _m*_n;
  chunks = new double[n2*num_chunks];
  memset(chunks, 0, sizeof(double)*_m*_n*num_chunks);
  it = _M.rawBegin();

  if (num_chunks ==1) // Do not need a q-adic representation.
    for (int i=0; i<n2; i++, ++it) {
      _D.convert(*(chunks+i), *it);
    }
  else  // Compute q-adic representation.
    for (int i=0; i<n2; i++, ++it) {
      integer tmp;
      double* pdbl = chunks + i;
      _D.convert(tmp, *it);
      if (tmp ==0)
        *pdbl=0;
      else
        if (tmp > 0) {
          size_t tmpsize    = tmp.size();
          size_t tmpbitsize = tmp.bitsize();
          size_t j;
          for (j=0; j<tmpsize-1; j++) {
            *pdbl = tmp[j] & 0xFFFF;
            *(pdbl+n2) = tmp[j] >> 16;
            pdbl += 2*n2;
          }
          if ((tmpbitsize - j*32) > 16 ) {
            *pdbl = tmp[tmpsize-1] & 0xFFFF;
```

```
            *(pdbl+n2) = tmp[tmpsize-1] >> 16;
          }
          else {
            *pdbl = tmp[tmpsize-1] & 0xFFFF;
          }
        }
        else {
          ++tmp;
          size_t tmpsize    = tmp.size();
          size_t tmpbitsize = tmp.bitsize();
          size_t j;

          for (j=0; j<tmpsize-1; j++) {
            *pdbl = 0xFFFF ^ (tmp[j] & 0xFFFF);
            *(pdbl+n2) = 0xFFFF ^ (tmp[j] >> 16);
            pdbl += 2*n2;
          }
          if ((tmpbitsize -j*32) > 16) {
            *pdbl = 0xFFFF ^ (tmp[tmpsize-1] & 0xFFFF);
            *(pdbl+n2) = 0xFFFF ^ (tmp[tmpsize-1] >> 16);
            pdbl += 2*n2;
            j=tmpsize<<1;
          }
          else {
            *pdbl = 0xFFFF ^ (tmp[tmpsize-1] & 0xFFFF);
            pdbl += n2;
            j = (tmpsize<<1) -1;
          }
          for (; j<num_chunks-1; j++, pdbl += n2)
            *pdbl = 0xFFFF;
          *pdbl = 1; //set the leading negative chunk for this entry.
        }
      }
      use_neg = !(!use_neg);
    }
  }

  // Compute y=Mx.
  // using BLAS matrix-vector product and GMP integer construction.
  Vector& applyV(Vector& y, Vector& x) const {

    linbox_check( _n == x.size());
    linbox_check( _m == y.size());

    if (!use_chunks){ // q-adic is not allowed
      _MD.vectorMul (y, _M, x);
    }
    else{
      // Convert x in a double representation.
      double* dx = new double[_n];
      for (size_t i=0; i<_n; i++) {
        _D.convert(dx[i], x[i]);
      }
      if (num_chunks == 1) {
        // Use directly BLAS dgemv function.
        double *ctd = new double[_m];
        cblas_dgemv(CblasRowMajor, CblasNoTrans, _m, _n,
                    1, chunks, _n, dx, 1, 0, ctd, 1);
        for (size_t i=0;i<_n;++i)
```

```
        _D.init(y[i],ctd[i]);
        delete[] ctd;
      }
      else {
        //rc: number of different series.
        int rc = (52 / chunk_size) + 1;

        //rclen: number of bytes in each of these series.
        int rclen = num_chunks*2 + 5;

        unsigned char* combined = new unsigned char[rc*_n*rclen];
        memset(combined, 0, rc*_n*rclen);

        //compute a product (chunk times x) for each chunk
        double* ctd = new double[_m];
        for (size_t i=0; i<num_chunks; i++) {
          cblas_dgemv(CblasRowMajor, CblasNoTrans, _m, _n,
                      1, chunks+(_m*_n*i), _n, dx, 1, 0, ctd, 1);
          if (!use_neg || i<num_chunks-1)
          for (size_t j=0; j<_n; j++) {
            // up to 53 bits will be ored-in, to be summed later.
            unsigned char* bitDest = combined;
            bitDest += rclen*((i % rc)*_n+j);
            long long mask = static_cast<long long>(ctd[j]);
            bitDest += 2*i;
            *((long long*) bitDest) |= mask;
          }
        }
        delete[] dx;

        for (size_t i=0; i<_n; i++) {
          LinBox::integer result, tmp;
          // Subtract leading term if negative.
          if (use_neg) {
            result = -ctd[i];
            result <<= (num_chunks-1)*16;
          }
          else
            result = 0;

          // Construct integer from byte arrays.
          for (int j=0; j<rc; j++) {
            unsigned char* thispos = combined + rclen*(j*_n+i);
            importWords(tmp, rclen, -1, 1, 0, 0, thispos);
            result += tmp;
          }
          _D.init(y[i], result);
        }
        delete[] combined;
        delete[] ctd;
      }
    }
    return y;
  }

  // Compute Y=MX.
  // using BLAS matrix-matrix product and GMP integer construction.
  IMatrix& applyM (IMatrix &Y, const IMatrix &X) const {
```

```
linbox_check( _n == X.rowdim());
linbox_check( _m == Y.rowdim());
linbox_check( Y.coldim() == X.coldim());

if (!use_chunks){
  _MD.mul (Y, _M, X);
}
else{
  size_t _k= X.coldim();
  double* dX = new double[_n*_k];
  for (size_t i=0; i<_n; i++)
    for(size_t j=0;j<_k;++j)
      _D.convert(dX[i*_k+j], X.getEntry(i,j));

  if (num_chunks == 1) {
    double *ctd = new double[_m*_k];
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                _m,_k,_n, 1,chunks,_n, dX, _k, 0, ctd, _k);

    for (size_t i=0;i<_m;++i)
      for (size_t j=0;j<_k;++j)
        _D.init(Y.refEntry(i,j),ctd[i*_k+j]);
    delete[] ctd;
    delete[] dX;
  }
  else {
    int rc = (52 / chunk_size) + 1;
    int rclen = num_chunks*2 + 5;
    unsigned char* combined = new unsigned char[rc*_m*_k*rclen];
    memset(combined, 0, rc*_m*_k*rclen);

    //compute a product (chunk times x) for each chunk
    double* ctd = new double[_m*_k];
    for (size_t i=0; i<num_chunks; i++) {
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                  _m,_k,_n, 1,chunks+(_m*_n*i),_n,dX,_k,0,ctd,_k);
      if (!use_neg || i<num_chunks-1)
        for (size_t j=0; j<_m*_k; j++) {
          unsigned char* bitDest = combined;
          bitDest += rclen*((i % rc)*_m*_k+j);
          long long mask = static_cast<long long>(ctd[j]);
          bitDest += 2*i;
          *((long long*) bitDest) |= mask;
        }
    }
    delete[] dX;

    for (size_t i=0; i<_m*_k; i++) {
      LinBox::integer result, tmp;
      if (use_neg) {
        result = -ctd[i];
        result <<= (num_chunks-1)*16;
      }
      else
        result = 0;

    for (int j=0; j<rc; j++) {
      unsigned char* thispos = combined + rclen*(j*_m*_k+i);
      importWords(tmp, rclen, -1, 1, 0, 0, thispos);
```

```
            result += tmp;
          }
          _D.init(*(Y.getWritePointer()+i), result);
        }
        delete[] combined;
        delete[] ctd;
      }
    }
    return Y;
  }
};

// Partial specialization of MatrixApplyDomain for BlasMatrix
template<>
template <class Domain>
class MatrixApplyDomain<Domain,
                        BlasMatrix<typename Domain::Element> >
  : public BlasMatrixApplyDomain<Domain,
                                 BlasMatrix<typename Domain::Element> >
{
  public:
    MatrixApplyDomain (const Domain &D,
                       const  BlasMatrix<typename Domain::Element> &M)
      : BlasMatrixApplyDomain<Domain,
                              BlasMatrix<typename Domain::Element> >(D,M){}
};

// Partial specialization of MatrixApplyDomain for DenseMatrix
template<>
template <class Domain>
class MatrixApplyDomain<Domain,
                        DenseMatrix<Domain> >
  : public BlasMatrixApplyDomain<Domain, DenseMatrix<Domain> >
{
  public:
    MatrixApplyDomain (const Domain &D, const DenseMatrix<Domain> &M)
      : BlasMatrixApplyDomain<Domain, DenseMatrix<Domain> > (D,M) {}
};
```

Table des figures

1.1 Physionomie du projet LinBox 8
1.2 Exemple d'héritage : modèle de vecteur à deux niveaux 11
1.3 Mécanisme des fonctions virtuelles 16
1.4 Hierarchie dans la bibliothèque LinBox 24

2.1 Spécifications du modèle de base des corps finis. 30
2.2 Structure de l'archétype des corps finis. 32
2.3 Code pour une interface abstraite compilable. 33
2.4 Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`). 41
2.5 Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`). 42
2.6 Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathtt{GF}(3^7)$ (`P3-1Ghz`). 43
2.7 Surcoût relatif des différents niveaux de l'archétype en fonction des implantations concrètes de corps finis sur $\mathtt{GF}(3^7)$ (`IA64-733Mhz`). 44
2.8 Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`P3-1Ghz`). 57
2.9 Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`IA64-733Mhz`). 57
2.10 Performances (en Mops) pour le corps premier $\mathbb{Z}_{32749}$ (`Xeon-2.66Ghz`). 58

3.1 Produit scalaire avec réduction modulaire paresseuse (`P3-933Mhz`) 75
3.2 Principe de la factorisation `LSP` 80
3.3 Principe de la factorisation `LUdivine` 80
3.4 Principe de la factorisation `LQUP` 81
3.5 Interface LinBox pour l'utilisation des BLAS 100

4.1 Interface LinBox pour la résolution des systèmes linéaires entiers 125

4.2 Principe de l'itérateur de développements p-adiques 129
4.3 Interface des conteneurs de développements p-adiques 129
4.4 Amélioration pratique de l'algorithme `FastReconstructSolution` par rapport à une reconstruction classique 134

Liste des tableaux

1.1 Contrôle d'accès des classes de base via l'héritage 12

2.1 Performances (secondes) de l'archétype des corps finis sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`). . . . 41
2.2 Performances (secondes) de l'archétype des corps finis sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`). . 42
2.3 Performances (secondes) de l'archétype des corps finis sur $\mathtt{GF}(3^7)$ (`P3-1Ghz`). . . . 43
2.4 Performances (secondes) de l'archétype des corps finis sur $\mathtt{GF}(3^7)$ (`IA64-733Mhz`). 44
2.5 Surcoût des wrappers sur $\mathbb{Z}_{1009}$ (`P3-1Ghz`). 54
2.6 Surcoût des wrappers sur $\mathbb{Z}_{1009}$ (`IA64-733Mhz`). 54
2.7 Performance (en Kops) pour un corps premier de taille 128 bits (`P3-1Ghz`). . . . 58
2.8 Performance (en Kops) pour un corps premier de taille 128 bits (`IA64-733Mhz`). . 58
2.9 Performance (en Kops) des corps premiers de taille 128 bits (`Xeon-2.66Ghz`). . . 58
2.10 Surcoût des wrappers des extensions algébriques (`P3-1Ghz`). 63
2.11 Surcoût des wrappers des extensions algébriques (`IA64-733Mhz`). 63
2.12 Performance (en Kops) pour l'extension algébrique $\mathtt{GF}(2^{512})$ (`P3-1Ghz`). 64
2.13 Performance (en Kops) pour l'extension algébrique $\mathtt{GF}(2^{512})$ (`Xeon-2.66Ghz`). . . 64
2.14 Performance pour l'extension algébrique $\mathtt{GF}(17^{100})$ (`P3-1Ghz`). 65
2.15 Performance pour l'extension algébrique $\mathtt{GF}(17^{100})$ (`Xeon-2.66Ghz`). 65

3.1 Performances (Mops) des routines `Trsm` pour `Modular<double>` (`P4-2.4Ghz`) . . 77
3.2 Performances (Mops) des routines `Trsm` pour `Givaro-ZpZ` (`P4-2.4Ghz`) 77
3.3 Performances (secondes) `LSP`, `LUdivine`, `LQUP` pour $\mathbb{Z}_{101}$ (`P4-2.4Ghz`) 82
3.4 Ressource (Mo) `LSP`, `LUdivine`, `LQUP` pour $\mathbb{Z}_{101}$ 82
3.5 Produit matriciel comparé à la factorisation LQUP sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`) . . 83
3.6 Produit matriciel par rapport à l'inverse sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`) 84
3.7 Produit matriciel par rapport à l'inverse sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`) 85
3.8 Produit matriciel par rapport à la base du noyau sur $\mathbb{Z}_{101}$ (`Xeon-2.66Ghz`) . . . 85

4.1 Comparaisons des méthodes d'évaluation de développement p-adique 132
4.2 Comparaisons des implantations de l'algorithme de Dixon (`Xeon-2.66Ghz`) . . . 148
4.3 Performances de la fonction `solveSingular` (`Xeon-2.66Ghz`) 149
4.4 Performances de la fonction `findRandomSolution` (`Xeon-2.66Ghz`) 149
4.5 Comparaisons des temps de calculs de solutions diophantiennes (`Xeon-2.66Ghz`) 158
4.6 Solutions diophantiennes en fonction de la taille du noyau pour des systèmes linéaires à coefficients sur 100 bits (`Xeon-2.66Ghz`) 159

Liste des codes

1.1 Exemple de classe 10
1.2 Exemple d'héritage 12
1.3 Exemple de polymorphisme statique 13
1.4 Exemple d'utilisation des *traits* 17
2.1 Archétype des corps finis 36
2.2 Archétype des éléments 37
2.3 Archétype des générateurs aléatoires 38
3.1 Interface de résolution de systèmes triangulaires des FFLAS (`ftrsm`) 92
3.2 Résolution de système linéaire triangulaire à partir des BLAS 93
3.3 Exemple d'utilisation de la routine `ftrsm` des FFLAS 95
3.4 Wrapper MAPLE pour les routines des paquetages FFLAS-FFPACK 96
3.5 Module MAPLE pour l'utilisation des routines FFLAS-FFPACK 99
3.6 Modèle de matrice LinBox pour les routines BLAS 101
3.7 Spécialisation de la classe `BlasMatrix` aux matrices triangulaires 102
3.8 Spécialisation de la classe `BlasMatrix` aux matrices transposées 103
3.9 Modèle de permutation LinBox pour les routines FFLAS-FFPACK 104
3.10 Interface LinBox pour la factorisation de matrices denses sur un corps fini 105
3.11 Implantation des systèmes linéaires au travers de la classe `LQUPMatrix` 107
3.12 Interface LinBox pour les routines FFLAS-FFPACK 109
3.13 Spécialisations pour la multiplication de matrices denses `BlasMatrix` 112
3.14 Spécialisation pour la multiplication de matrices triangulaires `BlasMatrix` 113

3.15 Spécialisations pour appliquer une permutation à une matrice `BlasMatrix` 114
3.16 Spécialisations pour la résolution de systèmes linéaires 116
4.1 Domaine de résolution de systèmes linéaires entiers 126
4.2 Interface de reconstruction rationnelle 130
4.3 Spécialisation de la classe LiftingContainerBase à l'algorithme de Dixon 136
4.4 Résolution de systèmes linéaires non singuliers par l'algorithme de Dixon 138
4.5 Utilisation de LinBox pour la résolution de systèmes linéaires diophantiens . . . 156
A.6 Interface LinBox des conteneurs de développements p-adiques 173
A.7 Interface LinBox des itérateurs de développements p-adiques 174
A.8 Évaluation des développements p-adiques 175
A.9 Implantation de la fonction `getRational` de la classe `RationalReconstruction` 176
A.10 Implantation de l'algorithme `DixonSingularSolve` 178
A.11 Domaine de calcul `MatrixApplyDomain` 183

Résumé : L'algèbre linéaire numérique a connu depuis quelques décennies des développements intensifs autant au niveau mathématique qu'informatique qui ont permis d'aboutir à de véritable standard logiciel comme BLAS ou LAPACK. Dans le cadre du calcul exact ou formel, la situation n'est pas aussi avancée, en particulier à cause de la diversité des problématiques et de la jeunesse des progrès théoriques. Cette thèse s'inscrit dans une tendance récente qui vise à fédérer des codes performants provenant de bibliothèques spécialisées au sein d'une unique plateforme de calcul. En particulier, l'émergence de bibliothèques robustes et portables comme GMP ou NTL pour le calcul exact s'avére être un réel atout pour le développement d'applications en algèbre linéaire exacte. Dans cette thèse, nous étudions la faisabilité et la pertinence de la réutilisation de codes spécialisés pour développer une bibliothèque d'algèbre linéaire exacte performante, à savoir la bibliothèque LinBox. Nous nous appuyons sur les mécanismes C++ de programmation générique (classes abtraites, classes templates) pour fournir une abstraction des composantes mathématiques et ainsi permettre le *plugin* de composants externes. Notre objectif est alors de concevoir et de valider des *boîtes à outils génériques* haut niveau dans LinBox pour l'implantation d'algorithmes en algèbre linéaire exacte. En particulier, nous proposons des routines de calcul hybride "exact/numérique" pour des matrices denses sur un corps finis permettant d'approcher les performances obtenues par des bibliothèques numériques comme LAPACK. À un plus haut niveau, ces routines nous permettent de valider la réutilisation de codes spécifiques sur un problème classique du calcul formel : la résolution de *systèmes linéaires diophantiens*. La bibliothèque LinBox est disponible à www.linalg.org.

Mots clés : algèbre linéaire exacte, corps finis, analyse algorithmique, développement algorithmique, bibliothèques numériques, interfaces, programmation générique, factorisations de matrices, systèmes linéaires diophantiens.

Abstract: For a few decades, numerical linear algebra has seen intensive developments in both mathematical and computer science theory which have led to genuine standard software like BLAS or LAPACK. In computer algebra the situation has not advanced as much, in particular because of the diversity of the problems and because of much of the theoretical progress have been done recently. This thesis falls into a recent class of work which aims at uniforming high-performance codes from many specialized libraries into a single platform of computation. In particular, the emergence of robust and portable libraries like GMP or NTL for exact computation has turned out to be a real asset for the development of applications in exact linear algebra. In this thesis, we study the feasibility and the relevance of the re-use of specialized codes to develop a high performance exact linear algebra library, namely the LinBox library. We use the generic programming mechanisms of C++ (abstract class, template class) to provide an abstraction of the mathematical objects and thus to allow the *plugin* of external components. Our objective is then to design and validate, in LinBox, high level *generic toolboxes* for the implementation of algorithms in exact linear algebra. In particular, we propose "exact/numeric" hybrid computation routines for dense matrices over finite fields which nearly match with the performance obtained by numerical libraries like LAPACK. On a higher level, we reuse these hybrid routines to solve very efficiently a classical problem of computer algebra : solving *diophantine linear systems*. Hence, this allowed us to validate the principle of code reuse in LinBox library and more generally in computer algebra. The LinBox library is available at www.linalg.org.

Keywords: exact linear algebra, finite field, algorithm analysis, algorithm design, numerical libraries, interfaces, generic programming, matrix factorization, diophantine linear system.