ÉCOLE POLYTECHNIQUE

Thèse présentée pour obtenir le grade de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

spécialité :

INFORMATIQUE

par

Florent CHABAUD

Titre :

RECHERCHE DE PERFORMANCE DANS L'ALGORITHMIQUE DES CORPS FINIS. APPLICATIONS À LA CRYPTOGRAPHIE.

Soutenue le 22 octobre 1996 devant le jury composé de :

M.	**Jacques STERN**	Directeur
MM.	**Paul CAMION**	Rapporteurs
	Jean VUILLEMIN	
Mme.	**Pascale CHARPIN**	Examinateurs
MM.	**Antoine JOUX**	
	François MORAIN	

Thèse présentée pour obtenir le grade de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

spécialité :

INFORMATIQUE

par

Florent CHABAUD

Titre :

RECHERCHE DE PERFORMANCE DANS L'ALGORITHMIQUE DES CORPS FINIS. APPLICATIONS À LA CRYPTOGRAPHIE.

Soutenue le 22 octobre 1996 devant le jury composé de :

M.	**Jacques STERN**	Directeur
MM.	**Paul CAMION**	Rapporteurs
	Jean VUILLEMIN	
Mme.	**Pascale CHARPIN**	Examinateurs
MM.	**Antoine JOUX**	
	François MORAIN	

Recherches effectuées au Laboratoire d'Informatique de l'École Normale Supérieure
URA 1327 du CNRS
45, rue d'Ulm, 75230 Paris Cedex 05

Remerciements

Je tiens tout d'abord à remercier tous ceux que j'ai pu omettre dans ce qui suit.

Cependant je ne puis oublier Jean Vuillemin qui a bien voulu effectuer le travail difficile de rapporteur et dont les remarques ont fortement contribué à rendre cette thèse plus lisible.

Je tiens de même à remercier Paul Camion dont l'aide a été précieuse, en particulier pour la rédaction du chapitre I.4. Les améliorations en cours de la librairie Zen *doivent beaucoup aux remarques et références bibliographiques qu'il m'a communiquées à la lecture de la première version de ce chapitre.*

Merci aussi à toute son équipe du projet CODES de l'INRIA pour leur accueil chaleureux lors de mes visites, et tout d'abord à Pascale Charpin qui me fait l'honneur de participer à mon mon jury de thèse, et dont les réponses à mes nombreuses questions ont toujours été pertinentes.

Je ne saurais oublier Anne Canteaut pour le travail fructueux que nous avons réalisé ensemble, ni Nicolas Sendrier pour ses explications toujours pédagogiques. Ma gratitude va aussi à Daniel Augot, entre autres pour son aide précieuse avec Axiom dans l'exemple comparatif de l'annexe A.2.

Mais les membres de l'INRIA n'ont pas été les seuls à me supporter... Le LIX m'a lui aussi souvent vu, et en particulier Reynald Lercier, à qui Zen *doit beaucoup, et dont je loue au passage le travail de relecture de cette thèse. Les discussions parfois difficiles que nous avons pu avoir ensemble, ainsi qu'avec François Morain, ont toujours abouti à une amélioration du résultat. Je ne peux donc que les remercier tous deux. Ma reconnaissance va aussi à Jean-Marc Steyaert et Michel Weinfeld qui ont aussi beaucoup contribué à cette thèse en me permettant de travailler quasi-quotidiennement au LIX. Merci aussi à Evelyne Rayssac qui gère magistralement le secrétariat du laboratoire, pour son aide dans les divers problèmes administratifs que j'ai pu rencontrer à cette occasion.*

Je remercie aussi la DGA de m'avoir permis d'effectuer ces travaux de recherche. Merci en particulier à Alain Quenzer et à Alain Lanusse pour leur gestion des optionnaires Recherche. Merci aussi à la direction du CELAR et à François Debout d'avoir accepté de ne pas me voir physiquement en poste parmi eux. Enfin merci à toute l'équipe de Philippe Villoing pour leur accueil depuis que je les ai rejoints, en particulier à François Daudé, que j'ai connu lors d'un stage effectué il y a maintenant quatre ans. Je suis aussi redevable à Jacqueline Demarthon et Elisabeth Dorion pour leur gestion efficace de ce cas particulier d'un membre du CELAR parisien. Je n'oublie pas Michel Minoux qui m'a assisté pendant ces trois ans de thèse en tant que parrain scientifique à la DGA et qui aurait participé à mon jury si ses contraintes d'emploi du temps ne l'avaient empêché. Je n'oublie pas non plus Antoine Joux, qui semble me précéder depuis son départ du LIENS, puisque nous nous retrouvons ensemble au CELAR. Merci d'avoir été mon parrain DGA et de participer aujourd'hui à mon jury.

Je dois aussi beaucoup à un autre Ingénieur de l'Armement passé par le LIENS, Jean-Marc Couveignes. J'espère que nos collaborations à venir seront encore fructueuses. Il doit paraître clair désormais que le LIENS a beaucoup contribué à me faire aimer la recherche. Cela est certainement dû à l'ambiance amicale et chaleureuse qui y règne. L'équipe du GRECC est à ce titre particulièrement agréable, et j'y ai passé de très bons moments. Merci donc par ordre alphabétique à Philippe Béguin, avec qui j'ai pour la première fois discuté du système de Chen, merci à Jean-Bernard Fischer dont l'accueil à Rennes m'a été très utile, à Louis Granboulan pour toutes ses astuces de maître ès C ès html *qu'il est, à Philippe Hoogvorst qui m'a en particulier conseillé l'idée de la section I.2.5, à David Pointcheval pour tous les travaux effectués ensemble sur divers sujets, à Serge Vaudenay pour la cryptanalyse linéaire que nous avions effectuée et qui avait donné [5-CV94].*

De nombreuses personnes ont aussi contribué à divers titre à cette thèse. Un grand merci à toute l'équipe du SPI et en particulier à Jacques Beigbeder qui gère de main de maître le réseau de l'ENS, même quand je

lance des process un peu partout ! Merci à Brigitte Van Elsen pour tout ce qu'elle apporte au DMI, merci à Sabine Glasson, Annie Iapteff, Fabienne Meunier et Nicole Mourgues pour toutes les fois où je les ai ennuyées avec mes problèmes administratifs et où elles m'ont répondu avec le sourire. Je remercie aussi Claus Schnorr puisque c'est lui qui a attiré mon attention sur le système de Chen lors d'un séminaire organisé par ses soins à Riezlern. Merci à Sami Harari du GECT, et Pascal Véron pour de nombreuses discussions intéressantes.

Je tiens aussi à remercier Gilles Brassard, Claude Crépeau et David Naccache. Merci à Olivier Perret qui m'a aidé à utiliser en partie les disponibilités du réseau de l'ENSTA. Merci à Antonella Cresti pour ses tiramisus (étaient-ils bons Serge ?). Un grand merci à Patrick Grelier pour son pénible et malgré tout efficace travail de relecture, ainsi qu'à Thierry Saurin et Fabrice Lefebvre du LIX.

Merci à mes parents.

Enfin, last but not least*, je me dois d'exprimer toute ma gratitude à Jacques Stern, qui par son aide constante a suscité en moi un intérêt profond pour la recherche. J'espère que ses nouvelles responsabilités de directeur du LIENS ne l'empêcheront pas d'avoir avec moi d'autres conversations, car je sais d'avance tout l'intérêt que l'on peut en tirer.*

Introduction

"Les occasions qui produisent les grands changements sont différentes, mais les causes sont toujours les mêmes."

Charles de Secondat de Montesquieu
De la Grandeur des Romains et de leur Décadence

Dans un premier temps, nous présentons un nouvel outil de programmation, la librairie Zen, qui a été développé conjointement par l'auteur et Reynald Lercier, du Laboratoire d'Informatique de l'École Polytechnique. Cette librairie permet la manipulation en langage C d'objets mathématiques définis sur les corps finis, voire sur des ensembles moins structurés.

Il est donc naturel de commencer par rappeler quelques notions d'algèbre (chapitre I.1), qui nous sont utiles par la suite dans la description de cet outil de programmation et des opérations qui le constituent. La librairie permet la manipulation des polynômes, des matrices, des séries et des courbes elliptiques. Nous ne présentons pas les opérations sur les séries et les courbes elliptiques puisque nous n'y avons pas contribué. En revanche, nous essayons d'expliquer la philosophie de la librairie dans le chapitre I.2. Il est toujours difficile, dans le cadre d'un travail commun de programmation, de distinguer les contributions. Ce chapitre présente donc la librairie dans son ensemble. Il est clair que toutes les possibilités décrites ne peuvent être le fruit d'un travail individuel, mais s'agissant de la conception profonde, il s'agit bien d'un travail commun.

Dans les chapitres qui suivent, nous décrivons quelques fonctionnalités que nous avons implantées plus particulièrement, ceci afin d'illustrer quelques uns des aspects de la programmation de la librairie. Le chapitre I.3 traite des opérations sur les éléments et en particulier de notre implantation de la multiplication de Karatsuba sur les grands entiers. Le problème rencontré ici est d'implanter un algorithme bien connu de la meilleure façon possible. Le chapitre I.4 présente lui un test d'irréductibilité de polynôme basé en particulier sur l'algorithme de Berlekamp. Il sert d'illustration aux possibilités de la librairie Zen, puisque la fonction réalisée fonctionne quel que soit le corps fini utilisé. Enfin, le chapitre I.5 évoque les structures de données utilisées dans l'implantation des opérations matricielles de la librairie. Il s'agit là d'un problème de nature un peu différente, puisque les structures de données utilisées dans la description d'une matrice ont une influence importante sur les performances des opérations matricielles.

Cette première partie ne saurait donc pas constituer une documentation de la librairie Zen puisqu'y sont omises beaucoup de ses possibilités, développées en particulier par Reynald Lercier. Elle se veut surtout une introduction aux possibilités de la librairie, et présente certaines astuces d'implantation qui rendent le résultat efficace.

La réalisation d'un tel outil ne présente d'intérêt que par ses utilisations possibles. Nous présentons donc dans un deuxième temps deux applications au décodage général de codes linéaires. Pour cela, nous rappelons quelques notions d'algèbre dans le chapitre II.1. Ce chapitre, prolongement du chapitre I.1, est plus spécialement axé sur la théorie des codes, et introduit les notions de base nécessaires, telles que les métriques de Hamming et de Gabidulin. Une nouvelle formulation du décodage des codes de Gabidulin y est aussi introduite. Nous présentons ensuite au chapitre II.2 un algorithme probabiliste de décodage général des codes correcteurs d'erreur en distance de Hamming. Cet algorithme, développé conjointement avec Anne Canteaut, nous permet en particulier de déterminer les distances minimales réelles de six codes BCH sur les douze encore inconnues en longueur 511.

Le chapitre II.3 décrit ensuite un algorithme, fruit d'une collaboration avec Jacques Stern, qui permet le décodage général déterministe des codes linéaires en distance de Gabidulin. Une conséquence de cet algorithme est l'existence d'une borne sur les dimensions des codes linéaires en métrique de Gabidulin, qui améliore dans certains cas, la borne de Singleton.

Bien qu'ayant permis d'obtenir quelques résultats de théorie des codes, ces algorithmes ont pour vocation première l'attaque des systèmes cryptographiques basés sur les codes correcteurs d'erreur. Nous présentons donc certains de ces systèmes et les résultats des attaques correspondantes (chapitre III.1), avant de conclure (chapitre III.2).

Bien entendu, nous ne suivons pas l'ordre chronologique de ces résultats, puisque c'est la nécessité d'une programmation efficace de ces cryptanalyses qui a finalement conduit à l'élaboration de la librairie Zen.

Table des matières

Première partie

Algorithmique des corps finis

Chapitre I.1

Rappels d'algèbre

"Hé Dieu ! si j'eusse estudié
Au temps de ma jeunesse folle,
Et à bonnes mœurs dédié,
J'eusse maison et couche molle.
Mais quoi? je fuyois l'école."

François Villon
Le Grand Testament

Nous allons présenter le plus succintement possible les quelques outils d'algèbre dont nous avons besoin par la suite. Des descriptions plus détaillées existent dans de nombreux ouvrages. On pourra par exemple considérer [2-RDO85], ou pour ce qui concerne plus particulièrement les corps finis, la "bible rouge" [2-LN86].

Les propriétés qui suivent sont très classiques et nous ne donnons donc aucune démonstration pour ces résultats. Nous commençons par rappeler les notions de relation d'équivalence, de morphisme et de structure quotient (I.1.1), puis les définitions des structures classiques de groupe, d'anneau, d'idéal et de corps (I.1.2). Enfin, nous nous intéressons plus particulièrement aux corps finis (I.1.3) en rappelant la notion d'extension polynomiale et le résultat qui fonde la notion de corps de Galois (théorème 57). Nous terminons par le rappel de la formulation du théorème des restes chinois (théorèmes 62 et 63), dont il est fait un usage important par la suite.

Rappelons au passage que c'est en effet au mathématicien français Évariste Galois que l'on doit le concept général de corps fini. Poursuivant les travaux des grands algébristes que furent Legendre et Gauss, il formula en effet le théorème fondamental de la section I.1.3.c.a. Par la suite, Gauss formula les fondements de ce qui allait devenir la théorie de Galois.

I.1.1 Relations et fonctions

I.1.1.a Relations

Définition 1. Soient E et F deux ensembles, une *relation binaire* $\mathcal{R}$ de E vers F est la donnée d'une partie $U \subset E \times F$. Un élément $x \in E$ est *en relation* avec $y \in F$ selon la relation $\mathcal{R}$, et on note $x \mathcal{R} y$ si $(x, y) \in U$. Lorsque $F = E$, $\mathcal{R}$ est une *relation sur l'ensemble* E.

I.1.1.a.a Réflexivité, transitivité

Définition 2. Une relation $\mathcal{R}$ sur E est *réflexive* si et seulement si pour tout élément $x \in E$, on a

$$x \mathcal{R} x.$$

Une relation $\mathcal{R}$ sur E est *transitive* si et seulement si pour tout triplet $(x, y, z) \in E \times E \times E$, on a

$$(x \mathcal{R} y \text{ et } y \mathcal{R} z) \Rightarrow (x \mathcal{R} z).$$

I.1.1.a.b Relation d'équivalence

Définition 3. Une relation $\mathcal{R}$ sur E est *symétrique* si et seulement si pour tout couple $(x, y) \in E \times E$, on a

$$x \mathcal{R} y \Leftrightarrow y \mathcal{R} x.$$

Une *relation d'équivalence* est une relation réflexive, symétrique et transitive.

I.1.1.a.c Relation d'ordre

Définition 4. Une relation $\mathcal{R}$ sur E est *anti-symétrique* si et seulement si pour tout couple $(x, y) \in E \times E$, on a

$$(x \mathcal{R} y \text{ et } y \mathcal{R} x) \Rightarrow (x = y).$$

Une *relation d'ordre* est une relation réflexive, anti-symétrique et transitive.

I.1.1.b Lois de composition internes

Définition 5. On appelle *loi de composition interne* dans un ensemble E toute application de $E \times E$ dans E.

Dans ce qui suit nous notons $\oplus$ et $\otimes$ des lois de composition internes.

I.1.1.b.a Associativité, commutativité, élément neutre

Définition 6. Une loi de composition interne $\oplus$ est *associative* si et seulement si pour tout triplet $(x, y, z) \in E \times E \times E$, on a

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

Une loi de composition interne $\oplus$ est *commutative* si et seulement si pour tout couple $(x, y) \in E \times E$, on a

$$x \oplus y = y \oplus x$$

Un élément $0 \in E$ est un *élément neutre* pour la loi de composition interne $\oplus$ si et seulement si pour tout élément $x \in E$, on a

$$x \oplus 0 = 0 \oplus x = x.$$

I.1.1.b.b Distributivité

Définition 7. Une loi de composition interne $\otimes$ est *distributive à droite* par rapport à la loi de composition interne $\oplus$ si et seulement si pour tout triplet $(x, y, z) \in E \times E \times E$, on a

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z).$$

On définit similairement la *distributivité à gauche*. On parle simplement de distributivité lorsque les deux propriétés sont vérifiées.

I.1.1.b.c Élements inversibles, involutifs, idempotents

Nous considérons désormais que la loi de composition interne $\oplus$ possède un élément neutre noté 0.

Définition 8. Un élément $x \in E$ est *inversible à droite* si et seulement si il existe $y \in E$ tel que

$$x \oplus y = 0.$$

On définit de même l'inversibilité à gauche.
Un élément $x \in E$ est *involutif* pour la loi de composition interne $\oplus$ si et seulement si il est son propre inverse :

$$x \oplus x = 0.$$

Un élément $x \in E$ est *idempotent* pour la loi de composition interne $\oplus$ si et seulement si :

$$x \oplus x = x.$$

I.1.1.c Homomorphisme, isomorphisme

Définition 9. Soient E et F deux ensembles munis respectivement des lois de composition internes $\oplus$ et $\otimes$. Une application f est un *homomorphisme* de $(E,\oplus)$ dans $(F,\otimes)$ si et seulement si pour tout couple $(x,y) \in E \times E$, on a

$$f(x \oplus y) = f(x) \otimes f(y).$$

Lorsque $F = E$, on parle d'*endomorphisme*.

Théorème 10. *Si un homomorphisme de $(E,\oplus)$ dans $(F,\otimes)$ est bijectif, sa réciproque est un homomorphisme de $(F,\otimes)$ dans $(E,\oplus)$. On parle alors d'*isomorphisme. *Si $F = E$, on parle d'*automorphisme.

I.1.1.d Structure quotient

I.1.1.d.a Classes d'équivalence

Définition 11. Soit E un ensemble muni d'une relation d'équivalence $\mathcal{R}$. Pour tout élément $x \in E$, on appelle *classe d'équivalence* de x dans E pour la relation $\mathcal{R}$ l'ensemble des éléments $y \in E$ en relation avec x. On la note $\dot{x}$.

Théorème 12. *Les classes d'équivalence d'une relation $\mathcal{R}$ forment une partition de E.*

Définition 13. On appelle *ensemble quotient* de E par $\mathcal{R}$ noté $E/_{\mathcal{R}}$ l'ensemble des classes d'équivalence de $\mathcal{R}$ sur E. L'application s de E dans $E/_{\mathcal{R}}$ qui associe la classe d'équivalence d'un élément à celui-ci est appelée *surjection canonique*. On la note

$$\begin{array}{rccc} s & : & E & \to & E/_{\mathcal{R}} \\ & & x & \mapsto & \dot{x} \end{array}$$

I.1.1.d.b Structure quotient

Définition 14. Soit E un ensemble muni d'une loi de composition interne $\oplus$ et d'une relation d'équivalence $\mathcal{R}$. On dit que $\mathcal{R}$ et $\oplus$ sont *compatibles* si et seulement si pour tout quadruplet $(a,b,c,d) \in E^4$ on a

$$(a\,\mathcal{R}\,b \text{ et } c\,\mathcal{R}\,d) \Rightarrow (a \oplus c)\,\mathcal{R}\,(b \oplus d).$$

Théorème 15. *Si $\mathcal{R}$ et $\oplus$ sont compatibles dans E, il existe une unique loi de composition interne $+$ sur $E/_{\mathcal{R}}$ telle que la surjection canonique soit un morphisme de $(E,\oplus)$ dans $(E/_{\mathcal{R}},+)$. La structure $(E/_{\mathcal{R}},+)$ est appelée* structure quotient *de $(E,\oplus)$ par $\mathcal{R}$.*

Théorème 16. *Les propriétés de commutativité, d'associativité, de distributivité, d'existence d'un élément neutreet d'inversibilité se transmettent à la structure quotient.*

I.1.2 Structures

I.1.2.a Groupes

Définition 17. Un ensemble G muni d'une loi de composition interne $\otimes$ est un *groupe* si et seulement si $\otimes$ est associative, possède un élément neutre et tout élément de G est inversible pour $\otimes$.

Définition 18. Soit $(G,\otimes)$ un groupe, on appelle *sous-groupe* de G une partie stable H de G pour la loi de composition interne $\otimes$, telle que $(H,\otimes)$ soit un groupe.

I.1.2.a.a Sous groupe engendré par une partie

Théorème 19. *Soit $(G,\otimes)$ un groupe et $A \subset G$. Il existe un plus petit sous-groupe de G contenant A, noté $\mathcal{G}(A)$. C'est la plus petite partie stable contenant tous les éléments de A et leurs inverses. On dit de $\mathcal{G}(A)$ que c'est le* sous-groupe de G engendré par A.

Définition 20. On appelle *groupe monogène* tout groupe engendré par un élément, et *groupe cyclique* tout groupe monogène fini.

I.1.2.a.b Groupe quotient

Théorème 21. *Soit $a \in (G, \otimes)$, l'application*

$$\begin{array}{rccc} \lambda_a & : & G & \to & G \\ & & g & \mapsto & a \otimes g \otimes a^{-1} \end{array}$$

est un automorphisme.

Définition 22. Un sous-groupe H de G est dit *distingué* dans G si et seulement si pour tout élément $a \in G$, on a

$$\lambda_a(H) = H.$$

Théorème 23. *Soit $\mathcal{R}$ une relation d'équivalence sur G compatible avec la loi de composition interne $\otimes$ alors il existe un unique sous-groupe H de G tel que*

$$x \,\mathcal{R}\, y \Leftrightarrow x^{-1} \otimes y \in H \Leftrightarrow y \otimes x^{-1}.$$

Ce sous-groupe est distingué dans G. Pour la relation d'équivalence $\mathcal{R}$, les éléments y appartenant à la classe d'équivalence de x sont donc tous les éléments de $x \otimes H$. On déduit du théorème 12 que les $x \otimes H$ forment une partition de G.

Théorème 24. *Soit H un sous-groupe distingué dans G, et $\mathcal{R}$ la relation d'équivalence compatible avec $\otimes$ associée à H. Il existe une unique loi de composition interne $\times$ sur $G/_{\mathcal{R}}$ telle que la surjection canonique soit un morphisme de $(G, \otimes)$ dans $(G/_{\mathcal{R}}, \times)$. La structure $(G/_{\mathcal{R}}, \times)$ est appelée* groupe quotient *de $(G, \otimes)$ par H et notée $G/_H$.*

Théorème 25. *Si G est un groupe commutatif, tout sous-groupe H de G est distingué dans G.*

I.1.2.a.c Groupes finis

Définition 26. Soit $(G, \otimes)$ un groupe, soient $a \in G$ et $\mathcal{G}(a)$ le sous-groupe engendré par a, alors si $\mathcal{G}(a)$ est fini, son cardinal est appelé *ordre de a pour la loi $\otimes$* et noté

$$\mathrm{ord}(a) = \mathrm{card}(\mathcal{G}(a)).$$

Remarque. On déduit du théorème 23 que dans le cas d'un groupe fini on a

$$\mathrm{ord}(a) | \mathrm{card}(G).$$

où | signale la "divisibilité" de $\mathrm{card}(G)$ par $\mathrm{ord}(a)$.

On suppose définis dans la suite l'ensemble $\mathbb{N}$ des entiers naturels, l'ensemble $\mathbb{Z}$ des entiers relatifs et on considère désormais l'addition, notée $+$, comme loi de composition interne conférant à $\mathbb{Z}$ la structure de groupe commutatif. Les relations d'ordre classiques sur $\mathbb{Z}$ sont notées comme de coutume.

Théorème 27. *Les sous-groupes additifs de $\mathbb{Z}$ sont définis par la donnée d'un entier naturel $n \in \mathbb{N}$ et notés :*

$$n\mathbb{Z} = \{y \in \mathbb{Z} \,/\, \exists x \in Z y = \sum_{i=1}^{n} x\}$$

Théorème 28. *D'après les théorèmes 25 et 24, on peut définir pour tout $n \in \mathbb{N}$ le groupe quotient $\mathbb{Z}/_{n\mathbb{Z}}$ appelé* groupe des entiers modulo n.

I.1.2.b Anneaux

Définition 29. Un *anneau* est la donnée d'un ensemble et de deux lois de composition internes $(A, \oplus, \otimes)$ tels que

- la structure $(A, \oplus)$ soit un groupe commutatif (par convention, on note 0 son élément neutre et $-x$ l'inverse d'un élément par la loi $\oplus$) ;
- la loi de composition interne $\otimes$ admette un élément neutre noté 1 ;
- la loi $\otimes$ soit distributive par rapport à la loi $\oplus$.

On parle d'anneau *commutatif* lorsque la seconde loi de composition interne $\otimes$ est elle même commutative.

Théorème 30. *Dans un anneau, 0 est un* élément absorbant, *donc qui n'admet pas d'inverse :*

$$\forall a \in A, a \otimes 0 = 0 \otimes a = 0.$$

On note $A^\star = A \backslash \{0\}$.

Définition 31. Un anneau dans lequel l'implication

$$ab = 0 \Rightarrow a = 0 \text{ ou } b = 0$$

est vérifiée pour tout couple d'éléments (a, b), est dit *intègre*.

On peut définir par récurrence la multiplication classique sur $\mathbb{N}$. Soit $\times$ la loi définie par

$$\begin{array}{rccc} \times & : & \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \\ & & (x, y) & \mapsto & \sum_{i=1}^{x} y \end{array}$$

et étendue à $\mathbb{Z}$ de la manière naturelle, alors $(\mathbb{Z}, +, \times)$ est un anneau commutatif intègre.

I.1.2.b.a Division, pgcd

Définition 32. On dit que t est un *diviseur* de z si et seulement si il existe q tel que $z = q \times t$. On dit de p qu'il est premier s'il n'accepte que 1 et lui même comme diviseur.

Définition 33. Une partie I d'un anneau $(A, \oplus, \otimes)$ est un *idéal* de A si et seulement si :

- pour tout couple $(x, y) \in I^2$ on a $x \oplus (-y) \in I$.
- pour tout couple $(a, x) \in A \times I$ on a $x \otimes a \in I$.

Théorème 34. *Pour toute partie $P \subset A$, il existe un plus petit idéal I de A contenant P. C'est par définition* l'idéal engendré par P.

Théorème 35. *Pour tout $a \in A$, $aA = \{y \in a \ / \ \exists x \in A, y = a \otimes x\}$ est un idéal de A. On dit de aA que c'est un* idéal principal *de A.*

Définition 36. Soient $a_1, \ldots, a_N$ N éléments non nuls d'un anneau A, on appelle *plus grand commun diviseur (pgcd)* et on note $a_1 \wedge \cdots \wedge a_N$ tout élément générateur de l'idéal engendré par $\{a_1, \ldots, a_N\}$. Si l'idéal engendré par $\{a_1, \ldots, a_N\}$ est l'anneau A tout entier, alors les a_i sont dits *premiers entre eux.*

Théorème 37 Bezout. *Soient $a_1, \ldots, a_N$ N éléments non nuls d'un anneau A, alors les a_i sont premiers entre eux si et seulement s'il existe $u_1, \ldots, u_N$ tel que*

$$\sum_{i=1}^{N} a_i u_i = 1.$$

Cas de l'anneau des entiers naturels

Théorème 38. *Tout idéal de* $(\mathbb{Z}, +, \times)$ *est principal.*

Théorème 39 (division euclidienne). *Pour tout* $(z,t) \in \mathbb{Z} \times \mathbb{Z}^\star$*, il existe un unique couple* $(q,r) \in \mathbb{Z} \times \mathbb{N}$ *tel que*

$$z = q \times t + r \text{ avec } r < |t|.$$

La définition 36 est bien entendu cohérente avec la définition classique du pgcd. Le théorème 39 signifie en particulier que tout nombre admet une décomposition unique en facteurs premiers. Si on note $a \triangle b$ le pgcd obtenu de façon classique par les décompositions en facteurs premiers de a et b, alors l'idéal engendré par $(a \triangle b)$ est effectivement le plus petit idéal contenant les idéaux $a\mathbb{Z}$ et $b\mathbb{Z}$. On a donc bien $a \triangle b = a \wedge b$ pour tout $(a,b) \in \mathbb{Z}^2$.

I.1.2.b.b Anneau quotient

Théorème 40. *Soit* $(A, \oplus, \otimes)$ *un anneau et* I *un idéal de* A*. La relation* $\mathcal{R}$ *définie par* $x \mathcal{R} y \Leftrightarrow x \oplus (-y) \in I$ *est une relation d'équivalence compatible avec les deux lois. L'ensemble quotient* $A/_{\mathcal{R}}$ *noté* $A/_I$ *muni des deux lois quotient a une structure d'anneau. On l'appelle* anneau quotient *de* A *par* I*.*

Note 41. L'ensemble quotient $\mathbb{Z}/_{n\mathbb{Z}}$ a une structure d'anneau. Celui-ci est intègre si et seulement si l'entier n est premier.

I.1.2.b.c Caractéristique

Définition 42. Soit A un anneau, on appelle *caractéristique* d'un anneau le plus petit entier n tel que

$$\forall a \in A \sum_{i=1}^{n} a = 0.$$

Si un tel entier n'existe pas, on parle de "*caractéristique nulle*".

Note 43. Les anneaux $\mathbb{Z}/_{n\mathbb{Z}}$ sont de caractéristique n.

I.1.2.c Corps

Définition 44. On appelle *corps* tout anneau dont tous les éléments non nuls sont inversibles. On appelle *corps premier* tout corps n'admettant que lui même comme sous-corps.

Théorème 45. *Tout corps est un anneau intègre. Réciproquement, tout anneau fini intègre est un corps fini.*

Corollaire 46. *Pour tout entier premier* p*,* $\mathbb{Z}/_{p\mathbb{Z}}$ *est un corps premier.*

I.1.2.c.a Corps des fractions d'un anneau intègre

Théorème 47. *Soit* A *un anneau intègre, soit* E *l'ensemble des couples de* $A \times (A^\star)$*, alors la relation* $\mathcal{R}$ *définie sur* E *par*

$$(n,d)\,\mathcal{R}\,(n',d') \Leftrightarrow nd' - n'd = 0$$

est une relation d'équivalence. Les lois définies sur E *par*

addition : $(n,d) + (n',d') = (nd' + n'd, dd')$,
multiplication : $(n,d) \times (n',d') = (nn', dd')$,

sont internes sur E*. La structure quotient* $(E/_{\mathcal{R}}, +, \times)$ *est un corps commutatif.*

Définition 48. Nous appelons corps $\mathbb{Q}$ des *fractions rationnelles* l'ensemble obtenu par l'application de la construction précédente à l'anneau intègre $\mathbb{Z}$.

I.1.3 Corps finis

I.1.3.a Anneau des polynômes

Nous considérons dans ce qui suit la définition classique des polynômes à une variable définis sur un anneau commutatif A. D'autre part nous rappelons que l'ensemble des polynômes à une variable X, noté $A[X]$, muni des lois de composition internes a une structure d'anneau. Tout polynôme $P \in A[X]$ s'exprime de manière unique dans la base canonique des puissances de la variables X :

$$P(X) = \sum_{i=0}^{d} a_i X^i,$$

où les coefficients a_i appartiennent à A.

Définition 49. Le degré de P est défini par

$$\deg P = \max\{i \in \mathbb{N} \,/\, \ a_i \neq 0\}.$$

Définition 50. Un polynôme est *monique* si et seulement si son coefficient de plus haut degré vaut 1 :

$$P(X) = \sum_{i=0}^{d-1} a_i X^i + X^d.$$

I.1.3.a.a Irréductibilité

Théorème 51 (division euclidienne). *Soient $A \in A[X]$ et $B \in (A[X])^\star$, il existe un unique couple de polynômes $(Q, R) \in A[X]^2$ tels que $\deg R < \deg B$ et*

$$A = BQ + R.$$

Définition 52. Un polynôme $P \in A[X]$ est *irréductible* sur A s'il est de degré supérieur à 1 et que pour tout polynôme de degré inférieur $Q \in A[X]$, $\deg Q < \deg P$, P et Q sont premiers entre eux.

I.1.3.a.b Zéros

Définition 53. On appelle *zéro* ou *racine* d'un polynôme $P(X)$ tout élément $a \in A$ tel que $P(a) = 0$.

Théorème 54. *Si $P \in A[X]$ admet une racine $a \in A$, alors P n'est pas irréductible ; il est divisible par le polynôme $(X - a)$.*

I.1.3.b Extension polynomiale

Nous avons vu par les théorèmes 35 et 40 comment étendre les anneaux. Nous notons $A[X]/_{P(X)}$ l'anneau quotient obtenu à partir de l'anneau des polynômes $A[X]$ défini sur l'anneau A et d'un polynôme $P(X)$ pris comme élément générateur de l'idéal principal $P(X)A[X]$. Dans l'anneau $\mathbb{Z}/_{n\mathbb{Z}}$ nous parlions d'opérations modulo n, *mutatis mutandis* nous parlerons ici d'opérations modulo le polynôme $P(X)$.

I.1.3.c Corps de Galois

Théorème 55. *Soit p un entier premier, notons $GF(p)$ le corps premier $\mathbb{Z}/_{p\mathbb{Z}}$. Soit $P(X)$ un polynôme irréductible sur $GF(p)$, l'anneau $GF(p)[X]/_{P(X)}$ a une structure de corps de cardinalité $p^{\deg P}$. Soit $Q(X)$ un polynôme irréductible sur $GF(p)$ distinct de $P(X)$ mais de même degré, alors le corps $GF(p)[X]/_{Q(X)}$ est isomorphe au précédent. Pour tout entier m et tout entier premier p, il existe au moins un polynôme irréductible de $GF(p)[X]$ de degré m.*

Théorème 56. *Si K est un corps fini de cardinal n, alors il existe un premier p et un entier m tel que $n = p^m$, et il existe un isomorphisme entre K et $GF(p^m)$.*

Tous ces corps étant isomorphes, on appelle *corps de Galois de cardinalité p^m*, noté $GF(p^m)$, un corps fini de cardinalité p^m obtenu par la construction précédente.

I.1.3.c.a Théorème de Galois

Théorème57. *Si $P(X)$ est un polynôme irréductible à coefficients dans $GF(p)$ de degré m, alors P a une racine α dans $GF(p^m)$.*

Corollaire58. *La famille $(\alpha, \alpha^2, \ldots, \alpha^{m-1})$ est une base que l'on dit* associée *à P.*

Corollaire59. *Le polynôme $P(X)$ a m racines distinctes dans $GF(p^m)$ qui sont*

$$(\alpha, \alpha^p, \alpha^{p^2}, \ldots, \alpha^{p^{m-1}}).$$

I.1.3.c.b Bases normales

Corollaire60. *Dans tout corps fini K de cardinal p^m, il existe au moins un élément $\beta \in K$, tel que la famille*

$$(\beta, \beta^p, \beta^{p^2}, \ldots, \beta^{p^{m-1}})$$

soit une base de K :

$$\forall x \in K \;\; x = \sum_{i=0}^{m-1} \alpha_i \beta^{p^i},$$

avec les α_i éléments de $\mathbb{Z}/_{p\mathbb{Z}}$. Une telle famille est appelée base normale *de K.*

Théorème61. *Dans toute extension polynomiale sur un corps fini K de cardinalité q, il existe une base normale*

$$(\beta, \beta^q, \beta^{q^2}, \ldots, \beta^{q^{m-1}}).$$

I.1.3.c.c Théorèmes des restes chinois

Théorème62. *Soient m et n deux entiers naturels premiers entre eux, alors $\mathbb{Z}/_{(mn)\mathbb{Z}}$ est isomorphe à $\mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}}$. Plus précisément, l'application*

$$\begin{array}{rccc} \theta & : & \mathbb{Z}/_{mn\mathbb{Z}} & \to & \mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}} \\ & & x & \mapsto & (x \bmod m, x \bmod n) \end{array}$$

est un isomorphisme et sa réciproque est

$$\theta^{-1}(x_m, x_n) = x_m n (n^{-1} \bmod m) + x_n m (m^{-1} \bmod n) \bmod mn.$$

Théorème63. *Soient P et Q deux polynômes sur $GF(q)$ premiers entre eux, alors $GF(q)[X]/_{PQ(X)}$ est isomorphe à $GF(q)[X]/_{P(X)} \times GF(q)[X]/_{Q(X)}$. Plus précisément, l'application*

$$\begin{array}{rccc} \theta & : & GF(q)[X]/_{PQ(X)} & \to & GF(q)[X]/_{P(X)} \times GF(q)[X]/_{Q(X)} \\ & & \Pi & \mapsto & (\Pi \bmod P, \Pi \bmod Q) \end{array}$$

est un isomorphisme et sa réciproque est

$$\theta^{-1}(\Pi_P, \Pi_Q) = \Pi_P Q (Q^{-1} \bmod P) + \Pi_Q P (P^{-1} \bmod Q) \bmod PQ.$$

Chapitre I.2

Librairie de programmation Zen

"Ne me dites pas que ce problème est difficile. S'il n'était pas difficile, ce ne serait pas un problème"

Maréchal Foch

Nous présentons maintenant la librairie de programmation Zen. Il s'agit d'un travail, en commun avec Reynald Lercier du Laboratoire d'Informatique de l'Ecole Polytechnique, qui a pour objectif de fournir un outil simple (!) de programmation en langage C pour le calcul dans les corps finis et plus généralement dans toute extension finie définie sur un anneau d'entiers. Comme souvent en pareil cas, la recherche de la simplicité peut entraîner des pertes d'efficacité et obtenir un outil à la fois simple et efficace nous a conduit au niveau de la programmation interne à un édifice relativement complexe. Ainsi, la librairie Zen compte actuellement plus de 65000 lignes de code ! Ce chapitre ne constitue pas une documentation de référence de la librairie, mais plutot une tentative d'explication des concepts utilisés.

La librairie Zen est un outil de programmation et non un outil de calcul formel. Pour résoudre un problème simple, il sera souvent plus rapide de se contenter d'un petit programme en Maple ou en Axiom. Par contre, si le problème s'avère engendrer des calculs effectifs intensifs, la librairie Zen permettra d'obtenir une amélioration très sensible de la rapidité des calculs. Ceci se fera au prix d'un travail de programmation relativement simple, c'est du moins le but recherché, de sorte que la transition depuis un programme Maple ou Axiom devrait être aisée dès lors que la programmation en C est à peu près maîtrisée.

Nous commençons par effectuer un état de l'art du calcul informatique dans les corps finis avant de rappeler brièvement l'historique de la librairie (I.2.1). Nous présentons ensuite l'environnement de programmation utilisé qui permet une portabilité facilitée (I.2.2). La section I.2.3 présente les principes de base de la librairie, à savoir les types définis, la représentation des anneaux premiers, le principe général de la syntaxe des opérations de Zen, la définition d'une extension polynomiale, et les différentes arithmétiques disponibles. Le problème de l'optimisation des calculs est ensuite abordé, avec les différentes options possibles prévues (I.2.4) : d'une part les précalculs, qui ne modifient pas la représentation interne des éléments, peuvent donc être effetués à tout moment, et qui améliorent généralement les performances d'une opération arithmétique particulière ; d'autre part les clones, qui utilisent une représentation différente de la représentation standard, en vue d'accélérer les opérations arithmétiques dans leur ensemble. Nous terminons en évoquant la gestion des erreurs de la librairie qui s'avère importante lors du déverminage d'un programme (I.2.5).

I.2.1 Introduction

I.2.1.a État de l'art

La question que l'on est en droit de se poser est alors celle du bien-fondé de cet investissement. En effet, la réalisation d'un tel outil ne s'impose que s'il n'existe pas par ailleurs. Or qu'en est-il :

1. Il existe des outils de calcul formel (Maple [3-CGG+90], Mathematica [3-AB92]...) dont les performances sont très moyennes au niveau de l'efficacité de calculs sur les grands entiers. Ces outils sont

en effet optimisés pour le calcul formel, et il ne faut pas leur demander plus. Typiquement, l'application à la cryptographie exige de pouvoir manipuler des nombres de 150 chiffres décimaux. De tels nombres sous Maple, par exemple, conduisent à des calculs lents. Les choses se gâtent encore si l'on souhaite travailler dans des extensions. Maple sait travailler sur des extensions de niveau 1 (du type $\mathbb{Z}/_{p\mathbb{Z}}[X]/_{P(X)}$). Avec un peu d'habileté, au prix d'un véritable travail de programmation, on peut arriver à travailler sur des extensions de niveau 2 (du type $(\mathbb{Z}/_{p\mathbb{Z}}[X]/_{P(X)})[Y]/_{Q(Y)}$). Même à ce niveau, les performances sont lamentables ; on pourra en juger sur le petit exemple de l'annexe A. Or nous souhaitions pouvoir travailler facilement sur des extensions de niveau 3 et plus, ce que Maple ne sait pas faire.

2. Un autre outil considérable, à savoir la librairie de calculs Pari [1-Par], semble assez curieusement ignorer le monde des corps finis. Les extensions polynomiales en particulier ne sont accessibles que de façon détournée.

3. A l'interface entre calcul formel et calcul effectif, Axiom [1-NAG] permet de travailler dans les extensions, mais de manière peu efficace.

4. Pour la programmation en langage C, la librairie BIGMOD [1-MorBM], développée par François Morain pour l'implantation de ses certificats de primalité par la méthode des courbes elliptiques, permet de travailler modulo un nombre entier [1-MorEC].

   À la base, elle utilise la librairie BIGNUM [7-SVH89] développée conjointement par l'INRIA et Digital PRL qui permet de manipuler de grands entiers. Cette dernière présente l'avantage de posséder pour un grand nombre d'architectures un noyau de fonctions écrit en assembleur, ce qui accélère grandement les calculs (de 30 à 50 % selon les calculs effectués et le type d'opérations utilisées). Malheureusement, force est de constater que les fonctions de BIGNUM sont très peu simples d'emploi. En outre, certaines d'entre elles peuvent entraîner des erreurs de calcul si elles sont mal employées. En effet, par souci d'efficacité, aucun test n'est effectué pour détecter d'éventuelles erreurs.

   Ceci fait de BIGMOD un outil efficace mais présentant les mêmes défauts que BIGNUM. En outre, BIGMOD ne permet pas de travailler sur plusieurs anneaux à la fois puisque le module est déclaré globalement. De plus, la syntaxe des fonctions BIGMOD est parfois source de confusions.

   Reconnaissons néanmoins que BIGMOD a été souvent une source d'inspiration pour la programmation de ZEN, et ce même si aucune fonction de BIGMOD n'a été reprise telle quelle.

5. Enfin, Lidia [1-LiD95] est une autre librairie de calcul qui permet des opérations efficaces, mais uniquement dans les corps finis premiers.

La réalisation d'un outil tel que nous le concevions s'est donc avérée nécessaire pour les problèmes qui nous intéressaient (voir partie II) et nous pensons que cet outil pourra s'avérer utile à d'autres.

I.2.1.b Historique

La librairie ZEN tire en fait ses racines d'un lointain passé ! En effet, la programmation de l'algorithme de Leon [4-Leo88] pour la cryptanalyse du système de McEliece [7-McE78] (voir chapitre II.2) avait conduit à l'élaboration de fonctions matricielles efficaces dans $GF(2)$. Par la suite, l'étude de l'algorithme de Sidel'nikov-Shestakov [4-SS92] a fourni l'occasion de développer des outils similaires dans $GF(2^m)$. Ces premières implantations utilisaient le format et certaines fonctions de la librairie BIGNUM [7-SVH89].

Parallelement, Reynald Lercier s'appuyant sur BIGMOD [1-MorBM], développait ses outils dans $GF(p)$ pour factoriser de grands entiers par la méthode des courbes elliptiques.

Or tous ces algorithmes fonctionnent également dans tout corps fini. Le besoin s'est donc fait sentir d'un échange d'outils. Mais, nos programmes à tous deux devaient de toute manière être ré-écrits puisque nos outils respectifs étaient différents.

Nos implantations utilisant à la base la même librairie BIGNUM, elles présentaient en outre des similitudes de conception. C'est alors qu'a germé l'idée d'une librairie qui permettrait de programmer une fois pour toutes nos algorithmes. Le but était donc d'obtenir un outil qui autorise l'utilisation d'un algorithme dans tout corps fini, quelle que soit la construction de celui-ci, sans changer une ligne du programme de calcul.

Nous avions tous les outils dans un format à peu près compatible (opérations sur les éléments, les polynômes et les matrices dans $GF(2)$ et $GF(p)$), restait à assembler les morceaux...

I.2.2 Compilation et environnement de programmation

Idéalement, une programmation rigoureuse en C doit permettre la portabilité sur toute machine possédant un compilateur de ce langage. Malheureusement, l'expérience prouve qu'il n'en est rien ! Ou plus exactement, une programmation recherchant l'efficacité risque assez facilement de transgresser ces règles. Il est donc absolument indispensable de fournir avec la librairie un programme de test qui permet, lors de la compilation sur une machine d'un nouveau type, de vérifier que des effets de bord n'apparaissent pas.

Cette recherche d'efficacité impose en outre d'utiliser sur chaque machine le compilateur le plus performant. C'est généralement le compilateur du constructeur pour une raison que nous verrons dans la section I.2.3. Or chaque compilateur a des options de compilation qui lui sont propres. Il fallait donc trouver un moyen d'obtenir automatiquement les paramètres de compilation à utiliser, ou en tout cas que cela soit transparent pour l'utilisateur.

D'autre part, la librairie Zen compte plus de 200 fichiers à compiler. Écrire pour chacun d'entre eux la portion de fichier Makefile correspondante se serait avéré très fastidieux et source d'erreurs nouvelles. Là encore un outil automatique était à trouver.

Comme ce genre de problème est commun à tout travail de programmation un peu conséquent, un tel outil existe déjà. Imake [3-DuB93] a été développé par les programmeurs du serveur X et permet par des fichiers de configurer chaque architecture pour autoriser la génération automatique des fichiers Makefile. On utilise ainsi des fichiers imakefile qui sont beaucoup plus faciles à gérer. En outre, comme le serveur X est généralement disponible sur toute machine Unix, ce programme l'est aussi. Nous avons ainsi pu vérifier que notre librairie se compilait sur des architectures largement différentes (voir figure 2.2), par la même séquence de commandes (voir figure 2.1), et que les tests étaient correctement effectués.

Dans un tout autre registre, il faut rappeler qu'une librairie n'est utilisable par d'autres utilisateurs que ses programmeurs que si elle dispose d'une documentation satisfaisante. Or, conserver une documentation à jour dans un outil en pleine évolution est une tâche difficile. Pour tenter de la faciliter, nous avons utilisé une idée de Knuth [1-KL] afin que les sources des programmes (en C) et ceux de la documentation (en LaTeX[3-Lam94]) soient tous dans des fichiers identiques. Nous n'avons pas repris tel quel le format proposé par Knuth, car il conduit à des fichiers difficilement lisibles. En effet, au format Cweb, les différentes parties du source C peuvent être permutées dans le fichier source et la compilation nécessite donc une étape de filtrage supplémentaire. C'est pourquoi, tout en gardant cette idée de stocker les deux types de sources dans les mêmes fichiers, nous avons préféré écrire notre propre programme. Celui-ci récupère les portions successives de documentation au sein des commentaires des fichiers sources C de la librairie et génère automatiquement un index facilitant l'accès rapide aux informations réparties dans les quelques 500 pages de la documentation. Il est aussi possible d'inclure des portions de source C dans les fichiers LaTeX. Ce concept permet ainsi lors de la modification de la programmation de modifier aussitôt la documentation correspondante.

I.2.3 Principes retenus

Pour permettre de conserver à la fois la simplicité de programmation et l'efficacité, il nous est très vite apparu qu'un concept de style orienté-objet était nécessaire. En effet, un ordinateur ne calcule pas efficacement de la même manière dans les corps de caractéristique 2 et dans ceux de caractéristique p. Tout le problème était donc d'arriver à cacher cette différence pour l'utilisateur.

Ce dernier travaille dans un anneau. À chaque type d'anneau est associée une famille d'algorithmes qui permettent de calculer efficacement dans cet anneau. Nous avons donc décidé que pour notre programmation, un anneau serait une structure contenant outre les paramètres de construction de l'anneau, une liste de pointeurs de fonctions. Lors de la construction d'un anneau, ces pointeurs sont initialisés selon le type de l'anneau. Chaque fonction de calcul fait intervenir l'anneau dans lequel on travaille, et c'est donc le pointeur de fonction qui est utilisé pour le calcul. Chaque opération ne fait donc intervenir qu'une déréférenciation

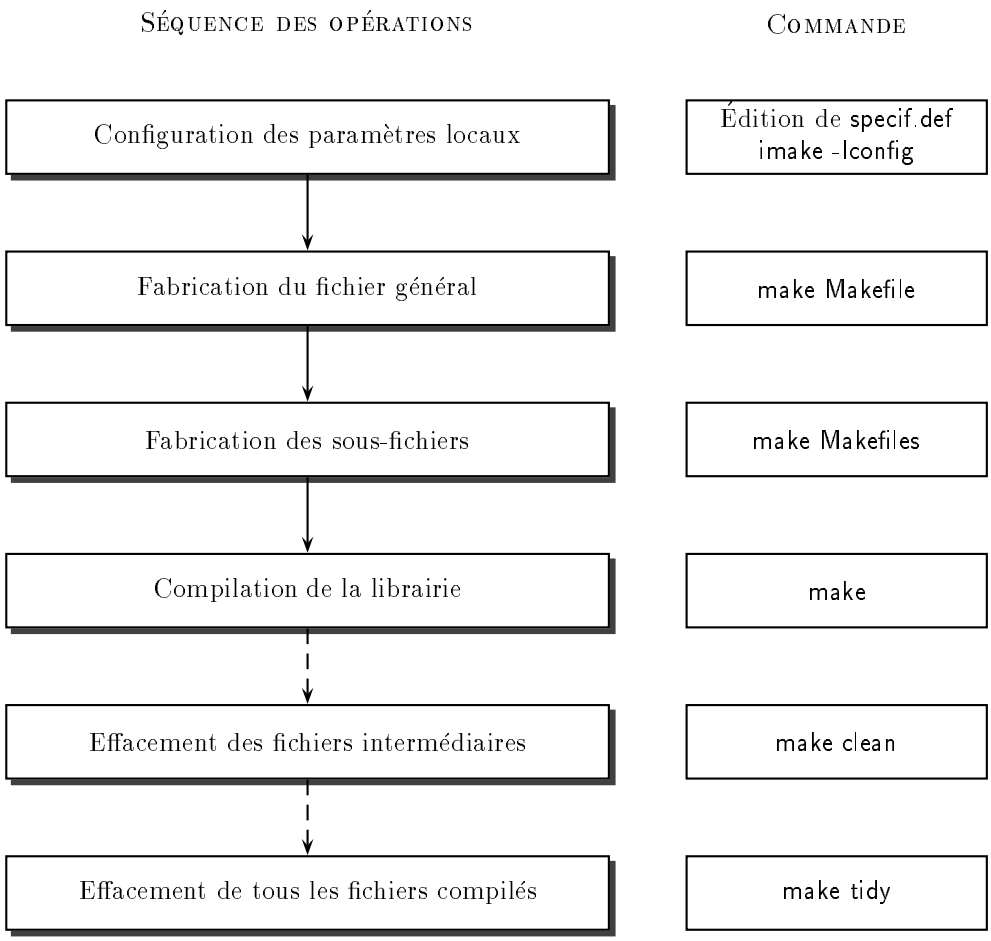


FIG. 2.1 – *Compilation de* ZEN.

Machine	Système d'exploitation	Compilateurs utilisés
Sun4	SunOS	cc, gcc
	Solaris	SUNWspro/cc, gcc
DEC	Ultrix	gcc
VAX		
Alpha	OSF	cc, gcc
HP	HP-UX	cc, gcc
PC	netbsd	gcc
	linux	
	OS2	

FIG. 2.2 – *Portabilité de* ZEN.

Type C	Objet mathématique
ZENRing	Un anneau, c'est-à-dire la donnée des opérations élémentaires sur les éléments de cet anneau. Tout programme doit commencer par la construction des différents ZENRings qui vont être utilisés.
ZENElt	Un élément d'un anneau. Selon le type de l'anneau, un ZENElt pourra être un élément de $\{0,1\}$ ou un polynôme de polynômes sur $\mathbb{Z}/_{251\mathbb{Z}}$, mais dans tous les cas il sera de type ZENElt et manipulable avec la même syntaxe de fonctions.
ZENPoly	Un polynôme de ZENElts. Comme le type ci-dessus, ce type est manipulable de la même manière dans tout anneau ZENRing. La seule contrainte est que le degré maximal du polynôme est exigé lors de la création d'un ZENPoly. Les opérations effectuées ne doivent donc pas conduire à l'obtention d'un degré supérieur.
ZENMat	Une matrice de ZENElts. Comme il s'agit d'un objet à deux dimensions, la structure de données utilisée consiste en une double référence sur un tableau de ZENElts. Cette double référence peut donc privilégier soit les lignes, soit les colonnes de la matrice. Dans le cas binaire, et selon les opérations, il peut être très avantageux d'utiliser une structure plutôt que l'autre, mais bien entendu, quelle que soit ce choix, le résultat obtenu sera le même. Cette double structure sera précisée au chapitre I.5.

FIG. 2.3 – *Types de* ZEN

supplémentaire à chaque appel de fonction. Ceci évite le recours aux tests qui sont très pénalisant dès lors qu'une procédure est appelée de l'intérieur d'une boucle.

En terminologie orienté-objet, on peut voir cette approche comme la réalisation de classes d'anneaux dans lesquelles les opérations sur les éléments-objets sont définies de manière propre. Mais alors, pourquoi ne pas avoir utilisé un langage comme le C++ ? La réponse est simple : parce que nous souhaitions programmer la librairie en C ! Cependant, cette affirmation péremptoire n'est pas aussi sectaire qu'elle en a l'air. En effet les compilateurs du langage C++ ont fait de très gros progrès ces dernières années. Mais ils restent néanmoins un peu en deça de leurs concurrents en C. Cela tient au fait que les noyaux Unix sont programmés en langage C, et chaque constructeur doit donc optimiser son compilateur C pour que sa machine soit la plus rapide. De ce fait, les compilateurs C des constructeurs utilisent au maximum les resources de la machine, alors que les compilateurs C++ ne peuvent qu'appliquer des méthodes d'optimisation plus générales. On peut noter d'ailleurs que les programmeurs de la librairie lidia [1-LiD95], écrite en C++, sont actuellement en train de reprogrammer plusieurs de leurs modules en C par souci d'efficacité.

Nous allons maintenant décrire plus précisément la librairie. Nous utiliserons ce Style d'Écriture pour indiquer toute fonction ou variable définie dans ZEN.

I.2.3.a Types

La librairie ZEN définit un certain nombre de nouveaux types. Les principaux sont énumérés dans le tableau de la figure 2.3.

I.2.3.b Anneaux premiers

Comme nous travaillons dans les anneaux finis, il nous faut commencer par fabriquer un anneau modulaire $\mathbb{Z}/_{n\mathbb{Z}}$. Dans un programme en C, on commence donc par déclarer une variable d'anneau ZENRing R. Il faut ensuite initialiser cette variable en utilisant une représentation de l'entier n. Puisque nous avons basé ZEN sur BIGNUM c'est donc à l'aide d'une représentation de ce type que nous définissons n. Rappelons brièvement qu'un entier n peut s'écrire de façon unique dans une base 2^s-aire. Dans BIGNUM, un entier est donc représenté par un couple (n,nl) de deux variables de type (BigNumDigit *,int). La valeur de n est alors:

$$n = \sum_{i=0}^{\mathsf{nl}-1} \mathsf{n}[i](2^s)^i.$$

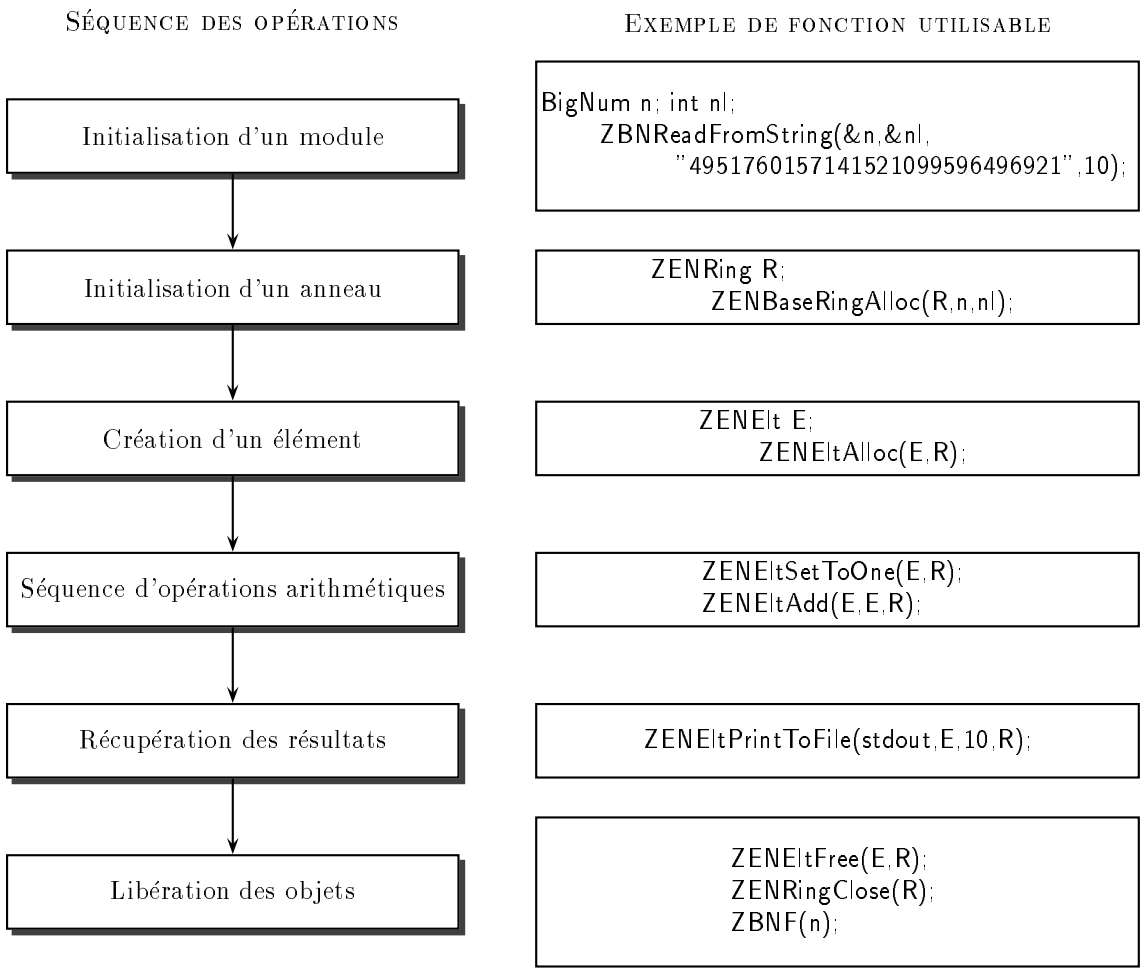


FIG. 2.4 – *Résumé des opérations sur un objet mathématique dans* $\mathbb{Z}/_{n\mathbb{Z}}$.

Dans notre cas, s est la taille d'un entier machine (SIZE_BLOC vaut 32 ou 64 bits).

L'opération d'initialisation se réalise alors simplement à l'aide de la fonction ZENBaseRingAlloc(R,n,nl). Cette fonction retourne 0 en cas de succès. En règle générale, toutes les fonctions de ZEN retournent 0 en cas de succès. Si une erreur se produit, par exemple une erreur d'allocation pour cause de mémoire insuffisante, une valeur d'erreur non nulle est retournée. Nous verrons au paragraphe I.2.5 comment utiliser au mieux cette gestion des erreurs. Certaines fonctions de ZEN ne retournent rien lorsqu'aucune erreur détectable n'est possible au sein de la fonction. Toutes ces informations sont disponibles dans la documentation de la librairie [7-CL96].

Pour utiliser des objets mathématiques sur cet anneau, il suffit de même de déclarer ces objets, puis de les initialiser. Ainsi, pour utiliser un élément de $\mathbb{Z}/_{n\mathbb{Z}}$, on déclare ZENElt E, puis on initialise cet élément par la commande ZENEltAlloc(E,R). On procède de même pour les autres types d'objets.

La librairie permet ensuite de fixer une valeur initiale par un certain nombre de commandes :

- ZENEltSetToZero(E,R) annule l'élément E.
- ZENEltSetToOne(E,R) donne la valeur de l'élément unité à E.
- ZENZToElt(E,n,nl,R) assigne la valeur de l'entier BIGNUM (n,nl).
- ZENEltReadFromString(E,s,base,R) lit la chaîne de caractères s dans la base indiquée (de 2 à 16), et assigne la valeur de E.
- ZENEltReadFromFile(E,fd,base,R) procède de même mais avec un fichier ouvert en lecture décrit par le pointeur FILE * fd.

Lorsqu'un objet n'est plus utilisé, il convient de libérer la place mémoire qu'il occupe, et qui peut parfois être importante. Pour un élément, on indique ZENEltFree(E,R), et pour un anneau ZENRingClose(R). La figure 2.4 résume les différentes opérations.

Types			Objet	Paramètres	Anneau
d'arithmétique	d'objet	d'opération	modifié		à utiliser
ZEN	Elt	Multiply	(X,	A, B,	R)

FIG. 2.5 – *Syntaxe de* ZEN

I.2.3.c Syntaxe de ZEN

Ces premiers exemples permettent déjà de comprendre ce que l'on peut appeler un peu pompeusement la sémantique de la librairie. Toutes les fonctions de la librairie ont en effet un nom et des paramètres qui obéissent à quelques règles simples permettant de deviner très rapidement la syntaxe de la fonction désirée. La figure 2.5 résume ces principes.

À cela il convient d'ajouter que toutes les "fonctions" de ZEN sont en fait des "macros", ce qui permet, de manière transparente pour l'utilisateur, de passer certains paramètres par adresse et non par valeur. Les véritables fonctions commencent toutes par le caractère "_". Ce sont ces fonctions qui sont affectées dans les pointeurs de fonction des anneaux.

Nous avons déjà vu les différents types d'objets (figure 2.3). Les opérations possibles sont par exemple l'addition, la soustraction, la négation, la multiplication, l'inversion... Ces fonctions sont trop nombreuses pour être énumérées ici, mais la documentation de la librairie les rassemble. Nous décrivons par ailleurs dans les chapitres qui suivent les opérations arithmétiques que nous avons plus particulièrement soignées. Ce qui nous amène à décrire les différents types d'arithmétiques énumérés figure 2.6. Chacun de ces types est défini dans un fichier descriptif à inclure. Mais en fait, pour la plupart des algorithmes, il suffira d'inclure le fichier zen.h et d'utiliser les fonctions de type ZEN.

I.2.3.d Extensions

L'avantage de ZEN est de permettre l'utilisation aisée des extensions polynomiales. En effet, la structure de la librairie sur ce point est essentiellement récursive. Ceci permet donc d'utiliser les opérations polynomiales de l'anneau sous-jacent comme base des fonctions sur les éléments de l'extension. Ainsi, si l'on souhaite définir sur l'anneau de la section I.2.3.b une extension $\mathbb{Z}/_{n\mathbb{Z}}[X]/_{P(X)}$, il suffit dès le premier anneau créé de déclarer un polynôme et une deuxième variable d'anneau (ZENRing R2 ZENPoly P). On définit ensuite le polynôme voulu par les commandes *ad hoc*. L'obtention de l'extension proprement dite se fait par la commande ZENRingExtAlloc(R2,P,R). Cette construction peut se répéter autant de fois que nécessaire. Bien entendu, l'augmentation du niveau de l'extension diminue la rapidité des calculs. Il est ainsi infiniment plus rapide et efficace de définir $GF(16)$ comme une extension de degré 4 sur $GF(2)$ que comme une double extension de degré 2 sur ce même corps.

I.2.3.e Arithmétiques

Les fonctions arithmétiques dans un ZENRing sont généralement répertoriées sous forme de pointeurs de fonction dans la structure d'anneau. À l'initialisation de l'anneau, ces pointeurs se voient affectés avec les fonctions correspondant à l'anneau demandé. D'un point de vue théorique, deux types d'anneaux suffiraient pour décrire tous les anneaux envisagés :

- le type d'anneau $\mathbb{Z}/_{n\mathbb{Z}}$;
- les extensions sur les anneaux définis récursivement.

Ceci conduirait à deux jeux de fonctions qui seraient utilisés selon le type d'anneau. En pratique, une telle construction n'est pas efficace. Il convient de distinguer des cas particuliers pour lesquels une programmation spéciale est intéressante. Le premier cas particulier est bien entendu $GF(2)$. La figure 2.6 décrit les différentes exceptions dont nous avons tenu compte. À ce titre, la structure adoptée par ZEN est intéressante car elle permet une grande souplesse. Rien n'empêche en effet de programmer au coup par coup quelques fonctions pour un cas particulier donné et de laisser les autres pointeurs sur des fonctions générales moins performantes. Cette conception modulaire permet d'envisager des développements futurs de la librairie au fur et à mesure des besoins.

I.2.4 Optimisations des calculs

DANS BEAUCOUP DE CAS, les fonctionnalités qui précèdent seront suffisantes pour une première programmation. Mais, en particulier en cryptographie, on peut être amené à exiger des opérations de calcul qu'elles soient les plus rapides possibles. Nous présentons donc maintenant les possibilités qu'offre la librairie dans cette optique.

I.2.4.a Précalculs

Nous n'avons pas encore évoqué l'un des principes de ZEN qui est de n'effectuer pour chaque opération que le strict minimum de ce qui est exigible pour celle-ci. En particulier, il est souvent possible d'accélérer des calculs ultérieurs au prix de l'exécution de précalculs. Cette opération n'est rentable que si les opérations ultérieures sont effectuées en grand nombre. C'est pourquoi il a été adopté dans ZEN la stratégie suivante :

- lors de la création d'un anneau, aucun précalcul n'est effectué ;
- il existe par contre, une structure de définition des précalculs (ZENPrc Prc) qui permet de préciser ses besoins à l'aide des fonctions prévues (ZENPrcSetAll(Prc) permet par exemple d'activer tous les précalculs). Il est ensuite possible de demander l'exécution de ces précalculs par la commande ZENRingAddPrc(R,Prc).

Les différents précalculs possibles sont décrits figure 2.7.

Type	Fichier à inclure	Description
ZEN	zen.h	C'est le type général. Toutes les fonctions qui commencent par ce préfixe peuvent être utilisées avec n'importe quel anneau. Ce sont le plus souvent des macros qui déréférencient le pointeur de fonction correspondant dans l'anneau. Ce peuvent être aussi des fonctions écrites de manière générique uniquement à partir d'autres fonctions de ce type. Pour la plupart des utilisateurs de la librairie, ce type restera le seul visible dans leurs programmes. Il faut noter que dans certains des types qui suivent, des fonctions de type général peuvent néanmoins être affectées dans les pointeurs de fonction. En effet, dans chacune de ces arithmétiques, seules les fonctions sur les éléments sont toujours particulières au type. Une fois ces opérations définies, les autres opérations (sur les polynômes, les matrices, etc...) sont soit des opérations génériques de type ZEN, soit des opérations spécialement optimisées pour ce type d'arithmétique.
Ze2	ze2.h	Le corps $GF(2)$ est à la fois très simple de par ses opérations élémentaires (xor et and) et très complexe dès lors que le besoin de calculer efficacement se fait sentir. En effet, lorsqu'on calcule sur des polynômes ou des matrices à coefficients dans $GF(2)$, il ne peut être question de stocker chaque élément sur un entier machine. Les structures de données utilisées dans l'arithmétique de $GF(2)$ sont donc très particulières. De ce fait, la totalité des opérations sur les polynômes et les matrices est particulière à cette arithmétique. Un utilisateur averti qui ne travaillerait que dans ce corps pourrait donc utiliser les fonctions préfixées par Ze2. Dans la plupart des cas, il est néanmoins préférable d'utiliser les fonctions de type général.
Zep	zep.h	Cette arithmétique correspond au cas général de $\mathbb{Z}/_{n\mathbb{Z}}$. Les opérations élémentaires sont basées sur les fonctions de BIGNUM ou sur des modifications de celles-ci (voir chapitre I.3). Chaque élément est en particulier représenté par un pointeur sur un tableau de BigNumDigits dont la taille est fixée par le module n déclaré à l'initialisation de l'anneau.
Zeps	zeps.h	Cette arithmétique dérive de la précédente. Elle optimise les opérations et diminue la mémoire utilisée lorsque le module utilisé tient sur un entier machine.
Zext	zext.h	Cette arithmétique correspond à une extension polynomiale définie sur un anneau. Ses opération utilisent les fonctions polynomiales de l'anneau sous-jacent (voir § I.2.3.d).
Zem	zem.h	Dans les anneaux premiers de module impair, l'utilisation de la représentation de Montgomery permet une amélioration des performances. Cette arithmétique est obtenue par clonage (voir § I.2.4.b).
Zetab	zetab.h	Dans certains anneaux, la tabulation complète des opérations est possible. On peut ainsi grandement accélérer les calculs. Comme la tabulation est une opérations longue, elle n'est pas activée par défaut (voir § I.2.4.b).
Zelog	zelog.h	Dans certain corps finis, la tabulation des opérations par des logarithmes dans la base d'un élément générateur est possible. La même remarque que ci-dessus s'applique (voir § I.2.4.b).
Zec	zec.h	Cette arithmétique permet l'utilisation de la représentation issue du théorème des restes chinois (voir § I.2.4.b.b).
Zef	zef.h	Cette arithmétique permet à titre expérimental de manipuler des éléments de $\mathbb{Q}$. La librairie ZEN n'ayant pas été conçue à l'origine pour travailler dans des ensembles infinis, cette arithmétique ne doit être considérée que comme une facilité offerte aux utilisateurs. En particulier, l'efficacité des calculs sur les fractions n'est pas assurée.
ZBN	zbn.h	Il ne s'agit pas ici d'un type d'arithmétique particulier, mais de l'ensemble des fonctions sur les grands entiers au format BIGNUM. Toutes les fonctions de BIGNUM sont renommées à l'aide du préfixe ZBN et nous y avons ajouté un certains nombres d'opérations, en particulier sur les entrées-sorties (voir § I.3.1).

FIG. 2.6 – *Arithmétiques de* ZEN

Précalcul	Description
ZENPRC_FINITE_FIELD	Ce drapeau permet de signaler à la librairie que l'anneau est un corps fini. Il n'induit aucun précalcul mais permet dans les précalculs correspondant aux autres drapeaux ci-dessous, de tirer profit de cette information.
ZENPRC_ELT_MULTIPLY	Accélération des multiplications dans les corps finis, en particulier par l'utilisation de l'algorithme de Karatsuba.
ZENPRC_ELT_EXP	Accélération des exponentiations. En particulier, dans le cas d'un corps fini, réduction de l'exposant modulo $q-1$, avec q la cardinalité du corps.
ZENPRC_TRACE	Accélération du calcul des traces.
ZENPRC_POLY_ROOTS_CANONICAL	Accélération du calcul des racines d'un polynôme dans les corps finis.

FIG. 2.7 – *Précalculs disponibles*

Clone	Description
ZENCLN_LOG	Lorsque le cardinal d'un corps fini est inférieur à 2^{16}, les éléments sont représentés par leur logarithme dans la base d'un élément générateur.
ZENCLN_TABULATE	Lorsque le cardinal d'un anneau est inférieur à 2^8, les éléments sont représentés par un index codé sur 8 bits. Toutes les opérations de base (addition, multiplication, inversion, etc...) sont tabulées.
ZENCLN_MONTGOMERY	Dans les anneaux premiers de module impair, les éléments sont représentés par leurs N-résidus et la réduction de Montgomery est utilisée.

FIG. 2.8 – *Clones disponibles*

Taille (bits)	128	256	512	768	1024
ZEN(anneau standard)	0.027	0.13	0.76	2.27	5.0
ZEN(clone+ précalculs)	0.013	0.08	0.53	1.65	3.8
ZEN(restes chinois)	0.006	0.028	0.16	0.48	1.07

FIG. 2.9 – *Exponentielle modulaire* $a^b \bmod pq$ *sur sparc II (temps donné en secondes)*

I.2.4.b Clones

I.2.4.b.a Clonage d'un ZENRing

Les précalculs ne modifient pas la représentation des éléments d'un anneau. Il est donc possible de les réaliser et d'utiliser par la suite directement les éléments précédemments obtenus.

Dans certains cas, on peut améliorer la rapidité des calculs en modifiant la représentation des éléments d'un anneau. Typiquement, lorsque l'anneau est de cardinalité suffisamment faible, il est possible d'entièrement tabuler les opérations arithmétiques. Les éléments d'un anneau sont alors représentés par un index dans un tableau. La représentation des éléments étant différentes, on peut parler de clone de l'anneau initial. Le même principe que pour les précalculs a été utilisé : à l'aide d'une structure de définition des clones (ZENCln Cln) et des fonctions prévues (ZENClnSetAll(Cln) permet par exemple d'obtenir le clone optimal), il est possible d'obtenir un clone par la commande R2=ZENRingClone(R,Cln). Les différents clones possibles sont décrits figure 2.8.

I.2.4.b.b Restes chinois

Les deux théorèmes des restes chinois (62 et 63) sont utilisables pour combiner des anneaux à l'aide de la fonction ZENRingChinese(R,N,PR). La représentation des éléments est celle des N-uplets de ZENElts sur les anneaux sous-jacents PR[0],...,PR[N-1]. La structure d'anneau R est donc le produit des anneaux sous-jacents. Lorsque ces derniers sont des corps finis, l'utilisation conjointe pour chacun d'entre eux des drapeaux de précalculs ZENPRC_FINITE_FIELD et ZENPRC_ELT_EXP permet par exemple d'obtenir des exponentiations modulaires beaucoup plus rapides. La figure 2.9 montre un exemple d'optimisation par ces outils de ZEN.

I.2.5 Gestion des erreurs

LA GESTION DES ERREURS est importante dans la librairie ZEN. En effet, comme les structures utilisées peuvent être des anneaux, le calcul d'un inverse peut se révéler impossible. Il faut donc que les opérations retournent dans ce cas une valeur indiquant cette exception. Il est bien précisé dans la documentation de la librairie les valeurs retournées par chaque fonction. En principe, on peut observer les retours suivants :

- 0 indique que la procédure s'est correctement déroulée.
- ZENERR ou ZENNULL selon le type retourné par la fonction, indique une erreur. La variable externe zen_error est alors affectée avec des indications sur la provenance de l'erreur. Un appel à la fonction ZENError() permet d'afficher un message succint correspondant à ces indications (type d'arithmétique où s'est produite l'erreur, fonction appelée, type de l'erreur).
- ZEN_NO_INVERSE indique qu'une opération a entraîné l'inversion d'un élément non inversible. Dans ce cas, l'élément en question est stocké dans l'anneau et peut être obtenu grâce à ZENRingFact(R). Si l'anneau est modulaire, il s'agit d'un facteur du module. S'il s'agit d'une extension, cet élément est soit un polynôme facteur du polynôme de définition, soit un facteur d'un module sous-jacent.
- D'autres valeurs peuvent être retournées par des fonctions particulières de la librairie. Nous aurons l'occasion de voir des fonctions de ce type dans le chapitre I.4.

Sur une idée de Philippe Hoogvorst, une autre possibilité est offerte pour faciliter le déverminage des programmes. Les retours de fonctions ci-dessus sont en fait encapsulés dans deux fonctions spéciales _ZENERR()

et _ZENNULL(). Il est donc possible, lorsqu'on compile ses programmes avec l'option de déverminage activée, et en utilisant un programme de déverminage, de placer un point d'arrêt dans ces fonctions. Il est ainsi plus facile de remonter la pile d'exécution du programme jusqu'à l'erreur rencontrée.

Chapitre I.3

Opérations sur les éléments

"Deux et deux, quatre.
Quatre et quatre, huit.
Huit et huit font seize.
Et seize et seize, que font-ils ?
Ils ne font rien seize et seize,
Et surtout pas trente-deux !"

Jacques Prévert
Le cancre

Nous présentons dans ce chapitre quelques uns des travaux d'implantation que nous avons réalisés dans la librairie Zen. Nous commençons par quelques unes des routines travaillant sur les grands entiers (I.3.1). Ces routines sont un complément à celles déjà en place dans la librairie BigNum. Nous nous attardons plus particulièrement sur notre implantation de l'algorithme de Karatsuba pour la multiplication des grands entiers (I.3.1.b.c), et sur les performances obtenues. Nous nous intéressons ensuite aux opérations modulaires (I.3.2), et plus particulièrement à l'exponentielle. Nous présentons notamment les résultats obtenus par notre implantation de la réduction de Montgomery. Enfin nous évoquons la possibilité de définir un ZENRing représentant le corps des fractions $\mathbb{Q}$ (I.3.3).

I.3.1 Opérations sur les entiers naturels

Les opérations les plus utilisées sont celles qui doivent être les plus efficaces. À ce titre, les opérations sur les grands entiers sont tout particulièrement à soigner. La librairie Zen est basée sur ce point sur la librairie BigNum [7-SVH89]. En effet, cette librairie, désormais éprouvée, dispose de noyaux assembleurs pour les opérations arithmétiques de base sur les grands entiers. Nous ne présentons donc dans la suite que les opérations qui lui ont été ajoutées.

I.3.1.a Entrées-sorties

La librairie BigNum ne dispose pas initialement de routines d'entrées-sorties pour les grands entiers au format BigNum. Nous avons donc ajouté un certain nombre de fonctions de ce type qui sont utilisées à un plus haut niveau par les fonctions de Zen :

- ZBNPrintToString(n,nl,base) retourne une chaîne de caractères représentant l'entier n dans la base indiquée.
- ZBNReadFromStringLazy(n,nl,s,base) lit au contraire la chaîne de caractères dans la base indiquée et affecte l'entier n préalablement alloué. Cette fonction ne peut être utilisée que lorsque la taille de l'entier lu est connue à l'avance, ce qui est le cas dans un anneau modulaire.
- ZBNReadFromString(&n,&nl,s,base) réalise la même opération mais en allouant l'entier à la bonne taille, qui peut donc ne pas être connue à l'avance, mais déterminée par la chaîne.

- ZBNPutToString(n,nl) retourne une chaîne de caractères représentant l'entier dans un format interne beaucoup plus compact.
- ZBNGetFromString(&n,&nl,s) inverse la fonction précédente.
- ZBNPutToStringLazy(n,nl) effectue la même opération en omettant la taille de l'entier, qui doit donc être connue par un autre moyen lors de la relecture.
- ZBNGetFromStringLazy(n,nl,s) inverse la fonction précédente.
- Les fonctions précédentes existent dans un format analogue pour l'écriture et la lecture de fichiers FILE *.

I.3.1.b Multiplication des grands entiers

I.3.1.b.a Décalages

Les décalages à droite et à gauche d'un grand entier ne sont possibles dans BIGNUM que lorsque le nombre de bits de glissement est inférieur à la taille d'un entier machine. Nous avons donc complété ces fonctions qui correspondent à des multiplications par les puissances de 2 et nous sont donc utiles :

- ZBNAnyShiftLeft(r,n,nl,shift) effectue un glissement à gauche de shift bits. Le résultat est placé dans r.
- ZBNAnyShiftRight(n,nl,shift) effectue un glissement à droite de shift bits.

Le format de ces deux fonctions est légèrement différent car le résultat r de la première procédure est normalement plus long que n, alors que dans le deuxième cas, l'opération peut s'effectuer en place. En fait, il est possible d'effectuer même dans le premier cas l'opération en place, à condition que l'allocation de n soit suffisante (on invoque alors ZBNAnyShiftLeft(n,n,nl,shift)).

I.3.1.b.b Algorithme standard

L'algorithme standard de multiplication est implanté dans la librairie BIGNUM. On y trouve aussi l'accélération classique pour le carré d'un nombre utilisant la symétrie de l'opération. Mais tous ces algorithmes sont regroupés dans une même fonction qui teste à chaque appel quel algorithme doit être utilisé. À plus haut niveau dans la librairie ZEN, nous savons *a priori* quel algorithme doit être utilisé. C'est pourquoi nous avons éclaté les algorithmes correspondant en trois fonctions :

- ZBNMultiply(p,pl,m,ml,n,nl) multiplie deux grands entiers par l'algorithme classique.
- ZBNLongSquare(p,pl,n,nl) élève au carré un grand entier en utilisant l'optimisation de la symétrie.
- ZBNSmallSquare(p,pl,n,nl) effectue un carré classique. En effet, lorsque l'entier est relativement petit, le carré classique est plus rapide car plus simple que le précédent.

Soyons clair, ces trois fonctions sont simplement reprises de la libraire BIGNUM. Il n'en est pas de même des suivantes.

I.3.1.b.c Algorithme de Karatsuba

Principe

L'algorithme de Karatsuba [2-Knu81b][page 258–259] est un algorithme très classique du type diviser pour régner. Il permet la multiplication de deux grands entiers, mais s'applique aussi à la multiplication des polynômes.

Soient trois entiers a, b et N tels que N soit inférieur à a et b. Alors, par division euclidienne on peut écrire :

$$\begin{aligned} a &= a_h N + a_b, \\ b &= b_h N + b_b. \end{aligned}$$

Les entiers a_h et b_h correspondent aux bits de haut degré de a et b. Inversement, les entiers a_b et b_b correspondent aux bits de bas degré de a et b. Dans l'algorithme classique de multiplication on effectue

$$\begin{aligned} ab &= (a_h N + a_b)(b_h N + b_b), \\ &= a_h b_h N^2 + (a_b b_h + a_h b_b)N + a_b b_b. \end{aligned}$$

Ceci conduit à quatre multiplications et trois additions, la multiplication par N pouvant être considérée instantanée si N est une puissance de 2 bien choisie. L'algorithme de Karatsuba repose sur l'observation suivante :

$$\begin{aligned} ab &= (a_h N + a_b)(b_h N + b_b), \\ &= a_h b_h N^2 + (a_b b_h + a_h b_b)N + a_b b_b, \\ &= a_h b_h N^2 + ((a_h - a_b)(b_b - b_h) + a_h b_h + a_b b_b)N + a_b b_b. \end{aligned}$$

Cette observation a de l'intérêt car on est alors conduit à 3 multiplications, 2 additions et 4 soustractions. Comme la soustraction et l'addition sont des opérations quasi-identiques, on en déduit aisément que l'algorithme de Karatsuba ne pourra être rentable que si une multiplication est au moins trois fois plus lente qu'une addition.

Il faut néanmoins noter que l'algorithme de Karatsuba est essentiellement récursif. En effet, $\frac{2}{3}$ des multiplications sont utilisées deux fois. Il nécessite aussi l'emploi d'un tampon pour stocker les résultats intermédiaires. Il apparaît donc que l'emploi de l'algorithme doit se limiter à de très grands nombres. En outre, lorsque la récurrence atteint des nombres trop courts, on a tout intérêt à reprendre l'algorithme classique. Ceci est d'autant plus vrai, dans notre cas, que la multiplication classique est en assembleur alors que la programmation de cet algorithme est en C.

Choix retenus

Pour la programmation de cet algorithme, deux options se présentent. On peut chercher à optimiser la mémoire requise par l'opération, c'est-à-dire diminuer au maximum la taille du tampon requis ; c'est cette voie qui a été choisie dans [7-Zim87]. Au contraire, constatant que désormais les machines disposent de plus en plus de mémoires conséquentes, nous avons volontairement utilisé au maximum la mémoire pour accélérer les calculs. C'est sans doute pour cette raison que contrairement à Zimmermann, nous obtenons une accélération de la multiplication d'entiers.

D'autre part, il est clair que c'est le nombre de mots machines utilisés pour représenter l'entier qui détermine quel algorithme doit être utilisé. Certains auteurs [9-LMS95] utilisent alors comme valeur de N la moitié de la plus grande puissance de 2 inférieure au nombre de blocs du grand entier. Pour l'éventuel reliquat (tous les entiers ne sont pas représentés par une puissance de 2^s bits !), l'algorithme classique est alors utilisé. Ceci conduit à une détérioration des performances de l'algorithme lorsqu'on s'éloigne des cas idéaux. Nous avons préféré utiliser au maximum les performances de l'algorithme :

- Lorsque deux entiers de taille différentes sont multipliés nous effectuons

  $$a_b(b_h N + b_b) = a_b b_h N + a_b b_b.$$

  La quantité $a_b b_h$ peut encore être déséquilibrée, mais elle est très souvent de taille suffisamment petite pour être effectuée par l'algorithme classique.

- Considérons maintenant la partie équilibrée $a_b b_b$. Si la longueur de ces entiers est paire, l'algorithme se poursuit sans problème. Sinon, on obtient

  $$\begin{aligned} ab &= (a_b + N a_h + N^2 a_r)(b_b + N b_h + N^2 b_r), \\ &= (a_b + N a_h)(b_b + N b_h) + N^2(a_r b + b_r(a_b + N a_h)). \end{aligned}$$

  Les reliquats a_r et b_r étant représentés par un entier machine, la correction à apporter est calculée par l'opération de BIGNUM ZBNMultiplyDigit écrite entièrement en assembleur. Cette stratégie permet d'obtenir un comportement "lisse" de l'algorithme (voir figure 3.1).

Optimisation

Nous l'avons déjà souligné, il est très difficile d'obtenir des programmes portables. C'est encore plus vrai lorsque la recherche de performances est le but recherché. En effet, telle astuce qui optimisera les opérations sur une machine, ralentira au contraire le même calcul sur une autre architecture. Ceci est particulièrement sensible avec la multiplication de Karatsuba qui est à la fois une opération courte (chaque multiplication ne prend que quelques mili-secondes) et complexe puisque récursive.

Pour tenter de contourner la difficulté, chaque portion de l'opération a donc été programmée, soit entièrement en C, soit en reprenant les fonctions de BIGNUM. Une série de 12 drapeaux de compilation permet de sélectionner le meilleur compromis. Pour déterminer celui-ci, toutes les combinaisons ont été testées sur un certain nombre d'architectures. Le test consistait à effectuer un grand nombre de multiplications aléatoires sur 512, 768 et 1024 bits en utilisant l'algorithme de Karatsuba récursivement jusqu'à la multiplication de deux grands entiers. Pour accélérer la procédure, les dernières étapes de la récursion ont de plus été écrites à part, ceci afin d'éviter l'emploi de boucles là où elles ne s'imposaient plus. La multiplication de Karatsuba de deux nombres de 4 entiers machines est ainsi écrite sans boucle.

D'autre part, une fois cette optimisation effectuée, il convenait de déterminer la limite en deça de laquelle la multiplication de Karatsuba devait être abandonnée au profit de l'algorithme classique. Cette détermination s'effectue aussi empiriquement.

Performances

Les résultats obtenus sur les différentes architectures permettent de conclure sur différents points :

1. Les optimisations par architecture sont très différentes. Ainsi, sur une machine sous solaris, on aura intérêt à ne pas utiliser les fonctions de BIGNUM dans la multiplication de Karatsuba. Par contre, sur une hp, l'utilisation des routines assembleur s'avère plus efficace. Le problème est que les variations de performance sont très sensible !

2. Dans tous les cas, par contre, on a pu observer que la limite à fixer pour le passage à la multiplication classique était de 8 blocs. Ceci correspond à la multiplication de deux nombres de 256 bits sur une machine 32 bits. L'effet de la multiplication de Karatsuba devient sensible dès qu'on multiplie deux nombres de plus de 16 entiers machines (soit 512 bits sur une machine comme la sparc II, voir figure 3.1).

3. Pour ce qui est du carré, le même phénomène se produit mais pour des nombres un peu plus grands encore. En effet, le carré étant plus rapide que la multiplication, les bénéfices de la stratégie diviser pour régner compensent moins rapidement les pertes dues à l'utilisation de la récursion.

4. Fort heureusement, les optimisations de la programmation deviennent moins sensibles dès que la limite est activée. En effet, l'application de cette limite implique que le temps de calcul relatif passé dans les procédures de type Karatsuba est moindre par rapport au temps total de calcul.

Utilisation dans la librairie ZEN

La multiplication de Karatsuba sur les entiers est activée dans ZEN par le drapeau de précalcul ZENPRC_ELT_KARATSUBA (voir § I.2.4.a). Pour l'anneau considéré, l'activation consiste alors à tester empiriquement les algorithmes de multiplication et de carré, et à sélectionner le plus efficace des deux algorithmes. Comme dit plus haut, la multiplication de Karatsuba ne sera donc choisie, le plus souvent, que pour des anneaux premiers dont la taille du module sera supérieure à 512 bits (voire 1024 bits sur une DEC Alpha dont le processeur est à 64 bits).

I.3.2 Opérations modulaires

NOUS VENONS D'ÉVOQUER le niveau le plus bas des opérations utilisées dans la librairie ZEN. Ces opérations sur les entiers sont directement utilisées par les opérations sur les éléments dans un anneau premier. Il s'y ajoute bien entendu la réduction modulaire. Toutes les opérations sur les éléments réalisent en effet l'opération qui les concerne sur les entiers avant d'effectuer la réduction par rapport au module de

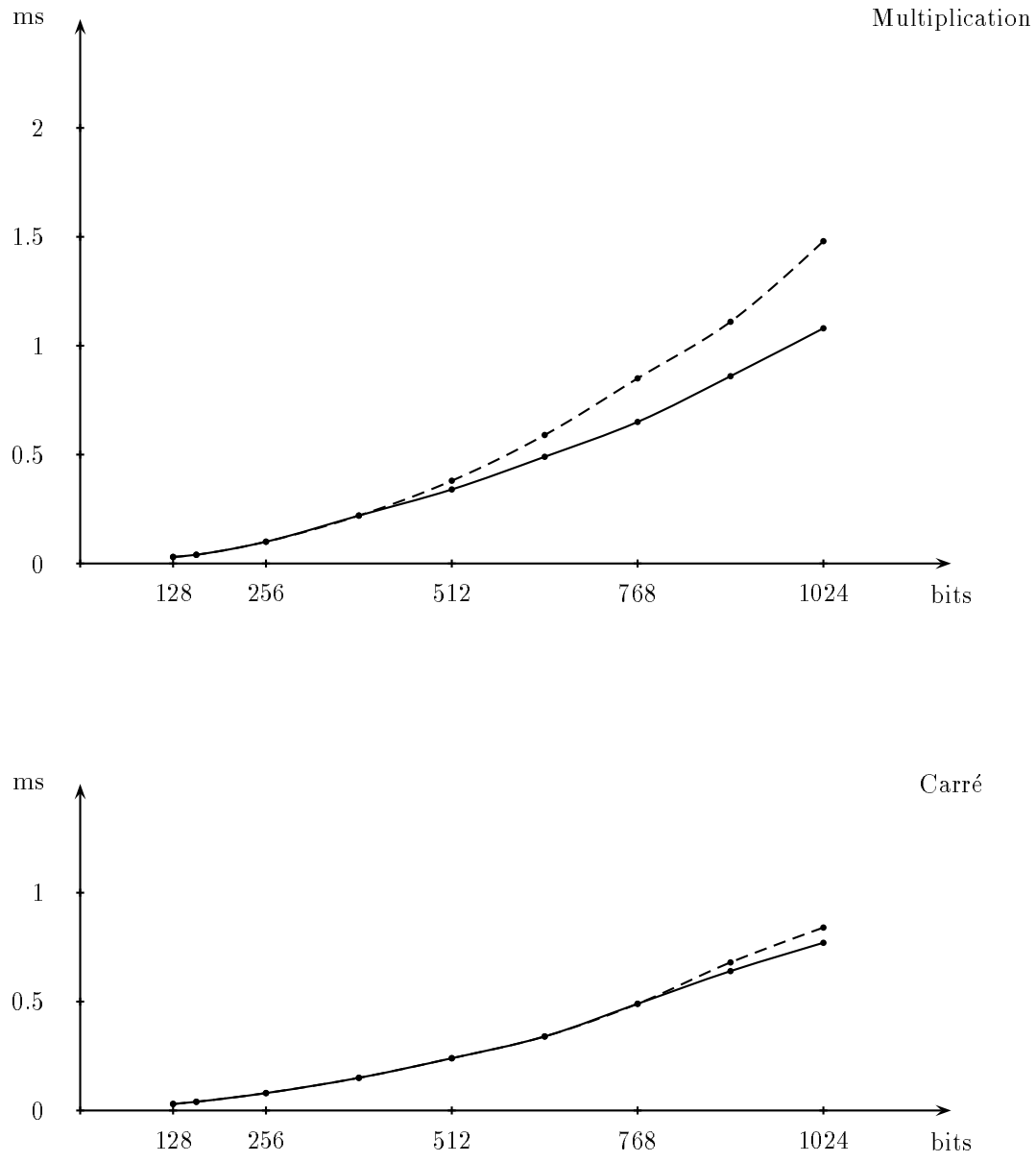


FIG. 3.1 – *Multiplication et carré sur sparc II : méthodes classique en pointillés et algorithme de Karatsuba en gras*

définition. Cette réduction peut s'effectuer par une simple division euclidienne. Dans BIGNUM, cette opération est partiellement écrite en assembleur. Comme pour la multiplication, il est donc difficile d'améliorer les performances de la réduction modulaire.

Concernant, l'inversion d'un élément, celle-ci peut être effectuée en utilisant le classique algorithme d'Euclide. Reynald Lercier a amélioré cette inversion en implantant un algorithme un peu plus efficace, l'algorithme de Lehmer. Une opération couramment utilisée dans les corps finis est l'exponentielle modulaire. Nous allons maintenant voir comment accélérer cette opération.

I.3.2.a Exponentielle

L'exponentielle est une opération cruciale dans beaucoup d'algorithmes. Elle peut néanmoins être écrite de manière générique en utilisant les opérations ZENEltSquare et ZENEltMultiply. La méthode de calcul, classique, est due à Legendre. On l'appelle aussi méthode binaire ou *square and multiply*. Elle est décrite en particulier dans [2-Knu81b, page 399].

La description par l'exemple est sans doute la plus évocatrice. Soit à calculer x^{27}. La représentation binaire de 27 est 2^4+2^3+2+1. Alors x^{27} peut se calculer en notant que

$$\begin{aligned} x^{27} &= x^{2^4+2^3+2+1} \\ &= x x^2 x^{2^3} x^{2^4} \\ &= \left(\left((x^2 \times x)^2\right)^2 \times x\right)^2 \times x. \end{aligned}$$

L'algorithme qui en découle est simple. Parcourant la représentation binaire de l'exposant, on effectue à chaque itération un carré, et si le bit de l'exposant vaut 1 une multiplication par x.

Une généralisation simple consiste à parcourir l'exposant par petits blocs de 4, 5, 6 bits, voire plus. Il convient alors de précalculer les x^k avec k les 2^4, 2^5 ou 2^6 différents exposants possibles. On comprend aisément dans ces conditions qu'une telle stratégie ne devient rentable que pour les grands exposants. Si on prend des blocs de 4 bits, il faut calculer d'entrée de jeu 16 exponentielles. La méthode n'est donc rentable que pour un exposant d'au moins $16 \times 4 = 64$ bits. De même pour 5 bits, il faut au moins 160 bits, et pour un bloc de 6 bits au moins 384 bits. La limite passe ensuite à 896 bits.

Cette méthode existant dans BIGMOD, elle a été reprise dans ZEN avec quelques améliorations. Il est en particulier possible de séparer les précalculs de l'exponentielle proprement dite. Ceci permet, si l'on souhaite effectuer plusieurs exponentielles d'un même élément de n'effectuer les précalculs correspondant qu'une seule fois. La méthode binaire par blocs est activée par le drapeau ZENPRC_ELT_EXP (voir § I.2.4.a).

I.3.2.b Réduction de Montgomery

L'idée de Montgomery a été implantée dans la libraire BIGMOD. Elle est reprise dans ZEN de manière différente. La description qui suit du principe de la réduction est toutefois largement inspirée de la documentation [1-MorBM].

Principe

Nous souhaitons dans ce qui suit travailler dans $\mathbb{Z}/N\mathbb{Z}$, avec N un entier impair. Soit R un entier supérieur à N et premier avec N. Alors l'application

$$\begin{array}{rccc} \hat{\phi} & : & \mathbb{Z}/N\mathbb{Z} & \to & \mathbb{Z}/N\mathbb{Z} \\ & & x & \mapsto & \hat{x} = Rx \bmod N \end{array}$$

est évidemment inversible. On appelera $\hat{x}$ le N-résidu de x.

D'après le théorème 37, il existe u et v deux entiers tels que

$$Ru - vN = 1,$$

et il est toujours possible de ramener ces entiers dans les intervalles

$$0 < u < N \text{ et } 0 < v < R.$$

Taille (bits)	128	256	512	768	1024
ZEN(anneau standard)	0.027	0.13	0.76	2.27	5.0
ZEN(clone+ précalculs)	0.013	0.08	0.53	1.65	3.8

FIG. 3.2 – *Exponentielle modulaire $a^b \bmod c$ sur sparc II (temps donné en secondes)*

Théorème 64. *Soit z un entier compris entre 0 et RN, alors l'algorithme suivant calcule $zR^{-1} mod N$.*

1. *On calcule $m = ((z \bmod R)v) \bmod R$.*
2. *Puis, on effectue $t = (z + mN)/R$.*
3. *Si t est supérieur à N, on lui retranche N.*
4. *On retourne t.*

Preuve. On a

$$\begin{aligned} m = zv \bmod R &\Leftrightarrow mN = z(vN) \bmod R, \\ &\Leftrightarrow mN = -z \bmod R, \\ &\Leftrightarrow mN + z = \alpha R, \end{aligned}$$

avec α un entier. Le calcul de t à l'étape 2 fournit donc bien un entier. En outre, $tR = z \bmod N$. Comme $0 \leq z < RN$ et $0 \leq m < R$, on a, après l'étape 2, $0 \leq t < 2N$, et donc la valeur de t retournée après l'étape 3 est bien $zR^{-1} \bmod N$. □

Nous noterons $\hat{\phi}^{-1}$ la procédure précédente.

L'idée de Montgomery est de remplacer les opérations modulaires sur les entiers, par des opérations sur les N-résidus. À première vue, cette idée peut sembler curieuse. En effet, si les opérations telles que l'addition et la soustraction vont rester inchangées, il n'en est pas de même pour la multiplication et la division. En effet, on a

$$\hat{x} \hat{\times} \hat{y} = \hat{\phi}^{-1}(\hat{x} \times \hat{y}),$$

car

$$(xy)R = (xR)(yR)R^{-1}.$$

Et d'autre part,

$$\widehat{(x^{-1})} = (\hat{x})^{-1}\hat{R}$$

car

$$(x^{-1})R = (xR)^{-1}R^2$$

Implantation dans ZEN L'utilisation dans ZEN de la réduction de Montgomery se fait par l'intermédiaire d'un clone. Elle suit la description ci-dessus à l'exception de l'inversion d'un élément qui s'avère plus rapide lorsqu'on effectue :

1. Une réduction de Montgomery qui ramène le N-résidu dans l'anneau modulaire initial.
2. On effectue ensuite l'inversion dans l'anneau modulaire initial.
3. On reconvertit ensuite le résultat sous forme d'un N-résidu.

Performances La figure 3.2 permet de montrer l'amélioration des performances qu'apporte la réduction de Montgomery sur l'exemple classique de l'exponentielle modulaire. Le gain observé va de 25 à 40 %.

Note 65. La réduction de Montgomery ne peut s'appliquer efficacement que pour des modules impairs. Pour en profiter dans le cas de modules pairs, il suffit de se rappeler le théorème chinois 62 (Voir section I.2.4.b.b).

I.3.3 Opérations sur les fractions

L'implantation d'opérations dans $\mathbb{Q}$ s'est avérée nécessaire pour certains calculs de probabilités qui nous sont utiles au chapitre II.2. Nous utilisons en particulier les deux opérations ZefEltFactorial et ZefEltBinomial qui fournissent respectivement la factorielle d'un entier, et le coefficient binomial de deux entiers, ainsi que les routines d'élimination gaussienne sur des matrices à coefficients dans $\mathbb{Q}$.

Cependant, la librairie Zen n'a pas été conçue dans ce but. L'adaptation des opérations dans $\mathbb{Q}$ à notre formalisme est donc forcément imparfaite. Notre implantation n'aura pas là la prétention de se mesurer à des outils très efficaces dans ce domaine comme la librairie Pari [1-Par].

Chapitre I.4

Opérations sur les polynômes

"La France est le seul pays du monde où, si vous ajoutez dix citoyens à dix autres, vous ne faites pas une addition, mais vingt divisions."

Pierre Daninos
Les carnets du Major Thompson

L'implantation des opérations polynomiales dans Zen s'est effectuée progressivement. Ces opérations sont cruciales dans la librairie car elles interviennent à plusieurs niveaux. D'une part en tant qu'opérations sur les polynômes proprement dits. D'autre part en tant qu'opérations sur les éléments lorsque l'anneau utilisé est une extension polynomiale.

Nous avions initialement été amené a programmer les opérations arithmétiques de $GF(2^m)$ (voir section I.2.1.b). Ceci revient en fait à effectuer des opérations polynomiales sur $GF(2)$. Ces routines initiales ont été reprises par Reynald Lercier et incorporées dans le moteur général de Zen avec des améliorations.

D'autre part, nous avons aussi implanté des opérations ayant un lien avec la factorisation des polynômes. Ces opérations sont implantées dans une surcouche de la librairie Zen : Zenfact. La plupart des opérations implantées dans Zenfact sont des opérations de haut niveau qui constituent des applications directes du principe initiateur de Zen, à savoir une programmation générique qui n'a pas à se préocupper du corps fini de définition.

Notre contribution à Zenfact consiste en des routines de test d'irréductibilité de polynômes (I.4.1). Nous présentons d'abord cette fonction et les valeurs qu'elle retourne (I.4.1.a), puis un premier algorithme très simple permettant de tester l'irréductibilité d'un polynôme (I.4.1.b). Nous décrivons ensuite une adaptation de l'algorithme de Berlekamp (I.4.1.c) et nous comparons ces deux algorithmes en terme d'efficacité (I.4.1.d). Nous donnons enfin une application permettant de se faire une idée des performances de l'implantation (I.4.1.e). Nous rappelons enfin quelques références pour le lecteur intéressé par le problème plus général de la factorisation de polynômes (I.4.2).

I.4.1 Irréductibilité

Nous présentons ci-après la fonction ZENPolyIsNotIrreducible, dont nous avons assuré l'implantation. Cette fonction constitue un test d'irréductibilité d'un polynôme. Toutefois, des développements récents dans la programmation de Zenfact pourraient entraîner des modifications dans l'implantation de cette fonction.

I.4.1.a Retours possibles de la fonction

La fonction ZENPolyIsNotIrreducible(P,R) prend en arguments un ZENPoly P défini sur un ZENRing R. Or les algorithmes utilisés ne sont prouvés corrects que dans le cas des corps finis. Comme la librairie Zen permet de travailler dans tout anneau fini, le résultat de cette fonction ne peut donc être garanti que dans

Donnée : Un polynôme $P(X)$ de $GF(q)[X]$ de degré d, où $q = p^m$.
Algorithme :

1. Calcul de la dérivée $P'(X)$.
2. Si $P'(X) = 0$
   $\rightarrow$ ZENPOLY_IS_P_POWER.
3. Calcul du pgcd $G = P \wedge P'$.
4. – Si $G \neq 1$.
   $\rightarrow$ ZENPOLY_HAS_REPEATED_FACTORS.
   – Sinon, le polynôme est prouvé sans facteur répété.

FIG. 4.1 – *Élimination des facteurs répétés*

le cas d'un corps fini. Dans ce cas, la fonction retourne ZENPOLY_IS_IRREDUCIBLE= 0 si et seulement si le polynôme est irréductible. La fonction peut aussi retourner l'une des valeurs non nulles suivantes :

- ZENERR si une erreur se produit durant l'exécution.
- ZEN_NO_INVERSE si un inverse a été impossible à calculer. On n'est donc pas dans le cas d'un corps fini.
- ZENPOLY_HAS_ZERO_ROOT si le polynôme a 0 comme racine triviale.
- ZENPOLY_HAS_ONE_ROOT si le polynôme a 1 comme racine triviale.
- ZENPOLY_HAS_REPEATED_FACTORS si le polynôme a des facteurs répétés.
- ZENPOLY_IS_P_POWER si le polynôme est l'élévation à la puissance caractéristique du corps d'un autre polynôme.
- ZENPOLY_IS_COMPOSED dans les autres cas de réducibilité.

Cette fonction est d'ailleurs un bon exemple des possibilités de la librairie puisqu'elle utilise une extension polynomiale pour effectuer les calculs nécessaires modulo le polynôme P, ainsi que les opérations matricielles.

I.4.1.b Algorithme par défaut

Le tout premier test effectué consiste à vérifier que le polynôme n'a pas 0 ou 1 comme racine triviale, auquel cas la fonction retourne ZENPOLY_HAS_ZERO_ROOT ou ZENPOLY_HAS_ONE_ROOT. Une fois ces tests très rapides effectués, une première phase de vérification de l'existence d'éventuels facteurs répétés est réalisée.

I.4.1.b.a Élimination des facteurs répétés

Il est très facile de voir si un polynôme $P(X)$ possède des facteurs répétés ou non. En effet, un simple pgcd avec sa dérivée en X permet de s'en rendre compte, ce qui s'effectue par l'algorithme d'Euclide. En caractéristique non nulle, une condition supplémentaire est cependant à prévoir :

- Si $P' = 0$, alors $P(X) = (G(X))^p$ où p est la caractéristique du corps. Dans ce cas, la fonction ZENPolyIsNotIrreducible retourne la valeur ZENPOLY_IS_P_POWER.
- Sinon, on effectue le pgcd avec la dérivée et si $P \wedge P' = D$, avec $D \neq 1$ alors on prouve que P/D n'a pas de facteur répété. La fonction retourne alors ZENPOLY_HAS_REPEATED_FACTORS.

Donnée : Un polynôme $P(X)$ de $GF(q)[X]$ de degré d, où $q = p^m$, sans facteur répété.
Algorithme :

1. La valeur j est initialisée à 1. La valeur $Q(x)$ à x.
2. Si $2j > \deg P(x)$ on stoppe
   $\rightarrow$ ZENPOLY_IS_IRREDUCIBLE.
3. Calcul de $(x^{q^j} - x) \bmod P(x)$. Pour cela, on effectue $Q(x) \leftarrow Q(x)^q \bmod P(x)$ puis $G(x) \leftarrow Q(x) - x \bmod P(x)$.
4. Si $G(x) = 0$
   $\rightarrow$ ZENPOLY_IS_COMPOSED.
5. Sinon, calcul de $G(x) \leftarrow G(x) \wedge P(x)$.
6. Si $G(x) \neq 1$
   $\rightarrow$ ZENPOLY_IS_COMPOSED.
7. On incrémente j, et retour à l'étape 2.

FIG. 4.2 – *Crible*

– Si $P \wedge P' = 1$, alors P n'a pas de facteur répété, et on peut appeler un autre algorithme pour vérifier l'irréductibilité.

Ceci nous donne finalement l'algorithme de la figure 4.1.

I.4.1.b.b Crible

Nous sommes maintenant ramenés au cas d'un polynôme dont la décomposition est de la forme

$$P = P_1 \cdots P_k,$$

où les P_k sont des polynômes irréductibles deux à deux distincts. La technique que nous utilisons est celle du crible ; elle est basée sur le théorème suivant :

Théorème 66 [2-LN86, page 91]. *Le polynôme $x^{q^k} - x \in GF(q)[x]$ est le produit de tous les polynômes moniques irréductibles de $GF(q)[x]$ de degré divisant k.*

Ce théorème permet par exemple de savoir facilement si un polynôme a des racines dans $GF(q)$. En effectuant son pgcd avec le polynôme $x^{q^1} - x$, on obtient en effet le produit de monômes $\prod_i x - \alpha_i$ qui divise P. Les α_i sont donc les racines du polynôme P.

Si le polynôme P n'a pas de racine, on peut effectuer le calcul de son pgcd avec $x^{q^2} - x$ et ainsi de suite. Le processus s'arrête donc à l'itération j dès que le degré du polynôme P est strictement inférieur à $2j$, puisque le polynôme est alors forcément irréductible.

Note 67. Les calculs des polynômes $x^{q^j} - x$ sont effectués modulo le polynôme P. Il peut arriver qu'on obtienne le résultat nul. Ceci signifie que $P(X)$ de degré d divise un produit de polynômes de degrés $j \leq d/2 < d$ (les étapes précédentes ont éliminé tous les polynômes de degré inférieur), donc que P est lui même un produit de polynômes de degré j.

Tout ceci conduit finalement à l'algorithme de la figure 4.2.

Le problème de l'algorithme ci-dessus est qu'il impose des opérations d'exponentiation modulo le polynôme P. L'algorithme de Berlekamp que nous décrivons maintenant est une alternative à cet algorithme qui permet de n'effectuer que des multiplications par X modulo le polynôme $P(X)$.

Donnée : Un polynôme $P(X)$ de $GF(q)[X]$ de degré d, sans facteur répété.
Algorithme :

1. Calcul des $X^{iq} \bmod P(X)$ pour i variant de 0 à $d-1$:

$$X^{iq} = \sum_{j=0}^{d-1} b_{i,j} X^j .$$

2. Obtention de la matrice $B - I$, avec

$$B = (b_{i,j}) \begin{array}{lll} 0 \leq & i & \leq d-1 \\ 0 \leq & j & \leq d-1 \end{array}$$

et calcul de son rang r.

3. Si $r = d-1$, le polynôme est irréductible.
→ ZENPOLY_IS_IRREDUCIBLE.

4. Sinon, le polynôme est composé.
→ ZENPOLY_IS_COMPOSED.

FIG. 4.3 – *Algorithme de Berlekamp*

I.4.1.c Algorithme de Berlekamp

L'algorithme de Berlekamp s'applique lui aussi à des polynômes $P(X)$ dont la décomposition est de la forme

$$P = P_1 \cdots P_k ,$$

où les P_k sont des polynômes irréductibles deux à deux distincts. La description de l'algorithme de Berlekamp est donnée dans [2-Knu81b] mais le chapitre 4 de [2-LN86] est beaucoup plus complet en ce qui concerne la factorisation des polynômes définis sur les corps finis.

Théorème 68. *Soient P un polynôme monique sur $GF(q)$ et H un polynôme tel que $H^q = H \bmod P$. Alors*

$$P(X) = \prod_{c \in GF(q)} P(X) \wedge (H(X) - c).$$

Preuve. Les polynômes $H(X) - c$ sont premiers entre eux deux à deux. Il en est donc de même des polynômes $P(X) \wedge (H(X) - c)$ et leur produit divise donc $P(X)$.

D'autre part, $P(X)$ divise $H^q(X) - H(X)$. Or tout élément de $GF(q)$ est racine du polynôme $X^q - X$. On a donc dans $GF(q)$ la relation

$$X^q - X = \prod_{c \in GF(q)} (X - c).$$

Donc $P(X)$ divise $\prod_{c \in GF(q)} H(X) - c$. Le polynôme monique $P(X)$ divise donc le polynôme monique $\prod_{c \in GF(q)} P(X) \wedge (H(X) - c)$. D'où l'égalité. □

Ce théorème permet donc d'envisager une factorisation au moins partielle de P. Il nous faut maintenant trouver un polynôme H tel que la condition $H^q = H \bmod P$ soit vérifiée.

Théorème 69. *Supposons connue la décomposition de P :*

$$P = P_1 \cdots P_k .$$

Alors il existe q^k polynômes H qui vérifient

$$H^q = H \bmod P.$$

Corps	Degré	Algorithme	VM (ms)	RPI (s)	NMC (%)	NMI (%)
$GF(2)$	100	Berlekamp	14	0.6	41	100
		crible	52	4.3	27	0
	1000	Berlekamp	2400	5.9	12	100
		crible	7000	64	88	0
$GF(3)$	10	Berlekamp	4.4	8.8	41	73
		crible	7.6	7.4	18	0
	100	Berlekamp	970	∞	12	100
		crible	930	∞	88	0
$GF(2^{50})$	2	Berlekamp	22	∞	63	64
		crible	28	∞	5	0
	20	Berlekamp	2900	∞	28	100
		crible	4700	∞	72	0
$GF(p)$ $p = 772968761728451$ (50 bits)	2	Berlekamp	20	0.01	41	36
		crible	24	0.01	4	0
	20	Berlekamp	2100	∞	37	100
		crible	5500	∞	63	0

FIG. 4.4 – *Comparaison des tests d'irréductibilité (PC-486 100MHz)*

Preuve. Comme $H^q - H = 0 \bmod P$, P divise $H^q - H = \prod_{c \in GF(q)} (H - c)$, donc tout facteur irréductible P_i divise l'un des $(H - c)$. On a donc pour tout i, $1 \leq i \leq k$, l'existence de $c_i \in GF(q)$ tel que

$$H = c_i \bmod P_i.$$

Inversement, par le théorème 63 des restes chinois, il existe un unique H qui verifie pour tout i, $1 \leq i \leq k$

$$H = c_i \bmod P_i \Rightarrow H^q = c_i \bmod P_i \Rightarrow H^q = H \bmod P_i,$$

ce qui entraîne $H^q = H \bmod P$. Comme il y a exactement q^k jeux de constantes c_i possibles, on en déduit le résultat. □

Il est aisé de déterminer les polynômes H. En effet si d est le degré de P, $H(X) = \sum_{i=0}^{d-1} a_i X^i$, et on peut utiliser les expressions

$$X^{iq} \bmod P(X) = \sum_{j=0}^{d-1} b_{i,j} X^j,$$

pour obtenir

$$\begin{aligned} H(X)^q &= H(X) \bmod P(X), \\ \sum_{i=0}^{d-1} a_i X^{iq} &= \sum_{i=0}^{d-1} a_i X^i \bmod P(X), \\ \sum_{j=0}^{d-1} (\sum_{i=0}^{d-1} b_{i,j} a_i) X^j &= \sum_{i=0}^{d-1} a_i X^i. \end{aligned}$$

On a là un système linéaire de d équations à d inconnues, dont on sait qu'il a q^k solutions. De manière matricielle, si B est la matrice des coefficients $b_{i,j}$, alors la matrice $B - I$ a pour rang $d - k$ [1].

Si le polynôme est irréductible, alors la matrice est de rang $d - 1$ et le système a pour solutions triviales les polynômes constants sur $GF(q)$. On en déduit l'algorithme de la figure 4.3.

1. Nous supposons ici connue la définition du rang d'une matrice. Celle ci apparaît au chapitre II.1.1.b où elle nous sera plus utile.

I.4.1.d Comparaison des algorithmes

Il est possible d'utiliser l'algorithme de crible décrit section I.4.1.b.b en faisant directement appel à la fonction _ZENPolyIsNotIrreducible(P,DDF,R)[2]. Toutefois, dans $GF(2)$ les performances sont moins bonnes que l'algorithme de Berlekamp. Il est cependant malaisé de comparer ces deux algorithmes car contrairement à l'algorithme de Berlekamp, l'algorithme de crible est très instable. Selon le polynôme testé, l'algorithme a en effet un comportement variable en terme de vitesse d'exécution. La forme du polynôme intervient pour beaucoup dans cette grande variabilité d'exécution.

Shoup a introduit dans [4-Sho95] des critères empiriques pour évaluer la rapidité des algorithmes de factorisation. Malheureusement, ces critères ne sont pas forcément adaptés à une fonction de vérification d'irréducibilité. Nous avons donc élaboré plusieurs critères pour comparer les deux algorithmes précédents. Ceux-ci sont uniquement basés sur l'intuition :

1. La vitesse moyenne (VM) qui consiste à choisir aléatoirement N polynômes, à tester leur irréductibilité avec chaque algorithme, et à mesurer le temps moyen de calcul par polynôme. Ces polynômes sont cependant choisis sans facteur répété, et avec une proportion constante de 10 % de polynômes irréductibles.

2. La recherche du polynôme irréductible de plus petit sous-degré (RPI) pour un degré donné (voir section I.4.1.e)[3].

3. Le nombre moyen (NM) de succès de l'algorithme, c'est-à-dire, pour un nombre N d'essais, le pourcentage de polynômes pour lesquels l'algorithme en question a été significativement[4] plus rapide. Ce résultat est donné dans le cas où le polynôme est irréductible (NMI) et dans le cas où il est composé (NMC). Les polynômes testés sont tous sans facteur répété, ni racine triviale.

Les différents résultats sont rassemblés figure 4.4. Ils montrent que l'algorithme de Berlekamp peut se comparer au crible même dans des corps de grande taille. Toutefois, ces résultats dépendent beaucoup du type de polynôme vérifié. Si ces polynômes sont le plus souvent irréductibles, alors le test à utiliser est sans conteste celui utilisant l'algorithme de Berlekamp, car le crible s'avère excessivement lent dans ce cas particulier. Si au contraire les polynômes sont le plus souvent composés, alors le crible permet d'en éliminer beaucoup très rapidement. Une meilleure solution consisterait sans doute à débuter par quelques itérations du crible, puis à utiliser l'algorithme de Berlekamp lorsque ces premiers tests sont passés. Une étude plus poussée sera nécessaire pour optimiser cette fonction. Pour le moment, l'algorithme de Berlekamp, du fait qu'il présente un temps de calcul stable, est utilisé par défaut.

I.4.1.e Application

Dans le cas des corps $GF(2^m)$, une méthode d'optimisation simple des opérations consiste à utiliser comme polynôme sur $GF(2)$ définissant le corps un polynôme de la forme

$$P(X) = X^m + f(X),$$

où f est un polynôme de très petit degré. Nous appelerons dorénavant *sous-degré de P* le degré du polynôme f.

Il est souhaitable d'avoir le sous-degré de P inférieur strictement à 16. En effet, on peut dans ce cas effectuer de manière beaucoup plus rapide la réduction par le polynôme P.

Dans le cas d'un corps, la question peut alors se poser de savoir dans quelle mesure un tel polynôme irréductible existe. Nous n'avons pas trouvé de réponse satisfaisante à cette question. Aussi, le plus simple a-t-il été d'essayer de trouver un polynôme de cette forme pour les corps $GF(2^m)$ avec m croissant. L'efficacité de l'algorithme de Berlekamp a permis de trouver de manière exhaustive, pour tous les degrés m de 2 à 3500, le polynôme irréductible sur $GF(2)$ de plus petit sous-degré. Ces résultats sont disponibles sur le réseau Internet [1-Cha95].

2. L'appel par défaut est _ZENPolyIsNotIrreducible(P,Berlekamp,R). Le sigle *DDF* correspond à *Distinct Degree Factorization* [7-FGP96].

3. Lorsque cette recherche est trop coûteuse en temps, le signe ∞ est utilisé.

4. Ceci explique pourquoi la somme des deux nombres ne donne pas toujours 100 %

Sous-degré	Degré	Polynôme
1	2	$1+X+X^2$
2	5	$1+X^2+X^5$
3	-	-
4	8	$1+X+X^3+X^4+X^8$
5	14	$1+X^5+X^{14}$
6	-	-
7	32	$1+X^2+X^3+X^7+X^{32}$
8	85	$1+X+X^2+X^8+X^{85}$
9	149	$1+X+X^3+X^4+X^5+X^6+X^7+X^9+X^{149}$
10	256	$1+X^2+X^5+X^{10}+X^{256}$
11	467	$1+X+X^6+X^{11}+X^{467}$
12	768	$1+X^6+X^7+X^9+X^{10}+X^{12}+X^{768}$
13	1532	$1+X+X^2+X^6+X^8+X^{10}+X^{11}+X^{13}+X^{1532}$
14	2996	$1+X+X^3+X^4+X^5+X^6+X^7+X^8+X^9+X^{10}+X^{11}+X^{14}+X^{2996}$
15	5969 (?)	$1+X^2+X^4+X^6+X^8+X^{10}+X^{11}+X^{15}+X^{5969}$

FIG. 4.5 – *Premières occurences des sous-degrés*

Le tableau 4.5 présente pour chaque sous-degré d inférieur à 14, le premier degré m pour lequel aucun polynôme irréductible de sous-degré inférieur à $d-1$ n'a été trouvé. Pour le sous-degré 15, le point d'interrogation indique que ce contre-exemple peut ne pas être de degré minimum, tous les degrés intermédiaires n'ayant pas été explorés.

Ces résultats sont résumés dans la courbe 4.6 qui permet d'envisager une extrapolation. Ceci conduit à penser que pour tous les corps finis de taille "raisonnable", l'utilisation d'un polynôme irréductible de faible sous-degré est possible.

On peut noter ici que l'utilisation pour ce genre d'étude exhaustive d'un programme de calcul formel comme Maple [3-CGG+90] est exclue. En effet, la simple vérification de l'irréductibilité du polynôme de degré 5969 trouvé par ZEN prend environ 50 minutes.

I.4.2 Factorisation

LES ALGORITHMES QUE NOUS AVONS DÉCRITS ci-dessus sont en fait des adaptations d'algorithmes de factorisation de polynômes. La factorisation d'un polynôme dans un corps fini est en effet un problème largement étudié pour lequel de nombreux algorithmes sont connus. On peut considérer qu'il est résolvable dans tous les cas par des algorithmes en temps temps polynomial, dont les performances dépendent cependant, à la fois de la caractéristique du corps, du degré de l'extension polynomiale si le corps n'est pas premier, et du degré du polynôme à factoriser. Le choix du meilleur algorithme est donc en lui-même un problème ardu, dont la solution passe le plus souvent par l'empirisme [4-Sho95].

La librairie ZEN ne disposait pas jusqu'à présent de fonctions de factorisation de polynômes. Cette lacune est en passe d'être comblée grâce au travail de Reynald Lercier, qui a tiré parti de références bibliographiques fournies par Paul Camion et François Morain que nous remercions.

Il ne saurait être question pour nous de décrire ici les algorithmes de factorisation de polynômes dans les corps finis. Tout au plus pouvons nous indiquer quelques références utiles pour ce problème.

Camion a introduit dans [4-Cam83] les notions suivantes : Soit $P(x)$ un polynôme de décomposition

$$P = P_1 \cdots P_k,$$

où tous les facteurs P_i sont des polynômes irréductibles.

Définition 70. Un polynôme $S(x)$ de $GF(q)[X]$ est un *polynôme factorisant* de $P(X)$, s'il existe i et i', $1 \leq i < i' \leq k$ tels que $S(x) = 0 \bmod P_i(x)$ et $S(x) \neq 0 \bmod P_{i'}(x)$.

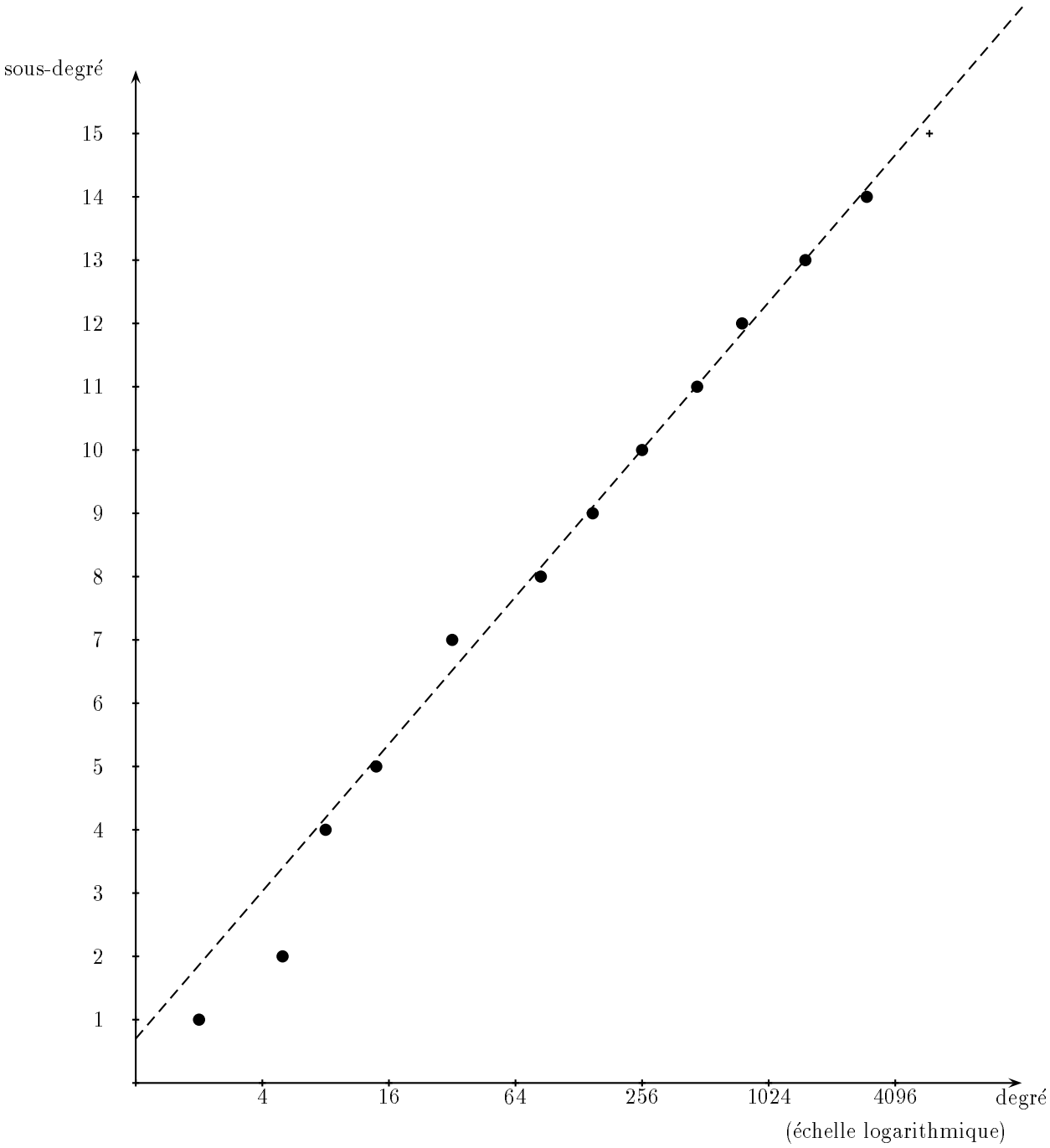


FIG. 4.6 – *Courbe des premières occurences des sous-degrés*

Il est équivalent de dire que le pgcd avec $P(X)$ d'un polynôme factorisant de $P(X)$ est un facteur non trivial de P.

Définition 71. Un polynôme $S(x)$ de $GF(q)[X]$ est un *polynôme séparateur* de $P(X)$, s'il existe $c \in GF(q)$ tel que $S(x) - c$ soit un polynôme factorisant de P.

Définition 72. Un ensemble $\mathcal{S}$ de polynômes de $GF(q)[X]$ est un *ensemble séparateur* (resp. *factorisant*) de $P(X)$, si pour tout couple (i, i'), $1 \leq i < i' \leq k$, il existe un polynôme séparateur (resp. factorisant) $S \in \mathcal{S}$ tel que $S(x) = 0 \bmod P_i(x)$ et $S(x) \neq 0 \bmod P_{i'}(x)$.

La factorisation d'un polynôme P revient donc à déterminer un ensemble factorisant de P.

Dans l'algorithme de Berlekamp [2-LN86, chap. 4] proprement dit, toute base de la matrice $B - I$ est un ensemble séparateur du polynôme. En énumérant les éléments $c \in GF(q)$ on obtient donc un ensemble factorisant de P. Ceci n'est donc applicable que dans les corps de petite taille. Camion décrit aussi dans [4-Cam83] une méthode d'obtention d'ensembles séparateurs à l'aide d'un opérateur trace. Cette méthode a été plus tard reprise par Thiong ly [4-Thi89] pour obtenir, dans une extension $GF(p^m)$ d'un corps premier $GF(p)$, des ensembles séparateurs pour lesquels l'énumération des éléments est limitée à $c \in GF(p)$. L'ensemble de ces algorithmes permet donc la factorisation dans les corps de petite caractéristique.

D'autre part, le crible décrit ci-dessus (cf. I.4.1.b.b) permet de se ramener dans tous les cas à des polynômes dont tous les facteurs irréductibles sont de même degré. En caractéristique différente de 2, l'algorithme probabiliste de Cantor-Zassenhaus [4-CZ81] permet de factoriser les polynômes de ce type, et est simple d'implantation. La complexité asymptotique d'une chaîne complète de factorisation utilisant cet algorithme est par ailleurs étudiée dans [7-FGP96]. Enfin, pour les corps de grande caractéristique, la référence en la matière est l'algorithme probabiliste de Shoup [4-Sho95].

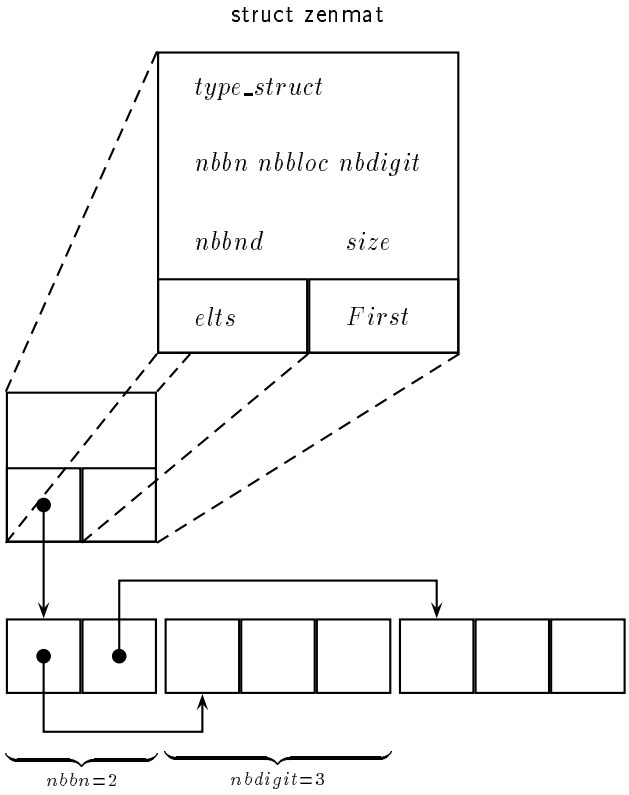


FIG. 5.1 – *Structure de données des matrices*

Chapitre I.5

Opérations sur les matrices

L'implantation des opérations matricielles dans Zen a nécessité de soigner les structures de données utilisées pour représenter les matrices. Nous présentons tout d'abord la structure générale utilisée (I.5.1), et évoquons celle des matrices de permutation. La multiplication de deux matrices peut parfois être accélérée en utilisant l'algorithme de Winograd. Toutefois, l'opération importante au niveau matriciel est celle qui permet l'inversion (I.5.2). Pour nos dimensions, le pivot de Gauss est l'algorithme le plus efficace. Dans le corps binaire, la structure matricielle est forcément différente et nous en étudions les conséquences sur les opérations arithmétiques (I.5.3). Dans d'autres cas particuliers, la structure de données utilisée est légèrement modifiée pour permettre une optimisation de la mémoire nécessaire au stockage d'une matrice (I.5.4). Enfin, lorsque la matrice est définie sur une extension polynomiale, son architecture doit tenir compte du caractère polynomial sous-jacent de ses éléments (I.5.4). Cette dernière structure permet de faciliter la conversion entre des matrices définies sur une extension polynomiale d'un anneau R, et des matrices définies sur R (I.5.5). Une telle conversion est par exemple nécessaire dans l'algorithme que nous décrivons chapitre II.3.

I.5.1 Cas général

Avec les polynômes, les matrices sont des outils très utiles et très utilisés. Il était donc indispensable qu'elles soient implantées dans la librairie Zen de la manière la plus efficace possible. On peut associer aux matrices carrées de taille fixe des structures d'anneaux non commutatifs et non intègres. Cette possibilité n'a pas été utilisée dans Zen car elle est par trop restrictive. Au contraire, toutes les opérations sont envisageables dès que les tailles des matrices sont compatibles.

Nous supposons ici connues toutes les notions classiques concernant les matrices, en particulier les opérations arithmétiques (addition, multiplication), la notion de matrice identité et d'inversibilité d'une matrice carrée. Nous utilisons aussi sans en rappeler la définition les matrices de permutation. La définition du rang d'une matrice est renvoyée au chapitre II.1.1.b.

I.5.1.a Structure de données

Une matrice est un tableau à deux entrées de coefficients. Il semble donc évident d'utiliser ce type de structure de données. Pourtant ce n'est pas cela qui est implanté dans la librairie Zen. En effet, il apparaît plus judicieux d'utiliser une structure de données en deux étages (voir figure 5.1). Chaque ligne de la matrice est représentée par un tableau à une entrée d'éléments, et la matrice est un tableau de lignes. En effet ce type de structure peut permettre de manipuler des lignes plus facilement. En fait, il peut être tout aussi intéressant de manipuler les colonnes de la matrice rapidement. C'est pourquoi les deux types de structure de données coexistent dans la librairie.

Physiquement, les deux types sont indistinguables, c'est donc l'emploi d'un indicateur dans la structure de la matrice qui indique le type de la matrice. Pour les mêmes raisons que précédemment, on utilise des pointeurs de fonctions dans la structure matricielle pour éviter d'avoir à tester dans certaines fonctions le type de la matrice.

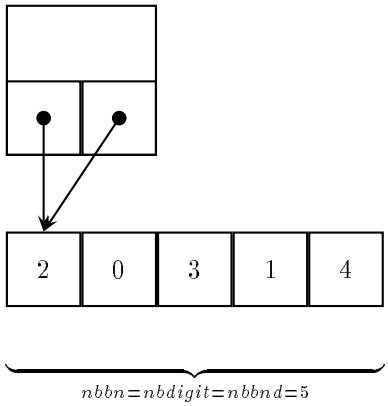


FIG. 5.2 – *Structure de données des matrices de permutation*

I.5.1.a.a Matrices

Pour allouer une matrice dans ZEN il suffit de faire appel à la fonction ZENMatAlloc(M,l,c,R) où l et c sont respectivement les nombres de lignes et de colonnes désirés. Le type de matrice alloué peut être forcé à l'aide des deux autres fonctions ZENMatRowAlloc(M,l,c,R) et ZENMatColAlloc(M,l,c,R). Dans le cas usuel, la matrice est allouée dans le type qui minimise la mémoire utilisée.

Une fois allouée, la matrice peut être utilisée avec les fonctions de la librairie dont la syntaxe est maintenant bien connue. Des fonctions d'extraction et d'affectation des coefficients de la matrice sont bien entendu disponibles.

I.5.1.a.b Permutations

Une matrice de permutation est une matrice très particulière qu'il est souhaitable de pouvoir représenter de manière plus économe qu'une matrice normale. Il est possible de définir des matrices de permutation avec la librairie ZEN. La structure de données utilisée mémorise alors un seul tableau correspondant aux indices permutés des lignes ou des colonnes (voir figure 5.2).

Deux fonctions de création existent donc :

- ZENPermutationRowAlloc(Pl,n,R) alloue une matrice de permutation qui sera considérée comme une matrice de permutation de lignes. La multiplication de cette matrice à droite effectuera donc une permutation de lignes. On voit ici l'intérêt de la structure de données utilisée. En effet si la matrice à permuter est aussi de type ligne, la permutation pourra s'effectuer très rapidement.

- ZENPermutationColAlloc(Pl,n,R) alloue une matrice de permutation qui sera considérée comme une matrice de permutation de colonnes. La multiplication de cette matrice à gauche effectuera donc une permutation de colonnes.

Il faut ici bien préciser que, par la suite, l'utilisation de ces matrices est transparente pour l'utilisateur. Les matrices de permutation peuvent être employées de la même façon que les autres matrices. Seules sont interdites les opérations qui pourraient affecter à une matrice de permutation une matrice de format différent.

I.5.1.b Opérations standard

I.5.1.b.a Opérations linéaires

Les opérations possibles sur les matrices correspondent exactement aux opérations sur les éléments. Le préfixe utilisé est simplement remplacé par ZENMat. Il s'y ajoute l'addition et la multiplication d'un scalaire, c'est-à-dire d'un ZENElt de l'anneau de définition de la matrice. Les algorithmes utilisés sont les algorithmes classiques qui sont le plus souvent linéaires en la taille de la matrice.

I.5.1.b.b Multiplication de Winograd

Lorsque la multiplication de deux éléments est beaucoup plus coûteuse que leur addition, l'algorithme de Winograd [2-Knu81b][pages 426–427] permet d'accélérer la multiplication matricielle. Dans cet algorithme, une mémorisation de calculs intermédiaires permet en effet de diminuer le nombre global de multiplications tout en augmentant le nombre d'additions. Nous omettons ici sa description qui ne présente pas d'intérêt particulier. Notons cependant que cet algorithme utilise la commutativité de la multiplication.

I.5.2 Inversions d'une matrice

Nous abordons ici l'opération sans doute la plus utilisée concernant les matrices. La diagonalisation permet en particulier le calcul de l'inverse d'une matrice ou du noyau d'une application linéaire. Il était donc indispensable de particulièrement soigner son implantation dans Zen.

I.5.2.a Élimination gaussienne

L'algorithme le plus connu pour inverser une matrice est le pivot de Gauss [2-Knu81a][page 304]. Son utilisation est quasi-universelle, et en particulier, le problème du choix du meilleur pivot a donné lieu à une littérature extrêmement fournie. Dans le cas qui nous occupe, nous n'aurons pas à nous préoccuper de ce choix. En effet, les calculs dans les corps finis sont exacts, et il n'y a donc pas de problème de stabilité de l'algorithme. La seule contrainte qui nous reste est donc celle du pivot non nul.

La fonction ZENMatGaussPlain(D,S,P,p_rk,R) est la fonction qui réalise le pivot de Gauss proprement dit. Elle prend en entrée trois matrices D, S et P. La première correspond à la matrice à diagonaliser. La deuxième mémorise les opérations effectuées lors de la diagonalisation. La troisième mémorise les permutations éventuelles nécessaires à la mise en place des pivots successifs.

Si la matrice D est de type ligne, la diagonalisation s'effectue par des combinaisons linéaires sur les lignes, et inversement si elle est de type colonne. La deuxième matrice doit être de même type et correspond aux opérations effectuées sur les lignes ou colonnes de la matrice initiale. La troisième doit être de type opposé et correspond aux opérations de permutation sur les colonnes ou les lignes de la matrice.

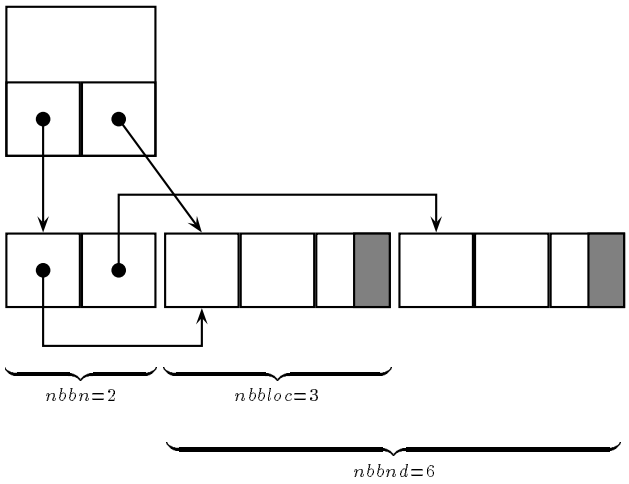
Plus mathématiquement, si la matrice à diagonaliser est de type ligne, alors le produit $S^{-1}DP^{-1}$ restera invariant. En particulier, si les matrices S et P sont initialisées à l'identité, alors, au retour de la fonction, D sera diagonalisée, et $S^{-1}DP^{-1}$ fournira la matrice initiale. *Mutatis mutandis* si la matrice est de type colonne, c'est le produit $P^{-1}DS^{-1}$ qui restera invariant.

Les opérations effectuées sur la matrice P étant uniquement des opérations de permutation, on pourra avoir intérêt à utiliser une matrice de permutation pour accélérer les calculs.

La fonction affecte d'autre part à *p_rk la valeur du rang de la matrice initiale.

Elle retourne une valeur entière parmi les suivantes :

- ZENERR si une erreur s'est produite.
- ZEN_NO_INVERSE si un élément non nul non inversible a été trouvé.
- ZENMAT_HAS_MAXIMAL_RANK= 0 si la matrice est de rang maximal.
- ZENMAT_HAS_KERNEL si la matrice a un noyau non nul.



$$(nbbloc - 1) \times SIZE\_BLOC \leq nbdigit < nbbloc \times SIZE\_BLOC$$

SIZE_BLOC est la taille d'un entier machine et vaut 32 ou 64 bits.

FIG. 5.3 – *Structure de données des matrices binaires*

I.5.2.b Inversion, noyau

L'élimination gaussienne est l'opération qui permet de réaliser l'inversion d'une matrice ou le calcul de son noyau. Ces opérations sont elles aussi disponibles dans la librairie. Toutefois, ces opérations n'ont pas besoin de toutes les informations fournies par la procédure. En effet, si on cherche à calculer l'inverse d'une matrice, alors on peut supposer que la matrice est inversible. Dans ce cas, la matrice de permutation devient inutile puisqu'on cherche à obtenir une relation

$$S^{-1}D = M \text{ avec } D = I.$$

De même, dans le cas du calcul du noyau, on cherche à obtenir une matrice génératrice G de ce noyau, c'est-à-dire telle que

$$MG = 0 \Leftrightarrow S^{-1}DP^{-1}G = 0 \Leftrightarrow DP^{-1}G = 0.$$

La matrice de combinaison des lignes devient là encore inutile et son calcul serait une perte de temps.

C'est pourquoi l'opération de pivot de Gauss proprement dite permet en fait de préciser que certains paramètres sont inutiles par l'emploi de pointeurs NULL. Seules les opérations utiles sont alors activées.

I.5.2.c Autres algorithmes d'inversion

La complexité asymptotique du pivot de Gauss est en $O(n^3)$. Pour l'utilisation sur des matrices quelconques, il existe d'autres algorithmes qui réalisent une meilleure complexité asymptotique. Mais ces algorithmes ont un coût de manipulation prohibitif. Le pivot de Gauss est dont toujours le plus rapide lorsque l'on souhaite manipuler des matrices mémorisables. En effet, même sur une machine disposant de 256Mo de mémoire, on ne peut guère espérer pouvoir manipuler de matrice carrée binaire de taille supérieure à 40000. Pour ces tailles, l'algorithme de Gauss est encore compétitif.

Dans le cas de matrices de très grandes dimensions mais creuses (telles que celles que l'on rencontre en factorisation d'entiers, par exemple) des algorithmes spécifiques existent, comme par exemple celui de Wiedemann [4-Wie86]. On pourra se référer à [1-Gra95] pour une implantation de ce type d'algorithme.

Type	Algorithme de multiplication utilisé
LLL	Chaque ligne de la matrice X est obtenue comme combinaison linéaire des lignes de la matrice B. Les coefficients de cette combinaison sont extraits bit-à-bit de la matrice A.
LLC	Chaque coefficient de la matrice X est obtenu en calculant le poids du ou-exclusif de la ligne et de la colonne correspondante de A et B. Si ce poids est pair, le coefficient est nul, sinon il vaut 1. Cet algorithme est un peu moins bon que le précédent, mais il reste très efficace car le calcul du poids se fait très rapidement par l'emploi d'une table des valeurs du poids de 16 bits consécutifs.
LCL	On utilise le même algorithme que pour la combinaison LLL.
LCC	On a là la pire combinaison possible. En effet, toutes les matrices doivent être accédées bit-à-bit. Il est certainement possible d'envisager une amélioration de cet algorithme, mais il est beaucoup plus simple d'eviter ce cas de figure en choisissant correctement ses matrices.
CLL	On prendra soin aussi d'eviter cette combinaison qui conduit au même algorithme avec accès bit-à-bit de toutes les matrices. La procédure utilisée est la meme que pour le cas LCC.
CLC	On utilise le même algorithme que la combinaison LLC. C'est d'ailleurs la même procédure qui est utilisée dans les deux cas.
CCL	On a là l'algorithme symétrique de celui de la combinaison LLL. Chaque colonne de la matrice X est obtenue par combinaison linéaire de colonnes de la matrice A.
CCC	Le même algoritme que pour CCL est utilisé.

FIG. 5.4 – *Multiplication matricielle dans $GF(2)$*

I.5.3 Cas de $GF(2)$

I.5.3.a Structure de données

La structure de données utilisée pour le cas binaire est de même type que ci-dessus (voir figure 5.3). Mais dans ce cas, le type de la matrice a beaucoup plus d'importance car chaque ligne ou colonne de la matrice est directement représentée par un BigNum. La rapidité d'une opération va donc cette fois-ci dépendre très étroitement du type de ses opérandes.

I.5.3.b Addition

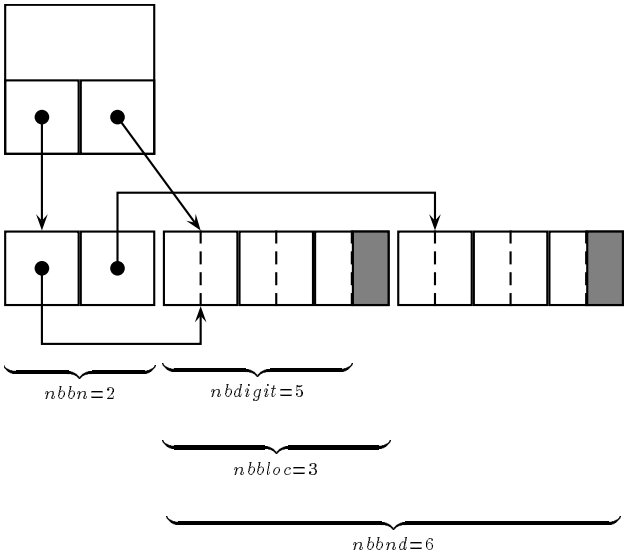
L'addition est un bon exemple de ce qui se passe. Dans $GF(2)$, l'addition de deux matrices est une opération très simple d'un point de vue mathématique. D'un point de vue informatique, par souci d'efficacité, il sera judicieux d'utiliser des matrices de même type. En effet, l'addition des deux matrices reviendra alors à effectuer une opération de ou-exclusif sur la séquence des BigNumDigit qui constitue la matrice. Au contraire si les types sont différents, il faudra extraire chacun des bits d'une matrice pour les ajouter à chacun des bits de l'autre.

Dans le premier cas, la matrice est gérée dans son ensemble, dans le second elle est gérée bit-à-bit. Il est clair que c'est la première méthode qui est la plus rapide. Dans l'arithmétique de ZEN on trouvera donc dans le cas de $GF(2)$ beaucoup d'opérations sur les matrices qui se dédoublent selon le type de leurs opérandes.

I.5.3.c Multiplication

La multiplication matricielle est une opération à trois opérandes. le type de ces opérandes intervient selon huit combinaisons possibles.

Dans le tableau de la figure 5.4 sont énumérés les différents algorithmes utilisés pour réaliser l'opération $X = A \times B$ selon le type du triplet (X, A, B). Ce type est abrégé. Ainsi CLC signifie que X est de type



$$(nbbloc - 1) \times SIZE\_BLOC \leq nbdigit \times S < nbbloc \times SIZE\_BLOC$$

S est la taille d'un élément et vaut :

Arithmétique	Valeur de S
ZETAB	8 bits
ZELOG	16 bits
ZEPS	SIZE_BLOC

FIG. 5.5 – *Structure de données des matrices à coefficients tabulés (*ZEPS, ZETAB *et* ZELOG*)*

colonne, A de type ligne et B de type colonne. On constate tout de suite que tous les algorithmes accèdent forcément à au moins l'une des matrices bit-à-bit.

L'utilisation de l'algorithme de Winograd est ici totalement exclue. En effet, la multiplication et l'addition sont deux opérations de même rapidité qui correspondent aux opérations `and` et `xor` du microprocesseur. Remplacer des multiplications par beaucoup plus d'additions n'est donc pas rentable.

I.5.3.d Conséquences sur le pivot de Gauss

La procédure utilisée pour le pivot de Gauss est aussi modifiée dans son principe par rapport à l'opération générale car il n'y a plus d'inversion ou de multiplication mais seulement des additions de lignes à réaliser. On comprend peut-être d'ailleurs mieux les contraintes imposées sur les types des matrices opérandes. Elles permettent d'obtenir au niveau central de la procédure de pivot des additions de lignes très rapides.

I.5.4 Autres cas spéciaux

PAR SOUCI D'EFFICACITÉ , nous avons vu qu'il existe dans ZEN des arithmétiques particulières. Au niveau matriciel, les algorithmes utilisés sont les mêmes que dans le cas général. Il faut néanmoins noter, que *primo*, ces opérations étant implantées avec des appels directs aux fonctions sur les éléments, elles sont forcément plus rapides que les fonctions générales qui nécessitent la déréférenciation des pointeurs de fonctions. *Secundo*, comme la multiplication est dans tous ces cas aussi rapide que l'addition, la multiplication de Winograd est désactivée.

D'autre part, bien que toujours basées sur les mêmes principes, les structures de données utilisées varient légèrement.

– Dans les cas où chaque coefficient de la matrice est stocké sur un entier machine, ceux-ci sont directement alloués par la structure de données. On retrouve ainsi une structure de données proche de celle

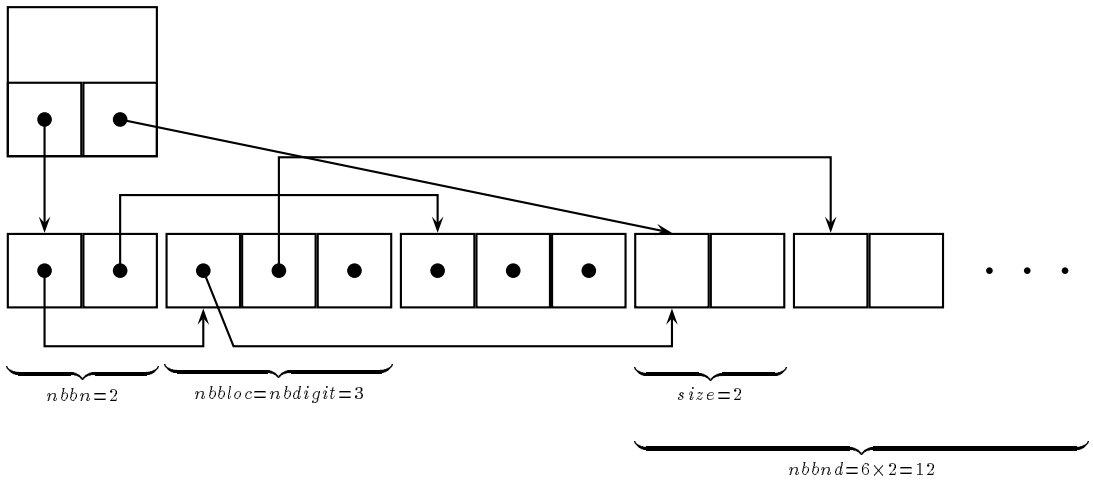


FIG. 5.6 – *Structure de données des matrices à coefficients modulaires (*ZEP*)*

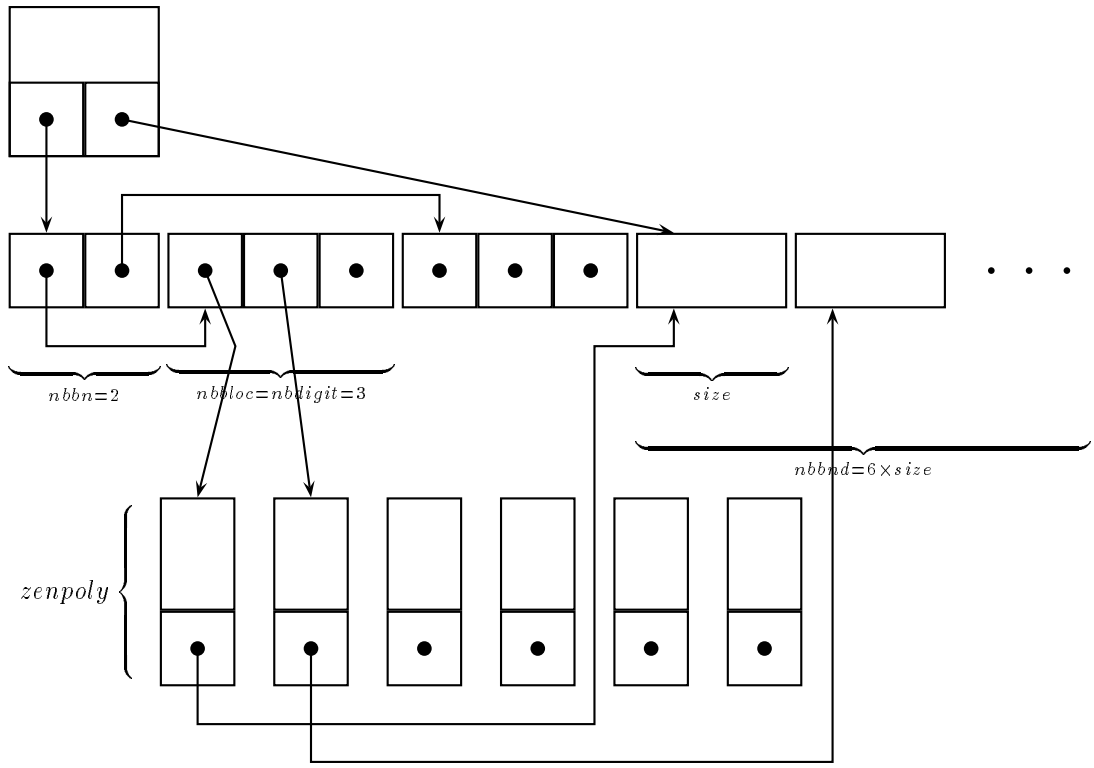


FIG. 5.7 – *Structure de données des matrices sur les extensions*

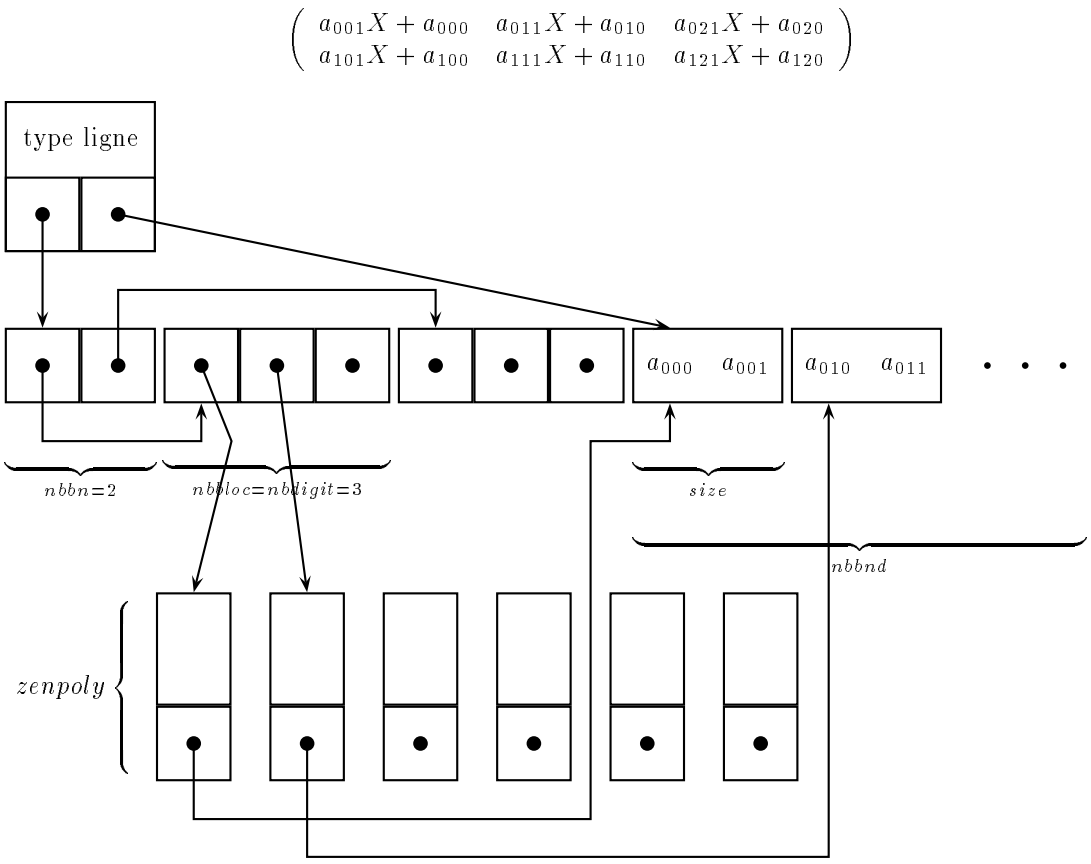


FIG. 5.8 – *Conversion de matrice : matrice initiale*

du cas binaire avec cette fois-ci nbbloc = nbdigit (voir figure 5.5).

- Dans le cas d'un anneau modulaire, l'allocation des grands entiers est aussi effectuée automatiquement (voir figure 5.6). Ceci permet de faire une allocation en une fois au lieu de faire appel à la fonction malloc pour chaque élément. Les différents pointeurs intermédiaires sont ensuite affectés.

- Dans le cas d'une extension polynomiale, les choses se compliquent encore puisque chaque élément est un polynôme (voir figure 5.7). Cette structure de données a été adoptée pour optimiser la conversion des matrices entre une extension polynomiale et son anneau sous-jacent (voir section I.5.5). Les coefficients des polynômes-coefficients de la matrice sont alloués en même temps que la structure de données de la matrice, tandis que les structures de données des polynômes sont allouées individuellement.

I.5.5 Conversion entre extensions

NOUS ABORDONS ICI un autre problème qui peut, à tort, ne pas sembler crucial. En effet ZEN a été écrite pour permettre des calculs dans les extensions. Or, le plus souvent, les algorithmes employés prennent un malin plaisir à considérer tantôt l'élément, tantôt le polynôme sur l'anneau sous-jacent. Cette conversion est très facile dans ZEN puisque le type ZENElt peut être considéré comme un ZENPoly sur l'anneau sous-jacent. Mais les choses se compliquent avec les matrices. En effet, lorsqu'on dispose d'une matrice à coefficients dans une extension (voir figure 5.8), on peut vouloir considérer l'une des deux matrices associées dans l'anneau sous-jacent (voir figures 5.9 et 5.10). Or la structure de ces trois matrices est différente. Une

$$\begin{pmatrix} a_{000} & a_{001} & a_{010} & a_{011} & a_{020} & a_{021} \\ a_{100} & a_{101} & a_{110} & a_{111} & a_{120} & a_{121} \end{pmatrix}$$

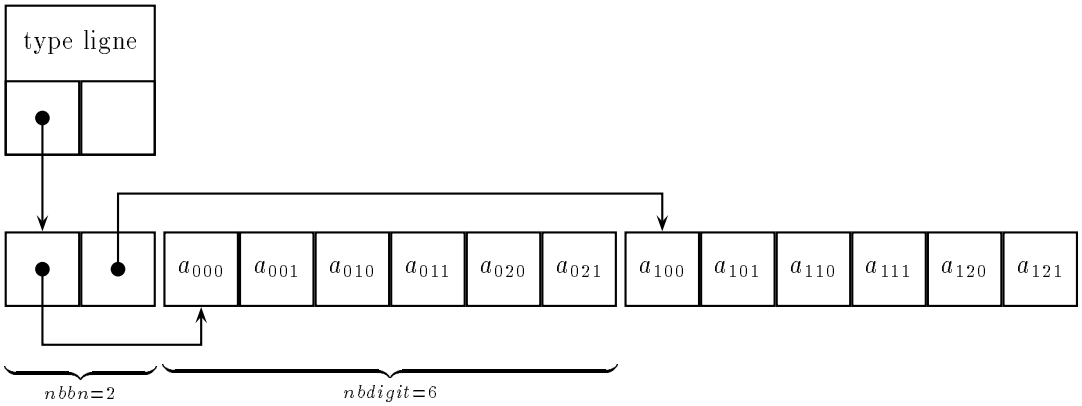


FIG. 5.9 – *Conversion de matrice : matrice type ligne*

$$\begin{pmatrix} a_{000} & a_{010} & a_{020} \\ a_{001} & a_{011} & a_{021} \\ a_{100} & a_{110} & a_{120} \\ a_{101} & a_{111} & a_{121} \end{pmatrix}$$

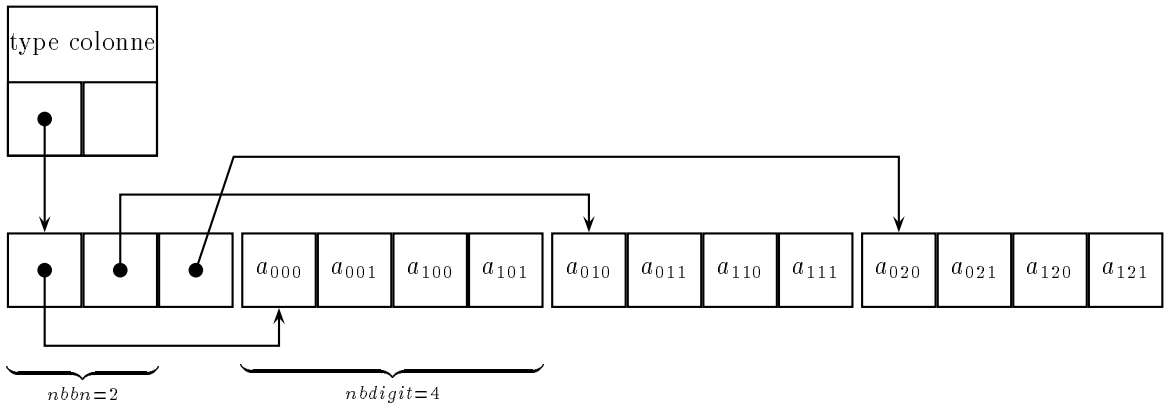


FIG. 5.10 – *Conversion de matrice : matrice type colonne*

opération de conversion est donc nécessaire et celle-ci doit être très rapide.

Les figures permettent de mieux comprendre l'utilité des structures de données utilisées. Lorsque les matrices sont de même type (figures 5.8 et 5.9), l'arrangement des coefficients sur l'anneau de base est le même dans les matrices définies, soit sur cet anneau, soit sur l'extension de cet anneau. La conversion reviendra donc à faire une affectation directe des blocs. Lorsque les deux matrices sont de types opposés (figures 5.8 et 5.10), les affectation peuvent encore se faire par petits blocs correspondant aux polynômes. Les choses se compliquent légèrement dans le cas de $GF(2)$ car les polynômes étant stockés sur un ou plusieurs BigNumDigits, il faut les raccorder entre eux avant d'effectuer l'affectation dans la matrice binaire. Le même genre de problème apparaît dans le cas des anneaux tabulés. Enfin, le problème symétrique se pose pour l'opération inverse.

Deuxième partie

Codes correcteurs d'erreur

Chapitre II.1

Notions de théorie des codes

Nous n'avons pas besoin, pour ce qui nous intéresse, d'entrer très avant dans la théorie des codes correcteurs d'erreur. Nous présentons néanmoins les quelques outils indispensables, car les articles cités utilisent tous la terminologie des codes. En effet la théorie des codes correcteurs n'est apparue que très récemment, vers la fin des années 1940, et pour des raisons largement techniques au départ. Le développement de la transmission sans fil avait montré les limites du système dans le cas de liaisons de mauvaise qualité et une solution technique devait être trouvée. Le vocabulaire utilisé est donc très lié à ces premiers travaux. Or les objets qui sont manipulés en théorie des codes ne sont en fait que des objets très classiques de l'algèbre. C'est donc très naturellement que, par la suite, la théorie s'est développée sur des bases mathématiques. Mais la terminologie est restée la même, sans doute parce que la motivation technique de ces travaux est toujours présente.

Comme pour le chapitre I.1, nous ne donnons que des résultats sans démonstration. Toutes les définitions d'algèbre utilisées ici ont été introduites dans le chapitre I.1, et les nouvelles notions introduites en constituent le prolongement. Le lecteur intéressé par les codes correcteurs d'erreur pourra se reporter à la "bible bleue" [2-MWS83], ou à l'ouvrage français plus technologique [2-PH89], ou encore à [2-PW80].

Nous voyons tout d'abord les quelques définitions nécessaires pour introduire un grand nombre de codes correcteurs (II.1.1). Nous introduisons en particulier les deux métriques qui nous intéressent dans la suite, à savoir la métrique, classique, de Hamming, et celle, moins classique, de Gabidulin (II.1.1.c).

Armés de ces outils, nous pouvons introduire la notion de code correcteur d'erreur (II.1.2). En utilisant le modèle du canal binaire symétrique (II.1.2.b), nous voyons l'intérêt de la métrique de Hamming, nous introduisons la notion de distance minimale d'un code, celle de code MDS et MRD, et nous montrons l'intérêt de la métrique de Gabidulin (II.1.2.c.a). Reste ensuite à décrire des codes connus : les codes de Hamming (II.1.2.d), les codes BCH (II.1.2.e), les codes de Reed-Solomon (II.1.2.f). Ces derniers permettent de définir la classe des codes alternants, qui contient en particulier la classe importante des codes de Goppa (II.1.2.g). Enfin, en métrique de Gabidulin, la classe des codes de Gabidulin est introduite (II.1.2.h).

Le point commun des différents codes présentés est qu'il existe des algorithmes en temps polynomial pour les décoder, ce qui est loin d'être toujours le cas (II.1.3.a). Le but de la section II.1.3 est de présenter un algorithme pour le décodage de ces codes, basé sur l'algorithme d'Euclide étendu (II.1.3.d). Ce rappel permet ensuite d'établir un parallèle avec l'algorithme de décodage des codes de Gabidulin (II.1.4) et d'en faire une description lisible.

II.1.1 Définitions

II.1.1.a Espaces vectoriels

Définition 73. Soit $(K, \oplus, \otimes)$ un corps commutatif, on appelle *espace vectoriel sur* K tout triplet $(E, \oplus, \odot)$ tel que les propriété suivantes soient vérifiées :

1. La structure $(E, \oplus)$ est un groupe commutatif.

2. La loi $\odot$ est externe pour E mais à opérateurs dans K c'est-à-dire que pour tout $\alpha \in K$ et tout élément $x \in E$

$$\alpha \odot x \in E.$$

Elle vérifie en outre :

(a) Pour tout $\alpha \in K$ et pour tout couple $(x, y) \in E^2$ on a

$$\alpha \odot (x \oplus y) = (\alpha \odot x) \oplus (\alpha \odot y).$$

(b) Pour tout couple $(\alpha, \beta) \in K^2$ et tout élément $x \in E$ on a la propriété symétrique de la précédente :

$$(\alpha \oplus \beta) \odot x = (\alpha \odot x) \oplus (\beta \odot x).$$

(c) Pour tout couple $(\alpha, \beta) \in K^2$ et tout élément $x \in E$ on a

$$\alpha \odot (\beta \odot x) = (\alpha \otimes \beta) \odot x.$$

(d) Soit 1 l'élément neutre pour $\otimes$, alors il est aussi élément neutre pour $\odot$: pour tout $x \in E$ on a

$$1 \odot x = x.$$

Les propriétés 2c et 2d font que la loi $\odot$ sera notée de la même façon que la loi $\otimes$, c'est-à-dire le plus souvent en omettant tout signe.

Définition 74. Une partie non vide F d'un $\mathbb{K}$-espace vectoriel $(E, \oplus, \otimes)$ est un *sous-espace vectoriel* de E si et seulement si $(F, \oplus, \otimes)$ a une structure d'espace vectoriel.

Théorème 75. *Soit F une partie non vide d'un espace vectoriel $(E, \oplus, \otimes)$, alors l'ensemble $\mathcal{V}(F)$ défini par*

$$\mathcal{V}(F) = \{x \in E \,/\, \exists n \in \mathbb{N}\ \exists (\alpha_1, \ldots, \alpha_n) \in \mathbb{K}^n\ \exists (x_1, \ldots, x_n) \in F^n\ x = \alpha_1 \otimes x_1 \oplus \cdots \oplus \alpha_n \otimes x_n\}$$

est le sous-espace vectoriel de E engendré par F. *Par définition, la quantité $\alpha_1 \otimes x_1 \oplus \cdots \oplus \alpha_n \otimes x_n$ est une* combinaison linéaire *des éléments $x_1, \ldots, x_n$. L'ensemble $\mathcal{V}(F)$ est donc l'ensemble des combinaisons linéaires d'éléments de F.*

Définition 76. Soit E un $\mathbb{K}$-espace vectoriel, on appelle *famille génératrice* de E toute partie finie F d'éléments de E telle que $\mathcal{V}(F) = E$. Si E admet une famille génératrice, on dit de E qu'il est de *dimension finie*. Une famille finie $G = \{x_1, \ldots, x_n\}$ d'éléments de E est *libre* si et seulement si

$$\forall i,\ 1 \leq i \leq n,\ x_i \notin \mathcal{V}(G \setminus \{x_i\}).$$

Au contraire, une famille finie $G = \{x_1, \ldots, x_n\}$ d'éléments de E est *liée* si et seulement si

$$\exists i,\ 1 \leq i \leq n,\ x_i \in \mathcal{V}(G \setminus \{x_i\}).$$

Une famille finie H d'éléments de E est une *base de E* si et seulement si elle est à la fois libre et génératrice.

Théorème 77. *Si E est un $\mathbb{K}$-espace vectoriel de dimension finie, alors toutes ses bases ont même cardinal appelé* dimension *de E.*

II.1.1.b Matrices

Rappelons que toute matrice M à m lignes et n colonnes à coefficients dans un corps $\mathbb{K}$ peut être considérée comme une famille $(c_1, \ldots, c_n)$ de n éléments,appelés *vecteurs colonnes* de M, d'un $\mathbb{K}$-espace vectoriel de dimension m. De même, on peut considérer M comme une famille $(\ell_1, \ldots, \ell_m)$ de m éléments,appelés *vecteurs lignes* de M, d'un $\mathbb{K}$-espace vectoriel de dimension n.

Théorème 78. *Pour une matrice $m \times n$ donnée M, les espaces vectoriels $\mathcal{V}(c_1, \ldots, c_n)$ et $\mathcal{V}(\ell_1, \ldots, \ell_m)$ sont de même dimension appelée* rang *de la matrice M et noté* $\mathrm{rg}(M)$.

On a évidemment

$$\mathrm{rg}(M) \leq \min(m,n).$$

La matrice M est *de rang maximal* si et seulement si $\mathrm{rg}(M) = \min(m,n)$.

Théorème 79. *Soit J une matrice carrée de taille n. J est inversible si et seulement si elle est de rang maximal $\mathrm{rg}(J) = n$. Soient G une matrice $m \times n$ et D une matrice $n \times \ell$ quelconques, alors si J est inversible*

$$\mathrm{rg}(GJ) = \mathrm{rg}(G) \text{ et } \mathrm{rg}(JD) = \mathrm{rg}(D).$$

D'autre part

$$\mathrm{rg}(GD) \leq \min(\mathrm{rg}(G), \mathrm{rg}(D)).$$

II.1.1.c Distances

Nous nous plaçons désormais (sauf indication contraire) dans le cas d'un corps fini quelconque $\mathbb{F}$. Il peut en particulier s'agir d'une extension polynomiale d'un corps fini premier (voir section I.1.3).

II.1.1.c.a Normes, distances

Définition 80. Soit E un $\mathbb{F}$-espace vectoriel, on dit que N est une *norme* sur E, ou encore que E est un *espace vectoriel normé* par N si et seulement si :

1. **axiome de positivité**
$$\begin{array}{rccc} N & : & E & \to & \mathbb{N} \\ & & x & \mapsto & N(x) \end{array}$$
2. **axiome de séparation**
$$\forall x \in E \; N(x) = 0 \Leftrightarrow x = 0.$$
3. **inégalité triangulaire**
$$\forall (x,y) \in E^2 \; N(x+y) \leq N(x) + N(y).$$
4. **axiome d'homogénéité**
$$\forall \lambda \in \mathbb{F}^\star, \; \forall x \in E, \; N(\lambda x) = N(x).$$

Note 81. Du fait que notre corps de base est fini, cette définition utilise un axiome d'homogénéité différent de la définition habituelle. Dans un corps $\mathbb{K}$ de caractéristique nulle, on aura en effet la condition

$$\forall \lambda \in \mathbb{K} \; \forall x \in E N(\lambda x) = |\lambda| N(x).$$

Cette condition n'est pas judicieuse dans un corps fini. La définition suivante est par contre identique dans les deux cas.

Définition 82. Soit E un sous-ensemble d'un $\mathbb{K}$-espace vectoriel (fini ou non), on dit que Δ est une *distance* (ou une *métrique*) sur E, ou encore que E est un *espace métrique* de distance Δ si et seulement si :

1. **axiome de positivité**
$$\begin{array}{rccc} \Delta & : & E \times E & \to & \mathbb{N} \\ & & (x,y) & \mapsto & \Delta(x,y) \end{array}$$
2. **axiome de symétrie**
$$\forall (x,y) \in E^2 \; \Delta(x,y) = \Delta(y,x).$$
3. **axiome de séparation**
$$\forall (x,y) \in E^2 \; \Delta(x,y) = 0 \Leftrightarrow x = y.$$
4. **inégalité triangulaire**
$$\forall (x,y,z) \in E^3 \; \Delta(x,z) \leq \Delta(x,y) + \Delta(y,z).$$

Note 83. On en déduit aisément que tout $\mathbb{F}$-espace vectoriel normé par N est un espace métrique de distance Δ définie par

$$\forall (x,y) \in E^2 \; \Delta(x,y) = N(x-y).$$

II.1.1.c.b Distance de Hamming

Dans ce qui suit, nous notons V_n l'ensemble des vecteurs de n éléments de $\mathbb{F}$, un corps fini. L'ensemble V_n muni des lois usuelles est un $\mathbb{F}$-espace vectoriel de dimension n. Tout élément $x \in V_n$ s'écrit dans la base canonique. Cette décomposition est notée classiquement $(x_1, \ldots, x_n)$.

Théorème 84 (Hamming). *L'espace V_n est normé par l'application*

$$\begin{array}{rccl} \mathrm{wt} & : & V_n & \to & \mathbb{N} \\ & & (x_1, \ldots, x_n) & \mapsto & \mathrm{card}\{i \,/\, \; x_i \neq 0\} \end{array}$$

Note 85. Cette application est appelée *poids de Hamming* du vecteur x. Elle est définie dans tout corps fini. Il n'en est pas de même de la suivante qui nécessite l'emploi d'une extension sur un corps fini premier ou non.

Théorème 86. *Soit N une norme sur V_n. Si la base canonique $(e_1, \ldots, e_n)$ de V_n est normée pour N*

$$\forall i,\ 1 \leq i \leq n\ N(e_i) = 1,$$

alors

$$\forall x \in V_n\ N(x) \leq \mathrm{wt}(x).$$

Note 87. Ce théorème est un peu moins classique, mais sa démonstration est immédiate. Il s'applique à la distance suivante et va nous permettre de ne considérer que la distance de Hamming dans ce qui suit, les résultats avec la distance de Gabidulin s'en déduisant.

II.1.1.c.c Distance de Gabidulin

Dans ce qui suit, nous notons V_n^m l'ensemble des vecteurs de n éléments de $\mathbb{F}^m$, une extension polynomiale de degré $m \geq 2$ sur un corps fini. L'ensemble V_n^m muni des lois usuelles est un $\mathbb{F}^m$-espace vectoriel de dimension n. Tout élément $x \in V_n^m$ s'écrit dans la base canonique. Cette décomposition est notée classiquement $(x_1, \ldots, x_n)$.

Chaque élément de $\mathbb{F}^m$ peut d'autre part se décomposer dans la base associée au polynôme $P(X) \in \mathbb{F}[X]$ de degré m qui définit $\mathbb{F}^m$ (voir section I.1.3). Pour tout $y \in \mathbb{F}^m$, on note donc $[y_{m-1} \cdots y_0]$ cette décomposition. En combinant ces deux décomposition, on peut associer à tout vecteur $x \in V_n^m$ une matrice à coefficients dans $\mathbb{F}$

$$A(x) = \begin{pmatrix} [& x_{1,m-1} & \cdots & x_{1,0} &] \\ & \vdots & & \vdots & \\ [& x_{n,m-1} & \cdots & x_{n,0} &] \end{pmatrix}$$

Cette matrice possède n lignes et m colonnes et son rang est donc inférieur ou égal à n et m.

Théorème 88 (Gabidulin). *L'espace V_n^m est normé par l'application*

$$\begin{array}{rccl} \mathrm{rk} & : & V_n^m & \to & \mathbb{N} \\ & & x & \mapsto & \mathrm{rg}(A(x)) \end{array}$$

Au premier abord cette norme peut sembler curieuse car la représentation d'un élément $x \in V_n^m$ n'est pas unique. En fait il suffit de voir que toutes les représentations sont liées par des isomorphismes linéaires (théorème 57). Le rang de la matrice $A(x)$ est donc uniquement déterminé par le vecteur x : c'est le *rang* du vecteur x. L'inégalité triangulaire se démontre ainsi aisément en remarquant que si on note $\mathcal{V}(x)$ l'espace vectoriel engendré par $x = (x_1, \ldots, x_n)$, alors $\mathrm{rk}(x) = \mathrm{rg}(A(x)) = \dim \mathcal{V}(x)$. Or $\mathcal{V}(x+y) \subset \mathcal{V}(x) \cup \mathcal{V}(y)$ et $\dim(\mathcal{V}(x) \cup \mathcal{V}(y)) \leq \dim(\mathcal{V}(x)) + \dim(\mathcal{V}(b))$.

II.1.2 Codes correcteurs d'erreur

Nous allons maintenant aborder plus spécialement le problème des codes correcteurs d'erreur. Toutefois, nous conservons pour ces premières définitions le contexte le plus général. Précisons cependant que nous supposons dans la suite que l'espace des messages possibles est "plein", c'est-à-dire que tous les messages sont equi-probables. Nous excluons donc tout "codage compressif" dont le but est généralement, par réduction de l'information redondante des messages, d'accélérer la transmission de l'information pertinente ou de minimiser le stockage de celle-ci. Au contraire, le but que nous poursuivons va nous conduire à rajouter de l'information redondante afin d'eviter la détérioration de l'information pertinente par les aléas de la technique.

II.1.2.a Codes

Définition 89. Soient $\mathbb{K}$ un corps, E_k un $\mathbb{K}$-espace vectoriel de dimension k et F_n un $\mathbb{K}$-espace vectoriel de dimension n, on appellera *code* toute application C injective de E_k dans F_n. Lorsque cette application est en outre linéaire, on parle de *code linéaire*. Nous appelerons k la *dimension* du code, n sera sa *longueur* et la quantité

$$R = \frac{k}{n} \leq 1$$

sera son *taux de transmission* ou *ratio*. Un *mot de code* est un élément de l'image de E_k par l'application C.

Dans toute la suite nous prendrons $E_k = \mathbb{K}^k$ et $F_n = \mathbb{K}^n$ avec $n > k$, c'est-à-dire que nous nous restreignons à des codes redondants.

II.1.2.a.a Codes linéaires

Définition 90. Soit C un code linéaire de $\mathbb{K}^k$ dans $\mathbb{K}^n$, alors une matrice G, de dimensions $k \times n$, à éléments dans $\mathbb{K}$, qui définit l'application linéaire C est appelée *matrice génératrice* du code.

Théorème 91. *La matrice G possède un noyau dans $\mathbb{K}^n$ non réduit à $\{0\}$. Une matrice H qui engendre ce noyau est une* matrice de contrôle *du code. C'est une matrice $\ell \times n$ avec $n \geq \ell \geq n - k$*

Définition 92. Un code linéaire C défini par G, une matrice génératrice $k \times n$, est *non dégénéré* si sa matrice de contrôle H est de dimension $(n-k) \times n$. Un mot du code C (ou par abus de langage un *mot de* G) est un élément $c \in \mathbb{K}^n$ tel que

$$Hc^t = 0.$$

Le code engendré par la matrice H (donc de matrice de contrôle G) est appelé *code dual de* C et noté $C^\perp$.

Définition 93. Soient G et G' (resp. H et H') deux matrices génératrices (resp. deux matrices de contrôle) de deux codes linéaires C et C', alors C et C' sont des codes équivalents si et seulement s'il existe une matrice inversible S et une matrice de permutation P telles que

$$G' = SGP \text{ (resp. } H' = SHP)$$

Par abus de langage on identifie souvent implicitement un code à sa classe d'équivalence.

II.1.2.b Modélisation

La modélisation du transfert de l'information par un canal binaire symétrique sans mémoire est très classique et nous la rappelons très brièvement figure 1.1. On se place dans le cas d'un transfert binaire de données, ce qui correspond aux contraintes techniques actuelles. Alors, on aura pour tout motif d'erreur v de n bits la probabilité

$$\text{Prob}\{e = v\} = \epsilon^{\text{wt}(v)}(1-\epsilon)^{n-\text{wt}(v)}.$$

Comme on peut supposer que la probabilité d'erreur vérifie $\epsilon < \frac{1}{2}$, on constate que la probabilité d'avoir une erreur v décroit avec le poids de Hamming de v.

Définition 94. Le *décodage à distance minimale* d'un mot reçu y consiste à trouver le motif d'erreur e de poids minimum (c'est-à-dire le plus probable dans notre modélisation) tel que $y \oplus e$ soit un mot de code.

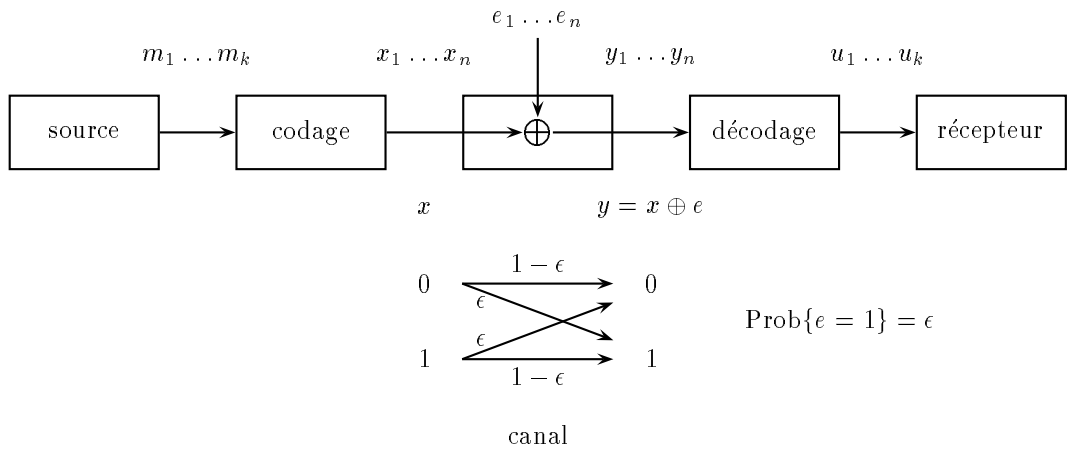


FIG. 1.1 – *Modélisation d'un canal de transmission réel : le canal binaire symétrique de probabilité de transmission erronée ϵ*

II.1.2.c Distance minimale d'un code

II.1.2.c.a Intérêt de la métrique de Hamming

Définition 95. Soit C un code défini sur l'espace métrique $\mathbb{K}^n$ de distance Δ, la *distance minimale* de C pour Δ est la quantité

$$d = \min_{(c,c') \in C^2} \Delta(c, c').$$

Théorème 96. *Dans la modélisation ci-dessus, un code de distance minimale d pour la distance de Hamming peut corriger jusqu'à $\left\lfloor \frac{1}{2}(d-1) \right\rfloor$ erreurs.*

Ce théorème ne signifie pas qu'un décodage erroné se produit dès que l'erreur est de poids de Hamming supérieur à $\left\lfloor \frac{1}{2}(d-1) \right\rfloor$. On peut d'ailleurs construire des codes de distance minimale très faible dont le pouvoir de correction effectif est pourtant très important [7-Sen95]. Il signifie par contre qu'un décodage erroné est impossible si la condition sur le poids de l'erreur est vérifiée.

II.1.2.c.b Borne de Singleton

Dans le cas d'un code linéaire de dimension k sur l'espace vectoriel normé $\mathbb{K}^n$ de norme N, la distance minimale du code est plus facile à obtenir:

$$d = \min_{c \in C} N(c).$$

Corollaire 97. *La distance minimale de tout code linéaire de longueur n et de dimension k vérifie*

$$d \leq n - k + 1.$$

Note 98. On peut définir pour la distance de Gabidulin la notion équivalente de distance minimale de Gabidulin, pour laquelle la borne de Singleton doit aussi être vérifiée.

II.1.2.c.c Codes MDS

Définition 99. Un code linéaire de longueur n, de dimension k, de distance minimale d qui vérifie

$$d = n - k + 1$$

est un code *MDS (Minimum Distance Separable)*.

Note 100. La notion équivalente pour la distance de Gabidulin est celle de code *MRD (Minimum Rank Distance separable)*.

Théorème 101. *Soit C un code linéaire de matrice génératrice G, de longueur n et de dimension k, les propositions suivantes sont équivalentes :*

1. *C est MDS ;*
2. *$C^\perp$ est MDS ;*
3. *toute famille de k vecteurs-colonnes de G est libre.*

II.1.2.c.d Intérêt de la distance de Gabidulin

La métrique de Gabidulin peut conduire à des codes correcteurs d'erreur plus efficaces. En effet, si un code a pour distance minimale de Gabidulin d, tout motif d'erreur de poids de Hamming $w < \lfloor \frac{1}{2}(d-1) \rfloor$ a un rang $r \leq w < \lfloor \frac{1}{2}(d-1) \rfloor$. Il sera donc corrigé. Mais en outre, tout motif de rang faible $r < \lfloor \frac{1}{2}(d-1) \rfloor$, qui peut donc néanmoins avoir un poids élevé, sera aussi corrigé. En particulier, un code MRD est aussi MDS. Il s'agit donc d'un code qui corrige tous les motifs d'erreurs "classiques" d'un code MDS, et qui corrige en outre des motifs de poids de Hamming très élevé mais de rang suffisamment faible.

II.1.2.d Codes de Hamming

Historiquement, les codes de Hamming constituent l'une des premières familles de codes étudiée. Leurs opérations de codage et décodage sont aisées et ils peuvent corriger une erreur.

Définition 102. Soit $r \geq 2$ un entier, Le *code de Hamming binaire à r bits de contrôle* est le code de longueur $n = 2^r - 1$ dont la matrice de parité H_r a pour vecteurs colonnes les n vecteurs binaires non nuls de longueur r.

Par exemple, le code de Hamming de longueur 7 a pour matrice de contrôle

$$\mathcal{H}_7 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

En utilisant le corps fini $GF(2^3)$, on peut obtenir une représentation plus concise de la matrice de contrôle $\mathcal{H}_7$. En effet, soit α un élément générateur de $GF(2^3)$, alors la matrice à coefficients dans $GF(2^3)$

$$H_7 = \begin{pmatrix} \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 & \alpha^4 & \alpha^5 & \alpha^6 \end{pmatrix}$$

considérée dans $GF(2)$ est équivalente à la matrice $\mathcal{H}_7$.

Définition 103. Soit $\mathcal{H}$ une matrice de contrôle d'un code linéaire défini sur une extension polynomiale $\mathbb{F}^m$ sur un corps fini $\mathbb{F}$. Le *code restreint à $\mathbb{F}$* défini par $\mathcal{H}$ est le code linéaire défini sur $\mathbb{F}$ par la matrice $\mathcal{H}$.

II.1.2.e Codes BCH

Les codes de Bose-Chaudhuri-Hocquenghem sont une géneralisation des codes de Hamming.

Définition 104. Soit $\mathbb{F}$ un corps fini de cardinal q et $\mathbb{F}^m$ une extension polynomiale de degré $m > 1$ sur $\mathbb{F}$. Soit α un élément générateur de $\mathbb{F}^m$. Le code BCH primitif au sens strict sur $\mathbb{F}$ de longueur $n = q^m - 1$ et de distance construite δ est l'ensemble des polynômes de $\mathbb{F}[X]/(X^{q^m-1} - 1)$ ayant pour racines $\alpha, \alpha^2, \ldots, \alpha^{\delta-1}$.

Note 105. La généralisation des codes de Hamming est sans doute plus claire quand on remarque que ce code est le code restreint à $\mathbb{F}$ du code linéaire de matrice de contrôle

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{n-1} \\ 1 & \alpha^2 & (\alpha^2)^2 & \ldots & (\alpha^2)^{n-1} \\ \vdots & & & & \\ 1 & \alpha^{\delta-1} & (\alpha^{\delta-1})^2 & \ldots & (\alpha^{\delta-1})^{n-1} \end{pmatrix}$$

Théorème 106. *La distance minimale de ces codes vérifie*

$$d \geq \delta.$$

Ils sont donc au moins t-correcteur avec

$$t = \left\lfloor \frac{\delta - 1}{2} \right\rfloor.$$

Note 107. Nous verrons au chapitre suivant que le plus souvent ces codes ont pour vraie distance minimale δ. Dans tous les cas, même lorsque $d > \delta$, il faut néanmoins noter que les algorithmes de décodage efficaces connus ne parviennent pas à décoder plus de t erreurs.

II.1.2.f Codes de Reed-Solomon

Les codes BCH tels que nous les avons définis sont toujours des codes restreints. Les codes de Reed-Solomon sont souvent présentés comme un cas particulier des codes BCH. Nous préférons les présenter de manière autonome car leur particularité est justement de ne pas être des codes restreints, ce qui leur confère d'intéressantes propriétés.

Définition 108. Soit $\mathbb{F}$ un corps fini de cardinal q. Soit $n = q - 1$. Soit α une racine n-ième de l'unité dans $\mathbb{F}$. Le code de Reed-Solomon sur $\mathbb{F}$ de longueur n et de distance construite δ est le code linéaire de matrice de contrôle

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \dots & \alpha^{n-1} \\ 1 & \alpha^2 & (\alpha^2)^2 & \dots & (\alpha^2)^{n-1} \\ \vdots & & & & \\ 1 & \alpha^{\delta-1} & (\alpha^{\delta-1})^2 & \dots & (\alpha^{\delta-1})^{n-1} \end{pmatrix}$$

En fait nous aurons surtout besoin de l'importante généralisation suivante.

Définition 109. Soit $\mathbb{F}$ un corps fini de cardinal q. Soit $A = (\alpha_1, \dots, \alpha_n)$ une famille d'éléments distincts de $\mathbb{F}$. Soit $V = (v_1, \dots, v_n)$ une famille d'éléments non nuls de $\mathbb{F}$ (mais non nécessairement distincts). Alors le code de Reed-Solomon généralisé de distance construite δ noté $GRS_\delta(A, V)$ est le code linéaire de matrice de contrôle

$$H = \begin{pmatrix} v_1 & v_2 & \dots & v_n \\ \alpha_1 v_1 & \alpha_2 v_2 & \dots & \alpha_n v_n \\ \alpha_1^2 v_1 & \alpha_2^2 v_2 & \dots & \alpha_n^2 v_n \\ \vdots & & & \\ \alpha_1^{\delta-2} v_1 & \alpha_2^{\delta-2} v_2 & \dots & \alpha_n^{\delta-2} v_n \end{pmatrix} \tag{II.1.1}$$

Note 110. Présentée sous cette forme, la généralisation n'est pas évidente. Elle provient en fait du procédé initial d'encodage utilisé pour les codes de Reed-Solomon. On pourra se reporter à [2-MWS83][chapitre 10] pour vérifier qu'un code code de Reed-Solomon est un code de Reed-Solomon généralisé où les éléments v_i sont tous égaux à 1, et les éléments α_i sont les puissances successives d'une racine n^{e} de l'unité.

Théorème 111. *Les codes de Reed-Solomon généralisés sont MDS.*

II.1.2.g Codes alternants

II.1.2.g.a Définition

Définition 112. Soit $\mathbb{K} = \mathbb{F}^m$ une extension polynomiale de $\mathbb{F}$. Soit C un code linéaire défini de $\mathbb{K}^k$ dans $\mathbb{K}^n$, alors le code C' défini de $\mathbb{F}^k$ dans $\mathbb{F}^n$ est le code C *restreint* à $\mathbb{F}$ si et seulement si tout mot du code C' est un mot du code C [1].

Il est facile de voir que si H est la matrice de contrôle du code C, alors une matrice de contrôle H' du code restreint C' s'en déduit en écrivant verticalement chaque composante sur $\mathbb{F}^m$ de la matrice H dans une associée au polynôme définissant l'extension polynomiale sur $\mathbb{F}$.

Définition 113. Un code alternant est un code de Reed-Solomon généralisé sur $\mathbb{F}^m$ restreint à $\mathbb{F}$.

1. Ou encore en utilisant l'image d'une application, $Im(C') = Im(C) \cap \mathbb{F}^n$.

II.1.2.g.b Codes de Goppa

Goppa a décrit de manière différente une sous-classe des code alternant qui présente des propriétés très intéressantes.

Définition Soit $\mathbb{F}$ un corps fini et $\mathbb{F}^m$ une extension polynomiale de degré m sur $\mathbb{F}$. Soit $g(z)$ un polynôme de $\mathbb{F}^m[z]$, de degré $\delta - 1$. Soit L l'ensemble des éléments de $\mathbb{F}^m$ qui ne sont pas des racines de g :

$$L = \{z \in \mathbb{F}^m \,/\; g(z) \neq 0\}$$

Notons $n = \mathrm{card}(L)$ et définissons les éléments de L par

$$L = \{\alpha_i \,/\quad 1 \leq i \leq n\}.$$

Alors, un code de Goppa de longueur n, associé à $g(z)$ est défini comme l'ensemble des vecteurs $C = (C_1, \ldots, C_n)$, qui vérifient

$$\sum_{i=1}^{n} \frac{C_i}{z - \alpha_i} = 0 \bmod g(z) \tag{II.1.2}$$

Ce code peut être vu comme la restriction du code défini sur $\mathbb{F}^m[z]/g(z)$ par la matrice de contrôle

$$H = \left((z-\alpha_1)^{-1} \quad \ldots \quad (z-\alpha_n)^{-1} \right).$$

On peut montrer [2-MWS83, chapitre 12.3] que ce code est aussi un code alternant. Il est de longueur n, de dimension $k \geq n - m \deg g$, de distance construite $\delta = \deg g + 1$, et sa distance minimale vérifie $d \geq \delta$.

Cas particulier des codes BCH Si on prend comme polynôme de Goppa $g(z) = z^{\delta-1}$, alors on retrouve le code BCH de distance construite δ.

Cas particulier des codes binaires de Goppa

Dans le cas où $q = 2$, on peut démontrer que si $\deg g(z) = t$ et si $g(z)$ n'a pas de facteurs irréductibles avec multiplicité, alors le code binaire de Goppa associé à g est t-correcteur (voir [4-Ber73][théorème 3]). Ceci signifie que pour les codes binaires ($q = 2$), il est possible de construire des codes tels que $|L| = 2^m$, c'est-à-dire que le polynôme de degré t, satisfaisant la condition précédente n'a aucune racine dans $GF(2^m)$. On obtient ainsi un code $(2^m, k, t)$ avec

$$k = 2^m - mt \tag{II.1.3}$$

Cette particularité des codes binaires de Goppa est très importante car elle revient à doubler la distance construite du code, plus exactement la distance minimale réelle d'un code binaire de Goppa est au moins le double de sa distance construite. Plus important encore, cette distance minimale est effectivement utilisée par les algorithmes de décodage que nous connaissons.

II.1.2.h Codes de Gabidulin

On peut trouver dans [2-MWS83][page 359] une référence à des codes de Gabidulin décrits comme un cas particulier des codes de Srivastava qui constituent une autre sous-classe des codes alternants. Pour notre part, nous appelons codes de Gabidulin les codes qui ont été décrits dans [4-Gab85]. Ces codes sont aux codes de Reed-Solomon généralisés ce que la distance de Gabidulin est à la distance de Hamming.

Définition114. Soit $\mathbb{F}$ un corps fini de cardinal q. Soit $m > 1$ un entier et $\mathbb{F}^m$ une extension polynomiale sur $\mathbb{F}$ de degré m. Soient $n \leq m$ un entier et $(h_1, \ldots, h_n)$ des éléments de $\mathbb{F}^m$ linéairement indépendants sur $\mathbb{F}$. Alors un code de Gabidulin de distance construite δ est le code linéaire de matrice de contrôle

$$H = \begin{pmatrix} h_1 & h_2 & \ldots & h_n \\ h_1^q & h_2^q & \ldots & h_n^q \\ \vdots & \vdots & & \vdots \\ h_1^{q^{\delta-2}} & h_2^{q^{\delta-2}} & \ldots & h_n^{q^{\delta-2}} \end{pmatrix}. \tag{II.1.4}$$

Théorème115. *Les codes de Gabidulin sont MRD.*

II.1.3 Décodage à distance minimale de Hamming

Nous présentons maintenant très succintement une méthode de décodage des codes qui précèdent. Cette méthode permet de décoder de manière efficace mais connaissant la structure du code utilisé. C'est là toute la différence avec les méthodes que nous décrivons dans le chapitre II.2 qui s'appliquent à tout code linéaire. Il faut noter que le but poursuivi n'est pas le même. Dans le premier cas, le code est utilisé pour la correction d'erreurs et le décodage doit donc être le plus rapide possible. Dans le deuxième cas, notre but est d'attaquer des systèmes cryptographiques pour lesquels la structure des codes utilisés est soit inexistante soit camouflée.

II.1.3.a Complexité

La différence peut se formaliser avec le résultat de complexité du théorème 116. Nous rappelons juste brièvement qu'un problème est dans la classe de complexité $\mathcal{P}$ s'il est résoluble en temps polynomial sur une machine de Turing (une modélisation d'ordinateur) [2]. La classe de complexité $\mathcal{NP}$ correspond aux problèmes pour lesquels la vérification d'une solution au problème est dans $\mathcal{P}$. On a donc

$$\mathcal{P} \subset \mathcal{NP}$$

mais la conjecture très probable

$$\mathcal{P} \neq \mathcal{NP} \tag{II.1.5}$$

reste un problème ouvert. La conjecture (II.1.5) provient en particulier de la notion de problème *complet* pour une classe. Un problème π est complet pour une classe de complexité $\mathcal{C}$ si et seulement s'il existe pour tout autre problème de la classe $\mathcal{C}$ un algorithme en temps polynomial qui le ramène à π. Or il existe beaucoup de problèmes $\mathcal{NP}$-complets [2-GJ78], dont certains très anciens comme celui dit du "voyageur de commerce", pour lesquels on ne connaît pas d'algorithme en temps polynomial les résolvant.

Théorème 116 [4-BMV78]. *Le décodage à distance minimale de Hamming d'un code linéaire sur $GF(2)$ est un problème $\mathcal{NP}$-complet.*

Au contraire, l'algorithme d'Euclide est manifestement un algorithme en temps polynomial. À supposer (II.1.5), ceci signifie que l'algorithme d'Euclide ne peut s'appliquer à tous les problèmes de décodage. Un résultat similaire existe dans le cas $GF(3)$ [6-Bar93].

Conjecture 1 *Le décodage à distance minimale de Hamming d'un code linéaire sur un corps fini est un problème $\mathcal{NP}$-complet.*

Conjecture 2 *Le décodage à distance minimale de Gabidulin d'un code linéaire sur un corps fini de caractéristique 2 est un problème $\mathcal{NP}$-complet.*

Conjecture 3 *Le décodage à distance minimale de Gabidulin d'un code linéaire sur un corps fini est un problème $\mathcal{NP}$-complet.*

Note 117. Les différents problèmes $\mathcal{NP}$-complets connus sont réunis dans [2-GJ78]. Il est d'usage de désigner par une abbréviation un problème donné. Le problème du décodage à distance minimale de Hamming d'un code linéaire sur $GF(2)$ est désigné par les initiales SD de *Syndrome Decoding*, abbréviation que nous utiliserons dans la suite. Pour le problème du décodage à distance minimale de Gabidulin nous proposons RDSD (*Rank Distance Syndrome Decoding*).

II.1.3.b Définitions préliminaires

Dans tout ce qui suit les vecteurs sont considérés comme étant des vecteurs colonnes. Nous donnons un algorithme de décodage qui s'applique au cas le plus général des codes alternants. Soit $\mathbb{F}$ un corps fini de cardinal q, et $\mathbb{F}^m$ une extension polynomiale de degré m sur $\mathbb{F}$. On considère un code alternant associé au

2. Le lecteur intéressé pourra se reporter à [2-Ste90] pour une introduction à la théorie de la complexité.

code de Reed-Solomon généralisé $GRS(A,V)$ défini par la matrice II.1.1. Notre problème est le décodage d'un mot reçu y (voir figure 1.1). Comme nous travaillons avec un code linéaire, nous avons

$$Hy = H(x \oplus e) = Hx \oplus He = He.$$

Par définition, le vecteur He est le *syndrome* du vecteur y. Notre problème est donc, connaissant ce syndrome, de déterminer le vecteur d'erreur $e \in (\mathbb{F})^n$.

Soit w le poids du vecteur d'erreur e. Introduisons la famille $(i_j)_{1 \leq j \leq w}$ des indices d'éléments non nuls Y_j de e. Soit $(e_1, \ldots, e_n)$ la base canonique de l'espace vectoriel $(\mathbb{F})^n$, alors

$$e = \sum_{j=1}^{w} Y_j e_{i_j}$$

Définition 118. On appelle *localisateurs* de e les éléments de $\mathbb{F}^m$

$$X_j = \alpha_{i_j},\ 1 \leq j \leq w.$$

On appelle *polynôme localisateur* de e le polynôme

$$\sigma(z) = \prod_{j=1}^{w} (1 - X_j z).$$

On appelle *polynôme évaluateur* de e le polynôme

$$\omega(z) = \sum_{j=1}^{w} v_{i_j} Y_j \prod_{\substack{1 \leq k \leq w \\ k \neq j}} (1 - X_k z).$$

Soit $(S_0, \ldots, S_{\delta-2})$ le syndrome du vecteur d'erreur e, on appelle *polynôme syndrome* de e le polynôme

$$S(z) = \sum_{j=0}^{\delta-2} S_j z^j$$

où

$$\forall i,\ 1 \leq i \leq \delta - 2,\ S_i = \sum_{j=1}^{w} v_{i_j} Y_j X_j^i. \tag{II.1.6}$$

II.1.3.c Principe du décodage des codes alternants

D'après les définitions qui précèdent, on a

$$\omega(X_j^{-1}) = v_{i_j} Y_j \prod_{k \neq j} (1 - X_k X_j^{-1}).$$

La connaissance des polynômes $\sigma(z)$ et $\omega(z)$ permet donc de déterminer les valeurs de X_j^{-1} (ce sont les racines du polynôme localisateur) et par suite de calculer les valeurs

$$Y_j = \frac{v_{i_j}^{-1} \omega(X_j^{-1})}{\prod_{k \neq j} (1 - X_k X_j^{-1})}$$

donc de déterminer entièrement le vecteur d'erreur. Or on remarque que $\sigma(z)$ et $\omega(z)$ sont premiers entre eux et que $\deg(\omega) < \deg(\sigma) = w$.

Théorème 119. *Pour un code alternant on a*

$$\omega(z) = \sigma(z) S(z) \bmod z^{\delta-1} \tag{II.1.7}$$

Preuve.

$$\begin{aligned}
\frac{\omega(z)}{\sigma(z)} &= \sum_{j=1}^{w} \frac{v_{i_j} Y_j}{1 - z X_j} \\
&= \sum_{j=1}^{w} v_{i_j} Y_j \sum_{k=0}^{\delta-2} (z X_j)^k \bmod z^{\delta-1} \\
&= \sum_{k=0}^{\delta-2} z^k \sum_{j=1}^{w} v_{i_j} Y_j X_j^k \bmod z^{\delta-1} \\
&= \sum_{k=0}^{\delta-2} z^k S_k \bmod z^{\delta-1} \\
&= S(z) \bmod z^{\delta-1}
\end{aligned}$$

□

L'équation II.1.7 s'appelle l'*équation de clef* du code alternant, et les polynômes $\omega(z)$ et $\sigma(z)$ sont ses seules solutions qui vérifient $\sigma(0) = 1$, $\deg(\sigma) \leq t$ et $\deg(\omega) < t$. Plusieurs algorithmes existent pour résoudre l'équation de clef. L'algorithme de Aho, Hopcroft et Ullman, cité dans [2-MWS83][page 369] est asymptotiquement le plus rapide, mais comme souvent en pareil cas, sa complexité le rend moins efficace pour des codes de longueur $n < 10^6$. L'algorithme de Berlekamp [4-Ber73][chapitre 7] est sans doute le plus efficace, mais nous préférons exposer celui beaucoup plus simple dérivé de l'algorithme d'Euclide.

II.1.3.d Algorithme d'Euclide

Théorème 120. *Soient deux polynômes $p(z)$ et $q(z)$ tels que $\deg p \geq \deg q$ Si on introduit les suites de polynômes $(q_i)_{i\geq 1}$ et $(r_i)_{i\geq 1}$ définies par les divisions euclidiennes successives suivantes :*

$$\begin{array}{rcll}
p(z) &=& q_1(z)q(z) + r_1(z) & \deg r_1 < \deg q \\
q(z) &=& q_2(z)r_1(z) + r_2(z) & \deg r_2 < \deg r_1 \\
r_1(z) &=& q_3(z)r_2(z) + r_3(z) & \deg r_3 < \deg r_2 \\
&\vdots & & \vdots \\
r_{j-2}(z) &=& q_j(z)r_{j-1}(z) + r_j(z) & \deg r_j < \deg r_{j-1} \\
r_{j-1}(z) &=& q_{j+1}(z)r_j(z) &
\end{array}$$

alors $r_j(z)$, le dernier reste non nul de ces divisions, est le pgcd de $p(z)$ et $q(z)$.

Corollaire 121. *Soient les suites de polynômes $(U_i)_{i\geq 1}$ et $(V_i)_{i\geq 1}$ définies par :*

$$\begin{array}{rclrcl}
U_{-1}(z) &=& 0 & U_0(z) &=& 1 \\
V_{-1}(z) &=& 1 & V_0(z) &=& 0
\end{array}$$

$$\left\{ \begin{array}{rcl}
U_i(z) &=& U_{i-2}(z) - q_i(z)U_{i-1}(z) \\
V_i(z) &=& V_{i-2}(z) - q_i(z)V_{i-1}(z)
\end{array} \right.$$

alors pour tout $i \geq 1$ on a

$$\left\{ \begin{array}{rcl}
r_i(z) &=& V_i(z)p(z) + U_i(z)q(z) \\
\deg U_i &=& \deg p - \deg r_{i-1} \\
U_i(z) &et& V_i(z) \text{ premiers entre eux}
\end{array} \right.$$

On appelle algorithme d'Euclide étendu l'algorithme qui en résulte. Si on l'applique aux polynômes $p(z) = z^{\delta-1}$ et $q(z) = S(z)$, alors il existe une itération k telle que $\deg r_{k-1} \geq t$ et $\deg r_k < t$, pour laquelle on a

$$r_k(z) \equiv U_k(z)S(z) \bmod z^{\delta-1}.$$

Donnée : Le polynôme syndrome $S(z)$, le taux de correction t du code et son polynôme de Goppa $g(z)$.
Algorithme :

1. On introduit huit variables $P(z)$, $Q(z)$, $D(z)$ et $R(z)$. Initialement

$$\begin{array}{rcl} P(z) = g(z) & \text{et} & Q(z) = S(z), \\ U_0(z) = 1 & \text{et} & U_1(z) = 0, \\ V_0(z) = 0 & \text{et} & V_1(z) = 1. \end{array}$$

2. Calcul de la division euclidienne de $P(z)$ par $Q(z)$:

$$P(z) = D(z)Q(z) + R(z) \text{ avec } \deg R < \deg Q.$$

3. Calcul de

$$\begin{array}{rcl} U_1(z) & \leftarrow & U_1(z) - D(z)U_0(z) \\ V_1(z) & \leftarrow & V_1(z) - D(z)V_0(z) \end{array}$$

4. Si $\deg R(z) < t$ est nul alors

$$\left\{ \begin{array}{rcl} \sigma(z) & = & \frac{U_1(z)}{U_1(0)}, \\ \omega(z) & = & \frac{R(z)}{U_1(0)}, \end{array} \right.$$

et l'algorithme est fini.

5. Sinon, on réalise les permutations

$$\begin{array}{rcl} P(z) \leftarrow Q(z) & \text{et} & Q(z) \leftarrow R(z), \\ U_1(z) \leftrightarrow U_0(z) & \text{et} & V_1(z) \leftrightarrow V_0(z) \end{array}$$

et on reprend à l'étape 2.

FIG. 1.2 – *Algorithme de décodage d'Euclide*

Mais en outre,

$$\begin{array}{rcl} \deg U_k & = & \deg p - \deg r_{k-1} \\ & = & \deg g - \deg r_{k-1} \\ & \leq & \delta - 1 - t \\ \deg U_k & \leq & t \end{array}$$

Comme $\sigma(0) = 1$ on peut donc poser

$$\left\{ \begin{array}{rcl} \sigma(z) & = & \frac{U_k(z)}{U_k(0)} \\ \omega(z) & = & \frac{r_k(z)}{U_k(0)} \end{array} \right.$$

II.1.3.e Cas des codes de Goppa

Ceci nous conduit à l'algorithme de la figure 1.2. Il est ici donné dans le cas d'un code de Goppa. On prendra donc dans ce cas $t = \left\lfloor \frac{\deg(g(z))}{2} \right\rfloor$. Rappelons que lorsqu'on utilise un code binaire de Goppa, le

taux de correction est amélioré $t = \lfloor \deg(g(z)) \rfloor$. Pour un code alternant général on prendra $g(z) = z^{\delta-1}$ et $t = \lfloor \frac{\delta-1}{2} \rfloor$.

II.1.4 Décodage à distance minimale de Gabidulin

Nous présentons maintenant un algorithme de décodage qui s'applique aux codes de Gabidulin. Il s'agit en fait d'une adaptation de l'algorithme de décodage précédent. Des algorithmes apparemment différents sont décrits dans [4-Gab85, 5-Gab91] mais nous espérons que le notre semblera plus lisible. Nous avons en effet cherché à calquer notre algorithme sur l'algorithme de décodage d'Euclide.

II.1.4.a Résultats préliminaires

Tous les résultats qui suivent sont classiques et très bien décrits dans [2-LN86].

Définition 122. Soit $\mathbb{F}$ un corps fini de cardinal q et $\mathbb{F}^m$ une extension polynomiale sur ce corps. On appelle *polynôme linéarisé* un polynôme à coefficients dans $\mathbb{F}^m$ de la forme

$$L(x) = \sum_{i=0}^{n} \alpha_i x^{q^i}.$$

Nous notons $\mathcal{L}_{\mathbb{F}^m}[z]$ l'ensemble des polynômes linéarisés à coefficients dans $\mathbb{F}^m$.

Note 123. Ces polynômes sont dits linéarisés car en tant que fonction polynôme, $L(x)$ est un opérateur linéaire du $\mathbb{F}$-espace vectoriel $\mathbb{F}^m$.

Définition 124. On appelle *multiplication symbolique* de deux polynômeslins $L(x)$ et $L'(x)$ l'opération notée $\otimes$ définie par

$$L(x) \otimes L'(x) = L(L'(x)).$$

Notons que le polynôme linéarisé z est l'élément neutre pour cette opération.

Théorème 125. *L'ensemble $(\mathcal{L}_{\mathbb{F}^m}[z], +, \otimes)$ a une structure d'anneau intègre non commutatif.*

II.1.4.b Définitions préliminaires

Soit $\mathbb{F}$ un corps fini de cardinal q, et $\mathbb{F}^m$ une extension polynomiale de degré m sur $\mathbb{F}$. On considère un code de Gabidulin défini sur $(\mathbb{F}^m)^n$ par la matrice II.1.4. Comme précédemment, nous avons un code linéaire et notre problème est donc, connaissant le syndrome $s = (s_1, \ldots, s_{\delta-2})$, de déterminer le vecteur d'erreur $e \in (\mathbb{F}^m)^n$. Soit r le rang du vecteur d'erreur e, alors il existe une famille libre $E_1, \ldots, E_r$ de vecteurs de $\mathbb{F}^m$ telle que chaque composante de e soit une combinaison linéaire de ces vecteurs. Soit $Y = (Y_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq r}}$ la matrice $n \times r$ à éléments dans $\mathbb{F}$ dont la i-ème ligne est formée des coefficients de la i-ème composante de e dans la base $E_1, \ldots, E_r$, alors on a

$$e = Y \begin{pmatrix} E_1 \\ \vdots \\ E_r \end{pmatrix}.$$

Par suite le syndrome vérifie

$$s = He = HY \begin{pmatrix} E_1 \\ \vdots \\ E_r \end{pmatrix}. \tag{II.1.8}$$

Posons comme nouvelle matrice $X = HY$. Comme la matrice Y est à éléments dans $\mathbb{F}$, on peut vérifier que la matrice X est de la forme

$$X = \begin{pmatrix} x_1 & x_2 & \ldots & x_r \\ x_1^q & x_2^q & \ldots & x_r^q \\ \vdots & \vdots & & \vdots \\ x_1^{q^{\delta-2}} & x_2^{q^{\delta-2}} & \ldots & x_r^{q^{\delta-2}} \end{pmatrix},$$

avec

$$\forall i,\ 1 \leq i \leq r,\ x_i = \sum_{j=1}^{n} Y_{i,j} h_j. \tag{II.1.9}$$

L'équation II.1.8 s'écrit alors

$$\forall i,\ 1 \leq i \leq \delta - 2,\ s_i = \sum_{j=1}^{r} E_j x_j^{q^i}. \tag{II.1.10}$$

Il est clair qu'il suffit de déterminer les valeurs x_i et E_i pour résoudre notre problème puisque les r équations linéaires (II.1.9) sur $\mathbb{F}^m$ fournissent mr équations sur $\mathbb{F}$ permettant de déterminer les nr valeurs $Y_{i,j}$. Rappelons en effet que $n \leq m$ et que les éléments $(h_1, \ldots, h_n)$ sont linéairement indépendants.

Par comparaison avec l'équation II.1.6 nous introduisons maintenant les définitions "linéarisées" qui suivent.

Définition 126. Nous appelons *localisateurs* de e les éléments $x_j,\ 1 \leq j \leq r$ de $\mathbb{F}^m$ définis par l'équation II.1.9.

Nous définissons le polynôme localisateur linéarisé

$$\begin{aligned} \sigma(z) &= \sigma_r(z) \otimes \cdot \otimes \sigma_1(z), \\ &= \bigotimes_{j=1}^{r} \sigma_j(z), \end{aligned}$$

où les polynômes linéarisés $\sigma_j(z)$ sont définis par récurrence :

$$\begin{aligned} \sigma_1(z) &= z - x_1^{1-q} z^q, \\ \sigma_j(z) &= z - \left[\sigma_{j-1}(x_j)\right]^{1-q} z^q \text{ pour tout } 1 < j \leq r. \end{aligned}$$

Il est clair que toutes les valeurs x_j sont des racines du polynôme linéarisé $\sigma(z)$.

Définition 127. On appelle *polynôme syndrome linéarisé* le polynôme

$$S(z) = \sum_{j=0}^{\delta-2} s_j z^{q^j},$$

où les coefficients s_j vérifient l'équation II.1.10.

Il nous faut maintenant déterminer l'équivalent linéarisé du polynôme évaluateur. La forme en est un peu plus complexe car la multiplication symbolique n'est pas commutative. Nous utilisons ici la division euclidienne qui existe malgré tout. Nous posons pour tout j compris entre 1 et r

$$(z - x_j^{q-1} z^q) \otimes \omega_j(z) + \rho_j z = \sigma(z). \tag{II.1.11}$$

Notons bien que le reste ρ_j n'est pas nul dans la plupart des cas du fait de la non commutativité de la multiplication symbolique. Le polynôme évaluateur linéarisé voit donc apparaître un terme correctif qui en tient compte :

$$\omega(z) = \sum_{j=1}^{r} \left[E_j x_j \omega_j(z) + \sum_{k=0}^{\delta-2} E_j \left(x_j \rho_j z\right)^{q^k} \right].$$

Avant de démontrer l'équation qui lie ces différents polynômes, voyons tout de suite comment les utiliser. Les racines du polynôme localisateur linéarisé permettent d'obtenir les valeurs x_i. Connaissant $\sigma(z)$, il est possible d'obtenir les valeurs x_i et donc les polynômes linéarisés $\omega_j(z)$ et les constantes ρ_j comme résultat de la division euclidienne (II.1.11). Par identification terme à terme avec l'expression du polynôme $\omega(z)$ on en déduit un système linéaire qui donne les valeurs des E_i.

II.1.4.c Équation de clef linéarisée

Théorème 128. *Pour un code de Gabidulin on a*

$$\omega(z) = S(z) \otimes \sigma(z) \bmod z^{q^{\delta-1}} \tag{II.1.12}$$

Preuve. Notons tout d'abord la propriété suivante, valable pour tout couple de polynômes linéarisés $(L(x), L'(x))$ et tout élément α, qui découle directement de la définition de la multiplication symbolique :

$$L(\alpha x) \otimes L'(x) = L(x) \otimes (\alpha L'(x)).$$

On a

$$\begin{aligned} S(z) &= \sum_{k=0}^{\delta-2} s_k z^{q^k}, \\ &= \sum_{k=0}^{\delta-2} \left(\sum_{j=1}^{r} E_j x_j^{q^k} \right) z^{q^k}, \\ &= \sum_{j=1}^{r} E_j A_j(z), \end{aligned}$$

avec

$$A_j(z) = \sum_{k=0}^{\delta-2} (z x_j)^{q^k}.$$

Nous cherchons à calculer $S(z) \otimes \sigma(z)$, alors remarquons que

$$\begin{aligned} A_j(z) \otimes \sigma(z) &= \left[\sum_{k=0}^{\delta-2} (z x_j)^{q^k} \right] \otimes \sigma(z) \\ &= \sum_{k=0}^{\delta-2} z^{q^k} \otimes \left[x_j \left((z - x_j^{q-1} z^q) \otimes \omega_j(z) + \rho_j z \right) \right] \\ &= \sum_{k=0}^{\delta-2} z^{q^k} \otimes \left[(x_j z - x_j^q z^q) \otimes \omega_j(z) + x_j \rho_j z \right] \\ &= \sum_{k=0}^{\delta-2} z^{q^k} \otimes \left[(z - z^q) \otimes x_j \omega_j(z) + x_j \rho_j z \right] \end{aligned}$$

Nous pouvons en outre noter que

$$\begin{aligned} \left[\sum_{k=0}^{\delta-2} z^{q^k} \right] \otimes (z - z^q) &= z - z^{q^{\delta-1}}, \\ &= z \bmod z^{q^{\delta-1}}, \end{aligned}$$

où la réduction modulaire s'effectue bien sûr au sens de la multiplication symbolique. Par suite

$$\begin{aligned} A_j(z) \otimes \sigma(z) &= x_j \omega_j(z) + \sum_{k=0}^{\delta-2} z^{q^k} \otimes (x_j \rho_j z) \bmod z^{q^{\delta-1}}, \\ E_j A_j(z) \otimes \sigma(z) &= E_j x_j \omega_j(z) + \sum_{k=0}^{\delta-2} E_j (x_j \rho_j z)^{q^k} \bmod z^{q^{\delta-1}}, \\ \sum_{j=1}^{r} E_j A_j(z) \otimes \sigma(z) &= \sum_{j=1}^{r} \left[E_j x_j \omega_j(z) + \left(\sum_{k=0}^{\delta-2} E_j (x_j \rho_j z)^{q^k} \right) \right] \bmod z^{q^{\delta-1}}, \\ S(z) \otimes \sigma(z) &= \omega(z) \bmod z^{q^{\delta-1}}. \end{aligned}$$

□

L'équation II.1.7 s'appelle l'*équation de clef linéarisée* du code de Gabidulin, et on peut démontrer par les mêmes arguments que pour les codes alternants que les polynômes $\omega(z)$ et $\sigma(z)$ sont ses seules solutions qui vérifient $\sigma(0) = 1$, $\deg(\sigma) \leq q^t$ et $\deg(\omega) < q^t$ où $t = \left\lfloor \frac{\delta-1}{2} \right\rfloor$ est le taux de correction du code pour la distance de Gabidulin.

II.1.4.d Algorithme de décodage des codes de Gabidulin

L'algorithme de décodage des codes de Gabidulin consiste donc à résoudre l'équation de clef linéarisée par l'algorithme d'Euclide étendu qui se déduit de la division euclidienne linéarisée. Une fois déterminés les polynômes linéarisés $\omega(z)$ et $\sigma(z)$, il est possible de calculer les racines du polynôme σ. Plus exactement, on détermine le noyau de l'application linéaire associée à $\sigma(z)$. Chaque élément d'une base de cette ensemble est donc en fait une combinaison linéaire des valeurs x_i, et la détermination des valeurs E_i associées se fait donc à une combinaison linéaire près.

L'analyse de l'algorithme montre que toutes ses étapes sont en $O(n^3)$ ce qui correspond à la complexité de l'algorithme de décodage des codes alternants. Toutefois, l'algorithme est sans doute un peu plus délicat à implanter et forcément un peu plus lent puisque des résolutions supplémentaires de systèmes linéaires sont nécessaires.

Chapitre II.2

Décodage général des codes correcteurs d'erreur – cas de la distance de Hamming

Nous allons maintenant présenter des algorithmes de décodage général de codes linéaires. Ces algorithmes ne tiennent aucun compte de la structure éventuelle du code car ils ont un but cryptanalytique. Or cette structure dans ce contexte est soit inexistante (code aléatoire), soit cachée par une méthode que nous supposons sûre. Nous préciserons au chapitre III.1 à quel type de cryptosystèmes ces algorithmes s'attaquent, et quels sont les autres types d'attaque possibles.

Les algorithmes existants ont d'abord été optimisés [5-Cha94], puis généralisés sous la forme d'un nouvel algorithme. Ce travail [7-CC95a] est le fruit d'une collaboration avec Anne Canteaut de l'INRIA.

Le problème du décodage général à distance minimale de Hamming d'un code linéaire est un problème difficile. Pour les codes binaires on sait qu'il est $\mathcal{NP}$-complet [4-BMV78]. Dans le cas général nous avons vu au paragraphe II.1.3.a que nous conjecturions que ce problème conservait sont caractère $\mathcal{NP}$-complet.

Dans ce chapitre, nous commençons par énumérer un certain nombre d'algorithmes existants qui permettent le décodage général des codes correcteurs (II.2.1). Ces algorithmes permettent d'envisager une généralisation dont nous étudions le facteur de travail (II.2.2). En incorporant une autre idée due à Omura, un algorithme itératif est obtenu et son temps de calcul estimé (II.2.3). Les résultats obtenus expérimentalement sont ensuite comparés aux autres algorithmes existants (II.2.4). L'utilisation de ces algorithmes a permis en outre de lever l'incertitude sur la distance minimale réelle de six codes BCH en longueur 511 (II.2.5).

II.2.1 Algorithmes existants

II.2.1.a Algorithme de Korzhik-Turkin

L'algorithme de Korzhik-Turkin [5-KT91] a été présenté à Eurocrypt'91. Nous ne le citons ici que pour rappeler que ses auteurs se sont depuis rétractés et que cet algorithme fantôme n'existe pas !

II.2.1.b Algorithme de Dumer

L'algorithme de Dumer [5-Dum92] consiste principalement à énumérer tous les mots possibles de poids inférieur au taux de correction du code et à vérifier si leur syndrome correspond à celui du mot à décoder. Les syndromes peuvent être précalculés à l'avance, et on obtient ainsi un algorithme de décodage en temps égal à celui de l'accès à la table des syndromes.

Malheureusement, cet algorithme n'est pas réaliste, même pour de petites dimensions car sa complexité en espace est exponentielle. Ainsi, même pour un code de longueur 256 corrigeant 14 erreurs, ceci conduit à plus de $\binom{256}{14} \simeq 2^{75}$ couples mot de code/syndrome à mémoriser.

Donnée : Une matrice génératrice G d'un code linéaire, et un vecteur $y = c + e$, où $c = mG$ est un mot de code codant le message M, et e est un vecteur d'erreur corrigeable, c'est-à-dire un mot de poids faible.
But : Déterminer le vecteur d'erreur e.
Paramètres : Un entier p. Dans le cas de l'algorithme de McEliece $p = 0$.
Algorithme :

1. Permutation aléatoire des colonnes de G par P. On obtient ainsi $\hat{G} = GP$
2. Élimination gaussienne sur $\hat{G}$ pour obtenir une forme systématique G' de la matrice génératrice
$$G' = (\, I_k \quad | \quad A \,),$$
3. On applique au vecteur à décoder les mêmes permutations de colonnes et on obtient $y' = (\, c'_k \oplus e'_k \quad | \quad c'_{n-k} \oplus e'_{n-k} \,)$.
4. Comme P est une permutation aléatoire, on peut supposer que les k premières colonnes de $\hat{G}$ sont telles que le poids de e'_k est inférieur à p. Cette hypothèse est vérifiable grâce à la forme systématique G' de la matrice génératrice. On énumère tous les motifs e'_k de poids inférieur à p et on en déduit pour chacun un vecteur d'erreur e''_{n-k} dont on peut vérifier s'il est de poids inférieur à la distance minimale du code minorée de p. Si c'est le cas, on réalise les permutations inverses des précédentes pour retrouver le vecteur d'erreur initial e, donc c puis m. Sinon, on reprend l'algorithme depuis le début.

FIG. 2.1 – *Algorithme de Lee-Brickell (algorithme de McEliece)*

Notons néanmoins que cet algorithme a une complexité asymptotique en espace du même ordre de grandeur que la complexité en temps des algorithmes que nous allons développer.

II.2.1.c Algorithme de Levitin-Hartman

L'algorithme de Levitin-Hartman [4-LH85] réalise un meilleur compromis. Il utilise la notion de mot de code voisin de zéro et permet une fois l'ensemble de ces mots connus de décoder le code. Mais dans le cas général, on ne sait pas déterminer de façon pratique l'ensemble des voisins de zéro (voir à ce sujet les discussions dans [7-Cha92] et [2-Til94]).

II.2.1.d Algorithme de McEliece

Nous verrons au chapitre suivant la description du cryptosystème de McEliece. D'ores et déjà nous décrivons figure 2.1 l'algorithme de McEliece qui fut présenté comme une attaque possible de ce cryptosystème. Bien que décrit [7-McE78] et étudié initialement [5-AM87] dans le cas binaire, il peut s'appliquer dans tout corps fini. Il s'agit d'un algorithme de décodage général des codes linéaires.

II.2.1.e Algorithme de Lee-Brickell

Lee et Brickell ont remarqué dans l'algorithme de McEliece que l'étape coûteuse était l'élimination gaussienne. Ils ont alors proposé [5-LB88] de vérifier à l'étape 4 si le poids du vecteur e_k n'était pas inférieur à un certain paramètre p. Lorsque $p = 0$ on retrouve l'algorithme de McEliece initial. Lorsque $p > 0$, ceci permet de réduire le nombre d'éliminations gaussiennes à effectuer. En contrepartie l'étape 4 devient une énumération des différents motifs d'erreurs e_k de poids inférieur à p possibles, donc voit sa complexité croître avec p (voir figure 2.1). De manière heuristique, Lee et Brickell avaient déterminé que le paramètre $p = 2$ était le meilleur compromis possible.

Donnée : Une matrice génératrice G d'un code linéaire de dimensions $[n, k]$. Un critère $\mathcal{C}$ dépendant du poids trouvé et du nombre d'itérations effectuées.
But : Déterminer un mot de code de G de poids faible.
Paramètres : Deux entiers σ et p.
Algorithme :

1. Initialisation de la variable $W \leftarrow n$.
2. Permutation aléatoire des colonnes de G par P. On obtient ainsi $\hat{G} = GP$
3. Élimination gaussienne limitée aux $k + \sigma$ dernières colonnes de $\hat{G}$ pour obtenir une matrice génératrice équivalente

$$\tilde{G} = \left(\begin{array}{c|c|c} B & Z & I_e \\ \hline D & 0 & 0 \end{array} \right),$$

où B est une matrice $(n - k - \sigma) \times e$, Z est une matrice $(k + \sigma - e) \times e$ et D est une matrice $(n - k - \sigma) \times (k - e)$ où $e \leq k + \sigma$.

4. En considérant uniquement la matrice Z, chercher les combinaisons linéaires qui conduisent à des mots de code tels que le poids des $k + \sigma$ colonnes sélectionnées à l'étape précédente soit inférieur à p.
5. Lorsque le critère précédent est vérifié, calcul de la combinaison linéaire complète et de son poids w. Si $w < W$, alors sortie du mot de code obtenu et ajustement $W \leftarrow w$. Si $e \neq 0$, alors toutes les combinaisons linéaires faisant intervenir la matrice D doivent être testées.
6. Si le critère $\mathcal{C}$ est vérifié alors arrêt du programme. Sinon, on reprend l'algorithme à l'étape 2.

FIG. 2.2 – *Algorithme de Leon*

II.2.1.f Algorithme de Leon

L'algorithme de Leon est un algorithme dont le but initial était de déterminer des mots de poids faible dans un code linéaire. Il permit en particulier de déterminer les distances minimales réelles de certains codes BCH. Nous décrivons figure 2.2 l'algorithme initial présenté dans [4-Leo88]. Cet algorithme utilise un critère d'arrêt $\mathcal{C}$ qui dépend du poids le plus faible W trouvé à l'itération N. Dans la présentation initiale de l'algorithme, ce critère permet d'assurer qu'avec une certaine probabilité $1-\epsilon$, aucun mot de code de poids strictement inférieur à W n'existe. Les paramètres proposés dans [4-Leo88] et partiellement confirmés par une étude asymptotique [5-Cha92] sont $p=\sigma=2$. Il s'avère néanmoins que ces paramètres ne sont pas optimaux pour les dimensions de codes que l'on considère usuellement.

II.2.1.g Algorithme de Stern

L'algorithme de Stern, obtenu de manière indépendante des précédents, s'applique exclusivement au cas binaire, en utilisant un compromis astucieux entre mémorisation et temps de calcul [5-Ste89]. Les paramètres optimaux obtenus par une étude asymptotique dans [5-Cha92] sont $p=3$ et $\ell=\lceil \log k \rceil$. Ici encore, ce paramétrage ne conduit malheureusement pas à un résultat optimal pour les dimensions qui nous intéressent.

La notation Stern($2\times p$,ℓ,$2\times p$) de la figure 2.3 sera expliquée au paragraphe II.2.2.b.c.

II.2.2 Généralisation

Nous présentons maintenant une généralisation des algorithmes précédents. Nous obtenons ainsi un algorithme de décodage général de tout code linéaire défini sur un corps fini $\mathbb{F}$.

II.2.2.a Un algorithme plus général

Nous considérons une matrice génératrice G d'un code linéaire de dimensions $[n,k]$. On note $\mathcal{N}=\{1,\ldots,n\}$, et les vecteurs colonnes de cette matrice sont notés G_i pour tout $i\in\mathcal{N}$. Les vecteurs lignes de la matrice G sont des mots de code notés $(c_j)_{1\leq j\leq k}$.

Définition 129. Une *fenêtre d'information* de G est un sous-ensemble $\mathcal{I}\subset\mathcal{N}$ à k éléments tel que $V=(G_i)_{i\in\mathcal{I}}$ soit une matrice inversible. On note de même $W=(G_i)_{i\in\mathcal{N}\setminus\mathcal{I}}$ la matrice complémentaire et la *décomposition* de G selon la fenêtre d'information $\mathcal{I}$ est notée

$$G=(V,W)_{\mathcal{I}}.$$

La *forme systématique* de G associée à la fenêtre d'information $\mathcal{I}$ est la matrice $V^{-1}G=(Id_k,V^{-1}W)$.

Alors tous les algorithmes qui précèdent peuvent être modélisés par la répétition de la séquence suivante :

1. Choix d'une fenêtre d'information.
2. Élimination gaussienne.
3. Recherche d'un mot de poids faible à l'aide d'heuristiques assez voisines :
   - pour l'Algorithme de Lee-Brickell, on suppose que la fenêtre d'information contient au plus p erreurs ;
   - pour l'algorithme de Leon, on suppose qu'une sélection de colonnes légèrement plus grande que la fenêtre d'information contient au plus p erreur. En fait, la description initiale de l'algorithme de Leon ne suppose pas le choix d'une fenêtre d'information mais nous avons montré [5-Cha94] qu'il était plus efficace de se placer dans ce cas ;
   - pour l'algorithme de Stern, on suppose que la fenêtre d'information contient $2p$ erreurs et qu'elles se répartissent pour moitié dans chaque jeu de colonnes X et Y.

Tous les objets utilisés sont définis sur $GF(2)$.

Donnée : Une matrice de contrôle H d'un code linéaire de dimensions $[n,k]$.
But : Déterminer un mot de code de H de poids égal à une valeur w.
Paramètres : Deux entiers ℓ et p.
Algorithme :

1. Permutation aléatoire des colonnes de H par P. On obtient ainsi $\hat{H} = HP$
2. Élimination gaussienne limitée aux $n-k$ dernières colonnes de $\hat{H}$ pour obtenir une matrice de contrôle équivalente
$$\tilde{H}_1 = (\, Q \quad | \quad I_{n-k}\,) .$$
Si l'élimination sur les $n-k$ dernières colonnes n'est pas possible pour cause de dégénerescence, alors on reprend à l'étape 1.
3. Partition aléatoire des colonnes de Q en deux sous ensembles et permutations correspondantes sur les colonnes pour obtenir une matrice équivalente de la forme
$$\hat{H}_2 = (\, X \quad | \quad Y \quad | \quad I_{n-k}\,) .$$
4. Choix aléatoire de ℓ lignes de la matrice et permutation de lignes correspondantes pour obtenir la forme
$$\tilde{H}_3 = \left(\begin{array}{c|c|c} X_\ell & Y_\ell & \\ \hline X_{n-k-\ell} & Y_{n-k-\ell} & J \end{array} \right) .$$
5. Pour chaque groupe P_X de p colonnes de X_ℓ, calculer leur somme $\sigma_\ell(P_X)$ et mémorisation pour chaque somme de la combinaison linéaire P_X correspondante.
6. Pour chaque groupe P_Y de p colonnes de Y_ℓ calculer la somme $\sigma_\ell(P_Y)$. Alors pour chaque groupe P_X mémorisé tel que $\sigma_\ell(P_X) = \sigma_\ell(P_Y)$, considérer les $2p$ colonnes $P_X \cup P_Y$ et calculer la somme V des $n-k-\ell$ lignes restantes.
7. Si le poids de V est $w-2p$ alors construire le mot de code de poids w correspondant et stopper. Sinon, reprendre à l'étape 1.

FIG. 2.3 – *Algorithme de Stern*$(2\times p, \ell, 2\times p)$

Donnée : Une matrice génératrice G d'un code linéaire de dimensions $[n, k]$.
But : Déterminer un mot de code de G de poids faible w.
Paramètres : Trois entiers p, σ et s.
Algorithme :

1. Permutation aléatoire des colonnes de G par P. On obtient ainsi $\hat{G} = GP$
2. Élimination gaussienne sur $\hat{G}$ pour obtenir une forme systématique G' de la matrice génératrice
$$G' = (\, I_k \quad | \quad Z \quad | \quad A \,) .$$
La matrice Z contient σ colonnes, et donc A est une matrice à k lignes et $n - k - \sigma$ colonnes.
3. En considérant uniquement la matrice Z, chercher les combinaisons linéaires d'au plus p lignes qui conduisent à des mots de code tels que le poids des $k + \sigma$ colonnes sélectionnées à l'étape précédente soit inférieur à s.
4. Lorsque le critère précédent est vérifié, calcul de la combinaison linéaire complète et de son poids w'. Si $w' = w$, alors sortie du mot de code obtenu après permutation inverse des colonnes et arrêt de l'algorithme. Sinon, on reprend l'algorithme à l'étape 1.

FIG. 2.4 – *Algorithme* $A(p, \sigma, s)$

Ceci nous conduit à la description de l'algorithme de la figure 2.4. Cet algorithme $A(p, \sigma, s)$ introduit trois paramètres :

p peut être décrit comme une mesure de flou sur la fenêtre d'information. En effet, si $p = 0$, on cherche des fenêtres d'information exactes. Si $p = 1$, on s'autorise une erreur, que l'on cherchera de manière exhaustive, à l'intérieur de la fenêtre d'information. Et ainsi de suite.

σ est une mesure de la proportion de mot qui va être considérée dans un premier temps. En effet, nous cherchons un mot de poids faible, il est donc probable que proportionnellement, sa restriction à σ bits soit aussi de poids faible.

s est le seuil de test. Lorsque le poids des σ bits est inférieur à un certain seuil déterminé par s, alors on teste si le mot complet est effectivement de poids faible.

II.2.2.b Variantes

II.2.2.b.a Décodage

En fait c'est une classe d'algorithmes que nous introduisons. Il est clair que l'algorithme A peut être adapté au décodage général en s'inspirant par exemple de la description de l'Algorithme de Lee-Brickell. Pour différencier ces deux types d'algorithmes, nous notons $A_{rec}(p, \sigma, s)$ l'algorithme décrit figure 2.4 et $A_{dec}(p, \sigma, s)$ sa variante de décodage. Alors $A_{dec}(0, 0, 0)$ est l'algorithme de McEliece et $A_{dec}(p, 0, p)$ est l'Algorithme de Lee-Brickell. D'autre part $A_{rec}(p, \sigma, p)$ est la variante de l'algorithme de Leon décrite dans [5-Cha94].

II.2.2.b.b Versions duales

On peut obtenir une version duale de l'algorithme A opérant sur une matrice de contrôle. En effet, si H est une matrice de contrôle associée à une matrice génératrice G, alors il est clair que $\mathcal{I}$ est une fenêtre d'information pour G si et seulement si $\mathcal{N} \setminus \mathcal{I}$ est une fenêtre d'information pour H. On peut alors représenter

par blocs la matrice de contrôle H' issue de l'élimination gaussienne de la première étape en privilégiant σ lignes :

$$H' = \left(\begin{array}{c|c} A & \\ \hline Z & I_{n-k} \end{array} \right).$$

En considérant des combinaisons linéaires des colonnes de Z, on obtient exactement le même critère que précédemment. Nous notons $A_\perp(p, \sigma, s)$ cette variante.

On pourrait envisager de même une variante permettant le décodage étant donné un syndrome. En fait, cette variante bien que possible n'est pas intéressante car l'opération d'élimination gaussienne ne laisse pas le syndrome invariant. Il est alors plus astucieux de déterminer un vecteur quelconque c admettant le même syndrome et de considérer le code engendré par les vecteurs lignes de G et c. Ce code a même longueur que le précédent, il contient un mot de poids faible qui est l'erreur recherchée et sa matrice de contrôle H'' est de dimension $(n-k)-1 \times n$. On peut donc lui appliquer l'algorithme $A_\perp$.

II.2.2.b.c Mémorisation partielle

Les descriptions précédentes sont valables dans tout corps fini. Nous nous intéressons maintenant plus particulièrement à $GF(2)$. Il ressort de [7-CC95a] que l'algorithme de Stern reste plus efficace dans le cas binaire grâce à un compromis mémoire/temps de calcul. En modifiant légèrement l'algorithme A, il est possible d'utiliser une stratégie similaire. À cette fin, nous remplaçons l'étape 3 de l'algorithme A :

3'a. Mémorisation des lignes de Z dans une table de hachage indexée par la valeur en binaire de la ligne. Cette table de hachage a donc 2^σ entrées.

3'b. Si une ligne de Z est nulle, calcul du poids de la ligne complète et comparaison avec w.

3'c. Pour chaque collision de la table de hachage, calcul du poids de la somme des deux lignes complètes correspondantes et comparaison avec w.

Cette description a implicitement fixé deux des degrés de liberté de l'algorithme initial. En effet, nous considérons obligatoirement des paires de lignes de Z, le paramètre p est donc en fait fixé à 2. De même, le paramètre de seuil est lui aussi fixé à 2 puisque nous cherchons des collisions dans la table de hachage. Nous notons donc cet algorithme $A'_{rec}(2, \sigma, 2)$.

On peut encore ajouter le test suivant :

3"d. Pour chaque somme de deux lignes de Z, consultation de la table de hachage et calcul éventuel du poids des sommes à trois lignes correspondantes.

On obtient ainsi l'algorithme $A'_{rec}(3, \sigma, 3)$. La description générale de l'algorithme $A'_{rec}(p, \sigma, p)$ s'en déduit aisément, mais l'expérience prouve qu'augmenter d'avantage la valeur de $p = s$ n'est pas efficace. Bien entendu les versions duale et de décodage correspondantes sont aussi envisageables.

C'est pourquoi nous utilisons la notation Stern($2 \times p$,ℓ,$2 \times p$). En effet, le paramètre ℓ de l'algorithme de Stern a le même rôle que notre paramètre σ. En outre, le flou introduit dans la fenêtre d'information par l'algorithme de Stern correspond bien à $2 \times p$ erreurs.

II.2.2.c Facteur de travail

II.2.2.c.a Principe de calcul

Le facteur de travail W est une estimation moyenne du nombre d'opérations élémentaires qu'effectue un algorithme. Pour le type d'algorithme qui nous occupe, cette estimation peut s'effectuer en calculant d'une part le nombre moyen $\bar{N}$ d'itérations à effectuer et le nombre d'opérations Q effectuées lors d'une itération. On a alors

$$W = \bar{N} \times Q.$$

Le nombre moyen d'itérations peut se calculer à l'aide d'outils classiques de probabilités. Il est aussi possible d'estimer le nombre moyen d'opérations par itération pour les algorithmes précédents [5-Cha94, 7-CC95a]. Toutefois, ces estimations ne sont que partiellement confirmées par l'expérience. On observe toujours de légers décalages et les paramètres optimaux d'un algorithme s'avèrent varier d'une machine à l'autre.

En effet, contrairement au nombre d'itérations de l'algorithme, qui s'appuie sur une estimation mathématique, la réalité informatique dépend de facteurs non contrôlés qui peuvent parfois conduire à des résultats surprenants.

Nous adoptons donc ici un point de vue plus expérimental qui consiste à mesurer le temps moyen Q d'une itération et à faire varier les paramètres de l'algorithme afin d'obtenir le meilleur facteur de travail W possible.

II.2.2.c.b Processus stochastique

Définition 130. On dit que $\{X_i\}_{i\in\mathbb{N}}$ est un *processus stochastique* si et seulement s'il existe pour tout entier i une *distribution de probabilités* π_i telle que

$$\forall u \in \mathcal{E},\ \mathrm{Prob}\{X_i = u\} = \pi_i(u).$$

Ceci implique en particulier que $\sum_{u\in\mathcal{E}} \pi_i(u) = 1$. Un processus stochastique $\{X_i\}_{i\in\mathbb{N}}$ est un *processus stochastique indépendant* si et seulement si pour toute itération i, pour toute suite $(u_1, \ldots, u_i)$ d'éléments de $\mathcal{E}$, la probabilité $\pi_i(u_i)$ est indépendante des itérations précédentes, c'est-à-dire

$$\mathrm{Prob}\{X_i = u_i \,/\, X_{i-1} = u_{i-1}, X_{i-2} = u_{i-2}, \ldots, X_0 = u_0\} = \mathrm{Prob}\{X_i = u_i\}.$$

II.2.2.c.c Nombre moyen d'itérations

Nous pouvons scinder l'espace d'états d'un algorithme en deux sous-espaces $\mathcal{S}$ et $\mathcal{F}$. Les états de $\mathcal{S}$ sont ceux qui conduisent à l'obtention du mot c ($\mathcal{S}uccess$) par opposition à ceux de l'espace $\mathcal{F}$ ($\mathcal{F}ailure$). Une fois parvenu dans un état $\mathcal{S}$, l'algorithme s'arrête.

Théorème 131. *Le nombre moyen d'itérations d'un processus stochastique indépendant d'espace d'états $\mathcal{E} = \mathcal{S} \cup \mathcal{F}$ et de distribution de probabilités $\pi_0(u)$ est*

$$\bar{N} = \frac{1}{\sum_{u\in\mathcal{S}} \pi_0(u)}.$$

Preuve. Chaque itération étant indépendante des précédentes, on peut calculer la probabilité de succès d'une itération.

$$\pi = \sum_{u\in\mathcal{S}} \pi_0(u).$$

Alors, le nombre moyen d'itérations peut se calculer aisément

$$\begin{aligned}
\bar{N} &= \sum_{n\geq 1} n\mathrm{Prob}\{N = n\} \\
&= \sum_{n\geq 1} n\left(\mathrm{Prob}\{N \geq n-1\} - \mathrm{Prob}\{N \geq n\}\right) \\
&= 1 + \sum_{n>0} \mathrm{Prob}\{N \geq n\} \\
&= \sum_{n\geq 0} (1-\pi)^n \\
&= \frac{1}{\pi}.
\end{aligned}$$

□

II.2.2.d Probabilité de succès

Pour toutes nos estimations, nous considérons que le code $[n, k]$ que nous étudions possède un mot c de poids w et un seul. Bien que le calcul soit donné dans le cadre de la recherche d'un mot de poids faible à partir d'une matrice génératrice, il est clair que les résultats restent inchangés pour les versions duale ou de décodage des algorithmes.

Nous allons considérer pour chaque itération i de nos algorithmes la variable X_i correspondant au nombre de coordonnées non nulles de c situées dans la fenêtre d'information de cette itération. Dans un souci de généralisation, nous pouvons considérer que tous nos algorithmes ont w états de succès $\mathcal{S} = \{(0)_{\mathcal{S}}, \ldots, (w)_{\mathcal{S}}\}$ et w états d'échec $\mathcal{F} = \{(0)_{\mathcal{F}}, \ldots, (w)_{\mathcal{F}}\}$. Pour tout u compris entre 0 et w, on peut calculer la probabilité conditionnelle de succès

$$\beta(u) = \text{Prob}\{X_i = (u)_{\mathcal{S}} \,/\, X_i = u\}.$$

Pour certaines valeurs de u, cette probabilité peut bien entendu s'annuler, ou au contraire valoir 1 selon les algorithmes. Alors, la probabilité de succès d'un algorithme s'écrit

$$\pi = \sum_{u=0}^{w} \beta(u) \text{Prob}\{X_i = u\}.$$

Lemme 132. *Soit m, un mot de ν éléments, de poids ω. Le nombre de sous-ensembles à κ éléments d'un ensemble à ν élément étant noté $\binom{\nu}{\kappa} = \frac{\nu!}{\kappa!(\nu-\kappa)!}$, la probabilité qu'une sélection de κ colonnes contiennent ρ éléments de m non nuls est notée $\left\{ {\nu, \omega \atop \kappa, \rho} \right\}$ et vaut*

$$\left\{ {\nu, \omega \atop \kappa, \rho} \right\} = \frac{\binom{\omega}{\rho}\binom{\nu-\omega}{\kappa-\rho}}{\binom{\nu}{\kappa}}.$$

Nous allons faire un usage intensif de ce lemme. En particulier, on a

$$\text{Prob}\{X_i = u\} = \left\{ {n, w \atop k, u} \right\}.$$

II.2.2.d.a Algorithme A

Considérons l'algorithme décrit figure 2.4. Le mot de code c de poids faible w sera détecté à une itération i si et seulement si :

1. La fenêtre d'information sélectionnée de k colonnes comprend au moins une erreur et au plus p erreurs[1].
2. Globalement, les $k + \sigma$ colonnes sélectionnées comprennent au plus s erreurs.

Tous les états u pour $u > p$ ou $u = 0$ sont donc des états d'echec et on a $\beta(u) = 0$. Dans les autres cas on a la probabilité conditionnelle de succès

$$\forall u,\ 1 \leq u \leq p,\ \beta_{A(p,\sigma,s)}(u) = \sum_{v=\max(0,\sigma-[(n-k)-(w-u)])}^{\min(\sigma,s-u)} \left\{ {n-k, w-u \atop \sigma, v} \right\}.$$

On en déduit la probabilité de succès de l'algorithme A

$$\pi(n,k,w)_{A(p,\sigma,s)} = \sum_{u=1}^{p} \left[\left\{ {n, w \atop k, u} \right\} \sum_{v=\max(0,\sigma-[(n-k)-(w-u)])}^{\min(\sigma,s-u)} \left\{ {n-k, w-u \atop \sigma, v} \right\} \right]. \quad \text{(II.2.1)}$$

II.2.2.d.b Algorithme A'

Dans le cas de l'algorithme A', toutes les combinaisons linéaires de p lignes sont considérées. On a donc les conditions de succès suivantes :

1. La fenêtre d'information sélectionnée de k colonnes comprend un nombre d'erreurs compris entre 1 et p.

1. Puisque nous avons une fenêtre d'information et que le mot c est une combinaison linéaire des lignes de la matrice génératrice, il doit posséder au moins une coordonnée non nulle dans la fenêtre d'information. Dans le cas de l'algorithme de décodage, on pourrait par contre avoir une fenêtre d'information ne contenant aucune erreur. En fait ceci oblige au niveau de l'implantation à un test supplémentaire coûteux pour un gain en probabilité de succès négligeable.

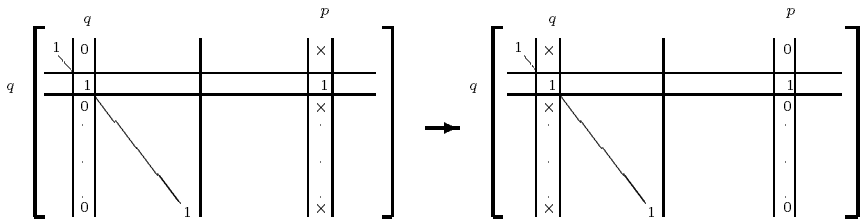


FIG. 2.5 – *Transition entre fenêtres d'informations proches (cas binaire)*

2. Les σ colonnes supplémentaires choisies n'en contiennent aucune.

On a donc

$$\begin{aligned}
\beta_{A'(p,\sigma,p)}(0) &= 0 \\
\beta_{A'(p,\sigma,p)}(u) &= \left\{ \begin{matrix} n-k, w-u \\ \sigma, 0 \end{matrix} \right\} \text{ pour tout } u,\ 1 \leq u \leq p \\
\beta_{A'(p,\sigma,p)}(u) &= 0 \text{ pour tout } u > p
\end{aligned}$$

II.2.3 Algorithme itératif

NOUS PRÉSENTONS MAINTENANT UNE AMÉLIORATION des algorithmes existant. Cette idée a été introduite initialement par Omura [4-Omu72], reprise par van Tilburg pour l'algorithme de McEliece [5-Til88], puis pour l'Algorithme de Lee-Brickell [7-CC93, 2-Til94, 6-CC94]. Elle s'applique à tous les algorithmes précédents.

II.2.3.a Principe

Définition 133. Deux fenêtres d'information $\mathcal{I}$ et $\mathcal{I}'$ d'une matrice G sont *proches* si et seulement s'il existe un entier $q \in \mathcal{I}$ et un entier $p \in \mathcal{N} \setminus \mathcal{I}$ tel que

$$\mathcal{I}' = (\mathcal{I} \setminus \{q\}) \cup \{p\}.$$

Deux fenêtres d'information quelconques peuvent toujours être reliées par une séquence de fenêtres d'informations proches. C'est pourquoi nous utiliserons désormais cette méthode itérative dans les algorithmes précédents pour faire varier la fenêtre d'information.

Théorème 134. *Soit $\mathcal{I}$ une fenêtre d'information d'une matrice G, et $G = (V, W)_{\mathcal{I}}$ la décomposition associée. Soit $Z = V^{-1}W$ la matrice de coefficients $(z_{i,j})_{\substack{i \in \mathcal{I} \\ j \in \mathcal{N} \setminus \mathcal{I}}}$ Soit q un entier de $\mathcal{I}$ et p un entier de $\mathcal{N} \setminus \mathcal{I}$, alors $\mathcal{I}' = (\mathcal{I} \setminus \{q\}) \cup \{p\}$ est une fenêtre d'information si et seulement si le coefficient $z_{q,p}$ est non nul.*

Preuve. On a $VZ = W$, donc si on note $(G_i)_{i \in \mathcal{N}}$ les vecteurs colonnes de G, on a

$$\forall j \in \mathcal{N} \setminus \mathcal{I}, G_j = \sum_{i \in \mathcal{I}} z_{i,j} G_i.$$

En particulier, pour $j = p$, on a

$$G_p = z_{q,p} G_q + \sum_{i \in \mathcal{I} \setminus \{q\}} z_{i,p} G_i.$$

Comme les colonnes indexées par $\mathcal{I}$ sont linéairement indépendantes, on en déduit que les colonnes indexées par $\mathcal{I}' = (\mathcal{I} \setminus \{q\}) \cup \{p\}$ le sont si et seulement si $z_{q,p}$ est non nul. □

II.2.3.b Algorithme itératif général

Ce résultat permet de préciser dans les algorithmes ci-dessus la manière efficace qui va pouvoir être employée pour effectuer la transition entre deux fenêtres d'informations proches. Soit $\mathcal{I}_i$ la fenêtre d'information utilisée à l'itération i d'un algorithme. Tous les algorithmes utilisent une forme systématique associée à $\mathcal{I}_i$ de la matrice utilisée. Notons A_i la matrice complémentaire de l'identité associée à cette forme systématique. La transition de la fenêtre d'information $\mathcal{I}_i$ à la fenêtre d'information $\mathcal{I}_{i+1}$ s'effectue en réalisant une permutation des colonnes d'indice p et q (voir figure 2.5). On réalise ensuite un pivot de Gauss pour retrouver une forme systématique. Bien entendu, l'intérêt de la méthode itérative est que le pivot peut s'effectuer en place. On ne conserve donc en mémoire que la matrice complémentaire de l'identité.

Le reste des algorithmes, c'est-à-dire la recherche heuristique de mots de poids faible, reste inchangé. Nous aurons donc les mêmes variantes d'algorithme que nous noterons par analogie B_{rec}, B_{dec}, $B_\perp$, B'_{dec}, B'_{rec}, $B'_\perp$.

II.2.3.c Temps de calcul

II.2.3.c.a Facteur de travail modifié

Le facteur de travail de cette famille d'algorithmes reste inchangé dans son principe :

$$W = N \times Q.$$

Il nous faut toutefois réévaluer le nombre moyen d'itérations de ces algorithmes. En effet le résultat du paragraphe II.2.2.c.c n'est plus valable car les itérations successives de l'algorithme ne peuvent plus être considérées comme indépendantes. Ceci nous amène à introduire et à utiliser d'autres outils classiques de probabilités [6-Neu90, 2-KS60].

II.2.3.c.b Chaîne de Markov

Définition135. Un processus stochastique $\{X_i\}_{i\in\mathbb{N}}$ est une *chaîne de Markov* si et seulement si pour toute itération i, pour toute suite $(u_1, \ldots, u_i)$ d'éléments de $\mathcal{E}$, la probabilité $\pi_i(u_i)$ dépend uniquement de l'itération précédente, c'est-à-dire

$$\text{Prob}\{X_i = u_i \,/\; X_{i-1} = u_{i-1}, X_{i-2} = u_{i-2}, \ldots, X_0 = u_0\} = \text{Prob}\{X_i = u_i \,/\; X_{i-1} = u_{i-1}\}.$$

Nous nous plaçons désormais dans le cas exclusif d'un espace d'états fini.

Définition136. Soit P, une matrice de coefficients réels positifs ou nuls $(p_{u,v})_{(u,v)\in\mathcal{E}^2}$. Alors P est une *matrice markovienne* si et seulement si pour tout $u \in \mathcal{E}$ on a

$$\sum_{v\in\mathcal{E}} P_{u,v} = 1.$$

On appelle chaîne de Markov stationnaire un processus stochastique qui vérifie pour toute itération i et tout couple $(u, v) \in \mathcal{E}^2$

$$\text{Prob}\{X_i = v \,/\; X_{i-1} = u\} = P_{u,v}.$$

Une chaîne de Markov stationnaire est donc entièrement définie par sa distribution de probabilités initiale π_0 et sa *matrice de transition markovienne* P.

II.2.3.c.c Nombre moyen d'itérations

Nous pouvons comme précédemment scinder l'espace d'états d'un algorithme en deux sous-espaces $\mathcal{S}$ et $\mathcal{F}$. Une fois parvenu dans un état $\mathcal{S}$, l'algorithme ne change plus d'état. Dans la matrice markovienne associée, les états u de $\mathcal{S}$ correspondent donc aux lignes de la matrice qui ne contiennent qu'un seul élément non nul $P_{u,u} = 1$.

Définition137. Soit $\{X_i\}_{i\in\mathbb{N}}$ une chaîne de Markov stationnaire de matrice de transition markovienne P, nous appelons *état absorbant* un état u qui vérifie la propriété ci-dessus que nous rappelons de manière formelle

$$P_{u,u} = 1 \text{ et } \forall v \in \mathcal{E},\ v \neq u,\ P_{u,v} = 0.$$

On appelle *chaîne de Markov finie* une chaîne de Markov stationnaire telle que tout état puisse être atteint et qui possède au moins un état absorbant.

Théorème 138 [2-KS60]. *Soit $\{X_i\}_{i\in\mathbb{N}}$ une chaîne de Markov finie de matrice de transition markovienne P. Soit $\mathcal{S}$ l'espace de ses états absorbants, $\mathcal{F}$ son complémentaire, et Q la matrice extraite de P correspondant à $\mathcal{F}$*

$$Q = (P_{u,v})_{(u,v)\in\mathcal{F}^2}.$$

Alors la matrice $Id_{|\mathcal{F}|} - Q$ a un inverse R appelé matrice fondamentale *de la chaîne de Markov finie. En outre*

$$R = (Id - Q)^{-1} = \sum_{m=0}^{\infty} Q^m.$$

Corollaire 139. *Soit $(\pi_0(u))_{u\in\mathcal{E}}$ la distribution de probabilités initiale de la chaîne de Markov finie, alors le nombre moyen d'itérations effectuées avant de se retrouver dans un état absorbant est*

$$\bar{N} = 1 + \sum_{u\in\mathcal{F}} \pi_0(u) \sum_{v\in\mathcal{F}} R_{u,v}$$

où $R_{u,v}$ est le coefficient de la u^e ligne et de la v^e colonne de la matrice fondamentale de la chaîne de Markov finie.

Preuve. Le nombre moyen d'itérations est classiquement

$$\begin{aligned}\bar{N} &= 1 + \sum_{n>0} \text{Prob}\{N \geq n\} \\ &= 1 + \sum_{n\geq 0} \text{Prob}\{X_n \in \mathcal{F}\}\end{aligned}$$

Nous pouvons faire intervenir l'état initial de la chaîne de Markov finie. Alors

$$\bar{N} = 1 + \sum_{n\geq 0} \sum_{u\in\mathcal{F}} \text{Prob}\{X_n \in \mathcal{F} / \ X_0 = u\}\pi_0(u).$$

Soit $Q = (P_{u,v})_{(u,v)\in\mathcal{F}^2}$, alors

$$\text{Prob}\{X_i \in \mathcal{F} / \ X_0 = u\} = \sum_{v\in\mathcal{F}} (Q^i)_{u,v}.$$

On en déduit

$$\begin{aligned}\bar{N} &= 1 + \sum_{n\geq 0} \sum_{u\in\mathcal{F}} \sum_{v\in\mathcal{F}} (Q^i)_{u,v}\pi_0(u) \\ &= 1 + \sum_{u\in\mathcal{F}} \pi_0(u) \sum_{v\in\mathcal{F}} (\sum_{n\geq 0} Q^i)_{u,v} \\ &= 1 + \sum_{u\in\mathcal{F}} \pi_0(u) \sum_{v\in\mathcal{F}} R_{u,v}\end{aligned}$$

□

Ce résultat permet donc de calculer le nombre moyen d'itérations des algorithmes de type B connaissant la matrice de transition markovienne de la chaîne de Markov finie. Nous allons maintenant préciser ces matrices de transition pour les différents algorithmes. Comme pour les algorithmes indépendants, il est clair que toutes les versions d'un algorithme ont même matrice de transition markovienne. Nous n'avons donc à étudier que les deux cas des algorithmes B et B'.

Cette méthode de calcul conduit à des résultats exacts, contrairement à ceux de van Tilburg [2-Til94], et s'applique à tous les algorithmes à fenêtre d'information.

Algorithme itératif général

Les critères de l'algorithme B sont les mêmes que ceux de l'algorithme A (voir paragraphe II.2.2.d.a). Nous introduisons donc le même espace d'états que précédemment $\mathcal{E} = \mathcal{S} \cup \mathcal{F}$ avec

$$\begin{aligned} \mathcal{S} &= \{(0)_{\mathcal{S}}, \ldots, (w)_{\mathcal{S}}\} \\ \mathcal{F} &= \{(0)_{\mathcal{F}}, \ldots, (w)_{\mathcal{F}}\} \end{aligned}$$

Il nous faut déterminer les éléments $(P_{u,v})_{(u,v)\in\mathcal{F}^2}$ de la matrice de transition markovienne. Par abus de notation nous écrivons

$$P_{u,v} = \text{Prob}\{X_{i+1} = (v)_{\mathcal{F}} \,/\, X_i = (u)_{\mathcal{F}}\}.$$

Tout d'abord, on peut noter que la matrice est tri-diagonale. En effet, puisqu'à chaque itération on n'effectue qu'une seule permutation, le nombre d'erreur dans la fenêtre d'information ne peut varier que d'une unité à la fois. Plus précisément, on a

$$\begin{aligned} \text{Prob}\{X_{i+1} = u \,/\, X_i = u\} &= \frac{k-u}{k} \times \frac{(n-k)-(w-u)}{n-k} + \frac{u}{k} \times \frac{w-u}{n-k}, \\ \text{Prob}\{X_{i+1} = u-1 \,/\, X_i = u\} &= \frac{u}{k} \times \frac{(n-k)-(w-u)}{n-k}, \\ \text{Prob}\{X_{i+1} = u+1 \,/\, X_i = u\} &= \frac{k-u}{k} \times \frac{w-u}{n-k}, \\ \text{Prob}\{X_{i+1} = v \,/\, X_i = u\} &= 0,\ \forall v \in \mathcal{N} \setminus \{u-1, u, u+1\}. \end{aligned}$$

Alors, si nous reprenons la probabilité conditionnelle de succès

$$\beta(u) = \text{Prob}\{X_i = (u)_{\mathcal{S}} \,/\, X_i = u\},$$

on a

$$\begin{aligned} P_{u,v} &= \text{Prob}\{X_{i+1} = (v)_{\mathcal{F}} \,/\, X_i = (u)_{\mathcal{F}}\}, \\ &= \text{Prob}\{X_{i+1} = (v)_{\mathcal{F}} \,/\, X_{i+1} = v\}\text{Prob}\{X_{i+1} = v \,/\, X_i = u\}\text{Prob}\{X_i = u \,/\, X_i = (u)_{\mathcal{F}}\}, \\ &= (1-\beta(v))\text{Prob}\{X_{i+1} = v \,/\, X_i = u\}. \end{aligned}$$

On a bien une matrice tri-diagonale de coefficients :

$$\begin{aligned} P_{u,u} &= (1-\beta(u)) \left[\frac{k-u}{k} \times \frac{(n-k)-(w-u)}{n-k} + \frac{u}{k} \times \frac{w-u}{n-k}\right], \\ P_{u,u-1} &= (1-\beta(u-1)) \left[\frac{u}{k} \times \frac{(n-k)-(w-u)}{n-k}\right], \\ P_{u,u+1} &= (1-\beta(u+1)) \left[\frac{k-u}{k} \times \frac{w-u}{n-k}\right], \\ P_{u,v} &= 0\ \forall v \in \mathcal{F} \setminus \{u-1, u, u+1\}. \end{aligned}$$

Selon l'algorithme considéré, on utilisera donc les probabilités conditionnelles de l'algorithme indépendant correspondant, qui ont été calculées paragraphe II.2.2.d.a.

II.2.4 Implantation

II.2.4.a Optimisation des paramètres

Nous avons implanté les différents algorithmes décrits en utilisant la librairie ZEN. Cette implantation auto-ajuste les paramètres des algorithmes pour optimiser ceux-ci. À titre de comparaison, nous reprenons figure 2.6 un exemple utilisé dans [7-CC95a]. La première ligne correspond à l'algorithme de Leon optimal

Machine utilisée	Algorithme utilisé	Nombre moyen d'itérations	Temps de calcul	
			par itér.	moyen
Sparc 10	Leon(2,4,2)	292	23 ms	6.6 s
	Stern(2×2,10,2×2)	113	48 ms	5.4 s
	A(2,8,4)	211	23 ms	4.8 s
	A'(3,7,3)	82	28 ms	2.3 s
Sparc 4	A(2,9,4)	213	27 ms	5.8 s
	A'(3,7,3)	82	36 ms	2.9 s

FIG. 2.6 – *Recherche d'un mot de poids 14 dans un code* $[256, 129]$ *(Algorithmes indépendants)*

Machine utilisée	Algorithme utilisé	Nombre moyen d'itérations	Temps de calcul	
			par itér.	moyen
Sparc 10	B(1,2,1)	20457	0.23 ms	4.7 s
	B'(2,7,2)	3666	0.55 ms	2.0 s
Sparc 4	B(1,2,1)	20457	0.28 ms	5.8 s
	B'(2,8,2)	3718	0.82 ms	3.0 s

FIG. 2.7 – *Recherche d'un mot de poids 14 dans un code* $[256, 129]$ *(Algorithmes itératifs)*

qui est aussi l'algorithme $A(2, 4, 2)$ et la deuxième à l'algorithme de Stern initialement décrit, en utilisant les paramètres optimaux. On constate ici expérimentalement que l'algorithme A améliore l'algorithme de Stern. Ceci provient uniquement du fait que les opérations effectuées à chaque itération sont plus simples et que l'utilisation d'une grande quantité de mémoire, nécessaire pour l'algoStern, dégrade les performances théoriques de cet algorithme. À l'opposé, on constate que l'algorithme A' améliore sensiblement le temps de calcul moyen pour résoudre ce problème. En effet, le gros avantage de l'algorithme A' est qu'il utilise peu de mémoire ce qui permet de tirer profit de celle-ci sans trop perdre en efficacité pratique.

D'autre part on peut constater figure 2.6 que les paramètres optimaux varient en fait d'une machine à l'autre. Néanmoins, quelle que soit la machine, l'algorithme A' reste le meilleur des algorithmes à itérations indépendantes.

Nous avons aussi implanté les algorithmes itératifs associés. Nous utilisons des structures matricielles de type ligne pour la version standard des algorithmes qui utilisent une matrice génératrice, et des structures matricielles de type colonne pour la version duale. Toutefois, l'opération d'élimination gaussienne doit s'effectuer impérativement par des opérations sur les lignes de la matrice. Mais dans le cas binaire, l'opération d'élimination gaussienne peut s'effectuer par des manipulations de colonnes. En effet, il est clair sur la figure 2.5 qu'à l'issue de l'élimination gaussienne la colonne d'indice p et la ligne d'indice q sont les seules qui resteront inchangées. Les lignes d'indice i dont le coefficient dans la colonne p vaut 1 sont additionnées de la ligne d'indice q. Il est facile de voir que cette opération revient à additionner la colonne d'indice p aux colonnes d'indice j dont le coefficient dans la ligne q vaut 1. Le caractère presque miraculeux de cette observation provient simplement du fait que dans $GF(2)$ l'inversion et la négation sont des opérations qui ne changent pas leurs arguments.

La figure 2.7 donne les résultats obtenus et montre une nouvelle amélioration des performances. Cette amélioration reste toutefois limitée car elle est compensée par l'optimisation des paramètres. Sur certaines machines, les résultats obtenus par les deux types d'algorithme sont très proches.

II.2.4.b Parallélisation des attaques

Une idée simple évoquée dans [5-Cha94] et reprise dans [2-Til94] consiste à factoriser les attaques. En effet, il est clair que dans la pratique c'est à plusieurs instances d'un même problème que l'on peut s'attaquer. Ainsi, dans le système de Stern, on peut chercher à trouver un secret correspondant à une clef publique prise dans une liste de plusieurs clefs. Or dans tous nos algorithmes, rien n'empêche dans l'étape de vérification de la fenêtre d'information, de considérer successivement N mots à décoder. On utilise ainsi une même

Machine utilisée	Algorithme utilisé	Nombre moyen d'itérations		Temps de calcul	
		théorique	mesuré	par itér.	moyen
Sparc 4	A(2,11,5)	207	205	20 ms	4.2 s
	A'(3,8,3)	91	89	26 ms	2.4 s

FIG. 2.8 – *Recherche d'un mot de poids 14 dans un code* [256, 129] *(Algorithmes non prouvés)*

élimination gaussienne pour plusieurs décodages. En fait, on se rend compte facilement que le gain obtenu va très vite s'équilibrer avec le nombre supplémentaire d'opérations à effectuer à chaque itération. Il n'est donc pas nécessaire d'augmenter le nombre N au delà de quelques dizaines.

van Tilburg décrit dans [2-Til94] une parrallélisation de son algorithme de décodage inspirée de cette méthode, mais qui s'applique à un seul syndrome à la fois. Cette parallélisation est en fait largement inefficace. En effet, si on reprend les résultats de van Tilburg, par exemple dans le cas du système de Stern on constate que l'algorithme initial a un nombre d'itérations de $2^{57.0}$ et un nombre d'opérations estimé de $2^{72.9}$. D'autre part l'algorithme parallélisé sur 2^{31} processeurs (paramètres optimaux selon van Tilburg) nécessite encore $2^{43.5}$ itérations et $2^{58.5}$. Or la simple distribution sur 2^{14} machines du problème conduirait au même résultat, et il est quand même plus "facile" de gérer un réseau de 16000 machines avec des processus indépendants, que de construire et programmer une machine parallèle à 2 milliards de processeurs !

II.2.4.c Amélioration non prouvée des algorithmes indépendants

Le problème des générateurs pseudo-aléatoires a été largement étudié. Bien que les générateurs classiques soient "suffisamment aléatoires" pour nos applications, on peut envisager l'amélioration des algorithmes couramment implantés. Dans les études évoquées [4-AGH+90, 4-LS92], on peut remarquer les propriétés liées au code simplexe qui est souvent utilisé comme sous-code d'un code concaténé pour obtenir un générateur pseudo-aléatoire satisfaisant de bonnes conditions.

L'idée nous est donc venue, dans le cas de nos algorithmes d'utiliser aussi ces propriétés du simplexe. Supposons que le code considéré G soit de dimensions $[2^a, 2^{a-1}]$, alors il existe un code de même longueur et de dimension $a-1$ dont les 2^{a-1} mots sont de poids 2^{a-1}. Ce code est le code simplexe. On peut considérer que chacun de ces mots définit une fenêtre d'information dans le code G. En outre, le passage entre deux fenêtres d'information du simplexe s'effectue par seulement 2^{a-2} permutations, et l'élimination gaussienne qui en résulte en est simplifiée d'autant. Or expérimentalement, sur 1000 tentatives, le nombre moyen de fenêtres d'information à considérer pour les algorithmes indépendants n'est pas modifié si on inclut pour chaque sélection aléatoire la vérification des 2^{a-1} fenêtres d'information associées définie par le simplexe (voir figure 2.8). Cette astuce s'étend aisément aux codes de dimensions différentes toujours sans modification du nombre moyen d'itérations.

II.2.5 Application aux codes BCH de longueur 511

LES CODES BCH DE LONGUEUR 511 ont été étudiés par de nombreux auteurs [4-KL72, 4-ACN92, 4-HS73, 2-PW80, 4-KT69]. Leur distance minimale réelle a en particulier été déterminée dans presque tous les cas. Seuls douze codes BCH de longueur 511 conservaient leur mystère, à savoir ceux de distances construites 29, 37, 41, 43, 51, 59, 61, 75, 77, 85, 87, 107.

L'utilisation par Anne Canteaut de l'algorithme de Stern itératif a permis de déterminer que les codes BCH de distances construites 29, 37, 41, 43 et 87 atteignent leur distance construite. Ce résultat a fait l'objet de [4-CC96].

En utilisant l'algorithme B', nous avons pu de même montrer que le code BCH de longueur 511 et de distances construites 51, atteint sa distance construite. En effet, si $GF(2^9)$ est défini par $X^9 + X^4 + 1$, alors le polynôme binaire dont les coefficients non nuls sont de degrés (6, 10, 11, 17, 22, 54, 57, 64, 76, 79, 85, 87, 93, 97, 101, 121, 122, 139, 140, 144, 154, 171, 177, 182, 198, 258, 287, 290, 294, 299, 309, 313, 333, 335, 350, 359, 361, 369, 370, 371, 395, 399, 405, 412, 435, 437, 452, 469, 474, 488, 491, 508) est de poids 52 et

appartient au code BCH de distance construite 51. Or il est bien connu que si un code BCH primitif au sens strict contient un mot de poids pair w, alors il contient aussi un mot de poids $w-1$.

Il semble que nous atteignions là les limites des possibilités de nos algorithmes, et que d'autres méthodes soient à rechercher pour les six derniers codes de distance minimale réelle inconnue. En particulier, il est vraisemblable que le code BCH de distance construite 107 n'atteigne pas sa distance construite, car dans ce cas, ses dimensions réduites auraient du nous permettre de lever le doute. De même, nos algorithmes ont été appliqués au code BCH de distance construite 85. Or, même au terme de deux ans de calcul cumulé sur une cinquantaine de machines rapides (sparc 5 ou équivalente), seuls des mots de poids 87 et 88 ont été trouvés.

Chapitre II.3

Décodage général des codes correcteurs d'erreur – cas de la distance de Gabidulin

Nous nous intéressons désormais au cas des codes linéaires utilisant la distance de Gabidulin. Il s'agit là d'un travail, commun avec Jacques Stern, dont le but était, là encore, la cryptanalyse de systèmes présentés au paragraphe III.1.5 [5-CS96]. Dans un premier temps, nous allons nous intéresser à la recherche de mots de rang fixé dans un code linéaire (II.3.1). L'algorithme proposé permet au passage de déduire une borne sur les dimensions que doivent respecter les codes linéaires en distance de Gabidulin (II.3.1.b.a). Il peut ensuite s'adapter au problème du décodage à distance minimale de Gabidulin dans le cas général (II.3.2). Nous finissons en décrivant plus précisément la méthode d'énumération des bases implantée à l'aide de Zen (II.3.3).

II.3.1 Recherche de mot de rang faible

De la même manière que dans le cas de la distance de Hamming, il existe un lien entre le problème du décodage général à distance minimale de Gabidulin et celui de la recherche d'un mot de rang faible. Dans le cas de la distance de Hamming, les algorithmes de décodage général présentés peuvent être vus comme des adaptations d'algorithmes de recherche. Il en est de même dans le cas de la distance de Gabidulin.

II.3.1.a Problème

Dans tout ce qui suit, nous considérons un code linéaire $[n, k]$ décrit par une matrice de contrôle H définie sur un corps fini $GF(q^m)$. Étant donné un entier $r \leq m$, le problème est de trouver un mot de code $\bar{s}$ de n éléments dans $GF(q^m)$, de rang r. Ce vecteur s vérifie donc l'équation

$$H\bar{s} = 0. \tag{II.3.1}$$

II.3.1.b Principe de l'algorithme

Nous supposons l'existence d'une solution $\bar{s}$ de rang r à l'équation (II.3.1). Alors, il existe r éléments $\theta_0, \ldots, \theta_{r-1}$ de $GF(q^m)$, linéairement indépendants dans $GF(q)$, et nr coefficients $\alpha_{j,k} \in GF(q)$, tels que pour tout j, $1 \leq j \leq n$, on ait

$$s_j = \sum_{k=0}^{r-1} \alpha_{j,k}\theta_k.$$

Soient $(h_{i,j})_{\substack{1 \leq i \leq n-k \\ 1 \leq j \leq n}}$ les coefficients de la matrice H. Alors l'équation (II.3.1) permet d'obtenir un système de $(n-k)$ relations sur $GF(q^m)$.

$$\forall i,\ 1 \leq i \leq n-k,\ \sum_{j=1}^{n} \sum_{k=0}^{r-1} h_{i,j} \alpha_{j,k} \theta_k = 0. \tag{II.3.2}$$

Dès que les $(\theta_0, \ldots, \theta_{r-1})$ sont connus, le système ci-dessus fournit un système linéaire redondant dans $GF(q)$, à nr inconnues $(\alpha_{j,k})_{\substack{1 \leq j \leq n \\ 0 \leq k \leq r-1}}$ et au plus $(n-k)m$ équations indépendantes.

Dans son principe, notre algorithme consiste donc à énumérer toutes les bases $(\theta_0, \ldots, \theta_{r-1})$ sur $GF(q^m)$ et à essayer de résoudre le système linéaire défini sur $GF(q)$ qui découle du système (II.3.2).

II.3.1.b.a Borne "à la Singleton"

Fixons pour un temps les r éléments $(\theta_0, \theta_1, \ldots, \theta_{r-1})$ de $GF(q^m)$. Le système (II.3.2) nous donne des solutions de l'équation (II.3.1) dès qu'il possède plus d'inconnues que d'équations, c'est-à-dire lorsque $nr > (n-k)m$. Dans tout code linéaire de dimensions $[n, k]$, cette inégalité entraîne que l'on peut trouver un mot de code de rang r si

$$r \geq \frac{(n-k)m+1}{n}.$$

Par suite, la distance minimale de Gabidulin d de tout code linéaire $[n, k]$ est majorée par

$$d \leq \left\lceil \frac{(n-k)m+1}{n} \right\rceil \tag{II.3.3}$$

On en déduit le théorème suivant

Théorème 140. *Il n'existe pas dans $GF(q^m)$ de codes MRD de longueur $n > m$.*

Preuve. Un code MRD atteint la borne de Singleton $d = n - k + 1$. Or on peut trouver un mot de code de rang $r = n - k$ dès que

$$\begin{array}{rcl} n-k & \geq & \dfrac{(n-k)m+1}{n} \\ n(n-k) & \geq & (n-k)m+1 \\ n(n-k) & > & m(n-k) \\ n & > & m \end{array}$$

□

Ceci signifie que notre borne (II.3.3) est meilleure que la borne de Singleton quand la longueur du code n est supérieure au degré m de l'extension polynomiale de définition du code. Ceci ne constitue pas une contradiction avec les codes MRD présentés par Gabidulin, puisque ce dernier se place exclusivement dans le cas où $n \leq m$ (voir chapitre II.1).

II.3.1.c Énumération sélective

Revenons à notre algorithme. Il y a au plus q^{mr} bases $(\theta_0, \ldots, \theta_{r-1})$ dans $GF(q^m)$. Même pour des codes de faibles dimensions, une énumération brutale risque donc de ne pas être possible. En fait, une analyse un peu plus fine du problème permet de limiter l'énumération à un nombre de bases bien plus faible.

Tout d'abord, on peut noter que pour tout $\theta \in GF(q^m)$, si $(s_1, \ldots, s_n)$ est solution de l'équation (II.3.1), alors $(\theta s_1, \ldots, \theta s_n)$ est aussi solution. On peut donc se limiter à l'énumération des bases de la forme $(1, \theta_1, \ldots, \theta_{r-1})$.

Soit $(b_1, \ldots, b_m)$ une base de $GF(q^m)$ dans $GF(q)$. Alors, pour tout élément B de $GF(q^m)$, il existe m éléments $\beta_1, \ldots, \beta_m$ de $GF(q)$ tels que

$$B = \sum_{\ell=1}^{m} \beta_\ell b_\ell.$$

Une telle base est par exemple obtenue par la représentation polynomiale de $GF(q^m)$. On a dans ce cas $b_\ell = X^{\ell-1}$. Nous allons noter cette représentation particulière de la façon suivante

$$B = [\beta_m \cdots \beta_1] = \beta_m X^{m-1} + \cdots + \beta_2 X + \beta_1,$$

et nous appelons *chiffres* les coefficients de cette représentation.

Comme $\theta_0 = 1$, nous avons $\theta_0 = [0 \cdots 01]$. Par suite, le dernier chiffre de chaque θ_i peut être fixé à zéro car

$$[\theta_{i,1} \cdots \theta_{i,m-1}\theta_{i,m}] = [\theta_{i,1} \cdots \theta_{i,m-1}0] + \theta_{i,m}[0 \cdots 01].$$

Ceci permet de réduire encore le nombre de bases à énumérer. Nous allons maintenant formaliser ces idées et calculer explicitement ce nombre.

Lemme 141 [2-LN86, page 455]. *Le nombre de matrices $m \times r$ de rang r sur $GF(q)$ est*

$$N_q(m,r) = q^{\frac{r(r-1)}{2}} \prod_{i=0}^{r-1} (q^{m-i} - 1).$$

Corollaire 142. *Le nombre $C_q(r)$ de matrices inversibles de dimension r sur $GF(q)$ est*

$$C_q(r) = q^{\frac{r(r-1)}{2}} \prod_{i=1}^{r} (q^i - 1).$$

Définition 143. Une *base stricte* de rang r, est une base $\Theta = (1, \theta_1, \ldots, \theta_{r-1})$ telle que les derniers chiffres des θ_i sont tous nuls.

Définition 144. Deux bases strictes Θ et Θ' sont *équivalentes* s'il existe une matrice inversible T sur $GF(q)$ de dimension r telle que

$$\Theta' = \begin{pmatrix} 1 \\ \theta'_1 \\ \vdots \\ \theta'_{r-1} \end{pmatrix} = T \begin{pmatrix} 1 \\ \theta_1 \\ \vdots \\ \theta_{r-1} \end{pmatrix} = T\Theta.$$

Il est clair qu'il suffit d'énumérer un élément dans chaque classe d'équivalence.

Lemme 145. *Le nombre de bases dans une classe d'équivalence est*

$$C_q(r-1) = q^{\frac{(r-1)(r-2)}{2}} \prod_{i=1}^{r-1} (q^i - 1).$$

Preuve. Soit T la matrice de transition markovienne entre deux bases strictes équivalentes Θ et Θ'. Alors, en utilisant une représentation par blocs de la matrice, on peut définir

$$T = \begin{pmatrix} \Delta & D \\ C & B \end{pmatrix}$$

avec

$$B = \begin{pmatrix} \beta_{1,1} & \cdots & \beta_{1,r-1} \\ \vdots & & \vdots \\ \beta_{r-1,1} & \cdots & \beta_{r-1,r-1} \end{pmatrix}, C = \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_{r-1} \end{pmatrix}, D = \begin{pmatrix} \delta_1 & \cdots & \delta_{r-1} \end{pmatrix},$$

avec $\Delta \in GF(q)$. Les matrices B, C et D sont à coefficients dans $GF(q)$.

Comme Θ' est une base stricte, elle vérifie $\theta'_0 = 1$. Par suite

$$\theta'_0 = [0 \cdots 01] = [0 \cdots 0\Delta] + \sum_{i=1}^{r-1} \delta_i [\theta_{i,m} \cdots \theta_{i,2}0].$$

Ceci nous donne un système linéaire redondant à m équations dans $GF(q)$ pour lequel les $r \leq m$ inconnues sont $\Delta, \delta_1, \ldots, \delta_{r-1}$. Ce système implique que $\Delta = 1$ et $D = 0$.

De plus, nous avons $\theta'_j = \gamma_j + \sum_{k=1}^{r-1} \beta_{j,k}\theta_k$. Comme le dernier chiffre de chaque θ'_i est nul, nous avons

$$[\theta'_{j,m} \cdots \theta'_{j,2} 0] = [0 \cdots 0 \gamma_j] + \sum_{k=1}^{r-1} \beta_{j,k} [\theta_{k,m} \cdots \theta_{k,2} 0],$$

d'où nous déduisons que le vecteur C est nul. Par suite, $\theta'_j = \sum_{k=1}^{r-1} \beta_{j,k}\theta_k$ et B doit être une matrice inversible sur $GF(q)$.

La matrice T est bien inversible et

$$T = \begin{pmatrix} 1 & 0 \\ 0 & B \end{pmatrix} \text{ et } T^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & B^{-1} \end{pmatrix}.$$

On déduit du corollaire 142 le nombre de matrices inversibles B qui est le nombre de matrices de transition T entre bases strictes équivalentes :

$$C_q(r-1) = q^{\frac{(r-1)(r-2)}{2}} \prod_{i=1}^{r-1} (q^i - 1).$$

□

Posons

$$D_q(m,r) = \frac{N_q(m,r)}{C_q(r)} = \frac{\prod_{i=m-(r-1)}^{m} (q^i - 1)}{\prod_{i=1}^{r} (q^i - 1)} = \prod_{i=1}^{r} \frac{q^{m-(i-1)} - 1}{q^i - 1}. \tag{II.3.4}$$

Le lemme 145 signifie qu'il suffit d'énumérer $D_q(m-1, r-1)$ bases strictes pour trouver une solution à l'équation (II.3.2). Ce nombre est aussi le nombre de codes distincts de dimensions $[m, r]$ sur $GF(q)$ (voir [2-MWS83, chapitre 15.2, théorème 9]).

Nous montrons maintenant qu'à une solution de l'équation (II.3.2) est associée une classe d'équivalence et une seule. Pour ce type d'algorithme, il n'y a donc pas de meilleure stratégie que l'énumération d'un représentant pour toutes les classes d'équivalence. Pour chacun de ces représentants, l'algorithme consiste à vérifier si le système linéaire correspondant à (II.3.2) dans $GF(q)$ a une solution.

Théorème 146. *Soit n un entier supérieur à r. Soient Θ et Θ' deux bases telles qu'il existe deux matrices $n \times r$ à coefficients dans $GF(q)$ A et A' de rang maximal r qui vérifient $A\Theta = A'\Theta'$. Alors Θ et Θ' sont équivalentes.*

Preuve. Comme A et A' sont de rang maximal, une élimination gaussienne permet d'obtenir deux matrices inversibles S et S' de dimensions n sur $GF(q)$ et deux matrices de permutation P et P' de taille r telles que, en représentant par blocs les matrices S et S', on ait

$$A = \begin{pmatrix} S_1 & S_2 \\ S_3 & S_4 \end{pmatrix} \begin{pmatrix} Id_r \\ 0 \end{pmatrix} P = \begin{pmatrix} S_1 \\ S_3 \end{pmatrix} P \text{ et } A' = \begin{pmatrix} S'_1 & S'_2 \\ S'_3 & S'_4 \end{pmatrix} \begin{pmatrix} Id_r \\ 0 \end{pmatrix} P' = \begin{pmatrix} S'_1 \\ S'_3 \end{pmatrix} P'.$$

Comme A et A' sont de rang r, ces relations impliquent que les deux matrices $r \times r$ à coefficients dans $GF(q)$, S_1 et S'_1 sont inversibles.

Ces relations sont aussi vérifiées dans $GF(q^m)$. Par suite, nous avons

$$S \begin{pmatrix} Id_r \\ 0 \end{pmatrix} P\Theta = S' \begin{pmatrix} Id_r \\ 0 \end{pmatrix} P'\Theta',$$

d'où l'on déduit que $S_1 P\Theta = S'_1 P'\Theta'$. Les deux bases sont donc équivalentes. □

II.3.1.d Analyse asymptotique

Il est possible d'estimer la complexité asymptotique de notre algorithme. D'une part, le nombre de bases strictes croît asymptotiquement comme $O(q^{(m-r)(r-1)})$. D'autre part, il est bien connu que le nombre d'opérations élémentaires d'une élimination gaussienne dans $GF(q)$ est $O((nr)^3)$. Pour la recherche dans un code linéaire $[n,k]$ défini sur $GF(q^m)$ d'un mot de code de rang r, la complexité asymptotique de notre algorithme est donc

$$O\left((nr)^3 q^{(m-r)(r-1)}\right).$$

II.3.2 Décodage général à distance minimale de Gabidulin

Notre problème est désormais celui du décodage à distance minimale de Gabidulin. Nous supposons donné un syndrome non nul $\bar{\sigma} = (\sigma_1, \ldots, \sigma_{n-k})$ défini sur $GF(q^m)$. Dans ce cas, le système (II.3.2) est remplacé par :

$$\begin{aligned} \forall i, 1 \leq i \leq n-k, \quad \sum_{j=1}^{n} \sum_{k=0}^{r-1} h_{i,j} \alpha_{j,k} \theta_k &= \sigma_i, \\ h_{i,j} \alpha_{j,0} + \sum_{j=1}^{n} \sum_{k=1}^{r-1} h_{i,j} \alpha_{j,k} \theta'_k &= \sigma_i \theta_0^{-1}, \end{aligned} \tag{II.3.5}$$

avec $\theta'_k = \theta_k/\theta_0$. Soit $(b_1, \ldots, b_m)$ une base de $GF(q^m)$ dans $GF(q)$. Il existe m éléments $\beta_1, \ldots, \beta_m$ de $GF(q)$ tels que

$$\frac{-1}{\theta_0} = \sum_{\ell=1}^{m} \beta_\ell b_\ell.$$

Alors, nous pouvons écrire

$$h_{i,j} \alpha_{j,0} + \sum_{j=1}^{n} \sum_{k=1}^{r-1} h_{i,j} \alpha_{j,k} \theta'_k + \sum_{\ell=1}^{m} \beta_\ell b_\ell \sigma_i = 0. \tag{II.3.6}$$

Ce système comporte $n-k$ relations sur $GF(q^m)$ qui donnent sur $GF(q)$ un système d'au plus $(n-k)m$ équations indépendantes à $nr+m$ inconnues. Le reste de l'algorithme demeure en tout point identique. L'énumération des bases strictes demeure inchangée. Seul le nombre d'inconnues du système linéaire obtenu à chaque itération est légèrement augmenté. Ceci conduit à une complexité asymptotique en

$$O\left((nr+m)^3 q^{(m-r)(r-1)}\right).$$

II.3.3 Implantation

II.3.3.a Méthode d'énumération

II.3.3.a.a Principe

Nous pouvons représenter de manière unique une base stricte $\Theta = (1, \theta_1, \ldots, \theta_{r-1})$ par une matrice $(r-1) \times (m-1)$ à coefficients dans $GF(q)$ en utilisant les chiffres représentant les θ_i

$$\Theta = \begin{pmatrix} \theta_{1,m} & \cdots & \theta_{1,2} \\ \vdots & & \vdots \\ \theta_{r-1,m} & \cdots & \theta_{r-1,2} \end{pmatrix}.$$

Soit $\succ$ la relation d'ordre lexicographique associée à la représentation polynomiale de $GF(q^m)$ sur $GF(q)$.

Nous cherchons à énumérer un représentant $(\theta_1, \ldots, \theta_{r-1})$ par classe d'équivalence. Sans nuire à la généralité, nous pouvons imposer que

$$\theta_1 \succ \ldots \succ \theta_{r-1}, \tag{II.3.7}$$

et que tous les chiffres les plus significatifs des θ_i soient égaux à un.

Considérons la matrice

$$\Theta_{0,0} = (\, Id_{r-1} \quad 0\,).$$

Cette matrice respecte la condition (II.3.7). En outre, pour toute matrice A_0 de dimensions $(m-r)\times(r-1)$ sur $GF(q)$, les bases

$$\Theta_{0,A} = (\, Id_{r-1} \quad A_0\,)$$

sont toutes de classes d'équivalence distinctes.

Nous appelons *racine* toute matrice R qui respecte les conditions (II.3.7) obtenue par permutation de colonnes de $\Theta_{0,0}$. Pour chaque racine, notre algorithme consiste alors à énumérer les matrices de complétion A_R. Un exemple permet sans doute de mieux comprendre cette énumération.

II.3.3.a.b Exemple

Considérons le paramètres

$$q = 3, m-1 = 3 \text{ et } r-1 = 2.$$

Nous avons tout d'abord

$$R_0 = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \end{pmatrix}.$$

Pour cette racine, nous énumérons les matrices de complétion $\begin{pmatrix} x \\ y \end{pmatrix}$. Ceci fournit $3^2 = 9$ bases strictes :

$$\Theta_{0,0} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

$$\Theta_{0,1} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \Theta_{0,2} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \end{pmatrix}, \Theta_{0,3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \Theta_{0,4} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix},$$

$$\Theta_{0,5} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}, \Theta_{0,6} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \end{pmatrix}, \Theta_{0,7} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \Theta_{0,8} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \end{pmatrix}.$$

La deuxième racine qui vérifie l'ordre lexicographique est

$$R_1 = \begin{pmatrix} 1 & x & 0 \\ 0 & y & 1 \end{pmatrix}.$$

Cette deuxième racine nous donne seulement 3 bases supplémentaires, car l'ordre lexicographique impose que y soit nul :

$$\Theta_{1,0} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \Theta_{1,1} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \Theta_{1,2} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

La dernière racine obtenue

$$\Theta_{2,0} = \begin{pmatrix} x & 1 & 0 \\ y & 0 & 1 \end{pmatrix},$$

ne fournit aucune autre base que $x = y = 0$.

On peut vérifier que le nombre de classes d'équivalence pour ce jeu de paramètres est effectivement

$$\frac{N_3(3,2)}{C_3(2)} = 13.$$

II.3.3.b Programmation

Ici encore, la librairie ZEN a permis une implantation rapide et efficace de ces algorithmes. Nous reviendrons au chapitre III.1 sur les résultats obtenus qui permettent la cryptanalyse de systèmes cryptographiques.

Troisième partie

Cryptographie

Chapitre III.1

Systèmes cryptographiques

La cryptologie est une science qui se divise en deux domaines complémentaires. La cryptographie d'une part cherche à construire des outils permettant d'assurer un certain nombre de fonctionnalités cryptographiques. La cryptanalyse d'autre part s'attaque aux outils précédents pour en tester la fiabilité, mais aussi afin de préciser, pour un système donné, le jeu de paramètres qu'il convient d'utiliser et la sécurité qu'on peut en attendre. Ces deux approches sont indissociables.

Nous commençons par indiquer les fonctionnalités demandées à la cryptographie (III.1.1). Nous donnons ensuite quelques indications sur la sécurité des systèmes cryptographiques, en précisant en particulier les ordres de grandeur temporels nécessaires (III.1.2). La plupart des fonctionnalités évoquées sont d'ores et déjà assurées par nombre de systèmes cryptographiques. Nous expliquons donc en quoi ceux utilisant les codes correcteurs d'erreur peuvent être intéressants (III.1.3). Nous énumérons alors un certain nombre de systèmes utilisant des codes en métrique de Hamming et étudions leur sécurité, en particulier vis-à-vis de nos attaques (III.1.4). La même étude est réalisée pour les systèmes en métrique de Gabidulin (III.1.5).

III.1.1 Fonctionnalités de cryptographie

Voyons maintenant très succintement les possibilités qu'offre la cyptographie[1]. Pour cela on utilise traditionnellement des scénarios à quatre personnages, Alice, Bob, Charlie et Ève [1-AB94]. Alice et Bob sont des utilisateurs normaux des systèmes, Charlie est un utilisateur qui est peut être indélicat, et Ève l'ennemie à abattre (!), c'est-à-dire une personne qui normalement ne doit pas avoir accès au système, ou tout du moins aux informations sensibles de ce système.

III.1.1.a Systèmes de chiffrement

Un système de chiffrement est un système qui permet l'envoi de messages confidentiels sur un réseau peu sûr. Alice veut envoyer un message à Bob sans que Charlie ni *a fortiori* Ève puisse en connaître le contenu.

III.1.1.b Systèmes d'identification

L'identification (ou encore authentification) consiste à prouver son identité ou plus généralement sa qualité à autrui. Un système d'identification permet donc à Alice de prouver à Charlie qui elle est sans que celui-ci puisse ensuite se faire passer pour Alice auprès de Bob. Le principe utilisé suppose qu'Alice dispose d'un secret que Charlie ne connaît pas. Par contre, Bob et Charlie savent qu'Alice possède ce secret et, si le système est sûr, qu'elle est la seule à le connaître. Alice doit donc prouver à Charlie qu'elle connaît ce secret, mais sans pour autant le révéler ce qui permettrait à Charlie de se faire passer pour elle auprès de Bob.

1. Nous nous sommes inspirés ici de l'introduction de la thèse de Vaudenay [8-Vau95]. Le lecteur intéressé par la cryptographie pourra d'ailleurs se reporter à l'ouvrage qu'il a traduit [2-Sti96], qui présente une bonne synthèse de la cryptologie actuelle.

Temps de calcul	Ordre de grandeur
2^0 s	Durée d'une identification (1 s)
2^7 s	Durée d'une communication téléphonique (3 min.)
2^{10} s	Durée d'un transfert de fichiers (20 min.)
2^{20} s	Traversée maritime de l'Atlantique (12 jours)
2^{30} s	1 mois de calcul sur 400 machines
2^{35} s	1 an de calcul sur 1000 machines
2^{40} s	Durée écoulée depuis la disparition de l'homme de Neandertal (35000 ans).
2^{50} s	Début de l'oligocène (35 millions d'années)
2^{60} s	Age estimé de l'Univers (30 milliards d'années)

FIG. 1.1 – *Ordres de grandeur temporels*

III.1.1.c Systèmes de signature

La signature consiste à associer physiquement à un message une identité, à garantir son intégrité et à en empêcher toute répudiation ultérieure. Alice qui reçoit un message signé de Bob doit être sure qu'il n'est pas de Charlie. Si elle en reçoit un de Charlie, et que ce dernier affirme par la suite qu'il n'a pas envoyé de message elle doit pouvoir prouver que seul Charlie a pu signer le message. Charlie ne doit pas non plus pouvoir substituer un message à un autre.

III.1.2 Sécurité cryptographique

LA SÉCURITÉ ACTUELLE DES SYSTÈMES CRYPTOGRAPHIQUES repose d'une part sur des résultats théoriques, mais aussi sur l'étude empirique des systèmes. En effet, les résultats théoriques permettent de sélectionner des familles de problèmes dont on sait qu'ils peuvent être difficile à résoudre. Par la suite, on sélectionne parmi ceux-ci des problèmes pour lesquels, en pratique, on ne sait pas trouver de solution en un temps raisonnable.

Une fois conçu dans le respect des règles théoriques, un système cryptographique doit donc être validé par la recherche des meilleures attaques possibles. Ceci permet de déterminer les jeux de paramètres nécessaires. La figure 1.1 présente les ordres de grandeurs utiles en cryptographie.

On peut distinguer deux types d'attaques d'un système cryptographique. Celles qui s'attaquent uniquement à une instance du problème (type I) et celles qui déterminent une clé secrète (type II). Ainsi, pour un système d'authentification, on pourra se satisfaire du fait que les attaques de type I connues ne puissent permettre de s'identifier frauduleusement dans un temps comparable à une identification honnête. Par contre, une attaque de type II qui prendrait $2^{35}s$ de temps de calcul condammnerait le jeu de paramètres correspondant puisqu'elle fournirait une clé secrète.

Enfin, il faut bien avoir à l'esprit que le rythme actuel de progression des performances techniques double la vitesse des machines tous les 18 mois. Ainsi, si une attaque prend aujourd'hui $2^{30}s$ (34 ans) sur une machine, ceci signifie que l'attaque prendra 6 mois dans 9 ans sur la meilleure machine d'alors.

En tenant compte de l'application du système cryptographique, de sa durée de vie et de l'ordre de grandeur temporel de son utilisation, on peut alors estimer le niveau de sécurité nécessaire pour les deux types d'attaque et déterminer en conséquence les paramètres du système.

III.1.3 Intérêt des cryptosystèmes basés sur la théorie des codes

DEPUIS L'APPARITION DES CONCEPTS de cryptographie à clef publique [4-DH76] et de protocoles à divulgation nulle [4-GMR89], la recherche de nouveaux systèmes de chiffrement et d'identification a été très active. Mais on constate qu'actuellement, la quasi-totalité des systèmes utilisés a une sécurité basée sur des problèmes de théorie des nombres. En schématisant, on peut même affirmer que la cryptographie

pratique actuelle se trouverait dans une situation très inconfortable si des avancées importantes se réalisaient dans le domaine de la factorisation.

Or des résultats récents de complexité semblent indiquer qu'une solution à ce problème mathématique pourrait être obtenue. En effet, en utilisant une modélisation d'ordinateur quantique il a été prouvé l'existence d'un algorithme en temps polynomial de factorisation [4-Sho96]. D'un point de vue technologique, la réalisation de la fonctionnalité quantique nécessaire semble encore lointaine, mais le fait qu'une seule opération soit nécessaire, à savoir une transformée de Fourier quantique, doit inciter à la prudence.

Il pourrait donc s'avérer nécessaire de recourir à des systèmes cryptographiques basés sur des problèmes différents et en particulier des problèmes $\mathcal{NP}$-complets [2-GJ78]. Il semble en effet que l'ordinateur quantique ne permette pas d'amélioration dans ce domaine. En particulier, les systèmes cryptographiques basés sur des problèmes de théorie des codes peuvent présenter une alternative intéressante car ce sont des outils connus, bien maîtrisés d'un point de vue technologique, et ne nécessitant donc que l'emploi d'opérations très simples.

Bien entendu, la sécurité de ces systèmes doit être soigneusement étudiée afin de préciser le paramétrage qu'elle nécessite. Nous présentons ici un certain nombre de ces cryptosystèmes et nous étudions pour chacun d'eux diverses attaques possibles. Nous utilisons en particulier les algorithmes précédents.

III.1.4 Systèmes utilisant la distance de Hamming

III.1.4.a Systèmes de chiffrement

III.1.4.a.a Système de McEliece

Le système de McEliece est le premier système de chiffrement à clé publique non basé sur la théorie des nombres. Bien que datant de 1978, il reste à ce jour le seul système de chiffrement de ce type a être inviolé. À notre connaissance, les autres systèmes de chiffrement non basés sur la théorie des nombres, et en particulier ceux que nous décrivons par la suite, ont tous été cassés soit irrémédiablement, soit dans les paramétrages initialement proposés. Même le système de Chor-Rivest

Principe

Le système de McEliece utilise un code correcteur d'erreur binaire de matrice génératrice G. Ce code doit avoir un taux de correction élevé et pouvoir être décodé à l'aide d'un algorithme rapide. Nous avons vu que ces algorithmes de décodage faisaient usage de la structure très particulière de certains codes. Le principe du système de McEliece est donc de camoufler cette structure pour obtenir un code équivalent au code initial mais pour lequel l'algorithme de décodage n'est pas applicable. Le système de McEliece est décrit figure 1.2 dans sa version initiale.

Codes utilisés

Le système de McEliece exige un certain nombre de contraintes bien connues pour assurer sa sécurité :

1. Le code utilisé doit appartenir à une famille de codes suffisamment grande pour éviter qu'une énumération de tous les codes correspondant aux paramètres publics ne soit possible. Ceci exclut en particulier l'utilisation des codes BCH.

2. Il ne doit pas être possible de retrouver une structure décodable à partir de la matrice camouflée. Nous reviendrons sur ce point au paragraphe III.1.4.a.d, mais sachons d'ores et déjà que des attaques rendent impossible l'utilisation des codes de Reed-Solomon généralisés, des codes concaténés, ainsi que des codes alternants à l'exception notable des codes binaires de Goppa.

Apparemment, seuls les codes binaires de Goppa semblent donc utilisables.

Notons enfin que des paramètres légèrement différents ont été proposé par Adams-Meijer [5-AM87] pour renforcer la sécurité du système de McEliece :

$$\begin{aligned} n &= 1024, \\ k &= 654, \\ t &= 37. \end{aligned}$$

Tous les objets utilisés sont définis sur $GF(2)$.

Clé secrète

- Une matrice génératrice G d'un code binaire de dimensions $[n, k]$ corrigeant t erreurs et l'algorithme de décodage qui s'y applique.
- Une matrice inversible quelconque S de dimension $k \times k$.
- Une matrice de permutation quelconque P de taille n.

Clé publique :

- La matrice génératrice $G' = SGP$ d'un code équivalent à G.
- Le taux de correction t du code G.

Chiffrement :
Le chiffré d'un message x de k bits est le message c de n bits défini par

$$c = mG' + e\,,$$

où e est une erreur aléatoire de poids de Hamming égal à t.
Déchiffrement :

1. Connaissant P, on calcule

$$\begin{aligned} cP^{-1} &= mG'P^{-1} + eP^{-1} \\ &= mSG + eP^{-1}. \end{aligned}$$

2. L'erreur eP^{-1} est de poids égal à t ; elle est donc corrigeable par le code G. On obtient ainsi le mot de code mSG, donc mS, puis m puisque S est connue et inversible.

Paramètres :
Dans le système initial de McEliece est utilisé un code binaire de Goppa de paramètres :

$$\begin{aligned} n &= 1024, \\ k &= 524, \\ t &= 50. \end{aligned}$$

FIG. 1.2 – *Système de McEliece*

III.1.4.a.b Système de Niederreiter

Description

Le système de Niederreiter est très voisin du système de McEliece. Il n'est donc pas surprenant que l'on puisse montrer l'équivalence des deux systèmes [4-LDW94]. Il nécessite bien entendu les mêmes critères que le système de McEliece. Nous le décrivons figure 1.3 sans préciser de paramètres. En effet, les codes proposés initialement par Niederreiter se sont tous révélés non sûrs [4-BO88]. On peut considérer que des paramètres identiques à ceux du système de McEliece sont nécessaires.

Le système de Niederreiter nécessite en outre un algorithme $\mathcal{A}$ permettant de coder r bits d'information sous forme d'un vecteur d'erreur e de n bits de poids de Hamming égal à la valeur t du taux de correction du code. De tels algorithmes existent [5-Sen95, 4-CR88]. Du point de vue de la sécurité, nous verrons qu'utiliser des vecteurs d'erreur de poids trop faible est dangereux.

Comparaison avec le système de McEliece

Il faut néanmoins noter un avantage du système de Niederreiter sur le système de McEliece. En effet le chiffrement de l'information est moins gourmand au niveau de la redondance. Prenons l'exemple du paramétrage classique du système de McEliece :

$$\begin{aligned} n &= 1024, \\ k &= 524, \\ t &= 50. \end{aligned}$$

Avec le système de McEliece, un message de 524 bits à chiffrer nécessite 1024 bits de transmission. Ceci implique un taux de transmission de $\frac{524}{1024} \simeq 0.51$. Avec le système de Niederreiter, le chiffré est un message de $n-k = 500$ bits. Mais combien de bits d'information sont utilisables ? En première approximation, il y a $\binom{1024}{50} \simeq 2^{284}$ vecteurs possibles. On obtient donc un taux de transmission de $\frac{284}{500} \simeq 0.57$.

On peut envisager d'améliorer ce taux de transmission en combinant les deux idées. En considérant à la base un système de McEliece on peut considérer que l'erreur e introduite permet de coder 284 bits d'information supplémentaires. Le taux de transmission qui en résulte est alors $\frac{808}{1024} \simeq 0.79$.

III.1.4.a.c Attaque statistique

Le système de Niederreiter présente une plus grande vulnérabilité à des attaques de type statistique. En fait c'est l'algorithme $\mathcal{A}$ qui peut présenter des faiblesses. En effet, si les 284 bits d'information de l'exemple précédent, utilisés dans l'algorithme de codage initial $\mathcal{A}$, sont significativement biaisés, alors l'espace des vecteurs d'erreur utilisés risque de se réduire considérablement. La sécurité du système n'est alors plus assurée.

Cette remarque s'applique de la même manière au système de McEliece modifié. Par contre, le système de McEliece initial pourrait sembler présenter moins de vulnérabilité puisque l'erreur choisie étant aléatoire un même message peut être chiffré de 2^{284} manières différentes.

Ne nous laissons quand même pas impressionner par ce chiffre, car deux chiffrés c_1 et c_2 d'un même message ne différent au plus que par $50+50 = 100$ bits sur 1024 soit moins que la distance minimale du code ce qui est normal. Ils sont donc parfaitement détectables. Donc si on chiffre deux fois le même message, ces deux chiffrés sont repérables. En outre, les bits qui diffèrent entre les deux chiffrés correspondent forcément à l'une des erreurs e_1 ou e_2. Il faut donc chercher parmi les bits inchangés une fenêtre d'information. Évaluons sur l'exemple précédent le paramétrage de ce nouveau problème. En moyenne $e_1 \oplus e_2$ a pour poids $50 + 50 - 2 \times \frac{50}{1024} \times 50 \simeq 95$. On peut donc éliminer 94–96 bits et il ne nous reste plus qu'à décoder une erreur d'au plus 3 bits dans un code aléatoire $[930, 524]$! Paradoxalement, c'est donc justement le fait que le même mesage soit chiffré de manière différente qui fournit un moyen de cryptanalyse.

III.1.4.a.d Attaques par recherche de la trappe

Une méthode d'attaque possible consiste à tenter de déterminer à partir de la clé publique de chiffrement la clé secrète sous-jacente. Dans le cas des système de McEliece et de Niederreiter il s'agit donc, à partir de la matrice génératrice camouflée

$$G' = SGP \tag{III.1.1}$$

Tous les objets utilisés sont définis sur un corps fini $GF(q)$ quelconque.

Clé secrète :

- Une matrice de contrôle H d'un code $[n,k]$ corrigeant t erreurs et l'algorithme de décodage qui s'y applique.
- Une matrice inversible quelconque M de dimension $(n-k)\times(n-k)$.
- Une matrice de permutation quelconque P de taille n.

Clé publique :

- La matrice de contrôle $H' = MHP$ d'un code équivalent à H.
- Un algorithme $\mathcal{A}$ permettant de coder r bits d'information sous forme d'un vecteur e de n bits de poids de Hamming égal à t. De tels algorithmes existent [5-Sen95, 4-CR88].

Chiffrement :

1. Un message m de r bits est tout d'abord codé par l'algorithme $\mathcal{A}$ sous forme d'un message $e = \mathcal{A}(m)$ de n bits.
2. Le chiffré du message est alors le syndrome s de $n-k$ par la matrice H'

$$s = H'e.$$

Déchiffrement :

1. Connaissant M, on calcule

$$\begin{aligned} M^{-1}s &= M^{-1}H'e \\ &= HPe. \end{aligned}$$

2. L'erreur Pe est de poids égal à t ; elle est donc corrigeable par le code H. L'algorithme de décodage fournit donc Pe, puis e puisque P est connue et inversible.
3. En inversant l'algorithme de codage initial on retrouve le message

$$m = \mathcal{A}^{-1}(e).$$

FIG. 1.3 – *Système de Niederreiter*

de retrouver la matrice G initiale. En fait, le problème est un peu moins difficile que cela car c'est moins la matrice G elle-même que la structure du code qui est à découvrir. En effet nous avons vu que les algorithmes de décodage des codes correcteurs utilisés dans ces systèmes nécessitent la connaissance d'une matrice de contrôle structurée du code. Dans le cas des codes alternants par exemple, toute matrice de contrôle ayant la structure de celle d'un code de Reed-Solomon généralisé permet le décodage. Le but des attaques qui suivent est donc de retrouver les structures de codes camouflés.

Algorithme de Sidel'nikov-Shestakov

L'algorithme de Sidel'nikov-Shestakov interdit l'emploi de code de Reed-Solomon généralisé sur $GF(q)$ dans les systèmes précédents. Il prouve en effet que le camouflage de l'équation (III.1.1) n'en est pas vraiment un et que retrouver la structure du code est possible à l'aide d'un algorithme en temps polynomial.

Gabidulin a proposé dans [5-GK94] une modification du système de Niederreiter qui permet d'éviter cette attaque. Elle consiste à ajouter à la matrice camouflée une matrice X de rang 1 sur $GF(q)$. Au niveau du déchiffrement, si x est un vecteur ligne non nul de X, il convient d'énumérer les $q-1$ vecteurs non nuls λx pour tout $\lambda \in GF(q)^\star$ et de décoder pour chacun d'eux le vecteur $c \oplus \lambda x$. Cette modification introduit donc un surcoût au niveau du déchiffrement.

Implantation

L'algorithme de Sidel'nikov-Shestakov a été décrit et implanté dans [8-Cha93] à partir d'une traduction de [4-SS92]. Cette implantation a été en partie à l'origine de l'élaboration de la librairie ZEN. Elle a permis de démontrer la faisabilité de l'attaque puisque pour des dimensions de l'ordre de celles employées dans le système de McEliece, l'algorithme retrouve la structure du code en quelques dizaines de secondes.

Algorithme de Sendrier

L'algorithme de Sendrier [7-Sen95] interdit l'emploi des codes concaténés dans les systèmes de chiffrement evoqués plus haut. La concaténation des codes est une opération qui permet l'obtention de bons codes correcteurs pour un coût d'implantation peu élevé. Elle consiste a considérer un code de faibles dimensions sur un alphabet $\mathbb{F}$ à q éléments puis à utiliser un second code de faibles dimensions pour coder les éléments de $\mathbb{F}$. Le code qui en résulte a généralement de très bonnes capacités de correction.

Malheureusement, Sendrier a prouvé que l'utilisation de ce type de construction ne pouvait absolument pas être camouflé par la méthode classique du système de McEliece (voir l'équation (III.1.1).

Algorithme d'Heiman

Gibson cite dans [5-Gib91] les travaux de Heiman [8-Hei87] qu'il ne nous a pour l'instant pas été possible d'obtenir. Selon Gibson, Heiman a mis au point des algorithmes de décodage rapides des codes alternants utilisables à partir de n'importe quelle matrice génératrice. Cependant, ces algorithmes ne permettraient le décodage des codes binaires de Goppa que jusqu'à la moitié de leur distance minimale réelle et le système de McEliece ne serait donc pas mis en danger. Ceci proviendrait du fait que les codes binaires de Goppa ont une distance minimale réelle double du degré du polynôme de Goppa utilisé (voir note II.1.2.g.b).

III.1.4.b Systèmes d'identification

Dans les systèmes d'identification est souvent utilisée une fonction de hachage. Une fonction de hachage est une fonction dont l'espace image est de taille beaucoup plus petite que son espace de définition. Par suite, il existe forcément des couples d'éléments (x, y) de l'espace de définition qui ont même image. Ces couples sont appelés *collisions*. Au sens cryptographique, une fonction de hachage doit être telle que l'obtention d'une collision soit quasi-impossible. Nous notons dans la suite $\langle x \rangle$ l'image de x par une fonction de hachage cryptographique.

Note 147. Les fonctions de hachage cryptographiques couramment utilisées sont définies de manière binaire. Elles envoient $GF(2)^n$ dans $GF(2)^k$ avec $n >> k$. Le paradoxe des anniversaires indique qu'une collision $\langle x \rangle = \langle y \rangle$ est obtenue avec une probabilité élevée dans un sous ensemble de $GF(2)^n$ de taille $O(\sqrt{2^k})$. C'est pourquoi les deux fonctions de hachage cryptographiques les plus courantes, MD5 [8-MD5] et SHA [8-SHA] retournent des valeurs de taille $k \geq 128$, respectivement de 160 et 128 bits.

Tous les objets utilisés sont définis sur $GF(2)$.

Clé secrète :
Chaque identifiant possède un vecteur s de n bits de poids w.
Clé publique :

- Une matrice de contrôle H d'un code binaire aléatoire $[n, k]$.
- La valeur w.
- Chaque identifiant publie le syndrome $i = Hs$.

Protocole :

Prouveur		Vérifieur
Choisit un vecteur aléatoire y de n bits et une matrice de permutation P de dimension n, et calcule $c_1 = \langle P \vert\vert Hy \rangle$ $c_2 = \langle Py \rangle$ $c_3 = \langle P(y \oplus s) \rangle$	$c_1, c_2, c_3 \longrightarrow$	
	$\longleftarrow b$	Choisit aléatoirement $b \in \{0, 1, 2\}$
	(y, P) ou $(y \oplus s, P)$ ou $(Py, Ps) \longrightarrow$	
Si $b = 0$, envoie y et P, ou si $b = 1$, envoie $y + s$ et P, ou si $b = 2$, envoie Py et Ps		Si $b = 0$, vérifie c_1 et c_2, si $b = 1$, vérifie c_1 en utilisant i et c_3, si $b = 2$ vérifie c_2, c_3 et le poids de Ps.

Paramètres : Les paramètres les plus faibles proposés sont

$$\begin{aligned} n &= 512, \\ k &= 256, \\ w &= 56. \end{aligned}$$

FIG. 1.4 – *Système de Stern*

III.1.4.b.a Système de Stern

Le système de Stern est basé sur la difficulté du problème SD. Il utilise une fonction de hachage cryptographique publique. Nous décrivons figure 1.4 le protocole d'identification, où $a||b$ désigne la concaténation des deux vecteurs de bits a et b.

Le point important du paramétrage de cet algorithme est que le poids w de la clé secrète de chaque identifiant, doit être légèrement inférieur à la borne de Gilbert-Varshamov. La borne de Gilbert-Varshamov (voir [2-MWS83]) correspond au comportement asymptotique de la distribution des poids d'un code aléatoire. Ainsi, si l'on considère un code aléatoire binaire de dimensions $[n,k]$, on peut estimer la probabilité qu'il contienne un mot de poids w. Puisqu'il y a $\binom{n}{w}$ mots de poids w, la proportion de ces mots dans l'espace total $GF(2)^n$ est

$$\frac{\binom{n}{w}}{2^n}.$$

Le code contenant 2^k mots, la probabilité qu'il contienne un mot de poids w est

$$GV_{n,k}(w) = \frac{\binom{n}{w}}{2^{n-k}}.$$

Schématiquement, on peut donc trouver des codes $[n,k]$ dont la distance minimale est

$$d = \min\{w \,/\; GV_{n,k}(w) \geq 1\}. \tag{III.1.2}$$

Par suite, l'équation (III.1.2) définit une borne inférieure d'existence de codes linéaires, qui est très utile pour l'étude asymptotique de familles de codes. Il semble d'autre part naturel à l'issue de cette description, que l'espérance de la distance minimale d'un code aléatoire $[n,k]$, notée par abus d, vérifie l'équation (III.1.2). Ce résultat est dû à Pierce [4-Pie67].

Nous décrirons plus loin (voir paragraphe III.1.4.c) l'utilisation des algorithmes présentés au chapitre II.1 comme attaque du système de Stern. D'ores et déjà nous pouvons noter que l'efficacité de ces algorithmes décroît avec le poids de l'erreur recherchée. En contrepartie, pour un code aléatoire, le nombre de mots de code augmente exponentiellement au delà de la borne de Gilbert-Varshamov. Ces deux phénomènes se compensent, si bien que pour l'attaque envisagée, on ne peut considérer comme difficile, c'est-à-dire comme cryptographiquement sûrs, que les instances du problème SD telles que le poids recherché soit proche de la borne de Gilbert-Varshamov [5-FS96]. En outre, pour un système d'identification, la possibilité d'avoir des identités secrètes avec même syndrome public nul pourraient s'avérer génante. Il convient donc de choisir pour le système de Stern un poids public très légèrement inférieur à la borne de Gilbert-Varshamov. Dans le cas d'un code $[512, 256]$, la distance minimale donnée par la borne de Gilbert-Varshamov est 58, et 56 est donc un paramétrage naturel.

III.1.4.b.b Système de Véron

Le système de Véron est au système de Stern ce que le système de McEliece est au système de Niederreiter. Il est basé sur le même problème mais présenté de façon duale. En plus des paramétrages identiques à ceux du système de Stern, Véron propose deux paramétrages supplémentaires (voir figure 1.6) qui permettent d'améliorer les performances pratiques des systèmes.

III.1.4.b.c Attaque possible

Les codes utilisés dans les système d'identification sont totalement aléatoires. Retrouver le secret à partir de l'information publique consiste donc à résoudre un problème de décodage général des codes correcteurs. Il s'agit, rappelons le encore, d'un problème $\mathcal{NP}$-complet.

III.1.4.c Attaque par décodage général

Les différents systèmes précédents peuvent tous être attaqués par les algorithmes présentés au chapitre II.1. Le dimensionnement de ces systèmes devra donc être choisi en conséquence. Ainsi, pour le système de McEliece, l'utilisation de codes de Goppa de longueur 512 est exclue. Par contre, les codes de longueur

Tous les objets utilisés sont définis sur $GF(2)$.

Clé secrète :
Chaque identifiant possède un vecteur e de n bits de poids w et un vecteur m de k bits.
Clé publique :

- Une matrice génératrice G d'un code binaire aléatoire $[n, k]$.
- La valeur w.
- Chaque identifiant publie le mot de code erroné $x = mG \oplus e$.

Protocole :

Prouveur	Vérifieur
Choisit un vecteur aléatoire u de k bits et une matrice de permutation P de dimension n, et calcule	
$c_1 = \langle P \rangle$ $c_2 = \langle (u \oplus m)GP \rangle$ $c_3 = \langle (uG \oplus x)P \rangle$	$\xrightarrow{c_1, c_2, c_3}$
	Choisit aléatoirement $b \in \{0, 1, 2\}$
	$\xleftarrow{b}$
$(u \oplus m, P)$ ou $((u \oplus m)GP, eP)$ ou (u, P)	$\xrightarrow{}$
Si $b = 0$, envoie $u \oplus m$ et P, ou si $b = 1$, envoie $(u \oplus m)GP$ et eP, ou si $b = 2$, envoie u et P	Si $b = 0$, vérifie c_1 et c_2, si $b = 1$, vérifie c_2 et c_3 et le poids de eP, si $b = 2$ vérifie c_1, c_3.

Paramètres : Les paramètres les plus faibles proposés sont

$$\begin{aligned} n &= 512, \\ k &= 120, \\ w &= 114. \end{aligned}$$

FIG. 1.5 – *Système de Véron*

Système	Type	Code	Poids	Attaque utilisée	Temps estimé
McEliece	Chiffrement	$[1024, 524]$	50	B'(2,8,2)	2^{40}
McEliece	Chiffrement	$[1024, 654]$	37	B'(2,8,2)	2^{42}
Stern	Identification	$[512, 256]$	56	B'(2,7,2)	2^{45} s
Véron	Identification	$[510, 255]$	56	B'(2,8,2)	2^{44} s
Véron	Identification	$[512, 120]$	114	B'(2,7,2)	2^{35} s

FIG. 1.6 – *Meilleures attaques par décodage contre différents systèmes utilisant SD*

Tous les objets utilisés sont définis sur $GF(2^m)$.

Clé secrète

- Une matrice génératrice G d'un code MRD de dimensions $[n,k]$ corrigeant $t = \frac{n-k}{2}$ erreurs.
- Une matrice inversible quelconque S de dimension $k \times k$.
- Une matrice génératrice D d'un code de distorsion $\delta < t$, de taux de distorsion τ, de dimensions $[n,k]$.

Clé publique

- La matrice génératrice $K = SG + D$.
- Le taux de correction $t - \delta$ du code K.

Chiffrement Le chiffré d'un message x de k éléments de $GF(2^m)$ est le message c de n éléments de $GF(2^m)$ défini par

$$c = xK + e,$$

où e est une erreur aléatoire de rang inférieur à $t - \delta$.
Déchiffrement L'erreur totale $xD + e$ étant de rang inférieur à t, l'algorithme de décodage permet de retrouver le message x initial.
Paramètres
Dans le système initial de Gabidulin les paramètres proposés sont :

$$\begin{aligned} m &= 20, \\ n &= 20, \\ k &= 12, \\ \delta &= 3, \\ \tau &= \frac{1}{\delta}. \end{aligned}$$

FIG. 1.7 – *Système de Gabidulin*

1024 primitivement proposés par McEliece présentent encore une bonne sécurité. Il faut cependant noter que le système s'affaiblit très rapidement dès qu'une certaine quantité d'information est connue sur le problème. Ainsi si on connaît X bits du message ou Y bits d'erreur, les algorithmes proposés permettent de compléter le décryptage.

De même, le système de Stern ne peut être utilisé avec des codes de longueur 256. Par contre, les codes de longueur 512 sont adaptés. La figure 1.6 résume les différents résultats. Les temps de calcul estimés seraient ceux obtenus sur une sparc 10 à 50 MHz, c'est-à-dire une station de travail désormais assez répandue et de vitesse moyenne. La dernière ligne de cette figure prouve que la version "optimale" du système de Véron est légèrement sous-dimensionnée si l'on considère la discussion du paragraphe III.1.2.

D'autre part, Véron propose dans [8-Vér95] le problème SD de paramètres $n = 96$, $k = 48$, $w = 35$, dans $GF(256)$ comme étant un problème de sécurité comparable à celle obtenue par les différents autres systèmes. Cette évaluation est basée sur les estimations théoriques de [7-CC95a]. L'implantation réalisée montre que ces estimations théoriques sont un peu pessimistes. En effet, le temps de calcul estimé pour ce problème est inférieur (2^{39} s pour l'algorithme B(1,5,5)). En outre, si on se ramène au problème de la recherche d'un mot de poids faible, on sait que tous les multiples du mot initial vont se retrouver dans le code étudié. Ceci nous donne donc 255 mots de poids 35 dans un code $[96, 49]$, d'où un temps de calcul de l'ordre de 2^{31} secondes.

III.1.5 Systèmes utilisant la distance de Gabidulin

III.1.5.a Systèmes de chiffrement

III.1.5.a.a Système de Gabidulin

Le système de Gabidulin est une modification du système de McEliece dans le but d'utiliser des codes MRD. Il a été pour la première fois décrit dans [5-GPT91]. Le principe du système reste le même à savoir camoufler la structure d'un code correcteur d'erreur afin de rendre impossible son décodage rapide. La différence réside dans la méthode de camouflage qui ne peut être la même que dans le cas de codes utilisant la distance de Hamming. Les codes utilisés sont les codes MRD décrits par Gabidulin dans [4-Gab85].

Définition 148. Nous appelons *code de distorsion* un code linéaire dont tous les mots ont un rang inférieur ou égal à δ. Cette quantité est appelée *distorsion* du code. Un tel code peut être représenté par une matrice génératrice de la forme

$$D = AB,$$

où A est une matrice à éléments dans $GF(2^m)$ de dimensions $k \times \delta$, de rang ∂ dans $GF(2^m)$, avec $1 \leq \partial \leq \delta$, et B est une matrice dans $GF(2)$ de dimensions $\delta \times n$ et de rang δ. Nous appelons *taux de distorsion* du code la quantité $\tau = \frac{\partial}{\delta}$.

III.1.5.a.b Attaque par recherche de la trappe

Algorithme de Gibson

Gibson a proposé dans [4-Gib95] une cryptanalyse du système initial de Gabidulin permettant de retrouver une structure exploitable du code correcteur. Ce dernier a donc proposé [5-Gab94] une modification de son système qui a malheureusement permis à Gibson d'obtenir une cryptanalyse encore plus efficace [5-Gib96].

Nouveau paramétrage proposé

Dans son dernier article [5-Gib96], Gibson propose un nouveau paramétrage pour le système de Gabidulin :

$$\begin{aligned} m &= 48, \\ n &= 48, \\ k &= 24, \\ \delta &= 7, \\ \tau &= \frac{2}{\delta}. \end{aligned}$$

III.1.5.b Systèmes d'identification

III.1.5.b.a Système de Chen

Le système de Chen est basé sur le problème RDSD. Sa première description [4-Che94] apparaît comme une modification du système de Girault basé lui sur le problème SD. Une nouvelle présentation (voir figure 1.8) en a été faite dans [5-Che95] avec un paramétrage différent.

Pour les mêmes raisons que dans le cas du système de Stern, il est nécessaire d'imposer que le paramètre public r soit inférieur à la distance minimale du code considéré. Nous pouvons donc d'ores et déjà noter que le paramétrage de [4-Che94], à savoir

$$\begin{aligned} m &= 8, \\ n &= 32, \\ k &= 16, \\ r &\geq 4, \end{aligned}$$

Tous les objets utilisés sont définis sur une extension $GF(2^m)$.

Clé secrète Chaque identifiant possède un vecteur $\bar{s}$ de n éléments de rang r.
Clé publique

- Une matrice de contrôle H d'un code linéaire aléatoire $[n, k]$.
- La valeur r.
- La représentation du corps $GF(2^m)$.
- Chaque identifiant publie le syndrome $\bar{\sigma} = H\bar{s}$.

Protocole

Prouveur		Vérifieur
Choisit un vecteur aléatoire $\bar{x}$ à n éléments dans $GF(q^m)$, une matrice de permutation P de taille n, et calcule		
$\bar{c} = HP\bar{x}$, $\bar{c}' = H\bar{x}$	$\bar{c}$, $\bar{c}'$ →	
	← λ	Choisit aléatoirement $\lambda \in GF(q^m)$
Calcule le vecteur à $(n - k)$ éléments $\bar{w} = \bar{x} + \lambda P^{-1}\bar{s}$	$\bar{w}$ →	
	← b	Choisit $b \in \{0, 1\}$
Si $b = 0$ dévoile P, sinon envoit $\bar{x}$.	P ou $\bar{x}$ →	Si $b = 0$, vérifie si $HP\bar{w} = \bar{c} + \lambda\bar{\sigma}$ et $H\bar{x} = \bar{c}'$.
Sinon vérifie le rang : $r(\bar{w} - \bar{x}, q) = \begin{cases} r, & \text{si } \lambda \neq 0; \\ 0, & \text{si } \lambda = 0. \end{cases}$		

Paramètres Les paramètres proposés dans [5-Che95] sont

$$\begin{aligned} m &= 16, \\ n &= 32, \\ k &= 16, \\ r &= 4. \end{aligned}$$

FIG. 1.8 – *Système de Chen*

Système	Type	Corps	Code	Rang	Temps estimé
Gabidulin	Chiffrement	$GF(2^{20})$	$[20, 12]$	3	2^{29} s
Gabidulin	Chiffrement	$GF(2^{48})$	$[48, 24]$	5	2^{175} s
Chen	Identification	$GF(2^{16})$	$[32, 16]$	4	2^{29} s

FIG. 1.9 – *Meilleures attaques par décodage contre différents systèmes utilisant RDSD*

est incohérent. En effet, considérons le code obtenu en adjoignant à la matrice de contrôle publique H un syndrome d'identité public $\bar{\sigma}$. Nous obtenons un code $[n+1, k+1] = [33, 17]$ qui possède un mot de rang inférieur ou égal à $r + 1 = 5$. La borne (II.3.3) démontrée au paragraphe II.3.1.b.a, impose

$$d \leq \left\lceil \frac{((n+1) - (k+1))m + 1}{(n+1)} \right\rceil \leq 4,$$

ce qui signifie que trouver des mots de rang 4, et *a fortiori* 5, dans ce code s'effectue par le biais d'une élimination gaussienne (voir paragraphe II.3.1.b).

III.1.5.c Attaque par décodage général

En utilisant l'algorithme décrit section II.3.2, il est possible de réaliser l'attaque des systèmes précédents. Dans le cas du système de Gabidulin, il suffit en effet de remarquer que la matrice génératrice publique définit un code correcteur qui corrige $\frac{n-k}{2} - \delta$ erreurs. L'algorithme peut donc s'appliquer avec ces paramètres (voir figure 1.9). Le système initial de Gabidulin apparaît donc de moindre sécurité que le système de McEliece, alors que le but poursuivi par Gabidulin était justement inverse. Cependant, le nouveau paramétrage proposé par Gibson présente lui une bonne sécurité contre notre attaque, tout en conservant un dimensionnement plus avantageux que le système de McEliece initial.

Il est à noter que dans le cas du système de Gabidulin, une attaque par recherche de la trappe est possible, les attaques de Gibson en sont la preuve. Toutefois, si un camouflage plus efficace de la structure des codes de Gabidulin rendait inopérante les attaques de ce type, alors les attaques sans structure connue resteraient les seules possibles. Les performances actuelles de ces attaques font que le dimensionnement des systèmes cryptographiques correspondants pourraient être très avantageux.

Le système de Chen, quant à lui, s'avère non sûr d'un point de vue cryptographique. Un nouveau paramétrage serait nécessaire, ainsi qu'une étude des caractéristiques d'un code aléatoire vis-à-vis de la distance de Gabidulin.

On peut remarquer dans [5-GPT91] qu'une première estimation d'une attaque par décodage général est donnée, mais le nombre d'itérations évalué est beaucoup plus grand que notre résultat :

$$D_q(n, r)(q^m - 1)(q^m - q) \ldots (q^m - q^{r-1}) \gg D_q(m, r).$$

Chapitre III.2

Conclusion

Nous avons présenté plusieurs algorithmes de décodage général des codes correcteurs d'erreur utilisant, tant la distance de Hamming que la distance de Gabidulin. Dans ce dernier cas, il s'agit, à notre connaissance, du premier algorithme proposé. La description de cet algorithme nous a permis de déduire une borne sur la distance minimale de Gabidulin d des codes linéaires $[n,k]$ sur $GF(q^m)$

$$d \leq \left\lceil \frac{(n-k)m+1}{n} \right\rceil ,$$

ce qui implique en particulier la non-existence de codes MRD de longueur $n > m$.

L'implantation de ces algorithmes a conduit à l'élaboration d'une librairie de programmation en C. En collaboration avec Reynald Lercier, cette librairie s'est développée jusqu'à l'obtention de Zen, un nouvel outil permettant la manipulation aisée et efficace d'objets (polynômes, matrices, séries, courbes elliptiques) définis sur des corps finis, et même plus généralement sur des anneaux finis.

Dans le cas de la distance de Hamming, notre travail en collaboration avec Anne Canteaut a permis de prouver la distance minimale réelle de 6 codes BCH de longueur 511 pour lesquels cette information n'était pas connue. Il s'agit des codes BCH de distance construite 29, 37, 41, 43, 51 et 87 qui atteignent tous leur distance construite.

Ces différent algorithmes ont en outre des applications cryptographiques. Ainsi l'algorithme B' constitue à ce jour la meilleure attaque connue de plusieurs cryptosystèmes (le temps estimé de calcul est indiqué entre parenthèses) :

1. Le système de McEliece dans sa version initiale (35,000 ans).
2. Le système de McEliece dans la paramétrisation d'Adams-Meijer [5-AM87] (140,000 ans).
3. Le système de Stern(1 million d'années).
4. Le système de Véron optimal (1000 ans).

Ce dernier système apparaît donc légèrement sous-dimensionné car tous les algorithmes proposés sont distribuable sur un réseau de stations. De même, dans le cas de la distance de Gabidulin, l'algorithme proposé permet une cryptanalyse réaliste sur un réseau de stations du système de Chen (20 ans).

Ces différents résultats permettent de valider les concepts retenus dans l'élaboration de la librairie Zen et d'espérer qu'elle permettra à l'avenir la programmation efficace d'autres applications.

Quatrième partie

Annexes

Annexe A

Exemple de programmes

Nous présentons ici un petit exemple de calcul effectuable en utilisant Zen et que nous réalisons d'abord en Maple [3-CGG+90] et Axiom [1-NAG].

A.1 Maple

Le logiciel maple [3-CGG+90] est devenu incontournable car ses possibilités recouvrent la quasi-totalité des domaines mathématiques. Cependant, le petit exemple qui suit montre que dans le cas des extensions polynomiales sur des corps finis, ce logiciel est moins performant.

A.1.a Anneaux modulaires

Dans les corps modulaires tout se passe bien :

```
             |\^/|     Maple V Release 3 (Ecole Normale Superieure)
           ._|\|   |/|_. Copyright (c) 1981-1994 by Waterloo Maple Software and
            \  MAPLE  /  the University of Waterloo. All rights reserved. Maple
            <____ ____>   and Maple V are registered trademarks of Waterloo Maple
                 |        Software. Type ? for help.
> q:=234776683;
                                        q := 234776683

> A:=234675;
                                         A := 234675

> 1/A mod q;
                                           11532995
```

Mais on commence déjà à avoir des problèmes si l'anneau n'est pas intègre :

```
> r:=8745287453;
                                       r := 8745287453

> isprime(r);
                                            false

> 1/A mod r;
Error, the modular inverse does not exist
> gcd(r,A);
                                              7

> A:=A/7;
                                          A := 33525

> 1/A mod r;
                                          5160567388
```

Ceci ne pose pas de problème mais il est dommage que Maple se contente d'un message d'erreur.

Considérons maintenant le polynôme monique

$$P(X) = X^3 + 3234234X^2 + 234234X + 124123. \tag{A.1}$$

Ce polynôme est irréductible dans $\mathbb{Z}/_{q\mathbb{Z}}$.

```
> e0:=124123;
                                    e0 := 124123
> e1:=234234;
                                    e1 := 234234
> e2:=3234234;
                                   e2 := 3234234
> P:=X^3+e2*X^2+e1*X+e0;
                              3             2
                     P := X  + 3234234 X  + 234234 X + 124123
> Irreduc(P) mod q;
                                         true
```

Mais Maple refuse de répondre si l'anneau modulaire n'est pas intègre :

```
> Irreduc(P) mod r;
Error, (in mod/Irreduc) the modulus must be a prime integer
```

Là encore, on peut regretter que Maple ne délivre qu'un message d'erreur, car dans la pratique cette information est généralement connue.

A.1.b Extension simple

Disposant d'un polynôme irréductible, on peut passer à une extension définie par ce polynôme. Cherchons par exemple à inverser l'élément

$$234234X + 3234234$$

dans l'extension $\mathbb{Z}/_{q\mathbb{Z}}[X]/_{P(X)}$.

```
> alias(x=RootOf(P));
                                         I, x
> E0:=e1*x+e2;
                               E0 := 234234 x + 3234234
> Normal(1/E0) mod q;
                                    2
                          226257462 x  + 25554697 x + 70525059
```

Pour cette inversion, on utilise une nouvelle fonction. Voyons maintenant si le polynôme

$$Q(X) = X^2 + (124123x^2 + 234234)X + (234234x + 3234234)$$

est irréductible dans cette extension

```
> E1:=e0*x^2 + e1;
                                          2
                               E1 := 124123 x  + 234234
>  Q:=X^2 + E1*X + E0;
                        2            2
                 Q := X  + (124123 x  + 234234) X + 234234 x + 3234234
> Irreduc(Q) mod q;
bytes used=1000024, alloc=720764, time=1.32
bytes used=2000152, alloc=1113908, time=2.78
                                         true
```

L'apparition de ce type de lignes dans l'affichage de Maple indique que le calcul effectué commence à devenir pesant. Pourtant, les paramètres utilisés sont de taille ridicule.

A.1.c Double extension

Tentons de renouveler l'extension comme précédemment.

```
> alias(y=RootOf(Q));
                                        I, x, y
> EE0:=y*E1 + E0;
                                    2
                EE0 := y (124123 x  + 234234) + 234234 x + 3234234
> Normal(1/EE0) mod q;
Error, (in mod/GetAlgExt)
only the single algebraic extension case is implemented
```

Puisque l'approche directe n'est pas possible, il nous faut donc calculer le pgcd des deux polynômes "à la main". Attention à ne pas confondre les deux fonctions `Gcd` et `Gcdex` et n'oublions pas qu'il faut en fait travailler directement sur les polynômes.

```
> EE0:=X*E1 + E0;
                                  2
                EE0 := (124123 x  + 234234) X + 234234 x + 3234234
> Gcdex(EE0,Q,X,'IEE0','JEE0') mod q;
                                             1
> IEE0;
                  2                                                2
    203916487 X x  + 83557476 X x + 113630396 X + 138383902 x  +
180392990 x + 206406567
> Expand(EE0*IEE0+JEE0*Q) mod q;
                                             1
```

Quittons le programme Maple :

```
> quit();
bytes used=2516652, alloc=1113908, time=3.58
3.590u 0.940s 4:08.90 1.8% 0+1429k 158+0io 214pf+0w
```

Le temps de calcul total utilisé par Maple sur une SparcII est donc d'environ 4 secondes.

A.2 Axiom

Le logiciel axiom [1-NAG] est lui aussi un logiciel de calcul formel mais qui s'apparente plus à un langage de programmation fortement typé. Nous tenons à remercier Daniel Augot qui a bien voulu programmer le petit exemple précédent en Axiom afin de permettre la comparaison.

Le temps à considérer dans la comparaison est le temps d'évaluation (EV) [1]

A.2.a Anneaux modulaires

Commençons donc par fixer nos paramètres.

```
q:=234776683
   (1)  234776683
                                                    Type: PositiveInteger
                                               Time: 0.03 (OT) = 0.03 sec
A:=234675
   (2)  234675
                                                    Type: PositiveInteger
                                               Time: 0.02 (IN) = 0.02 sec
```

1. INput time, OTher times, Garbage Collector, EValuation time ; ce sont les valeurs affichées lorsque `set message time on` est utilisé.

Comme Axiom est typé, il faut se définir une variable de domaine, ici le corps premier

```
D:=PrimeField(q)
   (3)  PrimeField 234776683
                                                        Type: Domain
                                              Time: 0.02 (OT) = 0.02 sec
```

Le calcul de l'inverse modulaire est un peu surprenant,

```
1/(A::D)
   (4)  11532995
                                              Type: PrimeField 234776683
                                              Time: 0.05 (IN) = 0.05 sec
```

Continuons comme précédemment,

```
r:=8745287453
   (5)  8745287453
                                                   Type: PositiveInteger
                                                             Time: 0 sec
prime? r
   (6)  false
                                                        Type: Boolean
                                              Time: 0.02 (EV) = 0.02 sec
```

Nous avons là le premier temps d'évaluation à considérer. Reprenons alors notre exemple d'anneau non intègre.

```
D:=ZMOD(r)
   (7)  IntegerMod 8745287453
                                                        Type: Domain
                                              Time: 0.02 (OT) = 0.02 sec
1/(A::D)
   There are 11 exposed and 11 unexposed library operations named /
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                                )display op /
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.

   Cannot find a definition or applicable library operation named /
      with argument type(s)
                                PositiveInteger
                             IntegerMod 8745287453
```

Nous avons là une erreur à l'exécution car cette fonction n'existe pas hors d'un corps. Un élément inversible n'aurait pas non plus été inversé. Par contre, la fonction qui suit existe dans tout anneau

```
recip(A::D)
   (8)  "failed"
                                               Type: Union("failed",...)
                                              Time: 0.03 (OT) = 0.03 sec
```

Mais, comme pour Maple, Axiom se contente d'un message d'erreur et le pgcd doit se calculer manuellement si on souhaite obtenir un facteur du module.

```
gcd(r,A)
   (9)  7
                                                    Type: PositiveInteger
                                                 Time: 0.02 (IN) = 0.02 sec
A:=A/7
   (10)  33525
                                                 Type: Fraction Integer
                                                 Time: 0.02 (OT) = 0.02 sec
D:=ZMOD(r)
   (11)  IntegerMod 8745287453
                                                              Type: Domain
                                                 Time: 0.02 (OT) = 0.02 sec
1/(A::D)
   There are 11 exposed and 11 unexposed library operations named (...)

recip(A::D)
   (12)  5160567388
                                 Type: Union(IntegerMod 8745287453,...)
                     Time: 0.03 (IN) + 0.02 (EV) + 0.03 (OT) = 0.08 sec
```

Nous vérifions ici que la première méthode de calcul ne donne aucun résultat dans un anneau. La deuxième, par contre, fournit cette fois-ci l'inverse désiré. Un deuxième temps d'évaluation est ici à prendre en compte. Nous considérons maintenant à nouveau le polynôme (A.1).

```
e0:=124123
   (13)  124123
                                                    Type: PositiveInteger
                                                 Time: 0.02 (OT) = 0.02 sec
e1:=234234
   (14)  234234
                                                    Type: PositiveInteger
                                                 Time: 0.03 (OT) = 0.03 sec
e2:=3234234
   (15)  3234234
                                                    Type: PositiveInteger
                                                 Time: 0.02 (OT) = 0.02 sec
P:=x**3+e2*x**2+e1*x+e0
           3           2
   (16)  x  + 3234234x  + 234234x + 124123
                                               Type: Polynomial Integer
                     Time: 0.15 (IN) + 0.03 (EV) + 0.17 (OT) = 0.35 sec
D:=PrimeField(q)
   (17)  PrimeField 234776683
                                                              Type: Domain
                                                 Time: 0.02 (OT) = 0.02 sec
P::SUP D
           3           2
   (18)  ?  + 3234234?  + 234234? + 124123
                Type: SparseUnivariatePolynomial PrimeField 234776683
                                 Time: 0.28 (IN) + 0.03 (OT) = 0.32 sec
irreducible? %
   (19)  true
                                                            Type: Boolean
                                 Time: 0.02 (EV) + 0.02 (OT) = 0.03 sec
```

Nous retrouvons ici des principes assez voisins de ceux de notre librairie, puisque l'évaluation se fait une fois le domaine fixé. Celle-ci prend à nouveau 0.02 seconde. Mais que se passe-t-il si l'anneau n'est pas intègre ?

```
D:=ZMOD(r)
   (20)  IntegerMod 8745287453
                                                          Type: Domain
                                              Time: 0.03 (OT) = 0.03 sec
P::SUP D
          3           2
   (21)  ?  + 3234234?  + 234234? + 124123
              Type: SparseUnivariatePolynomial IntegerMod 8745287453
                                Time: 0.20 (IN) + 0.03 (OT) = 0.23 sec
irreducible? %
   There are 1 exposed and 0 unexposed library operations named
      irreducible? (...)
```

Ici encore, Axiom refuse de répondre car la fonction en question n'existe que dans un corps. Cette fois-ci, par contre, il n'existe pas de solution de rechange, sauf à reprogrammer l'algorithme.

A.2.b Extension simple

Voyons maintenant comment gérer les extensions polynomiales en Axiom.

```
K:=FiniteFieldExtensionByPolynomial(PrimeField q,P:: SUP PrimeField q)
   (22)
  FiniteFieldExtensionByPolynomial(PrimeField 234776683,
?**3+3234234*?*?+234234
  *?+124123)
                                                          Type: Domain
                                Time: 0.30 (IN) + 0.03 (OT) = 0.33 sec
x:=generator()$K
   (23)  %A
Type: FiniteFieldExtensionByPolynomial(PrimeField 234776683,
?**3+3234234*?*?+234234*?+124123)
                                Time: 0.02 (IN) + 0.02 (EV) = 0.03 sec
E0:=e1*x+e2
   (24)  234234%A + 3234234
Type: FiniteFieldExtensionByPolynomial(PrimeField 234776683,
?**3+3234234*?*?+234234*?+124123)
                                Time: 0.03 (IN) + 0.03 (OT) = 0.07 sec
1/E0
                     2
   (25)  226257462%A  + 25554697%A + 70525059
Type: FiniteFieldExtensionByPolynomial(PrimeField 234776683,
?**3+3234234*?*?+234234*?+124123)
                   Time: 0.03 (IN) + 0.02 (EV) + 0.02 (OT) = 0.07 sec
```

Nous retrouvons ici une construction beaucoup plus mathématique que celle de Maple. Cette séquence d'instruction resemble donc assez à ce qui sera écrit en utilisant ZEN. Au total, cette inversion dans une

extension prend un temps d'évaluation supplémentaire de 0.04 seconde.

```
X:SUP K:=monomial(1,1)
   (27)  ?
Type: SparseUnivariatePolynomial FiniteFieldExtensionByPolynomial(
PrimeField 234776683,?**3+3234234*?*?+234234*?+124123)
                                          Time: 0.07 (OT) = 0.07 sec
Q:=X**2+E1*X+E0
          2            2
   (28)  ?  + (124123%A  + 234234)? + 234234%A + 3234234
Type: SparseUnivariatePolynomial FiniteFieldExtensionByPolynomial(
PrimeField 234776683,?**3+3234234*?*?+234234*?+124123)
                              Time: 0.02 (IN) + 0.08 (OT) = 0.10 sec
irreducible? %
   (29)  true
                                                      Type: Boolean
                              Time: 0.02 (EV) + 0.02 (OT) = 0.03 sec
```

Là encore, on peut remarquer la supériorité d'Axiom sur Maple. Le temps d'évaluation est d'ailleurs symbolique du fait qu'Axiom est certainement mieux programmé que Maple, tout du moins en ce qui concerne les extensions polynomiales.

A.2.c Double extension

C'est ici qu'Axiom va réellement devenir intéressant. En effet, alors que Maple ne sait pas manipuler des extensions d'extensions, la programmation typée d'Axiom le permet.

```
KK:=FFP(K,Q)
   (30)
  FiniteFieldExtensionByPolynomial(FiniteFieldExtensionByPolynomial(PrimeField
  234776683,?**3+3234234*?*?+234234*?+124123),?*?+(124123*%A*%A+234234)*?+23423
  4*A+3234234)
                                                      Type: Domain
                              Time: 0.03 (IN) + 0.07 (OT) = 0.10 sec
y:KK:=generator()
   (31)  %B
Type: FiniteFieldExtensionByPolynomial(FiniteFieldExtensionByPolynomial(
PrimeField 234776683,?**3+3234234*?*?+234234*?+124123),
?*?+(124123*%A*%A+234234)*?+234234*A+3234234)
                                                        Time: 0 sec
EE0:=y*E1 + E0;
Type: FiniteFieldExtensionByPolynomial(FiniteFieldExtensionByPolynomial(
PrimeField 234776683,?**3+3234234*?*?+234234*?+124123),
?*?+(124123*%A*%A+234234)*?+234234*A+3234234)
                              Time: 0.17 (IN) + 0.05 (OT) = 0.22 sec
1/EE0
   (33)
                 2                                              2
     (203916487%A  + 83557476%A + 113630396)%B + 138383902%A  + 180392990%A
   +
     206406567
Type: FiniteFieldExtensionByPolynomial(FiniteFieldExtensionByPolynomial(
PrimeField 234776683,?**3+3234234*?*?+234234*?+124123),
?*?+(124123*%A*%A+234234)*?+234234*A+3234234)
                 Time: 0.03 (IN) + 0.03 (EV) + 0.07 (OT) = 0.13 sec
```

Ici, comme dans le cas de ZEN, la même syntaxe de fonction permet d'obtenir le résultat de l'opération désirée, alors que le domaine utilisé est fondamentalement différent. Voici le temps total d'exécution calculé à partir des sorties ci-dessus

IN	EV	OT	Total
1.45	0.18	1.05	2.66

L'écart de 0.02 seconde est sans doute du à des arrondis. En ne considérant que le temps d'évaluation, on néglige de ce fait les temps des typages qui sont pourtant indispensables à Axiom. Retenons pourtant ce chiffre de 0.18 seconde qui correspond donc exclusivement aux calculs effectués. Nous allons ainsi pouvoir vérifier que la programmation utilisant ZEN est encore plus efficace que la programmation interne d'Axiom.

A.3 Comparaison avec ZEN

VOYONS MAINTENANT UN PROGRAMME ÉCRIT en C et utilisant ZEN qui réalise exactement les mêmes opérations. La sortie de ce programme est donnée ci-après :

```
q=234776683
A=234675
1/A mod q = 11532995
r=8745287453
A=234675
Elément non inversible : facteur du module = 7
P=(1)*X^3+(3234234)*X^2+(234234)*X+(124123) est irréductible.
PR=(1)*X^3+(3234234)*X^2+(234234)*X+(124123)
Elément non inversible : facteur du module = 19
RootOf(P)=t
RootOf(P)=x
E0=(0)*x^2+(234234)*x+(3234234)
1/E0 mod q = (226257462)*x^2+(25554697)*x+(70525059)
E1=(124123)*x^2+(234234)
Q=((0)*x^2+(1))*X^2+((124123)*x^2+(234234))*X+((0)*x^2+(234234)*x+
(3234234)) est irréductible.
RootOf(Q)=y
EE0=((124123)*x^2+(234234))*y+((0)*x^2+(234234)*x+(3234234))
1/EE0=((203916487)*x^2+(83557476)*x+(113630396))*y+((138383902)*x^2+
(180392990)*x+(206406567))
0.09u 0.09s 0:00.48 37.5%
```

On peut constater que l'exécution du programme est environ 40 fois plus rapide que celle en Maple. Ce chiffre est en fait largement sous-estimé car les opérations effectuées ici sont beaucoup trop simples pour permettre de juger de l'efficacité de la librairie. Même par rapport à Axiom, le temps d'exécution, qui comprend bien évidemment d'avantage d'opérations que la simple évaluation des calculs, est plus faible.

La programmation est donnée ci-dessous. Elle est évidemment plus complexe et moins lisible que les commandes Maple, mais rappelons que les buts recherchés ne sont pas les mêmes. Nous payons ici le prix de l'efficacité. Néanmoins, le respect des principes de la librairie ZEN (voir § I.2.3) rend la programmation relativement aisée et compréhensible, car somme toute assez proche d'une programmation Axiom :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "zen.h"
4
5  main()
6  {BigNum q,r;
7   int ql,rl;
8   ZENElt A,I,E0,E1,EE0;
```

```
 9  ZENRing K,R,E,EE;
10  ZENPoly P,Q,PR;
11
12  ZBNReadFromString(&q,&ql,"234776683",10);
13  printf("q="); ZBNPrintToFile(stdout,q,ql,10); printf("\n");
14  ZENBaseRingAlloc(K,q,ql);
15  ZENEltAlloc(A,K); ZENEltReadFromString(A,"234675",10,K);
16  printf("A="); ZENEltPrintToFile(stdout,A,10,K); printf("\n");
17
18  ZENEltAlloc(I,K); ZENEltInverse(I,A,K);
19  printf("1/A mod q = "); ZENEltPrintToFile(stdout,I,10,K); printf("\n");
20
21  ZENEltFree(A,K); ZENEltFree(I,K);
22
23  ZBNReadFromString(&r,&rl,"8745287453",10);
24  printf("r="); ZBNPrintToFile(stdout,r,rl,10); printf("\n");
25  ZENBaseRingAlloc(R,r,rl);
26
27  ZENEltAlloc(A,R); ZENEltReadFromString(A,"234675",10,R);
28  printf("A="); ZENEltPrintToFile(stdout,A,10,R); printf("\n");
29
30  ZENEltAlloc(I,R); if(ZENEltInverse(I,A,R)==ZEN_NO_INVERSE) {
31    printf("Elément non inversible : facteur du module = ");
32    ZENEltPrintToFile(stdout,ZENRingFact(R),10,R); printf("\n"); }
33
34  ZENEltFree(A,R); ZENEltFree(I,R);
35
36  ZENPolyReadFromString(P,"(1)*X^3+(3234234)*X^2+(234234)*X+(124123)",10,K);
37  printf("P="); ZENPolyPrintToFile(stdout,P,10,K);
38  if(ZENPolyIsNotIrreducible(P,K))
39    printf(" n'est pas irréductible.\n");
40  else printf(" est irréductible.\n");
41
42  ZENPolyReadFromString(PR,"(1)*X^3+(3234234)*X^2+(234234)*X+(124123)",10,R);
43  printf("PR="); ZENPolyPrintToFile(stdout,PR,10,R);
44  switch(ZENPolyIsNotIrreducible(PR,R)) {
45  case ZEN_NO_INVERSE:
46    printf("\nElément non inversible : facteur du module = ");
47    ZENEltPrintToFile(stdout,ZENRingFact(R),10,R); printf("\n"); break;
48  case ZENPOLY_IS_IRREDUCIBLE:
49    printf(" est irréductible ?\n"); break;
50  default:
51    printf(" n'est pas irréductible.\n"); break; }
52
53  ZENPolyFree(PR,R); ZENRingClose(R);
54
55  ZENExtRingAlloc(E,P,K); ZENPolyFree(P,K);
56  printf("RootOf(P)=%c\n",ZENRingVar(E));
57  ZENRingVar(E)='x'; printf("RootOf(P)=%c\n",ZENRingVar(E));
58
59  ZENEltAlloc(E0,E); ZENEltReadFromString(E0,"(0)*x^2+(234234)*x+(3234234)",10,E);
60  printf("E0="); ZENEltPrintToFile(stdout,E0,10,E); printf("\n");
61
62  ZENEltAlloc(E1,E); ZENEltInverse(E1,E0,E);
```

```
63   printf("1/E0 mod q = "); ZENEltPrintToFile(stdout,E1,10,E); printf("\n");
64
65   ZENEltReadFromString(E1,"(124123)*x^2+(234234)",10,E);
66   printf("E1="); ZENEltPrintToFile(stdout,E1,10,E); printf("\n");
67   /* Définition par coefficients du polynôme Q */
68   ZENPolyAlloc(Q,2,E); ZENPolySetToXi(Q,2,E);
69   ZENPolySetCoeff(Q,1,E1,E); ZENPolySetCoeff(Q,0,E0,E);
70   printf("Q="); ZENPolyPrintToFile(stdout,Q,10,E);
71   if(ZENPolyIsNotIrreducible(Q,E))
72     printf(" n'est pas irréductible.\n");
73   else printf(" est irréductible.\n");
74
75   ZENExtRingAlloc(EE,Q,E); ZENPolyFree(Q,E);
76   printf("RootOf(Q)=%c\n",ZENRingVar(EE));
77   /* Définition d'un élément de l'extension sous forme polynomiale */
78   ZENEltAlloc(EE0,EE); ZENPolySetToXi(ZENElt2Pol(EE0,EE),1,E);
79   ZENPolySetCoeff(ZENElt2Pol(EE0,EE),1,E1,E); ZENPolySetCoeff(ZENElt2Pol(EE0,EE),0,E0,E);
80   printf("EE0="); ZENEltPrintToFile(stdout,EE0,10,EE); printf("\n");
81
82   ZENEltFree(E0,E); ZENEltFree(E1,E);
83
84   ZENEltAlloc(E1,EE); ZENEltInverse(E1,EE0,EE);
85   printf("1/EE0="); ZENEltPrintToFile(stdout,E1,10,EE); printf("\n");
86
87   ZENEltFree(E1,EE); ZENEltFree(E0,E);
88   ZENRingClose(EE); ZENRingClose(E); ZENRingClose(K);
89   exit(0);
90 }
```

Annexe B

Références bibliographiques

Les références rassemblées ci-après sont ordonnées par ordre alphabétique dans chaque sous-section. L'ordre des sections correspond peu ou prou à l'ordre inverse de disponibilité des documents. Certains de ces documents sont aussi disponibles sur le réseau Internet aux adresses indiquées section B.1. C'est souvent le moyen le plus rapide d'obtenir le document cité, mais l'adresse indiquée peut être fluctuante dans le temps, voire disparaître.

B.1 Fichiers du réseau Internet

[1-AB94] Les Tchigaboux. *Alice et Bob*,
`http://www.iro.umontreal.ca/labs/theorique/Alice-Bob.html`

[1-Cha95] F. Chabaud. *Irreducible polynomials over GF(2)*,
`http://www.dmi.ens.fr/~chabaud/Poly/GF.html`

[1-CL96] F. Chabaud, and R. Lercier. *The Zen library*,
`http://lix.polytechnique.fr/~zen/`

[1-Gra95] Granboulan, L. *Méthodes itératives de résolution de systèmes linéaires creux. Parallélisation et applications en théorie des nombres*,
`http://www.dmi.ens.fr/dmi/equipes_dmi/grecc/granboul/data/creux.ps`

[1-Kan96] Technische Universität Berlin, *KASH: KANT Shell (Komputational Algebraic Number Theory)*
`ftp://ftp.math.tu-berlin.de/pub/algebra/Kant/Kash/Binaries`

[1-KL] D.E. Knuth, and S. Levy. *CWEB*.
`http://hpux.ask.uni-karlsruhe.de/man/Text/cweb-3.4d/`

[1-LiD95] LiDIA - Group *LiDIA Manual A library for computational number theory*,
`http://www.umbc.edu/pdsrc/docs/lidiahtml/lidiahtml.html`

[1-MorBM] F. Morain. *Modular arithmetic: the BigMod library*, contained in [1-MorEC]

[1-MorEC] F. Morain. *Elliptic Curves and Primality Proving (ECPP)*,
`ftp://ftp.inria.fr/INRIA/ecpp.V3.4.1.tar.Z`

[1-NAG] Numerical Algorithms Group *Axiom*,
`http://www.nag.co.uk:80/symbolic/AX.html`

[1-Par] *Pari/GP 1-39*.
`ftp://megrez.math.u-bordeaux.fr/pub/pari/`

B.2 Livres

B.2.a Ouvrages théoriques

[2-KS60] J.G. Kemeny and J.L. Snell. *Finite Markov chains.* D. van Nostrand, Princeton, 1960.

[2-Knu81] D.E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 2nd edition, 1981.

[2-Knu81a] D.E. Knuth. *Fundamental Algorithms.* Volume 1 of [2-Knu81].

[2-Knu81b] D.E. Knuth. *Seminumerical Algorithms.* Volume 2 of [2-Knu81].

[2-GJ78] M.R. Garey, and D. Johnson. *Computers and intractability, a guide to the theory of NP-completeness* W.H. Freeman and co., New-York, 1978.

[2-LN86] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, 1986.

[2-MWS83] F.J. MacWilliams and N.J.A. Sloane. *The theory of Error-Correcting Codes* North-Holland, 1983.

[2-PW80] W.W. Peterson and E.J. Weldon. *Error-correcting codes*, MIT Press, 2nd edition, 1980.

[2-PH89] A. Poli et Ll. Huguet. *Codes correcteurs*, Masson, 1989.

[2-RDO85] E. Ramis, C. Deschamps et J. Odoux. *Cours de Mathématiques spéciales*, volume 1 : algèbre. Masson, 1985.

[2-Ste90] J. Stern. *Fondements mathématiques de l'informatique*, McGraw-Hill, 1990.

[2-Sti96] D.R. Stinson. Traduction française de Serge Vaudenay. *Cryptographie : théorie et pratique,* traduction de S. Vaudenay, International Thomson Publishing, 1996.

[2-Til94] J. van Tilburg. *Security-analysis of a class of cryptosystems based on linear error-correcting codes*, Ph.D. thesis published by Royal PTT Nederland, september 1994.

B.2.b Manuels de logiciels

[3-AB92] M.L. Abell and J.P. Braselton. *The Mathematica Handbook*, Academic Press, 1992.

[3-CGG+90] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *Maple's reference manual*, 5th edition, 1990.

[3-DuB93] P. DuBois. *Software portability with imake*, O'Reilly & Associates, 1993.

[3-Lam94] L. Lamport. *LaTeX A documentation preparation system, Second edition*, Addison-Wesley, 1994.

B.3 Publications scientifiques

B.3.a Articles de revues

[4-AGH+90] N. Alon, O. Goldreich, J. Håstad, and R. Peralta. Simple constructions of almost k-wise independent random variables. 31th Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, IEEE Computer Society Press, 544–553, October 22-24, 1990 .

[4-ACN92] D. Augot, and P. Charpin, and N. Sendrier. Studying the locator polynomials of minimum weight codewords of BCH codes. *IEEE Trans. Inform. Theory*, IT-38(3):960–973, 1992.

[4-Ber73] E.R. Berlekamp. Goppa codes *IEEE Trans. Inform. Theory*, IT-19(5):590–592, Sep 1973.

[4-BMV78] E.R. Berlekamp, R.J. McEliece, and H.C.A. Van Tilborg. On the inherent intractability of certain coding problems *IEEE Trans. Inform. Theory*, IT-24(3):384–386, May 1978.

[4-BO88] E.F. Brickell, and A.M. Odlyzko. Cryptanalysis: a survey of recent results. A vérifier... *Proceedings of the IEEE*, 76(5):578–593, May 1988.

[4-Cam83] P. Camion. Improving an algorithm for factoring polynomials over a finite field and constructing large irreducible polynomials, *IEEE Trans. Inform. Theory*, IT-29(3):378–385, May 1983.

[4-CC96] A. Canteaut, and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to primitive narrow-sense BCH codes of length 511, to be published in *IEEE Trans. Inform. Theory*, also available by ftp [7-CC95b].

[4-CZ81] D.G. Cantor, and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of computation*,vol. 36, 587–592, 1981.

[4-Che94] K. Chen. Improved Girault identification scheme, *IEE Electronic letters*, 30(19), 1590–1591, Sep. 1994

[4-CR88] B. Chor, and R. Rivest. A knapsac-type public key cryptosystem based on arithmetic in finite fields, *IEEE Trans. Inform. Theory*, IT-34: 901–909, 1988.

[4-DH76] W. Diffie, and M.E. Hellman. New directions in cryptography, *IEEE Trans. Inform. Theory*, IT-22(6): 644–654, November 1976

[4-Gab85] E.M. Gabidulin. Theory of codes with maximum rank distance *Problems of information transmission*, translated from Russian by Plenum Pub. Corp. original paper: *Problemy Peredachi Informatsii*, vol. 21(1):3–16, Jan-Mar 1985.

[4-Gib95] J.K. Gibson. Severely denting the Gabidulin version of the McEliece cryptosystem, *Designs, Codes, and Cryptography*, 6:–, 1995

[4-GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems, *SIAM J. of computing*, 18: 186–208, 1989.

[4-HS73] H.J. Helgert, and R.D. Stinaff. Shortened BCH codes, *IEEE Trans. Inform. Theory*, 818–820, Nov 1973.

[4-KL72] T. Kasami, and S. Lin. Some results on the minimum weight of primitive BCH codes, *IEEE Trans. Inform. Theory*, 824–825, Nov 1972.

[4-KT69] T. Kasami, and N. Tokura. Some remarks on BCH bounds and minimum weights of binary primitive BCH codes, *IEEE Trans. Inform. Theory*, 408–413, May 1969.

[4-LS92] G. Lachaud, and J. Stern. Polynomial-time construction of codes I: linear codes with almost equal weights, *Applicable Algebra in Engineering, Communication and Computing*, pages 151–161, 1992.

[4-Leo88] J.S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Inform. Theory*, IT-34(5):1354–1359, September 1988.

[4-LH85] L.B. Levitin, and C.R.P. Hartman. A New Approach to the General Minimum Distance Decoding Problem: The Zero-Neighbors Algorithm, *IEEE Trans. Inform. Theory*, IT-31(3):378–384, May 1985.

[4-LDW94] Y.X. Li, R.H. Deng, and X.M. Wang. On the equivalence of McEliece's and Niederreiter's public-key cryptoystems. *IEEE Trans. Inform. Theory*, IT-40(1):271–273, January 1994.

[4-Omu72] J.K. Omura. Iterative decoding of linear codes by a modulo-2 linear program. *Discrete Math*, 3:193–208, 1972.

[4-Pie67] J.N. Pierce. Limit distributions of the minimum distance of random linear codes. *IEEE Trans. Inform. Theory*, 595–599, 1967.

[4-Sho96] P.W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, submitted to *SIAM J. of computing*, (see preliminary version [5-Sho94]), `http://netlib.att.com/math/people/shor/papers/index.html` 1996.

[4-Sho95] V. Shoup. A new polynomial factorization algorithm and its implementation. J. of Symbolic Computation, Vol. 20: 363–397, 1995.

[4-SS92] V.M. Sidel'nikovand S.O. Shestakov. On unsecurity of cryptosystems based on generalized Reed-Solomon codes. *Diskretnaya Math*, vol. 4, 57–63, 1992.

[4-Ste96] J. Stern. A new paradigm for public-key identification. to appear in *IEEE Trans. Inform. Theory*, 1996.

[4-Thi89] A.Thiong ly. A deterministic algorithm for factorizing polynomials over extensions $GF(p^m)$ of $GF(p)$, p a small prime. *J. of Information & Optimization Sciences*, Vol. 10(2): 337–344, 1989.

[4-Wie86] D.H. Wiedemann. Solving sparse linear equations over finite fields, *IEEE Trans. Inform. Theory*, IT-32:54–62, 1986.

B.3.b Articles de congrès scientifiques

[5-AM87] C. Adams, and H. Meijer. *Security related comments regarding McEliece's public-key cryptosystem*, Advances in cryptology, *CRYPTO'87,* Springer-Verlag, *LNCS* 293, 221–228, 1988.

[5-Cha94] F. Chabaud. *On the security of some cryptosystems based on error-correcting codes*, Advances in cryptology, *EUROCRYPT'94,* Springer-Verlag, *LNCS* 950, 131–139, 1995.

[5-Cha92] F. Chabaud. *Asymptotic analysis of probabilistic algorithms for finding short codewords*, *EUROCODE 92*, Springer-Verlag, *CISM Courses and Lectures* 339, 175–183, 1993.

[5-CS96] F. Chabaud, and J. Stern. The cryptographic security of the syndrome decoding problem for rank distance codes. to be published in *Asiacrypt'96*, Springer-Verlag, *LNCS*????, 1996.

[5-CV94] F. Chabaud, and S. Vaudenay. *Links between Differential and Linear Cryptanalysis*, Advances in cryptology, *EUROCRYPT'94,* Springer-Verlag, *LNCS* 950, 356–365, 1995. `ftp://ftp.ens.fr/pub/reports/liens/liens-94-3.A4.dvi`

[5-Che95] K. Chen. *A new identification algorithm*, Cryptography policy and algorithms conference, Springer-Verlag, *LNCS* 1029 244–249, 1996

[5-Dum92] I.I. Dumer. *Suboptimal decoding of linear codes*, *EUROCODE 92*, Springer-Verlag, *CISM Courses and Lectures* 339, 369–382, 1993.

[5-FS96] J.-B. Fischer, and J. Stern. *An efficient pseudo-random generator provably as secure as syndrome decoding,* Advances in cryptology, *EUROCRYPT'96,* Springer-Verlag, *LNCS* 1070, –, 1996.

[5-Gab91] E.M. Gabidulin. *A fast matrix decoding algorithm for rank-error-correcting codes*, Proc. of the 1st French-Soviet workshop *(Paris, 1991)*, 126–133, Springer-Verlag, *LNCS* 573, 1992.

[5-GK94] E.M. Gabidulin, and O. Kjelsen. *How to avoid the Sidel'nikov-Shestakov attack*, Coll. Error control, cryptology, and speech compression *(Moscow, 1993)*, 25–32, Springer-Verlag, *LNCS* 829, 1994.

[5-GPT91] E.M. Gabidulin, and A.V. Paramonov, and O.V. Tretjakov. *Ideals over a non-commutative ring and their application in cryptography*, Advances in cryptology, *EUROCRYPT'91,* Springer-Verlag, *LNCS* 547, 482–489, 1991.

[5-Gab94] E.M. Gabidulin. *On public-key cryptosystem based on linear codes: efficiency and weaknesses*, *Codes and ciphers, 4th IMA conf.*, –, IMA Press, 1995.

[5-Gib96] J.K. Gibson. *The security of the Gabidulin public-key cryptosystem*, Advances in cryptology, *EUROCRYPT'96,* Springer-Verlag, *LNCS* 1070, 212–223, 1996.

[5-Gib91] J.K. Gibson. *Equivalent Goppa codes and trapdoors to McEliece's public key cryptosystem*, Advances in cryptology, *EUROCRYPT'91,* Springer-Verlag, *LNCS* 547, 517–521, 1991.

[5-KT91] V.I. Korzhik, and A.I. Turkin. *Cryptanalysis of McEliece's public-key cryptosystem*, Advances in cryptology, *EUROCRYPT'91,* Springer-Verlag, *LNCS* 547, 68–70, 1991.

[5-LB88] P.J. Lee, and E.F. Brickell. *An observation on the security of McEliece's public-key cryptosystem*, Advances in cryptology, *EUROCRYPT'88,* Springer-Verlag, *LNCS* —, 275–280, 19–.

[5-Sen95] N. Sendrier. *Efficient generation of binary words of given weight Cryptography and coding, 5th IMA conf.* Springer-Verlag, *LNCS* 1025 –, 1996.

[5-Sho94] P.W. Shor. *Algorithms for quantum computation: Discrete logarithms and factoring,* 35th Annual Symposium on Foundations of Computer Science, Santa-Fe, NM, IEEE Computer Society Press, 124–134, November 20-22, 1994
`http://netlib.att.com/math/people/shor/papers/index.html`

[5-Ste89] J. Stern. *A method for finding codewords of small weight*, Coding Theory and Applications, Springer-Verlag, *LNCS* 388 106–113, 1989.

[5-Til88] J. van Tilburg. *On the McEliece public-key cryptosystem*, Advances in cryptology, *CRYPTO'88,* Springer-Verlag, *LNCS* 403, 119–131, 1990.

B.3.c Cours, actes de congrès ou séminaires scientifiques

[6-Bar93] S. Barg. Some new NP-complete coding problems. *Proc. 6th Swedish-Russian International Workshop on Information Theory*, Mölle, Sweden, 22-27 August 1993.

[6-CC94] A. Canteaut, and H. Chabanne. A further improvement of the work factor in an attempt at breaking McEliece's cryptosystem. *EUROCODE 94, Livre des résumés*, 169–173, INRIA, F-78153 Le Chesnay Cedex, oct 1994.

[6-Neu90] J. Neveu. Introduction aux probabilités, Cours de l'École Polytechnique, F-91128 Palaiseau Cedex, 1990.

B.3.d Rapports internes

[7-CC95a] A. Canteaut, and F. Chabaud. Improvements of the attacks on cryptosystems based on error-correcting codes Rapport 95-21, LIENS, 45, rue d'Ulm F-75230 Paris cedex 05, Jul 1995,
`ftp://ftp.ens.fr/pub/reports/liens/liens-95-21.A4.dvi`

[7-CC95b] A. Canteaut, and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to primitive narrow-sense BCH codes of length 511, to be publidhed in *IEEE Trans. Inform. Theory*, Rapport de recherche n° 2685, INRIA, F-78153 Le Chesnay Cedex, Oct 1995.
`ftp://ftp.inria.fr/INRIA/tech-reports/RR/RR-2685.ps.gz`

[7-CC93] H. Chabanne, and B. Courteau. Application de la méthode de décodage itérative d'Omura à la cryptanalyse du système de McEliece. Rapport 122, Université de Sherbrooke, Canada, 1993.

[7-Cha92] F. Chabaud. Recherche de mots de poids faible dans un code linéaire binaire aléatoire. Rapport de stage d'option, École Polytechnique, F-91128 Palaiseau Cedex, juin 1992.

[7-CL96] F. Chabaud and R. Lercier. Zen's reference manual. Rapport XXXX, LIX, École Polytechnique, F-91128 Palaiseau Cedex, 1996, in preparation, see also [1-CL96].

[7-FGP96] P. Flajolet, X. Gourdon, and D. Panario. Random polynomials and polynomial factorization, Rapport de recherche n° 2852, INRIA, F-78153 Le Chesnay Cedex, Mars 1996.
`ftp://ftp.inria.fr/INRIA/tech-reports/RR/RR-2852.ps.gz`

[7-McE78] R.J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report 42-44*, Jet Propulsion Lab. Calif. Inst. of Tech. pages 114–116, 1978.

[7-Sen95] N. Sendrier. On the structure of a randomly permuted concatenated code, Rapport de recherche n° 2460, INRIA, F-78153 Le Chesnay Cedex, Janvier 1995.

[7-SVH89] B. Serpette, J. Vuillemin, and J.C. Hervé. *BigNum: a portable and efficient package for arbitrary-precision arithmetic,* PRL Research Report #2, 1989.
`ftp://ftp.digital.com/pub/DEC/PRL/research-reports/PRL-RR-2.ps.Z` source files included in [1-MorEC]

[7-Zim87] P. Zimmermann. Multiplication rapide en le_lisp. Rapport de stage d'option, École Polytechnique, F-91128 Palaiseau Cedex, juillet 1987.

B.3.e Mémoires

[8-Cha93] F. Chabaud. Sécurité des crypto-systèmes de McEliece, *Mémoire de DEA*, 1993.

[8-Hei87] R. Heiman. On the security of cryptosystems baed on linear error-correcting codes *MSc. thesis* Feinberg graduate school of the Weizmann Institute of science, August 1987

[8-MD5] R. Rivest. The MD5 Message-Digest Algorithm, *Network Working Group Request for Comments: 1321*, April 1992.
`http://theory.lcs.mit.edu/ rivest/Rivest-MD5.txt`

[8-SHA] Secure Hash Standard. *Federal Information Processing Standard Publication # 180-1*, U.S. Department of Commerce, National Institute of Standards and Technology, 1995 (addendum to [8-SHS]).

[8-SHS] Secure Hash Standard. *Federal Information Processing Standard Publication # 180*, U.S. Department of Commerce, National Institute of Standards and Technology, 1993.

[8-Vau95] S. Vaudenay. La sécurité des primitives cryptographiques. *Thèse de doctorat*, Univ. de Paris VII, avril 1995

[8-Vér95] P. Véron. Problème SD, Opérateur Trace, Schémas d'identification et codes de Goppa, *Thèse de doctorat*, Univ. de Toulon, juillet 1995

B.3.f Divers

[9-LMS95] J.B. Lacy, D.P. Mitchell, and W.M. Schell. Cryptolib: Cryptography in software. *Rumpsession Crypto'95*, 1995.

Annexe C

Table des figures

Annexe D

Index

Nous présentons dans cette thèse Zen, une librairie de calcul dans toute extension finie d'un anneau fini. Cette librairie permet une programmation en C relativement aisée et efficace.

Nous présentons ensuite des applications qui utilisent cette librairie pour la cryptanalyse de systèmes basés sur les codes correcteurs d'erreur. Il s'agit en fait pricipalement de deux algorithmes de décodage général des codes linéaires utilisant la distance de Hamming d'une part, la distance de rang de Gabidulin d'autre part.

Nous en déduisons quelques résultats :

- la distance minimale réelle de codes BCH de longueur 511 ;
- une borne sur la distance minimale des codes linéaires utilisant la distance de Gabidulin ;
- le sous-dimensionnement d'un système d'identification proposé par Véron ;
- la cryptanalyse réaliste d'un système d'identification proposé par Chen.

Enfin, ces algorithmes constituent à notre connaissance les meilleures attaques actuelles, quoique non réalisables, sur une large variété de systèmes dont les plus connus sont ceux de McEliece, Niederreiter et Stern-SD.

Mots clés : Calcul dans les corps finis, Codes BCH, Codes correcteurs d'erreur, Codes de Gabidulin, Cryptographie, Cryptosystème de McEliece, Décodage par syndrome, Distance minimale, Extensions polynomiales, Métrique de Hamming, Métrique de rang.

We present in this thesis Zen, a programming library for computing in every finite extension over a finite ring. This library gives an easy and efficient way to implement in C such computations.

We then introduce some applications that were implemented using this library. These applications are related to the cryptanalysis of some cryptosystems based on error-correcting codes. Mainly, we present two algorithms for general decoding of linear error-correcting codes, using Hamming's metric on one hand, and Gabidulin's rank metric on the other.

We derive a few results

- the true minimum distance of some BCH codes of length 511,
- a bound on the minimum distance of linear codes using Gabidulin's metric,
- the underestimation of some parameters in an identification scheme proposed by Véron,
- a realist cryptanalysis of another identification scheme proposed by Chen.

We also note that our algorithms seems to be, to our knowledge, the best attacks against a large variety of cryptosystems including the well-known McEliece's, Niederreiter's and Stern's SD cryptosystems.

Keywords: BCH codes, Computations in finite fields, Cryptography, Error-correcting codes, Gabidulin codes, Hamming metric, McEliece cryptosystem, Minimum weight codewords, Polynomial extensions, Rank metric, Syndrome decoding.