Décodage des codes de Reed-Solomon

Fabien Arlotto et Marion Candau

Master 1 Cryptologie et Sécurité Informatique
Université Bordeaux 1

12 avril 2010

Table des matières

Introduction

Nous vivons aujourd'hui dans une société d'informations. Ces informations sont transmises d'un bout à l'autre du monde à la vitesse de la lumière et en temps réel. Malheureusement les technologies actuelles ne permettent pas d'assurer une transmision parfaite, d'inévitables erreurs de transmission se produisent en permanence et il a fallu parer à ce phénomène. Une des parades possibles réside dans les codes correcteurs d'erreurs.

Le mode d'action de ces codes correcteurs d'erreurs repose sur le principe suivant : au lieu d'envoyer les données à transmettre directement dans le canal de transmission, celles-ci sont préalablement encodées, c'est-à-dire qu'il leur est adjoint une petite quantité d'information supplémentaire, appelée redondance, de façon à ce qu'il soit possible, à la réception de la communication, de détecter d'éventuelles erreurs apparues pendant la transmission et alors de les corriger, c'est-à-dire retrouver les données qui ont été altérées au cours de leur transfert.

Dans ce projet nous nous intéresserons à un certain type de code, les codes de Reed-Solomon, ainsi qu'à différentes méthodes pour les décoder.
L'intérêt de ces codes est que ce sont des codes de paramètres optimaux c'est-à-dire qu'ils ont le meilleur rapport possible entres le nombre d'erreurs qu'ils permettent de corriger par rapport à la redondance qu'ils introduisent. De plus, il existe des algorithmes de décodage très efficaces. Ils sont donc très utilisés, par exemple dans les CD (où l'on utilise deux codes entrelacés), dans la transmission des données par ADSL ou par satellite, ou encore dans l'exploration spaciale.

Historique

L'article sur les codes de Reed-Solomon a été soumis par Irving Reed et Gustave Solomon au *Journal of the Society for Industrial and Applied Mathematics* le 21 janvier 1959 et a été publié en juin 1960 sous le titre « Polynomial Codes over Certain Finite Fields ».
Après la découverte des codes de Reed-Solomon en 1960, des recherches ont été initiées pour trouver un algorithme de décodage efficace. Dans leur article publié en 1960, Reed et Solomon ont proposé un algorithme de décodage basé sur la résolution d'un système d'équations. Cet algorithme n'était utilisable que pour les codes de longueur assez petite. Au début des années 60, les progrès dans la recherche d'un algorithme efficace de décodage étaient lents mais réguliers.
La percée est venue en 1967 quand Berlekamp a trouvé un algorithme efficace de décodage des codes non binaires dont les codes de Reed-Solomon.

En 1968, Massey a montré que le problème du décodage des codes cycliques est équivalent au problème de la synthèse du plus court registre à décalage avec rétroaction linéaire capable de générer une suite donnée. Massey a proposé ensuite un algorithme de décodage des codes cycliques basé sur un registre à décalage rapide, qui est équivalent à l'algorithme de Berlekamp. Cette approche basée sur les registres à décalage est désormais communément dénommé l'algorithme de Berlekamp-Massey.
En 1975, Sugiyama, Kasahara, Hirasawa ont montré que l'algorithme d'Euclide pouvait être aussi utilisé pour décoder efficacement les codes de Reed-Solomon. Une nouvelle approche, adoptée par Welch et Berlekamp en 1986, est de convertir le problème de décodage à un problème d'interpolation qui peut alors être résolu par l'algorithme de Berlekamp-Welch.

Objectifs

L'objectif de notre projet est de réaliser un programme qui décode efficacement les codes de Reed-Solomon de toutes tailles et de toutes dimensions à l'aide de la méthode utilisant l'algorithme d'Euclide du cours d'Arithmétique du premier semestre de notre master.

Plan du projet

Dans la première partie, nous verrons les principes du décodage de ces codes, c'est-à-dire les concepts et l'algorithme de décodage. Dans la deuxième partie, sera abordée l'implémentation de l'algorithme de décodage. Puis dans les troisièmes et quatrièmes parties, nous verrons successivement les résultats qui en découlent et les améliorations possibles. Enfin, ce que l'on peut conclure de ce projet sera abordé dans la dernière partie.

1 Principes du décodage des codes de Reed Solomon

1.1 Préliminaires

1.1.1 Codes

Un code de longueur n est une partie de $\mathbb{F}_q^n$ où $\mathbb{F}_q$ est un corps fini.
Un code linéaire de longueur n est un sous-espace vectoriel de $\mathbb{F}_q^n$.
La distance de Hamming entre deux vecteurs de $\mathbb{F}_q^n$ est le nombre de coordonnées qui diffèrent entre ces deux vecteurs. On définit alors la distance

minimale d'un code comme étant la plus petite distance non nulle entre deux de ses mots.
À un code linéaire on associe trois paramètres : $[n, k, d]$, où n est sa longueur, k sa dimension en tant que $\mathbb{F}_q$-espace vectoriel et d sa distance minimale.
Un code linéaire de paramètres $[n, k, d]$ permet la correction de, au maximum, $\left\lfloor \frac{d-1}{2} \right\rfloor$ erreurs.

1.1.2 Codes de Reed-Solomon

Soit $\mathcal{P} = \{\alpha_1, \ldots, \alpha_n\}$, une partie à n éléments de $\mathbb{F}_q$. Un code de Reed-Solomon est l'ensemble des vecteurs de $\mathbb{F}_q^n$ qui s'écrivent sous la forme $(f(\alpha_1), \ldots, f(\alpha_n))$ où $f(X)$ est un polynôme de $\mathbb{F}_q[X]$ de degré strictement inférieur à $k \leqslant n$. Les codes de Reed-Solomon sont des codes linéaires de paramètres $[n, k, n-k+1]$, ils permettent donc de corriger $\left\lfloor \frac{n-k}{2} \right\rfloor$ erreurs.

1.1.3 Objectif du décodage

L'objectif du décodage est de retrouver le mot de code émis à partir du mot de code reçu, potentiellement erroné. Pour cela, avec des codes de Reed-Solomon, on peut procéder de plusieurs manières, la plus évidente est de retrouver le polynôme $f(X)$ correspondant au mot de code émis, mais il est aussi possible de seulement déterminer les emplacements des symboles erronés du mot de code reçu, puis de retrouver les bons symboles qui auraient dû se trouver à ces emplacements.

1.2 Première méthode : décodage par l'algorithme d'Euclide

Nous allons définir deux objets mathématiques nécessaires à la suite :

Polynôme d'interpolation
Soit $r = (r_1, \ldots, r_n)$ le mot de code reçu, et $L_i(X)$ le i-ème polynôme de Lagrange associé aux points $(\alpha_j)_{1 \leqslant j \leqslant n}$, $i \in [\![1, n]\!]$, défini ainsi :

$$L_i(X) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{X - \alpha_j}{\alpha_i - \alpha_j} \tag{1}$$

Les polynômes de Lagrange ont ainsi la propriété :

$$\forall j \in [\![1, n]\!],\ L_i(\alpha_j) = \begin{cases} 1 & \text{si } j = i \\ 0 & \text{sinon} \end{cases}$$

On construit alors le polynôme d'interpolation du mot de code reçu r :

$$R(X) = \sum_{i=1}^{n} r_i \times L_i(X) \tag{2}$$

On a donc :

$$\forall i \in [\![1, n]\!],\ R(\alpha_i) = r_i$$

Polynôme localisateur d'erreur

Soit $c = (c_1, \ldots, c_n)$ le mot de code associé à $f(X)$, c'est-à-dire le mot de code envoyé.
Le polynôme localisateur d'erreur $E(X)$ est :

$$E(X) = \prod_{\substack{i \\ r_i \neq c_i}} (X - \alpha_i)$$

Le nombre d'erreurs corrigibles est borné par les caractéristiques du code, on a donc : $\deg(E(X)) \leqslant \left\lfloor \frac{n-k}{2} \right\rfloor$.

Notons :

$$Q(X) = \prod_{i=1}^{n} (X - \alpha_i) \tag{3}$$

On a :

$$\forall i \in [\![1, n]\!],\ R(\alpha_i)E(\alpha_i) = f(\alpha_i)E(\alpha_i)$$

D'où :

$$R(X)E(X) = f(X)E(X) \mod Q(X)$$

et donc :

$$R(X)E(X) = f(X)E(X) + k(X)Q(X)$$

et on obtient :

$$\frac{R(X)}{Q(X)} - \frac{k(X)}{E(X)} = \frac{f(X)}{Q(X)} \tag{4}$$

Les définitions et théorèmes suivants vont nous permettre de conclure :

Développement en fraction continue

L'écriture $[a_0, a_1 \ldots, a_m]$ où $a_i \in \mathbb{F}_q[X]$ désigne la fraction rationnelle, appelée fraction continue :

$$Z(X) = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots + \cfrac{1}{a_m}}}} \tag{5}$$

Une telle écriture se construit en faisant des divisions euclidiennes successives du numérateur par le dénominateur de $Z(X)$ et les a_i sont ainsi les quotients successifs de ces divisions.

On définit les suites (p_k) et (q_k), $k \in [\![0, m]\!]$, d'éléments de $\mathbb{F}_q(X)$ par

$$\begin{cases} p_0 &= a_0 \\ q_0 &= 1 \end{cases} \qquad \begin{cases} p_1 &= a_0 a_1 + 1 \\ q_1 &= a_1 \end{cases}$$

et $\forall k \geqslant 2$:

$$\begin{cases} p_k &= a_k p_{k-1} + p_{k-2} \\ q_k &= a_k q_{k-1} + q_{k-2} \end{cases} \tag{6}$$

La fraction rationnelle $\dfrac{p_k}{q_k}$ est la k-ème réduite de la fraction continue.

Théorème.[1]
Si $Z(X)$ et $\dfrac{p}{q}$ sont des fractions rationnelles telles que :

$$\deg\left(Z(X) - \frac{p}{q}\right) < -2\deg q$$

alors $\dfrac{p}{q}$ est une réduite du développement en fraction continue de $Z(X)$.

Or :

$$\deg \frac{f(X)}{Q(X)} < k - n \leqslant -2\deg(E(X))$$

[1] On renvoie au cours d'arithmétique de notre master [1] pour la démonstration.

D'où $\frac{k(X)}{E(X)}$ est une des réduites en fraction continue de $\frac{R(X)}{Q(X)}$.
Pour retrouver $f(X)$ il suffit donc de tester, pour toutes les réduites $\frac{k(X)}{E(X)}$ de dénominateur de degré $\leqslant \left\lfloor \frac{n-k}{2} \right\rfloor$ si :

$$R(X) - \frac{k(X)Q(X)}{E(X)} \tag{7}$$

est un polynôme de degré $< k$. Si c'est le cas, il s'agit de $f(X)$.

Cependant, pour décoder avec la méthode ci-dessus, il faut interpoler le mot de code reçu, et dès qu'il s'agit de code de longueur moyenne (par exemple 255) cela coûte cher. Ceci motive la deuxième méthode énoncée ci-dessous.

1.3 Deuxième méthode de décodage

Cette deuxième méthode est en fait une amélioration de la première, elle repose sur le fait que pour calculer la réduite de $\frac{R(X)}{Q(X)}$ égale à $\frac{k(X)}{E(X)}$, il n'est pas indispensable de connaître entièrement le polynôme $R(X)$, seuls les coefficients d'un certain nombre de ses termes de plus haut degré sont nécessaires.
Voyons combien de coefficients faut-il conserver au minimum.

Soit l le nombre de termes de petit degré dont on peut se passer. Effectuons la division euclidienne de $R(X)$ par X^l :

$$R(X) = X^l R_1(X) + R_2(X),\ \deg(R_2(X)) \leqslant l-1$$

On a donc, d'après (4) :

$$\frac{X^l R_1(X) + R_2(X)}{Q(X)} - \frac{k(X)}{E(X)} = \frac{f(X)}{Q(X)}$$

Ou encore :

$$\frac{X^l R_1(X)}{Q(X)} - \frac{k(X)}{E(X)} = \frac{f(X) - R_2(X)}{Q(X)}$$

Comme $\deg(f(X)) < k$, on a :

$$\deg\left(\frac{f(X) - R_2(X)}{Q(X)}\right) < \max(k,l) - n = \max(k-n, l-n)$$

Or on sait que : $\deg(E(X)) \leqslant \left\lfloor \frac{n-k}{2} \right\rfloor$. On veut donc, pour pouvoir appliquer le théorème à la fraction $\frac{X^l R_1(X)}{Q(X)}$:

$$\max(k-n, l-n) \leqslant -2\deg(E(X)) \leqslant -2 \times \left\lfloor \frac{n-k}{2} \right\rfloor \leqslant k-n$$

Pour cela, il suffit que :

$$l - n \leqslant k - n$$

C'est-à-dire :

$$l \leqslant k$$

D'où en écrivant $R(X) = X^k R_1(X) + R_2(X)$ avec $\deg(R_2(X)) < k$, on obtient que $\frac{k(X)}{E(X)}$ est également une réduite du développement en fraction continue de $\frac{X^k R_1(X)}{Q(X)}$.

Il est donc possible de se passer, au maximum, de tous les termes de $R(X)$ de degré inférieur à k pour calculer $\frac{k(X)}{E(X)}$. Ceci est intéressant car, au lieu de calculer le polynôme $R(X)$ tout entier à partir des polynômes de Lagrange L_i, le calcul de $X^k R_1(X)$ peut se faire à partir de versions tronquées de ces polynômes de Lagrange, constitués eux-même des seuls termes de degré k ou supérieur des polynômes de Lagrange, ce qui engendre beaucoup moins de calculs de coefficients et donc un gain de temps.
Cependant, cette méthode ne permet plus de calculer le polynôme $f(X)$ associé au mode de code émis. On connait seulement le polynôme localisateur d'erreurs dont on déduit la position des symboles erronés. On peut alors considérer que ces symboles erronés sont effacés, et alors faire de la correction d'effacements, via la résolution d'un système linéaire. On va établir ce système linéaire en utilisant la matrice de parité du code.

Matrice de parité d'un code

Soit

$$H = \begin{bmatrix} \vdots & \vdots & & \vdots \\ h_1 & h_2 & \dots & h_n \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

avec h_i représentant la colonne i. Si :

$$C = \left\{ (c_1, c_2, \ldots, c_n) \in \mathbb{F}_q^n \mid c_1 h_1 + c_2 h_2 + \ldots + c_n h_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \right\} \tag{8}$$

alors H est appelé la matrice de parité du code C.
Elle est de dimension $(n-k) \times n$.

On rappelle que $c = (c_1, \ldots, c_n)$ est le mot de code émis et $r = (r_1, \ldots, r_n)$ est le mot de code reçu. Le mot de code r devrait donc vérifier :

$$\sum_{i=1}^{n} r_i h_i = 0$$

Soit $\mathcal{E}$ l'ensemble des indices des symboles erronés, on a $\forall i \notin \mathcal{E}, r_i = c_i$ et, pour $i \in \mathcal{E}$, notons x_i les symboles non déterminés se trouvant aux positions des erreurs. On a :

$$\sum_{i \notin \mathcal{E}} c_i h_i + \sum_{j \in \mathcal{E}} x_j h_j = 0 \tag{9}$$

On résout ce système linéaire pour retrouver les symboles effacés, et ainsi on a corrigé le vecteur reçu.

2 Implémentation

Le choix du language dans lequel nous avons écrit le programme pour réaliser le décodage des codes de Reed-Solomon fut essentiellement déterminé par le fait que le décodage de ces codes fait intervenir des structures mathématiques évoluées (en particulier les corps finis), nous nous sommmes alors dirigés vers un langage orienté mathématiques et nous avons choisi le langage de script **GP** du système **PARI/GP**. Les fonctions écrites dans ce langage sont à utiliser en ligne de commande depuis l'interpréteur `gp`.

Dans la conception de notre programme nous nous sommes tout d'abord basés sur la méthode de décodage par l'algorithme d'Euclide telle qu'exposée dans le cours d'arithmétique de notre master [1], puis nous avons modifié notre programme afin qu'il applique les améliorations exposées dans la deuxième méthode de décodage (section 1.3).

2.1 Structure générale du programmme

Il faut ici constater que pour effectuer le décodage d'un mot de code selon les méthodes abordées, ces dernières font appel à certains objets mathématiques qui ne sont pas spécifiques au mot de code reçu mais au code en lui-même, c'est notamment le cas de l'ensemble $\mathcal{P}$, des polynômes de Lagrange L_i ou encore de la matrice de parité H du code pour la deuxième méthode. Mettre en place tous ces objets prend du temps et n'a pas besoin d'être refait à chaque fois que l'on demande la correction d'un nouveau mot de code pour peu que le code reste le même. Notre programme est alors séparé en deux fonctions principales :

- Une première, dite de pré-calcul, qui prend en argument toutes les caractéristiques du code considéré et construit les différentes structures de données nécessaires au décodage.
- Une deuxième, la fonction de correction, qui prend en argument un mot de code à décoder et les structures de données construites par la fonction de pré-calcul et effectue le décodage du mot de code.

Notre programme devait par ailleurs être le plus général possible alors un autre point important pour l'architecture du programme fut le fait qu'un code de Reed-Solomon peut être défini sur un corps premier ou non. Or la manipulation d'éléments de corps premiers diffère sensiblement de celle d'éléments de corps non premiers. Un effort particulier fut apporté pour restreindre les situations où il était nécessaire de faire la distinction entre les deux cas de façon à ce que notre programme puisse bien les traiter indifféremment, sans quoi écrire 2 programmes distincts aurait été plus simple. Il reste cependant un certain nombre de passage du code qui sont écrits spécifiquement pour chacun des deux cas.

Enfin, de manière à pouvoir les comparer, notre programme implémente les deux méthodes de décodage décrites et permet de choisir l'une ou l'autre. Pour la première méthode, la correction du mot de code est considérée comme achevée lorsque le programme a trouvé le polynôme $f(X)$ associé au mot de code émis, c'est donc cette valeur qui est retourné au terme de l'exécution de la fonction de correction.
Dans le cas de la deuxième méthode, comme celle-ci ne calcule plus le polynôme $f(X)$, elle retourne alors le mot de code corrigé dans le même format que celui du mot de code à corriger qui a été fourni au programme.

2.2 Représentation des données

La représentation des éléments d'un corps fini $\mathbb{F}_q$ n'est pas toujours intuitive, voici donc la façon que nous avons choisi pour notre programme.
Deux cas sont à distinguer : les corps finis premiers et ceux qui ne le sont pas. Un corps fini est dit premier si son nombre d'éléments q est premier.

Si le corps $\mathbb{F}_q$ sur lequel est défini le code est premier, alors ce corps est isomorphe à $\mathbb{Z}/q\mathbb{Z}$ et chacun de ses éléments est désigné par un représentant (un entier) de la classe de congruence qui lui est associé dans $\mathbb{Z}/q\mathbb{Z}$. En particulier la famille des $(\alpha_i)_{1 \leqslant i \leqslant n}$ et les symboles constituant le mot de code à corriger devront être fournis au programme selon cette représentation.

Si le corps n'est pas premier, alors il existe un nombre premier p et un entier m tel que $q = p^m$ et dans ces conditions $\mathbb{F}_q$ est isomorphe au corps K défini par :

$$K = \frac{\mathbb{F}_p[X]}{P(X) \times \mathbb{F}_p[X]} \tag{10}$$

où P est un polynôme irréductible de degré m dans $\mathbb{F}_p[X]$.
De plus si P est primitif, c'est-à-dire que l'élément $g = X \mod P(X) \in K$ est un générateur de K^*, on a :

$$\forall x \in K \backslash \{0\}, \exists\, l \in [\![0, q-2]\!], x = g^l$$

Le corps $\mathbb{F}_q$ sera alors assimilé à un corps K où le polynôme P qui aura servi à le définir devra être primitif et ses éléments non nuls seront représentés par la valeur de la puissance à laquelle il faut élever le générateur g pour les obtenir. Cependant afin de représenter l'élément nul par 0, l'unité sera désignée par la valeur $q-1$ (ce qui reste cohérent puisqu'on a : $1 = g^0 = g^{q-1}$). Ici aussi cette représentation devra être utilisé pour les arguments du programme, et donc l'utilisateur devra, en plus, indiquer au programme quel est le polynôme primitif qu'il a utilisé pour représenter ses éléments (au moyen du vecteur de ses coefficients).

A l'intérieur du programme, on construit les structures de données correspondant exactement à ces représentations (éléments de $\mathbb{Z}/q\mathbb{Z}$ si q est premier, éléments de K dans le cas contraire) avec lesquelles **gp** sait calculer.

2.3 Fonction de pré-calcul

Avant de pouvoir réaliser la correction d'un mot de code, il est nécessaire de mettre en place un certain nombre de structures mathématiques coûteuses à calculer mais qui dépendent seulement du code (mais pas du mot de code à corriger) ; elles peuvent donc être établies séparément, avant tout autre chose.

C'est le rôle de la fonction de pré-calcul `construct_field` dont la signature est :
`construct_field(vector_alpha, p, dim_code, trunc, {poly_coef = 0})`. [2]
Elle prend en argument le vecteur `vector_alpha` des représentations des éléments de la famille $(\alpha_i)_{1 \leqslant i \leqslant n}$, la caractéristique `p` du corps sur lequel est défini le code, la dimension du code `dim_code`, une variable booléenne [3] `trunc` pour choisir la méthode de décodage qui sera utilisée (0 pour la première et 1 pour la deuxième) et un argument optionnel [4] `poly_coef` qui contient, le cas échéant, les coefficients du polynôme $P(X)$ utilisé pour définir le corps K.

Cette fonction procède alors à la création de la famille $(\alpha_i)_{1 \leqslant i \leqslant n}$ comme éléments de $\mathbb{F}_q$ en utilisant un format compréhensible par `gp`, puis calcule les polynômes de Lagrange L_i associés à ladite famille en faisant appel, pour chacun d'entre eux, à la fonction `Li` à qui cette tache est dévolue. Ce calcul se fait simplement en appliquant leur définition (1), cependant ce calcul peut nécessiter un temps assez long, mais, une fois encore, ce n'est pas bien grave puisque ceci se fait indépendamment de toute opération de décodage à proprement parler. Elle calcule également le polynôme $Q(X)$ (voir (3)).

Dans le cas où c'est la deuxième méthode de décodage qui est appliquée, cette fonction calcule en plus une matrice de parité du code à partir d'une matrice génératrice écrite dans une forme bien particulière, voilà comment elle procède :

Matrice génératrice

On appelle matrice génératrice d'un code C, de dimension k, une matrice de la forme suivante :

$$G = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix}$$

où $\{g_1, g_2, \ldots, g_k\}$ est une base de C.
Elle est donc de dimension $k \times n$.

[2] **GP** est un langage non typé.
[3] En fait un entier valant 0 ou 1.
[4] C'est, là, la signification des accolades, si cet argument n'est pas précisé lors de l'appel à la fonction alors il lui sera affecté la valeur qui est précisée dans la signature.

On dit que la matrice génératrice est sous forme systématique si :

$$G = \begin{bmatrix} I_k & A \end{bmatrix}$$

où I_k est la matrice identité de taille k.

Passage d'un matrice génératrice systématique à une matrice de parité

Si la matrice génératrice est sous forme systématique, une matrice de parité se déduit alors de cette matrice génératrice par :

$$H = \begin{bmatrix} -{}^tA & I_{n-k} \end{bmatrix}$$

Ceci se voit facilement en constatant que le produit scalaire de n'importe quelle ligne de G avec n'importe quelle ligne de H écrite sous cette forme est nul. Ainsi, la définition (8) est bien vérifiée.

Pour construire une matrice génératrice systématique du code qui va être traité, on utilise une autre famille de polynôme de Lagrange, ceux associés à la famille de points $(\alpha_i)_{1 \leqslant i \leqslant k}$. Chacun d'eux est de degré $k-1 < k$, donc leur évaluation en chacun des points de $(\alpha_i)_{1 \leqslant i \leqslant n}$ crée bien un mot du code, par ailleurs, il n'y a qu'un seul d'entre eux pour lequel l'évaluation en l'un des k premiers points de $(\alpha_i)_{1 \leqslant i \leqslant n}$ ne vaut pas 0, ce qui nous dit que la famille des mots de code ainsi créés est une famille libre, enfin, comme ils sont au nombre de k, cette famille de mots de code constitue une base du code. Alors la matrice dont chacune des lignes est un de ces mots de code est bien une matrice génératrice du code.

Dans le programme, chacun des éléments de cette deuxième famille de polynômes de Lagrange est calculé en même temps que ceux de la première dans la fonction `Li` pour les k premiers points de $(\alpha_i)_{1 \leqslant i \leqslant n}$.
Cette matrice de parité est utilisée pour créer le système linéaire qui permettra de retrouver les bons symboles aux positions des erreurs. S'il s'agit d'un code défini sur un corps non premier les solutions trouvées sont accessibles sous forme de polynômes (c'est de cette façon que **PARI/GP** représente les éléments des corps finis non premiers de la forme de K). Afin de retourner le mot de code corrigé sous la même forme que les données fournis au programme, il faut trouver à quelle puissance le générateur g de K^* doit être

élevé pour obtenir chacune des solutions.

Pour cela nous avons décidé d'utiliser une table de logarithme en base g des éléments de $\mathbb{F}_q$, table qui est alors aussi calculée par cette fonction de pré-calcul en faisant appel à la fonction `table_of_log`.
Celle-ci utilise alors une autre valeur numérique associée à chaque polynôme et facilement calculable à partir de ceux-ci : chaque polynôme représentant un élément de K est un polynôme de $\mathbb{F}_p[X]$ de degré strictement inférieur à m (Cf. (10)). Ainsi on peut leur associer une valeur comprise entre 0 et $q-1$ telle que la juxtaposition de leurs coefficients corresponde à l'écriture en base p de cette valeur (où chacun des coefficients, appartenant à $\mathbb{F}_p$, est associé à son représentant canonique compris entre 0 et $p-1$). La fonction `table_of_log` crée alors un tableau à $q-1$ éléments tel que son i-ème élément soit la puissance à laquelle il faut élever g pour que le polynôme obtenu soit le polynôme à qui est associé la valeur i (le polynôme nul est traité séparément).

2.4 Fonction de correction

C'est la fonction `correcting_RS` qui réalise la correction d'erreur au sein d'un mot de code reçu, sa signature est :
`correcting_RS(vector_received, K, disp)`.
Elle prend en argument le vecteur `vector_received` des éléments du corps fini constituant le mot de code à décoder, `K` doit être le vecteur des structures de données retourné par la fonction `construct_field` et `disp` est une variable booléenne permettant de choisir si l'on souhaite que les étapes intermédiaires du processus de correction soient affichées ou non. Voici les étapes suivant lesquelles se déroule la correction d'erreur dans cette fonction.

Tout d'abord la fonction transforme les éléments de `vector_received` de la même façon que pour les éléments de la famille $(\alpha_i)_{1\leqslant i\leqslant n}$, puis calcule le polynôme d'interpolation $R(X)$ (défini en (2), variable `pol_interpol` dans le code) en utilisant les polynômes de Lagrange établis par la fonction de pré-calcul (ou son approximation si la deuxième méthode de décodage est utilisé). S'ensuit un test sur le degré de ce polynôme d'interpolation.

Si son degré est inférieur strictement à la dimension du code alors cela signifie qu'il n'y avait pas d'erreur dans le mot de code fourni au programme, dans le cas de la première méthode de décodage ce polynôme est égal au polynôme $f(X)$ associé au mot de code. Le programme retourne alors ce polynome si la première méthode de décodage est appliquée ou l'argument `vector_received` si la deuxième méthode est utilisée et s'achève.

Sinon, le programme commence une boucle dans laquelle il calculera les réduites successives de la fraction rationnelle $R(X)/Q(X)$ ($Q(X)$ est appelé

`divisor` dans le code), enregistrées dans la variable `reduce`, et testera pour chaque réduite si

$$\texttt{pol\_interpol} - \texttt{reduce} \times \texttt{divisor} \tag{11}$$

est un polynôme de degré strictement inférieur à la dimension du code.

Le calcul des réduites se fait à partir des éléments de la famille (a_i) utilisés pour l'écriture du développement en fraction continue de $R(X)/Q(X)$ (voir (5)) puis en appliquant la relation de récurrence (6). Les (a_i) sont facilement obtenus car égaux aux quotients successifs de l'algorithme d'Euclide appliqué au numérateur et au dénominateur de la fraction, c'est à dire, ici, $R(X)$ et $Q(X)$.

Donc, lorsqu'on a trouvé une réduite telle que la quantité (11) est un polynôme de degré strictement inférieur à la dimension du code cela signifie qu'on a trouvé la réduite que l'on cherchait, son dénominateur est égal au polynôme localisateur d'erreur et, dans le cas de la première méthode de décodage, cette quantité est égale au polynôme $f(X)$ associé au mot de code émis (Cf. (7)). Si c'est la première méthode de décodage qui est appliqué, le programme retourne alors cette quantité et s'achève. Sinon, dans le cas de la deuxième méthode, il faut alors déterminer quels sont les emplacements des erreurs puis retrouver les symboles du mot de code émis qui se trouvaient à l'origine à ces emplacements. Pour cela on appelle la fonction `deletions_correction` dont les arguments sont notamment la matrice de parité et la table de logarithme (dans le cas d'un code défini sur un corps non premier) calculées par la fonction de pré-calcul et le polynôme localisateur d'erreur évidement.

La fonction `deletions_correction` commence par factoriser le polynome localisateur d'erreur (en utilisant une fonction de la bibliothèque du système **PARI/GP**), puis extrait de cette factorisation ses racines. Elle recherche ensuite à quel élément de la famille $(\alpha_i)_{1 \leqslant i \leqslant n}$ correspond chacune de ces racines afin d'en déduire la position des erreurs dans le mot de code reçu, ceci permet de déterminer l'ensemble $\mathcal{E}$ des indices des symboles erronés. Elle peut alors maintenant créer le système linéaire (9) qui permet de retrouver les symboles originaux du mot de code émis situés aux emplacements des erreurs, puis le résout là aussi en utilisant une fonction de la bibliothèque de **PARI/GP** qui met en œuvre une résolution par pivot de Gauss. Une fois les solutions trouvées, avant de les retourner, la fonction se charge de le mettre dans le même format que celui utilisé pour les éléments de l'argument `vector_received` de la fonction de correction. Si le code est défini sur un corps premier alors il suffit de prendre un représentant de la classe de congruence de $\mathbb{Z}/q\mathbb{Z}$ mais si le code est défini sur un corps non premier, alors ces

solutions sont sous forme polynômiale et il faut donc utiliser la table de logarithme calculée à cet effet, en appelant la fonction `log_Fq` à qui on fournit la table de logarithme et un vecteur contenant l'ensemble des solutions que l'on veut changer de forme. Cette fonction obtient le logarithme en base g pour chacun des éléments du vecteur qu'on lui donne via le procédé expliqué en page 14. Ceci fait, la fonction `deletions_correction` retourne ces solutions avec les emplacements qu'elles doivent occuper dans le mot de code corrigé. Alors la fonction de correction effectue le remplacement des valeurs erronées dans le mot de code reçu `vector_received` par les valeurs des corrections fraichement calculées, retourne ce mot de code corrigé et s'achève.

Il faut noter que, si après le calcul de $\left\lfloor \frac{n-k}{2} \right\rfloor$ réduites, aucune d'entre elles n'a permis que la quantité (11) soit un polynôme de degré strictement inférieur à la dimension du code, alors cela signifie que le mot de code fourni à la fonction de correction contient plus d'erreurs que ce que le code permettait de corriger. Un message est alors affiché pour en informer l'utilisateur, et la fonction s'achève.

2.5 Fonction d'encodage

Nous avons ajouté à notre programme une fonction d'encodage systématique c'est-à-dire une fonction qui, étant donné un code de Reed-Solomon défini sur $\mathbb{F}_q$ de dimension k, est capable de créer un mot de code à partir de k symboles d'information appartenant à $\mathbb{F}_q$ de façon à ce que ces k symboles d'informations soient les k premiers symboles du mot de code. Il s'agit de la fonction `enconding` dont la signature est :
`encoding(K, vect_info)`.
Elle prend en argument le vecteur `vect_info` contenant les k symboles d'informations à encoder et `K` doit être le vecteur des structures de données retourné par la fonction `construct_field` qui aura été appelée telle que si l'on s'apprêtait à décoder des mots de code selon la deuxième méthode de décodage. La fonction utilise alors la matrice génératrice systématique G utilisée pour définir la matrice de parité dans l'implémentation de la seconde méthode de décodage. Alors, si v est le vecteur des k symboles d'information, le mot de code s est obtenu par la très simple formule :

$$s = v \times G$$

Remarque.
Afin d'accélérer quelques calculs de puissance, notre programme utilise une

fonction `expbin` qui implémente une méthode d'exponentiation binaire.
Et il contient également une fonction `display` qui permet l'affichage de polynômes sans toutefois l'afficher dans son ensemble lorsque celui-ci contient de nombreux termes, seuls ceux de plus haut degré et de plus bas degré sont alors affichés.

3 Résultats

Pour illustrer les différences de temps de calcul entre les deux méthodes nous allons utiliser un code de Reed-Solomon de longueur 255, construit sur $\mathbb{F}_{2^8}$ et de dimension 239. C'est le code le plus utilisé en pratique car il manipule des octets. Le nombre maximal d'erreurs corrigibles est de $\frac{255-239}{2} = 8$. Nous allons donc utiliser un mot de code contenant 8 erreurs. Le tableau 3.0.1 montre les différents temps de calcul des étapes du programme :

	Méthode 1		Méthode 2	
	temps	% temps total	temps	% temps total
Etape 1 : calculs préliminaires	137 ms	$7,2\%$	101 ms	$11,5\%$
Etape 2 : calcul du polynôme d'interpolation	1177 ms	$61,5\%$	305 ms	35%
Etape 3 : calcul des 8 réduites successives	88 ms	$4,6\%$	32 ms	$3,6\%$
Etape 4 : Méthode 1 : somme du temps de calcul de f aux 8 étapes , Méthode 2 : tester si on a le bon nombre d'erreurs	425 ms	$22,2\%$	345 ms	$39,3\%$
Etape 5 : résolution du système (Méthode 2 seulement)			53 ms	6%
Temps total d'exécution du programme	1827 ms	100%	836 ms	100%

FIG. 3.0.1 – Temps de calcul des étapes des deux méthodes de décodage

Dans le tableau 3.0.1 sont regroupées les étapes de calcul de réduites et

de calcul de f, car dans le programme, on calcule une réduite, puis on calcule le candidat pour f, si son degré est inférieur à k alors c'est le bon nombre d'erreur (et même le bon polynome f pour la méthode 1). Sinon on calcule la réduite suivante et ainsi de suite. Dans cet exemple, on calcule donc successivement 8 réduites et 8 fois le candidat pour f, car il y a 8 erreurs dans le mot de code.

On constate que l'étape la plus coûteuse dans la première méthode est bien le calcul du polynôme d'interpolation, puisqu'il utilise 60% du temps de calcul. Ce temps est divisé par environ 4 dans la deuxime méthode, donc le gain de temps est considérable d'autant plus que la résolution du système linéaire n'utilise que 6% du temps total, donc le temps rajouté est négligeable devant le temps qu'on gagne. Néanmoins l'étape 4 qui est essentiellement le calcul de $R(X) - \dfrac{k(X)Q(X)}{E(X)}$ est assez coûteuse également puisqu'elle utilise environ 35% du temps de calcul total dans les deux méthodes.

4 Améliorations possibles

Bien évidement notre programme n'est pas parfait ni le plus optimal possible, voici donc quelques points qu'il serait possible d'améliorer. Tout d'abord il serait possible de réécrire notre programme dans un langage de plus bas niveau tel que le C, cela aurait pour avantage d'avoir un code source plus près du langage machine et donc de manipuler des structures de données moins évoluées et donc moins lourdes ce qui accélérerait l'exécution du programme. Cependant, cela demanderait un plus grand travail de programmation puisqu'il faudrait mettre en place ces structures de donnés permettant de représenter les objets manipulés et il faudrait aussi se poser la question de comment réaliser les opérations mathématiques sur ces objets (addition et multiplication de polynômes, division euclidienne, factorisation), ce qui peut éventuellement se trouver dans des bibliothèques déjà existantes.

On pourrait aussi, afin d'accélérer encore la vitesse d'exécution du programme, utiliser tout simplement des tables d'opérations pour les plus basiques et donc les plus courantes telles que l'addition et la multiplication dans le corps fini sur lequel est défini le code. Le programme serait alors forcément beaucoup plus rapide puisque pour réaliser ces opérations, il n'y a plus aucun calcul arithmétique, mais seulement la lecture d'une case mémoire. Cependant, là encore, cela demanderait un surcroît de travail de programmation très important puisqu'il faudrait une implémentation spécifique de toutes les

opérations d'objets définis à partir des corps finis.

Enfin, notre programme ne sait travailler actuellement que sur un seul mot de code à chaque appel de la fonction de correction, il serait possible de le rendre capable de traiter plusieurs mots de code à la fois ou même, via une interface dédiée, un flux de mots de code.

5 Conclusion

Notre programme réalise bien le décodage de codes de Reed-Solomon de toute longueur, définis sur des corps finis premiers ou non. La deuxième méthode de décodage propose une amélioration significative de la premièr, en effet elle divise le temps de décodage par 2. Cependant ces performances restent bien en deçà de ce qui est nécessaire pour des applications réelles telles que la lecture d'un CD ou le traitement d'un flux vidéo haute définition qui nécessitent des débits de plusieurs milliers, ou même millions, d'octets par seconde. On comprend alors aisément pourquoi en pratique le décodage de code de Reed-Solomon n'utilise pas d'implémentation logicielle mais est effectué matériellement via des puces dédiées.

Références

[1] Zemor G. *Master CSI, Arithmétique 1 : corps finis et applications* (http://www.math.u-bordeaux.fr/~zemor/arit06.pdf)

[2] Wicker SB., Bhargava VK. *An Introduction to Reed-Solomon Codes* (http://media.wiley.com/product_data/excerpt/19/07803539/0780353919-2.pdf)

[3] *PARI/GP* (http://pari.math.u-bordeaux.fr/)