Ce TP comporte deux parties, la première partie propose des exercices d'illustration des principes de base des codes correcteurs, la deuxième partie explique une implémentation des codes de Reed-Solomon (utilisés notamment dans les CD).

Vous allez effectuer les exercices de ce TP dans une session interactive de Xcas, vous pourriez aussi les programmer en C++ en utilisant la librairie giac ou directement en C++ (mais dans ce dernier cas, certaines instructions comme la réduction sous forme échelonnée ou les opérations polynomiales doivent alors être programmées).

1 Xcas

Pour faire ce TP, vous devrez installer Xcas, qui est un logiciel de calcul formel capable entre autres de calculer dans les corps finis : cherchez sur google, puis installez le package debian. Au premier lancement, choisissez Xcas comme syntaxe. Pour obtenir de l'aide dans Xcas, utilisez le menu Aide ou tapez le début d'un nom de commande puis la touche de tabulation.

En Xcas, un élément a de $\mathbb{Z}/n\mathbb{Z}$ s'écrit `a % n`, un corps fini se définit par exemple pour $G = F_{256}$ par l'instruction `G:=GF(2,8,['a','G'])`, `A:=G(a)` désigne alors un élément primitif de G, G est donc égal à $\{0, A^0 = 1, A, A^2, ..., A^{254}\}$ et les éléments de G seront affichés par Xcas comme `G` de puissances de a.

Vous pouvez effectuer les opérations arithmétiques usuelles avec des éléments de corps finis (`+,-,*,/,%` et menus `Maths->Entier` et `Math->Polynomes`). Pour les polynômes à coefficients dans un corps fini, on utilise la représentation par la liste de ses coefficients en ordre décroissant, par exemple `poly1[A,A^3,0]` désigne un polynôme de degré 2 à coefficients dans le corps fini G dont le coefficient de x^2 est a, celui de x est a^3 et le coefficient constant est nul.

Un programme Xcas se définit par

```
f(arg1,...,argn):={
  local var1,...,varn;
  ...
}
```

Pour créer un programme, on utilise le menu Edit, ajouter programmme, puis les menus du niveau de programme créé pour ajouter des structures de controle (dont la syntaxe est identique à celle du C). Les principales différences avec le C sont l'instruction d'affectation (`:=` au lieu de `=`) et le fait que le langage est non typé (en-dehors de la distinction variable globale/locale). Utilisez `OK` pour interpréter le programme, vérifiez les messages d'erreur.

Pour mettre au point un programme en l'exécutant pas à pas, tapez `debug(nom_du_programme(arg1,...))`. Vous pouvez aussi utiliser l'instruction `print` pour afficher des informations intermédiaires.

2 TP

On appellera symbole d'information l'unité de base transmise, qu'on supposera appartenir à un corps fini K, par exemple pour un bit un élément de $K = \mathbb{Z}/2\mathbb{Z}$, ou pour un octet un élément du corps à 256 éléments $K = F_{256}$.

On veut coder un message de longueur k avec des possibilités de détection et de correction d'erreurs, pour cela on rajoute des symboles calculés à partir des précédents, on envoie un élément d'un code ayant n symboles.

2.1 Le bit de parité.

On prend $k = 7$ bits et $n = 8$ bits. On compte le nombre de 1 parmi les 7 bits envoyés, si ce nombre est pair, on envoie 0 comme 8ième bit, sinon 1. Au final le nombre de bits à 1 de l'octet (1 octet=8 bits) est pair. On peut ainsi détecter une erreur de transmission si à la réception le nombre de bits d'un octet est impair, mais on ne peut pas corriger d'erreurs. On peut aussi dire que l'octet représente un polynôme à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ divisible par $X + 1$.

Exercice : Écrire un programme Xcas permettant de rajouter un bit de parité à une liste composée de 7 bits. Puis un programme de vérification qui accepte ou non un octet selon sa parité. Vous représenterez l'octet par une liste de bits, avec le délimiteur `poly1[` pour pouvoir effectuer des opérations arithmétiques polynomiales, et vous effectuerez la vérification de deux manières, en comptant le nombre de 1 ou avec l'instruction `rem`.

2.2 Codes linéaires

Définition

On multiplie le vecteur des k symboles par une matrice M à coefficients dans K de taille $n \times k$ et on transmet l'image. Pour assurer qu'on peut identifier un antécédent unique à partir d'une image, il faut que M corresponde à une application linéaire

injective, ce qui entraine $n \geq k$. On dit qu'un vecteur de n symboles est un mot du code s'il est dans l'image de l'application linéaire.

Pour assurer l'injectivité tout en facilitant le décodage, on utilise souvent une matrice identité k, k comme sous-bloc de la matrice n, k, par exemple on prend l'identité pour les k premières lignes de M, on ajoute ensuite $n - k$ lignes.

Pour savoir si un vecteur est un mot de code, il faut vérifier qu'il est dans l'image de M. On peut par exemple vérifier qu'en ajoutant la colonne de ses coordonnées à M, on ne change pas le rang de M (qui doit être k).

Exercice : créez une matrice M de taille 7,4 injective. Puis un programme qui teste si un vecteur est un mot de code et en extrait alors la partie avant codage. Vérifiez votre programme avec un vecteur Mv, on doit obtenir un mot de code.
Instructions utiles : `idn` (matrice identité) `ker` (noyau d'une application linéaire), `rank` (rang), `tran` (tranposée), ... Pour créer une matrice, on peut coller les lignes de 2 matrices A et B par `[op(A),op(B)]` ou avec `blockmatrix`.

2.3 Codes polynomiaux

Définition
Il s'agit d'un cas particulier de codes linéaires. On se donne un polynôme $g(x)$ de degré $n - k$, On représente le message de longueur k à coder par un polynôme P de degré $k - 1$. On multiplie P par x^{n-k}, on calcule le reste R de la division de Px^{n-k} par g. On émet alors $Px^{n-k} - R$ qui est divisible par g. Les mots de code sont les polynômes divisibles par g.

Exercice : écrire de cette façon le codage du bit de parité. Puis une procédure Xcas de codage utilisant $g = X^7 + X^3 + 1$ (ce polynôme était utilisé par le Minitel). N.B. on obtient le polynôme X^{n-k} sous forme de polynome-liste dans Xcas par `poly1[1,0$(n-k)]`.

2.4 Détection et correction d'erreur

Si le mot recu n'est pas dans l'image de l'application linéaire il y a eu erreur de transmission. Sinon, il n'y a pas eu d'erreur *détectable* (il pourrait y avoir eu plusieurs erreurs qui se "compensent").

Plutôt que de demander la réémission du mot mal transmis (ce qui serait par exemple impossible en temps réel depuis un robot en orbite autour de Mars), on essaie d'ajouter suffisamment d'information pour pouvoir corriger des erreurs en supposant que leur nombre est majoré par N. (Si les erreurs de transmissions sont indépendantes, la probabilité d'avoir plus de N erreurs est ϵ^{N+1}, où ϵ est la probabilité d'une erreur de transmission. Par exemple si $\epsilon = 10^{-2}$ et qu'il s'agit de transmission d'images, on peut prendre $N = 2$ ou 3 sans grand risque).

Exemple : On ne peut pas corriger d'erreur avec le bit de parité.

2.5 Distances

La distance de Hamming de 2 mots est le nombre de symboles qui diffèrent. (il s'agit bien d'une distance au sens mathématique, elle vérifie l'inégalité triangulaire).

Exercice : écrire une procédure de calcul de la distance de Hamming de 2 mots.

La distance d'un code est la distance de Hamming minimale de 2 mots différents du code. Pour un code linéaire, la distance est aussi le nombre minimal de coefficients non nuls d'un vecteur non nul de l'image. Pour un code polynomial, la distance du code est le nombre minimal de coefficients non nuls d'un multiple de g de degré inférieur à n.

Exercice : quelle est la distance du code linéaire que vous avez créé plus haut ?

Majoration de la distance du code :
La distance minimale d'un code linéaire est inférieure ou égale à $n - k + 1$: en effet on écrit en ligne les coordonnées des images de la base canonique et on réduit par le pivot de Gauss, comme l'application linéaire est injective, le rang de la matrice est k, donc la réduction de Gauss crée $k - 1$ zéros dans chaque ligne, donc le nombre de coefficients non nuls de ces k lignes (qui sont toujours des mots de code) est au plus de $n - k + 1$.

Exercice : si votre code linéaire n'est pas de distance 3, modifiez les 3 dernières lignes pour réaliser un code de distance 3. On ne peut pas obtenir une distance $n - k + 1 = 4$ avec $n = 7$ et $k = 4$ dans $\mathbb{Z}/2\mathbb{Z}$, essayez ! Essayez sur $\mathbb{Z}/3\mathbb{Z}$ et $\mathbb{Z}/5\mathbb{Z}$.

N.B. : Pour les codes non polynomiaux, par exemple convolutifs, la distance n'est pas forcément le paramètre le mieux adapté à la correction d'erreurs.

2.6 Correction au mot le plus proche

Une stratégie de correction basée sur la distance consiste à trouver le mot de code le plus proche d'un mot donné. Si la distance d'un code est supérieure ou égale à $2t + 1$, et s'il existe un mot de code de distance inférieure à t au mot donné, alors ce mot de code est unique. On corrige alors le mot transmis en le remplaçant par le mot de code le plus proche.

Exercice : écrivez un programme permettant de corriger une erreur dans un mot dans votre code linéaire.

On dit qu'un code t-correcteur est parfait si la réunion des boules de centre un mot de code et de rayon t (pour la distance de Hamming) est disjointe et recouvre l'ensemble des mots (K^n).

Exercice : votre code linéaire sur $\mathbb{Z}/2\mathbb{Z}$ (celui de distance 3) est-il un code 1-correcteur parfait ?

3 Les codes de Reed-Solomon

Il s'agit de codes polynomiaux qui réalisent la distance maximale possible $n-k+1$. De plus la recherche du mot de code le plus proche peut se faire par un algorithme de Bézout avec arrêt prématuré.

3.1 Théorie

On se donne un générateur a de F_q^* et le polynôme $g(x) = (x-a)...(x-a^{2t})$ (donc $n-k = 2t$). Typiquement $q = 2^m$ avec $m = 8$, a est une racine d'un polynôme irréductible de degré m à coefficients dans $\mathbb{Z}/2$ qui ne divise pas $x^l - 1$ pour l diviseur strict de $2^m - 1$, en pratique, on factorise le quotient de $x^{2^m-1} - 1$ par le ppcm des $x^{(2^m-1)/p} - 1$ où p parcourt les diviseurs premiers de $2^m - 1$ et on en extrait un facteur de degré m.

Distance du code

Si la longueur n d'un mot vérifie $n \leq 2^m - 1$, alors la distance entre 2 mots du code est au moins de $2t+1$. En effet, si un polynome P de degré $< n$ est un multiple de g ayant moins de $2t+1$ coefficients non nuls,

$$P(x) = \sum_{k=1}^{2t} p_{i_k} x^{i_k}, \quad i_k < n$$

en écrivant $P(a) = ... = P(a^{2t}) = 0$, on obtient un déterminant de Van der Monde, on prouve qu'il est non nul en utilisant la condition $i_k < n$ et le fait que la première puissance de a telle que $a^x = 1$ est $x = 2^m - 1$.

Correction des erreurs

Soit $c(x)$ le polynome envoyé, $d(x)$ le polynôme recu, on suppose qu'il y a moins de t erreurs

$$d(x) - c(x) = e(x) = \sum_{k=1}^{\nu} \alpha_k x^{i_k}, \quad \nu \leq t$$

On calcule le polynome syndrome :

$$s(x) = \sum_{i=0}^{2t-1} d(a^{i+1}) x^i = \sum_{i=0}^{2t-1} e(a^{i+1}) x^i$$

on a donc :

$$\begin{aligned} s(x) &= \sum_{i=0}^{2t-1} \sum_{k=1}^{\nu} \alpha_k (a^{i+1})^{i_k} x^i \\ &= \sum_{k=1}^{\nu} \alpha_k \sum_{i=0}^{2t-1} (a^{i+1})^{i_k} x^i \\ &= \sum_{k=1}^{\nu} \alpha_k a^{i_k} \frac{(a^{i_k} x)^{2t} - 1}{a^{i_k} x - 1} \end{aligned}$$

On pose $l(x)$ le produit des dénominateurs (que l'on appelle polynôme localisateur, car ses racines permettent de trouver la position des symboles à corriger), on a alors

$$l(x)s(x) = \sum_{k=1}^{\nu} \alpha_k a^{i_k} ((a^{i_k} x)^{2t} - 1) \prod_{j \neq k, j \in [1,\nu]} (a^{i_j} x - 1) \tag{1}$$

Modulo x^{2t}, $l(x)s(x)$ est donc un polynôme w de degré inférieur ou égal à $\nu - 1$, donc strictement inférieur à t. Pour le calculer, on applique l'algorithme de Bézout à $s(x)$ et x^{2t} (dans F_q), en s'arrêtant au premier reste $w(x)$ dont le degré est strictement inférieur à t (au lieu d'aller jusqu'au calcul du PGCD de $s(x)$ et x^{2t}). Les relations sur les degrés (cf. approximants

de Padé) donnent alors en coefficient de $s(x)$ le polynôme $l(x)$ de degré inférieur ou égal à t. On en calcule les racines (en testant tous les éléments du corps avec Horner), donc la place des symboles erronés.

Pour calculer les valeurs α_k, on reprend la définition de w, c'est le terme de droite de l'équation (1) modulo x^{2t}, donc :

$$w(x) = \sum_{k=1}^{\nu} \alpha_k a^{i_k} (-1) \prod_{j \neq k, j \in [1,\nu]} (a^{i_j} x - 1)$$

Donc :

$$w(a^{-i_k}) = -\alpha_k a^{i_k} \prod_{j \neq k, j \in [1,\nu]} (a^{i_j} a^{-i_k} - 1)$$

Comme :

$$l(x) = (a^{i_k} x - 1) \prod_{j \neq k, j \in [1,\nu]} (a^{i_j} x - 1)$$

on a :

$$l'(a^{-i_k}) = a^{i_k} \prod_{j \neq k, j \in [1,\nu]} (a^{i_j} a^{-i_k} - 1)$$

Finalement :

$$\alpha_k = -\frac{w(a^{-i_k})}{l'(a^{-i_k})}$$

3.2 Avec Xcas

Ouvrez dans le menu Exemples, poly la session |reed_sol.xws|.

Le niveau 2 définit F_{256}, le niveau 3 un élément primitif. Pour calculer le polynôme émis correspondant à une suite d'octets, on crée la liste des éléments de F_{256} correspondant au moyen de l'application

```
f(y):=ifte (y==0,zero,a^y)
```

et de `map`, par exemple

```
l:=map([1,5,2,0],f)
```

Le polynome g peut se calculer avec `product`

```
g:=product(poly1[1,-a^k],k=1..2*t)
```

On rajoute $2t$ `G(0)` à la liste

```
l1:=poly1[op(l),seq(zero,2*t)]
```

ce qui revient à multiplier par x^{2t}, puis on calcule le reste de la division par g,

```
r1:=rem(l1,g)
```

que l'on retire (ou ajoute, c'est pareil) à `l1` pour transmission.

Ajoutons 2 erreurs pour tester la correction d'erreurs

```
r2:=r1+poly1[a,zero,zero,a^5,zero];
```

On calcule le polynome syndrome directement en représentation liste

```
s:=poly1[seq(horner(r2,a^(2*t-k)),k=0..(2*t-1))]
```

On effectue maintenant l'algorithme de Bézout sur x^{2t} et s en calculant uniquement les coefficients de s et en s'arretant au premier reste $w(x)$ dont le degré est strictement inférieur à t.

```
gf_bez(s,t):={ // s polynome liste, t entier
  local R0,R1,R2,v0,v1,v2; // R0=x^2t, R1=s, on calcule les v
  R0:=poly1[G(1),seq(zero,2*t)]; // x^2t
  R1:=s;
  v0:=poly1[];
  v1:=poly1[G(1)];
  while (degree(R1)>=t){
    q:=quo(R0,R1);
    R2:=R0-q*R1;
    v2:=v0-q*v1;
    R0:=R1;
    R1:=R2;
    v0:=v1;
    v1:=v2;
  }
  return v1,R1; // on renvoie le polynome localisateur et w
}
```

puis

```
(loc,w):=gf_bez(s,t)
```

donne le polynome localisateur et w.

Avec nos erreurs en positions $i_k = 1$ et $i_k = 4$, on vérifie bien que

`horner(loc,1/a)` et `horner(loc,1/a^4)`

sont nuls. Pour trouver la position des erreurs en général, on teste tous les a^{-k}

```
test_zero(loc,inva):={
  local pos,k;
  pos:=NULL;
  for (k:=0;k<255;k++){
    if (1/horner(loc,inva^k)==infinity) // astuce pour giac-0.5.0
      pos:=pos,k;
  }
  return pos;
}
```

Pour corriger les erreurs, il nous reste à calculer $\frac{w(a^{-i_k})}{l'(a^{-i_k})}$. On écrit un petit programme de dérivée de polynome-liste

```
diff_poly(l):={ // derivee d'un polynome liste
  local s,j;
  s:=degree(l);
  res:=poly1[0$s];
  for (j:=0;j<s;j++){
    res[j]:=(s-j)*l[j];
  }
  return res;
}
```

on l'applique

```
lprime=diff_poly(loc)
```

puis on évalue

```
k:=pos[0];horner(w,inva^k)/horner(lprime,inva^k)
```

et de même pour `k:=pos[1]`. On retrouve bien les erreurs introduites !

3.3 En C++

Vous pouvez utiliser un autre langage, mais en C++ vous pouvez utiliser la classe de polynomes
`http://www-fourier.ujf-grenoble.fr/~parisse/crypto/poly.tgz`
Définissez une classe avec un membre de donnée de type `int` représentant un élément de `GF(2,8)` et implémentez les opérations `+, -, *, /` (vous pouvez réutiliser le TP AES, mais attention si votre polynome n'est pas primitif au choix de a pour qu'il soit générateur), vous pouvez alors utiliser les opérations sur les polynômes de la classe ci-dessus. Il vous est maintenant possible de faire un programme C++ de codage et de correction d'erreurs en suivant le modèle de la section précédente.