LEBOCQ Titouan
DELAMARE Clément
GODET Arnaud
GUESPIN Antoine

Modélisation Mathématique : Les codes correcteurs d'erreurs

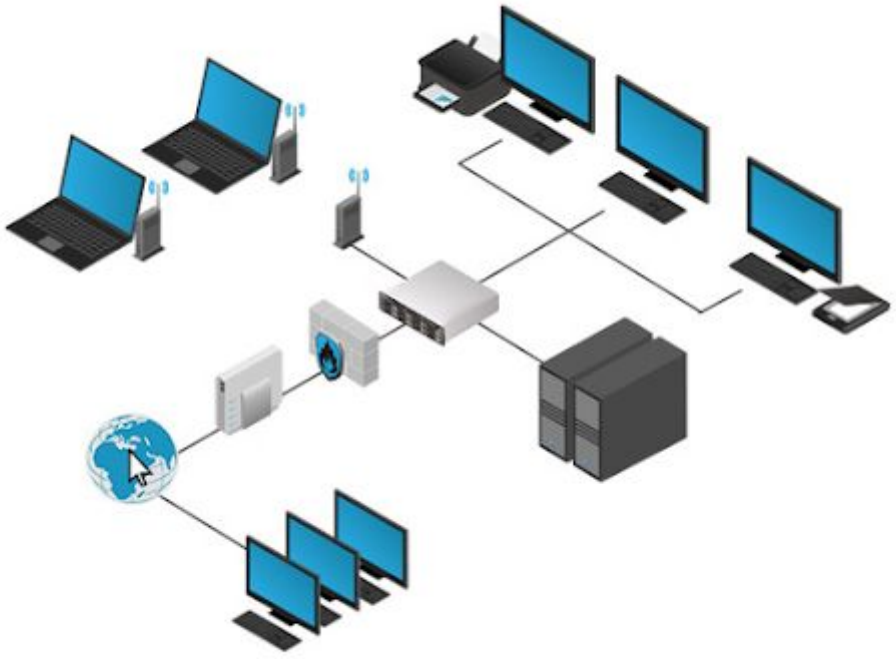
Introduction

Les erreurs un problème récurrent en informatique

En informatique, il est nécessaire de **stocker et d'échanger des informations** afin de pouvoir créer les logiciels et outils que nous utilisons tous les jours.

Les informations vont se trouver être **échangées** (réseau internet par exemple) et stockées (disque CD, serveur...). Seulement ces données peuvent se trouver être **altérées** suite à par exemple **un problème de transmission**. Pour **répondre à ce problème** récurrent en informatique de nombreux stratagèmes ont été mis en place notamment celui que nous avons étudié : les **code correcteurs d'erreurs**.

Ainsi un **code correcteur d'erreur** est une technique **d'encodage** de l'information visant à **détecter voir préserver** l'information dans un système informatique que ce soit lors de l'échange ou du stockage de l'information. Pour cela on viendra utiliser de la **redondance**.



Nous pouvons donc nous poser la question :

- **Est-il possible de stocker ou envoyer des informations sans les détériorer, de manière fiable et efficace ?**

Le principe d’un code correcteur d’erreurs

Il est fondamental de comprendre les principaux termes utilisés lorsque l’on parle de code correcteur d’erreurs.

Un code correcteur d’erreurs s’applique à un envoie de données appelé message. Ce message sera perçu comme l’addition de plusieurs blocs d’informations. On peut également appeler ça des mots d’informations ou vecteurs d’informations. Chaque bloc d’informations est composé d’un ou plusieurs bits, chacun prenant une valeur correspondant à un alphabet. Par exemple, si l’on prend un alphabet binaire, nous avons un bloc composé de bit ne pouvant prendre que les valeurs 0 et 1. Bien sûr, il existe une infinité d'alphabets qui correspondent à la manière de coder le message.

En clair, un message est composé de plusieurs suite de valeurs numériques. Le fait de le définir comme tel nous permet d’effectuer différentes opérations et d’appliquer des algorithmes identiques pour chaque suite de valeurs qui composent le message. Cela peut consister en ajoutant ce que l’on appelle, des bits de contrôle, à la fin de chaque bloc ou bien en modifiant complètement les blocs en évitant que deux blocs différents soient transformés en un même bloc déjà existant.

Tout ceci nous permet de vérifier l’intégrité des blocs composant le message et parfois de corriger les éventuelles erreurs de transmission. C’est ce que l’on appelle coder le message.

Le but d’un code correcteur d'erreurs est donc de vérifier que le message qui a été envoyé a bien été reçu sans erreurs.

Il existe plusieurs méthodes de codages connus qui ont chacune leur spécificité et qui permettent de remplir un rôle bien précis. Nous allons nous attarder sur trois d’entre elles en partant du plus simple pour aller au plus compliqué.

“Il y a le bon code correcteur d’erreur et le mauvais code correcteur d’erreur…” Arnaud Godet 2020

Même si le but reste de vérifier que nos données ont été transmis sans erreurs, on attend plusieurs choses d’un bon code correcteur d’erreur :

1. L’information ne doit pas être trop diluée (avoir le moins de redondance possible). Dans l’idéal, il doit être parfait.

2. On doit pouvoir détecter et corriger un nombre raisonnable d’erreurs.

3. L’algorithme de codage doit être suffisamment rapide.

4. L’algorithme de décodage, corrections incluses, doit être suffisamment rapide.

Pour plus de détails sur les codes correcteurs d’erreurs en général veuillez consulter l’annexe 1 : Theorie_information.ipynb.

Les codes linéaires

Code de répétitions

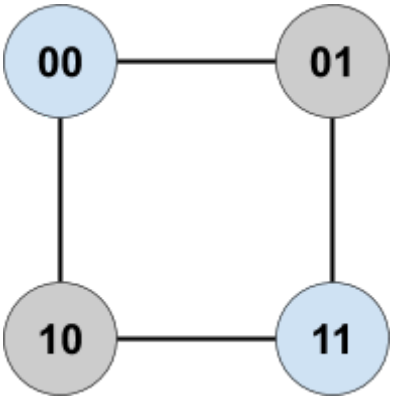
La première chose à laquelle nous pourrions penser afin de vérifier si un message est correct à sa réception serait de créer une répétition de ce message, c'est-à-dire ajouter une redondance lors de l'envoi de ce message. Prenons cette exemple :

Nous voulons envoyer le message suivant : “0” . Maintenant nous allons **coder** le message, c’est à dire nous allons ajouter la redondance d’après la table suivante :

Message	Message codée
0	00
1	11

Ce qui donne le message suivant : 00 pour le message ”0” et avec le message “1” nous obtiendrons le message : 11

Une fois le message encodé nous pouvons l’envoyer (ou le stocker). Une fois envoyées le récepteur à la possibilité de recevoir les messages suivant :



Sur cette représentation 2D, nous considérons que chaque bit est une dimension. Nous avons deux types de message reçu, les messages dits “**correct**” et les messages dits “**incorrects**” . En bleu nous avons les messages correct, et en gris les messages incorrects. Si nous recevons un message considéré comme incorrect, on peut en déduire qu’une erreur s'est glissée dans le message car lors du codage du message par l'émetteur il est impossible d’obtenir ces valeurs. Grâce à ce code nous pouvons donc détecter une erreur, mais en aucun cas la corriger.

En effet, il est impossible de corriger un message car sur la représentation 2D nous voyons que la distance entre deux messages corrects (nombre d'arête) est de 2. On appelle cette distance la **distance de Hamming**. Ce qui nous intéresse c’est de savoir la distance minimum, que l’on notera *δ.* Elle permet de savoir le nombre d'erreurs détectables dans un code ainsi que le nombre d'erreurs que l’on peut corriger. Pour obtenir ces valeurs il faut appliquer les formules suivantes :

Nombre d’erreur détectable	**Nombre d’erreur corrigible**
$e = \lfloor \frac{\delta}{2} \rfloor$	$t = \lfloor \frac{\delta - 1}{2} \rfloor$

Alors la question que l'on pourrait se poser est : comment pourrait-on faire pour pouvoir corriger une erreur ? Avec ce code c'est impossible car la δ est de 2, pour qu'un code linéaire permettre de corriger un erreur il faut que δ soit égal ou supérieur à 3. Pour vérifier nous pouvons appliquer les formules que nous avons vu ci-dessus.

Nombre d'erreur détectable

$$e = \lfloor \frac{\delta}{2} \rfloor$$
$$e = \lfloor \frac{3}{2} \rfloor$$
$$e = 1$$

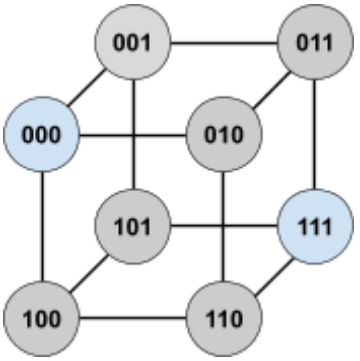
Nombre d'erreur corrigible

$$t = \lfloor \frac{\delta - 1}{2} \rfloor$$
$$t = \lfloor \frac{3 - 1}{2} \rfloor$$
$$t = \lfloor \frac{2}{2} \rfloor$$
$$t = 1$$

Pour augmenter δ il faut juste rajouter une redondance supplémentaire en plus, ce qui nous donne la table de conversion suivante :

Message	Message codée
0	000
1	111

Ce qui donne le message suivant : 000 pour le message "0" et avec le message "1" nous obtiendrons le message : 111 et si on représente les messages que l'on peut recevoir en 3D :



Corriger un message est maintenant possible, par exemple si nous recevons le message "011", il suffit de prendre le distance de Hamming le plus courte entre lui et un message correct. Ce qui nous donne 111, donc en reprenant notre table de conversion on peut en déduire que le message envoyé était "1". Et voilà, il est possible de corriger une erreur simplement !

Ce code que l'on appelle *code de répétition* est le code linéaire le plus simple, on le note [3,1,3]. Cette notation est utilisée pour connaître les performances d'un code linéaire. En effet les codes linéaires sont décrits par 3 parametres [n,k,δ] :

- n correspond à la taille du message, soit le message originel et la redondance. On appelle cette grandeur la **longueur** du code.
- k exprime la longueur du message une fois décodé.
- δ correspond à la distance minimum du code.

Par exemple, pour le code de répétitions que nous avons en premier, avait pour paramètres [2,1,2].

Le code de répétitions [3,1,3] est aussi un **code parfait**. Un code est dit parfait lorsqu'il permet de corriger et détecter une erreur tout en ne contenant aucune redondance superflue. Pour vérifier si un code est parfait il suffit de vérifier cette égalité :

$$n + 1 = 2^{n-k}$$

Donc avec les paramètres [3,1,3] :

$$3 + 1 = 2^{3-1}$$

$$4 = 4$$

L'égalité étant bien vérifiée, le code de répétitions [3,1,3] est un code parfait. Mais le code de répétitions [3,1,3] est le seul code de répétitions qui est un code parfait. En effet, si nous voulons par exemple augmenter la taille du message, il faudra aussi augmenter la redondance. Et alors le rendement deviendra très mauvais par exemple en ayant k = 2 cela nous donnera la table de conversion suivante :

Message	Message codée
00	000000
01	000111
10	111000
11	111111

Ce code de répétitions à donc pour paramètre [6,2,3]. Nous pouvons donc calculer le rendement de ce code grâce à cette formule :

$$\eta = \frac{k}{n}$$

Ce qui donne :

$$\eta = \frac{2}{6}$$

$$\eta = 33\%$$

Le rendement de ce code est de seulement 33%. De plus, ce code n'est pas parfait. En effet l'égalité permettant de voir si un code parfait n'est pas réalisable. Heureusement il existe un autre code avec un meilleurs rendement et qui lui est parfait, ce code est le code de Hamming.

Approche linéaire du code de répétitions

Avant de voir le code de Hamming, je propose de voir une approche linéaire de ce que nous avons vu précédemment. Tout d'abord pour coder un message nous allons utiliser une **matrice génératrice**. Pour commencer nous allons modéliser notre message sous forme de vecteur, par exemple pour le message "0" nous aurons ce vecteur : $\vec{v} = [0]$.

En multipliant ce vecteur par la matrice génératrice nous allons obtenir comme résultat le message encodé. Il sera donc prêt à être envoyé !

Dans le cas d'un code de répétitions, la matrice de génération que l'on appellera G est très simple, puis ce que le but est juste d'obtenir une répétition du message d'entrée. Par exemple pour le code de répétitions [3,1,3], notre matrice de génération sera : $G = [1\ 1\ 1]$.

Alors si nous faisons le produit de notre vecteur et de notre matrice de génération, nous devons obtenir le message encodé. Nous aurons donc pour notre vecteur $\vec{v}$:

$$\vec{u} = \vec{v}G$$

$$\vec{u} = [0][1\ 1\ 1]$$

$$\vec{u} = [0\ 0\ 0]$$

Le vecteur correspond bien aux valeurs que nous avons dans la table de conversion, nous pouvons aussi essayer avec $\vec{v} = [1]$:

$$\vec{u} = \vec{v}G$$

$$\vec{u} = [1][1\ 1\ 1]$$

$$\vec{u} = [1\ 1\ 1]$$

Maintenant, si nous voulons vérifier qu'un message reçu ne contient pas d'erreur, il existe une matrice de contrôle que nous appellerons H. Avant de voir cette matrice il faut d'abord comprendre comment nous pouvons vérifier un message pour un code de répétitions. Pour cela nous allons utiliser une opération xor. L'idée est de vérifier les égalités suivantes pour le vecteur $\vec{u}$ = [x y z] :

$$x \oplus y = 0$$

$$x \oplus z = 0$$

Si les deux égalité sont vérifiées alors le message est correct. Il est possible de vérifier ces deux égalités avec la matrice H. Dans le cas de notre code de répétitions [3,1,3] nous aurons :

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Pour vérifier si le message ne contient pas d'erreur, il suffit simplement de vérifier que le produit du vecteur transposé et la matrice de contrôle H modulo 2 soient égale au vecteur $\vec{w} = [0\ 0]$.

Dans cette, exemple on considère que le message envoyé est 1 :

$$\vec{0} = \vec{v}^T H$$

$$\vec{u} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\vec{0} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \xrightarrow{\text{On vient applique un modulo 2}} \vec{0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

L'égalité étant vérifiée, le message ne contient pas d'erreur mais si l'on vient ajouter une erreur au message :

$$\vec{0} = \vec{v}^T H$$

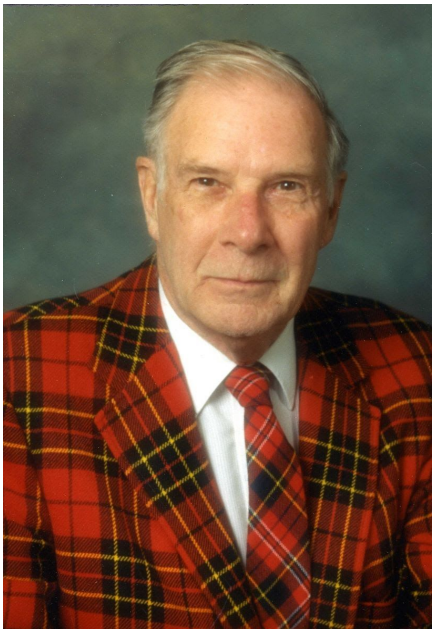
$$\vec{u} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\vec{0} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \xrightarrow{\text{On vient appliquer un modulo 2}} \vec{0} \neq \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

L'égalité n'étant pas vérifiée, il y a une erreur, mais alors comment trouver où se situe l'erreur ? Et bien c'est simple, ici l'erreur se trouve au bit 3, mais comment à partir du résultat on peut remonter à l'erreur ? Il suffit simplement d'utiliser la méthode de correction par syndrome. Pour cela on regarde la colonne qui correspond au vecteur syndrome dans la matrice de correction donc pour le résultat obtenu précédemment, on remarque que la colonne 3 de H correspond au vecteur syndrome donc on sait que l'erreur vient du bit 3.

Premier vrai code correcteur : le code de Hamming

Le code de répétitions vu précédemment avait des défauts : un rendement médiocre, un seul code parfait avec une petite longueur. Mais heureusement il y a d'autres codes comme celui de Hamming, et c'est celui-ci que nous allons voir. Avant de commencer nous vous proposons une petite rétrospective sur qui était Richard Hamming et pourquoi il a inventé le code Hamming.



Qui est Richard Hamming ?

Richard Wesley Hamming, né le 11 février 1915 à Chicago et décédé le 7 janvier 1998 à Monterey est un mathématicien célèbre. Il a notamment inventé un code portant son nom : le code de Hamming.

Histoire de la création du code de Hamming

Quand il a inventé ce code, Hamming travaillait alors sur un calculateur à carte perforée, problème : suite a des problèmes de transmission, le calculateur plante constamment. Ainsi pour palier a ce probleme il va chercher un moyen de limiter les erreurs de calculs en détectant et corrigeant certaines erreurs: c'est la création du code de Hamming.

Comment fonctionne le code de Hamming

Le code de Hamming reprend les fondamentaux de ce que l'on a vu précédemment avec le code de répétitions. Sauf qu' ici au lieu d'avoir des répétitions de bits, nous allons ajouter des bits de parités. Ils serviront à indiquer la parité d'une partie du message leurs étant individuellement désignés.

Pour comprendre comment le code de Hamming fonctionne nous allons utiliser les paramètres suivants [7,4,3]. Cela veut dire que nous aurons :

- n = 7, La longueur du message reçu sera donc de 7.
- k = 4, La longueur du message une fois décodé est de 4.
- δ = 3, La distance minimum étant de 3 nous allons pouvoir détecter et corriger une erreur.

Nous reviendrons plus tard sur comment nous avons déterminé ces paramètres.

Pour commencer dans notre vecteur $\vec{v}$ qui contient nos 7 bits, les bits de donnée que nous allons nommé dx pour les bits, ils correspondent à notre message original. Puis les bits px pour les bits de parité. Ce qui donnera un fois coder ce message :

$$\vec{v} = \begin{bmatrix} b1 & b2 & b3 & b4 & p1 & p2 & p3 \end{bmatrix}$$

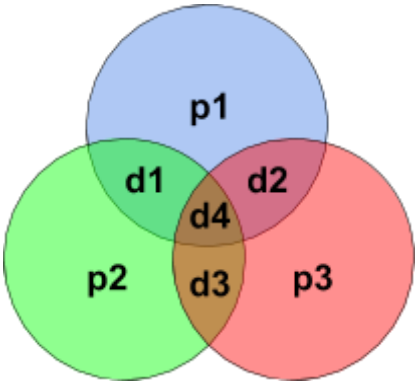
On va venir attribuer la valeur les bits de parité de cette manière :

$$p1 = d1 \oplus d2 \oplus d4$$

$$p2 = d1 \oplus d3 \oplus d4$$

$$p3 = d2 \oplus d3 \oplus d4$$

On peut aussi voir le code comme ceci :



Le choix des bits de parités n'est pas fait au hasard mais nous reviendra ultérieurement dessus.

Approche linéaire

Pour commencer si nous voulons coder notre message nous devons d'abord obtenir la matrice génératrice. Avec le code de Hamming celle-ci est composée de la matrice identité d'ordre k donc ici 4 ainsi que de la matrice de parité appelée P transposée. Ce qui donne :

$$G = \begin{bmatrix} I_k & | & P^T \end{bmatrix}$$

Donc pour obtenir la matrice génératrice, il nous faut le matrice de parité appelé P qui est obtenue à partir du calcul de parité que nous avons fait au dessus :

$$p1 = d1 \oplus d2 \oplus d4$$

$$p2 = d1 \oplus d3 \oplus d4$$

$$p3 = d2 \oplus d3 \oplus d4$$

Conversion en matrice →

$$P = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Ceci nous permet d'obtenir la matrice génératrice G suivante :

$$G = \begin{bmatrix} I_4 & | & P^T \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Et bien sûr la matrice de contrôle H pour vérifier les messages qui est composé de la matrice de parité et d'une matrice identité d'ordre n-k :

$$H = \begin{bmatrix} P & | & I_{n-k} \end{bmatrix}$$

$$H = \begin{bmatrix} P & | & I_{7-4} \end{bmatrix}$$

$$H = \begin{bmatrix} P & | & I_3 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Nous avons donc nos deux matrices G et H, nous pouvons donc coder, détecter et corriger. Prenons par exemple le message "0111".Pour commencer on passe notre message sous forme de vecteur.

$$\vec{v} = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

Ensuite nous allons encoder le message :

$$\vec{v} = \vec{v}G$$

$$\vec{u} = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\vec{u} = \begin{bmatrix} 0 & 1 & 1 & 1 & 2 & 2 & 3 \end{bmatrix}$$

appliquons le modulo 2

$$\vec{u} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Soit les deux situations suivantes une où le message reçu est correct et une autre où il est incorrect :

Message correct	Message incorrect
$\vec{u} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$	$\vec{u} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$
$\vec{w} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$	$\vec{w} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$
$\vec{w} = \begin{bmatrix} 2 & 2 & 4 \end{bmatrix}$	$\vec{w} = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$
appliquons le modulo 2	appliquons le modulo 2
$\vec{w} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$	$\vec{w} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$
Aucune erreur détectée !	Une erreur à été détectée, grâce au vecteur syndrome on retrouve que l'erreur est situé au bit 6

Déterminer les paramètres de Hamming

Pour déterminer les paramètres de Hamming de distance minimum 3, il est existe une formule plutôt simple :

$$[2^p - 1,\ 2^p - p - 1,\ 3]$$

p correspond à n-k soit le nombre bit de parité qui seront présent dans le code. Reprenons notre exemple où nous avons 3 bit de parité :

- $n = 2^p - 1 = 2^3 - 1 = 7$
- $k = 2^p - p - 1 = 2^3 - 3 - 1 = 4$

Nous retrouvons bien les paramètres [7,4,3]. Il est possible donc de déterminer tous les paramètres possible du code de Hamming si $p \geq 2$. Tous les futurs paramètres seront des codes parfaits.Pour exemple :

$$2^p - 1 + 1 = 2^p$$

$$2^p = 2^p$$

Pour un code de paramètre [7,4,3] nous avons un ratio de 4/7 = 57,1%. Il faut savoir que plus p est grand plus le rendement est meilleur par exemple avec p = 8 on obtient un rendement d'environ 96,9% !

Code Hamming dit "étendue"

Le problème actuellement c'est que si notre message contient deux erreurs alors il va en créer une troisième lors de la correction de celle-ci. Alors pour remédier à ce problème on va venir augmenter la distance de Hamming à 4, ce qui va nous permettre de détecter deux erreurs même si on ne pourra pas les corriger. Pour cela nous allons rajouter un bit parité qui fera la parité sur l'ensemble du code.

Pour déterminer les paramètres du code de Hamming étendue il faut utiliser ces formules :

$$[2^p,\ 2^p - p - 1,\ 4]$$

En reprenant les nombre de bit de parité du dernier code pour p :

- $n = 2^p - 1 = 2^3 = 8$
- $k = 2^p - p - 1 = 2^3 - 3 - 1 = 4$

On obtient un code de paramètre [7,4,4].

Implémentation du code de Hamming en python

Cf : Annexes 2 et 3 Hamming.ipynb & HammingRealCase.ipynb

Dans les exemples présentés dans l'annexe python, nous utiliserons un code de Hamming "étendue" [16,11,4]. Les positions des bits de parité sont répartis de façon à ce que le p1 vérifie tous les bits dont les numéros d'identifications possèdent le bit à la position 1 soit 1. Le bit p1 vérifiera donc les bits : d3 (0011),d5 (0101),d7 (0111),d9 (1001),d11 (1011),d15 (1111).

Cette méthode de répartition sera utilisée sur tous les bits de parité. Le bit de parité de Hamming étendue que l'on appellera p0 est situé en position 0, ce qui donnera l'arrangement suivant :

p0 (0000)	p1 (0001)	p2 (0010)	d3 (0011)
p4 (0100)	d5 (0101)	d6 (0110)	d7 (0111)
p8 (1000)	d9 (1001)	d10 (1010)	d11 (1011)
d12 (1100)	d13 (1101)	d14 (1110)	d15 (1111)

Lors d'une transmission on d'un stockage les erreurs parasite on souvent lieux au même endroit et sur le plusieurs bits c'est pour ca que nous allons entrelacer nos messages comme ça si une erreur à lieu sur 4 bit elle aura lieux sur 4 message différent ce qui permet de corriger individuellement ses erreurs.

Dans la première annexe vous retrouverez une implémentation basique de Hamming.

Dans la deuxième annexe vous retrouvez une implémentation avec une simulation de transmissions d'une image.

Conclusion sur Hamming

Le code de Hamming est un très bon correcteur il permet de corriger des erreurs simples mais malheureusement dans des conditions de transfert très défavorable il s'avère inefficaces. Mais il existe une solution, ce sont les codes cycliques.

Code correcteur d’erreurs cyclique

Grâce à l’étude d’un code de Hamming simple , nous avons vu l’utilité d’un code linéaire dans la détection d’erreurs et la transmission d’information. Néanmoins, il existe des types de codages beaucoup plus performant, beaucoup plus fiable et qui permettent de non seulement détecter les erreurs mais également de les corriger sous réserve d'altérations modérées. La différence avec cette méthode de codage plus avancée et la précédente, est le fait qu’elle ne s’applique que sur un des méthodes de codage spécifiques. On ne va plus parler de code correcteur d’erreurs linéaire mais de codes correcteur d’erreurs cycliques.

Tout d’abord, il faut savoir que cette famille de code forment un sous-groupe des codes correcteur d’erreurs linéaires. Cela veut dire que tout ce que l’on a vu précédemment sur les généralités des codes linéaires s’applique également aux codes cycliques. C’est pour cela que l’on peut définir un code cyclique comme un code linéaire tel que toute permutation circulaire d'un mot du code est encore un mot du code. Il n’est donc pas étonnant que le code de Hamming soit essentiel pour la plupart des codes cycliques, notamment pour mesurer la gravité de l'altération des mots de code. La différence entre un code linéaire et un code cyclique est le fait qu’un code cyclique s’appuie sur la théorie des corps finis ce qui permet d’avoir un cadre algorithmique beaucoup plus fort qu’un simple code linéaire. Ce qu’il faut savoir est que la théorie des corps finis s’appuie elle-même sur de nombreuses propriétés algébriques mais en grande partie sur les polynômes. Nous pouvons donc exploiter un nouveau moyen mathématique pour corriger les erreurs, cela permettra d’avoir une correction automatique de certaines altérations de message avec une fiabilité beaucoup plus forte. Il est donc logique que les codes de correction d’erreurs cycliques soient considérés comme des codes parfaits.

Les codes BCH

Un sous-ensemble des codes cycliques qui utilise des polynômes dans un champ défini (le champ de Galois). Ils offrent l’avantage de permettre un contrôle précis sur le nombre de symboles que l’on peut corriger dans un code. Les codes BCH sont d’abord définis pour les symboles binaires puis généralisés pour n’importe quel code p^m où p est un nombre premier et m un nombre positif.

Ils sont d’abord décrits dans des symboles binaires puis généralisés pour tout codes p^m, p un nombre premier et m tout entier positif. On va utiliser un ensemble de corps finis tel que GF (2^8) pour que chaque symbole soit codés sur 8 bits donc 1 octet.

Pour l’utilisation des codes BCH on à deux étapes :

Le **codage** (qui peut être systématique ou non) et le **décodage**

Le codage

Comme chaque polynôme étant un multiple du polynôme générateur est un mot de code BCH valide, l’encodage BCH est simplement le processus de trouver un polynôme qui a le générateur en facteur.

Lors du décodage on se contente de trouver le mot de code avec la distance de Hamming minimum par rapport au mot de code reçu, ainsi un code BCH peut être implémenter comme un code systématique ou non puisque de toute façon cela ne change rien à l’encodage.

En codage BCH non-systématique (coder le message comme un facteur) :

Le moyen le plus direct de trouver un polynôme qui est un multiple du polynôme générateur est de calculer le produit d'un polynôme choisi arbitrairement et du polynôme générateur. Dans ce cas, le polynôme arbitraire peut être choisi en utilisant les symboles du message comme coefficients.

BCH systématique (coder le message comme un préfixe) :

Pour les codes systématiques cela signifie que le message est mis en premier dans le code puis les bits de redondance donc si on affiche le code on aurait par exemple "hello world xf/0f" avec hello world étant nos données et xf/0f les données ajoutées pour permettre de contrôler et corriger les erreurs. L'un des intérêts de cette méthode est que le receveur peut retrouver le message original après avoir effectué la correction simplement en ignorant tout le contenu après le dernier bit d'information. (par exemple si on envoie un code de 255 bits avec 223 bits d'informations et le reste en redondance, il suffira de lire les 223 premiers bits pour voir le message). De plus, si on a plus d'erreurs que le maximum d'erreurs corrigibles, on peut simplement ne rien faire et lire le message tel quel pour peut-être avoir un message encore identifiable (si les erreurs sont sur les bits de redondance).

Le décodage

On peut le diviser en plusieurs sous-parties :

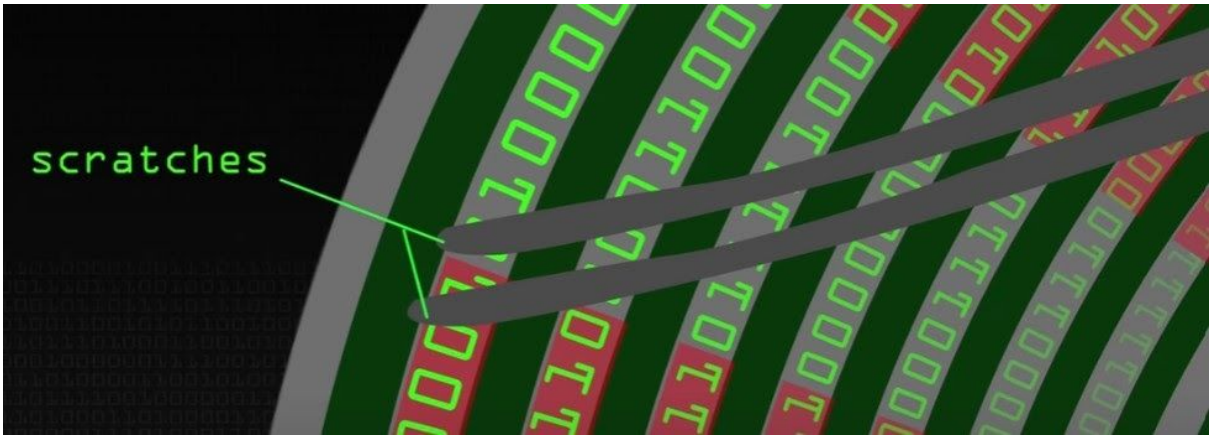
- trouver les syndromes
- localiser les erreurs
- corriger le tout

Au cours de certaines de ces étapes, l'algorithme de décodage peut déterminer qu'il y a trop d'erreurs pour que le code puisse être corrigible et ainsi il pourrait éventuellement produire un message valide différent du message original.

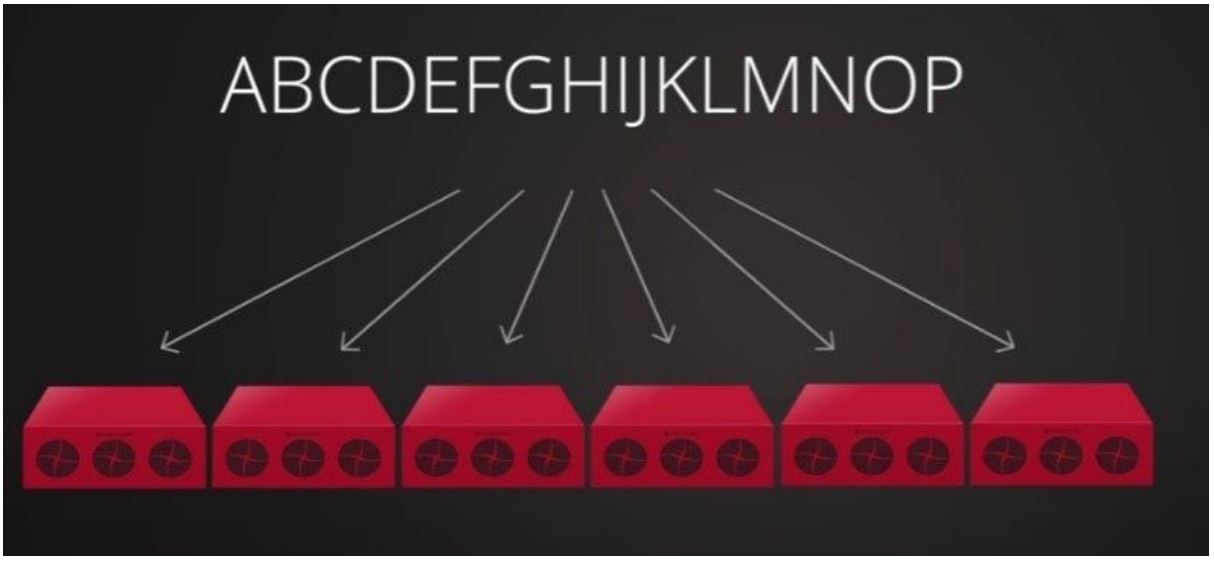
Voir un exemple d'implémentation dans l'annexe 1 : Theorie_information.ipynb ou bch.py

Le code de Reed Solomon

C'est un sous ensemble basé principalement sur les codes BCH, ce type de code correcteur est notamment utilisé dans les CDs. (vu que des rayures ou poussières sur un CD risques de corrompre plusieurs bits à la suite donc on ne pourrait pas utiliser un code correcteur comme celui de Hamming)



La théorie derrière le fonctionnement de Reed-Solomon est la même que celle des codes BCH expliqué plus haut sauf qu'il est optimal dans le sens où c'est un code **MDS (maximum distance separable)** donc il suit le "**Singleton bound**". Such a code is also called a maximum distance separable (MDS) code. aussi utilisé pour stocker des informations sur différents serveurs et s'assurer qu'il n'y ai pas d'erreurs :



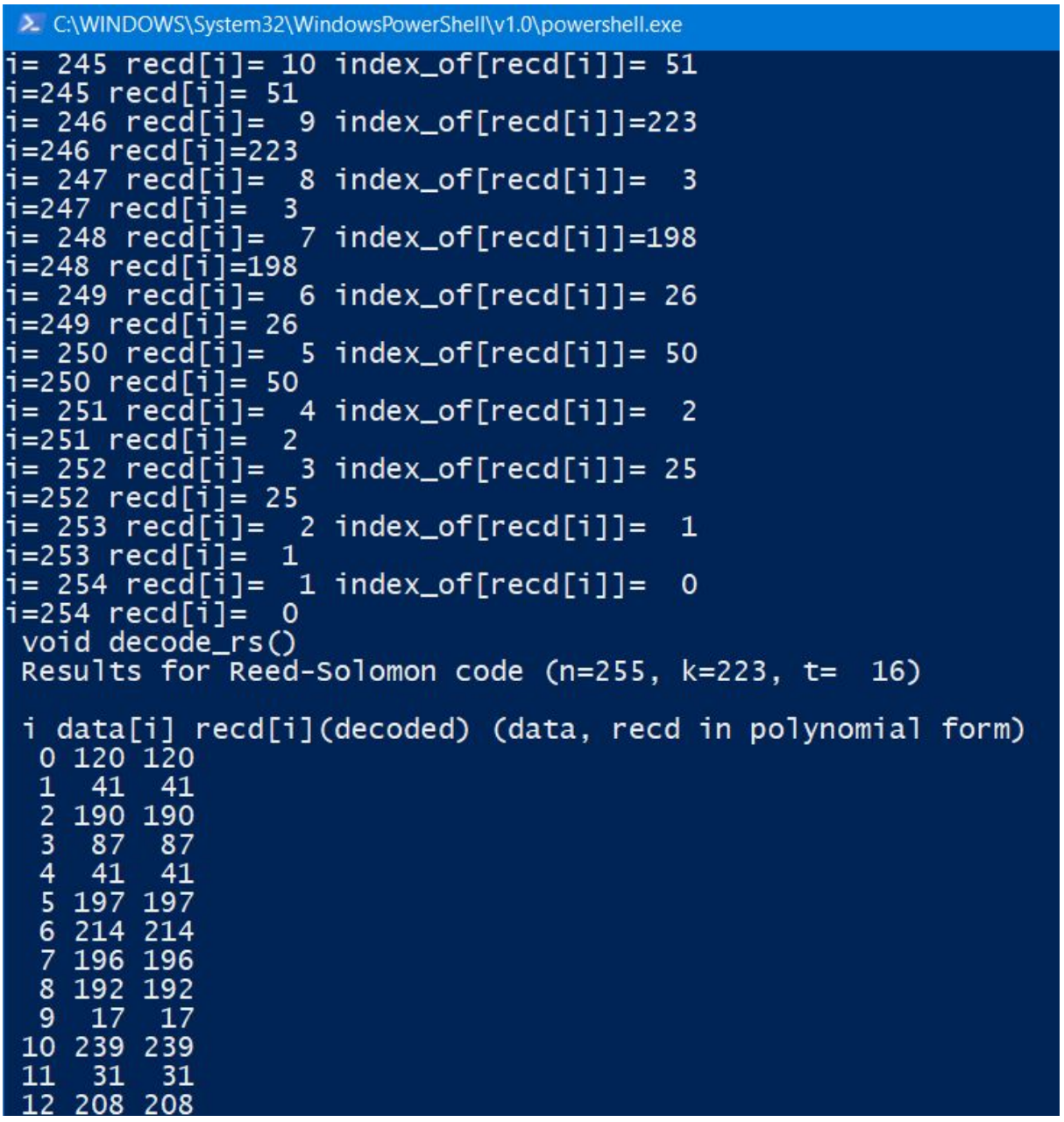
Explications vidéos pour Blackblaze qui utilise cette méthode https://youtu.be/jgO09opx56o

Voir un exemple d'implémentation dans l'annexe 4 : codeReed_Solomon.ipynb.

Il y a un exemple d'encodage et décodage avec Reed-Solomon en C (même si le code n'est pas très élégant, il vient de

http://dspace.calstate.edu/bitstream/handle/10211.2/2732/AlotaibiMasterProject.pdf?sequence=1) que vous pouvez voir dans le fichier **reedSolomon.c**.

Lors de l'exécution du code, on encode des données et les décode donc pour cela on génère le champ de Galois puis le polynôme générateur et pour la partie décodage on trouve le syndrome (ou vecteur d'erreurs), on trouve la localisation des erreurs si il n'y en a pas trop (si il y a trop d'erreurs pour toutes les corrigées alors on ne touche à rien), ensuite on cherche le facteur du polynôme pour finalement décoder les données. Lors de l'exécution du programme la console affiche beaucoup de texte notamment les données de base, les données une fois encodées puis le décodage.



Conclusion

En réponse à la problématique, oui, il est possible de stocker ou envoyer des informations sans les détériorer, de manière fiable et efficace.

En effet, pour gérer les erreurs, de nombreuses solutions ont été mises en place et celles-ci peuvent avoir différents domaines d'application comme les codes de Hamming pour la RAM où peu d'erreurs successives se produisent, et les codes de Reed-Solomon pour les CDs, serveurs, transmissions satellites etc.

Grâce à ces différentes méthodes, le stockage de données ou les communications à distance sont possibles pour tout le monde malgré une infinité de perturbations présentes dans notre quotidien.

Annexes

Annexe 1 : Théorie de l'information

Un document Jupyter qui donne des explications sur ce qu'est un code correcteur d'erreurs et les différents types qui existent.

Annexe 2 : Hamming

Un document Jupyter qui fournit une implémentation simple du code de Hamming.

Annexe 3 : Hamming real case

Un document Jupyter utilise le code de Hamming dans un exemple concret en lui fournissant un fichier avec des erreurs où il doit les détecter.

Annexe 4 : Reed-Solomon

Des explications sur Reed-Solomon ainsi qu'un exemple d'implémentation du code.

Sources

codes correcteurs linéaire https://youtu.be/dYrH5Cbcn6M

Explications simple sur les codes correcteurs d'erreurs : https://youtu.be/q-3BctoUpHE

Wikipedia :

https://fr.wikipedia.org/wiki/Th%C3%A9orie_de_l%27information

https://fr.wikipedia.org/wiki/Code_correcteur

https://fr.wikipedia.org/wiki/Code_lin%C3%A9aire

https://fr.wikipedia.org/wiki/Code_cyclique

https://fr.wikipedia.org/wiki/Code_de_Reed-Solomon

https://fr.wikipedia.org/wiki/D%C3%A9codage_par_syndrome

https://fr.wikipedia.org/wiki/Code_de_Hamming

Cours de Pierre Abrugati :

http://www.lirmm.fr/~chaumont/download/cours/codescorrecteur/Cours_Pierre_Abbrugiati.pdf

Cours ENS : http://perso.eleves.ens-rennes.fr/people/Emily.Clement/documents/COCO.pdf

Codes algébriques principaux : https://hal.inria.fr/inria-00543322/file/Presentation.pdf

Utilisation Reed-Solomon pour stockage sur serveurs :

https://www.youtube.com/watch?v=jgO09opx56o

https://www.backblaze.com/blog/reed-solomon/

Polynomes primitifs :

https://link.springer.com/content/pdf/bbm%3A978-1-4615-1509-8%2F1.pdf

Python Reed-Solomon : https://pypi.org/project/unireedsolomon/

MDS codes : https://www.johndcook.com/blog/2020/03/07/mds-codes/

Rapport en Master sur une implémentation de Reed-Solomon en C avec beaucoup d'explications sur les codes correcteurs en général :

http://dspace.calstate.edu/bitstream/handle/10211.2/2732/AlotaibiMasterProject.pdf?sequence=1

QR code et Reed-Solomon :

http://www-igm.univ-mlv.fr/~dr/XPOSE2011/QRCode/fonctionnement.html#:~:text=Le%20code%20de%20Hamming%20permet,entre%20deux%20mots%20du%20code.&text=Et%20ce%2C%20gr%C3%A2ce%20au%20code,Le%20code%20de%20Reed%2DSolomon.

Électronique, encodage et décodage de Reed-Solomon :

https://www.youtube.com/watch?v=KAZA6iVC23M