

UNIVERSITÉ DE NANTES

IUT DE NANTES

---

# Codes Correcteurs

---

*Auteurs :*

Brewal HENAFF  
Cédric BERLAND  
Nathan MARAVAL

*Cours :*

Modélisation  
Mathématique

16 janvier 2016



UNIVERSITÉ DE NANTES

# Sommaire

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>La théorie</b>	<b>2</b>
2.1	La détection d'erreur . . . . .	2
2.2	La correction d'erreur . . . . .	2
2.3	Le code de Hamming . . . . .	3
2.3.1	La distance de Hamming . . . . .	3
2.3.2	Le code de Hamming . . . . .	4
<b>3</b>	<b>Notre code</b>	<b>6</b>
3.1	L'envoi du message . . . . .	6
3.2	Le brouillage . . . . .	7
3.3	La réception et le décodage du message . . . . .	7
<b>4</b>	<b>d'assault</b>	<b>7</b>
4.1	Tables and Figures . . . . .	7
4.2	Mathematics . . . . .	7
4.3	Lists . . . . .	8
<b>A</b>	<b>Annexe 1</b>	<b>8</b>

# 1 Introduction

Ce projet à été réalisé dans le cadre de la formation en Modélisation Mathématique, en DUT Informatique à l'IUT de Nantes.

Il consiste à concevoir un programme permettant l'encodage d'un message binaire, puis par différentes méthodes que nous expliciterons plus tard, de le décoder et de corriger les possibles erreurs de transmission.

## 2 La théorie

Cette partie couvrira la théorie, les opérations effectués lors de ce projet, que ce soit lors de la transmission, ou bien de la réception, comme du brouillage qui sera effectué pour pouvoir tester les données.

### 2.1 La détection d'erreur

Il existe plusieurs méthodes pour détecter les erreurs, par exemple associer un mot à une lettre comme utilisé dans l'armée :

exemple : erreur  $\rightarrow$  Echo Romeo Romeo Echo Uniforme Romeo

En informatique on utilise ce qu'on appelle des "bits de parité" qui indiquent si le nombre de 1 dans l'octet est pair ou non

exemple : A  $\rightarrow$  1000001 et ajoutera donc un 0 puisque que il y a deux 1 on obtiendra donc pour A le code suivant "01000001".

Le bits de parité nous permet de savoir si lors de la transmission le message à été modifié, simplement en regardant si le bit de parité correspond toujours au reste du message (si le nombre de 1 est toujours pair).

### 2.2 La correction d'erreur

Les différents exemples ci-dessus ne permettent que de détecter les erreurs. Mais ce qui nous intéresse vraiment c'est la correction des erreurs.

Il existe des moyens simples, par exemple la duplication du message :

"erreur"  $\rightarrow$  eee rrr rrr eee uuu rrr

On répète chaque lettre 3 fois. Pourquoi 3 fois ? Et pas 2 ?

Imaginons que l'on multiplie seulement 2 fois, on reçoit le message suivant : "ââgmee", le message de base peut être "âge" mais aussi "âme".

En revanche si l'on multiplie 3 fois on recevra donc un message tel que "âââgmmeee", en supposant qu'il n'y ai eu qu'une erreur, le message envoyer est donc "âme".

Le problème de ce genre de code de correction c'est qu'ils sont très lourd, et coûteux, on transmet trois fois les données, et lorsque l'on voit le poids des données généralement transmissent (images, musiques, vidéos), on s'en rend vite compte.

## 2.3 Le code de Hamming

Depuis 1946, Richard Hamming travaille sur un système de calculateur peu fiable. En effet les données transmises sont souvent corrompus, et la machine finis invariablement par planté. C'est pour y remédier que Hamming va inventer son code correcteur.

### 2.3.1 La distance de Hamming

La distance de Hamming est une notion mathématique, évidemment défini par Hamming, qui permet de quantifier une différence entre deux séquences de symboles.

Par exemple :

Entre 000111 et 100101 on a 2 bits de différences.

→ Une distance de Hamming de 2

Une distance de Hamming va être applicable sur un alphabet finis, avec une même taille de bit pour tous les mots.

Par exemple :

000111 et 0111 ne sont pas comparables.

On définit notre alphabet de taille 4,  $A = \{000000 ; 000111 ; 111000 ; 111111\}$

La distance de Hamming fais office de code correcteur :

Le mot 0000001 a une distance de 1 avec 000000, de 2 avec 000111, de 4 avec 111000, et de 5 avec 111111.

Il est donc très probable que le mot reçu 000001 soit en fait le mot transmis 000000.

Pour utiliser la distance de Hamming, 2 notions sont importantes : la distance minimal et la capacité de correction.

La distance minimal va être la distance de Hamming entre les mots de l'alphabet, et la capacité de correction va représenter le nombre de bits pouvant être détectés et corrigés.

Pour illustrer cela reprenons nos exemples précédents :

**La duplication du message :**

On reprends le cas où l'on triple chaque symbole ( $a \rightarrow aaa$ )

Notre alphabet sera  $A = \{aaa; bbb; ccc; \dots; ddd\}$

On a donc une distance minimal de 6.

Et une capacité de correction de 1 bit : aab est corrigé en aaa.

**Le bit de parité :**

On a des mots de 3 bits, dont le premier est un bit de parité.

On obtient 4 possibilités :  $A = \{000; 101; 110; 011\}$

Cette fois la distance minimal est de 2.

Et on a une capacité de détection de 1 bit : 001 est une erreur, cependant il est à 1 de distance entre 000 et 101, donc impossible de corriger.

### 2.3.2 Le code de Hamming

Maintenant nous allons nous pencher sur une utilisation plus complexe et intéressante de la distance de Hamming : le code de Hamming. C'est un code correcteur actuel, dont la distance minimale est égale à trois et la capacité de correction et de 1 bit corrigé.

Généralement on utilise un code de Hamming  $C(7,4)$ . C'est à dire que l'on envoie 7 bits et que les 4 premiers contiennent les données à transmettre. Les 3 derniers servent à la détection des erreurs.

Il s'agit en fait d'un code correcteur parfait, c'est à dire qu'il corrige un bit avec le moins d'information possible (ici 3 bits). On peut le vérifier simplement, attention cependant on par du principe qu'il ne peut y avoir au plus qu'un bit erroné :

- On a 5 états possibles : 4 états d'erreurs + 1 état sans erreur
- On va ajouter p bits de correction, ce qui nous rajoute p états d'erreurs
- Or p bits peuvent contenir  $2^p$  informations

Ce qui nous donne l'équation à respecter :

$$\rightarrow 5 + p = 2^p$$

$$\rightarrow \text{Or } 5 + 3 = 2^3$$

Ainsi C(7,4) utilise le nombre minimal de bits pour transmettre l'information tout en corrigeant une erreur.

Maintenant voyons comment contenir la correction dans 3 bits !

Ces 3 derniers bits sont en fait l'addition, bit à bit, des 4 autres. En pratique cela donne ça :

Soit  $v_1, v_2, \dots, v_7$  les bits transmit et  $u_1, u_2, u_3$  et  $u_4$  les bits contenant le message.

$$v_1 = u_1,$$

$$v_2 = u_2,$$

$$v_3 = u_3,$$

$$v_4 = u_4,$$

$$v_5 = u_1 + u_2 + u_4,$$

$$v_6 = u_1 + u_3 + u_4,$$

$$v_7 = u_2 + u_3 + u_4,$$

Lorsque l'on reçoit le message (bits  $w_1, w_2, \dots, w_7$ ), on est face à 8 possibilités :

- (0) il n'y a pas d'erreur ;
- (1)  $w_1$  est erronée ;
- (2)  $w_2$  est erronée ;
- (3)  $w_3$  est erronée ;
- (4)  $w_4$  est erronée ;
- (5)  $w_5$  est erronée ;
- (6)  $w_6$  est erronée ;
- (7)  $w_7$  est erronée.

Grâce aux 3 derniers bits on peut savoir d'où provient l'erreur. Il suffit de les recalculer (bits  $W_5, W_6, W_7$ ) et de les comparer, pour obtenir un des huit cas précédent :

- (0) si  $w_5 = W_5$  et  $w_6 = W_6$  et  $w_7 = W_7$  ;

- (1) si  $w_5 \neq W_5$  et  $w_6 \neq W_6$  ;
- (2) si  $w_5 \neq W_5$  et  $w_7 \neq W_7$  ;
- (3) si  $w_6 \neq W_6$  et  $w_7 \neq W_7$  ;
- (4) si  $w_5 \neq W_5$  et  $w_6 \neq W_6$  et  $w_7 \neq W_7$  ;
- (5) si  $w_5 \neq W_5$  ;
- (6) si  $w_6 \neq W_6$  ;
- (7) si  $w_7 \neq W_7$ .

Cependant ce code ne permet de détecter et corriger qu'une seule erreur.

Matrice génératrice  $G_h$  :

$I_4$

bits de parité  $P$

$$G_h = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Matrice H :

$$H = \begin{pmatrix} P^T \\ I_{7-4} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrice de contrôle  $H_h^T$  :

$$H_h^T = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

## 3 Notre code

### 3.1 L'envoi du message

Test de code :

### 3.2 Le brouillage

Afin de pouvoir tester notre code dans des conditions d'erreurs réelles, nous utilisons un programme permettant de brouiller certains bits, aléatoirement. Ce brouillage nous permet de tester notre code décodeur, étant donné que la probabilité d'un brouillage dû au code d'encodage est extrêmement faible. Pour se faire, nous utilisons le code `transmit.c`, qui simule une transmission de message (comme par exemple l'envoi d'un message à un satellite en orbite, ce qui peut engendrer de nombreuses erreurs).

### 3.3 La réception et le décodage du message

## 4 d'assault

Comments can be added to the margins of the document using the `todo` command, as shown in the example on the right. You can also add inline comments too :

This is an inline comment.

Here's  
a com-  
ment  
in the  
mar-  
gin!

### 4.1 Tables and Figures

Use the `table` and `tabular` commands for basic tables — see Table ??, for example. You can upload a figure (JPEG, PNG or PDF) using the files menu. To include it in your document, use the `includegraphics` command as in the code for Figure ?? below.

### 4.2 Mathematics

L<sup>A</sup>T<sub>E</sub>X is great at typesetting mathematics. Let  $X_1, X_2, \dots, X_n$  be a sequence of independent and identically distributed random variables with  $E[X_i] = \mu$



and  $\text{Var}[X_i] = \sigma^2 < \infty$ , and let

$$S_n = \frac{X_1 + X_2 + \cdots + X_n}{n} = \frac{1}{n} \sum_i^n X_i$$

denote their mean. Then as  $n$  approaches infinity, the random variables  $\sqrt{n}(S_n - \mu)$  converge in distribution to a normal  $\mathcal{N}(0, \sigma^2)$ .

### 4.3 Lists

You can make lists with automatic numbering ...

1. Like this,
2. and like this.

...or bullet points ...

- Like this,
- and like this.

We hope you find write<sub>L</sub>A<sub>T</sub>E<sub>X</sub> useful, and please let us know if you have any feedback using the help menu above.

## A Annexe 1