TS226 - TP de codage canal

Encodage et décodage de codes convolutifs

Thomas MARCHAL - Maxime PETERLIN - Gabriel VERMEULEN

ENSEIRB-MATMECA, Bordeaux

12 juin 2014

Table des matières

1 Encodage de codes convolutifs

1.1 Principe d'encodage d'une séquence $\mathbf{m} = [m_0, \ldots, m_{L-1}]$ de bits à transmettre

Soit n la longueur du code convolutif binaire.
Soient $g_1(D), \ldots, g_n(D)$ les n polynômes générateurs à coefficients dans $\mathbb{F}_2$ servant à encoder la séquence $\mathbf{m} = [m_0, \ldots, m_{L-1}]$. On forme la matrice G dont les lignes sont les coefficients de ces derniers.

Soient $C(D)$ et $\mathbf{m}(D)$ respectivement les séquences de bits encodée et à transmettre que l'on représente sous forme polynomiale.

La relation permettant d'encoder le message $\mathbf{m}$ est la suivante :

$$C(D) = \mathbf{m}(D)G = [\mathbf{m}(D)g_1(D), \ldots, \mathbf{m}(D)g_n(D)] \quad (1)$$
$$= [C_1(D), \ldots, C_n(D)] \quad (2)$$

La séquence envoyée est alors

$$\underbrace{C_{1,0} \cdots C_{n,0}}_{t=0} \ \cdots \ \underbrace{C_{1,t} \cdots C_{n,t}}_{instant\ t}$$

1.2 Mémoire M du codeur

La mémoire M du codeur est donnée par le retard maximal des registres à décalage.
Mathématiquement, cela s'exprime de la manière suivante :

$$M = \max_{i \in [\![1,\ldots,n]\!]} deg(g_i)$$

1.3 Principe de fonctionnement de la fonction *codconv*

La fonction *codconv* prend en paramètre le message à encoder $\mathbf{m}$, ainsi qu'un vecteur g composé des polynômes générateurs associés au code sous forme octale et elle renvoie la séquence de bits à transmettre $\mathbf{C}$.

On cherche à créer une matrice $\mathbf{C}$ dont les lignes seraient les coefficients des polynômes $C_i(D)$, avec $i \in [\![1, \ldots, n]\!]$. C'est à partir de cette matrice qu'on obtiendra la séquence à envoyer.
Par souci de performances, on calcule d'avance la taille de cette matrice :

- Le nombre de lignes est égal au nombre de polynômes générateurs
- Le nombre de colonnes se trouve grâce à la formule permettant de calculer la taille du résultat de la convolution de deux vecteurs. En effet, comme les coefficients du produit de deux polynômes résultent de la convolution des coefficients de ces derniers, alors le produit de polynômes $\mathbf{m}(D)g_i(D)$ revient à convoluer le vecteur $[m_0, \ldots, m_{L-1}]$ et le vecteur contenant les coefficients de g_i. Ainsi, le nombre l de colonne est

$$l = max\ (n + L - 1,\ n,\ L)$$

Une fois que la matrice est initialisée, on la remplie de sorte que la nième ligne soit égale au produit de convolution modulo 2 du vecteur de bits $\mathbf{m}$ et du vecteur contenant les coefficients du polynômes g_n.

Il ne reste plus qu'à modifier la taille de la matrice pour obtenir la séquence de bits à transmettre dans l'ordre défini et expliqué supra.

2 Décodage des codes convolutifs : algorithme de Viterbi

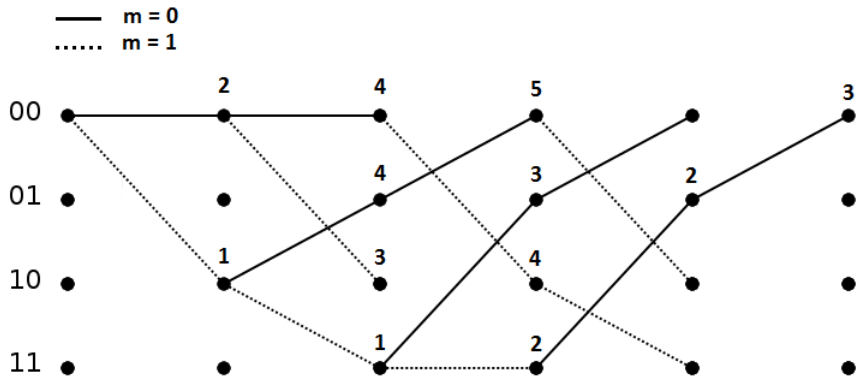
2.1 Principe de fonctionnement de la fonction *paramconv2*

La fonction *paramconv2* prend en paramètre un vecteur représentant les polynômes générateurs en utilisant le système octal. Le but de cette fonction est de renvoyer une matrice T qui représente le diagramme d'états du code, où $\mathbf{T}_{i,j}$ contient la sortie de l'encodeur associée à l'arête reliant l'état i à l'état j sous forme décimale.

On commence par tranformer les données octales des polynômes générateurs en binaire avec les fonctions *base2dec* et *dec2bin*. On calcule M en prenant $M = \underset{i\in[\![1,\dots,n]\!]}{max}\ deg(g_i)$ et on génère une matrice de taille 2^M initialisée à -1 pour différencier les arêtes utiles des non-existantes.
Ensuite on remplit la matrice aux endroits où les arêtes existent, c'est-à-dire pour chaque couple de lignes i et $i+1$ de la matrice, aux colonnes i et 2^{M-1}. Pour ce faire on effectue le calcul des équations de convolutions obtenues à partir des polynômes générateurs et des états possibles des registres.
Ainsi on obtient la matrice du diagramme d'états du code.

2.2 Exemple d'application de l'algorithme de Viterbi

On se propose de décoder la séquence $\mathbf{y} = [110110111010010]$ formée grâce au vecteur $\mathbf{g} = [1, 5, 7]$ caractérisant les polynômes générateurs.
Comme nous avons 3 polynômes générateurs, la longueur du code est, de même, 3.
A partir de ces informations, on peut établir le treillis suivant.



On parcourt le treillis en partant de la fin en prennant jusqu'à revenir à l'état initial.
Dans cet exemple, on obtient la séquence de bits 00111.
Il ne reste plus qu'à inverser l'ordre de la séquence et enlever les $M = 2$ derniers bits pour obtenir le message envoyé, qui est ici $\mathbf{m} = [111]$.

2.3 Principe de fonctionnement de la fonction *decodconv*

La fonction *decodconv* prend en paramètre le message à décoder $\mathbf{y}$, ainsi qu'un vecteur g composé des polynômes générateurs associés au code sous forme octale et elle renvoie le message envoyé $\mathbf{m}$.

On commence par créer une matrice de 2^M lignes et $\frac{L_y}{L_g} + 1$ colonnes qui contiendra les informations relatives au treillis, avec L_y la longueur du message encodé et L_g le nombre de polynômes générateurs.
Si L_y n'est pas divisible par L_g, cela veut dire que la taille du message encodé et le nombre de polynômes

générateurs ne correspondent pas, un message d'erreur est alors renvoyé.

Chaque composante de la matrice représentant le treillis va contenir une cell qui aura la structure suivante.

- Le numéro de la ligne au temps $t-1$ menant à la cell actuelle (i.e. au temps t).
- Le poids associé à la cell actuelle.
- Le bit associé au chemin.

Le treillis va ensuite être parcouru suivant ses colonnes et ses lignes pour remplir les différentes cell en fonction du diagramme d'état du code $\mathbf{T}$.

A chaque fois qu'on arrive sur une cell que l'on peut remplir (en se basant sur la matrice $\mathbf{T}$), alors on commence par calculer le poids du chemin
Ensuite, si la cell du treillis que l'on veut créer n'existe pas, on la crée en remplissant les informations de la structure définie supra. Par contre, si la cell existait déjà et qu'un autre chemin y mène, alors on compare les poids de ces derniers et on retient celui de poids le plus faible.

Une fois que le treillis est rempli, on le parcourt en sens inverse en partant de la cell située dans la première ligne de la dernière colonne grâce aux numéros de lignes enregistrés permettant d'identifier les noeuds précédents.

Il ne reste plus qu'a supprimer les M derniers bits pour ne pas prendre en compte la remise à zéro des registres.

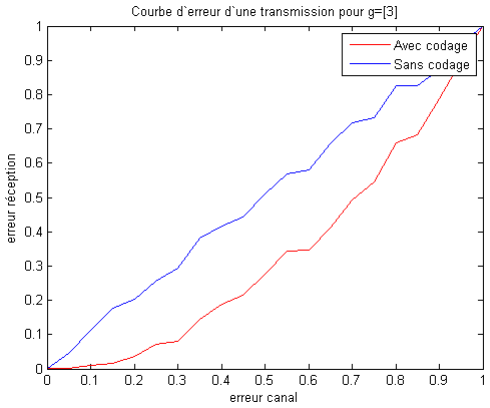
3 Performances

3.1 Performances sur canal binaire symétrique

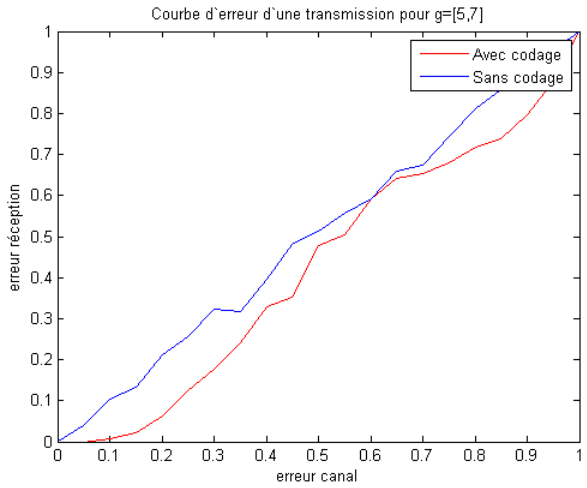
Le décodeur de Viterbi implémenté précédemment va nous permettre de tester les performances du *hard decoding* n'acceptant que des entrées à valeurs dans l'alphabet $\{0, 1\}$.

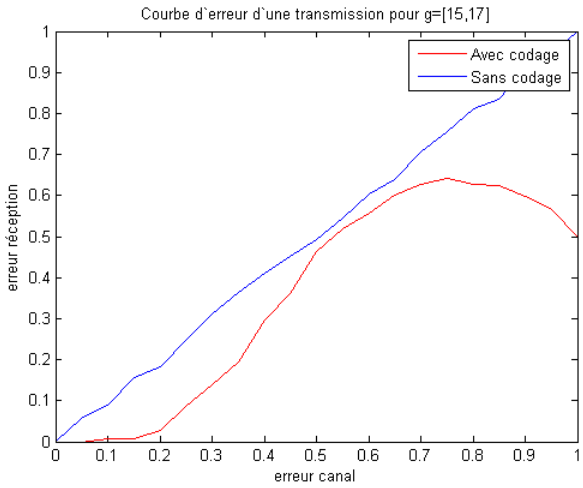
On se place sur un canal binaire symétrique de probabilité d'erreur p. Pour cela, la fonction *bsc* va parcourir un vecteur correspondant au message codé (ou au message initial dans le cas d'une transmission sans codage) et retourner le message transmit bit à bit avec une erreur de probabilité p.

Le message *tout à zéro* est transmis avec un codeur de génératrice $g = [3]$ (en parallèle du message non codé) dans le canal binaire symétrique précédent 500 fois pour nous permettre de moyenner le taux d'erreur à la réception et d'obtenir ainsi des probabilités d'erreurs fiables. La courbe d'erreur ainsi obtenu nous permet d'observer qu'une partie des erreurs survenues lors de la transmission à travers le canal a été corrigé. Le code est d'ailleurs plus performant pour une probabilité d'erreur canal $p = 0.5$ comme le montre la courbe de distance entre les deux messages reçus.



Le même test est effectué pour un codeur de génératrice $g = [5, 7]$ et un codeur de $g = [15, 17]$.





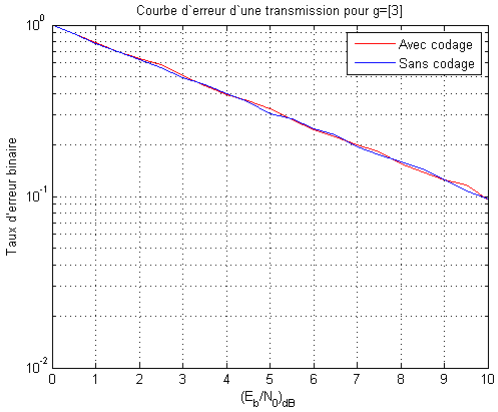
On remarque ainsi que le premier est moins performant que le codeur $g = [3]$ dû à sa proximité avec

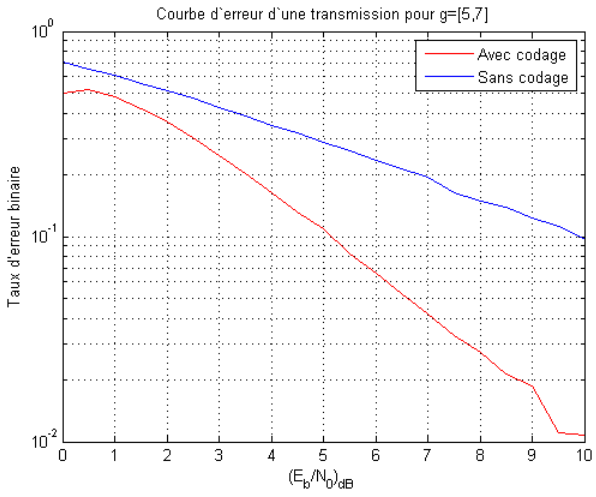
la courbe d'erreur d'une transmission sans codage. Cependant le second nous permet d'obtenir un meilleur décodage dans les intervalles $[0; 0.2]$ et $[0.8; 1]$. La connaissance de la probabilité d'erreur au décodage nous permettra donc de sélectionner le décodeur le plus performant (i.e. la génératrice à utiliser) comme dans l'exemple ci-dessus où le tout premier décodeur est plus performant pour $p \in [0.2; 0.8]$ et le troisième pour $p \in [0; 0.2] \bigcup [0.8; 1]$.

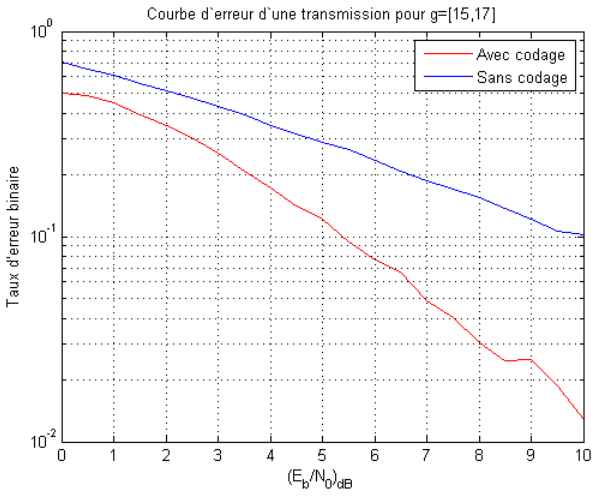
3.2 Performances sur canal gaussien

Le soft decoding diffère du hard decoding de par la manière dont est reçu et traité le signal.
En effet le décodage souple va admettre des entrées à valeurs dans un ensemble plus large que l'alphabet du code. Ainsi après le passage du message dans un canal bruité nous allons calculer les poids non pas avec une distance de Hamming mais avec une distance Euclidienne.
Comme précédemment, la fonction awgn nous permet de modéliser un canal gaussien par lequel va passer le message. La fonction decodconv est, quant à elle, modifiée pour nous permettre de calculer les poids par la distance Euclidienne et ainsi tracer, pour un message "tout à zéro", les probabilités d'erreurs par bit empiriques au décodage en fonction du SNR.

Les génératrices sont les mêmes que pour la partie précédente et nous permettent d'observer les performances du soft decoding dans les 3 cas.







On remarque comme attendu la décroissance du taux d'erreur binaire en fonction du SNR ainsi que l'amélioration à la réception qu'apporte le décodage comparé à une transmission sans codage, bien que le premier codeur de génératrice $g = [3]$ n'apporte pas d'améliorations significatives.