Ecole Nationale Supérieure d'Electronique,
Informatique, Télécommunications,
Mathématique et Mécanique de Bordeaux

TS226 : TP de codage de canal
Codes Convolutifs

Réalisé par :

SQUALLI HOUSSAINI Karima WANKIDA Oumaima

Encadré par :

M. TAJAN Romain Mme. ELLOUZE Malek

Année universitaire : 2021/2022

Table des matières

1 Introduction

Dans la continuité de l'étude des différents blocs de la chaîne de communication numérique initiés en première année, le codage de canal (ou codage correcteur d'erreur) a été introduit afin d'améliorer la qualité de la transmission.

Le codage de canal consiste à introduire de la redondance dans le signal émis, de manière à le protéger contre les erreurs potentielles du canal de transmission.

La nécessité d'introduire de la redondance est évidente. Supposons que l'on émette un message d'informations binaires, et que la détection se fasse bit par bit dans le récepteur. Dans ce cas, chaque bit émis contient de l'information, et toute erreur de transmission engendrera une perte d'information irréversible. Si, au contraire, on introduit des bits de redondance dans le message, on pourra détecter, voire même corriger des erreurs de transmission.

Dans ce TP, nous allons nous intéresser au cas des communications numériques codées à l'aide d'un code convolutif. Les codes convolutifs sont utilisés dans de nombreuses applications afin d'obtenir un transfert fiable de données, telles que la radio, les téléphones portables, les communications terrestres, satellitaires et spatialles.

L'architecture de la chaîne de communications à considérer est présentée sur la Figure 1.

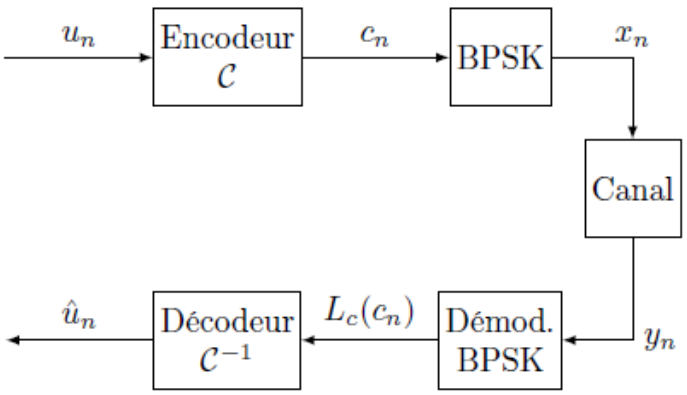

FIGURE 1 – Chaîne de trasmission codée

2 Codes convolutifs

2.1 Treillis

La sortie d'un codeur convolutif dépend d'un symbole courant à coder, du symbole précédent ainsi que du résultat de codage du symbole précéedent. Il est donc à états finis et peut être représenté par un diagramme d'états. Celui-ci peut être représenté à son tour sous forme d'un treillis faisant apparaître le temps. Les lignes repésentent les états, et chaque état à l'instant t est mis dans une colonne. Les Figures 2 et 3 représentent un exemple d'un diagramme d'états du code convolutif de notation octale $(5, 7)_8$ et le treillis associé [1].

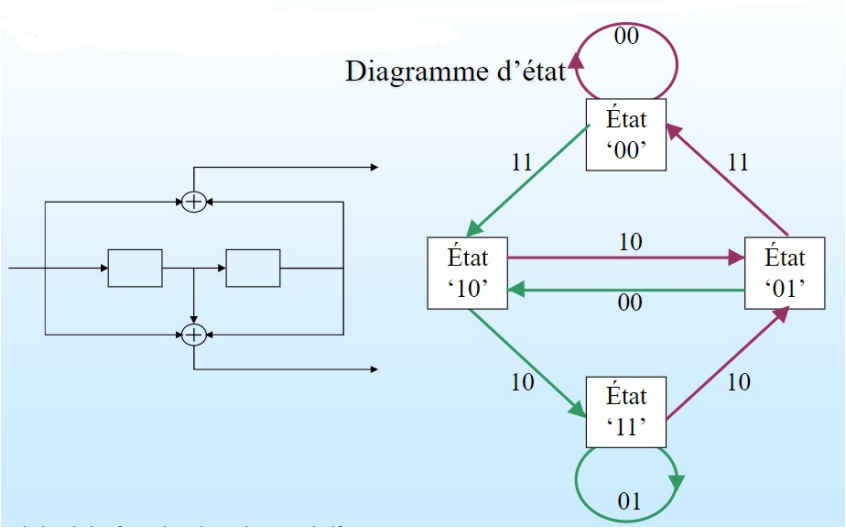

FIGURE 2 – Diagramme d'états du code convolutif $(5, 7)_8$

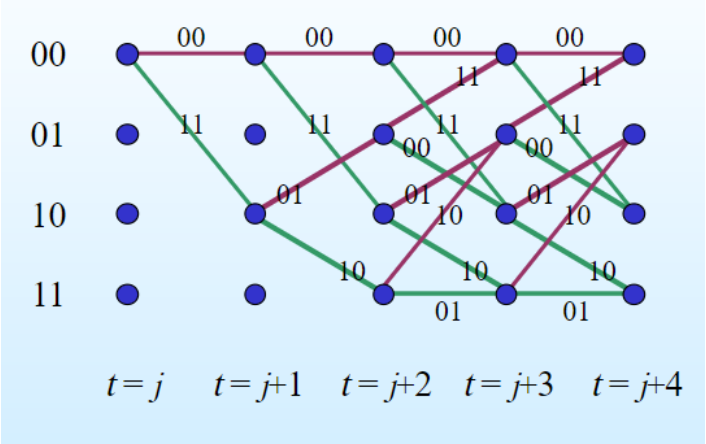

FIGURE 3 – Treillis associé

L'outil *Communications toolbox* de Matlab offre une structure de données qui permet de représenter les diagrammes d'états des codes convolutifs. Ces structures sont construites en utilisant la fonction *"poly2trellis"* de Matlab, qui prend en entrée la longueur de contrainte $\nu = \mathrm{m} + 1$ (m étant la mémoire du code), et le polynôme en notation octale sous forme de vecteur. Dans ce TP, la fonction *"poly2trellis"* a été utilisée pour implémenter les codes convolutifs suivants :

— $(2, 3)_8$: trellis = poly2trellis(2, [2 3]) ;

— $(5, 7)_8$: trellis = poly2trellis(3, [5 7]) ;

— $(13, 15)_8$: trellis = poly2trellis(4, [13 15]) ;

— $(133, 171)_8$: trellis = poly2trellis(7, [133 171]).

2.2 Encodage

La première fonction à implémenter est la fonction d'encodage *"cc_encode"*. Cette fonction prend quatre arguments en entrée :

— u : un vecteur de K symboles d'entrée représentant le messsage à encoder.
— *treillis* : une structure représentant le treillis.
— s_i : l'état initial de l'encodeur.
— *closed* : un booléen indiquant si le treillis est fermé ou pas.

et renvoie deux arguments :

— c : le mot de code de taille n_sK si le treillis est ouvert, n_sL s'il est fermé, avec n_s le nombre de sorties, et $L = n_s(K+m)$.
— s_f : l'état final de l'encodeur.

Tout d'abord, l'algorithme de la fonction parcourt le vecteur u, et remplit le vecteur c avec des valeurs dépendantes de l'état courant et de l'état suivant. En effet, pour tout i allant de 1 à K, *c(i) = treillis.outputs(etat_courant, u(i))*. La variable *etat_courant* est initialement égale à s_i, et à chaque itération, elle prend la valeur *treillis.nextStates(s_i, u(i))*.

La valeur 1 est ajoutée à chacun des indices (k,l) des tableaux *nextStates* et *outputs* de la structure *treillis* pour régler le souci de parcours de tableaux sur Matlab. s_f prend la valeur du dernier état courant.

Si la variable *closed* est égale à *true*, le treillis est fermé et donc un nombre de bits égal à n_sm doit être ajouté au mot de code. Comme le but est de revenir à l'état 0, pour un j allant de 1 à m, *etat_courant* $= \left\lfloor \frac{s\_i}{2} \right\rfloor$, et *c(j+longueur(c)) = treillis.outputs(etat_courant,0)*. De même, la valeur 1 est ajoutée à chacun des indices (k,l) du tableau *outputs*.

La dernière étape de cet algorithme est de convertir les valeurs décimales du vecteur c en bits grâce à la fonction *"de2bit"*. Désormais, le vecteur c est une matrice, il faut donc la convertir en vecteur grâce à la fonction *"reshape"* pour qu'il devienne un vecteur de longueur n_sK ou n_sL selon la fermeture du treillis.

2.3 Décodage

La seconde fonction à implémenter est la fonction d'encodage *"viterbi_ decode"*. Cette fonction prend quatre arguments en entrée :

— y : un vecteur de n_s observations du canal représentant le message à décoder.
— *treillis* : une structure représentant le treillis.
— s_i : l'état initial de l'encodeur.
— *closed* : un booléen indiquant si le treillis est fermé ou pas.

et renvoie un seul argument :

— c : le mot décodé de taille K.

Trois matrices seront utiles à notre décodage, toutes de même nombre de lignes (2^m) et de colonnes (L+1). La première matrice est celle contenant les métriques de branches, que nous avons nommée "*metrique*". La deuxième et troisième matrices sont nommées respectivement "*inputs_ mat*" et "*chemin_ mat*". La matrice "*inputs_ mat*" stocke dans chaque élément d'indice (i,j) la valeur de l'entrée menant à l'état i, et "*chemin_ mat*" stocke les états précédents correspondants aux métriques.

On initialise d'abord les éléments des trois matrices par la valeur *inf*. On remplit ensuite l'élément de "*metrique*" d'indices (s_i+1, 1) par la valeur 0. On parcourt cette matrice colonne par colonne, et pour toute valeur finie d'une colonne j, on remplit les l valeurs (l est le nombre de symboles en entrée) des états suivants (de la colonne j+1) par la valeur de métrique correspondante. À chaque fois qu'un élément d'indices (i,j) de la matrice "*metrique*" est modifié, on modifie la valeur des éléments de mêmes indices des deux autres matrices, "*chemin_ mat*" par l'état précédent (+1), et "*inputs_ mat*" par la valeur de l'entrée menant à l'état actuel.

De cette manière, il nous sera facile de remplir le vecteur u à partir des trois matrices.

La seule valeur de la matrice "*metrique*" qui nous est utile est le minimum de la dernière colonne dont on extrait l'indice de ligne. Ensuite, on parcourt inversement les deux autres matrices colonne par colonne, et à partir de la matrice "*chemin_ mat*" on déduit l'indice de l'entrée de l'état précédent à chaque itération stocké dans la matrice "*entrees_ mat*". Ainsi le vecteur u sera rempli inversement.

Finalement, si le treillis est fermé, on supprime du vecteur u les m bits ajoutés à l'étape de l'encodage.

2.4 Vérification des résultats

Afin de vérifier si nos fonctions d'encodage et de décodage sont bien implémentées, nous avons procédé par deux méthodes :

2.4.1 Méthode 1

Le script "*test.m*" teste l'encodage et le décodage d'un message u de longueur K=2048bits par différents treillis systématiques et non systématiques, fermés ou ouverts, avec une valeur différente de l'état initial s_i à chaque test.

La sortie de l'encodeur donne un vecteur c et, pour décoder le message, on donne en entrée à la fonction *"viterbi_ decode"* le vecteur $y = 1\text{-}2 \cdot c$.

La variable "*correct*" reçoit 1 si le vecteur de sortie du décodeur est identique au vecteur d'entrée, et 0 sinon, tandis que la variable "*error*" calcule le nombre de bits faux.

Quel que soit le treillis utilisé, le résultat est toujours le suivant :

```
error =

     0


correct =

     1
```

FIGURE 4 – Affichage après l'exécution de *test.m*

2.4.2 Méthode 2

L'utilisation de l'outil MATLAB *bertool* nous a aussi été utile pour tester notre encodeur/décodeur.

Nous avons affiché sur une même figure nos différentes courbes de TEB obtenues grâce à notre encodeur/décodeur, pour différents treillis, ainsi que celles obtenues par l'outil *bertool*. Pour le code convolutif de notation octale (2,3)_8, nous avons remarqué que les deux courbes de TEB (Taux d'Erreur Binaire) se superposent parfaitement à partir d'une valeur de $\frac{Eb}{N0}$ égale à 3 dB (Figure 5). Ceci est le cas pour tous les autres encodeurs .

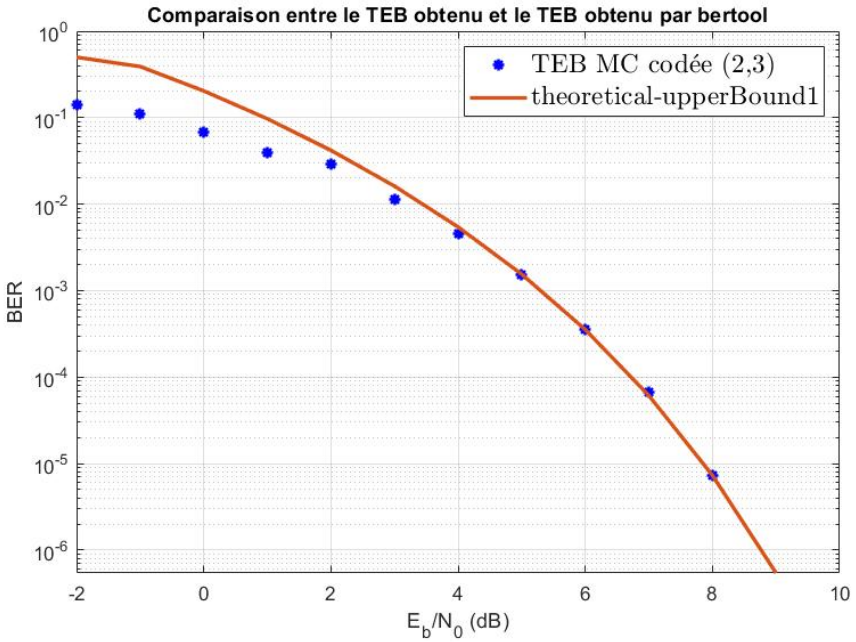

FIGURE 5 – Courbe comparative du TEB obtenu avec celui otbtenu par bertool

2.5 Etude de l'impact de la mémoire du code

Dans cette partie, nous allons faire une étude comparative de l'impact de la mémoire des différents codes convolutifs sur leurs performances en nous basant, tout d'abord sur le taux d'erreur binaire pour des valeurs de *SNR* (Rapport Signal sur Bruit) allant de -2 à 10 dB, et ensuite sur les valeurs de multiples paramètres (détaillés dans ce qui suit).

2.5.1 Taux d'erreur binaire

La Figure 6 illustre les courbes des TEB des différents codes convolutifs étudiés, obtenues grâce aux fonctions de l'encodage/décodage que nous avons implémentées.

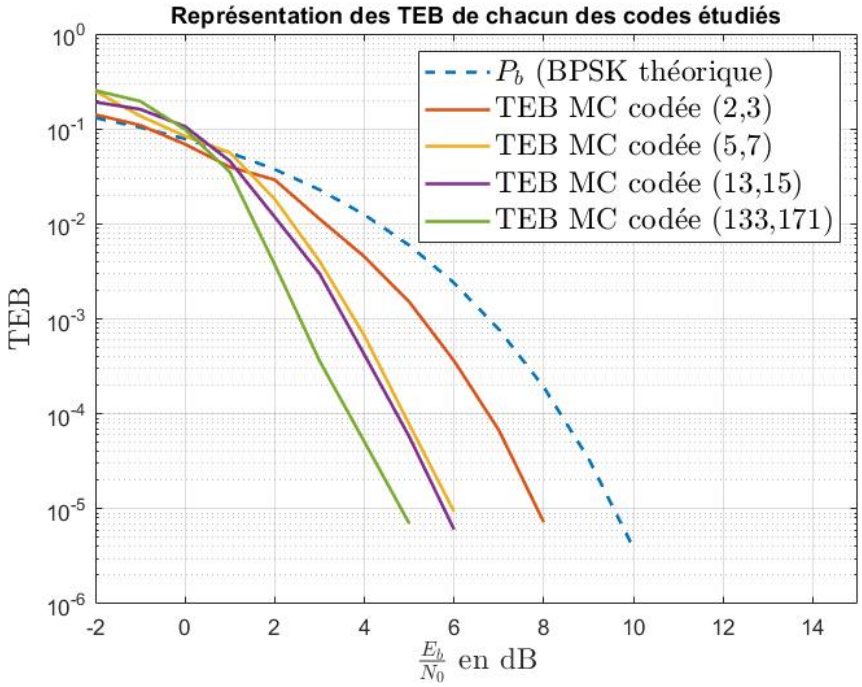

FIGURE 6 – Courbe représentant les TEB des différents codes convolutifs

$\Longrightarrow$ La Figure 6 montre clairement que plus on augmente la mémoire, plus le taux d'erreur binaire diminue pour différentes valeurs de $\frac{Eb}{N0}$ en dB, et par suite plus le code est précis. La mémoire est donc un paramètre important qui définit la précision de notre encodeur/décodeur.

2.5.2 Gain de codage

D'après la Figure 6, la valeur de $\frac{Eb}{N0}$ pour laquelle la stratégie non codée passe sous Pb=10^{-5} est approximativement 9,56 dB. De même, on relève les valeurs de $\frac{Eb}{N0}$ correspondant à Pb=10^{-5} pour les différents codes convolutifs :

— Pour $(2, 3)_8$: $\frac{Eb}{N0} = 7.9$ dB ;

— Pour $(5, 7)_8$: $\frac{Eb}{N0} = 6$ dB ;

— Pour $(13, 15)_8$: $\frac{Eb}{N0} = 5.9$ dB ;

— Pour $(133, 171)_8$: $\frac{Eb}{N0} = 4.8$ dB.

Le gain de codage de chacun de ces codes est donc le suivant :

— Pour $(2, 3)_8$: G = 1.66 dB ;

— Pour $(5, 7)_8$: G = 3.47 dB ;

— Pour $(13, 15)_8$: G = 3.66 dB ;

— Pour $(133, 171)_8$: G = 4.76 dB.

$\Longrightarrow$ L'analyse du gain de codage de ces différents codes nous permet de remarquer que plus la mémoire est grande, plus on gagne sur la performance de l'encodage.

2.5.3 Débit de sortie du récepteur

Ce débit a été évalué dans le script fourni. Nous obtenons pour la dernière valeur de $\frac{Eb}{N0}$ les valeurs du débit suivantes :

— Pour $(2, 3)_8$: D = 0.94 Mo/s ;

— Pour $(5, 7)_8$: D = 1.01 Mo/s ;

— Pour $(13, 15)_8$: D = 0.91 Mo/s ;

— Pour $(133, 171)_8$: D = 1.82 Mo/s.

$\Longrightarrow$ Pour les trois premiers codes convolutifs, caractérisés par des valeurs de mémoire assez proches (respectivement 1,2, et 3), la valeur du débit est restée à peu près la même, contrairement au quatrième code (mémoire=7) dont une importante augmentation du débit a été notée.

2.5.4 Tableau récapitulatif de synthèse

La Table 1 est une synthèse de notre étude explicitant pour chaque code (de treillis ouvert) son rendement, sa mémoire et sa distance minimale.

Rappelons que le rendement du code est le rapport entre le nombre de bits en entrée et le nombre de bits en sortie, et le nombre d'états est égale à 2^m, m étant la mémoire. La distance minimale de chacun des codes est obtenue grâce à la fonction *"distspec"*. *distspec(treillis)* donne en effet une structure de données, dont le champ *dfree* contient une valeur égale à la distance minimale du code.

Encodeur	Rendement	Mémoire	Nombre d'états	Distance minimale	Gain de codage (en dB)	Débit de décodage (en Mo/s)
$(2, 3)_8$	1/2	1	2	3	1.66	0.94
$(5, 7)_8$	1/2	2	4	5	3.47	1.01
$(13, 15)_8$	1/2	3	8	6	3.66	0.91
$(133, 171)_8$	1/2	7	128	10	4.76	1.82

TABLE 1 – Tableau récapitulatif

$\Longrightarrow$ Nous remarquons que la distance minimale est d'autant plus importante que la mémoire est grande. L'encodeur pourra donc détecter et corriger plus d'erreurs, ce qui favorisera sa performance.

2.6 Prédiction des performances

Nous avons étudié jusqu'ici les codes convolutifs en réalisant des simulations de la méthode de Monte Carlo. Dans cette partie, nous allons nous intéresser à l'étude des performances par la méthode de l'impulsion qui permet de déterminer le spectre de distance d'un code. Le principe de cette méthode est de rajouter une impulsion sur chaque bit systématique et de l'augmenter pas à pas jusqu'à ce que le décodeur se trompe dans la décision.

Une fonction *"methode_impulsion"* a été implémentée. Cette fonction prend cinq arguments en entrée : le treillis de l'encodeur, son état initial, le booléen *closed* et :

— *d_0* : le pas avec lequel l'impulsion est augmentée.

— *d_1* : la valeur maximale que peut atteindre cette impulsion.

Le traitement ne se fait que sur les bits systématiques, donc sur les mots de code d'indice pair(impair sur Matlab). En effet, supposons que nous avons encodé une séquence $[u_0,u_1,...,u_{N-1}]$, notre mot de code sera $[c_0,c_1,c_2,...,c_{2N-1}]$, où $c_0=u_0$ et $c_2=u_1$ (bits systématiques). Pour cela, il faudra avoir un encodage systématique, chose qui est assurée par la fonction *"poly2trellis"* (*trellis = poly2trellis(2,[3 2],3)* pour un code de notation octale $(2,3)_8$).

Cette fonction renvoie un vecteur A_d constitué de l'ensemble des entiers contenus dans v (sans répétition) et un vecteur d représentant leurs occurences. Ces deux vecteurs de retour ont été utilisés afin de calculer le TEP (Taux d'Erreurs de Paquets) correspondant à chaque valeur du SNR $\frac{Eb}{N0}$ par la formule suivante :

$$TEP = \sum_d \frac{1}{2}.A_d.erfc\left(\sqrt{dR\frac{E_b}{N_0}}\right)$$

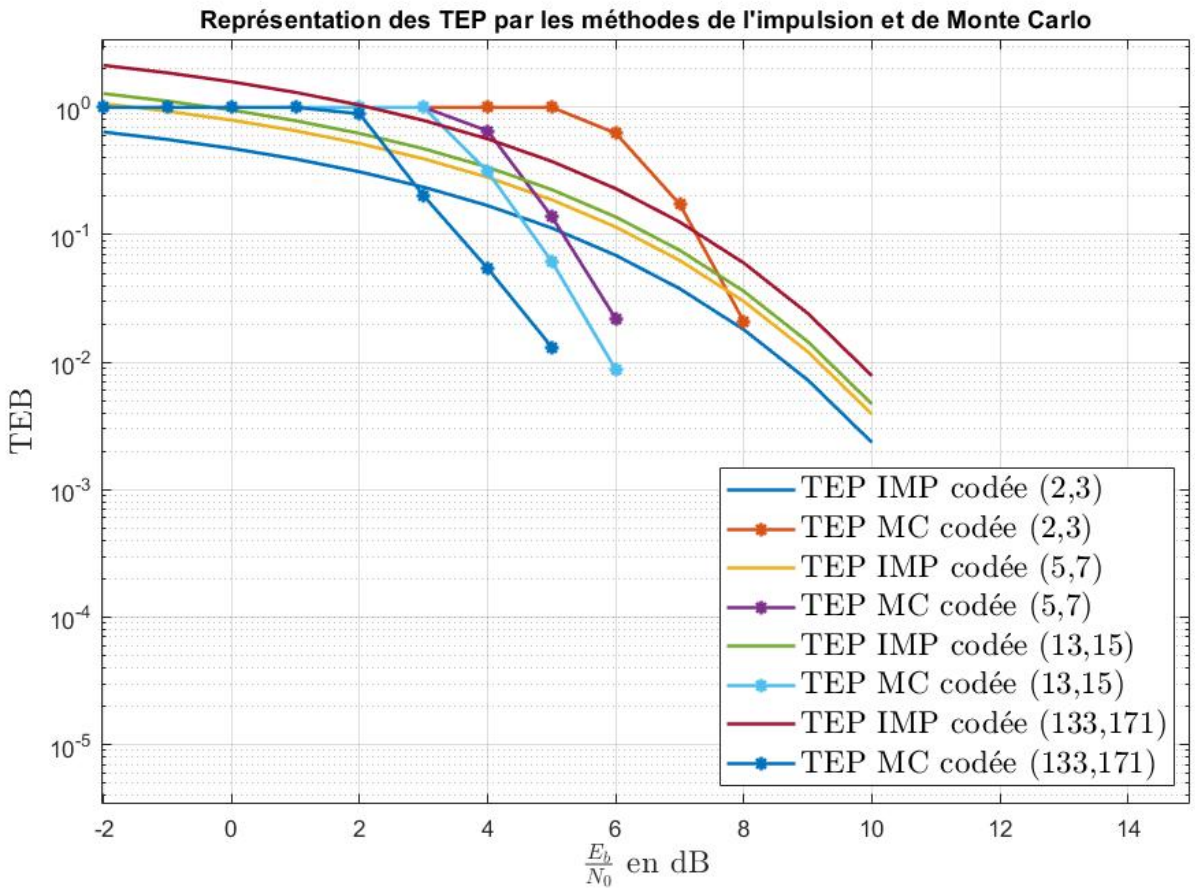

FIGURE 7 – Courbe comparative des TEP obtenus par les méthodes de l'impulsion et de Monte Carlo

$\Longrightarrow$ Les courbes de la la Figure 7 montrent qu'avec la méthode de l'impulsion, plus la mémoire est grande, plus le TEP augmente .Ce résultat est illogique, vu qu'il contredit ce que nous avons démontré auparavant : le code doit gagner en performance quand sa mémoire est importante. Il pourrait donc avoir des erreurs dans notre implémentation de l'algorithme de la méthode d'impulsion.

Nous avons aussi remarqué que le temps d'exécution du code cette méthode est relativement faible par rappport à celui de la méthode de Monte Carlo (quelques minutes contre des heures pour le code $(133,171)_8$).

3 Conclusion

Ce TP de codage de canal a été une bonne opportunité pour mettre en pratique les différentes notions théoriques vues en cours, mais aussi pour approfondir l'étude des blocs de la chaîne de communications numériques, améliorant ainsi la qualité de la transmission.

Nous avons tout d'abord vu la notion d'encodage qui permet de renvoyer le mot de code émis à partir du message envoyé en lui rajoutant de la redondance, pour le décoder ensuite au niveau du récepteur en utilisant l'algorithme de Viterbi. Le tracé des courbes du TEB des différents codes convolutifs nous a permis de conclure que la mémoire du code est un facteur important qui détermine sa performance : plus la mémoire est grande, plus le TEB diminue, et le code sera alors en mesure de détecter et de corriger plus d'erreurs.

Nous avons ensuite abordé la thématique de la performance du code par la méthode de l'impulsion au lieu de simulations de Monte Carlo, qui permet de déterminer le spectre de distance d'un code. Le fait d'étudier les performances des différents codes convolutifs par deux méthodes nous a permis d'avoir une vue plus globale sur le sujet.

Références

[1] (A. Migan), S. Argentieri (2010-2011) *Transmission de l'Information : Les Codes Convolutifs*, `https://slideplayer.fr/slide/1136401/`