Telecommunications

TS226

Rapport

TP de codage canal module TS226 – année 2021/2022

Etudiants :
Baudry Alexandre
(abaudry10@gmail.com)
Benjemaa Rayan
(rayan.benjemaa@gmail.com)

Professeurs :
M. Ellouze
R. Tajan

29 novembre 2021

Table des matières

1 Présentation du TP

Dans ce TP, On s'intéresse à différentes chaînes de communication numériques et l'on cherchera à comparer leurs performances selon plusieurs critères : la probabilité d'erreur binaire, le rendement, mais également le débit. Nous allons pour cela dans un premier temps implémenter sur *Matlab* notre chaîne de transmission codée à l'aide d'un code convolutif puis nous étudierons l'impact de la mémoire du code sur ses performances. On s'intéressera en particulier à 4 codes convolutifs :

— $(2, 3)_8$
— $(5, 7)_8$
— $(13, 15)_8$
— $(133, 171)_8$

Enfin dans une dernière partie nous verrons comment déterminer les performances des codes convolutifs sans avoir à réaliser de longues simulations de Monte Carlo. notamment à l'aide de la méthode de l'impulsion.

2 Codes convolutifs

Dans tout ce TP on considérera la chaîne de transmission suivante :

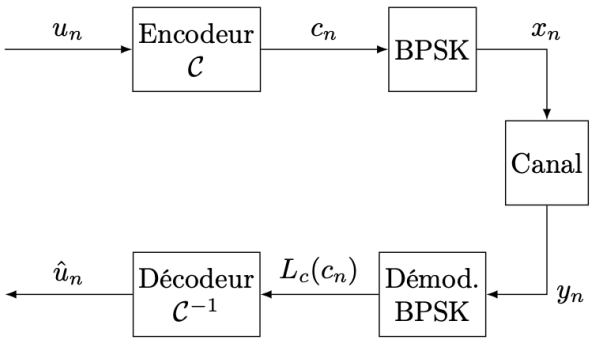

FIGURE 1 – Chaîne de transmission codée

Nous avons alors implémenté les fonctions d'encodage et de décodage des codes convolutifs à l'aide d'une structure donnée par la toolbox de *Matlab*. On construit cette structure à partir de la fonction *poly2trellis*.

2.1 Treillis

L'exemple donné dans le sujet est la structure renvoyée par la fonction pour un code convolutif $(5, 7)_8$. Les champs renvoyés par la structure sont alors :
numInputSymbols : 2
numOutputSymbols : 4
numStates : 4
Mais également deux champs très importants à savoir, *nextStates* un tableau représentant la

fonction suivante $s_{n+1} = f(s_n, u_n)$ avec s_n l'état à l'instant n et u_n le n^{ieme} symbole en entrée en décimal.

État \ Entrée	0	1
0	0	2
1	0	2
2	1	3
3	1	3

FIGURE 2 – Tableau des états suivants à partir des états précedents et du symbole d'entré

Ainsi sur cet exemple si nous sommes dans l'état 2 soit 10 en binaire et que le symbole d'entrée est 1 l'état suivant sera 3 soit 11 en binaire. De la même façon, la structure nous donne également accès à un champ *outputs* : tableau représentant la fonction suivante $c_n = g(s_n, u_n)$ avec c_n le mot de code de sortie à l'instant n.

État \ Entrée	0	1
0	0	3
1	3	0
2	1	2
3	2	1

FIGURE 3 – Tableau des sorties à partir des états courants et des symboles en entrée.

Ainsi ici si nous sommes dans un état 3 et que le smybole d'entrée est 1 la sortie est 2 soit 10 en binaire.

2.2 Encodage (Matlab)

Dans un un premier temps nous avons donc implémenté la fonction d'encodage d'un vecteur u de K symboles selon le prototype suivant :

function [c, s_f] = cc_encode(u, trellis, s_i, closed).

La structure treillis était donc précédemment définie comme expliquée dans la partie précédente. s_i correspondant à l'état initial que nous avons choisi nul par défaut et *closed* une variable valant 0 ou 1 selon si l'on voulait ou non que notre treillis soit fermé. Pour implémenter cette fonction, nous avons tout d'abord traité le cas où le treillis était ouvert. Pour cela on parcourt l'ensemble du vecteur d'entrée u. Puis l'on vérifie pour chaque symbole possible de sortie si la valeur de sortie lui correspond et dans ce cas ci on convertit la sortie en binaire avec le bon nombre de bits à l'aide de la fonction *dec2bin* dont le deuxième argument permet de décider du nombre de bit voulu. Cependant cette fonction nous renvoie une chaîne de caractères et non une

suite de bits, il a donc fallu convertir chacun des bits de la chaîne de caractère grâce à la fonction $str2num$ puis les ajouter au mot de code de sortie à l'aide d'une boucle for. Puis il fallait mettre à jour l'état selon l'entrée et ainsi de suite pour tous les symboles de notre vecteur u d'entrée. On obtient alors ainsi notre encodage adapté au treillis dans le cas où celui-ci est ouvert.

Pour la gestion de la fermeture il fallait s'assurer que notre état final s_f etait non nul (sinon la fermeture est inutile) et dans le cas où celui ci était non nul il fallait choisir la sortie associée à l'état le plus petit jusqu'à ce que le trellis soit fermé nous avons utilisé le même principe de concatenation que précédemment à l'aide des fonctions $dec2bin$ et $str2num$.

Cet encodeur à été implémenté de sorte qu'il puisse encoder n'importe quel mot de code peu importe le nombre de sorties. Particularité qui n'a pas été traitée dans le décodeur qui ne fonctionne que pour deux sorties.

2.3 Décodage de Viterbi (Matlab)

Après avoir codé la fonction d'encodage nous avons codé la fonction de décodage selon le prototype suivant :
function u = viterbi_decode(y, trellis, s_i, closed)
La fonction prend en entrée un signal modulé par une BPSK y ainsi que les mêmes arguments que la fonction d'encodage, et celle-ci renvoie le mot de code décodé.

L'algorithme de Viterbi est une méthode qui permet d'estimer la séquence émise, ici u, en mesurant l'écart du chemin associé aux symboles reçus par rapport à chaque chemin possible, on appelle ces écarts les métriques. Le chemin ayant le plus faible écart sera considéré comme le plus représentatif de la séquence émise réellement et permettra de la reconstituer. Cependant plutôt que de calculer toutes les métriques pour chaque combinaison d'états différents puis d'observer la combinaison ayant la métrique totale la plus faible on peut considérablement réduire le nombre de calcul en ne conservant, à chaque fois que deux états différents à un instant n de la séquence mènent à un même état à l'instant suivant, celui dont la métrique est la plus faible.

On initialise alors matrice de métriques qui listera pour chaque état la valeur de la plus petite somme des métriques qui permettent d'arriver à cet état. Cette matrice est initialisée avec des valeurs de 1000 afin que l'on puisse distinguer lors du remplissage de la matrice le cas où la case est vide (l'état n'a pas encore été atteint) ainsi que le cas où celle-ci est remplie (l'état à déjà été atteint par un autre chemin) et faciliter les comparaisons. On initialise également une matrice des états de même taille qui conservera elle la liste des états associés aux métriques calculées.

On remplit ces matrices en parcourant pour chaque symbole de y et en calculant pour chaque état la métrique associée (à l'aide d'un produit scalaire), dans la phase d'ouverture certains états ne seront pas encore atteins et dans ce cas on remplit directement la matrice avec la valeur de la métrique correspondante sinon comme expliqué précédemment on conserve la métrique, et son chemin associé, la plus faible.

Une fois la matrice de métrique remplie on détermine la liste dont la métrique finale est la

plus faible, c'est à cette métrique qu'on associe le chemin optimal. Il faut alors à présent parcourir le chemin des états associés à l'envers afin de retracer ce chemin optimal. A partir de ce chemin optimal on estime le mot de code initial en concaténant des 0 ou des 1 selon si l'entrée doit être un 1 ou un 0 pour passer d'une case du tableau du chemin optimal à une autre.

Dans le cas d'une fermeture du treillis on retire simplement les bits de fermeture (log2(trellis.numStates)).

3 Étude de l'impact de la mémoire du code

3.1 Etudes des courbes de TEB

Une fois notre fonction de décodage de Viterbi implémentée nous avons pu afficher les courbes des taux d'erreur binaire (TEB) ainsi que les TEP en fonction de $\frac{E_b}{N_0}$ pour un treillis ouvert.

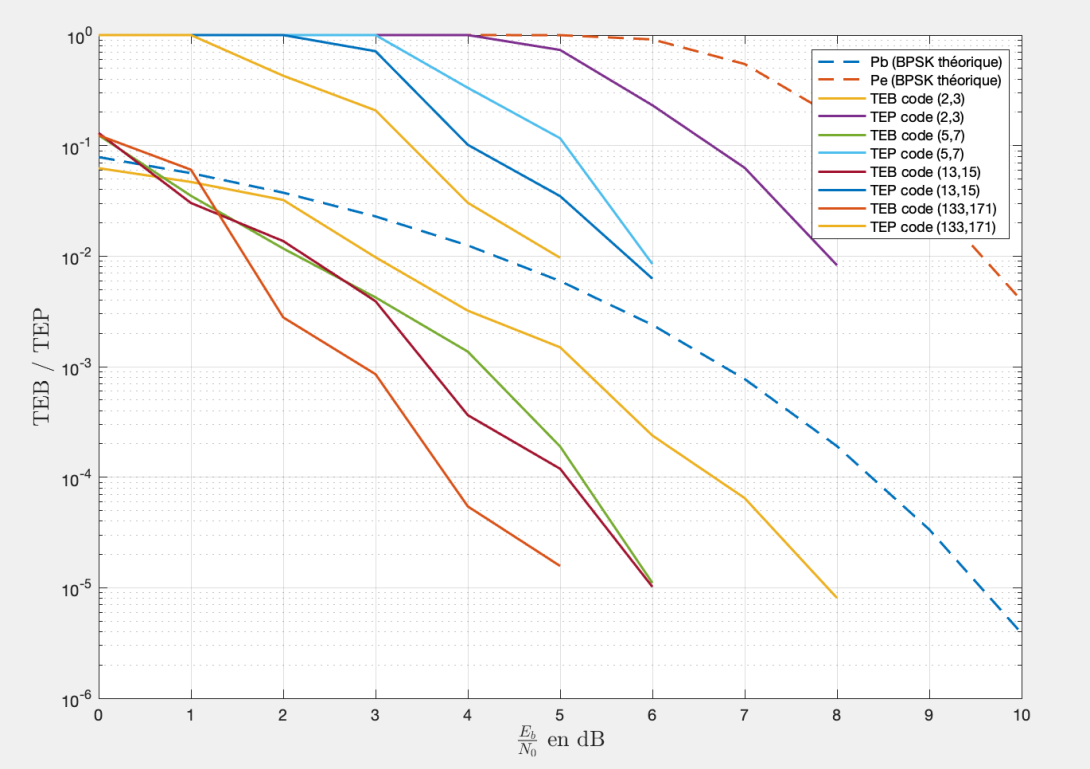

FIGURE 4 – taux d'erreur binaire (TEB) ainsi que les TEP en fonction de $\frac{E_b}{N_0}$. pour un treillis ouvert

De même pour un treillis fermé avec l'exemple du code convolutif $(2, 3)_8$:

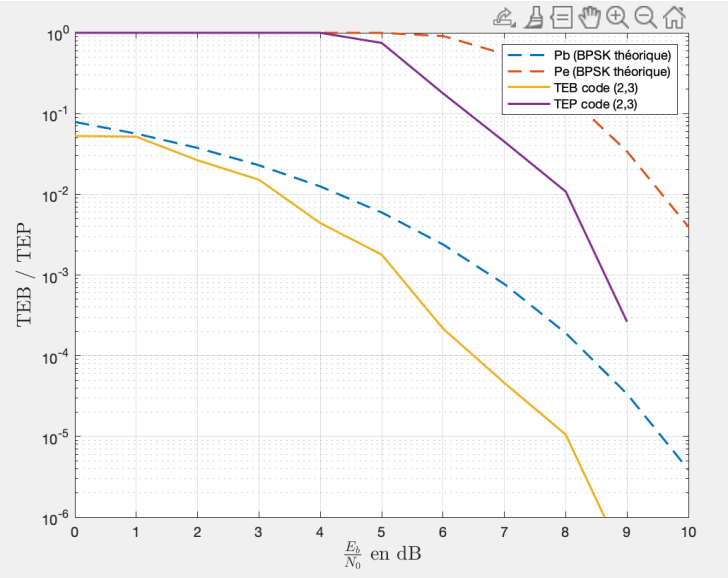

FIGURE 5 – taux d'erreur binaire (TEB) ainsi que les TEP en fonction de $\frac{E_b}{N_0}$. pour un treillis fermé

Notre algorithme a néanmoins ses limites, notamment dans sa faible vitesse d'éxécution liée à l'utilisation de la fonction *str2num* appelée un grand nombre de fois au cours de la boucle while.

3.2 Tableau de Synthèse Récapitulatif

Nous avons réalisé un tableau de synthèse récapitulatif explicitant pour chaque code leurs différentes caractéristiques :

Code	Rendement	Mémoire	Nbr Etats	Distance Minimale	Gain
$(2,3)_8$	0.5	1	2	3	1.6dB
$(5,7)_8$	0.5	2	4	5	3.6dB
$(13,15)_8$	0.5	3	8	6	3.7dB
$(133,171)_8$	0.5	6	64	8	4.6dB

Nous allons voir par la suite, que la donnée majeure qui influence toutes les autres est la mémoire (le rendement étant fixé).

3.3 Distance minimale

La distance minimale d'un code est la plus petite distance de Hamming entre 2 mots de code. Pour pouvoir l'analyser, nous nous sommes aidés de la fonction MATLAB : distspec qui permet de calculer le spectre de distance d'un code convolutif. Cette fonction renvoie une structure, dans laquelle on retrouve le champ SPECT.DFREE qui représente la distance minimale. Une fois les résultats obtenus, nous avons remarqué une corrélation entre la distance minimale et la mémoire

du code. On remarque que lorsque la mémoire augmente la distance minimale augmente. En reprenant la définition de la distance minimale et les impacts de la mémoire sur le code on peut interpréter cette relation. En effet, une grande complexité est synonyme d'une distance minimale importante.

3.4 Gain de codage

Le puissance de correction d'un code convolutif est définie par son gain de codage par rapport à un système non codé. On remarque que le type de décodage influe sur le gain de codage. Le gain le plus important est obtenu pour le code $(133, 171)_8$ ayant une mémoire de 6. En revanche, le gain le plus faible est obtenu pour le code $(2, 3)_8$ ayant une mémoire de 1. Cette différence entre les codes peut être expliquée par la différence de mémoire.
Effectivement, lorsque la mémoire augmente le nombre de bits erronés diminue ce qui favorise le gain de codage. En effet, on a pu voir tout au long de notre apprentissage l'impact d'une erreur sur le message et son interprétation. En ajoutant une notion de redondance dans le message transmis, les codes convolutifs avec une mémoire importante sont plus complexes et par conséquent plus fiables.

3.5 Débit de décodage

Précédemment, on a pu mettre en évidence une relation entre le gain de codage et la mémoire du code. Cette relation s'expliquait par la complexité des codes convolutifs à grande mémoire. Cependant, ce même argument va porter préjudice au débit de décodage. En effet, lorsque l'on augmente la mémoire, le débit de décodage est divisé par deux.
On en déduit la relation suivante :
$D_n = \frac{D_1}{2^(n-1)}$, en notant D_n le débit de décodage d'un code de mémoire n.

Cela s'explique par le fait que lorsque l'on incrémente la mémoire on double le nombre d'états et donc le nombre d'opérations dans la construction de notre tableau métrique. Cette relation nous a permis de montrer la corrélation suivante : en augmentant la mémoire on influe sur la complexité et le nombre de calculs du code et donc on réduit le débit de décodage.

3.6 Prédiction des performances

La longueur des simulations de Monte Carlo réalisés jusqu'à maintenant a été un frein important dans l'avancée du projet. Les méthodes de calculs n'étant pas optimisées, certaines réalisations dépassaient le quart d'heure. On nous propose dans cette partie d'implémenter la méthode de l'impulsion afin de réduire le temps de simulation. Malheureusement, nous n'avons pas pu finaliser cet algorithme mais le peu de résultats que l'on a pu voir nous ont montré l'importance d'associer ce type d'algorithme à la visualisation des performances des codes convolutifs.

4 Conclusion

Pour conclure, ce TP nous aura familiarisé avec l'exploitation des treillis via l'utilisation de la structure MATLAB *trellis*. Nous avons pu réellement nous appoprier le fonctionnement de l'encodage et du décodage des codes convolutifs. Ce TP nous a demandé beaucoup de rigueur dans l'implémentation des fonctions. En effet, la majorité de nos problèmes dans le code se sont avérés être des erreurs minimes de dimension ou d'indiçage plus que des erreurs de compréhension du principe général. De plus, ce TP a mis en lumière l'importance de la mémoire sur les autres caracteristiques du code. Neanmoins, bien que fonctionnels nos codes d'encodage et de décodage necessitent un temps d'execution important qui aurait surement pu être réduit avec une implémentation différente. Toutefois, ces temps de simulation particulièrement longs nous auront inculqué la patience, qualité de plus en plus rare chez les jeunes ingénieurs.