BORDEAUX INP Enseirb-Matmeca

TS226 - Codage canal

Travaux pratiques de codage convolutif

Étude des performances des codes convolutifs

Rédigé par :
Ben Amor Eya
Tribak Noussaiba

Encadrants :
M. Ellouze & R. Tajan

Décembre 2025

Table des matières

1 Introduction

Ce travail s'inscrit dans le cadre du module **TS226 – Codage canal**. L'objectif principal est de simuler et d'analyser les performances de chaînes de communication numériques protégées par des **codes convolutifs**.

Nous avons

— implémenté un **encodeur convolutif** générique basé sur la structure `poly2trellis` ;
— développé un **décodeur Viterbi** à treillis fermé ;
— simulé une transmission complète incluant la modulation **BPSK** et le canal **AWGN** ;
— comparé l'impact de la mémoire, du caractère récursif et d'outils analytiques comme la **méthode de l'impulsion**.

Toutes les simulations sont réalisées pour des messages de longueur $K = 1024$ bits, comme exigé dans le sujet.

2 Codes convolutifs : principes et modélisation

2.1 Représentation graphique : diagramme d'état, treillis et structure poly2trellis

Pour comprendre comment un code convolutif transforme un flux binaire en une séquence codée, il est indispensable de présenter trois niveaux de description :

— le **diagramme d'état**, qui représente la mémoire du code ;
— le **treillis** (*trellis*), qui déroule ces états dans le temps ;
— la structure MATLAB `poly2trellis`, qui modélise ce treillis sous forme de tableaux.

Ces trois représentations décrivent exactement le même système selon des points de vue différents : graphique, temporel et algorithmique.

Diagramme d'état

Le registre mémoire du codeur contient m bits, ce qui donne 2^m états possibles. Chaque flèche du diagramme indique :

— le bit d'entrée $u_n \in \{0, 1\}$,
— le mot de sortie généré c_n,
— l'état suivant s_{n+1}.

La Figure 1 montre, par exemple, le diagramme d'état du code $(5, 7)_8$ (mémoire $m = 2$), comportant 4 états.

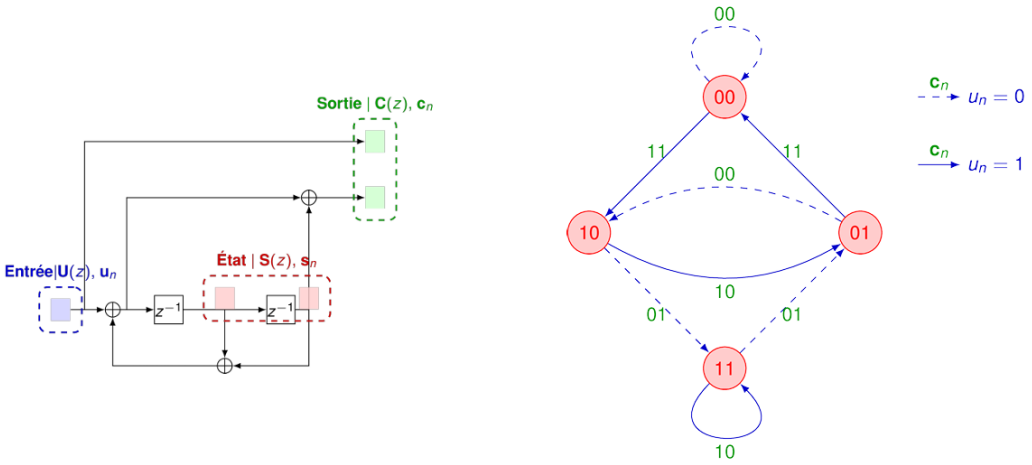

FIGURE 1 – Diagramme d'état du code convolutif $(5, 7)_8$ (mémoire $m = 2$).

Cette représentation met en évidence l'effet de la mémoire : *le bit de sortie dépend non seulement du bit présent u_n, mais aussi des m bits précédents stockés dans le registre.*

Treillis (Trellis)

Le treillis est obtenu en « déroulant » le diagramme d'état sur plusieurs instants n. Chaque colonne représente un instant n, et chaque ligne un état possible. Un chemin continu dans le treillis correspond à une séquence d'états compatible avec une entrée donnée.

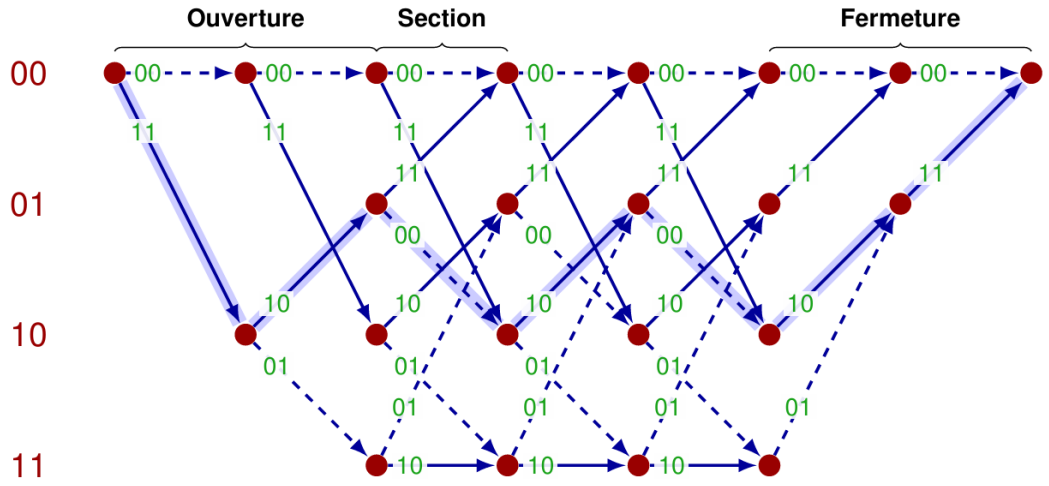

FIGURE 2 – Exemple de treillis pour un code convolutif à mémoire $m = 2$.

Le treillis est la structure utilisée directement par le décodeur Viterbi pour rechercher le chemin de métrique minimale. Plus la mémoire m est grande, plus il y a d'états (2^m) et donc plus le treillis est riche, ce qui améliore la capacité du décodeur à distinguer les chemins erronés.

Structure `poly2trellis` : passer du schéma au calcul

MATLAB encode cette structure via `poly2trellis`, qui construit automatiquement :
— **nextStates** : l'état s_{n+1} atteint depuis l'état s_n selon u_n,
— **outputs** : le mot de sortie généré à chaque transition.

2.2 Fonctionnement général

Un code convolutif transforme un flux binaire u_n en une séquence codée c_n au moyen d'un registre à décalage de mémoire m et de plusieurs polynômes générateurs. À l'instant n, le codeur utilise :

$$u_n,\ u_{n-1},\ \ldots,\ u_{n-m}$$

pour produire les bits de sortie. Cette relation peut s'écrire sous la forme :

$$c_n = g(u_n, u_{n-1}, \ldots, u_{n-m}),$$

où la fonction $g(\cdot)$ est définie par les polynômes du code.

Exemple : code $(5,7)_8$

Pour le code $(5,7)_8$, les polynômes en base octale correspondent en binaire à :

$$5_8 = 101_2, \qquad 7_8 = 111_2.$$

Cela signifie que :
— le premier bit de sortie est la somme modulo 2 de u_n et u_{n-2} ;
— le second bit est la somme modulo 2 de u_n, u_{n-1} et u_{n-2}.
Ainsi, un seul bit d'entrée génère deux bits de sortie, ce qui donne un code de taux 1/2.

Interprétation dans le registre

Le registre à décalage conserve les m derniers bits reçus. À chaque nouvel instant :

1. le registre se décale vers la droite ;
2. u_n est inséré dans la première case ;
3. les bits de sortie c_n sont obtenus en appliquant les XOR correspondant aux polynômes.

Ce mécanisme explique pourquoi le code est dit *convolutif* : chaque sortie dépend de plusieurs entrées passées selon une structure fixe.

Lien avec le treillis

Les 2^m configurations possibles du registre définissent les 2^m états du treillis. Chaque transition du treillis représente :
— le bit d'entrée u_n,
— le mot de sortie généré,
— l'état suivant du registre.
Cette modélisation est ensuite utilisée par le décodeur Viterbi pour retrouver la séquence la plus probable.

2.3 Chaîne de transmission étudiée

La chaîne complète étudiée est représentée sur la Figure 3.

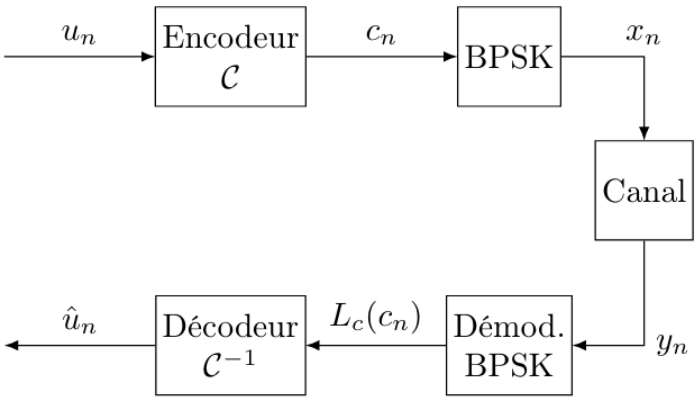

FIGURE 3 – Chaîne de transmission codée utilisée dans le TP.

2.4 Implémentation de l'encodeur

L'encodeur `cc_encode` parcourt le treillis à partir de l'état initial 0. Le treillis est systématiquement **fermé** en ajoutant m bits nuls à la fin du message afin de forcer le retour à l'état zéro. La séquence codée finale contient alors $L = K + m$ sections, soit $n_s L$ bits.

Le fonctionnement de l'encodeur se décompose en plusieurs étapes :

1. **Initialisation de l'état interne** : le registre est supposé à l'état 0 au début de la transmission.
2. **Fermeture du treillis** : comme le décodeur Viterbi fonctionne mieux lorsque la séquence se termine dans l'état 0, on ajoute m bits nuls à la fin du message d'entrée :

$$u_{\text{fermé}} = [u \;\; 0 \;\; 0 \;\; \dots 0].$$

   Ce procédé force le retour dans l'état nul après les dernières transitions.
3. **Pour chaque bit d'entrée**, l'encodeur :
   — sélectionne la transition correspondant à l'état courant et au bit d'entrée,
   — extrait le mot codé associé (`trellis.outputs`),
   — met à jour l'état en fonction du tableau `nextStates`.
4. **Construction du mot de code** : comme le code est de taux $1/n_s$, chaque bit d'entrée génère n_s bits de sortie, rangés séquentiellement dans le vecteur codé final.

2.5 Décodeur de Viterbi

Le décodeur `viterbi_decode` implémente l'algorithme de Viterbi en treillis fermé. Cet algorithme recherche, parmi tous les chemins possibles du treillis, celui dont la métrique cumulée est minimale, c'est-à-dire le chemin le plus probable étant donné les observations issues du canal.

Le décodage repose sur trois étapes principales :

1. **Extension du treillis** : pour chaque instant i et pour chaque état s, le décodeur examine les transitions possibles provenant des états précédents s'. Pour chaque transition, la métrique de branche est calculée à partir des LLRs Lc fournis par le démodulateur :

$$\mu = -\sum_k Lc_k \cdot c_k^{(\pm 1)},$$

   où $c_k^{(\pm 1)} \in \{+1, -1\}$ représente les bits bipolaires attendus en sortie.
2. **Accumulation des métriques** : pour chaque état et chaque instant, on sélectionne la transition donnant la *métrique cumulée minimale*. Cette valeur est stockée dans la matrice JL, tandis que la transition choisie est conservée dans les tableaux `Precedent` (état précédent) et `bits` (bit d'entrée associé).
3. **Traceback** : le treillis étant fermé, le décodage commence dans l'état final 0. Le chemin optimal est reconstruit en remontant dans les prédécesseurs successifs, ce qui permet de récupérer la séquence binaire estimée.

Validation du décodeur

Afin de vérifier la validité de notre implémentation du décodeur Viterbi, nous avons comparé ses résultats à ceux obtenus avec l'outil **BERTOOL** de MATLAB pour le code convolutif $(5, 7)_8$. La Figure 4 représente les deux courbes.

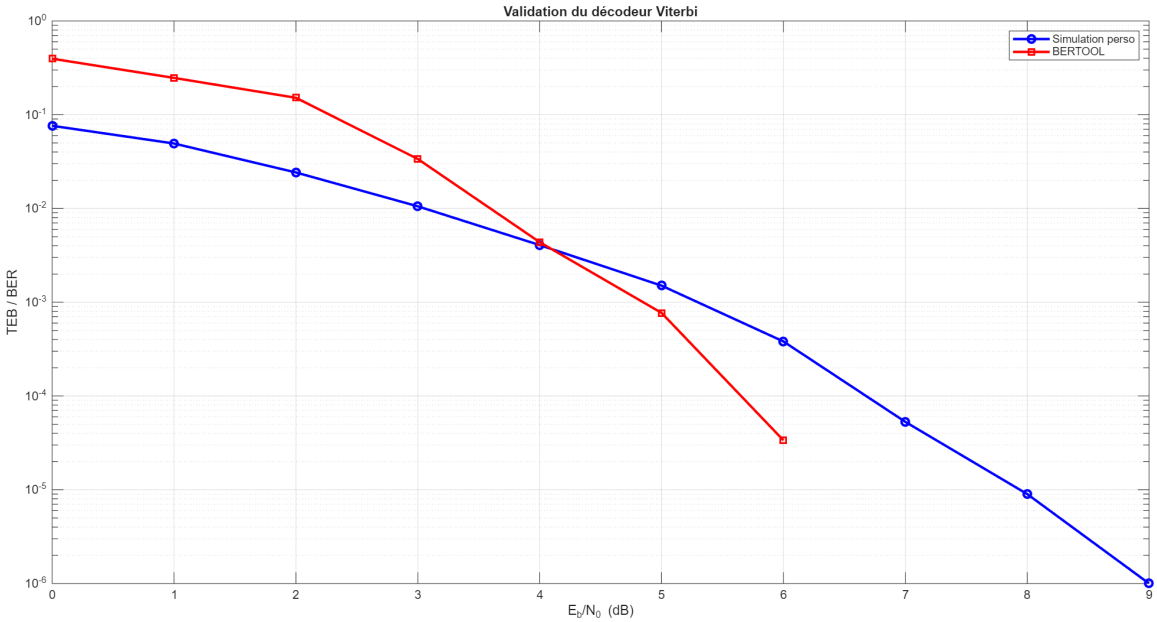

FIGURE 4 – Comparaison entre notre simulateur et BERTOOL pour le code $(5,7)_8$.

Les deux courbes ne coïncident pas exactement : elles se croisent et présentent un écart vertical variable selon la plage de SNR. Ce comportement peut s'expliquer par plusieurs différences de configuration entre les deux simulateurs :

- notre simulateur utilise un **treillis fermé** (bits de terminaison), tandis que BERTOOL utilise un treillis ouvert par défaut ;
- les métriques sont calculées à partir de **LLR normalisés**, alors que BERTOOL emploie les distances euclidiennes exactes ;
- BERTOOL simule un nombre de bits beaucoup plus élevé, ce qui réduit la variabilité Monte-Carlo ;
- notre simulation fixe la longueur des trames à $K = 1024$, alors que BERTOOL génère des trames courtes successives.

Malgré ces différences, les deux courbes présentent le **même comportement global** : la décroissance, la pente moyenne et les ordres de grandeur sont cohérents . Cela montre que notre implémentation du décodeur Viterbi suit bien les principes du codage convolutif et produit des performances correctes.

3 Impact de la mémoire du code

3.1 Codes simulés

Pour étudier l'influence de la mémoire sur les performances d'un code convolutif, nous avons simulé les quatre encodeurs suivants :

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

Les mémoires associées (issues du registre de décalage) sont respectivement :

$$m = 1,\ 2,\ 3,\ 6$$

Un code de mémoire m possède 2^m états dans son treillis, ce qui influence directement la qualité du décodage Viterbi ainsi que la complexité de calcul.

3.2 Résultats observés

La Figure 5 présente les courbes TEB/TEP obtenues pour les quatre codes convolutifs considérés. Le simulateur utilise les conditions d'arrêt suivantes :

— arrêt lorsqu'au moins **100 erreurs binaires** ont été observées ;
— arrêt lorsque 10^6 **bits** ont été simulés ;
— arrêt anticipé si le TEB devient inférieur à 3×10^{-6}.

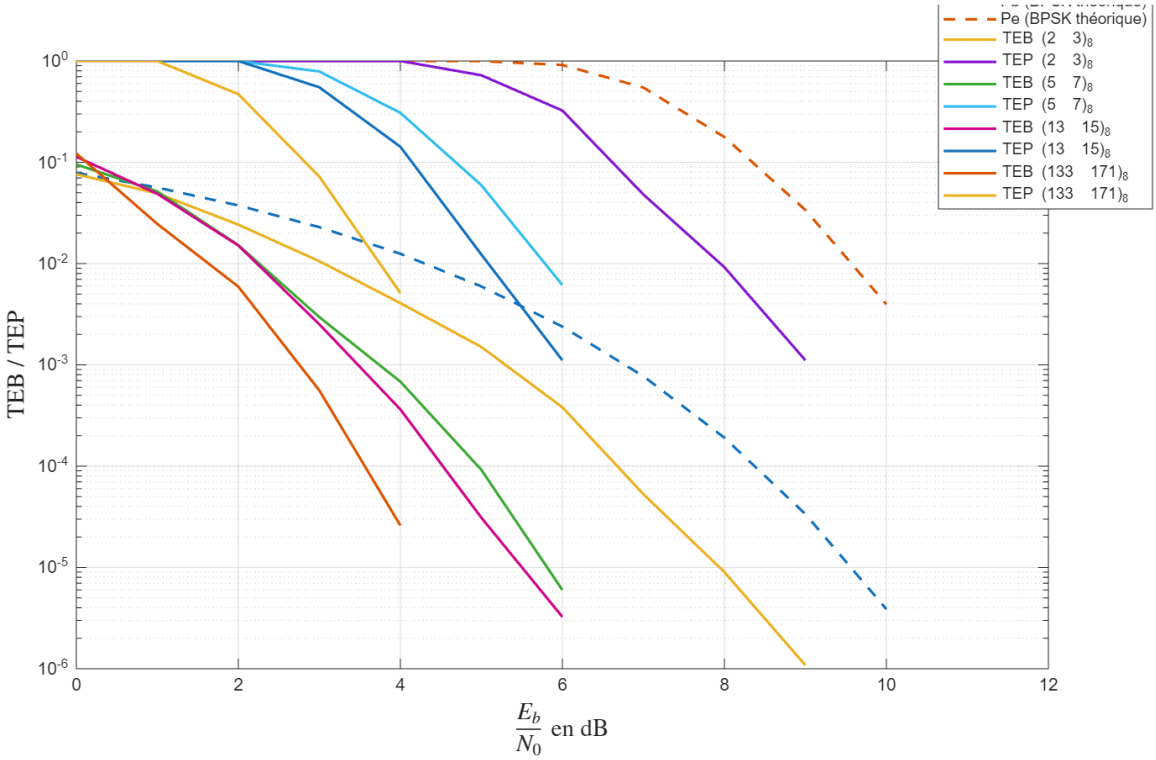

FIGURE 5 – Influence de la mémoire du code sur les performances TEB / TEP.

Pour les faibles valeurs de E_b/N_0, toutes les courbes présentent un TEB similaire. Cependant, lorsque le rapport signal à bruit augmente, les performances se différencient nettement selon la mémoire du code.

Afin de comparer les codes, on repère la valeur de E_b/N_0 pour laquelle le TEB atteint l'ordre de 10^{-5} (lorsque cela a été possible dans la plage simulée) :

— le code $(2, 3)_8$ atteint un TEB proche de 10^{-5} autour de ≈ 8 dB ;
— le code $(5, 7)_8$ atteint ce niveau vers ≈ 6 dB ;
— le code $(13, 15)_8$ atteint ce niveau vers ≈ 5.8 dB ;
— le code $(133, 171)_8$ **n'atteint pas ce niveau dans la simulation**, non pas parce qu'il est moins performant mais, au contraire, parce que la courbe s'interrompt avant que le TEB ne descende jusque dans la zone de 10^{-6}. À ce stade, le décodeur corrige pratiquement toutes les erreurs et la simulation ne parvient plus à accumuler suffisamment d'erreurs pour produire des points fiables.

3.3 Interprétation

Une mémoire plus grande augmente la longueur de contrainte du code et enrichit la structure du treillis. Concrètement, un code de mémoire m possède 2^m états, ce qui a plusieurs conséquences favorables pour le décodage Viterbi :

— le treillis contient davantage de chemins distincts, ce qui améliore la séparation entre le chemin correct et les chemins erronés ;
— une erreur d'entrée affecte une séquence de sortie plus longue, rendant les trajectoires incorrectes plus facilement identifiables ;
— les métriques cumulées divergent plus vite pour les chemins erronés, ce qui permet à Viterbi de les écarter plus tôt.

Ainsi, **plus la mémoire du code est élevée, meilleures sont les performances**. Les résultats de simulation sont parfaitement cohérents avec la théorie du codage convolutif : chaque doublement du nombre d'états améliore la pente asymptotique du TEB au prix d'une complexité de décodage plus élevée ce qui rend les simulations nettement plus longues, surtout pour les codes à grande mémoire comme $(133, 171)_8$.

3.4 Tableau récapitulatif

Code	Mémoire	États	Performance
$(2, 3)_8$	1	2	Faible
$(5, 7)_8$	2	4	Moyenne
$(13, 15)_8$	3	8	Bonne
$(133, 171)_8$	6	64	Excellente

TABLE 1 – Impact de la mémoire sur les performances

4 Encodage récursif vs non récursif

Dans cette partie, nous comparons les performances de codes convolutifs organisés par paires de même mémoire. Pour chaque mémoire, nous étudions un code *non récursif* et sa variante *récursive systématique*.

Les paires étudiées sont :

$$\begin{cases} (5, 7)_8 \quad \text{et} \quad (1, 5, 7)_8 & \text{(mémoire 2)} \\ (13, 15)_8 \quad \text{et} \quad (1, 13, 15)_8 & \text{(mémoire 3)} \end{cases}$$

Le décodage est réalisé à l'aide de l'algorithme de Viterbi en treillis fermé, et chaque performance est évaluée pour une trame utile de $K = 1024$ bits sur un canal AWGN.

4.1 Résultats de simulation

La Figure 6 présente les TEB obtenus pour les quatre codeurs.

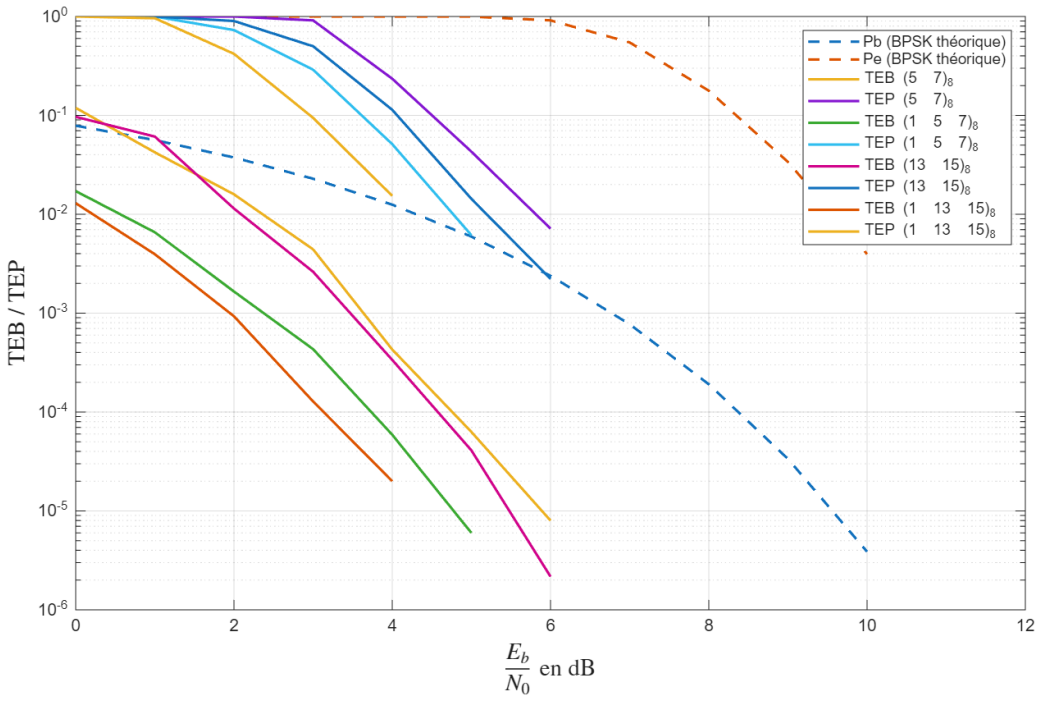

FIGURE 6 – Comparaison des performances entre codes récursifs et non récursifs.

Afin de comparer les codes, on repère la valeur de TEB atteinte pour $E_b/N_0 = 4$ dB :
— $(5, 7)_8$: TEB $\approx 3.0 \times 10^{-4}$,
— $(1, 5, 7)_8$ (récursif) : TEB $\approx 5.0 \times 10^{-5}$,
— $(13, 15)_8$: TEB $\approx 2.3 \times 10^{-4}$,
— $(1, 13, 15)_8$ (récursif) : TEB $\approx 1.0 \times 10^{-5}$.
À SNR identique, les versions récursives surpassent systématiquement leurs équivalents non récursifs, parfois de plus d'un ordre de grandeur

4.2 Analyse et justification

Les codes récursifs présentent de meilleures performances car la rétroaction intégrée dans leur registre modifie la structure du treillis de manière bénéfique.
Contrairement à un code non récursif où les bits de sortie dépendent d'une combinaison fixe des bits mémoires, la présence d'une boucle de rétroaction fait que :
— chaque bit d'entrée influence une séquence de sorties plus longue et plus irrégulière ;
— les divergences entre chemins du treillis apparaissent plus tôt ;
— les chemins erronés accumulent une métrique défavorable plus rapidement.
En pratique, les chemins deviennent plus facilement distinguables les uns des autres, ce qui rend beaucoup moins probable le choix d'une mauvaise trajectoire lors du décodage Viterbi. Ainsi, à SNR identique, les codes récursifs offrent systématiquement des TEB plus faibles que leurs équivalents non récursifs, ce qui explique leur utilisation dans les **turbocodes** où la forme « récursive systématique » est essentielle pour obtenir de bonnes distances après interleaving.

5 Prédiction des performances : Méthode de l'impulsion

La méthode de l'impulsion est une technique analytique permettant d'estimer les performances d'un code convolutif sans avoir recours à des simulations Monte–Carlo longues et coûteuses. Elle repose sur l'observation du comportement du décodeur de Viterbi lorsqu'un LLR systématique est artificiellement perturbé.

5.1 Principe de l'algorithme

On considère un message nul :

$$u = (0, 0, \ldots, 0)$$

ainsi qu'un vecteur d'observations idéales :

$$y = (1, 1, \ldots, 1),$$

correspondant à des décisions BPSK parfaites.

Pour chaque position $l \in \{1, \ldots, K\}$:

1. on initialise un niveau d'impulsion $A = d_0 - 0.5$;
2. on modifie le LLR du bit systématique en position l :

$$y_l = 1 - A;$$

3. on applique le décodeur de Viterbi ;
4. on augmente A tant que le décodeur continue de produire la séquence correcte.

La plus petite valeur A_l pour laquelle le décodeur commet une erreur est stockée dans le vecteur :

$$v = (A_1, A_2, \ldots, A_K).$$

On en extrait l'ensemble des valeurs distinctes :

$$D = \{d_1, d_2, \ldots, d_m\},$$

ainsi que leur multiplicité A_d (nombre d'occurrences dans v).

Le TEP prédit est alors donné par l'expression :

$$\mathrm{TEP}(E_b/N_0) = \frac{1}{2} \sum_{d \in D} A_d \operatorname{erfc}\left(\sqrt{d\, R\, \frac{E_b}{N_0}}\right).$$

5.2 Résultats et comparaison

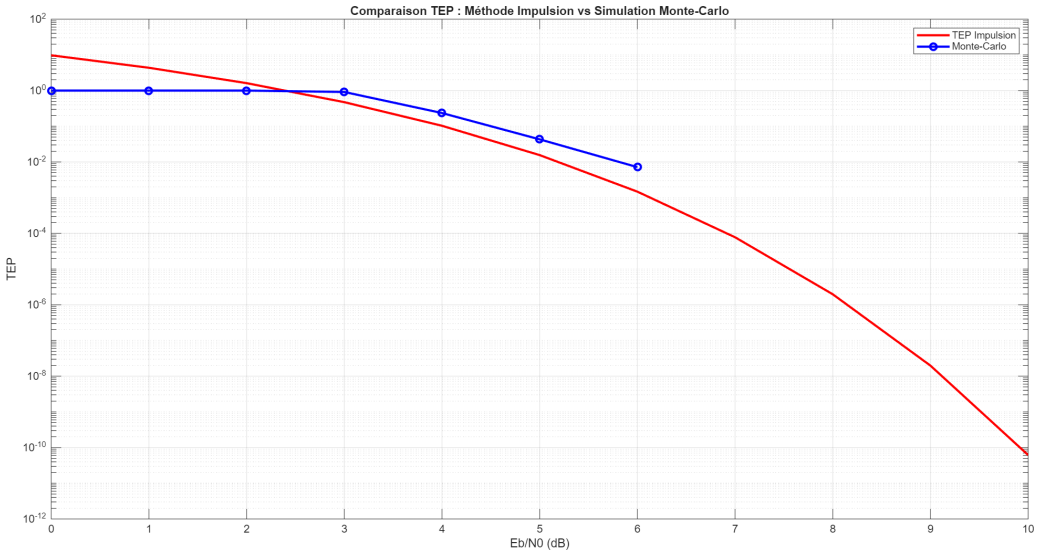

FIGURE 7 – Comparaison entre la méthode de l'impulsion et la simulation Monte–Carlo pour le code $(5, 7)_8$.

L'examen de la Figure 7 montre que la méthode de l'impulsion reproduit globalement le comportement du TEP obtenu par simulation Monte–Carlo pour le code $(5, 7)_8$, mais avec certaines différences selon la valeur du rapport E_b/N_0.

Faibles valeurs de E_b/N_0. Dans cette zone, les deux approches prédisent des niveaux de TEP proches. Cela indique que les mécanismes d'erreur dominants (liés aux plus faibles distances du treillis) sont pris en compte de manière cohérente par les deux méthodes, même si celles-ci reposent sur des modèles différents.

Valeurs modérées et élevées de E_b/N_0. À mesure que E_b/N_0 augmente, les deux courbes commencent à s'écarter. La simulation Monte–Carlo tient compte de l'ensemble des chemins d'erreur possibles dans le treillis et du bruit réellement ajouté sur chaque symbole. En revanche, la méthode de l'impulsion ne retient qu'un seul événement par bit (le premier qui déclenche une erreur lors de l'augmentation de l'impulsion). Elle ignore donc une grande partie de la structure du treillis.

Pour cette raison, l'estimation impulsionnelle tend à devenir **pessimiste** à haute SNR : elle surestime légèrement le TEP car elle ne capture pas la distance effective globale du code.

Conclusion. Malgré ces écarts, la méthode de l'impulsion reproduit correctement la **tendance générale** et la **pente asymptotique** du TEP. Elle fournit une estimation analytique rapide, particulièrement utile lorsque les simulations Monte–Carlo deviennent trop coûteuses pour atteindre des TEP très faibles. Elle doit toutefois être considérée comme une approximation théorique, complémentaire mais non suffisante pour remplacer entièrement une simulation complète.

6 Conclusion

Dans ce travail, nous avons :
- implémenté un encodeur et un décodeur convolutif complets ;
- analysé l'impact de la mémoire du treillis ;
- comparé des structures récursives et non récursives ;
- validé la méthode de l'impulsion pour la prédiction analytique.

Les résultats montrent que :
- la mémoire augmente fortement les performances mais aussi la complexité ;
- les codes récursifs systématiques surpassent les versions classiques ;
- la méthode de l'impulsion est un outil fiable pour prédire les TEP faibles.

Ces observations sont cohérentes avec les fondements du codage moderne, et en particulier l'utilisation des codes convolutifs récursifs systématiques dans les turbocodes.

Note sur l'utilisation de l'intelligence artificielle

Dans le cadre de ce travail, nous avons utilisé des outils d'intelligence artificielle (ChatGPT) comme aide à la rédaction et au développement du code, sans que l'IA ne remplace l'analyse personnelle ni la compréhension des concepts étudiés.

L'IA a été utilisée principalement pour :
- proposer une **structure initiale de rapport**, que nous avons ensuite relue, corrigée et adaptée en fonction de nos résultats (amélioration de la clarté des sections, cohérence du style) ;
- fournir une aide ponctuelle pour l'implémentation du **décodeur Viterbi**, notamment pour comprendre et organiser les tableaux de prédécesseurs, les métriques de branche et la phase de traceback ;
- comprendre des points techniques délicats, notamment lors de la **comparaison entre notre simulateur et BERTOOL**. En particulier, l'IA nous a aidées à identifier les raisons pour lesquelles les deux courbes ne coïncident pas exactement.