Optimisation des transmissions numériques grâce aux codes correcteurs d'erreurs

Mohammed Amine Mazouz

Épreuve de TIPE

Session 2024-2025

N° Candidat SCEI : 10518

Plan :

1 Introduction
2 Les erreurs de transmission dans un canal bruité
3 Les codes correcteurs d'erreurs
4 Code de Hamming
5 Codes Reed-Solomon
6 Mise en œuvre expérimentale
7 Bilan des performances
8 Conclusion
9 Annexe

Définitions fondamentales

Mot sur un corps fini

Soit $\mathbb{F}_q$ un corps fini à q éléments. Un **mot** est un vecteur $\underline{x} = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$, représentant un message à transmettre.

Distance de Hamming

La **distance de Hamming** entre deux mots $\underline{x}, \underline{y} \in \mathbb{F}_q^n$ est le nombre de positions où ils diffèrent :

$$d_H(\underline{x}, \underline{y}) = \#\left\{i \in \{1, \ldots, n\} \mid x_i \neq y_i\right\}$$

Poids de Hamming

Le **poids de Hamming** d'un mot $\underline{x} \in \mathbb{F}_q^n$ est le nombre de ses composantes non nulles :

$$W_H(\underline{x}) = \#\left\{i \in \{1, \ldots, n\} \mid x_i \neq 0\right\}$$

Code linéaire

Un **code linéaire** de longueur n sur le corps fini $\mathbb{F}_q$ est un sous-espace vectoriel $C \subseteq \mathbb{F}_q^n$ de dimension k. On dit alors que C est un code (n, k).

Définitions fondamentales (suite)

Distance minimale de Hamming

$$d = \min_{\substack{\underline{x},\underline{y}\in\mathcal{C} \\ \underline{x}\neq\underline{y}}} d_H(\underline{x},\underline{y}) = \min_{\substack{\underline{c}\in\mathcal{C} \\ \underline{c}\neq\underline{0}}} W_H(\underline{c})$$

Capacité de correction

Un code de distance minimale d peut corriger tout vecteur d'erreurs $\underline{e}$ de poids au plus

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor.$$

Matrice génératrice

Une matrice $G \in M_{k,n}(\mathbb{F}_q)$ est une **matrice génératrice** de $\mathcal{C}$ si ses lignes forment une base de $\mathcal{C}$. Elle définit l'encodage $x \mapsto xG$ pour $x \in \mathbb{F}_q^k$.

Matrice de contrôle

Le sous-espace orthogonal

$$\mathcal{C}^\perp = \{y \in \mathbb{F}_q^n \mid \forall x \in \mathcal{C}, \langle x, y\rangle = 0\}$$

de dimension $n-k$ admet une **matrice de contrôle** $H \in M_{n-k,n}(\mathbb{F}_q)$ génératrice de $\mathcal{C}^\perp$.

Modélisation du canal de communication

Canal bruité

Considérons un canal de communication discret sur un corps fini $\mathbb{F}_q$.
Un message original est un vecteur :

$$\underline{x} = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$$

Lors de la transmission, une erreur $\underline{e} \in \mathbb{F}_q^n$ peut altérer ce message, ce qui donne un mot reçu :

$$\underline{y} = \underline{x} + \underline{e}$$

L'addition est définie composante par composante dans $\mathbb{F}_q$.

Problématique fondamentale

Question centrale

Lorsqu'un mot $\underline{x}$ est transmis à travers un canal bruité, il peut être altéré en $\underline{y} = \underline{x} + \underline{e}$, où $\underline{e}$ est un vecteur d'erreurs inconnu.

Problème : peut-on retrouver le message original $\underline{x}$ à partir du mot reçu $\underline{y}$, sans connaître $\underline{e}$, à condition que l'erreur soit « petite » ?

$$\underline{x} \in \mathcal{C}, \quad \underline{y} = \underline{x} + \underline{e} \quad \Longrightarrow \quad \hat{x} = \underline{x}$$

Codes correcteurs : définition

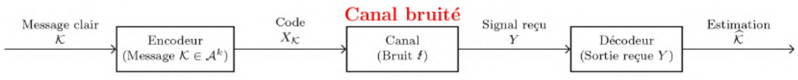
Définition

Un **code correcteur** est un sous-ensemble $\mathcal{C} \subset \mathbb{F}_q^n$ utilisé pour encoder les messages de manière à permettre la détection et la correction des erreurs introduites par un canal bruité.

Décodage

Étant donné un mot reçu $\underline{y} \in \mathbb{F}_q^n$, le **décodage** consiste à retrouver un mot $\hat{x} \in \mathcal{C}$ **proche** de $\underline{y}$ suivant la distance de Hamming.

Diagramme canal bruité



Bruit aléatoire($Y = X + N,\ N \sim \mathcal{N}(0, \sigma^2)$)

↓

Interférences(superposition de signaux)

↓

Atténuation($X \to \alpha X,\ \alpha \ll 1$)

↓

Distorsion(dispersion ou déformation)

↓

Brouillage(perturbation volontaire ou impulsive)

↓

Risque accru d'erreurs à la réception(perte de fiabilité des données)

Simulation d'un canal bruité (BSC)

Objectif

Simuler l'effet du bruit sur un message transmis, afin de motiver l'usage des codes correcteurs d'erreurs.

Modèle : canal binaire symétrique (BSC)

Un message binaire $M = (m_1, \dots, m_n) \in \{0,1\}^n$ est transmis à travers un canal où chaque bit m_i a une probabilité $p \in [0,1]$ d'être inversé.

Le bit reçu m_i' est donné par :

$$m_i' = \begin{cases} m_i & \text{avec probabilité } 1-p \\ 1-m_i & \text{avec probabilité } p \end{cases}$$

Simulation en Python d'un canal bruité

Principe

On génère un message binaire aléatoire M de longueur n, transmis à travers un canal simulé où chaque bit peut être inversé avec une probabilité p.

Interprétation de la probabilité p

- $p = 0$: transmission sans erreur.
- $0 < p < 0{,}1$: canal peu bruité.
- $p \approx 0{,}3$ ou plus : canal très bruité.

p est un paramètre libre représentant les conditions réelles de transmission (bruit thermique, distance, interférences...).

Résultats de la simulation

Exemple

$$\begin{array}{rl} \text{Message original} & M = [1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1] \\ \text{Message reçu} & M' = [1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0] \\ \text{Positions des erreurs} & \{3, 5, 8\} \end{array}$$

Avec $p = 0.2$, trois bits ont été inversés aléatoirement. Ces erreurs sont celles que les codes correcteurs tenteront de détecter et corriger.

Motivation mathématique

Modélisation mathématique

On modélise la transmission sur un canal bruité par :

$$y = x + e, \quad x \in \mathbb{F}_q^n,\ e \text{ vecteur d'erreurs.}$$

But : reconstruire x à partir de y, en exploitant une structure de redondance.

Motivation mathématique (suite)

Idée clé : redondance et structure

On introduit un **code correcteur** $\mathcal{C} \subset \mathbb{F}_q^n$ tel que deux mots distincts soient suffisamment *éloignés* :

$$\underline{x} \in \mathcal{C}, \quad W_H(\underline{e}) \leq t \implies \hat{x} = \underline{x}.$$

La capacité de correction dépend de la distance minimale entre les mots du code.

Deux types essentiels de codes

Code linéaire

Un code $\mathcal{C} \subset \mathbb{F}_q^n$ est dit **linéaire** s'il forme un *sous-espace vectoriel*. Il est caractérisé par :

- une matrice génératrice G (encodage)
- une matrice de contrôle H telle que $\mathcal{C} = \ker H$

Code parfait

Un code est **parfait** si toutes les erreurs de poids $\leq t$ sont corrigées sans ambiguïté :

$$\bigsqcup_{c \in \mathcal{C}} B(c,t) = \mathbb{F}_q^n$$

Vers les codes classiques

Objectif

Étudier deux familles classiques de codes correcteurs largement utilisées en pratique : :

- **Code de Hamming** : parfait, corrige 1 erreur.
- **Code de Reed-Solomon** : puissant, corrige plusieurs erreurs symboliques.

Ces codes seront présentés mathématiquement puis simulés.

Définition du code de Hamming $(7, 4)$

Principe

Un **code de Hamming** est un code linéaire $[n, k]$ qui corrige toutes les erreurs d'un seul bit.

- Longueur : $n = 7$
- Dimension : $k = 4$
- Distance minimale : $d_{\min} = 3 \Rightarrow t = 1$

Propriété

C'est un code **parfait** : chaque mot de $\mathbb{F}_2^7$ est à distance ≤ 1 d'un mot du code.

Définition du code de Hamming $(7, 4)$

Principe

Un **code de Hamming** est un code linéaire $[n, k]$ qui corrige toutes les erreurs d'un seul bit.

- Longueur : $n = 7$
- Dimension : $k = 4$
- Distance minimale : $d_{\text{min}} = 3 \Rightarrow t = 1$

Propriété

C'est un code **parfait** : chaque mot de $\mathbb{F}_2^7$ est à distance ≤ 1 d'un mot du code.

Encodage du Hamming $(7, 4)$

Matrice génératrice

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

Encodage

Pour $u = (u_1, u_2, u_3, u_4) \in \mathbb{F}_2^4$, le mot code est :

$$\underline{x} = uG \in \mathbb{F}_2^7$$

Décodage du Hamming $(7, 4)$

Matrice de contrôle

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

Décodage

- Calcul du **syndrome** : $s = Hy^T \in \mathbb{F}_2^3$
- Le syndrome indique la position de l'erreur (table de décodage)

Introduction aux codes Reed-Solomon

Les codes Reed-Solomon (RS) sont des codes correcteurs d'erreurs construits sur des corps finis $\mathbb{F}_q$, permettant de détecter et corriger un grand nombre d'erreurs.
Ils sont largement utilisés en communication numérique et en stockage (CD, DVD, QR codes, etc.).

Le principe clé : **représenter un message par un polynôme** et coder ce message en évaluant ce polynôme en plusieurs points distincts du corps fini.

Modélisation polynomiale du message

Soit un message $\mathbf{m} = (m_0, m_1, \ldots, m_{k-1}) \in \mathbb{F}_q^k$, avec $k \leq q$. On associe à ce message le polynôme

$$m(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{k-1} x^{k-1} \in \mathbb{F}_q[x].$$

L'**encodage** consiste à choisir $n \leq q$ points distincts $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{F}_q$ et à calculer :

$$\mathbf{c} = \big(m(\alpha_1), m(\alpha_2), \ldots, m(\alpha_n)\big) \in \mathbb{F}_q^n.$$

Ce vecteur $\mathbf{c}$ est le code Reed-Solomon associé au message.

Propriétés fondamentales des codes Reed-Solomon

- Le polynôme $m(x)$ a un degré inférieur à k, d'où une dimension du code égale à k.
- La distance minimale du code est $d = n - k + 1$.
- Cela permet de corriger jusqu'à $\left\lfloor \frac{d-1}{2} \right\rfloor$ erreurs.
- Les codes Reed-Solomon sont des codes **MDS** (Maximum Distance Separable), optimaux pour leur longueur et leur dimension.

Ces propriétés garantissent une correction d'erreurs très efficace, essentielle dans les systèmes numériques modernes.

Décodage des codes Reed-Solomon (aperçu)

Le décodage corrige les erreurs sur le vecteur reçu pour retrouver le message original.

Principales méthodes :

- Berlekamp-Massey
- Sugiyama (Euclide étendu)
- Décodage par interpolation

Principe : Résoudre des équations polynomiales pour localiser et corriger les erreurs.

Décodage d'un code Reed-Solomon (1/2)

Soit $r(x) = c(x) + e(x)$ le polynôme reçu.
Les **syndromes** S_i sont calculés par :

$$S_i = r(\alpha^i), \quad i = 1, \ldots, 2t,$$

où α est une racine primitive de $\mathbb{F}_q$.
Le **polynôme localisateur d'erreurs** $\Lambda(x)$ est défini par :

$$\Lambda(x) = \prod_{j=1}^{t} (1 - x\alpha_j),$$

avec α_j^{-1} les positions inverses des erreurs.
La relation clé est :

$$\Lambda(x)S(x) \equiv \Omega(x) \mod x^{2t}.$$

Objectif : trouver les racines de $\Lambda(x)$ pour localiser les erreurs.

Décodage d'un code Reed-Solomon (2/2)

Valeurs des erreurs e_j calculées par la formule de Forney :

$$e_j = -\frac{\Omega(\alpha_j^{-1})}{\Lambda'(\alpha_j^{-1})}$$

Correction :

$$c(x) = r(x) - e(x)$$

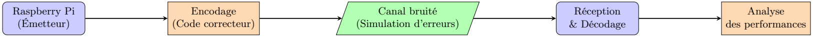
où $e(x)$ construit à partir des e_j et positions.

Calcul de $\Lambda(x)$ et $\Omega(x)$:

- Algorithme de Berlekamp-Massey
- Algorithme d'Euclide étendu

Résultat : Reconstruction exacte du message original.

Transition vers la modélisation pratique



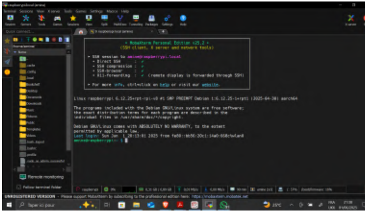
Ce schéma illustre le processus global du projet, de la transmission à la correction d'erreurs, avec un Raspberry Pi en émetteur.

Matériel utilisé et transmission

Raspberry Pi
Émetteur

$\longrightarrow$

Canal bruité / connexion Wi-Fi



PC
Récepteur

Objectif et déroulement de l'expérience

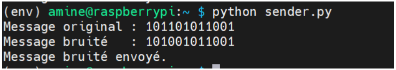
Objectif :

- Simuler une communication sans fil entre un Raspberry Pi (émetteur) et un PC (récepteur).
- Reproduire des erreurs de transmission via un canal bruité (taux d'erreur de 10%).

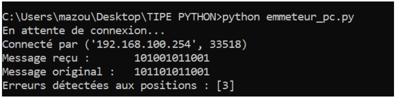
Déroulement :

- Le Raspberry Pi envoie un message binaire via une socket TCP.
- Le message passe par un canal simulé en Python qui introduit des erreurs aléatoires.
- Le PC reçoit le message et le compare avec l'original pour identifier les erreurs.

Résultat de la transmission



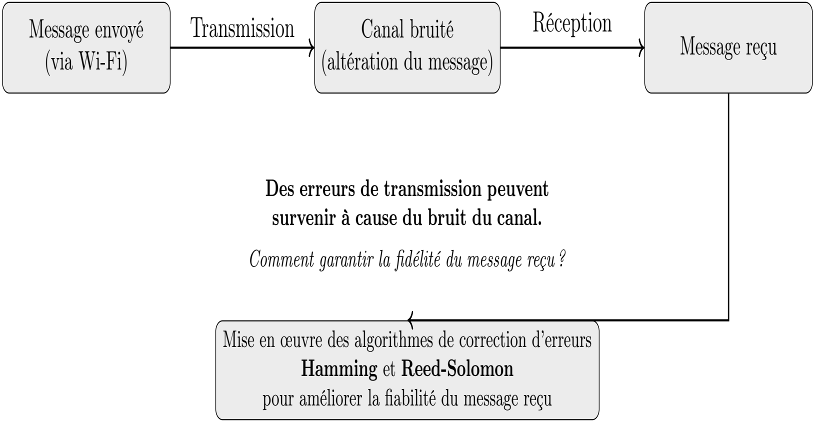
Émetteur (Raspberry Pi)



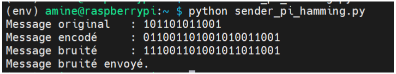
Récepteur (PC)

Une erreur a été introduite à la position 3 lors de la transmission.

Impact du canal bruité et correction d'erreurs



Correction avec Hamming (7,4)



Émetteur (Raspberry Pi)

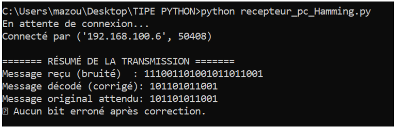
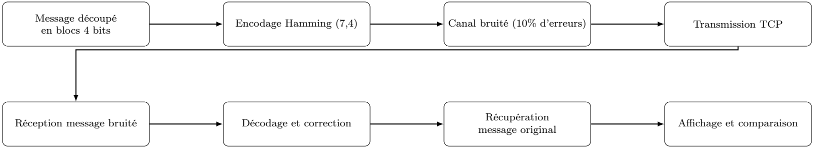


Figure – Récepteur – PC

Correction d'une erreur induite par le canal bruité grâce à l'algorithme Hamming.

Encodage / Décodage Hamming (7,4)



Hamming (7,4) corrige 1 erreur par bloc de 7 bits, assurant la fiabilité de la transmission.

Vers une correction plus robuste

L'expérience avec le code Hamming (7,4) a permis de :

- mettre en évidence les erreurs de transmission entre deux machines,
- démontrer la capacité de Hamming à détecter et corriger des erreurs simples.

Mais qu'en est-il des erreurs plus complexes ?

Explorons maintenant le code Reed-Solomon, plus adapté aux situations bruitées.

Expérience : Reed-Solomon

Objectif : Tester la robustesse du code **Reed-Solomon** sur une communication réelle entre un Raspberry Pi et un PC.

Méthode :

- Encodage du message avec Reed-Solomon sur le Raspberry Pi
- Transmission via socket TCP, avec bruit simulé
- Réception et décodage sur le PC
- Vérification de la capacité du code à **corriger plusieurs erreurs**

Correction avec Reed-Solomon (255, 223)

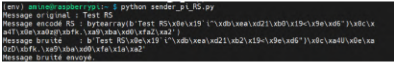


Figure – Émetteur (Raspberry Pi)

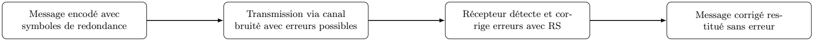
Figure – Récepteur (PC)

```
Message reçu (bytes) :
b'Test RS \x0e \x19 i^\db \ea \d2 1 \b2 \19 < ...

Message après correction : Test RS
```

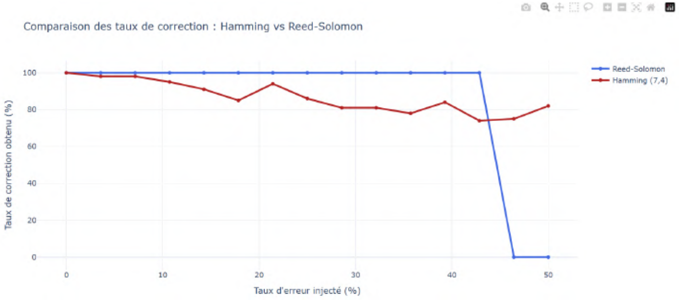
Correction d'erreurs de transmission réalisée avec un code Reed-Solomon (255, 223). Le message reçu contient des erreurs corrigées pour retrouver le message original.

Correction avec Reed-Solomon : résumé pratique



Reed-Solomon améliore la fiabilité des transmissions numériques en corrigeant automatiquement plusieurs erreurs.

Comparaison graphique des performances



Évolution du taux de correction en fonction du taux d'erreur injecté dans le canal.

Analyse quantitative des performances

- **Hamming (7,4)** corrige uniquement les erreurs simples : taux de correction chute dès que le taux d'erreur dépasse **15–20%**.
- **Reed-Solomon** reste efficace jusqu'à **40–45%** d'erreurs grâce à sa capacité à corriger plusieurs symboles.
- À partir de 25% d'erreurs, Hamming devient inutilisable, alors que Reed-Solomon conserve un taux de correction constant à **100%**.

Les simulations confirment la robustesse supérieure de Reed-Solomon dans les canaux bruités.

Conclusion

- Le code de Hamming est une solution simple et rapide, idéale pour corriger des erreurs isolées.
- Le code de Reed-Solomon offre une correction plus robuste, adaptée aux environnements très bruités.
- Ces techniques sont incontournables dans de nombreux domaines critiques :
  - Médecine à distance (chirurgie, télémédecine)
  - Stockage et transmission fiables (CD, QR codes)
- Ces résultats ouvrent des perspectives vers l'étude de codes encore plus performants, tels que les codes LDPC ou polaires, utilisés notamment dans la 5G et les communications spatiales.

Merci pour votre attention !

Simulation d'un canal BSC

```python
# Simulation Canal BSC
import numpy as np

# Paramètres
n = 8                    # Longueur du message
p = 0.2                  # Probabilité d'erreur sur chaque bit

# Génération du message original (binaire aléatoire)
M = np.random.randint(0, 2, n)

# Application du bruit (canal binaire symétrique)
E = np.random.rand(n) < p      # Tableau booléen : True si erreur sur le bit
M_bruite = M ^ E.astype(int)   # XOR bit à bit pour inverser les bits erronés

# Recherche des positions des erreurs
positions_erreurs = np.where(E)[0] + 1

# Affichage des résultats
print(f"Message original : {M}")
print(f"Message reçu     : {M_bruite}")
print(f"Positions des erreurs : {set(positions_erreurs)}")
```

Émission sans encodage (Raspberry Pi) – 1/2

```python
#Sender Raspberry Pi
import random
import socket

def canal_bruite(message, taux_erreur=0.1):
    message_bruite = ""
    for bit in message:
        if random.random() < taux_erreur:
            # Inverse le bit (0 -> 1, 1 -> 0)
            message_bruite += '1' if bit == '0' else '0'
        else:
            message_bruite += bit
    return message_bruite

HOST = '192.168.100.55'  # IP locale de mon PC (receiver)
PORT = 65432

message = "101101011001"  # Message binaire exemple

# Appliquer le canal bruité avec un taux d'erreur de 10%
message_bruite = canal_bruite(message, taux_erreur=0.1)

```

Émission sans encodage (Raspberry Pi) – 2/2

```python
print("Message original :", message)
print("Message bruité   :", message_bruite)

# Création du socket client (sender)
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((HOST, PORT))

# Envoi du message bruité
client_socket.sendall(message_bruite.encode())

print("Message bruité envoyé.")

client_socket.close()
```

Réception sans décodage (PC) – 1/2

```python
# Recepteur pc
import socket
def detecter_erreurs(message_recu, message_original):
    erreurs = []
    for i in range(len(message_original)):
        if message_recu[i] != message_original[i]:
            erreurs.append(i)
    return erreurs
HOST = '0.0.0.0'  # écoute sur toutes les interfaces
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print("En attente de connexion...")
    conn, addr = s.accept()
    with conn:
        print(f"Connecté par {addr}")
        data = conn.recv(1024)
        if not data:
            print("Pas de données reçues.")
```

Réception sans décodage (PC) – 2/2

```
else:
    message_recu = data.decode()
    message_original = "101101011001"  # Exemple

    print(f"Message reçu :       {message_recu}")
    print(f"Message original :   {message_original}")

    erreurs = detecter_erreurs(message_recu, message_original)
    if erreurs:
        print(f"Erreurs détectées aux positions : {erreurs}")
    else:
        print("Aucune erreur détectée.")
```

Émission avec encodage Hamming – 1/2

```python
# Sender Raspberry Pi avec Encodage Hamming (7,4)
import random
import socket
# Fonction d'encodage Hamming(7,4)
def hamming_encode(data_4bits):
    d = list(map(int, data_4bits))
    p1 = d[0] ^ d[1] ^ d[3]
    p2 = d[0] ^ d[2] ^ d[3]
    p3 = d[1] ^ d[2] ^ d[3]
    encoded = [p1, p2, d[0], p3, d[1], d[2], d[3]]
    return ''.join(str(bit) for bit in encoded)
# Canal bruité
def canal_bruite(message, taux_erreur=0.1):
    message_bruite = ""
    for bit in message:
        if random.random() < taux_erreur:
            message_bruite += '1' if bit == '0' else '0'
        else:
            message_bruite += bit
    return message_bruite

```

Émission avec encodage Hamming – 2/2

```
# Configuration
HOST = '192.168.100.55'  # IP du récepteur
PORT = 65432
message_original = "101101011001"  # 12 bits
# Étape 1 : Diviser le message en blocs de 4 bits
blocs_4bits = [message_original[i:i+4] for i in range(0, len(message_
# Étape 2 : Encoder chaque bloc avec Hamming(7,4)
message_encode = ''.join(hamming_encode(bloc) for bloc in blocs_4bits
# Étape 3 : Ajouter du bruit
message_bruite = canal_bruite(message_encode, taux_erreur=0.1)
# Affichage
print("Message original   :", message_original)
print("Message encodé     :", message_encode)
print("Message bruité     :", message_bruite)
# Étape 4 : Envoi
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((HOST, PORT))
client_socket.sendall(message_bruite.encode())
print("Message bruité envoyé.")
client_socket.close()
```

Réception avec décodage Hamming – 1/3

```python
# Recepteur pc decodage Hamming
import socket
def hamming74_decoder(bits):
    decoded = ""
    for i in range(0, len(bits), 7):
        block = bits[i:i+7]
        if len(block) < 7:
            continue  # ignore les blocs incomplets
        # Bits individuels
        r = list(map(int, block))
        p1, p2, d1, p3, d2, d3, d4 = r
        # Syndrome
        s1 = p1 ^ d1 ^ d2 ^ d4
        s2 = p2 ^ d1 ^ d3 ^ d4
        s3 = p3 ^ d2 ^ d3 ^ d4
        erreur_pos = s1 + (s2 << 1) + (s3 << 2)
        if erreur_pos != 0:
            # Corriger l'erreur
            erreur_index = erreur_pos - 1  # index 0-based
            if erreur_index < 7:
                r[erreur_index] ^= 1  # flip le bit erroné
        # Après correction, extraire les données
        p1, p2, d1, p3, d2, d3, d4 = r
        decoded += f"{d1}{d2}{d3}{d4}"
    return decoded
```

Réception avec décodage Hamming – 2/3

```python
def detecter_erreurs(message_recu, message_original):
    erreurs = []
    for i in range(min(len(message_recu), len(message_original))):
        if message_recu[i] != message_original[i]:
            erreurs.append(i)
    return erreurs
# Réseau
HOST = '0.0.0.0'
PORT = 65432
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print("En attente de connexion...")
    conn, addr = s.accept()
    with conn:
        print(f"Connecté par {addr}")
        data = conn.recv(1024)
        if not data:
            print("Pas de données reçues.")
        else:
            message_bruite = data.decode()

            # Décodage du message reçu
            message_decode = hamming74_decoder(message_bruite)

```

Réception avec décodage Hamming – 3/3

```
 # Message d'origine (doit correspondre à ce que le sender
# a encodé avant Hamming)
 message_original = "101101011001"  # 12 bits → 3 blocs Hamming
 #(3×7 = 21 bits envoyés)

 print("\n======= RÉSUMÉ DE LA TRANSMISSION =======")
 print(f"Message reçu (bruité)  : {message_bruite}")
 print(f"Message décodé (corrigé): {message_decode}")
 print(f"Message original attendu: {message_original}")

 erreurs = detecter_erreurs(message_decode, message_original)
 if erreurs:
     print(f" Erreurs restantes après correction aux positions: {err
 else:
     print(" Aucun bit erroné après correction.")
```

Émission avec encodage Reed-Solomon – 1/2

```python
#Sender Raspberry pi encodage RS
import socket
import random
from reedsolo import RSCodec
def canal_bruite(message, taux_erreur=0.01):
    message_bruite = bytearray()
    for byte in message:
        # Pour chaque bit dans le byte, on inverse avec une proba tau
        bits = list(f'{byte:08b}')
        for i in range(len(bits)):
            if random.random() < taux_erreur:
                bits[i] = '1' if bits[i] == '0' else '0'
        message_bruite.append(int(''.join(bits), 2))
    return bytes(message_bruite)
HOST = '192.168.100.55'  # IP de mon PC (récepteur)
PORT = 65432

message = "Test RS"  # message simple

rsc = RSCodec(32)  # 32 symboles de correction, corrige jusqu'à 16 er
```

Émission avec encodage Reed-Solomon – 2/2

```
message_code = rsc.encode(message.encode('utf-8'))

message_bruite = canal_bruite(message_code, taux_erreur=0.01)

print("Message original :", message)
print("Message encodé RS :", message_code)
print("Message bruité    :", message_bruite)
# Envoi via socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STR
client_socket.connect((HOST, PORT))
client_socket.sendall(message_bruite)
print("Message bruité envoyé.")
client_socket.close()
```

Réception avec décodage Reed-Solomon – 1/2

```python
# Recepteur RS PC
import socket
from reedsolo import RSCodec

rsc = RSCodec(32)  # même configuration qu'au sender

def decoder_message(message_bytes):
    try:
        decoded_tuple = rsc.decode(message_bytes)
        # decode() retourne un tuple (message_corrige, erreurs
        message_decode = decoded_tuple[0]
        return message_decode.decode('utf-8')
    except Exception as e:
        return f"[Erreur lors du décodage] : {e}"

HOST = '0.0.0.0'
PORT = 65432

```

Réception avec décodage Reed-Solomon – 2/2

```

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print("En attente de connexion...")
    conn, addr = s.accept()
    with conn:
        print(f"Connecté par {addr}")
        data = conn.recv(1024)
        if not data:
            print("Pas de données reçues.")
        else:
            print("Message reçu (bytes) :", data)
            message_corrige = decoder_message(data)
            print("Message après correction :", message_corri

```

Graphique comparatif – 1/4

```python
# Graphe Comparaison Performance
import numpy as np
import plotly.graph_objects as go
from reedsolo import RSCodec, ReedSolomonError
def simulate_hamming(error_rate, trials=100):
    def hamming_encode(msg):
        G = np.array([
            [1, 0, 0, 0, 0, 1, 1],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 1, 0, 1, 1, 0],
            [0, 0, 0, 1, 1, 1, 1]
        ])
        return np.dot(msg, G) % 2
    def hamming_decode(code):
        H = np.array([
            [0, 0, 0, 1, 1, 1, 1],
            [0, 1, 1, 0, 0, 1, 1],
            [1, 0, 1, 0, 1, 0, 1]
        ])
        syndrome = np.dot(H, code) % 2
        syndrome_decimal = int(''.join(str(int(x)) for x in syndrome[::-1
```

Graphique comparatif - 2/4

```python
        if syndrome_decimal != 0:
            code[syndrome_decimal - 1] ^= 1
        return code[:4]
    success_count = 0
    for _ in range(trials):
        msg = np.random.randint(0, 2, 4)
        encoded = hamming_encode(msg)
        noisy = encoded.copy()
        if np.random.rand() < error_rate:
            pos = np.random.randint(0, 7)
            noisy[pos] ^= 1
        decoded = hamming_decode(noisy)
        if np.array_equal(decoded, msg):
            success_count += 1
    return success_count / trials * 100

def simulate_rs(message_bytes, error_rate, nsym=32):
    rsc = RSCodec(nsym)
    encoded = rsc.encode(message_bytes)
    n_errors = int(len(encoded) * error_rate)
    noisy = bytearray(encoded)
    error_positions = np.random.choice(len(encoded), n_errors, replace=False)
```

Graphique comparatif – 3/4

```
        for pos in error_positions:
            noisy[pos] ^= np.random.randint(1, 256)
        try:
            rsc.decode(noisy)
            return 1.0
        except ReedSolomonError:
            return 0.0

def get_avg_rs_correction_rate(message, error_rate, trials=20):
    return np.mean([simulate_rs(message, error_rate) for _ in range(trials)]

# Simulation
error_rates = np.linspace(0, 0.5, 15)
rs_results = [get_avg_rs_correction_rate(b'Test RS', e) for e in error_rates
hamming_results = [simulate_hamming(e) for e in error_rates]

# Graphique
fig = go.Figure()

fig.add_trace(go.Scatter(x=error_rates*100, y=rs_results,
                         mode='lines+markers',
                         name='Reed-Solomon',
```

Graphique comparatif – 4/4

```
                                line=dict(color='royalblue', width=3)))

fig.add_trace(go.Scatter(x=error_rates*100, y=hamming_results,
                                mode='lines+markers',
                                name='Hamming (7,4)',
                                line=dict(color='firebrick', width=3)))

fig.update_layout(
    title='Comparaison des taux de correction : Hamming vs Reed-S
    xaxis_title='Taux d\'erreur injecté (%)',
    yaxis_title='Taux de correction obtenu (%)',
    xaxis=dict(range=[0, 50]),
    yaxis=dict(range=[0, 105]),
    template='plotly_white'
)

fig.show()
```