UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS
Licence informatique & vidéoludisme
Semestre 5

Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/is
Julien Lavauzelle <julien.lavauzelle@univ-paris8.fr>
https://www.lvzl.fr/teaching/2024-25/is.html

Introduction à la sécurité

TP 6 : Codes correcteurs d'err£urs

N'oubliez pas :
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu sur Moodle dans une archive. Faites en sorte que l'archive ne soit pas trop lourde (sinon il faudra nous prévenir). Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net !**

Dans ce TP :
- Implémenter différents codes correcteurs d'erreur.
- Modéliser des canaux de transmission élémentaires.
- Calculer un taux d'erreur résiduel et comparer différentes méthodes de correction d'erreur.

Exercice 0.

Recommandations.

1. Ce TP est à faire en Python.
2. Faites les questions dans l'ordre et pensez à tester votre code.
3. N'hésitez jamais à rajouter de la sortie de debug pour comprendre tout ce qui se passe.

Exercice 1.

Échauffement : code de répétition

1. Commençons par un premier code correcteur : le **code de répétition** de **longueur** n, où n est un entier ≥ 1. Son fonctionnement est très simple :
   - les messages (élémentaires) sont de taille 1 bit ;
   - pour encoder un bit, on le répète n fois ;
   - pour décoder un mot reçu formé de n bits $c_1 c_2 \dots c_n$, on compte le nombre de bits à 0 et le nombre de bits à 1 parmi les c_i, et on décide que le message initial correspond au bit le plus fréquent (c'est un vote majoritaire).

   Par exemple, pour $n = 3$, le bit `0` est encodé en un bloc de 3 bits `000` (appelé **mot de code**). Si ce bloc est altéré par une erreur, par exemple en deuxième position (`010`), alors les zéros restent majoritaires dans le mot de code, et on peut donc retrouver le message initial `0` par un vote majoritaire.

   Généralement, un code de répétition de longueur n permet donc de décoder jusque $\lfloor \frac{n-1}{2} \rfloor$ erreurs intervenues sur le mot transmis. Pour $n = 3$, on peut donc corriger une erreur, pour $n = 5$ deux erreurs, etc.

   → Implémentez une fonction `encode_repeat(b, n)` qui prend en entrée un bit `b` (représenté par un entier 0 ou 1), et qui retourne l'entier correspondant à la répétition du bit `b` une nombre `n` de fois.

   → Implémentez une fonction `decode_repeat(c, n)`, qui décode le mot `word` (représenté comme un entier entre 0 et $2^n - 1$) selon la majorité des bits qui le compose.

2. Pour observer et tester les transformations binaires que nous allons effectuer sur les données, il est utile d'afficher leur représentation binaire.

   → Implémentez une fonction `binary_print(value, size)`, qui affiche la représentation binaire de l'entier `value` sur `size` bits. On supposera donc que `value` est plus petit que $2^{\text{size}} - 1$.

   → Grâce à votre nouvelle fonction d'affichage graphique, vérifiez que l'encodage et le décodage sont corrects.

3. Nous allons maintenant modéliser un premier **canal de transmission**. Ce canal est simple : il prend en entrée un bloc de n bits (toujours représenté par un entier `value` compris entre 0 et $2^n - 1$), et il modifie exactement **un** bit du bloc, dont l'indice a été tiré aléatoirement parmi les n possibilités.

   → Implémentez une fonction `channel(value, n)`, qui modélise le canal présenté ci-dessus. La fonction retournera donc un entier réprésenté sur n bits.

4. → Écrire une procédure de test pour ce premier code correcteur. Vore procédure effectuera une certaine quantité de fois les étapes suivantes :
   — engendrer un bit aléatoire;
   — l'encoder avec le code de répétition de longueur 3 (cela produit un mot de code);
   — transmettre le mot de code obtenu à travers le canal;
   — décoder le mot bruité;
   — vérifier si le mot bruité décodé est bien égal au bit initial.

Exercice 2.

Un code correcteur d'expansion 2 : le code carré

1. Dans cet exercice, on s'intéresse à un code correcteur un peu plus avancé, qui permet de corriger une erreur, mais avec un facteur d'expansion moindre que celui du code de répétition.

   Ce code correcteur, parfois appelé code carré, a les paramètres suivants :
   — il prend en entrée des blocs de taille $k = 4$ bits (on dit que la dimension est 4);
   — il retourne des mots de taille $n = 8$ bits (on dit que la longueur est 8);
   — il permet de décoder 1 erreur de manière certaine.

   Écrivons le message de 4 bits à encoder sous la forme $m_0m_1m_2m_3$. Le code carré leur concatène $n - k = 8 - 4 = 4$ nouveaux bits $p_4p_5p_6p_7$ (appelés **bits de parité**, ou bits de **redondance**) de la manière suivante :
   — p_4 est le xor de m_0 et m_1 ;
   — p_5 est le xor de m_2 et m_3 ;
   — p_6 est le xor de m_0 et m_2 ;
   — p_7 est le xor de m_1 et m_3.
   Ces relations entre les bits de parité p_j et les bits de message m_i sont appelées **équations de parité**.

   La Figure 1 illustre le procédé d'encodage, et le rôle de bits de parité. Le nom "code carré" provient du fait que les 4 bits de message sont agencés dans un "carré" de taille 2×2.

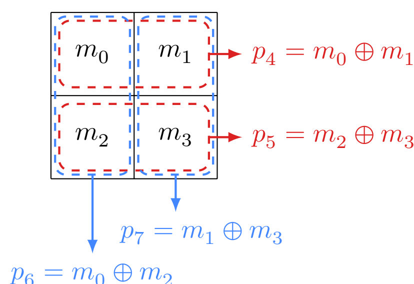

FIGURE 1 – Illustration du code carré. Les bits p_4 et p_5 (en rouge) contrôlent la validité des lignes, les bits p_6 et p_7 (en bleu) celle des colonnes.

   → Implémentez une fonction `encode_square(value)`, qui prend en entrée un message `value` de 4 bits représenté par un entier compris entre 0 et $2^4 - 1 = 15$, et encode ce message en un mot de 8 bits représenté par un entier compris entre 0 et 255.

2. Supposons maintenant qu'une erreur altère le mot de code. Si cette erreur se trouve parmi les quatre bits de message m_i, alors deux équations de parité ne seront plus vérifiées : celles correspondant à la "ligne" et la "colonne" de la position de l'erreur. En identifiant les équations de parité non-satisfaites, on peut donc **localiser** l'erreur et la corriger en "flippant" le bit correspondant.

   Une erreur sur un bit de **message** m_i corromp donc 2 équations de parité.

   Maintenant, si c'est un bit de parité p_j qui est altéré, alors une unique équation de parité sera non-satisfaite. Mais dans ce cas, les bits de message ne sont pas altérés, donc il n'y a rien à faire, si ce n'est retourner le bloc $m_0m_1m_2m_3$.

   Résumé du décodage. Considérons un mot $y = y_0y_1y_2y_3y_4y_5y_6y_7$ correspondant à un mot de code altéré par (au plus) une erreur. Pour corriger cette erreur, on procède la sorte :
   — Identifier, parmi les 4 équations de parité suivantes, lesquelles ne sont pas satisfaites : :

$$\begin{cases} y_0 \oplus y_1 \oplus y_4 = 0 \\ y_2 \oplus y_3 \oplus y_5 = 0 \\ y_0 \oplus y_2 \oplus y_6 = 0 \\ y_1 \oplus y_3 \oplus y_7 = 0 \end{cases}$$

   — S'il y en a 0 ou 1, retourner le bloc $y_0y_1y_2y_3$.

— S'il y en a 2, identifier l'indice i du bit y_i commun à ces deux équations de parité. Puis, flipper le bit y_i et retourner le bloc $y_0y_1y_2y_3$ correspondant.
— S'il y en a 3 ou 4, il y a trop d'erreur dans le message, on ne peut pas décoder (retourner quand même $y_0y_1y_2y_3$ pour la cohérence de sortie).

$\rightarrow$ Implémentez une fonction `decode_square(value)`, qui prend en entrée `value`, un bloc de 8 bits représenté par un entier compris entre 0 et 255, et décode ce bloc en un bloc de 4 bits représenté par un entier compris entre 0 et 15, selon l'algorithme présenté plus haut.

3. $\rightarrow$ Testez la validité de vos fonctions, en utilisant le canal modélisé dans l'exercice précédent. On pourra également s'aider de la fonction d'affichage binaire.
4. On souhaite maintenant encoder et décoder du texte, sous forme de chaînes de caractères. Pour cela, il faut découper nos chaînes en blocs de 4 bits, puis encoder successivement chacun des blocs obtenus.

   $\rightarrow$ Implémentez une fonction `cut(text)`, qui prend en entrée une chaine de caractères `text`, et qui retourne un tableau d'entiers compris entre 0 et 15 représentés sur 4 bits, dont la concaténation des bits est égale à celle de la chaîne de caractères.

   $\rightarrow$ Implémentez une fonction `glue(tab)`, qui effectue de procédé inverse de la focntion `cut(text)`.

   $\rightarrow$ Implémentez une fonction `encode_message(text)`, qui prend en entrée une chaine de caractères `text`, et qui retourne un tableau d'octets (type python : `bytearray`) correspondant à la concaténation des encodages de blocs de 4 bits issus du découpage de `text` par la fonction `cut`.

   $\rightarrow$ Implémentez une fonction `decode_message(tab)`, qui décode un tableau d'octets octet par octet, grâce à la fonction `decode_square` implémentée plus haut.
5. $\rightarrow$ Implémentez une fonction `channel_message(tab)`, qui introduit une erreur aléatoire dans chaque octet du tableau d'octets `tab` passé en paramètre. Puis, encodez, altérez et décodez les chaînes de caractères de votre choix.

Exercice 3.

Comparaison des méthodes sur le canal binaire symétrique

1. Le canal utilisé pour les deux premiers exercices n'est pas très réaliste : il provoque **exactement** une erreur sur une position aléatoire du mot à transmettre, peu importe la longueur de ce mot.

   Un canal plus réaliste est le **canal binaire symétrique** (souvent abrégé en BSC pour *binary symmetric channel*). Ce canal transmet le message bit par bit, et modifie chacun des bits de manière indépendante et avec probabilité p, où p est un paramètre compris entre 0 et 1. Autrement dit :
   — si le bit `0` est en entrée du canal, alors en sortie on a `1` avec probabilité p et `0` avec probabilité $1-p$;
   — si le bit `1` est en entrée du canal, alors en sortie on a `0` avec probabilité p et `1` avec probabilité $1-p$.

   $\rightarrow$ Implémentez une fonction `bsc(block, p, size)`, qui transmet un message `block` de longueur `size` via le canal binaire symétrique de paramètre `p`.
2. Maintenant, on souhaite mesurer le taux de transmission du canal, sur des messages de longueur 4 bits. Pour cela, on identifie 3 méthodes :
   (A) on transmet simplement le message à travers le canal, sans rien encoder;
   (B) on encode chacun des 4 bits avec le code de répétition de longueur 3, puis on transmet les 12 bits correspondants à travers le canal, et on essaie de décoder le résultat;
   (C) on encode un bloc de 4 bits avec le code carré, puis on transmet les 8 bits correspondants à travers le canal, et on essaie de décoder.

   Notre objectif est de comparer ces 3 méthodes.

   Pour chacune de ces méthodes, on dit que le message est **bien transmis** si, en sortie de canal et après décodage, on retrouve le message initial.

   $\rightarrow$ Pour chaque méthode (A), (B) et (C), écrire une fonction qui prend en paramètre p, effectue 1000 tirages aléatoires de messages de longueur 4, et compte le nombre de messages "bien transmis" à travers le canal. Puis, tester la méthode avec $p = 0.1$.

   $\rightarrow$ En faisant varier p par pas de 0.001 entre 0 et 0.5, afficher dans un graphique (avec matplotlib ou gnuplot par exemple) le taux de transmission de chaque méthode en fonction de p. Quelle méthode transmet le mieux l'information ?

   $\rightarrow$ Le **facteur d'expansion** d'un code est le ratio entre le nombre de bits transmis et le nombre de bits du message initial. Par exemple, il est de $\frac{8}{4} = 2$ pour le code carré. Comparer les **facteurs d'expansion** de chaque méthode. Au vu des résultats de la sous-question précédente, est-ce raisonnable ? Argumentez.