Université de Strasbourg

UFR Mathématique & Informatique
Licence 3 Semestre 6
Réseaux Locaux & Interconnexions
2021-2022

Projet RLI 2022 : Hadamard & Simplex

Ce projet peut être faire seul ou en binôme. Il est à rendre sur Moodle le **vendredi 13 mai** au plus tard.

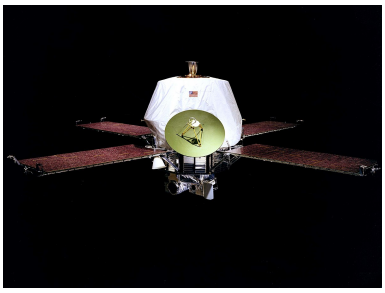
Avant de commencer

Ce projet comporte des questions théoriques, auxquelles vous devez répondre dans un rapport à fournir avec votre rendu. N'oubliez pas d'indiquer vos noms dans le rapport. Le rapport doit être un fichier pdf. La qualité du rapport sera prise en compte dans la notation.
Votre code doit être commenté, indenté, et doit compiler (n'oubliez pas de fournir un makefile). Commentez et décrivez vos choix d'implémentation lorsque nécessaire dans le rapport. Veillez à suivre les consignes concernant le prototype et le nom des fonctions à implémenter. Le code doit être fait dans le langage C.
Déposez une archive .zip avec les noms du binôme, contenant votre rapport et votre code. Le travail est à faire seul ou en binôme. On rappelle que le plagiat est sévèrement sanctionné.

1 Contexte

FIGURE 1 – Jacques Hadamard

Jacques Hadamard était un mathématicien français ayant vécu entre 1865 et 1963 [2]. Hadamard est connu pour de nombreux travaux dans des domaines variés (théorie des nombres, géométrie différentielle, analyse ...). Il a notamment prouvé le Théorème des Nombres Premiers (le nombre de nombres premiers inférieurs ou égaux à x est équivalent à $\frac{x}{log(x)}$ quand x tend vers l'infini).
Jacques Hadamard a également laissé son nom aux *matrices d'Hadamard*. Ces-dernières sont utilisées pour la *transformée d'Hadamard*, notamment utilisée à présent en informatique quantique (la porte d'Hadamard est une porte logique quantique agissant sur un qubit). Les matrices d'Hadamard apparaissent également dans les codes correcteurs d'erreur.



(a) La sonde Mariner 9 ayant utilisé les codes d'Hadamard

(b) Photo de Mariner 9 du sol martien après correction

Le *code correcteur d'Hadamard* se repose sur les matrices d'Hadamard. À noter que les matrices d'Hadamard ont été conçues bien avant le premier code correcteur d'erreur. Jacques Hadamard n'a donc pas inventé le code d'Hadamard lui-même. Par conséquent, le code d'Hadamard est parfois appelé le code Walsh-Hadamard [1].

Le code d'Hadamard est un code prenant un mot de taille k et donnant un mot du code de taille 2^k. Sa capacité de correction, que nous découvrirons plus tard, en fait un code intéressant malgré son rendement faible. La sonde *Mariner 9*, envoyée vers Mars en 1971 (et le premier véhicule spatial à orbiter une autre planète), utilisait notamment un code d'Hadamard [3]. L'utilisation de ce code a permis de renvoyer vers la Terre plus de 7000 images nettes de Mars (cartographiant 85% de cette dernière), malgré les nombreuses interférences (l'espace étant un medium entraînant $\approx$ 25% d'erreurs). La sonde Mariner 9 doit d'ailleurs s'écraser prochainement sur Mars (en mars 2022 selon les dernières estimations !).

Dans ce projet, nous implémenterons un code d'Hadamard (avec une très légère optimisation), investiguerons certaines de ces caractéristiques avant d'étudier la manière dont ce code est implémenté en pratique (i.e., en hardware).

2 Sujet

2.1 Découverte du code d'Hadamard

Une matrice génératrice d'un code d'Hadamard G_{hd} est décrite ci dessous.

$$G_{hd} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Pour des raisons de simplicité, nous travaillerons sur la version systématique du code d'Hadamard, dont la matrice G_{hds} est donnée ci-dessous [2]

1. Joseph Walsh, ayant laissé son nom au code Walsh-Hadamard, était un orateur invité au congrès international des mathématiciens en 1920 à Strasbourg !

2. Il est possible d'intervertir des colonnes et lignes d'une matrice génératrice sans affecter la puissance du code. Des codes ayant une même matrice génératrice à ces permutations près sont appelés équivalents, ou isomorphes.

$$G_{hds} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Question 1. Étudiez la matrice G_{hds}. Comment voit-on qu'elle est systématique ? Combien de bits de données encode-t-elle ? Quelle est la taille des mots du code ? Déduisez-en le rendement du code.

2.2 Implémentation du code d'Hadamard & analyse

Dans la suite ce projet, nous compterons les bits de gauche à droite. Ainsi, le premier bit de `1000000000000000` est 1. Son indice est 0. Les 4 premiers bits de `1010111100011000` sont `1010`. Nous travaillerons sur des *short* (i.e., des variables sur 16 bits). Vous trouverez à la fin de ce sujet quelques rappels sur les opérations bit à bit en C. On rappelle également qu'il est possible d'écrire des entiers en binaire en C à l'aide de la notation `0b`. Par exemple, il est possible de déclarer une variable en binaire via `short x = 0b1010101010101010;`.

Question 2. Pour simplifier votre travail, sa lecture, et le debuggage, commencez par écrire ces quatre fonctions utilitaires `set_nth_bit`, `read_nth_bit`, `print_word` et `chg_nth_bit`. **Pour implémenter ces fonctions, vous veillerez à n'utiliser que des opérateurs logiques bit à bit et/ou des shifts. L'utilisation de *if* ou autre structure de contrôle n'est pas autorisée (à l'exception de for pour la fonction `print_word`).**

1. La fonction `unsigned short set_nth_bit(int n, unsigned short m)` prend en argument un entier n, un mot (encodé sur un short) m, et mets le n-ieme bit du mot à 1. Le mot modifié est retourné.

   Exemple. `set_nth_bit(0, 0b0100000000000000)` renvoie `0b1100000000000000`.

2. La fonction `unsigned short get_nth_bit(int n, unsigned short m)` prend en argument un entier n, un mot (encodé sur un short) m, et retourne la valeur du nieme bit du mot.

   Exemple. `get_nth_bit(3, 0b0001000000000000` renvoie 1).

3. La fonction `unsigned short chg_nth_bit(int n, unsigned short m)` prend en argument un entier n, un mot (encodé sur un short) m, et change la valeur du nieme bit (de 0 à 1 ou de 1 à 0). Le mot modifié est renvoyé.

4. La fonction `void print_word( int k, unsigned short m)` prend en argument un mot (encodé sur un short) m et un entier k, et imprime les k premiers bits de m.

   Exemple. `print_word( 7, 0b1010101000111111) renvoie 0b1010101)`.

Question 3. Implémentez en C le code décrit par la matrice G_{hds}. Plus précisément, vous implémenterez une fonction `unsigned short encode_had_sys(unsigned short m)` prenant en argument un mot à encoder (occupant les 4 premiers bits de m), et y rajoutant (sur les 12 bits libres suivants) les bits de contrôle tels que définis par G_{hds}. Encore une fois, veillez à n'utiliser que des opérateurs bit à bit. Il n'est pas demandé de faire un code générique fonctionnant pour d'autres matrices.

Exemple. On considère par exemple un mot à encoder `1011`. On passe alors le mot `1011000000000000` à la fonction. Ici, `encode_had_sys(0b1011000000000000)` renverra le mot du code `0b1011001100011001`.

Question 4.1. Implémentez une fonction comptant le nombre de bits à 1 dans un mot. Plus précisément, la fonction `int cnt_bits(unsigned short n)` prendra en argument un `short` (par exemple, `0b1111000011110000`) et renverra le nombre de bits à 1 (8 dans l'exemple). Pour implémenter cette fonction, vous êtes encouragés à utiliser(/comprendre) l'algorithme de *Brian Kernighan* [1], une méthode élégante et efficace pour effectuer cette opération, dont la complexité dépend du nombre de bit à 1 du mot.

Question 4.2. Écrivez une fonction `int dist_code_hds()` qui calcule la distance de Hamming du code représenté par G_{hds}. Sachant la distance, déduisez combien d'erreur(s) ce code peut détecter/corriger.

Question 4.3. Le code d'Hadamard possède une particularité concernant la distance entre les mots du code le composant. Écrivez une fonction comparant les distances entre toutes les paires de mots du code valides. Que remarquez-vous de particulier concernant la distance entre n'importe quelle paire de mots du code valides ?

2.3 D'Hadamard au simplexe

Question 5. La matrice G_{hds} possède une colonne nulle. Selon vous, cette colonne est-elle utile ? Pourquoi ?

Question 6. Soit G_s une matrice de dimension $(4, 15)$ étant définie comme G_{hds} privée de sa colonne nulle.

$$G_{hds} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Le code résultant de cette matrice génératrice est appelé code *simplexe*[3].
Créez une nouvelle fonction `unsigned short encode_simplexe(unsigned short m)` qui encode un message selon la matrice G_s, de manière analogue à la question 3. À l'aide de vos fonctions précédentes, montrez que le code simplexe possède la même distance que le code d'Hadamard défini par G_{hds} malgré la colonne supprimée.
Note : La nouvelle matrice génératrice encodera des mots de 4bits vers des mots de 15bits. Créer des variables de 15bits en C n'est pas trivial (et changerait le prototype de nos fonctions). Vous pouvez donc stocker les mots de 15bits dans des *short*, et simplement ignorer le dernier bit (le plus à droite) dans votre code et vos affichages.

Question 8. Soit un code c_1 ayant comme matrice génératrice G et comme matrice de parité H. Soit c_2 un code ayant comme matrice génératrice H et comme matrice de parité G. On dit que c_1 est le *code dual* de c_2. Trouvez la matrice de parité H de G_s. Connaissez vous le code

3. Les mots du code d'un code simplexe dessine une forme géométrique particulière, un simplexe, lorsque représentés dans l'espace des mots possibles.

dual du code simplexe ? (Rappel : il est autorisé d'intervertir les lignes/colonnes).

Rappel : Pour les questions suivantes, souvenez-vous que le dernier bit est ignoré, les mots du code étant sur 15 bits.

Question 9. Implémentez une fonction `unsigned short decode(unsigned short m)`, prenant en argument un message reçu m et renvoyant le syndrome par H. Le syndrome sera placé sur les premiers bits de la variable renvoyée. Vérifiez que les messages m issus d'un mot encodé par la matrice G_s ont bien un syndrome nul. Utilisez les opérateurs logiques binaires. Il n'est pas demandé de faire un code fonctionnant pour d'autres matrices.
Exemple : `decode(0b1010101010101010)` devrait renvoyer `0b0001001100100000`, `0b00010011001` étant le syndrome.

Question 10.1 Observer le syndrome obtenu lors de l'ajout d'une, deux ou trois erreurs identiques sur des mots différents. Que remarquez-vous ?

Question 10.2 Ajoutez à votre fonction `decode` la capacité de corriger les erreurs via le système de décodage par syndrome vu en cours. Votre fonction doit être en mesure de corriger autant d'erreurs que la distance du code le permet. Détaillez la manière donc vous avez conçu votre système de décodage par syndrome dans votre rapport. Essayez de trouver une méthode de décodage efficace.
Exemple : `decode(0b0011101001001010)` devrait renvoyer `1101101001001010`.

Question 10.3 Montrez à l'aide de quelques exemples de votre choix que ce code est en mesure de corriger jusqu'à 3 erreurs, mais pas 4. Que se passe-t-il quand 8 erreurs surviennent ?

2.4 Le code simplexe en pratique : le simplexe polynomial

Question 11 Le code simplexe est également un code polynomial pouvant être défini par $P = x^{11} + x^{10} + x^9 + x^8 + x^6 + x^4 + x^3 + 1$. Plus précisément, P forme un code équivalent à celui décrit par G_s. Bien qu'a priori complexe, une division polynomiale dans F_2 peut être implémentée dans un registre à décalage relativement simple, rendant les code polynomiaux extrêmement répandus en pratique.
Dessinez le registre à décalage permettant de calculer le reste de la division polynomial d'un message reçu par ce polynomial générateur.

Question 12. Implémentez une fonction `unsigned short mod_poly_simplex(unsigned short m)` simulant le comportement du registre à décalage dessiné à la question précédente. La fonction prendra en argument le mot à encoder, et renverra les bits de contrôle (i.e., le modulo du message par le polynôme générateur).
Note : Vous pouvez vérifier que votre code est correct en vérifiant que la distance est la même que celle du code G_s. Vous pouvez également utiliser cet outil : `http://www.ee.unb.ca/cgi-bin/tervo/polygen2.pl?d=4&p=111101011001&s=1&c=1&a=0`.
Exemple : `mod_poly_simplex(0b1110000000000000));`
renverra `0b101100100010000` (`10110010001` étant le modulo résultat).

Question 13. Comme les codes polynomiaux sont linéaires, il est possible de faire du décodage par syndrome. Le syndrome est dans ce cas le modulo calculé, et ne dépend également que de l'erreur. Cependant, le code simplexe choisi ici n'est pas uniquement un code polynomial, mais

également un code cyclique : chaque mots du code valide peut-être obtenu en *shiftant*/décalant un autre mot valide. De ce fait, les codes cycliques peuvent être caractérisés par un mot uniquement. Les codes cycliques présentent un avantage pratique non-négligeable : la correction des mots reçus peut s'effectuer facilement en hardware, via une méthode plus élégante que les look-ups utilisés pour le décodage par syndrome [4]. L'algorithme de correction pour code cyclique peut se présenter comme suit :

1. Si le nombre de bits à 1 du syndrome (le reste de la division de m par P) est inférieur à 3 (le nombre d'erreurs corrigeables), alors le syndrome est le vecteur d'erreur. Il suffit donc de l'ajouter au message.
2. Sinon, faire un décalage circulaire de m d'un cran. Par exemple, `101110010111010` devient `011100101110101`. Retourner à l'étape 1.
3. Une fois le mot corrigé, le re-décaler à sa position originelle. Par exemple, si l'étape 2 a été effectuée 3 fois, il est nécessaire de re-décaler le mot 3 fois dans le sens inverse (toujours de manière circulaire) pour retrouver le mot de départ (corrigé).

Implémentez ce système de correction d'erreur pour code cyclique.
Votre fonction `unsigned short decode_poly(unsigned short m)` prendra en argument un message, et renverra le message corrigé. Montrez sur des exemples que votre code fonctionne. Pour les intéressés, l'article [4] illustre les circuits encodant cet algorithme pour certains codes spécifiques.

2.5 Codage & Décodage

Question 14. Le fichier `transmittedmessage` contient un message ayant été encodé via le polynôme P. Ce dernier a ensuite subi plusieurs erreurs (jusque 3 par blocs). À l'aide de votre fonction `decode`, corrigez le message.
Note : Pour encoder le message, chaque octet a été découpé en deux séquences de 4 bits chacune, qui ont été encodées à l'aide de P. Par exemple, l'octet `0b11101111` a été découpé en `0b1110` et `0b1111`. Une fois encodé, l'octet devient **1110**10110010001 et **1111**01011001000. Vous devez donc lire l'octet, corriger les erreurs, puis extraire les données.
Remarque : Le fichier est en réalité une image jpg.

Bonus. Encodez une suite de caractères (ou une image) de votre choix via le polynôme P. Ajoutez ensuite des erreurs à votre message (trois par bloc au maximum) et écrivez le dans un fichier. Joignez votre fichier à votre rendu de projet !

3 Rappel : Shift et opérateurs binaires

3.1 Décalage bit à bit (shift)

On rappelle que pour décaler tous les bits d'un nombre n de p bits vers le coté de poids fort, on utilise l'opérateur «. Pour le décalage vers les bits de poids faible on utilise l'opérateur ». Par exemple si n = 46 est un nombre de type char, alors on peut le représenter en binaire sous la forme 00101110.

```
n    : 00101110
n>>1 : 00010111
n<<1 : 01011100
```

En particulier, 1 « n représente le nombre 2^n et n » 1 le nombre $\frac{n}{2}$.

3.2 Opérateurs binaires

Les opérateurs de bits exécutent les opérations logiques ET, OU, OU exclusif (XOR) et NON sur tous les bits, pris un à un, de leurs opérandes entières. Le tableau ci-dessous résume le fonctionnement de ces opérateurs binaires.

& représente le ET logique et retourne 1 si les deux bits de même poids sont à 1. `01100000` & `01000000` renverra 01000000.

| représente le logique et retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux). `01100000` & `0000011` renverra 01100011.

$\wedge$ représente le OU exclusif et retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux). `01000000` $\wedge$ `11000001` renverra `10000001`.

$\sim$ représente le NON logique et inverse tous les bits. $\sim$ `00000000` renverra `11111111`.

Bibliographie

[1] Brian kernighan's algorithm. https://iq.opengenus.org/brian-kernighan-algorithm/. (Accessed on 03/13/2022).

[2] Jacques hadamard — wikipédia. https://fr.wikipedia.org/wiki/Jacques_Hadamard. (Accessed on 03/11/2022).

[3] Mariner 9 — wikipédia. https://fr.wikipedia.org/wiki/Mariner_9. (Accessed on 03/11/2022).

[4] T. Kasami. A decoding procedure for multiple-error-correcting cyclic codes. *IEEE Transactions on Information Theory*, 10(2) :134–138, 1964.