Correction d'erreurs et compression

INGRID DAUBECHIES
Princeton University

1. Comment se fait-il qu'un CD griffé puisse encore être joué sans problème ?

Sur un ordinateur ou sur un CD, l'information est enregistrée sous forme de zéros et de uns (voir figure 1). Si un CD est endommagé ou si la mémoire d'un ordinateur est atteinte par un rayonnement cosmique, ou s'il y a un disfonctionnement, alors des zéros peuvent être lus comme des uns, ou vice--versa. Comment pouvons-nous nous protéger de ceci ? Nous souhaitons

- Détecter des erreurs
- Corriger des erreurs

Commençons simplement. Imaginons que le signal soit

$$1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots$$

Après un événement qui altère les données il devient

$$1 \quad 0 \quad 1 \quad 1 \quad \underset{\equiv}{1} \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots$$

Il est impossible de détecter l'erreur.

Si, au lieu d'enregistrer la suite d'origine, nous doublons chaque bit, alors nous enregistrons

$$11 \quad 00 \quad 11 \quad 11 \quad 00 \quad 11 \quad 00 \quad 00 \quad 11 \quad 11 \quad 00 \quad \ldots$$

Adresse de l'auteur : Princeton University, Department of Mathematics and Program in Applied and Computational Mathematics, Fine Hall, Washington Road, Princeton, NJ 08544-1000, États-Unis d'Amérique
Texte traduit de l'anglais par Charlotte Bouckaert, UREM ULB

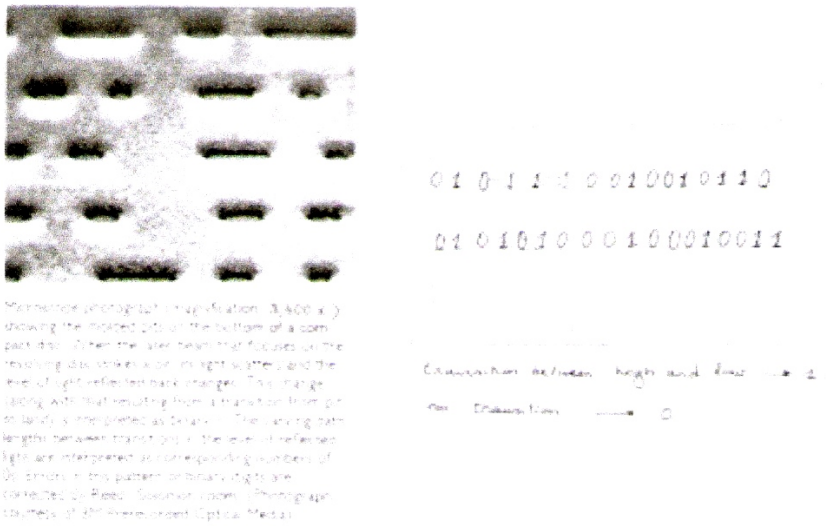

FIG. 1 – Comment les signaux sont enregistrés sur un CD

Après altération, certains bits ont « basculé » :

$$11 \quad 0\underline{\underline{1}} \quad 11 \quad 11 \quad 00 \quad 11 \quad 00 \quad 00 \quad 1\underline{\underline{0}} \quad 11 \quad 00 \quad \ldots$$

Maintenant nous pouvons voir que des erreurs ont été commises. La chaîne devrait être formée de paires de 00 ou 11 (les deux *mots de code* autorisés), mais il y a d'autres paires qui se sont immiscées, indiquant par là que des erreurs ont été commises ; mais nous n'avons aucun moyen de corriger ces erreurs : la paire 01 peut aussi bien provenir de 00 dans laquelle le deuxième 0 a été basculé en 1 que de 11 dont le premier 1 a été basculé en 0.

Si nous répétons chaque chiffre, non pas une fois mais deux, alors on obtient

$$111 \quad 000 \quad 111 \quad 111 \quad 000 \quad 111 \quad 000 \quad 000 \quad 111 \quad 111 \quad 000 \quad \ldots$$

Après altération, la chaîne pourrait ressembler à

$$111 \quad \underline{\underline{1}}00 \quad 111 \quad 111 \quad 000 \quad 1\underline{\underline{0}}1 \quad 000 \quad 000 \quad 111 \quad \underline{\underline{0}}11 \quad 000 \quad \ldots$$

Maintenant voyons non seulement où les erreurs se sont produites mais nous pouvons les corriger.

Tab. 1 – Table des erreurs sur 1 bit

Mot de code non codé	00001	01010	10100	11111
Altérations possibles sur 1 bit	10001	11010	00100	01111
	01001	00010	11100	10111
	00101	01110	10000	11011
	00011	01000	10110	11101
	00000	01011	10101	11110

- 100 est plus proche de 000 (1 bascule) que de 111 (2 bascules) ; ceci signifie qu'à moins d'avoir beaucoup de malchance 100 doit provenir de 000
- 101 doit provenir de 111
- 011 doit provenir de 111

Ainsi, en utilisant trois fois plus de mémoire, nous sommes capables de corriger les erreurs occasionnelles de bascule. Mais ceci demande une augmentation énorme de la capacité de mémoire. Serions-nous capables de concevoir une correction d'erreurs avec un schéma différent qui consommerait moins de mémoire ?

Nous pouvons, par exemple, remplacer chaque deux bits par une chaîne ou un mot de code, selon la règle suivante :

$$\begin{aligned} 00 &\longrightarrow 00001 \\ 01 &\longrightarrow 01010 \\ 10 &\longrightarrow 10100 \\ 11 &\longrightarrow 11111. \end{aligned}$$

Le message d'origine

$$1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad \ldots$$

devient

$$10100 \quad 11111 \quad 01010 \quad 00001 \quad 11111 \quad \ldots$$

Sommes-nous capables de corriger dans ce cas-ci ?

Pour nous assurer que nous sommes toujours capables de corriger les erreurs simples (un seul bit basculé) dans un mot de code, nous pouvons procéder de la manière suivante : dresser la liste des mots de code non

altérés, pour chaque mot de code dresser la liste des altérations possibles lors de la bascule d'un seul bit dans le mot de code. Nous obtenons alors la figure 2, où aucune chaîne de 5 bits ne se présente deux fois. Cela signifie que l'on peut retrouver le mot de code d'origine à partir de toute altération de 1 bit. Ce schéma est donc un code correcteur qui multiplie la mémoire nécessaire par un facteur 2,5, au lieu du facteur trois que nous avions précédemment et que nous sommes toujours capables de corriger les erreurs de bascule sur un seul bit. (Notons que nous devons supposer que les erreurs ne se produisent pas de manière trop rapprochée. Dans notre premier schéma, nous pouvions corriger les erreurs, pour autant qu'il y ait au plus une erreur dans chaque mot de code de 3 bits consécutifs. Dans le schéma que nous discutons maintenant, nous pouvons corriger les erreurs pour autant qu'il y ait au plus une erreur dans chaque mot de code de 5 bits consécutifs. De tels schémas ne sont utilisés que quand les erreurs se produisent avec une fréquence bien inférieure à celle-ci, comme dans le cas des perturbations de la mémoire d'un PC par les rayonnements cosmiques et donc ce n'est pas tellement important).

Nous pouvons aussi regarder le code correcteur que nous venons de construire de la manière suivante : deux mots de codes quelconques diffèrent au moins en trois positions.

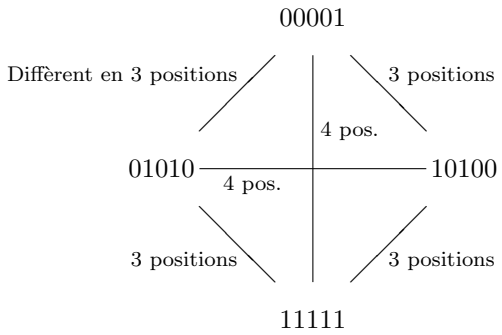

Fig. 2 – Les mots de code diffèrent en trois positions au moins

Comme les altérations des mots de code diffèrent de celui-ci en une seule position, cela signifie que les altérations de deux mots de code différents doivent être différentes, comme nous allons le montrer dans la démonstration suivante.

Les chaînes de 5 bits **mot-de-code1** et **altération1** ont au moins quatre bits identiques. Voir par exemple la figure 3.

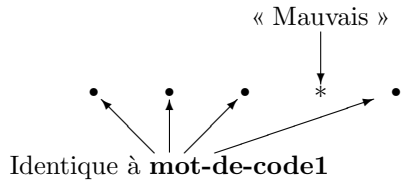

FIG. 3 – Un mot de code et une altération de celui-ci diffèrent d'une seule position (1)

Les chaînes de 5 bits **mot-de-code2** et **altération2** ont au moins quatre bits identiques. Voir par exemple la figure 4.

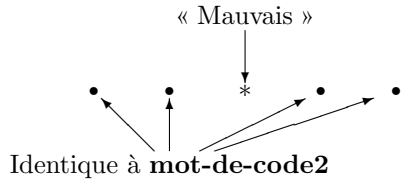

FIG. 4 – Un mot de code et une altération de celui-ci diffèrent d'une seule position (2)

Si deux altérations sont identiques, alors tous leurs bits sont identiques. Il y a donc trois bits identiques aux mêmes positions entre les chaînes mot--de-code1 et mot-de-code2 (dans l'exemple ci-dessus, les positions 1, 3 et 5). Il n'est donc pas possible que mot-de-code1 soit différent de mot-de-code2 puisque deux mots de code diffèrent en au moins trois positions.

Cependant, on n'utiliserait pas un tel code dans la pratique, parce qu'il gaspille de la place. Que voulons-nous dire par là ? Considérons la liste totale de tous les mots de code et de leurs altérations par bascule de un seul bit. Dans notre cas, cette liste comporte 24 entrées : il y a quatre mots de code de 5 bits et 5 altérations par mot de code, ce qui nous amène à $4 \times (1+5) = 24$. D'autre part, il y a $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$ chaînes de 5 bits possibles. Notre liste de 24 mots de code et altérations n'utilise pas toutes les chaînes possibles. On pourrait utiliser les huit chaînes restantes pour les erreurs de bascule de 2 bits sur certains mots de code (par exemple, 00110 est une

altération de 01010 par une bascule de 2 bits), mais ceci ne nous aide pas, puisque le code ne pourrait pas corriger toutes les erreurs de bascule de 2 bits (par exemple 11110 pourrait être une altération de 01010 par une bascule de 2 bits, mais aussi une altération de 11111 par une bascule de 1 bit).

Voyons maintenant si le code qui triple chaque bit gaspille de la place. L'encodage consiste en la correspondance

$$\begin{aligned} 0 &\longrightarrow 000 \\ 1 &\longrightarrow 111. \end{aligned}$$

Dans ce cas, voici la liste des mots de code et de leurs altérations :

Tab. 2 – Table des erreurs sur 1 bit

Mot de code non codé	000	111
Altérations possibles sur 1 bit	001	110
	010	101
	100	011

Cette liste comporte huit entrées et elle épuise **toutes** les chaînes possibles de trois bits (le nombre total de possibilités est de $2^3 = 8$). Un code pour lequel la liste des mots de code et de leurs altérations utilise tout l'espace disponible est appelé un code « parfait ». L'exemple du code avec des mots de code de 5 bits dont nous avons discuté précédemment n'est pas un code parfait. Les codes parfaits ont une jolie interprétation géométrique. Considérons les mots de code à trois bits et leurs altérations et identifions chacun d'eux à un sommet d'un cube à trois dimensions (voir la figure 5).

Les deux mots de code sont les sommets 000 et 111 et leurs altérations reliées au mot de code forment une sorte de tripode (voir la figure 6). Ces deux tripodes se correspondent et recouvrent tous les sommets du cube.

Pour le code à cinq bits que nous avons examiné, la représentation géométrique équivalente serait un cube à 5 dimensions (que nous ne pouvons pas dessiner). Chaque mot de code est au centre d'une étoile à 5 branches, mais les quatre étoiles ne recouvrent pas tous les sommets du cube. Nous ne pouvons pas dessiner ceci, mais nous pouvons donner la liste de tous les sommets et identifier les « étoiles à 5 branches » dans la table ci-dessous. Nous

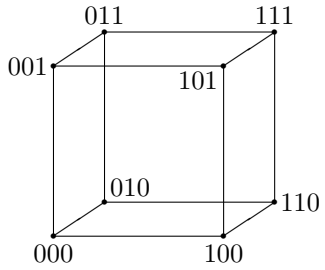

FIG. 5 – Les mots de code sur le cube

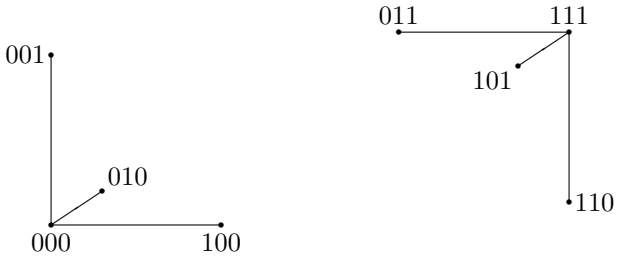

FIG. 6 – Les deux tripodes se correspondent

écrivons toutes les chaînes de 5 bits possibles en 6 colonnes, selon le nombre de 1 qu'elles comportent (première colonne : pas de 1, deuxième colonne : un seul 1, troisième colonne : deux 1, etc.). Nous entourons chaque mot de code avec un cadre différent (ovale, double ovale, rectangle en trait plein, rectangle en pointillés) et nous relions chaque mot de code à ses altérations dues aux erreurs de bascule de 1 bit. Nous encadrons les altérations avec un cadre de même type que celui du mot de code. Clairement, les étoiles à cinq branches ne se recouvrent pas, mais elles ne couvrent pas toute la table (voir figure 7).

Il y a huit groupes de cinq chiffres, c'est-à-dire huit sommets du cube à cinq dimensions qui ne sont pas couverts !

Le code de Hamming est un code qui améliore cette situation. Ici la correspondance remplace des chaînes de quatre bits par des mots de code

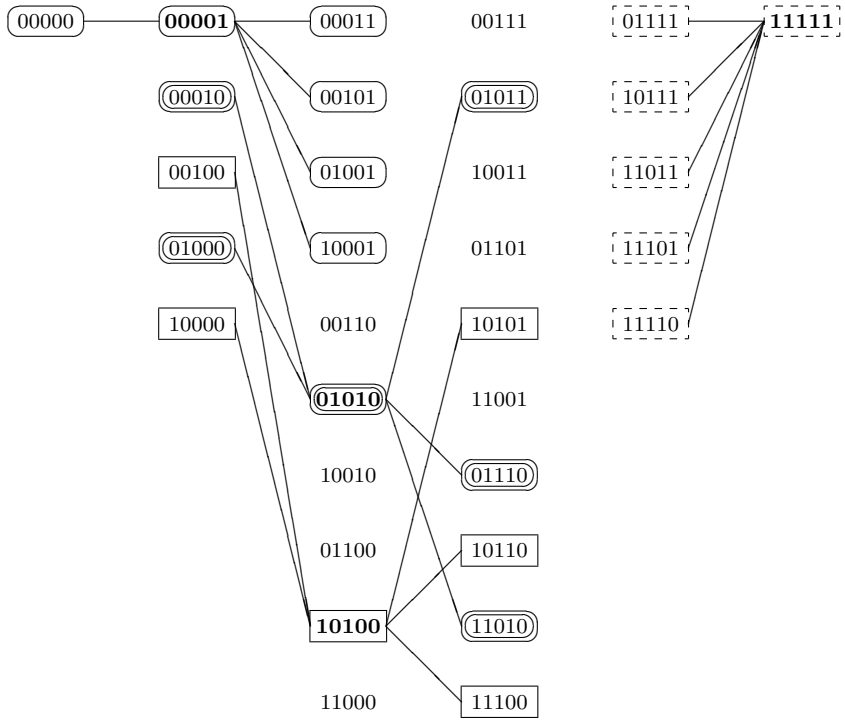

FIG. 7 – Les mots de code reliés à leurs altérations

de sept bits.

Message		Mot de code
0000	$\longrightarrow$	0000000
0001	$\longrightarrow$	0001011
0010	$\longrightarrow$	0010111
0100	$\longrightarrow$	0100101
1000	$\longrightarrow$	1000110
1100	$\longrightarrow$	1100011
1010	$\longrightarrow$	1010001
1001	$\longrightarrow$	1001101
0110	$\longrightarrow$	0110010
0101	$\longrightarrow$	0101110
0011	$\longrightarrow$	0011100
1110	$\longrightarrow$	1110100
1101	$\longrightarrow$	1101000
1011	$\longrightarrow$	1011010
0111	$\longrightarrow$	0111001
1111	$\longrightarrow$	1111111

(Nous utilisons la même convention que dans « For All Practical Purposes » et dans la vidéo projetée en classe ([1]), où la règle pour construire des mots de code est expliquée à l'aide de diagrammes de Venn.)

À nouveau, tous les mots de code diffèrent en au moins trois positions et ce code permet de corriger les erreurs de bascule de un bit. Ce code « vit » maintenant en dimension sept et chaque étoile dont le centre est un mot de code, relié à ses sept altérations de un bit, comporte huit éléments. Nous avons 16 mots de code en tout et les étoiles occupent $16 \times 8 = 128$ sommets du cube de dimension sept. Ce cube a exactement 2^7 sommets. Les étoiles occupent tous les sommets du cube et le code de Hamming est donc un code **parfait**.

Il a d'autres jolies propriétés.

Dans le code de Hamming, il y a un moyen simple de calculer la correspondance entre une chaîne de quatre bits et le mot de code de sept bits. (Nous pourrions, bien entendu, regarder dans une table, mais c'est beaucoup plus simple si nous ne procédons pas de cette manière. Pour 16 mots de code, c'est encore faisable, mais pour des codes plus grands comme les codes de Reed-Solomon, qui ont 256 mots de code ou plus, ...)

([1]) Voir le texte correspondant dans *MathAlive*

Voici comment on calcule les mots de code de 7 bits correspondants aux chaînes de quatre bits $b_1b_2b_3b_4$:

$$b_1b_2b_3b_4 \longrightarrow b_1b_2b_3b_4(b_1 \oplus b_2 \oplus b_3)(b_1 \oplus b_3 \oplus b_4)(b_2 \oplus b_3 \oplus b_4)$$

(le symbole $\oplus$ indique l'addition de parité : $1 \oplus 0 = 1$, $1 \oplus 1 = 0$, $1 \oplus 1 \oplus 1 = 1$; on parle aussi de somme directe). Ceci rend le code de Hamming linéaire : on obtient les mots de code à partir des données non codées grâce à un calcul linéaire simple. Cela signifie que pour chaque mot de code

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7$$

les trois contraintes suivantes sont vérifiées :

$$\begin{aligned} b_1 \oplus b_2 \oplus b_3 \oplus b_5 &= 0 \\ b_1 \oplus b_3 \oplus b_4 \oplus b_6 &= 0 \\ b_2 \oplus b_3 \oplus b_4 \oplus b_7 &= 0. \end{aligned}$$

Ceci nous conduit à un algorithme de décodage très simple. Imaginons que

$$\tilde{b_1} \quad \tilde{b_2} \quad \tilde{b_3} \quad \tilde{b_4} \quad \tilde{b_5} \quad \tilde{b_6} \quad \tilde{b_7}$$

soit un mot qui aurait pu être altéré. Nous pouvons alors calculer les trois « contrôles de parité » :

$$\begin{aligned} p_1 &= \tilde{b_1} \oplus \tilde{b_2} \oplus \tilde{b_3} \oplus \tilde{b_5} \\ p_2 &= \tilde{b_1} \oplus \tilde{b_3} \oplus \tilde{b_4} \oplus \tilde{b_6} \\ p_3 &= \tilde{b_2} \oplus \tilde{b_3} \oplus \tilde{b_4} \oplus \tilde{b_7} \end{aligned}$$

Suivant les valeurs obtenues par p_1, p_2, p_3 nous allons procéder différemment :

1. Si $p_1 = p_2 = p_3 = 0$, alors la chaîne $\tilde{b_1}\tilde{b_2}\tilde{b_3}\tilde{b_4}\tilde{b_5}\tilde{b_6}\tilde{b_7}$ est correcte et l'on obtient la chaîne d'origine en laissant tomber $\tilde{b_5}\tilde{b_6}\tilde{b_7}$.
2. Si l'un des p_j est égal à 1, et que les deux autres sont égaux à 0, alors on cherche le $\tilde{b_l}$ qui intervient dans p_j et pas dans les deux autres et on le bascule de manière à obtenir le mot de code correct.
   Exemple : $p_2 = 1$ et $p_1 = p_3 = 0$. Le seul $\tilde{b}$ qui intervienne dans p_2 et pas dans p_1 et p_3 est $\tilde{b_6}$. Donc, il faut basculer $\tilde{b_6}$ pour obtenir le mot de code correct. Ensuite, il suffit de laisser tomber $\tilde{b_5}\tilde{b_6}\tilde{b_7}$ pour

retrouver la chaîne de quatre bits. (En réalité, on peut directement laisser tomber $\tilde{b_5}\tilde{b_6}\tilde{b_7}$, parce que le bit basculé est toujours $\tilde{b_5}$ ou $\tilde{b_6}$ ou $\tilde{b_7}$ qu'on devra de toute manière laisser tomber.)

3. Si un seul des p_j est égal à 0 et que les deux autres sont égaux à 1, alors on cherche le bit $\tilde{b}$ qui intervient dans les deux bits de parité incorrects et pas dans celui qui est correct, et on bascule ce bit.
   Exemple : $p_1 = p_2 = 1$ et $p_3 = 0$. Le seul $\tilde{b}$ commun à p_1 et p_2 et qui n'intervienne pas dans p_3 est $\tilde{b_1}$. On bascule donc $\tilde{b_1}$ pour obtenir le mot de code correct.
4. Si $p_1 = p_2 = p_3 = 1$, alors on bascule $\tilde{b_3}$.

Donc, corriger des erreurs est très simple.

Notons que dans ce code de Hamming, une chaîne de quatre bits est encodée par un mot de code de 7 bits. Ce code multiplie la quantité de mémoire nécessaire par $\frac{7}{4} = 1{,}75$. Dans l'exemple précédent, qui accolait aussi trois bits aux chaînes à encoder, et qui transformait les chaînes de 2 bits en chaînes de 5 bits, le facteur de multiplication de la mémoire était de 2,5. C'est vraiment un code qui gaspille.

Que se passerait-il si on accolait quatre bits ? Pouvons-nous construire un code parfait ? Quelle taille aurait-il ? Il y a en effet des codes de Hamming « plus grands », qui sont encore parfaits et accolent quatre bits pour faire les mots de codes, ou cinq bits (avec un code encore plus grand), etc. Nous n'en discuterons pas ici en détail. Mais nous pouvons facilement calculer combien de bits il y aura dans les mots de code et dans les chaînes non codées. Examinons le cas où nous accolons quatre bits à une chaîne non codée pour la transformer en mots de code. Imaginons que nous commencions par des chaînes de n bits. Il y a 2^n chaînes possibles et donc il nous faut 2^n mots de code. Les mots de code ont une longueur de $(n + 4)$ bits (puisque nous accolons 4 bits) et ils « vivent » donc dans un espace dans lequel il y a 2^{n+4} chaînes possibles. Chaque mot de code est relié à ses altérations de un bit. L'ensemble forme une « étoile » qui comporte $1 + (n + 4) = n + 5$ éléments. Pour rendre le code parfait il faut que

$$\underset{\substack{\uparrow \\ \text{taille de} \\ \text{chaque étoile}}}{(n+5)} \quad \times \quad \underset{\substack{\uparrow \\ \text{nombre de} \\ \text{mots de code}}}{2^n} \quad = \quad \underset{\substack{\uparrow \\ \text{nombre total de chaînes} \\ \text{de longueur}(n+4)}}{2^{n+4}}$$

ou $(n + 5) = 2^4$. Ceci est vérifié pour $n = 2^4 - 5 = 16 - 5 = 11$. Nous avons alors un code parfait qui remplace des chaînes de 11 bits par des

mots de code de 15 bits et la quantité de mémoire nécessaire est multipliée par $\frac{15}{11} \approx 1{,}4$. Encore mieux ! Bien sûr, nous ne pouvons nous permettre qu'une seule erreur tous les 15 bits au lieu d'une tous les 7 bits comme précédemment.

À nouveau un joli code linéaire qui peut être facilement décodé. C'est code de Hamming 11–15. On peut construire toute une hiérarchie de codes de Hamming de cette manière. Ils sont beaucoup utilisés pour protéger les données des ordinateurs.

Bien entendu, on peut aussi construire des codes qui peuvent corriger **deux** erreurs sur un mot de code. Dans ce cas, deux mots de code quelconques doivent être différents en **cinq** positions.

Les codes utilisés sur les CD

Beaucoup de codes correcteurs d'erreurs ont été développés après le travail de Hamming. Environ dix ans après, Irving Reed et Gus Solomon ont développé ce qui est maintenant connu comme les codes de Reed-Solomon, utilisés pour coder des CD et dans la transmission d'information venant de l'espace (comme les images de Voyager, par exemple).

L'essentiel dans les codes de Reed-Solomon, c'est l'usage d'un autre **alphabet**.

Dans les codes de Hamming, nous travaillons uniquement avec les symboles 0 et 1. Nous aurions pu tout aussi bien les écrire comme a et b, et dans ce cas une chaîne à encoder pourrait être :

$$abbababaabaabbbaabaababbbbab\ldots$$

Changeons maintenant de point de vue, et cessons de considérer les a et b pris isolément comme les briques avec lesquelles on construit une chaîne. Choisissons plutôt les quatre paires aa, ab, ba, bb. La chaîne devient

$$ab \quad ba \quad ba \quad ba \quad ab \quad aa \quad bb \quad ba \quad ab \quad aa \quad ba \quad bb \quad bb \quad ab\ldots$$

Nous choisissons maintenant de travailler avec un plus grand alphabet, en écrivant pour chaque paire A au lieu de aa, B au lieu de ab, C au lieu de ba et D au lieu de bb. La chaîne devient alors :

$$B \quad C \quad C \quad C \quad B \quad A \quad D \quad D \quad C \quad B \quad A \quad C \quad D \quad D \quad B\ldots$$

Si on prend des blocs de lettres plus longs (que deux caractères), on agrandit encore l'alphabet.

Ce qui rend le code de Hamming particulièrement élégant, c'est qu'on peut facilement calculer les mots de code à partir des chaînes non codées, grâce à des additions simples en utilisant l'opération d'« addition de parité », plus communément appelée le « ou exclusif » ou le « xor ». Pour que ces nouveaux codes soient faciles à utiliser, on emploie une opération analogue au xor pour calculer les mots de codes et la correction d'erreurs (au lieu de devoir regarder dans une gigantesque table). Nous travaillons dans une collection finie de symboles dans laquelle certaines opérations spéciales sont définies. Les mathématiciens appellent ceci un **corps fini**. (Nous avons déjà rencontré des exemples de corps finis, quand nous avons considéré l'arithmétique modulaire ([2]) — quand on travaille avec les nombres « modulo 7 », on a un alphabet de lettres : 0, 1, 2, 3, 4, 5, 6, et nous savons comment donner un sens aux opérations d'addition et de multiplication sur cet alphabet.)

En pratique, les codes de Reed-Solomon utilisent des corps finis de 256 éléments. C'est comme s'ils avaient un « alphabet » de 256 « lettres ».

Un code RS qui corrige **une** lettre incorrecte est équivalent à un code binaire qui corrige **huit** erreurs de bascule de bits, car on peut imaginer chaque lettre de l'alphabet de 256 lettres comme une chaîne de 8 bits. (Nous devons nuancer cette affirmation : dans la représentation binaire des mots de code RS, huit erreurs consécutives ne se produisent pas nécessairement sur les huit bits correspondant à *une* lettre du code RS. Elles pourraient correspondre à deux lettres adjacentes du code RS et chevaucher leurs deux représentations en 8 bits. Plus précisément, [?] erreurs binaires consécutives peuvent correspondre à [?] lettres consécutives incorrectes du code RS.)

Pour encoder le son sur un CD audio, on utilise plusieurs stratégies.

1. Le code RS utilisé permet de corriger jusqu'à 40 lettres consécutives erronées du code RS.

2. De plus, l'information est dispersée par entrecroisement. Supposons que la chaîne des mots codés soit

$$c_1c_2c_3c_4c_5c_6c_7c_8c_9c_{10}c_{11}c_{12}c_{13}c_{14}c_{15}c_{16}c_{17}c_{18}c_{19}\ldots$$

([2]) Voir le texte correspondant dans *Math Alive*

Au lieu de les écrire dans l'ordre original, on entrecroise les mots de la manière suivante :

$$c_1c_{11}c_2c_{12}c_3c_{13}c_4c_{14}c_5c_{15}c_6c_{16}c_7c_{17}c_8c_{18}c_9c_{19}c_{10}\dots$$

Si, dans le nouveau code, deux lettres consécutives sont mal lues, cela représente en réalité deux erreurs séparées par dix symboles dans la suite d'origine ! De même, quatre erreurs consécutives correspondent à deux blocs de deux erreurs consécutives dans la suite d'origine, séparés par 10 symboles consécutifs. Si l'on entrecroise encore plus, on peut réduire d'autant plus la longueur des blocs erronés.

Pour les applications CD, le facteur d'entrecroisement est de 5, et l'information est fortement dispersée. Il en découle que si vous faites N erreurs consécutives sur la représentation entrecroisée, en réalité vous avez fait des groupes d'erreurs de longueur $N/5$ séparés l'un de l'autre par 5 symboles dans la représentation d'origine.

3. Enfin, chaque lettre de l'alphabet RS n'est pas encodée sur 8 bits (ce qui correspond à 256 possibilités), mais sur 10 bits avec un petit code correcteur d'erreurs (avec un schéma différent) inséré à ce niveau individuel.

La rayure que nous avons faite en classe sur le CD mesurait environ 0,25 mm de large. Comme 0,01 mm correspond à 8 bits, nous avons détruit environ 200 bits consécutifs. En développant le code 8–10 « extérieur », cela signifie que nous avons détruit $160 = 8/10 \times 200$ bits dans le langage RS, ou 20 mots RS (puisque chaque mot RS occupe 8 bits). À cause de l'entrecroisement, ceci correspond à 5 groupes distants de 5 lettres et formés de 4 lettres RS consécutives. Ceci n'est rien pour le code correcteur RS qui a été conçu pour corriger des erreurs bien plus longues que celle-ci !

2. Détecter les erreurs dans les codes-barre, les numéros de ticket d'avion, et encore d'autres choses

Dans les applications que nous avons discutées jusqu'à présent (mémoire d'ordinateur, CD, transmission d'images depuis l'espace), nous nous occupions de messages extrêmement longs et nous voulions non seulement détecter des erreurs mais les corriger.

Voici maintenant un ensemble tout à fait différent d'applications, pour lesquelles on veut lire un numéro relativement court (mettons dix chiffres) et détecter s'il est correct ou non — en général, on ne se soucie pas de le corriger automatiquement. Si l'on détecte une erreur, on demande juste une confirmation (numéro de carte de crédit pour les commandes par téléphone) ou l'on a recours à une procédure manuelle basée sur d'autres informations (taper le code UPC à une caisse enregistreuse au supermarché, corriger le code postal ZIP basé sur l'adresse).

Pour beaucoup de ces applications, la capacité de détection de certaines erreurs est incluse dans le numéro lui-même. Examinons quelques-uns de ces schémas ainsi que leurs mérites respectifs.

2.1. Les Traveler's Checks

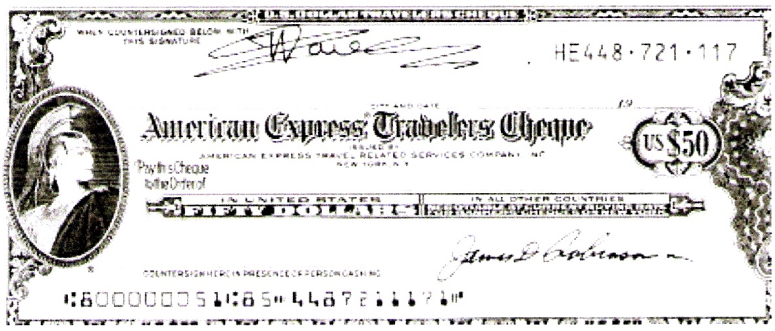

FIG. 8 – Les Traveler's Check d'American Express

Le numéro du chèque (voir figure 8) que l'on voit dans le coin supérieur gauche 448721117 est repris au bas du chèque avec un numéro ID 4487211171 :

4487211171
↑
ce chiffre supplémentaire est un chiffre de contrôle

Quelle est la relation du chiffre de contrôle avec les autres chiffres ? Faites la somme de tous les autres chiffres et prenez le reste de la division de cette

somme par 9 :

$$4+4+8+7+2+1+1+1+7 = 35 \equiv 8 \pmod 9.$$

Calculez ensuite $9-8=1$ (1 est le chiffre de contrôle). Ou, formulé autrement :

$$4+4+8+7+2+1+1+1+7+1 \equiv 0 \pmod 9.$$

La somme de tous les chiffres + chiffre de contrôle = multiple de 9.

Le chiffre de contrôle peut prendre les valeurs 0, 1, 2, ..., 8. Quelles erreurs pouvons-nous détecter ?

- Substitution de 0 par 9 et vice-versa, dans d'autres chiffres que le chiffre de contrôle : **NON**.
- Autres substitutions d'un seul chiffre par un autre : **OUI**.
- Inversion (c'est-à-dire, ...53... au lieu de ...35...) : **NON**.

La même méthode est utilisée pour les chèques de la US Postal Service.

2.2. Les tickets d'avion

Examinons maintenant le ticket d'avion (voir figure 9).

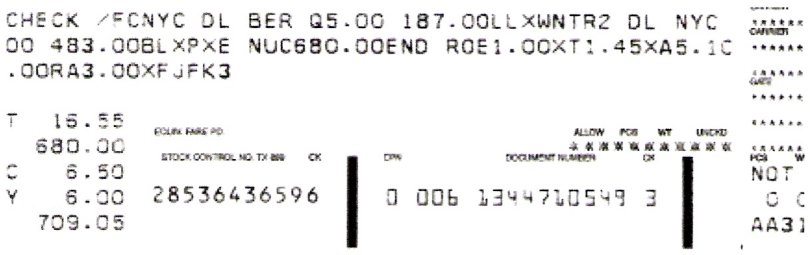

FIG. 9 – Ticket d'avion

N° de contrôle de stock Tx 889 CK
2853643659 6
↑
Le chiffre de contrôle est le reste de la division par 7 du nombre de dix chiffres

Le chiffre de contrôle peut prendre les valeurs 0, 1, ..., 6. Quelles erreurs peut-il détecter ?

- Les substitutions de 0 par 7 (et vice-versa), de 1 par 8 (et vice-versa), de 2 par 9 (et vice-versa) ne sont pas détectables sauf dans le chiffre de contrôle.
- Les autres substitutions sont détectables.
- Et les inversions ?

$$\begin{aligned} \text{correct :} &\ \ldots 34 \ldots C \\ \text{incorrect :} &\ \ldots 43 \ldots C \end{aligned}$$

Le chiffre de contrôle permet-il de détecter les erreurs d'inversion ? En d'autres termes, le chiffre de contrôle du deuxième nombre est-il égal à C ? Le premier nombre (sans le chiffre de contrôle) = (multiple de 7) + C. Le deuxième nombre (sans le chiffre de contrôle) = (premier nombre) + 9000 = (multiple de 7) + C + 1 (parce que le reste de la division de 9000 par 7 est égal à 1). Ceci signifie que le chiffre de contrôle doit être différent de C. Donc l'erreur est détectée. Cependant, toutes les inversions ne peuvent pas être détectées.

$$\begin{aligned} \text{correct :} &\ \ldots 18 \ldots C \\ \text{incorrect :} &\ \ldots 81 \ldots C \end{aligned}$$

Le premier nombre (sans le chiffre de contrôle) = (multiple de 7) + C. Le deuxième nombre (sans le chiffre de contrôle) = (premier nombre) + + 63 000 000 = (multiple de 7) + C. Cette inversion n'est pas détectée, parce qu'elle n'affecte pas le reste de la division par 7 et le chiffre de contrôle reste le même.

Ce système est utilisé par FedEx, UPS, AVIS et les agences de location de voitures.

2.3. Le système bancaire américain

$$\begin{array}{cccccccccc} 0 & 2 & 1 & 2 & 0 & 2 & 6 & 0 & & 9 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & \text{chiffre de contrôle} & \end{array}$$

Calculons

$$\begin{array}{rccccccccc} & 7a_1 & + 3a_2 & + 9a_3 & + 7a_4 & + 3a_5 & + 9a_6 & + 7a_7 & + 3a_8 & \\ = & 0 + & 6 + & 9 + & 14 + & 0 + & 18 + & 42 + & 0 & = 89 \equiv 9 \pmod{10}. \end{array}$$

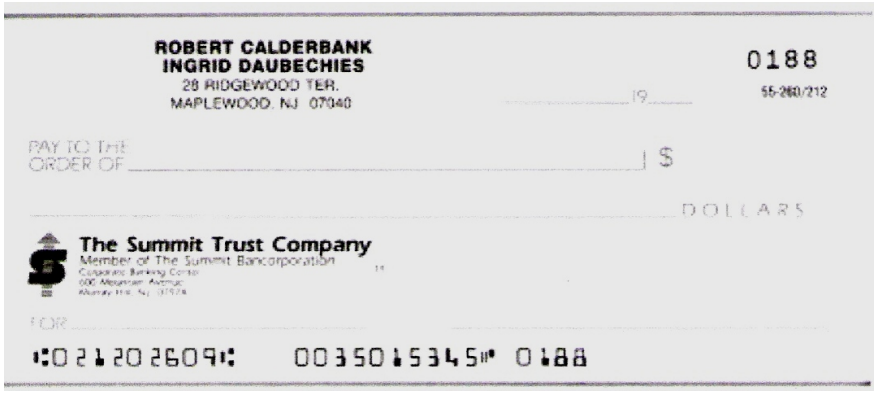

FIG. 10 – Chèque bancaire

Le chiffre de contrôle est donc 9.

Quelles erreurs peut-on détecter ?
- Erreurs d'un seul chiffre dans le chiffre de contrôle : toujours détectées.
- Erreurs d'un seul chiffre ailleurs ?

En première position :

$$\begin{array}{rcccccccc} \text{correct :} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{incorrect :} & a_1' & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

Alors $7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C$; $7a_1' + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C + 7(a_1' - a_1)$. L'erreur passera inaperçue uniquement si $7(a_1' - a_1)$ est un multiple de 10. Mais $a_1' - a_1$ peut prendre les valeurs de -9 à 9 (à l'exception de 0, puisque $a_1' \neq a_1$, il y a eu une erreur). Quand on multiplie ces valeurs par 7, on obtient les nombres

$$-63, -56, -49, -42, -35, -28, -21, -14, -7, 7, 14, 21, 28, 35, 42, 49, 56, 63$$

Il n'y a pas un seul multiple de 10 ! Toutes les erreurs en première position sont détectées. Le même argument s'applique pour la quatrième ou la septième position.

Qu'en est-il des erreurs en deuxième position ?

$$\begin{array}{rcccccccc} \text{correct :} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{incorrect :} & a_1 & a_2' & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

$7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C$;
$7a_1 + 3a_2' + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C +$
$+ 3(a_2' - a_2)$. À nouveau, $a_2' - a_2$ peut prendre les valeurs de -9 à 9 (à l'exception de 0, puisque $a_2' \neq a_2$, il y a eu une erreur) et multiplier par 3 donne une collection de toutes les valeurs possibles de $3(a_2' - a_2)$, dont aucune n'est un multiple de 10. Toutes les erreurs en deuxième (cinquième et huitième) position sont détectées.

Le même argument vaut pour les erreurs en troisième et sixième positions. Cette fois, on travaille avec $9(a_3' - a_3)$. Multiplier $a_3' - a_3$ par 9 avec $a_3' - a_3 \neq 0$ et compris entre -9 et 9, ne donne jamais un multiple de 10. Toutes les erreurs en troisième et sixième position seront détectées.

Et les erreurs d'inversion ? Par exemple :

$$\begin{array}{rcccccccc} \text{correct :} & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \\ \text{incorrect :} & a_2 & a_1 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & C \end{array}$$

$7a_1 + 3a_2 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C$;
$7a_2 + 3a_1 + 9a_3 + 7a_4 + 3a_5 + 9a_6 + 7a_7 + 3a_8 = (\text{multiple de } 10) + C +$
$+4(a_2 - a_1)$. Cette erreur **ne sera pas détectée** si $4(a_2 - a_1)$ est un multiple de 10. C'est le cas pour $a_2 - a_1 = 5$ ou -5. Ce sont les deux seuls cas. Exemples :

$$a_1 = 2 \quad a_2 = 7$$

ou

$$a_1 = 9 \quad a_2 = 4.$$

Et les erreurs d'inversion des autres chiffres consécutifs ?

$$\ldots a_2 a_3 \ldots\ldots$$

$$\ldots a_3 a_2 \ldots\ldots$$

n'est pas détectable si

$$3a_2 + 9a_3 - 3a_3 - 9a_2 = 6(a_3 - a_2)$$

est un multiple de 10. À nouveau, quand $a_3 - a_2 = 5$ ou -5.

L'inversion des troisième et quatrième chiffres

$$\ldots a_3 a_4 \ldots\ldots$$

$$\ldots a_4 a_3 \ldots\ldots$$

sera indétectable si

$$9a_3 + 7a_4 - 9a_4 - 7a_3 = 2(a_3 - a_2)$$

est un multiple de 10. À nouveau, lorsque $a_3 - a_4 = 5$ ou -5.

$$\left.\begin{array}{l} 7-3=4 \\ 3-9=-6 \\ 9-7=2 \end{array}\right\} \text{ ont le facteur 2 commun avec 10, base de l'arithmétique modulaire.}$$

$\uparrow$

Différence entre les poids consécutifs

Remarque : nous pouvons analyser le schéma du ticket d'avion de la même manière. Après tout diviser 6 483 839 228 par 7 revient à diviser

$$6 \times 1\,000\,000\,000 + 4 \times 100\,000\,000 + 8 \times 10\,000\,000 + 3 \times 1\,000\,000 \\ +8 \times 100\,000 + 3 \times 10\,000 + 9 \times 1000 + 3 \times 100 + 2 \times 10 + 8$$

par 7. Puisque

$$\begin{aligned} 10 &= \text{multiple de } 7 + \underline{3} \\ 100 &= \text{multiple de } 7 + \underline{2} \\ 1000 &= \text{multiple de } 7 + \underline{6} \\ 10\,000 &= \text{multiple de } 7 + \underline{4} \\ 100\,000 &= \text{multiple de } 7 + \underline{5} \\ 1\,000\,000 &= \text{multiple de } 7 + \underline{1} \\ 10\,000\,000 &= \text{multiple de } 7 + \underline{3} \\ 100\,000\,000 &= \text{multiple de } 7 + \underline{2} \\ 1\,000\,000\,000 &= \text{multiple de } 7 + \underline{6} \end{aligned}$$

Cela signifie que nous aurions pu regarder le reste modulo 7 comme la combinaison

$$6 \times \underline{6} + 4 \times \underline{2} + 8 \times \underline{3} + 3 \times \underline{1} + 8 \times \underline{5} + 3 \times \underline{4} + 9 \times \underline{6} + 3 \times \underline{2} + 2 \times \underline{3} + 8 \times \underline{1}$$

Plus généralement, si le nombre est

$$a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10},$$

alors le chiffre de contrôle modulo 7 est égal à la combinaison

$$6a_1 + 2a_2 + 3a_3 + a_4 + 5a_5 + 4a_6 + 6a_7 + 2a_8 + 3a_9 + a_{10}.$$

Procédons maintenant de la même manière pour voir si un chiffre erroné en sixième position, par exemple, est détectable.

$$\begin{array}{c} \ldots\ldots a_6 \ldots \\ \ldots\ldots a'_6 \ldots \end{array}$$

donne le même chiffre de contrôle si $4(a'_6 - a_6)$ est un multiple de 7. Ici, 4 n'a pas de facteur commun avec 7, ce qui nous arrange, mais $a'_6 - a_6$ peut prendre les valeurs de -9 à 9 (à l'exception de 0) et donc $a'_6 - a_6 = 7$ ou -7 sont des valeurs possibles... L'erreur est indétectable si nous substituons 0 pour 7 (et vice-versa), 1 pour 8 (et vice-versa), 2 pour 9 (et vice-versa).

2.4. Le système UPC

(Les codes-barres sur les produits alimentaires) Notre étude du code barres UPC comporte deux parties :

- La lecture du code
- Le chiffre de contrôle

Le code barres est une manière d'imprimer une représentation binaire des nombres grâce à des lignes épaisses ou minces et à des espaces étroits ou larges. Dans cette représentation, chaque chiffre décimal est encodé par 7 chiffres binaires. Voici comment nous devons lire le code barres :

- Fine ligne noire = 1
- Ligne noire épaisse = 11
- Ligne noire plus épaisse = 111
- Ligne noire très épaisse = 1111
- Espace étroit =0
- Espace large = 00
- Espace plus large =000
- Espace très large = 0000

Nous pouvons lire dans l'exemple ci-dessus (voir figure 11). Le code pour

FIG. 11 – Code UPC

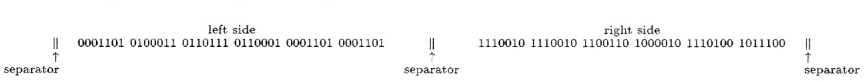

FIG. 12 – Lecture du code UPC

lire ceci en chiffres décimaux est le suivant :

	gauche	droite
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

(Ceci n'est pas le codage binaire des chiffres décimaux. Les groupes de zéros et de uns sont choisis de manière à donner au code des propriétés très spéciales. Voir plus loin.)

Dans l'exemple ci-dessus, nous avons 0 48500 00139 4. Le nombre encodé dans le code barres UPC comporte $6 + 6$ chiffres ou plutôt

1	+	5	+	5	+	1	chiffres.
↑		↑		↑		↑	
identifie le genre de produit		code du fabricant		code du produit		chiffre de contrôle	

Le chiffre de contrôle n'est pas toujours imprimé en chiffres décimaux, mais il est toujours présent dans le code barres.

Remarque :
- À gauche, chaque groupe de 7 zéros et uns comporte toujours un nombre **impair** de uns. À droite, chaque groupe de 7 zéros et uns comporte toujours un nombre **pair** de uns. La lecture automatique peut décider quand elle lit les choses à l'envers !
- À gauche : début = 0, fin = 1, pour tous les groupes. À droite : début = 1, fin = 0, pour tous les groupes (de manière à séparer les choses joliment).

Le chiffre de contrôle
Prenons les 11 premiers chiffres $a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}$ et calculons

$$3a_1 + a_2 + 3a_3 + a_4 + 3a_5 + a_6 + 3a_7 + a_8 + 3a_9 + a_{10} + 3a_{11} \equiv A \pmod{10} ;$$

alors le chiffre de contrôle C ou a_{12} est $a_{12} = C = 10 - A \pmod{10}$. Dans notre exemple,

$$3a_1 + a_2 + 3a_3 + a_4 + 3a_5 + a_6 + 3a_7 + a_8 + 3a_9 + a_{10} + 3a_{11} = \\ = 4 + 24 + 5 + 3 + 3 + 27 = 66,$$

donc $A = 6$ et $a_{12} = C = 4$.

2.5. L'ISBN (International Standard Book Number)

L'ISBN est le plus intelligent de ces codes de détection d'erreurs. Il détecte toutes les erreurs simples et toutes les inversions.

Tout numéro ISBN a la forme suivante :

$$\begin{array}{c} a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10} \\ \uparrow \\ C \end{array}$$

Comment peut-on trouver C à partir de $a_1 \dots a_9$? On calcule

$$10a_1 + 9a_2 + 8a_3 + 7a_4 + 6a_5 + 5a_6 + 4a_7 + 3a_8 + 2a_9 \equiv A \pmod{11} ;$$

alors $a_{10} = C = 11 - A \pmod{11}$.

Exemple : 0-7167-2378-6

$$10 \times 0 + 9 \times 7 + 8 \times 1 + 7 \times 6 + 6 \times 7 + 5 \times 2 + 4 \times 3 + 3 \times 7 + 2 \times 8 = 214 ;$$

$A = 5$, $a_{10} = 6$: OK !

Comme 11 est premier et que tous les poids (10, 9, 8, ..., 2) sont donc premiers avec 11, ceci détecte toutes les erreurs simples.

Et la détection des transpositions ? La différence de deux poids consécutifs est toujours égale à 1. Ne pas détecter une transposition entre le n^{e} et la $(n+1)^{\text{e}}$ position demande que $a_n - a_{n+1}$ soit un multiple de 11, ce qui est impossible. Donc toutes les transpositions sont détectées.

Seul inconvénient : parfois $a_{10} = 10$; dans ce cas, on le remplace par « X ».
Exemple : 0-273-00218-X.

2.6. Un autre code barres : le code postal ZIP aux USA

Il s'agit du code qu'on trouve sur beaucoup d'enveloppes de mailing (coupons réponse, par exemple voir figure 13). Chaque nombre ZIP+4 est représenté par $1 + 10 \times 5 + 1 = 52$ barres longues et courtes au bas de la lettre. (Voir par exemple la figure 13.)

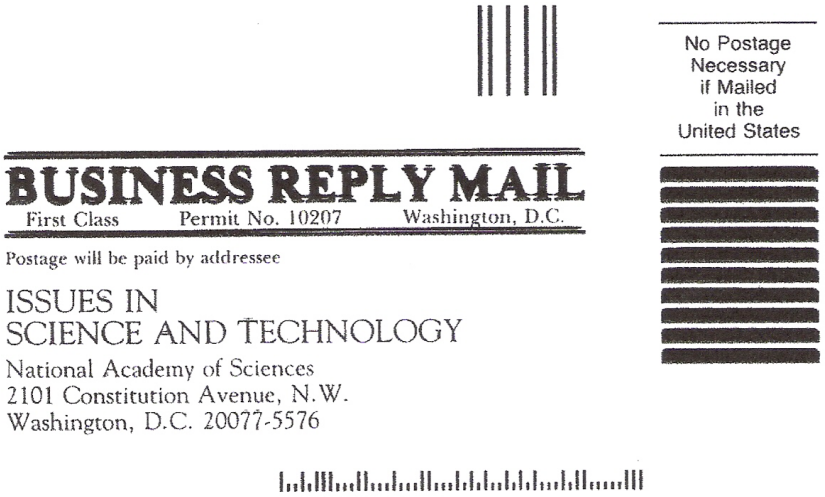

FIG. 13 – Code ZIP

Le dictionnaire pour lire ce code est le suivant :
- Les longues barres du début et de la fin sont juste des barres de garde.
- Chaque groupe de cinq barres encode un chiffre selon la correspondance suivante :

Decimal Digit	Bar Code	Binary Code
1		00011
2		00101
3		00110
4		01001
5		01010
6		01100
7		10001
8		10010
9		10100
0		11000

The Postnet bar code

FIG. 14 – Code ZIP

Le nombre ci-dessus est donc 2007755761. Il correspond au code ZIP+4 20077-5576. Le chiffre 1 à la fin est le chiffre de contrôle. Comment le construit-on ?

$$a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}$$
$$\uparrow$$
chiffre de contrôle

La somme $a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10}$ doit être un multiple de 10.
Dans l'exemple :

$$\underbrace{2+7+7+5+5+7+6+6}_{\text{total=39}}+1 \longrightarrow 40 \quad \text{OK !}$$

De plus, nous pouvons aussi corriger certaines erreurs ! Chaque groupe de cinq barres doit comporter **exactement** deux barres longues et trois courtes. Une erreur de lecture se produit quand une barre courte est prise pour une longue et vice-versa. Nous savons donc non seulement qu'il y a une erreur, mais nous savons aussi **où** elle s'est produite ! (Nous avons un test de

parité pour chaque groupe de cinq barres représentant un chiffre. Rappelez-vous les tests de parité des codes précédents !) Nous pouvons utiliser le chiffre de contrôle pour corriger l'erreur. Exemple :

FIG. 15 – Code ZIP

Ici, un groupe de cinq barres est formé de quatre barres courtes et d'une longue. Nous pouvons lire les autres chiffres et nous obtenons

$$2007?5671$$

On peut facilement retrouver le chiffre ? :

$$\underbrace{2+0+0+7+5+7+6+1}_{\text{total=33}}+?$$

Ce total doit être divisible par 10. Donc ? = 7. Des codes barres plus récents utilisent 12 chiffres :

$\underbrace{\text{ZIP}+4}_{\text{total=9}}+$ les deux derniers chiffres de l'adresse + chiffre de contrôle.

À nouveau la somme de tous les chiffres doit être divisible par 10. Un exemple est illustré par la figure 16.

FIG. 16 – Code ZIP