Codes correcteurs d'erreurs

La révolution des turbo-codes

Introduction

La communication avec les sondes spatiales, à l'autre bout du système solaire, pose le problème de la fiabilité du message transmis. Une transmission sur une telle distance est obligatoirement parasitée (notamment à cause de diverses sources de perturbations électromagnétiques). Pourtant, dans ce domaine et dans bien d'autres, il est primordial que les informations collectées par les sondes soient bien reçues. Il y a donc nécessité de « sécuriser » la transmission : c'est le rôle des codes correcteurs d'erreurs. On rajoute au message à transmettre des informations supplémentaires, qui permettent de reconstituer le message au niveau du récepteur.

On trouvera en Figure 1 la schématisation d'un canal de transmission sécurisé par un code correcteur d'erreurs.

I. Codes en bloc linéaires

1. Présentation

Pour un traitement informatique, c'est-à-dire automatisé, de l'information, on numérise le signal à transmettre (une image, un son…). On ramène ainsi celui-ci à une séquence de bits e_1e_2…. A cause des inévitables parasites qui détériorent le message, on ne peut pas envoyer cette séquence telle quelle.

Pour améliorer la fiabilité de la transmission des données, une des méthodes de codage les plus simples est alors de répéter chaque bit :

La séquence e_1e_2… sera ainsi transmise sous la forme $e_1e_1e_2e_2$….

Lors de la réception du message, le décodeur peut ainsi comparer chaque couple de bits reçus. S'ils sont différents, alors il y a détection d'erreur.

On voit ainsi qu'en doublant la longueur du message (mais aussi le temps de transmission), on parvient à détecter d'éventuelles erreurs.

Toutefois, ce codage simple ne permet pas de les corriger. Pour cela, on peut tripler les bits. Si on considère (ce qui est plus que raisonnable) qu'il y a au maximum une erreur pour chaque séquence de 3 bits, alors il est possible de les corriger : le décodeur n'a qu'à choisir le symbole qui apparaît deux fois dans chaque triplet reçu.

Si le canal de transmission n'est pas trop parasité, il paraît inutile d'ajouter autant de redondance au message transmis. On peut ainsi utiliser le système du bit de parité (qui ne permet que la détection d'erreurs) : le message est découpé en blocs de k bits, auxquels on ajoute un bit tel qu'il y ait un nombre pair de 1 dans le bloc transmis.

2. Codes en Bloc

Le message à transmettre est découpé en blocs de k bits, qui sont alors traités séparément par le codeur.

On appelle code un ensemble C de 2^k n-uplets de bits, avec n > k, dont les éléments sont appelés mots. Les blocs à transmettre sont traduits en des éléments du code, chacun des 2^k messages différents possibles correspondant de manière unique à un des mots du code.

On appelle distance de Hamming entre deux mots du code le nombre de composantes par lesquelles ils diffèrent. La distance minimale d'un code C, notée d, est alors la distance de Hamming minimale entre deux mots quelconques.

Un code est alors défini par les trois paramètres [n,k,d], où n est la taille du code, k sa dimension et d sa distance minimale.

Par exemple on peut considérer le code suivant, de paramètres 5,2,3:

Message	Mot correspondant
(0,0)	(0,0,1,1,0)
(0,1)	(0,1,0,1,1)
(1,0)	(1,0,1,0,1)
(1,1)	(1,1,0,0,0)

On montre qu'un code est capable de corriger $e = E((d-1)/2)$ erreurs[1]. En effet, si on sait qu'un mot x a été transmis avec e erreurs ou moins, et si on reçoit le n-uplet x' alors x est le mot a du code tel que $d(x', a)$ soit minimal : le décodage s'effectue donc en calculant les distances entre x' et tous les mots du code.

Le rapport d/n renseigne donc sur la fiabilité du code.

Le rapport R = k/n est appelé taux du code. Plus il est proche de 0 et plus la redondance introduite, et donc le temps de transmission, sont importants.

On voit qu'il est impossible d'avoir en même temps une fiabilité et un taux élevés. Toute la difficulté de la construction des codes est de trouver le bon compromis.

II. Théorie de l'information – Théorème de Shannon

On définit un *canal de transmission* comme un système physique permettant la transmission d'une information entre deux points distants. Le taux d'erreurs binaire (TEB) d'un message est le rapport du nombre de bits erronés par le nombre de bits du message.

En 1948, Shannon énonce dans « A Mathematical Theory of Information » le théorème fondamental de la théorie de l'information :

Tout canal de transmission admet un paramètre C, appelé capacité du canal, tel que pour tout $\varepsilon > 0$ et pour tout $R < C$, il existe un code de taux R permettant la transmission du message avec un taux d'erreurs binaire de ε.

En d'autres termes, on peut obtenir des transmissions aussi fiables que l'on veut, en utilisant des codes de taux plus petits que la capacité du canal. Cependant, ce théorème n'indique pas le moyen de construire de tels codes.

On cherche donc à construire des codes ayant un taux le plus élevé possible (pour des raisons de temps et de coût) et permettant une fiabilité arbitrairement grande.

Les codes classiques présentés plus haut ne permettent pas d'atteindre cette limite. On a donc développé d'autres systèmes de codages.

III. Codes convolutifs (ou convolutionnels)

1. Généralités

Le principe des codes convolutifs, inventés par Peter Elias en 1954, est non plus de découper le message en blocs finis, mais de le considérer comme une séquence semi-infinie $a_0a_1a_2\ldots$ de symboles qui passe à travers une succession de registres à décalage, dont le nombre est appelé *mémoire du code*.

Pour simplifier, et également parce que c'est ainsi pour la quasi-totalité des codes convolutifs utilisés, nous considérerons le cas où le message est constitué de bits.

Dans l'exemple représenté en Figure 2, a_t parvient au codeur à l'instant t. Les bits de sortie seront $X = a_t + a_{t-1} + a_{t-2}$ et $Y = a_t + a_{t-2}$ (addition modulo 2).

[1] E désigne ici la fonction partie entière

Supposons que le codeur reçoive le message 1011, les registres étant initialement tous deux à 0.

On constate (cf Figure 3) que la séquence codée devient 11 10 00 01, et que les registres seront finalement à l'état 11.

On représente le code par un « diagramme des états de transition », donné en Figure 4.

Les cases représentent les différents états des registres, et les flèches les transitions possibles (le premier chiffre correspond au bit d'entrée, les deux autres aux bits de sortie X et Y).

2. Codes NSC et RSC

Deux catégories de codes convolutifs ont été particulièrement étudiées et présentent des propriétés intéressantes :

- **Les codes non systématiques ou NSC (Non Systematic Convolutional codes)**

Un code convolutif est dit *systématique* si l'un des bits de sortie est identique au bit d'entrée.

Les codes NSC, dont celui présenté ci-dessous est un exemple, présentent l'avantage par rapport aux codes systématiques de fournir plus d'information : tout bit de sortie du codeur renseigne sur plusieurs bits du message codé. Le décodeur dispose donc de plus d'éléments dans un code NSC, et permet donc de corriger plus d'erreurs.

Pour cette raison, ce sont les codes NSC qui ont été principalement étudiés et utilisés jusqu'au début des années 1990. On constate expérimentalement que la puissance d'un code (sa capacité à corriger les erreurs) augmente plus ou moins linéairement avec la mémoire ν de ce code. On s'est donc attaché à augmenter ν ce qui pose des problèmes car la complexité du décodeur est en $\nu\, 2^{\nu}$ ($\nu = 8$ est aujourd'hui un quasi-maximum pour la plupart des applications).

- **Les codes systématiques récursifs ou RSC (Recursive Systematic Convolutional codes)**

Un code convolutif est dit *récursif* si la séquence passant dans les registres à décalages est « alimentée » par le contenu de ces registres.

Un exemple de codeur récursif est représenté en Figure 5.

Supposons que le codeur reçoive le message 1011, les registres étant initialement tous deux à 0.

La séquence codée devient (cf Figure 6) : 11 01 10 10, et les registres seront finalement à l'état 11.

On a constaté expérimentalement grâce aux travaux sur les turbo-codes (et une équipe de chercheurs australiens s'attache à le démontrer) que seuls les codes RSC sont susceptibles d'atteindre la limite de Shannon.

3. Algorithme de décodage

L'algorithme présenté ici est une version simplifiée du célèbre algorithme de Viterbi (inventé en 1970) appliquée à l'exemple du codeur de la Figure 2. Il est présenté en Annexe 2 en langage C.

Bien que le message constitue a priori une séquence semi-infinie, celui-ci sera découpé en blocs de grande taille (variant en pratique entre 100 et 100 000 bits, voire plus). Les registres à décalage sont remis à 0 entre chaque bloc.

Le principe est d'examiner tous les chemins possibles du message à travers le diagramme des états de transition (Figure 4), en supprimant au fur et à mesure les moins probables. Supposons par exemple (toujours dans le cas du codeur de la Figure 2) que le premier couple de bit du bloc reçu

soit 00. Si ε désigne la probabilité pour qu'un bit soit erroné, un rapide calcul utilisant la probabilité conditionnelle (on sait en effet que le couple sorti du codeur ne pouvait être que 00 ou 11 puisque les registres sont à 0) montre que la probabilité pour que le couple soit bien 00 est de $\frac{(1-\varepsilon)^2}{\varepsilon^2+(1-\varepsilon)^2}$ et celle que le couple soit 11 de $\frac{\varepsilon^2}{\varepsilon^2+(1-\varepsilon)^2}$. ε étant supposé « petit » devant 1, on trouve en effectuant un développement limité à l'ordre 2 : $1-\varepsilon^2$ et ε^2. On définit donc le chemin affecté de ces probabilités, et on continue avec le couple de bits suivant en tenant compte de l'état du registre actuel donné par le chemin parcouru.

Il serait bien sûr impossible de parcourir tous les chemins pour des raisons de mémoire et de temps. (cela en ferait 2^k où k est la taille du bloc !). En fait, il suffit d'en conserver 2^v (4 dans l'exemple) à chaque étape, un pour chaque état des registres. On conserve le chemin le plus probable pour arriver à cet état des registres, en effectuant un choix arbitraire dans le cas d'égalité des probabilités. Il ne reste plus qu'à choisir, à la fin du bloc, le chemin le plus probable.

On trouvera en Annexe 2 les résultats d'une simulation montrant l'efficacité du codage. On constate que celui-ci est d'autant plus puissant que la taille des blocs est grande. Toutefois, leur performance ne semblent pas extraordinaires par rapport à celles des codes classiques, de Hamming ou par triple répétition.

IV. Entrelacement

L'entrelacement consiste à permuter une séquence de bits de manière à ce que deux symboles proches à l'origine soient le plus éloignés possibles l'un de l'autre. Cela permet en particulier de transformer une erreur portant sur des bits regroupés en une erreur répartie sur l'ensemble de la séquence. Un exemple classique de l'utilité de l'entrelacement est celui des codes correcteurs d'erreurs protégeant les disques compacts : il s'agit d'un code en bloc de Reed-Salomon entrelacé qui permet de corriger plus de 4 000 erreurs consécutives dues à une poussière, une éraflure…

On cherche également, en particulier pour les turbo-codes, à réaliser une permutation aussi « chaotique » que possible. Aucune règle n'existe encore ici. Le tout est de trouver l'entrelaceur qui donnera les meilleurs résultats expérimentaux !

V. Turbo-codes

Les turbo-codes ont été inventés en 1991, et présentés à la communauté scientifique en 1993, par une équipe de l'Ecole Nationale Supérieure des Télécommunications de Brest dirigée par Claude Berrou et Alain Glavieux. Les spécialistes des codes correcteurs d'erreurs ont tout d'abord accueilli cette invention avec beaucoup de scepticisme, du fait des extraordinaires performances annoncées. Cependant, d'autres équipes, dans le monde entier, sont parvenues peu après aux mêmes résultats, ce qui a contribué au développement des turbo-codes. Ils ont été adoptés par toutes les agences spatiales mondiales, et seront utilisés dans la transmission des données du nouveau standard de téléphonie mobile qui va succéder au GSM. Toutefois, tous les résultats concernant ces codes n'ont été établis pour le moment que de manière expérimentale. C'est pourquoi l'on se contentera de constater leur efficacité, sans pouvoir la démontrer.

1. Codeur

Le principe des turbo-codes est l'utilisation conjointe de deux codeurs convolutifs récursifs, non pas en série, comme cela était déjà fait depuis de nombreuses années, mais en parallèle (cf Figure 7).

L'entrelaceur permet ainsi de coder avec le même codeur deux séquences d'autant plus différentes que l'entrelacement sera chaotique.

On constate sur le schéma représenté Figure 7 que le taux de codage R des turbo-codes est de $\frac{1}{3}$: trois bits de sortie pour un bit d'entrée. On peut le ramener à $\frac{1}{2}$ par un *poinçonnage* qui consiste à

ne garder à tout instant que l'un des bits Y_1 ou Y_2. Les turbo-codes peuvent ainsi être comparés aux codes convolutifs classiques, mais cette opération complexifie encore le décodage ; nous n'en tiendrons donc pas compte dans le paragraphe suivant.

2. Décodeur

Si le codage est relativement simple, le décodage est beaucoup plus complexe. Nous n'en présentons ici que le principe. La Figure 8 donne un schéma simplifié du turbo-décodeur, pour information.

Les entrelaceurs et le désentrelaceur utilisent la même permutation que l'entrelaceur du codeur. Ils permettent par exemple la comparaison de la séquence X avec la séquence Y_2, entrelacée au moment du codage.

Cette procédure de décodage est itérée un certain nombre de fois fixé à l'avance. La décision n'intervient qu'après. A chaque étape, le décodeur dispose des probabilités calculées à l'étape précédente, et se base sur ces probabilités, et non sur une « pré-decision ». C'est ce qui a valu aux turbo-codes leur nom : comme dans un moteur turbo, les « gaz d'échappement » sont réutilisés pour augmenter les performances.

3. Performances

Le graphique de la Figure 9 représente le Taux d'Erreurs Binaire d'un message codé avec un turbo-code de taux $R = \frac{1}{2}$ et passant par un canal dont le *SNR* (*Signal to Noise Ratio*) est E_b/N_0. Ce rapport caractérise la fiabilité du canal. Plus celui-ci est élevé, meilleure est la transmission. La limite de Shannon, représentée à 0.2 dB, correspond à un canal de capacité $\frac{1}{2}$. Ainsi, d'après le théorème de Shannon, pour tout *SNR* supérieur à 0.2 dB, il existe un code de taux $\frac{1}{2}$, telle que le TEB soit aussi petit que l'on veut.

A 0,35 dB de la limite de Shannon, on parvient à ramener le TEB à moins de 10^{-5}. A titre de comparaison, un code convolutif classique ($\nu = 8$) fournit dans les mêmes conditions un TEB voisin de 4.10^{-2} ! En fait, si k désigne la taille des blocs transmis, on obtient, avec les turbo-codes, des performances comparables à un code convolutif avec $\nu \approx \frac{k}{8}$, soit, en pratique de l'ordre de 100, ce qui serait bien entendu, irréalisable.

On constate que la qualité du message reçu augmente avec le nombre d'itération. De même, on peut constater expérimentalement que le choix de l'entrelaceur joue un rôle fondamental.

Conclusion

L'invention des turbo-codes est ainsi une véritable révolution dans le domaine des codes correcteurs d'erreurs. Leurs performances, qui se rapprochent des limites théoriques établies par Shannon, les présentent comme incontournables dans la plupart des domaines où la transmission de données intervient. Toutefois, ils ont encore un inconvénient : le délai, inhérent à la technique itérative de décodage, qui les rend impropres à la transmission de la voix en téléphonie.

Bibliographie

1. Publications

- BERROU C., PYNDIAH R., JEZEQUEL M., GLAVIEUX A., ADDE P., « La double correction des Turbo-Codes », *La Recherche*, n°315, pp. 34-37, décembre 1998.
- BERROU C., « Les turbo codes convolutifs », ENST Bretagne, mars 1999.
- COHEN G., DORNSTETTER J-L. et GODLEWSKI P., *Codes correcteurs d'erreurs : une introduction au codage algébrique*, Masson.
- DEMAZURE M., *Cours d'algèbre : primalité, divisibilité, codes*, Cassini.
- INGVARSON O. et SVENELL H., « Error performance of turbo codes » (Master's Thesis, 18/12/98), http://www.df.lth.se/~pi/exjobbet/
- LACHAUD G., VLADUT S., «Les codes correcteurs d'erreurs», *La Recherche*, col. 26, n°278, pp. 778-782, juillet-août 1995.
- SHANNON C. E., « A Mathematical Theory of Communication », *The Bell System Technical Journal*, vol. 27, pp. 379-423, 623-656, juillet-octobre 1948.
- URO M., *Théorie de l'information* (cours de DEA)

2. Conférence

- BERROU C., « Les turbo codes convolutifs », Université de Marne-la-vallée, 14/04/99.

Annexe 1 : Illustrations

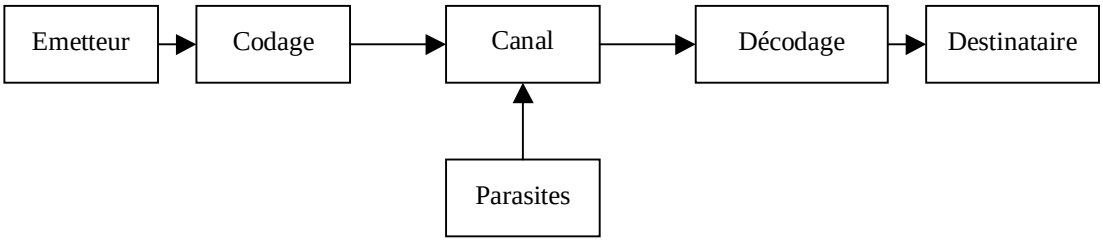


Figure 1 : Transmission d'un message à travers un canal parasité

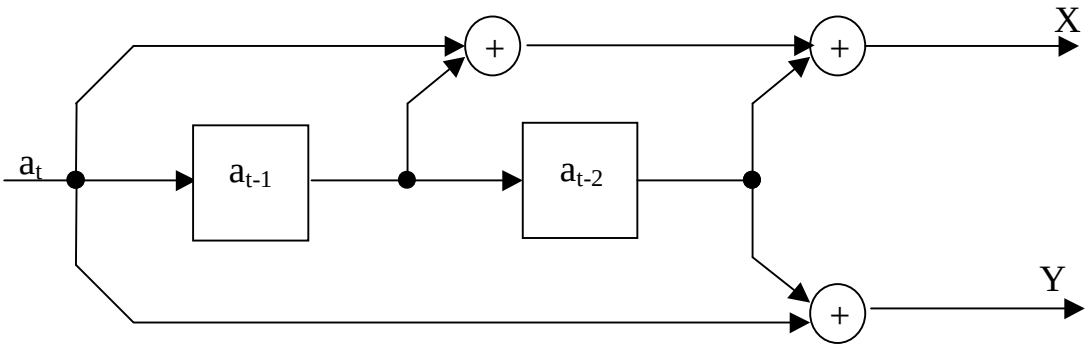


Figure 2 : Exemple de codeur convolutif

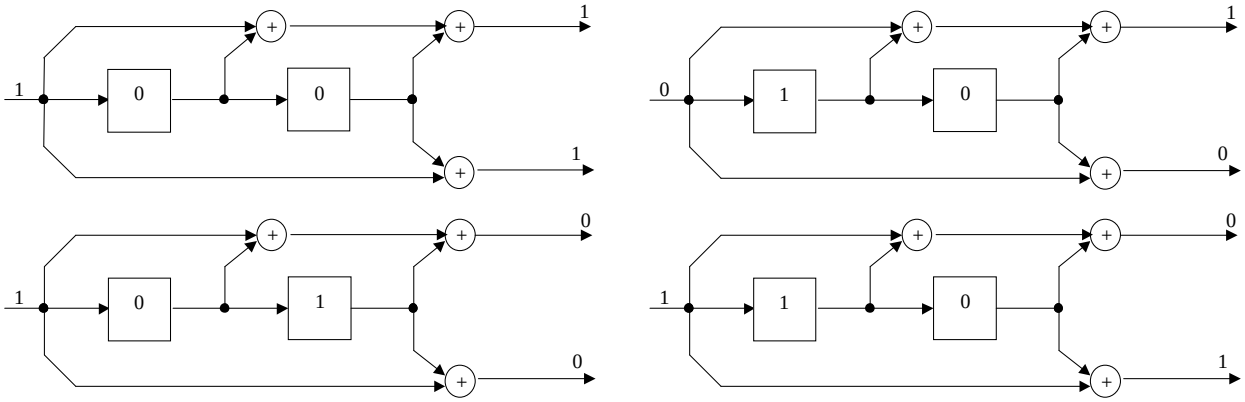


Figure 3 : Exemple de codage d'une séquence par le codeur de la figure 1

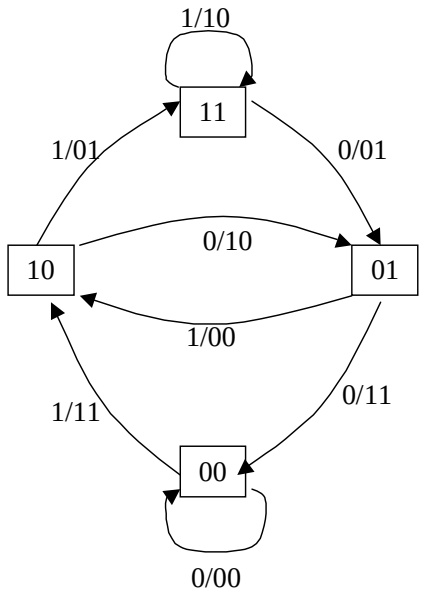


Figure 4 : Diagramme des états de transition du codeur de la Figure 2

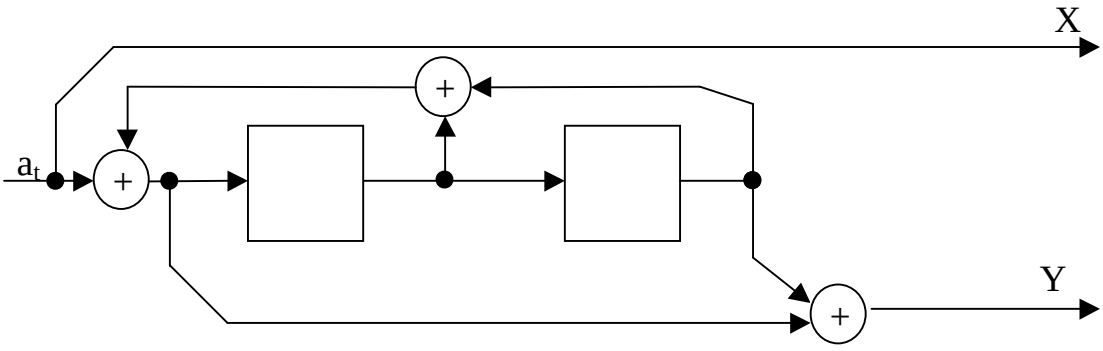


Figure 5 : Exemple de codeur RSC

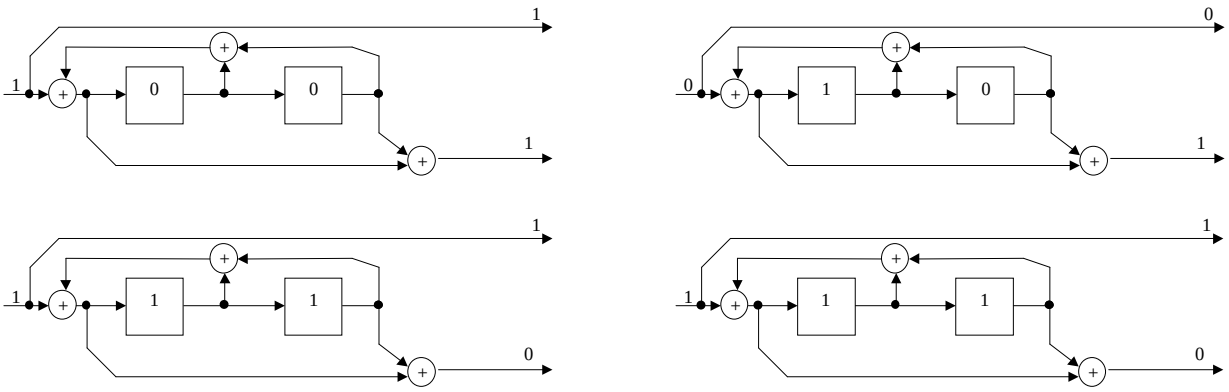


Figure 6 : Exemple de codage d'une séquence par le codeur de la Figure 5

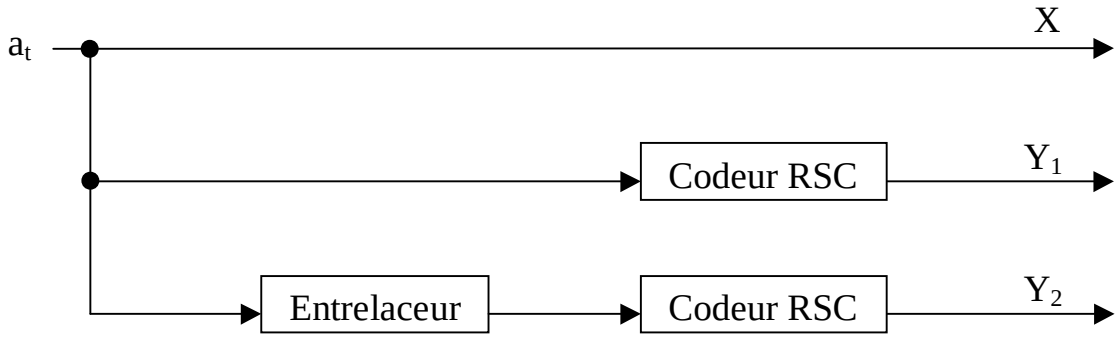


Figure 7 : Schéma de principe d'un turbo-codeur

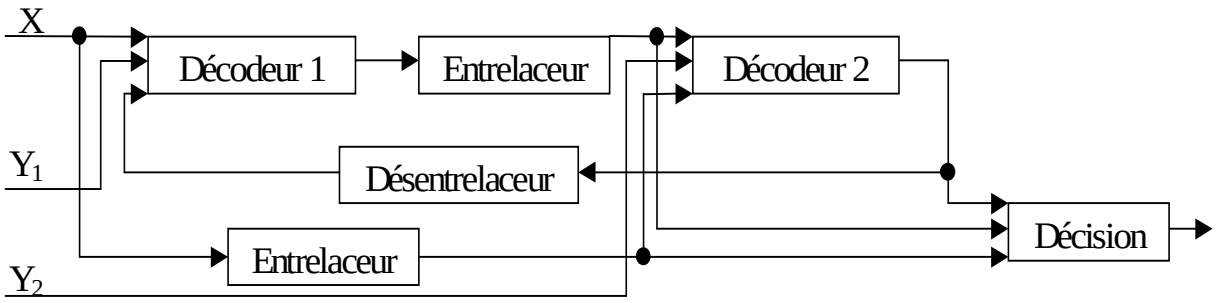


Figure 8 : Schéma de principe d'un turbo-décodeur

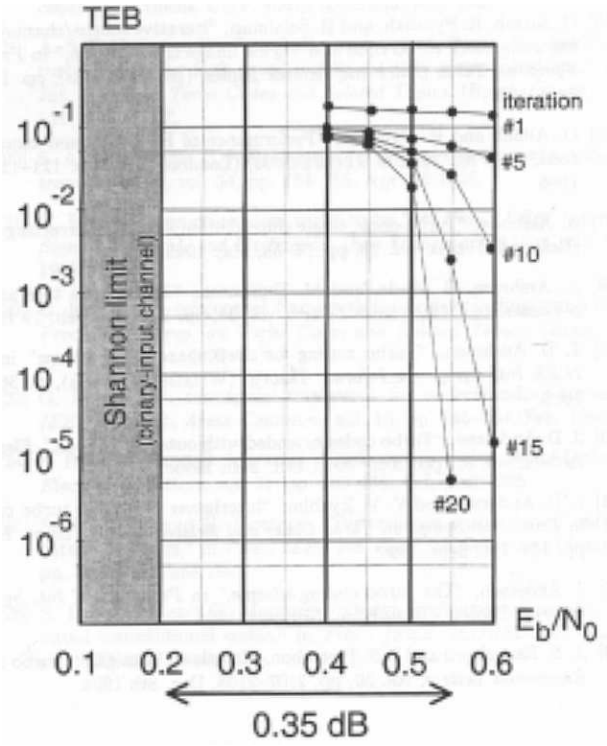


Figure 9 : Performances des turbo-codes (source : ENST Bretagne)

Annexe 2 : programmes et résultats

```
/* Header CDCRERR.H contenant diverses fonctions utilisées par les
programmes PARASITE.C, CODE.C, DECODE.C, CODCONV.C et DECCONV.C */

#include <stdio.h>

#ifndef __dj_include_cdcrerr_h_
#define __dj_include_cdcrerr_h_

typedef struct OCTET {
        char bit[8];
} octet;

int power(int x,int k) /* calcule x^k */
{
        int i, n=1;
        for (i=1;i<=k;i++)
                n=n*x;
        return n;
}

int lof(FILE *fichier) /* détermine la taille d'un fichier (en octets) */
{
        int pos, tmp;
        pos=ftell(fichier);
        if (fseek(fichier,0,SEEK_END)==0)
                tmp=ftell(fichier);
        else
                tmp=-1;
        if (fseek(fichier,pos,SEEK_SET)==0)
                return tmp;
        else
                return -1;
}

char octet2char (octet x) /* convertit un octet (structure définie plus */
{                         /* haut) en un caractère                       */
        unsigned char n = 0;
        int i;
        for (i=0;i<=7;i++)
                n = n + x.bit[i] * power(2,i);
        return (char) n;
}

octet char2octet (char x) /* convertit un caractère en un octet */
{
        unsigned char n=(unsigned char) x;
        int i,tmp;
        octet a;
        for (i = 7;i>=0;i--)
                if (n>=(tmp=power(2,i)))
                {
                        a.bit[i] = 1;
                        n = n - tmp;
                }
                else
                        a.bit[i]=0;
        return a;
}

#endif /* !__dj_include_cdcrerr_h_ */
```

```
/* PARASITE.C : parasitage aléatoire d'un fichier. */

#include "cdcrerr.h"
#include <stdlib.h> /* pour rand() et srand() */

int main(void)
{
        char nomfich1[13], nomfich2[13], reponse[10];
        FILE *fptr1, *fptr2;
        int l, i, j, deb;
        float prob;
        octet oc;
        srand(time()); /* initialisation du générateur de nombres aléatoires */
        printf("Fichier à parasiter : ");
        scanf("%s",&nomfich1);
        printf("Fichier de sortie : ");
        scanf("%s",&nomfich2);
        printf("Probabilité d'interférences : ");
        scanf("%f",&prob);
        printf("Nombre d'octets à épargner en début de fichier : ");
        scanf("%d",&deb); /* Permet de préserver l'entête d'un fichier Bitmap, */
                          /* afin qu'il puisse être lu par un logiciel de dessin */
        if (((fptr1=fopen(nomfich1,"rb"))!=NULL) &&
            ((fptr2=fopen(nomfich2,"wb"))!=NULL))
        {
                l=lof(fptr1);
                for (i=1;i<=l;i++)
                {
                        oc=char2octet(fgetc(fptr1));
                        for (j=0;j<=7;j++)
                                if ((((float) rand())/RAND_MAX<=prob) &&
                                    (i>deb))
                                        oc.bit[j]=(!oc.bit[j]);
                        fputc(octet2char(oc),fptr2);
                }
                fclose(fptr1);
                fclose(fptr2);
                return 0;
        }
        else
        {
                fclose(fptr1);
                fclose(fptr2);
                printf("Echec.");
                return 1;
        }
}
```

```
/* CODE.C : codage par triple répétition et code de Hamming */

#include "cdcrerr.h"

void cod_rep3(char x,FILE *fichier);     /* codage par triple répétition */
void cod_hamming(char x,FILE *fichier);  /* code de Hamming */

int main(void)
{
        char nomfich1[13], nomfich2[13];
        FILE *fptr1, *fptr2;
        int l, i, choix_code;
        char x;
        printf("Codes disponibles\n");
        printf("1) Triple répétition\n");
        printf("2) Hamming\n");
        printf("Votre choix : ");
        scanf("%d",&choix_code);
        printf("Fichier à coder : ");
        scanf("%s",&nomfich1);
        printf("Fichier de sortie : ");
        scanf("%s",&nomfich2);
        if (((fptr1=fopen(nomfich1,"rb"))!=NULL) &&
            ((fptr2=fopen(nomfich2,"wb"))!=NULL))
        {
                l=lof(fptr1);
                for (i=1;i<=l;i++)
                {
                        x=fgetc(fptr1);
                        switch (choix_code)
                        {
                                case 1: /* triple répétition */
                                {
                                        cod_rep3(x,fptr2);
                                        break;
                                }
                                case 2: /* Hamming */
                                {
                                        cod_hamming(x,fptr2);
                                        break;
                                }
                                default: ;
                        }
                }
                fclose(fptr1);
                fclose(fptr2);
                return 0;
        }
        else
        {
                fclose(fptr1);
                fclose(fptr2);
                printf("Echec.");
                return 1;
        }
}

void cod_rep3(char x,FILE *fichier)
{
        int j;
        for (j=1;j<=3;j++)
                fputc(x,fichier);
}

void cod_hamming(char x,FILE *fichier)
```

```
{
        int j;
        octet oldoc, oc[2];
        oldoc=char2octet(x);
        for (j=0;j<=1;j++)
        {
                oc[j].bit[0]=oldoc.bit[4*j];
                oc[j].bit[1]=(oldoc.bit[4*j]+oldoc.bit[4*j+1]) % 2;
                oc[j].bit[2]=(oldoc.bit[4*j+1]+oldoc.bit[4*j+2]) % 2;
                oc[j].bit[3]=(oldoc.bit[4*j]+oldoc.bit[4*j+2]+oldoc.bit[4*j+3]) % 2;
                oc[j].bit[4]=(oldoc.bit[4*j+1]+oldoc.bit[4*j+3]) % 2;
                oc[j].bit[5]=oldoc.bit[4*j+2];
                oc[j].bit[6]=oldoc.bit[4*j+3];
                fputc(octet2char(oc[j]),fichier);
        }
}
```

```
/* DECODE.C : décodage des codes par triple répétition et de Hamming */

#include "cdcrerr.h"

void dec_rep3(FILE *fichier1, FILE *fichier2); /* decodage triple répétition */
void dec_hamming(FILE *fichier1, FILE *fichier2);  /* décodage Hamming */
char hamming_s(octet o, char i); /* utilisé par dec_hamming */

int main(void)
{
        char nomfich1[13], nomfich2[13];
        FILE *fptr1, *fptr2;
        int choix_code;
        printf("Codes disponibles\n");
        printf("1) Triple répétition\n");
        printf("2) Hamming\n");
        printf("Votre choix : ");
        scanf("%d",&choix_code);
        printf("Fichier à décoder : ");
        scanf("%s",&nomfich1);
        printf("Fichier de sortie : ");
        scanf("%s",&nomfich2);
        if (((fptr1=fopen(nomfich1,"rb"))!=NULL) &&
            ((fptr2=fopen(nomfich2,"wb"))!=NULL))
        {
                switch (choix_code)
                {
                        case 1: /* triple répétition */
                        {
                                dec_rep3(fptr1,fptr2);
                                break;
                        }
                        case 2: /* Hamming */
                        {
                                dec_hamming(fptr1,fptr2);
                                break;
                        }
                        default: ;
                }
                fclose(fptr1);
                fclose(fptr2);
                return 0;
        }
        else
        {
                fclose(fptr1);
                fclose(fptr2);
                printf("Echec.");
                return 1;
        }
}

void dec_rep3(FILE *fichier1, FILE *fichier2)
{
        int i, j, l;
        octet oc[3], ocr;
        char x[3], r;
        l=lof(fichier1);
        for (i=1;i<=l/3;i++)
        {
                for (j=0;j<=2;j++)
                        x[j]=fgetc(fichier1);
                if ((x[0]==x[1])||(x[1]==x[2]))
                        r=x[1];
                else if (x[0]==x[2])
```

```
                        r=x[0];
                else
                {
                        for (j=0;j<=2;j++)
                                oc[j]=char2octet(x[j]);
                        for (j=0;j<=7;j++)
                                if ((oc[0].bit[j]==oc[1].bit[j]) ||
                                    (oc[1].bit[j]==oc[2].bit[j]))
                                        ocr.bit[j]=oc[1].bit[j];
                                else
                                        ocr.bit[j]=oc[0].bit[j];
                        r=octet2char(ocr);
                }
                fputc(r,fichier2);
        }
}

void dec_hamming(FILE *fichier1, FILE *fichier2)
{
        int i, j, k, l;
        octet oc[2], ocr;
        l=lof(fichier1);
        for (i=1;i<=l/2;i++)
        {
                for (j=0;j<=1;j++)
                {
                        oc[j]=char2octet(fgetc(fichier1));
                        k=-1;
                        switch (hamming_s(oc[j],0)*4
                              + hamming_s(oc[j],1)*2
                              + hamming_s(oc[j],2))
                        {
                                case 1: {k=2; break;}
                                case 2: {k=1; break;}
                                case 3: {k=4; break;}
                                case 4: {k=0; break;}
                                case 5: {k=6; break;}
                                case 6: {k=3; break;}
                                case 7: {k=5; break;}
                         }
                         if (k!=-1)
                                oc[j].bit[k]=!(oc[j].bit[k]);
                         ocr.bit[4*j]=oc[j].bit[0];
                         ocr.bit[4*j+1]=(oc[j].bit[1]+oc[j].bit[0]) % 2;
                         ocr.bit[4*j+2]=oc[j].bit[5];
                         ocr.bit[4*j+3]=oc[j].bit[6];
                }
                fputc(octet2char(ocr),fichier2);
        }
}

char hamming_s(octet o, char i)
{
        switch (i)
        {
                case 1:
                        return ((o.bit[1]+o.bit[3]+o.bit[4]+o.bit[5]) % 2);
                case 2:
                        return ((o.bit[2]+o.bit[4]+o.bit[5]+o.bit[6]) % 2);
                default:
                        return ((o.bit[0]+o.bit[3]+o.bit[5]+o.bit[6]) % 2);
        }
}
```

```
/* CODCONV.C : codage convolutif */

#include "cdcrerr.h"
#define MAX_BLOC 16384

void cod_conv(FILE *fich1,FILE *fich2);

int taille_bloc,l;

int main(void)
{
        char nomfich1[13], nomfich2[13];
        FILE *fptr1, *fptr2;
        int i, choix_code;
        printf("Fichier à coder : ");
        scanf("%s",&nomfich1);
        printf("Fichier de sortie : ");
        scanf("%s",&nomfich2);
        printf("Taille des blocs (de 16 à %d) : ",MAX_BLOC);
        scanf("%d",&taille_bloc);
        if (((fptr1=fopen(nomfich1,"rb"))!=NULL) &&
            ((fptr2=fopen(nomfich2,"wb"))!=NULL))
        {
                l=lof(fptr1);
                for (i=1;i<=(8*l-1)/(taille_bloc/2) + 1;i++)
                {
                        cod_conv(fptr1,fptr2);
                }
                fclose(fptr1);
                fclose(fptr2);
                return 0;
        }
        else
        {
                fclose(fptr1);
                fclose(fptr2);
                printf("Echec.");
                return 1;
        }
}

void cod_conv(FILE *fich1,FILE *fich2)
{
        int i,j,k;
        char registres=0, t;
        octet in,out;
        for (i=0;i<=taille_bloc/16-1;i++)
        {
                if (ftell(fich1)>=l) return;
                in=char2octet(fgetc(fich1));
                for (j=0;j<=1;j++)
                {
                        for (k=0;k<=3;k++)
                        {
                                switch (registres | in.bit[7-(4*j+k)] * 4)
                                {
                                        case 0:
                                        {
                                                t=0; break;
                                        }
                                        case 1:
                                        {
                                                registres=0; t=3; break;
                                        }
                                        case 2:
```

```
                                        {
                                                registres=1; t=2; break;
                                        }
                                        case 3:
                                        {
                                                registres=1; t=1; break;
                                        }
                                        case 4:
                                        {
                                                registres=2; t=3; break;
                                        }
                                        case 5:
                                        {
                                                registres=2; t=0; break;
                                        }
                                        case 6:
                                        {
                                                registres=3; t=1; break;
                                        }
                                        case 7:
                                                t=2;
                                }
                                out.bit[6-2*k]  = t & 1;
                                out.bit[7-(2*k)]= (t & 2)/2;
                        }
                        fputc(octet2char(out),fich2);
                }
        }
}
```

```
/* DECCONV.C : décodage convolutif */

#include <math.h>
#include <stdlib.h>
#include "cdcrerr.h"
#define MAX_BLOC 16384

char zone[MAX_BLOC/2];

struct liste {
  char courant;
  char ref;
  struct liste *precedent;
};

typedef struct liste *ptrliste;

typedef struct CHEMIN
{
  ptrliste contenu;
  float proba;
  char registres;
} chemin;

void dec_conv(FILE *fich1,FILE *fich2);
chemin *cherche_chemins (chemin *,int compteur, char x, int indice, FILE *fichier);
void copy_chemin(chemin *a, const chemin b);
void init_automate(float tab[16][2]);
void ajoute_chemin(chemin *a,char j);
void delete_chemin(chemin *c);

int taille_bloc, l;

float epsilon;

float probatransition[16][2];

int main(void)
{
        char nomfich1[13], nomfich2[13];
        FILE *fptr1, *fptr2;
        int i;
        printf("Fichier à décoder : ");
        scanf("%s",nomfich1);
        printf("Fichier de sortie : ");
        scanf("%s",nomfich2);
        printf("Taille des blocs (jusqu'à %d) : ",MAX_BLOC);
        scanf("%d",&taille_bloc);
        printf("Probabilité d'erreur : ");
        scanf("%f",&epsilon);
      init_automate(probatransition);

        if (((fptr1=fopen(nomfich1,"rb"))!=NULL) &&
            ((fptr2=fopen(nomfich2,"wb"))!=NULL))
        {
                l=lof(fptr1);
                for (i=1;i<=(8*l-1)/(taille_bloc) + 1;i++)
                {
                        dec_conv(fptr1,fptr2);
                }
                fclose(fptr1);
                fclose(fptr2);
                return 0;
        }
        else
```

```
        {
                fclose(fptr1);
                fclose(fptr2);
                printf("Echec.");
                return 1;
        }
}

void dec_conv(FILE *fich1,FILE *fich2)
{
        int i,j,k;
        float prob;
        chemin chemins[4];
        chemin *chptr;
        octet oc;
      ptrliste last;
        for (i=0;i<=3;i++)
        {
        chemins[i].contenu=NULL;
        chemins[i].proba=0;
        chemins[i].registres=0;
        }
        chptr=cherche_chemins(chemins,0,0,0,fich1);
        prob=1;
        for (i=0;i<=3;i++)
        {
                if (chptr[i].proba>prob || prob==1)
                {
                        k=i;
                        prob=chptr[i].proba;
                }
        }
      last=chptr[k].contenu;
      i=taille_bloc/2-1;
      while (last!=NULL)
        {
          zone[i--]=last->courant;
          last=last->precedent;
        }

        for (i=0;i<=taille_bloc/16-1;i++)
        {
                for (j=0;j<=7;j++)
                        oc.bit[7-j]=zone[i*8+j];
                fputc(octet2char(oc),fich2);
        }

      for(k=0;k<=3;k++)
        {
          delete_chemin(&chptr[k]);
          delete_chemin(&chemins[k]);
        }
}

void copy_chemin(chemin *a, const chemin b)
{
  a->contenu=b.contenu;
  if (a->contenu!=NULL) a->contenu->ref++;
  a->proba=b.proba;
  a->registres=b.registres;
}

void delete_chemin(chemin *c)
{
  ptrliste a=c->contenu;
```

```
  while (a!=NULL && a->ref==1)
    {
      ptrliste adetruire=a;
      a=a->precedent;
      free(adetruire);
    }
  if (a!=NULL) a->ref--;
  c->contenu=NULL;
}

void ajoute_chemin(chemin* a,char j)
{
  ptrliste new=(ptrliste)malloc(sizeof(struct liste));
  new->ref=1;
  new->courant=j;
  new->precedent=a->contenu;
  a->contenu=new;
}

void init_automate(float tab[16][2])
{
  int i;
  tab[0][0]=1-epsilon*epsilon;
  tab[0][1]=epsilon*epsilon;

  tab[1][0]=epsilon*epsilon;
  tab[1][1]=1-epsilon*epsilon;

  tab[2][0]=0.5;
  tab[2][1]=0.5;

  tab[3][0]=0.5;
  tab[3][1]=0.5;

  tab[4][0]=0.5;
  tab[4][1]=0.5;

  tab[5][0]=0.5;
  tab[5][1]=0.5;

  tab[6][0]=epsilon*epsilon;
  tab[6][1]=1-epsilon*epsilon;

  tab[7][0]=1-epsilon*epsilon;
  tab[7][1]=epsilon*epsilon;

  tab[8][0]=0.5;
  tab[8][1]=0.5;

  tab[9][0]=0.5;
  tab[9][1]=0.5;

  tab[10][0]=1-epsilon*epsilon;
  tab[10][1]=epsilon*epsilon;

  tab[11][0]=epsilon*epsilon;
  tab[11][1]=1-epsilon*epsilon;

  tab[12][0]=epsilon*epsilon;
  tab[12][1]=1-epsilon*epsilon;

  tab[13][0]=1-epsilon*epsilon;
  tab[13][1]=epsilon*epsilon;

  tab[14][0]=0.5;
```

```
  tab[14][1]=0.5;

  tab[15][0]=0.5;
  tab[15][1]=0.5;
  for(i=0;i<=15;i++)
    {
      tab[i][0]=log(tab[i][0]);
      tab[i][1]=log(tab[i][1]);
    }
}

chemin *cherche_chemins(chemin *Liste,int compteur, char x, int indice, FILE
*fichier)
{
        int i, j, k[4];
        char a;
        float prob[4];
        chemin *possibilites;
        if (compteur>=taille_bloc/2)
                return Liste;
        possibilites=malloc(8*sizeof(chemin));
        if (indice==0)
                x=fgetc(fichier);
        a=(x & (128+64))>>6;
        for (i=0;i<=3;i++)
        {
                for (j=0;j<=1;j++)
                {
              float p;
              p=probatransition[Liste[i].registres | (a * 4)][j];
              copy_chemin(&possibilites[4*j+i],Liste[i]);

              ajoute_chemin(&possibilites[4*j+i],j);
              possibilites[4*j+i].proba += p;
              possibilites[4*j+i].registres=(Liste[i].registres & 2)/2
                                                                  + 2 * j;
                }
        }
        for (i=0;i<=3;i++)
        {
                prob[i]=1;
                k[i]=-1;
      }
      for (j=0;j<=7;j++)
        if (possibilites[j].proba>prob[possibilites[j].registres] ||
prob[possibilites[j].registres]==1)
          {
            k[possibilites[j].registres]=j;
            prob[possibilites[j].registres]=possibilites[j].proba;
          }

      for(i=0;i<=3;i++) delete_chemin(&Liste[i]);
        for (i=0;i<=3;i++)
        {
                if (k[i] != -1)
              copy_chemin(&Liste[i],possibilites[k[i]]);
                else
              {
                copy_chemin(&Liste[i],Liste[0]);
              }
        }

      for(i=0;i<8;i++) delete_chemin(&possibilites[i]);
        free(possibilites);
        return (cherche_chemins(Liste,compteur+1,x<<2,(indice+3) % 4,fichier));
```

```
}
```

Simulation de l'efficacité d'un code correcteur d'erreur en cas de parasitage aléatoire d'un fichier

T.E.B. (Taux d'Erreur Binaire) = 5.10^{-2}

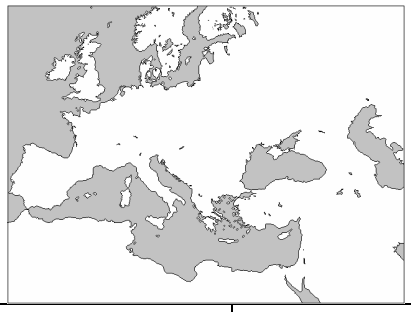
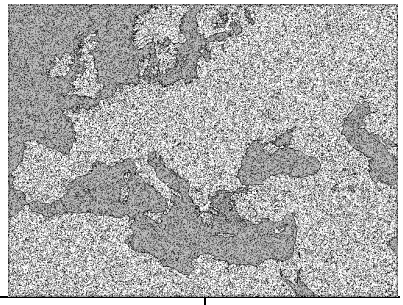
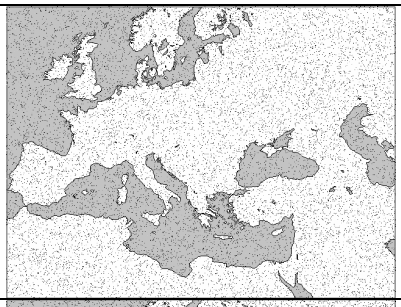
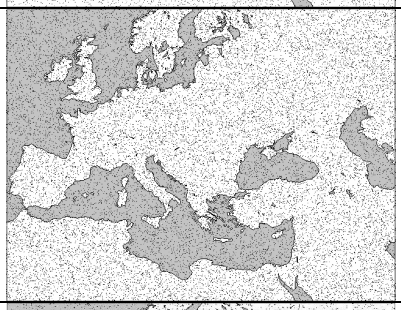
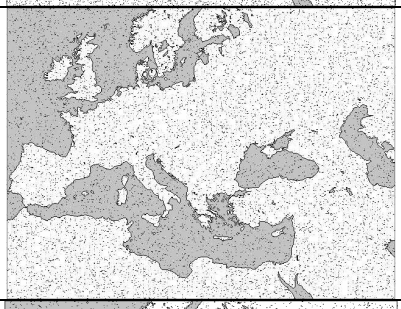
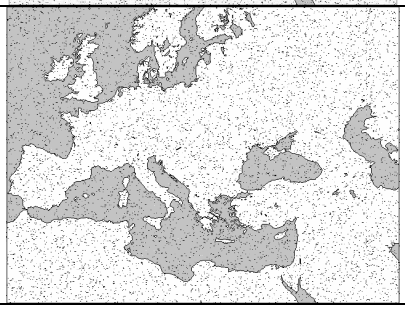


Image initiale

Image parasitée



Code utilisé	Longueur des blocs	Image décodée	Taux du code	Taux d'erreur binaire
Code par triple répétition	1 bit d'information et 2 bits de codage		$\frac{1}{3}$	$7,25.10^{-3}$
Code de Hamming	4 bits d'information et 3 bits de codage		$\frac{4}{7}$	$2,09.10^{-2}$
Code convolutif	Blocs de 128 bits		$\frac{1}{2}$	$1,54.10^{-2}$
Code convolutif	Blocs de 2048 bits		$\frac{1}{2}$	$1,24.10^{-2}$