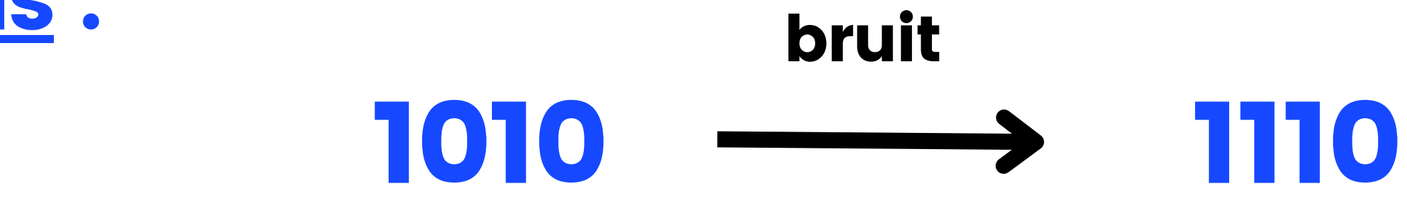
correction d'erreurs en communication, codes de Reed Muller

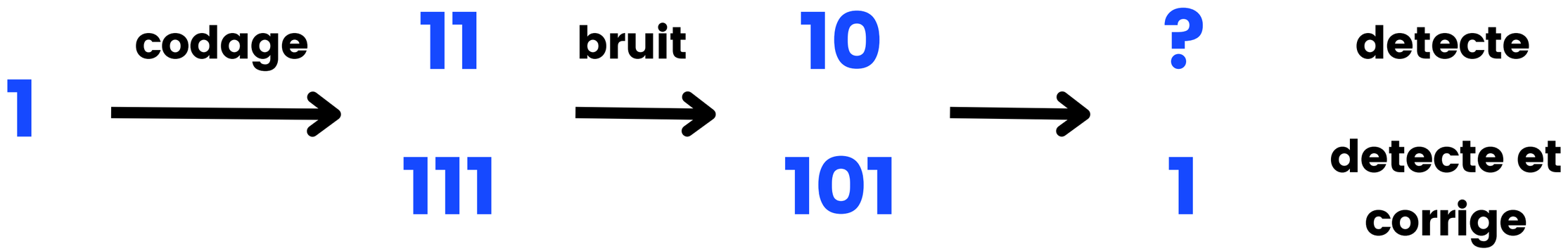
transition, transformation, conversion

dans quelle mesure l'utilisation de codes correcteurs est pertinente pour la transmission de données ?

- **perturbations** :



- **redondance** :



- **bit de controle** :

1+1 = 2 pair

1010 0 —bruit→ 1110 0

1+1+1= 3 impair → erreur

- **code de hamming**

structure en carré, placement avisé de plusieurs bits de contrôle

travail sur lignes et colones → localisation de l'erreur

corrige 1 erreur détecte 2 erreurs

distance minimale : d

plus petite distance de hamming entre deux mots distincts du code (nombre de bits qui diffèrent)

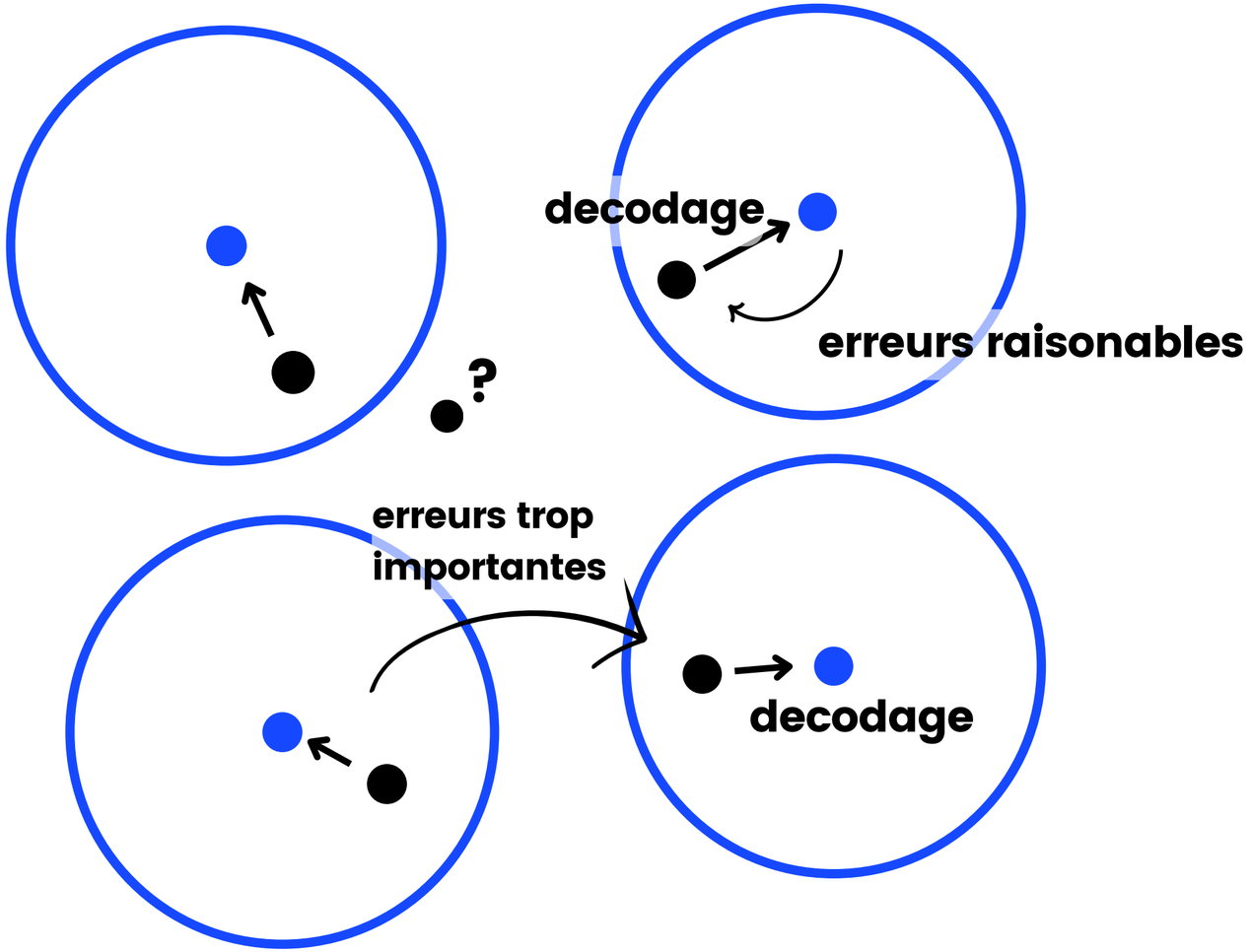
nombre d'erreurs corrigeables : $nb\_e = \lfloor (d-1)/2 \rfloor$

nombre d'erreurs qu'il est possible de corriger

taux d'information :

nombre de bits d'information / nombre de bits total

prendre le message recu et le rapporter au code le plus proche



on considèrera le canal de communication assez fiable pour décoder le message

- **paramètres :**

r → degré , m → nombre de variables

2^m - uplets x1, ..., xm

xi : alternance de 0 et 1 tous les $2^{(m-i)}$

pour m = 3

x0 = 1 1 1 1 1 1 1 1

x1 = 0 0 0 0 1 1 1 1

x2 = 0 0 1 1 0 0 1 1

x3 = 0 1 0 1 0 1 0 1

- **RM(1, m) :**

toutes les combinaisons linéaires de 1, x1, ..., xm de coefficents a0, ..., am

Les codes de Reed Muller : definition RM(1,m) et propriétés

- **base :**

(1 , x1 , … , xm)

- **matrice generatrice :**

$$\begin{bmatrix} 1 \\ x1 \\ \dots \\ xm \end{bmatrix}$$

- **distance :**

$d = 2^{m-1}$

distance minimale de Hamming entre deux codes

- **erreurs corrigeables :**

$\lfloor 2^{m-1}-1 / 2 \rfloor$

- **taux d'information**

$m+1 \ / \ 2^m$

message à encoder : (a0, a1, … , am)

l'encodage c → 2^m-uplet qui correspond à la combinaison linéaire des 2^m mots binaires dans l'ordre lexicographique pondérée par les ai

colonnes de la matrices génératrice → tous ces mots binaires

i allant de 0 à m : $c_i = a_0 + a_1 x_1[i] + \dots + a_m x_m[i]$

$$\underline{c} = [\, c_0, \dots, c_m \,] = [\, a_0, \dots, a_m \,] \begin{bmatrix} \underline{x_0}[0] & \dots & \underline{x_0}[2^m-1] \\ \dots & \dots & \dots \\ \underline{x_m}[0] & \dots & \underline{x_m}[2^m-1] \end{bmatrix}$$

décoder ai : $\underline{r}$ le mot reçu i : 1 à m

sommer $\underline{r}[a]$ et $\underline{r}[b]$ tel qu'ils correspondent à des paires identiques sauf à la i^{e} position

ex 1110 et 1100 (2^{m-1} mots de ce type)

ici $a2 = \underline{r}[a] + \underline{r}[b]$

$= (a0 \cdot 1 + a1 \cdot 1 + a2 \cdot 1 + a3 \cdot 0) + (a0 \cdot 1 + a1 \cdot 1 + a2 \cdot 0 + a3 \cdot 0)$

$= 0 + 0 + a2 + 0$

si il n'y a pas eu d'erreur

vote de majorité sur les 2^{m-1} ai que l'on a calculé

Les codes de Reed Muller : decodage (r=1) concept

on a maintenant $a_1 \dots a_m$ et $\underline{r}$

décoder a_0 :

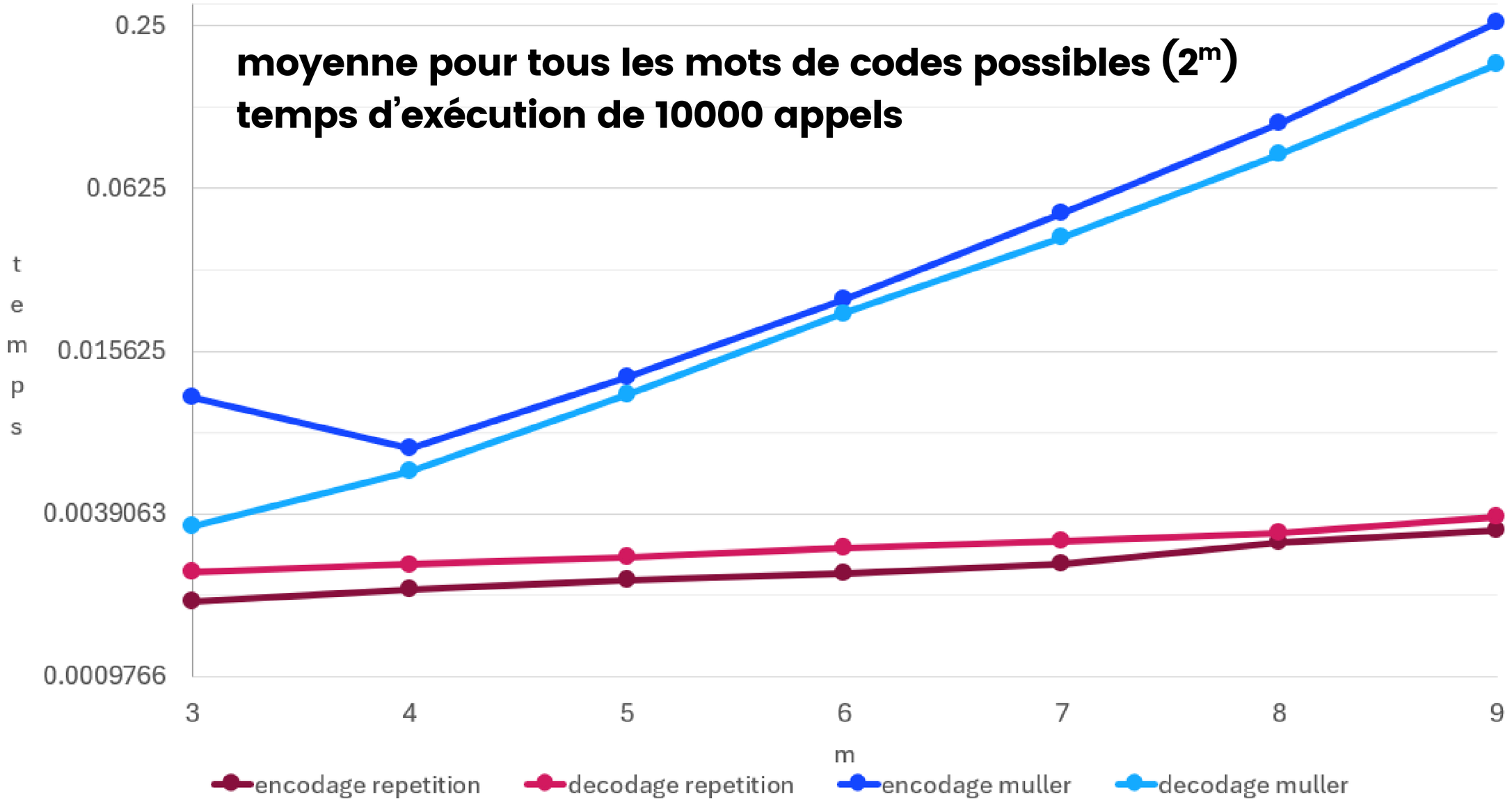
$$\underline{r}[i] = a_0 + a_1\underline{x_1}[i] + \dots + a_m\underline{x_m}[i]$$

$$a_0 = \underline{r}[i] + a_1\underline{x_1}[i] + \dots + a_m\underline{x_m}[i]$$ si il n'y a pas eu d'erreur

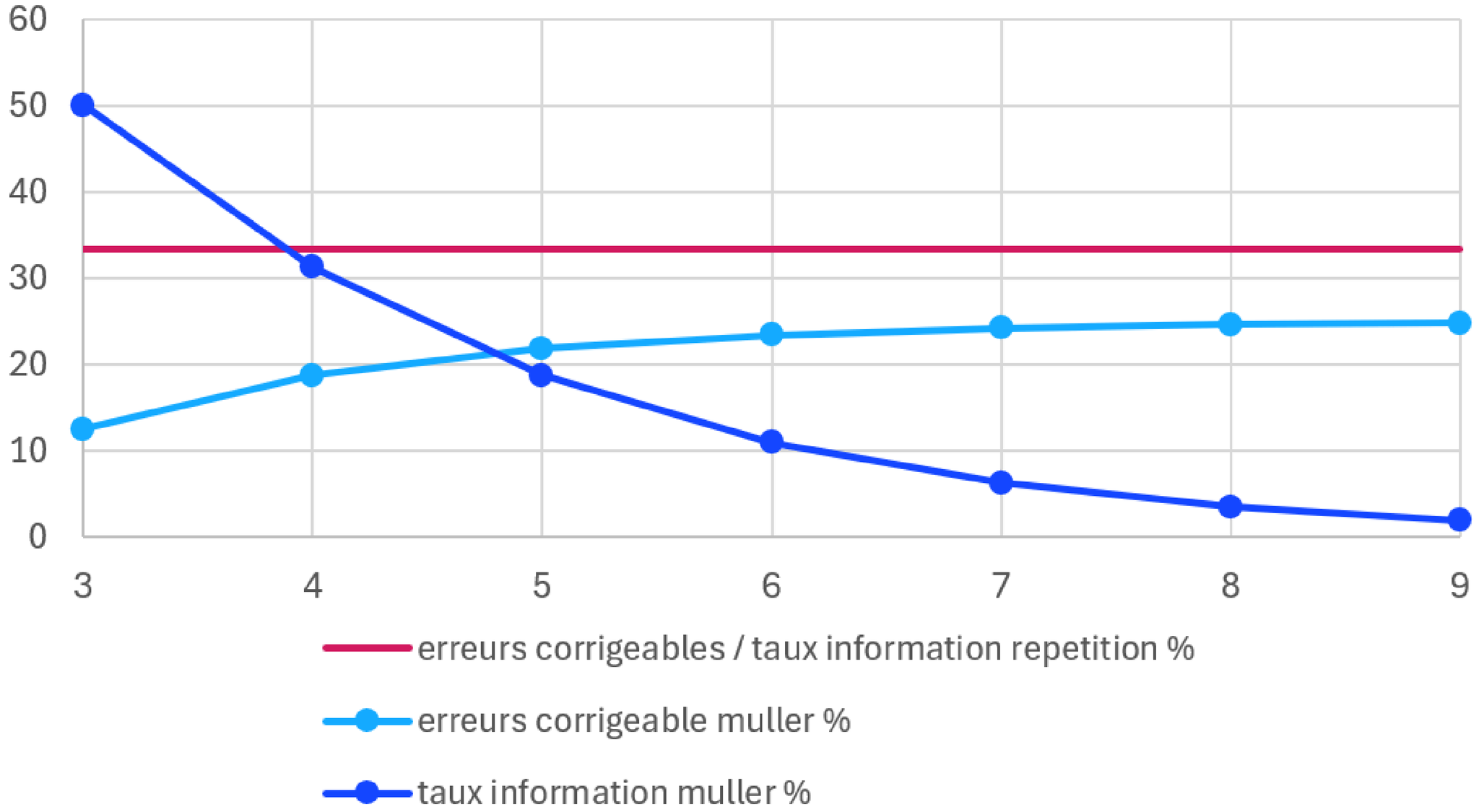
vote de majorité pour tout i

on a obtenu les $a_0, \dots, a_m$

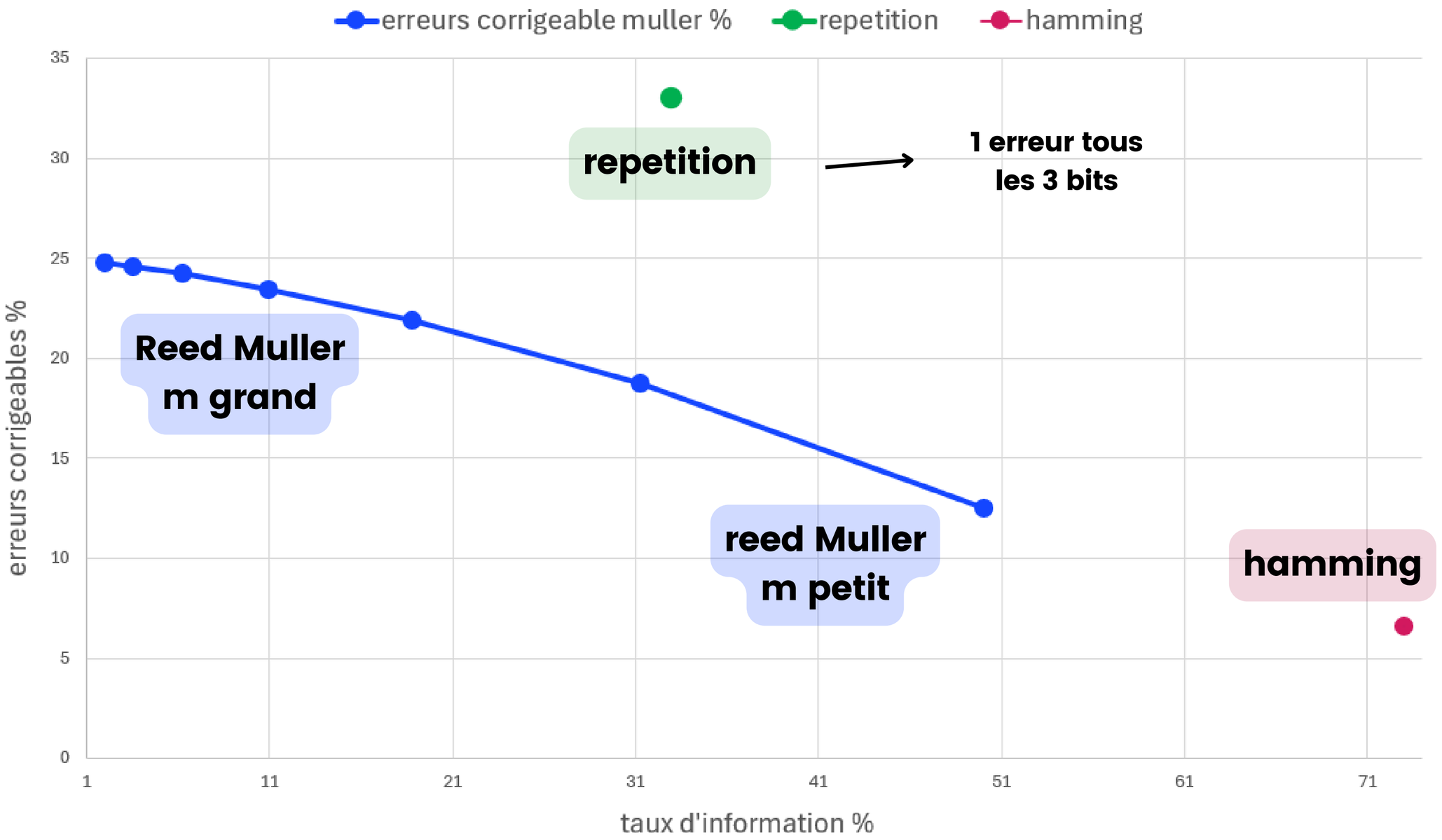
Les codes de Reed Muller : efficacité



Les codes de Reed Muller : résilience et taux d'information



Conclusion


erreurs corrigeable muller %
repetition
hamming
35
30
25
20
15
10
5
0
erreurs corrigeables %
repetition
1 erreur tous
les 3 bits
Reed Muller
m grand
reed Muller
m petit
hamming
1
11
21
31
41
51
61
71
taux d'information %

annexe

Les codes de Reed Muller : definition RM(r,m) et propriétés

- **base** : (1 , x1 , … , xm,
  x1•x2, …, xm−1•xm,
  …
  x1•…•xm)

- **matrice generatrice** :

$$\begin{Bmatrix} 1 \\ x1 \\ \dots \\ x1 \bullet x2 \\ \dots \\ x1 \bullet \dots \bullet xm \end{Bmatrix}$$

- **distance** : $d = 2^{(m-r)}$
  distance entre deux codes

- **erreurs corrigeables** :
  $\lfloor (d-1)/2 \rfloor$

- **taux d'information**
  $|base| / 2^m$
  $= \Sigma(i=0 \rightarrow r) \binom{m}{i} / 2\text{^}m$

corriger un maximum d'erreurs : maximiser la distance

erreurs corrigeables : $\lfloor (d-1)/2 \rfloor$ $\qquad d = 2^{(m-r)}$

-> minimiser r

les codes RM(1, m) corrigent les plus d'erreurs

- **distance :** $d = 2^{m-1}$
- **erreurs corrigeables :** $\lfloor (d-1)/2 \rfloor$
- **taux d'information :** $m+1 / 2^m$

mot nul : est un mot de code

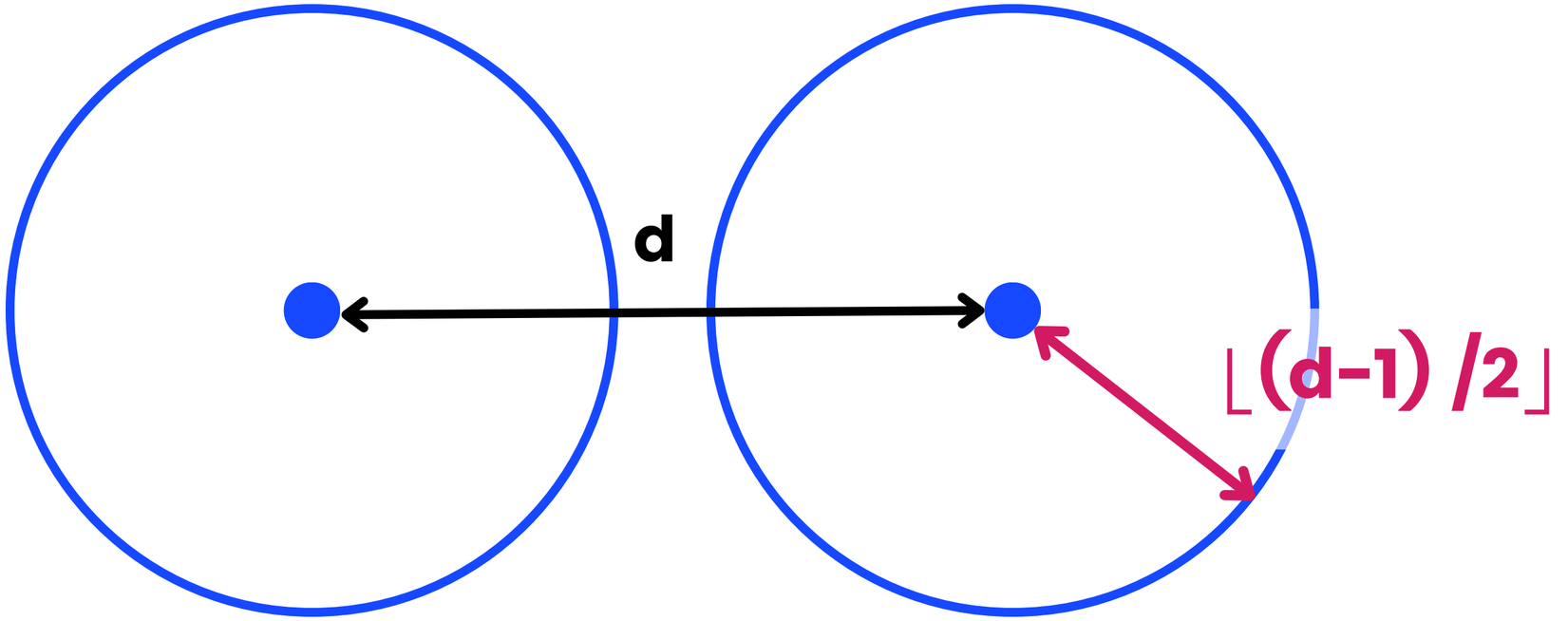
distance entre lui et le mot de poids minimal : poids de ce mot
c'est la distance minimale

mot de poids minimal : monôme de degré r (*)

poids minimal (nombre de bits à 1) : 2^{m-r}

(*) multiplication de deux mots : le $\underline{a} \bullet \underline{b}[i] = 1$ ssi $\underline{a}[i]=1$ et $\underline{b}[i]=1$
plus le degré est grand moins il y a de 1

Les codes de Reed Muller : formule du nombre d'erreurs dorrigeables



distance où il n'y a plus d'ambiguïté

s'apparente à des classes d'équivalences où les mots de codes sont les représentants

	m	longeur du code	erreurs corrigeables	%erreurs possibles	bits d'information	% information
	repetition			33		33
	hamming	15	1	7	11	73
reed muller	3	8	1	13	4	50
	4	16	3	19	5	31
	5	32	7	22	6	19
	6	64	15	23	7	11
	7	128	31	24	8	6
	8	256	63	25	9	4
	9	512	127	25	10	2

Code

```c
int* random_msg(int l){
    int* msg = malloc(l*sizeof(int));
    int a;
    for(int i=0; i< l; i=i+1){
        a=(rand()+i)%2;
        modif_msg(msg,l, i, a);
    }
    return msg;
}
```

```c
void randp_noize(int* msg, int max_noize, int p, int l){
    // p la probabilité qu un bit soit changé
    int proba;
    int compteur = 0;
    for(int i=0 ; i<max_noize; i=i+1){
        proba = rand()%(100+1) ;
        int k = rand()%(l);
        if(proba<=p ){
            modif_msg(msg,l, k, (msg[i]+1)%2);
            compteur = compteur+1;
        }
    }
}
```

```c
int** codage_repetition(int* message, int l){

    int** code = malloc(l*sizeof(int*));
    for(int i=0; i<l; i=i+1){
        code[i]=malloc(3*sizeof(l));
    }

    for(int i=0; i<l; i=i+1){
        for(int j=0; j<3; j=j+1){
            code[i][j]=message[i];
        }
    }
    return code;
}
```

```c
int* decodage_repetition(int** code_r, int l){

    int* message = malloc(l*sizeof(int));
    int i=0;
    int j=0;

    for(int i=0; i<l; i=i+1){
        message[i]= majorite(code_r[i][0], code_r[i][1], code_r[i][2]);
    }
    return message;
}
```

Code

```c
void phi_xi(int i, int m, int* result){
    /** donne les xi (i=0 -> m-1) */
    int bit=0;
    int puiss = puiss2(m-(i+1));
    int j=0;
    while(j<puiss2(m)){ //on recommence en alternant 0 et 1
        for(int k=0; k<puiss; k=k+1){// partie que 0 ou que 1 long 2^(m-(i+1))
            result[j]=bit;
            j=j+1;
        }
        bit=(bit+1)%2;
    }
}
```

```c
int** mat(int r, int m){
    int l=calcul_nb_ligne(r,m); // hauteur / nb vecteurs de la mat
    int c = puiss2(m); // largeur / longueur des vecteurs

    int** matrice = malloc(l*sizeof(int*));
    if (matrice == NULL) { printf("erreur matrice"); }

    // premiere ligne : init en 1
    matrice[0]=malloc(c*sizeof(int));
    if (matrice[0] == NULL) { printf("erreur ligne0 matrice"); }
    for(int j=0; j<c; j=j+1){ matrice[0][j]=1; }

    // lignes degre 1
    if(r>=1){
        int* reference = malloc (l*sizeof(int));
        // tableau contenant le dernier facteur du vecteur i a la case i
        if (reference == NULL) { printf("erreur reference"); }
        for(int i=1;i<=m;i=i+1){
            matrice[i]=malloc(c*sizeof(int));
            if (matrice[i] == NULL) { printf("erreur ligne matrice"); }

            phi_xi(i-1,m,matrice[i]); // x0 sur la case 1
            reference[i-1]=i;
        }
    // lignes degré sup a 2
        if(r>=2){
            int ind_dern_vect = 1; // donne l indice du dernier vecteur
            int j = 2; // num du vect a multiplier +1
            //(premier vect de degre 2 : x0x1 et x1 est sur la 2e ligne)

            for (int i=m+1 ;i<l; i=i+1){
                matrice[i]= malloc(c*sizeof(int)); //lignes de la matrice
                if (matrice[i] == NULL)
                { printf("erreur ligne mat mult"); }
                mult(matrice[ind_dern_vect],matrice[j], matrice[i],c);
                // matrice[ind_dern_vect] = vecteur de degre k-1
                // matrice[j] = vecteur de degre 1
                // matrice[i] = emplacement ou mettre le resultat
                reference[i-1]=j;
                if(j == m){
                    // quand on arrive sur une fin et que l on doit changer de vect de reference
                    j=reference[ind_dern_vect]+1;
                    ind_dern_vect = ind_dern_vect + 1;
                }else{
                    j=j+1; // cas normal
                }
            }
        }
        free(reference);
    }
    return matrice;
}
```

Code

```c
int* generer_mot_de_code(int* message, int** mat, int c, int m){
    // c est 2 puiss m : le nb de valuations possibles de x1...xm
    // on remarque que les colones de la matrice de degre 1 sont toutes les evaluations possibles
    //il suffit de multiplier les ai avec le bon coeff et de faire la somme.

    int* mot_de_code = malloc(c*sizeof(int));

    for(int i=0; i<c; i=i+1){
        mot_de_code[i]=0;
    }

    for(int col = 0; col<c; col=col+1){ // pour chaque colone (valuation de x1...xm)
        for(int ligne=0; ligne<=m; ligne=ligne+1){
            //on multiplie la ligne i-1(car on a commence a 0 et pas a 1) avec ai et on somme
            mot_de_code[col]=(mot_de_code[col]+ message[ligne]*mat[ligne][col]);
        }
        mot_de_code[col]=mot_de_code[col] %2;
    }
    return mot_de_code;
}
```

```c
int* decoder(int* code, int** matrice, int m, int c){
    // les a1 ... am sans a0
    int* valeurs_coefs = malloc((m+1)*sizeof(int));
    for(int i=0; i<=m; i=i+1){ // init coeffs a 0
        valeurs_coefs[i]=0;
    }
    int nb_sommes = puiss2(m-1);
    int* sommes = malloc( nb_sommes*sizeof(int));  /// il y a trop de pointeurs
    int a; int b;
    int compteur;
    int puiss ;
    for(int i = 1; i <= m; i = i + 1){
        compteur = 0;
        puiss = puiss2(m-i);
        for(int k = 0; k < c; k = k + 1){
            if ((k & puiss) == 0) { // selectionner a
                a = k;
                b = k | puiss;
                sommes[compteur] = (code[a] + code[b]) % 2;
                compteur = compteur +1;
            }
        }
        valeurs_coefs[i] = maximum_vraisemblance(sommes, nb_sommes);
    }
    //
    free(sommes);
    // pour a0
    int* mot_sans_a0 = malloc(c*sizeof(int));
    for(int i=0; i<c; i=i+1){
        mot_sans_a0[i]=0;       // init à 0
    }
    for(int ligne=1; ligne<=m; ligne=ligne+1){
        for(int col=0; col<c; col=col+1){
            mot_sans_a0[col]=mot_sans_a0[col]+valeurs_coefs[ligne]*matrice[ligne][col];
        }
    }
    for(int i=0; i<c; i=i+1){
        code[i]= (code[i] + mot_sans_a0[i])%2;
        //// on a change le code recu maintenant il contient des valeurs probable de a0
    }
    valeurs_coefs[0] = maximum_vraisemblance(code,c);
    free(mot_sans_a0);
    return valeurs_coefs;

}
```

Liste des fonctions

```
7   //////  definitions des objets
20      */
23      void controler_bit(int bit){ ···
30      }
31      int* cree_msg(int l){ ···
35      }
36      void modif_msg(int* msg, int l, int p, int bit){ ···
40      }
41      void inserer_msg( int* msg, int l){ ···
52      }
53      void lire_msg( int* msg, int l){ ···
59      }
60      int trouver_bit(int* msg, int puiss, int l){ ···
62      }
63      int* random_msg(int l){ ···
73      }
74      void noize(int* msg, int max_noize, int l){ ···
82      }
83      void randp_noize(int* msg, int max_noize, int p, int l){ ···
99      }
100 //
```

```
102 ////// codes simples : repetition ex 3
103
104     int** codage_repetition(int* message, int l){ ···
117     }
118     int majorite(int a, int b, int c){ ···
134     }
135     int* decodage_repetition(int** code_r, int l){ ···
145     }
146 //
```

```
148 ////// reed muller
149     // outils :
150
151     int puiss2(int n){ ···
161     }
162     int factorielle(int a){ ···
168     }
169     int calcul_nb_ligne(int r, int m){ // nb de lignes ···
176     }
177     int* mult(int* msg1, int* msg2, int* final,int c){ ···
183     }
184     void phi_xi(int i, int m, int* result){ ···
196     }
197     void affiche_matrice(int** matrice,int r, int m, int i){ ···
207     }
208     int** mat(int r, int m){ ···
260     }
261     int** multmat(int l1, int c1_l2, int c2, int** mat1, int** mat2){ ···
276     }
277 //
```

```
278 //codage
279
280     int* generer_mot_de_code(int* message, int** mat, int c, int m){ ···
299     }
300 //
301 //decodage
302
303     int maximum_vraisemblance(int* valeurs, int l){ ···
315     }
316
317     int* decoder(int* code, int** matrice, int m, int c){ ···
361     }
```

Main

```c
int main(void){
    printf("\n");
    srand(time(NULL));  // pour l'aléatoire
    int y = 10000;
    double* temps_encodage_repetition = malloc(y*sizeof(double));
    double* temps_decodage_repetition = malloc(y*sizeof(double));
    double* temps_encodage_muller = malloc(y*sizeof(double));
    double* temps_decodage_muller = malloc(y*sizeof(double));
    // CONSTANTES
        int r = 1;
        int m = 7 ;
        int c = puiss2(m);
        printf("r::%d, m::%d, c::%d\n",r,m,c);
        printf("\n");
        clock_t start, end;
        double cpu_time_used;
    //
    int** mots_possibles = mat(r,m);
    double moy_encodage_repetition =0;
    double moy_encodage_muller=0;
    double moy_decodage_repetition=0;
    double moy_decodage_muller=0;
    int* message = malloc((m+1)*sizeof(int));  // a enlever si on fait random
    for(int f=0; f<c; f=f+1){
        //MESSAGE
        for(int w=0; w<m+1; w=w+1){
            message[w]=mots_possibles[w][f];
        }
        //message = random_msg(m+1);
        //printf("message\n\n");
        //printf("    ");
        //lire_msg(message, m+1);
        //printf("\n");
        for(int z=0; z<y; z=z+1){
            //MATRICE
                start = time(NULL);  // debut encodage muller
                int** matrice = mat(r,m);
                int** matrice = mat(r,m);
                //printf("MATRICE \n\n");
                //affiche_matrice(matrice,r,m,calcul_nb_ligne(r,m));
                //printf("\n");
            //
            //  ENCODAGE
                int* code;
                //printf("ENCODAGE\n\n");

                /*int* message = malloc(4*sizeof(int));
                message[0]=1;
                message[1]=0;
                message[2]=1;
                message[3]=1;*/

                //printf("    ");
                code = generer_mot_de_code(message, matrice, c,m);
                end = time(NULL);  // fin encodage muller
                temps_encodage_muller[z]=(double)(end - start);
                //lire_msg(code, c);
                //printf("\n");
            //
            // DECODAGE
            //printf("ERREURS\n\n");
            randp_noize(code,1, 100, c);
            //printf("    ");
            //lire_msg(code, c);
            //printf("\nDECODAGE\n\n");
            start = time(NULL); // debut decodage muller
            int* decode = decoder(code, matrice, m,c);
            end = time(NULL); // end decodage muller
            temps_decodage_muller[z]=(double)(end - start);
            //printf("    ");
            //lire_msg(decode, m+1);
            /// CODE REPETITION
                //printf("\n CODE REPETITION : \n\n");
                int* message_repetition = random_msg(m+1);
                //lire_msg(message_repetition, m+1);
                ///// encodage
                start = time(NULL); // debut tps encodage repetition
                int** code_repetition = codage_repetition(message_repetition, m+1);
                end = time(NULL);// fin tps encodage repetition
                temps_encodage_repetition[z]=(double)(end - start);
                ///// erreurs
```

Main

```
446                 for(int i=0; i<m+1; i=i+1){
447                     randp_noize(code_repetition[i],1,100,3);
448                 }
449                 ///// decodage
450                 free(message_repetition);
451                 start = time(NULL); // debut tps decodage repetition
452                 message_repetition = decodage_repetition(code_repetition, m+1);
453                 end = time(NULL); // fin tps decodage repetition
454                 temps_decodage_repetition[z]=(double)(end - start);
455                 free(message_repetition);
456                 for(int h = 0; h<m+1; h=h+1){
457                     free(code_repetition[h]);
458                 }
459                 free(code_repetition);
460             //
461 >           // free···

471         }
472         for(int z=0; z<y; z=z+1){
473             moy_decodage_muller = moy_decodage_muller + temps_decodage_muller[z];
474             moy_encodage_muller = moy_encodage_muller + temps_encodage_muller[z];
475             moy_decodage_repetition = moy_decodage_repetition + temps_decodage_repetition[z];
476             moy_encodage_repetition = moy_encodage_repetition + temps_encodage_repetition[z];
477         }
478     }
479     free(message);
480     free(temps_decodage_muller);
481     free(temps_decodage_repetition);
482     free(temps_encodage_muller);
483     free(temps_encodage_repetition);
484     moy_decodage_muller = moy_decodage_muller/c ;
485     moy_encodage_muller = moy_encodage_muller/c ;
486     moy_decodage_repetition = moy_decodage_repetition/c;
487     moy_encodage_repetition = moy_encodage_repetition/c;
488     printf("\n\ner : %f, dr %f, em : %f, dm : %f\n\n", moy_encodage_repetition,moy_decodage_repetition,moy_encodage_muller,moy_decodage_muller);
489     for(int i=0; i<m+1; i=i+1){
490         free(mots_possibles[i]);
491     }
492     free(mots_possibles);
493 }
```