

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 ?**

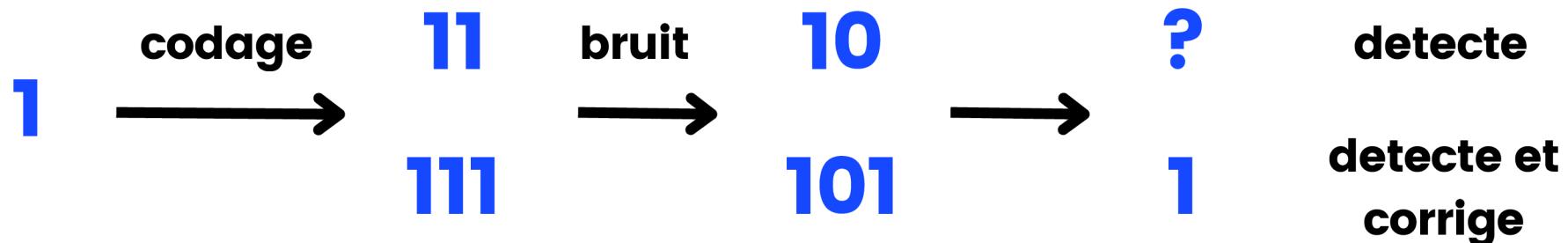
GRIMAUD Krawlya : 34692

Pourquoi utiliser des codes correcteurs ? comment procéder ?

- **perturbations** :



- **redondance** :



- bit de contrôle :

$1+1 = 2 \text{ pair}$

1010 0

bruit



1110 0

$1+1+1 = 3 \text{ impair} \rightarrow \text{erreur}$

- code de hamming

structure en carré, placement avisé de plusieurs bits de contrôle

travail sur lignes et colonnes → 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)

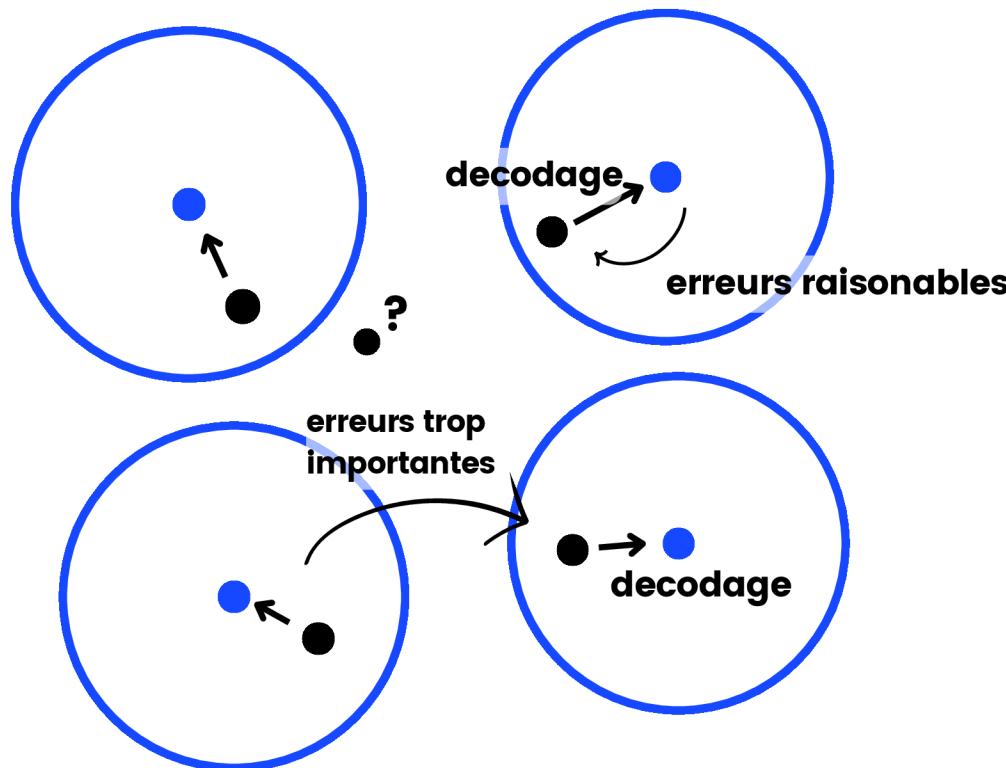
nombre d'erreurs corrigables : $nb_e = \lfloor (d-1) / 2 \rfloor$

nombre d'erreurs qu'il est possible de corriger

taux d'information :

nombre de bits d'information / nombre de bits total

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



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

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

- paramètres :

$r \rightarrow \text{degré}$, $m \rightarrow \text{nombre de variables}$

$2^m - \text{uplets } \underline{x_1}, \dots, \underline{x_m}$

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

pour $m = 3$

$\underline{x_0} = 11111111$

$\underline{x_1} = 00001111$

$\underline{x_2} = 00110011$

$\underline{x_3} = 01010101$

- RM(1,m) :

toutes les combinaisons linéaires de $1, \underline{x_1}, \dots, \underline{x_m}$ de coefficients a_0, \dots, a_m

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

- base :

(1, x1, ... , xm)

- distance :

$$d = 2^{m-1}$$

distance minimale de Hamming entre deux codes

- matrice génératrice :

$$\begin{bmatrix} 1 \\ \underline{x1} \\ \cdots \\ \underline{xm} \end{bmatrix}$$

- erreurs corrigables :

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

- taux d'information

$$m+1 / 2^m$$

Les codes de Reed Muller : encodage (r=1)

message à encoder : (a_0, a_1, \dots, a_m)

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

colonnes de la matrices génératrice \rightarrow tous ces mots binaires

i allant de 0 à m : $c_i = a_0 + a_1x_1[i] + \dots + a_mx_m[i]$

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

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

décoder ai : r le mot reçu i : 1 à m

sommer r[a] et 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)

$$\begin{aligned} \text{ici } a_2 &= r[a] + r[b] \\ &= (a_0 \cdot 1 + a_1 \cdot 1 + a_2 \cdot 1 + a_3 \cdot 0) + (a_0 \cdot 1 + a_1 \cdot 1 + a_2 \cdot 0 + a_3 \cdot 0) \\ &= 0 + 0 + a_2 + 0 \end{aligned}$$

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 r

décoder a_0 :

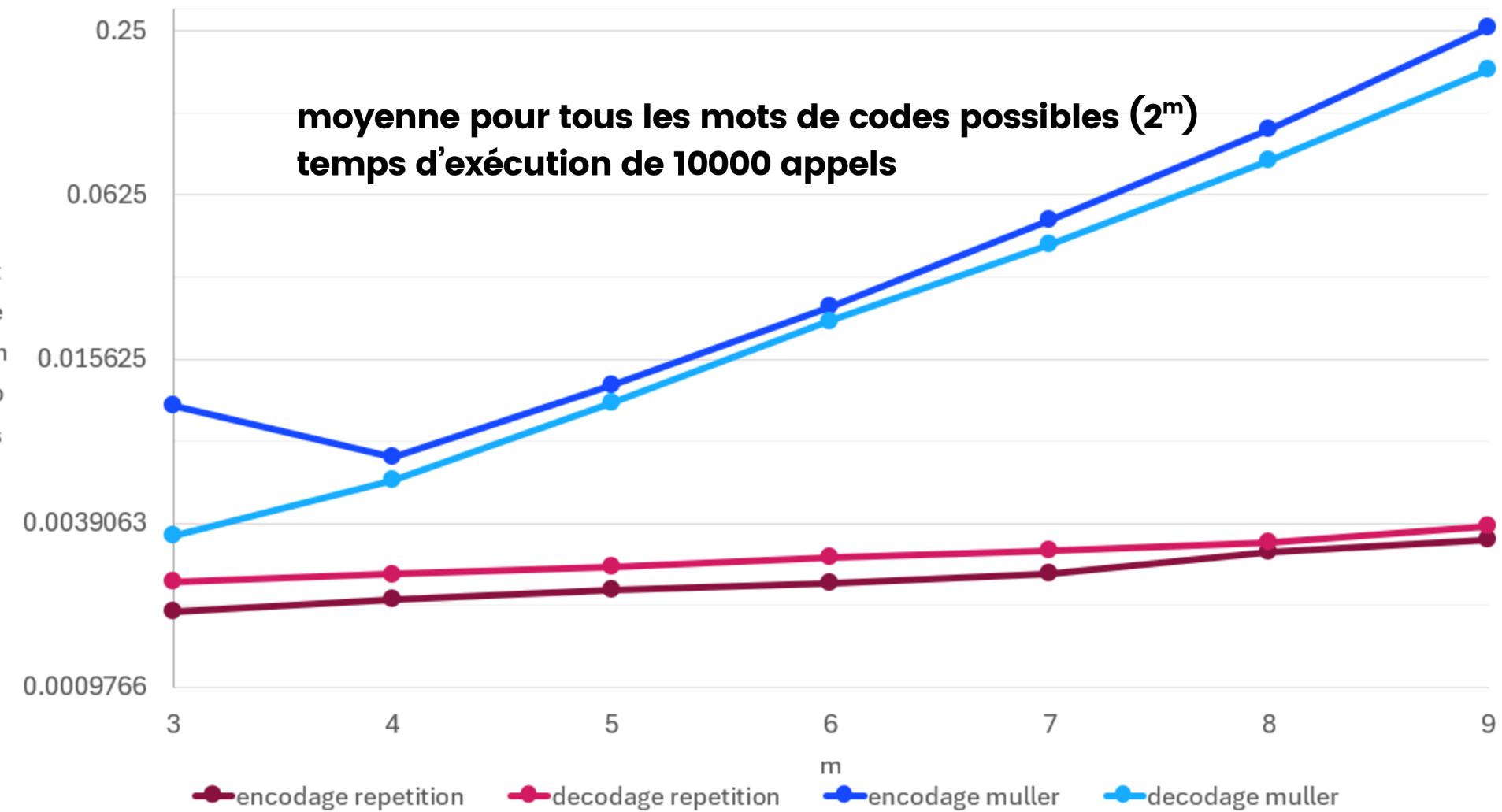
$$r[i] = a_0 + a_1 x_1[i] + \dots + a_m x_m[i]$$

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

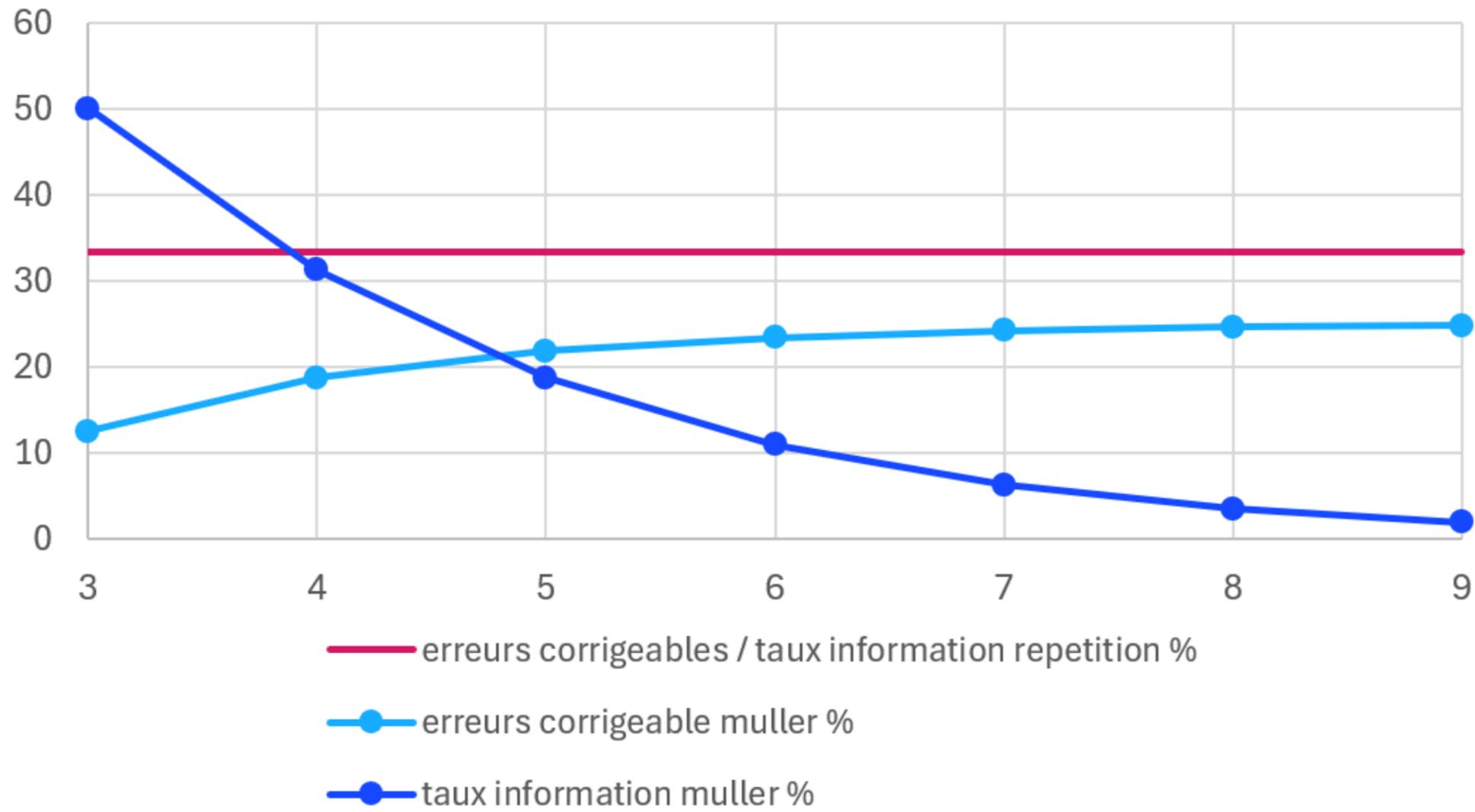
vote de majorité pour tout i

on a obtenu les a_0, \dots, a_m

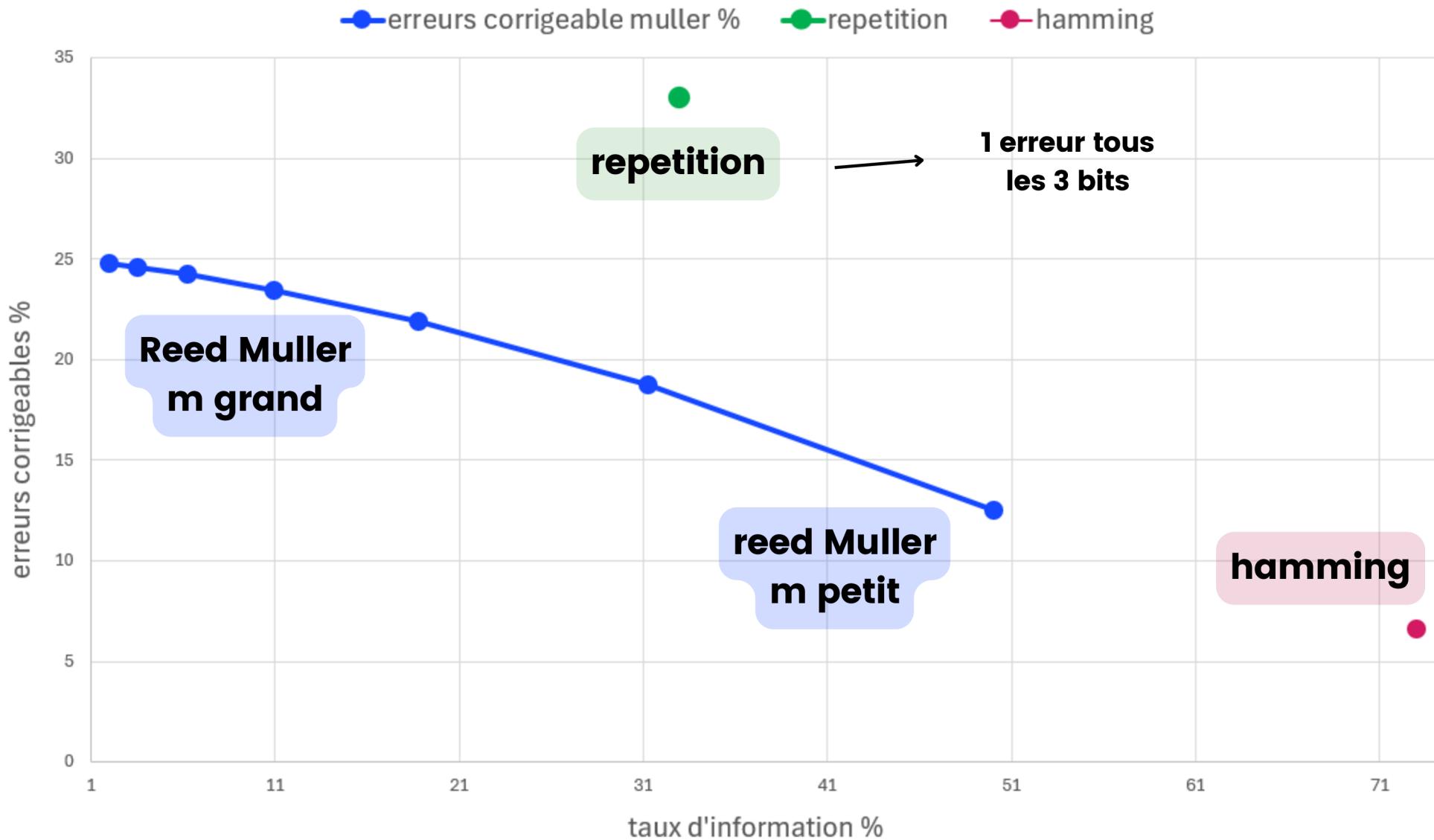
Les codes de Reed Muller : efficacité



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



Conclusion



annexe

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

- base : $(1, x_1, \dots, x_m,$
 $x_1 \cdot x_2, \dots, x_{m-1} \cdot x_m,$
 \dots
 $x_1 \cdot \dots \cdot x_m)$

- matrice génératrice :
- $$\left\{ \begin{array}{l} 1 \\ x_1 \\ \dots \\ x_1 \cdot x_2 \\ \dots \\ x_1 \cdot \dots \cdot x_m \end{array} \right\}$$

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

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

- taux d'information
 $|base| / 2^m$
 $= \sum_{i=0}^r \binom{m}{i} / 2^m$

corriger un maximum d'erreurs : maximiser la distance

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

-> minimiser r

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

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

mot nul : est un mot de code

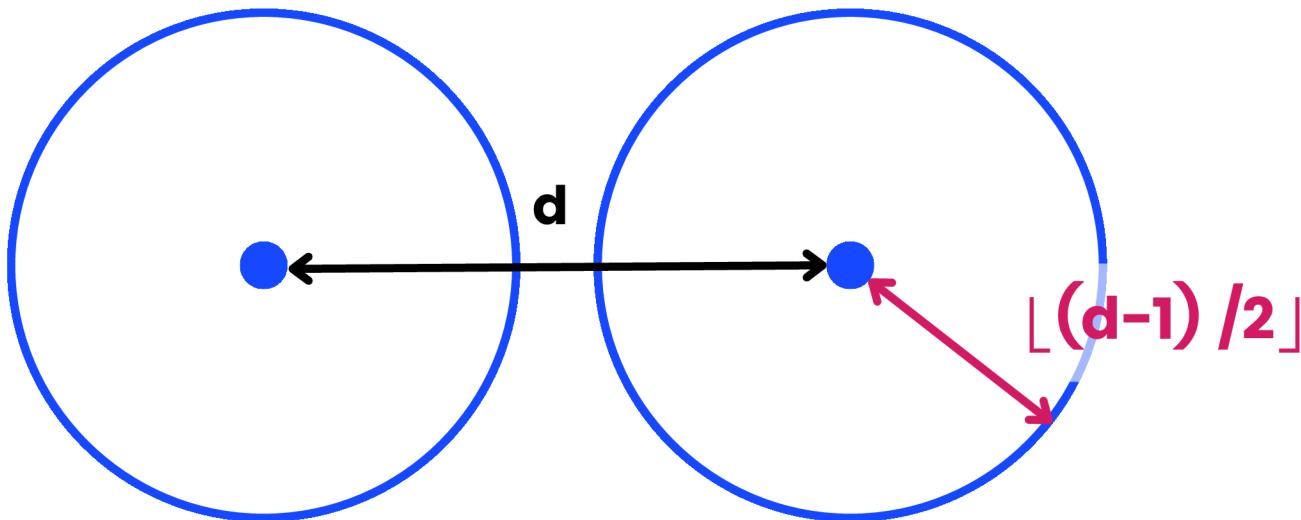
**distance entre lui et le mot de poids minimal : poids de ce mot
c'est la distance minimale**

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

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

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

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



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 corigeables	% 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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdbool.h>
4 #include <stddef.h>
5 #include <time.h>
6
7 /////////////// definitions des objets
8
9     typedef int* codeword;
10    typedef int* fcodeword;
11
12 /*
13     int* message : message normal
14     int* codeword : message transformé avec la matrice
15
16     int m : numero du plus grand xi
17     int lg : nombre de lignes de G
18     int cg : nombre de colonnes de G
19 */
20
21 int longueur_test;
22
23 /////////////// fonctions pour gerer les messages
24
25     void controler_bit(int bit){
26
27         /** verif bit = 0 ou 1 */
28         if(bit!=1 && bit!=0){
29             fprintf(stderr, "### : inserer 1 ou 0\n");
30             exit(0); //a changer pour dire qu'il y a eu un probleme
31         }
32     }
33
34     int* cree_msg(int l){
35         /** cree un msg de long <max_puiss> (pointeur sur int) */
36         int* msg = malloc((l)*sizeof(int));
37         return msg;
38     }
39
40     void modif_msg(int* msg, int l, int p, int bit){
41         /** modifie le bit de puissance p en bit dans msg */
42         controler_bit(bit);
43         msg[p] = bit;
44     }
45
46     void inserer_msg( int* msg, int l){
47         /* insere le msg scanne a l adresse donnee l ecrase ce qu'il y avait avant */
48
49         int bit;
50         printf("inserer msg de %d bits :\n",l+1);
51         for(int i=0 ; i<l; i=i+1){
52             printf("%d : ",i);
53             scanf("%d",&bit);
54             controler_bit(bit);
55             msg[l - i] = bit;
56         }
57     }
58
59     void lire_msg( int* msg, int l){
60         printf(": ");
61         for(int i=0; i< l ; i=i+1){
62             printf("%d", msg[i]);
63         }
64         printf("\n");
65     }
66
67     int trouver_bit(int* msg, int puiss, int l){
68
69         }
70         printf("\n");
71     }
72
73     int trouver_bit(int* msg, int puiss, int l){
74         return msg[l - puiss];
75     }
76
77     int* random_msg(int l){
78         int* msg = malloc(l*sizeof(int));
79         int a;
80
81         for(int i=0; i< l; i=i+1){
82             a=(rand()+i)%2;
83             modif_msg(msg,l, i, a);
84         }
85
86         return msg;
87     }
88
89     void noize(int* msg, int max_noize, int l){
90         int number = rand()%(max_noize + 1);
91         for(int i=0; i<=number; i=i+1){
92
93             int k = rand()%(l+1);
94             modif_msg(msg,l,k, (msg[k]+1)%2);
95         }
96     }
```

Code

```
95     }
96 }
97
98 void randp_noize(int* msg, int max_noize, int p, int l){
99     //proba de changer :
100    //int number = rand()%max_noize + 1;
101    int proba;
102    int compteur = 0;
103
104    for(int i=0 ; i<max_noize; i=i+1){
105        proba = rand()%(100+1) ;
106        int k = rand()%(l); // entre 0 et l-1
107
108        //pourrait changer plusieurs fois le meme
109        //bit possible probleme avec p
110        if(proba<=p ){
111            modif_msg(msg,l, k, (msg[i]+1)%2);
112            compteur = compteur+1;
113        }
114    }
115
116 //
```

```
118 ///// codes simples : repetition ex 3
119
120 int** codage_repetition(int* message, int l){
121
122     int** code = malloc(l*sizeof(int*));
123     for(int i=0; i<l; i=i+1){
124         code[i]=malloc(3*sizeof(1));
125     }
126
127     for(int i=0; i<l; i=i+1){
128         for(int j=0; j<3; j=j+1){
129             code[i][j]=message[i];
130         }
131     }
132
133     return code;
134 }
```

```
135     int majorite(int a, int b, int c){
136         int count=0;
137         if(a==0){
138             count = count+1;
139         }
140         if(b==0){
141             count = count +1;
142         }
143         if(c==0){
144             count = count +1;
145         }
146         if(count>=2){
147             return 0;
148         }else{
149             return 1;
150         }
151     }
152
153     int* decodage_repetition(int** code_r, int l){
154
155         int* message = malloc(l*sizeof(int));
156
157         int i=0;
158         int j=0;
159
160         for(int i=0; i<l; i=i+1){
161             message[i]= majorite(code_r[i][0], code_r[i][1], code_r[i][2]);
162         }
163
164     //
```

```
166 ///// reed muller
167
168     // outils :
169
170     int puiss2(int n){
171         if(n==0){
172             return 1;
173         }else{
174             int p=1;
175             for(int i=1; i<n; i=i+1){
176                 p=p*2;
177             }
178         }
179     }
```

Code

```
1//  
178     }  
179     return p;  
180 }  
181  
182 int factorielle(int a){  
183     int fact=1;  
184     for(int i=1; i<=a; i=i+1){  
185         fact = fact*i;  
186     }  
187     return fact;  
188 }  
189  
190 int calcul_nb_ligne(int r, int m){ // nb de lignes  
191     /** somme des (k parmi m) vecteurs possibles de degre k (r=0->k) */  
192     int s=0;  
193     for(int k=0; k<=r; k=k+1){  
194         s = s + (factorielle(m)/(factorielle(m-k)*factorielle(k))) ;  
195     }  
196  
197     }  
198     *  
199 >     int* add(int* msg1, int* msg2){ ...  
200 }  
201 >     int* barre(int* msg){ ...  
202 }  
203 >     int prod_scal(int* u, int* v){ ...  
204 }  
205     */  
206  
207     int* mult(int* msg1, int* msg2, int* final,int c){  
208         /** calcule la mult de 2 vecteurs  
209         for(int i=0; i<c; i=i+1){  
210             final[i]= msg1[i]*msg2[i];  
211         }  
212         return final;  
213     }  
214  
215     void phi_xi(int i, int m, int* result){  
216         /** donne les xi (i=0 -> m-1) */  
217         int hit=0;
```

```
240     int bit=0;  
241     int puiss = puiss2(m-(i+1));  
242     int j=0;  
243     while(j<puiss2(m)){ //on recommence en alternant 0 et 1  
244         for(int k=0; k<puiss; k=k+1){// partie que 0 ou que 1 long 2^(m-(i+1))  
245             result[j]=bit;  
246             j=j+1;  
247         }  
248         bit=(bit+1)%2;  
249     }  
250  
251     void affiche_matrice(int** matrice,int r, int m, int i){  
252         int l = i;  
253         int c = puiss2(m);  
254  
255         for(int i=0; i<l; i=i+1){  
256             for(int j=0; j<c; j=j+1){  
257                 printf("%d ", matrice[i][j]);  
258             }  
259             printf("\n");  
260         }  
261     }  
262  
263     int** mat(int r, int m){  
264         int l=calcul_nb_ligne(r,m); // hauteur / nb vecteurs de la mat  
265         int c = puiss2(m); // largeur / longueur des vecteurs  
266  
267         int** matrice = malloc(l*sizeof(int*));  
268         if (matrice == NULL)  
269             { printf("erreur matrice"); }  
270  
271         // premiere ligne : init en 1  
272         matrice[0]=malloc(c*sizeof(int));  
273         if (matrice[0] == NULL)  
274             { printf("erreur ligne0 matrice"); }  
275         for(int j=0; j<c; j=j+1)  
276             { matrice[0][j]=1; }  
277  
278         // lignes degré 1  
279         if(r>=1)  
280             {  
281                 int* reference = malloc (l*sizeof(int));  
282                 // tableau contenant le dernier facteur du vecteur i à la case i  
283                 if (reference == NULL)  
284                     { printf("erreur reference"); }  
285                 for(int i=1;i<m;i=i+1)  
286                 {  
287                     matrice[i]=malloc(c*sizeof(int));  
288                     if (matrice[i] == NULL) { printf("erreur ligne matrice"); }  
289  
290                     phi_xi(i-1,m,matrice[i]); // x0 sur la case 1  
291                     reference[i-1]=1;  
292                 }  
293  
294                 // lignes degré sup à 2  
295                 if(r>=2)  
296                 {  
297             }
```

Liste des fonctions

```

294
295 // lignes degré sup à 2
296 if(r>=2)
297 {
298     int ind_dern_vect = 1; // donne l indice du dernier vecteur
299     int j = 2; // num du vect à multiplier +1
300     // (premier vect de degré 2 : x0x1 et x1 est sur la 2e ligne)
301
302     for (int i=m+1 ;i<l; i=i+1)

303
304     {
305         matrice[i] = malloc(c*sizeof(int)); // lignes de la matrice
306         if (matrice[i] == NULL)
307             { printf("erreur ligne mat mult"); }
308
309         mult(matrice[ind_dern_vect],matrice[j], matrice[i],c);
310         // matrice[ind_dern_vect] = vecteur de degré k-1
311         // matrice[j] = vecteur de degré 1
312         // matrice[i] = emplacement où mettre le résultat
313
314         reference[i-1]=j;
315
316         if(j == m)
317         {
318             // quand on arrive sur une fin et que l on doit changer de vect de ref
319             j=reference[ind_dern_vect]+1;
320             ind_dern_vect = ind_dern_vect + 1;
321         }
322         else
323         {
324             j=j+1; // cas normal
325         }
326     }
327     free(reference);
328 }
329
330 return matrice;
331 }

332 int** multmat(int l1, int c1_l2, int c2, int** mat1, int** mat2){
333     int** mat = malloc(l1*sizeof(int*));
334     // AB[i][j] = somme de 0 à c1_l2 des A[i][k]*B[k][j]
335     for(int i=0; i<l1; i=i+1)
336     {
337         // les i
338         mat[i]=malloc(c1_l2*sizeof(int));
339         for(int j=0; i<c2; j=j+1)

340
341     { // les j
342         for(int l=0; l<c1_l2;l=l+1)
343         {
344             // les k
345             mat[i][j] = mat[i][j] + mat1[i][l]*mat[l][j];
346         }
347     }
348 }
349
350
351 //codage
352
353 int* generer_mot_de_code(int* message, int** mat, int c, int m){
354     // c est 2 puis m : le nb de évaluations possibles de x1...xm
355     // on remarque que les colonnes de la matrice de degré 1 sont toutes les évaluations possibles
356     // il suffit de multiplier les ai avec le bon coeff et de faire la somme.
357
358     int* mot_de_code = malloc(c*sizeof(int));
359
360
361     for(int i=0; i<c; i=i+1){
362         mot_de_code[i]=0;
363     }
364
365     for(int col = 0; col<c; col=col+1){ // pour chaque colonne (évaluation de x1...xm)
366         for(int ligne=0; ligne<m; ligne=ligne+1)
367             // on multiplie la ligne i-1(car on a commencé à 0 et pas à 1) avec ai et on somme
368             mot_de_code[col]=(mot_de_code[col]+ message[ligne]*mat[ligne][col]);
369     }
370     mot_de_code[col]=mot_de_code[col] % 2;
371 }
372
373 //decodage
374
375 int maximum_vraisemblance(int* valeurs, int l){
376     int count =0;
377     for(int i=0; i<l; i=i+1){

378         if(valeurs[i]==0){
379             count=count+1;
380         }
381     }
382     if(count >= l/2){
383         return 0;
384     }else{
385         return 1;
386     }
387 }
388
389
390 int* decoder(int* code, int** matrice, int m, int c){

391     // les a1 ... am sans a0
392
393     int* valeurs_coefs = malloc((m+1)*sizeof(int));
394
395     for(int i=0; i<=m; i=i+1){ // init coeffs à 0
396         valeurs_coefs[i]=0;
397     }
398
399 }

```

Main

```
401     int nb_sommes = puiss2(m-1);
402
403     int* sommes = malloc( nb_sommes*sizeof(int)); // il y a trop de pointeurs
404
405
406     int a; int b;
407     int compteur;
408     int puiss ;
409
410     for(int i = 1; i <= m; i = i + 1){
411
412         compteur = 0;
413         puiss = puiss2(m-i);
414
415         for(int k = 0; k < c; k = k + 1){
416             if ((k & puiss) == 0) {
417                 a = k;
418                 b = k | puiss;
419
420                 sommes[compteur] = (code[a] + code[b]) % 2;
421                 compteur = compteur +1;
422
423             }
424
425             valeurs_coefs[i] = maximum_vraisemblance(sommes, nb_sommes);
426
427         }
428
429     }
430
431     free(sommes);
432
433     // pour a0
434     int* mot_sans_a0 = malloc(c*sizeof(int));
435
436     for(int i=0; i<c; i=i+1){
437         mot_sans_a0[i]=0; // init a 0
438
439         for(int ligne=1; ligne<=m; ligne=ligne+1){
440
441             for(int col=0; col<c; col=col+1){
442                 mot_sans_a0[col]=mot_sans_a0[col]+valeurs_coefs[ligne]*matrice[ligne][col];
443             }
444
445             for(int i=0; i<c; i=i+1){
446                 code[i]=(code[i] + mot_sans_a0[i])%2;
447             }
448             // on a change le code recu maintenant il contient des valeurs probable de a0
449         }
450
451         valeurs_coefs[0] = maximum_vraisemblance(code,c);
452         free(mot_sans_a0);
453
454         return valeurs_coefs;
455
456     }
457
458 //
```

```
468     int main(void){
469
470         printf("\n");
471         srand(time(NULL)); // pour l'aléatoire
472         int y = 10000;
473
474         double* temps_encodeage_repetition = malloc(y*sizeof(double));
475         double* temps_decodeage_repetition = malloc(y*sizeof(double));
476
477         double* temps_encodeage_muller = malloc(y*sizeof(double));
478         double* temps_decodeage_muller = malloc(y*sizeof(double));
479
480         // CONSTANTES
481         int r = 1;
482         int m = 7 ;
483         int c = puiss2(m);
484         printf("r:%d, m:%d, c:%d\n",r,m,c);
485         printf("\n");
486         clock_t start, end;
487         double cpu_time_used;
488
489         //
490
491         int** mots_possibles = mat(r,m);
492
493         double moy_encodeage_repetition =0;
494         double moy_encodeage_muller=0;
495         double moy_decodeage_repetition=0;
496         double moy_decodeage_muller=0;
497
498         int* message = malloc((m+1)*sizeof(int)); // a enlever si on fait random
499
500         for(int f=0; f<c; f=f+1){
501
502             //MESSAGE
503
504             for(int w=0; w<m+1; w=w+1){
505
506                 message[w]=mots_possibles[w][f];
507
508             }
509
510             //message = random_msg(m+1);
511             //printf("message\n\n");
512             //printf("    ");
513             //lire_msg(message, m+1);
514
515             //printf("\n");
516
517             //
518
519             for(int z=0; z<y; z=z+1){
520
521
522                 //MATRICE
523                 start = time(NULL); // debut encodeage muller
524                 int** matrice = mat(r,m);
525                 //printf("MATRICE \n\n");
526                 //affiche_matrice(matrice,r,m,calcul_nb_ligne(r,m));
527                 //printf("\n");
528
529             }
```

Main

```
614
615     //printf("\n\n");
616
617 }
618
619
620
621 for(int z=0; z<y; z=z+1){
622     moy_decodage_muller = moy_decodage_muller + temps_decodage_muller[z];
623     moy_encodeage_muller = moy_encodeage_muller + temps_encodeage_muller[z];
624     moy_decodage_repetition = moy_decodage_repetition + temps_decodage_repetition[z];
625     moy_encodeage_repetition = moy_encodeage_repetition + temps_encodeage_repetition[z];
626 }
627
628 //moy_decodage_muller = moy_decodage_muller ;
629 //moy_encodeage_muller = moy_encodeage_muller ;
630 //moy_decodage_repetition = moy_decodage_repetition ;
631 //moy_encodeage_repetition = moy_encodeage_repetition;
632
633
634 }
635 free(message);
636 free(temps_decodage_muller);
637 free(temps_decodage_repetition);
638 free(temps_encodeage_muller);
639 free(temps_encodeage_repetition);
640
641
642 moy_decodage_muller = moy_decodage_muller/c ;
643 moy_encodeage_muller = moy_encodeage_muller/c ;
644 moy_decodage_repetition = moy_decodage_repetition/c;
645 moy_encodeage_repetition = moy_encodeage_repetition/c;
646 printf("\n\nner : %f, dr %f, em : %f, dm : %f\n\n", moy_encodeage_repetition,moy_decodage_repetition),moy_encodeage_muller,moy_decodage_muller);
647
648 for(int i=0; i<m+1; i=i+1){
649     free(mots_posibles[i]);
650 }
651 free(mots_posibles);
652 }
```