



Détection et correction d'erreurs: Hamming et Reed Solomon

Padovan Dorian 28434 MPI

Thème TIPE: Transition, transformation, conversion

Enjeux des codes correcteurs d'erreurs



- > Canaux bruités
- > Perte d'une partie du message
- > Contenu du message peut être altéré
- > Entrelacement

Plan



I- Vérifier la cohérence du message reçu

II- Description des codes de Hamming et Reed-Solomon

III- Implémentation et tests de performances

I-A) modèle d'un code et matrice génératrice



- Code linéaire $C(n, k)$: $\phi : (\mathbb{F})^k \longrightarrow (\mathbb{F})^n$
- Matrice génératrice $G \in M_{k,n}(\mathbb{F})$
- $A \in Gl_n(\mathbb{F})$: $G' = AG$ génère $C(n, k)$
- Si G est de la forme : $\begin{bmatrix} L & R \end{bmatrix}$
- Matrice normalisée est : $G' = \begin{bmatrix} I_k & T \end{bmatrix}$ avec $T = L^{-1}R$

I-A) Exemple matrice génératrice



code de parité $C(5, 4) : \phi(1000) = 10001$

$$x^T = [x_0, x_1, x_2, x_3]$$

$$\phi(x)^T = [x_0, x_1, x_2, x_3, x_0 + x_1 + x_2 + x_3 \bmod 2]$$

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

I-B) Matrice de contrôle

Soit $G' = \begin{bmatrix} I_k & T \end{bmatrix}$, la matrice génératrice normalisée de $C(n, k)$

Proposition 1

Soit H la matrice de contrôle (r, n) $H = \begin{bmatrix} T^T & -I_r \end{bmatrix}$. Alors $x^T = [x_1, \dots, x_n] \in C(n, k)$ si et seulement si $Hx = 0$

Par exemple pour le code de parité $C(5, 4)$: $H = [1, 1, 1, 1]$

II-A) Code de Hamming $C(2^n-1, 2^n-n-1)$



- idée clé : utiliser des bits de parité sur une partie du message
- Naïvement: 1000 -> 10001
- Hamming $C(7,4)$: 1000 -> 1110000

Codage d'un message avec le code de Hamming

message = 10001111010 -> 101000011111010

X 0000	1 0001	0 0010	1 0011
0 0100	0 0101	0 0110	0 0111
1 1000	1 1001	1 1010	1 1011
1 1100	0 1101	1 1110	0 1111

Codage de Hamming: 1-correcteur

message = 10001111010 -> 101010011111010

X 0000	1 0001	0 0010	1 0011
0 0100	1 0101	0 0110	0 0111
1 1000	1 1001	1 1010	1 1011
1 1100	0 1101	1 1110	0 1111

Trouver la position facilement de l'erreur avec l'addition bit à bit

	X	1	0	1
+0001	0000	0001	0010	0011
+0011				
+0101				
+1000	0	1	0	0
+1001	0100	0101	0110	0111
+1010				
+1011	1	1	1	1
+1100	1000	1001	1010	1011
+1110				
= 0101	1	0	1	0
	1100	1101	1110	1111

```

270 | t=[ i for i,bit in enumerate(message) if bit=='1']
271 | pos = 0
272 | for elt in t :
273 |     pos =pos ^ elt
274 | return pos

```

Codage de Hamming: 2 erreurs

message = 10001111010 -> 101010011110010

test renvoie 9=1001

X 0000	1 0001	0 0010	1 0011
0 0100	1 0101	0 0110	0 0111
1 1000	1 1001	1 1010	1 1011
0 1100	0 1101	1 1110	0 1111

Codage de Hamming étendu : détecter 2 erreurs

message = 10001111010 -> 0101010011110010

bits de parité sur
l'ensemble du
message

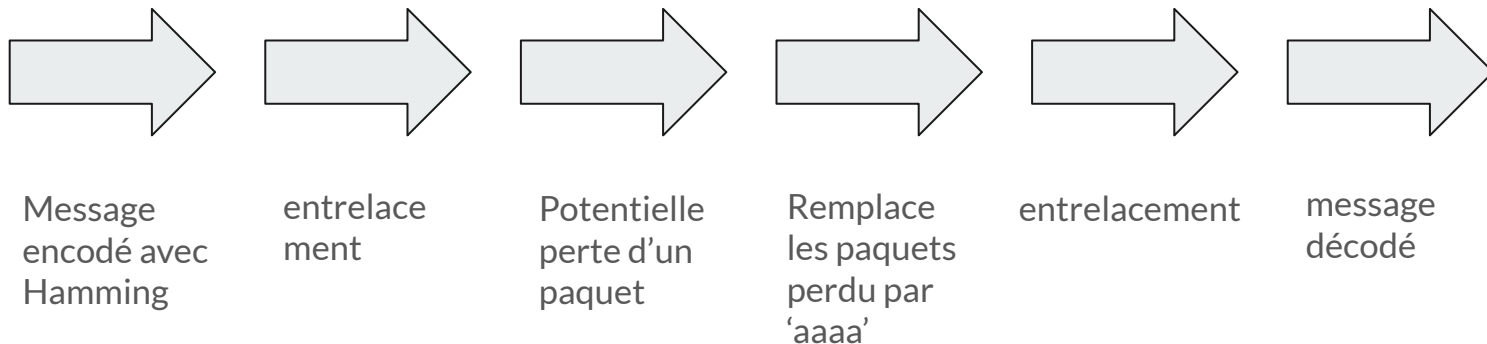
0 0000	1 0001	0 0010	1 0011
0 0100	1 0101	0 0110	0 0111
1 1000	1 1001	1 1010	1 1011
0 1100	0 1101	1 1110	0 1111

II-B)Entrelacement du message et Hamming

message découpé -> 'aaaa', 'bbbb', 'cccc', 'dddd'

message envoyé -> 'abcd', 'abcd', 'abcd', 'abcd'

message reçu -> 'abcd', 'abcd', ',', 'abcd'



```
686 assert message == hamming_decoding(entrelacement(perte_random(entrelacement(hamming_encoding(message,4)))))
```

II-C) Reed Solomon pour des pertes de données

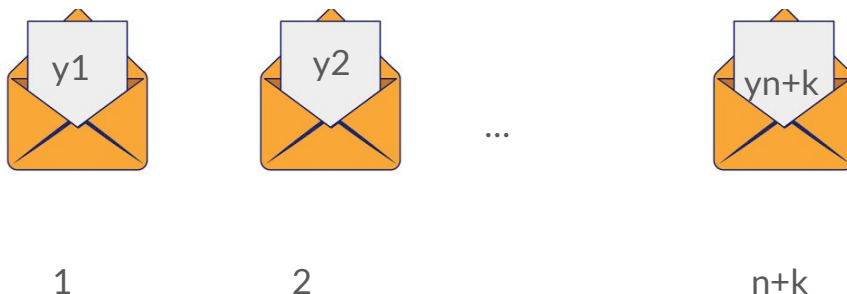
idée clé : Unicité des polynômes d'interpolation de Lagrange

1) Construction du polynôme Soit n couples de réels: $(1, y_1), (2, y_2), \dots, (n, y_n)$, on pose :

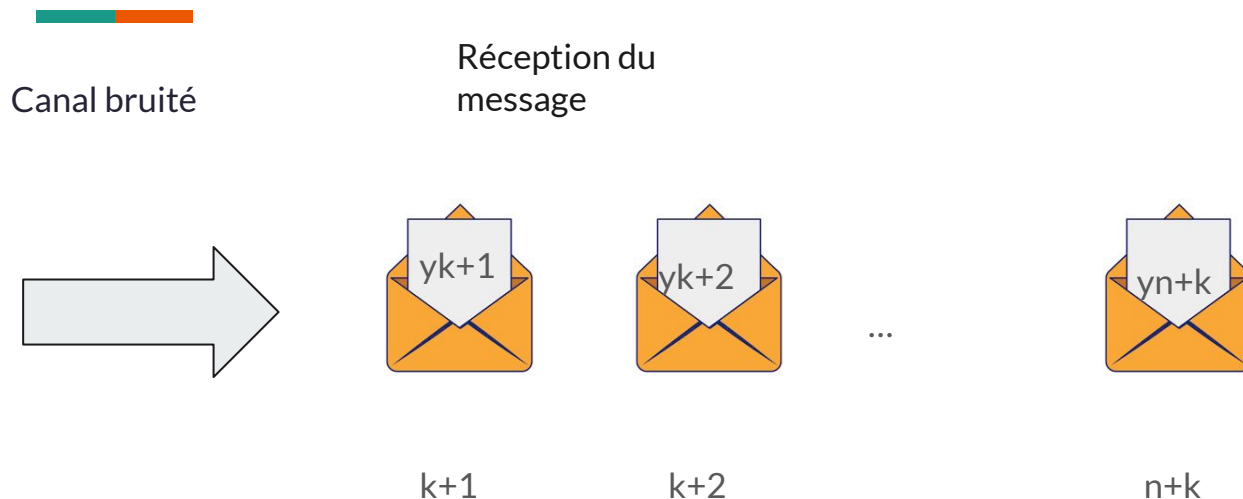
$$P(x) = \sum_{i=0}^n y_i \cdot \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x-j}{i-j} \text{ on a alors : } P(x_i) = y_i$$

2) Valeurs de redondance: évaluation de P en les valeurs : $(n+1, \dots, n+k)$

3) Envoie de $n+k$ paquets numérotés:



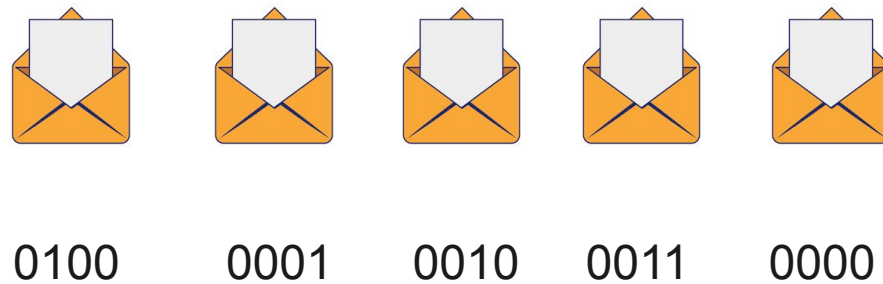
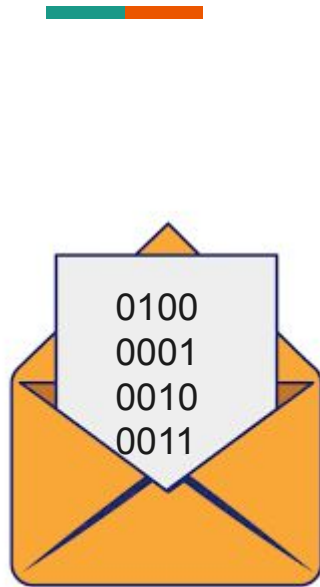
II-C) Reed Solomon pour des pertes de données



-> Reconstitue l'unique polynôme de Lagrange avec les n valeurs que l'on possède

-> On l'évalue en $1, 2, \dots, k$ pour retrouver le message

Encodage Reed-Solomon : perte de donnée



x=	0	1	2	3	4
P(x)=	4	1	2	3	0

$$(x_1, x_2, x_3, x_4) = (0, 1, 2, 3) \quad (y_1, y_2, y_3, y_4) = (4, 1, 2, 3)$$

-> Polynôme d'interpolation de lagrange: $P(X) = -\frac{2}{3}X^3 + 4X^2 - \frac{19}{18}X + 4$

$$\rightarrow P(4) = 0$$

III- A) Performance des codes de Hamming pour différentes valeurs de n

-> message = 'Hello! This is a simple ASCII text with numbers 1234567890 and symbols:
@#\$%^&*()-_+=[]{};:,<>?'

-> taille 672 bits en codage ASCII

-> Rajout de 0 à la fin du message

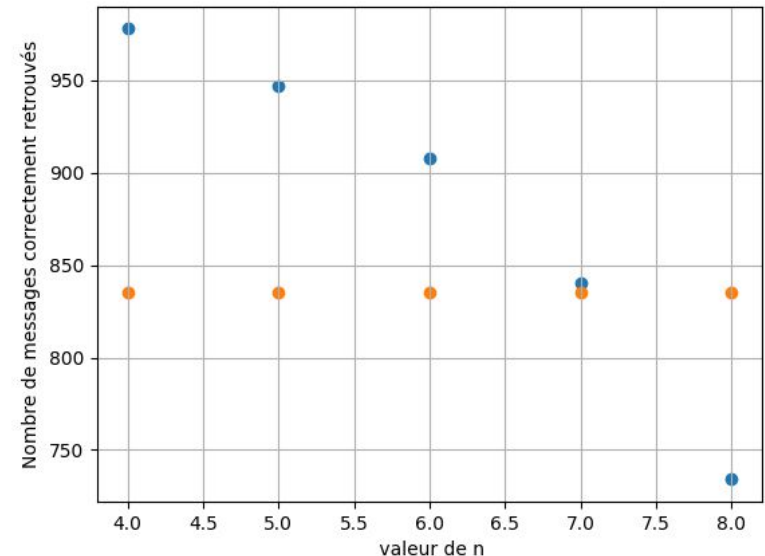
n=	3	4	5	6	7	8
taille	1352	992	832	768	768	768

III- A) Performance des codes de Hamming pour différentes valeurs de n



- > Pour la suite $n = 4$
- > probabilité de 1/500 qu'un bit soit altéré

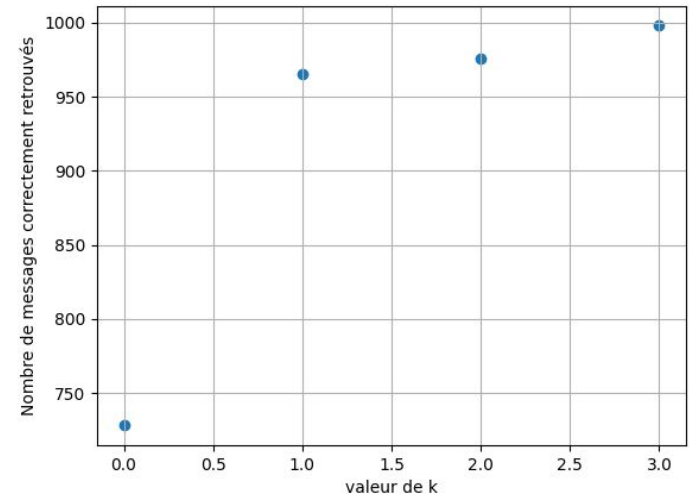
```
908 m='Hello! This is a simple ASCII text with numbers'
909 m=m+'1234567890 and symbols: @#$$%^&*()-_+[]{};:.,<>?'
910 mp = test_sans_cor(m,1000,500)
911 yy_values=[mp for i in range(len(x_values))]
912 for i in range(len(x_values)):
913     y_values.append(test_hamming(m,1000,x_values[i],500,0))
914     print(i)
915 plt.scatter(x_values,y_values) # points bleu
916 plt.xlabel(" valeur de n")
917 plt.scatter(x_values,yy_values) #points orange
918 plt.ylabel("Nombre de messages correctement retrouvés")
919 plt.grid()
920 plt.show()
```



||| -B) Performance Reed-Solomon pour différentes valeurs de k avec des pertes de paquets

- > Probabilité qu'un paquet soit perdu = $1/50$
- > Paquet de 64 bits (Norme IEEE 754)
- > Par la suite $k = 4$

```
969 x_values=[]
970 y_values=[]
971 nb=1000
972 m='This is a simple ASCII text '
973 for i in range(0,4):
974     print(i)
975     x_values.append(i)
976     y_values.append(test_RS(message,nb,50,i))
977 print(sum(y_values)/len(y_values))
978 plt.scatter(x_values,y_values)
979 plt.xlabel("valeur de k")
980 plt.ylabel("Nombre de messages correctement retrouvés")
981 plt.grid()
982 plt.show()
```

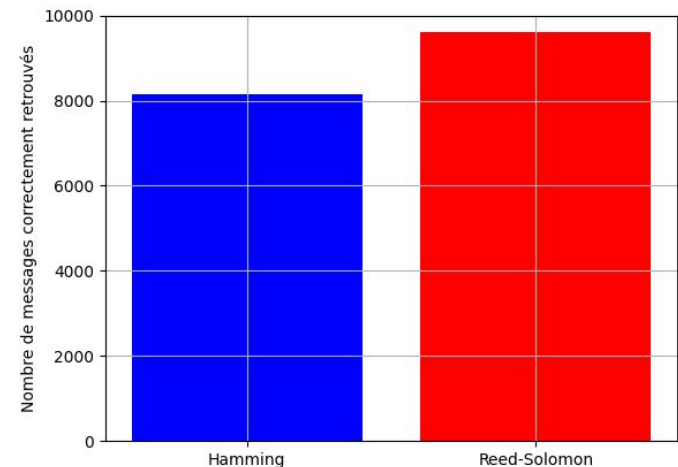


III-C)Hamming VS Reed-Solomon pour des pertes de paquets



- >message : This is a simple ASCII text
- > $n = 6$, $k = 4$
- >probabilité de $1/20$ qu'un paquet soit perdu
- >Hamming: 8152 , Reed-Solomon: 9598

```
998
999     nh=test_hamming_entrelace(message_h,nb,taille_hamming,20)
1000     nrs=test_RS(message_RS,nb,20,redondance_RS)
1001     categories = ['Hamming' , 'Reed-Solomon']
1002     values= [nh,nrs]
1003     colors=['blue','red']
1004     print(values)
1005     plt.bar(categories,values,color=colors)
1006     plt.ylim(0,nb)
1007     plt.xlabel("")
1008     plt.ylabel("Nombre de messages correctement retrouvés")
1009     plt.grid()
1010     plt.show()
```



Conclusion



Comparaison :

- Efficacité, complexité

Limites :

- Représentation binaire des nombres
- Amélioration possible



Merci de votre attention

Annexe 1 - Preuve matrice normalisé

Soit $G = A \cdot G'$, avec $A \in \mathcal{M}_k(\mathbb{F})$ une matrice inversible et G' une matrice génératrice de C .

Alors :

$$\text{Im}(G') = \{xG' \mid x \in \mathbb{F}^k\} = C,$$

et donc :

$$\text{Im}(G) = \{xG \mid x \in \mathbb{F}^k\} = \{xAG' \mid x \in \mathbb{F}^k\} = \{yG' \mid y = xA \in \mathbb{F}^k\} = \text{Im}(G').$$

Ainsi :

$$\text{Im}(G) = C.$$

Annexe 1 - Démonstration proposition 1

Preuve. Soit $x \in \mathbb{F}^n$, existe $z \in \mathbb{F}^k$ tel que $x = zG$ la matrice normalisé.
D'où :

$$xH^T = z[I_k \mid T] \begin{bmatrix} T^T \\ -I_k \end{bmatrix}$$

Par conséquent,

$$xH^T = z(T^T - T^T) = 0.$$

Réciproquement, si $xH^T = 0$ alors :

$$[x_1, x_2, \dots, x_n]H^T = 0,$$

Cela induit que pour $j = 1, \dots, r$ que :

$$[x_1, x_2, \dots, x_n][T_{1,j}, \dots, T_{k,j}]^T = 0,$$

Cela donne :

$$[x_{k+1}, \dots, x_{k+r}] = [x_1, \dots, x_k]T$$

Finalement, on a :

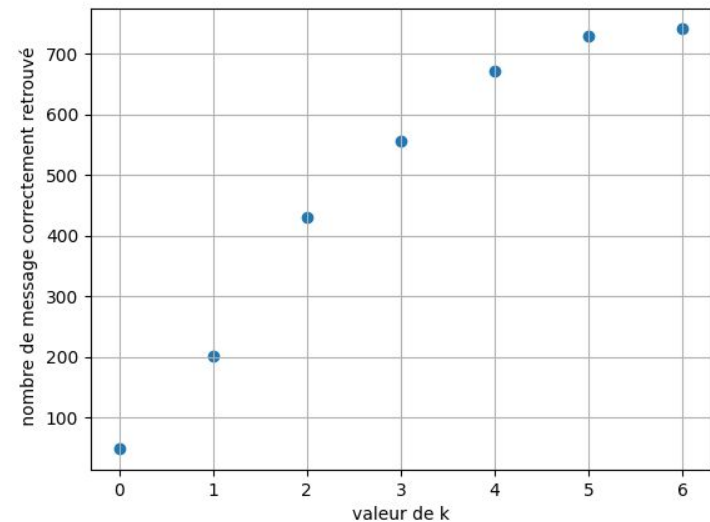
$$[x_1, \dots, x_n] = [x_1, \dots, x_k][Ik \mid T] = xG$$

ce qui montre que $x \in C$.

Annexe 2 - Performance Reed-Solomon corps fini avec différentes valeurs pour k

32 paquets de 7 bits

probabilité de 1/10 que le message soit perdu



Annexe 2 - Performance Reed-Solomon corps fini



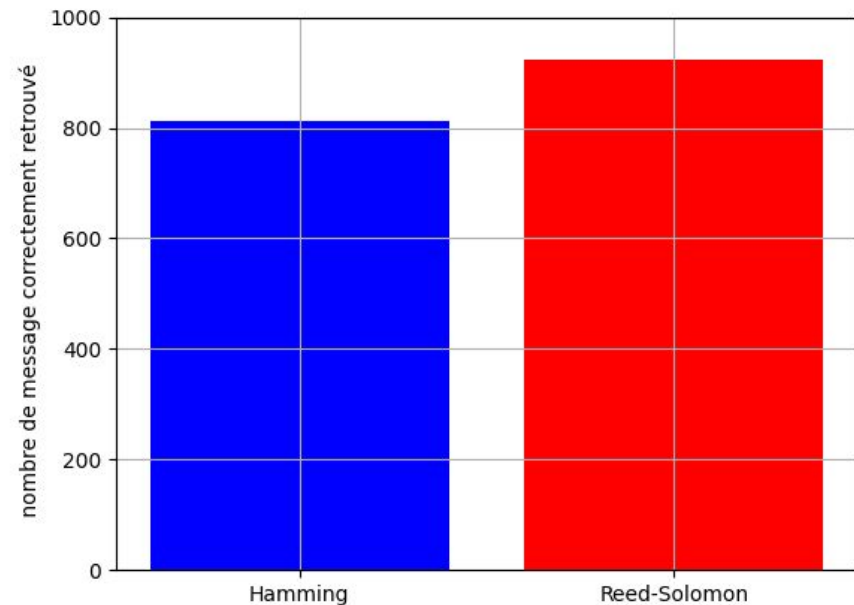
Message: This is a simple ASCII text

Probabilité de 1/20 qu'un paquet soit perdu

Paquets de 8 bits

Hamming(7,3): 813

Reed Solomon avec 5 caractères de redondance: 924



Annexe 3 code

```
2 from random import randint, random
3 from matplotlib import pyplot as plt
4 import numpy as np
5 from scipy.interpolate import lagrange
6 import struct
7
8
9 def modifie_i_eme(message: str, i: int, new: str):
10     """prend une chaîne de caractère et renvoie la même chaîne avec son ième caractère changé en new
11
12     Args:
13         message (str): chaîne
14         i (int): indice
15         new (str): nouveau caractère
16
17     Returns:
18         str: voir description
19     """
20     if i < 0 or i >= len(message): # Vérifie que l'indice est valide
21         return message
22     return message[:i] + new + message[i+1:]
23
24 def repr_bin_str(c: str):
25     """Convertit une chaîne en une représentation binaire
26
27     Args:
28         c (str): un str de taille 1
29
30     Returns:
31         str: représentation binaire de c
32     """
33     return format(ord(c), '07b')
34
35
36 def codage_binaire(chaine: str):
37     """converti une chaîne en une représentation binaire
38
39     Args:
40         chaine (str):
41
42     Returns:
43         str: concaténation de la représentation binaire de chacun des caractères de la chaîne en entrée
44     """
45     c = ''
46     for elt in chaine:
47         c = c + repr_bin_str(elt)
48     ..
```

Annexe 3 code

```
36 def codage_binaire(chaine:str):
48     return c
49
50 def uniforme(bit:str,proba:int):
51     """procède à l'opération binaire not sur le message avec une probabilité 1/p
52
53     Args:
54         bit (str): taille = 1
55         proba (int): proba>0
56
57     Returns:
58         str: renvoie l'opposé du bits
59     """
60     p = randint(1, proba)
61     if p == 1 :
62         return str(1-int(bit))
63     return bit
64
65 def random_change(message:str,loi:str,proba:int,period:int=0,taille_burst:int=0):
66     """modélise un canal bruité
67
68     Args:
69         message (str): message initial
70         loi (str): type d'erreurs expected: 'uniforme','periodique','burst_uni'
71         proba (int): probabilité qu'un bit ou qu'un ensemble de bits soit altéré
72         period (int, optional): si les erreurs arrivent de manière périodique. Defaults to 0.
73         taille_burst (int, optional): taille d'un bloc. Defaults to 0.
74
75     Returns:
76         str: le message avec probablement des erreurs
77     """
78     l=['uniforme','periodique','burst_uni']
79     #offset_period=randint(0,period-1)
80     assert loi in l
81     assert proba>1
82     n = len(message)
83     i=0
84     while i < n :
85         if loi == 'uniforme' :
86             message = modifie_i_eme(message,i,uniforme(message[i],proba))
87         elif loi == 'periodique':
88             assert period>0
89             if i%period == 0:
90                 message = modifie_i_eme(message,i,uniforme(message[i],1))
91         elif loi == 'burst_uni' :
92             assert taille_burst>0
93             if 1== randint(1, proba):
```

Annexe 3 code

```
65 def random_change(message:str,loi:str,proba:int,period:int=0,taille_burst:int=0):
93     if l== randint(1, proba) :
94         j = 0
95         while (i+j)<n and j<taille_burst :
96             message = modifie_i_eme(message,i,uniforme(message[i],1))
97             j = j +1
98             i=i+64
99     i=i+1
100     return message
101
102 def est_puissance_de_2(n:int):
103     """vérifie si n est une puissance de 2
104
105     Args:
106         n (int): un entier lambda
107
108     Returns:
109         bool: true ssi n est une puissance de 2
110     """
111     return n > 0 and (n & (n - 1)) == 0 # & et bit a bit
112
113 def decoupage_puissance_2(chaine:str,m:int):
114     """découpe un str en bloc de taille 2^m et rajoute des bits de parité pour correspondre au codage de hamming
115
116     Args:
117         chaine (str): str
118         m (int): >=4
119
120     Returns:
121         list: liste de str de taille 2^m
122     """
123     assert m>=3
124     n = len(chaine)
125     j = 0
126     p = 2 **m
127     c = ''.join([' ' for i in range(p)])
128     longueur=n//(p - m -1)
129     if longueur*11<n :
130         longueur= longueur +1
131     t=[c for i in range(longueur)]
132     h = 0
133     while j<n :
134         c = 'p'
135         if j + p -1-m< n :
136             for i in range(1,p):
137                 if est_puissance_de_2(i): # rajoute bits de redondance
```

Annexe 3 code

```
113 def decoupage_puissance_2(chaine:str,m:int):
138     c=c+ 'r'
139     else:
140         c =c+ chaine[j]
141         j = j +1
142     t[h]=c
143     h=h+1
144     else:
145         i = 1
146         while j <n:
147             if est_puissance_de_2(i):
148                 c= c + 'r'
149             else:
150                 c = c + chaine[j]
151                 j = j +1
152             i = i +1
153         for l in range(i,p): # on rajoute des 0 à la fin pour avoir un message de taille une puissance de 2
154             if est_puissance_de_2(l):
155                 c= c + 'r'
156             else:
157                 c = c + '0'
158                 j = j +1
159             if h<len(t) and len(t[h])<2**m :
160                 t[h]=c+'0'
161             else:
162                 t[h]=c
163             h=h+1
164     return t
165
166
167 def calculate_parity_positions(r:int):
168     """renvoie les positions des bits de parité dans un code de hamming
169
170     Args:
171         r (int): entier
172
173     Returns:
174         list: liste contenant les positions
175     """
176     # les positions de la case de redondance i sont ceux telles que le i eme bits de la case représenté est un 1
177     n = 2**r - 1
178     parity_positions = {i: [] for i in range(1, r+1)}
179     for i in range(1, n + 1):
180         for p in range(1, r + 1):
181             if (i & (2**(p - 1))):
182                 parity_positions[p].append(i)
```

Annexe 3 code

```
167 def calculate_parity_positions(r:int):
183     return parity_positions
184
185 def xor_bit(c1:str,c2:str):
186     """xor
187
188     Args:
189         c1 (str): '0' ou '1'
190         c2 (str): '0' ou '1'
191
192     Returns:
193         str: '0' ou '1'
194     """
195     if c1=='1':
196         if c2=='1':
197             return '0'
198         else:
199             return '1'
200     else:
201         if c2=='1':
202             return '1'
203         else:
204             return '0'
205
206 def parity_pos(pos:list,message:str):
207     """renvoie la somme (xor) des positions des bits '1' dans le message cela permet de trouver la valeur du bits de pari
208
209     Args:
210         pos (list): liste de position
211         message (str): chaine de '0' ou '1'
212
213     Returns:
214         str: '0' ou '1'
215     """
216     c = '0'
217     for elt in pos:
218         c = xor_bit(c,message[elt])
219     return c
220
221 def redondance(message:str,m:int):
222     """donne une valeurs aux bits de redondance
223
224     Args:
225         message (str): chaine de taille m
226         m (int): entier
227     """
```

Annexe 3 code

```
221 def redondance(message:str,m:int):
228     d = calculate_parity_positions(m)
229     for key in d:
230         indice = 2**(key-1)
231         message=modifie_i_eme(message,indice,parity_pos(d[key],message))
232     return message
233
234 def detecteur_1(message:str,m:int):
235     """Bit de détection d'erreur Hamming (bit 2)
236
237     Args:
238         message (str): str
239         m (int): pour faire une puissance de 2
240
241     Returns:
242         str: le message avec son premier caractère modifier pour hamming
243     """
244     n = 2**m
245     compte=0
246     for i in range(1,n):
247         if message[i]==1:
248             compte+=1
249     if compte%2==0:
250         return '0'+message[1:]
251     else:
252         return '1'+message[1:]
253
254 def positionErreur(message:str):
255     """
256     trouve la position de l'erreur s'il n'y en a qu'une
257     sinon renvoie 0
258     Args:
259         message (str): str
260     Returns:
261         int: indice de l'erreur
262     """
263     t=[ i for i,bit in enumerate(message) if bit=='1']
264     pos = 0
265     for elt in t :
266         pos =pos ^ elt
267     return pos
268
269 def correctionErreur(message:str):
270     """corrige une erreur
271
272     Args:
```


Annexe 3 code

```
269 def correction_erreur(message:str):
270     Args:
271         message (str): str
272
273     Returns:
274         str: message initial
275     """
276     i = position_erreur(message)
277     if i == 0 or i>=len(message):
278         return message
279     return modifie_i_eme(message,i,uniforme(message[i],1))
280
281 def hamming_encoding(message:str,m:int):
282     """codage de hamming du message avec des decoupages de taille 2^m
283
284     Args:
285         message (str): message initial
286         m (int): >=4
287
288     Returns:
289         list: liste de str de taille exactement 2^m représentant le message initial envoyé avec la redondance
290     """
291     c= codage_binaire(message) # transforme en binaire
292     #c='10001111010'
293     l = decoupage_puissance_2(c,m) # on découpe en plus petit message de taille 2**m
294     n = len(l)
295     for i in range(n):
296         l[i] = redondance(l[i],m) # calcul des bits de redondance
297         l[i] = detecteur_1(l[i],m) # calcul du bit de dectection
298     return l
299
300 def position_not_p2(m:int):
301     """pour un entier m donné renvoie un tableau contenant tous les entiers de 1 à m qui ne sont pas des puissances de 2
302
303     Args:
304         m (int): >=1
305
306     Returns:
307         list: voir description
308     """
309     pos=[]
310     for i in range(1,m):
311         if not est_puissance_de_2(i):
312             pos.append(i)
313     return pos
314
315 def hamming_decoding(l:list):
```

Annexe 3 code

```
317 def hamming_decoding(l:list):
318     """decode un message qui a subit une transformation de hamming
319
320     Args:
321         l (list): list de str de même taille contenant uniquement des 0 ou 1
322
323     Returns:
324         str: message decoder qui sera le message initial si il n'y a qu'une erreur
325     """
326     message=''
327     n= len(l)
328     m=len(l[0])
329     pos=position_not_p2(m)
330     for i in range(n):
331         l[i]=correction_erreur(l[i]) # corrige erreur de chaque morceau du message
332         for indice in pos:
333             message=message + l[i][indice] # on recupère le message sans les bits de redondance/parité
334     p = len(message)
335     h=p//7
336     m=''
337     for i in range(h): # on reconstruit le message ASCII
338         c=''
339         for j in range(7):
340             if message[7*i+j]!=' ': # ?
341                 c=c+message[7*i+j]
342             else:
343                 c=c+'0'
344         if len(c)==7:
345             asci_i = int(c,2)
346             if asci_i!=0:
347                 m=m+ chr(asci_i)
348     return m
349
350 #Reed-Solomon
351
352 def float64_to_bin(value:float):
353     """Convertit un float64 en binaire IEEE 754 (64 bits)."""
354     packed = struct.pack('!d', value) # '!d' = Big-endian double (64 bits)
355     return ''.join(f'{byte:08b}' for byte in packed)
356
357 def bin64_to_float(binary:str):
358     """Convertit une chaîne binaire IEEE 754 (64 bits) en float64."""
359     assert len(binary) == 64, "La chaîne doit contenir exactement 64 bits"
360
361     # Extraction des parties
362     signe = int(binary[0], 2)
```

Annexe 3 code

```
357 def bin64_to_float(binary:str):
358     # Extraction des parties
359     signe = int(binary[0], 2)
360     exposant = int(binary[1:12], 2)
361     mantisse = int(binary[12:], 2)
362
363     # Calcul du float64
364     if exposant == 0:
365         # Nombre subnormal (exposant = 0 → exposant réel = -1022)
366         valeur = (mantisse / (2**52)) * 2**-1022
367     elif exposant == 2047:
368         # Cas spéciaux : Infini ou NaN
369         if mantisse == 0:
370             valeur = float('inf')
371         else:
372             valeur = float('nan')
373     else:
374         # Nombre normalisé
375         valeur = (1 + mantisse / (2**52)) * 2**(exposant - 1023)
376
377     # Appliquer le signe
378     return -valeur if signe else valeur
379
380 def valeur_str(chaine:str):
381     """
382     renvoie pour chaine = m1m2m3m4 ...
383     Somme des mi*2**i
384
385     Args:
386         chaine (str): chaine de 0 et 1
387
388     Returns:
389         int:
390     """
391     v = 0
392     p = 1
393     for elt in chaine:
394         if elt == '1':
395             v = v + p
396             p = 2*p
397     return v
398
399 def inverse_valeur_str(v: int, p: int):
400     """inverse de la fonction précédente
401
402     Args:
403         v (int): la valeur renvoyer par la dernière fonction
```

Annexe 3 code

```
402 def inverse_valeur_str(v: int, p: int):
406     v (int): la valeur renvoyer par la dernière fonction
407     p (int): définir la taille de la chaîne renvoyer (rajoute des 0 non significatif)
408
409     Returns:
410     str:
411     """
412     if v == 0:
413         return "0".ljust(p, "0") # Retourne une chaîne de 14 zéros si v == 0
414
415     chaîne = ""
416     while v > 0:
417         chaîne += "1" if v % 2 == 1 else "0"
418         v //= 2
419
420     return chaîne.ljust(p, "0") # Complète avec des zéros à droite jusqu'à 14 caractères
421
422 def decoupage_taille_p(message: str, p: int):
423     """découpe un str en bloc de taille p
424
425     Args:
426     message (str): str
427     p (int): >0 ici p = 64
428
429     Returns:
430     list: liste de str de taille p
431     """
432     n = len(message)
433     q = n // p
434     r = n % p
435     l = ['' for i in range(p+1)]
436     for i in range(p+1):
437         c = ''
438         for j in range(q):
439             if i*q + j < n:
440                 c = c + message[i*q + j]
441         l[i] = c
442     if r != 0:
443         l.append(message[(p)*q+q:])
444     return l
445
446 def encodage_reed_solomon(message: str, redondance: int, p: int):
447     """
448     Code de Reed-Solomon avec des paquets.
449
450     Args:
```

Annexe 3 code

```
446 def encodage_reed_solomon(message:str,redondance:int,p:int):
450     Args:
451         message (str): message quelconque.
452         redondance (int): nombre de caractères de redondance.
453         p (int): taille des paquets.
454
455     Returns:
456         list: liste des paquets.
457     """
458     message=codage_binaire(message) # codage binaire
459     l = decoupage_taille_p(message,p) # découpage pour valeurs des polynomes
460     n = len(l)
461     m=n//2
462     x_points=np.array([i for i in range(n)])
463     for i in range(m,n):
464         x_points[i]+=redondance # ajouts de la redondance
465     y_points=np.array([np.float64(0) for i in range(n)])
466     for i in range(n):
467         y_points[i]=np.float64(valeur_str(l[i]))
468         l[i]=y_points[i]
469     P=lagrange(x_points,y_points) # construction du polynome
470     for r in range(redondance): # rajoute les valeurs de redondance
471         l.append(P(m+r))#float64_to_bin(P(m+r))
472     return l
473
474 def correction_perte(l,k):
475     """retrouve le message originale avec Polynome d'interpolation
476
477     Args:
478         l (list): de float64
479         k (int): nombre de caractères de redondance
480
481     Returns:
482         list: message
483     """
484     n = len(l)
485     m=(n-k)//2
486     x_points=np.array([i for i in range(n-k)])
487     for i in range(m,n-k):
488         x_points[i]+=k
489     y_values=['False' for i in range(n)]
490     c=0
491     for i in range(n):# on retrouve les valeurs non perdues
492         if l[i]!='':
493             y_values[i]=l[i]
494         else:
495             c=r+1
```

Annexe 3 code

```
477 def correction_perte(l,k):
478     c=c+1
479     if c>k:
480         return []
481     else:
482         x=[]
483         y=[]
484         dec=0
485         compteur=0
486         while compteur<n-k: # construction des tableaux x et y pour retrouver le polynome
487             if y_values[compteur]!='False':
488                 x.append(x_points[compteur])
489                 y.append(y_values[compteur])
490             else:
491                 x.append(m+dec)#x_points[compteur]+m-compteur+dec
492                 y.append(y_values[n-k+dec])
493                 dec+=1
494                 compteur+=1
495             if 'False' in y:
496                 return []
497             P=lagrange(x,y)
498             l_message=['' for i in range(n-k)]##on retrouve le message originale
499             m=(n-k)//2
500             i=0
501             j=0
502             while i<n:
503                 l_message[j]=(np rint((P(x_points[j]))))
504                 if i+1 ==m:
505                     i=i+k+1
506                 else:
507                     i=i+1
508                 j=j+1
509             return l_message
510
511 def retrouvemessageascii(l,p):
512     """reconstruction du message original
513
514     Args:
515         l (lst str): message binaire
516         p (int): taille des blocs
517
518     Returns:
519         str: message
520     """
521     nn=len(l)
522     n = p*nn
523     m=n//7
```

Annexe 3 code

```
528 def retrouvemessageascii(l,p):
540     m=n//7
541     message_bin=''
542     message=''
543     ll=l.copy()
544     for i in range(nn):
545         ll[i]=inverse_valeur_str(ll[i],p)
546         message_bin+=ll[i]
547     for i in range(m):
548         mm=''
549         if 7*i + 6 < n :
550             for j in range(7):
551                 mm+=message_bin[7*i+j]
552             message+=chr(int(mm,2))
553     mm=''.join([elt for elt in message if elt!='\x00' and elt!='\n' and elt!='\x0c' and elt!='\x05' and elt!='\x0b'])
554     return mm
555
556 def random_disappear(l,proba:int):
557     """probabilité de perdre le contenu d'un elt de la liste
558
559     Args:
560         l (lst str): liste
561         proba (int): plus grand que 1
562
563     Returns:
564         lst str: potentiellement des messages devenues vide
565     """
566     n = len(l)
567     for i in range(n) :
568         if random() < 1/proba :
569             l[i]=''
570     return l
571
572 def test_RS(m:str,int:int,proba:int,redondance:int):
573     """
574
575     Args:
576         m (str): message
577         int (int): nombre de fois que l'on test le procédé
578         taille (int): sur combien de bits utilisé hamming
579         proba (int): 1/proba = probabilité qu'un bits soit altéré (voir dans la fonction si uniforme ou pas)
580         redondance(int): nombre de redondance pour le polynome
581
582     Returns:
583         int: nombre de transmission correcte
584     """
```


Annexe 3 code

```
572 def test_RS(m:str,int:int,proba:int,redondance:int):
584     """
585     c=0
586     for i in range(int):
587         n=len(codage_binaire(m))//15 + 1
588         l=encodage_reed_solomon(m,redondance,n)
589         ll=random_disappear(l,proba) #probabilité d'enlever un bloc de 64 bits
590         l2= correction_perte(ll,redondance)
591         mm=retrouvemessageascii(l2,n)
592         if mm==m:
593             c=c+1
594         if i%(int/10)==0 and False:
595             print(i)
596     return c
597
598 def test_sans_cor(m:str,int:int,proba:int):
599     """
600
601     Args:
602         m (str): message
603         int (int): nombre de fois que l'on test le procedé
604         taille (int): sur combien de bits utilisé hamming
605         proba (int): 1/proba = probabilité qu'un bits soit altéré (voir dans la fonction si uniforme ou pas)
606
607     Returns:
608         int: nombre de transmission correcte
609     """
610     c=0
611     for i in range(int+1):
612         mm= random_change(m,'uniforme',proba)
613         if mm==m:
614             c=c+1
615     return c
616
617 def test_hamming(m:str,int:int,taille:int,proba:int,perce:int):
618     """Fonction de test du codage de Hamming dans un canal bruité.
619     On teste `int` fois la transmission et on compte le nombre de fois où le message est correctement retranscrit.
620
621
622     Args:
623         m (str): message
624         int (int): nombre de fois que l'on test le procedé
625         taille (int): sur combien de bits utilisé hamming
626         proba (int): 1/proba = probabilité qu'un bits soit altéré (voir dans la fonction si uniforme ou pas)
627         perce(int): = 0 corruption de donnée = 1 perte de donnée
628
629     Returns:
```


Annexe 3 code

```
617 def test_hamming(m:str,int:int,taille:int,proba:int,perte:int):
618     """
619     int: nombre de transmission correcte
620     """
621     c=0
622     corrompu= ''.join(['0' for i in range(2**taille)])
623     for i in range(int):
624         l=hamming_encoding(m,taille)
625         n=len(l)
626         for i in range(n):
627             if perte == 1 and random()<1/proba: # dans le cas d'une perte de donn  
628                 l[i]= corrompu
629             elif perte==0:
630                 l[i]= random_change(l[i],'uniforme',proba)
631         mm=hamming_decoding(l)
632         if mm==m:
633             c=c+1
634     return c
635
636 def entrelacement(l: list, p: int = 16):
637     """
638     Entrelace les bits de la liste 'l' par blocs de 'p' bits.
639     Chaque bloc est vu comme une ligne, et l'entrelacement se fait colonne par colonne.
640     """
641     n = len(l)
642     m = n // p # nombre de blocs
643     # Construire la matrice m x p (m lignes de p bits)
644     # Entrelacement : lire colonne par colonne
645     new_l = []
646     for k in range(m):
647         for i in range(p): # pour chaque colonne
648             mess=''
649             for j in range(p): # pour chaque ligne
650                 mess=mess+l[p*k+j][i]
651             new_l.append(mess)
652     for i in range(p*m,p*m+n%p):
653         new_l.append(l[i])
654     return new_l
655
656 message='This is a simple ASCII text '
657 n=len(codage_binaire(message)//15 + 1)
658 #print(len(hamming_encoding(message,6))* len(hamming_encoding(message,6)[0]))
659 #print(len(encodage_reed_solomon(message,4,n))* 64)
660 l1=[''.join([chr(j) for i in range(16)]) for j in range(97,97+16)]
661 l2=[''.join(format(j,'016b') ) for j in range(16)]
662 l1= l1 + l2
```

Annexe 3 code

```
676 def perte_random(l):
677     """
678     change un elt de la liste en 16 '0'
679     Args:
680         l (list): str
681
682     Returns:
683         list: str
684     """
685     n = len(l)
686     j=randint(1,n-1)
687     l[j]=''.join(['0' for i in range(len(l[j]))])
688     return l
689
690 m='This is a simple ASCII text This is a simple ASCII text This is a simple ASCII text This is a simple ASCII text This is a simple ASCII text'
691
692 assert entrelacement(entrelacement(l1))==l1
693 assert m == hamming_decoding(entrelacement(entrelacement(hamming_encoding(m,4))))
694 assert m == hamming_decoding(entrelacement(perte_random(entrelacement(hamming_encoding(m,4)))) # si 16 | len(hamming_encoding(m,4))
695
696
697 def poly_add(p1:list, p2:list, modulo:int):
698     """ addition de 2 polynomes modulo"""
699     length = max(len(p1), len(p2))
700     result = [0] * length
701     for i in range(length):
702         if i < len(p1):
703             a=p1[i]
704         else :
705             a=0
706         if i < len(p2):
707             b=p2[i]
708         else :
709             b=0
710         result[i] = (a + b) % modulo
711     return result
712
713 def poly_mul(p1:list, p2:list, modulo:int):
714     # multiplication de deux polynomes
715     result = [0] * (len(p1) + len(p2) - 1)
716     for i in range(len(p1)):
717         for j in range(len(p2)):
718             result[i + j] = (result[i + j] + p1[i] * p2[j]) % modulo
719     return result
720
721 def lagrange_poly(x:list, y:list, modulo=131):
```

Annexe 3 code

```
720
721 def lagrange_poly(x:list, y:list, modulo=131):
722     # renvoie sous forme d'une liste de coeffs le polynome d'interpolation de lagrange modulo modulo
723     n = len(x)
724     final_poly = [0]
725
726     for i in range(n):
727         # Construct L_i(x)
728         li_poly = [1]
729         denom = 1
730         for j in range(n):
731             if i != j:
732                 # Multiply by (x - xj)
733                 li_poly = poly_mul(li_poly, [-x[j] % modulo, 1], modulo)
734                 # Multiply denominator (xi - xj)
735                 denom = (denom * (x[i] - x[j])) % modulo
736
737         denom_inv = pow(denom, -1, modulo)
738         # Multiply L_i(x) by yi * denom_inv
739         li_scaled = [(coef * y[i] * denom_inv) % modulo for coef in li_poly]
740
741         # Add to the final polynomial
742         final_poly = poly_add(final_poly, li_scaled, modulo)
743
744     return final_poly
745
746
747 def format_polynomial(coeffs:list, modulo:int):
748     #pour afficher le polynôme
749     terms = []
750     for i, coeff in enumerate(coeffs):
751         coeff = coeff % modulo
752         if coeff == 0:
753             continue
754         if i == 0:
755             term = f"{coeff}"
756         elif i == 1:
757             term = f"{coeff}x" if coeff != 1 else "x"
758         else:
759             term = f"{coeff}x^{i}" if coeff != 1 else f"x^{i}"
760         terms.append(term)
761
762     if not terms:
763         return "0"
764     return " + ".join(terms[::-1]) + f" (mod {modulo})"
765
766
```

Annexe 3 code

```
766
767 def eval_poly(coeffs:list, x:int, modulo:int):
768     # renvoie P(x) où P est décrit par sa liste de ses coefficients
769     result = 0
770     for coef in reversed(coeffs):
771         result = (result * x + coef) % modulo
772     return result
773
774
775
776 def encodage_reed_solomon_fini(message:str, redondance:int, modulo=131):
777     """
778     Code de Reed-Solomon avec des paquets avec corps fini
779
780     Args:
781         message (str): message quelconque.
782         redondance (int): nombre de caractères de redondance.
783         modulo (int): modulo
784
785     Returns:
786         list: liste des paquets.
787     """
788     n = len(message)
789     l = decoupage_taille_p(message, n)
790     ll = [0 for i in range(n+redondance)]
791     x_points = [i for i in range(n)] # i+1 pour le test de la source
792     y_points = [ord(message[i]) for i in range(n)]
793     for i in range(n):
794         ll[i] = y_points[i]
795
796     P = lagrange_poly(x_points, y_points, modulo)
797     for r in range(redondance):
798         ll[n+r] = eval_poly(P, n+r, modulo)
799     return ll
800
801
802
803 def correction_perte_fini(l,k, modulo=131):
804     """retrouve le message originale avec Polynome d'interpolation modulo modulo
805
806     Args:
807         l (list): message potentiellement corrompu
808         k (int): nombre de caractères de redondance
809
810     Returns:
811         list: message
```

Annexe 3 code

```
803 def correction_perte_fini(l,k,modulo=131):
804     """
805     list: message
806     """
807     n = len(l)
808     x_points=[i for i in range(n)]
809     y_values=['False' for i in range(n)]
810     c=0
811     for i in range(n):# on retrouve les valeurs non perdues
812         if l[i]!='':
813             y_values[i]=(l[i])
814         else:
815             c=c+1
816     if c>k:
817         return [] # pas possible
818     else:
819         x=[]
820         y=[]
821         dec=0
822         compteur=0
823         while compteur<n-k:
824             if y_values[compteur]!='False':
825                 x.append(x_points[compteur])
826                 y.append(y_values[compteur])
827             else:
828                 x.append(x_points[n-k+dec])
829                 y.append(y_values[n-k+dec])
830                 dec+=1
831             compteur+=1
832         if 'False' in y:
833             return [] # pas possible
834         P=lagrange_poly(x,y,modulo)
835         l_message=['' for i in range(n-k)]
836         j=0
837         while j<n-k:
838             l_message[j]=eval_poly(P,x_points[j],modulo)
839             j=j+1
840         return l_message
841
842 def decodage_ascii_fini(l):
843     #renvoie le codage ASCII d'une liste d'entiers
844     return ''.join([chr(elt) for elt in l])
845
846 def test_RS_fini(m:str,int:int,proba:int,redondance:int):
847     """
848     Args:
```

Annexe 3 code

```
853 def test_RS_fini(m:str,int:int,proba:int,redondance:int):
854     Args:
855         m (str): message
856         int (int): nombre de fois que l'on test le procedé
857         taille (int): sur combien de bits utilisé hamming
858         proba (int): 1/proba = probabilité qu'un bits soit alteré (voir dans la fonction si uniforme ou pas)
859         redondance(int): nombre de redondance pour le polynome
860
861     Returns:
862         int: nombre de transmission correcte
863     """
864     c=0
865     for i in range(int):
866         l=encodage_reed_solomon_fini(m,redondance)
867         ll=random_disapear(l,proba) #probabilité d'enlever un bloc
868         l2= correction_perte_fini(ll,redondance)
869         mm=decodage_ascii_fini(l2)
870         if mm==m:
871             c=c+1
872         if i%(int/10)==0 and False:
873             print(i)
874     return c
875
876 def Reed_Solomon_fini_different_k():
877     message='This is a simple ASCII text '
878     ll=encodage_reed_solomon_fini(message,3)
879     print(len(ll))
880     print('test')
881     x_values=[]
882     y_values=[]
883     nb=1000
884     for i in range(0,7):
885         print(i)
886         x_values.append(i)
887         y_values.append(test_RS_fini(message,nb,10,i))
888     plt.scatter(x_values,y_values)
889     plt.xlabel("valeur de k")
890     plt.ylabel("Nombre de messages correctement retrouvés")
891     plt.grid()
892     plt.show()
893
894
895
896 # test 2**m avec m qui change pour hamming
897 def hamming_pour_different_n():
898     m='Hello! This is a simple ASCII text with numbers 1234567890 and symbols: @#$%^&*()-_=[\]{};:,.<>?'
899     ...
```


Annexe 3 code

```
898 def hamming_pour_different_n():
899     m='Hello! This is a simple ASCII text with numbers 1234567890 and symbols: @#%$^&*()-_+[]{};:,.<.>?'
900     print(len(codage_binaire(m)))
901     l=hamming_encoding(m,3)
902     print(len(l))
903     x_values=[3,4,5,6,7,8]
904     y_values=[]
905     m='Hello! This is a simple ASCII text with numbers'
906     m=m+'1234567890 and symbols: @#%$^&*()-_+[]{};:,.<.>?'
907     mp = test_sans_cor(m,1000,500)
908     yy_values=[mp for i in range(len(x_values))]
909     for i in range(len(x_values)):
910         y_values.append(test_hamming(m,1000,x_values[i],500,0))
911         print(i)
912     plt.scatter(x_values,y_values) # points bleu
913     plt.xlabel(" valeur de n")
914     plt.scatter(x_values,yy_values) #points orange
915     plt.ylabel("Nombre de messages correctement retrouvés")
916     plt.grid()
917     plt.show()
918
919 #hamming_pour_different_n()
920
921 # test niveau taille avec n=4
922
923 # test entrelacement Hamming vs RS
924
925 def test_hamming_entrelace(m:str,int:int,taille:int,proba:int):
926     """fonction de test du codage de hamming dans un canal bruité
927     on test int fois la fonction et on compte le nombre de fois où le message est correctement retranscrit
928
929     Args:
930         m (str): message
931         int (int): nombre de fois que l'on test le procedé
932         taille (int): sur combien de bits utilisé hamming
933         proba (int): 1/proba = probabilité qu'un bits soit altéré (voir dans la fonction si uniforme ou pas)
934         perte(int): = 0 corruption de donnée = 1 perte de donnée
935
936     Returns:
937         int: nombre de transmission correcte
938     """
939     c=0
940     corrompu= ''.join(['0' for i in range(2**taille)]) # si il y avait un 0 pas d'erreur et sinon il la detetera
941     for i in range(int):
942         l=hamming_encoding(m,taille)
943         n=len(l)
```

Annexe 3 code

```
928 def test_hamming_entrelace(m:str,int:int,taille:int,proba:int):
946     n=len(l)
947     l = entrelacement(l,2**taille)
948     for i in range(n):
949         if random()<1/proba: # dans le cas d'une perte de donn  
950             l[i]= corrompu
951     l = entrelacement(l,2**taille)
952     mm=hamming_decoding(l)
953     if mm==m:
954         c=c+1
955     return c
956
957
958 #Hamming_vs_Reed_Solomon()
959 # test valeur de RS
960
961
962 def Reed_Solomon_different_k():
963     message='This is a simple ASCII text '
964     m='This is a simple ASCII text '
965     n=len(codage_binaire(m))//15 + 1
966     ll=encodage_reed_solomon(message,3,n)
967     print(len(ll))
968     print('test')
969     x_values=[]
970     y_values=[]
971     nb=1000
972     m='This is a simple ASCII text '
973     for i in range(0,4):
974         print(i)
975         x_values.append(i)
976         y_values.append(test_RS(message,nb,50,i))
977     print(sum(y_values)/len(y_values))
978     plt.scatter(x_values,y_values)
979     plt.xlabel("valeur de k")
980     plt.ylabel("Nombre de messages correctement retrouv  s")
981     plt.grid()
982     plt.show()
983
984 #Reed_Solomon_different_k()
985
986
987 def Hamming_vs_Reed_Solomon():
988     message_h='This is a simple ASCII text '
989     message_RS='This is a simple ASCII text '
990     n=len(codage_binaire(message_RS))//15 + 1
991     l=hamming_encodage(message_h,n)
```


Annexe 3 code

```
987 def Hamming_vs_Reed_Solomon():
991     l=hamming_encoding(message_h,6)
992     ll=encodage_reed_solomon(message_RS,3,n)
993     print(len(l))
994     print(len(ll))
995     nb=10000
996     redondance_RS=6
997     taille_hamming=6
998
999     nh=test_hamming_entrelace(message_h,nb,taille_hamming,20)
1000     nrs=test_RS(message_RS,nb,20,redondance_RS)
1001     categories = ['Hamming' , 'Reed-Solomon']
1002     values= [nh,nrs]
1003     colors=['blue','red']
1004     print(values)
1005     plt.bar(categories,values,color=colors)
1006     plt.ylim(0,nb)
1007     plt.xlabel("")
1008     plt.ylabel("Nombre de messages correctement retrouvés")
1009     plt.grid()
1010     plt.show()
1011
1012
1013 Hamming_vs_Reed_Solomon()
1014
1015 def Hamming_vs_Reed_Solomon_fini():
1016     message_h='This is a simple ASCII text '
1017     message_RS='This is a simple ASCII text '
1018     l=hamming_encoding(message_h,3)
1019     ll=encodage_reed_solomon_fini(message_RS,4)
1020     print(l)
1021     print(len(l))
1022     print(ll)
1023     print(len(ll))
1024     nb=1000
1025     redondance_RS=5
1026     taille_hamming=3
1027
1028     nh=test_hamming_entrelace(message_h,nb,taille_hamming,35.555)
1029     nrs=test_RS_fini(message_RS,nb,35.555,redondance_RS)
1030     categories = ['Hamming' , 'Reed-Solomon']
1031     values= [nh,nrs]
1032     colors=['blue','red']
1033     print(values)
1034     plt.bar(categories,values,color=colors)
1035     plt.ylim(0,nb)
1036     plt.xlabel("")
1037
1038     values= [nh,nrs]
1039     colors=['blue','red']
1040     print(values)
1041     plt.bar(categories,values,color=colors)
1042     plt.ylim(0,nb)
1043     plt.xlabel("")
1044     plt.ylabel("Nombre de messages correctement retrouvés")
1045     plt.grid()
1046     plt.show()
1047
1048 #Hamming_vs_Reed_Solomon_fini()
```