Codes correcteurs d'erreurs

TIPE

Mathéo THOMAS

Codes correcteurs d'erreurs

TIPE

Mathéo THOMAS

Comment assurer l'intégrité de données lors de leur transmission en étudiant et simulant leur émission et réception, à travers l'application de codes correcteurs d'erreur.

Codes correcteurs d'erreurs

TIPE

Mathéo THOMAS

Comment assurer l'intégrité de données lors de leur transmission en étudiant et simulant leur émission et réception, à travers l'application de codes correcteurs d'erreur.

I. Principe des codes correcteurs d'erreurs

Distance de Hamming.

Distance de Hamming

Soient $m = m_1 \dots m_n,\ m' = m'_1 \dots m'_n \in \mathbb{F}_2^n$,

$$\delta(m, m') = \sum_{i=1}^{n} (m_i \oplus m'_i)$$

Distance de Hamming.

Distance de Hamming

Soient $m = m_1 \dots m_n,\ m' = m'_1 \dots m'_n \in \mathbb{F}_2^n$,

$$\delta(m, m') = \sum_{i=1}^{n} (m_i \oplus m'_i)$$

Boule de Hamming

Soient m un mot et t représentant le nombre d'erreurs,

$$\mathcal{B}_H(m, t) = \{m' \in \mathbb{F}_2^n \mid \delta(m, m') \leq t\}$$

Codes.

Code binaire

Un code binaire dont un mot de longueur k est codé en un mot de longueur n est une partie C de $\mathbb{F}_2^n$, noté $C(n, k)$.

Principe des codes correcteurs d'erreurs

Codes.

Code binaire

Un code binaire dont un mot de longueur k est codé en un mot de longueur n est une partie C de $\mathbb{F}_2^n$, noté $C(n, k)$.

Code linéaire

C est linéaire si et seulement si $\forall m, m' \in C,\ m \oplus m' \in C$.

Principe des codes correcteurs d'erreurs

Codes.

Code binaire

Un code binaire dont un mot de longueur k est codé en un mot de longueur n est une partie C de $\mathbb{F}_2^n$, noté $C(n, k)$.

Code linéaire

C est linéaire si et seulement si $\forall m, m' \in C,\ m \oplus m' \in C$.

Code parfait

$C(n, k, t)$ est parfait si et seulement si les boules de Hamming de rayon t forment une partition de l'ensemble des codes.

Encodage et décodage.

Matrice génératrice

Encodage : φ injective telle que $\boxed{C = \mathrm{Im}(\varphi)}$. Représentable matriciellement par $\varphi(m) = mG$, où G de la forme $G = (I_k|B)$, avec B matrice à k lignes et $n - k$ colonnes.

Encodage et décodage.

Matrice génératrice

Encodage : φ injective telle que $\boxed{C = \mathrm{Im}(\varphi)}$. Représentable matriciellement par $\varphi(m) = mG$, où G de la forme $G = (I_k|B)$, avec B matrice à k lignes et $n - k$ colonnes.

Matrice de contrôle

On pose $H = (-B^T|I_{n-k})$ si $G = (I_K|B)$.
Alors $\forall m \in C,\ Hm^T = 0$ et $HG^T = 0$.
H est la matrice de contrôle de C si $\boxed{C = \mathrm{Ker}(H)}$.

Principe des codes correcteurs d'erreurs

Encodage et décodage.

Matrice génératrice

Encodage : φ injective telle que $\boxed{C = \mathrm{Im}(\varphi)}$. Représentable matriciellement par $\varphi(m) = mG$, où G de la forme $G = (I_k|B)$, avec B matrice à k lignes et $n - k$ colonnes.

Matrice de contrôle

On pose $H = (-B^T|I_{n-k})$ si $G = (I_K|B)$.
Alors $\forall m \in C,\ Hm^T = 0$ et $HG^T = 0$.
H est la matrice de contrôle de C si $\boxed{C = \mathrm{Ker}(H)}$.

Syndrome

Soit $m \in \mathbb{F}_2^n$, $s = Hm^T$ est le syndrome de m.

Exemple simple

$G = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}, H = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

$m = (10)$. Le mot codé $w = mG = (10)\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} = (1010)$.

Erreur en deuxième position : $w' = (1110)$.

Syndrome de w' : $s = Hw'^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

s deuxième colonne de H, erreur en **deuxième position**.

II. Codage de Hamming

Distance minimale

La distance minimale δ_C d'un code C est

$$\delta_C = \min\{\delta(m, m') \mid (m, m' \in C^2, m \neq m')\}$$

Distance minimale

La distance minimale δ_C d'un code C est

$$\delta_C = \min\{\delta(m, m') \mid (m, m' \in C^2, m \neq m')\}$$

Code de Hamming

Soit $r = n - k$.
Un code de Hamming est un code de paramètres $(2^r - 1, 2^r - r - 1, \delta_C)$, linéaire et parfait.

Codage de Hamming

Soit $r = 4$, étudions $C(15, 11, 4)$.
$m = b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11} \in \mathbb{F}_2^{11}$,
$\varphi(m) = b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11}p_1p_2p_3p_4$,

$$G = (I_{11}|B) = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Codage de Hamming

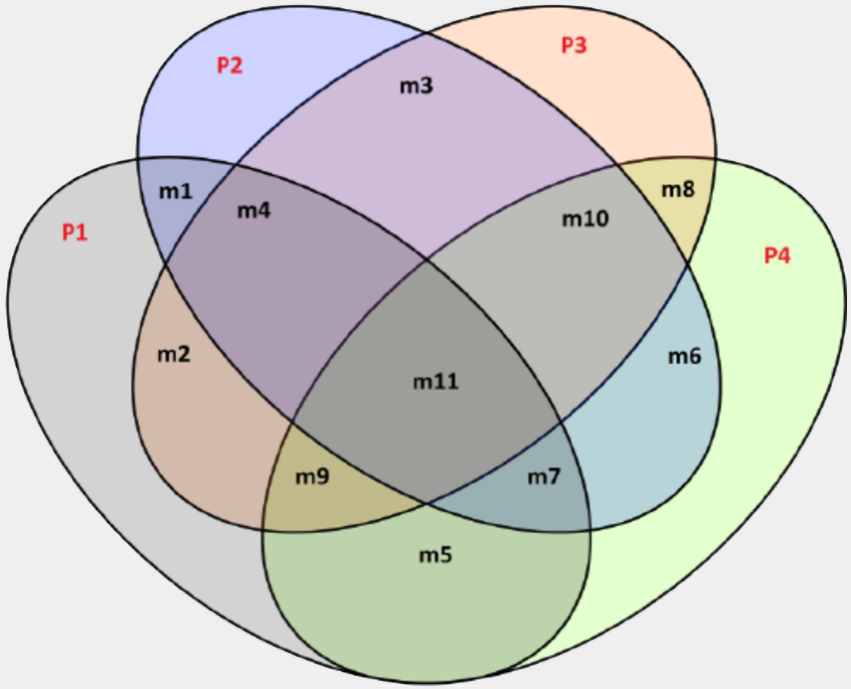
$$H = (-B^T | I_4) = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Prenons $m = (10010101111)$.
Alors $w = mG = (10010101111|0111)$.
Modifions le septième bit : $w' = (10010111111110111)$.

Alors $s = Hw'^T = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}$: septième colonne de H.

Codage de Hamming

$$\begin{cases} p_1 = b_1 \oplus b_2 \oplus b_4 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \\ p_2 = b_1 \oplus b_3 \oplus b_4 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \\ p_3 = b_2 \oplus b_3 \oplus b_4 \oplus b_8 \oplus b_9 \oplus b_{10} \oplus b_{11} \\ p_4 = b_5 \oplus b_6 \oplus b_7 \oplus b_8 \oplus b_9 \oplus b_{10} \oplus b_{11} \end{cases}$$



III. Simulation

Simulation

Étapes :

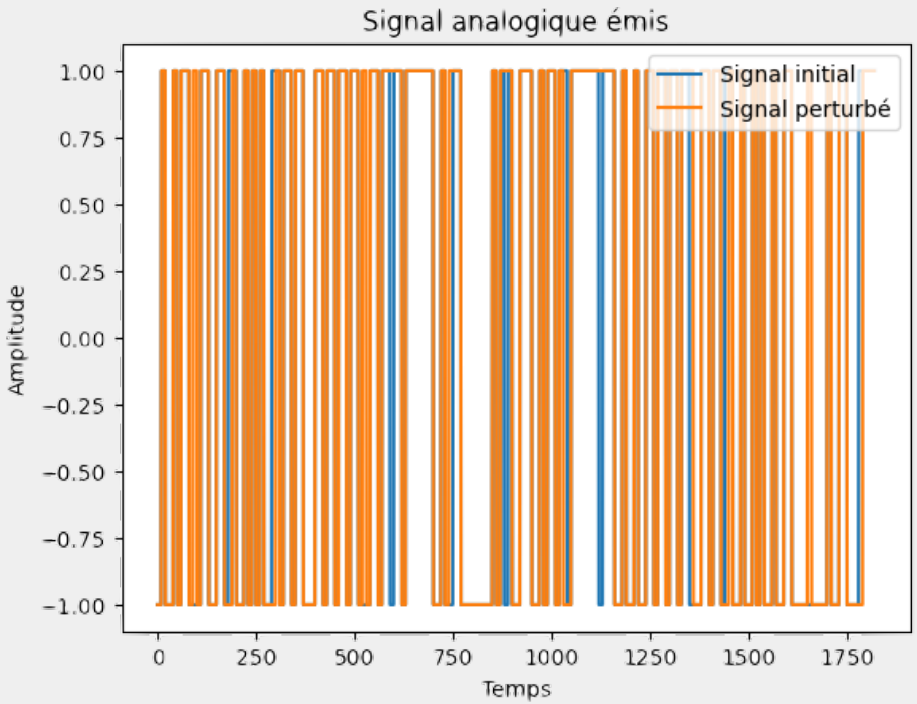
1. Convertir le message alphanumérique en message binaire.
2. Coder le message selon le code de Hamming.
3. Convertir le message en un signal "analogique".
4. Perturber le signal : inversion de bits + bruit.
5. Convertir le signal en message binaire.
6. Corriger le message.

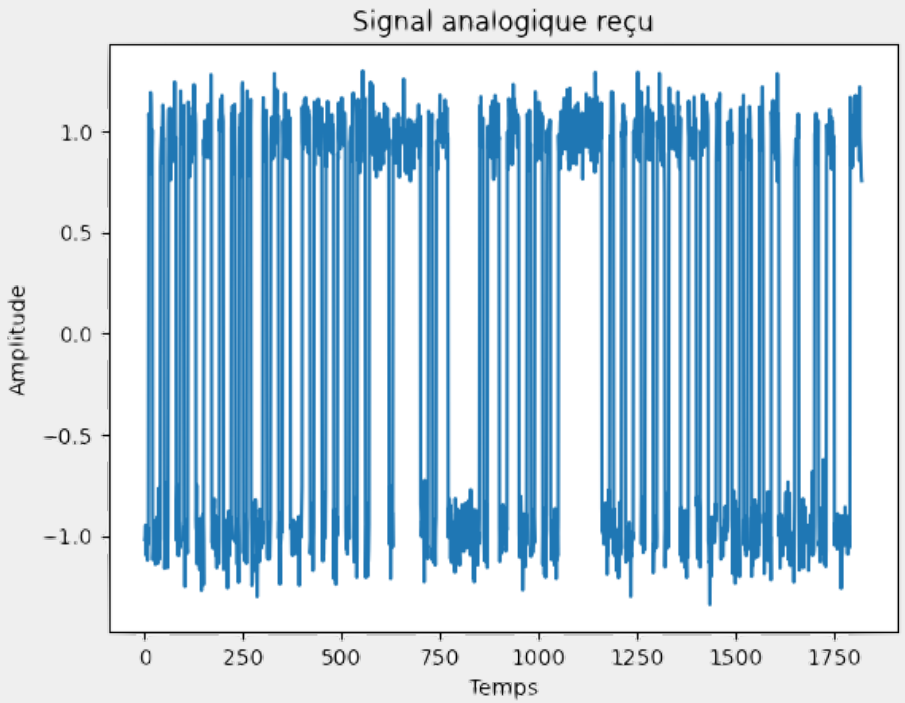
Simulation

Étape	Résultat
Message initial	Hello, World !
Conversion binaire	0100|1000...0001
Codage de Hamming	01001**01**|10001**10**...10001**11**

Simulation

Étape	Résultat
Message initial	Hello, World !
Conversion binaire	0100|1000...0001
Codage de Hamming	0100101|1000110...10001111



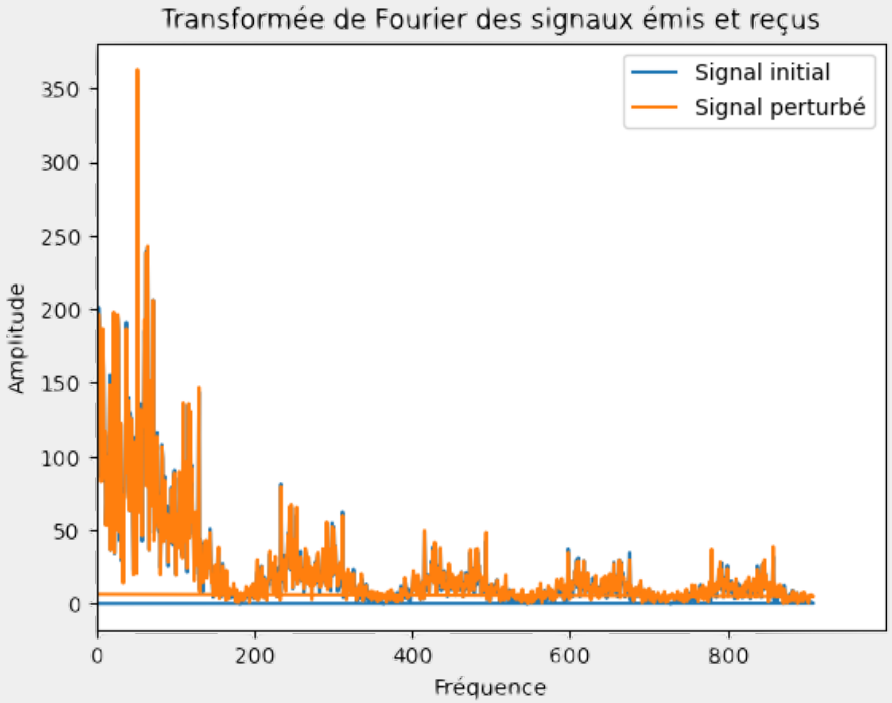


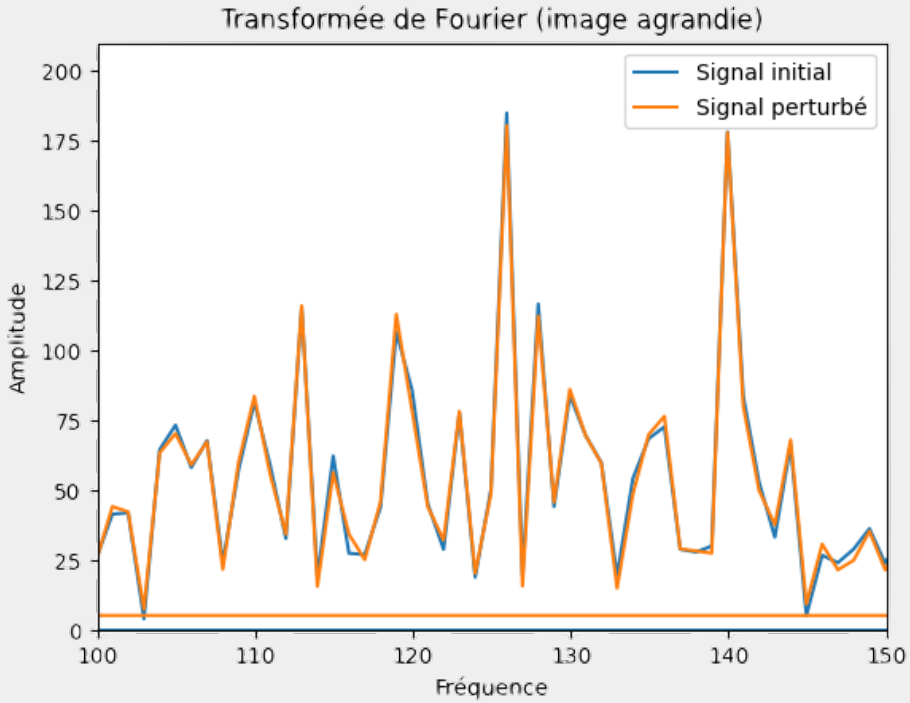
Simulation

Étape	Résultat
Message perturbé binaire	01101011000110...10011011
Message corrigé	01001011000110...10001111
Conversion alphanumérique	Hello, World !

Simulation

Étape	Résultat
Message perturbé binaire	01101011000110...10011011
Message corrigé	01001011000110...10001111
Conversion alphanumérique	Hello, World !





Fin

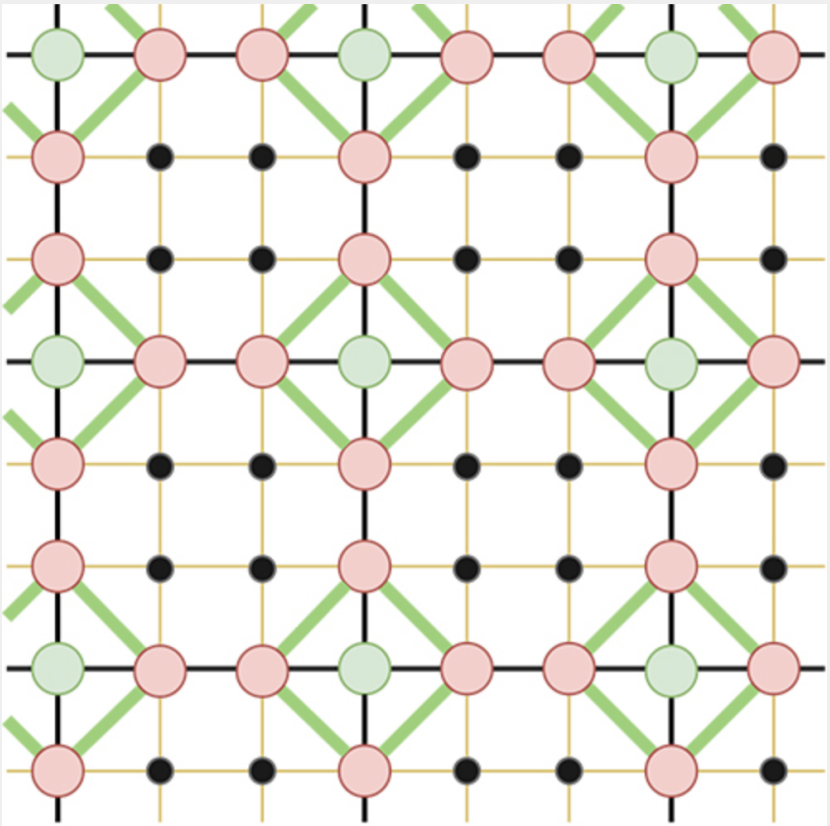
Références

Marc Chaumont.
Codes correcteurs d'erreurs.
https://www.lirmm.fr/~chaumont/download/cours/codescorrecteur/01_codes_correcteurs_d%27erreurs_1_transparent_par_page.pdf.

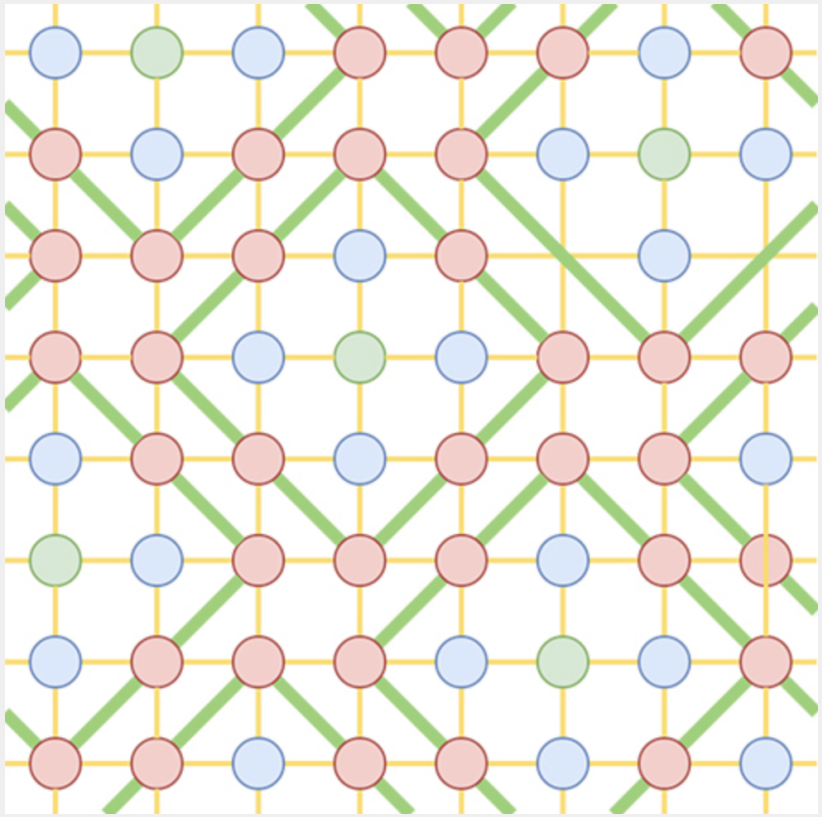
Étienne Duris.
Signalisation, codage, contrôle d'erreurs.
http://igm.univ-mlv.fr/~duris/RESEAU/L3/L3-phyCodage-20092010.pdf.

Jean Dupuy Emmy Duclos.
Codes correcteurs : Garder les erreurs À distance.
https://tangente-mag.com/article.php?id=6715.

Annexe



Code (source : Tangente)



Code parfait (source : Tangente)

Annexe

Affichage du programme

```
Message initial :  Hello, World !
Message converti en binaire :  01001000011001010110110001101100011011110010
Message encodé :  01001011000110011011001010100110110110001101101101100011
Message perturbé numérique :  01011011000110011011001000100110111110001101
Message corrigé 01001000011001010110110001101100011011110010110000010000001
Message final :  Hello, World !
```

Annexe

simulation.py

```
'''
15643 - THOMAS Mathéo
TIPE - Code correcteur d'erreur

Simulation de la transmission d'un signal subissant une perturbation
'''

############
# IMPORTS #
############
import numpy as np
import matplotlib.pyplot as plt
import random
import scipy.fft as sfft

#######
# DAC #
#######
treshold = 0.5

def alphaToBinary(msg):     # convertit un message alphanumérique en binaire (code ASCII)
    n = len(msg)
    res = ''
    for i in range(n):
        res += (f'{ord(msg[i]):08b}')
    return res

def binaryToAnalog(msg):    # crée un signal temporel à partir d'un message binaire (1 -> 1 in binary, -1 -> 0 in binary)
    N = 10*len(msg)
    n = len(msg)
    tab = np.linspace(0,0, N)
    cut = 0
    for i in range(n):
        #if i % 8 == 0:
        #    cut += 10
```

Annexe

```
36          for j in range(10):
37              if msg[i] == "1":
38                  tab[10*i+j+cut] = 1
39              else:
40                  tab[10*i+j+cut] = -1
41      return tab
42
43  def alphaToAnalog(m):       # conversion alphanumérique - 'analogique'
44      return binaryToAnalog(alphaToBinary(m))
45
46
47  def binaryToAlpha(msg):     # convertit un message binaire en alphanumérique (code ASCII)
48      res2 = ""
49      n = len(msg)
50      for i in range(n//8):
51          res1 = 0
52          n = 8
53          for j in range(8):
54              res1 += 2**(n-1)*int(msg[j+8*i])
55              n -= 1
56          res2 += chr(res1)
57      return res2
58
59  def analogToBinary(tab):    # convertit un signal temporel en binaire (1 -> 1 en binaire, -1 -> 0 en binaire)
60      N = len(tab)
61      i = 1
62      res = ""
63      while i < N:
64          if tab[i] >= treshold:
65              res += "1"
66          else:
67              res += "0"
68          i += 10
69      return res
70
71  def analogToAlpha(tab):     #conversion 'analogique' - alphanumérique
72      return binaryToAlpha(analogToBinary(tab))
73
74
```

Annexe

```
###########
# HAMMING #
###########
def add(x, y):              # XOR
    if x == y:
        return '0'
    else:
        return '1'

def encode(msg):            # calcule les bits de parité pour encoder msg
    msg += add(add(msg[0],msg[1]), msg[3])
    msg += add(add(msg[0],msg[2]), msg[3])
    msg += add(add(msg[1],msg[2]), msg[3])
    return msg

def encoder(msg):
    res = ''
    chunks, chunk_size = len(msg), 4
    m = [ msg[i:i+chunk_size] for i in range(0, chunks, chunk_size) ]
    for i in range(len(msg)//4):
        res += encode(m[i])
    return res

def opp(msg, i):            # inverse le bit de msg en position i
    l = list(msg)
    if l[i] == '1':
        l[i] = '0'
    else:
        l[i] = '1'
    return "".join(l)

def decode(msg):            # calcule et compare les bits de parité, et modifie en conséquence msg pour le corriger
    w4 = add(add(msg[0],msg[1]), msg[3])
    w5 = add(add(msg[0],msg[2]), msg[3])
    w6 = add(add(msg[1],msg[2]), msg[3])

    if w4 != msg[4] and w5 != msg[5] and w6 != msg[6]:
        # print("msg[3] false")
        return opp(msg, 3)
```

Annexe

```
114     elif w4 != msg[4] and w5 != msg[5]:
115         # print("msg[0] false")
116         return opp(msg, 0)
117     elif w4 != msg[4] and w6 != msg[6]:
118         # print("msg[1] false")
119         return opp(msg, 1)
120     elif w5 != msg[5] and w6 != msg[6]:
121         # print("msg[2] false")
122         return opp(msg, 2)
123     elif w4 != msg[4]:
124         # print("msg[4] false")
125         return opp(msg, 4)
126     elif w5 != msg[5]:
127         # print("msg[5] false")
128         return opp(msg, 5)
129     elif w6 != msg[6]:
130         # print("msg[6] false")
131         return opp(msg, 6)
132     # print("msg correct")
133     return msg
134 
135 def decoder(msg):
136     res = ''
137     chunks, chunk_size = len(msg), 7
138     m = [ msg[i:i+chunk_size] for i in range(0, chunks, chunk_size) ]
139     for i in range(len(msg)//7):
140         res += (decode(m[i]))[0:4]
141     return res
142 
143 def perturb(msg):           # inverse un bit de msg au hasard
144     r = random.randint(0, len(msg))
145     print ("bit modifié :", r)
146     return opp(msg, r)
147 
148 def perturbAnalog(tab):
149     r = (random.randint(0, len(tab)))//10
150     print("r : ", r)
151     for i in range(10):
152         if (tab[r*10 + i] == 1):
```

Annexe

```
153             tab[(r*10)+i] = -1
154         else :
155             tab[(r*10)+i] = 1
156     return tab
157 
158 def perturbAnalogMulti(tab):
159     print("perturbations : ", len(tab)//200)
160     for i in range(len(tab)//200):
161         tab = perturbAnalog(tab)
162     return tab
163 
164 def perturbAnalogMultiEvenly(tab):
165     for j in range((len(tab))//140):
166         r = random.randint(j*14, (j+1)*14)
167         for i in range(10):
168             if (tab[r*10 + i] == 1):
169                 tab[(r*10)+i] = -1
170             else :
171                 tab[(r*10)+i] = 1
172     return tab
173 
174 def fft(t, s):              # renvoie la décomposition en série de fourier du signal s où f est un tableau de fréquences
    et a est un tableau d'amplitudes complexes décrivant le signal # Paramètres : t : tab[float], points - s : tab[float], le
    signal s evalué en tous les points de t
175     N, fe = len(t), 1/(t[1]-t[0])
176     a = sfft.fft(s)
177     f = sfft.fftfreq(N, 1/fe)
178     return f, a
179 
180 
181 #############
182 # EXECUTION #
183 #############
184 # 1. Message initial
185 m = "Hello, World !"
186 print("Message initial : ", m)
187 atb = alphaToBinary(m)
188 print("Message converti en binaire : ", atb)
189 
190 # 2. Codage du message avec le code de Hamming
```

Annexe

```
191 encodemsg = encoder(atb)
192 print("Message encodé : ", encodemsg)
193
194 # 3. Conversion numérique-analogique
195 analogmsg = binaryToAnalog(encodemsg)
196
197 plt.figure(1)
198 plt.title("Signal analogique émis")
199 plt.xlabel("Temps")
200 plt.ylabel("Amplitude")
201
202 plt.plot(analogmsg)
203
204 # 4. Perturbation
205 # analogmsg = perturbAnalog(analogmsg)
206 # analogmsg = perturbAnalogMulti(analogmsg)
207 analogmsg = perturbAnalogMultiEvenly(analogmsg)     # Ajout d'erreurs sur les bits (inversion)
208 plt.plot(analogmsg)
209 plt.legend(["Signal initial", "Signal perturbé"], loc = "upper right")
210
211 analogmsgNoise = analogmsg + np.random.normal(0, .1, len(analogmsg))    # Ajout de bruit sur tout le spectre
212
213 plt.figure(2)
214 plt.title("Signal analogique reçu")
215 plt.xlabel("Temps")
216 plt.ylabel("Amplitude")
217 plt.plot(analogmsgNoise)
218
219 x = np.linspace(0, 1, len(analogmsg))
220
221 plt.figure(3)
222 plt.title("Transformée de Fourier (image agrandie)")
223 plt.xlabel("Fréquence")
224 plt.ylabel("Amplitude")
225 f1, an = fft(x, analogmsgNoise)
226 f2, a = fft(x, analogmsg)
227 plt.plot(f2, np.abs(a), f1, np.abs(an))
228 # plt.xlim(left=0)
229 plt.xlim(left=100, right =150)
```

Annexe

```
230 plt.ylim(0, 210)
231 plt.legend(["Signal initial", "Signal perturbé"])
232
233 # 5. Conversion analogique-numérique
234 decodemsg = analogToBinary(analogmsgNoise)
235 print("Message perturbé numérique : ", decodemsg)
236
237 # 6. Correction du message
238 msg = decoder(decodemsg)
239 print("Message corrigé", msg)
240
241 # 7. Message final
242 mf = binaryToAlpha(msg)
243 print("Message final : ", mf)
```