Fiabilité de l'information : utilisation des codes correcteurs d'erreur, Reed-Solomon

Tom Effernelli 23017

Problème :

Comment être sûr que le message reçu est bien celui envoyé ?

Causes d'erreurs :

- Attaque par un tiers
- Défaillances matérielles
- Erreurs de transmissions
- …

01 Code correcteur d’erreur

Notions et définitions

Code linéaire

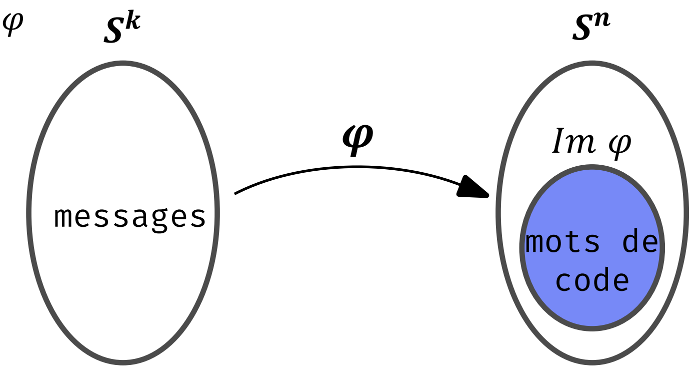
But : rajouter information (codage) pour corriger erreurs (décodage)

Symboles : $S = \{s_1, \dots, s_t\}$

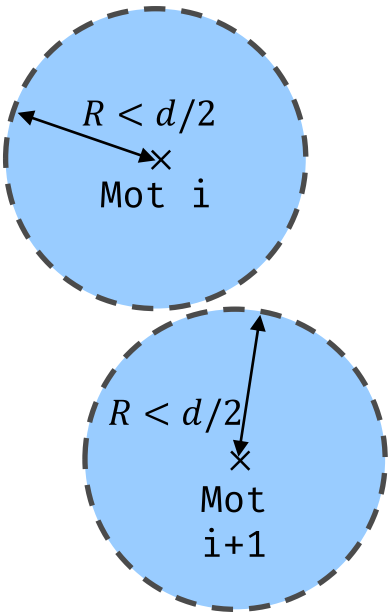
Application de codage : φ linéaire injective de S^k dans S^n

Code : espace vectoriel $Im\ \varphi$

Mots de code : éléments de $Im\ \varphi$



Détection, correction



Poids d'un mot : $\omega(c) = Card\{c_i \neq 0, i \in [\![1;n]\!]\}$

Distance minimale de correction : $d = \min\limits_{\substack{c,c' \in Im\,\varphi \\ c \neq c'}} \omega(c - c')$

Détecter erreur e, $\omega(e) < d$: si $m' = m + e$ alors $m' \notin Im\,\varphi$, sinon $e = m' - m \in Im\,\varphi$ exclus car $\omega(e) < d$.

Corriger erreur e, $\omega(e) < {}^{d}/_{2}$: si $m' = m_1 + e_1 = m_2 + e_2$ avec $\omega(e_{1,2}) < {}^{d}/_{2}$, alors $\omega(m_2 - m_1) = \omega(e_2 - e_1) < \omega(e_1) + \omega(e_2) < d$ donc $m_2 = m_1$.

404 NOT FOUND

02

Codes de Reed-Solomon

Corps finis, arithmétique modulaire, polynômes…

Représentation des mots

Représentation des mots par des polynômes :

$$m = (m_0, \dots, m_{i-1}) \in S^i \Leftrightarrow M(X) = \sum_{l=0}^{i-1} m_l X^l \in S[X]$$

Polynôme générateur

$\mathbb{K}$ corps fini à 2^i éléments, $S = \mathbb{K}$

α racine primitive ($\mathbb{K}^* = <\alpha>$)

d entier > 1

On pose :

$$G(X) = \prod_{i=1}^{d-1} (X - \alpha^i)$$

polynôme générateur du code de Reed-Solomon de distance minimale de correction d.

Codage

- **Paramètres :** message $A(X) = \sum_{i=0}^{k-1} a_i X^i$
- **But :** renvoyer mot de code $C(X)$

2. Division euclidienne : $X^{n-k}A(X) = G(X)U(X) + R(X)$
3. Renvoyer : $C(X) = X^{n-k}A(X) - R(X)$

Décodage

- **Paramètres :** message codé reçu $C' = C + E$, C mot de code, $E = \sum_{i=0}^{f} e_{r_i} X^{r_i}$ erreur
- **Hypothèses :** $\omega(E) < {}^{d}/_{2}$ (E corrigeable) $\Leftrightarrow f \leq \left\lfloor \frac{d-1}{2} \right\rfloor$
- **But :** déterminer E

1. $s_j = C'(\alpha^j) = C(\alpha^j) + \mathrm{E}(\alpha^j) = E(\alpha^j)$ pour $1 \leq j \leq d-1$
2. Si $\forall j\ s_j = 0$: renvoyer C'

   Sinon : a. $d-1$ équations (s_j) pour au maximum $2 \times \left\lfloor \frac{d-1}{2} \right\rfloor \leq d-1$ inconnues (position et valeur des coefficients de E) mais pas linéaire !

   b. équation clé

03

Réalisation pratique d’un code de Reed-Solomon

Programmation en Python

Pseudo-code

module *galois* (opérations élémentaires $+/\times/\log_\alpha$ sur le corps fini)

- **initialisation des paramètres**
- **fonctions relatives aux polynômes :** addition, multiplication, division euclidienne…
- **fonctions relatives au corps fini :** plus petite racine primitive, construction du polynôme générateur
- **codage**
- **décodage**
- **canal**

-> Annexe code n°1

Division euclidienne

```
▸ division_euclidienne(M, N):
    R <- M
    Q <- 0
    tant que deg(R) ≥ deg(N):
        Q <- Q + cd(R)/cd(N) X^(deg(R)−deg(N))
        R <- R − cd(R)/cd(N) X^(deg(R)−deg(N)) × N
    retourner Q, R
```

Recherche de racine primitive

- **recherche_racine_primitive**($corps\_fini$):

  pour α **dans** $corps\_fini$:

  i <- 1

  tant que $\alpha^i \neq 1$:

  i <- $\mathrm{i} + 1$

  si $i = Card(corps\_fini) - 1$:

  retourner α

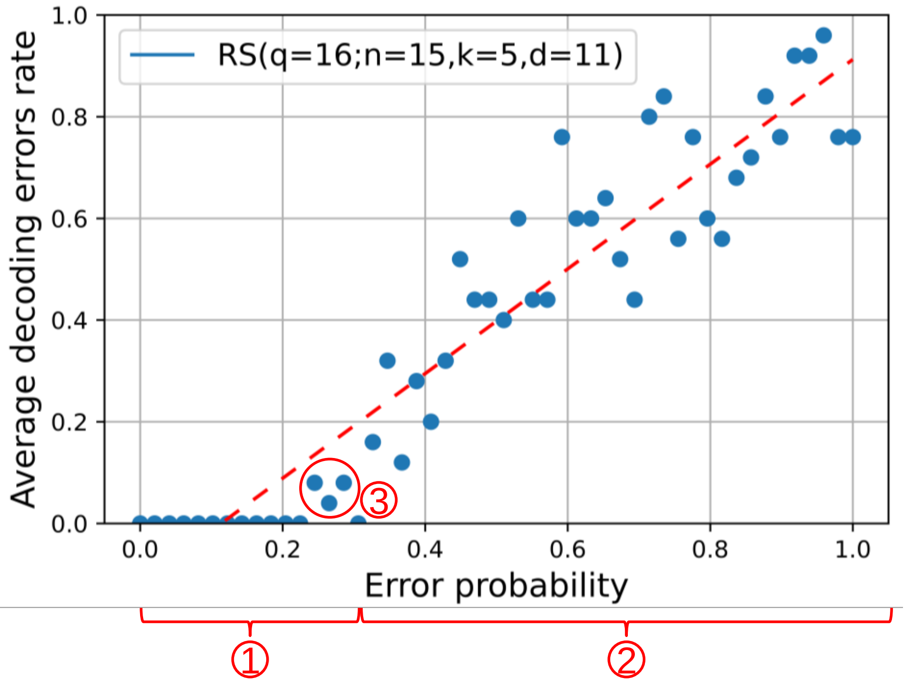
Canal

```
▶ canal(message, proba_seuil):
    pour symbole dans message:
        p <- probabilité_uniforme([0, 1])
        si p < proba_seuil:
            modifier symbole
    retourner message_modifié
```

04 Analyse du programme

Résultats et performances

Analyse générale

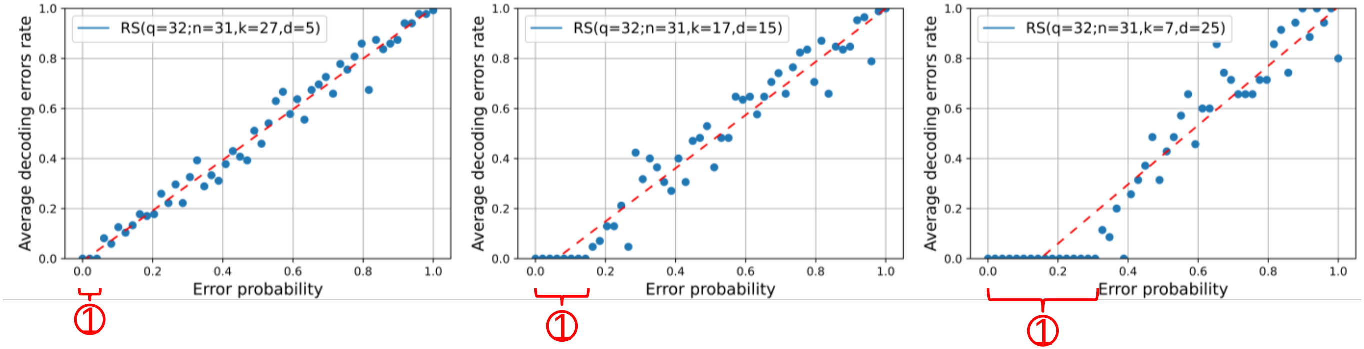


Zone 1 : erreurs corrigées, $p_{max} \times n = 0.32 \times 15 = 4{,}8 = \frac{d-1}{2}$ OK

Zone 2 : zone d'erreur

Zone 3 : erreurs dans la zone de correction. Le code considère (à tort) qu'un mot de code erroné ne l'est pas.

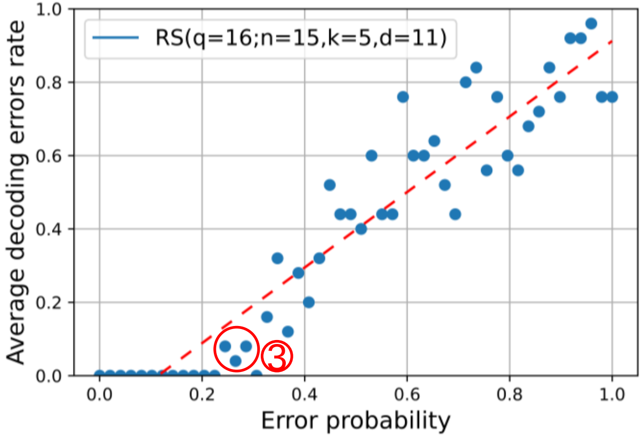
En augmentant d



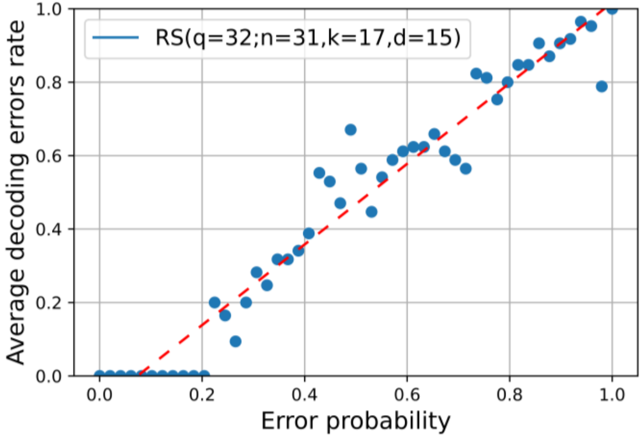
$d = 5$ $d = 15$ $d = 25$

Le nombre d'erreurs corrigées augmente avec d

En augmentant q



$q = 16$



$q = 32$

Disparition de la zone 3

Comparaison

Original

Hamming

Reed-Solomon

BCH

-> Annexe code n°2

05

Annexes

Equation clé, code, figures…

Equation clé

- **Paramètres :** message codé reçu $C' = C + E$, C mot de code, $E = \sum_{i=0}^{f} e_{r_i} X^{r_i}$ erreur
- **Inconnues :**
  1. nombre d'erreurs (f)
  2. position des erreurs (r_i)
  3. valeurs des erreurs (e_{r_i})

On a calculé : $s_j = C'\left(\alpha^j\right) = C\left(\alpha^j\right) + \mathrm{E}\left(\alpha^j\right) = E\left(\alpha^j\right) = \sum_{i=0}^{f} e_{r_i} \alpha^{j r_i}$ pour $1 \leq j \leq d-1$

On pose : $S(X) = \sum_{i=1}^{d-1} s_j X^{j-1}$ et $\sigma(X) = \prod_{i=1}^{f} (1 - \alpha^{r_i} X)$

Equation clé

***Proposition** : Il existe* $\omega(X) \in \mathbb{K}[X]$ *tel que* $S(X)\sigma(X) \equiv \omega(X)\ [X^{n-k}]$

***Lemme :** si il existe* $\sigma'(X)$ *et* $\omega'(X) \in \mathbb{K}[X]$ *tels que* $S(X)\sigma'(X) \equiv \omega'(X)\ [X^{n-k}]$ *alors il existe* $C(X) \in \mathbb{K}[X]$ *tel que* $\sigma'(X) = C(X)\sigma(X)$ *et* $\omega'(X) = C(X)\omega(X)$*.*

On construit, par algorithme d'Euclide étendu, des suites $(A_n)_n, (B_n)_n, (P_n)_n, (Q_n)_n \in \mathbb{K}[X]$ telles que :

$$\begin{cases} P_{n+1} = P_{n-1} - P_n Q_n \ \ avec \ \ P_0 = X^{n-k} \ \ et \ \ P_1 = S \\ A_{n+1} = A_{n-1} - A_n Q_n \ \ avec \ \ A_0 = 1 \ \ et \ \ A_1 = 0 \\ B_{n+1} = B_{n-1} - B_n Q_n \ \ avec \ \ B_0 = 0 \ \ et \ \ B_1 = 1 \end{cases}$$

Equation clé

***Proposition** : Soit i l'unique indice tel que $deg(P_{i-1}) \geq \frac{d-1}{2}$ et $deg(P_i) < \frac{d-1}{2}$. Alors on a $B_i(0) \neq 0$ et $\sigma(X) = \frac{B_i(X)}{B_i(0)}$.*

Cela nous permet d'accéder à $\sigma(X)$ puis à ses racines, les α^{-r_i}.

On en déduit les r_i.

On a déterminé f et les r_i.

Equation clé

Il s'agit enfin de déterminer la valeur des erreurs e_{r_i}.

On reprend notre système de $d-1$ équations à $d-1$ inconnues, comme on connait les α^{r_i} ce système est désormais linéaire !

On a alors :

$$\begin{cases} s_1 = e_{r_1}\alpha^{r_1} + e_{r_2}\alpha^{r_2} + \cdots + e_{r_f}\alpha^{r_f} \\ s_2 = e_{r_1}(\alpha^{r_1})^2 + e_{r_2}(\alpha^{r_2})^2 + \cdots + e_{r_f}(\alpha^{r_f})^2 \\ \vdots \\ s_f = e_{r_1}(\alpha^{r_1})^f + e_{r_2}(\alpha^{r_2})^f + \cdots + e_{r_f}(\alpha^{r_f})^f \end{cases} \Leftrightarrow \begin{pmatrix} e_{r_1} \\ e_{r_2} \\ \vdots \\ e_{r_f} \end{pmatrix} = \begin{pmatrix} \alpha^{r_1} & \alpha^{r_2} & \cdots & \alpha^{r_f} \\ (\alpha^{r_1})^2 & (\alpha^{r_2})^2 & & (\alpha^{r_f})^2 \\ \vdots & \vdots & & \vdots \\ (\alpha^{r_1})^f & (\alpha^{r_2})^f & & (\alpha^{r_f})^f \end{pmatrix}^{-1} \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_f \end{pmatrix}$$

On a déterminé toutes les inconnues, on connait finalement E !

```
import galois
import numpy as np
import random
import matplotlib.pyplot as plt
from scipy import stats

#defines RS code (q;n,k,d)
s=4
p=2 #p must be a prime number
q=p**s
n=q-1
d=7 #d must be odd and <=q-1
k=q-d

#basic verifications
if not(1<d<q): raise Exception("d must be strictly between 1 and {}!".format(q))
if (n-k)%2 != 0: raise Exception("n-k should be even!")
if s<=0 or p<=1 or d<=1 or k<=0: raise Exception("all parameters should be > 0!")

#defining our finite fields
F1 = galois.GF(p,s)

#polynomials related operations in GF
def get(F, P, i):
   try:
       return P[i]
   except IndexError:
       return F(0)

def poly_add(F,P,Q):
    R=[]
    u=max(len(P),len(Q))
    for i in range(u):
        R.append(int(get(F,P,i)+get(F,Q,i)))
    return F(R)

def poly_mult(F, P,Q):
    R=[]
    u=len(P)+len(Q)-1
    for i in range(u):
```

```
        coef=F(0)
        for k in range(i+1):
            coef+=get(F,P,k)*get(F,Q,i-k)
        R.append(int(coef))
    return F(R)

def evaluate(F,P,a):
    result=F(0)
    for i in range(len(P)):
        result+=P[i]*(a**i)
    return result

def is_null(F,P):
    null=True
    for i in range(len(P)):
        if P[i] != F(0): null=False
    return null

def weight(F,P):
    w=0
    for i in range(len(P)):
        if P[i] != F(0): w+=1
    return w

def clean(F,P):
    index=-1
    while len(P)>=1 and (P[index]==F(0)):
        P=np.delete(P,index)
    return P

def deepcopy(F,L):
    C=[]
    for i in range(len(L)):
        C.append(int(L[i]))
    return F(C)

def DE(F,M,N):
    Q=F([0])
    R=deepcopy(F,M)
    N=clean(F,N)
```

```
    b=N[-1]**-1
    while len(R)>=len(N):
        u=len(R)-len(N)
        a=int(R[-1]*b)
        factor=[0]*u
        factor.append(a)
        Q=poly_add(F,Q,F(factor))
        R=poly_add(F,R,-poly_mult(F,F(factor),N))
        R=clean(F,R)
    return Q,R

def roots(F,P):
    R=[]
    for i in F.elements:
        if evaluate(F,P,i)==F(0): R.append(int(i))
    return F(R)

#finite field related operations
def least_primitive_root(F):
    for i in F.elements:
        count=2
        while (i**count != F(1)) and (i**count != i):
            count+=1
        if count==F.order-1: return i
    raise Exception("not prime")

def get_RS_generator(F):
    alpha=least_primitive_root(F)
    G=F([1])
    for i in range(1,d):
        G=poly_mult(F,G,F([-alpha**i,1]))
    return G

#coding and decoding operations
def coding(F, M):
    if len(M) != k : raise Exception("message must be of length {}!".format(k))
    mess = F(M)
    G=get_RS_generator(F)
    factor = [0]*(n-k)
    factor.append(1)
```

```
    A=poly_mult(F,mess,F(factor))
    _,B=DE(F,A,G)
    return poly_add(F,B,A)

def decoding(F,C):
    valid=True
    alpha=least_primitive_root(F)
    #tests if coded message is valid
    for i in range(1,d):
        if evaluate(F,C,alpha**i) != F(0): valid=False
    if valid:
        M=[]
        for i in range(n-k,n):
            M.append(int(C[i]))
        print("No error occured during transmission. Original message : {}".format(M))
        return F(M)
    else:
        #syndromes calculation
        S=[]
        for i in range(1,d):
            S.append(int(evaluate(F,C,alpha**i)))
        #initialisation of extended euclidean algorithm
        P=[]
        Q=[]
        A=[]
        B=[]
        #arrays initialization
        factor = [0]*(n-k)
        factor.append(1)
        P.append(F(factor))
        P.append(F(S))
        A.append(F([1]))
        A.append(F([0]))
        B.append(F([0]))
        B.append(F([1]))
        #arrays recursive calculation
        while not(is_null(F,P[-1])):
            q,r=DE(F,P[-2],P[-1])
            P.append(r)
            Q.append(q)
```

```
            A.append(poly_add(F,A[-2],-poly_mult(F,q,A[-1])))
            B.append(poly_add(F,B[-2],-poly_mult(F,q,B[-1])))
        #searching for the good P element
        index=0
        for i in range(1,len(P)):
            if len(P[i])-1<(n-k)/2 and len(P[i-1])-1>=(n-k)/2: index = i
        #syndrome polynomial
        b_0_inv=(evaluate(F,B[index],F(0)))**-1
        sigma=b_0_inv*B[index]
        x=(roots(F,sigma))**-1
        r=np.array(x.log(alpha), dtype=int) #here are stored the positions of errors scattered in our coded message
        r.sort() #reordering to mach the syndromes values
        f=len(r)
        x_final=[]
        for i in range(f):
            x_final.append(int(alpha**r[i]))
        #inverting Gauss system to find error values
        X_g=F.Zeros((f,f))
        for i in range(f):
            for j in range(f):
                X_g[i][j]=int(F(x_final[j])**(i+1))
        S_g=F.Zeros((f,1))
        for i in range(f):
            S_g[i][0]=S[i]
        E=np.linalg.inv(X_g)@S_g
        #recovering message
        M=[]
        for i in range(n-k,n):
            if i in r:
                e, = np.where(r==i)
                M.append(int(C[i]-E[e[0]][0]))
            else: M.append(int(C[i]))
        print("An error occured during transmission. Original message : {}".format(M))
        return F(M)

#channel simulation
def channel(F,M,p):
    M_alt=[]
    F_inv=F.elements[1:]
    for i in range(len(M)):
```

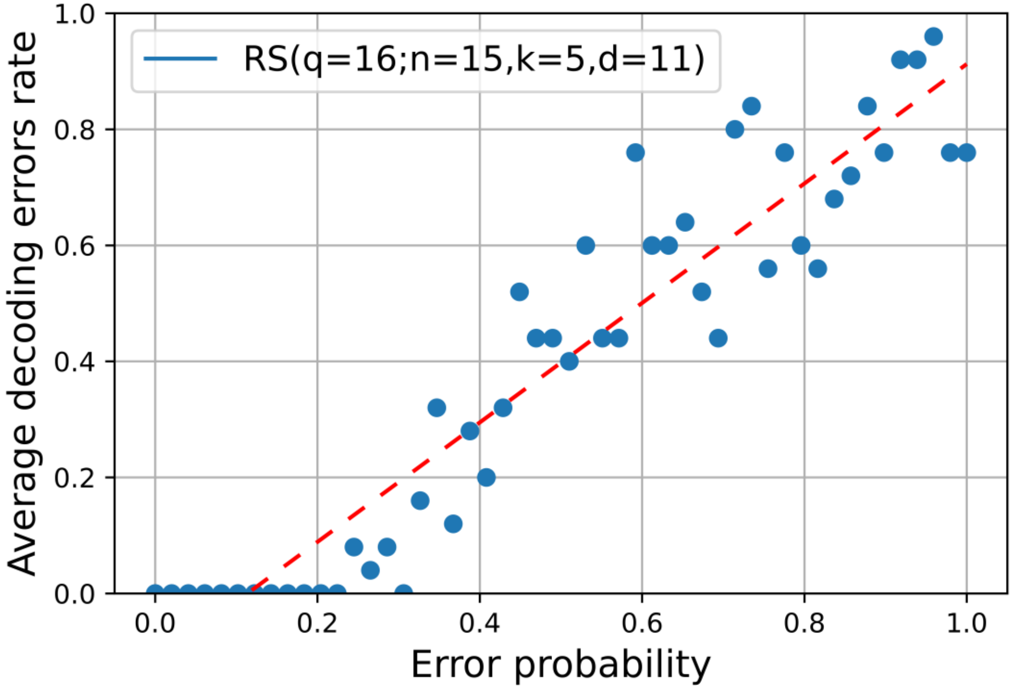
```
            proba=random.uniform(0,1)
            if proba<p:
                M_alt.append(int(M[i]+random.choice(F_inv)))
            else:
                M_alt.append(int(M[i]))
        return F(M_alt)

#statistics gathering
def statistics(F,samples,rate):
    #creating x and y axis
    P=np.linspace(0,1,samples)
    T=[]
    #randomly creating our message
    M=[]
    for i in range(k):
        M.append(int(random.choice(F.elements)))
    M=F(M)
    #making stats for each proba
    for p in P:
        nb_bits_error=0
        for _ in range(rate):
            C=coding(F,M)
            C_alt=channel(F,C,p)
            D=decoding(F,C_alt)
            w=weight(F,M-D)
            if w !=0 : nb_bits_error+=w
        average=nb_bits_error/rate
        T.append(average/k)
    plt.xscale("log")
    #pyplot graph
    points, = plt.plot(P,T,label="RS(q={};n={},k={},d={})".format(q,n,k,d),linestyle ='none',marker ='o')
    plt.legend(handles = [plt.plot([],ls="-", color=points.get_color())[0]], labels=[points.get_label()])
    #making linear regression
    slope, intercept, r_value, p_value, std_err = stats.linregress(P, T)
    def regrlin(x):
        return slope*x+intercept
    plt.plot(P,regrlin(P),color="red", linestyle=(0, (5, 5)))
    #basic parameters
    plt.ylim(0,1)
    plt.xlabel("Error probability")
```

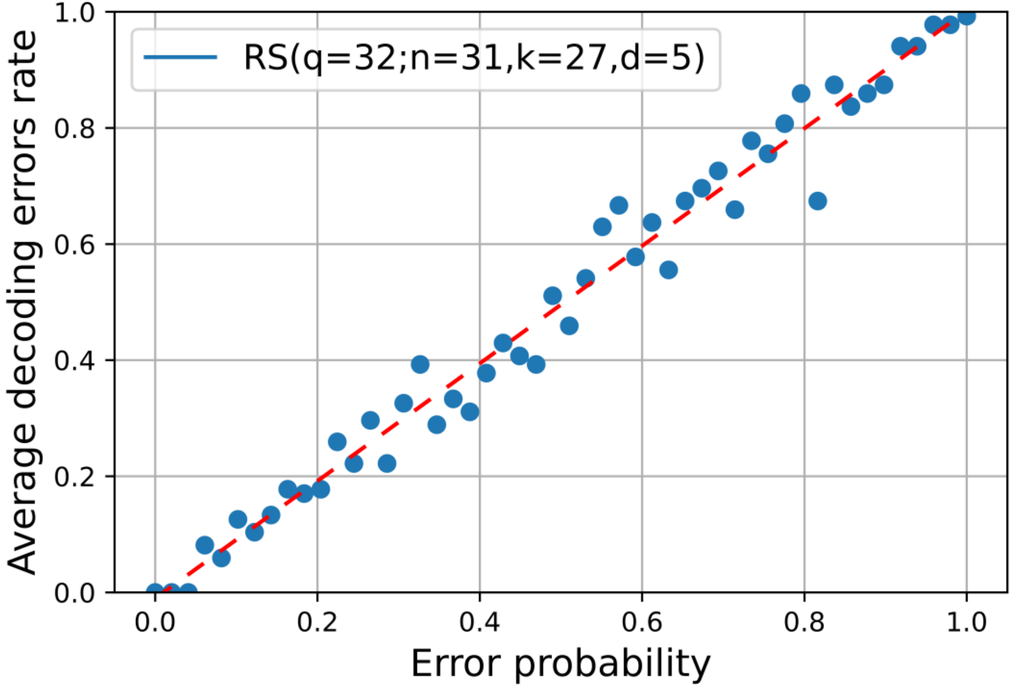
```
plt.ylabel("Average decoding errors rate")
plt.grid()
plt.savefig("RS",dpi=1200)
plt.show()
```

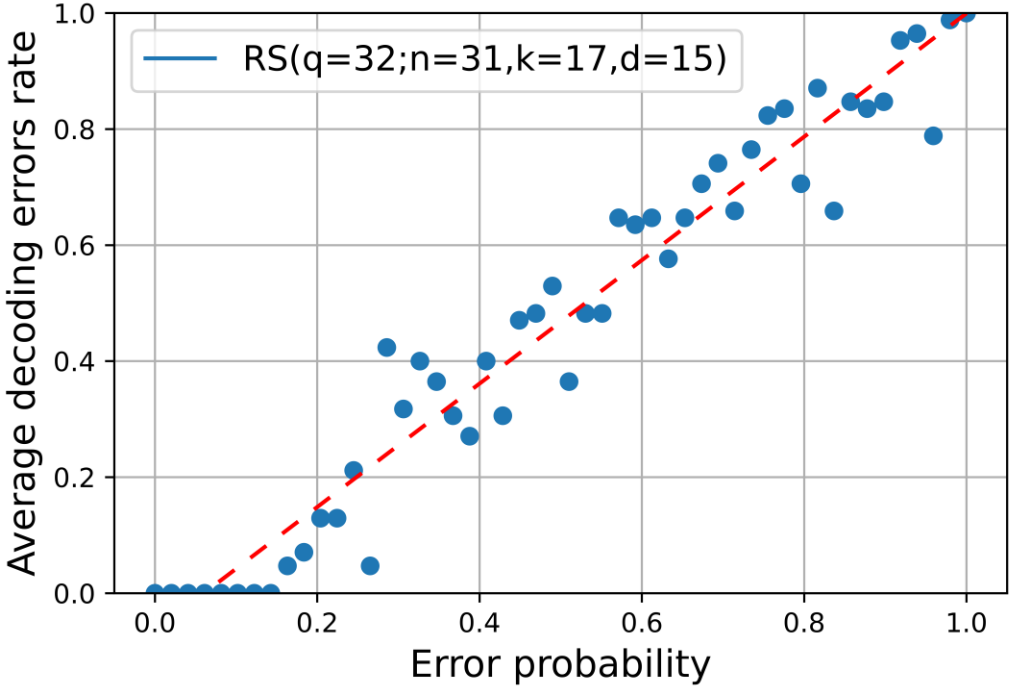
```
from PIL import Image, ImageFilter;
import scipy.ndimage
import matplotlib.image
import matplotlib.pyplot
import numpy as np

im = Image.open('Mario.png')

data = list(im.getdata());

data3 = np.array(data, dtype=np.uint8)

data4=[0]*44100

data4=np.unpackbits(data3)

data5=list(data4)

C=codes.HammingCode(GF(2),7);

Z=GF(2)^120
Y=GF(2)^127

datac=[0]*1411200*7

p=0.01;
Chan=channels.QarySymmetricChannel(GF(2)^127,p);

for i in range(0,11759):
    P=Z(data5[120*i:120*(i+1)])
    X=C.encode(P)
    datac[127*i:127*(i+1)]=X

datae=[0]*1411200*7

for i in range(0,11759):
    U=Y(datac[127*i:127*(i+1)])
    K=Chan.transmit(U)
    datae[127*i:127*(i+1)]=K

datam=[0]*1411200
```

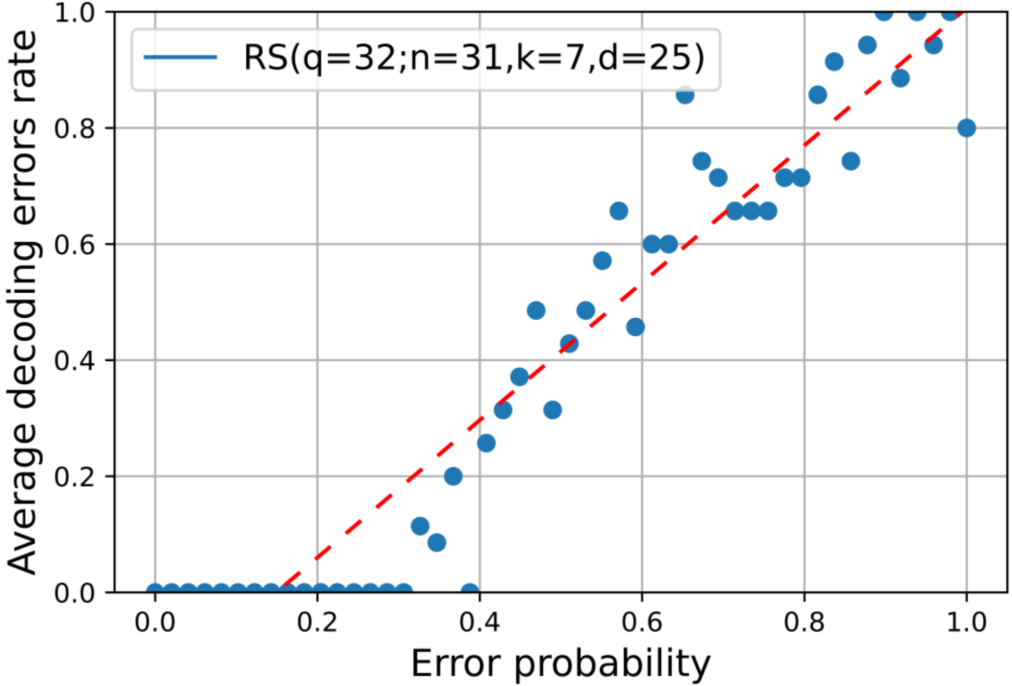
```
for i in range(0,11759):
    P=Y(datae[127*i:127*(i+1)])
    X=C.decode_to_message(P)
    datam[120*i:120*(i+1)]=X

datap=np.array(datam,dtype=np.uint8)

datar=np.packbits(datap)

datar=datar.reshape(44100,4)

datah=tuple(map(tuple,datar))

imf = Image.new("RGB",(210,210),"white")

imf.putdata(datah)

imf
```

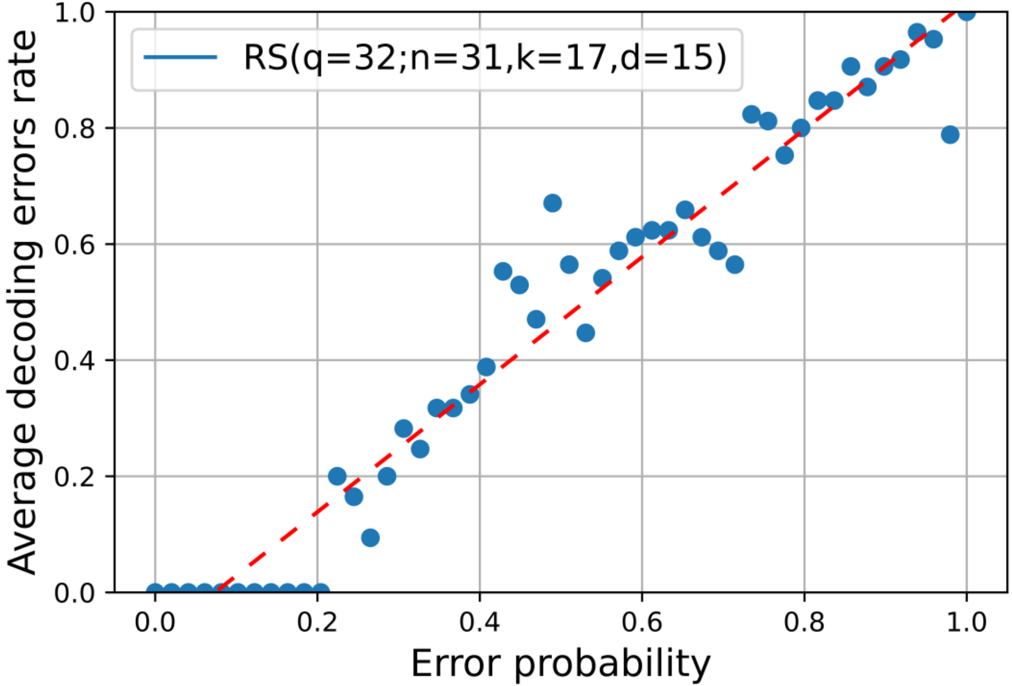
(code réalisé sur la plateforme https://cocalc.com/)


RS(q=16;n=15,k=5,d=11)
Average decoding errors rate
Error probability
0.0
0.2
0.4
0.6
0.8
1.0


RS(q=32;n=31,k=27,d=5)
Average decoding errors rate
Error probability
0.0
0.2
0.4
0.6
0.8
1.0


RS(q=32;n=31,k=17,d=15)
Average decoding errors rate
Error probability
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


RS(q=32;n=31,k=7,d=25)
Average decoding errors rate
Error probability
0.0
0.2
0.4
0.6
0.8
1.0


RS(q=32;n=31,k=17,d=15)
Average decoding errors rate
Error probability
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0