



Remedium Codes

Codes correcteurs avec des
carrés latins

Qu'est-ce qu'un carré latin ?

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$$

Groupe de Klein

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \end{pmatrix}$$

Quasigroupe

Carrés latins orthogonaux d'ordre 4

La vie mode d'emploi

Roman de **Georges Perec** publié en 1978
Prix Médicis 1978



Georges Perec
(1936 - 1982)



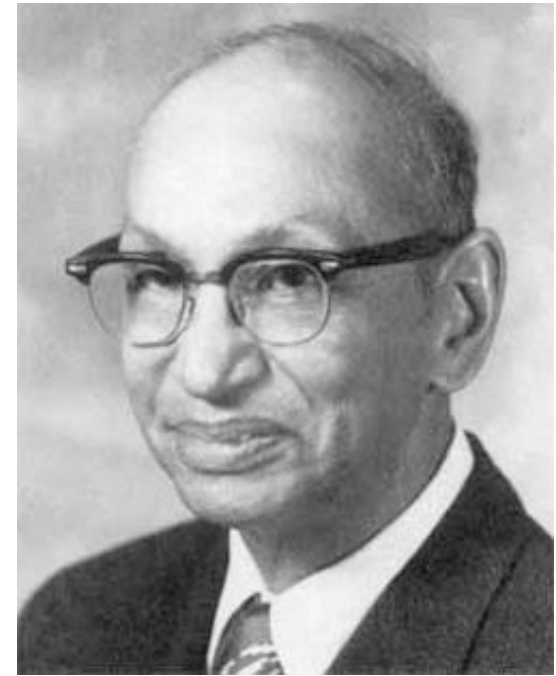
Les carrés latins à travers les âges



Leonhard Euler
(1707-1783)



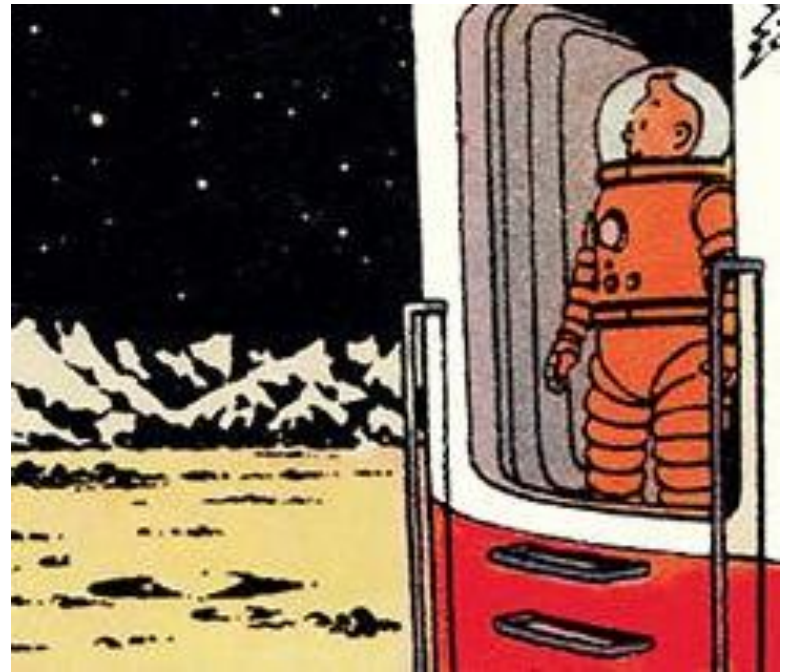
Gaston Tarry
(1843 - 1913)



S. S. Shrikhande
(1917 - 2020)

Présentation du problème

Correction de données endommagées lors
d'un voyage spatial



Variables de l'algorithme

On entre un code de taille m^2 .

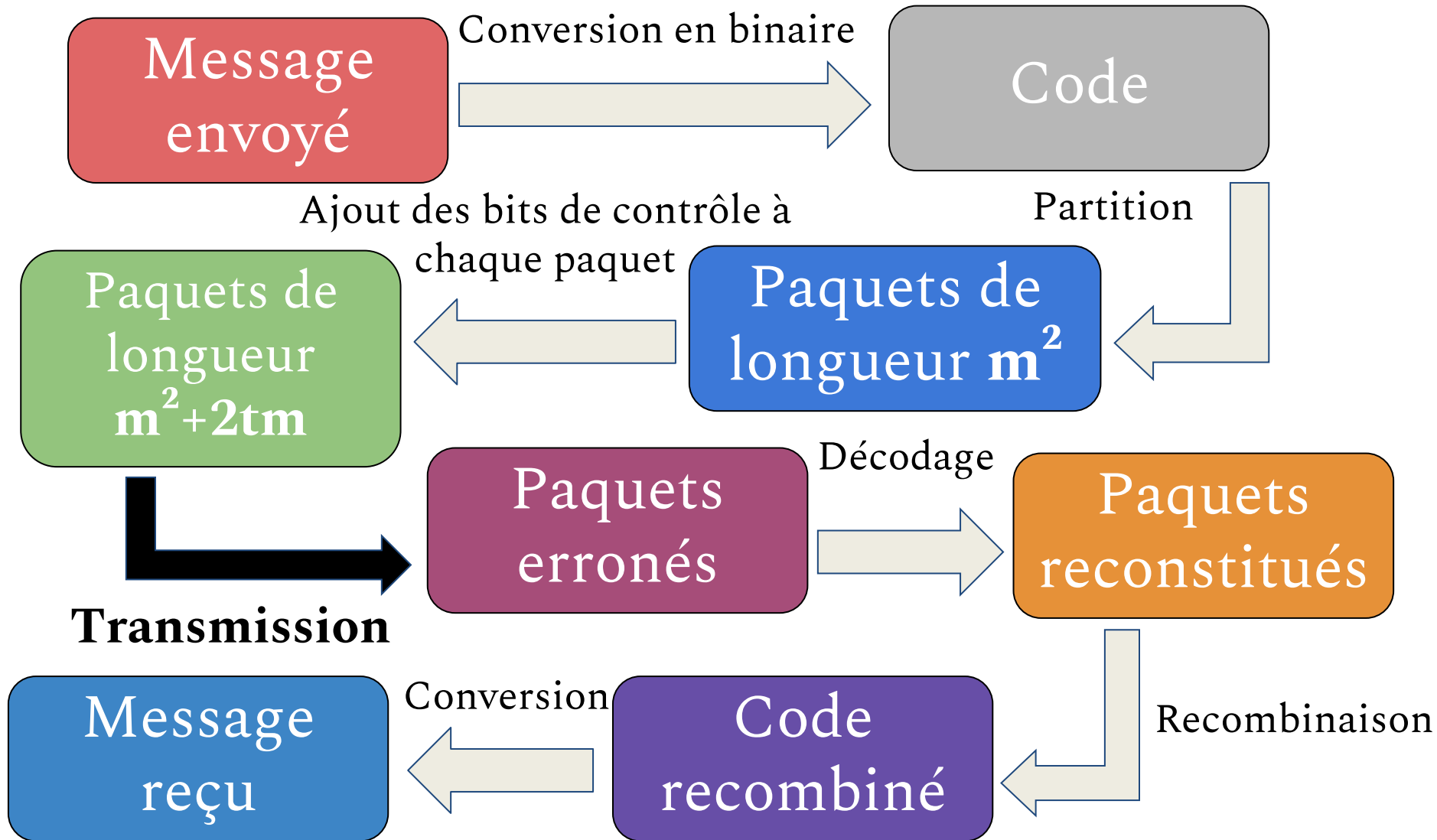
h est le nombre de carrés latins orthogonaux de taille m :

$$h \geq \min(p_i^{e_i} - 1)$$

Le code correcteur peut corriger t erreurs.

$$t \leq \frac{h}{2} + 1, \quad t \in \mathbb{N}^*$$

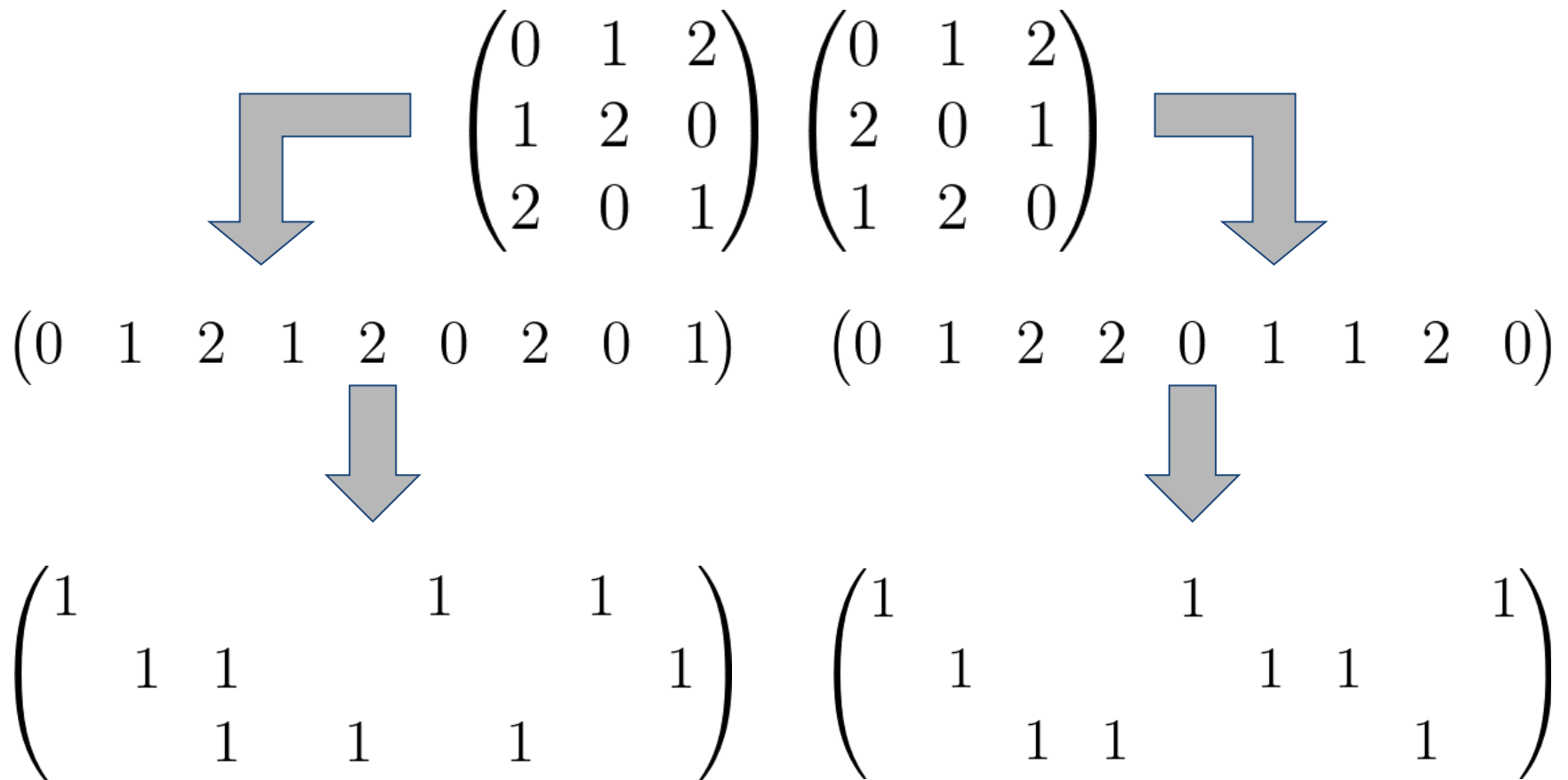
Structure de l'algorithme



Création de la matrice de contrôle

On prend les paramètres **m=3** et **t=2**

Voici deux carrés latins orthogonaux d'ordre 3 :



The diagram illustrates the construction of a control matrix from two orthogonal Latin squares of order 3. The process is shown in three stages:

1. Two orthogonal Latin squares of order 3 are shown side-by-side:

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

2. The squares are concatenated into a single 3x6 matrix:

$$(0 \ 1 \ 2 \ 1 \ 2 \ 0 \ 2 \ 0 \ 1) \quad (0 \ 1 \ 2 \ 2 \ 0 \ 1 \ 1 \ 2 \ 0)$$

3. The concatenated matrix is transformed into a 6x6 matrix of 1s and 0s, representing the control matrix:

$$\begin{pmatrix} 1 & & & & 1 & & 1 & \\ & 1 & 1 & & & & & 1 \\ & & 1 & & 1 & & 1 & \\ & & & 1 & & 1 & & \\ & & 1 & & 1 & & & \\ & & & & & & 1 & \end{pmatrix} \quad \begin{pmatrix} 1 & & & 1 & & & & 1 \\ & 1 & & & 1 & 1 & & \\ & & 1 & 1 & & & & 1 \\ & & & & & & 1 & \end{pmatrix}$$

Création de la matrice de contrôle

M1

Matrices lignes

M2

Matrices identité

M3 et M4

Matrices créées avec les carrés latins

Diagram illustrating a matrix structure with dimensions m^2 (width) and $2tm$ (height). The matrix is partitioned into four quadrants by a vertical line after the 5th column and a horizontal line after the 5th row. The matrix contains 1s and 0s in a symmetric pattern.

Ajout des bits de contrôle

On veut transmettre le code suivant A_1 :

$$A_1 = (0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0)$$

Les bits de contrôle sont l'image par **l'application linéaire H** :

$$\begin{pmatrix} 1 & 1 & 1 & & & & & & \\ & & & 1 & 1 & 1 & & & \\ & & & & & & 1 & 1 & 1 \\ 1 & & & 1 & & & 1 & & \\ & 1 & & & 1 & & & 1 & \\ & & 1 & & & 1 & & & 1 \\ 1 & & & & 1 & & 1 & & \\ & 1 & 1 & & & & & 1 & \\ & & 1 & & 1 & & 1 & & \\ 1 & & & 1 & & & & & 1 \\ & 1 & & & 1 & 1 & & & \\ & & 1 & 1 & & & 1 & & \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Le code à transmettre est alors A_2 :

$$A_2 = (0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1)$$

```
def bits_de_controle(d,m,t):  
    if len(d)!=m**2:  
        return "Mauvaise taille de  
liste"  
    H=matH(m,t)  
    for i in range(2*t*m):  
        L=[]  
        for j in range(m**2):  
            if H[i][j]==1:  
                L.append(d[j])  
        d.append(xor(L))  
    return d
```

Décodage

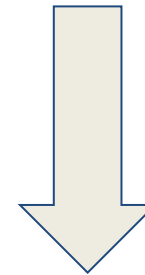
$$A_2 = (0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1)$$



Transmission : deux erreurs apparaissent

$$A_3 = (0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1)$$

```
def decode(d,m,t):
    L=[] #liste correspondant au code
    corrigé
    H=[d]+matH(m,t)
    for i in range(m**2):
        S=[d[i]] #liste des "suffrages"
        for j in range(1,2*m*t+1):
            X=[] #liste des éléments de la
            ligne
            if H[j][i]==1:
                for k in range(m**2+2*m*t):
                    if k!=i and H[j][k]==1:
                        X.append(d[k])
                S.append(xor(X))
            L.append(vote(S))
    return L
```



Décodage des
bits initiaux

$$A_4 = (0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0)$$

Application : transmission d'une image

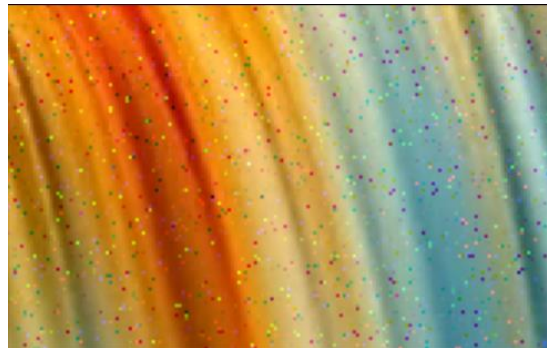
Transmission d'une image avec et sans contrôle

Probabilité d'erreur de transmission : **1 %**

Les erreurs sont uniformément réparties



Image envoyée



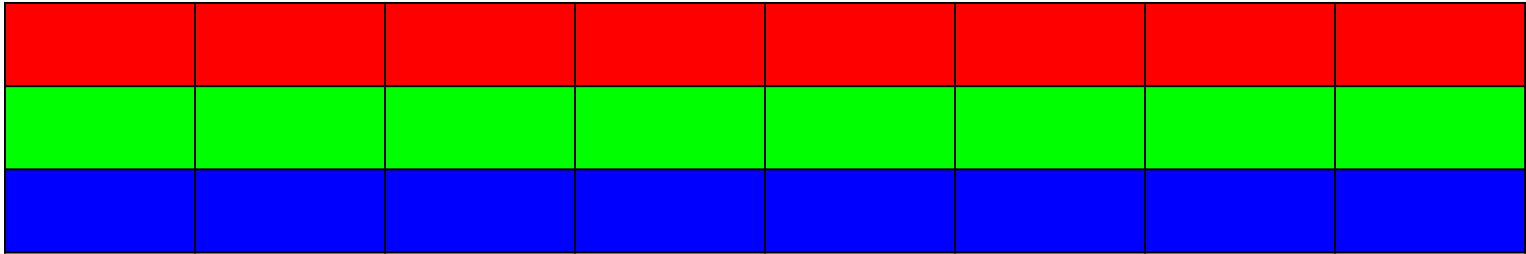
Sans contrôle
21 % d'erreurs



Avec contrôle ($m=7, t=4$)
0,1 % d'erreurs

Probabilité d'erreur sur un pixel

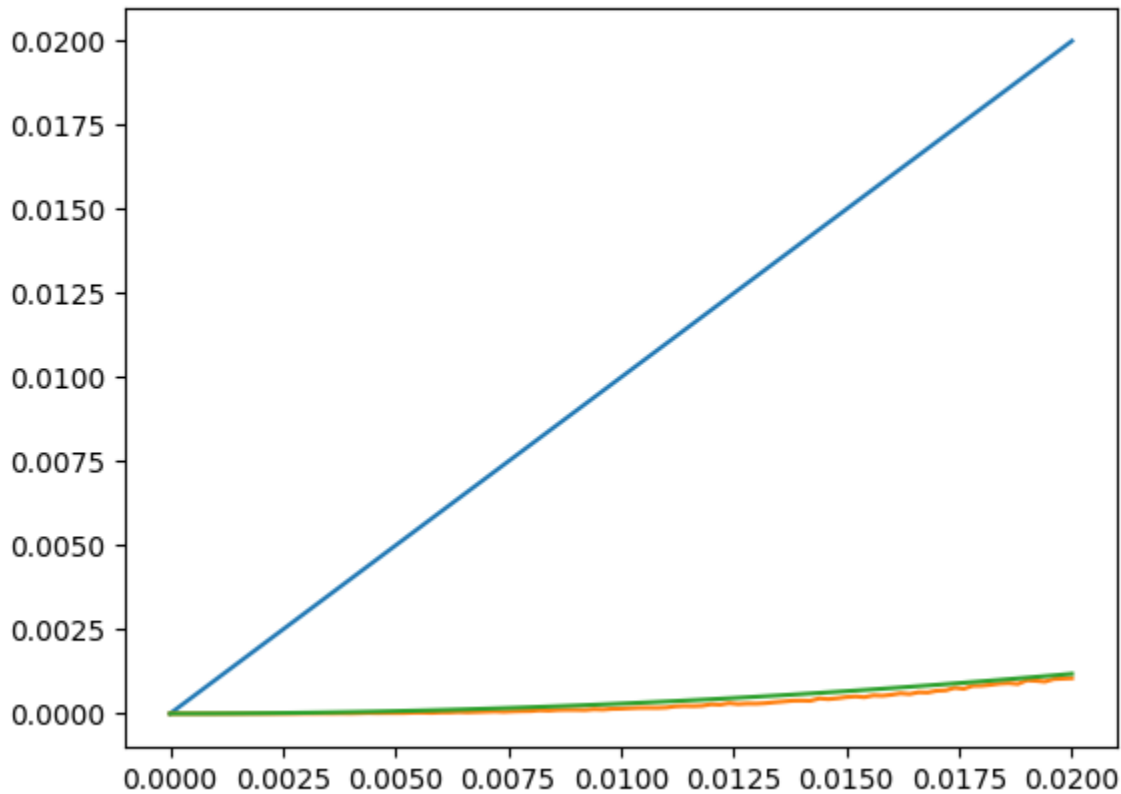
Un pixel est codé sur 24 bits :



$$\begin{aligned} P(\text{erreurs sur un pixel}) &= \sum_1^{24} \binom{24}{k} 0,01^k 0,99^{24-k} \\ &= 21,43\% \end{aligned}$$

Correction pour des faibles taux d'erreurs

Part d'erreurs lors de la réception



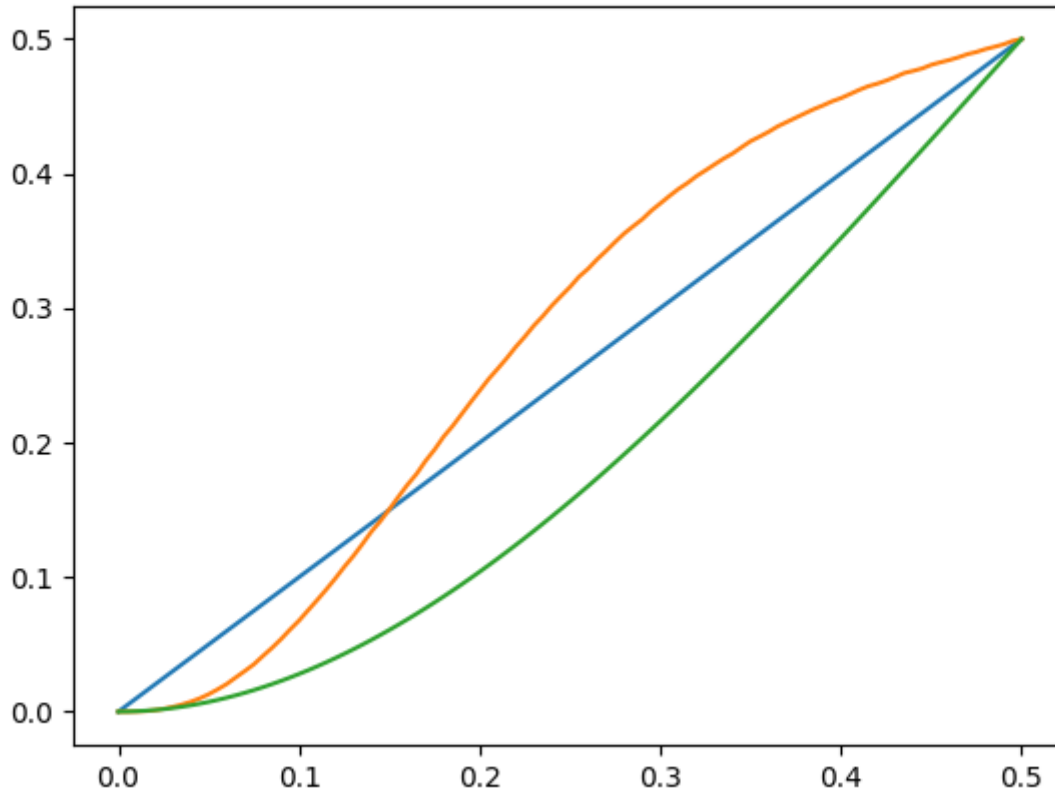
Part d'erreurs lors de la transmission

Fonctions de correction :

- Sans contrôle
- Contrôle avec les carrés latins (m=3, t=2)
- Trois transmissions successives

Correction pour toutes les taux d'erreurs

Part d'erreurs lors de la réception



Part d'erreurs lors de la transmission

Fonctions de correction :

■ Sans contrôle

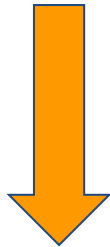
■ Contrôle avec
les carrés latins
($m=3$, $t=2$)

■ Trois
transmissions
successives

Erreurs aléatoires · Erreurs de rafale

Erreurs aléatoires

[1, 1, 0, 0, 1, 0, 1, 0, 1]



[1, **0**, 0, **1**, 1, 0, **0**, 0, 1]

Erreurs aléatoires
uniformément réparties

Erreurs de rafale

[1, 1, 0, 0, 1, 0, 1, 0, 1]



[1, **0**, **0**, **0**, **0**, **0**, **0**, 0, 1]

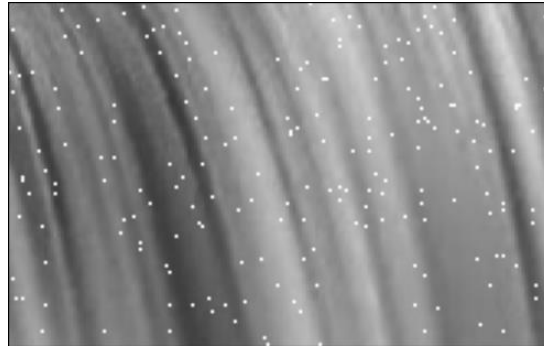
Erreurs localisées dans la
bande de rafale de longueur t
Ici, $t = 6$

Transmission d'une image, erreurs de rafale

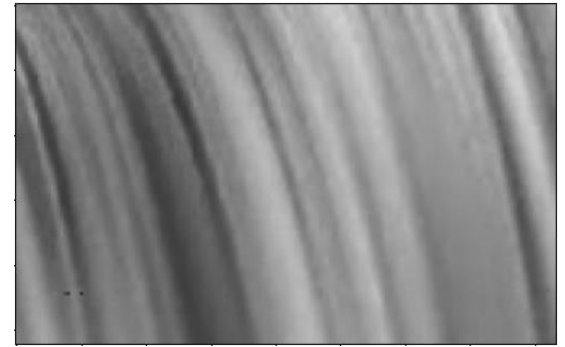
Transmission d'une image avec et sans contrôle
Erreurs de rafale



Image envoyée



*Sans contrôle
1,1 % d'erreurs*



*Avec contrôle
0,05 % d'erreurs*

Limites du code correcteur

Pour **m premier**, $t_{max} = \frac{m+1}{2}$

Rafale de longueur **t** : $m \geq 2t_{max} - 1$

Or, il doit y avoir au maximum **t** erreurs par paquet donc la proportion d'erreurs maximale est

$$\frac{t_{max}}{m^2 + 2t_{max}m} = \frac{m+1}{4m^2 + 2m} \sim \frac{1}{4m} \xrightarrow{m \rightarrow \infty} 0$$

Sources

- [1]: M. Y. Hsiao, D. C. Bossen and R. T. Chien, "Orthogonal Latin Square Codes," in *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 390-394, July 1970, doi: 10.1147/rd.144.0390
- [2]: R. Datta and N. A. Touba, « Generating Burst-error Correcting Codes from Orthogonal Latin Square Codes – a Graph Theoretic Approach » in 2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems
- [3]: Oulipo, *Atlas de littérature potentielle*, Gallimard, Paris, 1981
- [4]: J.H. Van Lint, R. M. Wilson, *A course in combinatorics*, Cambridge University Press, Cambridge, 1992
- [5]: H.B.Mann, “*Analysis and Design of Experiments*”, Dover Publications, New-York, 1949

Annexe : Structure du programme

Générations des carrés latins orthogonaux d'ordre **m** :

- `fact_premiere(m)`
- `nb_carres_orth(m)`
- `crea_tous_carres_orth(m)`

Construction de la matrice de contrôle :

- `matM1(m)`
- `matM2(m)`
- `vectVmu(mu, m, L)`
- `matMi(i, m, LAT)`
- `matH(m, t)`

Codage :

- `bits_de_controle(L,m,t)`
- `ajout_bits_de_controle(L,m,t)`

Transmission :

- `transmi_proba(L,p)`
- `transmi_rafale_8bit(L,d_min,d_max,n)`
- `trois_transmissions(L,p)`

Décodage :

- `decode(L,m,t)`
- `recombineur(L,m,t)`

Fonctions auxiliaires :

- `partition(L,dis)`
- `xor(L)`
- `vote(L)`

Traitement de l'image :

- `conv_binaire(n)`
- `convertisseur_jpeg_vers_binaire(I)`
- `liste4_a_1(L)`
- `liste1_a_4(L,n,m)`
- `conv_base_10(N)`
- `convertisseur_binaire_vers_jpeg(L)`

Les graphiques ont été tracés grâce au module
`matplotlib.pyplot`

```

def fact_premiere(n): #Renvoie la liste des facteurs premiers de n pour n<200
avec leur multiplicité
    PRE=[2,3,5,7,9,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97
    ,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,
    197,199]
    P=[]
    for i in PRE:
        while n%i==0:
            P.append(i)
            n=n//i
    m=len(P)
    L=[[P[0],1]]
    for j in range(1,m):
        if P[j]!=P[j-1]:
            L.append([P[j],0])
    c=0
    for k in range(1,m):
        if P[k]!=P[k-1]:
            c+=1
        L[c][1]+=1
    return L

def nb_carres_orth(n): # min(pi**ei - 1)
    P=fact_premiere(n)
    m=P[0][0]**P[0][1] - 1
    l=len(P)
    for i in range(l):
        if P[i][0]**P[i][1] - 1 < m:
            m=P[i][0]**P[i][1] - 1
    return m

```

```
def crea_tous_carres_orth(n) :  
    h=nb_carres_orth(n)  
    LAT=[]  
    for j in range(1,h+1):  
        #Creation des h carres  
        #Parcours vertical puis horizontal avec à  
chaque fois application des actions sur les lignes  
ou les colonnes  
        L=[]  
        for x in range(n):  
            L.append([])  
            for y in range(n):  
                L[x].append((j*x)%n)  
        for r in range(n):  
            for s in range(n):  
                L[s][r]=((L[s][r]+r)%n)  
        LAT.append(L)  
    return LAT
```

```

def matM1(m) : #crée la matrice M1
    M=[]
    for i in range(m):
        M.append([])
        for j in range(m*m):
            if j>=m*i and j<m*(i+1):
                M[i].append(1)
            else:
                M[i].append(0)
    return M

```

```

def matM2(m) : #crée la matrice M2
    M=[]
    for i in range(m):
        M.append([])
        for j in range(m*m):
            if j%m == i:
                M[i].append(1)
            else:
                M[i].append(0)
    return M

```

```

def vectVmu(mu,m,L) :
    V=[]
    for j in range(m):
        for k in range(m):
            if L[j][k]==mu:
                V.append(1)
            else:
                V.append(0)
    return V

```

```

def matMi (i,m,LAT) :
    L=LAT[i-3]
    Mi=[]
    for mu in range(m) :
        Mi.append(vectVmu(mu,m,L) )
    return Mi

```

```

def math(m,t) : #crée H
    H=[]#listes internes : lignes
    M1=matM1(m)
    M2=matM2(m)
    H=M1+M2
    LAT=crea_tous_carres_orth(m)
    for i in range(3,2*t+1) :
        M=matMi(i,m,LAT)
        H=H+M
    for j in range(2*t*m) :
        for k in range(2*t*m) :
            if k==j :
                H[j].append(1)
            else :
                H[j].append(0)
    return H

```



```

def bits_de_controle(d,m,t):
    if len(d) != m**2:
        return "Mauvaise taille de liste"
    H=matH(m,t)
    for i in range(2*t*m):
        L=[]
        for j in range(m**2):
            if H[i][j]==1: L.append(d[j])
        d.append(xor(L))
    return d

```

```

def ajout_bits_de_controle(L,m,t): #Prend une liste de bits et y
ajoute les bits de contrôle
    P=partition(L,m**2)
    N=[]
    for i in range(len(P)):
        B=bits_de_controle(P[i],m,t)
        for j in range(len(B)):
            N.append(B[j])
    return N

```

```
def transmi_proba(L,p): #pour chaque bit d'une liste de  
bits L, il y a une probabilité p qu'il y ait une erreur de  
transmission
```

```
    for j in range(len(L)):  
        if random()<p:  
            if L[j]==0:  
                L[j]=1  
            else:  
                L[j]=0  
    return L
```

```
def transmi_rafale_8bit(L,d_min,d_max,n):
```

```
    for j in range(n):  
        if d_min==d_max: d=d_min  
        else: d=random.randint(d_min,d_max)  
        R=random.randint(0,len(L)//8-1-8*d)  
        for i in range(8*R,8*R+8*d):  
            L[i]=1  
    return L
```

```
def trois_transmissions(L,p) :
```

```
    M1=L[:]
```

```
    M2=L[:]
```

```
    M3=L[:]
```

```
    M1=transmi_proba(M1,p)
```

```
    M2=transmi_proba(M2,p)
```

```
    M3=transmi_proba(M3,p)
```

```
    S=[]
```

```
    for i in range(len(L)) :
```

```
        v=vote([M1[i],M2[i],M3[i]])
```

```
        S.append(v)
```

```
    return S
```

```
def decode(d,m,t) :
```

```
    L=[] #liste correspondant au code corrigé
```

```
    H=[d]+matH(m,t)
```

```
    for i in range(m**2) :
```

```
        S=[d[i]] #liste des "suffrages"
```

```
        for j in range(1,2*m*t+1) :
```

```
            X=[] #liste des éléments de la ligne
```

```
            if H[j][i]==1:
```

```
                for k in range(m**2+2*m*t) :
```

```
                    if k!=i and H[j][k]==1:
```

```
                        X.append(d[k])
```

```
                S.append(xor(X))
```

```
            L.append(vote(S))
```

```
    return L
```

```
def recombineur(L,m,t) :
```

```
#L est la liste après transmission_image , la  
fonction renvoie une liste décodée de binaires
```

```
    P=partition(L,m**2+2*m*t)
```

```
    p=len(P)
```

```
    CORRI=[]
```

```
    for i in range(p) :
```

```
        CORRI.append(decode(list(P[i]),m,t))
```

```
    CORR=[]
```

```
        for j in range(len(CORRI)) :
```

```
            for k in range(len(CORRI[0])) :
```

```
                CORR.append(CORRI[j][k])
```

```
    for l in range(L[1]*L[2]%(m**2)) :
```

```
        a=CORR.pop()
```

```
    return CORR
```

def partition(L,dis): #partitionne une liste en bouts de longueur m^2 .
Si il y a des éléments en trop, on rajoute des 0

```
n=len(L)
c=0
k=0
M=[[]]
for i in range(n):
    c+=1
    M[k].append(L[i])
    if c==dis:
        M.append([])
        k+=1
        c=0
r=n%dis
for i in range(dis-r):
    M[k].append(0)
return(M)
```

def xor(L): #Fonction prenant en entrée une liste de bits, renvoyant
le xor de la liste

```
if len(L)<2:
    return "Liste trop petite"
else:
    B=abs(L[0]-L[1])
    for i in range(2,len(L)):
        B=abs(B-L[i])
    return B
```

```
def vote(L) : #compte les suffrages d'une liste de bits
    a0=0
    a1=0
    for i in range(len(L)):
        if L[i]==0: a0+=1
        elif L[i]==1: a1+=1
        else: return ("Vote interdit",L[i])
    if a0>=a1: return 0
    else: return 1
```

```
def conv_binaire(n):#Convertit un entier naturel de [0,255]
en binaire avec la division euclidienne
    B=[]
    N=n
    for i in range(8):
        r = N%2
        N = N//2
        B.append(r)
    list.reverse(B)
    return(B)
```



```
def convertisseur_jpeg_vers_binaire(L): #Convertit des listes d'images en format
jpeg en liste de listes binaires (un binaire entre 0 et 255 est codé par une
liste de 0 et de 1 de longueur 8)
```

```
    hau=len(L)
    lar=len(L[0])
    M=[]
    for i in range(hau):
        M.append([])
        for j in range(lar):
            M[i].append([0,0,0])
            for k in range(3):
                M[i][j][k]=conv_binaire(L[i][j][k])

    return (M)
```

```
def liste4_a_1(L):
```

```
    N=[]
    for i in range(len(L)):
        for j in range(len(L[i])):
            for k in range(len(L[i][j])):
                for l in range(len(L[i][j][k])):
                    N.append(L[i][j][k][l])

    return N
```

```

def listel_a_4(L,n,m):
    N1=[]
    c=0
    for i in range(n):
        N2=[]
        for j in range(m):
            N3=[]
            for k in range(3):
                N4=[]
                for l in range(8):
                    N4.append(L[c])
                    c+=1
                N3.append(N4)
            N2.append(N3)
        N1.append(N2)
    return N1

```

```

def conv_base_10(N):#convertit un binaire sous forme de liste en base
10 si il est inférieur à 255, len(N)=8 avec l'algorithme de Hörner.
    l=0
    for i in range(8):
        l= 2*l + N[i]
    return(l)

```

```
def convertisseur_binaire_vers_jpeg(L) :  
    hau = len(L)  
    lar = len(L[0])  
    M=[]  
    for i in range(hau):  
        M.append([])  
        for j in range(lar):  
            M[i].append([0,0,0])  
            for k in range(3):  
                M[i][j][k] = conv_base_10(L[i][j][k])  
    return (M)
```