# Transmission d'informations entre la Terre et les sondes spatiales

Tom Hubrecht

269

# Table des matières

# Introduction

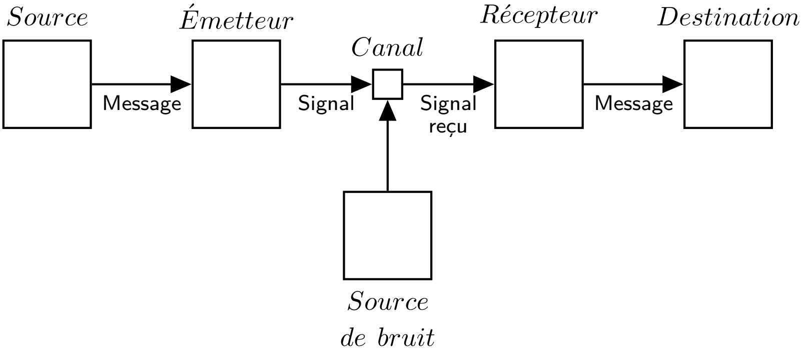
# Problématique

# Exploration spatiale

La sonde Voyager 2 communique avec la Terre alors qu'elle se trouve à 18 milliards de kilomètres.

# Modélisation

# Théorie de l'information (C. Shannon)



# Cas de l'espace

- Canal de propagation gaussien
- Capacité : $C = W.\log_2(1 + \frac{E_b}{N_0})\ bit.s^{-1}$
  - $W$ : nombre de bits émis par seconde
  - $\frac{E_b}{N_0}$ : rapport signal sur bruit

# Codes correcteurs d'erreurs

# Principe

# Principe

- Permettre de contrer l'action du bruit
- Se rapprocher de la limite de Shannon

# Définitions

- Taux de transmission : $R = \frac{\text{nombre de bits du message}}{\text{nombre de bits envoyés}}$
- Probabilité a posteriori (PAP)
- Rapport de vraisemblance logarithmique (LLR) : $\Lambda(d_k) = \log(\frac{\mathbf{P}(d_k=1)}{\mathbf{P}(d_k=0)})$

# Différentes méthodes

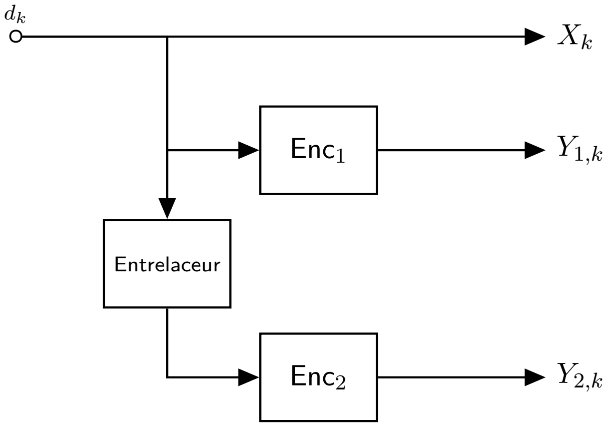
# Plusieurs classes de codes

- Turbocodes
- Codes LDPC (Low Density Parity Checks)

# Plusieurs types de décodage

- Décodage dur (Hard Decoding)
- Décodage à décision douce (Soft decoding)

# Turbocodes



# Décodage d'un message codé à l'aide de turbocodes

Soit $R_1^N$ le message reçu, $S_k$ l'état de codeur après lecture du bit $X_k$
$\forall k \in \{1, N\}$ on calcule $\Lambda(x_k)$ à l'aide de
$\alpha_k^i(m) = \frac{\mathbf{P}(X_k = i, S_k = m, R_1^k)}{\mathbf{P}(R_1^k)}\mathbf{P}(X_k = i, S_k = m | R_1^k)$ et
$\beta_k(m) = \frac{\mathbf{P}(R_{k+1}^N | S_k = m)}{\mathbf{P}(R_{k+1}^N | R_1^k)}$
on obtient alors $\mathbf{P}(x_k = 1 | R_1^N)$ et $\mathbf{P}(x_k = 0 | R_1^N)$ d'où $\Lambda(x_k)$
car $\mathbf{P}(x_k = i | R_1^N) = \sum_m \alpha_k^i(m)\beta_k(m)$

# Codes LDPC

- Sommes de contrôle
- Multiplication matricielle
- Utilisation de matrices peu denses

# Décodage d'un message codé à l'aide d'un code LDPC

- Traduction du message reçu à l'aide d'une fonction seuil
- Calcul des sommes au niveau des noeuds de contrôle
- Modification des noueds messagers reliés aux noeuds de contrôles non satisfaits
- Itération jus'à satisfaction des noeuds de contrôle où itération maximale atteinte

# Implémentation des codes correcteurs

# Représenter l'efficacité des codes

# Choix de l'image

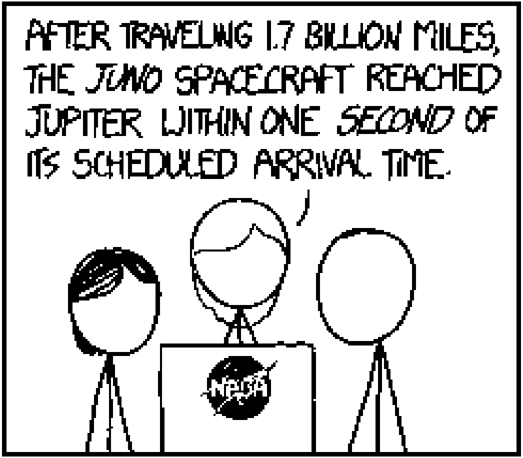



FIGURE – Juno, XKCD

# Processus

- Transformation de l'image en liste de bits
- Codage du message obtenu
- Rajout d'un bruit blanc de moyenne nulle et de variance $s^2$

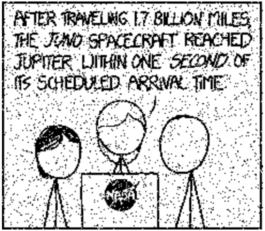
  $x_k = (2.d_k - 1) + a_k$ où $a_k \hookrightarrow \mathcal{N}(0, s)$

  $\frac{E_b}{N_0} = \frac{1}{s}$
- Enregistrement du message bruité
- Décodage du message et enregistrement
- Recréation des images obtenues à l'aide des listes de bits

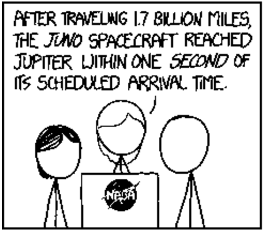
# Résultats

# Bruit faible

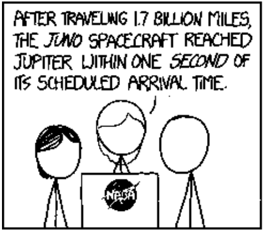
# $SNR_{dB} = 2.2$ dB




Avant décodage



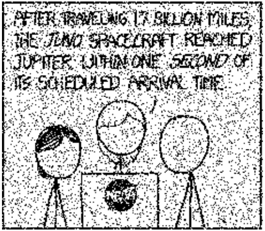

Après décodage avec turbocodes



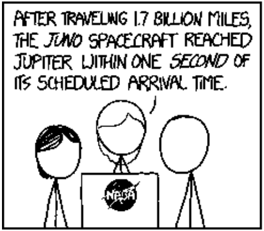

Après décodage avec codes LDPC

# Bruit élevé

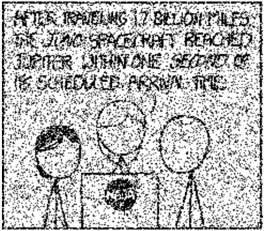
# $SNR_{dB} = 0.97$ dB




Avant décodage



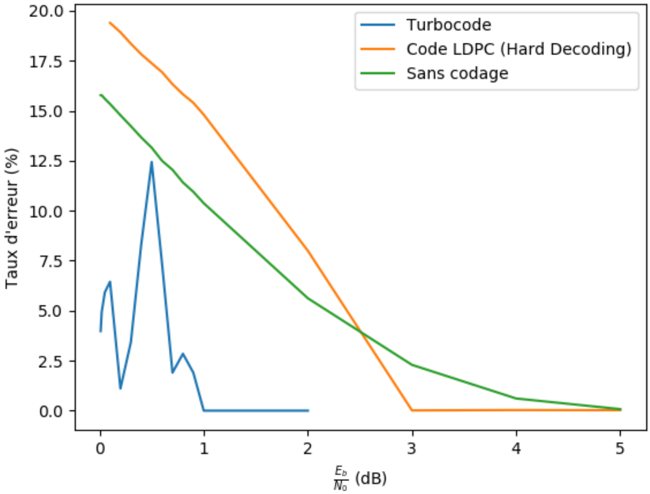

Après décodage avec turbocodes



Après décodage avec codes LDPC

# Taux d'erreur

# Taux d'erreur des deux codes



# Conclusion

# Bibliographie

- Erwan Lecomte. *Comment Voyager communique-t-elle avec la Terre ?*. Oct. 2018. URL : https://www.sciencesetavenir.fr/espace/comment-voyager-dialogue-t-elle-avec-la-terre_33840
- Claude E Shannon. « A Mathematical Theory of Communication ». In : *The Bell System Technical Journal* 27 (juil. 1948)
- Robert G. Gallager. « Low-Density Parity-Checks Codes ». Cambridge, Mass, juil. 1963
- C. Berrou, A. Glavieux et P. Thitimajshima. « Near Shannon Limit Error-Correcting Coding and Decoding : Turbo-Codes ». In : *Proceedings of ICC '93*. Sous la dir. d'IEEE. 1993
- CCSDS. *TC Synchronization and Channel Coding. Issue 3, Recommended Standard*. 231.0-B-3. Washington, D.C, sept. 2017
- XKCD. *Juno*. Mar. 2019. URL : https://xkcd.com/1703/

# Code Informatique

# Codage d’un turbocode

```c
h_list * encode_turbo(h_list *buf)
{
    h_list *buf_e = chl(3 * (buf->n + 4), 3 * (buf->n + 4));
    bar *registerA = initMemState(4);
    bar *registerB = initMemState(4);
    char a;
    char b;
    char d;
    for(size_t i = 0; i < buf->n; i++)
    {
        d = buf->list[i];
        buf_e->list[3 * i] = d;
        a = yieldEncode(d, registerA);
        buf_e->list[3 * i + 1] = a;
        d = buf->list[pi(i)];
        b = yieldEncode(d, registerB);
        buf_e->list[3 * i + 2] = b;
    }
    // Clean the registers
    for(size_t i = buf->n; i < (buf->n + 4); i++)
    {
        d = barget(registerA, 3) ^ barget(registerA, 2);
        buf_e->list[3 * i] = d;
        a = yieldEncode(d, registerA);
        buf_e->list[3 * i + 1] = a;
        d = barget(registerB, 3) ^ barget(registerB, 2);
        b = yieldEncode(d, registerB);
        buf_e->list[3 * i + 2] = b;
    }
    bardestroy(registerA);
    bardestroy(registerB);
    return buf_e;
}
```

```c
char yieldEncode(char d, bar *m)
{
    char a = d ^ barget(m, 0) ^ barget(m, 2) ^ barget(m, 3);
    char g = d ^ barget(m, 2) ^ barget(m, 3);

    barshl(m, 1, g);

    return a;
}
```

```c
h_list * decode_turbo_iter(s_list *buf, double s, size_t i_max)
{
    char done = 0;
    char interleaved = 0;
    size_t iter = 0;
    double Mm[2];
    size_t n = buf->n / 3;
    s_list *llr = csl(n, n);    // The 0-th bit is not considered
    s_list *X1 = csl(n, n);
    s_list *X2 = csl(n, buf->n);
    s_list *Y1 = csl(n, n);
    s_list *Y2 = csl(n, n);
    double t;
    int k;
    split_s(buf, X1, Y1, Y2);
    while (!done && iter < i_max)
    {
        iter ++;
        interleaved = 0;
        k = decode_part(X1, Y1, llr, s);
        substract_s(llr, X1);
        max_min(Mm, llr);
        if (Mm[0] > -5000000 || Mm[1] < 5000000)
        {
            interleaved = 1;
            // We need to interleave the llr to match the pattern of Y2
            interleave(X2, llr);
            k = decode_part(X2, Y2, llr, s);
            substract_s(llr, X2);
            // Update X1 with the new values
            deinterleave(X1, llr);}
        if (Mm[0] < -5000000 && Mm[1] > 5000000)
        {
            done = 1;}}
    h_list *res = recreate(llr, interleaved);
    return res;
}
```

# Initialisation

```
size_t n = X->n;
s_list *alpha = csl(32*(n + 1), 32*(n + 1));
s_list *beta = csl(16*(n + 1), 16*(n + 1));
s_list *gamma = csl(32*(n + 1), 32*(n + 1));
s_list *lambda = csl(32*(n + 1), 32*(n + 1));
s_list *a = csl(n + 1, n + 1);
double tmp[2];

size_t d; // The value of the k-th bit
size_t b; // The value of the k-th encoded bit
size_t m; // The previous state of the register
size_t i;
double x;
double y;

alpha->list[0] = 1.0;
alpha->list[1] = 1.0;
a->list[0] = 1.0;
a->list[n] = 1.0;
beta->list[16*n] = 1.0;

lambda->list[32 * n] = alpha->list[32 * n];
lambda->list[1 + 32 * n] = alpha->list[1 + 32 * n];
```

# Calcul des $\alpha$ et $\gamma$

```
for(size_t k = 1; k <= n; k++) // k-th bit of the message
 {
     x = X->list[k - 1];
     y = Y->list[k - 1];

     for(size_t S = 0; S < 16; S++) // Register state of the encoder
     {
         d = 0;
         m = S/2 + (S & 8) ^ 8*((S & 1) ^ d);
         b = d ^ (m & 1) ^ (m & 4)/4 ^ (m & 8)/8;
         gamma->list[2*S + 32*k] = pTrans(x, d, s) * pTrans(y, b, s);
         i = 2*m + 32*(k-1);
         alpha->list[2*S + 32*k] = gamma->list[2*S + 32*k] *
                                    (alpha->list[i] + alpha->list[1 + i]);

         d = 1;
         m = S/2 + (S & 8) ^ 8*((S & 1) ^ d);
         b = d ^ (m & 1) ^ (m & 4)/4 ^ (m & 8)/8;
         gamma->list[1 + 2*S + 32*k] = pTrans(x, d, s) * pTrans(y, b, s);
         i = 2*m + 32*(k-1);
         alpha->list[1 + 2*S + 32*k] = gamma->list[1 + 2*S + 32*k] *
                                    (alpha->list[i] + alpha->list[1 + i]);
         a->list[k] += alpha->list[2*S + 32*k] + alpha->list[1 + 2*S + 32*k];
     }

     for(size_t S = 0; S < 16; S++)
     {
         alpha->list[2*S + 32*k] /=  a->list[k];
         alpha->list[1 + 2*S + 32*k] /= a->list[k];
     }
 }
```

# Calcul des $\beta$ et $\lambda$

```
    for(int k = (n - 1); k > 0; k--)    // Compute the probabilities beta
    {
        for(size_t S = 0; S < 16; S++)
        {
            m = (2*S & 15) + ((S & 8)/8) ^ ((S & 4)/4);
            beta->list[S + 16*k] = beta->list[m + 16*(k + 1)] *
                                   gamma->list[2*m + 32*(k + 1)];

            m = (2*S & 15) + ((S & 8)/8) ^ (1 - (S & 4)/4);
            i = 2*m + 32*(k + 1);
            beta->list[S + 16*k] += beta->list[i / 2] * gamma->list[1 + i];
        }

        for(size_t S = 0; S < 16; S++)
        {
            beta->list[S + 16*k] /= a->list[k + 1];
            lambda->list[2*S + 32*k] = alpha->list[2*S + 32*k] *
                                       beta->list[S + 16*k];
            lambda->list[1 + 2*S + 32*k] = alpha->list[1 + 2*S + 32*k] *
                                       beta->list[S + 16*k];
        }

        tmp[0] = 0.0;
        tmp[1] = 0.0;

        for(size_t S = 0; S < 16; S++)
        {
            tmp[0] += lambda->list[2*S + 32*k];
            tmp[1] += lambda->list[1 + 2*S + 32*k];
        }

        llr->list[k - 1] = log(tmp[1] / tmp[0]);

```

# Création d'un code LDPC

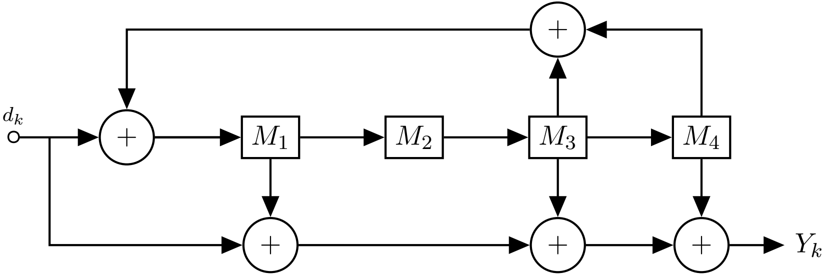
```c
h_matrix * create_base(size_t n, size_t j, size_t k)
{
    size_t m = (n * j) / k;
    h_matrix *res = chm(m, n);
    i_list *perm = cil(n, n);

    // Fill the first horizontal part of the matrix
    for (size_t i = 0; i < n; i++)
    {
        shm(res, 1, (i / k), i);
    }

    // Fill the j-1 other bands
    for (size_t i = 1; i < j; i++)
    {
        permutation(perm);
        for (size_t x = 0; x < n; x++)
        {
            shm(res, 1, ((perm->list[x] + i * n) / k), x);
        }
    }
    return res;
}
```

# Décodage d'un code LDPC

```c
int decode_ldpc_a_basic(a_matrix *mat, h_list *mes, size_t nb_max)
{
    h_list *verif = product_a(mat, mes);
    i_list *count = cil(mes->n, mes->n);
    i_list *max_errors = cil(mes->n, mes->n);
    size_t iter = 0;
    char correct = is_all_nil(verif);
    while (!correct && (iter < nb_max))
    {
        set_all_i_list(count, 0);
        for (size_t i = 0; i < verif->n; i++)
        {
            if (verif->list[i])
            {
                for (size_t j = 0; j < mat->list_n[i]->n; j++)
                {
                    count->list[mat->list_n[i]->list[j]] ++;
                }}}
        // Flip the bits with the most errors
        max_i_list(count, max_errors);
        for (size_t i = 0; i < max_errors->n; i++)
        {
            mes->list[max_errors->list[i]] ^= 1;
        }
        iter ++;
        product_a_in_place(mat, mes, verif);
        correct = is_all_nil(verif);
    }
    // Free used lists
    free_h_list(verif);
    free_i_list(count);
    free_i_list(max_errors);
    return iter;
}
```

# Turbocodes

# Composant Enc




| $d_k$ | $\emptyset$ | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | 0 | 1 | 3 | 6 | 12 | 9 | 3 | 7 | 15 | 15 |
| $Y_k$ | $\emptyset$ | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

# Décodage

# Processus de décodage

Le décodeur reçoit en entrée trois variables réelles pour le code avec $R = \frac{1}{3}$ :

$$x_k = (2.X_k - 1) + a_k$$
$$y_{1,k} = (2.Y_{1,k} - 1) + b_k$$
$$y_{2,k} = (2.Y_{2,k} - 1) + c_k$$

où $a_k, b_k$ et $c_k$ sont des variables aléatoires suivant une loi normale de moyenne nulle et de variance $\sigma^2$

# Principe de décodage

On note $S_k$ l'état de l'encodeur au moment k,
$S_k = (a_k, a_{k-1}, a_{k-2}, a_{k-3})$
$S_0 = S_N = 0$

La sortie du canal fournie à l'entrée du décodeur est la suite $R_1^N = (R_1, \ldots, R_k, \ldots, R_N)$ où $R_k = (x_k, y_{j,k})$

On introduit $\lambda_k^i(m) = \mathbf{P}(X_k = i, S_k = m / R_1^N)$, d'où

$$\mathbf{P}(d_k = i / R_1^N) = \sum_m \lambda_k^i \text{ et } \Lambda(X_k) = \log\left(\frac{\sum_m \lambda_k^1}{\sum_m \lambda_k^0}\right)$$

On introduit des fonctions :

- $\alpha_k^i(m) = \dfrac{\mathbf{P}(X_k = i, S_k = m, R_1^k)}{\mathbf{P}(R_1^k)} . \mathbf{P}(X_k = i, S_k = m / R_1^k)$
- $\beta_k(m) = \dfrac{\mathbf{P}(R_{k+1}^N / S_k = m)}{\mathbf{P}(R_{k+1}^N / R_1^k)}$
- $\gamma_i(R_k, m', m) = \mathbf{P}(X_k = i, R_k, S_k = m / S_{k-1} = m')$

On a de plus :

$\lambda_k^i(m) = \frac{\mathbf{P}(X_k=i,S_k=m,R_1^k,R_{k+1}^N)}{\mathbf{P}(R_1^k,R_{k+1}^N)}$

$= \frac{\mathbf{P}(X_k=i,S_k=m,R_1^k)}{\mathbf{P}(R_1^k)} . \frac{\mathbf{P}(R_{k+1}^N|X_k=i,S_k=m,R_1^k)}{\mathbf{P}(R_{k+1}^N|R_1^k)}$

Ainsi, par indépendance du message,

$\mathbf{P}(R_{k+1}^N | X_k = i, S_k = m, R_1^k) = \mathbf{P}(R_{k+1}^N | S_k = m)$, d'où

$\lambda_k^i(m) = \alpha_k^i(m) . \beta_k(m)$

$$\alpha_k^i(m) = \frac{\mathbf{P}(d_k = i, S_k = m, R_1^{k-1}, R_k)}{\mathbf{P}(R_1^{k-1}, R_k)}$$
$$= \frac{\mathbf{P}(d_k = i, S_k = m, R_k | R_1^{k-1})}{\mathbf{P}(R_k | R_1^{k-1})}$$

$$\mathbf{P}(d_k = i, S_k = m, R_k | R_1^{k-1}) =$$
$$\sum_{m'} \sum_{j=0}^{1} \mathbf{P}(d_k = i, S_k = m, d_{k-1} = j, S_{k-1} = m', R_k | R_1^{k-1})$$
$$= \sum_{m'} \sum_{j=0}^{1} \mathbf{P}(d_{k-1} = j, S_{k-1} = m' | R_1^{k-1}) \times$$
$$\mathbf{P}(d_k = i, S_k = m, R_k | d_{k-1} = j, S_{k-1} = m', R_1^{k-1})$$
$$= \sum_{m'} \sum_{j=0}^{1} \alpha_{k-1}^{j}(m') \gamma_i(R_k, m', m)$$

De même, on a :

$\mathbf{P}(R_k | R_1^{k-1}) = \sum_{m'} \sum_{m} \sum_{i=0}^{1} \sum_{j=0}^{1} \alpha_{k-1}^{j}(m') \gamma_i(R_k, m', m)$, d'où

$$\alpha_k^i(m) = \frac{\sum_{m'} \sum_{j=0}^{1} \alpha_{k-1}^{j}(m') \gamma_i(R_k, m', m)}{\sum_{m'} \sum_{m} \sum_{i=0}^{1} \sum_{j=0}^{1} \alpha_{k-1}^{j}(m') \gamma_i(R_k, m', m)}$$

$$\beta_k(m) = \frac{\sum_{m'} \sum_{i=0}^{1} \mathbf{P}(d_{k+1}{=}i, S_{k+1}{=}m', R_{k+1}, R_{k+2}^N | S_k{=}m)}{\mathbf{P}(R_{k+1}^N | R_1^k)}$$

$$= \frac{\sum_{m'} \sum_{i=0}^{1} \mathbf{P}(R_{k+2}^N | S_{k+1}{=}m')\mathbf{P}(d_{k+1}{=}i, S_{k+1}{=}m', R_{k+1}, | S_k{=}m)}{\mathbf{P}(R_{k+1}^N | R_1^k)}$$

$$= \frac{\sum_{m'} \sum_{i=0}^{1} \beta_{k+1}(m')\gamma_i(R_{k+1}, m, m')}{\mathbf{P}(R_{k+1} | R_1^k)}$$

$$= \frac{\sum_{m'} \sum_{i=0}^{1} \beta_{k+1}(m')\gamma_i(R_{k+1}, m, m')}{\sum_{m'} \sum_{m} \sum_{i=0}^{1} \sum_{j=0}^{1} \alpha_k^j(m')\gamma_i(R_{k+1}, m', m)}$$

$\gamma_i(R_k, m, m') = \mathbf{P}(X_k = i, (x_k, y_k), S_k = m | S_{k-1} = m')$
$= \mathbf{P}(x_k, y_k | X_k = i, S_k = m, S_{k-1} = m')$
$\mathbf{P}(X_k = i | S_k = m, S_{k-1} = m')\mathbf{P}(S_k = m | S_{k-1} = m')$ or,
$\mathbf{P}(X_k = i | S_k = m, S_{k-1} = m') \in \{0, 1\}$ et
$\mathbf{P}(x_k, y_k | X_k = i, S_k = m, S_{k-1} = m') = \mathbf{P}(x_k | X_k = i)\mathbf{P}(y_k | Y_k = j)$, enfin,
$\mathbf{P}(S_k = m | S_{k-1} = m') = \frac{1}{2}$

# Algorithme de calcul

- Étape 0 : On initialise les probabilités,
  $\alpha_0^i(0) = 1, \ \alpha_0^i(m) = 0 \quad \forall m \neq 0$
  $\beta_N(0) = 1, \ \beta_N(m) = 0 \quad \forall m \neq 0$
- Étape 1 : Pour chaque information reçue $R_k$, on calcule $\alpha_k^i(m)$, $\gamma_i(R_k, m', m)$
- Étape 2 : Après la réception du message, on calcule $\beta_k(m)$ puis $\lambda_k^i(m)$ et enfin $\Lambda(X_k)$

# Codes LDPC

# Fonctionnement

# Principe mathématique

- Le codage comme multiplication matricielle dans $\mathbb{F}_2$
- Utilisation de matrices peu denses $A \in \mathcal{M}_n(\mathbb{F}_2)$
- Code (n, j, k)

# Génération d'un code LDPC

Utilisation de la méthode de Robert Gallager pour un code (n, j, k).

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_j \end{pmatrix} \quad A_1 = \begin{pmatrix} 1 & 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \cdots & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 1 \end{pmatrix}$$

$$\forall l \in 2, j, \quad A_i = \left(C_{\sigma(1)} | \dots | C_{\sigma(j)}\right) \text{ où } A_1 = (C_1 | \dots | C_j)$$

La matrice de codage obtenue est alors $G = \begin{pmatrix} A \\ I \end{pmatrix}$ et la matrice de décodage $H = \begin{pmatrix} A & I \end{pmatrix}$

# Codage d'un message

Soit $m = (m_1, \ldots, m_n)$ un message à coder, on a $c = (c_1, \ldots, c_r)$ le message obtenu après codage,

$c = G.m^T$ où $r = \dfrac{nj}{k} + n$,

Le taux de transmition vaut donc $R = \dfrac{k}{k+j}$

Enfin, $H.c^T = 0$

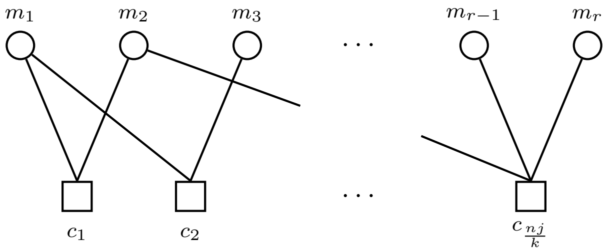
# Décodage

# Représentation de Tanner

Un graphe de Tanner est associé à la matrice de décodage H et comprend :

- $r$ noeuds messagers
- $\frac{n.j}{k}$ noeuds de contrôle

**Noeuds messagers**




**Noeuds de contrôle**