Rapport mini projet Code Correcteur

NDJAYE Ouseynou & BARRY Thierno IB.

L'objectif du mini projet étant de réaliser une application simulant le codage convolutif et le décodage de Viterbi, nous avons conçu BAN'DJO VitSim. Application écrite en java à des fins pédagogiques, effectuant ces taches citées plus haut et utilisant le port série afin d'établir un dialogue entre l'émetteur et le récepteur .

2008 – 2009

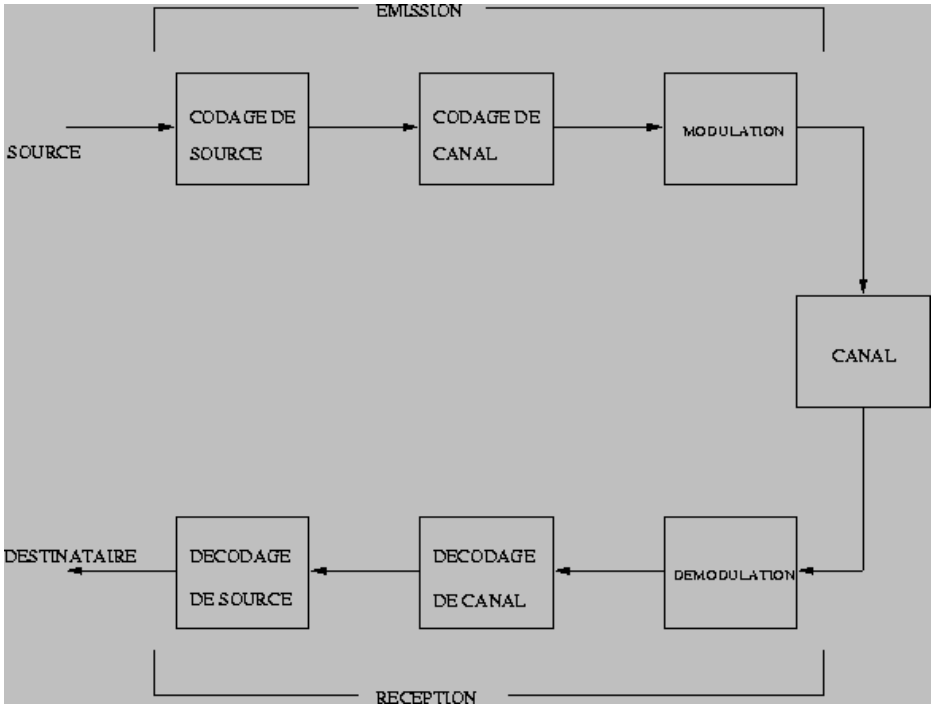
Sommaire :

1. Un bref rappel :

Dans les systèmes de transmission de données, la probabilité d'erreur est fonction du rapport signal à bruit Eb/N0 du canal à l'entrée du récepteur. L'évolution des télécommunications s'accompagne d'une demande toujours plus grande de la qualité de transmission. Il faut pour cela diminuer le taux d'erreur et par conséquent accroître le rapport signal à bruit. L'augmentation de la puissance du signal d'émission et/ou la diminution du facteur de bruit du canal sont des solutions envisageables. Elles engendrent cependant des problèmes de coût ou de technologie. L'autre solution est basée sur l'utilisation des codes correcteurs qui permettent d'augmenter les performances de transmission tout en conservant le meilleurs compromis possible entre la bande passante occupée et la puissance émise.

1.1. Modélisation d'une transmission numérique :

Un système de transmission numérique peut être schématisé de la façon suivante :



Trois opérations sont à effectuer à l'émission.

- Le codage de source permet de minimiser la quantité d'information nécessaire pour la transmission du message. La compression de données est un exemple de codage de source.
- Le codage de canal sert à lutter contre les perturbations apportées par le support de la transmission en remplaçant le message à transmettre par un message moins

vulnérable, en codant ce message par un code convolutif par exemple. L'algorithme de Viterbi est une technique utilisée pour le décodage de ces codes.

- La modulation permet d'adapter les caractéristiques du signal à celles d'un canal. Les modulations les plus couramment utilisées sont de type modulation de fréquence ou modulation de phase.

1.2. Codage de canal

Le principe de base du codage de canal consiste à remplacer le message à transmettre par un message plus long qui contient de la redondance. Sans redondance, chaque donnée du message est indispensable à la compréhension du message entier. Toute erreur dans une partie du message est donc susceptible de changer la signification du message. L'objectif de la redondance est de faire en sorte que les erreurs ne compromettent pas la compréhension globale du message.

Du fait de l'adjonction d'une redondance, le message effectivement transmis est plus long. Un code se caractérise par son rendement R. Si le codeur génère N bits à partir de K bits d'information, le rendement R vaut K/N.

Les données générées par le codeur sont appelées des symboles. Lors du décodage, les symboles reçus peuvent être des bits ou des mots binaires. Dans le premier cas, le système est dit à décision dure, dans le second à décision douce. Un système à décision douce présente de meilleures performances qu'un système à décision dure, mais au détriment d'une complexité plus grande du décodeur de Viterbi.

Il y a deux grandes familles de code :

- le codage en bloc. Le message décomposé en blocs de K bits, est remplacé par un bloc de N bits comprenant directement les K bits d'information et N-K bits de redondance calculés à partir des bits d'information, le codage d'un bloc se faisant indépendamment des précédents.
- le codage convolutif. À K bits d'information, le codeur associe N bits codés, mais contrairement au cas précédent. Le codage d'un bloc de K bits dépend non seulement de ce bloc mais également des blocs précédents.

Les codes convolutifs permettent une correction grossière en atteignant des taux d'erreurs de 10^{-6}. Par contre les codes en blocs permettent une correction plus fine en atteignant pour certaines applications des taux d'erreurs de 10^{-9}. Ces deux codes sont souvent associés pour obtenir de très bons taux d'erreurs.

1.3. Codage convolutif

Chaque bloc de N éléments binaires transmis (aussi appelés symboles) dépend non seulement du bloc de K éléments présents à son entrée mais aussi des m blocs précédents. Le codeur est constitué de m registres à décalage de K éléments binaires. Une logique combinatoire constituée de N générateurs linéaires de fonctions algébriques génère les blocs de N éléments binaires fournis par le codeur.

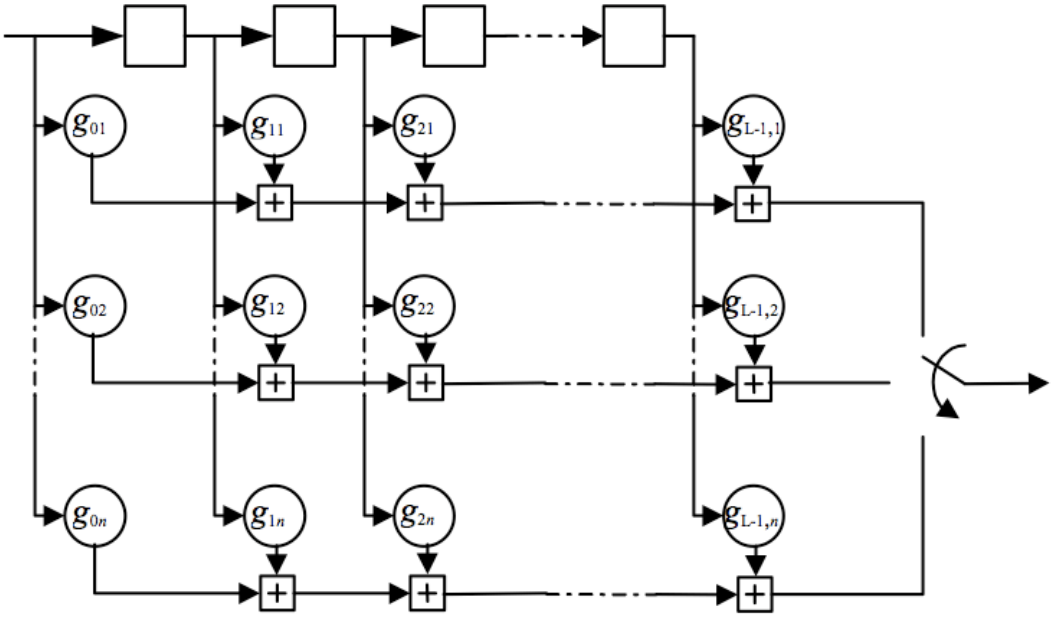


Fig. 1 : Codeur convolutif de rendement 1/n

La longueur du registre à décalage (m+1) définit la longueur de contrainte. Ce paramètre correspond au degré de mémoire introduit sur les bits de données. Plus ce paramètre est grand, plus le code est puissant, mais plus le décodeur est complexe. La logique combinatoire est caractérisée par ses polynômes générateurs qui explicitent les positions du registre à décalage prises en compte dans le calcul des symboles. Les principaux codes utilisés sont de la forme R=1/N (K=1). Ce type de code présente un rendement assez faible. Des codes de rendement plus important en sont dérivés; ces codes sont dits poinçonnés car ils sont obtenus en supprimant certains symboles générés par le code 1/N.

Le terme « convolutif » s'applique à ce type de codes parce que la suite de bits codés S peut s'exprimer comme le résultat de la convolution de la suite de bits d'information e par les coefficients g. En effet, puisque le code est linéaire, nous avons : S = e.G. En observant la forme particulière de G, les n bits en sortie du codeur à l'instant t, correspondant à une entrée, s'écrivent :

$$S_t = \sum_{u=\max(1,t-L+1)}^{t} e_k g_{u-k}$$

La quantité L est appelée longueur de contrainte du code convolutif.

La complexité du codeur est indépendante de la longueur de la séquence de bits d'information émise et ne dépend que de la longueur de contrainte du code.

1.4. Algorithmes de décodage:

Nous décrivons dans ce paragraphe deux algorithmes de décodage applicables aux codes convolutifs. Ils effectuent leurs opérations sur un treillis, qui représente, dans le cas du décodage des codes convolutifs, la structure du code. Ils sont cependant applicables plus

généralement à la recherche du chemin dans un treillis qui minimise un critère de distance additive. Ils sont donc aussi applicables au décodage des codes en bloc (par l'intermédiaire de leur représentation en treillis), à celui des modulations codées en treillis, où à la détection au maximum de vraisemblance sur un canal introduisant de l'interférence entre symboles, problème qui correspond à l'estimation des symboles d'un modèle de chaîne de Markov cachée, représentable lui aussi par un treillis.

1.5. Algorithme de Viterbi :

L'algorithme de Viterbi, comme tous les algorithmes de décodage des codes convolutifs, permet d'estimer la séquence émise en mesurant l'écart du chemin associé aux symboles reçus par rapport à chaque chemin possible. Le chemin ayant le plus faible écart sera considéré comme le plus représentatif de la séquence émise réellement et permettra de la reconstituer.

Principe :

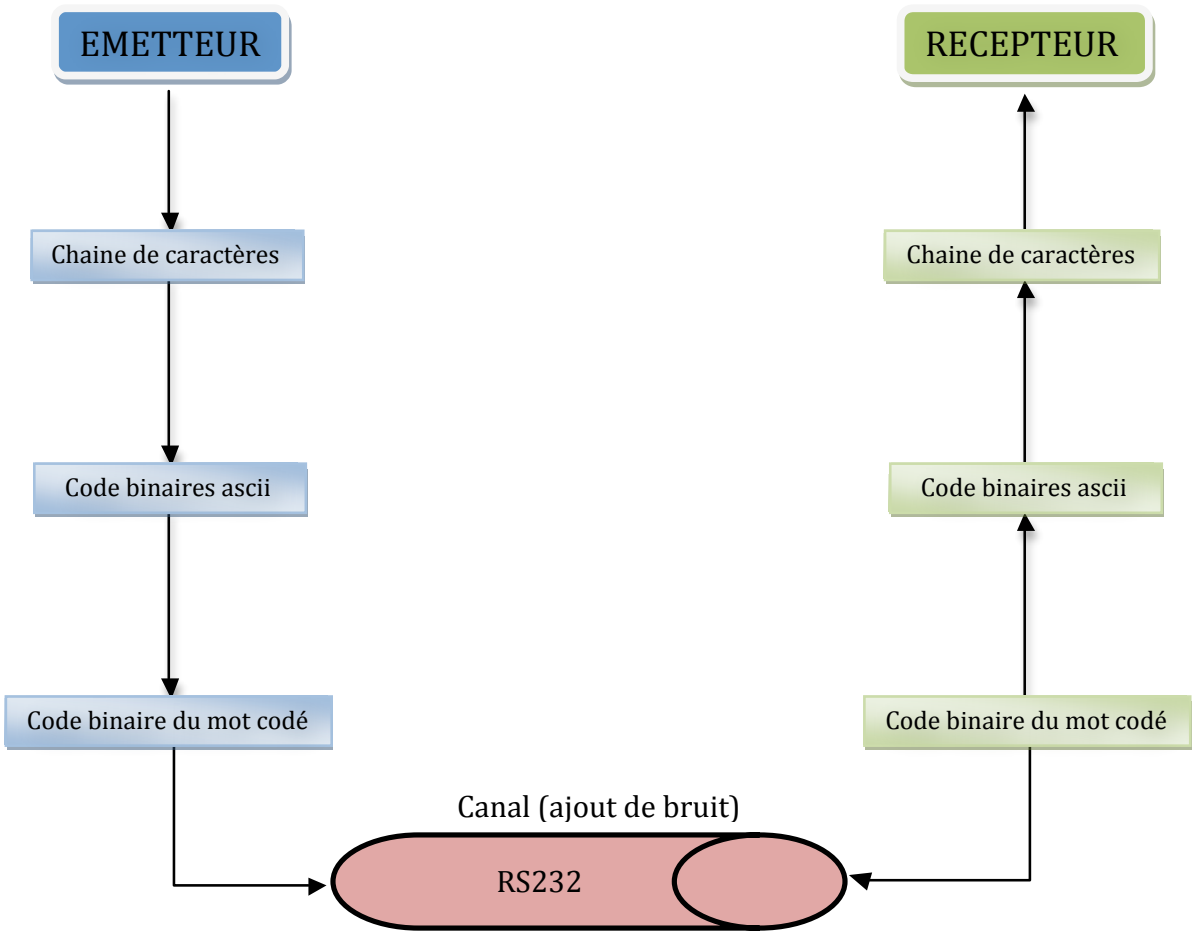
A chaque fois qu'un groupe de n éléments binaires arrive, on examine toutes les branches possibles du treillis. On calcule la distance de Hamming entre les e.b. affectés aux branches et les e.b. reçus. On ne garde que les branches donnant lieu à une distance minimale (ces branches forment le chemin survivant) et on affecte l'extrémité du chemin survivant d'une métrique égale à la somme de la métrique précédente et de la distance de la branche retenue.

2. Le mini projet :

Dans ce mini projet, il nous a été demandé de réaliser un simulateur effectuant le codage convolutif et décodage d'un code par l'algorithme de Viterbi. Tout en donnant la possibilité à un utilisateur d'introduire du bruit afin de vérifier les capacités correctrices du codage convolutif. Le simulateur devra aussi être capable d'ouvrir une connexion RS232 pour envoyer des données. Nous avons opté de développer cette solution en java.

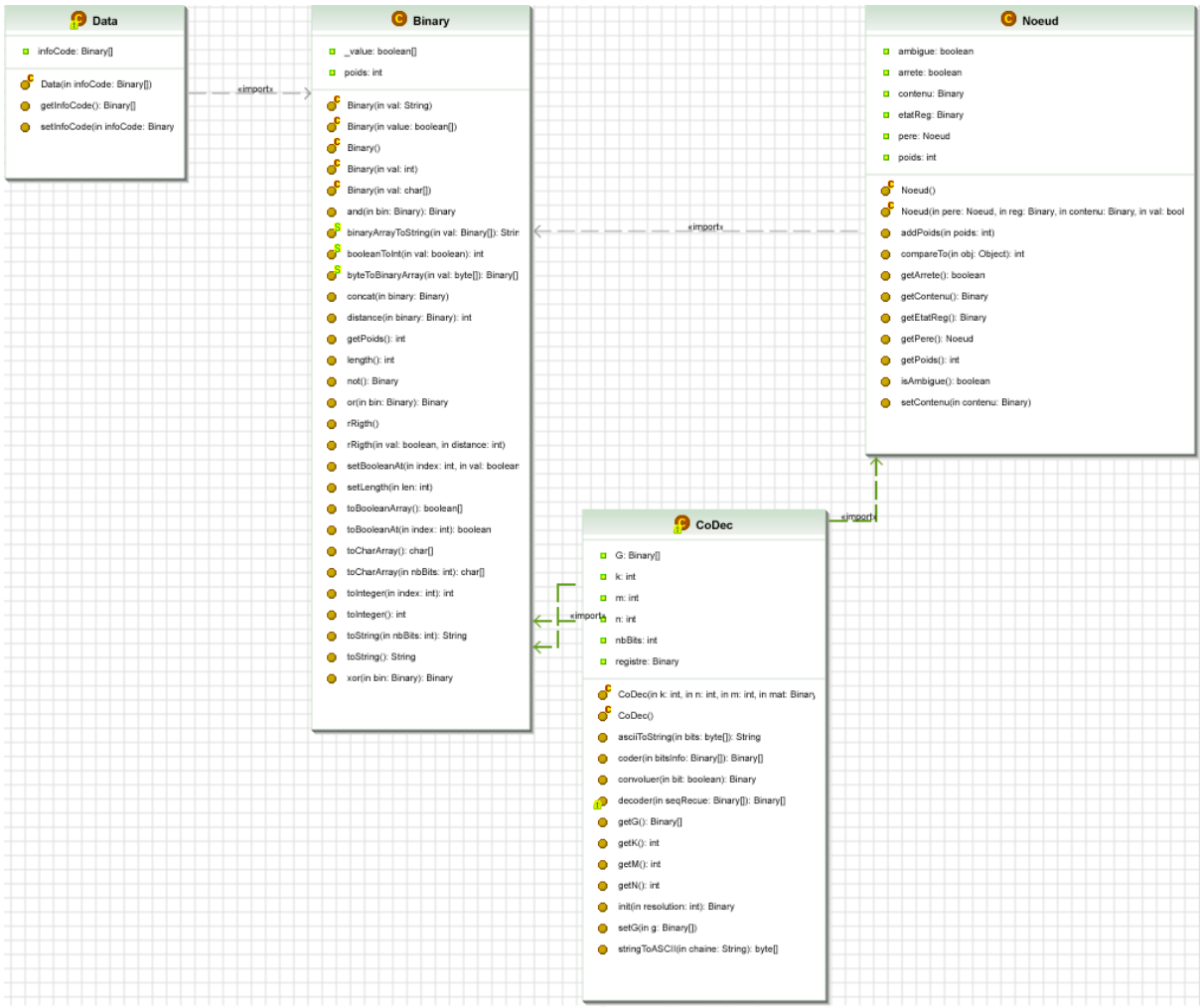
2.1. Description de BAN'DJO:

Afin de respecter le modèle d'une chaine de transmission, le simulateur BAN'DJO est structuré de la façon suivante :



Dans un premier temps, nous récupèrerons du côté émetteur une chaine de caractère que nous allons transformer en ASCII afin d'obtenir un suite binaire. Cette suite sera ensuite codée par un codeur convolutif avant d'être introduit dans le canal RS232 (bruité ou pas). A la réception, il sera fait les opérations inverses pour restituer une chaine de caractère.

2.2. Modélisation UML :



2.3. Description des différentes class java :

Dans un soucis de flexibilité, nous avons développé certaines class qui existait néanmoins dans les bibliothèques java.

- La class Binary : Cette dernière bien qu'existant dans la librairie java a été revue et adaptée à nos besoins, afin de nous faciliter la tache sur les suites de bit. Elle définit une suite binaire et possède un certain nombre de méthode pour les opérations binaires.
- La class Nœud : Dans cette class, nous avons essayé de modéliser un nœud de Viterbi. Avec un poids, un nœud père et aussi une branche qui mène a lui.
- La class Codec : Cette class définit quant à elle toutes les opérations de codage convolutif et décodage de Viterbi.

```
/**
 * @author N'DIAYE Ouseynou & BARRY Ibrahima
 * codeur convolutionnel et decodeur de viterbi.
 *
 * Bugs connus à ce jour:
 *  - Décodage du "?": Le décodeur rajoute un 0 en fin de séquence décodée.
 *                Le "?" est codé sur 6 bits et non 7.
 */
public class CoDec {
```

```
    /* ************************************************************************
     * k:       est le nombre de bits en entrée du codeur.
     * m:       est l'ordre du codeur. Par défaut, il vaut 3.
     * n:       est le nombre de bits en sortie du codeur ou nombre des matrices g.
     * nbBits:  nombre de bits pour chaque élément (ASCII est codé sur 7 bits).
     * G:    est un tablau des matrices de convolution.
     *          Par défaut (111; 101) = (7; 5)
     * ************************************************************************
     */

    private int k;
    private int m;
    private int n;
    private int nbBits = 7;
    private Binary[] G = {new Binary(7), new Binary(5)};
    private Binary registre;

    public CoDec(){
        k = 1;
        n = G.length;
        m = 3;
        registre = this.init(m); // initialisation du registre
    }
    public CoDec(int k, int n, int m, Binary[] mat){
        //if((k*(m-1)) > 127) k=Math.round(127/(m-1));
        this.k = k;
        this.m = m;
        this.G = mat;
        this.n = this.G.length;
        this.registre = this.init(m); // initialisation du registre
    }

    /* ********************************** *
     * Définition des méthodes de la class *
     * ********************************** */

    public byte[] stringToASCII(String chaine) throws
UnsupportedEncodingException{

        return chaine.getBytes("ASCII");
    }

    public String asciiToString(byte[] bits){

        return new String(bits);
    }

   public Binary convoluer(boolean bit){
        int c = 0, code=0;
        registre.rRigth(bit, 1);
        for(int j=0; j<n; j++){
            c=0;
            for(int i=0; i<m; i++){
                c ^= registre.toBooleanAt(i) ? G[j].toInteger(i):0;
            }
            code += (c<<(n-j-1));
        }
        return new Binary(code);
    }
```

Convertit une chaine en un tableau de byte (ASCII)

Convertit un tableau de byte (ASCII) en chaine (String)

Effectue un produit de convolution entre un bit et le registre

```
    /*
     * Codeur convolutionel
     */
    public Binary[] coder(Binary[] bitsInfo) {
        Vector<Binary> bitsCode = new Vector<Binary>();
        registre = init(m);
        for(int cpt=0; cpt<bitsInfo.length; cpt++){

            if(bitsInfo[cpt].length()<nbBits)
                bitsInfo[cpt] = new Binary(bitsInfo[cpt].toString(nbBits));

            for(int cptBit=0; cptBit < bitsInfo[cpt].length(); cptBit++){
                bitsCode.add(convoluer(bitsInfo[cpt].toBooleanAt(cptBit)));
            }
        }
        return bitsCode.toArray(new Binary[bitsCode.size()]);
    }


    /*
     * Décodeur de Viterbi
     */
    public Binary [] decoder(Binary[] seqRecue){
        ArrayList<Noeud>[] instants = new ArrayList[seqRecue.length];
        Binary contenu = new Binary();
        Noeud un, zero;

        // on réinitialise les registres à 00...0
        registre = this.init(m);
        contenu = this.init(n);
        // création de la racine soit l'état zero
        Noeud racine = new Noeud(null, this.registre, contenu, false);
        // création des deux fils de la racine
        instants[0] = new ArrayList<Noeud>();

        this.registre = new Binary(racine.getEtatReg().toBooleanArray());
      contenu=this.convoluer(false);
        zero    = new Noeud(racine, new
Binary(this.registre.toBooleanArray()), contenu, false);
        zero.addPoids(contenu.distance(seqRecue[0]));

        this.registre = new Binary(racine.getEtatReg().toBooleanArray());
        contenu=this.convoluer(true);
        un    = new Noeud(racine, new Binary(this.registre.toBooleanArray()),
contenu, true);
        un.addPoids(contenu.distance(seqRecue[0]));

        instants[0].add(zero);
        instants[0].add(un);

        // création des fils pour les prochains noeuds
        Iterator<Noeud> it;
        for(int cpt=1; cpt<seqRecue.length; cpt++){
            instants[cpt] = new ArrayList<Noeud>();

            it=instants[cpt-1].iterator();

            while(it.hasNext()){
                Noeud pere = it.next();

                this.registre = new Binary(pere.getEtatReg().toBooleanArray());
```

Fonction de codage convolutif, qui retour des mots codes

Fonction de décodage de Viterbi

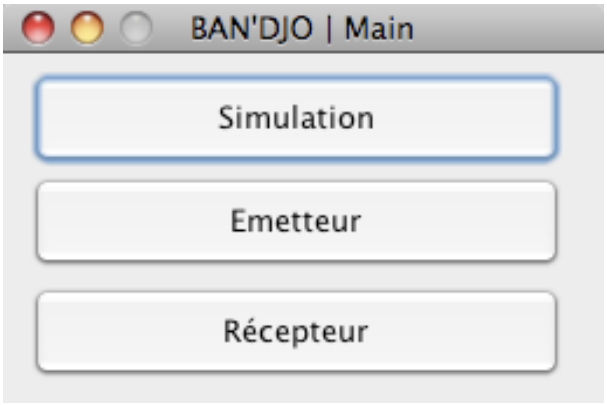
```
            contenu=this.convoluer(false);
            zero    = new Noeud(pere, new
Binary(this.registre.toBooleanArray()), contenu, false);
            zero.addPoids(contenu.distance(seqRecue[cpt]));

            this.registre = new Binary(pere.getEtatReg().toBooleanArray());
            contenu=this.convoluer(true);
            un    = new Noeud(pere, new
Binary(this.registre.toBooleanArray()), contenu, true);
            un.addPoids(contenu.distance(seqRecue[cpt]));

            instants[cpt].add(zero);
            instants[cpt].add(un);
         }

         /*
         * Nous trions la list à chaque instant afin d'éliminer les Noeuds
         * de poids élevés (distance Haming).
         * Si des Noeuds sont considérés comme ambigue, ils ne seront pas
supprimés afin
         * de décider ultérieurment.
         */
         Collections.sort(instants[cpt]);

         // il existe une ambiguité

         if(instants[cpt].get(0).isAmbigue())
            // si ambigue, on supprime tous ceux qui ne le sont pas.
            // car seul les Noeuds du haut de la list sont de poids faible
            for(int i=1; i<instants[cpt].size(); ){
               Noeud tmp = instants[cpt].get(i);
               if(!tmp.isAmbigue())
                  instants[cpt].remove(i);
               else i++;
            }
         else
            // si aucune ambiguité, on supprime tous les Noeuds de poids élevés.
            for(int i=1; i<instants[cpt].size(); ){
               instants[cpt].remove(i);
            }
      }
      Noeud tmp = instants[seqRecue.length-1].get(0);

      boolean[] cheminSurvivant = new boolean[seqRecue.length];
      for(int i=seqRecue.length-1; i>=0; i--){
         cheminSurvivant[i] = tmp.getArrete();
         tmp = tmp.getPere();
      }
      Vector<Binary> infoDecode = new Vector<Binary>();
      String chaine = "";
      for(int i=0; i<cheminSurvivant.length; i++){
         chaine += cheminSurvivant[i] ? "1":"0";
         if((i!=0) && ((i+1)%7==0)){
            infoDecode.add(new Binary(chaine));
            chaine="";
         }
      }


      return infoDecode.toArray(new Binary[infoDecode.size()]);
   }
}
```

- La class Data : Cette class est sérializable, c'est-à-dire qu'on peu l'échanger sur le réseau. De ce fait, c'est un objet de type Data qui sera envoyé sur le port série.

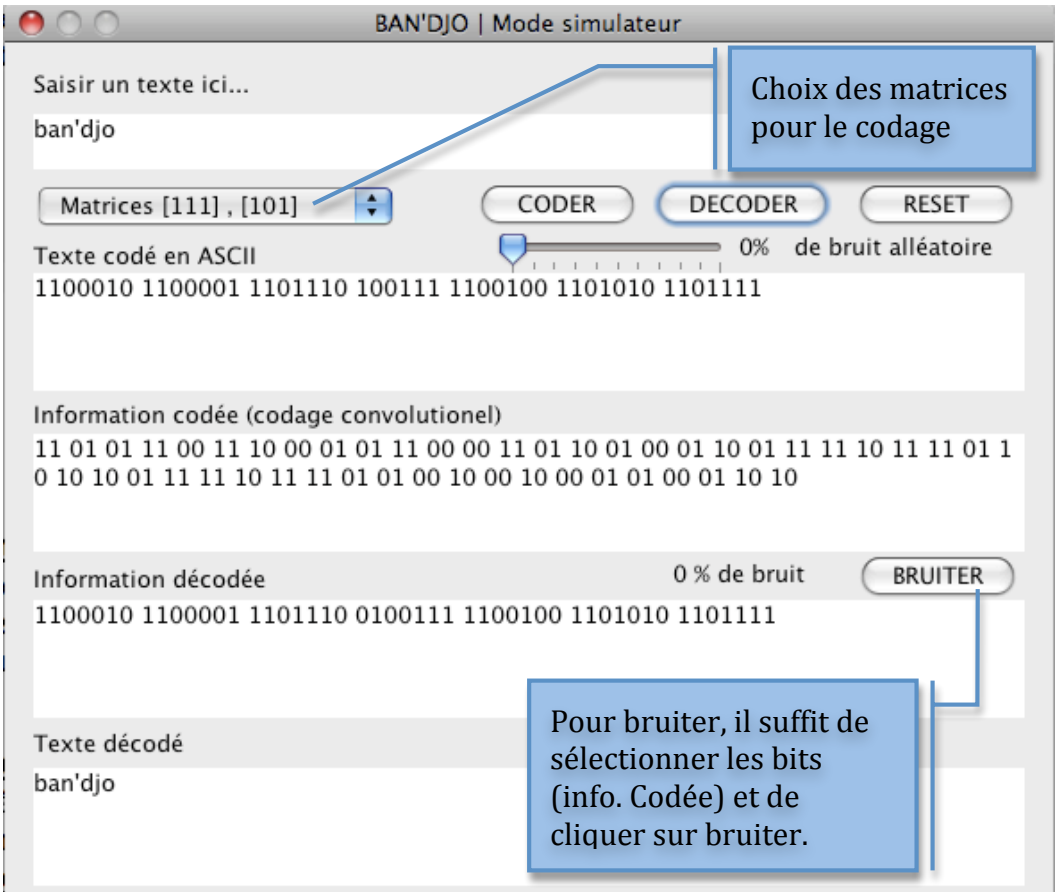
NB : L'application utilise aussi d'autres classes que nous avons écrites, mais qui ne sont pas dans ce diagramme de classe, car nous avons estimé qu'elles n'étais pas primordiales comme les classes « cœur » de BAN'DJO.

3. Utilisation de BAN'DJO :

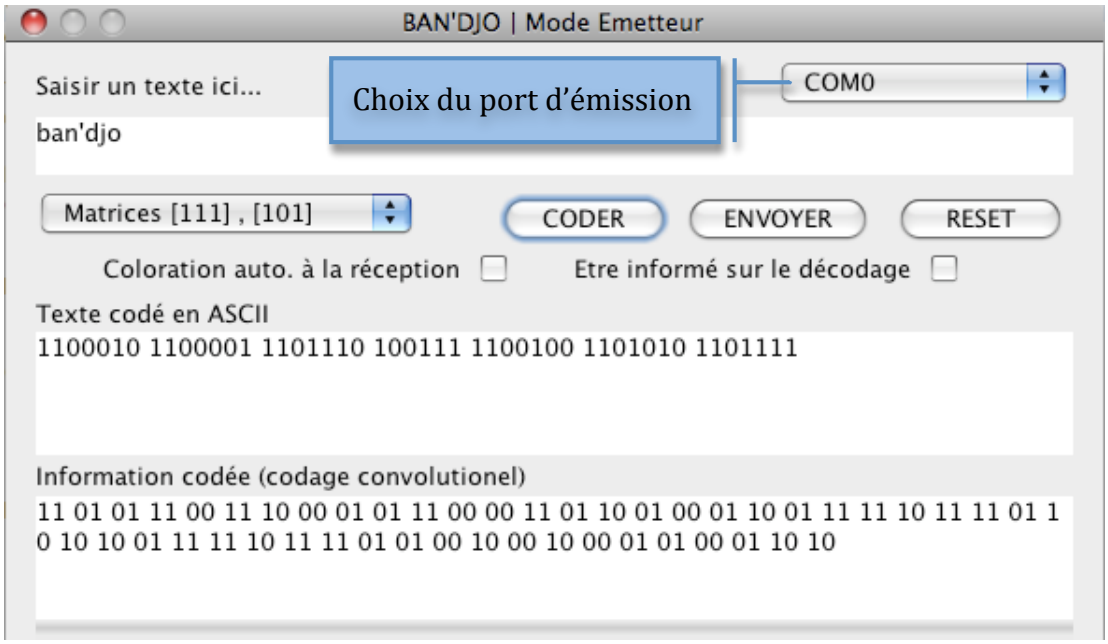
BAN'DJO étant un simulateur, nous avons voulus resté aussi près que possible de la réalité. En effet le logiciel fonctionne suivant trois modes, ayant chacun des caractéristiques particulières.



a. **Mode simulation :** Dans ce mode, nous n'effectuons pas de transmission RS232. Toute fois, nous pouvons coder et décoder. Ce mode est très efficace si on ne possède pas de port série ou simplement si on veut faire des études sur la convolution.



b. **Mode émetteur :** Ici, nous ne pouvons que coder et envoyer les mots codes vers un récepteur sur le port série. Nécessite au préalable l'ouverture d'une connexion RS232.



c. **Mode récepteur:** Nous nous comportons dans ce cas comme un récepteur classique. On se contente de décoder par l'algorithme de Viterbi.