



IMT Atlantique – Ingénieur
spécialité informatique, réseaux,
télécommunications (FIP)

Rapport SIT 213 – Atelier logiciel : simulation d'un système de transmission

Étudiants	Encadrants
<p>Jordan BAUMARD Roman GUERRY Pol GUILLOU Mathis MAQUENNE Maxime PERE</p> <p>Étudiants FIP 2A – IMT Atlantique</p>	<p>Julien MALLET Enseignant chercheur au département informatique – IMT Atlantique</p> <p>François-Xavier SOCHELEAU Enseignant chercheur au département MEE – IMT Atlantique</p>

À destination de l'équipe pédagogique FIP

5 Septembre 2024 – 27 Octobre 2024

Table des matières

I. Introduction	8
II. Itération 1 : Transmission back-to-back	9
2.1. Cahier des charges	9
2.2. Ajout des composants au système	10
2.2.1. Création de la classe SourceFixe	10
2.2.2. Création de la classe SourceAleatoire.....	11
2.2.3. Création de la classe TransmetteurParfait.....	12
2.2.4. Création de la classe DestinationFinale	13
2.3. Mise en fonctionnement du système	14
2.3.1. Instanciation des composants et connexions avec les sondes.....	14
2.3.2. Émission du message par la source	15
2.3.3. Gestion du taux d'erreur binaire	15
2.4. Automatisation du logiciel via des scripts	16
2.4.1. Script compile	16
2.4.2. Script genDoc	17
2.4.3. Script cleanAll	17
2.4.4. Script genDeliverable	18
2.5. Tests et résultats de la première itération	19
2.5.1. Script simulateur	19
2.5.2. Script runTests	19
2.5.3. Vérification de l'itération 1 avec le script <i>Deploiement</i>	21
III. Itération 2 : Transmission analogique non bruitée	22
3.1. Cahier des charges	22
3.2. Implémentation des fonctionnalités	23
3.2.2. Fonctionnement de l'émetteur et du récepteur	23
3.2.3. Création de classes dédiées à la seconde étape	24
3.2.4. Connexions des sondes à la chaîne de transmission.....	25
3.2.3. Intégration des nouveaux paramètres à la commande permettant d'utiliser le logiciel	25
3.3. Résultats obtenus	26
3.3.1. Simulation d'un signal NRZ binaire	26
3.3.2. Simulation d'un signal RZ binaire	27
3.3.3. Simulation d'un signal NRZT binaire	27
3.4. Tests effectués	29

3.4.1. Tests de la classe Emettre	29
3.4.2. Tests de la classe Recepteur	29
3.4.3. Tests de la classe SourceFixe	30
3.4.4. Tests de la classe SourceAleatoire	31
3.4.5. Tests de la classe TransmetteurParfait	31
3.4.6. Tests de la classe DestinationFinale.....	32
3.4.7. Couverture du code Java avec Emma	32
3.4.8. Modification du script runTests	33
3.5. Bilan de l'itération	34
IV. Itération 3 : Transmission non idéal avec canal bruité de type « gaussien »	35
4.1. Cahier des charges	35
4.2. Implémentation des fonctionnalités	37
4.2.1. Création de la classe TransmetteurGaussien	37
4.2.2. Nouvelle méthode de réception du signal bruité	37
4.2.3. Intégration de l'ajout de bruit à la commande permettant d'utiliser le logiciel	38
4.3. Résultats obtenus	39
4.3.1. Simulations avec les paramètres par défaut.....	39
4.3.2. Modification de la taille du message	40
4.3.2. Comparaison entre la puissance de bruit obtenue et la variance	41
4.3.3. Mise en relation du TEB et du SNR.....	41
4.3.4. Comparaison avec les études théoriques	42
4.3.5. Vérification du type de bruit ajouté.....	43
4.4. Tests effectués	44
4.4.1. Tests de la classe TransmetteurGaussien.....	44
4.4.2. Tests du TEB	45
4.4.3. Modification du script runTests	45
4.5. Bilan de l'itération	46
V. Itération 4 : Transmission analogique avec un canal bruité à trajets multiples	47
5.1. Cahier des charges	47
5.2. Implémentation des fonctionnalités	49
5.2.1. Création de la classe TransmetteurMultiTrajets.....	49
5.2.2. Réflexion sur la manière de réceptionner le signal.....	49
5.2.3. Intégration de l'ajout des trajets à la commande permettant d'utiliser le logiciel	50
5.3. Résultats obtenus	51

5.3.1. Résultats obtenus grâce aux sondes	51
5.3.2. Modification du nombre de trajets ajoutés	53
5.3.3. Modification du retard pour chaque trajet	54
5.3.4. Modification de l'amplitude pour chaque trajet	55
5.4. Tests effectués	57
5.4.1. Tests de la classe TransmetteurMultiTrajets	57
5.4.2. Modification du script runTests	58
5.5. Bilan de l'itération	59
VI. Itération 5 : Codage de canal	60
6.1. Cahier des charges	60
6.2. Implémentation des fonctionnalités	61
6.2.1. Création de la classe Codeur	61
6.2.2. Création de la classe décodeur.....	61
6.2.3. Intégration de l'utilisation d'un codage de canal à la commande permettant d'utiliser le logiciel	62
6.3. Résultats obtenus	63
6.3.1. Résultats obtenus sur des tests unitaires grâce à l'IDE.....	63
6.3.2. Efficacité du codeur dans un environnement bruité	64
6.4. Tests effectués	66
6.4.1. Tests des classe Codeur et Decodeur	66
6.4.2. Modification du script runTests	67
6.5. Bilan de l'itération	68

Figure 1 – Modélisation de la chaîne de transmission à l'étape 1	9
Figure 2 – Diagramme de classe correspondant à l'itération 1	10
Figure 3 – Code source de la classe SourceFixe	11
Figure 4 – Code source du premier constructeur de la classe SourceAleatoire	11
Figure 5 – Code source du deuxième constructeur de la classe SourceAleatoire	12
Figure 6 – Code source de la méthode recevoir() provenant de la classe TransmetteurParfait	12
Figure 7 – Code source de la méthode émettre() provenant de la classe TransmetteurParfait	13
Figure 8 – Code source de la classe DestinationFinale	13
Figure 9 – Constructeur de la classe Simulateur	14
Figure 10 – Méthode execute() présente dans la classe Simulateur	15
Figure 11 – Méthode calculTauxErreurBinaire() présente dans la classe Simulateur	15
Figure 12 – Code source du script compile	16
Figure 13 – Résultat du script compile	16
Figure 14 – Code source du script genDoc	17
Figure 15 - Résultat du script genDoc	17
Figure 16 – Code source du script cleanAll	18
Figure 17 - Résultat du script cleanAll	18
Figure 18 – Code source du script genDeliverable	18
Figure 19 – Code source du script Simulateur	19
Figure 20 – Code source du script runTests	20
Figure 21 – Résultats du scripts runTests	20
Figure 22 – Données remontées par les deux sondes	21
Figure 23 - Exécution du script Deploiement	21
Figure 24 - Modélisation de la chaîne de transmission à l'étape 2	22
Figure 25 - Fonctionnement du bloc émetteur	23
Figure 26 - Présentation des codes binaires en ligne NRZ, RZ et NRZT	23
Figure 27 - Diagramme de classes lié à l'itération 2	24
Figure 28 - Code permettant la connexion des sondes à notre système	25
Figure 29 - Commande permettant la simulation d'un signal NRZ binaire	26
Figure 30 - Résultats du signal NRZ binaire	26
Figure 31 - Commande permettant la simulation d'un signal RZ binaire	27
Figure 32 - Résultats du signal RZ binaire	27

Figure 33 - Valeur du signal analogique capté à la sortie de l'émetteur	27
Figure 34 - Commande permettant la simulation d'un signal RZ binaire	28
Figure 35 - Résultats du signal NRZT binaire	28
Figure 36 - Résultat des tests pour la classe Emettre.....	29
Figure 37 - Résultat des tests pour la classe Emettre.....	29
Figure 38 - Résultat des tests pour la classe SourceFixe	30
Figure 39 - Résultat des tests pour la classe SourceAleatoire	31
Figure 40 - Résultat des tests pour la classe TransmetteurParfait	31
Figure 41 - Résultat des tests pour la classe DestinationFinale	32
Figure 42 - Quantité de notre code couvert grâce aux tests JUnit	33
Figure 43 - Modification du script runTests pour l'itération 2	33
Figure 44 - Modélisation de la chaîne de transmission à l'étape 3	35
Figure 45 - Exemple de résultats attendus à la fin de la troisième étape pour un SNR valant 10 dB	36
Figure 46 - Diagramme de classe lié à l'itération 3	37
Figure 47 - Méthode de réception d'un signal bruité	38
Figure 48 - Simulation pour un code NRZT avec un SNR de 15 dB.....	39
Figure 49 – TEB et SNR pour les codages binaires NRZ, NRZT et RZ	40
Figure 50 – Simulation pour un code RZ avec un SNR de 9 dB et un message émis de 10000 booléens	40
Figure 51 - TEB et SNR pour un message de 10000 bits utilisant le code RZ binaire.....	40
Figure 52 - Corrélation entre la variance et la puissance moyenne du bruit	41
Figure 53 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (pratique)	41
Figure 54 - Graphe montrant le TEB en fonction du SNR par bit (E_bN_0)dB pour le code binaire NRZ (théorie et pratique)	42
Figure 55 - Histogramme des valeurs du bruit blanc gaussien additif ajouté dans le canal de transmission	43
Figure 56 - Résultats des tests pour la classe TransmetteurGaussien	44
Figure 57 - Couverture du package transmetteurs grâce à l'outil Emma	44
Figure 58 - Résultat de la classe TebTest.....	45
Figure 59 - Modification du script runTests pour l'itération 3	45
Figure 60 - Modélisation de la chaîne de transmission à l'étape 4	47
Figure 61 - Opération provoquant l'effet de multi-trajets sur un signal entrant.....	47
Figure 62 - Canal de transmission pour l'étape 4	49

Figure 63 - Résultats retournés par les sondes pour la première simulation de l'itération 4	51
Figure 64 - Résultats retournés par les sondes pour la seconde simulation de l'itération 4	52
Figure 65 - Paramètres permettant de lancer la troisième simulation de l'itération 4	52
Figure 66 - Résultats retournés par les sondes pour la troisième simulation de l'itération 4	53
Figure 67 - Graphique affichant le TEB en fonction du nombre de trajets indirects	54
Figure 68 - Graphique affichant le TEB, avec l'augmentation des retards entre chaque simulation	55
Figure 69 - Graphique affichant le TEB, avec l'augmentation de l'amplitude des TI entre chaque simulation	56
Figure 70 - Résultats des tests pour la classe TransmetteurGaussien	57
Figure 71 - Couverture du package transmetteur après avoir ajouté la classe TransmetteurMultiTrajets	57
Figure 72 - Modification du script runTests pour l'itération 4	58
Figure 73 - Modélisation de la chaîne de transmission à l'étape 5	60
Figure 74 - Exemple d'une erreur de transmission malgré le codage canal	60
Figure 75 - Zoom sur le codeur	61
Figure 76 - Zoom sur le décodeur	61
Figure 77 – Paramètres utilisés pour la première simulation	63
Figure 78 - TEB en fonction du SNR par bit avec et sans codage canal pour le code NRZ binaire	64
Figure 79 - Résultats des tests pour les classes Codeur et Decodeur	66
Figure 80 - Couverture du package codage après avoir ajouté les classes Codeur et Decodeur	66
Figure 81 - Modification du script runTests pour l'itération 5	67

I. Introduction

II. Itération 1 : Transmission back-to-back

2.1. Cahier des charges

Au cours de ce projet, nous simulons différentes chaînes de transmission grâce au langage de programmation JAVA. Cette première itération a pour objectif de mettre en place la chaîne de transmission la plus simple possible, qui est représentée par la figure ci-dessous.

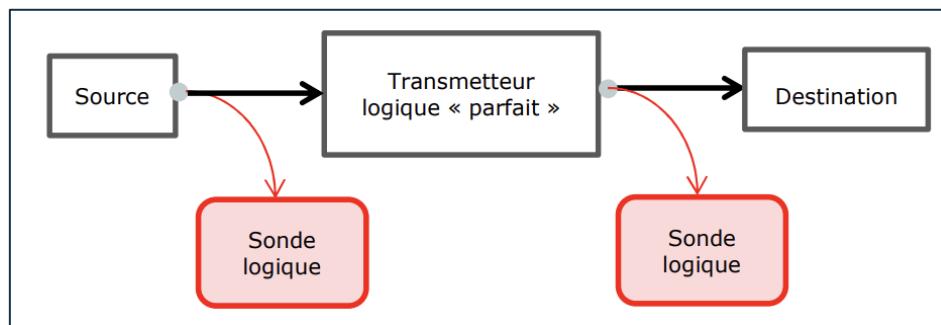


Figure 1 – Modélisation de la chaîne de transmission à l'étape 1

Cette première chaîne de transmission est composée des trois blocs suivants :

- Une **source** qui émet une séquence composée de '0' et de '1'. Cette séquence peut être fixe ou aléatoire ;
- Un **transmetteur logique « parfait »** qui réceptionne la séquence émise par la source et qui ne fait que la retransmettre vers la fin de la chaîne. Ce transmetteur logique a comme particularité d'être parfait, c'est-à-dire qu'aucune erreur ne se glisse dans la séquence booléenne entre la réception et l'émission. C'est donc le cas idéal que nous traitons durant cette première itération ;
- Enfin, la **destination** ne fait que recevoir le signal du transmetteur auquel elle est reliée.

Pour vérifier que le message envoyé est le même que le message reçu, nous plaçons deux sondes sur cette chaîne de transmission. La première en sortie de la source pour mesurer le signal émis au début de la chaîne. La seconde est placée juste après le transmetteur logique. Puisque ce dernier est connecté directement à la destination, cela revient à mesurer le signal en sortie de chaîne.

Une fois que nous aurons obtenu les deux séquences booléennes perçues par les sondes, il nous suffira de les comparer pour connaître le taux d'erreur binaire (TEB) de cette première chaîne de transmission. Sachant que nous sommes dans un cas idéal avec un transmetteur logique parfait, le TEB devrait être nul pour chacun des tests effectués sur ce système.

Nous avons divisé le travail de cette première itération en quatre étapes. Nous commencerons par découvrir le code source qui nous est fourni, auquel nous ajouterons plusieurs composants. Nous les connecterons ensuite au reste de la chaîne et nous y ajouterons les sondes présentées ci-dessus. Afin de répondre aux attentes du client, nous automatiserons le logiciel afin de faciliter son utilisation. Enfin, nous effectuerons une série de test sur le système conçu afin de vérifier son bon fonctionnement.

2.2. Ajout des composants au système

Bien qu'au fil des itérations, la chaîne de transmission sera modifiée, une base commune sera présente à chaque étape. En effet, le principe global du système reste le même : envoyer une suite d'informations qui passe par différents composants avant d'atteindre la destination de la chaîne. Par conséquent, la fondation de notre code JAVA sera lui aussi le même, au fur et à mesure des étapes. Cette base est composée de quatre classes abstraites qui feront office de modèle pour les classes que nous ajouterons par la suite. Cette première itération nous demande de créer quatre classes supplémentaires. Ces classes sont représentées en jaune sur la figure ci-dessous.

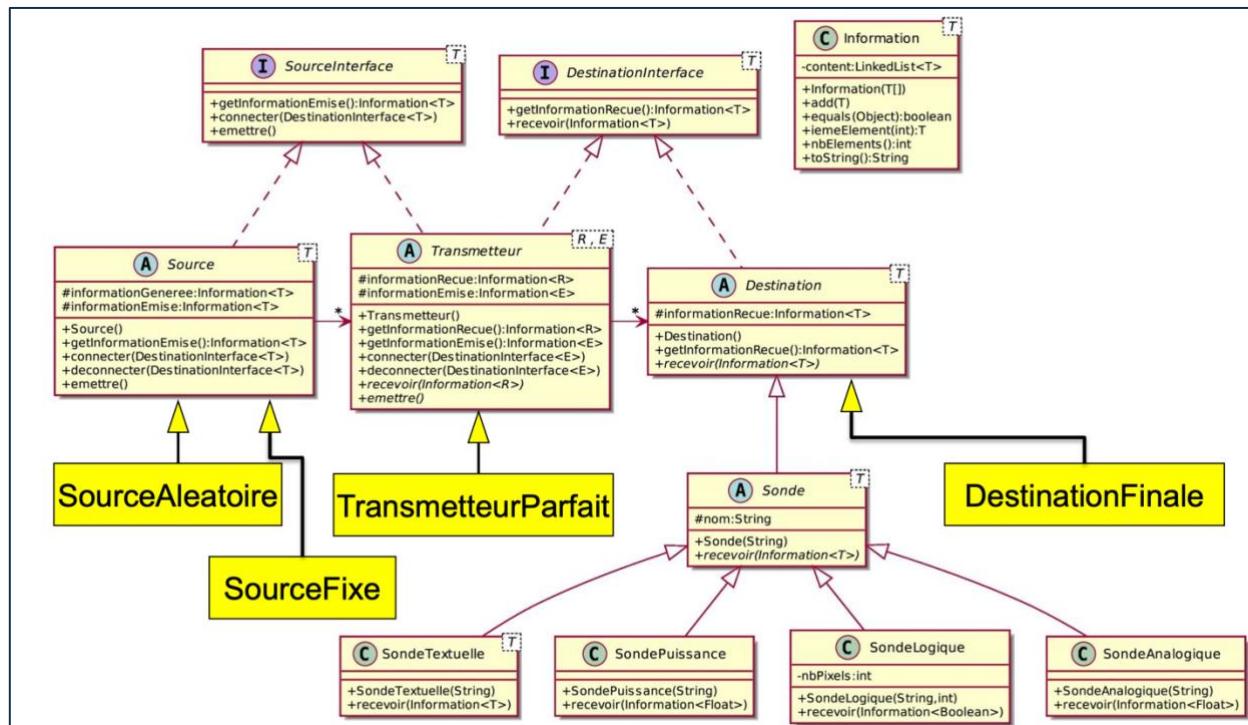


Figure 2 – Diagramme de classe correspondant à l'itération 1

2.2.1. Création de la classe SourceFixe

La première classe dont nous nous occupons est **SourceFixe**. Comme le montre la figure 2, cette classe hérite de la classe abstraite **Source**. Cette classe est appelée quand l'utilisateur donne en entrée du système la séquence booléenne qui est à transmettre.

```
public class SourceFixe extends Source<Boolean> {
    /**
     * Une source qui envoie toujours le même message
     */
    public SourceFixe (String messageString) {
        super();

        informationGeneree = new Information<Boolean>();

        for (int i = 0; i < messageString.length(); ++i) {
            boolean value = messageString.charAt(i) != '0';
            informationGeneree.add(value);
        }

        informationEmise = informationGeneree;
    }
}
```

Figure 3 – Code source de la classe **SourceFixe**

Le code de cette classe consiste juste en une boucle qui parcourt la séquence donnée par l'utilisateur. Si l'information est 0 la valeur ajoutée à la chaîne est FALSE, sinon c'est TRUE. Une fois la chaîne passée en revue, on assigne la variable informationEmise à la chaîne composée de TRUE et de FALSE.

2.2.2. Création de la classe SourceAleatoire

La classe **SourceAleatoire** est plus complexe puisqu'on peut l'appeler dans deux cas différents. Le point commun avec **SourceFixe** est qu'elle hérite également de la classe abstraite **Source**.

La première façon d'envoyer une séquence aléatoire pour l'utilisateur est de donner en paramètre un simple nombre. Ce nombre déterminera la taille de la séquence booléenne à envoyer. Dans ce cas c'est le constructeur montré en figure 4 qui sera appelé.

```
/**
 * Constructeur de la classe SourceAleatoire.
 * Génère une séquence aléatoire de bits de taille spécifiée.
 *
 * @param nbBitsMess le nombre de bits à générer dans la séquence.
 */
public SourceAleatoire(int nbBitsMess) {
    super();
    this.informationGeneree = new Information<Boolean>();

    Random random = new Random();
    // Génération de nbBitsMess bits aléatoires
    for (int i = 0; i < nbBitsMess; i++) {
        this.informationGeneree.add(random.nextBoolean());
    }

    this.informationEmise = this.informationGeneree;
}
```

Figure 4 – Code source du premier constructeur de la classe **SourceAleatoire**

Comme dans le cas d'une séquence fixe fournie par l'utilisateur, une nouvelle chaîne de booléen est créée et liée à la variable informationGeneree. Une boucle est ensuite parcourue autant de fois que la valeur passée en paramètre du constructeur. A chaque passage dans la boucle une valeur de booléen est choisie aléatoirement, puis ajoutée à la séquence. Enfin, la séquence finale est attribuée à la variable informationEmise.

La deuxième façon d'envoyer une séquence composée de '1' et de '0' de manière aléatoire est de faire appel à une graine grâce au paramètre seed. La figure 5 montre comment ce processus est utilisé.

```
* Constructeur de la classe SourceAleatoire avec graine (seed).
* Génère une séquence aléatoire de bits de taille spécifiée avec une graine pour la reproductibilité.
*
* @param taille la taille de la séquence de bits à générer.
* @param seed la graine utilisée pour initialiser le générateur aléatoire (permet la reproductibilité des séquences).
*/
public SourceAleatoire(int taille, int seed) {
    super();
    this.informationGenerée = new Information<>();

    Random random = new Random(seed);
    // Génération de taille bits aléatoires avec graine
    for (int i = 0; i < taille; i++) {
        this.informationGenerée.add(random.nextBoolean());
    }

    this.informationEmise = this.informationGenerée;
}
```

Figure 5 – Code source du deuxième constructeur de la classe **SourceAleatoire**

Une séquence de booléens est générée à nouveau et la graine donnée par l'utilisateur est également utilisée pour générer la séquence aléatoire. Cette fois, la nombre de passage dans la boucle dépend de la taille de la séquence de bits à transmettre.

2.2.3. Création de la classe TransmetteurParfait

Comme précisé auparavant, le transmetteur logique ne s'occupe, pour cette première itération, que de réceptionner le message émis par la source, puis de le transmettre à la destination. Par conséquent la classe **TransmetteurParfait**, qui hérite de la classe abstraite *Transmetteur*, ne contient que deux méthodes : recevoir() et emettre().

```
/**
 * recoit une information. Cette méthode, en fin d'exécution,
 * appelle la méthode émettre.
 *
 * @param information l'information reçue
 * @throws InformationNonConformeException si l'Information comporte une anomalie
 */
@Override
public void recevoir(Information<Boolean> information) throws InformationNonConformeException {
    this.informationRecue = information;
    emettre();
}
```

Figure 6 – Code source de la méthode recevoir() provenant de la classe **TransmetteurParfait**

Ce transmetteur devant être parfait, le message reçu en paramètre de la méthode n'a qu'à être émise. Pour faire cela la méthode recevoir() fait appelle à la deuxième méthode de cette classe : emettre().

```
/**
 * émet l'information construite par le transmetteur
 *
 * @throws InformationNonConformeException si l'Information comporte une anomalie
 */
@Override
public void emettre() throws InformationNonConformeException {
    this.informationEmise = this.informationRecue;

    // Émission vers les composants connectés
    for (DestinationInterface<Boolean> destinationConnectee : destinationsConnectees) {
        destinationConnectee.recevoir(this.informationEmise);
    }
}
```

Figure 7 – Code source de la méthode `emettre()` provenant de la classe **TransmetteurParfait**

Cette seconde méthode se contente de parcourir la liste des composants auquel le transmetteur est connecté, et de transmettre à chaque composant la séquence d'informations reçue.

2.2.4. Création de la classe DestinationFinale

Cette dernière classe **DestinationFinale** est extrêmement simple puisqu'elle ne fait que recevoir la séquence finale, et l'assigner à l'attribut de classe **informationRecue**.

```
public class DestinationFinale extends Destination<Boolean> {
    /**
     * recoit une information
     *
     * @param information l'information à recevoir
     * @throws InformationNonConformeException si l'Information comporte une anomalie
     */
    @Override
    public void recevoir(Information<Boolean> information) throws InformationNonConformeException {
        this.informationRecue = information;
    }
}
```

Figure 8 – Code source de la classe **DestinationFinale**

Maintenant que nous avons instancié l'ensemble des classes utiles pour la première étape de ce projet, nous devons connecter les différents composants du système entre eux. Le message rentré par l'utilisateur passera d'un bout à l'autre de la chaîne, avec un taux d'erreur binaire nul (transmetteur parfait).

2.3. Mise en fonctionnement du système

La mise en fonctionnement du système passe par la modification de la classe **Simulateur**, qui fait office de chef d'orchestre de notre logiciel. Nous commençons par instancier les différents composants puis nous les connectons entre eux. Une fois cela fait, nous intégrons les sondes dans notre chaîne de transmission. Enfin, il nous reste à lancer l'émission du message par la source, et à gérer le calcul du taux d'erreur binaire.

2.3.1. Instanciation des composants et connexions avec les sondes

Nous nous intéressons dans un premier temps au constructeur de la classe **Simulateur**. La figure 9 ci-dessous montre notre code.

```
public Simulateur(String[] args) throws ArgumentsException {
    // Analyser et récupérer les arguments
    analyseArguments(args);

    // Choix de la source en fonction des paramètres
    if (messageAleatoire) {
        if (aleatoireAvecGerme) {
            this.source = new SourceAleatoire(nbBitsMess, seed);
        } else {
            this.source = new SourceAleatoire(nbBitsMess);
        }
    } else {
        this.source = new SourceFixe(messageString);
    }

    // Instanciation des composants
    this.transmetteurLogique = new TransmetteurParfait();
    this.destination = new DestinationFinale();

    // Connexion des différents composants
    this.source.connecter(this.transmetteurLogique);
    this.transmetteurLogique.connecter(destination);

    // Connexion des sondes (si l'option -s est pas utilisée)
    if (affichage) {
        this.source.connecter(new SondeLogique( nom: "source", nbPixels: 200));
        this.transmetteurLogique.connecter(new SondeLogique( nom: "transmetteur", nbPixels: 200));
    }
}
```

Figure 9 – Constructeur de la classe **Simulateur**

Le code de ce constructeur est divisé en plusieurs étapes :

1. Tout d'abord, nous faisons appel à la méthode `analyseArguments()` pour vérifier que les paramètres fournis par l'utilisateur répondent aux attentes du logiciel. Si ce n'est pas le cas, l'exception `ArgumentsException` est levée ;
2. Une fois que les paramètres sont validés, nous choisissons la source à utiliser. Pour faire cela, nous regardons si une graine a été donnée par l'utilisateur, s'il utilise une séquence fixe ou encore s'il veut créer une séquence booléenne à partir d'un message ;
3. Nous instancions ensuite le transmetteur logique ainsi que le composant final de notre système ;
4. Ensuite, nous connectons entre eux les composants. Pour cette première étape, la source est connectée au transmetteur logique qui est lui-même connecté à la destination ;
5. Enfin, nous ajoutons à cette structure les deux sondes représentées sur la figure 1.

2.3.2. Émission du message par la source

Maintenant que la chaîne de transmission est formée, il nous suffit d'émettre la séquence de booléens pour qu'elle passe d'un bout à l'autre du système. Pour faire cela nous programmons la méthode `execute()` présente dans la classe **Simulateur**.

```
public void execute() throws Exception {
    // La source émet le message
    source.emettre();
}
```

Figure 10 – Méthode `execute()` présente dans la classe **Simulateur**

Cette simple méthode utilise la source choisie précédemment (fixe ou aléatoire) ainsi que la méthode `emettre()` présente dans la classe abstraite **Source**.

2.3.3. Gestion du taux d'erreur binaire

Pour rappel, le taux d'erreur binaire (TEB) est calculé après que la séquence booléenne soit passée dans l'ensemble des composants du système. Pour obtenir ce TEB, nous avons créé une méthode `calculTauxErreurBinaire()` au sein de la classe **Simulateur**.

```
/**
 * La méthode qui calcule le taux d'erreur binaire en comparant
 * les bits du message émis avec ceux du message reçu.
 *
 * @return La valeur du Taux d'erreur Binaire.
 */
± jordanbmrd +1
public float calculTauxErreurBinaire() {
    Information messageEmis = this.source.getInformationEmise();
    Information messageReçu = this.destination.getInformationReçue();

    // Vérification de la taille des messages
    if (messageEmis.nbElements() != messageReçu.nbElements()) {
        throw new IllegalArgumentException("La taille du message émis est différente de celle du message reçu.");
    }

    int nbBits = messageReçu.nbElements();
    if (nbBits == 0) {
        return 0f;
    }

    float nbErreurs = 0f;

    // Parcours des éléments pour comparer bit par bit
    for (int i = 0; i < nbBits; ++i) {
        if (messageEmis.iemeElement(i) != messageReçu.iemeElement(i)) {
            nbErreurs++;
        }
    }

    // Calcul du taux d'erreur
    return nbErreurs / nbBits;
}
```

Figure 11 – Méthode `calculTauxErreurBinaire()` présente dans la classe **Simulateur**

2.4. Automatisation du logiciel via des scripts

Notre chaîne de transmission composée d'une source, d'un transmetteur logique « parfait », d'une destination ainsi que de deux sondes est désormais opérationnelle. Il nous reste désormais à simplifier la gestion de notre logiciel. Pour faire cela, l'utilisateur final nous demande de fournir plusieurs scripts. Cette quatrième partie est consacrée au développement de ces scripts.

2.4.1. Script compile

Le premier script bash nommé *compile* permet de compiler l'ensemble du code source permettant à notre logiciel de fonctionner.

```
#!/bin/bash
💡
# Suppression du dossier bin/
rm -rf bin/

# Compilation du projet
echo "Compiling into bin/ folder"
javac -d bin/ src/**/*.java
echo "Done!"
```

Figure 12 – Code source du script *compile*

Ce programme commence par supprimer le contenu du fichier bin/ au cas où l'utilisateur aurait déjà compilé le programme JAVA auparavant. Il utilise ensuite la commande javac afin de compiler l'ensemble des fichiers JAVA contenus dans le dossier src du projet SIT_213. L'affichage de commentaires via la commande echo permet de vérifier que la compilation s'est bien passée comme le montre la figure ci-dessous.

```
[polguillou@pol Projet % ./compile
Compiling into bin/ folder
Done!
```

Nom	Date de modification	Taille	Type
destinations	aujourd'hui à 14:27	--	Dossier
Destination.class	aujourd'hui à 14:26	776 octets	Fichier...se Java
DestinationFinale.class	aujourd'hui à 14:26	578 octets	Fichier...se Java
DestinationInterface.class	aujourd'hui à 14:26	466 octets	Fichier...se Java
information	aujourd'hui à 14:27	--	Dossier
Information.class	aujourd'hui à 14:26	2 ko	Fichier...se Java
InformationNonConformeException.class	aujourd'hui à 14:26	401 octets	Fichier...se Java
simulateur	aujourd'hui à 14:27	--	Dossier
ArgumentsException.class	aujourd'hui à 14:26	311 octets	Fichier...se Java
Simulateur.class	aujourd'hui à 14:26	4 ko	Fichier...se Java
sources	aujourd'hui à 14:27	--	Dossier
Source.class	aujourd'hui à 14:26	2 ko	Fichier...se Java

Figure 13 – Résultat du script *compile*

2.4.2. Script genDoc

Le second script *genDoc* permet à l'utilisateur qui l'exécute de générer la documentation de l'ensemble du projet.

```
#!/bin/bash
💡
# Suppression du dossier docs/
rm -rf docs/

# Génération de la Javadoc
echo "Generating javadoc into /docs folder"
javadoc -d ./docs -quiet ./src/**/*.java -Xdoclint:none
echo "Done!"
```

Figure 14 – Code source du script *genDoc*

Comme pour le script *compile*, *genDoc* commence par supprimer le contenu du dossier *docs/* où est stockée la documentation du projet, afin de ne pas avoir de doublon ou des documentations obsolètes. Une fois que les anciennes versions sont supprimées, *genDoc* se sert de la commande *javadoc* pour insérer dans le dossier *docs/* la *javadoc*.

```
polguillou@pol Projet % ./genDoc
Generating javadoc into /docs folder
Done!
```

docs				
Nom	Date de modification	Taille	Type	
visualisations			-- Dossier	
VueValeur.html	aujourd'hui à 14:40	101 ko	Texte HTML	
VueCourbe.html	aujourd'hui à 14:40	106 ko	Texte HTML	
Vue.html	aujourd'hui à 14:40	106 ko	Texte HTML	
SondeTextuelle.html	aujourd'hui à 14:40	14 ko	Texte HTML	
SondePuissance.html	aujourd'hui à 14:40	15 ko	Texte HTML	
SondeLogique.html	aujourd'hui à 14:40	15 ko	Texte HTML	
SondeAnalogique.html	aujourd'hui à 14:40	15 ko	Texte HTML	
Sonde.html	aujourd'hui à 14:40	15 ko	Texte HTML	
package-tree.html	aujourd'hui à 14:40	7 ko	Texte HTML	
package-summary.html	aujourd'hui à 14:40	6 ko	Texte HTML	
type-search-index.js	aujourd'hui à 14:40	961 octets	text document	
transmetteurs			-- Dossier	
TransmetteurParfait.html	aujourd'hui à 14:40	17 ko	Texte HTML	
Transmetteur.html	aujourd'hui à 14:40	26 ko	Texte HTML	

Figure 15 - Résultat du script *genDoc*

2.4.3. Script cleanAll

Le script bash *cleanAll* est très simple, puisqu'il ne fait que supprimer le contenu des dossiers *bin/* et *docs/*. Grâce à cela, un projet peut être rendu comme étant « vierge » sans contenir les fichiers compilés ou la *Javadoc*.

```
#!/bin/bash
💡
# Suppression :
# - des archives générées
# - des fichiers compilés
# - de la Javadoc générée

echo "Cleaning..."
rm -f *.tar.gz
rm -rf bin/*
rm -rf docs/*
echo "Done!"
```

Figure 16 – Code source du script *cleanAll*

```
[polguillou@pol Projet % ./cleanAll
Cleaning...
Done!
[polguillou@pol Projet % ls bin/
[polguillou@pol Projet % ls docs/
```

Figure 17 - Résultat du script *cleanAll*

2.4.4. Script *genDeliverable*

Le dernier script de la catégorie « gestion du projet » se nomme *genDeliverable*. Il permet de créer une archive du projet complet au format .tar.gz.

```
#!/bin/bash
💡
# Clean le projet
./cleanAll

# Récupération des dernières sources du projet
# echo "Récupération de la dernière version du projet"
# git pull --quiet

# Création de l'archive
echo "Creating archive..."
tar --exclude='genDeliverable' --exclude='Déploiement' --exclude='Paramétrage' --exclude='.git'
--exclude='Livrables/' -czf GUILLOU-MAQUENNE-BAUMARD.SIT213.Étape1.tar.gz *
echo "Archive created with success!"
```

Figure 18 – Code source du script *genDeliverable*

Le programme *genDeliverable* commence par appeler le script *cleanAll* présenté ci-dessus. On se sert ensuite de la commande tar avec plusieurs paramètres pour archiver le dossier contenant l'ensemble du projet, à l'exception de certains fichiers dont l'utilisateur final n'a pas besoin. Quand nous exécutons ce script une archive est créée. C'est cette dernière que nous envoyons à l'utilisateur final.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Étape 5 : Codage de canal</p>	<p>Année scolaire 2024-2025</p> <p>Page : 19/68</p>
---	---	---

2.5. Tests et résultats de la première itération

En supplément des quatre scripts présentés précédemment, nous avons utilisé trois programmes supplémentaires afin de tester la première itération de ce projet à travers différents exemples.

2.5.1. Script simulateur

Pour rappel, c'est la classe java **Simulateur** qui permet de mettre en route notre logiciel et de démarrer la chaîne de transmission. Or, notre client souhaite pouvoir utiliser ce système grâce à l'appel d'un script *simulateur* suivi de plusieurs paramètres que nous avons étudiés auparavant (ajout de sondes, utilisation d'une graine...). Nous avons donc répondu à sa demande en confectionnant le script bash suivant :

```
#!/bin/bash

# shellcheck disable=SC2068
java -cp bin/ simulateur.Simulateur $@
```

Figure 19 – Code source du script Simulateur

Ce script lance simplement l'exécution de la classe *Simulateur* puis prend en compte les différents paramètres que l'utilisateur final veut prendre en compte dans la chaîne de transmission.

2.5.2. Script runTests

Pour cette première itération, notre client voulait également que nous passions une batterie de test à notre logiciel grâce à un dernier script. Ce script *runTests*, dont le code source est affiché ci-dessous, commence par compiler notre logiciel en faisant appel au script bash *compile*.

```
#!/bin/bash
#
# Compilation
./compile

# Liste des scénarios de tests à exécuter
test_cases=(
    "-mess 101000101"
    "-mess 325 -s"
    "-seed 1234"
)

# Initialisation du compteur d'échecs
failed_tests=0

# Boucle sur les scénarios de tests
for test_case in "${test_cases[@]}"; do
    ./simulateur "$test_case"
    if [ $? -ne 0 ]; then
        echo "Échec du test : $test_case"
        ((failed_tests++))
    fi
done

# Résumé des résultats des tests
if [ "$failed_tests" -ne 0 ]; then
    echo -e "\n$failed_tests tests échoués sur ${#test_cases[@]}"
    exit 1
else
    echo "Tous les tests sont passés avec succès"
fi
```

Le corps de ce programme est décomposé en trois parties :

- Premièrement, la liste des paramètres qui suivent l'appel du script *simulateur*. Nous avons déterminé les trois tests ci-contre puisqu'ils représentent les différentes utilisations que l'utilisateur pourrait faire du logiciel ;
- Ensuite, une boucle qui exécute un par un les tests configurés en première partie du script ;
- Enfin, des messages de sortie qui indiquent à l'utilisateur si les tests se sont bien déroulés ou certains n'ont pas pu

Figure 20 – Code source du script *runTests*

```
[polguillou@pol sit213 % ./runTests
Compiling into bin/ folder
Done!
java Simulateur -mess 101000101 => TEB : 0.0
java Simulateur -mess 325 -s => TEB : 0.0
java Simulateur -seed 1234 => TEB : 0.0
Tous les tests sont passés avec succès
```

Figure 21 – Résultats du script *runTests*

La figure 21 nous montre que les trois tests que nous avons effectués ont été exécutés sans problème. Par ailleurs, cette première itération simplifie les tests que nous avons effectués car nous sommes ici dans le cas d'un transmetteur parfait. Par conséquent, il nous suffit de vérifier que les tests sont arrivés à terme et que le Taux d'erreur vaut 0 à chaque fois. Enfin, nous pouvons observer sur la figure 22 les données remontées par les deux sondes installées sur la chaîne de transmission et utilisée lors du second test grâce au paramètre '*s*'.

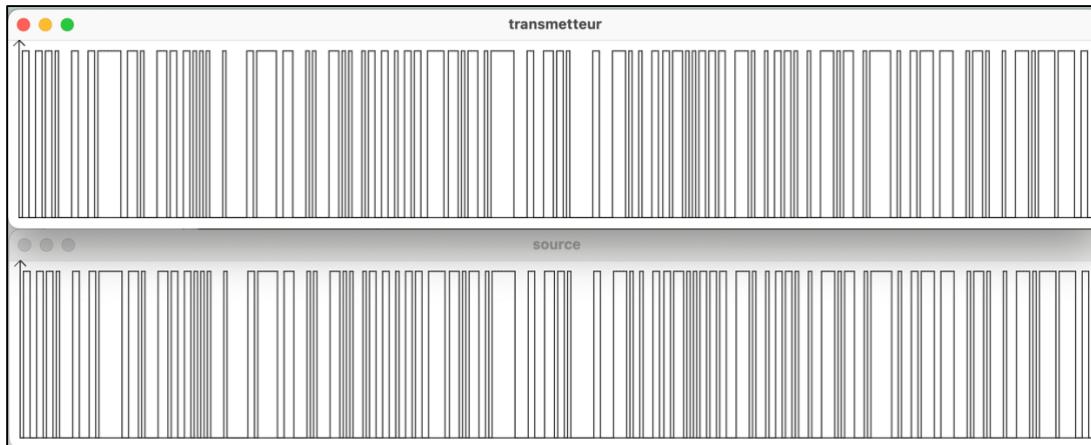


Figure 22 – Données remontées par les deux sondes

2.5.3. Vérification de l’itération 1 avec le script *Deploiement*

Pour conclure cette première itération, nous avons utilisé un nouveau script bash nommé *Deploiement*. A l’inverse des autres scripts, ce dernier nous a été fourni par le client lui-même. Il nous a conseillé de s’en servir sur l’archive que nous voulions lui rendre. En effet, ce programme passe en revue toutes les fonctionnalités que la première étape de ce projet doit contenir. Nous l’avons donc exécuté avec l’archive GUILLOU-MAQUENNE-BAUMARD.SIT213.Étape1.tar.gz.

```

Génération de la javadoc :
Generating javadoc into /docs folder
Done!
-n .
-n .

Lancement de l'autotest :
Compiling into bin/ folder
Done!
java Simulateur -mess 101000101 => TEB : 0.0
java Simulateur -mess 325 -s => TEB : 0.0
java Simulateur -seed 1234 => TEB : 0.0
Tous les tests sont passés avec succès
--- Déploiement terminé ! ---

```

Après avoir effectué une série de tests sur l’archive passée en paramètre, le script *Deploiement* nous annonce que le déploiement s’est terminé sans rencontrer d’erreur.

Figure 23 - Exécution du script *Deploiement*

III. Itération 2 : Transmission analogique non bruitée

3.1. Cahier des charges

Pour cette seconde itération, nous gardons le principe d'une chaîne de transmission parfaite. Pour rappel, nous simulons cela grâce à un transmetteur dit « parfait », c'est-à-dire qu'il n'ajoute pas de bruit au signal qu'il transmet. Même si cela n'arrive jamais en pratique, il est très pratique de commencer à implémenter notre logiciel dans cette situation puisque le taux d'erreur binaire est nul en sortie du système.

Comme lors de la précédente itération, notre chaîne comporte en plus du transmetteur, une source et une destination. Ces deux blocs sont situés aux extrémités de la chaîne de transmission et travaillent exclusivement avec des suites de booléens.

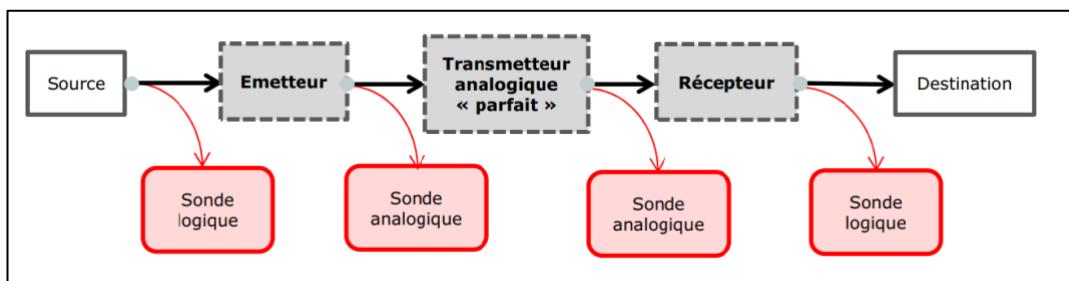


Figure 24 - Modélisation de la chaîne de transmission à l'étape 2

La figure 24 montre la chaîne de transmission complète dédiée à la seconde itération. Nous pouvons remarquer qu'en plus des trois blocs utilisés précédemment, un bloc **Emetteur** ainsi qu'un bloc **Récepteur** sont introduits dans le système :

- Le bloc **Emetteur** reçoit un signal numérique, c'est-à-dire une suite de booléens, depuis la source de la chaîne. Son rôle est de transformer ce signal numérique en signal analogique avant de l'envoyer au transmetteur analogique ;
- Le **Récepteur** s'occupe quant à lui de recevoir le signal analogique venant du transmetteur, et de le retransformer en signal numérique, avant de le transmettre à la destination.

Passer d'un type de signal à l'autre au milieu de la chaîne de transmission permet de mieux adapter le signal aux canaux de transmission. Cette manipulation à un réel intérêt dans des cas pratique où du bruit est ajouté par le canal de transmission. Cette opération facilitera la modulation et la démodulation du signal, afin de récupérer en sortie du système les données malgré les perturbations du canal. Cela prendra donc tout son sens lors des itérations suivants où notre signal subira du bruit de manière aléatoire.

Cette seconde itération introduit également dans notre logiciel différents types de signaux en entrée de notre système. Ces signaux (appelés également code) peuvent être sous trois formes différentes :

- **NRZ** Binaire ;
- **NRZT** Binaire ;
- **RZ** Binaire.

Chaque code a ses spécificités qui nous sont également rappelées dans le cahier des charges. Une d'entre elle est caractérisée les amplitudes maximales et minimales que chaque code peut avoir. En effet, il faut que les règles suivantes soient appliquées :

- Pour **NRZ** et **NRZT** : $A_{max} \geq 0 \text{ & } A_{min} \leq 0 \text{ & } A_{min} < A_{max}$
- Pour **RZ** : $A_{max} \geq 0 \text{ & } A_{min} = 0 \text{ & } A_{min} < A_{max}$

3.2. Implémentation des fonctionnalités

Sachant que ce logiciel est construit de manière itérative, c'est-à-dire que chaque semaine un nouveau bloc de fonctionnalités est introduit, la base de notre code Java reste la même. Nous devons en revanche la modifier pour y inclure, entre autres, l'émetteur et le récepteur dont nous avons parlé auparavant.

3.2.2. Fonctionnement de l'émetteur et du récepteur

Avant de passer à la partie technique et de développer en Java, il est important de comprendre comment fonctionnent ces deux éléments désormais essentiels pour transmettre l'information utile. Entre la réception et l'émission du message par le récepteur ce dernier effectue deux actions décrites par la figure ci-dessous.

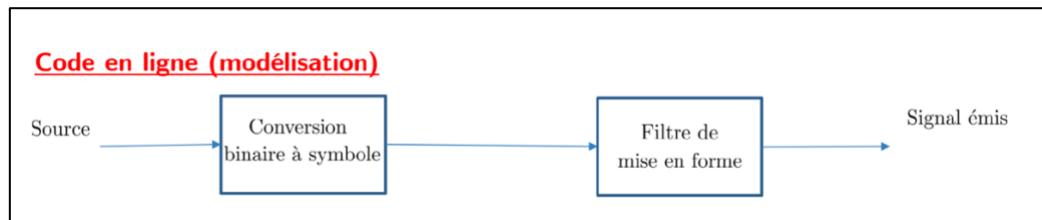


Figure 25 - Fonctionnement du bloc émetteur

L'émetteur commence par convertir, bit à bit, le signal qu'il reçoit de la source sous forme de symbole, afin qu'il devienne un signal analogique. Une fois cela fait, l'élément de notre système applique au signal analogique un filtre de mise en forme. C'est ce filtre qui définit le code de notre signal qui transite par le canal d'information. Dans notre cas nous avons affaire à trois codes binaires différents : **NRZ**, **RZ** et **NRZT**. La forme de ces trois signaux est présentée par la figure 26.

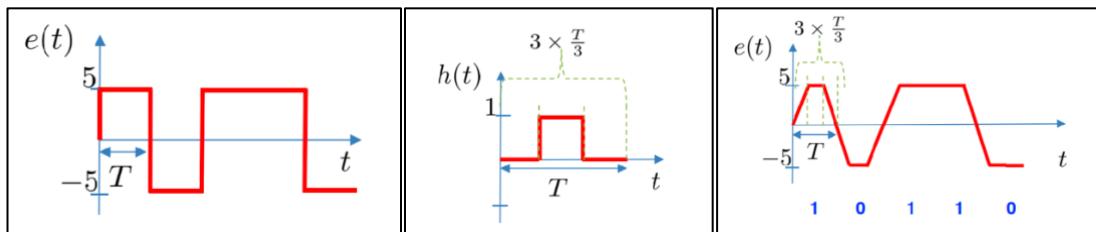


Figure 26 - Présentation des codes binaires en ligne NRZ, RZ et NRZT

La figure 26 nous permet d'observer que ces trois codes ont au moins un point commun : chaque bit peut être divisé en trois parties équitables. La différence notable se trouve dans la différence de forme entre ces trois tiers de période. Dans le cas du code NRZ binaire la valeur du signal reste constante. Pour un code RZ binaire l'amplitude ne vaut aMin ou aMax qu'au milieu du temps bit. Enfin, pour un signal du type NRZT binaire l'amplitude varie progressivement pour atteindre un extrémum, avant de rejoindre la valeur du prochain bit de manière progressive également. Nous pouvons d'ailleurs noter que c'est ce troisième exemple qui semble le plus proche de la réalité, puisque dans des cas réels l'amplitude du signal ne peut pas passer d'un extrême à l'autre en un instant t.

Dans le cas du récepteur, situé après le transmetteur parfait, ce sont les actions inverses qui sont effectuées. En effet, le signal analogique est d'abord remis sous sa forme d'origine, avant de repasser en un signal numérique.

Maintenant que nous avons bien compris comment chaque nouvel élément de la chaîne de transmission fonctionne, ainsi que les particularités de chaque code traité par notre logiciel, nous pouvons passer à la partie programmation Java.

3.2.3. Création de classes dédiées à la seconde étape

Pour rappel, lors de la première itération chaque élément de la chaîne de transmissions avait une classe Java qui lui était dédié. Nous avons donc continué sur cette lancée en créant une classe **Emetteur**, une classe **Recepteur** ainsi qu'une classe abstraite **Modulateur**.

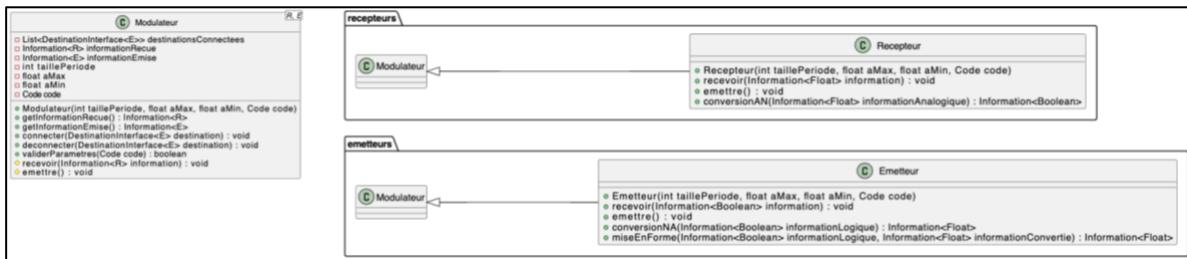


Figure 27 - Diagramme de classes lié à l'itération 2

Nous avons vu précédemment qu'au sein de la chaîne de transmission, l'émetteur et le récepteur ont beaucoup de similitudes comme le fait de recevoir et d'émettre des informations ou d'être connecter à différents éléments du système. Les informations manipulées par ces deux blocs sont également semblables (le type de code, l'amplitude minimum et maximum du signal à transmettre...). C'est pour ces raisons que nous avons intégré à notre programme une classe abstraite **Modulateur**. Grâce à cela les classes **Emetteur** et **Recepteur** n'ont qu'à venir piocher dans les méthodes décrites dans **Modulateur** et nous ne nous retrouvons pas avec des portions de code en double. La classe abstraite **Modulateur** contient également la méthode **validerParametre()** qui nous permet de vérifier que les valeurs d'amplitudes rentrées par l'utilisateur respecte les règles décrites précédemment.

- Pour **NRZ** et **NRZT** : $A_{max} \geq 0 \& A_{min} \leq 0 \& A_{min} < A_{max}$
- Pour **RZ** : $A_{max} \geq 0 \& A_{min} = 0 \& A_{min} < A_{max}$

La figure 27 montre également que notre code ne possède qu'une classe pour représenter l'émetteur de notre chaîne de transmission, ainsi qu'une classe pour simuler le récepteur. Sachant que la première étape (conversion binaire → symbole) est la même pour les trois codes binaires, cela ne nous semblait pas optimal de coder une classe par code binaire.

La différence majeure se trouve dans l'application ou non d'un filtre de mise en forme. Une simple condition dans notre code suffit à déterminer si le signal qui arrive en entrée de l'émetteur doit être filtré ou non. Si c'est le cas, une méthode **miseEnForme()** est appelée. Cette méthode analyse chaque bit et, à partir des tiers de périodes définis précédemment, peut appliquer un filtre différent en fonction du moment du bit qui est analysé.

Il nous a semblé encore plus logique de n'utiliser qu'une classe pour la partie réceptrice du système. En effet, pour les codes binaires NRZ, RZ et NRZT qui arrivent sous forme analogique il suffit de regarder la valeur extrême pour chaque bit. Nous la comparons ensuite avec les valeurs d'amplitude maximale et minimale données en entrées de la chaîne de transmission et cela nous permet de reproduire le signal numérique original.

3.2.4. Connexions des sondes à la chaîne de transmission

Un moyen très fiable de vérifier que le signal passe bien à travers notre chaîne de transmission est de visualiser le signal qui sort de chaque élément du système (la source, l'émetteur, le transmetteur, le récepteur et la destination). Pour faire cela nous utilisons des sondes que nous connectons aux différents blocs de notre structure. Lors de la première itération nous n'avions utilisé que des sondes logiques puisque le signal était numérique d'un bout à l'autre du système. Maintenant que le signal est sous forme analogique pendant une partie de son trajet, nous introduisons également dans notre code des sondes du même type que le signal. La figure 24 présentée dans la partie 3.1. montre les différentes sondes que nous plaçons tout le long de notre chaîne. Leur emplacement relève simplement de la logique :

- Si le signal qui sort du bloc est **numérique** nous utilisons une **sonde logique** ;
- Si le signal qui sort du bloc est **analogique** nous utilisons une **sonde analogique**.

```
// Connexion des sondes
if (affichage) {
    this.source.connecter(new SondeLogique( nom: "source " + code, nbPixels: 200));
    this.emetteur.connecter(new SondeAnalogique( nom: "emetteur " + code));
    this.transmetteurAnalogique.connecter(new SondeAnalogique( nom: "transmetteur " + code));
    this.recepteur.connecter(new SondeLogique( nom: "recepteur " + code, nbPixels: 200));
}
```

Figure 28 - Code permettant la connexion des sondes à notre système

Une fois que nous savons où placée telle ou telle sonde, il nous suffit d'utiliser la méthode `connecter()` pour relier chaque sonde aux éléments de notre chaîne de transmission, comme le montre la figure 28.

3.2.3. Intégration des nouveaux paramètres à la commande permettant d'utiliser le logiciel

La dernière chose qu'il reste à intégrer à notre code concerne l'utilisation du logiciel. Pour utiliser cette chaîne de transmission, l'utilisateur doit appeler le script *Simulateur* suivi de plusieurs paramètres qui décrivent le signal passé en entrée du système. Jusqu'à présent l'utilisateur pouvait :

- Préciser le message ou la longueur du message à émettre ;
- Utiliser ou non les sondes intégrées à la structure ;
- Générer un signal aléatoire à partir d'une graine donnée en paramètre.

Cette seconde itération doit désormais permettre à l'utilisateur de notre logiciel de choisir quel code binaire utiliser, ainsi que l'amplitude minimum et maximum que le signal aura. Pour faire cela, nous nous intéressons à classe mère de notre projet : **Simulateur**. C'est cette classe qui est exécutée lors de l'appel du script qui porte le même nom. De plus, c'est dans cette classe que les autres paramètres (-mess, -s, -seed...) sont gérés. Cela nous semblait donc logique d'intégrer les trois nouveaux paramètres au même endroit.

Le choix du code binaire pour l'utilisateur se fera grâce au paramètre « `-form` » suivi du nom du code choisi (**NRZ**, **NRZT** ou **RZ**). Pour ce qui est de l'amplitude, l'usage pourra se servir du paramètre « `-ampl` » suivi de la valeur des amplitudes minimales et maximales. Par défaut, le code binaire utilisé est RZ, l'amplitude minimale vaut 0 et l'amplitude maximale vaut 1.

3.3. Résultats obtenus

Après avoir implémenté les fonctionnalités liées à l'itération 2, il nous reste à appeler la classe **Simulateur** avec les nouveaux paramètres. Cela nous permet de vérifier si les résultats retournés par les sondes sont cohérents avec ce que l'on attend en théorie. Pour l'instant nous sommes dans le cas d'un transmetteur parfait qui n'ajoute pas de bruit à notre signal. Par conséquent, les deux sondes logiques donnent le même résultat, tout comme les deux sondes analogiques. Nous avons choisi de nous concentrer sur les sondes postées en sortie de l'émetteur et du récepteur puisque ce sont les deux éléments cruciaux de cette chaîne de transmission adaptée à la seconde itération. Pour chaque code nous décidons de simuler le signal correspondant à la chaîne booléenne suivante : « 1101000101 ». Nous avons choisi cette suite puisqu'elle contient une alternance entre les 0 et 1, mais également au moins deux bits d'affilée qui sont les mêmes.

3.3.1. Simulation d'un signal NRZ binaire

Le premier type de signal que nous passons en entrée de la chaîne de transmission utilise le code binaire NRZ dont le résultat théorique est présenté dans la partie 3.2.2.



```
Build and run
Modify options ▾ M

java 21 SDK of 'sit213' ✓ simulateur.Simulateur
-mess 1101000101 -code NRZ -aMax 5 -aMin -5 -s
```

Figure 29 - Commande permettant la simulation d'un signal NRZ binaire

Le code NRZ binaire permettant d'avoir une amplitude minimale inférieure à 0, nous en profitons pour avoir un signal centré en 0.

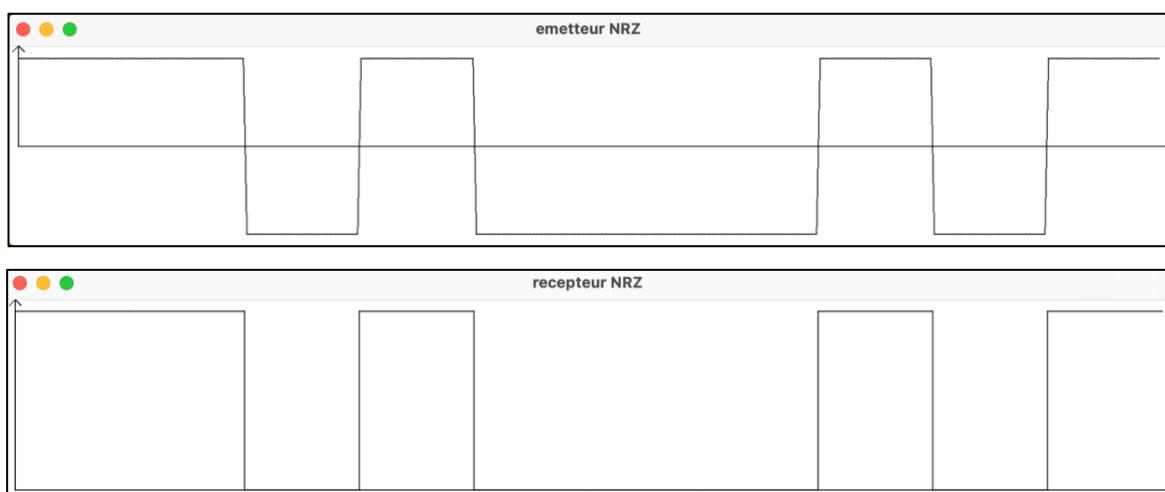


Figure 30 - Résultats du signal NRZ binaire

Le code NRZ binaire étant le seul code sur lequel aucun filtre n'est appliqué, il est normal que les signaux captés par les sondes logiques et analogiques soient très similaires. On remarque tout de même que le signal analogique capturé par l'émetteur ne passe pas d'un extrémum à l'autre. C'est tout à fait normal puisqu'il n'est pas possible dans un cas réel de passer en un instant d'une valeur x à une valeur $-x$. Les résultats obtenus ici sont donc ceux attendus.

3.3.2. Simulation d'un signal RZ binaire

Une des particularités du code RZ binaire est que son amplitude minimale doit forcément être égale. Modifions donc notre ligne de commande pour s'adapter à ce second type de signal.



```
Build and run
Modify options ▾ M

java 21 SDK of 'sit213' ✘ simulateur.Simulateur
-mess 1101000101 -code RZ -aMax 3 -aMin 0 -s
```

Figure 31 - Commande permettant la simulation d'un signal RZ binaire

Cette fois le signal qui passe arrive en entrée du récepteur est soumis à un filtre de mise en forme. Ce dernier est présenté sur la figure 26.

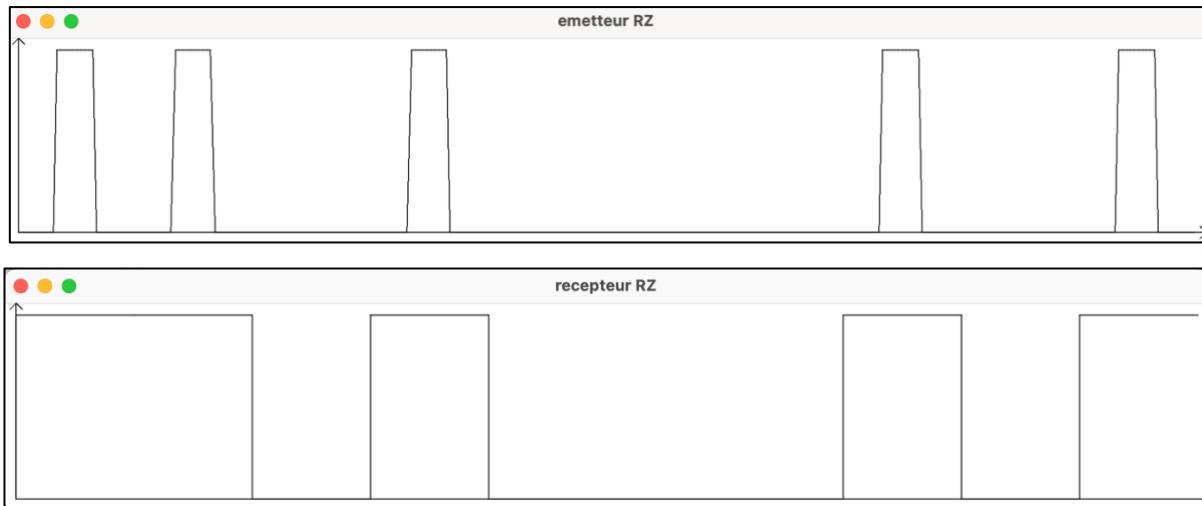


Figure 32 - Résultats du signal RZ binaire

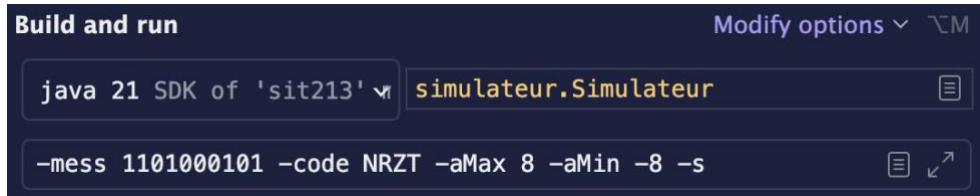
La figure 32 montre que le résultat retourné par les sondes analogiques sont extrêmement proches du résultat théorique présenté précédemment. Pour chaque bit logique le signal est bien décomposé en trois parties égales. Aux extrémités la valeur vaut toujours 0, et au milieu du bit elle vaut aMax ou aMin en fonction du booléen passé en entrée du système. Comme pour le code NRZ binaire nous pouvons observer que lorsque l'amplitude passe progressivement d'une valeur à l'autre dans le signal analogique, contrairement au résultat fourni par la sonde logique. Cela est dû uniquement aux sondes analogiques, puisqu'on ne retrouve aucune valeur intermédiaire quand on observe précisément les valeurs à la sortie de l'émetteur, comme le montre la figure 33.

Signal analogique à la sortie de l'émetteur pour un signal RZ binaire : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0

Figure 33 - Valeur du signal analogique capté à la sortie de l'émetteur

3.3.3. Simulation d'un signal NRZT binaire

Le troisième et dernier type de signal que nous simulons est le code NRZT binaire. Comme pour le code RZ binaire, l'émetteur applique au signal numérique un filtre de mise en forme dont parlé auparavant. Nous décidons d'utiliser des valeurs d'amplitude différentes pour ce code puisqu'il a moins de contraintes que celui traité précédemment.



```
Build and run
Modify options ⌂M

java 21 SDK of 'sit213' ↵ simulateur.Simulateur
-mess 1101000101 -code NRZT -aMax 8 -aMin -8 -s
```

Figure 34 - Commande permettant la simulation d'un signal RZ binaire

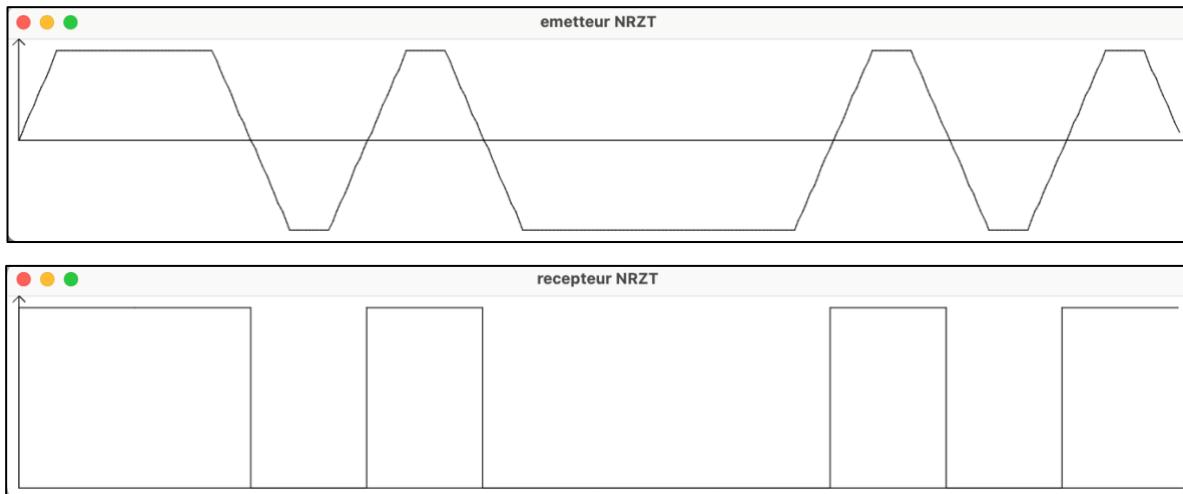


Figure 35 - Résultats du signal NRZT binaire

La complexité de ce filtre réside dans le fait qu'il ne réagit pas de la même manière en fonction du bit suivant. En effet, si dans la chaîne de booléens deux TRUE sont l'un après l'autre l'amplitude du signal restera la même pendant toute la suite de bits similaires. Par ailleurs, la transition entre deux bits différents (pour un signal numérique TRUE → FALSE ou FALSE → TRUE) est lissée directement grâce au filtre de mise en forme. Dans un cas réel cela permet une meilleure compatibilité entre les systèmes analogiques, notamment pour certains systèmes de transmission qui peuvent être sensibles aux changements rapides de signal.

3.4. Tests effectués

Nous avons mis en place une série de tests pour valider le bon fonctionnement des différentes composantes du système. Nous avons réalisé les tests unitaires à l'aide de JUnit 4. Nous avons également utilisé Emma afin d'évaluer la couverture de code et nous assurer que les tests couvrent toutes les parties de l'application. Pour simuler certains composants, nous avons utilisé EasyMock, qui permet de mocker et tester des fonctions.

Les tests que nous avons effectués couvrent à la fois les performances du projet ainsi que l'analyse des résultats (TEB). Nous avons décidé de mettre en place ces tests puisque cette itération le permet du fait que le transmetteur n'ajoute pas de bruit au signal.

3.4.1. Tests de la classe Emettre

Les tests liés à cette classe nous permettent de vérifier le bon fonctionnement du passage SIGNAL NUMÉRIQUE → ANALOGIQUE pour les codes binaires NRZ, NRZT et RZ. Nous pouvons voir sur la figure 36 que les trois tests que nous avons effectués sur cette classe sont exécutés et renvoient les résultats attendus.

- ✓ EmetteurTest
 - ✓ testEmettre
 - ✓ testRecevoir
 - ✓ testConversionNA

Figure 36 - Résultat des tests pour la classe Emettre

La figure ci-dessus montre la liste des tests qui sont effectués sur la classe Emetteur :

- testEmettre s'assure que la méthode emettre() génère bien une information émise pour différents types de codages binaires (NRZ, RZ, NRZT) ;
- testRecevoir vérifie que la méthode recevoir() de chaque émetteur reçoit correctement l'information avant de l'émettre ;
- testConversionNA examine la conversion numérique-analogique (NA) pour chaque type de codage. Il vérifie que le nombre d'éléments et les valeurs des signaux convertis sont corrects (ex. : valeur maximale atteinte dans le cas du NRZT) ;

3.4.2. Tests de la classe Recepteur

Les tests de la classe **Recepteur** visent à valider la réception, la conversion SIGNAL ANALOGIQUE → NUMÉRIQUE ainsi que la gestion des erreurs. La figure 37 nous permet de vérifier que l'ensemble des tests passés sur cette classe se sont déroulés sans erreur.

- ✓ RecepteurTest
 - ✓ testEmettre
 - ✓ testValiderParametres
 - ✓ testRecevoirInformationInvalide
 - ✓ testConversionANInformationVide
 - ✓ testRecevoirInformationVide
 - ✓ testValiderParametresInvalide
 - ✓ testConversionANInformationInvalide
 - ✓ testConversionAN
 - ✓ testRecevoirInformationValide

Figure 37 - Résultat des tests pour la classe Emettre

Cette classe de tests nous sert à vérifier le bon fonctionnement de l'ensemble des méthodes incluses dans la classe **Recepteur** :

- testRecevoirInformationValide vérifie que le récepteur reçoit correctement une information analogique valide et la transmet à la destination connectée après l'avoir convertie ;
- testRecevoirInformationInvalide s'assure que la réception d'une information nulle provoque une exception `InformationNonConformeException` ;
- testRecevoirInformationVide vérifie que la réception d'une information vide déclenche également une exception `InformationNonConformeException` ;
- testEmettre valide que le récepteur émet une information correcte après réception d'une information analogique valide. La taille de l'information émise est vérifiée après conversion en fonction de la période définie (ici, période de 2) ;
- testValiderParametres s'assure que la méthode `validerParametres` valide correctement les paramètres (type de codage, `aMin`, `aMax`...) ;
- testValiderParametresInvalide vérifie que des paramètres invalides, comme un intervalle incohérent où `aMin >= aMax`, déclenchent une exception ;
- testConversionAN s'assure que la conversion d'une information analogique valide en binaire est effectuée correctement. La taille de l'information binaire est vérifiée après la conversion ;
- testConversionANInformationInvalide s'assure que la tentative de conversion d'une information nulle en binaire lève une exception `InformationNonConformeException` ;
- testConversionANInformationVide vérifie que la conversion d'une information vide en binaire lève également une exception `InformationNonConformeException`.

3.4.3. Tests de la classe SourceFixe

Les tests effectués en lien avec la classe **SourceFixe** vérifient que les messages fournis par l'utilisateur, lors de l'utilisation du logiciel, sont générés et émis correctement.

- ✓ SourceFixeTest
 - ✓ testMessageStringWithOnlyZeros
 - ✓ testMessageStringWithOnlyOnes
 - ✓ testInformationEmise
 - ✓ testMessageWithAlternatingBits
 - ✓ testConstructor
 - ✓ testEmptyMessageString

Figure 38 - Résultat des tests pour la classe `SourceFixe`

Tout comme les précédentes classes, les méthodes de test suivantes ont été exécutées sans problème :

- testConstructor s'assure que le constructeur de **SourceFixe** génère correctement une séquence d'informations à partir d'un message binaire donné en paramètre ;
- testInformationEmise vérifie que l'information émise est identique à l'information générée ;
- testMessageStringWithOnlyOnes valide que le constructeur de **SourceFixe** génère correctement une séquence de true pour un message binaire composé uniquement de 1 ;
- testMessageStringWithOnlyZeros s'assure que le constructeur génère une séquence de false pour un message binaire composé uniquement de 0 ;
- testMessageWithAlternatingBits vérifie que la génération de séquences avec des bits alternés (ici 101010) est correcte ;

- testEmptyMessageString s'assure que la génération d'une séquence à partir d'un message vide produit une information vide.

3.4.4. Tests de la classe SourceAleatoire

Les tests liés à la classe **SourceAleatoire** visent à vérifier la génération correcte de séquences aléatoires d'informations. Encore une fois, tous les tests décrits ci-dessous se sont déroulés sans accroc.

- ✓ SourceAleatoireTest
 - ✓ testSequenceWithSeedReproducibility
 - ✓ testConstructorWithRandomBits
 - ✓ testConstructorWithSeed
 - ✓ testDifferentSeedsProduceDifferentSequences
 - ✓ testZeroLengthSequence
 - ✓ testInformationEmise

Figure 39 - Résultat des tests pour la classe SourceAleatoire

- testConstructorWithRandomBits vérifie que le constructeur de **SourceAleatoire**, lorsqu'il est appelé sans seed, génère une séquence aléatoire de bits de la bonne longueur ;
- testConstructorWithSeed vérifie que l'utilisation d'un seed permet de générer une séquence aléatoire reproducible ;
- testInformationEmise vérifie que la séquence d'informations générée par **SourceAleatoire** est correctement émise ;
- testZeroLengthSequence s'assure que la génération d'une séquence de longueur zéro ne produit pas d'erreur et retourne une séquence vide ;
- testSequenceWithSeedReproducibility valide que deux instances de **SourceAleatoire** initialisées avec le même seed produisent des séquences identiques ;
- testDifferentSeedsProduceDifferentSequences vérifie que l'utilisation de seeds différents produit des séquences différentes.

3.4.5. Tests de la classe TransmetteurParfait

Les tests exécutés dans la classe **TransmetteurParfaitTest** valident le fait que la transmission des informations se fait sans altération. La figure 40 montre la liste des méthodes de tests dont nous nous servons pour vérifier cela. Les objectifs de chacun de ces tests sont définis ci-dessous.

- ✓ TransmetteurParfaitTest
 - ✓ testRecevoirWithNullInformationThrowsException
 - ✓ testRecevoirAndEmettreWithValidInformation
 - ✓ testRecevoirEmptyInformation
 - ✓ testEmettreWithMultipleDestinations

Figure 40 - Résultat des tests pour la classe TransmetteurParfait

- testRecevoirAndEmettreWithValidInformation vérifie que la méthode `recevoir()` traite correctement les informations et que la méthode `emettre()` transmet ces informations de manière parfaite (sans altération) vers le récepteur ;
- testEmettreWithMultipleDestinations s'assure que la méthode `emettre()` fonctionne correctement avec plusieurs destinations connectées (récepteur + sondes), en envoyant la même information à toutes les destinations ;

- testRecevoirWithNullInformationThrowsException vérifie que l'envoi d'une information nulle à la méthode `recevoir()` lève une exception `InformationNonConformeException` ;
- testRecevoirEmptyInformation s'assure que l'émission d'une information vide ne provoque pas d'erreur et que les destinations reçoivent une information vide.

3.4.6. Tests de la classe DestinationFinale

Cette sixième et dernière classe de tests a pour objectif de vérifier la capacité de la destination de notre chaîne de transmission à recevoir et stocker correctement les informations, ainsi que de gérer les erreurs potentielles. Nous pouvons voir sur la figure 41 que les quatre types de tests que nous exécutons pour valider le bon fonctionnement de **DestinationFinale** passent tous sans lever d'erreurs inattendues.

- ✓ DestinationFinaleTest
 - ✓ testRecevoirSingleElementInformation
 - ✓ testRecevoirValidInformation
 - ✓ testRecevoirWithNullInformationThrowsException
 - ✓ testRecevoirEmptyInformation

Figure 41 - Résultat des tests pour la classe DestinationFinale

- testRecevoirValidInformation vérifie que la méthode `recevoir()` de la classe `DestinationFinale` stocke correctement une séquence d'informations valides. Le test compare les informations reçues avec celles envoyées par l'élément d'avant pour s'assurer qu'elles sont identiques ;
- testRecevoirWithNullInformationThrowsException s'assure que l'envoi d'une information nulle à la méthode `recevoir` déclenche une exception `InformationNonConformeException`;
- testRecevoirEmptyInformation valide que la réception d'une information vide ne déclenche pas d'erreur, et que l'information stockée est bien vide ;
- testRecevoirSingleElementInformation vérifie que la méthode `recevoir()` gère correctement une séquence contenant un seul élément.

3.4.7. Couverture du code Java avec Emma

Comme expliqué précédemment, nous nous servons de l'outil Emma afin de voir le pourcentage de notre code qui est couvert par les tests présentés dans les sous-parties précédentes. La figure 42 affichée ci-dessous montre, pour chaque classe de tests, le nombre de méthodes couvertes par les tests JUnit.

Element	Class, %	Method, %	Line, %
sources	100% (1/1)	40% (2/5)	50% (5/10)
Source	100% (1/1)	40% (2/5)	50% (5/10)

Element	Class, %	Method, %	Line, %
transmetteurs	100% (1/1)	100% (2/2)	100% (7/7)
TransmetteurParfait	100% (1/1)	100% (2/2)	100% (7/7)

Element	Class, %	Method, %	Line, %
destinations	100% (1/1)	100% (1/1)	100% (3/3)
DestinationFinale	100% (1/1)	100% (1/1)	100% (3/3)

Element ^	Class, %	Method, %	Line, %
modulation	100% (4/4)	93% (15/16)	90% (87/96)
emetteurs	100% (2/2)	100% (6/6)	92% (47/51)
Emetteur	100% (2/2)	100% (6/6)	92% (47/51)
recepteurs	100% (1/1)	100% (4/4)	95% (23/24)
Recepteur	100% (1/1)	100% (4/4)	95% (23/24)

Figure 42 - Quantité de notre code couvert grâce aux tests JUnit

Nous pouvons remarquer que sur les 28 méthodes contenues dans les principales classes de notre projet 24 sont concernées par les tests JUnit. Celles qui ne le sont pas sont souvent des méthodes qui ne servent qu'à afficher ou renvoyer des valeurs, donc effectuer des tests sur ces dernières présente peu d'intérêt.

3.4.8. Modification du script runTests

Nous avons également des tests en ligne de commande qui s'exécutent depuis notre script *runTests*. Ces tests nous permettent de nous assurer que le programme fonctionne correctement avec des paramètres différents. La partie du script *runTests* modifiée pour accueillir ces nouveaux tests est présentée dans la figure ci-dessous.

```
# Liste des scénarios de tests à exécuter
test_cases=(
    "-mess 10"                                # Test avec un message aléatoire de 10 bits
    "-mess 1010101"                            # Test avec un message fixe
    "-mess 110011 -code NRZ"                   # Test avec codage NRZ
    "-mess 0010011 -code RZ"                   # Test avec codage RZ
    "-aMax 5 -aMin -5 -code NRZT"             # Test avec des amplitudes extrêmes
    "-mess 50 -seed 1234"                      # Test avec une seed spécifique
    "-mess 1001110011100110001111000011110000111100001111000011110000000000" # Message long
    "-mess 1000000"                            # Test avec un message d'un million de bits
)

# Si on n'est pas dans un pipeline GitLab (la variable d'environnement CI n'est pas définie)
if [ -z "$CI" ]; then
    test_cases+=(
        "-s -code NRZT -aMin -4 -aMax 4"
        "-s -seed 27 -code RZ -aMin 0 -aMax 6"
        "-s -mess 12 -seed 1234 -code NRZ -aMin -5 -aMax 5"
        "-s -mess 01101101110 -code NRZT"
    )
fi
```

Figure 43 - Modification du script *runTests* pour l'itération 2

Nous avons séparé les tests visuels (avec le paramètre *-s*) des tests non visuels afin que la pipeline GitLab puisse s'exécuter correctement. Ainsi, l'exécution du script ne provoque pas un crash du logiciel ou une erreur lors du push vers le repository.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Étape 5 : Codage de canal</p>	<p>Année scolaire 2024-2025</p> <p>Page : 34/68</p>
---	---	--

3.5. Bilan de l’itération

Cette seconde itération nous a permis de franchir une étape importante dans la simulation de la transmission d'un signal à travers une chaîne de transmission dite parfaite. Les principaux objectifs étaient de mettre en place un émetteur et un récepteur capables de convertir des signaux numériques en signaux analogiques, et vice versa, tout en intégrant trois types de codage binaire : NRZ, RZ, et NRZT.

Nous avons réussi à implémenter les différentes classes nécessaires, notamment celles de l'émetteur, du récepteur, ainsi que la classe abstraite *Modulateur*, qui permet une gestion optimisée des paramètres communs à ces blocs. Cette abstraction a simplifié la gestion des codes binaires et a permis d'éviter la duplication de code.

Les résultats obtenus lors des simulations montrent que notre système fonctionne comme attendu, sans erreurs de transmission, puisque cette seconde étape comprenait l'absence de bruit au sein du canal de transmission. Nous avons validé les différentes simulations avec les sondes analogiques et logiques, et les signaux captés correspondent aux théories liées à chaque type de codage. Le code NRZT, avec ses transitions douces, se montre particulièrement adapté pour les transmissions analogiques.

Les tests effectués à l'aide de **JUnit** et la vérification de la couverture de code avec **Emma** ont confirmé la robustesse de notre implémentation. Nous avons également utilisé **EasyMock** pour simuler certaines composantes, ce qui a permis d'effectuer des tests plus exhaustifs.

En résumé, cette itération a non seulement permis de passer à une gestion des signaux analogiques, mais aussi de poser les bases pour la prochaine étape, où nous introduirons du bruit dans le canal de transmission. Le système est désormais prêt pour simuler des environnements de transmission plus réalistes et complexes.

IV. Itération 3 : Transmission non idéal avec canal bruité de type « gaussien »

4.1. Cahier des charges

Cette troisième itération ajoute un changement majeur dans notre chaîne de transmission. En effet, jusqu'à présent nous étudions une transmission parfaite sans qu'aucun bruit ne vienne perturber le signal émis depuis la source vers la destination. Désormais, notre client demande que l'ajout de bruit au sein du canal de transmission, comme le montre la figure 44.

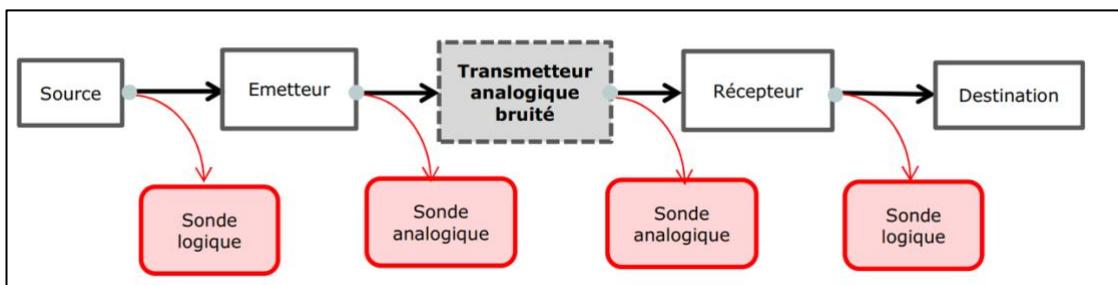


Figure 44 - Modélisation de la chaîne de transmission à l'étape 3

Nous pouvons remarquer sur la figure ci-dessus qu'à part le transmetteur analogique qui n'est plus parfait, notre chaîne de transmission reste la même en tous points. Mais l'étude de ce nouveau transmetteur est cruciale pour que notre logiciel réponde aux nouvelles demandes du client.

Dans un premier temps il nous est demandé que le bruit qui perturbe le signal soit un bruit **blanc additif gaussien**. Chacun de ces mots a une signification bien particulière :

- Un bruit **blanc** signifie que toutes les valeurs de ce bruit sont indépendantes et ont une puissance égale sur tout le spectre de fréquence ;
- Le fait qu'il soit **additif** indique simplement que ce bruit viendra s'ajouter au signal d'origine qui part de la source de notre structure ;
- Enfin, un bruit dit « **gaussien** » a pour particularité que la majorité des valeurs se situent autour de la moyenne, tout en suivant une distribution normale (ou gaussienne).

Maintenant que nous avons étudié le type de bruit à intégrer à notre système, il nous reste à déterminer comment le générer. Pour ce faire, nous utilisons la formule suivante :

$$b(n) = \sigma_b \sqrt{-2\ln(1 - a_1(n))} \cos(2\pi a_2(n)) \quad \begin{aligned} a_1(n) &\sim \mathcal{U}[0, 1[\text{ (loi uniforme)} \\ a_2(n) &\sim \mathcal{U}[0, 1[\end{aligned}$$

Dans la formule affichée ci-dessus :

- $b(n)$ est la valeur du bruit blanc gaussien à l'indice n ;
- σ_b représente l'écart-type du bruit gaussien ;
- $a_1(n)$ et $a_2(n)$ les variables aléatoires uniformes comprise entre 0 et 1 qui permettent de créer les valeurs suivant une distribution normale.

L'objectif de cette troisième étape est donc d'ajouter le bruit blanc gaussien présenté précédemment à notre signal d'origine provenant du bloc émetteur de notre système. Grâce à cela, chaîne de transmission ressemblera plus à ce que l'on peut retrouver en situation réelle. Le rôle du récepteur placé au bout du canal sera donc plus important puisqu'il devra, en plus de décoder le signal analogique, supprimer le bruit qui aura été ajouté afin de retrouver le signal d'origine, avec le moins d'erreur possible.

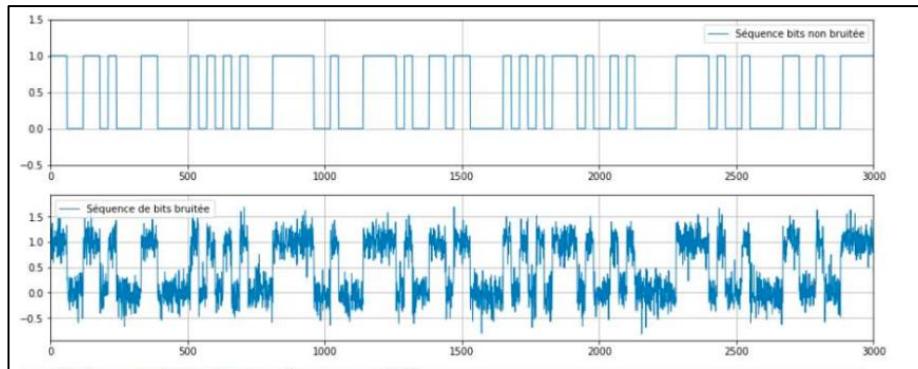


Figure 45 - Exemple de résultats attendus à la fin de la troisième étape pour un SNR valant 10 dB

Afin de réaliser différentes simulations, l'utilisateur final pourra préciser, en dB, le niveau de bruit qu'il souhaite ajouter au canal de transmission. Par défaut ce sera un transmetteur analogique parfait qui sera utilisé. La valeur qui nous intéressera le plus sera bien sur le taux d'erreur binaire (TEB). Pour rappel, ce calcul compare la chaîne émise par la source, et celle reçue par la destination. Dans le cas d'un transmetteur parfait ou quasi-parfait le TEB vaut 0 puisqu'aucune erreur n'est détecté. Il sera donc intéressant de voir, pour chaque code binaire, à partir de quel seuil de bruit des erreurs commencent à apparaître.

Enfin, un aspect essentiel de l'analyse du canal bruité est le rapport signal-bruit (SNR) par bit, défini ici comme $\frac{E_b}{N_0}$, où E_b représente l'énergie par bit, et N_0 la densité spectrale du bruit. Ce rapport est crucial pour évaluer les performances de la transmission car il détermine directement le taux d'erreur binaire (TEB). En effet, plus le SNR par bit est élevé, moins le bruit a d'influence sur le signal, réduisant ainsi les erreurs lors de la transmission. Inversement, un faible SNR augmente le risque d'erreurs. Ce rapport est particulièrement important pour ajuster la puissance du signal en fonction du niveau de bruit, permettant d'optimiser la qualité de la transmission dans des conditions bruitées. Dans notre simulation, le SNR par bit sera contrôlé pour observer l'impact du bruit blanc gaussien sur le taux d'erreur binaire. Pour obtenir cette valeur en dB nous utilisons la formule suivante : $(\frac{E_b}{N_0})_{dB} = 10 \log_{10}(\frac{P_s \times N}{2\sigma_b^2})$ avec :

- P_s la puissance du signal ;
- N le nombre d'échantillons par bit, qui est un des paramètres intégrés à notre logiciel lors de l'étape précédente ;
- σ_b^2 la puissance du bruit.

4.2. Implémentation des fonctionnalités

Comme lors de la seconde itération, les nouvelles fonctionnalités présentées dans la partie précédente nous obligent à modifier notre code Java afin que notre logiciel les prenne en compte. L'élément de notre système qui gère l'addition de bruit à notre signal d'origine est le transmetteur. Il nous faudra donc ajouter une classe dédiée à ce transmetteur gaussien. L'étude du cahier des charges nous a également montrée que le récepteur aura pour nouveau rôle la suppression de ce même bruit, afin de retransmettre à la destination le signal d'origine. Enfin, la classe mère de notre logiciel nommée **Simulateur** devra elle aussi être revue afin de traiter les nouveaux paramètres liés à la troisième étape de ce projet.

4.2.1. Création de la classe TransmetteurGaussian

Pour rappel, notre logiciel contenait une classe **TransmetteurParfait** qui s'occupait simplement de retransmettre le signal qu'il recevait de l'émetteur. Cette classe hérite de la classe abstraite *Transmetteur*. Nous avons donc simplement ajouté une nouvelle classe **TransmetteurGaussian**, qui hérite elle aussi de *Transmetteur*. Le diagramme de classe regroupant ces trois classes et affiché ci-dessous.



Figure 46 - Diagramme de classe lié à l'itération 3

La figure 46 nous permet de remarquer que la nouvelle classe **TransmetteurGaussian** est bien plus complexe que son homonyme **TransmetteurParfait**. En effet, en plus du fait d'émettre et de recevoir des informations, notre nouvelle classe doit également ajouter un signal de type bruit blanc gaussien au signal émis par l'émetteur. Pour faire cela, on génère un signal correspondant au bruit blanc gaussien. On parcourt ensuite le signal reçu et, pour chaque échantillon de ce signal, on ajoute une valeur du signal bruité. Ainsi, on se retrouve avec notre signal d'origine mélangé au bruit blanc gaussien. C'est cette addition de signal que le nouveau transmetteur envoie vers le récepteur du système.

4.2.2. Nouvelle méthode de réception du signal bruité

Le second changement important que nous avons implémenté à notre code Java concerne le récepteur. Lors de la seconde étape, nous avions décidé de regarder pour chaque bit les valeurs minimales et maximales présentes. Nous les comparions ensuite à l'amplitude déterminée par l'utilisateur du logiciel et cela nous permettait de déterminer si le bit passé en entrée de la chaîne de transmission était un « 0 » ou un « 1 ». Or, l'ajout de bruit bloque

totalelement cette méthode puisque qu'une valeur déjà maximale peut être encore augmentée, et une valeur correspondant à l'amplitude minimale peut se retrouver diminuer.

Nous avons donc dû opter pour une toute nouvelle méthode, basée cette fois sur la valeur moyenne de chaque bit. Pour les codages binaires NRZT et RZ nous incrémentons un compteur avec toutes les valeurs du signal bruité qui compose le bit étudié. Une fois que nous avons parcouru le bit en entier nous calculons la puissance moyenne du bit.

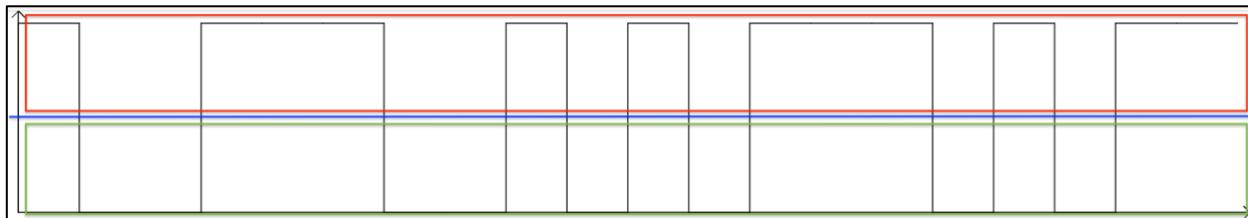


Figure 47 - Méthode de réception d'un signal bruité

Si la puissance moyenne calculé se situe dans la zone rouge, c'est-à-dire que

$P \geq \frac{aMin+aMax}{2}$, le signal débruité vaudra « 1 ». Si, à l'inverse, la puissance se situe dans la zone verte, c'est-à-dire que $P \leq \frac{aMin+aMax}{2}$, la valeur finale vaudra « 0 ». Pour le codage RZ binaire nous ne prenons que les échantillons présents dans le deuxième tier du bit, puisqu'il vaut 0 aux deux extrémités.

4.2.3. Intégration de l'ajout de bruit à la commande permettant d'utiliser le logiciel

Pour cette troisième étape, la modification de la classe **Simulateur** consistait simplement à prendre en compte du nouveau paramètre snrpb spécifié par notre client.

- Si le paramètre est spécifié, nous utilisons une transmission analogique bruitée. La valeur donnée par l'utilisateur correspond au rapport signal à bruit par bit en dB. Pour ce faire nous utilisons la classe **TransmetteurGaussien** présentée précédemment ;
- Si aucun bruit n'est spécifié, le signal transitant dans la chaîne de transmission n'est pas bruité. Dans ce cas, nous nous servons de la classe **TransmetteurParfait**.

4.3. Résultats obtenus

Maintenant que les fonctionnalités incluses dans cette troisième étape font partie intégrante de notre code, nous pouvons lancer des simulations qui ajoutent du bruit au signal d'origine au sein du canal de propagation. Pour faire cela, il nous suffit d'exécuter la classe Simulateur en faisant varier l'amplitude de notre signal, le bruit que nous ajoutons, le type de codage binaire utilisé...

4.3.1. Simulations avec les paramètres par défaut

Pour commencer nous décidons d'utiliser les paramètres par défaut afin de comparer seulement les trois codes binaires mis à notre disposition. Nous nous retrouvons donc avec les valeurs suivantes :

- Une amplitude allant de 0.0 à 1.0
- Chaque bit est composé de 30 échantillons
- Le message envoyé est un message aléatoire d'une longueur de 100 bits

Comme attendu, quand nous utilisons un SNR relativement élevé (15 dB ou plus), aucune erreur n'est détectée à la sortie de la chaîne de transmission, quel que soit le codage binaire utilisé.

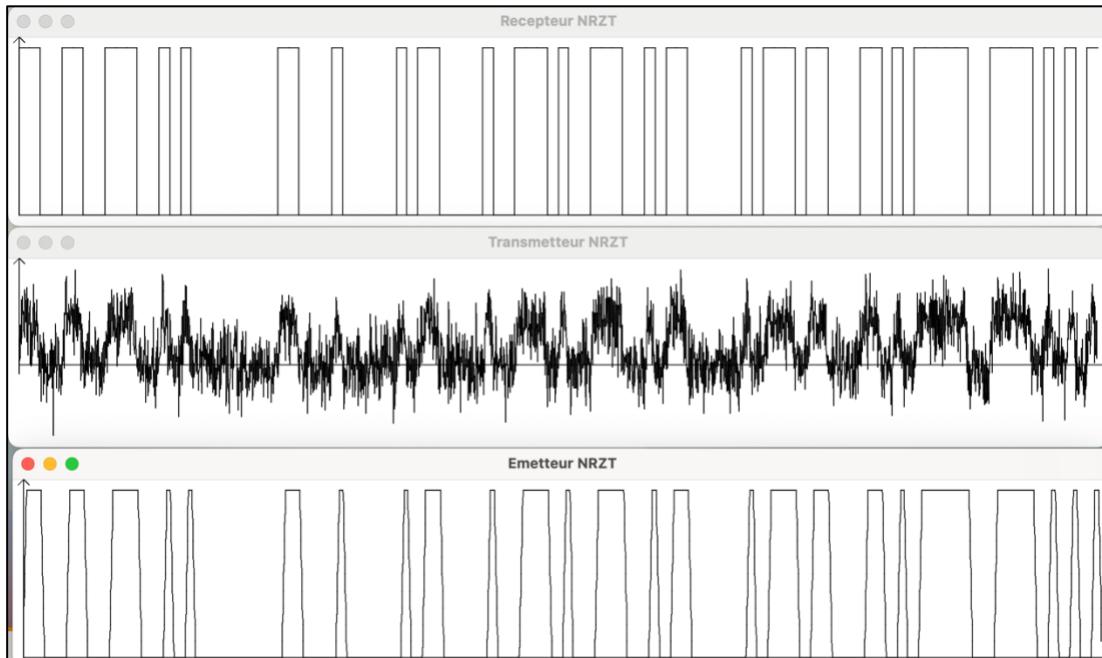


Figure 48 - Simulation pour un code NRZT avec un SNR de 15 dB

La figure ci-dessus montre le résultat pour le codage NRZT binaire, mais les sondes renvoient des résultats très similaires pour les deux autres codages, à l'exception de la mise en forme qui varie. Le seul paramètre que nous modifions pour cette première batterie de résultats est le rapport signal à bruit. En diminuant cette valeur, nous observons que des erreurs sont introduites dès que le SNR atteint environ 12 dB pour les codages binaires NRZT et NRZ. Le code RZ binaire réagit différemment puisqu'avec un SNR valant 9 dB ou plus le TEB vaut toujours 0. Les figures ci-dessous illustrent ce commentaire. Elles montrent également que le SNR par bit détecté par le logiciel grâce à la formule $(\frac{Eb}{N_0})_{dB} = 10 \log_{10}(\frac{P_s \times N}{\sigma_B^2})$ est très proche de la valeur rentrée en paramètre.

```
java Simulateur -mess 100 -form NRZ -snrpb 12 java Simulateur -mess 100 -form NRZT -snrpb 12 java Simulateur -mess 100 -form RZ -snrpb 9
=> TEB : 0.05 => TEB : 0.03 => TEB : 0.0
=> SNR par bit demandé : 12.0 => SNR par bit demandé : 12.0 => SNR par bit demandé : 9.0
=> SNR par bit réel obtenu : 11.761269 => SNR par bit réel obtenu : 11.769749 => SNR par bit réel obtenu : 8.913521
```

Figure 49 – TEB et SNR pour les codages binaires NRZ, NRZT et RZ

Pour rappel, lorsque le récepteur a affaire à un signal utilisant le codage RZ, il ne regarde que le tiers central de chaque bit. Ainsi, il y a trois fois moins d'échantillons qui sont analysés comparés aux deux autres codages. Nous pouvons donc penser qu'avec un nombre d'échantillons réduit à analyser les performances seraient meilleures ou, à l'inverse, utiliser un codage RZ dans des conditions moins favorables augmenterait le TEB pour un même rapport signal à bruit.

Par ailleurs, nous restons très loin du pire TEB théorique que l'on pourrait avoir qui est 0.5. En effet, avoir un taux d'erreur binaire valant 0.5 signifie que la moitié des bits du signal reçu sont faux mais nous ne pouvons pas savoir lesquels. Il est même préférable d'avoir un TEB valant par exemple 0.7, puisqu'il nous suffirait d'inverser le message reçu. Ainsi, le nouveau TEB vaudrait $1 - 0.7 = 0.3$.

4.3.2. Modification de la taille du message

La première façon de vérifier notre hypothèse est de changer la taille du message émis par la source de notre système. Nous augmentons la taille de ce dernier pour avoir une chaîne de 10000 booléens en entrée de la chaîne de transmission. Cette taille correspond à un e-mail qui serait envoyé sans pièce jointe, donc un exemple typique de signal qui transite en permanence.

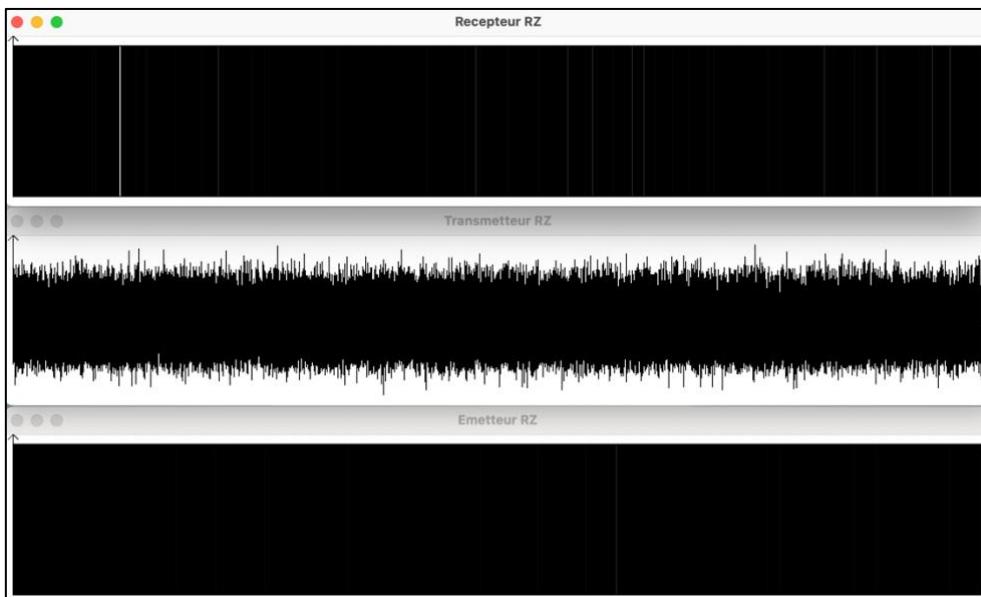


Figure 50 – Simulation pour un code RZ avec un SNR de 9 dB et un message émis de 10000 booléens

```
java Simulateur -mess 10000 -form RZ -snrpb 9 -s
=> TEB : 0.003
=> SNR par bit demandé : 9.0
=> SNR par bit réel obtenu : 8.994322
```

Figure 51 - TEB et SNR pour un message de 10000 bits utilisant le code RZ binaire

Bien que le code RZ reste dans ce cas encore plus efficace que NRZ et NRZT pour un même SNR (environ 0.07 pour un SNR de 9 dB), nous pouvons remarquer que son taux d'erreur binaire n'est plus nul. La théorie présentée dans le chapitre précédent s'avère donc réelle.

4.3.2. Comparaison entre la puissance de bruit obtenue et la variance

Le deuxième point que nous pouvons analyser concerne la puissance moyenne du bruit que nous ajoutons au signal d'origine et son lien avec la variance. En effet, nous pouvons remarquer sur les figures ci-dessous que, pour l'ensemble des simulations que nous exécutons, ces deux attributs ont des valeurs extrêmement proches.

```
java Simulateur -mess 500 -form RZ -snrpb 3 -ampl 0 5 java Simulateur -seed 127834 -form NRZT -snrpb -9 -ampl -4 4
=> TEB : 0.098                                     => TEB : 0.35
- Nombre de bits de la séquence : 500           - Nombre de bits de la séquence : 100
- Nombre d'échantillons par bit : 30            - Nombre d'échantillons par bit : 30
=> Puissance moyenne du bruit : 32.3138       => Puissance moyenne du bruit : 1503.0925
=> Variance : 32.827763                         => Variance : 1492.2595
=> Rapport signal-sur-bruit (S/N, en dB) : 3.0685327 => Rapport signal-sur-bruit (S/N, en dB) : -9.031413
```

Figure 52 - Corrélation entre la variance et la puissance moyenne du bruit

Notre bruit étant un signal de type bruit blanc gaussien, sa moyenne est nulle. Dans ce cas l'espérance de x^2 (le carré de l'amplitude) est directement égale à la variance σ^2 . Autrement dit, la puissance moyenne du bruit est égale à sa variance, car la moyenne est nulle et la variance mesure justement l'écart au carré des valeurs autour de cette moyenne. C'est pour cette raison que ces deux valeurs sont très proches.

4.3.3. Mise en relation du TEB et du SNR

Nous avons vu précédemment que, pour les trois codes binaires étudiés, plus le SNR en dB était faible, plus le nombre d'erreurs détectées à la sortie du système était élevé. Afin de vérifier cela, nous avons tracé le taux d'erreur binaire retourné par la chaîne de transmission pour un SNR différent à chaque itération. Nous avons lancé 75 simulations avec un SNR qui variait de 15 à -10 dB et un message de 100000 bits à chaque itération.

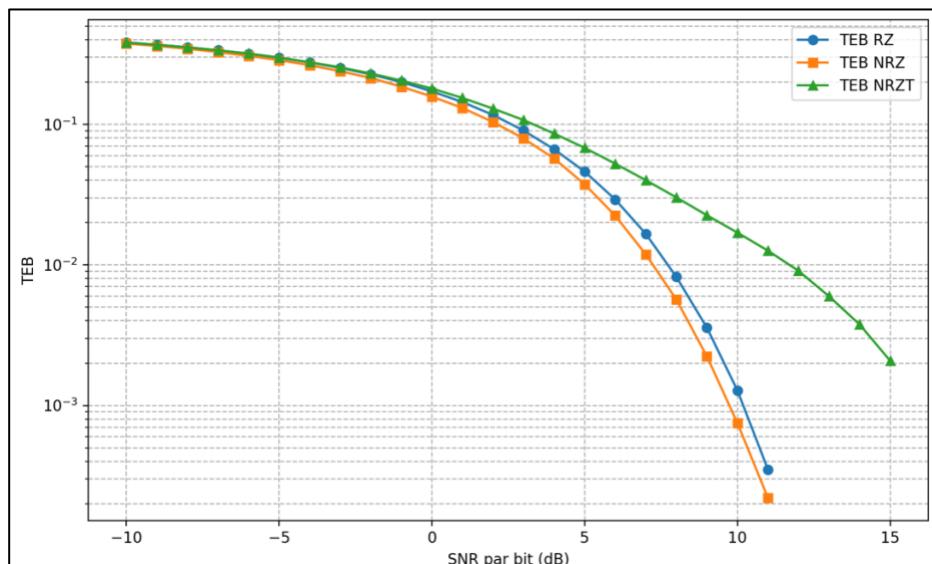


Figure 53 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (pratique)

Nous avons décidé de représenter le TEB par rapport au SNR sur une échelle semi-logarithmique, afin de pouvoir observer la variation du TEB sur plusieurs ordres de grandeur. De plus, cela nous permet de comparer nos résultats avec ceux connus dans des exemples de la vie courante. Par exemple, dans le cas d'un signal vidéo le TEB demandé est d'environ 10^{-9} , c'est-à-dire une erreur sur un bit pour un milliard de bit envoyé par la source.

En étudiant le graphe de la figure 53 nous pouvons remarquer que :

- Comme nous l'avions remarqué, le TEB diminue au fur et à mesure que le SNR augmente pour les trois types de modulation ;
- Le code NRZT semble être le moins performant pour un rapport signal à bruit élevé (de 5dB à 15dB). En effet, nous pouvons remarquer que le taux d'erreur binaire n'est jamais mieux que $2 \cdot 10^{-3}$;
- Pour ces mêmes valeurs de SNR, les codes binaires NRZ et RZ ont des performances très similaires, bien que le code NRZ semble un tout petit peu mieux ;
- Cependant, pour des valeurs de SNR en dB négatives, nous pouvons observer que les trois codes binaires se valent.

Ces différences entre les codes binaires peuvent être dû à la mise en forme qui n'est pas la même ou au fait que nous effectuons nos tests avec un nombre de bits relativement limité (100000 dans notre cas contre parfois des millions, voir des milliards de bits, dans des systèmes de communications modernes). En fonction des paramètres choisis, l'amplitude choisie peut également avoir un impact sur les résultats obtenus.

4.3.4. Comparaison avec les études théoriques

La partie précédente nous a montré le taux d'erreur binaire en fonction du SNR avec notre logiciel. Pour valider nos résultats, nous devons comparer ceux obtenus sur la figure 54 avec les courbes TEB par rapport au SNR théoriques pour le code NRZ binaire. Avec l'aide des équipes pédagogiques nous avons déterminé la formule suivante pour obtenir le TEB théorique en fonction du SNR pour le code NRZ binaire :

$$- \text{ NRZ binaire} \rightarrow TEB = \frac{1}{2} \times erfc\left(\sqrt{\frac{Eb}{N_0}}\right) \text{ avec } \frac{Eb}{N_0} \text{ en linéaire}$$

Nous visualisons ensuite la courbe résultante sur le graphique ci-dessous. Nous avons choisi d'utiliser également une échelle semi-logarithmique afin de pouvoir comparer plus simplement les résultats théoriques avec ceux obtenus grâce à notre logiciel.

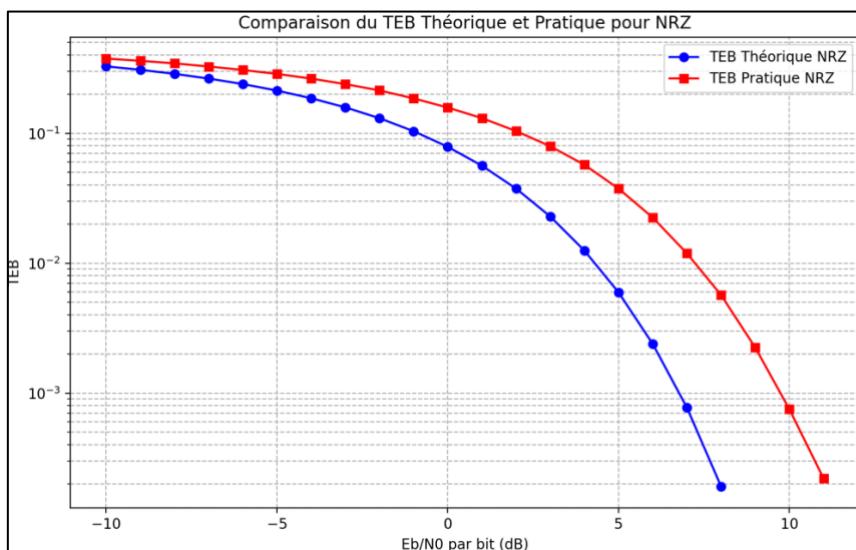


Figure 54 - Graphe montrant le TEB en fonction du SNR par bit ($\frac{Eb}{N_0}$) dB pour le code binaire NRZ (théorie et pratique)

Grâce à la figure 54 nous pouvons comparer, pour le code NRZ binaire, les performances de notre simulateur aux résultats théoriques. Le premier élément que nous pouvons comparer est l'allure des courbes. Nous observons qu'elle est la même dans les deux cas, ce qui nous rassure quant aux résultats obtenus en pratique. Si nous regardons ensuite plus en détail les résultats obtenus, nous pouvons voir que pour un SNR par bit relativement élevé ($\geq 5\text{dB}$) il y a un écart d'environ 3dB entre nos résultats et la théorie. Plus on se rapproche d'un SNR par bit nul, plus les résultats obtenus grâce au simulateur se rapprochent de ceux obtenus avec la méthode ERF Complémentaire. Enfin, avec un $(\frac{E_b}{N_0})\text{dB}$ négatif, les deux courbes sont très proches. Notre simulateur paraît donc être le plus proche de la réalité pour un SNR par bit très faible.

Cette différence de performances entre notre simulateur et la théorie peut être due à plusieurs facteurs. Tout d'abord, la modélisation de notre bruit est faite de manière pseudo-aléatoire. Cette génération n'est pas parfaite et pourrait légèrement différer du bruit idéal théorique, introduisant ainsi des divergences dans les résultats. Ces écarts pourraient également être dus au nombre d'échantillons que nous utilisons, ou à la manière dont nous émettons et nous réceptionnons l'information. Malheureusement nous n'avons pas réussi à obtenir un résultat plus proche de la théorie dans la suite de ce projet.

4.3.5. Vérification du type de bruit ajouté

Enfin, nous avons voulu vérifier que le bruit ajouté par le canal de transmission est bien gaussien. Pour rappel, les valeurs d'un bruit gaussien doivent suivre une loi normale (ou gaussienne) et être, pour la majorité d'entre elles, situées autour de la moyenne. Pour vérifier cela nous avons pris toutes les valeurs de bruit pour tous les échantillons d'un signal (plusieurs dizaines de milliers) et nous les avons mis sous la forme d'un histogramme. Cet histogramme est affiché en figure 55.

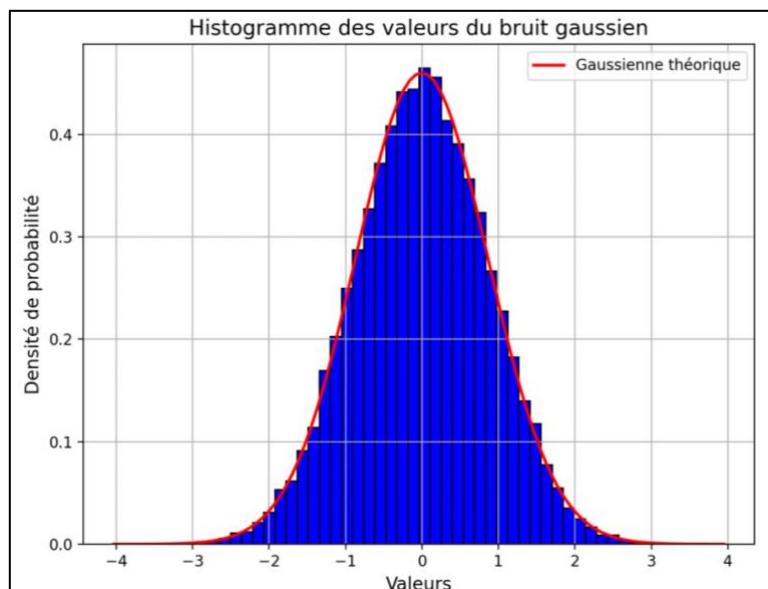


Figure 55 - Histogramme des valeurs du bruit blanc gaussien additif ajouté dans le canal de transmission

Nous pouvons observer sur la figure 55 que le bruit additif qui est produit par notre système ressemble parfaitement à une distribution gaussienne. Pour montrer cela nous avons afficher en rouge la densité spectrale de puissance (DSP) de notre bruit. Puisqu'il est gaussien, l'aire sous cette fonction vaut 1. Nous avons ensuite dimensionné notre histogramme pour que les valeurs maximales atteignent le maximum de la DSP. Nous pouvons observer que les valeurs du bruit remplissent parfaitement l'aire définie par la DSP. Nous en déduisons que le bruit créé aléatoirement par le canal de transmission suit bien une loi gaussienne.

4.4. Tests effectués

Notre logiciel incluant une nouvelle classe et de nouvelles fonctionnalités, nous avons ajouté de nouveaux tests en utilisant les mêmes outils que précédemment (JUnit 4, Emma et EasyMock).

4.4.1. Tests de la classe TransmetteurGaussien

Les tests effectués sur cette classe permettent de vérifier que la fonction de bruitage du signal, intégrée dans le transmetteur de notre système, fonctionne comme prévu. La figure ci-dessous montre que les quatre tests ont été exécutés sans problème.

✓ TransmetteurGaussienTest (tests)
✓ testEmettreAvecValeursNegatives
✓ testAjoutBruit
✓ testRecevoirInformationVide

Figure 56 - Résultats des tests pour la classe TransmetteurGaussien

La figure 56 liste les tests qui sont effectués sur la classe **TransmetteurGaussien** :

- testAjoutBruit vérifie que l'information bruitée est correctement reçue par la destination ;
- testEmettreAvecValeursNegatives teste la réception d'un signal bruité contenant des valeurs négatives ;
- testRecevoirInformationVide s'assure que lorsqu'un signal vide est émis, un signal vide est également réceptionné à la sortie du système.

En plus des tests JUnit 4 présentés ci-dessus, l'outil de couverture Emma nous permet de regarder à quel point la classe **TransmetteurGaussien** est couverte par des tests.

Element ^	Class, %	Method, %	Line, %	Branch, %
▼ transmetteurs	66% (2/3)	63% (12/19)	72% (42/58)	60% (12/20)
© Transmetteur	100% (1/1)	50% (3/6)	66% (6/9)	100% (0/0)
© TransmetteurGaussien	100% (1/1)	81% (9/11)	85% (36/42)	75% (12/16)
© TransmetteurParfait	0% (0/1)	0% (0/2)	0% (0/7)	0% (0/4)

Figure 57 - Couverture du package transmetteurs grâce à l'outil Emma

Nous pouvons remarquer grâce à la figure 57 que plus de 80% des méthodes de cette classe sont couvertes par les tests JUnit. Les deux méthodes qui ne sont pas testées sont :

- une méthode qui ne fait que retourner des valeurs ;
- un constructeur simplifié dont la version complexe a déjà été testée.

4.4.2. Tests du TEB

La seconde série de tests dédiée à cette itération ne concerne cette fois pas une classe en particulier, mais le calcul du taux d'erreur binaire. Ce résultat étant essentiel pour analyser nos résultats, nous devons être sûrs qu'il est bien renvoyé. Pour tester cela nous avons créé une classe TebTest qui effectue le test suivant :

```
✓ TebTest (tests)
  ✓ testSimulationMultipleTimes
```

Figure 58 - Résultat de la classe TebTest

Le test `testSimulationMultipleTimes` lance une simulation pour les codages binaires NRZT, RZ et NRZ et vérifie qu'un résultat est correctement renvoyé à chaque fois que le signal a atteint sa destination.

4.4.3. Modification du script runTests

Comme pour les étapes précédentes, nous effectuons toute une série de tests grâce au script `runTests1`. Les nouveaux tests que ce script comprend nous permettent d'effectuer des simulations en combinant tous les paramètres que notre logiciel gère. La partie du script qui comprend les tests associés à la troisième étape de notre projet est présentée ci-dessous.

```
# Liste des scénarios de tests à exécuter
test_cases=(
    "-mess 10"                      # Test avec un message aléatoire de 10 bits
    "-mess 1010101"                  # Test avec un message fixe
    "-mess 0010011 -form NRZ"        # Test avec codage NRZ
    "-mess 0010011 -form RZ"         # Test avec codage RZ
    "-mess 0010011 -form NRZT"       # Test avec codage NRZT
    "-ampl -5 -form NRZT"           # Test avec des amplitudes
    "-mess 50 -seed 1234"            # Test avec une seed spécifique
    "-mess 1001100111001100011100001110000111000011100001111000011111000000000" # Message long
    "-mess 1000000"                 # Test avec un message d'un million de bits

    "-mess 1010101 -snrb -20"        # SNR par bit faible
    "-mess 110011 -snrb 50"          # SNR par bit élevé

    "-mess 200 -form NRZ -snrb 30"   # SNR par bit avec format NRZ
    "-mess 100 -form NRZ -snrb 0"    # SNR par bit avec format NRZ
    "-mess 10101000 -form NRZ -snrb -30" # SNR par bit avec format NRZ
    "-mess 1000 -form RZ -snrb -30"  # SNR par bit avec format RZ
    "-seed 1234 -form RZ -snrb 0"    # SNR par bit avec format RZ
    "-seed 812873 -form RZ -snrb 30" # SNR par bit avec format RZ
    "-mess 23 -form NRZT -snrb -30"  # SNR par bit avec format NRZT
    "-mess 2000 -form NRZT -snrb 0"   # SNR par bit avec format NRZT
    "-mess 1010000101 -form NRZT -snrb 30" # SNR par bit avec format NRZT

    "-ampl -3 3 -form NRZT -snrb 30" # SNR par bit avec amplitudes personnalisées
)

# Si on n'est pas dans un pipeline GitLab (la variable d'environnement CI n'est pas définie)
if [ -z "$CI" ]; then
    test_cases+=(
        "-s -form NRZT -ampl -4 4"
        "-s -seed 27 -form RZ -ampl 0 6"
        "-s -mess 12 -seed 1234 -form NRZ -ampl -5 5"
    )
fi
```

Figure 59 - Modification du script runTests pour l'itération 3

Comme nous l'avons présenté dans la partie 4.3, nous effectuons également des tests visuels à l'aide des différentes sondes connectées à notre chaîne de transmission.

4.5. Bilan de l’itération

Cette troisième itération a abordé la simulation d'une transmission non idéale via un canal bruité de type « gaussien ». La gestion de notre système s'est donc révélée plus complexe par rapport aux itérations précédentes qui concernaient des transmissions idéales ou analogiques non bruitées.

Les modifications apportées pour gérer le bruit gaussien ont été testées rigoureusement, montrant que notre simulateur peut maintenir un taux d'erreur binaire contrôlé. Ces tests ont confirmé l'efficacité des ajustements réalisés, conformément aux attentes théoriques.

Les résultats obtenus dans la partie 4.3. nous ont montré qu'en fonction de la valeur du SNR (rapport signal à bruit) certains types de codages binaires obtenaient un meilleur TEB que les autres. C'était notamment le cas pour le codage RZ binaire.

Cet ajout de complexité a exigé une compréhension approfondie de l'impact du bruit sur les performances du système. Les améliorations apportées aux scripts de tests et de simulation nous ont permis d'améliorer l'efficacité de notre logiciel, notamment avec une gestion différente de la réception des messages.

En conclusion, cette itération a non seulement testé la capacité de notre système à gérer des conditions de transmission réalistes. Elle nous a également permis d'approfondir nos connaissances en matière de signaux présents au sein d'environnements bruités, et des paramètres qui les accompagnent.

V. Itération 4 : Transmission analogique avec un canal bruité à trajets multiples

5.1. Cahier des charges

Cette quatrième itération est dans la continuité de la troisième, en rendant notre canal de transmission de plus en plus réaliste. En effet, le signal émis par la source est désormais reçu avec du bruit. Cependant, le bruit n'est pas le seul effet contraignant qu'un signal subit. Par exemple, un récepteur placé en bout de chaîne de transmission peut subir l'arrivée de plusieurs trajets en plus du trajet initialement envoyé. Ce phénomène de multi-trajets se réfère à la situation où un signal émis atteint le récepteur par plusieurs chemins différents. Cela se produit lorsque les ondes émises rebondissent sur des objets comme des bâtiments, des montagnes, des arbres... chaque trajet peut varier en longueur, et donc en temps de parcours.

Dans notre cas, la chaîne de transmission vue de l'extérieur reste relativement simple, puisque c'est la même que pour l'étape précédente, comme le montre la figure ci-dessous.

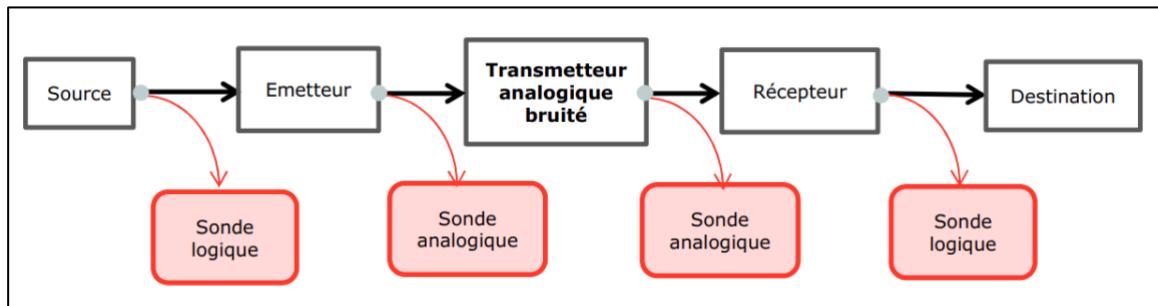


Figure 60 - Modélisation de la chaîne de transmission à l'étape 4

Au vu de ce que nous avons dit précédemment, l'effet de multi-trajets (également appelé Multipath Effect) se produit lorsque le signal se situe entre l'émetteur et le récepteur, puisque c'est à ce moment-là qu'il est dans un canal de transmission tel que l'air, un câble, le vide... C'est donc à ce moment-là que notre signal sera modifié. Pour simuler le multi-trajets, le signal d'origine subira les opérations décrites dans la figure 61.

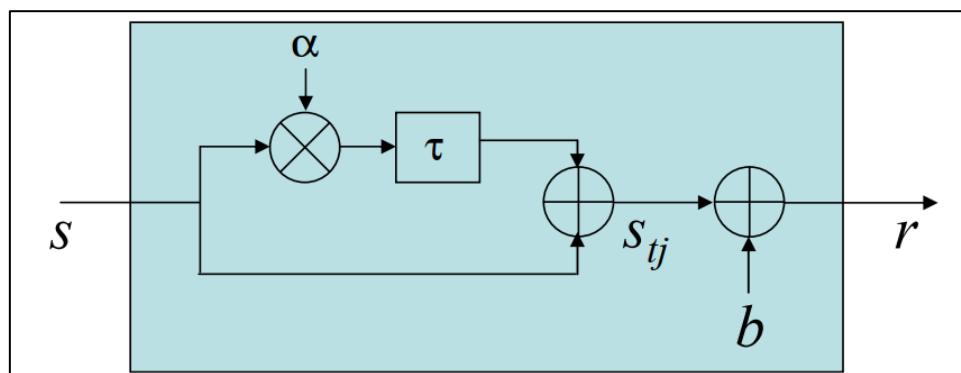


Figure 61 - Opération provoquant l'effet de multi-trajets sur un signal entrant

Pour arriver à simuler l'effet de multipath il faut additionner le signal d'origine $s(t)$ qui provient du transmetteur avec plusieurs autres signaux : les trajets qui peuvent aller de 1 à 5. Dans notre cas. Ces autres signaux, sont de la forme $trajet1(t) = \alpha s(t - \tau)$ avec :

- $s(t)$ le signal d'origine ;
- α l'atténuation du trajet par rapport au signal d'origine. Elle est comprise entre 0 et 1 ;
- τ le retard du trajet par rapport au signal d'origine.

Le signal, additionné aux trajets, ressort avec le nom S_{tj} . C'est ce signal S_{tj} auquel on ajoute le bruit blanc aléatoire gaussien que nous avons introduit lors de la précédente itération.

Pour que l'utilisateur final puisse simuler l'effet de multi-trajets notre client nous demande d'intégrer à notre logiciel le paramètre « ti » pour trajets indirects. Ce paramètre sera suivi de deux valeurs pour chaque trajet qu'il souhaite ajouter :

- « **dt** » représente le décalage temporel (en nombre d'échantillons) entre le signal d'origine et le trajet associé. Cette première valeur doit être un entier et représente τ ;
- « **ar** » représente l'amplitude relative du nouveau trajet. Nous l'avions introduite dans le paragraphe précédent comme étant α une valeur de type float comprise entre 0 et 1.

5.2. Implémentation des fonctionnalités

Chaque nouvelle fonctionnalité ajoutée à notre logiciel comprend la modification du code Java qui lui permet de fonctionner. Cette quatrième itération n'échappe pas à cette règle.

5.2.1. Création de la classe TransmetteurMultiTrajets

Comme précisé lors de l'introduction de cette étape, l'ajout de trajets au signal d'origine se fait entre l'émetteur et le récepteur. Nous avons également vu que les trajets étaient d'abord additionnés au signal envoyé par l'émetteur, avant que l'ensemble soit bruité. Pour répondre à ces contraintes nous avons décidé d'utiliser l'architecture présentée en figure 62 :

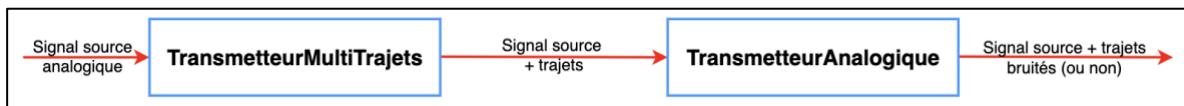


Figure 62 - Canal de transmission pour l'étape 4

Comme l'indique la figure ci-dessus, le signal qui sort de l'émetteur est envoyé vers un premier module, simulé par la classe Java **TransmetteurMultiTrajets**. Cette nouvelle classe s'occupe d'ajouter au signal envoyé par l'émetteur les différents trajets indirects entrés en paramètre par l'utilisateur. Une fois l'effet de multi-trajets appliqué au signal original, ce dernier est envoyé vers le transmetteur analogique. Si l'utilisateur a, en plus des trajets, ajouté du bruit, le signal S_{tj} sera traité par la classe **TransmetteurGaussian**, dont nous avons parlé à l'occasion de la troisième itération. Si ce n'est pas le cas, la classe **TransmetteurParfait** sera utilisée pour transmettre simplement le signal S_{tj} au récepteur. La difficulté que nous avons rencontré lors de cette étape concerne le nombre de multi-trajets que le client souhaite ajouter. En effet, pour les autres paramètres compris par le logiciel (amplitude, bruit, nombre d'échantillons...) nous avions seulement une valeur ou une paire de valeur à traiter. Pour les trajets indirects, notre logiciel doit pouvoir accepter l'ajout de cinq trajets en plus du signal initial. Par conséquent, nous avons dû parcourir, pour chaque trajet, la paire de valeurs associée avant de modifier le signal d'origine en conséquence.

5.2.2. Réflexion sur la manière de réceptionner le signal

Cette quatrième étape nous a également fait réfléchir sur la manière dont nous réceptionnons le signal grâce au récepteur. En effet, en plus de transformer le signal analogique reçu en un signal logique (suite de booléens), le récepteur nous rapporte plusieurs informations concernant le signal émis par la source. Une information très importante est le rapport signal à bruit (SNR). Pour l'obtenir, nous calculons le rapport des puissances entre la partie du signal reçu qui information, et le reste, qui constitue le bruit ajouté au signal au sein du canal de transmission. Plus ce rapport est élevé, plus la qualité du signal utile est importante par rapport au bruit qui l'entoure.

Or, l'ajout des trajets par la classe **TransmetteurMultiTrajets** a pour effet d'augmenter le signal utile, puisqu'ils ne sont pas considérés comme du bruit. Par conséquent, si nous ne prenons pas en compte l'effet de multi-trajet lors du calcul, ce dernier augmentera en même temps que le nombre de trajets. Ce n'est bien sûr pas réaliste puisque dans un cas réel l'effet de multi-trajets n'aide pas à trouver un signal utile dans un environnement bruité. Pour éviter cet effet de bord, nous avons dû faire attention à ce que tous les trajets soient pris en compte pour les calculs effectués au niveau du récepteur.

5.2.3. Intégration de l'ajout des trajets à la commande permettant d'utiliser le logiciel

Comme pour les autres itérations, la classe maîtresse de notre logiciel **Simulateur** a été modifiée pour intégrer le paramètre « -ti ». La particularité de ce paramètre est qu'il contient, pour chaque trajet ajouté, une paire de valeurs. Chaque couple de valeurs correspond à l'amplitude du trajet ainsi qu'au retard du trajet par rapport au signal original. Si l'utilisateur souhaite simuler l'effet de multi-trajets, notre logiciel vérifie que les paramètres spécifiés répondent au cahier des charges. Par défaut, aucun trajet n'est ajouté. Dans ce cas le signal reçu de l'émetteur correspond à celui reçu par le transmetteur analogique (bruité ou non).

5.3. Résultats obtenus

Notre logiciel peut désormais traiter les nouvelles fonctionnalités apportées par l'itération 4. Par conséquent, nous pouvons maintenant simuler l'effet de multi-trajets sur le signal émis par la source. Pour lancer ces simulations nous nous servons de la classe mère du projet : **Simulateur**, à laquelle nous spécifions les paramètres en entrée de notre système.

5.3.1. Résultats obtenus grâce aux sondes

Pour visualiser que l'effet de multi-trajets est bien appliqué, nous lançons trois simulations différentes avec l'amplitude par défaut, un peu de bruit (2 dB), 30 échantillons par symbole mais pas de trajets indirects pour l'instant. Les trois simulations présentées dans cette partie sont exécutées par le code binaire NRZT.

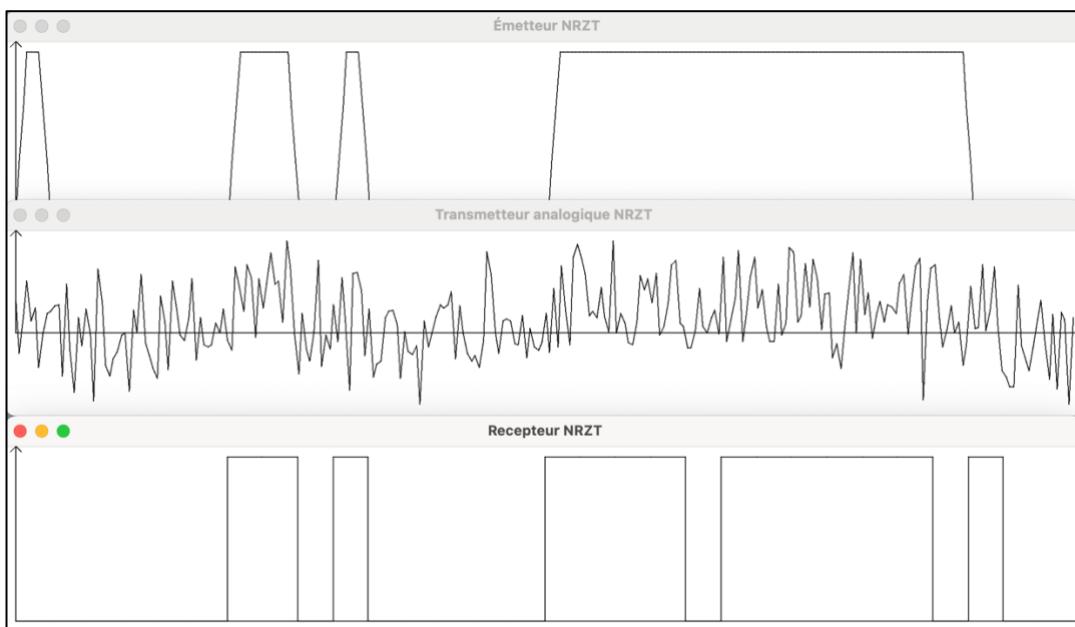
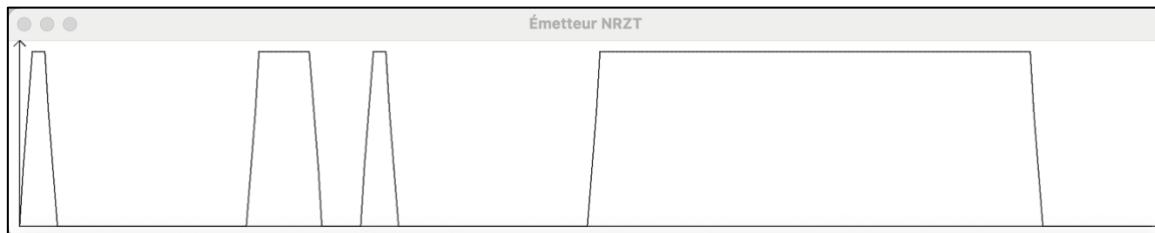


Figure 63 - Résultats retournés par les sondes pour la première simulation de l'itération 4

Les résultats obtenus avec cette simulation sont semblables à ceux de l'itération précédente, puisque pour l'instant nous nous servons uniquement des anciens paramètres intégrés au logiciel. Nous avions vu dans l'analyse de ces résultats qu'avec un SNRpb de 2 dB et en utilisant le code NRZT binaire le taux d'erreur binaire n'était pas nul. Il suffit de regarder les différentes fenêtres de la figure 63 pour remarquer que, en effet, certains bits émis par l'émetteur ont été modifiés dans le canal de transmission.

Pour la seconde simulation nous gardons les mêmes paramètres à l'exception du nombre de trajets indirects qui passe à 1.



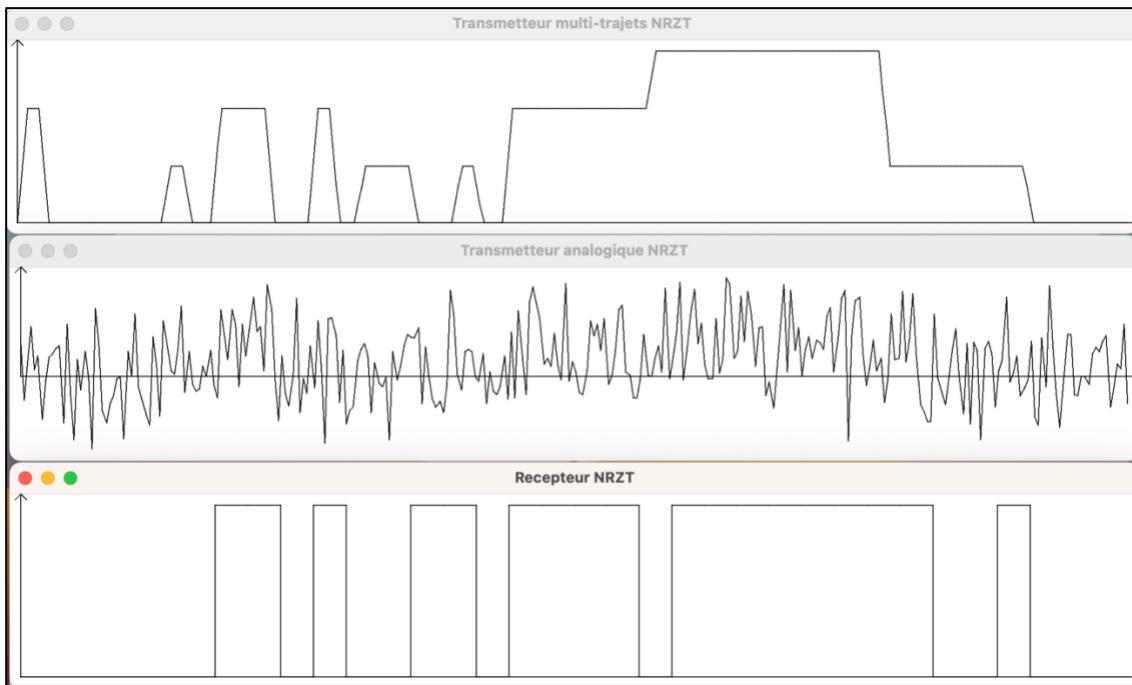


Figure 64 - Résultats retournés par les sondes pour la seconde simulation de l’itération 4

L'utilisation du paramètre « `-ti` » provoque l'appel de l'élément de notre chaîne qui ajoute l'effet de multi-trajet. Nous avons décidé de connecté une sonde analogique à la sortie de ce module pour capter le signal analogique d'origine additionner au trajet indirect. Nous pouvons remarquer que la forme du signal n'est plus la même après avoir ajouté un trajet indirect. En effet, nous avons utilisé un décalage de 40 échantillons et un rapport d'atténuation de 0.5. Cela veut dire que le signal indirect est égal au signal d'origine décalé de 40 échantillons et avec une amplitude deux fois inférieure. Nous avons expliqué précédemment que le bruit blanc aléatoire gaussien est ajouté sur le signal global (le signal d'origine ainsi que ses trajets indirects). Par conséquent, le signal bruité est également différent pour cette seconde itération, même avec l'utilisation d'une seed. Comme pour la première simulation, un test visuel nous montre que le taux d'erreur binaire n'est pas nul. Nous pouvons même remarquer que ce dernier a l'air plus important que lorsque nous n'avions que le signal direct.

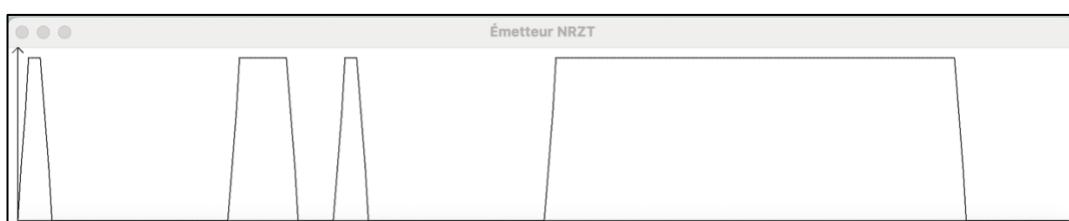
Enfin, pour cette troisième simulation nous décidons d'ajouter quatre trajets indirects au signal émis au départ de la chaîne de transmission. Pour faire cela, nous utilisons le script `simulateur` suivi des paramètres suivants :

```
Build and run                         Modify options ▾ M

java 21 SDK of 'sit213' module    simulateur.Simulateur

mess 30 -seed 12 -form NRZT -snrbp 2 -s -nbEch 9 -ti 40 0.5 20 0.7 35 0.3 70 0.65
```

Figure 65 - Paramètres permettant de lancer la troisième simulation de l’itération 4



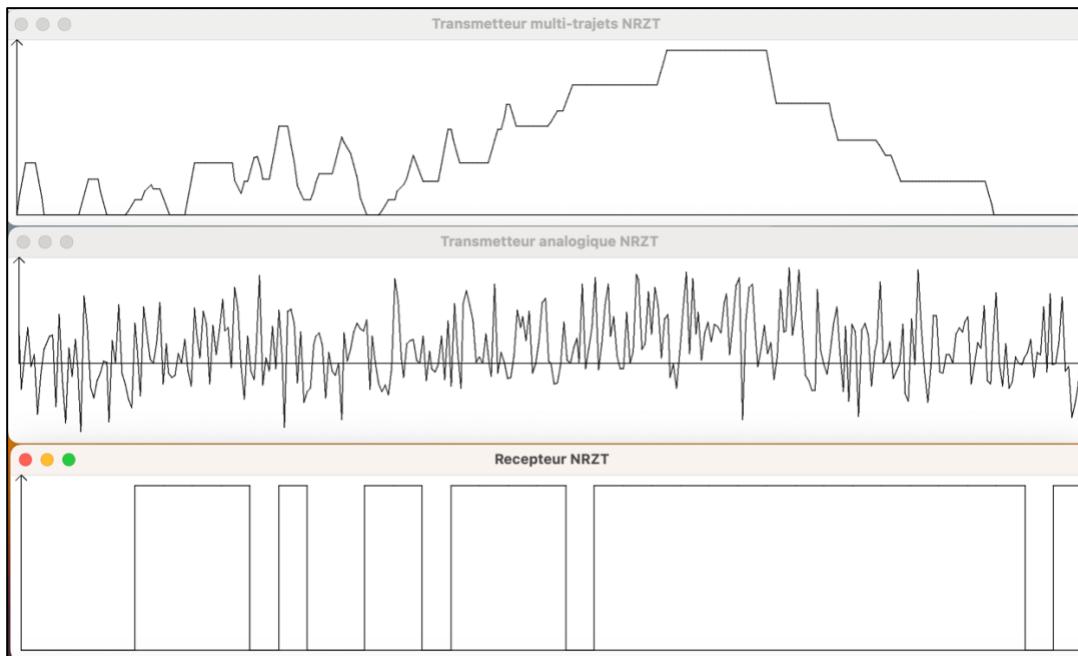


Figure 66 - Résultats retournés par les sondes pour la troisième simulation de l'itération 4

Comme lors de la précédente simulation, nous pouvons remarquer que le signal envoyé par l'émetteur change complètement de forme à partir du moment où on ajoute les trajets indirects. Par conséquent le signal bruité est encore plus déformé, et celui capté par le récepteur est très différent du signal émis par la source de notre système.

À la vue de ces trois simulations, nous pouvons penser que, plus le nombre de trajets indirects est élevé, plus le taux d'erreur binaire augmente. Nous allons confirmer, ou infirmer, cette théorie dans la prochaine partie.

5.3.2. Modification du nombre de trajets ajoutés

Pour vérifier l'hypothèse présentée ci-dessus, nous avons décidé de mettre en place un script Java au sein de la classe **ExportCSVMultiTrajets**. Cette classe permet de générer trois fichiers .csv.

Le premier d'entre eux permet de retourner la valeur du taux d'erreur binaire (TEB) et de la comparer au nombre de trajets indirects donné en entrée du simulateur. Pour avoir un résultat exhaustif, nous avons effectué six simulations. La première utilise le logiciel sans trajet indirect, puis incrémenté à chaque simulation le nombre de trajets multiples associés au signal d'origine. Sachant que l'utilisateur final du logiciel peut ajouter au maximum cinq trajets indirects, ce test nous permet de tester tous les cas possibles que le client pourra rencontrer.

Une fois le fichier .csv créé, un script Python « plot_teb_multi_trajets.py » nous permet d'afficher, pour chaque code binaire, le TEB en fonction des simulations exécutées.

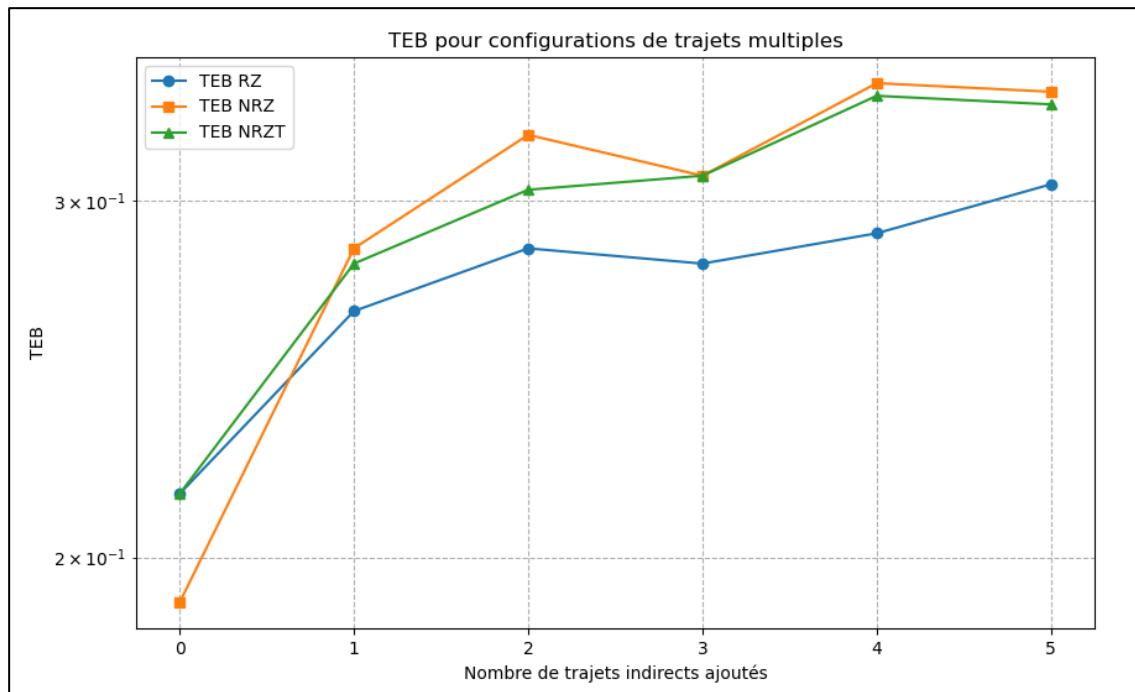


Figure 67 - Graphique affichant le TEB en fonction du nombre de trajets indirects

La figure 67 montre que, lorsque l'on augmente l'effet de multi-trajets, le TEB augmente également. Bien que nous l'avions supposé dans la partie précédente pour le code NRZT binaire, ce graphique nous permet de confirmer cette hypothèse pour les trois codes binaires que nous étudions. Par ailleurs, nous remarquons que le code binaire RZ reste le plus performant des trois types de modulation, tant que des trajets indirects sont ajoutés au signal d'origine. De plus, nous pouvons remarquer que le taux d'erreur binaire des trois codes évolue de la même manière pour les trois codes, malgré leurs différences (filtre de mise en forme, nombre d'échantillons...).

5.3.3. Modification du retard pour chaque trajet

Maintenant que nous avons la preuve que le nombre de trajets indirects influe bien sur la probabilité d'erreur binaire, nous devons trouver quel paramètre affecte le plus le TEB. Pour rappel, un signal indirect est composé de deux paramètres :

- α l'atténuation du trajet par rapport au signal d'origine ;
- τ le retard du trajet par rapport au signal d'origine.

Notre client demande que la valeur τ donnée en paramètre représente le nombre d'échantillons d'écart entre le trajet indirect et le signal d'origine. Par conséquent, un retard très faible, c'est-à-dire qui ne dépasse pas la taille d'un symbole, aura peu d'impact sur la bonne réception d'un message. Cependant, quand le retard associé à un trajet indirect vaut plus d'un symbole les effets sont bien plus néfastes puisque le trajet indirect se retrouve complètement décalé par rapport au signal d'origine.

La deuxième expérimentation mise en place par la classe **ExportCSVMultiTrajets** s'occupe, à chaque itération, de modifier le retard de chaque trajet indirect par rapport au signal d'origine. En effet, lors des 10 simulations exécutées, les retards τ sont incrémentés de 20 échantillons. La figure 68 affiche le TEB pour chaque simulation effectuée, sachant que la simulation n+1 a un retard plus important que la simulation n.

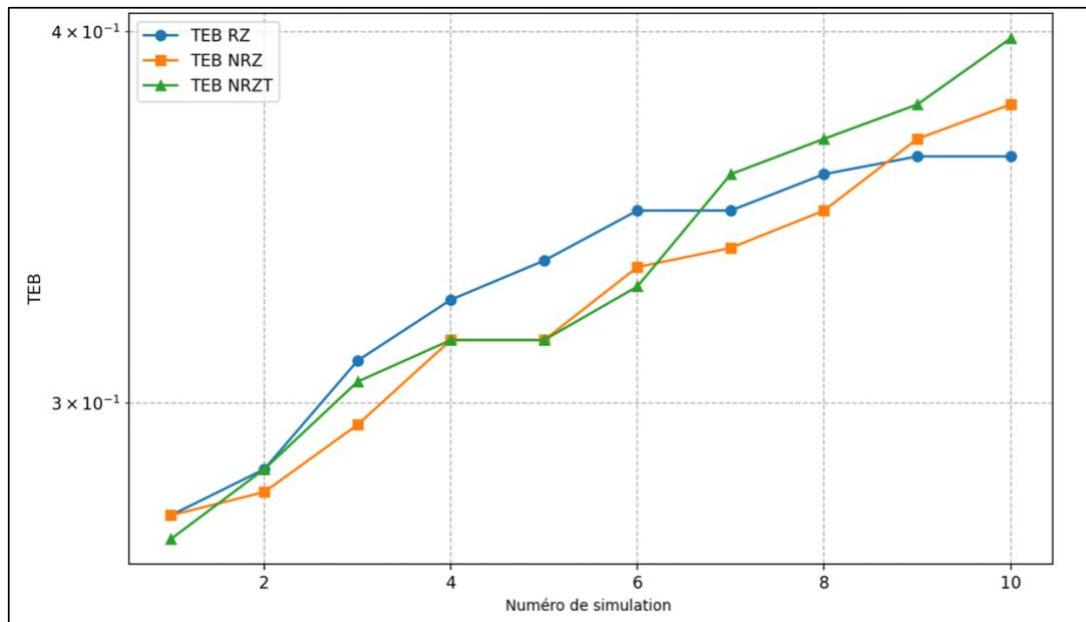


Figure 68 - Graphique affichant le TEB, avec l'augmentation des retards entre chaque simulation

Comme pour la première expérimentation, le résultat semble très clair grâce au graphique ci-dessus. En effet, nous pouvons remarquer que le décalage temporel des signaux indirects influe grandement sur le taux d'erreur binaire pour chacun des trois codes binaires. Comme pour le premier graphique, les trois types de modulations évolue de la même manière. De plus, à chaque itération le retard τ subit la même opération. Cette opération peut être représentée par une suite arithmétique de la forme $\tau_{n+1} = \tau_n + 20$. Cette suite arithmétique se retrouve sur la forme du graphique, puisque le TEB grandit de façon relativement linéaire, même si le TEB reste une probabilité. Par conséquent, il est impossible que les courbes visualisées soient parfaitement linéaires. Nous remarquons également que la courbe n'est pas monotone, et que parfois le TEB reste le même pendant deux simulations d'affilées. C'est le cas quand le retard τ du nouveau signal arrive par exemple au même niveau qu'un symbole, comme nous l'avons précisé précédemment. Enfin, nous pouvons noter que le TEB des trois codes binaires a évolué d'environ 0.13. Nous reviendrons sur cette valeur lorsque nous comparerons cette expérimentation à la prochaine.

5.3.4. Modification de l'amplitude pour chaque trajet

La troisième et dernière expérimentation que nous effectuons concerne le facteur d'atténuation α qui caractérise chaque trajet indirect. Nous appelons à nouveau la classe **ExportCSVMultiTrajets** pour récupérer le TEB pour chaque itération. Cette fois, chaque itération suit la suite géométrique suivante : $\alpha_{n+1} = \alpha_n \times 1.05$. La figure 69 affiche le TEB pour chaque simulation effectuée, sachant que les trajets indirects de la simulation $n+1$ ont une amplitude plus importante que ceux de la simulation n .

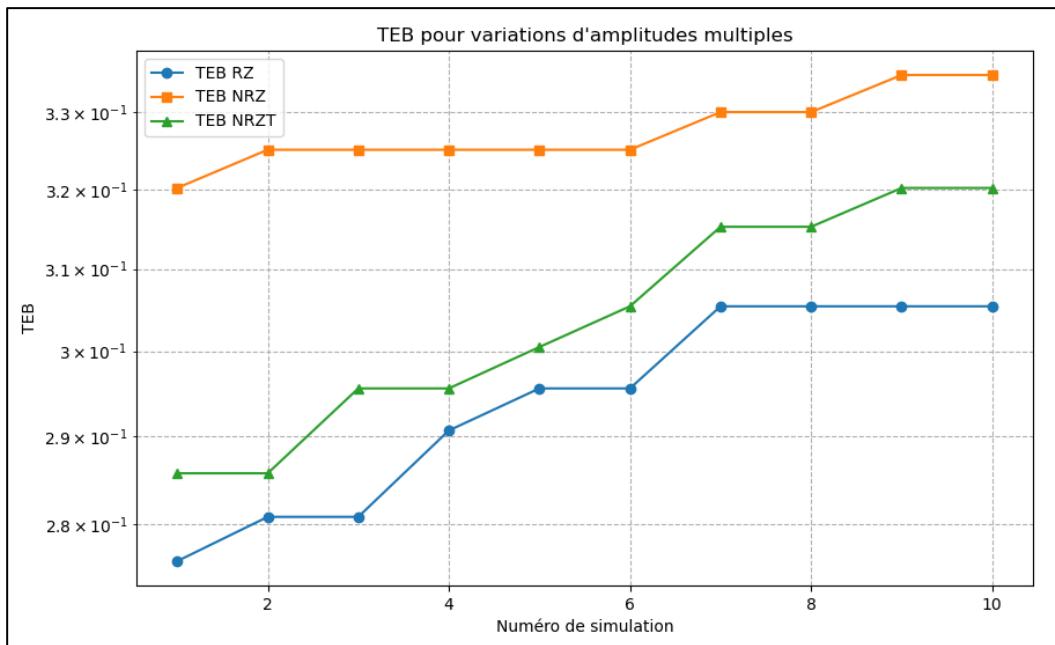


Figure 69 - Graphique affichant le TEB, avec l'augmentation de l'amplitude des TI entre chaque simulation

Cette fois encore, la modulation RZ binaire, représentée en bleu sur la figure 69, semble être le code binaire ayant les meilleures performances. Nous remarquons également que l'atténuation des trajets indirects α influe, comme les autres paramètres, sur le taux d'erreur binaire. En effet, plus un trajet indirect à une amplitude élevée par rapport au signal d'origine, plus il penchera dans la balance lors des calculs de moyennes qui servent à déterminer l'état final d'un bit. Cependant, nous pouvons remarquer que c'est beaucoup moins le cas pour le codage binaire NRZ. Cette différence peut être due à la manière dont le récepteur analyse les données reçues. De plus le codage NRZ, contrairement aux autres types de modulations traitées, conserve un niveau constant pendant toute la durée du symbole. Il est donc naturellement moins sensible aux changements d'amplitude.

Cependant, le TEB augmente d'environ 0.03. Si nous comparons ce résultat à celui obtenu lors de la seconde expérimentation, nous pouvons conclure que le retard des trajets indirects τ semble avoir une plus grande responsabilité dans l'augmentation du taux d'erreur binaire par rapport à l'atténuation α .

5.4. Tests effectués

Cette quatrième itération inclut une nouvelle classe qui nous permet de simuler l'effet de multi-trajets. Comme pour les autres étapes de ce projet, nous effectuons des tests grâce à divers outils (JUnit 4, Emma et EasyMock) pour vérifier que cette nouvelle classe fonctionne comme prévu.

5.4.1. Tests de la classe TransmetteurMultiTrajets

Les trois tests effectués sur cette classe permettent de vérifier que la simulation de l'effet de multi-trajets impacte bien le signal reçu par le récepteur. Nous vérifions également que le taux d'erreur binaire (TEB) ainsi que le rapport signal à bruit (SNR) sont correctement impactés par les trajets ajoutés au sein de la chaîne de transmission.

- ✓ **TransmetteurMultiTrajetsTest (tests)**
 - ✓ nbMultiTrajet
 - ✓ тебAugmente
 - ✓ snrAugmente

Figure 70 - Résultats des tests pour la classe TransmetteurGaussian

La figure 63 liste les tests effectués sur la classe **TransmetteurMultiTrajets**. Nous pouvons voir que les trois tests suivants se sont déroulés comme prévu :

- nbMultiTrajet vérifie que le nombre de trajets ajoutés par l'utilisateur correspond bien aux actions effectuées par le canal de transmission ;
- тебAugmente s'assure que le taux d'erreur binaire est plus élevé quand le nombre de trajets augmente, comme nous l'avons vu dans la partie 5.3 ;
- snrAugmente retourne la valeur du rapport signal à bruit après avoir augmenté le nombre de trajets. Le test vérifie que le SNR est bien retourné à chaque fois et qu'il n'augmente pas considérablement, comme nous l'avons expliqué lors de l'implémentation de cette itération.

Après avoir exécuté ces trois tests nous observons la couverture de notre nouvelle classe grâce à l'outil Emma.

Element ^	Class, %	Method, %	Line, %	Branch, %
▼ transmetteurs	75% (3/4)	65% (21/32)	78% (77/98)	71% (27/38)
© Transmetteur	100% (1/1)	20% (2/10)	42% (6/14)	100% (0/0)
© TransmetteurGaussian	100% (1/1)	93% (14/15)	89% (42/47)	75% (12/16)
© TransmetteurMultiTrajets	100% (1/1)	100% (5/5)	96% (29/30)	83% (15/18)
© TransmetteurParfait	0% (0/1)	0% (0/2)	0% (0/7)	0% (0/4)

Figure 71 - Couverture du package transmetteur après avoir ajouté la classe TransmetteurMultiTrajets

La figure ci-dessus nous montre que toutes les méthodes qui composent la classe **TransmetteurMultiTrajets** sont appelées par les tests décrits ci-dessus, ce qui démontre la bonne couverture de notre code.

5.4.2. Modification du script runTests

Comme depuis le début de ce projet, c'est le script bash *runTests* qui exécute tous les tests pour notre logiciel. Pour cette nouvelle itération, nous avons ajouté des tests unitaires qui se servent du paramètre « *-ti* ». Ce paramètre est combiné aux fonctionnalités programmées depuis le début de ce projet (amplitude, nombre d'échantillons, bruit...). La figure ci-dessous est un extrait des tests que nous exécutons, dont certains incluent l'ajout de trajets indirects.

```

"--mess 0010011 -form RZ -snrpb 0"      # SNR par bit avec format RZ
"--mess 0010011 -form RZ -snrpb 30"      # SNR par bit avec format RZ
"--mess 0010011 -form NRZT -snrpb -30"    # SNR par bit avec format NRZT
"--mess 0010011 -form NRZT -snrpb 0"      # SNR par bit avec format NRZT
"--mess 0010011 -form NRZT -snrpb 30"      # SNR par bit avec format NRZT
"--ampl -3 3 -form NRZT -snrpb 30"        # SNR par bit avec amplitudes personnalisées

"--mess 0010011 -ti 60 0.3"                # 1 trajet indirect
"--mess 0010011 -ti 0 1.0"                  # Trajet indirect sans décalage
"--mess 0010011 -ti 100 0.75 20 0.4"       # 2 trajets indirects
"--mess 0010011 -ti 100 0.9 20 0.6 40 0.5"  # 3 trajets indirects
"--mess 0010011 -ti 100 1.0 20 0.75 5 0.5 80 0.1 120 0.8"  # 5 trajets indirects
)

# Si on n'est pas dans un pipeline GitLab (la variable d'environnement CI n'est pas définie)
if [ -z "$CI" ]; then
    test_cases+=(
        "-s -form NRZT -ampl -4 4"
        "-s -seed 27 -form RZ -ampl 0 6"
        "-s -mess 12 -seed 1234 -form NRZ -ampl -5 5"
        "-s -mess 01101101110 -form NRZT"
        "-s -mess 01101101110 -form NRZ -snrpb 30"
        "-s -mess 01101101110 -form NRZT -snrpb 30"
        "-s -mess 01101101110 -form RZ -snrpb 30"
        "-s -mess 01101101110 -form RZ -ti 100 1.0 20 0.75 5 0.5 80 0.1 120 0.8"
    )
fi

```

Figure 72 - Modification du script *runTests* pour l'itération 4

Comme pour les précédentes itérations, nous effectuons des tests visuels à l'aide des sondes intégrées à la chaîne de transmission.

5.5. Bilan de l’itération

Cette dernière itération visait à intégrer et tester les fonctionnalités complètes de notre système de transmission, incluant la gestion du bruit gaussien et de l’effet de multi-trajets. Au cours de cette quatrième étape, plusieurs composants ont été développés et intégrés à notre logiciel, afin d’améliorer la fiabilité de notre système de transmission, tout en rendant notre canal plus réaliste qu’auparavant.

Les principales fonctionnalités ajoutées incluaient la gestion avancée du bruit et l’optimisation des paramètres de transmission pour minimiser le taux d’erreur binaire (TEB). La simulation des trajets indirects a permis de tester l’efficacité de notre système face aux perturbations réalistes. Nous avons également apporté des ajustements pour améliorer la performance globale.

Les résultats de cette quatrième itération ont été plutôt positifs, puisque notre système continue à gérer efficacement le bruit et les distorsions du canal de transmission. Bien que le TEB augmente avec l’intégration des trajets indirects, les méthodes dont nous nous servons pour démoduler notre signal nous permettent de garder une probabilité d’erreur satisfaisante, notamment avec le code RZ binaire.

Une série complète de tests a été effectuée, couvrant à la fois les fonctionnalités récemment implémentées et les scénarios de transmission plus complexes. Les tests ont validé la fonctionnalité de chaque composant et ont également aidé à identifier et à rectifier les dernières imperfections de notre système.

Pour résumer, cette itération nous a permis de continuer à développer notre système de transmission, tout en se rapprochant de plus en plus d’un canal réaliste. Ce dernier sait désormais gérer l’effet de multi-trajets combinés à un bruit blanc gaussien. Notre système est désormais prêt à passer au codage canal, afin d’améliorer le TEB obtenu en sortie de la structure.

VI. Itération 5 : Codage de canal

6.1. Cahier des charges

Cette cinquième itération est la dernière de ce projet a ajouté une fonctionnalité majeure à notre logiciel : le codage canal. Nous avons vu dans les précédentes itérations que l'ajout de perturbations au sein du canal de transmission (bruit blanc gaussien, multi-trajets) faisait augmenter le taux d'erreur binaire. Cette cinquième itération a pour but de réduire ce TEB.

Pour faire cela, nous allons introduire deux nouveaux modules à notre chaîne de transmission : un **codeur** et un **décodeur**. La figure ci-dessous indique où ils seront placés dans notre système.

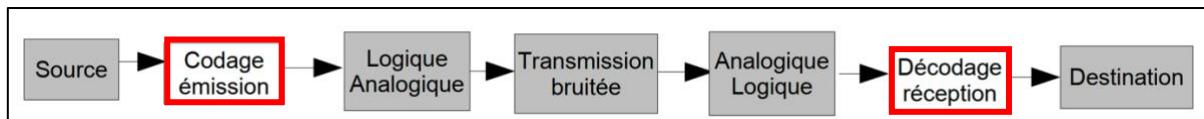


Figure 73 - Modélisation de la chaîne de transmission à l'étape 5

Comme le montre la figure 73, la source émet un signal binaire vers le **codeur** qui renvoie également un signal logique vers l'émetteur. De l'autre côté de la chaîne, c'est le schéma inverse qui se produit : le récepteur émet un signal numérique qui arrive vers le **décodeur** avant d'être transmis à la destination. Mais si on veut que le TEB diminue, ces deux modules ne se contenteront pas de seulement faire passer une suite de bits, ils devront également traiter le signal. Il existe plusieurs techniques de codage canal plus ou moins complexes. Le cahier des charges nous a demandé d'utiliser la suivante :

- Chaque bit qui arrive au niveau du **codeur** sera transformé en une séquence de trois bits bien définis. Un '0' sera transformé en '010' et un '1' sera transformé en '101'. Ainsi, on retrouve dans chaque séquence de trois bits deux fois le bit initialement transmis par la source ;
- En sortie de la chaîne de transmission, le **décodeur** recevra logiquement trois fois plus de bit que le codeur. Son rôle sera donc de retransformer chaque séquence de trois bits en un bit unique. Si la séquence de bits n'a pas subi d'erreur, il suffit de transformer les '010' en un '0' et les '101' en un '1'. C'est dans le cas où les trois bits analysés ne seraient pas conformes (ex : '111', '001'...). Dans ce cas, le décodeur observe à quelle séquence originale ('010' ou '101') les bits observés ressemblent le plus (exemple : '111' ≈ '101' ; '110' ≈ '010').

Grâce à cette méthode de codage canal, si une erreur s'est glissée dans une séquence de trois bits, elle sera détectée et corrigée. Malgré cela, cette méthode n'est pas infaillible puisqu'une séquence trois bits qui contient deux erreurs ne pourra pas être corrigée, comme le montre l'exemple ci-dessous.



Figure 74 - Exemple d'une erreur de transmission malgré le codage canal

Même si des erreurs peuvent apparaître, la mise en place d'un codage canal dans notre simulateur devrait permettre de réduire le taux d'erreur binaire. Par défaut, aucune correction d'erreur n'est apportée au signal. Si l'utilisateur veut utiliser le codage canal, il doit utiliser le paramètre « `-codeur` » pour s'en servir.

6.2. Implémentation des fonctionnalités

L'implémentation des fonctionnalités pour cette cinquième étape s'est avérée être assez simples. En effet, cette itération ne comprend que l'ajout de deux modules, et d'un nouveau paramètre « -codeur ». Nous avons décidé de créer une classe dédiée à chacun des nouveaux blocs de la chaîne, pour suivre la logique appliquée pour le reste de ce projet. Pour l'intégration du nouveau paramètre, nous avons modifié le code source de la classe **Simulateur**, qui regroupe tous les paramètres compris par notre simulateur.

6.2.1. Crédit de la classe Codeur

Cette nouvelle classe nommée **Codeur** vient se placer entre la source et l'émetteur. Comme nous l'avons expliqué lors de la présentation de cette étape, ce bloc reçoit et émet une suite binaire, il n'y a donc pas de conversion logique/analogique à traiter ici. La figure ci-dessous montre les différentes étapes par lesquelles passe le signal au moment du codage canal.

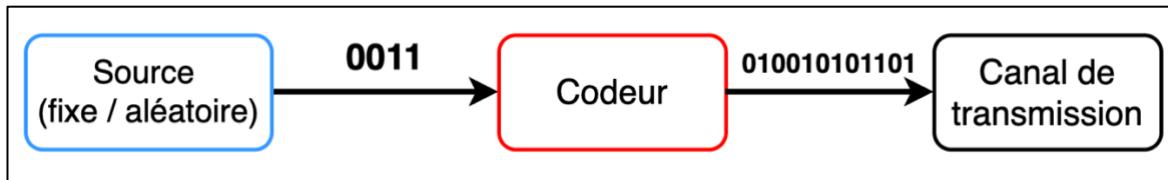


Figure 75 - Zoom sur le codeur

Le fonctionnement de notre codeur est très simple. Il parcourt tous les bits qu'il reçoit depuis la source. Chaque bit analysé est transformé en une séquence de trois bits, comme présenté dans la partie précédente. Enfin, il émet cette nouvelle suite binaire en direction du canal de transmission. Pour éviter les bugs, notre code source émet un message d'erreur si le message reçu par le codeur est vide.

6.2.2. Crédit de la classe décodeur

La classe **Décodeur** présente dans le package « codage » de notre code, subit le traitement inverse du codeur, en étant placé entre le récepteur et la destination.



Figure 76 - Zoom sur le décodeur

Malgré les apparences, cette seconde classe, même si elle va de pair avec **Codeur**, fût plus complexe à développer. En effet, contrairement à la phase de codage, des erreurs ont pu apparaître au sein du canal de transmission. Il faut donc, en plus de repasser d'une séquence de trois bits à un unique bit, s'occuper de détecter et de corriger ces erreurs. Pour faire cela, nous avons divisé notre code en plusieurs parties, en fonction de l'état de la séquence binaire à analyser. Comme précédemment, un signal logique arrive du canal de transmission, mais cette fois, il sera analysé trois bits par trois bits.

- Premier cas : les trois bits valent '010'. Pour notre méthode de codage canal, cela correspond à un '0' sans erreur ;

- Second cas : les trois bits valent ‘101’. Cela correspond à un ‘1’ si la séquence de trois bits n’a pas subi de changement ;
- Troisième cas : les trois bits ont une valeur différente des deux premiers cas. Cela signifie qu’au moins une erreur est apparue dans le canal de transmission. Dans ce cas, notre décodeur compare l’écart entre la chaîne analysée et les deux séquences de références, avant de la transformer en ‘0’ ou en ‘1’.

Enfin, la nouvelle séquence binaire obtenue, qui doit faire la même taille que celle émise par la source, est envoyée à la destination, qui représente la fin de notre système.

6.2.3. Intégration de l’utilisation d’un codage de canal à la commande permettant d’utiliser le logiciel

Pour rappel, c’est la classe **Simulateur** qui permet à notre logiciel de traiter les différents paramètres donnés par l’utilisateur lorsqu’il souhaite utiliser notre logiciel. Nous avons donc modifié cette classe pour qu’elle prenne en compte les deux nouvelles classes **Codeur** et **Decodeur**, ainsi que le paramètre « -codeur ».

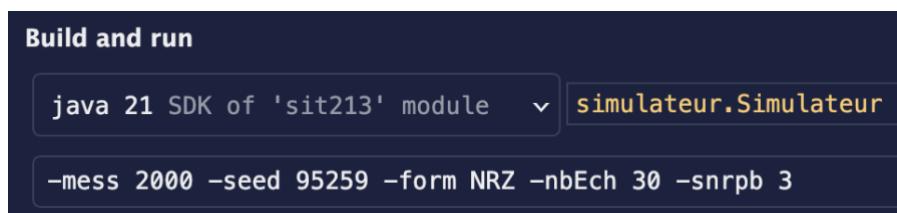
Si l’utilisateur ne spécifie pas qu’il souhaite utiliser un codage canal, alors la source sera directement reliée à l’émetteur, et au récepteur à la destination. A contrario, s’il décide d’utiliser le paramètre « -codeur » lors de l’appel du logiciel, les deux nouveaux blocs viendront se placer au sein de la chaîne de transmission (cf figure 73).

6.3. Résultats obtenus

Les nouvelles fonctionnalités apportées par l’itération 5 sont désormais gérées par notre simulateur. Nous pouvons lancer des simulations pour observer l’efficacité du codage canal sur nos simulations.

6.3.1. Résultats obtenus sur des tests unitaires grâce à l’IDE

Pour vérifier que le codage canal est efficace, nous commençons par effectuer des simulations uniques, avec différents paramètres, et d’observer le TEB obtenu à chaque fois.



```
Build and run

java 21 SDK of 'sit213' module      simulateur.Simulateur
-mess 2000 -seed 95259 -form NRZ -nbEch 30 -snrpb 3
```

Figure 77 – Paramètres utilisés pour la première simulation

Pour cette première simulation, nous testons simplement d’émettre un signal dans un environnement bruité, avec un SNR par bit de 3dB. Nous utilisons également le paramètre « -seed » pour pouvoir comparer l’utilisation ou non du codage canal dans les mêmes conditions, ce qui est rendu possible grâce à l’utilisation d’une seed. Lors de la simulation sans l’utilisation de la paire codeur/décodeur, nous obtenons un taux d’erreur binaire de 0.078. Nous exécutons ensuite la même commande, mais cette fois, nous ajoutons le paramètre « -codeur ». Cette fois-ci, nous obtenons un TEB de 0.024. Avec l’utilisation du codage canal, le taux d’erreur binaire a été divisé par 3.25, ce qui nous montre déjà que cette nouvelle fonctionnalité peut s’avérer très efficace dans des cas où on l’on veut un taux d’erreur binaire très faible.

Nous avons ensuite effectué la même simulation, mais cette fois en changeant la forme d’onde utilisée. Nous passons d’une modulation en NRZ binaire à une modulation en RZ binaire. Cette fois-ci, nous obtenons un TEB de 0.087 lorsque nous n’utilisons pas de codage canal, et 0.026 lorsque nous en utilisons un. Nous obtenons donc un rapport de 3.55. Avec ces paramètres, l’utilisation d’un codeur avec la modulation RZ binaire semble donc être plus efficace.

Notre logiciel permet également de simuler l’émission d’un signal NRZT binaire. Nous effectuons donc un troisième test, cette fois avec la modulation binaire NRZT. Sans l’utilisation du codage canal, nous obtenons un taux d’erreur binaire de 0.11 et 0.04 lorsque nous utilisons le paramètre « -codeur ». Avec la forme d’onde NRZT le rapport $\frac{TEB \text{ sans codeur}}{TEB \text{ avec codeur}} = 2.66$.

Nous pouvons remarquer, après la série de tests que nous venons d’effectuer, que l’efficacité du codage canal fasse à un environnement bruité semble suivre la même tendance que le TEB en fonction du SNR par bit pour les trois formes d’onde. En effet, les valeurs obtenues pour NRZ et RZ semblent être très proches. D’un autre côté, l’utilisation du codage canal pour la forme d’onde NRZT semble être moins efficace. Ces résultats peuvent sembler logiques puisque, comme nous l’avions vu sur la figure 53, la modulation NRZT binaire s’était révélé plus sensible aux bruits comparé aux deux autres formes d’ondes. Nous retrouvons donc ce résultat ici.

6.3.2. Efficacité du codeur dans un environnement bruité

Depuis le début de ce projet, nous nous sommes intéressés particulièrement à la forme d'onde NRZ binaire. D'abord, puisque nous avons pu observer qu'avec notre simulateur, c'est avec cette forme d'onde que nous obtenions le meilleur TEB en fonction du SNR par bit. De plus, c'est avec ce code binaire que nous avons pu tester la fiabilité de notre simulateur, en comparant les résultats théoriques avec nos simulations. C'est pour suivre cette logique que nous avons décidé d'effectuer toute une série de simulations pour mesurer l'efficacité du codage canal sur le TEB, en fonction du SNR par bit.

Pour faire cela, nous avons ajouté une nouvelle classe **TEBComparaisonCodeur** à notre code Java. Elle s'occupe d'enregistrer le taux d'erreur binaire en fonction du rapport signal à bruit qui évolue entre -10dB et 12dB dans un fichier CSV. Une fois que nous avons nos valeurs avec et sans utiliser le codage canal, un script Python se charge de les afficher sur un graphe afin d'obtenir la figure suivante :

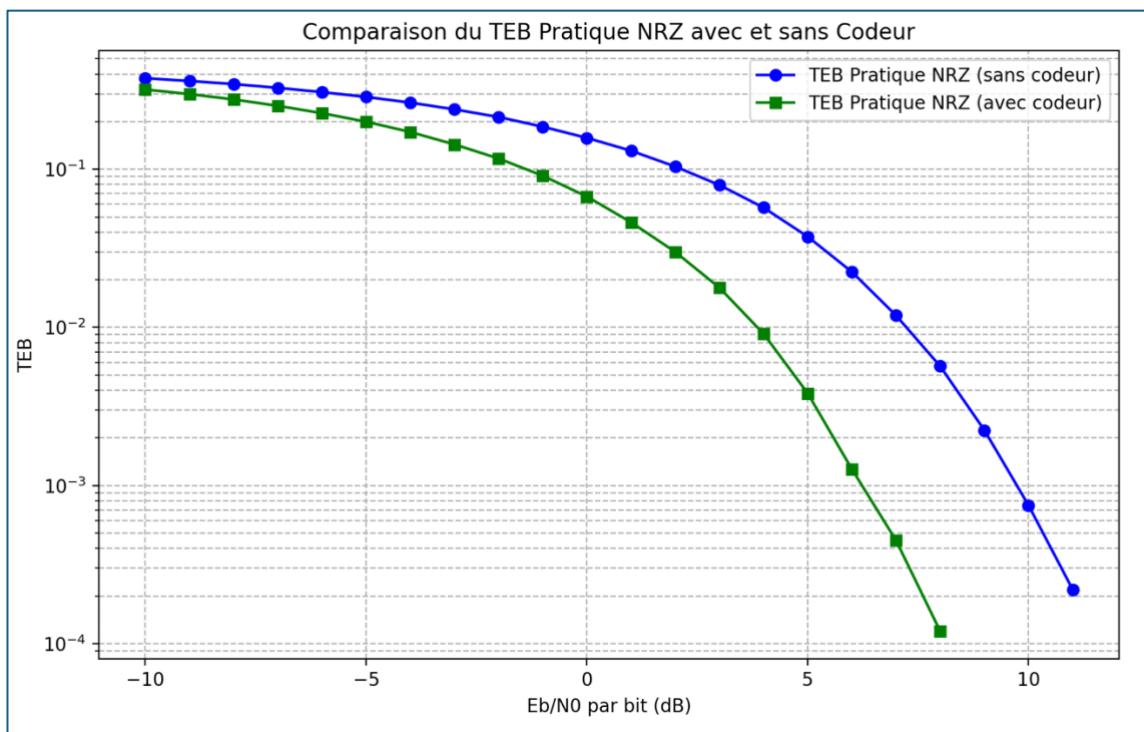


Figure 78 - TEB en fonction du SNR par bit avec et sans codage canal pour le code NRZ binaire

La première chose que nous pouvons observer sur la figure 78 est que le taux d'erreur binaire avec l'utilisation d'un codeur est toujours inférieur par rapport à la courbe bleue (avec codeur). Cela montre que le codage canal introduit dans notre logiciel est efficace.

Si nous regardons plus en détail, nous pouvons observer que c'est pour un rapport signal sur bruit élevé que l'écart entre les deux simulations est le plus élevé. Quand le signal subit peu de bruit il y a forcément moins d'erreurs introduites au sein du canal de transmission. Dans ces cas-là notre simulateur, avec l'aide du codage canal, arrive à supprimer quasiment l'intégralité des erreurs pour obtenir un TEB de 10^{-3} avec le codeur, contre 10^{-2} sans le codage canal.

Nous observons cependant que plus le signal est transmis dans un environnement bruité (avec un SNR par bit en dB faible), plus la différence de TEB entre les deux simulations est faible. Par exemple avec un $\left(\frac{E_b}{N_0}\right) dB$ qui vaut -5dB, le taux d'erreur binaire est de 2×10^{-1} avec le codage canal, et 3×10^{-1} sans l'utilisation du codeur. Cela s'explique par le fait qu'avec un bruit élevé le nombre d'erreurs est démultiplié. Par conséquent, il arrive très souvent que dans une

séquence de 3 bits, il y ait 2 erreurs. Or, nous avons vu précédemment que le type de codage canal retenu dans le cahier des charges n'est pas prévu pour corriger deux erreurs dans une suite de 3 bits. Par conséquent, on se retrouve avec un TEB relativement élevé pour un SNR par bit très faible, même s'il reste inférieur quand nous utilisons le codage canal.

Pour conclure sur les résultats que nous avons obtenus grâce aux différentes simulations effectuées, nous pouvons dire que le codage canal intégré à notre simulateur est efficace pour les trois formes d'onde RZ, NRZ et NRZT. L'expérience plus précise avec le code NRZ binaire nous a permis de voir que la synchronisation entre l'émetteur et le récepteur est plus performante pour un rapport signal à bruit élevé. Pour un SNR par bit plus faible le codage canal reste efficace, même si la différence de TEB obtenu avec et sans codeur est moins importante.

6.4. Tests effectués

Comme pour les fonctionnalités intégrées lors des précédentes itérations, nous avons créé une classe de tests dédiée au codage canal. Cette classe, nommée **CodeurDecodeurTest**, se sert du JUnit 4 pour simuler les tests qui seront présentés par la suite. Nous verrons également grâce à l'outil Emma la couverture d'instructions des classes **Codeur** et **Decodeur**, c'est-à-dire le pourcentage de code appelé lors de l'exécution de cette nouvelle classe de test. Certains de ces tests renvoient des valeurs nous permettant de vérifier que le test est concluant (True, False, Error...). D'autres sont des tests visuels. Ce sera donc à nous de vérifier qu'ils se sont déroulés comme nous le souhaitons.

6.4.1. Tests des classes Codeur et Decodeur

Nous avons effectué au total six tests pour ces deux classes qui intègrent le codage canal à notre simulateur. Ces tests permettent de vérifier que les signaux binaires sont bien codés et décodés, mais également l'efficacité du codage canal sur le TEB, ou encore que notre simulateur fonctionne bien avec des conditions extrêmes.

✓ ✓ CodeurDecodeurTest (tests)	36 ms
✓ testSNR	17 ms
✓ testDetectionCorrection	0 ms
✓ testChaine	1 ms
✓ testCodeur	0 ms
✓ testPerformance	18 ms
✓ testTransformation	0 ms

Figure 79 - Résultats des tests pour les classes Codeur et Decodeur

La figure 79 liste les tests effectués grâce à la classe **CodeurDecodeurTest**. Les ✓ verts nous montrent qu'ils se sont tous exécutés sans problème :

- testSNR vérifie le comportement du simulateur avec des entrées invalides ou des conditions extrêmes (SNR par bit très faible ou très fort) ;
- testDetectionCorrection s'assure que le décodeur détecte et corrige une erreur dans une séquence de trois bits ;
- testChaine vérifie que l'ensemble de la chaîne de transmission est fonctionnelle ;
- testCodeur s'assure que les classes **Codeur** et **Decodeur** transforme bien un bit en une séquence de trois bits, et inversement ;
- testPerformance retourne plusieurs variables (SNR par bit, TEB) avec et sans l'utilisation d'un codage canal ;
- testTransformation vérifie que la classe **Codeur** transforme bien un '0' en '010' et un '1' en '101'.

L'ensemble des tests réalisés à l'aide de JUnit 4 sont passés avec succès, nous vérifions la couverture de notre code avec l'outil Emma.

Element ^	Class, %	Method, %	Line, %	Branch, %
▼ codage	100% (3/3)	80% (8/10)	92% (46/50)	94% (32/34)
⌚ AbstractCodeur	100% (1/1)	60% (3/5)	77% (7/9)	100% (0/0)
⌚ Codeur	100% (1/1)	100% (2/2)	93% (14/15)	87% (7/8)
⌚ Decodeur	100% (1/1)	100% (3/3)	96% (25/26)	96% (25/26)

Figure 80 - Couverture du package codage après avoir ajouté les classes **Codeur** et **Decodeur**

Nous pouvons remarquer grâce à la figure 80 que l'ensemble des méthodes des classes **Codeur** et **Decodeur** sont appelés pendant l'exécution de la classe de tests. Cela ne veut pas dire que notre code ne contient pas de bug, mais nous aide à penser qu'il effectue les actions qu'il est censé faire.

6.4.2. Modification du script runTests

Cette cinquième itération apporte un nouveau paramètre « -codeur ». Par conséquent, nous modifions le script *runTests* pour qu'il contienne une nouvelle batterie de tests qui inclue ce nouveau paramètre.

```

"-mess 0010011 -form NRZT -codeur -nbEch 20 -ampl -4 4 -ti 3 0.4"
"-mess 01101101110 -form NRZ -codeur -nbEch 25 -ampl -2 2 -snrpb 9 -ti 4 0.3 6 0.2"
"-mess 100111001 -form RZ -codeur -nbEch 30 -snrpb -10 -ti 5 0.5"
"-mess 110011 -form NRZT -codeur -nbEch 15 -snrpb -6 -ti 2 0.6"
"-mess 1010 -form RZ -codeur -nbEch 35 -ampl 0 1 -snrpb 3 -ti 4 0.3 8 0.1"
"-mess 0010011 -form NRZ -codeur -nbEch 40 -ampl -10 10 -ti 5 0.2 70 0.4"
"-mess 110011 -form RZ -codeur -nbEch 22 -snrpb 0"
"-mess 01101101110 -form NRZT -codeur -nbEch 50 -snrpb 6 -ti 20 0.4 70 0.3 90 0.2"
"-mess 30 -form NRZT -codeur -nbEch 28 -ampl -5 5 -snrpb -2 -ti 15 0.5 6 0.2 30 0.8"
"-mess 1010 -form NRZT -codeur -nbEch 18 -ampl -3 7 -snrpb 4 -ti 2 0.3"
)

# Si on n'est pas dans un pipeline GitLab (la variable d'environnement CI n'est pas définie)
if [ -z "$CI" ]; then
    test_cases+=(
        "-s -form NRZT -ampl -4 4"
        "-s -seed 27 -form RZ -ampl 0 6"
        "-s -mess 12 -seed 1234 -form NRZ -ampl -5 5"
        "-s -mess 01101101110 -form NRZT"
        "-s -mess 01101101110 -form NRZ -snrpb 30"
        "-s -mess 01101101110 -form NRZT -snrpb 30"
        "-s -mess 01101101110 -form RZ -snrpb 30"
        "-s -mess 01101101110 -form RZ -ti 100 1.0 20 0.75 5 0.5 80 0.1 120 0.8"
        "-s -mess 250 -form NRZT -codeur -nbEch 15 -snrpb -6 -ti 2 0.6"
        "-s -mess 35 -form RZ -codeur -nbEch 35 -snrpb 3 -ti 4 0.3 8 0.1"
        "-s -mess 00101100011 -form NRZ -codeur -nbEch 40 -ampl -5 5 -ti 5 0.2 70 0.4"
    )

```

Figure 81 - Modification du script *runTests* pour l'itération 5

En plus d'ajouter des tests dans le script *runTests*, nous avons décidé d'ajouter une fonctionnalité importante pour détecter des bugs que nous aurions pu ne pas voir. En effet, nous avons remarqué à plusieurs reprises, lors de l'ajout de nouveaux paramètres, que des bugs apparaissaient. Par exemple, lorsque nous utilisions un paramètre précis derrière un autre paramètre, ou lorsque qu'un paramètre précis était placé au début de la commande. Malheureusement, pour détecter tous ces bugs, il faudrait appeler notre simulateur un très grand nombre de fois, en changeant le nombre et l'ordre des paramètres à chaque fois.

Pour ne pas avoir à faire cela, nous avons implémenté un nouveau bout de code dans le script *runTests*. Grâce à lui, dès que nous exécutons ce script, tous les tests que nous avons écrits depuis le début de ce projet, se retrouvent exécutés avec les paramètres qui sont dans un ordre aléatoire. Ainsi, après avoir exécuté ce script plusieurs fois, la probabilité qu'on ne remarque pas un bug du genre est très faible. Mettre en place cette méthode au début du projet nous aurait permis de corriger plus rapidement une grande partie de nos bugs, puisque la majorité venait des paramètres qui pouvaient être incompatible.

6.5. Bilan de l’itération

L’itération 5 de notre projet, consacrée au codage de canal, a permis d’intégrer des outils essentiels pour l’amélioration de la fiabilité de notre système de transmission dans des conditions bruitées. En développant et en intégrant les classes **Codeur** et **Decodeur**, nous avons réussi à implémenter un système de codage canal qui réduit significativement le taux d’erreur binaire (TEB), comme démontré par nos simulations.

Les essais effectués, notamment dans des conditions de transmission bruitées, ont validé la pertinence de notre stratégie de codage. L’introduction de cette nouvelle complexité s’est avérée justifiée au vu des améliorations notables du taux d’erreur binaire (TEB).

À travers cette itération, nous avons également amélioré notre processus de développement logiciel, en renforçant nos pratiques de tests et en peaufinant la documentation du code, assurant ainsi une meilleure maintenabilité de notre simulateur.

Nous avons également pu montrer, avec l’aide de plusieurs expériences, que la synchronisation entre l’émetteur et le récepteur se retrouve accrue lorsque le codage canal est utiliser. Ces dernières nous ont permis de voir que pour des valeurs de SNR par bit élevée notre simulateur ne laissait passer quasiment aucune erreur. Dans des environnements plus bruités, ce dernier se retrouvait un peu en difficultés, mais nous observions dans tous les cas une amélioration par rapport aux simulations sans codage canal.

En somme, cette itération a non seulement renforcé la robustesse de notre simulateur face aux perturbations extérieures, mais a aussi posé des bases solides pour les améliorations futures, alignées avec les objectifs de performance et de fiabilité de notre projet de simulation de système de transmission.

Tout cela nous sera utile lors de la mise en pratique présentée dans la prochaine partie. Nous aurons besoin d’utiliser notre simulateur sur un cas d’étude fourni par notre client, pour lui proposer une solution optimale qui réponde à ses besoins.