Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

Rapport de stage

Version 1.1

30 septembre 2022

Ingénieur Machine Learning Feature Extraction & Multiclass Classification

Madeleine Polycarpe
madeleine.polycarpe@ensta-bretagne.org

Résumé

Le stage faisant l'objet de ce rapport a été réalisé auprès de l'entreprise AVANTIX SAS du lundi 9 mai 2022, au vendredi 2 septembre 2022. Ce stage s'inscrit dans le cadre de la deuxième année de Formation d'Ingénieur sous Statut Etudiant de l'Ecole Nationale Supérieure de Techniques Avancées de Bretagne. L'étudiant est amené à endosser le rôle d'assistant technique ingénieur, ici dans le domaine du traitement du signal et de l'intelligence artificielle. L'objectif de ce stage, intitulé « Ingénieur Machine Learning – Feature Extraction & Multiclass Classification », est de déterminer si l'exploitation de signaux radars brutes est plus performante que celle des données obtenues à l'issue du prétraitement et de l'extraction des caractéristiques métiers pour l'apprentissage des algorithmes de Machine Learning. L'activité du stage se décompose en deux parties : la création d'un simulateur de signaux radar et la création d'une base de données simulées, puis l'exploitation de ces données afin d'entraîner divers algorithmes de Machine Learning à détecter et caractériser les signaux issus des radars appelés impulsions.

Mots clés

Rapport de stage, radar, ondes électromagnétiques, guerre électronique, simulateur de signaux, intelligence artificielle.

Abstract

The internship this report relates to was realised within AVANTIX SAS between Monday, May 9th, 2022, and Friday, September 2nd, 2022. This internship is a part of the ENSTA Bretagne (National School of Higher Education of Advanced Techniques of Britain) Engineering Training under Student Status' second year. The student is expected to assume the responsibilities of technical assistant engineer, here in the field of signal processing and artificial intelligence. The objective of this practice, untitled "Machine learning Engineer – Feature Extraction & Multiclass Classification", is to determine if the exploitation of radar signals using Machine Learning is more efficient than the exploitation of the characteristics extracted from them. The internship activity is divided in two parts: the creation of a radar signal simulator and the creation of simulated database, before exploiting this data to train various Machine Learning algorithms to recognise the transmitter broadcasting the signal.

Keywords

Internship report, radar, electromagnetic radiations, electronic warfare, signal simulator, artificial intelligence.

Table des matières

Introduction

Du lundi 9 mai 2022 au vendredi 2 septembre 2022, j'ai eu la chance de réaliser un stage en tant qu'assistant technique ingénieur au sein de l'entreprise AVANTIX SAS. Sous la responsabilité de Chakib BELAFDIL, l'intitulé de stage était le suivant : « Ingénieur Machine Learning - Feature Extraction & Multiclass Classification ».

Le but du stage consiste à déterminer la pertinence de l'utilisation du Machine Learning dans la détection et la caractérisation de signaux radar, par rapport à l'approche classique utilisant le traitement du signal. L'activité s'en est retrouvée décomposée en deux étapes. Tout d'abord, une première partie qui touche au traitement du signal, consistant à créer un simulateur de signaux radar, puis à créer une base simulée de signaux radars. Puis, une deuxième partie basée sur l'utilisation d'algorithmes de Machine Learning, où l'application de divers algorithmes de Machine Learning entrainés sur la base de données simulées permet de détecter les impulsions radars sur un enregistrement, puis de les caractériser.

Au-delà de l'intérêt technique qu'offre ce stage, il s'agissait également d'une opportunité de commencer à découvrir le domaine de l'industrie de défense, auquel je serai confrontée durant toute ma carrière en tant qu'Ingénieur en Etudes et Techniques de l'Armement. Travailler au profit des forces armées françaises est au cœur de ma motivation et cela a également contribué à mon choix de stage.

1 Présentation du contexte

1.1 Contexte général

Avantix SAS appartient à l'Entreprise de Services du Numérique (ESN) française Atos. Cette dernière définit sa mission comme étant de contribuer à façonner l'espace informationnel, tout en participant au développement de l'excellence scientifique et technologique. Avec plus de 111 000 employés à l'international, il s'agit de la première ESN française.

FIGURE 1 – Logo Avantix

FIGURE 2 – Logo Atos

Avantix, localisée à Aix en Provence, fut créée en 1979 sous le nom d'Amesys, avant d'être intégrée au groupe Bull en 2010. C'est ensuite au tour de Bull d'être racheté par Atos en 2014, qui centre pour sa part son activité sur le renseignement, la surveillance et la reconnaissance dans le domaine de la défense, par la conception de systèmes de hautes technologies permettant le contrôle du spectre électromagnétique, y compris les signaux de courte durée et de faible portée. L'activité réalisée par les quelques 560 employés se fait principalement au profit de la Direction Générale de l'Armement et des forces armées françaises.

Avantix appartient à l'unité de management Mission Critical System, au sein de la branche Big Data & Sécurité d'Atos, actuel fer de lance économique de l'entreprise. Son activité est divisée entre les domaines « Air, Land & Sea Electronics » et « Electronic Warfare ».

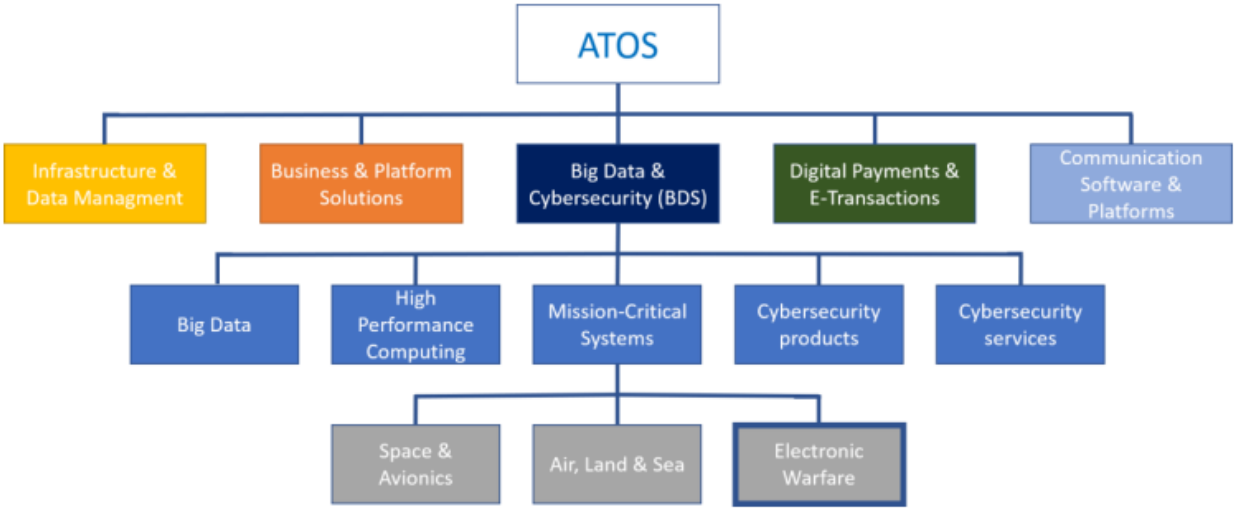

FIGURE 3 – Organisation des Unités de Management

69 des employés d'Avantix font partie du pôle dédié à la guerre électronique, divisé de la façon suivante :

FIGURE 4 – Organisation des équipes de l'unité de management Electronic Warfare

2 Problématisation du thème, des objectifs et des enjeux

2.1 Introduction à la guerre électronique

Depuis la Première Guerre Mondiale, qui a vu les premières utilisations des moyens radios à des fins opérationnelles [2], les armées ont graduellement accru leur dépendance au spectre électromagnétique. La guerre électronique est définie dans la doctrine militaire française comme « tout ce qui a trait aux opérations de combat effectuées dans l'environnement magnétique ». Trois domaines d'action distincts sont considérés : l'attaque, le soutien et la protection [1].

L'attaque électronique consistera à agir sur les émissions et réceptions de l'adversaire. Le brouillage, qui vise à rendre inexploitable les émissions électromagnétiques, se distingue du leurrage et de l'intrusion qui vise à induire en erreur les systèmes adverses humains ou non. Le but sera d'empêcher les communications ou la prise d'informations de l'ennemi, ou de les remplacer par de fausses informations.

Le soutien électronique, aussi appelé renseignement d'origine électro-magnétique (ROEM) ou Signal Intelligence (SIGINT) en anglais comprend deux facettes :

- Communication Intelligence (COMINT) : l'étude des communications afin d'intercepter les informations dont dispose l'adversaire.
- Electronic Intelligence (ELINT) : l'étude de signaux autres que les communications, notamment les signaux radars afin d'obtenir des renseignements sur les émetteurs, comme la position de l'ennemi, ou le type d'équipement utilisé par exemple.

Enfin, la protection électronique consiste à contrer les mesures d'attaque et de soutien de l'ennemi, en limitant les signaux qu'ils peuvent recevoir et la compréhension qu'ils peuvent en avoir. Il s'agit par exemple de limiter la signature radar d'un appareil, de choisir les bandes de fréquences utilisées de manière à masquer son activité ou de chiffrer ses communications.

2.2 Le système Avantix ELINT

Pour rappel, un radar, nommé ainsi pour Radio Detection And Ranging, émet des impulsions électromagnétiques et capte la réflexion de ces ondes dans son environnement pour obtenir des informations sur le dit environnement. Sont ainsi obtenues des informations sur la distance à laquelle peuvent se situer des objets, ainsi que potentiellement sur leur vitesse. Ce sont ces impulsions radar qui seront captées et exploitées par les systèmes d'Electronic Intelligence.

Le système Avantix ELINT, pour lequel ce sujet de stage a été édité, permet de mesurer la radiogoniométrie radar pour la surveillance d'une zone. Il assure les différentes fonctions suivantes :

- Interception des signaux radar
- Radiogoniométrie
- Enregistrement
- Traitement des données

— Analyse des données
— Stockage

L'entreprise est capable de fournir un système complet ou d'ajouter les fonctionnalités souhaitées à un système existant à la demande du client. Une large de bande de transmission radar est accessible, de 500Mhz à 18GHz.

2.3 Les enjeux de la guerre électronique

À la suite de la fin de la guerre froide et de l'effondrement du bloc soviétique, les puissances occidentales et particulièrement les Etats Unis se sont appuyés sur une connaissance approfondie des différents terrains d'intérêt stratégique afin de prévenir les conflits autant que faire se peut [7]. Cette connaissance, si elle a permis de maintenir pendant de nombreuses années une paix relative, s'appuie fortement sur un très fort développement des moyens électroniques de surveillance et de communications. Par conséquent, les différents partis sont devenus dépendants des dits moyens de télécommunications et d'autant plus vulnérables à la guerre électronique[3].
Face à la multiplication des conflits asymétriques et n'ayant pas fait face à un conflit de haute intensité depuis plusieurs décennies, les grandes puissances occidentales ont quelque peu délaissé la guerre électronique. L'utilisation de l'attaque électronique ciblée contre les soldats ukrainiens durant l'annexion de la Crimée a rappelé à chacun l'importance de la « guerre élec »[6].

2.4 La thématique du stage

Le stage, intitulé « Ingénieur Machine Learning - Feature Extraction & Multiclass Classification » était dans un premier temps décrit à l'aide des tâches suivantes :
— Créer un simulateur de signaux bruts
— Constituer une base de données de travail à partir du simulateur
— Proposer et implémenter des méthodes d'extraction de caractéristiques (de type spectrogramme entre-autres)
— Evaluer les performances des méthodes implémentées sur divers modèles de Machine Learning et Deep Learning
— Documenter les travaux et communiquer régulièrement vos résultats et avancements au groupe de travail

Pour relier ceci au système Avantix ELINT, le but serait de procéder au traitement et à l'analyse des données. L'entreprise dispose des caractéristiques associées aux émetteurs radar qu'il est possible de rencontrer sur un terrain opérationnel, fournies par le client. Le système actuellement implémenté utilise le traitement du signal pour identifier les caractéristiques de chaque impulsion, pour ensuite remonter au radar d'origine à l'aide de l'intelligence artificielle.

L'objectif du stage serait de déterminer si ce travail d'extraction des caractéristiques effectué pour l'instant à l'aide du traitement du signal pourrait être optimisé par l'utilisation du Machine Learning.

2.5 Evolutions du sujet de stage

La première étape mentionnée consiste à créer un simulateur de signaux bruts qui s'appuie sur les caractéristiques fournies par le client. À l'aide de ce simulateur sera constituée une base de données de travail de signaux bruts correspondant théoriquement aux mesures qui pourraient être réalisées sur le terrain. Cette étape, qui devait représenter seulement le début du stage, s'est retrouvée rallongée par la complexité du simulateur recherché. Cela a, de fait, laissé peu de temps pour l'extraction de caractéristique.

2.6 Importance du stage pour les différents acteurs

L'extraction des caractéristiques d'impulsions radar à l'aide du Machine Learning manque de précédents dans la littérature scientifique. Faire de l'exploration de cette option un sujet de stage permet à la personne qui s'y intéresse de s'y consacrer entièrement, sans fonctions transverses.

Le simulateur de signaux a par ailleurs vocation à être repris, modifié, complété et réutilisé pour évoluer avec les besoins de l'entreprise. Cela leur permet par ailleurs de s'affranchir du besoin de mesures fournies par le client dans un premier temps, et de la labélisation des enregistrements associés, très gourmande en moyens humains.

Du point de vue du ministère des armées, utiliser l'intelligence artificielle au bénéfice de la guerre électronique consisterait en un gain de temps humain important, donnée cruciale dans le contexte de manque de personnel actuel. Il est par ailleurs intéressant de noter que le Délégué Général pour l'Armement nouvellement nommé est spécialisé en Intelligence Artificielle.

3 Description de l'activité et des résultats du stage

3.1 Les termes utiles à la compréhension et les paramètres fournis

Un ensemble de données contenant des informations d'intérêt pour ce sujet fut fourni comme base de travail. Les impulsions émises par chaque radar seront caractérisées par les paramètres suivants :

— Durée d'impulsion : « Pulse width » (`pw`)
— Fréquence moyenne de l'impulsion : « Frequency » (`freq`)
— Durée entre le début de deux impulsions : « Pulse Repetition Interval » (`pri`)

Chaque radar simulé émettra donc des impulsions de durée et de fréquence déterminées. La différence entre les temps d'émission de deux pulsations sera également une donnée prise en compte. Cependant, ces données varient d'une impulsion à l'autre pour un même radar. Chacune de ces données est choisie pour les impulsions selon une séquence particulière mémorisée par le radar. L'ensemble de ces données est défini comme le « sous-mode » propre au radar. Les sous-modes utilisés pour définir les caractéristiques de chaque radar seront fournis sous la forme d'un fichier `pickle` contenant les paramètres nécessaires à la création des pulsations.

Les données fournies contiendront donc notamment pour chaque sous-mode les paramètres suivants :

— « Pulse width pattern » (`pw_pattern`) : contient les différentes durées d'impulsion successives possibles pour un sous-mode. La table à disposition ne contenant que des listes de longueur 1, il fut choisi de considérer que toutes les impulsions émises par un radar avaient la même durée d'impulsion et de ne pas tenir compte des cas ou le sous-mode possèderait plusieurs durées d'impulsion.
— « Frequency pattern » (`freq_pattern`) : contient les différentes fréquences moyennes successives auxquelles seront émises les impulsions d'un sous-mode.
— « Pulse Repetition Interval Pattern » (`pri_pattern`) : contient les différentes durées entre deux débuts d'impulsions successives auxquelles seront émises les impulsions d'un sous-mode. Certains sous-modes utilisent une loi de PRI de type « Jitter », auquel cas la variable `pri_pattern` prend la valeur d'un dictionnaire répertoriant deux flottants nommés « Min » et « Max ». La valeur de la PRI est alors choisie aléatoirement de manière uniforme entre ces deux valeurs minimum et maximum de la PRI.

Les tables fournies permettent également d'associer à chaque sous-mode la Puissance Equivalente Rayonnée de l'émetteur.

3.2 Présentation du simulateur de signaux

Idée globale

Le but est de simuler le signal reçu par un récepteur entouré de jusqu'à une dizaine de radar, dans un rayon de $100m$ à $40km$. Ces radars peuvent émettre sur entre $500MHz$ et $18GHz$, bande de fréquence subdivisée en différentes bandes d'écoute de largeur $1,5GHz$,

avec un recoupement d'au moins $100MHz$ entre deux bandes.

La fréquence d'échantillonnage sera fixée à $4GHz$. Afin de respecter le théorème de Shannon-Nyquist, chaque bande de fréquence est décalée pour coïncider avec la bande de fréquence $2.25GHz - 3.75GHz$. Ainsi, les phénomènes de recouvrement de spectre seront évités car aucun signal de fréquence inférieure à $2GHz$ n'est produit. L'utilisation d'une fréquence aussi basse est permise dans la réalité par l'utilisation de capteurs hétérodynes.

Le temps d'écoute est compris entre $100\mu s$ et $100ms$, et sera uniforme sur une base de données. L'impédance du convertisseur sera de 50Ω. La densité de bruit de la chaîne de réception sera initialisée à $-100dBm/kHz$ et la puissance de bruit de la conversion analogique numérique sera de $-60dBfs$.

Les radars émetteurs seront considérés pointés sur le récepteur, et le récepteur sera considéré isotrope.

Afin de limiter la taille des bases de données simulées, ne seront considérées pour chaque classe que les bandes de fréquence dîtes « d'intérêt ». Le choix de ces bandes, enregistrées dans chaque classe à l'aide de l'attribut « `reached_bands` », sera précisé pour chaque classe.

De nombreuses fonctions de tracé sont disponibles dans les diverses classes, afin de vérifier l'efficacité du code proposé. Certaines peuvent nécessiter d'être exécutées à l'aide du débugger, fenêtre de tracé par fenêtre de tracé, en fonction des capacités de la machine utilisée.

L'application de la méthode Agile a été privilégiée, en gardant une version exécutable du code à tout moment.

3.2.1 Générer les impulsions

La première étape consistait à générer pour chaque sous-mode ses impulsions, à l'aide de la formule suivante :

$$x(t) = Asin(2\pi ft + \phi)$$

avec t compris entre 0 et la durée d'impulsion associée au sous-mode.

A et ϕ sont initialisés à 1 et 0 respectivement. f est ici la « fréquence adaptée » obtenue pour chaque fréquence `freq` comprise dans la liste `freq_pattern`. Est appelée fréquence adaptée la valeur de fréquence obtenue en décalant le bande de fréquence considérée sur la bande de fréquence $2.25GHz - 3.75GHz$.

3.2.2 Passer à la Programmation Orientée Objet

La deuxième étape était de représenter les impulsions précédemment calculées à l'aide d'une classe. Nommée `SyntheticPulse`, cette dernière aura a pour but de calculer le signal de chaque impulsion, en fonction de la fréquence et de la durée d'impulsion fournies à l'initialisation. Les bandes de fréquence d'intérêt sont définies comme étant celles

auxquelles appartient la fréquence de l'impulsion, issue de la liste « `freq_pattern` ». Il peut y en avoir une ou deux si la fréquence appartient à la zone de recoupement entre deux bandes.

Dans le cas ou la fréquence de l'impulsion appartient à deux bandes de fréquence distinctes, deux enregistrements de l'impulsion sont calculés, un premier en utilisant la fréquence calculée en décalant la bande de fréquence la plus basse, puis un deuxième en utilisant la fréquence calculée en décalant la bande de fréquence la plus haute. Le calcul de ces nouvelles fréquences permet de retourner par la suite le signal calculé pour chacune.

La création de la classe `Transmitter` permet de générer toutes les impulsions existantes pour un sous-mode. Les bandes fréquences d'intérêt considérées seront l'ensemble des bandes de fréquence sur lesquelles une impulsion sera émise.

L'ensemble des impulsions possibles sera calculé à la création de l'instance de la classe `Transmitter`. Pour un radar dont la PRI ne suit pas une loi de type « Jitter », il est possible de définir un motif d'impulsions répété continuellement par le radar. En effet, les impulsions seront émises avec une fréquence initiale appartenant à un motif de fréquence `freq_pattern`, et en respectant une durée entre deux impulsions également définie par un motif `pri_pattern`. La longueur du motif répété par le radar sera alors le Plus Petit Commun Multiple des longueurs de ces deux motifs.

En effet, par définition du PPCM, il existe deux entiers k et l premiers entre eux tels que :

$$\begin{aligned} PPCM(\texttt{len(pri\_pattern)}, \texttt{len(freq\_pattern)}) &= k * \texttt{len(pri\_pattern)} \\ &= l * \texttt{len(freq\_pattern)} \end{aligned}$$

Ainsi, le radar émettra comme première impulsion I_0 une impulsion de fréquence initiale `freq_pattern[0]` et attendra un temps `pri_pattern[0]` à partir du début de la première impulsion pour émettre la deuxième. Par définition du PPCM, la première impulsion faisant suite au motif d'impulsions sera la première impulsion depuis I_0 à posséder les mêmes propriétés.

Dans le cas d'un radar de type Jitter, toutes les impulsions de fréquence initiale comprise dans `freq_pattern` seront initialisée. La durée entre deux impulsions sera choisie aléatoirement entre les valeurs minimale et maximale fournies par le dictionnaire `pri_pattern` à l'étape suivante.

3.2.3 Simuler l'environnement d'émission

L'environnement d'émission aura pour but de retourner au récepteur uniquement les signaux tels qu'ils seraient reçus sur le terrain, à savoir un enregistrement par bande d'écoute, sans distinction en fonction de l'émetteur de chaque impulsion. C'est le rôle de la classe `Environment` qui initialisera par ailleurs les instances de la classe `Transmitter`

qui représenteront nos émetteurs.

La classe `Transmitter` retournera à la classe `Environment` le signal émis sur chaque bande d'intérêt de `Transmitter` sur une période d'écoute débutant à un moment t_0 choisi aléatoirement compris entre 0 et la durée du motif de l'émetteur.

Ainsi, après avoir initialisé chaque impulsion et recréé ainsi le motif d'impulsions répété par l'émetteur sous forme de liste, la classe `Transmitter` s'attachera à créer le vecteur associé.

Elle s'appuie pour cela sur les temps de début de chaque impulsion, calculés à l'aide de la PRI de chaque impulsion, de la durée du motif et des fréquences initiales de chaque impulsion. Une méthode spécifique permettra de calculer cela pour une itération du motif des impulsions en fréquence d'un radar « Jitter », initialisant aléatoirement les PRI comme précisé précédemment.

Caractéristique du sous-mode et défini en fonction des motifs de fréquences et de durées entre le début de deux impulsions, le vecteur calculé représentera le signal sur un motif et aura une fréquence d'échantillonnage définie par celle des impulsions. Il s'agira donc de compléter par des zéros aux moments où aucune impulsion n'est émise. Ce vecteur est calculé sur chaque bande de fréquence d'intérêt de l'émetteur.

L'étape suivante est de générer pour chaque bande de fréquence d'intérêt un vecteur correspondant à l'émission d'un émetteur sur une période d'écoute. Ce dernier est obtenu pour chaque bande de fréquence par concaténation du motif du radar afin de correspondre à la durée recherchée, en tenant compte du temps de début d'écoute t_0.Une méthode spécifique assure la même fonction pour les radars de type « Jitter », notamment en regénérant un nouveau motif en PRI à chaque itération.

3.2.4 Simuler l'environnement de réception

Afin de simuler l'environnement de réception, il a été choisi de diviser les étapes de réception entre la classe `Environment`, déjà crée et chargée d'assurer les étapes ayant lieu avant interception des signaux radar, et la classe `Receptor`, chargée de toutes les actions effectuées par le récepteur et ayant lieu après interception. Leurs bandes de fréquence d'intérêt seront l'union des bandes de fréquence d'intérêt des divers émetteurs présents dans l'environnement. L'instance de la classe `Environment` qui nous intéresse sera crée à l'instanciation de la classe `Receptor`.

Le signal émis pendant la période d'écoute pour chaque émetteur sur chaque bande de fréquence d'intérêt de l'émetteur est récupéré, et la somme est calculée sur chaque bande de fréquence d'intérêt de l'environment afin d'obtenir le signal reçu par le récepteur.

3.2.5 Tenir compte des bruits et des atténuations

Le signal transmis par les émetteurs au récepteur est atténué lorsqu'il traverse l'environnement. Cette atténuation est calculée à l'aide de la formule de Friis [5] :

$$P_R = P_T * G_R * G_T * \frac{\lambda}{4 * \pi * R}^2$$

On calcule le produit de la puissance et du gain de l'antenne d'émission.

$$\begin{aligned} P_T * G_T &= PIRE \\ &= PIRE_{max} * \frac{x(t)}{x_{max}}^2 \\ &= 1.65ERP * x(t)^2 \end{aligned}$$

Par ailleurs,

$$G_R = 1$$

car l'antenne de réception est considérée isotrope.

$$\lambda = \frac{c}{f}$$

On calcule ainsi la puissance reçue par l'antenne de réception :

$$P_R = 1.65ERP * \frac{x(t) * c}{4 * \pi * f * R}^2$$

La Puissance Apparente Rayonnée (ERP pour Efficient Radiated Power en anglais) est fournie pour chaque sous-mode.

Ce calcul est effectué directement sur le motif de chaque émetteur. C'est ainsi la puissance du signal qui est récupérée par la classe `Environment` puis par la classe `Receptor`.

3.2.6 Assurer la conversion analogique numérique

La classe `Receptor` sera responsable de la conversion analogique numérique, puis de l'ajout des bruits de la chaîne de réception et de conversion analogique numérique au signal quantifié [8].

La densité de bruit de la chaîne de réception est initialisée à $-100dBm/kHz$ et la pleine échelle du convertisseur est fixée à $-30dBm$. Le bruit de la chaîne de réception est ainsi généré par une loi gaussienne d'écart-type σ. L'écart-type est calculé de la façon suivante :

$$\begin{aligned} \sigma &= \sqrt{VAR} \\ &= \sqrt{DSP} \\ &= \sqrt{\frac{\texttt{noise\_power}}{\texttt{sampling\_frequency}}} \end{aligned}$$

Sachant que la puissance du bruit est calculée de la façon suivante :

$$\begin{aligned}\texttt{noise\_power} &= \texttt{noise\_density} * \texttt{frequency\_bands\_width} \\ &= 10^{\frac{\texttt{noise\_density\_dbfs}}{10}} * \texttt{frequency\_bands\_width} \\ &= 10^{\frac{\texttt{noise\_density\_dbm} - \texttt{signal\_density\_dbm}}{10}} * \texttt{frequency\_bands\_width}\end{aligned}$$

Le bruit de la conversion analogique numérique est calculé de la même façon pour une puissance de bruit de $-60dBfs$. La valeur maximale de la tension obtenue suite à la conversion analogique numérique est calculée pour une impédance de 50 Ω.

Le bruit de la chaîne de réception est ajouté au signal avant conversion. Le maximale de la tension obtenue correspondra à 2^{15}, puisque le signal sera codé sur $16bits$ et qu'il a été choisi de garder les entiers négatifs. Le bruit de quantification sera donc multiplié par 2^{15} et ajouté au signal juste avant quantification.

3.2.7 Implémenter les modulations

Le choix a été fait d'accorder à chaque sous-mode une modulation. Sur 60 sous-modes, il a été choisi que 30 ne seraient pas modulés, 15 seraient modulés en fréquence et 15 seraient modulés en phase.

L'exploitation des données fournies par le client a permis de mettre en évidence l'utilisation pour les modulations de phase de séquences de Barker [4], ainsi que de séquences détaillées dans les données, associé à un déphasage. Chaque impulsion modulée en fréquence sera donc divisée en un nombre de tronçons égal à la longueur de la séquence considérée, composée de 1, de 0 et de -1. La phase de chaque tronçon sera alors le produit du coefficient associé au tronçon et de la valeur de déphasage associée au sous-mode. Varier les associations entre ces angles et les séquences de Barker a permis de créer douze possibles modulations, auxquelles s'ajoutent les séquences détaillées dans les données.

Toutes les modulations en fréquence seront des chirps. Les fréquences de modulation sont choisies linéairement en fonction de la durée d'impulsion. Le maximum de la fréquence d'impulsion est initialisé à $200MHz$ et est atteint pour une valeur minimum de la durée d'impulsion, tandis que le minimum de la fréquence d'impulsion de $50MHz$ est atteint pour une valeur maximum de la durée d'impulsion. La moitié des modulations en fréquence correspondront à des chirps ascendants et l'autre moitié à des chirps descendants.

Le DataFrame contenant les sous-modes sera modifiée à l'initialisation pour ajouter à chaque sous-mode la modulation correspondante. Cette version modifiée sera sauvegardée et réutilisée pour les simulations suivantes.

Il s'agit alors de modifier la méthode de calcul des impulsions dans la classe `SyntheticPulse`.

3.2.8 Définir le format des fichiers rendus

La classe `Receptor` génère un dossier contenant les informations relatives à l'écoute enregistrée dans l'environnement associé, nommé en fonction d'un identifiant et de la date.

Cedossier contient trois fichiers distincts, appelés respectivement `labels`, `data` et `config`.

A partir des temps de début et des fréquences de chaque impulsion, la classe `Transmitter` calculera, en plus de l'écoute sur une bande pour l'émetteur considéré, les temps d'émission et les fréquences caractéristiques des impulsions enregistrées sur cette écoute. La classe `Environment` génèrera une liste unique, répertoriant pour chaque impulsion un tuple contenant le temps de début d'émission, de fin d'émission, la fréquence, la durée d'impulsion, la distance de l'émetteur au récepteur et l'identificateur de l'émetteur dans l'environnement. Les listes de chaque bande d'écoute de fréquence sont transformées en array pour être triées par la suite en fonction du temps de début d'impulsion. Le fichier `labels` sera généré à partir d'un DataFrame répertoriant les caractéristiques de chaque impulsion, et la bande de fréquence sur laquelle elle a été émise, qui sera enregistré au format `csv`.

Le fichier `data` sera un fichier `csv` contenant un tableau dont chaque colonne représentera une bande de fréquence d'intérêt pour le récepteur et chaque ligne correspondra à l'enregistrement obtenu à l'instant t répertorié dans la première colonne.

Le fichier `config` sera un fichier `txt` une recopie partielles des « Settings » utilisés, ainsi que la liste des caractéristiques propres à chaque émetteur créé dans l'environnement, qui ne sont pas répertoriées dans le DataFrame initial caractérisant les sous-modes.

3.2.9 Les limites du simulateur

Les radars sont considérés pointés sur le récepteur, et le récepteur est considéré isotrope. Si cette simplification facilite grandement l'implémentation du simulateur, elle est cependant peu représentative de la réalité, et mériterait d'être corrigée.

Puisque le motif de durée d'impulsion était de longueur égale à 1 pour chaque sous-mode de la table de référence, le simulateur ne tient pas compte de la possibilité qu'un sous-mode ait différentes durées d'impulsion.

Lors de la création d'une écoute d'un radar de type « Jitter », l'enregistrement sur chaque bande de fréquence d'intérêt pour le radar est calculé tour à tour. Puisque plusieurs motifs sont générés aléatoirement à l'enregistrement de chaque bande, les écoutes sur les différentes bandes de fréquence d'intérêt ne correspondront pas à la même période d'émission du radar.

3.3 La base de données générée

Le simulateur présenté précédemment a permis de générer des bases de données de la forme suivante :

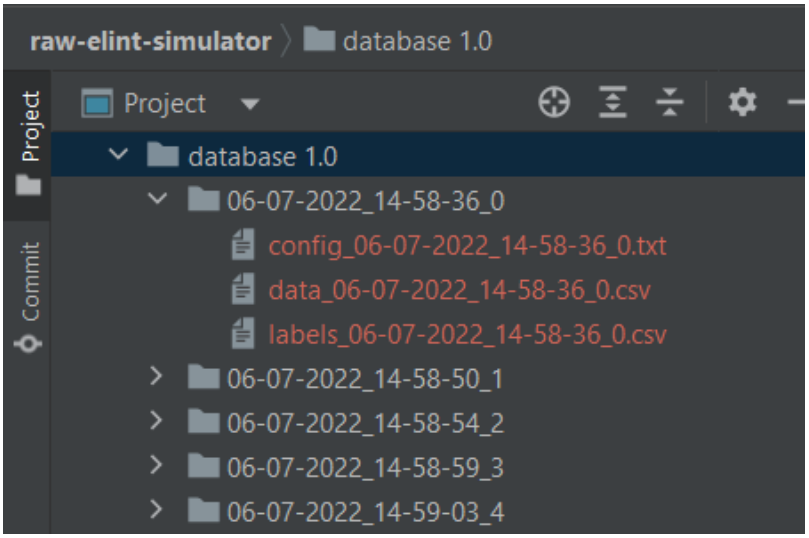

FIGURE 5 – Architecture des fichiers pour une base de données

Chaque dossier est identifié par sa date de création et un numéro, unique par base de données, et contient trois fichiers :

— Le fichier « `config` », au format `txt`, contiendra tous les paramètres d'initialisation de l'environnement d'émission de l'enregistrement. Il contiendra également la liste des émetteurs, ainsi que leurs paramètres de création.

— Le fichier « `data` », au format `csv`, contient un tableau répertoriant les enregistrements simulés du récepteur sur les différentes bandes de fréquence d'écoute d'intérêt. La première colonne du tableau contiendra l'échelle de temps de l'enregistrement.

— Le fichier « `labels` », au format `csv`, contiendra les informations relatives à chaque impulsion, à savoir un numéro d'identification unique par bande d'écoute, le début de l'impulsion sur l'échelle de temps disponible dans le fichier « `data` », la fin de l'impulsion, la fréquence moyenne, la distance entre le récepteur et l'émetteur de l'impulsion, l'identification de l'émetteur et la bande de fréquence sur laquelle l'impulsion sera captée. L'accès à l'identification de l'émetteur donnera par ailleurs accès à la modulation de l'impulsion notamment.

```
### Settings ###

SAMPLING_FREQUENCY = 4000000000.0
FREQUENCY_BANDS = [5.00e+08 1.90e+09 3.30e+09 4.70e+09 6.10e+09 7.50e+09 8.90e+09 1.03e+10
 1.17e+10 1.31e+10 1.45e+10 1.59e+10 1.65e+10]
FREQUENCY_BANDS_WIDTH = 1500000000.0
OVERLAP = 100000000.0
MIN_FREQUENCY_BANDS = 500000000.0
MAX_FREQUENCY_BANDS = 18000000000.0
LISTENING_TIME = 0.0001
MINIMAL_DISTANCE = 100.0
MAXIMAL_DISTANCE = 40000.0
IMPEDANCE = 50.0
MIN_FREQUENCY_FMOP = 5000000.0
MAX_FREQUENCY_FMOP = 200000000.0


### Transmitters ###

Transmitter_id, Submode, Distance, MOP
[Transmitter 0, 56, 9785.3249443684, ('PMOP', (180, [1, 1, 1, -1, -1, 1, -1]))]
[Transmitter 1, 46, 31916.429477416685, ('PMOP', (180, [1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
[Transmitter 2, 22, 16631.128572100894, No MOP]
[Transmitter 3, 40, 7002.995323004132, ('PMOP', (60, [1, 1, 1, -1, -1, 1, -1]))]
[Transmitter 4, 59, 21997.070579387302, ('PMOP', (90, [1, 1, 1, -1, -1, 1, -1]))]
[Transmitter 5, 57, 28151.326406418695, ('FMOP', 196870093.5046752)]
[Transmitter 6, 35, 27011.984637042857, ('PMOP', (45, [1, 1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1]))]
[Transmitter 7, 40, 15050.650518015447, ('PMOP', (60, [1, 1, 1, -1, -1, 1, -1]))]
[Transmitter 8, 2, 17614.569038778067, No MOP]
```

FIGURE 6 – Exemple d'un fichier `config`

3.4 L'analyse exploratoire

L'analyse exploratoire des bases de données générées a pour but de confirmer que ces dernières sont représentatives des données d'origine. Cela a permis à plusieurs reprises de corriger le code du simulateur. Ainsi, sur 5 bases de données, seules les deux dernières ont été conservées. La cinquième base de données possède l'avantage d'être bien plus volumineuse puisque qu'elle contient 500 simulations conte 50 pour la quatrième base de données. Les paramètres d'intérêt considérés sont les suivants:

— Le nombre d'émetteurs pour chaque environnement

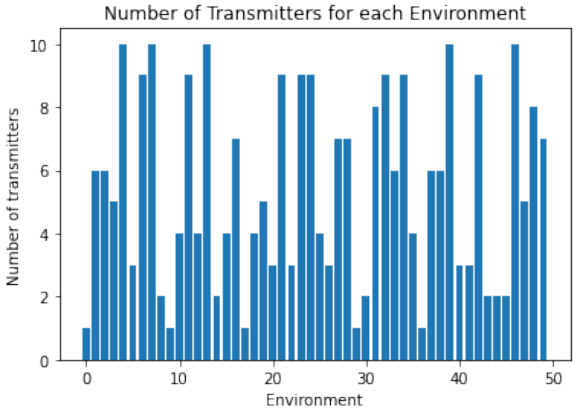

FIGURE 7 – Nombre d'émetteurs pour chaque environnement dans la base de données 4

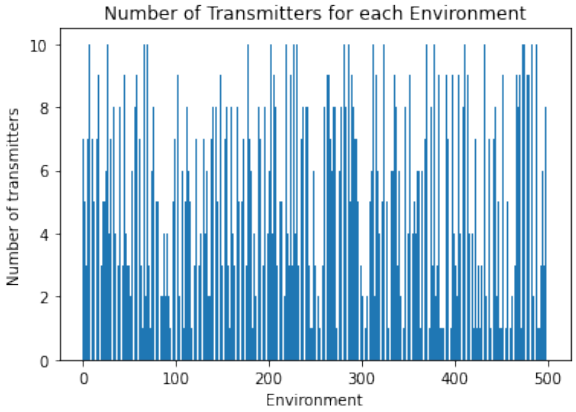

FIGURE 8 – Nombre d'émetteurs pour chaque environnement dans la base de données 5

— Le nombre d'émetteurs de chaque sous-mode

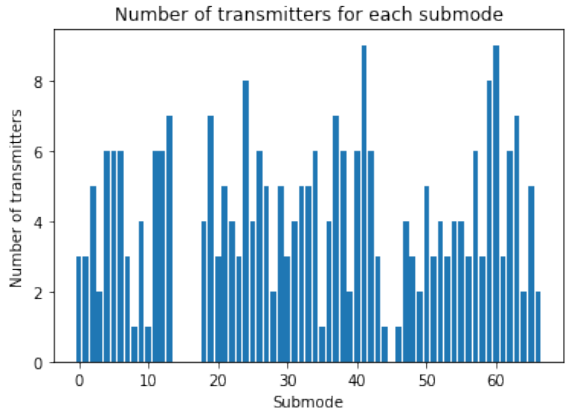

FIGURE 9 – Nombre d'émetteurs de chaque sous-mode dans la base de données 4

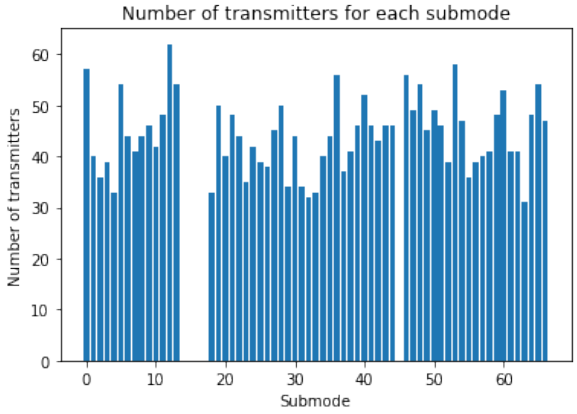

FIGURE 10 – Nombre d'émetteurs de chaque sous-mode dans la base de données 5

— Le nombre d'émetteurs de chaque sous-mode par environnement

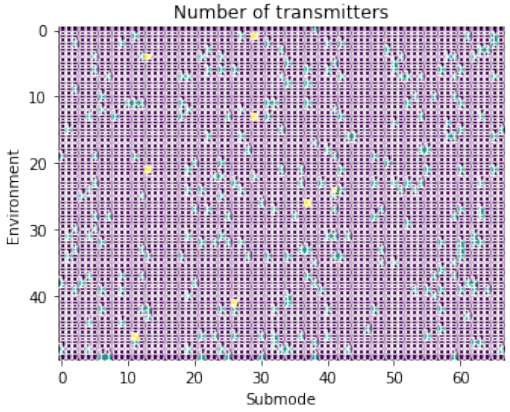

FIGURE 11 – Nombre d'émetteurs de chaque sous-mode par environnement dans la base de données 4

— Le nombre d'impulsions par acquisition

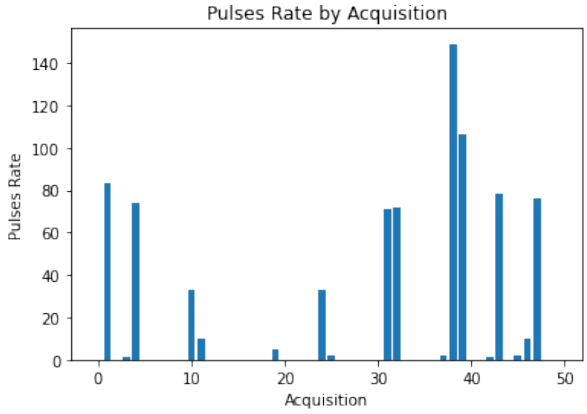

FIGURE 12 – Nombre d'impulsions par acquisition dans la base de données 4

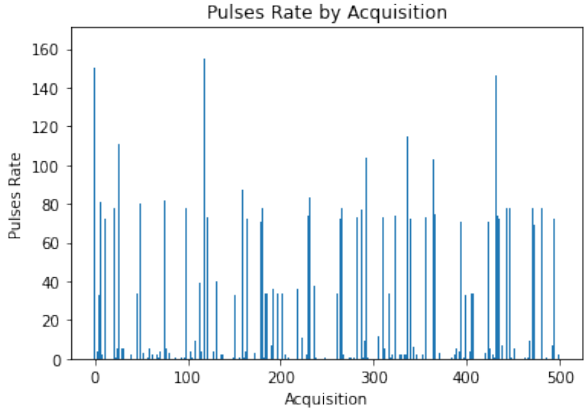

FIGURE 13 – Nombre d'impulsions par acquisition dans la base de données 5

— Le nombre de bandes écoutées par environnement

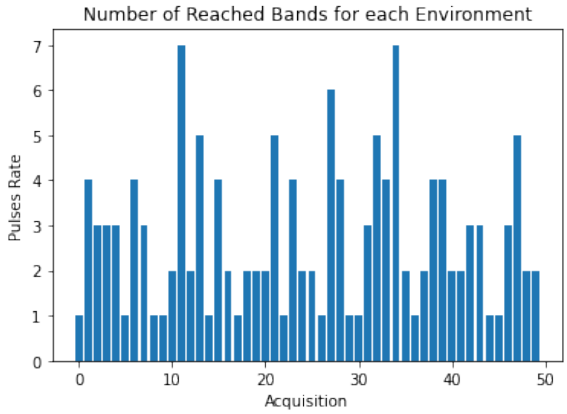

FIGURE 14 – Nombre de bandes écoutées par environnement dans la base de données 4

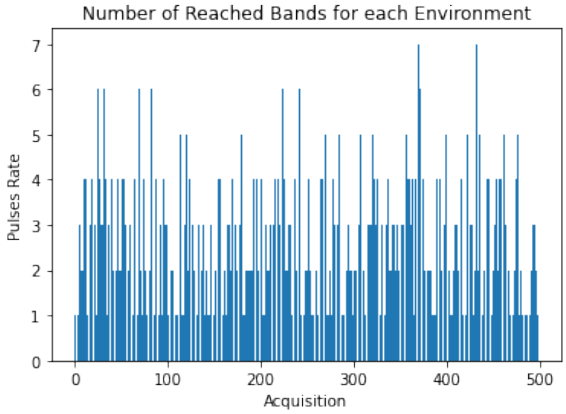

FIGURE 15 – Nombre de bandes écoutées par environnement dans la base de données 5

— Les bandes écoutées pour chaque environnement

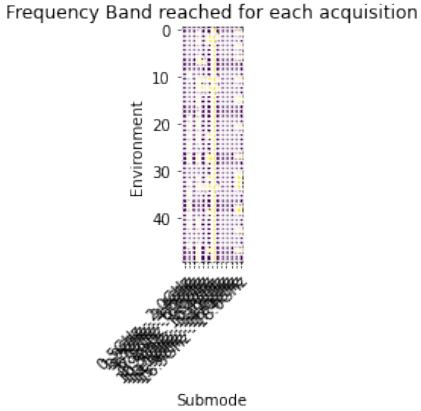

FIGURE 16 – Bandes écoutées pour chaque environnement dans la base de données 4

— La distance entre émetteur et récepteur (moyenne, médiane, par environnement et global)

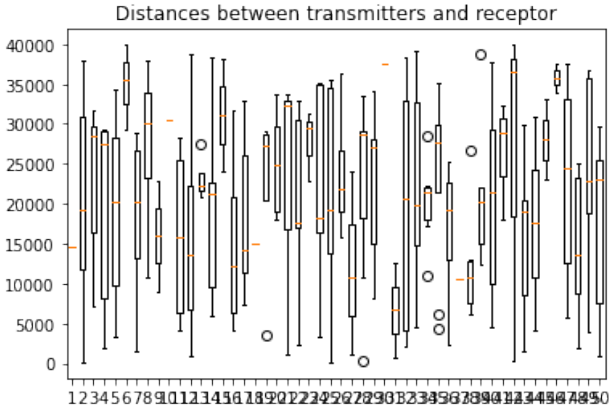

FIGURE 17 – Distance entre émetteur et récepteur dans la base de données 4

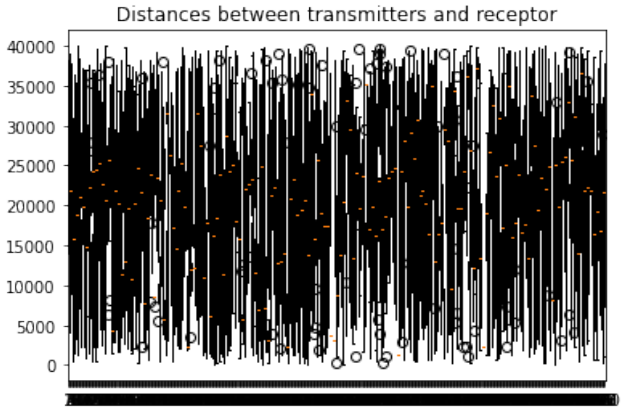

FIGURE 18 – Distance entre émetteur et récepteur dans la base de données 5

Les paramètres propres aux impulsions, et non aux enregistrements sont également considérés.

— La fréquence pour chaque impulsion, par émetteur, pour chaque environnement.

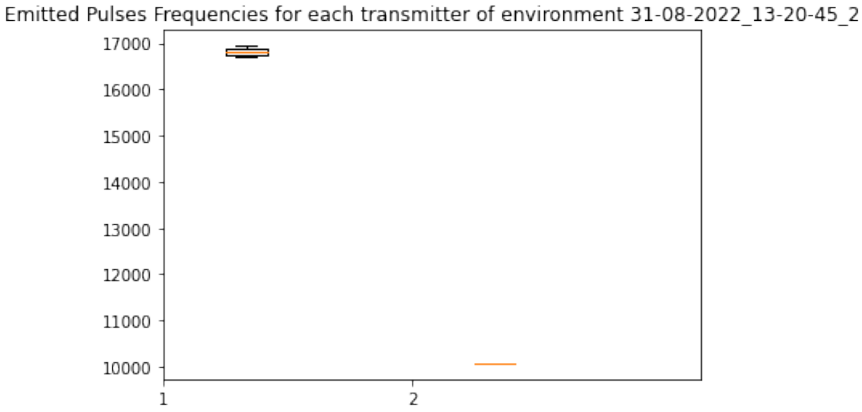

FIGURE 19 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 2 de la base de données 4

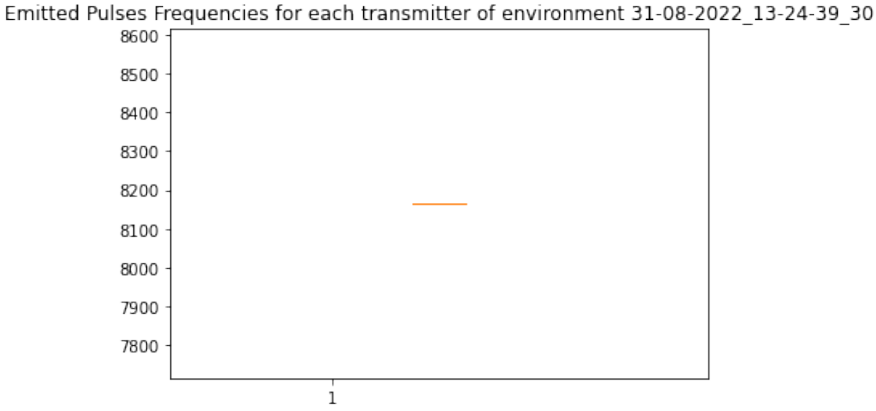

FIGURE 20 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 30 de la base de données 4

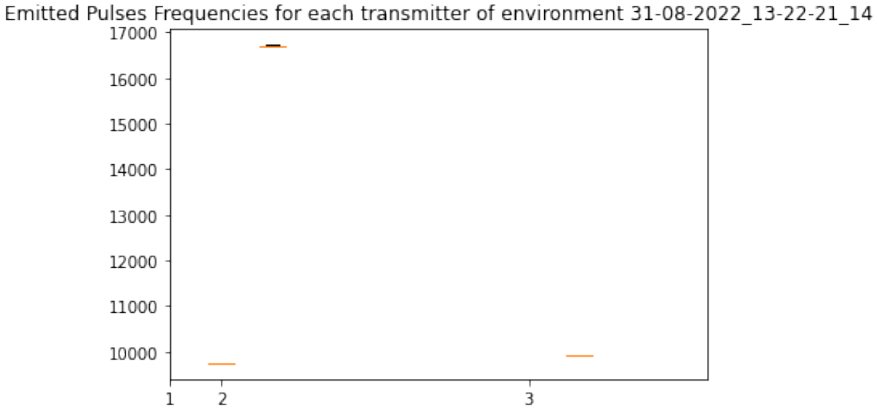

FIGURE 21 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 14 de la base de données 4

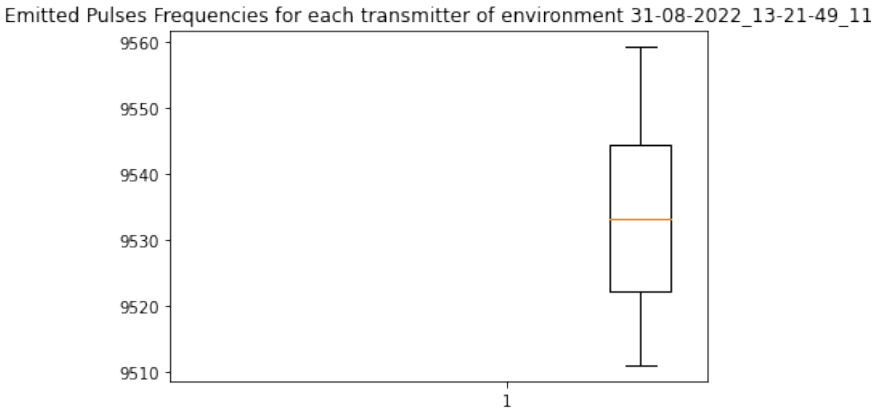

FIGURE 22 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 11 de la base de données 4

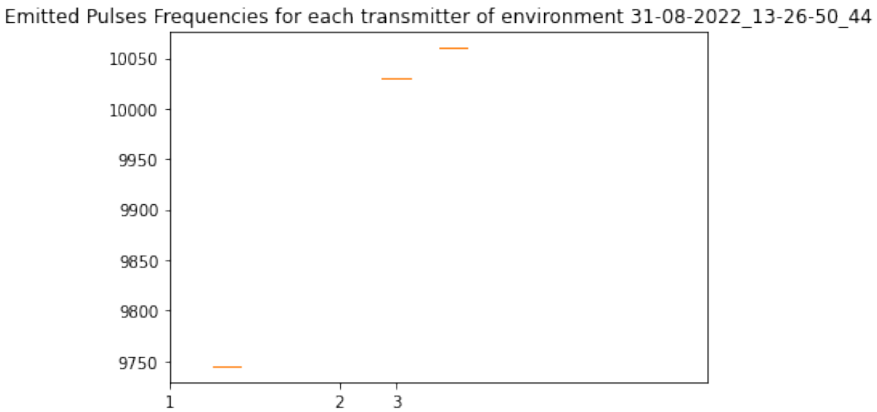

FIGURE 23 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 44 de la base de données 4

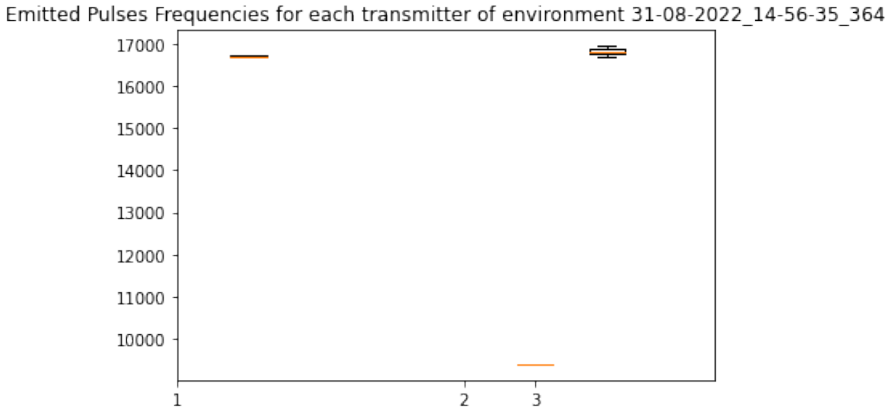

FIGURE 24 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 364 de la base de données 5

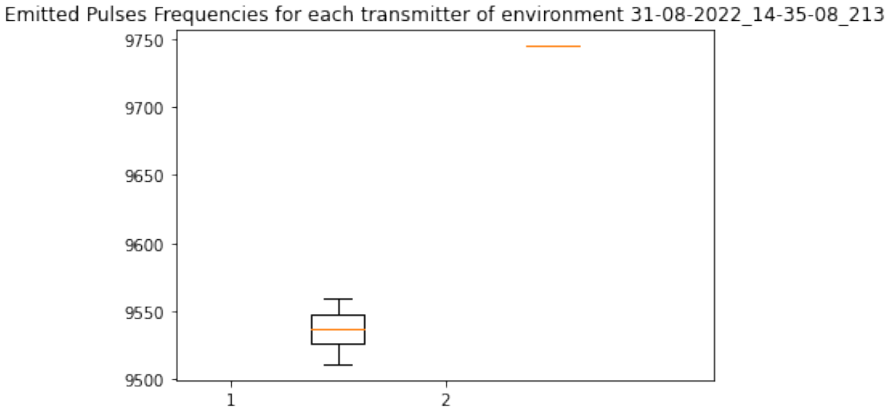

FIGURE 25 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 213 de la base de données 5

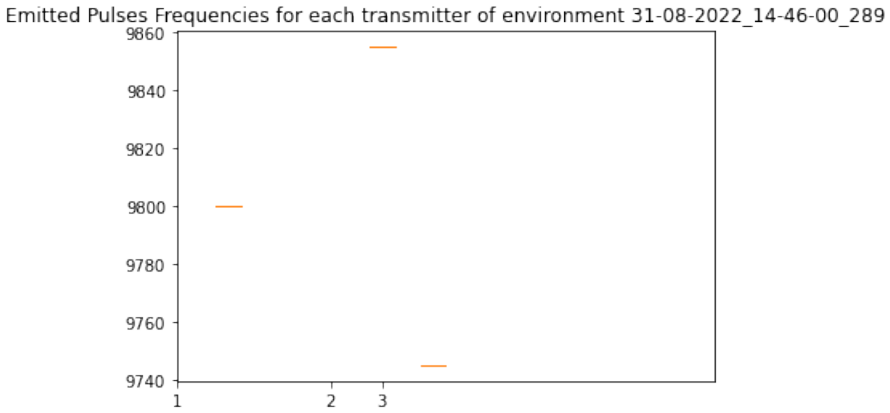

FIGURE 26 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 289 de la base de données 5

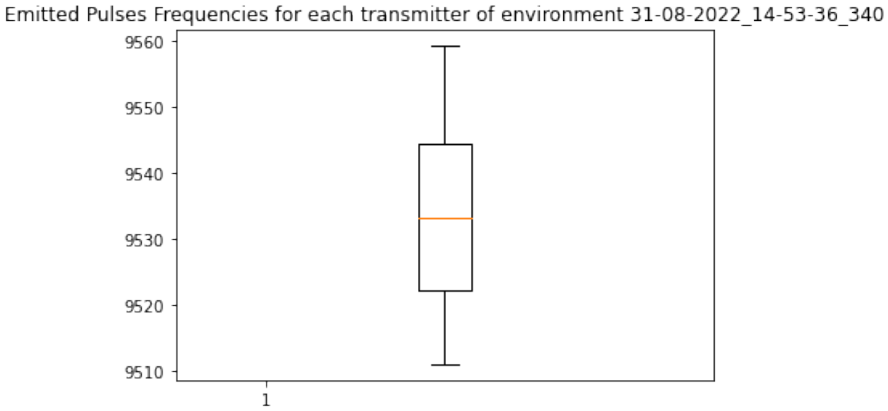

FIGURE 27 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 340 de la base de données 5

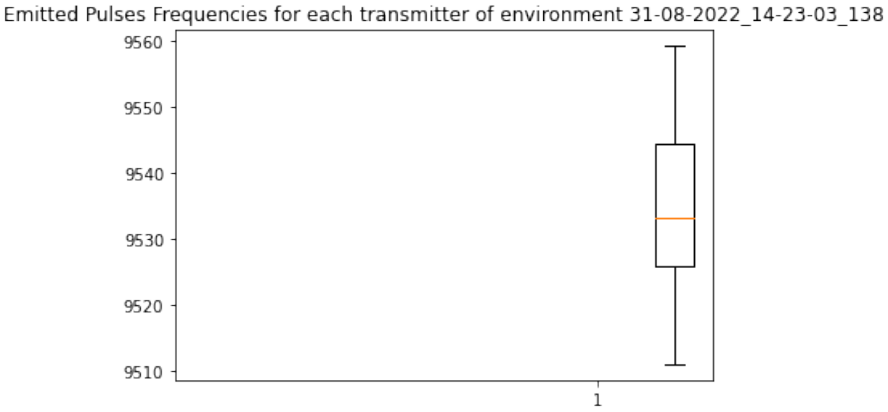

FIGURE 28 – Fréquence pour chaque impulsion, par émetteur, pour l'environnement 138 de la base de données 5

— La fréquence pour chaque impulsion, par environnement.

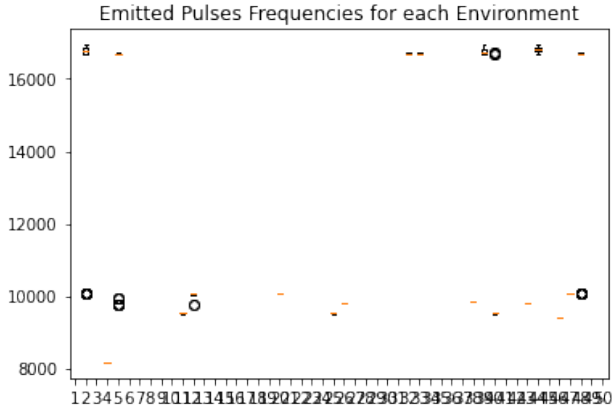

FIGURE 29 – Fréquence pour chaque impulsion, par environnement dans la base de données 4

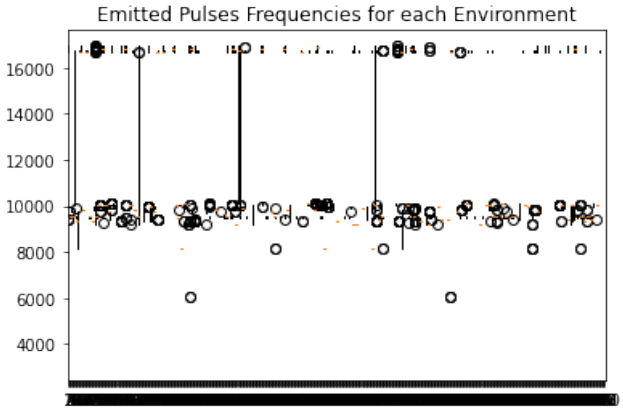

FIGURE 30 – Fréquence pour chaque impulsion, par environnement dans la base de données 5

— La durée d'impulsion pour chaque impulsion, par environnement.

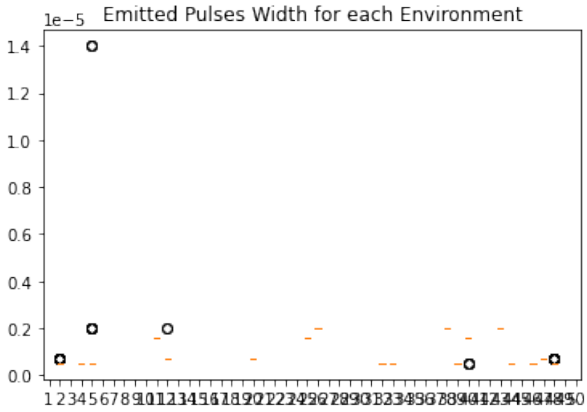

FIGURE 31 – Durée d'impulsion pour chaque impulsion, par environnement dans la base de données 4

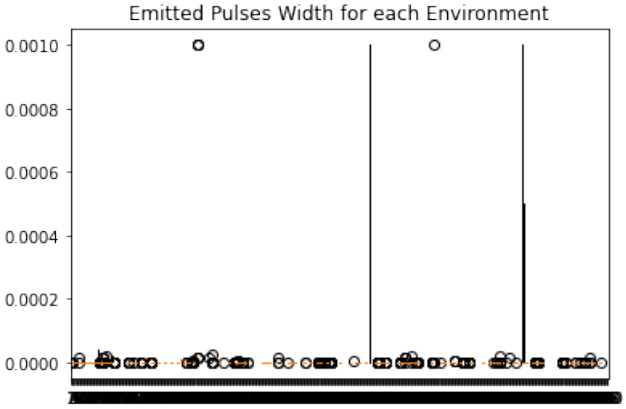

FIGURE 32 – Durée d'impulsion pour chaque impulsion, par environnement dans la base de données 5

— Le nombre d'impulsions de chaque type de modulation

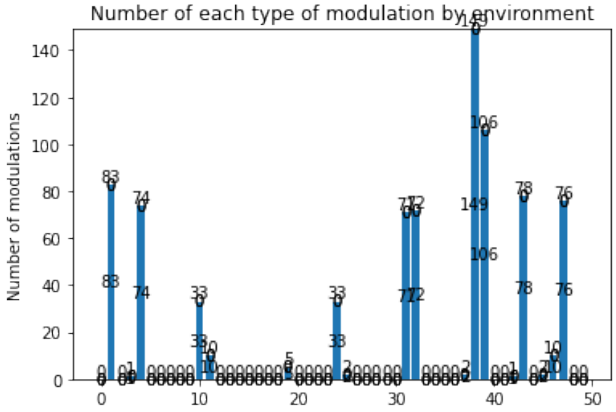

FIGURE 33 – Nombre d'impulsions de chaque type de modulation dans la simulation 4

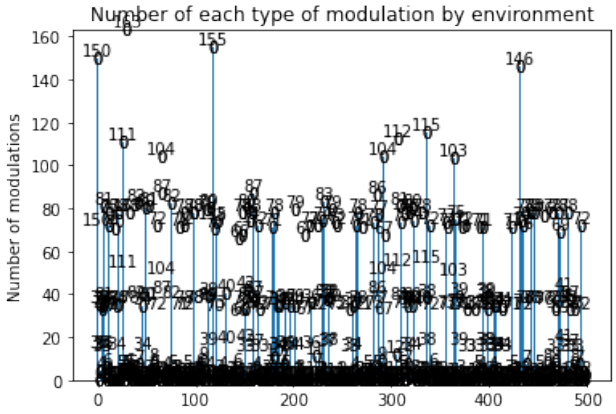

FIGURE 34 – Nombre d'impulsions de chaque type de modulation dans la simulation 5

3.5 L'exploitation des signaux générés

L'exploitation des signaux générés comprendra plusieurs étapes. Tout d'abord, il est utile de rappeler que les caractéristiques à identifier pour chaque impulsion sont : la durée d'impulsion, la fréquence moyenne d'impulsion, l'intervalle de répétition entre deux impulsions et la modulation de l'impulsion. Il s'agira tout d'abord d'isoler les impulsions sur le signal avant de pouvoir identifier leurs caractéristiques.

3.5.1 L'utilisation du spectrogramme

Pour identifier les impulsions, il a été choisi par l'entreprise de représenter le spectrogramme des signaux de la base de données avant d'exploiter ces derniers à l'aide d'algorithmes de Machine Learning. Le spectrogramme est une représentation temps-fréquence du signal. Il va représenter la puissance émise en fonction du temps en abscisses et de la fréquence en ordonnée, par une carte de fréquentation.

Les impulsions de fréquence constante seront alors représentées par un segment horizontal plus clair, tandis que les signaux modulés en fréquence, qui sont ici des chirps, seront représentés par un segment diagonal. Les abscisses et les ordonnées de ces segments seront indicatifs de la fréquence et des temps d'émission des impulsions.

3.5.2 Identification des impulsions

Pour identifier une impulsion, il est nécessaire d'obtenir son temps de début et de fin, ainsi que sa fréquence. L'avantage de l'utilisation du spectrogramme est que cela revient à repérer les abscisses et ordonnées minimales et maximales de la représentation de l'impulsion sur le spectrogramme.

Le but de l'algorithme de Machine Learning employé est donc de tracer un cadre (ou "bounding box") autour de chaque objet d'intérêt. Les coordonnées de ce cadre correspondent aux coordonnées en temps et en fréquence de l'impulsion étudiée.

Les algorithmes de segmentation d'image envisagés dans cet objectif reposaient tous sur les réseaux de neurones convolutifs. Ces derniers se décomposent en deux blocs de réseau de neurones, le premier, spécifique aux réseaux de neurones convolutifs,ayant pour rôle d'extraire les caractéristiques des données d'entrées et fournissant ses conclusions au deuxième bloc, lequel utilisera les dîtes caractéristiques pour classifier lesdonnées d'entrées.

Le modèle U-Net a in fine été préféré aux modèles YOLO (pour You Only Look Once) ou RCNN (Region-based Convolutional Neural Network). Cette partie a été confiée à Axel Ducamp, stagiaire, principalement par manque de temps.

Un seul spectrogramme est représenté par écoute. Par conséquent, la précision peut manquer. Il pourrait être nécessaire de représenter plus précisément la période correspondant à une impulsion.

3.5.3 Extraction des caractéristiques des impulsions

Parmi les caractéristiques d'intérêt de nos signaux, seule la modulation ne découle pas des informations obtenues sur le spectrogramme. Les représentations temps-fréquence des signaux modulés en fréquence peuvent facilement être distinguées de celles des signaux sans modulation ou avec modulation d'angles à l'aide d'un algorithme de classification d'images. Là encore, il sera peut-être nécessaire de tracer une représentation plus précise du spectrogramme de l'impulsion, afin de distinguer les segments horizontaux des segments diagonaux, plutôt que d'avoir seulement accès aux points accessibles sur le spectrogramme de l'ensemble de l'acquisition.

La distinction des signaux non modulés et avec modulation d'angles ne peut pas simplement se faire par exploitation du spectrogramme. Ce problème n'a pas été traité par manque de temps.

3.6 Evaluation des modèles testés

Les résultats de l'évaluation du modèle U-Net sont les suivants :

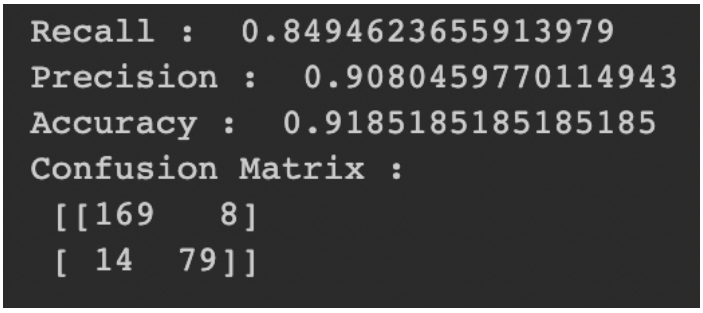

FIGURE 35 – Résultats d'évaluation de notre modèle

On observe une plus grande difficulté du modèle à cadrer la fréquence de l'impulsion, probablement du fait d'erreurs de labélisation lors du passage de fréquence d'un signals à pixels d'un spectrogramme. Des exemples de reconnaissance d'impulsion par le modèle sont présentés en annexe A.6.

Conclusion

La conception d'un simulateur de signaux radar a permis de balayer l'ensemble des diverses notions impliquées dans la chaîne de traitement du signal. Si le simulateur est prévu pour couvrir nombre de situations variées, il gagnerait néanmoins à prendre en compte les diagrammes d'antennes des radars émetteurs. L'exploitation des signaux simulés mérite davantage de temps, ne serait-ce qu'afin d'identifier les impulsions. S'il n'est pas possible en l'état de répondre à la problématique, la production du simulateur de signaux constitue réellement la valeur ajoutée du stage pour l'entreprise.

A Annexe

A.1 Classe `SyntheticPulse`

```
# We define here our class Synthetic Pulse. The main method is compute_pulse
# which output one or two frequency bands where the pulse is received

from bisect import bisect

import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as sig

from raw_simulator.config_settings import get_settings

global_params = get_settings("GLOBAL")
SAMPLING_FREQUENCY = float(global_params['SAMPLING_FREQUENCY'])
FREQUENCY_BANDS = global_params['FREQUENCY_BANDS']


class SyntheticPulse:
    def __init__(self, frequency, pw, modulation_frequency=None,
     phase_modulation_pattern=None):
        """
        Method that instantiate class Synthetic Pulse.
            Parameters:
                frequency: float, central emission frequency of our pulse, in MHz.
                pw: float, pulse width, in µs.
                phase: float, phase amplitude in degrees for phase modulation if
                 phase_modulation_pattern is not None.
                modulation_frequency: float, difference between the beginning and
                 end frequency for frequency modulated signal, in Hz.
                phase_modulation_pattern: numpy.ndarray, pattern transmitted
                 through phase modulation. Coefficients are within {-1, 0, 1}
weighted by the angle
            Returns:
                self: SyntheticPulse.
        """
        # Define predefined attributes
        self.frequency = frequency * 1e6
        self.pw = pw
        self.time_steps = np.arange(0., self.pw, 1. / SAMPLING_FREQUENCY)
        self.modulation_frequency = modulation_frequency
        self.phase_modulation_pattern = phase_modulation_pattern

        # Compute computation frequency(ies) and band(s).
```

```
        self.adapted_frequencies, self.bands = self.__shift_frequency()

        # Compute pulse on each reached band.
        self.pulses = self.compute_pulse()

    def __shift_frequency(self):
        """
        Method that computes the frequency shift ie. the pulse's central frequency
        translated within the 2.25GHz - 3.75GHz interval, which we will use to
        acquire the pulse with a lower sampling frequency. It is a known signal
        processing technique that shift the real bandwidth signal in a lower bandwidth
        signal in order to sample it with a lower sampling frequency.
        In some cases, the frequency of the pulse can belong to two frequency
        bands so the method return both in this case.
            Parameters:
                self: SyntheticPulse.
            Returns:
                list, containing the shifted frequency. In case the central frequency
                    belongs to two frequency bands, contains both reduced
                    frequencies otherwise only one band is returned.
                list, containing the smallest frequency of each band containing
                    the central frequency. Elements belong to FREQUENCY_BANDS.
        """
        i = bisect(FREQUENCY_BANDS, self.frequency - 1.5e9)
        f1 = self.frequency - FREQUENCY_BANDS[i] + 2.25e9
        if self.frequency > FREQUENCY_BANDS[i + 1]:
            f2 = self.frequency - FREQUENCY_BANDS[i + 1] + 2.25e9
            return [f1, f2], [FREQUENCY_BANDS[i], FREQUENCY_BANDS[i + 1]]
        return [f1], [FREQUENCY_BANDS[i]]

    def compute_pulse(self):
        """
        Method that computes the pulses for each of the reduced frequencies if there
        are several so that we can compute the signal with a limited sampling frequency.
        If the central frequency belongs to two frequency bands, the method returns
        both reduced frequencies.
            Parameters:
                self: SyntheticPulse.
            Returns:
                pulses: list, contains the numpy.ndarray containing our signal.
        """
        pulses = []
        if self.phase_modulation_pattern is not None:
            # Compute pulses in case of phase modulation.
            # Our pulse is divided in l segments, l being the length of our phase
            # modulation pattern. Each segment is computed with a phase equal
```

```
            # to the produce of the phase amplitude and the appropriate
            # coefficient of our pattern (within {-1; 0; 1}).

            # Compute the phases pattern
            phases = np.zeros(np.shape(self.time_steps))
            k = len(self.time_steps) // len(self.phase_modulation_pattern)
            for i, phase_value in enumerate(self.phase_modulation_pattern):
                phases[i * k:i * (k + 1)] = phase_value
            phases[(len(self.phase_modulation_pattern) - 1) * k:] =
             self.phase_modulation_pattern[-1]

            # Compute the list of pulses
            for ad_f in self.adapted_frequencies:
                pulses.append(np.sin(2 * np.pi * ad_f * self.time_steps + phases))
        else:
            # Compute pulses in case of frequency modulation or in
            # the absence of modulation.

            # Compute the list of pulses
            for ad_f in self.adapted_frequencies:
                # If the modulation frequency is null, the pulse is computed
                # as s(t) = A * sin(2*pi*f*t)
                signal = sig.chirp(self.time_steps,
                                   f0=ad_f - self.modulation_frequency / 2,
                                   f1=ad_f + self.modulation_frequency / 2,
                                   t1=self.pw,
                                   method='linear')
                pulses.append(signal)
        return pulses

    def display(self):
        """
        Method that displays the synthetic pulse previously computed on
        each of the frequency bands reached.
            Parameters:
                self: SyntheticPulse.
        """
        plt.figure()
        for i in range(len(self.bands)):
            plt.subplot(len(self.bands), 1, i + 1)
            plt.plot(self.time_steps, self.pulses[i])
            plt.title("Signal with frequency " + str(round(self.frequency, 2)) +
                      " over band " + str(round(self.bands[i] * 1e-9, 2)) +
                      'GHz - ' + str(round(self.bands[i] * 1e-9, 2) + 1.5) + 'GHz')
        plt.show()
```

```
def display_fft(self, width=512):
    """
    Method that displays the Fast Fourier Transform of the different segments
    of specified width of the synthetic pulse previously computed on each of
    the frequency bands reached.
        Parameters:
            self: SyntheticPulse.
            width: int, number of points computed on each segment.
    """
    for i in range(len(self.bands)):
        for j in range(len(self.pulses[i]) // width):
            fft = np.fft.fft(self.pulses[i][j * width:(j + 1) * width])
            frequency_steps = np.fft.fftfreq(width, width / SAMPLING_FREQUENCY)
            plt.figure()
            plt.plot(frequency_steps, fft)
            plt.title("Fast Fourier Transform of segment " + str(j) +
                      " of Synthetic Pulse over band " +
                      str(round(self.bands[i] * 1e-9, 2)) + 'GHz - ' +
                      str(round(self.bands[i] * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_stft(self):
    """
    Method that displays the Short Fourier Transform of the synthetic pulse
    previously computed on each of the frequency bands reached.
        Parameters:
            self: SyntheticPulse.
    """
    for i in range(len(self.bands)):
        frequency_steps, time_steps, stft = sig.stft(self.pulses[i],
              SAMPLING_FREQUENCY,
          nperseg=32)
        plt.figure()
        plt.pcolormesh(time_steps, frequency_steps, np.abs(stft))
        plt.title("Short Time Fourier Transform of Synthetic Pulse over band " +
                  str(round(self.bands[i] * 1e-9, 2)) + 'GHz - ' +
                  str(round(self.bands[i] * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_spectrogram(self):
    """
    Method that displays the spectrogram of the synthetic pulse previously
    computed on each of the frequency bands reached.
        Parameters:
            self: SyntheticPulse.
    """
```

```
for i in range(len(self.bands)):
    frequency_steps, time_steps, spectrogram =
     sig.spectrogram(self.pulses[i], SAMPLING_FREQUENCY,
                           nperseg=32)
    plt.figure()
    plt.pcolormesh(time_steps, frequency_steps, spectrogram)
    plt.title("Spectrogram of Synthetic Pulse over band " +
                str(round(self.bands[i] * 1e-9, 2)) + 'GHz - ' +
                str(round(self.bands[i] * 1e-9, 2) + 1.5) + 'GHz')
plt.show()
```

A.2 Classe `Transmitter`

```
# We define here our class Transmitter.

import random
from bisect import bisect

import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as sig

from raw_simulator.config_settings import get_settings, get_submodes
from raw_simulator.SyntheticPulse import SyntheticPulse

global_params = get_settings("GLOBAL")
SAMPLING_FREQUENCY = float(global_params['SAMPLING_FREQUENCY'])
FREQUENCY_BANDS = global_params['FREQUENCY_BANDS']
FREQUENCY_BANDS_WIDTH = float(global_params['FREQUENCY_BANDS_WIDTH'])
MINIMAL_DISTANCE = float(global_params['MINIMAL_DISTANCE'])
MAXIMAL_DISTANCE = float(global_params['MAXIMAL_DISTANCE'])
LISTENING_TIME = float(global_params['LISTENING_TIME'])


class Transmitter:
    def __init__(self, path_submodes_add, submode_number=-1, freq_pattern=None,
       pw_pattern=None, pri_pattern=None, dist=-1):
        """
        Method that instantiate class Transmitter.
            Parameters:
                path_submodes_add: str, path to submodes table.
                submode_number: int, number of the submode of created transmitter
                freq_pattern: list, pattern of frequencies respected by the
                 transmitter's pulses, in MHz.
                pw_pattern: list, contains only one pulse width, used by
                 every pulse created by our Transmitter, in µs.
                pri_pattern: list, contains the pattern of pulse repetition
                 intervals, respected by the transmitter, in µs.
                dist: int, distance between our transmitter and the
                 receptor associated to the environment, in m.
            Returns:
                self: Transmitter.
        """
        self.submodes = get_submodes(path_submodes_add)

        # Define basic attributes
        if submode_number == -1:
```

```
            self.submode_number = random.choice(self.submodes.index)
        else:
            self.submode_number = submode_number
        if freq_pattern is None:
            self.freq_pattern = self.submodes.freq_pattern[self.submode_number]
        else:
            self.freq_pattern = freq_pattern
        if pw_pattern is None:
            self.pw = self.submodes.pw_pattern[self.submode_number][0] * 1e-6
        else:
            self.pw = pw_pattern[0] * 1e-6
        if pri_pattern is None:
            self.pri_pattern = self.submodes.pri_pattern[self.submode_number]
        else:
            self.pri_pattern = pri_pattern
        self.time_steps = np.arange(0., self.pw, 1. / SAMPLING_FREQUENCY)
        self.modulation_characteristics = self.submodes.modulation[self.submode_number]
        self.modulation_frequency = 0
        if type(self.modulation_characteristics) == tuple:
            self.phase, self.modulation_pattern = self.modulation_characteristics
        else:
            self.modulation_frequency = self.modulation_characteristics

        # Define the dictionary of pulses signals, which contains, for each
        # reached frequency band, the whole pattern
        # of pulses possibly computed by our transmitter as a list of numpy.ndarray.
        self.pulses_pattern = self.compute_pattern_submode()

        # Compute the list of bands over which our transmitter broadcasts.
        self.reached_bands = self.compute_bands()

        # Compute a dictionary of the times of emission, which contains, for each
        # reached frequency band, the
        # numpy.ndarray of the times of emission of the different pulses over the
        # transmitter's pattern on the
        # specified band; the duration of the pattern; and a dictionary of the
        # frequencies of the pulses, which
        # contains, for each reached frequency band, the numpy.ndarray of
        # the frequencies of the different pulses over
        # the transmitter's pattern on the specified band.
        self.emission_times, self.pattern_duration, self.frequencies =
         self.compute_pattern_emission_times()
        if dist == -1:
            self.distance = random.random() * (MAXIMAL_DISTANCE -
              MINIMAL_DISTANCE) +
MINIMAL_DISTANCE
```

```
        else:
            self.distance = dist

        # Compute the dictionary containing, for each of the reached
        # frequency bands, the whole signal over a pattern,
        # included the zero values between pulses, as a numpy.ndarray.
        self.patterns = self.generate_pattern()

        # Compute the attenuated power received by the receptor of the
        # pattern transmitted
        self.attenuated_power = self.compute_attenuated_power()

    def compute_pattern_submode(self):
        """
        Method that computes the different Synthetic Pulses possible
        for our Transmitter, taking into account
        modulation and returns the complete list of possible pulses signals.
            Parameters:
                self: Transmitter.
            Returns:
                pulses_pattern: dict, contains for each of the reach frequency
                    band the list of all the pulses computed
                    on this band over the transmitter's pattern.
        """
        # Initialize variables
        pulses_pattern = {}
        for f in FREQUENCY_BANDS:
            pulses_pattern[f] = []

        # The pattern's length will be the least common multiplier of the
        # length of the pulse repetition intervals
        # pattern and frequency pattern. This is decided assuming the
        # pulse width pattern contains a single element.
        # Then : n = k * length(pri_pattern) = l * length(freq_pattern)
        # The pattern will contain k repetitions of pri_pattern and l
        # repetitions of freq_pattern, and, a property of
        # the least common multiplier, there can not be a smaller
        # number which would allow to have a complete number of
        # both patterns.
        n = np.lcm(len(self.pri_pattern), len(self.freq_pattern))
        if isinstance(self.pri_pattern[0], dict):
            n = len(self.freq_pattern)

        for i in range(n):
            f = self.freq_pattern[(i % len(self.freq_pattern))]
```

```
            # Compute pulses with phase modulation
            if type(self.modulation_characteristics) == tuple:
                phase_modulation_pattern = np.array(self.modulation_pattern)
                 * self.phase * np.pi / 180
                pulse = SyntheticPulse(frequency=f, pw=self.pw,
                 phase_modulation_pattern=phase_modulation_pattern)

            # Compute pulses with frequency modulation and without modulation
            else:
                pulse = SyntheticPulse(frequency=f, pw=self.pw,
                 modulation_frequency=self.modulation_frequency)

            # Adds computed pulse to the list of pulses over the pattern
            for j in range(len(pulse.bands)):
                pulses_pattern[pulse.bands[j]].append(pulse.pulses[j])

        return pulses_pattern

    def compute_bands(self):
        """
        Method that returns the list of bands reached by the transmitter.
            Parameters:
                self: Transmitter.
            Returns:
                reached_bands: list, contains the element of
                 FREQUENCY_BANDS on which at least a pulse is
emitted by our Transmitter.
        """
        reached_bands = []
        for band in FREQUENCY_BANDS:
            if self.pulses_pattern[band]:
                reached_bands.append(band)
        return reached_bands

    def compute_pattern_emission_times_jitter(self):
        """
        Method that computes the emission times, the duration time and
        the frequency pattern of our pattern for submode with priLawType Jitter.
            Parameters:
                self: Transmitter
            Returns:
                emission_times: dict, contains, for each reached frequency band,
                    the numpy.ndarray of the times of emission of the different
                    pulses over the transmitter's pattern on the specified band.
                t: float, the duration of the pattern
                frequencies: dict, contains, for each reached frequency band,
```

```
                the numpy.ndarray of the frequencies of  the different pulses
                over the transmitter's pattern on the specified band.
        """
        # Initialise variables
        t = 0
        emission_times = {}
        frequencies = {}
        for band in self.reached_bands:
            emission_times[band] = []
            frequencies[band] = []

        for i in range(len(self.freq_pattern)):
            for band in self.reached_bands:
                # Add pulse emission_time and frequency to adapted
                # dictionary if pulse is emitted on frequency band.
                if self.freq_pattern[i] - band < 1.5e9:
                    emission_times[band].append(t)
                    frequencies[band].append(self.freq_pattern[i])
            # Increment pattern duration time with random value of
            # pulse repetition interval.
            t += (np.random.random() * (self.pri_pattern[0]['Max'] -
             self.pri_pattern[0]['Min']) +
                  self.pri_pattern[0]['Min']) * 1e-6

        # Convert dictionaries lists to numpy.ndarray
        for band in self.reached_bands:
            emission_times[band].append(t)
            emission_times[band] = np.array(emission_times[band])
            frequencies[band] = np.array((frequencies[band]))
        return emission_times, t, frequencies

    def compute_pattern_emission_times(self):
        """
        Method that computes the emission times, the duration
        time and the frequency pattern of our pattern.
            Parameters:
                self: Transmitter
            Returns:
                emission_times: dict, contains, for each reached
                    frequency band, the numpy.ndarray of the times of
                    emission of the different pulses over the transmitter's
                    pattern on the specified band.
                t: float, the duration of the pattern
                frequencies: dict, contains, for each reached frequency
                    band, the numpy.ndarray of the frequencies of
                    the different pulses over the transmitter's pattern
```

```
                on the specified band.
    """
    # Case where priLaw is a Jitter
    if isinstance(self.pri_pattern[0], dict):
        return self.compute_pattern_emission_times_jitter()

    # Initialise variables
    t = 0
    emission_times = {}
    frequencies = {}
    for band in self.reached_bands:
        emission_times[band] = []
        frequencies[band] = []

    # The pattern's length will be the least common multiplier
    # of the length of the pulse repetition intervals
    # pattern and frequency pattern. This is decided assuming
    # the pulse width pattern contains a single element.
    # Then : n = k * length(pri_pattern) = l * length(freq_pattern)
    # The pattern will contain k repetitions of pri_pattern and l
    # repetitions of freq_pattern, and, a property of
    # the least common multiplier, there can not be a smaller
    # number which would allow to have a complete number of
    # both patterns.
    for i in range(np.lcm(len(self.pri_pattern), len(self.freq_pattern))):
        f = self.freq_pattern[(i % len(self.freq_pattern))]
        for band in self.reached_bands:
            # Add pulse emission_time and frequency to adapted
            # dictionary if pulse is emitted on frequency band.
            if f - band < 1.5e9:
                emission_times[band].append(t)
                frequencies[band].append(self.freq_pattern[
                 (i % len(self.freq_pattern))])

        # Increment pattern duration time with pulse repetition interval.
        t += self.pri_pattern[i % len(self.pri_pattern)] * 1e-6

    # Convert dictionaries lists to numpy.ndarray
    for band in self.reached_bands:
        emission_times[band].append(t)
        emission_times[band] = np.array(emission_times[band])
        frequencies[band] = np.array((frequencies[band]))
    return emission_times, t, frequencies

def display(self, band=None):
    """
```

```
        Method that displays the different pulses our Transmitter
        previously computed on a specified band of
        frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency
                 band displayed, belong to reached bands.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        for i in range(len(self.pulses_pattern[band])):
            plt.figure()
            plt.plot(self.time_steps, self.pulses_pattern[band][i])
            plt.title("Pulse of frequency " +
                  str(self.freq_pattern[i % len(self.freq_pattern)]) +
              " over band " + str(round(band * 1e-9, 2)) + 'GHz - ' +
              str(round(band * 1e-9, 2) + 1.5) + "GHz emitted at " +
                      str(self.emission_times[band][i]))
        plt.show()

    def display_fft(self, width=32000, band=None):
        """
        Method that displays the Fast Fourier Transform of the
        different segments of specified width of the different
        pulses our Transmitter previously computed on a
        specified band of frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency
                band displayed, belong to reached bands.
                width: int, number of points displayed on each segment
                represented.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        if width > len(self.pulses_pattern[band][0]):
            width = 2 ** int(np.log2(len(self.pulses_pattern[band][0])))
        for j in range(len(self.pulses_pattern[band])):
            for i in range(len(self.pulses_pattern[band][j]) // width):
                if any(self.pulses_pattern[band][j]
                 [i * width:(i + 1) * width] !=
np.zeros(width)):
                    fft = np.fft.fft(self.pulses_pattern[band]
                     [j][i * width:(i + 1) * width])
                    frequency_steps = np.fft.fftfreq(width,
                     width / SAMPLING_FREQUENCY)
```

```
                plt.figure()
                plt.plot(frequency_steps, fft)
                plt.title("Fast Fourier Transform of segment " +
                      str(j) + " of pulse of frequency " +
                          str(self.freq_pattern[i % len(self.freq_pattern)]) +
                          " over band " + str(round(band * 1e-9, 2)) +
                          'GHz - ' + str(round(band * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_stft(self, band=None):
    """
    Method that displays the Short Time Fourier Transform
    of the different pulses our Transmitter previously
    computed on a specified band of frequencies.
        Parameters:
            self: Transmitter
            band: float, the smallest  frequency of frequency
            band displayed, belong to reached bands.
    """
    if band is None:
        band = random.choice(self.reached_bands)
    for i in range(len(self.pulses_pattern[band])):
        frequency_steps, time_steps, stft = sig.stft(self.pulses_pattern[band][i],
         SAMPLING_FREQUENCY)
        plt.figure()
        plt.pcolormesh(time_steps, frequency_steps, np.abs(stft))
        plt.title("Short Time Fourier Transform of pulse of frequency " +
                  str(self.freq_pattern[i % len(self.freq_pattern)]) +
                  " over band " + str(round(band * 1e-9, 2)) +
                  'GHz - ' + str(round(band * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def generate_pattern(self):
    """
    Method that computes the dictionary containing, for each of
    the bands reached, the whole signal over a pattern
    as a numpy.ndarray, including the null values between each pulse.
        Parameters:
            self: Transmitter
        Returns:
            transmitted_signals: dict, contains, for each of the reached
                frequency bands, a numpy.ndarray the
                contains the signal over a pattern, including the null
                values between each pulse.
    """
    # Initialise variables
```

```
        transmitted_signals = {}
        for band in self.reached_bands:
            signal = np.zeros(int(self.pattern_duration * SAMPLING_FREQUENCY))
            segments = []
            index = [0]

            # Add iteratively the pulse and pulse repetition interval null
            # signal to the
            for i in range(len(self.pulses_pattern[band])):
                # Compute the number of points to represent of the next
                # pulse repetition interval.
                n_pts = int((self.emission_times[band][i + 1] -
                             self.emission_times[band][i]) * SAMPLING_FREQUENCY)
                index.append(index[-1] + n_pts)

                # Create a numpy.ndarray with the previously computed
                # number of points.
                large_pulse = np.zeros(n_pts)
                # First points of the vector take value of the pulse
                # existing on this interval
                large_pulse[:len(self.pulses_pattern[band][i])]
                        = self.pulses_pattern[band][i]
                # Add interval numpy.ndarray to list
                segments.append(large_pulse)

            # Convert list of successive numpy.ndarray to large numpy.ndarray.
            for i in range(len(index) - 1):
                signal[index[i]:index[i + 1]] = segments[i]
            transmitted_signals[band] = signal
        return transmitted_signals

    def display_pattern(self, width=32000, band=None):
        """
        Method that displays the different segments of specified width
        of the pattern our Transmitter previously
        computed on a specified band of frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency band
                      displayed, belong to reached bands.
                width: int, number of points displayed on each segment
                      represented.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        for i in range(len(self.patterns[band]) // width):
```

```
        if any(self.patterns[band][i * width:(i + 1) * width] !=
         np.zeros(width)):
            plt.figure()
            plt.plot(self.patterns[band][i * width:(i + 1) * width])
            plt.title("Segment " + str(i) + " of pattern over band " +
                      str(round(band * 1e-9, 2)) + 'GHz - ' +
                      str(round(band * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_fft_pattern(self, width=32000, band=None):
    """
    Method that displays the Fast Fourier Transform of the
    different segments of specified width of the pattern our
    Transmitter previously computed on a specified band
    of frequencies.
        Parameters:
            self: Transmitter
            band: float, the smallest  frequency of frequency
                  band displayed, belong to reached bands.
            width: int, number of points displayed on each
                  segment represented.
    """
    if band is None:
        band = random.choice(self.reached_bands)
    for i in range(len(self.patterns[band]) // width):
        if any(self.patterns[band][i * width:(i + 1) * width] != np.zeros(width)):
            fft = np.fft.fft(self.patterns[band][i * width:(i + 1) * width])
            frequency_steps = np.fft.fftfreq(width, width / SAMPLING_FREQUENCY)
            plt.figure()
            plt.plot(frequency_steps, fft)
            plt.title("Fast Fourier Transform of segment " + str(i) +
                      " of pattern over band " +
                      str(round(band * 1e-9, 2)) + 'GHz - ' +
                      str(round(band * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_stft_pattern(self, width=32000, band=None):
    """
    Method that displays the Short Time Fourier Transform
     of the different segments of specified width of the
    pattern our Transmitter previously computed on a
    specified band of frequencies.
        Parameters:
            self: Transmitter
            band: float, the smallest  frequency of frequency
                  band displayed, belong to reached bands.
```

```
            width: int, number of points displayed on each
                 segment represented.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        for i in range(len(self.patterns[band]) // width):
            if any(self.patterns[band][i * width:(i + 1) * width] !=
                       np.zeros(width)):
                frequency_steps, time_steps, stft =
                       sig.stft(self.patterns[band][i * width:(i + 1) * width],
                                                SAMPLING_FREQUENCY)
                plt.figure()
                plt.pcolormesh(time_steps, frequency_steps, np.abs(stft))
                plt.title("Short Time Fourier Transform of segment " + str(i) +
                          " of pattern over band " +
                         str(round(band * 1e-9, 2)) + 'GHz - ' +
                         str(round(band * 1e-9, 2) + 1.5) + 'GHz')
        plt.show()

    def new_jitter_pattern(self):
        """
        Method that computes a new pattern for our Transmitter
        of priLawType Jitter, since there is no fixed pattern for
        those transmitters.
            Parameters:
                self: Transmitter
            Returns:
                attenuated_power: dict, """
        if isinstance(self.pri_pattern, dict):
            # Compute a new set of emission_times and pattern
            # duration randomly
            self.emission_times, self.pattern_duration, self.frequencies =
                      self.compute_pattern_emission_times_jitter()

            # Compute a new pattern dictionary using the newly
            # computed emission_times
            self.patterns = self.generate_pattern()

            # Compute the attenuated power transmitted to the receptor
            # associated to the environment by our transmitter,
            # applying the attenuation function to our newly computed pattern.
            self.attenuated_power = self.compute_attenuated_power()
        return self.attenuated_power

    def compute_attenuated_power(self):
        """
```

```
    Method that returns the power transmitted to the receptor
    associated to the environment, in a dictionary
    addressing each band.
        Parameters:
            self: Transmitter
        Returns:
            attenuated_power: dict, contains, for each of the frequency
                bands reached, the numpy.ndarray
                representing the power transmitted to the receptor
                associated to the environment by our Transmitter.
    """
    erp = self.submodes.erp[self.submode_number]
    attenuated_power = {}
    for band in self.reached_bands:
        attenuated_power[band] = 1.65 * erp * ((self.patterns[band] * 3e8 /
                                               (4 * np.pi *
                                               (band + (FREQUENCY_BANDS_WIDTH / 2)) *
                                               self.distance))
                                                ** 2)
    return attenuated_power

def display_attenuated_power(self, width=32000, band=None):
    """
    Method that displays the different segments of specified width
    of the attenuated transmitted power our
    Transmitter previously computed on a specified band of frequencies.
        Parameters:
            self: Transmitter
            band: float, the smallest  frequency of frequency band
                    displayed, belong to reached bands.
            width: int, number of points displayed on each segment represented.
    """
    if band is None:
        band = random.choice(self.reached_bands)
    for i in range(len(self.attenuated_power[band]) // width):
        if any(self.attenuated_power[band][i * width:(i + 1) * width] !=
                np.zeros(width)):
            plt.figure()
            plt.plot(self.attenuated_power[band][i * width:(i + 1) * width])
            plt.title("Segment " + str(i) +
                        " of attenuated transmitted power over band " +
                    str(round(band * 1e-9, 2)) + 'GHz - ' +
                    str(round(band * 1e-9, 2) + 1.5) + 'GHz')
    plt.show()

def display_fft_attenuated_power(self, width=32000, band=None):
```

```
        """
        Method that displays the Fast Fourier Transform of the
        different segments of specified width of the attenuated
        transmitted power our Transmitter previously
        computed on a specified band of frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency band
                      displayed, belong to reached bands.
                width: int, number of points displayed on each segment represented.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        for i in range(len(self.attenuated_power[band]) // width):
            if any(self.attenuated_power[band][i * width:(i + 1) * width] !=
                   np.zeros(width)):
                fft = np.fft.fft(self.attenuated_power[band][i * width:
                                                            (i + 1) * width])
                frequency_steps = np.fft.fftfreq(width,
                                 width / SAMPLING_FREQUENCY)
                plt.figure()
                plt.plot(frequency_steps, fft)
                plt.title("Fast Fourier Transform of segment " +
                          str(i) + " of attenuated transmitted power over band "
                          + str(round(band * 1e-9, 2)) + 'GHz - ' +
                          str(round(band * 1e-9, 2) + 1.5) + 'GHz')
                plt.show()

    def display_stft_attenuated_power(self, width=32000, band=None):
        """
        Method that displays the Short Time Fourier Transform of
        the different segments of specified width of the
        attenuated transmitted power our Transmitter previously
         computed on a specified band of frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency
                    band displayed, belong to reached bands.
                width: int, number of points displayed on each
                    segment represented.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        for i in range(len(self.attenuated_power[band]) // width):
            if any(self.attenuated_power[band][i * width:
                                (i + 1) * width] != np.zeros(width)):
```

```
            frequency_steps, time_steps, stft =
                        sig.stft(self.attenuated_power[band]
                                                [i * width:(i + 1) * width],
                                 SAMPLING_FREQUENCY)
            plt.figure()
            plt.pcolormesh(time_steps, frequency_steps, np.abs(stft))
            plt.title("Short Time Fourier Transform of segment " + str(i) +
                      " of attenuated transmitted power over band " +
                      str(round(band * 1e-9, 2)) + 'GHz - ' +
                      str(round(band * 1e-9, 2) + 1.5) + 'GHz')
            plt.show()

    def display_complete_attenuated_power(self, band=None):
        """
        Method that displays the attenuated transmitted power our
        Transmitter previously computed on a specified band
        of frequencies.
            Parameters:
                self: Transmitter
                band: float, the smallest  frequency of frequency
                     band displayed, belong to reached bands.
        """
        if band is None:
            band = random.choice(self.reached_bands)
        plt.figure()
        plt.plot(self.attenuated_power[band])
        plt.title("Attenuated transmitted power over band " +
                  str(round(band * 1e-9, 2)) + 'GHz - ' +
                  str(round(band * 1e-9, 2) + 1.5) + 'GHz')
        plt.show()

    def compute_acquisition_trans(self, band, t_0):
        """
        Method that returns the attenuated power transmitted by
        the environment toward the associated receptor over the
        specified frequency band, as a numpy.ndarray, over an
        acquisition step of duration LISTENING_TIME beginning at
        t_0.
        Parameters:
            self: Transmitter
            band: float, the smallest frequency of the specified
            frequency band, belongs to FREQUENCY_BAND.
            t_0: float, difference between beginning time of
            acquisition and beginning time of the first pattern
                considered.
        Returns:
```

```
        sig_: numpy.ndarray, contains the attenuated power
            transmitted by the environment toward the associated
            receptor over the specified frequency band.
        emission_times: numpy.ndarray, times of emissions of the
            different pulses transmitted by the transmitter
            over our acquisition.
        frequencies: numpy.ndarray, frequencies of the different
            pulses transmitted by the transmitter over our
            acquisition.
    """
    t_end = LISTENING_TIME + t_0
    time_steps = np.arange(0, self.pattern_duration, 1. / SAMPLING_FREQUENCY)
    i = bisect(time_steps, t_0)
    i_em = bisect(self.emission_times[band], t_0)

    # Case where our acquisition is included in our pattern
    if t_end < self.pattern_duration:
        j = bisect(time_steps, t_end)
        j_em = bisect(self.emission_times[band], t_end) - 1
        return self.attenuated_power[band][i:j],
               [np.array(self.emission_times[band][i_em:j_em]) - t_0], \
               [np.array(self.frequencies[band][i_em:j_em])]

    # When our acquisition ends after the end of our first pattern
    else:
        # Initialise vector sig_
        sig_ = np.zeros(int(LISTENING_TIME * SAMPLING_FREQUENCY))

        i = len(self.attenuated_power[band]) - i

        # The i first points of our acquisition are the i last points
        # of our pattern
        sig_[:i] = self.attenuated_power[band][-i:]

        j = i

        # Add the characteristics of the i_em last pulses of the
        # pattern to our emission_times and frequencies
        # lists.
        emission_times = [self.emission_times[band][i_em:] - t_0]
        frequencies = [self.frequencies[band][i_em:]]

        # When the next pattern is saved until the end by the acquisition
        while j + len(self.attenuated_power[band]) < len(sig_):
            j += len(self.attenuated_power[band])
```

```
            # Add the pattern to our acquisition
            sig_[i:j] = self.attenuated_power[band]

            # Add the characteristics of each pulse of the
            # pattern to our emission_times and frequencies lists.
            emission_times.append(np.array(self.emission_times[band][1:]) +
             emission_times[-1][-1])
            frequencies.append(np.array(self.frequencies[band]))

            i = j

        # When only a part of the next pattern is taken into account
        j = len(sig_) - j

        # Add the j first points of our pattern are the j
        # last points of our acquisition
        sig_[-j:] = self.attenuated_power[band][:j]

        j_em = bisect(np.array(self.emission_times[band]), time_steps[j])

        # Add the characteristics of the j_em first pulses of the
        # pattern to our emission_times and frequencies
        # lists.
        emission_times.append(np.array(self.emission_times[band][1:j_em])
         + emission_times[-1][-1])
        frequencies.append(np.array(self.frequencies[band][:j_em]))

        return sig_, emission_times, frequencies

def compute_acquisition_trans_jitter(self, band, t_0):
    """
    Method that returns the attenuated power transmitted by
    the environment toward the associated receptor over the
    specified frequency band, as a numpy.ndarray, over an
    acquisition step of duration LISTENING_TIME beginning at
    t_0 for a transmitter with priLawType Jitter.
    Parameters:
        self: Transmitter
        band: float, the smallest frequency of the specified
            frequency band, belongs to FREQUENCY_BAND.
        t_0: float, difference between beginning time of acquisition
            and beginning time of the first pattern
            considered.
    Returns:
        sig_: numpy.ndarray, contains the attenuated power
            transmitted by the environment toward the associated
```

```
            receptor over the specified frequency band.
        emission_times: numpy.ndarray, times of emissions of
            the different pulses transmitted by the transmitter
            over our acquisition.
        frequencies: numpy.ndarray, frequencies of the different
            pulses transmitted by the transmitter over our
            acquisition.
    """
    power_pattern = self.attenuated_power[band]
    t_end = LISTENING_TIME + t_0
    time_steps = np.arange(0, self.pattern_duration, 1. / SAMPLING_FREQUENCY)
    i = bisect(time_steps, t_0)
    i_em = bisect(self.emission_times, t_0)

    # Case where our acquisition is included in our pattern
    if t_end < self.pattern_duration:
        j = bisect(time_steps, t_end) - 1
        j_em = bisect(self.emission_times, t_end) - 1
        return power_pattern[i:j],
                  [self.emission_times[band][i_em:j_em] - t_0],
                  [np.array(self.frequencies[band][i_em:j_em])]

    # When our acquisition ends after the end of our first pattern
    else:
        # Initialise vector sig_
        sig_ = np.zeros(int(LISTENING_TIME * SAMPLING_FREQUENCY))

        i = len(power_pattern) - i

        # The i first points of our acquisition are
        # the i last points of our pattern
        sig_[:i] = power_pattern[-i:]

        j = i
        l_em = len(self.emission_times[band])

        # Add the characteristics of the i_em last pulses of the
        # pattern to our emission_times and frequencies
        # lists.
        emission_times = [self.emission_times[band][l_em - i_em:] - t_0]
        frequencies = [self.frequencies[band][i_em:]]

        # Allow to take Jitters into account by computing a
        # new pattern. The pulses pattern taken into account
        # remain the same but the pulse repetition intervals
        # are modified, chosen randomly within certain values,
```

```
# to avoid repetition of the pulse repetition interval
# pattern as would be done for other priLaws.
# However, this code does not allow to capt the
# acquisition on different frequency bands with the same
# emission times, since emission times are here
# chosen randomly for each band
power_pattern = self.new_jitter_pattern()[band]

# When the next pattern is saved until the end by the acquisition
while j + len(power_pattern) < len(sig_):
    j += len(power_pattern)

    # Add the pattern to our acquisition
    sig_[i:j] = power_pattern

    # Add the characteristics of each pulse of the
    # pattern to our emission_times and frequencies lists.
    emission_times.append(self.emission_times[band][1:]
                          + emission_times[-1][-1])
    frequencies.append(np.array(self.frequencies[band]))

    i = j
    power_pattern = self.new_jitter_pattern()[band]

# When only a part of the next pattern is taken into account
j = len(sig_) - j

# Add the j first points of our pattern are
# the j last points of our acquisition
sig_[-j:] = power_pattern[:j]
j_em = bisect(self.emission_times, time_steps[j])

# Add the characteristics of the j_em first pulses of
# the pattern to our emission_times and frequencies
# lists.
emission_times.append(self.emission_times[band][1:j_em] +
                              emission_times[-1][-1] - t_0)
frequencies.append(np.array(self.emission_times[band][:j_em]))
return sig_, emission_times, frequencies
```

A.3 Classe Environment

```
# We define here our class Environment. It allows to
# generate several transmitters with their distances to the receptor
# and generate the added signals in each frequency band.

import numpy as np
import matplotlib.pyplot as plt
import random

from raw_simulator.config_settings import get_settings, get_submodes
from raw_simulator.Transmitter import Transmitter

global_params = get_settings("GLOBAL")
FREQUENCY_BANDS = global_params['FREQUENCY_BANDS']

transmitter_params = get_settings("ENVIRONMENT")
MIN_TRANSMITTERS = int(transmitter_params['MIN_TRANSMITTERS'])
MAX_TRANSMITTERS = int(transmitter_params['MAX_TRANSMITTERS'])


class Environment:
    def __init__(self, path_submodes_add,
                 n_transmitters=None,
                 transmitters_characteristics=None,
                 transmitters_distances=None):
        """
        Method that instantiate class Environment.
            Parameters:
                path_submodes_add: str, path to submodes table.
                n_transmitters: int, number of transmitters to create
                transmitters_characteristics: list, contains the
                    possible characteristics of the transmitters we
                    will create.
                transmitters_distances: list, contains the distances
                    between the receptor the environment is
                    associated to and each of the transmitters.
            Returns:
                self: Environment.
        """
        # Import submode dataFrame
        self.submodes = get_submodes(path_submodes_add)

        # Define the number of transmitters within the environment
        if n_transmitters is None and transmitters_characteristics:
            self.n_transmitters = len(transmitters_characteristics)
```

```
elif n_transmitters is None:
    self.n_transmitters = random.randrange(MIN_TRANSMITTERS,
                            MAX_TRANSMITTERS + 1)
else:
    self.n_transmitters = n_transmitters

# Define the list of transmitters within the environment
self.transmitters = self.create_transmitters(path_submodes_add,
                                             transmitters_characteristics,
                                             transmitters_distances)

# Compute the frequency bands of interest depending
# on the emission frequency of the transmitters
self.reached_bands = self.compute_bands()

# Compute the dictionaries received_power.
# For each transmitter within self.transmitters,
# received_powers contains a dictionary, and,
# for each band within self.reached_bands,
# this dictionary contains the power of the
# attenuated signal received
# by the environment receptor from this particular
# transmitter of frequency within the band.
# The dictionaries contain numpy.ndarray referenced
# by the smallest frequency of each band.
# self.received_power contains dictionaries
# referenced by the transmitter.
self.received_power = {}
for trans in self.transmitters:
    self.received_power[trans] = {}
    for band in trans.reached_bands:
        self.received_power[trans][band] = trans.attenuated_power[band]

# Compute dictionaries acquisition and emission_characteristics
# For each frequency band within self.reach_bands, acquisition
# contains the power transmitted by the environment
# toward the associated receptor over the band, as a numpy.ndarray.

# For each frequency band within self.reached_bands,
# emission_characteristics contains the characteristics of
# the pulses computed over the signal received from
#  the environment of frequency within the band.
# The dictionary contains for each band a structured
# array which, for each pulse contains the following
# information: 'Time_Start', 'Time_End', 'Frequency',
# 'Pulse Width', 'Distance', 'Transmitter_ID'
```

```
    self.acquisition, self.emission_characteristics = self.compute_acquisition()

def create_transmitters(self, path_submodes_add,
 transmitters_characteristics=None, transmitters_distances=None):
    """
    Method that returns the list of transmitters depending on the
    specified characteristics.
        Parameters:
            self: Environment.
            path_submodes_add: str, path to submodes table.
            transmitters_characteristics: list, contains the possible
                characteristics of the transmitters we
                will create.
            transmitters_distances: list, contains the distances
                between the receptor the environment is
                associated to and each of the transmitters.
        Returns:
            transmitters: list, transmitters created depending
               on the specified characteristics.
    """
    transmitters = []

    # Initialize transmitters' characteristics
    if transmitters_characteristics is None:
        transmitters_characteristics =
                random.choices(self.submodes.index,
                                    k=self.n_transmitters)
    elif len(transmitters_characteristics) != self.n_transmitters:
        transmitters_characteristics =
                 random.sample(transmitters_characteristics,
                                    k=self.n_transmitters)

    # Create list of transmitters according to the
    # characteristics previously defined
    if transmitters_distances is None:
        for trans_char in transmitters_characteristics:
            if isinstance(trans_char, list):
                transmitters.append(
                    Transmitter(path_submodes_add,
                                pw_pattern=trans_char[0],
                                freq_pattern=trans_char[1],
                                pri_pattern=trans_char[2]))
            else:
                transmitters.append(
                    Transmitter(path_submodes_add, submode_number=trans_char))
    else:
```

```
        for i in range(np.lcm(len(transmitters_characteristics),
                len(transmitters_distances))):
            trans_char = transmitters_characteristics[i]
            if isinstance(trans_char, list):
                transmitters.append(
                    Transmitter(path_submodes_add,
                                pw_pattern=trans_char[0],
                                freq_pattern=trans_char[1],
                                pri_pattern=trans_char[2],
                                dist=transmitters_distances[i]))
            else:
                transmitters.append(
                    Transmitter(path_submodes_add,
                                submode_number=trans_char,
                                dist=transmitters_distances[i]))
    return transmitters

def compute_bands(self):
    """
    Method that computes the frequency bands of interest
    depending on the emission frequency of the transmitters.
        Parameters:
            self: Environment
    """
    reached_bands = []
    for band in FREQUENCY_BANDS:
        if any([band in trans.reached_bands
                        for trans in self.transmitters]):
            reached_bands.append(band)
    return reached_bands

def display(self, band=None, trans=None):
    """
    Method that displays the received power for a
    specified transmitter over a specified band.
        Parameters:
            self: Environment
            band: float, the smallest frequency of the band
                   considered, belongs to FREQUENCY_BANDS.
            trans: Transmitter, belongs to self.transmitters.
    """
    if trans is None:
        trans = random.choice(self.transmitters)
    if band is None:
        band = random.choice(trans.reached_bands)
    plt.figure()
```

```
        plt.plot(self.received_power[trans][band])
        plt.title("Power received from transmitter " +
                  str(self.transmitters.index(trans)) + " over band " +
                  str(round(band * 1e-9, 2)) + 'GHz - ' +
                  str(round(band * 1e-9, 2) + 1.5) + "GHz")
        plt.show()

    def compute_acquisition_trans(self, trans=None, band=None, t_0=None):
        """
        Method that returns the power received from a
        specified transmitter over a specified frequency band.
            Parameters:
                self: Environment
                trans: Transmitter, source of the transmitted power computed
                band: float, the smallest frequency of
                    the frequency band in listening, belongs to FREQUENCY_BANDS.
                t_0: float, difference between the beginning
                    time of the first pattern detected and the beginning time
                    of the listening.
            Returns:
                sig: numpy.ndarray, power received from a
                      specified transmitter over a specified frequency band
                emission_characteristics: list, contains the
                     characteristics of every pulse computed by the transmitter
                    over the signal sig, as the following list :
                    ['Time_Start', 'Time_End', 'Frequency',
                     'Pulse Width', 'Distance', 'Transmitter_ID']

        """
        # Define the necessary parameters
        if trans is None:
            trans = random.choice(self.transmitters)
        if band is None:
            band = random.choice(trans.reached_bands)
        if t_0 is None:
            t_0 = random.random() * trans.pattern_duration

        # Compute the power received from specified transmitter,
        # and two lists of each pulse's emission time and
        # frequencies
        if isinstance(trans.pri_pattern, dict):
            sig, emission_times, frequencies =
                        trans.compute_acquisition_trans_jitter(band, t_0)
        else:
            sig, emission_times, frequencies =
                        trans.compute_acquisition_trans(band, t_0)
```

```
        # Compute emission_characteristics
        emission_characteristics = []
        for j in range(len(frequencies[0])):
            emission_characteristics.append((emission_times[0][j],
                                             emission_times[0][j] + trans.pw,
                                             frequencies[0][j], trans.pw,
                                             trans.distance,
                                             self.transmitters.index(trans)))
        if len(frequencies) >= 2:
            for i in range(1, len(emission_times)):
                emission_characteristics.append((emission_times[i - 1][-1],
                                                 emission_times[i - 1][-1] + trans.pw,
                                                 frequencies[i][0], trans.pw,
                                                 trans.distance,
                                                 self.transmitters.index(trans)))
                for j in range(len(frequencies[i]) - 1):
                    emission_characteristics.append((emission_times[i][j],
                                                 emission_times[i][j] + trans.pw,
                                                 frequencies[i][j + 1], trans.pw,
                                                 trans.distance,
                                                 self.transmitters.index(trans)))
        return sig, emission_characteristics

    def compute_acquisition(self, t_0=None):
        """
        Method that returns the power transmitted by the environment
        to the associated receptor over each of the reached
        frequency bands.
            Parameters:
                self: Environment
                t_0: dict, contains for each transmitter the difference
                    between the beginning time of the first pattern
                    detected and the beginning time of the listening.
            Returns:
                acquisition: dict, contains for each of the reached
                    frequency bands the sum of the power received from
                    each transmitter over the specified frequency band
                emission_characteristics: structured numpy.ndarray,
                    contains the characteristics of every pulse
                    computed by any of the transmitters over the signal
                    acquisition, with columns containing the
                    following data:
                    'Time_Start', 'Time_End', 'Frequency',
                    'Pulse Width', 'Distance', 'Transmitter_ID'
        """
```

```
# Initialise working dictionaries
acquisition = {}
emission_characteristics_list = {}  # Dictionary containing
# for each frequency band a list of the emission
# characteristics of each pulse from each transmitter

for trans in self.transmitters:
    for band in trans.reached_bands:
        # Compute signal transmitted by specified
        # transmitter over specified frequency band
        if t_0:
            aux_sig, emission_char = self.compute_acquisition_trans(trans,
             band, t_0[trans])
        else:
            aux_sig, emission_char = self.compute_acquisition_trans(trans,
                                     band)

        # Add transmitted signal to dictionary acquisition
        if band in acquisition:
            acquisition[band] += aux_sig
        else:
            acquisition[band] = aux_sig
            emission_characteristics_list[band] = []  # Initialise list of
                  # emission characteristics

        # Add each pulse's characteristics to the specified
        # band's associated list in dictionary
        # emission_characteristics list
        for emission_c in emission_char:
            emission_characteristics_list[band].append(emission_c)

data_type = [('Time_Start', float), ('Time_End', float),
             ('Frequency', float), ('Pulse Width', float),
             ('Distance', float), ('Transmitter_ID', int)]

# Convert the dictionary of lists
# emission_characteristics_list to a dictionary of
# sorted structured
# numpy.ndarray emission_characteristics
emission_characteristics = {}
for band in self.reached_bands:
    emission_characteristics[band] =
                    np.array(emission_characteristics_list[band],
                    dtype=data_type)
    np.sort(emission_characteristics[band], order=['Time_Start'])
```

```
    return acquisition, emission_characteristics

def display_acquisition(self):
    """
    Method that displays the received power array for
    each frequency band reached.
        Parameters:
            self: Environment
    """
    for band in self.reached_bands:
        plt.figure()
        plt.plot(self.acquisition[band])
        plt.title("Power received over band " +
                   str(round(band * 1e-9, 2)) + 'GHz - ' +
                  str(round(band * 1e-9, 2) + 1.5) + "GHz")
        plt.show()
```

A.4 Classe `Receptor`

```
# We define here our class Receptor. Acquire the frequency
#  band generated from Environment and digitize each band.
# The digitization is done with an 16bits-ADC characterized by its impedance.
# All the digitized bands can be written in a CSV file.

from bisect import bisect
import csv
import datetime
import os.path

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.signal as sig

from raw_simulator.config_settings import get_settings, get_submodes
from raw_simulator.Environment import Environment

global_params = get_settings("GLOBAL")
SAMPLING_FREQUENCY = float(global_params['SAMPLING_FREQUENCY'])

# numpy.array containing the smallest frequency for
# each of the considered frequency bands, in Hz.
FREQUENCY_BANDS = global_params['FREQUENCY_BANDS']

# Frequency bands characteristics in Hz.
MIN_FREQUENCY_BANDS = float(global_params['MIN_FREQUENCY_BANDS'])
MAX_FREQUENCY_BANDS = float(global_params['MAX_FREQUENCY_BANDS'])
FREQUENCY_BANDS_WIDTH = float(global_params['FREQUENCY_BANDS_WIDTH'])
OVERLAP = float(global_params['OVERLAP'])

# Minimal and maximal distance between our receptor and the
# transmitters within our environment, in meters.
MINIMAL_DISTANCE = float(global_params['MINIMAL_DISTANCE'])
MAXIMAL_DISTANCE = float(global_params['MAXIMAL_DISTANCE'])

# Duration of ou reception, in seconds.
LISTENING_TIME = float(global_params['LISTENING_TIME'])

receptor_params = get_settings("RECEPTOR")

# Impedance of the ADC
IMPEDANCE = float(receptor_params['IMPEDANCE'])
```

```
# Reception channel noise density in dBm/kHz
RECEPTION_CHANNEL_NOISE_DENSITY = float(receptor_params
['RECEPTION_CHANNEL_NOISE_DENSITY'])

# Quantification noise power in dBfs
QUANTIFICATION_NOISE_POWER = float(receptor_params['QUANTIFICATION_NOISE_POWER'])

# Frequency modulation settings data
MIN_FREQUENCY_FMOP = float(receptor_params['MIN_FREQUENCY_FMOP'])
MAX_FREQUENCY_FMOP = float(receptor_params['MAX_FREQUENCY_FMOP'])


class Receptor:
    def __init__(self, path_submodes_add, data_writing_path,
                 n_transmitters=None, transmitters_characteristics=None,
                 transmitters_distances=None, id_=None):
        """
        Method that instantiate class Receptor.
            Parameters:
                path_submodes_add: str, path to submodes table.
                data_writing_path: str, path to the directory
                 we want to save our simulated signals in.
                n_transmitters: int, number of transmitters to
                 create in our environment. If None,
number is randomly
                 chosen between MIN_TRANSMITTERS
and MAX_TRANSMITTERS
                transmitters_characteristics: list, contains the
                    possible characteristics of the transmitters we
                    will create within our environment.
                transmitters_distances: list, contains the distances
                    between the receptor and each of the
                    transmitters.
            Returns:
                self: Receptor.
        """
        # Import submodes dataFrame
        self.path_submodes_add = path_submodes_add

        self.data_writing_path = data_writing_path

        # Create the simulated environment our receptor evolves in
        self.environment = Environment(path_submodes_add,
                                       n_transmitters,
                                       transmitters_characteristics,
                                       transmitters_distances)
```

```
# Compute the frequency bands of interest depending
# on the emission frequency of the transmitters within our
# environment.
self.reached_bands = self.environment.reached_bands

# Compute the dictionaries received_powers and
# emission_characteristics.
# For each band within self.reached_bands,
# received_powers contains the power of the attenuated
# signal received
# from the environment of frequency within the band.
# The dictionary contains numpy.ndarray referenced by the
# smallest frequency of each band.
# For each band within self.reached_bands,
# emission_characteristics contains the characteristics
# of the pulses
# computed over the signal received from the environment
# of frequency within the band. The dictionary contains
# for each band a structured array which, for each pulse
#contains the following information :
# 'Time_Start', 'Time_End', 'Frequency',
# 'Pulse Width', 'Distance', 'Transmitter_ID'
self.received_powers = {}
self.emission_characteristics = {}
for band in self.reached_bands:
    self.received_powers[band] = np.copy(
     self.environment.acquisition[band])
    self.emission_characteristics[band] = np.copy(
     self.environment.emission_characteristics[band])

# Compute the addition of the reception channel and
# quantification noise as well as the analog-to-digital
# conversion. self.quantified_signals is a dictionary
# containing numpy.ndarray of int16 bits referenced by the
# smallest frequency of each frequency band within self.reached_bands.
self.quantified_signals = self.compute_quantified_signals()

# self.results is a dictionary containing numpy.ndarray
# of int16 bits referenced by the string description of
# each frequency band within self.reached_bands.
self.results = self.compute_results()

# If the id_ is specified, save the results in self.data_writing_path
if id_ is not None:
    self.dump_results(id_)
```

```
def reception_channel_noise(self):
    """
    Method that returns the transmission chain's gaussian
    noise for each band, to be added to the received
    signal.
        Parameters:
            self: Receptor
        Returns:
            noise: dict, containing for each of the reached
                frequency bands the numpy.ndarray of the noise
    """
    noise_power = 10 ** ((RECEPTION_CHANNEL_NOISE_DENSITY - 30) / 10) *
           FREQUENCY_BANDS_WIDTH
    standard_deviation = np.sqrt(noise_power / SAMPLING_FREQUENCY)
    noise = {}
    for band in self.reached_bands:
        noise[band] = np.random.normal(0, standard_deviation,
                            np.shape(self.received_powers[band]))
    return noise

def quantification_noise(self):
    """
    Method that returns the quantification noise considered
    gaussian for each band, to be added to the received
    signal.
        Parameters:
            self: Receptor
        Returns:
            noise: dict, containing for each of the reached
                  frequency bands the numpy.ndarray of the noise
    """
    noise_power = 10 ** (QUANTIFICATION_NOISE_POWER / 10)
    standard_deviation = np.sqrt(noise_power / SAMPLING_FREQUENCY)
    noise = {}
    for band in self.reached_bands:
        noise[band] = np.random.normal(0, standard_deviation,
                        np.shape(self.received_powers[band])) * (2 ** 15)
    return noise

def max_erp_by_f_2(self):
    """
    Method that returns the maximum value of the ERP/(f**2) ratio.
        Parameters:
            self: Receptor
        Returns:
```

```
            float, maximum value of the ERP/(f**2) ratio.
        """
        submodes = get_submodes(self.path_submodes_add)
        erp_by_f_2 = []
        for i in range(np.max(submodes.index) + 1):
            erp_by_f_2.append([])
            if i in submodes.index:
                for f in submodes.freq_pattern[i]:
                    erp_by_f_2[i].append(submodes.erp[i] / ((f * 1e6) ** 2))
        erp_by_f_2_max = []
        for i in range(np.max(submodes.index) + 1):
            erp_by_f_2_max.append(0)
            if i in submodes.index:
                erp_by_f_2_max[i] = np.max(erp_by_f_2[i])
        i = np.argmax(erp_by_f_2_max)
        j = np.argmax(erp_by_f_2[i])
        j_ = bisect(FREQUENCY_BANDS, (submodes.freq_pattern[i][j] * 1e6) - 1.5e9)
        return submodes.erp[i] / ((FREQUENCY_BANDS[j_] +
                       (FREQUENCY_BANDS_WIDTH / 2)) ** 2)

    def compute_quantified_signals(self):
        """
        Method that computes the addition of the reception
        channel and quantification noise as well as the
        analog-to-digital conversion.
            Parameters:
                self: Receptor
            Returns:
                signals: dict, containing numpy.ndarray of int16 bits referenced by tl
                    smallest frequency of each frequency band within self.reached_banc
        """
        # Compute maximum power value
        p_max = 1.65 * self.max_erp_by_f_2() *
                     ((3e8 / (4 * np.pi * MINIMAL_DISTANCE)) ** 2)

        # Compute maximum voltage value
        maximal_voltage = np.sqrt(p_max * IMPEDANCE)

        # Compute the reception channel's noise
        rc_noise = self.reception_channel_noise()

        # Compute the analog-to-digital conversion noise
        atd_noise = self.quantification_noise()

        signals = {}
        for band in self.reached_bands:
```

```
        # Compute received signal from received power
        #  and add the reception channel's noise
        signals[band] = np.sqrt(self.received_powers[band]) +
                        rc_noise[band]

        # Multiplication of the signal to change
        # to the digital domain of values
        signals[band] = signals[band] * np.sqrt(IMPEDANCE)
                        * (2 ** 15) / maximal_voltage

        # Add the analog-to-digital noise to processed signal
        signals[band] += atd_noise[band]

        # Conversion to int16
        signals[band] = np.int16(signals[band])
    return signals

@staticmethod
def display_quantified_signals(quantified_signals, band):
    """
    Method that displays the quantified signals
    on the specified frequency band.
        Parameters:
            quantified_signals: dict
            band: float, in FREQUENCY_BANDS
    """
    plt.figure()
    plt.plot(quantified_signals[band])
    plt.title("Quantified signal over band " +
              str(round(band * 1e-9, 2)) + 'GHz - ' +
              str(round(band * 1e-9, 2) + 1.5) + "GHz")
    plt.show()

def compute_results(self):
    """
    Method that returns the results of the signal processing,
    hence the result of the analog-to-digital
    conversion of the received signal on each frequency band,
    previously computed as self.quantified_signals,
    only here indexed by the bands names.
        Parameters:
            self: Receptor
        Returns:
            results: dict This method 'compute_results'.
    """
    results_labels = {}
```

```
        results = {}
        for band in self.reached_bands:
            results_labels[band] = str(round(band * 1e-9, 2)) +
            'GHz - ' + str(round(band * 1e-9, 2) + 1.5) + 'GHz'
            results[results_labels[band]] = self.quantified_signals[band]
        return results

    def display(self):
        """
        Method that displays the quantified signals on
        each of the reached frequency bands.
            Parameters:
                self: Receptor
        """
        for band in list(self.results):
            plt.figure()
            plt.plot(self.results[band])
            plt.title("Quantified signal over band " + band)
            plt.show()

    def display_fft(self, width=5244288):
        """
        Method that displays the Fast Fourier Transform
        of the different segments of specified width of the
        quantified signals on each of the reached frequency bands.
            Parameters:
                self: Receptor
                width: int, number of points represented on each segment.
        """
        for band in list(self.results):
            for j in range(len(self.results[band]) // width):
                fft = np.fft.fft(self.results[band][j * width:(j + 1) * width])
                frequency_steps = np.fft.fftfreq(width, width / SAMPLING_FREQUENCY)
                plt.figure()
                plt.plot(frequency_steps, fft)
                plt.title("Fast Fourier Transform of segment " +
                      str(j) + " of quantified signal over " + band)
                plt.figure()
                plt.plot(self.results[band][j * width:(j + 1) * width])
                plt.show()

    def display_spectrogram(self):
        """
        Method that displays the spectrogram of
        the different segments of specified width of the
        quantified signals on each of the reached frequency bands.
```

```
        Parameters:
            self: Receptor
    """
    for band in list(self.results):
        frequency_steps, time_steps, spectrogram =
                    sig.spectrogram(self.results[band],
                                                SAMPLING_FREQUENCY,
                                                   nperseg=32)
        plt.figure()
        plt.pcolormesh(time_steps, frequency_steps, spectrogram)
        plt.title("Spectrogram of quantified signal over " + band)
        plt.show()

def dump_results(self, id_):
    """
    Method that saves the files containing the
     transmitters' configuration data, the result quantified signal on
    each reached frequency band and the emission
    characteristics of each pulse in a new directory named
    after the date and identification string in the data writing path.
        Parameters:
            self: Receptor
            id_: str, identification string.
    """
    date = datetime.datetime.now()
    dt_string = date.strftime("%d-%m-%Y_%H-%M-%S_")
    os.mkdir(os.path.join(self.data_writing_path, dt_string + id_))
    self.write_labels(id_, dt_string)
    self.write_data(id_, dt_string)
    self.write_config(id_, dt_string)

def write_labels(self, id_, date):
    """
    Method that saves the file containing the emission
    characteristics of each pulse in directory created in
    method dump_results
        Parameters:
            self: Receptor.
            id_: str, identification string.
            date: str, date string.
    """
    dataframes = []
    for band in self.reached_bands:
        dataframe = pd.DataFrame(data=self.emission_characteristics[band],
                                 columns=['Time_Start', 'Time_End',
                                          'Frequency', 'Pulse Width',
```

```
                                            'Distance',
                                            'Transmitter_ID'])

            dataframe['band'] = str(round(band * 1e-9, 2)) +
                   'GHz - ' + str(round(band * 1e-9, 2) + 1.5) + "GHz"
            dataframes.append(dataframe)
        result = pd.concat(dataframes)
        result.to_csv(os.path.join(os.path.join(self.data_writing_path,
                           date + id_), 'labels_' + date + id_ + '.csv'))

    def write_data(self, id_, date):
        """
        Method that saves the file containing the result
        quantified signal on each reached frequency band in
        directory created in method dump_results.
            Parameters:
                self: Receptor.
                id_: str, identification string.
                date: str, date string.
        """
        with open(os.path.join(os.path.join(self.data_writing_path,
                 date + id_), 'data_' + date + id_ + '.csv'), 'w',
                  newline='') as csvfile:
            fieldnames = ['time_steps'] + list(self.results)
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

            writer.writeheader()
            time_steps = np.arange(0., LISTENING_TIME,
                      1. / SAMPLING_FREQUENCY)
            for i in range(len(time_steps)):
                line = {'time_steps': round(time_steps[i],
                      int(np.log10(SAMPLING_FREQUENCY) + 3))}
                for band in list(self.results):
                    line[band] = self.results[band][i]
                writer.writerow(line)

    def write_config(self, id_, date):
        """
        Method that saves the file containing the transmitters'
        configuration data in directory created in method
        dump_results.
            Parameters:
                self: Receptor.
                id_: str, identification string.
                date: str, date string.
        """
```

```
with open(os.path.join(os.path.join(self.data_writing_path,
        date + id_), 'config_' + date + id_ + '.txt'),
          'w') as f:
    f.write('### Settings ###\n')
    f.write('\nSAMPLING_FREQUENCY = ' + str(SAMPLING_FREQUENCY))
    f.write('\nFREQUENCY_BANDS = ' + str(FREQUENCY_BANDS))
    f.write('\nFREQUENCY_BANDS_WIDTH = ' + str(FREQUENCY_BANDS_WIDTH))
    f.write('\nOVERLAP = ' + str(OVERLAP))
    f.write('\nMIN_FREQUENCY_BANDS = ' + str(MIN_FREQUENCY_BANDS))
    f.write('\nMAX_FREQUENCY_BANDS = ' + str(MAX_FREQUENCY_BANDS))
    f.write('\nLISTENING_TIME = ' + str(LISTENING_TIME))
    f.write('\nMINIMAL_DISTANCE = ' + str(MINIMAL_DISTANCE))
    f.write('\nMAXIMAL_DISTANCE = ' + str(MAXIMAL_DISTANCE))
    f.write('\nIMPEDANCE = ' + str(IMPEDANCE))
    f.write('\nMIN_FREQUENCY_FMOP = ' + str(MIN_FREQUENCY_FMOP))
    f.write('\nMAX_FREQUENCY_FMOP = ' + str(MAX_FREQUENCY_FMOP))
    f.write('\n\n\n### Transmitters ###\n')
    f.write('\nTransmitter_id, Submode, Distance, MOP')
    i = 0
    for trans in self.environment.transmitters:
        if type(trans.modulation_characteristics) == tuple:
            mop = str(('PMOP', trans.modulation_characteristics))
        elif trans.modulation_frequency != 0:
            mop = str(('FMOP', trans.modulation_frequency))
        else:
            mop = 'No MOP'
        f.write('\n[Transmitter ' + str(i) + ', ' +
                str(trans.submode_number) + ', ' +
                str(trans.distance) +
                ', ' + mop + ']')
        i += 1
```

A.5 Notation

RAPPORT D'EVALUATION
ASSESSMENT REPORT

Merci de retourner ce rapport par courrier ou par voie électronique en fin du stage à :
At the end of the internship, please return this report via mail or email to:

ENSTA Bretagne – Bureau des stages - 2 rue François Verny - 29806 BREST cedex 9 – FRANCE
☎ 00.33 (0) 2.98.34.87.70 / stages@ensta-bretagne.fr

I - ORGANISME / *HOST ORGANISATION*

NOM / *Name* ATOS

Adresse / *Address* 655 Avenue Galilée
13799 Aix en Provence - Cedex 3

Tél / *Phone (including country and area code)*

Nom du superviseur / *Name of internship supervisor*
BELAFDIL Chakib

Fonction / *Function* Ingénieur d'étude

Adresse e-mail / *E-mail address* chakib.belafdil@atos.net

Nom du stagiaire accueilli / *Name of intern* POLYCARPE Madeleine

II - EVALUATION / *ASSESSMENT*

Veuillez attribuer une note, en encerclant la lettre appropriée, pour chacune des caractéristiques suivantes. Cette note devra se situer entre **A (très bien)** et **F (très faible)**
*Please attribute a mark from **A (excellent)** to **F (very weak).***

MISSION / *TASK*

- La mission de départ a-t-elle été remplie ? (A) B C D E F
  Was the initial contract carried out to your satisfaction?
- Manquait-il au stagiaire des connaissances ? ☐ oui/*yes* ☒ non/*no*
  Was the intern lacking skills?

  Si oui, lesquelles ? / *If so, which skills?*

ESPRIT D'EQUIPE / *TEAM SPIRIT*

- Le stagiaire s'est-il bien intégré dans l'organisme d'accueil (disponible, sérieux, s'est adapté au travail en groupe) / *Did the intern easily integrate the host organisation? (flexible, conscientious, adapted to team work)*

  (A) B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here*

COMPORTEMENT AU TRAVAIL / *BEHAVIOUR TOWARDS WORK*

Le comportement du stagiaire était-il conforme à vos attentes (Ponctuel, ordonné, respectueux, soucieux de participer et d'acquérir de nouvelles connaissances) ?
Did the intern live up to expectations? (Punctual, methodical, responsive to management instructions, attentive to quality, concerned with acquiring new skills)?

A B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____
_____

INITIATIVE – AUTONOMIE / *INITIATIVE – AUTONOMY*

Le stagiaire s'est –il rapidement adapté à de nouvelles situations ? A B C D E F
(Proposition de solutions aux problèmes rencontrés, autonomie dans le travail, etc.)

Did the intern adapt well to new situations? A B C D E F
(eg. suggested solutions to problems encountered, demonstrated autonomy in his/her job, etc.)

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____
_____

CULTUREL – COMMUNICATION / *CULTURAL – COMMUNICATION*

Le stagiaire était-il ouvert, d'une manière générale, à la communication ? A B C D E F
Was the intern open to listening and expressing himself /herself?

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____
_____

OPINION GLOBALE / *OVERALL ASSESSMENT*

❖ La valeur technique du stagiaire était : A B C D E F
Please evaluate the technical skills of the intern:

III - PARTENARIAT FUTUR / *FUTURE PARTNERSHIP*

❖ Etes-vous prêt à accueillir un autre stagiaire l'an prochain ?

Would you be willing to host another intern next year? ☒ oui/*yes* ☐ non/*no*

Fait à Aix-en-Provence, le 01/09/2022
In _____, on _____

Signature Entreprise _____ Signature stagiaire
Company stamp _____ *Intern's signature*

Merci pour votre coopération
We thank you very much for your cooperation

A.6 Exemples de reconnaissance d'impulsion

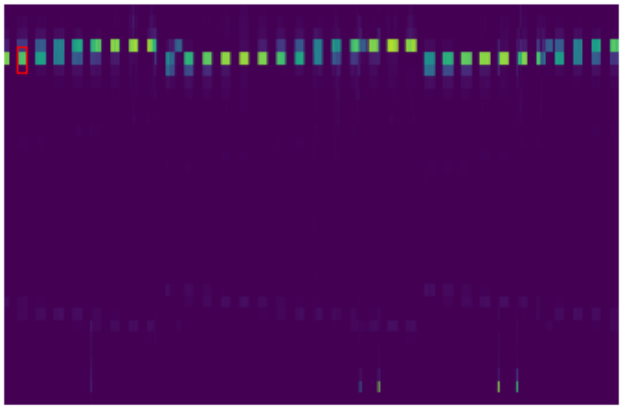

FIGURE 36 – Exemple de reconnaissance d'une impulsion par notre modèle

FIGURE 37 – Exemple de reconnaissance d'une impulsion par notre modèle

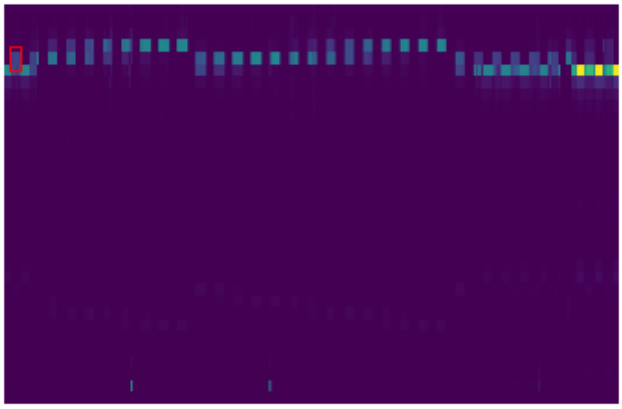

FIGURE 38 – Exemple de reconnaissance d'une impulsion par notre modèle

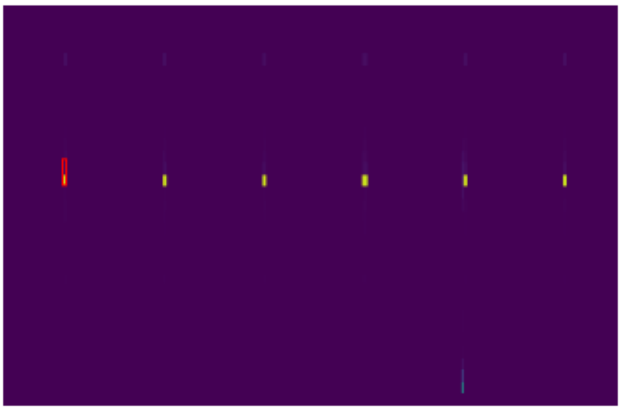

FIGURE 39 – Exemple de reconnaissance d'une impulsion par notre modèle

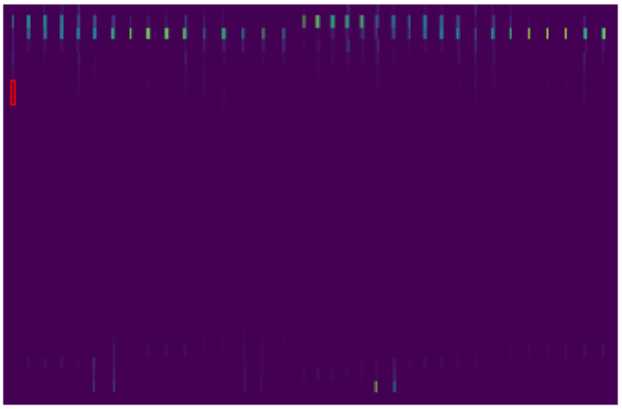

FIGURE 40 – Exemple de reconnaissance d'une impulsion par notre modèle

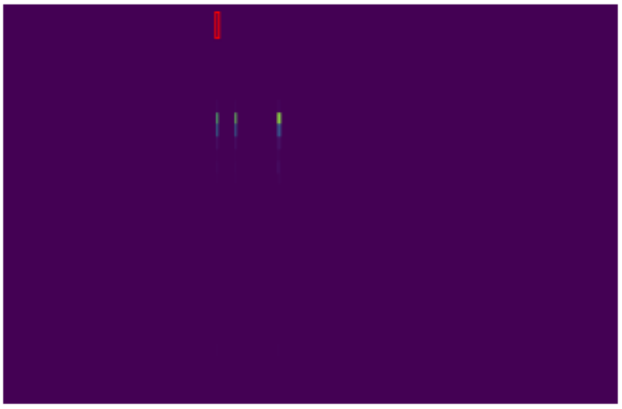

FIGURE 41 – Exemple de reconnaissance d'une impulsion par notre modèle

FIGURE 42 – Exemple de reconnaissance d'une impulsion par notre modèle

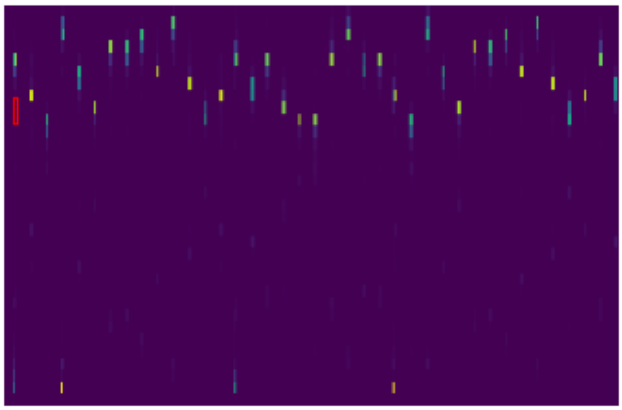

FIGURE 43 – Exemple de reconnaissance d'une impulsion par notre modèle

Table des figures

Références

[1] Adamy, D. L. *Introduction to Electronic Warfare Modeling and Simulation.* Artech House, 2006. 4

[2] Degoulange, J.-M. *Les Ecoutes de la Victoire, l'histoire secrète des services d'écoute français (1914-1918).* Pierre de Taillac, 2019. 4

[3] Dujardin, O. La guerre électronique dans les conflits d'aujourd'hui. *Institut d'Etudes Géopolitiques Appliquées* (septembre 2021). 5

[4] Genova, J. *Electronic Warfare Signal Processing.* Artech House, 2018. 12

[5] Johnson, R. C. *Antenna Engineering Handbook.* McGraw-Hill, 1984. 11

[6] Letertre, O., Justel, P., Lechâble, R., and Dossé, S. Regards croisés sur la guerre électronique. *Focus stratégique, Ifri*, n °90 (juillet 2019). 5

[7] Siffre, G. J.-P. *La Guerre Electronique : Maître des ondes, maître du monde.* Lavauzelle, 2003. 5

[8] Wiley, R. G. *Electronic Intelligence : The Interception and Analysis of Radar Signals.* Artech House, 2006. 11