RAPPORT TECHNIQUE

Création d’un système de détection passif et de localisation de drones.

CPNV MA-Métier

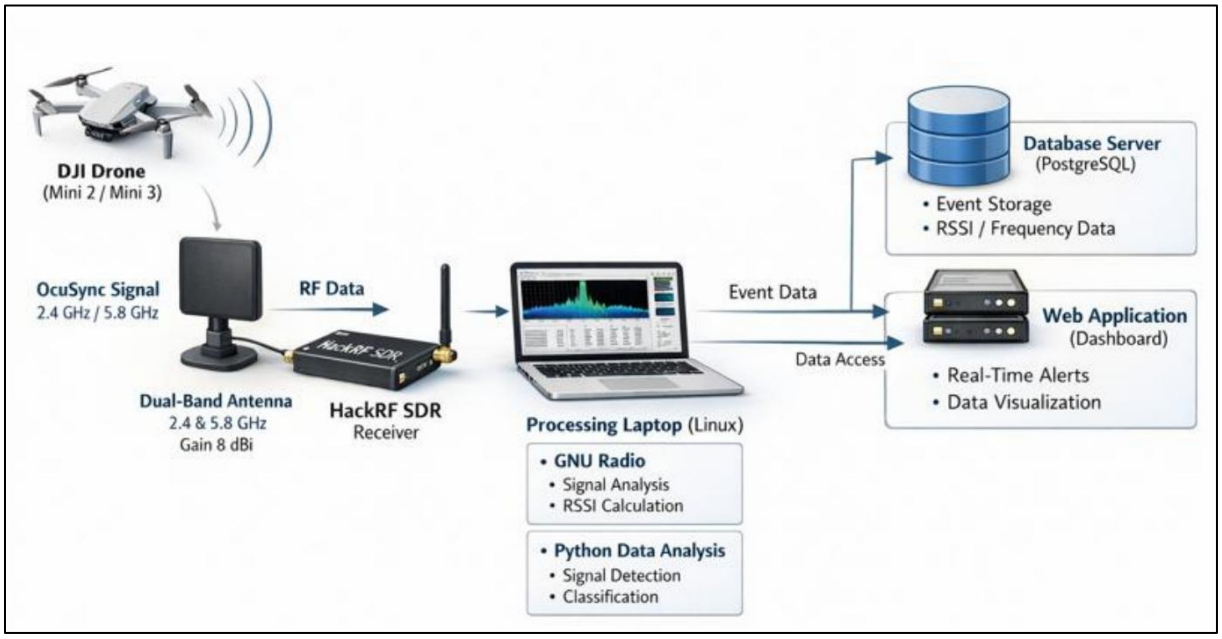
Table des matières

Architecture générale

Introduction

L'objectif de ce projet est la création d'un système de détection passif et de localisation de drones. Le système vise à surveiller l'espace aérien en captant les signaux radiofréquences (RF) émis par les drones (notamment les modèles DJI Mini 2 et 3). En utilisant une approche passive, le système détecte la présence de drones sans émettre d'ondes lui-même, en se basant sur l'analyse spectrale et la force du signal (RSSI) pour estimer leur distance relative.

Schéma global du système



Le schéma global illustre une chaîne de traitement allant de la capture physique du signal jusqu'à l'affichage utilisateur :

Source : Un drone émet des signaux (type OcuSync) sur les bandes 2.4 GHz ou 5.8 GHz.

Capture : Une antenne dual-band (gain 8 dBi) capte le signal et le transmet au récepteur HackRF SDR.

Traitement local : Un ordinateur sous Linux traite les données RF brutes via GNU Radio (analyse de signal, calcul RSSI) et des scripts Python (détection et classification).

Stockage et Interface : Les données d'événements sont envoyées vers un serveur de base de données PostgreSQL, tandis qu'une application Web affiche les alertes et la visualisation des données en temps réel.

Description des modules (RF, Python, Web, DB)

Module RF : Utilise le matériel HackRF et une antenne spécifique pour surveiller les bandes de fréquences où opèrent les drones. Son rôle est de numériser le spectre radio.

Module Python : Responsable de la logique intelligente. Il comprend des scripts pour la détection (filtrage des signaux suspects), le calcul du RSSI et l'estimation de la localisation (distance relative).

Module Web : Composé d'un backend (API) et d'un frontend (Dashboard), il permet la visualisation des données et la gestion des alertes pour l'utilisateur final.

Module DB (Base de données) : Assure le stockage persistant des logs, des positions estimées et de l'historique des événements de détection via PostgreSQL.

Flux de données (du signal RF → interface web)

Le cheminement de l'information suit les étapes suivantes :

- Le drone émet un signal RF capté par l'antenne.
- Le HackRF numérise ce signal pour le rendre exploitable par l'ordinateur.
- GNU Radio extrait les données du spectre et calcule le RSSI.
- Le script Python analyse ces données pour détecter un signal suspect (en utilisant un système de score basé sur la bande passante, la durée et la puissance).
- Python calcule ensuite la distance relative et la tendance (rapprochement ou éloignement).
- Les résultats sont enregistrés dans la base SQL.
- Le Dashboard Web récupère ces données via l'API pour les afficher en temps réel.

Partie RF / GNU Radio

Configuration du HackRF

Le HackRF One est configuré pour agir comme un récepteur passif balayant les bandes critiques.

- **Fréquences surveillées :** Le système scanne les bandes **2.4 GHz et 5.8 GHz**, fréquences standard utilisées par les protocoles de drones comme DJI OcuSync.
- **Sample rate utilisé :** Pour garantir une capture fidèle des signaux à large bande, un taux d'échantillonnage de **20 MHz** (20e6) est configuré.
- **Gain RF :** Une antenne dual-band avec un **gain de 8 dBi** est utilisée pour optimiser la réception. Le gain interne du HackRF est ajusté pour maximiser le rapport signal/bruit sans saturer le récepteur.
- **Bande passante :** La détection se focalise sur des signaux occupant généralement plus de **10 MHz**, caractéristique des protocoles modernes.
- **Tests de réception simples :** La calibration est validée par la réception de signaux **Wi-Fi et Bluetooth**, servant de références de fréquences et de puissance pour s'assurer que le matériel capte correctement l'environnement RF ambiant.

Chaîne de traitement du signal

La chaîne traite le flux depuis l'antenne jusqu'à l'analyse logique.

- **Blocs GNU Radio :**
  - **Source HackRF :** Récupération des échantillons IQ bruts.
  - **Filtre :** Isolation des bandes d'intérêt et réduction du bruit hors bande.
  - **FFT (Fast Fourier Transform) :** Conversion du signal temporel en données fréquentielles pour l'analyse spectrale.
- **Pipeline :** Le flux suit le chemin : **HackRF → Filtrage numérique → FFT → Analyse du spectre → Export des données vers Python** via un serveur TCP (souvent sur le port 5005).

Algorithmes de détection

La détection repose sur l'identification de pics stables dans le spectre et sur un seuil RSSI. Le système différencie le bruit des signaux réels en analysant le pattern spectral caractéristique des protocoles de communication de drones.

Bande passante (> 10 MHz) : Identifie les protocoles larges type OFDM (+0.3).

RSSI (> -65 dBm) : Confirme la proximité et la puissance du signal (+0.2).**Stabilité (Variance RSSI < 40) :** Différencie un signal contrôlé (drone) du bruit aléatoire (+0.3).

Durée (> 0.03s) : Vérifie la persistance de l'émission caractéristique d'une liaison active (+0.2).

Analyse spectrale

L'analyse spectrale permet de transformer les ondes brutes en informations exploitables.

- **Visualisation :** Le spectre est visualisé via des outils comme un **spectrogramme (waterfall)**, affichant la puissance du signal en fonction du temps et de la fréquence.
- **Identification des pics :** Le système identifie les pics stables au-dessus du bruit de fond. Un pic est considéré comme un signal potentiel s'il dépasse un seuil de puissance moyen calculé sur l'ensemble du scan.
- **Interprétation :** Les signaux de drones se distinguent par des formes spectrales rectangulaires et larges (OFDM). Une émission persistante sur une bande large avec une puissance stable indique la présence d'un drone en communication avec sa télécommande.

Partie Python (détection + localisation)

Scripts utilisés

- Les scripts de détections (*detector.py, features.py*) sont regroupés dans un dossier « detection ». Ils permettent notemment de trier les signaux entre eux, par exemple en enregistrant que ceux qui pourraient ressembler à ceux d'un drone. Farah a utilisé pour cela un système de score. Si le score final est supérieur ou égal à 0,7 (il augmente grâce à différents facteurs, comme la durée ou le résultat du RSSI, par exemple), le signal est enregistré.

  Librairies utilisées : Numpy

- Le dossier « rf_inputs » contient les scripts nécessaires à l'extraction des données depuis le GNURadio. Ceux-ci sont
- Le dossier « localization » contient les fichiers *estimator.py*, *models.py*, *rssi_model.py* et *tracker.py*. Ces fichiers servent à traiter les données reçues ; par exemple en utilisant un RSSI pour convertir une durée relative en distance relative, qui pourra ensuite potentiellement être convertie en distance.
- Finalement, le dossier « storage » contient les scripts utilisés pour envoyer les signaux traités vers la base de données. Ils sont *database.py*, *models.py* ainsi que le fichier *model.sql*.

Module detection

Ce module est responsable de la détection des drones à partir des signaux radio-fréquence (RF) captés par le système. Il analyse les caractéristiques des signaux pour identifier la présence probable d'un drone.

detector.py

Ce fichier contient la logique de décision pour déterminer si un signal RF correspond probablement à un drone.

Fonction is_drone(features)

Cette fonction est le cœur du système de détection. Elle prend en entrée un dictionnaire de caractéristiques extraites d'un signal RF et retourne une décision binaire (drone ou non) ainsi qu'un score de confiance.

Paramètres :

- features : dictionnaire contenant les caractéristiques du signal RF (bandwidth, rssi, rssi_variance, duration)

Retourne :

- Un tuple (is_drone, score) où is_drone est un booléen et score est un nombre entre 0 et 1

Logique de détection :

La fonction utilise un système de notation (scoring) basé sur quatre critères principaux :

- Bande passante (bandwidth > 10 MHz) : +0.3 points

Les drones modernes utilisent des protocoles de communication à large bande (Wi-Fi, OFDM) qui occupent généralement plus de 10 MHz. Si la bande passante détectée dépasse ce seuil, cela indique probablement une communication de type drone.

- Force du signal (RSSI > -65 dBm) : +0.2 points

Un RSSI (Received Signal Strength Indicator) supérieur à -65 dBm indique que la source est relativement proche et émet avec une puissance suffisante, caractéristique typique des drones en vol.

- Stabilité du signal (rssi_variance < 40) : +0.3 points

Les drones en vol maintiennent généralement une position stable ou se déplacent de manière contrôlée, ce qui se traduit par une faible variance du RSSI. Une variance élevée indiquerait plutôt un signal intermittent ou du bruit.

- Durée d'une émission (duration > 0.03 secondes) : +0.2 points

Les drones maintiennent une liaison radio continue avec leur télécommande. Une émission persistante de plus de 30 millisecondes suggère une communication active et non un simple bruit ponctuel.

Décision finale :

Si le score total atteint ou dépasse 0.7 (70%), le signal est classifié comme provenant probablement d'un drone. Ce seuil a été choisi pour équilibrer la sensibilité (détecter les vrais drones) et la spécificité (éviter les fausses alertes).

features.py

Ce fichier est responsable de l'extraction des caractéristiques techniques à partir des données brutes de scan RF.

Fonction extract_scan_features(scan_lines)

Cette fonction analyse les données d'un scan RF et en extrait des métriques quantitatives utilisables pour la détection.

Paramètres :

- scan_lines : liste de lignes de texte provenant d'un fichier de scan RF

Format des données d'entrée :

Chaque ligne du fichier de scan suit la structure suivante (format CSV) :

- Position 0 : date
- Position 1 : heure
- Position 2 : fréquence de début (f_start en Hz)
- Position 3 : fréquence de fin (f_end en Hz)
- Position 4 : bande passante
- Position 5 : nombre d’échantillons
- Positions 6+ : valeurs RSSI mesurées

Processus d'extraction :

- Étape 1 : Parsing des données

Pour chaque ligne, la fonction sépare les valeurs par virgules et extrait les fréquences de début et fin ainsi que toutes les valeurs RSSI. Les valeurs non numériques sont ignorées pour éviter les erreurs.

- Étape 2 : Calcul du RSSI moyen

Toutes les valeurs RSSI collectées sont moyennées pour obtenir un indicateur global de la puissance du signal reçu. Cela donne une mesure de la proximité et de la puissance d'une émission de la source.

- Étape 3 : Calcul de la variance du RSSI

La variance mesure la stabilité du signal dans le temps. Un signal stable (faible variance) indique une source stationnaire ou se déplaçant de manière régulière, tandis qu'une forte variance suggère du bruit ou des interférences.

- Étape 4 : Calcul de la bande passante totale

La bande passante est calculée comme la différence entre la fréquence maximale et minimale observées dans toutes les lignes du scan. Cela donne une indication du spectre utilisé par le signal.

- Étape 5 : Estimation de la durée

La durée est estimée en multipliant le nombre de lignes de scan par 0.01 seconde (hypothèse : chaque ligne représente 10 ms de mesure).

Valeur de retour :

Un dictionnaire Python contenant :

- rssi : puissance moyenne du signal en dBm (float)
- rssi_variance : variance de la puissance (float)
- bandwidth : bande passante totale en Hz (float)
- duration : durée estimée de l'émission en secondes (float)

Gestion des erreurs :

Si aucune donnée RSSI exploitable n'est trouvée dans les lignes du scan, la fonction lève une exception ValueError avec le message « Aucune donnée RSSI exploitable ». Cela permet au programme appelant de gérer ce cas d'erreur.

Module géolocalisation

Ce module gère l'estimation de la position et de la distance du drone par rapport au récepteur RF. Avec un seul capteur, la géolocalisation précise est impossible, mais le système peut estimer une distance relative et une zone approximative.

estimator.py

Ce fichier fournit les fonctions d'une estimation de position basées sur la puissance du signal reçu.

Fonction estimate_position(receiver_pos, rssi)

Cette fonction estime la distance relative entre le récepteur RF et la source du signal (drone).

Paramètres :

- receiver_pos : dictionnaire contenant la position GPS du récepteur (clés pour latitude, pour longitude)
- rssi : valeur du RSSI mesuré en dBm (float)

Processus d'estimation :

- Étape 1 : Conversion RSSI vers distance

La fonction appelle rssi_to_distance(rssi) pour convertir la puissance du signal reçu en une estimation de distance. Cette distance est RELATIVE et non absolue, car elle dépend de nombreux facteurs non calibrés (puissance d'émission du drone, obstacles, conditions atmosphériques).

- Étape 2 : Classification en zones

La distance relative est ensuite utilisée pour classer le drone dans l'une des trois zones suivantes :

  - proche : distance < 1.5 (unités relatives)
  - moyenne : 1.5 ≤ distance < 4.0
  - loin : distance ≥ 4.0

Valeur de retour :

Un dictionnaire contenant :

  - distance_relative : distance estimée en unités relatives (float)
  - zone : classification en zone (string :proche,moyenne, ou loin)
  - receiver_lat : latitude du récepteur (float)
  - receiver_lon : longitude du récepteur (float)

Limitations importantes :

Avec un seul capteur RF, il est IMPOSSIBLE de déterminer :

  - La direction du drone

- La position GPS absolue du drone
- L'altitude du drone

Pour obtenir ces informations, il faudrait soit :

- Plusieurs capteurs RF pour la triangulation
- Un système radar avec capacité de direction
- Une antenne directionnelle rotative

rssi_model.py

Ce fichier implémente le modèle de propagation radio pour convertir le RSSI en distance.

Fonction rssi_to_distance(rssi)

Cette fonction utilise un modèle de propagation en espace libre pour estimer la distance à partir du RSSI.

Paramètre :

- rssi : valeur du RSSI mesuré en dBm (float)

Modèle mathématique utilisé :

La fonction utilise le modèle de propagation logarithmique inversé :

distance = 10^((RSSI_ref - RSSI) / (10 × n))

Où :

- RSSI_ref = 40.0 dBm (RSSI de référence quand le drone est proche)
- n = 2.0 (exposant de perte de propagation, 2.0 pour un environnement libre)
- RSSI = valeur mesurée en dBm

Explication du modèle :

- Exposant de perte de propagation (n = 2.0)

Dans un environnement idéal en espace libre, la puissance du signal décroît avec le carré de la distance (loi en $1/r^2$). En environnement urbain ou avec obstacles, n peut aller de 3 à 5, mais ici on utilise 2.0 comme approximation conservative.

- RSSI de référence (40.0 dBm)

Cette valeur représente le RSSI attendu lorsque le drone est à une distance de référence proche. C'est un paramètre qui devrait être calibré expérimentalement pour chaque configuration matérielle.

Valeur de retour :

- Une distance RELATIVE en unités arbitraires (float)

AVERTISSEMENT IMPORTANT :

Ce modèle N'EST PAS CALIBRÉ en mètres. Les distances retournées sont RELATIVES et servent uniquement à comparer les variations dans le temps. Pour obtenir des distances réelles en mètres, il faudrait :

- Calibrer le système avec des mesures à distances connues
- Prendre en compte la puissance d'émission du drone
- Ajuster l'exposant selon l'environnement réel
- Compenser les variations d'une antenne et de gain

tracker.py

Ce fichier implémente le suivi temporel de la distance pour déterminer si le drone se rapproche ou s'éloigne.

Classe DistanceTracker

Cette classe analyse l'évolution de la distance dans le temps en maintenant un historique des mesures récentes.

Méthode __init__(window_size=5)

Constructeur de la classe qui initialise l'historique des distances.

- Paramètre : window_size (défaut = 5) : nombre de mesures à conserver dans l'historique
- Attribut créé : self.history : une deque (file double-ended) qui stocke les dernières distances

La deque est utilisée car elle permet d'ajouter et retirer efficacement des éléments aux extrémités, et elle limite automatiquement sa taille à maxlen.

Méthode update(distance)

Cette méthode met à jour l'historique avec une nouvelle mesure de distance et détermine la tendance du mouvement.

Paramètre :

- distance : nouvelle valeur de distance à ajouter à l'historique (float)

Logique d'analyse :

- Étape 1 : Ajout de la nouvelle mesure

La distance est ajoutée à l'historique. Si l'historique est plein, la mesure la plus ancienne est automatiquement supprimée (grâce à maxlen de la deque).

- Étape 2 : Vérification des données disponibles

Si l'historique contient moins de 2 mesures, il est impossible de déterminer une tendance. La fonction retourne inconnue.

- Étape 3 : Calcul de la variation

La variation totale (delta) est calculée comme la différence entre la dernière mesure et la première mesure de l'historique :

delta = distance_actuelle - distance_la_plus_ancienne

- Étape 4 : Classification de la tendance

Selon la valeur de delta :

  - Si delta < -0.2 : rapprochement (le drone se rapproche)
  - Si delta > 0.2 :eloignement (le drone s'éloigne)
  - Sinon : stable (le drone reste à distance approximativement constante)

Valeur de retour :

- Une chaîne de caractères :rapprochement, eloignement, stable, ou inconnu;

Seuils de détection :

Les seuils de ±0.2 ont été choisis pour filtrer les petites fluctuations dues au bruit de mesure. Ces valeurs peuvent être ajustées selon la précision souhaitée :

  - Seuil plus petit : plus sensible aux petits mouvements, mais plus de fausses détections
  - Seuil plus grand : moins sensible, détecte seulement les mouvements importants

models.py

Ce fichier définit la classe de données pour représenter une détection.

Classe Detection

Cette classe encapsule toutes les informations relatives à une détection de drone.

Attributs de la classe :

- timestamp : date et heure de la détection (généralement un objet datetime Python)

Utilisé pour tracer l'historique temporel des détections et pour la sauvegarde en base de données.

- freq : fréquence centrale du signal détecté en Hertz (float)

Indique la bande de fréquence sur laquelle le drone communique. Les drones commerciaux utilisent typiquement 2.4 GHz ou 5.8 GHz.

- rssi : puissance du signal reçu en dBm (float)

Mesure de la force du signal, utilisée pour estimer la distance. Plus le RSSI est élevé (proche de 0), plus le signal est fort.

- bandwidth : largeur de bande occupée par le signal en Hertz (float)

Indique le spectre utilisé par la communication. Les protocoles modernes (Wi-Fi, OFDM) utilisent des bandes larges (10-80 MHz).

- distance : distance estimée entre le récepteur et la source en unités relatives (float)

Distance calculée à partir du RSSI via le modèle de propagation. Cette valeur est relative et non calibrée en mètres.

Utilisation typique :

Cette classe est utilisée comme structure de données pour passer les informations de détection entre différents modules du système. Elle peut être sérialisée pour sauvegarde en base de données ou affichage dans l'interface utilisateur.

Module RF_INPUT

Ce module gère l'acquisition des signaux radiofréquence depuis différentes sources : simulation, fichiers de données, ou matériel SDR (Software-Defined Radio) réel.

adapter.py

Ce fichier est prévu pour l'interface avec un SDR matériel réel.

Fonction read_from_sdr()

Cette fonction est un emplacement (placeholder) pour la future intégration d'un SDR matériel.

État actuel :

La fonction lève une exception NotImplementedError avec le message SDR réel non connecté. Cela signifie qu'elle n'est pas encore implémentée.

Implémentation future :

Lorsqu'un SDR sera connecté, cette fonction devra :

- Configurer le SDR (fréquence centrale, gain, taux d'échantillonnage)
- Lancer l'acquisition des échantillons IQ
- Retourner un tableau NumPy d'échantillons complexes (dtype=np.complex64)

SDR compatibles :

- HackRF One : SDR populaire, bande 1 MHz - 6 GHz

mock.py

Ce fichier génère des signaux RF simulés pour le développement et les tests sans matériel SDR.

Fonction generate_iq_samples(fs=20e6, duration=0.05)

Cette fonction génère un signal IQ complexe simulé qui ressemble à un signal OFDM typique des drones.

Paramètres :

- fs : fréquence d'échantillonnage en Hz (défaut = 20 MHz)
- duration : durée du signal en secondes (défaut = 0.05 s = 50 ms)

Processus de génération :

- Étape 1 : Calcul du nombre d'échantillons

num_samples = int(fs × duration)

Exemple : 20 MHz × 0.05 s = 1 000 000 échantillons

- Étape 2 : Création du vecteur temps

t = [0, 1/fs, 2/fs, ..., (num_samples-1)/fs]

Ce vecteur représente les instants d'échantillonnage.

- Étape 3 : Génération du signal porteur

signal = exp(j × 2π × 1 MHz × t)

Cette formule génère une sinusoïde complexe (signal IQ) à 1 MHz. Le signal IQ représente à la fois l'amplitude et la phase du signal RF.

- Étape 4 : Ajout de bruit

Un bruit blanc gaussien complexe est ajouté avec un écart-type de 0.1 pour simuler les conditions réelles de réception. Le bruit est composé de deux parties :

- Composante réelle : np.random.randn(num_samples)
- Composante imaginaire : j × np.random.randn(num_samples)

Valeur de retour :

- Un tableau NumPy de dtype complex64 contenant les échantillons IQ

Note technique :

Les échantillons IQ (In-phase/Quadrature) sont la représentation standard des signaux RF en bande de base. Chaque échantillon est un nombre complexe où :

- Partie réelle (I) : composante en phase
- Partie imaginaire (Q) : composante en quadrature (déphasée de 90°)

Cette représentation permet de capturer à la fois l'amplitude et la phase du signal RF.

reader.py

Ce fichier fournit des fonctions pour lire des fichiers IQ binaires.

Fonction read_iq_file(path)

Cette fonction lit un fichier binaire contenant des échantillons IQ complexes.

Paramètre :

- path : chemin complet vers le fichier binaire IQ (string)

Format de fichier attendu :

Le fichier doit contenir des échantillons IQ au format binaire complex64 (32 bits pour la partie réelle + 32 bits pour la partie imaginaire = 64 bits par échantillon).

Processus de lecture :

- Étape 1 : Vérification de l'existence du fichier

La fonction vérifie que le fichier existe avec os.path.exists(). Si le fichier est introuvable, elle lève une exception FileNotFoundError.

- Étape 2 : Chargement des données

Le fichier est lu avec np.fromfile() qui interprète les données binaires comme un tableau de nombres complexes (complex64).

- Étape 3 : Validation

Si le fichier est vide (iq_samples.size == 0), une exception ValueError est levée avec le message Fichier IQ vide ou non valide.

Valeur de retour :

- Un tableau NumPy de dtype complex64 contenant les échantillons IQ lus

Fonction read_from_sdr()

Identique à adapter.py, cette fonction est un placeholder pour la future lecture directe depuis un SDR. Elle lève actuellement une NotImplementedError.

source.py

Ce fichier centralise la récupération des signaux RF depuis différentes sources. C'est le point d'entrée principal pour l'acquisition de données.

Fonction get_iq_samples(source, file_path, tcp_host, tcp_port)

Cette fonction récupère les échantillons IQ selon la source choisie par l'utilisateur.

Paramètres :

- source : type de source de données (simulation, txt, ou tcp;)
- file_path : chemin vers le fichier texte (utilisé si source = txt)
- tcp_host : adresse IP du serveur TCP (utilisé si source = tcp)
- tcp_port : port du serveur TCP (utilisé si source = tcp)

Mode 1 : source = "simulation"

Utilise la fonction generate_iq_samples() de mock.py pour créer un signal simulé. Idéal pour le développement sans matériel.

Mode 2 : source = "txt"

Lit un fichier texte contenant des résultats de scan RF (format CSV).

Processus de lecture du fichier texte :

- Étape 1 : Détermination du chemin

Si file_path n'est pas fourni, utilise scan_results.txt par défaut.

- Étape 2 : Lecture ligne par ligne

Le fichier est ouvert en mode lecture et chaque ligne est traitée. Les lignes vides sont ignorées.

- Étape 3 : Parsing des valeurs RF

Chaque ligne est séparée par des virgules. Les valeurs RF se trouvent à partir de la position 6 (positions 0-5 contiennent les métadonnées : date, heure, fréquences, etc.).

- Étape 4 : Conversion en nombres complexes

Chaque valeur RF est convertie en float et utilisée comme partie réelle d'un nombre complexe (partie imaginaire = 0). Les valeurs non numériques sont ignorées via un try-except.

- Étape 5 : Création du tableau

Toutes les valeurs collectées sont converties en tableau NumPy de dtype complex64.

Mode 3 : source = "tcp"

Récupère les échantillons IQ depuis un serveur TCP, typiquement GNU Radio qui stream les données en temps réel.

Configuration TCP :

- Host par défaut : 10.229.56.152
- Port par défaut : 5005

Ces valeurs peuvent être surchargées en passant tcp_host et tcp_port.

Gestion d'erreur :

Si la source spécifiée n'est pas reconnue, la fonction lève une ValueError avec le message Source IQ inconnue.

Architecture modulaire :

Ce design permet au reste du projet d'être indépendant de la source de données. On peut facilement :

- Tester avec des simulations
- Rejouer des captures enregistrées
- Passer en production avec un SDR réel

Sans modifier le code de détection et de localisation.

tcp_socket.py

Ce fichier implémente la communication TCP pour recevoir des échantillons IQ depuis un serveur externe (comme GNU Radio).

Fonction read_from_tcp (host, port, buffer size)

Cette fonction établit une connexion TCP, envoie une commande au serveur, et récupère les données binaires IQ.

Paramètres :

- host : adresse IP du serveur (défaut = 10.229.56.152)
- port : port TCP du serveur (défaut = 5005)
- buffer_size : taille du buffer de réception en octets (défaut = 4096)

Processus de communication :

- Étape 1 : Création du socket

Un socket TCP (SOCK_STREAM) est créé avec la famille d'adresses IPv4 (AF_INET).

- Étape 2 : Connexion au serveur

Le socket se connecte à l'adresse (host, port) spécifiée. Si le serveur n'est pas accessible, une exception socket.error sera levée.

- Étape 3 : Envoi de la commande

La commande hock-sweep\n est envoyée au serveur. Cette commande spécifique demande au serveur de démarrer un balayage fréquentiel et de retourner les données.

Note : hock-sweep semble être une commande propriétaire spécifique au serveur utilisé.

- Étape 4 : Réception des données

Le socket reçoit jusqu'à buffer_size octets de données binaires. Si plus de données sont disponibles, elles seront perdues (limitation actuelle).

- Étape 5 : Conversion en échantillons IQ

Les données brutes reçues sont converties en tableau NumPy de complex64 avec np.frombuffer(). Cela suppose que le serveur envoie les données au bon format (32 bits float pour I + 32 bits float pour Q).

- Étape 6 : Fermeture de la connexion

Le socket est fermé proprement avec sock.close().

Valeur de retour :

- Un tableau NumPy de dtype complex64 contenant les échantillons IQ reçus

Limitations actuelles :

- Une seule lecture de buffer_size octets
- Pas de gestion de reconnexion automatique

- Pas de timeout configuré
- Pas de validation des données reçues

Améliorations possibles :

- Implémenter une lecture en boucle pour collecter plus de données
- Ajouter un timeout pour éviter les blocages
- Vérifier la taille des données reçues
- Gérer les erreurs de connexion avec retry

Module storage

Ce module gère la persistance des données de détection dans une base de données PostgreSQL. Il permet de sauvegarder les détections, les événements de signal, et les alertes pour une analyse historique.

database.py

Ce fichier est le cœur du système de stockage. Il gère la connexion à PostgreSQL et fournit toutes les fonctions de lecture/écriture.

Configuration de la base de données

Chargement depuis .env :

Le fichier recherche un fichier .env dans le répertoire parent et charge les paramètres de connexion. Format du fichier .env :

```
DB_NAME=localisation_dronesdb DB_USER=admin DB_PASSWORD=admin DB_HOST=localhost DB_PORT=5432
```

Dictionnaire DB_CONFIG :

Contient les paramètres de connexion PostgreSQL :

- dbname : nom de la base de données
- user : nom d'utilisateur PostgreSQL
- password : mot de passe
- host : adresse IP du serveur (localhost pour local)
- port : port PostgreSQL (5432 par défaut)

Fonction get_connection()

Cette fonction établit une connexion à PostgreSQL.

Mécanisme de cache :

Une variable globale DB_AVAILABLE sert de cache pour éviter les tentatives de connexion répétées si la base est indisponible. Trois états possibles :

- None : premier appel, pas encore testé
- True : connexion réussie précédemment
- False : connexion échouée, ne plus réessayer

Gestion d'erreur :

Si la connexion échoue (psycopg2.Error), la fonction :

- Affiche un message d'erreur
- Marque DB_AVAILABLE = False
- Retourne None

Cela permet au système de continuer à fonctionner même sans base de données.

Fonction init_database()

Cette fonction crée les tables nécessaires si elles n'existent pas encore.

Tables créées :

- Table Drones

Stocke les informations sur chaque drone détecté :

- ID : identifiant unique auto-incrémenté (SERIAL PRIMARY KEY)
- drone_type : type de drone (TEXT NOT NULL)
- detected_at : timestamp de la détection (TIMESTAMP DEFAULT CURRENT_TIMESTAMP)

- Table Signal_events

Stocke les événements RF associés à chaque drone :

- ID : identifiant unique (SERIAL PRIMARY KEY)
- ID_Drones : référence au drone (INT REFERENCES Drones(ID))
- signal_strength : force du signal en dBm (INT)
- approx_distance : distance approximative (INT)
- relative_power : puissance relative (REAL)
- bandwidth : bande passante en Hz (REAL)
- score : score de confiance 0-100 (INT)
- detection_time : timestamp Unix (REAL)

- Table Alerts

Stocke les alertes générées par le système :

- ID : identifiant unique (SERIAL PRIMARY KEY)
- level : niveau d'alerte (TEXT NOT NULL)
- message : message descriptif (TEXT NOT NULL)
- created_at : timestamp de création (TIMESTAMP DEFAULT CURRENT_TIMESTAMP)

Mécanisme de transaction :

Si une erreur survient pendant la création des tables :

- conn.rollback() : annule toutes les modifications
- Le message d'erreur est affiché
- La fonction retourne False

Sinon, conn.commit() valide les changements et retourne True.

Fonction save_drone_detection(drone_type)

Enregistre un nouveau drone dans la base de données.

Paramètre :

- drone_type : type du drone détecté (défaut = “Unknown”)

Processus d'insertion :

- Exécute une requête INSERT INTO Drones
- Utilise RETURNING ID pour récupérer l'ID généré automatiquement
- Récupère cet ID avec cursor.fetchone()[0]
- Valide avec conn.commit()
- Retourne l'ID du drone créé

Fonction save_signal_event(drone_id, ...)

Enregistre un événement RF associé à un drone.

Paramètres :

- drone_id : ID du drone associé (INT)
- signal_strength : force du signal (INT)
- approx_distance : distance approximative (INT)
- relative_power : puissance relative (REAL)
- bandwidth : bande passante (REAL)
- score : score de confiance (INT)
- detection_time : timestamp Unix (REAL)

Fonction save_detection(timestamp, rssi, bandwidth, score, distance, drone_type)

Fonction de haut niveau qui combine save_drone_detection et save_signal_event en une seule transaction.

Processus :

- Étape 1 : Crée un nouveau drone avec save_drone_detection()
- Étape 2 : Si succès, crée l'événement RF associé avec save_signal_event()
- Étape 3 : Retourne (drone_id, event_id)

Conversion des données :

La fonction convertit les types Python vers les types PostgreSQL :

- timestamp.timestamp() : convertit datetime en timestamp Unix
- int(rssi) : convertit float en entier
- int(score * 100) : convertit score 0-1 en pourcentage 0-100

Fonction get_recent_detections(limit)

Récupère les dernières détections depuis la base de données.

Paramètre :

- limit : nombre maximum de détections à récupérer (défaut = 10)

Requête SQL utilisée :

La fonction exécute une jointure LEFT JOIN entre Drones et Signal_events pour récupérer à la fois les informations du drone et de l'événement RF associé. Les résultats sont triés par detected_at décroissant (plus récents en premier) et limités au nombre spécifié.

Valeur de retour :

Une liste de tuples, chaque tuple contenant :

- [0] : ID du drone
- [1] : Type du drone
- [2] : Timestamp de détection
- [3] : Force du signal
- [4] : Distance approximative
- [5] : Puissance relative
- [6] : Bande passante
- [7] : Score

models.py (storage)

Ce fichier redéfinit la classe Detection pour le module storage. C'est une duplication de geolocation/models.py.

Remarque sur l'architecture :

Il serait préférable d'avoir une seule définition de la classe Detection dans un module commun pour éviter la duplication de code. Cette structure actuelle peut mener à des incohérences si les deux classes évoluent différemment.

schema.sql

Ce fichier contient le schéma SQL complet pour créer la base de données.

Améliorations par rapport à database.py :

Le fichier schema.sql ajoute des index pour améliorer les performances :

- idx_drones_detected_at ON Drones(detected_at DESC)

Accélère les requêtes qui recherchent les détections les plus récentes.

- idx_signal_events_drone_id ON Signal_events(ID_Drones)

Accélère les jointures entre Drones et Signal_events.

- idx_alerts_level ON Alerts(level)

Permet de filtrer rapidement les alertes par niveau.

- idx_alerts_created_at ON Alerts(created_at DESC)

Accélère la récupération des alertes récentes.

Utilisation :

Ce fichier peut être exécuté directement avec psql pour créer la base complète :

```
psql -U admin -d localisation_dronesdb -f schema.sql
```

test_database.py

Ce fichier fournit un script de test pour vérifier le bon fonctionnement de la base de données.

Note importante :

Le script actuel est configuré en mode sans base de données, ce qui signifie que tous les tests retournent immédiatement sans rien faire. C'est visible dans les fonctions qui affichent des messages comme Test de connexion à PostgreSQL ignoré (mode sans base de données).

Fonctions de test définies :

- test_connection()

Devrait tester la connexion à PostgreSQL en appelant get_connection() et en vérifiant que la connexion réussit.

- test_init_database()

Devrait tester l'initialisation des tables en appelant init_database() et en vérifiant que toutes les tables sont créées.

- test_insert_data()

Devrait tester l'insertion de données dans les trois tables :

    - Insertion d'un drone avec save_drone_detection()
    - Insertion d'un événement RF avec save_signal_event()
    - Insertion d'une alerte avec save_alert() (note : cette fonction n'existe pas dans database.py)
    - Test de save_detection() qui combine drone + événement

- test_read_data()

Devrait tester la lecture des données :

    - Récupération des détections récentes avec get_recent_detections()
    - Récupération des alertes avec get_alerts() (note : cette fonction n'existe pas dans database.py)

Problèmes détectés :

    - Les fonctions save_alert() et get_alerts() sont appelées mais n'existent pas dans database.py
    - Le mode sans base de données rend les tests inutilisables
    - Pas de vérification réelle des résultats (assertions)

Pour activer les tests :

Il faudrait :

    - Retirer le mode sans base de données
    - Implémenter save_alert() et get_alerts() dans database.py
    - Ajouter des assertions pour vérifier les résultats

- Configurer une base de données de test séparée

Algorithmes RSSI

- Un RSSI est une opération permettant d'estimer le point d'origine d'une émission d'onde (en l'occurrence, radio) grâce à une trilatération utilisant la puissance des signaux captés. Les valeurs rendues par cette opération sont en *décibel - milliwatt* [dBm].

  Malheureusement, nous ne possèdons qu'une seule antenne, et au moment où j'écris ces lignes, nous n'avons pas encore trouvé d'alternative pour déterminer une position précise.

  C'est pourquoi nous nous contentons pour l'instant de calculer la distance du signal grâce à sa puissance.

- *Comment vous lissez les valeurs (moyenne mobile)*

- Pour convertir des dBm en km, nous avons utilisé une formule assez compliquée :

  $$d = 10^{((rssi_0 - rssi) / (10 * n))}$$

  Où *rssi* est la puissance reçue en dBm ;

  $rssi_0$ est la puissance reçue à 1m, également en dBm ;

  n est le facteur d'atténuation en *décibel par mètre* [dB/m] – exprime la perte de signal par rapport à la distance ;

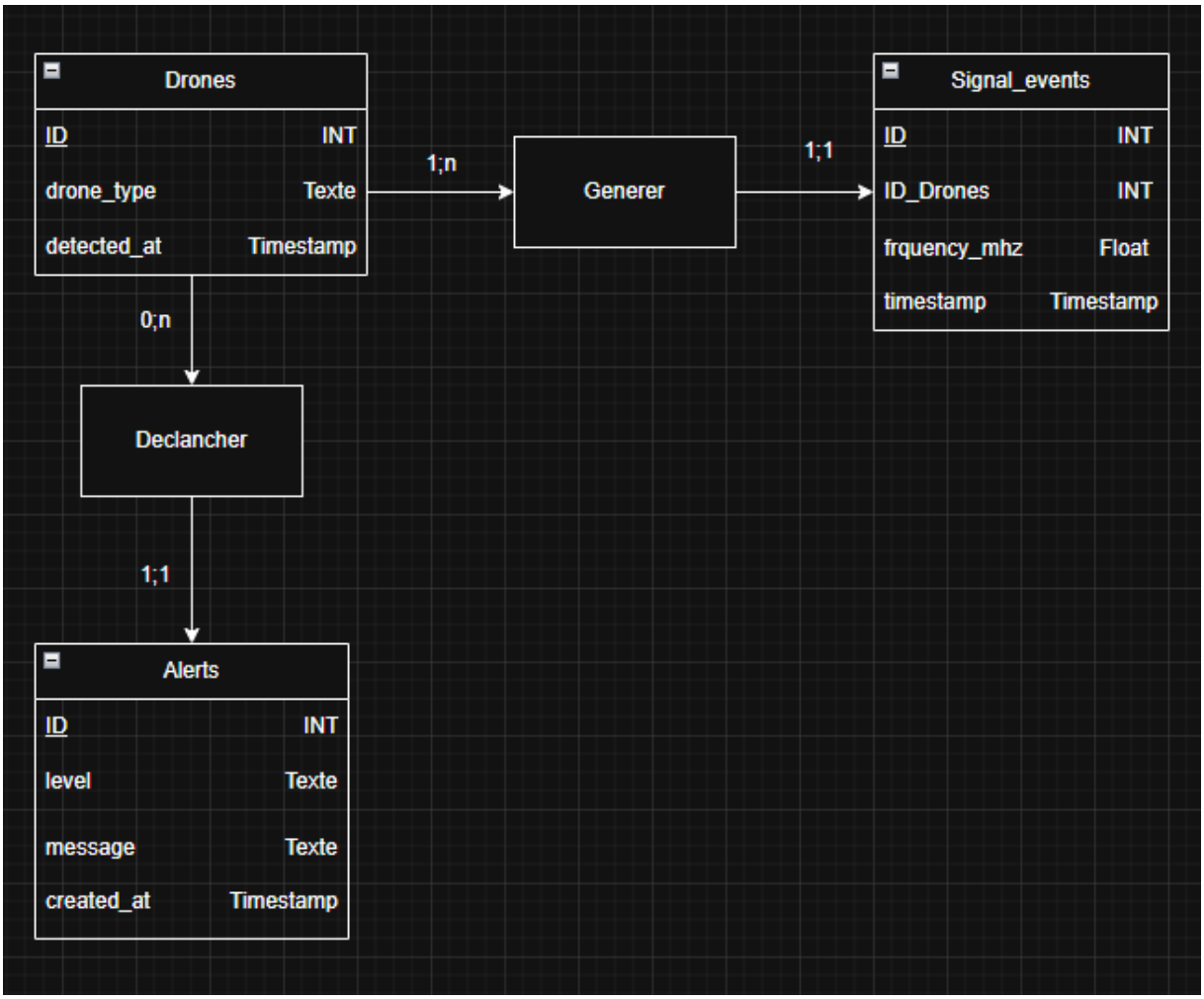
  et finalement, d_0 est la distance de référence.

Désavantages :

Il est hélas très difficile d'estimer des positions précises avec une seule antenne et sans décrypter les signaux. De plus, les distances calculées via RSSI peuvent être altérées par les éléments du décor et les conditions météo, les résultats obtenus peuvent donc parfois être assez approximatifs.

Base de données

Schéma SQL



Le schéma ci-dessous représente le Modèle Conceptuel de Données (MCD) de la base de données du projet. Il décrit les différentes entités (tables), leurs attributs ainsi que les relations existantes entre elles.

Ce MCD a servi de base pour la création du schéma SQL et l'implémentation de la base de données sous PostgreSQL.

Tables et relations

Table Drones

Rôle :
La table Drones représente les drones détectés par le système. Chaque enregistrement correspond à un drone identifié.

Colonnes principales :

- id (INT, clé primaire) : identifiant unique du drone
- drone_type (TEXT) : type ou catégorie du drone détecté
- detected_at (TIMESTAMP) : date et heure de la détection du drone

Relations :

- Un drone peut générer plusieurs événements de signal (1,n avec Signal_events)
- Un drone peut déclencher une alerte (0,n vers Alerts)

Table Signal_events

Rôle :
La table Signal_events stocke les événements liés aux signaux détectés provenant des drones.

Colonnes principales :

- id (INT, clé primaire) : identifiant de l'événement
- id_drones (INT, clé étrangère) : référence au drone concerné
- frequency_mhz (FLOAT) : fréquence du signal détecté
- timestamp (TIMESTAMP) : date et heure de l'événement

Relations :

- Chaque événement de signal est lié à un seul drone (1,1)
- Un drone peut générer plusieurs événements (1,n)

Table Alerts

Rôle :
La table Alerts contient les alertes générées par le système lorsqu'un comportement ou un signal particulier est détecté. Elle n'est pas utilisée par le projet.

Colonnes principales :

- id (INT, clé primaire) : identifiant de l'alerte
- level (TEXT) : niveau de gravité de l'alerte
- message (TEXT) : description de l'alerte

- created_at (TIMESTAMP) : date et heure de création de l'alerte

Relations :

- Une alerte est déclenchée par un drone
- La relation entre Drones et Alerts est de type 0,n vers 1,1

Clés primaires et étrangères

- Clés primaires :
  - Drones.id
  - Signal_events.id
  - Alerts.id
- Clés étrangères :
  - Signal_events.id_drones → Drones.id

Justification du choix

Pourquoi PostgreSQL ?

PostgreSQL a été choisi car le projet nécessite une base de données relationnelle, robuste et capable de gérer plusieurs connexions simultanées.

Avantages dans notre contexte :

- Support des relations complexes (clés étrangères)
- Bonnes performances en lecture/écriture
- Adapté à un serveur web multi-utilisateurs
- Compatible avec Docker, Node.js et Express
- Évolutif pour des besoins futurs (augmentation du volume de données)

Pourquoi pas SQLite ?

SQLite est adapté aux projets simples ou embarqués, mais il est moins approprié pour un système nécessitant plusieurs accès simultanés et une architecture client-serveur.

Interface Web

Architecture backend

Le backend constitue le cœur logique du système. Il assure la centralisation des données provenant des drones et leur redistribution vers l'interface utilisateur.

- **Framework utilisé : Flask (Python)**. Ce framework a été choisi pour sa légèreté, sa rapidité de déploiement et sa capacité à s'interfacer facilement avec des scripts de traitement de données en temps réel.
- **Format des échanges :** Toutes les communications entre le serveur et les clients se font au format **JSON**, garantissant une structure de donnée légère et universelle.

API

Le système expose plusieurs points d'entrée (endpoints) permettant la gestion complète de la flotte :

- `/login`
  - **Rôle :** Authentification sécurisée de l'utilisateur.
  - **Données :** Reçoit un identifiant et un mot de passe ; renvoie un jeton de session ou une confirmation de connexion.
- `/detections`
  - **Rôle :** Récupération de l'historique des objets détectés.
  - **Données :** Renvoie une liste d'objets (type d'objet, indice de confiance, horodatage).
- `/positions`
  - **Rôle :** Monitoring géographique de la flotte.
  - **Données :** Renvoie les coordonnées GPS (Latitude, Longitude, Altitude) des drones actifs.
- `/alerts`
  - **Rôle :** Gestion des alertes critiques.
  - **Données :** Renvoie les notifications d'incidents (intrusion, batterie faible, zone interdite).

Frontend

L'interface utilisateur est conçue pour offrir une expérience fluide et réactive aux opérateurs de sécurité.

- **Technologies utilisées : HTML5, CSS3, et JavaScript (Vanilla)**.
- **Structure de l'interface :**
    - **Accueil / Dashboard :** Présentation visuelle des statistiques clés du système (nombre de drones connectés, alertes en cours).
    - **Carte (Live Map) :** Visualisation dynamique des drones sur un fond de carte, permettant un suivi géographique précis.
    - **Historique :** Interface de consultation des données passées pour l'analyse après mission.

Communication en Temps Réel

L'élément central de la réactivité du système repose sur l'implémentation du **temps réel**.

- **Mécanisme :** Utilisation du protocole **WebSockets** via le module **Socket.io**.
- **Fonctionnement :** Contrairement au modèle HTTP classique (requête/réponse), le serveur "pousse" les informations vers le navigateur instantanément.
- **Application concrète :** Lorsqu'un drone détecte un mouvement, l'alerte apparaît sur le dashboard en moins d'une seconde, sans que l'opérateur n'ait besoin de rafraîchir sa page.

Gestion de l’authentification

- Login / mot de passe

L’utilisateur doit se connecter à l’aide d’un identifiant et d’un mot de passe via une page de login. Les informations saisies sont envoyées au backend qui vérifie leur validité avant d’autoriser l’accès…

- Sessions ou tokens

Après une connexion réussie, une session est créée.

Cette session permet de maintenir l’utilisateur connecté lors de la navigation entre les différentes pages du site.

Dans certain cas, un token peut-être utilisé pour identifier l’utilisateur lors des échanges entre le frontend et l’API.

- *Protection des routes*

Les routes sensibles de l’application (détections, positions, alertes, export) sont protégées.

Un utilisateur non authentifié ne peut pas accéder à ces routes.

Si l’utilisateur n’est pas connecté, il est automatiquement redirigé vers la page de login.

Conclusion

Le projet de détection et de localisation de drones a représenté un défi technique majeur, alliant radiofréquences, traitement de signal et développement logiciel complexe. Malgré les obstacles rencontrés, la majorité des objectifs a été atteinte.

Défis matériels et résilience

Le déroulement du projet a été fortement impacté par des contraintes logistiques et matérielles indépendantes de notre volonté. La réception tardive du HackRF One, à mi-parcours du projet, a considérablement réduit notre fenêtre de tests réels. De plus, les dysfonctionnements matériels suspectés sur l'unité reçue (sensibilité instable et non réception d'ondes) ont rendu l'acquisition de signaux réels difficilement exploitable. Avec le recul, une phase de diagnostic matériel plus précoce ou l'utilisation temporaire d'un matériel de secours aurait été préférable, mais l'incertitude sur la fiabilité du composant nous a forcés à faire preuve d'adaptabilité.

Stratégie de simulation

Afin de ne pas paralyser le développement du reste de l'infrastructure, nous avons pris la décision stratégique de basculer sur un mode de simulation de données RF. Ce choix s'est avéré crucial : il a permis de valider l'intégralité de la chaîne de valeur du projet.

Les scripts Python de détection et d'estimation ont pu être affinés sur des données simulées cohérentes.

La base de données (BDD) a été correctement structurée et peuplée.

L'application Web est aujourd'hui fonctionnelle et sécurisée, affichant en temps réel des détections et des trajectoires précises sur la carte.

Bilan final

Bien que la réception physique via le HackRF n'ait pas été optimale, le cœur du système est opérationnel. Nous livrons une plateforme modulaire et "Plug & Play" : l'architecture est prête à passer d'un mode simulation à une réception réelle dès qu'un matériel fonctionnel sera déployé. Ce projet nous a permis d'acquérir une expertise solide en gestion de flux de données temps réel et en ingénierie système, confirmant que la robustesse d'un logiciel se mesure aussi à sa capacité à simuler un environnement complexe lorsque le matériel fait défaut.

Ressenti personnel

Abdoulaye Sow :

Sur ce projet, ma mission s'est concentrée sur la création de l'interface de contrôle et de l'architecture backend. Mon objectif était de transformer des flux de données bruts en une expérience utilisateur fluide et exploitable.

- Conception de l'interface : Je me suis attaché à créer un dashboard professionnel où la donnée est lisible immédiatement. Le défi était de lier la carte temps réel et l'historique pour qu'un opérateur puisse prendre des décisions rapides.
- Architecture "Plug & Play" : Même si je n'ai pas géré la partie hardware (SDR/Radio), j'ai conçu le backend de manière à ce qu'il soit totalement modulaire. Grâce à l'API et au mode simulation, la plateforme est prête à recevoir les données de n'importe quel capteur radio dès qu'ils sont connectés.
- Apprentissage technique : Ce travail m'a permis de perfectionner ma gestion du temps réel avec Socket.io et la conteneurisation avec Docker. Assurer la communication entre une base de données PostgreSQL, un serveur Flask et une interface dynamique a été la partie la plus stimulante de mon travail.

Abhijeet Panwar :

Ce projet m’a aidé pour évaluer mes compétences en informatique et mieux connaître le domaine et le travail en groupe. Même si entant que première année je n’ai pas pu travailler beaucoup sur le sujet mais j’ai veçu le moment quand il faut travailler en groupe et comment.

Dani Dordevic :

Ce travail m’a permis de mieux comprendre les étapes à suivre. Même si certaines parties étaient difficiles au départ, j’ai gagné en confiance grâce à une méthode claire et à l’aide disponible. Cette expérience a été globalement positive et enrichissante.

Esteban Perez Narcisi :

Un module qui n’avait pas lieu d’être malgré la possibilité d’acquérir quelques compétences

Farah Mohamed Ahmed :

Le projet est bien mais pas le temps de le faire en 3 semaines et on n'a pas fait assez de teste pour régler les problèmes qu’on a eu lors du premier teste mais sinon c’était bien.

Louan Baconnier :

J’ai beaucoup aimé ces trois semaines de MA-MÉTIER. Je trouve ce type de projet particulièrement utile et intéressant, permettant d’entraîner nos compétences de travail de

groupe, sociales, informatiques ainsi que nos connaissances. L'ambiance était bonne et motivante.

Les choses à améliorer pour l'année prochaine seraient éventuellement d'homogénéiser au maximum les horaires de tout le monde (par ex. : 8h15-15h50 pour tout le monde tous les jours).

Jorge Miguel Pinto Costa :

Sur le plan personnel, ce projet a été une expérience formatrice et intense. Ayant pour ambition d'évoluer vers le management d'équipe, prendre la tête d'un groupe de sept personnes, soit le double des autres équipes, a été un défi de taille. Cette responsabilité m'a permis de me confronter aux réalités de la gestion humaine, notamment sur l'importance de la communication et de la répétition des consignes pour maintenir une cohésion de groupe. Malgré une répartition des tâches parfois inégale au sein de l'équipe, j'ai eu à cœur de porter le projet et d'assurer le suivi global, ce qui m'a permis de développer une vision transverse du système, de la radiofréquence à l'interface web.

En tant que référent technique, j'ai ressenti une pression importante, particulièrement face au point bloquant du HackRF. Être l'interlocuteur principal sur l'avancement du projet tout en gérant des problèmes matériels complexes a été une source de frustration, mais cela m'a forcé à chercher des solutions alternatives, comme la mise en place du mode simulation. Je tiens à remercier les professeurs pour leur soutien précieux lors de ces phases critiques. Finalement, malgré la brièveté du projet qui a condensé les enjeux, je repars avec une satisfaction réelle : celle d'avoir su adapter notre stratégie pour livrer un produit fonctionnel et d'avoir pu tester, en conditions réelles, les responsabilités d'un leader technique.

Thomas Mendes :

Faire un petit paragraphe sur ton ressenti.