

Rapport d'Expérimentation

Mini-projet : Géolocalisation sans GPS

WiFi Sniffing + LoRaWAN + Estimation de position + Visualisation

KANG Zhangyi

EI4 21506147

zhangyi.kang@etu.sorbonne-universite.fr

3 février 2026

Résumé

Ce rapport présente la conception et l'évaluation d'un système de géolocalisation sans GPS basé sur l'observation des points d'accès WiFi environnants (*RSSI*, *BSSID*, *SSID*), la transmission des mesures via LoRaWAN vers un serveur (TTN), l'estimation de position par différentes méthodes (pondération, trilateration, IA) réalisée côté Python, et la visualisation sur carte (Node-RED / Leaflet / OpenStreetMap).

Table des matières

1 Contexte et objectifs	3
1.1 Objectifs du mini-projet	3
2 Architecture globale du système	3
2.1 Vue d'ensemble	3
2.2 Matériel et logiciels	4
3 Étape 1 : WiFi sniffing et acquisition des données	4
3.1 Objectif de l'étape	4
3.2 Principe du WiFi sniffing	4
3.3 Implémentation sur ESP32	5
3.4 Validation expérimentale	5
4 Étape 2 : Transmission des données via LoRaWAN	5
4.1 Objectif de l'étape	5
4.2 Principe général	6
4.3 Implémentation sur ESP32 et LoRa-E5	6
4.4 Fragmentation des messages et gestion des contraintes LoRaWAN	6
4.5 Réception des données sur TTN	6
4.6 Réception, traitement et stockage des données côté serveur	7
5 Étape 3 : Construction de la base de données des points d'accès	8
5.1 Objectif de l'étape	8
5.2 Approche initiale basée sur l'API Wigle	8

5.3	Limites observées de l'API Wigle	9
5.4	Génération d'une base de données simulée	9
6	Étape 4 : Estimation de la position du nœud	10
6.1	Objectif et hypothèses	10
6.2	Jeu de données simulé pour l'évaluation	10
6.3	Modèle RSSI → distance	10
6.4	Méthodes comparées	10
6.5	Critères d'évaluation	11
6.6	Résultats	11
7	Étape 5 : Visualisation des résultats sur carte (Node-RED / OpenStreetMap)	11
7.1	Objectif	11
7.2	Architecture Node-RED	11
7.3	Rendu cartographique (Leaflet)	12
7.4	Problème d'affichage et correction	13
8	Discussion : limites et améliorations	13
A	Code ESP32 – WiFi sniffing passif	13
B	Code ESP32 – WiFi sniffing et transmission LoRaWAN	15
C	TTN – Uplink Payload Formatter	19
D	Script Python – Réception MQTT et stockage CSV	19
E	Script Python – Construction de la base AP via Wigle	21
F	Script Python – Génération d'une base AP simulée	25
G	Script Python – Génération de trames uplink simulées	28
H	Script Python – Estimation de position et comparaison des méthodes	31

1 Contexte et objectifs

1.1 Objectifs du mini-projet

- Déetecter des AP WiFi et extraire *SSID*, *BSSID*, *RSSI*, canal.
- Transmettre un ensemble d'observations via LoRaWAN (TTN).
- Construire une base de données locale des AP avec positions.
- Estimer la position du nœud à partir des *RSSI*.
- Visualiser les résultats (AP + position estimée) sur carte.

2 Architecture globale du système

2.1 Vue d'ensemble

La figure 1 illustre l'architecture globale du système de géolocalisation sans GPS développé dans ce projet. Le système est organisé autour d'une chaîne de traitement distribuée, dans laquelle chaque composant matériel et logiciel remplit un rôle bien défini.

Le nœud embarqué ESP32 est chargé de l'acquisition des données. Il effectue un WiFi sniffing passif afin de détecter les points d'accès environnants et de mesurer leurs caractéristiques, notamment l'adresse MAC (BSSID) et la puissance du signal reçu (RSSI). Ces informations sont transmises au module LoRa-E5 via une liaison série (UART).

Le module LoRa-E5 assure ensuite l'envoi des données vers le réseau LoRaWAN. Les trames sont reçues par The Things Network (TTN), qui joue le rôle de serveur réseau et permet l'acheminement des messages vers l'infrastructure serveur via MQTT.

Le serveur Python, basé sur FastAPI, constitue le cœur du système. Il est responsable de la réception et du décodage des messages MQTT, de la construction de la base de données des points d'accès WiFi, ainsi que du calcul de la position estimée du nœud à partir des mesures RSSI. Le serveur peut également exploiter des sources de données externes, telles que l'API Wigle, afin d'enrichir la base de données des points d'accès avec des informations de localisation.

Enfin, les positions estimées sont publiées via MQTT et exploitées par Node-RED. Node-RED est utilisé pour le stockage final des résultats et leur visualisation sous forme de tableau de bord et de carte, sans intervenir dans les calculs de géolocalisation.

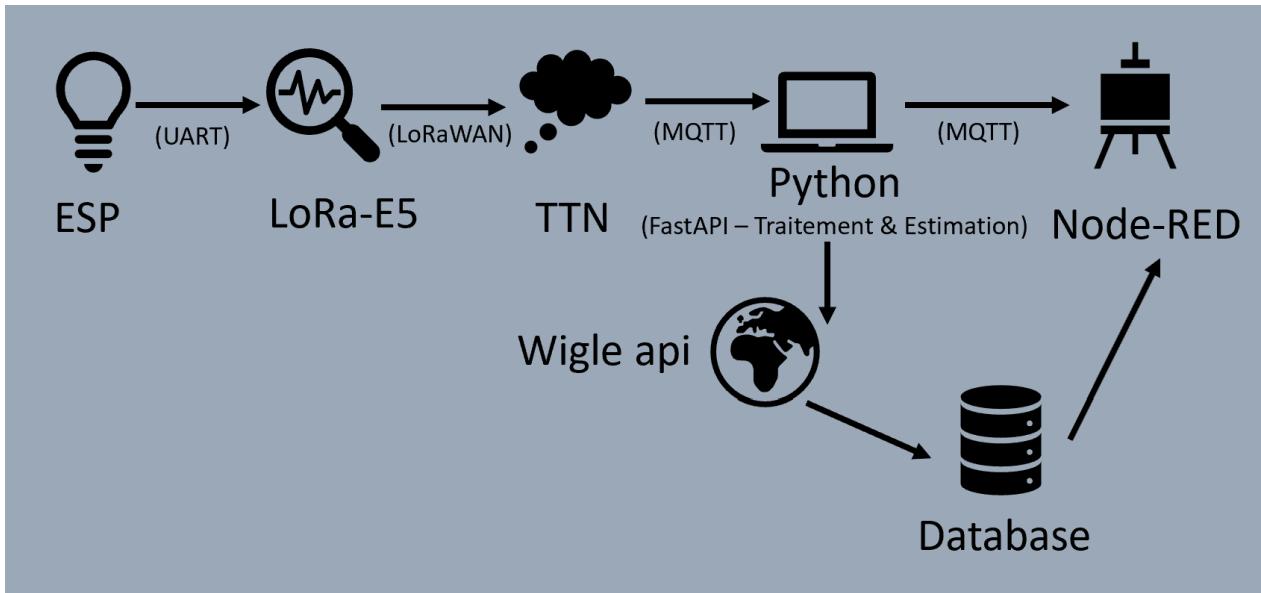


FIGURE 1 – Architecture globale du système de géolocalisation sans GPS

2.2 Matériel et logiciels

- Matériel : ESP32, modem LoRa-E5, alimentation, PC.
- Logiciels : Arduino/PlatformIO, TTN, Node-RED, Python/FastAPI, Leaflet/OpenStreetMap.

3 Étape 1 : WiFi sniffing et acquisition des données

3.1 Objectif de l'étape

L'objectif de cette première étape est d'identifier et de caractériser les points d'accès WiFi présents dans l'environnement du nœud, sans établir de connexion active. Les informations collectées serviront de base à la construction d'une base de données des points d'accès et, ultérieurement, à l'estimation de la position du dispositif sans recours au GPS.

3.2 Principe du WiFi sniffing

Le WiFi sniffing consiste à utiliser l'ESP32 en mode station passive afin de détecter les points d'accès diffusant des trames de balisage (*beacon frames*). Contrairement à un mode client classique, aucune association avec les réseaux détectés n'est réalisée, ce qui permet une observation non intrusive de l'environnement radio.

Pour chaque point d'accès détecté, les paramètres suivants sont extraits :

- le nom du réseau (*SSID*),
- l'adresse MAC du point d'accès (*BSSID*),
- la puissance du signal reçu (*RSSI*, en dBm),
- le canal WiFi utilisé.

3.3 Implémentation sur ESP32

L'ESP32 est configuré en mode station (`WIFI_STA`) avec désactivation de toute connexion automatique. Le scan WiFi est déclenché périodiquement à l'aide de la fonction `WiFi.scanNetworks()`, en incluant également les réseaux à *SSID*caché.

Chaque cycle de scan dure environ 1 à 1,5 seconde et permet d'obtenir une liste complète des points d'accès visibles à un instant donné. Les résultats sont ensuite formatés sous forme de messages JSON, facilitant leur transmission et leur traitement lors des étapes suivantes du projet.

Le code source complet de cette implémentation est présenté en annexe (Annexe A).

3.4 Validation expérimentale

Afin de valider le bon fonctionnement du WiFi sniffing, les sorties série de l'ESP32 ont été observées à l'aide du moniteur série. La figure 2 présente un exemple de trames affichées lors d'un scan WiFi, montrant les points d'accès détectés ainsi que leurs paramètres (*SSID*, *BSSID*, *RSSI* et canal).

```
10:54:49.329 -> ✓ 25 AP detecte:  
10:54:49.360 -> {"ssid":"Narcisss", "mac":":06:E8:40:A7:47:F3", "rssi":-28, "ch":6}  
10:54:49.360 -> {"ssid":"makamaka", "mac":":3A:F8:1B:47:F6:66", "rssi":-43, "ch":11}  
10:54:49.360 -> {"ssid":"Core-M5", "mac":":1A:72:F5:26:AE:57", "rssi":-57, "ch":11}  
10:54:49.360 -> {"ssid": "", "mac":":B0:6E:BF:66:61:08", "rssi":-67, "ch":4}  
10:54:49.360 -> {"ssid":"DIRECT-H1iscEB86CDA6G0tF", "mac":":52:57:9C:BF:C5:44", "rssi":-74, "ch":11}  
10:54:49.392 -> {"ssid":"SCAI", "mac":":58:97:BD:CD:93:E3", "rssi":-75, "ch":1}  
10:54:49.392 -> {"ssid":"eduspot", "mac":":58:97:BD:CD:93:E0", "rssi":-75, "ch":1}  
10:54:49.392 -> {"ssid":"eduroam", "mac":":58:97:BD:CD:93:E2", "rssi":-75, "ch":1}  
10:54:49.392 -> {"ssid":"CONGRES", "mac":":58:97:BD:CD:93:E1", "rssi":-75, "ch":1}  
10:54:49.392 -> {"ssid":"ISCD", "mac":":E8:65:D4:72:F9:79", "rssi":-76, "ch":6}  
10:54:49.392 -> {"ssid":"eduspot", "mac":":58:97:BD:CD:9C:60", "rssi":-82, "ch":6}  
10:54:49.424 -> {"ssid":"SCAI", "mac":":58:97:BD:CD:9C:63", "rssi":-83, "ch":6}  
10:54:49.424 -> {"ssid":"CONGRES", "mac":":58:97:BD:CD:9C:61", "rssi":-84, "ch":6}  
10:54:49.424 -> {"ssid":"eduroam", "mac":":58:97:BD:CD:9C:62", "rssi":-84, "ch":6}  
10:54:49.424 -> {"ssid":"ISCD", "mac":":E8:65:D4:72:F9:69", "rssi":-84, "ch":11}  
10:54:49.424 -> {"ssid":"M", "mac":":32:00:E2:FB:E0:3B", "rssi":-87, "ch":1}  
10:54:49.424 -> {"ssid":"eduspot", "mac":":70:DF:2F:9E:25:C0", "rssi":-87, "ch":6}  
10:54:49.457 -> {"ssid":"eduroam", "mac":":70:DF:2F:9E:25:C2", "rssi":-87, "ch":6}  
10:54:49.457 -> {"ssid":"eduroam", "mac":":58:97:BD:80:CC:B2", "rssi":-87, "ch":11}  
10:54:49.457 -> {"ssid":"CONGRES", "mac":":58:97:BD:80:CC:B1", "rssi":-87, "ch":11}  
10:54:49.457 -> {"ssid":"eduspot", "mac":":58:97:BD:80:CC:B0", "rssi":-88, "ch":11}  
10:54:49.457 -> {"ssid":"SCAI", "mac":":70:DF:2F:9E:25:C3", "rssi":-89, "ch":6}  
10:54:49.457 -> {"ssid":"SCAI", "mac":":58:97:BD:80:CC:B3", "rssi":-89, "ch":11}  
10:54:49.504 -> {"ssid":"SCAI", "mac":":58:97:BD:CD:92:63", "rssi":-91, "ch":1}  
10:54:49.504 -> {"ssid":"CONGRES", "mac":":58:97:BD:CD:92:61", "rssi":-94, "ch":1}
```

FIGURE 2 – Sortie du moniteur série de l'ESP32 lors du WiFi sniffing passif

Ces résultats confirment que l'ESP32 est capable de détecter de manière fiable les points d'accès WiFi environnants et de fournir des mesures exploitables pour les étapes suivantes du projet.

4 Étape 2 : Transmission des données via LoRaWAN

4.1 Objectif de l'étape

Cette étape vise à transmettre les données issues du WiFi sniffing vers un serveur central en utilisant le réseau LoRaWAN. Les contraintes principales sont la taille limitée des trames, le respect du duty-

cycle et la fiabilité de la transmission.

4.2 Principe général

Après chaque scan WiFi, les informations collectées sont regroupées dans une structure JSON compacte, puis transmises via un modem LoRa-E5 configuré en mode OTAA sur le réseau The Things Network (TTN).

Afin de respecter les limitations de taille de charge utile imposées par LoRaWAN, les résultats d'un scan sont fragmentés en plusieurs messages, chacun contenant un nombre limité de points d'accès WiFi.

4.3 Implémentation sur ESP32 et LoRa-E5

L'ESP32 assure à la fois le WiFi sniffing et la communication avec le modem LoRa-E5 via une interface série UART.

Le modem est configuré en mode LoRaWAN OTAA (classe A, région EU868). Une procédure de *join* avec tentatives successives est mise en place afin d'améliorer la robustesse lors de l'initialisation du lien LoRaWAN.

Les données WiFi sont encodées en JSON, puis converties en format hexadécimal avant l'envoi, conformément aux commandes AT du module LoRa-E5. Le code complet de cette implémentation est fourni en annexe (Annexe [B](#)).

4.4 Fragmentation des messages et gestion des contraintes LoRaWAN

Lors des premiers tests, l'envoi de l'ensemble des points d'accès détectés dans une seule trame LoRaWAN s'est révélé impossible en raison de la taille excessive du message. De plus, l'envoi de messages successifs trop rapprochés provoquait des réponses du type *module busy* de la part du modem LoRa-E5.

Pour résoudre ces problèmes, une stratégie de fragmentation a été mise en place :

- chaque scan WiFi est associé à un identifiant unique (**sid**) ;
- les points d'accès sont répartis sur plusieurs fragments numérotés (**i/n**) ;
- un délai est introduit entre deux transmissions successives afin de respecter le duty-cycle LoRaWAN et d'éviter la saturation du module.

Cette approche permet d'assurer une transmission fiable et complète des données, tout en restant compatible avec les contraintes du réseau LoRaWAN.

4.5 Réception des données sur TTN

Les messages LoRaWAN sont reçus par le serveur The Things Network et visualisés en temps réel à l'aide de l'interface *Live Data*. Le Décodeur de payload Uplink (TTN) est fourni en annexe (Annexe [C](#)).

La figure [3](#) présente la liste des messages uplink reçus, tandis que la figure [4](#) montre le contenu détaillé d'un message décodé.

↑ 12:13:41	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 2, n: 5, sid: 1009932, t: 1009 }	7B 22 73 69 64 22 3A 31
↑ 12:13:41	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:13:32	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 1, n: 5, sid: 1009932, t: 1009 }	7B 22 73 69 64 22 3A 31
↑ 12:13:32	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:12:42	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 4, n: 5, sid: 934688, t: 934 }	7B 22 73 69 64 22 3A 39 33
↑ 12:12:42	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:12:34	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 3, n: 5, sid: 934688, t: 934 }	7B 22 73 69 64 22 3A 39 33
↑ 12:12:34	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:12:25	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 2, n: 5, sid: 934688, t: 934 }	7B 22 73 69 64 22 3A 39 33
↑ 12:12:25	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:12:17	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 1, n: 5, sid: 934688, t: 934 }	7B 22 73 69 64 22 3A 39 33
↑ 12:12:17	Successfully processed dat...	DevAddr: 26 0B 3C B4		
↑ 12:12:09	Forward uplink data message	DevAddr: 26 0B 3C B4	Payload: { a: [...], i: 0, n: 5, sid: 934688, t: 934 }	7B 22 73 69 64 22 3A 39 33

FIGURE 3 – Liste des messages uplink reçus sur TTN (Live Data)

The screenshot shows the TTN Device Overview interface for a device named "device2". The interface includes tabs for "Device overview", "Live data" (which is selected), "Messaging", "Location", "Payload formatters", and "Settings". The "Live data" tab displays a table of uplink messages with columns for TIME, TYPE, DATA PREVIEW, and EVENT DETAILS. The table lists 14 messages from 12:13:41 to 12:12:09. The "DATA PREVIEW" column shows DevAddrs and some numerical values. The "EVENT DETAILS" column shows detailed message payloads and decoded payloads. One message at index 37 is expanded to show its decoded payload, which contains an array of strings representing MAC addresses and network names.

TIME	TYPE	DATA PREVIEW	EVENT DETAILS
↑ 12:13:57	Successfully processed dat...	DevAddr: 26 0B 3C	
↓ 12:13:49	Schedule data downlink for...	DevAddr: 26 0B 3C	
↑ 12:13:49	Forward uplink data message	DevAddr: 26 0B 3C	37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
↑ 12:13:41	Forward uplink data message	DevAddr: 26 0B 3C	"decoded_payload": { "a": [["58:97:BD:BB:FE:C1", -59, 6, "CONGRES"], ["58:97:BD:CD:92:60", -71, 1, "eduspot"], ["58:97:BD:CD:92:62", -71, 1, "edu roam"], ["58:97:BD:CD:92:61", -71, 1, "CONGRES"]] }
↑ 12:13:32	Forward uplink data message	DevAddr: 26 0B 3C	
↑ 12:13:32	Successfully processed dat...	DevAddr: 26 0B 3C	
↑ 12:12:42	Forward uplink data message	DevAddr: 26 0B 3C	
↑ 12:12:42	Successfully processed dat...	DevAddr: 26 0B 3C	
↑ 12:12:34	Forward uplink data message	DevAddr: 26 0B 3C	
↑ 12:12:34	Successfully processed dat...	DevAddr: 26 0B 3C	
↑ 12:12:25	Forward uplink data message	DevAddr: 26 0B 3C	
↑ 12:12:25	Successfully processed dat...	DevAddr: 26 0B 3C	

FIGURE 4 – Détail d'un message uplink décodé sur TTN

4.6 Réception, traitement et stockage des données côté serveur

Les messages uplink décodés par TTN sont récupérés via le protocole MQTT et traités par une application Python.

Cette application repose sur deux composants principaux :

- un client MQTT, utilisé pour s'abonner aux messages *uplink* publiés par The Things Network ;

- un service web léger basé sur *FastAPI*, permettant de superviser l'état du système et de préparer une exploitation ultérieure des données.

Chaque fragment est décodé et ensuite décomposé en enregistrements élémentaires correspondant aux différents points d'accès détectés. Les données sont stockées de manière persistante dans un fichier CSV.

Le code complet de cette application Python est présenté en annexe (Annexe D).

La structure du fichier `wifi_sniffing_data.csv` est présentée dans le tableau 1.

Champ	Description
received_at	Horodatage de réception côté TTN
device_id	Identifiant du noeud LoRaWAN
sid	Identifiant du scan WiFi
frag_i / frag_n	Index et nombre total de fragments
esp_t	Temps interne ESP32
ap_index	Index du point d'accès dans le fragment
bssid	Adresse MAC du point d'accès
rssi	Puissance du signal reçu (dBm)
channel	Canal WiFi
ssid	Nom du réseau WiFi

TABLE 1 – Structure du fichier `wifi_sniffing_data.csv`

5 Étape 3 : Construction de la base de données des points d'accès

5.1 Objectif de l'étape

Cette étape a pour objectif d'associer à chaque point d'accès WiFi détecté une position géographique estimée. Cette base de données constitue un élément central du système, car elle permet de relier les mesures RSSI à des coordonnées spatiales lors de l'estimation de la position du noeud.

5.2 Approche initiale basée sur l'API Wigle

Dans un premier temps, une approche automatique basée sur l'API publique de Wigle.net a été mise en œuvre afin d'obtenir les coordonnées GPS associées aux points d'accès WiFi détectés. À partir des données collectées lors de l'étape précédente, les couples (*BSSID*, *SSID*) uniques ont été extraits puis utilisés comme clés de recherche auprès de l'API Wigle.

Afin de limiter les erreurs de géolocalisation, plusieurs mécanismes de filtrage ont été introduits :

- restriction de la recherche à une zone géographique correspondant à l'agglomération parisienne ;
- filtrage par nom de réseau (*SSID*) lorsque celui-ci est disponible ;
- normalisation des adresses MAC afin d'assurer la cohérence des requêtes.

Le script Python complet implémentant cette méthode est présenté en annexe (Annexe E).

5.3 Limites observées de l'API Wigle

Les expérimentations ont toutefois mis en évidence plusieurs limites importantes dans l'utilisation de l'API Wigle pour la construction d'une base de données fiable.

Tout d'abord, en l'absence de contrainte géographique stricte, les coordonnées retournées peuvent se situer très loin de la zone réelle d'expérimentation, rendant les résultats inexploitables pour un usage local.

Ensuite, lorsque le filtrage par *SSID*n'est pas appliqué, certaines réponses associent une adresse MAC à un nom de réseau différent de celui observé lors du WiFi sniffing, ce qui introduit une ambiguïté sur l'identité réelle du point d'accès.

Enfin, même après application de ces filtres, les informations temporelles fournies par Wigle (*last-time*) révèlent que de nombreux points d'accès n'ont pas été observés récemment. Certains enregistrements reposent sur des observations datant de plusieurs années, ce qui est problématique dans des environnements WiFi dynamiques.

BSSID	SSID	Latitude	Longitude	QoS	Dernière observation
04:D9:F5:74:E0:B0	PPIAdmin	48.8533	2.3773	0	2023-02-12
08:5A:11:28:8A:10	fablabstaff	48.8027	2.3735	7	2021-10-21
08:5A:11:28:8A:12	fablabo	48.8376	2.4162	5	2007-10-03
08:5A:11:28:8A:14	fablab-bad	48.8401	2.3871	1	2019-10-19
18:64:72:05:D9:20	IPGP-Guests	48.8610	2.2592	0	2009-12-10

TABLE 2 – Exemples de résultats retournés par l'API Wigle montrant l'hétérogénéité de la qualité et de la fraîcheur des données

5.4 Génération d'une base de données simulée

En raison de la fiabilité limitée des données issues de bases publiques, une base de données simulée des points d'accès WiFi a été utilisée pour les expérimentations.

Chaque adresse MAC (*BSSID*) détectée lors du WiFi sniffing est associée à une position géographique artificielle, générée aléatoirement dans une zone restreinte autour de la position réelle du dispositif. La génération est réalisée de manière contrôlée et reproductible à l'aide d'une graine fixe.

La base de données obtenue est stockée dans un fichier CSV, dont la structure est présentée dans le tableau 3. Le script Python correspondant est fourni en annexe (Annexe F).

Champ	Description
bssid	Adresse MAC du point d'accès WiFi
lat	Latitude simulée du point d'accès
lon	Longitude simulée du point d'accès
source	Origine et paramètres de génération (centre, rayon, graine)

TABLE 3 – Structure de la base de données simulée des points d'accès WiFi

6 Étape 4 : Estimation de la position du nœud

6.1 Objectif et hypothèses

L'objectif est d'estimer la position du nœud à partir des RSSI mesurés sur un ensemble de points d'accès dont la position est connue (base AP, Étape 3). Dans la suite, les tests sont réalisés sur des données simulées cohérentes avec le périmètre expérimental (rayon ≈ 60 m), afin d'évaluer et comparer plusieurs méthodes de positionnement.

6.2 Jeu de données simulé pour l'évaluation

Un générateur de trames *uplink* au format TTN a été mis en place afin de produire des observations réalistes : une position “vraie” du noeud est tirée aléatoirement autour du point central, puis les RSSI sont simulés à partir de la distance aux AP (avec bruit). Pour chaque trame, seuls les $N = 6$ AP les plus forts sont conservés, ce qui correspond à une contrainte réaliste de charge utile LoRaWAN.

Le script de génération est fourni en annexe (Annexe G).

6.3 Modèle RSSI → distance

Pour la méthode par multilateration, le RSSI est converti en distance via un modèle log-distance :

$$d = d_0 \cdot 10^{\frac{RSSI_0 - RSSI}{10n}}, \quad (1)$$

où $RSSI_0$ est le RSSI à $d_0 = 1$ m, et n l'exposant de perte de trajet. Dans nos tests simulés, les paramètres ont été choisis cohérents avec la génération des données ($RSSI_0$ et n fixés).

6.4 Méthodes comparées

Trois approches ont été comparées :

(M1) Multilateration (mathématique). Après conversion des RSSI en distances, la position $\hat{\mathbf{x}}$ est obtenue en minimisant l'erreur au sens des moindres carrés entre distances estimées et distances géométriques aux AP (résolution itérative type Gauss–Newton). Cette méthode nécessite au moins trois AP.

(M2) Barycentre pondéré par RSSI (algorithmique). La position est approximée par une moyenne pondérée des coordonnées des AP :

$$\hat{\mathbf{x}} = \frac{\sum_i w_i \mathbf{x}_i}{\sum_i w_i}, \quad w_i \propto 10^{\frac{RSSI_i + C}{10}}, \quad (2)$$

où \mathbf{x}_i est la position de l'AP i et C est une constante de décalage utilisée pour éviter des poids trop faibles.

(M3) KNN “fingerprinting” (IA). Un modèle KNN est appliqué sur un vecteur RSSI (*fingerprint*) construit sur un sous-ensemble d'AP. Les vecteurs d'entraînement sont générés par simulation sur une grille aléatoire de positions ; l'inférence consiste à rechercher les k voisins les plus proches en espace RSSI et à moyenner leurs positions (avec pondération).

Le script complet d'évaluation (les trois méthodes + comparaison) est fourni en annexe (Annexe H).

6.5 Critères d'évaluation

Deux indicateurs ont été utilisés :

- **cohérence RSSI (RMSE en dB)** : à partir d'une position estimée, on prédit le RSSI attendu pour chaque AP via le modèle, puis on calcule l'écart quadratique moyen avec le RSSI observé ;
- **résidu de distances (RMS en m)** pour la multilateration : mesure la cohérence interne entre les distances estimées et la solution optimisée.

6.6 Résultats

Les résultats sur 20 trames simulées (6 observations par trame) montrent que la multilateration (ML) obtient, en moyenne, la meilleure cohérence RSSI, suivie de près par le barycentre pondéré (WC). La méthode KNN, entraînée ici sur un jeu de données synthétique, reste moins stable et présente une erreur moyenne nettement plus élevée.

Méthode	Meilleure (nb)	RMSE RSSI moyen (dB)	Rang moyen (1=meilleur)
WC (pondérée)	7	4.237	1.8
ML (multilateration)	12	4.127	1.4
KNN (fingerprinting)	1	12.804	2.8

TABLE 4 – Synthèse des performances sur 20 trames simulées.

À noter que la multilateration présente également un résidu de distances moyen de l'ordre de 3.08 m, ce qui indique une solution globalement cohérente avec les distances déduites des RSSI (dans le cadre du modèle simulé).

7 Étape 5 : Visualisation des résultats sur carte (Node-RED / OpenStreetMap)

7.1 Objectif

L'objectif de cette étape est de fournir une visualisation claire et exploitable : (i) les points d'accès connus (base AP), (ii) les AP effectivement utilisés lors d'une estimation, et (iii) la position estimée du nœud en temps (quasi) réel.

7.2 Architecture Node-RED

La visualisation est réalisée avec Node-RED (Dashboard) et une carte Leaflet basée sur OpenStreetMap. Le flux se décompose en deux entrées principales :

- **Chargement de la base AP** au démarrage : lecture du fichier `ap_db.csv`, parsing, puis injection sur la carte (topic `apdb`).

- Réception des résultats de géolocalisation via MQTT (topic `sim/geo_result`) : parsing JSON puis séparation en deux messages destinés à la carte : (a) la couche `obs_aps` (AP utilisés), (b) la couche `node` (position estimée).

La figure 5 présente la structure du flux Node-RED.

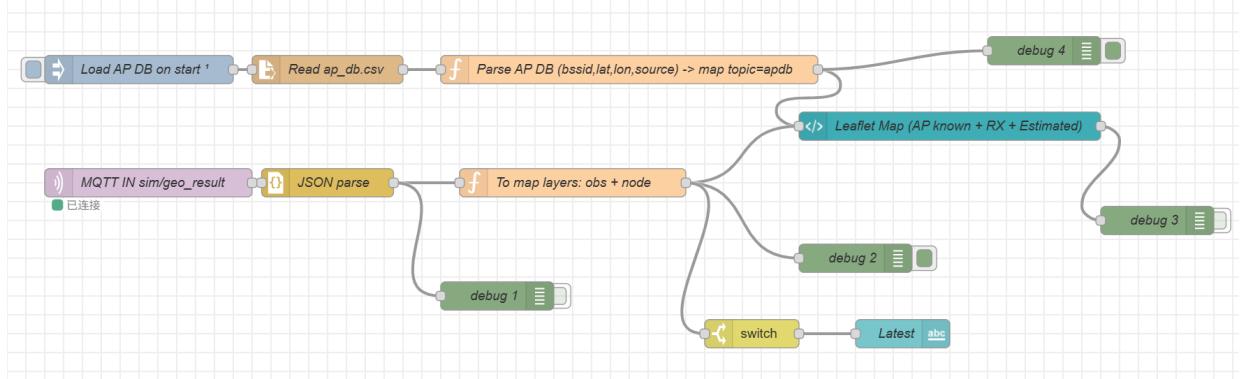


FIGURE 5 – Flux Node-RED : chargement de la base AP, réception MQTT, séparation des couches et affichage Dashboard

7.3 Rendu cartographique (Leaflet)

La carte Leaflet est structurée en trois calques superposés :

- **AP connus (DB)** : marqueurs fixes issus de `ap_db.csv`.
- **AP observés / utilisés** : marqueurs affichés à partir des données reçues, avec une taille liée au RSSI afin d'indiquer visuellement la force du signal.
- **Position estimée du noeud** : marqueur distinct (rouge) mis à jour à chaque message, avec recentrage automatique de la carte.

Chaque marqueur contient une fenêtre d'information (*popup*) facilitant le contrôle des données (BS-SID, RSSI, canal, métadonnées `sid/ts`, méthode et nombre d'AP utilisés).

La figure 6 illustre l'interface finale : points d'accès connus, AP utilisés, et position estimée du noeud sur la carte.

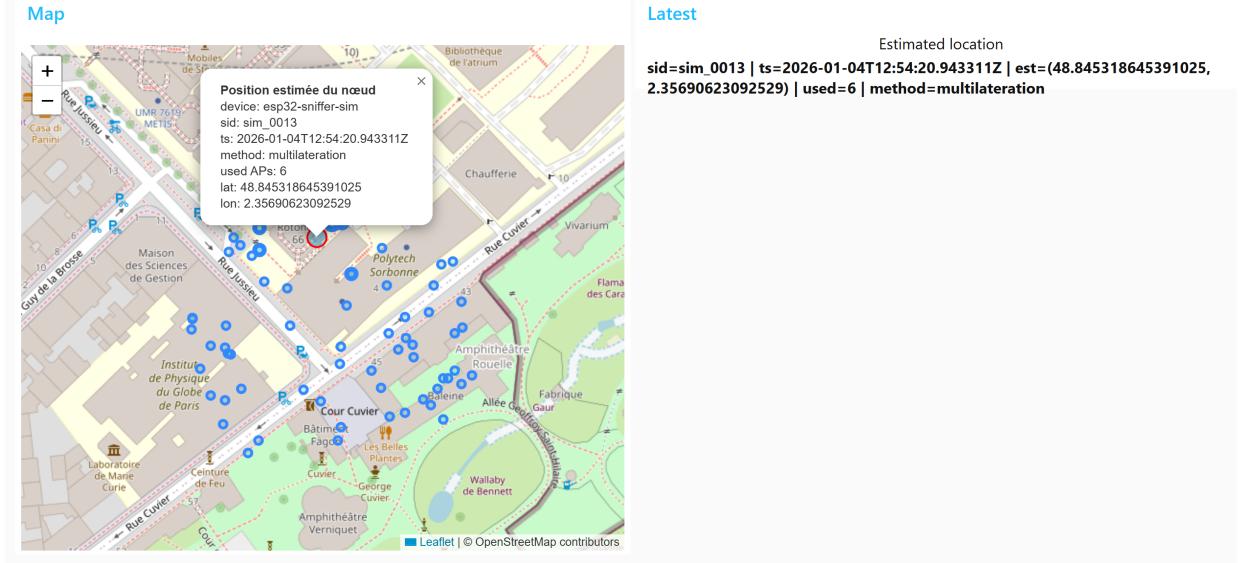


FIGURE 6 – Interface Dashboard : AP connus (DB), AP utilisés (observations) et position estimée du nœud

7.4 Problème d'affichage et correction

Lors des premiers tests, la carte OpenStreetMap n'était rendue que partiellement dans l'interface, un comportement classique lorsque Leaflet est initialisé avant que le conteneur HTML ne soit dimensionné (Dashboard/onglet). La correction retenue consiste à forcer un recalcul de taille via `invalidateSize()` après l'initialisation (avec un délai), ce qui stabilise l'affichage.

8 Discussion : limites et améliorations

Les expérimentations montrent que la géolocalisation par WiFi sniffing est fonctionnelle dans un cadre contrôlé, mais que la précision reste limitée par plusieurs facteurs.

La principale source d'incertitude provient du RSSI, dont la variabilité dépend fortement de l'environnement (obstacles, multi-trajets). Les modèles RSSI–distance utilisés sont donc approximatifs et sensibles aux paramètres choisis.

La construction de la base des points d'accès constitue également un point critique. L'utilisation d'une base simulée a permis de comparer les méthodes de positionnement, mais ne reflète pas entièrement un environnement réel, où les AP peuvent être déplacés ou reconfigurés.

Des améliorations possibles incluent une calibration expérimentale du modèle RSSI, l'introduction d'un filtrage temporel des mesures et l'enrichissement de la base AP par des relevés réels.

A Code ESP32 – WiFi sniffing passif

Listing 1 – ESP32 en mode station passive pour le WiFi sniffing

```

1  /*
2   * ESP32 en mode station passive (scan WiFi)

```

```

3  * Sortie série à 115200 bauds :
4  *   SSID, BSSID (MAC), RSSI, canal
5  * Durée typique d'un scan : environ 1 à 1,5 seconde
6  */
7
8 #include <WiFi.h>
9
10 void setup() {
11     Serial.begin(115200);
12
13     // Configuration de l'ESP32 en mode station (sans connexion)
14     WiFi.mode(WIFI_STA);
15     WiFi.disconnect();
16     delay(100);
17
18     Serial.println("\n-----\u2022ESP32\u2022WiFi\u2022Sniffer\u2022-----");
19 }
20
21 void loop() {
22     // Lancement d'un scan WiFi passif
23     // Les deux paramètres à true permettent de détecter
24     // également les réseaux à SSID caché
25     int n = WiFi.scanNetworks(false, true);
26
27     if (n == 0) {
28         Serial.println("Aucun\u2022point\u2022d'accès\u2022WiFi\u2022detecté");
29     } else {
30         Serial.printf("%d\u2022points\u2022d'accès\u2022WiFi\u2022detectés\u2022:\n", n);
31
32         for (int i = 0; i < n; ++i) {
33             String ssid      = WiFi.SSID(i);
34             String bssid     = WiFi.BSSIDstr(i);    // Format : 84:16:F9:AA:3C:21
35             int32_t rssi      = WiFi.RSSI(i);
36             uint8_t channel   = WiFi.channel(i);
37
38             // Affichage des résultats sous forme JSON
39             Serial.printf(
40                 "{\"ssid\":\"%s\", \"mac\": \"%s\", \"rssi\":%d, \"ch\":%d}\n",
41                 ssid.c_str(),
42                 bssid.c_str(),
43                 rssi,
44                 channel
45             );
46         }
47     }
48
49     // Libération de la mémoire utilisée par le scan
50     WiFi.scanDelete();
51 }
```

```

52     // Delai avant le prochain scan (5 secondes)
53     delay(5000);
54 }
```

B Code ESP32 – WiFi sniffing et transmission LoRaWAN

Listing 2 – ESP32 en mode station passive pour le WiFi sniffing

```

1  /*
2  *  ESP32 + LoRa-E5
3  *  Step 2: WiFi sniffing + LoRaWAN transmission (TTN)
4  *  - Scan WiFi access points
5  *  - Build compact JSON payload
6  *  - Send via LoRa-E5 using AT commands
7  *  - JOIN retry mechanism and protection
8  */
9
10 #include <WiFi.h>
11 #include <HardwareSerial.h>
12
13 /* ----- LoRa UART ----- */
14 HardwareSerial LoraSerial(2);      // RX=16, TX=17
15
16 /* ----- TTN parameters ----- */
17 const char *app_eui = "3345979272562857";
18 const char *dev_eui = "70B3D57ED007398E";
19 const char *app_key = "C4663E9406A879F44EE74D93D977A14D";
20
21 /* ----- Configuration ----- */
22 const int MAX_AP = 20;           // Maximum number of APs kept
23                         per scan
23 const int APS_PER_PACKET = 4;    // Number of APs per LoRa
24                         packet
24 const unsigned long TX_INTERVAL = 30000; // Transmission interval (30 s)
25 const unsigned long FRAG_DELAY_MS = 5000; // Delay between fragments (
26                         EU868 friendly)
26 unsigned long last_tx = 0;
27
28 /* ----- JOIN retry configuration ----- */
29 const int JOIN_RETRY_MAX = 5;
30 const unsigned long JOIN_RETRY_DELAY = 5000;
31
32 /* ----- Global state ----- */
33 bool lora_joined = false;
34
35 /* ----- Function prototypes ----- */
36 void send_scan_fragmented();
37 void init_lora_e5();
38 bool join_lora();
```

```

39 | void send_lora_packet(const String &json);
40 | bool send_at(String cmd, int wait = 1000);
41 |
42 | /* ----- SETUP ----- */
43 | void setup() {
44 |   Serial.begin(115200);
45 |   delay(1000);
46 |
47 |   /* WiFi initialization */
48 |   WiFi.mode(WIFI_STA);
49 |   WiFi.disconnect(true);
50 |   delay(100);
51 |
52 |   /* LoRa UART initialization */
53 |   LoraSerial.begin(9600, SERIAL_8N1, 16, 17);
54 |
55 |   Serial.println("\n----\u2022ESP32\u2022WiFi\u2022Sniffer\u2022+\u2022LoRaWAN\u2022----");
56 |
57 |   init_lora_e5();
58 |
59 |   /* JOIN with retry */
60 |   bool joined = false;
61 |   int retry = 0;
62 |
63 |   while (!joined && retry < JOIN_RETRY_MAX) {
64 |     Serial.printf("JOIN\u2022attempt\u2022%d\u2022/\u2022%d\u2022\n", retry + 1, JOIN_RETRY_MAX);
65 |
66 |     joined = join_lora();
67 |
68 |     if (!joined) {
69 |       retry++;
70 |       Serial.println("JOIN\u2022failed,\u2022retrying... ");
71 |       delay(JOIN_RETRY_DELAY);
72 |     }
73 |   }
74 |
75 |   if (!joined) {
76 |     Serial.println("Unable\u2022to\u2022join\u2022LoRaWAN\u2022network");
77 |   } else {
78 |     Serial.println("LoRaWAN\u2022JOIN\u2022success");
79 |     lora_joined = true;
80 |   }
81 |
82 |
83 |   /* ----- LOOP ----- */
84 |   void loop() {
85 |     if (!lora_joined) {
86 |       delay(1000);
87 |       return;

```

```

88     }
89
90     if (millis() - last_tx > TX_INTERVAL) {
91         send_scan_fragmented(); // One scan, multiple packets
92         last_tx = millis();
93     }
94 }
95
96 /* ----- WiFi scan and JSON ----- */
97
98 String json_escape(const String &in) {
99     String out = "";
100    for (size_t i = 0; i < in.length(); i++) {
101        char c = in[i];
102        if (c == '\') out += "\\\"\"";
103        else if (c == '\\') out += "\\\\"";
104        else out += c;
105    }
106    return out;
107 }
108
109 void send_scan_fragmented() {
110     int n = WiFi.scanNetworks(false, true);
111     if (n <= 0) {
112         Serial.println("No access point found");
113         return;
114     }
115
116     n = min(n, MAX_AP);
117
118     unsigned long ts = millis() / 1000;
119     unsigned long sid = millis();
120
121     int total = (n + APS_PER_PACKET - 1) / APS_PER_PACKET;
122
123     for (int p = 0; p < total; p++) {
124         int start = p * APS_PER_PACKET;
125         int end = min(start + APS_PER_PACKET, n);
126
127         String json = "{\"sid\": " + String(sid) +
128                     ", \"i\": " + String(p) +
129                     ", \"n\": " + String(total) +
130                     ", \"t\": " + String(ts) +
131                     ", \"a\": [";
132
133         for (int k = start; k < end; k++) {
134             if (k > start) json += ",";
135
136             String ssid = json_escape(WiFi.SSID(k)); // SSID may be empty

```

```

137
138     json += "[\"";
139     json += WiFi.BSSIDstr(k);
140     json += "\",\" + String(WiFi.RSSI(k));
141     json += "\",\" + String(WiFi.channel(k));
142     json += "\",\" + ssid + \"\"]\"";
143 }
144
145 json += "]}";
146
147 Serial.println("JSON:\u2022" + json);
148 send_lora_packet(json);
149 delay(FRAG_DELAY_MS);
150 }
151
152 WiFi.scanDelete();
153 }
154
155 /* ----- LoRa initialization ----- */
156 void init_lora_e5() {
157     send_at("AT");
158     send_at("AT+MODE=LWOTAA");
159
160     send_at(String("AT+ID=AppEui,\") + app_eui + "\");
161     send_at(String("AT+ID=DevEui,\") + dev_eui + "\");
162     send_at(String("AT+KEY=APPKEY,\") + app_key + "\");
163
164     send_at("AT+DR=EU868"); // Region
165     send_at("AT+DR=5"); // DR5 = SF7 (maximum payload)
166     send_at("AT+CLASS=A");
167 }
168
169 /* ----- JOIN ----- */
170 bool join_lora() {
171     LoraSerial.println("AT+JOIN");
172     Serial.println(">\u2022AT+JOIN");
173
174     unsigned long start = millis();
175     while (millis() - start < 10000) {
176         if (LoraSerial.available()) {
177             String r = LoraSerial.readString();
178             Serial.print(r);
179
180             if (r.indexOf("Network\u2022joined") >= 0) return true;
181             if (r.indexOf("JOIN\u2022failed") >= 0) return false;
182         }
183     }
184     return false;
185 }

```

```

186
187 /* ----- Send packet ----- */
188 void send_lora_packet(const String &json) {
189     String hex = "";
190     for (int i = 0; i < json.length(); i++) {
191         byte c = json[i];
192         if (c < 16) hex += "0";
193         hex += String(c, HEX);
194     }
195     hex.toUpperCase();
196
197     String cmd = "AT+MSGHEX=\\" + hex + "\\";
198     send_at(cmd, 3000);
199 }
200
201 /* ----- AT helper ----- */
202 bool send_at(String cmd, int wait) {
203     LoraSerial.println(cmd);
204     Serial.println(">_ " + cmd);
205     delay(wait);
206
207     while (LoraSerial.available()) {
208         Serial.write(LoraSerial.read());
209     }
210     Serial.println();
211     return true;
212 }
```

C TTN – Uplink Payload Formatter

Listing 3 – Décodeur de payload Uplink (TTN)

```

1 function decodeUplink(input) {
2     let s = "";
3     for (let i = 0; i < input.bytes.length; i++) {
4         s += String.fromCharCode(input.bytes[i]);
5     }
6     try {
7         return { data: JSON.parse(s) };
8     } catch (e) {
9         return { errors: ["Invalid JSON"], data: { raw: s } };
10    }
11 }
```

D Script Python – Réception MQTT et stockage CSV

Listing 4 – Collecte des uplinks TTN via MQTT et archivage au format CSV (service FastAPI)

```
1 import json
```

```

2 import csv
3 import threading
4 import os
5
6 from fastapi import FastAPI
7 import paho.mqtt.client as mqtt
8
9 # TTN / The Things Stack MQTT endpoint (EUI)
10 MQTT_BROKER = "eui.cloud.thethings.network"
11 MQTT_PORT = 1883
12 MQTT_TOPIC = "v3/+devices/+up"
13
14 # Credentials (do not hardcode secrets in production)
15 MQTT_USERNAME = os.getenv("TTN_MQTT_USERNAME", "iottp2@ttn")
16 MQTT_PASSWORD = os.getenv("TTN_MQTT_PASSWORD", "REPLACE_WITH_API_KEY")
17
18 CSV_FILE = "wifi_sniffing_data.csv"
19
20 app = FastAPI()
21
22 # Initialize CSV file with header if missing
23 if not os.path.exists(CSV_FILE):
24     with open(CSV_FILE, "w", newline="", encoding="utf-8") as f:
25         csv.writer(f).writerow([
26             "received_at", "device_id",
27             "sid", "frag_i", "frag_n", "esp_t",
28             "ap_index", "bssid", "rssi", "channel", "ssid"
29         ])
30
31 def on_message(client, userdata, msg):
32     # Parse TTN uplink JSON message
33     uplink = json.loads(msg.payload.decode("utf-8"))
34
35     device_id = uplink["end_device_ids"]["device_id"]
36     received_at = uplink["received_at"]
37     decoded = uplink.get("uplink_message", {}).get("decoded_payload", {})
38
39     # Fragment metadata (sid = scan id, i/n = fragment index/total)
40     sid = decoded.get("sid")
41     frag_i = decoded.get("i")
42     frag_n = decoded.get("n")
43     esp_t = decoded.get("t")
44     aps = decoded.get("a", [])
45
46     # Build CSV rows (one row per detected AP)
47     rows = []
48     for idx, ap in enumerate(aps, start=1):
49         # AP format: [bssid, rssi, channel, ssid]
50         if isinstance(ap, list) and len(ap) >= 3:

```

```

51     bssid = ap[0]
52     rssi = ap[1]
53     ch = ap[2]
54     ssid = ap[3] if len(ap) >= 4 else ""
55     rows.append([
56         received_at, device_id,
57         sid, frag_i, frag_n, esp_t,
58         idx, bssid, rssi, ch, ssid
59     ])
60
61 # Append to CSV
62 if rows:
63     with open(CSV_FILE, "a", newline="", encoding="utf-8") as f:
64         csv.writer(f).writerows(rows)
65
66 print(f"device_id:{sid} frag={frag_i}/{frag_n} saved:{len(rows)}\nAP(s)")
67
68 def mqtt_loop():
69     client = mqtt.Client()
70     client.username_pw_set(MQTT_USERNAME, MQTT_PASSWORD)
71     client.on_message = on_message
72
73     client.connect(MQTT_BROKER, MQTT_PORT, 60)
74     client.subscribe(MQTT_TOPIC)
75     client.loop_forever()
76
77 # Run MQTT client in a background thread
78 threading.Thread(target=mqtt_loop, daemon=True).start()
79
80 @app.get("/")
81 def root():
82     # Minimal status endpoint (for monitoring)
83     return {"status": "TTN\u2022MQTT\u2022listener\u2022running", "csv": CSV_FILE}

```

E Script Python – Construction de la base AP via Wigle

Listing 5 – Construction de la base AP via Wigle

```

1 import time
2 import os
3 import pandas as pd
4 import requests
5 from requests.auth import HTTPBasicAuth
6
7 RAW_CSV = "wifi_sniffing_data.csv"
8 AP_DB_CSV = "ap_db_wigle.csv"
9
10 API_NAME = "AID..."

```

```

11 API_TOKEN = "..."
12 WIGLE_URL = "https://api.wigle.net/api/v2/network/search"
13 WIGLE_SLEEP_SEC = 1.0
14
15 # Geographic bounding box for the Paris area
16 PARIS_MIN_LAT = 48.80
17 PARIS_MAX_LAT = 48.90
18 PARIS_MIN_LON = 2.25
19 PARIS_MAX_LON = 2.42
20
21 # Number of processed entries before saving to disk
22 # (1 = safest, higher values = faster)
23 SAVE_EVERY_N = 1
24
25 def normalize_bssid(x):
26     if pd.isna(x):
27         return None
28     x = str(x).strip().upper().replace("-", ":")
29     if ":" not in x and len(x) == 12:
30         x = ":".join([x[i:i+2] for i in range(0, 12, 2)])
31     return x
32
33 def safe_save_csv(df: pd.DataFrame, path: str):
34     """Atomic CSV write: write to a temporary file, then replace the
35     original file
36     to avoid corruption in case of interruption."""
37     tmp_path = path + ".tmp"
38     df.to_csv(tmp_path, index=False)
39     os.replace(tmp_path, path)
40
41 # 1) Extract unique (bssid, ssid) pairs for API queries
42 df = pd.read_csv(RAW_CSV)
43 df["bssid"] = df["bssid"].apply(normalize_bssid)
44 df = df.dropna(subset=["bssid"])
45 df["ssid"] = df["ssid"].astype(str).fillna("").str.strip()
46 ssid_pair_set = set(df[["bssid", "ssid"]].apply(tuple, axis=1))
47 ssid_pair_list = sorted(list(ssid_pair_set))
48
49 # 2) Load existing AP database to avoid duplicate queries
50 columns = ["bssid", "ssid", "lat", "lon", "qos", "lasttime", "source"]
51 try:
52     ap_db = pd.read_csv(AP_DB_CSV)
53     for c in columns:
54         if c not in ap_db.columns:
55             ap_db[c] = pd.NA
56     ap_db = ap_db[columns]
57     ap_db["bssid"] = ap_db["bssid"].apply(normalize_bssid)
58     ap_db["ssid"] = ap_db["ssid"].astype(str).fillna("").str.strip()
59 except FileNotFoundError:

```

```

59     ap_db = pd.DataFrame(columns=columns)
60
61 # Already known (bssid, ssid) pairs
62 known = set(ap_db[["bssid", "ssid"]].apply(tuple, axis=1))
63 to_query = [pair for pair in ssid_pair_list if pair not in known]
64
65 def wigle_lookup(bssid, ssid):
66     print(f"Querying Wigle for {bssid} with SSID={ssid} (Paris area only")
67     print(f"    ...")
68     try:
69         params = {
70             "netid": bssid,
71             "latrange1": PARIS_MIN_LAT,
72             "latrange2": PARIS_MAX_LAT,
73             "longrange1": PARIS_MIN_LON,
74             "longrange2": PARIS_MAX_LON
75         }
76         # Add SSID constraint only when a valid name is available
77         if ssid and ssid.lower() not in ["none", "nan", ""]:
78             params["ssid"] = ssid
79         r = requests.get(
80             WIGLE_URL,
81             params=params,
82             auth=HTTPBasicAuth(API_NAME, API_TOKEN),
83             timeout=15
84         )
85     except requests.RequestException as e:
86         print(f"Request error for {bssid}/{ssid}: {e}")
87         return None
88
89     if r.status_code == 429:
90         print(f"Rate limited (429) for {bssid}; consider increasing"
91             f" WIGLE_SLEEP_SEC")
92         return None
93
94     try:
95         r.raise_for_status()
96     except Exception as e:
97         print(f"HTTP error for {bssid}: {e} (status {r.status_code})")
98         print(r.text[:500])
99         return None
100
101     try:
102         js = r.json()
103     except ValueError as e:
104         print(f"Failed to parse JSON for {bssid}: {e}; response snippet:{r.text[:500]}")
105         return None

```

```

105     # Debug output (can be disabled for production use)
106     print(js)
107     if not js.get("success"):
108         return None
109
110     # First, filter results by exact SSID match (case-insensitive)
111     results = js.get("results", [])
112     filtered = [
113         r for r in results
114         if str(r.get("ssid", "")).strip().lower() == ssid.strip().lower()
115     ]
116
117     # Keep only results located inside the Paris bounding box
118     filtered_in_paris = []
119     for r in filtered:
120         try:
121             lat = float(r.get("trilat", 0))
122             lon = float(r.get("trilong", 0))
123             if PARIS_MIN_LAT <= lat <= PARIS_MAX_LAT and PARIS_MIN_LON <=
124                 lon <= PARIS_MAX_LON:
125                 filtered_in_paris.append(r)
126         except Exception:
127             continue
128
129     if filtered_in_paris:
130         return filtered_in_paris[0]
131     elif filtered:
132         # Fallback: SSID matches but outside the Paris area
133         return filtered[0]
134
135     # No SSID match: keep only results inside the Paris area
136     results_in_paris = []
137     for r in results:
138         try:
139             lat = float(r.get("trilat", 0))
140             lon = float(r.get("trilong", 0))
141             if PARIS_MIN_LAT <= lat <= PARIS_MAX_LAT and PARIS_MIN_LON <=
142                 lon <= PARIS_MAX_LON:
143                 results_in_paris.append(r)
144         except Exception:
145             continue
146
147     if results_in_paris:
148         return results_in_paris[0]
149     elif results:
150         # Final fallback: return the first available result
151         return results[0]
152     else:
153         return None

```

```

152
153     saved_since_last = 0
154
155     for idx, (bssid, ssid) in enumerate(to_query, 1):
156         try:
157             res = wigle_lookup(bssid, ssid)
158             if res:
159                 row = {
160                     "bssid": bssid,
161                     "ssid": res.get("ssid"),
162                     "lat": res.get("trilat"),
163                     "lon": res.get("trilong"),
164                     "qos": res.get("qos"),
165                     "lasttime": res.get("lasttime"),
166                     "source": "wigle"
167                 }
168             else:
169                 row = {"bssid": bssid, "ssid": ssid, "source": "wigle_notfound"}
170
171         except Exception as e:
172             row = {"bssid": bssid, "ssid": ssid, "source": "wigle_error"}
173             print(f"Error querying {bssid}/{ssid}: {e}")
174
175         ap_db = pd.concat([ap_db, pd.DataFrame([row])], ignore_index=True)
176
177         # Deduplication: keep only the most recent entry for each (bssid, ssid)
178         # pair
179         ap_db = ap_db.drop_duplicates(subset=["bssid", "ssid"], keep="last")
180
181         # Keep only access points with valid coordinates
182         ap_db_saved = ap_db.dropna(subset=["lat", "lon"])
183
184         saved_since_last += 1
185
186         if saved_since_last >= SAVE_EVERY_N:
187             time.sleep(WIGLE_SLEEP_SEC)
188             safe_save_csv(ap_db_saved, AP_DB_CSV)
189             print(f"Saved {len(ap_db_saved)} rows to {AP_DB_CSV} (processed {idx}/{len(to_query)})")
190             saved_since_last = 0
191
192         # Final save after processing all entries
193         safe_save_csv(ap_db.dropna(subset=["lat", "lon"]), AP_DB_CSV)
194         print("AP database ready:", AP_DB_CSV)

```

F Script Python – Génération d'une base AP simulée

Listing 6 – Génération d'une base de données AP simulée

```

1 import os
2 import math
3 import pandas as pd
4 import random
5 import re
6
7 RAW_CSV = "wifi_sniffing_data.csv"
8 AP_DB_CSV = "ap_db_fake.csv"
9
10 # Reference location used as the center of synthetic coordinates
11 CENTER_LAT = 48.844953058524794
12 CENTER_LON = 2.3569938188665716
13
14 # Radius of synthetic point distribution in meters
15 # 50--60 m roughly corresponds to the same building or nearby area
16 RADIUS_M = 60
17
18 # Fixed random seed to ensure reproducible results
19 SEED = 20260103
20
21 # Regular expression for validating MAC address format
22 MAC_RE = re.compile(r"^(?:[0-9A-F]{2}:){5}[0-9A-F]{2}$")
23
24 def normalize_bssid(x):
25     # Normalize BSSID format to uppercase colon-separated MAC address
26     if pd.isna(x):
27         return None
28     x = str(x).strip().upper().replace("-", ":").replace(":", "")
29     if ":" not in x and len(x) == 12 and all(c in "0123456789ABCDEF" for c
30         in x):
31         x = ":".join([x[i:i+2] for i in range(0, 12, 2)])
32     return x if MAC_RE.match(x) else None
33
34 def random_point_near(lat, lon, radius_m, rng: random.Random):
35     """
36     Generate a random point within radius_m around (lat, lon).
37     This approximation is sufficient for small-scale local experiments.
38     The sqrt term ensures uniform distribution over the area.
39     """
40     r = radius_m * math.sqrt(rng.random())
41     theta = 2 * math.pi * rng.random()
42
43     # Approximate conversion:
44     # 1 degree latitude is about 111320 meters
45     # Longitude scaling depends on latitude
46     dlat = (r * math.cos(theta)) / 111320.0
47     dlon = (r * math.sin(theta)) / (111320.0 * math.cos(math.radians(lat)))

```

```

48     return lat + dlat, lon + dlon
49
50 def safe_save_csv(df: pd.DataFrame, path: str):
51     # Atomic CSV write to avoid partial file corruption
52     tmp = path + ".tmp"
53     df.to_csv(tmp, index=False)
54     os.replace(tmp, path)
55
56 def main():
57     rng = random.Random(SEED)
58
59     df = pd.read_csv(RAW_CSV)
60     if "bssid" not in df.columns:
61         raise ValueError("CSV must contain a 'bssid' column.")
62
63     df["bssid"] = df["bssid"].apply(normalize_bssid)
64     df = df.dropna(subset=["bssid"])
65     bssids = sorted(df["bssid"].unique().tolist())
66     print("Unique BSSIDs found:", len(bssids))
67
68     # Load existing AP database (incremental update)
69     columns = ["bssid", "lat", "lon", "source"]
70     try:
71         ap_db = pd.read_csv(AP_DB_CSV)
72         for c in columns:
73             if c not in ap_db.columns:
74                 ap_db[c] = pd.NA
75         ap_db = ap_db[columns]
76         ap_db["bssid"] = ap_db["bssid"].apply(normalize_bssid)
77         ap_db = ap_db.dropna(subset=["bssid"]).drop_duplicates(
78             subset=["bssid"], keep="last")
79     )
80     except FileNotFoundError:
81         ap_db = pd.DataFrame(columns=columns)
82
83     known = set(ap_db["bssid"].tolist())
84     to_add = [b for b in bssids if b not in known]
85     print("New BSSIDs to generate:", len(to_add))
86
87     rows = []
88     for bssid in to_add:
89         lat, lon = random_point_near(CENTER_LAT, CENTER_LON, RADIUS_M, rng
90             )
91         rows.append({
92             "bssid": bssid,
93             "lat": lat,
94             "lon": lon,
95             "source": (

```

```

96             f"\{CENTER_LON:.6f}\_{r}\{RADIUS_M}\_m\_seed_{SEED}""
97         )
98     })
99
100    if rows:
101        ap_db = pd.concat([ap_db, pd.DataFrame(rows)], ignore_index=True)
102        ap_db = ap_db.drop_duplicates(subset=["bssid"], keep="last")
103
104        safe_save_csv(ap_db, AP_DB_CSV)
105        print("Wrote:", AP_DB_CSV, "rows:", len(ap_db))
106        print("Center:", CENTER_LAT, CENTER_LON,
107              "Radius(m):", RADIUS_M, "Seed:", SEED)
108
109    if __name__ == "__main__":
110        main()

```

G Script Python – Génération de trames uplink simulées

Listing 7 – Simulation de TTN uplink messages

```

1 import csv
2 import json
3 import math
4 import random
5 from datetime import datetime, timezone
6 from typing import List, Dict, Tuple
7
8 # -----
9 # User configuration
10 # -----
11 CENTER_LAT = 48.844953058524794
12 CENTER_LON = 2.3569938188665716
13 RADIUS_M = 60.0
14
15 AP_DB_CSV = "ap_db_fake.csv"
16
17 DEVICE_ID = "esp32-sniffer-sim"
18 MAX_AP = 6                      # Should match MAX_AP on the ESP side
19 DROP_RSSI_BELOW = -92            # Discard very weak signals (tunable)
20 CHANNELS = [1, 6, 11]           # Simplified channel selection
21
22 # RSSI model parameters (tuned to resemble real measurements)
23 RSSIO_DBM = -45.0                # Reference RSSI at 1 meter
24 N_PATHLOSS = 2.3                 # Typical indoor value: 2.2 to 3.5
25 NOISE_SIGMA = 2.5                # Noise standard deviation in dB
26
27 # -----
28 # Latitude / longitude utilities
29 # -----

```

```

30 EARTH_R = 6378137.0
31
32 def xy_to_ll(x: float, y: float, lat0: float, lon0: float) -> Tuple[float,
33     float]:
33     phi0 = math.radians(lat0)
34     lam0 = math.radians(lon0)
35     phi = y / EARTH_R + phi0
36     lam = x / (EARTH_R * math.cos((phi + phi0) / 2.0)) + lam0
37     return math.degrees(phi), math.degrees(lam)
38
39 def haversine_m(lat1, lon1, lat2, lon2) -> float:
40     p1, p2 = math.radians(lat1), math.radians(lat2)
41     dphi = p2 - p1
42     dl = math.radians(lon2 - lon1)
43     a = math.sin(dphi / 2) ** 2 + math.cos(p1) * math.cos(p2) * math.sin(
44         dl / 2) ** 2
45     return 2 * EARTH_R * math.asin(math.sqrt(a))
46
47 # -----
47 # RSSI simulation: log-distance model with noise
48 # -----
49 def rssи_from_distance(d_m: float,
50                         rssи0_dbm: float = RSSI0_DBM,
51                         d0_m: float = 1.0,
52                         n: float = N_PATHLOSS,
53                         noise_sigma: float = NOISE_SIGMA) -> float:
54     d = max(d_m, 0.3)
55     mean = rssи0_dbm - 10.0 * n * math.log10(d / d0_m)
56     return mean + random.gauss(0.0, noise_sigma)
57
58 # -----
59 # Load access point database
60 # ap_db_fake.csv columns: bssid, lat, lon, source
61 # -----
62 def load_ap_db(path: str) -> List[Dict]:
63     aps = []
64     with open(path, "r", encoding="utf-8", newline="") as f:
65         reader = csv.DictReader(f)
66         for row in reader:
67             # Handle case differences and extra spaces
68             bssid = (row.get("bssid") or "").strip()
69             lat = float(row["lat"])
70             lon = float(row["lon"])
71             source = (row.get("source") or "").strip()
72             if bssid:
73                 aps.append({
74                     "bssid": bssid,
75                     "lat": lat,
76                     "lon": lon,

```

```

77         "source": source
78     })
79     return aps
80
81 # -----
82 # Generate a random true device location near the center
83 #
84 def random_point_near_center(radius_m: float = RADIUS_M) -> Tuple[float,
85     float]:
86     r = radius_m * math.sqrt(random.random())
87     ang = 2 * math.pi * random.random()
88     x = r * math.cos(ang)
89     y = r * math.sin(ang)
90     return xy_to_ll(x, y, CENTER_LAT, CENTER_LON)
91
92 # -----
93 # Generate one simulated scan result (decoded_payload field "a")
94 # a format: [[bssid, rssi, channel, ssid], ...]
95 # SSID is not available in the database; source is used as a placeholder
96 #
97 def simulate_scan_payload(aps: List[Dict],
98                           true_lat: float,
99                           true_lon: float,
100                          max_ap: int = MAX_AP) -> List[List]:
101     candidates = []
102     for ap in aps:
103         d = haversine_m(true_lat, true_lon, ap["lat"], ap["lon"])
104         rssi = rssi_from_distance(d)
105         if rssi < DROP_RSSI_BELOW:
106             continue
107         bssid = ap["bssid"]
108         ch = random.choice(CHANNELS)
109         ssid = ap["source"] # Placeholder value if SSID is unknown
110         candidates.append([bssid, int(round(rssi)), ch, ssid])
111
112     # Sort by RSSI (strongest first) and keep top max_ap entries
113     candidates.sort(key=lambda x: x[1], reverse=True)
114     return candidates[:max_ap]
115
116 # -----
117 # Build a TTN-style uplink JSON object
118 # Compatible with the on_message() handler
119 #
120 def build_uplink_json(decoded_payload: Dict,
121                         device_id: str = DEVICE_ID) -> Dict:
122     now = datetime.now(timezone.utc).isoformat().replace("+00:00", "Z")
123     return {
124         "received_at": now,
125         "end_device_ids": {"device_id": device_id},

```

```

125         "uplink_message": {"decoded_payload": decoded_payload}
126     }
127
128 # -----
129 # Demo: generate N simulated uplinks and write them to a file
130 #
131 def main(n_messages: int = 5, out_jsonl: str = "sim_uplinks.jsonl"):
132     random.seed(7)
133     aps = load_ap_db(AP_DB_CSV)
134
135     with open(out_jsonl, "w", encoding="utf-8") as f:
136         for k in range(n_messages):
137             true_lat, true_lon = random_point_near_center(RADIUS_M)
138
139             # Generate one simulated scan
140             a_list = simulate_scan_payload(aps, true_lat, true_lon, MAX_AP)
141
142             decoded = {
143                 "sid": f"sim_{k+1:04d}", # Session identifier
144                 "i": 1, # Fragment index
145                 "n": 1, # Total number of fragments
146                 "t": int(time.time()) if "time" in globals() else 0,
147                 "a": a_list
148             }
149
150             uplink = build_uplink_json(decoded, DEVICE_ID)
151             f.write(json.dumps(uplink, ensure_ascii=False) + "\n")
152
153     print(f"Wrote {n_messages} simulated uplinks to {out_jsonl}")
154
155 if __name__ == "__main__":
156     import time
157     main(n_messages=20)

```

H Script Python – Estimation de position et comparaison des méthodes

Listing 8 – Comparaison des méthodes de localisation (multilatération, centroïde pondéré et KNN)

```

1 import json, csv, math, random
2 from dataclasses import dataclass
3 from typing import Dict, List, Tuple, Optional
4
5 import numpy as np
6 import pandas as pd
7
8 #
# Parameters to tune for your scenario

```

```

10 # -----
11 AP_DB_CSV = "ap_db_fake.csv"
12 UPLINKS_JSONL = "sim_uplinks.jsonl"
13
14 # Log-distance model parameters for RSSI to distance
15 # Matching the simulation parameters improves consistency
16 RSSIO_DBM = -45.0      # Reference RSSI at 1 meter (same as in the simulator
17 )                      # Path-loss exponent (same as in the simulator)
18 DO_M = 1.0
19
20 # KNN fingerprinting training set settings
21 # The training set is generated synthetically by sampling locations and
22 # simulating RSSI vectors
23 TRAIN_SAMPLES = 2000
24 TRAIN_RADIUS_M = 100.0      # Use the same scale as the synthetic DB radius
25 KNN_K = 5                  # Smaller K, then use weighted averaging
26 MISSING_RSSI = -100.0      # Fill missing APs with -100 dBm (very weak or
27 # not visible)
28 RSSI_NOISE_SIGMA = 1.5      # Add noise to training data to reduce
29 # overfitting
30
31 # -----
32 # Geographic conversions (small area: lat/lon <-> local plane)
33 # -----
34 EARTH_R = 6378137.0
35
36 def ll_to_xy_m(lat: float, lon: float, lat0: float, lon0: float) -> Tuple[
37     float, float]:
38     phi = math.radians(lat)
39     phi0 = math.radians(lat0)
40     lam = math.radians(lon)
41     lam0 = math.radians(lon0)
42     x = (lam - lam0) * math.cos((phi + phi0) / 2.0) * EARTH_R
43     y = (phi - phi0) * EARTH_R
44     return x, y
45
46 def xy_to_ll(x: float, y: float, lat0: float, lon0: float) -> Tuple[float,
47     float]:
48     phi0 = math.radians(lat0)
49     lam0 = math.radians(lon0)
50     phi = y / EARTH_R + phi0
51     lam = x / (EARTH_R * math.cos((phi + phi0) / 2.0)) + lam0
52     return math.degrees(phi), math.degrees(lam)
53
54 # -----

```

```

53 # RSSI <-> distance (log-distance model)
54 # -----
55 def distance_from_rssi(rssi_dbm: float,
56                         rssi0_dbm: float = RSSIO_DBM,
57                         d0_m: float = D0_M,
58                         n: float = N_PATHLOSS) -> float:
59     return d0_m * (10.0 ** ((rssi0_dbm - rssi_dbm) / (10.0 * n)))
60
61 def rssi_from_distance(d_m: float,
62                         rssi0_dbm: float = RSSIO_DBM,
63                         d0_m: float = D0_M,
64                         n: float = N_PATHLOSS,
65                         noise_sigma: float = 0.0) -> float:
66     d = max(d_m, 0.3)
67     base = rssi0_dbm - 10.0 * n * math.log10(d / d0_m)
68     if noise_sigma > 0:
69         return base + random.gauss(0.0, noise_sigma)
70     return base
71
72 # -----
73 # Data structures
74 # -----
75 @dataclass
76 class AP:
77     bssid: str
78     lat: float
79     lon: float
80     source: str
81
82 @dataclass
83 class Obs:
84     bssid: str
85     rssi: float
86
87 # -----
88 # File loading helpers
89 # -----
90 def load_ap_db(path: str) -> Dict[str, AP]:
91     out = {}
92     with open(path, "r", encoding="utf-8", newline="") as f:
93         r = csv.DictReader(f)
94         for row in r:
95             bssid = (row.get("bssid") or "").strip()
96             if not bssid:
97                 continue
98             out[bssid] = AP(
99                 bssid=bssid,
100                lat=float(row["lat"]),
101               lon=float(row["lon"]),

```

```

102             source=(row.get("source") or "").strip(),
103         )
104     return out
105
106 def iter_uplinks(jsonl_path: str):
107     with open(jsonl_path, "r", encoding="utf-8") as f:
108         for line in f:
109             line = line.strip()
110             if not line:
111                 continue
112             yield json.loads(line)
113
114 def get_obs_from_uplink(u: dict) -> Tuple[str, List[Obs]]:
115     decoded = u.get("uplink_message", {}).get("decoded_payload", {})
116     sid = str(decoded.get("sid", ""))
117     a = decoded.get("a", []) or []
118     obs = []
119     for item in a:
120         # item format: [bssid, rssi, channel, ssid]
121         if isinstance(item, list) and len(item) >= 2:
122             obs.append(Obs(bssid=str(item[0]).strip(), rssi=float(item[1])))
123
124     # Sort observations by RSSI (strongest first)
125     obs.sort(key=lambda x: x.rssi, reverse=True)
126     return sid, obs
127
128 # -----
129 # Method 2: RSSI-weighted centroid
130 # -----
131 def estimate_weighted_centroid(obs: List[Obs], ap_map: Dict[str, AP],
132                                 lat0: float, lon0: float) -> Optional[Tuple[float, float]]:
133     xs, ys, ws = [], [], []
134     for o in obs:
135         ap = ap_map.get(o.bssid)
136         if not ap:
137             continue
138         x, y = ll_to_xy_m(ap.lat, ap.lon, lat0, lon0)
139         # Higher RSSI => larger weight (convert dBm to linear power,
140         # shifted to avoid tiny values)
141         w = 10.0 ** ((o.rssi + 100.0) / 10.0)
142         xs.append(x); ys.append(y); ws.append(w)
143
144     if len(ws) < 1:
145         return None
146     x_hat = float(np.average(xs, weights=ws))
147     y_hat = float(np.average(ys, weights=ws))
148     return xy_to_ll(x_hat, y_hat, lat0, lon0)

```

```

148 # -----
149 # Method 1: multilateration (Gauss-Newton least squares)
150 # Convert RSSI to distance, then fit position by minimizing range
151 # residuals
152 # -----
153 def estimate_multilateration(obs: List[Obs], ap_map: Dict[str, AP],
154                               lat0: float, lon0: float,
155                               max_iter: int = 30) -> Tuple[Optional[Tuple[
156                                   float, float]], Optional[float]]:
157     pts = []
158     ds = []
159     for o in obs:
160         ap = ap_map.get(o.bssid)
161         if not ap:
162             continue
163         xi, yi = ll_to_xy_m(ap.lat, ap.lon, lat0, lon0)
164         di = distance_from_rssi(o.rssi)
165         pts.append((xi, yi))
166         ds.append(di)
167
168     if len(ds) < 3:
169         return None, None
170
171     pts = np.array(pts, dtype=float)    # (k, 2)
172     ds = np.array(ds, dtype=float)    # (k,)
173
174     # Initialization: geometric center of AP positions
175     x = float(pts[:, 0].mean())
176     y = float(pts[:, 1].mean())
177
178     for _ in range(max_iter):
179         dx = x - pts[:, 0]
180         dy = y - pts[:, 1]
181         ri = np.sqrt(dx * dx + dy * dy)
182         ri = np.maximum(ri, 1e-6)
183
184         f = (ri - ds).reshape(-1, 1)    # residuals
185         J = np.column_stack((dx / ri, dy / ri))
186
187         A = J.T @ J + 1e-3 * np.eye(2)    # damping
188         b = -J.T @ f
189
190         delta = np.linalg.solve(A, b).flatten()
191         x += float(delta[0])
192         y += float(delta[1])
193
194         if float(np.linalg.norm(delta)) < 1e-3:
195             break

```

```

195     # RMS residual in meters (smaller means more self-consistent)
196     dx = x - pts[:, 0]
197     dy = y - pts[:, 1]
198     ri = np.sqrt(dx * dx + dy * dy)
199     rms = float(np.sqrt(np.mean((ri - ds) ** 2)))
200
201     return xy_to_ll(x, y, lat0, lon0), rms
202
203 # -----
204 # Method 3: KNN fingerprinting
205 # Training set: sample positions and generate RSSI vectors using the model
206 # (synthetic fingerprint DB)
207 # Inference: nearest neighbors in RSSI space, then weighted average of
208 # their positions
209 # -----
210
211 def build_ap_list_for_vectors(ap_map: Dict[str, AP], lat0: float, lon0: float) -> List[str]:
212     # AP selection strategy: prefer APs that are well distributed within
213     # the area
214     # Favor medium-distance APs to avoid extremely close or extremely far
215     # ones
216     items = []
217     for bssid, ap in ap_map.items():
218         x, y = ll_to_xy_m(ap.lat, ap.lon, lat0, lon0)
219         d = math.sqrt(x * x + y * y)
220         # Prefer APs around 50 m (more uniform coverage inside the
221         # training region)
222         score = -abs(d - 50.0)
223         items.append((score, bssid))
224     items.sort(key=lambda t: t[0], reverse=True)
225     return [b for _, b in items[:TOP_AP_FOR_VEC]]
226
227
228 def random_point_in_radius(lat0: float, lon0: float, radius_m: float) ->
229     Tuple[float, float]:
230     r = radius_m * math.sqrt(random.random())
231     ang = 2 * math.pi * random.random()
232     x = r * math.cos(ang)
233     y = r * math.sin(ang)
234     return xy_to_ll(x, y, lat0, lon0)
235
236
237 def build_fingerprint_training(ap_map: Dict[str, AP], lat0: float, lon0: float,
238                                 ap_list: List[str],
239                                 n_samples: int = TRAIN_SAMPLES,
240                                 radius_m: float = TRAIN_RADIUS_M,
241                                 seed: int = 7) -> Tuple[np.ndarray, np.ndarray]:
242     random.seed(seed)
243     X = np.zeros((n_samples, len(ap_list)), dtype=float)

```

```

235     Y = np.zeros((n_samples, 2), dtype=float) # xy meters
236
237     for i in range(n_samples):
238         lat, lon = random_point_in_radius(lat0, lon0, radius_m)
239         x, y = ll_to_xy_m(lat, lon, lat0, lon0)
240         Y[i] = [x, y]
241
242         for j, bssid in enumerate(ap_list):
243             ap = ap_map[bssid]
244             ax, ay = ll_to_xy_m(ap.lat, ap.lon, lat0, lon0)
245             d = math.sqrt((x - ax) ** 2 + (y - ay) ** 2)
246             # Add noise to match the test distribution
247             base_rssi = rssi_from_distance(d, noise_sigma=0.0)
248             X[i, j] = base_rssi + random.gauss(0.0, RSSI_NOISE_SIGMA)
249
250     return X, Y
251
252 def obs_to_vector(obs: List[Obs], ap_list: List[str]) -> np.ndarray:
253     m = {o.bssid: o.rssi for o in obs}
254     v = np.array([m.get(b, MISSING_RSSI) for b in ap_list], dtype=float)
255     return v
256
257 def estimate_knn(obs: List[Obs], ap_list: List[str],
258                   X_train: np.ndarray, Y_train: np.ndarray,
259                   lat0: float, lon0: float,
260                   k: int = KNN_K) -> Optional[Tuple[float, float]]:
261     if X_train.shape[0] < k:
262         return None
263     v = obs_to_vector(obs, ap_list)
264
265     # Weighted Euclidean distance for RSSI vectors
266     # Reduce the weight of missing values
267     diff = X_train - v.reshape(1, -1)
268     weights = np.ones_like(diff)
269     missing_mask = (v == MISSING_RSSI) | (X_train == MISSING_RSSI)
270     weights[missing_mask] = 0.1
271
272     d2 = np.sum(weights * diff * diff, axis=1)
273
274     # Select the k nearest neighbors
275     idx = np.argpartition(d2, k)[:k]
276     distances = np.sqrt(d2[idx])
277
278     # Distance-weighted average of neighbor positions (avoid division by
279     # zero)
280     inv_dist = 1.0 / (distances + 1e-6)
281     weights_k = inv_dist / np.sum(inv_dist)
282
283     xy_hat = np.average(Y_train[idx], axis=0, weights=weights_k)

```

```

283     return xy_to_ll(float(xy_hat[0]), float(xy_hat[1]), lat0, lon0)
284
285 # -----
286 # Evaluation: self-consistency checks
287 # For multilateration: range residual RMS (meters) is already computed
288 # For centroid and KNN: compute RSSI RMSE (dB) between observed and
289 # predicted RSSI at the estimate
290 # -----
291 def rsssi_rmse_at_position(est_ll: Tuple[float, float], obs: List[Obs],
292                             ap_map: Dict[str, AP], lat0: float, lon0: float)
293                             -> Optional[float]:
294
295     lat, lon = est_ll
296     x, y = ll_to_xy_m(lat, lon, lat0, lon0)
297     errs = []
298     for o in obs:
299         ap = ap_map.get(o.bssid)
300         if not ap:
301             continue
302         ax, ay = ll_to_xy_m(ap.lat, ap.lon, lat0, lon0)
303         d = math.sqrt((x - ax) ** 2 + (y - ay) ** 2)
304         pred = rsssi_from_distance(d)
305         errs.append((pred - o.rssi) ** 2)
306     if not errs:
307         return None
308     return float(math.sqrt(sum(errs) / len(errs)))
309
310 def main():
311     ap_map = load_ap_db(AP_DB_CSV)
312     if not ap_map:
313         raise RuntimeError("AP_DB is empty. Check ap_db_fake.csv")
314
315     # Reference origin: use the mean AP position (more stable than a fixed
316     # center)
317     lat0 = float(np.mean([ap.lat for ap in ap_map.values()]))
318     lon0 = float(np.mean([ap.lon for ap in ap_map.values()]))
319
320     # KNN training
321     ap_list = build_ap_list_for_vectors(ap_map, lat0, lon0)
322     X_train, Y_train = build_fingerprint_training(ap_map, lat0, lon0,
323                                                 ap_list)
324
325     rows = []
326     for u in iter_uplinks(UPLINKS_JSONL):
327         sid, obs = get_obs_from_uplink(u)
328         if len(obs) < 1:
329             continue
330
331         # Method: weighted centroid
332         est_wc = estimate_weighted_centroid(obs, ap_map, lat0, lon0)

```

```

328     wc_rmse = rssi_rmse_at_position(est_wc, obs, ap_map, lat0, lon0)
329         if est_wc else None
330
330     # Method: multilateration
331     est_ml, ml_rms_m = estimate_multilateration(obs, ap_map, lat0,
332         lon0)
332     ml_rmse = rssi_rmse_at_position(est_ml, obs, ap_map, lat0, lon0)
333         if est_ml else None
333
334     # Method: KNN fingerprinting
335     est_knn = estimate_knn(obs, ap_list, X_train, Y_train, lat0, lon0,
336         k=KNN_K)
336     knn_rmse = rssi_rmse_at_position(est_knn, obs, ap_map, lat0, lon0)
337         if est_knn else None
337
338     rows.append({
339         "sid": sid,
340         "n_obs": len(obs),
341
342         "wc_lat": est_wc[0] if est_wc else None,
343         "wc_lon": est_wc[1] if est_wc else None,
344         "wc_rssi_rmse_db": wc_rmse,
345
346         "ml_lat": est_ml[0] if est_ml else None,
347         "ml_lon": est_ml[1] if est_ml else None,
348         "ml_range_rms_m": ml_rms_m,
349         "ml_rssi_rmse_db": ml_rmse,
350
351         "knn_lat": est_knn[0] if est_knn else None,
352         "knn_lon": est_knn[1] if est_knn else None,
353         "knn_rssi_rmse_db": knn_rmse,
354     })
355
356     df = pd.DataFrame(rows)
357
358     # Add aggregate comparison fields
359     def safe_mean(s):
360         s = pd.to_numeric(s, errors="coerce")
361         return float(s.dropna().mean()) if s.notna().any() else None
362
363     # Determine the best method per row (based on RSSI RMSE, lower is
364     # better)
364     def get_best_method(row):
365         methods = []
366         if pd.notna(row["wc_rssi_rmse_db"]):
367             methods.append(("WC", row["wc_rssi_rmse_db"]))
368         if pd.notna(row["ml_rssi_rmse_db"]):
369             methods.append(("ML", row["ml_rssi_rmse_db"]))
370         if pd.notna(row["knn_rssi_rmse_db"]):

```

```

371         methods.append(("KNN", row["knn_rssi_rmse_db"]))
372     if not methods:
373         return None
374     return min(methods, key=lambda x: x[1])[0]
375
376 # Compute ranks per row (based on RSSI RMSE, lower is better)
377 def get_rank(row):
378     methods = []
379     if pd.notna(row["wc_rssi_rmse_db"]):
380         methods.append(("WC", row["wc_rssi_rmse_db"]))
381     if pd.notna(row["ml_rssi_rmse_db"]):
382         methods.append(("ML", row["ml_rssi_rmse_db"]))
383     if pd.notna(row["knn_rssi_rmse_db"]):
384         methods.append(("KNN", row["knn_rssi_rmse_db"]))
385     if not methods:
386         return None
387     sorted_methods = sorted(methods, key=lambda x: x[1])
388     ranks = {}
389     current_rank = 1
390     prev_rmse = None
391     for method, rmse in sorted_methods:
392         if prev_rmse is not None and abs(rmse - prev_rmse) < 1e-6:
393             ranks[method] = current_rank - 1
394         else:
395             ranks[method] = current_rank
396             current_rank += 1
397         prev_rmse = rmse
398     return ranks
399
400 df["best_method"] = df.apply(get_best_method, axis=1)
401 df["wc_rank"] = df.apply(lambda row: get_rank(row).get("WC") if
402     get_rank(row) else None, axis=1)
403 df["ml_rank"] = df.apply(lambda row: get_rank(row).get("ML") if
404     get_rank(row) else None, axis=1)
405 df["knn_rank"] = df.apply(lambda row: get_rank(row).get("KNN") if
406     get_rank(row) else None, axis=1)
407
408 # Summary statistics rows
409 summary_rows = []
410 best_counts = df["best_method"].value_counts().to_dict()
411
412 summary_rows.append({
413     "sid": "====SUMMARY====",
414     "n_obs": None,
415     "wc_lat": None, "wc_lon": None, "wc_rssi_rmse_db": None,
416     "ml_lat": None, "ml_lon": None, "ml_range_rms_m": None, "ml_rssi_rmse_db": None,
417     "knn_lat": None, "knn_lon": None, "knn_rssi_rmse_db": None,

```

```

415     "best_method": f"BestCounts: WC={best_counts.get('WC', 0)}, ML={best_counts.get('ML', 0)}, KNN={best_counts.get('KNN', 0)}",
416     "wc_rank": None, "ml_rank": None, "knn_rank": None,
417   })
418
419   summary_rows.append({
420     "sid": "====_AVERAGE_RMSE_===",
421     "n_obs": safe_mean(df["n_obs"]),
422     "wc_lat": None, "wc_lon": None, "wc_rssi_rmse_db": safe_mean(df["wc_rssi_rmse_db"]),
423     "ml_lat": None, "ml_lon": None, "ml_range_rms_m": safe_mean(df["ml_range_rms_m"]),
424     "ml_rssi_rmse_db": safe_mean(df["ml_rssi_rmse_db"]),
425     "knn_lat": None, "knn_lon": None, "knn_rssi_rmse_db": safe_mean(df["knn_rssi_rmse_db"]),
426     "best_method": None,
427     "wc_rank": None, "ml_rank": None, "knn_rank": None,
428   })
429
430   summary_rows.append({
431     "sid": "====_AVERAGE_RANK_===",
432     "n_obs": None,
433     "wc_lat": None, "wc_lon": None, "wc_rssi_rmse_db": None,
434     "ml_lat": None, "ml_lon": None, "ml_range_rms_m": None,
435     "ml_rssi_rmse_db": None,
436     "knn_lat": None, "knn_lon": None, "knn_rssi_rmse_db": None,
437     "best_method": None,
438     "wc_rank": safe_mean(df["wc_rank"]),
439     "ml_rank": safe_mean(df["ml_rank"]),
440     "knn_rank": safe_mean(df["knn_rank"]),
441   })
442
443   df_summary = pd.DataFrame(summary_rows)
444   df_with_summary = pd.concat([df, df_summary], ignore_index=True)
445
446   df_with_summary.to_csv("position_estimates_compare.csv", index=False,
447                         encoding="utf-8")
448   print("wrote:_position_estimates_compare.csv")
449   print(df.head(10))
450
451   print("\n====_SUMMARY_===")
452   print(f"Total samples: {len(df)}")
453   print(f"Average observations per sample: {safe_mean(df['n_obs']):.2f}")
454   )
455   print("\nBest method counts:")
456   for method, count in best_counts.items():
457     print(f"  {method}: {count} ({count/len(df)*100:.1f}%)")
458   print("\nAverage RSSI RMSE (dB):")
459   print(f"  WC: {safe_mean(df['wc_rssi_rmse_db']):.3f}")

```

```
457     print(f"ML:{safe_mean(df['ml_rssi_rmse_db']):.3f}"))
458     print(f"KNN:{safe_mean(df['knn_rssi_rmse_db']):.3f}"))
459     print("\nAverage Rank (1=best, lower is better):")
460     print(f"WC:{safe_mean(df['wc_rank']):.2f}")
461     print(f"ML:{safe_mean(df['ml_rank']):.2f}")
462     print(f"KNN:{safe_mean(df['knn_rank']):.2f}")
463
464 if __name__ == "__main__":
465     main()
```