

## Projet IMDS4A 2024/2025



### Planification de Trajet pour Véhicules Électriques à l'aide d'OpenStreetMap

**Résumé :**

Le projet IMDS4A vise à développer une solution de planification de trajets pour véhicules électriques (VE) en utilisant OpenStreetMap (OSM) et des données ouvertes. L'objectif principal est de créer un programme permettant d'optimiser les trajets des VE en fonction de critères tels que le temps et la distance, tout en intégrant une base de données détaillée sur les stations de recharge et les modèles de batteries. Ce projet repose sur la consolidation des bases de données des stations de recharge et des modèles de batteries, l'optimisation de l'algorithme de planification et la création d'une interface interactive via OSM.

**Mots-clés :** véhicules électriques, planification de trajets, OpenStreetMap, optimisation, stations de recharge, bases de données, algorithmes, visualisation cartographique.

**Abstract :**

The IMDS4A project aims to develop a route planning solution for electric vehicles (EVs) using OpenStreetMap (OSM) and open data. The main objective is to create a program that optimizes EV routes based on criteria such as time and distance while integrating a detailed database of charging stations and battery models. This project relies on consolidating databases of charging stations and battery models, optimizing the route planning algorithm, and creating an interactive interface via OSM.

**Keywords :** electric vehicles, route planning, OpenStreetMap, optimization, charging stations, databases, algorithms, cartographic visualization.

# Table des matières

<b>Projet IMDS4A 2024/2025 .....</b>	<b>1</b>
<b>Planification de Trajet pour Véhicules Électriques à l'aide d'OpenStreetMap .....</b>	<b>1</b>
Introduction.....	5
Contexte global : l'essor des véhicules électriques .....	5
Une opportunité d'apprentissage concret.....	5
État de l'art : solutions existantes et positionnement du projet .....	5
Objectifs du projet.....	6
Apports scientifiques et méthodologiques.....	6
1. Recherche et intégration des bases de données des bornes de recharge et des véhicules électriques .....	7
1.1. Construction de la Base de données des véhicules électriques.....	7
1.1.1. Outils utilisés .....	8
1.1.2. Méthodologie de Collecte de données .....	8
1.1.3. Visualisation et Exploitation de la Base de Données.....	9
1.2. Construction de la Base de données des Bornes de recharges.....	11
1.2.1. Structure de la table des bornes de recharge.....	11
1.2.2. Visualisation et Exploitation de la Base de Données.....	12
1.3. Découpage géographique de la France en carrés.....	14
1.3.1. Création des Zones Géographiques .....	14
1.3.2. Création de la base de données CARRES .....	15
2. Programme principal : Planification du trajet optimisé entre deux points .....	17
2.1. Planification des trajets optimisés en fonction des carrés .....	17
2.2. Cœur de l'algorithme .....	18
2.3. Algorithme de Planification du Trajet .....	19
2.4. Analyse de complexité.....	21
3. Sampling des bornes.....	22
3.1. Utilisation des Carrés.....	22
3.2. Traitements des bornes récupérées.....	22

4. Exemple de visualisation avec Folium .....	25
4.1. Résultats avec voiture ayant une grande autonomie .....	25
4.2. Comparaison avec voiture ayant une petite autonomie .....	28
5. Axes d'améliorations .....	30
5.1. Analyse des Performances du Programme .....	30
5.2. Affinement du Modèle en Fonction des Paramètres Réels .....	31
5.3. Optimisation des Performances et Précision des Algorithmes .....	32
Conclusion Générale .....	33
<i>Bibliographie</i> .....	34



## Introduction

### Contexte global : l'essor des véhicules électriques

La transition énergétique et le développement durable représentent des enjeux majeurs pour lutter contre les effets du changement climatique. Dans cette dynamique, les véhicules électriques (VE) s'imposent comme une solution efficace grâce à leur impact environnemental réduit. Cependant, leur adoption est encore freinée par des défis techniques et logistiques, tels que l'autonomie limitée des batteries, la diversité des modèles de recharge et le manque d'infrastructures standardisées.

Ces contraintes rendent indispensable la création d'outils de planification de trajets adaptés, capables d'optimiser l'utilisation des véhicules électriques en fonction des besoins des conducteurs et des spécificités de leur véhicule. Le projet IMDS4A s'inscrit dans cette dynamique en développant une solution dédiée, reposant sur la plateforme libre et collaborative OpenStreetMap (OSM). Cette approche permet de tirer parti des données ouvertes tout en proposant une alternative innovante et personnalisée aux solutions existantes.

### Une opportunité d'apprentissage concret

Ce projet a constitué une opportunité unique de travailler sur un problème réel, nécessitant la mise en pratique des compétences acquises au cours de notre formation en ingénierie mathématiques et data science. À travers ce développement, nous avons pu mettre en pratique et renforcer nos connaissances en :

- Gestion de bases de données, en consolidant des informations sur les stations de recharge.
- Algorithmes d'optimisation, pour la planification des itinéraires.
- Visualisation cartographique, en exploitant les fonctionnalités d'OpenStreetMap et Folium .

Par ailleurs, ce travail ne s'est pas limité à une simple application des savoirs existants. Il nous a également poussé à apprendre de nouvelles technologies et à expérimenter des méthodologies pour résoudre les problématiques rencontrées. Cette démarche nous a permis de renforcer notre adaptabilité face à des défis imprévus et de découvrir des procédés pour atteindre nos objectifs.

### État de l'art : solutions existantes et positionnement du projet

Le marché propose déjà plusieurs outils pour la planification des trajets en VE, parmi lesquels :

- A Better Route Planner (ABRP) : Une application populaire permettant d'optimiser les trajets en fonction de la charge de la batterie et des préférences utilisateur.
- PlugShare : Un service axé sur la localisation des stations de recharge.
- Chargemap : Une application de planification d'itinéraire intelligent pour voitures électriques en fonction des bornes de recharge présentes tout le long du chemin.

Ces solutions, bien qu'efficaces pour certaines fonctionnalités, présentent des limites notables. Elles sont souvent coûteuses, propriétaires, ou restreintes à des scénarios d'utilisation spécifiques. Contrairement à ces approches, notre projet ne s'est pas inspiré directement de ces outils, mais repose sur une réflexion indépendante. L'objectif était de concevoir une solution unique, adaptée aux besoins définis, tout en exploitant la flexibilité des données ouvertes et des outils collaboratifs d'OpenStreetMap.



## Objectifs du projet

L'objectif principal du projet IMDS4A était de développer un programme complet et performant permettant de :

- Planifier des trajets optimisés pour les VE, en tenant compte de critères tels que le temps, le coût et la distance.
- Intégrer une base de données enrichie, regroupant des informations détaillées sur les stations de recharge et les modèles de batteries.
- Offrir une visualisation intuitive et interactive des itinéraires grâce aux outils OSM.

## Apports scientifiques et méthodologiques

Ce projet s'est révélé particulièrement formateur, tant sur le plan technique que méthodologique. Il nous a permis de :

- Associer algorithmes, gestion des données et visualisation dans un même projet.
- Expérimenter de nouvelles approches pour résoudre des problèmes complexes, en utilisant des outils comme SQLite pour les bases de données ou Folium pour la création de cartes interactives.
- Développer des compétences en gestion de projet, en apprenant à structurer notre travail autour d'objectifs clairs et mesurables.

En outre, ce projet a renforcé notre capacité à collaborer efficacement dans un contexte d'équipe, en partageant nos compétences et en apprenant les uns des autres.

## 1. Recherche et intégration des bases de données des bornes de recharge et des véhicules électriques

### 1.1. Construction de la Base de données des véhicules électriques

Dans le cadre de notre projet, nous avons eu besoin de constituer une base de données répertoriant divers modèles de véhicules électriques en fonction de leurs caractéristiques techniques. Pour ce faire, nous avons exploité les informations disponibles sur le site "Fiches-Auto" [1], qui regroupe des données complètes sur les véhicules électriques de plusieurs marques (cf. figure1).

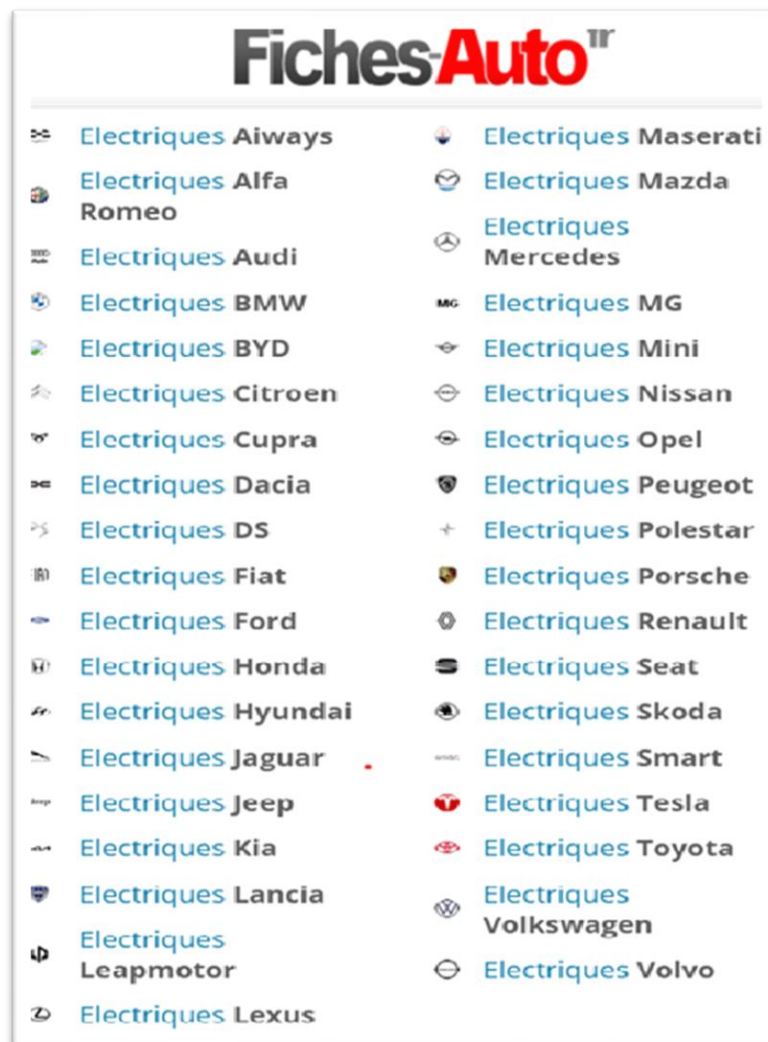


Figure 1 - Fiche auto sur les VE [1]



### 1.1.1. Outils utilisés

Pour la gestion et le stockage de ces données, nous avons opté pour **SQLite3** [2], une bibliothèque Python permettant de créer et d'exploiter des bases de données relationnelles en utilisant le langage SQL.

Afin d'extraire les données du site "Fiches-Auto", nous avons eu recours à une technique appelée **Web Scraping** (cf. **figure 2**), qui consiste à collecter automatiquement des informations accessibles sur Internet. Pour réaliser cette opération, nous avons utilisé les bibliothèques Python **BeautifulSoup** et **requests** [3],[4].

```
# on se connecte a la base de donnée
data=sqlite3.connect('DataBase.db')

# Lien du site sur lequel on veut extraire les données
url = 'https://www.fiches-auto.fr/articles-auto/voiture-electrique/s-2531-fiches-techniques-voitures-electriques.php'

rep_serv= requests.get(url)

if rep_serv.ok==False:
    exit()

# Permet pour la suite de scruter le code HTML de la page pour trouver les
# informations qu'on cherche avec BS4
soup=BeautifulSoup(rep_serv.text,'lxml')
```

Figure 2 - Récupération des données du site

### 1.1.2. Méthodologie de Collecte de données

Le processus de collecte des données repose sur plusieurs étapes clés :

1. **Envoi d'une requête HTTP** : Nous utilisons la bibliothèque requests pour interroger le serveur du site "Fiches-Auto" et obtenir le contenu HTML des pages concernées. Il est à noter que certains sites, comme celui de Tesla, bloquent l'accès aux requêtes automatisées, mais ce n'est pas le cas ici.
2. **Extraction des informations pertinentes** :
  - Avec BeautifulSoup, nous analysons le code HTML des pages ciblées.
  - Nous avons ciblé la section /article-auto/voiture-electrique/fiche-technique du site, où sont centralisées les informations techniques des véhicules.
  - Étant donné que la structure HTML de ces fiches est similaire pour chaque voiture, nous avons développé un algorithme capable de parcourir automatiquement ces pages et d'extraire les données de manière systématique.
3. **Traitement et stockage des données** :
  - On se sert principalement des balises qui sont présentes dans le code HTML pour sélectionner les informations souhaitées et naviguer de page en page. Nous avons sélectionné les données les plus pertinentes pour notre projet, à savoir : **l'autonomie, la consommation énergétique et la capacité de la batterie.**



- Une fois extraites, ces informations sont formatées et insérées dans une base de données SQLite3 afin de faciliter leur exploitation ultérieure.

### 1.1.3. Visualisation et Exploitation de la Base de Données

Une fois la base de données constituée, nous l'avons visualisée et vérifiée à l'aide de **SQLite Studio** [5], un outil permettant de manipuler facilement les bases SQLite et d'exécuter des requêtes SQL pour vérifier la cohérence et la qualité des données collectées (cf. **figure 3**).

id	modele	charge_time	charge_time	autonomie	consommati	batterie_Kw
1	1 Cybertruck Propulsion 2023	29	14.5	402	17	0
2	2 Cybertruck Transmission intégrale 2023	39	19.5	547	17	123
3	3 Cybertruck Cyberbeast 2023	39	19.5	515	17	123
4	4 Model 3 Propulsion 2021	30	15	491	17	60
5	5 Model 3 Propulsion 2023	30	15	513	17	60
6	6 Model 3 SR+ 2018	34	17	409	17	55
7	7 Model 3 Grande Autonomie 2018	25	12.5	560	17	77
8	8 Model 3 Grande Autonomie 2021	26	13	602	17	82
9	9 Model 3 Grande Autonomie 2023	25	12.5	678	17	79
10	16 Model S Standard Range 2018	31	15.5	450	17	75
11	18 Model S 75D 2017	31	15.5	480	17	74
12	23 Model S Long Range 2018	31	15.5	652	17	98
13	24 Model S Base 2021	31	15.5	652	17	98
14	29 Model S Plaid 2021	31	15.5	637	17	98
15	30 Model X 75D 2016	31	15.5	381	17	74
16	31 Model X 90D 2016	26	13	414	17	84.5
17	32 Model X 100D 2016	31	15.5	475	17	98
18	34 Model X P90D 2016	26	13	402	17	84.5
19	35 Model X P100D 2016	31	15.5	465	17	98
20	36 Model X Base 2021	31	15.5	560	17	98
21	37 Model X Long Range 2018	31	15.5	507	17	98
22	38 Model X Plaid 2021	31	15.5	236	17	98
23	39 Model Y Propulsion Berlin 2022	36	18	455	17	58
24	40 Model Y Propulsion shanghai 2022	30	15	455	17	60
25	41 Model Y Grande Autonomie Propulsion RWD 2024	25	12.5	600	17	79
26	42 Model Y Grande Autonomie 2021	26	13	533	17	82
27	43 Model Y Grande Autonomie 2023	25	12.5	533	17	79

Figure 3 - Base de données sur la marque Tesla

Nous avons extrait des informations provenant de sites spécialisés sur les véhicules électriques. Chaque véhicule a été identifié par un ID unique, et pour chaque modèle, nous avons récupéré son autonomie, sa consommation moyenne (en kWh/100 km) et la capacité de la batterie (en kWh). La table dédiée aux véhicules contient toutes les informations nécessaires à la planification d'un trajet optimisé.

```
# Maintenant on va relever les proprietes des voitures
car_url=requests.get('https://www.fiches-auto.fr'+ a2['href'])
soup3=BeautifulSoup(car_url.text,'lxml')
# Ici on va chercher l'autonomie des voitures
# Cela permet de récupérer le code des <td> comportant a l'interieur le mot autonomie
tds3=soup3.findAll('td', style=lambda value: value and 'background-color:#' in value)
for td3 in tds3:
    a3=td3.find('strong')
    if re.search(r'\bkm(?:\w|\s|\/)',a3.text):
        auto=a3.text
        break

#Maintenant on va chercher le temps de charge de chaque voiture à 80 % et + 20 %
tds3=soup3.findAll('p')

for td3 in tds3:
    if 'Temps de charge DC 0-80%' in td3.text:
        a3=td3.find('strong')
        charge_time80=int(a3.text)
        charge_time100 = 0.5*charge_time80
        break
    # On calcule le temps de 80-100 en considérant V2=0.5*V1
# On recherche la consommation et la capacité de la batterie
tds3=soup3.findAll('strong')
lev=0
for td3 in tds3:
    if 'kWh/100km' in td3.text:
        conso=td3.text
        break
    if re.search(r'\bkWh(?:\w|\s|\/)',td3.text) and lev==0:
        batterie=td3.text
        lev=1
ajouter_modele(data,marque, voiture,charge_time80,enleve_cara(auto,"km"),charge_time100,enleve_cara(conso,"kWh/100km"),enleve_cara(batterie,"kWh"))
```

Figure 4 - Script de récupération des caractéristiques des voitures électriques

L'autonomie de la voiture est recherchée en identifiant les éléments <td> contenant le mot autonomie. Une expression régulière est utilisée pour vérifier la présence du mot km, garantissant ainsi que la valeur correcte est extraite.

Le script détermine ensuite le temps de charge à 80 % en scrutant les paragraphes (<p>) de la page pour une mention de Temps de charge DC 0-80%. Une fois cette valeur extraite, le temps de charge de 80 à 100 % est estimé comme étant à moitié du temps initial.

Enfin, la consommation d'énergie (kWh/100km) et la capacité de la batterie (kWh) sont extraites des balises <strong>. Une boucle parcourt ces balises et utilise des expressions régulières pour identifier et extraire les valeurs correspondantes.

Toutes ces informations sont ensuite enregistrées dans la base de données en appelant la fonction **ajouter\_modele**, qui prend en paramètre les données de la voiture (marque, modèle, autonomie, temps de charge, consommation et capacité de la batterie) en s'assurant de nettoyer les unités de mesure avec la fonction **enleve\_cara**.

## 1.2. Construction de la Base de données des Bornes de recharges

Dans le cadre de notre projet, nous avons dû récupérer et structurer les données relatives aux bornes de recharge afin de les intégrer dans notre système de planification de trajets. Pour cela, nous avons opté pour les données mises à disposition sur le site du gouvernement [6]. Ces données sont disponibles sous forme de fichiers CSV et contiennent des informations essentielles telles que l'ID unique de chaque borne, ses coordonnées géographiques (latitude et longitude) ainsi que sa puissance de recharge. Ces éléments sont indispensables pour optimiser les trajets en véhicules électriques.

### 1.2.1. Structure de la table des bornes de recharge

Contrairement à la base de données des véhicules électriques, le Web Scraping n'était pas nécessaire puisque les données étaient librement accessibles sous forme de fichiers CSV. Nous avons donc directement téléchargé et intégré ces données dans notre base de données SQLite.

```
data=sqlite3.connect('Borne.db')
cursor=data.cursor()
cursor.execute("SELECT * FROM CARRES")

# Récupération des résultats
DB= cursor.fetchall()

cursor.execute(f'''
    CREATE TABLE IF NOT EXISTS BORNES (
        idBorne NUMBER(6) PRIMARY KEY,
        x REAL,
        y REAL,
        puissance_KW REAL,
        idCarre NUMBER(2) CONSTRAINT CF_B REFERENCES CARRES(idCarre)
    )
''')
data.commit()
data.close()
i=1
for x in coordi:
    data=sqlite3.connect('Borne.db')
    for W in DB:
        # on regarde si la borne est a l'intérieur du carré
        if (min(W[1], W[3]) <= x[0][0] <= max(W[1], W[3])) and (min(W[2], W[4]) <= x[0][1] <= max(W[2], W[4])):
            cursor=data.cursor()
            cursor.execute("BEGIN TRANSACTION")
            cursor.execute(f'''
                INSERT INTO BORNES
                (idBorne,x, y,puissance_KW,idCarre)
                VALUES (?, ?, ?, ?, ?)
                ''', (i,x[0][0],x[0][1],x[1],W[0]))
            i=i+1
            data.commit()
            data.close()
```

Figure 5 - Création de la base de données BORNES

Cette table contient plusieurs champs clés tels que l'identifiant unique de la borne (**idBorne**), ses coordonnées géographiques (**latitude et longitude**), sa puissance de recharge (**puissance\_KW**) et un champ de référence (**idCarre**) permettant d'associer chaque borne à une subdivision géographique pour une meilleure optimisation des trajets.

Afin d'automatiser le processus d'importation et d'organisation des données, nous avons développé un script Python utilisant le module sqlite3. Ce script commence par établir une connexion à la base de données (Borne.db) (cf. figure 5), puis lit le fichier CSV pour extraire les informations des bornes.

Chaque borne est ensuite analysée afin de vérifier si elle appartient à une subdivision géographique préalablement définie dans une autre table (CARRES). Une fois cette vérification effectuée, la borne est insérée dans la base de données avec toutes ses informations associées, et un identifiant unique (idBorne) lui est attribué. Pour garantir l'intégrité des données, nous avons mis en place des transactions SQL (BEGIN TRANSACTION), permettant d'assurer la cohérence et d'éviter toute perte d'informations en cas d'erreur durant l'importation. Enfin, la connexion à la base de données est fermée après chaque mise à jour pour garantir la stabilité et la fiabilité du système.

### 1.2.2. Visualisation et Exploitation de la Base de Données

Une fois les bornes intégrées, nous avons utilisé SQLite Studio pour visualiser et valider les données. Cet outil nous a permis d'exécuter des requêtes SQL afin de garantir la cohérence et la qualité des informations stockées.

	idBorne	x	y	v	puissance KW	idCarre
1	1	48.723084		-0.056488	300	7
2	2	47.095986		2.240692	400	13
3	3	47.301879		0.891121	400	12
4	4	44.361374		-0.852195	400	9
5	5	44.361165		-0.849669	400	9
6	6	43.937583		-1.092778	400	9
7	7	43.937391		-1.08937	400	9
8	8	47.650769		0.470145	200	7
9	9	45.1437823		4.1161033	22	19
10	10	43.41959147913006		3.40760912322576	22	14
11	11	48.83267793516981		2.49356956759058	22	12
12	12	41.908579309038		8.65788830175805	22	25
13	13	43.476583984941		5.476711409891	22	19
14	14	43.3292004491334		5.14376626549764	36	19
15	15	45.704218		4.578864	11	18
16	16	50.684918		3.207306	22	11
17	17	50.68494		3.207246	22	11
18	18	50.684984		3.207124	22	11
19	19	50.684898		3.20737	22	11
20	20	48.931356		2.212336	22	12
21	21	50.685013		3.207047	22	11
22	22	50.684964		3.207187	22	11
23	23	50.685049		3.206962	22	11
24	24	50.684876		3.207433	22	11
25	25	48.841032		2.798854	22	12

Figure 6 - Base de données des bornes sur SQLite

Pour mieux comprendre la répartition des bornes de recharge sur le territoire, nous avons également généré une carte des bornes en France grâce à la bibliothèque Folium [7], permettant d'observer leur répartition géographique (cf. figure 7).

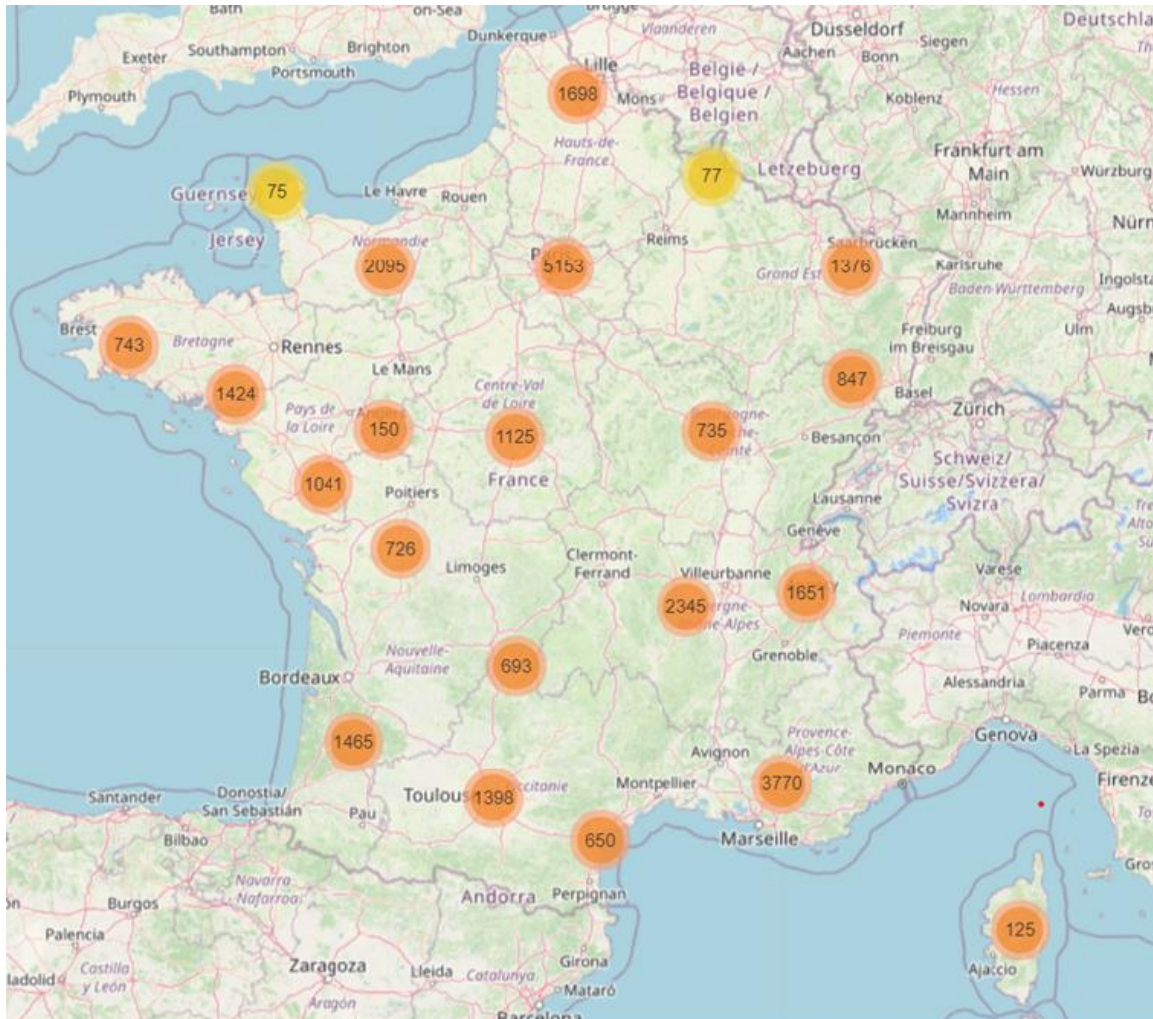


Figure 7 - Bornes en France

### 1.3. Découpage géographique de la France en carrés

Après avoir intégré les données des bornes de recharge et des véhicules électriques dans nos bases de données SQLite, une étape clé de notre travail a consisté à créer un système géographique permettant d'optimiser la localisation des bornes de recharge tout au long du trajet d'un véhicule. L'objectif était de simuler un parcours entre deux points, A et B, tout en tenant compte de l'autonomie du véhicule et des bornes disponibles sur la route, de manière à minimiser le temps et la distance de recharge.

Afin d'optimiser la gestion des bornes de recharge et de limiter la charge de traitement des requêtes, nous avons décidé de subdiviser la France en **25 zones géographiques**. Cette subdivision permet d'accélérer l'accès aux bornes pertinentes en ne chargeant que celles situées à proximité immédiate du trajet de l'utilisateur. En procédant ainsi, nous pouvons facilement localiser les bornes qui se trouvent à proximité d'un itinéraire donné et déterminer celles qui sont les plus accessibles en fonction de l'itinéraire du véhicule.

#### 1.3.1. Création des Zones Géographiques

La division du territoire a été réalisée en définissant **25 carrés (cf. figure 8)**, chacun caractérisé par ses coordonnées géographiques (coin haut gauche et bas droit) et un identifiant unique (idCarre). Lors de l'intégration des bornes de recharge, chaque borne s'est vu attribuer un idCarre correspondant à la zone dans laquelle elle est située. Cette approche permet de structurer efficacement les données et d'accélérer leur exploitation lors du calcul des itinéraires.

Avant d'associer les bornes aux zones géographiques, nous avons procédé à un nettoyage des données pour garantir leur qualité et leur fiabilité. Cette phase de prétraitement a permis de :

- **Éliminer les doublons**, évitant ainsi la présence de plusieurs entrées identiques pour une même borne.
- **Supprimer les bornes incomplètes**, notamment celles dépourvues de coordonnées géographiques ou d'indications sur leur puissance de recharge ainsi que les bornes ayant une puissance de rechargement nulle.
- **Harmoniser les types de données**, afin de garantir une compatibilité et une exploitation fluide lors des requêtes SQL ultérieures.



```
# On démarre avec 4 points qui vont nous servir de repère pour crée le cadrillage
base_carre=[[51.124,9.537],[51.124,-5.142],[41.303,9.537],[41.303,-5.142]]
subdiv_carre=[]
# division en n carre

n=5
x=(base_carre[1][0]-base_carre[2][0])/n
y=(base_carre[1][1]-base_carre[2][1])/n

for i in range(0,n+1):
    for j in range(0,n+1):
        subdiv_carre.append([base_carre[1][0]-j*x,base_carre[1][1]-i*y])
```

Figure 8 - Création d'une subdivision de la zone étudiée

### 1.3.2. Création de la base de données CARRES

Le scripte présenté (cf. figure 8) permet de créer dans la même base de données que les voitures électriques et bornes i.e. **Borne.db** la table **CARRES** (cf. figure 9) destinée à stocker des informations géographiques relatives à des carrés. La table contient cinq colonnes : un identifiant idCarre, des coordonnées géographiques xlat, ylat, xlo, et ylo représentant respectivement les latitudes et longitudes de certains points.

```
#création de la base de donnée

data=sqlite3.connect('Borne.db')

carre='carre'
k=1
i=6
cursor=data.cursor()
cursor.execute(f'''
CREATE TABLE IF NOT EXISTS CARRES (
            idCarre NUMBER(2) PRIMARY KEY,
            xlat REAL,
            ylat REAL,
            xlo REAL,
            ylo REAL
        )
''')
data.commit()
```

Figure 9 - Création de la base de données CARRES



L'identifiant nous permettra d'associer à chaque borne un idCarre pour savoir dans quelle carré la borne est présente. Pour finir on regarde pour chaque borne dans quel carré elle est et on ajoute tout cela dans une base de données pour obtenir la table (cf. figure 10).

	idCarre	xlat	ylat	xlo	ylo
1	1	51.124	-2.2062	49.1598	-5.142
2	6	51.124	-2.2062	49.1598	0.7296
3	2	49.1598	-2.2062	47.1956	-5.142
4	7	49.1598	-2.2062	47.1956	0.7296
5	3	47.1956	-2.2062	45.2314	-5.142
6	8	47.1956	-2.2062	45.2314	0.7296
7	4	45.2314	-2.2062	43.2672	-5.142
8	9	45.2314	-2.2062	43.2672	0.7296
9	5	43.2672	-2.2062	41.303	-5.142
10	10	43.2672	-2.2062	41.303	0.7296
11	11	51.124	3.6654	49.1598	0.7296
12	16	51.124	3.6654	49.1598	6.6012
13	12	49.1598	3.6654	47.1956	0.7296
14	17	49.1598	3.6654	47.1956	6.6012
15	13	47.1956	3.6654	45.2314	0.7296
16	18	47.1956	3.6654	45.2314	6.6012
17	14	45.2314	3.6654	43.2672	0.7296
18	19	45.2314	3.6654	43.2672	6.6012
19	15	43.2672	3.6654	41.303	0.7296
20	20	43.2672	3.6654	41.303	6.6012
21	21	51.124	9.537	49.1598	6.6012
22	22	49.1598	9.537	47.1956	6.6012
23	23	47.1956	9.537	45.2314	6.6012
24	24	45.2314	9.537	43.2672	6.6012
25	25	43.2672	9.537	41.303	6.6012

Figure 10 - Base de données CARRES sur SQLite

L'intégration de la base de données des bornes de recharge constitue une étape fondamentale de notre projet. Grâce aux données disponibles, nous avons pu récupérer des informations essentielles sur les stations de recharge, garantissant ainsi une source fiable pour l'optimisation des trajets en véhicules électriques. L'utilisation de SQLite3 a permis de stocker ces données de manière structurée et efficace, facilitant leur exploitation par la suite.

Pour optimiser la rapidité et la pertinence des requêtes, nous avons subdivisé la France en 25 zones géographiques. Cette approche réduit considérablement la quantité de données chargées lors du calcul des trajets, permettant ainsi de ne considérer que les bornes les plus pertinentes. En complément, un important travail de nettoyage et de correction des données a été réalisé afin d'éliminer les doublons, de compléter les informations manquantes et d'assurer une homogénéité des formats utilisés.

Avec une base de données fiable et optimisée, nous pouvons désormais nous concentrer sur l'objectif principal du projet : la **planification intelligente des trajets en véhicule électrique**. La prochaine étape consiste à développer un algorithme permettant d'optimiser les itinéraires en tenant compte de l'autonomie des véhicules, de la consommation énergétique et des bornes de recharge disponibles sur le parcours et comprendre entre l'objectif du projet et le lien entre la subdivision de la France en 25 zones géographiques.

## 2. Programme principal : Planification du trajet optimisé entre deux points

Le programme principal du projet se concentre sur la planification d'un trajet optimisé entre deux points, A et B, en prenant en compte l'autonomie du véhicule, sa consommation énergétique, et les arrêts nécessaires pour recharger la batterie. Voici les étapes détaillées du processus :

### 2.1. Planification des trajets optimisés en fonction des carrés

Lorsqu'un conducteur souhaite effectuer un trajet entre deux points, A et B, le système doit déterminer les carrés dans lesquels l'itinéraire passe. L'itinéraire entre ces deux points est d'abord tracé sur une carte, puis découpé en segments correspondant aux zones géographiques (les carrés). Pour chaque segment de l'itinéraire, nous identifions les bornes de recharge présentes dans le carré correspondant à ce segment.

Nous avons utilisé un algorithme de **recherche de chemin** pour identifier les segments du trajet et associer les bornes de recharge aux différents carrés traversés par l'itinéraire. Par exemple, si un véhicule part de Paris (point A) et se rend à Lyon (point B), l'itinéraire sera découpé en segments géographiques (carrés) représentant les différentes zones traversées, et pour chaque segment, nous rechercherons les bornes de recharge les plus proches de l'itinéraire.

Les bornes ainsi identifiées peuvent être associées à des critères comme :

- La distance entre l'itinéraire et la borne
- La proximité de la route la plus optimisée pour un trajet entre A et B
- La puissance de recharge des bornes, pour favoriser celles qui permettront une recharge plus rapide

Ce découpage géographique a plusieurs avantages. Il permet de réduire la recherche de bornes en filtrant les zones géographiques pertinentes, ce qui améliore l'efficacité du système et permet de proposer des solutions de recharge adaptées à l'itinéraire du véhicule.

L'algorithme doit non seulement trouver le chemin le plus rapide mais aussi optimiser les pauses de recharge, en tenant compte du temps de rechargement et de l'état de la batterie.

## 2.2. Cœur de l'algorithme

### Algorithme de Planification de Trajet Optimisé

#### Définition des Variables :

- $N$  : Nombre total de bornes (incluant le point de départ  $A$  et l'arrivée  $B$ ).
- $e$  : Niveau de batterie du véhicule (en %).
- $r$  : Niveau de batterie après recharge (en %).
- $b_i$  : Borne en position  $i$ .
- $b_k$  : Borne en position  $k$  ( $k > i$ ).
- $temps\_recharge(e, r)$  : Temps nécessaire pour recharger de  $e$  à  $r$  %.
- $temps\_trajet(b_i, b_k)$  : Temps pour aller de la borne  $b_i$  à la borne  $b_k$ .
- $conso$  : Consommation d'énergie du véhicule entre deux bornes.
- $Bfs$  : Temps total du trajet incluant la recharge et le déplacement.
- $T[i][e]$  : Temps minimal pour atteindre la borne  $B$  depuis la borne  $b_i$  avec  $e$  % de batterie.

---

#### Algorithm 1 Planification du Trajet Optimisé

---

```

1: Début
2: for  $i \leftarrow N - 2$  to 0 do
3:   for  $e \leftarrow 100$  to 0 step  $-5$  do
4:      $T[i][e] \leftarrow +\infty$ 
5:     for  $k \leftarrow i + 1$  to  $N$  do
6:       for  $r \leftarrow e$  to 100 step 5 do
7:         if Assez de batterie pour atteindre  $b_k$  then
8:            $Bfs \leftarrow temps\_recharge(e, r) + temps\_trajet(b_i, b_k) +$ 
              $T[k][r - conso]$ 
9:           if  $Bfs < T[i][e]$  then
10:             $T[i][e] \leftarrow Bfs$ 
11:           end if
12:         end if
13:       end for
14:     end for
15:   end for
16: end for

```

---

Figure 11 - Algorithme principal

L'algorithme (cf. figure 11) commence par la borne d'arrivée  $B$  et remonte ensuite vers le départ  $S$ . Cela permet de calculer, pour chaque borne, le meilleur moyen d'atteindre  $B$  à partir des bornes suivantes, en utilisant les résultats déjà calculés pour ces bornes. Ce processus rétroactif permet d'optimiser le chemin de  $S$  à  $B$ .

Ensuite, pour chaque borne, l'algorithme examine tous les niveaux de batterie possibles, de 100 % à 0 %, par étapes de 5 %. Cela permet de prendre en compte les différents scénarios où la batterie pourrait être pleine, à moitié pleine, ou presque vide, et de tester les solutions adaptées à chaque situation.

À chaque niveau de batterie, l'algorithme teste les trajets possibles vers les bornes suivantes, c'est-à-dire celles plus éloignées. Il vérifie si le véhicule a suffisamment de batterie pour atteindre la prochaine borne sans avoir besoin de se recharger. Si ce n'est pas le cas, il explore les possibilités de recharge avant de continuer le trajet.

Si la batterie est insuffisante pour atteindre la borne suivante, l'algorithme calcule le temps nécessaire pour recharger la batterie et ajoute ce temps au temps de trajet entre les deux bornes. Il teste différents niveaux de recharge et choisit celui qui permet de réduire au maximum le temps total du trajet.

Enfin, l'algorithme additionne le temps de recharge et le temps de trajet entre chaque paire de bornes, et sélectionne le temps le plus court pour chaque scénario (c'est-à-dire pour chaque combinaison de niveau de batterie et de borne). Il met ainsi à jour, à chaque étape, le meilleur temps possible pour atteindre B, garantissant que le trajet soit effectué dans les meilleures conditions.

### 2.3. Algorithme de Planification du Trajet

L'algorithme doit prendre en compte plusieurs critères pour déterminer les arrêts optimaux :

- La distance entre l'itinéraire et la borne,
- La puissance de recharge des bornes,
- Le niveau de batterie du véhicule.

L'objectif est de réduire le temps total de trajet en optimisant à la fois la distance parcourue et le temps passé en recharge.

Pour mieux comprendre la logique de notre algorithme, nous représentons le problème sous forme de tableau de cases (**cf. figure 12**), où chaque borne de recharge est un nœud et les liaisons entre ces bornes représentent le temps de trajet, incluant le déplacement et la recharge éventuelle.



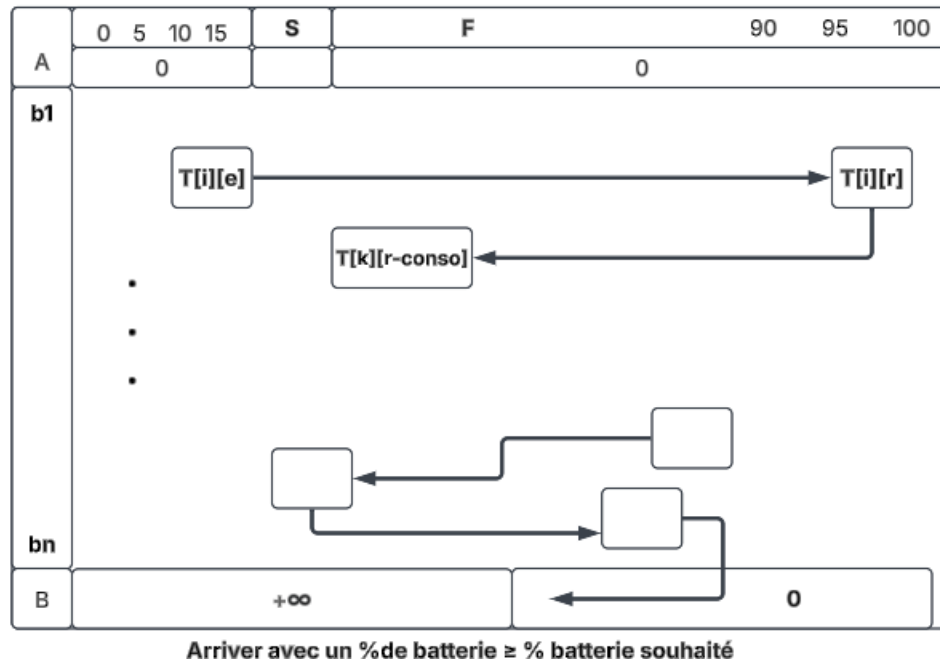


Figure 12 - Schéma représentatif de l'algo

L'itinéraire est structuré autour des éléments suivants :

- **A** : Point de départ du trajet.
- **B** : Destination finale du trajet.
- **B<sub>i</sub> (pour i=1, ..., n)** : Bornes de recharge situées entre A et B.
- **S** : Pourcentage de batterie au départ.
- **F** : Pourcentage de batterie souhaité à l'arrivée.

Le tableau de calcul utilisé, **T[N][21]**, est construit de la manière suivante :

- **N** représente le nombre total de points considérés (les B<sub>i</sub> ainsi que A et B), soit n+2.
- Chaque colonne correspond à un état de la batterie (**de 0% à 100% en paliers de 5%**), soit 21 niveaux possibles.
- Chaque cellule T[i][j] contient le temps minimal pour atteindre la borne B<sub>i</sub> avec un pourcentage de batterie donné.

Le processus d'initialisation de ce tableau suit ces règles :

- À l'arrivée (**B**), les valeurs correspondant aux niveaux de batterie inférieurs à **F** sont fixées à  $+\infty$ , indiquant qu'il est impossible d'atteindre la destination avec une batterie insuffisante.
- Les valeurs correspondant aux niveaux de batterie supérieurs ou égaux à **F** sont fixées à 0, car aucun temps de trajet supplémentaire n'est requis une fois l'objectif atteint.

L'algorithme procède ensuite en remontant progressivement depuis B jusqu'à A, en calculant pour chaque borne  $B_i$  le meilleur choix de recharge et de déplacement pour atteindre BBB dans un temps minimal.

## 2.4. Analyse de complexité

L'algorithme mis en place suit une **complexité de  $O(n^2p^2)$** . Cette complexité découle du fait que nous travaillons avec un **tableau de dimensions (n, p)**, où **n** représente le nombre de bornes et **p** les niveaux de recharge possibles.

Chaque case de ce tableau nécessite de comparer  **$n \times p$  valeurs**, ce qui explique pourquoi la complexité devient quadratique par rapport à ces deux paramètres. En d'autres termes, pour chaque borne et chaque niveau de recharge, nous devons tester toutes les autres bornes et tous les autres niveaux de charge afin de trouver le trajet optimal.

Dans le **pire des cas**, c'est-à-dire la situation où l'on maximise le nombre d'éléments traités, nous avons :

- **$p = 21$** , car nous considérons des niveaux de charge allant de **0% à 100% par paliers de 5%**.
- **$n - 2 = 97$** , ce qui correspond au nombre de bornes retenues après le filtrage effectué sur le trajet.

La limite du nombre de points utilisables simultanément vient du fait que nous utilisons l'API OSRM, qui impose une restriction à 99 points par requête. Cela signifie que notre algorithme doit inclure le point de départ (A) et le point d'arrivée (B) en plus des bornes sélectionnées, soit  $n = 99$  au total dans le pire des scénarios.

### 3. Sampling des bornes

Dans cette section, nous abordons la méthode de *sampling* des bornes autour d'un trajet déterminé par l'algorithme OSRM, utilisé pour calculer les trajets sans tenir compte des bornes de recharge. L'objectif est de déterminer les bornes présentes autour du trajet en se basant sur une grille de carrés couvrant la France, tout en optimisant le processus de sélection pour ne garder que les bornes les plus pertinentes.

#### 3.1. Utilisation des Carrés

La première étape consiste à utiliser l'API OSRM [8] pour calculer un trajet, ce qui permet de récupérer une liste de points représentant le parcours. Cette liste est composée de plusieurs coordonnées géographiques  $Liste\_points = [(lat_0, lon_0), \dots, (lat_N, lon_N)]$ , où chaque point représente un segment du trajet. L'objectif est de déterminer quels carrés géographiques se trouvent autour de ce trajet et d'identifier les bornes de recharge qui y sont situées.

Afin de rendre le processus plus efficace, la France est divisée en 25 carrés géographiques égaux. L'algorithme parcourt la liste des points du trajet en effectuant des sauts réguliers, à un pas arbitraire de 500. En fonction de la longueur du trajet, ce pas peut générer entre 1 000 et 20 000 points. Pour chaque point du trajet, l'algorithme identifie le carré auquel il appartient. Ensuite, il génère quatre nouveaux points à partir de chaque point du trajet en ajoutant des déplacements vers le Nord, le Sud, l'Est et l'Ouest, chacun étant situé à 5 km du point d'origine, ce qui permet d'étendre la zone d'exploration autour du trajet.

Si ces nouveaux points se trouvent dans un carré différent de celui du point original, l'algorithme récupère les bornes présentes dans le carré correspondant. Cette approche permet de localiser rapidement toutes les bornes les plus proches du trajet, réduisant ainsi considérablement le temps de calcul. Une stratégie est également mise en place pour gérer les points dans la direction inverse du trajet. Par exemple, pour un trajet de Clermont à Paris, un point est également ajouté au sud, ce qui permet de vérifier les bornes de manière plus exhaustive, évitant ainsi de sauter accidentellement des carrés importants.

#### 3.2. Traitements des bornes récupérées

Maintenant que les bornes situées autour du trajet ont été identifiées, il est nécessaire de les filtrer afin de ne conserver que les plus pertinentes. Cette étape est essentielle, car l'API OSRM impose une limitation stricte sur le nombre de bornes pouvant être prises en compte, fixé à un maximum de 97. Il est donc crucial d'appliquer un processus de sélection efficace pour ne garder que les bornes les plus utiles au trajet.

Les critères déterminants pour cette sélection sont principalement la position des bornes par rapport au trajet et leur puissance de charge. Une borne située trop loin du trajet n'est pas intéressante, car elle impliquerait un détour significatif, ce qui nuirait à l'optimisation du temps de parcours.



Pour effectuer cette sélection, un algorithme de *sampling* (cf. **figure 13**) est mis en place.

---

**Algorithm 1** Algorithme de Sampling des bornes

---

```
1: Initialisation : Liste des bornes sélectionnées Borne  $\leftarrow []$ 
2: if N_liste_points < 3000 then
3:   for i allant de 0 à N_liste_points avec un pas de 50 do
4:     Borne_temp  $\leftarrow$  Select_borne(Liste_points[i], 2)
5:     BorneMax  $\leftarrow$  nlargest(1, Borne_temp, puissance)
6:     if BorneMax  $\notin$  Borne then
7:       Borne.append(BorneMax)
8:     end if
9:   end for
10: else
11:   for i allant de 0 à N_liste_points avec un pas de 250 do
12:     Borne_temp  $\leftarrow$  Select_borne(Liste_points[i], 10)
13:     BorneMax  $\leftarrow$  nlargest(1, Borne_temp, puissance)
14:     if BorneMax  $\notin$  Borne then
15:       Borne.append(BorneMax)
16:     end if
17:   end for
18: end if
```

---

**Figure 13 - Algorithme de sampling**

L'idée est de parcourir la liste des points du parcours par pas de 50 et de sélectionner la borne de recharge la plus intéressante en fonction de critères définis. Si le chemin est relativement court (moins de 3000 points), on applique d'abord un rayon de sélection de 2 km autour de chaque point du chemin, car cela permet de prendre plus de bornes sur le chemin, puisque le nombre de bornes sera réduit si l'on prend un pas et un rayon trop grand. A chaque itération, on retient uniquement la borne qui fournit la puissance de charge maximale.

Dans le cas de trajet plus long (plus de 3000 points), nous élargissons ensuite le **rayon** de recherche à **10 km** afin de maximiser les chances d'obtenir une borne pertinente. Cette méthode permet de rester sous la limite des 97 bornes imposées, même pour les trajets les plus longs. Si aucune borne plus puissante n'est disponible, alors la borne précédente est conservée, et aucune nouvelle borne n'est ajoutée.

Les trajets les plus longs en France comportent généralement moins de 20 000 points. Par exemple, un trajet reliant Berga, au nord de l'Espagne, à Lille totalise environ 14 241 points. Cette contrainte permet de s'assurer que la condition de sélection d'un maximum de 97 bornes est toujours respectée. En effet, avec un pas de 225 points, la répartition maximale des bornes s'établit à  $20000/250=80$ , ce qui reste inférieur à la limite imposée de 97 bornes.

L'algorithme présente plusieurs avantages. Il garantit notamment qu'un trajet de A à B puisse toujours être trouvé, car les bornes sont réparties de manière uniforme le long du parcours. Cette approche assure un équilibre entre optimisation du temps de calcul et faisabilité du trajet.

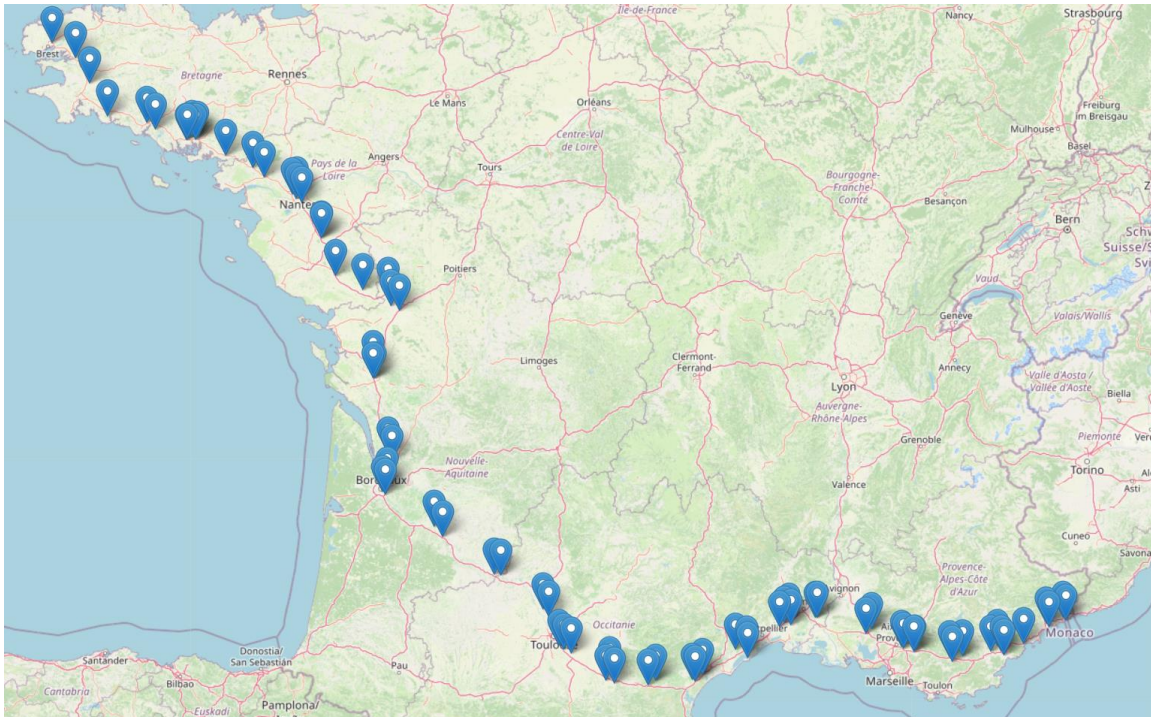


Figure 11 - Trajet Monaco-Brest

En observant la répartition des bornes pour différents trajets, comme Monaco-Brest (cf. figure 14), on constate que l'algorithme parvient à produire une répartition relativement homogène des bornes de recharge le long de la route. Cette régularité garantit un accès constant aux infrastructures de recharge sans créer de zones dépourvues de bornes sur le trajet.

Enfin, il est possible d'affiner l'algorithme en ajustant certains paramètres tels que le pas entre les points du trajet ou la distance maximale autorisée entre le trajet et une borne sélectionnée. Ces ajustements permettent d'optimiser encore davantage la sélection des bornes en équilibrant le nombre total de bornes retenues et la minimisation des détours nécessaires pour recharger le véhicule.

#### 4. Exemple de visualisation avec Folium

Une fois que l'algorithme a déterminé l'itinéraire optimal, il est important de le visualiser pour que l'utilisateur puisse facilement suivre le trajet et voir les arrêts de recharge. Pour cela, nous utilisons **Folium**, une bibliothèque Python qui permet de générer des cartes interactives.

Le programme génère une carte montrant :

- L'itinéraire principal entre A et B.
- Les arrêts de recharge, représentés par des marqueurs sur la carte.

##### 4.1. Résultats avec voiture ayant une grande autonomie

Pour mieux analyser l'impact de l'autonomie des véhicules électriques sur la planification d'un trajet, nous comparons le même itinéraire effectué avec deux types de voitures : l'une disposant d'une grande autonomie et l'autre d'une autonomie plus réduite. Cette comparaison permet de mettre en évidence les différences en termes de nombre d'arrêts pour la recharge, de temps total de trajet et d'optimisation du parcours.

Nous prenons comme cas d'étude un trajet entre Clermont-Ferrand et Paris. Pour une meilleure compréhension et une interaction plus intuitive, une interface graphique a été développée. Cette interface permet à l'utilisateur de sélectionner plusieurs paramètres essentiels, notamment :

- La marque du véhicule, comme Tesla.
- Le modèle précis, par exemple, une Tesla Model Y Propulsion Berlin 2022, qui possède une batterie d'une capacité de 85 kWh et une consommation moyenne de 17 kWh/100 km.

Dans cette interface graphique [9], avec l'aide de la bibliothèque **Tkinter**, une fois que l'utilisateur a sélectionné la marque de la voiture, les modèles spécifiques associés à cette marque seront automatiquement affichés (cf. **figure 15**). Cela permet à l'utilisateur de choisir un modèle précis de voiture parmi ceux disponibles dans la base de données, qui a été préalablement créée et contient les informations relatives à chaque véhicule, telles que l'autonomie, la consommation et d'autres caractéristiques techniques. Cette fonctionnalité facilite la sélection d'un véhicule tout en assurant la cohérence des données utilisées pour le calcul du trajet.

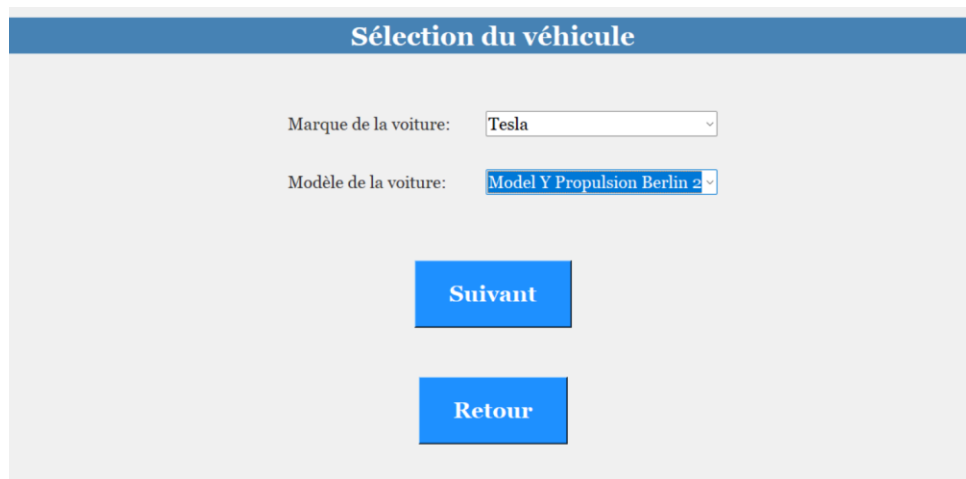


Figure 12 - Sélection du véhicule

Ensuite, l'utilisateur peut spécifier les paramètres du trajet en remplissant plusieurs champs importants :

- **Lieu de départ** : L'utilisateur renseigne la ville ou le point de départ du trajet.
- **Lieu d'arrivée** : L'utilisateur indique la destination du trajet.
- **Pourcentage de batterie au départ** : L'utilisateur définit le niveau de batterie de départ en pourcentage. Ce paramètre permet de calculer si des arrêts pour recharge sont nécessaires dès le départ.
- **Pourcentage de batterie à l'arrivée** : Ce champ permet à l'utilisateur de définir le niveau de batterie qu'il souhaite avoir à l'arrivée, ce qui peut influencer le choix des bornes de recharge et l'optimisation du trajet (cf. figure 16).

### Entrez les villes de départ et d'arrivée

Ville de départ:	<input type="text" value="Clermont-Ferrand"/>
Ville d'arrivée:	<input type="text" value="Paris"/>
Pourcentage au départ:	<input type="text" value="20"/>
Pourcentage à l'arrivée:	<input type="text" value="80"/>

Tracer l'itinéraire

Retour

Figure 13 - Critères pour le trajet

Une fois que l'utilisateur a rempli tous les paramètres nécessaires et que l'algorithme a effectué les calculs pour déterminer le trajet optimal, il est possible de **tracer l'itinéraire** sur une carte interactive (cf. figure 17). Ce tracé est présenté dans une interface en format **HTML**, permettant à l'utilisateur de visualiser le trajet en temps réel.



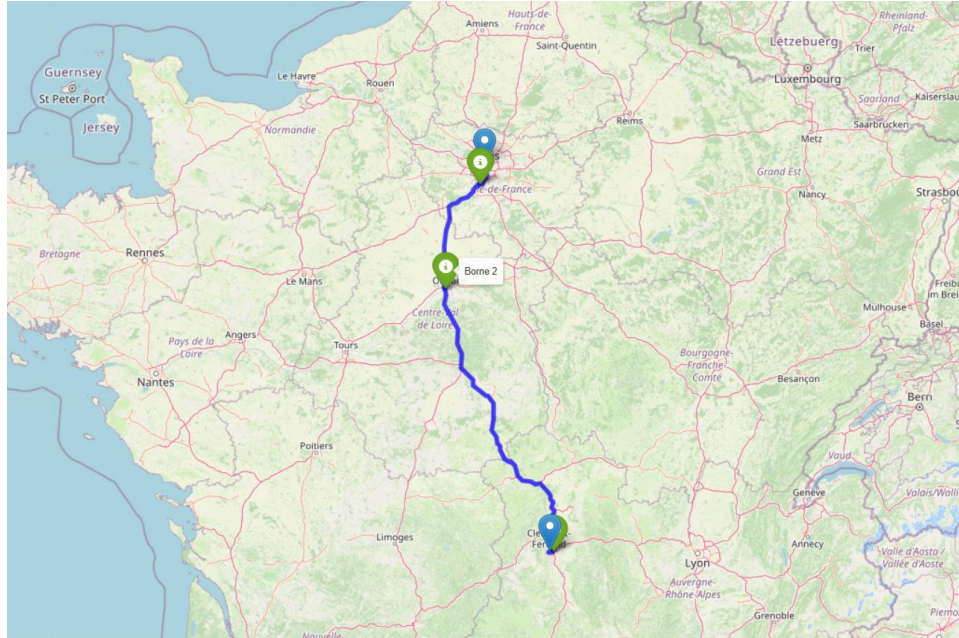


Figure 14 - Itinéraire du trajet Paris-Clermont

Dans la fenêtre graphique, l'utilisateur peut observer les différentes **étapes du trajet** (cf. figure 18), y compris les points de recharge et la distance restante avant chaque arrêt. En plus du tracé de la route, les **caractéristiques du trajet** sont également affichées, telles que :

- Le nombre de bornes de recharge nécessaires sur le parcours.
- Le temps estimé pour le trajet.
- Le pourcentage de batterie jusqu'auquel recharger.

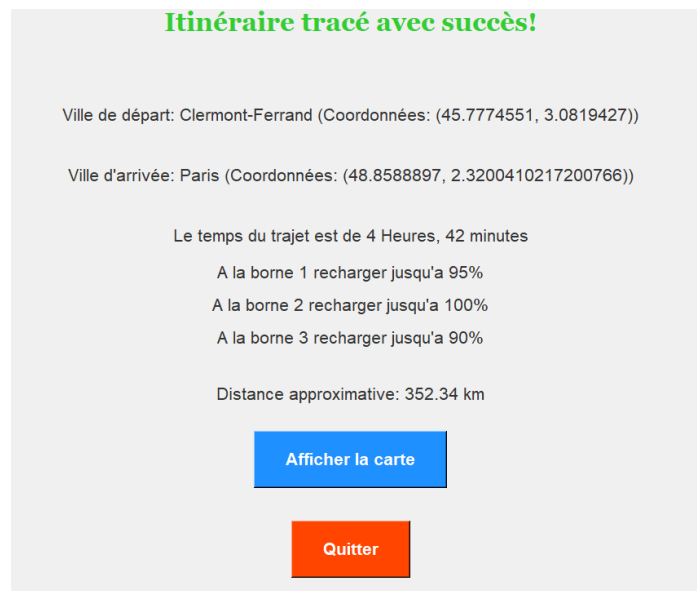


Figure 18 - Détails du trajet

Ainsi, l'interface graphique offre une vue d'ensemble complète de l'itinéraire, permettant à l'utilisateur de suivre facilement l'évolution du trajet et d'anticiper les besoins en recharge. Cela facilite la compréhension de l'optimisation du trajet pour chaque type de véhicule, tout en offrant un moyen simple et interactif de gérer les paramètres du voyage.

#### 4.2. Comparaison avec voiture ayant une petite autonomie

Pour la voiture à petite autonomie, nous avons choisi un autre exemple de véhicule électrique, la **Citroën C-Zero 2012**. Ce véhicule dispose d'une autonomie plus réduite, avec une batterie de **16 kWh** et une consommation de **12.5 kWh/100 km** d'après notre base de données.

Les paramètres du trajet restent identiques à ceux choisis pour la voiture à grande autonomie, afin de maintenir une comparaison équitable entre les deux véhicules. Nous utilisons donc les mêmes points de départ, d'arrivée, et les mêmes niveaux de batterie au départ et à l'arrivée.

Une fois ces paramètres définis, l'algorithme calcule l'itinéraire optimal pour ce véhicule, prenant en compte la moindre autonomie de la batterie et ajustant le nombre de recharges nécessaires tout au long du trajet.

Le résultat de ce calcul permet de déterminer les bornes de recharge nécessaires et les étapes du trajet. Ce calcul montre les différences en termes de **temps de trajet**, **nombre de recharges** et **optimisation des arrêts**, comparé à la voiture à grande autonomie (cf. figure 19 et 20).

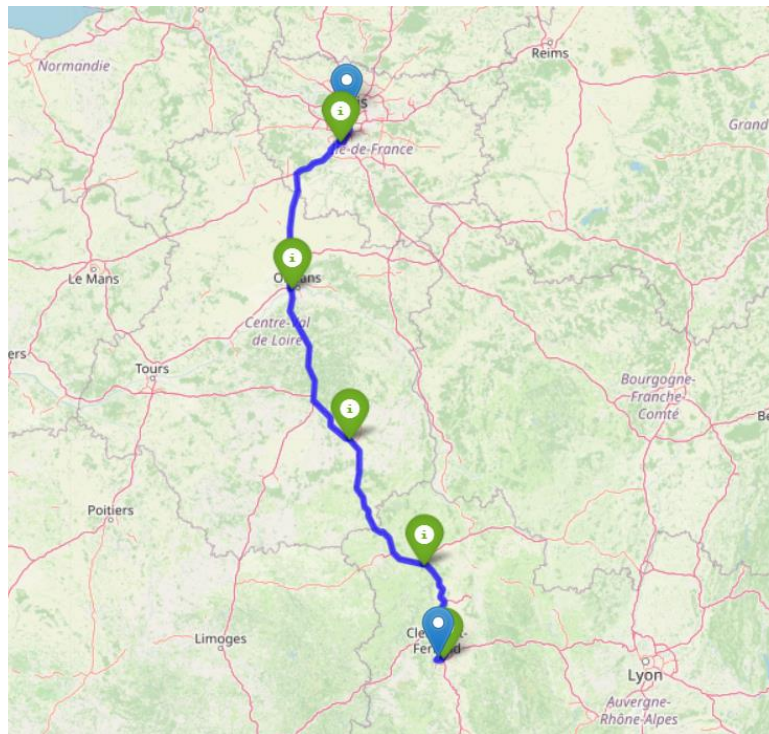


Figure 19 - Trajet Paris-Clermont avec une autonomie plus faible

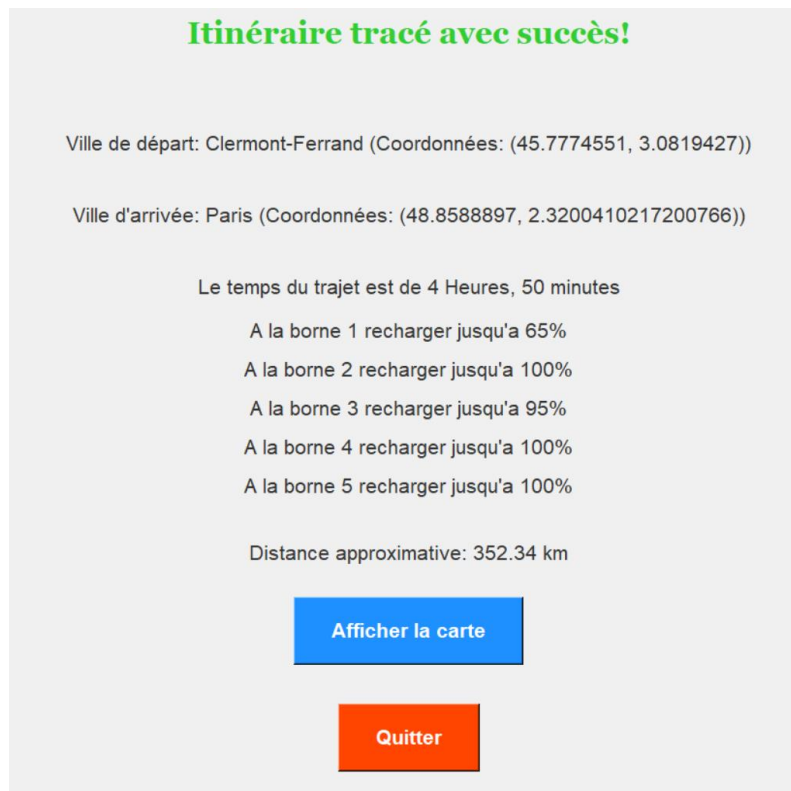


Figure 20 - Détails du trajet

Les résultats obtenus entre les deux véhicules, la **Tesla** avec une grande autonomie et la **Citroën C-Zero** avec une autonomie plus réduite, montrent des différences cohérentes et logiques. La Tesla, avec une batterie de 85 kWh, nécessite trois recharges durant le trajet Clermont-Ferrand – Paris. Cela est relativement faible par rapport à la distance parcourue et montre l'efficacité d'un véhicule avec une grande autonomie pour des trajets longs. En revanche, la Citroën C-Zero, qui dispose d'une autonomie de 16 kWh, doit effectuer cinq recharges pour le même trajet. Cette différence est attendue, car la capacité de la batterie de la Citroën étant plus faible, elle doit être rechargée plus souvent, ce qui entraîne un nombre plus élevé de pauses pour la recharge.

Ce besoin de plus de recharges a également un impact sur le temps total du trajet. La Citroën met environ huit minutes de plus que la Tesla pour arriver à destination. Cette différence de temps s'explique par la fréquence plus élevée des arrêts de recharge. Bien que chaque arrêt de recharge puisse être relativement rapide, ces arrêts plus fréquents s'accumulent et augmentent le temps total nécessaire pour compléter le trajet.

Ainsi, même si le temps supplémentaire n'est pas énorme sur un trajet de longue distance, il devient significatif lorsqu'on le compare au trajet effectué par un véhicule avec une autonomie plus grande.

En outre, ces résultats montrent clairement l'importance de l'autonomie de la batterie sur le temps total d'un trajet. Plus l'autonomie du véhicule est grande, moins il est nécessaire de s'arrêter pour recharger, ce qui permet de réduire le temps global du trajet. Pour des trajets comme celui de Clermont-Ferrand à Paris, la différence d'autonomie entre les deux véhicules se traduit par une réduction significative du nombre d'arrêts et donc du temps de trajet. Cela met en évidence l'efficacité des véhicules à forte autonomie, particulièrement pour les trajets longs, où les arrêts fréquents deviennent un facteur important dans la gestion du temps de conduite.



## 5. Axes d'améliorations

### 5.1. Analyse des Performances du Programme

Afin d'optimiser le temps de calcul de notre programme, nous avons procédé à une analyse détaillée de ses performances à l'aide de l'outil **CProfile** [10]. Cela permet de mieux comprendre où le programme passe le plus de temps et d'identifier les zones à améliorer.

Prenons l'exemple du trajet Monaco – Brest :

- Nombre de points dans le trajet : 17 939
- Nombre de bornes après échantillonnage : 72

```
5695519 function calls (5695270 primitive calls) in 12.944 seconds
```

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.112	0.112	12.943	12.943	c:\Users\loicd\Desktop\Projet\Fonction_trajet.py:14(tracer_trajet2)
5	0.000	0.000	7.408	1.482	C:\Users\loicd\anaconda3\Lib\site-packages\requests\sessions.py:500(request)
5	0.000	0.000	7.396	1.479	C:\Users\loicd\anaconda3\Lib\site-packages\requests\sessions.py:673(send)
68	0.001	0.000	6.975	0.103	C:\Users\loicd\anaconda3\Lib\socket.py:694(readinto)
3	0.000	0.000	6.962	2.321	C:\Users\loicd\anaconda3\Lib\site-packages\requests\api.py:62(get)
3	0.000	0.000	6.961	2.320	C:\Users\loicd\anaconda3\Lib\site-packages\requests\api.py:14(request)
5	0.000	0.000	6.928	1.386	C:\Users\loicd\anaconda3\Lib\site-packages\requests\adapters.py:535(send)
5	0.000	0.000	6.923	1.385	C:\Users\loicd\anaconda3\Lib\site-packages\urllib3\connectionpool.py:594(urlopen)
5	0.000	0.000	6.915	1.383	C:\Users\loicd\anaconda3\Lib\site-packages\urllib3\connectionpool.py:379(_make_request)
66	6.719	0.102	6.719	0.102	{method 'recv_into' of 'socket.socket' objects}
5	0.000	0.000	6.540	1.308	C:\Users\loicd\anaconda3\Lib\site-packages\urllib3\connection.py:438(getresponse)
5	0.000	0.000	6.537	1.307	C:\Users\loicd\anaconda3\Lib\http\client.py:1384(getresponse)
5	0.000	0.000	6.537	1.307	C:\Users\loicd\anaconda3\Lib\http\client.py:324(begin)
5	0.000	0.000	6.533	1.307	C:\Users\loicd\anaconda3\Lib\http\client.py:291(_read_status)
52	0.000	0.000	6.532	0.126	{method 'readline' of '_io.BufferedReader' objects}
394	0.255	0.001	2.261	0.006	c:\Users\loicd\Desktop\Projet\itineraire_fonction.py:56(points_km)
667327	1.391	0.000	2.002	0.000	c:\Users\loicd\Desktop\Projet\itineraire_fonction.py:33(haversine)
1	1.763	1.763	1.763	1.763	c:\Users\loicd\Desktop\Projet\itineraire_fonction.py:106(temps_trajet)
2	0.000	0.000	1.044	0.522	c:\Users\loicd\Desktop\Projet\itineraire_fonction.py:24(get_coordinates)
174	0.663	0.004	0.663	0.004	{method 'fetchall' of 'sqlite3.Cursor' objects}
2	0.000	0.000	0.596	0.298	C:\Users\loicd\anaconda3\Lib\site-packages\geopy\geocoders\nominatim.py:53(__init__)
2	0.000	0.000	0.596	0.298	C:\Users\loicd\anaconda3\Lib\site-packages\geopy\geocoders\base.py:219(__init__)

Figure 21 - Temps passé dans le programme

L'analyse des résultats (cf. figure 21) montre que le principal facteur ralentissant l'exécution du programme est lié aux appels réseau, notamment les méthodes **requests**, **recv\_into**, et **get**. En comparaison, des fonctions comme **haversine** (calculant la distance entre deux points) et **temps\_trajet**, qui représentent à elle seule plus de 12 secondes d'exécution totale, prennent beaucoup moins de temps et ont un impact marginal sur les performances globales.



Malgré un nombre important d'appels à la fonction haversine (667 327 appels), la perte de temps la plus significative provient donc de l'interaction avec l'API OSRM. Afin d'améliorer l'efficacité de l'algorithme, il est pertinent de se concentrer sur l'optimisation des appels réseau. Plusieurs pistes peuvent être envisagées, telles que l'amélioration de l'efficacité des requêtes réseau ou la mise en place de processus asynchrones. Cela permettrait de réduire considérablement le temps d'attente lié aux interactions avec l'API et, par conséquent, d'optimiser le temps total de calcul du programme.

## 5.2. Affinement du Modèle en Fonction des Paramètres Réels

Dans le cadre de notre projet, certains paramètres ont été omis, soit pour simplifier l'avancement du projet, soit à cause des difficultés rencontrées pour obtenir ces données sur internet. Cependant, pour rendre notre modèle encore plus précis et proche de la réalité, plusieurs éléments doivent être ajoutés.

Tout d'abord, il est essentiel de **prendre en compte les différentes prises des bornes de recharge**. En effet, toutes les voitures ne sont pas compatibles avec toutes les bornes disponibles. Il serait donc nécessaire de vérifier la compatibilité des prises entre le véhicule et la borne pour un calcul plus réaliste du temps de recharge et éviter des erreurs liées à l'incompatibilité.

Ensuite, il serait pertinent d'ajouter une contrainte supplémentaire en permettant à l'utilisateur de **définir un prix maximum** qu'il souhaite ne pas dépasser pour le trajet, en fonction des abonnements ou des coûts liés à la recharge. Cette fonctionnalité permettrait de rendre l'outil plus adapté aux besoins des utilisateurs, en tenant compte des coûts associés aux trajets en véhicule électrique.

Enfin, dans notre modèle, nous avons supposé **un temps de recharge linéaire**. Or, en réalité, le temps de recharge varie en fonction du type de véhicule, de la batterie, de l'état de la batterie et d'autres facteurs. Par exemple, les derniers pourcentages de batterie, entre 80% et 100%, nécessitent généralement plus de temps de recharge car les batteries ralentissent leur charge à mesure qu'elles se remplissent pour éviter d'endommager la cellule. Il serait donc nécessaire d'intégrer une fonction qui calcule le temps de recharge réel en fonction des caractéristiques du véhicule utilisé.

Ces améliorations nécessiteront de renforcer les bases de données, notamment celles des véhicules et des bornes de recharge, afin de fournir plus de détails sur chaque élément (par exemple, les types de prises disponibles et les temps de recharge spécifiques à chaque véhicule et borne). Ce travail supplémentaire permettrait d'offrir un modèle encore plus fiable et précis pour la planification de trajets avec des véhicules électriques.

### 5.3. Optimisation des Performances et Précision des Algorithmes

Comme observé précédemment, certaines des méthodes utilisées dans le projet nécessitent un temps de calcul relativement élevé. Ces méthodes peuvent être ajustées ou remplacées par des approches plus efficaces pour optimiser les performances et la précision de notre système.

L'une des premières améliorations possibles concerne la **récupération des bornes dans les subdivisions** de la France. Actuellement, la France est divisée en 25 carrés égaux, mais en divisant le pays en un plus grand nombre de subdivisions, il serait possible de **réduire le nombre de bornes récupérées** tout en améliorant la précision de celles qui sont réellement proches du trajet initial. Cela permettrait de gagner du temps de calcul en se concentrant uniquement sur les bornes pertinentes et proches du trajet, tout en maintenant la qualité des résultats.

En ce qui concerne le **sampling des bornes**, il existe d'autres méthodes plus rapides et plus précises pour choisir les bornes les plus pertinentes autour du trajet. L'approche actuelle pourrait être améliorée afin de réduire le nombre de calculs nécessaires, tout en obtenant des résultats encore plus précis.

De plus, le programme donne des résultats cohérents et satisfaisants dans l'ensemble. Cependant, il n'est pas encore suffisamment optimisé et précis pour être utilisé dans un contexte réel, principalement en raison de l'absence de certains paramètres clés. **Avoir la base de données OpenStreetMap en local** pourrait considérablement réduire le temps de calcul et améliorer l'efficacité de l'application. Comme nous l'avons constaté, la principale source de perte de temps réside dans les échanges avec l'API OSRM (**cf. figure 21**).

En résumé, notre algorithme offre une solution efficace et pertinente pour le calcul des trajets optimisés avec bornes de recharge, en prenant en compte la consommation d'énergie et les temps de recharge. Les résultats obtenus sont cohérents et satisfaisants, permettant d'établir un trajet réaliste entre différentes destinations. Toutefois, il reste des axes d'amélioration pour augmenter la précision et optimiser les performances. Par exemple, la prise en compte des différentes prises de bornes et l'adaptation des temps de recharge selon les spécifications du véhicule contribueraient à affiner davantage les résultats. Avec ces améliorations, notre algorithme pourra se rapprocher encore plus de la réalité et fournir une expérience utilisateur optimale, tout en conservant sa fiabilité et son efficacité.



## Conclusion Générale

En conclusion, le projet IMDS4A nous a permis d'explorer en profondeur les enjeux de la planification de trajets pour véhicules électriques et d'apporter une solution optimisée grâce à l'utilisation de données ouvertes et d'algorithmes avancés. En intégrant des bases de données complètes sur les véhicules électriques et les bornes de recharge, en découpant le territoire en zones géographiques et en développant un algorithme performant de planification, nous avons pu proposer une approche innovante et adaptée aux besoins des conducteurs de VE.

Les résultats obtenus ont validé notre méthodologie, avec un calcul d'itinéraires tenant compte des contraintes d'autonomie, des temps de recharge et de l'optimisation du temps de trajet. L'utilisation d'outils comme SQLite, OSRM [11] et Folium a permis de structurer et de visualiser les données de manière efficace, offrant ainsi une interface à la fois intuitive et interactive.

Cependant, des pistes d'amélioration subsistent, en particulier concernant l'optimisation des performances, l'affinement des modèles de recharge, ainsi que l'intégration de nouveaux paramètres tels que la compatibilité des prises ou les coûts de recharge. Ces perspectives ouvrent la voie à des évolutions futures, pouvant renforcer l'efficacité et la précision de notre solution.

Ce projet nous a offert une expérience enrichissante en combinant théorie et pratique, nous permettant de développer des compétences avancées en gestion de données, optimisation algorithmique et développement d'applications intelligentes. Il constitue ainsi une base solide pour des recherches et des projets futurs dans divers domaines liés à l'optimisation et au traitement des données.

## *Bibliographie*

- [1] Base de données des voitures électriques : <https://www.fiches-auto.fr/articles-auto/voiture-electrique/s-2531-fiches-techniques-voitures-electriques.php> , consulté en septembre 2024
- [2] Sqlite3 : <https://docs.python.org/3/library/sqlite3.html> , consulté en octobre 2024
- [3] Beautiful Soup : <https://beautiful-soup-4.readthedocs.io/en/latest/> , consulté en octobre 2024
- [4] Request : <https://docs.python-requests.org/en/latest/modules/requests/api/> , consulté en octobre 2024
- [5] SQLite Studio : <https://sqlitestudio.pl/> , consulté en octobre 2024
- [6] Base de données des bornes : <https://www.data.gouv.fr/fr/datasets/fichier-consolide-des-bornes-de-recharge-pour-vehicules-electriques/> , consulté en septembre 2024
- [7] Folium : <https://python-visualization.github.io/folium/latest/> , consulté en octobre 2024
- [8] API OSRM : <https://project-osrm.org/docs/v5.5.1/api/#general-options> , consulté en octobre 2024
- [9] Interface Graphique : <https://docs.python.org/3/library/tkinter.html> , consulté en janvier 2025
- [10] Profilers : <https://docs.python.org/3/library/profile.html> , consulté en décembre 2024
- [11] OpenStreetMap : [www.openstreetmap.org](http://www.openstreetmap.org) , consulté en septembre 2024
- [12] Python : <https://www.python.org/>
- [13] Visual Studio Code : <https://code.visualstudio.com/>