BORDEAUX INP Enseirb-Matmeca

Rapport de projet avancé :

Flight Tracker

Auteurs :

Grégoire DOMER

Théo Chobert

Renaud Barrau

Melvyn Priam

Département : Electronique

Année universitaire : 2023-2024

Professeur :

Romain Tajan

Table des matières

I Introduction

L'objectif du projet avancé est de réaliser un simulateur de flight tracker (suiveur d'avion) à l'aide d'une bibliothèque python AFF3CT.

I.1 Présentation d'AFF3CT

AFF3CT est un simulateur et une bibliothèque dédiée au codage de canal. Il s'agit d'une librairie écrite en C++ qui supporte une grande gamme de codes correcteurs d'erreurs (codes Turbo, codes polaires, Low-density Parity-Check [LDPC]) et qui se base sur des simulations de Monte Carlo. AFF3CT peut s'utiliser en ligne de commande et permet d'assembler des chaînes de communication de la manière que l'on souhaite. C'est cette fonctionnalité qui nous a permis de créer le suiveur d'avion.

En effet, AFF3CT permet de créer des modules, tâches et sockets qui sont éléments fondamentaux de l'architecture des chaîne de simulations. Ensemble, ces concepts permettent de construire des simulations de systèmes de communications numériques en assemblant de manière modulaire des tâches dans des modules, en utilisant les sockets pour permettre la communication des séquences (données) entre ces modules (1).

I.1.a Module :

- Un module représente une opération ou une fonction spécifique dans une simulation.
- Il est conçu pour être flexible et réutilisable.
- Les modules accomplissent des tâches spécifiques, comme l'encodage, la détection d'erreurs, la modulation, etc.

I.1.b Tâche :

- Une tâche est une unité de travail dans AFF3CT.
- Les tâches abstraient le flux de traitement dans une simulation, définissant comment les données sont combinés et traitées pour atteindre un objectif.

I.1.c Socket :

- Une socket est une interface de communication entre les modules.
- Les modules se connectent via leurs sockets d'entrée et de sortie pour transférer des données entre eux.
- Un module peut avoir plusieurs sockets d'entrée et de sortie pour gérer les échanges de données complexes.

I.1.d Séquences :

- Une séquence est une suite de données transmis entre les modules.
- Elles permettent de modéliser fidèlement les flux de données.
- Elles peuvent être de différents types.

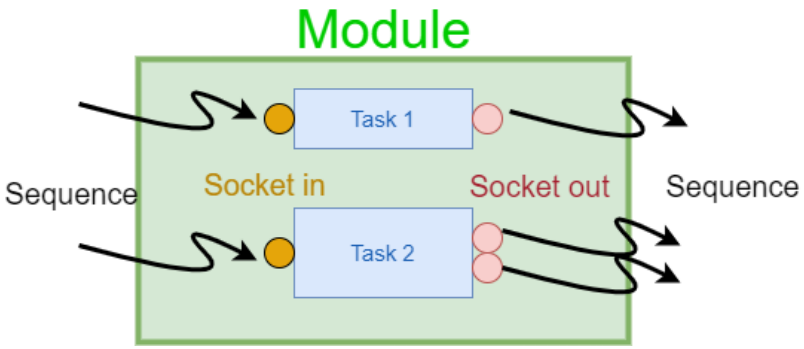

FIGURE 1 – Exemple de module AFF3CT

I.2 Cahier des charges

Le cahier des charges est le suivant :

- ✈ Un avion doit émettre une trame toutes les 2 secondes (2).
- ✈ Chaque trame doit durer 120 μs (2).
- ✈ Chaque trame doit contenir :
  - ✎ Le temps auquel elle a été émise.
  - ✎ L'identifiant de l'avion qui l'émet.
  - ✎ La longitude.
  - ✎ La latitude.
  - ✎ L'altitude.

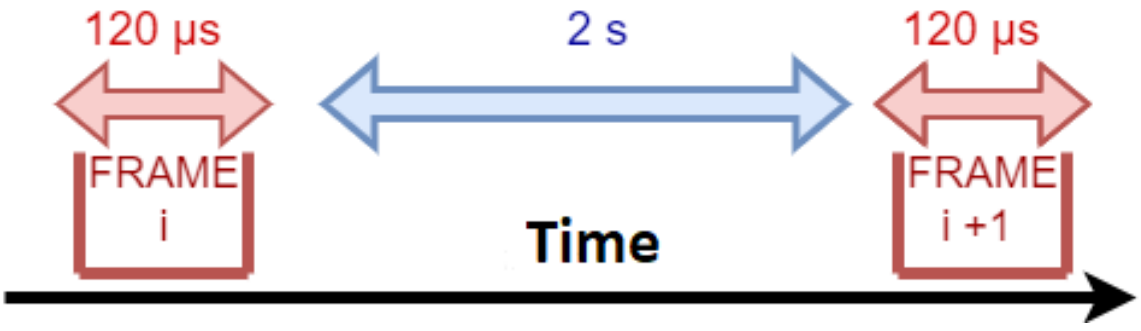

FIGURE 2 – Spécifications temporelles d'une trame

- ✈ Lorsque deux trames sont perçues par le récepteur en même temps (collision), ces dernières doivent être déclarées comme telle et ne sont pas traitées (3).

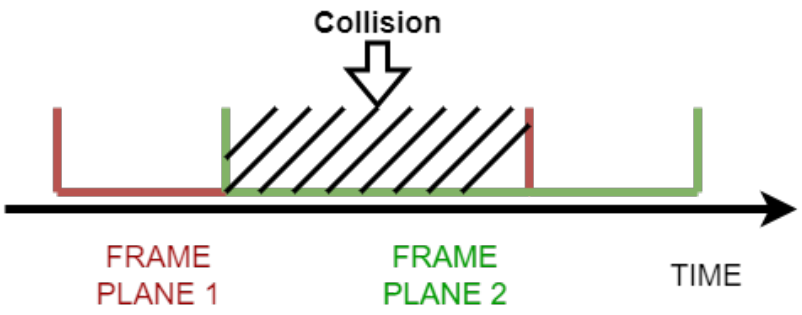

FIGURE 3 – Gestion des collisions entre deux trames émises par des avions différents.

- ✈ Affichage Temps réel sur une carte dynamique des trajectoires d'avions (1 μs pour le pas d'affichage) (4).
- ✈ Le tracker doit être capable de suivre de nombreux avions en même temps (une centaine) (4).

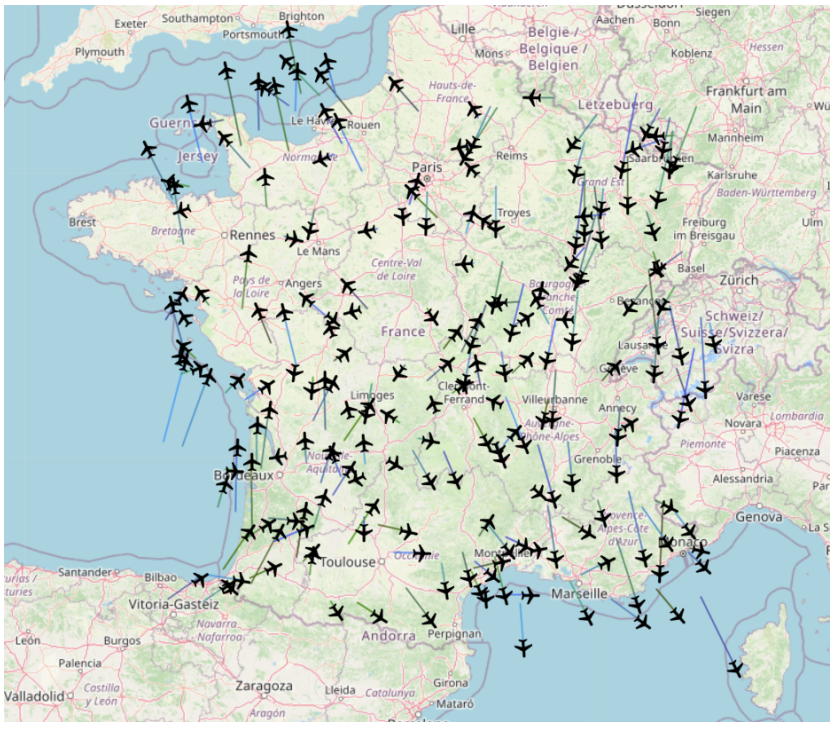

FIGURE 4 – Affichage dynamique des trajectoires d'avions

I.3 Architecture proposée

L'idée ici est de réaliser le suivi d'une simulation de trafic aérien sur une carte de la france. Pour cela, nous avons diviser le travail en 3 modules (5) :

- Le premier module génère des faux avions. Autrement dit, il génère un fichier csv de test qui indiquera, pour chaque avion, les trames qui doivent être émises par ces derniers durant toute la simulation. Ce module est complété par un bloc qui encodera ces trames de telle sorte qu'elles soient comprises et utilisables par le second module.
- Le second module consiste à gérer les collisions entre des trames émises par différents avions et perçues par le récepteur sur un même créneau de temps. Le résultat de ces collisions est stocké dans un fichier.
- Le dernier module correspond au module d'affichage sur carte dynamique développée en Javascript.

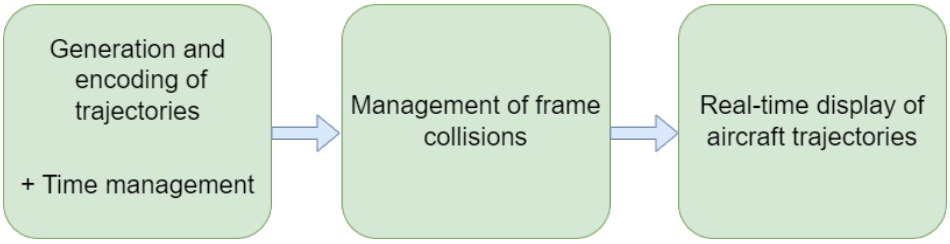

FIGURE 5 – Séparation des tâches à effectuer

Plusieurs difficultés se sont présentées devant nous durant le projet : prise en main d'AFF3CT, gestion des collisions de trame, formatage efficace et intelligent des données, beaucoup de trajectoires d'avions à gérer en même temps sous python (qui est peu performant et non parallélisable).

II Module 1 :

Ce premier module a pour objectif de créer un fichier csv de test et d'envoyer les trames des différents avions au second module après les avoir encodées.

II.1 Fichier CSV

La première étape à effectuer pour notre chaîne de simulation est de générer des trajectoires pour chacun de nos avions. Les différentes trajectoires d'avions sont enregistrées les unes à la suite des autres dans le fichier. Chaque nouvelle ligne du fichier correspond à une nouvelle position de l'avion 2 secondes après la précédente. Pour des raisons de simplifications, nous considérons qu'un avion a une vitesse rectiligne uniforme. Ainsi, Une trajectoire d'avion est déterminer selon cette procédure :

- Déterminer la position initiale. On choisit une longitude et latitude au hasard comprises dans celles de la France ainsi qu'un altitude comprise entre 1 000 et 5 000m.
- On choisit un angle pour la direction (par rapport à l'équateur).
- On détermine un pas de calcul sur la longitude et la latitude en fonction du temps de simulation désiré.
- On détermine les nouvelles coordonnées pour chaque pas de temps (2 secondes) et on écrit les nouvelles coordonnées dans le fichier CSV selon ce format : Temps, ID, Longitude, Latitude, altitude.
- On réitère ces calculs pour chaque avion. Chaque avion possède un ID qui lui est propre.

Nous obtenons ainsi un fichier CSV de ce format :

```
0,0,-0.6405574912485612,42.713396069588256,39690
2000000,0,-0.6373779092184977,42.71657565161832,39690
4000000,0,-0.634198327188434,42.719755233648385,39690
6000000,0,-0.6310187451583705,42.72293481567845,39690
8000000,0,-0.627839163128307,42.726114397708514,39690
10000000,0,-0.6246595810982435,42.72929397973857,39690
12000000,0,-0.6214799990681799,42.732473561768636,39690
14000000,0,-0.6183004170381163,42.7356531437987,39690
16000000,0,-0.6151208350080528,42.738832725828765,39690
18000000,0,-0.6119412529779893,42.74201230785883,39690
20000000,0,-0.6087616709479257,42.74519188988889,39690
22000000,0,-0.6055820889178621,42.74837147191896,39690
24000000,0,-0.6024025068877986,42.75155105394902,39690
26000000,0,-0.599222924857735,42.75473063597908,39690
```

FIGURE 6 – Format du fichier CSV.

II.2 Encodage des données

La deuxième étape de notre chaîne de simulation est l'encodage des données sous un format spécifique pour être envoyées dans la *socket out* en direction du prochain module qui gère les collisions. Nous avons choisi d'envoyer la trame toutes les microsecondes pendant 120 microsecondes. Ainsi nous lisons le fichier CSV précédemment créé et nous encodons les données selon la procédure suivante :

- Trouver la ligne où l'ID de l'avion choisi apparaît pour la première fois avec son temps de départ *t0*.
- Si le temps est entre *t0* et *t0+119*, envoie de la trame encodée.

- ✈ Si le temps est de *t0+120*, passage à la ligne suivante et changement de la valeur de *t0*.
- ✈ Si le temps est supérieur à celui de simulation, envoie d'une trame remplie de F.
- ✈ Sinon, envoie d'une trame remplie de 0.
- ✈ Incrémentation du temps de *1µs*, et ainsi de suite jusqu'à la fin de la simulation.

Le format de la trame est la suivante :

```
txt = "<{TEMPS:010d}us_{IDD:06d}_{LONGITUDE:+010.6f}_{LATITUDE:+010.6f}_{ALTITUDE:05d}feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!>"
```

Figure 7 – Format de la trame.

Elle est constituée de 120 caractères avec 2 caractères de séparation pour distinguer le début et la fin d'une trame.

Le module étant complet, nous le simulons et nous obtenons le résultat escompté : les trames sont bien envoyées. Mais le temps pour encoder les trajectoires et les envoyer au prochain module par le biais de la *socket out* prend beaucoup de temps. En effet, nous mettons 10 minutes pour générer 1 minute de simulation avec 2 avions.

Le bloc a été entièrement optimisé en Python, passer en C++ donnerait donc un meilleur résultat.

III Module 2 : Gestion des collisions

III.1 Description de la tâche à réaliser

L'objectif de ce module est de simuler les collisions entre des trames d'avions. Ces collisions apparaissent lorsque 2 trames sont reçues simultanément, entièrement ou partiellement.
La figure 14 illustre ce principe de collision, avec en bleu, 2 trames envoyées à un instant trop proche, résultant d'une collision en rouge. Cette collision a comme conséquence de rendre inutilisable le reste des données transmises, ici en orange.

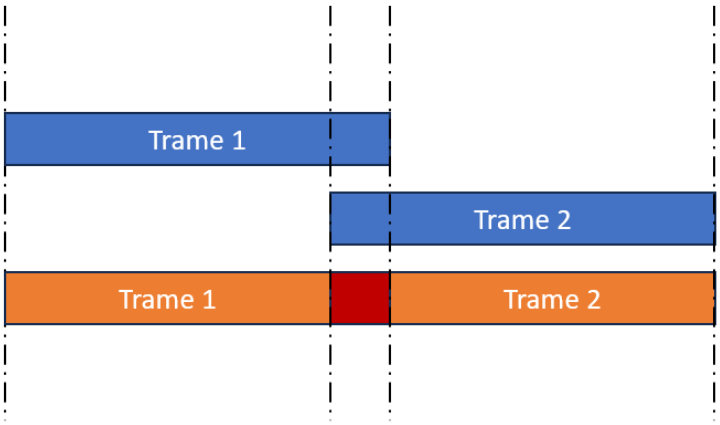

FIGURE 8 – Exemple de collision entre 2 trames

Afin de pouvoir arrêter automatiquement la simulation, nous avons défini 3 types de trames que les avions peuvent transmettre :

- Une trame de positionnement, contenant les informations identiques à celle du fichier csv des avions,
- Pas de données à transmettre (remplie de 0),
- Fin des données (remplie de -1, soit 0xFF pour un entier sur 8 bits signé).

Grâce à celles-ci, les avions peuvent transmettre ou non des données et indiquer leur fin de transmission. Ensuite, il suffit de vérifier si tous les avions on fini de transmettre pour arrêter la simulation.

III.2 Développement du module de collisions en Python

La tâche à effectuer pour le module est plutôt simple car il s'agit d'une somme. Il faut cependant faire attention à ne pas prendre en compte les trames indiquant la fin de transmission des avions.

Pour détecter les collisions, il suffit de savoir si plusieurs données sont présentent en même temps. Si plusieurs données sont transmissent en même temps, on transmet -2 (0xFE), différencié de -1 (0xFF) qui pourrait être interprété comme une fin de transmission, s'il arrivé au début de la trame.

Après plusieurs améliorations algorithmiques, le code de la tâche AFF3CT est devenu celui ci-dessous :

```
def __sum(self, inp, y):
    array_reshaped = []
```

```
    for i in range(0, self.n_avions):
        # Ne prend pas en compte les avions termin s
        if inp[i][0,0] != -1:
            array_reshaped.append(inp[i][0])

    # S'il n'y a plus d'avions a traiter
    if len(array_reshaped) == 0:
        y[0][:] = -1
        return -1 # met status a -1

    array_summed = np.sum(array_reshaped, axis=0, dtype=np.int32)
    array_nzero = np.count_nonzero(array_reshaped, axis=0)

    y[0] = np.where(array_nzero <= 1, array_summed, -2)
    return 0
```

Après avoir implémenté le code en Python, nous avons rencontré un problème de performances. Il nous a donc fallut passer sur une option plus efficace comme le C++ ;

III.3 Portage en C++

Nous portons le module en C++ afin améliorer ses performances calculatoires, car le Python rajoute une importante sur-couche, ralentissant l'exécution.

Après ce portage, nous constations une grande réduction du temps de calcul, passant de 174 secondes d'exécution, à seulement 20, soit 8 fois plus rapide.

MODULE	TASK	CALLS	TIME (s)	PERC (%)
py_sum	sum	58000002	174.13	45.80
py_PLANE	generate_plane	58000002	104.10	27.38
py_PLANE	generate_plane	58000002	99.32	26.13
py_save_c	save	58000002	2.58	0.61
TOTAL	*	58000002	380.13	100.00

MODULE	TASK	CALLS	TIME (s)	PERC (%)
py_PLANE	generate_plane	58000002	102.68	45.97
py_PLANE	generate_plane	58000002	98.00	43.88
py_sum_c	sum	58000002	20.07	8.99
py_save_c	save	58000002	2.60	1.16
TOTAL	*	58000002	223.34	100.00

FIGURE 9 – Comparaison du gain des performances du module de collisions

IV Module 3 : Sauvegarde des données

Ce nouveau module est la suite direct du module de collision. Il formate les données post-collisions pour pouvoir être enregistrées dans un fichier texte. Il converti les octets entrés en collision en '#', les trames vides par des espaces et garde le reste tels quels.

IV.1 Développement du module de sauvegarde en Python

Dans un premier temps, nous avons tenté d'enregistrer chaque trame sur une ligne dans le fichier, mais nous nous sommes vite confronté à un problème de taille. En effet, avec une résolution de 1 Trame/μs, nous générons 120 Mo/s de donnés. De plus, il inclus uniquement 120 Trames pour 2 000 000 de lignes, ce qui est très inefficace. Pour lutter contre ce problème, une option est proposé pour ne pas enregistrer les données, mais seulement effectuer les calcules. Cependant, la vitesse d'exécution est encore trop lente, limitant les autres modules.

Le code de cette tâche est donné ci-dessous :

```
def __save(self, x, stop):
    if stop[0,0] != -1:
        if self.dont_save:
            return 0
        if(x[0, 0] != 0):
            self.file.write(self.parseData(x[0]) + os.linesep)
    else:
        self.file.close()
        self.toggle_done()

    return 0

def parseData(self, data):
    data[data == -1] = ord('#')
    data[data == 0] = ord('␣')
    return str(data, 'ascii')
```

Afin de palier à ces différents problèmes un portage en C++ a été décider.

IV.2 Portage en C++

Ce portage à comme objectif d'améliorer le temps de calcule et de réduire l'utilisation de l'espace disque. L'objectif de l'optimisation est de renseigner le nombre de trames vide sur une ligne, au lieu de les écrire. De cette manière, on limite le nombre d'écritures sur le disque, réduisant la taille du fichier de sortie, ainsi que de d'éviter l'attente que cette lecture soit complétée par le programme. Un exemple de rendu est visible sur la figure 10 dissociant les trames des lignes vides.

Avec cette modification, nous réduisons le temps de calcule de 220 secondes à seulement 9, soit une réduction par 25. De plus, le fichier de sortie est maintenant plus que de 800ko.

```
113 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
114 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
115 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
116 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
117 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
118 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
119 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
120 <0000000000us_000000_-02.556567_+47.640011_26225feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
121 [999880 x NOTHING
122 <0001000000us_000001_-02.154075_+46.290141_48721feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
123 <0001000000us_000001_-02.154075_+46.290141_48721feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
124 <0001000000us_000001_-02.154075_+46.290141_48721feet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

FIGURE 10 – Partie du fichier de sorti avant formatage

V Module 4 : Affichage des trajectoires

Ce dernier module a pour objectif d'apporter une représentation visuelle des trames que nous traitons qui sont pour le moment sous la forme de lignes dans un fichier texte.

V.1 Carte dynamique

Le but est d'afficher une carte du monde permettant de visualiser les différentes trajectoires des avions actuellement en vols, avec un rafraîchissement automatique des données. Parmi les différents langages de programmation, le Javascript sera celui retenu pour réaliser ce module, notamment grâce aux librairies disponibles simplifiant la tâche

En effet, la librairie Leaflet en Javascript permet l'affichage d'une carte du monde dynamique ainsi que d'autres éléments que l'on peut superposer (logo, marqueurs, lignes, etc...).

FIGURE 11 – Carte sur navigateur web

V.2 Extraction des données

Pour afficher les différentes coordonnées de chaque avion, nous devons ouvrir et récupérer les données du fichier texte contenant les trames. Pour pouvoir être interprété en Javascript, ce dernier doit-être légèrement modifié. Le fichier que nous importons depuis l'interface web possède la structure suivante :

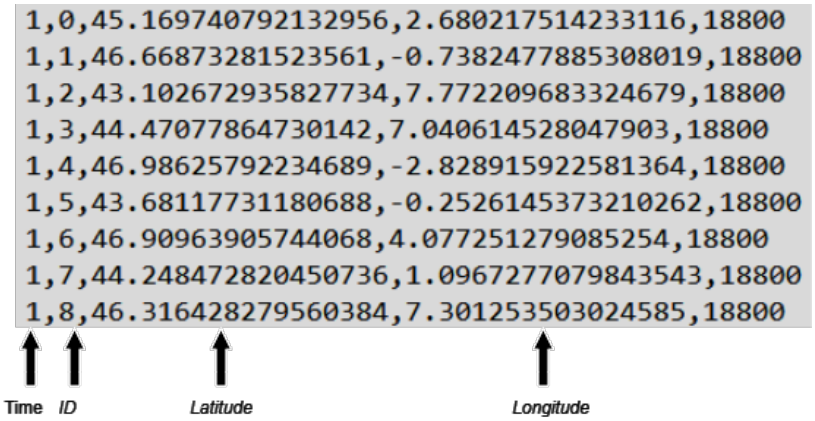

FIGURE 12 – format des trames

Nous retrouvons sur chaque ligne :

- Le temps relatif à l'émission de la trame (t=1 au départ)
- L'ID de l'avion qui permet de l'identifier. Ce paramètre sera utile lors de la création de "polylines" dans la section suivante
- Les coordonnées sur la carte avec la latitude et la longitude de l'avion
- Le dernier paramètre est l'altitude, mais ce dernier ne sera pas exploité ici étant donnée que la carte est en 2D.

Pour extraire toutes ces données, nous allons ouvrir et lire le fichier texte afin de récupérer les champs cités précédemment dans chaque ligne. Nous stockons toutes ces valeurs dans des tableaux tampons afin de pouvoir les réutiliser avec les polylines.

V.3 Exploitation et données et animation

Pour réaliser une animation, nous allons utiliser une propriété de Javascript intitulée "setInterval" permettant d'appeler une fonction toutes les X secondes. Cette fonction met à jour l'affichage en ajoutant les points correspondants aux polylines.

Les polylines sont des éléments graphiques inclus dans la librairie Leaflet et permettent de tracer des lignes superposées à la carte déjà présente. Il suffit d'ajouter des points à la ligne avec ses coordonnées. Ainsi, toutes les secondes, une fonction de mise à jour est appelée et permet de rajouter des nouvelles coordonnées sur la carte.

```
function update_polylines(){
    if(offset < T_id.length){
        for(compteur = 0; compteur < payload; compteur++){
            polyline_list[T_id[compteur + offset]].addLatLng([T_lat[compteur + offset],T_long[compteur + offset]]);
            if (offset > 0){
                var angle = calculateBearing([T_lat[compteur + offset - payload],T_long[compteur + offset - payload]], [T_lat[compteur + offset],T_long[compteur + offset]
            }
            console.log(angle);
            var planeMarker = new L.Marker([T_lat[compteur + offset],T_long[compteur + offset]], { icon: planeIcon, rotationAngle: angle});
            plane_array.push(planeMarker);
            map_.addLayer(plane_array[compteur + offset]);
            if(offset > 0){
                map_.removeLayer(plane_array[compteur + offset - payload]);
            }
            }
        offset = offset + payload;
        console.log("T : "+ offset/payload);
    }
    else{
        console.log("fin d'animation");
    }
}
```

FIGURE 13 – fonction de mise à jour

Ainsi, toutes les secondes, la fonction récupère un lot de coordonnées préalablement stockées dans des tableaux tampons. La quantité de coordonnées mise à jour dépend du nombre total d'avions différents. Pour 200 avions, le code récupère des paquets de 200 coordonnées toutes les secondes. Cette technique permet d'obtenir une animation cohérente avec les coordonnées du fichier texte.

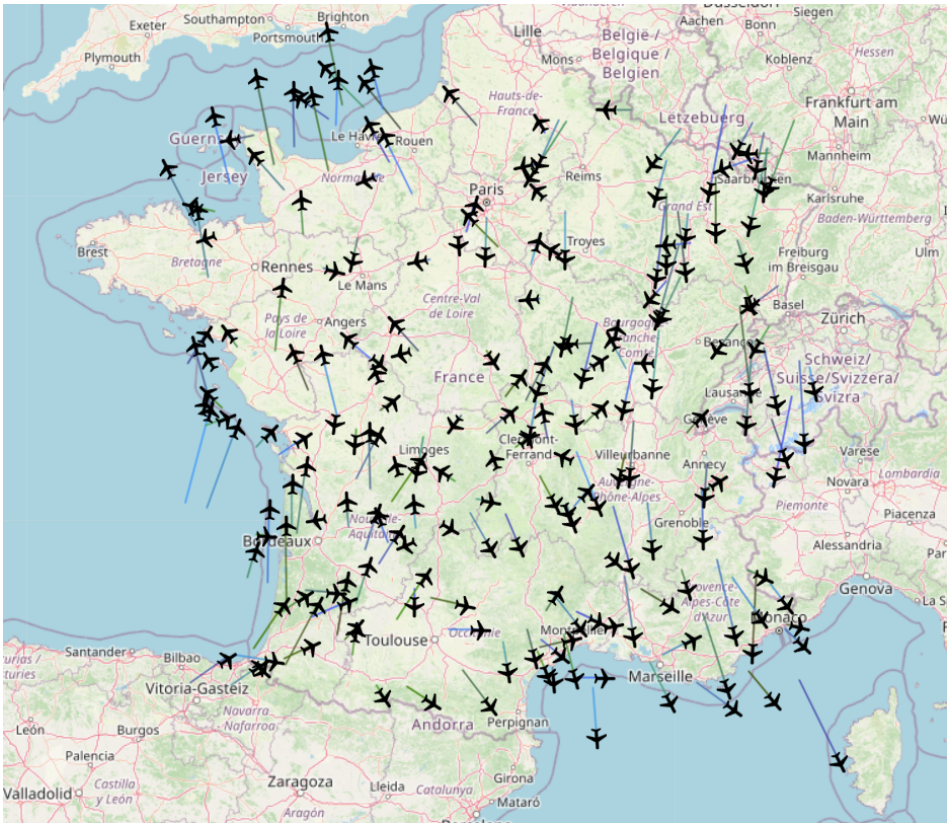

FIGURE 14 – animation finale Javascript

VI Conclusion

Au travers de ce projet, nous avons pu mettre en oeuvre Aff3ct et ses différentes fonctionnalités dans le but de générer un flux de trames d'avions. L'utilisation du Javascript s'est imposée pour réaliser une interface graphique au vu des nombreuses librairies et outils développés pour ce langage.

Bien que le projet soit fonctionnel, le problème du temps-réel n'as pas pu être résolu pour différentes raisons :

- La génération des trames reste lente, ce qui pose un problème majeur pour des simulations temps-réel.
- Il est difficile de traiter les données d'un fichier texte à la volée sans l'avoir fermé (fonction close()). De plus, un fichier ouvert ne prends pas en compte les possibles nouvelles lignes écrites.
- L'interfaçage de Python et Javascript n'est pas évidente au vu des différences entre ces 2 langages.

Pour remédier à ce problème, une solution pourrait-être d'utiliser des sockets web pour envoyer les données sur le port local de la machine. Cela permettrait de plus facilement récupérer les données générées pour les afficher, mais cela impose la création d'un serveur stockant les données. Cela imposerait également de repenser toute la chaîne de transmission de données entre les différents modules.

VII Références

— github AFF3CT : https ://github.com/aff3ct/aff3ct/blob/master/README.md
— Chatgpt : https ://chat.openai.com/