PIE: Pilotage multi drone depuis un MALE

Rapport de livrable logiciel

Anthony Kalaydjian - Vincent Bittard - Pierre Boulogne - Clément Bézard - Florine Lefer

Avril 2023

Contents

Introduction

L'objectif du programme réalisé est de mettre en oeuvre une simulation fonctionnelle modélisant l'implémentation d'une véritable stratégie d'essaim de drone, et son application dans le cadre de la traque d'un véhicule.

Au vu des technologies utilisées dans l'évenementiel, avec des *drone light shows*, il pourrait sembler que les stratégies d'essaim de drone sont déjà en place et qu'elle sont efficaces. En réalité, la plupart de cest technologies utilisées ne fonctionnent qu'en commendant chaque drone indépendamment des autres, et en les confinant dans un volume d'espace déterminé.

L'objectif de ce projet est donc d'établir une stratégie permettant de contrôler un essaim de drone de mannière dynamique, en prenant en compte, lors de la commande, le comportement de chaque drone.

Les applications qui ont été choisies sont donc la poursuite d'un véhicule en mouvement, ainsi que la cartographie d'une zone géographique.

A chaque drone, nous allons associer un rayon de vision, au sein duquel nous admetterons que le drone possède les informations de position et de vitesse du véhicule à tracker. L'algorithme de poursuite prendra donc en compte la vision, ou non, de la cible par l'un des drones, et affectera une commande spécifique à chacun.

1 Programme

1.1 Librairies utilisées

Pour l'affichage de la simulation, c'est la librairie *Pygame* qui sera utilisée. Pygame vient également avec un package *Vector2*, qui permet de réaliser des opérations d'algèbre linéaire simples, tels que la rotation d'un vecteur, le calcul de la norme d'un vecteur...

On utilisera également occasionnellement *numpy* et *random* pour par exemple créér des bruit blancs vectoriels.

1.2 Classes utilisées

1.2.1 Classe *Vehicle*

Pour la classe véhicule, deux types de véhicules sont proposés. Un premier qui peut être comandé par l'utilisateur, à l'aide des flèche directionnelles. Le deuxième type de véhicule est autonome et son déplacement est régit selon une loi choisie. Les différences entre ces deux classes seront abordées plus loin. Nous allons maintenant décrire les fonctionnalités communes aux deux classes:

Attributs

- *position* : position du véhicule
- *velocity* : vitesse du véhicule
- *angle* : angle d'orientation du véhicule
- *length* : hyperparamètre lié au rayon de courbure de rotation du véhicule
- *max_acceleration* : accélération maximale du véhicule
- *min_acceleration* : accélération minimale du véhicule
- *max_steering* : angle maximal du voalnt du véhicule
- *max_velocity* : vitesse maximale
- *min_velocity* : vitesse minimale
- *brake_deceleration* : décélération de freinage
- *free_deceleration* : décélération lié aux frottements
- *max_velocity = max_velocity_0/scaling* : ajustement de la vitesse maximale, suivant le paramètre de scaling du modèle
- *min_velocity = min_velocity_0/scaling* : ajustement de la vitesse minimale, suivant le paramètre de scaling du modèle
- *acceleration* : accélération
- *steering* : angle de rotation de la roue
- *steering_acc* : la dérivée de *steering*

Véhicule controlable

Pour ce qui est du véhicule contrôlable, les commandes se font grâce aux inputs de l'utilisateurs. Seuls ceux associés aux flèches directionnelles et à la barre espace sont prises en compte et permettent de controler le véhicule de manière intuitive.

Accélération :

- Si la flèche du haut est appuyée, alors si le véhicule recule, on décélère. Si le véhicule avançait, alors on incrémente l'accélération de dt.
- La barre espace permet de freiner brutalement.
- Le symétrique de la flèche du haut est effectué pour l'appui sur la flècle du bas.

Rotation :

- Si la flèche droite est appuyée, on décrémente *steering* de *k_steering*dt*, ce qui correspond à tourner le volant vers la droite.
- Si c'est la flèche gauche qui est appuyée, on effectue cette fois une incrémentation de *steering* par le même facteur, ce qui correspond à tourner le volant vers la gauche.
- Si aucune de ces deux flèche n'est appuyée, on réinitialise *steering* à 0.

Véhicule autonome

Le but de cette partie est de créer une véhicule autonome avec une trajectoire assez lisse pour ressembler à une conduite humaine tout en autorisant certains mouvements brusques pour simuler une fuite.
Les actions du véhicule sont aléatoires, afin d'avoir une trajectoire lisse comme dit précédemment c'est l'accélération qui est aléatoire et non la vitesse ou la position. La norme de l'accélération est donc à chaque instant une variable gaussienne, décentrée afin d'avoir en moyenne une accélération positive. De même, la variation de l'angle du volant du conducteur est une gaussienne centrée en 0.
Pour simuler des mouvements brusques qu'aurait une conducteur pourchassé, on effectue par moments des décélérations brutales, les dates de ces décélérations suivent une loi géomètrique et la vitesse en fin de décélération suit une loi uniforme, elle sera comprise entre 0 (le véhicule est à l'arrêt complet) et la moitié de la vitesse initiale de la voiture pour qu'il y ait effectivement un freinage effectif et brusque.
C'est une première approche simple qui peut être optimisée en suivant par exemple plusieurs pistes :

- Plutôt que de prendre des dates aléatoires pour les instants de freinage on pourrait laisser des instants aléatoires mais plus rares et ajouter des instants

déterministes dépendant de la position des drones en chasse. Par exemple il pourrait être judicieux de freiner quand le drone passe au dessus de la voiture et opère une manoeuvre pour se retrouver derrière la voiture, ainsi le drone pourrait perdre la voiture de vue.

- L'approche implémentée ici ne permet qu'un freinage brusque mais pas une réaccélération soudaine ou un changement soudain de trajectoire. Ainsi pour l'instant il est assez facile pour les drones de garder la voiture en visuel car elle reste sur sa trajectoire avec une forte probabilité et n'aura que ralentie, mais si on rajoute un changement de direction (par exemple dans la direction opposée de la direction initiale ou dans la directin opposée par rapport à celle du drone) et une accélération brutale, la voiture peut sortir du champ de vision des drones.
- Notre simulation n'inclue pas de reliefs mais dans une simulation plus poussée qui prendrait par exemple en compte le relief justement il pourrait être intéréssant pour un conducteur chassé de se cacher via ces reliefs (montagnes ou ville par exemple) pour échapper à la vision des drone.

Un modèle plus complet et plus astucieux permettrait ainsi de vérifier la bonne qualité de l'algorithme de suivi des drones voire de l'optimiser si les drones n'arrivent plus à suivre.

1.2.2 Classe *Drone*

Dans le cas du drone, nous nous sommes proposés de garder le même modèle que celui utilisé pour le véhicule. Les attributs du drones sont donc très similaires (à quelques ajouts prêts), et les variables telles que *steering* ne sont que des variables de calcul, et n'ont pas physiquement réels.

Attributs

- *position* : position du drone
- *velocity* : vitesse du drone
- *angle* : angle d'orientation du drone dans le repère fixe associé au jeu.
- *length* : hyperparamètre li e au rayon de courbure de rotation du drone
- *max_acceleration* : accélération maximale
- *max_steering* : angle maximal de la "roue" du drone
- *max_velocity* : vitesse maximale
- *min_velocity* : vitesse minimale
- *steering_acc* : dérivée de *steering*

– *swarm_index* : indice du drone au sein de l'essaim.

– *vision_radius* : rayon de vision du drone.

– *acceleration* : accélération du drone.

– *steering* : angle de la "roue" du drone

Calcul d'angles Il faut faire attention lorsque l'on calcule un angle entre deux vecteurs du plan. Etant une librairie écrite par des informaticien, le repère utilisé par pygame n'est pas orthonormé. En effet, pour être analogue à la position des pixels sur un écran, l'axe des ordonnées est inversé et donne le repère suivant :

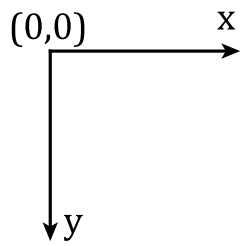

Figure 1: Repère NON-orthonormé de *Pygame*

La classe *Drone* possède également une fonction *update* qui permet de mettre à jour les attributs du drone en fonction du comportement désiré, c'est à dire de poursuivre le véhicule en mouvement. Cette fonction sera développé dans la section *Algorithme de poursuite*.

1.2.3 Classe *Swarm*

La classe Swarm est la classe qui va contrôler l'intégration générale de l'essaim de drones. Il possède entre autre la liste des *N_drones* drones, ainsi que de la position de chaque drone dans une liste, ce qui sera utile lors de la phase de mise à jour de chaque drone.

Elle possède également la fonction *update* qui appelle elle-même la fonction *update* de chaque drone.

1.2.4 Classe *Swarm_game*

1.3 Constantes globales

On met à disposition de l'utilisateur des constantes globales, qui permettent de contrôler finement le modèle utilisé. Parmi les paramètres que l'on peut modifier, il y a le nombre de drones de l'essaim, mais aussi toutes les constantes utilisées dans les contrôleurs. Il y a également les dimensions de la fenêtre d'affichage.

1.4 Algorithme de poursuite

1.4.1 Dynamique utilisée

Pour la rotation des véhicules dans le plan, on se basera sur la géométrie directionnelle d'Ackermann.

Le calcul du rayon de braquage des véhicules est calculé comme suit[1]:

$$R = \frac{D}{sin(\zeta)}$$

Avec:

- R le rayon de braquage qui désigne le plus court rayon de l'arc de cercle qu'un véhicule est apte à décrire lorsqu'il braque [2].
- D la distance entre les essieux du véhicule, qui fait référence à la variable *length*.
- ζ l'angle de la roue, qui fait référence à la variable *steering*.

Mettons nous maintenant à l'itération k et considérons comme aquis les valeurs de la vitesse du véhicule $\vec{v}_k$, ainsi que de l'angle ζ_k. Les calculs de ces deux quantités sera développé dans la partie controleur qui suit.

Connaissant l'angle de la roue, ainsi que la longueur inter-essieu, on peut calculer le rayon de braquage R_k.

A partir de cela, et connaissant la vitesse du véhicule, on peut remonter à la vitesse angulaire qu'aura le véhicule à l'itération k comme suit:

$$\Omega_k = \frac{v_k}{R_k}$$

On calcule enfin la nouvelle position du véhicule, ainsi que sa nouvelle orientation:

$$\begin{aligned} \vec{x}_{k+1} &= \vec{x}_k + \mathcal{R}(\zeta)\,\vec{v}_k dt \\ \alpha_{k+1} &= \alpha_k + \Omega_k dt \end{aligned}$$

Où $\mathcal{R}(\zeta) := \begin{pmatrix} cos(\zeta) & -\sin(\zeta) \\ sin(\zeta) & cos(\zeta) \end{pmatrix}$ est la matrice de rotation d'angle ζ du plan.

[1] https://autoinfome.blogspot.com/p/turning-radius-turning-circle.html
[2] https://fr.wikipedia.org/wiki/Rayon_de_braquage

1.4.2 Contrôle et asservissement dynamique

Controleur

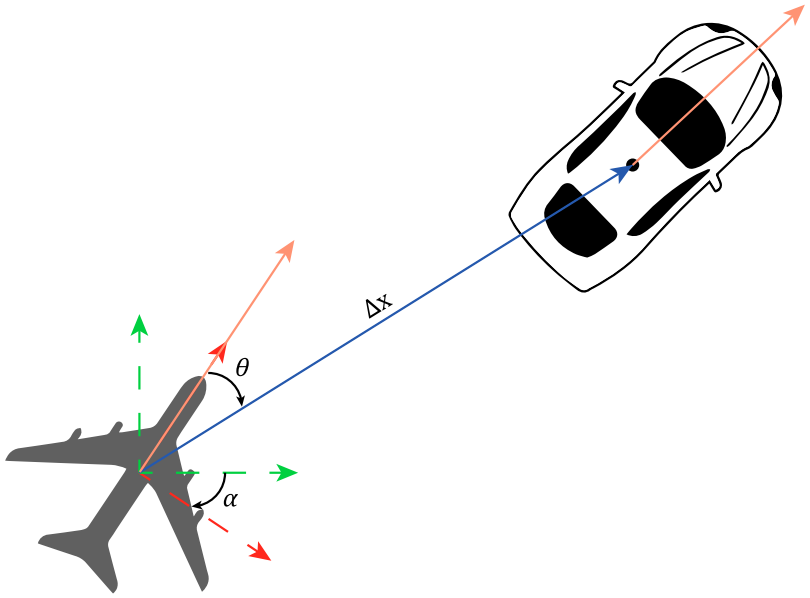

Figure 2: Controleur en poursuite

Le controle va être effectué sur l'accélération du drone, ainsi que sa variable *steering*.

$$\begin{cases} \text{acceleration} = k_{position}\Delta x + k_{velocity}\dot{\Delta x} \\ \zeta = k_\theta \theta + k_\omega \dot{\theta} \end{cases} \tag{1}$$

Mécanisme anti-collision

Empêcher les collision entre les drones est assez complexe. La solution que nous avons retenue est de modifier le contrôle énoncé précédemment en y incluant des termes relatifs aux emplacements de chaque drone.

Pour prendre en compte la collision,

$$\begin{cases} \text{acceleration} = k_{position}\Delta x + k_{velocity}\dot{\Delta x} + k_\beta \sum_{i=1}^{N_{\text{drones}}} \beta_i \\ \zeta = k_\theta \theta + k_\omega \dot{\theta} + k_\gamma \sum_{i=1}^{N_{\text{drones}}} \gamma_i \end{cases} \tag{2}$$

ζ fait toujours référence à *steering*

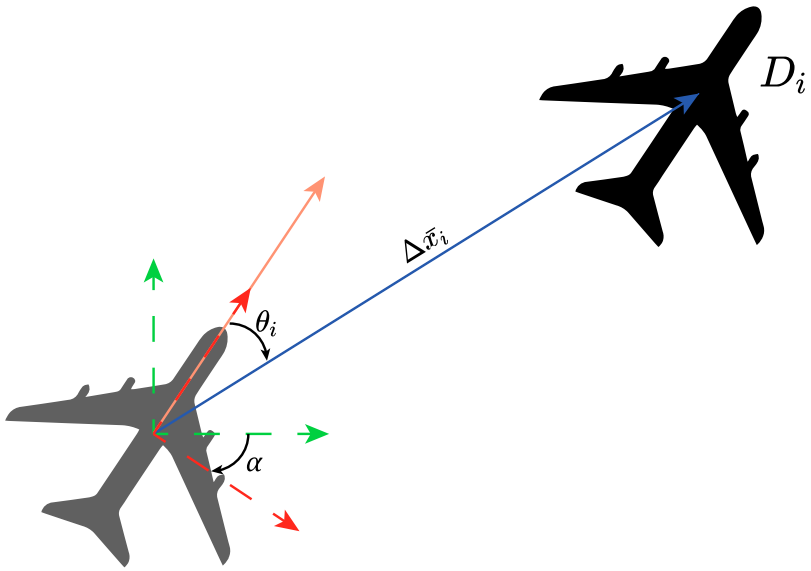

Figure 3: Controleur en poursuite évitant les collisions

où l'on définit, γ_i et β_i comme suit:

$$\beta_i = \begin{cases} -\frac{1}{\Delta x_i^2}|cos(\theta_i)| & \text{si } \theta_i \in [-\pi/2, \pi/2] \\ \frac{1}{\Delta x_i^2}|cos(\theta_i)| & \text{sinon} \end{cases}$$

$\delta \in [0, \pi/2]$ un angle de référence arbitraire.

$$\gamma_i = \begin{cases} -\text{sign}(\theta_i) k_{\beta_1}(1 + \frac{1}{\Delta x_i^2} + k_{\beta_2}\frac{1}{\Delta \dot{x_i}^2}) & \text{si } \theta_i \in [-\delta, \delta] \\ -\text{sign}(\theta_i)\frac{1}{\Delta x_i^2} & \text{sinon} \end{cases}$$

Les motivations derrière ce choix de contrôleur sont les suivantes :

Controleur de l'acceleration
Dans le cas du contrôleur de l'accélération, il faut rajouter un terme qui dépend de l'état de tous les drones environants. L'objectif de ce terme est de ralentir lorsqu'un drone s'approche près de notre drone par en face. A l'inverse, le contrôleur va permettre d'accélérer lorsqu'un drone se rapproche par derrière. Un potentiel en $1/r^2$ a été choisi comme critère de proximité entre les drones. Le cosinus inclut lui la dimmension "devant" et "derrière" dans le contrôleur.

Controleur de *steering*
Le contrôleur de la variable *steering* est un peu plus complexe. Notons d'abord que

quelle que soit la distance du drône i à notre drone courant, si θ_i est relativement petit en valeur absolue, alors la commande γ_i sera d'au moins $-\text{sign}(\theta_i)k_{\beta_1}$. Ainsi, dès qu'un drone sera devant notre drone, ce dernier essaiera de tourner dans le sens opposé à θ_i. On ajoute également un terme en $1/\Delta x^2$ pour prendre en compte la proximité des deux drones et intensifier la rotation lorsque les drones sont proches. Enfin, le dernier terme proportionnel à $1/\dot{\Delta x}^2$ prend en compte le fait que si les drones se rapprochent rapidement l'un de l'autre, alors il faut tourner d'autant plus vite.

Choix des constantes de contrôle
Malheureusement, il n'y a pas de protocole direct pour calculer les coefficients optimaux du contrôle. Les coefficients que nous avons choisis ont été choisis relativement au ressentit que donnait la simulation. Des protocoles plus exhaustifs pourront être développées dans le futur pour pouvoir optimiser les valeurs de ces paramètres.

1.4.3 Contrôle sans visuel actif de la cible

Dès que plus aucun de nos drone n'a de visuel sur la cible, notre essaim change d'état ($Swarm.state$), pour passer en mode recherche, illustrée figure 4 La logique qui a été utilisée pour effectuer la recherche se base sur la réutilisation du code de tracking du véhicule, à ceci près que ce que l'on track ne sera plus le véhicule mais une voiture virtuelle. A chaque drone sera assigné une voiture virtuelle à poursuivre. Le comportement des voitures virtuelles est lié au dernier visuel que l'on a de la voiture. La première voiture virtuelle se dirige dans la direction de la voiture à la vitesse

Notons x_{old} la dernière position connue de la cible, et v_{old} sa dernière vitesse connue.

$\forall i \in [0, N_{\text{drones}} - 1]$ on définit la voiture virtuelle i comme ayant les caractéristiques qui suivent, en fonction du nombre d'itérations k qu'il y a eu depuis la perte de visuel sur la voiture.

$$\begin{cases} \vec{v_i} = \vec{v}_{old} \\ \vec{x_i} = \vec{x}_{old} + k\ dt\ \vec{v}_{old} + (-1)^i k_{spacing}\ r \lfloor \frac{i+1}{2} \rfloor\ \vec{n} \end{cases}$$

Ainsi, la voiture virtuelle va à la vitesse constante $\vec{v}_{old}$. Sa position n'est que l'intégration de sa vitesse entre l'instant de perte de visuel et l'itération k. On rajoute finalement le terme $(-1)^i k_{spacing}\ r \lfloor \frac{i+1}{2} \rfloor \vec{n}$ pour espacer nos voitures virtuelles selon l'axe normal $\vec{n}$ qui est la rotation du vecteur normalisé de $\vec{v}_{old}$ de $\pi/2$. Dans ce terme, $k_{spacing}\ r$ représente un multiple du rayon de vision des drones. Un vol en formation rapprochée aura un $k_{spacing}$ inclut entre 1 et 2. Voir la Figure 5.

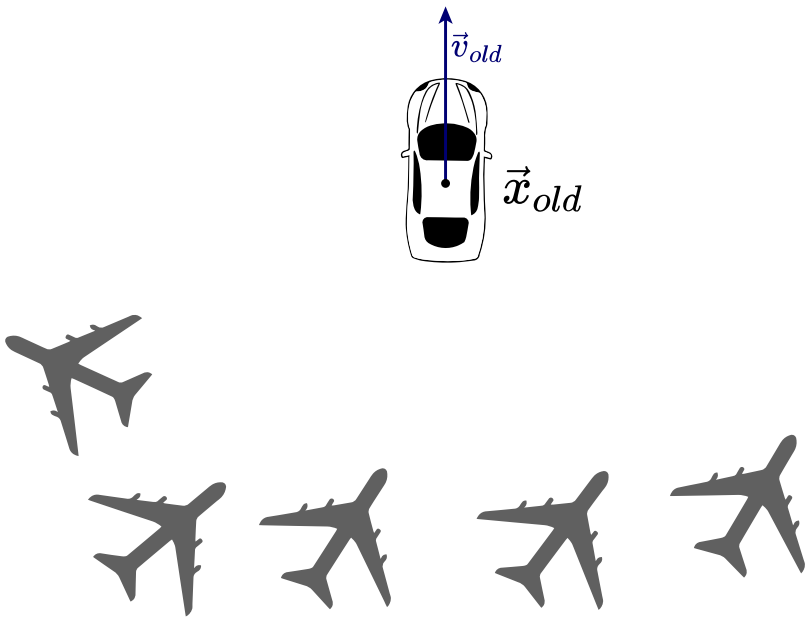

Figure 4: dernière itération précédent la perte de vision sur la cible

Affectation des drones à chauque voiture virtuelle

L'assignation des voitures virtuelles à chaque drone se fait de la façon suivante (voir figure 6) : On calcule d'abord le vecteur $\vec{n}$, qui est le vecteur normalisé de $\vec{v}_{old}$ tourné de $-\pi/2$.

On calcule ensuite pour chaque drone P_i le vecteur $\vec{x_i} - \vec{v}_{old}$ qui le relie à $\vec{x}_{old}$.

Enfin, on calcule la composante $\vec{v_i}$ selon $\vec{n}$ de ces vecteurs, ce qui, projeté dans la direction $\vec{n}$, nous donne un ensemble de valeurs positivies si le projeté P_i' de P_i sur la droite portée par $\vec{n}$ est à droite de $\vec{x}_{old}$, et négatif sinon.

Il ne reste plus qu'à trier ces valeurs de la plus petite à la plus grande, en affectant séquentiellement le drone associé à la voiture virtuelle la plus "à gauche".

Bien que cet algorithme semble correct, son implémentation a posé de nombreux problèmes et bugs dans le programme complet. En raison des contraintes de temps, il a été écarté du programme. Une étude pourra être faite par la suite pour éliminer les erreurs liées à son implémentation.

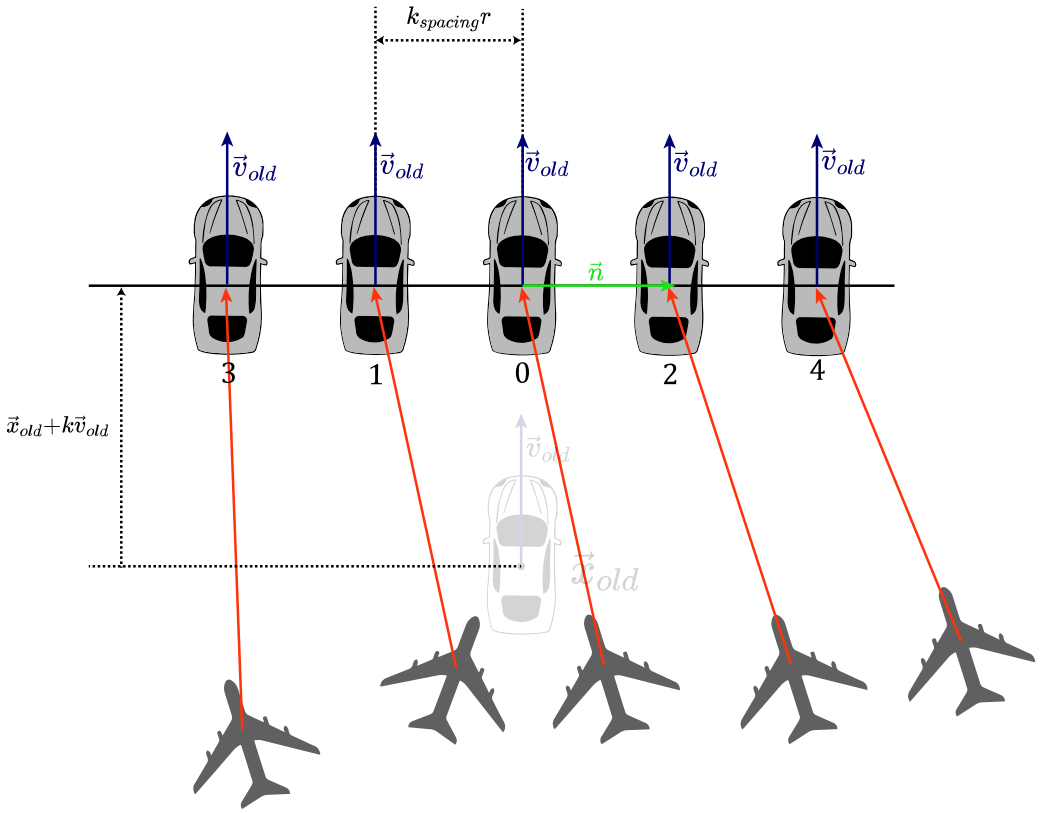

Figure 5: Phase de recherche et voitures virtuelles, itération k

1.5 Algorithme de cartographie

Le but de cette deuxième partie algorithmique est de permettre à des drones de surveiller des zones précises. Il s'agit d'affecter les drones à ces zones, puis de leur faire balayer ces zones à des fins de surveillance.

L'algorithme de cartographie s'appuie sur les classes définies précédement. Etant une tâche séparée de la poursuite, plusieurs classes de l'algorithme de poursuite sont dupliquées pour s'adapter à la cartographie, nous ne les détaillerons donc pas ici afin de ne pas se répéter. L'algorithme de poursuite d'une voiture par un drone a lui été réutilisé dans la cartographie pour le balayage de zones, comme nous l'expliquerons.

1.5.1 Division d'un anneau en sous zones

La difficulté dans cet algorithme réside dans la discrétisation de la forme en anneau. La bibliothèque *Shapely.Geometry* fournit à partir de coordonnées du centre et de

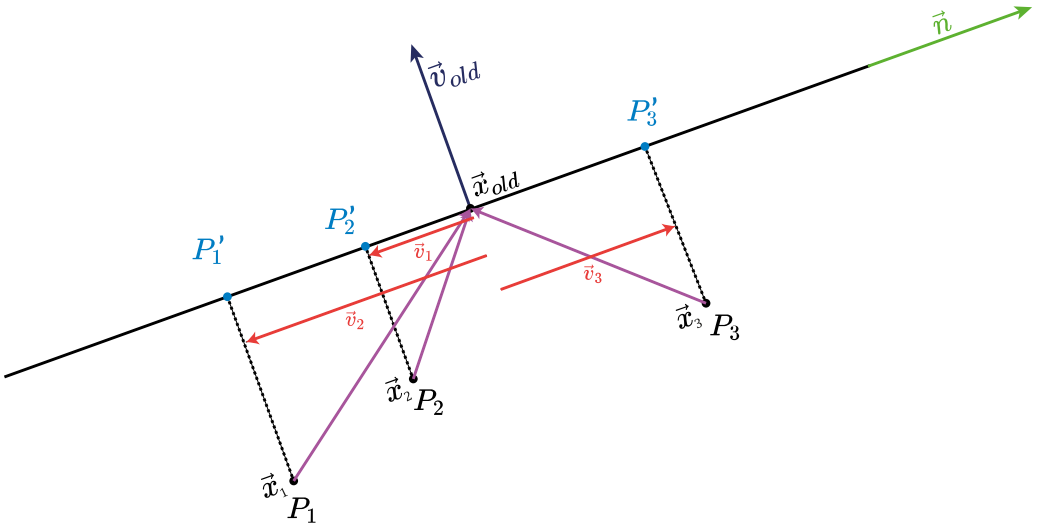

Figure 6: Assignation de la voiture virtuelle à chaque drone

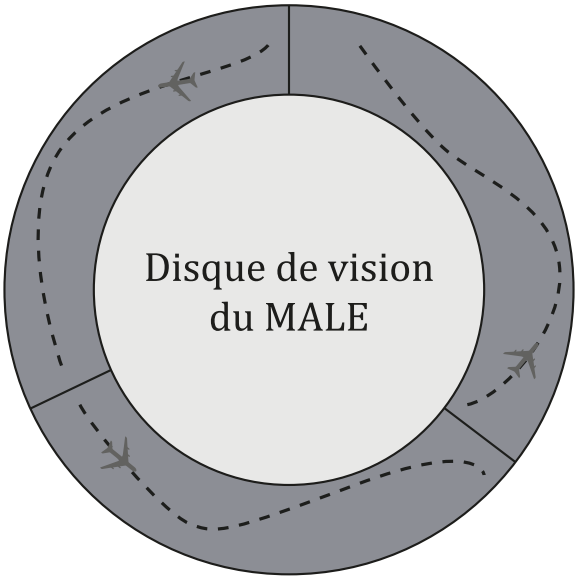

Figure 7: Extension de la vision du drone MALE

son rayon, des points appartenant au cercle à distance régulière les uns des autres. S'en suit la division de l'anneau en sous zones, en divisant le tableau contenant les coordonnées de son cercle extérieur et celui de son cercle intérieur en parties de tailles presque égales.

Les conversions de type ont constitué le principal travail de débuggage dans ce code.

1.5.2 Division d'une zone rectangulaire en sous zones

Il peut être intéressant d'affecter plusieurs drones à une même zone, de forme rectangulaire cette fois. En effet, cela permet de couvrir la zone plus rapidement.
Pour ce faire, il faut découper la zone en sous zones de tailles égales. La forme des sous zones n'importe pas car ce qui compte est seulement la distance que va parcourir chacun des drones, c'est ce qui définira le temps passé à balayer la zone.
Il est donc possible de privilégier la solution la plus simple, c'est-à-dire la divison en "tranches" le long du côté le plus grand. L'algorithme mis en place effectue cette division et retourne les coordonnées des 4 coins de chaque sous zone rectangulaire.

1.5.3 Affectation de drones à des zones selon les plus courts chemins

Il s'agit là d'affecter les drones à des zones à surveiller. Pour des questions de rapidité opérationnelles, il faut affecter à chaque zone à surveiller le drone le plus proche. Pour être optimal, il faudrait même prendre en compte la distance drone - zone par rapport à la taille de cette zone. En effet, la situation optimale est celle dans laquelle tous les drones ont la même charge de travail. Si tous les drones finissent la surveillance en avance et qu'ils attendent un autre drone, le rapport de surveillance complet est retardé. Il aurait été préférable de répartir cette charge de travail supplémentaire sur les autres drones. Seulement, ce problème algorithmique n'admet pas de solution optimale, il n'est à ce jour pas résolu.

La solution retenue est donc, pour chaque zone, de lui affecter le drone le plus proche. Pour cela, c'est encore une fois la bibliothèque *Shapely.Geometry* qui est utilisée. Celle-ci fournit en effet une fonction qui calcule la distance d'un point (la position du drone) à une zone rectangulaire définie par les coordonnées de ses sommets. Les affectations *drone i- zone i* sont ensuite enregistrées comme attributs des classes utilisées.

1.5.4 Balayage de zone

Le balayage de zone est une partie délicate. En effet, il faut trouver un "chemin" sur lequel faire passer le drone pour qu'il balaye toute une zone donnée. La première étape est donc de fabriquer une *mesh* de la zone. En effet, il s'agit de discrétiser la zone pour indiquer le chemin à parcourir au drone.

Une mesh est obtenue grâce à la bibliothèque numpy.array, à laquelle on indique le pas de la mesh. Celui-ci définit la distance entre 2 points successifs de la mesh. La fonction utilisée retourne une liste contenant ces points. Cependant, cette liste

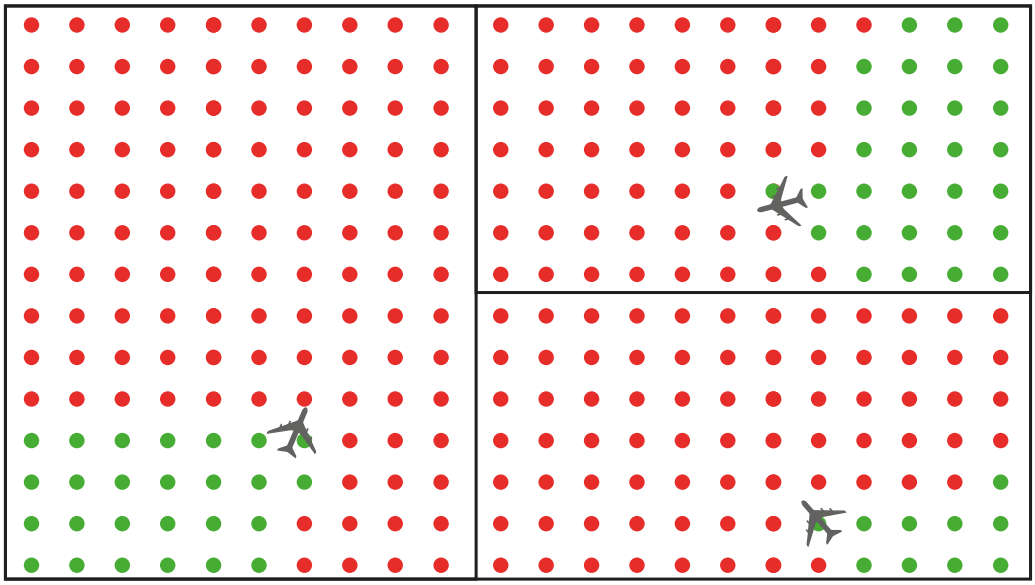

Figure 8: Cartographie d'une zone rectangulaire

ordonne les points de façon arbitraire. L'ordre obtenu n'est en fait pas idéal pour un parcours de zone, faisant inutilement du chemin en trop.
Pour une zone rectangulaire, il s'agit juste de réordonner la liste pour faire un chemin, un balayage en lacets. Ceci n'a pas été trop compliqué à implémenter.
Pour une zone qui est une partie d'un anneau, c'est beaucoup plus complexe. Il aurait fallu implémenter une grosse batterie de tests pour espérer pouvoir bien ordonner les points dans la liste, c'est-à-dire en ne faisant pas repasser le drone 2 fois par le même endroit. Faute de temps, nous n'avons pas pu tenter d'implémenter un tel algorithme.

Une fois la mesh construite, il faut effectuer le balayage par le drone. Pour cela, nous avons réutilisé l'algorithme de poursuite de cible en l'adaptant. Ici, la cible est immobile, ce sont les points du chemin par lequel doit passer le drone pour atteindre pour balayer la zone attribuée. Tous ces points se situent dans une même liste. Lorsque le drone atteint le point défini comme cible, ce point est retiré de la liste et le point suivant dans l'ordre est défini comme cible. Le drone balaye ainsi la zone sans repasser par la même case.

Notons cependant que cela nécéssite de définir correctement certains paramètres. L'espacement des points de la mesh doit idéalement être égal au rayon de vision du drone. En pratique, il est légèrement plus grand car cela évite les problèmes dûs à des mini défauts de déviation de trajectoire du drone. Le rayon de courbure de

la trajectoire du drône doit être réglé en fonction du rayon du champ de vision de celui-ci. Si le rayon est faible, le pas de la mesh est faible (puisque qu'égal) et le parcours en lacet de la zone rectangulaire nécessite, aux extrémités de la zone, un virage important.

Conclusion

A partir d'un modèle simple de véhicule se déplaçant dans le plan 2D, nous avons pu concevoir et implémenter deux types d'algorithmes particulièrement utiles lors de la poursuite d'un autre véhicule et de l'exploration d'une zone inconnue.

D'abord un algorithme de poursuite se basant sur un controleur agissant sur l'accélération des drones ainsi que de l'angle de rotation de la "roue" modélisant leur capacité à tourner au plus selon un angle et un cercle de braquage.

Ensuite, un algorithme de cartographie qui permet d'abord d'assigner des drones à plusieurs zones à surveiller de façon efficace en terme de temps, puis de balayer ces zones afin de collecter des informations.