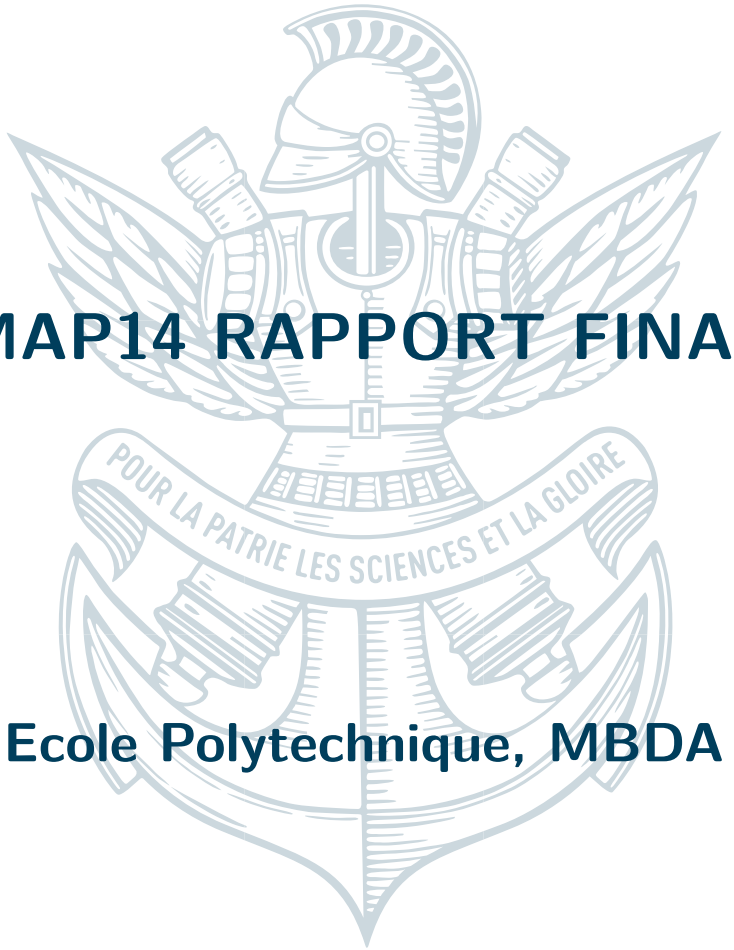


MAP14 RAPPORT FINAL

Ecole Polytechnique, MBDA

20 avril 2025

Mohssine Bakraoui, Andréa Bourelly, Aurélien Castillon,
Joseph Combourieu, Imad El-Hassouni, Mohamed-Réda Salhi

mbda.png

TABLE DES MATIÈRES

1
OBJECTIFS ET PRÉSENTATION DU MODÈLE

Les essaims de drones sont devenus un outil technologique incontournable dans de nombreux domaines. En effet, les simulations montrent que les essaims de drones présentent un grand intérêt dans de nombreux cadres tels que l'extinction de feu, le quadrillage de zones ou certaines opérations militaires. L'objectif de notre PSC est le développement d'un contrôleur permettant la navigation d'un essaim de drones dans un environnement à obstacles. Les obstacles ne constituent pas la seule contrainte qui se pose à notre essaim. On souhaite en effet que celui-ci respecte tant que possible une formation donnée et bien évidemment qu'il n'y ai pas de collisions entre les drones. A ces contraintes s'ajoute également une contrainte dynamique. Cette étude se décline en deux volets : un volet mathématique, comprenant la modélisation théorique du mouvement d'un essaim de drônes, ainsi qu'un volet informatique, qui se caractérise par l'implémentation numérique de notre modèle théorique.

La question de navigation d'un essaim de drones dans un milieu à obstacles, bien que récente, n'est pas inédite. Bien des approches ont auparavant été considérées. La plus connue d'entre elle est l'approche dite de *leader-follower*. Le point clé de cette approche est de désigner un drone comme le *leader*, de précalculer sa trajectoire dans l'environnement à obstacles, et de faire suivre les autre drones ce *leader*. Cette méthode est décrite avec précision dans l'article écrit par Jialong Zhang [2]. L'essaim de drônes, ici une formation triangulaire, parvient avec succès à éviter un seul obstacle statique cylindrique situé au mileiu de la carte tout en maintenant la formation triangulaire. Cependant, le succès obtenu sur ce cas précis est difficilement réplicable à d'autres cas plus complexes. En effet, cette stratégie de *leader-follower* impose une trop grande rigidité de formation qui peut conduire à des collisions si l'environnement est de nature plus complexe. En effet, les drones suivant tous la trajectoire du *leader*, ce dernier peut éviter un obstacle sans que ce soit le cas pour les autres, notamment dans un environnement avec de nombreux obstacles. Une autre approche pour le controle d'essaim de drones est celle de noyau virtuel, telle que detaillée par Enming Wu [3]. Cette méthode repose sur un algorithme de k-means pour regrouper les drones en cluster centrés autour de noyaux virtuels. Les différents drones d'un même cluster sont alors soumis à une force d'attraction vers le noyau, ce qui permet d'encoder une formation. Cette démarche offre une meilleure flexibilité de la formation, permettant ainsi de mieux éviter les obstacles dans un environnement complexe mais comporte néanmoins un grand problème calculatoire. En effet, cette méthode repose sur une communication d'un très grand nombre de paires de drones différentes, nombre d'autant plus grand que l'essaim est vaste. Ainsi, d'un

point de vue calculatoire, ce modèle atteint ses limites pour des grands essaims de drones.

Pour résumer, les approches jusque-là considérées semblent opposer flexibilité de la formation et large taille d'essaim. En effet, dans bien des approches, une grande flexibilité de la formation permet certes de naviguer dans un environnement complexe mais au prix d'un coût informatique très élevé. A l'inverse, certaines méthodes efficaces calculatoirement sont forcées de faire un compromis sur la flexibilité de la formation et sont ainsi moins performantes dans un environnement compliqué. Pour concilier ces deux approches, nous avons utilisé la théorie des jeux à champs moyens, dont l'avantage principal est d'éviter une communications entre drones de type $N - N$, c'est-à-dire entre toutes les paires de drones, ce qui est très coûteux d'un point de vue calculatoire mais de faire interagir les drones avec la moyenne des drones pour passer à une communication entre drones de type $N-1$. Notre démarche s'inspire de l'article de Guofang Wang [1], qui présente une modélisation théorique de la navigation d'un essaim de drones en utilisant la théorie des jeux à champs moyens ainsi qu'une ébauche d'implémentation informatique. Cette démarche nous a intéressante à considérer dans la mesure où l'article met en évidence de bonnes performances tant sur le maintien de la formation que sur la capacité du contrôleur à éviter des obstacles de nature variée. Si notre démarche personnelle s'inspire de cet article, nous nous en sommes largement émancipés en proposant des fonction de coût tant pour la formation, que pour les obstacles que pour les collisions inter drones, radicalement différentes. Par ailleurs, nous avons intégralement implémenté informatiquement le modèle théorique proposé, ce qui nous a d'une part permis de vérifier l'efficacité de notre modèle et d'autre part nous a conduit à revisiter et à préciser certains détails omis dans le papier.

Ainsi, notre étude se décline en deux volets : un théorique et un informatique. Pour poser le cadre de notre étude théorique, nous introduisons la densité de probabilité ρ représentant la répartition spatiale des drones, telle que l'intégrale $\int_A \rho(x,t)dx$ corresponde à la probabilité de trouver un drone dans la région A à l'instant t. L'objectif est de déterminer précisément cette densité $\rho(x,t)$ afin de modéliser efficacement le comportement collectif des drones.

Afin d'évaluer et optimiser la réalisation des objectifs de la mission, nous définissons des fonctions de coût sur la densité $\rho(x,t)$. Ces fonctions de coût reflètent plusieurs critères essentiels : le maintien d'une formation prédéfinie, l'atteinte d'une cible ou d'un objectif final, ainsi que l'évitement efficace des obstacles fixes et des collisions inter-drones.

La densité ρ pénalise au cours du temps les zones de l'espace contenant des obstacles ou éloignées de la position finale en leur attribuant des probabilités faibles. En fixant un instant donné, on espère retrouver une densité de probabilité qui suit la même loi que celle de la formation désirée à translation près, assurant ainsi la cohérence spatiale du groupe de drones tout au long de l'évolution temporelle.

En ce qui concerne le volet informatique, nous avons recours à des réseaux de neurones pour implémenter informatiquement les résultats théoriques. Plus spécifiquement, en suivant la démarche proposée dans l'article *A Mean-Field Game Control for Large-Scale Swarm Formation Flight in Dense Environments*, nous avons recours à des Réseaux antagonistes génératifs (RAG) pour simuler ρ. L'idée des RAG est de mettre en opposition deux réseaux de neurones. Un réseau générateur est chargé de produire des estimations de la densité de population ρ, tandis qu'un second réseau, appelé le discriminateur, a pour rôle d'évaluer si ces estimations correspondent à de vraies densités. Ces deux réseaux fonctionnent en compétition : le générateur tente de tromper le discriminateur, qui s'améliore en retour pour mieux détecter les fausses données. Cette dynamique antagoniste les pousse à converger vers une approximation optimale de $\rho(x,t)$. En pratique, le système n'aboutit pas à une expression explicite de la densité, mais à un réseau capable de générer des trajectoires de positions successives, intégrant de façon implicite la densité optimale.

Cette approche s'inscrit dans une modélisation à deux joueurs : le Joueur 1 représente le champ moyen — autrement dit, l'essaim global des drones — dont la stratégie est précisément cette densité ρ, issue d'un modèle liant les positions initiales à leur évolution temporelle. Cette densité reflète l'organisation collective de l'essaim dans l'espace et dans le temps, dictée par les objectifs de la mission. Le Joueur 2, quant à lui, incarne un agent générique dont la stratégie repose sur une fonction de valeur ϕ, visant à fournir la meilleure réponse individuelle face à la dynamique globale du système.

2
OPTIMISATION SOUS CONTRÔLE

Dans toute l'étude théorique, on se place dans $\Omega = \mathbb{R}^3$.

2.1 Fonctions de coût, cas discret

Commençons par donner une fonction de coût dans un cadre discret dans le but de motiver les fonctions de coût que l'on donnera par la suite lors du passage au continu. On envisage ainsi dans un premier temps le cas discret de N drônes, dont on note $(x_i)_i$ les positions dans $\mathbb{R}^3$, dans un environnement contenant K obstacles fixes. Dans un soucis de simplicité, les obstacles que l'on considère sont tous sphériques. Pour créer des formes plus compliquées d'obstacles, on pourra ainsi empiler différentes sphères pour approximer d'autres constructions. La donnée des K obstacles

est alors :

$$(o_1, r_1) \dots (o_K, r_K) \tag{1}$$

où les o_i sont les centres des obstacles et r_i les rayons. Dans cette optique, la distance que l'on considère et alors toujours la distance euclidienne, que l'on notera $\|.\|$ par simplicité.

On décompose notre fonction de coûts en trois fonctions de coût. La première est une fonction de côut relative aux obstacles de l'environnement. On attend de cette fonction qu'elle explose si l'un des drones s'approche d'un obstacle. Cependant, pour ne pas biaiser la valeur de la fonction de côut, qui lorsque l'on est suffisamment loin d'un obstacle doit se focaliser sur le coût de la formation, la fonction de coût relative aux obstacles doit être nulle en dehors des régions de proximité des obstacles. On note $\mathcal{F}_1$ cette fonction que l'on définit de la sorte :

$$\mathcal{F}_1((x_i)_i, t) = \frac{1}{N} \sum_{i=1}^{N} G_1(x_i, t) \tag{2}$$

où l'on a posé :

$$G_1(x_i, t) = \sum_{k=1}^{K} \mathbf{1}_{\|x_i(t) - o_k\| < 2r_k} \gamma_k \frac{1}{\max(\|x_i(t) - o_k\| - r_k, 0) + 10^{-8}} \tag{3}$$

où γ_k est un coefficient d'importance de l'obstacle. Dans les tests qui ont été menés, γ_k a toujours été fixé à 1. La distance se retrouvant au dénominateur Cette fonction explose bien lorsque l'un des drones se rapproche d'un obstacle. L'ajout du max permet de garantir que le terme spécifique à l'obstacle k de la fonction de coût est uniformément égale à $\propto 10^8$ si un drone se retrouve à l'intérieur de la sphère de centre o_k et de rayon r_k. Le terme 10^{-8} permet de s'assurer que l'on ne divise jamais par 0, particulièrement important dans le cadre informatique où les erreurs d'arrondi entraîneraient souvent une division par 0. Enfin, l'ajout de l'indicatrice nous assure que la fonction de coût relative à un obstacle (o_k, r_k) est nulle dès lors que tous les drones sont à une distance supérieure à $2r_k$ du centre.

La deuxième fonction de coût est une qui modélise la non collision entre les drones de la formation. Cette fonction doit respecter des contraintes de nature similaire à celle qui modélise la non collision des drones avec les obstacles. En effet, elle doit exploser lorsque deux drones se rapprochent trop l'un de l'autre, et doit être nulle si les drones sont deux à deux distants d'une distance supérieure à une distance de sécurité $\varepsilon > 0$ (de l'ordre de la longueur de la taille caractéristique des drones), pour ne pas biaiser la fonction de coût totale. On appelle F_2 cette fonction que l'on définit par

$$\mathcal{F}_2((x_i)_i, t) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{F}_{2,i}((x_j)_j, t) \tag{4}$$

avec

$$\mathcal{F}_{2,i}((x_j)_j, t) = \frac{1}{N} \sum_{j \neq i} \mathbf{1}_{\|x_i - x_j\| < \varepsilon} \times \frac{1}{\|x_i - x_j\|^2 + 10^{-8}} \tag{5}$$

Cette fonction explose bien lorsque deux drônes sont trop proches et est bien nulle lorsque tous les drônes sont à une distance d'au moins ε. Encore une fois, le terme 10^{-8} permet de s'assurer que l'on ne divise pas par 0.

Se pose alors la question de la fonction de coût qui modélise le respect d'une formation imposée. On présente ici ce qui est donné dans l'article [1], sachant que cette idée sera ensuite volontairement abandonnée par la suite. Le respect d'une formation donnée passe ici par les matrices laplaciennes. On définit ainsi $\mathcal{F}_3$ notre fonction de coût relative au respect d'une formation attendue :

$$\mathcal{F}_3((x_i)_i, t) = \left\| \hat{L}((x_i)_i, t) - \hat{L}_e(t) \right\|_F^2, \tag{6}$$

où $\hat{L}$ désigne la matrice de Laplace normalisée de la formation et $\hat{L}_e$ désigne la matrice de Laplace de la formation idéale souhaitée. On détaille ce qu'est la matrice de Laplace :

Considérons un graphe $\mathcal{G}$, où le sommet i représente le $i^{\text{ème}}$ agent avec un vecteur de position $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$. Une arête $e_{ij} \in \mathcal{E}$ qui relie le sommet i au sommet j signifie que ces agents peuvent mesurer leur distance géométrique. Dans ce travail, chaque agent communique avec tous les autres, ce qui fait que le graphe de formation $\mathcal{G}$ est complet. Chaque arête est pondérée par une valeur positive correspondant à la distance au carré entre les agents i et j :

$$w_{ij} = \|p_i - p_j\|^2, \quad (i,j) \in \mathcal{E}, \tag{7}$$

où $\|\cdot\|$ désigne la norme euclidienne.

Nous définissons ensuite la matrice d'adjacence $\mathbf{A} = (A_{ij}) \in \mathbb{R}^{N \times N}$ et la matrice de degré $\mathbf{D} = (D_{ij}) \in \mathbb{R}^{N \times N}$ de la formation $\mathcal{G}$ comme suit :

$$A_{ij} = w_{ij}, \tag{8}$$

$$D_{ij} = \begin{cases} \sum_j A_{ij} & \text{si } i = j, \\ 0 & \text{sinon.} \end{cases} \tag{9}$$

La matrice Laplacienne correspondante est alors définie par :

$$\mathbf{L} = \mathbf{D} - \mathbf{A}. \tag{10}$$

Enfin, la matrice Laplacienne normalisée symétrique est donnée par :

$$\hat{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}, \tag{11}$$

où $\mathbf{I} \in \mathbb{R}^{N \times N}$ est la matrice identité.

La matrice de Laplace normalisée symétrique contient l'information géométrique de la formation de drones.

On a note

$$\mathcal{F} = \alpha_1\mathcal{F}_1 + \alpha_2\mathcal{F}_2 + \alpha_3\mathcal{F}_3$$

Les coefficiens $\alpha_1, \alpha_2, \alpha_3$ sont des coefficients de pondération des coûts.

On ajoute également un cout indépendant du temps correspondant au degré de réalisation de l'objectif final. On souhaite faire converger en moyenne les drones vers une position finale x_f. En considérant que la formation sera à peu près maintenue à l'arrivée, on prépare le terrain pour le cas continu et on ramène la convergence point par point à une convergence en moyenne avec maintien de la formation. On pose alors :

$$\mathcal{G}((x_i)_i) = \alpha_4 \left\| \frac{1}{N}\sum_{i=1}^{N} x_i \ - x_f \right\| \tag{12}$$

Le coefficient α_4 est un coefficient de pondération du coût terminal. En pratique, trouver des valeurs adéquates pour les coefficients de pondération α_j, $j \in [\![1, 4]\!]$ est une difficulté majeure et revêt une importance cruciale dans la réalisation simultanée de tous les objectifs. Nous reviendrons sur le choix de ces coefficients dans la partie expérimentale.

Dans un cadre pratique idéale, les drones sont contrôlés par une impulsion u_i : $\mathbb{R} \to \mathbb{R}^4$ avec u= $[u, \tau_\psi, \tau_\theta, \tau_\varphi]$. Ceci signifie que les drones tournent sur eux-mêmes et sont propulsés par une force de norme u dans la direction induite par l'état du drone. C'est l'approche adoptée dans l'article [1]. Par soucis de simplification, on définit un contrôleur $u_i : \mathbb{R} \to \mathbb{R}^3$ par $u_i(t) = [v_{i,x}(t), v_{i,y}(t), v_{i,z}(t)]$. De plus, on néglige la dynamique liée aux interactions extérieures (quitte à adapter le contrôle u pour compenser l'effet extérieur). Nos équations dynamiques sont alors simplifiées et l'on obtient :

$$\begin{cases} \dot{x} = v_{i,x} \\ \dot{y} = v_{i,y} \\ \dot{z} = v_{i,z} \end{cases} \tag{13}$$

On peut maintenant définir la fonction de coût total

$$\mathcal{J}((x_i)_i, (u_i)_i) = \int_0^T L((u_i)_i, t) + \mathcal{F}((x_i)_i, t)dt \ + \ \mathcal{G}((x_i)_i) \tag{14}$$

T est l'instant d'arrivée de l'essaim de drones, de sorte que l'on intègre sur tout le chemin parcouru. On a noté :

$$L((u_i)_i, t) = \frac{1}{2N}\sum_{i=1}^{N} \|u_i(t)\|^2 \tag{15}$$

qui représente le coût relatif au carburant utilisé pour la propulsion des drones.
On ajoute de plus un terme de bruit blanc à notre dynamique, cela nous permet e modéliser une situation dans laquelle les drones sont soumis à des mesures bruitées. On est maintenant prêts à définir le problème sous la forme d'un problème d'optimisation sous contrainte :

$$\inf_{(u_i)_i} \mathcal{J}_i((x_i)_i, (u_i)_i) \quad \text{sous la contrainte} \quad d\mathbf{x}_i = \mathbf{u}_i dt + \sigma d\omega_i(t). \tag{16}$$

où $d\omega_i$ mouvement est un mouvement Brownien

2.2 Utilisation de la théorie des jeux à champs moyens

Lorsque la taille N de l'echantillon augmente, la complexité des calculs des fonctions coût augmente grandement, au point où ces calculs seront beaucoup trop lourds à executer drone par drone. Dans cette optique, le passage au cas continu et le recours à la théorie des jeux à champs moyens semble être une approche pertinente. On fait donc l'hypothèse que les drones suivent une densité $\rho(x,t)$. On note $\rho_0(x) = \rho(x,0)$ qui décrit simultanément la formation initiale avec laquelle part les drones ainsi que la formation idéale que les drones doivent au mieux respecter. Plus précisément, à un instant t, on souhaiterait avoir, si cela n'engendre pas de collision avec un obstacle (notamment dans un environnement complexe) :

$$\rho(x,t) - \mathbb{E}_x(\rho(x,t)) \approx \rho_0(x)$$

En raisonnant avec cette densité $\rho(x,t)$, on implémente maintenant les fonctions de coût dans le cas continu en se basant sur l'intuition du cas discret. Ainsi, la fonction de coût relative aux obstacles devient :

$$\mathcal{F}_1(\rho,t) = \sum_{k=1}^{K} \int_\Omega \mathbf{1}_{\|x-o_k\|<2r_k} \frac{1}{\max(\|x-o_k\| - r_k, 0) + 10^{-8}} \rho(x,t)dx \tag{17}$$

De la même manière, la fonction de coût relative aux collisions inter obstacles devient :

$$\mathcal{F}_2(\rho,t) = \int_\Omega \int_\Omega \mathbf{1}_{\|x-y\|<\varepsilon} \times \frac{\rho(x)\rho(y)}{\|x-y\|^2 + 10^{-8}} dxdy \tag{18}$$

Notre coût terminal associé à la réalisation de l'objectif devient :

$$\mathcal{G}(\rho) = \alpha_4 \int_\Omega |\rho(x,t) - (\rho_0(x - x_f))| dx \tag{19}$$

De même, en passant au continu, on a plus seulement un nombre discret de fonctions de contrôle $u_i(t)$ mais une fonction à deux variables $\mathbf{u}(x,t)$

$$L(\mathbf{u},t) = \frac{1}{2} \int_\Omega \|u(x,t)\|^2 \rho(x,t) dx \tag{20}$$

Jusqu'à présent, l'adaptation des fonctions de coût précédentes au cas continu ne posait pas de problème. Cependant, l'adaptation au cas continu de la fonction de coût relative à la formation pose problème. En effet, nous avons volontairement décider de considérer une autre piste que celle qui a recours aux matrices laplaciennes et ce pour deux raisons :

— Tout d'abord, nous n'avons touvé dans la littérature aucun prolongement continu de la matrice laplacienne et les articles qui y ont recours sont flous quant à son expression. L'article [1] par exemple n'explicite jamais ce passage au continu pour la fonction de coût de formation.

— Surtout, une approche qui étendrait l'idée de la matrice laplacienne pour mesurer l'écart à la formation souhaitée dans le cas continu serait couteuse informatiquement. En effet, il faudrait faire une intégrale double sur tout l'espace pour mesurer les écarts entre les drônes. Or, comme il sera détaillé dans l'implémentation informatique, ces intégrations sont très couteuses car il faut mesurer avec précision l'écart entre les formations.

Alors, nous avons décidé d'opter pour une approche nettement plus naturelle vis à vis du problème qui nous est donné et de modéliser l'écart à la formation idéale par :

$$\mathcal{F}_3(\rho, t) = \|\rho(x,t) - \mathbb{E}_x(\rho(x,t)) - \rho_0(x)||_p \tag{21}$$

Ici, $\|.\|_p$ désigne la norme de l'espace L^p. Nos fonctions de coût étant toutes posées, on peut définir la fonction de coût global :

$$\mathcal{J}(\rho, \mathbf{u}) = \int_0^T L(\mathbf{u}) + \mathcal{F}(\rho, t)\, dt \; + \; \mathcal{G}(\rho) \tag{22}$$

sous la contrainte :

$$dx = h(x(t), \rho(x,t), \mathbf{u}(x,t))\, dt + \sigma\, d\omega(t). \tag{23}$$

La contrainte telle qu'exprimée ci-dessus a l'inconvénient de contenir une composante de bruit représentée par le processus de Wiener $dw(t)$. La théorie générale des processus stochastique nous permet d'éliminer cette composante en montrant que l'équation ci-dessus se traduit en équation sur la densité de probabilité $\rho(x,t)$. Celle-ci est appelée équation de Fokker-Planck et s'écrit comme suit :

$$\partial_t \rho - \frac{\sigma^2}{2}\Delta\rho \; + \; \nabla.(\rho h) = 0 \tag{24}$$

Dans le cadre simplifié mais suffisant dans lequel nous travaillons nous avons $h = \mathbf{u}$. Mais nous allons fournir une démonstration générale de l'équivalence entre les deux équations.

2.3 Démonstration de l'équation de Fokker-Planck

Ce théorème repose sur certaines hypothèses qui sont les suivantes. Comme c'est le cas en pratique nous supposons que les fonctions $\partial_t\rho(.,t)$ et $\rho(.,t)$ sont respectivement dans $L^2(\mathbb{R}^3)$ et $H^2(\mathbb{R}^3)$.

Commençons par introduire certaines notations. Soient $s, t \in \mathbb{R}^+$ avec $s < t$ et Γ_s, Γ_t deux ensembles mesurables on pose $P(s, \Gamma_s; t, \Gamma_t) = \mathbb{P}(\mathbf{x}(s) \in \Gamma_s | \mathbf{x}(t) \in \Gamma_t)$ et si $x \in \mathbb{R}^3$ l'ensemble dX au produit cartésien $\Pi_1^d[x_i, x_i + dx_i[$. De plus on note $\Gamma_0 \subseteq \mathbb{R}^3$ un ensemble dans lequel sont inclus tous les drones initialement. Avec ces notations :

$$P(t, dX_x; 0, \Gamma_0) = \mathbb{P}(\mathbf{x}(t) \in dX_x | \mathbf{x}(0) \in \Gamma_0) = \mathbb{P}(\mathbf{x}(t) \in dX_x) = \rho(x, t)dx$$

Démonstrons d'abord le sens direct. Supposons que l'on ait l'équation différentielle stochastique suivante :

$$d\mathbf{x} = h(\mathbf{x}(t), \rho(\mathbf{x}, t), \mathbf{u}(\mathbf{x}, t))dt \ + \ \sigma dw(t)$$

Soit $\phi \in C_c^\infty(\mathbb{R}^3, \mathbb{R})$ à support compact, ϕ vérifie les conditions de régularité nécessaires pour appliquer le lemme d'Ito, que l'on peut écrire sous cette forme :

$$\begin{aligned} d(\phi(\mathbf{x}(t)) &= \nabla\phi(\mathbf{x}(t)) \cdot d\mathbf{x}(t) \ + \ \frac{\sigma^2}{2}\Delta\phi(\mathbf{x}(t))dt \\ &= (\nabla\phi(\mathbf{x}) \cdot h(\mathbf{x}, \rho(\mathbf{x}, t), \mathbf{u}(\mathbf{x}, t)) \ + \ \frac{\sigma^2}{2}\Delta\phi(\mathbf{x}))dt \ + \ \sigma\nabla\phi(\mathbf{x}) \cdot dw(t) \end{aligned}$$

Nous notons $\mathcal{L}_t = h(., \rho(., t), \mathbf{u}(., t)) \cdot \nabla \ + \ \frac{\sigma^2}{2}\Delta$ l'opérateur linéaire intérvenant dans l'expression de la différentielle ci-dessus.

Au vu de la formule précédente pour $t \in \mathbb{R}^+$ au voisinage de 0 nous pouvons écrire le développement limité suivant pour le processus stochastique $\phi(\mathbf{x})$:

$$\phi(\mathbf{x}(t + s)) - \phi(\mathbf{x}(t)) = s(\mathcal{L}_t\phi)(\mathbf{x}(t)) \ + \ \sigma\nabla\phi(\mathbf{x}(t)) \cdot (dw(t)(s)) \ + \ o(s)$$

Or nous savons que si $w(t)$ est un processus de Wiener alors $\forall t, s \in \mathbb{R}^+ \, w(t + s) - w(t) \sim \mathcal{N}(0, sI_3)$. La différentielle de $w(t)$ suit donc une loi normale centrée, elle est donc d'espérence nulle. De plus $dw(t) = w(t + dt) - w(t)$ est indépendante des résultats passés, à fortiori son espérence conditionnelle par rapport à l'évènement $\{\mathbf{x}(t) = x\}$ est donc nulle. Alors, en prenant l'espérence conditionnelle par rapport à ce même évènement nous en déduisons la propriété suivante :

$$\mathbb{E}(\phi(\mathbf{x}(t + s)) - \phi(\mathbf{x}(t))|\mathbf{x}(t) = x) = s(\mathcal{L}_t\phi)(x) \ + \ o(s) \tag{25}$$

Considérons la quantité suivante :

$$\frac{d}{dt}(\mathbb{E}(\phi(\mathbf{x})))$$

Développons-la d'une première façon à l'aide d'une interversion dérivation-intégrale :

$$\begin{aligned} \frac{d}{dt}(\mathbb{E}(\phi(\mathbf{x}))) &= \frac{d}{dt}\int_{\mathbb{R}^3} \phi(x)\rho(x, t)dx \\ &= \int_{\mathbb{R}^3} \phi(x)\partial_t\rho(x, t)dx \end{aligned} \tag{26}$$

D'autre part en développant selon la définition de la dérivée puis en utilisant successivement l'équation de Chapman-Kolmogorov, Fubini et une interversion limite-intégrale nous obtenons les calculs suivants :

$$\begin{aligned}
\frac{d}{dt}(\mathbb{E}(\phi(\mathbf{x}))) &= \frac{d}{dt}\int_{\mathbb{R}^3}\phi(x)\rho(x,t)dx \\
&= \lim_{h\to 0}\frac{1}{h}\int_{\mathbb{R}^3}\phi(x)(\rho(x,t+h)-\rho(x,t))dx \qquad \text{(Définition de la dérivée)} \\
&= \lim_{h\to 0}\frac{1}{h}\int_{x}\phi(x)(P(t+h,dX;0,\Gamma_0)-P(t,dX;0,\Gamma_0)) \quad \text{(Réécriture en} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{termes de probabilité)} \\
&= \lim_{h\to 0}\frac{1}{h}\int_{x}\phi(x)(\int_{y}P(t+h,dX;t,\{y\})P(t,dY;0,\Gamma_0)-P(t,dX;0,\Gamma_0)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Chapman-Kolmogorov)} \\
&= \lim_{h\to 0}\frac{1}{h}(-\int_{y}\phi(y)P(t,dY;0,\Gamma_0)+\int_{x}\int_{y}\phi(x)P(t+h,dX;t,\{y\})P(t,dY;0,\Gamma_0) \\
&= \lim_{h\to 0}\frac{1}{h}(\int_{y}P(t,dY;0,\Gamma_0)(\int_{x}\phi(x)P(t+h,dX;t,\{y\})-\phi(y))) \qquad \text{(Fubini)} \\
&= \int_{y}P(t,dY;0,\Gamma_0)\lim_{h\to 0}\frac{1}{h}(\int_{x}\phi(x)P(t+h,dX;t,\{y\})-\phi(y)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(interversion limite-intégrale)}
\end{aligned}$$

Nou remarquons que :

$$\begin{aligned}
\mathbb{E}(\phi(\mathbf{x}(t+s))-\phi(\mathbf{x}(t))|\mathbf{x}(t)=x) &= \mathbb{E}(\phi(\mathbf{x}(t+s))|\mathbf{x}(t)=x)-\mathbb{E}(\phi(\mathbf{x}(t))|\mathbf{x}(t)=x) \\
&= \int_{y}\phi(y)P(t+h,dY;t,\{x\})-\phi(x)
\end{aligned}$$

Grâce à (25) nous obtenons l'équation :

$$\begin{aligned}
\frac{d}{dt}\mathbb{E}(\phi(\mathbf{x})) &= \int_{y}P(t,dY;0,\Gamma_0)\lim_{h\to 0}\frac{1}{h}(\int_{x}\phi(x)P(t+h,dX;t,\{y\})-\phi(y)) \\
&= \int_{y}P(t,dY;0,\Gamma_0)\lim_{h\to 0}\frac{1}{h}\mathbb{E}(\phi(\mathbf{x}(t+h))-\phi(\mathbf{x}(t))|\mathbf{x}(t)=y) \\
&= \int_{y}\mathcal{L}_t(\phi)(y)P(t,dY;0,\Gamma_0) \\
&= \int_{\mathbb{R}^3}\mathcal{L}_t(\phi)(y)\rho(y,t)dy \\
&= \int_{\mathbb{R}^3}\phi(y)\mathcal{L}_t^*(\rho(.,t))(y)dy \qquad (27)
\end{aligned}$$

Finalement des calculs (26) et (27) nous pouvons déduire l'égalité suivante :

$$\forall\phi\in C_c^\infty(\mathbb{R}^3,\mathbb{R}),\ \int_{\mathbb{R}^3}\phi(x)(\mathcal{L}_t^*(\rho(.,t))(x))\ dx=\int_{\mathbb{R}^3}\phi(x)\partial_t\rho(x,t)dx$$

Cette égalité est vraie pour toutes les fonctions C^∞ à support compact. Cette famille de fonctions étant dense dans $L^2(\mathbb{R}^3)$, nous pouvons déduire l'égalité entre les deux membres de l'intégrale ce qui s'écrit pour ρ :

$$\partial_t \rho - \mathcal{L}_t^*(\rho(.,t)) = 0 \tag{28}$$

Il nous reste encore à déterminer $\mathcal{L}_t^*$. Pour celà nous allons procéder à des intégrations par parties grâce aux formules de Green. Pour ce faire donnons nous f_1, f_2 deux fonctions $C_c^\infty(\mathbb{R}^3, \mathbb{R})$ à support compact. Commençons par rappeler les formules de Green.

Formule de Green-Gauss Soit Ω un ouvert de $\mathbb{R}^3$, soient $\vec{F}$ un champ de vecteurs de classe C^1 et $u \in C^1(\overline{\Omega}, \mathbb{R})$, la formule suivante est vraie :

$$\int_\Omega u\,(\nabla \cdot \vec{F})\,dV = -\int_\Omega \nabla u \cdot \vec{F}\,dV + \int_{\partial\Omega} u(\vec{F}\cdot\vec{n})\,dS \tag{29}$$

Si l'on considère maintenant deux fonctions $u, v \in C^2(\overline{\Omega}, \mathbb{R})$. ∇v et ∇u sont tous deux des champs de vecteurs de classe C^1. Nous pouvons donc utiliser cette formule pour obtenir les égalités suivantes :

$$\int_\Omega u\Delta v = \int_\Omega u\,(\nabla\cdot\nabla v)\,dV = -\int_\Omega \nabla u\cdot\nabla v\,dV + \int_{\partial\Omega} u(\nabla v\cdot\vec{n})\,dS$$

$$\int_\Omega v\Delta u = \int_\Omega v\,(\nabla\cdot\nabla u)\,dV = -\int_\Omega \nabla v\cdot\nabla u\,dV + \int_{\partial\Omega} v(\nabla u\cdot\vec{n})\,dS$$

En combinant les deux équations on obtient finalement une nouvelle formule :

$$\int_\Omega v\Delta u - u\Delta v = \int_{\partial\Omega} (v(\nabla u\cdot\vec{n}) - u(\nabla v\cdot\vec{n}))\,dS \tag{30}$$

Appliquons ces formules à nos deux fonctions f_1, f_2 à support compact en prenant Ω une boule ouverte centrée en 0 de rayon assez grand pour avoir $\forall x \in \mathbb{R}^3 \setminus \Omega\ \forall k \in \mathbb{N}, d^k(f_1)(x) = d^k(f_2)(x) = f_1(x) = f_2(x) = 0$ en développant nous obtenons :

$$\begin{aligned}
\int_{\mathbb{R}^3} f_1\mathcal{L}_t(f_2)\,dV &= \int_\Omega f_1(h(.,t)\cdot\nabla f_2 + \frac{\sigma^2}{2}\Delta f_2)\,dV = \int_\Omega f_1(h(.,t)\cdot\nabla f_2)\,dV + \int_\Omega \frac{\sigma^2}{2} f_1\Delta f_2\,dV \\
&= \int_\Omega f_1(h(.,t)\cdot\nabla f_2)\,dV + \frac{\sigma^2}{2}\int_\Omega f_2\Delta f_1\,dV \\
&\qquad \text{(utilisation de (30) et du fait que } f_{1|\partial\Omega} = f_{2|\partial\Omega} = 0) \\
&= -\int_\Omega f_2\nabla\cdot(f_1 h(.,t))\,dV + \frac{\sigma^2}{2}\int_\Omega f_2\Delta f_1\,dV \\
&\qquad \text{(utilisation de (6) avec } \vec{F} = f_1 h(.,t) \text{ et du fait que } f_{1|\partial\Omega} = f_{2|\partial\Omega} = 0) \\
&= \int_{\mathbb{R}^3} f_2(\frac{\sigma^2}{2}\Delta f_1 - \nabla\cdot(f_1 h(.,t)))\,dV
\end{aligned} \tag{31}$$

On peut déduire des calculs précédents (31) que dans le cas de deux fonctions C^∞ à support compact $\mathcal{L}_t^* = \frac{\sigma^2}{2}\Delta - \nabla\cdot(h(.,t).)$. Cette famille de fonction est dense

dans l'espace de Sobolev $H^2(\mathbb{R}^3)$ pour la norme usuelle. De plus la divergence et le Laplacien sont des opérateurs linéaires continus de $H^2(\mathbb{R}^3)$ dans $L^2(\mathbb{R}^3)$. Or la norme dans $L^2(\mathbb{R}^3)$ est plus grossière que la norme usuelle de $H^2(\mathbb{R}^3)$, si bien que si les familles $(f_1)_{n\in\mathbb{N}}$ et $(f_2)_{n\in\mathbb{N}}$ convergent vers $F_1, F_2 \in H^2(\mathbb{R}^3)$ pour la norme de $H^2(\mathbb{R}^3)$ alors $\langle f_{1,n}, \mathcal{L}_t^*(f_{2,n})\rangle_{L^2(\mathbb{R}^3)}$ converge vers $\langle F_1, \mathcal{L}_t^*(F_2)\rangle_{L^2(\mathbb{R}^3)}$. Nous pouvons en déduire que $\mathcal{L}_t^* = \frac{\sigma^2}{2}\Delta - \nabla\cdot(h(.,t).)$ pour le produit scalaire de $L^2(\mathbb{R}^3)$ sur l'espace $H^2(\mathbb{R}^3)$. Et finalement l'équation de Fokker-Planck pour ρ :

$$\boxed{\partial_t\rho - \frac{\sigma^2}{2}\Delta\rho \ + \ \nabla.(\rho h) = 0} \tag{24}$$

Nous admettrons en partie l'implication inverse dont la démonstration complète sort du cadre d'étude de ce projet. Nous n'en donnons qu'une ébauche ci dessous :

- Il est d'abord nécessaire de montrer que la densité de probabilité conditionnelle par rapport à la filtration $\{\mathbf{x}(s), s \le u\}$ pour $u \in \mathbb{R}^+$ vérifie l'équation de Fokker-Planck.
- Les calculs effectués en (26) et (27) peuvent alors être réadaptés pour montrer que quelque soit $\phi \in H^2(\mathbb{R}^3)$:

  $\lim_{h\to 0}\frac{1}{h}\mathbb{E}(\mathbb{E}(\phi(\mathbf{x}(u_2+h))-\phi(\mathbf{x}(u_2)) \mid \mathbf{x}(s), s\le u_2) \mid \mathbf{x}(s), s\le u_1) = \mathbb{E}(\mathcal{L}_t\phi(\mathbf{x}(u_2)) \mid \mathbf{x}(s), s\le u_1)$

  Après intégration par rapport à la variable u_2 on obtient le résultat suivant :

  $$\mathbb{E}(\phi(\mathbf{x}(u_2)) - \phi(\mathbf{x}(u_1)) - \int_{u_1}^{u_2}\mathcal{L}_s(\phi)(\mathbf{x}(t))\ dt|\mathbf{x}(s), s\le u_1) = 0$$

  Qui traduit le fait que le processus stochastique $\phi(\mathbf{x}(t)) - \int_0^t \mathcal{L}_s(\phi)(\mathbf{x}(s))\ ds$ est une martingale.
- La fin de la preuve consiste à appliquer le théorème de représentation des martingales dans le cas Brownien afin de deduire l'équation stochastique vérifiée par le processus $\mathbf{x}$.

Replaçons nous dans le cadre de notre hypothèse $h = \mathbf{u}$. Notre problème est maintenant résumé sous la forme du problème d'optimisation suivant :

$$\begin{aligned} &\inf_{\rho,u}\mathcal{J}(\rho(x,t),\mathbf{u}(x,t)) \\ \text{avec}\quad &\partial_t\rho - \frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u}) = 0, \\ &\rho(x,0) = \rho_0(x), \end{aligned}$$

avec ρ_0 la formation initialement choisie.

2.4 Réécriture du problème d'optimisation

Premièrement nous supposerons dans cette section comme c'est le cas dans le cadre pratique que la fonction de densité ρ appartient à $H^2(\mathbb{R}^4)$.

Comme il est d'usage dans de nombreux problèmes d'optimisation considérons le Lagrangien associé, il s'écrit :

$$\begin{aligned}\mathcal{L}(\rho,\mathbf{u},\phi) &= \mathcal{J}(\rho,\mathbf{u}) \ + \ \langle \phi, \partial_t\rho - \frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u})\rangle \quad \text{(En prenant le produit scalaire de } L^2(\mathbb{R}^4)) \\ &= \int_0^T \left\{ \int_\Omega \frac{\|\mathbf{u}\|^2}{2}\rho(x,t)\,dx + \mathcal{F}(\rho(.,t)) \right\} dt + \mathcal{G}(\rho(\cdot,T)) \\ &\quad + \int_0^T \left\{ \int_\Omega \phi(x,t)(\partial_t\rho(x,t) - \frac{\sigma^2}{2}\Delta\rho(x,t) \ + \ \nabla\cdot(\rho.\mathbf{u})(x,t)\ dx) \right\} dt \end{aligned} \tag{32}$$

Le problème de minimisation précédent est donc équivalent au problème suivant :

$$\inf_{\rho,\mathbf{u}} \sup_{\phi} \mathcal{L}(\rho,\mathbf{u},\phi)$$

(De plus $\mathcal{J}(\rho,\mathbf{u})$ est infinie à l'infini dans le fermé défini par nos contraintes. En effet si... $(\mathbf{u}_n,\rho_n)$ est une suite non bornée En admettant que la fonction de coûts est convexe pour les familles de fonctions $(\rho,\mathbf{u})$ qui vérifient la contrainte imposée par l'équation de Fokker-Planck. $\mathcal{J}$ admet donc un minimum global vérifiant nos contraintes, qui d'après le théorème de Kuhn et Tucker, correspond à l'unique point selle du Lagrangien. Le Lagrangien admet donc un point selle, le théorème de dualité forte nous permet d'affirmer que le problème primal est équivalent au dual ci-dessous) :

$$\sup_{\phi}\inf_{\rho,u} \int_0^T \left\{ \int_\Omega \rho(x,t)\frac{\|u(x,t)\|^2}{2}\,dx + \mathcal{F}(\rho(\cdot,t)) \right\} dt + \mathcal{G}(\rho(\cdot,T)) \ + \ \int_0^T\int_\Omega \phi(x,t)\left(\partial_t\rho - \frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u})(x,t)\right) dx\,dt \tag{33}$$

Comme mentionné dans la section précédente, l'ensemble des fonctions $C^\infty \cap L^2$ est dense dans L^2, le suprémum de notre Lagrangien sur L^2 est donc égal au suprémum sur $C^\infty \cap L^2$. Nous pouvons donc supposer sans perdre de généralité que $\phi \in C^\infty$.

Alors, en développant le terme de droite nous obtenons les égalités suivantes :

$$\begin{aligned}
&\int_0^T \int_\Omega \phi(x,t)\left(\partial_t\rho - \frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u})(x,t)\right)dx\,dt \\
&= \int_0^T \int_\Omega \phi(x,t)\left(-\frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u})(x,t)\right)dx\,dt \;+\; \int_\Omega\int_0^T \phi(x,t)\partial_t\rho(x,t)dt\,dx \quad \text{(Fubini)} \\
&= \int_0^T \int_\Omega \phi(x,t)\left(-\frac{\sigma^2}{2}\Delta\rho + \nabla\cdot(\rho.\mathbf{u})(x,t)\right)dx\,dt \;+\; \int_\Omega \phi(x,T)\rho(x,T)dx - \int_\Omega \phi(x,0)\rho(x,0)dx \\
&\quad - \int_\Omega\int_0^T \partial_t\phi(x,t)\rho(x,t)dt\,dx \qquad \text{(Intégration par parties sur } \mathbb{R}) \\
&= -\int_0^T \int_\Omega \left(\mathbf{u}\cdot\nabla\phi \;+\; \frac{\sigma^2}{2}\Delta\phi\right)\rho(x,t)dx\,dt \;+\; \int_\Omega \phi(x,T)\rho(x,T)dx - \int_\Omega \phi(x,0)\rho(x,0)dx \\
&\quad - \int_\Omega\int_0^T \partial_t\phi(x,t)\rho(x,t)dt\,dx \qquad\qquad (34)
\end{aligned}$$

(On a calculé dans la section précédente l'adjoint de l'opérateur $\frac{\sigma^2}{2}\Delta - \nabla\cdot(h.)$ sur $L^2(\mathbb{R}^3)$ de plus toutes les fonctions sont à support compact sur Ω)

En réinsérant l'expression ci-dessus nous sommes amenés à résoudre :

$$\sup_\phi \inf_{\rho,\mathbf{u}} \int_0^T \left\{\int_\Omega \left(\partial_t\phi + \frac{\sigma^2}{2}\Delta\phi + \frac{\|\mathbf{u}(x,t)\|^2}{2} + h\cdot\nabla\phi\right)\rho(x,t)\,dx + \mathcal{F}(\rho(\cdot,t))\right\}dt + \int_\Omega \phi(x,0)\rho_0(x)\,dx - \int_\Omega \phi(x,T)\rho(x,T)\,dx + \mathcal{G}(\rho(\cdot,T)). \tag{35}$$

La formule ci-dessus ne contient plus qu'un terme dépendant de $\mathbf{u}$:

$$\int_0^T \int_\Omega \left(\frac{\|\mathbf{u}(x,t)\|^2}{2} + \mathbf{u}\cdot\nabla\phi(x,t)\right)\rho(x,t)dx\,dt$$

L'infimum de cette expression selon $\mathbf{u}$ ne dépend donc que de ce terme. Nous avons l'égalité suivante :

$$\begin{aligned}
\inf_{\mathbf{u}} \int_0^T \int_\Omega \left(\frac{\|\mathbf{u}(x,t)\|^2}{2} + \mathbf{u}\cdot\nabla\phi(x,t)\right)\rho(x,t)dx\,dt &= \int_0^T \int_\Omega \inf_u \left\{\frac{\|u\|^2}{2} + u\cdot\nabla\phi(x,t)\right\}dx\,dt \\
&= \int_0^T \int_\Omega H(x,t,\nabla\phi)dx\,dt
\end{aligned}$$

Où H est un Hamiltonien défini par $H(x,t,p) = \inf_u \frac{\|u\|^2}{2} + u\cdot p(x,t)$.

En effet la fonction $\frac{\|u\|^2}{2} + u\cdot p$ est α convexe, elle admet donc un unique minimum global atteint en $u \in \mathbb{R}^3$. Les conditions d'optimalité en u s'écrivent : $u+p=0$. On a donc $u=-p$. Par conséquent si l'on considère la fonction $\mathbf{u} = -\nabla\phi$, celle-ci minimise terme à terme $\frac{\|u\|^2}{2} + u\cdot\nabla\phi(x,t)$ or cette fonction appartient bien à l'ensemble fermé sur lequel non minimisons notre coût, elle correspond donc à l'unique fonction minimisante, d'où l'égalité ci-dessus. En outre nous obtenons $H(x,t,\nabla\phi) = \frac{-\|\nabla\phi(x,t)\|^2}{2}$.

En insérant l'égalité ci-dessus notre problème s'écrit finalement :

$$\sup_{\phi}\inf_{\rho}\int_0^T\left\{\int_\Omega\left(\partial_t\phi+\frac{\sigma^2}{2}\Delta\phi+H(x,t,\nabla\phi)\right)\rho(x,t)\,dx+\mathcal{F}(\rho(\cdot,t))\right\}dt+\int_\Omega\phi(x,0)\rho_0(x)\,dx-\int_\Omega\phi(x,T)\rho(x,T)\,dx+\mathcal{G}(\rho(\cdot,T)) \tag{36}$$

2.5 Implémentation de réseaux de neurones de type RAG

Evidemment la résolution théorique d'un telle problème semble très difficile pour ne pas dire impossible. Une approche numérique est ainsi nécessaire. On choisit d'implémenter des réseaux de neurones de types RAG (comme présentés en introduction) pour résoudre informatiquement ce problème. Dans notre cas, le générateur cherche à simuler au mieux ρ quand le discriminateur tente de similuer au mieux ϕ, la fonction de valeur. On introduit deux réseaux de neurones $N_\theta(z,t)$ qui rend des vecteurs dans $\mathbb{R}^3$ et $N_\omega(x,t)$ qui renvoie un nombre. On crée à partir de ces réseaux les deux quantités :

$$\phi_\omega(\mathbf{x},t)=(1-\frac{t}{T})N_\omega(\mathbf{x},t)+\frac{t}{T}g(\mathbf{x}),\quad G_\theta(\mathbf{z},t)=(1-\frac{t}{T})\mathbf{z}+\frac{t}{T}N_\theta(\mathbf{z},t), \tag{37}$$

ces deux expressions étant choisies pour encoder les conditions initiales et finales. En effet, par exemple à $t=0$, $G_\theta(z,0)=z$ qui renvoie bien la position initale, et à $t=T$, $\phi_\omega(x,T)$ renvoie le coût fixe g. Pour entrainer nos réseaux de neurones, on génère un certain nombre K de variables aléatoires de position $(z_k)_{1\leq k\leq K}$ suivant la distribution ρ_0 et K variables aléatoires de temps $(t_k)_{1\leq k\leq K}$ suivant la loi uniforme sur $[0,T]$. On définit alors les fonctions de côut suivantes, qui d'après la loi des grands nombres, doivent converger vers la quantité que l'on cherche à optimiser dans (20) :

$$\text{cout}_\phi=\frac{1}{K}\sum_{k=1}^K\phi_\omega(x_k,0)+\frac{1}{K}\sum_{k=1}^K\left(\partial_t\phi_\omega(x_k,t_k)+\frac{\sigma^2}{2}\Delta\phi_\omega(x_k,t_k)+H(\nabla_x\phi_\omega(x_k,t_k),x_k)\right), \tag{38}$$

$$\text{cout}_G=\frac{1}{K}\sum_{k=1}^K\left(\partial_t\phi_\omega(G_\theta(z_k,t_k),t_k)+\frac{\sigma^2}{2}\Delta\phi_\omega(G_\theta(z_k,t_k),t_k)+H(\nabla_x\phi_\omega(G_\theta(z_k,t_k),t_k),x_k)+f(G_\theta(z_k,t_k),t_k)\right). \tag{39}$$

- L'optimisation se fait grâce aux réseaux de neurones qui réalisent des descentes de gradient pour minimiser les fonctions de côut.

3
IMPLÉMENTATION NUMÉRIQUE

3.1 Réseaux de neuronnes

Commençons par une présentation générale des réseaux de neurones suivi d'une courte du modèle GAN.

3.1.1 • Définition

Un réseau de neurones est composé de plusieurs couches, chacune composée de un ou plusieurs neurones. Un neurone contient une valeur dans $[0, 1]$ (pour une fonction d'activation sigmoïde) et est relié à des neurones de la couche précédente et de la couche suivante. La première couche contient les informations passées en entrée au réseau. Exemple : Si l'entrée est une image en noir et blanc de 32 par 32 pixels, on aura attribué 1024 neurones à la première couche. La dernière couche est la sortie du réseau de neurones. Exemple pour un réseau de neurones qui prend une image en entrée.

- Le réseau détermine quel chiffre est représenté → dix neurones correspondant à $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ dans la dernière couche.
- Le réseau détermine si l'image représente un chiffre → deux neurones pour accepté ou rejetté dans la dernière couche.
- Le réseau génère une image en 16x16 → 256 neurones dans la dernière couche.

Quant aux couches intermédiaires, leur nombre et leur composition sont choisis lors de la création du réseau. Chaque couche récupère l'information de la couche qui la précède et transmet à la couche qui la suit. L'exemple suivant pour trois couches de trois neurones se généralise à toute échelle.

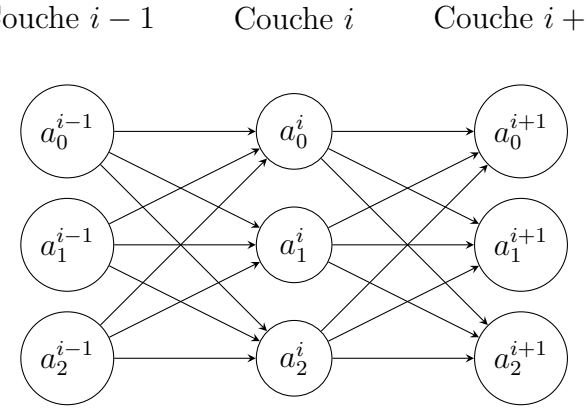


Figure 1 – Réseau de neurones à trois couches

La valeur de a_1^i est calculée par : $a_1^{(i)} = \sigma(w_{1,0}^{(i)} \cdot a_0^{(i-1)} + w_{1,1}^{(i)} \cdot a_1^{(i-1)} + w_{1,2}^{(i)} \cdot a_2^{(i-1)} + b_1^{(i)})$ Où les coefficients $w_{k,j}^i$ et b_k^i sont des paramètres du réseau, et σ est la fonction d'activation (ramène $\mathbb{R}$ sur $]0, 1[$ de manière continue, strictement croissante

et bijective. Par exemple $\sigma(x) = \frac{1}{1+e^{-x}}$) Plus formellement, si les couches $i-1$ et i ont respectivement n et m neurones, on a

pour tout $0 \leqslant j \leqslant m-1$, $\quad a_j^{(i)} = \sigma\left(\sum_{k=0}^{n-1} w_{j,k}^{(i)} \cdot a_k^{(i-1)} + b_j^{(i)}\right)$

Si on prend l'exemple dans lequel on veut déterminer le chiffre représenté dans une image en noir et blanc de 32x32 pixels, le réseau de neurones est une fonction $f_{(w_{k,j}^{(i)})_{i,j,k},(b_j^{(i)})_{i,j}} : \mathbb{R}^{1024} \to [0,1]^{\{0,1,2,3,4,5,6,7,8,9\}}$ Si on choisit de donner quatre couches à ce réseau, et de donner 16 neurones aux couches intermédiaires, on obtient 16842 paramètres $(w_{k,j}^{(i)})_{i,j,k}$ et $(b_j^{(i)})_{i,j}$. Entraîner le réseau de neurones revient à trouver des paramètres qui minimisent le coût sur un ensemble d'entraînement donné.

3.1.2 • Fonction coût

On dispose d'une entrée In (Ex : une image), d'une sortie attendue $Out\_Expected$ (Ex : le chiffre représenté sur l'image), de la sortie du réseau de neurones Out (Ex : un chiffre). Le coût est défini par $||Out - Out\_Expected||^2$. C'est ce coût qu'on cherche à minimiser puisqu'on veut que le réseau donne des résultats les plus proches possible des résultats attendus.Exemple :

Figure 2 – Entrée donnée au réseau de neuronnes

Sortie du réseau : $\begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.7 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.9 \end{pmatrix}$ Résultat attendu : $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Minimiser ce coût revient à diminuer simultanément les faux positifs et les faux négatifs.

Pour un couple $(In, Out\_Expected)$, on peut définir une fonction coût du réseau de neurones qui dépend des paramètres : $C_{In,Out\_Expected}\left(\begin{pmatrix} w \\ b \end{pmatrix}\right)$

3.1.3 • Entraînement

Pour entraîner le réseau de neurones, on calcule $\nabla C_{In,Out\_Expected}\left(\begin{pmatrix} w \\ b \end{pmatrix}\right)$, et on déplace le vecteur $\begin{pmatrix} w \\ b \end{pmatrix}$ dans le sens opposé au gradient : $\begin{pmatrix} w \\ b \end{pmatrix} \leftarrow \begin{pmatrix} w \\ b \end{pmatrix} - \eta \cdot \nabla C\left(\begin{pmatrix} w \\ b \end{pmatrix}\right)$ Le pas η utilisé pour modifier ces valeurs est appelé le taux d'apprentissage, ou learning rate. C'est un paramètre clef dans l'optimisation, en effet, un pas trop grand risque de dépasser le minimum de C et de ne pas converger, et un pas trop petit demande un nombre d'itérations inutilement grand.

Concrètement, pour calculer $\nabla C\left(\begin{pmatrix} w \\ b \end{pmatrix}\right)$, pytorch (la bibliothèque que nous utilisons) construit un graphe qui suit toute les opérations ayant mené à C, et calcule les dérivées partielles de chaque $w_{k,j}^{(i)}$ et $b_k^{(i)}$ en remontant ce graphe.

```
a = torch.tensor(2.0,requires_grad = True)
b = torch.tensor(3.0,requires_grad = True)
c = a * b
d = torch.tensor(5.0,requires_grad = True)
e = c * d
e.backward()
```

Lors de l'exécution de ce code, torch crée le graphe ci dessous. La ligne `e.backward()` calcule les dérivées partielles de e par rapport à chacune des composantes précédents, et chaque tensor `x` aura pour attribut `grad` la dérivée partielle $\frac{\partial e}{\partial x}$.

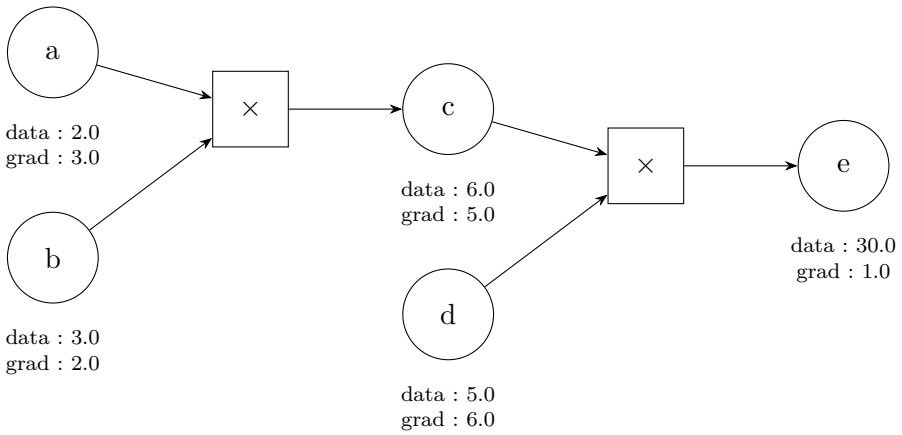


Figure 3 – Graphe de calcul généré par PyTorch

3.1.4 • Le modèle GAN (Generative Adversarial Neural network)

La partie précédente traite de l'apprentissage d'un réseau de neurones par intervention humaine en donnant des couples $(input, output)$ sur lesquels le réseau s'entraîne. Il s'agit d'apprentissage supervisé. Un GAN est un modèle de réseaux de neurones qui fonctionne en apprentissage non supervisé. Deux réseaux : G le générateur et D le discriminateur s'affrontent. Un GAN fonctionne comme un jeu à deux joueurs. Le générateur G cherche à tromper le discriminateur D. En entraînant le GAN, on résout un problème de type minimax :
$\min_G \max_D V(D,G)$ avec $V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$
D maximise sa probabilité de trouver les faux, et G minimise cette probabilité en générant des données de plus en plus crédibles. Lors d'une phase d'entraînement, si D détecte le faux créé par G, il gagne et G est mis à jour. Sinon, D est mis à jour.

Les GAN permettent un apprentissage non supervisé quand on ne dispose pas de bases de données suffisantes. Ils ont été largement utilisés pour la génération d'images, de parole...

On a opté pour une architecture de GAN basée sur un réseau résiduel (ResNet) qui est particulièrement adaptée à la résolution de problèmes d'optimisation sous contrainte de type contrôle optimal. L'ajout de connexions résiduelles (le terme $+\,\texttt{self.skip\_weight} \times x$) permet de stabiliser l'entraînement, favorisant le passage du gradient à travers les couches, ce qui est essentiel dans l'approximation de fonctions complexes et profondes telles que la fonction de valeur ou la densité dans notre cas. C'est l'architecture souvent utilisée pour la modélisation des solutions des équations de type Hamilton-Jacobi-Bellman (HJB) ou Fokker-Planck (FP).

Ce type de réseau s'intègre naturellement dans les approches modernes telles que les Physics-Informed Neural Networks (PINNs) ou les Jeux à Champ Moyen (MFG), où l'on cherche à imposer les équations de contrôle (HJB, FP, ou leurs couplages) directement dans la fonction de perte. La structure résiduelle permet de

conserver une bonne différentiabilité de bout en bout, indispensable pour appliquer efficacement les méthodes de descente de gradient sur des pertes qui impliquent des dérivées partielles de la sortie du réseau.

```python
class ResBlock(nn.Module):
    """
    Bloc résiduel : applique une transformation linéaire,
    une activation non-linéaire, puis ajoute une connexion
    directe pondérée à l'entrée initiale.
    """
    def __init__(self, in_features, out_features, activation=nn.ReLU(), skip_weight=0.5):
        super(ResBlock, self).__init__()
        self.linear = nn.Linear(in_features, out_features)
        self.activation = activation
        self.skip_weight = skip_weight

    def forward(self, x):
        return self.activation(self.linear(x)) + self.skip_weight * x


class ResNet(nn.Module):
    """
    Réseau résiduel simple :
    - Couche d'entrée suivie d'une tanh
    - Trois blocs résiduels
    - Couche de sortie linéaire
    """
    def __init__(self, input_dim, output_dim, hidden_dim=100, activation=nn.ReLU()):
        super(ResNet, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.resblock1 = ResBlock(hidden_dim, hidden_dim, activation)
        self.resblock2 = ResBlock(hidden_dim, hidden_dim, activation)
        self.resblock3 = ResBlock(hidden_dim, hidden_dim, activation)
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.tanh(self.input_layer(x))
        x = self.resblock1(x)
        x = self.resblock2(x)
        x = self.resblock3(x)
        return self.output_layer(x)
```

3.2 En pratique : définition des fonctions de coût et entraînement

Dans ce paragraphe, on s'intéresse à certains partis pris dans la rédaction du code que l'on trouve en annexe qui implémente la méthode GAN pour résoudre notre problème d'optimisation. Le premier de ces partis pris concerne la fonction de densité souhaitée ρ_0. Expliquons comment nous générons cette fonction dans notre programme. Nous choisissons un ensemble de points, noté P (les points doivent être de sorte que l'on ait ρ_0 de moyenne nulle), dans l'espace où l'on souhaite inscrire la formation, puis nous définissons ρ_0 par

$$\forall (x,y,z) \in \mathbb{R}^3, \quad \rho_e(x,y,z) = \frac{1}{(2\pi\sigma^2)^{3/2}\,\#P} \sum_{p\in P} \exp\left(-\frac{\|(x,y,z)-p\|^2}{2\sigma^2}\right).$$

Par exemple, pour encoder une densité initiale triangulaire, P sera les 3 sommets de mon triangle. Dans cette équation, $\sigma > 0$ désigne l'écart type. Ce paramètre permet d'ajuster la précision : plus sa valeur est petite, plus les pics redescendent fort autour des points passés, mais plus il faut d'époques pour entraîner un modèle fonctionnel dont les performances soient satisfaisantes. Cela est implémenté dans le programme :

```
def rho(xyz: torch.Tensor) -> torch.Tensor:  # xyz : (N, 3)
    """
    Calcule la densité attendue en chaque point xyz.
    xyz : tenseur de forme (N, 3)
    Retour : tenseur 1-D de taille N
    """
    diff  = xyz.unsqueeze(1) - points.unsqueeze(0)      # (N, n, 3)
    dist2 = (diff ** 2).sum(dim=-1)                     # (N, n)
    gauss = torch.exp(-dist2 / (2 * variance))          # (N, n)
    norm  = (2 * torch.pi * variance) ** 1.5            # (2²)^{3/2}
    return gauss.sum(dim=1) / (n * norm)                # (N,)
```

Ces lignes de code nous permettent d'obtenir une densité caractérisant la formation souhaitée. Il est essentiel d'utiliser exclusivement les fonctions de la bibliothèque `torch` afin de conserver le calcul automatique du gradient et de pouvoir optimiser ensuite les fonctions de coût qui les exploitent. Pour évaluer le coût de formation, défini par

$$\mathcal{F}_{\text{formation}} = \int_\Omega |\rho_{centrée} - \rho_0|\,\mathrm{d}x,$$

il est indispensable de disposer d'une approximation de la densité réelle ρ. En pratique, nous ne savons qu'échantillonner cette densité ; il faut donc construire un *estimateur*.

Notre procédure est la suivante : nous générons d'abord un échantillon de points selon ρ, puis nous le *centrons* en soustrayant la moyenne empirique afin d'obtenir

l'ensemble B. L'intuition est qu'une formation visée peut correspondre, par exemple, à une densité à trois pics décrivant un triangle. Si l'échantillon n'était pas recentré, les points simulés se situeraient dans un emplacement spatial potentiellement très éloigné des pics de ρ_0, et l'apprentissage se résumerait alors à « maintenir les drones au mauvais endroit ».

À partir de l'ensemble centré B, nous retenons le noyau gaussien suivant comme estimateur de densité :

$$\forall (x,y,z) \in \mathbb{R}^3, \quad \hat{\rho}(x,y,z) = \frac{1}{(2\pi\sigma^2)^{3/2}\,\#B} \sum_{p \in B} \exp\left(-\frac{\|(x,y,z)-p\|^2}{2\sigma^2}\right).$$

Le paramètre σ est identique à celui employé dans ρ_e. Les lignes de code ci-dessous implémentent exactement les idées présentées précédemment :

```python
def f_formation(x, device=torch.device("cpu")):
    # Centrage
    x_centered = x - x.mean(dim=0, keepdim=True)  # (N, 3)

    # Densité estimée à partir de x_centered
    sigma = torch.sqrt(torch.tensor(0.05))
    def density_estimated(pts):
        diff = pts.unsqueeze(1) - x_centered.unsqueeze(0)  # (M, N, 3)
        dist2 = (diff ** 2).sum(dim=-1)  # (M, N)
        gaussians = torch.exp(-dist2 / (2 * sigma**2))  # (M, N)
        norm_const = torch.sqrt(torch.tensor(2 * torch.pi, device=device)) * sigma
        return gaussians.sum(dim=1) / (x_centered.shape[0] * norm_const)

    # Distance L1 entre les deux
    d = distance_L1_torch(density_real, density_estimated, n_grid=40, device=device)
    return d
```

Au début du projet, nous avons fait appel à plusieurs bibliothèques spécialisées pour calculer l'intégrale avec une grande précision. Toutefois, cette solution exigeait une puissance de calcul considérable et allongeait le temps d'exécution de manière spectaculaire. Nous avons donc choisi de sacrifier une partie de la précision pour réduire à la fois la complexité et la durée des calculs : l'intégrale est désormais approchée par une *somme de Riemann tridimensionnelle*.

```python
def distance_L1_torch(p_func, q_func, n_grid=10,
                      a=-1.0, b=3.0,
                      device=torch.device("cpu")):
    coords = torch.linspace(a, b, n_grid, device=device)
    dx = (b - a) / n_grid
    grid = torch.stack(torch.meshgrid(coords, coords, coords,
                                      indexing='ij'), dim=-1)  # (n, n, n, 3)
    flat_grid = grid.view(-1, 3)                               # (n^3, 3)
```

```
9
10      p_vals = p_func(flat_grid)  # (n^3,)
11      q_vals = q_func(flat_grid)  # (n^3,)
12
13      return torch.sum(torch.abs(p_vals - q_vals)) * dx**3
```

Cette approche présente un autre atout majeur : en restant entièrement dans l'écosystème `torch`, on conserve la différentiabilité de l'algorithme. Il est ainsi possible d'optimiser la fonction de formation par descente de gradient. Cependant, les pics associés à la mixture de Gaussienne nous permettant d'estimer nos densités (initiales comme à un instant donné) étant étroits, il est nécessaire de considérer une valeur de n_grid relativement élevée, en pratique on a bien souvent pris $n\_grid = 40$ ou $n\_grid = 50$.

3.3 Résultats et mise en perspective

On présente ici certains des résultats obtenus grâce à nos simulations numériques. La majorité des tests ont été conduits avec des formations polygonales, principalement triangulaire et pentagonale. Notre démarche expérimentale a consisté à implémenter les contraintes une à une. On s'est d'abord assurés que les drones, dans un environnement sans obstacles, convergeaient bien en moyenne vers le point donné. La convergence de l'entraînement était rapide, seulement 200 époques furent amplement nécessaires pour que les drones atteignent leur objectif, et efficace : le programme faisait tourner près de 10 époques par seconde.

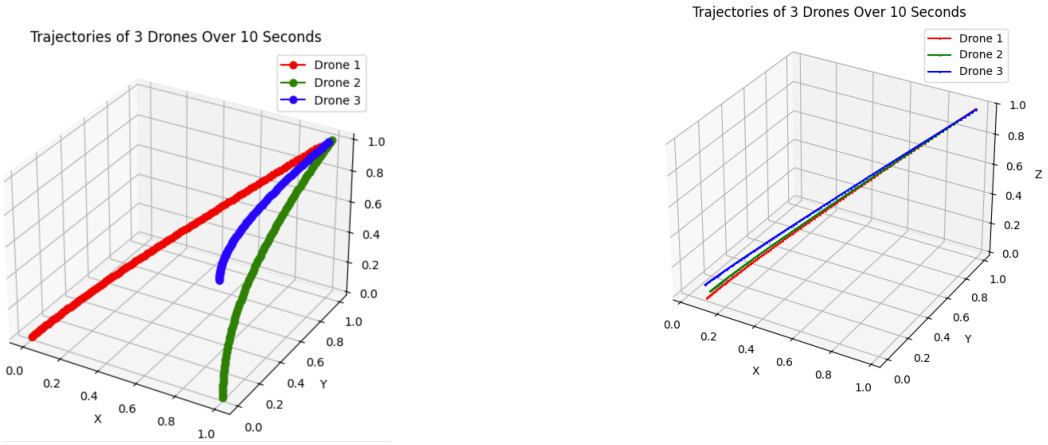


Figure 4 – Convergence en moyenne de 2 formations triangulaires vers $x_f = [1, 1, 1]$

Nous avons ensuite cherché à ajouter en plus la contrainte des obstacles et de la non collision des drones entre eux. Cette contrainte a légèrement diminué l'efficacité

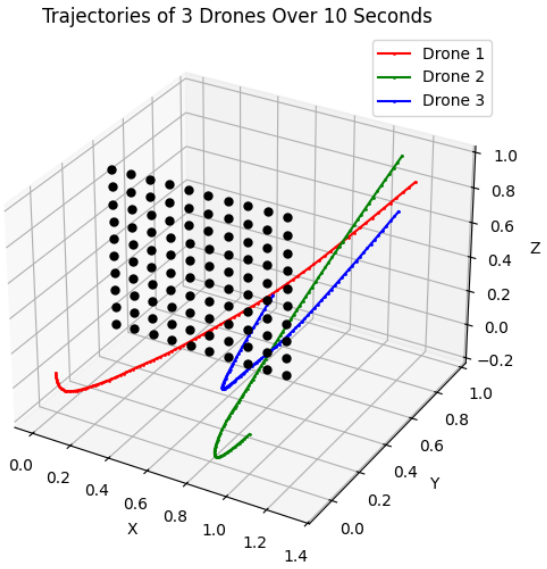
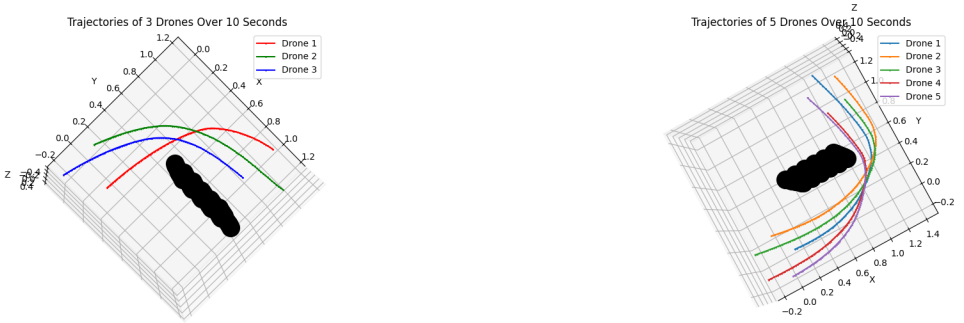


Figure 5 – Pas de prise en compte de la formation. Les obstacles sont des boules de rayon 0.02cm. $x_f = [1, 1, 1]$

de compilation du code : près de 5 époques par seconde étaient compilées. Cependant, la vitesse de convergence du modèle a nécessité la simulation de 500 époques dans des environnements denses d'obstacles (voir Fig 5). Par ailleurs, l'implémentation de cette nouvelle contrainte a également révélé la nécessité de pondérer les coûts pour aiguiller la descente de gradient et réaliser simultanément plusieurs objectifs. En effet, nous avons constaté que pondérer à la hausse le coût final de convergence en moyenne de la formation était indispensable pour la convergence (tout court et efficacement) de l'essaim de drones. Nous avons ainsi imposé $\alpha_4 = 10$ pour cette épreuve (cf. équation (19))

Enfin, nous avons implémenté la dernière contrainte, celle de la formation, la plus dure. En effet, celle-ci a considérablement fait baisser la vitesse de convergence, tant un minimum de 1500 époques était nécessaire pour observer une convergence satisfaisante des fonctions de coût. Par ailleurs, comme l'on s'y attendait, la complexité du programme a beaucoup augmenté, passant de plusieurs époques à la seconde à une époque toutes les quelques secondes (en fonction de la formation). L'implémentation de la fonction de coût relative à la formation s'est déclinée en deux étapes. La première consistait à ne prendre en compte que la formation finale pour s'assurer qu'après avoir évité les obstacles, l'essaim se remettait en formation. Le résultat de cette tentative est représenté Figure 6. On observe que les drones ont en quelque sorte était permuté entre l'état initial et l'état final. La deuxième étape dans cette implémentation était le contrôle de la formation à intervalle régulier du trajet. On a pris en pratique tous les 5e ou tous les 10e de trajet. Ceci a permis de maintenir effi-

cacement la formation tout au long du trajet tout en évitant l'obstacle. Notons qu'ici aussi une pondération des fonctions de coût s'est avérée nécessaire pour la réalisation simultanée des différents objectifs. Nous avons ainsi pris $\alpha_1 = 100$, $\alpha_2 = \alpha_3 = 1$ et $\alpha_4 = 500$. Les résultats de ces simulations sont visibles Figure 7.

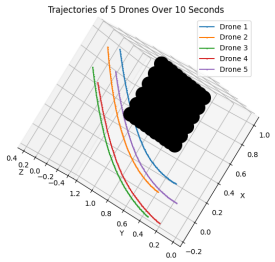


(a) Formation triangulaire

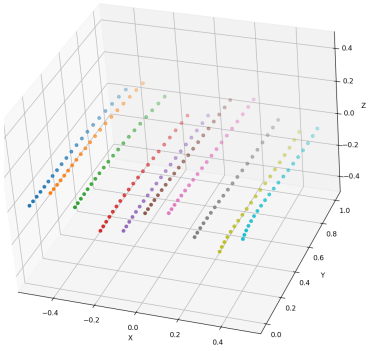
(b) Formation en pentagone

Figure 6 – Evitement d'un mur d'obstacles puis reconfiguration en formation. $x_f = [1, 1, 0]$

Cette figure illustre la capacité de notre modèle à générer des trajectoires coordonnées pour un essaim de 5 drones en en formation en pentagone, et pour un essaim de10 drones, évoluant dans une formation complexe en vague. Malgré la difficulté associée à ce type de configuration, caractérisée par des variations synchronisées dans l'espace et le temps, le modèle parvient à maintenir une cohérence de groupe tout en évitant les obstacles. Cela démontre non seulement la flexibilité du réseau dans l'apprentissage de formations non triviales, mais également sa robustesse dans des environnements contraints.



(a) Formation en pentagone



(b) Formation en vague

Figure 7 – Formations initiales en pentagone et en vague maintenues tout au long du trajet. $x_f = [1, 1, 0]$

4 ANNEXE

```python
##########################################################################
# Bloc 1 : Importations et configuration globale
##########################################################################
import math as m
import torch
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
from sklearn.neighbors import KernelDensity
from mpl_toolkits.mplot3d import Axes3D  # Pour les tracés 3D

# Détection du GPU si disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


##########################################################################
# Bloc 2 : Définition des blocs de base et des réseaux
##########################################################################

# Bloc résiduel : applique une transformation linéaire suivie d'une activation, avec une
class ResBlock(nn.Module):
    def __init__(self, in_features, out_features, activation=nn.ReLU(), skip_weight=0.5):
        super(ResBlock, self).__init__()
```

```
        self.linear = nn.Linear(in_features, out_features)
        self.activation = activation
        self.skip_weight = skip_weight

    def forward(self, x):
        return self.activation(self.linear(x)) + self.skip_weight * x

# Réseau résiduel : couche d'entrée, trois blocs résiduels, et une couche de sortie
class ResNet(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim=100, activation=nn.ReLU()):
        super(ResNet, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.resblock1 = ResBlock(hidden_dim, hidden_dim, activation)
        self.resblock2 = ResBlock(hidden_dim, hidden_dim, activation)
        self.resblock3 = ResBlock(hidden_dim, hidden_dim, activation)
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.tanh(self.input_layer(x))
        x = self.resblock1(x)
        x = self.resblock2(x)
        x = self.resblock3(x)
        return self.output_layer(x)

# Réseau NOmega : approxime la fonction valeur  (réseau critère du GAN)
class NOmega(nn.Module):
    def __init__(self):
        super(NOmega, self).__init__()
        self.net = ResNet(input_dim=4, output_dim=1, activation=nn.Tanh())

    def forward(self, x, t):
        input_data = torch.cat([x, t], dim=-1)
        return self.net(input_data)

# Réseau NTheta : génère les trajectoires (réseau générateur du GAN)
class NTheta(nn.Module):
    def __init__(self):
        super(NTheta, self).__init__()
        self.net = ResNet(input_dim=4, output_dim=3, activation=nn.ReLU())

    def forward(self, z, t):
        input_data = torch.cat([z, t], dim=-1)
        return self.net(input_data)


######################################################################
```

```python
# Bloc 3 : Conditions aux bords et fonctions utilitaires
#########################################################################

# Cible finale moyenne des trajectoires (formation visée)
x_target = torch.tensor([1, 1, 0])

# Fonction de coût terminal utilisée dans la condition de bord de
def g(x):
    return torch.norm(x.mean(dim=0) - x_target.to(device))

# Construction de la fonction valeur avec condition aux bords temporelle
def phi_omega(x, t, N_omega):
    return (1 - t) * N_omega(x, t) + t * g(x)

# Générateur avec interpolation temporelle entre bruit latent et sortie du réseau
def G_theta(z, t, N_theta):
    return (1 - t) * z + t * N_theta(z, t)

# Pénalité de collision : moyenne des inverses des distances carrées entre agents
def f_collision(x_batch):
    diff = x_batch.unsqueeze(1) - x_batch.unsqueeze(0)
    dist_sq = torch.sum(diff ** 2, dim=-1)
    mask = ~torch.eye(dist_sq.size(0), dtype=torch.bool, device=dist_sq.device)
    epsilon = 1e-8
    dist_sq_no_diag = dist_sq.masked_select(mask).view(dist_sq.size(0), -1)
    loss_matrix = 1.0 / (dist_sq_no_diag + epsilon)
    return 0 if loss_matrix.mean() > 1 else loss_matrix.mean()

# Pénalité d'obstacle : pénalise les points trop proches des obstacles fixes
def f_obstacle(x, obstacles):
    cost = 0
    batch_size = x.size(0)
    for obstacle in obstacles:
        obstacle_tensor = torch.tensor(obstacle, device=x.device, dtype=x.dtype) if not i
        for i in range(batch_size):
            Q = torch.norm(x[i] - obstacle_tensor)
            if Q < 0.2:
                cost += 1.0 / (max(Q - 0.1, 0) + 1e-8)
    return cost / batch_size

# Définition d'une grille régulière d'obstacles dans l'espace 3D
obstacles = [[(i + 5)/10, 0.5, (j - 5)/10] for i in range(6) for j in range(10)]

# Génération d'une formation polygonale régulière (formation cible)
def generate_polygon_torch(n, w, device=torch.device("cpu")):
    return torch.tensor([[w*m.cos(2*k*m.pi/n), w*m.sin(2*k*m.pi/n), 0] for k in range(n)]
```

```python

# Construction d'une densité cible à partir de Gaussiennes centrées sur les sommets du po
def genere_densite_initiale_torch(n, w, variance, device=torch.device("cpu")):
    points = generate_polygon_torch(n, w, device=device)
    def rho(xyz):
        diff = xyz.unsqueeze(1) - points.unsqueeze(0)
        dist2 = (diff ** 2).sum(dim=-1)
        gaussians = torch.exp(-dist2 / (2 * variance))
        norm_const = torch.sqrt(torch.tensor(2 * torch.pi * variance, device=device))
        return gaussians.sum(dim=1) / (n * norm_const)
    return rho

# Estimateur de densité Gaussien à noyau exponentiel pour des points 3D
class GaussianTorch:
    def __init__(self, data, bandwidth):
        self.data = data
        self.bandwidth = bandwidth
        self.length = data.size(0)

    def profile(self, t):
        return torch.exp(-t)

    def density(self, xyz):
        diff = xyz.unsqueeze(1) - self.data.unsqueeze(0)
        dist2 = (diff ** 2).sum(dim=-1)
        kernel_vals = self.profile(dist2 / (self.bandwidth ** 2))
        return kernel_vals.sum(dim=1) / (torch.sqrt(torch.tensor(2 * torch.pi, device=xyz

# Calcul différentiable de la distance L1 entre deux densités sur une grille 3D
def distance_L1_torch(p_func, q_func, n_grid=10, a=-1.0, b=3.0, device=torch.device("cpu"
    coords = torch.linspace(a, b, n_grid, device=device)
    dx = (b - a) / n_grid
    grid = torch.stack(torch.meshgrid(coords, coords, coords, indexing='ij'), dim=-1)
    flat_grid = grid.view(-1, 3)
    return torch.sum(torch.abs(p_func(flat_grid) - q_func(flat_grid))) * dx ** 3

# Fonction f_formation : pénalise la différence entre la densité générée et la densité ci
density_real = genere_densite_initiale_torch(3, 0.2, 0.05)
def f_formation(x, device=torch.device("cpu")):
    x_centered = x - x.mean(dim=0, keepdim=True)
    sigma = torch.sqrt(torch.tensor(0.05))
    def density_estimated(pts):
        diff = pts.unsqueeze(1) - x_centered.unsqueeze(0)
        dist2 = (diff ** 2).sum(dim=-1)
        gaussians = torch.exp(-dist2 / (2 * sigma**2))
        norm_const = torch.sqrt(torch.tensor(2 * torch.pi, device=device)) * sigma
```

```
164             return gaussians.sum(dim=1) / (x_centered.shape[0] * norm_const)
165         d = distance_L1_torch(density_real, density_estimated, n_grid=40, device=device)
166         print(d)
167         return d
168     ########################################################################
169     # Bloc 4 : Fonctions de coût pour  (NOmega) et le générateur (NTheta)
170     ########################################################################
171
172     # Échantillonnage de points selon une densité initiale polygonale (avec bruit) par reject
173     def sample_from_density(n, sigma, w, n_samples, a=-1, b=3):
174         polygon = generate_polygon_torch(n, w)
175         sommet = torch.randint(low=0, high=n, size=(n_samples,))
176         samples = polygon[sommet]
177         samples += torch.rand(n_samples, 3, device=device) * sigma - sigma/2
178         return samples
179
180     # Fonction de coût pour  : inclut le résidu HJB, la régularisation, et la pénalité de col
181     def compute_loss_phi(N_omega, N_theta, batch_size, T, sigma, lambda_reg, n):
182         variance = sigma ** 2
183         w = 0.2
184         z = sample_from_density(n, sigma, w, batch_size)
185         t = torch.rand(batch_size, 1, requires_grad=True, device=device) * T
186         x_list = [G_theta(z[i:i+1], t[i:i+1], N_theta)[0] for i in range(batch_size)]
187         x = torch.stack(x_list)
188         x.requires_grad_()
189
190         phi_val = phi_omega(x, t, N_omega)
191         grad_phi_x, grad_phi_t = torch.autograd.grad(
192             phi_val, (x, t),
193             grad_outputs=torch.ones_like(phi_val),
194             create_graph=True
195         )
196
197         # Approximation du Laplacien de  par somme des dérivées secondes
198         laplacian = 0
199         for i in range(3):
200             second_deriv = torch.autograd.grad(
201                 grad_phi_x[:, i], x,
202                 grad_outputs=torch.ones_like(grad_phi_x[:, i]),
203                 create_graph=True
204             )[0][:, i]
205             laplacian += second_deriv
206
207         H_phi = torch.norm(grad_phi_x, dim=-1, keepdim=True)
208         loss_phi_terms = phi_omega(x, torch.zeros_like(t), N_omega) + grad_phi_t + (sigma**2
209         loss_phi_mean = loss_phi_terms.mean()
```

```
210
211     # Résidu HJB (régularisation)
212     HJB_residual = torch.zeros(batch_size, device=device)
213     for i in range(batch_size):
214         HJB_residual[i] = torch.norm(
215             grad_phi_t[i] + (sigma**2 / 2) * laplacian[i] + H_phi[i]
216         )
217     loss_HJB = lambda_reg * HJB_residual.mean()
218
219     return loss_phi_mean + loss_HJB + f_collision(x)
220
221 # Fonction de coût pour le générateur : suit le champ de valeur, respecte la formation et
222 def compute_loss_G(N_omega, N_theta, batch_size, T, sigma, n):
223     variance = sigma ** 2
224     w = 0.2
225     z = sample_from_density(n, sigma, w, batch_size)
226     t = torch.rand(batch_size, 1, requires_grad=True, device=device) * T
227     x_list = [G_theta(z[i:i+1], t[i:i+1], N_theta)[0] for i in range(batch_size)]
228     x = torch.stack(x_list)
229     x.requires_grad_()
230
231     phi_val = phi_omega(x, t, N_omega)
232     phi_val.requires_grad_()
233     grad_phi_x, grad_phi_t = torch.autograd.grad(
234         phi_val, (x, t),
235         grad_outputs=torch.ones_like(phi_val),
236         create_graph=True
237     )
238
239     laplacian = 0
240     for i in range(3):
241         second_deriv = torch.autograd.grad(
242             grad_phi_x[:, i], x,
243             grad_outputs=torch.ones_like(grad_phi_x[:, i]),
244             create_graph=True
245         )[0][:, i]
246         laplacian += second_deriv
247
248     H_phi = torch.norm(grad_phi_x, dim=-1, keepdim=True)
249     loss_G_terms = grad_phi_t + (sigma**2 / 2) * laplacian + H_phi
250
251     x_final = G_theta(z, torch.ones_like(t), N_theta)
252     formation_loss = f_formation(x_final)
253     target_loss = torch.norm(x_final.mean(dim=0) - x_target.to(device))
254
255     return loss_G_terms.mean() + 100 * target_loss + 500 * formation_loss + f_obstacle(x,
```

```
256
257 ########################################################################
258 # Bloc 5 : Fonction de test - Affichage des trajectoires de n drones
259 ########################################################################
260
261 # Simule et affiche les trajectoires générées pour une formation polygonale de drones
262 def test_triangle_trajectories(N_theta, w, n, total_time=10.0, num_steps=100):
263     z_triangle = torch.tensor(generate_polygon_torch(n, w))  # vecteurs latents fixés pou
264     trajectories = []  # liste des trajectoires de chaque drone
265     times = torch.linspace(0, total_time, num_steps, device=device)
266
267     for i in range(n):
268         traj = []
269         for t_phys in times:
270             t_norm = t_phys / total_time
271             t_tensor = torch.tensor([[t_norm]], device=device)
272             z = z_triangle[i:i+1]
273             pos = G_theta(z, t_tensor, N_theta)
274             traj.append(pos[0])
275         traj = torch.stack(traj)
276         trajectories.append(traj.cpu().detach().numpy())
277
278     # Affichage des trajectoires 3D
279     fig = plt.figure(figsize=(8, 6))
280     ax = fig.add_subplot(111, projection="3d")
281     distance_matrix = np.zeros((n, n))
282     for i in range(n):
283         traj = trajectories[i]
284         ax.plot(traj[:, 0], traj[:, 1], traj[:, 2], marker='o', label=f'Drone {i+1}', mar
285         print("Position finale du drone", i, ":", traj[-1])
286         for j in range(n):
287             distance_matrix[i, j] = np.linalg.norm(trajectories[i][-1] - trajectories[j][
288     print("Matrice des distances finales entre drones :")
289     print(distance_matrix)
290
291     # Affichage des obstacles
292     for i in obstacles:
293         ax.plot([i[0]], [i[1]], [i[2]], 'ko', markersize=20)
294
295     ax.set_title(f"Trajectoires de {n} drones sur {total_time} secondes")
296     ax.set_xlabel("X")
297     ax.set_ylabel("Y")
298     ax.set_zlabel("Z")
299     ax.set_zlim(-0.5, 0.5)
300     ax.legend()
301     plt.show()
```

```
302
303
304 ########################################################################
305 # Bloc 6 : Boucle d'entraînement principale et test
306 ########################################################################
307
308 def main():
309     # Hyperparamètres
310     batch_size = 24
311     T = 1.0                          # horizon temporel (normalisé)
312     sigma = np.sqrt(0.05)           # coefficient de diffusion
313     epochs = 2000                   # nombre d'itérations d'entraînement
314     lambda_reg = 1.0                # poids de régularisation HJB
315     n = 3                           # nombre de drones
316     w = 0.2                         # rayon du polygone cible
317
318     learning_rate_phi = 4e-4
319     learning_rate_gen = 1e-4
320
321     # Instanciation des réseaux et envoi sur le device
322     N_omega = NOmega().to(device)
323     N_theta = NTheta().to(device)
324
325     optimizer_phi = optim.Adam(N_omega.parameters(), lr=learning_rate_phi,
326                                betas=(0.5, 0.9), weight_decay=1e-4)
327     optimizer_theta = optim.Adam(N_theta.parameters(), lr=learning_rate_gen,
328                                  betas=(0.5, 0.9), weight_decay=1e-4)
329
330     # Boucle d'entraînement principale
331     for epoch in tqdm(range(epochs)):
332         optimizer_phi.zero_grad()
333         loss_phi_val = compute_loss_phi(N_omega, N_theta, batch_size, T, sigma, lambda_re
334         loss_phi_val.backward()
335         optimizer_phi.step()
336
337         optimizer_theta.zero_grad()
338         loss_gen_val = compute_loss_G(N_omega, N_theta, batch_size, T, sigma, n)
339         loss_gen_val.backward()
340         optimizer_theta.step()
341
342         if epoch % 10 == 0:
343             print(f"Epoch {epoch} | Perte  : {loss_phi_val.item():.4f} | Perte G : {loss_
344
345     # Test final : affichage des trajectoires simulées
346     test_triangle_trajectories(N_theta, w, n, total_time=10.0, num_steps=100)
347
```

```
348
349 # Point d'entrée du programme
350 if __name__ == "__main__":
351     main()
352
```

RÉFÉRENCES

[1] uofang Wang, Wang Yao, Xiao Zhang and Ziming Li *A Mean-Field Game Control for Large-Scale Swarm Formation Flight in Dense Environments*, Addison-Wesley, 1994.

[2] ialong Zhang, Jianguo Yan, Pu Zhang, Xiangjie Kong *Collision Avoidance in Fixed-Wing UAV Formation Flight Based on a Consensus Control Algorithm*

[3] NMING WU, YIDONG SUN , JIANYU HUANG, CHAO ZHANG, AND ZHIHENG LI *Multi UAV Cluster Control Method Based on Virtual Core in Improved Artificial Potential Field*

[4] Alex Tong Lin, Samy Wu Fung, Wuchen Li, Levon Nurbekyan Stanley J. Osher *Alternating the Population and Control Neural Networks to Solve High-Dimensional Stochastic Mean-Field Games*, 2023

[5] Karthik Elamvazhuthi, *Benamou-Brenier Formulation of Optimal Transport for Nonlinear Control Systems on* $\mathbb{R}^d$, Department of Mechanical Engineering University of California, Riverside CA 92521, USA, 2024

[6] Bernt Øksendal, *Stochastic Differential Equations : An Introduction with Applications*, 8^e édition, Springer, 2013, pp 1-54

[7] André Schlichting, *From Fokker-Planck to SDES and back via the martingale problem*, notes, Université de Bonn, 2015, pp 1-5