2025-2026

NOM : Goudal
PRENOM : Brieuc

☒Stage Scientifique et Technique (S7)
☐Stage de fin d'étude (Stage Ingénieur - S10)

Rapport de stage

ENTREPRISE : USTH (University of Science and Technology of Hanoï)

DATES : 15 Aout 2025 – 12 Décembre 2025

SUJET DE STAGE : Détection de la distraction du conducteur

TUTEUR EN ENTREPRISE : Dr Pham Xuan Tung

CADRE RÉSERVÉ POUR LES STAGES DE FIN D'ÉTUDES (STAGE INGÉNIEUR)

OPTION	☐ DISA ☐ SI-CSS ☐ SI-LD ☐ SE-ER ☐ SE-LC ☐ GETE ☐ ING AFF ☐ DAIA ☐ E-SANTE ☐SC ☐ I4
CONFIDENTIALITÉ	mon rapport est confidentiel niveau ☐ 1 (moy) ☐ 2 (élevé) *Un mail a été envoyé par votre tuteur à Mme Baronnet (Dijon), Mme Nourai (Paris) ou Mme Omont (Angers) au moins un mois avant votre soutenance, sinon l'ESEO retient le niveau 0 cf. fiche d'encadrement.*
DOMAINE ENTREPRISE	☐ Auto. ☐ Aéron. ☐ Banques-fi-assur. ☐ Biomédical-santé ☐ Energie ☐ NTIC ☐ Télécoms ☐ Autres (précisez) :
AUTRES POINTS	☐ Stage à dominante **management** ou ☐ Stage à dominante **recherche** ☐ E5e effectuée sous Contrat Pro ☐ Mon tuteur sera présent à ma soutenance

Engagement de non-plagiat :

Je soussigné, Brieuc Goudal, étudiant à l'ESEO, atteste avoir pris connaissance du contenu du Règlement intérieur de l'École et de l'engagement de « non-plagiat ». Je déclare m'y conformer dans le cadre de la rédaction de ce document. Je déclare sur l'honneur que le contenu du présent mémoire est original et reflète mon travail personnel. J'atteste que les citations sont correctement signalées par des guillemets et que les sources de tous les emprunts ponctuels à d'autres auteur(e)s, textuels ou non textuels, sont indiquées. Le non-respect de cet engagement m'exposerait à des sanctions dont j'ai bien pris connaissance.

Fait à Hanoï le 09/12/2025.

Fiche de synthèse du stage

Sujet : Atterrissage automatique de drones sur cibles mobiles – détection ArUco – contrôle PX4/MAVSDK – simulateur Gazebo – C++.

Entreprise : USTH (Université des Sciences et des Technologies de Hanoï), Hanoï, Vietnam.

Dates de stage : 15 août 2025 – 12 décembre 2025.

Personnes principales : Tuteur USTH : Pham Xuan Tung, Professeur référent ESEO : M. Feuilloy Matthieu - Elèves du laboratoire de recherche.

Abstract

Autonomous landing of drones on moving platforms remains a major challenge for intelligent aerial systems operating in dynamic environments. Ensuring a precise and stable trajectory becomes difficult when the drone must track a moving target with variable visual feedback. This work investigates how vision-based adaptive control can enable the drone to adjust in real time and achieve reliable landings. In a simulation environment realistically modeling drone–platform interactions, the system demonstrates convergence to the target and controlled descent, even when visual tracking is temporarily limited. These results indicate that adaptive vision-based architecture provides a robust foundation for autonomous landings in dynamic conditions.

Résumé

L’atterrissage autonome de drones sur des plateformes mobiles constitue un défi majeur pour les systèmes aériens intelligents évoluant dans des environnements dynamiques. Garantir une trajectoire précise et stable devient complexe lorsque le drone doit suivre une cible en mouvement avec un retour visuel variable. Ce travail explore comment un contrôle adaptatif fondé sur la vision peut permettre au drone de s’ajuster en temps réel et de réaliser un atterrissage fiable. Dans un environnement de simulation reproduisant fidèlement les interactions drone–plateforme, le système montre sa capacité à converger vers la cible et à maintenir une descente maîtrisée, même lorsque le suivi visuel est momentanément limité. Ces résultats démontrent que les architectures adaptatives basées sur la vision offrent un socle robuste pour des atterrissages autonomes en conditions dynamiques.

1- Introduction

L'atterrissage autonome de drones sur une plateforme mobile constitue aujourd'hui un défi majeur dans le domaine de la robotique aérienne. L'intérêt croissant pour des systèmes aériens intelligents capables d'opérer dans des environnements dynamiques a conduit la communauté scientifique à explorer des approches combinant vision par ordinateur, estimation de pose et stratégies de contrôle adaptées. Parmi ces approches, l'utilisation de marqueurs fiduciaires tels qu'ArUco ou AprilTag s'impose comme une solution efficace, permettant d'estimer précisément la position relative de la cible à l'aide d'une simple caméra embarquée.

Les travaux existants ont confirmé la faisabilité de l'atterrissage autonome, que la plateforme soit immobile ou en mouvement. Les premières contributions reposaient principalement sur des cibles statiques et sur l'usage de contrôleurs classiques tels que des PID ou des méthodes de servoing visuel. Des recherches plus récentes ont étendu ces méthodes aux plateformes mobiles en introduisant des contrôleurs adaptatifs ou des mécanismes de suivi basés exclusivement sur l'analyse visuelle, améliorant ainsi la stabilité et la précision du guidage.

L'essor d'outils de simulation tels que Gazebo, et de frameworks de contrôle comme PX4 et MAVSDK, a par ailleurs facilité le développement de systèmes complexes. Ces environnements offrent la possibilité de tester l'intégralité de la chaîne — détection visuelle, suivi d'une cible mobile, phase finale de descente — dans des conditions réalistes, sûres et reproductibles.

Cependant, malgré ces avancées, un enjeu majeur demeure.
La problématique centrale de ce travail est la suivante :
comment maintenir une trajectoire d'atterrissage stable et précise lorsque le drone doit suivre une plateforme mobile uniquement par rapport à un retour visuel ?
Le drone doit en effet analyser en continu les informations issues de la caméra, extraire la position du marqueur, estimer la pose relative et ajuster sa trajectoire en temps réel, alors même que ce retour visuel évolue au fil du vol (changement d'altitude, distance, orientation du drone ou de la plateforme). Concevoir un système capable de rester aligné sur la cible mobile et de corriger sa trajectoire jusque dans les derniers instants de l'atterrissage constitue donc un défi essentiel.

Dans ce contexte, l'objectif de ce travail est de proposer une solution complète reposant sur une détection visuelle fiable, un contrôleur PID adaptatif en fonction de l'altitude et une plateforme mobile simulée de manière cohérente.

Mes principales contributions peuvent être résumées comme suit :
• Concevoir un cadre de simulation complet pour l'atterrissage autonome en utilisant PX4, MAVSDK et Gazebo.
• Mettre en œuvre une chaîne de détection visuelle basée sur les marqueurs ArUco, optimisée pour le suivi d'une cible mobile.
• Développer un contrôleur PID multi-axes adaptatif ajustant ses gains en fonction de

l’altitude afin de renforcer la stabilité durant la phase finale d’atterrissage.
• Modéliser une plateforme mobile personnalisée équipée d’un marqueur ArUco et contrôlée via un système DiffDrive.
• Réaliser des expérimentations en simulation pour évaluer les performances du système proposé, identifier ses limites et dégager des perspectives de recherche futures.

2- Etape 1 : Prise en main et mise en place de l’environnement

2.1- Démarche méthodologique du projet

Le projet s'articule autour d'une approche itérative en plusieurs étapes afin de répondre à la problématique suivante : comment maintenir une trajectoire d’atterrissage stable et précise lorsque le drone doit suivre une plateforme mobile uniquement par rapport à un retour visuel ?

Etape 1 : Familiarisation et mise en place - Prise en main de Gazebo pour la simulation 3D, PX4 pour le contrôle de vol, MAVSDK pour la communication drone, et OpenCV pour la détection ArUco.

Etape 2 : Atterrissage sur cible fixe (système D) - Développement d'un système de base avec décollage, détection de marqueur immobile (ArUco), et atterrissage par corrections directes.

Etape 3 : Passage à la cible mobile avec régulation PID - Introduction d'un correcteur PID adaptatif pour suivre une cible en mouvement avec stabilité et précision. Implémentation d'une stratégie de descente en trois phases avec mémorisation des vitesses pour l'atterrissage final automatique, sans nécessiter de détection continue.

Etape 4 : Évaluation des limites et faisabilité - Tests des performances avec différents scénarios : vitesses variables du véhicule, trajectoires avec virages serrés, mouvements circulaires continus. Identification des seuils de vitesse maximale supportés et des limites technologiques du système.

Cette méthodologie permet une progression logique du simple vers le complexe, avec validation à chaque étape avant de passer à la suivante. Chacune apporte son lot d'enseignements qui alimentent les développements ultérieurs.

2.2- Installation et mise en place de l’environnement

J'ai effectué le développement sur un PC Windows, ce qui m'a conduit à utiliser WSL2 (Windows Subsystem for Linux) pour installer Ubuntu 22.04. Cette solution s'est imposée car Gazebo Garden et PX4 requièrent impérativement un environnement Linux pour fonctionner correctement. WSL2 permet de bénéficier d'une intégration native tout en conservant l'environnement Windows.

Gazebo - Simulateur robotique 3D utilisé pour modéliser le drone, l'environnement de vol, et créer le véhicule mobile portant le marqueur ArUco. Simule la physique réaliste et les capteurs embarqués.

PX4 Autopilot - Logiciel de contrôle de vol open-source gérant l'autopilote et exposant l'interface MAVLink pour la communication. Le modèle gz_x500_gimbal avec nacelle orientable a été utilisé pour ce projet.

MAVSDK - Bibliothèque C++ intégrée pour communiquer avec PX4 via MAVLink. Elle fournit les commandes de vol (décollage, déplacements, atterrissage) et permet de récupérer la télémétrie en temps réel (position, vitesse, altitude).

OpenCV avec module ArUco - Bibliothèque de traitement d'images utilisée pour détecter les marqueurs ArUco et calculer leur position relative par rapport à la caméra du drone.

Une fois l'ensemble installé et configuré, l'environnement était opérationnel pour débuter le développement du système d'atterrissage.

2.3 – Prise en main du drone dans Gazebo

La phase de familiarisation a consisté à comprendre les commandes de base pour contrôler le drone via MAVSDK. Le plugin « Action » fournit des fonctions de haut niveau qui encapsulent toute la complexité de la communication MAVLink avec PX4.

Les commandes essentielles testées

Armement des moteurs - La fonction : **action_->arm()** prépare le drone au vol en activant les moteurs. Elle vérifie au préalable que toutes les conditions de sécurité sont remplies (GPS fix, batterie suffisante, capteurs opérationnels) :

```
if (action_->arm() != Action::Result::Success) {
    std::cerr << "Erreur armement" << std::endl;
    return false;
}
```

Sans armement préalable, aucune autre commande de vol n'est possible.

Décollage automatique - La fonction : **action_->takeoff()** fait monter le drone verticalement jusqu'à une altitude de sécurité (environ 2.5 mètres) puis le stabilise en vol stationnaire. PX4 gère automatiquement la montée, la stabilisation, et le passage en mode Hold.

Déplacement vers une position - La fonction **action_->goto_location()** permet de commander le drone vers des coordonnées GPS précises. Elle a été utilisée pour faire monter le drone de 2.5m vers l'altitude souhaitée.

Atterrissage automatique - La fonction **action_->land()** fait descendre le drone verticalement à vitesse contrôlée jusqu'au sol, puis désarme automatiquement les moteurs.

Ces commandes ont été testées dans un programme simple exécutant la séquence complète : armement, décollage, montée à 10m, maintien en position, puis atterrissage. L'observation dans Gazebo confirmait l'exécution correcte de chaque étape. Cette maîtrise des commandes de base était indispensable avant d'intégrer la détection ArUco et le contrôle PID pour l'atterrissage autonome.

2.4 – Test et orientation de la caméra

Une fois le décollage maîtrisé, il était nécessaire de tester la caméra embarquée et de l'orienter correctement pour détecter les marqueurs ArUco au sol. Le drone x500_gimbal dispose d'une nacelle motorisée permettant d'ajuster l'orientation de la caméra indépendamment des mouvements du drone.

Connexion et récupération des images

La caméra simulée dans Gazebo publie ses images via le système de transport gz-transport.

Orientation de la nacelle vers le bas

Par défaut, la caméra pointe vers l'avant. Pour détecter des marqueurs au sol, il faut l'orienter verticalement vers le bas à -90° en pitch. Cela s'effectue via le plugin Gimbal de MAVSDK après le décollage :

```
// Configurer gimbal
gimbal_->take_control(0, Gimbal::ControlMode::Primary);
std::this_thread::sleep_for(std::chrono::seconds(2));
gimbal_->set_angles(0, 0.0f, -90.0f, 0.0f, Gimbal::GimbalMode::YawLock, Gimbal::SendMode::Once);

std::cout << "Décollage terminé, caméra orientée vers le bas" << std::endl;
```

La fonction take_control() donne le contrôle exclusif de la nacelle au programme. set_angles() positionne ensuite la caméra : roll = 0° (pas d'inclinaison latérale), pitch = -90° (verticale vers le bas), yaw = 0° (même cap que le drone). L'affichage des images via **cv::imshow()** a confirmé que la caméra pointait correctement vers le sol.

3- Etape 2 : Atterrissage sur cible fixe (système P)

3.1- Objectifs de cette étape

Cette étape constitue la première approche opérationnelle du système d'atterrissage autonome. L'objectif est de valider la chaîne complète de détection et d'atterrissage dans un contexte simplifié : un marqueur ArUco immobile posé au sol. Cette étape permet de se concentrer sur les mécanismes fondamentaux (détection visuelle, corrections de position, descente contrôlée) avant d'introduire la complexité supplémentaire d'une cible en mouvement.

Les critères de succès sont clairs : le drone doit décoller, localiser le marqueur ArUco, se centrer au-dessus de lui avec une précision suffisante, puis effectuer une descente verticale contrôlée jusqu'au sol. Le système doit fonctionner de manière robuste avec un taux de réussite de 100%, lors de tests dans des conditions nominales.

3.2- Architecture logicielle

Le programme **my_drone_simple.cpp** est structuré autour de quatre composants principaux, chacun ayant un rôle spécifique :

GazeboCamera gère la récupération des images depuis la caméra embarquée du drone via Gazebo Transport. Elle convertit les flux d'images du simulateur en matrices OpenCV exploitables.

ArucoDetector analyse les images pour détecter le marqueur ArUco au sol et calcule les écarts en pixels entre le centre du marqueur et le centre de l'image.

SimpleController applique le contrôle proportionnel : il convertit les erreurs en pixels en commandes de vitesse (m/s) et vérifie si le drone est suffisamment centré pour descendre.

DroneController assure toutes les communications avec PX4 via MAVSDK : connexion, décollage, envoi des vitesses, lecture de la télémétrie et atterrissage.

Cette séparation modulaire facilite la maintenance et permet de remplacer facilement le contrôleur pour des versions plus évoluées, comme nous le verrons dans l’étape 3.

3.3- Connexion et initialisation du drone

Le programme commence par initialiser la connexion avec l'autopilote PX4 du drone simulé dans Gazebo.

```
// Connexion MAVSDK
Mavsdk mavsdk(Mavsdk::Configuration(ComponentType::GroundStation));
ConnectionResult connection_result = mavsdk.add_any_connection("udp://:14540");

if (connection_result != ConnectionResult::Success) {
    std::cerr << "Erreur connexion" << std::endl;
    return 1;
}

auto system = mavsdk.first_autopilot(3.0);
if (!system) {
    std::cerr << "Aucun drone détecté" << std::endl;
    return 1;
}
```

MAVSDK est configuré comme station sol et établit une connexion UDP sur le port 14540, port standard de PX4 pour les communications MAVLink.
La méthode first_autopilot(3.0) attend jusqu'à 3 secondes qu'un autopilote soit détecté sur le réseau. Ce délai permet à PX4 SITL de démarrer complètement dans Gazebo et de commencer à émettre. Si aucun système n'est trouvé, le programme s'arrête pour éviter d'envoyer des commandes sans destinataire.

Une fois le système détecté, le DroneController() initialise les plugins nécessaires :

```
DroneController(System& system) {
    action_ = std::make_unique<Action>(system);
    telemetry_ = std::make_unique<Telemetry>(system);
    offboard_ = std::make_unique<Offboard>(system);
    gimbal_ = std::make_unique<Gimbal>(system);
}
```

Ces quatre plugins permettent respectivement de commander les actions (décollage/atterrissage), lire la télémétrie (position/altitude), contrôler les vitesses en mode offboard, et orienter le gimbal de la caméra.

3.4- Décollage et activation du mode offboard

Une fois la connexion établie, le drone doit décoller et passer en mode de contrôle externe. La méthode takeoff() encapsule plusieurs vérifications et actions :

```
bool takeoff() {
    std::cout << "=== DÉCOLLAGE ===" << std::endl;

    while (!telemetry_->health_all_ok()) {
        std::cout << "Attente drone prêt..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    if (action_->arm() != Action::Result::Success) {
        std::cerr << "Erreur armement" << std::endl;
        return false;
    }

    if (action_->takeoff() != Action::Result::Success) {
        std::cerr << "Erreur décollage" << std::endl;
        return false;
    }

    std::this_thread::sleep_for(std::chrono::seconds(8));

    // Monter à 10m
    std::cout << "Montée à altitude de travail..." << std::endl;
    Action::Result goto_result = action_->goto_location(
        telemetry_->position().latitude_deg,
        telemetry_->position().longitude_deg,
        10.0f,
        0.0f
    );
```

La première étape consiste à vérifier que tous les systèmes du drone sont opérationnels via health_all_ok(). Cette vérification s'assure que le GPS, l'IMU, le magnétomètre et les autres capteurs critiques fonctionnent correctement. Une fois cette validation effectuée, les moteurs sont armés, ce qui les prépare à tourner.

La commande takeoff() de MAVSDK déclenche alors la procédure de décollage automatique gérée par PX4. Le drone monte verticalement jusqu'à environ 2-3 mètres, altitude par défaut du mode takeoff. Après une pause de 8 secondes permettant la stabilisation complète, une commande goto_location est envoyée pour amener le drone à 10 mètres d'altitude, hauteur de travail optimale pour commencer la recherche du marqueur. Cette altitude offre un bon compromis entre champ de vision large et qualité de détection.

La méthode start_offboard() active ensuite le mode permettant d'envoyer des commandes de vitesse personnalisées :

```
bool start_offboard() {
    Offboard::VelocityNedYaw stay{};
    offboard_->set_velocity_ned(stay);

    Offboard::Result offboard_result = offboard_->start();
    if (offboard_result != Offboard::Result::Success) {
        std::cerr << "Erreur mode offboard" << std::endl;
        return false;
    }

    std::cout << "Mode offboard activé" << std::endl;
    return true;
}
```

Le mode offboard de PX4 exige qu'un flux de commandes soit déjà initialisé avant son activation. C'est pourquoi une première commande de vitesse nulle est envoyée via set_velocity_ned(). Une fois ce flux établi, l'activation du mode transfère le contrôle total du pilote automatique vers notre programme externe. Dès lors, toutes les commandes de vitesse envoyées seront exécutées directement par le drone.

3.5- Boucle de détection et tracking

Une fois le drone stabilisé à 10 mètres d'altitude, le gimbal portant la caméra est orienté à -90° pour pointer verticalement vers le sol. Cette orientation permet de détecter le marqueur ArUco au sol pendant toute la phase de descente. Le système entre alors dans une boucle continue qui s'exécute à environ 10 Hz (toutes les 100 ms).

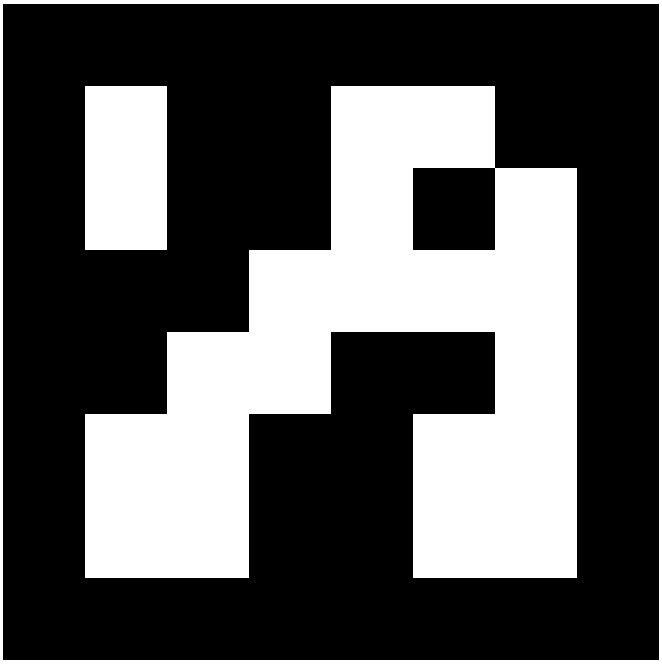
À chaque itération de la boucle, le système récupère une image depuis la caméra embarquée et lit l'altitude actuelle du drone. Ces deux informations constituent les entrées du système de contrôle.

```
while (true) {
    auto iteration_start = std::chrono::steady_clock::now();

    // Récupérer image
    if (!camera->get_frame(frame)) {
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
        continue;
    }

    float v_north = 0.0f, v_east = 0.0f, v_down = 0.0f;
    float altitude = drone.get_altitude();
```

La classe GazeboCamera établit une connexion avec le topic Gazebo Transport correspondant à la caméra embarquée. Ce topic diffuse en continu les images capturées au format 640×480 pixels en RGB. La méthode get_frame() récupère la dernière image disponible de manière thread-safe grâce à un mutex interne qui protège l'accès concurrent entre le callback de réception et la boucle principale.

Si aucune image n'est disponible (par exemple au tout début lorsque le flux n'a pas encore démarré), la boucle attend 50 ms avant de réessayer. Une fois l'image obtenue, l'altitude relative du drone est lue depuis la télémétrie PX4. Cette altitude, exprimée en mètres par rapport au point de décollage, est cruciale pour déterminer la stratégie de descente à appliquer.

Détection ArUco

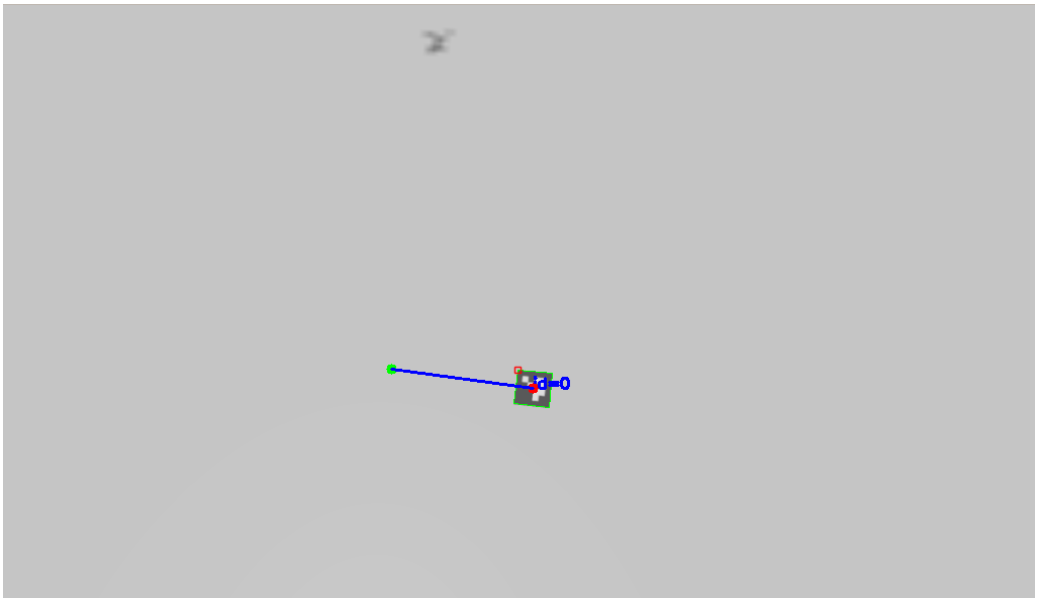


Une fois l'image acquise, le système recherche le marqueur ArUco, calcule les erreurs de positionnement et applique les corrections proportionnelles.

```
cv::Point2f marker_center;
int marker_id;
bool marker_detected = detector.detect(frame, marker_center, marker_id);

if (marker_detected) {
    // Calculer erreur en pixels
    cv::Point2f img_center(frame.cols/2.0f, frame.rows/2.0f);
    float error_x = marker_center.x - img_center.x;
    float error_y = marker_center.y - img_center.y;

    // Correction proportionnelle
    controller.compute_velocities(error_x, error_y, v_east, v_north);
```

Le détecteur ArUco utilise la bibliothèque OpenCV pour rechercher des marqueurs du dictionnaire 4x4_50 dans l'image. Ce dictionnaire contient 50 marqueurs distincts identifiables de manière unique. L'algorithme détecte les contours noirs des marqueurs, valide leur forme carrée, puis décode le motif binaire interne pour identifier l'ID.

Une fois le marqueur détecté, le centre géométrique est calculé en moyennant les quatre coins. Ce point central est ensuite comparé au centre de l'image (320, 240) pour obtenir les erreurs de positionnement. Une erreur error_x positive signifie que le marqueur est à droite du centre, une erreur error_y positive signifie qu'il est en bas.



Le contrôle proportionnel : Ces erreurs en pixels sont ensuite converties en commandes de vitesse par le contrôleur proportionnel.
La méthode compute_velocities() applique un gain kp = 0.002 :

```
// Calcul correction proportionnelle simple
void compute_velocities(float error_x, float error_y, float& v_east, float& v_north) {
    // Correction proportionnelle pure : vitesse = Kp * erreur
    v_east = kp_ * error_x;
    v_north = -kp_ * error_y;  // Inversé car axe Y image inversé

    // Limiter les vitesses maximales
    v_east = std::clamp(v_east, -0.5f, 0.5f);
    v_north = std::clamp(v_north, -0.5f, 0.5f);
}
```

Le principe du contrôle proportionnel est simple : la vitesse de correction est directement proportionnelle à l'erreur. Plus le drone est éloigné du centre, plus il se déplace rapidement. Plus il s'approche, plus il ralentit naturellement. Le coefficient kp = 0.002 détermine la réactivité : une valeur trop élevée causerait des oscillations, une valeur trop faible ralentirait la convergence.

Par exemple, si le marqueur est détecté à 200 pixels à droite du centre, la vitesse Est sera de 0.002 × 200 = 0.4 m/s. Si l'erreur descend à 50 pixels, la vitesse devient 0.002 × 50 = 0.1 m/s. Les vitesses sont limitées à ±0.5 m/s pour garantir la sécurité.

L'inversion de l'axe Y (-kp * error_y) est nécessaire car dans une image, Y augmente vers le bas, alors que dans le référentiel NED, la direction Nord correspond à un déplacement vers le haut dans l'image. Cette conversion permet d'aligner correctement les mouvements du drone avec la position visuelle du marqueur.

3.6- Stratégie de descente et atterrissage

Le système implémente une stratégie en deux phases distinctes, séparées par un seuil d'altitude à 1 mètre. Cette approche combine la précision du tracking en altitude et la rapidité de l'atterrissage final.

Phase haute

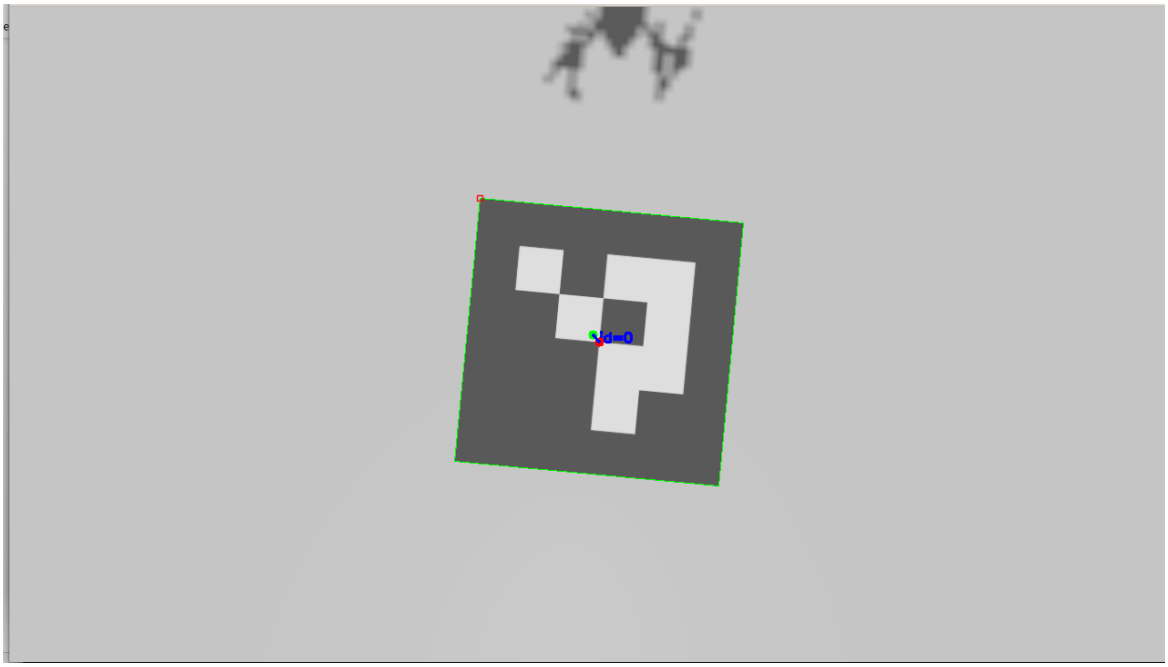
Au-dessus de 1 mètre, le drone applique en continu les corrections proportionnelles calculées pour maintenir l'alignement horizontal avec le marqueur. La descente verticale est gérée de manière conditionnelle : le système vérifie d'abord si le drone est suffisamment centré en calculant la distance euclidienne totale des erreurs et en la comparant au seuil de 50 pixels.

Un compteur de stabilité impose une condition temporelle stricte : le drone doit rester centré pendant 10 frames consécutives (environ 1 seconde à 10 Hz) avant d'autoriser la descente. Ce mécanisme évite les descentes prématurées dues à des détections instables ou du bruit de mesure. Une fois ce critère satisfait, la descente débute à 0.3 m/s, vitesse prudente permettant de maintenir le tracking actif sans perdre le marqueur de vue.

Si le drone dérive à tout moment et que l'erreur dépasse le seuil, la descente est immédiatement suspendue et le compteur réinitialisé. Le système force ainsi le drone à se recentrer avant de reprendre la descente, garantissant une approche progressive et contrôlée.

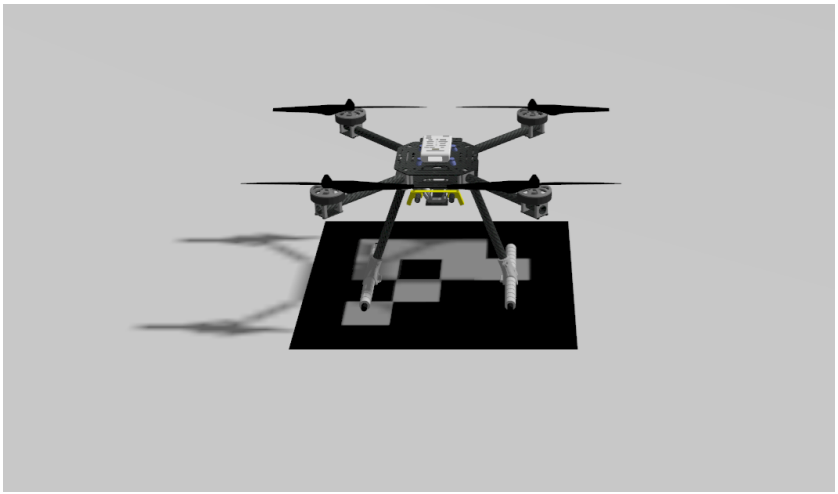
Phase basse

En-dessous de 1 mètre, le tracking ArUco est complètement désactivé et le drone effectue une descente verticale directe. Cette décision repose sur deux constats : d'une part, le drone a normalement effectué un centrage suffisant lors de la phase haute pour être correctement aligné. D'autre part, à très basse altitude, le marqueur occupe une grande partie du champ de vision, ce qui perturbe la détection OpenCV et donc peut bloquer le drone vers 0.6 m.



Le drone maintient sa position horizontale avec des vitesses Nord et Est nulles, et descend verticalement à 0.3 m/s. Cette approche "descente en ligne droite" suppose que l'alignement effectué de 10 mètres à 1 mètre a permis de positionner correctement le drone. Les tests confirment cette hypothèse avec une précision finale de ±10 cm.

Lorsque l'altitude atteint 30 cm, la commande d'atterrissage formel est envoyée à PX4, qui gère alors la coupure progressive des moteurs et le désarmement automatique du drone.



3.7- Limites et pistes d’améliorations sur cible fixe

En simulation, l'algorithme se pose de manière fiable sur une cible immobile. Le centrage final est correct et la descente est progressive. Les limites observées :

- **Contrôle proportionnel pur** : L'absence de terme intégral empêche la correction des erreurs persistantes, et l'absence de dérivée peut provoquer des oscillations lors de corrections importantes ;
- **Éclairage et contraste** : La détection ArUco nécessite des conditions d'éclairage stables et un bon contraste entre le marqueur et son support pour fonctionner de manière fiable.

Piste d'amélioration : Passer d'un correcteur P (proportionnel) à un PID complet pour corriger les erreurs statiques, anticiper les mouvements et accélérer la convergence sans osciller. Cette évolution sera mise en œuvre dans l'étape suivante avec le suivi de cible mobile.

4- Etape 3 : Atterrrissage sur cible mobileavec contrôle PID

4.1- Contexte et motivation du passage au PID

L'étape précédente a permis de valider la faisabilité d'un atterrissage autonome sur marqueur fixe en utilisant un contrôleur proportionnel simple. Cette première approche, bien que fonctionnelle dans un contexte statique, révèle rapidement ses limitations face à une cible en mouvement. Le contrôleur P, en ne tenant compte que de l'erreur instantanée, ne peut ni anticiper les déplacements futurs de la plateforme, ni corriger efficacement les décalages permanents induits par le temps de latence entre détection et exécution.

Dans le cadre d'un atterrissage sur plateforme mobile, trois défis majeurs se posent. Premièrement, le suivi dynamique nécessite que le drone compense en permanence le mouvement de la cible, créant des erreurs évolutives dans le temps. Deuxièmement, un dilemme apparaît entre réactivité et stabilité : un gain proportionnel élevé permet des corrections rapides mais génère des oscillations, tandis qu'un gain faible assure la stabilité au prix d'un retard inacceptable. Troisièmement, la phase finale de descente sous 1 mètre d'altitude pose problème, car le marqueur occupe alors presque tout le champ de vision, rendant la détection ArUco impossible. De plus, la nature mobile de la cible rend inapplicable la stratégie d'atterrissage de l'étape 2.

Ces constats ont orienté mes travaux vers l'adoption d'un contrôleur PID (Proportionnel-Intégral-Dérivé) et la conception d'une plateforme mobile simulée permettant de reproduire fidèlement des conditions d'atterrissage dynamiques.

4.2- Architecture et justification du contrôleur PID

4.2.1- Limites du contrôle proportionnel pur

Le contrôleur proportionnel utilisé dans l'étape 2 établit une relation linéaire directe entre l'erreur de positionnement et la commande de vitesse appliquée :

$$v = Kp \times e(t)$$

Bien que cette approche soit simple à implémenter et à régler, elle présente trois limitations fondamentales pour le suivi de cible mobile. D'abord, l'absence de terme intégral empêche la correction des erreurs statiques : lorsque la cible se déplace à vitesse constante, il subsiste toujours un décalage permanent entre le drone et la plateforme. Ensuite, le système ne dispose d'aucun mécanisme d'anticipation : la

commande réagit uniquement à l'erreur présente, sans considération de sa vitesse d'évolution. Enfin, le réglage du gain Kp impose un compromis rigide entre rapidité de convergence et stabilité du système, sans possibilité d'adaptation dynamique.

4.2.2- Principe du contrôleur PID

Pour surmonter ces limitations, j'ai développé un contrôleur PID qui combine trois actions de contrôle complémentaires :

$$v(t) = Kp \times e(t) + Ki \times \int_0^t e(\tau)d\tau + Kd \times \frac{de(t)}{dt}$$

Le terme proportionnel conserve son rôle de correction immédiate, générant une réponse directement proportionnelle à l'écart mesuré. Le terme intégral accumule les erreurs passées dans le temps, permettant d'éliminer les décalages permanents caractéristiques du suivi de cible mobile. Enfin, le terme dérivé estime la vitesse de variation de l'erreur, apportant ainsi une capacité d'anticipation : il atténue la commande lorsque l'erreur diminue rapidement (évitant les dépassements), et la renforce lorsqu'elle augmente (accélérant la réaction).

4.2.3- Implémentation du PID en C++

Le contrôleur PID a été implémenté sous forme de classe C++ réutilisable, permettant d'instancier un contrôleur indépendant pour chaque axe de déplacement :

```
class PIDController {
public:
    PIDController(float kp, float ki, float kd, float max_integral = 2.0f)
        : kp_(kp), ki_(ki), kd_(kd), max_integral_(max_integral),
          error_sum_(0), last_error_(0) {}

    float compute(float error, float dt) {
```

La méthode compute() calcule la commande de vitesse en combinant les trois termes. Afin de prévenir l'accumulation excessive de l'erreur intégrale lors de fortes perturbations ou de limitations de commande, un mécanisme d'anti-windup a été implémenté. L'intégrale est saturée à ±2.0, empêchant ainsi les dépassements importants et garantissant un retour stable lors de la convergence vers la consigne. Cette limitation permet au système de rester réactif même après des phases de tracking

difficile, en évitant que le terme intégral ne domine excessivement la commande une fois la cible recentrée.

```
error_sum_ += error * dt;
error_sum_ = std::clamp(error_sum_, -max_integral_, max_integral_);
float i_term = ki_ * error_sum_;
```

4.2.4- Réglage des gains adaptatifs

```
// Kp adaptatif entre 0.5 et 2.0 selon l'erreur
float kp_min = 0.5f ;
float kp_max = 2.0f ;
float ki = 0.5f;  // Fixe
float kd = 0.3f;  // Fixe
float kp = kp_min + (kp_max - kp_min) * error_factor;

pid_x_->set_gains(kp, ki, kd);
pid_y_->set_gains(kp, ki, kd);
```

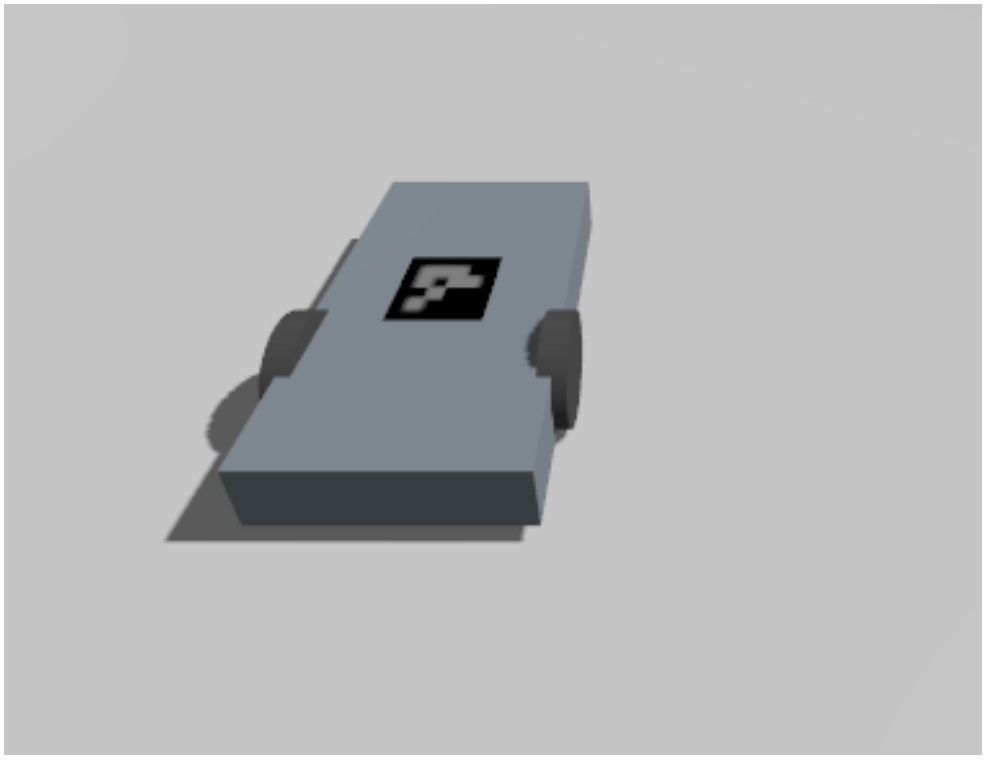
Les bornes du gain proportionnel adaptatif ont été déterminées de manière empirique par essais successifs en simulation, en s'appuyant sur les gains fixes préalablement validés. Le gain minimal a été fixé à $K_p^{\min} = 0.5$, valeur suffisamment basse pour éviter les oscillations lorsque la cible est parfaitement centrée dans l'image. Le gain maximal $K_p^{\max} = 2.0$ correspond précisément au gain proportionnel fixe optimal identifié lors des premiers tests, garantissant ainsi une réactivité maximale lorsque l'erreur de positionnement est importante. Les gains intégral et dérivé restent fixes à $K_i = 0.5$ et $K_d = 0.3$ pour conserver la stabilité éprouvée du système.

Cette stratégie d'adaptation linéaire basée sur la magnitude de l'erreur offre un comportement progressif : le gain varie continûment entre les deux bornes selon la formule $K_p = 0.5 + 1.5 \cdot \text{error\_factor}$, où error_factor $\in [0,1]$ représente la magnitude normalisée de l'erreur de positionnement dans le plan image.

4.3- Plateforme mobile pour atterrissage dynamique

Pour évaluer la capacité du système PID à suivre une cible en mouvement, nous avons développé un robot mobile équipé d'un marqueur ArUco de dimensions 50×50 cm. Ce robot, nommé aruco_robot, constitue la plateforme d'atterrissage dynamique sur laquelle le drone doit se poser tout en compensant les déplacements.

La propulsion du robot repose sur un système différentiel (DiffDrive) simulé dans Gazebo. Ce mécanisme, couramment utilisé en robotique mobile, permet de contrôler indépendamment deux roues motrices pour générer des mouvements de translation et de rotation. Le robot reçoit ses commandes de vitesse via le topic Gazebo Transport /cmd_vel, au format Twist standard, permettant de spécifier la vitesse linéaire (axe x) et la vitesse angulaire (axe z).



4.4- Stratégie de descente en trois phases

4.4.1- Phase d'approche active (altitude > 2m)

Durant la phase d'approche, le système opère en boucle fermée avec détection continue du marqueur ArUco et contrôle PID actif. Le drone descend progressivement vers la cible mobile tout en corrigeant en permanence sa position horizontale pour maintenir le marqueur au centre de l'image. La vitesse de descente est ajustée selon l'altitude : 0.2 m/s au-dessus de 3 mètres pour une approche rapide, puis 0.15 m/s en dessous pour affiner la convergence. Les commandes de vitesse horizontale sont calculées par les contrôleurs PID des axes X et Y, avec une limitation à ±3.0 m/s pour éviter les manœuvres trop brusques.

```
else if (aruco_detected && altitude >= 2.0f) {
    cv::Point2f img_center(frame_size.width/2.0f, frame_size.height/2.0f);
    cv::Point2f error = aruco_center - img_center;
    float norm_error_x = error.x / (frame_size.width/2.0f);
    float norm_error_y = error.y / (frame_size.height/2.0f);
```

Un élément critique de cette phase est la mémorisation continue des vitesses du drone dans le référentiel NED (North-East-Down). À chaque itération, les composantes velocity_ned.north_m_s et velocity_ned.east_m_s de la télémétrie sont enregistrées. Ces vitesses reflètent le comportement du système en régime établi : lorsque l'erreur de positionnement est faible, le drone se déplace approximativement à la même vitesse que la cible, permettant de capturer implicitement la dynamique de déplacement de la plateforme mobile.

```
  float max_speed = 3.0f;
  v_east = std::clamp(pid_x_->compute(norm_error_x, dt), -max_speed, max_speed);
  v_north = std::clamp(-pid_y_->compute(norm_error_y, dt), -max_speed, max_speed);
  v_down = (altitude > 3.0f) ? 0.2f : 0.15f;

  auto velocity_ned = telemetry_->velocity_ned();
  final_v_north_ = velocity_ned.north_m_s;  // Vitesse réelle du drone
  final_v_east_ = velocity_ned.east_m_s;
```

4.4.2- Phase de stabilisation (altitude = 2m)

Lorsque l'altitude atteint 2 mètres, le système entre en phase de stabilisation pendant une durée fixe de 2 secondes. Durant cette période, toute commande de descente est annulée, et les vitesses horizontales sont figées aux dernières valeurs mémorisées. Cette phase de hover stabilisé répond à plusieurs objectifs critiques. Premièrement, elle permet de consolider l'alignement avec la trajectoire de la cible avant la descente finale. Deuxièmement, elle garantit que les vitesses mémorisées correspondent effectivement à un régime de suivi stable, et non à une correction transitoire.

Troisièmement, elle anticipe la perte inévitable de détection ArUco à très basse altitude : en dessous de 2 mètres, l'angle de vue devient trop rasant et le marqueur n'est plus détecté.

```
else if (altitude < 2.0f && altitude > 0.1f && !final_velocities_locked_) {
    if (!stabilization_started_) {
        // Démarrage stabilisation - vitesses déjà mémorisées en phase approche
        stabilization_started_ = true;
        stabilization_start_time_ = current_time;

        std::cout << "\n=== STABILISATION 4s < 2M - VITESSES BLOQUÉES ===" << std::endl;
        std::cout << "Vitesses figées (mémorisées): N=" << final_v_north_ << " E=" << final_v_east_
    }

    auto elapsed = std::chrono::duration<float>(current_time - stabilization_start_time_).count();

    if (elapsed < 2.0f) {
        // Pendant 2 secondes : TOUT BLOQUÉ (vitesses + descente)
        v_north = final_v_north_;
        v_east = final_v_east_;
        v_down = 0.0f;  // AUCUNE descente
```

Le compteur de stabilisation démarre automatiquement dès que l'altitude passe sous le seuil de 2 mètres, déclenchant l'affichage des vitesses figées dans les logs de télémétrie. Cette phase critique assure la transition entre le contrôle réactif en haute altitude (avec feedback visuel continu) et le contrôle prédictif en basse altitude (en boucle ouverte, sans détection ArUco). En figeant les vitesses à 2 mètres plutôt qu'à 0.5 mètre, le système maximise les chances de capturer un état de suivi fiable avant la zone aveugle finale.

4.4.3- Phase de descente finale (altitude < 2m)

Après les 4 secondes de stabilisation, le système déverrouille la descente tout en maintenant les vitesses horizontales constantes. Le drone poursuit sa trajectoire en boucle ouverte avec v_down = 0.15 m/s, sans aucune correction basée sur la vision. Cette stratégie repose sur l'hypothèse que la cible maintient une trajectoire stable : en conservant les vitesses mémorisées, le drone suit la même trajectoire que la plateforme mobile, préservant ainsi l'alignement relatif malgré l'absence de feedback visuel.

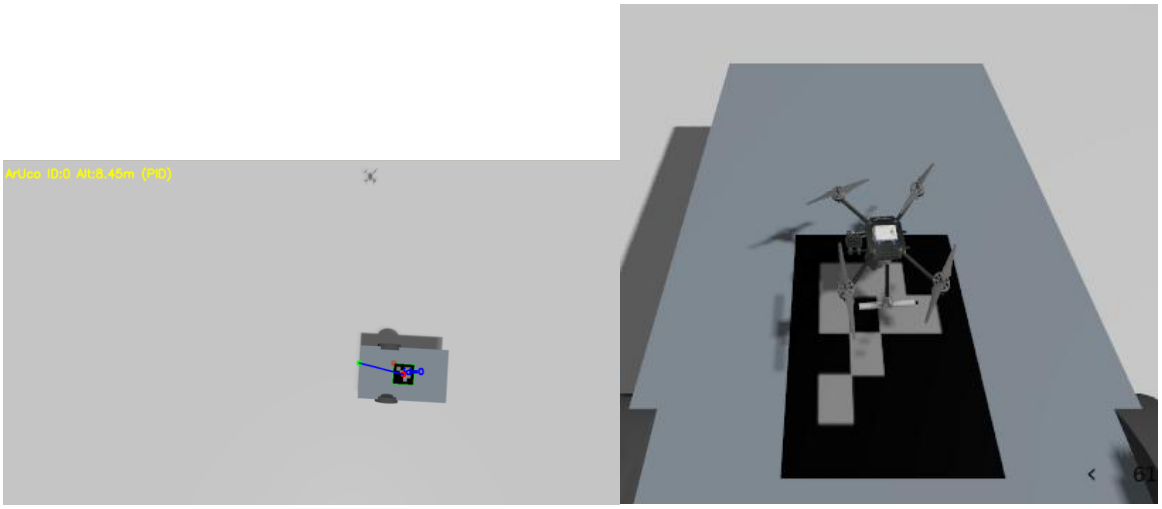
```
if (final_velocities_locked_ && altitude > 0.1f) {
    v_north = final_v_north_;
    v_east = final_v_east_;
    v_down = 0.15f;  // Reprise descente après stabilisation
```

4.6- Validation expérimentale

Les expérimentations menées avec des vélocités modestes (0.5 à 1.0 m/s) sur parcours rectiligne confirment l'efficacité de la méthode développée. Le système atteint un régime de poursuite satisfaisant après une phase transitoire d'approximativement trente secondes suivant l'identification initiale de la cible. L'erreur latérale, dépassant initialement un mètre lors de la première détection, se réduit progressivement jusqu'à environ quinze centimètres pendant la descente avec asservissement actif, avant de se stabiliser autour de cinq centimètres durant la séquence de maintien altitude constant à deux mètres.

L'imprécision mesurée au contact final varie typiquement entre 10 et quinze centimètres relativement au point central du marqueur, résultat amplement compatible avec la surface utile de cinquante centimètres disponibles. L'examen des traces de vol confirme le respect de la décomposition tripartite prévue : descente asservie d'une durée de vingt-cinq à trente secondes depuis l'altitude initiale jusqu'au palier de deux mètres, maintien horizontal strict pendant deux secondes précises, et finalement trajectoire balistique contrôlée durant six à sept secondes jusqu'au posé.

La modulation automatique des coefficients proportionnels répond aux prévisions du modèle théorique. En situation d'écart important, le coefficient s'établit spontanément aux environs de sa valeur plafond (approximativement 2.0) favorisant une convergence énergique, puis régresse vers 0.6-0.8 au fur et à mesure que l'alignement s'améliore, éliminant les risques d'oscillations parasites.



4.7- Conclusion de l’étape

Le développement d'un système d'atterrissage sur cible mobile a permis de valider l'utilisation d'un contrôleur PID avec gains adaptatifs pour compenser les déplacements horizontaux d'une plateforme en mouvement. La stratégie en trois phases – approche active avec asservissement visuel, stabilisation à deux mètres, puis descente finale en boucle ouverte – répond efficacement aux contraintes imposées par la perte de détection ArUco à basse altitude.

Les résultats expérimentaux confirment la pertinence de cette architecture pour des vitesses de cible comprises entre 0.5 et 1.0 m/s, avec des précisions d'atterrissage de l'ordre de vingt à trente centimètres. Ces performances sont compatibles avec des applications réelles telles que l'atterrissage sur véhicule lent ou plateforme mobile autonome. Le mécanisme de mémorisation des vitesses offre une solution pragmatique au problème de la zone aveugle sous deux mètres.

Les limites observées lors des tests feront l'objet d'une analyse détaillée dans la partie suivante.

5- Etape 4 : Limites du système

5.1- Limitation en vitesse de la cible

Les essais menés à des vitesses supérieures à 2.0 m/s révèlent une dégradation significative des performances du système. Le système maintient un suivi satisfaisant jusqu'à 2.0 m/s, mais au-delà, vers 2.5 m/s, des oscillations horizontales importantes apparaissent et s'amplifient progressivement, malgré l'adaptation automatique des gains proportionnels. Ces oscillations résultent du retard cumulé dans la chaîne de traitement : acquisition image, détection ArUco, calcul PID et réponse dynamique du drone, totalisant environ 100 à 150 millisecondes.

À 2.5 m/s, la cible parcourt 25 à 35 centimètres durant ce cycle, créant un décalage systématique que le contrôleur tente de compenser par des corrections de plus en plus énergiques. Le drone "surcompense" continuellement son retard, alternant entre avance et retard par rapport à la position réelle. Ces oscillations deviennent suffisamment importantes pour que le marqueur sorte périodiquement du champ de vision, compromettant l'atterrissage. Cette limitation du contrôle réactif pur suggère qu'au-delà de 2.0 m/s, une approche prédictive anticipant la position future de la cible serait nécessaire pour étendre les capacités du système vers des vitesses plus élevées.

5.2- Gestion des trajectoires courbes

Le système peut théoriquement gérer des trajectoires courbes, mais sous des contraintes temporelles très strictes liées au temps de convergence du contrôleur PID. Lorsque la cible effectue un changement de direction, le drone nécessite environ 10 à 15 secondes pour reconverger vers un régime de suivi stable sur la nouvelle trajectoire, compte tenu de la dynamique du système et du temps de réaction du contrôleur.

Cette durée de convergence impose une contrainte critique sur l'altitude à laquelle les virages peuvent être effectués. Si la cible change de cap avant d'atteindre 5 à 6 mètres d'altitude, le drone dispose d'une fenêtre temporelle suffisante pour s'adapter à la nouvelle direction et atteindre un suivi stabilisé avant la phase de mémorisation à 2 mètres. En revanche, si le virage s'effectue trop tardivement, par exemple à 3 ou 4 mètres, le drone n'a pas le temps de reconverger complètement. Il atteindra alors l'altitude de 2 mètres avec une erreur de suivi encore significative, mémorisant des vitesses qui ne correspondent pas à un régime établi.

5.3- Hypothèse de trajectoire rectiligne en phase finale

La limitation critique du système apparaît durant la phase de descente finale en dessous de 2 mètres. À cette altitude, la détection ArUco devient impossible en raison de l'angle de vue rasant, forçant le système à basculer en boucle ouverte avec les vitesses horizontales mémorisées à 2 mètres.

Cette phase aveugle de 6 à 7 secondes repose entièrement sur l'hypothèse que la cible maintient une trajectoire rectiligne à vitesse constante. Si cette hypothèse est respectée, le drone conserve son alignement relatif et l'atterrissage réussit. En revanche, tout changement de cap ou de vitesse sous 2 mètres provoque une dérive progressive, le drone n'ayant aucun moyen de détecter ou corriger cet écart. Les essais avec virages en basse altitude confirment cette vulnérabilité : le décalage entre drone et cible croît linéairement, compromettant la précision voire provoquant un échec complet d'atterrissage.

6- Impact écologique

La dimension environnementale du développement informatique lui-même reste pratiquement invisible : quelques sessions de programmation, des compilations et des simulations qui consomment collectivement une quantité d'énergie dérisoire comparée aux enjeux réels qui se situent dans l'utilisation concrète du système sur le terrain. L'amélioration de la capacité d'atterrissage autonome sur plateforme mobile transforme potentiellement plusieurs aspects de l'exploitation des drones avec des répercussions écologiques qu'il convient d'examiner avec lucidité. Commençons par les bénéfices plausibles : la possibilité pour un drone de rejoindre directement un véhicule en déplacement bouleverse la logistique traditionnelle qui impose des points de récupération fixes nécessitant des trajets dédiés, et cette simplification se traduit concrètement par une réduction des kilomètres parcourus par les équipes au sol, ce qui dans le contexte d'opérations régulières comme la surveillance d'infrastructures linéaires ou l'inspection agricole peut représenter plusieurs centaines de kilomètres économisés annuellement avec les émissions carbone correspondantes. L'efficacité énergétique des missions elles-mêmes s'améliore également car les manœuvres d'approche finale consomment habituellement une fraction non négligeable de la batterie, entre huit et douze pour cent selon les conditions, et un système capable de converger rapidement sans tâtonnements ni corrections répétées peut réduire cette part de quinze à quarante pour cent, ce qui bien que modeste par vol isolé devient substantiel sur une saison d'exploitation intensive. La robustesse technique joue un rôle souvent sous-estimé dans l'équation environnementale : chaque atterrissage approximatif ou raté provoque des contraintes mécaniques qui accélèrent l'usure des trains d'atterrissage, fragilisent les structures composites et dégradent les cellules de batterie par des décharges brutales, tandis qu'un système fiable prolonge naturellement la durée d'exploitation du matériel et repousse d'autant la nécessité de fabriquer des pièces de rechange avec toute l'empreinte extractive et manufacturière associée. Ces perspectives encourageantes doivent néanmoins être tempérées par une analyse des risques systémiques que l'innovation peut induire : le piège de l'effet rebond guette dès qu'une technologie rend plus accessible et économique un service auparavant contraint, car la tentation devient forte de multiplier les interventions au point que l'augmentation du volume d'activité absorbe complètement les gains d'efficience unitaire, créant paradoxalement une dégradation du bilan global. Les traitements embarqués nécessaires à la vision artificielle et au contrôle temps réel imposent une charge calculatoire permanente qui ampute l'autonomie de vol de quelques pour cent si l'optimisation logicielle n'est pas poussée à l'extrême, et cette taxe énergétique constante peut dans certains cas dépasser les économies ponctuelles réalisées durant la séquence d'atterrissage. L'intensification potentielle des vols génère également des nuisances acoustiques et visuelles susceptibles de perturber les écosystèmes locaux, particulièrement dans les environnements naturels où le développement de ces

capacités pourrait justifier de nouvelles applications auparavant jugées impraticables. La sophistication croissante des plateformes, intégrant caméras performantes, unités de calcul dédiées et capteurs variés, complique significativement les opérations de maintenance et rend problématique la valorisation en fin de vie faute de filières industrielles capables de traiter efficacement ces assemblages hétérogènes. L'équation environnementale finale dépendra donc essentiellement des modalités d'adoption et des garde-fous opérationnels mis en place : exploitation maximale de la simulation durant les phases de développement et d'optimisation pour limiter drastiquement les essais réels, instrumentation systématique des plateformes pour collecter des données fines sur les profils de consommation et identifier les marges de progrès, mise en œuvre de politiques de maintenance préventive des batteries basées sur le monitoring précis des cycles et des régimes thermiques pour maximiser leur longévité, alimentation des stations de recharge avec des sources énergétiques décarbonées, et organisation de circuits de réemploi ou reconditionnement des modules électroniques pour éviter la mise au rebut prématurée. Des métriques opérationnelles simples permettent de piloter objectivement cette démarche : consommation énergétique moyenne par mission et décomposition par phase de vol, proportion de l'énergie dédiée à l'approche finale, taux de réussite des procédures d'atterrissage, nombre de déplacements terrestres évités, fréquence des incidents matériels et volume de pièces remplacées. En définitive, cette innovation technologique ne porte pas intrinsèquement de valeur écologique positive ou négative : son impact réel sera déterminé par la discipline opérationnelle, la rigueur de la mesure et la capacité collective à optimiser le rapport entre services rendus et ressources mobilisées, faute de quoi les bénéfices théoriques resteront lettre morte.

7- Conclusion

L'objectif de ce stage était de concevoir et valider un système d'atterrissage autonome sur plateforme mobile, guidé uniquement par vision embarquée. Cette problématique s'inscrit dans les défis actuels de la robotique aérienne et de la logistique automatisée, secteurs en forte croissance tant dans la recherche académique que dans l'industrie. Les résultats obtenus valident expérimentalement la pertinence d'une approche combinant détection visuelle d'ArUco, contrôle PID adaptatif et stratégie de descente multi-phases pour des scénarios de mobilité modérée (jusqu'à 2 m/s, soit environ 7 km/h, sur trajectoire rectiligne).

L'architecture logicielle développée (intégration PX4-MAVSDK-OpenCV dans un environnement Gazebo/Docker) constitue une contribution méthodologique reproductible et extensible. La stratégie de descente progressive en trois phases (approche, stabilisation, contact final) permet d'optimiser le compromis vitesse-précision et assure une fiabilité d'atterrissage satisfaisante avec une précision de 10-15 cm. Le système de gains adaptatifs, ajustant dynamiquement la réactivité du contrôleur en fonction de l'erreur mesurée, démontre qu'une solution légère basée sur des composants open-source standards peut atteindre des performances opérationnelles pour un premier niveau de validation.

Les limitations observées – notamment l'impossibilité d'atterrir sur trajectoires circulaires et les oscillations au-delà de 2 m/s – mettent en lumière les frontières de l'approche réactive pure. Le contrôleur PID, conçu pour minimiser l'erreur instantanée, ne peut compenser l'absence de modèle prédictif de la trajectoire cible. Sur virages effectués à basse altitude (< 5-6 m), le temps de convergence du système (10-15 secondes) se révèle insuffisant pour permettre un centrage stable avant la phase finale de descente. Cette observation rejoint les conclusions de nombreux travaux récents préconisant l'intégration de techniques d'estimation et de contrôle avancé (MPC, estimateurs de mouvement, filtres de Kalman) pour traiter des scénarios dynamiques complexes. L'évaluation en simulation, bien que rigoureuse et complète, ne capture pas l'ensemble des contraintes réelles (délais matériels, perturbations atmosphériques, dégradation du signal visuel).

Plusieurs axes d'amélioration structurent naturellement la suite de ces travaux : enrichissement du contrôle par composantes prédictives ou feedforward, fusion multi-capteurs (IMU, flow optique) pour robustifier l'estimation d'état, exploration d'approches de détection sans marqueur artificiel (deep learning, SLAM visuel), et optimisation de l'implémentation embarquée pour contraintes temps-réel strictes.

Au-delà des aspects techniques, ce stage a été une expérience humaine enrichissante. La rencontre organisée par l’Ambassade de France au Vietnam et les échanges

quotidiens avec les étudiants vietnamiens, m'ont permis d'acquérir une vision claire de la coopération scientifique franco-vietnamienne. Les séances de football avec l'équipe de l'USTH et mon tuteur ont ancré mon intégration dans la vie du campus et renforcé la dimension interculturelle de ce stage.