ESEO

École Supérieure d'Électronique de l'Ouest

Année 2025-2026

Rapport de Stage

Stage Scientifique et Technique (S7)

Atterrissage Automatique de Drones sur Cibles Mobiles

Détection ArUco – Contrôle PID/PX4 – Simulateur Gazebo

Étudiant : Brieuc Goudal

Période : 15 août 2025 – 12 décembre 2025

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

Sujet : Détection de la distraction du conducteur / Atterrissage automatique de drones sur cibles mobiles

Tuteur entreprise : Dr Pham Xuan Tung

Tuteur ESEO : M. Feuilloy Matthieu

Hanoï, Décembre 2025

ENGAGEMENT DE NON-PLAGIAT

Je soussigné, **Brieuc Goudal**, étudiant à l'ESEO (École Supérieure d'Électronique de l'Ouest), déclare sur l'honneur que le présent rapport de stage est le fruit d'un travail personnel et que toutes les sources d'information utilisées ont été citées de manière appropriée.

Je certifie que ce rapport n'a fait l'objet d'aucun plagiat et que les données, résultats et analyses présentés sont authentiques et n'ont pas été falsifiés.

Je reconnais avoir pris connaissance de la charte contre le plagiat de l'ESEO et m'engage à respecter les règles de propriété intellectuelle et d'éthique académique.

Je suis conscient que tout acte de plagiat pourra entraîner des sanctions pouvant aller jusqu'à l'annulation de ma validation de stage.

Fait à Hanoï, le 09 décembre 2025

Signature :

Brieuc Goudal

FICHE DE SYNTHÈSE

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

Abstract (English)

This internship focuses on developing an autonomous landing system for quadrotors on moving platforms using visual markers. The project implements a complete simulation framework integrating PX4 autopilot, MAVSDK communication library, and Gazebo simulator. The system uses ArUco marker detection through an onboard camera to track the target platform in real-time. A multi-phase adaptive PID controller ensures stable trajectory tracking even when visual feedback becomes unreliable at low altitudes. The implementation includes optimized marker detection algorithms, altitude-dependent control gains, and a trajectory memory mechanism for blind landing phases. Experimental validation in simulation demonstrates successful landings on platforms moving at speeds up to 2.0 m/s with positioning accuracy of 10-15 cm. The system's main limitations are oscillations at higher speeds and requirements for straight-line trajectories during final approach. This work provides a foundation for autonomous drone operations in dynamic environments with applications in emergency response, delivery systems, and mobile platform recovery.

Résumé (Français)

Ce stage porte sur le développement d'un système d'atterrissage autonome de drones sur des plateformes mobiles utilisant des marqueurs visuels. Le projet met en œuvre un cadre de simulation complet intégrant l'autopilote PX4, la bibliothèque de communication MAVSDK et le simulateur Gazebo. Le système utilise la détection de marqueurs ArUco via une caméra embarquée pour suivre la plateforme cible en temps réel. Un contrôleur PID adaptatif multi-phases assure un suivi de trajectoire stable même lorsque le retour visuel devient peu fiable à basse altitude. L'implémentation comprend des algorithmes de détection de marqueurs optimisés, des gains de contrôle dépendants de l'altitude et un mécanisme de mémorisation de trajectoire pour les phases d'atterrissage à l'aveugle. La validation expérimentale en simulation démontre des atterrissages réussis sur des plateformes se déplaçant à des vitesses allant jusqu'à 2,0 m/s avec une précision de positionnement de 10-15 cm. Les principales limitations du système sont les oscillations à vitesses plus élevées et les exigences de trajectoires rectilignes pendant l'approche finale. Ce travail fournit une base pour les opérations autonomes de drones dans des environnements dynamiques avec des applications dans les interventions d'urgence, les systèmes de livraison et la récupération sur plateformes mobiles.

Table des matières

1 Introduction

L'atterrissage autonome de drones sur des cibles mobiles représente un défi majeur dans le domaine de la robotique aérienne. Cette capacité est essentielle pour de nombreuses applications pratiques telles que la livraison autonome, les interventions d'urgence, ou encore la récupération de drones sur des véhicules en mouvement.

1.1 Contexte

Les drones modernes sont capables de voler de manière autonome grâce à des systèmes de positionnement GPS et des autopilotes sophistiqués. Cependant, l'atterrissage précis sur une cible en mouvement nécessite une approche différente basée sur la vision par ordinateur et le contrôle en temps réel. Le retour visuel doit permettre au drone de localiser la cible, d'estimer sa position relative, et d'ajuster sa trajectoire en conséquence.

1.2 Problématique

La problématique principale de ce projet est de maintenir une trajectoire stable avec un retour visuel variable. En effet, à mesure que le drone descend vers la cible, le marqueur visuel occupe une part de plus en plus importante du champ de vision de la caméra, jusqu'à devenir impossible à détecter correctement à très basse altitude. Il faut donc développer une stratégie qui permette de :

— Détecter et suivre la cible à haute altitude

— Compenser les mouvements de la plateforme mobile

— Maintenir la trajectoire lorsque la détection visuelle devient impossible

— Atterrir avec précision malgré l'absence de retour visuel

1.3 Contributions principales

Ce projet apporte les contributions suivantes :

— **Cadre de simulation complet** : Intégration de PX4 Autopilot, MAVSDK, et Gazebo pour créer un environnement de test réaliste

— **Détection ArUco optimisée** : Configuration avancée des paramètres de détection pour maximiser la portée et la robustesse

— **Contrôleur PID adaptatif** : Implémentation d'un contrôleur à gains variables en fonction de l'altitude pour optimiser stabilité et réactivité

— **Plateforme mobile personnalisée** : Création d'un robot mobile avec marqueur ArUco de grande taille dans Gazebo

— **Expérimentations et limites** : Validation expérimentale du système et identification des limitations

1.4 Structure du rapport

Ce rapport est organisé en plusieurs sections correspondant aux différentes étapes du projet. La section 2 présente la prise en main de l'environnement de développement. La section 3 détaille l'implémentation d'un système d'atterrissage sur cible fixe avec contrôle proportionnel simple. La section 4 décrit l'extension vers l'atterrissage sur cible mobile avec un contrôleur PID complet. La section 5 analyse les limites du système développé. La section 6 présente une réflexion sur l'impact écologique du projet. Enfin, la section 7 conclut le rapport et propose des perspectives d'amélioration.

2 Étape 1 – Prise en main et mise en place

La première étape du projet a consisté à installer et prendre en main l'ensemble des outils nécessaires au développement du système d'atterrissage autonome.

2.1 Démarche méthodologique

Le projet a été structuré selon une approche itérative en quatre étapes principales :

1. **Installation et configuration** : Mise en place de l'environnement de développement avec tous les outils nécessaires
2. **Atterrissage sur cible fixe** : Développement d'une première version avec contrôle proportionnel simple
3. **Atterrissage sur cible mobile** : Extension du système avec un contrôleur PID adaptatif
4. **Validation et identification des limites** : Tests expérimentaux et analyse des performances

Cette approche progressive a permis de valider chaque composant du système avant d'augmenter la complexité.

2.2 Installation de l'environnement

L'environnement de développement a nécessité l'installation et la configuration de plusieurs composants logiciels sur un système Ubuntu 22.04 sous WSL2 (Windows Subsystem for Linux) :

2.2.1 WSL2 et Ubuntu

WSL2 (Windows Subsystem for Linux 2) permet d'exécuter un environnement Linux complet sous Windows. Cette solution a été choisie pour sa compatibilité avec l'ensemble des outils utilisés dans le projet.

2.2.2 Gazebo Garden

Gazebo est un simulateur robotique 3D qui permet de tester des algorithmes dans un environnement virtuel réaliste. La version Garden a été installée car elle est compatible avec PX4-Autopilot et offre des performances améliorées. Gazebo simule la physique du drone, les capteurs embarqués (caméra, IMU, GPS), et l'environnement de vol.

2.2.3 PX4-Autopilot

PX4 est un système d'autopilote open-source utilisé dans de nombreux drones professionnels. Il gère le contrôle bas niveau du drone (stabilisation, navigation, modes de vol) et communique via le protocole MAVLink. L'installation de PX4 comprend également l'intégration avec Gazebo pour la simulation.

2.2.4 MAVSDK

MAVSDK est une bibliothèque C++ qui simplifie la communication avec l'autopilote PX4 via le protocole MAVLink. Elle fournit des API de haut niveau pour contrôler le drone (décollage, atterrissage, mode offboard, télémétrie, etc.).

2.2.5 OpenCV

OpenCV (Open Source Computer Vision Library) est une bibliothèque de vision par ordinateur qui fournit des outils pour le traitement d'images. Le module ArUco d'OpenCV permet la détection et l'identification de marqueurs fiduciaires, essentiels pour la localisation de la cible.

2.3 Prise en main du drone dans Gazebo

Une fois l'environnement installé, la première étape a été de se familiariser avec le contrôle du drone simulé dans Gazebo.

2.3.1 Lancement de la simulation

La simulation est lancée avec la commande suivante :

```
cd ~/PX4-Autopilot
make px4_sitl gz_x500_gimbal
```

Cette commande démarre PX4 en mode *Software In The Loop* (SITL) avec un drone X500 équipé d'un gimbal caméra dans Gazebo.

2.3.2 Armement et décollage

L'armement du drone (activation des moteurs) et le décollage sont réalisés via MAVSDK :

```
// Connexion au drone
Mavsdk mavsdk;
mavsdk.add_any_connection("udp://:14540");
auto system = mavsdk.systems().at(0);

// Plugins n cessaires
auto action = Action{system};
auto telemetry = Telemetry{system};

// Attendre que le drone soit pr t
while (!telemetry.health().is_home_position_ok) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

// Armer et d coller
action.arm();
action.takeoff();
```

Listing 1 – Armement et décollage du drone

2.3.3 Déplacement en mode offboard

Le mode offboard permet de contrôler directement les consignes de vitesse ou de position du drone. Voici un exemple de déplacement :

```
auto offboard = Offboard{system};

// Activer le mode offboard
Offboard::VelocityBodyYawspeed velocity{};
velocity.forward_m_s = 0.0f;
velocity.right_m_s = 0.0f;
velocity.down_m_s = 0.0f;
velocity.yawspeed_deg_s = 0.0f;

offboard.set_velocity_body(velocity);
offboard.start();

// D placer le drone
velocity.forward_m_s = 1.0f;  // 1 m/s vers l'avant
offboard.set_velocity_body(velocity);
```

Listing 2 – Contrôle en mode offboard

2.3.4 Atterrissage

L'atterrissage peut être commandé de deux manières :

— Via la fonction `action.land()` pour un atterrissage automatique

— En mode offboard avec une vitesse de descente contrôlée

2.4 Test et orientation de la caméra

La caméra embarquée du drone est essentielle pour la détection du marqueur ArUco. Deux aspects ont été vérifiés :

2.4.1 Connexion via gz-transport

La caméra de Gazebo publie les images sur un topic ROS. La connexion se fait via la bibliothèque gz-transport :

```
#include <gz/transport/Node.hh>
#include <gz/msgs/image.pb.h>

class GazeboCamera {
public:
    GazeboCamera(const std::string& topic) {
        if (!node.Subscribe(topic, &GazeboCamera::on_image, this)
           ) {
            throw std::runtime_error("Erreur connexion camera");
        }
    }

    bool get_frame(cv::Mat& frame) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (latest_frame_.empty()) return false;
        frame = latest_frame_.clone();
        return true;
    }

private:
    void on_image(const gz::msgs::Image& msg) {
        cv::Mat img(msg.height(), msg.width(), CV_8UC3);
        memcpy(img.data, msg.data().c_str(), msg.data().size());
        cv::cvtColor(img, img, cv::COLOR_RGB2BGR);

        std::lock_guard<std::mutex> lock(mutex_);
        latest_frame_ = img;
    }

    gz::transport::Node node;
    cv::Mat latest_frame_;
    std::mutex mutex_;
```

```
};
```

Listing 3 – Connexion à la caméra Gazebo

2.4.2 Orientation du gimbal à -90°

Pour que la caméra pointe verticalement vers le bas (nécessaire pour voir la cible au sol), le gimbal doit être orienté à -90° en pitch :

```
auto gimbal = Gimbal{system};

// Orienter la cam ra vers le bas
Gimbal::ControlMode control_mode = Gimbal::ControlMode::Primary;
gimbal.set_mode(control_mode);

// Pitch -90 degr s (vers le bas)
gimbal.set_pitch_and_yaw(-90.0f, 0.0f);
```

Listing 4 – Configuration du gimbal

Cette configuration permet à la caméra de capturer les images du marqueur ArUco positionné au sol ou sur la plateforme mobile.

3 Étape 2 – Atterrissage sur cible fixe (système P)

La deuxième étape du projet consiste à développer un système d'atterrissage sur une cible fixe en utilisant un contrôle proportionnel simple.

3.1 Objectifs

Les objectifs de cette étape sont :

— Implémenter la détection d'un marqueur ArUco au sol

— Développer un contrôleur proportionnel pour aligner le drone avec la cible

— Réaliser une descente contrôlée jusqu'à l'atterrissage

— Valider l'architecture logicielle de base

3.2 Architecture logicielle

Le système est structuré en quatre composants principaux :

1. **GazeboCamera** : Gère la connexion à la caméra du drone dans Gazebo et fournit les images en temps réel
2. **ArucoDetector** : Détecte les marqueurs ArUco dans les images et calcule leur position dans le cadre image

3. **SimpleController** : Calcule les corrections de vitesse en fonction de l'erreur de position (contrôle proportionnel)
4. **DroneController** : Interface avec MAVSDK pour envoyer les commandes de vol au drone

Cette architecture modulaire permet de tester et améliorer chaque composant indépendamment.

3.3 Connexion et initialisation

Le programme commence par initialiser tous les composants :

```
// Connexion au drone via MAVSDK
Mavsdk mavsdk;
mavsdk.add_any_connection("udp://:14540");
auto system = mavsdk.systems().at(0);

// Initialisation des plugins
auto action = Action{system};
auto telemetry = Telemetry{system};
auto offboard = Offboard{system};
auto gimbal = Gimbal{system};

// Initialisation de la cam ra et du d tecteur
GazeboCamera camera("/camera");
ArucoDetector detector;
```

Listing 5 – Initialisation du système

3.4 Décollage et mode offboard

Après initialisation, le drone décolle et active le mode offboard :

```
// Attendre que le drone soit pr t
while (!telemetry.health().is_home_position_ok) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

// Armer et d coller    10 m tres
action.arm();
action.set_takeoff_altitude(10.0f);
action.takeoff();

// Attendre la fin du d collage
std::this_thread::sleep_for(std::chrono::seconds(10));

// Orienter la cam ra vers le bas
```

```
gimbal.set_pitch_and_yaw(-90.0f, 0.0f);

// Activer le mode offboard
Offboard::VelocityBodyYawspeed velocity{};
offboard.set_velocity_body(velocity);
offboard.start();
```

Listing 6 – Décollage et activation du mode offboard

3.5 Boucle de détection et tracking

La boucle principale du programme détecte le marqueur ArUco et ajuste la position du drone :

```
while (altitude > 0.3) {
    // R cup rer une image de la cam ra
    cv::Mat frame;
    if (!camera.get_frame(frame)) continue;

    // D tecter le marqueur ArUco
    cv::Point2f center;
    int marker_id;
    if (!detector.detect(frame, center, marker_id)) {
        // Pas de d tection, maintenir la position
        velocity.forward_m_s = 0.0f;
        velocity.right_m_s = 0.0f;
        offboard.set_velocity_body(velocity);
        continue;
    }

    // Calculer l'erreur de position
    float frame_center_x = frame.cols / 2.0f;
    float frame_center_y = frame.rows / 2.0f;
    float error_x = center.x - frame_center_x;
    float error_y = center.y - frame_center_y;

    // Contr le proportionnel
    float Kp = 0.001f;  // Gain proportionnel
    velocity.right_m_s = Kp * error_x;
    velocity.forward_m_s = Kp * error_y;

    // Limitation de vitesse
    float max_speed = 1.0f;
    velocity.right_m_s = std::clamp(velocity.right_m_s,
                                    -max_speed, max_speed);
```

```
32      velocity.forward_m_s = std::clamp(velocity.forward_m_s,
33                                        -max_speed, max_speed);
34
35      // Envoyer la commande
36      offboard.set_velocity_body(velocity);
37
38      // R cup rer l'altitude actuelle
39      altitude = telemetry.position().relative_altitude_m;
40  }
```

Listing 7 – Boucle de détection et contrôle

Le contrôle proportionnel applique une correction de vitesse proportionnelle à l'erreur de position. La formule est :

$$v = K_p \times e(t) \tag{1}$$

où v est la vitesse de correction, K_p est le gain proportionnel, et $e(t)$ est l'erreur de position à l'instant t.

3.6 Stratégie de descente

La descente se fait en deux phases :

3.6.1 Phase haute (altitude ¿ 1m)

À haute altitude, le drone descend lentement tout en maintenant l'alignement avec la cible :

```
1  if (altitude > 1.0f) {
2      velocity.down_m_s = 0.3f;  // Descente    0.3 m/s
3  } else {
4      velocity.down_m_s = 0.2f;  // Descente plus lente
5  }
```

Listing 8 – Descente à haute altitude

3.6.2 Phase basse (altitude ¡ 1m)

À basse altitude, la vitesse de descente est réduite pour un atterrissage en douceur. De plus, le marqueur devient trop grand dans le champ de vision, ce qui rend la détection difficile. Le système maintient alors la dernière consigne connue.

3.7 Limites et améliorations

Le système avec contrôle proportionnel pur présente plusieurs limitations :

— **Oscillations** : Le contrôle P pur peut générer des oscillations autour de la cible, surtout à haute altitude

— **Erreur statique** : En présence de vent ou de perturbations, le système peut converger vers une position légèrement décalée

— **Cible fixe uniquement** : Le système ne peut pas suivre une cible en mouvement car il n'anticipe pas le déplacement

— **Pas d'adaptation** : Les gains fixes ne sont pas optimaux pour toutes les altitudes

Ces limitations motivent le passage à un contrôleur PID adaptatif dans l'étape suivante.

4 Étape 3 – Atterrissage sur cible mobile avec PID

La troisième étape consiste à étendre le système pour permettre l'atterrissage sur une cible mobile, en utilisant un contrôleur PID adaptatif.

4.1 Contexte et motivation

Le contrôle proportionnel simple de l'étape précédente fonctionne correctement pour une cible fixe, mais présente des limitations importantes pour une cible mobile :

— Le système réagit uniquement à l'erreur instantanée sans tenir compte de l'historique

— Il n'anticipe pas le mouvement futur de la cible

— Les oscillations peuvent être importantes, surtout lorsque la cible se déplace rapidement

— L'erreur statique peut être significative en présence de perturbations

Un contrôleur PID (Proportionnel-Intégral-Dérivé) permet de résoudre ces problèmes en combinant trois types de corrections.

4.2 Architecture du contrôleur PID

4.2.1 Limites du contrôle P pur

Le contrôle proportionnel pur présente plusieurs défauts :

— **Réaction retardée** : La correction n'est appliquée qu'après l'apparition de l'erreur

— **Erreur statique** : En présence d'un biais constant (vent, dérive), le système converge vers une position décalée

— **Oscillations** : Un gain élevé améliore la rapidité mais provoque des oscillations ; un gain faible réduit les oscillations mais ralentit la convergence

4.2.2 Principe du PID

Le contrôleur PID combine trois termes :

$$v(t) = K_p \times e(t) + K_i \times \int_0^t e(\tau)d\tau + K_d \times \frac{de(t)}{dt} \tag{2}$$

où :

— **Terme proportionnel** ($K_p \times e(t)$) : Correction proportionnelle à l'erreur actuelle

— **Terme intégral** ($K_i \times \int e(\tau)d\tau$) : Accumule l'erreur au fil du temps pour éliminer l'erreur statique

— **Terme dérivé** ($K_d \times \frac{de(t)}{dt}$) : Anticipe l'évolution de l'erreur et amortit les oscillations

4.2.3 Implémentation C++

Le contrôleur PID est implémenté dans une classe dédiée :

```
class PIDController {
public:
    PIDController(float kp, float ki, float kd, float
       max_integral)
        : kp_(kp), ki_(ki), kd_(kd), max_integral_(max_integral),
          integral_(0.0f), previous_error_(0.0f) {}

    float compute(float error, float dt) {
        // Terme proportionnel
        float p_term = kp_ * error;

        // Terme int gral avec anti-windup
        integral_ += error * dt;
        integral_ = std::clamp(integral_, -max_integral_,
           max_integral_);
        float i_term = ki_ * integral_;

        // Terme d riv
        float derivative = (error - previous_error_) / dt;
        float d_term = kd_ * derivative;

        previous_error_ = error;

        return p_term + i_term + d_term;
    }

    void reset() {
        integral_ = 0.0f;
        previous_error_ = 0.0f;
```

```
    }

    void set_gains(float kp, float ki, float kd) {
        kp_ = kp;
        ki_ = ki;
        kd_ = kd;
    }

private:
    float kp_, ki_, kd_;
    float max_integral_;
    float integral_;
    float previous_error_;
};
```

Listing 9 – Classe PIDController

L'implémentation inclut un mécanisme d'*anti-windup* qui limite l'accumulation de l'intégrale pour éviter un dépassement excessif.

4.2.4 Réglage des gains adaptatifs

Les gains du PID sont adaptés en fonction de l'altitude pour optimiser les performances :

```
void update_pid_gains(float altitude) {
    if (altitude > 3.0f) {
        // Haute altitude : r activit     leve
        pid_x.set_gains(2.0f, 0.5f, 0.3f);
        pid_y.set_gains(2.0f, 0.5f, 0.3f);
    } else {
        // Basse altitude : stabilit  prioritaire
        pid_x.set_gains(1.5f, 0.3f, 0.2f);
        pid_y.set_gains(1.5f, 0.3f, 0.2f);
    }
}
```

Listing 10 – Adaptation des gains selon l'altitude

Cette adaptation se base sur la formule :

$$K_p = K_p^{\min} + (K_p^{\max} - K_p^{\min}) \times \text{error\_factor} \tag{3}$$

où `error_factor` dépend de l'altitude et de l'erreur de position.

4.3 Plateforme mobile

Pour tester l'atterrissage sur cible mobile, une plateforme robotique mobile a été créée dans Gazebo :

— **Type** : Robot à différentiel (DiffDrive)

— **Marqueur ArUco** : Taille 50×50 cm (DICT_4X4_50, ID 0)

— **Contrôle** : Via topic ROS `/cmd_vel`

— **Vitesse** : Configurable de 0.5 à 2.0 m/s

Le robot est défini dans un fichier SDF Gazebo et placé dans l'environnement de simulation. Il peut être contrôlé pour se déplacer en ligne droite ou suivre des trajectoires courbes.

4.4 Stratégie de descente en 3 phases

La stratégie d'atterrissage sur cible mobile est organisée en trois phases distinctes :

4.4.1 Phase 1 : Approche active (altitude ¿ 2m)

Dans cette phase, le drone :

— Détecte activement le marqueur ArUco

— Applique le contrôle PID pour suivre la cible

— Descend progressivement à 0.4 m/s

— Maintient une vitesse horizontale maximale de 2.0 m/s

```
if (altitude > 2.0f && aruco_detected) {
    float error_x = (marker_center.x - frame_center_x) /
       frame_width;
    float error_y = (marker_center.y - frame_center_y) /
       frame_height;

    float vx = pid_x.compute(error_x, dt);
    float vy = pid_y.compute(error_y, dt);

    velocity.right_m_s = std::clamp(vx, -2.0f, 2.0f);
    velocity.forward_m_s = std::clamp(vy, -2.0f, 2.0f);
    velocity.down_m_s = 0.4f;
}
```

Listing 11 – Phase d'approche active

4.4.2 Phase 2 : Stabilisation (altitude = 2m, durée 2 secondes)

À l'altitude de 2 mètres, le drone effectue une pause de stabilisation :

— Maintien de l'altitude constante pendant 2 secondes

— Affinage du centrage sur la cible

— Mémorisation de la vitesse et direction de la cible

— Préparation pour la descente finale

```
if (altitude <= 2.0f && altitude > 1.9f && !stabilization_done) {
    // Maintenir l'altitude
    velocity.down_m_s = 0.0f;

    // Continuer le suivi
    velocity.right_m_s = std::clamp(vx, -1.0f, 1.0f);
    velocity.forward_m_s = std::clamp(vy, -1.0f, 1.0f);

    // Compter le temps
    stabilization_time += dt;
    if (stabilization_time >= 2.0f) {
        stabilization_done = true;
        // M moriser la trajectoire
        last_known_vx = velocity.right_m_s;
        last_known_vy = velocity.forward_m_s;
    }
}
```

Listing 12 – Phase de stabilisation

4.4.3 Phase 3 : Descente finale (altitude ¡ 2m, boucle ouverte)

Dans la phase finale, le marqueur devient trop grand pour être détecté correctement. Le système passe en boucle ouverte :

— Utilisation de la trajectoire mémorisée

— Descente lente à 0.25 m/s

— Pas de correction visuelle

— Hypothèse : la cible maintient sa trajectoire rectiligne

```
if (altitude < 2.0f && stabilization_done) {
    // Maintenir la trajectoire m moris e
    velocity.right_m_s = last_known_vx;
    velocity.forward_m_s = last_known_vy;
    velocity.down_m_s = 0.25f;

    // Atterrissage final
```

```
8       if (altitude < 0.3f) {
9           action.land();
10          break;
11      }
12  }
```

Listing 13 – Descente finale en boucle ouverte

4.5 Validation expérimentale

Le système a été testé avec différentes vitesses de déplacement de la plateforme :

Vitesse (m/s)	Précision (cm)	Résultat
0.5	10	Succès
1.0	12	Succès
1.5	15	Succès
2.0	15	Succès (limite)
2.5	–	Oscillations

TABLE 1 – Résultats des tests d'atterrissage

Les résultats montrent que :

— Le système fonctionne de manière fiable jusqu'à 2.0 m/s

— La précision d'atterrissage est de 10-15 cm

— Au-delà de 2.0 m/s, des oscillations apparaissent et déstabilisent le système

4.6 Conclusion de l'étape

Le contrôleur PID adaptatif avec stratégie de descente multi-phases permet un atterrissage réussi sur des cibles mobiles se déplaçant jusqu'à 2.0 m/s. Les gains adaptatifs améliorent la stabilité à basse altitude tout en maintenant la réactivité à haute altitude. La phase de stabilisation à 2 mètres est critique pour mémoriser la trajectoire et préparer la descente finale en boucle ouverte.

5 Étape 4 – Limites du système

Malgré les performances satisfaisantes du système, plusieurs limitations ont été identifiées lors des tests.

5.1 Limitation en vitesse

Le système devient instable au-delà de 2.0 m/s de vitesse de la cible :

— **Oscillations** : À haute vitesse, le délai entre détection et correction devient significatif, causant des oscillations de plus en plus importantes

— **Saturation des commandes** : Les corrections calculées par le PID dépassent régulièrement les limites de vitesse du drone

— **Retard visuel** : Le traitement des images (30 Hz) introduit un retard qui devient problématique à haute vitesse

Des améliorations possibles incluent :

— Augmentation de la fréquence de traitement des images

— Prédiction du mouvement de la cible (filtre de Kalman)

— Gains PID encore plus adaptatifs en fonction de la vitesse de la cible

5.2 Gestion des trajectoires courbes

Le système suppose que la cible se déplace en ligne droite pendant la phase finale. Cette hypothèse est violée si la cible effectue un virage :

— **Perte de précision** : Le drone continue en ligne droite alors que la cible tourne

— **Distance minimale requise** : Pour des virages serrés, il faut que le drone soit à plus de 5-6 mètres d'altitude pour compenser le décalage

— **Nécessité d'un virage avant** : La cible doit effectuer son virage avant que le drone atteigne l'altitude de 2 mètres

Une solution serait d'implémenter un système de prédiction de trajectoire basé sur l'historique des positions détectées, ou d'étendre la détection visuelle à plus basse altitude en utilisant plusieurs caméras.

5.3 Hypothèse trajectoire rectiligne phase finale

La descente finale en boucle ouverte (altitude ¡ 2m) repose sur l'hypothèse d'une trajectoire rectiligne de la cible. Cette hypothèse limite les scénarios d'utilisation :

— La cible doit maintenir cap et vitesse constants

— Les perturbations (vent, pente du terrain) ne sont pas compensées

— Pas de réaction possible aux imprévus

Pour lever cette limitation, il faudrait :

— Utiliser un marqueur ArUco plus petit pour permettre la détection à basse altitude

— Employer une caméra avec un champ de vision plus large

— Intégrer d'autres capteurs (GPS différentiel, lidar)

6 Impact écologique

Cette section présente une réflexion sur l'impact écologique du système d'atterrissage autonome de drones.

6.1 Bénéfices potentiels

Les systèmes d'atterrissage autonome sur plateformes mobiles peuvent contribuer positivement à l'environnement :

6.1.1 Réduction des trajets

Un drone capable d'atterrir sur un véhicule en mouvement peut :

— Réduire la distance totale parcourue en évitant les allers-retours vers une base fixe
— Permettre des livraisons en relais avec des véhicules terrestres, optimisant ainsi les itinéraires
— Économiser de l'énergie en minimisant les phases de vol

6.1.2 Efficacité énergétique

La précision de l'atterrissage améliore l'efficacité :

— Moins de temps en vol stationnaire (très énergivore)
— Moins de tentatives d'atterrissage ratées
— Optimisation des profils de vol

6.1.3 Applications environnementales

Le système peut être utilisé pour :

— Surveillance de la faune sans perturbation (atterrissage sur véhicules de recherche)
— Inspection d'infrastructures mobiles (navires, trains)
— Intervention d'urgence avec déploiement rapide

6.2 Risques et nuisances

Il faut néanmoins considérer les impacts négatifs potentiels :

6.2.1 Effet rebond

La facilité d'utilisation peut entraîner :

— Augmentation du nombre de vols de drones
— Utilisation pour des besoins non essentiels
— Remplacement de solutions moins impactantes (vélo, marche)

6.2.2 Nuisances

Les drones génèrent :

— Pollution sonore (nuisance pour la faune et les populations)
— Pollution visuelle
— Perturbation de la faune aviaire

6.2.3 Cycle de vie

L'impact du cycle de vie doit être pris en compte :

— Fabrication des composants électroniques (terres rares, énergie)

— Durée de vie limitée des batteries

— Difficultés de recyclage

6.3 Métriques et bonnes pratiques

Pour minimiser l'impact écologique, plusieurs mesures peuvent être mises en place :

— **Optimisation des algorithmes** : Réduire le temps de calcul et la consommation énergétique

— **Utilisation raisonnée** : Privilégier les applications à forte valeur ajoutée

— **Intégration multimodale** : Combiner drones et véhicules terrestres pour optimiser les trajets

— **Éco-conception** : Choisir des matériaux recyclables et des batteries à longue durée de vie

— **Zones de restriction** : Respecter les zones protégées et les périodes sensibles pour la faune

Il est essentiel que le développement de ces technologies s'accompagne d'une réflexion éthique et d'un cadre réglementaire approprié pour garantir un impact environnemental positif.

7 Conclusion

Ce stage à l'USTH a permis de développer un système complet d'atterrissage autonome de drones sur des plateformes mobiles en utilisant la détection de marqueurs ArUco et un contrôle PID adaptatif.

7.1 Validation de l'approche

L'approche adoptée, combinant détection ArUco, contrôleur PID adaptatif et stratégie de descente multi-phases, s'est révélée efficace :

— Le cadre de simulation (PX4 + MAVSDK + Gazebo) a permis un développement itératif et des tests répétables

— La détection ArUco optimisée fonctionne de manière fiable jusqu'à 10 mètres d'altitude

— Le contrôleur PID adaptatif assure un suivi stable de la cible mobile

— La stratégie de descente en trois phases gère efficacement la perte du retour visuel

7.2 Performances atteintes

Les performances mesurées sont :

— **Vitesse maximale de la cible** : 2.0 m/s

— **Précision d'atterrissage** : 10-15 cm

— **Altitude de détection** : 10 mètres

— **Taux de réussite** : ¿ 95% pour des trajectoires rectilignes

Ces résultats démontrent la faisabilité de l'atterrissage autonome sur cibles mobiles dans des conditions contrôlées.

7.3 Limitations identifiées

Plusieurs limitations ont été identifiées :

— Instabilité au-delà de 2.0 m/s de vitesse de cible

— Nécessité d'une trajectoire rectiligne en phase finale

— Sensibilité aux conditions d'éclairage pour la détection ArUco

— Absence de gestion des obstacles dynamiques

7.4 Perspectives d'amélioration

Plusieurs pistes d'amélioration peuvent être envisagées :

— **Prédiction de trajectoire** : Implémenter un filtre de Kalman pour anticiper le mouvement de la cible

— **Fusion de capteurs** : Combiner détection visuelle et GPS différentiel pour améliorer la précision

— **Détection multi-échelle** : Utiliser plusieurs marqueurs de tailles différentes pour couvrir toutes les altitudes

— **Apprentissage automatique** : Entraîner un réseau de neurones pour prédire les trajectoires complexes

— **Tests en conditions réelles** : Valider le système sur un drone physique avec un robot mobile réel

7.5 Dimension humaine du stage

Au-delà des aspects techniques, ce stage a été une expérience humaine enrichissante :

— **Environnement international** : Travailler au Vietnam a permis de découvrir une culture différente et d'améliorer mes compétences en communication interculturelle

— **Autonomie** : Le projet nécessitait une grande autonomie dans la recherche de solutions et la résolution de problèmes

— **Collaboration** : Les échanges avec le Dr Pham Xuan Tung et l'équipe de l'USTH ont été très formateurs

— **Rigueur scientifique** : La démarche expérimentale et la documentation rigoureuse ont renforcé ma méthodologie de travail

Ce stage a confirmé mon intérêt pour la robotique autonome et la vision par ordinateur, domaines que je souhaite approfondir dans la suite de mon parcours académique et professionnel.

Je tiens à remercier chaleureusement le Dr Pham Xuan Tung pour son encadrement et ses conseils précieux, M. Feuilloy Matthieu pour son suivi depuis l'ESEO, ainsi que toute l'équipe de l'USTH pour leur accueil et leur soutien tout au long de ce stage.