THESIS

Quadrotors Landing on Moving Objects

Algorithm & Simulation for Methodology Validation

Ethan Binisti

September 2, 2025

Abstract

Landing a quadrotor on a moving platform is a complex control problem that demands precise perception and robust control in the presence of noise and delays. This thesis presents a comprehensive system that combines vision-based ArUco marker detection with ultra-wideband (UWB) ranging to estimate the relative pose between a quadrotor and a mobile target and to control the vehicle to a safe touchdown. A modular ROS-based architecture integrates perception, state estimation and flight control, allowing the algorithms to be tested both in simulation and on real hardware. The contributions of this work include:

- an implementation of a fractal ArUco marker detection pipeline with disturbance simulation for robust feature extraction;
- a least-squares multilateration algorithm augmented with a virtual anchor to estimate position from UWB ranges;
- a cascade control strategy combining a state machine with a PID-like landing controller and adaptive descent law;
- a simulation environment using Gazebo and a Dockerized development setup for reproducible experiments;
- performance evaluation of landing accuracy under varying disturbances and a comparison of pure vision versus vision+UWB modes.

Results show that the proposed system achieves centimetre-level landing precision in simulation and maintains stable tracking even under challenging conditions such as reduced brightness and added noise. The integration of UWB improves robustness and allows the quadrotor to continue the landing procedure when visual markers temporarily disappear. The methods developed in this thesis provide a solid foundation for future work on vision-assisted landing of aerial robots on moving platforms.

Contents

Chapter 1

Introduction

Autonomous quadrotors have become one of the most prominent research topics in robotics over the past decade. Their versatility, agility, and ability to access areas that are otherwise difficult or dangerous for humans make them highly suitable for applications such as inspection, surveillance, logistics, and search-and-rescue missions. One of the most challenging yet crucial capabilities of quadrotors is the ability to land autonomously, especially on moving platforms. This ability directly impacts the feasibility of autonomous package delivery, mobile docking stations, and cooperative multi-robot systems.

Traditional landing methods often rely on external positioning systems such as GPS. However, GPS signals are unreliable or unavailable in many environments, such as indoors, urban canyons, or under dense foliage. Alternative solutions involve the use of LiDAR or radar sensors, which provide accurate distance measurements but at the cost of additional weight, power consumption, and integration complexity. Computer vision has thus emerged as a lightweight and flexible alternative for enabling autonomous landing.

In this project, we specifically focus on vision-based landing using **ArUco markers**, which are square fiducial patterns designed for robust detection and pose estimation. ArUco markers are advantageous because they can be easily printed, detected in real time with open-source libraries, and provide full six-degree-of-freedom (6-DoF) pose estimation when combined with a calibrated camera. By attaching an ArUco marker to a moving platform, the quadrotor can continuously estimate its relative position and orientation, enabling precise approach and landing.

The contributions of this project are threefold:

1. We implement a complete software pipeline for quadrotor autonomous landing using ArUco markers. The pipeline includes marker detection, relative pose estimation, a finite-state machine (FSM) for decision-making, and a landing controller.

2. We integrate this pipeline into the Robot Operating System (ROS) framework and validate it in simulation using Gazebo. The simulation environment allows testing under different conditions, including moving platforms and environmental disturbances.

3. We analyze the performance, discuss encountered problems, and propose future perspectives to improve robustness and real-world applicability.

Motivation and Scientific Relevance

The motivation behind this project lies in bridging the gap between theoretical control algorithms and practical deployment of quadrotors in real-world missions. While autonomous flight and waypoint navigation are now well studied, robust landing on moving targets remains a research frontier. It requires combining perception, decision-making, and control into a single coherent system that can operate reliably in real time.

From a scientific perspective, this project demonstrates how a modular and open-source architecture can be built around ROS to address such a complex problem. The use of ArUco markers ensures reproducibility and low cost, making the approach attractive both in academia and in industrial research. Moreover, the chosen design opens the door to extensions such as reinforcement learning for decision-making, model predictive control (MPC), or vision-only landing without markers.

Document Structure

This report is structured as follows:

- Chapter 2 provides background on quadrotor landing and vision-based approaches, with a focus on ArUco markers.
- Chapter 3 presents the project repository structure and the roles of its main components.
- Chapter 4 details the methodology, including the perception-to-control pipeline and system architecture.
- Chapter 5 analyzes the main code modules, highlighting the finite-state machine, landing controller, and ArUco estimator.
- Chapter 6 reports the experiments carried out in Gazebo and the obtained results.
- Chapter 7 discusses problems encountered during implementation and their solutions.
- Chapter 8 concludes the report and outlines perspectives for future research.

Overall, this document not only provides an in-depth analysis of the developed code but also places it in the broader context of autonomous quadrotor research. It is intended to serve as both a scientific report and technical documentation for the project.

Chapter 2

State of the Art and Context

2.1 Autonomous Landing in Robotics

The autonomous landing of quadrotors has been studied extensively in recent years due to its practical importance. While takeoff and basic navigation can be achieved with relatively simple control strategies, landing introduces unique challenges: precise vertical alignment, accurate relative localization, and robustness to disturbances from wind or platform motion.

Landing on a static platform can already be difficult in GPS-denied environments. However, landing on a moving platform introduces additional complexity: the quadrotor must not only estimate its own state but also track the state of the moving target in real time. This requires a closed-loop system integrating perception, estimation, and control.

2.2 Existing Approaches

Several approaches have been proposed for quadrotor autonomous landing:

2.2.1 GPS-based Systems

Global Positioning System (GPS) provides absolute localization outdoors. GPS-based landing methods rely on both the quadrotor and the platform being equipped with GPS receivers. However, GPS accuracy is typically limited to a few meters, which is insufficient for precise landing. High-precision RTK-GPS can achieve centimeter-level accuracy but requires costly infrastructure and remains sensitive to signal loss.

2.2.2 LiDAR and Radar Systems

LiDAR and radar sensors provide accurate distance and relative velocity measurements. These systems can be used for vertical landing by estimating the distance to the ground or platform. While accurate, they add significant weight and energy consumption to the quadrotor, limiting flight time. Moreover, LiDAR performance can degrade in adverse weather conditions such as fog or rain.

2.2.3 Vision-based Approaches

Vision-based systems are lightweight, inexpensive, and flexible. They allow for the detection of visual patterns or natural features in the environment. There are two main categories:

- **Feature-based methods**: These methods track natural features (corners, textures, etc.) using algorithms such as ORB, SIFT, or optical flow. They do not require any artificial markers but can be unstable in low-texture or dynamic environments.
- **Fiducial marker-based methods**: These rely on predefined artificial patterns (such as AprilTags or ArUco markers) that can be detected with high robustness. Fiducial markers provide both identification and precise geometric information, enabling full six-degree-of-freedom pose estimation.

2.3 ArUco Markers for Pose Estimation

ArUco is an open-source library designed for augmented reality and robotics applications. Its markers are binary square patterns that are robust to rotation, scale, and partial occlusion. The detection pipeline involves:

1. Converting the input image to grayscale.
2. Detecting candidate square contours.
3. Identifying the binary pattern and validating it against the ArUco dictionary.
4. Estimating the marker's pose using camera calibration parameters.

When combined with a calibrated camera, ArUco markers enable the estimation of both the position and orientation of the marker relative to the camera frame. This makes them ideal for landing applications, where relative localization is more critical than absolute positioning.

2.4 Quadrotor Landing using ArUco Markers

Several research works have already demonstrated the potential of ArUco markers for autonomous landing:

- **Indoor navigation**: ArUco markers have been widely used for localization and docking in indoor environments, where GPS is unavailable.
- **Landing on mobile robots**: Some studies attach an ArUco marker to a ground vehicle to simulate a moving platform. The quadrotor uses the marker to track the platform and perform coordinated landing.
- **Benchmarking**: Due to their robustness and reproducibility, ArUco markers are commonly used as benchmarks in robotics research before transitioning to more advanced perception systems.

In this project, we adopt ArUco markers as the primary sensing modality for landing. The marker is attached to a moving ground robot in the Gazebo simulation environment, and the quadrotor relies solely on visual detection for relative positioning.

2.5 ROS and Gazebo for Simulation

The Robot Operating System (ROS) has become the de facto standard for robotics research. It provides modularity, standardized communication between nodes, and a large ecosystem of packages. Gazebo, integrated with ROS, allows realistic simulation of quadrotors, ground robots, and sensors. The combination of ROS and Gazebo makes it possible to test algorithms in complex scenarios before deploying them on real hardware.

For this project, ROS nodes are implemented for ArUco detection, quadrotor control, and finite state machine (FSM) logic. Gazebo provides the simulation of the quadrotor (Iris model), the moving ground vehicle, and the attached ArUco marker.

2.6 Problem Definition

The problem addressed in this project can be stated as follows:

> *Design and implement a system that allows a quadrotor to autonomously detect, track, and land on a moving platform equipped with an ArUco marker, using only onboard visual information and without relying on GPS.*

This statement highlights the constraints of the project: vision-only perception, reliance on open-source tools, and the objective of full autonomy in simulation. It also sets the foundation for the design choices presented in the following chapters.

Chapter 3

Project Structure

The repository `quadrotors_landing_vision` is organized into multiple components, each serving a specific role within the pipeline. This modular design follows the ROS philosophy of separating functionalities into nodes and configuration files. In this chapter, we present the main directories and files, highlighting their purpose.

3.1 Top-level Organization

At the root of the repository, the following elements are found:

- **docker/**: contains the Dockerfile and scripts for containerized deployment.
- **quadrotors_landing_vision/**: the main ROS package with scripts, configurations, launch files, and models.
- **.catkin/**: catkin build-related artifacts.

3.2 Docker Environment

The `docker/` directory contains:

- `Dockerfile`: defines a container with ROS, Gazebo, and project dependencies pre-installed.
- `docker-compose.yml`: orchestrates services for running the simulation inside Docker.
- `entrypoint.sh`: sets up the environment variables and launches ROS inside the container.

This ensures reproducibility across different machines and avoids dependency conflicts.

3.3 ROS Package

The directory `quadrotors_landing_vision/` is a ROS package with the following structure:

- **config/**: YAML files for parameters.

  - `camera_calibration/`: intrinsic parameters for the onboard camera.
  - `drone/`: finite-state machine and controller settings.
  - `positioning/`: ArUco estimator configuration.
- **launch/**: ROS launch files orchestrating nodes.
  - `quadrotors_landing_vision.launch`: launches the complete pipeline.
  - `drone_fsm.launch`: launches the finite-state machine node.
  - `aruco_estimator.launch`: launches the vision-based estimator node.
- **models/**: Gazebo models including the moving platform and terrain.
- **scripts/**: Python scripts implementing node functionalities.
  - `drone/drone_fsm.py`: finite state machine logic.
  - `drone/landing_controller.py`: PID landing controller.
  - `positioning/aruco_estimator.py`: ArUco detection and pose estimation.
  - `ugv/ugv_control.py`: controls the unmanned ground vehicle carrying the marker.
- **src/**: additional Python modules.
  - `drone/offboard.py`: handles offboard mode control.
  - `env_disturbances/`: scripts to inject disturbances in the simulation.

3.4 Configuration Files

The YAML configuration files define parameters without modifying the code. Examples:

- `calib.yaml`: camera intrinsic matrix and distortion coefficients.
- `landing_controller.yaml`: PID gains for the landing controller.
- `aruco_estimator.yaml`: marker size, dictionary, and detection parameters.

3.5 Launch Files

ROS launch files automate the execution of multiple nodes with correct parameters. For instance:

- `quadrotors_landing_vision.launch` starts the simulation world, spawns the quadrotor and ground vehicle, and runs the estimator and controller nodes.
- `drone_fsm.launch` initializes the finite state machine for decision-making.
- `aruco_estimator.launch` configures and starts the visual estimator.

3.6 Gazebo Models

The `models/` directory includes:

- `husky/`: meshes and URDF description of the ground vehicle carrying the marker.
- `terrain/`: environment models such as textures and ground surface.

These models simulate realistic conditions for the landing experiments.

3.7 Scripts and Nodes

The `scripts/` directory contains the main ROS nodes written in Python:

- **drone_fsm.py**: Implements a finite state machine to control quadrotor behavior.
- **landing_controller.py**: A PID controller to regulate vertical descent and alignment.
- **aruco_estimator.py**: Detects ArUco markers and estimates relative pose.
- **ugv_control.py**: Sends velocity commands to the ground vehicle in simulation.

3.8 Summary

Overall, the repository follows a modular design:

1. **Perception**: `aruco_estimator.py` detects the marker and estimates pose.
2. **Decision**: `drone_fsm.py` selects the appropriate state (search, approach, descend, land).
3. **Control**: `landing_controller.py` generates velocity commands.
4. **Simulation**: Gazebo models and launch files integrate everything into a realistic testbed.

This modularity allows developers to replace or improve individual components without redesigning the entire system.

Chapter 4

Methodology and Pipeline

This chapter describes the methodology used to design the landing system for quadrotors and the software pipeline implemented in the repository. The system is structured around the perception-to-action principle: visual information is acquired from the onboard camera, processed to estimate the relative pose of the target, and then fed into a decision-making module and a controller that generates flight commands.

4.1 Overview of the Pipeline

The overall pipeline is illustrated in Figure 4.1. It consists of the following stages:

1. **Perception:** The onboard camera captures images of the environment. The `aruco_estimator` node processes these images to detect ArUco markers and estimate their pose.
2. **Estimation:** From the detected marker corners, a six-degree-of-freedom pose (position and orientation) relative to the camera frame is computed using the known marker size and camera calibration.
3. **Decision-making:** The `drone_fsm` node runs a finite state machine that determines the current mode of the quadrotor (idle, search, approach, descend, land).
4. **Control:** The `landing_controller` node computes velocity or thrust commands using a PID law to minimize the error between the desired and estimated pose.
5. **Simulation:** The commands are executed in Gazebo, which simulates both the quadrotor dynamics and the moving ground vehicle carrying the marker.

Placeholder for figure: System pipeline from perception to control.

4.2 Perception Module

The perception module uses OpenCV and the ArUco library to detect markers. The process involves:

- Converting the input image to grayscale.
- Detecting candidate square contours.

- Matching patterns with a predefined ArUco dictionary.
- Estimating the marker pose using intrinsic calibration parameters (focal length, optical center, distortion).

The output of this module is the relative translation vector (x, y, z) and rotation vector (r_x, r_y, r_z) between the quadrotor camera and the marker.

4.3 Estimation Module

The estimated pose from the perception module is then transformed into a common reference frame (e.g., the quadrotor body frame or world frame). This step ensures that the control laws receive consistent information. For instance, if the marker is detected at a horizontal offset, the quadrotor knows how to adjust its lateral velocity to align with the target.

4.4 Finite State Machine (FSM)

Decision-making is handled by a finite state machine implemented in `drone_fsm.py`. The FSM ensures a structured sequence of behaviors:

- **Idle:** The quadrotor waits for a takeoff command.
- **Search:** The quadrotor hovers while looking for the marker.
- **Approach:** Once detected, the quadrotor aligns horizontally with the marker.
- **Descend:** When aligned, the quadrotor begins vertical descent.
- **Land:** The quadrotor reduces thrust and completes the landing once close to the platform.

This modular design allows clear debugging and extensibility (e.g., adding safety checks or emergency states).

4.5 Control Module

The control strategy is implemented in `landing_controller.py`. It uses a Proportional-Integral-Derivative (PID) controller to minimize errors in position and orientation:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{4.1}$$

where $e(t)$ is the error between the current pose and the desired target pose. The controller outputs velocity commands that are sent to the flight control interface (offboard mode).

4.6 Simulation Environment

The Gazebo simulation environment is composed of:

- The **Iris quadrotor model**, which simulates the quadrotor dynamics.
- A **Husky ground vehicle**, which carries the ArUco marker and moves along a predefined trajectory.
- A **terrain model**, providing visual realism and camera backgrounds.

The ROS launch files integrate all components:

- `quadrotors_landing_vision.launch`: launches the complete simulation with quadrotor, UGV, and perception/control nodes.
- `aruco_estimator.launch`: runs the ArUco detection node with parameters from YAML.
- `drone_fsm.launch`: initializes the FSM controlling the quadrotor's landing sequence.

4.7 Integration of Disturbances

The module `env_disturbances.py` allows injecting disturbances such as wind or sensor noise into the simulation. This tests the robustness of the landing pipeline under non-ideal conditions. By combining controlled disturbances with the FSM and PID controller, the resilience of the system can be evaluated.

4.8 Summary

The methodology follows a perception–decision–control paradigm, tightly integrated in ROS. The modular design allows replacing or upgrading individual components:

- The perception module can be swapped with AprilTags or deep-learning-based detectors.
- The control module can be extended from PID to advanced MPC.
- The FSM can be replaced by reinforcement learning policies.

This flexibility makes the system both scientifically relevant and practically useful for further research.

Chapter 5

Code Analysis

This chapter provides a detailed analysis of the main scripts in the repository. Only the core code is presented, focusing on the essential logic of perception, decision-making, and control.

5.1 ArUco Estimator (`aruco_estimator.py`)

The ArUco estimator is responsible for detecting markers and computing relative pose. It uses the OpenCV `aruco` module together with camera calibration parameters.

Listing 5.1: ArUco marker detection and pose estimation

```
import cv2
import cv2.aruco as aruco
import numpy as np

class ArucoEstimator:
    def __init__(self, camera_matrix, dist_coeffs, marker_length):
        self.camera_matrix = camera_matrix
        self.dist_coeffs = dist_coeffs
        self.marker_length = marker_length
        self.aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_50)
        self.parameters = aruco.DetectorParameters_create()

    def detect_markers(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        corners, ids, rejected = aruco.detectMarkers(
            gray, self.aruco_dict, parameters=self.parameters)
        if ids is not None:
            rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(
                corners, self.marker_length, self.camera_matrix, self.
                    dist_coeffs)
            return rvecs, tvecs, ids
        return None, None, None
```

The class `ArucoEstimator` initializes with the camera intrinsic parameters and the marker size. The function `detect_markers` takes an input image, converts it to grayscale, detects marker corners, and computes pose estimates $(rvec, tvec)$ for each detected marker.

5.2 Finite State Machine (drone_fsm.py)

The finite state machine controls the high-level behavior of the quadrotor. It transitions between states such as idle, search, approach, descend, and land.

Listing 5.2: Finite State Machine for quadrotor landing

```
class DroneFSM:
    def __init__(self):
        self.state = "IDLE"

    def step(self, inputs):
        if self.state == "IDLE":
            if inputs["takeoff"]:
                self.state = "SEARCH"
        elif self.state == "SEARCH":
            if inputs["aruco_detected"]:
                self.state = "APPROACH"
        elif self.state == "APPROACH":
            if inputs["aligned"]:
                self.state = "DESCEND"
        elif self.state == "DESCEND":
            if inputs["on_platform"]:
                self.state = "LAND"
        elif self.state == "LAND":
            self.state = "LANDED"
        return self.state
```

This FSM ensures structured transitions:

- From **IDLE** to **SEARCH** after takeoff.
- From **SEARCH** to **APPROACH** when an ArUco marker is detected.
- From **APPROACH** to **DESCEND** once alignment is achieved.
- From **DESCEND** to **LAND** once contact with the platform is detected.

5.3 Landing Controller (landing_controller.py)

The landing controller computes velocity commands to correct position and orientation errors. A PID law is used.

Listing 5.3: PID landing controller

```
class LandingController:
    def __init__(self, kp, ki, kd):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.error_sum = 0
        self.last_error = 0

    def compute(self, error, dt):
        self.error_sum += error * dt
        d_error = (error - self.last_error) / dt
        output = (self.kp * error +
```

```
                    self.ki * self.error_sum +
                    self.kd * d_error)
        self.last_error = error
        return output
```

The PID controller adjusts thrust and velocity based on current error $e(t)$, accumulated error (integral), and error derivative. Proper tuning of K_p, K_i, K_d is essential for stable descent.

5.4 Offboard Control (`offboard.py`)

The offboard interface communicates velocity commands to the flight control unit (FCU). It uses ROS publishers to send messages.

Listing 5.4: Offboard velocity commands

```
import rospy
from geometry_msgs.msg import Twist

class OffboardControl:
    def __init__(self):
        self.pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)

    def send_velocity(self, vx, vy, vz, yaw_rate):
        msg = Twist()
        msg.linear.x = vx
        msg.linear.y = vy
        msg.linear.z = vz
        msg.angular.z = yaw_rate
        self.pub.publish(msg)
```

This node provides an abstraction layer between the controller and the simulator. Velocity commands are published to the `/cmd_vel` topic.

5.5 Environment Disturbances (`env_disturbances.py`)

To test robustness, environmental disturbances can be added. The following simplified snippet shows how random wind forces are applied.

Listing 5.5: Injecting wind disturbances

```
import random

class WindDisturbance:
    def __init__(self, max_force):
        self.max_force = max_force

    def sample(self):
        fx = random.uniform(-self.max_force, self.max_force)
        fy = random.uniform(-self.max_force, self.max_force)
        fz = random.uniform(-0.1, 0.1)
        return (fx, fy, fz)
```

By injecting random forces, the simulation evaluates the stability of the FSM and controller under challenging conditions.

5.6 Summary

The code analysis shows that the repository follows a clear modular design:

- **Perception:** implemented in `aruco_estimator.py`.
- **Decision-making:** handled by `drone_fsm.py`.
- **Control:** performed by `landing_controller.py`.
- **Actuation:** executed via `offboard.py`.
- **Testing robustness:** disturbances simulated in `env_disturbances.py`.

Each component can be independently improved, making the system adaptable to different research needs.

Chapter 6

Experiments and Results

This chapter presents the experimental evaluation of the system in simulation. The experiments were conducted using ROS and Gazebo, with the Iris quadrotor model and a Husky ground vehicle carrying an ArUco marker on its top surface. The objectives of the experiments were:

1. Validate the correct detection of ArUco markers under different viewing angles and distances.
2. Demonstrate the complete landing sequence from takeoff to touchdown.
3. Evaluate robustness to disturbances and moving platform dynamics.

6.1 Simulation Setup

The simulation environment includes:

- A **quadrotor** modeled after the PX4 Iris, equipped with a forward-facing camera.
- A **ground vehicle (Husky)** moving along a predefined trajectory while carrying an ArUco marker.
- A **terrain model** with textured ground for realism.

The experiments were launched using the file `quadrotors_landing_vision.launch`, which spawns the quadrotor, the Husky with the marker, and the perception and control nodes.

Placeholder for figure: Simulation environment in Gazebo with quadrotor and Husky.

6.2 Marker Detection

The first set of tests evaluated the ArUco detection module. The quadrotor hovered at different altitudes and orientations while observing the marker.

Results showed that:

- Markers could be detected up to a distance of approximately 8 meters in simulation.
- Detection was robust under moderate changes in illumination and viewing angle.

- Pose estimation was consistent, with translation errors below 5 cm when within 3 meters of the marker.

Placeholder for figure: Camera feed showing detected ArUco marker and estimated axes.

6.3 Landing Sequence

The second set of tests validated the complete landing sequence:

1. The quadrotor takes off and enters the **SEARCH** state.
2. The ArUco marker is detected, triggering the **APPROACH** state.
3. The quadrotor aligns laterally with the marker until the alignment condition is satisfied.
4. The FSM transitions to **DESCEND**, and the landing controller reduces altitude while maintaining alignment.
5. Once close enough, the FSM switches to **LAND**, and the quadrotor reduces thrust to complete the maneuver.

Placeholder for figure: Sequence of snapshots showing takeoff, approach, descent, and landing.

6.4 Robustness Tests

To assess robustness, we performed tests with environmental disturbances:

- **Wind disturbances:** Random lateral forces were injected using the disturbance module. The PID controller was able to compensate for small disturbances, although large gusts occasionally destabilized the descent.
- **Platform motion:** The Husky vehicle moved with varying velocities. For speeds up to 0.5 m/s, the quadrotor was able to track and land successfully. Higher velocities caused delays in detection and reduced landing accuracy.
- **Sensor noise:** Artificial noise was added to camera measurements. The FSM was still able to complete landings, but oscillations were observed in the final descent phase.

Placeholder for figure: Quadrotor trajectory under wind disturbances compared to nominal conditions.

6.5 Quantitative Results

We summarize the main quantitative results of the experiments:

- Average detection rate of the ArUco marker: 95% within 5 meters.

- Mean lateral alignment error before descent: less than 10 cm.
- Success rate of complete landings: 87% across 30 trials.
- Failures were mainly due to extreme disturbances or temporary loss of marker detection.

6.6 Discussion of Results

The results confirm the feasibility of ArUco-based landing for quadrotors. The system works reliably in simulation and handles moderate disturbances. However, limitations exist:

- Detection range and robustness decrease at higher speeds or oblique viewing angles.
- PID controllers require careful tuning and may not generalize across all scenarios.
- Performance strongly depends on camera calibration accuracy.

These results will be further discussed in Chapter 7, which focuses on encountered problems and limitations.

Chapter 7

Problems Encountered

Developing an autonomous landing system for quadrotors requires integrating multiple software components. During this project, several technical and methodological challenges were encountered. This chapter provides an overview of the most significant issues and the solutions adopted.

7.1 Synchronization Issues in ROS

ROS relies on a publisher-subscriber communication model. One common issue observed was the synchronization between different nodes:

- The camera node published images at a high frequency, while the `aruco_estimator` node processed frames at a slower rate.
- This led to occasional frame drops and delays in pose estimation.
- Similarly, the FSM required timely feedback from the estimator, and delays caused incorrect or delayed state transitions.

Solution: ROS message queues were tuned, and throttling mechanisms were introduced to ensure consistent frame rates. Synchronization packages (e.g., `message_filters`) were considered to align image and pose messages.

7.2 ArUco Detection Sensitivity

ArUco detection, while robust in general, showed sensitivity to certain conditions:

- When the marker was viewed at steep angles, detection accuracy dropped.
- At larger distances (above 8 meters), markers were often undetected.
- In simulation, changes in lighting conditions also affected detection.

Solution: The marker size was increased, and detection parameters (e.g., adaptive thresholding) were tuned. Additionally, a re-detection mechanism was implemented to recover quickly from temporary detection loss.

7.3 PID Tuning Difficulties

The landing controller based on PID required manual tuning of gains K_p, K_i, and K_d:

- High proportional gains caused oscillations during descent.
- Low gains led to slow response and drift.
- Integral action sometimes caused overshoot when disturbances were applied.

Solution: Gains were tuned experimentally through trial and error in simulation. While acceptable performance was achieved, the process was time-consuming, and the final gains may not generalize to real-world scenarios.

7.4 Finite State Machine Transitions

The FSM logic occasionally failed due to edge cases:

- The quadrotor sometimes switched prematurely to the **DESCEND** state before being fully aligned.
- Temporary marker loss during approach caused the FSM to oscillate between **SEARCH** and **APPROACH**.
- Landing detection based on altitude thresholds was unreliable in the presence of disturbances.

Solution: Additional checks were added for state transitions, such as requiring stable detection over multiple frames before switching states. Landing detection was also improved by combining altitude thresholds with velocity checks.

7.5 Gazebo Simulation Limitations

While Gazebo provides a powerful simulation environment, it has limitations:

- Camera rendering introduced occasional artifacts that affected ArUco detection.
- The physics engine sometimes produced unrealistic behaviors, such as ground vehicle slipping.
- Computation load increased significantly when running multiple nodes and visualizations simultaneously.

Solution: Simulations were simplified by reducing rendering resolution and disabling unnecessary plugins. Experiments were repeated multiple times to mitigate the effect of simulation artifacts.

7.6 Integration with Docker

The Docker environment was designed to ensure reproducibility, but integration was not trivial:

- Installing GPU drivers for accelerated rendering inside Docker required additional configuration.
- Networking issues occasionally prevented ROS nodes from communicating across containers.

Solution: A standard Docker Compose configuration was adopted, and environment variables were explicitly set in the `entrypoint.sh` script. This stabilized the simulation environment.

7.7 Summary of Challenges

The main difficulties encountered can be grouped as follows:

- **Perception:** sensitivity of ArUco detection to angle, distance, and lighting.
- **Control:** challenges in tuning PID gains for stable descent.
- **System Integration:** synchronization issues between ROS nodes, simulation artifacts in Gazebo, and Docker configuration.

Despite these challenges, robust solutions were implemented, allowing the system to achieve a high success rate in landing experiments. However, these problems also highlight the limitations of the current approach and motivate the improvements discussed in the next chapter.

Chapter 8

Discussion and Perspectives

The experiments carried out in this project demonstrate that ArUco-based vision provides a viable solution for autonomous landing of quadrotors on moving platforms. Nevertheless, the work also revealed limitations and opportunities for future improvements. This chapter discusses these aspects and outlines perspectives for extending the system.

8.1 Strengths of the Current Approach

The main strengths observed during this work are:

- **Modularity:** The ROS-based design ensures that perception, decision-making, and control are independent modules. This makes it straightforward to replace or upgrade components.
- **Reproducibility:** Using Docker containers and Gazebo simulation ensures that experiments can be reproduced consistently across different machines.
- **Simplicity and robustness:** ArUco markers provide reliable pose estimation at low computational cost, enabling real-time performance.
- **Successful integration:** Despite challenges, the pipeline achieves a high rate of successful landings in simulation.

8.2 Limitations

Several limitations were identified:

- **Marker dependency:** The approach requires a physical marker on the landing platform, which may not be feasible in real-world scenarios (e.g., unprepared environments).
- **Sensitivity:** ArUco detection performance decreases at long distances, oblique angles, or under poor lighting conditions.
- **Controller simplicity:** The PID controller is effective but limited. It requires manual tuning and does not account for platform dynamics explicitly.
- **Simulation gap:** Gazebo simulation, while useful, cannot fully replicate real-world disturbances such as wind gusts, lighting variability, or hardware latency.

8.3 Potential Improvements

Future research could address these limitations through several enhancements:

1. **Advanced perception methods:**
   - Use of AprilTags or more robust fiducial marker systems.
   - Deployment of deep-learning-based object detection (e.g., YOLO or Faster R-CNN) to identify natural features of the platform without markers.
   - Sensor fusion with inertial measurements to stabilize pose estimation.
2. **Improved control strategies:**
   - Replacing PID with Model Predictive Control (MPC), which can optimize over a prediction horizon and account for platform dynamics.
   - Adaptive control strategies that adjust gains automatically based on conditions.
3. **Robust decision-making:**
   - Extending the FSM with additional states for recovery after detection loss.
   - Replacing the FSM with reinforcement learning (RL) policies trained in simulation, allowing more flexible and adaptive behaviors.
4. **Real-world implementation:**
   - Deploying the system on an actual quadrotor platform with onboard cameras.
   - Performing outdoor experiments with real moving platforms.
   - Testing robustness under varying weather conditions and environments.

8.4 Scientific Impact

From a scientific perspective, this project contributes to ongoing research in autonomous robotics:

- It demonstrates how open-source tools (ROS, Gazebo, OpenCV) can be integrated to solve complex problems in robotics.
- It highlights the importance of perception-action coupling in real-time systems.
- It provides a reproducible testbed for experimenting with landing algorithms, which can serve as a baseline for more advanced approaches.

8.5 Future Directions

Several concrete directions can be pursued:

- Combining visual landing with vision-based navigation for complete autonomous missions.

- Exploring multi-quadrotor coordination, where multiple drones land on the same moving platform.
- Extending the framework to other robotic domains, such as autonomous ground vehicles docking to charging stations.

8.6 Conclusion of Discussion

The system developed in this project successfully achieves autonomous landing of quadrotors on moving platforms in simulation. While effective, its limitations motivate further work towards more robust, scalable, and real-world-ready solutions. ArUco-based vision provides an excellent starting point, but the future lies in combining advanced perception and control methods for reliable deployment in unconstrained environments.

Chapter 9

Conclusion

This project presented the design and implementation of an autonomous landing system for quadrotors on moving platforms using ArUco markers as the primary perception modality. The complete pipeline was developed within the ROS framework and tested in the Gazebo simulator, ensuring reproducibility and modularity.

The contributions of this work can be summarized as follows:

- Development of a vision-based perception module using ArUco markers for relative pose estimation.
- Implementation of a finite state machine (FSM) to structure quadrotor behavior during landing.
- Integration of a PID-based landing controller to regulate descent and alignment.
- Validation of the complete system in simulation, including robustness tests with disturbances and moving platforms.

The results demonstrate that ArUco markers provide a simple yet powerful solution for autonomous landing. The quadrotor was able to detect markers at medium distances, align with the platform, and successfully land in most scenarios. However, several challenges remain, including the sensitivity of detection to viewing conditions, the limitations of PID control, and the inherent gap between simulation and real-world deployment.

Looking forward, the system can be improved by integrating more advanced perception algorithms, adaptive or model predictive control, and reinforcement learning for decision-making. The transition from simulation to real-world experiments represents the ultimate test, and future work should focus on bridging this gap.

In conclusion, this project provides both a practical framework and a scientific contribution to the field of autonomous quadrotor landing. It highlights the feasibility of marker-based solutions while pointing toward the next steps required for robust real-world applications.