TRAJECTORY TRACKING CONTROL OF A QUADROTOR DURING COOPERATIVE OPERATION BETWEEN UAV AND UGV

(Revised Version)

by

AJIBOLA FOWOWE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

JULY 2016

Acknowledgements

First of all, I am grateful to **The Almighty God** for establishing me to complete this project.

I wish to express my sincere thanks to my supervisor, Prof. Dr. Frank Lewis, for his unwavering support, guidance and insight throughout the research process. I would also like to remark the great motivation he has given to me throughout the entire process and for providing me with all the necessary facilities for completing this project.

My sincere gratitude to Dr. Aditya Das. I am grateful and indebted to him for his expert, sincere and valuable guidance on the subject of Robotics.

I am very grateful to my friend and Lab partner Arifuddin Mohammed for the invaluable support, especially during the period I was away, for the many late evenings spent in the lab and the online meetings we had to have to complete this project.

My gratitude to Hans-Arne Driescher for guiding me during the revision of my thesis report. He was patient with me throughout the process and pointed me in the right path. Thank you Arne for all your feedbacks.

And finally, I would like to thank all my close friends and family. You have all encouraged and believed in me throughout my graduate program. These are the people, whom by their words of encouragement, allow me to go on.

May 15, 2016

Abstract

TRAJECTORY TRACKING CONTROL OF A QUADROTOR DURING COOPERATIVE OPERATION BETWEEN UAV AND UGV

Ajibola Fowowe, MS
The University of Texas at Arlington, 2016

Supervising Professor: Dr. Frank Lewis

In recent times, there has been an increase in the use and application of unmanned aerial vehicles (UAVs). UAVs are used for various operations ranging from military applications to civilian purposes such as traffic monitoring, photography, surveillance and others. Within the UAV family, rotorcrafts take precedence over the fixed wing aircraft especially because of their unique features such as vertical takeoff and landing, increased payload, high maneuverability and more. They can be used to perform dangerous tasks and access environments that pose danger to man such as observatory wood and building fire, military purposes etc.

This project focused on one of the various applications of the UAVs; cooperative operations between UAVs and ground vehicles. Scenarios may arise where we need the UAV to take on independent surveillance task in an unknown environment. I present a solution for the cooperative operation between UAVs and ground vehicles in unknown environment using visual navigation and onboard sensors of a small-scale, low-cost quadrotor in unknown environments.

My approach relies on a monocular camera and onboard sensors as the main drivers and therefore does not need external tracking aids like GPS. Computations are carried out on an external laptop that communicates over wireless LAN with the quadrotor using Robot Operating System (ROS). This approach consists of two major components: a monocular SLAM system for pose estimation and an autonomous landing system using a PID controller to control the position and orientation of the drone.

During the cooperative operation, the ground vehicle acts as the master, a mobile launch/landing pad for the UAV. Cooperative operation between mobile robots can be very critical in complex scenarios such as the UAV mapping out an area for obstacles and optimal navigation path for the UGV, also the UAV can be sent to areas inaccessible by the ground vehicle for observation and it returns back to the ground vehicle after this operation.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1

Introduction

This project describes work that has been done in the context of the R & D research project of Advanced Controls and Sensors group (ACS), which is ongoing in the Autonomous Systems Laboratory (ASL) of the University of Texas at Arlington Research Institute (UTARI) at the University of Texas at Arlington, in USA.

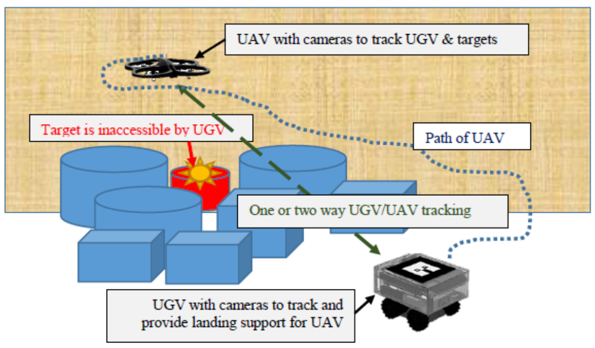

UAV with cameras to track UGV & targets
Target is inaccessible by UGV
Path of UAV
One or two way UGV/UAV tracking
UGV with cameras to track and provide landing support for UAV

Figure 1-1 Cooperative Operation between UAV and UGV

In recent years, research interest in autonomous micro-aerial vehicles (MAVS) has rapidly grown. Significant progress has been made, recent examples include aggressive flight maneuvers [1, 2], ping-pong [3] and collaborative construction tasks [4]. However, all of these systems require some form of external motion capture systems. Flying in unknown, GPS-denied environments is still an open research problem. The key challenges here are to localize the robot purely from its own sensor data and to robustly navigate it even under potential sensor loss. This requires both a solution to the so-called simultaneous localization and mapping (SLAM) problem as well as robust state estimation and control methods. These challenges are even more expressed on low-cost hardware with inaccurate actuators, noisy sensors, significant delays and limited onboard computation resources.

For solving the SLAM problem on MAVs, different types of sensors such as laser range scanners [5], monocular cameras [6, 7], stereo cameras [8] and RGB-D sensors [9] have been explored in the past. In my point of view, monocular cameras provide two major advantages above other modalities: (1) the amount of information that can be acquired is immense compared to their low weight, power consumption, size and cost, which are unmatched by any other type of sensor and (2) in contrast to depth measuring devices, the range of a monocular camera is virtually unlimited –allowing a monocular SLAM system to operate both in small, confined and large open environments. The drawback however is, that the scale of the environment cannot be determined from monocular vision alone, additional sensors such as an IMU are required.

In this project, I present a strategy where the quadcopter UAV can work cooperatively with the ground vehicle. During joint missions, the UAV can be sent on visual operations to areas inaccessible by the ground vehicle and can safely return to the ground vehicle after mission completion. In this case, the operator would need to only control the ground vehicle while the UAV performs aerial operations. After the completion of the aerial operation, a landing command is sent to the UAV and it autonomously locates the ground vehicle and lands on it. For this project, a visual marker is used in determining the location and bearing of the UAV relative to the ground vehicle.

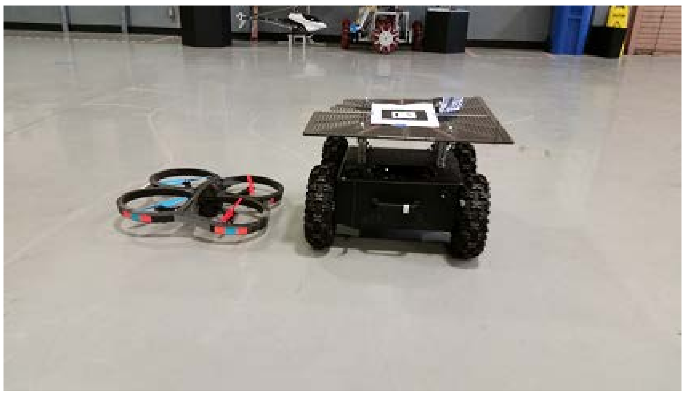


Figure 1-2 AR.Drone and Dr. Robot Jaguar Mobile Platform

Cooperative operations between UAVs and ground vehicles is a rapidly growing area of research with major breakthroughs in computer vision. The cooperation between flying vehicles and robotic ground platforms is rapidly spreading as performing tools to be used for data gathering, search and rescue operations, civil protection and safety issues [10]. Computer vision plays a very important role in this as some form of visual information is needed for all of these operations. Computer vision is much more than a technique to sense and recover environment information from an UAV, it should play a main role regarding UAVs' functionality because of the big amount of information that can be extracted [11].

During navigation, UAV onboard sensors can be used to feedback critical information about the state of the UAV. This however does not answer critical information about the position of the UAV. In GPS enabled areas, the position of the UAV can be easily determined using a GPS based navigation system. My project focuses on areas without GPS access, the development of a computer vision based navigation system that allows an UAV to navigate unknown environment and towards targets of interest such as the landing pad. This project also investigates autonomous landing techniques for a stationary landing base.

Currently, some applications have been developed to help determine the location of an UAV relative to a stationary point. Radio direction finding (RDF) is the measurement of the direction from which a received signal was transmitted [12]. RDFs have evolved over time, following the creation of smarter electronics. The need for big antennas that measured signal strengths has been pushed aside. Using a custom-built Radio Direction Finder, readings from a known transmitter was used to compare various Bayesian reasoning-based filtering algorithms for tracking and locating Mini-UAVs [13].

A piece that inspired this project was the work done by Jakob Engel, using a monocular, keyframe-based simultaneous localization and mapping (SLAM) system for pose estimation [14]. This enabled him to produce accurate position estimates using only onboard sensors. Assuming the ground vehicle is mobile and has a series of tasks to perform that require it moving from point to point, we will need the UAV to be mobile as well, following the UGV to its different locations. It is therefore imminent we design a mobile landing pad for take-off and landing on the UGV. See figure 1-2 for the structural modifications made to the UGV to accommodate a landing pad. Adding to Engel’s work [14], an autonomous landing system for cooperative operation has been incorporated.

In this project, a marker is placed on the ground vehicle to help the UAV identify the position and orientation of the landing pad. The onboard cameras on the low-cost AR.Drone are used for visual navigation as well as image collection. The images collected from the onboard cameras are processed and used to estimate the pose of the marker on the ground vehicle during landing.

The information derived from the marker detection system in conjunction with the onboard sensors is used to plan a landing profile for the UAV. Here, I used two possible approaches for guiding the UAV from an initial point to the landing pad on the UGV. The first, the UAV is guided from an initial point in space while maintaining its altitude to a predetermined point directly above the ground vehicle where the UAV is properly aligned with the landing pad in a straight line in the absence of any obstacle. The altitude is then reduced using a controller till the UAV ends up on the UGV. The bottom camera on the UAV is used to maintain visual contact with the marker on the UGV.

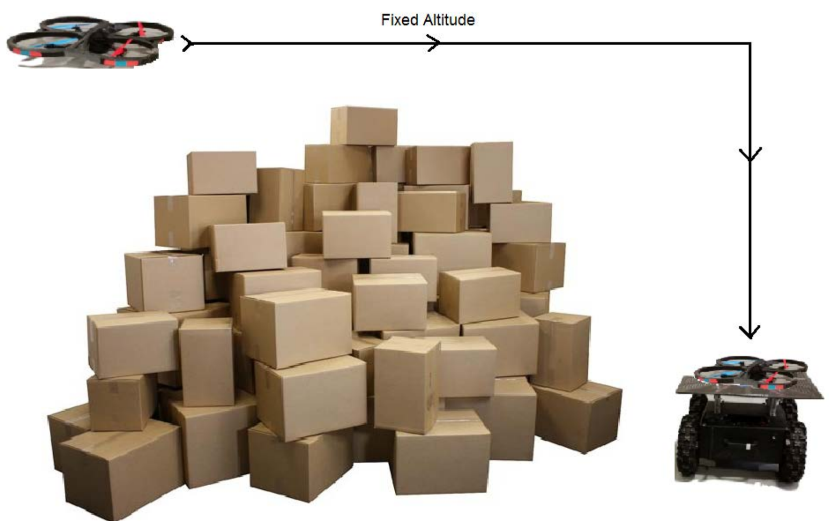


Figure 1-3 First Landing Approach

The second, in the absence of any obstacle, the UAV is guided through a perpendicular straight line from an initial point to the landing pad. This approach involves travelling through the shortest possible distance from the initial point to the landing pad on the UGV.

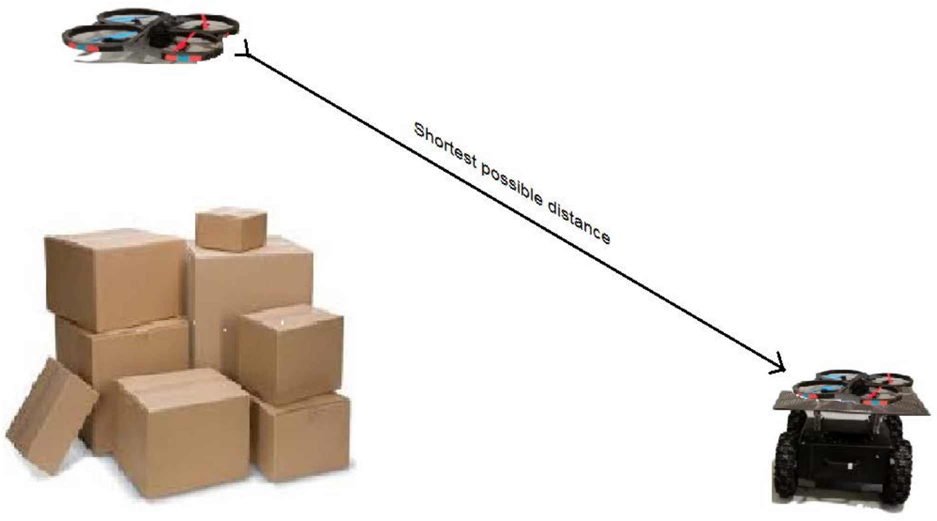


Figure 1-4 Second Landing Approach

Outline of Report

This report consists of the following chapters. Below is a small description of these chapters.

- **Introduction:** This chapter describes the overview of this master thesis. It all includes related work and the goal of the thesis.

- **Quadrotor:** This chapter discusses the general dynamics of a quadrotor. It introduces the AR.Drone and its specifications. It also introduces the software application ROS used for this thesis.
- **Simultaneous Localization and Mapping:** This chapter introduces the monocular SLAM and the work done by [14].
- **Autonomous Landing Approach:** This chapter discusses the marker recognition algorithm and the landing approach used.
- **Control:** This chapter introduces the control algorithm used in getting the AR.Drone from an initial point to the goal position during navigation and landing.
- **Implementation:** This chapter explains how this thesis was executed and how the different algorithms were integrated together.
- **Results:** This chapter describes the results of the work done at UTARI.
- **Conclusion:** Describes a conclusion on the thesis work
- **Future Work:** Discusses areas of improvement and possible future work.

Chapter 2

Quadrotor

For this project, I used the Parrot AR.Drone for implementation. The Parrot AR.Drone was revealed at the international CES 2010 in Las Vegas along with demonstration of the iOS applications used to control it [15]. The Parrot AR.Drone was designed to be controlled using a smart device. The Parrot AR.Drone is built using very light and robust materials. The main structure is made of carbon-fiber tubes and fiber-reinforced PA66 plastic parts. The hull is injected with expanded polypropylene EPP [16]. EPP is an engineered plastic foam material and this material allows the drone to survive crashes. It has two hulls, one designed for indoor application and the other for outdoor applications. It has dimensions of 52.5cm x 51.5cm with the hull and 45cm x 29cm without the hull. The drone itself weighs 360g without hull and 400g with the hull.

The Parrot AR.Drone has high efficiency propellers powered by 4 brushless motors with 28,500 RPM [31]. There is a Lithium polymer battery onboard the drone with a rated voltage of 11.1V. The rotors safety system automatically locks the propellers in the event of any contact with them, making it safe for lab and indoor use.

The drone is equipped with two cameras (one directed forward and one directed downward), an ultrasound altimeter, a 3-axis accelerometer, a 2-axis gyroscope (measuring pitch and roll angle) and a 1-axis yaw precision gyroscope [16]. The onboard controller is composed of a 32bits ARM processor with 128 Mb DDR Ram, on which a BusyBox composed GNU/Linux distribution is running.

The Parrot AR.Drone is often used in research institutions because of its low cost and high maneuverability. The AR.Drone has six degrees of freedom, with a miniaturized inertial measurement unit tracking the pitch, roll and yaw for use in stabilization [16].

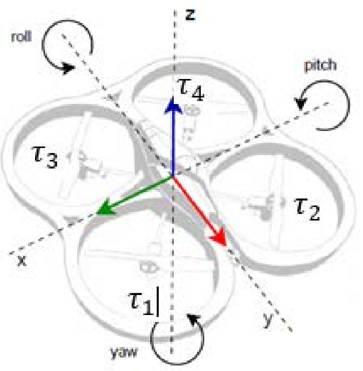


Figure 2-1 Schematics of the Parrot AR.Drone

2.1 Dynamics of a Quadrotor

A quadrotor helicopter (quadcopter) is a helicopter which has four equally spaced rotors, usually arranged at the corners of a square body [17]. These four rotors are controlled independently using electronic assistance to provide the maneuverability desired. Brushless motors are usually used for quadcopter application. Quadcopters have six degrees of freedom (DOF) which includes three translational and three rotational DOFs.

$$X = \begin{bmatrix} x \\ y \\ z \\ \emptyset \\ \theta \\ \varphi \end{bmatrix} \begin{array}{l} \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \text{Position Navigational States} \\ \left.\begin{array}{l} \text{Roll} \\ \text{Pitch} \\ \text{Yaw} \end{array}\right\} \text{Angular position attitudes} \end{array} \tag{2.1}$$

These six degrees of freedom are controlled by four independent inputs making the quadcopter under-actuated and a nonlinear system. Having defined the position and velocity of the quadrotor in the inertial frame as $(x, y, z)^T$ and $(\dot{x}, \dot{y}, \dot{z})^T$. Likewise, the rotational angles defined as $(\phi, \theta, \varphi)^T$ and the rotational velocities as $(\dot{\phi}, \dot{\theta}, \dot{\psi})^T$. To convert these angular velocities into the angular velocity vector w in the body frame, we use the following relation:

$$w = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \dot{\theta} \tag{2.2}$$

Using Euler angle conventions, we can relate the rotation matrix R from the body frame to the inertial frame.

$$R = \begin{bmatrix} c_\theta c_\varphi & -c_\emptyset s_\varphi + s_\emptyset s_\theta c_\varphi & s_\emptyset s_\varphi + c_\emptyset s_\theta c_\varphi \\ c_\theta s_\varphi & c_\emptyset c_\varphi + s_\emptyset s_\theta s_\varphi & -s_\emptyset c_\varphi + c_\emptyset s_\theta s_\varphi \\ -s_\theta & s_\emptyset c_\theta & c_\emptyset c_\theta \end{bmatrix} \tag{2.3}$$

Each rotor contributes some torque about the z axis, this torque keeps the propeller spinning and provides thrust. Torque is proportional to the thrust τ by a constant ratio determined by the blade configuration and parameters. The quadcopter is controlled by independently varying the speed of the four rotors [17]. The inputs u to the quadcopter:

- Vertical Acceleration: $u_1 = \uparrow \tau_1 \uparrow \tau_2 \uparrow \tau_3 \uparrow \tau_4$ where two opposite ones rotate clockwise and the other two counter-clockwise. This cancels out their respective torques.
- Rolling or Pitch Rotation: u_2 = $= \uparrow \tau_1 \; or \uparrow \tau_2 \; or \uparrow \tau_3 \; or \uparrow \tau_4$ while the thrust of the opposite rotor is decreased.

- Yawing Movement: $u_3 = \uparrow \tau_1 \uparrow \tau_4 + \downarrow \tau_2 \downarrow \tau_3$ or $\downarrow \tau_1 \downarrow \tau_4 + \uparrow \tau_2 \uparrow \tau_3$ depending on the direction of yaw.

  where u is the input and τ_i are the thrust from the ith rotor.

2.2 Parrot AR.Drone

Cameras

The AR.Drone has two on-board cameras, a front camera pointing forward and a vertical camera pointing downward. The front camera has a $93°$ wide-angle diagonal lens camera with a video frequency of 15 fps and a resolution of 640 x 480 pixels, covering a field of view of 73.5° x 58.5°. This camera is equipped for 1D tag detection

Due to the used fish eye lens, the image is subject to significant radial distortion. Furthermore rapid drone movements produce strong motion blur, as well as linear distortion due to the camera's rolling shutter.

Gyroscopes and Altimeter

For altitude measurement, an ultrasound based altimeter capable of measuring up to 6m is installed on the drone. A proprietary filter on board of the AR.Drone converts the angular velocities to an estimated attitude (orientation). The measured roll and pitch angles are, with a deviation of only up to 0.5°. The yaw measurements however drift significantly over time (with up to 60° per minute, differing from drone to drone – much lower values have also been reported). The AR.Drone also sends update on its body accelerations and body velocity. These estimates are based on the inertia measurements, aerodynamic model and visual odometry obtained from the relative motion between the camera frames.

Software

The Parrot AR.Drone comes with all software required to fly the quadrotor. Due to the drone being a commercial product which is primarily sold as high-tech toy and not as a tool for research.

UDP – User Datagram Protocol

Communication with the drone happens over the User Datagram Protocol UDP. This is one of the dominant transport-layer protocols in use. The UDP is a simple model protocol, it allows you send datagrams in other words packets from one machine to another. These are received at the other end as the same packets. UDP binds a combination of an IP address and a service port on both ends and as such establishes host-to-host communication. The AR.Drone communication interface is built of three UDP ports:

- Navigation Data Port = 5554: This port is used to retrieve data back from the drone such as its position, speed, engine rotation speed etc. This navigation data port also include data detection information that can be used to create augmented reality. Data is sent approximately 30 times per second.
- On-Board video Port = 5555: A video stream is sent by the AR.Drone to the client device via this port.
- AT Command Port = 5556: This port is used in sending commands to the drone.

Another communication channel called Control Port can be established on TCP port 5559 to transfer critical data, by opposition to the other data that can be lost with no dangerous effect. It is used to retrieve configuration data and to acknowledge important information such as the sending of configuration information [19].

2.3 ROS

Robot Operating System ROS is the robot frame work application used for this project. ROS is an open-source, meta-operating system for your robot. It provides services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management [20]. The main ROS client libraries are C++, Python and LISP. These are geared toward a UNIX-like system, primarily because of their dependence on large collections of open-source software dependencies.

The robotics research community has a collection of good algorithms for common tasks such as navigation, motion planning, mapping and many others. These existing algorithms are useful as they help avoid the need to re-implement each algorithm for new systems. ROS standard packages provide stable, debugged implementations of many important robotics algorithms. ROS is designed to allow small, mostly independent programs called Nodes that all run at the same time and can communicate with one another.

Communication between the nodes is done by sending Messages and these Messages are organized into named Topics. The logic here is that a node that wants to share information will publish messages on the appropriate topic or topics. A node that wants to receive information will subscribe to the topic or topics that it's interested in. [21] has detailed key points for ROS familiarization.

The hardware in my project:

- AR.Drone 2.0
- Onground computation station (Laptop with Ubuntu 14.04 and ROS indigo).

The Software Used:

- ROS Indigo
- Ubuntu 14.04

Existing ROS packages used:

- Ardrone_autonomy for AR.Drone 2.0
- Tum_ardrone for PID controller, EKF and odometry estimation [14]
- Ar_track_alvar for Marker Recognition [24]

Chapter 3

Simultaneous Localization and Mapping

Navigating in previously unknown areas is an ongoing area of research in the robotics community. In areas without GPS access, how do we determine the location of a robot? This is a fundamental question asked in robot navigation. For this project, how do we determine the exact location of the AR.Drone since it does not have GPS capabilities or how do we determine how far it has travelled from a reference point. Simultaneous Localization and Mapping (SLAM) is the ability to simultaneously localize a robot and accurately map its environment, this is considered as the prerequisite for autonomous robots. In this project, the AR.Drone's sensor data is used to both recover its flight path and build a map of its environment.

The solution to Localization can be divided into two areas of time constraint: online computing and offline computing. Online computing involves the ability to use the response as input for decision-making, this is used by autonomous robots. There is a time constraint involved and a limited amount of computations can be performed. Offline computing is more time accommodating, allowing more complex computations. The major setback of offline computing is decision-making. Decisions can be made on the go, the robot has to wait to make a decision.

We have seen great advances in real-time 3D vision in recent years, enabled by continuous algorithmic improvements, the continuing increase in commodity processing power and better camera technology. This has given rise to visual navigation. High definition images can be extracted and processed to build SLAM systems. For this project, the images from the video stream along with the data from the onboard sensors are used to re-estimate the position of the robot at regular intervals. Monocular SLAM where a single agile camera moves through a mostly static scene was for a long time focused on mapping only, now, efforts is geared towards improving the quality of the scene reconstruction which can be achieved in real-time.

The hardware needed to implement the monocular is very simple, giving it an advantage over other SLAM methods. The disadvantage however is that the algorithms involved is more complex and so the software is complicated. This is essentially because the depth of an image cannot be captured from the image but has to be recalculated through analyzing images in the chain of frames in the video. Monocular SLAM calculates depth through comparing images captured at different times and at overall positions of the robot system.

3.2 Other SLAM Techniques

Recreating 3D from 2D images is a known problem in the area of computer vision that has attracted immense attention in mobile robotics. Several techniques have been developed to combat this problem, out of which I will be introducing the work of [14] used in this project for autonomous navigation.

Several Augmented Reality (AR) systems operate on prior knowledge of the user's environment but for our application, we will explore tracking and mapping using a calibrated camera in a previously unknown scene without any known objects or initialization target, while building a map of the environment.

FastSLAM is a factored solution to the SLAM problem, it uses an algorithm that decomposes the SLAM problem into a robot localization problem and a collection of landmark estimation problems that are conditioned on the robot pose estimate [22]. It uses a modified particle filter similar to the Monte Carlo localization (MCL) algorithm for estimating the posterior over robots paths. The FastSLAM algorithm implements the update equation using Extended Kalman Filter (EKF). FastSLAM's EKF is similar to the traditional EKF for SLAM in that it approximates the measurement model using a linear Gaussian function [22].

A key limitation of EKF-based approaches is their computational complexity of incorporating an observation. Sensor updates require time quadratic in the number of landmarks to compute. This complexity stems from the fact that the covariance matrix maintained by the Kalman filters has elements which must be updated even if just a single landmark is observed. The quadratic complexity limits the number of landmarks that can be handled by this approach to only a few hundred. Parallel Tracking and Mapping (PTAM) is a Keyframe-based approach that differs from the filtering based approach used by the FastSLAM.

Instead of marginalizing out previous poses and summarizing all information within a probability distribution (filtering), keyframe-based approaches retains selected subset of previous observations - called keyframes - explicitly representing past knowledge gained. Parallel Tracking and Mapping (PTAM) is a simple and low-entry-level solution to the SLAM problem. It uses a camera tracking system for augmented reality. It requires no markers, existing models of the environment or inertial sensors. PTAM uses FAST corners that are cheap to detect and process. Tracking and Mapping run in two parallel threads, making tracking no longer probabilistically-dependent on the map creation procedure. With this, there are no limitations on the type or robustness of the tracking used. The limitation of PTAM is the need to perform stereo initialization in a unique way [32].

3.3 Monocular, Keyframe-Based SLAM

The Extended Kalman Filter Based Monocular SLAM was discussed by Gu Zhaopeng in [34]. His work involved both a vision only system and for systems where some kind of kinematic sensor capture such as an IMU (Inertial Measurement Unit) is used. For most vision based systems, the camera state is comprised of position, rotation quaternion, velocity and angular velocity of the camera, while the feature state is represented by 3D world coordinates. He showed that if some kind of kinematic sensor such as IMU, the accelerate motion model is always the best way.

For this project, we used the Keyframe-Based SLAM for localization and mapping of the AR.Drone during navigation. This approach was introduced by Engel is [14]. This SLAM process has two independently running loop: tracking and mapping. An existing map is required as an initialization point from which these two independent loops are built. The Keyframe-Based SLAM for localization and mapping can therefore be divided into three main parts: Initialization, mapping and tracking.

This approach involves the use of Keypoints which are small unique distinguishable points in the images captured by the AR.Drone camera. Processing all the images from the AR.Drone was computationally impossible, leading to the idea of using Keypoints. The Keypoints are identified by using the FAST corner detector.

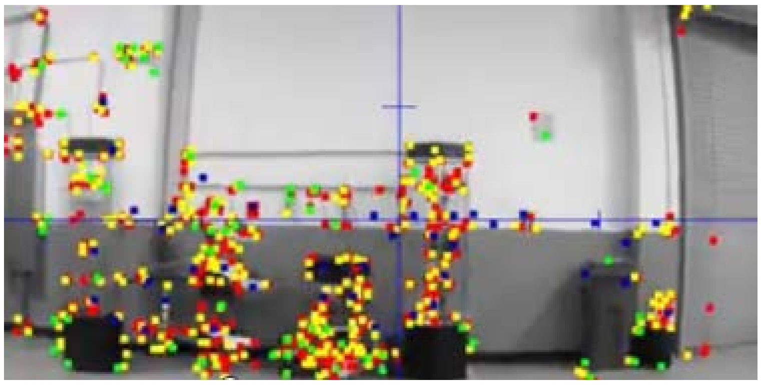


Figure 3-4 Locations of keypoints used by PTAM for tracking

Chapter 4

Autonomous Landing Approach

This chapter presents the autonomous landing approach and discusses the issues related to integrating it with the camera-based Monocular SLAM system. Recall, there are two main modes of operation for the AR.Drone in this thesis; first, the autonomous navigation and second, the autonomous landing. Autonomous navigation using monocular SLAM was introduced in chapter 3. The operator is fully in control of switching between these modes. The mode switching is done by entering a landing command. The goal here is to make sure that each task is completed before a mode switch is initiated.

The algorithm for the autonomous landing can be divided into two main sequences; the marker recognition and then the Final Landing approach. To get the AR.Drone to its goal position, we first need to determine its exact position with respect to the goal position. This information is then fed into a controller that minimizes the position difference by controlling the drone's actions.

For this thesis, an Open CV computer vision library [24] is used to recognize the goal position which is the landing pad. There are two visual trackers used in this thesis: PTAM for autonomous navigation mode and ALVAR for autonomous landing mode. Due to this, there is a need to switch between both visual trackers when the operator switches between modes. This is done via dynamic reconfiguration support by the ROS application. Dynamic reconfiguration in ROS allows you to change the node parameters at any time without having to restart the node.

ALVAR is an excellent, high-performance, robust and easy to use library for tracking. The image processing algorithm was not developed in this thesis work. ARTag is a marker system for supporting augmented reality. An ARTag is placed on the landing pad and acts as a marker used by the AR.Drone for identifying and tracking the pose of the landing spot. Here, we used a bundle consisting of multiple markers. This allows for more stable tracking and pose estimates with varying heights. The Open CV program automatically calculates spatial relationships between markers in a bundle.

4.1 Marker Recognition

There are a variety of available Open CV implementations using square markers: ARToolKit, ARTag, BinARyID and ALVAR etc. The marker recognition used in this thesis provides adaptive thresholding to handle a variety of lighting conditions, optical flow based tracking for more stable pose estimation and an improved marker identification method that does not significantly slow down as the number of markers increases.

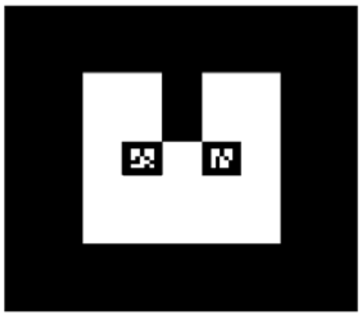


Figure 4-1 Bundle Marker

The bundle marker consists of multiple markers, this helps to correctly identify the markers at different heights. The larger marker can be easily identified at greater height and can be used for identification and pose estimation at a further distance while the smaller marker can be used to correctly determine the pose estimates at closer distances. A threshold is used to determine when new markers can be detected under uncertainty. To detect the marker for landing, both onboard cameras on the AR.Drone are used. The square markers allow both position and orientation to be extracted from the pattern of the marker.

For better marker detection, the image used for the marker should have a low level of detail and also should not be rotationally symmetric. Reflections are likely to impair marker detection so a non-reflective material should be used for producing the marker's image [25]. The performance of the image processing algorithm is affected by the ambient lighting conditions, therefore for the best results, a room with consistent lighting condition all over is used.

Marker Identification Process

Images captured by the onboard cameras are usually processed by mixing it to grayscale and thresholding it. For color images, each pixel is represented by three 8-bit channels for the colors red, green and blue which combined can represent $(2^8)^3$ = 16777216 colors. A grayscale image can be calculated from the RGB values in different ways depending on the use-case (e.g. adaption to human perception or computer vision) and the properties of the camera. In many cases the simple average (R+B+G)/3 can be used.

Segmentation detects large areas of connected pixels with the same color and labels these sets of pixels. Different information are extracted from each set, such as area, center of mass and color. A contour detection is executed to find the corners in the image. It identifies points where the difference in intensity is significant. Lines are fitted over the whole image to find any projective transformation of a square. Squares identified become the marker candidates [26].

The squares are then normalized to form patterns of squares. The thresholding operation is a function that generates binary images and compares to a library of feature vectors of known markers by correlation.

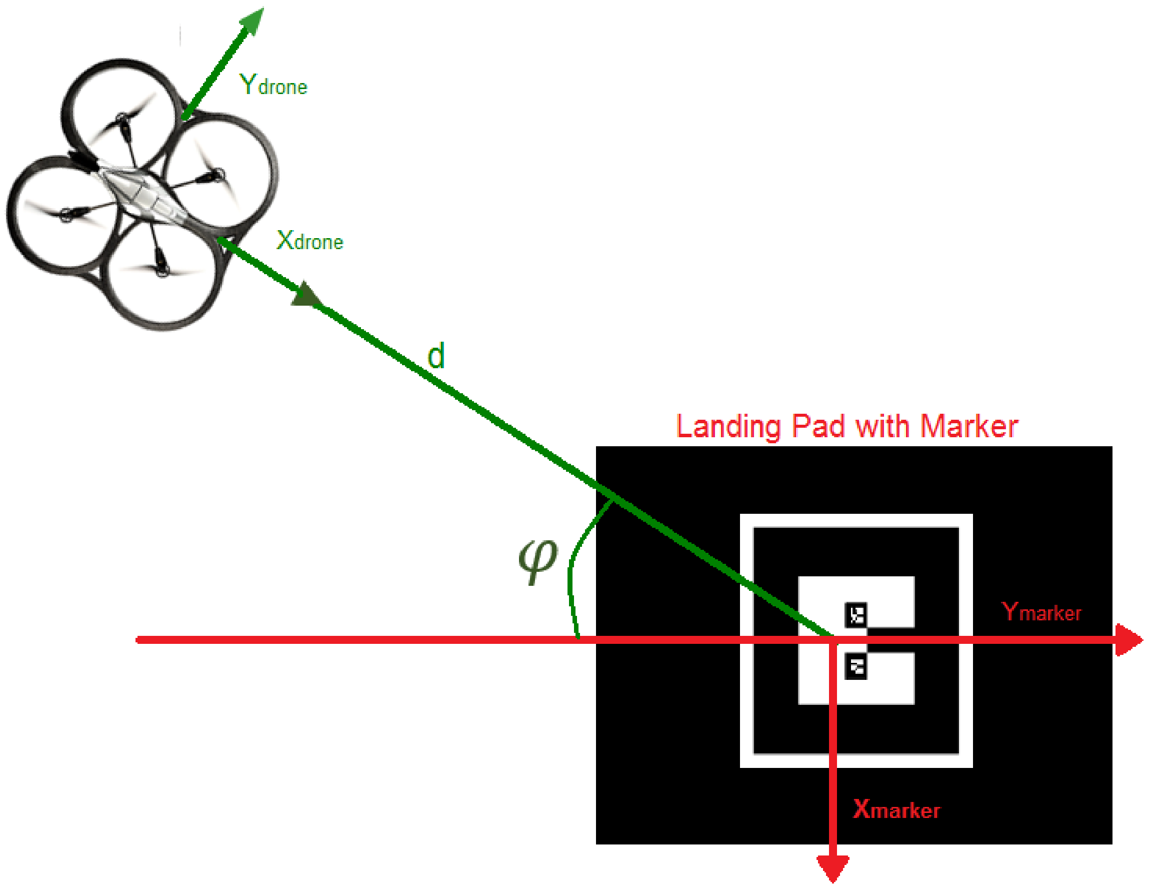


Figure 4-2 The Coordinate System

NOTE: The z-coordinate system was omitted from the figure above.

Once a best possible marker is selected, the markers position p_x, p_y, p_z, is determined along with its orientation described by a rotation matrix R.

$$[R| \quad p] = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & p_x \\ r_{2,1} & r_{2,2} & r_{2,3} & p_y \\ r_{3,1} & r_{3,2} & r_{3,3} & p_z \end{bmatrix} \tag{4.1}$$

R is expressed in terms of angles around the camera base vectors. Introducing φ as the rotation of the marker around the cameras z-axis, $\emptyset$ *and* θ represent the marker in the x and y respectively. Simplifying further with notations such as $s_x = \sin x$ *and* $c_x = \cos x$ for all $x = \in \ \varphi, \emptyset$ *and* θ.The rotation matrix R then becomes:

$$R = \begin{bmatrix} c_\theta c_\varphi & -c_\emptyset s_\varphi + s_\emptyset s_\theta c_\varphi & s_\emptyset s_\varphi + c_\emptyset s_\theta c_\varphi \\ c_\theta s_\varphi & c_\emptyset c_\varphi + s_\emptyset s_\theta s_\varphi & -s_\emptyset c_\varphi + c_\emptyset s_\theta s_\varphi \\ -s_\theta & s_\emptyset c_\theta & c_\emptyset c_\theta \end{bmatrix} \quad (4.2)$$

We are interested in the position information $p_x \, and \, p_y$ as well as the yaw angle φ. The pitch and roll angles can be derived and used to drive the AR.Drone to points in $\mathrm{x_{marker}}, \mathrm{y_{marker}}$ and $\mathrm{z_{marker}}$ With this, a new rotation matrix for the yaw angle which is not influenced by roll and pitch is derived.

$$\text{New rotation matrix } R_{new} = \begin{bmatrix} cos_\varphi & -sin_\varphi & 0 \\ sin_\varphi & cos_\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.3)$$

The yaw is a counterclockwise rotation of φ about the z-axis. The yaw angle is derived by calculating

$$\varphi = atan2(r_{21}, r_{11}) \quad (4.4)$$

For this project, the UGV is always steady with respect to the world. Therefore, I have chosen the UGV as the main coordinate system. The UGV is our goal point as we are trying to get the AR.Drone back on the UGV. Taking this goal point as:

$$Goal_{ref} = \begin{bmatrix} Goal_x \\ Goal_y \\ Goal_z \end{bmatrix} \quad (4.5)$$

Where $\mathrm{x_{marker}}, \mathrm{y_{marker}}$ and $\mathrm{z_{marker}}$ are the $Goal_x$, $Goal_y$ and $Goal_z$ respectively.

Pose Estimation using ALVAR

Using ALVAR, the ar_pose_marker topic publishes a pose object which contains the pose and orientation information. This pose information gives the point position in 3D coordinate of the marker while the orientation information is of quaternion type revealing four values x,y,z and w. The quaternion type is converted to roll, pitch and yaw using the function getRPY(). With this, the position and orientation of the marker is determined using the camera onboard the AR.Drone.

The AR.Drone coordinate system which is the center of the camera is taken as “Drone”. The marker is always steady with respect to the world, the coordinate system given by the marker is chosen as the main coordinate system. Assume the marker frame and the onboard camera frame are $\mathrm{P^A}$ and $\mathrm{P^C}$, then

$$\mathrm{P^A} = \mathrm{T_C^A P^C} \quad (4.6)$$

$$= (\mathrm{T_C^A})^{-1} \mathrm{P^C} \quad (4.7)$$

$$= T_A^C P^C \tag{4.8}$$

Where T_A^C and T_C^A represent respectively the transform from AR tag to camera frame and from the camera frame to AR tag frame. With the ALVAR, this has been done so we easily derive the position of the AR.Drone with respect to the marker.

The z coordinate point position outputted by ALVAR is used to control the altitude without any transformation. Note that we are landing the AR.Drone on the marker placed on the UGV, so a direct transformation of the z coordinate point position outputted by ALVAR works fine for altitude control. This will be taken as $Goal_z$.

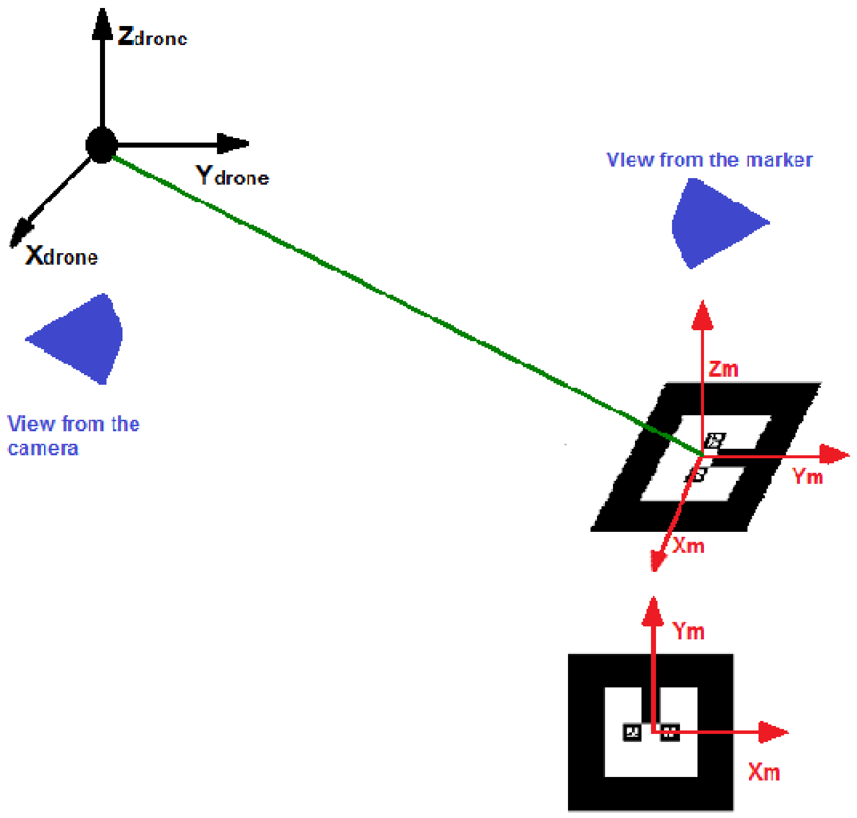


Figure 4-3 View Point of Coordinate Systems

Using figure 4-3 as an illustration, you would notice that both viewpoints share the same axis except in the y-axis coordinate. For ease of calculating the position error, I made the steady frame (marker frame) the reference frame. Initially, when the goal positions were established, they were established from the Camera's coordinate system. Since the UGV is always steady as compared to the moving UAV and we need to keep updating the UAV's position from the goal point. I rotated the error and changed the sign in the y-axis to express it in the drone's coordinate system. The error in the coordinate system becomes

$$\hat{e} = Goal_{ref} - \begin{bmatrix} Drone_x \\ Drone_y \\ Drone_z \end{bmatrix} = \begin{bmatrix} Goal_x \\ Goal_y \\ Goal_z \end{bmatrix} - \begin{bmatrix} Drone_x \\ Drone_y \\ Drone_z \end{bmatrix} \tag{4.9}$$

After being rotated becomes:

$$e = R_{new} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Goal_x \\ Goal_y \\ Goal_z \end{bmatrix} - \begin{bmatrix} Drone_x \\ Drone_y \\ Drone_z \end{bmatrix} \tag{4.10}$$

4.2 Finding the Marker

After the landing command has been entered by the operator, the AR.Drone will need to locate the marker on the UGV before a final landing trajectory is developed. A good approach would be to take the AR.Drone to its original take off point using the monocular SLAM but we must account for a possible change in the position of the UGV during the surveillance operation of the AR.Drone. We cannot assume the UGV would always remain at the same position throughout the AR.Drone flight. A search algorithm was developed to help the UAV locate the UGV.

Using specified spacing, a series of path corners are calculated based on a grid pattern within which way points are placed. Four corners and the center of the grid are identified and the AR.Drone flies to all five positions to execute a search operation. These positions are chosen to ensure the UAV can identify the marker from another position if an obstruction lies in the direct line of sight. At this specified positions, the AR.Drone completes a yaw rotation about the z-axis at a defined rotation rate to ensure a good image quality for marker detection is captured by the onboard cameras. The rotation operation at the initial corners is a $90°$ rotation while the rotation at the center and subsequent corners is a complete $360°$ rotation,see figure 4-4.

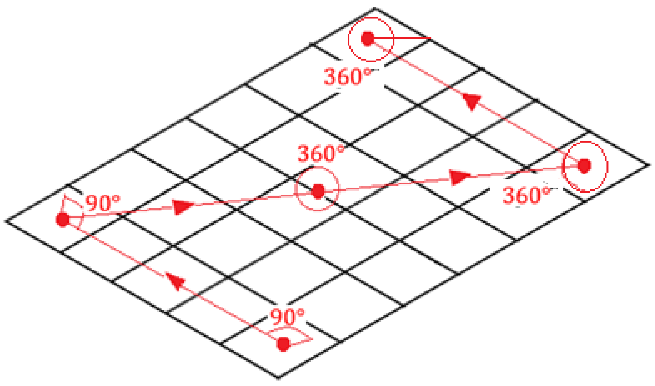


Figure 4-4 Search Pattern

The search path is developed by first generating a grid of rectangles. The summation of these rectangles give the breadth and width of the area captured by the camera onboard the

AR.Drone at a certain height. A challenge encountered was defining the area for the grid of rectangles. I was faced with the question of "How Far Can the Camera See?". The main elements considered were

- Object size: The first question here is, what the size of the object to be detected is. Then, I need to determine what level of detail to resolve about the object. For this project, I used Johnson Criteria which defines DRI (Detection Recognition Identification) using the number of pixels required on the object to make an accurate assessment.
- Camera field of view: The camera's field of view FOV affects how far the camera can see. FOV is the amount of given screen captured by the camera. The FOV is defined by three elements: Lens, sensor format within the camera and the camera zoom position in relation to the scene.
- Image resolution: The camera's image resolution defines the detail an image holds. A higher resolution means more image detail.

The only drawback with this approach is the time taken to turn about its yaw axis as turning is a time-consuming operation.

4.3 Pose Estimation

Estimating the position and orientation of an object in real time poses a major problem for vision-based pose estimation. Many of this vision-based pose estimation rely on an extended Kalman Filter (EKF). The state estimation of the AR.Drone is based on the Extended Kalman Filter (EKF). The EKF is an extension of the Linear Kalman Filter to nonlinear systems [26]. The AR.Drone has a variety of sensors and each sensor has some uncertainty in the measurement, the Kalman Filter is therefore needed to filter out the data and make a better estimation of the vehicle's state.

The EKF rely on known noise statistics, initial object pose and sufficiently high sampling rates for good approximation of measurement-function linearization.

Model of a Kalman Filter [27] is expressed as

$$X_k = A_k X_{k-1} + B_k U_k + w_k \tag{4.11}$$

Where

A_k is the state transition model which is applied to the previous state X_{k-1}

B_k is the control-input model which is applied to the control vector U_k

w_k is the process noise which is assumed to be drawn from a zero mean multivariate normal distribution with covariance Q_k

$w_k \sim N(0, Q_k)$

Then the measurement which can be used to derive the estimated state is made according to

$$Z_k = H_k X_k + v_k \quad (4.12)$$

Where

H_k is the observation model which translates the estimated state space into the observed space

v_k is the observation noise which is assumed to be zero mean Gaussian white noise with covariance R_k

$$v_k \sim N(0, R_k) \quad (4.13)$$

The Kalman filter predicts the process state from a previous state then the predicted state compensated by noise measurement will be fed back into the model to become the previous state for the next prediction.

Extended Kalman Filter

The Extended Kalman Filter solves the problem of trying to estimate the state by a linear process, it has the de facto standard in the theory of nonlinear state estimation. Tracking a rigid body using an EKF enables the use of a priori information on the measurement noise and type of motion to tune the filter. The pose estimates from the ALVAR are fused with data sent from the onboard sensors using an EKF. The EKF computes an estimate of the AR.Drone's pose, speed and a prediction of its future state. For a detailed breakdown, see [29].

The EKF process can be divided into two stages:

1. Time Update (Prediction) at k=0:

   Project the state ahead

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \quad (4.14)$$

   Project the error covariance ahead

$$P_k^- = AP_{k-1}A^T + Q \quad (4.15)$$

2. Measurement Update (Correction):

   Compute the Kalman Gain

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad (4.16)$$

   Update the estimate via z_k

$$\hat{x}_k = \hat{x}_k^- + K_k(Z_k - H\hat{x}_k^-) \tag{4.17}$$

Update the error covariance

$$P_k = (I - K_k H)P_k^- \tag{4.18}$$

The output at k will be the input at k+1

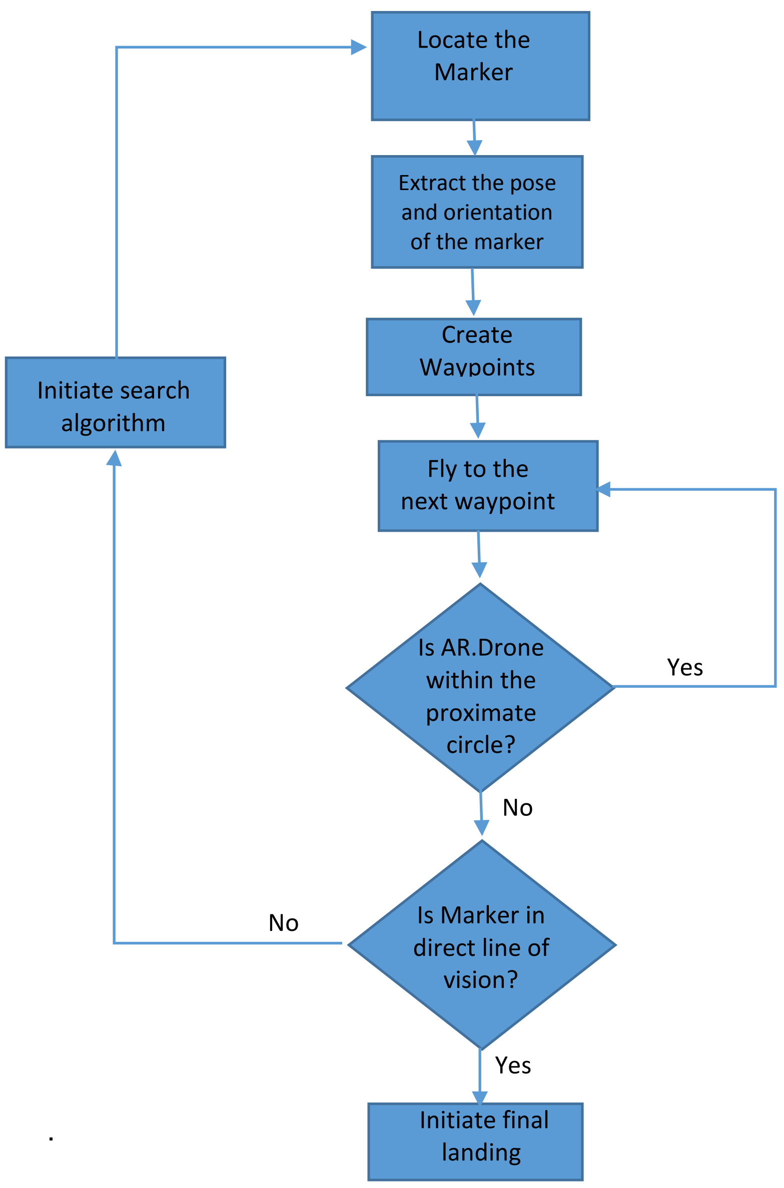
4.4 Path Planning

After the marker has been detected and the position and orientation of the marker has been determine which is our landing pad, the next step is navigating the AR.Drone to that point. I have devised two different landing approaches:

1. Navigating the AR.Drone through a fixed altitude to a point right above the UGV and then decreasing the altitude while maintaining its x and y position, see figure 1-3.
2. Navigating the drone directly to the UGV by simultaneously decreasing the altitude and orientating the drone towards the landing pad.

For both navigation methods, my approach involves a multi waypoint navigation method along the desired flight path from the AR.Drone's initial location and the landing pad. A waypoint is a set of coordinates that specify a specific point in physical space. For this flight path, we have determined it is obstacle free as the AR.Drone has direct line of sight with the marker, therefore making it a straight line path. No path finding or obstacle avoidance algorithm is needed for this. The waypoint navigation alongside the PID controller instructs the drone where to fly, at what height to fly at and what speed to fly at.

These waypoints lists the points for the AR.Drone to fly to in a defined order. The AR.Drone will hardly fly to its exact waypoint, therefore if it flies within the proximate circle, the program considers it as it meeting its goal. The waypoint navigation algorithm is shown below. The program continuously checks the distance between the current position of the AR.Drone and the next waypoint. If this distance is smaller than the radius of the proximate circle, then the AR.Drone flies to the next waypoint


Locate the Marker
Extract the pose and orientation of the marker
Create Waypoints
Fly to the next waypoint
Is AR.Drone within the proximate circle?
Yes
No
Is Marker in direct line of vision?
No
Yes
Initiate final landing
Initiate search algorithm

.

Figure 4-5 Waypoint Algorithm Flowchart

This approach would reduce the error per waypoint rather than have the error accumulate throughout flight path. The disadvantage with this approach is determining how fast or slow the AR.Drone should travel. For short distance waypoints, a small derivative gain is needed to prevent excessive jerky movement. The new goal position is updated at the waypoints and since the new distance to goal is small, a high derivative gain will cause an overshoot of the goal position.

For this thesis, we have used the same PID gain values irrespective of the distance between the waypoints. A future improvement will be selecting gain values based on the distances between the waypoints.

Chapter 5

Control

Control theory is an interdisciplinary branch of engineering and mathematics that deals with dynamic systems with inputs and how their behavior is modified using a feedback system. A dynamic system is a system whos behavior evolves with time. Typically these systems have input and output and our interest is to understand how the input affects the output or vice versa. In order to control the AR.Drone, which is a dynamic system, a feedback control is usually necessary. The dynamics for this AR.Drone is nonlinear and underactuated, making it a little difficult to design a controller for this kind of dynamic system.

This thesis is based on the AR.Drone following a defined trajectory to a desired x, y and z position. The trajectory is a desired position and orientation which is a function of time. The AR.Drone has only four control inputs but six degrees of freedom, three translational movements and three rotational movement. The AR.Drone can only move in these degrees of movement, therefore the AR.Drone is an underactuated system. For this thesis, we are controlling four of its degrees of freedom, which are: the vector position p and the yaw angle ψ. Our trajectory control problem can then be defined as designing a control input u such that the quadrotor vector position p and orientation ψ can asymptotically follow a given reference trajectory. Both $p - Goal_{ref}$ and $\psi - \psi_{ref}$ converges to zero as $t \to \infty$.

This chapter presents a proportional-integral-derivative controller (PID controller), the most common form of feedback control. The algorithm for a PID controller is described below

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t) \tag{5.1}$$

where u is the control input and e is the control error. For this thesis, the control error is the difference between the desired trajectory and the real trajectory of the AR.Drone. The control signal is the weighted sum of three separate control mechanisms:

- the proportional part which adjusts the output signal in direct proportion to the current value of the error e(t). The adjustable parameter is the proportional gain K_p. The error reduces with increasing gain but the possibility of oscillation will increase.
- the integral part corrects the offset $\int_0^t e(\tau) d\tau$, that may occur between the desired value and the system output automatically due to external disturbances over time [30]. The

steady state error disappears when the integral part is used for constant disturbances. The adjustable parameter is T_i where $K_i = \frac{K_p}{T_i}$

- the derivative part depends on the predicted future error, based on the derivative error with respect to time e(t). The adjustable parameter is T_d where $K_d = K_p \cdot T_d$

Usually, K_p, K_i and K_d are the tuned parameters and can be found by a variety of experimental or theoretical methods. In many cases it is sufficient to determine them experimentally by trial and error.

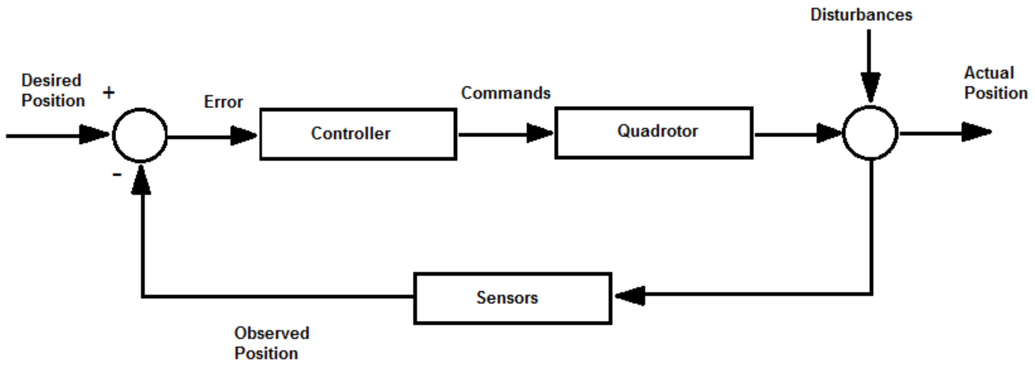


Figure 5-1 Schematic representation of a general control loop

The quality of a control system can be measured by the convergence time t_{conv}, measuring how long it takes until e(t) stays within a specified, small interval around zero. The effect of these three parts of a PID-controller is shown below [26].

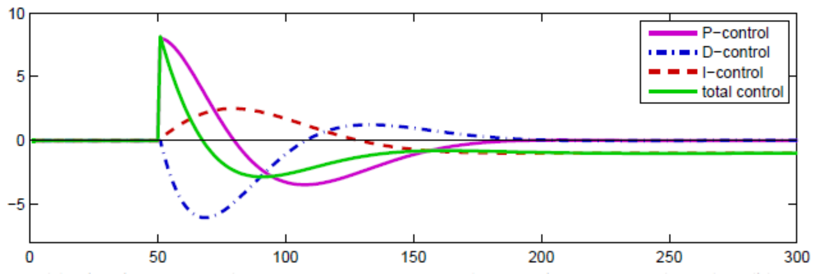


Figure 5-2 The three distinct control terms working together

It is more effective to break down full system models, reducing it such that control problems can be attacked specifically. For the AR.Drone, two major control problems are considered: altitude control and trajectory tracking. The objective here is to design a controller to track desired trajectories published by the Marker as its landing pad. This is achieved by designing four separate PID controllers controlling the altitude position, pitch, yaw and roll angles. The outputs of these controllers determine the tilt, vertical speed and angular speed of the AR.Drone. For more information on PID controllers, review [32]. The speed estimates from the Kalman filter is used for this. Let the predicted states of the drone be $(x, y, z, \dot{x}, \dot{y}, \dot{z}, \emptyset, \theta, \varphi, \dot{\varphi})^T$ and the coordinates of the landing pad as well as the orientation be $(\hat{x}, \hat{y}, \hat{z}, \hat{\varphi})$.

For this thesis, I have used the drone controller from [14] used in the navigation operation. The gain values for the PID controller were determined experimentally to better suit the waypoints navigation during landing.

Altitude Control

Altitude control is very important for the execution of this project, especially considering the different landing approaches discussed in chapter 4. The first landing approach method requires that the AR.Drone holds its altitude position while translating in the x-y plane. The ultrasound sensor on the AR.Drone is used to determine its altitude. Let the altitude measured by ultrasound be z_{ground}. The z_{ground} at the time the landing command was entered would become our z_{ref}. The error is then defined as

$$e_z = z - z_{ref} \tag{5.2}$$

A control input for the altitude is designed as

$$u_z = 0.4 \cdot e_z + 0.1 \cdot \dot{z} + 0.01 \int e_z \tag{5.3}$$

The second landing approach requires the AR.Drone to follow waypoints along the shortest possible path to the marker, see chapter 4. This requires a change in altitude as the AR.Drone descends toward the marker. The $z_{waypoint}$ is used to control the altitude of the AR.Drone along its landing trajectory. Refer to chapter 4 for more information on how the waypoints are generated and switched during landing. The error is defined as

$$e_z = z - z_{waypoint} \tag{5.4}$$

where z is the altitude of the drone.

As previously mentioned, the same altitude control input is used for this.

Yaw Control

The AR.Drone has an onboard control of the rotational velocity about the yaw-axis, based on information received from the onboard cameras and the inertial measurement unit. The rotation about the yaw axis is as previously mentioned (See chapter 4) is represented in the rotation matrix

$$R_{new} = \begin{bmatrix} cos_{\varphi} & -sin_{\varphi} & 0 \\ sin_{\varphi} & cos_{\varphi} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

where $-\pi < \varphi < \pi$.

The method for determining the yaw error is to assume the reference yaw angle $\varphi = 0$, therefore

$$e_{yaw} = \varphi_{ref} - \varphi = \varphi_{ref} \tag{5.6}$$

This way, the reference angle will always be half a rotation from the discontinuity and the risk of crossing it is minimal. The resulting control signal is

$$u_{yaw} = 0.02 \cdot e_{yaw} \tag{5.7}$$

x-y Tracking

We have determined the yaw and altitude control. The next step will be determining a pitch and roll controller for getting the AR.Drone to the waypoints coordinate during the landing operation. Using the PID control input from [14]:

$$u_{\emptyset} = (0.35(\hat{x} - x) + 0.2\dot{x}) \cos\varphi - (0.35(\hat{y} - y) + 0.2\dot{y}) \sin\varphi \tag{5.8}$$

$$u_{\theta} = -(0.35(\hat{x} - x) + 0.2\dot{x}) \sin\varphi - (0.35(\hat{y} - y) + 0.2\dot{y}) \cos\varphi \tag{5.9}$$

Chapter 6

Implementation

This chapter describes the approach to the problem and the system developed and implemented in the course of this project. My approach consists of three major components:

Monocular SLAM: a monocular SLAM algorithm as described in Chapter 3 is implemented to navigate the drone in a previously unknown area without GPS access. The video frames captured by the onboard camera is used to determine the pose of the AR.Drone. The scale of the map is determined by comparing sensor data with PTAM's pose estimate. This is the implementation of an already completed work by [26].

Image Processing and Data Extraction: An Open CV implementation called ALVAR is used for the marker recognition. A marker is placed on the UGV and it is used by the AR.Drone in locating the ground vehicle and in determining the position and orientation of the ground vehicle. This information is used in determining the navigation goal point.

Autonomous Landing: Two simple algorithms were developed for two different landing approaches. These algorithms are used to get the AR.Drone from any initial position to the goal position which is the landing pad on the UGV. A marker is placed on this landing pad and it is used to update the UAVs position during landing. A PID controller is used to fly the AR.Drone to the target position.

6.1 Simulation Platform

Due to already existing physical damages on the AR.Drone and its sensors from other experiments carried out in the lab, I noticed that the roll, pitch and yaw of the AR.Drone are not exactly the right values. These errors are system errors from the AR.Drone sensors. Because of this, this project was implemented on two different platforms. The Monocular slam was implemented in the lab on a physical AR.Drone while the autonomous landing operation was implemented in ROS Gazebo.

ROS Gazebo is a ROS package that provides the necessary interfaces to simulate a robot in the Gazebo 3D rigid body simulator for robots. It integrates with ROS using ROS messages, services and dynamic reconfigure. In ROS Gazebo, an empty world file is used for this simulation. Models and objects with their position can be added to the workspace. The laws of physics may be enabled to allow for a more real scenario. For this thesis, the AR.Drone model was added. Boxes were used to simulate obstacles preventing the AR.Drone from seeing the marker.

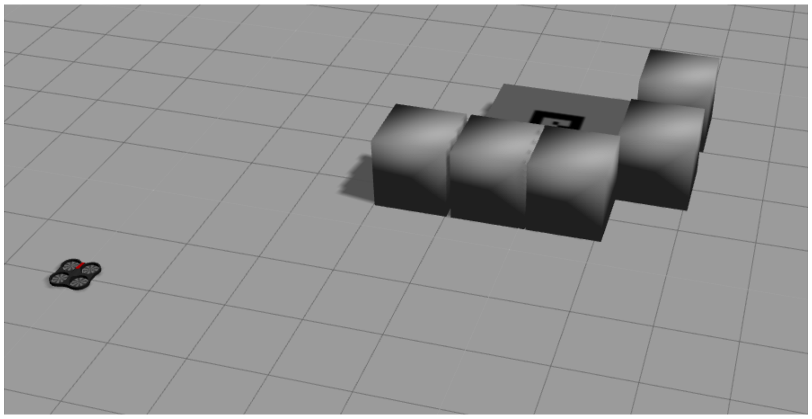


Figure 6-1 Workspace Simulation in ROS Gazebo

6.2 Software Architecture

The major software challenge in this project was integrating all the different applications and their platforms on a single application to achieve the project's aim. The sensory data from the AR.Drone was used to create a complete feedback system, providing information on the state of the AR.Drone. As a dynamic system, it is essential that we are able to maintain communication between the AR.Drone and the ground station. This is achieved by creating two parallel processes: one for running the software's framework and AR.Drone downlink while the other is for running the major components of this project.

These parallel processes communicate using named pipes for interchanging message, as well as shared memory regions, mutexes and events for asynchronously interchanging video, navigational and control data.

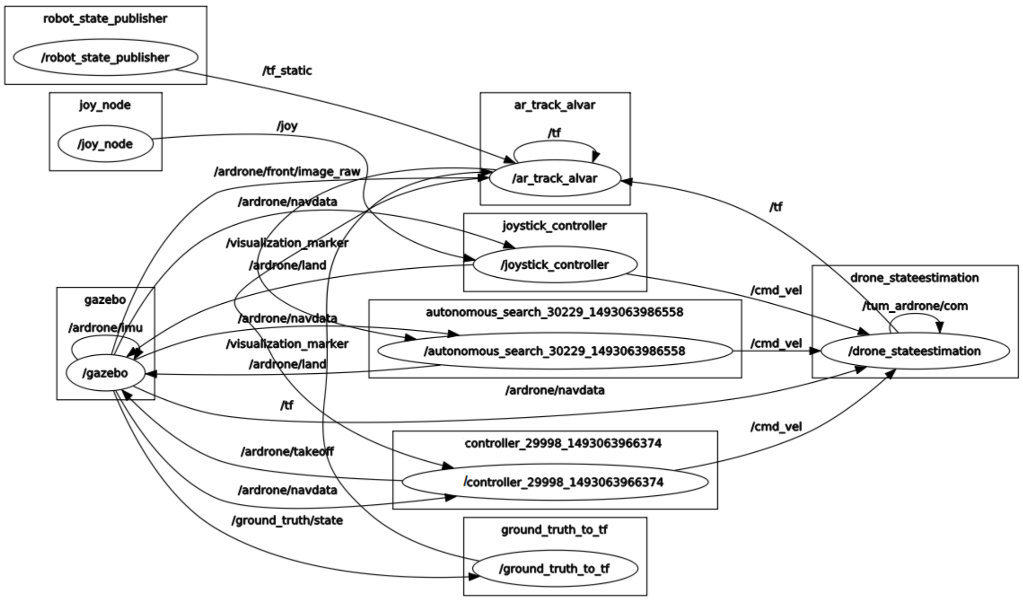


Figure 6-2 System Topic and Node Relationship

The overall project is based on two different operations: Monocular SLAM operation and Autonomous Landing System operation. During the monocular slam, an open-source PTAM is used for localization and mapping, using the onboard camera for detecting keypoints. While during the autonomous landing operation, the camera is used for detecting the marker and estimating its pose. As observed, the two different operations entail different tasks for the onboard cameras and therefore the application of the onboard camera switches with a change in system operation.

6.2.1. Scale Estimation

Scale estimation is important for navigating a quadrotor. The absolute scale of a map cannot be obtained by only using the visual odometry from a monocular camera. An external or prior information related to the depth of the reconstructed points or measurements of the camera motion is required. An estimate of the baseline length between the two captures is used in determining the scale factor. For this project, the scale is estimated using the ultrasound height measurements and integrating over the measured vertical speeds.

6.3. State Estimation

For an autonomous drone, it is important for it to have a perception. The world state cannot be observed directly, therefore the AR.Drone needs an estimate of its state. These estimates are derived from the sensors onboard the AR.Drone and are updated continuously. The extended Kalman filter is used to estimate the world and update it continuously. It combines the estimates provided by PTAM with sensor measurements provided by the IMU.

6.3.1. The State Space

The internal state of the Kalman filter is defined to be

$$x(t) \coloneqq (x, y, z, \dot{x}, \dot{y}, \dot{z}, \emptyset, \theta, \varphi, \dot{\varphi})^T \in \boldsymbol{R}^{10} \tag{6.1}$$

Where

- x, y and z correspond to the world-coordinates of the AR.Drone center in metres,
- $\dot{x}$, $\dot{y}$ and $\dot{z}$ correspond to the velocity of the AR.Drone in meters per second,
- $\emptyset, \theta\ and\ \varphi$ correspond to roll angle, pitch angle and yaw angle in degree, representing the drone's orientation.
- $\dot{\varphi}$ corresponds to the yaw-rotational speed in degree per second.

With the state now defined, we know exactly what parameters are needed to have a complete model of the platform.

Chapter 7

Flight Tests

This chapter shows that the performance of the developed system which is still a work in progress with several areas of improvement. First, I implemented the already completed work by [14] then evaluated the proposed autonomous landing approach experimentally by observing the precision of the controller to the landing pad.

Search Algorithm

First, I tested the search algorithm in the ROS Gazebo environment. For this test, the marker recognition was turned off to ensure the AR.Drone completes a full search algorithm without any interference from the marker.

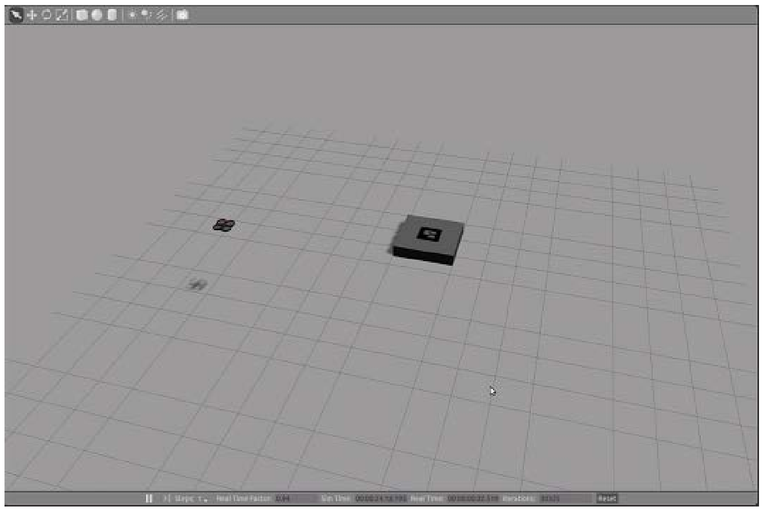


Figure 7-1 Screenshot of ROS Gazebo environment for search simulation

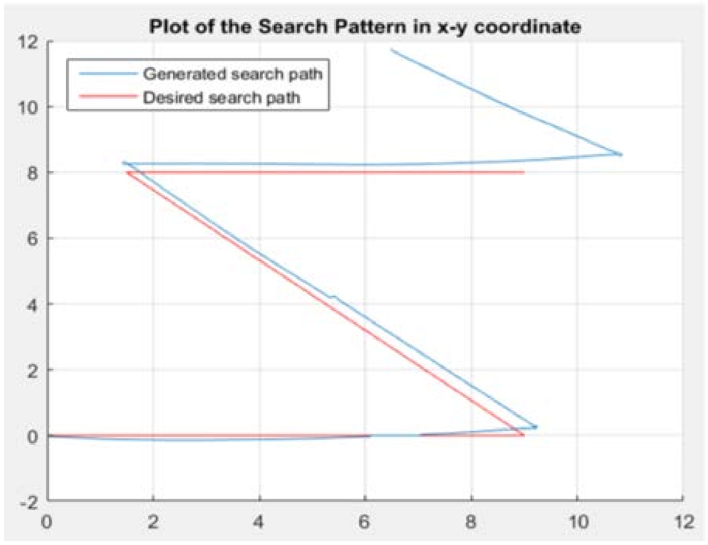


Figure 7-2 Trajectory Plot of Search Pattern in x-y coordinate

First Approach

For the first landing approach, I have considered two different scenarios.

Scenario 1: The AR.Drone takes off and locates the marker without any obstruction in its FOV. The timeline for this is shown in fig 7-3.

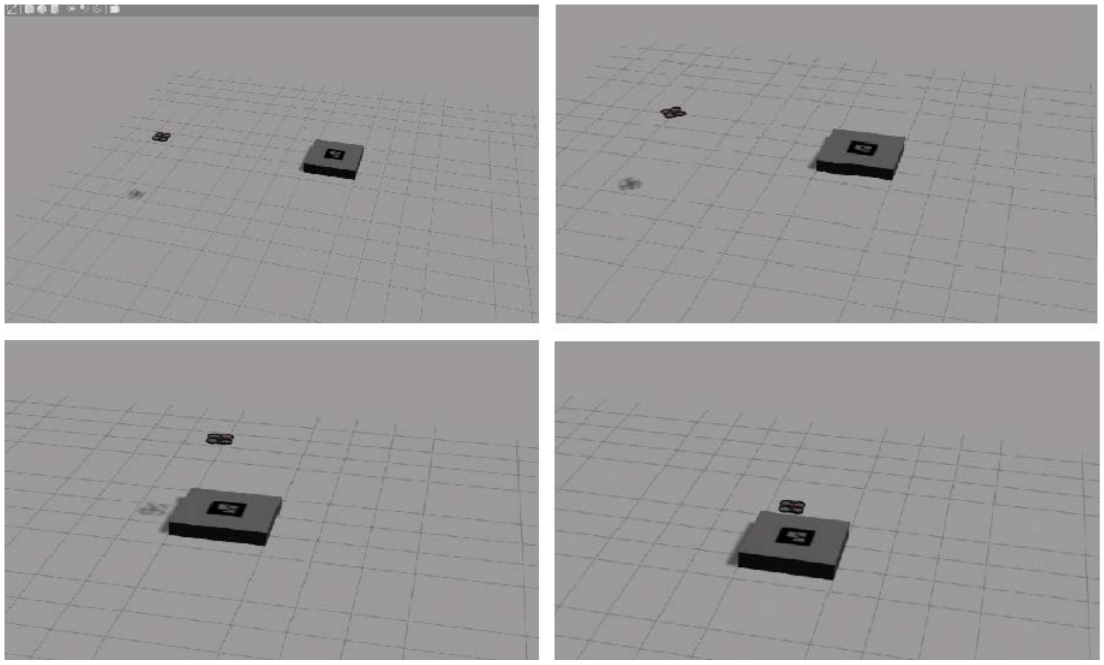


Figure 7-3 Gazebo Simulation using the first approach without any obstruction

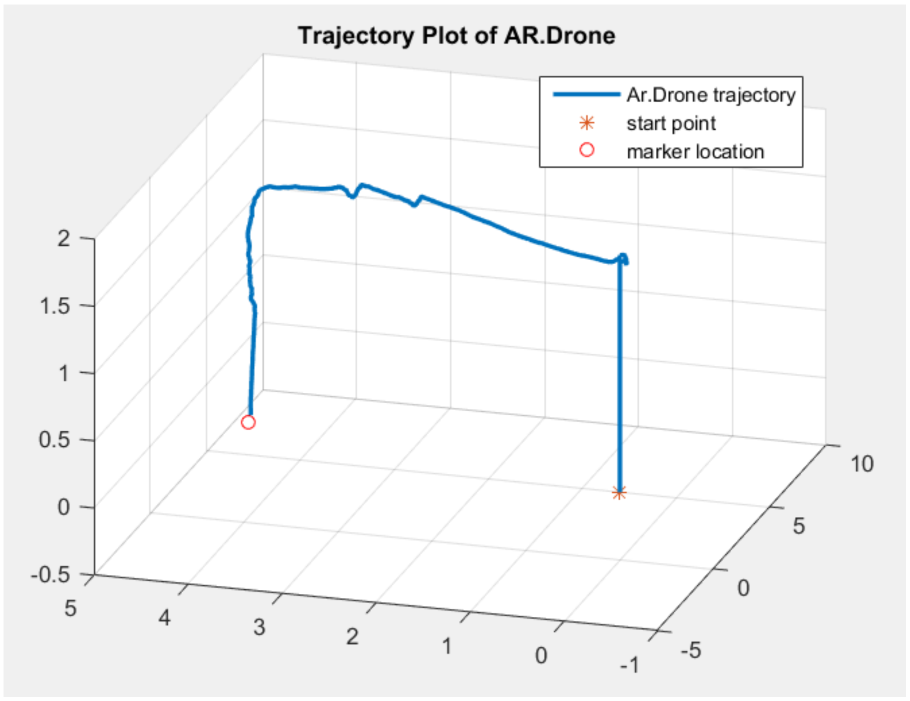


Figure 7-4 Trajectory Plot of the first landing approach

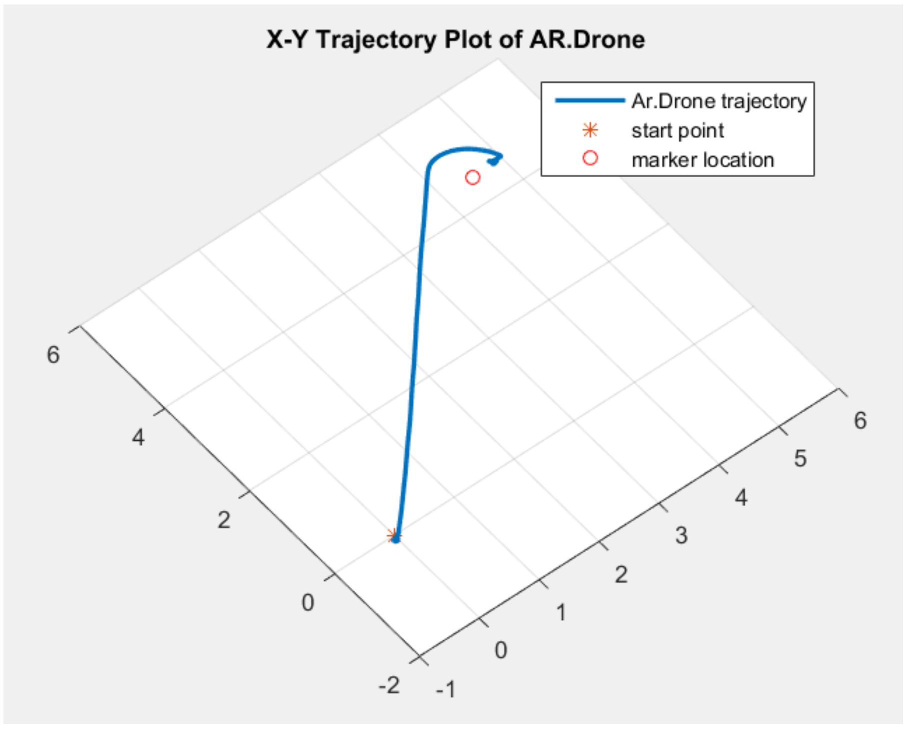


Figure 7-5 X-Y Plot of the first landing approach

Observe the behavior of the AR.Drone in the X-Y Plot of the first landing approach, the bottom camera corrects position of the AR.Drone when landing.

Scenario 2: An obstruction is placed in the FOV of the AR.Drone and the search algorithm is used to locate the marker. The AR.Drone flies some distance using the search algorithm before locating the marker.

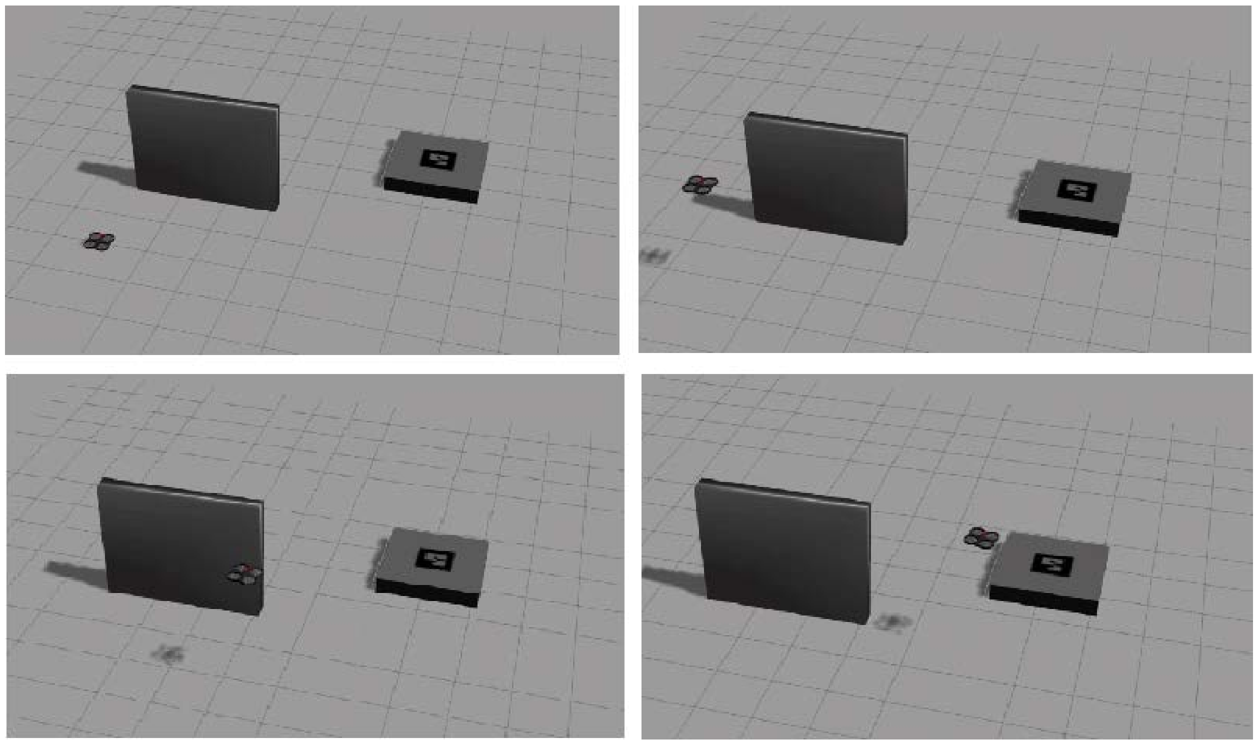


Figure 7-6 Landing using the first approach with an obstruction in its FOV

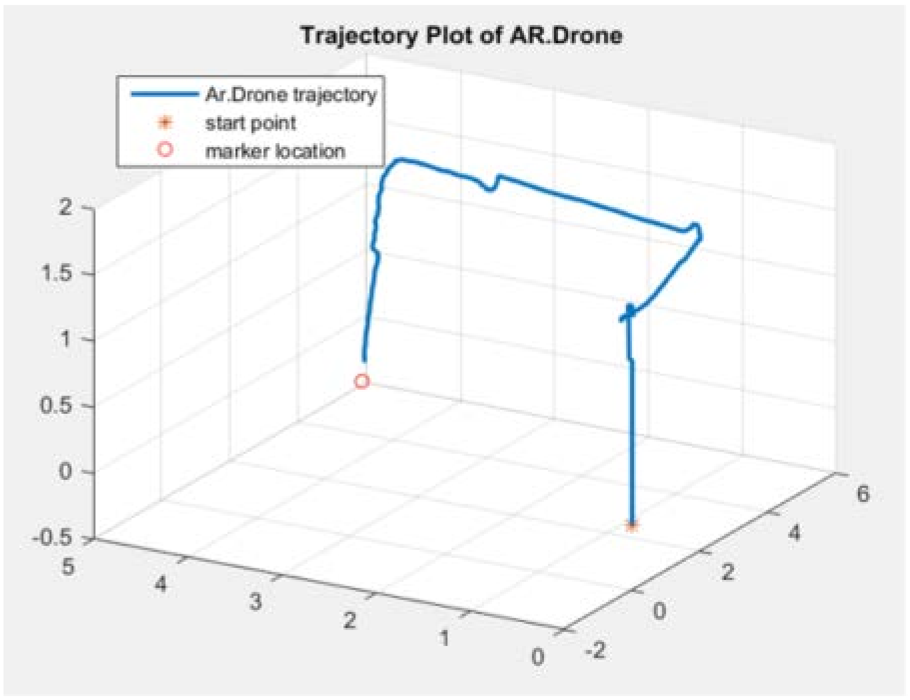


Figure 7-7 Trajectory Plot of the first landing approach using the search algorithm

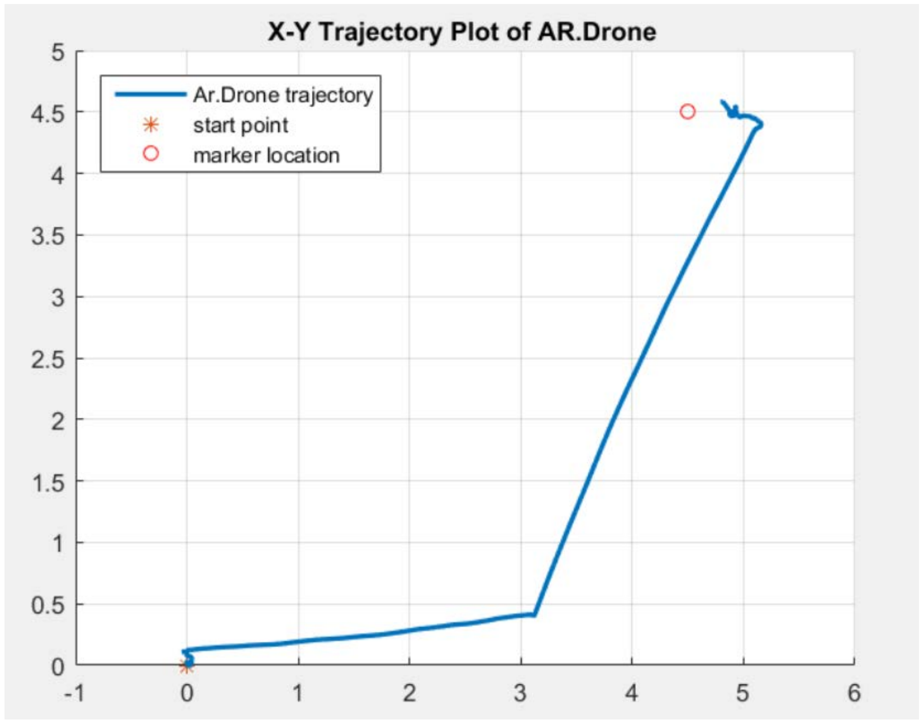


Figure 7-8 X-Y Trajectory Plot of the first landing approach using the search algorithm

Second Approach

Also just like the first approach, two scenarios were considered for this: one without an obstruction to the FOV and the other with. The logic behind inserting an obstruction is to allow the use of the search algorithm.

Scenario 1: The AR.Drone takes off, locates the marker and lands on it without any obstruction in its FOV. The timeline for this is shown in fig 7-9.

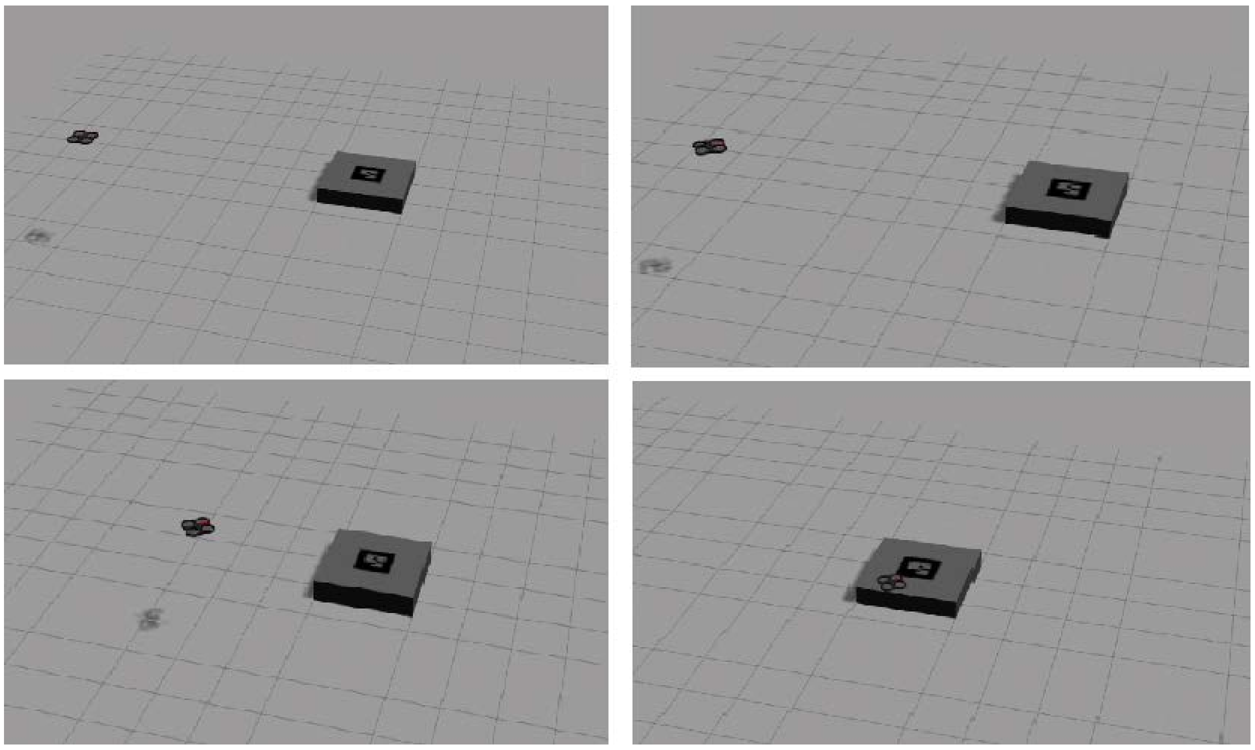


Figure 7-9 Landing using the first approach without an obstruction in its FOV

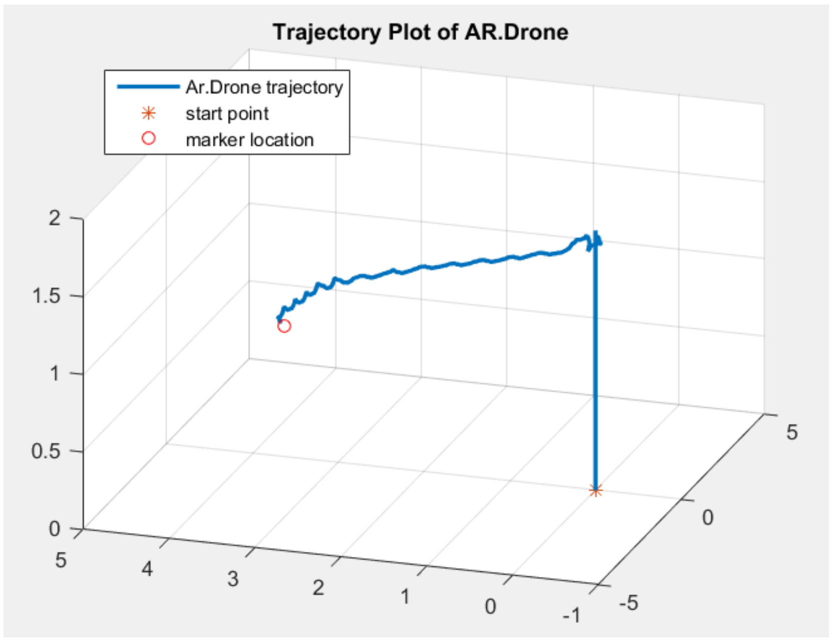


Figure 7-10 Trajectory Plot of the second landing approach

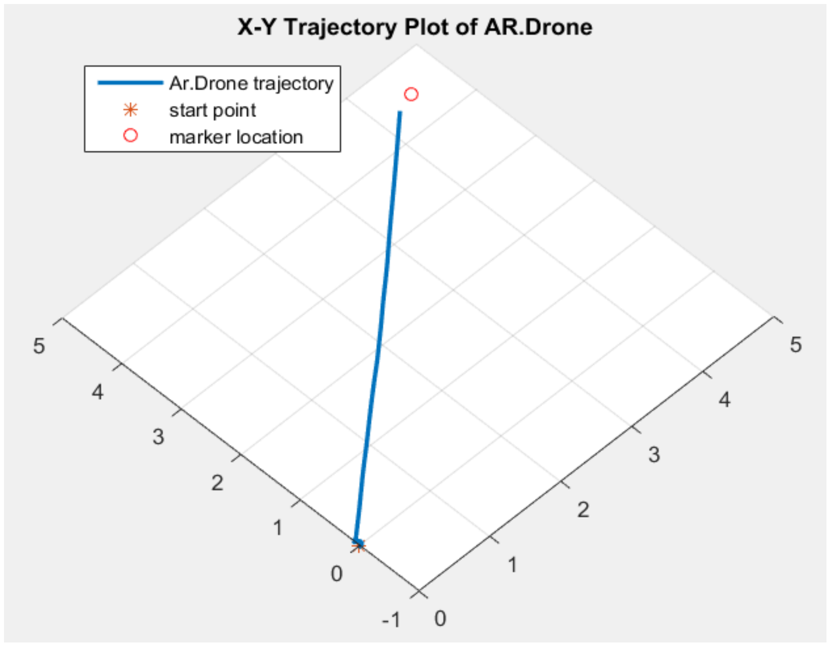


Figure 7-11 X-Y Trajectory Plot of the second landing approach

Scenario 2: An obstruction is placed in the FOV of the AR.Drone and the search algorithm is used to locate the marker.

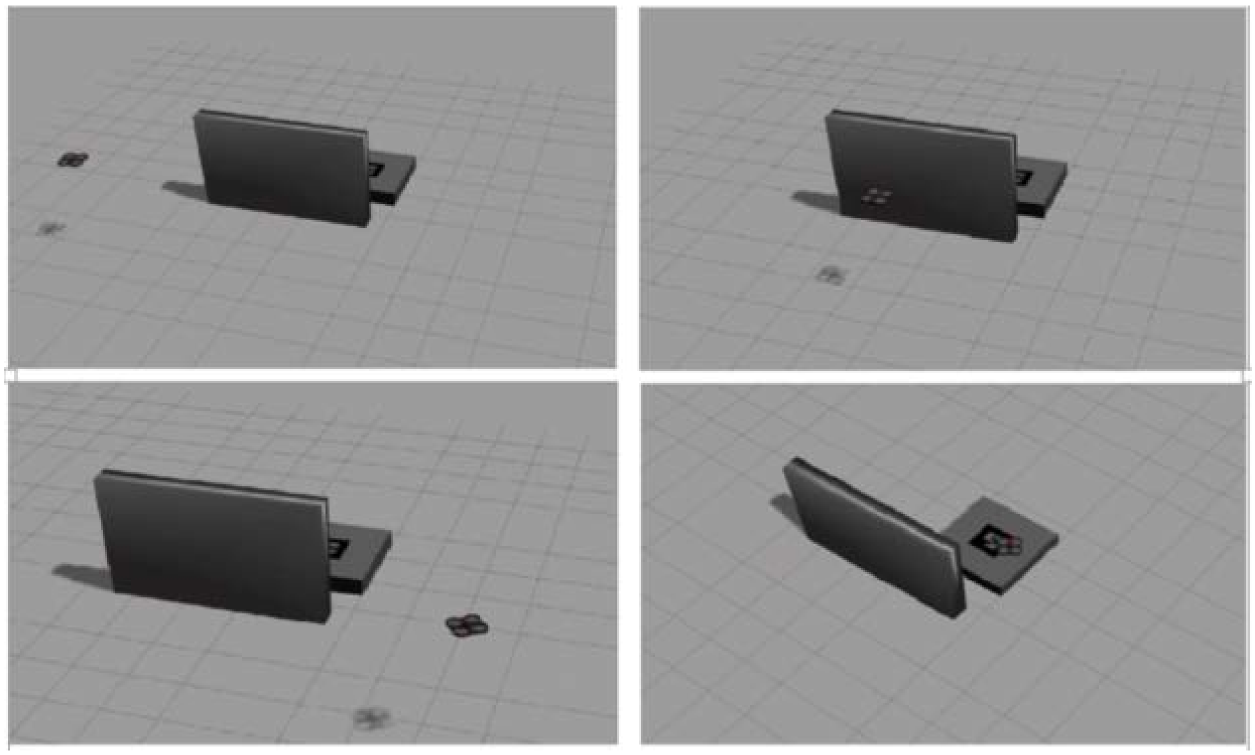


Figure 7-12 Landing using the second approach with an obstruction in its FOV

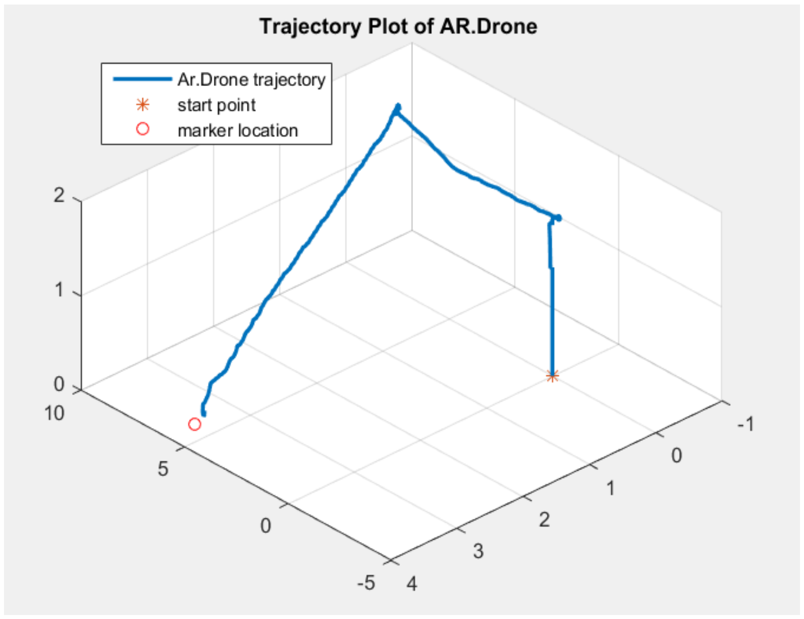


Figure 7-13 Trajectory Plot of the second landing approach

Chapter 8

Conclusion

Autonomous landing is a key operation during joint operations between UAVs and UGVs. It is especially important to use a mobile launching and landing pad. This adds flexibility to the operation as well as eradicates the need for an extensive ground equipment. In this thesis, I was able to extend the work already done by [14] while incorporating an autonomous landing system in it.

The AR.Drone was able to localize and navigate in a previously unknown environment without GPS access using a monocular camera onboard. Also, the AR.Drone was able to land autonomously on a mobile base. To achieve the landing operation, a marker recognition system was used in estimating the pose of the landing pad and a search algorithm was used in locating the marker. A PID controller was used for the control signals: roll, pitch, yaw and altitude to navigate the drone from an initial point to the landing pad.

The drawback experienced during this thesis was in the first landing approach. During the first landing approach, the position of the marker is determined once and never updated during the reminder of the flight. Without the AR.Drone updating its pose estimate from the landing pad, it drifts away from the landing pad. Once the marker's position has been entered, the onboard camera switches from the front camera to the bottom camera which corrects the AR.Drone's position for landing when it hovers over the landing pad.

Overall, the second landing system is a more precise system as the AR.Drone continuously updates its position using the feedback from the front camera. The front camera maintains line of sight on the landing pad and continuously updates its landing position.

Chapter 9

Future Work

This thesis emphasized on autonomous landing, there are still several areas of joint operations between UAVs and UGVs to be exploited. There were some limitations during the implementation of this project that can be addressed in future work, which will be discussed below:

Joint SLAM for UAV and UGV

The AR.Drone runs its SLAM on a ground computation station. A camera can be installed on the UGV and the same ground computation station can be used to run a SLAM system for the UGV as well. PTAM can be extended to both the UGV and UAV to run a joint SLAM on the ground computation station within the same coordinate system. This will eliminate the need to search for the landing station as the location of the UGV can be easily determined within the SLAM system.

Incorporating a Radio Finder

In this thesis, I was limited by how far away the AR.Drone could fly from the marker. Locating the marker posed a serious challenge as the onboard camera could not pick up the marker beyond a certain extent. It was easier to use the search algorithm when the search area is of considerable size. A preferred approach would be using a radio direction finder in large areas. With a radio finder, distance would not be a factor. Using two or more measurements from different locations, the location of an unknown transmitter can be determined.

Introducing a Learning Control

The development of a more stable controller will definitely improve the overall behavior of the system. Learning systems have been considered as high level intelligent systems. For this thesis, a PID controller was used for controlling the AR.Drone. A learning controller will be an improvement for the compensation of unknown periodic disturbances, allowing the AR.Drone react better to changing scenarios. A learning controller such as neural networks compensates the disturbance without redesigning existing control loops.

Onboard Computation

The memory and computational limits of the control board which run onboard the AR.Drone had to be taken into consideration during the development of the application. For my thesis, all three channels mentioned in chapter 2 were used to acquire data as well as send commands via a wireless network. This caused sufficient delays. An improvement will be performing all computations required onboard as well as increasing the onboard memory capacity. In this case, one can access the drone cameras and onboard sensors directly without a delay caused by the wireless data transfer. Faster control loops are achievable and even when an application is running on the control board, the internal controllers which take care of the drone stability can be active.

Running both Front and Bottom Cameras Simultaneous

A disadvantage of the camera system used on the AR.Drone is that a user cannot obtain live feeds from both cameras simultaneously. Instead, the user needs to select either the front and bottom camera. For this thesis, I switched between both cameras to maintain line of sight on the marker. Switching the modes is not instant, it takes about 270 ms and during the switching time, the provided image contains invalid data. The ability to use both onboard cameras simultaneously for the marker's pose estimation will definitely yield better results during the landing operation as the onboard camera will be able to maintain visual contact on the marker throughout the landing operation.

Chapter 10

Appendix

This section includes some modifications to the original source code developed by Jakob Engel. Refer to [33] for the original code and instructions on how to execute the code using the ROS platform.

Modifications were made to the ControlNode.cpp and ControlNode.h files to handle the marker callback.

In the ControlNode.h file, add the line below beneath “//ROS message callbacks”

- Marker_callback(const visualization_msgs::Marker& msg);

Replace the content of the ControlNode.cpp with the source code below:

```
#include "ros/ros.h"
#include "ros/callback_queue.h"
#include <ros/package.h>
#include "ControlNode.h"

#include "geometry_msgs/Twist.h"
#include "../HelperFunctions.h"
#include "tum_ardrone/filter_state.h"
#include "std_msgs/String.h"
#include <sys/stat.h>
#include <string>
#include <cmath>

// include KI's
#include "KI/KIAutoInit.h"
#include "KI/KIFlyTo.h"
#include "KI/KILand.h"
#include "KI/KIProcedure.h"

//These Gazebo libraries are needed only when using Gazebo for simulation
#include "gazebo/gazebo.hh"
#include "gazebo/transport/TransportTypes.hh"
#include "gazebo/msgs/MessageTypes.hh"
#include "gazebo/common/Time.hh"
#include "gazebo/common/Plugin.hh"

//ros package for marker  ar_tag
//http://wiki.ros.org/rviz/Tutorials/Markers:%20Basic%20Shapes
#include <visualization_msgs/Marker.h>
#include <gazebo_msgs/ModelStates.h>

//For image processing
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include <tf/transform_datatypes.h>
#include <tf/transform_listener.h>

using namespace std;
using namespace tum_ardrone;
using namespace cv;
using namespace gazebo;


static const std::string OPENCV_WINDOW="Image window";

pthread_mutex_t ControlNode::logControl_CS = PTHREAD_MUTEX_INITIALIZER;


ControlNode::ControlNode()
{
    control_channel = nh_.resolveName("cmd_vel");
    dronepose_channel = nh_.resolveName("ardrone/predictedPose");
    command_channel = nh_.resolveName("tum_ardrone/com");
    takeoff_channel = nh_.resolveName("ardrone/takeoff");
    land_channel = nh_.resolveName("ardrone/land");
    toggleState_channel = nh_.resolveName("ardrone/reset");

    packagePath = ros::package::getPath("tum_ardrone");

    std::string val;
    float valFloat;

    ros::param::get("~minPublishFreq", val);
    if(val.size()>0)
            sscanf(val.c_str(), "%f", &valFloat);
    else
            valFloat = 110;
    minPublishFreq = valFloat;
    cout << "set minPublishFreq to " << valFloat << "ms"<< endl;


   // other internal vars
   logfileControl = 0;
   hoverCommand.gaz = hoverCommand.pitch = hoverCommand.roll = hoverCommand.yaw = 0;
   lastControlSentMS = 0;

   // channels
   dronepose_sub = nh_.subscribe(dronepose_channel, 10, &ControlNode::droneposeCb, this);
   //ar_tag  marker topic
   marker_sub = node.subscribe("/visualization_marker", 1, &ControlNode::Marker_callback,this);
   vel_pub             = nh_.advertise<geometry_msgs::Twist>(control_channel,1);
   tum_ardrone_pub           = nh_.advertise<std_msgs::String>(command_channel,50);
   tum_ardrone_sub           = nh_.subscribe(command_channel,50, &ControlNode::comCb, this);
   takeoff_pub       = nh_.advertise<std_msgs::Empty>(takeoff_channel,1);
   land_pub          = nh_.advertise<std_msgs::Empty>(land_channel,1);
   toggleState_pub             = nh_.advertise<std_msgs::Empty>(toggleState_channel,1);

  // services handler
 setReference_ = nh_.advertiseService("drone_autopilot/setReference", &ControlNode::setReference,
this);
setMaxControl_ = nh_.advertiseService("drone_autopilot/setMaxControl", &ControlNode::setMaxControl,
this);
setInitialReachDistance_ = nh_.advertiseService("drone_autopilot/setInitialReachDistance",
&ControlNode::setInitialReachDistance, this);
```

```
        setStayWithinDistance_ = nh_.advertiseService("drone_autopilot/setStayWithinDistance",
&ControlNode::setStayWithinDistance, this);
        setStayTime_ = nh_.advertiseService("drone_autopilot/setStayTime", &ControlNode::setStayTime, this);
        startControl_ = nh_.advertiseService("drone_autopilot/start", &ControlNode::start, this);
        stopControl_ = nh_.advertiseService("drone_autopilot/stop", &ControlNode::stop, this);
        clearCommands_ = nh_.advertiseService("drone_autopilot/clearCommands", &ControlNode::clear, this);
        hover_ = nh_.advertiseService("drone_autopilot/hover", &ControlNode::hover, this);
        lockScaleFP_ = nh_.advertiseService("drone_autopilot/lockScaleFP", &ControlNode::lockScaleFP, this);

        // internals
        parameter_referenceZero = DronePosition(TooN::makeVector(0,0,0),0);
        parameter_MaxControl = 1;
        parameter_InitialReachDist = 0.2;
        parameter_StayWithinDist = 0.5;
        parameter_StayTime = 2;
        isControlling = false;
        currentKI = NULL;
        lastSentControl = ControlCommand(0,0,0,0);
        bool stamp_sync ;
        bool first_navdata ;
        bool first_marker ;
        bool Marker_back;
        ControlNode::MarkerTarget marker;

        //to deal with camera image
        cv::Mat img_raw;
        cv_bridge::CvImagePtr cv_ptr;
        CvFont font;

        // create controller
        controller = DroneController();
        controller.node = this;
        }

        ControlNode::~ControlNode()
        {

        }

        pthread_mutex_t ControlNode::commandQueue_CS = PTHREAD_MUTEX_INITIALIZER;
        void ControlNode::droneposeCb(const tum_ardrone::filter_stateConstPtr statePtr)
        first_navdata(true), first_marker(true), Marker_back(false)
        {
                // do controlling
                pthread_mutex_lock(&commandQueue_CS);

                // as long as no KI present:
                // pop next KI (if next KI present).
                while(currentKI == NULL && commandQueue.size() > 0)
                        popNextCommand(statePtr);

                // if there is no current KI now, we obviously have no current goal -> send drone hover
                if(currentKI != NULL)
                {
                        // let current KI control.
                        this->updateControl(statePtr);
```

```
	}
	else if(isControlling)
	{
		sendControlToDrone(hoverCommand);
		ROS_DEBUG("Autopilot is Controlling, but there is no KI -> sending HOVER");
	}


	pthread_mutex_unlock(&commandQueue_CS);
}

// pops next command(s) from queue (until one is found thats not "done" yet).
// assumes propery of command queue lock exists (!)
void ControlNode::popNextCommand(const tum_ardrone::filter_stateConstPtr statePtr)
{
	// should actually not happen., but to make shure:
	// delete existing KI.
	if(currentKI != NULL)
	{
		delete currentKI;
		currentKI = NULL;
	}

	// read next command.
	while(currentKI == NULL && commandQueue.size() > 0)
	{
		std::string command = commandQueue.front();
		commandQueue.pop_front();
		bool commandUnderstood = false;

		// print me
		ROS_INFO("executing command: %s",command.c_str());

		int p;
		char buf[100];
		float parameters[10];

		// replace macros
		if((p = command.find("$POSE$")) != std::string::npos)
		{
			snprintf(buf,100,
"%.3f %.3f %.3f %.3f",statePtr->x,statePtr->y,statePtr->z,statePtr->yaw);
			command.replace(p,6,buf);
		}
		if((p = command.find("$REFERENCE$")) != std::string::npos)
		{
			snprintf(buf,100,
"%.3f %.3f %.3f %.3f",parameter_referenceZero.pos[0],parameter_referenceZero.pos[1],parameter_refer
enceZero.pos[2],parameter_referenceZero.yaw);
			command.replace(p,11,buf);
		}

		// -------- commands -----------
		// autoInit
		if(sscanf(command.c_str(),"autoInit %f %f %f %f",&parameters[0], &parameters[1],
&parameters[2], &parameters[3]) == 4)
```

```
			{
				currentKI = new 
KIAutoInit(true,parameters[0],parameters[1],parameters[2],parameters[3],true);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			}

			else if(sscanf(command.c_str(),"autoTakeover %f %f %f %f",&parameters[0], 
&parameters[1], &parameters[2], &parameters[3]) == 4)
			{
				currentKI = new 
KIAutoInit(true,parameters[0],parameters[1],parameters[2],parameters[3],false);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			}

			// takeoff
			else if(command == "takeoff")
			{
				currentKI = new KIAutoInit(false);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			}

			// setOffset
			else if(sscanf(command.c_str(),"setReference %f %f %f %f",&parameters[0], 
&parameters[1], &parameters[2], &parameters[3]) == 4)
			{
				parameter_referenceZero = 
DronePosition(TooN::makeVector(parameters[0],parameters[1],parameters[2]),parameters[3]);
				commandUnderstood = true;
			}

			// setMaxControl
			else if(sscanf(command.c_str(),"setMaxControl %f",&parameters[0]) == 1)
			{
				parameter_MaxControl = parameters[0];
				commandUnderstood = true;
			}

			// setInitialReachDist
			else if(sscanf(command.c_str(),"setInitialReachDist %f",&parameters[0]) == 1)
			{
				parameter_InitialReachDist = parameters[0];
				commandUnderstood = true;
			}

			// setStayWithinDist
			else if(sscanf(command.c_str(),"setStayWithinDist %f",&parameters[0]) == 1)
			{
				parameter_StayWithinDist = parameters[0];
				commandUnderstood = true;
			}

			// setStayTime
			else if(sscanf(command.c_str(),"setStayTime %f",&parameters[0]) == 1)
			{
				parameter_StayTime = 0;
				commandUnderstood = true;
			}

			// goto
			else if(sscanf(command.c_str(),"goto %f %f %f %f",&parameters[0], &parameters[1],
&parameters[2], &parameters[3]) == 4)
			{

				currentKI = new KIFlyTo(
					DronePosition(
					TooN::makeVector(parameters[0],parameters[1],parameters[2]) +
parameter_referenceZero.pos,
						parameters[3] + parameter_referenceZero.yaw),
					parameter_StayTime,
					parameter_MaxControl,
					parameter_InitialReachDist,
					parameter_StayWithinDist
					);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;

			}

			//find marker
			else if(sscanf(command.c_str(),"find marker")
			{


				currentKI = new KIFlyTo(
					DronePosition(
					TooN::makeVector(waypoints_x[1],waypoints_y[1],waypoints_z[1]) +
parameter_referenceZero.pos,
						marker.yaw + parameter_referenceZero.yaw),
					parameter_StayTime,
					parameter_MaxControl,
					parameter_InitialReachDist,
					parameter_StayWithinDist
					);
			currentKI->setPointers(this,&controller);
				commandUnderstood = true;


			// moveBy
			else if(sscanf(command.c_str(),"moveBy %f %f %f %f",&parameters[0], &parameters[1],
&parameters[2], &parameters[3]) == 4)
			{
				currentKI = new KIFlyTo(
					DronePosition(
					TooN::makeVector(parameters[0],parameters[1],parameters[2]) +
controller.getCurrentTarget().pos,
						parameters[3] + controller.getCurrentTarget().yaw),
					parameter_StayTime,
					parameter_MaxControl,
					parameter_InitialReachDist,
					parameter_StayWithinDist
					);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;

			}

			// moveByRel
			else if(sscanf(command.c_str(),"moveByRel %f %f %f %f",&parameters[0],
&parameters[1], &parameters[2], &parameters[3]) == 4)
			{
				currentKI = new KIFlyTo(
					DronePosition(

		TooN::makeVector(parameters[0]+statePtr->x,parameters[1]+statePtr->y,parameters[2]+statePtr-
>z),
						parameters[3] + statePtr->yaw),
					parameter_StayTime,
					parameter_MaxControl,
					parameter_InitialReachDist,
					parameter_StayWithinDist
					);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;

			}

			// land
			else if(command == "land")
			{
				currentKI = new KILand();
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			}

			// setScaleFP
			else if(command == "lockScaleFP")
			{
				publishCommand("p lockScaleFP");
				commandUnderstood = true;
			}

			if(!commandUnderstood)
				ROS_INFO("unknown command, skipping!");
		}

}

void ControlNode::comCb(const std_msgs::StringConstPtr str)
{
		// only handle commands with prefix c
		if(str->data.length() > 2 && str->data.substr(0,2) == "c ")
		{
			std::string cmd =str->data.substr(2,str->data.length()-2);

			if(cmd.length() == 4 && cmd.substr(0,4) == "stop")
		{
			stopControl();
		}
		else if(cmd.length() == 5 && cmd.substr(0,5) == "start")
		{
			startControl();
		}
		else if(cmd.length() == 13 && cmd.substr(0,13) == "clearCommands")
		{
			clearCommands();
		}
		else
		{
			pthread_mutex_lock(&commandQueue_CS);
			commandQueue.push_back(cmd);
			pthread_mutex_unlock(&commandQueue_CS);
		}
	}

	// global command: toggle log
	if(str->data.length() == 9 && str->data.substr(0,9) == "toggleLog")
	{
		this->toogleLogging();
	}
}

void ControlNode::Loop()
{
	ros::Time last = ros::Time::now();
	ros::Time lastStateUpdate = ros::Time::now();

	while (nh_.ok())
	{

		// -------------- 1. spin for 50ms, do main controlling part here. ---------------
		while((ros::Time::now() - last) < ros::Duration(minPublishFreq / 1000.0))
			ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(minPublishFreq /
1000.0 - (ros::Time::now() - last).toSec()));
		last = ros::Time::now();


		// -------------- 2. send hover (maybe). ---------------
		if(isControlling && getMS(ros::Time::now()) - lastControlSentMS > minPublishFreq)
		{
			sendControlToDrone(hoverCommand);
			ROS_WARN("Autopilot enabled, but no estimated pose received - sending
HOVER.");
		}

		// -------------- 2. update info. ---------------
		if((ros::Time::now() - lastStateUpdate) > ros::Duration(0.4))
		{
			reSendInfo();
			lastStateUpdate = ros::Time::now();
		}
	}
```

```
}

//Waypoint Generator from the drone's initial point to the landing pad.
void ControlNode::waypoint(const target_x, const target_y, const target_z)
{
                double ref_x=0;
                double ref_y=0;
                double ref_z=0;
                double goal_x = target_x; //landing pad x
                double goal_y = target_y //landing pad y
                double goal_z = target_z //landing pad z
                //state.yaw
                int number_of_waypoints=4;
                int i;
                double stepsize_x = goal_x/number_of_waypoints;
                double stepsize_y = goal_y/number_of_waypoints;
                double stepsize_z = goal_z/number_of_waypoints;
                double waypoints_x[number_of_waypoints], waypoints_y[number_of_waypoints],
waypoints_z[number_of_waypoints];

                waypoints_x[3] = goal_x;
                waypoints_y[3] = goal_y;
                waypoints_z[3] = goal_z;

                waypoints_x[0] = ref_x;
                waypoints_y[0] = ref_y;
                waypoints_z[0] = ref_z;

                for (i=1; i < number_of_waypoints-1; i++)
                {
                waypoints_x[i]= waypoints_x[i-1] + stepsize_x;
                waypoints_y[i]= waypoints_y[i-1] + stepsize_y;
                waypoints_z[i]= waypoints_z[i-1] + stepsize_z;
                }


                return 0;
}


        void ControlNode::Marker_callback(const visualization_msgs::Marker& msg)
        {
                int waypoint_array;
                Marker_back = true;
                //Center of marker is (0,0,0)
                cout<<"Marker in:"<<msg.header.stamp.toSec()<<endl;


                tf::Pose marker_in_camera;
                tf::Pose camera_in_marker;


                tf::poseMsgToTF(msg.pose, marker_in_camera);
                camera_in_marker = marker_in_camera.inverse();

                marker.x = camera_in_marker.getOrigin().x();//m,  position wrt. tag center
```

```
			marker.y = camera_in_marker.getOrigin().y();//m,  position wrt. tag center
//For the first landing approach, make marker.z= 0
			marker.z = camera_in_marker.getOrigin().z();//m
			marker.yaw = tf::getYaw(camera_in_marker.getRotation() );

			tf::Matrix3x3 m(camera_in_marker.getRotation());
			double t_r, t_p, t_y;
			m.getRPY(t_r, t_p, t_y);
			state.roll  = t_r * 180 / M_PI;
			state.pitch = t_p * 180 / M_PI;
			state.yaw   = t_y * 180 / M_PI;
			tf::poseTFToMsg(camera_in_marker, state.pose);


//For second approach method
//For the first waypoint navigation
			if(first_marker){
				 waypoint_array=1;
				first_marker = false;
				waypoint(marker.x, marker.y, marker.z);
				currentKI = new KIFlyTo(
				DronePosition(
				TooN::makeVector(waypoints_x[waypoint_array], waypoints_y[waypoint_array],
waypoints_z[waypoint_array]) + parameter_referenceZero.pos, parameters[3] +
parameter_referenceZero.yaw),
				parameter_StayTime,
				parameter_MaxControl,
			parameter_InitialReachDist,
			parameter_StayWithinDist);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			waypoint_array=2;
			}
//For the subsequent waypoint navigation. Once the error to the goal point is <0.3, a new KI is sent for the
next waypoint.
			if( (!first_marker ) && ( new_err[0] <=0.3) && ( new_err[1] <=0.3) && ( new_err[2] <=0.3) )
			{
			currentKI = new KIFlyTo(
				DronePosition(
				TooN::makeVector(waypoints_x[waypoint_array], waypoints_y[waypoint_array],
waypoints_z[waypoint_array]) + parameter_referenceZero.pos,				parameters[3] +
parameter_referenceZero.yaw),
				parameter_StayTime,
				parameter_MaxControl,
			parameter_InitialReachDist,
			parameter_StayWithinDist);
				currentKI->setPointers(this,&controller);
				commandUnderstood = true;
			waypoint_array=waypoint+1;
			}
			if( (!first_marker ) && ( new_err[0] <=0.3) && ( new_err[1] <=0.3) && ( new_err[2] <=0.3)
&& commandQueue.size() == 0)
{
sendLand()
}
}
```

```
void ControlNode::dynConfCb(tum_ardrone::AutopilotParamsConfig &config, uint32_t level)
{
	controller.Ki_gaz = config.Ki_gaz;
	controller.Kd_gaz = config.Kd_gaz;
	controller.Kp_gaz = config.Kp_gaz;

	controller.Ki_rp = config.Ki_rp;
	controller.Kd_rp = config.Kd_rp;
	controller.Kp_rp = config.Kp_rp;

	controller.Ki_yaw = config.Ki_yaw;
	controller.Kd_yaw = config.Kd_yaw;
	controller.Kp_yaw = config.Kp_yaw;

	controller.max_gaz_drop = config.max_gaz_drop;
	controller.max_gaz_rise = config.max_gaz_rise;
	controller.max_rp = config.max_rp;
	controller.max_yaw = config.max_yaw;
	controller.agressiveness = config.agressiveness;
	controller.rise_fac = config.rise_fac;
}

pthread_mutex_t ControlNode::tum_ardrone_CS = PTHREAD_MUTEX_INITIALIZER;
void ControlNode::publishCommand(std::string c)
{
	std_msgs::String s;
	s.data = c.c_str();
	pthread_mutex_lock(&tum_ardrone_CS);
	tum_ardrone_pub.publish(s);
	pthread_mutex_unlock(&tum_ardrone_CS);
}


void ControlNode::toogleLogging()
{
	// logging has yet to be integrated.
}

void ControlNode::sendControlToDrone(ControlCommand cmd)
{
	geometry_msgs::Twist cmdT;
	cmdT.angular.z = -cmd.yaw;
	cmdT.linear.z = cmd.gaz;
	cmdT.linear.x = -cmd.pitch;
	cmdT.linear.y = -cmd.roll;

	// assume that while actively controlling, the above for will never be equal to zero, so i will never
hover.
	cmdT.angular.x = cmdT.angular.y = 0;

	if(isControlling)
	{
		vel_pub.publish(cmdT);
		lastSentControl = cmd;
	}
	lastControlSentMS = getMS(ros::Time::now());
}

void ControlNode::sendLand()
{
	if(isControlling)
		land_pub.publish(std_msgs::Empty());
}
void ControlNode::sendTakeoff()
{
	if(isControlling)
		takeoff_pub.publish(std_msgs::Empty());
}
void ControlNode::sendToggleState()
{
	if(isControlling)
		toggleState_pub.publish(std_msgs::Empty());
}
void ControlNode::reSendInfo()
{
	/*
	Idle / Controlling (Queue: X)
	Current:
	Next:
	Target: X,X,X,X
	Error: X,X,X,X
	*/

	DronePosition p = controller.getCurrentTarget();
	TooN::Vector<4> e = controller.getLastErr();
	double ea = sqrt(e[0]*e[0] + e[1]*e[1] + e[2]*e[2]);
	snprintf(buf,500,"u c %s (Queue: %d)\nCurrent: %s\nNext: %s\nTarget:
(%.2f,  %.2f,  %.2f), %.1f\nError: (%.2f,  %.2f,  %.2f), %.1f (|.| %.2f)\nCont.: r %.2f, p %.2f, g %.2f, y %.2f",
			isControlling ? "Controlling" : "Idle",
			(int)commandQueue.size(),
			currentKI == NULL ? "NULL" : currentKI->command.c_str(),
			commandQueue.size() > 0 ? commandQueue.front().c_str() : "NULL",
			p.pos[0],p.pos[1],p.pos[2],p.yaw,
			e[0],e[1],e[2],e[3], ea,
			lastSentControl.roll, lastSentControl.pitch, lastSentControl.gaz,
lastSentControl.yaw);

	publishCommand(buf);
}

void ControlNode::startControl() {
	isControlling = true;
	publishCommand("u l Autopilot: Start Controlling");
	ROS_INFO("START CONTROLLING!");
}

void ControlNode::stopControl() {
	isControlling = false;
	publishCommand("u l Autopilot: Stop Controlling");
	ROS_INFO("STOP CONTROLLING!");
```

```
}

void ControlNode::updateControl(const tum_ardrone::filter_stateConstPtr statePtr) {
	if (currentKI->update(statePtr) && commandQueue.size() > 0) {
		delete currentKI;
		currentKI = NULL;
	}
}

void ControlNode::clearCommands() {
	pthread_mutex_lock(&commandQueue_CS);
	commandQueue.clear();						// clear command queue.
	controller.clearTarget();					// clear current controller target
	if(currentKI != NULL) delete currentKI;	// destroy & delete KI.
	currentKI = NULL;
	pthread_mutex_unlock(&commandQueue_CS);
	publishCommand("u l Autopilot: Cleared Command Queue");
	ROS_INFO("Cleared Command Queue!");
}

bool ControlNode::setReference(SetReference::Request& req, SetReference::Response& res)
{
	ROS_INFO("calling service setReference");
	parameter_referenceZero = DronePosition(TooN::makeVector(req.x, req.y, req.z), req.heading);
	res.status = true;
	return true;
}

bool ControlNode::setMaxControl(SetMaxControl::Request& req, SetMaxControl::Response& res)
{
	ROS_INFO("calling service setMaxControl");
	parameter_MaxControl = req.speed;
	res.status = true;
	return true;
}

bool ControlNode::setInitialReachDistance(SetInitialReachDistance::Request& req,
SetInitialReachDistance::Response& res)
{
	ROS_INFO("calling service setInitialReachDistance");
	parameter_InitialReachDist = req.distance;
	res.status = true;
	return true;
}

bool ControlNode::setStayWithinDistance(SetStayWithinDistance::Request& req,
SetStayWithinDistance::Response& res) {
	ROS_INFO("calling service setStayWithinDistance");
	parameter_StayWithinDist = req.distance;
	res.status = true;
	return true;
}

bool ControlNode::setStayTime(SetStayTime::Request& req, SetStayTime::Response& res) {
	ROS_INFO("calling service setStayTime");
	parameter_StayTime = req.duration;
```

```
		res.status = true;
		return true;
}

bool ControlNode::start(std_srvs::Empty::Request& req, std_srvs::Empty::Response& res) {
		ROS_INFO("calling service start");
		this->startControl();
		return true;
}

bool ControlNode::stop(std_srvs::Empty::Request&, std_srvs::Empty::Response&) {
		ROS_INFO("calling service stop");
		this->stopControl();
		return true;
}

bool ControlNode::clear(std_srvs::Empty::Request&, std_srvs::Empty::Response&) {
		ROS_INFO("calling service clearCommands");
		this->clearCommands();
		return true;
}

bool ControlNode::hover(std_srvs::Empty::Request&, std_srvs::Empty::Response&) {
		ROS_INFO("calling service hover");
		this->sendControlToDrone(hoverCommand);
		return true;
}

bool ControlNode::lockScaleFP(std_srvs::Empty::Request&, std_srvs::Empty::Response&) {
		ROS_INFO("calling service lockScaleFP");
		this->publishCommand("p lockScaleFP");
		return true;
}
```

Bibliography

[1] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart. Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments. In Proc. of the International Conference on Robotics and Automation (ICRA), 2011.

[2] ARDrone Flyers. AR.Drone — ARDrone-Flyers.com, 2011. [http://www. ardrone-flyers.com/].

[3] H. Bay, T. Tuytelaars, and L.V. Gool. SURF: Speeded-up robust features. In Proc. of the European Conference on Computer Vision (ECCV), 2008.

[4] C. Bills, J. Chen, and A. Saxena. Autonomous MAV flight in indoor environments using single image perspective cues. In Proc. of the International Conference on Robotics and Automation (ICRA), 2011.

[5] J. Canny. A computational approach to edge detection. Conference on Pattern Analysis and Machine Intelligence, PAMI-8(6):679 – 698, 1986.

[6] H. Deng, W. Zhang, E. Mortensen, T. Dietterich, and L. Shapiro. Principal curvature-based region detector for object recognition. In Proc. of the Conference on Computer Vision and Pattern Recognition (CVPR), 2007.

[7] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: Part I. Robotics & Automation Magazine, 13(2):99 – 110, 2006.

[8] E. Eade and T. Drummond. Edge landmarks in monocular SLAM. Image and Vision Computing, 27(5):588 – 596, 2009.

[9] C. Harris and M. Stephens. A combined corner and edge detector. In Proc. of the Alvey Vision Conference, 1988.

[10] G. Muscato, F. Bonaccorso, L. Cantelli, D. Longo and C.D. Melita: “Volcanic environments: Robots for Exploration and Measurement”, IEEE Robotics & Automation Magazine, Vol.19, 2012.

[11] Pascual Campoy, Juan F. Correa, Ivan Mondragon, Carol Martinez, Miguel Olivares, Luis Mejias, Jorge Artieda: Computer Vision Onboard UAVs for civilian tasks. Computer Vision Group, Universidad Politecnica Madrid.

[12] "Direction Finding" *Wikipedia: The Free Encyclopedia*. Wikimedia Foundation, Inc. 22 July 2004. Web. 10 Aug. 2004.

[13] Braden R.Huber: "Radio Determination on Mini-UAV Platforms: Tracking and Locating Radio Transmitters", All Theses and Dissertations. BYU Scholars Archive (2009).

[14] Jakob J. Engel: "Autonomous Camera-Based Navigation of a Quadcopter". Technical University of Munich, 2011.

[15] "The iPhone: "Now There's a Helicopter for that". Associated Press in the New York Times, 2010.

[16] Chris Anderson: "Parrot AR.Drones specs: ARM9, Linux, 6DoF IMU, Ultrasonic sensor, WiFi…WOW!", 2010.

[17] Andrew Gibiansky: "Quadcopter Dynamics, Simulation and Control", Andrew.gibiansky.com, 2012. [http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics/].

[18] C Balas: "Modelling and Linear Control of a Quadrotor", Cranfield University, 2007.

[19] Stephane Piskorski and Nicolas Brulez: "AR.Drone Developer Guide", University of Washington Computer Science and Engineering Community, 2011.

[20] http://wiki.ros.org/ROS/Introduction

[21] Jason M.O'Kane: "A Gentle Introduction to ROS", https://cse.sc.edu/~jokane/agitr/

[22] Michael Montemerlo, Sebastian Thrun, Daphne Koller and Ben Wegbreit: "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem", Stanford, 2004.

[23] G. Klein and D. Murray: " Parallel tracking and mapping for small AR packages", In Proc. Of the International Symposium on Mixed and Augmented Reality (ISMAR), 2007.

[24] http://www.ros.org/wiki/ar_track_alvar

[25] https://github.com/artoolkit/artoolkit-docs/blob/master/3_Marker_Training/marker_about.md

[26] Martin Hirzer: "Marker Detection For Augmented Reality Applications", Institute for Computer Graphics and Vision, 2008.

[27] Alonzo Kelly: "Mobile Robotics: Mathematics, Models and Models", Cambridge University Press, 2014.

[28] Extended Kalman Filter, http://en.wikipedia.org/wiki/Extended_Kalman_Filter

[29] Engel, J., Sturm, J., Cremers, D., “Scale-aware navigationof a low-cost quadrocopter with a monocular camera, Department of Computer Science”, Technical University of Munich, Germany, Robotics and Autonomous Systems 62, pp. 1646–1656, 2014.

[30] Karl Johan Astrom: “Control System Design”, 2002.

[31] Sean Nicholls: “ARDroneDocs”, https://github.com/seannicholls/ARDroneDocs/wiki/Technical-Specifications.

[32] Karl Johan Aström and Tore Hägglund: “PID Controllers, Theory, Design and Tuning”, Instrument Society of America, 1995

[33] http://wiki.ros.org/tum_ardrone