Autonomous Landing of a Multicopter Using Computer Vision

Thesis of 30 ECTS submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Joshua Springer

Reykjavík University; Department of Computer Science
Mälardalen University; School of Innovation, Design, and Engineering

January 26, 2021

Acknowledgements

I would like to thank my family for their undying love, support, and encouragement which have gotten me this far.

My thanks go to Mälardalen University, Reykjavík University, and the Nordic Council, without whose generosity in the MDH Scholarship and the Nordic Master Programme, I would not have had the opportunity to study my passion in two wonderful countries.

Abstract

Takeoff, normal flight, and even specialized tasks such as taking pictures, are essentially solved problems in autonomous drone flight. This notably excludes landing, which typically requires a pilot because of its inherently risky nature. This project attempts to solve the problem of autonomous drone landing using computer vision and fiducial markers - and specifically does *not* use GPS as a primary means of navigation during landing. The system described in this thesis extends the functionality of the widely-used, open-source ArduPilot software which runs on many drones today, and which has only primitive functionality for autonomous landing. This system is implemented as a set of ROS modules which interact with ArduPilot for control. It is designed to be easily integrated into existing ArduPilot drone systems through the addition of a companion board and gimbal-mounted camera. Results are presented to evaluate the system's performance with regard to pose estimation and landing capabilities within Gazebo 9 simulator. Guidelines are provided for migration to a physical system as a gateway to future work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Autonomous drone landing is risky. The sensitive maneuvers and high navigational accuracy required for landing make this a task prone to crashes. As a consequence, robust solutions for autonomous drone landing are typically unavailable in the open source community, while robust, autonomous, open source solutions already exist for takeoff, waypoint-to-waypoint flight, and even other tasks such as photography and videography. Ultimately this means that landing is the main factor that precludes fully autonomous drone missions in most cases. The goal of this project is therefore to develop a robust solution to the problem of autonomous drone landing, which is supported by current, reliable software and technology, and can be easily integrated into existing drone systems.

Current landing solutions are primitive. One such solution, called `precision_land` [1], exists in the open source autopilot software *ArduPilot*. This method localizes the landing platform using only its x and y positions in the frame of a camera mounted in a fixed position, usually in line with the downward normal vector to the drone's body. It then attempts to maintain a constant rate of descent until it detects that it can no longer descend, at which point the drone is considered to have landed. This setup has multiple drawbacks. First, the downward-facing camera limits the field of view wherein the landing platform can be recognized. Furthermore, since the positional control of the drone is inherently based on the drone's attitude, positional changes (and inherent attitude changes) can also obfuscate the landing platform from the camera's field of view. Second, the primitive descent policy of simply maintaining constant downward velocity is not ideal. It would be preferable to base the descent rate on the altitude of the drone above the landing platform to allow for a quick, initial approach and slow, smooth contact with the landing platform. Additional tools such as LIDAR or infrared sensors can add this functionality, but they also complicate the system and require more power, communication, and calibration. Third, the fixed, downward-facing camera can often mean that the landing platform is too large to be identified when the drone is extremely close to landing platform, as it will not be contained entirely within the camera's field of view. A typical workaround is to assume that the drone is oriented correctly, and to blindly commit to a landing during this final, crucial stage. This method is inherently dangerous, as even slight errors or wind can cause fatal crashes. Fourth, the landing is controlled using only 2 components of the drone's positional displacement from the landing platform. This means that additional alignment of the drone to the landing platform's yaw orientation, for example, is impossible. Another primitive, autonomous landing solution is simply to navigate to a given waypoint using global positioning system (GPS) and to maintain the given latitude and longitude coordinates while descending to the

ground. This is feasible only in environments that are conducive to very strong, reliable GPS signal - such as an open field on a clear day. In less ideal environments - such as urban canyons, mountain valleys, or even open fields on a cloudy day - GPS alone cannot provide the required accuracy to land on a small platform, as typical GPS position estimates can sometimes vary from reality by as much as 3 meters [2].

Given the limitations and drawbacks of the existing autonomous landing methods, it is possible to outline the properties of a *robust* method:

1. The landing controller should be able to track the landing platform's pose over a wide range of relative distances and orientations. This will allow the drone to continue its landing process even if it does not approach the landing platform from a specific orientation.

2. The landing controller should not be primarily GPS-based, but should use some other method which allows for highly-accurate localization.

3. Dynamic control should be applied to all controlled components of the drone's position/velocity during landing, instead of arbitrary, static values such as a constant rate of descent.

4. At no point in the landing process should the drone blindly descend without verifying that its position relative to the landing platform is conducive to a successful landing. Landings should be halted in the event that a successful landing cannot be reasonably ensured.

5. The landing controller must be simple, in that it only uses a minimal set of tools - preferably tools that are in common use on current drone systems. This will make integration of the landing controller into physical drone systems easier in the real world.

The system described in this thesis is designed with the stated requirements in mind. It is targeted at the real world applications of landing a drone on a stationary platform or a moving platform such as the top of a vehicle. At a high level, the landing system is based primarily on computer vision, rather than GPS. The landing platform is fitted with a fiducial marker (explained in Section 1.2.2). A gimbal-mounted camera identifies and tracks the landing platform via the fiducial marker. The landing algorithm receives the location of the fiducial marker relative to the drone and calculates target velocities which direct the drone towards the landing platform. The landing algorithm then communicates these velocities to the flight control software. From the acquisition of the landing target to contact with the landing platform, conditions of the drone are monitored in order to determine whether the landing should be aborted or not. If the landing is aborted - for example in the event that the landing target is lost, the drone loiters in a stationary position using GPS and inertial measurement unit (IMU). Other, similar methods are described in Section 1.3.

More specifically, the landing system is developed in robot operations system (ROS) - an open source, modular robotics control framework with many existing modules. The system is built on the ArduCopter release of the wider ArduPilot code stack with the goal of minimal invasiveness. As the system will be used on a micro aerial vehicle (MAV), it will use the "common" dialect of the MAVLink communication protocol, which is the default communication protocol within ArduPilot. The ROS modules will be available to the open source community - links are provided in Section A.1.

1.2 Background

1.2.1 Autonomous Drone Flight

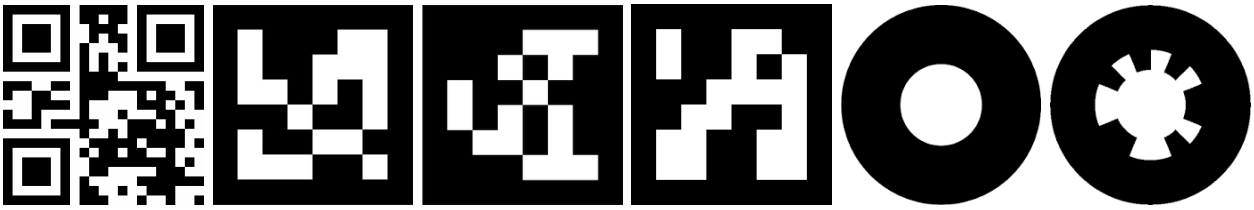
Robust hardware and software systems exist for enabling autonomous drone flight in the open source community. Several microprocessors have been developed for this purpose, such as the Pixhawk family [3]. Hardware additions, such as the Navio2 [4], have been developed as shields for the Raspberry Pi family. This hardware provides critical sensors to flight control software, such as positional data from a GPS, air pressure from a barometer (which helps to determine altitude), orientation data from an IMU, etc. Power modules regulate battery power to the microprocessor system and also provide useful information pertaining to the system's battery voltage and current draw. Other peripherals can be added through interfaces such as SPI, SBus, analog sensors, UART, 3-pin servo connections, and USB. Telemetry systems enable wireless communication between these vehicles and ground control stations, which provide high-level control such as "takeoff," "land," and point-and-click waypoint selection, as well as system parameter reconfiguration.

In the more primitive environment of a microprocessor such as the Pixhawk, a real time operating system executes the flight control tasks, which can guarantee that critical tasks happen on time. In a fuller operating system such as Raspbian (the Raspberry Pi operating system), the autopilot software typically runs as a system service. Running the autopilot software on a primitive system provides a better guarantee of real time performance of the software, but makes it more difficult to extend the software functionality. Running the autopilot software in a full operating system makes it far easier to add additional functionality by simply running additional programs, but this comes at the price of less real time performance. The autopilot software gathers data from the sensors in order to determine its vehicle's conditions, after which it can determine the proper way to control the vehicle's actuators in order to accomplish its goals. In multirotor drones, the main actuators that the flight control software manages are the motors, which control the drone's attitude. Other typical actuators include gimbals, which are used to aim cameras.

Currently there are 2 main autopilot software distributions which are available open source: ArduPilot and PX4. Different branches of these distributions target different vehicle families, such as quadcopters, hexacopters, octacopters, fixed-wing drones, ground vehicles, and even submarines. They use a lightweight communication protocol called MAVLink, which provides common message sets and allows for efficient data transfer between vehicles and ground control stations. MAVLink itself has several libraries which allow it to interface with other software, such as PyMAVLink (a Python implementation of the message set), MAVproxy (a Python MAVLink server), and MAVROS (which allows MAVLink-enabled vehicles to interface with ROS modules).

ArduPilot

ArduPilot is a popular implementation of the aforementioned open source autopilot software. It was chosen for this project because of its large user base (which provides quick testing and bug fixes), because of the familiarity of the author to the software in previous work, and because of time constraints in the context of this project. Learning a new autopilot system would slow down other development. However, the system developed in this thesis interacts with ArduPilot externally, without editing the ArduPilot code. Since the interfaces used by the system are common to both ArduPilot and PX4, the system could easily be ported to work with PX4. However, testing of this is beyond the scope of this thesis.



(a) QR Code (b) AR Tag (c) April Tag (d) ArUco (e) WhyCon (f) WhyCode

Figure 1.1: Common fiducial markers.

1.2.2 Fiducial Markers

Fiducial markers, examples of which are shown in Figure 1.1, have been used in computer vision in the recent past for computationally cheap, unambiguous determination of an object's orientation and position in space. These markers are easy to produce, as they are planar and can typically be printed in a range of sizes on a normal sheet of paper and be fastened to whichever surface needs to be identified. This makes fiducial markers a convenient solution for enabling a drone to precisely identify and approach a landing pad that is labeled with a fiducial marker. A well known fiducial marker is the QR Code, which is able to store much more data than typical fiducial markers in the robotics domain. However, this high data density gives the QR Codes an intricate design which is prohibitively difficult to fully process in many robotics applications wherein pose distortion, motion blur, and varying light conditions may obscure the image.

Within the robotics domain, fiducial markers tend to carry less data than QR Codes while being more robustly identifiable in a variety of conditions. Many different fiducial markers and corresponding identification systems are available, such as ARTag, AprilTag, WhyCon, and WhyCode. ARTag, a shortened form of *Augmented Reality* Tag, is a bi-tonal, square symbol consisting of a solid background with a 6x6 grid of high contrast interior cells which can encode 2002 different identifying codes [5]. One significant issue in applying ARTag is that its detection algorithm is not public, which makes further development difficult or impossible. Only general information about the detection algorithm for ARTag is available.

AprilTag is a very similar fiducial marker to ARTag, with the principal difference being that it is open source. AprilTags use the same general form as ARTag: a black square with a 6x6 interior region of black or white squares which encode a binary sequence and ultimately denote an identifier. The detection algorithm analyzes a given image using the image's gradient and attempts to find four-sided dark regions. The detection algorithm has, by design, a low false-negative rate and a high false-positive rate, and it is therefore coupled with a decoding algorithm that verifies whether the detected squares form a proper AprilTag identifier. While this coupling does increase the accuracy of reported AprilTag detections, it also means that an accurate reading of the identifier is a necessary part of the detection, which makes the AprilTag harder to detect in noisy conditions or from long distances [6]. Still, AprilTags have been used with high success in a wide variety of robotics applications to allow vision systems to identify and orient objects in 3 dimensional space. AprilTag detectors have been made open source and are freely available in both Python and C++.

Aruco is another fiducial marker following the black and white square paradigm of AR Tag and April Tag [7]. It has a black border and a configurable number of ID bits which determines the black and white interior pattern. Its algorithm works by first detecting and extracting the most prominent contours in its input image. Then, it analyzes the inner region of the contours in order to extract the marker's ID code. The image is converted to black and white, and each interior square region receives a value of 0 or 1 corresponding to the color of the majority of its pixels. The black

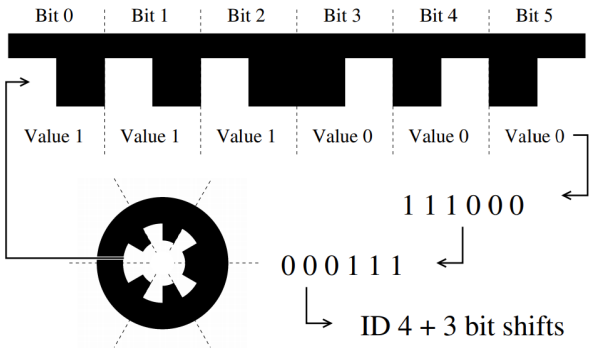


Figure 1.2: The WhyCode marker and "Necklace" encoding, from Lightbody et al..

border is a necessary part of the marker, so a check for this feature functions as a first rejection test. Then, the detected marker ID is compared to a dictionary of set IDs. If a corresponding ID does not exist in the dictionary, error correction is applied to determine whether the detected ID is within some distance to an existing ID. If a corresponding ID can then be found, the marker is detected and its pose is estimated.

WhyCon is a circular marker developed by Nitsche et al. at the University of Buenos Aires in 2015 [8]. The simple marker consists of an outer black circle whose center is covered by an inner white circle. The simplicity of the design makes the marker easy to identify even with high perspective distortion - such as, for example, when viewing it from an extreme angle. The detection algorithm is simple and computationally cheap. The relevant image is first searched for dark pixels, and flood-fill is then used to detect a contiguous region of dark pixels. The centroid of the dark region is then the starting point for a second search of a light region, which is flood-filled in a similar way. If the inner region is indeed lighter than the darker region, then the elliptical properties of the symbol are verified. Specifically, the detection algorithm verifies that the semi-axes and centroids of each region are roughly aligned, and that the ratio of the pixel areas of each region is roughly as expected in the known WhyCon tag. When applied to a video feed, the search for the black and white regions in frame n begins at the pixel positions where those regions were found in frame $n-1$, which decreases the computational load of the algorithm. A WhyCon marker's position and orientation in 2 dimensions can be detected very quickly. However, the drawback to WhyCon is that not all components of its rotational orientation cannot be determined, since it has full radial symmetry. Further, multiple WhyCon markers cannot be distinguished from one another because they do not contain identifiers. Still, it remains a useful and cheap marker. The WhyCon infrastructure is provided in a ROS module and is available in the open source community.

As an extension to WhyCon, Lightbody et al. developed the WhyCode marker in 2017. WhyCode uses the characteristic exterior, circular, black region and interior, circular, white region, but also uses a "Necklace" encoding, which is a Manchester encoding of a binary value that is wrapped around the interior white region. This is illustrated in Figure 1.2 (from [9]). This property of WhyCode markers allows them to break the radial symmetry of WhyCon markers, and also allows them to be distinguished from each other. Some patterns do still have radial symmetry with one another, as some WhyCode IDs have the same shape rotated by some angle. WhyCode markers use essentially the same flood-fill and ellipse-based detection system that WhyCon markers use, but with the added component of a Necklace decoder. The radial asymmetry allows the full spacial pose of the marker to be determined - including the rotational orientation (with some ambiguity based on ID, although this is conceptually able to be overcome). This added benefit makes the marker more appealing than

its WhyCon predecessor, while still retaining almost all of the WhyCon computational cheapness. Furthermore, the detection of WhyCode markers is independent from the decoding of the Necklace. This is important because it allows the marker to still be detected in cases where its more intricate encoding is obscured, such as in scenarios of high motion blur or extreme perspective distortion. It also gives the WhyCode marker an edge over the widely-used AprilTag, which must be detected simultaneously with its encoding.

1.2.3 PID Controllers

The most common form of feedback process controller is the Proportional-Integral-Derivative (PID) controller [10], and indeed these are the most common feedback controllers used in the ArduPilot software. Given a system with a variable to be controlled, an actuator to provide control effort in order to change the variable, and a sensor to read the state of the controlled variable, a PID controller can produce a smooth and predictable change in the state of the controlled variable, causing it to approach and arrive at a set point. A scalar (k_p, k_i, or k_d) controls the influence of each of the controller components (proportional, integral, or derivative, respectively) on the control effort. The effect of the proportional component is to apply a control effort which is proportional to the system's error - that is, the difference between the set point and the current state. This causes the state to approach the set point. The effect of the derivative component is to apply a dampening to the control effort which is proportional to the derivative of the error, in order to slow the state's approach to the set point. The effect of the integral component is to apply a control effort which is proportional to the integral of the error, the benefit of which is to remove persistent, steady state errors which must be identified over time and which are therefore not identifiable by the instantaneous readings of the error or its derivative.

The values of the gains must be tuned for each system to which a PID controller will be applied. Some methods exist to analyze systems in order to determine effective gains [10], however it is also possible to tune the gains manually with some experience. Aside from the gains, it is also necessary to set some other parameters, such as the minimum control effort, the maximum control effort, and the integral windup limit. The minimum and maximum control effort values provide a means of protecting the actuator or system from excessive control forces which may go beyond physical or process constraints. The integral windup limit sets the maximum effect that the integral component may have on the control effort, motivated by the fact that the integral component can easily cause saturation and unstable system response. PID controller tuning is a solved problem and is therefore not described in great detail in this project. [11] and [10] provide guidelines for this, such as the Ziegler-Nichols method.

1.2.4 Quaternions

Quaternions offer a reliable means of representing rotations in 3-dimensional space. The calculations required to manipulate them are more computationally efficient than matrix operations. Quaternions also do not suffer from the potential ambiguity inherent in representing rotations with 3-dimensional vectors storing values for pitch, roll, and yaw. Pitch, roll, and yaw (rotations in the transverse axis, longitudinal axis, and normal axis, respectively) are visualized in Figure 1.3, from [12]. These 3-dimensional vectors suffer from loss of information if 2 of the rotational axes become aligned (gimbal lock), and from not providing a single, unique representation for each possible normalized rotation. For these reasons, quaternions are the default data object used to describe rotations in robotics and avionics. A detailed mathematical examination of quaternions is beyond the scope of this project. However, the main ideas behind using quaternions to represent rotation

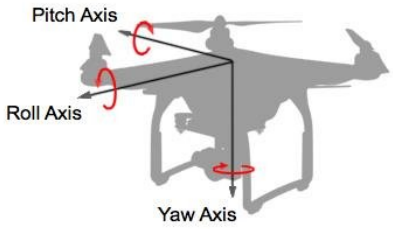


Figure 1.3: Visualization of pitch, roll, and yaw.

are that they can efficiently be composed with one another, and they represent rotation unambiguously. Mathematically rigorous definitions and explanations of the operations that are relevant to quaternions, as well as the history therein, can be found in [13].

The Transform 2 library (`tf2`) provides implementations of these data structures and the relevant operations within the ROS infrastructure. The Geometry Messages library (`geometry_msgs`) provides message definitions for efficiently representing quaternions in message passing. This project employs these libraries instead of re-inventing the wheel in this area.

1.2.5 Robot Operating System

ROS (Robot Operating System) [14] is an open source framework for developing modular, flexible, robust robotic software. A large variety of modules, libraries, and conventions are included in the ROS umbrella. Functional parts of robotic applications are divided into ROS modules and libraries, whose definitions provide lists of dependencies and installation instructions, as well as application code. The `roscore` node (process) provides a means for additional nodes to communicate using *topics*. Topics can be of any data type - such as Boolean variables, integer variables, float variables, images, custom types, etc. Nodes are launched either independently or using `.launch` files, which specify pre-launch requirements (such as prerequisite nodes), launch parameters, and which may launch additional nodes. A key benefit to using ROS is that there are many existing packages supplying many different functionalities, such as PID controllers (see Section 1.2.3), fiducial marker frameworks (see Section 1.2.2), drivers for cameras (which provide the camera's image as a topic), and many useful data structures such as quaternions (see Section 1.2.4), vectors. Another important library is the Transform 2 library (TF2) [15], which provides invaluable support for coordinate system transforms. Using this library, nodes can generate time-stamped representations of the positions and orientations of relevant physical components of the robot. These transforms are tagged by name and organized into a tree, after which point, they can be composed with one another to derive further transformations. This is useful in determining the positions of components on the robot itself, and also objects in space.

1.2.6 Gazebo Simulator

Gazebo [16] is an open source simulation tool developed for Linux. It provides a 3-dimensional environment and a physics engine which allow users to simulate interaction between objects and the environment. Models and worlds can be defined and edited using `.sdf` and `.world` files respectively. These files allow users to specify the defining characteristics of each model, such as collision planes which determine how models can make contact with other models or other components within the environment. Gazebo provides interfaces for simulated sensors, such as inertial measurement units, cameras, LIDAR units, etc., and these sensors can be included in `.sdf`

files in order to provide the relevant information to the model. Plugins can also be included in both `.sdf` and `.world` files in order to provide additional functionality, such as PID controllers and simulated wind. Several other open source simulators are also available. However Gazebo's existing, readily-compatible plugins for both ArduPilot and ROS make it a good candidate for this project.

ArduPilot Gazebo Plugin

Some Gazebo models and worlds have been developed to allow ArduPilot to interact with Gazebo. They are included in [17]. For example, the Iris quadcopter (used in this project) is a small quadcopter fitted with a gimbal. The gimbal is a simple, 2-dimensional gimbal with two "revolute" joints - a yaw joint referred to as `iris_gimbal_mount`, and a pitch joint referred to as `tilt_joint`. [17] specifies an `iris_ardupilot` world file which includes the Iris quadcopter and defines a planar world with a runway. A Python script `sim_vehicle.py` provides an interface between the Iris and an instance of ArduPilot. The script reads the data from the simulated sensors aboard the Iris and provides throttle signals to its motor plugins. The motor plugins provide simulated thrust in order to realize the effects of the control system and animate the drone model. These allow the drone system to accurately simulate the behavior of the drone as it is controlled by ArduPilot.

Integration of ArduPilot, Ground Control Stations, and Gazebo

ArduPilot can fully interact with and control the Iris quadcopter in Gazebo. The Iris quadcopter model is, as well as several others, specifically set up to mimic the mechanics of an actual quadcopter, and ArduPilot controls the thrust values applied to the motor mounts of the quadcopter. As in a real life scenario, this is the only means of controlling the attitude, velocity, and acceleration of the drone. An instance of ArduPilot can be launched in order to interact specifically with the Iris within Gazebo using the following command: `sim_vehicle.py ArduCopter -f gazebo-iris`. This launches the "copter" version of ArduPilot which is further configured using the `gazebo-iris` configuration file. ArduPilot further opens User Datagram Protocol (UDP) ports in order to communicate with Ground Control Software (GCS) instances using the MAVlink communication protocol. In this project, care has been taken to leave both the ArduPilot software and MAVLink protocol unmodified, such that it may be applied to existing drone systems without deep modification.

Several different GCS programs are also available open-source. These typically offer a GUI for visualization and temporal tracking of a vehicle's position in a map, forms for adjusting various parameters, displays for sensor values, and, importantly, a concise, GUI-based method of sending MAVLink commands to the vehicle. The QGroundControl software [18] is the software chosen for interacting with the simulation in this project purely because of its simplicity of installation. Alternatives would also work, but the particular choice of GCS software is unimportant.

1.3 Related Work

Wynn [19] has developed a method for landing on a moving platform using fiducial markers to track the landing platform, with the initial aid of GPS. A larger marker allows recognition of the landing platform from far distances. A smaller marker of the same form is embedded inside the larger marker to allow for identification at close distances. After the landing platform is localized, different control states direct the drone's approach towards the marker - first causing the drone to approach quickly in the x and y dimensions, while maintaining a sufficient altitude above the marker (in the z dimension), and then gradually lowering to a small distance above the marker. At this

point, the drone commits to a landing and lowers itself until detecting a successful landing, since the proximity of the camera to the landing pad means that the marker is no longer fully contained within the field of view of the camera, and thus can no longer be tracked. Other control states include switching from *patrol mode* to *tracking mode* once the relevant marker has been detected continuously for a small amount of time, and aborting a landing if the marker has not been detected for 2 seconds continuously. This method also takes into account the swaying of the landing platform itself, which is mounted on a barge.

Borowczyk et al. [20] have implemented a system allowing a DJI Matrice to land on a golf cart using GPS with wirelessly transmitted position. The drone uses a proportional navigation (PN) system for initial approach. This initial approach is carried out at a fixed altitude and a gimbal-mounted camera is used for initial detection of a fiducial marker mounted to the landing platform. A fixed, downward-facing camera then detects the visual fiducial marker and a PID controller manages close-range approach. A constant descent velocity is set during the final phase of landing. This method allows for successful landing on a platform moving at speeds of up to 50 km/h. Recommended future work includes using multiple fiducial markers of different sizes to identify the landing platform, as well as a single gimbal-mounted camera instead of the dual camera setup.

Falanga et al. [21] outline a method for landing a quadrotor running the PX4 autopilot software on a moving platform indoors. The landing platform is fitted with a specific marker made up of a cross and a circle. The drone uses 2 cameras, the first mounted straight down from the drone, and the second mounted at a 45°angle down and towards the front of the drone. The images from these cameras are used to solve a perspective-n-point (PnP) problem which finds the relative pose of the landing platform's marker. A distance sensor helps to scale the vision-based pose estimation. The onboard computer determines optimal approach trajectories using this information. A Kalman filter makes the process robust to missed detections and helps to determine the velocity of the landing platform. Successful landings were reported with the landing platform moving up to 1.2 m/s.

Wubben et al. [2] use the typical setup of a hexacopter drone with a single camera in a fixed, downward facing orientation to identify a landing platform via 2 ArUco markers. A Pixhawk controls the drone using ArduPilot, and a Raspberry Pi handles image processing and fiducial identification. The method reports successful and accurate landings, but also occasional failures due to visual loss of the landing platform. This visual loss was caused by sudden gusts of wind which pushed the drone away from the landing platform and out of the fixed camera's field of view.

Pluckter et al. [22] have developed a method for precisely landing a drone in an *unstructured* environment, which is to say an environment that has not been significantly artificially marked. On takeoff, the drone visually captures key points of interest in its environment. It then performs its mission and returns to a location above its takeoff location using GPS. Subsequent visual analysis of the surroundings and comparison of this information with the similar information captured at takeoff allow the drone to localize itself. This process is continued throughout the entire landing.

Polvara et al. [23] introduce a method of training and testing deep Q-networks (DQN) for landing drones in a simulated environment with simple outputs (left, right, forward, back, land), feeding the networks low-resolution images as input. Multiple networks were trained for specialized tasks, such as policy control, approach, and descent. The method performed with only slightly less accuracy than the conventional vision-based methods which use fiducial markers. However, the caveat is that the conventional methods can fail when the fiducial marker cannot be detected, whereas the method presented by Polvara et al still achieved a relatively high success rate.

Patrick Irmisch [24], in his master thesis at the Technical University of Berlin, compared distance estimation of AprilTags and WhyCon markers (Shown in Figure 1.1) using both monocular and stereo computer vision algorithms. This was done in an effort to use computer vision as a way

to estimate the distance between two trains as one approached the other for coupling. The markers were printed such that they occupied the same area, and multiple tests were carried out to determine the error in distance estimation when viewed from angles of 0°, 30°, and 60°. The stereo vision methods were tested using multiple matching algorithms, including the PnP, and semi-global mapping (SGM). The distance from the camera to the markers was estimated by these methods when the true distance varied between 10 and 80 meters. Distance estimation (that is, distance on the z-axis) is arguably the hardest part of a pose to estimate, since it involves composite measurements and calculations. Positions on the x-axis and y-axis can be easily determined through simple pixel analysis, and distortion can be used to measure the angle between the camera and the fiducial marker in each axis when the form of the fiducial marker is known. However, z-axis distance estimation is influenced by the distance itself, since at longer distances the fiducial marker occupies less pixels, meaning that the resolution between different distances is obscured. All of the fiducial markers show some robustness in their ability to be detected even when viewed from a far distance or from an angle. The real-world experiment showed that the distance from the camera to a WhyCon marker could be estimated with less error than that of an AprilTag marker. The WhyCon marker can also be detected from farther away, whereas the AprilTag marker was not detectable at a distance of 40 meters. The WhyCon marker was detected at a distance of 40 meters even at a viewing angle of 60°. Simulations showed more robust detection of both AprilTag and WhyCon markers, but this is not particularly important in the context of the real-world. Uncertainty in distance estimation actually increased in the tested stereo vision methods, likely owing to the doubled uncertainty in camera calibration parameters and camera orientation. However, a small decrease in distance estimation error was attained when combining both monocular and stereo methods. Ultimately, the results imply that a single WhyCon marker is the most suitable for relative position estimation between vehicles. Irmisch goes on to recommend that WhyCon markers be used with a stereo vision setup, with SGM as the detection algorithm.

Guo et al. [25] illustrate a monocular pose estimation system which is used to estimate the position of a multicopter indoors and without GPS. The given scenario involves a drone determining its position above a mat on the floor with several April Tag markers printed on it, each with a different identifier. The single camera is fixed to point facing straight down from the drone. Their system uses a set of translation and rotation matrices to calculate the position of the drone in the indoor flying space. The translation and rotation matrices from the camera to the tag are computed when the detection algorithm for the AprilTag identifier determines the pose of the AprilTag with respect to the camera. These are converted from pixel distances to real-world lengths using the intrinsic parameters of the camera. After these are determined, the position P_M of the drone on the map can be calculated directly because the rest of the values are constant. The accurate position of the drone is then derived from P_M using a Kalman filter to reduce noise. Detailed results are not presented, but the drone is able to fly from tag to tag following a planned path with high accuracy, estimating its position only from identifying and determining the relative position of April Tag markers in the map.

1.4 Structure of this Thesis

The motivated design and data flow of the gimbal and landing controllers is presented in Chapter 1, along with the supporting software infrastructure, and conditions of the simulation environment. Quantitative analyses of the performances of these systems within the simulator are presented in Chapter 3. Guidelines for the migration of this system from the simulator to a physical drone are presented in Chapter 4.

Chapter 2

Methods

2.1 Landing System Requirements and Design Overview

The landing controller must be robust to the positional noise apparent in normal GPS modules, which can cause mishaps during landing, meaning it must have a positional accuracy on the order of about 0.2 meters when targeting a landing platform with a diameter of 1 meter. It must also allow for landing on both stationary and moving landing platforms. To this end, the landing controller does not use GPS as its main navigational source, but instead uses fiducial markers as mentioned in Section 1.2.2, to estimate its position with respect to the landing platform. In order to land safely, the landing controller must have a way to identify and stop a failed landing, which could be caused by erratic landing platform movement, drift due to wind, obfuscation of the landing platform's fiducial markers, etc. In order to accomplish this, a descent region is generated, as shown in Figure 2.14. The radius of this region increases exponentially in the altitude of the drone above the landing pad. Outside of this region, the drone is not allowed to descend. This gives the drone time to correct its horizontal position and maintain a safe descent. This is outlined in Section 2.4.5.

The landing platform must be recognizable under a wide variety of conditions. A gimbal-mounted camera increases the range in which the landing platform can be identified by the camera. This requires the addition of a gimbal controller which aims the camera for the duration of the landing, explained in Section 2.3. A large WhyCon marker allows for easy recognition of the landing platform from long distances, while a smaller April Tag marker provides continuous pose estimation throughout the final descent, when the WhyCon marker is not entirely visible in the camera's field of view.

The landing controller and gimbal controller must be easy to integrate into existing drone systems. As mentioned in Section 1.2.1, the chosen autopilot software is ArduPilot, which uses the MAVLink communication protocol. To allow the proposed landing system to be easily integrated into existing drone systems, the ArduPilot code and MAVLink dialects are not edited. The landing controller and gimbal controller are developed as ROS modules which can be run on a companion board, interfacing with ArduPilot for control. On more advanced flight controllers, such as Navio2-enabled Raspberry Pis (which have full operating systems), these ROS modules can be run on the same board as the ArduPilot software, as system services. In order to abstract from primitive motor throttle commands and specific drone body types, the landing controller controls the drone using only high-level velocity set points. This leaves the low-level motor control to ArduPilot itself. The hardware tool set is also kept simple (just a gimbal-mounted camera and a companion board) for this requirement.

2.2 System Evaluation through Simulation

Simulation is a fast and cheap way of testing drone control algorithms, as it cuts out the overhead of physical components and the risk of fatal, expensive crashes. However, it does have the added overhead of adapting a specific model to a specific simulation environment, which can be a daunting task. Furthermore, the results of the simulation - particularly the model physics - likely will not translate directly to the real world. Ultimately this means that simulation can be aptly used for testing the landing system at a high level, but that transitioning to real world flight will involve incremental re-testing. In other words, simulation is only a good first step.

2.2.1 Setup and Tools

The Gazebo 9 simulator and the ArduPilot-compatible Iris quadcopter drone model (and included gimbal model) are provided in the open source community, as discussed in Section 1.2.6 and Section 1.2.6 respectively. These systems have existing integration with each other and with ROS, making them an ideal combination of tools for this project. The development of a new drone model for simulation is beyond the scope of this project. However, even though there is a slight discrepancy in that the simulation model is a quadcopter and the destination platform is a hexacopter, the proposed landing system is abstracted enough to overcome this difference. All of the commands sent by the landing controller are target velocities rather than primitive controls such as throttle signals to specific motors. The frame (quadcopter or hexacopter) can be manually set in the ArduPilot on-board firmware.

Simulation World

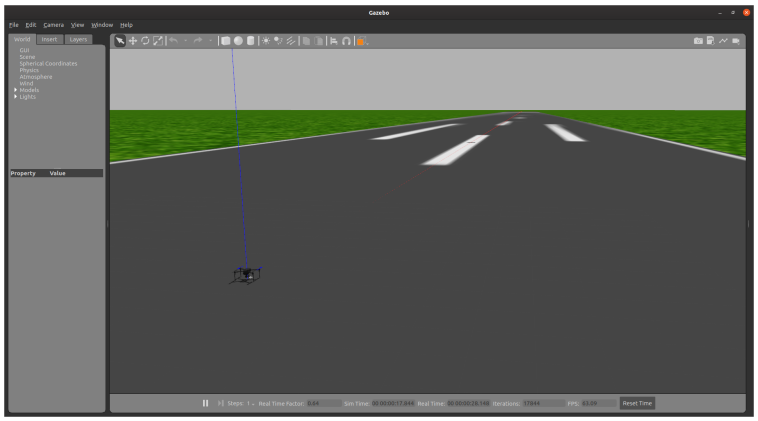


Figure 2.1: Gazebo GUI with the Iris quadcopter model.

The simulation uses Gazebo 9 as its base, as shown in Figure 2.1. However, in order to interface with ROS, Gazebo is launched through ROS itself, with the following command: `roslaunch gazebo_ros sandbox.launch`, where `sandbox.launch` opens a "sandbox" world created for the purpose of experimentation. It is an edited version of the `iris_ardupilot.world` which includes a landing platform made from fiducial markers, explained in Section 2.3.5. It includes the specification for the Iris quadcopter, along with an edited version of the Iris' gimbal. The edited version of the gimbal adds a weightless, simulated camera sensor that provides its detected image to an instance of ROS as a ROS topic. This particular sensor is available from the `gazebo_ros_pkgs` repository. This is necessary because the included gimbal model does not include a camera sensor.

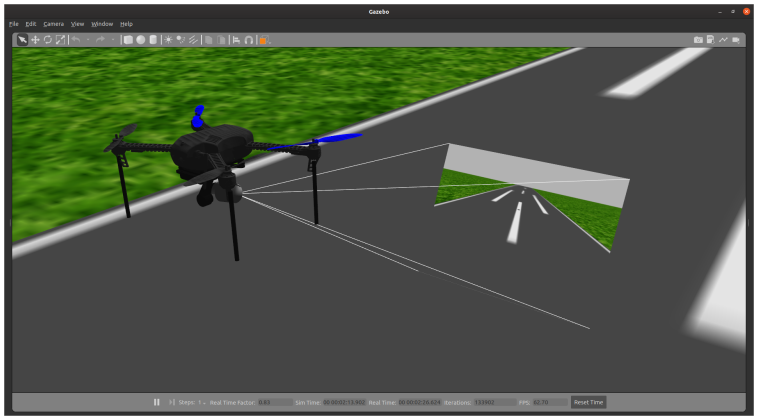


Figure 2.2: The view of the drone's camera in Gazebo.

The typical paradigm for adding a simulated sensor (such as this camera) to a Gazebo model involves adding not only the sensor itself, but also a "link" component which contains the sensor and inherently has mass. This paradigm was not followed here because the addition of such a link with any mass, including, zero and near-zero mass values such as 10^{-10} kg, caused prohibitively unstable in-flight behavior of this particular model. The weightless camera sensor was thus attached simply to the gimbal's `tilt_link`. The definitions for the edited models and worlds are available at [26]. The simulated camera view is shown in Figure 2.2. This visualization allows developers to have an intuitive sense of the drone's field of view, which is especially important in a situation like this, when the camera rotates on a gimbal.

GCS Software

The chosen GCS software for this project, as mentioned in Section 1.2.6, is QGroundControl, which is depicted in Figure 2.3. There are several GCS alternatives available open source. This particular one was chosen purely because of its ease of installation, but this choice is not important to the outcome of this project. It is only necessary to have some GCS software, as it provides easy manual control of the drone in the absence of a conventional radio remote controller during simulation. This is important particularly in the early stages of testing, to perform ad hoc testing and to determine strategies for more formal, automated testing as will be described in Chapter 3.

2.3 Gimbal Controller

2.3.1 Overview

In experiments using drones with a mounted camera, it is typical to see a camera with a fixed mounting angle. This provides simplicity in the estimation of the drone's pose relative to some fiducial marker, but limits the ability of the drone to detect the marker, as the drone itself must be facing the marker in order to detect it in the first place. A key aspect of this project is the development of a method for pose estimation of a drone relative to a fiducial marker using a gimbal-mounted camera for a wider range both of relative pose between the marker and the drone, and a wider range of acceptable behaviors for the drone during landing. This requires a slightly more involved system for coordinate transforms, as well a method for aiming the gimbal at the marker and tracking it over time.

Figure 2.3: The QGroundControl software.

2.3.2 Data Flow

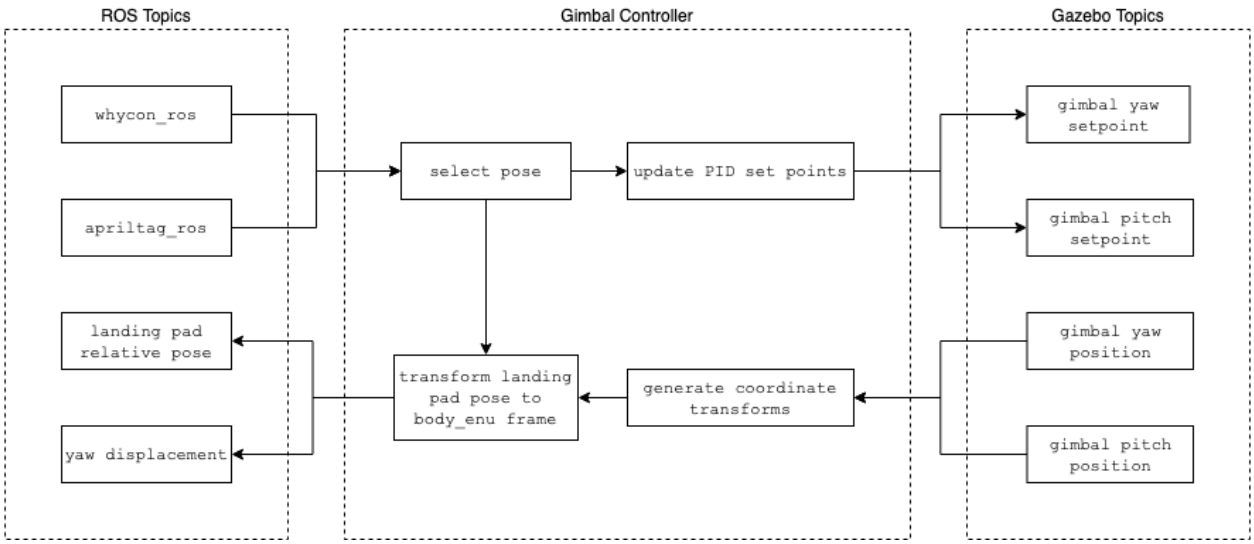


Figure 2.4: Gimbal controller data flow diagram.

The diagram in Figure 2.4 gives a high-level overview of the gimbal controller's functionality. The WhyCon and April Tag modules publish the pose of their possibly identified fiducial markers after analyzing the image from the simulated camera. If either module finds no marker in the image, then that module publishes no pose. If both markers are detected at the same time, only the April Tag marker is selected, and if only a WhyCon marker is detected, then it is used. The gimbal controller then updates the PID set points according to Equation 2.1 and these are used by the PID controller instances in Gazebo. The gimbal controller subscribes to the true yaw and pitch positions of the gimbal, provided by Gazebo, which are not necessarily equal to the set points. These gimbal yaw and pitch positions provide the necessary information to generate the coordinate transform from the camera frame to the body "ENU" (East, North, Up) frame. The landing platform pose is then

transformed to the body ENU frame and is published as its own topic. The "yaw displacement," denoting the angle between the north axis of the drone and the north axis of the landing pad, is also published as a topic.

The typical NWU (North, West, Up) coordinate system of Gazebo is visualized in Figure 2.5, with the red axis line pointing to the drone's "north," the green axis line pointing to the drone's "west," and the blue axis line pointing up. This is in contrast to the typical ROS East-North-Up (ENU) coordinate system, in which the axis lines in Figure 2.5 would be rotated by 90°clockwise in the "up" axis, so that the red axis points east, the green axis points north, and the blue axis points up. These lines represent the positive directions of their respective axes.

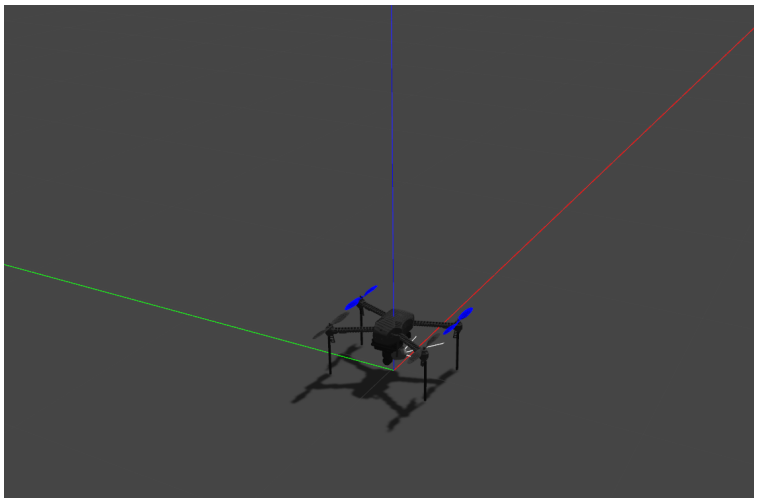


Figure 2.5: NWU (North, West, Up) Coordinate System within Gazebo.

2.3.3 PID Controllers for Gimbal Control

PID systems (outlined in Section 1.2.3) provide a natural solution to the problem of aiming the gimbal, with the goal being to keep the marker in the center of the camera's field of view at all times. It would also be natural, therefore, to control the position of the camera based on the pixel positions of the detected fiducial markers. However, both the April Tag and WhyCon systems provide the positions of their markers in the form of a pose (*not* pixel locations) - a specification of the linear translation and spacial orientation from the viewpoint to the marker. These poses are composed of a 3-dimensional vector providing the translation in (simulated) meters, and a quaternion providing the orientation. It is simple to control the position of the camera using the poses of the detected fiducial markers - instead of using pixel locations - in order to avoid additional edits to the already existing code, and also to avoid potentially required specificity to any hardware system (cameras may have different pixel ranges and therefore different centers).

Gazebo Plugin Implementation for Gimbal Control

A Gazebo plugin was added to the simulated drone's gimbal provide PID control of the orientation of the gimbal and camera. The plugin is defined as a ROS module, is included in the model's definition `.sdf` file, and its library is loaded by Gazebo at runtime. This plugin subscribes to "setpoint" topics that are provided by the `gimbal_controller` ROS module. As the gimbal has 2 degrees of freedom (yaw and pitch), 2 corresponding PID controllers set the orientation of

their respective gimbal axes. The gimbal has artificial limits of $\theta_{yaw} \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ and $\theta_{pitch} \in [0, \pi]$ for simplicity. $\theta_{yaw} = 0$ describes a situation where the camera is pointing directly forward (with respect to the drone's local coordinate frame), and θ_{yaw} increases as the camera rotates clockwise. $\theta_{pitch} = 0$ describes a situation where the camera is pointing forward, directly level to the drone. θ_{pitch} increases as the camera rotates towards vertical-down. The 2 PID controllers are able to control the position of the camera only in these ranges, which helps the system to avoid unnecessary rotations during simulation and also mimics the natural limits of physical systems. The gains of these systems were determined experimentally and will be described in Section 3.5. The plugin also publishes the true angles of the gimbal in each dimension as ROS topics for the subsequent calculation of coordinate transforms, explained in Section 2.3.6.

2.3.4 Aiming the Camera

The gimbal controller has callback functions for both WhyCon and April Tag detections, where it updates the set points (target states) of both PID controllers. Let SP_x be the set point of the gimbal's yaw axis, and let SP_y be the set point of the gimbal's pitch axis. The yaw and pitch axes of the gimbal can be visualized in Figure 2.2, where the yaw of the gimbal is in line with the yaw of the drone ($\theta_{yaw} = 0$), and the pitch of the gimbal is slightly less than that of the drone ($\theta_{pitch} > 0$). Then, upon detection of a fiducial marker, the set points are updated as in equation 2.1:

$$\begin{aligned} SP_x &= SP_x - k\frac{x}{z} \\ SP_y &= SP_y + k\frac{y}{z} \end{aligned} \tag{2.1}$$

where k is an experimentally determined scalar, and x, y, z are the linear components of the marker's pose in the west, down, and north axes of the camera respectively. The callback functions *increment* the set points instead of calculating them directly. The difference in the sign of the increment occurs because of the specific nature of Gazebo's axis conventions. This particular aspect would require special attention and verification when transitioning to a real world environment. The increment is scaled inversely to z in order to ensure that the system works over a range of distances.

2.3.5 Landing Pad Design and Prioritization of Fiducial Marker Detections

As shown in Figure 2.6, the landing pad is made up of two markers. The system recognizes the larger (1 meter in diameter), more easily identifiable WhyCon marker before it recognizes the April Tag marker (with side length 0.3125 meters). However, since the April Tag marker is used for final descent at close range, the gimbal controller prioritizes the April Tag marker over the WhyCon marker. So in the case that both markers are visible, the gimbal controller aims the camera at the April Tag marker. If the April Tag marker becomes obstructed, the gimbal controller then aims the camera at the WhyCon marker again. This prioritization is illustrated in Figure 2.7a through 2.7d. In the detection windows, detected WhyCon markers are highlighted, and detected April Tag markers have their IDs printed above the marker itself. The fact that the April Tag marker is flat on the landing pad's plane does mean that, at a low altitude, it will be somewhat difficult to recognize the April Tag's pose. However, this is a challenge to overcome anyway, as it keeps the landing pad simple and completely flat (and therefore free of obstructions). The center of the April Tag marker is positioned 0.75 meters north of the center of the WhyCon marker. This allows for the marker to be adequately close to the drone during final descent that it can always be easily recognized. However it also means that the April Tag marker is adequately displaced from the camera to always be in the camera's field of view. This distance is likely to be adjusted in real world experiments.

Figure 2.6: Landing pad design.

The original design for the landing platform included only a single WhyCode marker, but this was abandoned for reasons that will be discussed in Section 3.3.1.

A Gazebo model of the landing pad allows for control of the landing pad's velocity. It is essentially the same as the original landing pad, with the difference that the fiducial marker textures are added to a model with two collision planes, to simulate an entity separate from the ground. The friction is set quite high on the top collision plane (`mu` = `mu2` = 100000), in order to avoid artificial slipping of the drone on the landing pad, which would not happen on a properly designed physical landing pad. The friction on the lower collision plane is set to 0 in order to avoid unintended bouncing and extraneous drag effects which result from interaction with the ground collision plane. A separate Gazebo plugin controls the linear and angular velocity of the landing pad via ROS topics. It is necessary to continually set the velocity of the landing pad even in stationary landing situations, as the forces applied by the drone upon contact can cause the landing pad to sink into the ground, even with well-defined collision planes. The drone then detects this movement and attempts to correct, resulting in unintended behavior. Setting the velocity of the landing pad avoids this issue. In `sandbox.world`, the parameter `contact_surface_layer` was increased to 0.01 in order to avoid further bouncing between the drone and the landing pad itself. The landing platform model is shown in Figure 2.8. The size of the WhyCon marker (where the drone is supposed to land) is 1 meter in diameter, while the drone legs are roughly positioned at the corners of a rectangles with side lengths 0.44 meters and 0.26 meters. This allows the drone to fit entirely on the WhyCon marker with space to spare, while still keeping the landing platform small enough to be manageable in the real world, in scenarios where it can be mounted to the top of a car or bus, or simply positioned on the ground.

The landing pad velocity controller plugin subscribes to a topic `/landing_pad/cmd_vel` of type `geometry_msgs/Twist`, the components of which specify the components of the landing platform velocity in each of the 3 dimensions. A callback function within the plugin sets the values

(a) WhyCon detection only.

(b) No April Tag detection.

(c) Off-center WhyCon detection.

(d) Centered April Tag with ID “0” overlayed.

Figure 2.7: Prioritization of fiducial marker detections.

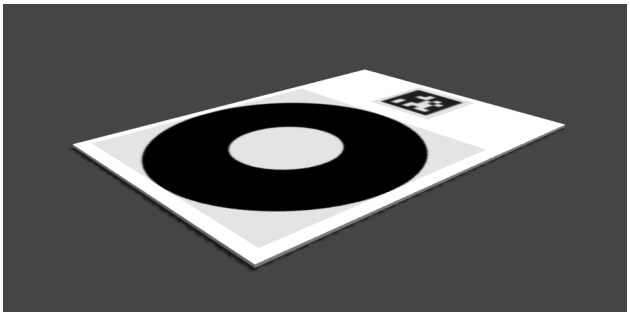


Figure 2.8: The separate landing pad model.

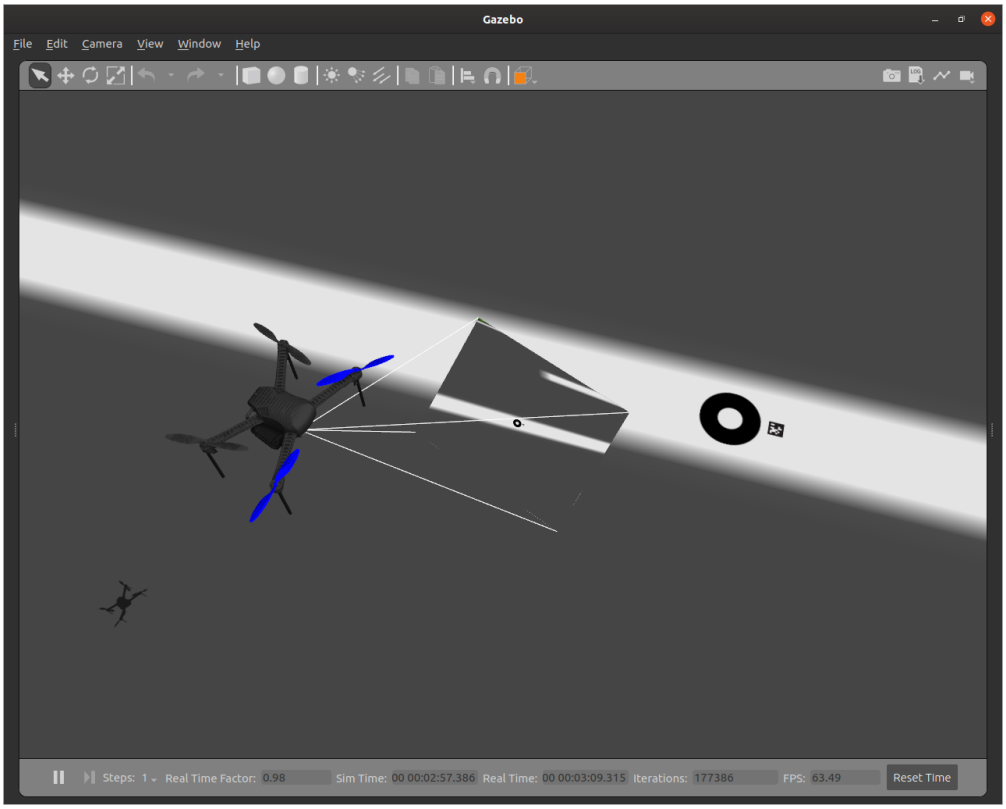


Figure 2.9: Illustration of the drone after detecting the landing pad.

in the Gazebo engine, and also sets the value of the landing pad's angular velocity to 0 in all directions. This conforms to the assumption that the velocity of the landing pad will be predictable and cooperative from the point of view of the drone. The velocity of the landing pad is set manually during testing, from the terminal using the `rostopic pub` command. This is published at a high frequency in order to negate extraneous physics effects.

2.3.6 Coordinate System Transforms

The second function of the gimbal controller is to generate coordinate system transforms based on the position of the gimbal. This is necessary because the camera is almost never directly in line with the drone's axis system, as shown in Figure 2.9. The blue propellers indicate the front of the drone, and the camera is offset counter-clockwise and down so that it may point at the detected fiducial markers.

The role of the transforms in this scenario is to take into account the rotational displacement of the camera with respect to the drone's intrinsic, local axes, so that the pose of the landing pad can be calculated in terms of the drone itself. As a simple example to illustrate this point: if the camera is rotated to $\theta_{yaw} = -\frac{pi}{2}$ and $\theta_{pitch} = \frac{pi}{4}$, and a marker is detected 10 meters forward (in the camera's coordinate system), the gimbal controller generates a transform that will place the marker 10 meters to the left of the drone. Since the gimbal has only 2 degrees of freedom, the gimbal controller can generate the relevant transform using only the supplied pitch and roll topics from the gimbal controller plugin. This structure corresponds to a real world scenario wherein the gimbal has a mounted IMU from which the corresponding values can be extracted.

Although the yaw position of the camera is a truly necessary component of determining the

relative pose with a rotating gimbal, the pitch is not. A second method of pose estimation in this scenario is therefore to "straighten" the detected pose, so that the orientation is in the reference frame of the fiducial marker's normal vector. The result is that the marker's rotation is no longer needed in subsequent calculations, and the translational elements of the pose correspond to the distances in the 3 dimensions of a coordinate system centered on the marker. This can be accomplished by simply creating a transform whose translational element $t =< 0, 0, 0 >$ (no translation), and whose rotational element is the inverse of the marker's rotation in the original pose frame. After this transform is generated, a second transform corrects for the yaw displacement of the camera. In the case of a WhyCon marker which has no intrinsic origin in terms of yaw, a second transform corrects for the yaw displacement of the camera by adding a rotation equal to the inverse of the camera's yaw. In the case of a WhyCode or April Tag marker which do intrinsically have unambiguous yaw orientations in their pose, the yaw used to correct the pose is equal to the difference between the gimbal's yaw displacement and the marker's yaw displacement. The inverse of this value is the rotational component of the second transform. If θ_1 represents the yaw of the camera, θ_2 represents the yaw of the marker, and θ_c represents the yaw used to correct the pose, then θ_c can be simply calculated as follows in equation 2.2. In the case of a WhyCon marker, θ_2 is just 0.

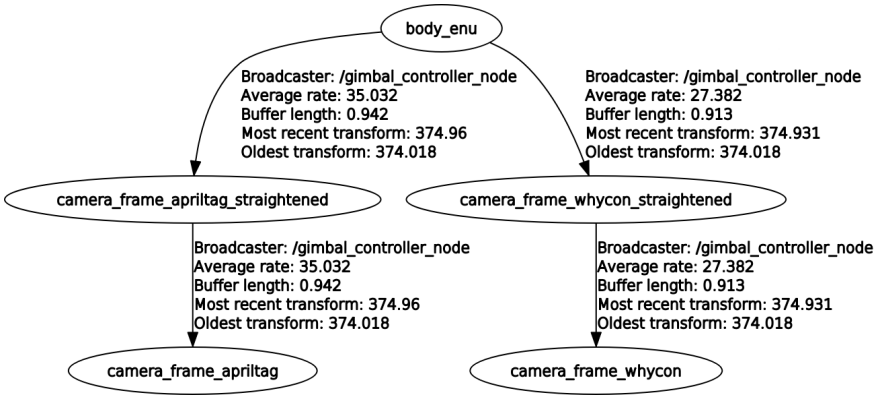


Figure 2.10: Transforms used for calculating the drone's pose relative to the landing pad.

Figure 2.10 shows the transforms involved in calculating the drone's position relative to the April Tag or WhyCon markers. Each node contains the name of the transform frame. The function of each frame is as follows:

1. `camera_frame_apriltag`: the transform representing the pose of the April Tag marker within the camera's field of view. This is taken from the April Tag ROS module which is analyzing the camera's published image.

2. `camera_frame_apriltag_straightened`: the transform representing the April Tag's pose when transformed to have a rotation quaternion of $(x, y, z, w) = (0, 0, 0, 1)$. This essentially gives the pose of the camera with respect to the marker. If the rotation of the April Tag marker in the camera's field of view is r_a, then this transformation composes a rotation of r_a^{-1}. In this case, since the April Tag's pose has unambiguous yaw, the yaw is included in this calculation.

3. `camera_frame_whycon`: the transform representing the pose of the WhyCon marker within the camera's field of view. Since the WhyCon marker does not have an unambiguous yaw position, this aspect of the pose is ignored. This is taken from the WhyCon ROS module which is analyzing the camera's published image.

4. `camera_frame_whycon_straightened`: the transform representing the pose of the camera with respect to the WhyCon marker. This is calculated similarly to the straightened April Tag transform, but without considering the yaw of the marker.

5. `body_enu`: the pose of the drone itself with respect to either the WhyCon or April Tag marker. This is calculated by composing the yaw component of the gimbal's rotation onto the previously straightened transforms. The relative displacement from the marker to the drone can be used in further calculations after it is transformed in this way. It is put in the ENU coordinate frame in order to conform to the MAVROS standard.

The transform library TF2 (used here) is the standard ROS transform library and provides an efficient means of keeping track of all published transforms

$$\theta_c = -(\theta_1 - \theta_2) \tag{2.2}$$

When the drone is far from the landing pad and only the WhyCon marker can be identified, the gimbal controller cannot determine any information about the yaw orientation of the landing platform. This is because the WhyCon marker has rotational symmetry. Initially, the landing platform was made up of a WhyCode marker which can allow for determination of the marker's yaw orientation. However, this led to complications and was eventually abandoned for this project, as will be explained in Section 3.3.1. The current landing platform design, using a WhyCon marker for recognition of the landing pad from a far distance, and an April Tag marker for close-range descent and accurate pose estimation, is sufficient for this project.

2.4 Landing Controller

2.4.1 Overview

The landing controller is a ROS module which controls the drone's approach and descent towards the landing platform. It subscribes to data topics published by the gimbal controller, such as the relative displacement of the landing pad with respect to the drone. Then the landing controller determines the commands to send to ArduPilot in order to direct the drone towards the landing pad. It communicates with ArduPilot via a separate ROS module called `MAVROS`, the sole purpose of which is to act as a connector between ROS modules and any vehicle using the MAVlink communication protocol. The `MAVROS` module opens the relevant UDP or Transmission Control Protocol (TCP) connections with instances of MAVlink-enabled software and enables transmission of data to and from these connections via ROS topics. These connections allow the landing controller to send control commands to the drone in the language of the flight controller, without changing the software on the flight controller.

2.4.2 Data Flow

The data flow for the landing controller is shown in Figure 2.11. The landing controller subscribes to a ROS topic containing the relative pose from the drone to the landing pad. It then transforms this pose from the camera's coordinate frame to the drone's local coordinate frame, using transforms that are also published by the gimbal controller. The landing controller publishes the states for each linear component of the displacement to each of the PID controllers. It also continually publishes the set points for the PID controllers - each with a value of 0. The planar distance to the landing pad is calculated from the relative pose, and this subsequently determines the landing phase and

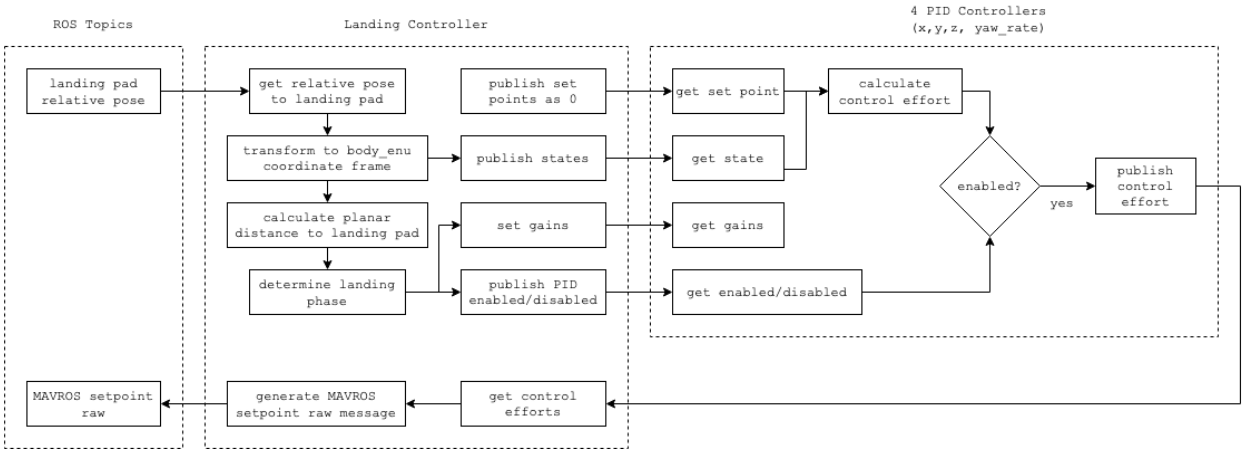


Figure 2.11: Data flow for the landing controller.

therefore the gains of the PID controllers and whether the PID controllers should be enabled or disabled. The PID controllers subscribe to this data, calculate the corresponding control efforts and publish the control efforts if they are enabled. Finally, the landing controller subscribes to the control effort topics, generates a message to be consumed by MAVROS, and publishes that message.

2.4.3 Velocity PID Controllers

PID controller instances control the velocity of the drone in 4 axes that are local to the drone: east, north, up, and yaw (positive yaw is clockwise). The state values for these controllers are the component of the drone's displacement from the landing pad in the relevant axis. Since the goal of the landing controller is to direct the drone towards the landing pad - in other words, to decrease the magnitude of the displacement - the landing controller continually publishes 0 as the set point to each of these controllers when they are active. A control policy both enables/disables these controllers and changes the gains of these controllers based on the phase of the landing process. The PID controllers publish their "control effort" topics after receiving data on their state and set point topics. These control effort topics are of course determined according to their gains. The landing controller subscribes to the control effort topics and translates these into a control message which is passed to ArduPilot via MAVROS.

2.4.4 MAVROS Interface

MAVROS handles communication between ROS and ArduPilot by translating ROS topics into MAVLink messages. For controlling position, velocity, and acceleration set points, the topic of interest is `/mavros/setpoint_raw/local`. This topic allows ROS to forward the target velocities that are calculated by the PID controllers to the drone itself. This does mean that there are two sets of PID controllers - one set which calculates the target velocities, and one set that realizes these target velocities by setting the lower-level pulse width modulation (PWM) throttle signals on the drone itself, while interfacing with the IMU to control otherwise unspecified targets such as attitude, acceleration, etc. This abstraction from the low-level commands is what allows the landing controller to be portable to different drone models and frames. The drone's intrinsic PID controllers which maintain the stable, controllable flight of the drone are tuned for their specific tasks, and the higher level PID systems are tuned for the specific task of realizing an efficient and safe trajectory

```
void set_velocity_target_enu( geometry_msgs::Vector3 _target_velocity,
                              double _target_yaw_rate )
{
        mavros_msgs::PositionTarget buffer;

        buffer.header.stamp = ros::Time::now();
        buffer.header.frame_id = "world";
        buffer.coordinate_frame = 8; // FRAME_BODY_NED
        buffer.type_mask = 1991; // only use velocity x, y, z, yaw_rate

        buffer.velocity.x = _target_velocity.x;
        buffer.velocity.y = _target_velocity.y;
        buffer.velocity.z = _target_velocity.z;

        buffer.yaw_rate = _target_yaw_rate;

        setpoint_raw_local_publisher.publish(buffer);
}
```

Listing 2.1: The simple function that allows the landing controller to control the drone's target velocities.

towards the landing platform.

Listing 2.1 shows the method for sending the control message to the drone. The target velocity and yaw rate represent the control effort variables calculated by the PID controllers. The `/mavros/setpoint_raw/local` topic receives a message of type `PositionTarget`, for which a buffer has been declared. The message is stamped with the current time and tagged with the frame identifier of "world." Multiple coordinate frames are available for the interpretation of the set points, but the only relevant one for this project is the coordinate frame that is local to the drone: `FRAME_BODY_NED` with code 8. A type mask tells the drone which parts of the message are relevant - in this case, only the linear velocities x, y, z and yaw rate are published, so a value of 1991 is used. The message is then published to the topic.

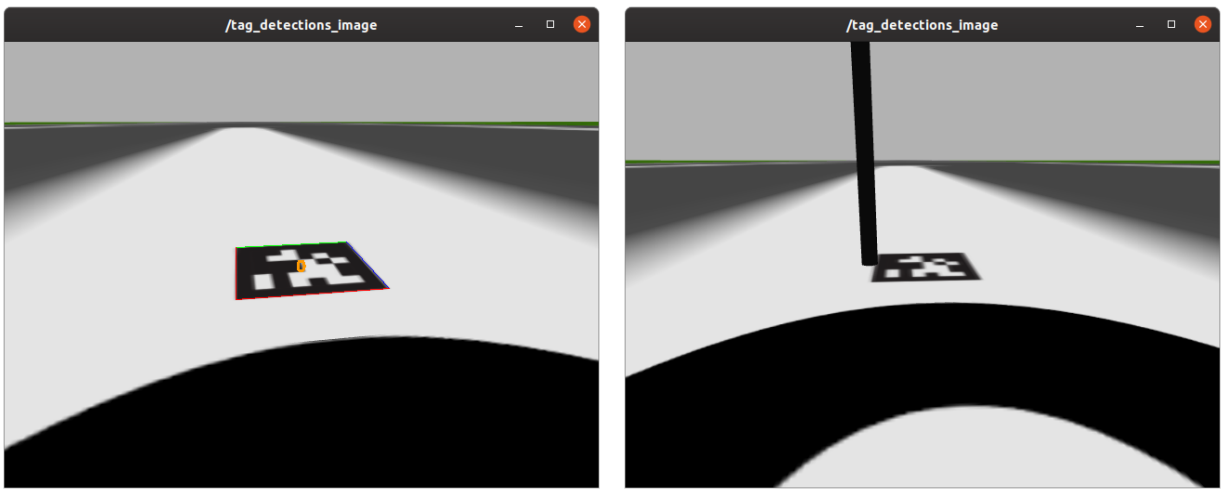
There is some tedium in this setup which must be considered. First, the standard coordinate system within Gazebo is North-West-Up (NWU), while the standard coordinate system within MAVlink is North-East-Down (NED). While these are both right-handed coordinate systems and therefore a "typical" and simple transformation (a rotation by π radians about the N axis) can be used, it is easier to manually change the signs of the corresponding components of the target velocity vector. Further, although MAVLink typically uses the NWU coordinate system, MAVROS itself takes the target velocity vector in the ENU coordinate system, which is not necessarily obvious from the documentation. However, when these aspects are accounted for, the software works as intended.

2.4.5 Control Policy

The landing controller enables/disables the PID controllers and changes their gains depending on conditions during the landing process. Initially the drone's velocity controllers have high P-gains and low D-gains. This allows the drone to approach the landing pad quickly from far distances. As the drone gets closer to the landing pad, the I and D gains become more important and are

set to experimentally-determined values (presented in Section 3.5) to avoid the quintessential PID overshoot and instead conserve battery and time. The standard ROS PID module has been slightly modified to allow the gains to be reconfigured quickly using a single topic, rather than the existing method of using `rqt_reconfigure` which requires a separate module and which seems to react slowly to the reconfiguration.[1]

Although a drone can move in any direction, it is necessary to control the yaw of the drone during descent, as the legs of the drone can obstruct the tracked fiducial marker. Although WhyCon markers tend to be robust to minor obstructions, the more intricate April Tag markers are not. The Iris, for example, has a black airframe which can merge with the black boundary and tiles of April Tag markers in the field of view of the camera, forming a contiguous black region which does not conform to the required structure of April Tag markers and prevents April Tag identification. Figure 2.12 illustrates this point. In Figure 2.12a, the April Tag marker is clearly identified (hence the ID is printed on the screen) even at a somewhat high angle of deflection, but in Figure 2.12b, the slight intersection of the drone's leg with the April Tag marker prevents identification. In this case, yaw correction ensures that the drone is aligned to the landing platform in such a way that the April Tag marker is unobstructed by the drone itself throughout the descent. However, yaw correction is only allowed once the April Tag marker has been identified, since the April Tag marker is the only marker on the landing pad which provides a yaw orientation. In contrast to April Tag's obvious sensitivity, Figure 2.13 shows the robustness of WhyCon identification. The shadow of the drone forms a contiguous black region extending the WhyCon marker's black region. (White and black are inverted in the picture in order to make the detection more clear.) However, the WhyCon system still correctly identifies the marker and places its center in the correct location, as shown by the very faint green and red dots.



(a) April Tag not obstructed, and therefore successfully identified.

(b) April Tag slightly obstructed, and therefore not identified at all.

Figure 2.12: Even slight obstructions can prevent identification of April Tag markers.

Finally, descent is allowed or disallowed based on a dynamic threshold δ on the planar distance

[1]The original PID module code can be found at `https://bitbucket.org/AndyZe/pid/src/master/` and the slightly edited version can be found at `https://github.com/uzgit/pid`.

Figure 2.13: WhyCon identification is robust even to significant obstructions.

d_p to the landing pad. This threshold follows an exponential function as outlined in Equation 2.3:

$$\delta = k_1 e^{k_2 z} \tag{2.3}$$

where

- δ is the planar distance threshold, below which the drone is allowed to descend,
- k_1 and k_2 are constants determined through exponential fitting of a curve to experimentally-determined constraints, and
- z is the vertical distance from the drone to the landing pad, or equivalently the altitude of the drone above the landing pad.

The drone is allowed to descend if $d_p = \sqrt{x^2 + y^2} < \delta$, where x and y are the distances from the drone to the landing pad in the drone's East and North axes respectively. This means that, the higher the drone is above the landing pad, the farther away it is allowed to descend. Conversely, the lower the altitude of the drone, the closer the drone must be to landing pad in order to descend - ensuring that the drone does not descend to the ground in any place except for the landing pad. This is visualized in Figure 2.14, where the descent region is the interior of the plotted surface. The motivation for this is that, with the movements and vibrations of the drone - and the inherent inaccuracies in monocular pose estimation - the drone is able to better estimate its pose when it is closer to the landing pad (e.g. after it has partially descended). The specific landing phases are outlined in Table 2.1.

Each of the landing phases is used only to enable and disable PID controllers and change their gains. However there are two other, redundant checks performed at every iteration of the landing controller's loop. First, the yaw of the drone is locked if the drone's yaw is aligned within 5 degrees of the landing pad's yaw orientation. This is done simply by disabling the yaw PID controller and setting a target yaw velocity of 0. Second, if the drone is not in the `LANDED` phase, a separate check disables descent if the drone is outside of the descent region, and enables it otherwise. This is done by disabling the PID controller for the up direction and setting a target up velocity of 0, and re-enabling the PID controller respectively.

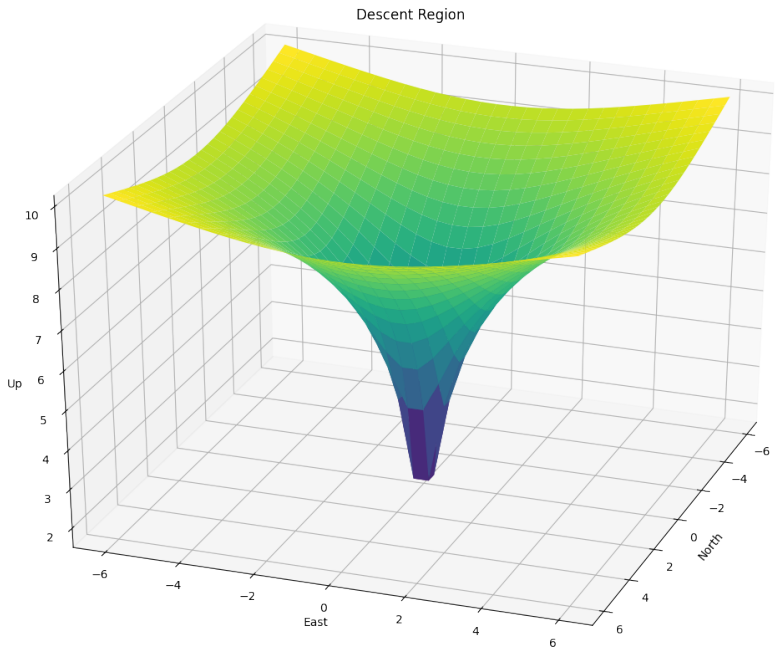


Figure 2.14: An example descent region for a $k_1 = 0.15, k_2 = 0.4$, with the landing platform positioned at $(x, y, z) = (0, 0, 0)$. The drone is allowed to descend in the interior of the plotted surface.

The landing controller is enabled or disabled based on the PWM signal given on a user-defined channel. In this case, channel 11 is used, and a PWM signal on this channel with a duty cycle of more than 50% enables the landing controller. It is disabled otherwise. This allows manual landing abort and also integration of an autonomous landing into semi-autonomous flights.

At the time of this writing, ArduPilot does not appear to have a feature which allows the sudden disarming of a drone's motors, which would be quite useful after the drone has landed. However, ArduPilot does automatically disarm the motors after it detects that the drone has landed, even when the final velocity is non-zero, and even when non-zero target velocities are set. In order to use ArduPilot without editing it, this fact is leveraged, and the motors are allowed to automatically disarm. It is important that the landing pad maintain a constant velocity until the drone disarms.

NOT_LANDING	This is the mode set when the landing controller has been disabled or the landing pad is not yet recognized. Setting this mode triggers a target velocity of 0 in all dimensions, effectively aborting an ongoing landing. This can happen if the user disables the landing or if the landing pad leaves the field of view of the camera for a specified abort time.
APPROACH	This mode is the first mode set when the landing controller is enabled and the landing pad is located. This mode uses high P-gain, and low D-gain for a quick initial approach, and a 0 I-gain in order not to bias the drone's final velocity.
CLOSE_APPROACH	This mode is set once the drone's planar distance to the landing pad is less than some experimentally-determined multiple of the descent distance. The P-gain is lowered and the D-gain is raised in order to gradually slow the drone's approach. The I-gain is still set to 0 at this point. The motivation for this landing phase is to avoid the drone overshooting the landing pad after a quick initial approach.
YAW_CORRECTION	During this mode, the drone's yaw PID controller is enabled, which allows the drone to align itself with the landing pad so that the drone's body does not obscure the April Tag marker during final descent. The P-gain of the east and north PID controllers is reduced yet again, and the D-gain increased, in order to avoid overshoot. Descent to some minimum altitude is allowed during this phase, however full descent is not allowed until the drone is aligned within 5 degrees of the landing pad's orientation.
DESCENT	During this mode, the yaw PID controller is disabled and the heading of the drone is locked by setting its target yaw velocity to 0. Descent below the previously mentioned minimum altitude is allowed. The PID controllers in the north, east, and up directions are still enabled for final velocity correction.
LANDED	All PID controllers are disabled, east and north target velocities are preserved and continually sent as velocity targets to the drone. The target velocity in the up direction is set to some constant negative value and also sent to the flight controller.

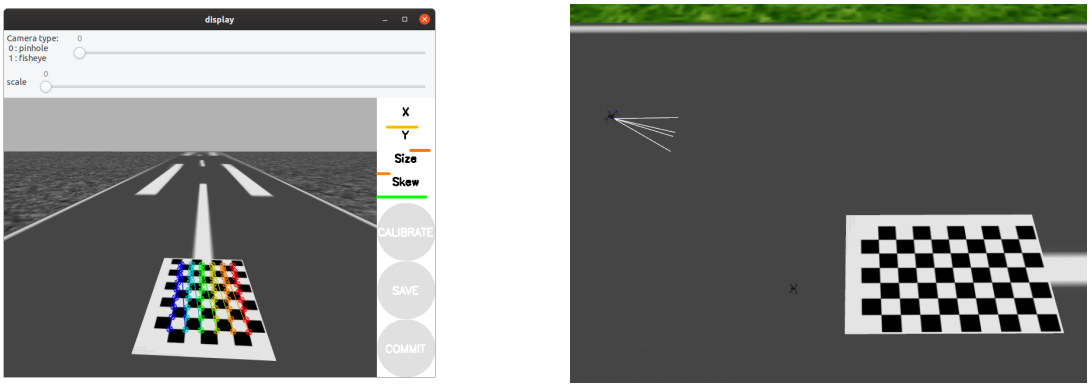
Table 2.1: Landing Phases

Chapter 3

Simulation Results

3.1 Simulated Camera Calibration

As described in Section 2.2.1, a standard Gazebo ROS camera sensor is "mounted" to the Iris' gimbal and provides a simulated camera image as a ROS topic. The important aspects of this code are the specifications for the camera's distortion coefficients, which are all set to 0.0. Any library using a camera for image processing will require these values. Instead of passing these values to the WhyCon and April Tag libraries directly, the `calibrate_camera` ROS module was used to generate a `.yaml` file containing the empirically determined values. This method is a de facto standard way of calibrating a camera in ROS, and can be used during migration to a physical system. The `calibrate_camera` script requires a rectangular "chessboard" defined by $m-1$ and $n-1$ where m is the amount of squares along one side, and n is the amount of squares along a perpendicular side. This chessboard was inserted into `sandbox.world` as a texture, as shown in Figure 3.1. The sizes of the squares in the chessboard are an important factor in calibration, and they were approximately the size of a single 1-meter grid square. The drone was directed around the chessboard in order to view it from many angles and distances, after which a calibration file was generated.



(a) "Chessboard" detection.

(b) The calibration "chessboard" from above.

Figure 3.1: Calibration of the simulated camera.

The calibration generated the following data which has been rounded for conciseness. The camera intrinsics matrix K contains f_x, f_y which are the focal lengths of the camera in the x and y dimensions respectively, in pixel units. It also contains c_x, c_y which represent the coordinates of a

principal point that should be near the image center. After calibration, the simulated camera sensor is calculated to have focal lengths f_x, f_y of 398.2 and 390.8 pixels respectively. With a resolution of 640x480 pixels, the calibrated values of $c_x = 299.9$ and $c_y = 227.5$ are only *near* the image center - slightly to the upper left.

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 398.2 & 0 & 299.9 \\ 0 & 390.8 & 227.5 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

The distortion coefficients determined by the camera calibration, D_T, are roughly equivalent to those set in the camera plugin's parameters. The radial distortion coefficients k_1, k_2, k_3 are set to exactly zero but are calculated as only near-zero after the calibration. The tangential distortion coefficients p_1, p_2, corresponding to T1 and T2 in the camera plugin parameters, are also non-zero but near-zero.

$$D^T = \begin{pmatrix} k_1 \\ k_2 \\ p_1 \\ p_2 \\ k_3 \end{pmatrix} = \begin{pmatrix} 0.0576 \\ -0.0321 \\ 0.0041 \\ -0.0203 \\ 0 \end{pmatrix} \tag{3.2}$$

It is important to note that the process of calibrating the camera module has been developed out of necessity because typical real world cameras have non-negligible distortion coefficients. The values determined for D^T are significantly lower in magnitude than those of a typical, real camera.

3.2 Gimbal Controller

As stated in Section 2.3.4, the main job of the gimbal controller is to aim the camera directly at the landing platform. It does this by using 2 PID controllers - one on its pitch angle and one on its yaw angle - to control its 2 degrees of freedom. Initially, PD controllers were used for this purpose, but they were inadequate as the simulated gimbal has spring forces which tend to keep the gimbal's positions at their original points. As the magnitude of the pitch or yaw angles increases, the force required to further increase the angle's magnitude increases. Over time, a small integral gain helps correct the error caused by this change in required force. Table 3.1 shows the gains for the gimbal PID controllers.

Controller	k_p	k_i	k_d
Pitch	0.25	0.1	0.025
Yaw	0.25	0.1	0.025

Table 3.1: Gimbal controller PID gains.

The x and y components of the un-transformed landing platform pose comprise the input for target angle calculation, and these target angles are used as the set points for the PID controllers. The x and y components are scaled by a factor inversely proportional to the z component in order to allow the PID controllers to function in a stable way over a variety of distances. If the x and y components are used without scaling, the PID controllers wildly overshoot at long distances and undershoot at short distances. Figures 3.2 and 3.3 show the performance of the gimbal controller during 10 landings. In both figures, each line represents a single attempt to aim camera over the

course of the landing. Time $t = 0$ represents that time of first recognition of the landing platform. At about 30 seconds in each landing sequences, the drone has made contact with the landing platform. The graph continues in time until the drone disarms, completing the landing. Figure 3.2 shows that, after some oscillation, the yaw of the gimbal is adjusted so that the marker is in the center of the camera's view in the x direction. Figure 3.3 shows similar results with regards to the gimbal's pitch, but with an added, temporary bias during which time the gimbal is pointed slightly lower than the landing platform. This is because, when the drone is close to the landing pad, the April Tag marker is almost directly in front of the drone and therefore the spring resistance in the gimbal is high. The error is eventually corrected by the integral gain and the marker becomes centered in the y direction of the camera frame. Centering of the landing platform in the camera's field of view is represented by the convergence of each line to 0. However, even though the ideal goal of the gimbal controller is to keep the landing platform centered in this way, it is only critical that it must keep the landing platform in the camera's field of view throughout a variety of orientations and displacements. These figures show that the gimbal controller does accomplish this goal.

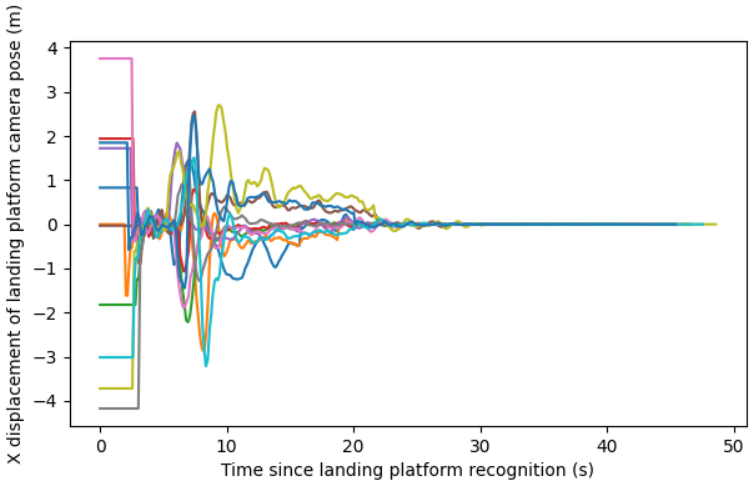


Figure 3.2: Landing platform x displacement in camera frame versus time. Each line represents a single attempt to aim the camera during landing.

The pose of the landing platform in the z direction behaves differently, in that it converges to some non-zero distance representing the depth of the landing platform in the camera frame. As the drone approaches the landing platform, this depth of course decreases to near-zero as the drone lands.

3.3 WhyCon Pose Estimation

WhyCon markers, as previously known, provide a good means of pose estimation in 3 dimensions, especially in stationary scenarios. Stationary WhyCon pose estimation was therefore the first step in testing the drone's ability to identify the landing platform's position. In configuring the WhyCon system, the outer diameter of the WhyCon marker is a required point of data. The WhyCon module makes it easy to configure this in real time using `rqt_reconfigure`. The calibration was quite simple regardless, as the marker was intentionally sized to 1 (simulated) meter in Gazebo. The results of the system calibrated in this way are shown below.

The WhyCon marker is the intended landing site of the drone. In the stationary landing scenario,

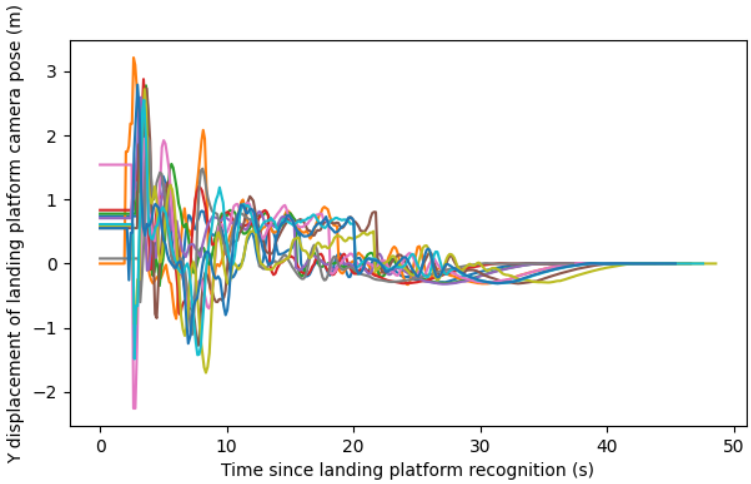


Figure 3.3: Landing platform y displacement in camera frame versus time. Each line represents a single attempt to aim the camera during landing.

it was placed at (0, 30, 0) and, as shown in Figure 3.5, the drone is able to reliably estimate the pose of the WhyCon marker. There is transient angular movement in the camera from the moment that the system recognizes the marker to the moment that the camera centers on the marker, which cannot practically be avoided except in truly theoretical scenarios. Moreover, this transient state provides noise which the system must overcome in a real scenario, so neglecting it does no good. Table 3.2 outlines the average pose estimations in each dimension (μ_x, μ_y, μ_z), as well as the standard deviations in each dimension $(\sigma_x, \sigma_y, \sigma_z)$. The average estimations of the landing pad's position in the plane are roughly accurate, $\mu_x = 0.008 \approx 0$ and $\mu_y = 29.704 \approx 30$. Their standard deviations of $\sigma_x = 0.996$ and $\sigma_y = 0.518$ are acceptable and expected levels of variation. The pose estimate in the z axis is somewhat harder to estimate conceptually - especially with a monocular camera - and this is reflected in the clear overestimation of the relative distance from the drone to the marker in the z axis. While the true value of the position of the WhyCon marker in the z axis is 0, the average estimate value is $\mu_z = -0.050$, actually putting the marker below the ground! However, an overestimate in the z axis of the WhyCon marker is not prohibitively destructive, as the WhyCon marker is not even used for final descent since it will by then be out of the field of view of the camera.

μ_x	0.008
μ_y	29.704
μ_z	-0.050
σ_x	0.996
σ_y	0.518
σ_z	0.142
n	1292

Table 3.2: Means and standard variations for stationary WhyCon estimation.

Figure 3.5 shows some of the outliers in the WhyCon pose estimation. It is worth mentioning that these outliers exist, although they do not drastically affect the pose estimation. One mitigating factor that negates the effect of these outliers is that the WhyCon "callback" function estimates the

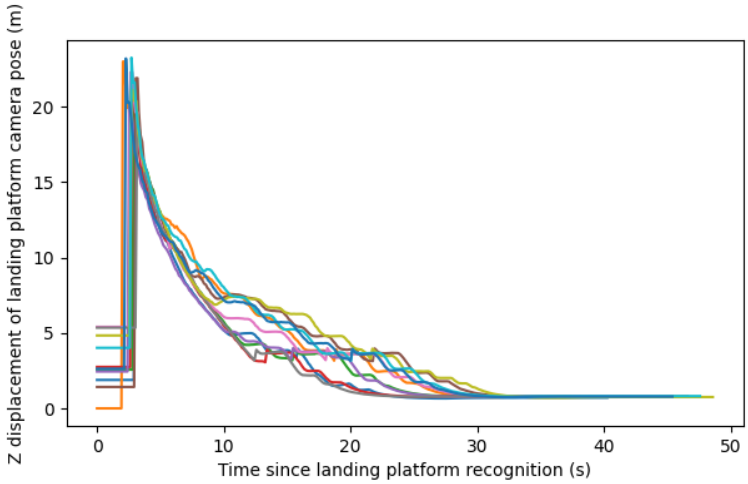


Figure 3.4: Landing platform z displacement in camera frame versus time. Each line represents a single attempt to aim the camera during landing.

pose of the landing pad at roughly 30 Hz, such that the 17 outliers represent just slightly more than 0.5 seconds of actual runtime, and the rest of the estimates are much closer to the correct value. Further, as differently colored dots represent different samplings, each outlier represents only a minuscule amount of inaccurate pose estimation time per approach. These outliers seem to be caused by the initially inaccurate pose estimation of the marker upon its first detection in a moving frame.

Figure 3.6 shows a graph of the magnitude of the 3-dimensional WhyCon pose estimate error over the course of 10 approaches. That is, if the error e of the estimate has components e_x, e_y, e_z, the plotted value for the magnitude of this error is simply $|e| = \sqrt{e_x^2 + e_y^2 + e_z^2}$: the Euclidean distance between the true value and estimated value of the position of the WhyCon marker. These results generally show an increase in pose estimation error when the distance from the marker increases. A regression line following Equation 3.3 estimates the relationship between the WhyCon pose estimation error and the distance to the WhyCon marker, where y denotes the pose estimation error and x denotes the distance to the marker. The standard error for this regression is $\sigma = 0.0154$ over the $n = 1292$ samples. The main takeaway from this result is that the WhyCon marker can be used from a distance of about 1 meter to a distance of about 18 meters. When the drone is closer than 1 meter to the landing platform, the WhyCon marker is out of the camera's field of view, and when the drone is sufficiently far away from the landing platform, the marker is simply not recognizable.

$$y = -0.004x^3 + 0.180x^2 - 0.876x + 1.843 \tag{3.3}$$

3.3.1 WhyCode Trials

The original landing platform design was a single WhyCode marker. This minimalistic design could simplify the system by lessening the amount of coordinate system transforms and components within the gimbal controller, as well as the physical landing platform itself. However, due to two issues, this option was abandoned in favor of the aforementioned landing platform design.

The first issue is that each set of WhyCode markers (where a group is defined by the number of

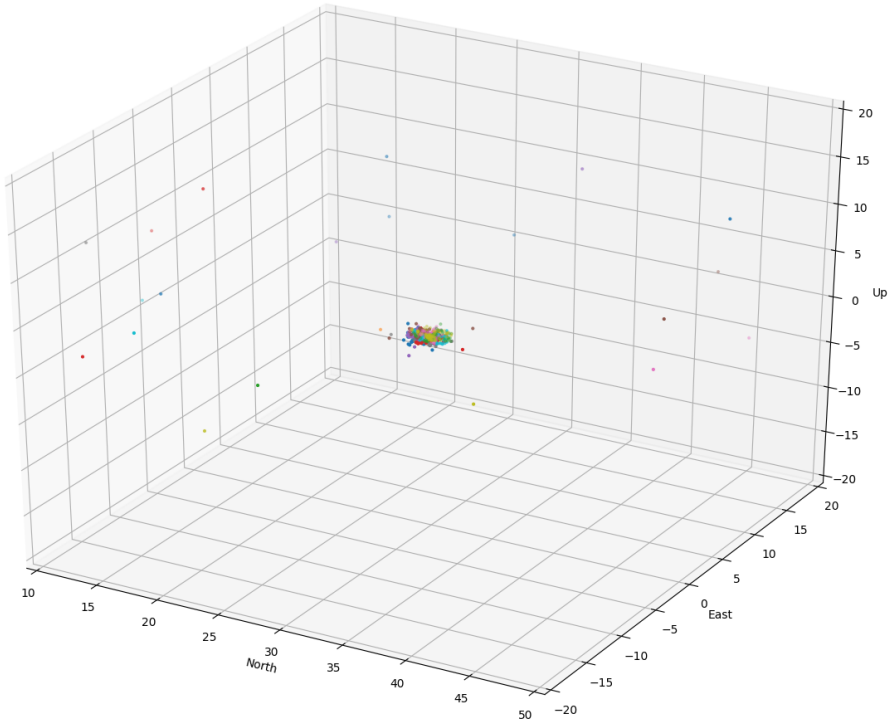


Figure 3.5: WhyCon stationary pose estimation with outliers included. This shows that the majority of the poses are concentrated in a small area, and the outliers are relatively few.

bits contained in the group's IDs) contains rotationally symmetric markers. This is true for markers with more than a single ID bit. This is shown in Figure 3.7, where the number of ID bits is 2. Moreover, markers such as those with IDs 3 and 4 do not provide an unambiguous orientation, as there is no way to define a unique "front" on the marker. However, the markers with IDs 1 and 2, though rotationally equivalent to each other, do have a potential "front" which would allow for the unambiguous determination of the marker's yaw. The original plan for the system design involved placing the "front" of the marker at the center of the larger white semi-circle of a WhyCode marker with ID 1. The anticipated workaround for the problem of the marker's rotational symmetry was to use the natural orientation of the marker if the `whycon_ros` system recognized its ID as 1, and to apply an initial rotation of π radians about the marker's central z axis if it was recognized with an ID of 2. In theory this method should still work, but it was never fully attempted because of the second issue.

The second issue with using WhyCode markers came from the unanticipated, destructive interaction between the PID systems used to aim the camera, and the WhyCode system. The exact cause of the issue has not been determined, but it corresponds to sign changes in the positional components of the WhyCode marker's pose. This issue only happens specifically when the marker is being identified, and it does not happen every time the marker is identified. This made the issue very hard to isolate. Further investigation into this issue is necessary.

The issue can be illustrated through an experiment, the results of which are shown in Figures 3.8 and 3.9, where the drone was stationary on the ground, viewing a marker in front of it, with the gimbal controller aiming the camera at the marker. As shown in Figure 3.8, the x and y components

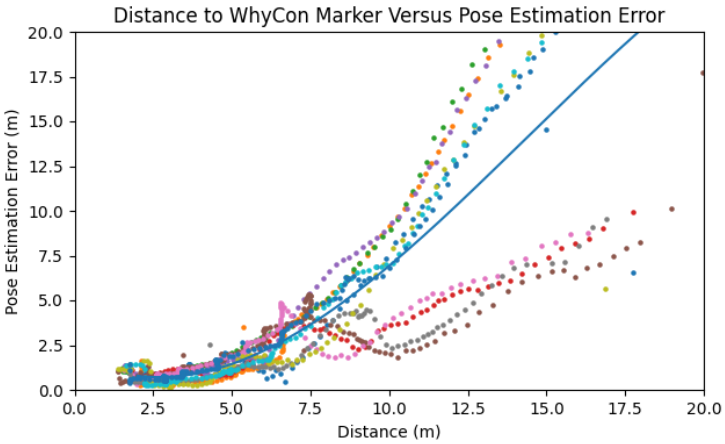


Figure 3.6: Distance to WhyCon marker vs. magnitude of 3-dimensional pose estimate error.

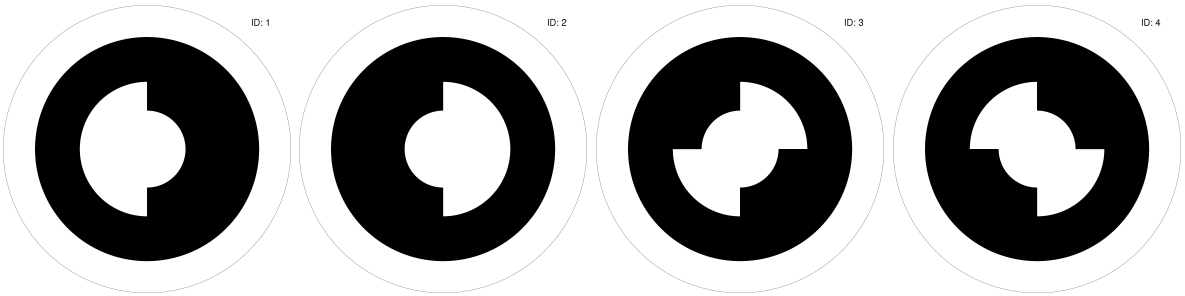


Figure 3.7: Rotationally symmetric WhyCode markers with 2 ID bits, generated from the LCAS `whycode_id_gen`.

of the marker within the camera frame are close to 0, but oscillate, after the marker is detected. This oscillation is to be expected because of the nature of PID controllers, and it is not inherently a problem in itself. However, the orientation of the marker is an important component of the pose, and, as Figure 3.9 shows, the small, acceptable oscillation in the positional components corresponds to large, unacceptable oscillation in the marker's recognized orientation. The oscillation in the orientation does not represent noise, but rather an axial flip without a change in the perceived marker ID. The ultimate result of this was that the marker's pose could not be reliably determined while the WhyCon module's identification functionality was running. Because of this issue, the identified WhyCode markers were temporarily abandoned in favor of the April Tag markers for the determination of the landing platform's yaw.

A simple, "bandaid" method for dealing with the WhyCode orientation problem shown in Figures 3.9 and 3.8 has been implemented. Let i represent the index of a detection, and let be a quaternion (Explained in Section 1.2.4) representing d_i representing the ith detected orientation. At detection i, the gimbal controller stores the orientation, d_i, of the marker. It compares the *inverse* of the detected orientation at frame $i+1$, d_{i+1}^{-1} to the orientation of the marker at detection $i+1$. If the angle represented by d_{i+1}^{-1} is within some small angular displacement θ_d from the angle represented by d_i, then the inverse is assumed to be the correct orientation, is used for further calculation, and is stored for comparison at detection $i+2$. Figure 3.10 shows how this method maintains continuity within the same test shown in Figure 3.9, when $\theta_d = 2°$. It is somewhat difficult to identify the

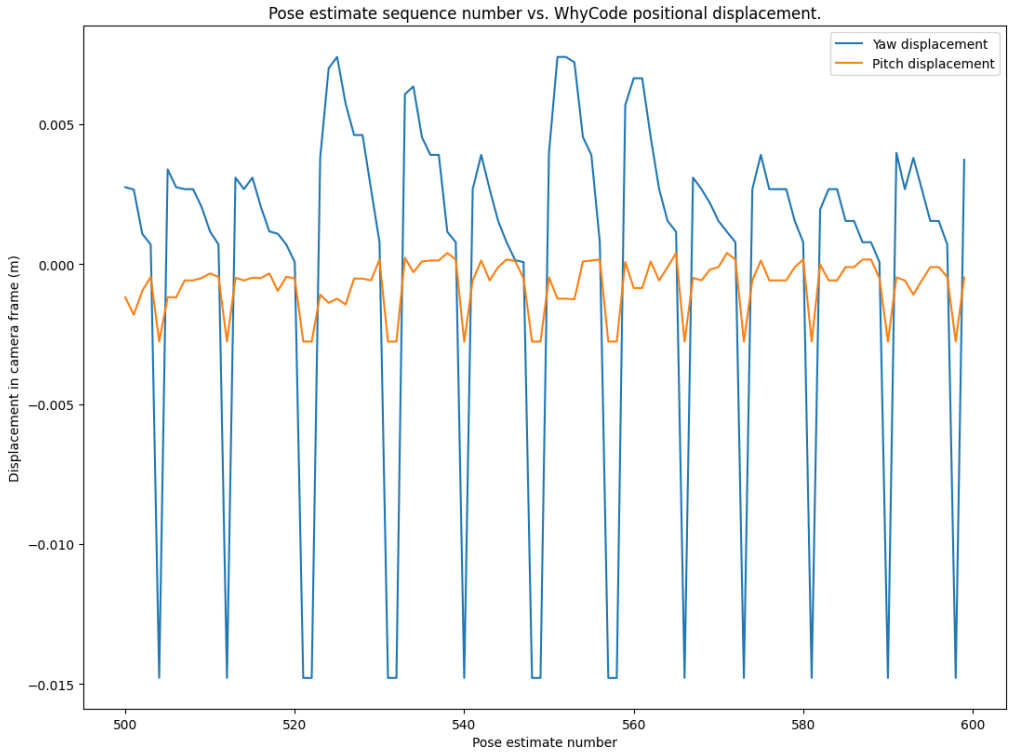


Figure 3.8: x and y position components of the WhyCode marker's pose in the camera frame.

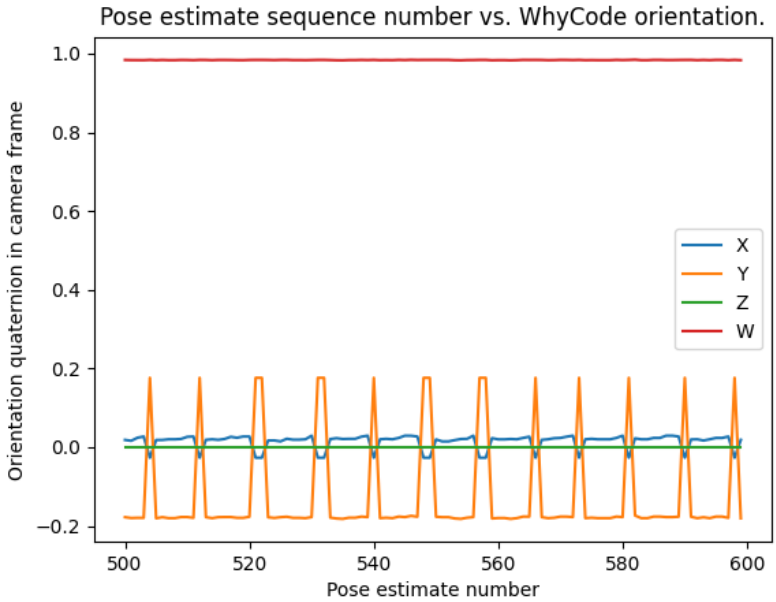


Figure 3.9: x, y, z, w orientation components of the WhyCode marker's pose in the camera frame.

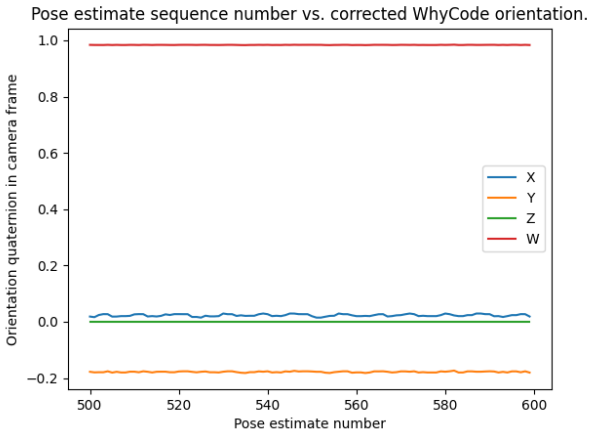


Figure 3.10: WhyCode marker orientation components after "fixing." This shows that the persistent flipping shown in Figure 3.9 has been removed, and the perceived orientation of the WhyCode marker is stable.

cause of this error within the WhyCode source, so as a proof of concept, this "bandaid" method was used to correct the issue in this particular scenario. While the fix corrects the orientation of the marker in this particular scenario, it did not solve the issue in testing generally.

3.4 April Tag Pose Estimation

Similar experiments to those used to test WhyCon pose estimation were carried out for the April Tag markers. Figure 3.11 shows the wide spread of high-error pose estimation for the April Tag over the course of 10 approaches. It is not surprising that the marker of size 0.3125 meters by 0.3125 meters is not accurately located from the same distance as the WhyCon marker. However, since the April Tag marker is only to be used in late descent, this can be overlooked in favor of the marker's performance at lower altitudes. Table 3.3 outlines the performance of April Tag pose estimation during 10 approaches at altitudes between 0 and 10 meters. The standard deviations of the measurements decrease as distance from the marker decreases, while the means remain in the same acceptable region. Similarly to WhyCon, the z component of the April Tag's pose is somewhat incorrectly estimated - but not prohibitively so.

μ_x	0.0720
μ_y	30.060
μ_z	-0.394
σ_x	1.230
σ_y	1.297
σ_z	0.589
n	2707

Table 3.3: Means and standard variations for April Tag estimation.

The pose estimation is visualized in Figure 3.11. The pose was estimated at altitudes varying from 0 to 10 meters in motion, revealing an exponential relationship between the distance from the marker and error in pose estimation magnitude as shown in Figure 3.12. The main takeaway from

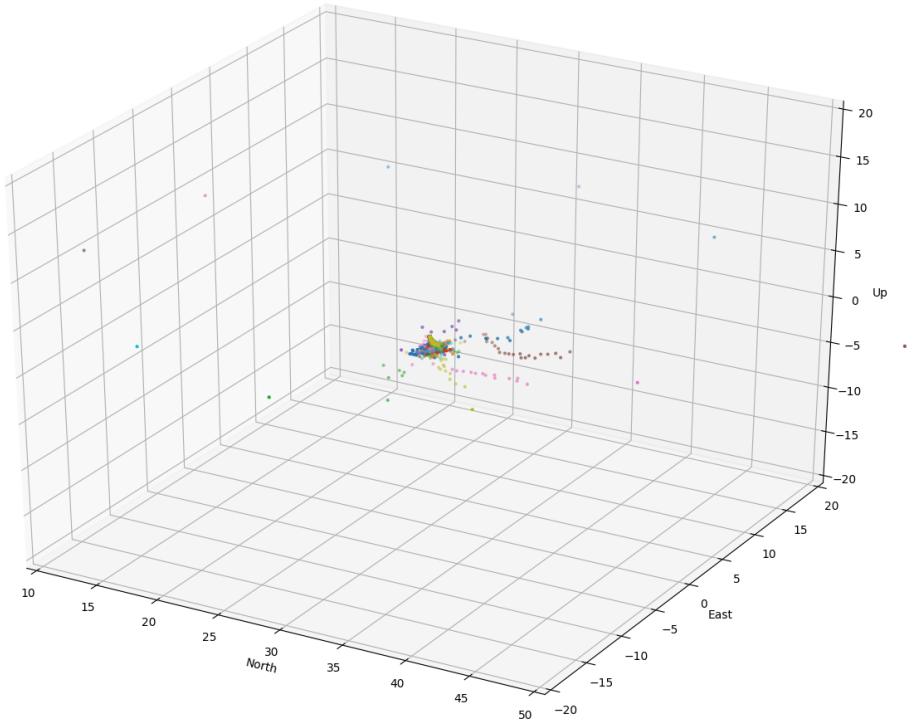


Figure 3.11: April Tag pose estimation.

this test is that the error in pose estimation increases with the distance from the marker to the drone, but at close distances the pose estimate is very accurate. Furthermore, the pose estimate is useful to the system throughout the approaches, even in spite of its error. A regression following Equation 3.4 describes the an estimate of the relationship between the pose estimation error and the distance to the marker, where y denotes the error and x denotes the distance to the marker. The standard error for this regression is $\sigma = 0.0047$ meters over the samples.

$$y = -0.002x^3 + 0.098x^2 - 0.095x + 0.256 \tag{3.4}$$

3.4.1 Yaw Estimation

The yaw of the April Tag is a necessary data point during autonomous landing, as the drone must align itself to the landing pad in order to guarantee that a fiducial marker (in this case, the April Tag) will be in its field of view during the final part of the landing sequence. Thus, the performance of the April Tag's yaw recognition is evaluated. Figure 3.13 shows the perceived yaw angle during the 10 test approaches, and Table 3.4 describes them statistically.

μ_{yaw}	3.149
σ_{yaw}	0.044

Table 3.4

These show that the yaw estimation of the April Tag marker is quite accurate over its detection range. An angle of $\frac{\pi}{2}$ radians represents due north, however the April Tag marker has been rotated

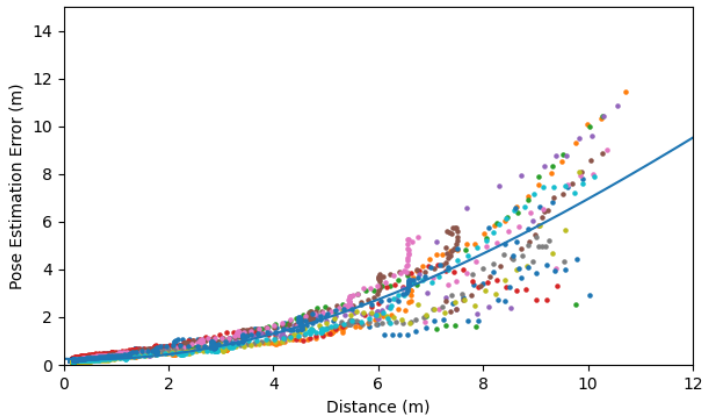


Figure 3.12: Pose estimation error versus distance for the April Tag marker.

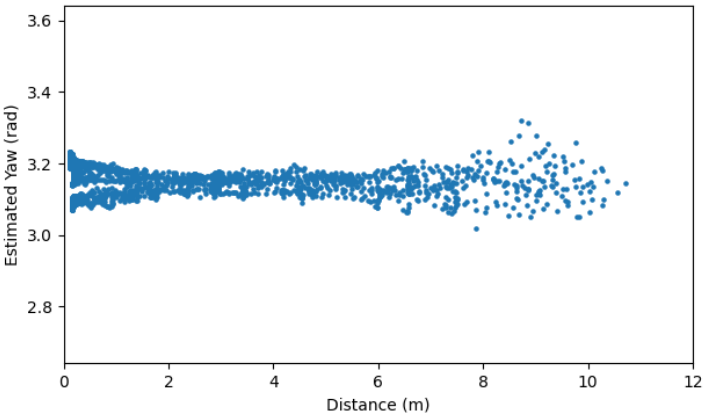


Figure 3.13: Estimation of the April Tag marker's yaw during the pose estimation approaches.

by an additional $\frac{\pi}{2}$ radians in its position on the landing platform, giving the estimated yaw of $3.149 \approx \pi$ radians. Figure 3.14 shows the absolute value of the April Tag's yaw estimation error. It is apparent that the yaw estimation error has increases when the distance from the landing platform is both very high and very low. A regression following Equation 3.5 describes the relationship between the yaw estimation error and the distance to the April Tag marker, with y denoting the yaw estimation error in radians and x denoting the distance to the marker in meters. The standard error of the regression is $\sigma = 0.0306$ radians over the $n = 2707$ samples.

$$y = 0.003x^2 - 0.018x + 0.051 \tag{3.5}$$

The variance of the error in the April Tag's yaw estimation increases at very close distances. This is likely caused by the high angle of deflection of the marker in the camera's field of view, as shown by Figure 3.15. However, this small issue can be overcome easily with the landing policy. The yaw position is locked if the yaw displacement of the drone is less than 5 degrees, or approximately 0.0873 radians, meaning that small variations in the yaw measurement will have no effect on the drone's behavior.

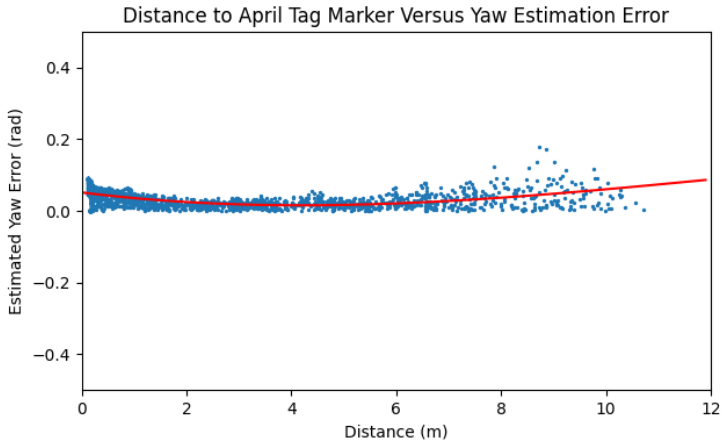


Figure 3.14: April Tag yaw estimation error with regression.

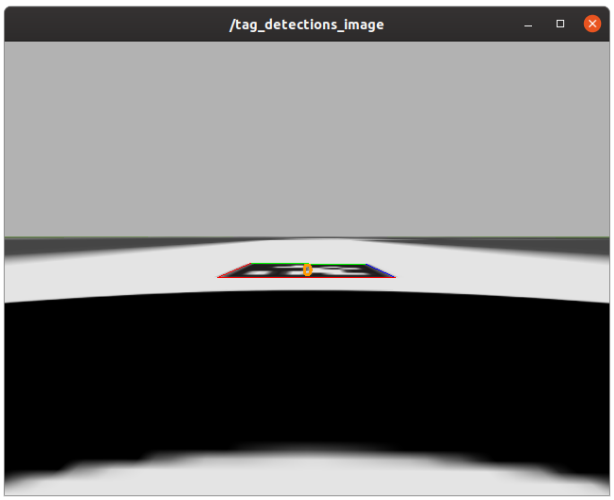


Figure 3.15: The view from the drone immediately after landing, with the April Tag's high angle of deflection.

3.5 Initial PID Controller Tuning

PID controllers have parameters which must be tuned for the system and conditions that they will face. The first step in landing on a stationary platform is to tune these parameters using a priori knowledge of PID controller behavior and simple trial and error. Importantly, in a scenario involving a stationary landing platform with stable, well-tuned PID controllers, all of the state errors will converge to 0. This means that the integral gain of each of the PID controllers can be left at 0, making the control purely PD-based. This is conceptually different from a scenario wherein the landing pad is moving, which requires the additional integral gain (described in Section 3.7). The approach taken in this project is to first tune the PID systems for a stable landing on a stationary landing platform, and then transition to moving landing scenarios, tuning only the integral gain.

3.5.1 North and East Velocity Controllers

The goals in tuning the gains of the east and north velocity PID controllers in order of priority are as follows:

1. **Avoid overshoot.** Overshooting the landing platform causes complications in aiming the camera and increases both time and energy spent in landing.

2. **Avoid undershoot.** Undershooting the landing platform increases the time and energy spent in landing. Avoiding overshoot takes precedence over avoiding undershoot. Overshoot is eliminated first. If this causes undershoot then the undershoot is gradually eliminated as long as it does not cause overshoot.

3. **Minimize energy required for landing.** The energy required for landing must comprise only a small portion of the battery's capacity in order to maximize usable flight time for mission tasks. This metric is obviously related to the time spent for landing, but differs in that it accounts for energy-intensive maneuvers such as course correction.

4. **Minimize time required for landing.** As the landing time increases, so does the possibility changes in the environment and the energy required for the landing.

Initial, informal experiments showed that PID parameters shown in Table 3.5 provided an imperfect but relatively reliable performance in the drone's approach towards the landing pad. These initial experiments decrease the search space for the PID parameters and therefore decrease the time for testing PID gains more rigorously. The parameters are all negative because the PID controllers in this scenario are reverse-acting - that is, a positive change in the control effort produces a negative change in the error between the set point and the state variable. Phase 1 refers to the PID gains used upon initial recognition of the landing platform. Phase 2 refers to the PID gains used when the drone is displaced by an intermediate distance from the landing platform. Phase 3 refers to the PID gains used when the drone is in its final approach.

Phase	k_p	p_i	k_d
1	-0.7	-0.0	-0.4
2	-0.4	-0.0	-0.7
3	-0.3	-0.0	-0.9

Table 3.5: Initial north and east PID tuning parameters. The same parameters are used for both systems.

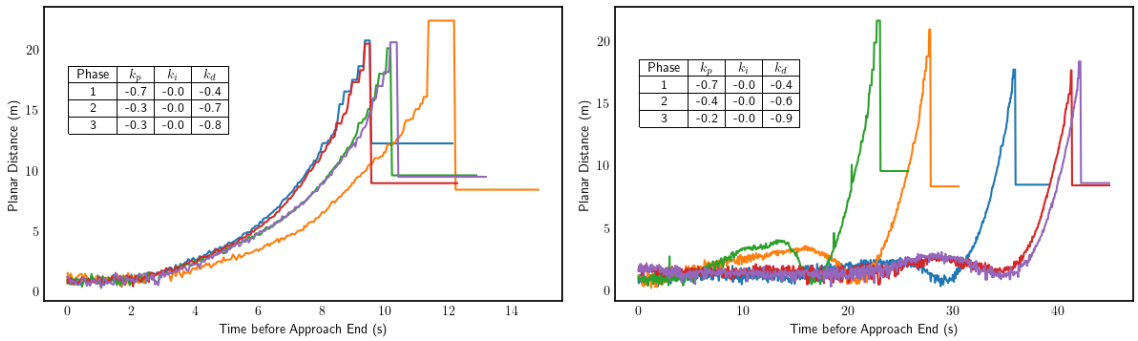
The search space for the north and east PID parameters is comprised of all combinations of $k_p \pm 0.1$ and $k_d \pm 0.1$ for phases 1 and 2. The gains for phase 3 are kept constant in order to save time during the slow testing in Gazebo. Each of the combinations of gains is tested over 5 approaches. The approach series starts at the origin with an altitude of 10 meters, with the drone facing directly south in order to ensure that the landing platform is not in the drone's field of view. The landing controller is initially disabled by sending a PWM signal of $1100\mu s$ on RC channel 11. Then, the drone is sent to a point 200 meters directly north of its original position, while also facing directly north. At this point, the landing controller is enabled by sending a PWM signal of 1900 μs on RC channel 11. With each approach, the drone's starting position is moved 1 meter to the right in order to have variation in both planar dimensions of the drone's displacement from the landing pad. When the landing pad, positioned at 30 meters north of the origin, comes into the drone's field

of view, the landing controller's north and east velocity PID controllers begin to control the drone. An approach is considered successful if the drone maintains a planar distance of no more than 1.6 meters from the landing pad (calculated through pose estimation) for 3 consecutive seconds. The system is given a maximum of 45 seconds from the acquisition of the landing platform to achieve a success before the approach is considered a failure. After a success or failure, the landing system is disabled, the drone is positioned at the starting point of the next approach, and the test is started again. For consistency, this process is automated using a Python script `control_pid_ne.py`. In this way, the drone follows exactly the same initial trajectory with the same velocity before acquisition of the landing platform. During these tests, the yaw correction and altitude correction of the landing system are inhibited to isolate the performance of the north and east PID controllers exclusively.

During each approach, the Python script records the time and energy required by the drone from the point of visual acquisition of the landing platform to a success or failure. The value for time is the ROS simulation time. The value for the energy is taken from the `battery_status` MAVlink message 147. This message provides a value in hecto Joules (hJ) given a properly calibrated power module onboard the drone. The gain combinations are rated according to the average time and energy required over the course of the 5 approaches. The 81 gain combinations are described by 405 separate approaches, but the results of the best combinations are shown in Table 3.6.

k_{p1}	k_{i1}	k_{d1}	k_{p2}	k_{i2}	k_{d2}	k_{p3}	k_{i3}	k_{d3}	t_{ave} (s)	e_{ave} (hJ)
-0.7	-0.0	-0.4	-0.3	-0.0	-0.7	-0.3	-0.0	-0.8	13.0	52.4
-0.7	-0.0	-0.4	-0.3	-0.0	-0.7	-0.3	-0.0	-0.9	13.7	54.8
-0.7	-0.0	-0.4	-0.3	-0.0	-0.8	-0.3	-0.0	-0.9	13.8	55.4
-0.7	-0.0	-0.4	-0.3	-0.0	-0.8	-0.3	-0.0	-1.0	13.9	55.2
-0.7	-0.0	-0.4	-0.3	-0.0	-0.8	-0.2	-0.0	-0.8	15.0	60.0

Table 3.6: Best 5 initial gain combinations by time and energy for the north and east PID controllers



(a) Best-performing N/E PID parameters. (b) Badly-performing N/E PID parameters.

Figure 3.16: Interesting initial north and east PID performances.

The graph of the best initial north and east PID gains is shown in Figure 3.16a. The time plotted is the time *before* a success or failure, so time $t = 0$ indicates the point at which the drone's planar position has been sufficiently close to the landing platform's planar position for 3 seconds. The distance plotted is the difference in the planar positions of the drone and the landing pad. The perceived distance to the landing pad steadily and predictably decreases without salient local minima or maxima, which indicates that there is no positional overshoot. On the other hand, Figure

3.16b shows the performance of badly-performing PID parameters. Of the 5 approach attempts, only 3 were successful within the 45-second time limit, even with initial conditions which were negligibly different from those in Figure 3.16a. Moreover, the salient local minima and maxima in the perceived distances shows that the drone approached the landing pad quickly, but overshot the target position and had to backtrack, which significantly increased the time and energy required for the approach. It is also important to note that these gains pertain to the most ideal scenario of landing on a stationary landing pad with no wind or other external factors.

Figure 3.16 additionally shows that the initial distance estimates (and inherently the initial pose estimates) of the drone relative to the landing platform are inaccurate. This means that the landing pad - the WhyCon marker in particular, since it is identified first - can be recognized in the drone's field of view before the pose can be accurately determined. This can be seen in all of the depicted approaches (and indeed in all of the test results) via the phenomenon that the initial distance estimate of about 10 meters, when the landing pad is just barely identifiable, jumps to about 18 meters when it becomes *properly* identifiable. A number of factors are at play in causing this: namely the relatively small pixel area of the WhyCon marker within the drone's field of view, the angular displacement between the camera and the landing pad, and the motion of the camera as the gimbal controller attempts to aim the camera at the landing pad. A key aspect of the success of these tests is that the drone is moving towards the landing platform at the point when it is initially identified. This allows the pose estimation accuracy - and inherently the distance estimation accuracy - to increase over time.

3.5.2 Up Velocity Controller

The "up" PID controller is used to control the downward velocity of the drone during descent towards the landing platform. It is oriented upwards instead of downwards in order to conform to the typical east-north-up coordinate system orientation that is used in MAVROS and ArduPilot. As in the initial tuning for the north and east velocity PID controllers, the main goals in setting the initial gains for the velocity PID controller in the "up" direction are to prevent overshoot, prevent undershoot, and minimize time and energy spent. The most important of these goals is to prevent overshoot, as overshoot in this scenario means hitting the landing pad at higher than expected velocity.

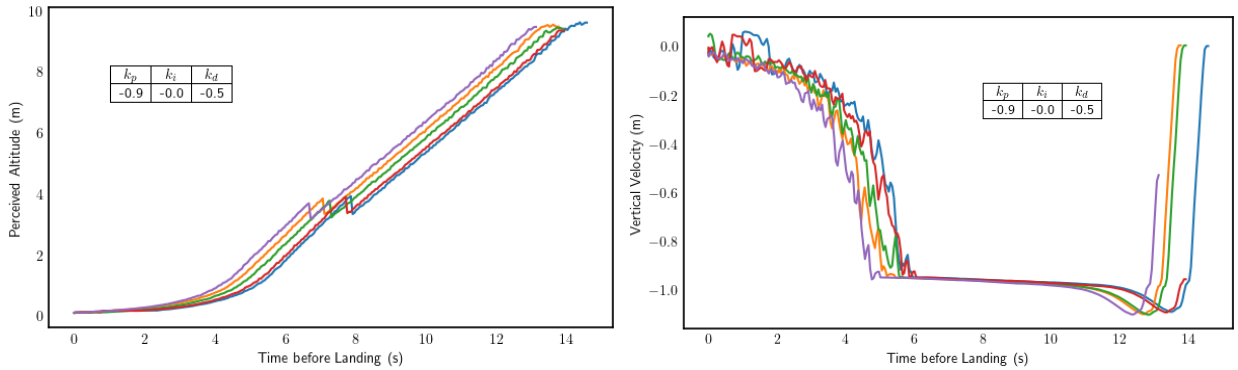
Similarly to the north and east PID testing, the process of testing the up PID was automated by a Python script `control_pid_u.py`. The landing platform was positioned 30 meters directly north of the origin, and the drone was positioned directly above the landing platform at an altitude of 10 meters with the camera aimed directly at the landing platform and the landing controller disabled. At the point when the landing controller is enabled, a single test is considered to have begun, and the Python script records the position and velocity of the drone throughout the landing until the motors are disarmed. This is repeated for several sets of PD gains. The integral gain is not necessary here because the drone is not affected by persistent non-zero error in this dimension. The tests are repeated for all combinations of k_p and k_d, where $k_p \in \{0.6, 0.7, 0.8, 0.9\}$ and $k_d \in \{0.4, 0.5, 0.6, 0.7\}$. Each combination of gains is tested 5 times, giving a total of 80 runs. During these tests, the base descent radius is increased to 30 centimeters in order to avoid extraneous energy and time expenditure resulting from unavoidable drift in the drone's east and north positions. The best 5 gain combinations are shown in Table 3.7.

Over the 80 tests, the average landing time for each combination is only in the range of $[13.8s, 19.7s]$, with a mean of $\mu_t = 16.06s$ and standard deviation of $\sigma_t = 3.14s$. The corresponding values for the energy expended are in the range $[53.6\text{hJ}, 73.4\text{hJ}]$, with mean $\mu_e = 61.75\text{hJ}$ and standard deviation $\sigma_e = 8.33\text{hJ}$. This means that the performances of all tested gain combinations

k_p	k_i	k_d	t_{ave} (s)	e_{ave} (hJ)
-0.9	-0.0	-0.5	13.8	53.6
-0.8	-0.0	-0.4	13.8	54.0
-0.8	-0.0	-0.5	14.3	56.2
-0.9	-0.0	-0.7	14.4	56.0
-0.7	-0.0	-0.4	14.6	57.0

Table 3.7: Best gain combinations for up velocity PID controller.

are fairly similar, however the gains $k_p = -09, k_i = -0.0, k_d = -0.5$ are chosen according to the aforementioned goals. The performance of the best combination of gains is shown in Figure 3.17.



(a) Perceived altitude vs. time before landing.

(b) Vertical velocity vs. time before landing.

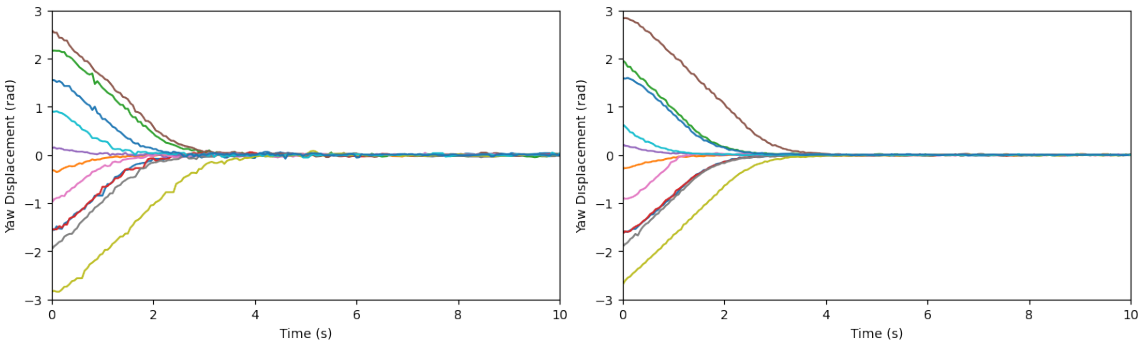
Figure 3.17: Visualization of best U PID gain performance.

Figure 3.17a shows the near-linear descent of the drone towards the landing platform in the early part of the landing, with a gradual slowdown as the drone gets closer and closer to touchdown. Figure 3.17b shows the same phenomenon in terms of velocity. The magnitude of the velocity increases suddenly after the landing controller is enabled, then remains relatively constant until the derivative component of the PID system slows the velocity to 0.

3.5.3 Yaw Velocity Controller

The gains for the yaw velocity controller were determined experimentally through tests similar to those for the other PID controllers. Specifically, the test is automated using a Python script `control_yaw_testing.py`. First, the drone is positioned directly above the landing pad at an altitude of 10 meters. With the landing controller disabled, the drone rotates to 10 different initial angles which are equally placed in the interval $\theta_{initial} \in [0, 2\pi)$. The landing controller is then enabled, with the linear velocity set points always equal to 0, in order to isolate the behavior of the yaw PID controller. The time and yaw displacement are recorded for 10 seconds. The process is repeated at the same location for the remaining initial angles. The entire process is then repeated for multiple positions.

Since the Iris model has very accurate control over its yaw position and velocity, the yaw velocity PID controller performs very well. The results are illustrated in Figure 3.18.



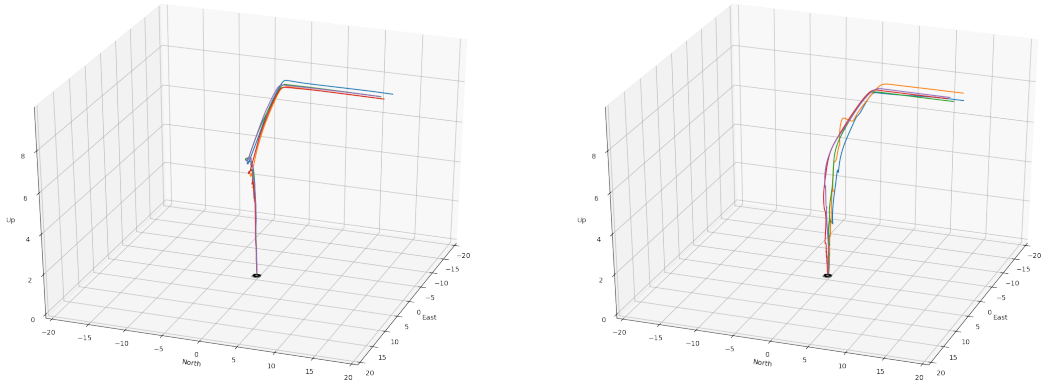
(a) Yaw correction with drone directly above landing pad at altitude of 10 meters.

(b) Yaw correction with drone 2 meters east of landing platform at altitude of 7 meters.

Figure 3.18: Visualization of velocity yaw controller performance.

3.6 Stationary Landing Scenarios

The first test of the landing system in its entirety is a series of 5 landing sequences starting with the drone at the origin and the landing platform 30 meters directly north of the origin. These are essentially a repeat of the test for the north and east velocity PID systems, but with the up and yaw PID systems active. The results for the initial gains are visualized in Figure 3.19a. After initial descent, it is easy to see a sharp inflection point representing positional overshoot overshoot. Although the parameters do allow the system to land successfully, the overshoot is not ideal. The system parameters were re-tuned manually over the course of several experiments. The descent region was slightly enlarged, giving the drone more area to continue its descent without stopping, and causing earlier PID reconfiguration. The derivative gain k_d was increased in phases 2 and 3 in order to slow the drone more drastically. The proportional gain was decreased in phase 2, also to slow the drone's approach. The resulting, smoother performance is shown in Figure 3.19b, and the performances are compared in Table 3.8.



(a) Initial parameters (positional overshoot).

(b) Looser, more efficient parameters.

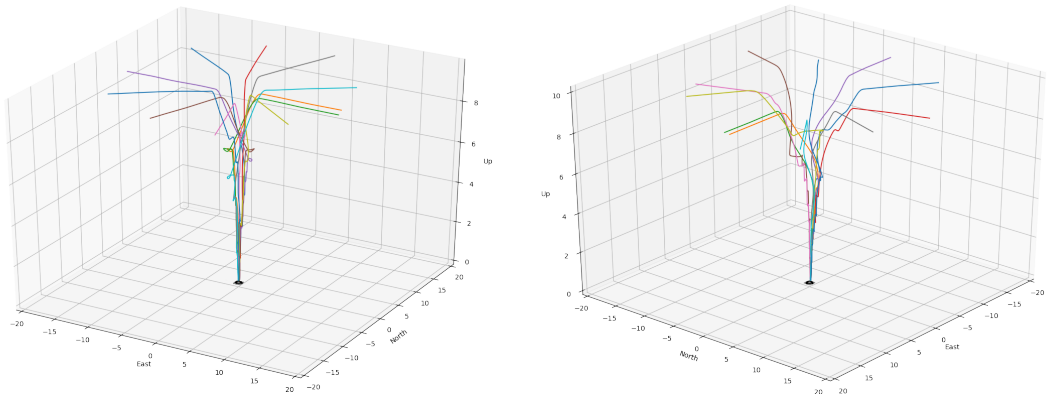
Figure 3.19: Comparison of initial parameters versus manually-tuned parameters.

Figure 3.20a represents the landing trajectories taken by the drone over a series of 10 landings from various angles around the landing platform using the initially-determined PID gains and system parameters. Gazebo provides the coordinates of the Iris model via a ROS topic and these are used

	k_{p1}	k_{i1}	k_{d1}	k_{p2}	k_{i2}	k_{d2}	k_{p3}	k_{i3}	k_{d3}	μ_t (s)	σ_t (s)	μ_e (hJ)	σ_e (hJ)
Initial	-0.7	-0.0	-0.4	-0.3	-0.0	-0.7	-0.3	-0.0	-0.8	39.8	0.65	130.0	2.1
Improved	-0.7	-0.0	-0.4	-0.3	-0.0	-0.8	-0.3	-0.0	-0.9	35.6	1.8	112.8	7.5

Table 3.8: Comparison of initial and manually-tuned gains and performances during 5 linear landing tests.

as a true value for the position of the drone. The landing pad is positioned at $(x, y, z) = (0, 30, 0)$ in the ENU coordinate frame starting at the origin of `sandbox.world`. For clarity, the plotted trajectories begin at the moment that the drone first detects the landing pad and continue until the drone has landed. The approaches begin at a height of 10 meters. A Python script places the drone initially in 10 equidistant starting points which are positioned around the landing pad at a radius of 30 meters. The landing controller is then enabled and the drone is sent towards the landing pad. Upon detection of the landing pad, the landing controller takes control of the drone until it lands and disarms, at which point the drone begins another landing test at the next starting point.



(a) Initial parameters (positional overshoot). (b) Looser, more efficient parameters.

Figure 3.20: Radial landing tests.

	μ_t	σ_t	μ_e	σ_e
Initial	37.5	3.06	120.5	12.2
Improved	36.8	4.38	117.8	17.3

Table 3.9: Caption

A key result from this test is that the landing trajectories are not completely smooth, particularly in the cases where the drone's yaw is significantly different from that of the landing platform. This is because, when the drone orients its yaw to that of the landing platform, the drone's angular velocity means that the PID control efforts in the north and east direction are not fully aligned to the landing pad. This causes the drone to drift slightly out of the descent region, stopping the descent and giving the drone more time to correct its position. However, in all cases, the drone lands successfully.

Figure 3.21 shows the trajectories of these same landings from the point of the view of the drone's pose estimation system. The perceived positive displacement in the north direction, and small perceived displacement in the east direction describe the fact that the drone is approaching

the landing platform head-on.

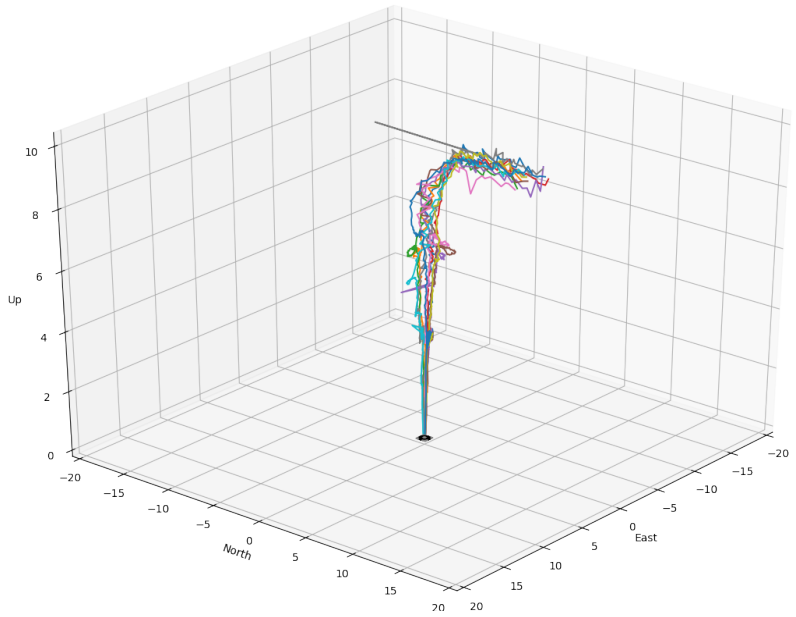
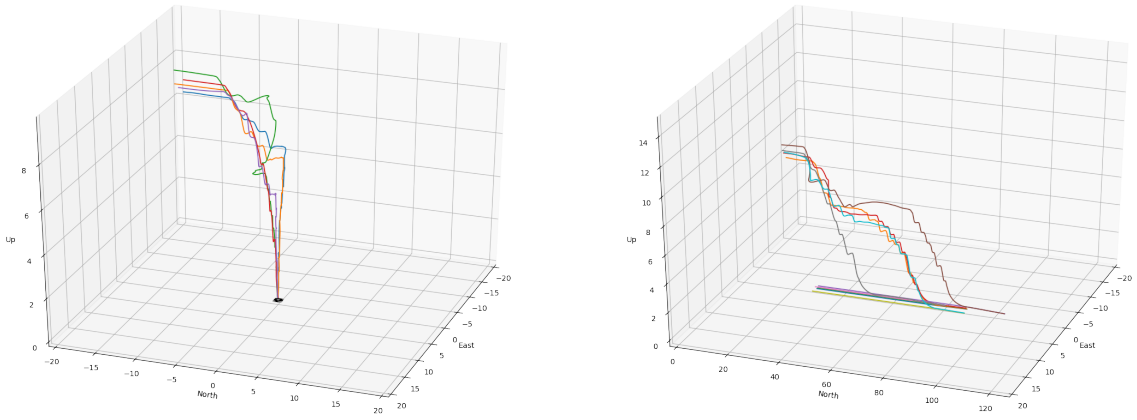


Figure 3.21: Estimated trajectories of the initial landing sequences with respect to the drone's position and orientation.

The descent region for this series of landings is defined using Equation 2.3 with $k_1 = 0.12$ and $k_2 = 0.36$ in order to allow for descent starting at a planar distance of about 4.4 meters from the landing pad when the drone is at an altitude of 10 meters. This does mean that the drone is still allowed to descend when displaced by up to 0.12 meters from the center of the landing pad at an altitude of 0 meters. These constraints can be changed to suit the conditions of the landing pad and the performance of the drone.

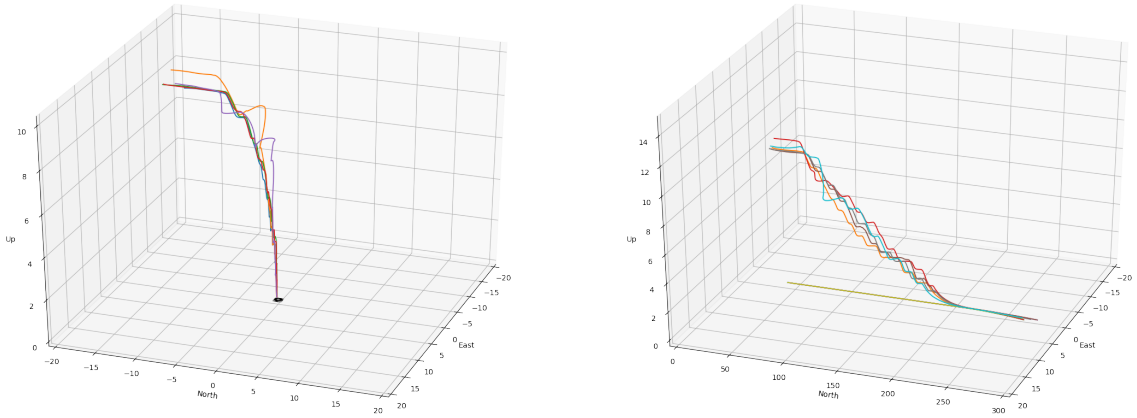
3.7 Moving Landing Scenarios

In order to test the landing system in the context of a moving landing platform, a test similar to the stationary linear landing tests was carried out. A Python script `control_moving_landings.py` automates this process. The drone is first positioned at the origin at an altitude of 10 meters, with a yaw of -1.6 radians (meaning that it is pointed directly south with the landing system disabled. The landing platform is positioned 30 meters directly north of the origin. The drone is then sent to a point 200 meters north of the origin, facing at a yaw position of 1.6 radians (directly north), with the landing system enabled. Before the drone identifies the landing platform, the landing platform begins to move directly north at a given speed which is specific to each test. The time, positions of the drone and landing platform, energy consumed, and the pose estimate of the landing platform relative to the drone are recorded from the moment that the drone recognizes the landing platform. When the drone recognizes the landing platform, the landing controller takes control and directs the drone to land on the moving landing platform. The test stops when ArduPilot detects the landing and automatically disarms the motors, at which point the drone returns to its original position and orientation at the origin, the landing platform is sent to its original position and orientation, and the test begins again. Figure 3.22 visualizes the trajectory of the drone relative to the landing pad over the course of 5 landings wherein the drone is moving at a constant velocity of 1 $\frac{m}{s}$.



(a) Relative landing trajectory. (b) Absolute landing trajectory.

Figure 3.22: Moving land tests with landing platform speed 1 $\frac{m}{s}$



(a) Relative landing trajectory. (b) Absolute landing trajectory.

Figure 3.23: Moving land tests with landing platform speed 5 $\frac{m}{s}$

For landing at 1 $\frac{m}{s}$, the north and east velocity PID controllers required an integral gain of $k_i = 0.22$ in the last phase of landing. The landings required 61.09 seconds and 213.6 hJ on average. As shown in Figure 3.22a, the drone initially recognized the landing platform at a planar distance of about 15 meters away, at an altitude of 10 meters. Figure 3.22b shows the distance over which the drone executed the landing, which was about 60.89 meters - consistent with the time required when considering that the landing pad was moving at 1 $\frac{m}{s}$. Figure 3.23 shows a similar test, with the landing platform moving at a speed of 5 $\frac{m}{s}$. The landings required 45.22 seconds and 159.8 hJ on average. The better performance is likely due to the fact that k_i was set to 0.1 in both phases 2 and 3, which allowed for a generally smoother descent than in the 1 $\frac{m}{s}$ test, where k_i was 0 in phase 2. The PD control in phase 2 meant that the integral component of the control effort started increasing only in phase 3. This slowly-increasing, long-term integral effect should be considered in tuning the velocity PID controllers in a physical system.

3.8 Radial Landings in Wind

The ArduPilot Gazebo repository provides a plugin for simulating changing wind, which was leveraged in order to see the resulting change in behavior of the drone during landing, which is especially important in real-world landing scenarios where wind will always be present. Another radial landing test was carried out in the presence of wind. The horizontal components of the wind are biased with base velocities of 1 $\frac{m}{s}$ in both the north and west directions. A sine function with an amplitude of 1 $\frac{m}{s}$ and a period of 20 seconds changes the magnitude continuously. Gaussian noise with a mean of 1 $\frac{m}{s}$ and standard deviation of 0.5 $\frac{m}{s}$ is also added. The direction of the of the wind also changes over time with both a sine function with amplitude 1 rad and a period of 10, as well as with Gaussian noise with a mean of 1 rad and standard deviation of 0.5 rad. Figure 3.24 shows the trajectories of the drone during this test, required 40.7 seconds and 137.5 hJ on average.

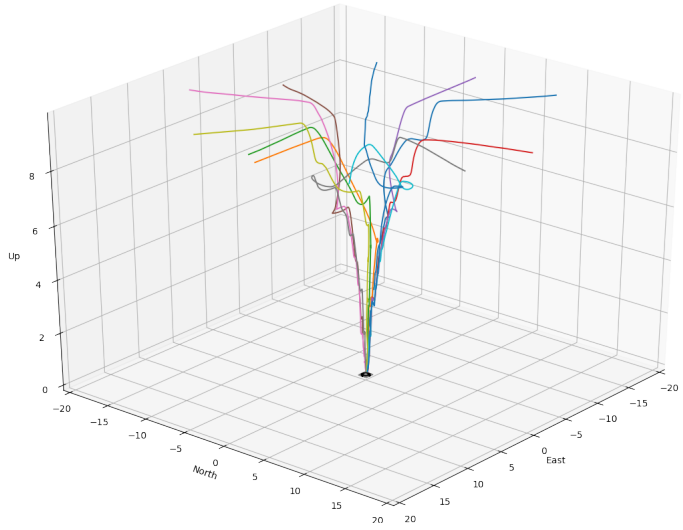


Figure 3.24: Trajectories of the drone relative to the landing platform over 11 landings with wind.

It was necessary and beneficial to add a small integral gain of $k_i = -0.01$ in phase 3 in order to overcome the persistent error caused by the biased wind. This is representative of a real world scenario. It was also necessary to increase the size of the descent region at lower altitudes. The constants k_1 and k_2 from Equation 2.3 were set to 0.24 and 0.36. Additionally, the high wind caused the drone to have a biased attitude in order to maintain its position over the landing pad. This did not prohibit landing, but it did mean that not all of the legs of the drone made contact with the landing pad at the same time. To minimize the effects of this, the altitude at which the drone committed to the landing and dropped to the landing platform was increased from 9 cm to 14 cm. This is important because, if some but not all of the legs are touching the landing platform, the degrees of freedom of the drone are reduced. This reduces the ability of the drone to control its attitude - and therefore its velocity as well. The time during which the drone experiences this reduction in its freedom of movement must be minimized in order to guarantee a stable landing.

Chapter 4

Future Migration to a Physical System

4.1 Hexacopter Design

The most commonly pictured multirotor drone is typically a quadcopter. A hexacopter, as opposed to a quadcopter, was chosen as the drone type for this project. The hexacopter design's 6 rotors allow for redundancy in the case of motor failure, it has a higher load capacity than an equivalent quadcopter, and its extra thrust and weight give it more control in strong wind. Closed drone systems, such as many of the DJI models, function as "black boxes" which is a problem in this case, since the goal is to develop a system which interacts with the flight controller. Therefore, only drone models which are highly configurable have been considered.

4.1.1 Basic Drone Hardware Requirements

The specific drone model to be used was chosen based on the following factors:

- **Flight Time**
  Longer flight times obviously permit more testing, which is preferable. Having to recharge batteries between flights also means more flights are required for testing, and overhead time increases.
- **Load Capacity**
  Higher load capacities allow the drone to carry instruments or cameras for later experiments, so that the drone is not only useful in the context of this project.
- **Cost**
  Financial constraints always pose an issue to real-world testing, especially in the context of drone flight in rough weather, where parts may wear out quickly or crashes may occur. Initial cost should be minimized in order to permit replacement and additional purchases of parts.
- **Availability of Parts**
  The parts used must be in continuous manufacture in order to ensure that the drone can still be used in the event of component failure.

Drone System Comparison

Multiple drone platforms have been considered for the physical migration of this landing system. Although it is infeasible and unnecessary to consider all drone models on the market, Table 4.1 outlines the main deciding factors in choosing among an existing 600-size hexacopter system, and

two popular kits of similar size - the Tarot 680 Pro and the Hobby Power F550. The main issue with the existing hexacopter is that it is a few years old. Although it uses the well-reviewed DJI E600 propulsion system, this system is no longer in manufacture by DJI. This means that, if one of the motors or speed controllers fails, it would likely be better to replace all of them for consistency. The Tarot 680 Pro and Hobby Power F550 are also well-reviewed in online stores, and come in bundles with all necessary propulsion components which are affordable. Flight times of the models are those which use standard hardware setups with no loads. Load capacities are taken from the specifications given for each model.[1][2] When the specifications are not supplied, estimates are derived from the performance of typical components that are used with the models. Such is the case with the estimated flight time and load capacity of the Hobby Power F550 with recommended Emax MT2213 motors.[3] The listed price is the typical price of the drone frame as well as a set of typical motors and speed controllers.

Model	Flight Time (min)	Load Capacity (kg)	Base Cost (US$)	Parts Availability
Current Hexacopter	30	2.5	0	No
Tarot 680 Pro	30	2.5	320	Yes
Hobby Power F550	20	2.0	170	Yes

Table 4.1: Drone System Comparison

The $0 cost of the current hexacopter system is misleading, as the fact that the propulsion system is discontinued may mean that unintended costs are quite high - comparable to the cost of building another drone. The HobbyPower F550's smaller size means that it is intended for 11.1 V batteries, decreasing overall load capacity and flight time.

Table 4.2 shows a decision matrix for the selection of one of the considered drone models. Flight time and availability of parts are the factors that are weighted the highest, although load capacity and base cost are also considered. Of course, such a decision is not an exact science, and the variety of available parts influences both price and performance. This is an estimate, but provides a reasonable justification for the choice of drone model. The current hexacopter is not chosen mostly because of its age and inherent lack of available corresponding parts, although it does provide good flight time and load capacity. The Hobby Power F550 is cheap, but does not as much flight time as the other models, and has a lighter load capacity. The Tarot 680 Pro provides adequate flight time and load capacity, as well as a combo which is designed and rated for the given performance. These facts, as well as the fact that its price (including motors and speed controllers), is relatively affordable, also make it possible to create 2 usable drones. These drones will be identical except for differing flight controller setups, as outlined in Section 4.1.3.

Factor	Weight	Current Hexacopter	Tarot 680	Hobby Power F550
Flight Time	0.4	5	5	3
Load Capacity	0.1	5	4	3
Base Cost	0.2	5	2	4
Parts Availability	0.3	0	5	5
Total	1.0	3.5	4.3	3.8

Table 4.2: Decision matrix for the possible drone systems.

[1] DJI E600 Specs: `https://www.dji.com/e600/spec_v1-doc`
[2] Tarot 680 Specs: `http://www.helipal.com/tarot-fy680-pro-hexacopter-frame-set.html`
[3] Emax MT2213 specs: `https://www.rcmoment.com/p-rm4532.html`

4.1.2 Selected Components

- **Airframe:** Tarot 680 Pro, as shown in Figure 4.1a and 4.1b. This is a 695 mm diameter, carbon fiber drone frame with arms and landing skids that fold to make the drone more portable. It is rated for 6 kg of thrust (with about 2.5 kg payload) using 22.2 V batteries and 330 mm propellers. The final diameter of the drone is thus 1025 mm.

(a) The airframe expanded.

(b) The airframe folded.

Figure 4.1: The Tarot 680 Pro Airframe.[4]

- **Motors:** Tarot 4108 (380 kv), as recommended by the model package. These are the recommended motors for the model, with a diameter of $40.6 \approx 41$ mm and stator height of 8 mm, comprising the "4108" model number. They are rated for 22.2 V batteries.
- **Speed controllers:** Hobbywing XRotor 40A, as recommended by the model package.
- **Batteries:** Specific batteries have not been selected. The constraints for selection of batteries come from the selected power components and recommendations for the model. The battery should be a 22.2 V lithium-polymer battery with a capacity of about 10,000 mAh and a continuous discharge rate of at least 20 C. A smaller, 12.6 V battery should supply the flight controller and companion board with regulated power in order to isolate these computational components from transient effects caused by spikes in current draw by the motors.

4.1.3 Flight Controllers

Multiple flight controller setups have been considered for this project. Each flight controller and the corresponding hardware setup have specific advantages and disadvantages, which are explained herein. The software architecture has been designed to accommodate multiple hardware architectures intrinsically, so that multiple hardware architectures may be compared (see Section 2.4).

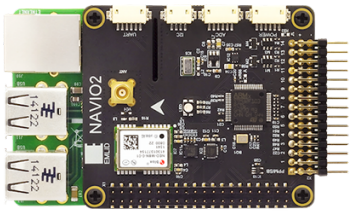
Multiple computational hardware setups will be tested:

- **Navio2 with Raspberry Pi 3 B+**
  The Navio2 is a Raspberry Pi-targeted hat that provides IMU, GPS, universal asynchronous receiver-transmitter (UART), analog-to-digital converter (ADC), inter-integrated circuit (I^2C), and PWM interfaces adequate for controlling drone hardware. It is shown connected to a Raspberry Pi 3 B+ in figure 4.2a. The Navio2 is designed to run the ArduPilot software and has

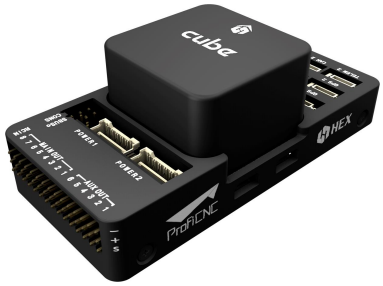
[4]Image source: `http://www.alpha-rc-heli.com/shop/tarot-680-pro-hexacopter-kit/`

a corresponding `waf` board configuration. This is the de facto default setup when running ArduPilot on a Linux computer. The advantages of this setup are that it is extensively tested and relatively easy to extend, in that the Raspberry Pi has multiple USB ports and a camera interface with a specially-designed camera module for computer vision functions. ROS is also tested for Raspberry Pi and the Raspbian operating system. The disadvantages of this setup are that the limited processing power of the Raspberry Pi must be shared between the ArduPilot process, the ROS processes, and the standard Raspbian processes. This is a concern because the ArduPilot and ROS processes must run in near real-time - that is to say, without delays. A possible workaround to this issue is to assign specific cores to the ArduPilot and ROS processes respectively. However, the vision processing involved with detecting the landing pad's fiducial marker(s) can be processor-intensive. Any graphics processing or neural network processing could potentially be exported to a Google Coral Accelerator via USB. While this would add valuable processing power, it would also complicate the system by adding more points of failure in an environment subject to vibration and acceleration.

- **Pixhawk 2 Cube**
  In the case that configuring the Google Coral Dev or NVIDIA Jetson Nano for the Navio2 is infeasible (shown hereafter), these will be used as companion boards for the Pixhawk 2 Cube. The Pixhawk will run ArduPilot on its own and be connected via USB or UART to the companion board, with minor changes to the software configuration, but no changes to the software architecture. The extra board does make the hardware system somewhat more complex, but provides more reliability in that the ArduPilot software will be the sole program running on the board. The companion boards in this scenario will run only the necessary ROS modules to carry out the image processing for the fiducial markers, generate the necessary coordinate system transforms, control the gimbal, and send velocity control messages to ArduPilot on the Pixhawk.



(a) Raspberry Pi 3 B+ with Navio2 hat.[5]

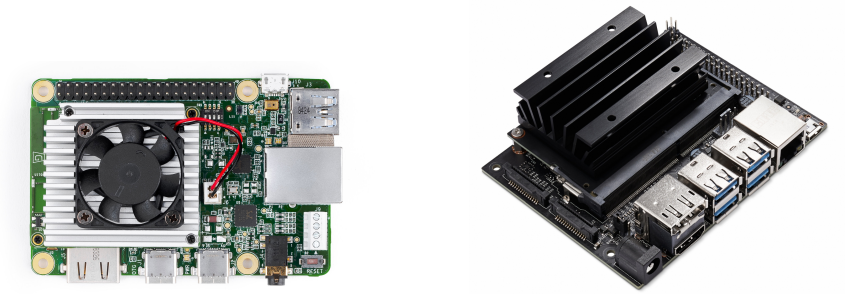


(b) The Pixhawk 2 Cube Hero[6]

- **Navio2 with Google Coral Dev**
  Google's Coral Dev board (shown in figure 4.3a) is similar to the Raspberry Pi in its form but also features Google's Edge tensor processing unit (TPU), which allows for much faster graphics and neural network processing than that of the Raspberry Pi and similar boards. It also has a camera port and specially-designed camera module, but only a single USB port. The advantage of the Google Coral Dev is that the image processing and any potential neural network processing could be exported to the TPU for faster processing, keeping the load on the CPU significantly lower. Since the Navio2 is developed for the Raspberry Pi specifically, some effort will be required to configure the boards to work together.

[2]Image source: `https://docs.emlid.com/navio2/img/Navio2WithPaspberryPi.png`
[3]Image source: `https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk-2.html`

- **Navio2 with NVIDIA Jetson Nano**
  NVIDIA's Jetson Nano is similar to the Raspberry Pi and Google Coral Dev Boards. It uses the Raspberry Pi camera module, has an onboard GPU and is also targeted to embedded artificial intelligence and neural network applications. The image processing for the fiducial markers would be exported to the GPU as much as possible. It, like the Google Coral, is theoretically compatible with the Navio2 after some configuration.



(a) Google's Coral Dev board.[7] (b) NVIDIA Jetson Nano.[8]

Figure 4.3: The computationally stronger companion boards.

4.2 Real World Testing

The simulator allows for testing of the landing and gimbal controllers without consideration of logistical, environmental, and financial constraints. It is therefore "easy" to judge the performance of a particular system in the simulator. However, real world testing requires a significantly more conservative mindset, taking into consideration limited battery capacity and charging time, weather conditions that may hinder or bias testing, and the inevitable risk of crashing and destroying the drone either by human error or system failure. For these reasons, testing must be targeted and agile. Furthermore, priority should be given to tests which can be carried out in the absence of actual flight, in order to minimize risk and logistical efforts.

4.2.1 Drone Construction and Setup

First, the hexacopter system, laid out in Section 4.1 must be properly constructed and tuned. The specific process of constructing the drone is outside of the scope of this project, but emphasis should be placed on ensuring all necessary component functionality and physical security of the onboard components. Special care and independent verification should be taken to ensure the proper calibration of the power sensor in order to guarantee the accuracy of the power readings used to judge the efficiency of a landing, as well as the remaining in-flight battery capacity. The drone's native velocity and position PID control systems must be well tuned to allow for highly reliable positional control, as this is the fundamental basis of the landing control system. The drone should be tested first through conventional RC flight and then through complex point-to-point flight plans. If w_i and w_{i+1} represent the navigational way points between which the drone is currently traveling, cross track error is the distance from the drone to the straight line between w_i and w_{i+1}. Minimization of cross track error can improve the drone's positional control. This

[4]Image source: `https://www.aliexpress.com/item/33025840366.html`
[5]Image source: `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`

can be accomplished through careful tuning of the drone's attitude, velocity, and position PID control systems. All of these PID controllers should be tuned to minimize both correction time and overshoot.

4.2.2 Companion Board Software Setup

The companion boards - in this case, the Google Coral Dev and NVIDIA Jetson Nano must be loaded with Debian Mendel and Ubuntu respectively, and all necessary ROS libraries must be installed for running the landing and gimbal controllers. These modules should be installed from source and most of the dependencies should be installed using the Debian package manager. Specifics for the installation will be included in the Github repositories included in Section A.1. Initial verification of basic functionality includes testing of both the WhyCon and April Tag fiducial systems using connected camera modules to estimate the pose of smaller, test markers from various distances.

4.2.3 Fiducial System Calibration and Initial Pose Estimation

The gimbal must be examined closely to determine the correct method of extracting its orientation data. If it is possible to extract the yaw from the gimbal's IMU then it will be necessary to route this information to the companion board in order to enable calculation of the necessary coordinate system transforms. If the control signal for the gimbal's yaw position provides an absolute position instead of a velocity, then it is possible to use this value instead, with no extra communication overhead. It is not necessary to extract the pitch and roll values, but these could provide good information in the analysis of the gimbal controller's performance.

Once the specifics of the information flow between the gimbal and companion board have been determined, the ability of the gimbal controller to aim the gimbal at the marker should be evaluated, as in Section 3.2. This requires no in-flight operations, and can be done in a lab or other open environment. With the camera-mounted gimbal placed at a high vantage point, several positions should be measured and labeled on the floor, so that their true relative horizontal positions with respect to the camera are known with certainty. Calibration of the April Tag and WhyCon system can then take place according to the guidelines provided by each system. In the case of WhyCon this involves setting the size of the marker, and in the case of April Tag this involves setting both the size and the ID. The pose estimation capabilities of the camera can then be determined using statistical analysis. This test will inherently involve more ambient discoloration and distortion than the simulator, as the physical camera is very likely to have non-zero distortion coefficients, unlike the simulated camera module. With the gimbal controller disabled, each marker should be placed off-center in the camera's field of view. Then the gimbal controller should be enabled and suddenly center the marker in the camera frame, providing an accurate pose with relatively low pose estimation error the entire time.

4.2.4 Flight Controller Integration and Initial Flights

The communication between ArduPilot and the landing controller can be reconfigured using the MAVROS launch file. Instead of communicating with the `sim_vehicle.py` program, MAVROS will communicate with the ArduPilot executable in the case that the system architecture involves only a single board - for example those architectures using a Navio2. In the case that the system has a flight controller board and a companion board, MAVROS will communicate with the ArduPilot executable over USB. Successful communication can be easily verified through inspection of MAVROS topics and ground control station logs which show the specific velocity set point messages that are generated by MAVROS during landing.

Before the landing controller is enabled, in-flight pose estimation tests must be carried out to evaluate the performance of the pose estimation system with real-world motion, vibration, and ambient light. The in-flight performance of the gimbal controller must also be evaluated and its PID parameters may need to be refined, especially to minimize oscillation. In this scenario, the pose estimates will need to be calculated using GPS or some other external positioning system, in the absence of ground truths provided by a simulator. This will necessarily add noise into the system and it should therefore be expected that the pose estimation will be less accurate. The drone's GPS position should be considered as a good estimate for the drone's position. However, the GPS position can be corrected through comparison of the drone's perceived position when it is at known landmarks. For example, if a landmark is at a location with accurately-determined coordinates, the drone's perceived location via GPS can be corrected using the difference between this value and the true position of the landmark. This consideration should be made for each test, as environmental conditions will change the performance of the GPS over time. The estimated coordinates of the landing pad should be compared to known coordinates of the landing pad. The difference in these positions can then be converted to a more readily usable unit such as meters within the drone's local reference frame.

4.2.5 Velocity PID Controller Tuning

The experiments in Section 3.5 can be adapted to suit a real-world scenario. First, tuning of the north and east velocity PID controllers should allow the drone to approach the landing pad and stop directly above it quickly, without overshooting. These tests should be done with a more conservative mindset than that used in the simulator. The parameters should be adjusted after each approach in order to conserve battery and reduce testing time. The yaw PID controller can be tuned in the same way as in the simulator, since each test is relatively quick and power-efficient. The up PID controller should be tuned similarly to the method used in the simulator as well, but with the imperative that overshoot can be fatal for the drone, in that it may cause a crash. This means that parameters should be adjusted with extreme caution, with only small increases in k_p and small decreases in k_d. Unfortunately, the PID tuning requires a human finesse, which adds an element of uncertainty into the process. Throughout this entire process, a human operator must be ready to disable the landing controller and take control of the drone at any time, in order to protect against malfunctions, unpredictable weather conditions, and any other faults.

4.2.6 Landings

Once all of the PID controllers have been tuned, the landing controller can be tested as a whole. Stationary landing tests should take place first, as they are simpler. Then moving tests should be carried out. These tests can take much the same form as those in Sections 3.6 and 3.7, wherein the drone is first positioned so that the landing pad is not recognizable. The landing controller should be enabled manually and then the drone should be directed towards the landing platform in such a way that the landing platform will appear clearly to be recognized in the camera's field of view. Under human supervision, the landing controller should take control of the drone during approach and landing. The human supervisor should retake control in the event of any anomalies. The landings should be conducted from a variety of initial orientations and speeds. Landing performance can be evaluated based on time from landing platform recognition to touchdown, and based on energy consumption. Unfortunately, these tests will inherently be subject to wind other weather conditions, as they must be conducted outside.

Chapter 5

Conclusions

5.1 Future Work

The most pressing future work is to test the proposed landing system on a physical drone as outlined in Section 4.2. This task was originally intended to be a phase of this project, but the COVID-19 pandemic and subsequent close of facilities at Reykjavik University rendered it infeasible within the given time frame.

The destructive interaction between the gimbal controller PID systems and the WhyCode system, mentioned in Section 3.3.1, should be examined and fixed so that WhyCode markers can be successfully employed. It would be especially useful if the WhyCode marker could be made small enough that it would fit in the landed drone's frame of view for the entire landing process, but large enough that it could be recognized from sufficiently far away - so that it could be used as the sole marker for the landing pad. This would decrease the complexity of the landing system and platform. It also is reasonable to suspect that the WhyCon and WhyCode markers will perform significantly better than the April Tag marker once real world ambient light, dust, shadows, and vibration come into play, according to the results of [24], mentioned in Section 1.2.2.

A notable feature which would improve the system's pose estimation is to filter the pose. This could be achieved with a Kalman filter similarly to [20], [21], [27], and, in cases where the landing platform contains multiple fiducial markers, through sensor fusion of all estimated poses. This functionality would be simple to integrate into the existing system and is supported by some existing ROS packages. In addition to filtering the pose, the assumption that the landing platform is exactly level should be eliminated. This can be accomplished through consideration of the pitch and roll components of the camera's orientation during the coordinate system transformations (since the orientation of the IMUs is relative to gravity), thereby applying one additional rotation to the current transformations. This consideration would make the landing controller more robust, particularly in cases where the landing platform is moving over terrain that is not flat.

Different, more adaptable control systems should be tested. The prime goals of this project were proof of concept, system integration, and initial implementation. The sheer amount of subproblems made the task of testing multiple velocity controllers infeasible, given logistical and time limitations. This made PID controllers an attractive option, as they are relatively simple to deal with and implement. However, in most cases they must be manually tuned - a process which is both time-consuming and specific to a small range of system configurations. For example, two seemingly identical drones may require slightly different PID tuning parameters for the same landing scenario, and, as seen in this project, the same drone has different PID tuning parameters for different landing scenarios. Although PID controllers are sufficient in proving the concept, it is likely possible to

implement more flexible, and "smarter" velocity control systems that can be used unmodified (or with only slight modification) in a large range of scenarios and on multiple systems. This is a good application for artificial intelligence and machine learning algorithms, for which the selected companion boards are great options. If Gazebo can be set up to run well in a headless fashion, then the tuning of any numerical parameters of this system via some machine learning algorithm could make a good future experiment. This is because it is difficult or impossible to determine truly optimal control parameters through manual trial and error. Metrics such as estimated power expenses or time from landing pad identification to landing could rate parameter sets in a meaningful way. These parameters could be compared in a variety of stationary and moving scenarios, with differing weather conditions and differing user-determined constraints such as the specific descent area or touchdown velocity.

5.2 Contributions

This thesis achieved the goal of providing a simulator-tested solution as proof of concept for an autonomous landing controller. The solution is robust with regards to the requirements listed in Section 2.1. Using fiducial markers, it can track a landing platform over a wide range of orientations and distances because of the automatically-aimed, gimbal-mounted camera (described in Sections 2.3 and 3.2). It avoids unsafe landings according to the control policy described in Section 2.4.5. It uses a non-invasive software architecture (outlined in Section 2.4) designed to preserve the real-time operating system aspects of ArduPilot, and to allow the system to be easily portable to multiple hardware setups. The method has been tested in the Gazebo 9 simulator with stationary landing platforms (Section 3.6), moving landing platforms (Section 3.7), and in the presence of wind (Section 3.8). The landing controller is developed as a set of ROS modules for which the code is available in Section A.1. The anticipated design of a physical drone for testing is outlined in Section 4.1, and instructions for testing are provided in Section 4.2. Although the system has been tested ArduPilot, it can theoretically work with the MAVLink-enabled PX4 software as well.

Appendix A

Code Repositories

A.1 Code Repositories

All of the code for this project is available on GitHub.

Gimbal Controller	`https://github.com/uzgit/gimbal_controller`
Gimbal Gazebo Plugin	`https://github.com/uzgit/gimbal_controller_plugin`
Landing Controller	`https://github.com/uzgit/landing_controller`
Edited ROS PID controller	`https://github.com/uzgit/pid`
Original ROS PID controller	`https://github.com/AndyZe/pid-release`
Edited ArduPilot Gazebo	`https://github.com/uzgit/ardupilot_gazebo`
Original ArduPilot Gazebo	`https://github.com/SwiftGust/ardupilot_gazebo`
Flight Analysis Scripts	`https://github.com/uzgit/flight_analysis`

References

[1] ArduPilot Dev Team. Precision landing and loiter with ir-lock. (accessed: 2020.6.5). [Online]. Available: https://ardupilot.org/copter/docs/precision-landing-with-irlock.html

[2] J. Wubben, F. Fabra, C. Calafate, T. Krzeszowski, J. Marquez-Barja, J.-C. Cano, and P. Manzoni, "Accurate landing of unmanned aerial vehicles using ground pattern recognition," *Electronics*, vol. 8, p. 1532, 12 2019.

[3] L. Meier. Pixhawk. (accessed: 2020.6.5). [Online]. Available: https://pixhawk.org/

[4] Emlid. Navio2. (accessed: 2020.6.5). [Online]. Available: https://emlid.com/navio/

[5] M. Fiala and M. Fiala, "ARTag, a fiducial marker system using digital techniques," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 - Volume 02*, ser. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 590–596. [Online]. Available: https://doi.org/10.1109/CVPR.2005.74

[6] E. Olson, "Apriltag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3400–3407.

[7] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, p. 2280–2292, 06 2014.

[8] M. Nitsche, T. Krajník, P. Čížek, M. Mejail, and T. Duckett, "Whycon: An efficent, marker-based localization system," in *IROS Workshop on Open Source Aerial Robotics*, 2015.

[9] P. Lightbody, T. Krajník, and M. Hanheide, "A versatile high-performance visual fiducial marker detection system with scalable identity encoding," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 276–282. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019709

[10] R. Paz, "The design of the PID controller," *Klipsch School of Electrical and Computer Engineering*, June 2001.

[11] K. Åström, P. Albertos, and J. Quevedo, "PID control," *Control Engineering Practice*, vol. 9, p. 1159–1161, 11 2001.

[12] J. Demange, "Next-generation positioning for direct georeferencing of multispectral imagery from an Unmanned Aerial System (UAS): applications in Precision Agriculture," Ph.D. dissertation, 06 2019.

[13] A. J. Hanson, *Visualizing Quaternions*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[14] Robot Operating System (ROS). (accessed: 2020.6.5). [Online]. Available: https://www.ros.org/

[15] Transform library (tf2). (accessed: 2020.6.5). [Online]. Available: http://wiki.ros.org/tf2

[16] Gazebo: Robot simulation made easy. (accessed: 2020.6.5). [Online]. Available: http://gazebosim.org/

[17] Ardupilot gazebo plugin & models. (accessed: 2020.6.5). [Online]. Available: https://github.com/SwiftGust/ardupilot_gazebo

[18] Qgroundcontrol: Intuitive and powerful ground control station for the mavlink protocol. (accessed: 2020.6.5). [Online]. Available: http://qgroundcontrol.com/

[19] J. S. Wynn, "Visual servoing for precision shipboard landing of an autonomous multirotor aircraft system," Master's thesis, Brigham Young University, 9 2018.

[20] A. Borowczyk, N. Tien, A. Nguyen, D. Nguyen, D. Saussie, and J. Le Ny, "Autonomous landing of a multirotor micro air vehicle on a high velocity ground vehicle," *IFAC-PapersOnLine*, 11 2016.

[21] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, "Vision-based autonomous quadrotor landing on a moving platform," 10 2017.

[22] K. Pluckter and S. Scherer, *Precision UAV Landing in Unstructured Environments*, 01 2020, pp. 177–187.

[23] R. Polvara, M. Patacchiola, S. Sharma, J. Wan, A. Manning, R. Sutton, and A. Cangelosi, "Toward end-to-end control for UAV autonomous landing via deep reinforcement learning," in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2018, pp. 115–123.

[24] P. Irmisch, "Camera-based distance estimation for autonomous vehicles," Ph.D. dissertation, Technische Universität Berlin, 12 2017.

[25] G. Zhenglong, F. Qiang, and Q. Quan, "Pose estimation for multicopters based on monocular vision and apriltag," in *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 4717–4722.

[26] Edited ardupilot gazebo repository (gazebo 9 branch). (accessed: 2020.6.5). [Online]. Available: https://github.com/uzgit/ardupilot_gazebo/tree/gazebo9

[27] D. Lee, T. Ryan, and H. J. Kim, "Autonomous landing of a VTOL UAV on a moving platform using image-based visual servoing," in *2012 IEEE International Conference on Robotics and Automation*, May 2012, pp. 971–976.