Real Time, Onboard-only Landing Site Evaluation for Autonomous Drones

by

Joshua Springer

Thesis proposal submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

November 2021

Thesis Committee:

Marcel Kyas, Supervisor
Professor, Reykjavík University, Iceland

Gylfi Þór Guðmundsson, Supervisor
Adjunct Professor, Reykjavík University, Iceland

Joseph Foley, Advisor
Professor, Reykjavík University, Iceland

External Person, Examiner
Role, University, Country

Real Time, Onboard-only Landing Site Evaluation for Autonomous Drones

Joshua Springer

November 2021

Abstract

Landing is a last, unsolved problem in autonomous multi-rotor drone flight. Many other tasks such as takeoff, waypoint-to-waypoint flight, and miscellaneous mission tasks (e.g. collection of images and video) have been reliably automated. Yet, autonomous landing remains largely a manual task because of its inherently risky, sensitive nature. The critical result of this is that fully autonomous mission cycles are just out of reach with current technology, in many contexts. Prior work towards autonomous multi-rotor landing has been subject to at least one of several disadvantages - either it depends principally on GPS and is therefore subject to possibly fatal inaccuracy (especially in Iceland), it has relied on detection of special markers (known a priori) with a downward facing camera (such that it may easily lose sight of the markers during approach and descent), it uses differences in pixel speed to deduce terrain topology (but therefore depends on motion), or it has depended on sophisticated ground stations to carry out the computationally expensive processing required for terrain analysis. The proposed research targets the problem of autonomous landing with the goal of creating an algorithm to reliably land multi-rotor drones with the following constraints: 1. having no prior knowledge of a landing site or GPS position, 2. executing in real time with a critical deadline, and 3. using only the limited computational environment onboard a drone.

Thus far, we have created and tested several drone platforms with 3 different flight control software stacks (ArduPilot, PX4, and DJI), which have revealed many challenges of operating drones in Iceland: low GPS accuracy, unpredictable winds, and the commonality of rain. While a large, weatherproof drone system can withstand wind and rain, low GPS accuracy has proven to be a real obstacle. DJI drones and flight controllers have shown to vastly out-perform others in this regard, and thus provide a good base moving forward. Further, we have tested landing algorithms based on fiducial markers in simulation, but differing from previous work in that we use a gimbal-mounted camera to allow it to track the marker over time. This has involved the further development of existing fiducial systems (April Tag and WhyCode) to optimize for execution on embedded hardware, and to test the accuracy of their orientation estimation. Finally, we have created a proof of concept of such algorithms on a physical drone.

Moving forward, we plan to expand on the following research questions: **RQ1.** With what methods can a drone autonomously identify a safe, previously unknown landing site? **RQ2.** What data do such methods require? **RQ3.** How can such methods execute in real time, and in the power-limited environment of a drone? We hypothesize that **H1:** A U-net, or variants such as Residual U-net with pre-/post-processing steps for image rectification and communication with flight control software will be able to recognize landing sights from drone sensor data. **H2:** This data can be point clouds from LIDAR units or RGBD (image + depth) cameras, which are small enough to be embedded onto a drone. **H3:** An embedded TPU, GPU, or FPGA will be able to execute the method onboard a drone in real time. We plan to generate training/testing data sets of LIDAR/RGBD point clouds in AirSim, and collect further testing data sets from the real world using our existing drone platforms. This data will serve as the basis for training several neural network models based on the U-net architecture. Promising algorithms will be optimized using pruning and will be tested for execution speed and power consumption on physical hardware. Any sufficiently fast/accurate methods will be tested in real landing scenarios.

Ég veit ekki hvernig á að tala islensku

Joshua Springer

November 2021

Útdráttur

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Acknowledgements

I would like to thank the Flying Spaghetii Monster and his noodly appendage.

Contents

Chapter 1

Introduction

Figure 1.1: Non-ideal, human-assisted landing in the absence of an autonomous, safe landing method that considers the surrounding environment.

1.1 Problem Statement and Motivation

The goal of the proposed research is to explore the topic of autonomous, unstructured drone landing. Current autonomous landing methods have at least one of the following disadvantages: they are blind to obstacles, they require previously *known* landing sites, they depend on sophisticated ground control stations for offloading of expensive computation. This proposed research targets a gap in current autonomous landing methods. Specifically, we aim to develop a method for quickly analyzing terrain and identifying safe landing sites using only embedded computational hardware and a minimal set of sensors.

Landing is a particularly difficult aspect of drone flight, owing mainly to its risky nature and required precision. As a result, most drone landings are carried out by (or under the supervision of) a human operator, inherently limiting the applicability of autonomous drones. Some autopilot software includes an Application Programming Interface (API) for *precision landing*, which allows a drone to localize and direct itself with respect to a landing pad during an autonomous landing, according to data provided by external sensors and programs. However, there is no particular method of autonomous landing in widespread use. As autonomous and semi-autonomous drones are not able to reliably handle landings on rough terrain or in non-ideal

conditions, human operators often disable autonomous control during landing (opting for full manual control), or abuse/hack the landing system by descending to a low altitude, grabbing hold of the drone, and disabling the motors, as shown in Figure 1.1. Aside from potentially exposing users to dangerous rotors, this landing technique showcases the limitations induced by a lack of autonomous landing method.

In sufficiently flat, large areas, fully autonomous drone missions can end with a GPS-based autonomous landing which is blind to obstacles in the environment. However, intuitively and demonstrably, this can lead to crash-landings at landing sites that have obstacles within the error radius of the GPS, which can be anywhere from a few centimeters to a few meters. In the available open source autopilot softwares, obstacles are simply not handled, and drones will continue their landing attempts even if fatally obstructed.

1.2 Background

1.2.1 Autopilot Software/Hardware

The most prominent, open source drone autopilot software packages are ArduPilot [1] and PX4 [7], which can integrate easily with many additional/custom software packages. DJI drones, while the most commonly used consumer-grade drones, use proprietary, closed source autopilot software that has a limited API for interacting with external software. Thus, ArduPilot and PX4 and custom drones are typically used for research on drones themselves, while DJI software and drones are typically used for consumer/commercial tasks. ArduPilot and PX4 communicate using the same open source, customizable protocol - MAVLink [4] - which has APIs in many different programming languages as well as with Robot Operating System (ROS).

1.2.2 Robotics Software

ROS is a common framework for robotics software that faciliates communication, interactivity, and compatibility between cooperating software modules. It uses a Publisher/Subscriber model of message passing to allow concurrent programs to communicate, and even provides infrastructure for communication between threads running on different hardware platforms. Within the ROS universe are open-source, ready-built tools that are fundamental to common robotics tasks, such as interfaces to many different sensors and cameras, modules for rectification, compression, and transmission of images and point clouds, PID controllers, coordinate system tranforms, and much more. ROS provides a structured means of generating modular solutions to complex robotics problems.

1.2.3 Simulation Software

Simulation allows robotics developers to test algorithms quickly without logistical concerns such as weather, transportation, damage to hardware, etc. Gazebo is one of the most common robotics simulators and provides a physics engine, many simulated worlds and models of items/vehicles, integration with many robotics tools, and - importantly - lots of user-generated content, software plugins, and information. It has integration with ROS, so that simulated sensors and actuators can provide data and take commands from ROS modules. This allows developers to analyze the behavior of their robots during development cheaply and quickly. Additionally, Gazebo has integration with both PX4 and ArduPilot which, combined with existing (and free) drone models, can be used to simulate autonomous drone control algorithms in depth. AirSim is another robotics simulator, but it focuses on providing accurate physics and graphics and is therefore a heavier software package. It also has integration with ArduPilot and PX4, so that

1.2.4 Fiducial Markers

1.3 Related Work

Chapter 2

Current Progress

2.1 Building and Flying Two Hexacopters

After finishing a master thesis [12] wherein I developed an algorithm for autonomously landing a drone using fiducial markers in simulation, the next step was to test this method on physical platforms. The algorithm required identifying fiducial markers through image analysis, tracking the markers via a gimbal-mounted camera, calculating position targets in order to direct the drone towards the landing pad, and communicating those position targets to the flight control software. The base frame for the drones are the Tarot 680 hexacopter kit, which provides a good thrust-to-weight ratio, good flight stability, and space for mounting multiple computational components. A combination of Raspberry Pi and Navio2 [5] shield serve as a flight controller which can communicate with a companion board. The companion boards (a Google Coral Dev board and an NVIDIA Jetson Nano) communicate via a USB network to the flight controller and perform all heavy computations involving image analysis, coordinate system transforms, PID control, and position target generation.

An overview the components is as follows:

- **11.1 V LiPo Battery:** this battery provides power to a battery eliminator circuit (BEC) for isolation of the power system for the computational electronics (the flight controller and companion board).
- **BEC (Battery Eliminator Circuit):** the BEC transforms 11.1V power to 5V power for the flight controller and companion board. The flight controller and companion board each have their own 4A channel to meet their given power requirements.
- **Flight Controller:** this combination of a Raspberry Pi 3 B+ and Navio2 shield runs the ArduPilot software to control the drone, and communicates with the companion board to control the gimbal.
- **Telemetry Radio:** the telemetry radio provides two-way communication between the flight controller and a ground control station that is also fitted with its own telemetry radio. It is connected to the flight controller via USB. The software on the ground control station provides an interface for real-time status messages and sending high-level commands.
- **RC Receiver:** the RC receiver provides a one-way radio link between the pilot's transmitter and the flight controller, allowing the pilot to manually control the drone. It is connected to the flight controller via SBus which provides an 8-channel multiplexed PWM signal to reduce the needed wires and space. This provides an interface for control by a human pilot, which is often used in testing but will eventually be mostly unused.
- **22.2 V LiPo Battery:** this battery provides power to the speed controllers and gimbal.

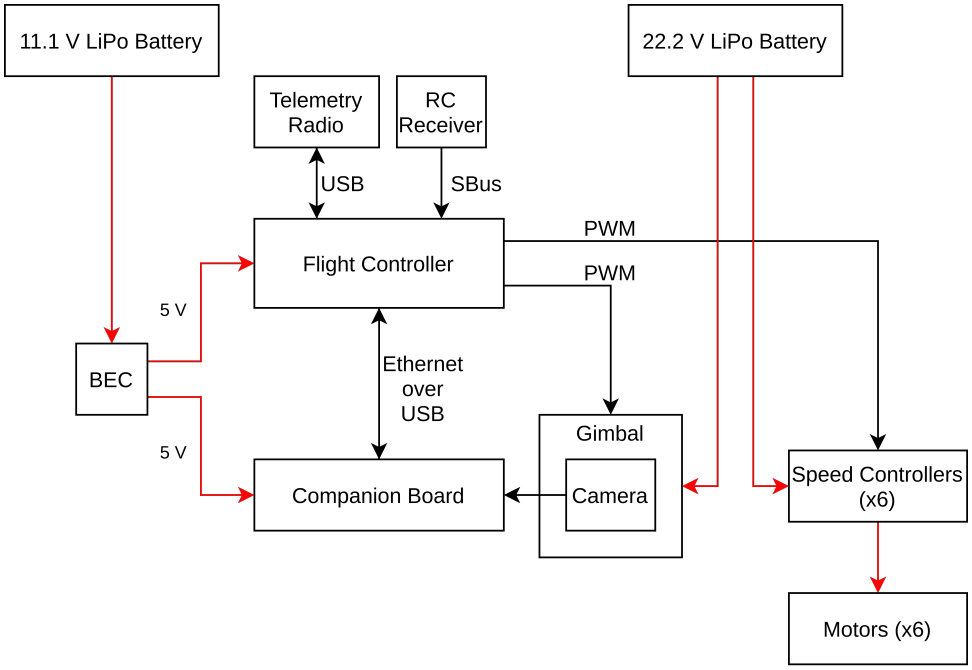


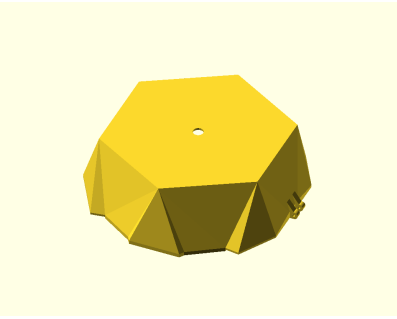
Figure 2.1: Hardware Setup

- **Speed Controllers:** the speed controllers receive a PWM signal from the flight controller which indicates a throttle value. They then provide corresponding power signals to the motors.
- **Motors:** the motors spin propellers to provide thrust in order to control the drone's position in the air.
- **Gimbal:** the gimbal controls the orientation of the camera based on PWM signals from the flight controller which indicate target angles. Its onboard IMU and driver filter the motion of the camera in order to provide a smooth camera image.
- **Companion Board:** the companion board reads an image from the camera and calculates the position of the landing pad relative to the drone. It then communicates this information to the flight controller via an Ethernet over USB connection using ROS.

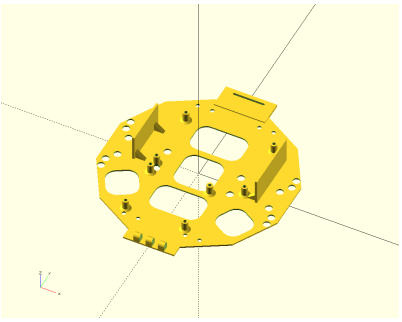
The computational components require some protection from the harsh Icelandic weather, and we therefore designed and 3D-printed a component mounting plate with a connector for a canopy. We also designed and printed cases to protect camera modules and allow them to be mounted in a gimbal with a GoPro form factor. The final versions of these components (after several iterations) can be seen in Figure 2.2. The fully assembled hexacopters are shown in Figure 2.3

The algorithm is implemented as a set of ROS modules, as explained below:

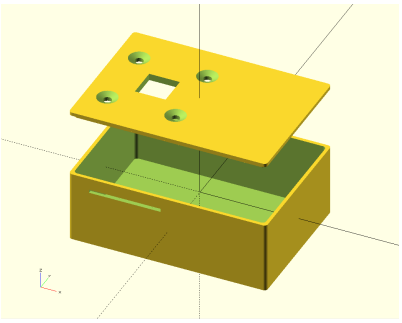
1. `gscam` retrieves camera input and makes it available as a ROS topic,
2. `whycon_ros` analyzes camera images to detect WhyCon/WhyCode markers and determine their pose,
3. `gimbal_controller` reads the poses of any detected markers, passes this information as input to two PID systems, converts the output of the PID systems to PWM outputs, and forwards the PWM outputs to the autopilot software to control the gimbal.
4. `landing_controller` reads the poses of any detected markers, performs coordinate system transforms to generate a target position, and forwards them to the autopilot software.



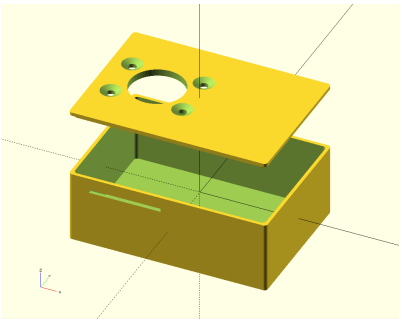
Protective Canopy.



Component mounting plate.



Google Coral camera case.

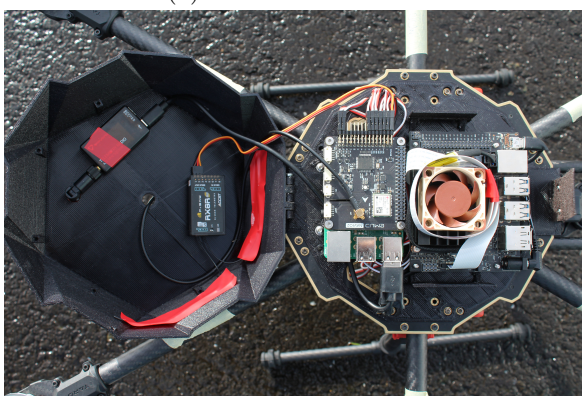


Jetson Nano camera case.

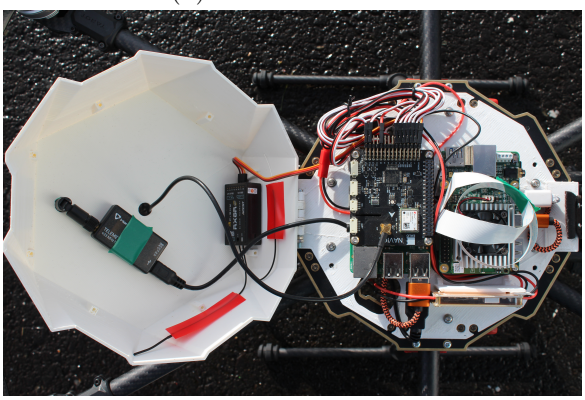
Figure 2.2: 3D printed parts for the Tarot hexacopters.

(a) The Jetson drone.

(b) The Coral drone.

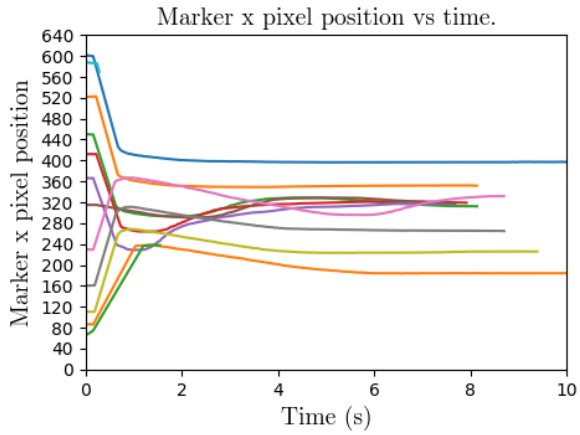


(c) The Jetson drone's electronics compartment.

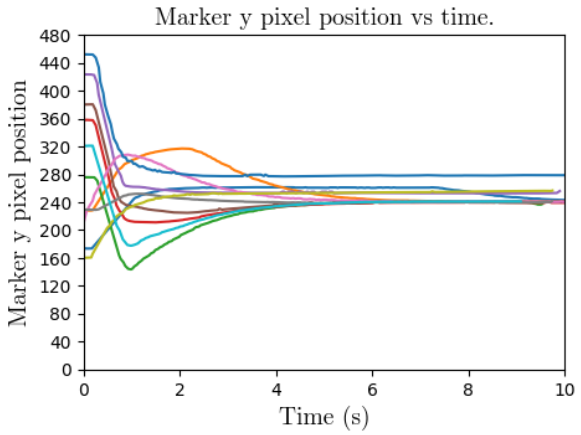


(d) The Coral drone's electronics compartment.

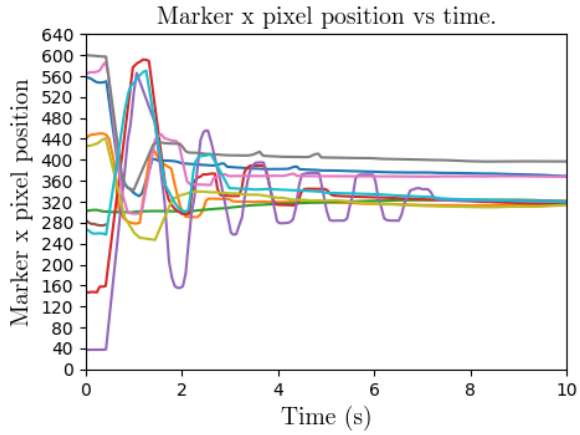
Figure 2.3: The assembled drones and their electronics compartments.



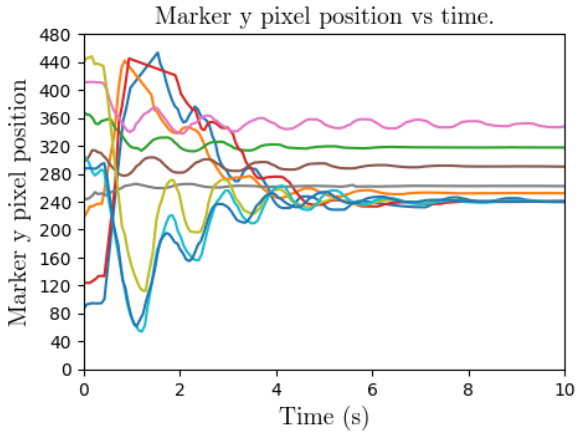
(a) Performance of the Jetson drone aiming the gimbal in the x axis.



(b) Performance of the Jetson drone aiming the gimbal in the y axis.



(c) Performance of the Coral drone aiming the gimbal in the x axis.



(d) Performance of the Coral drone aiming the gimbal in the y axis.

Figure 2.4: Performance of aiming the gimbals.

2.1.1 Results

The drones fly with good stability even in high winds, with an estimated 15-20 minute flight time. [1] In lab tests and in flight, they are able to detect and track fiducial markers using the method tested in simulation. However, only the more lightweight WhyCon/WhyCode fiducial system was used in testing, instead of the April Tag system, because of the time constraints of this summer project. The performance of the drones in tracking the markers is shown in Figure 2.4, where each subfigure shows the position of the marker in the camera frame in the given axis, with a resolution of 640x480 pixels after resizing in order to decrease the computational requirements of the image analysis. The wide angle lens of the Jetson Nano camera module results in much smoother tracking, since each pixel corresponds to a larger distance than with the Google Coral camera module.

The drones are able to approach the landing pad autonomously, but have not yet touched down autonomously. This is due to two principal factors. First, GPS precision is low in Iceland (i.e. geometric dilution of precision (GDOP) is relatively high) because of Iceland's distance from the equator. Methods of autonomous positioning within the ArduPilot framework which were not ostensibly GPS-based are eventually translated into lat/lon/alt position targets, and the drones attempted to navigate to them with GPS only (instead of other methods such as dead

[1] Video of some of the flights and landing attempts can be found at: https://vimeo.com/461576798

reckoning). Since the GDOP was prohibitively high, the drones could not accurately estimate their position. Therefore, they could only follow a coherent path for a limited amount of time when navigating autonomously, after which the trajectory was unpredictable, even if the position target commands were correct.

Second, there is a fundamental challenge with fiducial markers which comes as a result of the limitations of embedding 2-dimensional shapes into 3-dimensional spaces. While the position of the marker in the camera frame can be detected unambiguously, the orientation of the marker cannot. Especially when the marker is almost normal to the camera's view, its orientation becomes increasingly ambiguous in the roll and pitch components. (This can be intuitively visualized by imagining the difference between the appearances of a marker when it is at $(R, P, Y) = (1°, 0, 0)$ versus when it is at $(R, P, Y) = (-1°, 0, 0)$. These orientations would be difficult to distinguish even for the human eye, and much more difficult for a camera with 640x480 pixels.) Tests in simulation revealed this phenomenon, however, because of a variety of factors (a more "perfect" camera/marker/world, better GPS, lack of logistical constraints, etc.), this phenomenon was not prohibitive in simulation, and successful landings were plentiful.

These two problems set the stage for more research and modifications to the fiducial systems, outlined in Sections 2.2 and 2.3.

2.2 WhyCode Modifications

2.2.1 Background

The WhyCon fiducial marker, shown in Figure 2.5 consists of a white circle inside of a larger black circle. The identification algorithm is computationally simple:

- An input image is searched for white regions.
- The detector flood-fills and determines a measure of circularity for each white region.
- The detector seeks a black region directly outside of each white region, flood fills it, and determines a measure of circularity for it.
- The circularity measures are compared, and the white/black region combination is called a WhyCon marker if it is adequately circular (under projection).
- The position of the marker is determined using the pinhole model of a monocular camera with known intrinsic parameters (pixel resolution, lens distortion, principal points, focal lengths), and the diameter of the marker.
- The major and minor axes of the elliptical regions provide a basis from which to determine the marker's orientation.

The WhyCon software's purpose is to determine the pose (position *and* orientation) of detected WhyCon markers with respect to the camera field of view. The position of the markers can be determined with centimeter accuracy in most use cases. However, the full radial symmetry of the WhyCon marker inherently prevents the assignment or recognition of a "yaw" orientation - that is, only two components of the marker's rotation can be determined.

WhyCode expands on WhyCon to add an ID and yaw orientation to the plain WhyCon marker. While the majority of the detection algorithm is the same, an ID is encoded onto the marker in the form of a Manchester encoding that is wrapped around the inner, white circle, as shown in Figure 2.6. By sampling along the circle halfway between the white circle and black circle, the detector can determine the ID encoding. Figure 2.7 shows potential collisions between WhyCode markers that are rotationally symmetric but have different IDs. While it is impossible to distinguish between rotationally symmetric markers, this problem can be overcome

Figure 2.5: WhyCon marker.

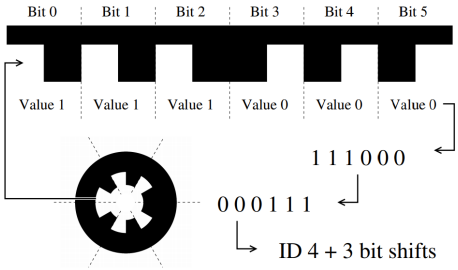


Figure 2.6: WhyCode ID Manchester "Necklace" Encoding.

by bit-shifting the ID's binary string to its lowest value in all cases before determining the ID. This also gives a means of determining a "yaw origin" so that the orientation of the marker can be determined in 3 dimensions.

2.2.2 Orientation Ambiguity

In contrast to the accurately/unambiguously recognizeable position of WhyCode markers, the orientation is ambiguous in many circumstances, as can be seen in Figure 2.8, particularly during time $t \in [4, 8]$. This ambiguity comes from the fact that the orientation is principally calculated from the semi-axes of the marker's ellipses under the assumption that the marker is truly a circle. Calculating the orientation based on these semi-axes gives *two* candidate solutions that satisfy the assumption that the marker is facing the camera, and it is difficult to determine which of these candidates represents reality, especially when the marker is normal or near-normal to the camera's field of view. The original WhyCode software [8] arbitrarily chooses the first solution as correct, and as such it does not need much analysis except to say that it has a high likelihood of choosing the wrong solution.

In the context of fiducial-based drone landing, the issue of orientation ambiguity affects a drone's ability to estimate its own pose when its camera is mounted on a gimbal. Many gimbals do not output their orientations to the flight controller, and their orientations cannot be directly calculated from control signals in many cases, since many gimbals have sophisticated control systems. While a control signal may communicate a *target* orientation, many gimbals have their own IMU and take into account angular motion and acceleration in addition to a control signal. Therefore, it is difficult to determine the orientation of the gimbal simply based on its control

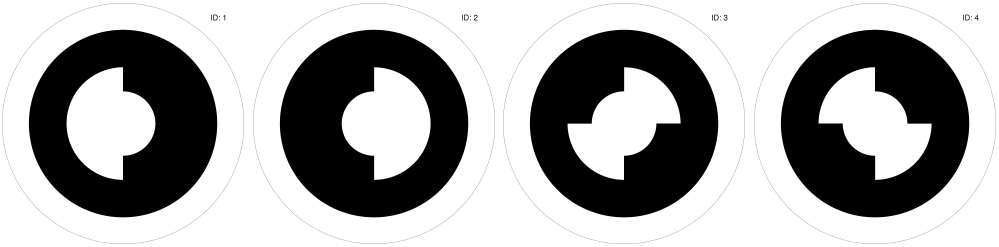


Figure 2.7: Rotational symmetry in 2-bit WhyCode markers.

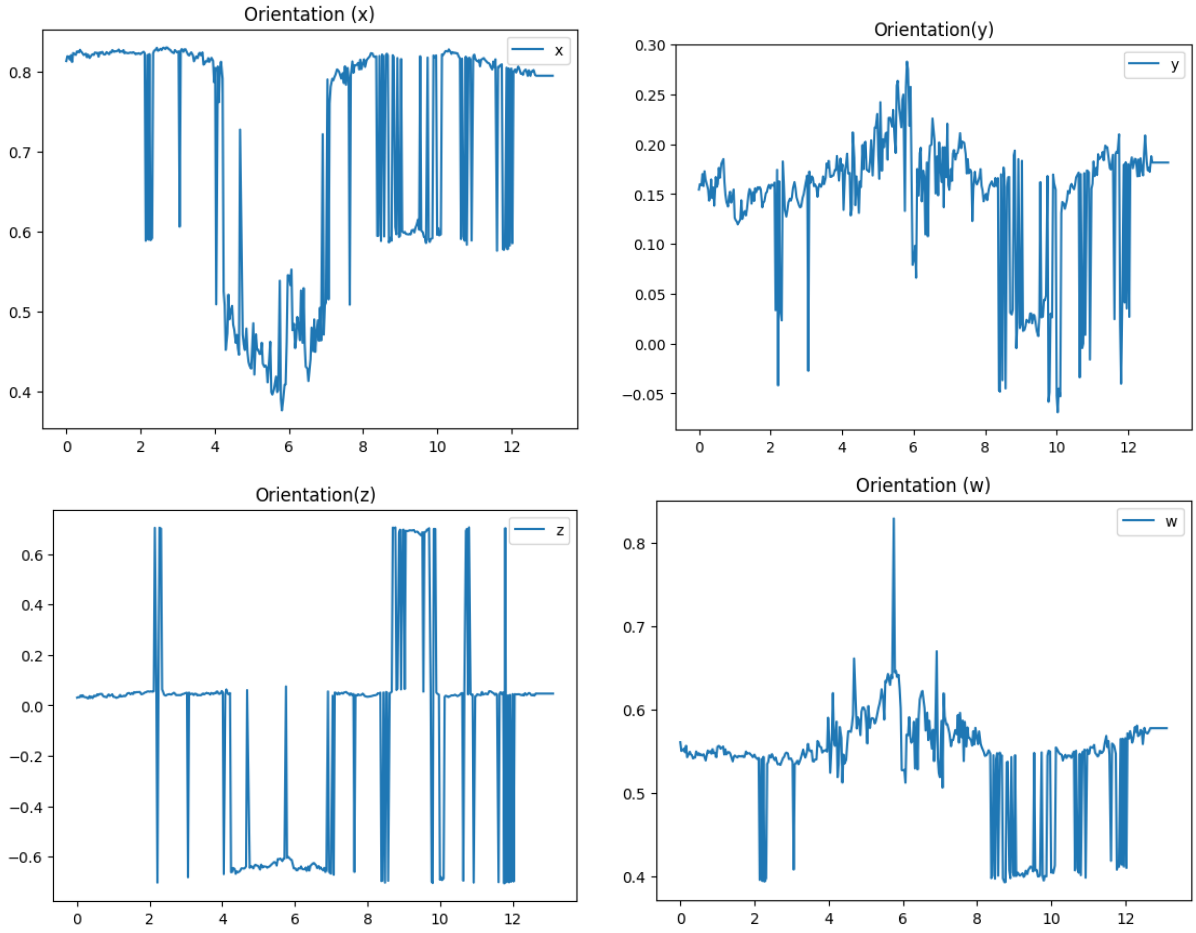


Figure 2.8: Components of the orientation quaternion for a WhyCode marker. Emphasis is placed on the discontinuities in all components, and sign flipping in the Z component.

signals. Without the orientation of either the fiducial marker *or* the gimbal, more assumptions about the landing scenario are required, such as (for example) that the camera is facing directly down always.

When a drone uses a fiducial marker to determine its position with respect to a landing pad without the assumption that its camera is in a fixed orientation, it can (theoretically) use the marker's perceived orientation instead. The drone can rotate the marker's *position* by the inverse of its *orientation* in order to determine its own position in the reference frame of the marker. However, such transformations, which depend on the ambiguous orientation of the marker, also exhibit such ambiguities - discontinuities in the orientation ultimately result in discontinuities in the perceived location of the drone.

Refinements of the WhyCon/WhyCode software, such as that proposed by Jiri Ulrich [13] (shown in) attempt to disambiguate the orientation, with partial success. This method assumes that all "teeth" that form the marker's ID should have the same number of sample points, and therefore chooses the candidate solution that has lower variance of the number of sample points per tooth. The samples are collected along the ellipse that is halfway between the white and black ellipses of the marker. The candidate solutions predict this ellipse to be in slightly different places, owing to the distortion caused by the camera lens. The rationale for the method is that the correct solution should predict the ellipse to be in its correct place, minimizing the variance in the number of sample points that coincide with each tooth. Conversely, the incorrect solution should predict the ellipse to be in the incorrect place, and therefore fewer sample points should coincide with teeth one side, increasing the variance. This provides a baseline on which

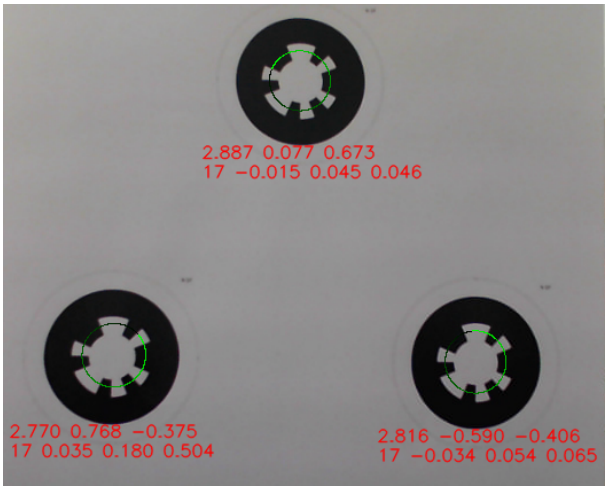


Figure 2.9: Example from Jiri Ulrich's code, showing the circular sampling locations.

to disambiguate the solutions, but does not reliably choose the correct one, especially as the candidate solutions get closer and closer and the marker becomes more and more normal to the camera.

2.2.3 Proposed Solutions

As part of this project, I explore two different ways of solving the orientation ambiguity.

Radial Tooth Edge Sampling

The first method (the `ellipse_sampling` branch of [11]) is similar to that proposed by Jiri Ulrich, in that it is based on decreasing variance in sample points along the teeth. After the two candidate solutions for the orientation and the marker ID have been determined, a second phase of sampling begins. Each candidate solution predicts the center of the inner, white circle in pixel coordinates. (Since this system is designed to be lightweight, the center is not actually calculated, nor needed, to determine the position of the marker, and explicit calculation of this center is neglected by default.) Given the structure of WhyCode markers, we can assume that the circle along which the ID is sampled has a radius of $1.5r$, where r is the radius of the inner circle, and the radius of the circle bounding the teeth is $2r$. Further, from the ID sampling, we can identify the angle that goes through the center of each tooth. This makes it possible to test how well the candidate solution is centered on the marker at many points, and leverages the fact that camera distortion increases (and therefore prediction errors increase in magnitude) as one moves away from the center of the image.

The method does decrease the number of discontinuities in the perceived camera location, as shown in Figure 2.12, at the cost of significantly more sample points (more than double the original number). Unfortunately, it does not eliminate the discontinuities enough to justify the additional computational requirements.

Co-planar Markers

The second method works under the assumption that all markers are co-planar. Since the WhyCode system can accurately and quickly determine the positions of the markers, this assumption allows the orientation of co-planar markers to be determined unambiguously, *when 3 or more markers are detected simultaneously.* For each input image, the WhyCode algorithm identifies all markers and stores their positions. Additional code assumes that the markers are in a co-planar "bundle" and attempts to find the normal vector to the plane implied by the markers' locations

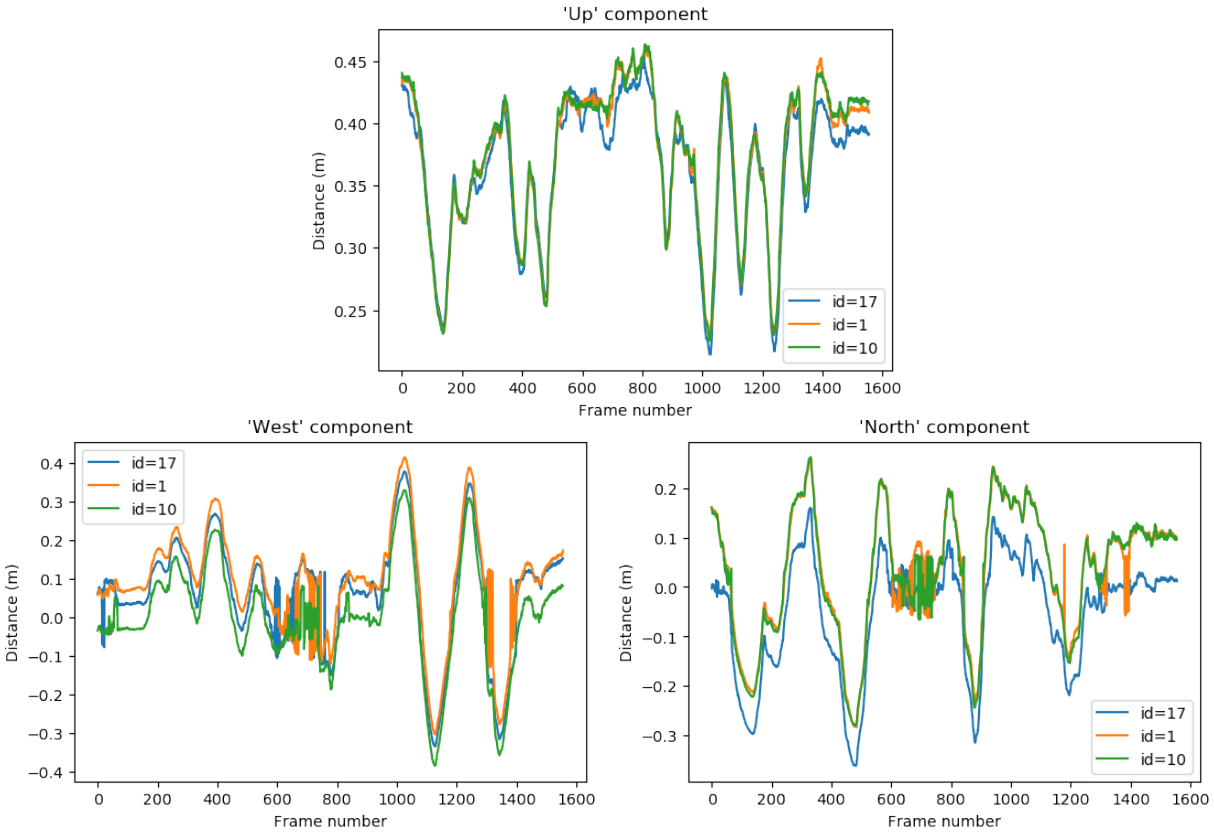


Figure 2.10: Each of the components of the perceived camera translation for 8-bit WhyCode markers with IDs 1, 10, and 17, using Jiri Ulrich's decision metric. The discontinuities occur mostly in the "west" and "north" components between frames 600 and 800, but also between frames 1200-1400.

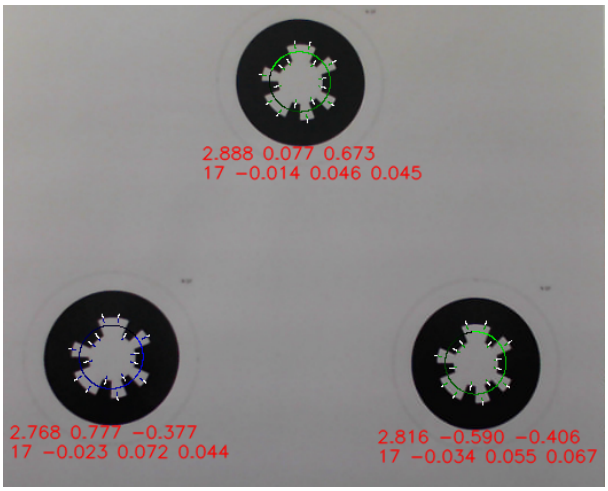


Figure 2.11: Example of sampling WhyCode markers on the radial edges of the teeth. The black/blue circle illustrates ID sampling locations, with the bluest samples occurring first, and blackest samples occuring last. Each of the blue/white radial line segments illustrates the sampling of the predicted edges of the teeth. The method chooses that solution that minimizes the variances of the ratio of blue to white on these line segments.

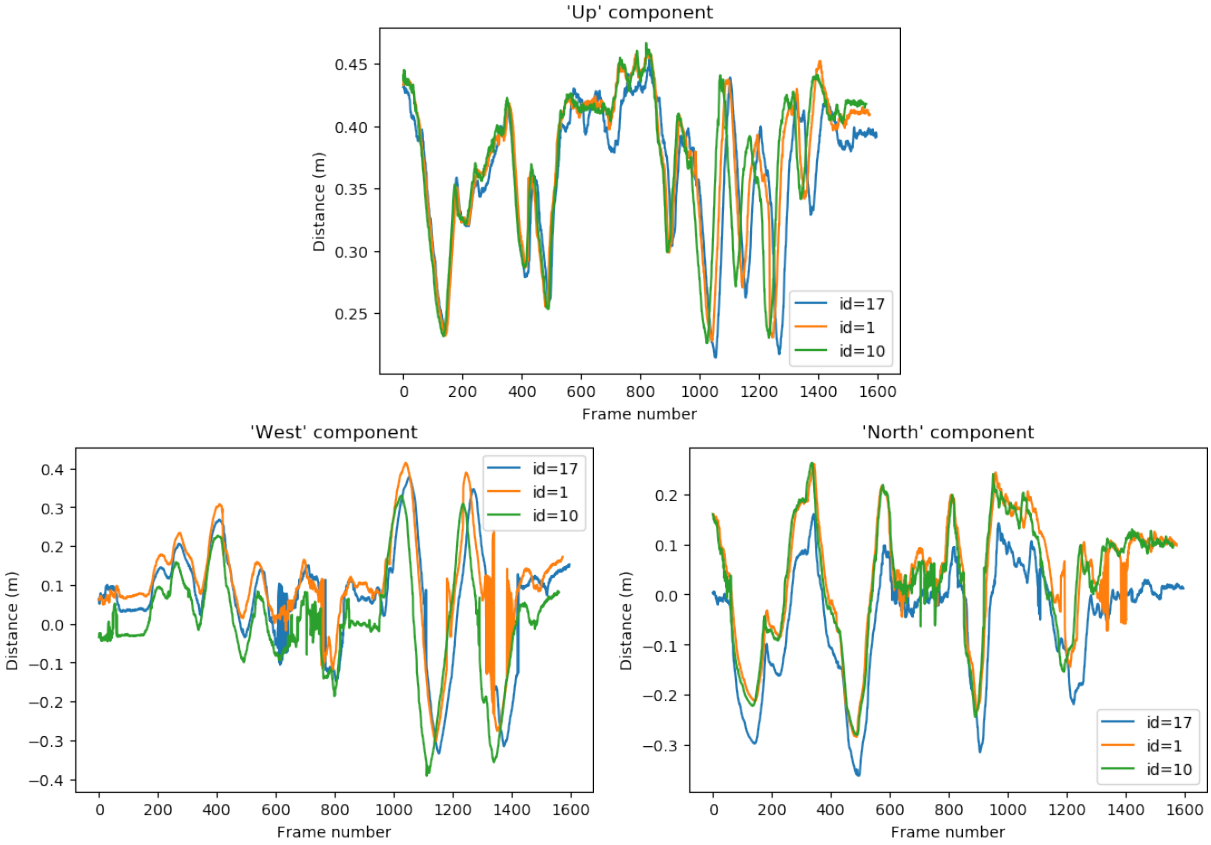


Figure 2.12: Each of the components of the perceived camera translation for 8-bit WhyCode markers with IDs 1, 10, and 17, using the tooth edge sampling decision metric. The discontinuities still occur mostly in the "west" and "north" components between frames 600 and 800, but also between frames 1200-1400.

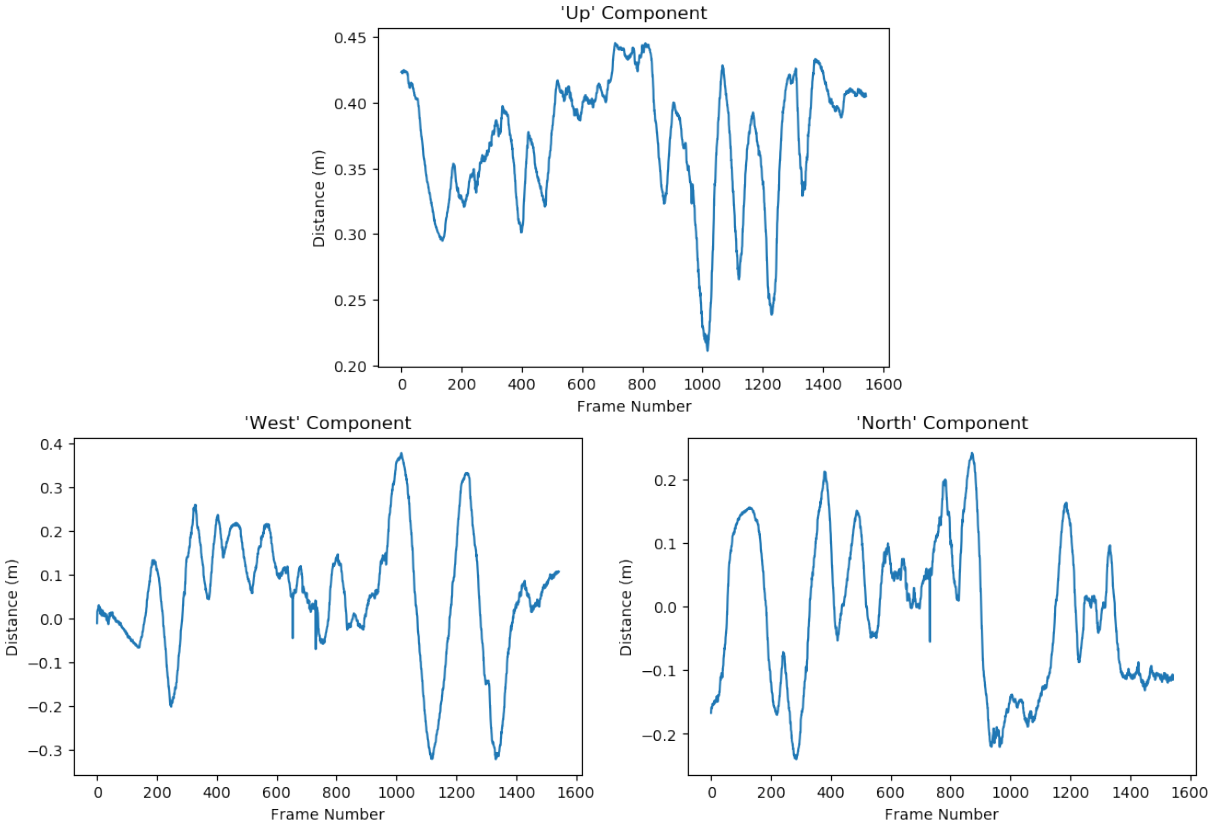


Figure 2.13: Each of the components of the perceived camera translation for a "bundle" of co-planar 8-bit WhyCode markers with IDs 1, 10, and 17. This algorithm significantly reduces the number of discontinuities.

using regression. The normal vector is used to calculate the pitch and roll components of the bundle's orientation. The yaw of the bundle is assigned to be the yaw of the markers, which is determined unambiguously from the ID. The position of the bundle is the average position of its constituent markers.

This algorithm significantly reduces the number of discontinuities in the perception of the camera location, as shown in Figure 2.13. This comes at the computational cost of performing a regression to calculate the orientation of the plane at each frame. It also adds additional constraints: at least 3 co-planar (and non-colinear) markers must be visible at all times. This is problematic with a system such as WhyCode, which does not allow for embedding markers within other markers, because the required view area increases significantly to accommodate 3 markers simultaneously.

2.2.4 Conclusion

WhyCode, though a powerful and computationally efficient fiducial system, is not well-suited to the task of landing a drone with a gimbal-mounted camera. While the system provides accurate, efficient, and scalable position estimation, the ambiguities and discontinuities in its orientation estimation prevent the required pose transformations without prohibitively expensive algorithmic extensions or constraints.

2.3 April Tag Modifications

April Tag [9][14][6] is a popular fiducial marker formed by a series of black and white squares. Several marker families are available, each with its own layout, as shown in Figure 2.14. The

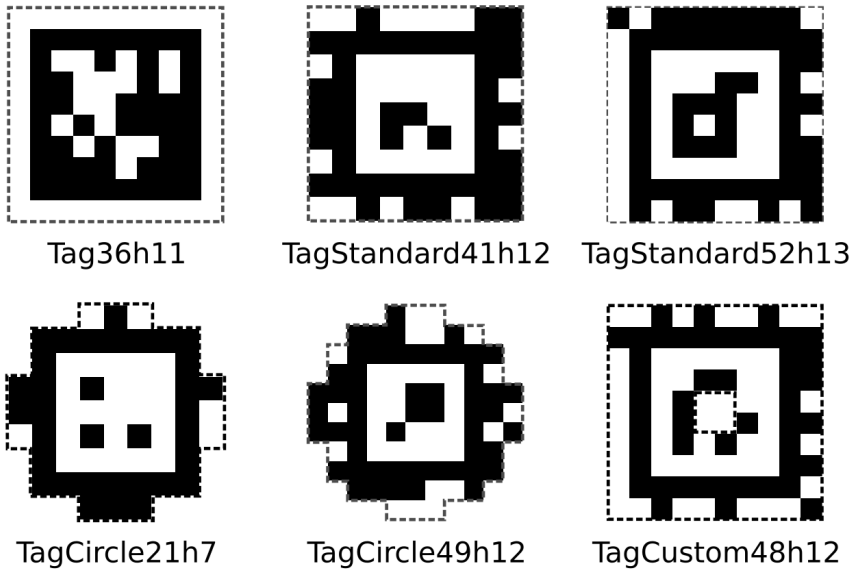


Figure 2.14: Several available April Tag families (from the paper Flexible Layouts for Fiducial Tags [6])

essential parts of the marker include a square border of black squares with a white border on the inside. Other square sections can either be "data bits" which contribute to the ID of the marker, or they can be unused and either not contribute to the marker definition at all, or provide a place for embedding another tag.

April Tag markers, in contrast to WhyCode markers, cannot be recognized independently of their ID. A benefit of this is that each marker can be assigned its own size before runtime, so many markers of different sizes and IDs can be used simultaneously. Figure 2.14f shows the family April Tag 48h12, which has 48 ID bits, and where each valid ID has a hamming distance of at least 12 from the others. When the April Tag family is generated, all arrangements of black and white squares that fit the requirements of the family definition (as well as isomorphic versions) are put into a hash table that is used to determine the validity of detected tags at runtime.

The 48h12 family has 4 undefined bits in its center, where another marker in the same family can be placed. This marker embedding is crucial for drone landing because it allows the drone to have reliable pose estimation throughout the entire landing. For example, if a landing pad uses a single marker that is recognizable from far away, it is likely that the marker will not be visible during the last stages of landing where the marker may eclipse the camera's field of view. Non-embedded markers of different sizes can be placed on a landing pad in non-intersecting positions, but this implies more complications in tracking the markers during landing. Embedding concentric markers in one another and marking them as a tag bundle reduces the complexity regarding coordinate system transforms and tracking. The 48h12 family is therefore a natural candidate for marking landing pads.

2.3.1 Re-testing in Simulation with April Tag 48h12

After moving away from the WhyCode system for the purpose of autonomous landing, I re-implemented the method from the thesis to use only embedded 48h12 April Tags and re-tested successfully in simulation. The re-implementation involved adding to the April Tag ROS code with special focus on tag bundles, marker tracking, and coordinate system transforms. The landing pads are defined as concentric, co-planar tag bundles with the same yaw orientation that

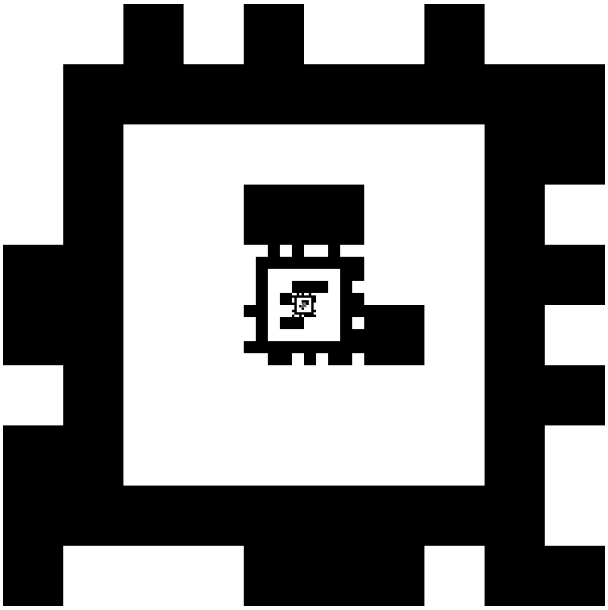


Figure 2.15: The landing pad for the given tests, with 48h12 IDs 3, 2, 1, and 0 in order of decreasing size.

take the position and orientation of their largest detected marker, expose their pixel centers u, v (for PID tracking), and calculate their camera translation (position rotated by the inverse of their orientation) before exposure to other ROS nodes through messages. The code is available at its Github repository [2]. The marker bundle forming the landing pad is shown in Figure 2.15.

The re-implementation was successful in simulation[2], as shown by the landing trajectories in Figure 2.16. Additionally, the marker bundle tracking allowed for robust, uninterrupted tracking during landing, as shown by Figure 2.17.

2.3.2 Testing on Physical Hardware, and the creation of April Tag 24h10

A fundamental limitation of simulation in general is that it does not allow one to test all aspects of a given system. In contrast to simulated experiments on a desktop, which could easily run April Tag 48h12 at ≥ 30 Hz, initial experiments on a Raspberry Pi 3 showed that the system was only able to analyze 640x480 pixel images at a rate of 1.5Hz, which is not sufficient for a precision landing. Additionally, the 42211 markers of the 48h12 family create a huge (> 1 GB) hash table on the Raspberry Pi, needlessly hogging valuable memory. However, no other default family of April Tags allows for marker embedding. This necessitated a new marker family that both allows for marker embedding and has sufficiently fast runtime performance on embedded hardware such as a Raspberry Pi 3 - April Tag 24h10. This family attemps to minimize the size of the markers, maximize the number of marker IDs in the family, and maintain the ability to embed markers. The definition is shown in Table 2.1, and the set of valid markers in Figure 2.18.

2.3.3 Testing with PX4's Precision Land

April Tag is subject to orientation ambiguity, similar to the WhyCon/WhyCode markers mentioned in Section 2.2. When conducting an autonomous landing via a series of position setpoints, such as in the original thesis [12], the orientation ambiguity is not prohibitively destructive - essentially because the orientation is correct "most of the time." However, when using a method

[2]Video from testing this method in simulation with April Tag 48h12 embedded markers can be found here: `https://vimeo.com/manage/videos/526223833`

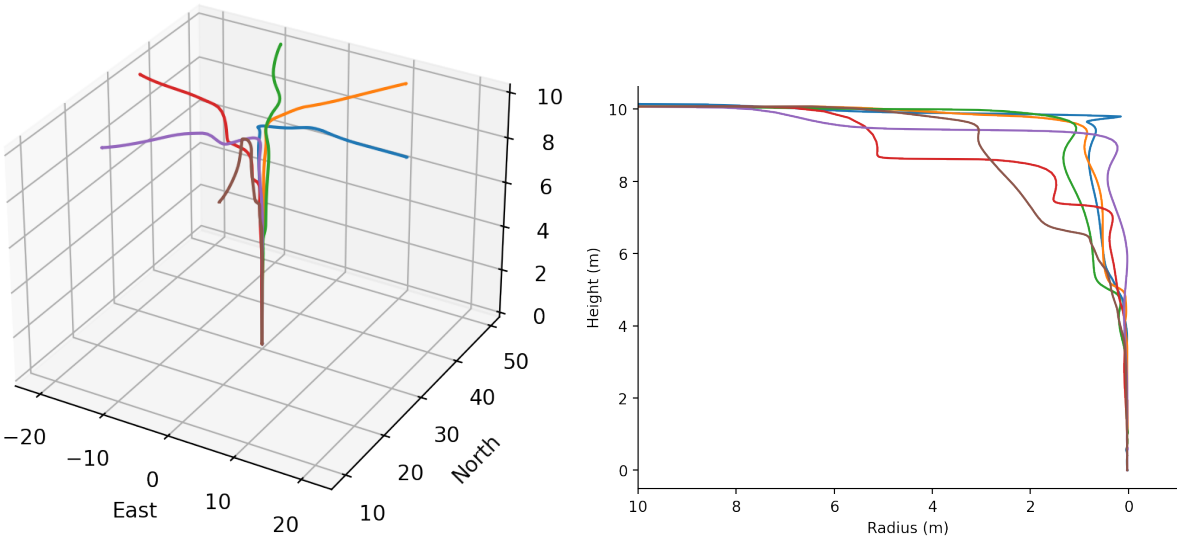


Figure 2.16: Left: landing trajectories for a series of simulated landings, centered around the landing pad. Right: the horizontal distance of the drone from the landing pad versus its height during landing. Both: these landings are executed by the code from the original thesis, using ArduPilot.

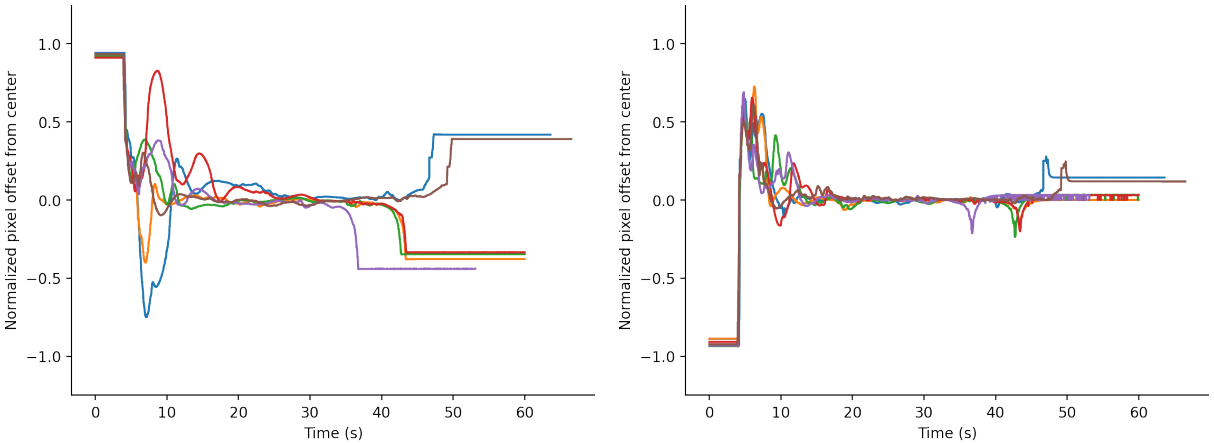


Figure 2.17: Pixel positions of the landing pad marker bundle during the landings depicted in Figure 2.16.

d	d	d	d	d	d	d
d	w	w	w	w	w	d
d	w	b	b	b	w	d
d	w	b	x	b	w	d
d	w	b	b	b	w	d
d	w	w	w	w	w	d
d	d	d	d	d	d	d

Table 2.1: Definition for the April Tag 24h10 family. Each square is marked by a letter representing its function. Data squares are marked as 'd,' white squares as 'w,' black sqares as 'b,' and unused squares as 'x.'

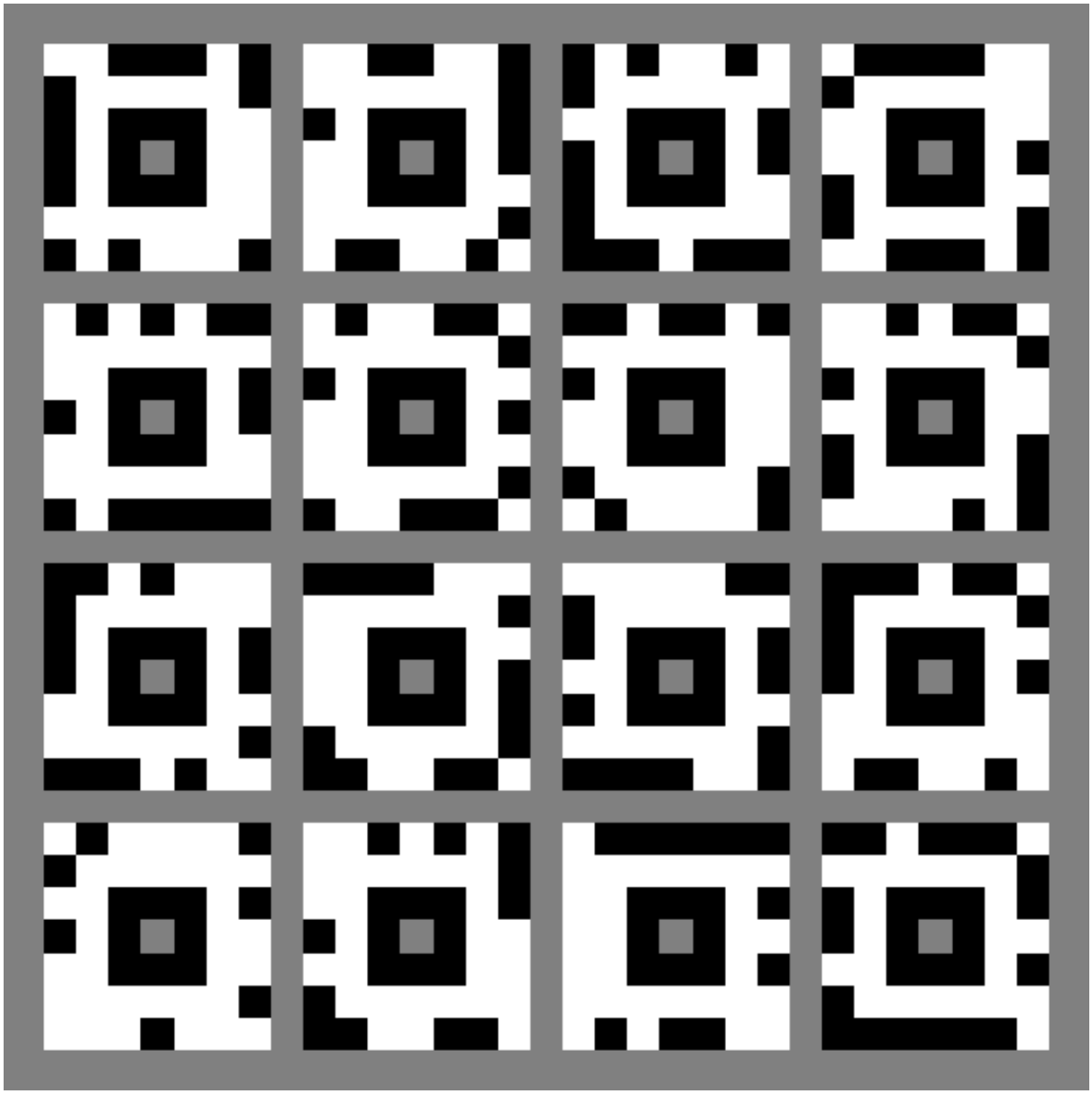


Figure 2.18: The set of markers in the April Tag 24h10 family. The grey squares are not part of the marker definitions.

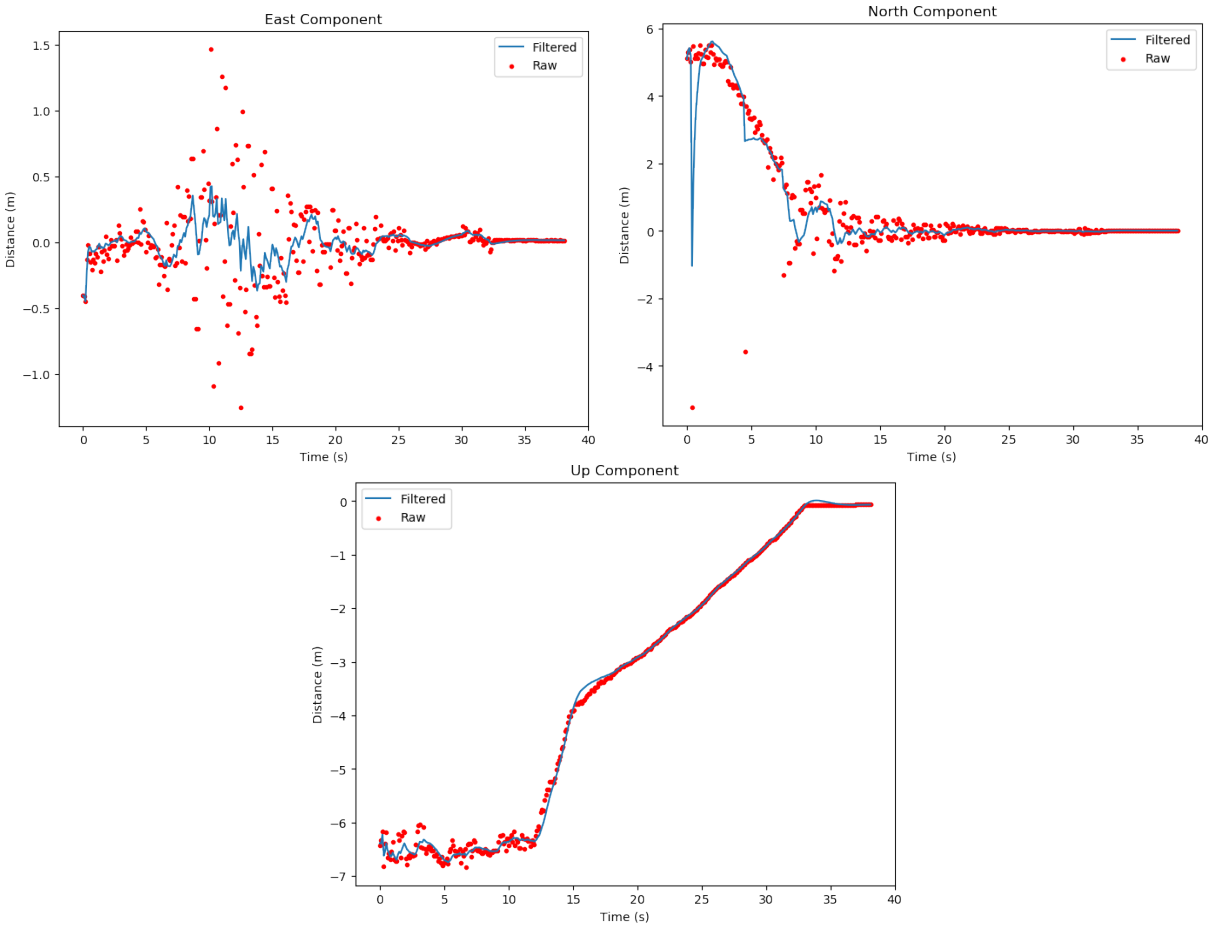


Figure 2.19: Unfiltered and filtered estimates of the landing pad's position during landing.

such as PX4's precision land function, a single orientation flip can make the landing fail. This is because the method involves approaching the landing pad horizontally first (facing the landing pad directly), and then descending vertically once the drone is adequately close. The algorithm assumes it has reached or surpassed the landing pad location once it gets a single erroneous pose estimate for the landing pad.

One way around this issue is to simply filter the raw position estimate data with a Kalman filter, as shown in Figure 2.19. The red data points represent the raw data, and the blue, continuous line represents the filtered position estimate that is passed to the PX4 flight software. Since the drone approaches the landing pad head-on, the "north" component of the landing pad's position is typically the largest. (This is just to say that the landing pad is in front of the drone, not to the side.) Thus, filtering is most important in the case of the north component. If the orientation flips (thereby causing a sign flip in the position), the drone will think that it has already passed the landing pad, and will attempt to land wherever it happens to be when the flip happens. The particularly dangerous data points are therefore the negative outliers in the early stages of the landing. Filtering is useful on the east component to reduce noise. Filtering is not essential on the up component because orientation flipping does not change the perceived height. Variation in the up component is simply considered noise. In spite of the fluctuating raw and filtered values, the position estimate data can be used for precision landing, as demonstrated in Figure 2.20 which shows the horizontal distance from the landing pad versus altitude for multiple such landings. A video of an example landing using the method is available online. [3]

[3]Example landing using PX4's precland: `https://vimeo.com/625488249`

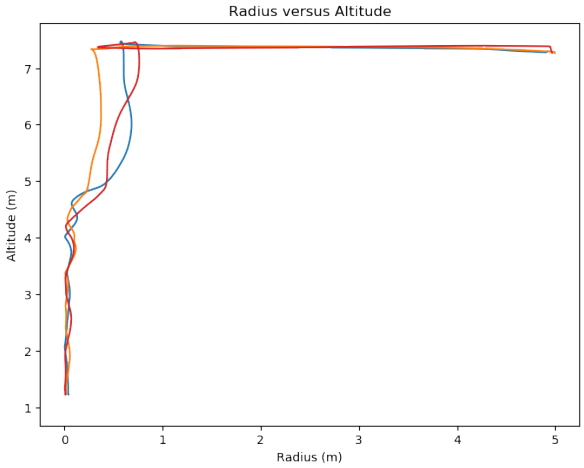


Figure 2.20: Horizontal distance from the landing pad versus altitude for 3 PX4 precision landings using the filtered April Tag 24h10 system. This graph should be read as if the drone is approaching from the right and moving left.

2.3.4 Conclusion

Both the April Tag and WhyCon/WhyCode systems suffer from fundamental issues of orientation ambiguity. April Tag's more intricate structure makes its perception of orientation more robust than that of WhyCon/WhyCode, but it still does not eliminate the ambiguity entirely, and we can see the destructive consequences of this if the orientation is used in other calculations. Still, simple Kalman filtering can mitigate the issue enough that the system can be used for precision landing. Furthermore, the flexible layout of April Tag makes it a better choice for precision landing. The ability to embed concentric markers within each other is critical to maintaining a continuous view of the landing pad from both near and far, and the ability to change the number and layout of data bits allows users to adjust the markers according to their computational constraints.

2.4 Experiments with AirSim

2.5 Small Drone

In an effort to avoid some of the logistical limitations induced by the larger drones, e.g. transportation, material costs in the case of crashes, maintenance, overhead from the creation of component covers and mounts, and the use of large batteries, I have adapted a drone design [3] from Thingiverse.com for the purposes of autonomous landing. The base design is a very simple quadcopter platform with a large area for mounting electronics, a flexible motor mount, a symmetric arm design which allows all arms to be interchangeable with one another, and a simple press-fit system for attaching the arms to the body. After modifications, it now includes a pitch-gimbal mount for a Raspberry Pi camera module that is controlled by a micro servo, mounting holes for various components (including the Raspberry Pi, power distribution board, metal legs, and standoffs for an "upper deck"). All modifications were made with OpenSCAD and are available in the drone's github repository [10] The drone runs on a 3S (11.1V) lithium polymer power system, with a DJI Snail propulsion system.

Figure 2.21: The IR drone in flight with Joshua (left) operating the drone and Baldur (right) operating the gimbal.

2.6 Heavy Lift Drone with Infrared Camera

Drone-based remote sensing is a new and powerful tool in many fields, including geology. To this end, Geologist Christopher Hamilton brought a heavy lift drone to Iceland with the plan of collecting infrared video at active lava sites. The drone is theoretically capable of lifting itself with an extra load of 25 kg. This high weight capacity is necessary to lift multiple large batteries, a large gimbal, and an infrared camera (FLIR). The drone before flight can be seen in Figure 2.22. Along with Baldur, I assembled and tested the drone, making modifications to accomplish the goal of obtaining aerial IR footage of the lava field in Geldingadalur.

The drone uses a WuKong (DJI) industrial flight controller, 6 2-horsepower motors, a 22.2 V, 22000 mAh LiPo battery, and has a diameter of more than a meter. It sits on a gimbal with the mounted IR camera. The gimbal uses a 14.8 V battery to control the orientation of the camera, and receives RC signals from a secondary, dedicated receiver. (One operator controls the drone while the other controls the gimbal.) The camera does not have stringent power requirements, but in this setup the only option for powering it was to provide a 48 V PoE (Power over Ethernet) source. A Raspberry Pi 3 connects to the camera over ethernet and records the streamed video to a file, using a special version of GStreamer developed for the FLIR.

Due to time constraints (about 48 hours from receipt of the gimbal to flying over the lava field), much of the drone has been quickly improvised. Many of these things must be rectified in the near future for further testing. For example, the mounting plate between the gimbal and the drone is completely inadequate (too flexible, not enough points of contact, etc.), and must be replaced with a more solid/purpose-made mounting plate. The 48 V requirement of the camera required an additional 3 4S battery packs wired in series and injected into the ethernet, and this can be replaced with a transformer that can generate 48 V from the same 14 V power source that the gimbal uses. The IR video is not streamed to the operators during flight, so the operators must guess what the camera is seeing, and attempt to obtain good footage.

Still, even with all these disadvantages, the drone flew successfully two times over the lava field at Geldingadalur and obtained aerial IR footage of the lava, as can be seen in Figures 2.21 and 2.23. Additionally, the BBC program documenting the event[4], and the raw IR footage[5] can be found online.

[4]BBC program documenting the drone flight: `https://www.youtube.com/watch?v=6SIgFPhhRPE&t=1190s`
[5]`https://vimeo.com/580302507`

Figure 2.22: The assembled drone before flight.

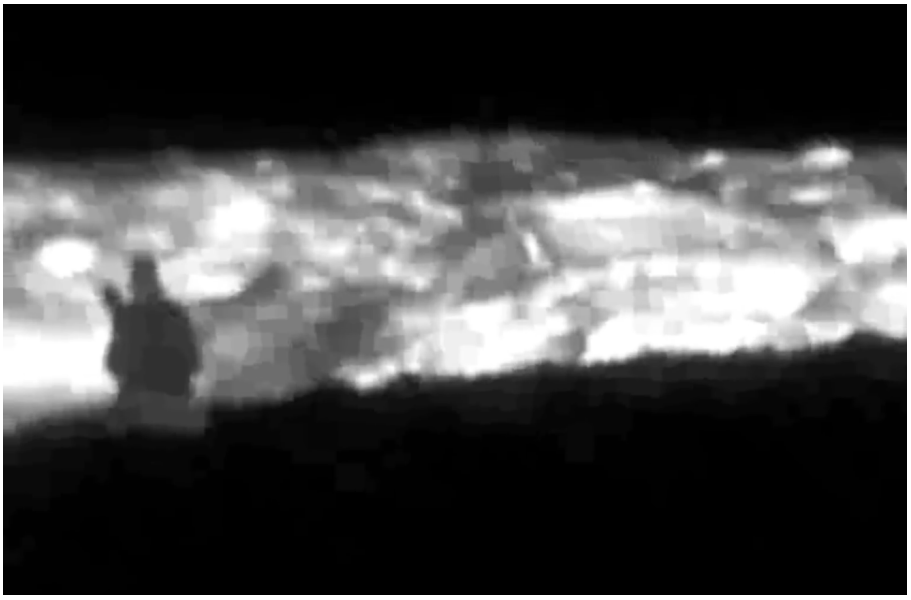


Figure 2.23: An example still picture from the IR footage of the drone.

2.7 Autonomous Landing Proof of Concept

The last part of the initial phase of this research has been to successfully develop a proof of concept of the autonomous landing algorithm on a physical drone, which was also the goal of the first part (in Section 2.1). The crucial difference is that we have used a DJI Spark as the drone platform, which is stable enough to remain nearly still without human intervention, even in a GPS-denied environment. However, even though the proof of concept is successful, its implementation is significantly constrained because of its basis on the DJI Mobile SDK, as explained below. The DJI Mobile SDK is necessary because the DJI Spark has no companion board, but anticipated future drone systems will have a more ideal setup using the DJI Onboard SDK.

2.7.1 System Architecture

The system is outlined in Figure 2.24. All data is transmitted from the Spark to its controller via a wireless connection. The controller interfaces with an Android tablet to expose the drone's data and API. The app must authenticate with DJI using a user-specific API key before it can communicate with the drone, which drives home the proprietary, block-box nature of the system. Importantly, after the app has successfully authenticated once, it can re-authenticate without an internet connection for some period of time, meaning that an internet connection is not required in the field. The Android app then receives data about the condition of the drone, including all flight data (e.g. attitude, velocities, flight modes, etc.), and the video stream from the camera. The app periodically decodes bitmap frames from the video stream, compresses them into black-and-white .webp images, and transmits them over a WiFi connection to a companion board. The companion board then carries detects April Tags in the video frames, determines what action to take according to the control policy, instantiates a command message, and sends that command back to the app. The app then reads the command, creates a virtual stick input via the DJI SDK, updates the GUI to visually show the command, and sends the command to the drone. The drone then carries out the command.

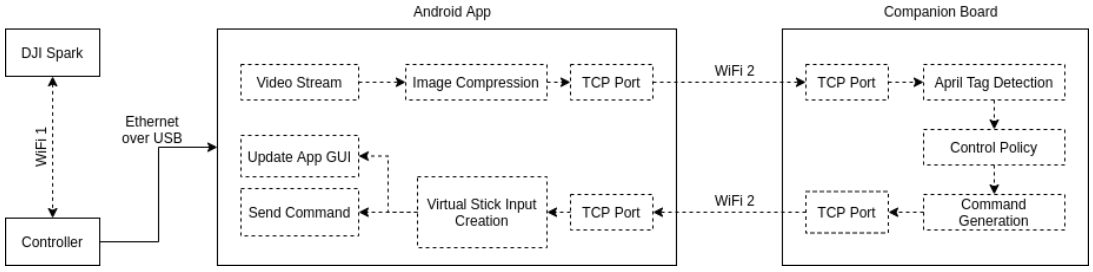


Figure 2.24: Data flow for the Spark autonomous landing system.

The virtual stick inputs simulate input as if it were coming from the controller. Each stick command can be in the interval $[-1, 1]$. The drone can interpret those stick commands as setpoints either for position or velocity. Since we are abstracting from position, we only use the velocity mode. Therefore, a command on the pitch stick of 1 means that the drone should move forward at its maximum velocity.

2.7.2 Results

The system is able to reliably land the drone without human intervention. [6] So far, we have tested it with April Tag 48h12 and April Tag 24h12 landing pads. As anticipated, the April Tag 24h12 tags are subject to somewhat common instances of orientation ambiguity, whereas

[6] Autonomous landing demonstration video: `https://vimeo.com/manage/videos/646612738`

the April Tag 48h12 tags are mostly free of orientation ambiguity. In the demonstration video, this orientation ambiguity appears as jittery control inputs from the companion board, as well as (sometimes) jittery behavior from the drone. Surprisingly - and fortunately - this behavior appears to be non-destructive in this case.

The Android app has 5 sliders to control the drone:

1. Pitch (translated to forward/backward speed)
2. Roll (translated to left/right speed)
3. Yaw (translated to yaw speed, i.e. rotational speed in the Z axis)
4. Throttle (translated to vertical speed)
5. Gimbal Tilt (translated to gimbal tilt speed)

As well as buttons to enable or disable virtual control in the flight controller, to command the drone to takeoff, and to command the drone to land (blindly). The app works just well enough to allow for basic testing of this proof of concept, but has only received minimal development efforts because it is almost pure overhead. Therefore, it crashes often. The proposed research will avoid the Mobile SDK and will therefore avoid the need for a custom app. However, subsequent app development can be the subject of a future project if necessary. Still, the app has worked well enough to allow for many autonomous landings. [7]

2.7.3 Limitations and Implementation Quirks

The Mobile SDK architecture is not well-suited to this application for a number of reasons. First, there is only one wired connection on the tablet, meaning that at least one connection must be wireless - either the connection to the controller, or the connection to the companion board. The tablet is not fast enough to do April Tag image analysis on its own, and there would be significantly increased development efforts required to install April Tag on the tablet alone (so the companion board is required). Moreover, connecting the tablet to the controller via its WiFi interface incurs prohibitive latency as a result of the large amount of processing required. It is also prohibitively slow to connect two clients over the controller's WiFi simultaneously. Second, the frames of the video stream are available in bitmap format, and each frame is prohibitively large/slow to transmit over a wireless connection to the companion board. In order to transmit "video" in "real time," the app converts bitmap frames to black and white, and then compresses them into .webp format with 10% quality. This allows a framerate of about 6 Hz, with a delay in each frame of about 0.25 seconds, to be delivered to the companion board. Amazingly, this is enough to land the drone with reasonable accuracy. Third, the Spark requires a blind commitment in the last stage of landing - where the flight controller forces the gimbal to point directly forward, apparently to protect it from the ground since it has very close clearance. So, although the drone responds to commands during this phase of landing, the companion board cannot see the landing pad and therefore cannot generate commands for last minute correction.

[7] The autonomous landing demonstration from the point of view of the app: `https://vimeo.com/manage/videos/646620944`

Chapter 3

Research Plan

Here we outline the plan for using the knowledge and physical drone platforms generated until now to create a method for landing in unstructured environments. The critical difference is that we will avoid making major assumptions about the landing area, e.g. that it uses a specific type of fiducial marker. The goal is to create a lightweight method for analyzing terrain in real time, identifying safe landing sites, and navigating to those landing sites without using some exterior information such as GPS.

3.1 Data Set Generation

We will create a synthetic data set of LIDAR point clouds and RGBD images, and corresponding segmentation masks in AirSim, and we will collect similar data from the real world on a smaller scale (without segmentation masks). The labels for the data set will be "safety masks" which label each pixel The synthetic data will be labeled automatically using slow, detailed terrain analysis methods, and leveraging the segmentation masks in order to have a notion of *material*. This is to say that, even though calm water may be flat (and therefore safe, according to its topology), it is obviously not a safe landing site and can be ruled out just by being water. Conversely, an area of grass that may appear rough is not necessarily a dangerous landing site. By using the segmentation masks in creating the labels, we hope to discriminate between potential landing sites on another level than simply topology. However, topology will be the principle way of determining the safety of a landing site, and the most important aspects will be to minimal roughness, minimal slant, and maximal size. So the ideal landing site will be a large, smooth, level area having the appearance of a "safe" material.

We will use a consistent pipeline for processing and labeling the images, which will be similar to traditional image analysis. First, we will rectify the depth images before processing them, to remove inevitable sensor distortion. In reality, the distortion will not be entirely eliminated, but will be mitigated. We will record the orientation of the sensor when the image is collected, and then transform the depth image so that it is oriented vertically down. We will process the rectified image to generate at least two safety masks - one for topological safety and the other for material safety. Then we will combine these intermediate safety masks to create an overall mask, which will serve as the label. The exact sizes of the training and testing data sets have yet to be determined, and as such we will rather focus on creating a script to automatically generate data sets of specified sizes. The data sets will be taken from multiple simulated environments to try to give an unbiased sense of what possible good landing sites look like.

The real world data will serve as a final testing set, which will give a notion of how well the synthetic data correspond to the real world. We will the label the real world data in a similar way to the synthetic data, with the exception that we will not have corresponding segmentation masks. Additionally, it is too time-consuming to create both the labels and segmentation masks by hand. As a result, we will have to take special care in evaluating the performance of the

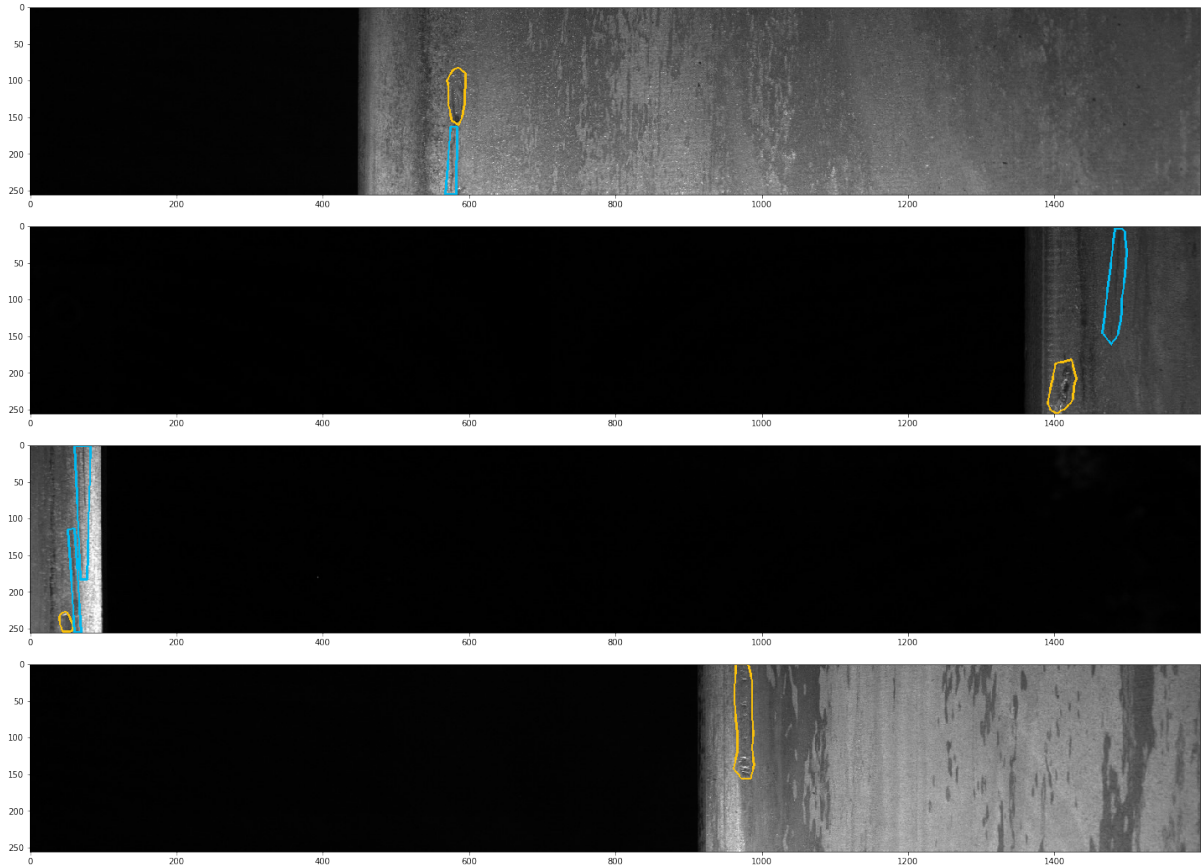


Figure 3.1: Samples of typical steel defect images with their defect masks outlined.

terrain classifiers on the real world data set. It is likely that we will not be able to simply consider how well the predicted mask matches the label. We anticipate that the terrain classifiers may have to be evaluated on real world data purely by the landing locations they select, in order to be feasible in terms of time and logistics. In this case, a human who is familiar with the surveyed area can mark landing locations as feasible or infeasible by hand.

3.2 Terrain Classifier Creation

We will train, test, and evaluate multiple neural networks in their ability to predict the "correct" safety mask of the landing sites in the synthetic data sets. We will focus first on using U-nets to map from the input LIDAR or RGBD image to the output mask, as they have been proven to be good classifiers in many different scenarios, from terrain analysis, to tissue analysis, to road identification, etc. We anticipate that the main challenge in this domain will not be to train the networks, but rather to keep them small enough so that they can execute quickly on embedded hardware. Nevertheless, we will start by focusing strictly on maximizing the ability of the networks to accurately predict the safety masks.

As an example of how this will work, we take an example from my project in RU's Deep Learning course, wherein my group member and I used a Residual U-net to classify defects in manufactured steel. More than 17,000 images were labeled by hand, according to which of 4 defects they contained. Figure 3.1 shows an example of the input data from this project, where the yellow and blue outlines show the locations of different types of defect areas.

This particular project was very time-limited, so the network trained for only 4 epochs, but was able to achieve about 50% prediction accuracy in that time, and its output for an example image can be seen in Figure 3.2. This project also gave some insight into accuracy metrics

that can help bias the output for/against false positives/negatives. For example, in the case of the steel defect recognition, we used a Tversky metric to reduce the amount of false negative defect classifications because a second step of the process is for a human to manually examine the defect-classified steel before discarding it. In this context it is better to make the network more "cautious," in the hopes that alerting the human to some false defects will mean that the network never incorrectly approves defected steel for sale. Conversely, in the case of autonomous drones, we would like the networks to be conservative in their identification of safe landing sites, so that they never direct the drone to land in unsafe landing sites, even though they may reject some landing sites that are actually safe. This is purely to give an intuition of the form of the problem we will use the classifiers to solve.

3.3 Testing in Simulation

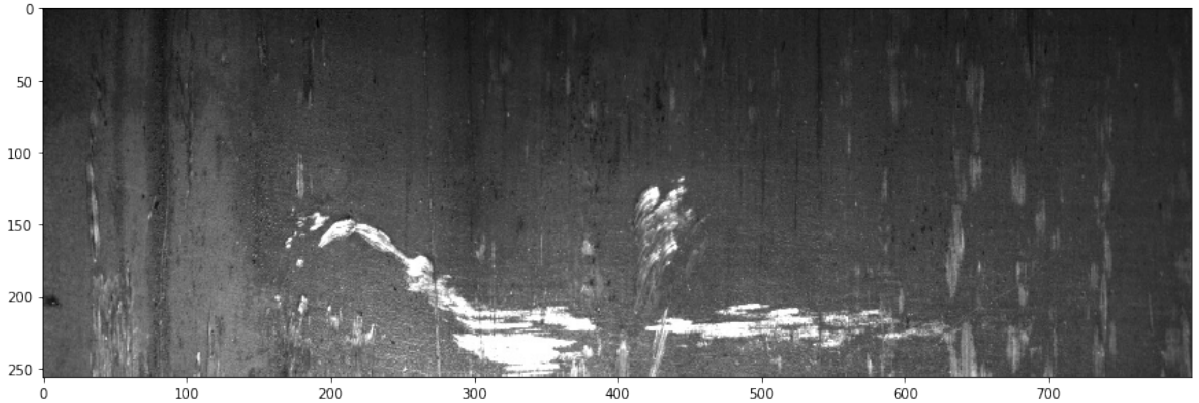
As the terrain classifiers will output a mask, we will have to post-process the data to make it compatible with an autonomous drone system. We will therefore create an output wrapper to find contiguous regions of high predicted safety (i.e. estimated safety above some experimentally-determined threshold), and to track these regions over time. We will then select a region for landing, and create commands to direct the drone towards that region in a similar manner to the method outlined in Section 2.7. We will make no explicit estimate of the region's position relative to the drone in terms of real world distance, as this is unnecessary and requires more sophisticated wrappers which take into account the distortion parameters of the camera in order to associate pixels with physical points (as in the case of fiducial markers). It is simpler and more generalizable to different platforms to choose a safe landing region, calculate its centroid, normalize its coordinates to the interval $[-1, 1]$ based on the pixel dimensions of the camera/mask, and then convert these values to Virtual Stick inputs (in the case of a DJI flight controller), or to velocity setpoints (in the case of ArduPilot or PX4 in simulation).

After creating the output wrapper, we will move to testing the networks in simulated environments in AirSim. We will use multiple environments - both those from which the training data originates, and also unfamiliar environments - to get an idea of the generalizability of the networks. These networks will be evaluated on the number of successful landings they execute in a number of different scenarios.

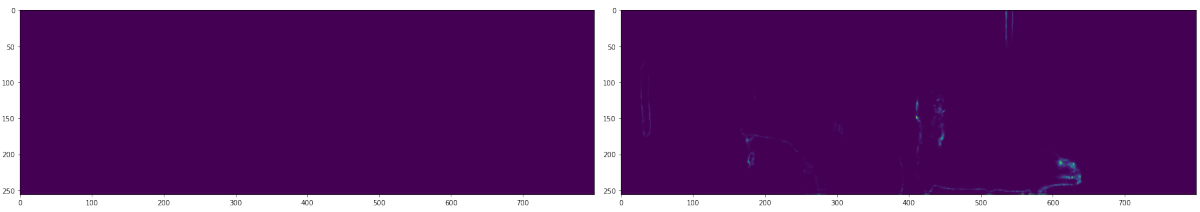
3.4 Testing on Physical Hardware

The first phase of this testing is to port the high-performing networks from testing in simulation to the real hardware. So far, we are targeting our existing Google Coral Dev Board, which has an embedded TPU, and an NVIDIA Jetson Nano, which has an embedded GPU. We will implement the pre-/post-processing wrappers for rectification and command generation in the same way as in the testing in simulation, and instantiate the trained networks on the embedded hardware. We can then pipe data from the synthetic or real-world data sets into the physical system to collect empirical results for the framerate achieved and power/computational requirements in making predictions using the networks. Piping data to the system will allow us to test the hardware/software in isolation, such that the effect of installing it onto our drone systems will be predictable. This is an area where we may have to carefully re-consider purchasing new embedded computational hardware, if none of the networks run fast enough, or if they all take too much power. Aside from the Google Coral and Jetson Nano, we will also consider some FPGA in the future.

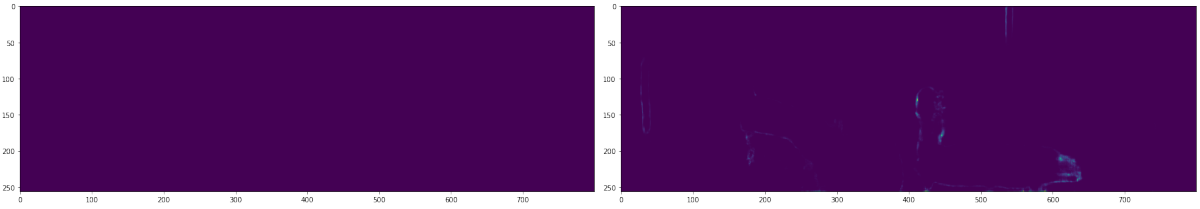
If any of the combinations of networks and hardware work with enough accuracy, at a high enough framerate, and by consuming little enough power, we will install this hardware onto a physical drone and test the system in a real autonomous landing scenario.



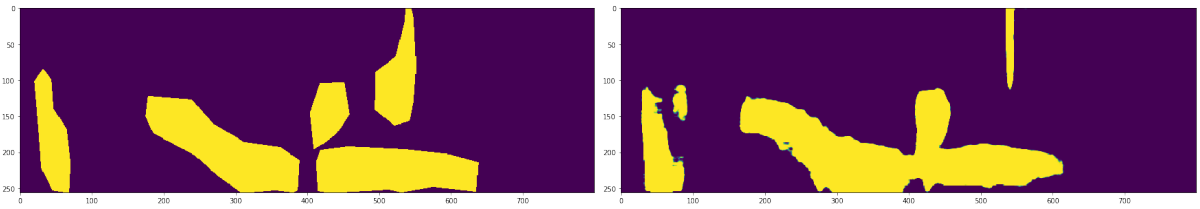
(a) Defect 1 Mask



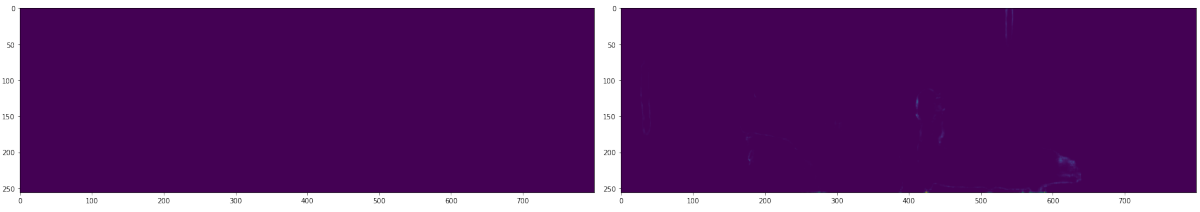
(b) Defect 1 Prediction



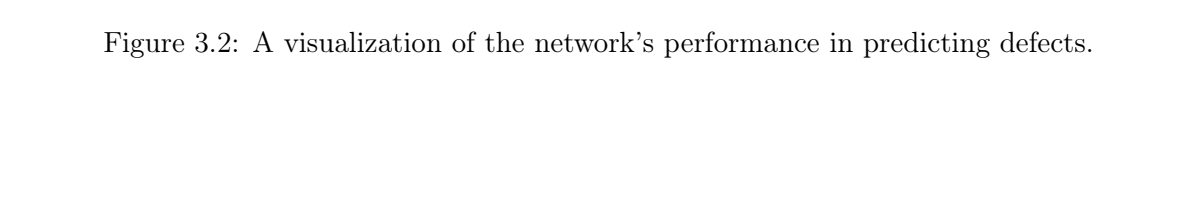
(c) Defect 2 Mask



(d) Defect 2 Prediction



(e) Defect 3 Mask



(f) Defect 3 Prediction

(g) Defect 4 Mask

(h) Defect 4 Prediction

Figure 3.2: A visualization of the network's performance in predicting defects.

3.5 Choice of Flight Software/Hardware and Drone Upgrades

There are two general stages of testing in the context of drone control algorithms: testing in simulation and testing in the real world. Testing in simulation is an intuitive first step in testing, as it allows to test control algorithms without added logistical, time, or financial cost. It also provides an idealized world, free of complicating factors such as wind, battery life, noisy sensors, etc., isolating the control algorithm as the only testing variable. However, simulation does not prove that a system works in real life. Therefore, after control algorithms have been proven to work in theory (simulation), they must prove themselves again in the real world, with all the implied complexities.

ArduPilot and PX4 are key software packages that can be used for testing in simulation. Previous work in this project and in others have proven worthiness of PX4 and Gazebo in terms of simulated autonomous control. They have complete integration with both simulation environments that are of interest in the upcoming research: Gazebo and AirSim. Both softwares have APIs in Python, C++, and ROS. This research requires position control - and particularly body offset position control. This means that the drone must follow a position setpoint that is given in the coordinate frame that is centered on the drone's body and oriented with the same heading as the drone. The position setpoint is given in terms of ENU (East, North, Up) where East refers to the drone's right, North refers to the drone's front, and Up refers to the drone's top. The setpoint is given in meters, so a position setpoint of (E, N, U) = (-2, 1, -1) means that the drone should move 2 meters to the right, 1 meter forward, and 1 meter down. In ArduPilot, this type of body offset position setpoint is technically supported, but the firmware translates the position setpoint directly to a GPS coordinate and relies on GPS for navigating to that place. PX4 works also offers an API for generating body offset position setpoints. in some cases the position setpoint must be translated into the NED (North, East, Down) frame, by simply switching the North and East components, and negating the Up component to form the Down component. In PX4, precision land setpoints are only supported in the coordinate frame centered on the drone's home (takeoff) position, and oriented due north, and the coordinate frame is referred to as `LOCAL_NED`. For generating a body offset position setpoint, one therefore must add the current position and heading to the target position and heading, but otherwise it works in a similar way to the position setpoints. Both ArduPilot and PX4 also offers APIs for controlling a gimbal to aim cameras. In ArduPilot this can be done with an RC override command, where RC input can be programmatically generated and then forwarded to the output channels that control a gimbal, as if the software is controlling the gimbal via switches/sliders on the controller. PX4 provides a more sophisticated `vmount` interface that controls the gimbal with possible (user-configurable) consideration of the orientation of the drone.

DJI flight control software (and corresponding hardware) are more suited to real world testing because of its greater stability and reliability. ArduPilot and PX4 are open source, so while they lack the profit motive and black-box nature of DJI, they also unfortunately lack its robustness in terms of flight capability. The most crucial result of this is the inability of such systems to enter an autonomous flight mode when GPS precision is low (i.e. GDOP is high) which is the case nearly always in Iceland. Potential solutions include using RTK systems, but these require base stations, increasing the hardware complexity of the required system. They also require that the base station be enabled for some time before increasing GPS precision. By contrast, many DJI drones (those with external distance sensors) are stable in completely GPS-denied environments, and allow for autonomous control even in those circumstances. Additionally, DJI flight controllers can function in GPS mode with a higher GDOP, such that they allow autonomous flight with lower GPS quality, albeit with minor drift/inaccuracy. In this case, drones with ArduPilot or PX4 can even be subject to fly-offs (complete loss of control).

In terms of their APIs that enable autonomous control, ArduPilot and PX4 offer more open interfaces with implementations in C++, Python, and ROS, and DJI offers a more closed

solution with fewer options. However, the more open interfaces of ArduPilot and PX4 (via e.g. MAVLink/PyMAVLink) are poorly documented and have occasional quirky/unexpected behavior that requires extra overhead efforts to address. For example, certain message/command attributes are sometimes ignored with no explanation in the documentation, and explanations can only sometimes be found in user forums (and may require independent verification). In some cases of aiming a gimbal at targets in PX4, there has even been destructive behavior where a change in the commanded gimbal tilt resulted in an unexpected and uncontrollable change in gimbal pan, with no explanation. This implies more development time in the best case, and crashes in the worst case. While DJI's API is more closed, it is better documented and does not exhibit such unexpected behavior in our experiments. Autonomous control of the drone and gimbal is available primarily through virtual stick inputs, wherein the companion board requests virtual control from the flight controller, and then can control the drone and gimbal by simulating user control inputs, which can correspond either to target velocities or positions. This allows for more intuitive autonomous control with high level checks, such as maintenance of obstacle avoidance. In terms of disadvantages, drones without a companion board *must* interface via the Mobile SDK through either an iOS or Android app, which implies many constraints on methods of connection and transmission of data. For example, in the case of the autonomously landing DJI Spark, video from the drone needed to be compressed on the *tablet* in an Android app, before wireless transmission to the companion board. However, in the case of custom drones which do include companion boards, the connection infrastructure and programming setup are more flexible and less constraining.

Overall, ArduPilot and PX4 are perfect options in the case of simulation, since they are free and provide reliable control in idealized simulation environments, but they do not perform as well as similar DJI products in the non-idealized environment of Iceland. However, control methods developed in simulation with ArduPilot or PX4 can be ported with relative ease to DJI flight controllers in the real world. Therefore, we plan to use ArduPilot and PX4 for the first phase of testing (in simulation), and DJI products for real world testing.

We will change at least one of the hexacopter systems developed in the early part of this research to use either a DJI N3 or A3. These are the later versions of the Naza V2 and other DJI flight controllers which have been shown to provide stable GPS-based control in Iceland. The main difference between them and the current DJI flight controllers that we have is that the new ones support the Onboard SDK, and therefore allow customized programmatic control, in a similar way to the DJI Spark. However, we have enough room on our custom drone systems to embed significantly more hardware than the Spark has. We will carry out the real world tests and data collection with the upgraded drone(s).

3.6 Risk Analysis

There is little risk in the synthetic data set generation, as we have already generated similar (albeit more basic) data sets in the past, and running AirSim does not pose any logistical risk or hazard. There is slight risk in collecting the real world data sets, from the ever-present risk of damage to equipment from crashes, to the difficulty in collecting data with high enough resolution and operating within the motion/distance constraints of the sensors. To mitigate this, we will run several missions to collect real world data over a long period of time, so that our method of collecting and tagging the data can be refined.

We anticipate that the terrain classifier creation should pose little risk, in that it should be feasible to create classifiers that give the desired results on the training data. This is justified by the fact that the types of networks we will be training have been shown to perform well in a variety of similar classification tasks. However, there is the risk that the bulk of the training data, being synthetic, will not adequately represent the real world. This would mean that the terrain classifiers can perform well in simulation but not in the real world. We cannot mitigate

this risk by only using real world data because of the time and effort required to label the data. We can attempt to mitigate it by generating synthetic data from simulated environments that are similar to the environment where we will be testing - i.e. by limiting the scope of the data to environments similar to Iceland, instead of using e.g. tropical environments, or contrived environments that do not represent the real world at all. Additionally, we can use human-made post-processing wrappers to handle undesired behavior - e.g. selecting conservatively from the safe landing regions that the network identifies.

Testing in simulation will have low risk, as AirSim already has plugins for both ArduPilot and PX4, as well as drone models that can fly autonomously in the simulation. This step will mostly consist of integrating existing components with the terrain classifiers and pre-/post-processing wrappers.

The first part of testing on physical platforms (in a lab setting), poses low risk. Creating the infrastructure to transfer data to an external board is not difficult, and measuring the framerate and power consumption is also not difficult. There is a risk in the case that none of the terrain classifiers can run fast enough, or with sufficiently little power consumption, on our particular hardware platforms. In this case we will need to consider getting different platforms, but at the very least we will have collected relevant and helpful data on the computational requirements of the terrain classifiers.

Testing the system for real world landings is the most risky part of the research, not just in terms of logistical considerations and potential of crashes, but also in that it depends on the high performance of the systems developed in the previous steps, and also on their integration together in a single system. The biggest risk that we anticipate is that the training data does not correspond well enough to the real world for the system to properly identify safe landing sites. Although we will try several ways to mitigate this risk, it cannot be entirely eliminated. However, in the case that the synthetic data do not transfer to the real world, we have a two-pronged basis on which to move forward. First, we will have already conducted research on the types of network architectures, sensor data, and pre-/post-processing wrappers that perform well in simulation, and we will still be able to use these. Second, we will have physical drone platforms that we can use to collect more real-world data, and we can label this data according to simple, slow, human-made methods, without considering segmentation masks. Then, we can re-train the networks that we have on purely real-world data.

Bibliography

[1] Ardupilot.org.

[2] Edited April Tag Repository. (accessed: 2021.3.25).

[3] Thingiverse (arks007). Raspberry pi drone (niacam: Txsef 2016-2017). (accessed: 2021.9.24).

[4] Dronecode. MAVLink Micro Air Vehicle Communication Protocol. (accessed: 2021.5.19).

[5] Emlid. Navio2. (accessed: 2020.6.5).

[6] M. Krogius, A. Haggenmiller, and E. Olson. Flexible Layouts for Fiducial Tags. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1898–1903, 2019.

[7] Lorenz Meier. Pixhawk. (accessed: 2020.6.5).

[8] Matias Nitsche and Tomáš Krajník. Original whycon ros github repository. (accessed: 2021.9.30).

[9] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407, 2011.

[10] Joshua Springer. Raspberry pi drone. (accessed: 2021.9.27).

[11] Joshua Springer. Whycon ros github repository. (accessed: 2021.9.30).

[12] Joshua Springer. Autonomous Landing of a Multicopter Using Computer Vision. Master's thesis, Reykjavík University, 2020.

[13] Jiri Ulrich. Whycon ros github repository. (accessed: 2021.9.30).

[14] John Wang and Edwin Olson. Apriltag 2: Efficient and robust fiducial detection. pages 4193–4198, 10 2016.