Dynamic Baseline Binocular Stereo Using Multirotor UAVs

Joseph Bolling, Ankush Gola, '15

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

Final Report

May 1, 2015

Professor Andrew A. Houck
Professor Peter J. Ramadge
ELE 497/498
69 pages

This thesis represents my own work in accordance with University regulations.

Abstract

We design and implement a 3D film-making system in which the left and right perspectives are shot from independent unmanned aerial vehicles (UAVs), thereby producing a stereo vision system with a dynamically variable baseline and elevated perspective. This system allows for novel visual effects due to the amplified depth perception achieved from the wide baseline. Furthermore "shrinking" and "growing" effects are achievable by dynamically varying the baseline. Our designs are motivated by research into human stereopsis and existing work in computer vision. We discuss our UAV platform design, techniques for UAV control and synchronization, current progress in enhancing vehicle state estimation through vision-based UAV localization, and our video stabilization pipeline. We achieve video that is comfortably viewable and lay the groundwork for further system enhancements.

Acknowledgements

To see through the eyes of a giant you first have to stand on the shoulders of giants. Thanks to everyone I interacted with while completing this thesis, your smiles and support kept me on track even at my most frantic. Specifically: Thanks to Ankush for fixing my python code, staying up all night to get one last plot, never hesitating to try the next moonshot idea we come up with, and for constantly laughing with me. Its been a great four years.

Thanks to Professor Houck, for his willingness to take time to advise us even at his busiest, his never-ending insights, and for challenging us to do better without ever putting us down. Without Professor Houcks faith in us, we would never have attempted such a project.

Thanks to Professor Ramadge for taking an interest in our project without ever being required to. Ankush and I started this project with a vague idea of what image processing was, and only managed to solidify it into a working knowledge of the field with the help of Professor Ramadge.

Thanks to Professors Verma, Kornhauser, and Stengel for pushing me academically and extra-curricularly in ways I didnt know I could be pushed. They have inspired my interests and taught me lessons that will stay with me forever.

Thanks to everyone at PAVE- Gabe, Dan, Horia, and Marcus for learning what a Kalman Filter was with me. Derrick for his ability to solve literally any problem. Kevin, Artur, John, Travis, Chris, and Stephen for taking up the mantle and making this club into more than I ever hoped it could be. Chip for being the mentor we needed. To the Outdoor Action family thank you for challenging me to be more than just an engineer. Rick, Caroline, Kevin, LTs, TSTs, WFRs, and LNTMEs, and all the other abbreviations, thanks for helping me to learn and have fun doing it. To Penna and the Princeton Chapel Choir, thank you for giving me a home the last two years, and then for moving that home to Southern Spain for a week. Its been a blast. To everyone at Terrace Food=Love.

Thanks to my freshman hallway Michael, Caleb, Alvina, Danielle, Josh, Lexi, and the rest - for all the inside jokes. Thanks to Todd and Dave for making me realize how incredibly cool engineering is.

Above all thank you to my family. Amy, Sarah, Di, James, Mom, and Dad. There are no words that can describe you, but here are a few anyways. Thanks for being parents, siblings, role models, friends, and mentors all at once.

Finally thank you to Abby for listening, walking, laughing, crying, cuddling, and learning with me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The 3D film-making industry is rapidly expanding, but has seen no novel effects that take advantage of emerging 3D viewing media. Furthermore, the commercial Unmanned Aerial Vehicle (UAV) industry is poised for rapid growth, especially in the aerial cinematography sector.

We propose a relatively low-cost, wide and dynamic-baseline stereo vision system implemented using multirotor UAVs. Our goal is to introduce new, depth-perception enhancing effects to the viewer (i.e. to give viewers an increased sense of scale). The purpose of implementing the system with UAVs is to enhance the effect with dynamic views and aerial perspective.

1.2 System Overview

In our system, two UAVs autonomously fly in a preprogrammed formation while individually taking videos of a scene. The UAVs communicate to a ground control station, which monitors vehicle health. The raw video is then processed and stabilized using data about each vehicle's state (position and attitude) recorded during flight. Enhanced state estimation can achieved by incorporating computer vision-based UAV localization. The processed videos are then ready to be viewed using a 3D display. Figure 1.1 shows an animation of our system in flight. Here, two UAVs fly in a formation while simultaneously recording video.

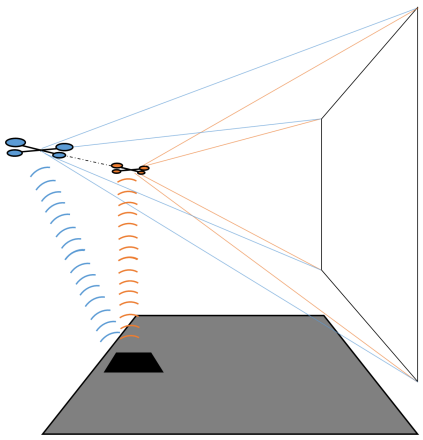


Figure 1.1: Perspective diagram of a wide-baseline UAV flight.

Chapter 2

Platform

Our platform is composed of two multirotor UAVs and filming cameras, an auxiliary camera and embedded processor to allow vision-based localization, a 3D viewing device, and base station for vehicle control. Careful thought went into picking the our system components.

2.1 3DR X8 Multirotor UAVs

In selecting UAVs, we identified a variety of criteria on which to compare systems. The platform selected needed to be capable of outdoor flight with sufficient stability for high-resolution video capture. It had to be able to lift more than its ship configuration in weight, to accommodate the hardware additions described in section 4.2. Mechanical robustness was also a consideration, as hard landings were likely to occur during flight trials. Foremost on our list of criteria was ease of modification, so as to minimize overhead in controlling and interacting with the system.

After considering a variety of UAVs from consumer toys to high-grade research platforms, we selected the 3D Robotics X8 (shown in Figure 2.1). The X8 has 8 rotors in an X configuration, stacked vertically in pairs of two. This provides similar flight dynamics and frame weight to a traditional quadrotor, but with the redundancy, stability, and lift capacity of 8 rotors. The X8 is equipped with a PixHawk flight controller running APMCopter firmware. The Pixhawk accepts pulse-position modulated (PPM) radio commands. It also supports telemetry data output and control input over a UART connection using the popular MAVLink protocol. On the X8, this is connected to a telemetry radio, allowing a ground station computer to

communicate directly with the Pixhawk during flight. In addition to sensor measurements delivered over the telemetry link at rates of up to 4Hz, high-frequency data for sensor measurements and internal data structures can be logged on-board and downloaded after a flight. This is discussed further in Section 5.2. Both the hardware and firmware of the PixHawk are open source, allowing us to gather a low-level understanding to the flight stabilization and localization processes when necessary.

Figure 2.1: Unmodified X8 UAV designed by 3D Robotics.

The PixHawk acts as a safety measure for the system, since it can still land the vehicle if radio control is lost or a battery reaches a dangerously low voltage. The PixHawk has several different modes of operation, which support different degrees of autonomous and user control. This allows for a safety override functionality. The ground station places the X8 in `GUIDED` mode before sending commands. `GUIDED` mode can be interrupted from the radio controller and the UAV can be placed in a different mode to stop the execution of autonomous commands in the case of an emergency.

2.1.1 Sensor Configuration

In its stock configuration, the X8 is equipped with an advanced sensor set sufficient for basic localization and airborne stabilization. The Pixhawk flight control board has an integrated Inertial Measurement Unit consisting of an L3GD20 3-axis 16-bit gyroscope, an LSM303D 3-axis 14-bit magnetometer, and an MPU 6000 3-axis gyroscope. The Pixhawk also has an MS5611 barometer used for altitude measurements [1]. Mounted on an elevated mast (visible near the top of Figure 2.1) are a uBlox GPS and an additional HMC5883L compass unit [2]. The sensor data from these

devices are fused in a real-time Extended Kalman Filter running on the Pixhawk as part of the X8's embedded APMCopter firmware.

2.2 GoPro Hero 3 White

Each of our X8 UAVs is equipped with a GoPro Hero 3 White camera. The GoPro makes an excellent camera for drone photography because of its small size and weight. The Hero 3 provides 1080p video at 30fps, and is equipped with a wide-angle lens. This lens facilitates dynamic cropping and repositioning of the displayed portion of a frame, which will be useful in our digital stabilization and rectification algorithms, discussed in Section 5.3.

The GoPro's final strength lies in the variety of robust electronically-stabilized mounts that have already been developed. We selected the Tarot T-2D brushless gimbal as a mount for our systems. The T-2D provides mechanical roll and pitch stabilization to the GoPro using an onboard inertial measurement unit and can be powered by the X8's main battery. The T-2D significantly attenuates two of the dominant noise sources in the X8, since the UAV rolls and pitches to strafe forward, backward, left and right.

2.3 Logitech C920

To augment the X8's on-board sensor suite, we use a camera for relative position measurement, as described in Chapter 4. For this camera, we selected the Logitech C920 webcam. The C920 is capable of capturing full 1080p video with an on-board H264 encoder, providing detailed imagery even at long distances. It also has auto-focus capabilities suitable for outdoor use. Perhaps most useful is the extensive body of work available online detailing how to use the C920 in robotics projects, including software libraries, example code, and hardware modification guides. Specifically, we remove the weighted base as described in [3], considerably reducing the flight weight of the camera system. Also useful are Darling and Molloy's notes on acquiring fast video with the C920 on the Beaglebone Black [4], [5].

2.4 Beaglebone Black

To run the C920 and collect images when it is mounted on the X8 we us a Beaglebone Black embedded Linux computer running Debian, shown in Figure 2.2. The Beagle-Bone is equipped with an ARM Cortex A8 processor at 1GHz, 512 MB of RAM, and 4GB of on-board flash storage. We expand this with a 32 GB SD card to provide room to store enough video for several flights. The Beaglebone supports the OpenCV and Video4Linux libraries which we use to capture video. In addition to a USB port, the Beaglebone Black has 65 GPIO pins total, with hardware support for UART, SPI, I2C, and PWM communication.

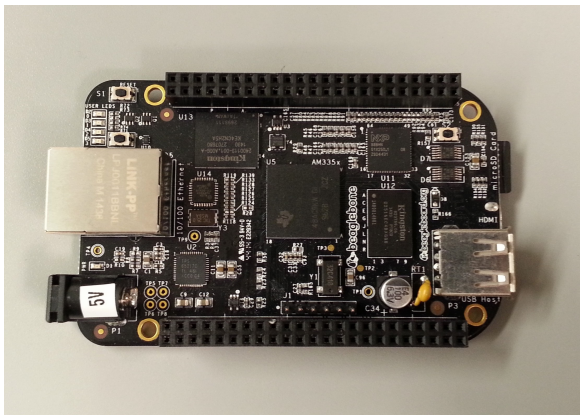


Figure 2.2: Beaglebone Black embedded Linux computer.

2.5 Oculus Rift

While any 3D display technology should be sufficient for viewing our footage, we selected the Oculus Rift Development Kit 2 virtual reality headset for several reasons. The Rift provides an immersive viewing experience even at low resolutions. It has extensive development support including an SDK and documented example projects and, at $350, the Rift is considerably cheaper than multi-viewer 3D media, such as 3D TVs and projectors.

The Oculus Rift consists of two screens suspended in front of a viewer's eyes, each displaying image data for the left or right perspective. A converging lens is placed in front of each screen to allow the viewer to comfortably focus on the screen even though it is located very close to his or her face. Two different sets of lenses are provided, which can be swapped out to provide a lower power for viewers who are already nearsighted. The Rift features a resolution of 960x1080 per eye.

2.6 Ground Control Station

The two X8 UAVs are controlled from a ground control station running code to keep their movements synchronized, as described in Chapter 3. The ground control station is implemented on a 2013 Macbook Air (fittingly) with a 1.3 GHz Intel Ivy Bridge Core i5 processor and 4 GB of RAM.

Chapter 3

UAV Control

As mentioned in Section 2.1, the X8 multirotor UAV comes equipped with a sensor suite and Pixhawk flight controller for basic control and stabilization. However, a higher-level control scheme is needed to fly the two X8 vehicles in formations necessary to capture wide and dynamic baseline stereo footage. In our approach, we set up a ground control station (GCS) that is able to receive messages from both vehicles and send commands accordingly.

3.1 Overview

The GCS runs three processes: one client process for per vehicle that transmits and receives messages directly from the corresponding UAV, as well as a server process that monitors and synchronizes the clients. We implemented the finite-state machine (FSM) controller shown in Figure 3.1. Our server program takes in a starting and ending waypoint for each vehicle. Each waypoint contains a latitude, longitude, and altitude (in meters). Each vehicle takes turns arming (enabling motors), taking off, and flying to the first programmed waypoint. This is done to prevent any possible collisions. After the two vehicles have reached their initial waypoints, they simultaneously fly, in a straight line, to their goal waypoint. During formation flight, the server process synchronizes the vehicles as much as possible to prevent one vehicle lagging behind the other. Figure 3.2 shows three of the many possible formations that can be executed with our setup.

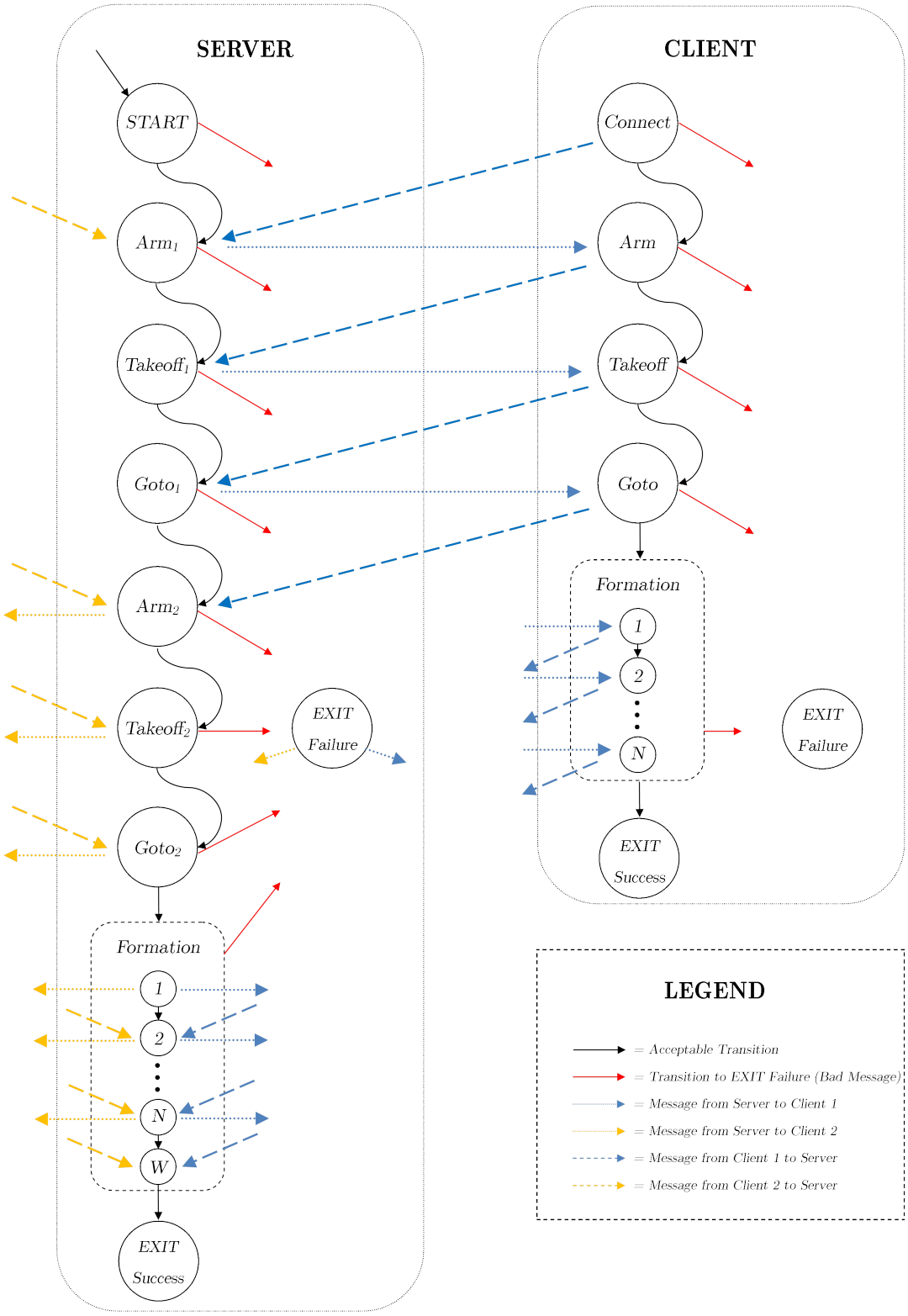


Figure 3.1: FSM-style controller running on server and client processes (second client not shown).

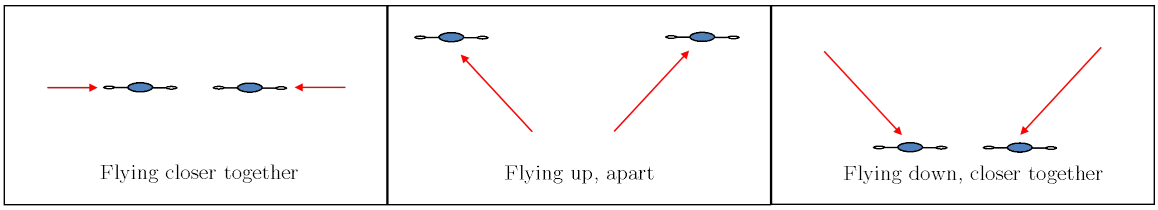


Figure 3.2: Three possible flight formations for dynamic baseline effects.

3.1.1 Server States

The goal of the server is to monitor the messages from each client process, handle errors, and close the connections when necessary. Closing connections is especially important to prevent undefined behavior. The server sends messages to each client to enable the clients' next actions.

Start

In this state, the server listens for the two clients to accept the socket connection (this will be discussed more in Section 3.2.3). Additionally, there is a timeout for the clients to accept the connections; after the time allotted, the server transitions to the `EXIT_FAILURE` state and closes any open connections. Once connections are established, the server transitions to the first `Arm` state.

Arm First

The server waits for a dummy message from each client to assess connection health. This is done as a sanity check. If the messages are corrupted or do not reach in a timely manner, the server transitions to the `EXIT_FAILURE` state and closes any open connections. After the messages have been verified, the server sends an "okay to arm" message to the first client and transitions to the first `Takeoff` state.

Takeoff First

The server listens for an "armed" message from the first client. If the message indicates that something went wrong during vehicle arming, the server transitions to the `EXIT_FAILURE` state and closes any open connections. If the message indicates successful vehicle arming, the vehicle sends an "okay to takeoff" message encoded with a target altitude to the first client. The server also transitions to the first `Goto` state.

Goto First

The server listens for a "taken off" message from the first client. If the message indicates any sort of error, the server transitions to the `EXIT_FAILURE` state and closes any open connections. If the message indicates successful vehicle takeoff, the server sends an initial "Goto" message encoded with a GPS waypoint to the first client. This waypoint is the starting location for the formation the UAV will be flying. The server also transitions to the second `Arm` state, the state that handles arming the second UAV.

Arm Second, Takeoff Second, Goto Second

The explanation for these states is nearly identical to the previous three states, except that the server is interacting with the second client.

Formation

The goal of the set of states labeled `Formation` is to synchronize the two vehicles as much as possible while they are flying their programmed formation. The formation path for each copter is broken down into several intermediate waypoints. The server waits until both vehicles have reached each intermediate step before allowing the vehicles to move on. If any intermediate message from the vehicles indicates failure to arrive at a waypoint, the server transitions to the `EXIT_FAILURE` state and closes any open connections.

Exit Success

The server transitions to an `EXIT_SUCCESS` state once it receives messages from both vehicles indicating that they have received their goal waypoint. The server closes the open connections.

3.1.2 Client States

One client process is launched per vehicle. Each client program interfaces directly with a vehicle through DroneKit subroutines across a telemetry link (see Section 3.2.2). The goal of the client is to assess the health of the messages coming from the vehicle over the link, and relay these messages to the server process. Examples

of error messages that would force a transition to `EXIT_FAILURE` include messages indicating low battery, sensor glitches, and timeouts.

Connect

The client attempts to connect to the server and sends a dummy message upon connecting. If the client detects that the server is not running, the client transitions to the `EXIT_FAILURE` state.

Arm

The client attempts to arm the vehicle tied to the client process. After receiving the appropriate message from the server, the client sends an "arm" signal across telemetry and switches the vehicle to `GUIDED` mode (the mode that allows for autonomous control). The client then waits until it receives an acknowledgment from the vehicle indicating successful arming and mode switching. If an error message is received from the vehicle, or if the allotted time has passed (indicated by a timeout variable), the client transitions to the `EXIT_FAILURE` state and sends an error message to the server.

Takeoff

The client attempts to fly the vehicle directly upwards to the desired altitude. After receiving the appropriate message from the server, the client sends a "takeoff" signal encoded with a target altitude, obtained from the server, to the vehicle. The client then waits for acknowledgement from the vehicle indicating that it is within a certain range of the desired altitude. If an error message is received from the vehicle, or if the allotted time to reach the desired altitude has passed, the client transitions to the `EXIT_FAILURE` state and sends an error message to the server.

Goto

The client attempts to fly the vehicle to the first programmed waypoint. After receiving an appropriate message from the server, the client process sends a "goto" signal encoded with the initial waypoint to the vehicle. The client process then waits for an acknowledgement from the vehicle indicating that it is within a certain radius of the desired location. Once again, if an error message is received from the vehicle or if the allotted time to reach the waypoint has passed, the client transitions to the `EXIT_FAILURE` state and sends an error message to the server.

Formation

In the set of states labeled `Formation`, the client process flies the vehicle along the specified path. As mentioned in Section 3.1.1, the formation path for each vehicle is broken down into several intermediate waypoints. The client process waits until the vehicle has reached an intermediate waypoint before messaging the server. The client does not proceed to the next intermediate waypoint until it has received an instruction to do so from the server. If any error message is received during an intermediate state, the client transitions to the `EXIT_FAILURE` state and sends an error message to the server.

Exit Success

Once the vehicle has reached desired waypoint, the client process transitions to the `EXIT_SUCCESS` state and closes the connection with the server.

3.2 Implementation Details

We implemented the FSM controller running on our GCS entirely in the Python programming language. This was done mainly to take advantage of the several open source Python modules for UAV and inter-process communication. Figure 3.3 shows a diagram of our software stack.

3.2.1 MAVLink Bindings

As mentioned in Section 2.1, the Pixhawk control board receives control input over a UART connection using the Micro Air Vehicle Communication Protocol (MAVLink) [6]. This is connected to a telemetry radio to allow for control from a remote transmitter using the MAVLink protocol. The MAVLink protocol allows for high-efficiency packet transmission and is well tested across several platforms including the Pixhawk and Parrot AR Drone.

A GCS must transmit messages using the MAVLink protocol in order to control a MAVLink-enabled vehicle. By default, MAVLink provides a binding to the C programming language that allows the GCS to transmit C structures encoding commands. A Python library named `pymavlink` built upon the C library provides Python bindings and allows for the transmission of Python objects with MAVLink.

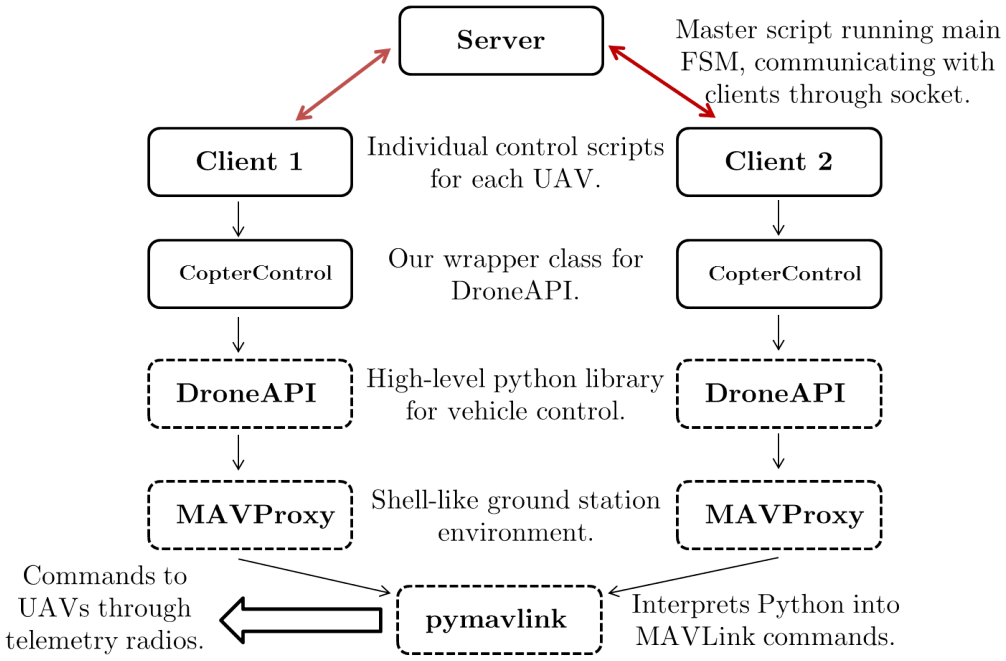


Figure 3.3: Software stack used. Dotted lines indicate third-party software

We do not directly use `pymavlink` in our implementations; however, both MAVProxy and DroneKit (see Section 3.2.2) do. The `pymavlink` MAVLink binding provides the first layer of abstraction in our software stack.

3.2.2 DroneKit

DroneKit [7] is a set of open source application program interfaces (APIs) developed by 3D Robotics that allows for the development of software applications to control UAVs using the MAVLink protocol. It is built upon the `pymavlink` module. Currently, DroneKit provides APIs for Python, Java, and Android.

DroneKit's Python module, `droneapi` provides a library of high-level classes for general vehicle control. For our purposes, the most important class is `droneapi.lib.Vehicle`, which provides several convenient methods. One `Vehicle` class is instantiated per UAV connection. Once instantiated, we can use methods like `Vehicle.commands.goto(lat, lon, alt)` to send the vehicle to a specific waypoint, `Vehicle.commands.takeoff(alt)` to initiate a takeoff, and `Vehicle.armed = True` to arm the vehicle. We can also use `droneapi` to change the vehicle's mode, and send raw MAVLink messages.

Though the `droneapi` module is extensive, well-documented, and easy to use, it still does not provide all of the functionality that we require for vehicle control. For this reason, we implemented our own wrapper module for `droneapi` called

`copter_control`. Our `copter_control` module contains a `CopterControl` class that wraps several important `droneapi` methods, as well as new functions for setting vehicle yaw and velocity (`CopterControl.set_yaw(th)` and `CopterControl.set_velocity(v_x, v_y, v_z)`.

It is required that scripts using the `dronapi` module be run in the MAVProxy [8] environment. MAVProxy is an open source, shell-like GCS environment built upon `pymavlink`. Using MAVProxy is helpful because (1) all relevant UAV communication modules needed by `droneapi` are loaded automatically and (2) through the use of multithreading, it provides a live-feed of vehicle status information while a `droneapi` script is running. However, a downside is that only one MAVProxy process can be spawned per vehicle; our solution to this problem is described in Section 3.2.3.

To execute a `droneapi` script, we first spawn MAVproxy via `python mavproxy.py --args`. Here `args` is the serial port on the laptop that the telemetry radio is connected to. Once the MAVProxy shell loads, we can then load the `droneapi` module by typing `module load droneapi`. We can then execute a script by typing `api start script.py`. Figure 3.4 shows a screenshot of a what a typical MAVProxy session might look like when running a simple `droneapi` script. Note that messages from the vehicle and vehicle mode are displayed even while the script is executing.

```
STABILIZE> api start simple_goto.py
STABILIZE> Got MAVLink msg: MISSION_ACK {target_system : 255, target_component : 0, type : 0}
GUIDED> Mode GUIDED
APIThread-0 exiting...
Got MAVLink msg: MISSION_ACK {target_system : 255, target_component : 0, type : 0}
```

Figure 3.4: Screenshot of MAVProxy shell when running a simple `droneapi` script.

3.2.3 Inter-Process Communication

One of the downsides of using DroneKit and MAVproxy is the lack of implicit multi-vehicle support. One MAVProxy process can only handle one vehicle connection. To overcome this, we launch two client scripts in two different MAVProxy processes that communicate with a server process through socket connections (this was briefly discussed in Section 3.1). Each MAVProxy process is spawned by specifying the serial port of the telemetry radio communicating with the corresponding vehicle.

We utilize the Python `socket` module for inter-process communication (IPC). Upon being spawn, the server process sets up a socket connection and binds to `localhost:8080` using the `socket.socket` constructor and `socket.bind`. The port

8080 is arbitrary, and any high-number port is okay to use. The server waits for clients to connect using the `socket.listen` method.

Once a connection is established, message passing is handled using the `socket.recv` and `socket.send` methods. Both methods expect a string format. The `socket.recv` call is blocking, meaning that it waits until a message has been received before continuing execution.

We require a message format that allows us to encode commands and metadata, as well as being easy to interpret pro grammatically. For these reasons, we chose to pass messages in JavaScript Object Notation, or JSON. JSON is convenient because it allows us to format each message in a consistent format and is easily interpreted to Python dictionary data types using the Python `json` module. For example, to send a "takeoff to 15 meters" message to a client from the server, we construct a dictionary of the form {`"MSG":"TAKEOFF", "ARGS":[15]`}. The `ARGS` entry has a different meaning depending on the message. The dictionary can be dumped into a JSON string using `json.dumps`; on the client's end, a JSON string can be loaded into a dictionary for easy interpretation using `json.loads`.

Chapter 4

Vision-Based UAV Localization

In order to increase the effectiveness of the video stabilization performed in post (see Section 5), we require a precise relative position estimate between the two UAVs. While the Pixhawk flight controller logs an extended Kalman filtered state estimate that we can retrieve for video stabilization, we wish to increase the confidence of this estimate through other techniques.

To this end, we considered several possibilities for accurately measuring the baseline between the two drones. Our initial selected technique consisted of a laser-based method, in which a reflective laser distance meter is used to measure the baseline of the system. This laser would be mounted on one UAV using a two degree-of-freedom targeting gimbal. Also mounted to this gimbal is a targeting camera that is used to track an image on the other UAV, thereby keeping the laser centered on its target. While we made significant progress on on the laser measurement system, our on-board Beaglebone Black processor was not fast enough to allow for target tracking in real time. Thus, centering the laser on the target in real time is not feasible with our hardware (this is discussed more in Section 4.1).

While real-time laser measurement is not possible, we are able to get obtain reasonable relative position estimates using purely computer-vision. Specifically, we mount a neon-pink ball on one of the vehicles and a statically mounted Logitech C920 webcam on the other (see Section 4.2). During flight, the webcam records video of the other vehicle. In post-processing, we run a ball-tracking algorithm that logs both the radius and position of the ball in the image plane. The radius is proportional to the distance, while the ball's position in the image plane gives us an estimate of the other vehicle's relative altitude and relative position along the axis perpendicular to the baseline. Currently, we discard the radius measurement due to excessive noise.

4.1 Laser-Based Distance Measurement

While vision-based ball tracking provides a reliable method for measuring the relative translation of the two UAVs in the directions orthogonal to the axis of the tracking camera, depth measurements made using the apparent radius of the ball in the image plane are less accurate. To compensate for this, we designed a system for using a laser distance meter to measure the distance from the tracking camera to the ball. The system uses a UNI-T UT390B laser distance meter, shown in Figure 4.1. The UT390B is lightweight and small but highly accurate, providing sub-cm accuracy and a range of up to 45m, all for about $50. The UT390B is packaged as a hand-held laser tape measure. However, in addition to displaying measurements on an LCD screen, it transfers its data over 115200 Baud UART accessible via an on-board debug port. Instructions for modifying the UT390B are available online from Fuller [9].

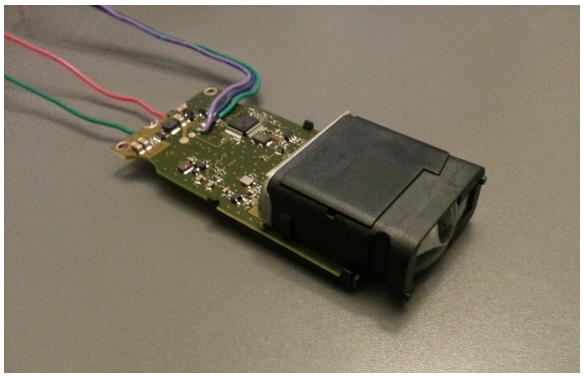


Figure 4.1: UT390B laser distance meter used for depth measurement.

The UT390B and tracking webcam are mounted together in a 3D-printed housing on a two degree-of-freedom gimbal actuated by high-torque servomotors, shown in Figure 4.2. The laser system is powered by two voltage regulators - an LM7833 to provide a constant 3.3V supply to the UT390B itself, and an LM7805 to provide power to the servos. These can be powered using the X8's onboard 12V supply, which is intended for use with on-board video equipment.

A working prototype of the laser system was tested using a Beaglebone Black as a control computer running the image capture and HSV masking routine described in Section 4.4. While functional, the Beaglebone was unable to run the masking algorithm quickly enough for responsive tracking at image sizes larger than 135x240 pixels. This severely limited the system's usefulness in tracking applications where the ball was further than three or four feet away, as would be the case in our system.

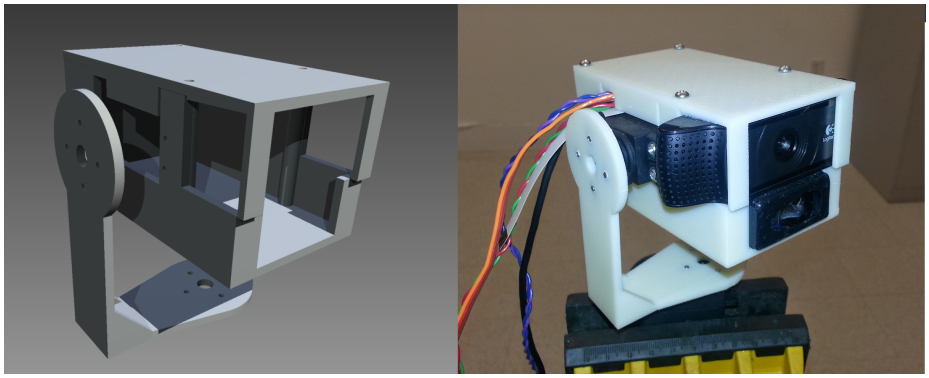


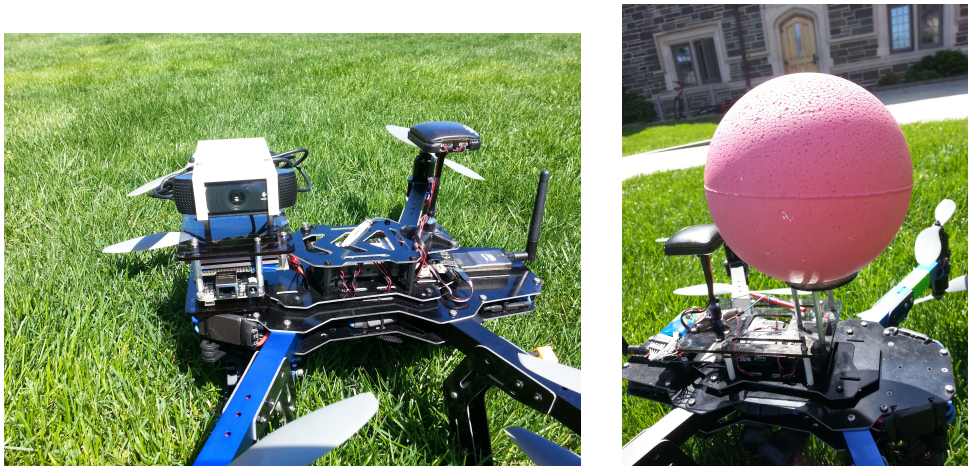
Figure 4.2: CAD model and 3d printed tracking gimbal for laser and targeting webcam.

4.1.1 Required Improvements for Practical Use

The laser system described above could be made fast enough for in-flight tracking if the speed of the masking algorithm were increased or if a faster processing platform was used. An algorithm that used a combination of image cropping and subsampling to reduce the size of the image before searching for the ball could improve performance. These size reduction operations could be performed based on the predicted position of the ball using its history. Additionally, the processing platform could be upgraded to a multicore or GPU-based system. Unfortunately, many processor architectures that fit the speed requirements of the tracking system, such as the Nvidia Tegra, would overload the X8's on-board power supply, so there is a balance to be struck between tracking performance and conformance to the power and weight capabilities of the X8.

4.2 Flight Hardware Configuration

Without sufficient on-board processing power to run the full tracking gimbal and laser, we statically mount the C920 webcam and Beaglebone Black near the front of one X8, such that the camera is oriented sideways to observe the partner UAV. The Beaglebone and camera are powered using a L78S05CV 5V regulator with a 10 watt heat sink. The L7805CV can supply up to 2A of current, which is the recommended amount for the Beaglebone Black when powering other components through the USB port. The mount is constructed using laser-cut acrylic, aluminum standoffs and 3D-printed components, designed to provide a firm and protective housing while providing access to the Beaglebone's ports, buttons, and status LEDs for debugging. On the



(a) C920 webcam and Beaglebone Black (b) Vision target ball

Figure 4.3: Mounting Configurations

second X8, a pink Styrofoam ball is mounted to act as a target for the vision tracking algorithms.

The use of three separate cameras in the system necessitates a complicated startup routine to ensure that the data from all sources is properly synchronized. The two Gopros, the C920, and both Pixhawk flight controllers must all have their data referenced to a unified time frame for the rectification process described in Chapter 5. We synchronize using the power LEDs on the two Pixhawks. The Pixhawk logs are referenced to the moment 340ms after the power LED turns on. By filming the power LEDs with all three cameras at during the initialization of the Pixhawks, we can determine the time offset between videos and the time offset between each video and the Pixhawk logs.

4.3 Existing Work

We considered several object tracking and detection schemes, such as mean shift, ensemble of exemplar SVMs, and feature-based matching, before settling on our final ball-tracking algorithm. Here, we provide a brief overview of the methods as relevant to our project.

The mean shift algorithm, proposed by Cominiciu et al. [10], attempts to draw a bounding box around an area of an image with maximum pixel intensity. The image must first be filtered to highlight the pixels of the object to track. Mean shift is an iterative algorithm that shifts the bounding window until the weighted mean of the

pixel intensity is in the center of the window. For our purposes, however, mean shift does not suffice. The main drawback is that the size of the bounding box does not scale with respect to the size of the object we are tracking. Since we wish to relate the size of the tracked object to a relative distance measurement, knowing the size of the bounding box is important. Mean shift is also not robust to occlusions [11].

The ensemble of exemplar support vector machines (SVMs) object detector, proposed by Malisiewicz et al. [12], works by training a separate SVM (see Section 4.4.2) on different instances of an object and then using a sliding window detector to calculate a response strength. The algorithm is robust and performs well, but is unnecessarily complex and inefficient for our purposes. Unlike Malisiewicz et al., we are trying to detect a specific object, not a category of objects.

Feature-based matching for object detection involves extracting keypoints (such as SIFT [13]) in a template and scene, then determining correspondences between the template and scene for object location [11]. During the early stages of our project, we experimented with feature-based matching. While showing promising results, our algorithm was computationally expensive and did not work for detecting templates farther than 5 meters from the camera. This is because feature-rich keypoints become much harder to detect as the template gets smaller.

4.4 Ball-tracking Algorithm

Our current method for vision-based UAV localization relies on a ball-tracking algorithm that is fairly robust to occlusions, illumination variations, and distance. Additionally, our algorithm is efficient and simple to implement. Our method consists of (1) background segmentation of the video frame using pixel classification, (2) erosion and dilation for removing noise, (3) circle detection using the circle Hough Transform (CHT) [14], and (4) low pass filtering radius and position of the detected circle (see Algorithm 1). The crux of our method lies in the circle detection step using the CHT (see Section 4.4.3). It should be noted that object detection using the CHT (and general Hough Transform) is not novel (see Liu et al. [15] and Yu et al. [16] for examples, as well as Yuen et al.[17] for a good discussion of the Hough Transform applied to object detection). Additionally, the ability to segment out the background due to the unique color of the ball greatly reduces the need for overly complex algorithms for object detection. However, though our method provides us with usable results, the Conditional Density Propagation (CONDENSATION) tracking algorithm [18] is likely to improve tracking results in cluttered environments, and can be explored in

the future.

We implemented the ball-tracking algorithm entirely in the Python programming language, making use of the OpenCV (`cv2`) [19] and scikit-learn (`sklearn`) [20] libraries.

Algorithm 1 Ball-tracking

```
while True do
    frame_num, frame ← getVideoFrame()
    if not isValid(frame) then
        quit()
    end if
    mask ← segmentBackground(frame)
    mask ← close(mask)
    mask ← open(mask)
    mask ← gaussianBlur(mask)
    center_x, center_y, radius = houghCircleTransform(mask)
    if isValid(center_x, center_y, radius) then
        center_x, center_y, radius ← iirFilter(center_x, center_y, radius)
        log(frame_num, center_x, center_y, radius)
    end if
end while
```

4.4.1 Naive Masking

The first step of our method is to segment out the background to highlight the ball, making it easier to detect. Our initial approach consisted of converting the video frame to hue-saturation-value (HSV) representation by utilizing the `cv2.cvtColor` function. We then thresholded the frame, masking out pixels not within a certain range of HSV values (this was done with help of the `cv2.inRange` function). The result was a binary image, ideally with value `0` for a background pixel and value `1` for a ball pixel. The benefit of working with the HSV model rather than the additive RGB model is being able to separate color components from intensity, making parameter tuning more intuitive.

Though this approach is intuitive and easy to implement, it requires constant HSV threshold parameter tuning for different lighting situations, a tedious process. Furthermore, it is often difficult to get reasonable masking since six parameters (an upper and lower bound for H, S, and V) must be tuned manually. If the range is too small, small disturbances can severely hinder masking. On the other hand, if the range is too high, background pixels are falsely classified as ball pixels.

4.4.2 Image Segmentation using the Support Vector Machine

We implement background segmentation using a support vector machine (SVM), saving us from excessive parameter adjustment.

Overview of the SVM

The SVM is a supervised learning model used for classification. Given a set of labeled training data that are linearly separable into two classes, the SVM attempts to a fit a hyperplane through the data that represents the largest margin between the two classes.

Formally, the training data, $\mathcal{D}$, consisting of n points can be denoted as

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) \quad | \quad \mathbf{x}_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^{n} \tag{4.4.1}$$

where each $\mathbf{x}_i$ is a p-dimensional vector. A hyperplane $\mathcal{H}$ takes the form

$$\mathbf{w} \bullet \mathbf{x} + b = 0 \tag{4.4.2}$$

where $\mathbf{w} \in \mathbb{R}^p$ is normal to the hyperplane and $b \in \mathbb{R}$ is an offset. A hyperplane $\mathcal{H}$ divides the space $\mathbb{R}^p$ into a positive half space and negative half space. Each point $\mathbf{x} \in \mathbb{R}^p$ can be assigned to a label $\hat{y} = +1$ if $\mathbf{w} \bullet \mathbf{x} + b \geq 0$ or $\hat{y} = -1$ if $\mathbf{w} \bullet \mathbf{x} + b < 0$. These conditions can be written compactly as

$$y_i(\mathbf{w} \bullet \mathbf{x}_i + b) > 0 \tag{4.4.3}$$

for $i = 1, ..., n$. Furthermore there exists a scalar $\alpha > 0$ such that

$$y_i(\alpha\mathbf{w} \bullet \mathbf{x}_i + \alpha b) \geq 1. \tag{4.4.4}$$

Since $(\mathbf{w}, b)$ and $(\alpha\mathbf{w}, \alpha b)$ define the same hyperplane, we can interpret 4.4.4 as

$$\min_i y_i(\mathbf{w} \bullet \mathbf{x}_i + b) = 1. \tag{4.4.5}$$

The distance between a point $\mathbf{x}_i$ and hyperplane $\mathcal{H}$ can be written as

$$d(\mathbf{x}_i, \mathcal{H}) = \frac{y_i(\mathbf{w} \bullet \mathbf{x}_i + b)}{\|\mathbf{w}\|}. \tag{4.4.6}$$

We can thus maximize the margin of the hyperplane $\mathcal{H}$ by minimizing $\|\mathbf{w}\|$. For mathematical convenience, the optimization problem is phrased in terms of $\frac{1}{2}\|\mathbf{w}\|^2$:

$$\begin{aligned} &\min_{\mathbf{w}\in\mathbb{R}^p, b\in\mathbb{R}} \frac{1}{2}\|\mathbf{w}\|^2 \\ &\quad s.t. \quad y_i(\mathbf{w} \bullet \mathbf{x}_i + b) \geq 1 \\ &\qquad \min_i y_i(\mathbf{w} \bullet \mathbf{x}_i + b) = 1, \quad i = 1, ..., n. \end{aligned} \tag{4.4.7}$$

It is not feasible to assume that the training data are linearly separable. Thus, the concept of soft margin is introduced to allow for mislabeled training examples. The new optimization problem can be written as

$$\begin{aligned} &\min_{\mathbf{w}\in\mathbb{R}^p, b\in\mathbb{R}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} s_i \\ &\quad s.t. \quad y_i(\mathbf{w} \bullet \mathbf{x}_i + b) \geq 1 - s_i \\ &\qquad s_i \geq 0, \quad i = 1, ..., n. \end{aligned} \tag{4.4.8}$$

where each s_i is a slack variable representing the degree of misclassification of point $\mathbf{x}_i$ and C is the penalty for misclassification. A higher C parameter leads to better classification of training data, but can lead to overfitting. The dual form of this problem can be written as

$$\begin{aligned} &\max_{\alpha\in\mathbb{R}^n} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j (\mathbf{x}_i \bullet \mathbf{x}_j) \\ &\quad s.t. \quad 0 \leq \alpha_i \leq C \\ &\qquad \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} \tag{4.4.9}$$

It is also not feasible to assume that the training data are perfectly separable. It is often better to create a nonlinear classification model. To do this, Vapnik et al. introduced the "kernel trick" to replace the inner product $(\mathbf{x}_i \bullet \mathbf{x}_j)$ with a nonlinear kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$. This allows for classification in a transformed, often higher-dimensional, feature space. One of the most widely used kernels is the radial basis

function, which is denoted by

$$k(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}. \tag{4.4.10}$$

We use the radial basis function kernel for pixel classification. [21]

Pixel Classification Using the SVM

We use a two-class support vector machine to classify pixels in the video frame as either "ball" pixels or "background" pixels. Each $\mathbf{x}_i \in \mathbb{R}^3$ is a pixel represented in RGB color space.

To train the classifier, we compiled a set of images of the pink ball in various lighting situations. We also compiled images of backgrounds that we would likely encounter in the flight videos. After formatting the pixel data, we train a SVM by creating an instance of the `sklearn.svm.SVC` and calling the `fit(X, y)` function. Here, `X` is the training data and the vector `y` contains class labels. The `sklearn.svm.SVC` class is based off the popular LIBSVM [22] library. During preliminary tests with the trained model, false positives are weeded out through hard negative mining and `C` and `gamma` parameters are tuned. Background segmentation is achieved by passing in a formatted video frame to the model's `predict` function. The result is a binary mask, ideally with value `0` for a background pixel and value `1` for a ball pixel.

We obtain much more consistent, and often better, results using the SVM for background segmentation over HSV mask. For a comparison of ball detection results using HSV (using the same HSV parameters throughout) and SVM background segmentation, see figure 4.4. Notice how HSV segmentation allows for ball detection in cloudy and indoor environments, but not sunny environments. The SVM trained with a single training data set performs well in all three environments.

4.4.3 Circle Detection

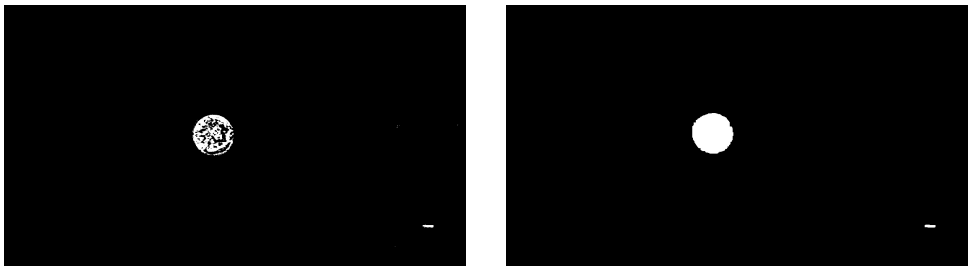
After the background has been successfully segmented out, circle detection (via the CHT) is used to locate the ball in the image plane.

Preprocessing

We find that it is helpful to preprocess the mask before running the CHT algorithm. Often times, the binary mask is noisy, containing falsely detected pixels and black

Figure 4.4: Comparison of ball detection results displayed on original frame using HSV (top) and SVM (bottom) background segmentation.

specks in the ball region. Eroding and dilating the mask helps to solve both of these problems. Erosion consists of convolving the binary mask with a square matrix kernel of `1`'s; a pixel in the image remains `1` only if all of the pixels in under the kernel have value `1`, else it is changed to `0`. Dilation works in the same way, except that a pixel in the image is `1` if at least one pixel under the kernel is `1`. Dilation followed by erosion, or "closing" helps to fill in the black specks found on the object. Erosion followed dilation, or "opening" helps to remove white specks in the background [23]. We close, then open the mask retrieved through pixel classification using the `cv2.morphologyEx` function. See Figure 4.5 for a comparison of the raw mask and the mask after morphological operations.



(a) Binary mask with no morphological operations

(b) Binary mask with morphological operations

Figure 4.5: Morphological Operations

We find that blurring the binary mask with a Gaussian kernel (after converting

the raw mask to a `float` representation) helps to remove high frequency noise, aiding with the edge detection portion of the circle detection method described in the next section. A Gaussian kernel is a square matrix that samples the 2D Gaussian function

$$f(x, y; \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(x^2+y^2\right)/2\sigma^2} \tag{4.4.11}$$

within a certain range. We implement Gaussian blurring using the `cv2.GaussianBlur` function.

Circle Hough Transform

After preprocessing the binary mask retrieved from background segmentation, we use the circle Hough Transform to detect the ball's radius and position in the image plane.

Recall that a circle can be parametrized as

$$\begin{aligned} x &= a + R\cos(\theta) \\ y &= b + R\sin(\theta) \end{aligned} \tag{4.4.12}$$

where (a, b) are the coordinates of the center of the circle and R is the radius. The basic idea of the CHT is that if a point (x, y) is fixed, then the parameter space (a, b, R) is 3-dimensional. The set of all parameters satisfying 4.4.12 is a hollow cone with apex $(x, y, 0)$. Thus, in 3D space, circle parameters can be identified by the intersection conic surfaces that are defined by points on the circle. This problem is divided into two stages: (1) fixing radius and building a 2D accumulator matrix to find optimal circle centers and (2) finding the optimal radii in 1D parameter space. [14]

We use the `cv2.HoughCircles(image, method, param1, param2)` function for circle detection using the CHT method. The function returns a list `(x, y, r)` triples in order of confidence. The `method` parameter specifies which algorithm to use. We use the Hough Gradient method (`cv2.HOUGH_GRADIENT`) described by Yuen et al. [17], which is efficient in both execution time and memory since it does not require the expensive accumulator matrices described above. This method first runs a gradient-based edge detection scheme. It then uses the gradient's orientation in order to determine the line on which the circle radius lives. The `param1` parameter specifies the gradient value to use, while `param2` is the accumulator threshold. As `param2` decreases, more false circles are detected. However, since we are interested in detecting

only one circle by retrieving the first element of the returned list, we can keep `param2` fairly small.

4.4.4 Infinite Impulse Response Filter

We find that while the detected circle position and radius is accurate, it is jittery from frame to frame. To overcome this, we low-pass filter the detected circle positions and radii from frame to frame with a single-tap, recursive, infinite impulse response (IIR) filter of the form:

$$\mathbf{y}[t] = \alpha\mathbf{y}[t-1] + (1-\alpha)\mathbf{x}[t]. \tag{4.4.13}$$

Here $\mathbf{x}[t]$ is the unfiltered state of the current video frame. In our case, the state is (x, y, r), denoting the detected circle's (x, y) location and radius, respectively. The $\mathbf{y}[t-1]$ term is the filtered state of the previous frame, while $\mathbf{y}[t]$ is the current frame's filtered state. The α parameter determines how strongly high frequencies are attenuated. A higher α value leads to a higher low-pass filter. Figure 4.6 shows the absolute value of the filter's frequency response for different values of α. Note how high frequencies are more attenuated for higher values of α. [24]

Empirically, we found that an a value of $\alpha = 0.5$ worked best for our application. It should be noted that the idea to use low-pass filtering in object tracking applications was brought up during a final project for COS 429: Computer Vision completed by Ankush Gola and David Fridovich-Keil.

4.4.5 State Estimation Using Detected Ball

While robust, our ball-tracking algorithm fails to detect the ball in certain frames. However, we can overcome this quite nicely using interpolation. In either case, having found the pixel coordinates of the target ball, we now seek a reliable method for estimating the location of the ball (and by extension the UAV carrying carrying it) in three-dimensional space. Figure 4.7 shows the detected ball in a video frame.

We begin by defining several different coordinate systems. The first coordinate system is the two-dimensional pixel coordinate frame, which is centered on the top left corner of the each image. Its x axis extends to the right across the top of the image and its y axis extends downwards along the left edge. Pixel coordinates take

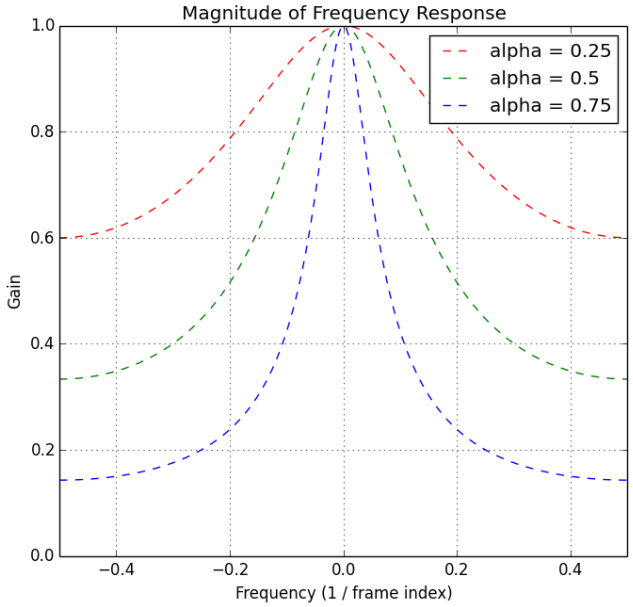


Figure 4.6: IIR filter frequency response, plotted for different values of α.

Figure 4.7: Ball detected in a video frame.

the form of positive numbers restricted by the bounds of the image:

$$\begin{bmatrix} x \\ y \end{bmatrix}, x \in [0, w), y \in [0, h) \tag{4.4.14}$$

We can also represent points in the image using their homogeneous coordinates, allowing affine and projective transformations to be represented using a matrix. Homogeneous coordinates can be thought of as an extension of standard Euclidean coordinates where two vectors are equivalent if they differ by a constant factor. Homogeneous coordinates define a point in projective space $\mathbb{P}^n$. For a point in our 2D pixel frame, the homogeneous coordinates take the form of a 3-vector $(x, y, z)^T$. To convert the

point to $\mathbb{R}^2$, simply divide by the last element z: $(x/z, y/z)^T$, assuming z is non-zero [25]. For the purposes of the image plane homogeneous coordinate system, all points with the same $(x/z, y/z)^T$ are the same point.

Our next coordinate system is the camera frame, $\mathcal{C}$. The camera frame is a three-dimensional coordinate system centered at the focal point of the camera's lens, in front of the image plane. Its z axis extends outward through the camera lens in the direction of the scene, the x axis is oriented out the right side of the camera, and the y axis extends downward through the bottom of the camera. To transform points in the camera frame to image homogeneous coordinates, we use the intrinsic camera matrix $\mathbf{F}$:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{image} = \mathbf{F} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\mathcal{C}} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\mathcal{C}} \tag{4.4.15}$$

f_x and f_y are the focal length of the camera in the x and y directions. $(c_x, c_y)^T$ is the center of the image in pixel coordinates. We can calculate the intrinsic matrix using opencv's function `cv2.calibrateCamera(objectPoints, imagePoints)`, which uses a set of image points and their corresponding camera coordinates to calculate the camera's intrinsic matrix. We acquire these points by taking images of a checkerboard pattern with a known dimension and planar shape (figure 4.8), using the image acquisition script provided by Rillahan [26]. For the Logitech C920 webcam, we calculate

$$\mathbf{F} \approx \begin{bmatrix} 1528 & 0 & 967 \\ 0 & 1524 & 572 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4.16}$$

Since $\mathbf{F}$ governs the projection of points from the camera coordinate frame into the image pixel homogeneous frame, we can use $\mathbf{F}^{-1}$ to find pixels' camera coordinates. It's important to note that the depth of a pixel in camera coordinates cannot be determined from its pixel coordinates alone. The result of the $\mathbf{F}^{-1}$ transformation is merely a direction in the camera frame. As with the homogeneous pixel coordinates, this vector can be scaled by any factor still represent the same point.

We convert this direction to a point by assuming the distance between the two UAVs is approximately the desired baseline b. We approximate the location of the ball in camera coordinates as:

$$\mathbf{X}_{\mathcal{C}} = \frac{b}{\|\mathbf{F}^{-1}\mathbf{X}_{image}\|} \mathbf{F}^{-1}\mathbf{X}_{image} \tag{4.4.17}$$

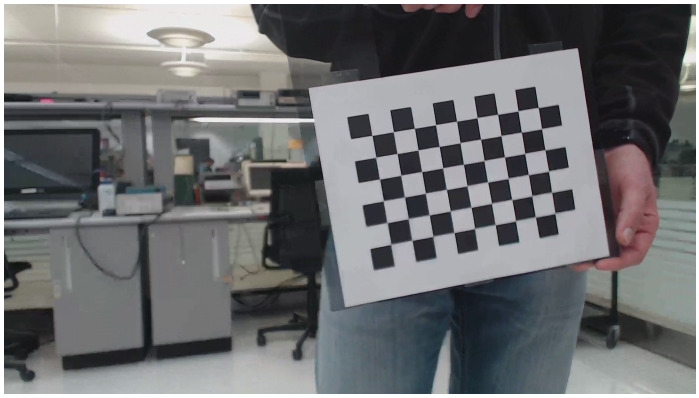


Figure 4.8: Typical camera calibration image.

For the state estimate of the ball to be useful in the sensor fusion techniques of Section 5.2, we must not only construct an estimated location for the ball, but also a covariance matrix describing our certainty in that estimate in three-dimensional space. For now, we represent the covariance by selecting six points that might be drawn from the probability distribution representing the ball's location. These points are located close to the ball in the x and y dimensions of the camera frame, but far away from it in the z dimension, indicating that our state estimate has a high degree of certainty in the ball's position perpendicular to the z axis, but very little information about its depth. These points are propagated through subsequent frames with the center point, and their covariance is calculated as the last step in the state estimation process. These covariance points are shown in green in Figure 4.9.

Having determined the location of the mean and edges of the ball's location distribution in the camera frame, we then transform these points into the X8 body frame, $\mathcal{B}$. This frame has its positive x axis extending out the front of the X8, y axis directed out the right side of the UAV, and z axis directed downward out of the bottom of the X8. To move into the body frame from the camera frame, we merely swap the negative z axis with the positive y axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\mathcal{B}} = \mathbf{R}_{\mathcal{C}}^{\mathcal{B}} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\mathcal{C}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\mathcal{C}} \tag{4.4.18}$$

This is equivalent to a roll rotation of 90^o downward, as we will see.

Our next coordinate frame is the vehicle frame, $\mathcal{V}$, which has the same center as the body frame, but axes aligned with the cardinal directions. Positive x is North, positive y is East, and positive z is directed downwards. The X8's onboard IMU

provides a highly accurate estimate of its roll (φ), pitch (θ) and yaw (ψ). As noted in [21] and [27], these angles provide the Tait-Bryan parameterization of the rotation from the vehicle frame to the body frame. The corresponding rotation matrix can be calculated as follows:

$$\begin{aligned}
\mathbf{R}_{\mathcal{V}}^{\mathcal{B}} &= \mathbf{R}_{\varphi}\mathbf{R}_{\theta}\mathbf{R}_{\psi} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & \sin(\varphi) \\ 0 & -\sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c(\theta)c(\psi) & c(\theta)s(\psi) & -s(\theta) \\ s(\varphi)s(\theta)c(\psi) - c(\varphi)s(\psi) & s(\varphi)s(\theta)s(\psi) + c(\varphi)c(\psi) & s(\varphi)c(\theta) \\ c(\varphi)s(\theta)c(\psi) + s(\varphi)s(\psi) & c(\varphi)s(\theta)s(\psi) - s(\varphi)c(\psi) & c(\varphi)c(\theta) \end{bmatrix}
\end{aligned} \tag{4.4.19}$$

Each individual rotation matrix $\mathbf{R}_{\alpha}$ applies a clockwise rotation to a point about the axis in question. It's important to note that this clockwise rotation (which would not be considered right-handed) corresponds to a counter-clockwise rotation of the axis itself. Thus, if the angles (φ, θ, ψ) denote the heading of the multirotor in the vehicle frame, $\mathbf{R}_{\mathcal{V}}^{\mathcal{B}}$ represents the rotation that must be applied to points in the vehicle frame to find their coordinates in the body frame. We are interested in $(\mathbf{R}_{\mathcal{V}}^{\mathcal{B}})^{-1} = \mathbf{R}_{\mathcal{B}}^{\mathcal{V}}$, which transforms points from the body frame the vehicle frame.

Our final reference frame is the inertial frame, $\mathcal{I}$. This coordinate frame has axes parallel to the vehicle frame, but translated such that the origin of the coordinate system is always located at the position from which the X8 took off. The X8's GPS and on-board sensor fusion techniques provide a location estimate of the observing UAV in this frame, which can be used to translate between the vehicle and inertial frames. If $\mathbf{T}$ is the location of the X8 in the inertial frame, the location of a point in the vehicle frame is:

$$\mathbf{X}_{\mathcal{I}} = \mathbf{X}_{\mathcal{V}} + \mathbf{T} \tag{4.4.20}$$

Putting all this together, we can transform a point in homogeneous image coordinates to the inertial frame using:

$$\mathbf{X}_{\mathcal{I}} = \mathbf{R}_{\mathcal{B}}^{\mathcal{V}}\mathbf{R}_{\mathcal{C}}^{\mathcal{B}}\mathbf{F}^{-1}\left(\frac{b}{\|\mathbf{F}^{-1}\mathbf{X}_{image}\|}\mathbf{F}^{-1}\mathbf{X}_{image}\right) + \mathbf{T} \tag{4.4.21}$$

The red path shown in figure 4.9 is produced by applying this transformation to

a dataset collected during an actual flight. Appropriately, the path of the visually localized UAV appears to grow less accurate as it grows further from the observing UAV.

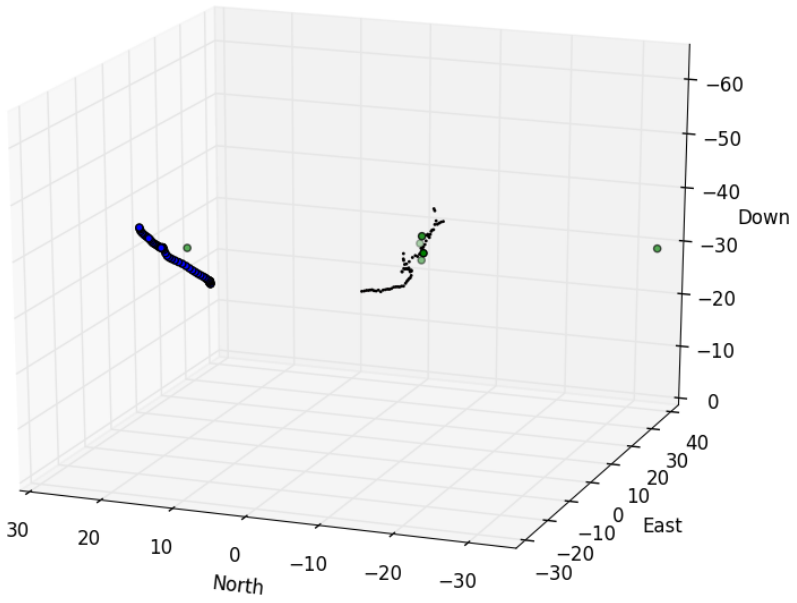


Figure 4.9: 3D plot of vision-based state estimation output in the inertial frame. Blue shows the position of the tracking UAV based on GPS and internal state estimation filters. Red shows the estimated position of the tracked UAV based only on visual information. Green points indicate the variance estimate transformed into the inertial frame for a single measurement.

Chapter 5

Post-Processing and Video Stabilization

Video clips collected with the X8s are subject to a variety of noise sources. The most significant of these are due to the physical movements of the UAVs in response to perturbations such as wind and atmospheric conditions. These movements result in image distortions as the perspective of each camera is translated and rotated in three-dimensional space, which must be accounted for in order for the video to be comfortably viewed in stereo. We establish maximum tolerable amplitudes for image movement in Section 5.1 and describe our technique for reducing apparent movements to within these thresholds in Section 5.3. Section 5.2 describes our state estimation and sensor fusion method for refining our estimate of each X8's position and heading, a prerequisite for accurate rectification.

5.1 Thresholds for Comfortable Viewing

Before we attempt to rectify images collected in flight into comfortably viewable 3D video, we must first establish the human tolerances for camera movement in a stereo system.

5.1.1 Human Depth Perception

The human visual system uses a variety of cues to provide depth information. These include lighting, texture, and occlusion contours in monocular images. Depth infor-

mation is also gained through stereopsis, wherein the horizontal disparities between images from different eyes provide 3D information. Stereopsis augments monocular cues and aids with depth perception even when a relatively complete depth map can be constructed from solely monocular cues [28]. In stereopsis, cortical signals from the brain instruct the eyes to converge, or toe inward, until horizontal disparities in a region of focus are eliminated. The two images are then fused together and perceived as a single 3D scene with depth information [29].

Visual Discomfort

While stereopsis is a very effective means of depth perception in the real world, there are a variety of factors that can cause discomfort or failure in the human stereo vision system when exposed to simulated stereopic effects. Park et al. describe several of these effects, including asthenopia (eyestrain), nausea, and reduced visual sensitivity. Diplopia (double vision) can occur if the horizontal disparity between the two images is too great [29]. Limits on the region in which stereopsis can occur have been studied by many including Yeh and Silverstein, who determine that diplopia begins when the horizontal disparity between two images exceeds the fusional area of the viewer [30]. These problems can easily arise in a poorly designed 3D image viewing system, and care must be taken to ensure that viewer discomfort does not arise. We attempt to characterize the specific sensitivity thresholds for users of the Oculus Rift in Section 5.1.2.

Error Correction

The human visual system corrects for misalignment of the two eyes largely through angular adjustments through the extraocular muscles around each eye. Six muscles on each eye are responsible for these rotations. Graham provides a thorough overview of their function: the lateral and medial recti rotate the eyes in the horizontal plane, towards or away from the nose. The superior rectus and inferior oblique both raise the eyes and control their roll, depending on the convergence of the gaze. The inferior rectus and superior oblique both lower the eyes and control their role, again depending on the convergence of the gaze [31].

The visual system's ability to correct for misalignments in specific directions is known as fusional amplitude. Fusional amplitude is greater in the horizontal direction, since it relies on the same advanced structures that mediate the convergence and divergence of the eyes [32]. This implies that our system must not permit a severe

amount of vertical disparity between the two images.

5.1.2 Oculus Rift Simulations

We use simulated three-dimensional scenes viewed with the Oculus Rift to characterize the tolerances for movement unique to our viewing medium. All simulations are produced in Blender, an open-source 3D modeling and graphics suite, using python scripting to automate batch processes.

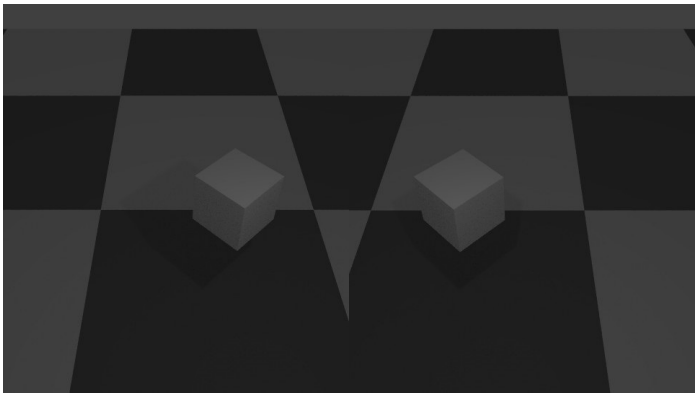


Figure 5.1: Left and Right eye perspectives of simulation environment.

In a simple 3D environment featuring a floating cube (shown in Figure 5.1), we begin by generating independent Gaussian-distributed camera movements in three dimensions. Viewers begin to have difficulty fusing the images when the standard deviation of the noise between the two cameras rises above 4% of the baseline. At this point, the cube begins to move in and out of fusion, with viewers seeing double and having difficulty focusing. This indicates that for aerial shots taken at a baseline of 10m, the individual images must not appear to be taken from perspectives with more than 40cm of undesired movement.

Humans with average vision rarely have difficulty viewing objects at short distances, since the standard human focal range and ocular convergence limits allow a person to view objects within a few centimeters of their face, which is a much shorter distance than is encountered in everyday life. However, our system will increase the minimum viewing distance significantly, since an expanded baseline stereo system requires much higher rates of convergence to view objects at the same absolute distance (see Figure 5.2). Additionally, the geometry of the Oculus Rift places further constraints on minimum viewing distance. For these reasons, a precise characterization of the minimum viewing distance is also important to our project. We find that objects located closer than 350% of the baseline to the subject become difficult to fuse

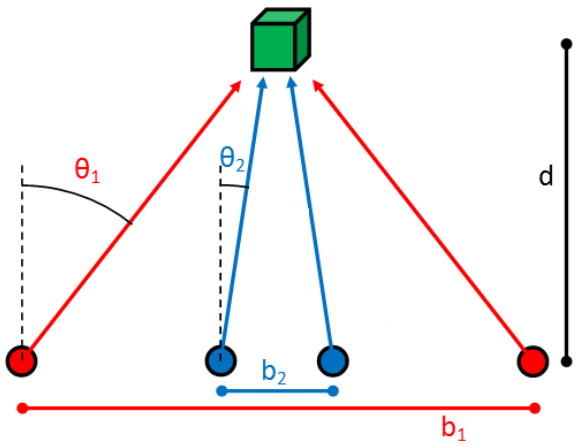


Figure 5.2: Geometric diagram of stereopsis at two different baselines, $b_1 > b_2$. Accordingly, the angle of convergence for the lines of sight is greater for larger baselines when viewing subjects at the same distance

in the Oculus Rift. Very close objects require uncomfortable degrees of convergence, and also begin to blur as the focal length assumed by the brain to correspond to such high degrees of convergence is outside focal powers that can be corrected by the Rift's lenses. This means that two UAVs flying at a baseline of 10m should not be used to film subjects closer than 35m.

5.2 Sensor fusion

Given data from both UAVs' real-time state estimation routines and relative position information from the vision tracking system described in chapter 4, we seek a cohesive state estimate for both UAVs that incorporates the data from both sources. State space techniques for sensor fusion have the benefit of accounting for the history of measurements from all sources when estimating the state at any given moment. This can considerably increase their accuracy provided their method for propagating the state estimate across time accurately represents the dynamics of the real system.

5.2.1 Dynamic Model

Before we can apply state-space filtering techniques to the X8, we must develop a dynamic model which describes how the multirotor system evolves over time and responds to control inputs. The dynamics of a traditional quadrotor are well-studied and can be adapted to the X8 multirotor without much trouble. As in Section 4.4.5,

we discuss coordinate systems in terms of the body, vehicle, and inertial frames. From Beard [27] and Tang [33], we know that the dynamics of a traditional quadrotor system can be characterized using a 12-dimensional state vector:

$$\mathbf{x} = \begin{pmatrix} x & y & z & u & v & w & \varphi & \theta & \psi & p & q & r \end{pmatrix}^T \tag{5.2.1}$$

Where $(x, y, z)^T$ represents the location of the quadrotor in inertial coordinates using the North-East-Down (NED) positive axes convention, $(u, v, w)^T$ is the corresponding right-handed velocity vector in the quadrotor's body frame, $(\varphi, \theta, \psi)^T$ is the roll-pitch-yaw rotation vector to move from the vehicle frame to the quadrotor's body frame, and $(p, q, r)^T$ is the corresponding angular velocity vector in the vehicle frame.

The three-dimensional rotation matrix to rotate from the vehicle frame to the body from can be calculated using Equation 4.4.19, restated here for convenience.

$$\begin{aligned}
\mathbf{R} &= \mathbf{R}_\varphi \mathbf{R}_\theta \mathbf{R}_\psi \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & \sin(\varphi) \\ 0 & -\sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c(\theta)c(\psi) & c(\theta)s(\psi) & -s(\theta) \\ s(\varphi)s(\theta)c(\psi) - c(\varphi)s(\psi) & s(\varphi)s(\theta)s(\psi) + c(\varphi)c(\psi) & s(\varphi)c(\theta) \\ c(\varphi)s(\theta)c(\psi) + s(\varphi)s(\psi) & c(\varphi)s(\theta)s(\psi) - s(\varphi)c(\psi) & c(\varphi)c(\theta) \end{bmatrix}
\end{aligned}$$

This rotation matrix allows us to express the first set of differential state equations, which describe the evolution of the quadcopter's position in terms of its body velocity:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \mathbf{R}^{-1} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{5.2.2}$$

where $\mathbf{R}^{-1} = \mathbf{R}^T$ transforms the body frame velocities to the vehicle frame. The body frame velocities change according to:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \begin{bmatrix} -g\sin(\theta) \\ g\cos(\theta)\sin(\varphi) \\ g\cos(\theta)\cos(\varphi) \end{bmatrix} + 1/m \begin{bmatrix} 0 \\ 0 \\ -F \end{bmatrix} \tag{5.2.3}$$

where the first term in the right hand side of the equation represents the Coriolis Effect as the quadcopter rotates in the inertial frame, the second represents the force of gravity transformed into the body frame, and the third represents the net thrust of all motors in the body frame. The transformation from the vehicle frame angular velocities $(p, q, r)^T$ to the angular displacements $(\varphi, \theta, \psi)^T$ produces the relevant differential equation:

$$\begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\varphi)\tan(\theta) & \cos(\varphi)\tan(\theta) \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi)\sec(\theta) & \cos(\varphi)\sec(\theta) \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{5.2.4}$$

and finally, assuming the inertia matrix of the quadcopter is diagonal:

$$\mathbf{I} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \tag{5.2.5}$$

Our final differential equation is:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{I_y - I_z}{I_x} qr \\ \frac{I_z - I_x}{I_y} pr \\ \frac{I_x - I_y}{I_z} pq \end{bmatrix} + \begin{bmatrix} \frac{\tau_\varphi}{I_x} \\ \frac{\tau_\theta}{I_y} \\ \frac{\tau_\psi}{I_z} \end{bmatrix} \tag{5.2.6}$$

where τ_φ, τ_θ, and τ_{psi} are the torques about the roll, pitch, and yaw axes respectively. We can further simplify Equations 5.2.3 and 5.2.6 if we assume that the second-order angular velocity terms are small:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} -g\sin(\theta) \\ g\cos(\theta)\sin(\varphi) \\ g\cos(\theta)\cos(\varphi) \end{bmatrix} + 1/m \begin{bmatrix} 0 \\ 0 \\ -F \end{bmatrix} \tag{5.2.7}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{\tau_\varphi}{I_x} \\ \frac{\tau_\theta}{I_y} \\ \frac{\tau_\psi}{I_z} \end{bmatrix} \tag{5.2.8}$$

Equations 5.2.2, 5.2.4, 5.2.7 and 5.2.8 provide a nonlinear, time-invariant dynamic model relating a set of generalized control inputs $(F, \tau_\varphi, \tau_\theta, \tau_\psi)^T$ and the state elements

from Equation 5.2.1 to the first derivative of the state.

Adaptation of Model to X8

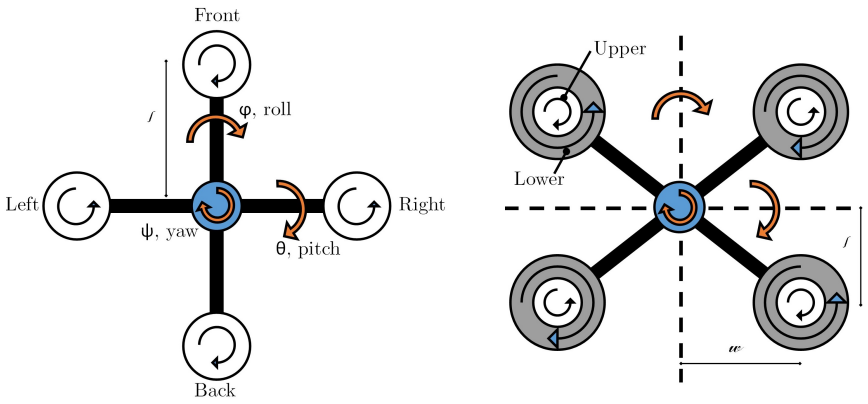


Figure 5.3: Traditional quadcopter body frame (left) and X8 body frame (right) with directions of positive rotation indicated.

To adapt the generalized quadrotor model to the X8, we first must express the control inputs $(F, \tau_\varphi, \tau_\theta, \tau_\psi)^T$ in terms of the X8's eight motor inputs. The torques and forces on a traditional quadrotor can be expressed as a linear combination of motor inputs:

$$\begin{bmatrix} F \\ \tau_\varphi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_1 & k_1 & k_1 & k_1 \\ 0 & -lk_1 & 0 & lk_1 \\ lk_1 & 0 & -lk_1 & 0 \\ -k_2 & k_2 & -k_2 & k_2 \end{bmatrix} \begin{bmatrix} \delta_f \\ \delta_r \\ \delta_b \\ \delta_l \end{bmatrix} \tag{5.2.9}$$

Here δ_f, δ_r, δ_b and δ_l are the front, right, back, and left motor inputs, respectively, and l is the distance from each motor to the center of the UAV. To adapt this to an X8, we first modify the motor matrix to describe a quadrotor with a body frame whose axes do not pass through the locations of the motors. Instead the motors are distributed at the front right, front left, back right, and back left corners of the quadrotor (Figure 5.3).

$$\begin{bmatrix} F \\ \tau_\varphi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_1 & k_1 & k_1 & k_1 \\ -wk_1 & wk_1 & -wk_1 & wk_1 \\ lk_1 & lk_1 & -lk_1 & -lk_1 \\ -k_2 & k_2 & -k_2 & k_2 \end{bmatrix} \begin{bmatrix} \delta_{fr} \\ \delta_{fl} \\ \delta_{br} \\ \delta_{bl} \end{bmatrix} \tag{5.2.10}$$

where l is distance from each motor to the pitch axis and w is the distance from each motor to the roll axis.

Having adapted the motor matrix to a multirotor with an X motor configuration, we now modify it for an 8-motor system. This changes the 4×4 motor matrix to a 4×8 matrix describing how the 8 motor inputs affect the force and torques.

$$\begin{bmatrix} F \\ \tau_\varphi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_1 & k_1 & k_1 & k_1 & k_1 & k_1 & k_1 & k_1 \\ -wk_1 & wk_1 & -wk_1 & wk_1 & -wk_1 & wk_1 & -wk_1 & wk_1 \\ lk_1 & lk_1 & -lk_1 & -lk_1 & lk_1 & lk_1 & -lk_1 & -lk_1 \\ k_2 & -k_2 & -k_2 & k_2 & -k_2 & k_2 & k_2 & -k_2 \end{bmatrix} \begin{bmatrix} \delta_{fru} \\ \delta_{flu} \\ \delta_{bru} \\ \delta_{blu} \\ \delta_{frl} \\ \delta_{fll} \\ \delta_{brl} \\ \delta_{bll} \end{bmatrix} \tag{5.2.11}$$

where δ_{flu} is the front-left-upper motor input, δ_{brl} is the back-right-lower motor input, and so forth. This is the motor matrix for the X8.

Having developed a dynamic model, we seek to identify the values of the parameters unique to our system. Many papers set out to characterize the system's response to angular control inputs, modeling the UAV's low-level controllers in the process. This is the technique used by both Tang [33] and Bouffard [34]. While this approach is useful for real-time filtering, where angular commands may be the only known control inputs, we have access to the Pixhawk's exact command outputs for all eight motors through the UAV's on-board logs. This makes it possible to model the UAV's physical dynamics alone, without prepending approximate angular controllers to the model. In the model presented in Equations 5.2.2, 5.2.4, 5.2.7, 5.2.8, and 5.2.11, there are technically eight parameters. However, since k_2 and I_z only appear together, we need only identify the value of the other six parameters and k_2/I_z.

The parameters m, l, and w are easily measured directly, and their corresponding values are shown in table 5.1. k_1 is calculated using Equations 5.2.7 and 5.2.11. We take the third dimension of the body frame and rearrange to find:

$$\dot{w} = g\cos(\theta)cos(\varphi) - \frac{k_1}{m}\sum \delta \tag{5.2.12}$$

$$k_1 = \frac{g\cos(\theta)cos(\varphi) - \dot{w}}{1/m\sum \delta} \tag{5.2.13}$$

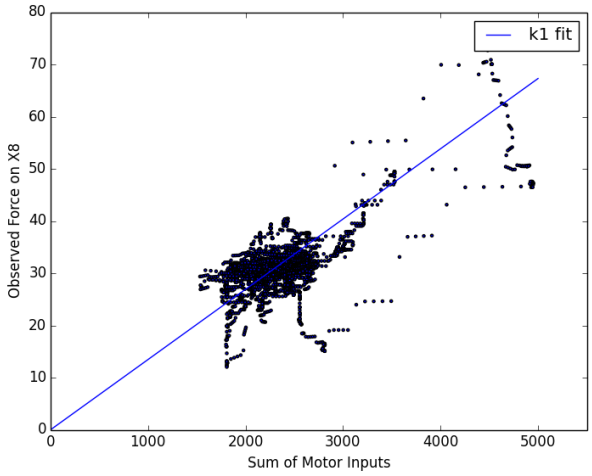


Figure 5.4: Plot showing linear fit for parameter k_1 using flight data. Deviations from the fit line are due to atmospheric perturbations and winds, which are unaccounted for in our model and assumed to be zero-mean.

To calculate the values of I_x and I_y, we take a similar approach with Equation 5.2.8.

$$\dot{p} = \frac{wk_1}{I_x}(-\delta_{fru} + \delta_{flu} - \delta_{bru} + \delta_{blu} - \delta_{frl} + \delta_{fll} - \delta_{brl} + \delta_{bll}) \tag{5.2.14}$$

$$\dot{q} = \frac{lk_1}{I_y}(\delta_{fru} + \delta_{flu} - \delta_{bru} - \delta_{blu} + \delta_{frl} + \delta_{fll} - \delta_{brl} - \delta_{bll}) \tag{5.2.15}$$

$$I_x = \frac{wk_1}{\dot{p}}(-\delta_{fru} + \delta_{flu} - \delta_{bru} + \delta_{blu} - \delta_{frl} + \delta_{fll} - \delta_{brl} + \delta_{bll}) \tag{5.2.16}$$

$$I_y = \frac{lk_1}{\dot{q}}(\delta_{fru} + \delta_{flu} - \delta_{bru} - \delta_{blu} + \delta_{frl} + \delta_{fll} - \delta_{brl} - \delta_{bll}) \tag{5.2.17}$$

The same is possible for the final model parameter, k_2/I_z.

$$\dot{r} = \frac{k_2}{I_z}(\delta_{fru} - \delta_{flu} - \delta_{bru} + \delta_{blu} - \delta_{frl} + \delta_{fll} + \delta_{brl} - \delta_{bll}) \tag{5.2.18}$$

$$\frac{k_2}{I_z} = \frac{\dot{r}}{\delta_{fru} - \delta_{flu} - \delta_{bru} + \delta_{blu} - \delta_{frl} + \delta_{fll} + \delta_{brl} - \delta_{bll}} \tag{5.2.19}$$

To find values for these parameters empirically, we fly the X8 through a variety of accelerative maneuvers while logging motor values and accelerations from the on-board IMU. We then fit the parameter values to the recorded data. The fit for k_1 is shown in Figure 5.4. The parameters determined through these fitting processes are shown in Table 5.1.

Table 5.1: Dynamic model parameters as measured on the X8.

Parameter	X8 with target ball
m	2.9 kg
l	0.1516 m
w	0.2357 m
I_x	0.3789 kg*m^2
I_y	0.06193 kg*m^2
k_2/I_z	0.00009875 hz^2
k_1	0.00922 N

5.2.2 Unscented Kalman Filter

Having derived an effective system model, we wish to combine our camera-based position estimate with the X8's GPS- and IMU-based position estimate using state-space filtering techniques. This sensor fusion problem is well-studied, and near-optimal solutions exist for many classes of systems. The Kalman filter is one of the most fundamental state space filtering techniques, providing an optimal solution for linear systems with Gaussian white measurement noise. The Kalman filter maintains an estimate of the system mean $\hat{\mathbf{x}}$ and its covariance $\mathbf{P}$. The filter operates in two steps: predict and update. In the predict step, the system mean and covariance matrix are propagated forward in time through the system model, causing the covariance to expand as the uncertainty in the system model decreases the certainty in the state estimate. In the update step, the propagated state estimate is fused with a new round of sensor measurements, reducing the covariance. The crux of the filter is the Kalman gain, which determines how heavily the existing state estimate should be weighted against the new measurements [35]. Our system is nonlinear, so the traditional Kalman filter is not applicable, but several variants of the Kalman filter have been derived for use with nonlinear systems. These are not necessarily optimal solutions to the state estimation problem, but they typically perform well.

The unscented Kalman Filter works by propagating so-called sigma points $\mathcal{X}_i$ instead of directly propagating the system covariance estimate. These are points selected from the state distribution and propagated forward such that their covariance and mean propagated forward will provide an accurate "unscented" estimate of the true system covariance and mean. The core of the algorithm is the unscented transform, which computes the mean and covariance of the sigma points after passing

through the state transition function:

$$\begin{aligned} \hat{\mathbf{x}} &= \sum_i w_i \mathcal{X}_i \\ \mathbf{P} &= \sum_i w_i (\mathcal{X}_i - \hat{\mathbf{x}})(\mathcal{X}_i - \hat{\mathbf{x}})^T. \end{aligned} \tag{5.2.20}$$

Here, each w_i is a weighting factor

$$w_i = \frac{1}{(2n + k)} \tag{5.2.21}$$

where n is the state dimensional and k is a scale factor.

Predict

In the predict step, the Unscented Kalman Filter calculates the mean and covariance of system for the time step using the process model. First, the sigma points $\mathcal{X}$ and weights $\mathbf{w} = (w_1, ...w_n)$ are calculated. The sigma points are fed through the process model:

$$\mathcal{X}_f = f(\mathcal{X}, t)) \tag{5.2.22}$$

where f is the process model.

Then, the prediction equations are applied:

$$\begin{aligned} \hat{\mathbf{x}}^- &= \sum_{i=1}^{n} w_i \mathcal{X}_{fi} \\ \mathbf{P}^- &= \sum_{i=1}^{n} w_i (\mathcal{X}_{fi} - \hat{\mathbf{x}}_i^-)(\mathcal{X}_{fi} - \hat{\mathbf{x}}_i^-)^T + \mathbf{Q}. \end{aligned} \tag{5.2.23}$$

Here, $\hat{\mathbf{x}}^-$ and $\mathbf{P}^-$ are the state estimate and covariance estimate before the update step and $\mathbf{Q}$ is the process noise matrix. In our case, we calculated $\mathbf{Q}$ through trial and error.

Update

The update step of the filter takes place in measurement space. In our case, noisy measurements from the vehicle logs and vision-based localization are fused to produce a more confident state estimate. First, the sigma points calculated in the previous

step are fed through the measurement function, $h(\mathcal{X})$:

$$\mathcal{X}_h = h(\mathcal{X}) \tag{5.2.24}$$

.

The mean and covariance of these points is then calculated:

$$\begin{aligned} \mathbf{x}_z &= \sum_{i=1}^{n} w_i \mathcal{X}_{hi} \\ \mathbf{P}_z &= \sum_{i=1}^{n} w_i (\mathcal{X}_{hi} - \mathbf{x}_i)(\mathcal{X}_{hi} - \mathbf{x}_i)^T + \mathbf{R}. \end{aligned} \tag{5.2.25}$$

Here, $\mathbf{R}$ is the measurement noise matrix. See Section 4.4.5 for details about how we calculate this matrix.

The residuals $\mathbf{y}$ between the measurement vector $\mathbf{z}$ and expected measurement $\mathbf{x}_z$ are calculated:

$$\mathbf{y} = \mathbf{z} - \mathbf{x}_z \tag{5.2.26}$$

The cross variance of the state and the measurements is given by:

$$\mathbf{P}_{xz} = \sum_{i=1}^{n} w_i (\mathcal{X}_i - \mathbf{x}_i)(\mathcal{X}_h i - \mathbf{x}_z i)^T. \tag{5.2.27}$$

. The Kalman gain $\mathbf{K}$ can then be computed:

$$\mathbf{K} = \mathbf{P}_{xz} {\mathbf{P}_\mathbf{z}}^{-1}. \tag{5.2.28}$$

Finally, the new state estimates and covariances can be computed:

$$\begin{aligned} \hat{\mathbf{x}} &= \hat{\mathbf{x}}^- + \mathbf{K}\mathbf{y} \\ \mathbf{P} &= \mathbf{P}^- - \mathbf{P}\mathbf{K}\mathbf{P}_z\mathbf{K}^T. \end{aligned} \tag{5.2.29}$$

[36]

Implementation and Results

We implemented our sensor fusion using the unscented Kalman Filter module provided in the Python `filterpy` library [37]. Specifically, we wrote code to model the

process mentioned in Section 5.2.1, and retrieve measurements from our ball-tracking and vehicle motor input logs. We then used the `update` and `predict` functions provided by `filterpy.kalman.UKF` for the relevant time steps during our formation flight to retrieve the enhanced state estimate of the observed vehicle.

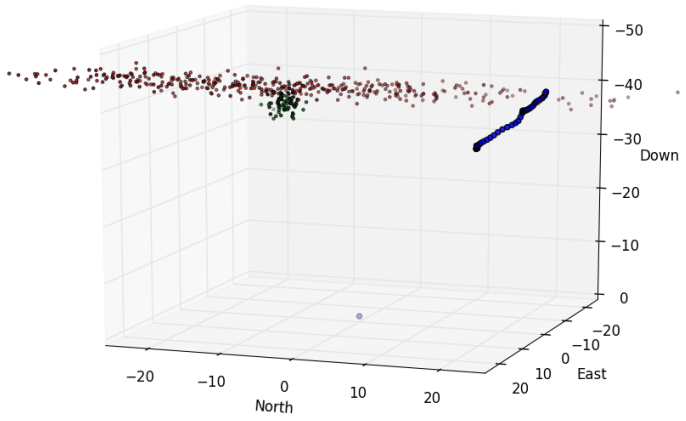


Figure 5.5: 3D plot of sensor sensor fusion probability distributions for a specific instance in time. The blue points represent the path of the observer drone. The camera estimate distribution is represented by the red points and the observed X8's distribution is shown in green. Black points represent the fused state estimate, at the intersection of the two sensor distributions.

Figure 5.5 displays the shapes of the relevant covariance distributions for a single instance in time. As noted in Section 4.4.5, the covariance of the ball location is long in the direction parallel to the C920 camera axis, but thin in directions perpendicular to this axis. The covariance of the X8's on-board state estimate is rounder, but slightly larger in the directions where the camera state estimate is very thin. The fused state lies more or less at the intersection of these two distributions, with deviations due to the weighting of the propagated state estimate and covariance, which is not show in Figure 5.5. Figure 5.6 shows the means of the various state estimates across time for a single flight. The fused state estimate is smooth due to the state propagation steps and the smoothness of the X8 state estimates, but as the X8's GPS-based estimate begins to drift off course, the fused estimate remains centered on the visual data.

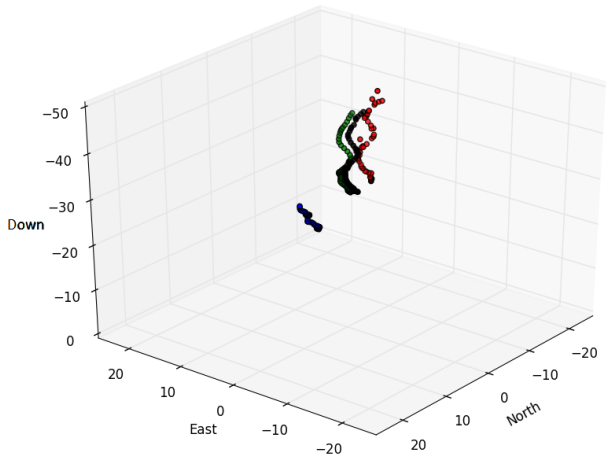


Figure 5.6: 3D plot of the X8's location estimate (green), image-based state estimate (red), and fused state estimate (black). The observer UAV's path is shown in blue in the foreground.

5.3 Camera Frame Corrections

After fusing our sensor data into an accurate state history, we apply perspective corrections to the image to undo the undesired movements and rectify the images into a comfortably-viewed stereo pair within the tolerances discussed in section 5.1. Several existing methods for this style of stereo rectification have been proposed, discussed in section 5.3.1. While effective for individual stereo pairs, none of these techniques provide persistence of movement across time, which is vital to comfortable video viewing. We discuss our modification to existing techniques to provide this stability in sections 5.3.2 and 5.3.3.

5.3.1 Existing Stereo Rectification Techniques

One way to reduce visual discomfort and enhance stereopsis from our stereo video is to transform each pair of frames such that corresponding points are aligned along the vertical axis. This technique is called stereo rectification. Stereo rectification is used in practice to reduce the complexity of correspondence matching [25]. We explored two of the most popular techniques to perform stereo image rectification, described by Hartley [38] and Fusiello et al. [39] and tried applying them to our video stabilization problem. However, we decided to move to simpler techniques described in 5.3.2 and 5.3.3. This was done for two main reasons: (1) the stereo rectification techniques unpredictably shear frames, (2) the rectification techniques do not let us specify a

target yaw and pitch for the camera perspectives. We concluded that these techniques are overly complex for our purposes and are best used for rectifying individual stereo frames to reduce the difficulty of finding correspondences. However, we provide an overview of both mentioned methods.

Epipolar Geometry

Before we provide a explanations of Hartley's and Fusiello et al.'s methods, we provide a brief overview of epipolar geometry.

Epipolar geometry describes the projective geometry between two different views of an object. It is essential for understanding stereo rectification.

The most important concept in epipolar geometry is the fundamental matrix, denoted by $\mathbf{F}$, a 3×3 matrix of rank 2. Let $\mathbf{X}$ denote a point in 3D world coordinates in $\mathbb{P}^3$ and $(\mathbf{x^1}, \mathbf{x^2})$ denote the projection of $\mathbf{X}$ to the image planes in $\mathbb{P}^2$ of the first and second cameras, respectively. The points in the image plane satisfy the relation

$$\mathbf{x^2}^T \mathbf{F} \mathbf{x^1} = \mathbf{0}. \tag{5.3.1}$$

Figure 5.7 diagrams important features in the epipolar geometry between two cameras. $\mathbf{X}$ is a point in world coordinates and $(\mathbf{x^1}, \mathbf{x^2})$ are the projections onto the image planes of the two cameras. The line segment connecting the camera apertures $\mathbf{C^1}$ and $\mathbf{C^2}$ is called the baseline. The plane π is called an epipolar plane, as is any plane containing the baseline. The points $\mathbf{p^1}$ and $\mathbf{p^2}$ are called the epipoles and are particularly important in epipolar geometry. By definition, the epipoles are the points of intersection between the baseline and image planes. The epipoles have four properties important for our purposes: (1) all pairs of matching points $(\mathbf{x^1}, \mathbf{x^2})$ exist on a line (an epipolar line) containing their image planes' respective epipoles, (2) the epipoles are the nullspace of $\mathbf{F}$ and $\mathbf{F}^T$, (3) if the image planes are parallel to the baseline, the epipoles go to infinity, (4) each epipole is the projection of the other camera's aperture. [25]

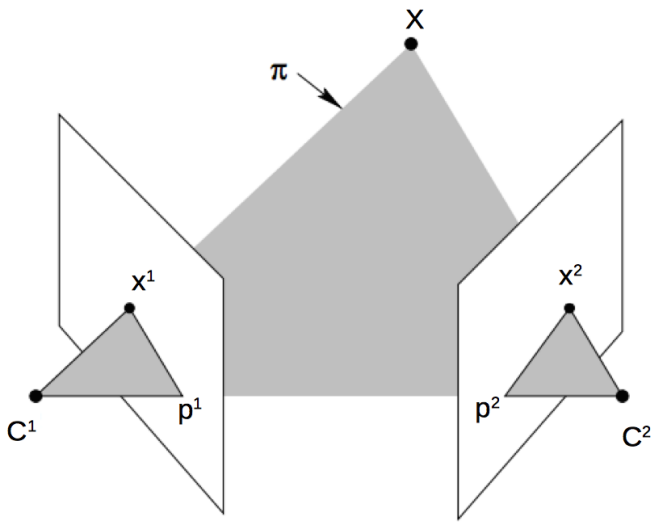


Figure 5.7: .Epipolar geometry between two image planes in a stereo setup. Adapted from [25].

Hartley

Hartley's method attempts find a homography $\mathbf{H^1}$ to warp the first camera's image plane based on the constraint that the image plane's epipole $\mathbf{p^1}$ gets mapped to infinity $(1, 0, 0)^T$. With the constraint, $\mathbf{H^1}$ still has four degrees of freedom. It is then necessary to find a matching homographic transformation $\mathbf{H^2}$ to warp the second camera's image plane. Hartley's chooses a second homography $\mathbf{H^2}$ that minimizes the least-square distance of corresponding points on the horizontal axis of the rectified pair of images.

Hartley's method works for both uncalibrated and calibrated cameras. If the camera system is uncalibrated (i.e. each camera's intrinsic matrix and the $[\mathbf{R}|\mathbf{t}]$ between the cameras is unknown), the fundamental matrix $\mathbf{F}$ must be estimated by matching keypoints (such as SIFT[13]) and using an iterative technique such as random sample consensus (RANSAC) [40] to account for keypoint outliers. Fundamental matrix estimation introduces errors and significant computational complexity, so it desirable to have a calibrated system. The epipoles can then be calculated by solving for the nullspace of $\mathbf{F}$ and $\mathbf{F^T}$. If the camera system is calibrated, the epipoles can be calculated using the fact that each epipole is the projection of the other camera's aperture:

$$\begin{aligned} \mathbf{p^1} &= \mathbf{P_1 O^2} \\ \mathbf{p^2} &= \mathbf{P_2 O^1} \end{aligned} \tag{5.3.2}$$

Here $(\mathbf{P_1}, \mathbf{P_2})$ are the camera matrices of the first and second cameras, respectively and $(\mathbf{O^2}, \mathbf{O^2})$ are the respective camera centers in $\mathbb{P}^2$. [38]

Fusiello, Trucco, and Verri

Fusiello et al.'s method produces similar results to Hartley's method but always assumes that the stereo setup is calibrated.

The algorithm is remarkably simple and consists of (1) introducing a new coordinate system with the x-axis parallel to the baseline, y-axis orthogonal to the new x and old z axis, and new z-axis orthogonal to the new x and y axes, (2) producing a new extrinsic matrix $\mathbf{R}$ based on the new coordinate system, (3) producing an adjusted pair of intrinsic camera matrices $\mathbf{K}$ that is arbitrary, (4) computing the new camera matrices using the new extrinsic rotation matrix defined by $\mathbf{R}$ and a new $\mathbf{t}$ defined as $-\mathbf{RO^i}$, where each $\mathbf{O^i}$ is the respective camera center, and (5) extracting a transformation $\mathbf{T_i}$ for each camera by multiplying the first 3 columns of the newly found camera matrices by the inverse of the old camera matrices.

Though Fusiello's method is more compact and easier to implement in practice than Hartley's, it fails remarkably when there is pure forward translation between the two image planes. [39]

5.3.2 Rotational Corrections

Figure 5.8: Raw stereo pair from a 40m baseline.

The rectification problem presented by our system is somewhat unique in that it requires a progression of well-aligned stereo pairs that also present smooth and continuous motion across time. While Fusiello's and Hartley's methods are effective for individual stereo pairs, we desire not only to rectify our stereo images to each other,

but also to the prior and subsequent images. We therefore develop a rectification scheme that allows us to designate a desired attitude for both cameras and rotate both images to align with that attitude. This is similar to Fusiello's method, but it is permitted and expected to leave a residual translational disparity between the two camera coordinate systems. We discuss methods for rectifying this translational offset in section 5.3.3.

We begin by identifying the camera projection matrix for the GoPro, as we did for the C920 in section 4.4.5. Again using Rillahan's image collection script [26], we collect a total of 80 calibration images of our checkerboard pattern using the GoPro. Our empirically calibrated camera matrix for the GoPro at full resolution is:

$$\mathbf{F} \approx \begin{bmatrix} 1653 & 0 & 936 \\ 0 & 1665 & 530 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.3.3}$$

The GoPro differs from the C920 in that it applies a significant barrel distortion to the images, causing straight lines in the image to bulge towards the edges of each frame. To remove this distortion, we can also use `cv2.calibrateCamera` to return a set of five distortion coefficients. These coefficients describe a function for mapping points from their directly projected $(x, y)^T$ pixel coordinates to distorted coordinates that account for the camera's radial and tangential distortion. This function is as follows:

$$x' = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2) \tag{5.3.4}$$

$$y' = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy \tag{5.3.5}$$

Where $r^2 = x^2 + y^2$ [19]. With the GoPro, we calculate the following values:

$$k_1 \approx -0.4838, \quad k_2 \approx 0.4303, \quad k_3 \approx -0.3120, \quad p_1 \approx -0.00088, \quad p_2 \approx 0.00879 \tag{5.3.6}$$

We can use these parameters and the opencv function `cv2.undistort()` to remove the distortion from the GoPro. This is an important step, since the standard homographic image rotations assume that the image being rotated is not heavily distorted. A sample stereo pair after undistortion is shown in figure 5.9. For comparison, the original image is shown in figure 5.8.

After removing the image's barrel distortion, we can go about applying our rota-

Figure 5.9: Stereo pair after undistortion. Straight lines in the camera frame now appear straight in the image.

tional corrections to rectify the image to our desired attitude. As described in section 2.2, a Tarot gimbal is used with each GoPro to stabilize rotations around the pitch and roll axes. The gimbal does not stabilize in the yaw direction. Our process for correcting yaw movement is mathematically similar to the process for rotating through different reference frames discussed in section 4.4.5.

We begin with a multiplication by $\mathbf{F}^{-1}$ to transform points from homogeneous image coordinates to the camera frame. Because of the gimbal, which decouples the camera's movement from that of the quadcopter body frame, we do not transform from the camera frame into the body frame as in section 4.4.5. Instead, we transform into an intermediate frame, $\mathcal{L}$, which has its x axis directed forward out of the camera lens, its y axis directed out the right side of the camera, and its z axis directed down through the bottom of the camera. The corresponding rotation matrix is:

$$\mathbf{R}_{\mathcal{C}}^{\mathcal{L}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.3.7}$$

From here, we can apply equation 4.4.19 with $\varphi = 0$ (roll is removed by the gimbal). For θ, we use the pitch of the GoPro in the gimbal. This is held constant by the gimbal, but is not necessarily zero, since the Tarot gimbal allows the user to set a desired pitch. The vehicle yaw provides the final Tait-Bryan parameter, ψ. The resulting rotation matrix, $\mathbf{R}_{\mathcal{V}}^{\mathcal{L}}$, can be inverted and used to translate points into the vehicle frame. We apply the reverse process using the desired yaw heading to transform points back into the desired camera frame. We also allow the user to set a desired pitch for each camera that remains constant throughout a video shot, since

manufacturing differences between the two gimbals produce pitch disparities between the images (visible in figure 5.9). Our final transformation is as follows:

$$\mathbf{P} = \mathbf{F}(\mathbf{R}_{\mathcal{C}}^{\mathcal{L}})^{-1}\mathbf{R}(0, \theta_{desired}, \psi_{desired})\mathbf{R}(0, \theta_{actual}, \psi_{actual})^{-1}\mathbf{R}_{\mathcal{C}}^{\mathcal{L}}\mathbf{F}^{-1} \tag{5.3.8}$$

We use the opencv function `cv2.warpPerspective` to apply $\mathbf{P}$ to all points in an image. A resulting stereo pair is shown in figure 5.10.

Figure 5.10: Stereo pair after rotational corrections.

5.3.3 Translational Corrections

Applying rotational corrections to images collected with our X8 system removes the majority of undesired movements in the image plane. However, In general, it is not possible to undo a translation applied to an image using only homographic transformations. The act of translating a camera causes objects to change location relative to each other in the image plane, and in many cases new occlusions may occur or previously occluded objects may become visible. This means that complete knowledge of a scene in three dimensions is necessary to perfectly undo general translations. We can make approximate translational corrections if we assume the scene is planar, meaning that all points in the image plane exist on the same plane when projected into the camera frame. In this case, there are no occlusions to worry about and the relationships between pixels in the image are such that the camera can be transformed with a homography.

We can define a plane in the inertial frame using

$$1 = \mathbf{k}\mathbf{X}_{inertial} \tag{5.3.9}$$

where $\mathbf{k}$ is orthogonal to the plane and has a norm of 1/(the distance to the plane from the origin). As explained by Ramadge [21], the homography to transform an image from a position with translation $\mathbf{T}_1$ in $\mathcal{I}$ to a position with translation $\mathbf{T}_2$ in $\mathcal{I}$ is

$$\mathbf{H} = \mathbf{P} - \frac{1}{1 - \mathbf{k}\mathbf{T}_1}\mathbf{F}(\mathbf{R}_{\mathcal{L}}^{\mathcal{C}}\mathbf{R}_{desired})^T(\mathbf{T}_1 - \mathbf{T}_2)(\mathbf{F}^{-T}(\mathbf{R}_{actual}^{-1}\mathbf{R}_{\mathcal{C}}^{\mathcal{L}})^T\mathbf{k})^T \tag{5.3.10}$$

Where P is defined in equation 5.3.8, $\mathbf{R}_{desired}$ is the rotation from the vehicle frame to the desired intermediate frame and $mathbf R_{actual}$ is the rotation from the vehicle frame to the actual intermediate frame (discussed in section 5.3.3).

(a) Before rectification

(b) After rectification to move the perspective 1m to the right and down

Figure 5.11: Planar image rectification

If desired, our system allows the user to designate a plane equation for which this homography will be computed and applied to the images, again using `cv2.warpPerspective`. While this technique is effective for such scenes as are shown in figure 5.11, the majority of scenes captured from a UAV in flight are not particularly flat, and so the planar image assumption does not hold. The user can tune the plane equation to stabilize those objects which are of interest, but other parts of the scene will be distorted.

Chapter 6

Conclusions and Future Work

Using our system, we are successfully able to obtain stable stereo footage viewable in 3D with novel effects, indicating that we are within the tolerances for human stereo fusion. We are also able to successfully fuse the vehicle's state estimate with state estimates calculated through vision-based techniques. However, there are several improvements that can be made to our system. Regardless, our work, especially in vision-based UAV localization, vehicle modeling, and sensor fusion for enhanced vehicle state estimation, lays the groundwork for future system enhancement.

6.1 Depth-Aware Translational Corrections

Perhaps the biggest improvement that can be made to our system is the introduction of depth-aware translation video corrections. Currently, we use a planar image approximation mentioned in 5.3.3 that is a very poor approximation for several scenes. Depth-aware translational corrections would make better use of the state estimate achieved.

Calculating depth information for every point in the image from a single stereo pair is often not feasible. Techniques that calculate dense point clouds, like Structure from Motion (SFM) [41], work best with multiple frames of the scene. Furthermore, these techniques are extremely computationally expensive.

We believe that the problem can be reduced by determining depth at only keypoints of the frame. This can be determined through stereo correspondence matching [25]. Once depth information is calculated for a reasonable number of points, we can fit a three-dimensional polygonal surface to the point cloud using a technique

such as Delaunay triangulation [42]. Each face of the polygonal surface can then be translated with an individual planar image approximation. This technique would be quite computationally expensive, but we believe efficiency can be achieved through the use of GPU programming.

6.2 Online System

An online system would provide interesting new applications, including depth-perception enhanced augmented reality and territory modeling. The ideal online system would allow a user to control the vehicles while receiving a live, stabilized, stream of stereo video.

With our platform, we are able to receive live state estimates over telemetry from a vehicle at a rate of 4Hz (as opposed to the 50Hz sampling rate available from the offline logs). If we were to implement an online system, the dynamic model described in 5.2.1 would have to be reduced to account for the low sampling rate of live sensor outputs.

An extremely high-bandwidth channel would have to be established between each vehicle and the ground control station in order to stream live HD video. We believe that free space optical communication (FSO) [43] would be the best option for this. Realistically, reasonable results can also be achieved over network connections. Furthermore, substantial computational power would be needed to stabilize the videos in real time. This can be achieved through the use of GPUs.

Bibliography

[1] 3D Robotics. 3dr pixhawk. https://store.3drobotics.com/products/3dr-pixhawk, 2015.

[2] 3D Robotics. 3dr ublox gps with compass kit. https://store.3drobotics.com/products/3dr-gps-ublox-with-compass, 2015.

[3] [Banto]. Removing the hinged base from the logitech c920 webcam [video file]. https://www.youtube.com/watch?v=a39iWgSwaBk, 2012.

[4] Michael Darling. How to achieve 30 fps with beaglebone black, opencv, and logitech c920 webcam. http://blog.lemoneerlabs.com/3rdParty/Darling_BBB_30fps_DRAFT.html, 2013.

[5] Derek Molloy. Beaglebone: Video capture and image processing on embedded linux using opencv [video file]. http://www.youtube.com/watch?v=8QouvYMfmQo, 2013.

[6] Lorenz Meier, JF Camacho, B Godbolt, J Goppert, L Heng, M Lizarraga, et al. Mavlink: Micro air vehicle communication protocol. http://qgroundcontrol.org/mavlink/start, 2013.

[7] Kevin Hester. Dronekit. http://dronekit.io/, 2015.

[8] Andrew Tridgell and Stephen Dade. Mavproxy: A uav ground station software package for mavlink based systems. http://tridge.github.io/MAVProxy/, 2013.

[9] Andrew Fuller. Arduino laser tape measure. http://blog.qartis.com/arduino-laser-distance-meter/, 2013.

[10] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[11] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[12] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A Efros. Ensemble of exemplar-svms for object detection and beyond. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 89–96. IEEE, 2011.

[13] Tony Lindeberg. Scale invariant feature transform. *Scholarpedia*, 7(5):10491, 2012.

[14] John Illingworth and Josef Kittler. The adaptive hough transform. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (5):690–698, 1987.

[15] Hong Liu, Yueliang Qian, and Shouxun Lin. Detecting persons using hough circle transform in surveillance video. In *VISAPP (2)*, pages 267–270, 2010.

[16] Xinguo Yu, Changsheng Xu, Hon Wai Leong, Qi Tian, Qing Tang, and Kong Wah Wan. Trajectory-based ball detection and tracking with applications to semantic analysis of broadcast soccer video. In *Proceedings of the eleventh ACM international conference on Multimedia*, pages 11–20. ACM, 2003.

[17] HK Yuen, John Princen, John Illingworth, and Josef Kittler. Comparative study of hough transform methods for circle finding. *Image and vision computing*, 8(1):71–77, 1990.

[18] Michael Isard and Andrew Blake. Condensation - conditional density propagation for visual tracking. *International journal of computer vision*, 29(1):5–28, 1998.

[19] G. Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Peter Ramadge. Ele 488: Image processing. University Course, Spring 2015.

[22] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[23] Gary Bradski and Adrian Kaehler. *Learning Opencv, 1st Edition*. O'Reilly Media, Inc., first edition, 2008.

[24] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing (3rd Ed.): Principles, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[25] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[26] Chris Rillahan. Gopro lens distortion removal. `http://www.theeminentcodfish.com/gopro-calibration/`, 2015.

[27] Randal W Beard. Quadrotor dynamics and control. *Brigham Young University*, 2008.

[28] Young Lim Lee and Jeffrey A Saunders. Stereo improves 3d shape discrimination even when rich monocular shape cues are available. *Journal of vision*, 11(9):6, 2011.

[29] J Park, H Oh, S Lee, and AC Bovik. 3d visual discomfort predictor: Analysis of disparity and neural activity statistics. 2014.

[30] Yei-Yu Yeh and Louis D Silverstein. Limits of fusion and depth judgment in stereoscopic color displays. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 32(1):45–60, 1990.

[31] Robert Graham. Extraocular muscle actions. `http://emedicine.medscape.com/article/1189759-overview`, 2013.

[32] John A Pratt-Johnson. Central disruption of fusional amplitude. *The British journal of ophthalmology*, 57(5):347, 1973.

[33] Sarah Tang. Vision-based control for autonomous quadrotor uavs. Undergraduate thesis, Princeton University, 2013.

[34] Patrick Bouffard. On-board model predictive control of a quadrotor helicopter: Design, implementation, and experiments. Technical report, DTIC Document, 2012.

[35] Robert F Stengel. *Optimal control and estimation*. Courier Corporation, 1994.

[36] Eric A Wan and Rudolph Van Der Merwe. The unscented kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158. IEEE, 2000.

[37] Roger Labbe. *Kalman and Bayesian Filters in Python*. First edition, 2013.

[38] Richard I Hartley. Theory and practice of projective rectification. *International Journal of Computer Vision*, 35(2):115–127, 1999.

[39] Andrea Fusiello, Emanuele Trucco, and Alessandro Verri. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications*, 12(1):16–22, 2000.

[40] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[41] Jan J Koenderink, Andrea J Van Doorn, et al. Affine structure from motion. *JOSA A*, 8(2):377–385, 1991.

[42] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[43] Vincent WS Chan. Free-space optical communications. *Lightwave Technology, Journal of*, 24(12):4750–4762, 2006.

Appendix A

Code

Since our code is quite lengthy, we provide a link to a github respository containing all of our relevant code files: `https://github.com/agola11/SeniorThesis2015`.

The `copter_control` directory contains files relevant to our control scheme described in Chapter 3. Here, the most important files are the server and client implementations, `sock_server.py` and `copter_client.py` respectively.

The `ball_tracker` directory contains code relevant to our ball-tracking algorithm described in Chapter 4. The important files are our IIR filter module `iir_filter.py`, ball-tracker module `ball_tracker.py` (which contains majority of the logic), and the files under the `\svm` directory for model training and testing.

Finally, the `stereo_rectify` folder contains files relevant to our post-processing pipeline described in Chapter 5. Here, the files `ukf.py` show our sensor fusion logic, while the `*_reader.py` and `*_rectify.py` classes show logic for reading logs and video stabilization, respectively.