AUTONOMOUS CLOSE FORMATION FLIGHT OF SMALL UAVS USING VISION-BASED LOCALIZATION

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Michael B. Darling

May 2014

COMMITTEE MEMBERSHIP

TITLE:	Autonomous Close Formation Flight of Small UAVs Using Vision-Based Localization
AUTHOR:	Michael B. Darling
DATE SUBMITTED:	May 2014
COMMITTEE CHAIR:	Eric Mehiel, Ph.D., Associate Professor, Aerospace Engineering Department
COMMITTEE MEMBER:	Rob McDonald, Ph.D., Associate Professor, Aerospace Engineering Department
COMMITTEE MEMBER:	Lynne Slivovsky, Ph.D., Professor, Electrical Engineering Department
COMMITTEE MEMBER:	Nicholas Brake, M.S., Staff Aeronautical Engineer, Lockheed Martin

ABSTRACT

Autonomous Close Formation Flight of Small UAVs Using Vision-Based Localization

Michael B. Darling

As Unmanned Aerial Vehicles (UAVs) are integrated into the national airspace to comply with the 2012 Federal Aviation Administration Reauthorization Act, new civilian uses for robotic aircraft will come about in addition to the more obvious military applications. One particular area of interest for UAV development is the autonomous cooperative control of multiple UAVs. In this thesis, a decentralized leader-follower control strategy is designed, implemented, and tested from the follower's perspective using vision-based localization.

The tasks of localization and control were carried out with separate processing hardware dedicated to each task. First, software was written to estimate the relative state of a lead UAV in real-time from video captured by a camera on-board the following UAV. The software, written using *OpenCV* computer vision libraries and executed on an embedded single-board computer, uses the Efficient Perspective-n-Point algorithm to compute the 3-D pose from a set of 2-D image points. High-intensity, red, light emitting diodes (LEDs) were affixed to specific locations on the lead aircraft's airframe to simplify the task if extracting the 2-D image points from video. Next, the following vehicle was controlled by modifying a commercially available, open source, waypoint-guided autopilot to navigate using the relative state vector provided by the vision software. A custom Hardware-In-Loop (HIL) simulation station was set up and used to derive the required localization update rate for various flight patterns and levels of atmospheric turbulence. HIL simulation showed that it should be possible to maintain formation, with a vehicle separation of 50 ± 6 feet and localization estimates updated at 10 Hz, for a range of flight conditions. Finally, the system was implemented into low-cost remote controlled aircraft and flight tested to demonstrate formation convergence to 65.5 ± 15 feet of separation.

ACKNOWLEDGMENTS

First, I would like to thank Dr. Eric Mehiel for serving as my adviser and guiding my efforts throughout this project. I would also like to acknowledge the other members of my committee, Dr. Lynne Slivovsky, and Nicholas Brake, with special thanks to Dr. Rob McDonald for graciously providing me access to the Unmanned Aerial Systems Laboratory. Only with the resources of the UAS Lab was it possible for this work to be realized in a tangible form. Also, thanks to Kendra Bubert for skillfully handling a number of purchasing headaches that could have otherwise brought this thesis to a standstill.

Since this project makes use of many "open-source" components, I naturally found a great deal of help from the online open source community. Of most help was Matthew Witherwax, to whom I would like to express my deepest gratitude for helping me to solve the most difficult hardware challenge I encountered during the course of this thesis. Others providing notable assistance included Craig Elder of 3D Robotics, Zouhair Mahboubi, and Cal Poly alumnus Alexander Corcoran. Thank you all.

I appreciate the time and expertise provided by my volunteer pilots Clint Wachter, Martin Bialy, Brad Schab, and Robbie Campbell, without whom flight testing would not have been possible. To my cohorts, Christian Lopez, Adam Chase, Brian Marchini, Alex Gary, and Pat Meyer, thank you for your occasional technical guidance, your day-to-day support, and most of all your friendship.

I am also indebted to those who contributed to my successes from outside the lab—both those who have been part of my life for many years and those who I have had the pleasure of befriending more recently over the course of my graduate studies. Thank you for your encouragement, humor, and helping me to navigate the personal challenges I encountered along my journey. Without you, I wouldn't have made it out with my sanity (relatively) intact.

Finally, I would like to thank my family for their continued and loving support. For my sister, Shannon, and her son, Zachary, who have provided me with the constant reminder to not take everything in life too seriously. I am also grateful for my parents Wally and

Debbie, who have supported me financially and given me the rare opportunity of beginning my professional career without the burden of student debt. I only hope that I can put such a precious gift to its best possible use. More importantly, my parents have had a tremendous impact on my life by shaping me into the person I am today. For that, I am eternally grateful.

– Michael B. Darling

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 The Demand for Formation Flight-Capable UAS

In the past decade, unmanned aerial vehicles (UAV) have seen rapid increases in use for both military and civilian applications. The majority of advancements in UAV technology have come about through the heavy use of unmanned aerial systems (UAS) by the United States Armed Forces for intelligence, surveillance, and reconnaissance (ISR) in the Middle East. Despite recent cuts to the U.S. defense budget, UAVs persist as a high priority asset to the Department of Defense due to their proven success. Forecasts predict that total global expenditures on unmanned aircraft systems will increase from $6.6 billion in 2013 to $11.4 billion in 2022. [1] A fraction of this growth is due to the realization of new, nonmilitary applications for UAVs such as disaster relief, search and rescue, monitoring of weather or wildlife, terrain mapping, crop dusting, commercial transport, or use by local police or Border Patrol. The Federal Aviation Administration (FAA) has recognized the potential applications of domestic UAVs and is currently working to develop a set of regulations that will allow unmanned aircraft to be integrated into the national airspace by 2015 as required by the 2012 FAA Reauthorization Act. [2]

If UAVs are to be integrated into the current fleet of military aircraft or the national airspace, they must be capable of interacting with other aircraft—manned and unmanned, alike. For military aircraft, this might mean that a UAV should be capable of carrying out cooperative missions. For example, the Request for Information (RFI) recently released by the U.S. Navy for an Unmanned Carrier Launched Airborne Surveillance and Strike

(UCLASS) system requires compatibility with the current Navy infrastructure. [3] The RFI demands the recommended system to have the ability to operate onboard the deck of an aircraft carrier in the presence of other aircraft and the deck crew, receive fuel from U.S. Navy and U.S. Air Force style airborne tankers, and operate within the National Airspace System. [3] Likewise, civil UAVs operating in FAA controlled airspace will be required to have sense-and-avoid capability to prevent midair collisions between manned and unmanned aircraft. In either scenario, the capability for a UAV to localize nearby aircraft is of the utmost importance for UAS integration and has become a major area of UAV research.

The formation flight of unmanned vehicles is a topic that has also gained significant attention as a means of extending the capability of UAVs. Flying in a V-formation has been shown to result in fuel savings of up to 18% by taking advantage of the upwash generated by the tip vortices of the aircraft ahead. [4] Since the endurance of a UAV is not inhibited by the physical needs of a human pilot, range and on-station time can be drastically increased with automated aerial refueling (AAR) capability, limited only by the need to land for repairs and maintenance. Cooperative tasks such as search and rescue, terrain mapping, communications relaying, and border patrol could all take advantage of formation flight to efficiently cover large areas. Automated formation flight would also allow multiple aircraft to be controlled by a single operator, which could be especially advantageous for reducing operating costs associated with cargo transport missions. Finally, some have proposed the application of formation flight as a means of managing air traffic near busy airports. [5]

1.2 Types of Formation Flight

1.2.1 Control Approaches

There are three primary approaches to handling multiple-vehicle control: *centralized*, *distributed*, and *decentralized control*. [6] Each method varies by how information is shared between the vehicles in the formation, and how controller processing is divided across each of the members. While this thesis focuses on the decentralized control approach, all three methods are briefly presented here for completeness.

Centralized control requires all of the formation members to communicate with a single centralized controller. The centralized controller may be onboard a designated vehicle in the formation, but is most often separate from the formation and kept in a static location, such as a Ground Control Station (GCS). Centralized control offers the best performance, but requires heavy communication and has a single point of failure, which severely limits its application outside of academic research.

Distributed control divides the control task evenly across each of the vehicles but still uses some communication between the members to share state information. This approach provides moderate performance and requires less-heavy communication compared to a centralized approach. Another feature of distributed control is that it makes the system more robust by eliminating the single point failure of a centralized control scheme. The formation could be easily designed to continue on towards its goal, even if a vehicle is lost from the formation.

Decentralized control, which is the focus of this thesis, seeks to make the members completely independent by eliminating communication entirely. This requires each vehicle to be individually capable of sensing the state of its neighbors, and to carry out the necessary control processing with onboard processors. While decentralized control generally has the poorest performance of these three schemes, it has some advantages that make it appealing for many real-world applications. Perhaps the most obvious is the need for stealth, which requires communication blackout to go undetected. Also, decentralized control inherently makes the system robust to GPS lost-link situations. Since GPS cannot be used for relative navigation in the absence of a communications link, other sensors would be required for localization, thereby eliminating the formation's dependency on having a stable GPS signal.

1.2.2 Formation Strategies

There are many different formation strategies, however three appear most frequently in literature: *leader-follower*, *virtual leader*, and *behavioral* approaches. [7] These schemes each use a different set of rules to hold formation with multiple vehicles. This thesis investigates

the formation flight of only two vehicles, so naturally, a leader-follower approach is used. However, the ultimate goal of vision-based localization is to achieve decentralized formation flight with multiple UAVs, so all three strategies should be addressed.

In the *leader-follower* approach, each "following" vehicle references a "leader". This can be done in either of two ways, called "leader-mode" and "front-mode": [6]

1. In leader-mode, each follower directly references a single leader that is common to all followers of the formation

2. In front-mode, each follower references the vehicle ahead and nearest to it (its "local leader") regardless of whether or not its local leader is the global leader of the formation or another follower. This relationship is cascaded through the followers up to the leader, which is responsible for guiding the formation.

The leader-follower approach is the simplest of the three formation strategies and when operating in front-mode, can adapt to lost members—including the loss of the global leader. The front-mode does, however, slow the transient response of the formation due to the cascading effect of controller error propagation.

The *virtual leader* strategy works similarly to the leader-mode of the leader-follower approach discussed above, except all vehicles in the formation act as followers. A point in space is defined as the "virtual leader" to which all members directly reference. The formation behavior is guided by dynamically updating the position of the virtual leader. By using a virtual point in space as the leader rather than an actual vehicle, the formation is made robust to the loss of a global leader. This approach also avoids the delayed transient response due to error propagation since each vehicle is directly referencing the universal leader. However, due to a lack of feedback between the individual aircraft, collision avoidance between formation members relies solely on the ability of all the aircraft to hold formation within tolerance.

The *behavioral* approach looks to the natural behavior of migratory birds for inspiration of a scheme that allows each vehicle to "sense" the rest of the formation. [8] All aircraft in the

formation reference a point in space called the "Formation Geometry Center" (FGC) that dynamically changes with the relative distances between the aircraft. The FGC is defined by integrating a set of differential equations that relate the FGC velocity to the velocities of the aircraft in the formation. If a member falls out of formation, the FGC shifts relative to the formation and is sensed by the other members. The formation momentarily deviates from the desired flight path to allow the strayed aircraft to rejoin the formation, and then continues as a single formation towards the goal.

1.3 Advantages of Vision-Based Localization

Although there are many sensors capable of providing localization for precision relative navigation, machine vision is by far the most promising and mature technology currently under consideration. Sensor selection for formation flight is closely tied to the"sense-and-avoid" problem, for which the FAA requires "a method that provides an equivalent level of safety, comparable to sense-and-avoid requirements for manned aircraft." [9] The FAA also cites "radar observation, *forward or side looking cameras*, electronic detection systems, visual observation from one or more ground sites, monitor[ing] by patrol or chase aircraft or a combination thereof" as suitable methods of achieving such capability. In Reference [10], Karhoff et. al. evaluate potential sensor types for integration on the General Atomics MQ-1 Warrior UAV for sense-and-avoid capability so that it may be safely operated in National Airspace under Federal Aviation Regulations (FARs). Of the 13 configurations considered, visual sensors were found to be the best available technology based on their size, cost, technological maturity, bandwidth, required power, and weight.

Electro-optical sensors are appealing for close proximity formation flight for a number of other reasons, including the need for low observability in enemy airspace. Since EO sensors are passive, they can be operated without being detected, and cannot be jammed like radar or other active sensing technologies. In an automated aerial refueling scenario, EO sensors do not emit any kind of radiation that could raise potential health concerns for the boom operator of the tanker aircraft. Additionally, the use of cameras for localization eliminates

the need for any inter-vehicle communication or a stable GPS link. Machine vision is also well suited for the real-time, high bandwidth sensing that is required for close proximity formation flight, granted enough processing power is available.

For small-scale UAVs, research almost exclusively focuses on EO sensors due to the extremely restrictive size, weight, and power requirements of small aircraft. Camera technology has progressed rapidly in recent years due to the advancement of cellular telephones. Today, cameras are extremely small and lightweight, have low power consumption, provide relatively high quality images, and can be acquired at very little cost. [9] As discussed in Section 2.2, cameras can be calibrated to negate the effects of lens distortion in order to produce high-quality localization estimates even with inexpensive equipment. The only drawback to using vision-based localization on small UAVs is the need for a fast, low-power processor with a small form factor to be run onboard the vehicle. (Transmitting video data over a communications link would be unfeasible due to the high bandwidth required, long transmission delay, and potential need to maintain low observability.) Fortunately, embedded computers have also greatly diminished in size and cost. Today, small prototyping boards having a footprint just larger than a credit card are available with up to 1 GHz of processing power for under $50. This combination of small, low-power, high quality, and commercially available cameras and embedded computers give machine vision great utility as a method of accomplishing situational awareness on small UAVs.

1.4 Related Multiple-UAV Formation Work

Of the vast body of UAV research, a sizable portion considers problems associated with multi-UAV navigation. For example, [11] provides a brief summary of current research into *cooperative UAV mapping* of large areas. In [12] a *collision-avoidance* algorithm is developed for UAVs using a single vision sensor. Due to the pressing need for AAR capable UAVs, perhaps the fastest growing area of multi-UAV research is aimed at *autonomous formation flight*.

Most existing work in the area of formation flight focuses on either the sensing tech-

nology itself, or aims to develop the control algorithms under the assumption that relative state information is already available. For example, in [13] the VisNav sensor system is designed for probe-and-drogue style automated aerial refueling applications and essentially works as an "intelligent" optical sensor by detecting structured beacons of light emitted from the drogue basket. In [14], research is even more focused to present a feature extraction method for electro-optical-based systems using Harris edge detection and the global nearest neighbor algorithm for data association. On the other hand, some works only seek to address the control laws, as in [15] which presents an LQR controller for AAR docking under the presumption that relative localization is already accomplished by some kind of machine vision sensor. Often times, this sort of work is carried out through the simulation phase while hardware implementation and demonstration is left to others in the form of "future work". Only a small subset of the current research addresses both aspects together: sensing *and* the control system design for autonomous formation flight. In most cases, the scope of the work is only opened up when implementation and flight test is the ultimate goal of the project.

There have been a small number of full-scale demonstrations such as in [16], where an autonomous close formation flight controller was flight tested aboard a Learjet LJ-25 following behind a USAF C-12 using differential GPS. More recently, as part of DARPA's $33 million Autonomous High Altitude Refueling program, Northrop Grumman retrofitted two Global Hawk UAVs to autonomously refuel in midair by combining differential GPS with machine vision.

However, without the full backing of the U.S. military, full-scale implementation is often not possible due to the level of infrastructure and high costs involved. For this reason, most of the body of academic research targets development on smaller, lower-cost radio controlled (RC) aircraft for proof-of-concept demonstration. Though RC aircraft have fewer barriers to entry than their full-scale counterparts, they pose some unique challenges. Due to their small size, RC aircraft have a severely limited payload capacity for carrying additional sensors and avionics. This limitation makes small, lightweight, low-power, sensing and processing hardware a "must" for implementing advanced control techniques onto such small

vehicles. For example, in [17] structured Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) are investigated as potential solutions for computationally intensive tasks such as video processing onboard small UAVs. As hardware continues to evolve beyond the capability of present-day single-board computers (SBCs), increasingly complex processing techniques will become possible onboard these small UAVs.

In recent years, University of Pennsylvania's GRASP laboratory and MIT's ACL program have both gained attention for their research into the cooperative control of quadcopters. While the level of control is indeed impressive, both groups use a complex indoor motion capture system to provide extremely accurate situational awareness from the perspective of a global observer, allowing the sensors and processing hardware to be moved off of the UAVs. The problem with this approach is that it has severely limited application in the real-world, where the environment is not so easily controlled.

In the following sections, a few projects having strong similarity to this thesis are described in detail. All of these projects sought to demonstrate autonomous formation flight of small UAVs. Each example uses a slightly different approach and showed varying levels of success. In some cases, demonstration through flight testing was never accomplished due to time constraints.

1.4.1 Formation Flight controller for Multiple Small UAVs

Of the works surveyed in preparation for this thesis, one of the most straightforward approaches to leader-follower formation flight comes from the Air Force Institute of Technology's Advanced Navigation Technology Center in the form of a master's thesis. [18] The author, McCarthy, developed a formation flight system to be used in a "wrap-around" manner with the closed-source Piccolo II waypoint-guided autopilot.

McCarthy's system works by remotely processing the navigation commands for the follower from the ground. A GCS unit provided by the autopilot's manufacturer receives telemetry and control data from both UAVs at 1 Hz then sends it to a separate laptop computer over an RS-232 serial connection. The laptop, running custom operator interface

software written using the Piccolo's Communication Software Development Kit, reads the telemetry and control data from the GCS unit. Then, the formation flight algorithm is applied to compute an updated waypoint and throttle command for the following aircraft. This information is then sent back to the GCS unit and added to the queue for transmission to the following UAV. Once the follower receives the updated commands, its autopilot operates the servos accordingly.

Although this dynamic updating of waypoints provides a "wrap-around" solution to controlling a closed-source autopilot it poses some obvious problems, namely a large time delay. Though McCarthy never reached his goal of flight testing the system due to time constraints, he was able to asses performance through Hardware-in-Loop simulation. McCarthy found that there was a 4-6 second delay in the follower's ability to react to the leader's movements. He estimated that with this kind of delay, the minimum separation distance that would give the follower enough time to safely react to avoid collision with the leader would be approximately 210 feet.

1.4.2 UAV Formation Flight Work at Cal Poly

Over the course of the past few years, a number of projects dealing with small UAVs have come through the Cal Poly Aerospace Engineering Department. Of particular interest to this thesis are those projects focusing on situational awareness, formation flight, or using similar hardware. In 2010, Shane Wallace submitted his master's thesis on a UAV terrain avoidance system using potential function guidance. In 2013, Christian Lopez completed his thesis, which built upon the simulation work of Cal Poly alumnus, Masamitsu Tsuruta, by implementing potential function guidance into small UAVs equipped with the open source *ArduPilot Mega* autopilot board and demonstrating the ability to fly in formation through flight testing. Concurrent with this thesis, another student, Brian Marchini, has demonstrated transition-to-hover flight using adaptive control with the *ArduPilot Mega*. Another concurrent thesis by Cory Hackett-Robles aims to develop a *real-time* hardware-in-loop simulator for the *ArduPilot Mega*.

Although none of these projects use vision for localization or the same leader-follower scheme for guidance and navigation, they have all been invaluable resources to this thesis in the form of lessons-learned and a support base for hardware troubleshooting. Of most use has been Christain Lopez's thesis [19], from which this project derives some baseline requirements.

Using the *ArduPilot Mega* and other open source hardware, Christain developed an outer-loop formation flight controller based on a virtual waypoint implementation of a potential function guidance algorithm. The project required the introduction of an inter-UAV communication network, Hardware-In-Loop simulation, and a flight testing phase. Christian's formation flight controller successfully used GPS for localization, but also brought some of the shortfalls of using GPS to light. During flight testing, Christian encountered poor GPS accuracy and lost-link situations that caused the formation to fail. In the end, Christian was able to demonstrate formation convergence in a leader-follower scenario to 115 ± 16 feet.

1.4.3 Camera Based Localization for Autonomous UAV Formation Flight

By far, the most helpful published work for this thesis was a project that was done by a group of five students at Stanford University. [20] The project sought to achieve close formation flight of two small UAVs using computer vision for localization. The scale of the two RC airplanes ($\approx$2 meters) can be seen in Figure 1.1.

To achieve in-flight localization, the team affixed five "high-intensity" red LEDs to known positions on the lead aircraft's airframe to contrast with the background of the sky, and mounted a camera to the following aircraft to capture video of the leader in real-time. A sample video frame taken from the following aircraft can be seen in Figure 1.2. The video frames were then processed on a 1.6 GHz fit-PC2 fanless x86 computer running Linux. The seven brightest areas in the red channel of the image were taken as potential image points for the LEDs and passed to an Orthogonal Iteration algorithm to compute the relative position and orientation of the lead aircraft with respect to the follower. Since

Figure 1.1: Two RC aircraft used by students at Stanford University

the Orthogonal Iteration algorithm requires knowledge of which image point corresponds to *which* LED, all feasible combinations were passed through the algorithm and the solution with the lowest reprojection error was used. The estimate from vision was then fused with GPS and other sensor data using an Extended Kalman Filter. With this combination of hardware and software, the team was able to accomplish localization updates at a frame rate of up to 25 Hz.

For control, the team developed a model-based LQG controller to command the leader to fly a circle pattern with as large of a radius as possible while staying within line of sight. The follower used a similar control implementation, but adjusts its speed, altitude, and position relative to the circle based on the relative navigation solution computed by the vision subsystem. The controller code was processed on the same fit-PC2 computer running a version of the open source *Paparazzi* [21] autopilot software. Hardware-in-Loop simulation was used to test the circle tracking algorithm implemented on the leader, but no multi-aircraft simulations were carried out.

In flight tests, the team was only able to demonstrate the ability to fly in formation using GPS for localization. (They were *not* able to use localization estimates from the

Figure 1.2: View from camera on the trailing aircraft

vision system to control the trailing aircraft because the Extended Kalman Filter used to fuse vision estimates with other sensors had not been "completely vetted".) However, some flights provided samples of video with the leader in frame for periods of up to 5 minutes. The team was able to localize the UAV from the video samples by applying the vision algorithms discussed above, but were ultimately unable to validate the experimental localization estimates due to a lack of a ground truth measurement system. The article published by the group also mentioned that wind disturbances, noisy sensor measurements, and poor GPS altitude accuracy made it difficult to fly steady, repeatable circles.

1.5 Objective of Thesis

This primary aim of this thesis to expand upon previous small UAV work carried out at Cal Poly by designing, implementing, and demonstrating a decentralized leader-follower formation flight control system using vision-based sensing technologies. In order to achieve this primary goal, the following project milestones must be completed:

- Design the system architecture, and select and acquire system components (hardware

and software)

- Develop software required for vision-based localization, including the ability to:
  - Detect and extract features from video imagery
  - Reliably estimate the relative state of a lead UAV with respect to a following UAV from extracted feature points
  - Run in real-time on embedded hardware
- Modify existing autopilot software to add capability for relative navigation
- Develop a method of simulating relative navigation-based formation flights
- Demonstrate successful formation flight through flight testing

Small aircraft are notoriously difficult to obtain accurate dynamic models for because they rarely fly in steady conditions and are easily perturbed even by slight atmospheric disturbances. In order to limit the scope of an already multifaceted project, this thesis makes no attempt to dynamically model the aircraft being controlled which is a problem worthy of a master's thesis in itself. Without a dynamic model of the system, no performance goals could be derived for this project. The best performance target comes from Christian Lopez's own work. It should be possible to improve upon the 115 ± 16 foot formation convergence achieved by Christian due to the higher accuracy and bandwidth of vision-based sensing compared to GPS for relative navigation.

Of secondary importance, this thesis also serves to contribute to the development of UAV projects at Cal Poly by developing a baseline vision system. Future students interested in camera-based obstacle avoidance, camera-based target tracking, or other vision applications for small robotic vehicles may find the baseline system presented here to be a useful starting point from which to begin their own research. Additionally, this project makes heavy use of open source hardware and software, making it a worthwhile opportunity to explore the feasibility of using open source technology in an academic context. Though academic researchers have used open source software for years, open source *hardware* is a relatively new concept that has grown in popularity over the the past decade.

1.6 Organization of Thesis

This work is divided into 9 chapters. Following this introductory chapter is Chapter 2, which presents background information on the reference frames used in this thesis and a mathematical explanation of the EPnP algorithm at the core of the vision software. Chapter 3 argues the case for the open source concept and describes the hardware and software components selected for this project. Chapters 4 and 5 describe how the feature detection and 3-D state estimation components of the computer vision software work, respectively. Chapter 6 explains the controller architecture of the standard autopilot software and goes on to describe how the software was modified to include relative navigation capability to satisfy the needs of this thesis. Chapter 7 introduces the hardware-in-loop simulation tools that were developed and used and presents a summary of results from simulated flight tests. Chapter 8 documents the results from real-world flight tests of the system in its final form. Finally, Chapter 9 summarizes the lessons learned from this experience and provides recommendations for potential improvements and/or future work. For reference, a list of symbols and abbreviations is provided in Appendix A.

For those who may wish to expand upon this work, other potentially useful information can be found in Appendices B to E. The appendices contain drawings for various mounting hardware, trade studies on camera resolution and focal angle, a flight test manual and preflight checklist, and a document describing the steps necessary to configure the hardware used in this thesis for capturing from a USB webcam at a suitably high frame rate.

Chapter 2

Background

2.1 Reference Frames and Coordinate Transformations

Relative navigation depends on the ability to accurately solve for the 3-D position and orientation of one vehicle with respect to another. Since each aircraft has its own body frame, sensing occurs in the camera frame, and navigation occurs in the formation frame, it is important that coordinate transformations between these frames are handled very carefully. This section will define each of the reference frames used by the control implementation presented in this thesis as well as the general method for converting between reference frames.

2.1.1 Body Frames

Each aircraft flying in formation has its own body frame of reference with its origin at the center of mass. The x-axis, x_b, is directed along the longitudinal axis through the nose, y_b is positive pointing out the right wing, and z_b points through the belly, as shown in Figure 2.1. The body frame moves and rotates with the aircraft during flight to stay aligned with the airframe by these conventions. The body frame can be found by rotating the local North-East-Down (NED) frame about z_{NED} by the heading (ψ) angle, y_{NED} by the pitch (θ) angle, and x_{NED} by the roll (ϕ) angle, in that order.

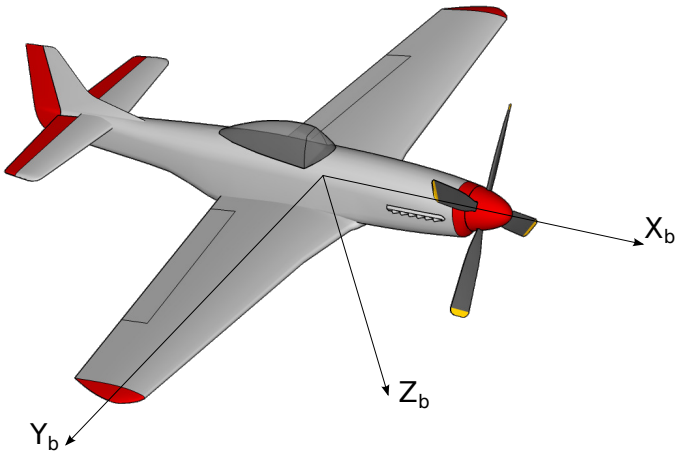


Figure 2.1: Body frame definition

2.1.2 Camera Frame

The camera frame is similar to the body frames of the leader and follower aircraft except that it is attached to the camera body. Although the camera frame is roughly aligned with the body axes of the follower aircraft, its origin does not coincide with the body frame. More importantly, any angular misalignment with the body axes due to mounting error could contribute to the overall relative localization error. For this reason, a separate camera frame must be defined and calibrated so that the localization estimate can be related from camera coordinates to body frame coordinates.

The camera frame is centered at the principal point (where the optical axis intersects the image plane of the camera). The x-axis, x_c, points out the lens of the camera along the optical axis, y_c points out the right side of the camera, and z_c points out the bottom. Like the body frames, the camera frame maintains fixed to the camera as it moves in 3-D space. The camera frame can be found by rotation through the relative Euler angles with respect to the body axes $(\phi_{c/b}, \theta_{c/b}, \psi_{c/b})$. The camera frame is pictured in Figure 2.2.

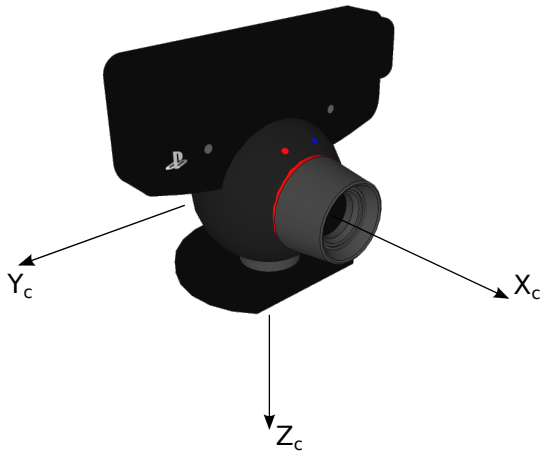


Figure 2.2: Camera frame definition

2.1.3 Formation Frame

For navigation, the formation frame was chosen as the preferred control frame because it minimized the number of modifications that had to be made to the existing autopilot firmware. Since the standard autopilot uses a path-to-bank PID scheme with a commanded bearing error as feedback for lateral control, it was relatively straightforward to dynamically compute a new bearing error based on the relative position of the leader.

The formation frame is similar to the North-East-Down frame, but is rotated through the heading angle, ψ, of the follower. The x-axis, x_f is in the direction of the nose of the following aircraft and perpendicular to z_f, which points straight down from the center of mass of the follower to the center of the Earth. The remaining axis, y_f, is orthogonal to the other two axes and roughly points in the direction of the right wingtip. Note that, unlike in [22], the "formation frame" used here is fixed at the *follower's* center of mass and rotated about its heading angle instead of being attached to the leader. This convention was chosen due to the larger separation distance between the aircraft in formation relative to the size of their airframes.

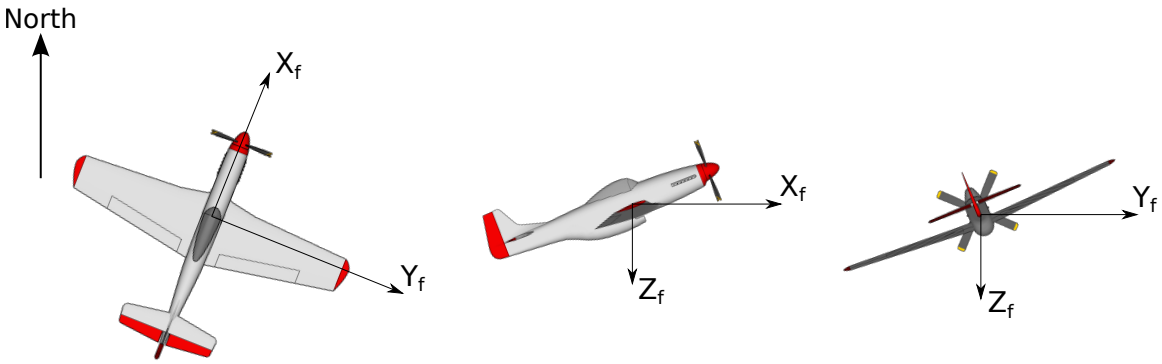


Figure 2.3: Formation frame definition

2.1.4 General Coordinate Transformation

In the most general sense, transforming between frames requires both a rotation and a translation. We consider two frames—Frame 1 and Frame 2—each with a unique origin and orientation. If we begin with a point expressed in terms of Frame 1 coordinates and wish to transform it into Frame 2 coordinates, we can apply the following equation

$$\vec{x}_2 = \mathbf{C}_1^2 \cdot \vec{x}_1 + t_{1/2,2}, \tag{2.1}$$

where x_1 and x_2 are the point expressed in Frame 1 and Frame 2 coordinates, respectively, $\mathbf{C}_1^2$ is the rotation matrix that rotates Frame 1 through $(\phi_{2/1}, \theta_{2/1}, \psi_{2/1})$ into Frame 2, and $t_{1/2,2}$ is the translation vector of Frame 1 relative to Frame 2 in Frame 2 coordinates. The rotation matrix can be expanded in shorthand as

$$\mathbf{C}_1^2 = \begin{bmatrix} \mathrm{c}\theta\mathrm{c}\psi & \mathrm{c}\theta\mathrm{s}\psi & -\mathrm{s}\theta \\ -\mathrm{s}\phi\mathrm{s}\theta\mathrm{c}\psi - \mathrm{c}\phi\mathrm{s}\psi & \mathrm{s}\phi\mathrm{s}\theta\mathrm{s}\psi + \mathrm{c}\phi\mathrm{c}\psi & \mathrm{s}\phi\mathrm{c}\theta \\ \mathrm{c}\phi\mathrm{s}\theta\mathrm{c}\psi + \mathrm{s}\phi\mathrm{s}\psi & \mathrm{c}\phi\mathrm{s}\theta\mathrm{s}\psi - \mathrm{s}\phi\mathrm{c}\psi & \mathrm{c}\phi\mathrm{c}\theta \end{bmatrix},$$

where $(\phi_{2/1}, \theta_{2/1}, \psi_{2/1})$ and sin/cos have been shortened as (ϕ, θ, ψ) and s/c, respectively. This method can be used to transform across any two coordinate frames when the relative Euler angles and translation vector are known.

2.2 Perspective-n-Point (PnP) Solution

The vision algorithms used to localize the lead aircraft from video make use of a calibrated camera model. The camera model defines how points in the real, three-dimensional, world are projected onto the two-dimensional image plane as pixel locations, given the geometry of the camera and the distortion of the lens. Solving the inverse problem of computing the 3-D position and orientation, or "pose", of a known object from 2-D image points is known within the computer vision community as the Perspective-n-Point (PnP) problem. This section will present the equations that define the camera model as well as introduce the PnP algorithm implemented in this thesis—the Efficient Perspective-n-Point (EPnP) algorithm.

2.2.1 Camera Model

The simplest camera model, called the *pinhole* camera model [23], does not account for lens distortion. It assumes that all rays of light entering the camera pass through a single point (a tiny aperture), and then get projected onto the image plane behind it. The distance between the pinhole plane and the image plane is called the focal length, f. Z is the distance along the optical axis between the pinhole plane and the object point. X and Y describe the position of the object in orthogonal directions to the optical axis. The light from the object point gets projected onto the image plane at the location (x, y), where

$$x = f\frac{X}{Z}, \qquad\qquad y = f\frac{Y}{Z}.$$

(Since the image will be mirrored on the image plane, it is conventional to define x and y to be positive pointing in the $-X$ and $-Y$ directions, respectively.)

In real cameras, it may be the case that the optical axis does not pass through the image plane at the exact center of the image sensor due to a misalignment created during manufacturing. The distance, in pixels, from the center of the image sensor to the intersection of the optical axis with the image plane (called the "principal point") can be described by the parameters c_x and c_y. Also, we are typically interested in knowing (x, y) in units of

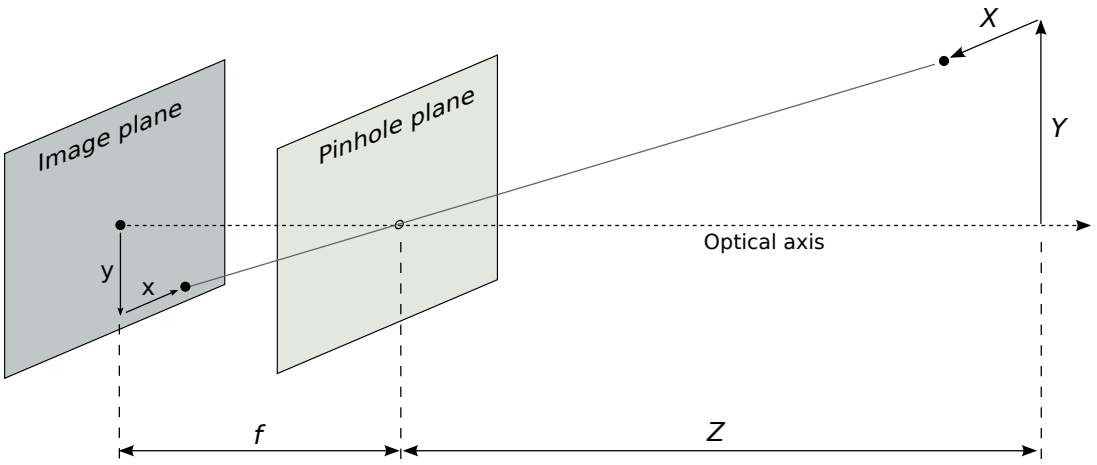


Figure 2.4: Pinhole camera model

pixels instead of units of physical distance. For this reason, we define f_x and f_y which are equal to the focal length, f, times the size of the individual image sensor elements, s_x and s_y, which may be rectangular or square. The image location in pixels (u, v), can now be written as

$$u = f_x \left(\frac{X}{Z}\right) + c_x, \qquad\qquad v = f_y \left(\frac{Y}{Z}\right) + c_y. \tag{2.1}$$

These equations map 3-D points (X, Y, Z) to the corresponding 2-D pixel locations (u, v), using the pinhole camera model. However, if we have a 3-D object of known geometry with dimensions specified in its own body frame of reference (X_{obj}, Y_{obj}, Z_{obj}) we must first apply the coordinate transformation that relates the body axes of the object to the camera frame of reference. Only after we apply the coordinate transformation can we compute the corresponding pixel locations. The location of an object point can be transformed into camera coordinates by the following equation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R \begin{bmatrix} X_{\text{obj}} \\ Y_{\text{obj}} \\ Z_{\text{obj}} \end{bmatrix} + t, \tag{2.2}$$

where R is the rotation matrix that rotates the object's body frame into the camera frame and t is the relative translation vector between the object's body frame and the camera frame, with respect to the camera frame.

Combining Eqs. (2.1) and (2.2), and combining R and t to become the joint rotation-translation matrix, $[R|t]$, we get the following matrix equation:

$$w \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X_{\text{obj}} \\ Y_{\text{obj}} \\ Z_{\text{obj}} \\ 1 \end{bmatrix} \tag{2.3}$$

where w is the scalar projective parameter.

In reality, cameras do not fit the pinhole model because they exhibit lens distortions due to the manufacturing process. The shape of the lens itself can contribute to radial distortions while any misalignment of the image sensor can contribute to tangential distortions. Fortunately, the effects of lens distortion can be removed if a camera is properly calibrated. By going through the calibration process, the lens distortion can be measured and subsequent images can be corrected for distortion. This makes it possible to use even inexpensive cameras, manufactured to loose tolerances, for high-quality pose estimation.

When we wish to account for distortion in the camera model, we must first obtain the k_1, k_2, and k_3 values, which are determined by the radial distortion, and p_1 and p_2, which are determined by the tangential distortion. The parameters k_1, k_2, k_3, p_1, and p_2 can all be found through calibration. Then, starting from Eq. (2.2), we can map 3-D object points to 2-D image points, in units of pixels, using

$$X' = X/Z \tag{2.4}$$

$$Y' = Y/Z \tag{2.5}$$

$$r^2 = X'^2 + Y'^2 \tag{2.6}$$

$$X'' = X'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 X'Y' + p_2(r^2 + 2X'^2) \tag{2.7}$$

$$Y'' = Y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2Y'^2) + 2p_2 X'Y' \tag{2.8}$$

$$u = f_x X'' + c_x \tag{2.9}$$

$$v = f_y Y'' + c_y. \tag{2.10}$$

This set of equations makes up the camera model used by the vision algorithms implemented by this thesis.

2.2.2 EPnP Algorithm Overview

When solving a PnP problem, we actually seek the joint rotation-translation matrix $[R|t]$ that describes the object's position and orientation (pose) relative to the camera frame, given a set of n 3-D to 2-D point correspondences. Consider Eq. (2.3) and Eqs. (2.4) to (2.10): The image points, (u, v), are captured by the camera and are therefore known, the intrinsic camera properties (f_x, f_y, c_x, c_y, k_n, p_1, and p_2) are obtained beforehand through calibration, and the model geometry (X_{obj}, Y_{obj}, Z_{obj}) is assumed to be known. Unfortunately, the pose cannot be solved directly due to a number of reasons:

- Error in measuring the object's geometry
- Error in measuring the camera's intrinsic parameters through calibration
- Error in determining the pixel location of the image points to sub-pixel accuracy
- The object may be flexible (not-rigid) and may not always match the measured geometry

Therefore, the PnP problem becomes one of optimization where the reprojection error should be minimized. Most PnP algorithms solve for the pose iteratively, which is very accurate when the solution converges properly. However, iterative methods can sometimes converge to a *local minimum*—resulting in an incorrect pose estimate. Additionally, iterative methods can be slow to converge if they are not "primed" with a close initial guess. The method used in this thesis utilizes the non-iterative Efficient Perspective-n-Point (EPnP) method [24] to compute a slightly less accurate pose, which is then used as the initial guess with Gauss-Newton optimization to quickly refine the pose estimate. This approach significantly reduces computation time without compromising accuracy and prevents the optimizer from converging to a local minimum.

The EPnP algorithm first reduces the n 3-D reference points to a weighted sum of four "virtual control points". The n 3-D reference points can be written as

$$p_i, \quad i = 1, \ldots, n$$

and the 4 control points can be written as

$$c_j, \quad j = 1, \ldots, 4.$$

Each reference point can then be described by

$$p_i = \sum_{j=1}^{4} \alpha_{ij} c_j, \quad \text{with} \quad \sum_{j=1}^{4} \alpha_{ij} = 1, \tag{2.11}$$

where α_{ij} are homogeneous barycentric coordinates. The control points can be chosen arbitrarily, but in practice are typically chosen such that one control point lies at the centroid of the reference points and the rest form a basis aligned with the principal directions of the data.

From here, we will consider the simplified projection model from Eq. (2.3). For clarity, however, keep in mind that X and Y could easily be replaced with X'' and Y'' if we chose to account for distortion. We will let the coordinates of c_j be notated as $(\hat{X}_j, \hat{Y}_j, \hat{Z}_j)$. The projection model can now be written in terms of the four virtual control points as

$$\forall i, \quad w_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \sum_{j=1}^{4} \alpha_{ij} \begin{bmatrix} \hat{X}_j \\ \hat{Y}_j \\ \hat{Z}_j \end{bmatrix}. \tag{2.12}$$

This leaves the 12 coordinates of the control points $\{(\hat{X}_j, \hat{Y}_j, \hat{Z}_j)\}_{j=1,\ldots,4}$ and the n projective parameters $\{w_i\}_{i=1,\ldots,n}$ as the unknowns of the linear system. Equation (2.12) can be rewritten as two expressions to eliminate w_i

$$\begin{aligned} \sum_{j=1}^{4} \alpha_{ij} f_x \hat{X}_j + \alpha_{ij}(c_x - u_i)\hat{Z}_j = 0, \\ \sum_{j=1}^{4} \alpha_{ij} f_y \hat{Y}_j + \alpha_{ij}(c_y - v_i)\hat{Z}_j = 0. \end{aligned} \tag{2.13}$$

These equations can then be concatenated for all i to create a linear system of the form

$$\mathbf{M}x = 0, \tag{2.14}$$

where $x = \left[c_1^\top, c_2^\top, c_3^\top, c_4^\top\right]^\top$ is a 12×1 vector, and $\mathbf{M}$ is a $2n \times 12$ matrix formed by arranging the coefficients of Eq. (2.13). The solution to a linear system of the form $\mathbf{M}x = 0$

is the null space of $\mathbf{M}$, and can be found as the set of null eigenvectors of $\mathbf{M}^\top\mathbf{M}$. We choose the subset of the null eigenvectors that minimize the reprojection error. The main advantage of this algorithm is that no matter the number, n, of point correspondences, the product of $\mathbf{M}^\top\mathbf{M}$ will be of constant size 12×12 and of $O(n)$ complexity to compute. The details of computing the eigenvectors can be found in [24].

Once the virtual control points have been found in *camera coordinates*, we can solve for the Euclidean motion that aligns the camera frame with the object frame by applying any of the methods presented in [25], [26], or [27]. The methods are too lengthy to describe in detail here. Instead, the reader is referred to the original documents. This will provide $[R|t]$, which is the solution to the PnP problem.

Chapter 3

Development Platform and Hardware

The purpose of this chapter is to introduce the major components used to assemble a working formation flight system: the autopilot board, camera, embedded Linux computer and associated video processing software, and finally, the airframe itself. Whenever possible, low-cost, open source components were selected.

3.1 The Case for Open Source

Although the open source concept sometimes has its own set of problems (a lack of structured support, insufficient documentation, limited beta testing), it offers a significant advantage in cost. For example, the open source *ArduPilot Mega 2.5* autopilot board used by this thesis can be obtained for approximately $160. A comparable closed source autopilot system, such as the newly released *Piccolo Nano* by Cloud Cap Technology, costs over $1,000. Prior to the *Piccolo Nano*, many commercial autopilot systems cost in excess of $5k.

Although there is no formalized support for open source projects, most provide support through an online-based community of user/developers actively contributing to the project. While there is no guarantee that support requests will be met by someone with appropriate experience, it is generally possible to seek out help for basic troubleshooting. At the very least, the open source forums make it easy to establish contact with others in the community who are facing the same problems, making it possible to work collaboratively to reach a solution.

The main advantage, however, of using open source components for a project such as this is the unparalleled level of customization. Unlike proprietary hardware and software, which cannot be used outside of the intended application, open source components can be modified without limitation. This fact is illustrated in [18], where the author has to go to great lengths to implement a formation flight controller for small UAVs using the *Piccolo II* autopilot. Due to the closed source nature of the autopilot, the relative navigation capability had to be implemented as part of the ground station software. The GCS was configured to receive telemetry data from both airplanes, compute a new waypoint and throttle command, and finally "insert" the waypoint to the following aircraft's queue and command airspeed by broadcasting from the GCS back to the trailing aircraft. It is easy to see how this approach can be problematic if the communications link is temporarily lost. Likewise, the delays associated with wirelessly transmitting telemetry and commands can contribute to reduced bandwidth of the guidance controller.

3.2 Hardware Architecture

Here, the general hardware architecture is introduced to familiarize the reader with the main components of the system and how they interact with one another. The hardware is not much different from a standard RC setup except for the addition of the autopilot, vision processing equipment (follower only), and the LEDs and their dedicated power supply subsystem (leader only). Both the leader and follower are equipped with an *ArduPilot Mega* (APM) autopilot, which receives control inputs from the RC receiver and outputs throttle and servo commands.

In the case of the leader, the APM board also outputs a signal which is used to switch the LEDs ON or OFF using a relay circuit driven by an N-channel MOSFET. This allows the pilot to remotely switch the LEDs to conserve energy stored by a dedicated LED battery and to prevent overheating of the LEDs. Since the LEDs are rated for a forward voltage of 16.1V, a DC/DC voltage step-up converter is used to increase the 3-cell lithium polymer battery voltage from 11.1V.

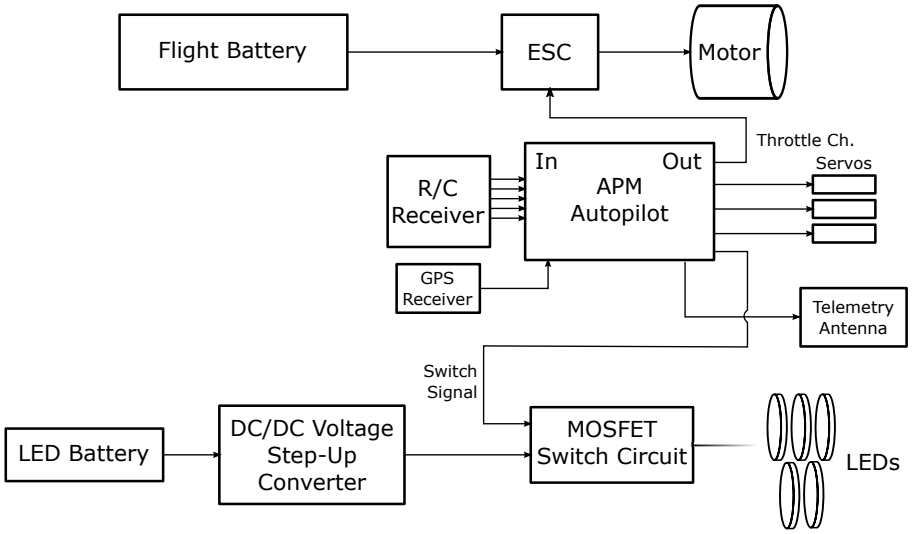


Figure 3.1: Hardware Architecture for Leading UAV

Conversely, the follower is equipped with a USB webcam for capturing live video and a single board embedded Linux computer for processing the video stream in real-time. The embedded computer passes the estimated relative state vector to the APM autopilot over serial communication, which is then used by the APM for guidance and control. The single-board computer requires a power supply at 5V, which is provided by a battery eliminator circuit (BEC).

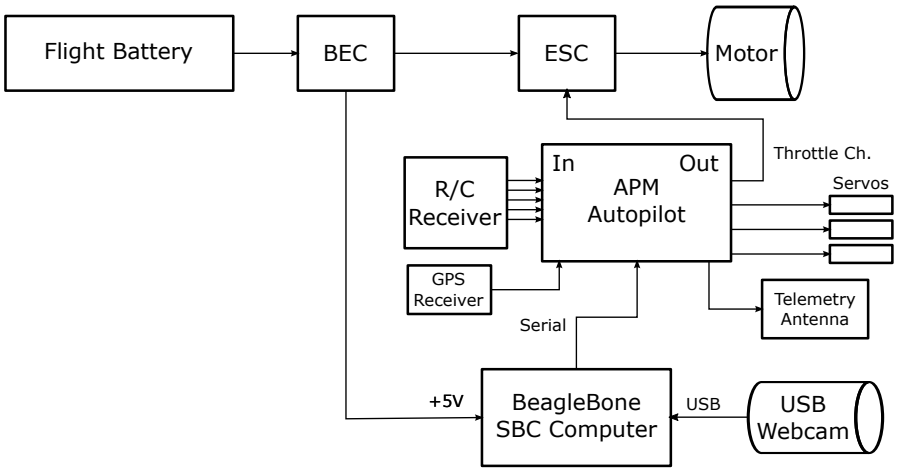


Figure 3.2: Hardware Architecture for Following UAV

3.3 Autopilot

As already mentioned, the autopilot board selected for this project is the *ArduPilot Mega 2.5* (APM), shown in Figure 3.3. the APM is sold by 3D Robotics and supported through the DIY Drones online community. The APM is a complete open source microcontroller-based autopilot system that includes all necessary sensors (3-axis gyro, accelerometer, magnetometer, barometric altitude sensor, GPS receiver, etc). The board can be loaded with a number of different versions of open source firmware for use with fixed wing aircraft, helicopters, multi-copters, or even land or aquatic vehicles. This thesis develops upon Version 2.68 of the *ArduPlane* firmware for fixed-wing aircraft. Although depreciated, this version was selected for its stability with Hardware-In-Loop simulation and PID-based lateral guidance. Later versions of *ArduPlane* have known issues running Hardware-In-Loop simulation and use L1 adaptive control for lateral guidance, which would have been more difficult to modify for formation flight. Technical details about the APM are listed in Table 3.1.

Figure 3.3: ArduPilot Mega 2.5 autopilot board

Table 3.1: ArduPilot Mega 2.5 specifications

Components:	
Microcontroller	Atmel ATMEGA2560 and ATMEGA32U-2
Digital compass	Honeywell HMC5883L-TR
Inertial Measurement Unit	Invensense 6 DoF Accelerometer/Gyro MPU-6000
Barometric Pressure Sensor	Measurement Specialties MS5611-01BA03
Physical Dimensions:	
W × H × D	1.60” × 0.26” × 2.63”
Weight	0.81 oz

3.4 Camera

Initially, the *Sony PlayStation Eye* USB webcam was selected for video acquisition due to its low cost (under $15), high frame rate, relatively high resolution, and availability of drivers for the Linux operating system. However, as explained in Appendix E, the PlayStation Eye could not deliver a sufficiently high frame rate to the embedded vision processing computer due to a problematic USB driver. Instead, the more expensive *Logitech C920* USB webcam ($75), shown in Figure 3.4 was used because it can deliver compressed frames in MJPEG format, which requires less USB bandwidth. The mounting bracket was removed from the camera to reduce weight and the transparent glass lens was removed as it was found to cause distortions in the appearance of bright lights. Later, a plastic tinted lens was affixed to the front of the camera to reduce the image exposure in bright ambient light situations.

Figure 3.4: Logitech C290 USB Webcam

3.5 Vision Computing

3.5.1 Embedded Linux Computer

Video processing and pose estimation was done using the open source *BeagleBone Black* embedded Linux single-board computer (SBC), sold by BeagleBoard.org. The board, pictured in Figure 3.5, features a 1 GHz processor, 512 MB of DDR3 RAM, and is only 3.4×2.1

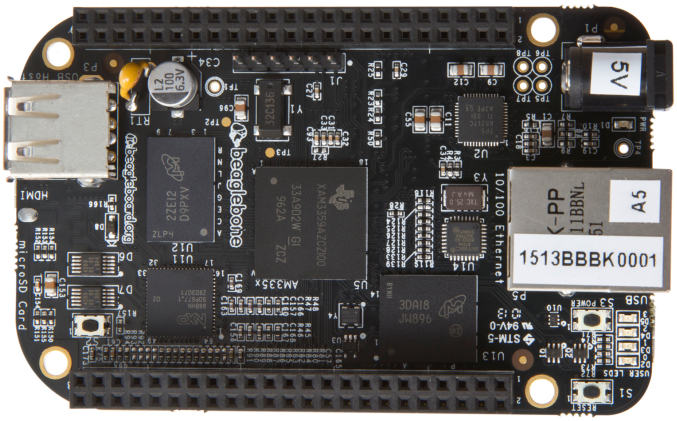


Figure 3.5: BeagleBone Black single-board embedded Linux computer

inches in size. The board has USB connectivity, making it easy to interface with a USB webcam, and UART header pins so that it can communicate with the *ArduPilot Mega* over a standard RS-232 serial connection. Moreover, the SBC is very inexpensive at only $45. Since the *BeagleBone Black* is capable of running versions of the Linux operating system, it is possible to use the Ubuntu OS and its associated package management tool, Aptitude, for installing software libraries, compilers, command line tools, etc. The ability to program in high-level languages, such as C/C++ and Python, using pre-built libraries makes development with the *BeagleBone Black* easy and powerful. Details about the board are listed below:

- 1 GHz superscalar ARM Cortex-A-8 AM3359 processor
- 512 MB DDR3 RAM
- 3D graphics accelerator
- NEON floating-point accelerator
- 2 GB 8-bit eMMC on-board flash storage
- MicroSD slot for additional user data or operating systems
- 1× USB 2.0 client port
- 1× USB 2.0 host port
- Ethernet
- Micro-HDMI audio/visual output
- 2× 46-pin headers
- Total size: 3.4” × 2.1”

3.5.2 OpenCV Software Libraries

OpenCV is a free, open source, cross-platform computer vision library originally developed by Intel, and now supported by Willow Garage and Itseez. *OpenCV* was designed with efficiency in mind and a focus on *real-time* video processing. *OpenCV* is extremely popular, with over 6 million downloads and a user community of more than 47 thousand people. It has many applications such as factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, robotics, augmented reality, motion tracking, facial recognition, 3-D pose estimation, and even includes a machine learning library. It is also easy to develop with *OpenCV* since it has interfaces for programming in C, C++, Python, and Java. The combination of real-time processing and high-level programming make *OpenCV* a natural choice for a project such as this, where time and resources are limited.

3.6 Airframe

Figure 3.6: Penguin V2 ARF kit with M2815 Motor and 60A ESC

Early work assumed that the *SkySurfer*, sold by Banana Hobby, would be used as the test bed for the proposed system. However, as the project developed, it became clear that a larger airframe was required to handle the large payloads. The airframe for both the leader and follower was revised to the *Penguin*, sold by ReadyMadeRC.com and shown in

Figure 3.6. The aircraft has 4-channel controls, features durable EPO foam construction, and has a high-mounted pusher propeller. The trainer-like configuration and availability of replacement parts make it easy to repair if any damage should it occur. The Penguin is specifically marketed for autonomous and first-person-view applications since it has sufficient volume to carry additional electronics payloads and includes easy-to-interface camera mounting hardware. The model has a 67.7 inch wingspan and is 48.5 inches long.

3.7 High-Intensity LEDs

An important step in any vision-based navigation system is the process of detecting and identifying features in the video images. The problem of reliably detecting features is often complex, computationally expensive, and could be discussed at such a length that it would easily fill a master's thesis in itself (as in [28]). Since this thesis aims to present a complete vision-based guidance system, the task of identifying features was made simpler by affixing five high-intensity red light emitting diodes (LEDs) to the leader's airframe, as done in Mahboubi et al. [20]. The color red is used to contrast with the backdrop of the blue sky and is easier to work with than wavelengths outside of the visible spectra, such as infrared. Personal correspondence with Mahboubi revealed that the group used *Luminus PT-120* red LEDs which have a typical drive current of 30A at a 25% duty cycle and brightness of approximately 2,000 lumens.

At the cost of making feature detection slightly more difficult, *Luxeon Star's 714 Lumen 7-LED assembly*, shown in Figure 3.7, was chosen to make powering the LEDs easier. Each Luxeon Star LED has a typical forward voltage of 16.1V and operate at a nominal current of 700mA. The LEDs are 40mm in diameter and required the design of custom mounting hardware to affix the LEDs to the airframe of the lead UAV (Appendix B). The LEDs were powered by a dedicated 3-cell lithium polymer battery and DC/DC voltage step-up converter. A power MOSFET driven relay circuit was designed to make the LEDs remotely switchable so as to avoid overheating and/or depleting the LiPo battery prematurely.

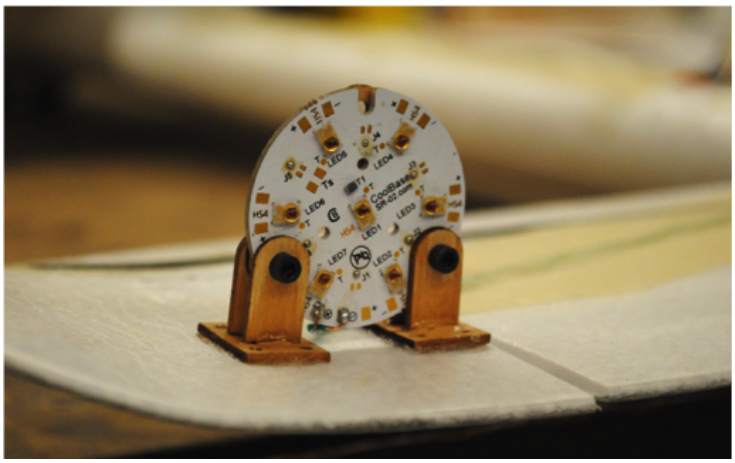


Figure 3.7: Luxeon Star 7-LED Assembly (714 lm)

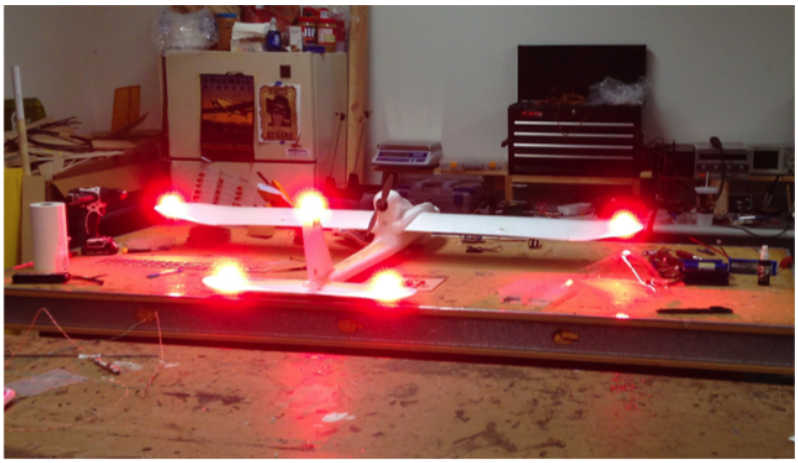


Figure 3.8: Red LEDs mounted to airframe in predefined positions

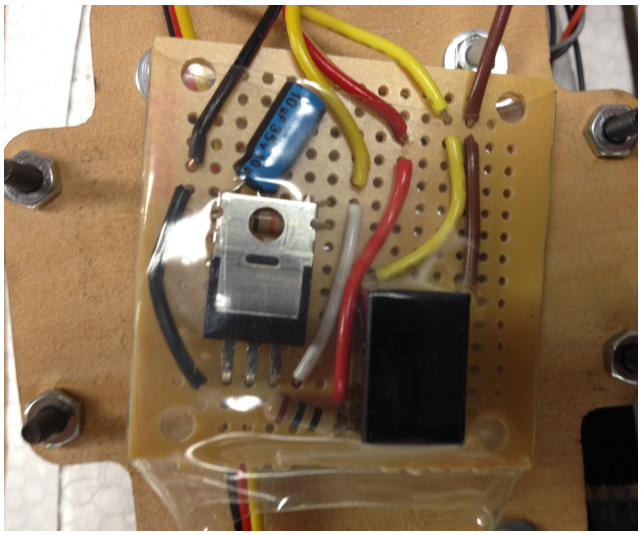


Figure 3.9: MOSFET driven relay circuit for remote LED switching

Chapter 4

LED Detection and Tracking

One of the greatest challenges for any vision-based navigation system is having the ability extract feature points from video images. Though feature extraction is an essential prerequisite to pose estimation, this thesis does not focus on extraction methods. Instead, the concept of using bright LEDs to make easy-to-detect image features was adopted from [20]. A simple, yet robust feature extraction algorithm was developed. First, the vision software is introduced in its entirety in Section 4.1. Section 4.2 describes work that was carried out early-on in the design process in attempt to derive the required LED brightness. Section 4.3 describes the process flow of the LED detection scheme, as it is implemented into the final revision of the vision processing software.

4.1 Vision Software Overview

Before moving forward, the vision software should be introduced in its complete form at the most basic level. Doing so will help by providing a road map for understanding an otherwise complex software system. Figure 4.1 presents the general process flow of the computer vision software employed by this thesis. At the beginning of execution, the system is initialized by reading in user-defined settings, the 3-D geometry of the lead UAV, and camera calibration data as well as opening the USB webcam for capturing. Once initialization is complete, the video capture/processing loop begins. For each frame of video, an image is captured and LEDs are detected from the raw image data (the subject of this chapter). A vector of image points corresponding to the LED locations is used as input to the 3-D pose estimation

algorithm where the relative state of the lead UAV is computed. The state vector is then passed through a thresholded Kalman filter to remove any outlying observations and the resulting estimated state is then sent to the autopilot over a serial connection before a new frame is captured and the process repeats.

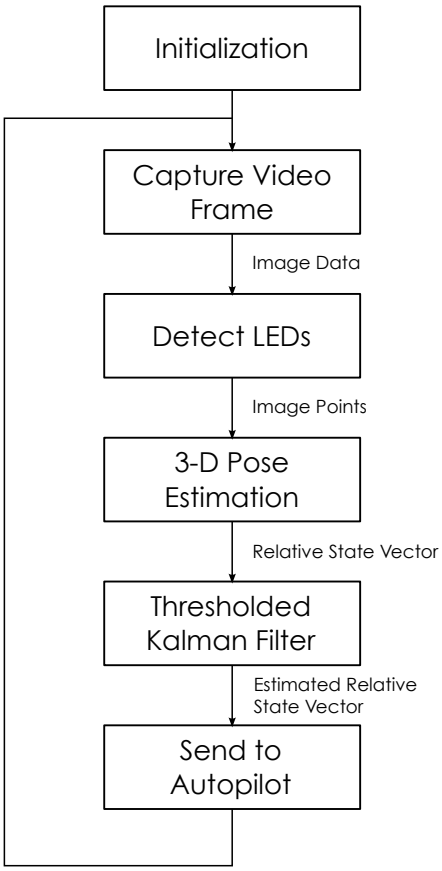


Figure 4.1: Vision subsystem process flow diagram

4.2 Deriving Required LED Brightness

Before the *Luxeon Star 714 lumen 7-LED assemblies* were selected, efforts were made to derive the required LED brightness for successful feature detection in lumens. Since there are an overwhelmingly large number of factors that can affect the success or failure of detecting any given feature in a single frame (camera exposure, lens distortion, imaging sensor noise, ambient lighting conditions, processing techniques, etc.) the task was approached with a

combination of outdoor testing and manipulation of the governing optical equations.

4.2.1 Engineering Approach

The idea was to experimentally determine the distance from which a less-expensive 50 lumen red LED could reliably be detected in various background scenes. From that, it is possible to predict the brightness of an LED required to detect it from the desired distance. This can be done by manipulating the following equations:

$$\Phi_v = I_v \cdot \Omega \tag{4.1}$$

where Φ_v is the luminous flux in lumens, I_v is the luminous intensity in candelas, and Ω is the angular span in steradians. The angular span accounts for non-uniform brightness and is a function of the beam angle, 2θ. It can be described by

$$\Omega = 2\pi \left(1 - \cos\theta\right). \tag{4.2}$$

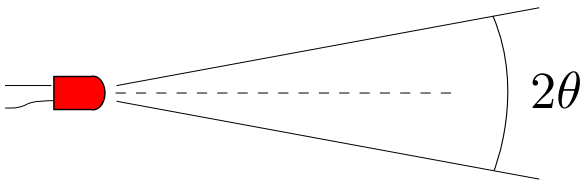


Figure 4.2: LED beam width

Illuminance is a measure of the luminous flux per unit area and is given in units of lux (1 lux = 1 $\frac{\text{lumen}}{\text{m}^2}$). The illuminance can be thought of as the perceived brightness of a light source at a given distance. Often times, video cameras have a specified minimum illuminance level at which the camera will record a satisfactory image. The illumination from the LED can be expressed as

$$E_v = \frac{I_v}{D^2}, \tag{4.3}$$

where D is the distance from the light source in meters.

By substituting Eqs. (4.1) and (4.2) into Eq. (4.3) and solving for E_v, we arrive at the following equation

$$E_v = \frac{\Phi_v}{2\pi\,(1 - cos\theta)\,D^2}, \tag{4.4}$$

which is convenient because it allows the illuminance to be described as a function of the LED brightness, Φ_v, and beam angle, 2θ, and the distance from the light source, D. Therefore, it is possible to experimentally determine the E_v value at which an LED of a known brightness and beam angle can no longer be detected reliably simply by measuring the distance of the LED from the camera, D. Then the necessary LED brightness can be determined by assuming a beam angle and back-substituting the resulting value of E_v and the desired distance, D, into Eq. (4.4).

4.2.2 Outdoor Testing

Testing was carried out by running a custom LED detection program written with *OpenCV* to detect red LEDs based on brightness, hue, saturation, and morphological properties after thresholding. The *PlayStation Eye* camera was mounted to a stationary tripod positioned near the middle of the runway at Cal Poly's Educational Flight Range. (At the time of testing, the *PlayStation Eye* was still being considered for the USB webcam.) A 50 lumen LED was affixed to the end of a stick and moved abut the video image at a distance of 5 to 40 feet in 5 foot increments. The test was repeated for the camera pointing in six different directions: parallel to the runway in both directions, perpendicular to the runway in both directions, and directly towards and directly away from the setting sun.

For each scene, a single frame from the video was used to generate a histogram and a thresholded (binary) image of the scene's brightness. To quantify the "goodness" of a scene for detecting LEDs, the ratios $\frac{M3}{M3_u}$ and $\frac{M_t3}{M_t3_u}$ were computed. $\frac{M3}{M3_u}$ is the ratio of the third moment of the image histogram rotated about the vertical axis divided by the third moment of an idealized uniformly distributed image histogram. $\frac{M_t3}{M_t3_u}$ is the ratio of the third moment of the histogram *above the threshold value* rotated about the vertical axis divided by the same moment of a uniformly distributed histogram. In general, lower ratios indicate that a scene is better suited for LED detection as it has fewer saturated

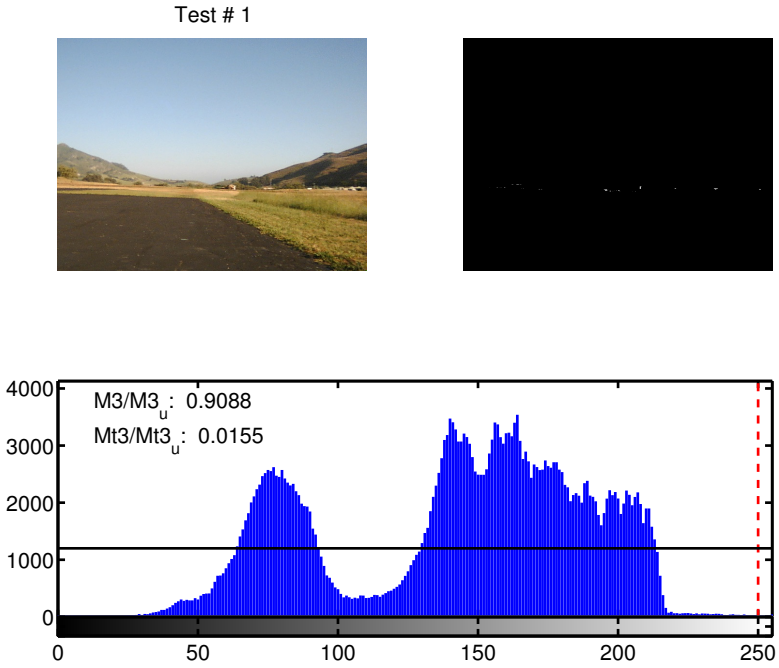


Figure 4.3: A "good" scene for detecting LEDs with few saturated pixels

pixels. Figures 4.3 and 4.4 illustrate "good" and "bad" scenes, respectively. The first scene has very few pixels with values above the threshold whereas the second scene has a large area of saturated pixels, making it is extremely difficult to detect LEDs with any degree of reliability.

To estimate the probability that a LED is detected in any given frame, a script was written in MATLAB to assist the user in identifying the correct (x, y) position of the LED throughout the test. Sample frames saved during the tests at 1 Hz intervals were presented in MATLAB so that the user could move a cursor over the image and click on the LED to provide the script with the approximate position of the LED in the image, as shown in Figure 4.5, where the red circles represent possible LEDs and the green circle represents the most-probable LED position based on the feature's properties. In this case, the green circle *is* the LED and the red circles are false positives (noise).

The (x, y) positions provided by the user were interpolated between video frames to

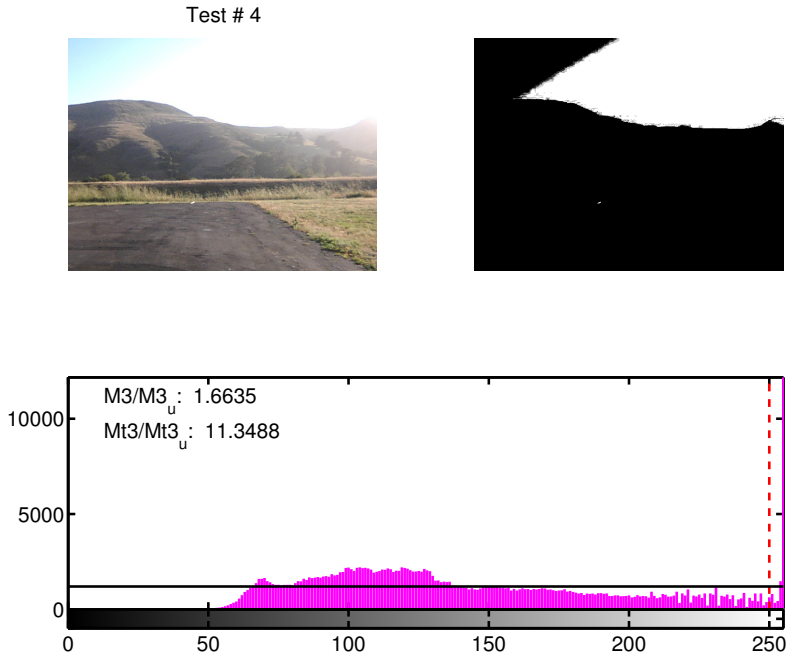


Figure 4.4: A "bad" scene for detecting LEDs with many saturated pixels

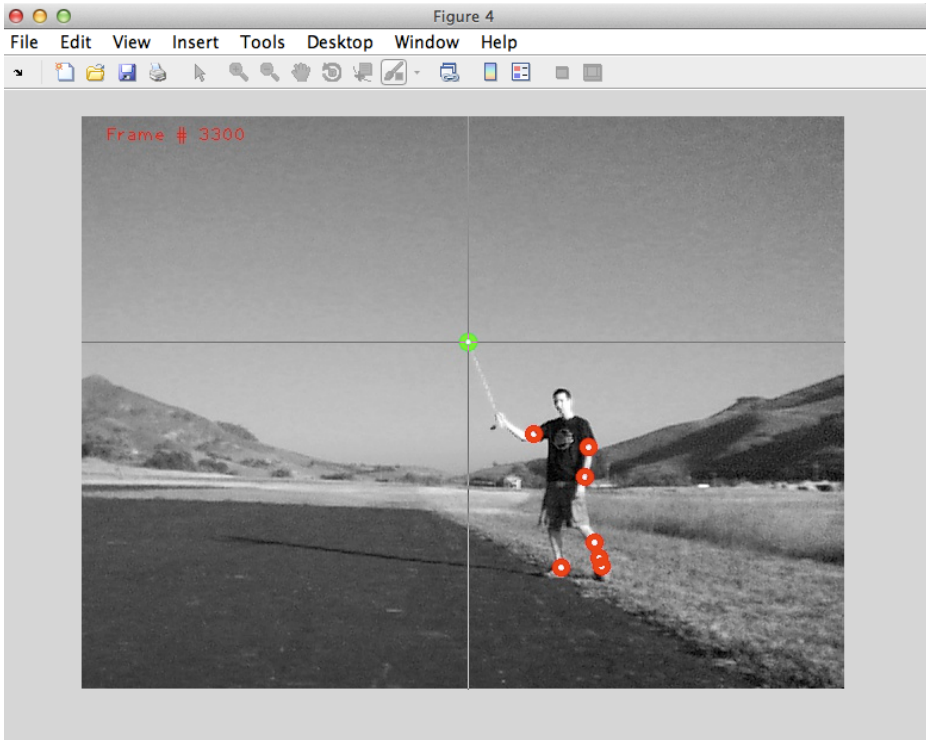


Figure 4.5: MATLAB software tool used to track actual LED position in image

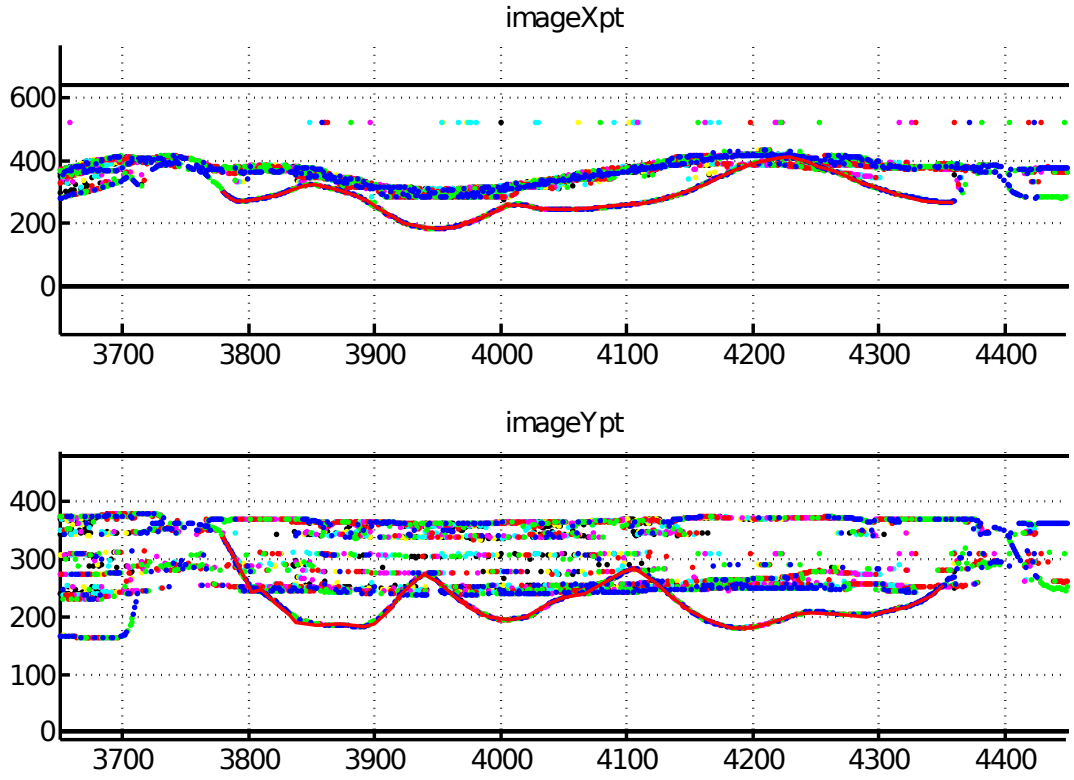


Figure 4.6: (x, y) **positions of detected feature points**

prescribe the "ideal" motion of the LED. The (x, y) positions of all detected points (including false positives) are plotted in Figure 4.6. In cases where multiple image points were detected, the points are plotted in different colors. The user-supplied "ideal" LED motion is plotted as a red curve over the two plots and for each frame the minimum error, $|\Delta x| + |\Delta y|$, is computed. The minimum error of each frame is compared to a threshold value and the LED is considered to be successfully detected if the error is less than the threshold and unsuccessfully detected otherwise.

The above process was repeated for each of the six scenes and the percentage of frames with successfully detected LEDs was plotted against the distance between the LED and the camera. Figure 4.7 shows the variability in LED detection rate due to the nature of the background scenery. (Each scene is represented by a curve of a different color.) Figure 4.8 presents the probabilities raised to the power of 5 to predict the probability that *all five*

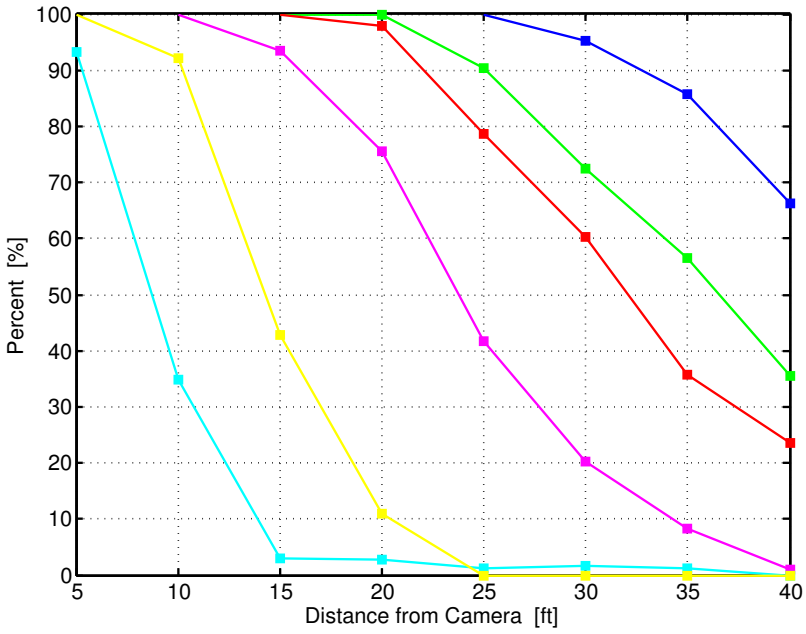


Figure 4.7: Probability of detecting single LED

LEDs would be successfully detected, assuming that the probability of detecting any single LED is independent of the success or failure of detecting others.

Unfortunately, this work was abandoned here for a number of reasons. First, it was not clear what LED detection probability was required for successful relative navigation since poor LED detection could be mitigated with state estimation techniques and Kalman filtering. Also, one could assume that the success or failure of detecting an LED is stochastically independent, however this is untrue because often an LED will fail to be detected in several subsequent frames due to the similarity of the images from frame-to-frame. Finally, these tests were all performed using the *PlayStation Eye*. It wasn't until later that the *Logitech C920* was substituted for the reasons outlined in Appendix E. Moreover, a tinted lens was later added to the webcam to reduce the number of saturated pixels, effectively retaining more visual information that could be used to better-detect the LEDs.

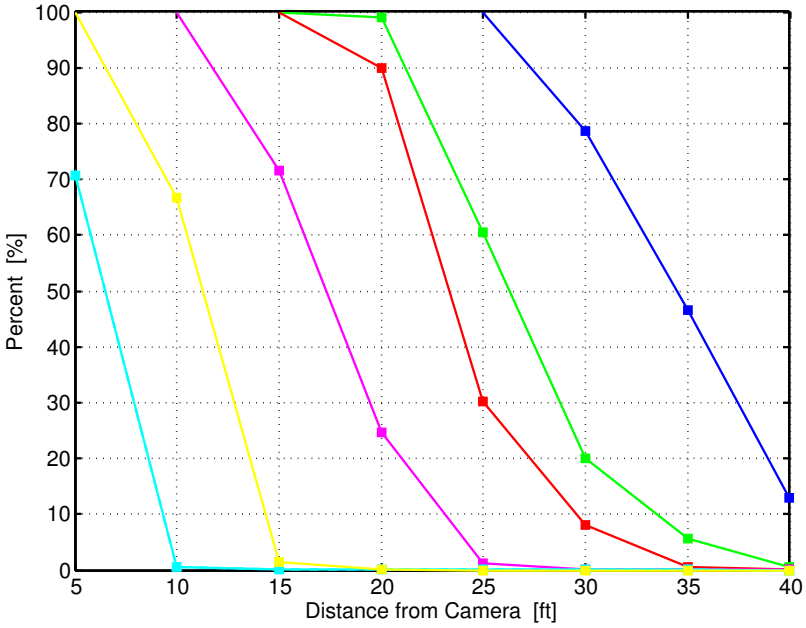


Figure 4.8: Probability of detecting five LEDs

4.3 LED Detection Algorithm

The general process for extracting the LED pixel coordinates from video is illustrated in Figure 4.9. Once a color image has been provided by the USB webcam, the red, green, and blue (RGB) channels are extracted as new "gray" images. The red channel is then thresholded to generate a binary image. (Pixels with a value *above* a certain threshold are replaced with a binary "1" while pixels *below* the threshold are replaced with a binary "0".) If the thresholded value is chosen correctly, LEDs should appear in the image as groups of white spots (1's) while the background pixels appear black (0's). Of course, bright areas in the scene will contribute to image noise and may incorrectly be thresholded as foreground pixels. Such features will be filtered out later.

Next, the border following algorithm from [29] is employed to detect external contours from the binary image. Properties of the vectorized contour objects can then be computed and used for filtering and prioritizing the contours by their expected likelihood of being LEDs. (The filtering and prioritizing techniques are somewhat complex and explained in

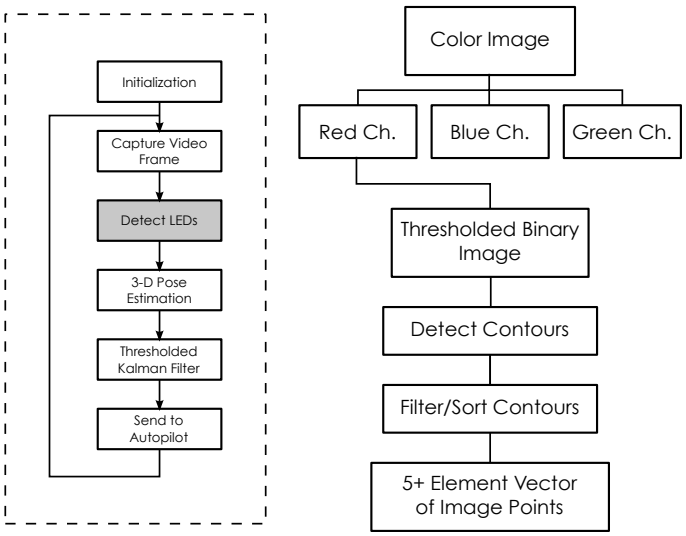


Figure 4.9: LED Detection process flow diagram

the next section.) Finally, a vector of image points corresponding to the LED image points is passed on to the pose estimation algorithm where the relative state vector will be computed.

4.3.1 Contour Filtering Techniques

The complex filtering algorithm presented here was developed over the course of the entire project. Improvements were made largely by process of trial and error. As the vision software was tested on new hardware, in new outdoor environments, and designed to track new objects, problems were encountered with the feature detection portion of the system and fixes had to be made. Many of the changes sought to take advantage of prior knowledge from previous video frames. In other words, it is assumed that the (x, y) pixel locations of the LEDs only change by *small* increments from frame-to-frame. The final algorithm, explained here, is the culmination of those improvements.

The process flow diagram for the contour filtering/sorting algorithm is shown in Figure 4.10 and can be described by the following steps:

1. Beginning with a set of vectorized contour objects, a rectangular region of interest (ROI) is defined around the reprojected image points from the previous video frame

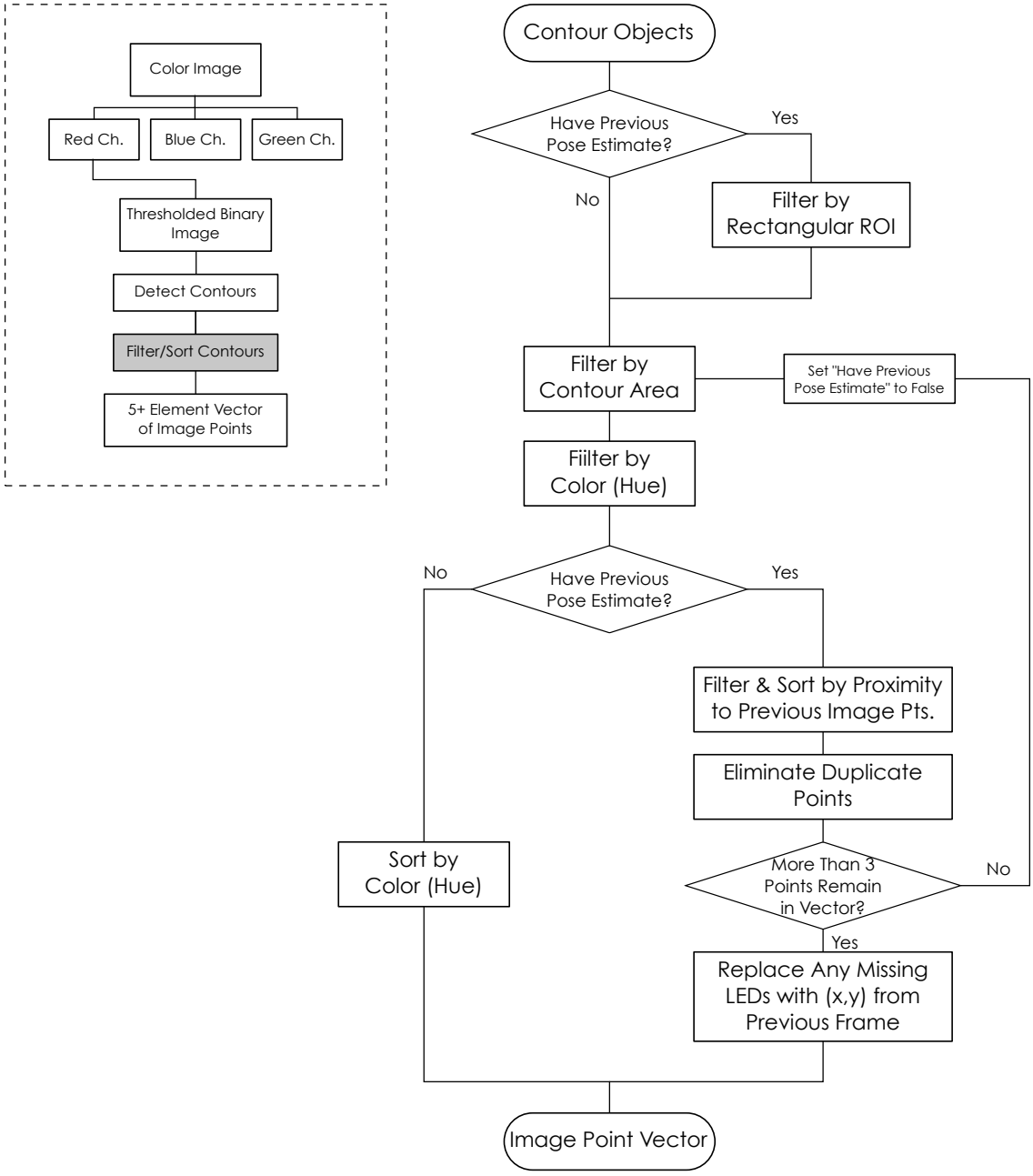


Figure 4.10: LED Detection filtering process flow diagram

if, and only if, a pose estimate was successfully found in the previous frame. The ROI is the minimum bounding rectangle plus some additional margin on all sides (shown in Figure 4.11).

2. Next, the area of each contour is computed and any contours having an area that falls outside of the prescribed min/max range are eliminated from the set. (The contour area is the total number of pixels that fall on or within its border.)

3. The average hue of the contour is computed by referencing the original composite RGB image. Any contours with a hue value falling outside of the acceptable bounds are eliminated.

4a. If the previous frame yielded a successful pose estimate:

   (a) A circular region of interest is defined around each of the reprojected image points of the LEDs from the previous frame (see Figure 4.11). The radius of the ROIs is chosen beforehand as a percentage of the apparent wingspan, in pixels. Contours that do not lie inside any of the circular ROIs are elminated. The remaining contours are grouped by the ROI in which they are contained.

   (b) If more than one image point falls inside a circular ROI, the points are sorted by decreasing Euclidian distance from the center of the ROI.

   (c) Any contours that appear as duplicates in more than one ROI group, are eliminated so that only one occurrence of that contour remains. (The occurrence with the shortest Euclidian distance to its respective ROI center is preserved.)

   (d) If, at this point, fewer than 3 circular ROIs contain non-duplicate contours, then the algorithm returns to Step 2 with the revised assumption that the previous frame did *not* provide a successful state estimate.

   (e) For any circular ROI's not containing contours from the current frame (indicating "undetected" LEDs), the center of the circular ROI is taken as the corresponding LED's image point.

4b. If the previous frame did *not* yield a successful pose estimate, then contours are sorted

according to their color. They are ranked in decreasing order of their "closeness" to the ideal hue.

5. The vector of LED image points is then returned. The vector contains at least 5 elements (corresponding to each of the 5 LEDs), but may contain more if the "have previous pose" condition was not met. This allows the pose estimation algorithm to iterate by "swapping" image points until it finds a 5-element subset that results in a suitably low reprojection error. In the case that the "have previous pose" condition *is* maintained throughout the process, then the image point vector will be exactly 5 elements long and the points will be pre-sorted according to their matching LED. This allows the correspondence algorithm, described in Section 5.4, to be bypassed altogether for the current frame.

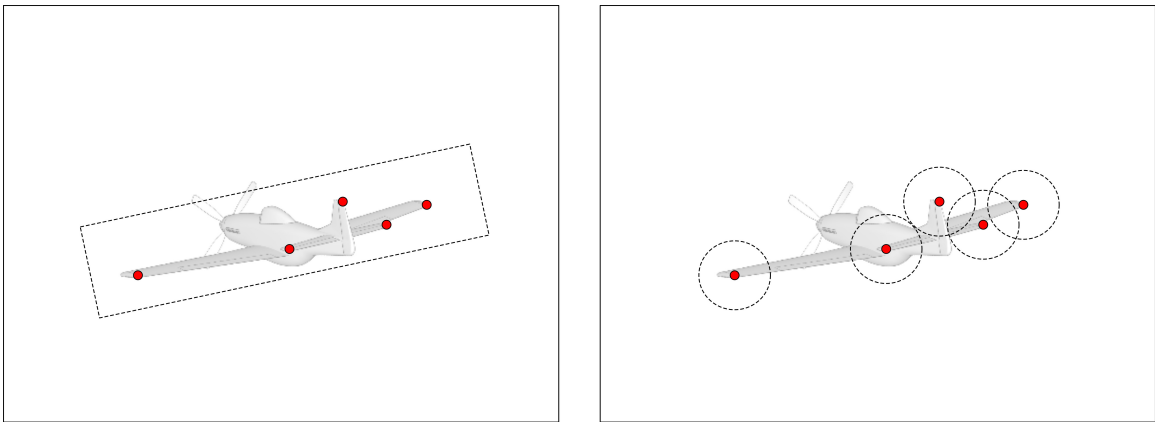


Figure 4.11: Minimum bounding rectangular ROI (left) and circular LED ROIs (right)

Chapter 5

3-D Pose Estimation

This chapter describes how a relative state estimate is computed in the context of the vision software developed for this project. Before continuing with this chapter, the reader should be familiar with the general flow of the vision software, as explained in Section 4.1. Section 5.1 explains the process of calibrating the USB camera so that lens distortion effects can be removed from the image. Section 5.2 presents the general process flow used by the state estimation portion of the software and Sections 5.3 and 5.4 explain how image points are rearranged and correlated, respectively. Finally, Section 5.5 explains the need for outlier robust state estimation and the implementation and testing of a *thresholded* Kalman filter.

5.1 Calibration

As described in Section 2.2, the intrinsic properties of the camera must be measured through a set of calibration procedures before a pose can be estimated with any degree of accuracy. Fortunately, camera calibration is a very simple procedure once working calibration code has been developed. (The mathematics of calibration using planar homography are not addressed here. Instead, the reader is referred to [23].) For this thesis, the existing program `camera_calibration.cpp` was used, which is provided along with the OpenCV source code in addition to other sample programs.

The calibration procedure requires a set of sample photos to be taken of a "chessboard", with known geometry, from multiple views. Boards with alternating black and white squares are often used for camera calibration because the size of their squares is easily measured,

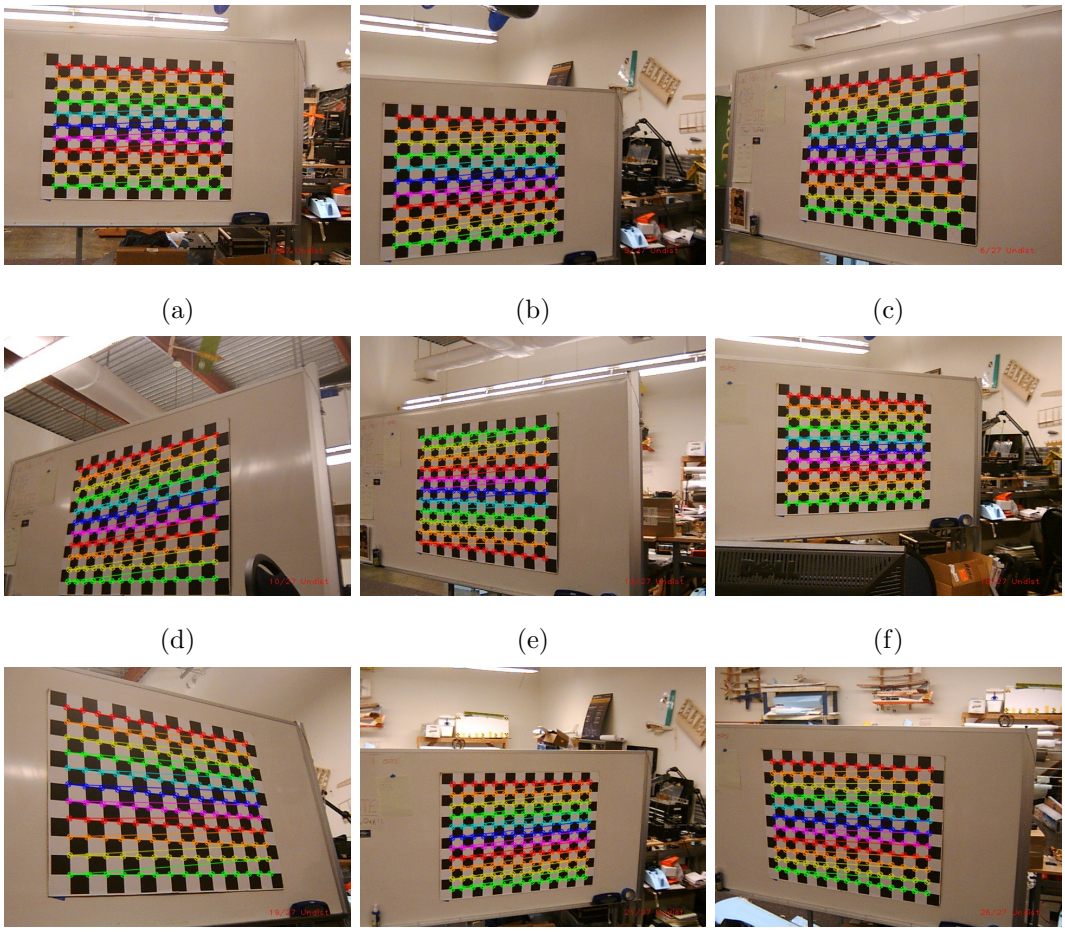
their pattern is known to lie on a flat surface, and it is relatively easy to determine the corner locations to sub-pixel accuracy. By applying the concepts of planar homography (the projective mapping from points on a 2-D planar surface to the camera's imager), the intrinsic properties of the camera can be estimated through iteration. Once the distortion coefficients are known, it is then possible to "undistort" images by correcting for distortion effects. An example can be seen in Figure 5.1.

Figure 5.1: Camera image before undistortion (left) and after undistortion (right) [23]

The Logitech C920 was calibrated at 640×480 resolution and at a "zoom" setting that resulted in a focal angle that nearly matched the 56° setting of the PlayStation 3 Eye. A 15×11-square board with squares of 75 mm in size was used for the calibration. Thirty-three photos were taken of the chessboard—each from a different view. All 33 images were then used to estimate the distortion coefficients of the camera using the `camera_calibration.cpp` program. A sample of the images used for calibration can be seen in Figure 5.2 with the detected chessboard corners marked. Once the intrinsic properties had been estimated, the calibration program automatically saved a YAML file containing all relevant calibration data that is easily read-in using existing OpenCV functions. The intrinsic parameters are listed in Table 5.1.

5.2 Relative State Estimation

The relative state estimation algorithm is shown as a process flow diagram in Figure 5.3. For each video frame, the state estimation begins with a vector of five or more image points,



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 5.2: Sample calibration images

Table 5.1: Camera calibration and distortion coefficients

Symbol	Value	Units
f_x	603.8306	mm
f_y	603.8306	mm
c_x	311.7872	pixels
c_y	217.6215	pixels
k_1	0.035975	–
k_2	0.186248	–
k_3	-0.838948	–
p_1	-0.005521	–
p_2	-0.006650	–

which may or not be pre-correlated by by the algorithm described in Section 4.3.1.

1. If the image points vector is larger than the number of LEDs on the leader (5), then points are "swapped", following the scheme described in Section 5.3.

2. Then, if the image points vector has not been pre-correlated, then the 2-D image points are matched to their respective 3-D points of the model geometry using the algorithm described in Section 5.4

3. Next, the EPnP algorithm is employed to solve for the pose estimate. The relative translation and rotation (in Euler angles) are computed, and the scaled reprojection error is determined.

4. If the reprojection error is less than a pre-defined primary error tolerance, then the algorithm returns immediately with the current state estimate. Otherwise, it is compared to a secondary error tolerance.

5. If the reprojection error is less than the secondary tolerance, then the current state is stored to memory for later comparison. Otherwise the algorithm continues.

6. If the maximum number of point-swaps and correlations has been reached for the size of the image points vector, then the algorithm continues. Otherwise it returns to the

beginning to compute a new pose estimate with the image points swapped and/or correlated in a different manner.

7. If the algorithm reaches the maximum number of swaps and correlations, and at least one state estimate was stored to memory after having a reprojection error less than the secondary tolerance, then the pose with the lowest reprojection error is returned. Otherwise, the algorithm fails and no state estimate is returned.

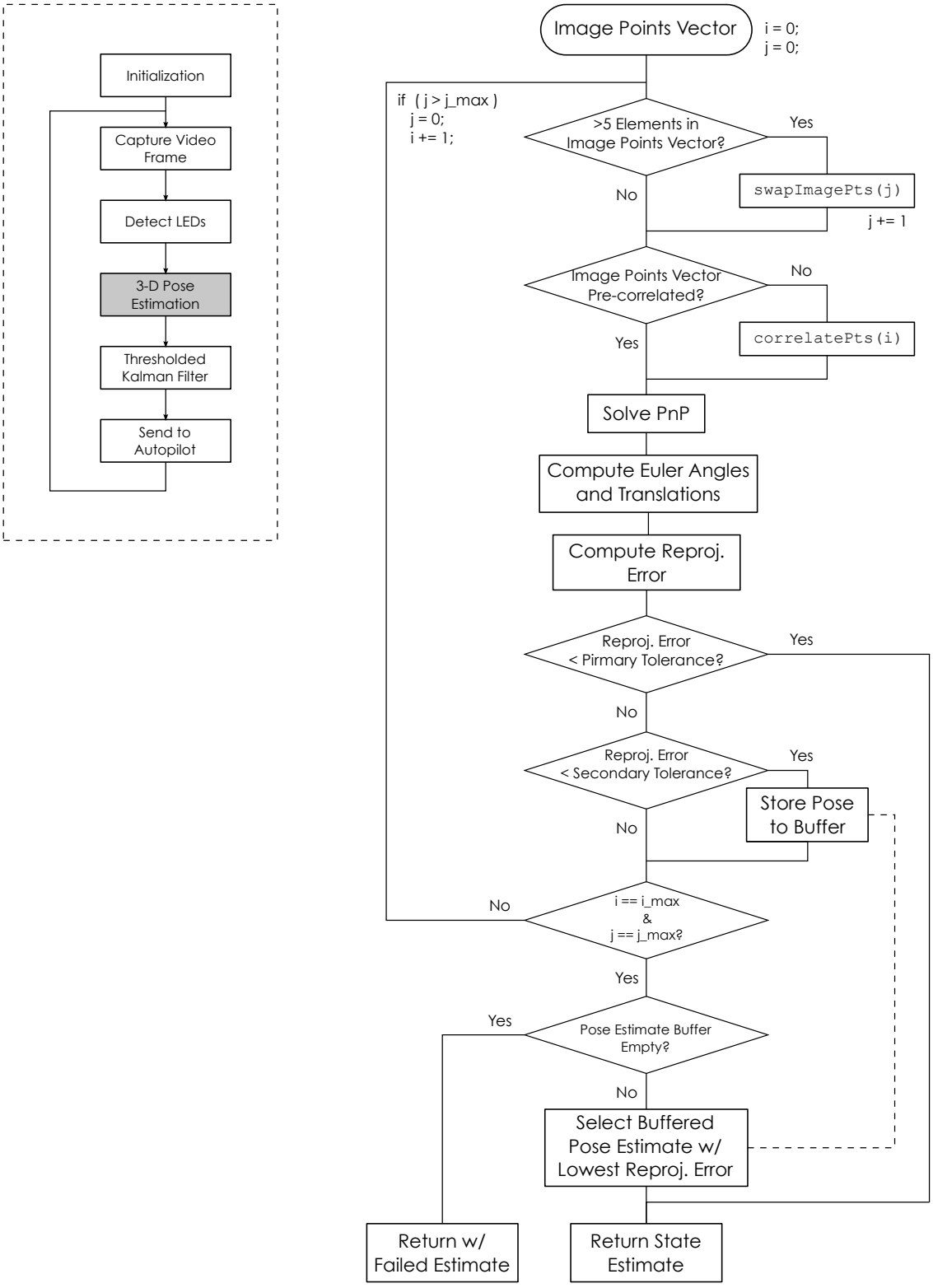


Figure 5.3: Relative state estimation process flow diagram

5.3 Image Point Swapping Scheme

Since the image points vector may sometimes contain more elements than there are LEDs affixed to the leader (5), it is sometimes necessary to "swap" among the provided image points to come up with exactly five points to be passed to the 2-D/3-D correspondence algorithm, followed by the EPnP solver. For each iteration, j, through the swapping scheme, a different subset of points is used. Table 5.2 summarizes how the set of 5 image points is selected.

Vectors are represented as *column* vectors with numbers to signify the *element* of the original vector. The zeroth iteration does not make any change to the vector. Circled numbers represent swapped elements with respect to the original image points vector. If there are not enough elements to perform the swap for the current iteration, then the maximum number of iterations, $j_{\max}$, has been exceeded.

Table 5.2: Image point swapping scheme

		Iteration (j)								
	0	1	2	3	4	5	6	7	8	9
Passed to EPnP	1	1	1	1	1	(6)	1	1	1	1
	2	2	2	2	(6)	2	2	2	2	2
	3	3	3	(6)	3	3	3	3	3	3
	4	4	(6)	4	4	4	4	(7)	4	(8)
	5	(6)	5	5	5	5	(7)	5	(8)	5
Not Used	6	(5)	(4)	(3)	(2)	(1)	6	6	6	6
	7	7	7	7	7	7	(5)	(4)	7	7
	8	8	8	8	8	8	8	8	(5)	(4)

5.4 2-D/3-D Point Correspondence Algorithm

Having identified the image points of the LEDs and knowing the geometry of the leader's airframe is not sufficient to compute a pose. Each of the five 2-D image points must first be *matched* to their corresponding LEDs. For simplicity, the following numbering convention will be used to identify the LEDs:

1. Left Wingtip
2. Left Horizontal
3. Vertical
4. Right Horizontal
5. Right Wingtip

Since only five LEDs were used and the 3-D geometry is relatively simple, a custom algorithm was developed for correlating the LEDs. It was necessary to make a few reasonable assumptions in order to preserve the simplicity of the algorithm since computation time is critical whenever processing is done on an SBC in real time. The resulting algorithm is described by the following steps and illustrated in Figure 5.4:

1. Compute the mean image point from the five LED image points.

2. Identify the LED that lies the farthest Euclidean distance from the mean point and identify it as a wingtip (either LED#1 or #5).

3. Determine whether the LED identified in (2) lies to the left or right side of the mean point. If the LED lies to the left side of the image, it is LED#1; if it lies to the right, it is LED#5. *This makes the assumption that the relative bank angle between the leader and follower is less than* $90°$.

4. Determine the LED that lies the shortest Euclidean distance from the LED identified in (3) and identify it as the horizontal of the same side. (If LED#1 was identified in (3), then this LED is #2; if LED#5 was previously identified, then this LED is #4).

5. *Assume the point with the* ***farthest*** *Euclidean distance from the LED found in (3) is the opposite wingtip.*

6. Compute the angle made by the main wing in the image, $\arctan\left(\frac{\Delta v}{\Delta u}\right)$.

7. Compute the angles formed by drawing imaginary lines between the LED identified in (4) to the two remaining points and compare the angles to that measured in (6). The point that results in the angle closest to the angle formed by the main wing is the remaining horizontal.

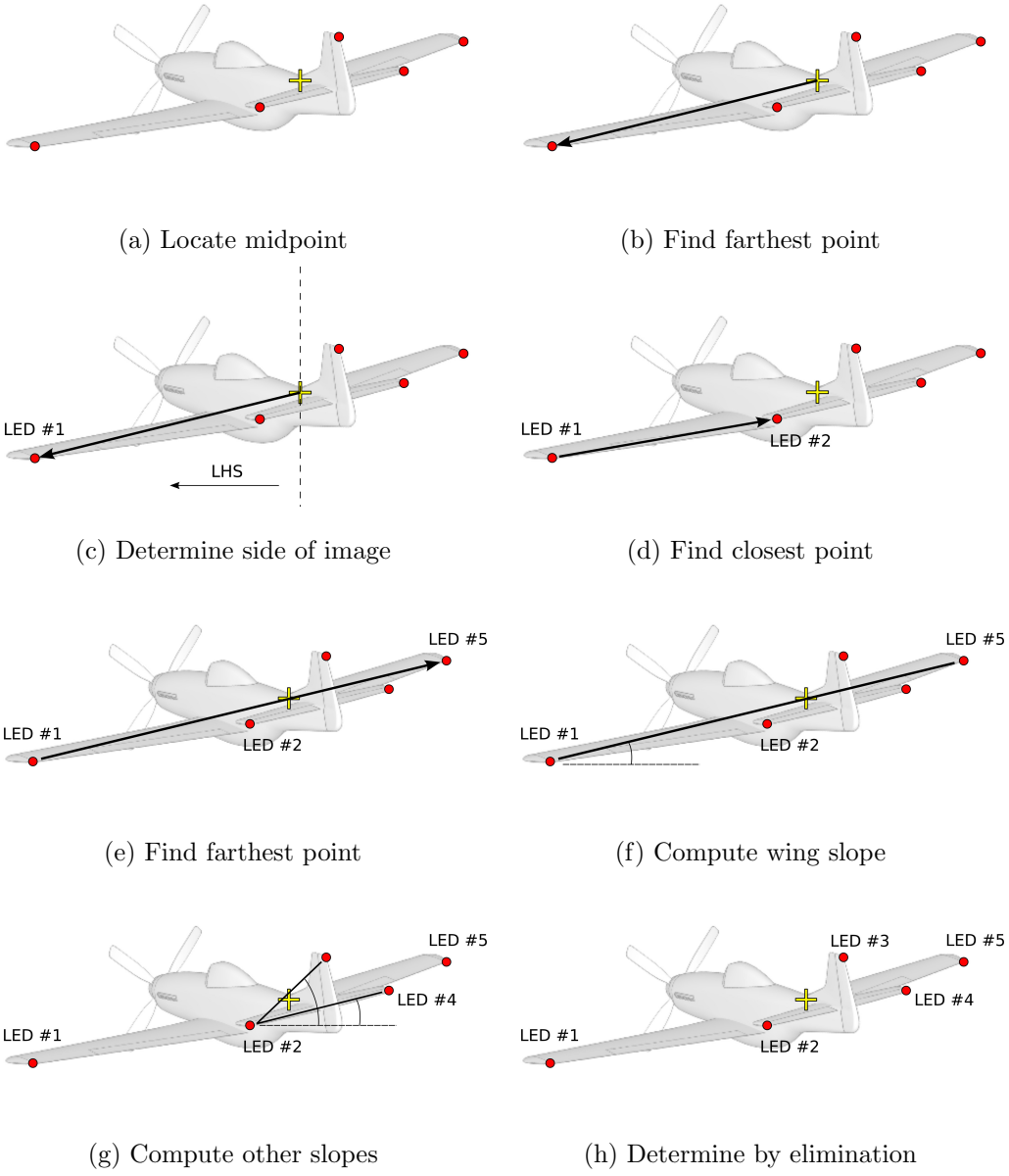
8. By elimination, the remaining LED is the vertical, LED#3.

9. The correlations are used to solve for the pose estimate, and the scaled reprojection error (reprojection error divided by the maximum distance between the five image points, in pixels) is computed.

10. If the scaled reprojection error is below a *primary* threshold value, the pose estimate is immediately accepted, and the algorithm returns. Otherwise, the threshold is compared to a *secondary* threshold value. If the error is less than the *secondary* threshold, then the pose estimate and associated error is stored to a memory buffer.

11. If the algorithm has not yet returned with an accepted pose estimate, the assumption in (5) is revised to use the *second farthest* point, and steps 6—10 are re-executed. If the

primary reprojection error tolerance has not been satisfied after the second iteration, the steps are repeated using the *thrid farthest* point. If, after the third iteration, at least one of the three iterations satisfies the *secondary* criteria, the pose with the lowest scaled reprojection error is accepted and the algorithm returns. Otherwise, the aircraft is considered to be in an infeasible orientation and the pose estimate is discarded completely.



(a) Locate midpoint

(b) Find farthest point

(c) Determine side of image

(d) Find closest point

(e) Find farthest point

(f) Compute wing slope

(g) Compute other slopes

(h) Determine by elimination

Figure 5.4: LED Correlation Algorithm

5.5 Outlier Robust State Filtering

During ground tests it was observed that, on rare occasions, the state estimation algorithm would return an incorrect pose due to reflections off of background objects being incorrectly interpreted as LEDs. This was especially true in bright outdoor environments with cluttered background scenery. The incorrect estimates typically only lasted for a duration of 0.5 seconds or less, but still raised concerns for the system's overall performance.

To mitigate this problem, two filtering techniques were used. First, the state estimate was compared to a set of limits to determine whether or not the state was feasible. For example, the relative roll was constrained to be within +/-90 degrees, since neither aircraft should be inverted during the maneuvers. Second, a *thresholded* Kalman filter was implemented, inspired by the work of Ting et al. [30].

Originally, a standard discrete Kalman filter was considered and tested offline on artificially generated sample data. (The prediction and update equations from [31] are summarized below for completeness.) However, depending on the choice of values for the process and measurement noise covariance matrices (Q and R, respectively), the outlying state measurements were either given too much weight in the Kalman update equations or the filter was overdamped and would have contributed to significantly lower control system bandwidth.

Time update equations:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{5.1}$$

$$P_k^- = AP_{k-1}A^\top + Q \tag{5.2}$$

Measurement update equations:

$$K_k = P_k^- C^\top \left(CP_k^- C^\top + R\right)^{-1} \tag{5.3}$$

$$\hat{x}_k = \hat{x}_k^- + K_k\left(z_k - C\hat{x}_k^-\right) \tag{5.4}$$

$$P_k = \left(I - K_k C\right) P_k^- \tag{5.5}$$

Next, the modified outlier robust Kalman filter presented in [30] was tested. The outlier

robust Kalman filter uses Bayesian weights and an Expectation-Maximization framework to "de-weight" outlying observations. However, the method was found to be overly complex and computationally expensive for this application. The algorithm requires the observation matrix, state transition matrix, and process and measurement noise covariances to be "learned" in real-time. The learning process requires a large number of sub-computations to be evaluated and stored to memory, ultimately making the method far more complex than this project requires.

Instead, a thresholded Kalman filter was ultimately implemented by adapting MATLAB code written by Ting for comparison with her own Bayesian weighted Kalman filter. The filter rejects outlying observations by thresholding on the Mahalanobis distance, which can be computed according to Eqs. (5.6) to (5.9).

$$S = CP_k^- C + R \tag{5.6}$$

$$r = \hat{z}_k - C\hat{x}_k^- \tag{5.7}$$

$$d = r^\top S^{-1} r \tag{5.8}$$

$$\text{Mahalanobis Dist.} = \left\| \left(r^\top S^{-1} r \right)^\top \left(r^\top S^{-1} r \right) \right\| \tag{5.9}$$

If the Mahalanobis distance of the new observation exceeds a certain threshold, then it is considered to be an outlier and is ignored in the Kalman update equations. As Ting points out, a disadvantage of the thresholded Kalman filter is that it must be hand-tuned manually. However, it was discovered that erroneous pose measurements have a *very* large Mahalanobis distance compared to more accurate observations, making the filter relatively insensitive to the choice of threshold value. Figure 5.5 shows the thresholded Kalman filtered state estimate for some sample data assuming a camera frame rate of 30 fps with 10 outlying observations, each lasting for a duration of 15 sequential frames.

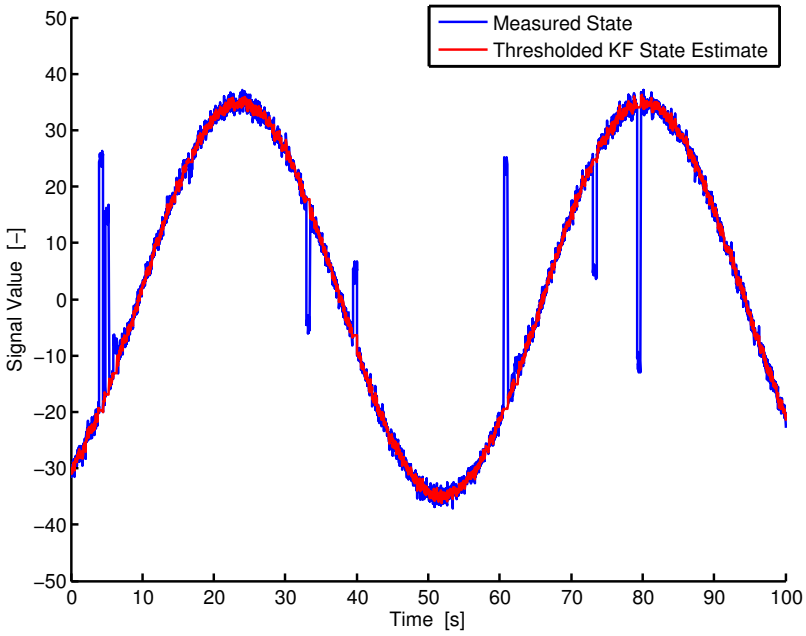


Figure 5.5: Thresholded Kalman filter applied to artificially generated sample data

Chapter 6

Autopilot Firmware Modifications

Since the standard release of the ArduPlane v2.68 firmware does not include formation flight capability, the source code had to be modified to include a custom flight mode. The firmware changes included: adding the ability to receive a relative localization solution over UART serial ports; modifications to the pitch, roll, and throttle PID controllers; the option to log additional data; and some additional changes specific to HIL simulation. This chapter will serve to provide a brief overview of the structure of the original ArduPlane firmware and to describe the changes that were made to incorporate formation flight capability.

6.1 Standard ArduPlane Firmware

Before discussing the changes that had to be made to the flight software, it is necessary to provide some background on the standard, unmodified ArduPlane firmware. This section serves to introduce the embedded PID control structure used by the software, describe how the software is implemented and executed on the APM board, and present the individual, unmodified PID controllers used for guidance and navigation. For a more detailed discussion of the ArduPlane project and its standard functionality, the reader is directed to Chapter 6 of Christian Lopez's thesis [19].

6.1.1 Embedded PID Control Structure

Version 2.68 of the ArduPlane firmware uses embedded Proportional-Integral-Derivative (PID) controllers for guidance, navigation, and control of the UAV. At the highest level, the

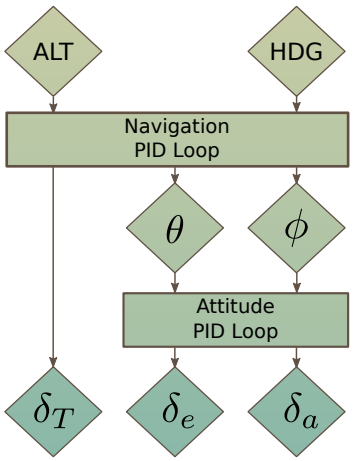


Figure 6.1: ArduPlane embedded PID control structure (adapted from [19])

navigation PID loop operates on a commanded heading and altitude to compute throttle, pitch angle, and roll angle commands. An *attitude* PID loop then computes the elevator and aileron control surface deflections, as seen in Figure 6.1.

Only some of the flight modes, however, use both PID loops. The `MANUAL` flight mode, for example, commands throttle and control surface deflections by direct passthrough of the transmitter signal. The `STABILIZE` and `FBW` (Fly-By-Wire) modes use only the attitude PID loop. Only the fully autonomous modes—`AUTO` and `GUIDED`—utilize the navigation loop in addition to the attitude loop.

6.1.2 Software Process Flow

The ArduPlane firmware runs on the APM board as four loops, shown in Figure 6.2: a fast loop (50 Hz), a medium loop (10 Hz), a slow loop (3.5 Hz), and a one-second loop. Within each iteration of the 50 Hz loop, one of the four control sequences from the 10 Hz loop is executed depending on the value of a loop counter, which is incremented with each iteration of the fast loop. Every fourth iteration of the medium loop, one of three control sequences is executed within the 3.5 Hz loop. Finally, a one-second loop is called at approximately 1 Hz.

The fast loop includes operations that must be computed at a high rate to ensure

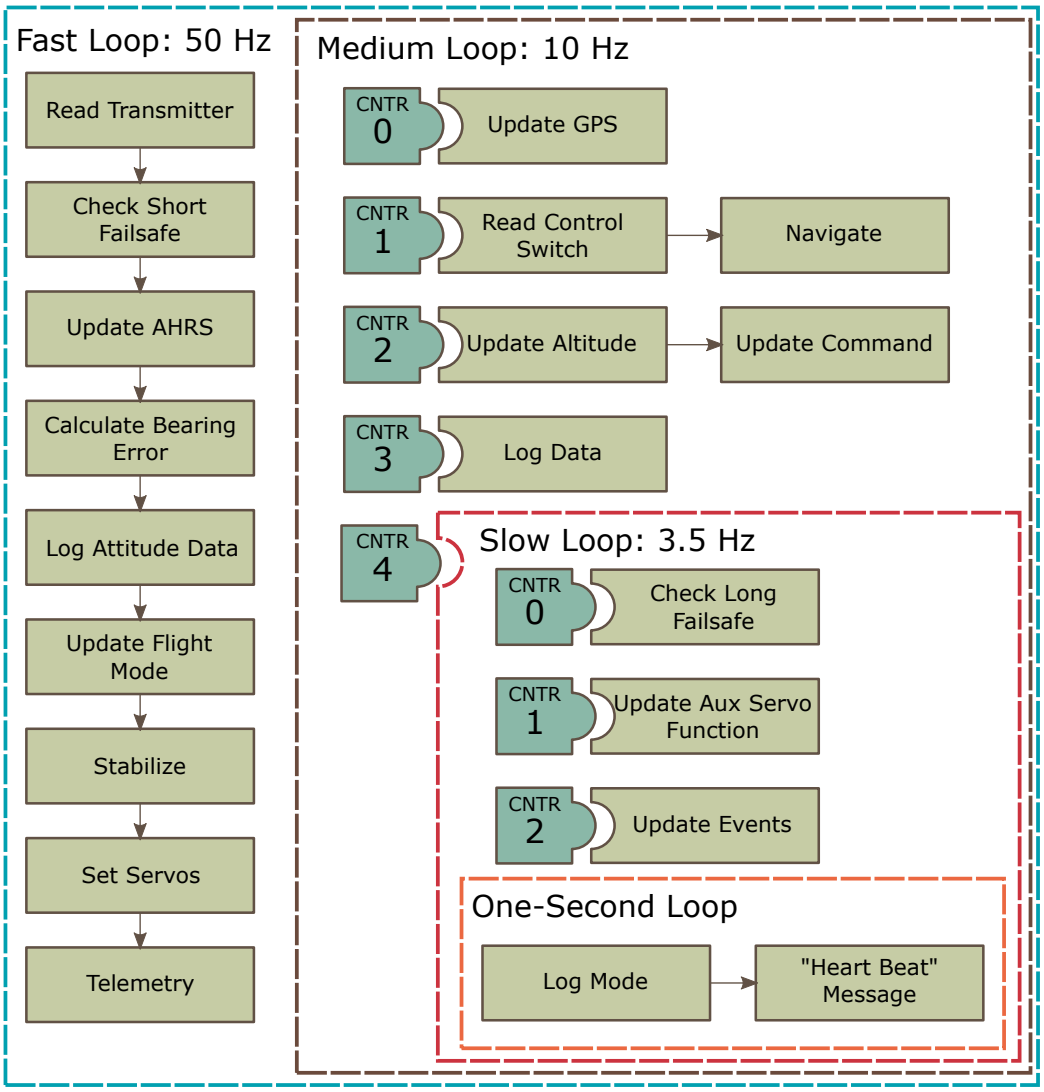


Figure 6.2: Standard ArduPlane firmware process flow diagram (adapted from [19])

desirable handling qualities and safe operation of the UAV. These include reading from the transmitter, updating measurements from the inertial sensors, logging attitude data, executing the next time-step for all PID controllers (occurs in the `update_flight_mode()` function), and setting the servos to their PWM values.

The medium loop is used to update measurements from sensors that require less bandwidth and less critical operations such as updating the GPS location and barometric altitude measurements, reading the control mode, updating the heading and altitude commands (occurs in the `navigate()` function), and logging navigation-related flight data.

The slow loop and one-second loops are only used to call non-critical functions such as logging the current flight mode and checking for the `HEARTBEAT` MAVLink message to ensure that the telemetry link (if any) is still connected.

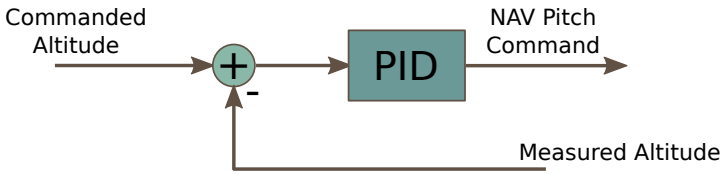
6.1.3 Pitch, Throttle, and Roll PID Controllers

Since nearly all significant changes to the ArduPlane PID controllers occurred in the navigation loop, only those controllers will be introduced in detail. The navigation loop is made up of three separate PID controllers: throttle, pitch, and roll. The architecture for each of the standard ArduPlane v2.68 navigation PID controllers is shown in Figure 6.3.

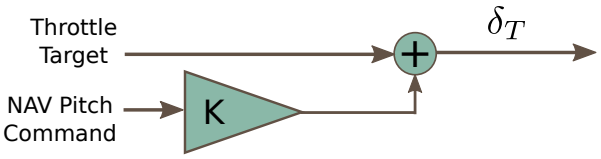
The pitch controller operates in either of two modes, depending on whether or not airspeed data is available. Since this thesis does not make use of an airspeed probe, only the controllers that do not require airspeed will be discussed. The navigation pitch controller uses a PID that operates on an altitude error. Commanded altitude is computed by linearly ramping the previous waypoint with the current waypoint based on distance. The GPS and barometric altitude are mixed as a weighted sum to generate the measured altitude, which is fed back to generate the error signal. The output of the PID is used as the "NAV Pitch" command, which in turn is constrained by a minimum and maximum pitch angle and passed to the attitude PID loop.

When airspeed data is unavailable (due to the absence of a pitot probe), a simple feedforward PID controller is used for throttle. The NAV Pitch command generated by the

Original Pitch Controller



Original Throttle Controller



Original Roll Controller

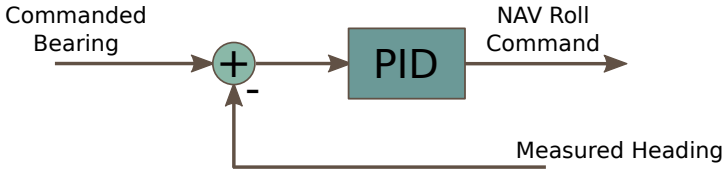


Figure 6.3: Original navigation controller architecture

above pitch controller is essentially multiplied by a gain and added to the target throttle. The controller can be analytically described by:

if NAV Pitch ≥ 0

$$\delta_T = \text{Thr. Target} + \Big(\text{Thr. Max} - \text{Thr. Target}\Big)\frac{\text{NAV Pitch Cmd}}{\text{Pitch Lim. Max}}$$

if NAV Pitch < 0

$$\delta_T = \text{Thr. Target} - \Big(\text{Thr. Target} - \text{Thr. Min}\Big)\frac{\text{NAV Pitch Cmd}}{\text{Pitch Lim. Min}}.$$

The throttle command is then constrained by a minimum and maximum allowable percent throttle before it is converted to a PWM value and sent to the electronic speed controller.

Finally, the lateral axes uses a PID controller that acts on a bearing error. The commanded bearing is a function of the *absolute* bearing from the aircraft to the target and the crosstrack error. The aircraft's heading, as measured by the compass, is subtracted from the commanded bearing to generate the error signal and is passed through the PID controller to generate the "NAV Roll" command, which is then constrained by a maximum bank angle and fed into the attitude PID loop.

6.2 Modified ArduPlane Firmware

Whenever possible, efforts were made to utilize as much of the existing ArduPlane source code as possible while still maintaining modularity by taking advantage of object-oriented programming. The ability to fly in formation was implemented by introducing a new "`REL_NAV`" flight mode, creating the `RelNAV` class, adding the ability to store new parameters to the non-volatile EEPROM, adding new information to be logged along with other flight data, and modifications to the navigation PID loops when the `REL_NAV` control mode is active. The modified structure of the source code can be seen in Figure 6.4 with added functions in red and modified functions in blue.

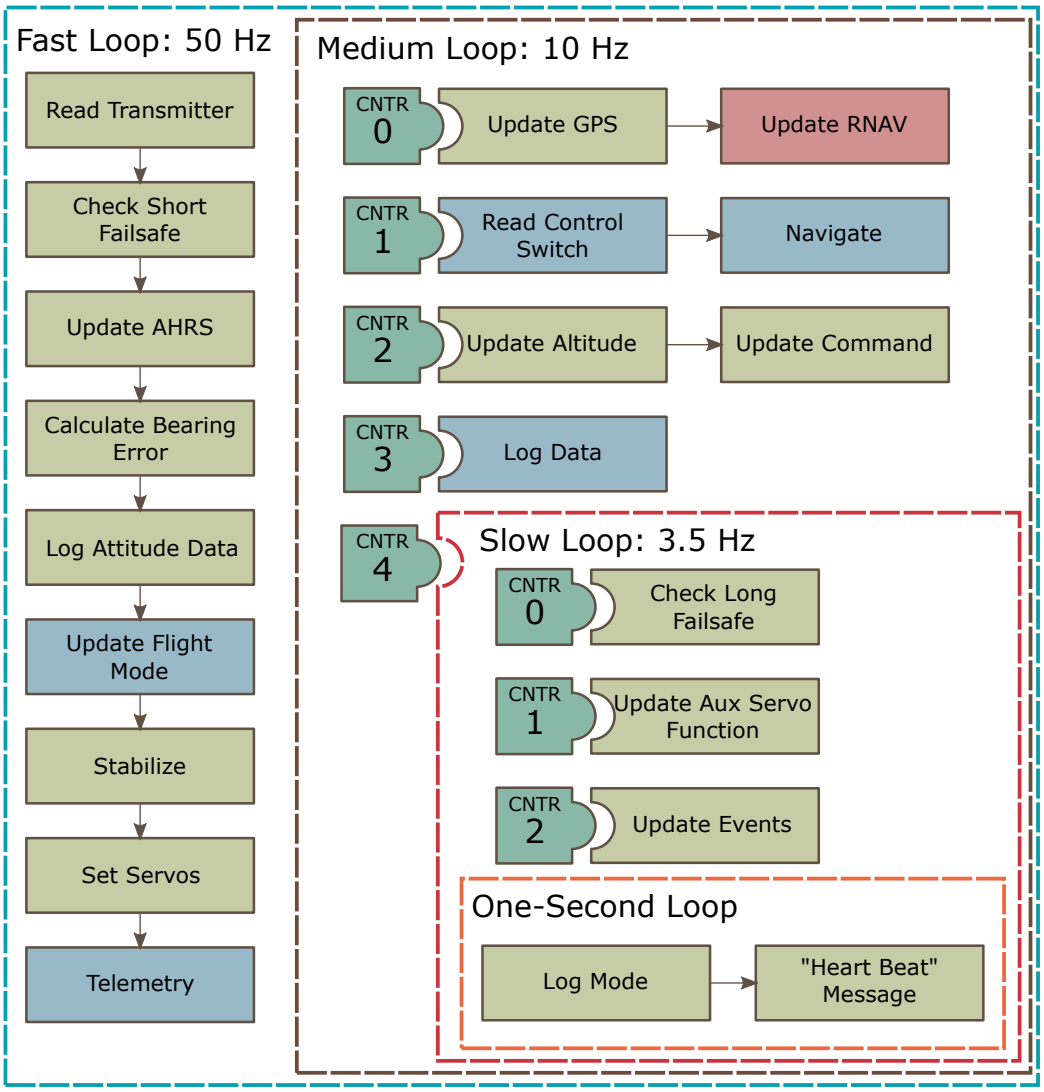


Figure 6.4: Modified ArduPlane firmware process flow diagram

The first change that was made to the flight software was to add a new flight mode for formation flight using relative navigation data. Doing so allows the RC pilot to activate the `REL_NAV` mode by flipping a switch on the RC transmitter to begin autonomous formation flight. Likewise, the pilot can easily switch back to `MANUAL` or another flight mode to regain control of the aircraft.

Since the new flight mode requires some new parameters to function properly (such as controller gains, and other options), the ArduPlane firmware was modified to be capable of reading/writing these additional parameters to and from the non-volatile EEPROM. This makes it easy to tune the new PID controllers by uploading custom parameter files to the board's EEPROM, either through a USB serial connection or even in-flight with a wireless telemetry link.

The `RelNAV` class was written to incorporate a number of new member functions and variables required for control using relative navigation information. By encapsulating as much of the new functionality as possible into a class, modularity in the code is preserved. The `RelNAV` class accomplishes the following functions:

- The `update()` member function reads a localization estimate from the vision subsystem or other relative navigation sensor over a UART serial port.
- The `updateDCM()` member function computes the direction cosine matrix required to rotate the localization estimate from the camera frame to the formation frame. The function then applies the rotation and computes the relative bearing to the target and separation distance between UAVs.
- When compiled in HIL Simulation mode, the class determines whether or not all LEDs are in frame.
- The class implements a zero-order-hold during failed localization updates. If a new estimate is not provided within a prespecified timeout period, steady-level flight is commanded.

In order to analyze and debug the newly implemented functionality, an additional

"RNAV" message was added to the flight log data. The RNAV message includes the raw relative navigation data received from the vision subsystem over serial as well as the inputs/outputs of the modified navigation PID controllers.

6.2.1 Pitch, Throttle, and Roll PID Controllers

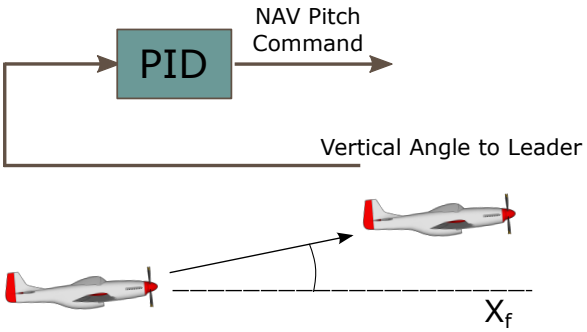
In order to make formation flight possible, the navigation PID controllers had to be modified to utilize the relative localization estimates from vision. Although a number of different control architectures were considered and tested in HIL simulation, only the final PID architectures will be presented here.

An important part of any control implementation is the choice of control reference frame. Typically, pilots flying in close formation (such as in-air refueling) think in terms of the lead aircraft's body frame and mentally decouple motion in separate axes. [22] However, since the ArduPlane software is well-established and this thesis seeks to utilize the existing software as much as possible, the formation frame described in Section 2.1.3 is used as the control frame. This choice minimizes the number of flight software modifications while still providing suitable performance so long as the two aircraft remain relatively level and maneuver slowly. In order to control using the formation frame, the (X_c, Y_c, Z_c) coordinates from the vision-based localization were rotated from the camera frame into the formation frame using mounting offsets known beforehand from calibration and the roll, pitch, and yaw angles measured by the inertial sensors aboard the follower.

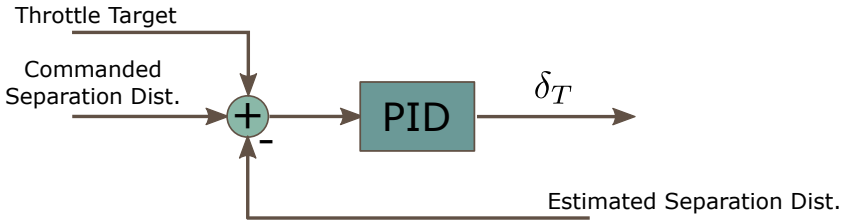
Since a pose estimate can only be computed when the leader is entirely within the frame of the follower's body mounted camera, the PID controllers were modified with the objective of keeping the follower's nose pointed at the lead aircraft while maintaining a safe separation distance. The modified controller architectures are presented in Figure 6.5.

The pitch controller was designed to act as a regulator by feeding back the vertical angle between the leader and follower. The vertical angle is measured as the angle between the relative translation vector between the two aircraft and its projection onto the X-Y plane of the formation frame. When the error is driven to zero by the PID controller, the leader is

Modified Pitch Controller



Modified Throttle Controller



Modified Roll Controller

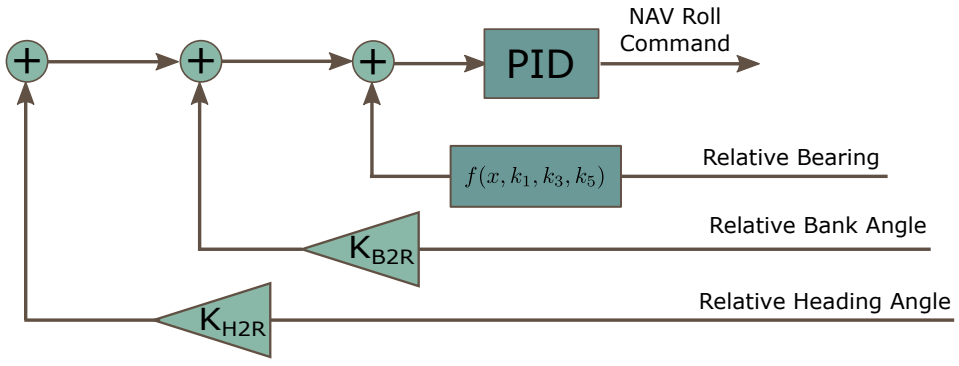


Figure 6.5: Modified navigation controller architecture

vertically centered in the camera's frame (assuming the follower's bank angle is zero). Since the leader is expected to fly at a relatively constant altitude, no attempt is made to utilize the relative pitch estimate provided by the vision subsystem to enhance performance.

The throttle controller was altered to maintain safe separation distance between the two UAVs. The separation distance is computed as the over-ground distance between aircraft. The measured separation distance is subtracted from the commanded separation distance (a constant parameter that is defined pre-flight) to generate a distance error. This error is passed to the PID controller which outputs the throttle setting as a percentage. In other words, if the UAVs are farther apart than desired, the throttle will increase; if the UAVs are too close, the throttle will decrease.

The roll PID controller was the most difficult to modify and required several design iterations until a suitable control architecture was found. At first, the roll controller was modified to drive the relative bearing $\left(\arctan \frac{Y_f}{X_f}\right)$ to zero. However, this approach becomes problematic in certain situations where the lead aircraft is to one side of the video frame but is banking the opposite direction. As illustrated by Figure 6.6, if the aircraft is on the left side of the frame, a negative bearing error is passed into the PID controller and a left bank is commanded. The lead aircraft will quickly travel off the right side of the frame since the aircraft are banking in opposite directions. This issue can be mitigated by feeding back additional relative navigation information that can be provided by the vision subsystem—namely the relative bank and heading, with respect to the follower's body frame.

Although increasing the derivative gain of the PID controller could theoretically improve performance without feeding back additional data, numerically computed derivative signals can be very noisy on embedded systems which have low floating point precision. Since the bank angle is related to turn rate, the relative bank angle acts as pseudo-derivative information. Likewise, the follower is better able to keep the leader in frame by attempting to match heading as well.

The roll controller was further improved by feeding back a nonlinear error signal based on the relative bearing instead of the relative bearing itself. It was observed through HIL

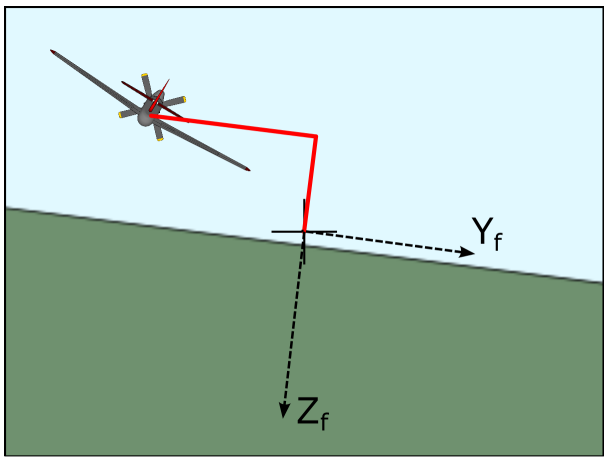


Figure 6.6: Incorrect bearing error

simulation that placing too much weight on the relative bank and heading angles would cause the formation to break up at greater separation distances. In order to de-weight the relative bank and heading angle contributions, the relative bearing error was fed back in a nonlinear form using the following polynomial equation:

$$f(x, k_1, k_3, k_5) = k_1 x + k_3 x^3 + k_5 x^5. \tag{6.1}$$

This approach effectively causes the controller gain to increase sharply when the lead aircraft nears the edges of the video frame while behaving approximately linear when the lead aircraft is near the center of the frame. This approach resulted in better lateral tracking during HIL simulation without inducing roll oscillations. The shape of the error signal can be visualized in Figure 6.7.

All three signals (relative bearing, relative bank angle, and relative heading angle) are fed back and combined as a weighted sum. The resulting error signal is passed into the PID controller to generate the NAV Roll command. As before, the NAV Pitch, throttle, and NAV Roll commands are all constrained by their respective minimum and maximum allowable values.

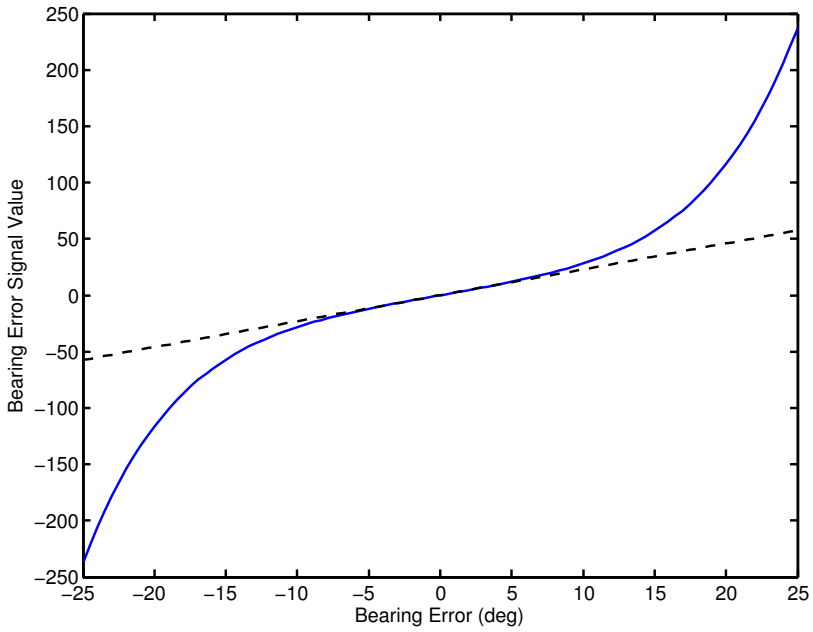


Figure 6.7: Nonlinear relative bearing error feedback signal

Chapter 7

Hardware-In-Loop Testing

The APM 2.5 board and ArduPlane firmware come standard with the ability to perform Hardware-In-Loop (HIL) simulation with X-Plane or FlightGear flight simulation software. In order to extend the simulator for relative navigation between two UAVs, a custom HIL simulation station had to be set up to simulate the output of the vision subsystem. The simulator was used to analyze frame rate requirements for the vision subsystem to ensure suitable flight performance. Additionally, HIL simulation proved to be a useful debugging tool by being used in parallel with autopilot software development to help troubleshoot code additions as they were added.

7.1 HIL Simulation Setup

7.1.1 Single UAV

The standard ArduPlane HIL setup is only intended for the simulation of a single aircraft and requires three main components: X-Plane, the Mission Planner software, and the APM board itself, connected as shown in Figure 7.1. X-Plane and Mission Planner can either be set up to run on a single computer or on separate PCs connected over a Local Area Network (LAN). X-Plane simulates the flight dynamics of the UAV, the APM board executes a subset of the flight software, and Mission Planner acts as a communications bridge between X-Plane and the APM hardware.

X-Plane simulates the flight physics using blade element theory, which essentially divides

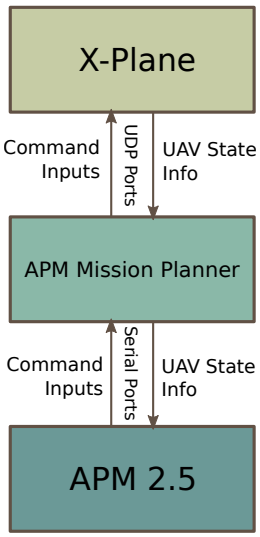


Figure 7.1: HIL simulation architecture for a single aircraft

the model airframe into many small pieces, computes the forces and moments on each element, and sums the contributions to solve for the resultant forces and moments acting on the model as a whole. The HiLStar17F model airframe, contributed by Mike Pursifull and open-sourced to the DIY Drones community, was used. [32] The HiLStar17F virtual aircraft aims to serve as a stand-in for Sky Surfer "like" RC aircraft. Due to the limitations of X-Plane, which was not originally designed for small aircraft weighing less than five pounds, the creator of the virtual model scaled the mass by three times and the physical dimensions by a factor of approximately 1.7 to preserve the same lift-to-weight ratio as the RC airframe that the HiLStar17F targets. The propulsion system was also scaled to mimic the power-to-weight of the target vehicle. Although the HiLStar17F simulator model has not yet been validated with actual flight test data, it has been developed and used extensively within the DIY Drones community by experienced RC pilots, giving some credibility to the accuracy of its handling qualities.

Mission Planner is necessary since X-Plane can only communicate simulated flight data over a LAN using UDP (User Datagram Protocol) ports and uses its own custom message protocol. X-Plane messages are received by Mission Planner, interpreted, and restructured into packets following the "MAVLink" protocol that can be understood by the autopilot,

Figure 7.2: HiLStar17F airframe in X-Plane 9

and finally transmitted to the APM board over a USB serial connection. Command inputs from the autopilot follow the reverse process through the Mission Planner to X-Plane to drive the control surfaces and throttle.

When the autopilot firmware is compiled in HIL mode, the autopilot receives simulated state information (global position and attitude) from the simulator over the USB serial connection. The APM also receives from the RC transmitter, exactly as it would in actual flight, allowing the user to switch flight modes or fly manually in the simulation environment. Since the autopilot does *not* receive simulated accelerations or roll rates from X-Plane, the sensor layer (comprising ≈25% of the total flight software) is not executed during HIL simulation. Because all of the high-level flight software is executed, HIL simulation is suitable for testing basic functionality and performance, however, it cannot be expected to capture subtle issues that might occur outside of simulation such as malfunctioning sensors, RAM overflow, etc.

7.1.2 Relative Navigation with Multiple UAVs

Extending the HIL simulation setup to work for relative navigation with multiple UAVs required the development of a more complicated network. Three computers were connected to one another over a single LAN: two PCs were dedicated to running X-Plane and Mission

Planner for each the leader and follower, and an additional PC was used to run a Simulink model acting as a relative navigation engine.

Leader/Follower Simulation Stations

The leader and follower simulation stations were set up similarly to the single-aircraft case, with a few minor modifications, as shown in Figure 7.3. In order to make the lead aircraft visible in the follower's instance of X-Plane and vice versa, the built-in multiplayer feature in X-Plane was used to establish communication between the separate instances of X-Plane over the LAN via UDP ports. In order to compute a relative navigation solution, a relative navigation engine had to be introduced that was capable of receiving state information from both instances of X-Plane over separate UDP ports. Since the Mission Planner requires the same state information from X-Plane as the relative navigation engine, and UDP ports are typically "port-to-port", a helper Python script was run on both the leader and follower simulation stations to split the input UDP port into two separate UDP ports that could be sent to the relative navigation engine and the Mission Planner.

Relative Navigation Engine

As previously mentioned, the relative navigation engine was developed in the form of a Simulink model. A simplified process flow diagram for the model is pictured in Figure 7.4. First, two "UDP Receive" blocks are used to receive simulated flight data from both instances of X-Plane. The messages are then parsed using X-Plane's custom message protocol to retrieve relevant state information as a vector that can be more easily manipulated within Simulink. The pertinent data includes the global position (X, Y, Z) in X-Plane's own Cartesian coordinate system and attitude (ϕ, θ, ψ) data, while any additional information sent by X-Plane is left unused.

The center of X-Plane's X, Y, Z coordinate system depends on the geographic region that scenery is currently loaded for and points in the North-East-Down directions, respectively. X-Plane's Cartesian coordinate system is preferred over geodetic coordinates because

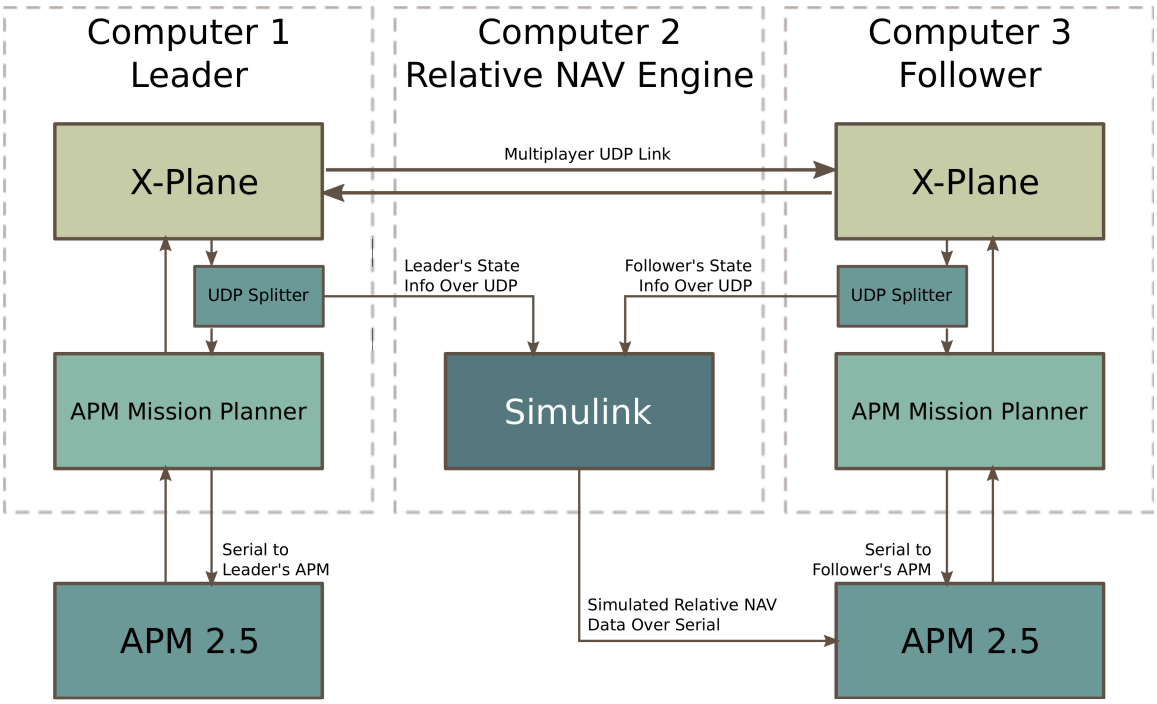


Figure 7.3: HIL simulation architecture for relative navigation

flight dynamic modeling and graphical rendering is done in this frame. Since our only concern is the *relative* position between the aircraft, this coordinate frame works well as long as the aircraft are close enough to reference the same coordinate system. (For small-scale aircraft operating in a confined area, the referenced coordinate system will be identical between the two UAVs.)

The appropriate coordinate transformations are then applied to compute the relative state vector of the lead aircraft with respect to the follower's body axes. A zero order hold block is used to simulate a frame rate delay and the output is fed to an LED projection model. The projection model determines the 3-D positions of the LEDs mounted to the leader's airframe and computes the projection of those points onto the image plane using the simplified pinhole camera model from Eq. (2.3). The camera model uses the focal length found from calibrating the actual webcam and assumes the principal point to be located at the center of the frame. Distortion effects are ignored by the model. Finally, the LED projection model determines which LEDs lie within the boundary of the image, and appends a bitmask to the state vector holding this information.

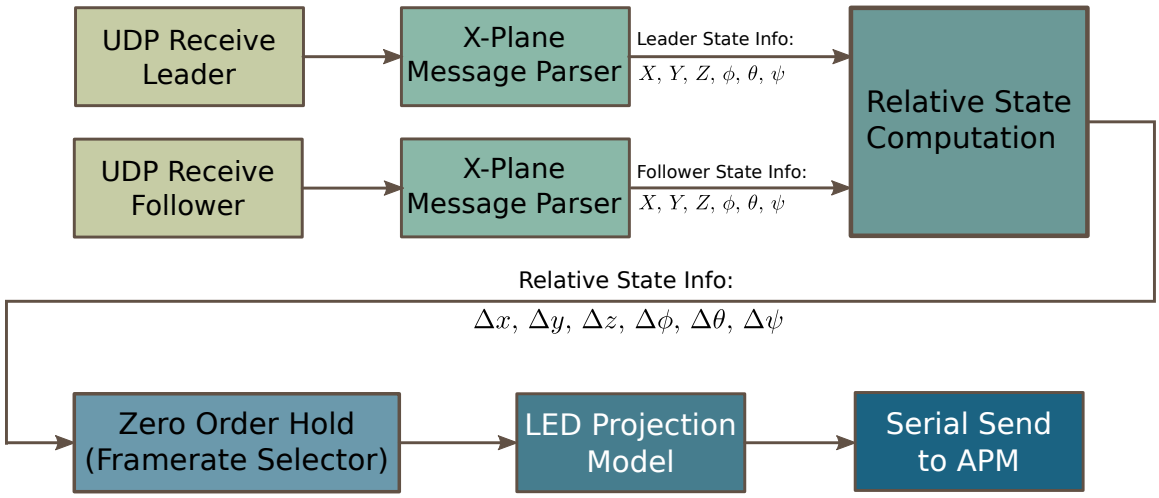


Figure 7.4: Relative NAV engine process diagram

The appended state vector is then passed to a "Serial Send" block that assembles a message using a custom protocol and sends it to the APM over a USB cable to the same Universal Asynchronous Receiver/Transmitter (UART) header pins that the embedded Linux computer connects to during actual flight. The modified software running on the APM then receives the simulated relative navigation solution, and checks the bitmask to determine whether or not all LEDs are visible in the frame. If so, the new information is used for navigation, otherwise the data is discarded and the previous solution is held until the next message is received.

7.2 Simulation Procedure

Once the HIL simulation network had been set up and the ArduPlane firmware had been appropriately modified, HIL simulation was used to evaluate the performance of the controller design and to derive approximate frame rate requirements for the vision subsystem. A set of test procedures were produced as well as a test matrix of various simulation cases.

7.2.1 Starting Situation

Every simulation was started by initializing the leader and follower UAVs to specific starting situations in the simulation environment at a position near the Cal Poly Educational Flight Range (EFR)—a remote control flight range that is owned and operated by the university. The UAVs begin at an altitude of 500 ft AGL to eliminate the need to takeoff and climb in each simulation case. Since the UAV dimensions are scaled by a factor of 1.7 in X-Plane, the commanded separation distance was also scaled by the same factor. (The follower was commanded to follow at a distance of 26 meters, or 85.3 feet, in simulation, which is equivalent to 50 feet in real-world scale.) At the starting locations, the UAVs begin in level flight and are separated by a distance of approximately 118 ft (≈ 70 ft real-world). The lead UAV begins in the approximate center of the follower's field of view to guarantee that the vehicles will form up correctly before entering the flight pattern.

7.2.2 Flight Patterns

Simulations were conducted using each of five different flight patterns: three circular patterns with a radius of 175, 100, and 50 meters and two figure eight patterns with 100 and 50 meter radius lobes. All circular patterns were flown counterclockwise. The patterns, shown in Figure 7.5, were aligned with the starting locations in such a way that the UAVs enter the pattern tangentially. Doing so promotes a smoother transition into the flight pattern from level flight.

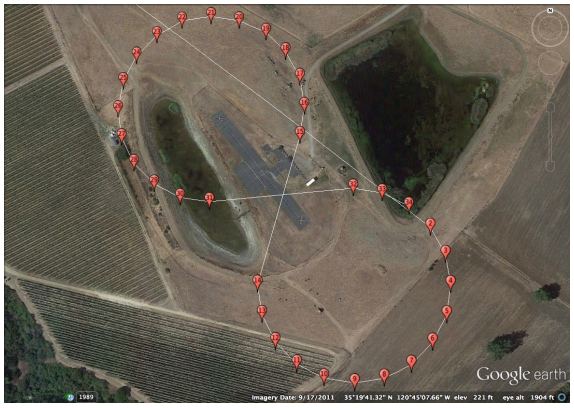
7.2.3 Procedures

Due to the high number of software and hardware components in the HIL simulation setup, it was necessary to establish a set of test procedures that could be followed with the execution of each simulation case. The procedures were used to ensure that all elements of the HIL network were communicating properly, all simulation cases were initialized to the predetermined starting point, and that flight data was recorded both on-board the APM and within X-Plane in a consistent manner. The following steps were followed during each

(a) Circles: 175m radius

(b) Circles: 100m radius

(c) Circles: 50m radius



(d) Figure Eights: 100m radius lobes

(e) Figure Eights: 50m radius lobes

Figure 7.5: HIL simulation flight patterns

run of the HIL simulator:

1. Startup all instances of X-Plane, APM Mission Planner, Python UDP splitter scripts, and Relative Navigation Engine Simulink block diagram.
2. Set relative navigation engine to desired frame rate setting.
3. *Pause* X-Plane and initialize both aircraft to initial starting locations.
4. Synchronize X-Plane times by enabling multiplayer link and setting environment date and time, then disable multiplayer link.[1]
5. Select waypoints for desired flight pattern and upload to leader through APM Mission Planner.
6. Clear APM log files through command line interface of APM Mission Planner and ensure correct logs are enabled for both boards (`ATTITUDE_MED`, `GPS`, `CTUN`, `NTUN`, `MODE`, `CMD`, `CUR`, `RNAV`).
7. Reboot APM boards through command line interface to reset current waypoint.
8. Establish connection between X-Plane and Mission Planner software by navigating to Simulation tab of APM Mission Planner and selecting "Start Sim", then connect to both APM boards by selecting "Connect".
9. Confirm that the leader has WP #1 set as the current waypoint by switching into `AUTO` mode. Confirm that follower is receiving relative navigation data over serial by switching into `REL_NAV` mode, verifying receipt of data using serial terminal, and switching back into `MANUAL`.
10. Begin logging data in both instances of X-Plane and confirm that log messages 1, 3, 8, 11, 18, 19, 20, 21, 25, and 26 are all being logged.
11. *Simultaneously* un-pause both instances of X-Plane while *also* switching the follower into `REL_NAV` mode. (Starting simultaneously makes it easier to synchronize flight data during analysis.)
12. Save all flight data (APM & X-Plane log files, APM param files) and record test notes.

[1] Although having the multiplayer link enabled allows for visual observation of both aircraft in X-Plane, it was discovered that having the multiplayer link active affected the flight dynamics of the following UAV in an adverse way. X-Plane attempts to model the effect of the leader's wake on the following UAV. Unfortunately, the effects were observed to be unrealistic. To maintian a better flight model, the multiplayer link was disconnected during simulated flight.

7.2.4 Data Reduction Techniques

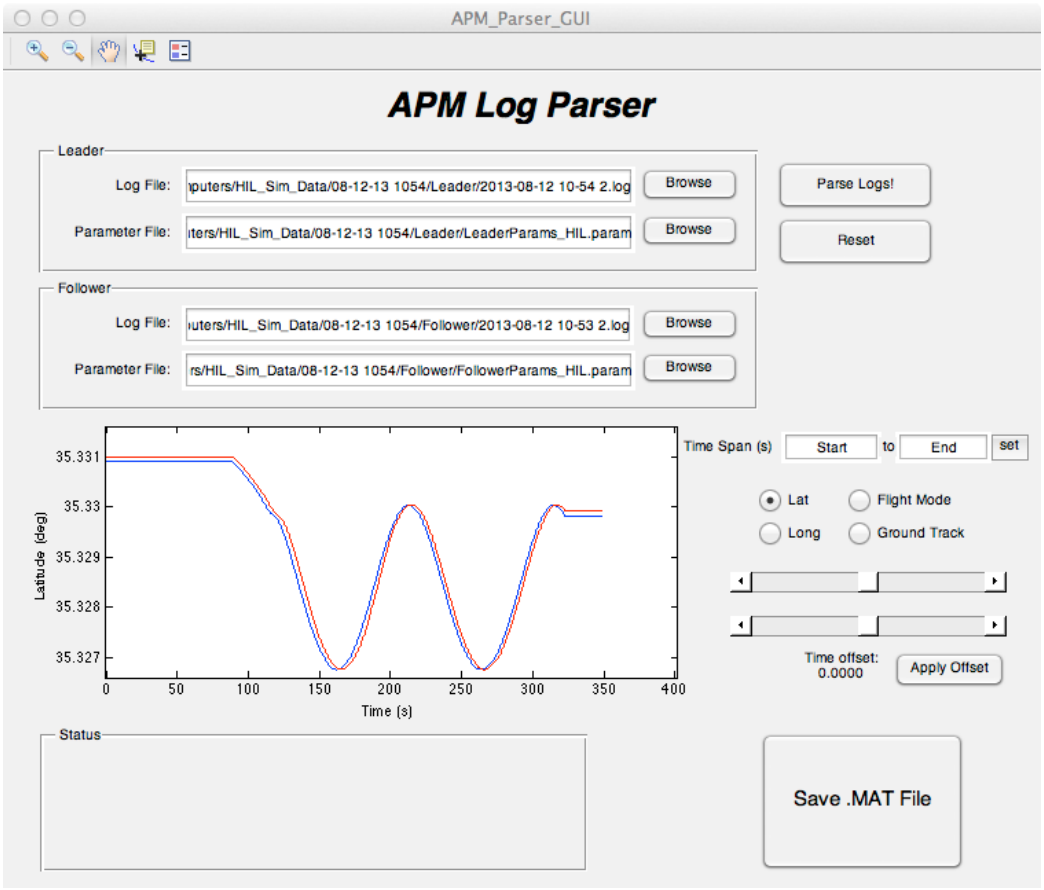
To simplify the repetitive process of reducing and analyzing flight data from the simulation run cases, two MATLAB Graphical User Interfaces (GUIs) were written. These GUIs, shown in Figure 7.6, made it possible to visualize the test data as it was manipulated, making it easy to confirm that data was being parsed accurately.

The first of the two GUIs, the "APM Log Parser", was used to read-in the ASCII formatted APM log files and parameter files for both the leader and follower aircraft. The log files contain information such as the GPS time and position, attitude, inner and outer loop controller command signals, the active flight mode, queued waypoints (leader only), and relative navigation information (follower only). The parameter files document the flight configuration—primarily controller gains, saturation limits, and target values. Once this data has been read-into MATLAB, the user can plot latitude, longitude, the flight mode, or the ground track of the vehicles. Using sliders built-into the GUI, the leader and follower's log files can be synchronized by shifting the signals in time. Once the logs have been aligned, the data can be trimmed and stored to a ".mat" file for future use.

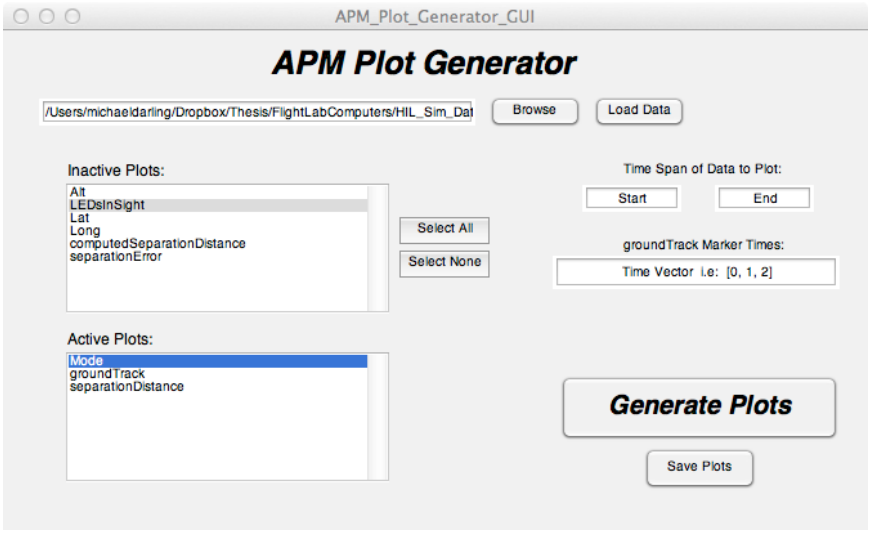
The second GUI, the "APM Plot Generator", was used to repeatably produce various plots from the logged data. The APM Plot Generator imports a ".mat" file generated by the APM Log Parser GUI, then the user can select any number of plots from a list and have the GUI generate those plots. When the user is satisfied with the plots, they can export all of the figures in Encapsulated PostScript (EPS) format to a new folder.

7.3 Results

Simulations were carried out for each of the five circular and figure eight flight patterns discussed in Section 7.2.2. For each pattern, the test was repeated with simulated vision update rates of 10, 7, 4, and 1 fps, in that order. Additional simulations were run with various levels of turbulence in an attempt to quantify the effect of atmospheric disturbances on the control scheme.



(a) APM Log Parser GUI



(b) APM Plot Generator GUI

Figure 7.6: MATLAB GUIs used for data reduction

7.3.1 Separation Distance Convergence

For a simulated formation to be considered successful, the follower should demonstrate the ability to maintain a predetermined separation distance from the leader. Since the two aircraft began every simulation with a separation distance *greater* than the target value of 85.3 ft (50 ft real-world scale), the vehicle separation should be observed to steadily *decrease* throughout the simulation and eventually converge to the target value.

Figure 7.7 shows the convergence of the leader/follower configuration for a sample simulation run in which the vehicles were flying the 100 meter radius figure eight course with a simulated frame rate of 7 fps and no wind. The separation is simply calculated as the norm of the 3-D translation vector between the two aircraft. The simulation begins when both instances of X-Plane were un-paused, approximately 50 seconds after the APM boards began logging flight data. It is interesting to note that the separation distance does not begin to converge until approximately 150 seconds into the simulation. This behavior can be attributed to the change in altitude of the UAVs, which is shown in Figure 7.8.

Due to the way the APM Mission Planner software writes waypoints[1], the lead aircraft is commanded to climb to an altitude 100 meters above its starting location by the time it reaches the first waypoint. In the time that the two UAVs are climbing as they approach the flight pattern, most of the follower's additional energy from increasing throttle goes into altitude gain rather than increased speed. It's not until the leader levels to hold constant altitude that the follower can begin to converge to the predetermined separation distance. After a transient time of approximately 30 seconds, the follower converges to and holds the target separation to within ± 10 ft in simulation (± 5.9 ft real-world scale).

While the throttle controller could benefit from improvements to better control vehicle separation with altitude changes, the observed behavior is suitable for this thesis, which only seeks to demonstrate the ability to hold formation at constant altitude.

[1]The APM Mission Planner redefines the "home" location to the current GPS position and altitude and writes waypoints 100 meters above the home location, unless the user specifies otherwise. In HIL simulation the "home" location was set to the starting X-Plane situation.

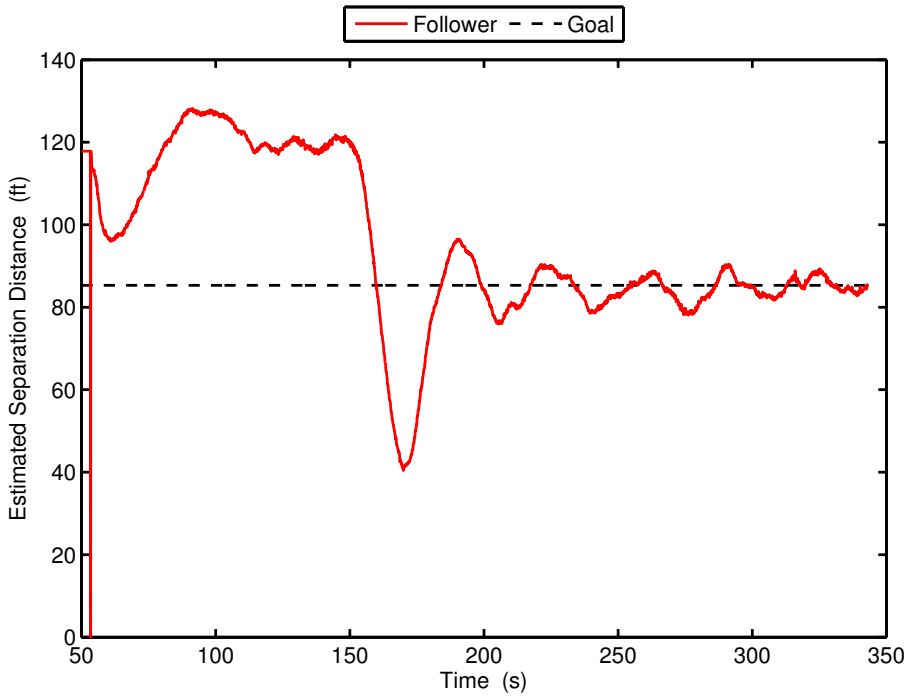


Figure 7.7: Vehicle separation distance convergence (100m figure eight, 7 fps, no wind)

7.3.2 Altitude/Pitch Tracking

Figure 7.8 illustrates the ability of the controller to track altitude changes, through example of the same simulation case as mentioned above. The follower climbs with the leader to 100 meters above the starting position, then levels out. The follower exhibits slightly larger and higher frequency oscillations in altitude compared to the leader. By the second oscillation, near the simulation time of 200 seconds, the altitude difference has converged to within ± 8 ft. These oscillations would have to be damped out in a very close formation or docking scenario, but are acceptable for the type of flight that is being considered here.

7.3.3 Lateral Tracking

In most simulations where the UAVs broke formation, it was determined that the leader fell out of frame due to insufficient lateral tracking. Failures were only observed at simulated

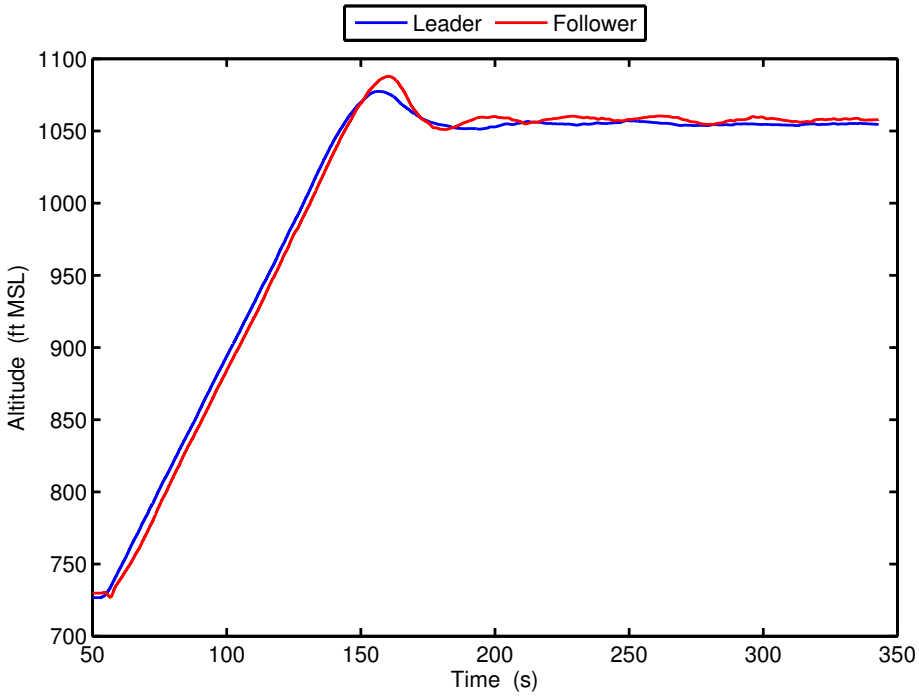


Figure 7.8: Leader and follower's altitude (100m figure eight, 7 fps, no wind)

frame rates below 10 fps and generally occurred during more aggressive maneuvers.

Figure 7.9 shows the ground track of the leader and follower UAVs for two complete laps of the 100 meter figure eight pattern with a simulated frame rate of 7 fps and no wind. Although the leader does not pass directly through the waypoints (and tends to fly *outside* of the turns), the follower tracks the leader quite well. In this scenario, the large figure eight pattern allows for gradual transitions between turns and does not require fast response by the following aircraft.

When the pattern becomes more aggressive, however, the follower is required to react more quickly to the leader's transitions from one tight-radius turn to another of the opposite direction. If the follower fails to keep the leader in sight of its body mounted camera, it no longer receives relative navigation estimates and after five seconds returns to wings level flight.

An example of a failed formation attempt is illustrated in Figure 7.10, which shows the

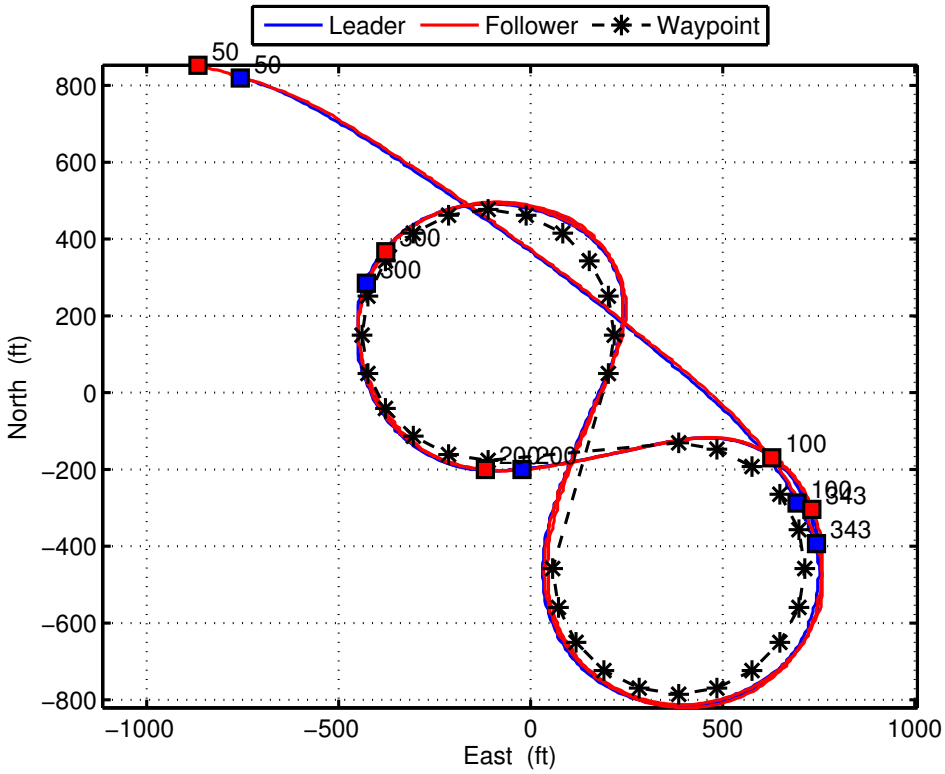


Figure 7.9: Leader and follower's ground track (100m figure eight, 7 fps, no wind)

ground track for a portion of the 50 meter figure eight flight pattern simulation with a frame rate of 7 fps and no wind. With the smaller figure eight pattern, the leader does a poor job of following the commanded waypoints, weaving far outside of the turn and turning back sharply to correct its course. Close examination of the simulated flight data revealed that at $t = 183$s, the lead aircraft had drifted to the far left side of the video frame as it entered the left hand turn of the figure eight pattern. In turn, the follower was commanded to bank hard left. The following aircraft overcorrected and eventually lost sight of the lead UAV as it passed through the lower right side of the video frame at $t = 187$s. The follower held this last known localization estimate, banking hard right in a nose down attitude. After 5 seconds without regaining sight of the lead UAV, the follower returned to wings level flight at $t = 192$s.

The roll oscillations that caused the formation to fail in this particular simulation case might have been reduced by retuning the controller gains for the 50 meter radius figure

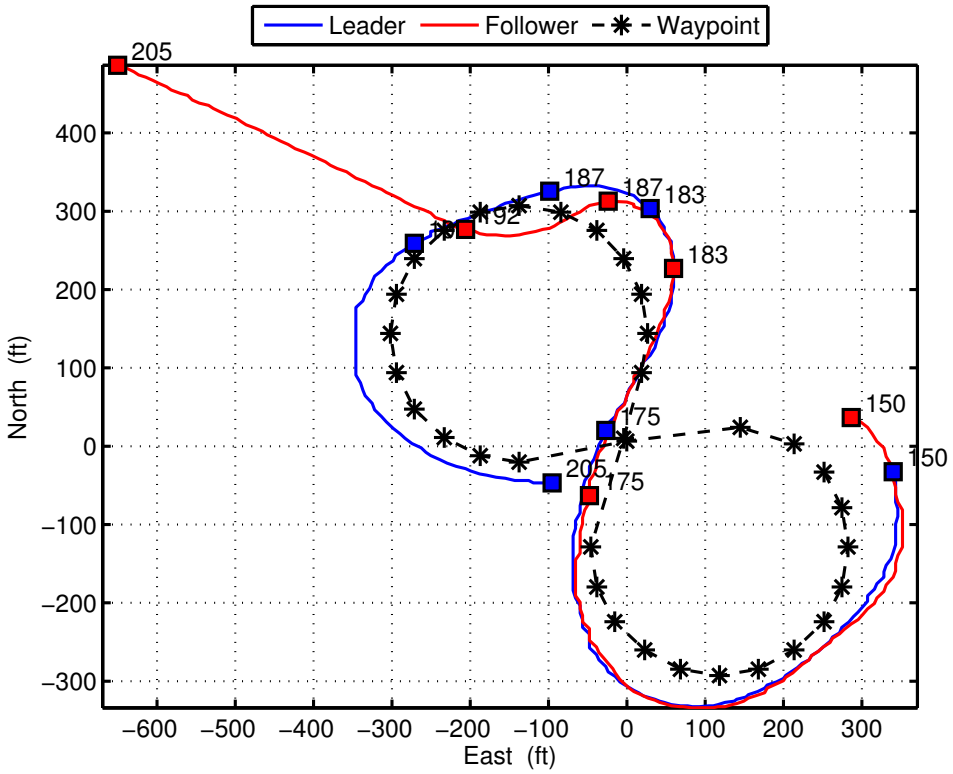


Figure 7.10: Leader and follower's ground track (50m figure eight, 7 fps, no wind)

eight course. However, the purpose of HIL simulation is to compare controller performance over a set of controlled variables. Using a different set of controller gains for each flight pattern would make it impossible to compare results and arrive at meaningful conclusions.

7.3.4 Summary of Frame Rate Tests

The first set of HIL simulation runs sought to investigate the effects of varying relative navigation updates on controller performance. The three circular and two figure eight flight patterns were each flown with with simulated vision update rates of 10, 7, 4, and 1 fps, in that order. Table 7.1 summarizes the results of the frame rate tests.

All three of the circular courses could be flown successfully with relative navigation updates as slow as 4 Hz with degrading performance (oscillations in bank) until eventual failure near 2 Hz. The larger figure eight pattern exhibited similar trends, but the 50 m figure eight pattern started to show degrading performance even at 7 fps.

Table 7.1: HIL simulation test matrix (no wind, no turbulence)

Pattern	No. Laps	Frame Rate (fps) 10	7	4	1	Notes
Circles 175m radius	2	✓	✓	✓	✗	
Circles 100m radius	2	✓	✓	✓	NT	
Circles 50m radius	3	✓	✓	✓	✗	Roll oscillations and occasional loss of LEDs at 5, 4, 3 fps Failure at 2 fps
Figure Eight 100m radius lobes	2	✓	✓	✓	✗	Slight weaving at 3 fps Failure at 2 fps
Figure Eight 50m radius lobes	2	✓	✗	✗	NT	Failure after some successful laps at 7 fps Failure at 5 fps

✓ = Sufficient, ✗ = Insufficient, NT = Not Tested

7.3.5 Summary of Turbulence Level Tests

HIL simulations were also carried out with various simulated turbulence levels and frame rates on the 100 meter radius circular course. The turbulence level was set through X-Plane's environment settings by adjusting a slider that ranges in value from 0–9. For each of three simulated frame rates (10, 7, and 4 fps), the simulation was initialized with the two UAVs at their starting locations with turbulence level (TL) zero. After approximately one minute, the simulation was paused and the turbulence level was gradually increased and then the simulation was resumed. This procedure was repeated until formation failure.

Since the X-Plane user manual does not specify how the 0–9 scale relates to real-world turbulence[2], samples of simulation data at various turbulence levels were compared to a sample of actual flight data that was acquired during a short test flight of a single UAV operating in the `STABILIZE` flight mode. The acceleration in the Z-axis of the vehicle's body frame was selected as the best gauge of atmospheric turbulence encountered by the UAV. Figure 7.11 shows the acceleration in the Z-axis of the simulated and flight test data versus time while Figure 7.12 presents the turbulence levels in the form of a periodogram using

[2]In addition, there was no clear connection between X-Plane's 0–9 scale and the FAA turbulence categories (light, moderate, severe, and extreme) or its United Kingdom equivalent, the Civil Aviation Authority.

Welch's power spectral density estimation method.

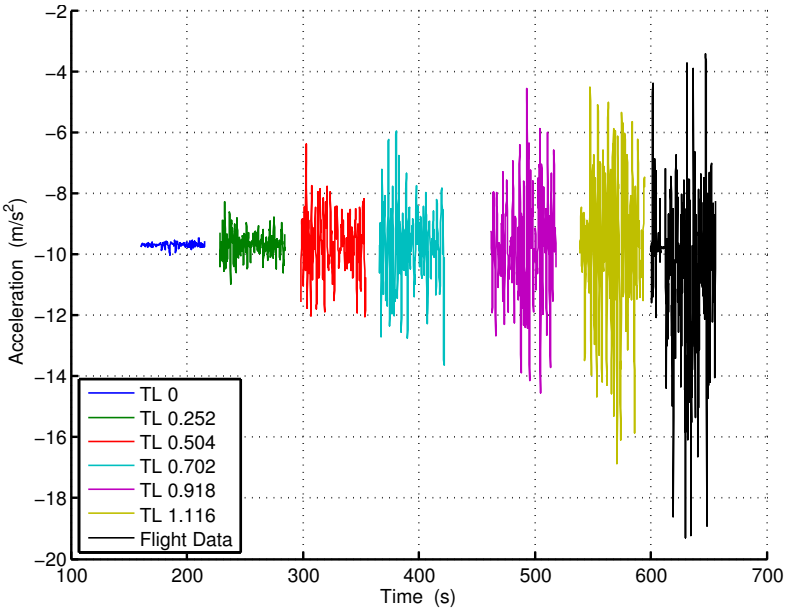


Figure 7.11: Turbulence time series

Although it is difficult to match the actual flight test data to a particular turbulence level, it is reasonable to conclude that the turbulence encountered in flight is comparable to the higher turbulence levels used in simulation. The real-world data has a higher signal power at higher frequencies and exhibits more extreme values, however this can be partly attributed to noisy sensor measurements which are not present in simulation.

Table 7.2 presents the simulation results for each frame rate and turbulence level on a scale of 0 to 9. Although the formation breaks up as low as TL 1.116 at the highest frame rate (10 fps), it is important to keep in mind that X-Plane was designed with the intention of simulating full-scale aircraft—not smaller scale R/C models, which can be affected by relatively light winds.

As expected, the simulations show that faster vision system localization updates allow higher intensities of turbulence to be tolerated, with the maximum tolerable turbulence level of 0.918 at 10 Hz. (Although the vision subsystem could theoretically provide navigation updates faster than 10 Hz, the system as a whole is limited by the navigation loop of the ArduPlane firmware which operates at 10 Hz.) Considering the fact that actual flight data

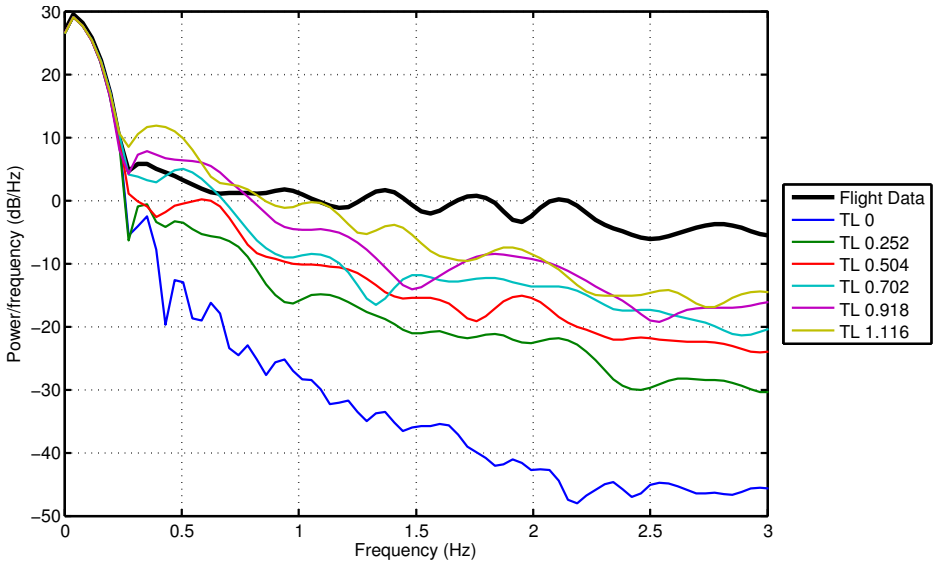


Figure 7.12: Turbulence power spectral density

Table 7.2: HIL simulation test matrix (no wind, simulated turbulence)

Pattern, Freq.	Turbulence level (0-9)						Notes
	0	0.252	0.504	0.702	0.918	1.116	
100m Circles, 10 fps	✓	✓	✓	✓	✓	✗	Immediate failure at TL 1.116
100m Circles, 7 fps	✓	✓	✓	✓	✗	NT	
100m Circles, 4 fps	✓	✓	✓	✗	NT	NT	

✓ = Sufficient, ✗ = Insufficient, NT = Not Tested, TL = Turbulence Level)

most closely resembles the highest simulated turbulence levels in Figures 7.11 and 7.12, the vision subsystem should operate at the highest frame rate possible, up to 10 fps, to maximize the likelihood of success in actual flight.

Chapter 8

Flight Test and Analysis

Flight tests were carried out at the Cal Poly Educational Flight Range (EFR)—an RC airfield surrounded by open space, suitable for operating unmanned aircraft within line-of-sight range. Flight tests were carried out with a minimum of three personnel: a pilot to operate each UAV and a test coordinator.

In order to reduce the likelihood of errors in the testing process, a flight test manual was written and used as a reference document for pre- and post-flight procedures. The manual, which appears in Appendix D, outlines the process of setting up the hardware systems, initializing the autopilot and vision computer, and downloading the log files and flight video recorded during each flight test.

8.1 Preliminary Flights

Before any attempts at formation flight could be made, a series of preliminary flights had to be performed in order to test the basic functionality and safety of the airframes as well as to tune the control gains of the autopilots. (Due to shortfalls in X-Plane's dynamic modeling of small-scale airplanes, the controller gains used in HIL simulation were relatively useless for real-world flight and had to be re-tuned.) These flights were carried out as follows:

1. Each the leader and follower were flown separately to verify their airworthiness and to adjust transmitter settings for trimmed flight. The aircraft were flown while equipped *only* with standard RC equipment to minimize the potential risk for collateral damage to other system components.

2. Autopilots were installed into each the leader and follower and the aircraft were flown separately to tune the inner-loop (stability) controller gains. This process involved flying in `Stabilize` and `FBW-A` modes and adjusting the PID gains remotely via the GCS until the aircraft were responsive to control inputs with sufficient damping to prevent any unwanted oscillations.

3. With autopilots installed, the leader and follower were flown separately to tune the outer-loop (navigation) controller gains. Waypoints in the shape of a large rectangular pattern were uploaded to each autopilot prior to flight. The UAVs navigated the prescribed flight pattern in the `AUTO` flight mode and the outer-loop PID gains were adjusted from the GCS until the track could be flown accurately without zig-zagging between waypoints.

Figure 8.1: Outer-loop gain tuning mission

8.2 Formation Flight Planning

Special care had to be taken while planning for formation flight attempts. Since the follower is only capable of localizing the leader while it appears within the camera's field of view,

rendezvous proved to be the most difficult task of flight testing. Coupled with the limited capacity of the flight batteries, every effort had to be made to streamline the pre-flight process and carefully orchestrate attempts to rendezvous.

8.2.1 Waypoints

The earliest flights sought to rendezvous by having the leader autonomously fly a circular pattern of waypoints and the follower manually flown behind the leader. It was quickly realized that, from the perspective of being on the ground, the pilot controlling the following UAV would not be capable of matching the leader's altitude and airspeed.

Instead, the rendezvous strategy was revised to have *both* UAVs autonomously navigate a series of identical waypoints. The waypoints were arranged into an oval "race track" pattern with semi-circular turns separated by stretches of straight-line flying, as shown in Figure 8.2. The course is approximately 1/2 mile end-to-end and 1/4 mile wide with the EFR runway located at its center. The straight portions provided a better opportunity for aligning the follower behind the lead UAV compared to turning flight. The follower's flight software was also updated to wirelessly transmit messages to the GCS describing the state of the vision system (whether or not the leader was successfully being tracked).

At the beginning of each flight, both airplanes would simultaneously take off under manual control, then at sufficient altitude the leader would be switched to `AUTO` to enter the flight pattern. After a brief pause, the follower would also be switched to `AUTO` so as to enter the pattern behind the leader. While in flight, either aircraft's flight path could be manipulated by "nudging" the control inputs prescribed by the autopilot or by briefly reclaiming manual control. Once tracking success was confirmed by the test coordinator, stationed at the GCS, the following UAV would be switched to the `REL_NAV` flight mode by its pilot, enabling the follower to be autonomously controlled according to vision-based localization alone. It should be restated that at no time did either UAV have situational awareness of the other (except for the follower localizing the leader through vision). All autonomous waypoint navigation was done absolutely using GPS and INS sensors.

Figure 8.2: Pre-Programmed Autonomous Waypoints

8.2.2 In-Flight Video Recording and FPV Equipment

A GoPro HERO 3+ camera (pictured at top of Figure 8.3a), was added to the following UAV to record high definition in-flight video at 720p resolution and 60 fps. Since the GoPro was used independently of all other systems, it simply had to be secured to the follower's airframe by means of an adhesive-backed mounting point. Footage from the GoPro was useful in synchronizing the leader and follower's log files with respect to time and reviewing the footage also served as a first-step in diagnosing the cause of eventual failure for each formation attempt.

In the final flights, a First Person View (FPV) system was introduced onto the following UAV to aid its pilot in rendezvousing with the lead aircraft. From a first person perspective, the pilot was much more successful in centering the leader within the camera's field of view and maintaining appropriate vehicle separation prior to initiating the `REL_NAV` flight mode. The FPV system included the PilotHD camera (720p, 30 fps), a 250 mW 5.8 GHz transmitter, and a Dominator VGA headset—all sold by Fat Shark RC Vision Systems. The PilotHD camera can be seen pictured in the middle of Figure 8.3a along with the transmitter antenna protruding from behind the GoPro.

(a) GoPro, FPV system, and webcam

(b) FPV goggles

Figure 8.3: First-Person-View (FPV) system

8.3 Formation Flight Results

Eleven flights were carried out, providing a total of 25 formation attempts, each summarized in Table 8.1. The longest formation lasted 1 minute, 29 seconds and covered more than one-half of a lap around the waypoint course. Throughout flight testing, no major crashes occurred. One minor accident was caused by an aileron servo failure on the following UAV upon takeoff which resulted in cosmetic damage only.

Without exception, formations eventually failed due to one of three main causes:

1. The follower was switched into `REL_NAV` at an inopportune time, with the leader near the edge of the camera's field of view. In this scenario, the large initial controller error induced divergent oscillations in roll and/or pitch until the leader was no longer visible within the camera's field of view.

2. The scene was backlit by either bright sunlight or light reflected off of clouds, causing the vision-based localization to fail. In these situations, the vision system was unable to correctly identify the five red LEDs in the image, making relative state estimation impossible.

3. The lateral control gains from Eq. (6.1) were not tuned appropriately for the separation between UAVs. Either the nonlinear heading-to-bank gains were set too highly for the target separation, or the separation error grew too large.

Upon reviewing the flight data and video from each flight test, areas for potential improvement were identified and minor changes were made to the system. For example, the lead UAV was painted matte gray to reduce the amount of light reflected from the top surfaces of its wings, improving the follower's ability to detect the red LEDs. When flying in `REL_NAV` mode, the follower exhibited high frequency oscillations in commanded throttle caused by the derivative component of the throttle PID controller. To mitigate this problem, a low-pass filter was added to reduce the high frequency noise. Parameters such as controller gains, target separation distance, and cruse throttle were also adjusted accordingly between flights.

Table 8.1: Flight Test Summary

Flight/ Attempt No.	Formation Duration (m:ss)	Reason for Failure
1.x	—	Failed to rendezvous with follower under manual control
2.1	0:06	Bad initialization
3.1	0:11	Roll oscillations due to choice of lateral controller gains
3.2	0:07	Scene backlit by clouds/sun
4.1	0:04	Bad initialization
4.2	0:26	Scene backlit by clouds/sun
4.3	0:21	Scene backlit by clouds/sun
—	—	Minor crash on takeoff due to aileron failure on follower
5.1	0:04	Bad initialization
5.2	1:16	Scene backlit by clouds/sun
5.3	1:23	Visual localization failed momentarily due to scene backlighting. When localization resumed, follower was too close to lead UAV and entered a stall causing reduced control authority. Leader exited the camera's field of view and formation could not be recovered.
6.1	1:23	Scene backlit by sun
6.2	0:04	Bad initialization
6.3	0:14	Scene backlit by sun
7.1	0:25	Follower was commanded to fly closer to lead UAV. Lateral control gains set too high for commanded distance, causing roll oscillations until leader finally exited camera's field of view.
8.1	0:04	Bad initialization
8.2	0:33	Roll oscillations until leader exited camera's field of view
8.3	0:07	Bad initialization
9.1	**01:29**	**Scene backlit by sun**
9.2	0:39	Scene partially backlit by sun caused an incorrect pose estimate, briefly, causing roll oscillations until formation failure.
9.3	**00:46**	**Leader made an aggressive right bank. Follower reacted appropriately, but entered divergent roll oscillations after the leader transitioned to a left bank.**
9.4	0:06	Bad initialization
10.1	0:16	Vision system failed to detect LEDs in presence of bright ambient light
10.2	**1:05**	**Vision system failed to detect LEDs in presence of bright ambient light**
11.1	0:51	Roll oscillations
11.2	0:09	Roll oscillations
11.3	0:12	Vision system detected LEDs poorly due to bright ambient light

Three specific formation attempts will be discussed in detail in this section—a completed 180° turn, a series of S-turns flown by manually controlling the leader, and a flight that demonstrates altitude tracking.

8.3.1 Completed 180° Turn Demonstration

The first flight to complete a 180° turn was also the longest duration flight (1 minute, 29 seconds) and appears as Flight 9.1 in Table 8.1. The eventual failure of the formation was caused entirely by scene backlighting from the rising sun, which appeared just above the horizon.

Figure 8.4 shows the ground track of the both UAVs, as recorded by their respective autopilots according to GPS data. The plot spans from $t = 603$ to $t = 689$ seconds, during which the follower was in `REL_NAV` mode. The first left-hand turn was made away from the sun and completed successfully. The follower suffered from some lightly damped roll oscillations after exiting the turn, but proceeded to track its leader along the straight portion of the course. After $t = 685$ seconds, the localization failed due to backlighting from the sun, and the follower attempted to continue navigating based on the last successful localization estimate.

Figure 8.4 suggests that, before entering the first turn, the follower was tracking to the inside (left) of the lead UAV. However, video evidence from the GoPro and stored images from the webcam (Figures 8.5a and 8.5b) indicate that the follower was actually tracking slightly right of center behind the lead UAV at the time `REL_NAV` was engaged. (There was negligible wind on the morning of the flight, discrediting the theory that the UAVs may have been crabbed into the wind.) This illustrates how poor GPS accuracy could potentially be a shortfall for precision relative navigation if using GPS alone.

After nearly $1^1/_2$ minutes, the formation eventually failed as the leader entered the next turn of the course, causing it to be backlit by the sun. Figures 8.5c to 8.5f illustrate how the leader was successfully localized just prior to the turn, but failed to be localized once backlit. (Figure 8.5f shows the last successful pose estimate reprojected onto the image,

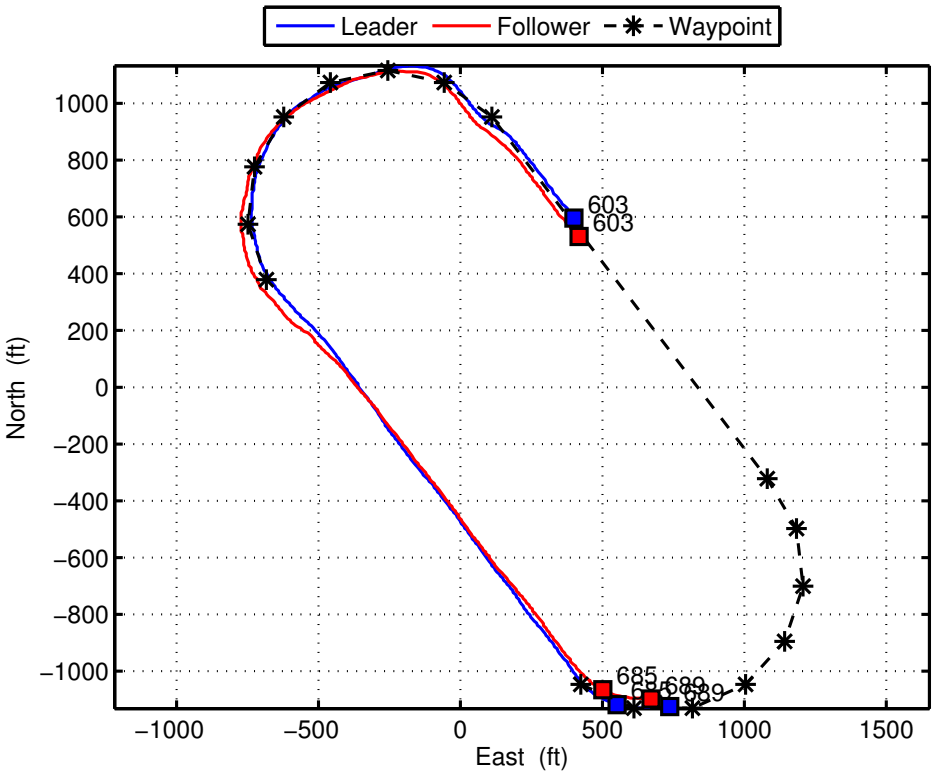


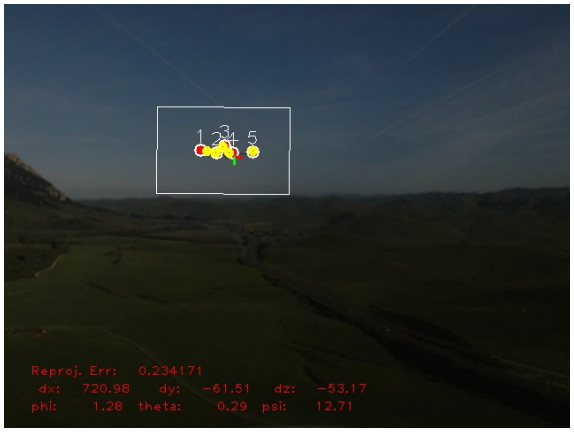
Figure 8.4: Ground track during 180° turn

which is held until a new estimate can be made.)

Figure 8.6 shows the separation distance (computed as the norm of the vision-estimated 3-D translation vector between the two UAVs) plotted against time. (Vision-based measurements were determined to be more accurate than GPS for measuring vehicle separation due to the relatively large magnitude of GPS error.) The target separation of 20 meters (65.5 feet) is held to within ± 15 feet for the entirety of the formation attempt. After $t = 685$ seconds, the leader could not be localized due to scene backlighting and the estimated vehicle separation is held constant until a new pose estimate can be made.

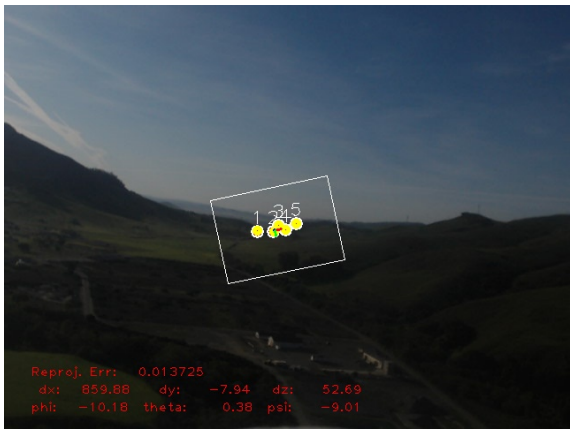
Figure 8.7 shows the altitude in feet AGL of both the leader and follower UAVs. Altitude is measured independently on each UAV using a combination of barometric pressure and GPS. Since altitude was measured absolutely, error in either UAV's measurement may contribute to the relative error with a compounding effect. Since the leader was commanded to hold constant altitude, it is difficult to discern altitude tracking from this figure. However from $t = 603$ to $t = 640$ seconds both airplanes generally climb, then descend until $t = 660$ seconds, followed by another slight ascent until the formation finally breaks down near $t = 685$ seconds.

(a) GoPro footage at $t \approx 603$s



(b) Webcam image at $t \approx 603$s

(c) GoPro footage while tracking



(d) PnP estimate while tracking

(e) GoPro footage during tracking failure

(f) PnP failure due to backlit scene

Figure 8.5: Flight video from 180° turn

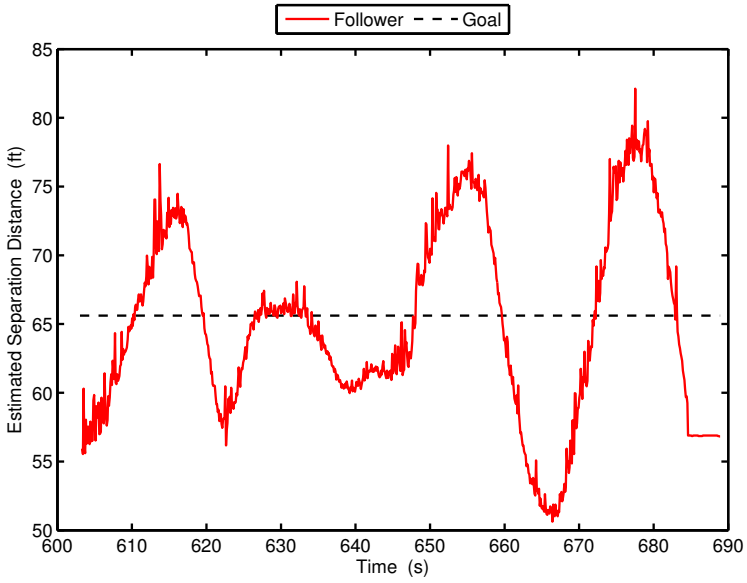


Figure 8.6: Separation distance during 180° turn

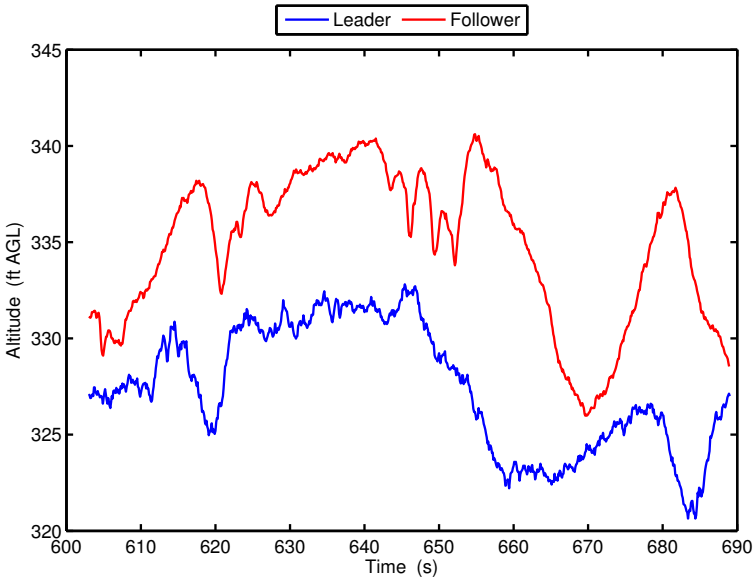


Figure 8.7: Altitude tracking during 180° turn

8.3.2 Shallow S-Turn Demonstration

In addition to flying in a "race track" pattern, an attempt was made to demonstrate formation flight during slightly more aggressive maneuvers in formation attempt 9.3. After rendezvous was accomplished successfully and the follower reached steady-state leader-tracking, control of the lead UAV was taken over by its pilot to fly a series of S-turns. The follower held formation through the S-turns until it eventually suffered from oscillations in roll after the leader made an aggressive right bank followed by another sharp left bank. The divergent oscillations eventually caused the leader to exit the field of view of the follower's camera, making recovery impossible.

Figure 8.8 shows the ground track of the two vehicles for the duration of the attempt. The marker at $t = 880$ seconds, identifies the approximate time that the leader initiated its sharp right-handed bank. Images from the flight video immediately before and just after the bank can be seen in Figures 8.9a to 8.9d. Figure 8.10 plots the follower's bank angle against time, providing further evidence that the roll oscillations were caused by the leader's aggressive maneuver. Near $t = 894$ seconds, the leader is lost from the field of view and was no longer able to be localized by the following UAV. Had the lateral controller gains been reduced, it is unlikely that these roll oscillations would have been so extreme and the formation might have been able to continue.

The vision-estimated vehicle separation is presented in Figure 8.11. When the `REL_NAV` flight mode is initially engaged, the separation distance is 95 feet (roughly 50% greater than the target value of 65.5 feet). The plot clearly illustrates vehicle separation convergence, though it is lightly damped. After 40 seconds the vehicle separation has converged to within ± 8 feet (about 12%) of the target value. Due to the break-up of the formation, it was not observed whether or not the system might have been capable of converging within tighter tolerances.

The altitude of both leader and follower are shown in Figure 8.12. Throughout the S-turns, the leader was able to hold constant altitude more precisely than in attempt 9.1, with maximum variation less than ± 5 feet AGL. Prior to the formation's failure, the follower

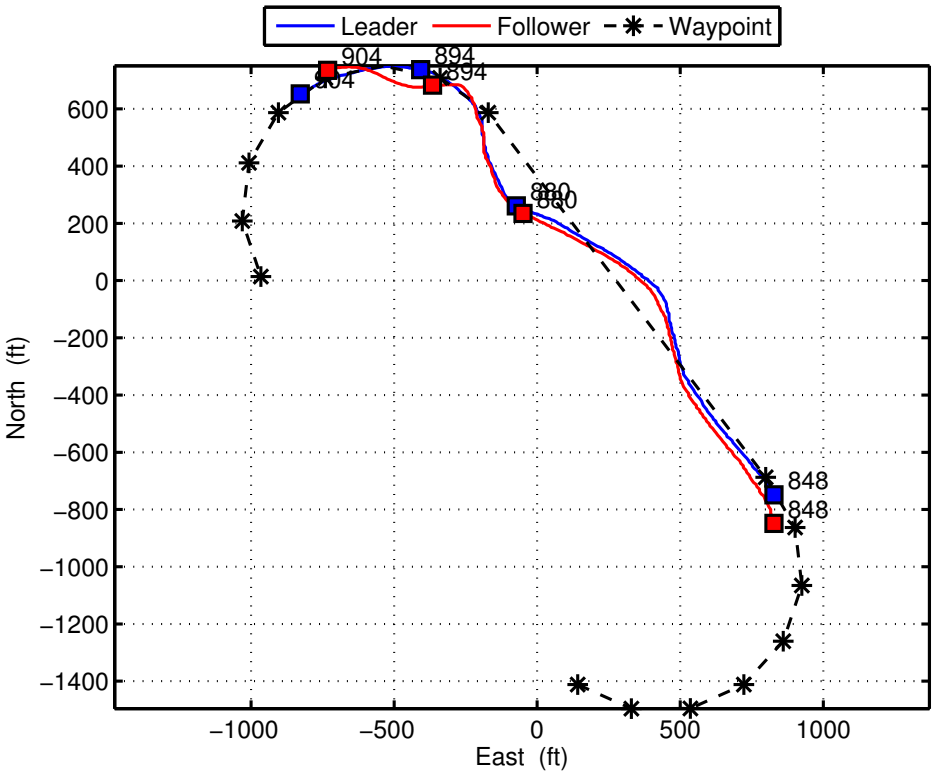
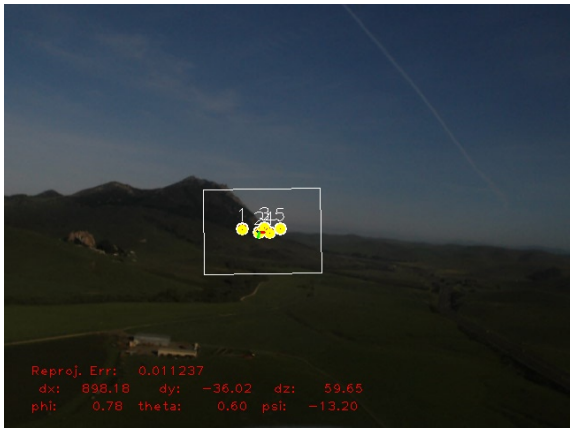


Figure 8.8: Ground track during S-turns

(a) GoPro footage before aggressive bank



(b) PnP estimate before aggressive bank

(c) GoPro footage during aggressive bank

(d) PnP estimate during aggressive bank

(e) GoPro footage during roll oscillations

(f) PnP estimate during roll oscillations

Figure 8.9: Flight video from S-turns

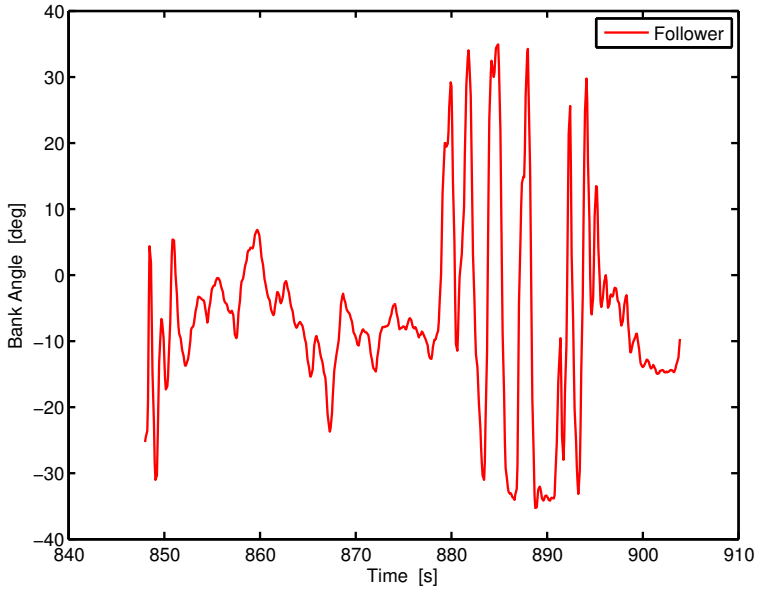


Figure 8.10: Roll oscillations caused by leader's aggressive bank

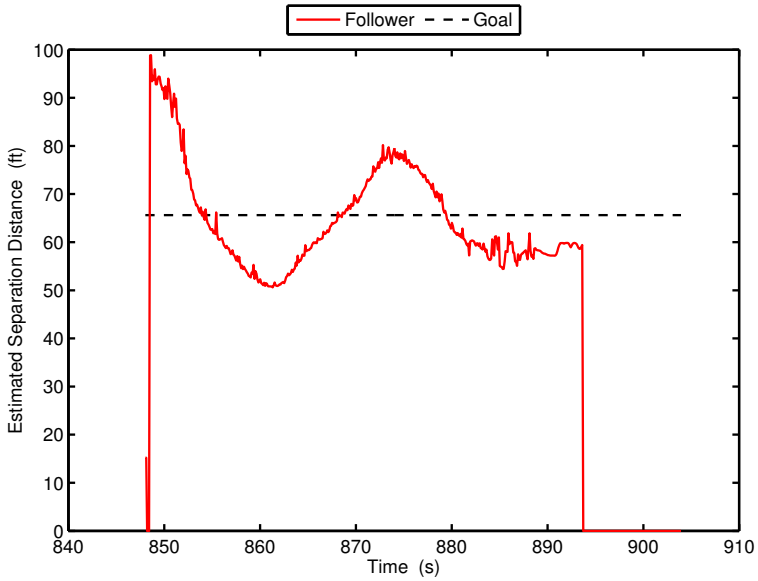


Figure 8.11: Separation distance during S-turns

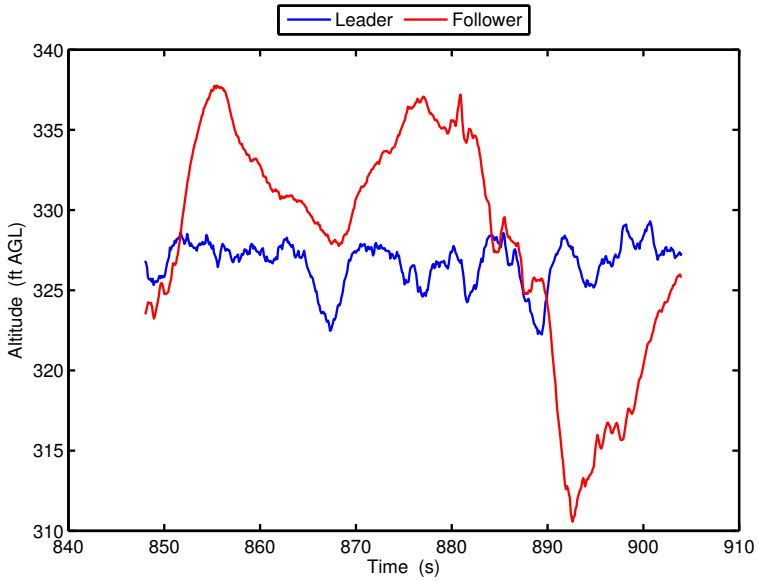


Figure 8.12: Altitude tracking during S-turns

matched the leader's altitude to within ±11 feet.

8.3.3 Altitude Tracking Demonstration

Flights 11 and 12 were added in a final attempt to demonstrate formation flight for multiple laps. The flights were carried out in the middle of the day, when the sun would be directly overhead in hopes that it would not interfere with the vision system's ability to track the leader. Unfortunately, rendezvous proved to be difficult even with FPV equipment due to moderate winds, intense ambient light from the midday sun, and light cloud cover. (White clouds disperse the sun's light across the sky and potentially into the camera's field of view.) The vision system had difficulty tracking the leader whenever flying towards the southeast, making a multi-lap formation impossible. Instead the flight time was used to demonstrate the ability to track changes in altitude by having the pilot of the lead UAV command a 50 foot climb by "nudging" the controls while in `AUTO`. Figure 8.13 shows the ground track of both UAVs while flying in formation. Once again, the large discrepancy the GPS-based ground track data and video recorded from the flight show the significance of GPS error.

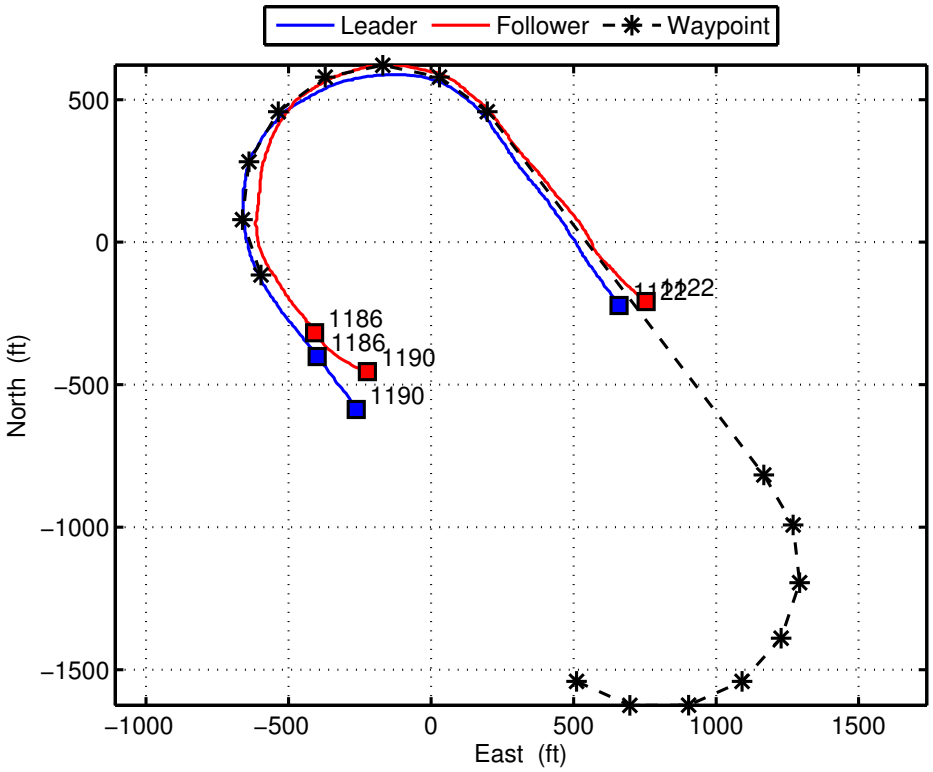


Figure 8.13: Ground track during ≈ 50 ft climb

Figure 8.14 presents video images from the final moments of the formation. The three image pairs were captured within roughly 1 second of each other. Figures 8.14a to 8.14d show that the vision system was tracking the leader while it was both above and below the horizon. However, in Figures 8.14e to 8.14f, the lead UAV could not be tracked. The exact cause of the localization failure is not known, but some observations suggest that the vision system was not appropriately tuned for the bright setting. From the flight data, the separation distance between the UAVs was observed to be increasing at the end of the formation attempt. As the aircraft grew farther apart, the LEDs were perceived by the follower with less intensity. The pixels making up the image of the LEDs may have fallen below the brightness threshold for the binarizing filter used by the feature detection algorithm, causing them to go undetected. Had the threshold value been better adjusted for the bright daylight, the formation might have been more successful.

Figure 8.15 shows formation convergence to within approximately ± 15 feet and Figure 8.16, plots the altitude of both UAVs. Throughout the turn (beginning at $t = 1140$ seconds), the lead UAV loses nearly 30 feet in altitude. At $t = 1160$ seconds, the leader begins to steadily climb until it has gained more than 50 feet in altitude, after which it immediately begins a descent at $t = 1184$ seconds. Even with these changes in altitude, the follower was successful in maintaining its position behind the leader. Only after the vision system failed to detect the LEDs did the formation finally break down.

(a) GoPro footage at end of climb

(b) Webcam image at end of climb

(c) GoPro footage at beginning of descent

(d) Webcam image at beginning of descent

(e) GoPro footage during localization failure

(f) Webcam image during localization failure

Figure 8.14: Flight video from 180° turn

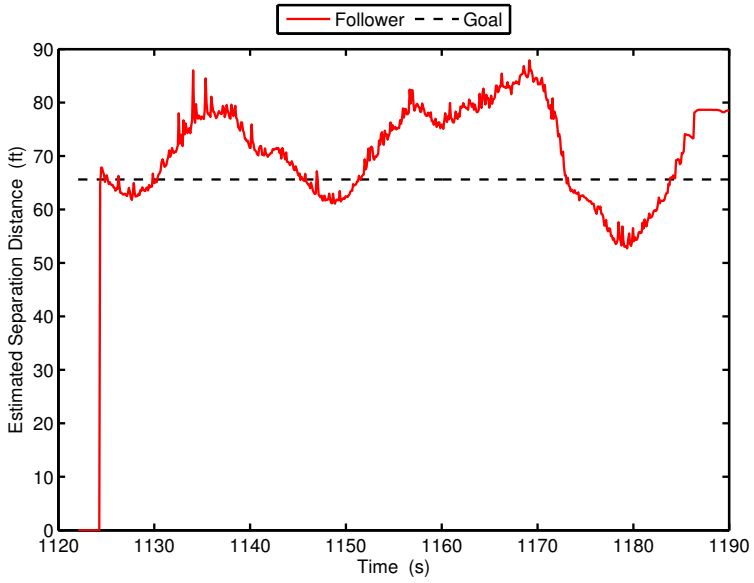


Figure 8.15: Separation distance during ≈ 50 ft climb

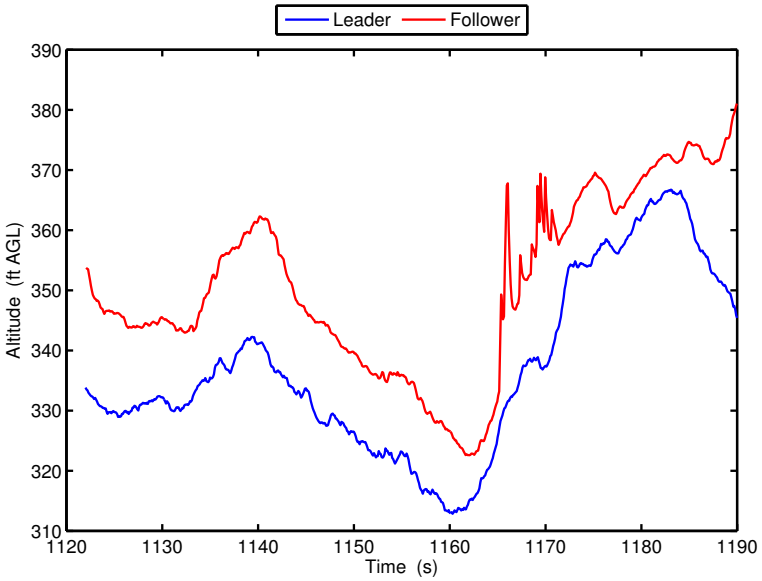


Figure 8.16: Altitude tracking throughout ≈ 50 ft climb

8.4 Comparison to Previous Works

To put the these results into perspective, they can be compared against previous formation demonstrations by other groups. Table 8.2 lists a number of different works along with their author, the formation/localization strategies employed, and the approximate vehicle convergence as demonstrated through flight testing (or in some cases HIL Simulation).

Formation attempts in this work demonstrated convergence between a leader/follower pair to within 65.5 ± 15 feet using vision-based localization alone. In contrast to Lopez's work, which made use of potential function guidance and GPS state-sharing over a peer-to-peer network, this shows a 43 percent reduction in vehicle convergence. A 68 percent improvement was made over McCarthy's simulations. His work, which originally sought to demonstrate leader/follower formation by using a portable GCS as a "wrapper" around a closed-source waypoint guided autopilot never made it to flight testing due to time constraints. In his HIL simulations, McCarthy used the GCS to receive telemetry data from both autopilots, compute the relative states of the UAVs, then dynamically update the follower's current waypoint goal. The only other project in this list to make use of vision-based localization is that of Mahboubi, et al., which never actually incorporated vision data into the relative state estimation during flight.

Table 8.2: Comparison to Previous Works

Author	Project Title	Formation Strategy	Convergence
Darling	**Autonomous Close Formation Flight of Small UAVs Using Vision-Based Localization**	**leader/follower vision**	**65.5 ± 15 ft**
Lopez	UAV Formation Flight Utilizing a Low Cost, Open Source Configuration [19]	PFG, GPS	115 ± 18 ft
Mahboubi, et al.	Camera based localization for autonomous UAV formation flight [20]	leader/follower vision+GPS	N/A
McCarthy	Characterization of UAV Performance and Development of a Formation Flight Controller for Multiple Small UAVs [18]	leader/follower GPS	207 ± 98 ft†

†HIL Simulation Result (not demonstrated through flight test)

Chapter 9

Conclusions and Recommendations

In this work, a vision-based leader/follower formation flight control system was designed, simulated, and implemented onto low-cost, open source embedded hardware for flight testing on remote controlled aircraft. Custom vision software was developed from the ground up to localize the lead UAV from video using *OpenCV* computer vision libraries, and was later uploaded to a *BeagleBone Black* embedded Linux computer. Version 2.68 of the waypoint guided *ArduPlane* autopilot software was modified to incorporate relative navigation capability.

The vision software was written around the EPnP algorithm, which provides an accurate 6DOF state of a 3-D object with known geometry from a vector of n 2-D image points. Feature detection was simplified by placing high-intensity red LEDs at predefined locations on the airframe of the lead UAV. By combining computer vision techniques such as binary thresholding, and contour finding, the LEDs could then be identified by the vision software and passed to the EPnP algorithm for relative state estimation.

Modifications had to be made to the autopilot flight software to make relative navigation possible. The PID controllers in the navigation loop were changed to operate on relative state information provided by the vision system when operating in the custom `REL_NAV` flight mode. The flight software also had to undergo changes to interface with additional hardware such as LED switching and communicating serially with the embedded Linux computer over UART.

A Hardware-In-Loop simulation environment was set up to serve as a debugging tool

during the development of the flight software and to derive some baseline expectations for the system's performance. The HIL simulation required two desktop computers running X-Plane and Mission Planner plus a third computer to simulate relative state information, all connected over a LAN network. Leader/follower formation convergence was demonstrated through HIL simulation to within 50 ± 6 feet.

Finally, the system was demonstrated through a series of eleven flight tests which provided 25 opportunities for establishing formation flight. Many of these attempts failed prematurely due to bad initialization with large initial controller errors. In other cases, the formations eventually broke down due to oscillations in roll or scene backlighting, which caused the vision-based localization to fail. Despite these challenges, flight tests did produce 1 minute, 29 seconds of contiguous formation flight and demonstrations of successful tracking through a 180° turn, a series of S-turns, and changes in altitude. Flight data provided clear evidence that vision-based localization provides a much higher degree of accuracy for relative navigation than GPS. The flight tests conducted as part of this thesis demonstrated vehicle convergence to within 65.5 ± 15 feet of one another using vision-based localization—a 43 percent improvement over the 115 ± 18 foot convergence achieved by former Cal Poly master's student Christian Lopez using GPS and potential function guidance on similar hardware, and an even greater improvement over the 207 ± 98 foot convergence of McCarthy's GPS-guided leader/follower demonstrations. [19, 18]

9.1 Lessons Learned

A wealth of experience has come through this project in the form of lessons learned. The highly interdisciplinary nature of this thesis has brought to light the many challenges of bringing hardware and software together to produce a working real-world system as well as the careful planning that must be done to orchestrate safe and meaningful flight tests.

Despite using widely supported, open-source, off-the shelf components wherever possible, hardware implementation posed some of the most daunting problems faced in this thesis. By far, integrating the USB webcam with the embedded Linux computer proved

to be the most frustrating of these issues. The USB webcam, which should have worked with the *BeagleBone Black* "right out of the box", took months to interface correctly. The cause of the problem was eventually traced back to a bug in the *BeagleBone*'s USB driver which caused extraneous interrupts to be sent to the CPU, severely limiting the board's USB bandwidth. Before the source of the problem was identified by a member of the online open-source support community, a workaround was devised with the assistance of other user/developers working with the *BeagleBone* for their own robotic vision projects. The details of the fix are outlined in Appendix E, which is meant to be an informal document for release to the open-source community. Ultimately, the issue was sidestepped by transferring MJPEG-compressed video frames rather than using the uncompressed YUYV pixel format, then using the SIMD (Single-Instruction-Multiple-Data) features of the board to accelerate image decompression. The vision software was also rewritten as a multithreaded application to put valuable CPU cycles to work by processing video instead of waiting idle for I/O from the USB webcam to complete. Although the webcam interfacing ordeal caused a frustrating delay to the project's overall progress, it showed the importance of having a good fundamental understanding of the hardware being used, including its capabilities and features as well as its limitations.

The LEDs required for localizing the lead UAV also posed unique challenges that were not easily addressed such as providing appropriate voltage to the LED assemblies by means of a DC/DC step-up converter, routing wiring throughout the airframe, and selecting a reasonably sized secondary battery with sufficient energy storage. The LEDs also had to be securely mounted to more fragile parts of the airframe without causing damage or adverse handling characteristics. Finally, the LEDs had to be made remotely switchable to avoid overheating while the aircraft was on the ground, were they could not be cooled by convection.

Some of the advantages and disadvantages of using vision as a localization strategy were clearly manifested through flight testing. Although relative state estimation from vision provided much higher accuracy than GPS, it was prone to premature failure depending on environmental conditions and a limited field of view.

Finally, rigorous preflight checks, ground testing, and careful electronics packaging were shown to drastically reduce the frequency and severity of crashes. These lessons, which Christian Lopez and others at Cal Poly have learned the hard way, were taken to heart in this work. Many hours were spent devising ways to securely package the various flight systems in such a way as to minimize the likelihood of electrical shorts or having components vibrate loose (see Appendix B). Also, any time changes were made to either the autopilot or vision software, the software was uploaded to the embedded hardware and tested on the ground in its full flight configuration prior to flight. After more than 11 separate flights and 25 formation attempts with only one minor accident, it is reasonable to conclude that these safety measures were a contributing factor to the overall level of system reliability.

9.2 Future Work

While this project has made some notable strides in the area of decentralized formation flight, there are many ways in which this work could be improved or extended. Due to the extensive scope of this thesis, it was often necessary to take an empirical approach to solving the various engineering problems in the interest of time. Should this work be picked up in the future, it is recommended that either the vision subsystem, control law development, or dynamic modeling of the aircraft be selected as an area of study in itself and explored in greater depth.

The most obvious place for improvement of the vision subsystem would be to enhance the robustness of the feature detection software. Improved algorithms for multiple target tracking and data association could be employed to more reliably detect and correlate the LED image points. Some level of trajectory estimation of the leader's state may also prove beneficial to the system's overall performance. Additionally, new strategies should be sought for detecting features even in unfavorable lighting conditions such as object backlighting. A color filter could be used with the camera to permit only a specific wavelength of light to pass through to the camera's imager—making the LEDs more apparent and therefore easier to identify. Another major limitation is the system's inability to localize a lead

UAV once it exits the field of view of the onboard camera. Possible solutions might be to increase the effective field of view by using an array of cameras or a gimbaled camera, which maintain the purity of decentralized formation flight. One might also eventually consider eliminating the dependence on bright LEDs for feature detection entirely. Reliable and accurate feature detection is a critical prerequisite to 3-D pose estimation, so this area demands more research before a definitive conclusion could be made about the viability of using vision-based localization in real-world applications.

Throughout the course of flight testing, it became apparent that the system's performance was highly sensitive to the choice of controller gains and the target separation distance. Often times, the follower suffered from roll oscillations that caused eventual break-up of the formation. Since the lateral control gains, especially, must be re-tuned according to the following distance, the system could benefit from a controller that is more adaptable to such changes. Modifications to the fundamental control laws could potentially reduce the large set of gains that require tuning through the time consuming and tedious process of trial-and-error. Perhaps more robust control approaches should be considered such as adaptive or LQR/LQG control.

Finally, more work should be done in the area of modeling the system dynamics. Having an accurate model would have made it possible to derive more meaningful system requirements from software-based or hardware-in-loop simulation. For example, framerate and pose estimation error requirements could be derived to satisfy a given level of performance. Having a dynamic model would also make it possible to develop control laws from a more analytic perspective.

Since rendezvous proved to be a major challenge, it would be interesting to bring together the lessons learned here with the work of Christian Lopez. GPS and peer-to-peer communication could be utilized in the far-field with vision-based formation flight taking over in the near-field. The marriage of these two control approaches would make tighter formations possible with a high level of robustness.

Hopefully, this work also sets the stage for future work in UAV formation flight and

robotic vision that expands far beyond simply improving upon what has been presented here. Despite the difficulties encountered with putting together a working vision system, the combined use of the *BeagleBone Black*, *OpenCV*, and a USB webcam served well for this type of application, where weight and volume constraints are severely limiting. This vision system could be adopted as a baseline hardware architecture for accomplishing other objectives like vision-based terrain avoidance using optical flow. Software could be developed to avoid collisions with other aircraft equipped with standard red, green, and strobe navigation lights. The vision system could also be modified slightly to accommodate a second camera for stereo vision, which has shown promise as an imaging technique for Simultaneous Localization and Mapping (SLAM). [33]

For those who wish to improve or expand upon the system presented here, all software is freely accessible in the form two Git repositories, listed below:

Vision Software:

`https://github.com/mdarling39/LinuxVision/tree/master`

ArduPlane Flight Software:

`https://github.com/mdarling39/APM-Vision/tree/master`

9.3 Closing Remarks

This experience has, in many ways, been a process of "Learn By Doing". However, it is also an example of learning from others. Although the results of this work may have surpassed those before it, the successes presented here were only made possible with the knowledge gained through those works, their authors, and the contributions of the open-source support community. Hopefully the lessons learned through this project will benefit future engineers in the same way, as they carry autonomous flight to new heights.

BIBLIOGRAPHY

[1] Glennon J. Harrison. Unmanned aircraft systems (UAS): Manufacturing trends. Technical report, Congressional Research Service, 2013.

[2] Federal Aviation Administration. FAA makes progress with UAS integration. Internet, May 2012.

[3] United States Navy. A-RFI request for information for unmanned carrier launch airborne surveillance and strike (UCLASS) system key capabilities, March 2010.

[4] Ben Iannotta. Vortex draws flight research forward. *Aerospace America*, pages 26–30, March 2002.

[5] Thierry Miquel, Felix Mora-Camino, and Karim Achaibou. A feedback linearizing controller for relative guidance between commercial aircraft. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, volume 1111, 2003.

[6] Jeffrey M. Fowler and Raffaello D'aAndrea. A formation flight experiment: Constructing a testbed for research in control of interconnected systems. *IEEE Control Systems Magazine*, pages 35–43, October 2003 2008.

[7] Masamitsu Tsuruta. An integrated formation flight algorithm via potential function guidance and biometrics. Master's thesis, California Polytechnic State University, San Luis Obispo, April 2008.

[8] Fabrizio Giulietti, Lorenzo Pollini, Mario Innocenti, et al. Formation flight control: A behavioral approach. *AIAA paper*, 4239, 2001.

[9] A. Dennis, J. Archibald, B. edwards, and D.J. Lee. On-board vision-based sense-and-avoid for small uavs. *AIAA Guidance Navigation and Control*, 2008.

[10] Brian C. Karhoff, Jason I. Limb, Shane W. Oravsky, and Alfred D. Shephard. Eyes in the domestic sky: An assessment of sense and avoid technology for the army's "warrior" unmanned aerial vehicle. In *IEEE Systems and Information Engineering Design Symposium*, 2006.

[11] Allison Ryan, Marco Zennaro, Adam Howell, Raja Sengupta, and J Karl Hedrick. An overview of emerging results in cooperative uav control. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 602–607. IEEE, 2004.

[12] Hyunjin Choi, Youdan Kim, and Inseok Hwang. Reactive collision avoidance of unmanned aerial vehicles using a single vision sensor. *Journal of Guidance, Control, and Dynamics.*

[13] John Valasek, Kiran Gunnam, Jennifer Kimmett, John L Junkins, Declan Hughes, and Monish D Tandale. Vision-based sensor and navigation system for autonomous air refueling. *Journal of Guidance, Control, and Dynamics*, 28(5):979–989, 2005.

[14] James H Spencer. Optical tracking for relative positioning in automated aerial refueling. Technical report, DTIC Document, 2007.

[15] Mario Luca Fravolini, Antonio Ficola, Giampiero Campa, Marcello Rosario Napolitano, and Brad Seanor. Modeling and control issues for autonomous aerial refueling for uavs using a probe–drogue refueling system. *Aerospace science and technology*, 8(7):611–618, 2004.

[16] Steven M Ross. Formation flight control for aerial refueling. Technical report, DTIC Document, 2006.

[17] Shoaib Ehsan and Klaus D McDonald-Maier. On-board vision processing for small uavs: Time to rethink strategy. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 75–81. IEEE, 2009.

[18] Patrick A. McCarthy. Characterization of uav performance and development of a formation flight control for multiple small uavs. Master's thesis, Air Force Institute of Technology, 2006.

[19] Christian Lopez. UAV Formation flight utilizing a low cost, open source configuration. Master's thesis, California Polytechnic State University, 2013.

[20] Zouhair Mahboubi, Zico Kolter, Tao Wang, Geoffrey Bower, and Andrew Y Ng. Camera based localization for autonomous uav formation flight. In *Proceedings of the AIAA@ Infotech Conference*, 2011.

[21] The paparazzi project. `http://paparazzi.enac.fr/wiki/Main_Page`.

[22] Steven M. Ross. Formation flight control for aerial refueling. Master's thesis, Air Force Institute of Technology, 2006.

[23] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly, 2008.

[24] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. EPnP: Accurate non-iterative O(n) solution to the PnP problem. *International Journal of Computer Vision*, 2008.

[25] Berthold KP Horn, Hugh M Hilden, and Shahriar Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *JOSA A*, 5(7):1127–1135, 1988.

[26] K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-squares fitting of two 3-d point sets. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (5):698–700, 1987.

[27] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on pattern analysis and machine intelligence*, 13(4):376–380, 1991.

[28] James H. Spencer. Optical tracking for relative positioning in automated aerial refueling. Master's thesis, Air Force Institute of Technology, 2007.

[29] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985.

[30] J-A Ting, Evangelos Theodorou, and Stefan Schaal. A kalman filter for robust outlier detection. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 1514–1519. IEEE, 2007.

[31] Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995.

[32] Mike Pursifull. HiLStar 17F developer's guide.

[33] Pantelis Elinas and James J Little. Stereo vision slam: Near real-time learning of 3d point-landmark and 2d occupancy-grid maps using particle filters. In *Visual SLAM Workshop*. Citeseer, 2007.

[34] Jongki Moon, Ramachandra Sattigeri, JVR Prasad, and Anthony J Calise. Adaptive guidance and control for autonomous formation flight. In *American Helicopter Society 63rd Annual Forum*, 2007.

[35] James Doebbler, Theresa Spaeth, John Valasek, Mark J Monda, and Hanspeter Schaub. Boom and receptacle autonomous air refueling using visual snake optical sensor. *Journal of Guidance, Control, and Dynamics*, 30(6):1753–1769, 2007.

[36] Artur Loza, Miguel A Patricio, Jesús Garcia, and José M Molina. Advanced algorithms for real-time video tracking with multiple targets. In *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pages 125–131. IEEE, 2008.

Appendix A

Nomenclature

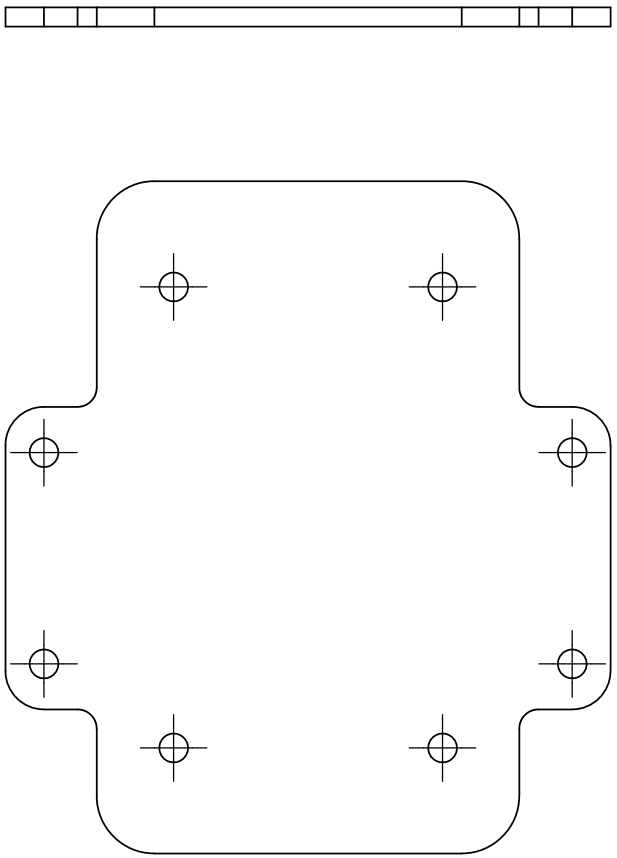
LIST OF SYMBOLS

A	State transition matrix	60
B	Control input model	60
u_k	Control vector at step k	60
P_k^-	*A priori* estimate error covariance at step k	60
P_k	*A posterori* estimate error covariance at step k	60
Q	Process noise covariance	60
R	Measurement noise covariance	60
K_k	Kalman gain at step k	61
C	Observation matrix	61
z_k	Measurement at step k	61
S	*A posteriori* prediction covariance	61
δ_T	Throttle setting	64
δ_e	Elevator deflection	64
δ_a	Aileron deflection	64
k_1, k_3, k_5	Polynomial error feedback coefficients	74

LIST OF ABBREVIATIONS

Appendix B

Drawings

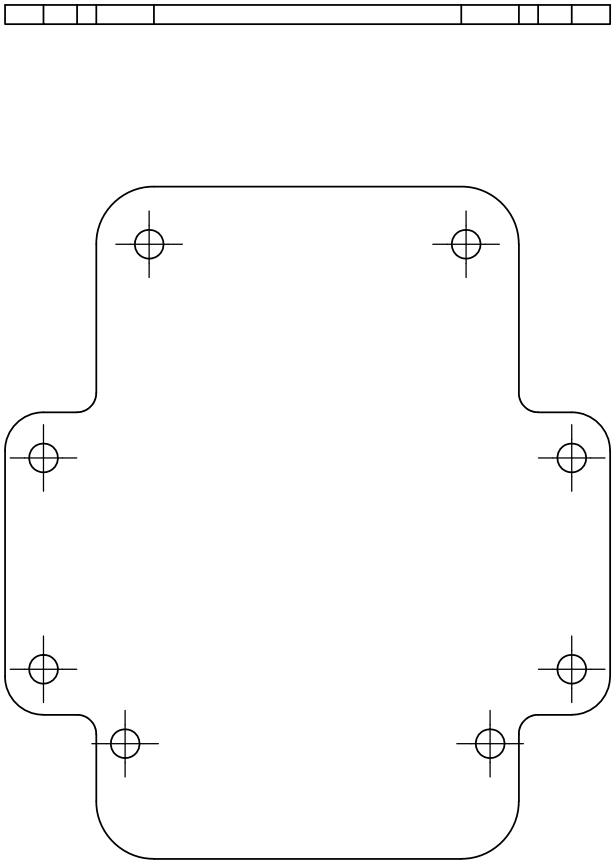


		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

TITLE:

SIZE	DWG. NO.	REV
A	APM_Base	
SCALE: 1:1	WEIGHT:	SHEET 1 OF 1

4 3 2 1

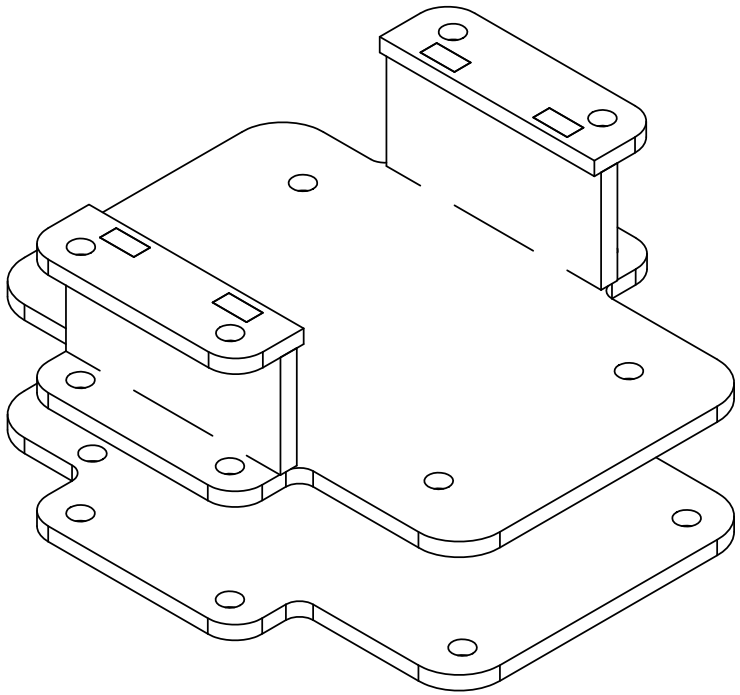


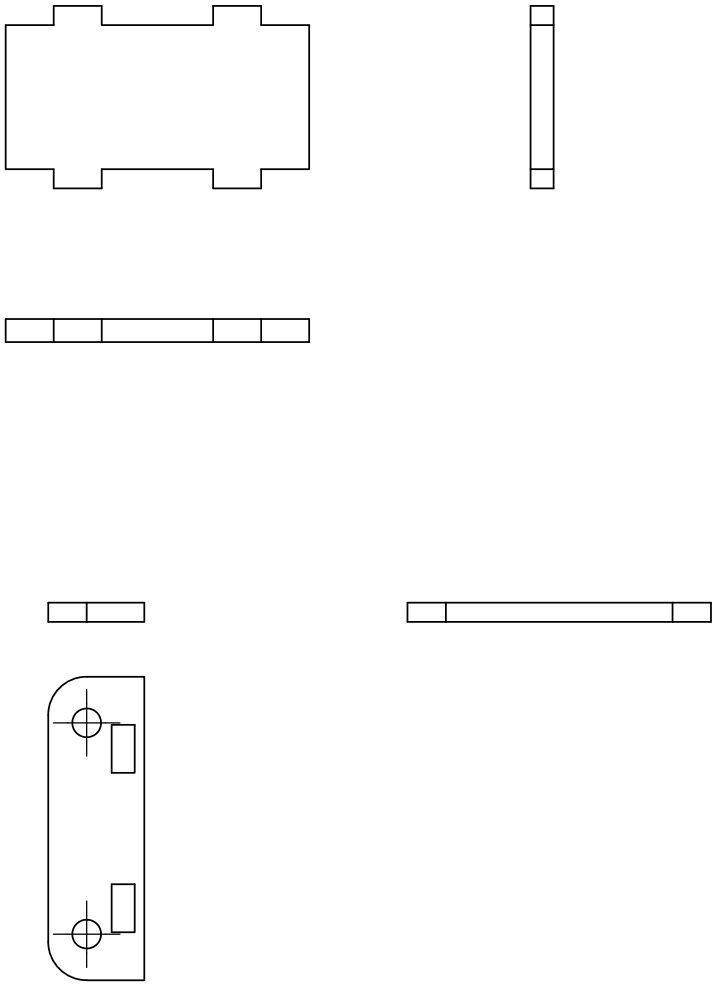
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

TITLE:

SIZE	DWG. NO.	REV
A	BBB_Base2	
SCALE: 1:1	WEIGHT:	SHEET 1 OF 1

4 3 2 1



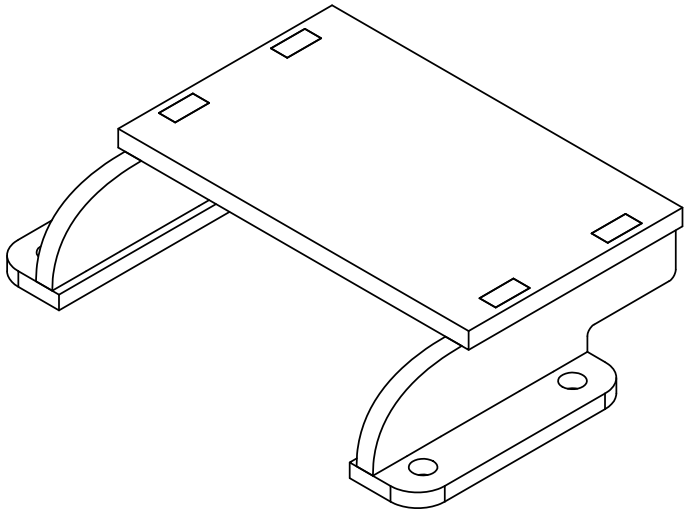


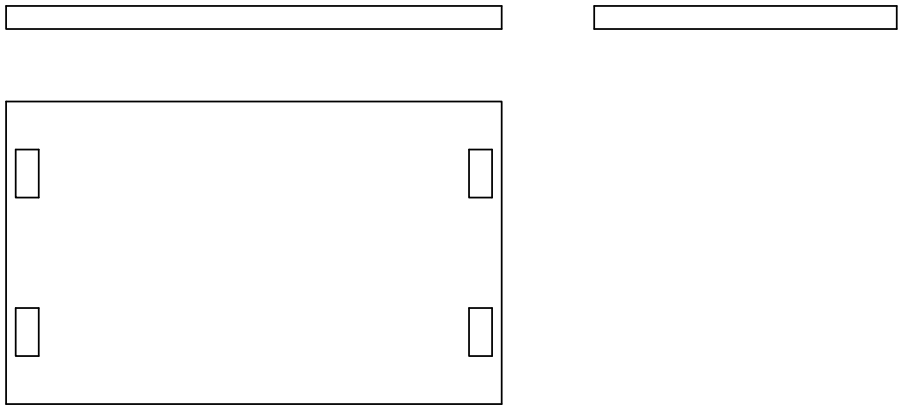
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

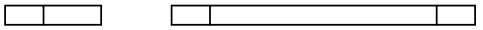
TITLE:

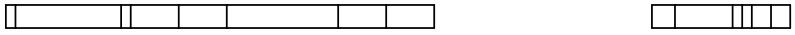
SIZE	DWG. NO.	REV
A	ElectronicsChassis	
SCALE: 1:1	WEIGHT:	SHEET 1 OF 1

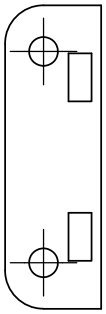
4 3 2 1

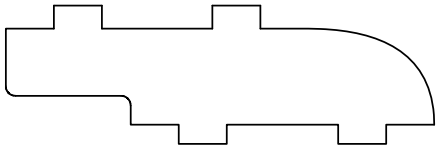






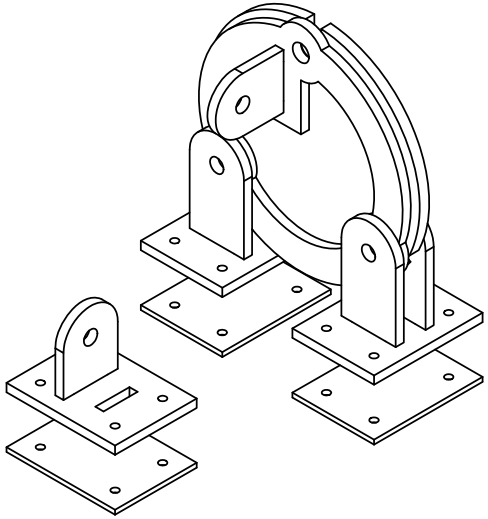


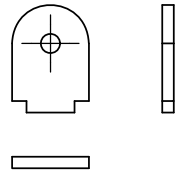


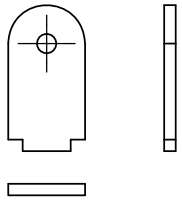


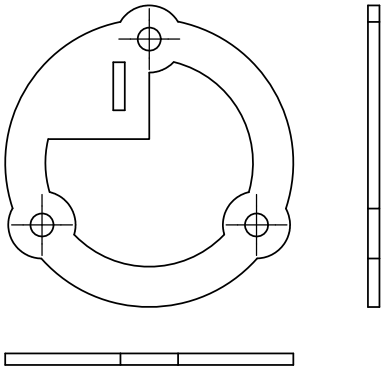
		UNLESS OTHERWISE SPECIFIED:		NAME	DATE	
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN			TITLE:
			CHECKED			
			ENG APPR.			
			MFG APPR.			
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.			
		MATERIAL	COMMENTS:			SIZE: A — DWG. NO.: GoPro_Mount — REV
NEXT ASSY	USED ON	FINISH				
APPLICATION		DO NOT SCALE DRAWING				SCALE: 1:1 — WEIGHT: — SHEET 1 OF 1

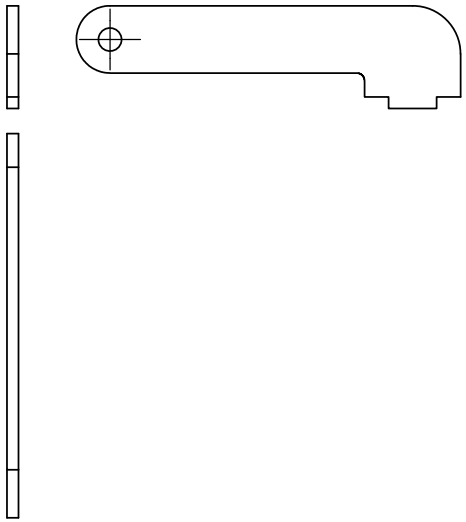
4 3 2 1

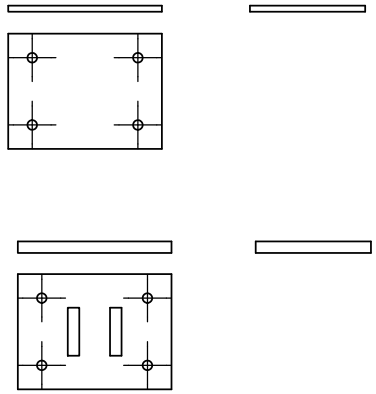


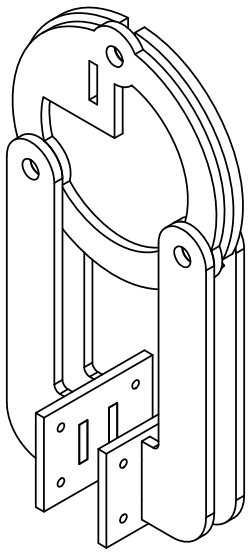












		UNLESS OTHERWISE SPECIFIED:		NAME	DATE	
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN			
			CHECKED			TITLE:
			ENG APPR.			
			MFG APPR.			
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.			
		MATERIAL	COMMENTS:			SIZE DWG. NO. REV
NEXT ASSY	USED ON	FINISH				ALED_Mount2
APPLICATION		DO NOT SCALE DRAWING				SCALE: 1:1 WEIGHT: SHEET 1 OF 1

4 3 2 1

Appendix C

Required Camera Resolution and Focal Angle

While the *PlayStation Eye* was still under consideration for use as the USB webcam, some consideration was given to the camera's focal angle range and resolution. Due to the nature of optic systems, there is an inherent trade-off between near-field and far-field performance. A wide focal angle allows the camera to completely capture larger objects at a set distance, but at longer distances, the "pixels on target" will be much lower. Conversely, a narrow focal angle will capture a target in more detail from afar, but will not be able to completely capture an object up-close. Increasing camera resolution allows a wide focal angle to be used without losing as much detail in the far-field, but comes at a great computational expense.

Early work sought to visualize the trade-offs between focal angle and resolution. Figure C.1 shows focal angle contours plotted against pixels per square inch (ppi) and the distance from the camera to the target in feet. Since the *PlayStation Eye* features an adjustable focal angle (56 or 75 degrees), both curves are included. There is some lower bound (not shown) in pixels-per-square inch at which the lead vehicle would not be able to be detected. The limit is highly dependent on a vast number of variables such as LED brightness, camera exposure, background scenery, the nature of the feature detection algorithm, etc. However, for the LEDs to be distinguishable from random white nose in the images, they should be comprised of at least 4-5 or more pixels. If the LED is approximately 3 square inches in area, about 1.5 ppi would be needed to adequately detect it at any given distance.

In the near-field case, the field of view is limited by the minimum distance at which the target can be entirely captured within the frame of the image. Figure C.2 shows focal angle

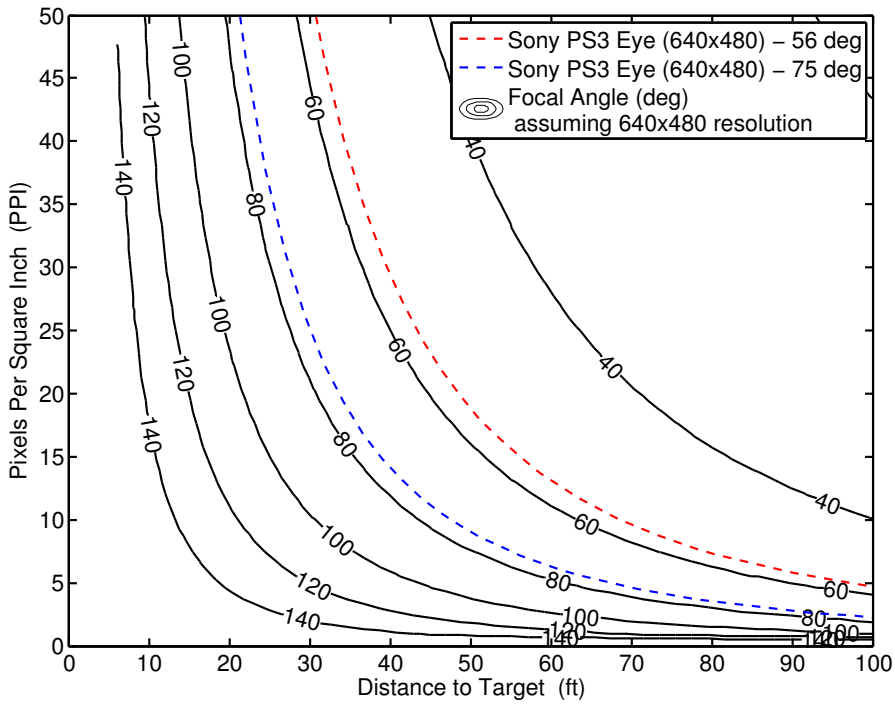


Figure C.1: Pixel density (far-field)

contours plotted against the horizontal field of view against the distance to the target. This early work assumes the wingspan of the *SkySurfer* airframe, which was originally considered before eventually transitioning to the *Penguin*. Again, the two focal angles of the *PlayStation Eye* are plotted. The orange constraint is the wingspan of the *SkySurfer* and the red constraint is the horizontal (y_b), distance from one wingtip to the opposite tip of the horizontal stabilizer. This constraint was included because the EPnP algorithm used by this thesis only requires *four* points to compute a pose estimate. Theoretically, the leader's pose could be computed in the region between the orange and red constraints, but would require extremely robust vision software. Instead, the orange wingspan constraint should be used as a reference with comfortable margin to account for tracking errors, camera vibrations, and atmospheric disturbances—all of which could cause the leader to fall out of frame of the camera.

The camera resolution is limited to those supported by the camera, most popularly

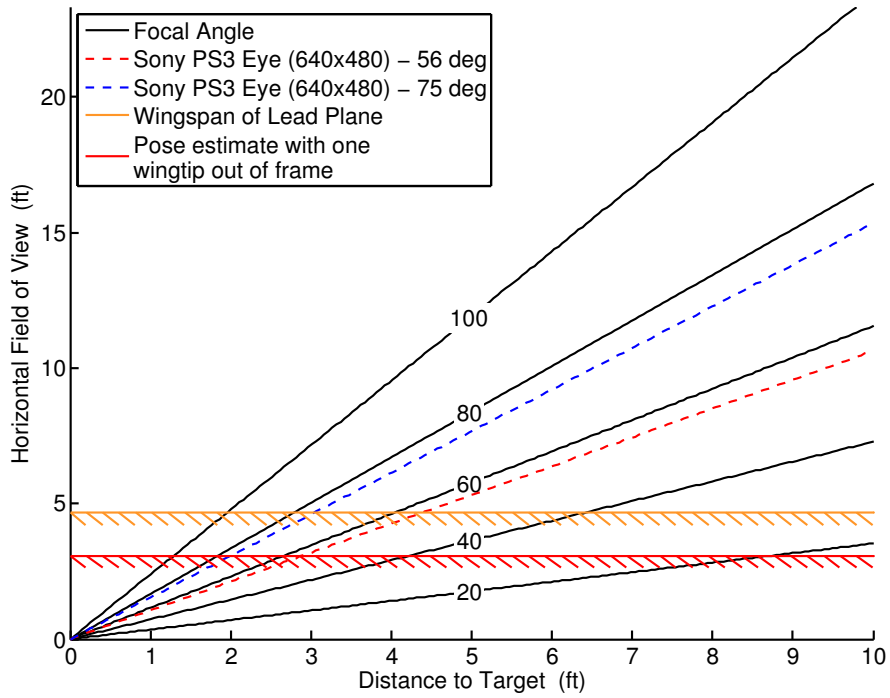


Figure C.2: Field of view (near-field)

320×240 and 640×480. Although higher resolutions are possible on the *Logitech C920*, they would be computationally expensive and could not be used in real-time applications running on the *BeagleBone Black*. Therefore the resolution was set to 640×480 because it was the highest supported resolution that could run real-time on the embedded SBC.

From this analysis, it was decided that a narrower focal angle should be used during earlier formation attempts, where the vehicles would have more separation. If early flights were successful, a wider focal angle could be used, the camera could be re-calibrated, and attempts at tighter formation flights could be made.

Appendix D

Flight Test Manual

FLIGHT TEST MANUAL

"CLOSE FORMATION FLIGHT OF SMALL UAVS USING VISION-BASED LOCALIZATION"

BY: MICHAEL DARLING

TABLE OF CONTENTS

TABLE OF FIGURES

1. Pre-flight Hardware Set-up

1.1. Electrical Power and Control Signal Routing

- Flight battery is connected through emergency shutoff loop to ESC
- ESC provides power to motor through bullet connectors
- Servo wire from ESC is connected to APM throttle output
- All other servo outputs are connected to corresponding APM output
- Receiver channels are connected to APM inputs
- A wired power jumper is used to short across the APM input and output rails to provide power to APM electronics

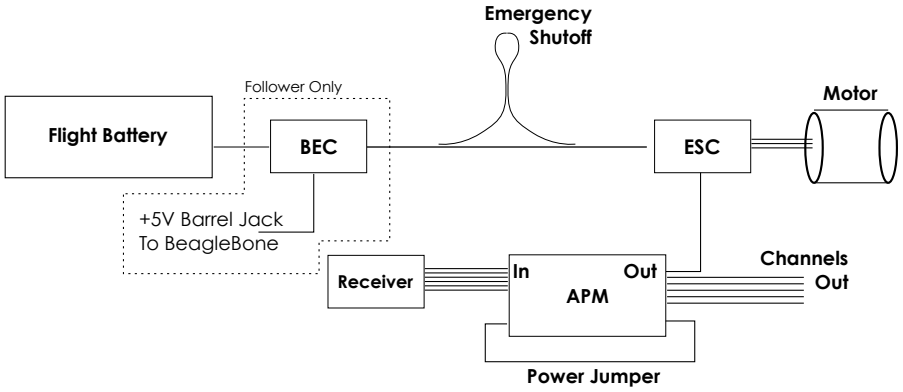


Figure 1: Power and control signal connection diagram

- Emergency power shutoff loop is passed through hole at rear of aircraft canopy
- Power circuit can be opened or closed with the canopy fastened to the airframe by connecting or removing the emergency shutoff loop, respectively

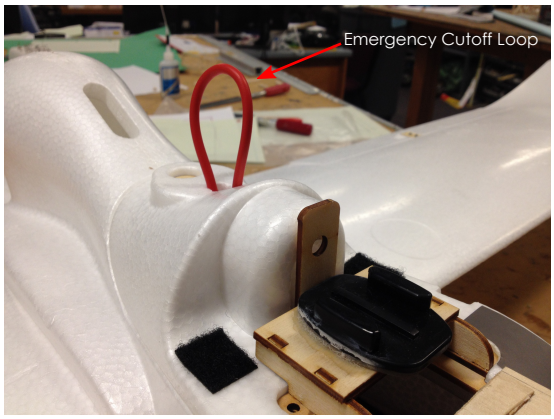


Figure 2: Emergency power shutoff

- Servo channels are (1) Ailerons, (2) Elevator, (3) Throttle, (4), Rudder
- For the *leader only*, channel (5) is reserved for the LED switch
- Power jumper is used to connect any remaining input/output
- Signal wire always towards center of APM (ground wire towards edges of board)

- GPS receiver connects to labeled port shown in Figure 3
- Telemetry kit connects to labeled port shown in Figure 3

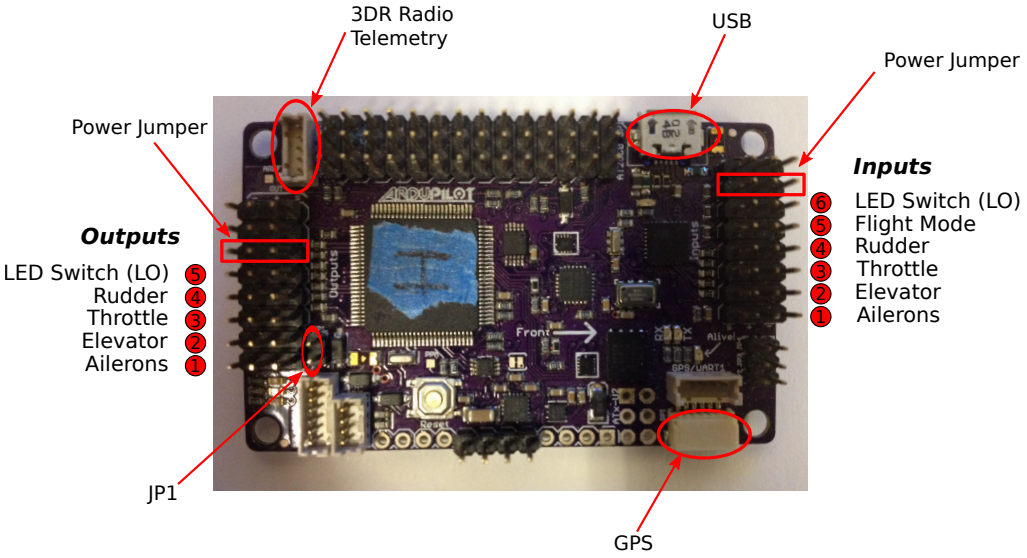


FIGURE 3: APM WIRING SCHEMATIC (LO) = "LEADER ONLY"

- APM input channels (1) to (4) connected to corresponding receiver channel
- Flight mode input (5) is connected to "GEAR", and LED Switch input (6) is connected to "AUX 1" on Receiver

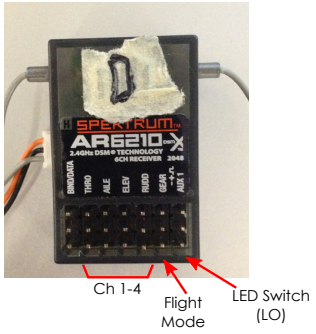


FIGURE 4: RECEIVER CONNECTIONS

1.2. LED SUBSYSTEM (LEADER ONLY)

- Input (6) is reserved for LED switching and connects to "AUX 1" from the receiver
- Output (5) is reserved for LED switching and connects to ground and signal of switching circuit (use only ground and signal wires)
- DC/DC power converter is connected to dedicated LED LiPo batteries on input side, and provides power to LED circuitry on output side
- Power converter should be set to approximately 13.4 Volts via voltage adjust set screw
- LEDs can be remotely switched using the Flap/Gyro switch on the Tx

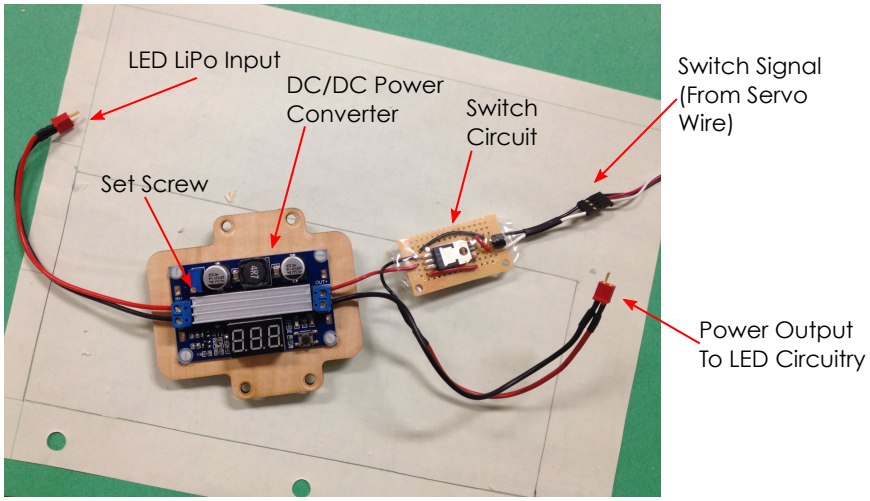


FIGURE 5: LED POWER SUPPLY AND SWITCHING CIRCUIT

1.3. VISION SUBSYSTEM (FOLLOWER ONLY)

- BeagleBone Black (BBB) is powered by 5V barrel jack from Castle Link BEC
- Logitech C920 camera connects to USB port
- Preformatted microSD card inserted into microSD slot for external photo storage
- BBB connects serially to APM UART2 through logic level converter
    - Pins 2, 4, 24, and 26 are connected to Gnd, LV, Rx, Tx of logic level converter, respectively
    - APM UART2 pins (Bk, R, Y, G) connect to Gnd, HV, Rx, Tx, respectively
- BBB is connected to GCS via secure shell using miniUSB connection

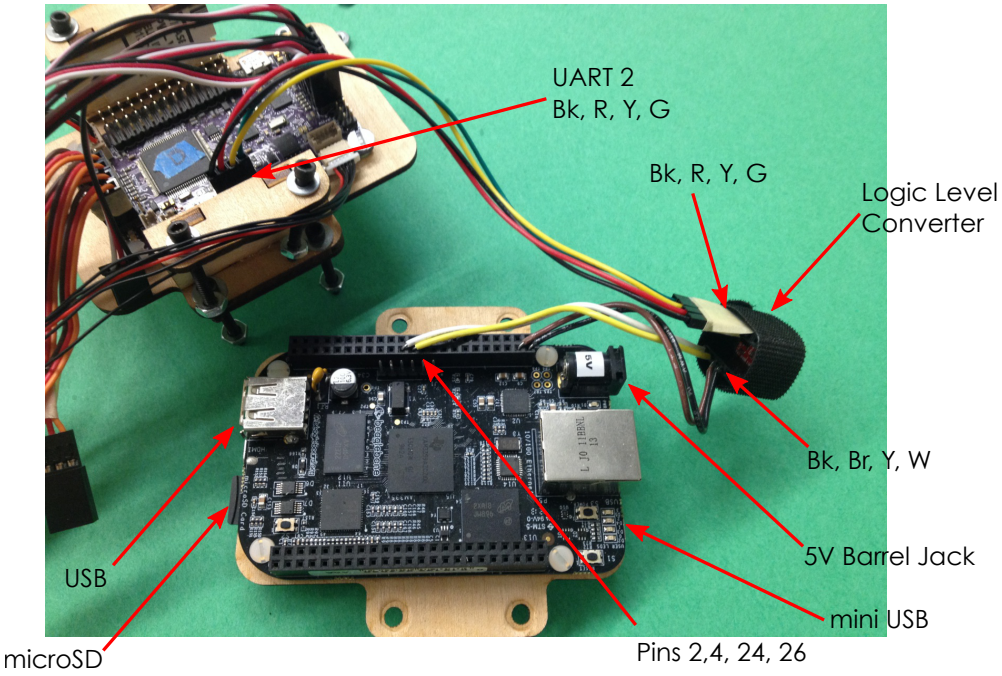


FIGURE 6: BEAGLEBONE BLACK WIRING SCHEMATIC

- Logitech C920 is affixed to custom camera mount and held securely using two rubber bands
- Masking tape is used to hold tinted lens in place

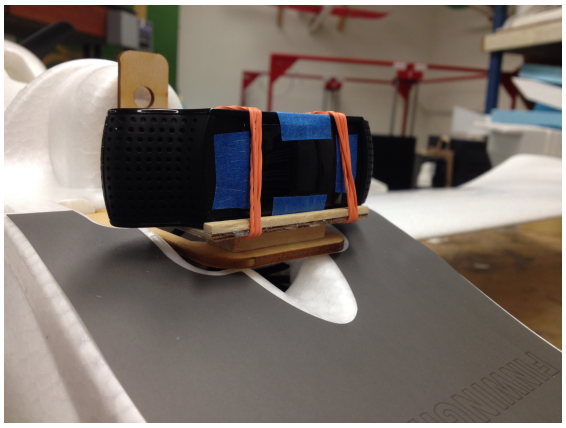


FIGURE 7: LOGITECH C920 CAMERA MOUNTING

- Optionally, GoPro can be mounted behind Logitech C920 to capture high-definition footage of flight tests

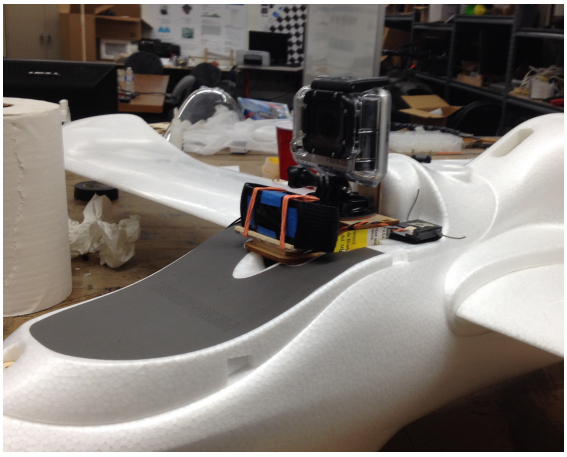


FIGURE 8: OPTIONAL GOPRO CAMERA MOUNTING

1.4. PACKAGING

- Receiver antenna and GPS receiver are held to top of canopy with Velcro
  - Wires are routed into fuselage through hole in top of balsa chassis
- On follower, C920 camera (and optionally GoPro) is mounted to top mounting surface
  - Extra camera wiring is held with Velcro to bottom of canopy
  - Logic level converter is secured with Velcro to bottom-rear of canopy
- APM, BBB, and DC/DC power converter are held in place with balsa mounting chassis
- Telemetry kit antenna is fastened to bottom of canopy with Velcro
- On leader, switching circuit is fastened to bottom of balsa chassis with Velcro
- APM input servo wires are routed over APM to aft end of balsa chassis where they connect to the receiver

- Labeled servo extension cables are attached to the APM output channels so that they can be connected to servos while chassis is mounted to canopy

FIGURE 9: ELECTRONICS PACKAGING FOR FOLLOWER

FIGURE 10: ELECTRONICS PACKAGING FOR LEADER

2. PRE-FLIGHT SOFTWARE CONFIGURATION

2.1. APM FIRMWARE (CONFIGURATION, COMPILATION, AND UPLOADING)

- Ensure the latest firmware is installed
  - Open Windows on a desktop PC in the Flight Lab (Windows XP no longer virtualized on GCS laptop computer)
  - From Start menu, open Git Bash terminal
  - cd /c/Users/mdarling/Desktop/APM-Vision
  - Make sure on GroundStation branch appears after command prompt (`~/Desktop/APM-Vision <GroundStation>`)
  - git gui
  - Branch → Checkout → Local Branch → GroundStation

- Merge → Local Merge → Tracking Branch → origin/GroundStation

- Configure and upload APM firmware for flight testing
  - Open Visual Studio by clicking desktop icon
  - File → Open → Project/Solution → C:\Users\mdarling\Desktop\APM-Vision\ArduPlane\ArduPlane.sln
  - Open APM_Config.h and ensure that:
    - HIL_MODE is set to HIL_MODE_DISABLED
    - MY_DEBUG can contain any value
    - HAS_LEDS should be set to 1 for the leader and 0 for the follower
    - LED_CH should be set to 6

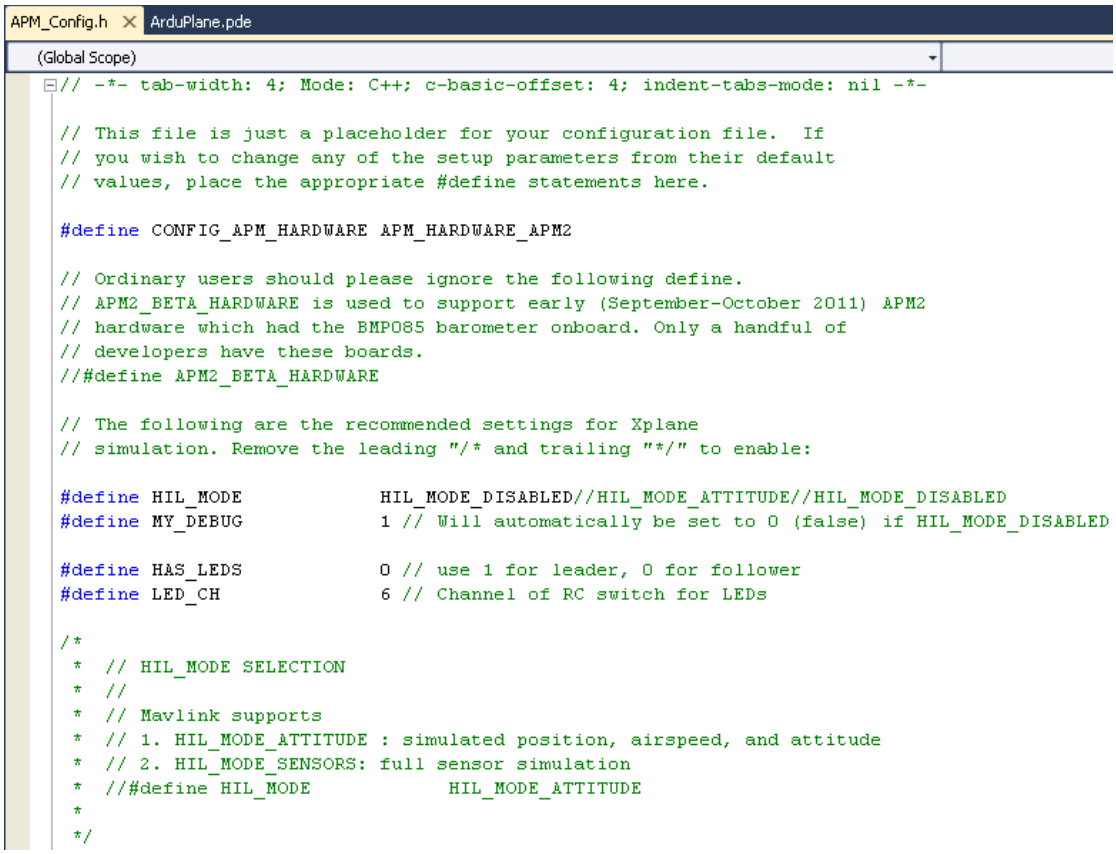


FIGURE 11: APM_CONFIG.H SETTINGS

  - Build → Build Solution to ensure firmware contains no errors
  - Connect APM to computer via USB
  - In Visual Studio, make sure Arduino Mega is selected and appropriate COM port

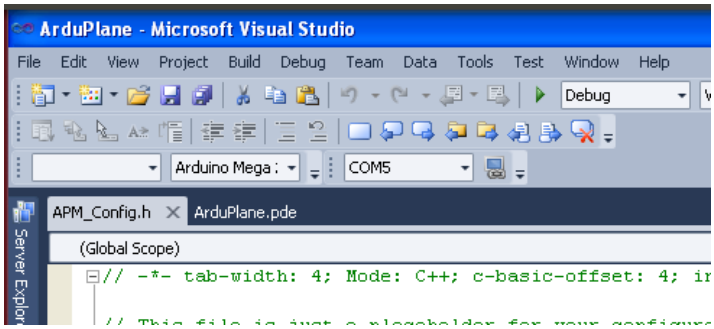


FIGURE 12: BOARD AND COM PORT SELECTION

- Upload firmware to board by clicking "Play" button or Debug → Start Debugging
- Once upload is complete, close Visual Studio and disconnect APM board

2.2. VISION SOFTWARE (CONFIGURATION, UPLOAD, AND COMPILATION)

- Ensure latest source code is downloaded
  - Open a terminal and `cd ~/Desktop/CompleteVision_MAIN`
  - Open Visual Studio by clicking desktop icon
  - `git gui`
  - Merge→ Local Merge → Tracking Branch → origin/master
- Open Code::Blocks IDE in Ubuntu
- Open vision software project File → Open → `~/Desktop/CompleteVision_MAIN/CompleteVision_MAIN/CompleteVision_MAIN.cbp`
- Open Config.hpp. There are a number of settings, but most notably:
  - `#define OUTDOOR` (make sure not commented out or set to other value)
  - `frameSkip_ms = 500;` (can be set to change interval that frames are saved)
- Build (and Run) to ensure no errors, save project and close Code::Blocks IDE
- Provide power to BBB through 5V barrel jack and connect to host with USB cable (eject when File Manager pops up)

- Updating flight code
  - Open two terminal windows on host computer
  - In first window, ssh into BBB (`ssh ubuntu@192.168.7.2, password: temppwd`) Confirm `~/MAIN` directory exists and cd into it
  - In the second terminal window, navigate to `!/Desktop/CompleteVision_MAIN` on the host computer
  - In second window, execute `./upload` to transfer files to `~/MAIN` on BBB
- Building flight code
  - In a terminal window, connect to BBB over ssh, navigate to `~/MAIN` on BBB
  - Build vision program with optimization flags `./build optimize`
  - Confirm the code runs by typing `./main` (ctrl + c to exit, USB webcam must be attached, make sure `Penguin_Geom.txt` and `C920-640x480_IntrinsicParams.yml` are read-in correctly, make sure framerate is sufficiently high (> 20 fps), consider pointing camera towards Leader w/ LEDs to confirm reasonable pose estimate is found)

- Type `exit` to leave the ssh session and unplug BBB, if desired

2.3. Channel Reversal and Mode Switching

- Open File Manager and navigate to `~/Desktop/APM Ground Control`
- Double-click "MissionPlanner" and choose to execute from the pop-up menu
- Connect APM to airplane servos (Use a separate ESC to avoid powering throttle), and connect to telemetry kit
- Connect corresponding telemetry kit to host computer USB
- Choose `/dev/ttyUSB#` from the drop-down menu and 57600 baud rate when using telemetry, then "connect". (When connecting over USB, choose `/dev/serial/by-id/usb-Arduino__...` and 115200 baud rate.)

Figure 13: APM Telemetry connection settings

- Click Configuration tab, can switch channels by
  - Checking or unchecking channel reverse boxes and recalibrating, OR
  - Going to Advanced Params → Adv. Parameter List → Save to make sure RC channel reversing is saved for later flights
- Save parameter files by going to Advanced Params → Adv Parameter List → Save
- To modify mode switches, go to Configuration → Flight Modes, then using the drop-down menu, select the appropriate flight mode (For custom flight modes, must go to Adv. Parameter List and modify FltMode# to the correct flight mode. ***Rel_NAV is Flight Mode #17 and should appear as "AUTO" or "UNKNOWN" in MissionPlanner or QGroundControl)***

3. Preflight Procedures

3.1. Channel Reversal, Mode Switching, and CG Checks

- Emergency power shutoff loop removed, turn Tx on and switch to manual flight mode
- Apply power to APM and ESC by replacing power shutoff loop
- Confirm that control surfaces deflect in correct directions and propeller spins with the correct rotation
- Connect to APM through Mission Planner via telemetry link
- In Configuration tab, set THROTTLE_MIN = 0 and write params
- Switch into stabilize mode on Tx and confirm that surfaces deflect in correct directions, and motor doesn't spin at zero throttle. Also, ensure that surfaces deflect to provide

corrective control when APM is perturbed from level. (Make sure that rudder deflection mixes properly with aileron.)
    - If any surface deflection is reversed, modify the value of RC_CH_REV for the corresponding channel (Configuration → Adv. Param → RC_CH_REV)
    - If the rudder/aileron mix is reversed, then change the sign of KFF_RDDRMIX
    - Write new parameters (upload to APM) and save to file
- To modify mode switches, go to Configuration → Flight Modes, then using the drop-down menu, select the appropriate flight mode (For custom flight modes, must go to Adv. Parameter List and modify FltMode# to the correct flight mode. ***Rel_NAV is Flight Mode #17 and should appear as "AUTO" or "UNKNOWN" in MissionPlanner or QGroundControl)***
- Navigate to Flight Data tab of mission planner. Cycle through flight modes and confirm flight mode in HUD. (Custom flight modes will appear as "Unknown".)
- Check that CG is located at ¼ chord with all components in place

3.2. Clear Flight Logs

- Remove throttle (power) and servo wires from APM outputs (to prevent hard-over servo deflections)
- Provide power to APM output from separate ESC and battery (to keep from providing power to motor)
- Connect to APM *over USB*
- Navigate to Terminal tab, then type `logs [enter] erase[enter]`
- Disconnect in Mission Planner, remove power, and reconnect servo wires and ESC to throttle channel
- To clear any logs stored in APM log directory of host computer, navigate to `~/Desktop/Flight Logs` and execute script to clear logs

3.3. Upload Parameter Files

- Power via external ESC (remove ESC throttle channel)
- Connect using Mission Planner via telemetry link
- Configuration → Adv. Params → Adv. Param List → Load `~/Desktop/APM-Vision/ParamFiles/…` → Write Params
- Compare params to ensure upload occurred successful (look for no or few deltas)

3.4. Upload Waypoint Files

- Connect through Mission Planner via telemetry link
- Navigate to Flight Planner tab
- Right-click on map → File Load/Save → Load WP File
- Write WPs
- Note: When creating waypoints, need at least one waypoint following a "DO_JUMP"

3.5. Initialize APM Sensors

- Follow above procedures to connect and mount all components
- When ready, replace the emergency power shutoff loop to provide power to APM, servos, and motor
- Hold wings level until initialization is complete

3.6. Establish Telemetry Link

- Mission Planner (Single UAV)
  - From File Manager, double-click `~/Desktop/APM Ground Control/MissionPlanner`
  - Plug in telemetry kit
  - In Mission planner, select /dev/ttyUSB0, 57600 baud, and press "connect"
  - Navigate to the Flight Data tab
  - Ensure GPS is working (aircraft appears in correct location on map)
- QGroundControl (Multi-UAV)
  - From File Manager, double-click `~/Desktop/APM Ground Control/QGroundControl`
  - Click APM in pop-up window
  - Click "Connect LIR", choose /dev/ttyUSB0, 57600 baud
  - Navigate to Mission tab and select "New Link" (/dev/ttyUSB1, 57600)
  - Ensure GPS working on both UAVs (appear in correct location on map)

3.7. Run BBB Vision Subsystem Software

- Connect to BBB over USB and ssh (`ubuntu@192.168.7.2, psswd: temppwd`)
- Begin a new screen session `screen [enter]`
- Navigate to `~/MAIN` and execute the software by typing `./main`
- Once the program is running smoothly, detach from screen session `Ctl + a, d`
- Exit from ssh by typing `exit`
- Disconnect BBB from USB

4. Post-flight procedures

4.1. Interrupting Vision Subsystem Software

- Connect to BBB over USB and ssh (`ubuntu@192.168.7.2, psswd: temppwd`)
- Reattach to running screen session `screen –R`
- Stop software by typing `Ctl + c`
- Exit screen session by typing `exit`
- Leave shell by typing `exit` again

4.2. Downloading BBB Images

- Open a new terminal and navigate to `~/Desktop/Flight Logs/Images`
- Download images to a new directory, "foo", by typing `./downloadData foo`
- A compressed archive, "foo.tar.gz" will be created in `~/Desktop/Flight Logs/Images/`
- Disconnect BBB from USB and remove power

4.3. Downloading APM Log Files

- Open Mission Planner and connect to APM over USB (if fails to disconnect, add user to dialout group:
  `sudo gpasswd –add mdarling dialout` (log out, then back in))
- Go to Terminal tab → Log Download

- A pop-up window should appear followed by a warning message (Click OK on warning message)
- Check log(s) to be downloaded and click "Download These Logs" button
- Use "browse" button in main window to confirm logs transferred OK
- In terminal, navigate to `~/Desktop/Flight Logs` and type
  `./logcp [leader/follower] logName`
  i.e.
  `./logcp -L LeaderFlight1` (saves logs to ./Leader/LeaderFlight1)
  `./logcp -F FollowerFlight1` (saves logs to ./Follower/FollowerFlight1)
- ***Confirm that logs were transferred properly using File Manager. If not, then repeat these steps from the beginning***
- In Mission Planner, clear logs using Terminal tab and typing
  `logs [enter] erase [enter]`

5. Appendix

1. Connecting to BBB over ssh using GNU screen

- Open a terminal window and type `ssh ubuntu@192.168.7.2` (psswd: temppwd)
- Begin a screen session `screen [enter]`
- (Begin any processes desired, i.e. `./MAIN/main`)
- Detach from screen session `Ctl + a, d`
- Can now `exit` from ssh session and disconnect USB, then later reconnect USB and ssh using above procedure
- Reattach to last screen session `screen -R`
- Can terminate any running processes with `Ctl + c`
- Leave screen session with `exit`, then `exit` again to leave ssh session

2. Backing up/Restoring BBB Disk Image

- Creating a backup image:
  - Insert a bootable microSD card with a "Live" Ubuntu image (4GB card in labeled case)
  - With board powered off, insert uSD card. Hold boot button and apply power until all four USER LEDs glow solid
  - Confirm that BBB is booted from Live uSD `ls >>Ubuntu_Live_uSD`
  - Backup the image to the desired location
    `sudo dd if=/def/mmcblk1 bs=1M | ssh mdarling@192.168.7.1 "dd of=/home/mdarling/Desktop/BBB_Backup_Images/BBB_BACKUP_NAME.img"`
- Restoring from a backup image:
  - (Repeat steps 1-3 from above to boot from Live uSD)
  - Restore from desired backup
    `ssh mdarling@192.168.7.1 "dd if=/home/mdarling/Desktop/BBB_Backup_Images/DESIRED_BBB_BACKUP.img bs=1M" | sudo dd of=/dev/mmcblk1 bs=1M`

3. Re-Flashing microSD Card and Configuring /dev/media on BBB (untested)

- Power off BBB and remove uSD. Use card reader and adapter to connect to host computer
- Reformat uSD card to FAT32 format using gParted
- With uSD still connected to host, create a file uEnv.txt with the following contents:

```
mmcdev=1
bootpart=1:2
mmcroot=/dev/mmcblk1p2 ro
optargs=quiet
```

- Remove card from host and insert into BBB. Power up BBB, and check results of:
  `ls /dev/mmcblk*`
- Should see something like: (especially looking for `/dev/mmcblk0p1`)

```
/dev/mmcblk0        /dev/mmcblk1        /dev/mmcblk1boot1   /dev/mmcblk1p2
/dev/mmcblk0p1      /dev/mmcblk1boot0   /dev/mmcblk1p1
```

- Check contents of `/etc/fstab` (`sudo nano /etc/fstab`)
- Should contain the line:

```
  # Automount microSD card to /media/ubuntu/microSD/dev/mmcblk0p1/
media/ubuntu/microSD vfat rw,suid,dev,exec,auto,user,
async,uid=1000,gid=Ubuntu
```

- Running vision software should automatically generate `TestImages` directory. If not, make sure `/meda/Ubuntu/microSD` is a directory and execute:
  `mkdir /media/Ubuntu/microSD/TestImages`

4. Configuring UART Ports on BBB

- ssh into BBB as normal, then execute the following commands to mount the bootloader

```
mkdir /mnt/boot
mount /dev/mmcblk0p1 /mnt/boot
nano /mnt/boot/uEnv.txt
```

- Add the following to the end of uEnv.txt

```
#Enable UART1 on Boot
optargs=capemgr.enable_partno=BB-UART1
```

Appendix E

OpenCV With SIMD Acceleration

Much of the work in this thesis would not have been possible without the help and support of the open source community. The success of "open source" as a concept is highly dependent on voluntary contributions to the community. The following informal "How-To" document was written and released as a draft into the open source community as a way of giving back by helping others to compile OpenCV to take advantage of SIMD hardware acceleration.

[DRAFT] How to Achieve 30 fps with BeagleBone Black, OpenCV, and Logitech C920 Webcam [DRAFT]

Michael Darling
FndrPlayer39@gmail.com

February 3, 2014

This "How-to" outlines some of the issues associated with video capture on the BeagleBone Black (BBB) for robotic vision applications using a USB webcam and OpenCV, and presents a possible solution for improved video capture performance along with a set of detailed instructions. In particular, this document addresses the challenge of achieving a suitably high framerate (30 fps) for robotic vision applications making use of the BBB. If you wish, you can skip the introductory material and jump directly to the How-to.

Contents

Problem Background

Many robotic vision applications require video capture and frame processing to occur at a high rate of speed. For example, my master's thesis seeks to implement autonomous close formation flight of two remote controlled aircraft using computer vision for localization. In short, the lead aircraft will be outfitted with very high intensity LEDs on each of its wingtips and tail surfaces while the following aircraft will be equipped with a vision system consisting of the BeagleBone Black and a USB webcam. Both vehicles will be controlled using the open source ArduPilot Mega autopilot. The OpenCV vision software running on the BBB will detect the LEDs in each video frame, estimate the 3-D relative position and orientation of the leader, and pass this information over to the autopilot which will handle the control. This type of application requires high bandwidth localization estimates for the follower to maintain its position behind the leader. The rate at which video frames can be processed and captured becomes even more important considering the fact that the LEDs may not be detected in every frame due to background noise in the image.

First Attempts with the PlayStation 3 Eye

For me, a reasonable place to start was by using the PlayStation 3 Eye USB webcam in combination with OpenCV. The PS3 Eye is used by many "do-it-yourself" robotics enthusiasts due to its availability, very low cost, high framerate (up to 120 fps at 320×240 and up to 60 fps at 640×480 resolution), and multi-platform support by 3rd party drivers. [1] Unfortunately for those who want to use the PS3 Eye with the BBB, this kind of performance can't be expected—at least not easily.

If you were to set up a simple OpenCV video capture program and attempted to operate the PS3 eye at 640×480 resolution at 60 fps, you would end up with repeated "select timeout" errors and no video frames. You would have identical results at 30 and 15 fps as well, however if you settle for 320×240 resolution, you would get a stream of video, but always at 30 fps regardless of the framerate you set. *Why?* It turns out that the OpenCV functions for setting the video framerate do not work (at least for Video4Linux devices) and the default 30 fps is used. In order to set the camera framerate, you have to write your own capture code using the Video4Linux2 API. [2] But even after using custom capture code, you would find that you can only acquire 640×480 frames with the camera set to 15 fps. And even then you would *actually* be capturing frames at about 7 fps at best.

After more days... weeks... months than I'm willing to admit, I finally came to find that issue lies with the way the PS3 Eye transfers the frames over USB. The PS3 Eye captures video in the uncompressed YUYV pixel format and transfers the frames in bulk mode, which guarantees data transmission but has no guarantee of latency. In the case of the BBB, the large amount of data being sent by the webcam saturates the bulk allotment on the BBB's USB bandwidth and select timeout errors result. [3]

Shift to the Logitech C920 and MJPEG Format

In order to reduce the USB bandwidth required by the webcam, a compressed pixel format such as MJPEG or H.264 can be used. The PS3 Eye does not support video compression, so I looked to the Logitech C920 USB webcam instead. H.264 compression comes at a cost however, and will set you back about $72 to purchase a C920 on Amazon. (Other cameras, such as the Logitech C270 also support the MJPEG pixel format, which I ended up using over H.264. However, using the Logitech C270 will require a little extra work. The C270 does not include the Huffman table as part of the MJPEG stream and may need to be added for the video frames to be correctly read by OpenCV and other programs. See [3] for more.)

Since the C920 transfers compressed images in *isochronos* mode, it can easily deliver 640×480 frames at 30 fps using very little CPU. [3] If you save the video frames to individual JPEG image files, you can easily transfer them to your desktop computer and view them in any image viewer. If we want to use the MJPEG stream in a vision application written with OpenCV though, these images will have to be decompressed in real-time and converted to a cv::Mat object so that OpenCV can work with the image.

JPEG Decompression with OpenCV

Luckily, OpenCV includes functions for decoding images from a buffer, specifically the `cvDecodeImage()` and `imdecode()` functions, depending on if you are working in C or C++. [4] [5] The primary reason for using MJPEG over the H.264 compression format is that MJPEG uses *intra*frame compression whereas H.264 uses *inter*frame compression. Put simply, each MJPEG frame gets compressed individually as a JPEG image and each compressed frame is independent of all others. H.264 on the other hand, uses interframe prediction to "take advantage from temporal redundancy between neighboring frames to achieve higher compression rates". [6] While this is good for compressing video streams meant to be viewed as a continuous stream, it is not well-suited for embedded vision applications since H.264 decompression is CPU intensive and can exhibit decompression artifacts and lag when there is a lot of motion in the video.

If you installed OpenCV on your BBB with a package manager such as Ubuntu's Advanced Packaging Tool (`apt-get`) or Ångstrom's `opkg`, odds are that you will still only see about 10-20 fps when you try and capture at 640×480 resolution at the 30 fps setting. And if you profile your code by including calls to `time()` from the `<ctime>` header file, you will see that most of the time spent by your program is dedicated to decoding the image and converting it to the cv::Mat object. Moreover, the decompression eats up nearly all of the CPU. Luckily, there are some steps that you can take to significantly reduce decompression time and CPU usage—leaving more time and resources for your vision program to process the frames.

How-To: Achieve 30 fps

Disclaimer

Please keep in mind that I am *not* an expert in embedded Linux, OpenCV, or C/C++ programming. I am a graduate student studying aerospace engineering. I have only been working with embedded hardware, Linux, OpenCV, and C/C++ for about a year. My thesis has taken me on a detour into investigating ways to improve the framerate when using a USB webcam with the BeagleBone Black. This "How-To" is essentially a compilation of other resources and an outline of the

steps that I used to solve this particular problem—your mileage may vary. As always, it is your responsibility to understand the commands you are invoking. I have spent a lot of time on this problem and have relied heavily on the help of the open source community. I have put this guide together as a way of giving back to the open source community, and hope that some can find it useful. If you have any comments or suggestions for improving this "How-To" please email me at the address provided at the top of this page.

Prerequisites

This process can be followed with some slight variations to your setup. For reference, here is a list of what hardware and software I used.

- BeagleBone Black, Rev. A5
- BBB running Ubuntu 13.04 eMMC "flasher" image provided by [7]
- Logitech C920 USB webcam
- x86 PC running Ubuntu 13.04
- LAN network
- USB thumb drive

Before attempting any of these steps, you should already be familiar with the basics of using `ssh` to connect to the BBB over USB or a LAN network. You should also be comfortable working from the Linux command line and have some experience with GNU compilers, cmake, and the "configure, make, make install" build processes.

Objective

The main objective of this How-to is to take advantage of NEON hardware acceleration available on the BBB. [8] The details of how NEON acceleration works are a bit over my head, but its usefulness is obvious: "NEON technology can accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis by at least 3x the performance of ARMv5 and at least 2x the performance of ARMv6 SIMD." You can also see the clear benefits here. [16]

In order to use NEON, we will (1) build and install a more optimized JPEG codec called "libjpeg-turbo", and (2) rebuild OpenCV with NEON enabled. The latter is a bit tricky due to the limited processing power and storage capacity of the BBB. To speed up the build, I will introduce an easy way to cross-compile large projects for the BBB.

1. Install libjpeg-turbo

libjpeg-turbo is a highly-optimized version of the libjpeg JPEG codec library that is designed to take advantage of NEON acceleration. According to the libjpeg-turbo project page, the library is capable of encoding/decoding JPEG images 2–4 times faster than the standard libjpeg library. [9]

On the BBB, download the libjpeg-turbo source tarball and then extract it.

```
wget http://sourceforge.net/projects/libjpeg-turbo/files/1.3.0/libjpeg-turbo-1.3.
0.tar.gz
tar xzvf libjpeg-turbo-1.3.0.tar.gz
```

Enter the source directory, then create a build directory and enter it.

```
cd libjpeg-turbo-1.3.0
mkdir build
cd build
```

By default libjpeg-turbo will install into `/opt/libjpeg-turbo`. You may install to a different directory by passing the `--prefix` option to the configure script. However, the remainder of these instructions will assume that libjpeg-turbo was installed in its default location.

```
../configure CPPFLAGS='-O3 -pipe -fPIC -mfpu=neon -mfloat-abi=hard'
make
sudo make install
```

Note that the `-O3`, `-fPIC`, and `-mfpu=neon` are particularly important as they enable code optimization, position-independent code generation, and NEON hardware acceleration, respectively.

2. Setting up for Distributed Cross-Compilation (Optional, but Recommended)

Since OpenCV is such a large project and the BBB has limited processing power, it is much more convenient to set up cross-compilation. Typically, setting up a cross-compilation environment can be a tedious process and is especially cumbersome when building a large project that has many dependencies which, in turn, depend on other dependencies, etc. . . etc.

Fortunately with *distributed* cross compiling, you can take advantage of the libraries already installed on the BBB while still using your (probably x86) PC to cross-compile the object files much faster than if they were compiled locally. With distributed cross-compilation you can execute the build from the BBB just as if you were building on the BBB, itself. Here is how you can set up a distributed cross-compiler with `distcc`: [10]

On your PC, download the 32-bit version of Linaro GCC and the associated libraries. I found the appropriate cross-compiler from [11].

```
sudo apt-get install ia32-libs
wget https://launchpad.net/linaro-toolchain-binaries/trunk/2013.08/+download/gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux.tar.xz
```

Extract the files and set the cross-compiler (`CC`) to the one you just installed

```
tar xzvf gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux.tar.xz
export CC=‘pwd‘/gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux/bin/arm-linux-gnueabihf-
```

Confirm that the correct cross-compiler is active.

```
${CC}gcc --version

>> arm-linux-gnueabihf-gcc (crosstool-NG linaro-1.13.1-4.8-2013.08 - Linaro GCC 2013.08) 4.8.2 20130805 (prerelease)
>> Copyright (C) 2013 Free Software Foundation, Inc.
>> This is free software; see the source for copying conditions.  There is NO
>> warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Next, we will test that the compiler works by cross-compiling a simple test program for the BBB. Create a file called `hello_world.c` and paste into it the following:

```
int main(void)
{
  printf("Hello, cross-compilation world !\n");
  return 0;
}
```

Compile the program with:

```
${CC}gcc hello_world.c -o hello_world
```

Try running the program on your PC and confirm that it does *not* run.

```
./hello_world
```

You should be returned an error similar to the following:

```
bash:  ./hello_world:  cannot execute binary file
```

Now copy the executable to the BBB using `scp`. Your command should be similar to:

```
scp hello_world ubuntu@192.168.7.2:~/hello_world
```

Then execute the program on the BBB. You should see the “Hello, cross-compilation world !” message written to stdout.

```
cd ~
./hello_world
```

```
>> Hello, cross-compilation world !
```

If the above example of cross-compilation worked, you can now move on to setting up for *distributed* cross-compilation. We will begin by installing `distcc` on *both* the PC and the BBB. According to [11], we should build the most recent version of `distcc` from source to take advantage of a few features.

On **both** the PC and the BBB, begin by installing some prerequisite packages.

```
sudo apt-get install subversion autoconf automake python python-dev binutils-dev
libgtk2.0-dev
```

Again, on both machines, download the distcc source, and install it

```
svn checkout http://distcc.googlecode.com/svn/trunk/distcc-read-only
cd distcc-read-only
./autogen.sh
./configure --with-gtk --disable-Werror
make
sudo make install
```

Now that `distcc` is installed, we need to create some symbolic links on the BBB. Be careful to use the correct paths here or else you may overwrite one or more of your GNU compilers.

First, check which `gcc` and `distcc` are being called by default

```
which gcc
>> /usr/bin/gcc
which distcc
>> /usr/local/bin/distcc
```

Create the symlinks.

```
sudo ln -s /usr/local/bin/distcc /usr/local/bin/gcc
sudo ln -s /usr/local/bin/distcc /usr/local/bin/g++
sudo ln -s /usr/local/bin/distcc /usr/local/bin/c++
sudo ln -s /usr/local/bin/distcc /usr/local/bin/cpp
```

Now check your `PATH` variable to see if `/usr/local/bin` is included *before* `/usr/bin`. If it is not, then prepend `/usr/local/bin` to your `PATH`. For example:

```
echo $PATH
>> /usr/sbin:/usr/bin:/sbin:/bin
export PATH=/usr/local/bin:$PATH
```

So now you can check `distcc` is being called correctly (through the symlink you just created) by checking the following:

```
which gcc
/usr/local/bin/gcc
```

If this doesn't check out, then go back and make sure that you have all the symbolic links correct and prepended `/usr/local/bin` to your `PATH` variable. It is important that you add to the *beginning* of the path since gcc/cc/g++/c++ get called in the order they appear on the path.

Now we will create some environment variables for `distcc` in order to control some settings. I will just introduce the commands here, but if you want to read more, refer to [11] and [12]. You will need to have a working network (Either over LAN or USB through which the BBB and your PC can communicate.) The first environment variable sets the IP address of your PC that will be doing the compilation followed by a forward slash and the "number of jobs per machine". A good rule of thumb is to use twice your number of processor cores. So for me, I would use:

```
export DISTCC_HOSTS="192.168.2.3/4"
```

For the rest of the environment variables use the following:

```
export DISTCC_BACKOFF_PERIOD=0
export DISTCC_IO_TIMEOUT=3000
export DISTCC_SKIP_LOCAL_RETRY=1
```

In my case, I found that I had to execute the following command on the BBB before trying to build OpenCV so that the cmake build process would use `gcc` rather than `cc`, which I did not have installed:

```
export CC=/usr/local/bin/gcc
```

Now, we have to move back to the *PC* and create some similar symlinks.

```
cd gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux/bin
ln -s arm-linux-gnueabihf-gcc gcc
ln -s arm-linux-gnueabihf-cc cc
ln -s arm-linux-gnueabihf-g++ g++
ln -s arm-linux-gnueabihf-c++ c++
ln -s arm-linux-gnueabihf-cpp cpp
```

You will have to prepend to your path as well to make the cross-compiler active.

```
export PATH=$HOME/gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux/bin:$PATH
which gcc
>> /home/uname/gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux/bin/gcc
```

When you are about ready to start compiling OpenCV, you can launch the `distcc` daemon with the following command (Your command may be different depending on the number of jobs, and your BBB's IP address):

```
distccd --daemon --jobs 4 --allow 192.168.2.9 --verbose --log-stderr --no-detach
```

At this point, you could begin building/compiling any program or project on the BBB and you should see some activity on your PC as it receives the jobs from the BBB and compiles the object files. In this case, you probably want to build OpenCV—for which you should go through the next section. But once you are done building, you will most likely want to be able to disable distcc so that you can compile programs natively on your BBB and PC again. Here is how you can disable `distcc`.

On the BBB, disable all of the symlinks you created.

```
sudo rm /usr/local/bin/{gcc, g++, cpp, c++}
```

The easiest way to restore all of your environment variables is simply to restart the BBB.

```
sudo reboot
```

Now you will want to confirm that gcc is called instead of distcc. You can use the following commands. You may also want to try compiling a small hello_world.c example.

```
which gcc
>> /usr/bin/gcc
```

If you end up with errors that the gcc compiler doesn't exist, then you can remove then reinstall the compilers with:

```
sudo apt-get remove --purge build-essential
sudo apt-get install build-essential
```

Then we will do similarly on the PC. Make sure you are in the directory that you installed the cross-compiler: `gcc-linaro-arm-linux-gnueabihf-4.8-2013.08_linux/bin`

```
sudo rm gcc cc cpp g++ c++
```

Then reboot your computer to restore the `PATH`. Again, test to confirm that you can compile programs locally on your PC.

3. Building OpenCV with libjpeg-turbo and NEON

Now we will build OpenCV with libjpeg-turbo as the JPEG codec and with NEON hardware acceleration enabled. You do not have to use distributed cross-compiling, as described above, but it will dramatically reduce the build time. Due to the limited storage on the BBB, you will probably need some kind of external storage. I actually used a 16 GB μSD card and a USB card reader, but a regular USB thumb drive will probably work fine.

The first thing we have to do is reformat the USB drive with an ext2 partition. You might be able to use another kind of filesystem, but it needs to support symbolic links—which ext2 does. Depending on your OS this process will be different, but if your are running Ubuntu insert the USB drive and start up GParted. [13] (If GParted is not installed, you can install it with `sudo apt-get install gparted`.) In the top right corner select your USB drive. *Please be **certain** that you have chosen the correct device, or else you could end up erasing data on your PC's hard drive.* Go to Device > Create Partition Table and accept the prompt. Then to create the ext2 partition, right click on the "unallocated space" partition and in the File System drop-down menu, select ext2 and "Add" the partition. When complete, eject the USB drive and remove it.

From the BBB's terminal we need to mount the device. To know which drive is the USB drive, use the following command, insert the USB drive, and repeat the command. The new drive is the thumb drive. I will assume this `/dev/sda` with one partition, `/dev/sda1` as it was for me.

```
ls /dev/sd*
```

I found that it was easiest to do the following commands as root. (Be careful!)

```
sudo su
```

Create a mount point and mount the filesystem.

```
mkdir -p /mnt/ext2
mount -t ext2 /dev/sda1 /mnt/ext2
```

Navigate into the directory and download the OpenCV source code, then extract the files. [14]

```
cd /mnt/ext2
wget git clone https://github.com/Itseez/opencv.git
```

Enter the directory that is created—whatever it is called. Then create a build directory and enter that.

```
cd OpenCV
mkdir release
cd release
```

Now we will run cmake with a bunch of flags that should enable libjpeg-turbo and NEON. note that some of these flags, such as `USE_VFPV3=ON` and `USE_NEON=ON` may have no effect, as they only work when cross-compiling without `distcc` [15]. That is okay—the `-mfpu=neon` flag will enable NEON for us. I've just gone ahead and included all of the flags that I used anyways. If you are using `distcc` to cross-compile, make sure that you see some activity in your PC's terminal after you execute the cmake command. (The PC should compile a few programs while cmake tests the compilers it has available.)

```
cmake -D CMAKE_C_FLAGS='-O3 -mfpu=neon -mfloat-abi=hard' -D
CMAKE_CXX_FLAGS='-O3 -mfpu=neon -mfloat-abi=hard' -D CMAKE_BUILD_TYPE=RELEASE
-D CMAKE_INSTALL_PREFIX=/usr/local -D BUILD_PYTHON_SUPPORT=ON -DWITH_JPEG=ON
-DBUILD_JPEG=OFF -DJPEG_INCLUDE_DIR=/opt/libjpeg-turbo/include/
-DJPEG_LIBRARY=/opt/libjpeg-turbo/lib/libjpeg.a -DUSE_VFPV3=ON -DUSE_NEON=ON ..
```

Take careful note of the information displayed to stdout after cmake runs. You will want to make sure that `FFMPEG` support is enabled, `-mfpu=neon` appears in the release build flags, and that the JPEG codec is libjpeg-turbo. If everything looks okay, go ahead and begin the build... Even with `distcc`, the build will take awhile.

```
make
```

Now you can install the files to their default location, `/usr/local` and exit as the root user.

```
make install
exit
```

At this point, you will want to make sure that you can compile a simple OpenCV program. You can use the `framegrabberCV.c` program used in the testing section. I did this by navigating to the directory where I saved the program and executing: (See the Testing section)

```
gcc framegrabberCV.c -o framegrabberCV `pkg-config --cflags --opencv opencv
libv4l2` -lm
```

If the program compiles successfully, its likely that everything installed correctly. We can now unmount the USB drive. (NOTE: I had some issues with the USB drive being corrupted after removing it from the BBB. You may want to first compress the OpenCV source directory (including build files) to a `.tar.gz` and transfer it to your PC using `scp`.) When you are ready to remove the drive:

```
sudo umount /dev/sda1
sudo umount /mnt/ext2
sudo eject /dev/sda
sudo rm -rf /mnt/ext2
```

4. Testing

Testing the Framerate

Now it's time to test the framerate. First, download the program `framegrabberCV.c` to the BBB and compile it.

```
gcc framegrabberCV.c -o framegrabberCV `pkg-config --cflags --opencv opencv
libv4l2` -lm
```

I wanted to make sure the processor was running at its full 1 GHz, so I set it using:

```
sudo cpufreq-set -g performance
```

Note that you can set this back to default by executing:

```
sudo cpufreq-set -g ondemand
```

Make sure that the webcam is plugged in. Now you can test the framerate. [17]

```
time ./framegrabberCV -f mjpeg -H 480 -W 640 -c 1000 -I 30 -o
```

Now take the the number of frames you captured and converted to OpenCV image objects (in this case, 1000) and divide it by the "real" time provided by the `time` function to get the framerate.

Incorporating Custom Capture Code as an OpenCV object

If you like, you can make some changes to the custom capture code in `frmegrabberCV.c` and compile it as a C++ class instead of a standalone command line program. That way you can create a capture object within your OpenCV code, grab and decode images, and process it using OpenCV functions and methods.

< To be continued...>

For now, take a look at [18]

Acknowledgments

I would like to give thanks to those in the open source community who have been enormously helpful. Thanks to Matthew Witherwax for all of his help in arriving at a solution [3]; Martin Fox for providing custom Video4Linux capture code [18]; Alexander Corcoran for answering my newbie Linux questions [19]; Robert C. Nelson for maintaining Ubuntu for BBB [7]; Max Thrun for answering some questions about the PS3 Eye drivers [20]; and everyone else who has had the patience to help me along my way.

References

[1] http://mechomaniac.com/node/32

[2] http://linuxtv.org/downloads/v4l-dvb-apis/

[3] http://blog.lemoneerlabs.com/post/BBB-webcams

[4] http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html

[5] http://blog.lemoneerlabs.com/post/opencv-mjpeg

[6] http://en.wikipedia.org/wiki/Inter_frame

[7] http://elinux.org/BeagleBoardUbuntu

[8] http://www.arm.com/products/processors/technologies/neon.php

[9] http://libjpeg-turbo.virtualgl.org

[10] http://jeremy-nicola.info/portfolio-item/cross-compilation-distributed-compilation-for-the-raspberry-pi/

[11] http://eewiki.net/display/linuxonarm/BeagleBone+Black#BeagleBoneBlack-ARMCrossCompiler:GCC

[12] https://code.google.com/p/distcc/

[13] http://www.sitepoint.com/ubuntu-12-04-lts-precise-pangolin-using-gparted-to-partition-a-hard-disk/

[14] http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html#linux-installation

[15] http://docs.opencv.org/doc/tutorials/introduction/crosscompilation/arm_crosscompile_with_cmake.html

[16] http://elinux.org/images/1/1c/Optimizing_the_Embedded_Platform_Using_OpenCV.pdf

[17] https://groups.google.com/forum/#!msg/beagleboard/G5Xs2JuwD_4/gCl48Nj61BwJ

[18] https://bitbucket.org/beldenfox/cvcapture/src

[19] http://digitalcommons.calpoly.edu/cpesp/50/

[20] http://bear24rw.blogspot.com/2009/11/ps3-eye-driver-patch.html

Appendix

framegrabberCV.c (Matthew Witherwax)

Download

```
/*******************************************************************************
*   framegrabber Version 0.1                                                   *
*   Copyright (C) 2013 by Matthew Witherwax (lemoneer)                         *
*   lemoneer@outlook.com                                                       *
*   blog.lemoneerlabs.com                                                      *
*                                                                              *
*   based on V4L2 Specification, Appendix B: Video Capture Example             *
*   (http://linuxtv.org/downloads/v4l-dvb-apis/capture-example.html)           *
*   and work by Matthew Witherwax on v4l2grab                                  *
*   (https://github.com/twam/v4l2grab)                                         *
*******************************************************************************
*******************************************************************************
                            BSD LICENSE

Copyright (c) 2013, Matthew Witherwax
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <assert.h>
#include <getopt.h>             /* getopt_long() */
#include <fcntl.h>              /* low-level i/o */
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
```

```
#include <sys/ioctl.h>
#include <time.h>

#include <linux/videodev2.h>

#include "opencv2/core/core_c.h"
#include "opencv2/highgui/highgui_c.h"

#define CLEAR(x) memset(&(x), 0, sizeof(x))

enum io_method {
IO_METHOD_READ,
IO_METHOD_MMAP,
IO_METHOD_USERPTR,
};

struct buffer {
void *start;
size_t length;
};

static char         *dev_name = "/dev/video0";
static enum         io_method io = IO_METHOD_MMAP;
static int          fd = -1;
struct buffer       *buffers;
static unsigned int n_buffers;
static int          out_buf;
static int          frame_count = 1;
static int          set_format;
static unsigned int width = 640;
static unsigned int height = 480;
static unsigned int fps = 30;
static unsigned int timeout = 1;
static unsigned int timeouts_max = 1;
static char         *out_name = "capture.jpg";

/* Allowed formats: V4L2_PIX_FMT_YUYV, V4L2_PIX_FMT_MJPEG, V4L2_PIX_FMT_H264
*  The default will not be used unless the width and/or height is specified
*  but the user does not specify a pixel format */
static unsigned int pixel_format = V4L2_PIX_FMT_MJPEG;

/* Signal Handling
 * Clean up on Ctrl-C as opposed to leaving
 * the device in an inconsistent state*/
static int s_interrupted = 0;
static void s_signal_handler (int signal_value)
{
s_interrupted = 1;
}

static void s_catch_signals (void)
{
struct sigaction action;
action.sa_handler = s_signal_handler;
action.sa_flags = 0;
sigemptyset (&action.sa_mask);
sigaction (SIGINT, &action, NULL);
sigaction (SIGTERM, &action, NULL);
}
```

```
static void errno_exit(const char *s) {
fprintf(stderr, "%s error %d, %s\n", s, errno, strerror(errno));
exit(EXIT_FAILURE);
}

static int xioctl(int fh, int request, void *arg) {
int r;

do {
r = ioctl(fh, request, arg);
} while (-1 == r && EINTR == errno);

return r;
}

static int countP = 0;
static int cc = 0;
static void process_image(const void *p, int size) {
        if (out_buf) {
// Mike, Take this out to convert
// every frame captured
if (countP % 3 != 0)
{
countP += 1;
return;
}
countP += 1;
cc += 1;
CvMat mat;
IplImage * img;

mat = cvMat(480, 640, CV_8UC3, (void*)p);

// decode the image
img = cvDecodeImage(&mat, 1);

            // release the image
cvReleaseImage(&img);
}
}

static int read_frame(void) {
struct v4l2_buffer buf;
unsigned int i;

switch (io) {
case IO_METHOD_READ:
if (-1 == read(fd, buffers[0].start, buffers[0].length)) {
switch (errno) {
case EAGAIN:
return 0;

case EIO:
/* Could ignore EIO, see spec. */

/* fall through */

default:
errno_exit("read");
}
```

```
}

process_image(buffers[0].start, buffers[0].length);
break;

case IO_METHOD_MMAP:
CLEAR(buf);

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;

if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
switch (errno) {
case EAGAIN:
return 0;

case EIO:
/* Could ignore EIO, see spec. */

/* fall through */

default:
errno_exit("VIDIOC_DQBUF");
}
}

assert(buf.index < n_buffers);

process_image(buffers[buf.index].start, buf.bytesused);

if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
errno_exit("VIDIOC_QBUF");
break;

case IO_METHOD_USERPTR:
CLEAR(buf);

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_USERPTR;

if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
switch (errno) {
case EAGAIN:
return 0;

case EIO:
/* Could ignore EIO, see spec. */

/* fall through */

default:
errno_exit("VIDIOC_DQBUF");
}
}

for (i = 0; i < n_buffers; ++i)
if (buf.m.userptr == (unsigned long) buffers[i].start
&& buf.length == buffers[i].length)
break;
```

```
assert(i < n_buffers);

process_image((void *) buf.m.userptr, buf.bytesused);

if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
errno_exit("VIDIOC_QBUF");
break;
}

return 1;
}

static void grab_frames(void) {
        clock_t begin, end;
        double time_spent;

unsigned int count;
unsigned int timeout_count;

count = frame_count;
timeout_count = timeouts_max;

        begin = clock();

while (count-- > 0) {
for (;;) {

if (s_interrupted) {
fprintf(stderr, "\nInterrupt received - aborting capture\n");
return;
}

fd_set fds;
struct timeval tv;
int r;

FD_ZERO(&fds);
FD_SET(fd, &fds);

/* Timeout. */
tv.tv_sec = timeout;
tv.tv_usec = 0;

r = select(fd + 1, &fds, NULL, NULL, &tv);

if (-1 == r) {
if (EINTR == errno)
continue;
errno_exit("select");
}

if (0 == r) {
if (timeout_count > 0) {
timeout_count--;
} else {
fprintf(stderr, "select timeout\n");
exit(EXIT_FAILURE);
}
}
```

```
if (read_frame())
break;
/* EAGAIN - continue select loop. */
}
}
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        fprintf(stderr, "Captured %i frames and Processed %i in %f seconds\n", frame_count, cc, time_spent);
}

static void mainloop(void) {
grab_frames();
}

static void stop_capturing(void) {
enum v4l2_buf_type type;

switch (io) {
case IO_METHOD_READ:
/* Nothing to do. */
break;

case IO_METHOD_MMAP:
case IO_METHOD_USERPTR:
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (-1 == xioctl(fd, VIDIOC_STREAMOFF, &type))
errno_exit("VIDIOC_STREAMOFF");
break;
}
}

static void start_capturing(void) {
unsigned int i;
enum v4l2_buf_type type;

switch (io) {
case IO_METHOD_READ:
/* Nothing to do. */
break;

case IO_METHOD_MMAP:
for (i = 0; i < n_buffers; ++i) {
struct v4l2_buffer buf;

CLEAR(buf);
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index = i;

if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
errno_exit("VIDIOC_QBUF");
}
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
errno_exit("VIDIOC_STREAMON");
break;

case IO_METHOD_USERPTR:
for (i = 0; i < n_buffers; ++i) {
struct v4l2_buffer buf;
```

```
CLEAR(buf);
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_USERPTR;
buf.index = i;
buf.m.userptr = (unsigned long) buffers[i].start;
buf.length = buffers[i].length;

if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
errno_exit("VIDIOC_QBUF");
}
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
errno_exit("VIDIOC_STREAMON");
break;
}
}

static void uninit_device(void) {
unsigned int i;

switch (io) {
case IO_METHOD_READ:
free(buffers[0].start);
break;

case IO_METHOD_MMAP:
for (i = 0; i < n_buffers; ++i)
if (-1 == munmap(buffers[i].start, buffers[i].length))
errno_exit("munmap");
break;

case IO_METHOD_USERPTR:
for (i = 0; i < n_buffers; ++i)
free(buffers[i].start);
break;
}

free(buffers);
}

static void init_read(unsigned int buffer_size) {
buffers = calloc(1, sizeof (*buffers));

if (!buffers) {
fprintf(stderr, "Out of memory\n");
exit(EXIT_FAILURE);
}

buffers[0].length = buffer_size;
buffers[0].start = malloc(buffer_size);

if (!buffers[0].start) {
fprintf(stderr, "Out of memory\n");
exit(EXIT_FAILURE);
}
}

static void init_mmap(void) {
struct v4l2_requestbuffers req;
```

```
CLEAR(req);

req.count = 4;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;

if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
if (EINVAL == errno) {
fprintf(stderr, "%s does not support "
"memory mapping\n", dev_name);
exit(EXIT_FAILURE);
} else {
errno_exit("VIDIOC_REQBUFS");
}
}

if (req.count < 2) {
fprintf(stderr, "Insufficient buffer memory on %s\n",
dev_name);
exit(EXIT_FAILURE);
}

buffers = calloc(req.count, sizeof (*buffers));

if (!buffers) {
fprintf(stderr, "Out of memory\n");
exit(EXIT_FAILURE);
}

for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
struct v4l2_buffer buf;

CLEAR(buf);

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index = n_buffers;

if (-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
errno_exit("VIDIOC_QUERYBUF");

buffers[n_buffers].length = buf.length;
buffers[n_buffers].start =
mmap(NULL /* start anywhere */,
buf.length,
PROT_READ | PROT_WRITE /* required */,
MAP_SHARED /* recommended */,
fd, buf.m.offset);

if (MAP_FAILED == buffers[n_buffers].start)
errno_exit("mmap");
}
}

static void init_userp(unsigned int buffer_size) {
struct v4l2_requestbuffers req;

CLEAR(req);
```

```
req.count = 4;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_USERPTR;

if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
if (EINVAL == errno) {
fprintf(stderr, "%s does not support "
"user pointer i/o\n", dev_name);
exit(EXIT_FAILURE);
} else {
errno_exit("VIDIOC_REQBUFS");
}
}

buffers = calloc(4, sizeof (*buffers));

if (!buffers) {
fprintf(stderr, "Out of memory\n");
exit(EXIT_FAILURE);
}

for (n_buffers = 0; n_buffers < 4; ++n_buffers) {
buffers[n_buffers].length = buffer_size;
buffers[n_buffers].start = malloc(buffer_size);

if (!buffers[n_buffers].start) {
fprintf(stderr, "Out of memory\n");
exit(EXIT_FAILURE);
}
}
}

static void init_device(void) {
struct v4l2_capability cap;
struct v4l2_cropcap cropcap;
struct v4l2_crop crop;
struct v4l2_format fmt;
struct v4l2_streamparm frameint;
unsigned int min;

if (-1 == xioctl(fd, VIDIOC_QUERYCAP, &cap)) {
if (EINVAL == errno) {
fprintf(stderr, "%s is no V4L2 device\n",
dev_name);
exit(EXIT_FAILURE);
} else {
errno_exit("VIDIOC_QUERYCAP");
}
}

if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
fprintf(stderr, "%s is no video capture device\n",
dev_name);
exit(EXIT_FAILURE);
}

switch (io) {
case IO_METHOD_READ:
if (!(cap.capabilities & V4L2_CAP_READWRITE)) {
fprintf(stderr, "%s does not support read i/o\n",
```

```
dev_name);
exit(EXIT_FAILURE);
}
break;

case IO_METHOD_MMAP:
case IO_METHOD_USERPTR:
if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
fprintf(stderr, "%s does not support streaming i/o\n",
dev_name);
exit(EXIT_FAILURE);
}
break;
}


/* Select video input, video standard and tune here. */


CLEAR(cropcap);

cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (0 == xioctl(fd, VIDIOC_CROPCAP, &cropcap)) {
crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
crop.c = cropcap.defrect; /* reset to default */

if (-1 == xioctl(fd, VIDIOC_S_CROP, &crop)) {
switch (errno) {
case EINVAL:
/* Cropping not supported. */
break;
default:
/* Errors ignored. */
break;
}
}
} else {
/* Errors ignored. */
}


CLEAR(fmt);

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (set_format) {
fmt.fmt.pix.width = width;
fmt.fmt.pix.height = height;
fmt.fmt.pix.pixelformat = pixel_format;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;

if (-1 == xioctl(fd, VIDIOC_S_FMT, &fmt))
errno_exit("VIDIOC_S_FMT");

                if (fmt.fmt.pix.pixelformat != pixel_format) {
                        fprintf(stderr,"Libv4l didn't accept pixel format. Can't proceed.\n");
                        exit(EXIT_FAILURE);
                }

/* Note VIDIOC_S_FMT may change width and height. */
```

```
} else {
/* Preserve original settings as set by v4l2-ctl for example */
if (-1 == xioctl(fd, VIDIOC_G_FMT, &fmt))
errno_exit("VIDIOC_G_FMT");
}

CLEAR(frameint);

/* Attempt to set the frame interval. */
frameint.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
frameint.parm.capture.timeperframe.numerator = 1;
frameint.parm.capture.timeperframe.denominator = fps;
if (-1 == xioctl(fd, VIDIOC_S_PARM, &frameint))
fprintf(stderr, "Unable to set frame interval.\n");

/* Buggy driver paranoia. */
min = fmt.fmt.pix.width * 2;
if (fmt.fmt.pix.bytesperline < min)
fmt.fmt.pix.bytesperline = min;
min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
if (fmt.fmt.pix.sizeimage < min)
fmt.fmt.pix.sizeimage = min;

switch (io) {
case IO_METHOD_READ:
init_read(fmt.fmt.pix.sizeimage);
break;

case IO_METHOD_MMAP:
init_mmap();
break;

case IO_METHOD_USERPTR:
init_userp(fmt.fmt.pix.sizeimage);
break;
}
}

static void close_device(void) {
if (-1 == close(fd))
errno_exit("close");

fd = -1;
}

static void open_device(void) {
struct stat st;

if (-1 == stat(dev_name, &st)) {
fprintf(stderr, "Cannot identify '%s': %d, %s\n",
dev_name, errno, strerror(errno));
exit(EXIT_FAILURE);
}

if (!S_ISCHR(st.st_mode)) {
fprintf(stderr, "%s is no device\n", dev_name);
exit(EXIT_FAILURE);
}

fd = open(dev_name, O_RDWR /* required */ | O_NONBLOCK, 0);
```

```
if (-1 == fd) {
fprintf(stderr, "Cannot open '%s': %d, %s\n",
dev_name, errno, strerror(errno));
exit(EXIT_FAILURE);
}
}

static void usage(FILE *fp, int argc, char **argv) {
fprintf(fp,
"Usage: %s [options]\n\n"
"Version 1.0\n"
"Options:\n"
"-d | --device name   Video device name [%s]\n"
"-h | --help          Print this message\n"
"-m | --mmap          Use memory mapped buffers [default]\n"
"-r | --read          Use read() calls\n"
"-u | --userp         Use application allocated buffers\n"
"-W | --width         Set image width\n"
"-H | --height        Set image height\n"
"-I | --interval      Set frame interval (fps) [%i]\n"
"-f | --format        Set pixel format [YUYV | MJPG | H264]\n"
"-t | --timeout       Set capture timeout in seconds [%i]\n"
"-T | --timeouts-max  Set the maximum number of timeouts [%i]\n"
"-o | --output        Outputs stream to stdout\n"
"-c | --count         Number of frames to grab [%i]\n"
"",
argv[0], dev_name, fps, timeout, timeouts_max, frame_count);
}

static const char short_options[] = "d:hmruW:H:I:f:t:T:oc:";

static const struct option
long_options[] = {
{ "device",        required_argument, NULL, 'd'},
{ "help",          no_argument,       NULL, 'h'},
{ "mmap",          no_argument,       NULL, 'm'},
{ "read",          no_argument,       NULL, 'r'},
{ "userp",         no_argument,       NULL, 'u'},
{ "width",         required_argument, NULL, 'W'},
{ "height",        required_argument, NULL, 'H'},
{ "interval",      required_argument, NULL, 'I'},
{ "format",        required_argument, NULL, 'f'},
{ "timeout",       required_argument, NULL, 't'},
{ "timeouts-max",  required_argument, NULL, 'T'},
{ "output",        no_argument,       NULL, 'o'},
{ "count",         required_argument, NULL, 'c'},
{ 0, 0, 0, 0}
};

int main(int argc, char **argv) {
s_catch_signals ();
for (;;) {
int idx;
int c;

c = getopt_long(argc, argv,
short_options, long_options, &idx);

if (-1 == c)
```

```
break;

switch (c) {
case 0: /* getopt_long() flag */
break;

case 'd':
dev_name = optarg;
break;

case 'h':
usage(stdout, argc, argv);
exit(EXIT_SUCCESS);

case 'm':
io = IO_METHOD_MMAP;
break;

case 'r':
io = IO_METHOD_READ;
break;

case 'u':
io = IO_METHOD_USERPTR;
break;

case 'W':
// set width
width = atoi(optarg);
set_format++;
break;

case 'H':
// set height
height = atoi(optarg);
set_format++;
break;

case 'I':
// set fps
fps = atoi(optarg);
break;

case 'f':
// set pixel format
if (strcmp(optarg, "YUYV") == 0 || strcmp(optarg, "yuyv") == 0) {
pixel_format = V4L2_PIX_FMT_YUYV;
set_format++;
} else if (strcmp(optarg, "MJPG") == 0 || strcmp(optarg, "mjpg") == 0) {
pixel_format = V4L2_PIX_FMT_MJPEG;
set_format++;
} else if (strcmp(optarg, "H264") == 0 || strcmp(optarg, "h264") == 0) {
pixel_format = V4L2_PIX_FMT_H264;
set_format++;
}
break;

case 't':
// set timeout
timeout = atoi(optarg);
```

```
break;

case 'T':
// set max timeout
timeouts_max = atoi(optarg);
break;

case 'o':
out_buf++;
break;

case 'c':
errno = 0;
frame_count = strtol(optarg, NULL, 0);
if (errno)
errno_exit(optarg);
break;

default:
usage(stderr, argc, argv);
exit(EXIT_FAILURE);
}
}
        clock_t begin, end;
        double time_spent;

        begin = clock();
open_device();
init_device();
start_capturing();
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        fprintf(stderr, "Startup took %f seconds\n", time_spent);

mainloop();

        begin = clock();
stop_capturing();
uninit_device();
close_device();
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        fprintf(stderr, "Shutdown took %f seconds\n", time_spent);
fprintf(stderr, "\n");
return 0;
}
```