2016

Development of a swarm control platform for educational and research applications

Justin Noronha
Iowa State University

Development of a swarm control platform for educational and research applications

by

Justin Eugene Noronha

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Electrical Engineering

Program of Study Committee:
Nicola Elia, Major Professor
Umesh Vaidya
Phillip Jones

Iowa State University

Ames, Iowa

2016

DEDICATION

I dedicate this thesis to my Father, Colin, who always encouraged me to chase my dreams and supported me no matter what I chose to do. His fight with cancer this past year has shown me the meaning of true strength and has inspired me even on my darkest days. I look forward to spending many more years with him.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 2:

Chapter 3:

Chapter 4:

Chapter 5:

Chapter 6:

Appendix A:

Appendix B:

LIST OF TABLES

Chapter 4:

Chapter 6:

TERMINOLOGY

- Crazyflie – An open source nano-quadcopter system
- PCB – Printed Circuit Board
- FPGA – Field Programmable Gate Array
- Firmware – The software run onboard the Crazyflie PCB
- Client – The User run software on the PC base station.
- Radio – The USB radio antenna to communicate between Crazyflie and Client
- ack – Acknowledge (Packet Transmission)
- ARC – Auto-Retry Count
- ARD – Auto-Retry Delay
- R_x – Receive Radio Packets
- T_x – Transmit Radio Packets
- Trilateration – A method of position estimation using the intersection of 3 spheres (similar to Triangulation but without using any angles, only radii)
- UDP – User Datagram Protocol
- Camera System – OptiTrack Motion Capture Camera system
- Localization – Using some method to determine an objects position
- VRPN – Virtual Reality Peripheral Network (multicast UDP protocol)
- Distro – Linux Distribution
- GPS – Global Positioning System
- GUI – Graphical User Interface
- IR – Infrared

- OS – Operating System
- VM – Virtual Machine
- User – The Human running the Client
- ROS – Robotic Operating System

ACKNOWLEDGEMENTS

I would like to thank my Major Professor, Dr. Nicola Elia, for his generous support and guidance throughout my graduate career. I would also like to thank the National Science Foundation (NSF) for their financial support of my project via grant **CNS-1239319**. I would also like to thank Dr. Phillip Jones for sharing his wisdom in embedded systems and programming tips with me throughout the development of this project. I also want to thank Paul Uhing, Ian McInerney, Josh Bertram, and Matt Rich for their insight and help in the lab testing and troubleshooting any issues I couldn't solve on my own. I'm going to miss all those random Friday projects fixing up the lab. It was truly a pleasure working with all of you.

Lastly, I want to thank my family. Their love and support is what kept me motivated and was essential to my successes here at Iowa State.

ABSTRACT

Swarm robotics is a new and quickly growing field of research that has many applications to the real-world. The idea is to use a coordinated group of devices to perform tasks that are either unsafe or infeasible for a single device to accomplish.

While a lot of research is being done on swarms, surprisingly, there are not many physical platforms available to apply these ideas to. Being able to take this research out of simulation will greatly improve the quality and feasibility of these ideas outside of near-perfect conditions.

In this thesis we develop a new swarm control platform (that can function as either a Centralized or Distributed system), as well as potential research applications and educational exercises that can be created using this platform. The platform utilizes an open source Nano-Quadcopter called the Crazyflie and uses the OptiTrack motion capture camera system for localization. It is designed for scalability using the Virtual Reality Peripheral Network (VRPN), a type of Multi-Cast UDP protocol, to transmit localization data. We use this location data and set of nine nested PID controllers to command the Crazyflie to any location in space. It also supports flying multiple Crazyflies on one USB Radio, and multiple radios per computer, further reducing the scale up cost.

As a proof of concept, we crafted a few applications for the Platform to demonstrate its abilities. These examples range from simple single Crazyflie autonomous flight, to more complex gesture controlled multi-Crazyflie master-slave ‘follow-the-leader’ type systems. We intend to extend these systems to test even more complex problems in optimization and stochastic lossy networks, as well as creating lab experiments and educational resources for students to learn more about controls.

Based around C and C++, the objective of this platform is to provide an accessible and quick tool to researchers and professors to craft unique and interesting hands-on experiences in distributed control systems.

CHAPTER 1. INTRODUCTION

In swarm robotics a swarm is a coordinated group of devices, be it robots or sensors, that can perform a multitude of tasks that a single device would be unable to accomplish alone in a timely fashion. For example, a search and rescue mission would be greatly improved by the ability to spread out a number of drones and sweep them over a wide area. If any of the individual drones found some debris or object of interest in their area, then that information would be shared with the rest of the drones and they would autonomously reduce the search area until they converge on the target.

This can be done in one of two ways. We either have Centralized Control, where a central command station relays information between everyone, or Distributed Control, where the drones themselves are aware of their goal internally and there is no central command.

The platform we develop in this thesis can be either centralized or distributed, depending on the configuration the application calls for. We can set up the platform with any type of controller being calculated directly on the Client itself, for Centralized control, or we can take those same controllers programmed in the same language and place it in the Firmware of each Crazyflie instead for a Distributed control system. The goal of this platform is to be as open, versatile, and safe as possible. This way no matter what new ideas are conceived in the future, we will still be able to implement them into the platform and get results.

One of the major benefits of a system like this is its scalability. Not only in scaling the size of the swarm, but also in scaling the physical size of each Quadcopter. When testing new applications things don't always go to plan. The Crazyflie is a small quadcopter that can

only really do damage to itself, yet all of the designs we create for the Crazyflie can be physically scaled up to larger more powerful quadcopters with a single peripheral board. This means that, like in most engineering disciplines, we can run tests in the Crazyflie's small scale factor and then, once it's consistently functioning, we can scale up to a larger quadcopter. This is another way in which the system shows its safety and versatility.

1.1 Motivation

The ideal case was to find an open source platform that had swarm capabilities built in and ready to develop. Before our search we laid out the following characteristics as the most important:

- Inexpensive
- Accessible (Open Source)
- Pre-Built Swarm Functionality
- Scalable

Surprisingly, there were near to none available at all, let alone ones that fulfilled those criteria. There were only two options that came close. One was the Crazyflie, an open source inexpensive Nano-Quadcopter, and the other was the ZANO. The ZANO was an unreleased Kickstarter project at the time that proposed a quadcopter with swarm capabilities built in and utilizing Wi-Fi to network between quads. The ZANO was not completely open source but they were going to provide an API and development tools to further expand its capabilities. Each platform had its benefits and drawbacks.

The Crazyflie was completely open, so we would have access to how every system and subroutine was working behind the scenes. This is a huge benefit for developing a

platform like this because we can truly do anything we want with the platform. The downside is that it was going to take a lot of background work to get the swarm functionality implemented, and we really wanted to get right to developing the applications and educational exercises for the system instead.

The ZANO was the platform that showed the most promise for what we wanted to accomplish, it had the swarm functionality built in and used Wi-Fi to communicate so we could have it integrate with a variety of systems. The problem was that the project was still in its Kickstarter phases and was not available just yet. As with any Kickstarter project, we approached it with caution and we did not feel comfortable jumping right in until it had more public exposure and testing to verify all of their claims.

In the end we decided to use the Crazyflie as our base platform. Although it was going to take a lot of work to get set up, we felt it was a better option than taking a gamble on the promises of a project that hadn't been fully tested publicly yet. As it turns out, this was the right choice. A couple weeks after the ZANO's release people started reporting that it was nowhere near what they had been promising, and that it was going to need a lot of work to get it there. Then after only a couple months the company declared bankruptcy and left all of their backers with essentially nothing.

1.2 Literature Review

There is a huge variety of research being done in swarm robotics, examples range from replicating biological behavior [2] to studying consensus theory in multi-agent systems [1].

In [2] the authors discuss and simulate a particle swarm optimization algorithm that was originally used to model unpredictable social behaviors of animals, like birds flocking

together and how they respond to external stimuli. They found that the algorithm can be applied to optimizing any kind of dynamic system with very few parameters to adjust.

In [1] the authors found that they could apply the consensus principle to a multi-agent system (read: swarm) that would allow a coordinated group to hold a formation without any external commands. They even applied their simulation to a single Crazyflie and had decent tracking results. They were only able to test on a single Crazyflie though because, as of a year ago, that is all the Crazyflie could do. The work done in [1] was published at the same time we started development of our Crazyflie Swarm Platform, in July 2015.

With this much research being done in the theoretical realm, we were surprised to find that there were almost no physical systems being developed to apply these ideas to. So we started development of our own platform using the Crazyflie as a starting point.

Since we are working with an open source system, there are other researchers working in parallel with us in developing similar platforms for the Crazyflie but for different languages. In [3] Wolfgang Hoenig has been developing a Crazyflie swarm in the Robotic Operating System (ROS) language and using it to enable interaction between virtual and physical objects in different spaces, what he calls Mixed Reality.

Our platform differs in both our direction towards educational use, as well as research purposes and the language that it is designed in.

In the end, we want a platform that can take any of these theoretical ideas and apply them to a physical system. We can then take the results and mold them into educational exercises for students to see how well the results compare to simulation in non-idealized conditions, all within the same versatile platform.

1.3 Objective

The objective of this platform is to provide a versatile, accessible and modular tool to researchers and professors to craft unique and interesting hands-on experiences in distributed control systems.

The goal of the platform was to build upon an inexpensive robotics system that supported swarm control out of the box. We could then improve upon the systems and work on applying some theoretical research topics like stochastic lossy communication models as well as convex optimization techniques to the swarm behavior.

We also wanted to design the platform to be modular from the start. This way we could design any type of controller we wanted and just plug it into the code without having to modify any other systems.

1.4 Overview of Contributions

A brief summary of the contributions discussed in this thesis are covered here:

1. Migrated Client from the Virtual Machine to RedHat Linux distribution.
2. Improved *libcflie* Crazyflie Client Application Programmer Interface (API) [8]
    - User interface improvements and increased platform utility
3. Integrate the Camera System and VRPN into the Client
4. Implemented X, Y, Z, and Yaw PID's and Tuning
5. Increased Radio Link Scalability
6. Reduced Communications Packet Delay
7. Yaw Synchronization
8. Created Educational Resources and Exercises

- Created Crazyflie Swarm Platform Wiki
- PID Tuning and Model Derivation Educational Lab Exercise
- Trilateration Educational Lab Exercise

9. Created Applications using the Platform to Showcase its Research Potential
    - Multi-Crazyflie Gesture Controlled Swarm
    - Integrated Platform with a single-camera computer vision tracking system for alternative flight localization methods.

For a more in depth look at any of these contributions, see Chapter 4 for Platform Development (page 21), Chapter 5 for Applications (page 51), and Chapter 6 for Educational Resources (page 59).

CHAPTER 2. CRAZYFLIE PLATFORM

The Crazyflie is an open source project created and maintained by Bitcraze. There are currently two models available for purchase, the Crazyflie 1.0 and the Crazyflie 2.0. As the name implies, the Crazyflie 2.0 is the new and improved version of the original Crazyflie 1.0. The Crazyflie 2.0 has a larger battery, larger motors for more lift power, Bluetooth LE support, and an expansion board port that can auto-detect the attached peripheral.

For this thesis we will be working exclusively with Crazyflie 2.0's which we will refer to as just Crazyflie for all future mentions.

Figure 2.1 – High Level Platform Architecture

At the highest level, the Crazyflie consists of two independent systems, as shown in Figure 2.1. There's the firmware that resides on the Crazyflie itself, which performs critical functions such as sensor measurement, data logging, and flight stabilization, as well as many other smaller tasks. The second system is the software client that is run from a base station PC separate from the Crazyflie. This client maintains the flight control routines, VRPN location data stream, and data log storage. It can be thought of like a commander to the Crazyflies. These two systems communicate with each other over the air via a USB Radio Dongle that is plugged into the Client Computer.

It is important to emphasize the difference between the Firmware system and the Software Client, so from here on we will refer to them as just Firmware and Client respectively.

2.1 Firmware

The Firmware is the code that is run directly on the Crazyflie itself, it is written in C. The most important function of the Firmware is to ensure that the Crazyflie is performing the way it's told to by the Client. Shown in Figure 2.2, there are actually two dedicated microcontrollers that make up the Firmware itself.

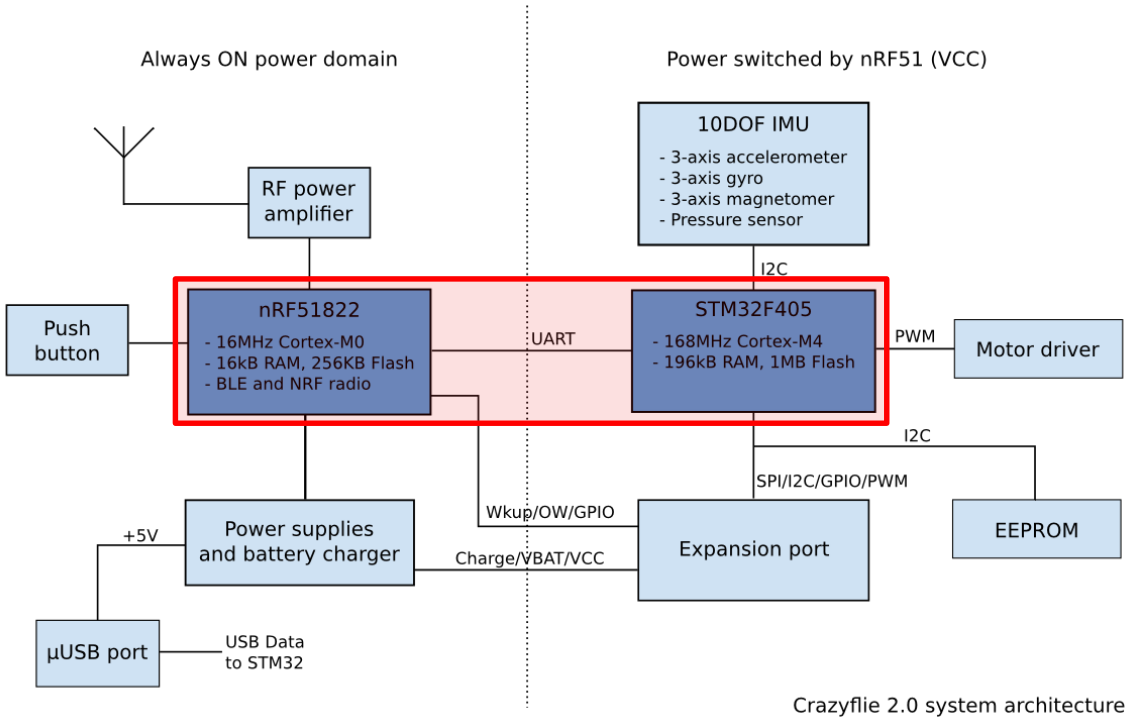

Figure 2.2 – The Crazyflie 2.0 Firmware Architecture [17]

The first half of the Firmware is an nRF51822 chip that handles all the radio communications subroutines. Things like packet composition, un-packaging received packets, power management, and send/receive functions.

The other half is the brains of the Firmware, the STM32F405 chip. This is where all the computationally intensive subroutines are executed. Here we take the commands received by the nRF chip and onboard sensor information to calculate desired control outputs. The

STM chip is also in charge of handling all the logging subroutines and peripheral detection at runtime.

2.1.1 nRF51822 (MCU Cortex-M0)

As you can see in Figure 2.2, this chip performs all of the functions that need to either be constantly running or are secondary functions of the Firmware. The chip is always on even when the Crazyflie is powered off and handles bootloading/flashing the Firmware as well as power management and battery monitoring, and most importantly maintaining the Radio Communications link. It has a direct communications line to the STM chip and communicates the raw packets received from the Radio transmitter via UART.

2.1.2 STM32F405 (MCU Cortex-M4)

On this chip we run all the controllers used for flight stabilization. The calculations are done onboard for both angular rate and angular position of pitch, and roll, as well as the angular rate for the yaw. The two types of controllers are actually updating at different frequencies. The angular rate controllers are updated at a rate of 250 Hz and the angular position controllers are updated at a rate of 100 Hz.

2.3 Client

The Client handles the user input side of things. This is where either a manual joystick controller or an autonomous control scheme would be run in order to command the Crazyflie in some way. The default Client is designed so that it can be run on linux,

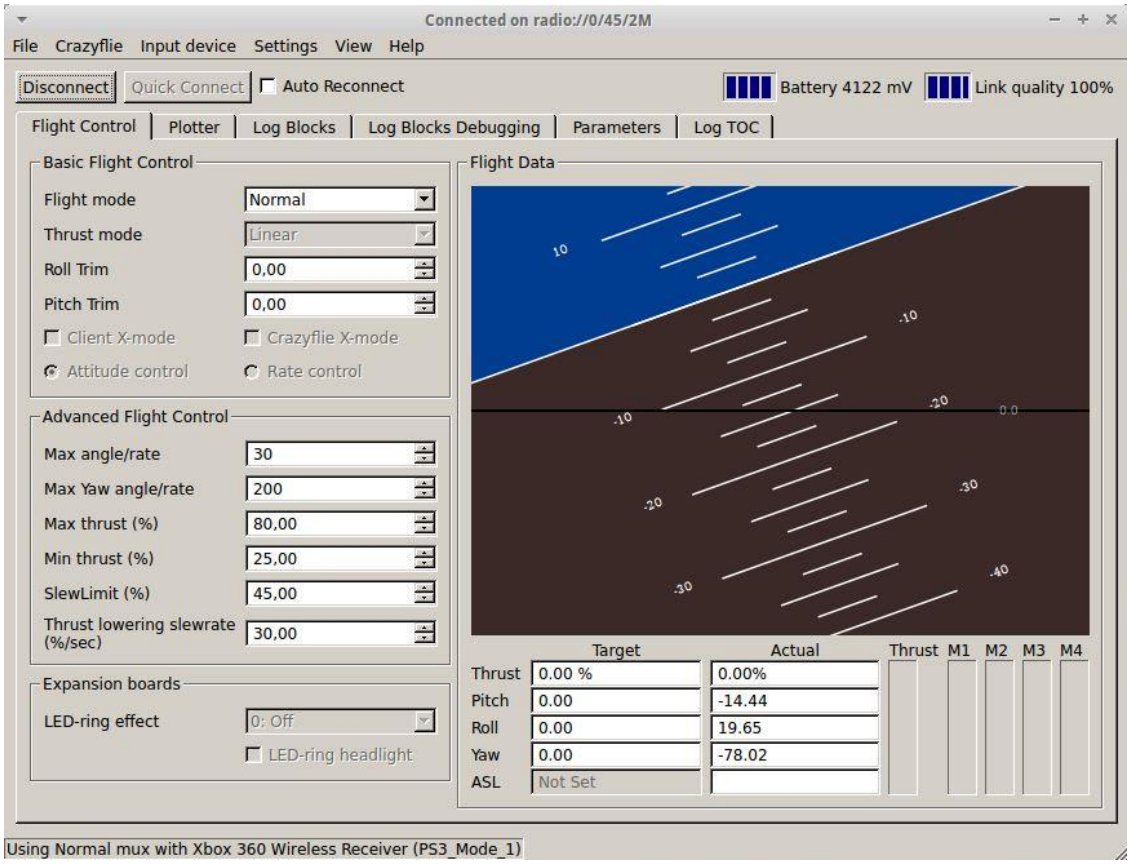

Figure 2.3 – Original Crazyflie Client GUI [10]

Windows, and Mac which means that almost anyone will be able to utilize and further develop the platform, no matter their preference.

The Client was originally built in python and had a Graphical User Interface (GUI), shown in Figure 2.3, for interacting with the Crazyflie settings and parameters. From the GUI we could edit specific parameters on the fly, we could start logging variables that would record and plot the data being received back from the Crazyflie, and we could even flash the

firmware on the Crazyflie over the air. All of this was customizable to do anything we could imagine.

For our platform we actually reverse engineered and recreated the python Client in C++. Therefore, with this new platform being both programmed in C++ and open source we can open up a whole new path for people to explore new ideas and create new applications! We will go into more depth on the Client in Chapter 4 below.

2.4 Radio Dongle

The final element required to run this system is the communications link between the Client and the Firmware. This task is performed by a Nordic Semiconductor 2.4 GHz USB radio antenna. The following are some important specifications [24]:

- Transmission Range: ~ 1 km
- Output Power: up to 20 dBm (100 mW)
- Radio Channels: 0 – 125 channels
- Data Rates: 250 Kbps, 1 Mbps, 2 Mbps
- Packet Payload: up to 32 bytes

Figure 2.4 – CrazyRadio USB Antenna [24]

From this USB radio we get a lot of versatility out of an inexpensive and low power package. We can use this radio for indoor and outdoor applications due to the range, and we can use it for flying multiple Crazyflies due to the many channels and data rates. The potential growth of this platform will only be bound by the creativity of the students.

2.5 Limitations

As with any system there are always some limitations. One of the major limitations to the Crazyflie is that the swarm framework is non-existent out of the box. It doesn't have multi-Crazyflie support on one Radio out of the box and even setting up multiple Radios per computer was something that we had to modify ourselves. In addition, once we did get multiple Crazyflies per Radio, the Radio bandwidth places a hard limit on the number of Crazyflies per Radio to 3. Another limitation was in the compatibility of the Client GUI to other Operating Systems like Windows and non-Ubuntu Linux Distributions. We could only get the GUI working using the VirtualBox provided by Bitcraze. Lastly, there is a limitation on the payload that each Crazyflie can lift. A single Crazyflie can only lift up to 15 grams, this means that we have to keep track of the weight we are adding to the Crazyflie if we attach additional sensors or trackables. As we add more weight the battery duration will decrease as well, so we need to carefully consider the sensor options available. Most of these limitations will be addressed in Chapter 4 below.

CHAPTER 3: BASIC SETUP

In this chapter we will cover some background on the 'out-of-the-box' capabilities of the external systems that make up our Crazyflie Swarm Platform. This will create some perspective on where we started off versus where we are now. Below is a brief summary of what will be covered:

1. Crazyflie Virtual Machine Client
    a. Good starting tool for learning the open-source architecture of the Crazyflie
2. OptiTrack Motion Capture Camera System
    a. How our Platform is able to track the location of the Crazyflies
3. Crazyflie Communications Packet Structure
4. *libcflie* – Crazyflie Client C++ Application Programmer Interface (API)
    a. Reverse engineered Client communications protocol in C++

Each one of these systems is an integral part of the Swarm Platform we have today.

3.1 VirtualBox

VirtualBox is another open source program that is able to emulate a separate yet fully functional Operating System (OS) on your computer. This is known as a virtual machine (VM) and in this case is extremely useful due to its portability.

For beginners, Bitcraze has constructed a fully functional Linux VM that can be used to get accustomed to the systems and design of the Client and the Firmware, all without needing to troubleshoot through all the dependencies and installations required to run it on your own.

We used the VM to figure out how the Crazyflie was handling its PID's on the Firmware. From that we were able to formulate an external control scheme on the Client, for controlling the Crazyflie's location, which was able to interface with the Firmware rate controllers. We also were able to familiarize ourselves with installing custom content and modifying the original systems.

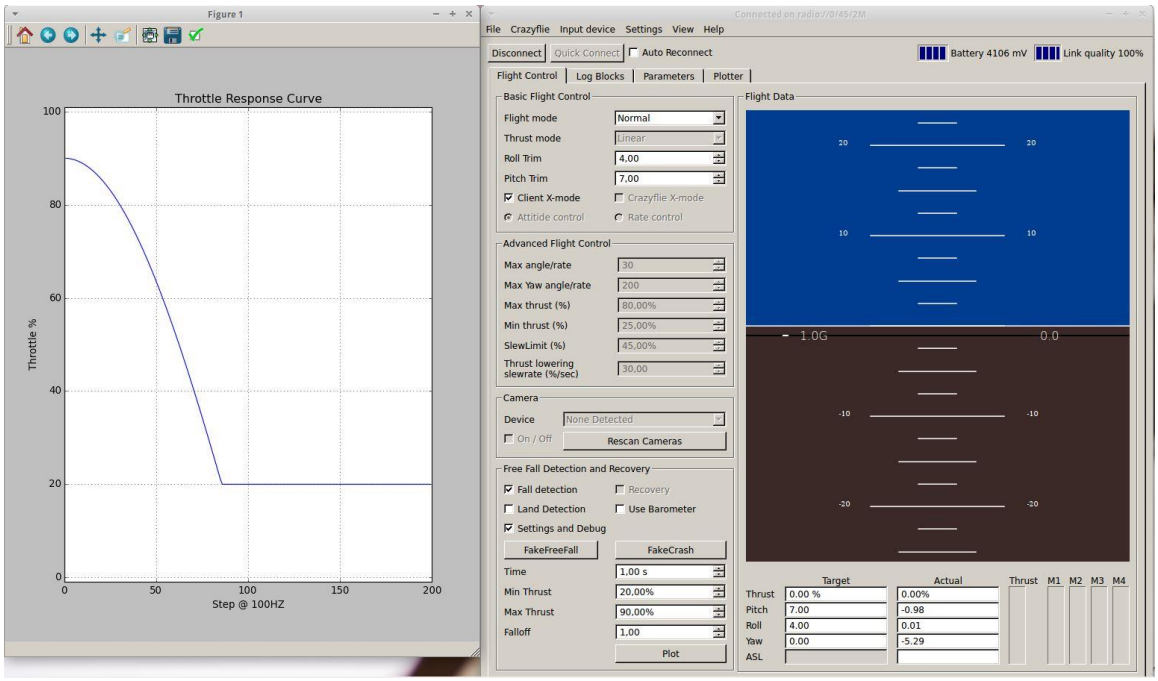

Figure 3.1 – Customized GUI with Fall Detection and Auto-Recovery Options [7]

We installed a modified Client GUI, shown in Figure 3.1, with some custom software created by Oliver Dunkley [7] that could detect when the Crazyflie was in freefall and even trigger a safety routine that would attempt to recover and land the Crazyflie. All we needed was the accelerometer data for free fall, and the barometer data for landing, both of which are easily accessible through the Client's built in logging system.

The downside we ran into with the VM was in connectivity to networks outside of the VM, this was specifically an issue with receiving the VRPN location data. Since we are running this emulated VM inside another computer there is a complex process of tunneling

information from outside the computer up into the VM. For most things like internet and USB ports the VM comes pre-configured to support those connections, but when we need to tunnel in a custom data packet from a Multi-cast UDP server like in VRPN's case it becomes a challenge to pass that data from the server to the computer and then into the VM. So much so that we decided, now that we have a good understanding of the systems involved, it was easier and more beneficial in the long run to move our set up to a dedicated Linux machine in the lab.

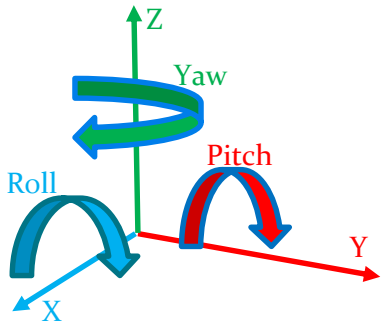

Figure 3.2 - Coordinate Axes and Rotation Around Axes

3.2 OptiTrack Motion Capture Cameras and VRPN

In order to control the position of the Crazyflie we first need to know where it is in space. This is known as Localization and without it we would not be able to accurately control the position of the Crazyflie. There are many different kinds of localization and each has its own varying level of accuracy. Examples range all the way from extremely accurate physics simulations of celestial mechanics, where we can predict the motions of planets, rockets, satellites, and stars, down to simple observations. Even just looking at an object with the naked eye can be considered a type of localization, like when flying a Crazyflie by hand!

For localization in this platform we are using the OptiTrack Motion Capture Camera system which we will refer to as just Camera System from this point forward.

3.2.1 Trackables and Constellations

We can think of the Camera System as being similar to the Global Positioning System (GPS), but on a much smaller scale that works well indoors. We have 12 cameras that are

able to emit coded pulses of infrared (IR) light, which are then reflected back to the cameras via special IR reflective spheres, called Trackables, mounted on the object we want to track. The software is then able to use the time it took the pulse to return to calculate a distance that the object is from that single camera. It does this measurement for all 12 cameras and then uses these distances to determine the X, Y, and Z position coordinates.

Figure 3.3 – Motion Capture Cameras 7, 8, and 9

The software is able to replicate this process at a rate of 100 Hz, or once every 0.01 seconds. The reason the IR light is 'coded' is so that the reflections of light from one IR camera won't trigger the sensor of another camera. This greatly improves the computational speed as we can have all 12 of these cameras are emitting light at the same time rather than taking turns. All of this is just for one Trackable.

We can also set up a group of trackables on an object and once the software is notified of the placement of the trackables it will create a Constellation out of the multiple trackables. The constellation acts similar to a rigid body, shaped by connecting the edges

between the trackable at each vertex. In Figure 3.4 we can see an example of how a couple of Crazyflie constellations look in the Camera System software.

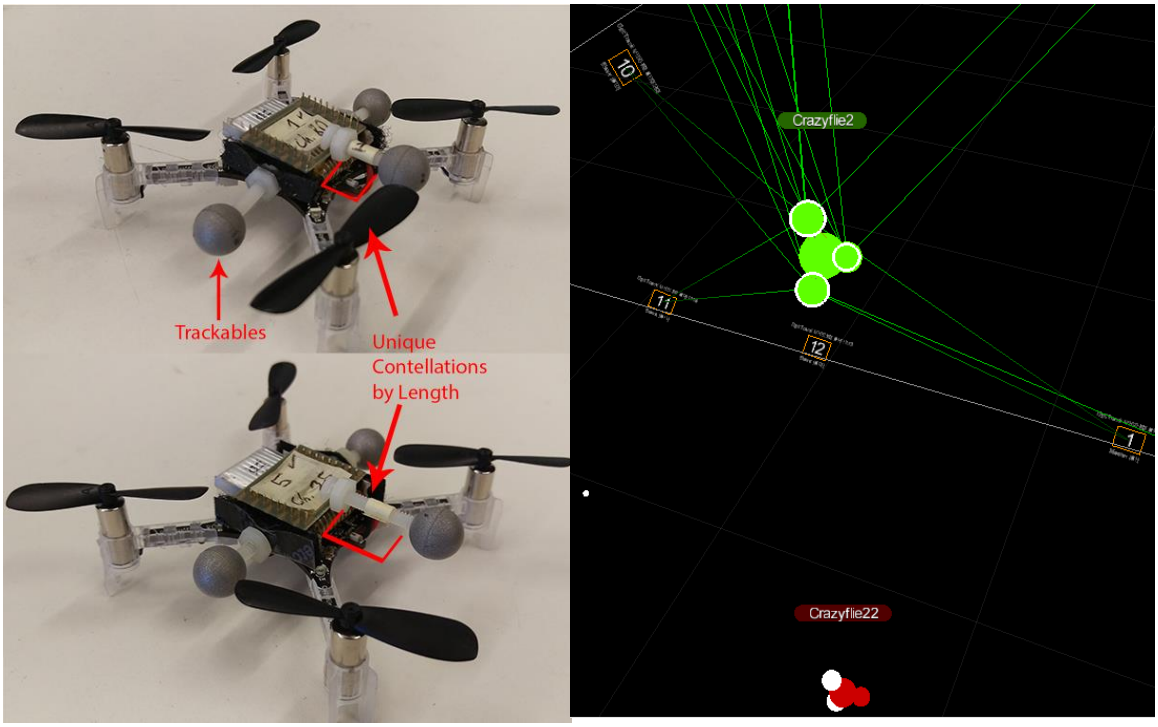

Figure 3.4 – Crazyflies with Constellation (Left), Constellation in OptiTrack Software (Right)

Now with this constellation created, during localization the position coordinates being sent are now of the constellation's center rather than the coordinates of each individual trackable. (The center of the constellation being the centroid of shape formed by the trackables).

There are many benefits that come from using the constellations rather than just the single trackable. First, the measurement error is reduced due to using the location information from multiple trackables to calculate the centroid. Secondly, the software can now find not only the location coordinates but also the orientation, or the Pitch, Roll, and Yaw of the constellation by solving for the Euler angles of the constellation. Lastly and most importantly for the swarm with unique constellation arrangements, as shown in Figure 3.4, we can determine the location and orientation of multiple objects in the same space!

The current Camera System is capable of tracking up to 7-8 trackables per constellation, and up to 7-8 unique constellations at a time. We are looking to upgrade that amount with more cameras and newer software soon.

3.3 Crazyflie Client C-Library – libcflie

The Client can be thought of like a flight control tower at an airport. It is monitoring the locations of all the Crazyflies in flight and telling them where they should be depending on what parameters the User has set up. It has very specific communications protocols and packet structuring that needs to be replicated in order to establish a connection between the Client and the Crazyflie.

3.3.1 Communications Packet Structure

The communications protocol is a packet based system with packets that are 32 bytes in length, consisting of a 1-byte header and 31-bytes of data, seen in Figure 3.5.

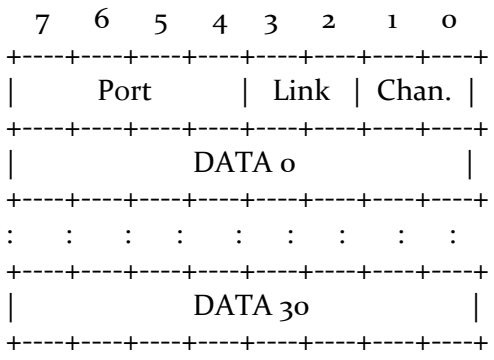

Figure 3.5 – Crazyflie Radio Packet Structure [15]

The header itself is made up of 3 parts:

- Port (4 bits): Routes packet between highest level sub-systems
    - Examples: Logging, Flight Control, Console, etc.
- Link (2 bits): Unused, reserved for later use.
- Channel (2 bits): Routes packets further within a sub-system
    - Examples: ‘logging -> log data access’ or ‘console -> print text’, etc.

These packets can become complex very quickly, for example just to read a log variable from the Crazyflie we would have to create on the Client a packet that has:

- Port = 5 (Routes to Logging sub-system)
- Channel = 1 (Inside of Logging routes to ‘Log Settings Access’)
- Data = Byte 0, Byte 1, Byte 2
  - Byte 0 is 0x03 (‘Start Block’ command)
  - Byte 1 is the desired ‘Log Block ID Number’
  - Byte 2 is Logging Period (in 10 ms intervals)

We can see that even inside the data bytes is another even lower level command structure than the Port and Channel, adding even more complexity. Now that we told the Crazyflie we want a log variable reported to us at a certain rate, the Crazyflie is going to send packets back to us as well, shown in Figure 3.6.

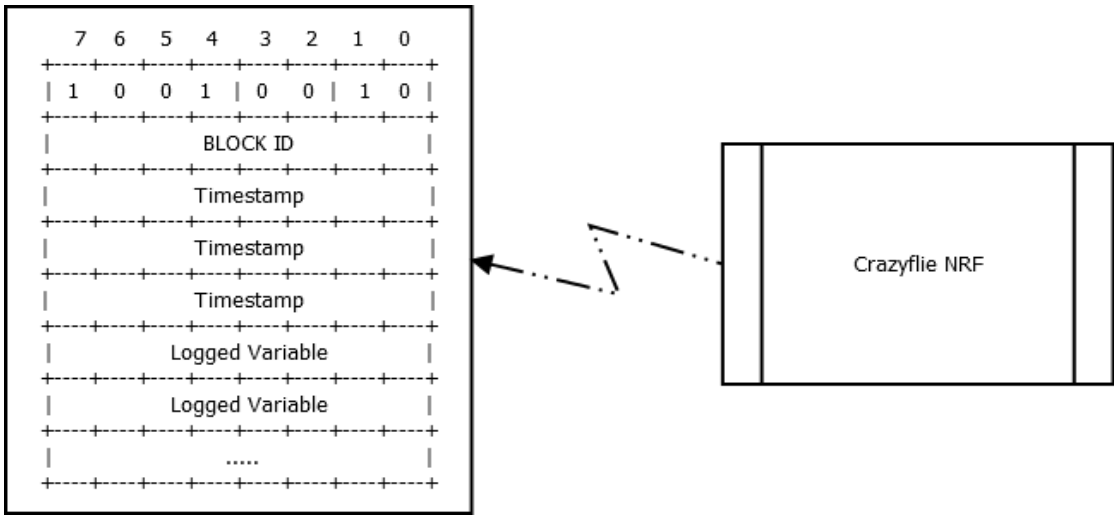

Figure 3.6 – Log Variable Packet Sent from Crazyflie Firmware to Client at Specified Frequency

Detailed information on this communications protocol as well as the command values are available online [15] for reference. So how do we get this complex protocol to function in C++?

3.3.2 Crazyflie Client C++ API

Fortunately, since this is an open source project, we found an API created by Jan Winkler called *libcflie* [8] that established these communications protocols already in C++. Jan had already dug through the low level technical requirements needed to recreate the communications protocol, packet system, and logging infrastructure and packaged them up into easy to use functions in C. Jan's work was incredibly helpful and saved us months of hard work that we would have spent debugging and troubleshooting this communications protocol.

Although all of this low level work had been established, there was still a lot to want from this library. It was missing things like exporting that log data to an external log file, and multi-Crazyflie and multi-Radio support. There was no control architecture, it was really just a simple test bed for sending and receiving messages from the Crazyflie. This is where our work begins.

CHAPTER 4: CONTRIBUTIONS

4.1 Summary of Contributions

The system we came up with was built from the ground up, composed around a couple core libraries built in C and C++. This way we would have a unified language for both Client and Firmware. A majority of the system modifications were done on the Client side, with only a handful of Firmware modifications. Below is a brief summary of our contributions:

1. Migrated Client from the Virtual Machine to RedHat Linux distribution.
    - Manual installation of dependencies and changing Client from Python to C++.
    - Unable to use the GUI due to missing dependencies that were unavailable on our distribution. Instead we use a terminal window for user interfacing.
2. Improved *libcflie* Crazyflie Client Application Programmer Interface (API) [8]
    - Keyboard Input – Allows manual variable modification during runtime.
    - Flight States – Swap out controllers mid-flight, utilizes keyboard input.
    - Data Logging – Created system to write log data to an external file, so we can use any type of data processing program to parse and visualize the data.
3. Integrate the Camera System and VRPN into the Client
    - We use the cameras to figure out the locations of each Crazyflie and then send that information to the Client.
    - Set up VRPN communications protocol in our Client to continuously receive the location and orientation of any constellations set up in its field of view from the Camera System.

4. Implemented X, Y, Z, and Yaw PID's and Tuning
5. Increased Radio Link Scalability
    - Radio can now support multiple Crazyflies per radio and multiple Radios per computer.
6. Reduced Communications Packet Delay
    - Clearing out the VRPN packet buffer
    - Shortening Crazyflie packet drop retries and wait time
7. Yaw Synchronization
    - Two independent measurement systems (Crazyflie and Camera System) need to be synced at the start of each flight.

4.2 RedHat Linux

The move to a dedicated Linux was not easy, we had to change to an entirely different Linux Distribution (distro), since the VirtualBox provided by Bitcraze was running an Ubuntu distro. We only had a RedHat distro so we needed to find and install most of the Client's dependencies manually. That being said there were some dependencies that we weren't able to get without extensive amounts of work, so we had to compromise and get the dependencies that were critical to functionality. Things like GLFW and libUSB were the most important since they made sure the Client could communicate with the Radio and the Crazyflie. The biggest downside was that we lost the ability to run the Client GUI, but that wasn't too much of an issue since this platform is more focused on developing the C and C++ side of things rather than in Python.

Once we had the critical dependencies working we ran some tests, using the terminal as the Client, to make sure we could still send and receive data via the Radio. These were just simple tests like setting the thrust output and reading back and printing accelerometer values. This was also a good way to get some introductory practice using the libcflie functions.

4.3 Crazyflie Client C-Library – libcflie

4.3.1 Keyboard Input

One of the first things we needed to modify in libcflie was to create a system that would allow us more control over how the program was executed. Early tests were just simple while loops that would perform a specific action until the program was terminated. We wanted more control, to be able to send certain commands like take off, or land, or even changing flight modes midflight.

One way to accomplish this was to develop a keystroke input check. So every time a key is pressed on the keyboard an interrupt is triggered and the key value pressed is stored. We can then use this to trigger the events we described earlier. An example of an event we could trigger is shown in Figure 4.1, the *quit* button.

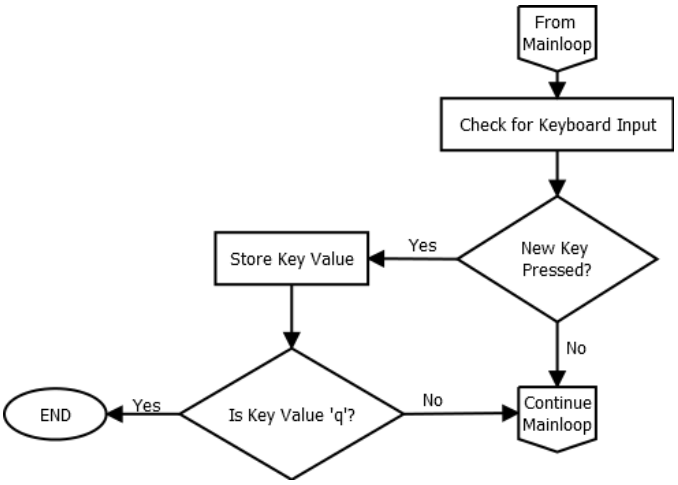

Figure 4.1 – Flowchart of the Keyboard Input Function

Specifically, this helped us solve one big problem, safely shutting off the Crazyflies at the end of a test. When you end the program manually

the Crazyflies will continue to execute whatever the last command sent was for about 2 additional seconds before realizing the connection has timed out. So when a test goes wrong or we lose a trackable, even if you were to kill the program the Crazyflie would continue to execute the last known setpoint, which can be an issue if the last setpoint sent commanded a roll of 20 degrees. Now with the keyboard input we can quickly shut off the Crazyflies by sending a 0 setpoint without having to end the program if things get out of hand.

4.3.2 Flight States

With the keyboard input in place a logical extension of it would be to create flight states that we could trigger with a single keystroke. These flight states could be anything we wanted, ranging anywhere from simple states like 'GROUNDED_MODE' where we just sent 0 setpoints to all channels, or 'TAKEOFF_MODE' where we would send a fixed position setpoint. All the way to more complex states such as 'HAND_MODE' where the setpoints varied and were entirely dependent on the location of a trackable attached to your hand. We will talk more about this mode more in the

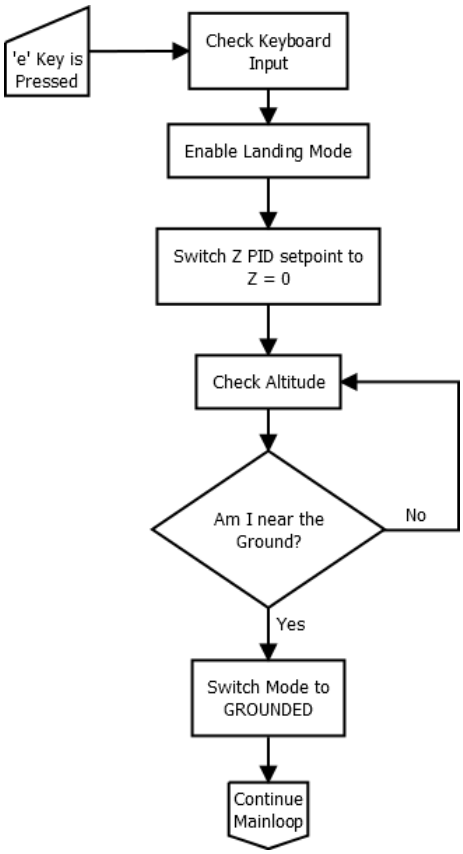

Figure 4.2 – Flowchart of Landing Mode Flight State

CHAPTER 5: A section. The flight states themselves are actually an enum structure, this way there is no way to have multiple flight states active at the same time. This protects the platform from conflicting states and race conditions that could occur.

In Figure 4.2 we show an example of how the Landing Mode flight state operates once it's been triggered. In the flowchart we can even see that this flight state will trigger a switch to another flight state when conditions are satisfied. This shows the versatility of this system, and the potential it has to design new and more complex flight states.

4.3.3 Data Logging

We also wanted to expand the logging system that had been built into libcflie. In terms of development this was the most important feature that was added to the system hands down. We used this system to more effectively tune the PID's, debug the multi-Crazyflie packet swapping, and even for reducing the packet delay! All of which we will talk about in more detail below.

The original logging system was really only a system that could read a single value back from the Crazyflie sensors, there was no storage or processing of that value.

So we created a system, shown in Figure 4.3, where at the start of each flight we would create a new text file and populate it with column headers and units for each variable we were logging. Then within the callbacks we would write the variable values into that text file into their corresponding columns. We also developed a log parser in Matlab to decipher these text files and populate the workspace with all the data we had collected. All the plots in this thesis were created from this framework.

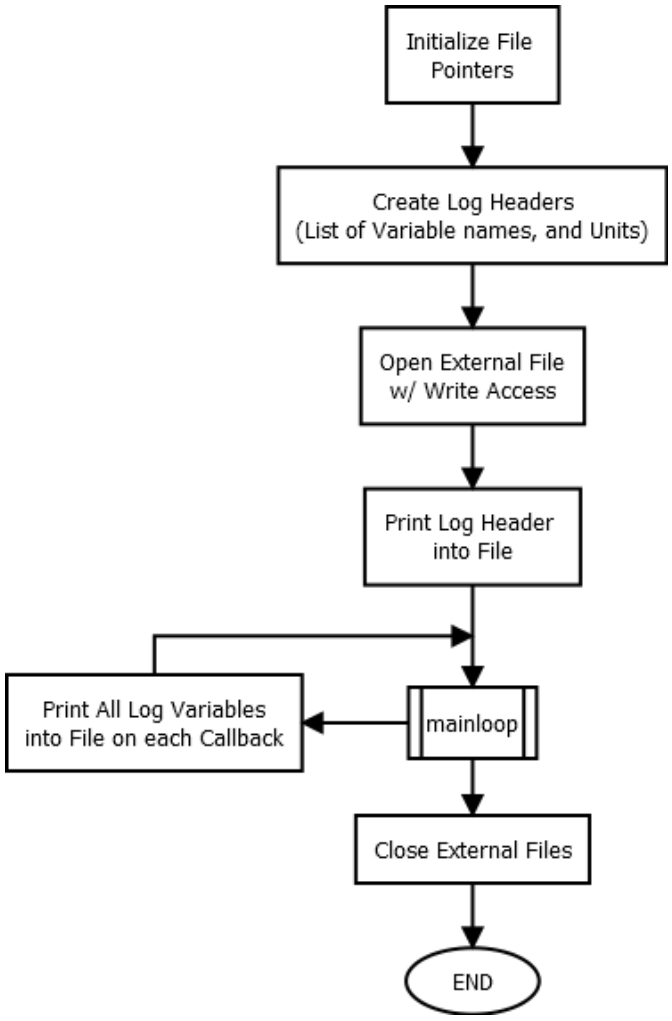

Figure 4.3 – Flowchart of the Logging Process

4.3.4 Swarm Platform Main Loop Cycle

If we take all the previous parts of section 4.2 we discussed and put them together we can see how the Platform is actually utilizing these functions during runtime. All these simple systems build on each other to create complex processes.

In Figure 4.4 we show a flow chart of the current Swarm Platform's main loop cycle. So we can see how all these individual systems are coming together and building off each other. It starts by checking if a key has been pressed, and what key value it was. We specifically look for the key value 'q' because we picked the key 'q' as a quit button to terminate the main loop and end the program. There are many other key value checks, like changing flight modes, that we have implemented but don't show due to lack of space.

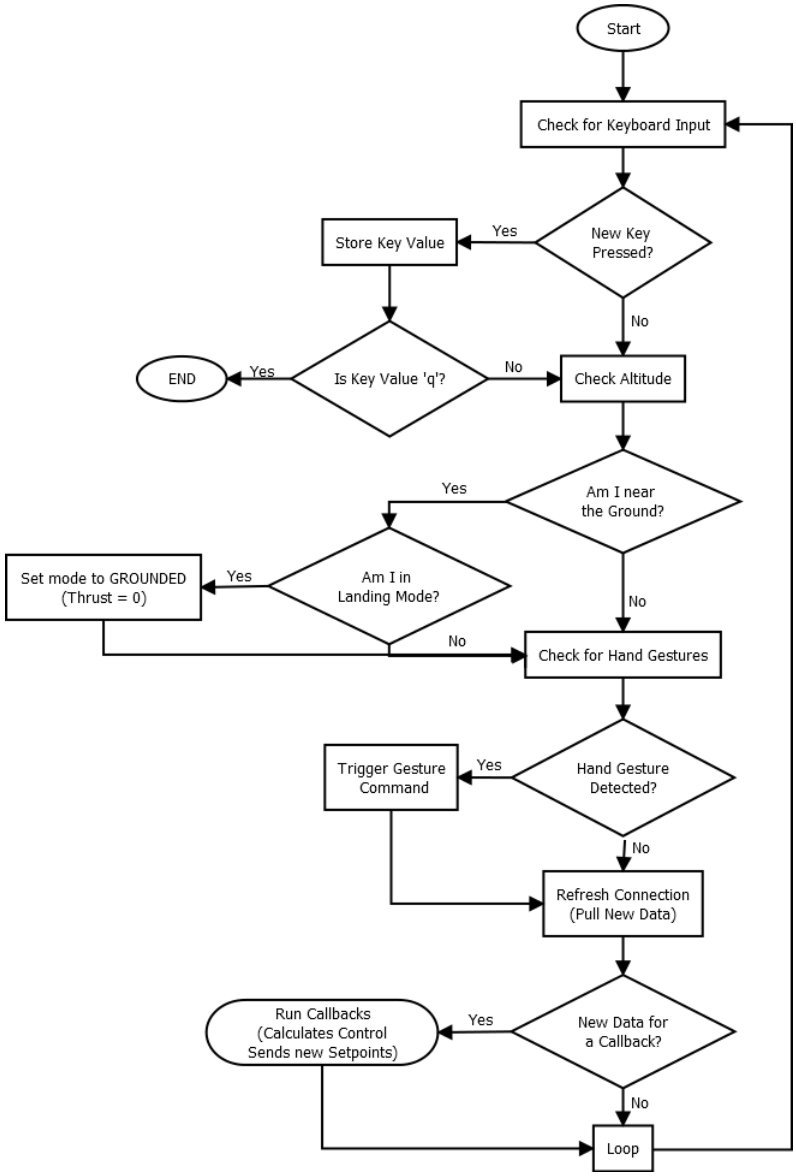

Figure 4.4 – Flowchart of the Swarm Platform main loop

Next up the loop goes into some flight mode checks to see if the system is in landing mode and needs to cut power to the motors and switch to Grounded Mode to land. It looks for hand gestures and will trigger the corresponding action if so.

Lastly, the loop will check the VRPN server for new location data. If there is no new data then we just loop and start over, but if there is new data we go straight into the callback to process the data. It's in this callback where we calculate the control setpoints, send and receive data from the Crazyflie, and write the log data to an external file.

4.4 Location PID's

For stable flight of the Crazyflie we needed to create and tune 4 new PID's (3 for X, Y, and Z position, and 1 for yaw angular position). These PID's then needed to be tuned so that the controller output would keep the Crazyflie at a desired position even with external disturbances, like packet drops or air currents, present. The process of tuning though is not an easy task, it can be extremely challenging and complicated depending on the system.

4.4.1 Creating the New PID's

Implementing the PID's was as simple as just programming the controller in C++, and placing it on the Client. We decided to keep the PID controllers on the Client because that is where we had the Crazyflie location data, and it was easier to work with than trying to add more PIDs onto the Firmware. We plan on migrating this system onto the Firmware in the future, but it requires a re-work of the communications packet code in order to send just

the location data over the Radio. So for now we have the 4 PID's running on the Client and 5 PID's running on the Firmware.

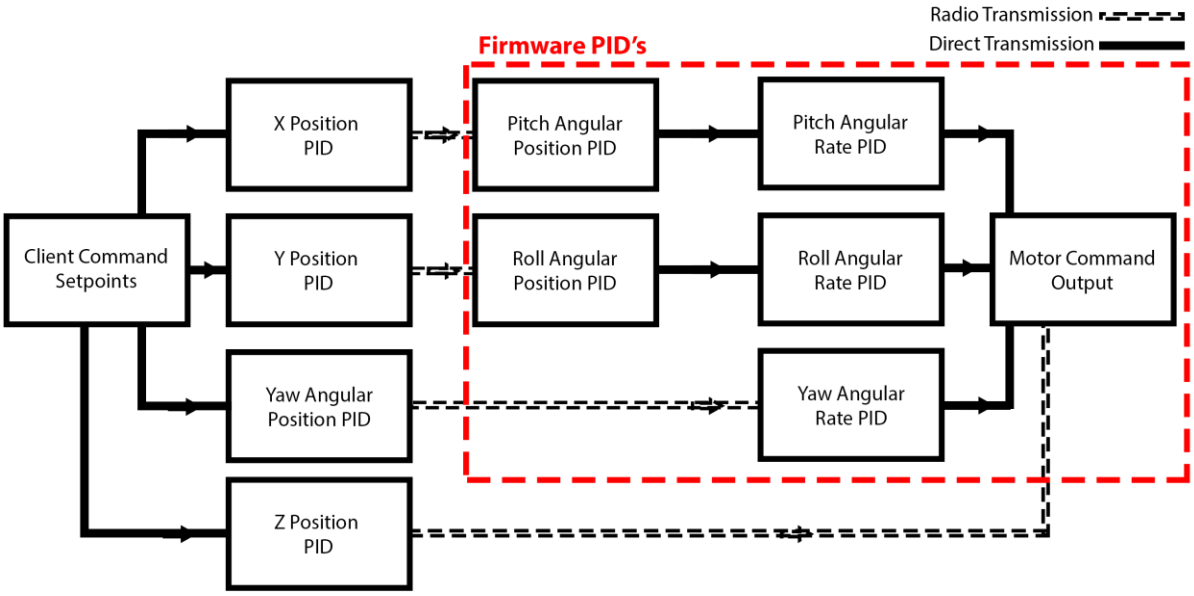

Figure 4.5 – Block Diagram of the Full PID Architecture

So now we have 9 PID's running for each Crazyflie in the swarm. In Figure 4.5 is a block diagram representing the control scheme we are using for controlling each Crazyflie. There's 3 for X, Y, Z location control, and 1 for yaw angle, located on the Client, whose output then gets sent to the Crazyflie over the radio, into 4 corresponding PID's on the Firmware for pitch, roll, yaw, and thrust respectively. This angular position output is then fed into the 3 corresponding angular rate controllers which are also run on the Firmware.

4.4.2 Tuning the PID's

The PID's we needed to tune were the X, Y, and Z position controllers that enable the Crazyflie to converge on any desired position in space. At the early stages of the platform we didn't have any logging features implemented so we needed to tune these PID's by hand. The process of tuning these controllers by hand is pretty standard, and follows a kind of educated trial and error approach. Since we know the basic properties and effects that varying the

constants will have, we can start with a guess, and then hone that guess closer to the value we want using that intuition. We usually start with tuning just the Proportional constant (k_p) with the other two constants set to zero. Starting with low value we then slowly increase this k_p until the controller produces a marginally stable oscillatory response, bordering the edge of instability, and then we reduce it a bit.

Now we have this system that is oscillating around the point we want, so we need to dampen those oscillations. This is where the Derivative constant (k_d) comes in. Again, we start out small and slowly increase the k_d constant until the oscillations we had are sufficiently damped. We need to be careful here though, if we increase the k_d term too much we will actually amplify the noise in the controller causing a lot of issues in the response. In most cases this is all we need to get a stable response from our controller, but we could still have some offset from the setpoint in the controller error. This is an issue when the controller is fighting some constant force like gravity or similar to the trim on a handheld RC controller. We have this problem in the Z controller, as we struggle to control thrust output to match the force of gravity. As you may have guessed, this issue is accounted for by the last term, the Integral constant (k_i).

In the same way as the other two constants, we start out low and gradually increase the k_i constant until that offset disappears. This is the most 'dangerous' of all the constants and requires the careful consideration on what behavior is acceptable. If we increase the k_i too much we run the risk of destabilizing the entire system, introducing oscillations that the other constants won't be able to keep under control.

All of this is just for tuning one PID controller, and we have 3 that we need to tune. So we applied this process to the X, Y, and Z PID's on our platform. Luckily in the case of

the Crazyflie we can assume our 3 location PID's are independent and, even better, that the constants we need to tune in the X and Y controllers are identical. So we can reduce the amount of tuning down to only two PID's. The Z PID, which controls thrust output and maintains the Crazyflie's height, and the X PID (that we duplicate to the Y PID) that controls the pitch (and roll) of the Crazyflie. Of the two we needed to tune the Z PID first, while just letting the Crazyflie rate controllers keep it level. This way we would at least be able get off the ground and reach a stable hover before attempting to tune the X and Y position controllers that influence the pitch and roll.

With these 3 PID's tuned we can fly a Crazyflie autonomously and have it fly to and hold its position in space. In Table 4.1 we show the PID constants we are currently using for the Swarm Platform as a result of this tuning process.

Table 4.1 – Table of Tuned PID Constants and Update Rates

PID Constants	X and Y	Z	Pitch	Roll	Yaw	Pitch Rate	Roll Rate	Yaw Rate
k_p	20	10000	10	10	10	250	250	70
k_d	22	2000	0	0	1	2.5	2.5	0
k_i	10	15000	4	4	0.35	200	500	16.7
Update Rate (Hz)	100	100	100	100	100	250	250	250

It's not the absolute best we can do, but it's quite stable for being tuned by hand. We'd like to finish up a model for the system and then we can use that to solve for some more effective constants. We will show a simple example of this PID controller in action in Section 6.2 below. The best part is that now we can use copies of these same PID's for any of the Crazyflies we add to the swarm!

The 5 PID's onboard the Firmware for angular position and angular rate come tuned out of the box so that the Crazyflie can be flown by hand without any effort, but this also means that there is room for improvement since our platform is running the flight control. Tuning the Firmware PID's is something we would like to do in the future as the other parts of the platform come together and more complex applications are developed.

4.5 Radio Link

The greatest challenge that we faced in the creation of this platform was in developing the communications protocol and radio driver software to control multiple Crazyflies simultaneously. Originally, the Client was only able to support one USB radio per computer, and only one Crazyflie per USB radio. Since this platform was built to control a swarm of Crazyflies, the end goal of this prospect was to be able to fly as many Crazyflies as we could with one single USB radio. This way we could maximize the scalability of the platform to larger numbers of Crazyflies. Yet, as with all development, it took multiple iterations and prototypes to finally reach our goal.

4.5.1 Multi-Computer Single Radio

The first step was to extend the platform to function on multiple computers. This required each computer to have its own USB radio and the flight control program to be installed on each independently. Then each computer would be dedicated to a single Crazyflie in the space, as shown in Figure 4.6.

The challenge here is to make sure that each computer is still able to receive information about the other Crazyflies even if they have no control over them. To do this we utilized a property of the VRPN communications protocol. We ran the VRPN server as a multi-cast UDP node, meaning that the location information for all constellations can be broadcast to each computer that is listening simultaneously without the need to set up a network of many point-to-point UDP tunnels.

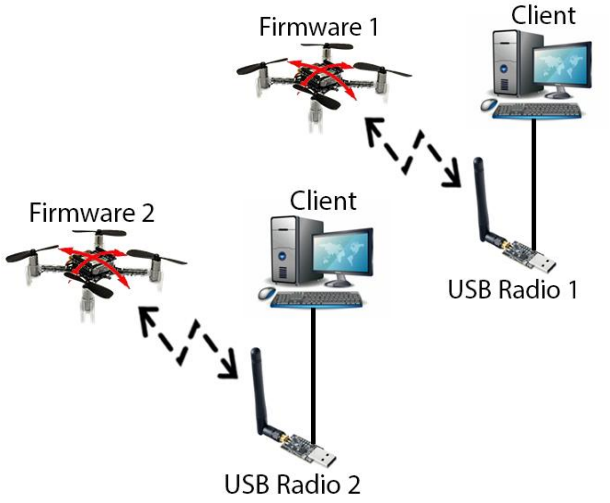

Figure 4.6 – Multiple Computer Swarm Architecture

The benefit of this is that we can essentially run carbon copies of the same flight control program on multiple computers, and have each computer only act upon one Crazyflie. All the computers are still listening to the reports from all the other Crazyflies, so they know the locations of all the other Crazyflies in the swarm, but they are only in charge of flying their own Crazyflie. The drawback to this is that for each Crazyflie we want to add to the swarm we need a new computer and USB radio to fly it. This expense for scalability is unacceptable for a swarm application, so we had to do better. What if we could attach multiple USB radios to one computer?

4.5.2 Single Computer Multi-Radio

Getting multiple USB radios to function on one computer was more difficult a problem than we thought it would be. When the flight control program is run there are many things that need to be initialized, and one of those things is the USB radio handshake. The program needs to know where to send the packets so that the Radio can transmit the commands to the Crazyflie. It does this via one of the dependencies called libUSB, which contains many helper functions, the most important ones in this case being, locating a USB's address and establishing a connection to the desired USB. Previously the Radio initialization would populate a list of valid Radios and did not care about which radio was being selected from that list, so it would always pick the first one and set that as the Radio address. So if we wanted to initialize more than one Radio this would need to change because we want each Radio to get a unique USB address.

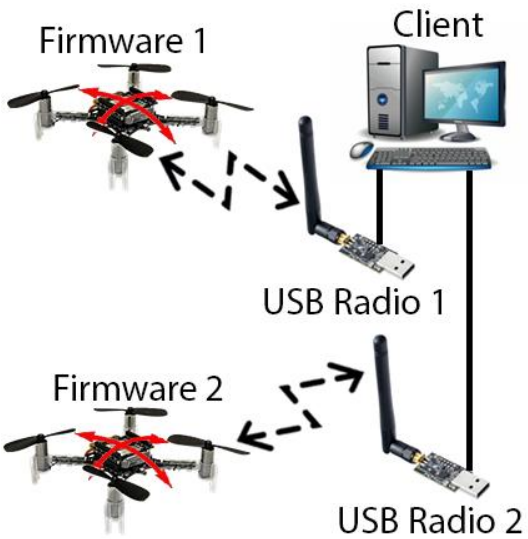

Figure 4.7 – Multi-Radio Architecture

The modified Radio initialization now runs a specified number of times corresponding to the number of radios you want to initialize, as seen in Figure 4.8, and it will keep track of how many times it has looped. Each time it runs it will populate the list of valid USB devices, specified by their product and vendor ID's, and then depending on the number of times it has looped it will access the USB at the corresponding index on the list.

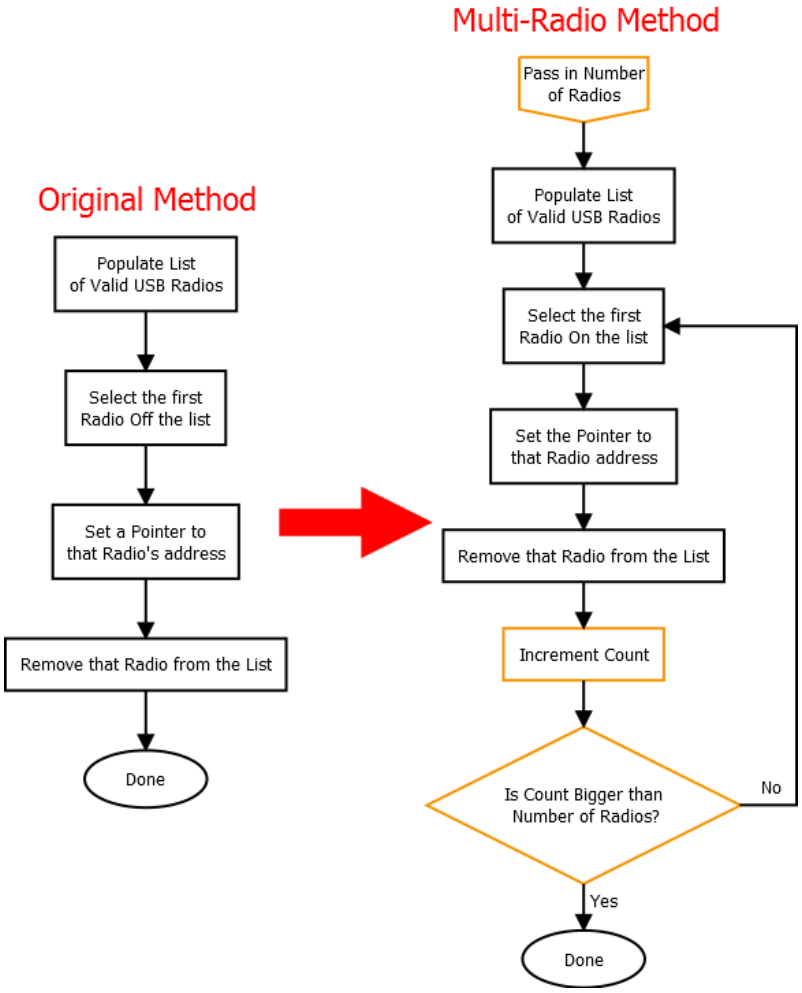

Figure 4.8 – Flowchart Comparison of Radio Initialization Methods

So for example, if we wanted to initialize two radios, it will run through once and assign Radio 1 to the first USB address in the list, and then it will loop around and then assign Radio 2 to the second USB address in the list.

With that functional we were able to send commands to two different Crazyflies from the same computer using two Radios, but when we sent those commands both Crazyflies would max out their thrust and fly in every direction! We thought it may be that the Radios were not being initialized properly, that maybe the Radios were being assigned to the Crazyflie they weren't supposed to control. This would explain why the Crazyflies are

receiving commands but the wrong commands. We tried a multitude of tests to try to diagnose the problem. The odd part was that the platform still worked when only one Crazyflie was flying at a time, and as long as you only had one in the air at a time they would achieve stable flight, but the instant you tried to fly both they would crash. So the issue must not be in the radio communications, and instead in the controller.

It turns out that the way we had set up our PID controller for the Multi-Computer case broke down when we ran two PID controllers on the same computer. When we initialized the controller it was set up to create a pointer to a specified address, but when the flight control program was modified to initialize multiple controllers with different instance names it would actually just create two different PID controllers that both pointed to the same address

Figure 4.9 – Multi-PID Initialization Before (Left) and After (Right)

space. This becomes a problem when at each time step you do a calculation for one PID and store the control error. When you calculate the next PID error, the control error from the previous PID will be overwritten. This process repeats so that neither PID functions since all three PID components function entirely based upon the error measurement. This explains precisely why the system would function perfectly fine when flying one Crazyflie at a time, but would break when trying to fly both Crazyflies simultaneously. So we added a pointer

address parameter to the initialization function, as you can see in Figure 4.9, and made sure that the PID's structs we were creating were stored at different locations. This way no matter how many PID's we called for, none of them would overlap.

With this seemingly simple change the flight control program was able to get multiple radios initialized and sending commands correctly. Now the platform is scalable to the number of USB ports we can have on a single computer. While this is an acceptable scalability it still means we need to purchase a new USB radio for each Crazyflie we want to fly, but we can still do better.

4.5.3 Single Radio Multi-Crazyflie

Figure 4.10 – Single Radio Swarm Architecture

To improve scalability even further we modified the code base to enable a single Radio to control multiple Crazyflies simultaneously. To do this we needed to implement a way to switch the Radio's channel on the fly, changing it to the corresponding value for each callback called. After solving the Multi-Radio problem this was pretty straightforward. We needed to move the stored channel value in the Radio class so that it would be associated instead with the Crazyflie class, and the same goes for the Data Rate. This way instead of each Radio having an assigned channel and data rate, we can assign each Crazyflie instance a fixed channel and data rate, and have the Radio vary its channel depending on which Crazyflie it wants to talk to. To make things easier, we created a helper function that we can use to switch the radio

channel at any time. We then tested the communications manually by just sending simple constant thrust commands before moving on to fully autonomous flight. The manual test seemed to work just fine to the naked eye so we then tested autonomous flight and that's when we found out we had a big problem.

Since we're running multiple Crazyflies on 1 Radio there is a possibility that we may get packets mixed up during transmission, but we were certain this would not be the case because the Crazyflies were on different channels. So we made the assumption that since the callbacks for each Crazyflie were separate, that when we requested data from one Crazyflie that we would always get that Crazyflie's data. The reason we thought we could do this was due to the nature of the ack system that we mentioned earlier. Which will essentially stall the Client program until it receives that ack, and then once it receives that data it then moves on to the next Crazyflie etc. As it turns out this is not always the case. After running some tests, we noticed some strange instabilities in the flight. Nothing that was completely destabilizing, but it was clearly impacting performance. In the long run it would have become a much bigger problem as we added more Crazyflies and had them performing more complex maneuvers and applications.

The autonomous flight test we ran was using 1 Radio to control 2 Crazyflies and to have them hold a desired position. Looking at the logs from this flight test we can see that a majority of the packets being received from the Crazyflies are getting swapped around. In Figure 4.11 we are plotting in red two separate plots of the yaw of each Crazyflie being tested. As you can see every few seconds the reported yaw of one of the Crazyflies will jump to the exact value of the other Crazyflie's yaw.

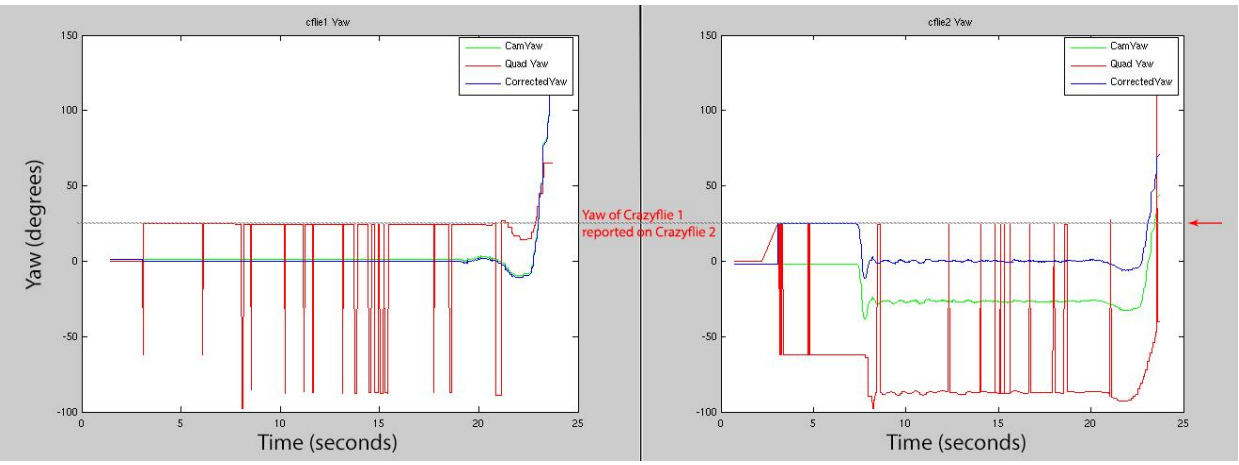

Figure 4.11 – Single Radio Packet Swapping: Yaw of Crazyflie 1 (Left) and Yaw of Crazyflie 2 (Right)

The reason it would do this is due to another buffer in the Radio itself storing data that we did not account for. Whenever an ack failed to come back in the time allotted by the ARD the program would just move on, but if that packet was then able to fully transmit after that time had expired then that packet would be buffered in the Radio for the next data request. When that next data request comes from the other Crazyflie we run into the issue of packet swapping.

The fix for this was relatively simple, all we needed to do was to process all the stored up packets and clear the buffer. Since we don't really care about any packets that are sent after a callback has ended we decided to clear the buffer at the start of every callback before we switched to the Radio channel we wanted to broadcast on.

One thing we discovered about the Radio during these tests is that the Radio bandwidth limits stable flight to no more than 3 Crazyflies per Radio. This is due partially to the physical time it takes to switch channels and transmit packets at the lowest level.

4.6 Packet Delay Reduction

During the development of the Radio interfacing, we noticed some issues with the stability when flying the Crazyflies over an extended period of time. We could see from the log files that there were delays being introduced into the system that were not being cancelled out by the controller. There were actually two problems contributing to this delay. The first originating from the VRPN communications packet buffer, and the second from packet drop between the Client and the Crazyflie.

4.6.1 VRPN Packet Buffer

There was an interesting phenomenon that we kept seeing occasionally when running tests. It was almost like the system was exhibiting some kind of non-causal behavior, shown in Figure 4.12. If we imagine the Camera System data (blue) as a depiction of 'reality' due to its direct visual observation of the constellation in motion, as

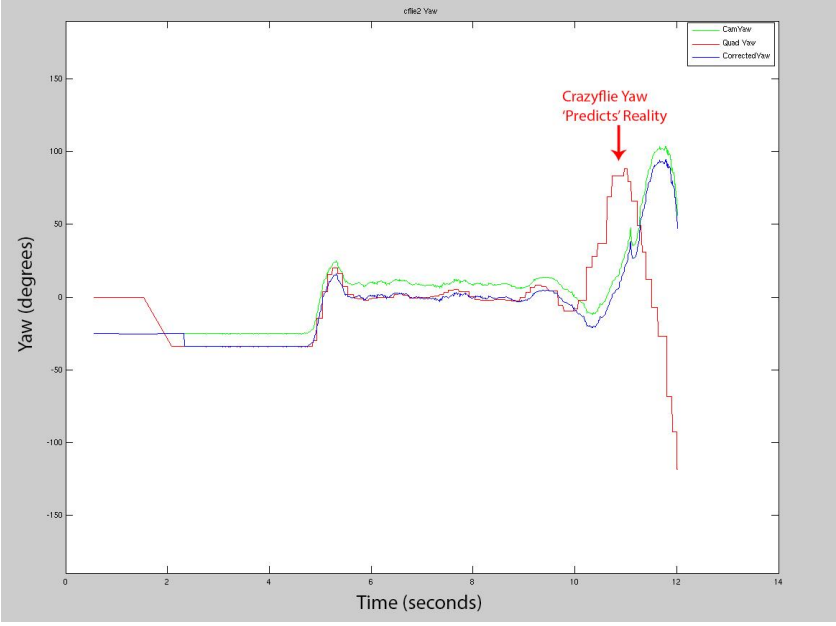

Figure 4.12 – Crazyflie Yaw Pre-empts Camera Measurement

compared to the data from the Crazyflie (red) which is more indirect being based on an accelerometer that can be skewed and accumulate error. The response we saw was almost as if the Crazyflie data was pre-empting the Camera System data, which shouldn't be possible due to the nature of the measurements. If the Crazyflie were to report movement, in let's say the yaw as in Figure 4.12 then it would have already have had to move that amount physically. This movement would have already been picked up by the Camera System before or at least at the same time as the Crazyflie measurement. This baffled us for some time, but after some brainstorming we found the root of the problem.

As we mentioned the VRPN system sends out a new packet of data every 0.01 seconds, if we aren't able to process the main loop within that period of time then VRPN will start to store those un-called packets in a buffer. The problem lies in the order which it stores those packets. As a UDP protocol it stores the oldest data at the front of the buffer, ready to be pulled off at the next call, and all the new data gets put at the back of the pile. You can think of it like waiting in a line at an amusement park. While this is good for applications where data ordering is important, for example like in video streaming, it is a huge problem for a controls application. If the loop misses that timing deadline even once, then our entire system is now permanently delayed by 0.01 seconds. Even if we only miss the deadline 1% of the time, this will still lead to an unstable flight controller after a certain period of time.

This was confirmed when we logged the time between callbacks as shown in Figure 4.13. The low valleys we see in Figure 4.13 (bottom), after implementation, are when the system has detected a packet backup and is clearing out those packets. If you look closely almost all of these valleys occurs immediately following a tall spike. This makes sense

because a tall spike indicates that the time between calling that loop again took longer than usual.

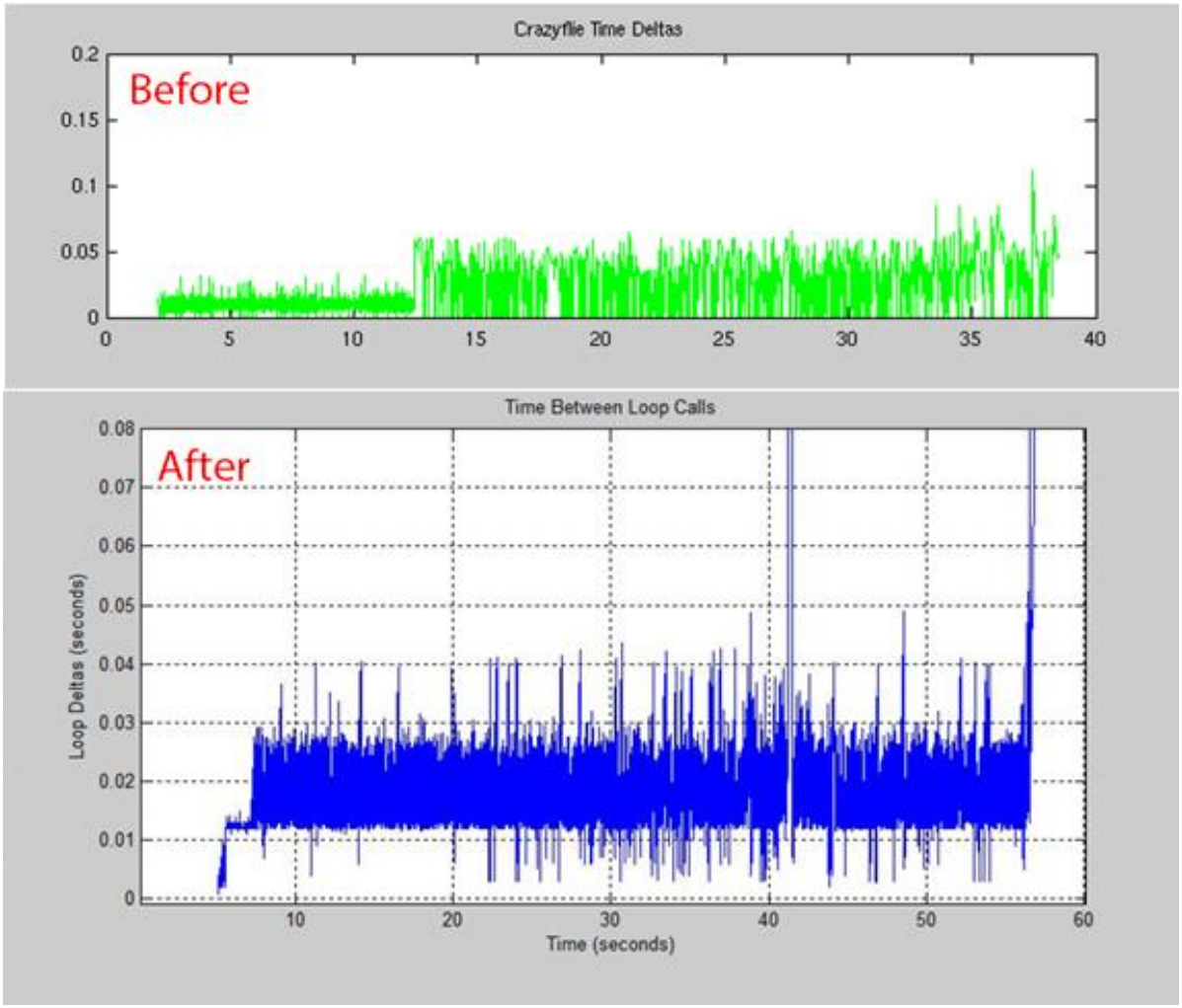

Figure 4.13 – Time Between Subsequent Loop Calls: Before VRPN Packet Buffer Clearing Implemented (Top), After Buffer Clearing Implemented (Bottom)

We also see that before we implemented the packet clearing the time between loop calls was peaking at around 0.05 seconds, and then after implementation the peaks decreased to around 0.03 seconds on average and a more consistent average loop delta becomes clear (~0.015 seconds). So with this new implementation we are able to reduce the overall delay by 0.02 seconds. While these peaks are still larger than the new data deadline, we were able to basically halve the time between subsequent loop calls and remove the permanent delay

that was being introduced by the buffer, so it's progress in the right direction. There are a few ideas we would like to implement in the future to further improve this timing discrepancy, like threading each Radio to communicate in parallel, rather than sequentially like we have now.

The benefit of controls applications though is that we don't need to act on every single piece of data that comes in, we only need the most recent data! So we can essentially throw out those old packets in the buffer and just use the newest packet, as long as we account for how much time has passed between the data packets actually acted upon. Unfortunately, VRPN does not have a built in easy way to clear the buffer so we had to craft a work around. So what we did, as the flow chart shows in Figure 4.14, whenever a callback was called we looked at how long it had been since that callback had been called last. Then if we detected the callback took longer than 0.01 seconds to repeat we would use that time to figure out how many packets were missed. Then, instead of running the usual control code in the callback, we would call the corresponding callback that many times in a row, ignoring everything inside the callback. This gives us merely an

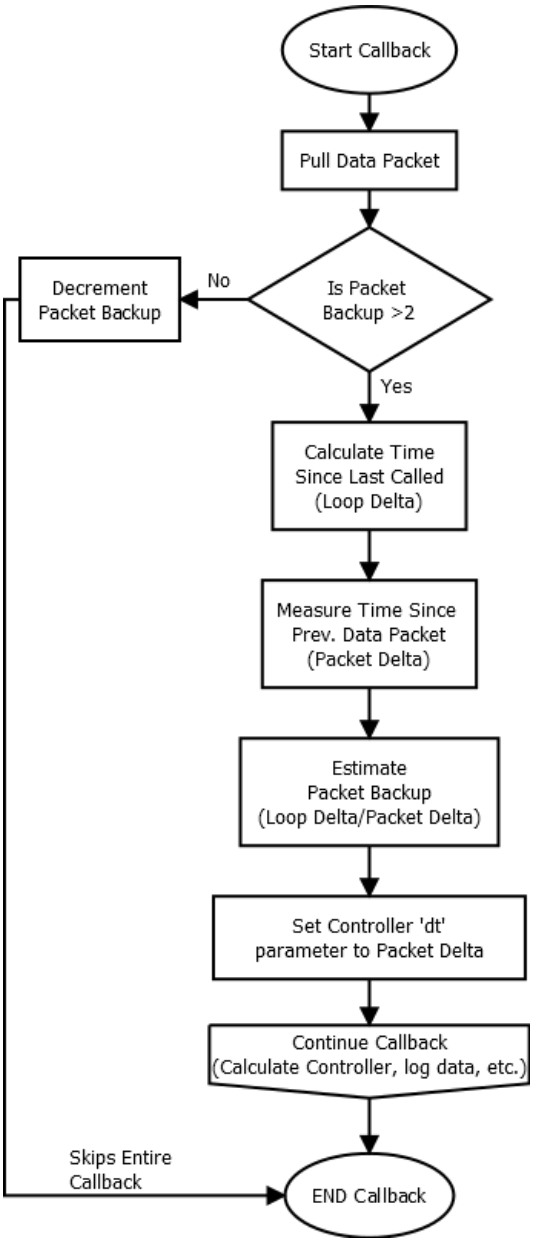

Figure 4.14 - VRPN Packet Backup Calculation

estimate of how many packets we could have missed. There's always the chance that one of the packets we missed was dropped during transmission anyways, but in that case the only downside would be that we skip the callback one extra time. This would then effectively clear out the VRPN packet buffer as quickly as it could, and get on to process the most recent packet.

4.6.2 Crazyflie Packet Drop

The communications protocol between the Client and the Crazyflie is pretty standard for packet based transmission. We can perform the usual functions at the Client like only sending commands to the Crazyflie, or only reading values from the Crazyflie, or both sending and receiving. The protocol also includes customizable acknowledge functionality. So when a packet is sent from the Client we wait for an acknowledge (ack) to come back from the Crazyflie, if there is no acknowledge received then we will try to resend the packet.

The period of time we wait for an ack, or the Auto-Retry Delay (ARD), and the number of times we attempt to resend, or the Auto-Retry Count (ARC), is fully customizable within the Client software. By default, the ARD is set to 4000 µs and the ARC is set to 3 retries. The ARD time alone is almost half of our total loop deadline of 0.01 seconds before new data arrives. This can become a huge delay factor if we drop even one packet let alone allowing 3 in a row! Keep in mind that we're still only talking about a single Callback causing this much delay. We still have other Callbacks for the other Crazyflies that we need to get to and process, all before that 0.01 second deadline. The ARC and ARD need to be reduced for the platform to have any chance at controlling a whole swarm.

We can see in Figure 4.15 how the platform behaves with the default values. The goal is for the Crazyflie reported yaw (red) to match the yaw that the Camera System sees from

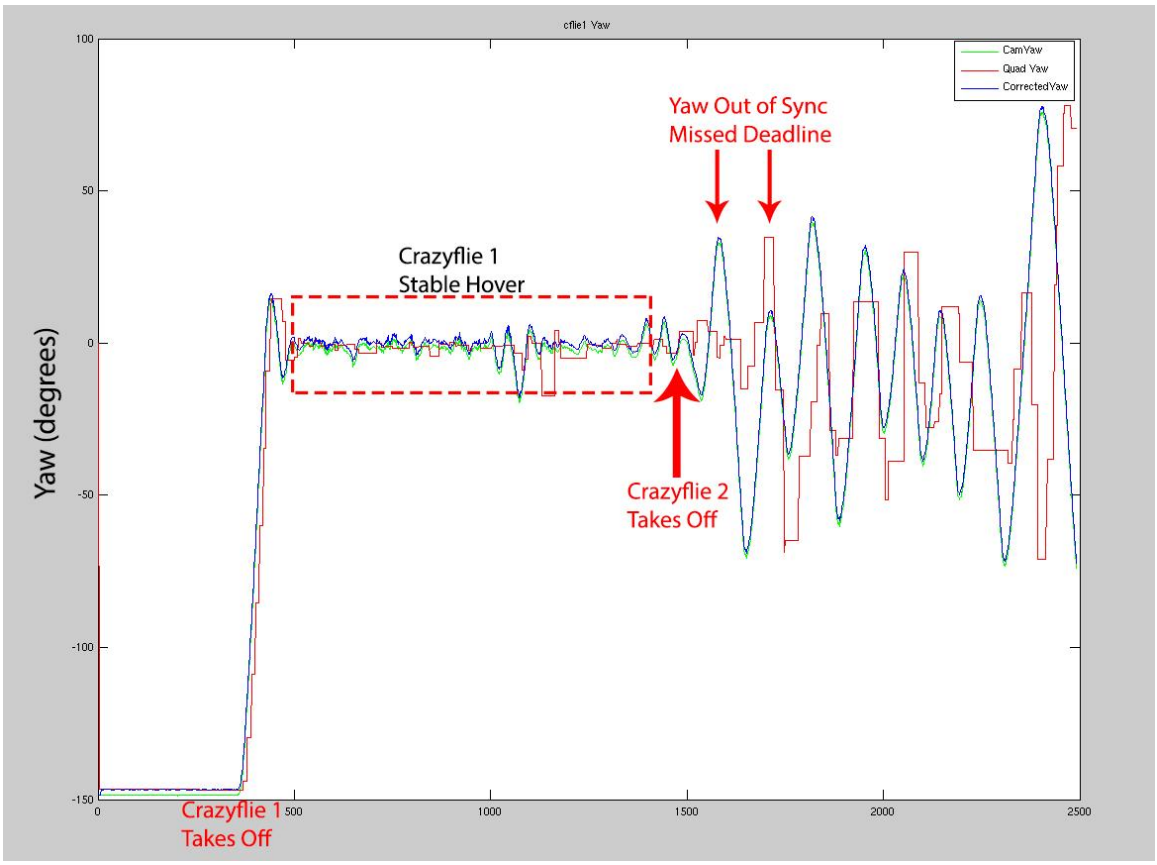

Figure 4.15 – Delayed Crazyflie Yaw with ARC = 3 and ARD = 4000 µs while flying 2 Crazyflies

the constellation (blue). We can see that when flying just one Crazyflie alone the platform maintains stable flight. This is due to the fact that even if a few packets drop we still remain within the 0.01 second deadline since this is the only Callback we have to process. We can even see that during this stable flight period there is at least one instance where a packet was dropped towards the end, but the platform was still able to recover. The real problems start when we command Crazyflie 2 to take off, so now we are controlling 2 Crazyflies simultaneously and we have 2 callbacks to contend with. The platform is no longer able to meet those deadlines with the default ARC and ARD settings, so the measurements get out of sync.

To fix this, first we'll modify the ARC, or number of retries that the platform attempts for a single packet. Normally in control applications, like we mentioned in the VRPN buffer section, we can just ignore packets if they are causing delays in the system. So in most cases we would set it up so that we wouldn't have any ack retries. The interesting thing about the Crazyflie protocol though is that if we want to read data from the Crazyflie, then the data sent back is always attached to the ack that the Crazyflie sends. Due to a quirk in the protocol configuration, we have to set the number of ack retries to at least 1 or the Crazyflie won't bother sending any ack at all, and hence we won't be able to read any data back from the Crazyflie. So we reduced the ARC from 3 down to 1.

The ARD term is more complicated to tune. Of course we want it to be as small as possible to reduce delays, but depending on the amount of Radio interference of the environment this term will vary. This means we also need to take the Radio data rate into account as well.

There are three frequencies for the data rate, 250 Kbps, 1 Mbps, and 2 Mbps. The default setting for the data rate is 250 Kbps. In our lab environment we found that using 250 Kbps was not sufficient. When using 250 Kbps we had packets dropping far too frequently, even with the new acknowledge settings. It was just taking too long to get the packets over the air before interference hit. So we opted to go with the 2 Mbps rate to minimize that air-time. This also has the added benefit of reducing delay all in its own due to the fact that we can transmit the data faster and move on to the next one.

With this new data rate, we found that a safe value for the ARD was around 2000µs to ensure a reliable data stream while significantly reducing the potential delay to the platform on a dropped packet. Tests done by Bitcraze show that the absolute best we can do

on average ranges from 360 µs to 1.26 ms latency with a packet size ranging from 1 byte to 32 bytes respectively at 2 Mbps data rate and no retries, which at worst is about a tenth of our total deadline.

4.7 Yaw Correction

For this platform we have two independent measurement systems interacting with each other. We have the Crazyflie's onboard sensors keeping track of what it thinks is its orientation, and we also have the OptiTrack motion capture camera system keeping track of its own version of the Crazyflie's orientation. In most cases these two orientations do not match up, which is inherently an issue for the platform because we need to know the orientation to be able to control its flight. The problem lies in how the two systems are initialized and how they react to changes in orientation.

When we initialize a system we are essentially defining 3 orthonormal vectors that form a coordinate basis, shown in Figure 4.16. We can then use these vectors, or axes, as a reference for any movement of the system in both lateral movement (X, Y, Z) along the axes, and rotational movement (roll, pitch, yaw) around the axes, respectively. In the case of our platform and the Camera system they both have the same 3 vectors defined, the only discrepancy between them is in the rotation around the z-axis, also known as the yaw. This is due to the fact that the X-Y plane is essentially the ground. Therefore, since we start each test on the ground, we know that that the X-Y plane is aligned in both the Camera and the Crazyflie measurements, but we don't know if the individual X and Y axes themselves are aligned. So we need to be able to synchronize the

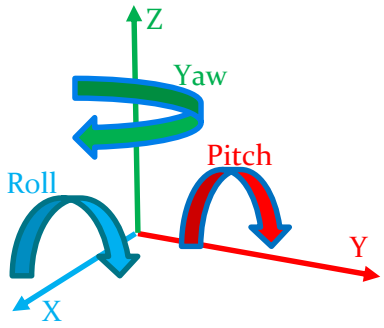

Figure 4.16 – Coordinate Axes and Rotation Around Axes

two sets of axes to match by rotating the vectors along the Z-axis (the yaw) to ensure the X and Y axes are aligned in both measurement systems.

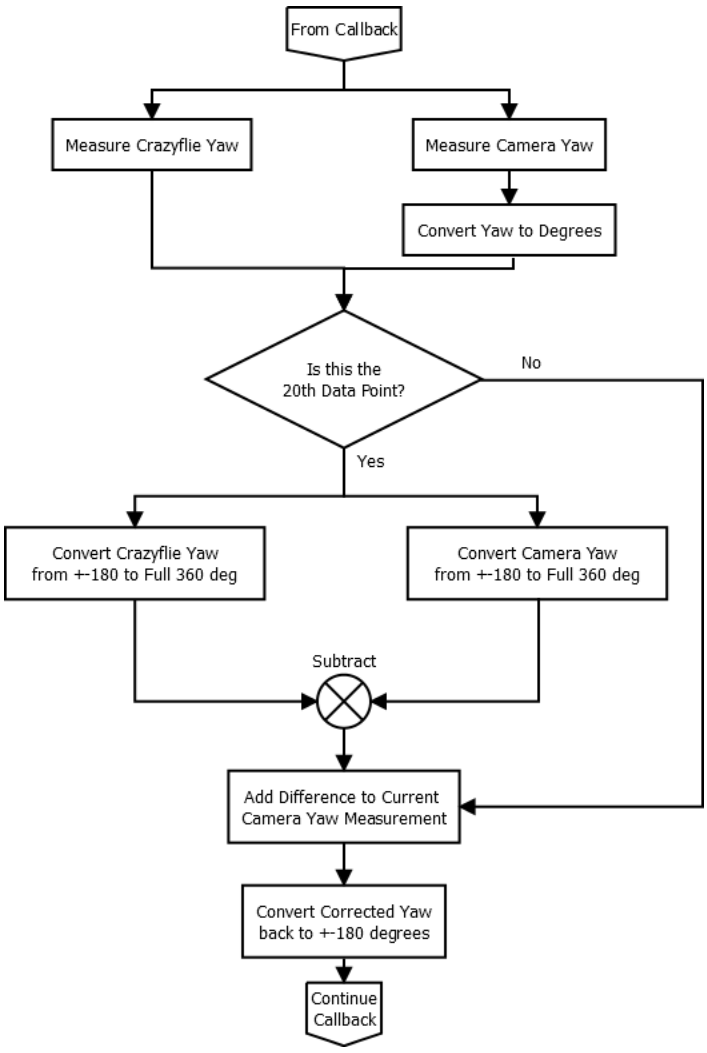

Figure 4.17 – Flowchart of Yaw Synchronization

The amount of discrepancy between the two yaw values is entirely dependent on how the Crazyflie is oriented when you power it on. So it's not as easy as just figuring out a constant that we can use over and over for each flight. What we need to do, as shown in Figure 4.17, is at the start of each flight we need to take both yaw measurements and figure out the difference between the two values. This difference is then the constant offset we can use for the duration of the flight.

In Figure 4.18, we ran a test to see the yaw synchronization process in action at the beginning of the flight. In this case at about 2 seconds in during the initialization phase, we take a measurement from both the Camera System (Green) and the Crazyflie (Red) at the same time step and figure out the difference between them. We then apply that difference to the Camera Yaw to get the 'Corrected Yaw' (Blue) synchronized value that we use as input into the controller.

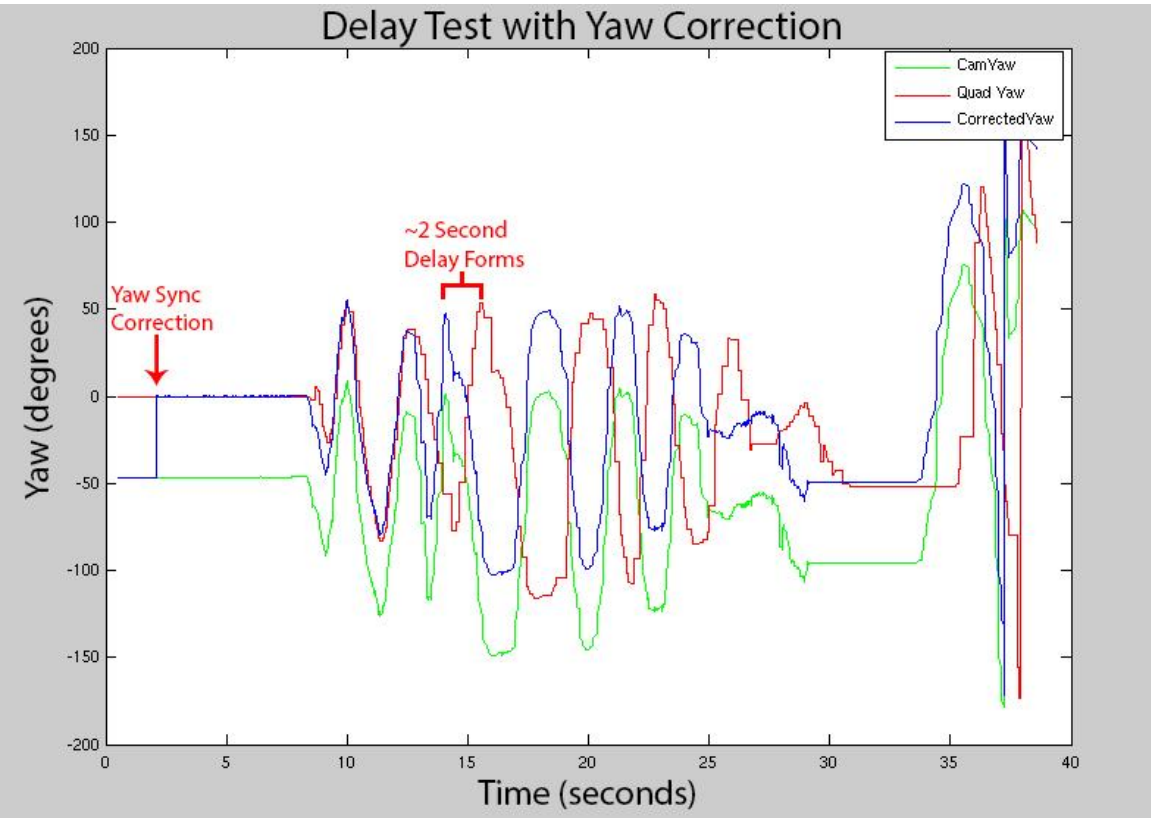

Figure 4.18 – Crazyflie Yaw Synchronization during Delay Test: Camera Yaw (green), Crazyflie Yaw (Red), Corrected Yaw (Blue).

In this case our yaw offset between the two systems was around 50 degrees, so that value was added to the Camera yaw to get the Corrected yaw. The goal here, visually, is to get the Corrected yaw (Blue) to line up with the Crazyflie yaw (Red). This means that the two measurement systems are aligned in yaw and thus aligned in all other axes as well.

This synchronization process was actually a result of our testing on reducing packet delay. So the test you see in Figure 4.18 still has some of the delay issues we discussed in the previous section, the important part of the test is to show how the yaw synchronization works at the start of a test.

A couple minor issues we ran into that are worth mentioning. The Crazyflie reports all measured angles in degrees, whereas the Camera System will report the measured angles in radians, so we made sure to take that into account when trying to sync these two systems.

The second issue is that the Crazyflie has the opposite ‘polarity’ for the yaw measurement. So a clockwise rotation would be seen as a positive change in yaw on the Camera system, and a negative change in yaw on the Crazyflie.

CHAPTER 5: APPLICATIONS

In this chapter we will cover a couple of applications that we created specifically to test out the capabilities of our Swarm Platform. A brief summary of the applications:

1. Gesture controlled swarm that has been programmed with gesture commands for take-off, fly in formation, following an object, and landing.
2. Flight control system designed on our Swarm Platform to be integrated with a single-camera computer vision tracking system implemented on a field programmable gate array (FPGA).

These applications showcase the significance of our contributions discussed in Chapter 4 above, and the versatility and potential that this Swarm Platform has for the future.

5.1 Gesture Controlled Multi-Crazyflie Swarm

In this example rather than just showing off a simple swarm formation, we have added in a constellation attached to your hand. This constellation is also tracked by the camera system, same as the Crazyflies, but we have programmed the Client to look for specific movements of that hand constellation. These gestures can trigger anything, from changing flight modes to modifying PID parameters, all in real-time.

Some simple gestures we have implemented for this example are as follows:

1. Raise hand above head (> 1.6 m) to command Crazyflies to takeoff and hover above their starting point.
2. Lower hand below knees to command Crazyflies to land (< 0.4 m).

3. Tilt hand 90° towards you to change flight mode to ‘Follow-the-Leader’ (the swarm formation will now follow the position of your hand)
4. Tilt hand 90° to the right or left to trigger Emergency Stop (Crazyflie thrust command will be forced to 0)

To show off a demonstration of this in action we use 2 Radios, 4 Crazyflies, 36 PID’s, and a custom constellation attached to our hand. In this case the ‘Follow-the-Leader’ mode modifies the position setpoints of all the Crazyflies into a function of another object’s position, in this case they follow our hand. We can see the results of the flight in Figure 5.1.

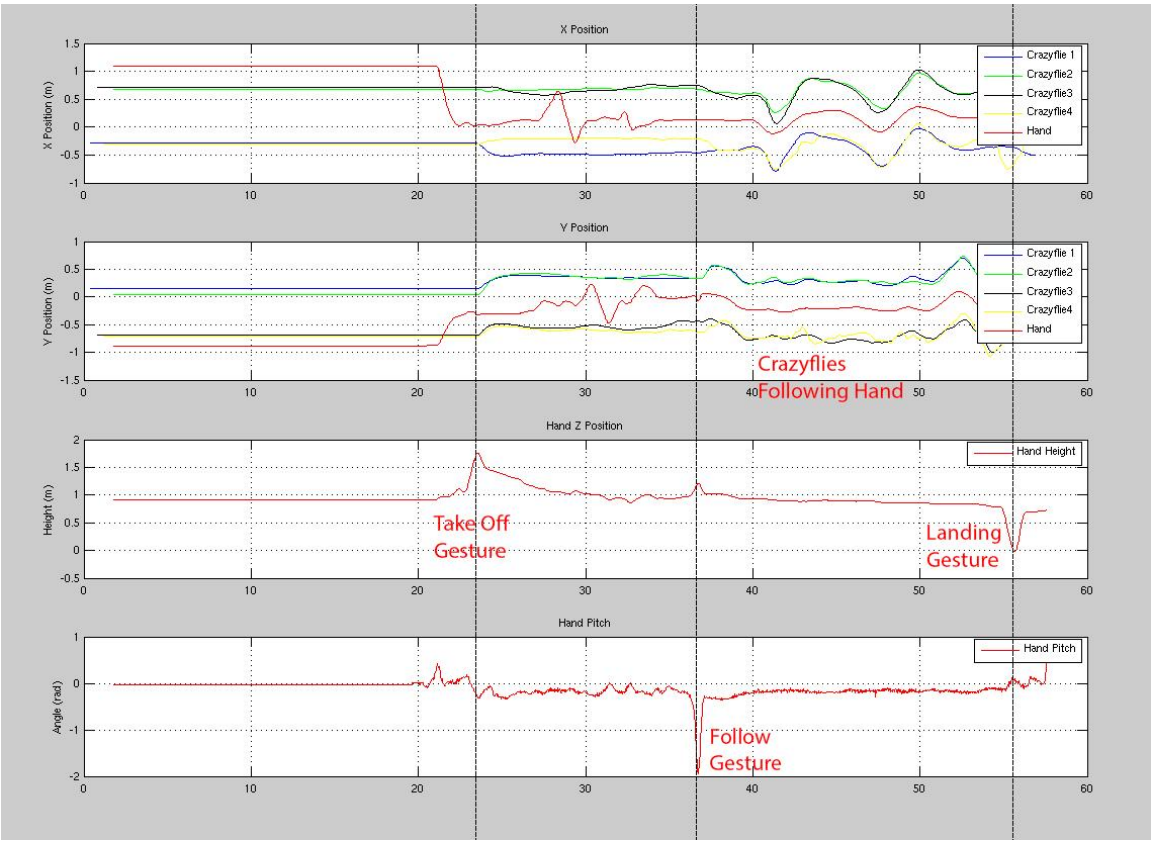

Figure 5.1 – Flight Test of Gesture Based Swarm Formation
X-Position of Crazyflies and Hand (Top), Y-Positions of Crazyflies and Hand (Top-Middle), Hand Height (Bottom-Middle), Hand Pitch Angle (Bottom)

The top two subplots show the X and Y positions respectively of each quad and our hand (red). The bottom two subplots show the height of our hand and the pitch angle of our hand respectively to show when gesture triggers are detected.

We start the test and around 22 seconds in we raise our hand above and the Crazyflies take off and hold their position. Between 22 and 36 seconds we can see that the hand trackable is moving but it has no effect on the Crazyflies, they are still holding position. At 36 seconds we rotate our hand towards us and that triggers the 'Follow-the-leader' gesture. Now the Crazyflie formation is following the location of our hand and continues to do so until the landing gesture is detected at 56 seconds and the test ends. Fortunately, we did not need the Emergency Stop gesture during this test.

Analyzing Figure 5.1 we can see that the swarm tracking is very good, with an overshoot of only about 0.05 meters while tracking our hand. This means that our PID controllers are well tuned and can handle rapid changes in their setpoints while still remaining stable.

Another way we can analyze the performance of a test is in the computational timing of the platform. As we start adding more Crazyflies to the swarm we also add more computation time to the Client main loop. We need to make sure we are staying within a reasonably bounded loop time so that we can keep up with the 100 Hz data update rate, otherwise the platform won't be able to correct for errors quick enough and the Crazyflies will crash. So we need to look at the performance of the system, specifically the loop timing. How long did it take to calculate the PID's, transmit the commands, and log the data for each Crazyflie? How much time passed between subsequent loop calls?

In Figure 5.2 we look at the timing log for the same test. We plot the height of each Crazyflie (top), for reference on when the test started, ended, and when the Crazyflies were actually flying versus sitting on the ground. We also recorded the time it takes for each individual Crazyflie loop to finish (middle), as well as the time between subsequent calls of the same loop (bottom), also known as the loop delta.

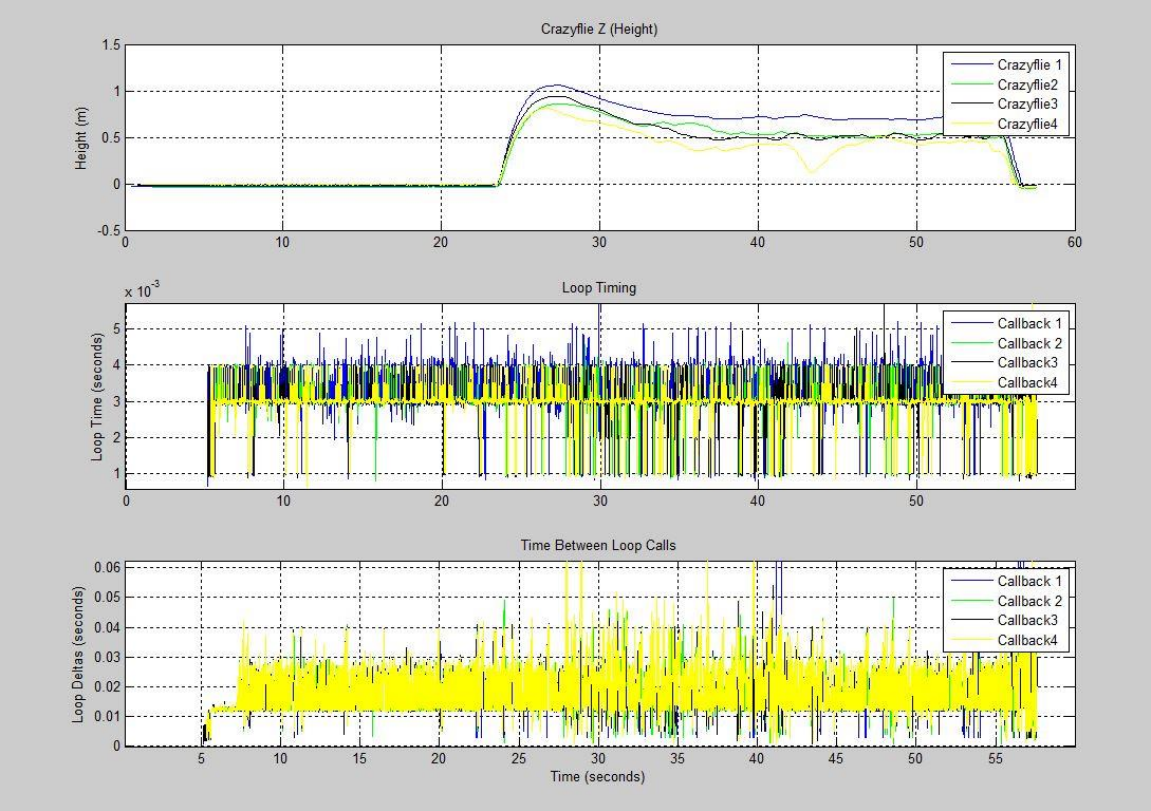

Figure 5.2 – Timing Diagram for Gesture Based Swarm Formation Test
Crazyflie Z-Position (Top), Single Crazyflie Loop Time (Middle), Time Between Subsequent Loop Calls (Bottom)

First we'll focus on the callback (loop) duration (middle), this represents the amount of time it takes just to calculate the controller outputs, send them to the Crazyflie via the Radio, and log data for one single callback. We can see that it is pretty consistent, staying around 3-4 ms per callback. The valleys you see that are recorded below 1 ms are actually the result of the platform clearing out the VRPN buffer. As you may have noticed these lower

valleys start occurring more frequently once the Crazyflies are up in the air, and usually follow immediately after a very tall spike. Which makes sense because there is more to calculate, record, and send over the Radio when we're trying to stabilize the system in flight compared to resting on the ground.

One issue we are currently working on is reducing this duration. As you can see, with 4 callbacks running at about 3 ms per callback (on average) we actually have a total loop time of 12 ms with just the callbacks alone, which is exactly what we see in the loop deltas time (middle). If we recall the time it takes the camera system to create new data is 10 ms on average. This is not ideal, as we need to down-sample on every other packet, but it is better than risking de-stabilizing the system. With the variable down-sampling we are still able to maintain smooth flight, but this is not a sufficient solution, as we scale the system up to include greater numbers that timing is going to increase as well and eventually the down-sampling won't be enough to keep the system stable.

If we look at the time between callbacks being called, we can see that the platform is still able to maintain stable flight even though we could miss the deadline by a factor of 3. This is mostly an issue due to how the callbacks are called and the priority in which they are ordered. Currently the system will run through the callbacks sequentially, checking if there is new data to process. If there is new data, then it will interrupt and jump into that callback and execute the code. This means that the callback at the bottom of the list tends to get neglected in terms of time. The effect this has, which we can see in Figure 5.3, is that the first Crazyflie will fly perfectly due to the fact that he's at the top of the list on each scan. The fourth Crazyflie will not have the same luxury and must wait his turn every time before getting a chance to send setpoints over the radio. So if the other callbacks take too long it will impact

Crazyflie 4's performance but have no effect on the other Crazyflies. That's why we see the Crazyflie 4 (yellow) dominating the time between callbacks. We are currently working on multiple methods of reducing the amount of time it takes to process each Crazyflie, like threading each Radio to run in parallel, and moving the PID calculations from the Client to the Firmware.

This example is a good demonstration of the cumulative progress we have made in developing this platform. It uses multiple Radios per computer, multiple Crazyflies per Radio, it shows the flight mode system, gesture system, and even data logging output. It uses a majority of the new systems we put in place as well as showcases the platform's ability to control multiple Crazyflies at the same time, even under heavy timing loads.

5.2 Controlled Flight: Single-Camera Computer Vision Tracking

In this example we took our Swarm Platform and integrated it with a hardware-accelerated computer vision tracking program developed by a fellow researcher, John Haughery [19]. This computer vision tracker was implemented on a Xylinx Field-Programmable Gate Array (FPGA) and used a single camera to track objects of a certain color. The idea is similar to the camera system we use in the lab but for a much lower cost, albeit at a lower accuracy when compared to the camera system. The goal was to see how much worse the accuracy was, if we could use this system to get somewhat similar results, it would drastically lower the entry level costs of distributing this platform at other universities and even high schools!

It could process images at a rate just over 60 frames per second [21] which means that its update rate was about 60 Hz as compared to the camera system's 100 Hz. We also attached IR LED sources to reflect light back into the camera lens using the same IR

reflective material as the camera system. For a proof of concept, we ran this test with 1 Radio, 1 Crazyflie with 9 PID's, and the Xylinx FPGA with a camera peripheral.

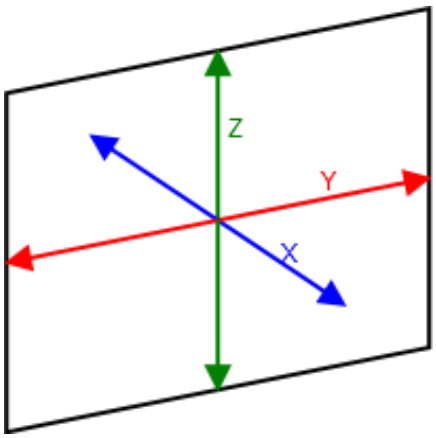

Figure 5.3 – Coordinate Directions for Computer Vision Tracking System (The box represents the 2-D plane in which we do not need to estimate position)

Integrating the platform with this new computer vision system required re-tuning of all the X, Y, and Z location PID's and implementing a new UART serial communications link between the FPGA and the Client so that we could get the location data from the FPGA. The system had only one camera which limits its viewpoint to essentially 2-Dimensions, so we were actually missing a degree of freedom when trying to control the Crazyflie. This means that while we could accurately track the Crazyflie's position in both Y and Z, we needed to develop a method of estimating the Crazyflie's 'depth' (X-axis), shown in Figure 5.3. To do this we calculated our estimate based on the size of the object (in number of pixels) we were tracking. This gave us a very rough estimate of its distance from the camera, and we were able to achieve somewhat stable flight. Of course this is nowhere near the best way to estimate the depth, but we wanted to show a proof of concept of this platforms flexibility and potential for any kind of project a student could imagine.

The Crazyflie logging had not been implemented yet though at the time of testing, so we do not have any plots to analyze performance directly. From the videos [22] [23] we do have of the tests we can approximate the performance of the hybrid system.

In both Y and Z directions we estimate that the controller was able to maintain a setpoint to about ±10% error, but as expected there was significant error and drifting in the

depth estimate, that was only able to maintain its setpoint to about ±40% error. Part of this error comes down to the re-tuning of the PID's as well as the slower update rate of the computer vision tracker, but the results are promising. If we were to further tune the PID's and extend the system to 2 cameras and use a stereostropic algorithm we could produce a much more accurate depth estimate in the same way that our human depth perception works.

This example showcases how the platform can be easily integrated with other systems and how it can be used to provide interesting hands-on experiences for students to learn and develop their skills in controls and programming. With some design refinement and using a more complex algorithm we could get some impressive results

CHAPTER 6: PROPOSED EDUCATIONAL USES

In this chapter we will talk about some examples of educational resources and exercises that we have been developing using our platform. These teaching materials have not yet been tested in a classroom setting so we will not be able to draw any conclusions to how effective these methods are, but I think that giving students more opportunities to get hands-on with the concepts they learn in the classroom will improve their interest in and retention of the material.

6.1 Wiki

The primary way that students will learn about the platform will be via a Wiki that we have built specifically for this platform. We decided to create a wiki because a wiki can evolve right alongside the platform development. As more people work on and develop the platform, we can document their progress and troubles, and when they inevitably move on their knowledge can stay behind and help guide the next generation. Even if a student doesn't want to develop the platform further, they can still access those resources and learn all the intricacies of the systems. If they run into problems, it's possible that someone else has already solved them, so this wiki will keep track of all those solutions for the future students to come.

To give you an idea of the type of content you'll find on the wiki [18], we have provided a few examples, in Figure 6.2 and Figure 6.1, of pages we have already created for this platform. This wiki will continue to grow and update as the platform is developed further in the future.

Radio Initialization

The following steps are how to Initialize a Radio. This code is currently run at the start of `main` in eris_vrpn.cpp. The process below can be extended to **any number of Radios** (currently 2 Radios shown)

1. Create CrazyRadio Pointers and Crazyflie Pointers

```
CCrazyRadio *crRadio1 = 0;
CCrazyRadio *crRadio2 = 0;

CCrazyflie *cflieCopter1 = 0;
CCrazyflie *cflieCopter2 = 0;
CCrazyflie *cflieCopter3 = 0;
CCrazyflie *cflieCopter4 = 0;
```

Note: You must have a pointer for EACH Radio you want to initialize.

2. Initialize First CrazyRadio

```
crRadio1 = new CCrazyRadio(0);   // dongle number (starting at 0)
if( crRadio1->startRadio() )   //assigns pointer to USB location
{
```

Figure 6.2 – Partial Wiki Page: Steps to Initialize Radio [18]

Now the initialization of the Controllers is complete.

Calculating the PID's

1. Each loop we want to calculate the PID output (code in *pid.c*):

```
float pidUpdateAngle(PidObject* pid, const float measured, const bool updateError) //**USED FOR ATT
{
        float output;

        if(updateError)
        {
                pid->error = pid->desired - measured;  //Calculates current error
        }

        pid->integ += pid->error * pid->dt;            //Calculates integral term
        if(pid->integ > pid->iLimit)                   //Checks that the integral term stays below
        {
                pid->integ = pid->iLimit;
        }
        else if(pid->integ < pid->iLimitLow)
        {
                pid->integ = pid->iLimitLow;
        }

        pid->deriv = (pid->error - pid->prevError) / pid->dt;  //Calculates derivative term

        pid->outP = pid->kp* pid->error;
        pid->outI = pid->ki * pid->integ;
        pid->outD = pid->kd * pid->deriv;

        output = pid->outP + pid->outI + pid->outD;  //Adds up all 3 calculated terms for output

        if(output > pid->angularLimit)               //Limits amount Crazyflie can tip
        {
                output = pid->angularLimit;
        }

        if(output < pid->angularLimitLow)            //Limits amount Crazyflie can tip
        {
                output = pid->angularLimitLow;
        }

        pid->prevError = pid->error;                 //Stores the current error for next loop
```

Figure 6.1 – Partial Wiki Page Example: Steps for Calculating the PID output [18]

6.2 PID Educational Exercise

For this exercise we already teach the students PID controllers in the classroom, but we generally never apply these controllers to anything. Of all the types of controllers the PID is one of the most widely applied controllers in industry due to its 'one-size-fits-all' nature in Single-Input Single-Output (SISO) systems. In most cases you just need to plug in the system input, set a reference setpoint, and then tune the controller until you get the desired output. It doesn't matter what units you're using. It doesn't matter if its pressure, or temperature, or speed. As long as you can tune a PID controller correctly, it will give you the desired response.

Our platform will finally give the students a chance to see how this PID theory works in practice, as well as teach them some basic Simulink modeling and model verification skills.

Learning Objectives:

1. What are PID's, how do they work
2. Physical response of each constant (kp, ki, kd) and how changing them effects the controller.
3. How to go about tuning a system by hand, applying the theoretical ideas learned in class to reality
4. How to build a model in Simulink
5. Identifying Model Parameters
6. The value of a well Modeled system

Teaching Content:

First off we need to teach the students how a PID works, and how the 3 individual components that make up the PID effect the behavior of the controller. I won't go into all of the details now, but the basic overview is:

- k_p is the proportional constant, it acts as a multiplier directly on the current error (between the actual measurement and the desired setpoint) and produces an output to correct that error. The drawback with this term alone is that there is no way to slow down until after you pass the setpoint.
- k_d is the derivative constant, it acts on the change in error measurements (difference between previous error and current error) and produces an output that acts as a kind of 'damper' in the system. The benefit of this term is that it can slow down the response of the controller so that we don't fly past the setpoint.
- k_i is the integral constant, it acts on the amount of error accumulated over time. The benefit of this term is that it can correct a constant steady state error, such as when the system is fighting a constant force (i.e. a Crazyflie fighting gravity). The drawback is that this term can create more instability if not tuned correctly.

Lab Work:

Once these concepts are sufficiently understood the students start working in the lab. In the lab the students will be given a Crazyflie with PID constants that are not tuned well. In order to reduce the headache for the students and make everything safer, the PID's will be tuned well enough to keep the Crazyflie 'stable', but with obvious oscillations and performance issues.

They will be asked to tune the PID's to meet certain specifications, such as a settling time or minimizing overshoot, etc. They could even get bonus points for doing more than asked, etc.

Reducing Academic Dishonesty:

Since its pretty easy for working PID constants to be passed around (cheating) we would ask each group to achieve different specifications to reduce the possibility of cheating.

6.2.1 Lab Results Example (Single Crazyflie PID Test)

This is an example of results that a student would report during the Hand Tuned PID part of the exercise once they are confident that their design meets the specifications. The test is to see if the 9 PID's are able to fly a single Crazyflie via the Client and hold a specified

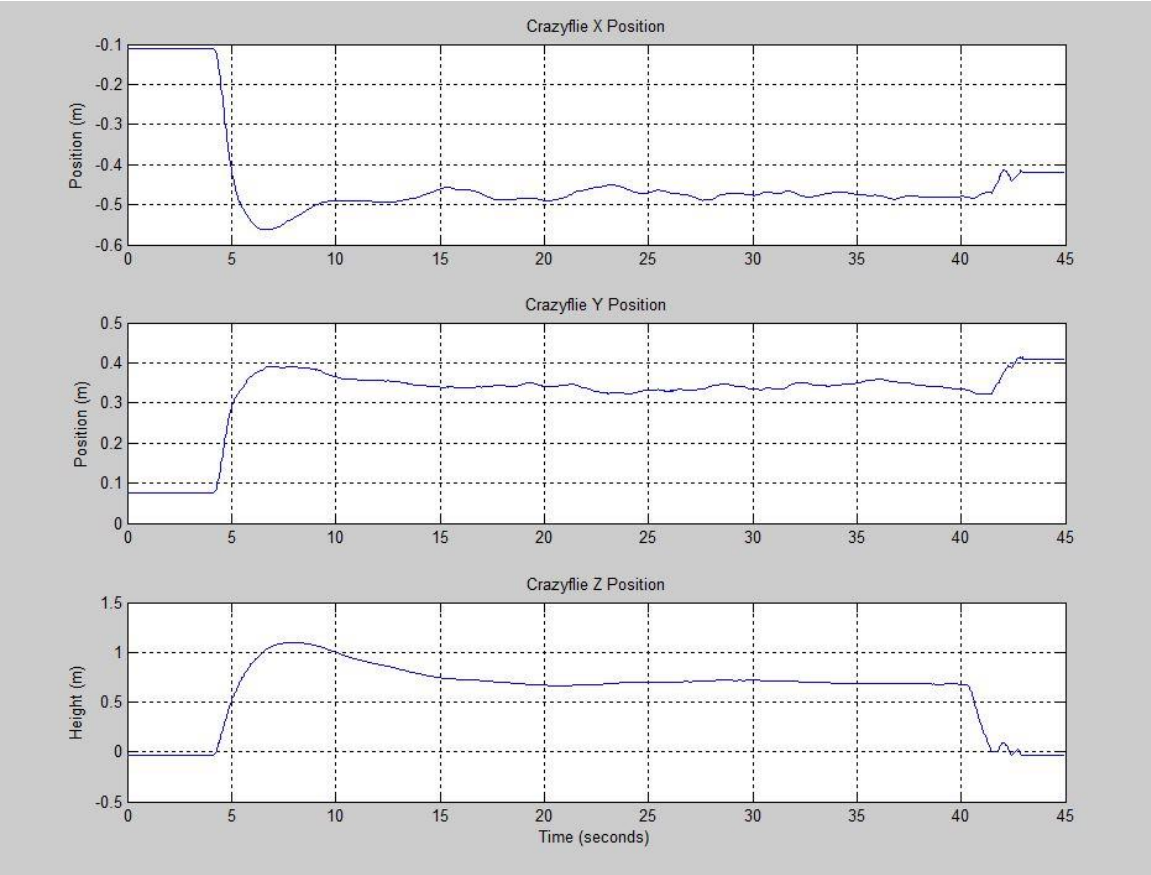

Figure 6.3 – Single Crazyflie X, Y, and Z Response (Hold Position)

position. We used 1 Radio, and 1 Crazyflie and the 9 hand-tuned PID's to run this test.

In Figure 6.3 we can see the Crazyflie starts on the ground, and then 4 seconds into the test we command the Crazyflie to the setpoint $X = -0.475$, $Y = 0.340$, and $Z = 0.750$ (units in meters). We can see that the flight behavior remains stable after reaching its steady state 10 to 15 seconds after takeoff. So we were able to achieve a settling time of about 5 to 10 seconds, and a rise time of about 1 to 2 seconds.

Table 6.1 – Example X-Position PID Constants

Symbol	Quantity
k_p	20
k_i	10
k_d	22

Example table of PID values the students would report. (This example is only for 1 of the 4 PID's they would have to tune; the others have been omitted to conserve space.)

The PID values that we used to get this response are shown in Table 6.1.

We then move on to teaching them how to model a PID controlled system in Simulink and have them identify model parameters and verify their model with the PID constants they tuned and actual Crazyflie flight data recorded in the log files.

Once the model is checked to be correct they can use the model to generate optimized PID constants for the system. They can then compare the results with what they did by hand to see how close/far off they were, and how much easier things are when you have an accurate model of the system.

We are currently working on deriving a functioning model for the Crazyflie system, but it is more challenging than we expected. This issue is discussed in greater detail in the Future Work section below. Once we get a functional model of the system we can then expand this exercise into the Simulink modeling and derivation of the system parameters we found, providing a solutions manual for the lab work that the students will complete.

6.3 Trilateration Location Estimation

Another example of a potential educational exercise is developing another method of localization for the Crazyflie. The idea is to use the Radio Signal Strength received from a transmission to estimate the distance one Crazyflie is from another, or alternatively the distance between one Crazyflie and the Radio itself. Then using multiple measurements of this strength from different locations, we should be able to estimate the position of a Crazyflie with enough measurements. This concept is not a new idea, it has been done before in GPS, radar, and even in the camera system we use in the lab today utilizes this concept. This would be different in that we could use the swarm itself to locate the position of a Crazyflie.

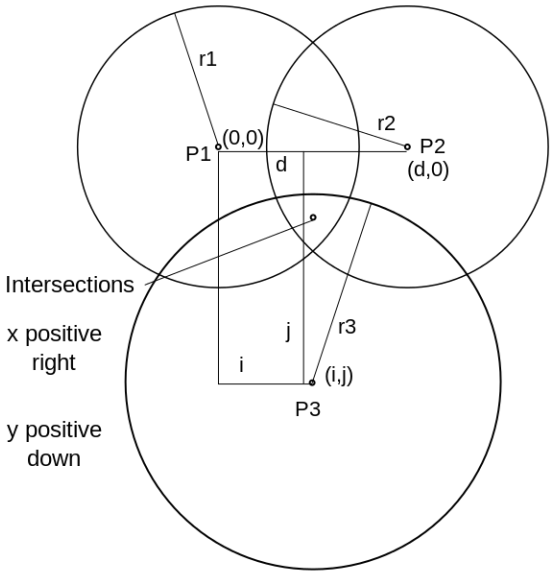

Figure 6.4 – Trilateration Concept Drawing (Intersection of 3 Spheres to Estimate Location) [14]

I think this problem would be interesting to students because it has many practical uses in the real-world and it would expose them to the concepts of stochastic noise modeling and noise reduction techniques that are common in controls applications.

Learning Objectives:

1. Discovering how trilateration is done (intersection of 3 or more spheres to estimate position) solving the equations
2. Creating a stochastic noise model (Gaussian Noise, Rayleigh Noise, etc.)
3. Simulating noise model and Trilateration estimation technique

4. Applying Trilateration estimation to Crazyflie Platform
5. Comparison between Crazyflie and Simulation
6. Noise reduction techniques

(See APPENDIX B: for more detail)

Teaching Content:

We have 3 Radios well-spaced around the unknown object. Each Radio represents the center of a sphere, with each one measuring the radio signal strength received from the unknown object, and using the signal strength to determine the radius of its sphere. We then solve for the intersection of the 3 spheres and get 2 potential solutions, one with a positive Z component and one with a negative Z component (see APPENDIX B: for more detail on how to solve these equations). Luckily, in flight applications we usually define the ground level as the origin, so if we had a negative Z estimate then that means that either we have already failed to recover (Crazyflie has crashed), or that this estimate is wrong. So we can easily eliminate that negative Z solution, leaving us with just 1 potential solution for X,Y, and Z if there is an intersection of the 3 spheres.

Then the class will cover how to generate a noise model consisting of Gaussian and Rayleigh noise to simulate a lossy transmission. For now we use a random walk to simulate the position of a Crazyflie hovering in place with no global localization method (GPS). This random walk is used to simulate a dead-reckoning behavior (prone to drift). Although this model could change depending on the situation we are trying to simulate (i.e. Fixed wing drone). With these two simulations we will show them how to simulate the trilateration estimate using 3 spheres and 1 unknown location narrowing the estimate down to 1 possible solution.

Lab Results Example (3 Radio Trilateration Estimate):

In Figure 6.5, using Matlab, we plot the simulation results we got from 1000 partial realizations of our trilateration estimate. The 3 Radios are located at $[0, -3, 2]$, $[1, 1, 5]$, and $[-3, 1, 1]$ (in meters), and since we are using a random walk with mean 0 to model the signal strength loss for each Radio, we expect the true location to be at the centroid of those 3 positions. Therefore, the results of our simulation yield:

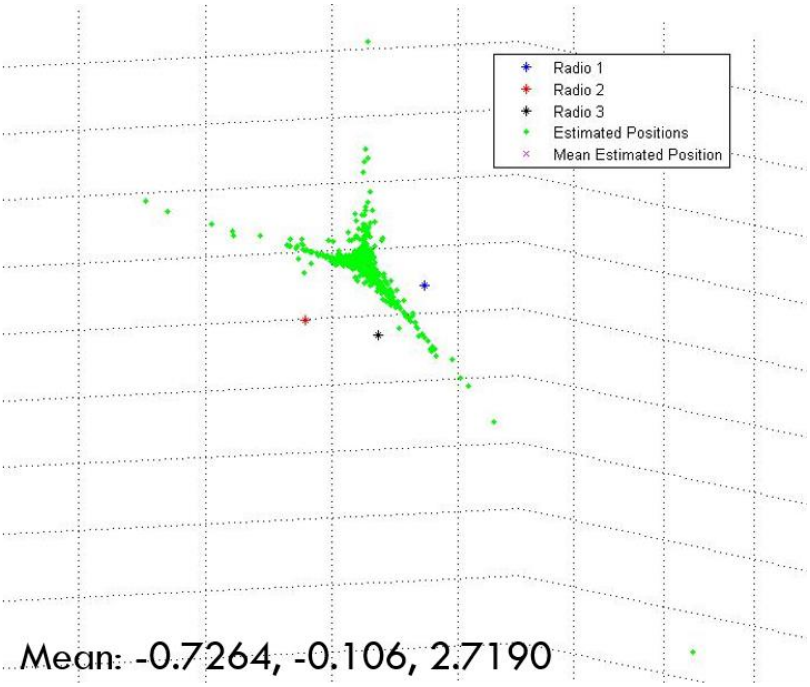

Figure 6.5 – Simulated Trilateration Estimate using Simulated Noise Profile and Random Walk to Model a Hovering Crazyflie

- 'True' location is $[-0.6667, -0.3333, 2.6667]$ meters
- Estimated location is $[-0.7264, -0.106, 2.7190]$ meters
- This gives us an error range of about $\pm 0.227\ m$ at worst

While this was purely simulation, the noise model we derived did have a large amount of variance ($\sigma \approx 7$) so the estimate shows promise, but we have not been able to attempt a test on the Crazyflie yet due to some Firmware flashing issues. Once we have those tests complete, we can compare results and create a solutions manual for grading the students' lab work.

If we still have time afterwards we could then discuss some methods of noise reduction and test them out on the simulations and real-world tests they have been working on.

For more detail on how we actually solved this problem, and the noise models and random walk implementation we used please see [APPENDIX B:].

CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1 Summary

In this work we developed a platform that is inexpensive, versatile, and accessible. Modifying the Crazyflie platform we took what used to control only one Crazyflie by hand, and turned it into an autonomously controlled, multi-Crazyflie, multi-Radio platform. We were able to re-design the Client in C and C++ while still maintaining most of the functionality of the original Client and GUI. We integrated this system with the OptiTrack Camera system and set up a VRPN connection to get the localization data from the OptiTrack software. We were then able to tune the X, Y, and Z location PID's that we created for a single Crazyflie and expand that control scheme into simultaneously controlling the flight of multiple Crazyflies. We also implemented a gesture recognition system and were able to control the Crazyflies flight utilizing those gestures.

With the flight controller functional we also were able to expand the platforms Radio interface to include multi-Radio support and we were able to support multiple Crazyflies per Radio. This greatly improved the scalability of the platform by enabling up to 3 Crazyflies per Radio and an unlimited number of Radios per computer as long as you don't end up overlapping radio channels.

Yet there's still a lot of work we can do on the platform to make it even more useful.

7.2 Model Derivation and Verification

One thing we are currently working on is developing a model for the Crazyflie. The model will open up a lot of potential for educational experiments that can be planned, and more importantly performed safely. Normally this is the very first step in a controls project.

The challenge with modeling the Crazyflie though comes down to its small size. It is very difficult to measure model parameters when the measurements start approaching your margin of error, as is the case with the Crazyflie. Others have also tried parameterizing the Crazyflie and even reported their results in [25]. When we applied their parameters to our physical model, the simulation resulted in an unstable system whereas our actual flight was stable. This alone shows how difficult it is to determine these parameters.

At the start the goal of this project was to develop a system for taking experiments out of simulation, and while it still is the driving force behind the project, there is also a need for simulating these experiments before we try them in the real world.

7.3 Firmware Location PID's

We mentioned the issues of timing in the Multi-Crazyflie example and how running the callbacks takes a significant portion of time away from meeting the new data deadline. The best way to solve this problem is to take the PID's being calculated on the Client and offload them to the Firmware of each individual Crazyflie. This way the platform would behave as a distributed control network, where the only thing the client does is relay the Crazyflies location data directly to each Crazyflie and logs the data. This would speed up the loop time immensely, and allow the Crazyflies themselves to spread out the workload of the platform.

Another benefit of spreading the workload means you can scale the platform to manage even greater numbers than before. Currently without spreading the workload we are approaching our limit on the size of the swarm. The system starts to become unstable as you approach a consistent time between callbacks of around 0.04 - 0.05 seconds. Currently our

worst case scenario has a consistent time between callbacks of around 0.03 seconds as you can see in Figure 5.2.

7.4 Parallelization

Another option for reducing this timing issue would be to allow the callbacks to process in parallel. This way we could calculate control outputs for more than one Crazyflie at a time, which is the bulk of the time taken in the callback, and then gather those outputs and send them to their respective Crazyflies over the Radio. This would not be the preferred strategy as we could start running into issues with blocking as the Radio tries to send one set of data when another callback finishes. This would just be an interesting alternative that could be applied to other portions of the code like the keyboard inputs or sensor measurements.

7.5 Tuning Angular Rate PID's on Firmware

The angular rate constants for the PID's that Bitcraze provided have room for improvement. The problem is that in order to tune these parameters we need to re-flash the firmware each and every time. This would take incredibly long to tune in this manner as we slowly increase those constants. We would like to develop a system in which we can send parameter constants via the Radio to the Crazyflie and be able to modify those parameters on the fly. This would significantly decrease tuning time and could be used for many other useful applications onboard the Crazyflie.

7.6 Inter-Crazyflie Communication

We would also like to enable communication between Crazyflies, that way we could eventually eliminate the Client all together if we needed to. This would require rewriting the

radio code on the NRF chip in the Firmware to allow transmission and receive functionality on the Crazyflie itself. Currently the Crazyflie can only receive and is always listening for commands from the Client. The code base to enable transmission is there since the Crazyflie has to send an ack and data packets back to the Client when requested, so the rework wouldn't be that complicated. The bulk of the work would be in figuring out what to do and where to store the data that is being transmitted between Crazyflies.

7.7 Trilateration using Radio Signal Strength Indicator (RSSI)

We developed a method of locating a Crazyflie by taking multiple RSSI measurements from Radios located around the room. We used trilateration, where you solve for the intersection of multiple spheres to locate a position in space. The code for solving the sphere equations and converting the result into the basis vectors of our coordinate space is complete, we just need the RSSI data from the Crazyflie. With 3 spheres we can solve the equations down to 2 possible locations, where we can usually eliminate one option due to its infeasibility (such as a ±Z axis result), but if we use 4 Radios then we can solve the equations down to a single unique estimate of the Crazyflies location.

There were a couple ways we planned on accomplishing this. The first way was to place 3-4 Radios around the room at fixed known locations and then recording the Radio signal strength of the single Crazyflie to each of the Radios. That would give us 3-4 spheres to solve for and we could use the code we developed to figure out the Crazyflie's location. The other method involved using 3-4 known Crazyflies (with attached constellations) and 1 Unknown Crazyflie (no constellation) and measuring the radio strength between all known Crazyflies and the Unknown Crazyflie. This method is more complex and would require reworking the NRF radio code on the Firmware to allow T_x and R_x from the Crazyflie, but

this would be the ideal case as we could use this method anywhere and not be restricted to using fixed radio locations.

Unfortunately, we were not able to get the RSSI values reporting back from the Crazyflie logs. It would always report back 0 whenever we attempted to read the value. This project is extremely close to being viable, we just didn't have enough time to sort out the Firmware issues causing the value to not get recorded.

APPENDIX A: PID CONTROLLER

Here we will show the technical details behind the PID control algorithm and controller layout.

(To Do)

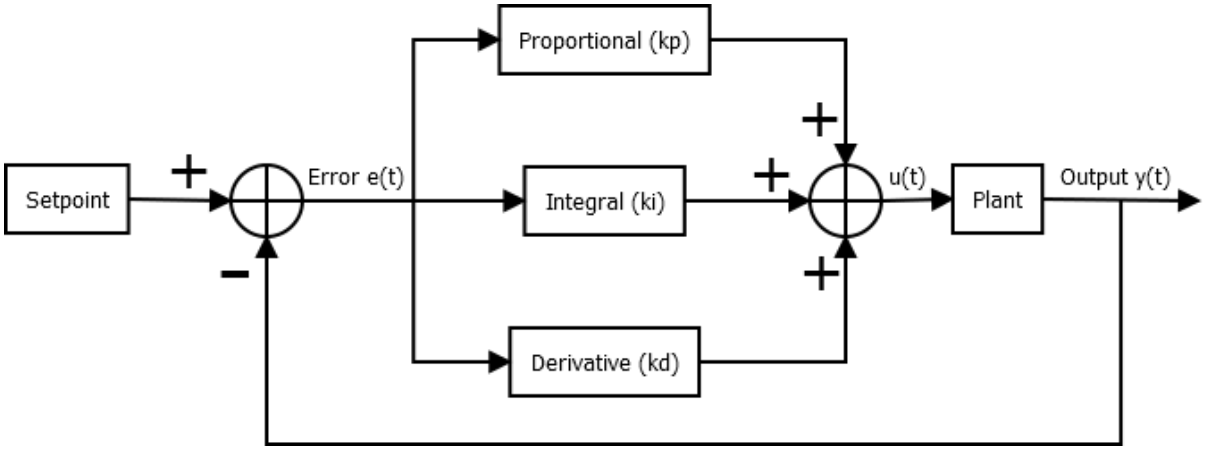

Figure A.1 – Block Diagram of a PID controller

PID Control Equation [26]:

From the block diagram shown in Figure A.1 we get (Eq. A.1 in continuous time:

$$\underline{\text{Continuous Time PID Equation:}}\ u(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau + k_d \frac{de(t)}{dt} \qquad \textbf{(Eq. A.1)}$$

We can see that the controller is essentially using the error between the desired setpoint and the actual system state, and doing 3 separate calculations. It then takes the results of those 3 calculations and sums them all together to get the controller output $u(t)$ which is used as an input into the system we are trying to control.

Discretize the PID Equation:

If we take (Eq. A.1 we can discretize it for use in an embedded control system, like our Swarm Platform. The continuous-time integral can be approximated by the Riemann sum

$$\int_0^t e(\tau)\, d\tau \approx \sum_{j=0}^{n} e[n]\, \Delta n \qquad \textbf{(Eq. A.2)}$$

where the Δn term is represented as the update rate of the PID controller (in our case 0.01 s).

The derivative term can be approximated by the change in error between 2 consecutive time steps, or the slope.

$$\frac{de(t)}{dt} \approx e[n] - e[n-1] \quad \textbf{(Eq. A.3)}$$

If we substitute these approximations in, we get a discretized PID control equation that we can utilize in discrete systems, like embedded controls applications. The discrete approximation is shown below in (Eq. A.4.

Discrete PID Equation: $u[n] = k_p\, e[n] + k_i \sum_{j=0}^{n} e[j] + k_d\, (e[n] - e[n-1])$ **(Eq. A.4)**

We implemented this control algorithm in C for our platform as shown below:

Implemented in C Code:

```
float pidUpdateAngle(PidObject* pid, const float measured, const bool updateError) //**USED FOR
ATTITUDE CONTROL
{
          float output;

          if(updateError)
          {
                    pid->error = pid->desired - measured;  //Calculates current error
          }

          pid->integ += pid->error * pid->dt;            //Calculates integral term
          if(pid->integ > pid->iLimit)                   //Checks that the integral term stays
below the integral limit (anti-windup)
          {
                    pid->integ = pid->iLimit;
          }
          else if(pid->integ < pid->iLimitLow)
          {
                    pid->integ = pid->iLimitLow;
          }

          pid->deriv = (pid->error - pid->prevError) / pid->dt;  //Calculates derivative term

          pid->outP = pid->kp* pid->error;
```

```
            pid->outI = pid->ki * pid->integ;
            pid->outD = pid->kd * pid->deriv;

            output = pid->outP + pid->outI + pid->outD;  //Adds up all 3 calculated terms for output

            if(output > pid->angularLimit)              //Limits amount Crazyflie can tip
            {
                        output = pid->angularLimit;
            }

            if(output < pid->angularLimitLow)             //Limits amount Crazyflie can tip
            {
                        output = pid->angularLimitLow;
            }

            pid->prevError = pid->error;                 //Stores the current error for next loop

            return output;
}
```

APPENDIX B: TRILATERATION METHOD

B.1 Trilateration Position Estimate [14]

Rather than relying on a pure dead reckoning method to estimate position, we can add a secondary system as a way to get a 'second opinion' on the lost quad's location. Using the other quads in the swarm we ping packets off of the lost quad and read back the RSSI signal strength information. With three or more different known quads we can represent this radio strength as three spheres of varying radii. Then we can apply Trilateration to estimate the lost quad's position from the intersection of the spheres. A depiction of this process is shown in Figure B.1.

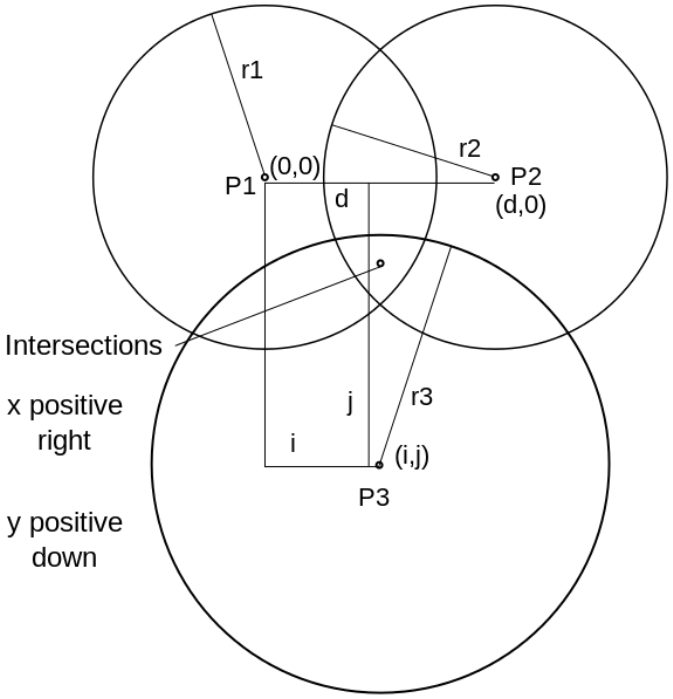

Figure B.1 – Trilateration Estimate [14]

Combining these two estimates would allow us to have a much better estimate than that of which either system alone would provide. We can get a good local position from the dead reckoning, while periodically updating the radio strength estimate. This balance ensures that we don't accumulate too much error over time with the dead reckoning, and that we don't sacrifice too much battery power by constantly pinging the radio strength between quads. Each system makes up for the weaknesses of the other, making them an ideal combination.

Calculating the location estimate via trilateration is straightforward. For the instance depicted in Figure B.1 we have a few limiting assumptions.

1. Radio P1 must be located at the origin (0,0)
2. Radio P2 must be along the x-axis
3. All radios are in the $z = 0$ plane

These assumptions are in place for ease of computation and for our case would invalidate any real world system trying to use this method. First, we will demonstrate the calculation in its simplified form and then work on getting around these assumptions.

Starting with the 3 sphere equations (variables are as shown in Figure B.1)

- $r_1^2 = x^2 + y^2 + z^2$ **(Eq. B.3)**
- $r_2^2 = (x - d)^2 + y^2 + z^2$ **(Eq. B.2)**
- $r_3^2 = (x - i)^2 + (y - j)^2 + z^2$ **(Eq. B.1)**

We can then solve for x, y, and z

- $x = \frac{r_1^2 - r_2^2 + d^2}{2d}$ **(Eq. B.6)**
- $y = \frac{r_1^2 - r_3^2 - x^2 + (x-i)^2 + j^2}{2j}$ **(Eq. B.4)**
- $z = \pm\sqrt{r_1^2 - x^2 - y^2}$ $(2\ possible\ solutions)$ **(Eq. B.5)**

Note that this method, with only 3 spheres, can narrow down the possible position to no more than two locations. Fortunately, we can easily eliminate one of the two potential solutions due to the nature of the application itself. In almost all cases the z-plane will be oriented with ground level as origin. So if we get a position estimate with a negative z-

coordinate then we've either already failed to recover the quad or the solution must be false. We can therefore make the assumption that only the positive z-coordinate solution is the only valid estimate.

Now that we have the solutions to X, Y, and Z to get around the previous assumptions we can redefine the three basis vectors that define the coordinate plane. This is because no matter how we orient the system, the three centers always form a unique plane that can be shifted and rotated around to match the orientation we need. (Variables used are identical to the ones in Figure B.1)

Unit Vectors

$$U_x = \frac{P_2 - P_1}{\|P_2 - P_1\|} \quad \textbf{(Eq. B.8)}$$

$$\mathrm{U_y} = \frac{\mathrm{P_3} - P_1 - iU_x}{\|\mathrm{P_3} - P_1 - i\,U_x\|} \quad \textbf{(Eq. B.9)}$$

$$U_z = U_x \times U_y \quad \textbf{(Eq. B.7)}$$

Distance Vectors

$$i = U_x(P_3 - P_1) \text{ (Signed x Magnitude from } P_1 \text{ to } P_3) \quad \textbf{(Eq. B.12)}$$

$$j = U_y(P_3 - P_1) \text{ (Signed y Magnitude from } P_1 \text{ to } P_3) \quad \textbf{(Eq. B.10)}$$

$$d = \|P_2 - P_1\| \text{ (Distance between } P_1 \text{ and } P_2) \quad \textbf{(Eq. B.11)}$$

We can then solve the simplified trilateration for X, Y, and Z using the above quantities and that will give us the estimated position in new coordinate space as:

$$\boxed{\widehat{p_{1,2}} = P_1 + xU_x + yU_y \pm zU_z} \quad \textbf{(Eq. B.13)}$$

Using this method increases the complexity a bit, but the benefit of being able to use this tool at any orientation of the radio plane is worth the extra computation! Now that we have

this tool at our disposal, we need some way to relate the power lost in the radio channel to a distance from the radio itself.

B.2 Log-Distance Path Loss Model

In practical applications relating the radio path loss is as simple as just subtracting the power received from the power the sender originally transmitted with. This power loss is known as the Path Loss of a radio signal, and it can be attributed to a vast array of interferences and disturbances. Things like shadow fading, where large objects blocking a direct line of sight (LOS) causing power losses as the signal passes through, and multipath fading, where reflections of the original signal arrive at the receiver with a time-delay and power reduction. There are many other reasons for power losses, but these two are what we are going to focus on for now.

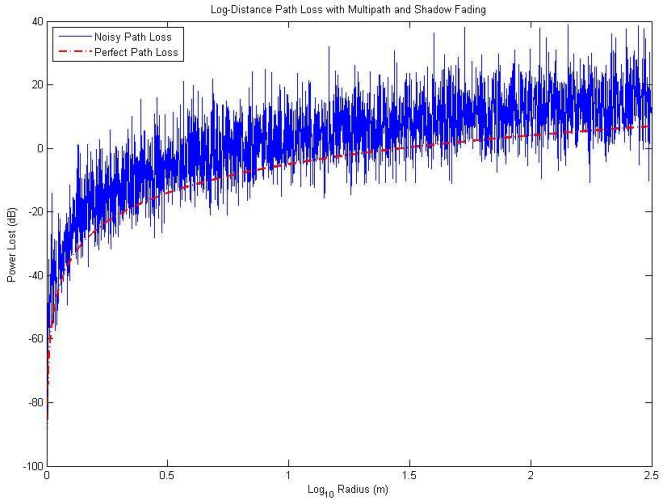

Figure B.2 – Simulated Path Loss Model ($\alpha = \sigma = 7, \gamma = 3$)

In order to model these effects, we used the Log-Distance Path Loss Model. The model is as follows:

$$Path\ Loss = PL_0 + 10\gamma \log_{10}\left(\frac{d}{d_0}\right) + N \qquad \textbf{(Eq. B.14)}$$

- d_0 is a reference distance chosen as an environment baseline (usually in meters)
- PL_0 is the path loss (dB) at a d_0
- γ is the Path Loss Exponent, also gathered from empirical measurements

- d is the distance (m) from the radio receiver
- N is the Noise profile used (can be a combination of noise distributions)

As mentioned previously we modeled the noise (N) to include shadow fading and multipath fading. The shadow fading was modeled as a zero mean Gaussian random process with parameter σ, and the multipath fading was modeled as a Rayleigh random process with parameter α. Both types were considered as additive noise. You can see the effects of this noise in Figure B.2. The red dotted line is the model without any additive noise, and the blue is the model with both types of noise included. From empirical studies [13] we chose both $\alpha = \sigma = 7$ and the loss exponent $\gamma = 3$. The path loss exponent ranges from $\gamma \approx 2$ when transmitting in free space, to $\gamma \approx 4$ when transmitting with no direct LOS. The reason these values were chosen specifically were to simulate a medium noise indoor environment with a lot of multipath noise.

One thing to note is that varying these parameters doesn't seem to significantly change the amount of power loss at similar distances, if you compare Figure B.2 and Figure B.3. Even when doubling the noise with $\alpha = \sigma = 14$ and $\gamma = 4$, in Figure B.3, the loss only has its standard deviation grow by about 7, which makes sense because we're really only changing the variance of the random processes. As we will see in the results, it does not cause a significant loss in accuracy of the position estimation.

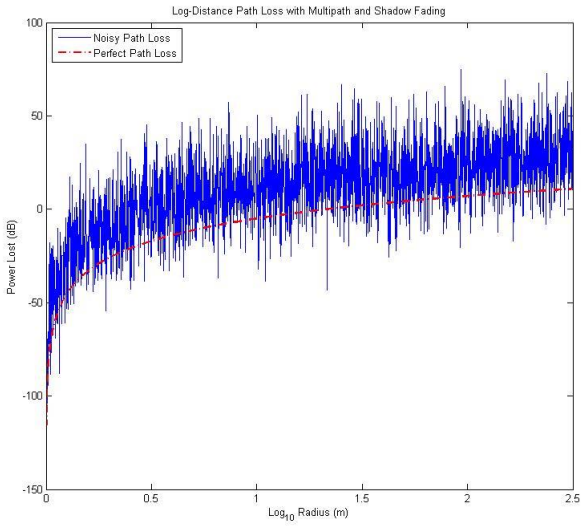

Figure B.3 – Path Loss with more Noise ($\alpha = \sigma = 14, \gamma = 4$)

Now that we have a model we can relate the power lost to a specific distance from the transmitter. By solving for the distance (d) in the equation above we get the following.

$$d = d_0 10^{\frac{Path\ Loss - PL_0 - N}{10\gamma}}$$

The last thing we need to solve this problem is to model the behavior of the lost quad.

B.3 Lost Quad Model - Random Walk

If we recall the application we covered previously, we have a quad with a malfunctioning GPS chip. The quad's only hope of survival now is to try to maintain its position via an emergency dead reckoning mode. As we mentioned earlier, this method of localization is only accurate for a limited amount of time. With each time step it accumulates some error that cannot be accounted for without some other means of positioning. Thus over time the quad will drift away from the position it's trying to hold, unaware that it has moved at all.

This behavior is perfectly described as that of a Random Walk. In order to simulate a random walk we needed to construct a discrete partial realization of a process where its standard deviation grows with time, but maintains a zero mean. To accomplish this we took a Gaussian white noise process ($x(t)$) and used a Riemann approximation to discretize the process.

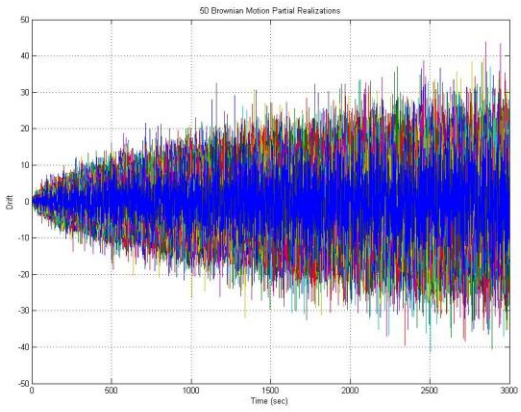

Figure B.4 – 50 Random Walk Partial Realizations

$$y(t) = \int_{\tau=0}^{t} x(\tau) \Rightarrow \boxed{y(k\Delta) = \sum_{j=1}^{k} \tilde{x}(j\Delta)\Delta} \qquad \textbf{(Eq. B.15)}$$

- $y(k\Delta)$ is a discretized partial realization
- Δ is the step size of the discrete process
- $x(\tau)$ is Gaussian white noise with zero mean and parameter σ

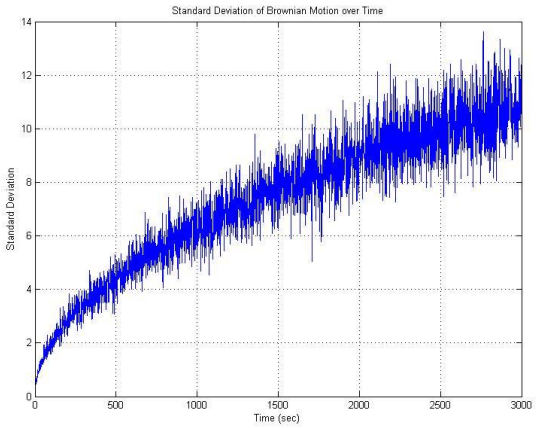

Figure B.5 – Standard Deviation of Random Walks over Time

In Figure B.4 we ran 50 partial realizations of this process and you can easily see that it fulfills the requirements we wanted, specifically the zero mean and time varying standard deviation. Figure B.5 shows how the standard deviation changes with time.

B.4 Matlab Code

Random Walk Implementation

```
figure(1)
var_x_rw = 0.7; %orignal 0.7
delta = 0.2; %orignal 0.2
sum1 = zeros(30,2500);
sum2 = zeros(1,2500);
sum3 = 0;
g = 0;

x_rw = zeros(1,2500);

for m = 1:1:50
        for k = 1:1:3000
                x_rw = normrnd(0,var_x_rw,k,1);
                sum3 = 0;
```

```
                for i = 1:1:k
                        x_rw(i) = x_rw(i)*delta;
                        sum3 = sum3 + x_rw(i);
                end
                sum2(k) = sum3;
        end
        g = g + 1;
        for h = 1:3000
        sum1(g,h) = sum2(h);
        end
end

std(sum1(:,20));

plot(sum1')
grid on
title '50 Brownian Motion Partial Realizations'

ylabel 'Drift'

figure(2)
for p = 1:3000
        std_rw(p) = std(sum1(:, p));
end

plot(std_rw)
grid on
title 'Standard Deviation of Brownian Motion over Time'
xlabel 'Time (sec)'
ylabel 'Standard Deviation'

sum1 = sum1(:,501:end); %Removed first 500 to assume lost quad does not start
right at origin

randIndex = randi(m, 3, 1);

pathLoss1 = sum1(randIndex(1),:)'; %Change to 3 power losses and apply to log dist
loss
pathLoss2 = sum1(randIndex(2),:)';
pathLoss3 = sum1(randIndex(3),:)';

figure(3)

subplot(3,1,1)
plot(pathLoss1,'-b')
```

```
title '3 Channel Random Walk Path Losses'
ylabel 'Packet Power Loss (dB)'
xlabel 'Time (seconds)'

subplot(3,1,2)
plot(pathLoss2,'-g')
ylabel 'Packet Power Loss (dB)'
xlabel 'Time (seconds)'

subplot(3,1,3)
plot(pathLoss3, '-k')
ylabel 'Packet Power Loss (dB)'
xlabel 'Time (seconds)'
```

Log-distance Path Model

```
pathLoss = zeros(2500,1);
distRef = 1; %Reference Distance in m (d0) = 1 meters
pLossRef = -5; %Measured Power lost at ref distance in dB
pLossRef = repmat(pLossRef, length(pathLoss),1);
dist = [0:1/1000:(length(pathLoss)-1)/1000]';

sigma = 7; %Common value for indoor office space with decent interference
pLossExp = 3; %Common value for indoor office space with decent interference
%Wireless communications principles and practices, T. S. Rappaport, 2002,
Prentice-Hall

shadowNoise = normrnd(0,sigma,length(pathLoss),1); %Gaussian RV zero mean,
representing shadow fading
multiNoise = raylrnd(sigma,length(pathLoss),1); %Rayleigh Distributed noise
representing Multipath
% directNoise = random('rician', length(pathLoss),1);

perfPathLoss = pLossRef + (10*pLossExp * log10(dist./distRef));
pathLoss = pLossRef + (10*pLossExp * log10(dist./distRef)) + shadowNoise +
multiNoise;
figure(4)

hold off
plot(dist, pathLoss, '-b')
hold on
```

```
plot(dist, perfPathLoss, '-.r', 'LineWidth', 2.0)
title 'Log-Distance Path Loss with Multipath and Shadow Fading'
ylabel 'Power Lost (dB)'
xlabel 'Log_1_0 Radius (m)'
legend('Noisy Path Loss', 'Perfect Path Loss', 'Location', 'NorthWest');
```

Applying Path Loss Model to Random Walk Power Losses

```
%Noisy Path Loss Radii
r1 = distRef * 10.^((pathLoss1 - pLossRef - shadowNoise -
multiNoise)/(10*pLossExp));
shadowNoise = normrnd(0,sigma,length(pathLoss),1); %Gaussian RV zero mean,
representing shadow fading
multiNoise = raylrnd(sigma,length(pathLoss),1); %Rayleigh Distributed noise
representing Multipath

r2 = distRef * 10.^((pathLoss2 - pLossRef - shadowNoise -
multiNoise)/(10*pLossExp));
shadowNoise = normrnd(0,sigma,length(pathLoss),1); %Gaussian RV zero mean,
representing shadow fading
multiNoise = raylrnd(sigma,length(pathLoss),1); %Rayleigh Distributed noise
representing Multipath

r3 = distRef * 10.^((pathLoss3 - pLossRef - shadowNoise -
multiNoise)/(10*pLossExp));
```

Basis Vectors for Trilateration

```
posEst = zeros(2500,3);

P1 = [0 -3 2]; %Vectors of radio locations from origin
P2 = [1 1 5]; %Good
P3 = [-3 1 1];

% P1 = [0 -3 2]; %Vectors of radio locations from origin
% P2 = [1 -3 5]; %BAD (Aligned in y-axis)
% P3 = [-3 -3 1];

xUnitVec = (P2 - P1)/norm(P2-P1,2);
xMag13 = xUnitVec*(P3-P1)'; %xMag of vector P1 to P3

yUnitVec = (P3 - P1 - xMag13*xUnitVec)/norm(P3 - P1 - xMag13*xUnitVec,2);
```

```
yMag13 = yUnitVec*(P3-P1)'; %yMag of vector P1 to P3

zUnitVec = cross(xUnitVec, yUnitVec);

mag12 = norm(P2-P1,2);
```

Solving the Sphere Equations for x, y, and z (Noisy Path Loss)

```
x = (r1.^2 - r2.^2 + mag12^2)/(2*mag12);

y = ((r1.^2 - r3.^2 + xMag13^2 + yMag13^2)/(2*yMag13)) - (xMag13/yMag13)*x;

z = sqrt(r1.^2 - x.^2 - y.^2);

zAdjust = z*zUnitVec;

if(min(real(zAdjust(:,3))) < 0) %Eliminates the -z solution as an option
        z = -z;
        'Using -z'
else
        z = z;
        'Using +z'
End

posEst(:,1:3) = repmat(P1,length(x*xUnitVec),1) + (x * xUnitVec) + (y * yUnitVec) +
(z * zUnitVec);

posEst = real(posEst); %Solutions could be complex if no solution found, ignore
these

figure(5)
hold off
plot(posEst(:,1),'-b')
hold on
plot(posEst(:,2),'-g')
plot(posEst(:,3),'-k')
title 'Trilateration Position Estimation'
ylabel 'Position (m)'
xlabel 'Time (s)'
legend('X Position', 'Y Position', 'Z Position')
```

Create Coordinate Space

```
figure(6)
hold off
plot3(P1(1),P1(2),P1(3), '*b',P2(1),P2(2),P2(3),'*r',P3(1),P3(2),P3(3),'*k')
grid on
title 'Coordinate Space' %Plots the Locations of the Radios in the space
xlabel 'X Position (m)'
xlim([-20,20])
ylabel 'Y Position (m)'
ylim([-20,20])
zlabel 'Z Position (m)'
zlim([-20,20])
hold on

plot3(posEst(:,1),posEst(:,2),posEst(:,3),'.g') %Plot the position estimates in the
space
meanPosEst = mean(posEst)
stdPosEst = std(posEst)
plot3(meanPosEst(1),meanPosEst(2),meanPosEst(3),'xm')
legend('Radio 1', 'Radio 2', 'Radio 3', 'Estimated Positions','Mean Estimated Position')
```

Principle Component Analysis (PCA) of Position Estimate

```
[U, S, V] = svd(posEst);

for i = 1:3
        singDiag(i) = S(i,i);
        singDiag = sort(singDiag, 'descend');
end

figure(7)

plot(singDiag) %Plotting Singular Values
grid on
title('Singular Values')

k = 1;

vk = V(:,1:k);

for m = 1:k
        scorek(:,m) = U(:,m)*singDiag(m); %Calculates the Top k Scores
```

```
end

posEstk = scorek(:,1)*vk';

figure(8)
hold off
plot3(P1(1),P1(2),P1(3), '*b',P2(1),P2(2),P3(3),'*r',P3(1),P3(2),P3(3),'*k')
grid on
title 'Coordinate Space' %Plots the Locations of the Radios in the space
xlabel 'X Position (m)'
xlim([-20,20])
ylabel 'Y Position (m)'
ylim([-20,20])
zlabel 'Z Position (m)'
zlim([-20,20])
hold on

plot3(posEstk(:,1),posEstk(:,2),posEstk(:,3),'.g') %Plot the position estimates in the
space
meanPosEstk = mean(posEstk)
stdPosEstk = std(posEstk)
plot3(meanPosEstk(1),meanPosEstk(2),meanPosEstk(3),'xm')
legend('Radio 1', 'Radio 2', 'Radio 3', 'PCA Estimated Positions','PCA Mean Estimated
Position')
```

REFERENCES

1. G. Casadei, M. Furci, R. Naldi and L. Marconi, "Quadrotors motion coordination using consensus principle," 2015 American Control Conference (ACC), Chicago, IL, 2015, pp. 3842-3847. doi: 10.1109/ACC.2015.7171929. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7171929&isnumber=7170700

2. Eberhart and Yuhui Shi, "Particle swarm optimization: developments, applications and resources," Evolutionary Computation, 2001. Proceedings of the 2001 Congress on, Seoul, 2001, pp. 81-86 vol. 1. doi: 10.1109/CEC.2001.934374. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=934374&isnumber=20223

3. W. Hönig, C. Milanes, L. Scaria, T. Phan, M. Bolas and N. Ayanian, "Mixed reality for robotics," *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, Hamburg, 2015, pp. 5382-5387. doi: 10.1109/IROS.2015.7354138. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7354138&isnumber=7353104

4. *Vrpn 7.31 source code*, [Online]. Available: https://github.com/vrpn/vrpn/releases/tag/version_07.31.

5. *Tracking Tools 2.4.0 User's Guide,* OptiTrack.

6. M. Rich, "*Model development, system identification, and control of a quadrotor helicopter*," Master's thesis, Iowa State University, Ames, IA, 2012.

7. Dunkley, Oliver, *Crazyflie PC client – With Free Fall Recovery and Camera Overlay HUD*, (2015). Repository: https://bitbucket.org/omwdunkley/crazyflie-pc-client-camera.

8. Winkler, Jan, *Crazyflie Nano C++ Client Library*, (2015). Repository: https://github.com/fairlight1337/libcflie

9. *Crazyflie Nano Quadcopter Firmware*, (2015). Repository: https://github.com/bitcraze/crazyflie-firmware

10. *Bitcraze Virtual Machine*. [Online] Available: https://wiki.bitcraze.io/projects:virtualmachine:index

11. *Crazyflie Wiki*. [Online] Available: https://wiki.bitcraze.io/index

12. Oguejiofor, O., et al. "Outdoor localization system using RSSI measurement of wireless sensor network." *International Journal of Innovative Technology and Exploring Engineering* 2.2 (2013): 1-6.

13. Sharma, Navin Kumar. "A weighted center of mass based trilateration approach for locating wireless devices in indoor environment." *Proceedings of the 4th ACM international workshop on Mobility management and wireless access*. ACM, 2006.

14. Trilateration 3 Sphere Example, Wikipedia, (2016). Available: https://en.wikipedia.org/wiki/Trilateration

15. *Crazyflie Wiki – Communication Protocol.* [Online] Available: https://wiki.bitcraze.io/projects:crazyflie:firmware:comm_protocol

16. Distributed Autonomous Networked Control Lab – Crazyflie Swarm. [Online] Available: http://wikis.ece.iastate.edu/distributed-autonomous-and-networked-control-lab/index.php/Crazyflie_Swarm

17. Crazyflie Firmware Architechure Figure (2016). [Online]. Available: https://wiki.bitcraze.io/projects:crazyflie2:architecture:indexCrazyflie

18. Swarm Platform Wiki, (2016). [Online]. Available: http://wikis.ece.iastate.edu/distributed-autonomous-and-networked-control-lab/index.php/Crazyflie_Swarm

19. J. Haughery, "Hardware-Accelerated Machine Vision using Field-Programmable Gate Arrays (FPGA)," presented at *The Association of Technology, Management, and Applied Engineering Conference*, Orlando, FL, 2016.

20. Crazyflie Nano-Quadcopter. [Online]. Available: https://www.bitcraze.io/

21. Hardware-Accelerated Object Tracking Demo Video. [Online]. Available: https://youtu.be/l0zgXhSaA5A

22. Hardware-Accelerated Computer Vision Crazyflie Flight Test 1 Video. [Online]. Available: https://youtu.be/nHF6UnezgLk

23. Hardware-Accelerated Computer Vision Crazyflie Flight Test 2 Video. [Online]. Available: https://youtu.be/BQgCf_04ZQg

24. CrazyRadio PA Specifications (2015). [Online]. Available: https://www.bitcraze.io/crazyradio-pa/

25. B. Landry, "Planning and control for quadrotor flight through cluttered environments," M.E. thesis, Dept. of Elect. Eng. and Comput. Sci., Massachusetts Institute of Technology, 2015.

26. J. Zambreno, P. Jones, “CprE 488 – Embedded Systems Design: Lecture 7 – Embedded Control Systems”, PowerPoint Presentation. (2015) [Online]. Available: http://class.ece.iastate.edu/cpre488/lectures/Lect-07.pdf

27. Wireless communications principles and practices, T. S. Rappaport, 2002, Prentice-Hall