Development of a multi-agent quadrotor research platform with distributed computational capabilities

by

Ian McInerney

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
Phillip Jones
Umesh Vaidya
Fritz Keinert

The student author and the program of study committee are solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

DEDICATION

To my family and friends who have supported and encouraged me throughout my educational career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Dr. Nicola Elia *Professor of Electrical Engineering*

For providing the laboratory space, camera localization system, and the Crazyflies used in this thesis. For providing insight during the controller design and algorithm creation, and for serving as my major professor.

Dr. Xu Ma *PhD Graduate, Electrical Engineering*

For collaborating on the theoretical derivation of the localization algorithm.

Matthew Rich *PhD Candidate, Electrical Engineering*

For providing assistance with the physical modeling of the Crazyflie as well as the controller design.

James Benson *Aerospace Engineering Department Laboratory Staff*

For providing the Logger Pro system used in the parameter identification.

Financial Support

This work was partially supported by NSF grants CNS-1239319 and CCF-1320643.

ABSTRACT

Research on multi-agent systems of UAVs is of growing interest in the research community, with specific interest in the testing of novel algorithms on actual systems. Many existing testbeds already exist, but the majority of them utilize expensive quadrotors for their agents. With the recent surge in interest from consumers, companies have started to market lower cost quadrotor options. One such option is the Crazyflie 2.0 from Bitcraze. This quadrotor measures just 10cm from rotor to rotor, uses open-source firmware, and has developed a strong community backing. This work develops a multi-agent testbed using the Crazyflie 2.0.

This work presents a parameterization of the Crazyflie quadrotor so it can be modeled and have more advanced controllers designed for it. Additionally, this work discusses the default control loop of the Crazyflie 2.0. Then nested-loop PID controllers are designed and compared against the simulated physics model.

A software system that is capable of controlling multiple flying Crazyflie's is also presented. This system is also capable of modifying the controller at runtime, and implementing distributed computation on the Crazyflie.

Finally, a novel algorithm for localization of a target object using distance-only measurements is presented. This algorithm uses optimization dynamics to solve a non-convex QCQP formulation of the problem in a distributed manner. The algorithm is presented and then implemented using the distributed computation framework presented in this work.

CHAPTER 1. INTRODUCTION

This thesis presents a multi-agent research platform based on the Crazyflie 2.0 quadrotor from Bitcraze. This research platform is capable of supporting multiple quadrotors flying at once, and also the ability to perform distributed computations using the quadrotor's CPU.

1.1 Literature review

1.1.1 Quadrotor Modeling

Quadrotors have gained a lot of interest in the research community over the past 5 years, and many papers and dissertations have been published describing the physics models of quadrotors.

In [3], a full featured non-linear physics model is developed. This physics model includes factors such as: the loss of rotor thrust due to airflow over the propellers (in all directions), a center of mass that is not coincident with the origin of the body frame, and many other things. This work also explored the application of the nested loop PID structure on a Gaui quadrotor, along with other more advanced controllers.

Some literature has also been published related to the modeling of the Crazyflie specifically. For instance, in [4] a model for the Crazyflie 1.0 is developed, and also parameters are given for the quadrotor. The Crazyflie 2.0 has also been studied in the existing literature. A model of the Crazyflie using differential flatness is developed in [5]. That work also identified some physical parameters such as rotor constants, moments of inertias and masses. A more rigorous study of the Crazyflie platform was conducted in [6], where moments of inertias are measured, along with mappings of the input command to thrust/torque and aerodynamic drag coefficients for flight through the air.

1.1.2 Multi-Agent Research Platforms

In recent years, there has been a large push to develop systems that allow for experimental verification of mutli-agent swarm strategies and controller architectures. Many universities have developed systems to prototype algorithms in a mutli-agent environment, including the Massachusetts Institute of Technology's RAVEN testbed [7], the University of Pennsylvania's GRASP lab [8, 9], ETH Zurich's Flying Machine Arena [10, 11], the University of Southern California's Crazyswarm [12], and the University of Bologna [13].

The MIT RAVEN testbed [7] supports both ground-based agents and UAV agents, and also allows for new agents to be added or removed depending on the test being performed. This system utilizes a Vicon motion capture system to gather position data of all the agents. This position data is then fed into a master controller to determine the mission plan (such as desired trajectories, desired agent positions, etc.). Those plans are then sent to the agent systems for implementation. In the case of the UAV systems, the actual flying systems are standard RC quadrotors (the Dragonflyer V Ti Pro) receiving their input (for thrust/roll/pitch/yaw commands) over a Pule Position Modulated (PPM) signal. All outer-loop control (such as position) is done off-agent on dedicated processors, with the resulting commands transmitted over the RC link.

The University of Pennsylvania GRASP testbed [8, 9] is designed solely for aerial robotics research. Its main large flight vehicle is the Hummingbird quadrotor from Ascending Technologies, with another custom developed quadrotor used for cases where smaller agents are required (such as large swarms). The position of the agents is sensed using a Vicon motion capture system, which is then connected to the agents using the Robot Operating System (ROS). Each quadrotor is connected to the system using Zigbee transceivers, which send and receive commands and can transmit log data back to the ground. The small quadrotors (described in [9]), take only angular position setpoints as command input and then perform on-board control of angular position and rate. All other trajectory generation and position control is done by a ground-based computer.

ETH Zurich's Flying Machine Arena [10, 11] is a very large indoor flying space equipped with a Vicon motion capture system with a sensed volume measuring 10 meters on each side. This space primarily uses the Ascending Technologies Hummingbird quadrotor as its flight vehicle, however other experimental systems can be tested in it (such as the distributed flight array [14] or an omnidirectional cube [15]). The system is designed similar to the RAVEN and GRASP testbeds discussed earlier, where the agents all communicate with a central network that consists of ground-based control computers and the agents. Those control computers monitor all the agents' states, and communicate with the agents over a non-ack'd communicatons link (so reception of a packet is not guaranteed). These control channels may be standard ethernet, or another industrial wireless channel (such as Zigbee). In this system, the agents are able to provide data to the ground control computers, as well as receive commands from them.

The University of Bologna has developed a multi-agent testbed [13] using the Crazyflie 1.0 quadrotor from Bitcraze. This quadrotor is a small, inexpensive quadrotor that contains an on-board ARM Cortex M3 processor for its primary data processing and control. In this system, the quadrotors communicate with a ground station computer over a Bluetooth radio link developed by Bitcraze. Each Crazyflie gets its own dedicated ground radio (called the CrazyRadio) on the ground computer. The quadrotor only contains two control loops, the angular position and rate loops. The other control loops are contained within MATLAB and Simulink algorithms running on the ground station. The ground station receives position information for each quadrotor from an OptiTrack motion capture system.

The University of Southern California has developed a multi-agent test system, called the Crazyswarm [12], based around the Crazyflie 2.0 from Bitcraze. This is the successor to the Crazyflie 1.0 which was used in the University of Bologna's testbed. It still communicates over a Bluetooth radio link with a ground station computer, but the Crazyswarm system modified the firmware and communications protocol to allow for multiple Crazyflies to run on a single radio. This allows them to run 39 Crazyflies on just 3 radios. In order to accomplish this, they implemented broadcast packets that are not ack'd by the quadrotors. This allows them to transmit the position of up to 2 Crazyflies in every packet. When flying, the swarm is not able to log data back to the ground station, since doing so will cause latency issues and delays

in receiving the position information from the ground station. They have implemented the majority of the controls processing on-board the Crazyflie, allowing for internal control of its position and internal trajectory generation.

The Crazyswarm uses a Vicon camera system to determine the location of the Crazyflie's when flying. They do not use the proprietary tracking software that Vicon sells with the camera system though, and instead have implemented a tracking system based upon the Iterative Closest Point algorithm [16]. This custom algorithm allows for every Crazyflie in their swarm to have the same arrangement of camera system markers, since the physical size of the Crazyflie makes creating a lot of distinct marker arrangements very difficult.

1.2 Existing Infrastructure

The main purpose of this work is to extend the system developed in [1]. The existing system was developed with two different use cases:

- Educational activites such as teaching about system modeling, parameterization and controller design
- Research activites such as experimenting with swarming or multi-agent computational algorithms

1.2.1 Crazyflie

The system was originally developed using the the Crazyflie 2.0 quadrotor as its flight vechicle. This quadrotor is small, lightweight, very durable, and costs around $200 (a more in-depth description of the Crazyflie can be found in section 2.2). The Crazyflies used in this system used largely un-modified firmware, with the only modifications being the addition of some logging signals. In this setup, the Crazyflie was only performing angular position (pitch and roll) and rate stabilization on-board (using internal sensors for those control loops). The x, y, z and yaw controllers were implemented on a ground station computer.

In order to measure the Crazyflie's position, an OptiTrack camera system was utilized. The camera system requires IR reflective trackables to be mounted onto the Crazyflie, as seen in figure 1.1. These trackables were attached using bolts which were then attached to the Crazyflie frame using Velcro. A major problem with this mounting method is that when the Crazyflie would do fast or aggressive maneuvers, the trackables might fall off, causing the camera system to be unable to calculate the Crazyflie's position. The loss of the position data would then cause the Crazyflie to crash.

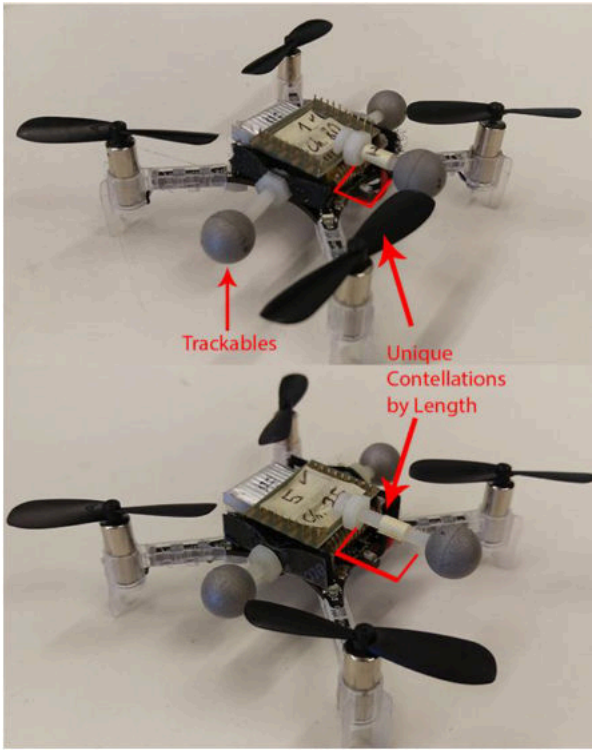


Figure 1.1: Existing Crazyflie hardware and trackable mounting system [1]

1.2.2 Control Software

This system used a single ground computer running Red Hat Linux 6, and a custom developed C++ application for controlling the Crazyflies. The C++ application was developed using the libcflie library by Jan Winkler [17] as the basis for the radio communications. This library is composed of four main C++ classes: CCRTPPacket, CTOC, CCrazyRadio, and CCrazyflie.

The CCRTPPacket class is a class that defines a single Crazy RealTime Packet (CRTP) packet for transmission to or from the Crazyflie. This class allows for the CRTP packet to be easily modified and passed around in the source code. The CTOC class implements the ability for the software to interface with the logging system and the parameter system on the Crazyflie, allowing for log data to be received from the Crazyflie and for certain constants in the firmware to be modified at run-time (through the parameter system). The CCrazyRadio class is a wrapper class for libUSB for handling communication with the CrazyRadio made by Bitcraze. It sets up the USB transactions to send and receive the CRTP packets over the radio dongle, and also allows for the radio parameters (such as channel, power, and datarate) to be modified. The final C++ class in that library is the CCrazyflie class. That class contains all of the functions needed to control the Crazyflie quadrotor, and creates the CRTP packets to send data to the quadrotor. This was designed to be the main class that any external software needs to interface with in order to use the Crazyflie.

The position and yaw data is received from the camera system over an ethernet network in the lab. The data is transmitted using the Virtual Reality Peripheral Network (VRPN) protocol [18]. The VRPN library utilizes a callback system for its data reception handling. Every time through the main loop, there is a VRPN *mainloop()* function that must be called for each Crazyflie. If a new packet with position information has been received, a callback is called by that function.

The software developed in [1] is a single-threaded piece of software that is designed to read in commands from the user (as keyboard input), then translate those into a desired command for the Crazyflie controllers. The software also used the PID controller source code from the Crazyflie firmware to implement 4 PIDs on the ground station computer: an x, y, z, and yaw PID. These PIDs ran inside the VRPN callbacks and would use the command input from the user as a reference, and use the just-received position data to generate the desired attitude commands to send to the Crazyflie. The overall software architecture can be seen in figure 1.2.

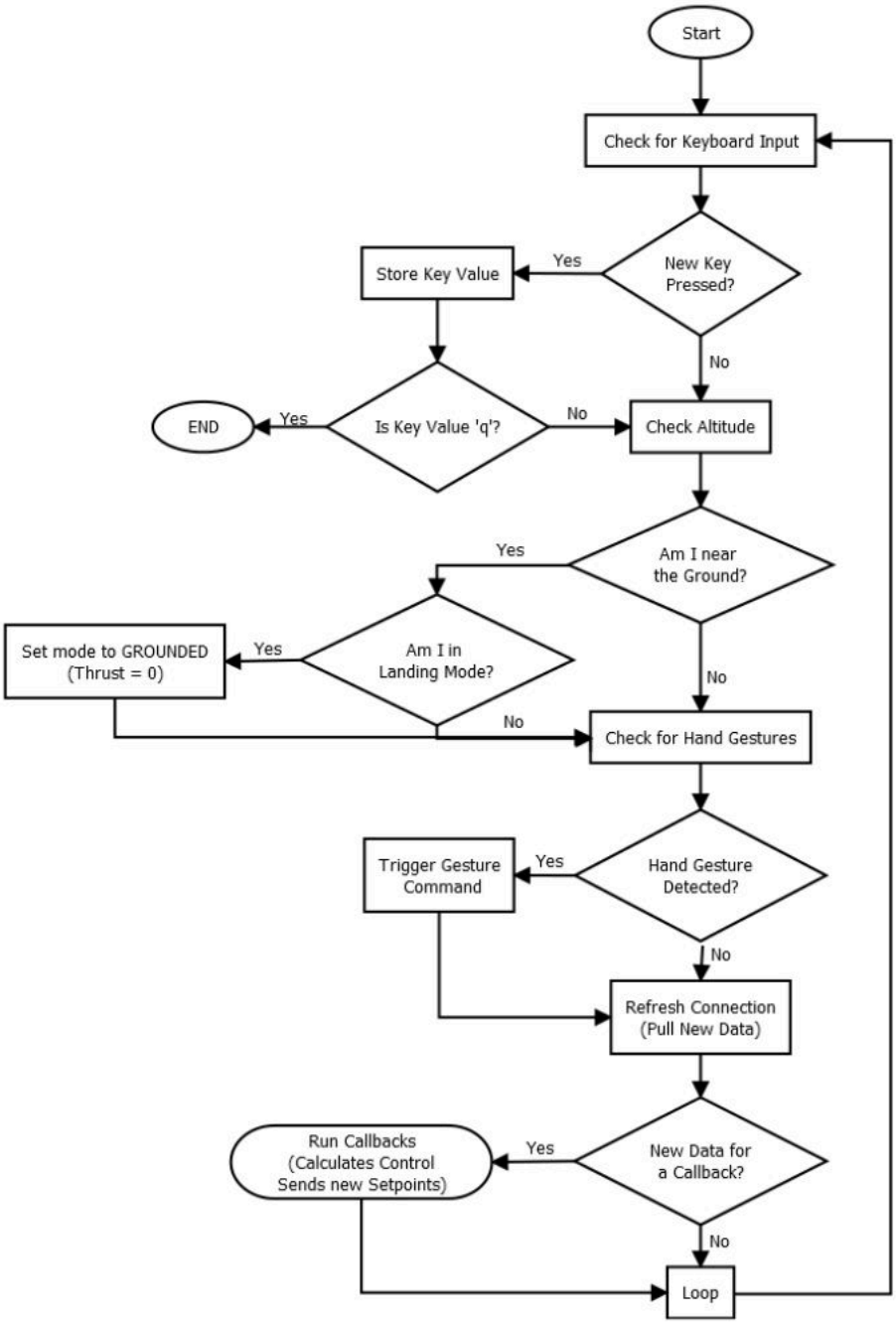


Figure 1.2: Flow of the existing single-threaded software [1]

Since this software is single-threaded, it handles everything in a sequential manner. This means that any delay in computing or communications affects every Crazyflie, and the more Crazyflies that are added the longer the main loop takes and consequently the interval between controller updates increases.

1.3 Contributions

1.3.1 Goal

The goal of this work is to extend the system from [1] to include the following:

- Parameterization of the Crazyflie system to allow for modeling and model-based control design
- Migration of all controllers onto the Crazyflie firmware
- Methods to modify on-board controllers while the quadcopter is running
- Multi-threaded ground station application
- Framework for performing distributed computation using the Crazyflies

This system will then be demonstrated on an application that requires distributed computation capability.

1.4 Novelty

This work presents several novel contributions. First, the parameterization done in this work is more fine-grained than that presented in previous works (such as [4, 5, 6]. The work done here includes not only the physical subsystem modeling, but also the modeling of the existing firmware structure.

Second, this work presents a system that is capable of supporting multiple Crazyflies flying at one time while performing distributed computations. While some larger systems such as GRASP and the Flying Machine Arena could support the distributed computations those systems use larger and more expensive quadrotors. The comparable system would be the CrazySwarm. This system does not contain the ability for distributed computation though, and focuses mainly on emulating swarm behavior.

Third, this work presents a novel algorithm for performing object localization using a multi-agent system with only distance measurements to the target. This method was developed in collaboration with Dr. Xu Ma, a former PhD student in our research group at Iowa State. This system is based on the solution of a non-convex Quadratically Constrained Quadratic Program (QCQP) using optimization dynamics methods. The Crazyflie system is then used to test this method in real life.

1.5 Organization

This work is organized into 9 chapters. Chapter 1 presents the literature review for quadrotor modeling and multi-agent testbeds. In addition it presents the existing system, and outlines the contributions of this work. Chapter 2 presents an overview of the overall system, the Crazyflie hardware platform, and the distributed computation framework that was developed. Chapter 3 presents an introduction to attitude representation and the physics of quadrotors. Chapter 4 presents the new software developed for this work. Chapter 5 presents a discussion on the control flow of the on-board controller for the Crazyflie. Chapter 6 presents the parameterization methods and final parameters for the Crazyflie 2.0. Chapter 7 presents a description of the on-board PID controllers, a comparison between a non-linear simulation and actual flight data, and the design of a linear static-gain state-feedback controller. Chapter 8 presents the object localization problem and the proposed algorithm based upon optimization dynamics. This chapter also presents the test setup and test results from applying the algorithm. Finally chapter 9 summarizes the work and provides future directions.

At the end there are also two appendices discussing in more detail some items from this work. Appendix A discusses the inconsistencies found in the default control loop of the Crazyflie and the control loop from [1]. Appendix B presents the source code of the localization algorithm from chapter 8.

CHAPTER 2. SYSTEM OVERVIEW

The overall system described in this work is composed of three main parts:

1. An OptiTrack camera system for localization
2. The Crazyflie quadrotor
3. A ground station computer for control

2.1 Camera System

To localize the Crazyflie's in the flight area, a camera system composed of 12 OptiTrack VR100:R2 cameras [19] is utilized. These cameras operate by emitting a flash of IR light immediately before capturing a frame. This IR light is reflected off of distinct trackables located on the object being tracked, and the reflected light is then captured by the cameras. This generates a point in the camera's image, and then using the Tracking Tools software from OptiTrack the images from all the cameras are combined into a point cloud in 3D space.

In the Tracking Tools software, groups of three or more trackables (also called a constellation of trackables) can be created, forming a rigid body. The system then tracks the rigid body as it moves through the space, providing information about the rigid body's location in 3-space and its pose (in quaternion format). The position and pose information is then transmitted over a lab-wide network using the Virtual Reality Peripheral Network (VRPN) [18] protocol at a rate of 100Hz.

2.2 Crazyflie Quadrotor Platform

2.2.1 Overview

The Crazyflie is a small quadrotor, termed a nano-quadrotor, that measures 9.5cm from rotor to rotor diagonally, and takes up a horizontal area of 12cm by 12cm when flying. It has one main circuit board that contains all the electronics and motor control hardware, and then 4 legs that attach to the circuit board. These legs provide holders for the motors, and also landing gear to support the quadrotor. A Crazyflie can be seen in figure 2.1.

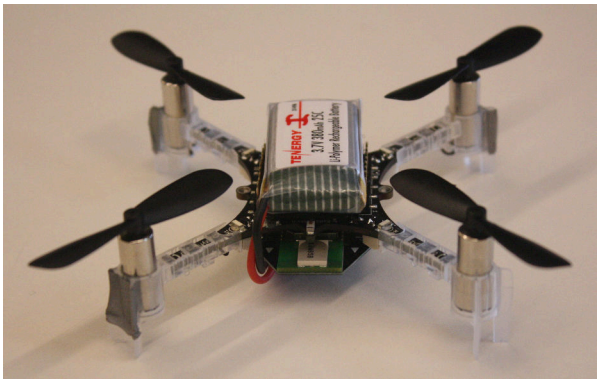


Figure 2.1: Crazyflie quadrotor with Loco Positioning deck and 380mAh battery

The propulsion system is composed of 4 brushed DC motors, each spinning a propeller measuring 4.6cm in diameter to provide the thrust required for flight and maneuvers.

2.2.2 Electronics

The Crazyflie electronics system is composed of two microprocessors: an ST Micro STM32F405 running at 168MHz, and a Nordic Semiconductor nRF51822 running at 16MHz. The STM32F405 is the main processor, performing all of the control and sensing functions for the Crazyflie. The nRF51822 is a secondary processor responsible for performing the radio communications and also power management of the Crazyflie. An overall diagram of the electronics system can be seen in figure 2.2. The Crazyflie uses an Invensense MPU-9250 sensor for its attitude estimation. The MPU-9250 contains 3 different sensors: a 3-axes accelerometer, a 3-axes gyroscope, and a 3-axes magnetometer.

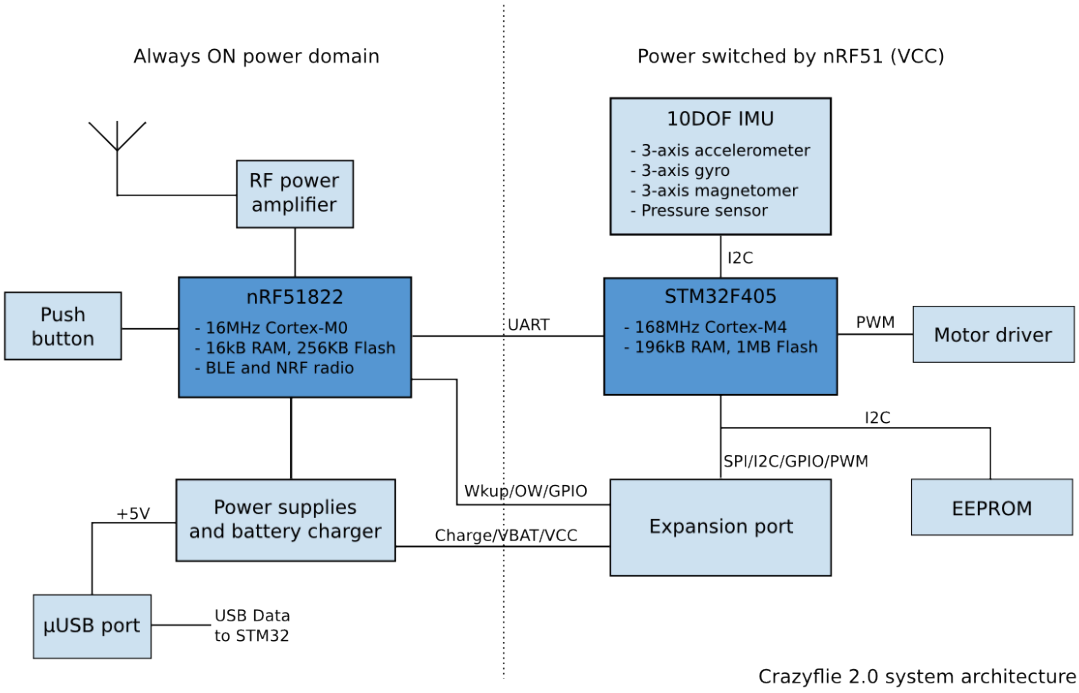


Figure 2.2: Architecture of the Crazyflie electronics [2]

The Crazyflie receives its control inputs over a 2.4GHz Bluetooth link. This link is handled by the nRF51822 chip on the Crazyflie, and interfaces to a computer through a CrazyRadio dongle (which is a 2.4GHz to USB bridge developed by Bitcraze).

2.2.3 Firmware

The Crazyflie operates using firmware[1] developed with the C programming language, and FreeRTOS (an open-source real-time operating system [20])[2]. In the firmware, each main part is contained within its own task. This means that each part has its own function running in an infinite loop, and then the FreeRTOS system periodically interrupts the task to allow for the others to execute.

[1]The description of the firmware in this section is a general overview. Details of specific changes made to the firmware will be discussed in later sections.

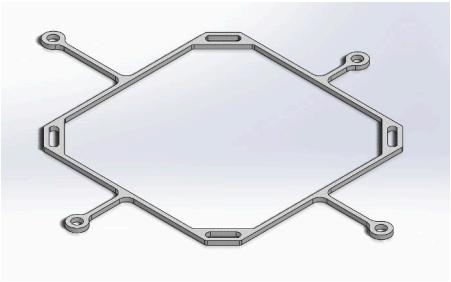
[2]In this work, only the firmware running on the STM32F405 chip was modified and analyzed, the firmware running on the nRF51822 was not examined.

2.2.4 Trackable Mounting

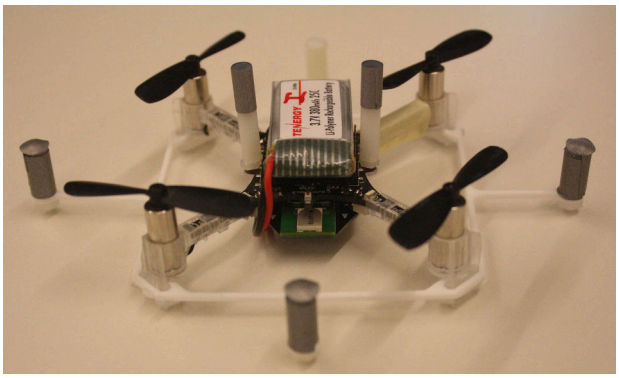
In order for the Optitrack camera system to locate each Crazyflie, they must be outfitted with at least three IR reflective markers to create the rigid body. Additionally, the Tracking Tools software requires that each rigid body have a distinct trackable constellation to guarantee reliable tracking. If the constellation is not unique between two bodies, the software may confuse the bodies and provide inaccurate data.

The existing system used pre-made IR trackable orbs that can be purchased directly from OptiTrack. Three of these orbs were mounted at varying positions on the Crazyflie to create the constellation, as seen in figure 1.1. The mounting system utilized bolts attached to the Crazyflie with Velcro as the primary method of attaching the orbs. The issue is, when the Crazyflie would perform aggressive maneuvers (such as a fast takeoff), the velcro adhesive would fail and the orb would fall off. This meant the constellation was no longer trackable, and no position data was available.

To overcome this issue, a frame for the trackables was used. This frame is based off of a design created in the MIT CSAIL lab for holding their constellation during flight [5, 21]. A CAD rendering of the frame used can be seen in figure 2.3a.



(a) CAD rendering of frame



(b) Crazyflie with trackables attached

Figure 2.3: Trackable mounting system for the Crazyflie quadrotor

Additionally, the use of the trackable orbs on the Crazyflie posed issues with increasing the mass and moving the center of mass away from the origin of the principal body axes. This is because each trackable orb weighed approximately 1 gram, and the support hardware for each orb was another 1-2 grams. This meant that approximately 10 grams of mass was added to

the Crazyflie by the orbs, mounting hardware, and other mass used to re-center the center of mass. This additional mass caused a reduction in flight-time and thrust available for performing maneuvers.

Instead, the trackable orbs were replaced by custom trackables fashioned out of nylon standoffs and IR reflective tape. One standoff and its mounting hardware had a mass of 0.5 grams. These standoffs could be placed at 6 different locations on the Crazyflie: four on the frame and two on the main board. An arrangement of these standoff-based trackables on a Crazyflie can be seen in figure 2.3b. Overall, the Crazyflie quadrotor with a 380mAh battery, and the new trackables had a mass of 36 grams.

2.3 Ground Station

The third main part of this system is the ground station computer. This computer is the interface between the lab-wide VRPN network and the CrazyRadio dongle, translating the position information into the CRTP packets. Additionally, it is where the user enters commands for the Crazyflie, and acts as a network router for the distributed computation feature.

The software has been extended from [1], and was designed using C++ to run on Red Hat Linux 6/7 or Fedora 23. More details about the ground station is in section 4.1.

2.4 Distributed Computation

2.4.1 Ideal System

The ideal system for performing distributed computation on the Crazyflies would have the following features:

- Mesh-network structure (agent-to-agent communication)
- Ability for each agent to run different algorithms
- Expandable to large number of agents
- Interaction between algorithm and control loop

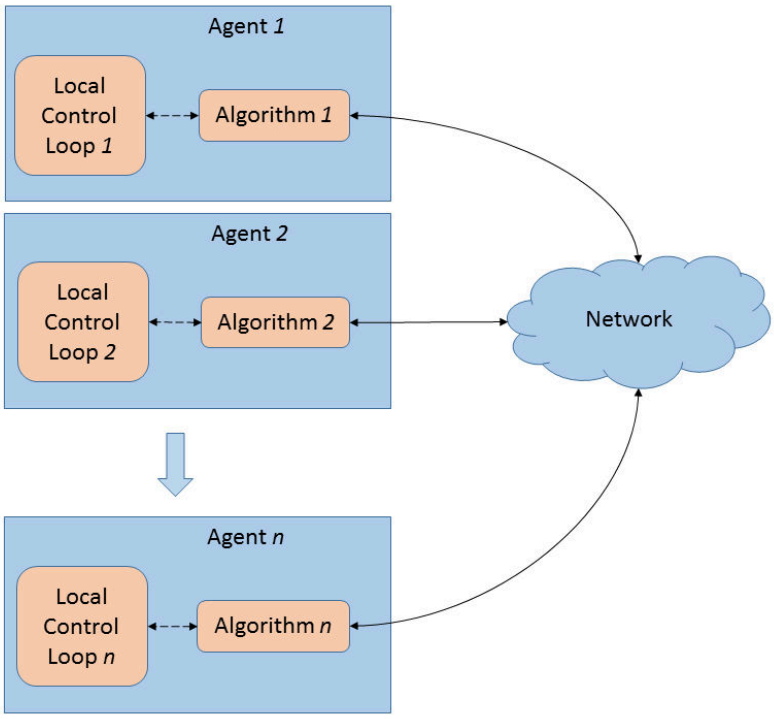


Figure 2.4: Ideal distributed computation system

2.4.2 Realized System

In reality, the ideal system is not realizable on the current Crazyflie hardware due to limitations in its communications system. Instead, the systen shown in figure 2.5 was implemented.

The Crazyflie communications system is currently designed to talk directly with a computer using the CrazyRadio dongle and the nRF51 chip. When this system was designed, the computer was thought of as the originator for all important communications, and the Crazyflie would only provide brief status updates. This means that the communications stack was implemented such that the Crazyflie cannot initiate communications, it can only return data as a response to a data packet from the computer.

This means that the Crazyflie's cannot natively have a mesh network capability, all communications must go through a host computer. This also limits the number of agents on a network to a small number, directly related to the number of CrazyRadio dongles available on the host computer. For example, testing with the computation example in chapter 8 (with 100

computations per second) shows that when more than two Crazyflie's are placed on a single radio when the algorithm runs, then the data update rate for the control loop can slow down and cause unstable flight.

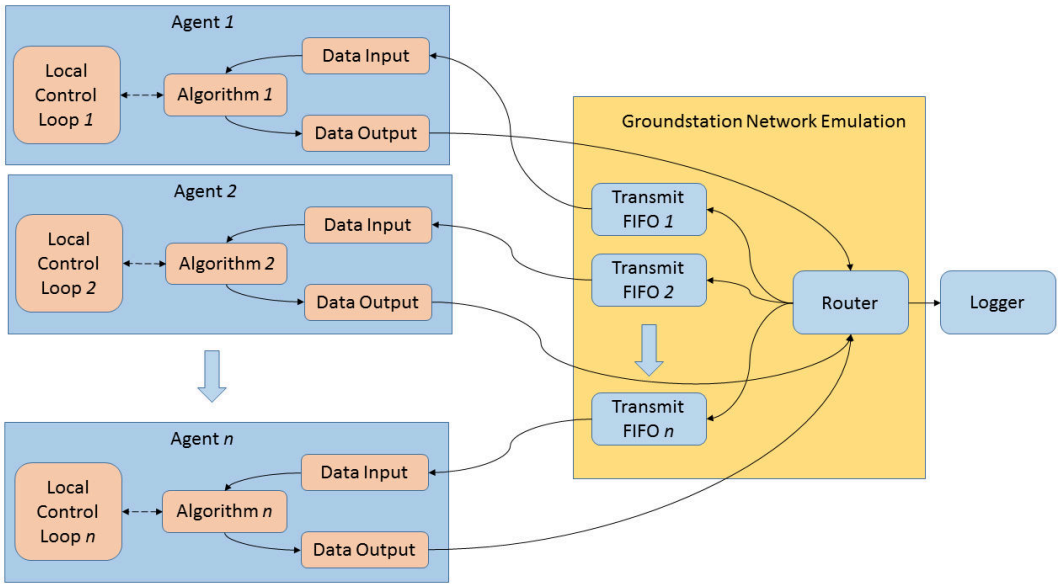


Figure 2.5: Realized distributed computation system

The system is broken up into two distinct components: Ground Station Network Emulation and the Agent.

2.4.2.1 Ground Station Network Emulation

To overcome the communications limitation, this system utilizes a simulated network topology, run as a packet router on the ground station computer. This router can be loaded with an arbitrary network structure, which can be updated at run-time. The router treats all communications links as directed links, so both undirected and directed communications topologies can be implemented on this system.

Since this network coexists with the existing communications infrastructure of the Crazyflie, it relies on constant packet transmissions to the Crazyflie from the ground station (so the Crazyflie can reply with the algorithm output). When the ground station receives an algorithm

output packet, it is sent to the network router. The router examines the from address, and looks up all the agents that are to receive the packet. The packet is then placed into the FIFO for each receiving agent, where it is then sent to the agent at the next opportunity.

2.4.2.2 Agent

The agent component is the driving piece behind the computational network. Each agent is responsible for executing its algorithm's computations at the desired interval, and then transmitting the result over the network when desired. This means that the computation on the agent will occur at the desired rate, independent of data reception.

A key component of the agent's algorithm structure is the tie-in between the control loop and the algorithm computation. Inside each control loop iteration there is a call to the algorithm component. This call does two things:

1. Pass the current state vector to the algorithm
2. Update the control setpoints with values from the algorithm

This interconnection allows for the algorithm to use current state data (such as position, velocity, etc) in its computations, and then also to modify the setpoint of the control loops based on the computations (such as the current linear position).

CHAPTER 3. QUADROTOR MODELING PRELIMIARIES

In order to understand some of the aspects of this work, it is important to understand some of the basic fundamentals of the physics of quadrotors. This chapter does not provide a very in-depth description of the physics model of the quadrotor. For an in-depth discussion, the reader is directed to [3] and the references therein.

3.1 Representation of Axis Rotations

As an aerial vehicle, a quadrotor has 6 degrees of freedom - movement in $\mathbb{R}^3$ (x, y and z) and movement in the angular space (roll, pitch and yaw). Representing a system with just translational movement in 3D space is straightforward, since the only change will be in the origin location. This means that a simple addition/subtraction can be done to move the quadrotor's body frame into aligment with the inertial frame.

However, since the quadrotor can move in the angular space as well, the mapping of the body frame into the inertial frame must be done using a rotation matrix. Rotation matrices are special matrices belonging to the Special Orthogonal Group 3 ($\mathbf{SO}(3)$). Matrices in the $\mathbf{SO}(3)$ group have the property of being orthogonal, ie $A^T = A^{-1}$, and also are related to the rotation of a coordinate system in $\mathbb{R}^3$, namely a matrix in $\mathbf{SO}(3)$ can be used to represent any arbitrary rotation in $\mathbb{R}^3$. The matrix representing the rotation is referred to as a rotation matrix.

3.1.1 Euler Angles

Rotation in $\mathbb{R}^3$ can be thought of as three separate rotations, applied sequentially to the coordinate system. These three rotations are commonly referred to as Euler angles. Each of the three rotations will bring a specific plane of the body frame into alignment with the inertial frame. Since the rotations are applied sequentially, the order in which the rotation matrices are formed is important, in fact there are 12 different sequences of rotation in Euler angles.

For this work, the sequence Yaw -> Pitch -> Roll, ($\psi \rightarrow \theta \rightarrow \phi$) is used. This rotation is applied in 4 steps:

1. Translation - The translation is applied to move the inertial frame origin onto the body frame origin. This produces the intermediate frame $E' = [\, e'_x \; e'_y \; e'_z \,]^T$
2. Yaw rotation - The yaw rotation is applied to rotate E' around e'_z by ψ forming another intermediate frame $E'' = [\, e''_x \; e''_y \; e''_z \,]^T$
3. Pitch rotation - The pitch rotation is applied to rotate E'' around e''_y by θ forming another intermediate frame $E''' = [\, e'''_x \; e'''_y \; e'''_z \,]^T$
4. Roll rotation - The roll rotation is applied to rotate E''' around e'''_x by ϕ forming the final body frame

Each rotation can be expressed as a vector-matrix product, ie.

$$E'' = E' \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} = E' R_1$$

$$E''' = E'' \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} = E'' R_2$$

$$B = E''' \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} = E''' R_3$$

Which can then be further expressed as the product of those rotations: $B = E'R_1R_2R_3$. The matrix product $L_{EB} = R_1R_2R_3$ is then the rotation matrix to move the inertial frame into the body frame, ie $B = E'L_{EB}$. To move from the body frame to the inertial frame, simply do $L_{BE} = L_{EB}^{-1}$, which since L_{EB} is in $\mathbf{SO}(3)$, $L_{BE} = L_{EB}^T$. Meaning to go from body frame to inertial frame do: $E' = L_{EB}B$.

3.1.2 Quaternions

Euler angles are not the only way of representing rotations though, a method called quaternions can also be used. The quaternions are a set of generalized complex numbers with certain conditions, but the details of quaternions are not needed to understand this work so they are omitted (for more on the quaternions, see [22]). In the quaternion system, there are 4 numbers used to represent the rotation, ie $q = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T$. These numbers have the additional constraint that $\|q\|^2 = 1$, ie. $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$.

The quaternions can then be used to represent rotations through the rotation matrix

$$B = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_0q_2 + q_1q_3) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} E'$$

Additionally, the quaternion representation of rotations can be converted into the Euler angle representation (following the sequence discussed above) using:

$$\begin{aligned} \theta &= \sin^{-1}\left(-2(q_1q_3 - q_0q_2)\right) \\ \phi &= \tan^{-1}\left(\frac{2(q_0q_1 + q_2q_3)}{q_0^2 - q_1^2 - q_2^2 + q_3^2}\right) \\ \psi &= \tan^{-1}\left(\frac{2(q_0q_3 + q_1q_2)}{q_0^2 + q_1^2 - q_2^2 - q_3^2}\right) \end{aligned}$$

3.1.3 Euler Rates

The rate of change of the Euler angles is not the same as the body angle rate of change. Instead, a rotation matrix must be applied to the body rates to find the Euler rates. In the sequence used in this work, the ϕ rate is equal to the angular velocity around b_x, meaning $p = \dot{\phi}$. The remaining rates must have the above rotation matrices applied to them in sequence, ie

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = R_3 R_2 \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + R_3 \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{3.1}$$

Grouping the matrices into a single transformation produces

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \sin(\phi)\cos(\theta) \\ 0 & -\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = A_{BE} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{3.2}$$

Taking the inverse of A_{BE} will then convert the body rates into the euler rates, ie:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)/\cos(\theta) & \cos(\phi)/\cos(\theta) \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} = A_{EB} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3.3}$$

3.2 Rigid Body Dynamics

The rigid body dynamics of the quadrotor are very complex, and consist of 12 different states:

$$\Lambda = \begin{bmatrix} u & v & w & p & q & r & x & y & z & \phi & \theta & \psi \end{bmatrix}^T \tag{3.4}$$

For the non-linear rigid-body dynamics, only six of these states are needed, and can be grouped into the following two subgroups:
Translational velocities in the body frame:

$${}^B v_o = \begin{bmatrix} u & v & w \end{bmatrix}$$

Angular velocities in the body frame:

$$^{B}\Omega_o = \begin{bmatrix} p & q & r \end{bmatrix}$$

Then, the non-linear dynamics can be represented by:

$$\begin{bmatrix} ^{B}\dot{v}_o \\ ^{B}\dot{\Omega} \end{bmatrix} = \begin{bmatrix} mI & -m[^{B}r_{oc}] \\ 0 & J \end{bmatrix}^{-1} \begin{bmatrix} ^{B}F - {}^{B}\Omega \times m(^{B}v_o + {}^{B}\Omega \times {}^{B}r_{oc}) \\ ^{B}Q - {}^{B}\Omega \times J^{B}\Omega - {}^{B}r_{oc} \times {}^{B}F \end{bmatrix} \tag{3.5}$$

Where I is the identity matrix, m is the mass, J is the moment of inertia matrix (about the body axes), $^{B}r_{oc}$ is the vector from the body origin to the center of mass, ^{B}Q are the input torques in the body frame, and ^{B}F are the input forces in the body frame

By then assuming that the center of mass is located at the origin of the body axes (ie. $^{B}r_{oc} = 0$) and the moment of inertia J is diagonal, the state space equation is

$$\begin{bmatrix} ^{B}\dot{v}_o \\ ^{B}\dot{\Omega} \\ ^{E}\dot{r}_0 \\ \dot{\Theta} \end{bmatrix} = \begin{bmatrix} \frac{1}{m}{}^{B}F - {}^{B}\Omega \times {}^{B}v_o \\ J^{-1B}Q - J^{-1B}\Omega \times J^{B}\Omega \\ L_{EB}{}^{B}v_o \\ A_{EB}{}^{B}\Omega \end{bmatrix} \tag{3.6}$$

Where Θ are the Euler angles and r_o is the earth frame position.

3.3 Powertrain

The powertrain of the quadrotor consists of three distinct pieces:

1. Motor driver (Electronic Speed Controller)
2. Motor
3. Rotor

Where the input to the powertrain system is the command to send to the motor driver/ESC and the output of the powertrain are forces and torques on the body frame of the quadrotor.

Customarily, the first two pieces are lumped together into one piece, ie. the ESC is lumped together with the motor, and then the third is modeled separately for some constants but is included for others.

3.3.1 Rotor Thrust and Drag Torque

The rotor produces thrust and drag when it spins. The thrust can be modeled as a quadratic relation versus rotor speed, ie

$$|T| = K_T \omega^2$$

Where the rotor speed is ω, the thrust produced is T and the thrust constant is K_T.

The drag torque of the rotor can be modeled as a quadratic relation versus rotor speed, ie

$$Q = -K_d \omega^2 \Gamma_i$$

Where Q is the torque produced, K_d is the rotor drag torque constant, and Γ_i is the unit vector for rotor i giving its direction of rotation.

3.3.2 Motor Velocity Dynamics

The angular velocity of the motor shaft (which is also the angular velocity of the rotor), is a dynamical system with the differential equation

$$J_r \dot{\omega} = \frac{1}{R_m K_Q} u_p V_b - \frac{1}{R_m K_Q K_V} \omega - \frac{1}{K_Q} i_f - K_d \omega^2$$

Where J_r is the moment of inertia of the rotor and motor system, R_m is the resistance of the motor windings, K_Q is the motor torque constant, K_V is the motor back-emf constant, i_f is the motor no-load current, V_b is the battery voltage, and K_d is the rotor drag torque constant from above.

If the motor transient is not required (such as for calculating steady-state velocities), the above system can be placed at equilbrium

$$0 = \frac{1}{R_m K_Q} u_p V_b - \frac{1}{R_m K_Q K_V} \omega - \frac{1}{K_Q} i_f - K_d \omega^2$$

Then, for the rotor steady-state velocity given all other parameters, solve the above quadratic equation to find:

$$\omega = \frac{-1 + \sqrt{1 - 4 R_m K_V K_Q K_d (K_V R_m i_f - K_V u_p V_b)}}{2 R_m K_V K_Q K_d}$$

CHAPTER 4. SOFTWARE SYSTEMS

4.1 Groundstation Client

The groundstation client developed in this work is based upon the initial work in [1], which is in turn based on the *libcflie* developed by Jan Winkler [17]

4.1.1 Overall Structure

The overall structure of this client software is a multi-threaded application with 4 main thread types: User input, User output, VRPN processing, and CrazyRadio.

The user input thread is responsible for monitoring the keyboard for typed commands, interpreting the commands received, and then forwarding them onto the appropriate Crazyflie for the actuation. This thread just passes the commands from the user into a command queue in the Crazyflie for the final parsing and actuation, no actual setpoints or modifications to the controller happen in this thread.

The user output thread is responsible for updating the terminal display with pertinent information about each Crazyflie's flight status. This information includes the current status (Grounded, Takeoff, Hover, Landing, etc), the current setpoints, the current position, and information about the Crazyflie's attitude (both from the camera system and from the quadrotor itself). This display can also be modified to include other information from the Crazyflie, such as sensor readings, computation values, etc.

The VRPN processing thread is responsible for constantly calling the VRPN *mainloop* functions for the main connection and for each trackable. From these function the VRPN library will process the packets received from the server by calling trackable callbacks embedded in the

Crazyflie C++ class. These callbacks do not do any computation work, they simply update data structures in the Crazyflie class (unlike the callbacks in [1] which actually performed the control function as well).

The final thread type is the CrazyRadio thread. This thread is the workhorse of the software, where all of the control decisions are made and all of the communications take place. More information about this thread can be found in section 4.1.3.

4.1.2 Startup Routine

When the client software first starts, it runs as a single-threaded application. This application will first prepare various subsystems of the client, such as the computation network, computation logger, VRPN callbacks, and then move onto initializing each individual Crazyflie one at a time. Initialization of the Crazyflie includes setting of the controller parameters, setting of the computation parameters, reading the logging and parameters tables, and starting the desired logging blocks.

During each stage of the startup, output is displayed to the user so they can verify the operation and data being sent (to ensure the software is sending the appropriate controller values, configuring the network properly, etc.). Sample output from the startup routine can be seen in figure 4.1.

Once the startup routine is over, the program pauses so the user can inspect the startup and the Crazyflies before running the actual control software. Once the user continues the program, all the other threads are spawned.

4.1.3 CrazyRadio Interfaces

The CrazyRadio class contains all of the functions necessary to interface with a CrazyRadio and send packets to a Crazyflie. Additionally, this class contains the main loop for a thread that handles all interfacing with the radio, and controlling the Crazyflies associated with the radio.

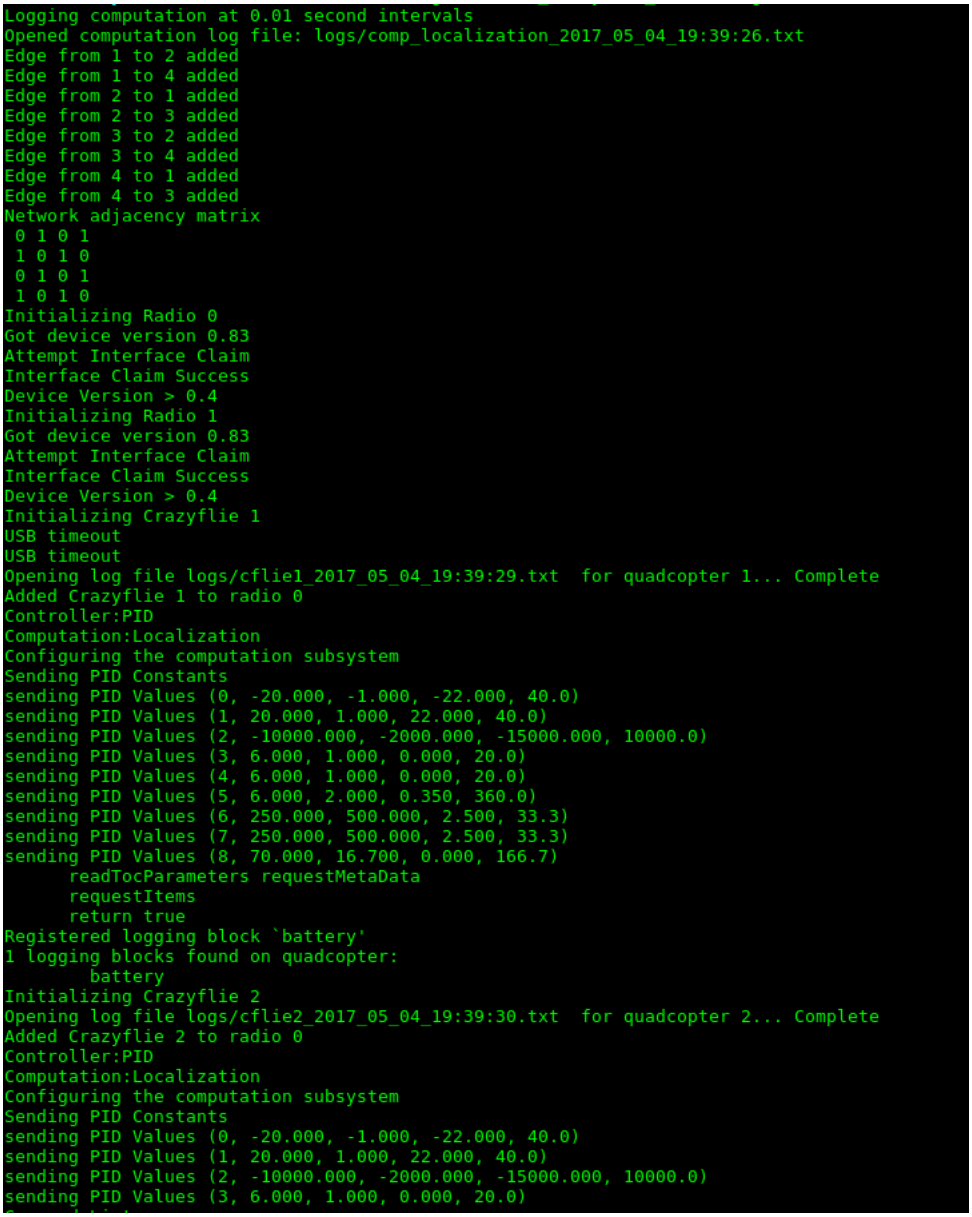


Figure 4.1: Terminal display during client startup

During the startup routine, the Crazyflies are associated with a CrazyRadio. That process passes the Crazyflie object to the CrazyRadio so the radio then can access the Crazyflie. Once startup is complete and the radio thread has been spawned, the thread operates in an infinite loop performing the following actions for each associated Crazyflie:

1. Set radio channel and datarate for the Crazyflie
2. Call the Crazyflie's *cycle()* function
3. Call the Crazyflie's *logger()* function
4. Move to next Crazyflie associated with the radio

4.1.4 Crazyflie Interfaces

The main workhorse of the software is the Crazyflie class. This class contains all of the functions required for packetizing the data to send to the radio, parsing the user commands into setpoints, and handling the logging of data.

4.1.4.1 Main cycle function

The main function of the class is a *cycle()* function. This function follows the flow given in figure 4.2, where first it handles the cases where the Crazyflie is taking off or landing, then sends built-up packets from the network and controller update. Once those are sent, it parses the commands from the user into setpoint changes. Then it will send the new setpoints if there are any (the setpoints are not repeated, they are only sent once), and then it will send position data if there is any new data. If there is no new position data, the loop sends a *dummy* packet. This packet contains no data, but gives a chance for the Crazyflie to respond with any accumulated packets in its transmit queue (so the Crazyflie can send back at least one packet per cycle).

Along with the *cycle()* function there are many different support functions with roles such as:

- Get/Set position data
- Get/Set setpoints
- Set controller parameters

- Configure on-board computation system
- Modify the local coordinate system

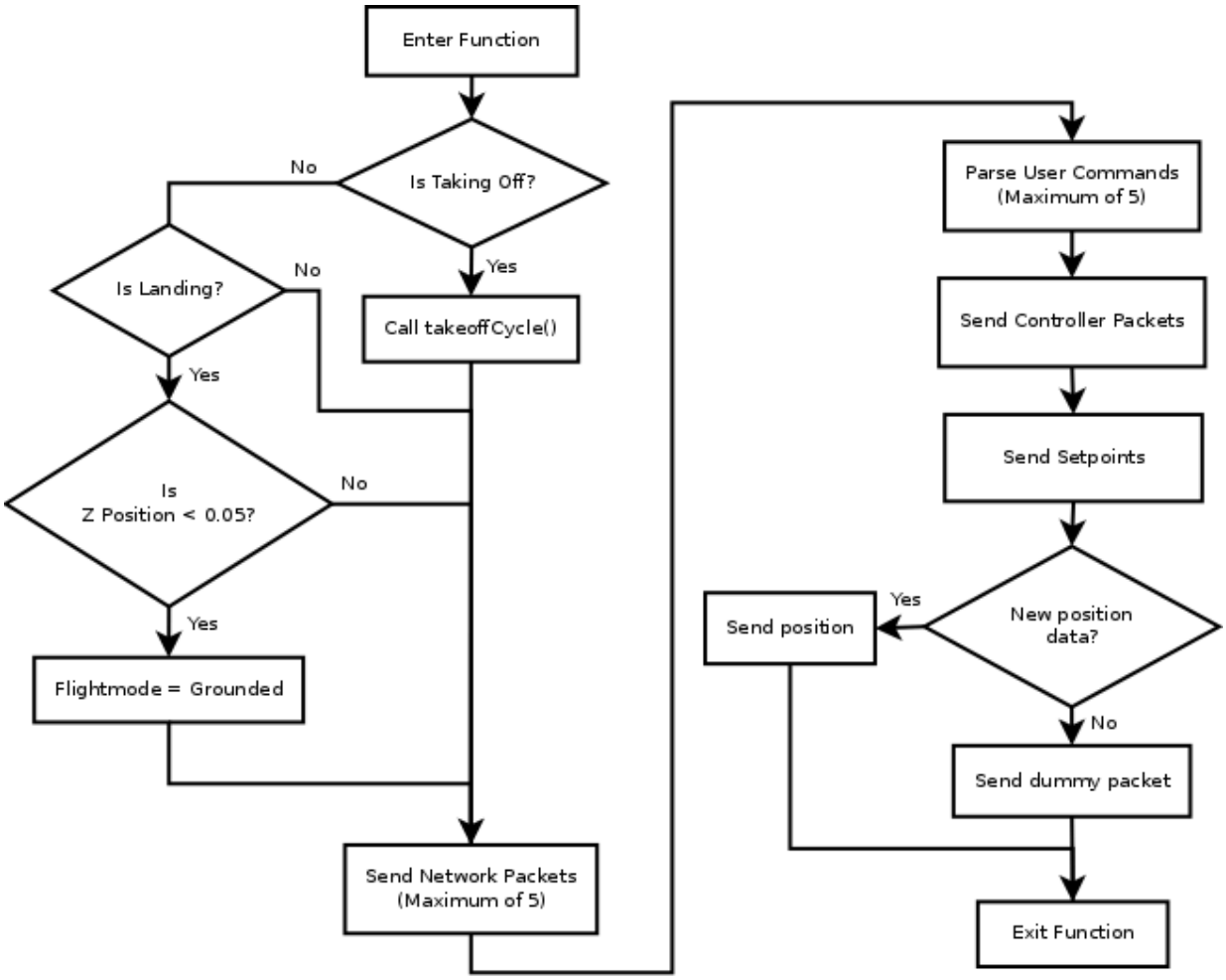


Figure 4.2: Flow of the Crazyflie *cycle()* function

4.1.4.2 Takeoff Routine

This software is also responsible for managing the takeoff routine of the Crazyflies. In this system, the z axis controllers are switched off during the takeoff routine, so from the initiation of takeoff to the end of the final step there is no control of the thrust input. The takeoff routine is then defined by different steps of base thrust values, which are switched as different conditions occur. There are two possible conditions for switching the thrust:

- Position - Change if the z position reaches a threshold value
- Time - Change after a certain number of seconds elapsed since the last change

For example, the normal takeoff routine used is shown in table 4.1.

Table 4.1: Sample takeoff routine

Base Thrust	**Condition for next step**
19000	1 second elapsed
54000	z position < -0.3m
50000	Final thrust value

4.1.4.3 Local Coordinate System

The Crazyflie class contains the ability for the Crazyflie to be translated into a local coordinate system instead of the global camera coordinate system. This process is transparent to the Crazyflie itself, since the translation is handled in the ground station client software, so the Crazyflie will continue executing the controllers as normal.

When the Crazyflie is first switched into the local coordinate system, both the setpoints and the position are translated, so the Crazyflie will maintain the same position. Any subsequent setpoint commands will be relative to the local coordinate system then, and all received camera system data will be translated into the local coordinate frame upon reception. When the Crazyflie is switched out of the local system back to the global system, the setpoint is again translated, so there is no large change in position during the coordinate system change.

4.1.4.4 Callback Functions

The Crazyflie class contains two callbacks: a VRPN packet callback, and a CRTP packet callback. The VRPN packet callback is associated with the VRPN tracker used in the VRPN thread, so this function is called whenever a new position is received.

The CRTP packet callback is called whenever the radio receives a packet from a Crazyflie. This callback function examines the packet's port information and routes it appropriately. Currently only 3 ports are utilized in the callback, given in table 4.2.

Table 4.2: CRTP ports utilized in the *libcflie* callback

Port	Routing
Logging	Pass packet into the logging subfunction to update internal variables
Console	Write console data to file
Computation	Pass packet into the network router

4.1.4.5 CRTP Communications

The Crazyflie/CrazyRadio class has two modes for transmitting packets over the radio link:

- Send and Receive
- Send only

The names for these two modes are slightly misleading, since technically the Crazyflie is always sending packets back to the ground if a packet is transmitted to it. Instead, these modes refer to whether the software should expect a response packet to the exact command sent. This response packet is sent back with the same port and channel information as the sent packet.

If a packet is sent requesting a response (ie. the send and receive mode), then the radio will continue to send packets to the Crazyflie until it receives a response with the desired port and channel. While the radio is processing a Send and Receive mode packet, the radio is blocking all other transmissions on it. This is the equivalent of a TCP protocol on the radio stream.

The other mode is the send only mode. In this mode the packet will only be sent to the Crazyflie once, and only one response will be received. This mode is designed for faster transmission of data and is the equivalent of a UDP protocol on the radio stream.

In the software, all position and setpoint commands use the send only mode, along with all computation packets. The controller update packets utilize the send and receive mode though, so those commands should not be sent during flight (otherwise the radio link may not send position updates as required).

A block diagram of the CRTP packet's journey through the software is in figure 4.3.

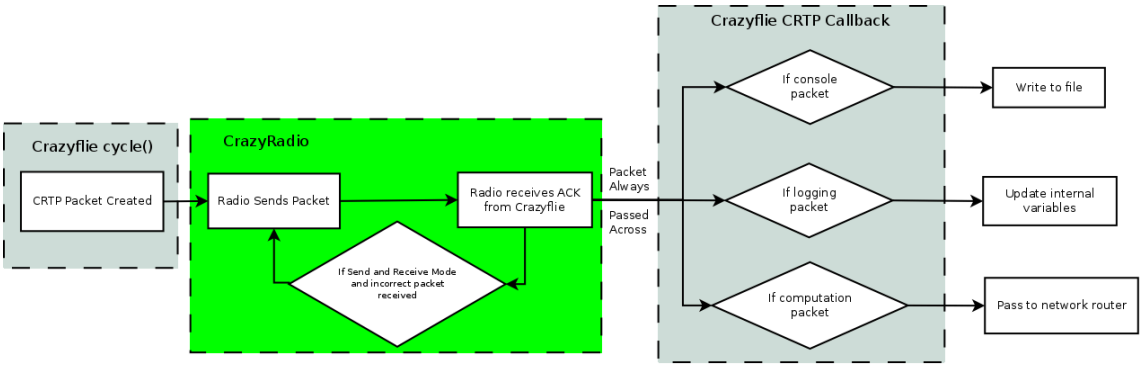


Figure 4.3: Progression of a CRTP packet through the software

4.1.5 Computation Network Emulation

The last subsystem of the ground station client is the emulation of a distributed network for the computational system on the Crazyflies. This part is designed to route the computational data packets from one agent to its neighbors, over an arbitrary network topology. This system can handle both bidirectional and directional communication links, since the way the neighbors are specified is through defining the edges as a to/from relation.

At system startup, the network graph is initialized by defining the number of total agents in the network, and then entering each edge individually into the network. This network object is then passed into the Crazyflie object, where it is used inside the CRTP packet callback. When a computation packet is received in the CRTP callback, the packetReceived function of the network is called. This function will go through the neighbors of the agent, and add the received packet to their sending queue. Then the packet is sent to a logger instance, so all the computational traffic can be logged for offline analysis.

During the *cycle()* function in the Crazyflie object, the Crazyflie calls the network system with its agent number and receives the next packet waiting to be transmitted. This system is designed around a First-In First-Out (FIFO) buffer, so the oldest packet in the queue is the next one sent.

4.2 Crazyflie Firmware

In this work, three major features were added to the Crazyflie firmware:

- On-board position control
- The ability to update controller parameters without reflashing the firmware
- The ability to perform distributed computation with multiple Crazyflies

4.2.1 On-Board Position Control

One of the changes made to the Crazyflie firmware was to move the position controller on-board. In the original system from [1], the position controller was on the ground station computer with the radio link sending the pitch/roll attitude commands and the yaw rate commands. In this work, the commander subsystem of the Crazyflie was modified to receive two new datatypes:

- Position Data - 4 floating point numbers representing the current position of the Crazyflie in the flight volume (x, y, z and yaw)
- Position Setpoints - 4 floating point numbers representing the desired position of the Crazyflie in the flight volume (x, y, z and yaw)

Additionally, inside the setpoints packet there were extra fields to include the desired base thrust value and also a flag to reset the controllers (to allow for clearing of integrators).

This system also included a watchdog on the position data packet, so if a new data packet was not received within 2 seconds of the previous one, the controllers will be disabled and the Crazyflie will fall to the ground. This was done so that if the Crazyflie flew out of range of the camera system, it would automatically stop.

The actual position controllers use the same PID controller functions as the on-board attitude and rate controllers, but have their own variables for storing the gains and setpoints. These controllers were added into the firmware, creating the nested loop shown in figure 7.2.

4.2.2 Controller Update

Another major change in the Crazyflie firmware was the addition of a CRTP packet port to facilitate the modification of the controllers on-line. This port contains three different channels, as given in table 4.3.

Table 4.3: Channels contained on the controller update CRTP port

Channel #	Operation
0	Query controller type
1	Call controller update function
2	Modify the thrust controller

For channel 0, the Crazyflie will respond with an integer value providing the currently active controller. This allows for the ground station to determine if there is a PID controller running, a state feedback controller running, or if the controller is bypassed. This allows the ground station to make decisions about setting the controller (such as not allowing PID updates if the controller is not a PID type).

For channel 1, the Crazyflie will pass the update packet to a controller specific parsing function. This is usually used for setting controller gains, which will have different formats depending upon the controller.

Channel 2 allows for the ground station to enable and disable the thrust controller. When the thrust controller is disabled, the mixer's u_T input is being fed directly from the base thrust command in the setpoint packet, with no modification. When the controller is enabled, the u_T input has the thrust controller output added onto it. This command is usually used during the takeoff routine to turn on/off the height controller.

Each controller is responsible for implementing the controller specific parsing function as it wants, and if desired can even contain a sub-op code if the controller wants to be able to have multiple kinds of update commands specific to it.

4.2.3 Computation System

The last major change to the Crazyflie is the addition of the ability for the Crazyflie to operate as an agent in a network with distributed computation (computation run on the agents). To accomplish this, a new CRTP port was added that has 3 channels, given in table 4.4.

Table 4.4: Channels contained on the computation system CRTP port

Channel #	Operation
0	Query computation type
1	Call computation data received function
2	Call the computation configure function

Channel 0 responds with an integer value representing the kind of computation installed on the Crazyflie.

Channel 1 is the main channel used in the computation block, since it is the channel that the node data will be sent and received on. Data sent over this channel will be forwarded by the ground station network forwarder in section 4.1.5 to all the neighboring quadrotors. The Crazyflie will receive the data from the forwarder over this channel, and then save that data into its memory for the computation to use in the future.

Channel 2 allows for the ground station to update parameters of the computation. This channel makes use of an additional operation code in the first byte of data. The possible operation codes are given in table 4.5.

Table 4.5: Opcodes for the computation update packets

Opcode	Operation
0	Enable/Disable the computation
1	Call computation update function
2	Modify the rate of the computation

Operation code 0 allows for the computation to be disabled and enabled. Operation code 1 allows for specifics of the installed computation to be modified. For instance, the computation discussed in chapter 8 has multiple parameters such as the step size, node number, and other

parameters that are set through this operation code. While operation code 2 allows for the rate at which the computation runs to be modified. By default, the computations run at 100Hz, but they can be modified to run up to 1000Hz (if the computation can execute that quickly).

The actual computation process runs in its own task on the Crazyflie. This allows for the computation to be done independently of the control loop and not interfere with its operation. The computation task does have hooks in the control loop though, allowing the computation task to read the current quadrotor state (such as position, velocity, pose, etc), and also read and modify the setpoints of the controller. This provides maximum flexibility for the computational framework since now the computation could be used to implement calculations to do distributed control of the group of Crazyflies.

CHAPTER 5. MODELING OF THE CRAZYFLIE ON-BOARD CONTROLLER

5.1 Overall Flow

In any controller there are two main tasks: sensing and control. These tasks are also sequential tasks usually, since the control to be computed requires knowledge of the current system state. In the Crazyflie these two tasks are present, and contain smaller parts as well. The overall structure of the controller flow used with the Crazyflie's in this work can be seen in figure 5.1. Note that this controller flow is not the same as in the default Crazyflie firmware. The default control flow contains several discrepancies in it that make modeling the quadrotor difficult, so those discrepancies were identified and fixed. More discussion on the discrepancies can be found in appendix A.

5.2 Sensing Subsystem

The first stop in the controller flow is the sensing subsystem. This subsystem's purpose is to estimate the current attitude and angular rates of the quadrotor. To do this, the Crazyflie has two sensors: an accelerometer and a gyroscope. The accelerometer allows the Crazyflie to measure the gravitational vector (in the quadcopter body frame) and the gyroscope allows the Crazyflie to measure the angular rates about the principal body axes.

On the Crazyflie, those two sensors are contained in a single chip, the Invensense MPU-9250 [23]. The Crazyflie mounting of this chip has the sensor axes system not aligned with the aerospace axes system the model and controllers use. The chip instead uses a right-handed coordinate system where the x axis is to the left, and z is up. This means the first step after reading the sensor is to move the sensor readings into the body frame from the sensor frame.

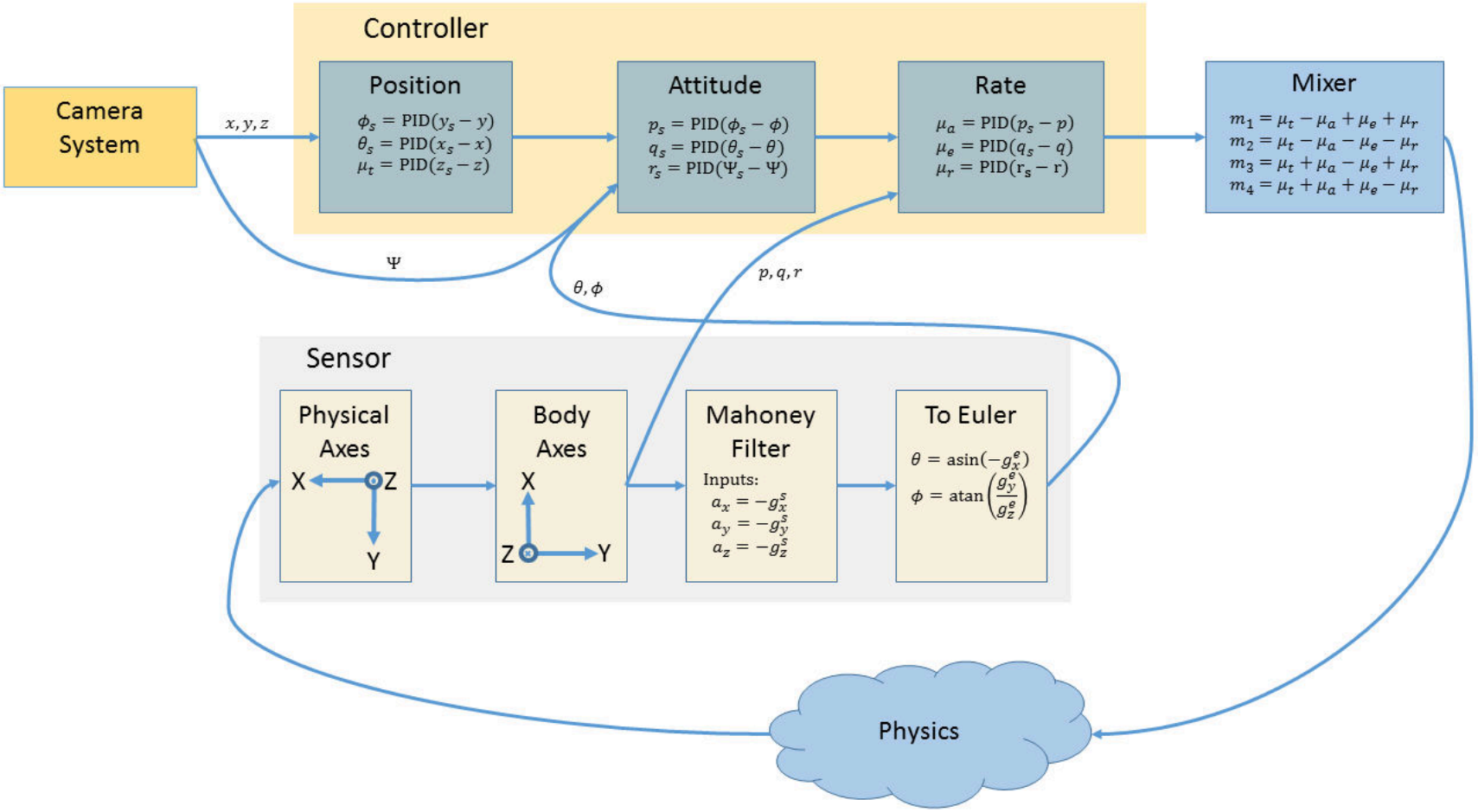


Figure 5.1: Mathematical structure of the firmware's control loop

5.2.1 Attitude Estimation

Once the sensor readings are aligned with the body frame, the actual attitude estimation occurs. On the Crazyflie there are three options for estimating the attitude: an Extended Kalman Filter [24], a Madgwick Filter [25], and a Mahoney filter [26]. In the firmware, the Mahoney filter is the one that is active by default.

The Mahoney filter is a recursive filter that operates in the following manner:

1. Take the sensed gravitational vector and angular rates as input
2. Compute the estimated gravitational vector using the previous pose estimate
3. Compute the error between the graviational vectors
4. Update the pose estimate using the error and the angular rates

The filter keeps track of the pose of the UAV using the quaternion representation for rotation.

The filter in equation form is:

$$e = \bar{g} \times \hat{g} \tag{5.1a}$$

$$\delta = K_p e + K_i \int e \tag{5.1b}$$

$$\dot{\hat{q}} = \frac{1}{2}\hat{q} \otimes \mathbf{p}(\bar{\Omega} + \delta) \tag{5.1c}$$

Where $\hat{g}$ is the estimated normalized gravitational vector, $\bar{g}$ is the measured gravitational vector, $\bar{\Omega}$ is the measured angular rates, and $\hat{q}$ is the estimated attitude in quaternions. Note that the operations $\otimes$ and $\mathbf{p}()$ are the quaternion multiplication and pure quaternion operation respectively.

Breaking down the filter operations in small chunks, there are three distinct phases. In the first phase, (5.1a) is used to figure out the angular error between the estimated gravitational vector and the measured gravitational vector. In the second phase, that angular error is fed into the PI controller (5.1b), which allows for a tuned filter response. This PI controller will drive the estimated vector to the measured vector, and remove the steady state error between them. The third step is the filter update phase. In this phase the measured angular velocity is combined with the PI output and used to update the quaternion estimate in (5.1c). A block diagram of this filter can be seen in 5.2.

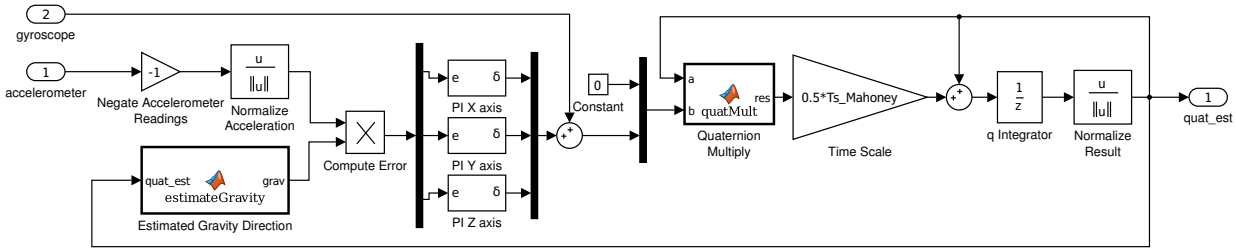


Figure 5.2: Block diagram of the Mahoney filter

Usually, UAVs use an accelerometer to measure the graviational field direction. These accelerometers actually measure a vector in the opposite direction of gravity [27], so the measured gravitational vector is actually $\bar{g} = -a$ where a is the accelerometer reading. The original Crazyflie firmware actually used the raw accelerometer vector in the filter computation, instead of the gravitational vector. This work uses firmware modified to use the gravitational vector instead. More description of the filter input discrepancy can be found in appendix A.

5.3 Controller Subsystem

The controller subsystem on the Crazyflie's is designed to be a modular component. In the firmware, the controller functions get passed all the states and all the setpoints, so the controller function can use any available information about the quadrotor in its computation. Several controllers have been implemented on the Crazyflie, including nested-loop PIDs (which are the default controllers), the large-angle nonlinear from [28] (implemented with slight modifications by [12]), and many others. This framework also makes it simple to implement custom controllers such as a generic state feedback, or other nonlinear controllers.

More discussion on the nested-loop PID controllers, including their structure and control response can be found in section 7.1.

5.3.1 Thrust Compensation

Quadrotors are usually battery powered devices, with a finite amount of power available per charge to fly with. As the quadrotor flys, it depletes the charge available in the battery causing a lower voltage output. This voltage drop means the software motor command does not correspond with a uniform speed command over the flight of the quadrotor. When doing linear control design, the nominal battery voltage is normally used (which is 3.7V for the Crazyflie). This poses a problem though, since at the beginning of a flight the battery may be charged up to 4.1V, and then at the end the battery may be down to 3.2V. This represents a drastic change in the produced speed of the motors over the flight.

To overcome this, the default firmware includes a thrust compensation function for the brushed motors. The implementation code for the function is:

```
float thrust = ((float)ithrust / 65536.0f) * 60;
float volts = -0.0006239 * thrust * thrust + 0.088 * thrust;
float supply_voltage = pmGetBatteryVoltage();
float percentage = volts / supply_voltage;
percentage = percentage > 1.0 ? 1.0 : percentage;
ratio = percentage * UINT16_MAX;
```

This code performs a remapping of the thrust commanded (*ithrust*) to the actual motor command (*ratio*). A plot of this algorithm can be seen in figure 5.3a. Note in this plot that the actual command output never reaches 1 even though 1 is inputted, so over the range 3.2 to 4.1 volts the motors will never run at full speed. This means that the firmware will be artificially limiting the thrust commands possible, making it so the Crazyflie can only lift 37 grams (including itself) with the battery at 3.7V.

Instead of the default thrust compensation, the Crazyflie was modified to use a scale-factor based method. This comprised two equations:

$$s = 1 + \frac{3.7 - V_b}{3.7} \tag{5.2a}$$

$$T_a = s * T_c \tag{5.2b}$$

Where V_b is the current battery voltage, T_c is the software commanded thrust and T_a is the thrust actually written to the motors. A plot of the implemented thrust compensation can be seen in figure 5.3b. Basically, this method would produce a thrust scaling such that at 3.7V if software commanded maximum thrust, then the actual command would be maximum thrust. This allows the Crazyflie to lift 49 grams (including itself) with the battery at 3.7V.

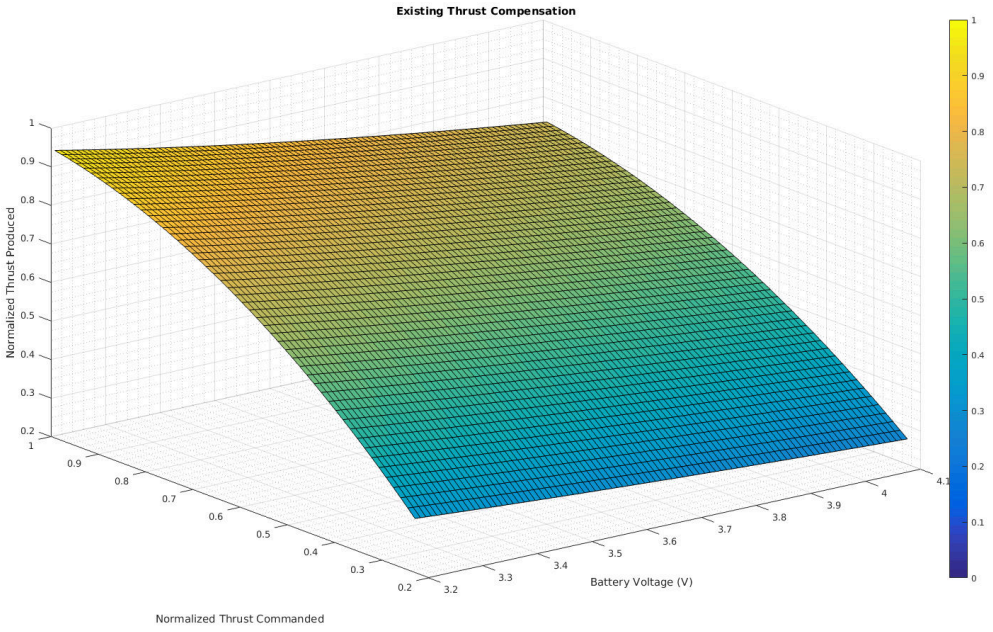
5.3.2 Input Mixing

The final step in the controller module is to mix the controller outputs (u_T, u_A, u_E and u_R) into the motor inputs. This is accomplished using a simple linear mixing system called a mixing matrix. This matrix takes the controller outputs and performs a weighted sum of them to compute the motor outputs. The weights on the controller inputs are determined by the direction a motor must change to create a positive force. For example, if a motor must increase speed for the pitch angle to increase, then the weight on u_E will be positive.

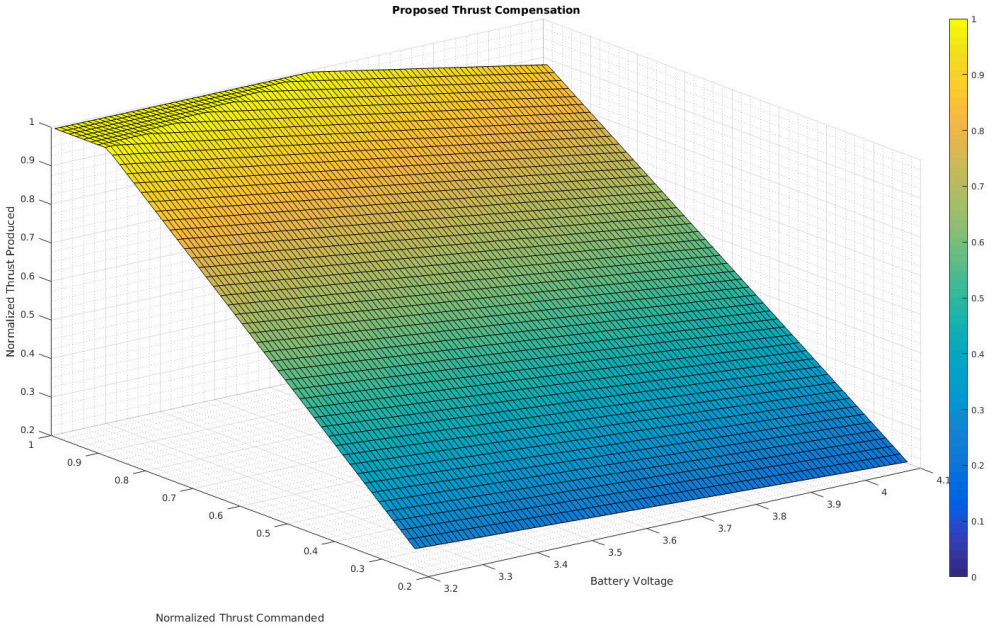
There are two common configurations for the motors on a quadrotor: in a cross configuration, where the motors are located on the principal body axes, and in an X configuration, where the motors are located on a line rotated 45° from the body axes. The configuration of the motors affects the mixing matrix structure, for instance a cross configuration would introduce zeros into the matrix since only two motors can affect pitch and two can affect roll. In the cross configuration though, every motor affects every rotation.

The Crazyflie 2.0 uses the cross configuration, meaning it uses the mixing matrix given in (5.3) for its motor mapping. Note in this mixer that the u_A and u_R inputs are scaled by $1/2$, this is just a quirk of the firmware though and is not needed.

$$\begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix} = \begin{bmatrix} 1 & -\frac{1}{2} & \frac{1}{2} & 1 \\ 1 & -\frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & 1 \\ 1 & \frac{1}{2} & \frac{1}{2} & -1 \end{bmatrix} \begin{bmatrix} u_T \\ u_A \\ u_E \\ u_R \end{bmatrix} \tag{5.3}$$



(a) Existing thrust remapping



(b) Proposed thrust remapping

Figure 5.3: Thrust compensation methods for overcoming battery drop

CHAPTER 6. PARAMETERIZATION OF THE CRAZYFLIE QUADROTOR

6.1 Mass

The mass of the quadrotor was measured using an Ohaus Model CS200 scale. This scale has a maximum capacity of 200g, with an accuracy of 0.1g [29]. The mass of the Crazyflie was measured in 3 configurations (all without trackables/trackable support frame): no battery, standard 240mAh battery, 380mAh battery. The masses are given in tables 6.2b, 6.2c, 6.2d respectively.

6.2 Motors

The Crazyflie uses 4 brushed DC motors to spin the rotors, which have the parameters given in table 6.1. These motors are designed for 4.2V operation with a maximum current of 1A. The datasheet specification lists the motor back-emf constant, K_v, as 14,000 RPM (or 1466.1 $\frac{\text{rad}}{\text{Vs}}$) [30]. The other parameter needed is the motor torque constant. For brushed DC motors, this constant is usually equal to the motor back-emf constant [31].

The motor resistance was measured by using a 4-port test system attached to the motor leads. The 4-port test system allows for the tester to remove the resistance of the test leads, which is important when measuring small resistance values.

The other important parameter is the motor's no-load current. This is the current that is required to turn the motor and overcome the internal inertia of the rotor. This parameter is measured by removing all loading from the motor (so no rotor is attached), then varying the input voltage and recording the measured current. Eventually the current will plateau at a constant value as the voltage is changing. This value is the no-load current, i_f.

Table 6.1: Motor parameters for the Crazyflie brushed DC motors

Motor Parameter	Measured Value
K_v	1466.1 $\frac{\text{rad}}{\text{Vs}}$
K_Q	1466.1 $\frac{\text{Nm}}{\text{A}}$
R_m	1.1Ω
i_f	0.0182A

6.3 Moment of Inertia

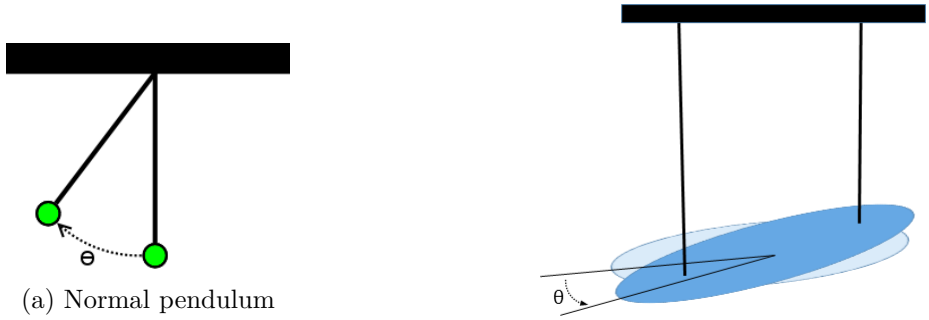
Determining the moment of inertia for the Crazyflie quadrotor is difficult due to its small size and mass. Prior work has focused on either using accurate CAD models with the proper densities of parts to compute the values (eg. [5] for the Crazyflie 2.0 or [32] for the Crazyflie 1.0) or mounting the Crazyflie as a pendulum and measuring the period of its swing (eg. [6]). Since the Crazyflie's used in this work utilize a larger battery than standard, the moment of inertia measurements in previous works would not be valid, so new measurements were needed.

Previous experiments in the lab at Iowa State had measured the moment of inertia of a quadrotor using a known torque applied to a rotating test stand. Based upon the measured angular position and applied torque, the moment of inertia could then be computed [3]. This method worked well for that quadrotor since its inertias were on the order of 10^{-3} and the test stand had inertia on the order of 10^{-3} [33]. However, since the Crazyflie's inertia has been previously estimated to be on the order of 10^{-5}, that test stand will not allow for accurate measurement of the moment of inertia. Instead, this work uses a bifilar pendulum to measure the moment of inertia about each of the three main rotational axes on the Crazyflie. The bifilar pendulum has been used in other works to measure the moment of inertia of objects ranging from tennis rackets [34], to model aircraft [35, 36].

6.3.1 Bifilar Pendulum Theory

A normal pendulum consists of an object attached to a single string which then is secured to a beam. To start rotation, the object is pulled vertically towards the beam while keeping the string taut, and then released. The motion of the object is then in an arc traced in a single vertical plane, as shown in figure 6.1a.

In a bifilar pendulum, the object is attached to two strings that are then run parallel to each other up to the beam where they are attached. To start rotation, the object is rotated horizontally about a point centered between the two strings. This then causes the object to begin rotating in the horizontal plane, as shown in figure 6.1b.



(a) Normal pendulum

(b) Bifilar pendulum

Figure 6.1: Illustration of angular path for the normal pendulum and bifilar pendulum

When the object is rotated in the horizontal plane, the fact that the bifilar pendulum's wires cannot change length means that the object is actually pulled up slightly from its equilibrium position. This slight movement is enough to create gravitational potential energy in the system in excess of the equilibrium energy. When it is released, this potential energy is subsequently converted into kinetic energy for the pendulum's movement, and so the rotation begins. In the real-world, the pendulum will experience damping effects due to the aerodynamic drag of the object and the viscous damping of the pendulum. This causes the rotations to gradually decay.

To develop a physics model for this system, it is common to use the Lagrangian dynamics approach to examine the energy transfer in the system [36]. Through this approach, the nonlinear mathematical model given in (6.1) can be found with the variables listed below.

θ = Angular position relative to equilibrium
I = Moment of inertia of the object
K = Aerodynamic drag constant
C = Viscous damping constant
D = Distance between the wires
m = Mass of the object
h = Height of wires (from center of mass to attachment point)

$$\ddot{\theta} + \left[\frac{K}{I}\dot{\theta}\left|\dot{\theta}\right| + \frac{C}{I}\dot{\theta}\right] + \left(\frac{mgD^2}{4Ih}\right)\frac{\sin\theta}{\sqrt{1-(1/2)(D/h)^2(1-\cos\theta)}} = 0 \tag{6.1}$$

Additionally, the nonlinear mathematical model can be linearized about the equilibrium point $\theta = 0$, giving (6.2) where A is the average amplitude of the oscillations.

$$\ddot{\theta} + \left[\frac{8AK}{3\pi I} + \frac{C}{I}\right]\dot{\theta} + \left(\frac{mgD^2}{4Ih}\right)\theta = 0 \tag{6.2}$$

Exploiting the fact that this is a second-order system, the moment of inertia can be found in relation to the other given variables and the natural frequency of the oscillation, since

$$\omega_n^2 = \frac{mgD^2}{4Ih}$$

Solving for I then produces (6.3) [37].

$$I = \frac{mgD^2}{4h\omega_n^2} \tag{6.3}$$

6.3.2 Measurement Procedure

When using the bifilar pendulum technique to measure the moment of inertia, the setup of the experiment is critical to the accuracy of the results. When setting up the pendulum itself two things must be kept in mind, otherwise errors in the measurements will occur:

- The wires must be parallel to each other
- The point of rotation (located halfway between the wires) should be located at the center of mass of the object

There are two parameters in the calculation that are user-controllable and can be used to improve the accuracy of the measurements. It has been determined analytically in [37] that h should be as large as possible. This helps because when h is large, the frequency of oscillation is decreased making its measurement more accurate. Additionally, they discuss how there is an optimum value for D based upon the desired error variance, damping coefficients, and moment of inertia. This value is difficult to calculate a priori though, so in this work the wire spacing was determined by the ease of physically attaching the strings to the Crazyflie.

(a) I_{xx} mounting

(b) I_{zz} mounting

Figure 6.2: Mounting orientations for the bifilar pendulum experiments.

The physical mounting of the Crazyflie for measuring the I_{xx} and I_{yy} can be seen in figure 6.2a. Fishing line was attached to the motors on the Crazyflie, making $D = 6.35$cm. This fishing line was then connected to the ceiling of the lab so that the height was as large as possible at $h = 222.25$cm. For mounting the Crazyflie to measure the I_{zz} value, the fishing line was attached to the plastic motor support legs. In this position though, the Crazyflie's center of mass is higher up than the motor supports, causing it to want to flip upside down. To overcome this, the Crazyflie was just mounted upside down, since this will have no effect on the measured moment of inertia but make the mounting more stable.

The Crazyflie under test was instrumented with small pieces of IR reflective tape at three positions, allowing it to be tracked using the Optitrack camera system in the lab. The angular position was then measured using the camera system. While normally a small initial angular displacement should be used to avoid nonlinear affects, the size of the Crazyflie made measuring small angles and rotating it to small angles difficult. So a larger angular displacement was used

initially, meaning the full non-linear model is needed for the final parameter estimation. The oscillations were then recorded for 2 minutes and the data was exported so that it could be post-processed in MATLAB.

6.3.3 Results

To do the parameter estimation for the moment of inertia, the nonlinear dynamics equation in (6.1) was implemented in a Simulink block diagram. The parameters K, C, and I were made tunnable by a script. Then a simulation was run starting at the maximum angular deflection and running for as long as the dataset. The simulated trajectory was plotted against the recorded data and a manual process of tunning the parameters was used to hone in on the parameters. A sample plot showing the simulated trajectory versus the measured trajectory can be seen in figure 6.3. The final moments of inertia can be seen in tables 6.2b, 6.2c, and 6.2d for no-battery, the 240mAh battery and the 380mAh battery respectively.

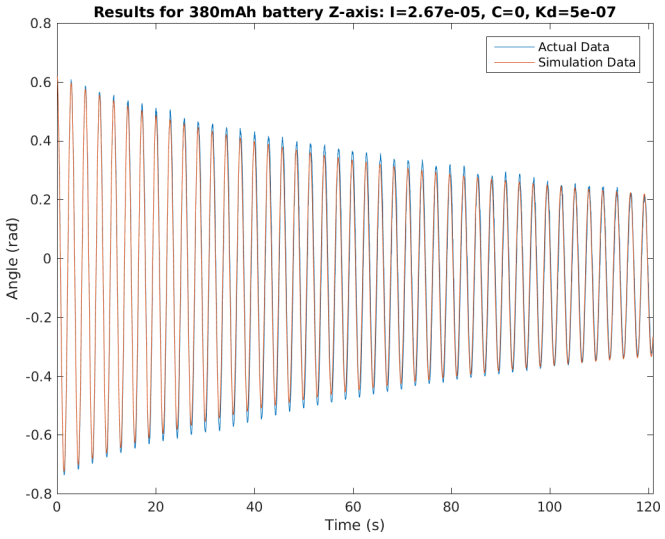


Figure 6.3: Plot showing the bifilar pendulum's measured angular position versus the simulated angular position

Comparing the results presented here against those in [5, 6] for the Crazyflie 2.0 shows that the computed moments of inertia are similar. Additionally, based upon the results in [6], the assumption that the non-diagonal terms in the moment of inertia matrix are zero is valid, since their results show the non-diagonal terms to be at least an order of magnitude smaller than the diagonal components.

6.4 Rotor Parameters

In the model of the quadrotor that is being used, there are three main parameters associated with the rotor subsystem: the thrust constant, the in-plane drag constant, and the equivalent moment of inertia of the rotor and motor shaft. While each of these three parameters could be analytically found using various dimensions and assumptions about the design of the rotor (see the discussion in section 4.2 of [3]), those calculations require parameters which are difficult to measure and not widely available. Instead, methods to measure the parameters directly were used, and the appropriate experimental setup was created.

6.4.1 Measurement Setup

6.4.1.1 Measured Quantities

In order to calculate the three parameters for the rotor system, the following physical quantities must be simultaneously measured:

1. Thrust produced
2. Rotor angular speed
3. Motor current draw
4. Commanded duty cycle

The method of measuring each of those quantities is now discussed

Thrust

In order to measure the thrust produced by the rotors, the Crazyflie was secured to a Vernier Dual-Range Force Sensor [38]. This sensor is able to measure the force applied to it on either a 10 or 50 Newton scale with an accuracy of 0.01N and 0.05N respectively. For these experiments, it was set on the 10N scale. This sensor was connected to a Vernier LabPro interface which was attached to a computer running the Vernier LoggerPro software [39].

Rotor angular speed

Since the rotors of the Crazyflie are very small and it uses brushed motors, the method of measuring rotor speed given in [3] is not possible. Instead a new method was devised that used a photointerrupter to count the number of times a rotor blade passed through the beam in 1 second. More details on this device can be found in the next section.

Motor current draw

To measure the current draw of the motors, the display on the power supply was used. This power supply was capable of supplying up to 5A at the desired voltages, and could measure in the milliamp range.

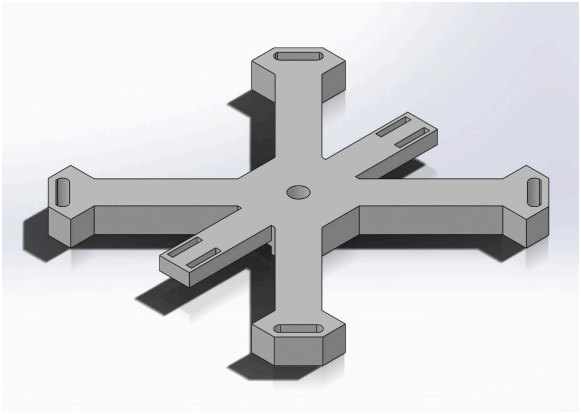
Commanded duty cycle

The commanded duty cycle was measured as the value sent directly to the *setMotorRatio()* function in the Crazyflie firmware. Additionally, when performing these test the thrust compensation feature of the firmware was disabled since this ffunction will change the duty cycle actually sent to the motor from what is desired. A special firmware version was developed that bypassed the controller structure and allowed for the motors to be set directly from the ground station software.

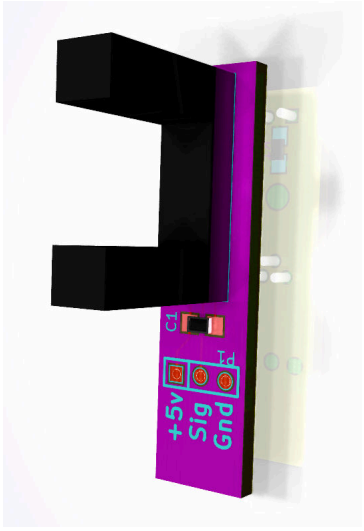
6.4.1.2 Physical Test Apparatus

The physical test apparatus used in these experiments consists of two parts: mechanical and electrical[1].

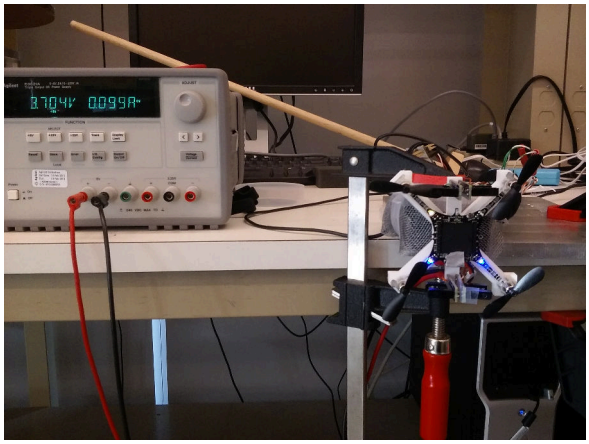
[1]The CAD files, electrical schematics, and source code files can be found at https://github.com/imciner2/Crazyflie_Tools



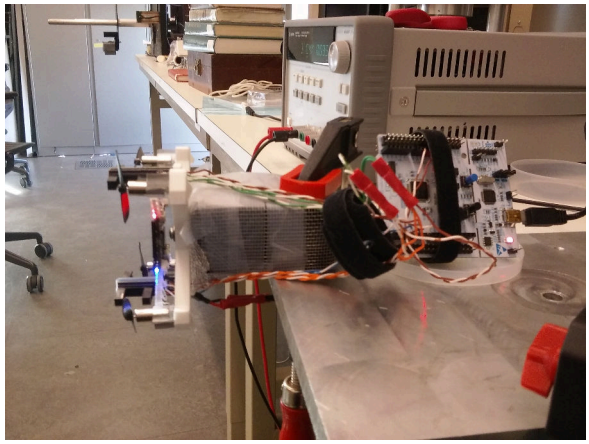
(a) CAD rendering of the force sensor mount



(b) Circuit board for the photointerrupter



(c) Front view



(d) Side view

Figure 6.4: Setup used to measure the rotor parameters.

The main portion of the mechanical part consists of a 3D printed base in the shape of an X. The Crazyflie's feet attach into slots at the end of each leg of the X, and the center of the base contains a mounting hole and slot to attach to the force sensor and prevent rotation. There are also two supports on the base to hold the four photointerrupter boards so that the propeller passes through the beam in the center of the opening. A CAD rendering of the base can be seen in figure 6.4a. Additionally, the force sensor was wrapped in aluminum window screen to shield it from electrical noise. It was found that the Crazyflie's Bluetooth radio easily interferes with the force sensor, creating measurement bias and more high frequency noise.

Each photointerrupter is mounted on a custom designed circuit board. That circuit board contains the passive electrical components necessary for each photointerrupter, and easy wire attachments for the power and signal wires. Additionally, the circuit board acts as the main support to hold the photointerrupter vertical on the base. A rendering of the board can be seen in figure 6.4b.

The photointerrupter used is the Sharp GP1A57HRJ00F [40]. It consists of an IR transmitter and IR receiver, with a gap of 1cm between them. It outputs a 5V logical signal indicating if the IR beam is sensed at the receiver (5V if the gap is clear, 0V if the gap is blocked). The four photointerrupters are wired back to a single ST Microelectronics Nucleo F401RE development board [41]. This development board contains an ST Microelectronics STM32F401RE ARM core processor and integrated JTAG programmer.

There are two different software programs that were developed for the STM32 processor, one to measure the rotor speed and the other to measure the motor transient. To measure the motor speed, the software counts the number of times the photointerrupter beam is broken in one second. The motor speed is then half of that number (since the Crazyflie's have two blades on the rotors). The measured rotor speeds are reported over the serial port every two seconds for recording.

To measure the motor transient, the software counts the amount of time between beam breaks. This time when multiplied by two gives the rotor's instantaneous angular period. The raw period values are sent over the serial port to a recording computer, where they are post-processed in MATLAB to compute the instaneous angular velocity.

6.4.1.3 Procedure

In order to get the best parameter fit possible for the thrust constant and the in-plane drag constant, data was collected at 21 duty cycles between 15000 and 65000, and at five different supply voltages (3.2V, 3.5V, 3.7V, 4.0V, and 4.2V).

The procedure for gathering the data is as follows:

1. Set the power supply to the desired voltage
2. Set the motors to the desired duty cycle using the software
3. Collect 1 minute of force data in LoggerPro then average the data
4. Record:

   Measured angular speed of each rotor

   The average force data

   Power supply current

To measure the equivalent moment of inertia, the rise-time of the motor-rotor system was measured using the following procedure

1. Command the motor to a speed so that it is just barely spinning
2. Provide a step change in the speed to just below full-speed
3. Record the instantaneous speed as the step occurs

6.4.2 Thrust Constant Calculation

The rotor thrust constant K_t relates the rotor angular speed to the thrust produced by the rotor through

$$|T| = K_t \omega^2$$

In these experiments, the thrust measured is the combined thrust of all four rotors. This means that at each datapoint the relation is

$$|T| = K_t(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2)$$

Since 105 datapoints were collected, an overdetermined system of equations is created of the form

$$\begin{bmatrix} |T_1| \\ |T_2| \\ \vdots \\ |T_{105}| \end{bmatrix} = K_t \begin{bmatrix} \omega_{1,1}^2 + \omega_{2,1}^2 + \omega_{3,1}^2 + \omega_{4,1}^2 \\ \omega_{1,2}^2 + \omega_{2,2}^2 + \omega_{3,2}^2 + \omega_{4,2}^2 \\ \vdots \\ \omega_{1,105}^2 + \omega_{2,105}^2 + \omega_{3,105}^2 + \omega_{4,105}^2 \end{bmatrix}$$

This system can then be solved for K_t using least-squares regression producing

$$K_t = 1.7449 \times 10^{-8}$$

The resulting fit line can be seen in figure 6.5a.

6.4.3 Rotor Drag Constant Calculation

The rotor drag constant relates the drag torque generated by the rotor's spinning motion to the current rotor speed through

$$T_d = k_d \omega^2$$

Calculating this parameter is not as simple as the thrust constant from the previous section, since measuring the actual torque produced by the rotor is not a simple task. Instead, the drag torque can be estimated by observing the steady-state velocity of the rotor at a given command (as described in [3]), since the steady-state velocity ω is given by

$$0 = \frac{1}{R_m K_Q} u_p V_b - \frac{1}{R_m K_Q K_v} \omega - \frac{1}{K_Q} i_f - K_d \omega^2 \tag{6.4}$$

Finding K_d using this relation now depends on the following physical parameters: R_m, K_Q, K_v and i_f. (all of which are known). Additionally, three different variables must be measured at every data point for this equation:

- u_p - The command given to the motor (in the range $[0, 1]$)
- V_b - The voltage being supplied to the motor
- ω - The angular velocity of the rotor

In this work, those variables were measured using the setup described in 6.4.1.2, while using the procedure given in section 6.4.1.3. Each rotor speed command created 4 instances of (6.4), meaning there are 420 equations. Now the unknown parameter K_d could be solved using the following linear relation

$$\begin{bmatrix} \frac{1}{R_m K_Q} u_{p(1,1)} V_{b(1,1)} - \frac{1}{R_m K_Q K_v} \omega_{1,1} - \frac{1}{K_Q} i_f \\ \frac{1}{R_m K_Q} u_{p(2,1)} V_{b(2,1)} - \frac{1}{R_m K_Q K_v} \omega_{2,1} - \frac{1}{K_Q} i_f \\ \frac{1}{R_m K_Q} u_{p(3,1)} V_{b(3,1)} - \frac{1}{R_m K_Q K_v} \omega_{3,1} - \frac{1}{K_Q} i_f \\ \frac{1}{R_m K_Q} u_{p(4,1)} V_{b(4,1)} - \frac{1}{R_m K_Q K_v} \omega_{4,1} - \frac{1}{K_Q} i_f \\ \frac{1}{R_m K_Q} u_{p(1,2)} V_{b(1,2)} - \frac{1}{R_m K_Q K_v} \omega_{1,2} - \frac{1}{K_Q} i_f \\ \vdots \\ \frac{1}{R_m K_Q} u_{p(3,105)} V_{b(3,105)} - \frac{1}{R_m K_Q K_v} \omega_{3,105} - \frac{1}{K_Q} i_f \\ \frac{1}{R_m K_Q} u_{p(4,105)} V_{b(4,105)} - \frac{1}{R_m K_Q K_v} \omega_{4,105} - \frac{1}{K_Q} i_f \end{bmatrix} = K_d \begin{bmatrix} \omega_{1,1}^2 \\ \omega_{2,1}^2 \\ \omega_{3,1}^2 \\ \omega_{4,1}^2 \\ \omega_{1,2}^2 \\ \vdots \\ \omega_{3,105}^2 \\ \omega_{4,105}^2 \end{bmatrix}$$

By then solving this over-determined system as a least-squares problem, the following value was found, which created the fit line seen in figure 6.5b.

$$K_d = 1.6881 \times 10^{-10}$$

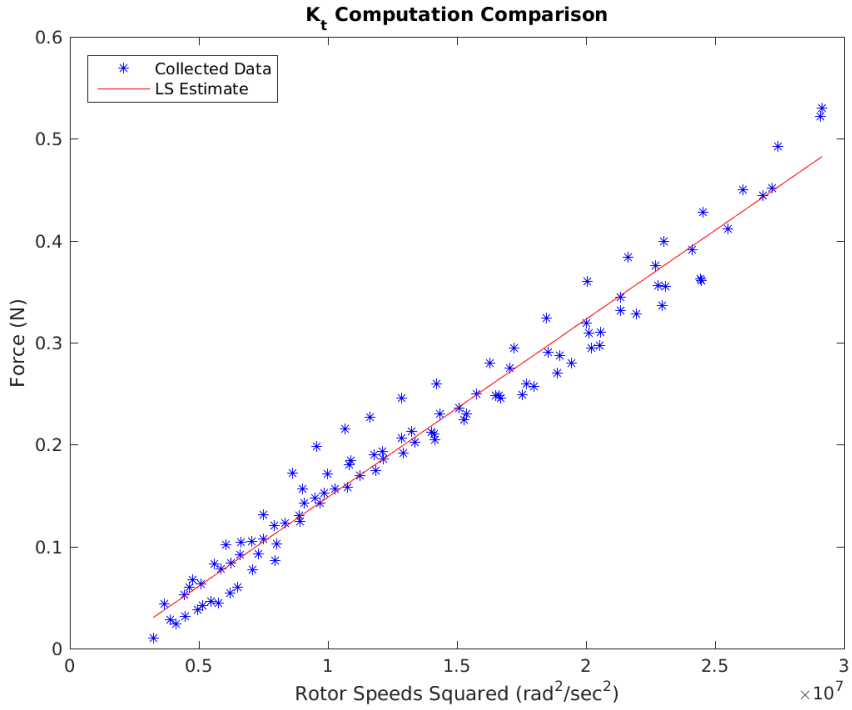
6.4.4 Equivalent Moment of Inertia Calculation

To compute the equivalent moment of inertia for the motor-rotor system, the first-order transfer function model of the rotor speed generation is used

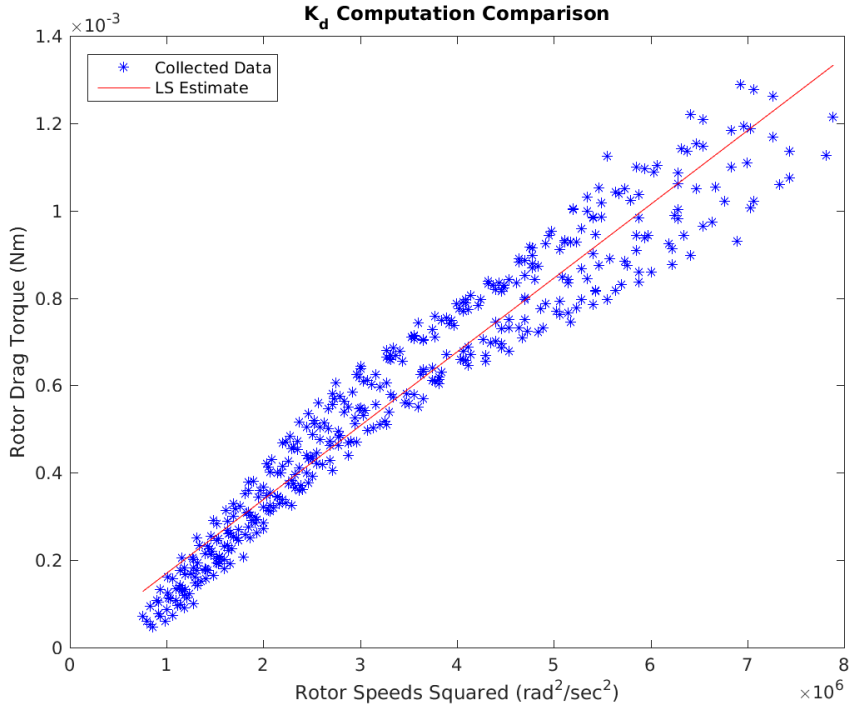
$$\omega(s) = \frac{K_v}{R_m K_v K_q J_r s + 1} U(s)$$

A measurement of the time constant of the physical system was then made by providing a step input and using the second of the software programs discussed in 6.4.1.2 to record the response. The resulting step response can be seen in figure 6.6. The time constant is then the time between the step starting and when it crosses 67% of the final value, in this case $\tau = 0.0703s$. By then using the fact that $\tau = R_m K_v K_q J_r$ and all parameters other than J_r are known, the final value of J_r can be found, producing

$$J_r = 2.9747 \times 10^{-8}$$



(a) Rotor force curve



(b) Rotor drag curve

Figure 6.5: Computed rotor parameter versus the measured dataset

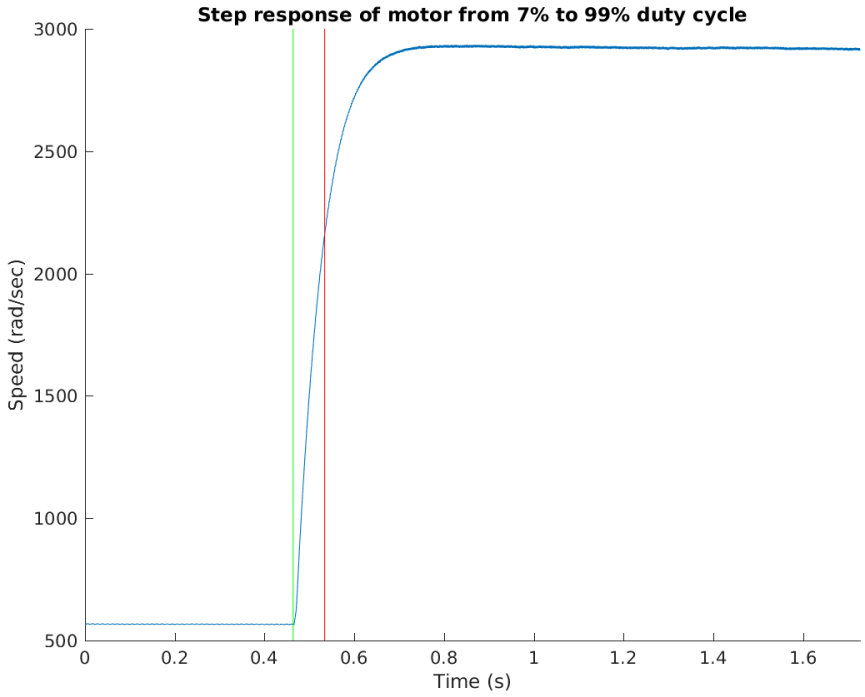


Figure 6.6: Step response of the motor-propeller system when commanded duty cycle is changed

6.5 Additional Parameters

Since the model being used in this work is the one developed in [3], there are several other parameters required from the Crazyflie, namely the K_H and δT parameters along with direction unit-vectors for the rotor forces and drag constants.

The unit vectors for the rotor forces are simple to derive, since the rotor produces thrust in the negative z direction only, the unit vector is $\Gamma_T = [0,0,-1]^T$. The unit vectors for the rotor drag constants are similar, just dependent upon the direction of spin for the rotor (since the drag will be in the direction opposite the spin). This means that rotors 1 and 2 have $\Gamma_\Omega = [0,0,1]^T$ and rotors 3 and 4 have $\Gamma_\Omega = [0,0,-1]^T$.

The K_H and δT terms are harder to compute though, and as discussed in [3], must be done iteratively using the simulation. The K_H term is hand-tuned until the simulation matches with the actual data for the x and y axes step response, while the δT parameter can be computed through comparing the quadratic approximation of the rotor force with more complex equations

relating the rotor force to the rotor speed. For this work, both of those terms are neglected. This can be done because the simulations run with $K_H = 0$ and $\delta T = 0$ compare very well to the actual flight data captured using the PID controllers from section 7.1.

6.6 Overall Parameters

The tables inside table 6.2 contain all the parameters estimated for the Crazyflie quadrotor. They are broken apart into 4 subtables:

- Table 6.2a contains parameters related to the Crazyflie quadrotor in general.
- Table 6.2b contains physical parameters of the Crazyflie with no battery.
- Table 6.2c contains physical parameters of the Crazyflie with the default 240mAh battery.
- Table 6.2d contains physical parameters of the Crazyflie with a larger 380mAh battery.

Table 6.2: Parameters for the Crazyflie quadrotor

(a) General parameters

Parameter	**Value**	**Units**	**Description**
r_x	0.033	m	Distance from rotor hub to center of mass along b_x axis
r_y	0.033	m	Distance from rotor hub to center of mass along b_y axis
r_z	0.010	m	Distance from rotor hub to center of mass along b_z axis
K_t	1.7449×10^{-8}	$\frac{\text{kgm}}{\text{rad}^2}$	Rotor thrust constant
K_d	1.6881×10^{-10}	$\frac{\text{kg}}{\text{rad}}$	Rotor drag constant
J_r	2.9747×10^{-8}	kgm^2	Motor and rotor moment of inertia
τ_r	0.0703	s	Motor and rotor time constant
K_Q	1.4661×10^3	$\frac{\text{Nm}}{\text{A}}$	Motor torque constant
K_v	1.4661×10^3	$\frac{\text{rad}}{\text{V s}}$	Motor back-EMF constant
R_m	1.1	Ω	Motor winding resistance
i_f	0.0182	A	Motor no-load current
$P_\perp$	1400	(none)	Minimum software command to turn motor
$\tilde{P}_\perp$	0	(none)	Minimum software command for motor
$P_\top$	65,536	(none)	Maximum software command for motor

(b) Parameters for the Crazyflie quadrotor with no battery

Parameter	**Value**	**Units**	**Description**
m	0.0211	kg	Mass of quadrotor
I_{xx}	1.310×10^{-5}	kgm^2	Moment of inertia about the b_x axis
I_{yy}	1.290×10^{-5}	kgm^2	Moment of inertia about the b_y axis
I_{zz}	2.175×10^{-5}	kgm^2	Moment of inertia about the b_z axis

(c) Parameters for the Crazyflie quadrotor with standard 240mAh battery

Parameter	**Value**	**Units**	**Description**
m	0.0284	kg	Mass of quadrotor
I_{xx}	1.329×10^{-5}	kgm^2	Moment of inertia about the b_x axis
I_{yy}	1.333×10^{-5}	kgm^2	Moment of inertia about the b_y axis
I_{zz}	2.640×10^{-5}	kgm^2	Moment of inertia about the b_z axis

(d) Parameters for the Crazyflie quadrotor with 380mAh battery

Parameter	**Value**	**Units**	**Description**
m	0.0317	kg	Mass of quadrotor
I_{xx}	1.433×10^{-5}	kgm^2	Moment of inertia about the b_x axis
I_{yy}	1.473×10^{-5}	kgm^2	Moment of inertia about the b_y axis
I_{zz}	2.670×10^{-5}	kgm^2	Moment of inertia about the b_z axis

CHAPTER 7. MODEL VERIFICATION AND CONTROLLER DESIGN

With the quadrotor parameterization conducted in chapter 6 and the physics model of the quadrotor described in [3], the quadrotor system can be simulated in Simulink to view the response of the physics to different controllers, and also to design more complex model-based controllers. The first controllers examined were nested loop PIDs. This control scheme is common on quadrotors, and comes in the standard Crazyflie firmware. This controller scheme was also used to stabilize the quadrotor so that the model and the actual response could be compared to gauge the accuracy of the model.

Next, state feedback controllers were designed using the model. Initially an LQR design methodology was used to find the feedback gains, however flight tests showed that the LQR had large steady-state offsets on x and y along with oscillations on the Euler angles. To reduce the steady-state offsets and the oscillations, integrator states were added to the controller on the x, y, z, ϕ, θ and ψ states.

7.1 PID Controller

The first controllers examined on the Crazyflie quadrotor were Proportional-Integral-Derivative (PID) controllers. PID controllers are one of the most prevalent controllers in the hobbyist quadrotor market due to the ease of their implementation and also the ability for their control response to be tuned with little knowledge of the underlying physics. The Crazyflie in this work uses 9 different PID controllers in a nested-loop configuration (as shown in figure 7.2).

7.1.1 Theory

The PID controller is one of the most ubiquitous controllers in use today, and its theory is covered extensively in most texts for an undergraduate control systems course (see e.g. [42], [43]). What follows is a short summary of the relevant theory.

The overall PID controller takes the s-domain form shown in (7.1). This equation shows the input-output relation for the controller where the input signal is the error signal (reference signal minus the current system state) and the output signal is the actuator input [42]. It consists of three distinct components: a proportional term, an integral term and a derivative term.

$$\frac{U(s)}{E(s)} = k_p + \frac{k_i}{s} + k_d s \tag{7.1}$$

Each component plays a different part in controlling the system. The proportional component makes the controller react to the instantaneous error, so if the system has large error then the controller produces a large output. The integral term examines the error over time, which will reduce the steady-state error of the system. The derivative term examines the rate of change of the error (the speed at which the error changes), which will allow for the controller to dampen high-speed oscillations in the response.

Each term contains a coefficient which specifies the amount the term affects the controller output. The process of finding the right coefficients is referred to as tuning the controller. This process can be difficult for complex or unstable systems. In this case, the Crazyflie comes with its internal control loops already tuned with stabilizing coefficients, so the tuning focused on the outer loops.

In order to implement the PID controller shown in (7.1) on the Crazyflie, it needs to be discretized. In this case, the PID utilizes the right-sided rectangle rule (or backward rule) for its integral calculation and a backward difference method for its derivative. Using the appropriate transforms from [44] and [45], the complete discretized forumla for the PID in the z-domain can be seen in (7.2). Note that this has a fourth parameter in it, T_s, which is the sampling rate of the controller.

$$\frac{U(z)}{E(z)} = k_p + k_i \left(\frac{T_s z}{z-1} \right) + k_d \left(\frac{z-1}{T_s z} \right) \tag{7.2}$$

7.1.2 Implementation

The discrete PID controller from (7.2) is implemented in the Crazyflie using the C programming language, and is the default controller shipped with the Crazyflie. A block diagram of the controller implementation can be seen in figure 7.1. It utilizes floating-point computation and contains three important features:

- Integral term saturation
- Resettable integral term
- No derivative term immediately after reset

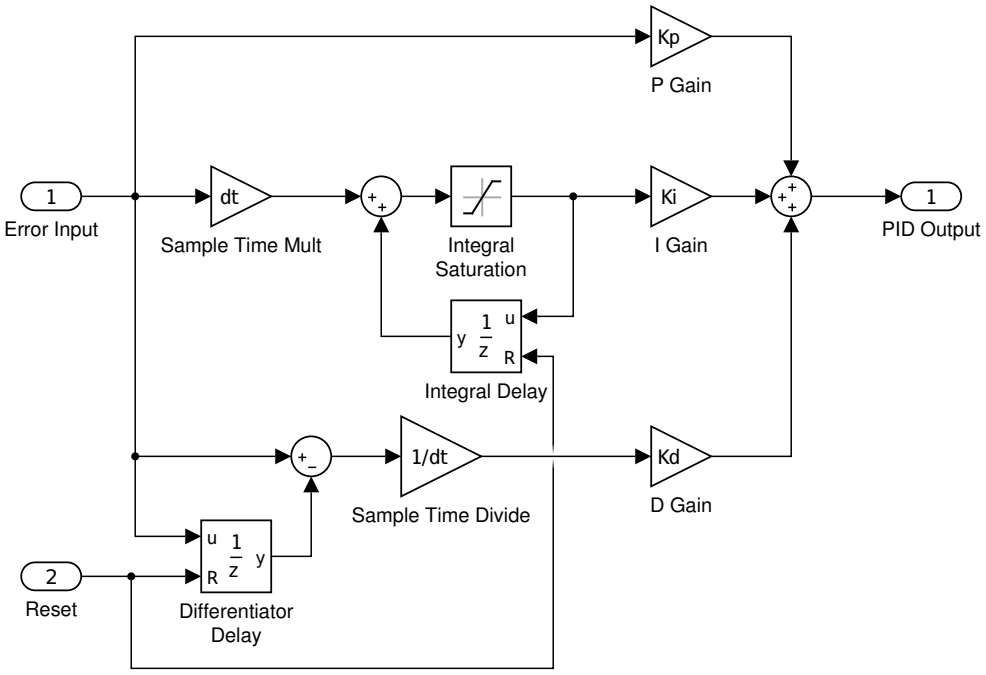


Figure 7.1: Block diagram for the PID implemented on the Crazyflie

The above PID control block was replicated to create 9 different controllers on the Crazyflie firmware. These controllers are arranged in a nested loop architecture. The outer most loop contains 3 controllers for controlling the x, y, z position of the quadrotor in the flight area. From this loop, the z controller output is fed directly into the mixing matrix as the u_t input, with the scaling thrust compensation from (5.2) applied. The output of the x and y controllers are fed into the reference inputs to the pitch and roll controllers respectively.

The middle loop contains 3 controllers responsible for controlling the angular position of the quadrotor. As mentioned before the reference input for pitch and roll came from controllers on the outer loop, while the reference for the yaw controller is a user-provided value. Additionally, the sensor input for the controllers come from different places. The roll and pitch controllers receive their Euler angle sensor input from the on-board Mahoney filter (which uses the on-board sensors for finding the orientation), and the yaw controller receives its Euler angle from the external camera system.

The inner most loop contains the controllers responsible for controlling the angular rate of the quadrotor. The output of each of the three middle loops provides the reference inputs to the inner most loop controllers. The sensor inputs for these three controllers come from the on-board gyroscope sensor. These controllers provide their output directly to three mixing matrix channels (no thrust compensation is used on these three channels).

A diagram showing the interconnection of the nested loop PID structure can be seen in figure 7.2.

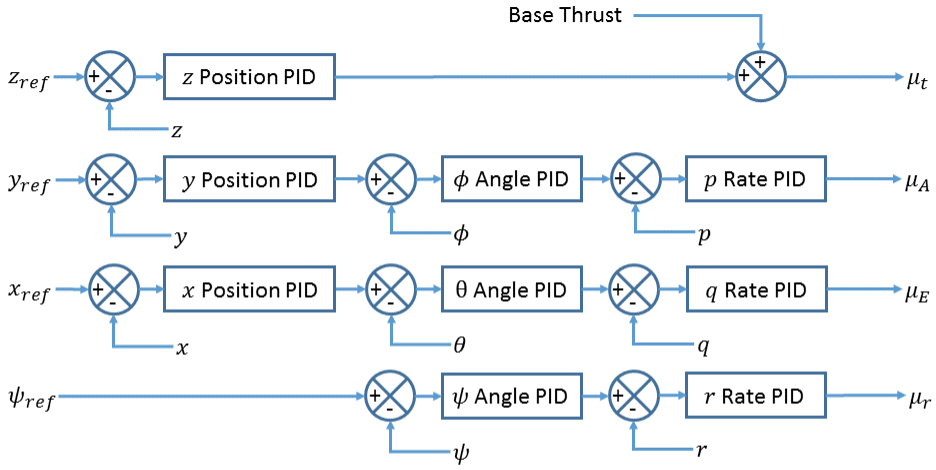


Figure 7.2: Nested loop PID architecture

The Crazyflie comes with PID controllers already designed for the inner two loops, which were used as a baseline point for tuning stabilizing controllers on the outer loop. The final PID values used on the Crazyflie are given in table 7.1.

Table 7.1: PID controller parameters

	p **Rate**	q **Rate**	r **Rate**	ϕ **Angle**	θ **angle**	ψ **Angle**	**Y**	**X**	**Z**
T_s	0.002s	0.002s	0.002s	0.002s	0.002s	0.01s	0.01s	0.01s	0.01s
K_p	250.0	250.0	70.0	6.0	6.0	6.0	20.0	-20.0	-10000
K_i	500.0	500.0	16.7	1.0	1.0	0.0	1.0	-1.0	-2000
K_d	2.5	2.5	0.0	0.0	0.0	0.35	22	-22	-15000
i_{limit}	33.3	33.3	166.7	20.0	20.0	360.0	40	40	10000

7.1.3 Pseudo-Nonlinear Extension

The PID structure defined in the previous subsection works well for when the quadrotor is hovering at an angle of 0° yaw. However, since the x and y position sensors and setpoints are in the inertial frame of reference, if the quadrotor yaws any, the two frames will no longer be in-sync and the mapping of x motion to pitch angle is no longer completely valid (for instance, under a 90° yaw the roll angle would then affect the position on the inertial x axes).

To allow for the quadrotor to fly while at a non-zero yaw angle, a remapping of the x and y axes must be done. This mapping is referred to as a pseudo-nonlinear extension to the controllers in [3]. The mapping consists of simply rotating the errors in the inertial frame to be inside the body frame, which can be accomplished through the rotation matrix for a yaw rotation. This matrix is applied to the error signal before it is given to the controllers, so the PID controllers are actually operating on the body frame x and y axes. The mapping is:

$$\begin{bmatrix} \hat{x}_b \\ \hat{y}_b \\ \hat{z}_b \end{bmatrix} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} \tag{7.3}$$

With the x_e, y_e and z_e being the inertial frame errors, and $\hat{x}_b$, $\hat{y}_b$ and $\hat{z}_b$ being the errors moved into the body frame under the currently sensed yaw, ψ.

7.1.4 Response

To examine the response of the Crazyflie controllers, step inputs were given to the three major axes (x, y, and z) on both the physical quadrotor system and the non-linear physics model. The results comparing the step response of the three axes can be seen in figures 7.3, 7.4 and 7.5.

On the x axes, multiple step input changes were commanded of the quadrotor. The response can be seen in figure 7.3. In this response, the simulation corresponds very closely with the actual quadrotor response in the position step response. One thing to note though is the pitch angle of the quadrotor (as measured by the on-board Mahoney filter) has an offset when the quadrotor is just hovering. This is most-likely caused by the physical center of mass being offset from the origin of the body frame, whereas the model assumes the center of mass is located at the origin of the body frame. The physical response of the pitch angle to a setpoint change (caused by the step on the position) still matches the expected simulation during the transient period though.

On the y axes, step input changes were commanded of the quadrotor. The response can be seen in figure 7.4. In this response, the simulation result corresponds very closely with the actual quadrotor response. Similar to the pitch angle, the roll angle has a slight offset when the quadrotor is just hovering, and it is probably due to the same reason. The physical response of the roll angle does still seem to match the simulated response during the transient period.

The response of the quadrotor to a step on the z axes can be seen in figure 7.5. This response shows a slight inconsistency between the simulated and experimental responses. The actual response appears to be slightly slower than the simulated response, and has a slightly higher overshoot. This discrepancy is most likely due to a modeling error in the mass of the quadrotor. The model uses the idealized mass determined in section 6, while the actual quadrotor is flying with a trackable frame attached, adding mass to the overall system.

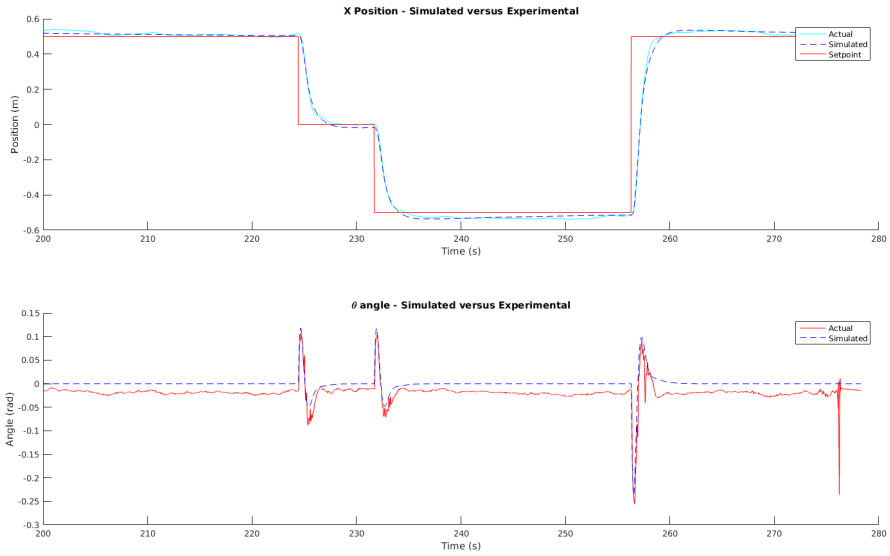


Figure 7.3: Response of the PID controller to a step input in the x direction.

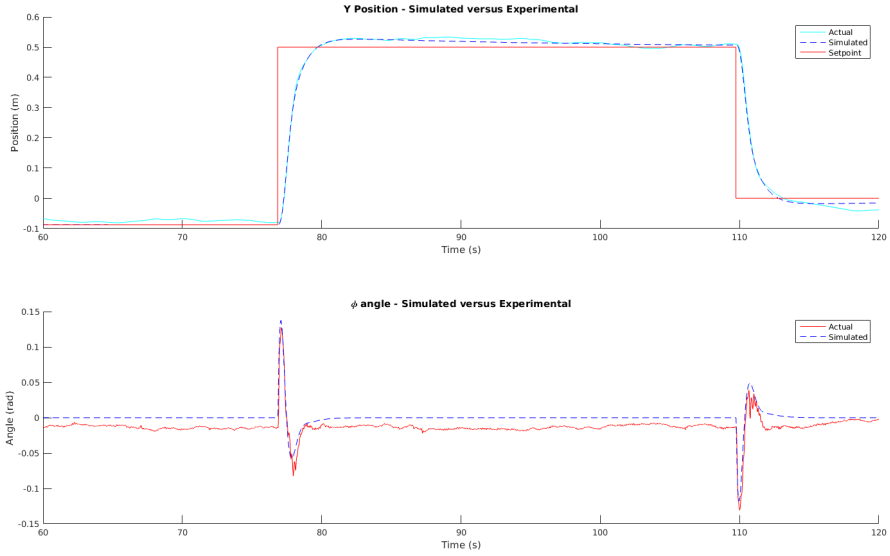


Figure 7.4: Response of the PID controller to a step input in the y direction.

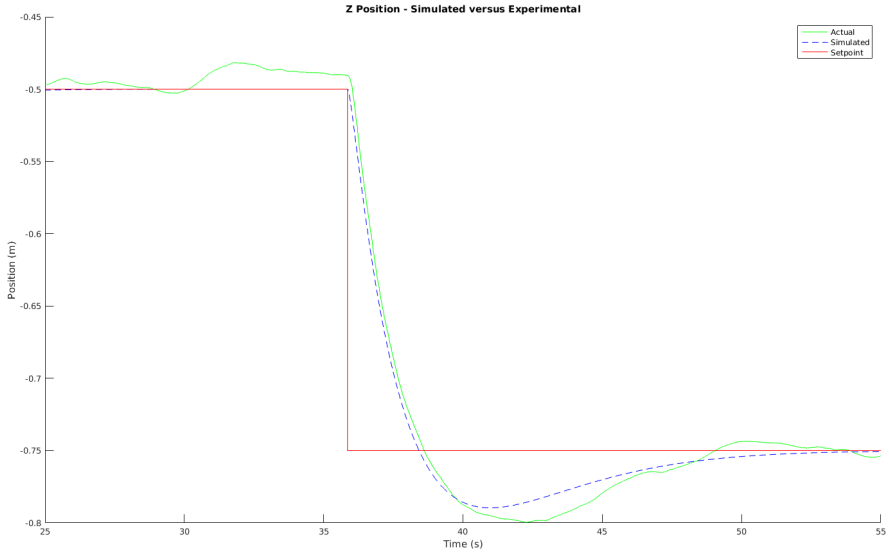


Figure 7.5: Response of the actual PID controller (green line) and simulated PID controller (blue line) to a step input (red line) in the z direction.

7.2 Linear State-Feedback Controller

The next controller type implemented on the Crazyflie platform was a linear state-feedback controller.

7.2.1 Linearized Model

In order to design the state-feedback controllers, a linear model of the Crazyflie dynamics was created. This model uses the physical dynamics derived in [3], and the parameters found in section 6.

For the model, the state ordering used is:

$$\Lambda = \begin{bmatrix} u & v & w & p & q & r & x & y & z & \psi & \theta & \phi \end{bmatrix}^T \tag{7.4}$$

This state vector does not include any rotor speed states. Those states were removed from the system for controller design (so the command output is directly proportional to the motor speed with no transient). This was done because there is no way of directly measuring the rotor speeds on the Crazyflie, so an estimator would have been needed. Additionally, since the rotor state transient is much faster than the quadrotor response (by an order of magnitude or more), the controller response would not be greatly affected by having those states excluded. So by removing those states, a full state estimator was not needed.

The input ordering used on the model is:

$$U = \begin{bmatrix} u_t & u_A & u_E & u_r \end{bmatrix}^T \tag{7.5}$$

This forms the linear system

$$\dot{\Lambda} = A_s \Lambda + B_s U$$

Where A_s and B_s are given in (7.6).

$$A_s = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -9.81 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9.81 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B_s = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -0.0003 & 0 & 0 & 0 \\ 0 & 0.0092 & 0 & 0 \\ 0 & 0 & 0.0091 & 0 \\ 0 & 0 & 0 & 0.0029 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.6}$$

7.2.2 Controller Implementation

The state space controller implemented on the Crazyflie was designed to be a generic linear static-gain state-feedback (SGSF) controller, where the gain matrix can be modified in-flight by the ground station (similar to the PID update architecture). To make this controller generic, the 12 quadrotor states were supplemented with 12 additional integrator states, making 24 total controller states (so the controller K matrix is 4×24). The 12 additional states are simply integrator states on each of the quadrotor states' error terms, allowing for the controller to use any integral term desired without having to modify the Crazyflie's firmware. An overview of the structure can be seen in figure 7.6.

For the Crazyflie system used here, all the physical states (except for linear velocity) are available directly from sensors. The choice was made to simplify implementation by using direct sensor measurements, and then determine the linear velocity by a numerical derivative. The integrator states were added in software thru a numerical integration using the right-sided rectangle rule (or backward rule). Additionally, the pseudo non-linear extension from section 7.1.3 was implemented on the linear position and linear velocity errors, and their integrals.

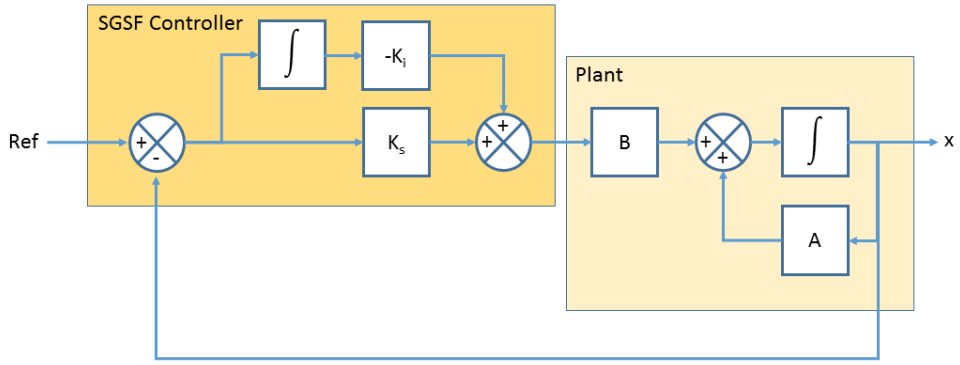


Figure 7.6: State space controller structure

7.2.3 Controller Design

The controller design for the Crazyflie was conducted in 2 parts: 1) state-feedback using the 12 physical states, 2) addition of control using integrator states.

When doing the design for the SGSF controller without integration, the linearized model was used. Since the model was linearized about the hover condition using the state vector given in (7.4) and input ordering in (7.5), this introduces a saw-tooth structure to the resulting K matrix, namely (where x is a non-zero value):

$$K = \begin{bmatrix} 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 \\ 0 & x & 0 & x & 0 & 0 & 0 & x & 0 & x & 0 & 0 \\ x & 0 & 0 & 0 & x & 0 & x & 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

This K matrix can be designed a number of ways, with the most common for quadrotors being using the Linear Quadratic Regulator technique (LQR) (see eg. [3, 46, 47, 48, 49]).

7.2.3.1 LQR Design Methodology

This section contains a brief introduction to the theory of LQR controllers, more in-depth discussions can be found in the control literature (eg. [42, 43] and other textbooks).

The idea behind LQR control is to design a controller to regulate the physical system to a desired state vector (usually the origin) while minimizing a quadratic cost function given by:

$$J = \int x^T Q x + x^T S u + u^T R u \tag{7.7}$$

In this cost function, there are 3 weighting matrices:

Q - Penalizes deviation of the states from the origin (note: $Q \succeq 0$)

R - Penalizes large control inputs (note: $R \succ 0$)

S - Cross term (penalizes state deviation in relation to control effort)

When the infinite-horizon optimal control problem using the cost function in (7.7) is solved, the result is a constant gain matrix, denoted as K.

7.2.3.2 Crazyflie Controller

In this work, the initial state-space controller was designed using an LQR methodology. This controller did not behave well in practice though, with large oscillations on the x and y response, and an inability to hold its position. What followed was iterative selection of Q and R matrices in an attempt to get better physical response. This desgin process proved to be tedious and did not result in good stabilizing controllers.

Instead, the structure of the K matrix was exploited, and hand-tuning of the control gains was done. The hand-tuning was possible because of the saw-tooth structure of the K matrix. This meant that each input corresponded to only a select number of states.

For instance, the u_A input was affected only by 4 states, v, p, y and ϕ. This combination of states can be thought of as a PD controller on the ϕ Euler angle and a PD controller on the y position. So hand-tuning the controller could be done using the rules for hand-tuning a PD controller. This led to the K_s matrix shown below.

$$K_s = \begin{bmatrix} 0 & 0 & -8345 & 0 & 0 & 0 & 0 & 0 & -18000 & 0 & 0 & 0 \\ 0 & 1500 & 0 & 2400 & 0 & 0 & 0 & 5119 & 0 & 8173 & 0 & 0 \\ -1500 & 0 & 0 & 0 & 2340 & 0 & -5000 & 0 & 0 & 0 & 6825 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5952 & 0 & 0 & 0 & 0 & 0 & 40000 \end{bmatrix}$$

The next step was to add integration on the outermost loop, namely integration on the x, y, z, and ϕ states. These constants were chosen by a hand-tuning process as well and ended up being 1000, -1000, 5000 and -500 respectively.

Flight tests with only K_s and integration on the outermost loop still had oscillations appearing on the Euler angles (ϕ and θ), which in turn caused oscillations on the x and y axis. Addition of integral action onto the ϕ and θ states improved the performance by reducing the oscillations on the Euler angles. The addition of these integral terms is similar to what was done with the PID controller in section 7.1. Additionally, this phenomenon was briefly noted in [48], and their results also show improved performance with the additional integral terms. The K_i matrix containing the integral gains is:

$$K_i = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1000 & 0 & 1000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 & 0 & 0 & -1000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -500 \end{bmatrix}$$

While the addition of the ϕ and θ improved performance, their values cannot be found using traditional control design methods on the linearized model (such as the LQR method) and must be found using hand-tuning.

One of the ways that the integral terms can be added into the control design scheme is through augmenting the physical system with new integrator states for the error from hover (assumed zero state for simplicity), ie:

$$A = \begin{bmatrix} A_s & 0 \\ -I_{12\times 12} & 0 \end{bmatrix} \qquad B = \begin{bmatrix} B_s \\ 0 \end{bmatrix}$$

where I is the identity matrix.

Since the parameters derived in chapter 6 did not include the cross-terms, the linearized model becomes pure integrators linking states together (as seen in 7.2.1). This means that the controllability matrix has rank 16, which is rank deficient making the system uncontrollable technically. This is not a problem in practice though for 2 reasons:

1. The 8 “uncontrollable” states are integrating stable states, so their value will be bounded, making them stable
2. The non-linear dynamics are not pure integration, so the non-linear system will be controllable with the new integration states

7.2.4 System Response

To examine the response of the SGSF controller when implemented on the Crazyflie, step inputs were given on the x, y and z axes, and the quadrotor's response was recorded. The response can be seen in figures 7.7, 7.8, and 7.9 respectively. These show a comparison between the SGSF controller and the PID controller from section 7.1.

Examining the response graphs, the SGSF controller responds to the step-input faster, but has more oscillations around the setpoint than the PID does. With the z axes response, the SGSF controller responds faster, with slightly more overshoot, but recovers to the setpoint quicker than the PID.

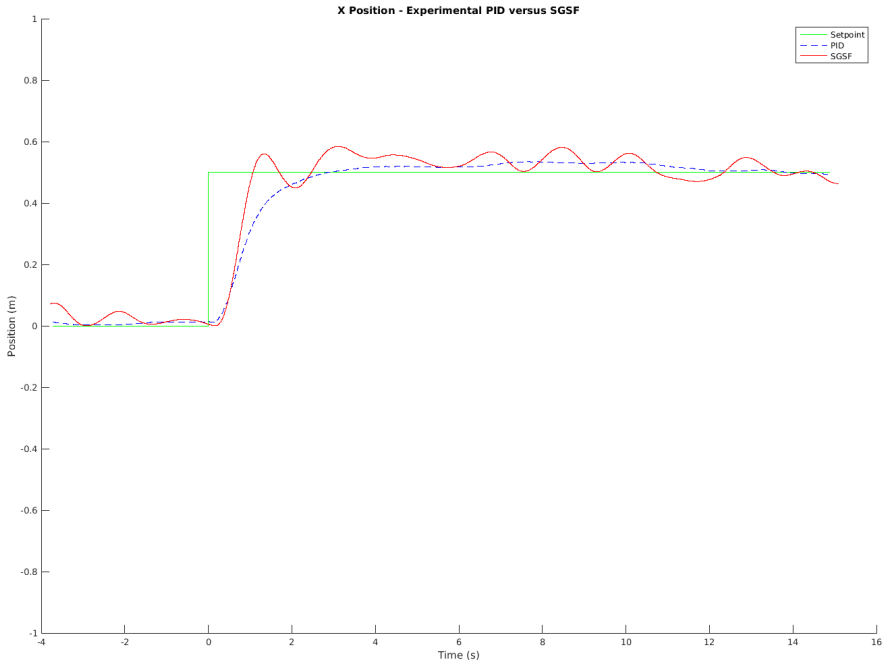


Figure 7.7: Response of the implemented SGSF (red line) and PID (blue line) controllers to a step input (green line) in the x direction.

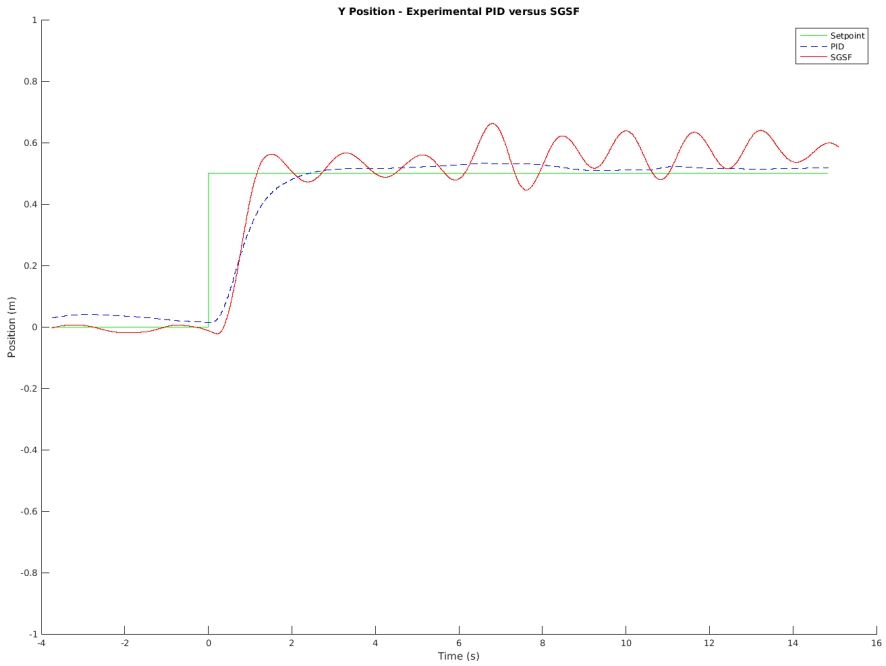


Figure 7.8: Response of the implemented SGSF (red line) and PID (blue line) controllers to a step input (green line) in the y direction.

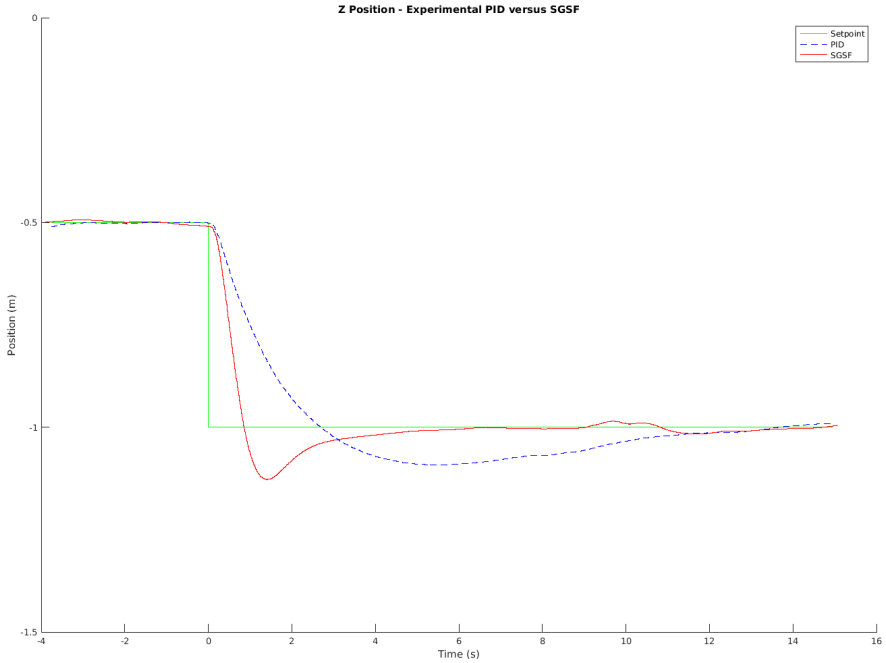


Figure 7.9: Response of the implemented SGSF (red line) and PID (blue line) controllers to a step input (green line) in the z direction.

7.3 Observations

Overall, the controllers developed in this work are capable of stabilizing the position of the Crazyflie. The proposed state space controller was difficult to obtain through normal methods (such as LQR design methodology), so instead the structure of the K matrix was exploited to hand-tune the gains.

One of the reasons the state space controller was difficult to design is due to the small size of the Crazyflie. The small size makes the Crazyflie more susceptible to disturbances, such as brief gusts of air (e.g. from an HVAC system), than a larger quadrotor with more mass would be. Additionally, the LQR controller was difficult to design because the initial weights were not obvious/known. This meant that a lot of time was spent trying out different weights to see their behavior.

An additional part of the Crazyflie that has a drastic effect on the controller performance is the center of mass location. The model used in this work assumes a center of mass located at the origin of the body coordinate system. In actuality, the center of mass of the Crazyflie is offset due to the battery and trackable frame. Before the trackable frame was redesigned to add counterweights, the controllers would have difficulty stabilizing during takeoff. Normally the Crazyflie would shoot to one side, then slowly recover to the desired setpoint. This also introduced asymmetry in the angular movements, with movement in certain directions faster than others. With the redesigned trackable frame, the center of gravity was moved closer to the origin, allowing for smoother takeoffs. However, lingering effects still remain.

In order to overcome the center of mass offset, experimental measurement of its location should be conducted. This can then be fed into the model, and a new linearized system can be developed. This system will not be as nice to work with as the one in this work, since it will contain cross terms between the axes. Additionally, the addition of the center of mass will remove the saw-tooth pattern in the SGSF controller matrix and make K be fully populated.

An additional model parameter to revisit would be the K_H term. In chapter 6, it was believed based upon the experimental PID comparison against the simulation that the model was in good agreement with $K_H = 0$. Future work should revisit that assumption, and use the SGSF controller to determine if a better value for K_H exists.

Finally, some more advanced controllers should be explored, such as the $\mathbf{SO}(3)$ controller proposed by [28]. This controller has been shown to work nicely on the system in [12], so it should provide a good starting point for experiments in advanced control.

CHAPTER 8. COOPERATIVE LOCALIZATION OF A STATIONARY OBJECT USING DISTANCE-ONLY MEASUREMENTS

This chapter contains the derivation, and then the implementation of a novel distributed algorithm for computing the position of a stationary object using distance-only measurements. The theoretical and simulation results were derived in collaboration with Dr. Xu Ma, a former PhD student in our research group at Iowa State University. This algorithm is also presented in [50], along with additional simulation results and comparisons against existing localization algorithms. This chapter also contains results of the experimental implementation of the proposed algorithm.

8.1 Problem Overview

One possible use for a multi-agent system is to gather information on a target object and then estimate its location. There are two main ways this can be accomplished: Triangulation and Trilateration. Triangulation is where the agents sense the relative angle of a received signal, to create a bearing to the object. Then with multiple bearings the agents can determine where the bearings intersect and estimate the position of the target at the intersection point.

Trilateration uses only distance measurements from the agents to the target as its information. This creates a sphere around each agent where the target could be located. By then finding the intersection point of these spheres, the target can be located. As precise as sensors are, all distance measurements will contain noise and inaccuracies, so the intersection point may be either a region or the spheres may never intersect. To deal with this, the trilateration problem is formatted as an optimization problem where the target location is chosen that minimizes the error between the measured and estimated distances.

8.1.1 Original Problem

The method that comes most naturally from the optimization problem is called Range-Based Least Squares (R-LS) is used, which takes the form of the optimization problem in (8.1).

$$\underset{\hat{p}}{\text{minimize}} \quad \sum_{i=1}^{N} (r_i - \|\hat{p} - p_i\|)^2 \tag{8.1}$$

Where $\hat{p}$ is the estimated target location, and p_i, r_i are the ith agent's location and distance the target respectively. This method seeks to find the estimated position that makes the measured distance as close to the actual distance as possible (in the 2-norm sense). This method is also the Maximum-likelihood estimator for the trilateration problem when the distance measurements have additive zero-mean guassian noise [51].

Rewriting (8.1) as a constrained optimization problem produces (8.2). Where a_i has the physical meaning of being the distance between the estimated target location and the measured target location.

$$\begin{aligned} \underset{\hat{x_t},\hat{y_t},\hat{z_t},a_i}{\text{minimize}} \quad & \sum_{i=1}^{N} a_i^2 \\ \text{subject to} \quad & \|\hat{p}_i - p_i\|^2 = (a_i - r_i)^2 \qquad \forall i = 1, 2, \ldots, N \end{aligned} \tag{8.2}$$

In this form it can be seen that the R-LS method produces a non-convex optimization problem, specifically an equality constrained Quadratically-Constrained Quadratic Program.

Since (8.2) takes the form of a QCQP, it can be rewritten as a semidefinite program with the following variable definitions:

$$g_i = \|\hat{p} - p_i\|, \quad X = \begin{bmatrix} \hat{p} \\ 1 \end{bmatrix} \begin{bmatrix} \hat{p}^T & 1 \end{bmatrix}, \quad G = \begin{bmatrix} \hat{g} \\ 1 \end{bmatrix} \begin{bmatrix} \hat{g}^T & 1 \end{bmatrix}, \quad C_i = \begin{bmatrix} I & -p_i \\ -p_i^T & \|p_i\|^2 \end{bmatrix}$$

The semidefinite program version of (8.2) is then given in (8.3).

$$\begin{aligned} \underset{X,G}{\text{minimize}} \quad & \sum_{i=1}^{N} G_{ii} - 2r_i G_{m+1,i} + r_i^2 \\ \text{subject to} \quad & G_{ii} = \mathbf{Tr}(C_i X) \qquad \forall i = 1, 2, \ldots, N \\ & G \succeq 0, X \succeq 0 \\ & G_{m+1,m+1} = X_{4,4} = 1 \\ & \mathbf{Rank}(X) = \mathbf{Rank}(G) = 1 \end{aligned} \tag{8.3}$$

Note that this problem has a rank 1 constraint on the matrices X and G, which is a non-convex constraint. Many methods (eg. [51, 52, 53, 54]) further relax the problem to create a convex problem by simply removing that constraint (which is referred to as a semidefinite relaxation), which produces the Semidefinite Relaxation (SDR) method.

Theoretical results developed in [54] show that the rank of the matrix G will always be 1 for the SDR method, but there exist cases where the rank of matrix X will be greater than 1 (e.g. example 1 in [54]). In the cases where X has rank greater than 1, the solution is no longer exact and is instead an approximate solution, which can be found by doing a rank-one approximation of the matrix X.

8.1.2 Squared Approximation

An alternative method of solving (8.1) is to instead square the ranges, creating what is known as the Squared-Range Least Squares (SR-LS) method, shown in (8.4).

$$\underset{\hat{p}}{\text{minimize}} \quad \sum_{i=1}^{N} \left(r_i^2 - \|\hat{p} - p_i\|^2 \right)^2 \tag{8.4}$$

This problem is also non-convex, but prior results have shown that the form of (8.4) is the same as a generalized trust region subproblem (GTRS). The GTRS problems allow for conditions of optimality to be derived, which led to the creation of a numerical algorithm in [54] that can find the global minimizer of the problem. Note that this problem formulation is no longer a Maximum-likelihood estimator for the target object's position, so with the ability to solve the problem comes a loss of estimation properties.

8.2 Proposed Method

The method we propose is based on the R-LS method in (8.1), but done over a multi-agent networked system where each agent has an estimated position of the target. In order to use this method, the following three assumptions are made:

Assumption 1. *The number of agents should be greater than the dimension of the space being searched.*

Assumption 2. *The agents are distributed in the search space such that they do not form a lower dimensional space (e.g. for a 3D search space the agents are not co-planar).*

Assumption 3. *The communications network is a simple, connected graph.*

Since each agent will have their own estimate of the target's position, three more constraints must be introduced to force the estimates to converge to a single value. These constraints are based on the network's Laplacian matrix L, and when added to (8.1) create (8.5).

$$\begin{aligned} \underset{\hat{x}_t,\hat{y}_t,\hat{z}_t,a_i}{\text{minimize}} \quad & \sum_{i=1}^{N} a_i^2 \\ \text{subject to} \quad & L\hat{x} = 0 \\ & L\hat{y} = 0 \\ & L\hat{z} = 0 \\ & \|\hat{p}_i - p_i\|^2 = (a_i - r_i)^2 \qquad \forall i = 1, 2, \ldots, N \end{aligned} \tag{8.5}$$

8.3 Algorithm Derivation

To derive the algorithm, we start with the minimization problem given in (8.5) and introduce three quadratic penalty terms. These terms will penalize differences in the estimated position of the target across the agents in the network, forcing faster convergence to one solution. The modified problem can be seen in (8.6).

$$\underset{\hat{x}_t,\hat{y}_t,\hat{z}_t,a_i}{\text{minimize}} \quad \sum_{i=1}^{N} a_i^2 + \left(\|\hat{p}_i - p_i\|^2 - (a_i - r_i)^2\right)^2 + k_1 \hat{x}^T L \hat{x} + k_2 \hat{y}^T L \hat{y} + k_3 \hat{z}^T L \hat{z} \tag{8.6a}$$

$$\text{subject to} \quad L\hat{x} = 0 \tag{8.6b}$$

$$L\hat{y} = 0 \tag{8.6c}$$

$$L\hat{z} = 0 \tag{8.6d}$$

$$\|\hat{p}_i - p_i\|^2 = (a_i - r_i)^2 \quad \forall i = 1, 2, \ldots, N \tag{8.6e}$$

Then we define the Lagrangian function for the optimization problem given in (8.6), letting the dual variables be represented by μ, α, β, and γ. By then also expanding the norm function, the Lagrangian in (8.7) is derived. Note that this function is actually the fully augmented Lagrangian function of (8.5).

$$\begin{aligned} L(\hat{x},\hat{y},\hat{z},a,\mu,\alpha,\beta,\gamma) =& k_1\hat{x}^T L\hat{x}+k_2\hat{y}^T L\hat{y}+k_3\hat{z}^T L\hat{z}+\alpha^T L\hat{x}+\beta^T L\hat{y}+\gamma^T L\hat{z} \\ &+\sum_{i=1}^{N}\Big(a_i^2+\left[(\hat{x}_i-x_i)^2+(\hat{y}_i-y_i)^2+(\hat{z}_i-z_i)^2-(a_i-r_i)^2\right]^2 \\ &+\mu_i\left[(\hat{x}_i-x_i)^2+(\hat{y}_i-y_i)^2+(\hat{z}_i-z_i)^2-(a_i-r_i)^2\right]\Big) \end{aligned} \tag{8.7}$$

The original optimization problem can then be solved by finding the solution to the min-max optimization problem in (8.8).

$$\max_{\mu,\alpha,\beta,\gamma}\quad\min_{\hat{x},\hat{y},\hat{z},a_i}\quad L(\hat{x},\hat{y},\hat{z},a,\mu,\alpha,\beta,\gamma) \tag{8.8}$$

To solve this problem, we use the optimization dynamics approach. This approach is based around finding a dynamical system whose equilibrium points (the points where the derivatives of the states are zero) are the solution of the opimization problem. This comes from the Karush-Kuhn-Tucker (KKT) conditions that an optimal point must satisfy. Specifically, The first KKT condition states that

$$\nabla_x L(x,\mu)=0$$

Examining this, it can be seen that this condition can also be interpreted as an equilibrium condition for a dynamical system, where the state variables are the primal variables of the optimization problem. This condition also provides an obvious choice for the dynamical system, namely choose ∇L as the dynamical system.

In order to get convergence of the dynamical system, it is necessary to negate the differential equations for the primal variables, as discussed in [55, 56]. The final system of differential equations is given in (8.9).

$$\dot{\mu}_i = (\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2 + (\hat{z}_i - z_i)^2 - (a_i - r_i)^2 \tag{8.9a}$$

$$\dot{a}_i = 2\left(2\dot{\mu}_i + \mu_i\right)(a_i - r_i) - 2a_i \tag{8.9b}$$

$$\dot{\hat{x}}_i = -2\left(2\dot{\mu}_i + \mu_i\right)(\hat{x}_i - x_i) - e_i^T L\alpha - 2k_1 e_i^T L\hat{x} \tag{8.9c}$$

$$\dot{\hat{y}}_i = -2\left(2\dot{\mu}_i + \mu_i\right)(\hat{y}_i - y_i) - e_i^T L\beta - 2k_2 e_i^T L\hat{y} \tag{8.9d}$$

$$\dot{\hat{z}}_i = -2\left(2\dot{\mu}_i + \mu_i\right)(\hat{z}_i - z_i) - e_i^T L\gamma - 2k_3 e_i^T L\hat{z} \tag{8.9e}$$

$$\dot{\alpha}_i = e_i^T L\hat{x} \tag{8.9f}$$

$$\dot{\beta}_i = e_i^T L\hat{y} \tag{8.9g}$$

$$\dot{\gamma}_i = e_i^T L\hat{z} \tag{8.9h}$$

Prior work by Xu Ma in [55] and [56] has shown that a dynamical system derived from a QCQP problem will converge to the optimal point if the Hessian of the Lagrangian is positive definite at the optimal point. This allows for a convergence theory to be developed.

Theorem 1. *The optimal solution to the problem given in* (8.8) *occurs at an equilibrium point of the dynamical system in* (8.9).

Proof. Any optimal solution for the problem in (8.8) must satisfy the KKT conditions, the first of which is that the derivative of the Lagrangian with respect to the primal variables must be zero at the optimal point:

$$\nabla_x L(x^*, \lambda^*) = 0$$

It can be seen by inspection that equations (8.9b) through (8.9e) are equivalent to the derivatives of (8.7) with respect to the primal variables.

The equilibrium points of (8.9) occur when the derivative terms equal zero, eg. $\dot{\hat{x}}=0$, $\dot{\hat{y}}=0$, $\dot{\hat{z}}=0$, etc. Therefore when (8.9) reaches equilibrium, it is equivalent to the Lagrangian in (8.7) having its primal derivatives equal to zero.

This implies that the optimal solution to (8.8) occurs at an equilibrium point of (8.9).

□

Note: The following lemma was derived with the help of Xu Ma.

Lemma 1. *The Hessian of the augmented Lagrangian (8.7) with respect to the primal variables is positive definite at the equilibrium point.*

Proof. Define the primal variable as $[\hat{x}\ \hat{y}\ \hat{z}\ a]^T$ and let $U = \mathrm{diag}(\mu_1, \mu_2, \ldots, \mu_N)$.

Suppose that the primal equilibrium is $(x^\star, y^\star, z^\star, a^\star)$, and then by algebraic computation, the Hessian of (8.7) evaluated at this equilibrium is given by

$$
\begin{aligned}
H^\star &= 2B + 8D \\
&= 2\begin{bmatrix} B_1 & 0 & 0 & 0 \\ 0 & B_2 & 0 & 0 \\ 0 & 0 & B_3 & 0 \\ 0 & 0 & 0 & B_4 \end{bmatrix} + 8\begin{bmatrix} D_{xx} & D_{xy} & D_{xz} & D_{xa} \\ D_{xy} & D_{yy} & D_{yz} & D_{ya} \\ D_{zx} & D_{yz} & D_{zz} & D_{za} \\ D_{xa} & D_{ya} & D_{za} & D_{aa} \end{bmatrix}
\end{aligned}
$$

where $B_1 = U + k_1 L$, $B_2 = U + k_2 L$, $B_3 = U + k_3 L$, $B_4 = I - U$. Moreover, we have

$$
\begin{aligned}
D_{xx} &= \mathbf{diag}\{(\hat{x}_1^\star - x_1)^2, (\hat{x}_2^\star - x_2)^2, \cdots, (\hat{x}_N^\star - x_N)^2\} \\
D_{yy} &= \mathbf{diag}\{(\hat{y}_1^\star - y_1)^2, (\hat{y}_2^\star - y_2)^2, \cdots, (\hat{y}_N^\star - y_N)^2\} \\
D_{zz} &= \mathbf{diag}\{(\hat{z}_1^\star - z_1)^2, (\hat{z}_2^\star - z_2)^2, \cdots, (\hat{z}_N^\star - z_N)^2\} \\
D_{aa} &= \mathbf{diag}\{(a_1^\star - r_1)^2, (a_2^\star - r_2)^2, \cdots, (a_N^\star - r_N)^2\} \\
D_{xy} &= \mathbf{diag}\{(\hat{x}_1^\star - x_1)(\hat{y}_1^\star - y_1), \cdots, (\hat{x}_N^\star - x_N)(\hat{y}_N^\star - y_N)\} \\
D_{xz} &= \mathbf{diag}\{(\hat{x}_1^\star - x_1)(\hat{z}_1^\star - z_1), \cdots, (\hat{x}_N^\star - x_N)(\hat{z}_N^\star - z_N)\} \\
D_{yz} &= \mathbf{diag}\{(\hat{y}_1^\star - y_1)(\hat{z}_1^\star - z_1), \cdots, (\hat{y}_N^\star - y_N)(\hat{z}_N^\star - z_N)\} \\
D_{xa} &= \mathbf{diag}\{(\hat{x}_1^\star - x_1)(a_1^\star - r_1), \cdots, (\hat{x}_N^\star - x_N)(a_N^\star - r_N)\} \\
D_{za} &= \mathbf{diag}\{(\hat{z}_1^\star - z_1)(a_1^\star - r_1), \cdots, (\hat{z}_N^\star - z_N)(a_N^\star - r_N)\} \\
D_{ya} &= \mathbf{diag}\{(\hat{y}_1^\star - y_1)(a_1^\star - r_1), \cdots, (\hat{y}_N^\star - y_N)(a_N^\star - r_N)\}.
\end{aligned}
$$

Now it is not difficult to verify two points: 1) For those k_1, k_2, k_3 large enough, the positive semidefinite Laplacian L will make the B matrix positive semidefinite. 2) The matrix D is also positive semidefinite since all of its principal minors are positive semidefinite. Therefore, the Hessian $H^\star$ is at least positive semidefinite.

Next we show that $H^\star$ is positive definite. Because of assumption 3 the graph is fully connected, so we know that L has a one-dimensional null space given by $[1\ 1\cdots 1]^T$. As a result, the B matrix has a three-dimensional null space given by

$$\begin{aligned}\textbf{span}\Big\{&[1\ 1\cdots 1, 0\ 0\cdots 0, 0\ 0\cdots 0, 0\ 0\cdots 0]^T,\\ &[0\ 0\cdots 0, 1\ 1\cdots 1, 0\ 0\cdots 0, 0\ 0\cdots 0]^T,\\ &[0\ 0\cdots 0, 0\ 0\cdots 0, 1\ 1\cdots 1, 0\ 0\cdots 0]^T\Big\}.\end{aligned}$$

Checking these vectors one by one, we find that none of them is in the null space of D, which means that those two matrices B and D can compensate the null space of each other. Therefore, we can finally conclude that $H^\star$ is positive definite. □

Theorem 2. *There exist values for $k_1>0$, $k_2>0$, and $k_3>0$ such that if the system in (8.9) starts in a neighborhood around the optimal point (x^*, λ^*), it will converge to (x^*, λ^*).*

Proof. By theorem 7 in [56], a general QCQP problem has convergent optimization dynamics if the Hessian of the associated Lagrangian is positive definite when evaluated at the equilibrium point.

From lemma 1, the Hessian of the augmented Lagrangian in (8.7) at the equilibrium point is positive definite. Therefore by [56], the optimization dynamics in (8.9) will converge to a unique solution when started in a neighborhood around (x^*, λ^*). □

So, using theorem 2, we know that the dynamical system in (8.9) will converge under mild conditions to a point, which according to theorem 1 is the optimal point for (8.6).

8.4 Experimental Setup

An important step in developing an algorithm is experimental implementation and verification of its performance. For this algorithm, that means developing a method to measure the distance between the agent and a target node. There are two main methods for measuring the distance between objects: RF time of flight and acoustic time of flight. Both methods broadcast a signal, then use the propagation time of the signal in the medium (usually air) to determine the distance between the emitter and the sensor.

Bitcraze, the manufacturer of the Crazyflie, has developed a system that relies on RF time of flight for determining the distance between a Crazyflie and a target node. This system was developed as a localization system for the Crazyflie, where multiple nodes would be located throughout the flight area and then the Crazyflie would use the distance to each node to estimate its location in the flight area [57]. In this work, the system is used in reverse. The firmware of the Crazyflie was modified to provide the raw distance data in meters to the computational algorithm.

8.4.1 Distance Sensor

The Crazyflie's distance measurement system is based upon an Ultra-Wideband (UWB) RF transceiver called the Decawave DWM1000. There is one transceiver located on a Crazyflie deck, and another on a standalone board, called the targent node. The Crazyflie with the deck can be seen in figure 2.3b, and the target node can be seen in figure 8.1. Trackables were also attached to the target node to allow for the reading of its position using the camera system, so the algorithm result could be compared against the actual location.

Bitcraze has also released an experimental feature on the localization system allowing for the tags and anchors to operate in a Time Division Multiple Access (TDMA) manner. This allows for multiple Crazyflie decks to measure the distance to a single anchor, by dividing the transmit times into 4ms slots. Each deck is then assigned a specific slot for it to perform its ranging. This is the mode used in these experiments, so that the four Crazyflies can all measure the distance to the target node. Additionally, the firmware was modified to allow the TDMA

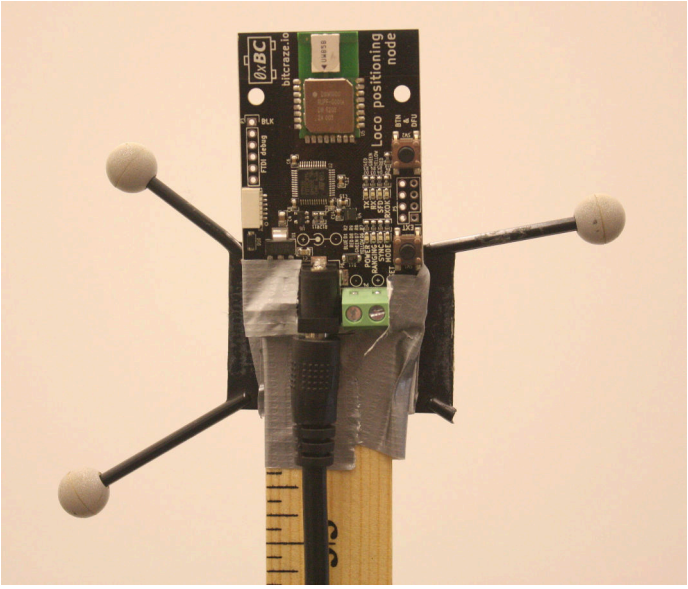


Figure 8.1: Target node used for the trilateration experiments

slot to be selected by software instead of being hardcoded at build time. This allowed for every Crazyflie to receive the same firmware, but then assign a timeslot based upon the agent number it was assigned by the control software.

The advertised accuracy of the localization system using these distance measurements is on the order of 10cm [57]. Experimental data of just the distance measurement system was collected by moving the Crazyflie with a localization deck around the target node in the camera system volume. The actual location of both objects was recorded, so the actual distance could be calculated in the analysis phase. The results of this testing can be seen in figure 8.2. These results show that the error being experienced by a node in this test setup is closer to 30cm, but that the distance measure still tracks the changes in distance. This implies that the sensor has non-zero mean noise (which is similar to the results reported in [58]).

8.4.2 Algorithm Implementation

The system given in (8.9) is a continuous-time dynamical system, while the experimental setup constructed is a digital (sampled-data) system. This means that the system is not natively compatible, and instead must be sampled. In this work, the system was sampled and

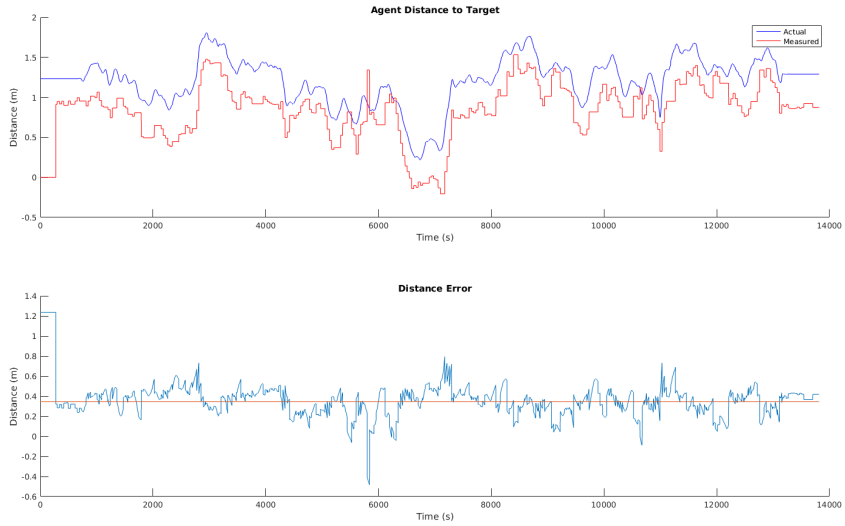


Figure 8.2: Experimental results showing ranging error of localization system

implemented using the Forward-Euler Method. This method of solving a continuous time ODE (given by $\dot{x} = f(x)$) takes the form

$$x_{n+1} = x_n + hf(x_n) \tag{8.10}$$

Where x_{n+1} and x_n are the state values at sample $n+1$ and n respectively, and h is the time interval between the two samples.

This discrete-time algorithm was then implemented in the computational framework developed for the Crazyflies in section 4.2.3. In this algorithm, only 6 variables need to be shared between a node and its neighbors: $\hat{x}_i$, $\hat{y}_i$, $\hat{z}_i$, α_i, β_i, and γ_i. These six variables were placed into a CRTP packet (along with the measured distance, for logging purposes), which was then transmitted to the ground station after every computation. The groundstation was configured to act as a network forwarder, so it would forward the received CRTP packet to all the neighboring agents based upon a predetermined network structure.

Each agent would execute the above discrete-time state update at a rate of 100Hz. This would occur whether or not new data was received, so if no new data was received from a neighboring agent the previously received data would be used for the next update. There was a timeout feature implemented, so that if no new data is received from an agent in 1 second, that agent's data is removed from the computation.

8.5 Experimental Results

To assess the performance of the algorithm, two different tests were run: static tests and flight tests.

8.5.1 Static Test

In the static test, the agents were statically placed on chairs/objects around the target node. A sample setup can be seen in figure 8.3. In that setup, there are 4 agents placed around the target node in the center. The agents are placed such that they do not all lie in a single plane (so that assumption 2 is satisfied).

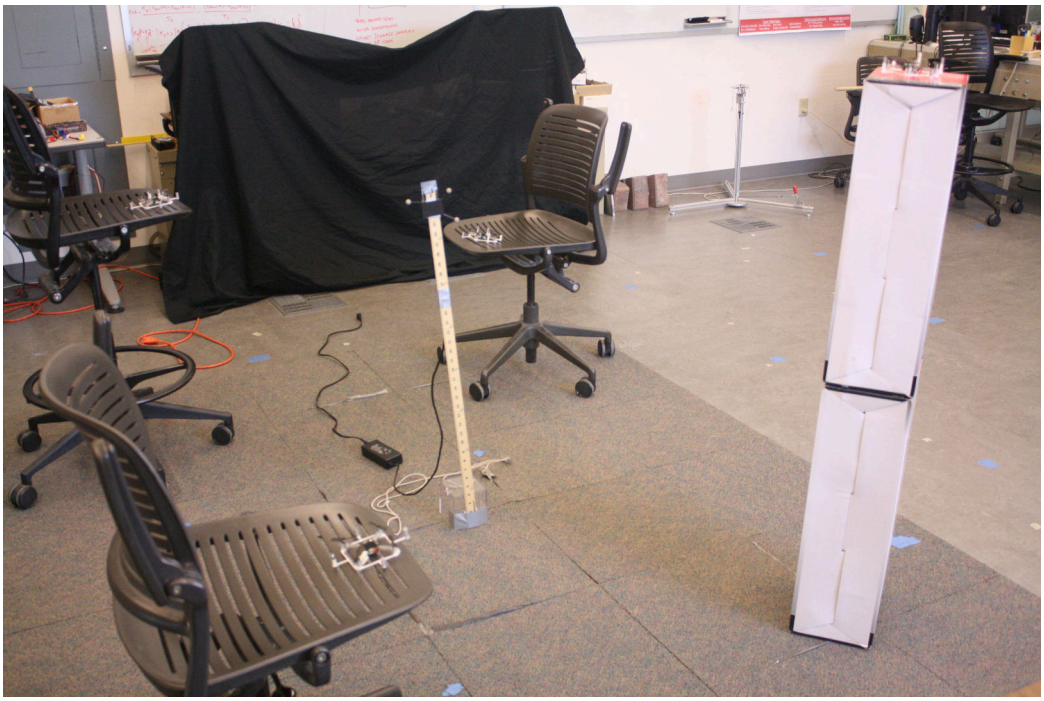


Figure 8.3: Setup used for a static test of the algorithm

The network used in these tests was a ring configuration with the graph Laplacian given in (8.11), and a step size for the Forward-Euler method of h=0.01.

$$L = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix} \tag{8.11}$$

The localization algorithm was then started and allowed to run for approximately 75 seconds. By that time the agents had converged to an estimated position for the target node. Figure 8.4 shows the distances measured by the agents compared with the actual distance as computed from the camera system. Figure 8.5 then shows the estimated positions and figure 8.6 shows the error in the estimated positions.

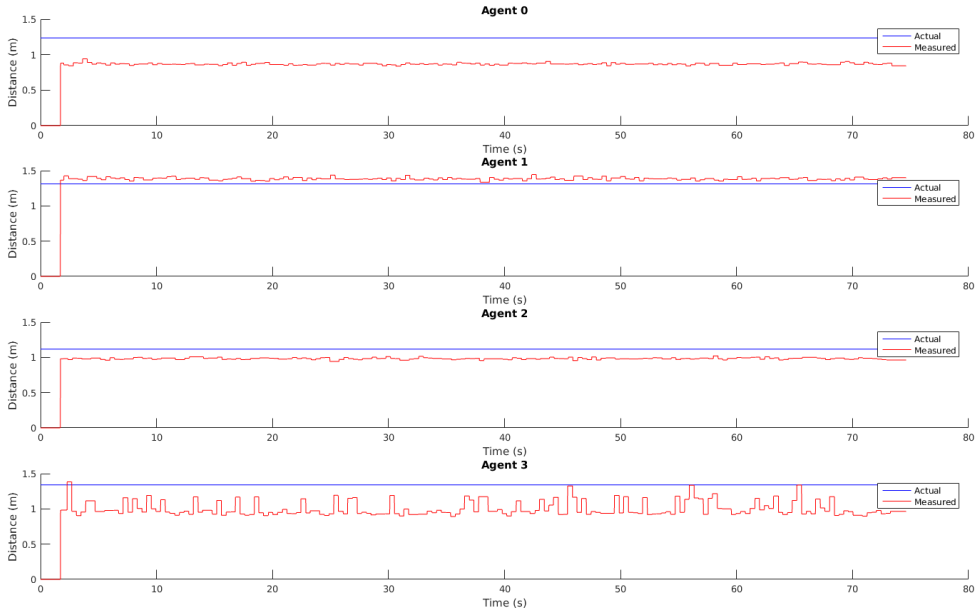


Figure 8.4: Measured versus actual distance for the static test

From these figures, it can be seen that the agents were able to localize the node with an error on the order of 10cm, and were able to converge to a consensus about the estimated position in less than 10 seconds. Note however that the estimate in figure 8.4 is offset from the true measurement. This is caused by the fact that the sensor in use is a biased sensor. When the algorithm is simulated using the actual data from the test but with the sensor bias removed, the resulting estimate is closer to the actual location. The results of this test can be seen in figure 8.7.

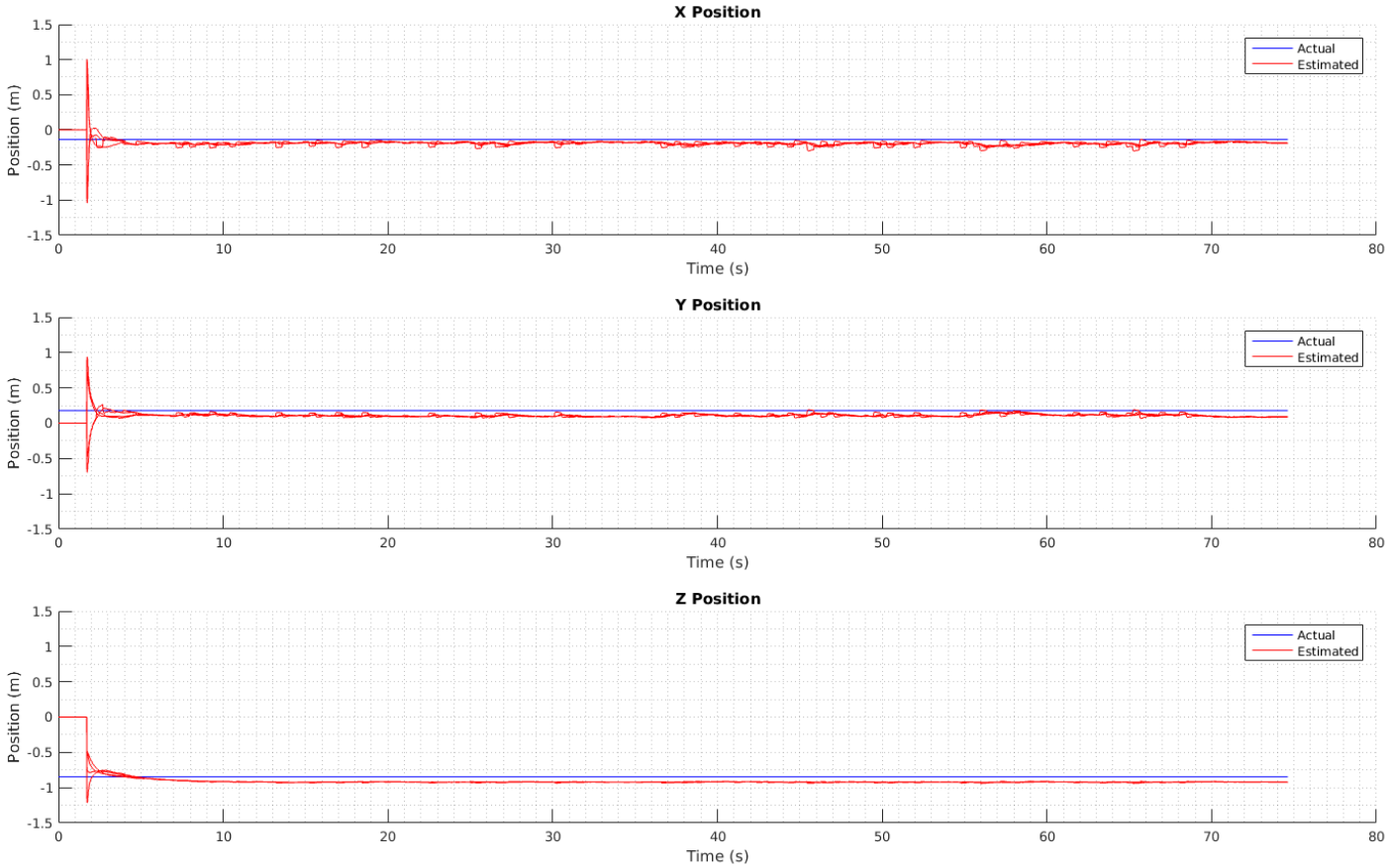


Figure 8.5: Experimental position estimate of the target node from a static test with 4 agents.

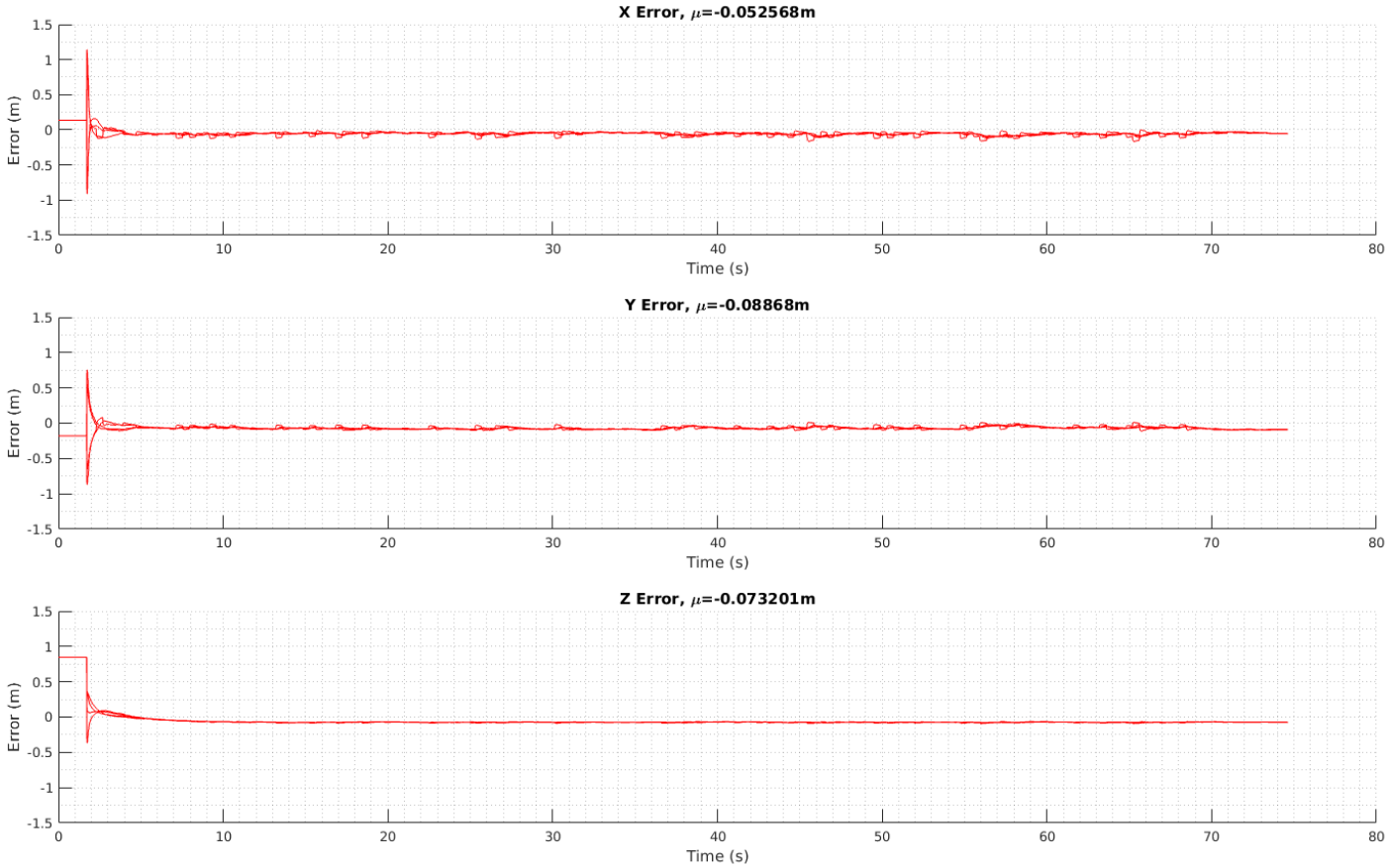


Figure 8.6: Experimental error in position estimates from a static test with 4 agents.

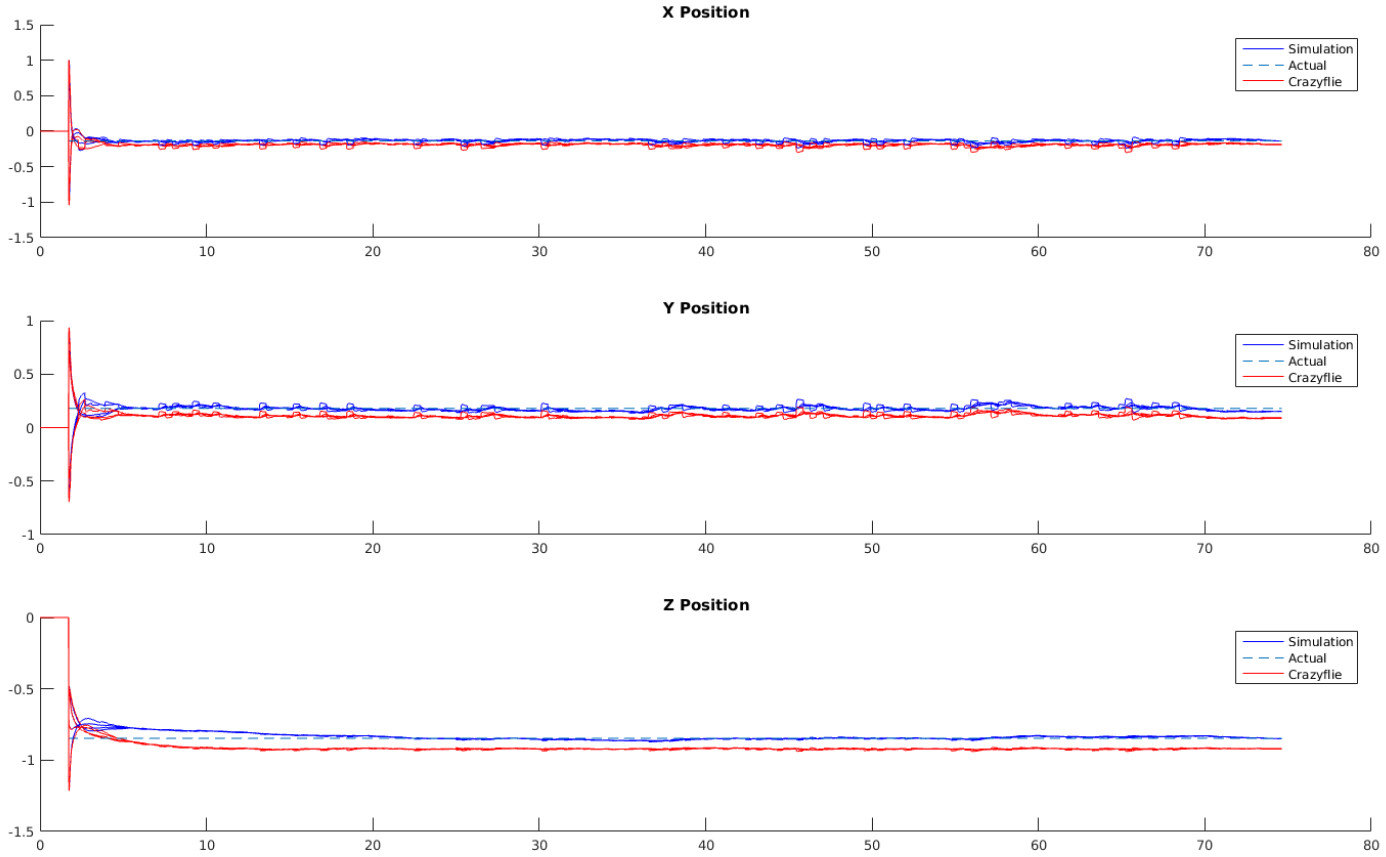


Figure 8.7: Results from a simulation showing estimated location if sensor bias were removed (blue line) for the static test

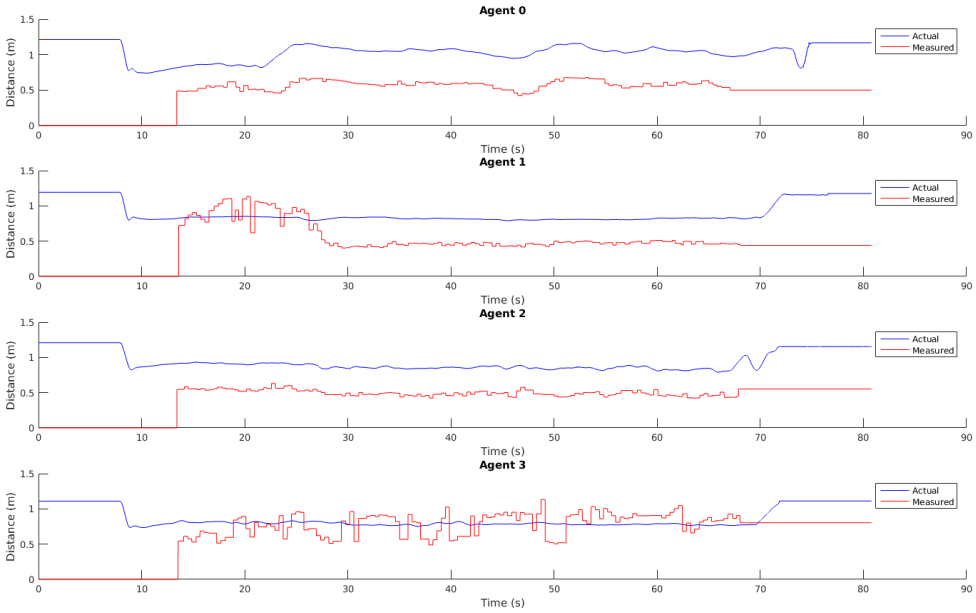


Figure 8.8: Measured versus actual distance for the flight test

8.5.2 Flight Test

After the static test was conducted, flight tests were conducted. These tests involved actually having the agents takeoff, and hover at a predetermined point. While hovering the agents are localizing the target node using the distance measurements. The results of one such experiment with 4 agents is reported here. As seen in figure 8.8, the errors in the distance measurements were slightly worse than the static test measurements. Additionally, during this test run, agent 0 was moved up on the z axes partway through the computation (between 20 and 30 seconds in). Based upon figure 8.8, that change also affected agent 1's measured distance.

The results of the localization can be seen in figures 8.9 and 8.10. The errors in the computed position (shown in figure 8.10) are on the order of 10-12cm, and the step on the z axes for agent 0 seems to have reduced the error by half (from a 25cm error to a 12cm error). The data collected in this test was then fed into the simulated algorithm with the sensor bias removed (similar to the static test). This resulted in the estimate of the location converging closer to the actual location, as seen in figure 8.11.

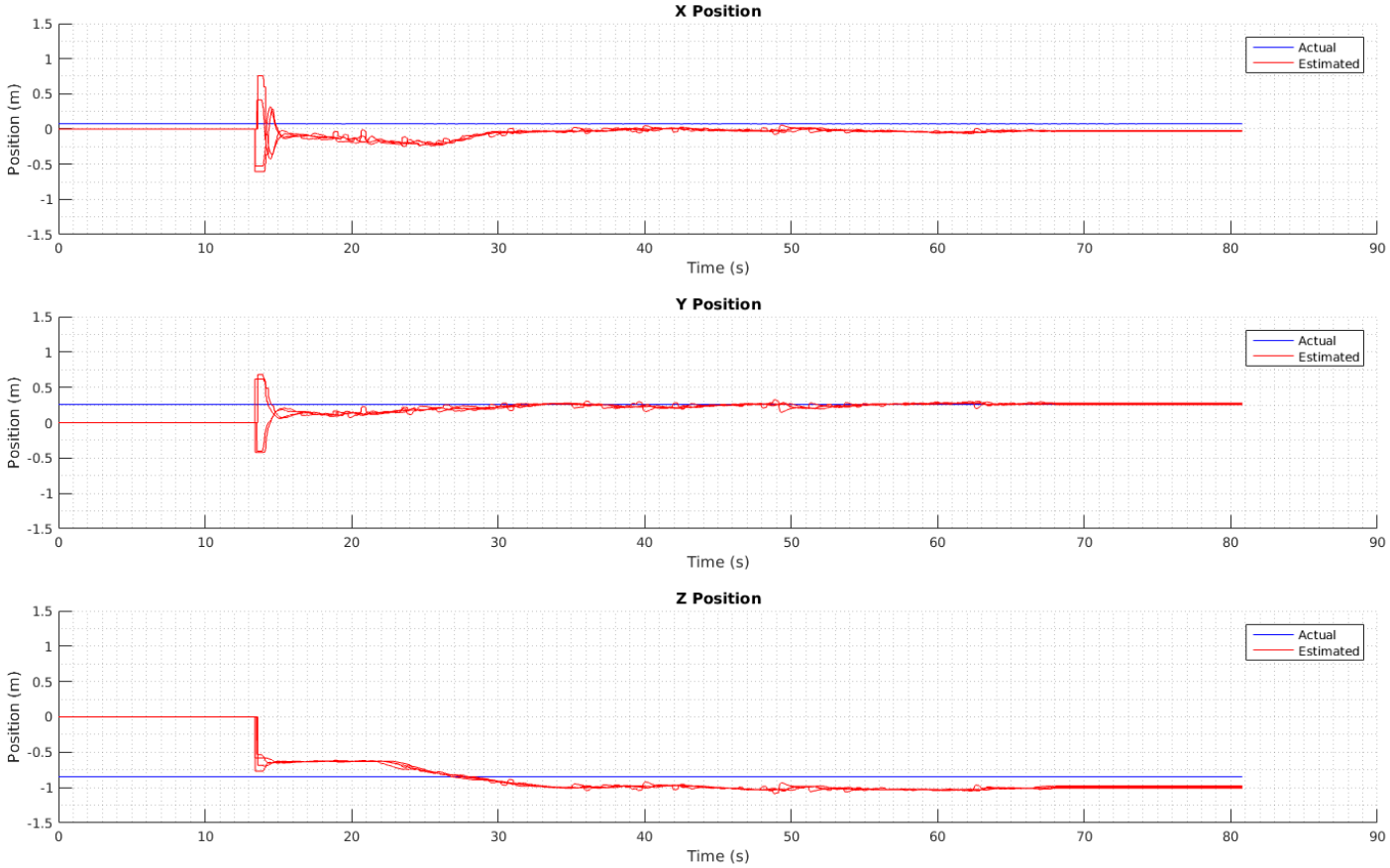


Figure 8.9: Experimental position estimate of the target node for a flight test with 4 agents.

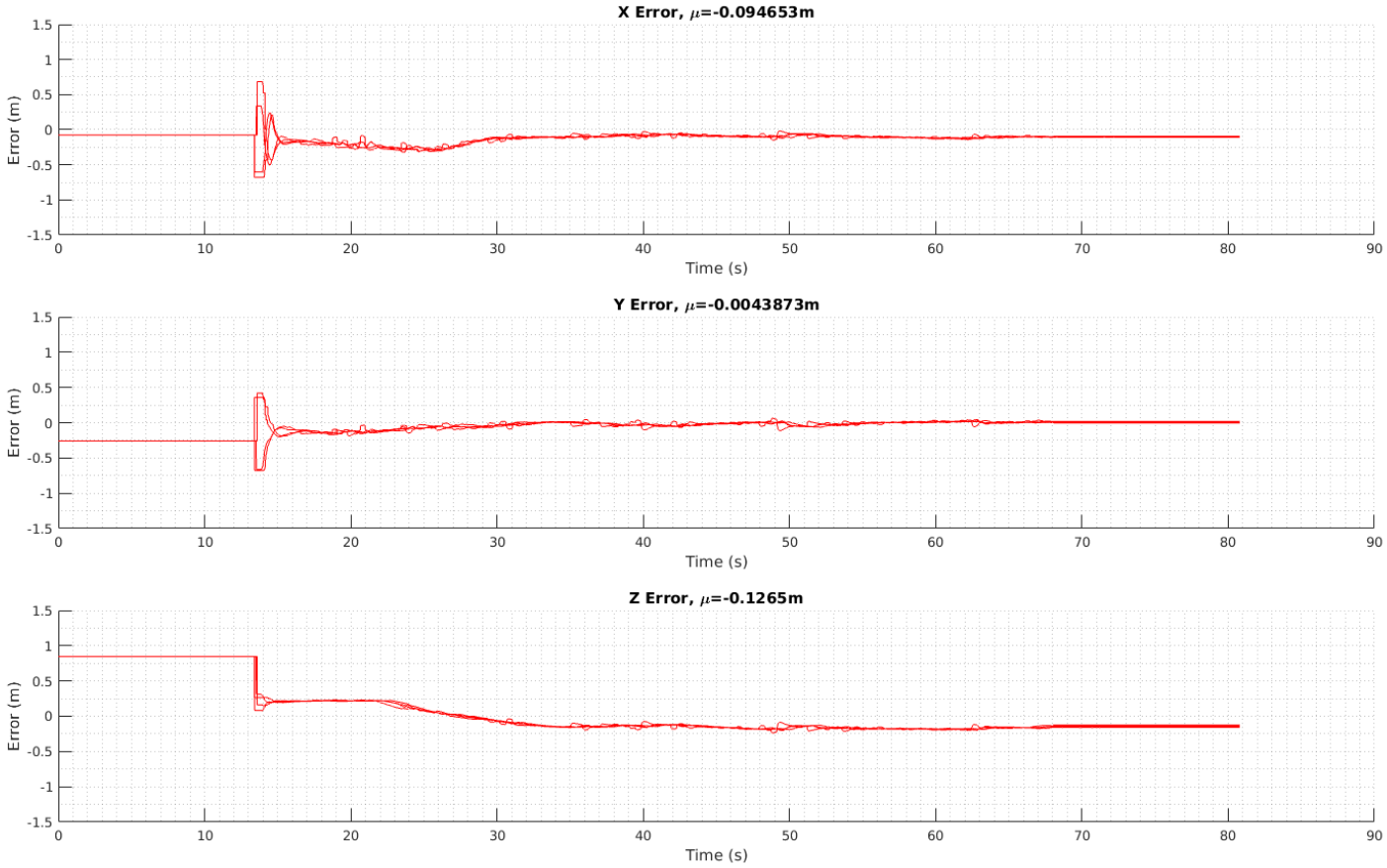


Figure 8.10: Experimental error in position estimates for a flight test with 4 agents.

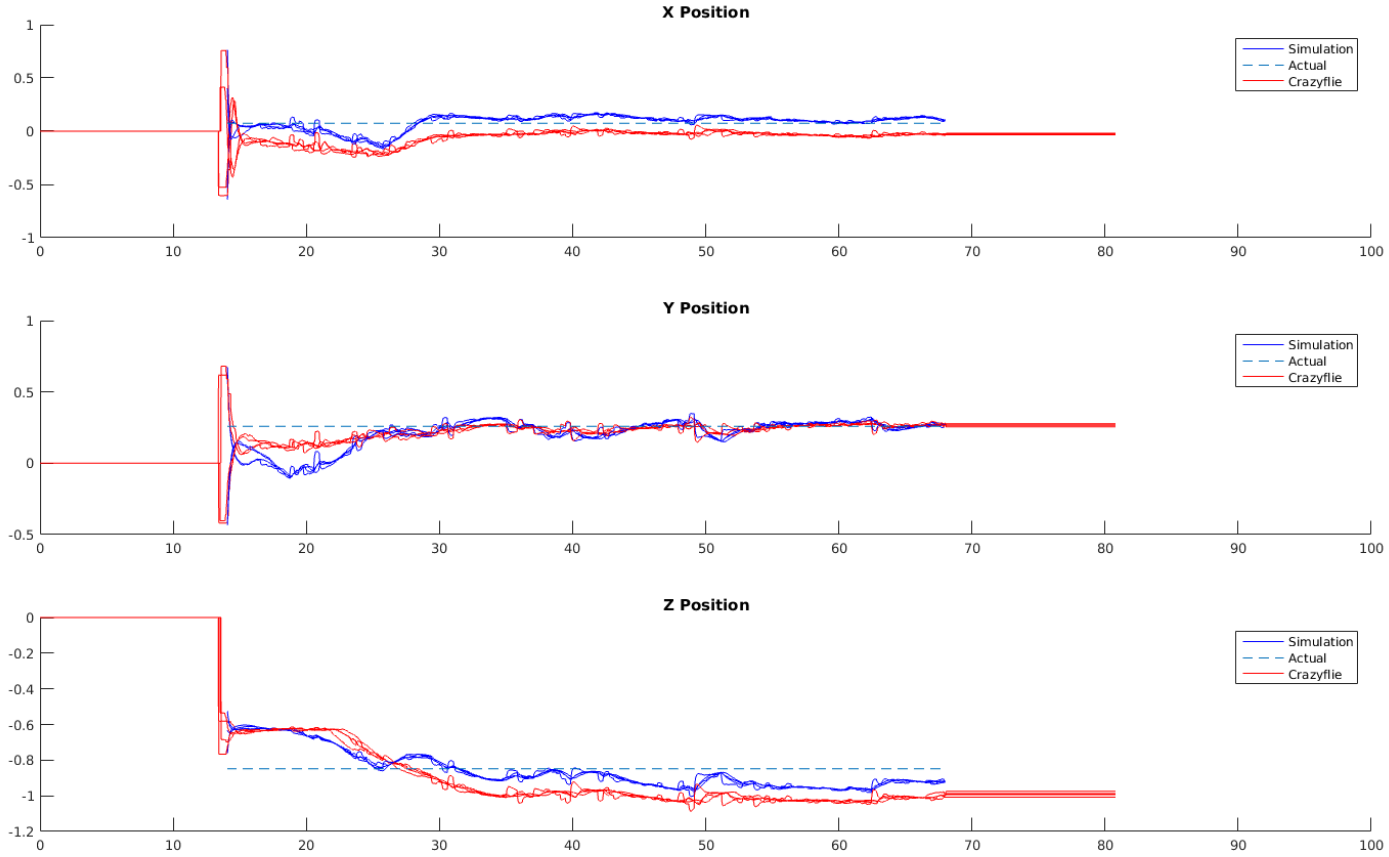


Figure 8.11: Results from a simulation showing estimated location if sensor bias were removed (blue line) for the flight test

8.6 Observations

Based upon all the experimental results presented in this section, it can be seen that the proposed algorithm can localize a target object under real-world conditions. When comparing the two different tests (static versus flight), the static test appears to have a smaller error on the three axes when using the biased sensors, and also has smoother state trajectories. This is caused by the fact that there is less noise in the distributed system, since the positions of the agents are fixed.

When running the flight test, the estimated position is a noisier trajectory, due to the increased noise on the position of the agent. The agent positions were noisier in the flight test because the positional control loop on the Crazyflie's had a degraded communications channel with the camera system. When all of the Crazyflies were passing data on the network, the ground station control loop was slowed down, so the position packets to the Crazyflies were slightly delayed, causing slight oscillations on the Crazyflie position.

As can be seen in the static test (figure 8.7), the majority of the bias on the estimated position in this test system is caused by the bias in the sensor measurements. This shows that this algorithm needs an unbiased sensor, or a sensor with a method for removing its bias. While the current experimental setup is sufficient for these tests, future work should be done to develop a distance measurement system with less measurement bias.

Another avenue to explore to remove the bias in the estimated positon is to examine how the bias changes when more agents are added to the computation. Adding more agents spread across the search space could theoretically reduce the effect of the measurement bias by moving the centroid of the error region closer to the actual location. This experimental system is currently limited by the amount of actual flight space available, so adding more agents will be difficult. Expanding the flight area, and developing better communications strategies (such as the Wifi network) will allow for this to be tested.

Overall, the algorithm presented is an effective way to localize a target object using a swarm of agents with distributed computational capability. As these experiments showed though, this algorithm is extremely sensitive to measurement bias with the low number of agents tested here.

CHAPTER 9. CONCLUSION

9.1 Summary

This thesis presents a system capable of being used for researching aerial multi-agent systems that need to perform distributed computations. Crazyflie quadrotors are used as the aerial agents, with modified firmware to allow for them to control their own position using the external camera system, as well as perform on-board computations and share the results with their neighboring agents. This system utilizes a ground control computer running custom C++ software to provide position updates and setpoint commands to each agent. Additionally, to overcome the current limitations of the Crazyflie communications system the ground control computer is capable of handling multiple CrazyRadios and also doing the message routing between Crazyflie's for the distributed computation.

Various physical parameters of the Crazyflie were measured and calculated in chapter 6, and then the model from [3] was used to derive a physics simulation of the system. As seen by comparing the model's response to that of the quadrotor flying with the nested-loop PID architecture, the measured physical parameters allow for accurate modeling of the quadrotor system. The model and parameters allow for more advanced controllers to be designed, such as the state-space controllers in section 7.2.

Finally, a distributed algorithm to perform object localization using only distance measurements was derived. This algorithm solves a non-convex QCQP using the optimization dynamics approach. In this algorithm, each agent has a local estimate of the position as well as the estimates from its neighbors. Using that information, the agent computes an update to its estimate to be shared over the network with its neighbors. Laboratory experiments showing the convergence and implementation of this algorithm were carried out. These experiments

utilized 4 agents equipped with the Bitcraze Loco Positioning System hardware for distance measurement using RF time of flight. These experiments showed algorithm convergence to a solution near the actual one under non-ideal sensor conditions and better convergence under ideal sensor conditions.

9.2 Future Work

While this system is operational and can be used for demonstrating and refining distributed algorithms (such as the one presented in chapter 8), there are still improvements that could be made to it, namely:

1. Implement mesh-networking in the Crazyflie swarm
2. Explore more advanced controllers, such as the large-angle controller in [28] coupled with more advanced $\mathbf{SO}(3)$ estimators such as those examined in [59]
3. Allow the swarm to contain more agents by implementing methods to allow the camera system to track similar constellations
4. Migrate the ground control software to run on the Robot Operating System (ROS)

Further work can also be done on the localization algorithm proposed in chapter 8, namely:

1. Conduct experiments with a larger number of agents (to explore network connectivity and measurement bias)
2. More closely examine the effect of k_1, k_2 and k_3 on the convergence of the algorithm, and determine a way of choosing the constants
3. Determine the best initial conditions to start the algorithm with to guarantee the largest region of convergence

APPENDIX A. CRAZYFLIE FIRMWARE CONTROL LOOP INCONSISTENCIES

During the modeling process, two discrepancies were noted when comparing measured datasets (such as the dataset given in figure A.1) from the system in [1]:

- The internal quadcopter roll corresponded with the camera system pitch (and the quadcopter pitch with the camera system roll)
- The internal quadcopter roll was negated compared to the camera system pitch

A thorough examination of the Crazyflie firmware revealed 5 inconsistencies compared to the expected operation of a quadroto control loop[1], and an additional two inconsistencies in the operation of the ground station control software from [1].

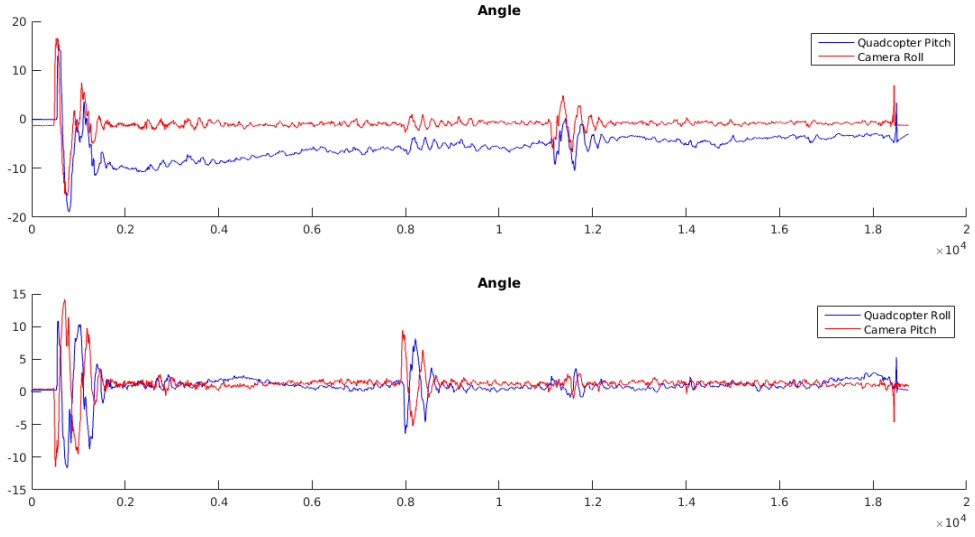


Figure A.1: Comparison of the captured angles using the system developed in [1].

[1]This discussion focuses on the control loop found in the 2016.09 release of the Crazyflie firmware, available at https://github.com/bitcraze/crazyflie-firmware/releases/tag/2016.09

The inconsistencies are listed below, and are highlighted in red and numbered in figure A.2. Numbers 1-5 are located in the Crazyflie firmware, while number 6 and 7 are located in the control software from [1].

Inconsistencies:

1. Body axes used for filter computation is not aligned with the aerospace coordinate system
2. The vector accelerometer inputs to the Mahoney filter are feeding in the negative of the gravitational vector
3. The computed pitch angle is negated
4. The yaw rate PID output is negated before entering the mixer
5. The q body-rate measurement is negated before being used in the q rate PID computation
6. The measured yaw angle is negated before transmission to the Crazyflie
7. The x positon PID controls the roll angle, and the y positon PID controls the pitch angle

The Crazyflie is able to be flown using the software and firmware containing these issues, however the accurate modeling of the firmware and physics system is very difficult with them. After examining the issues, 1 and 2 appear to be the reason for the existence of 3-5 in the firmware and 6 in the software. The last one, 7, is caused by an incorrect definition of 0° yaw when defining the flight system.

What follows is a brief description of each inconsistency and the change made to the firmware to work around it.

Inconsistency 1: Incorrect Body Axes

Most aerospace systems utilize an inertial axes system called the aerospace axes. These axes are a right-handed coordinate system with the positive x axis pointing towards the front, and the positive z axis pointing down towards the earth [60]. This means that estimators designed for aerospace applications are designed to determine the orientation of the craft with respect to the aerospace axes. Since the Mahoney filter estimator used on the Crazyflie's is designed for estimating the attitude of a UAV, it assumes that the body axes system will align with the inertial axes if no rotation has occured [26].

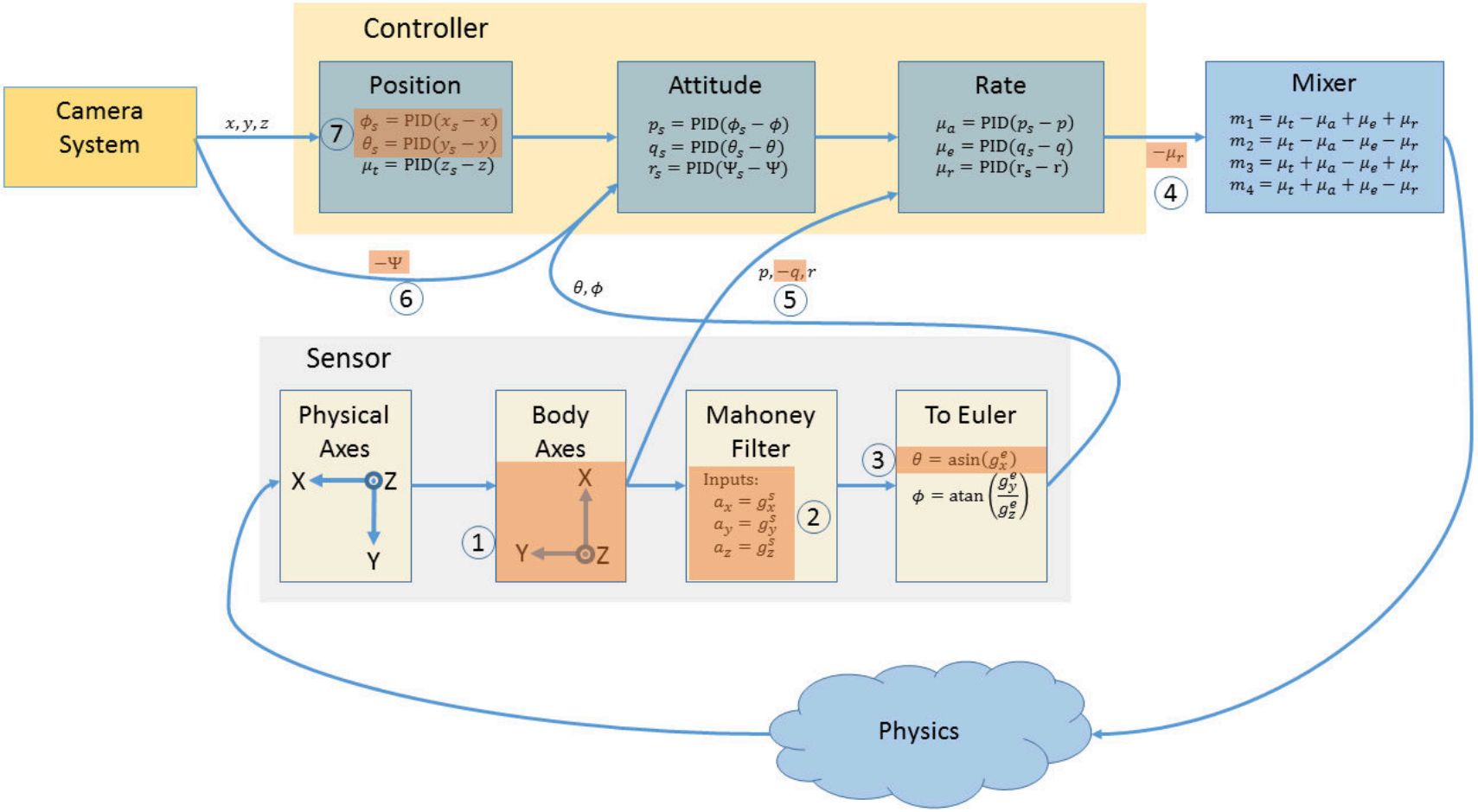


Figure A.2: Mathematical structure of the default firmware's control loop with inconsistencies labeled

On the Crazyflie, these frames are further complicated by the fact that the sensor is in its own frame with the positive x axis pointing left and positive z pointing up. In order to do the angle estimation, the sensor measurements need to be converted into the body frame. The way this is done in the main Crazyflie firmware release is to swap the x and y readings, then negate the x reading, while leaving z alone. eg.

$$x^b = -y^s \qquad y^b = x^s \qquad z^b = z^s \qquad \text{(Incorrect conversion)}$$

This actually moves the sensor readings into a coordinate system where the positive x axis is to the front and the positive z axis is up. This axes system is the aerospace system rotated with a roll of 180° (or equivalently a yaw of 180° followed by a pitch of 180°).

The correct method of converting the sensor frame into the body frame would be to negate all the sensor readings, then swap the x and y readings. eg.

$$x^b = -y^s \qquad y^b = -x^s \qquad z^b = -z^s \qquad \text{(Correct conversion)}$$

Inconsistency 2: Incorrect Gravitational Vector

As discussed in section 5.2.1, the Crazyflie utilizes a Mahoney filter for its attitude estimation by default. This filter assumes that the inertial axes is the aerospace system, and that all sensor readings are in that coordinate system.

As input, the Mahoney filter takes in the sensed gravitational vector g^b and the sensed angular rates about the body axes. Note, this means that in the aerospace system, $g^b = [\,0\;0\;1\,]$ when the quadcopter is not rotated from the inertial frame. However an accelerometer actually senses an acceleration vector of $a = [\,0\;0\;-1\,]$ when the sensor axes are perfectly aligned with the inertial axes [27]. Therefore it is necessary to negate all the accelerometer readings before providing them to the Mahoney filter.

In the 2016.09 release version, the raw accelerometer values in the body axes were passed directly into the Mahoney filter, instead of negating the raw accelerometer values.

$$g^b = a^b \qquad \text{(Incorrect)}$$

$$g^b = -a^b \qquad \text{(Correct)}$$

Inconsistency 3: Negated Pitch Angle

The attitude estimated by the Mahoney filter is natively in a quaternion representation, while the control loops require the attitude in its Euler angle form. This means that the two must be converted. The code does the conversion for pitch using the following formula:

$$\sin^{-1}(2(q_1q_3 - q_0q_2)) \qquad \text{(Incorrect conversion)}$$

This formula is missing a negative sign though. It should instead be:

$$\sin^{-1}(-2(q_1q_3 - q_0q_2)) \qquad \text{(Correct conversion [60])}$$

Inconsistency 4: Negated Yaw Mixer Command

After the r rate PID output, there is a negative sign on the μ_r command before it enters the mixer. This sign is reversing the control input, so a positive PID constant and positive error will actually produce a negative yaw command. In normal operation, a positive error and a positive PID constant should produce a positive yaw command. This negative sign should not be present between the PIDs and the mixer.

This negative sign is necessary because the shift from the sensor axes to the body axes frame does not negate the gyroscope readings on the z axes (the measurement producing the r rate measurement). This lack of negation means the controller should be negated, which this sign change is doing.

Inconsistency 5: Negated q Rate Measurement

In the control loop, the q rate measurement from the gyroscope is negated before being fed into the PID for computing the error.

This negative sign is necessary because the shift from the sensor axes to the body axes frame does not negate the gyroscope readings on the y axes (the measurement producing the q rate measurement). This lack of negation could be handled three ways: negating the mixer command (like inconsistency 4 did), negate the error input, or negate the PID constants. In this case, the input received from the Pitch angular controller is already negated (due to inconsistency 3), so the error negation is what was done. To finish negating the error, the sensed value was negated.

Inconsistency 6: Negated Yaw Angle

The control software developed in [1] utilized the camera system for the yaw angle instead of the internal yaw computation on the Crazyflie. Before the yaw measurement was sent to the PID for controlling the yaw angle, it was negated.

Since the camera is used as the sensor, the controlled yaw angle is relative to an aerospace axes system on the Crazyflie, whereas the Crazyflie had an axes system rotated 180° from that. This means that the two readings would differ in sign, making a negative sign necessary in order not to destabilize the r rate loop (since the rate loop already had a negated measurement due to inconsistency 1 and the output was negated in inconsistency 4, the PID required the reference input to be negated so that all the negations would cancel).

Inconsistency 7: Swapped Position PID Angle Outputs

In the control software, the PIDs for the x and y axes were computed using the camera system measurements (so they were in the camera system reference frame), while the internal PIDs for the roll and pitch were computed using the internal measurements (so they are in the Crazyflie reference frame). Normally the x PID will control the pitch angle and the y PID will control the roll angle. In this case however, they were reversed. The x PID controlled the roll angle and the y PID controller the pitch angle.

This is due to a rotation in the orientation of the Crazyflie's when physically flown. When aligned inside the camera system, the front of the Crazyflie's was facing the negative y camera axes. This made a positive roll command move the quadcopter in positive x and a positive pitch command move the quadcopter in positive y.

APPENDIX B. LOCALIZATION ALGORITHM SOURCE CODE

The following source code snippet is the source code used to implement the algorithm from chapter 8 on the Crazyflie firmware for the distributed computation. Note the following:

- This source code uses the FreeRTOS task tick counter as a clock source for timing information
- The data structure *localData* contains the primal and dual variables for this agent
- The array *nodeData* contains the primal and dual variables received from other agents
- When new data is received, the time in *lastNodeTime* is updated

Source code for implementing the algorithm:

```
// The number of nodes in the computation
float numComputingNodes = 0;

// Go through and make sure all the data is valid
uint32_t currentTick = xTaskGetTickCount();
for ( int i = 0; i < MAX_NUM_NODES; i++ ) {
  float timeDelta = currentTick - lastNodeTime[i];
  if ( (timeDelta > M2T(1000)) && (useNode[i] != 0) ) {
    // If the data is over 1 second old, it is old data
    // and the node should not be used for calculation
    useNode[i] = 0;
    DEBUG_PRINT("Node %d has gone stale\n", i);
  }

  numComputingNodes += (float) useNode[i];
}

localData.measuredRadius = ranges[anchorNumber];

// Compute some differences used a lot in the computation
float diffX = (localData.estimatedX - localPosition.x);
float diffY = (localData.estimatedY - localPosition.y);
float diffZ = (localData.estimatedZ - localPosition.z);
float diffA = (a - localData.measuredRadius);

// This term is needed by the others
dmu = (diffX * diffX ) + (diffY * diffY) + (diffZ * diffZ) - (diffA * diffA);

// This term is used by almost every other derivative
float pd = (2.0*dmu + mu);
```

```
da = 2.0 * pd * diffA - 2.0*a;

// These are based upon data from the other nodes
dx = -2.0 * pd * diffX
     - numComputingNodes*localData.alpha
     - twok1 * numComputingNodes*localData.estimatedX;
dy = -2.0 * pd * diffY
     - numComputingNodes*localData.beta
     - twok2 * numComputingNodes*localData.estimatedY;
dz = -2.0 * pd * diffZ
     - numComputingNodes*localData.gamma
     - twok3 * numComputingNodes*localData.estimatedZ;

dalpha = numComputingNodes*localData.estimatedX;
dbeta  = numComputingNodes*localData.estimatedY;
dgamma = numComputingNodes*localData.estimatedZ;

// Go through and compute the updates
for (int i = 0; i < MAX_NUM_NODES; i++ ) {
  // Make sure the node should be used
  if (useNode[i] != 0) {
    dx += ( nodeData[i].alpha + twok1*nodeData[i].estimatedX );
    dy += ( nodeData[i].beta  + twok2*nodeData[i].estimatedY );
    dz += ( nodeData[i].gamma + twok3*nodeData[i].estimatedZ );

    dalpha -= nodeData[i].estimatedX;
    dbeta  -= nodeData[i].estimatedY;
    dgamma -= nodeData[i].estimatedZ;
  }
}

// Update the local information (simple Euler's method)
a  += (stepSize * da);
mu += (stepSize * dmu);

// Update the information to share (simple Euler's method)
localData.estimatedX += (stepSize * dx);
localData.estimatedY += (stepSize * dy);
localData.estimatedZ += (stepSize * dz);

localData.alpha += (stepSize * dalpha);
localData.beta  += (stepSize * dbeta);
localData.gamma += (stepSize * dgamma);
```

BIBLIOGRAPHY

[1] J. Noronha, "Development of a swarm control platform for educational and research applications," Master's Thesis, Iowa State University, 2016.

[2] Crazyflie 2.0 system architecture. Bitcraze Wiki. [Online]. Available: https://wiki.bitcraze.io/projects:crazyflie2:architecture:index

[3] M. Rich, "Model development, system identification, and control of a quadrotor helicopter," Master's Thesis, Iowa State University, 2012.

[4] M. Gopabhat Madhusudhan, "Control of Crazyflie nano quadcopter using Simulink," Master's Thesis, California State University, Long Beach, 2016.

[5] B. Landry, "Planning and control for quadrotor flight through cluttered environments," Master's Thesis, Massachusetts Institute of Technology, 2015.

[6] J. Forster, "System Identification of the Crazyflie 2.0 Nano Quadrocopter," Bachelor's Thesis, ETH Zurich, 2015.

[7] J. P. How, B. Bethke, A. Frank, D. Dale, and J. Vian, "Real-time indoor autonomous vehicle test environment," *IEEE Control Systems Magazine*, vol. 28, no. 2, pp. 51–64, 2008.

[8] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "The GRASP multiple micro-UAV testbed," *IEEE Robotics and Automation Magazine*, pp. 56–65, 2010.

[9] A. Kushleyev, D. Mellinger, C. Powers, and V. Kumar, "Towards a swarm of agile micro quadrotors," *Autonomous Robots*, vol. 35, no. 4, pp. 287–300, 2013.

[10] S. Lupashin, A. Schöllig, M. Hehn, and R. D'Andrea, "The flying machine arena as of 2010," in *2011 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011.

[11] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea, "A platform for aerial robotics research and demonstration: The Flying Machine Arena," *Mechatronics*, vol. 24, no. 1, pp. 41–54, 2014.

[12] J. A. Preiss, H. Wolfgang, G. S. Sukhatme, and N. Ayanian, "Crazyswarm: A Large Nano-Quadcopter Swarm," in *2017 International Conference on Robotics and Automation (ICRA)*. Singapore: IEEE, 2017.

[13] M. Furci, G. Casadei, R. Naldi, R. G. Sanfelice, and L. Marconi, "An open-source architecture for control and coordination of a swarm of micro-quadrotors," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2015, pp. 139–146.

[14] R. Oung, F. Bourgault, M. Donovan, and R. D'Andrea, "The Distributed Flight Array," in *2010 IEEE International Conference on Robotics and Automation*. Anchorage, AK: IEEE, may 2010.

[15] D. Brescianini and R. D'Andrea, "Design, modeling and control of an omni-directional aerial vehicle," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2016, pp. 3261–3266.

[16] P. J. Besl and N. D. McKay, "A method for registration of 3-D shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 239–256, feb 1992.

[17] J. Winkler. libcflie. [Online]. Available: https://github.com/fairlight1337/libcflie

[18] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser, "VRPN: a device-independent, network-transparent VR peripheral system," in *VRST '01 Proceedings of the ACM symposium on Virtual reality software and technology*. Baniff, Alberta, Canada: ACM, 2001, pp. 55–61.

[19] OptiTrack - Flex 3. OptiTrack. [Online]. Available: http://optitrack.com/products/flex-3/

[20] Freertos - market leading rtos for embedded systems. Real Time Engineers Ltd. [Online]. Available: http://www.freertos.org/

[21] P. Florence. (2015) Crazyflie CAD files. [Online; accessed January 27, 2017]. [Online]. Available: https://github.com/peteflorence/crazyflie-CAD

[22] C. Triola, "Special Orthogonal Groups and Rotations," Honors Thesis, University of Mary Washington, 2009. [Online]. Available: http://files.umwblogs.org/blogs.dir/4710/files/2010/10/honors_triola.pdf

[23] MPU9250 | InvenSense. InvenSense. [Online]. Available: https://www.invensense.com/products/motion-tracking/9-axis/mpu-9250/

[24] M. W. Mueller, M. Hamer, and R. D. Andrea, "Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation," in *International Conference on Robotics and Automation*, 2015, pp. 1730–1736.

[25] S. O. Madgwick, A. J. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm," in *2011 IEEE International Conference on Rehabilitation Robotics*. Zurich: IEEE, jun 2011, pp. 1–7.

[26] M. Euston, P. Coote, R. Mahony, J. Kim, and T. Hamel, "A complementary filter for attitude estimation of a fixed-wing UAV," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 2008, pp. 340–345.

[27] M. Pedley, "Tilt Sensing Using a Three-Axis Accelerometer," Freescale Semiconductor, Tech. Rep., 2013. [Online]. Available: http://www.nxp.com/assets/documents/data/en/application-notes/AN3461.pdf

[28] D. Mellinger, "Trajectory Generation and Control for Quadrotors," PhD Dissertation, University of Pennsylvania, 2012.

[29] CS Series. Ohaus. [Online]. Available: http://www.ohaus.com/en-US/CSSeries

[30] 4 x 7 mm dc-motor pack for crazyflie 2.0. Bitcraze. [Online]. Available: https://store.bitcraze.io/collections/spare-parts-crazyflie-2-0/products/4-x-7-mm-dc-motor-pack-for-crazyflie-2

[31] Reading the motor constants from typical performance characteristics. Precision Microdrives. [Online]. Available: https://www.precisionmicrodrives.com/tech-blog/2014/02/02/reading-motor-constants-typical-performance-characteristics

[32] G. P. Subramanian, "Non-Linear Control Strategies for Quadrotors and Cubesats," Master's Thesis, University of Illinois at Urbana-Champaign, 2015.

[33] *Manual for the Model 220: Industrial Emulator / Servo Trainer (Instructor's Edition)*, Educational Control Products, 1995.

[34] J. Spurr, S. Goodwill, J. Kelley, and S. Haake, "Measuring the inertial properties of a tennis racket," in *Procedia Engineering*. Elsevier, 2014, pp. 569–574.

[35] A. R. Kim, P. Vivekanandan, P. McNamee, I. Sheppard, A. Blevins, and A. Sizemore, "Dynamic Modeling and Simulation of A Quadcopter with Motor Dynamics," in *AIAA Modeling and Simulation Technologies Conference*. AIAA, 2017.

[36] M. R. Jardin and E. R. Mueller, "Optimized Measurements of UAV Mass Moment of Inertia with a Bifilar Pendulum," in *AIAA Guidance, Navigation and Control Conference and Exhibit*. AIAA, 2007.

[37] ——, "Optimized Measurements of Unmanned-Air-Vehicle Mass Moment of Inertia with a Bifilar Pendulum," *Journal of Aircraft*, vol. 46, no. 3, pp. 763–775, 2009.

[38] Dual-Range Force Sensor. Vernier Software and Technology. [Online]. Available: https://www.vernier.com/products/sensors/force-sensors/dfs-bta

[39] LoggerPro 3. Vernier Software and Technology. [Online]. Available: https://www.vernier.com/products/software/lp/

[40] Sharp GP1A57HRJ00F datasheet. Sharp. [Online]. Available: https://cdn.sharpsde.com/fileadmin/products/Optoelectronics/Isolation%20Devices/Specs_Photointerrupter/GP1A57HRJ00F_03Oct05_DS_D3-A03901FEN.pdf

[41] NUCLEO F401RE. ST Microelectronics. [Online]. Available: http://www.st.com/en/evaluation-tools/nucleo-f401re.html

[42] N. S. Nise, *Control Systems Engineering*, 4th ed. John Wiley & Sons, 2004.

[43] K. Ogata, *Modern Control Engineering*, 5th ed. Prentice Hall, 2010.

[44] C. L. Phillips and N. H. Troy, *Digital Control System Analysis and Design*, 3rd ed. Prentice Hall, 1995.

[45] G. F. Franklin, J. D. Powell, and M. Workmann, *Digital Control of Dynamic Systems*, 3rd ed. Ellis-Kagle Press, 1998.

[46] D. Zivkovic, "Real World Implementation of Control Algorithms on a Nano Quadcopter," Bachelor's Thesis, ETH Zurich, 2017.

[47] A. Kim and J. Seitz, "Setup of a Distributed Autonomous Flying Test Bed using nano quadcopters," Bachelor's Thesis, ETH Zurich, 2016.

[48] C. Luis and J. L. Ny, "Design of a Trajectory Tracking Controller for a Nanoquadcopter," Mobile Robotics and Autonomous Systems Laboratory, Polytechnique Montreal, Tech. Rep., aug 2016. [Online]. Available: http://arxiv.org/abs/1608.05786

[49] M. F. Everett, "LQR with Integral Feedback on a Parrot Minidrone," Massachusetts Institute of Technology, Tech. Rep., 2015. [Online]. Available: http://mfe.scripts.mit.edu/portfolio/img/portfolio/16.31/16.31longreport.pdf

[50] I. McInerney, X. Ma, and N. Elia, "Cooperative localization from imprecise range only measurements: a distributed non-convex QCQP approach," in *56th IEEE Conference on Decision and Control (CDC)*. Melbourne, Australia: IEEE, 2017, In Press.

[51] K. W. Cheung, W. K. Ma, and H. C. So, "Accurate approximation algorithm for toa-based maximum likelihood mobile location using semidefinite programming," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, May 2004.

[52] P. Biswas and Y. Ye, "Semidefinite Programming for Ad Hoc Wireless Sensor Network Localization," in *Proceedings of the 3rd international symposium on Information processing in sensor networks*. Berkeley, California, USA: ACM, 2004, pp. 46–54.

[53] P. Biswas, T.-C. Liang, T.-C. Wang, and Y. Ye, "Semidefinite Programming Based Algorithms for Sensor Network Localization," *ACM Transactions on Sensor Networks*, vol. 2, no. 2, pp. 188–220, 2006.

[54] A. Beck, P. Stoica, and J. Li, "Exact and Approximate Solutions of Source Localization Problems," *IEEE Transactions on Signal Processing*, vol. 56, no. 5, pp. 1770–1778, 2008.

[55] X. Ma and N. Elia, "Convergence Analysis for the Primal-Dual Gradient Dynamics Associated with Optimal Power Flow Problems," in *2015 European Controls Conference*. Linz, Austria: IEEE, 2015, pp. 1261–1266.

[56] ——, "Seeking Saddle-Points for Non-Convex QCQPs via the Distributed Optimization Dynamics Approach," in *Proceedings of the 22nd International Symposium on Mathematical Theory of Networks and Systems*. Minneapolis, MN, USA: University of Minnesota Digital Conservancy, 2016.

[57] Loco positioning system. Bitcraze. [Online]. Available: https://www.bitcraze.io/loco-pos-system/

[58] N. Jakob, "Optimization-Based Localization in GPS-Denied Environments Using UWB Range Sensors," Semester Project, ETH Zurich, 2016.

[59] M. Figueirôa, A. Moutinho, and J. R. Azinheira, "Attitude Estimation in SO(3): A Comparative Case Study," in *2014 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. Espinho, Portugal: IEEE, 2014.

[60] J. Rolfe and K. J. Staples, *Flight Simulation*, ser. Cambridge Aerospace Series. Cambridge University Press, 1986.