הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Low-Cost Aerial Platform for Guidance, Navigation and Control System Design

Nathaniel Drellich

Low-Cost Aerial Platform for Guidance, Navigation and Control System Design

Comprehensive Engineering Project

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering

Nathaniel Drellich

Submitted to the Senate of the
Technion - Israel Institute of Technology
Nisan 5781 Haifa March 2021

This research was carried out under the supervision of Prof. Daniel Zelazo, in the Faculty of Aerospace Engineering.

The generous financial help of the Technion is gratefully acknowledged.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Abstract

This thesis describes the development and implementation of a low-cost, end-to-end research platform for guidance, navigation, and control systems design of a quadcopter. A combination of hardware and software is developed to provide a suite of options available to researchers to design, simulate, and validate guidance, navigation, and control systems. The primary application of this platform is to enable a "rapid prototyping" environment to allow for drastic reductions in time and cost with the development of these systems. We present a quadrotor aerial vehicle design using off the shelf components combined with 3D-printable parts. A nonlinear and linearized model of the quadcopter dynamics are derived using Newton's and Euler's laws. A modular `Simulink`-based simulator is developed as an easily modified simulation environment which can accommodate various guidance, navigation, and control systems, as well as a variety of simulation parameters. A set of physical testing equipment was also designed and developed for constrained flight testing. An open source, C++ flight control system was developed to be modular and user-friendly in order to run the quadcopter. Multiple libraries were developed to implement different state estimators, controllers, and guidance packages, along with templates for the end-user to use to develop their own. Live logging software was also included to allow for the study of results. The complete platform is used to apply an implementation workflow taking the development of a new system from the theoretical framework through to practical implementation in a short period of time. The methods of validation and verification of the simulator environment are described, and results of the complete implementation workflow are shown and discussed.

List of Symbols and Abbreviations

Abbreviations

ABS	:	acrylonitrile butadiene styrene
BLE	:	bluetooth low-energy
CG	:	center of gravity
ESC	:	electronic speed controller
FSF	:	full-state feedback
GNC	:	guidance, navigation, and control
HPF	:	high-pass filter
IMU	:	inertial measurement unit
LED	:	light emitting diode
LPF	:	low-pass filter
LQR	:	linear quadratic regulator
MIMO	:	multiple-input multiple-output
MEMS	:	microelectromechanical system
MMA	:	motor-mixing-algorithm
PID	:	proportional-integral-derivative
PWM	:	pulse-width modulator
SISO	:	single-input single-output
TWR	:	thrust-to-weight ratio
UAV	:	unamnned aerial vehicle

Chapter 1

Introduction

In recent years, major advancements in electronics have led to a massive leap in microcomputer performance which simultaneously can reduce the size, weight, and cost of each unit. Likewise, advances in sensor miniaturization have created minute sensors with minimal power requirements and marginal cost. One consequence of this has been an explosion of growth in the unmanned aerial vehicle (UAV) industry.

Quadcopters have become the most prevalent UAV in the public sphere due to a variety of factors including, but not limited to, their relatively lower cost when compared with other UAVs, their ability to hover, and their capability to maneuver and navigate in small and constricted spaces. Quadcopters have seen increasing usage in a wide variety of fields ranging from agriculture [4, 5, 6] to military [7, 8] to delivery services [9, 10, 11].

In part due to their rising popularity, and in part due to their inherently unstable and highly nonlinear dynamics, quadcopters have become a popular platform upon which controls research has been focused [12, 13]. Furthermore, the quadcopter has six degrees of freedom which make it a useful platform for the investigation of various guidance laws and implementation of guidance systems. Lastly, the nature of the platform itself requires a relatively high degree of accuracy in state estimation to remain stable. These factors, among others, make the quadcopter an excellent platform for guidance, navigation, and controls (GNC) research. Additionally, the quadcopter serves as an effective educational platform for control systems design. Demonstrative of this fact, in recent years multiple leading technical universities, among them MIT and the Technion, have introduced control labs using quadcopters [14, 15].

Motivation

Predominately, guidance, navigation and controls system design has been done via theoretical work and simulation. This is in a large part due to the high costs of practical implementation and experimentation. However, while powerful tools in understanding and predicting how a system responds to various inputs, simulations are by no means an equivalent replacement for hardware experimentation. No simulation perfectly models

the real world or the various components of a system, and due to inaccuracies of the model, expected performance can differ significantly from the real world results. Additionally, flaws in model or bugs in the code can lead to results which bear no relation to the actual real world system. Such errors can easily become compounded and lead to disastrous results when the simulation provides a given scenario which the researcher expects to find. While these examples are by no means exhaustive, they are illustrative as to the potential downsides to simulation based research and as to obvious advantages of practical implementation.

While the advantages in practical implementation become obvious, one must justify the cost. The often high cost of a platform increases the reluctance to purchase physical hardware. This reluctance is further increased when the knowledge that the new systems being designed may have a relatively high likelihood of damaging, or destroying, the hardware.

While there are a number of inexpensive off-the-shelf quadcopters, they usually consist of proprietary modules even when advertised as open-source. This significantly detracts from their utility as a research platform as it shifts the focus of the user from research to hacking together workaround solutions. Additionally, open-source flight control software is often restricted to specific platforms and hardware, and may be incredibly user-unfriendly to modify beyond very specific presets. As such, this makes working with existing products to implement entirely new systems often an arduous process where the majority of the time is spent getting the vehicle to fly.

By reducing the costs and making a user-friendly flight control package, users of the platform can spend their time implementing their own GNC systems, rather than spending their time attempting to get the quadcopter to fly at all and going through a long and time consuming debugging process with software.

The development of the quadcopter vehicle itself and the flight control package does not however solve all issues. A complete conception-to-simulation-to-implementation process should be developed to simplify the experimental process, while reducing costs. As noted, while imperfect, simulators are still very powerful tools for the design and testing of GNC systems. Consequently, the implementation of a highly modular simulation environment with an easy to use interface can further boost efficiency, while simultaneously reducing costs and safety risks by allowing for new systems to be tested in a perfectly safe manner under a wide variety of conditions before ever being brought to the hardware. Additionally, the development of physical testing equipment can significantly improve the testing process of the hardware in a constrained manner, reducing safety hazards, and increasing testing efficiency at a marginal cost penalty.

With the development of the aforementioned components, a platform can be developed which can enable a “rapid-prototyping” model of system design. Such a model of system design should allow fro the rapid development of a new system by taking it through a simple process. First a new sytem is designed, it is the tested in a simulation environment to detemine whether or not it achieves desired results. Once desired

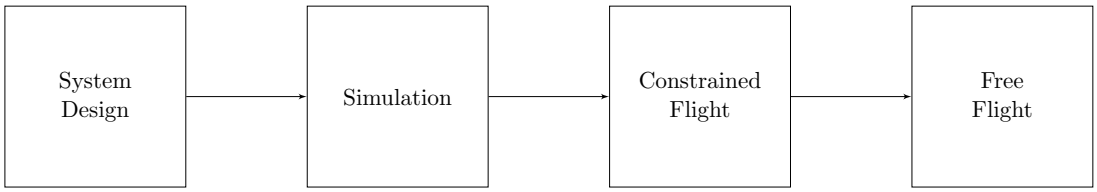


Figure 1.1: A block diagram of the generalized workflow.

results are achieved, the system can then begin hardware testing, first in a constrained environment which limits the degrees-of-freedom to only those necessary for testing, and finally using free flight. The generalized workflow can be found in Figure I.1. Under this model, researchers can have a durable, low-cost platform which can easily have a variety of systems implemented on it for testing in a short window of time which can then be used to verify theoretical results on real hardware.

Thesis Objectives

The goal of this thesis is the development of a low-cost aerial platform for guidance, navigation, and control systems design. To this end, four primary objectives are in play.

i) The design and fabrication of an autonomous quadrotor aerial vehicle.

ii) The design and fabrication of physical testing equipment for the quadcopter.

iii) Development of a modular six degrees of freedom simulation environment.

iv) Development of a modular software package to control the quadcopter hardware.

Thesis Structure

The structure of this thesis is organized such that each section builds towards the development of the complete platform itself. Chapter 2 deals with high level design philosophy driving the thesis. Chapter 3 describes the design process of the platform. This includes the design of the quadcopter itself, along with the physical testing equipment, and software development. Chapter 4 delves into the underlying mathematical model describing the dynamics and kinematics of the quadcopter leading to a nonlinear and linearized model of the quadcopter. Chapter 5 deals with the design and implementation of the estimators and controllers for both the simulation environment and the flight code which runs the hardware. It additionally details the development and implementation of guidance, navigation, and control systems in the simulator environment and flight code. Chapter 6 details the model verification and validation process used to determine that the simulator environment is properly designed to generate valid simulations which properly match real world tests and expectations. Additionally, it discusses various results obtained using the proposed implementation workflow. Lastly, Chapter 7 provides conclusions and some final remarks.

Chapter 2

Design Philosophy

The purpose of this platform is to provide an end-to-end research platform through which a user can implement guidance, navigation, and control systems starting from the theoretical phase, through simulation, and finally finish with implementation of the system on physical hardware. In concept, this platform should enable a "rapid-prototyping" workflow for the development of these systems allowing for a significant reduction in both development time and cost. The general idea is to provide a mechanism in which a streamlined workflow can be developed to allow for rapid iteration and adjustment.

The introduction of 3D printers heralded a revolution in the mechanical design process by allowing for the rapid, on-site prototyping of new parts at low cost. The development of a suitable analog of this process for the development of guidance, navigation, and control systems design can be seen as the beginnings of another such revolution.

To this end, the quadcopter itself and the surrounding systems must be properly understood (Figure 2.1). The quadcopter itself is a dynamic system, which receives input from the control system driving its actuators to determine its dynamics and kinematics. Additionally, environmental affects and real world disturbances will impact the dynamics and kinematics as well, predominately in a negative manner by perturbing the quadcopter from the desired state. The navigation system is negatively impacted by the environment as well as it produces noises which cause inaccurate measurements by the sensors. Lastly, the guidance system is what determines the trajectory that the quadcopter will take. Proper design of the guidance, navigation, and control systems in conjunction with each other allow us to get desired performance from the vehicle.

2.1 Implementation Workflow

As mentioned above, an effective and streamlined workflow is imperative to allow for this "rapid-prototyping" method of systems design. To this end, an implementation workflow was developed. This workflow is a series of specific steps designed to allow for

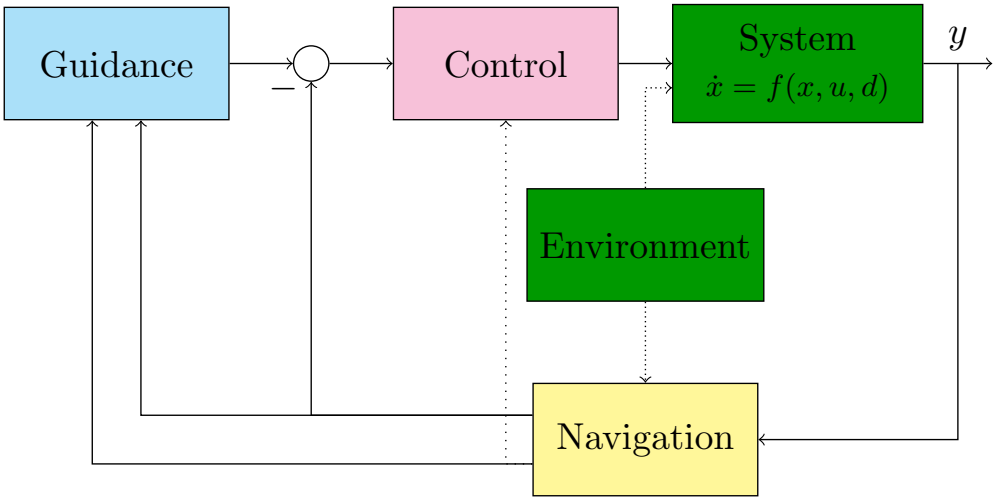


Figure 2.1: A block diagram of the quadcopter system.

rapid prototyping of new systems, while simultaneously keeping both the hardware and the users safe.

The workflow (Figure 2.2) begins with the mathematical model and mission requirements of the system. Once these are defined, GNC systems can be designed. Once designed, the systems should be implemented into a a simulator environment. The simulator is a powerful tool which allows for rapid testing and iteration. When properly designed the simulator also provides a method for iteratively testing the system in a manner of increasing fidelity to the real world (Figure 2.3). Through this process, simulation is done first on the linearized system while assuming access to full state information. This allows us to ensure that our control and guidance systems work correctly in ideal circumstance. After, the feedback path is changed from full state information to the information coming from our navigation system in the absence of any noises. This allows us to ensure that our estimators don't have any blatant errors in them, before enabling dynamic noises to the process and determine whether or not the GNC systems together can compensate for them. Once verified, the system dynamics are changed to the nonlinear system dynamics to produce a higher fidelity simulation of the system. If this test is successful, and the workflow completed, then the user may begin testing on the hardware itself. Once the system or systems have been sufficiently tested in simulation under various conditions, if they meet the desired specifications they can begin being tested on physical hardware. Both for the safety of the user, and for the safety of the quadcopter itself, it is recommended to test the quadcopter while it has physical testing equipment constraining its degrees of freedom. For example, as the attitude control of the quadcopter is the critical aspect for flight stability, testing the flight controller's ability to regulate the attitude in a gimbal works to ensure that the quadcopter will not fly wildly out of control during flight.This is to ensure that

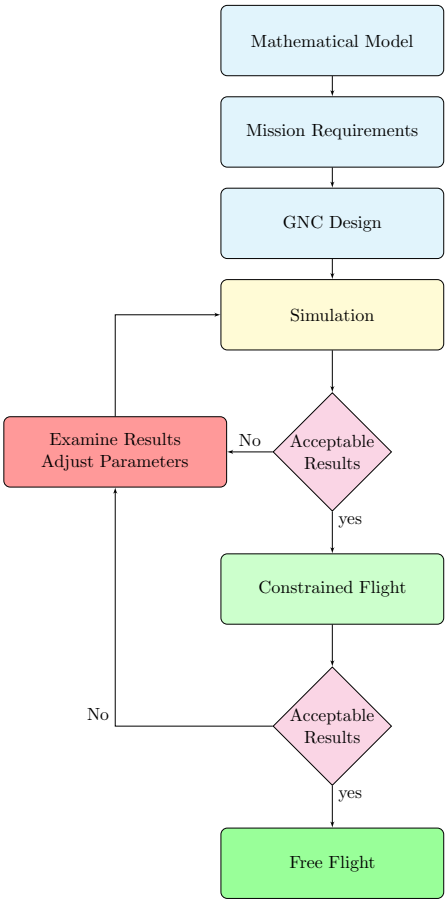


Figure 2.2: The research platform's implementation workflow.

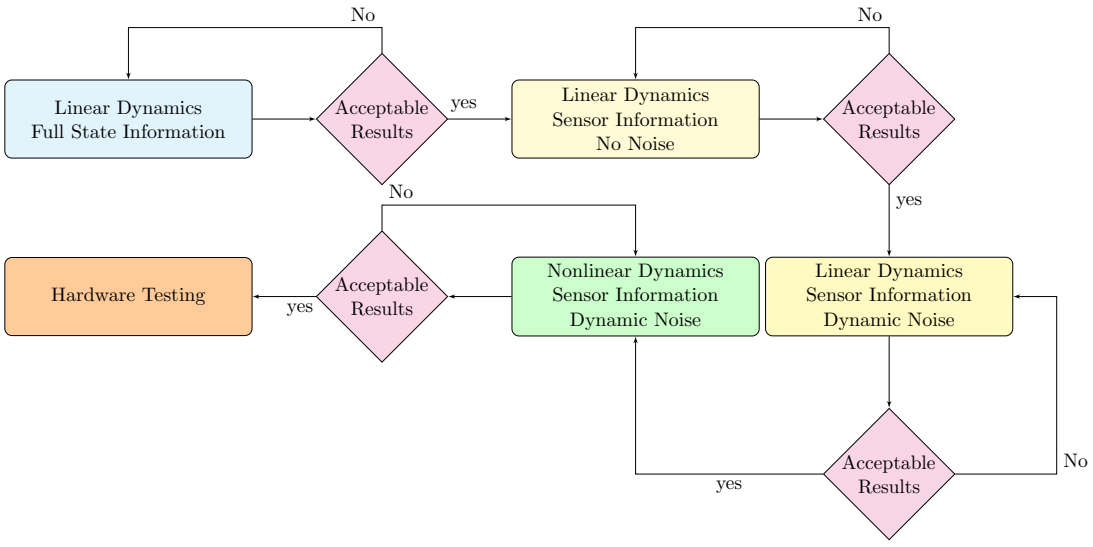


Figure 2.3: The simulation workflow for the testing of new systems.

each aspect of the systems being tested are properly implemented on the hardware and will not cause unexpected, and potentially damaging, behavior in free flight. Once the

quadcopter has been properly tested in constrained flight, it can then begin free flight testing. Assuming successful flight, the system can then be considered validated and new research can begin.

The process from simulation to implementation on the physical hardware has the potential to be exceedingly fast. Furthermore, if all the various components of the platform are properly designed, the time required to pass through the workflow can be further reduced allowing for a significant increase in research efficiency.

Chapter 3

Platform Design

This chapter deals with the design of the complete research platform. This includes the quadcopter, the physical testing equipment, and the software.

3.1 Quadcopter

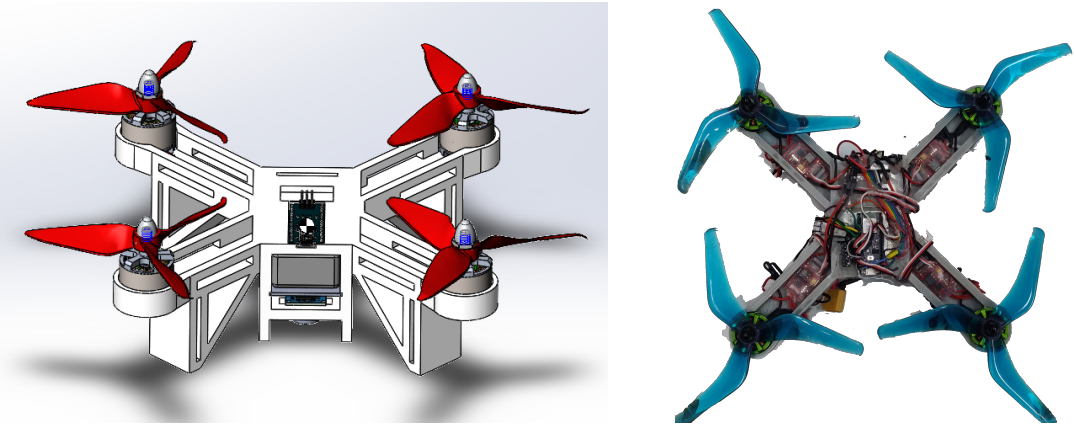
The quadcopter forms the basis of the research platform. The development of the quadcopter involved the mechanical design of the quadcopter frame, selection of electronics and motors, and ensuring compatibility between all parts.

3.1.1 Design Requirements and Constraints

The expected flight environment for the quadcopter is in a flight laboratory. In general, such an area has a limited space, which will constrain the flight envelope of the quadcopter. Additionally, as the quadcopter is a rotorcraft, we require a thrust-to-weight ratio (TWR) that exceeds 1 to achieve lift-off. A TWR range of $1.5 - 4$ is the desired region for the design of the quadcopter. In this range, the quadcopter has sufficient power for flight and maneuvering within the lab environment. Lastly, the expected flight duration is on the order of $0.5 - 5$ minutes. This flight duration provides adequate time to test a large variety trajectories in the flight area.

In order to function properly as a flight platform, the quadcopter must also meet the following criteria:

- The quadcopter must be a low-cost platform to ensure ease of accessibility and reduction in research costs.
- To further reduce cost, and maintain simplicity of the design and fabrication, all purchased parts must be off-the-shelf components.
- Any part that is not purchased off-the-shelf must be 3D printable in house by the researcher.



(a) Solidworks Model of the assembled quadcopter. (b) Fabricated quadcopter vehicle.

Figure 3.1: Assembled Quadcopter.

- The quadcopter weight must be minimized to guarantee a TWR in the desired range.
- To survive inevitable crashes during testing, the quadcopter itself must be highly durable.
- The quadcopter must have sufficient power to guarantee the requirements determined by the flight envelope.

In addition to the constraints placed upon the design by the flight envelope and design requirements, the quadcopter design faces other constraints.

- The quadcopter must minimize the design complexity, both to ensure it can be printed correctly, and so that assembly and construction is simple for the end-user.
- Material selection is limited by the 3D printer. For purposes of this platform, we are limited to a plastic frame.
- The maximum quadcopter size is constrained by the printable volume of the 3D printer. This limits both the size and number of components we may use.

3.1.2 Design

The quadcopter vehicle consists of a frame, four motors, four electronic speed controllers (ESCs), a battery, and various electronic components described in Table 3.1. The design of the quadcopter (Figure 3.1) was a multi-step iterative process involving the selection of the electronic components and design of the frame itself, under the requirements and constraints given in Section 3.1.1. The 3D printers used for this project were Ultimaker S3 Extended and the Ultimaker S5 3D printers. The Ultimaker S3 Extended has a build volume of $230 \times 190 \times 200[mm]$ however, the full print area of the build plate could not be used due to limitations of the software controlling the 3D printer. The frame of

Arduino Nano 33 BLE Sense
The Arduino Nano 33 BLE Sense [16] is the micro-controller chosen to run the flight code of the quadcopter. The BLE Sense is a very small and lightweight micro-controller with numerous sensors built into the chip itself, including but not limited to; a 9-axis IMU, humidity and temperature sensor, and barometric sensor. Additionally it has built in BLE capability. The board runs with a clock speed of 64MHZ, contains 1MB of CPU flash memory, and 64KB of RAM. The micro-controller also has 14 digital input/output pins and 8 analog input pins. Six of the digital input/output pins may be used as pulse-width modulators (PWMs).
Ultrasonic Transducer
The quadcopter uses a downward the HC-SR04 Ultrasonic Distance Sensor for use in height estimation. The HC-SR04 is a high accuracy ultrasonic sensor with a sensing range of $2[cm]$ to $4[m]$. Within these bounds, the HC-SR04 has an accuracy of $\pm 0.1[mm]$.
Optical Flow Sensor
The quadcopter uses the CJMCU-3901 Optical Flow Sensor. The optical flow sensor is designed to operate in ranges greater than $80[mm]$ above a given surface. The CJMCU-3901 is treated as a black-box lateral velocity estimator which runs at $100[Hz]$ and has a high degree of accuracy in its measurements.
Motors
The quadcopter is run using four 2212 920KV Brushless motors. For every volt applied to the motor, the motor will generate 920 RPM under no-load conditions. The motor is rated for a 2-3S LiPo Battery Cell, allowing for a nominal voltage of $11.1[V]$ and a no-load RPM of 10212 RPM.
Electronic Speed Controller
Each motor is powered and controlled by its own $10[A]$ ESC that provides 3-phase alternating current. The ESCs keeps each motor rotating at an rpm determined by the pulse width they receive from the PWM command signal from the Arduino flight controller.
Propellers
Each motor has a 6145 Triple Blade propeller made over polycarbonate. This propeller-motor combination is used to generate $400[grf]$ of thrust per motor, for a total of $1600[grf]$ thrust available to the quadcopter.
Battery
The quadcopter uses a 3S, 1500mAh, 40C LiPo battery. This combination of battery, ESC, and motors allows for a flight time of $5-10$ minutes depending on flight conditions, with an average current draw of approximately $9[A]$.

Table 3.1: Hardware components of the quadcopter.

the quadcopter was designed with a length of $190.25[mm]$ and a width of $196.26[mm]$ due to constraints in 3D printing area, which allowed for less than $1[mm]$ of clearance

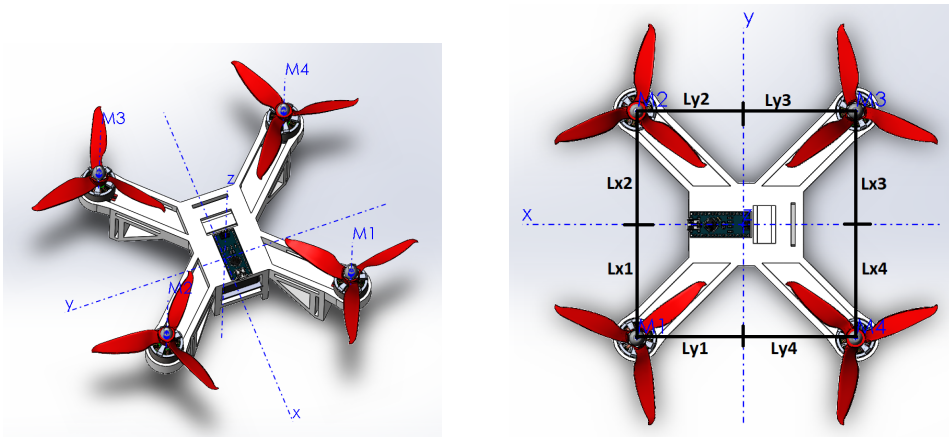
around the edge of the available print area. Later, using the Ultimaker S5 3D printer with its larger $330 \times 240 \times 300[mm]$ build area, the quadcopter frame was not increased in size, rather it was printed with a raft support system to minimize any deformation of the frame during printing. These print area constraints defined the maximize size in which the quadcopter frame could be printed, and thus, the size of the quadcopter itself. Motor mounts were added with a diameter of $34[mm]$ to accommodate standard sized motors, and mounts were designed for electronic components. An engineering drawing of the quadcopter frame can be found in Appendix A and the properties of the fully assembled quadcopter are contained in Table 3.2.

The quadcopter frame was designed in SolidWorks and printed initially on an Ultimaker S3 Extended 3D printer using acrylonitrile butadiene styrene (ABS) plastic. Additional frames were printed using an Ultimaker S5 3D printer. By utilizing SolidWorks for the design of the quadcopter, the final design could be directly be exported as a `.stl` file type for 3D printing. Additionally, Solidworks contains a large library of available material types with all of their mechanical properties stored, using this SolidWorks can also output the mechanical and mass properties of the design. This could be done both for the quadcopter frame alone, and for the quadcopter with its integrated components. When printed, ABS produces strong, stiff parts with a high resistance to physical impacts. These properties combined with the low cost of ABS and its ease of printing makes it an ideal candidate for the frame of the quadcopter.

Total Mass:	$0.588[kg]$
Moment of Inertia:	$\begin{bmatrix} 0.002390945 & 0 & 0 \\ 0 & 0.003543197 & 0 \\ 0 & 0 & 0.002317318 \end{bmatrix} [kg\ m^2]$
Arm Lengths:	$L_{x_1} = 81[mm]$ $L_{y_1} = 76[mm]$ $L_{x_2} = 81[mm]$ $L_{y_2} = 76[mm]$ $L_{x_3} = 81[mm]$ $L_{y_3} = 80[mm]$ $L_{x_4} = 81[mm]$ $L_{y_4} = 80[mm]$
Frame Material:	ABS Plastic

Table 3.2: Physical properties of the quadcopter.

In order to develop a mathematical model of our quadcopter, the geometry of the quadcopter must first be defined. The distance from the motors to the center of gravity (cg) determine the torque generated by the motors. As well, numbering each motor is necessary to define the output of each motor when the equations of motion are defined. We label the front left motor as $M1$ with motors $M2$, $M3$, and $M4$ continuing around



(a) Quadcopter motor locations.

(b) Quadcopter arm lengths.

Figure 3.2: Quadcopter geometry.

the quadcopter in a clockwise fashion (Figure 3.2a). Motors 1 and 3 rotate clockwise, whereas motors 2 and 4 rotate counterclockwise. Once labeled, we can easily define the arm lengths necessary for determining the torques (Figure 3.2b). The reference frames, axes of rotation, and thrust direction of the quadcopter are defined in Figure 3.3. In the aforementioned figures, $L_{x_i}\,\forall\, i \in \{1,2,3,4\}$ refer to the distances of each motor from the x-axis, $L_{y_i}\,\forall\, i \in \{1,2,3,4\}$ refer to the distances of each motor from the y-axis, p, q, and r refer to the body frame angular rotation rates around the x, y, and z-axes respectively, and T represents the total thrust of the quadcopter as determined by the sum of the thrusts $T_i\,\forall\, i \in \{1,2,3,4\}$ generated by each motor respectively.

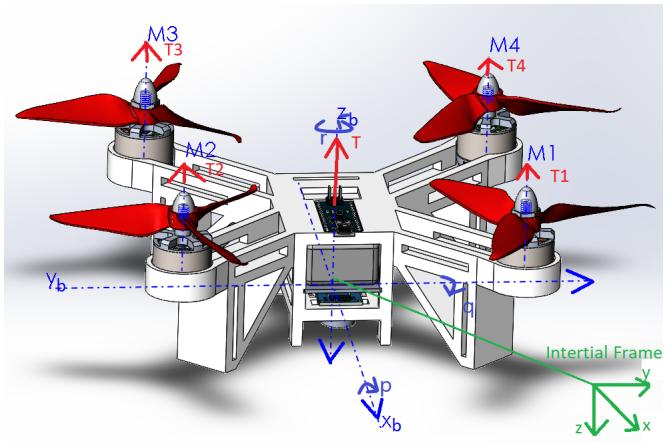


Figure 3.3: Quadcopter frames of reference, forces and rotation axes.

3.1.3 Sensors

As per Table 3.1, the quadcopter was designed using the Arduino Nano 33 BLE Sense as the base platform. The BLE Sense comes equipped with multiple of on-board sensors.

Additional sensors were then added to the quadcopter to provide additional sensor input for a variety of methods of state estimation.

Inertial Measurement Unit

The Inertial Measurement Unit (IMU) comprises of a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer. The IMU used in this quadcopter is the LSM9DS1. The LSM9DS1 [17] is a microelectromechical system (MEMS) developed by STMicroelectronics as a small, low-power, high-resolution motion sensor. Microelectromechical systems combine conventional semiconductor electronics with beams, gears, accelerometers, gyroscopes, switches, sensors and other mechanical structures of microscopic size to create tiny, integrated devices and systems.

The LSM9DS1 is an IMU which performs very well compared to other sensors in the low-cost price range, up to and including sensors more than twice its purchase price [18]. The LSM9DS1 can run at a frequencies of 119$[Hz]$, 238$[Hz]$, 476$[Hz]$, or 952$[Hz]$. Even at the minimum frequency, the LSM9DS1 provides reading suitable for estimation and control. As well, it has been successfully implemented as the IMU for multiple different quadcopters in the past[19, 20], reinforcing validity of its use in this platform. Finally, the LSM9DS1 has multiple pre-built libraries to access the IMU data and it comes pre-installed on the Arduino board, which reduces the work required to implement it into the system. These factors lead to it being the IMU of choice for this platform.

We now briefly describe the components inside the IMU.

- **Accelerometer:** The accelerometer is a sensor which measures the linear acceleration of the quadcopter. An accelerometer functions by using a damped mass on a spring and measuring its position to determine acceleration. When the accelerometer experiences some acceleration, the mass is displaced from its initial position to the point at which the spring can accelerate the mass at the same rate as the casing. By having capacitive "fingers" on the proof mass interspersed between capacitive "fingers" on the stationary casing, this displacement can be measured by the changes in capacitance. Having many such fingers in parallel increases the change in capacitance per unit of distance moved and allows for greater precision. This design can be seen in Figure 3.4.

- **Gyroscope:** The gyroscope is a sensor which measures the angular velocity of the quadcopter. A common form of a MEMS Gyroscope is a vibrating structure gyroscope, which uses a vibrating structure to determine the rate of rotation. This design can be seen in Figure 3.5. The underlying principle takes advantage of the fact that a vibrating object tends to continue vibrating in the same plane, even if its support rotates, and takes advantage of the *Coriolis effect*. The Coriolis effect causes the object to exert a force on its support, and by measuring this force the rate of rotation can be determined.

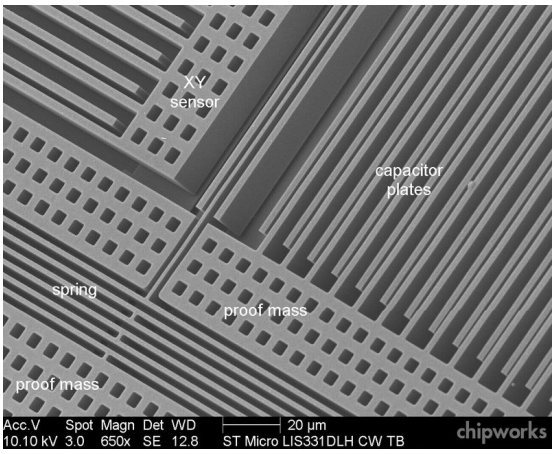


Figure 3.4: A MEMS Accelerometer sensor unit[1].

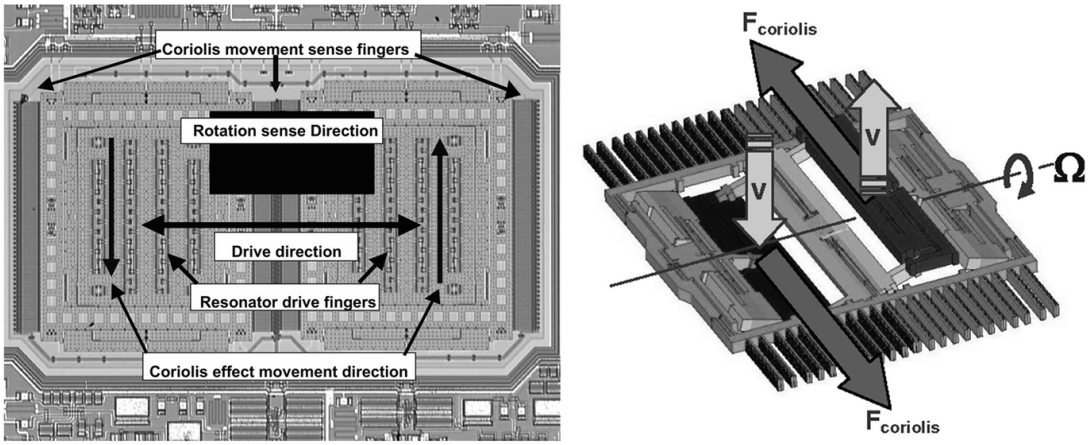


Figure 3.5: A MEMS Gyroscope sensor unit[2].

- **Magnetometer:** The magnetometer is a sensor which measures the earth's magnetic field to determine the heading of the quadcopter. The most common MEMS magnetometer works using the Hall Effect. A current is passed through a conductive plate in the sensor, and as a magnetic field affects the plate, it deflects the electrons causing one side of the plate to be slightly more positive than the other which generates a voltage. Reading the voltage, we can determine the magnetic field strength and direction. The magnetometer functions by suspending a coil of conductive material in the casing by use of torsional beams, as seen in Figure 3.6. A current is passed through the coil, and in the presence of an external magnetic field orthogonal to the current, the generated Lorentz forces cause a rotation of the coil around the torsional beams. By measuring the torsion, the field strength can be determined analytically.

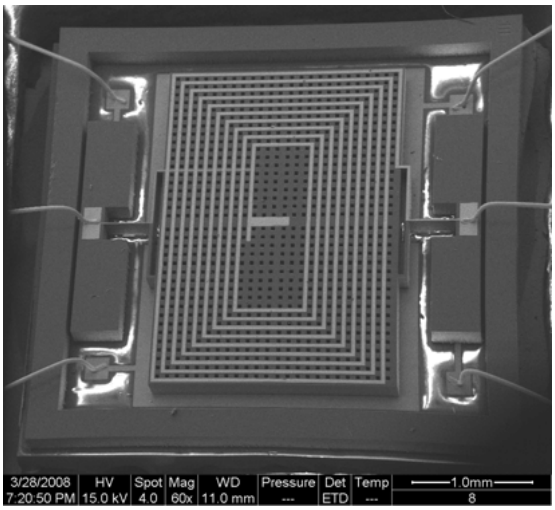


Figure 3.6: A MEMS Magnetometer sensor unit[3].

Barometric Pressure Sensor

The barometric pressure sensor measures the ambient pressure, which varies as a function of altitude. As the altitude increases, the pressure decreases. The change in pressure can be used to determine the absolute altitude. It must be noted however, that the barometric pressure sensor gives exceedingly noisy results. The barometric pressure sensor used in this quadcopter is the LPS22HB MEMS barometric pressure sensor developed by STMicroelectronics [21]. The barometric pressure sensors contains a small, thin, disk-shaped box or capsule with a diaphragm as the side in contact with the atmosphere. The inside of the capsule is usually under a partial vacuum and sealed, with the diaphragm held extended by a spring, and expands or contracts with regard to changes in atmospheric or gas pressure, as seen in Figure 3.7.

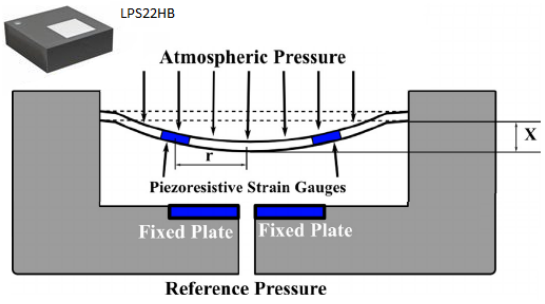


Figure 3.7: Model of a barometric pressure sensor.

While the LPS22HB is a fairly noisy sensor, all MEMS barometric pressure sensors suffer from this issue. Additionally, a properly calibrated low-pass filter can generate usable altitude readings over time using this sensor. As a supplementing sensor for height estimation, which comes pre-installed on the Arduino and has pre-built software

packages to work with, this makes it a useful sensor to introduce to the platform.

Ultrasonic Transducer

The quadcopter was designed to use an HC-SR04 Ultrasonic Distance Sensor. The HCR-SR04 [22] is an ultrasonic transducer, which is a sensor that measures the distance from a surface. The ultrasonic transducer works by emitting a ultrasonic pulse which then reflects off a surface and can be received the by the sensor. The time between the pulse being sent and it returning can be used to accurately ($\pm$0.01cm) determine the range from the surface. As this sensor is highly accurate in the region in which it

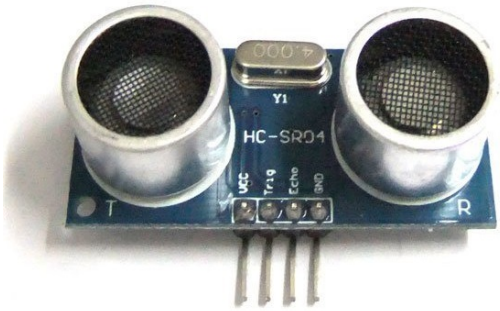


Figure 3.8: HC-SR04 Ultrasonic Distance Sensor.

can accurately receive echos of the pulse output ($2[cm] - 4[m]$), and the quadcopter is designed to fly in a lab environment, flights exceeding $3[m]$ in height are unlikely. The low cost ($1) and high accuracy of this sensor, along with the simplicity of connecting to the Arduino make it an ideal sensor for height estimation. Software written to control the ultrasonic sensor and measure distances is included in Appendix B

Optical Flow Sensor

This quadcopter is designed to use the CJMCU-3901 Optical Flow Sensor. The CJMCU-3901 Optical Flow Sensor is a generic board built around the PMW-3901 Optical Motion Tracking Chip [23]. An optical flow sensor estimates the linear velocity and yaw angle by use of a camera and comparing successive frames [24, 25]. By identifying landmarks in the image, the optical flow algorithm can determine the linear velocity and change in yaw angle based on the translation and rotation of said landmarks. With a known altitude, the velocity can be estimated by determining the size of each pixel, counting the number of pixels the landmark has translated across, and then the velocity may be determined by calculating the distance traveled between the known time between frame. Figure (3.9) provides a simplified visual representation of how the optical flow algorithm works.

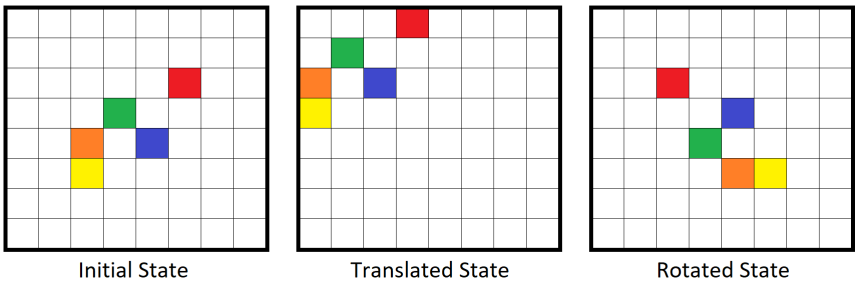


Figure 3.9: A brief explanation of the optical flow algorithm.

3.2 Test-Bench Platform

The test-bench platform was designed for simple testing of the quadcopter in a constrained environment prior to free flight testing. The test-bench platform was designed in SolidWorks and built in part by hand and in part by 3D printing necessary components. Constraining the quadcopter's flight prior to free flight testing is advantageous as it allows for the testing of control loops and estimation in a safe manner to both the researcher and the quadcopter. Additionally, the thrust measurement of the quadcopter is critical for developing the motor output model. As the equations to determine the thrust are highly complex, this makes them prone to give incorrect values due to a wide range of parameters which may not have sufficient precision. Therefore, the ability to generate a thrust model for each motor based on empirical data is incredibly useful.

3.2.1 Variable Degree of Freedom Gimbal

The variable degree of freedom gimbal can be used to test the quadcopter in up to three degrees of freedom (pitch, roll, and yaw). By adjusting the gimbal, various degrees of freedom can be "locked" allowing for the testing along individual rotational axes, or along any combination of rotational axes. The gimbal is composed of two 3D printed

(a) Solidworks Model. (b) Fabricated Unit.

Figure 3.10: A variable degree of freedom gimbal.

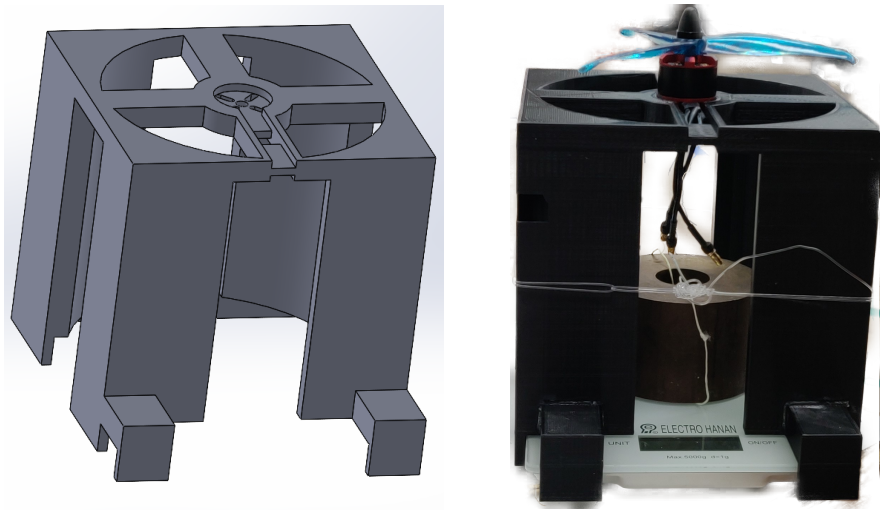
(a) Side view. (b) Top view.

Figure 3.11: Quadcopter mounted in the variable degree of freedom gimbal.

frames joined by 2[mm] metal rods that are passed through bearings with a 6[mm] outer diameter and a 2[mm] inner diameter. The outer frame is set in a mount attached to the base platform which is clamped to the table and has a large bearing to allow for yawing motion (Figure 6.24). A screw is used to prevent any yawing motion when desired. The printed parts of the gimbal were all printed using ABS plastic. Once set up, the quadcopter may be mounted into the gimbal by passing the metal rods into the bearing at the sides of the quadcopter (Figure 3.11).

3.2.2 Thrust Measurement Test Bench

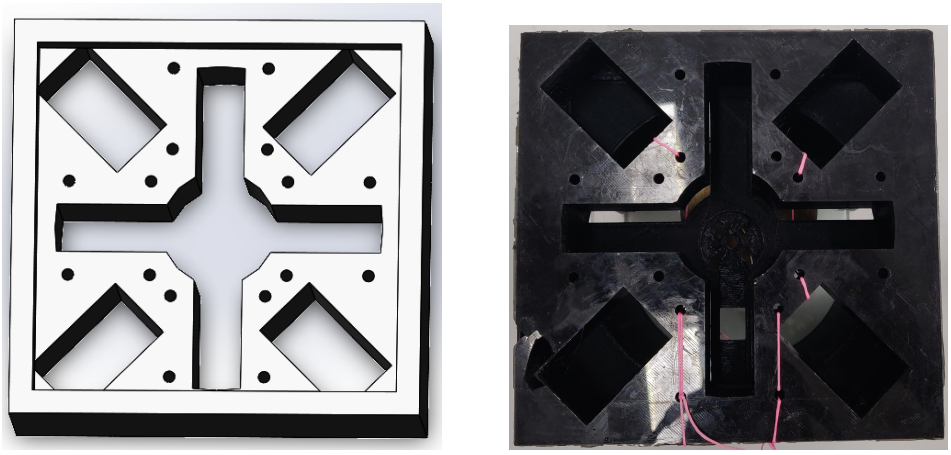
The thrust measuring test bench is two pieces of equipment. The individual motor thrust test bench allows for the measurement of thrust of a single motor at a time. The



(a) Solidworks Model. (b) Fabricated Unit.

Figure 3.12: A motor thrust measurement test bench for a single motor.

second piece of equipment is the quadcopter motor thrust test bench, which slots on top of the individual motor thrust test bench and can test the motor thrust output of the



(a) Solidworks Model. (b) Fabricated Unit.

Figure 3.13: A motor thrust test bench for the assembled quadcopter.

quadcopter itself, either with an individual motor running, all four motors running, or any combination of motors running. This allows for validation of the individual motor thrust calculations when multiple motors are interacting. The thrust measurement test bench was designed in Solidworks and 3D printed with ABS plastic. The test bench was attached to a scale by use of 3 screws running underneath grooves on the underside of the scale, and a $4[kg]$ weight was placed inside to ensure the motors and the quadcopter would not lift off during thrust testing. The procedure for generating thrust profiles using the thrust measurement test bench can be found in Appendix C. This procedure (Figure. 3.14) provides a simple empirical method of determining the thrust generated by each motor-propeller-ESC unit in response to any given command sent to the ESC. Once determined, thrust profiles for each motor-propeller-ESC unit can be integrated into the flight code via the process listed in Appendix D.

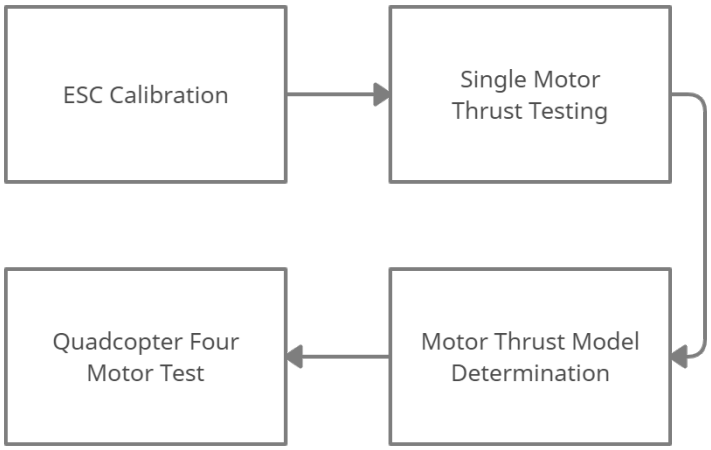


Figure 3.14: A workflow to determine thrust profiles for the quadcopter motors.

3.3 Hardware Fabrication

Fabrication of the platform includes the quadcopter and the physical testing equipment. The majority of all parts were either purchased off the shelf, or printed using an Ultimaker S5 3D printer. The print time for the quadcopter frame was approximately 19 hours, with the print time of each of the thrust test bench components taking on the order of 13 hours. The gimbal parts required approximately 6 hours to print. Once printed, the platform was fabricated was by hand. Sensors, motors and ESCs were mounted in the frame, and the wiring was hand-soldered according to the wiring diagram (Appendix. E).

3.4 Software

The development of the software for the research platform is what allows the platform as a whole to function. Key pieces of software were developed to this end. An Arduino project was written to allow for thrust measurements and direct control of individual and multiple motors simultaneously. A flight simulator was developed using `MATLAB` and `Simulink` for initial testing of the GNC systems. Lastly, a C++ based flight code package was built to run the quadcopter, along with a BLE based logger written for `MATLAB`.

3.4.1 Thrust Measurement

A small piece of software was written as an Arduino project (`.ino` file) for thrust measurement. The software allows for the calibration of the ESCs by sending PWM signal of 2000 μs when the motors start up, and the sending a PWM signal of 1000 μs to set the PWM signal range for the ESCs. An automatic thrust test function slowly increases thrust from 0% to 100% over a set period of time. Additionally, there is a manual thrust test function where the user can send a desired percentage value to ESC of any enabled motor to see what thrust is generated. A fourth function enables and disables the desired motors. The fifth and final function allows for running a manual test on all four motors at once, and the ability to sequentially activate/deactivate motors, or run all 4 simultaneously. The thrust measurement code can be found in Appendix F.

3.4.2 Flight Simulator

A modular flight simulator (Figure 3.15) was written using `MATLAB` and `Simulink`. The flight simulator is a modular piece of software allowing for users to easily design and implement new systems for simulation. The three primary modules are the guidance,

navigation, and control modules. Additionally, there are modules for the plant dynamics (linear and nonlinear dynamics), sensor modules (with or without noises), environmental effects (constant or dynamic), and motor processes (saturation of the motors or not, and process noise or not). Additionally, the simulator allows for easy modification of the quadcopter and sensor properties to allow for rapid adaption to use with other flight platforms. It is important to note that any module may be modified to any extent desired by the end-user, and provided the given inputs and outputs do not change, no impact should be felt by any other module in regards to the functionality of the simulator as a whole. The flight simulator has a file structure that can be seen in

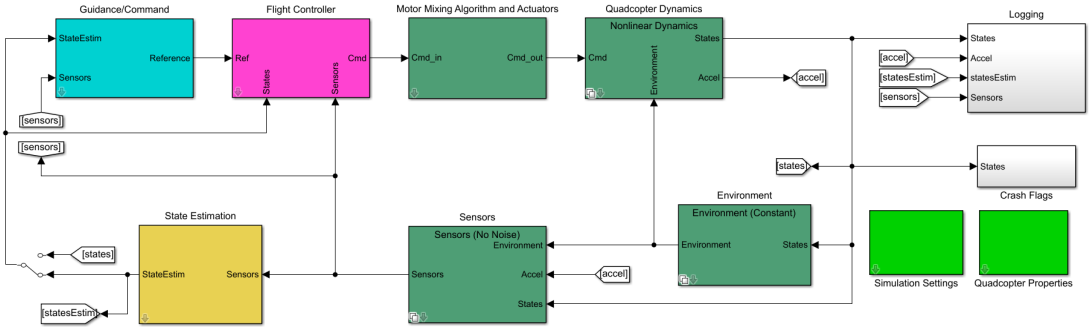


Figure 3.15: `Simulink`-based flight simulator.

Figure 3.16. This directory contains a library folder for each of the GNC systems, as well as a folder with the dynamic models and sensor models used for the simulator. A utilities folder includes various functions required to let the simulator run, along with the `setup.m` and `Visualization.m` files included in the top level folder. Addition of models or systems to the appropriate folders allows for simple addition of new options to the flight simulator software.

Below, we will discuss each block of the flight simulator. For each block being discussed, we will highlight the the block in question by placing a red box around it.

Quadcopter Properties Block

The quadcopter properties block (Figure 3.17) produces a pop-up menu which allows the user to input the quadcopter's physical properties (Figure 3.18). This menu comes pre-loaded with the physical properties of the quadcopter designed with this platform, however, it can easily be adjusted to account for other quadcopters as well.

Simulation Settings Block

The simulation settings block (Figure 3.19) produces a pop-up menu which allows the user to select and modify the simulation's parameters and the simulation's output (Figure 3.20). The user may choose the length of the simulation, as well as whether or not to display plots when the simulation is done. If the user decides to display

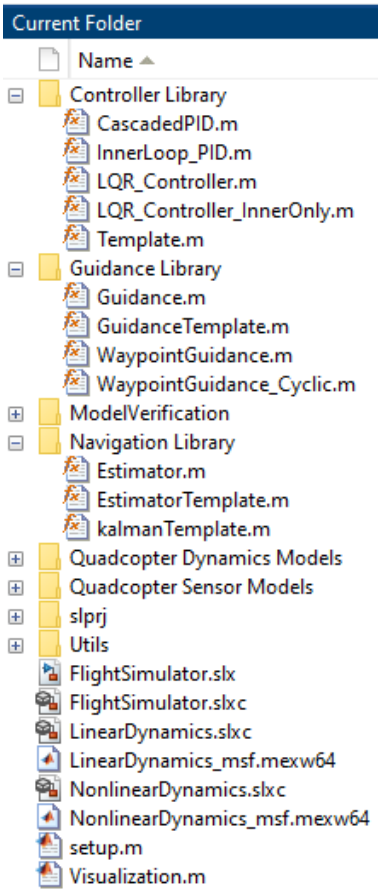


Figure 3.16: File directory of the flight simulator.

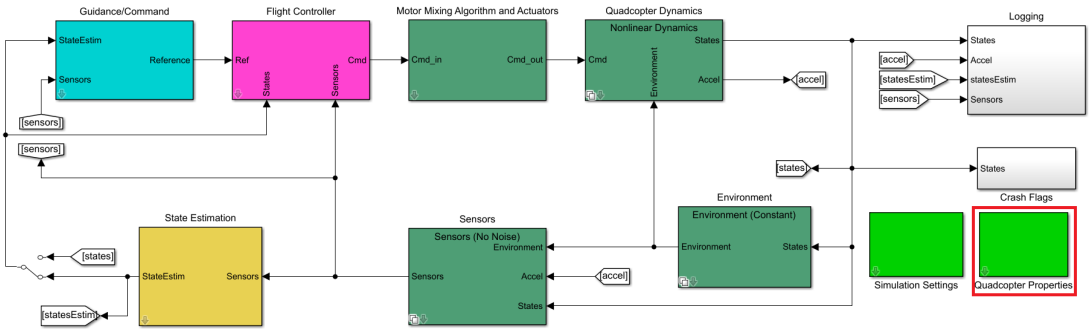


Figure 3.17: Quadcopter Properties Block of the flight simulator.

plots, they may then choose which plots they would like displayed as well. Lastly, the simulation settings block allows the user to set the frequencies at which the sensors and the flight controller run. The frequencies come pre-loaded with the correct values for the given quadcopter but may be modified for use with other quadcopters as well.

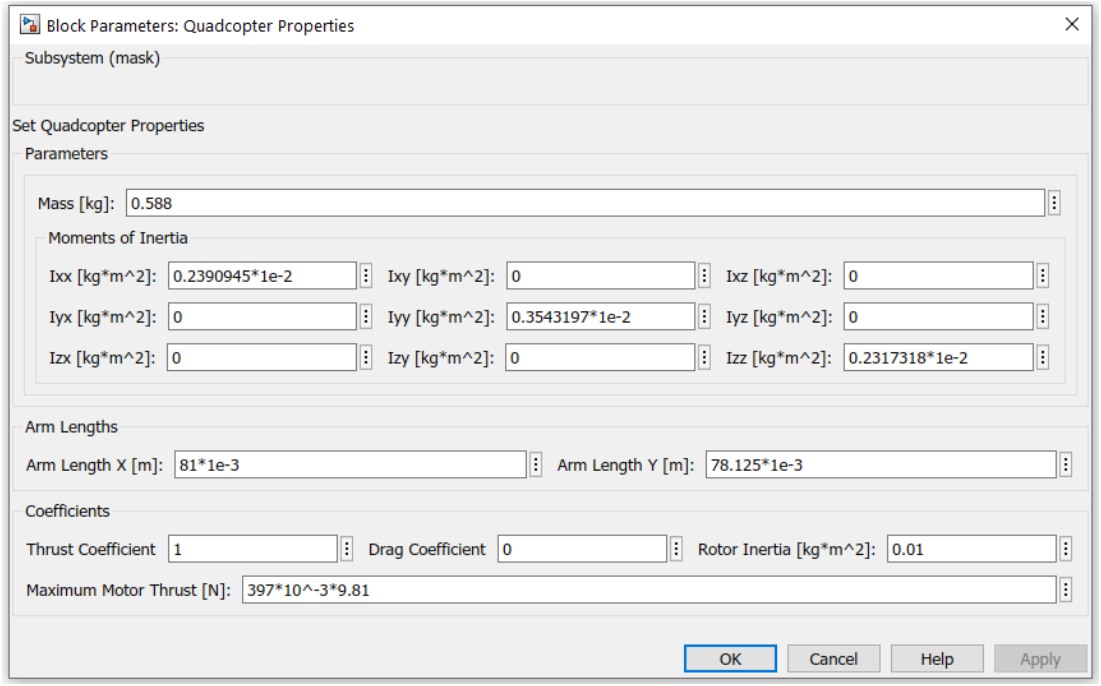


Figure 3.18: Quadcopter Properties Menu for the flight simulator.

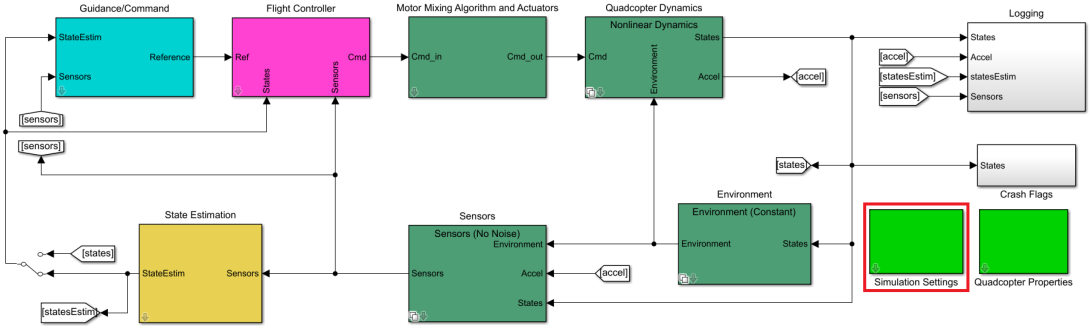


Figure 3.19: Simulation Settings Block of the flight simulator.

State Information Switch

The flight simulator is built to easily allow for changes in various systems. To this end, a feedback switch was added to the simulator. This switch allows the user to change between the guidance and control systems receiving full-state information fed back to them, or only information fed back from the state estimators in the navigation system (Figure 3.21).

Guidance System Block

The guidance subsystem block (Figure 3.22) is a masked subsystem which contains all of the components necessary to choose a guidance system desired by the user. This

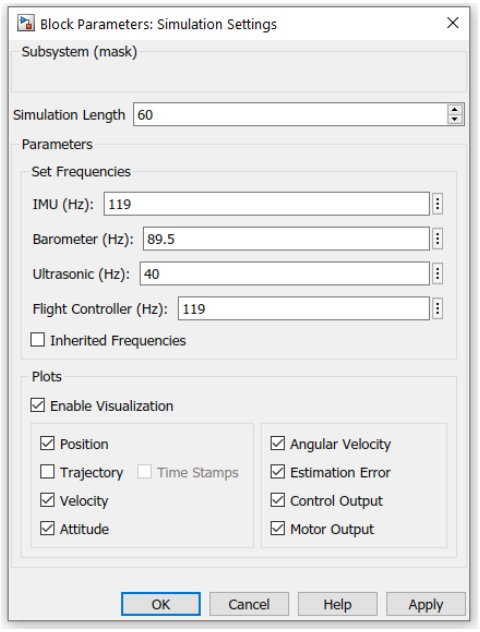


Figure 3.20: Simulation Settings Menu for the flight simulator.

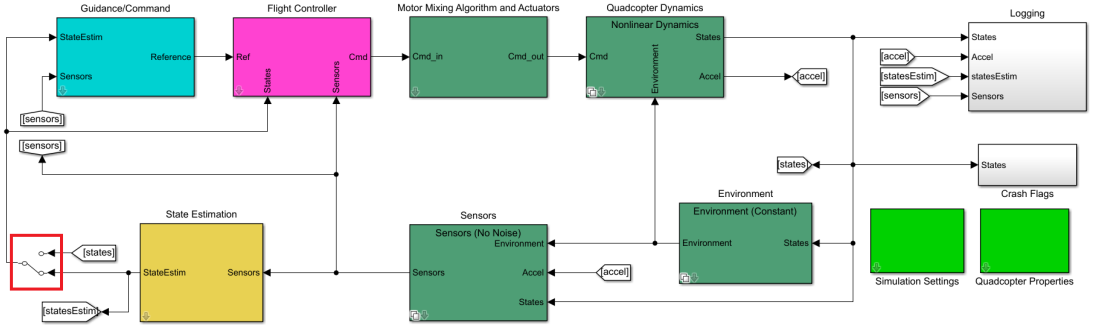


Figure 3.21: State Information Switch for the flight simulator.

subsystem block, when clicked, generates a drop-down menu of guidance systems from all of the guidance system `.m`-files stored in the guidance library folder, with the exception of template files (Figure 3.23).

Motor-Mixing-Algorithm and Actuators Subsystem Block

The MMA and Actuators subsystem block (Figure 3.24) produces a pop-up menu which allows the user to select the various parameters affecting the actuation of the quadcopter (Figure 3.25). Motor saturation may be optionally enabled or disabled to better reflect the physical system. The saturation of the motors is on a per-motor basis, and takes an upper limit of the maximum thrust each motor can generate, and a lower limit set as the idle percentage of the ESCs, below which the motors cease to function. Additionally, process noises may be introduced into the system to better

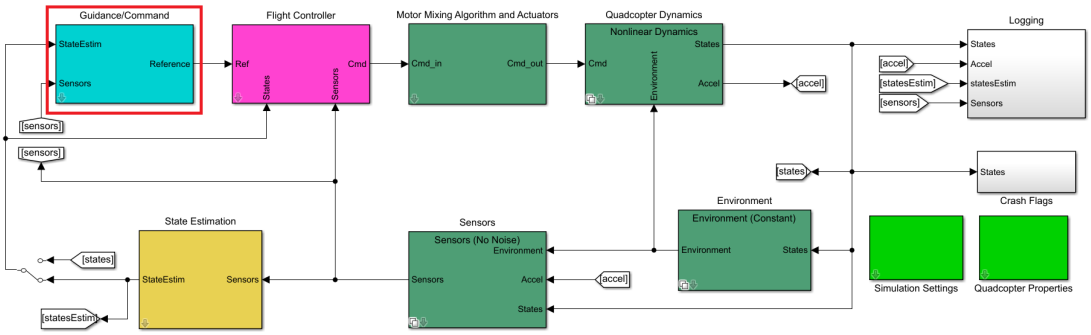


Figure 3.22: Guidance Subsystem Block of the flight simulator.

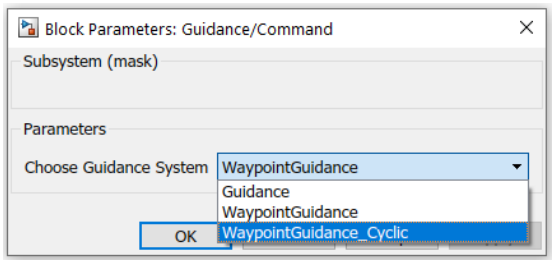


Figure 3.23: Built-in guidance systems for the flight simulator.

simulate inaccuracies in the motor thrust profiles and of inconsistencies in the thrust of the various motor-propeller-ESC units. The process noises when enabled are considered white noises, with each motor getting it's own unique random seed to start the white noise process.

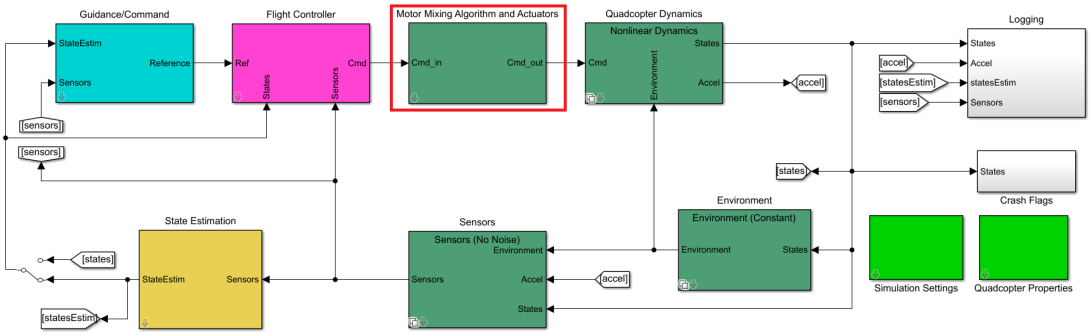


Figure 3.24: Motor-Mixing-Algorithm and actuators subsystem block of the flight simulator.

Dynamics Subsystem Block

The dynamics subsystem block (Figure 3.26) produces a pop-up menu which allows the user to select between various dynamics for the quadcopter (Figure 3.27). Included in the platform is a linear and a nonlinear model.

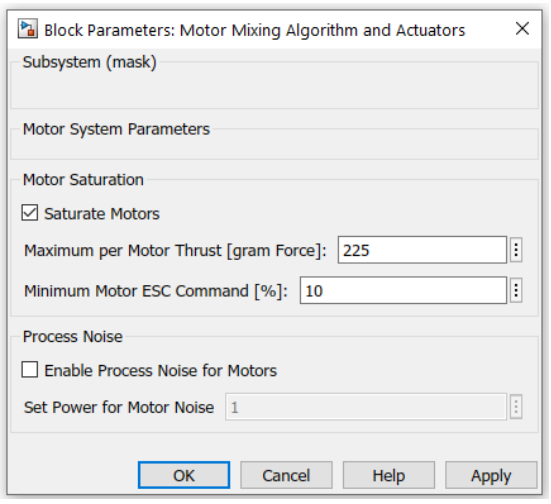


Figure 3.25: Motor-Mixing-Algorithm and actuators subsystem Menu for the flight simulator.

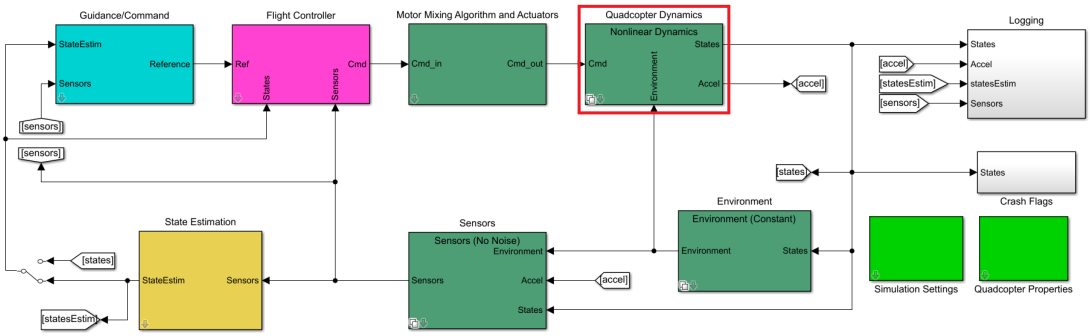


Figure 3.26: Quadcopter Dynamics Subsystem Block of the flight simulator.

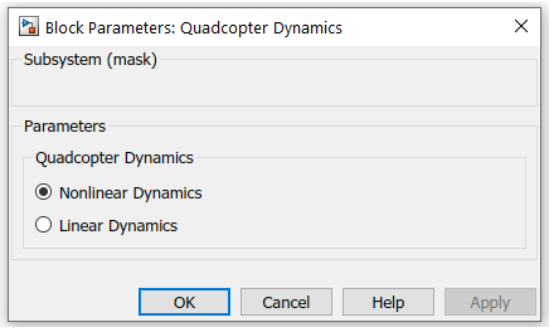


Figure 3.27: Quadcopter Dynamics Subsystem Menu for the flight simulator.

Control System Block

The control subsystem block (Figure 3.28) is a masked subsystem which contains all of the components necessary to choose a control system desired by the user. This subsystem block, when clicked, generates a drop-down menu of control systems from all

of the control system `.m`-files stored in the navigation library folder, with the exception of template files (Figure 3.29).

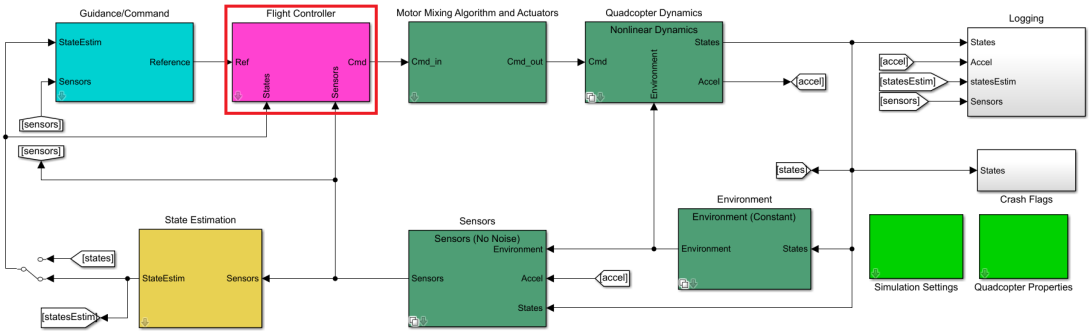


Figure 3.28: Control Subsystem Block of the flight simulator.

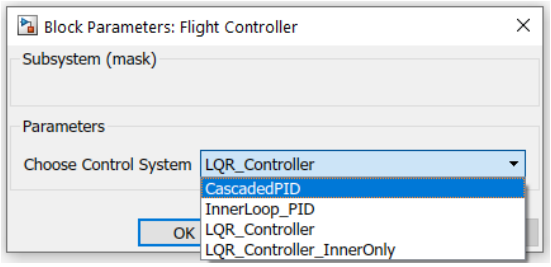


Figure 3.29: Built-in control systems for the flight simulator.

Environment Subsystem Block

The environment subsystem block (Figure 3.30) produces a pop-up menu which allows the user to select the simulations environmental effects, whether or not they are constant or dynamic (Figure 3.31). Constant environment uses static values for various parameters in the simulation (including but limited to, gravity, and air density), whereas the dynamic environment subsystem modifies these values based on the current state and location of the quadcopter. It is recommended in general to use the constant environment due to compilation time required for the dynamic subsystem.

Sensors Subsystem Block

The sensors subsystem block (Figure 3.32) produces a pop-up menu which allows the user to select whether there is sensor noise (dynamics sensors), or whether they function as ideal sensors with no noise (Figure 3.33). Sensor parameters are set to correlate with the sensors used on the physical hardware.

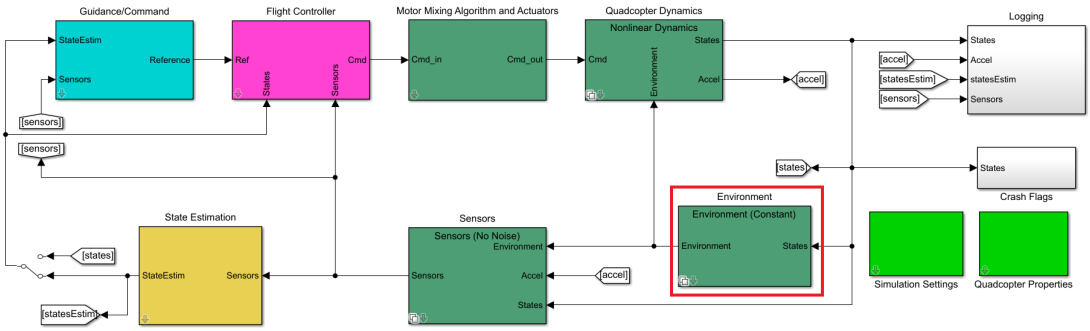


Figure 3.30: Environment Subsystem Block of the flight simulator.

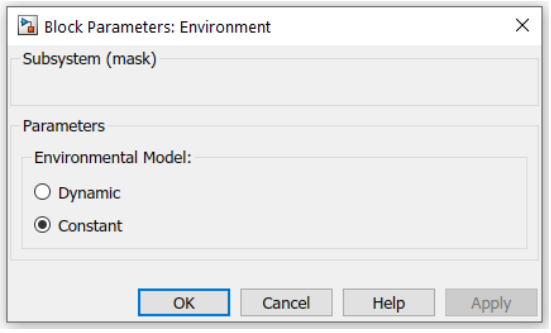


Figure 3.31: Environment Subsystem Menu for the flight simulator.

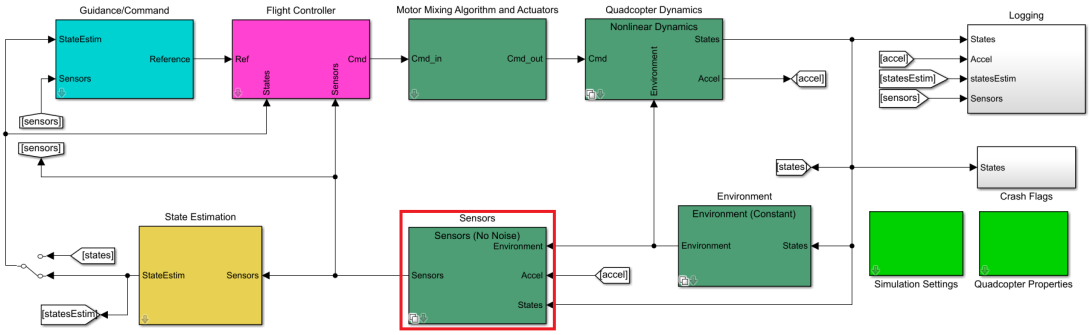


Figure 3.32: Sensors Subsystem Block of the flight simulator.

Navigation System Block

The navigation subsystem block (Figure 3.34) is a masked subsystem which contains all of the components necessary to choose a navigation system desired by the user. This subsystem block, when clicked, generates a drop-down menu of navigation systems from all of the navigation systems `.m`-files stored in the navigation library folder, with the exception of template files (Figure 3.35).

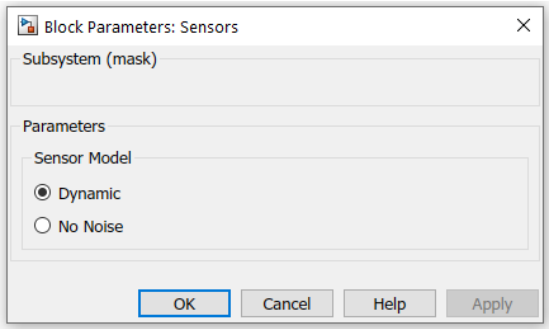


Figure 3.33: Sensors Subsystem Menu for the flight simulator.

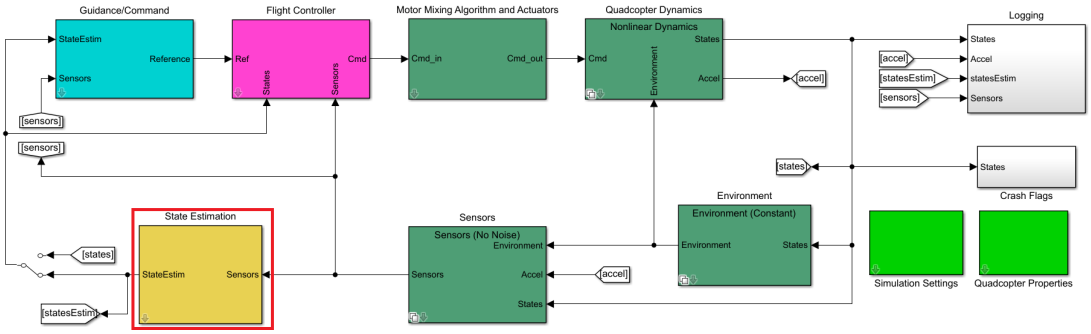


Figure 3.34: Navigation Subsystem Block of the flight simulator.

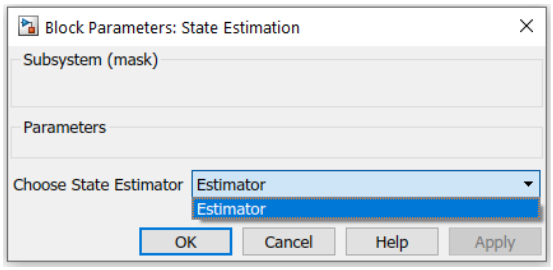


Figure 3.35: Built-in navigation systems for the flight simulator.

3.4.3 Flight Code

As the Arduino can natively compile C++ code, the flight code was written in C++. C++is an object oriented language, which allowed for increased modularity, along with simple to implement reuse of code. The software was designed as a package to allow for autonomous flight of the quadcopter. It was built in a modular fashion in order to allow for ease of use for GNC system design of any particular aspect while allowing the other modules to continue to function. The software was designed to be as user-friendly as possibly and require minimal modification of the base code to run the quadcopter. To this end, a main function was designed which requires editing of only 5 lines of code (Snippet 1) to modify and run the flight code to use the desired packages. The user

```
//-------------------------------------------------------------------------------------------
// USER SETUP IS DONE HERE
//-------------------------------------------------------------------------------------------

// Restrict Quadcopter Dynamics to Fulfill Linearization Assumptions
bool Linear = true;

/* Define Navigation Packages List
Inside the curly brackets, input (in order) the navigation packages desired from the
EstPackages.hpp file. If a package you want to use is not in the EstPackages.hpp file, but
it is an available filter, you can include it by encapsulating the class name in curly
brackets.
*/
std::list<Package> packages = {AttitudeComplementaryPackage, Altitude_Kalman,
Verbatim_pqr, LateralPosition_Kalman};
// Feed the desired navigation packages to the Navigation class for state estimation.
Navigation nav(packages);

// Guidance System goes on this line
queue<Waypoint> waypoints ({
     Waypoint{0, 0, 1}, //
     Waypoint{1, 0, 1}, //
     Waypoint{-1, 0, 1} //
 });
auto guidance = WaypointGuidance(waypoints, false);

/* Choose a Control System
The Control System can be changed by changed the Class of the Control System object csys.
Choose a control system from the library of control system (which is made up by one or
more controllers from the controller library). Leave the name of the object "csys"
unchanged. recovery_csys is the recovery mode control system that enables when the quadcopter
begins to leave the linear range, if the linear restriction is enabled.
*/
CascadedPID csys; //Attitude_LQR csys
InnerLoopPID recovery_csys;

//-------------------------------------------------------------------------------------------
// END USER SETUP HERE. NO *REQUIRED* CHANGES TO CODE BEYOND THIS POINT
//-------------------------------------------------------------------------------------------
```

Snippet 1: User setup of the flight code.

merely needs to choose the following:

- Whether or not the quadcopter is restricted to flight in the region in which the small angles approximation holds true ($\phi, \theta \leq 10°$).
- Which navigation packages to feed to the navigation system.
- Which guidance package to use.
- The desired control system.
- A recovery control system to be used if the quadcopter leaves the linear range (is the linear flag is true.)

Once the user has followed the above steps, the flight code may then be flashed to the Arduino board and flight testing can commence. When the quadcopter begins startup

it begins by initializing all of the data structures and GNC systems required to run. It then begins a five second calibration phase of the sensors. After calibration, the quadcopter waits for a start command signal to be received over BLE. Once the signal has been received, the main function loop begins to poll the sensors for measurement updates and the BLE to determine if a stop signal was received. When received, the measurements are fed into the navigation package, which provides a state estimate to the guidance package. The guidance package then provides the reference signal to the control package. The recovery subsystem determines whether or not the quadcopter needs to enter flight recovery mode and/or shutdown, and if so, it initiates these changes to the control, if not however, the control signal is sent to the motors. All recorded and calculated information is then broadcasted via the logging subsystem, and the loop repeats. A flowchart of this process can be found in Figure 3.36.

The recovery subsystem (Figure 3.37) is used to both ensure safe flight, and allow for both emergency and remote shutdown of the quadcopter. When active, the recovery subsystem ensures that the quadcopter does not leave the region in which the small angle assumption holds valid ($\leq 10°$). If the quadcopter begins to exeed this threshold, then when the attitude reaches $15°$ the flight recovery subsystem overwrite the guidance law to provide a new reference signal of maintaining an altitude of $0°$ and maintaining altitude. Additionally, it uses the recovery controller, which is a preset controller known to regulate the attitude well. In the event the quadcopter continues to exceed the allowed attitude, and reaches an attitude angle of $20°$, the system assumes that the quadcopter has lost control and initiates an emergency shutdown.

The flight code was written in the Visual Studio Code code editor with the PlatformIO plugin to provide Arduino compatibility. Via PlatformIO the code could be flashed over USB directly to the Arduino. While Visual Studio Code and PlatformIO were used to develop the code, any code editor compatible with the Arduino Nano 33 BLE Sense may of course be used to update the code of flash new firmware to the quadcopter.

Flight Code File Directory

A view of the flight code file directory can be found in Figure 3.38. It comprises of numerous files which run the quadcopter. One of the trade-offs required to make the flight code more user friendly to the user, was increasing the complexity of the back-end of the code. Fortunately, the Arduino has the computational resources to handle this with negligible impact. To this end, the majority of the files do not require any user interactions. For general flight using existing GNC system in the flight code, only the `main.cpp` file requires any interaction. In-depth explanations as to the files required for the implementation of new systems can be found in Chapter 5.

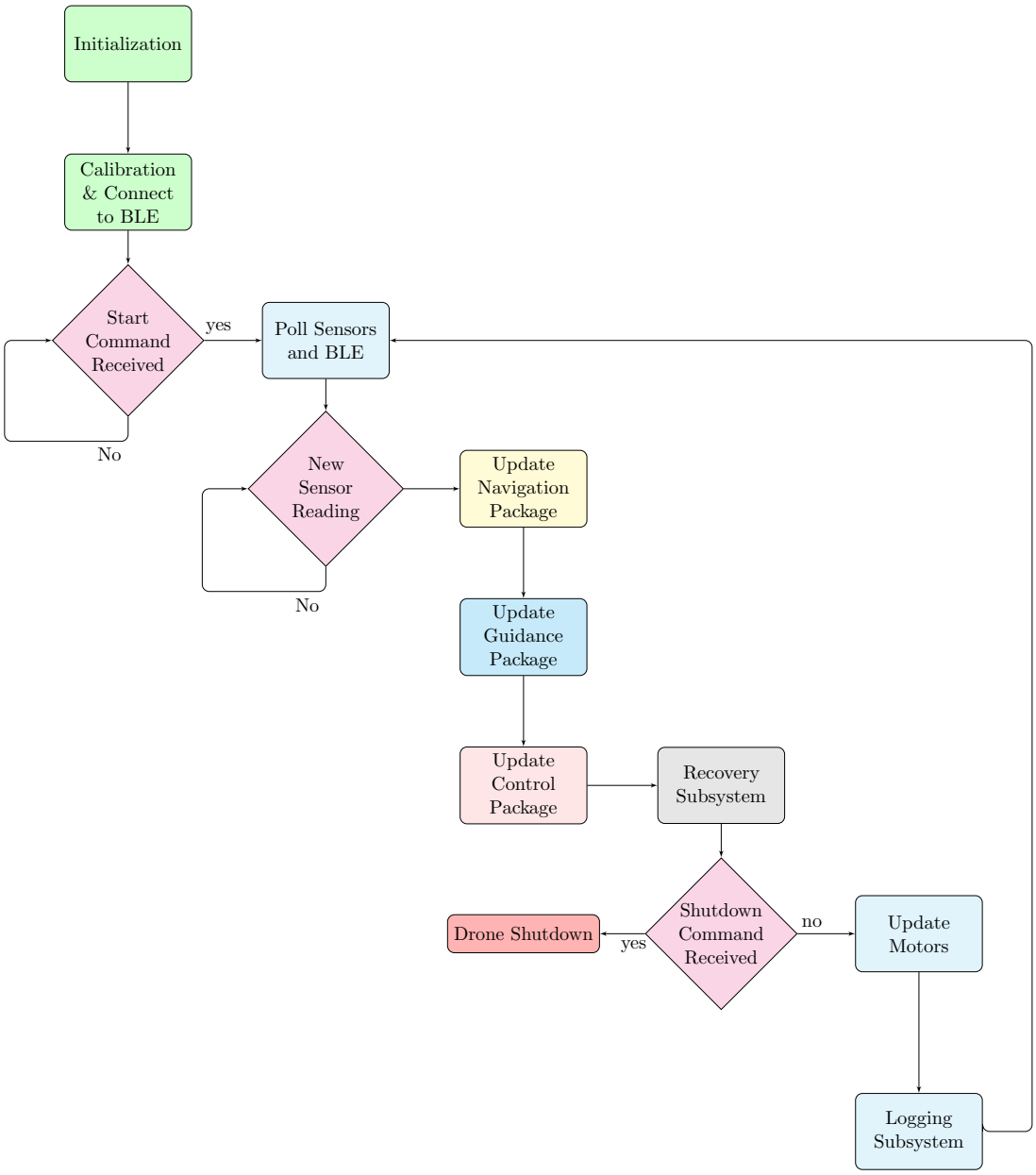


Figure 3.36: Flowchart of the flight code logic.

3.4.4 Bluetooth Live Data Logger

The final piece of software was the data logger. The data logger was written in `MATLAB` and runs in real-time logging the data sent by the quadcopter over BLE. The logger interface allows the user to choose which bluetooth characteristics to read from the quadcopter, and which plots to display during the flight (Figure 3.39. The quadcopter is started via the logger interface, and can have the emergency shutdown signal sent from the logger as well. On completion of a flight the output plots can be manipulated by the user to show only the desired data. It must be noted that in the logger output, the z-axis is inverted with the positive direction showing "up" rather than "down". This is

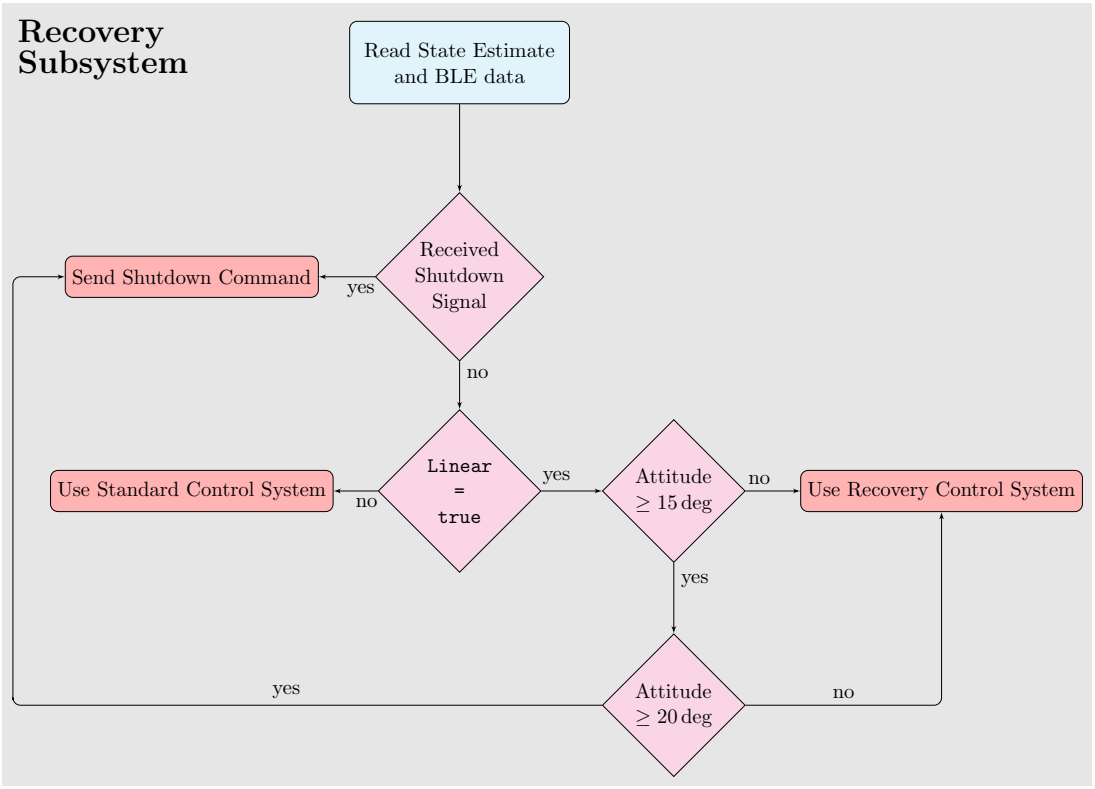


Figure 3.37: Flowchart of the flight recovery subsystem.

for purposes of display only, and to help viewers better intuitively understand what is happening in each plot.

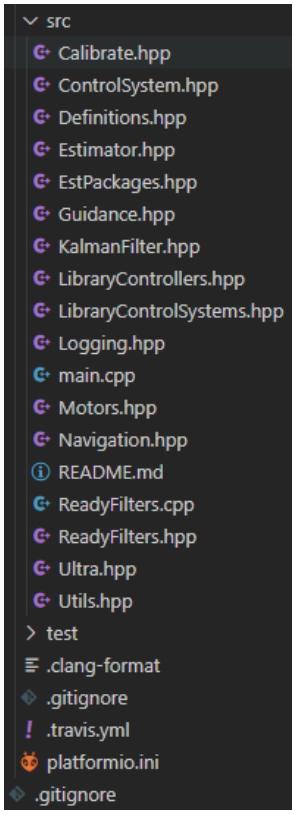


Figure 3.38: File directory of the flight code.

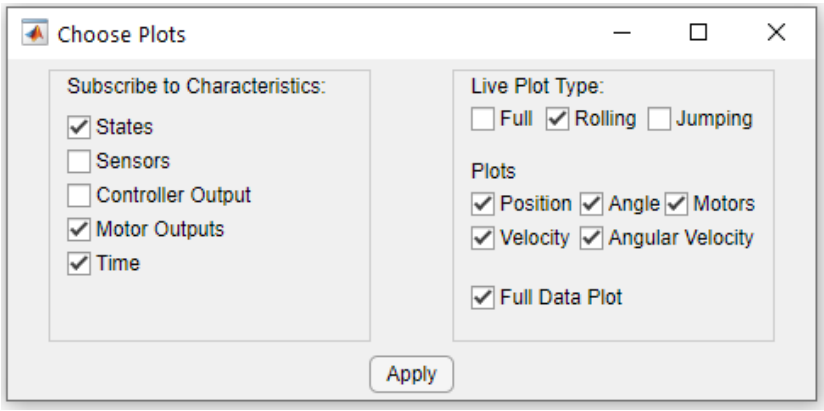


Figure 3.39: Startup Menu for the BLE Live Logging Tool.

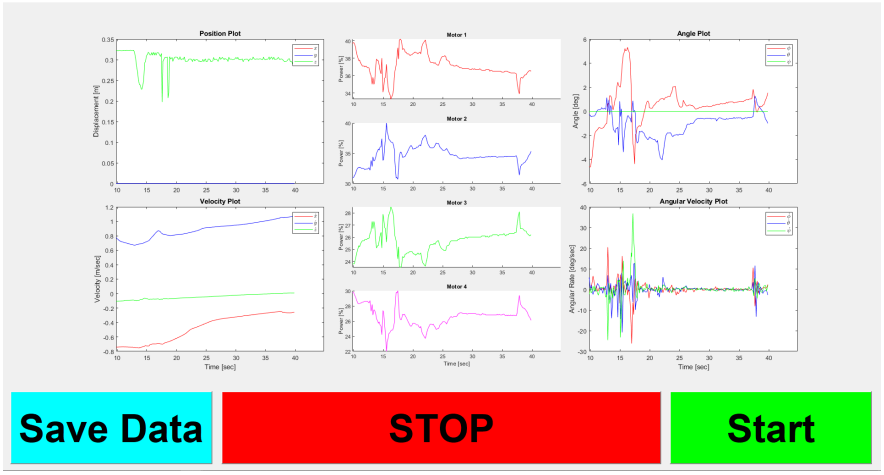
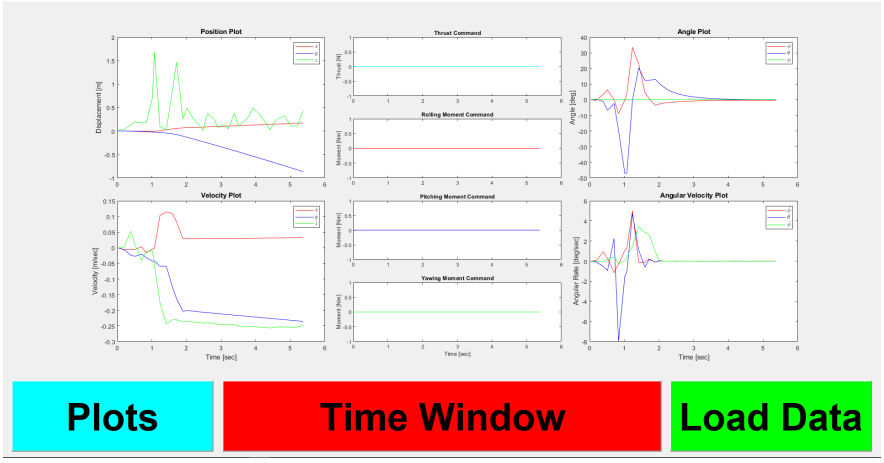
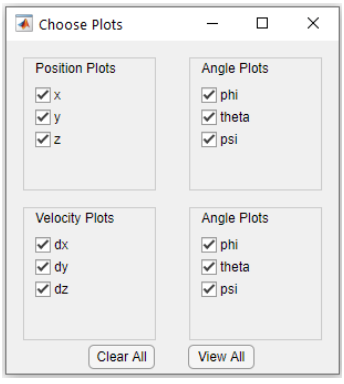


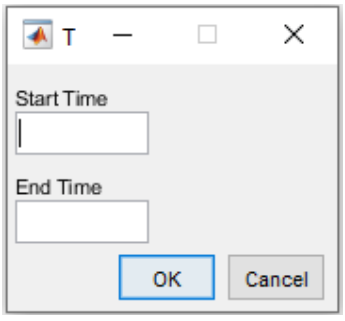
Figure 3.40: GUI for the BLE Live Logging tool.



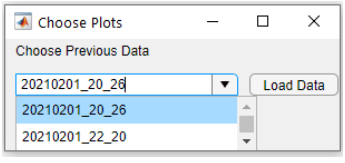
(a) Flight Analysis GUI.



(b) Plots Menu.



(c) Time Window Menu.



(d) Load Data Menu.

Figure 3.41: Flight analysis GUI for the BLE Live Logging Tool.

Chapter 4

Quadcopter Dynamics

The chapter deals with the derivation of the requisite frames of reference and mathematical models necessary to form the underlying basis for the development of the dynamic model of the quadcopter. This is required for the development of the simulator itself, and for the development of controllers and estimators for the quadcopter.

4.1 Frames of Reference

A frame of reference, or reference frame, is the combination of a coordinate system and a set of reference points that can uniquely fix the location and orientation of the coordinate system and standardize measurements within said frame. Reference frames are broadly classified into two main categories: inertial and non-inertial. An inertial reference frame is a non-accelerating reference frame in which Newton's first law of motion holds. Over a small enough geographic region and low speeds, we can approximate an inertial frame of reference using a three-dimensional coordinate system using points on the Earth as our reference points. It should be noted however, that while for the purposes of this thesis and the platform designed therein we can treat this as an inertial reference frame, due to factors such as the rotation of the Earth, it is not in actuality an inertial reference frame. In contrast, a non-inertial reference frame is a frame of reference which is undergoing some acceleration with respect to an inertial reference frame. As such, a body in a non-inertial reference frame with no forces acting upon it may experience an acceleration with respect to the frame of reference due to fictitious forces. A special case of a non-inertial reference frame is a rotating reference frame which rotates relative to an inertial reference frame. A rotating reference frame is characterized by three fictitious forces. These are the centrifugal force, the Coriolis force, and if the rotation rate is not constant, the Euler force; see, for example [26].

4.1.1 Frame Transformations

To derive the mathematical and physical models which govern the quadcopter, we must first understand in which frame of reference we will be working. A quadcopter is a

vehicle with six degrees of freedom, allowing for movement along three perpendicular axes, and rotation around each of those axis. As such, we may define the reference frame as a three-dimensional coordinate system with its origin coinciding with center of mass of the quadcopter. We call this non-inertial reference frame the *body frame*.

While the body frame is a convenient reference frame for describing the motions of the quadcopter and the forces generated by it, it was a relatively inconvenient frame of reference by which to guide the quadcopter or determine its position. A more accessible frame of reference is the inertial reference frame. This frame is also based on a three-dimensional coordinate system, but it uses reference points not related to the quadcopter itself to set an origin. This allows for a simple set of coordinates to be used to describe the position of the quadcopter in its environment irrespective of the orientation of the quadcopter itself.

Both frames of reference (body and inertial) have their own advantages and disadvantages. As such, it is useful to work in each frame of reference where they are best suited and to transition between the two as necessary to benefit from their advantages without the disadvantages. To do so, we use frame transformations. Any point or rigid body in a given reference frame can be described in another reference frame through the use of a translation and rotation of the vector describing the position. As a reference frame is a defined in a three-dimensional coordinate system, the simplest form of this a is a Cartesian coordinate system in which three orthogonal planes intersect to form the coordinate system. A point in this system can be defined as the vector $\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ where x, y, and z represent the distance from the origin of the frame along a given axes defined where the planes intersect. Other coordinate systems exist, such as polar, cylindrical and spherical coordinate systems, however they will not be used in the scope of this thesis. To transition between two different reference frames (of the same type of coordinate system), two different types of motion may be used: rotation and translation. Rotation is the motion of the space around at least one fixed point while preserving the distances between the points in the space. A translation on the other hand, moves every point in a given space a fixed distance in a given direction. Taking these definitions, we can then define the transformation between reference frames as

$$\mathbf{p}^{\mathcal{B}} = R^{\mathcal{B}}_{\mathcal{A}}\mathbf{p}^{\mathcal{A}} + \mathbf{t}^{\mathcal{A}}, \tag{4.1}$$

where $\mathbf{p}^{\mathcal{B}}$ denotes a position vector expressed in frame $\mathcal{B}$, $R^{\mathcal{B}}_{\mathcal{A}}$ is a rotation matrix from frame $\mathcal{A}$ to frame $\mathcal{B}$, and $\mathbf{t}^{\mathcal{A}}$ is a vector expressed in frame $\mathcal{A}$ translating the origin of frame $\mathcal{A}$ to $\mathcal{B}$. A rotation matrix is an orthonormal matrix, such that $R^{-1} = R^T$, which leads to

$$\mathbf{p}^{\mathcal{B}} = R^{\mathcal{B}}_{\mathcal{A}}\mathbf{p}^{\mathcal{A}} \iff \mathbf{p}^{\mathcal{A}} = (R^{\mathcal{B}}_{\mathcal{A}})^T\mathbf{p}^{\mathcal{B}} = R^{\mathcal{A}}_{\mathcal{B}}\mathbf{p}^{\mathcal{B}}, \tag{4.2}$$

For purposes of convention, we use the superscript $(\ )^{\mathcal{B}}$ to represent the body frame

and the superscript $(\ )^{\mathcal{I}}$ to represent the inertial frame. The translation is rather simple to calculate as it is just the displacement of the body from the origin of the inertial frame.To simplify notation, for the remainder of this thesis, we define $R \triangleq R^{\mathcal{I}}_{\mathcal{B}}$ and $R^T \triangleq R^{\mathcal{B}}_{\mathcal{I}}$.

In this thesis, we will define the body frame as centered on the quadcopter center of mass, with the x-axis ($x^{\mathcal{B}}$) pointing the forward direction, the y-axis ($y^{\mathcal{B}}$) pointing to the right, and the z-axis ($z^{\mathcal{B}}$) pointing down (Figure 3.3). We will additionally define the inertial frame ($x^{\mathcal{I}}$, $y^{\mathcal{I}}$, $z^{\mathcal{I}}$) to be somewhere on the surface of the earth. For simplicity, we assume that it is aligned with the quadcopter initial (grounded) position.

4.1.2 Euler Angles and Rotation Matrices

Any two independent orthonormal coordinate frames, in three-dimensional space, can be related by a sequence of three or fewer rotations about coordinate axes, where no two successive rotations may be about the same axis. Euler angles represent these rotations with ϕ representing the rotation around the *x-axis* (roll), θ representing the rotation around the *y-axis* (pitch), and ψ representing the rotation around the *z-axis* (yaw) (Figure 4.1).

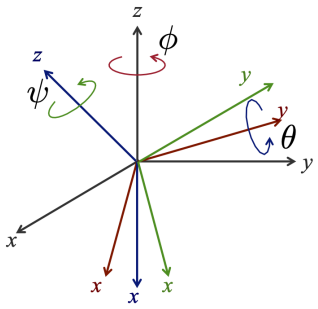


Figure 4.1: Euler angles.

Each rotation is around its own intermediate reference frame. Assuming a three-dimensional coordinate system, we can then define the rotations around ϕ (4.3), θ (4.4), and ψ (4.5) respectively [27],

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, \tag{4.3}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{4.4}$$

$$R_z(\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.5}$$

Changing the order of rotation changes the solution, and as such, the order must be consistent. A common rotation order in the aerospace industry used to rotate from the body frame to the inertial frame is the yaw-pitch-roll (ypr) order of rotation. This order of rotation produces a rotation matrix (4.6). For sake of convention, we define $c_\alpha = \cos(\alpha)$ and $s_\alpha = \sin(\alpha)$. We can transform the inertial frame to the body frame as follows

$$R = R_x(\phi)R_y(\theta)R_z(\psi) = \begin{bmatrix} c_\psi\, c_\theta & c_\theta\, s_\psi & -s_\theta \\ c_\psi\, s_\phi\, s_\theta - c_\phi\, s_\psi & c_\phi\, c_\psi + s_\phi\, s_\psi\, s_\theta & c_\theta\, s_\phi \\ s_\phi\, s_\psi + c_\phi\, c_\psi\, s_\theta & c_\phi\, s_\psi\, s_\theta - c_\psi\, s_\phi & c_\phi\, c_\theta \end{bmatrix}, \tag{4.6}$$

and we can transform the body frame to the inertial frame using the transpose

$$R^T = R_x(\phi)^T R_y(\theta)^T R_z(\psi)^T = \begin{bmatrix} c_\theta\, c_\psi & s_\phi\, s_\theta\, c_\psi - c_\phi\, s_\psi & c_\phi\, s_\theta\, c_\psi + s_\phi\, s_\psi \\ c_\theta\, s_\psi & s_\phi\, s_\theta\, s_\psi + c_\phi\, c_\psi & c_\phi\, s_\theta\, s_\psi - s_\phi\, c_\psi \\ -s_\theta & s_\phi\, c_\theta & c_\phi\, c_\theta \end{bmatrix}. \tag{4.7}$$

If our velocity in the body frame is defined as $\mathbf{v}^{\mathcal{B}} = \begin{bmatrix} u & v & w \end{bmatrix}^T$, and our velocity in the inertial frame is defined as $\mathbf{v}^{\mathcal{I}} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T$, we can transform our body-frame velocity to our inertial frame velocity via $\mathbf{v}^{\mathcal{I}} = R\mathbf{v}^{\mathcal{B}}$, which yields,

$$\begin{cases} \dot{x} &= w\left(s_\phi s_\psi + c_\phi c_\psi s_\theta\right) - v\left(c_\phi s_\psi - c_\psi s_\phi s_\theta\right) + u(c_\psi c_\theta) \\ \dot{y} &= v\left(c_\phi c_\psi + s_\phi s_\psi s_\theta\right) - w\left(c_\psi s_\phi - c_\phi s_\psi s_\theta\right) + u\left(c_\theta s_\psi\right). \\ \dot{z} &= w\left(c_\phi c_\theta\right) - u\left(s_\theta\right) + v\left(c_\theta s_\phi\right) \end{cases} \tag{4.8}$$

While taking the time derivative of the Euler angles will generate corresponding angular rates, the angular velocity vector is defined as the vector pointing along the axis of rotation of the rotating quadcopter, where p, q, and r represent the rotations around the body-frame x, y, and z axes respectively. Therefore, time derivatives of the Euler angles must then be rotated in sequence to find the angular velocity,

$$\begin{aligned} \begin{bmatrix} p \\ q \\ r \end{bmatrix} &= R(\phi)R(\theta) \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + R(\phi) \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}. \end{aligned} \tag{4.9}$$

This can be reformulated to give the time derivative of the Euler angles in terms of the body frame angular velocities,

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & s_\phi\, t_\theta & c_\phi\, t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & \frac{s_\phi}{c_\theta} & \frac{c_\phi}{c_\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \tag{4.10}$$

where we define $t_\alpha = \tan(\alpha)$ [28].

4.2 Quadcopter Dynamics and Kinematics

To begin preparing a controller for our quadcopter, we must first derive the mathematical models which govern the dynamics of the quadcopter.

4.2.1 Assumptions

A quadcopter is a highly nonlinear system and subject to a multitude of forces during flight. However, as the quadcopter will be of relatively small size, slow moving, and relatively rigid, we may make the following assumptions:

1. The quadcopter is a rigid body.
2. The Center of Gravity and the Center of Mass coincide with the geometric center of the quadcopter.
3. The Moment of Inertia of the propellers are negligible.
4. At low airspeed, the aerodynamic drag is negligible.
5. The freestream velocity is negligible.
6. There is no blade-flap.

Under the above assumptions, we can examine the forces, torques, and moments of inertia to develop the equations of motion of the system.

4.2.2 Forces

The quadcopter generates forces using four propellers attached to motors. These forces determine the quadcopter's dynamics.

Thrust and Drag

Due to the motors and propellers of the quadcopter being fixed along the same plane in the body-frame (Figure 3.3), the quadcopter can generate thrust only along the *z-axis* of the body frame. However, as the relevant frame of reference for guidance, navigation and control of the quadcopter is the inertial reference frame, we derive the total forces

on the quadcopter in the inertial reference frame. Additionally, note that the thrust along the *z-axis* is negative due to the *z-axis* pointing downward and the thrust pointing up.

$$\sum F = \mathrm{m}\ddot{\mathbf{p}} = \begin{bmatrix} 0 \\ 0 \\ \mathrm{m}g \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ -T \end{bmatrix} \tag{4.11}$$

Here, $\sum F$ is the total force acting on the quadcopter, m is the mass of the quadcopter, g is acceleration due to gravity, and T is the total thrust generated by the propellers of the quadcopter. To calculate the thrust generated by the propellers we employ the *blade element theory* (BET)[29].BET is a mathematical process used to determine the behavior of propellers. By sectioning the blade into multiple small parts and evaluating the forces on each section, then integrating the forces along the blade and over one revolution the forces and moments can be calculated. The thrust generated by the propeller of the quadcopter is equal to the lift generated by the propeller, and can be calculated using

$$L = \frac{1}{2}\rho S V^2 C_L, \tag{4.12}$$

where ρ is the atmospheric density, S is the surface area of blade segment, C_L is the lift coefficient (determined by the shape of the blade and angle of attack), and V is the relative velocity between the blade and the air. Under the assumption of a negligible freestream velocity, the air is standing still, and therefore the velocity between the blade and the air can be defined at any point along the blade as

$$V = \omega_{m_i} r_b \tag{4.13}$$

where V is the velocity between the blade and the air, ω_{m_i} is the rotational velocity of the given blade, and r_b is the distance of the point in question from the axis of rotation. (4.12) can now be rewritten, substituting L for T_i, defining the radius of the propeller as R_{prop}, and simplified by consolidating the terms into

$$T_i = \frac{1}{2}\rho S R_{prop}^2 \omega_{m_i}^2 C_T = k_m \omega_{m_i}^2. \tag{4.14}$$

Summing the thrust generated by each propeller determines the total thrust of the quadcopter,

$$T = \sum_{i=1}^{4} T_i = \sum_{i=1}^{4} k_m \omega_{m_i}^2. \tag{4.15}$$

In the same manner in which lift is generated by the propellers, drag is generated as well. The lift is perpendicular to the plane of motion, drag is generated in the plane of motion in the opposite direction of the motion. The drag equation can be calculated as

$$D = \frac{1}{2}\rho S V^2 C_D. \tag{4.16}$$

As with the lift equation, we can rewrite (4.16) by substituting in (4.13) and consolidating the terms to yield

$$D = k_d \omega_{m_i}^2. \tag{4.17}$$

4.2.3 Torques

From classical mechanics and the geometry of the quadcopter (Chapter 3.1.2), we can calculate torques around the *x-axis* and *y-axis* in the body frame simply as the sum of the cross-products of the thrust generated by each motor and the distance of the motors from the center of mass. The torque around $x^{\mathcal{B}}$ is defined as τ_ϕ, around $y^{\mathcal{B}}$ is defined as τ_θ, and around $z^{\mathcal{B}}$ is defined as τ_ψ. Accounting for the quadcopter's geometry (Figure 3.3) we can then determine the torques generated from each motor, and the total torque acting upon the quadcopter. This geometry leads to a positive or negative torque being generated based on the direction along the x and y-axes that the motor is located from relative to the center of gravity. Motors "forward" of the center of gravity will generate a positive torque around the x-axis, while motors "behind" the center of gravity will generate negative torques around the x-axis. Likewise, motors to the "right" and "left" of the center of gravity will generate positive and negative torques respectively around the y-axis yieding,

$$\tau_\phi = \mathrm{L_{x_1}} T_1 - \mathrm{L_{x_2}} T_2 - \mathrm{L_{x_3}} T_3 + \mathrm{L_{x_4}} T_4 \tag{4.18}$$

$$\tau_\theta = \mathrm{L_{y_1}} T_1 + \mathrm{L_{y_2}} T_2 - \mathrm{L_{y_3}} T_3 - \mathrm{L_{y_4}} T_4. \tag{4.19}$$

From (4.16) we can define the torque that each propeller generates around the *z-axis* in the body frame by multiplying by the radius of the propeller R_{prop}, yielding

$$\tau_{D_i} = \frac{1}{2}\rho \mathrm{R}_{prop} S V^2 C_D = \frac{1}{2}\rho \mathrm{R}_{prop} S (\omega_{m_i} \mathrm{R}_{prop})^2 C_D = b\omega_{m_i}^2, \tag{4.20}$$

where $b = \frac{1}{2}\rho \mathrm{R}_{prop}^3 S C_D$. We can see that τ_D (4.20) resembles T (4.14) in form. If we define a scaling factor $c = \frac{b}{k_m}$ then we can find the given torque for each motor,

$$\tau_{D_i} = cT_i = ck_m\omega_{m_i}^2. \tag{4.21}$$

If we sum the torques generated around the body frame *z-axis* by each of the motors, we can calculate the total torque around that axis. From the geometry of the quadcopter (Chapter 3.1.2) we know that motors 1 and 3 spin clockwise, while motors 2 and 4 spin counter-clockwise, which causes the torques generated by said motor pairs to be in opposite direction. This, along with (4.21), this yields

$$\tau_\psi = c(T_1 - T_2 + T_3 - T_4). \tag{4.22}$$

We define the torque vector as $\tau = \begin{bmatrix} \tau_\phi & \tau_\theta & \tau_\psi \end{bmatrix}^T$.

4.2.4 Equations of Motion

The Newton-Euler equations (4.23) characterize the translational and rotational dynamics of a rigid body, and can be written as

$$\begin{bmatrix} F^{\mathcal{B}} \\ \tau^{\mathcal{B}} \end{bmatrix} = \begin{bmatrix} \mathrm{mI}_3 & 0 \\ 0 & I_{cm} \end{bmatrix} \begin{bmatrix} {a_{cm}}^{\mathcal{B}} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} \omega \times \mathrm{m}v^{\mathcal{B}} \\ \omega \times I_{cm}\omega \end{bmatrix}, \tag{4.23}$$

where $\mathbf{I}_3$ is the 3×3 identity matrix, ${a_{cm}}^{\mathcal{B}}$ represents the linear accelerations in the body frame, I_{cm} represents the inertia matrix (4.24), ω represents the vector of the angular velocities $\begin{bmatrix} p & q & r \end{bmatrix}^T$ and $\dot{\omega}$ represents the angular acceleration. Under the assumption that the quadcopter is symmetric, the inertia matrix becomes

$$I_{cm} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}. \tag{4.24}$$

When the reference frame is fixed to the center of mass of the rotating body, (4.23) can then be simplified as

$$\begin{bmatrix} F^{\mathcal{B}} \\ \tau^{\mathcal{B}} \end{bmatrix} = \begin{bmatrix} \mathrm{m}\mathbf{I}_3 & 0 \\ 0 & I_{cm} \end{bmatrix} \begin{bmatrix} {a_{cm}}^{\mathcal{B}} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} 0 \\ \omega \times I_{cm}\omega \end{bmatrix}. \tag{4.25}$$

Using (4.25) and (4.11), accounting for our assumption that the aerodynamic drag at the given speeds is negligible, and accounting for the gravitational force g as well, we can derive the translational dynamics of the quadcopter in the body frame and rotate them into the inertial frame,

$$\begin{cases} \ddot{x} &= -\frac{T}{\mathrm{m}}(\sin\phi\sin\psi + \cos\phi\cos\psi\sin\theta) \\ \ddot{y} &= -\frac{T}{\mathrm{m}}(\cos\phi\sin\psi\sin\theta - \cos\psi\sin\phi) \\ \ddot{z} &= g - \frac{T}{\mathrm{m}}(\cos\phi\cos\theta) \end{cases}. \tag{4.26}$$

We can develop the rotational dynamics of the quadcopter in the body frame from (4.18)-(4.22) and (4.25),

$$\begin{cases} \tau_\phi = I_{xx}\dot{p} - I_{yy}qr + I_{zz}qr \\ \tau_\theta = I_{yy}\dot{q} + I_{xx}pr - I_{zz}pr \\ \tau_\psi = I_{zz}\dot{r} - I_{xx}pq + I_{yy}pq \end{cases}. \tag{4.27}$$

We can rearrange (4.27) to isolate the angular accelerations,

$$\begin{cases} \dot{p} = \frac{\tau_\phi}{I_{xx}} + \frac{I_{yy}-I_{zz}}{I_{xx}} qr \\ \dot{q} = \frac{\tau_\theta}{I_{yy}} + \frac{I_{zz}-I_{xx}}{I_{yy}} pr \\ \dot{r} = \frac{\tau_\psi}{I_{zz}} + \frac{I_{xx}-I_{yy}}{I_{zz}} pq \end{cases} . \tag{4.28}$$

By expanding (4.10), we receive:

$$\begin{cases} \dot{\phi} &= p + r\left(c_\phi t_\theta\right) + q(s_\phi t_\theta) \\ \dot{\psi} &= q(c_\phi) - r(s_\phi) \\ \dot{\theta} &= r\frac{c_\phi}{c_\theta} + q\frac{s_\phi}{c_\theta} \end{cases} . \tag{4.29}$$

4.2.5 Actuator Dynamics

A quadcopter is a system with six degrees of freedom, however, it has only four actuators. This means that our system is an under-actuated system, the outcome of this being that some motions will be uncontrollable at any given time and certain motions will be coupled to each other. Indeed, the quadcopter could not travel left or right without rolling first. Any controller must therefore take this into account. From the equations of motion (4.25) it is simple to see that the forces and the torques are the possible was to control the position and the orientation of the quadcopter. As force is only generated by the quadcopter in a single direction in the body frame and the torques can be generated along three axis, we find that there are four possible methods of controlling the quadcopter using our actuators. We begin by defining the four control inputs of the quadcopter,

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T = \begin{bmatrix} T & \tau_\phi & \tau_\theta & \tau_\psi \end{bmatrix}^T . \tag{4.30}$$

Where $\mathbf{u}$ is the control input vector composed of a thrust input, as well as rolling, pitching, and yawing moment inputs used to determine the rotational rates and linear accelerations of the quadcopter as per (4.11) and (4.28). Using (4.15), (4.18), (4.19), and (4.22), we can determine the motor speeds required to generate the required a given control input,

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \underbrace{\begin{bmatrix} k_m & k_m & k_m & k_m \\ \mathrm{L_{x_1}} k_m & -\mathrm{L_{x_2}} k_m & -\mathrm{L_{x_3}} k_m & \mathrm{L_{x_4}} k_m \\ \mathrm{L_{y_1}} k_m & \mathrm{L_{y_2}} k_m & -\mathrm{L_{y_3}} k_m & -\mathrm{L_{y_4}} k_m \\ ck_m & -ck_m & ck_m & -ck_m \end{bmatrix}}_{\mathrm{M}} \begin{bmatrix} \omega_{m_1}^2 \\ \omega_{m_2}^2 \\ \omega_{m_3}^2 \\ \omega_{m_4}^2 \end{bmatrix} . \tag{4.31}$$

where k_m is the constant defined in (4.14), c is taken from (4.16), and $\mathrm{L_{x_i}}$ and $\mathrm{L_{y_i}}$ are taken from Chapter 3.1.2. Further, by taking the inverse of (4.31) we can convert our

control input directly to the required motor outputs,

$$\begin{bmatrix} \omega_{m_1}^2 \\ \omega_{m_2}^2 \\ \omega_{m_3}^2 \\ \omega_{m_4}^2 \end{bmatrix} = \mathrm{M}^{-1} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}, \tag{4.32}$$

This is called the *motor-mixing algorithm* (MMA). The MMA recieves a control input from the control system, the desired forces and moments, and outputs desired motor speeds for each of the four motors. The motor speeds are then sent to the electronic speed controllers which provide the appropriate voltage ($\mathbf{V}_i$) to each of the four motors. The applied voltage spins the motors up to the correct speed and in the correct direction which, when combined with the propellers, generates the thrust (4.15) and torques (4.18),(4.19),(4.22) which then control the quadcopters dynamics (Figure. 4.2).

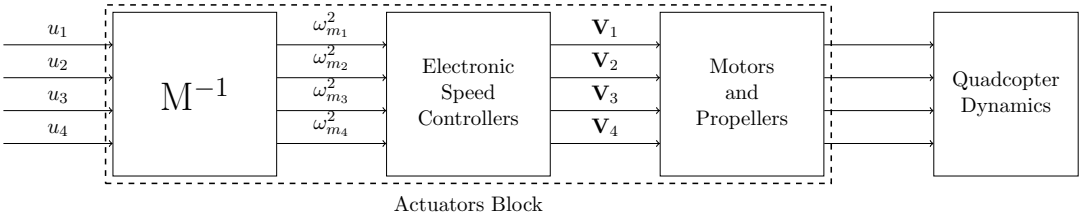


Figure 4.2: Block diagram of the actuators subsystem.

4.3 State Space Representation and Linearization

State-space representation is a mathematical model of a physics system by use of a set of input, output, and state variables. State space representation allows for the representation of a system in matrix form with the state of the system being represented as a vector. This representation leads to a convenient and compact method of modeling and analyzing systems with multiple inputs and outputs.

4.3.1 State-Space Representation

The state vector χ is the vector containing the states determining where an object is located, how it is oriented and how it is moving. For the state space representation, we will only use the inertial frame. As such, to simplify notation, the superscript $\mathcal{I}$ will not be used. We define $\eta = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^T$ as the Euler angle vector. From these, we can define the state vector,

$$\chi = \begin{bmatrix} \mathbf{p^T} & \eta^T & \mathbf{v}^T & \dot{\eta}^T \end{bmatrix}^T = \begin{bmatrix} \chi_{1-3}^T & \chi_{4-6}^T & \chi_{7-9}^T & \chi_{10-12}^T \end{bmatrix}^T, \tag{4.33}$$

where χ_{i-j} represents index i to j of the vector. Taking the derivative of the state space vectors, we find

$$\dot{\chi} = \begin{bmatrix} \mathbf{v}^T & \dot{\eta}^T & {a_{cm}}^T & \ddot{\eta}^T \end{bmatrix}^T = \begin{bmatrix} \dot{\chi}_{1-3} & \dot{\chi}_{4-6} & \dot{\chi}_{7-9} & \dot{\chi}_{10-12} \end{bmatrix}^T. \tag{4.34}$$

From this, and (4.26), (4.28), (4.8), and (4.10), we derive the nonlinear model (4.35).

$$\dot{\chi} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \frac{-T}{\mathrm{m}}(s_\phi s_\psi + c_\phi c_\psi s_\theta) \\ \frac{-T}{\mathrm{m}}(c_\phi s_\psi s_\theta - c_\psi s_\phi) \\ g - \frac{T}{\mathrm{m}}(c_\phi c_\theta) \\ \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \chi_7 \\ \chi_8 \\ \chi_9 \\ \chi_{10} \\ \chi_{11} \\ \chi_{12} \\ \frac{-u_1}{\mathrm{m}}(s_{\chi_4} s_{\chi_6} + c_{\chi_4} c_{\chi_6} s_{\chi_5}) \\ \frac{-u_1}{\mathrm{m}}(c_{\chi_4} s_{\chi_6} s_{\chi_4} - c_{\chi_6} s_{\chi_4}) \\ g - \frac{u_1}{\mathrm{m}}(c_{\chi_4} c_{\chi_5}) \\ g_1(\chi, u) \\ g_2(\chi, u) \\ g_3(\chi, u) \end{bmatrix} = f(\chi, u), \tag{4.35}$$

where the functions $g(\chi, u) = \begin{bmatrix} g_1(\chi, u) & g_2(\chi, u) & g_3(\chi, u) \end{bmatrix}^T$ are equal to,

$$\begin{aligned} \begin{bmatrix} g_1(\chi, u) \\ g_2(\chi, u) \\ g_3(\chi, u) \end{bmatrix} &= I^{-1} R \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} \\ &= I^{-1} \left(u_2 \begin{bmatrix} c_{\chi_6}\, c_{\chi_5} \\ c_{\chi_5}\, s_{\chi_{12}} \\ -s_{\chi_5} \end{bmatrix} + u_3 \begin{bmatrix} c_{\chi_6}\, s_{\chi_4}\, s_{\chi_5} - c_{\chi_4}\, s_{\chi_6} \\ c_{\chi_4}\, c_{\chi_6} + s_{\chi_4}\, s_{\chi_6}\, s_{\chi_5} \\ c_{\chi_5}\, s_{\chi_4} \end{bmatrix} + u_4 \begin{bmatrix} s_{\chi_4}\, s_{\chi_6} + c_{\chi_4}\, c_{\chi_6}\, s_{\chi_5} \\ c_{\chi_4}\, s_{\chi_6}\, s_{\chi_5} - c_{\chi_6}\, s_{\chi_4} \\ c_{\chi_4}\, c_{\chi_5} \end{bmatrix} \right). \end{aligned} \tag{4.36}$$

4.3.2 Linearization of the State Space

Developing a linearized state space of the quadcopter will allow us to use classical control methods to control the state of the quadcopter. The key to linearization is finding the equilibrium point of the system, or where $\dot{\chi} = 0$. From (4.35) and (4.36) we

can determine that the equilibrium vector is where the following assumption hold true,

$$\begin{cases} T = \mathrm{m}g \\ \dot{x} = \dot{y} = \dot{z} = 0 \Longleftrightarrow u = v = w = 0 \\ \phi = \theta = 0 \\ \dot{\phi} = \dot{\theta} = \dot{\psi} = 0 \Longleftrightarrow p = q = r = 0 \end{cases} \tag{4.37}$$

This yields the equilibrium state vector,

$$\chi_{eq} = \begin{bmatrix} \bar{x} & \bar{y} & \bar{z} & 0 & 0 & \bar{\psi} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T, \tag{4.38}$$

where the $\bar{(\,)}$ operator denotes that the given parameter can have any real value. It is worth noting that at this equilibrium point, the rotation matrix in (4.10) is equal to the identity matrix, and therefore, $\dot{\eta}$=ω. Linearizing the system around this point yields:

$$\dot{\chi} = A(\chi - \chi_{eq}) + B(u - u_{eq}), \tag{4.39}$$

Where A and B are the Jacobian matrices,

$$\begin{cases} A = \frac{\partial f}{\partial \chi}(\chi, u)|_{(\chi,u)=(\chi_{eq},u_{eq})} \\ \\ B = \frac{\partial f}{\partial u}(\chi, u)|_{(\chi,u)=(\chi_{eq},u_{eq})} \end{cases} . \tag{4.40}$$

This yields

$$\dot{\chi} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -gs_{\psi_{eq}} & -gc_{\psi_{eq}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & gc_{\psi_{eq}} & -gs_{\psi_{eq}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_{xx}} & 0 & 0 \\ 0 & 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}}_{B} \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}. \tag{4.41}$$

Since the equilibrium heading angle may be any arbitrary value, for sake of convenience, we may set our equilibrium heading angle as our initial heading angle. As the initial state of the quadcopter is used to define our inertial reference frame, this yields $\psi_{eq} = \psi_0 = 0°$.

This allows us to further simplify our state space matrices to,

$$\dot{\chi} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -g & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_{xx}} & 0 & 0 \\ 0 & 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix} \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}. \tag{4.42}$$

The system output ($y = \mathbf{C}x + \mathbf{D}u$) is dependent on the sensors of the system. If we were to assume perfect information, the matrix $\mathbf{C}$ would be the identity matrix I_{12}, however in the real world, this is rarely, if ever, the case. As such, we require the quadcopter design to incorporate appropriate sensors (Chapter 3) and the proper implementation of state estimators (Chapter 5).

Chapter 5

Guidance, Navigation, and Control System Implementation

This chapter deals with the implementation of GNC systems on the research platform. This includes the implementation of GNC systems in the simulation environment, and in the on-board flight code. This chapter deals with the implementation of the pre-built controllers, estimators, and GNC systems. Additionally, it will present a general overview as to the implementation of new controllers, estimators and GNC system by the user. It is important to note that while each GNC system can be designed in continuous time, the hardware itself is digital, and thus runs in discrete time. As such, for any such GNC system designed in continuous time from the start, it must be discretized before being implemented into either the simulator or the flight code.

5.1 Guidance

The guidance system provides the desired trajectory and states of the vehicle to the control system. It does this by taking inputs from the navigation system, processing them using a guidance law, and outputting the result to the control system (Figure 2.1).

5.1.1 Simulator Implementation

The simulator implements guidance systems via `.m`-files. Any guidance system files stored in the guidance library folder (Figure 3.16) may be accessed by the system. Implementation of any existing guidance system can be achieved via selecting the guidance subsystem block (Figure 3.22) and selecting the desired system from the drop-down menu (Figure. 3.23).

Implementation of a new guidance system may be performed simple by adding a new `.m`-file to the guidance library. A template for the guidance system `.m`-file can be found in Snippet 2. The guidance system inputs are the time since the start of the simulation, and the state estimates and the calibrated sensor readings, received from

the navigation system. A twelve state reference bus matching the quadcopter states (4.33) is then declared to provide the output of the system. It is important to declare this bus as it is the input that the controller is expecting to receive. Unused states may be set to zero, or in cases where they do not affect the control, to NaN if desired.

Once the reference state vector is defined, the guidance law may be implemented with each calculation or consideration setting the appropriate reference signal. The output is then sent to the controller.

```
function Ref = GuidanceTemplate(time, state,y)

% State Estimates
StateEstim = state;

% Sensor Model
Sensors = y;

%% Guidance Law
% Reference State
Ref.x          = 0;    % [m]
Ref.y          = 0;    % [m]
Ref.z          = 1;    % [m]
Ref.phi        = 0;    % [rad]
Ref.theta      = 0;    % [rad]
Ref.psi        = 0;    % [rad]
Ref.dx         = 0;    % [m/s]
Ref.dy         = 0;    % [m/s]
Ref.dz         = 0;    % [m/s]
Ref.dphi       = 0;    % [deg/sec]
Ref.dtheta     = 0;    % [deg/sec]
Ref.dpsi       = 0;    % [deg/sec]

%% Set Guidance Law Here
% Reference States Output is sent to Control System

end
```

Snippet 2: Guidance system template for the flight simulator.

Two forms of guidance systems included in the platform, an open-loop guidance system (Snippet 22) and waypoint guidance (Snippets 23, 24) can be found in Appendix H. It is worth noting that in the waypoint based guidance laws, a `persistent` variable type is used. This variable type preserves the value of the variable between function calls, allowing for the simulator to repeatedly call the guidance law, without losing the data from the previous time steps.

5.1.2 Flight Code Implementation

The flight code is all written in C++. This generally would make the flight code less accessible to those without a strong programming background. To alleviate this, measures were taken to ensure that implementation of guidance systems, both existing and of new user created system should be as simple and straight forward as possible. From Snippet 1, we can see in the flight code set-up a line of code that begins with:

`auto guidance =`

This is where the guidance system is implemented in the main function. Succeeding `auto guidance =` the user may simply input any of the existing guidance systems. The guidance systems are found in the file `Guidance.hpp`. A template guidance system can be found at the top of the file, and in Snippet 3. During each update step, the guidance system will take the sensors readings and state estimates from the navigation system, and output a desired state as the reference signal for the control system, as described in Figure 3.36.

```
class GuidanceTemplate : public Guidance {
    // Choose desired reference states and define them as floats
    float z;

    public:
    GuidanceTemplate(float z) {
        // Set pointers to desired states
        this->z = z;
    }

    void update(SensorReadings &readings, StateVector &state, StateVector &desired_state) {
        // Update desired states based off of guidance law
        desired_state.z = z;
    }
};
```

Snippet 3: Guidance system template for the flight code.

In any guidance system, undefined references are automatically set to zero. To provide further examples of guidance systems, two guidance systems are provided, a go-to-point guidance system in Snippet 25, and a waypoint guidance system in Snippet 26 can be found in Appendix H.

5.2 Navigation

The navigation system provides a state estimate of the vehicle. It does this by taking inputs from the sensors, processing them using estimators and outputting the result to the guidance system and the control system. An explanation of each of the included guidance systems may be found in Appendix I. This section will focus on their implementation.

5.2.1 Simulator Implementation

The simulator implements navigation systems via `.m`-files. Any navigation system files stored in the navigation library folder (Figure 3.16) may be accessed by the system. Implementation of any existing navigation system can be achieved via selecting the navigation subsystem block (Figure 3.34) and selecting the desired system from the drop-down menu (Figure. 3.35).

Implementation of a new navigation system may be performed simple by adding a new `.m`-file to the navigation library. A template for the navigation system `.m`-file can be found in Snippets 4, 5, and 6. The navigation system inputs are the time delta (dt) between sensor readings, and the calibrated sensor readings. A twelve state state estimate bus matching the quadcopter states (4.33) is then declared to provide the output of the system. It is important to declare this bus as it is the input that the guidance and control systems are expecting to receive. Additionally, a second twelve state state vector is defined as a `persistent` variable. This vector is used to store the previous state which is required for integration and derivation of the states in discrete time. The navigation template file displays four different estimators (attitude, angular rate, vertical position/velocity, and lateral position/velocity), each one using a separate type of state estimator (complementary filter, low-pass filter, direct sensor reading/calculation, and Kalman filter respectively). The standard navigation system used in the simulator is a complementary filter for the attitude, direct sensor readings for the angular rate, and Kalman filters for the height and lateral position estimators respectively.

```
function StatesEstim = EstimatorTemplate(dt, Sensors_calib)

%% Define Sensor and State Inputs
Sensors = Sensors_calib;

%% Define State Estimate Structure
StatesEstim.x       = 0;
StatesEstim.y       = 0;
StatesEstim.z       = 0;
StatesEstim.phi     = 0;
StatesEstim.theta   = 0;
StatesEstim.psi     = 0;
StatesEstim.dx      = 0;
StatesEstim.dy      = 0;
StatesEstim.dz      = 0;
StatesEstim.dphi    = 0;
StatesEstim.dtheta  = 0;
StatesEstim.dpsi    = 0;
```

Snippet 4: Navigation system template for the flight simulator: Part 1.

```
%% Define Previous State Estimate
persistent prev_state
if isempty(prev_state)
    prev_state.x       = 0;
    prev_state.y       = 0;
    prev_state.z       = 0;
    prev_state.phi     = 0;
    prev_state.theta   = 0;
    prev_state.psi     = 0;
    prev_state.dx      = 0;
    prev_state.dy      = 0;
    prev_state.dz      = 0;
    prev_state.dphi    = 0;
    prev_state.dtheta  = 0;
    prev_state.dpsi    = 0;
end
%% Attitude Estimation - Complementary Filter
% Required Constants
g         = 9.81;
alpha     = 0.99;
alpha_psi = 0.95;

% Roll
StatesEstim.phi       = atan(Sensors.a_y/Sensors.a_z)*(1-alpha)+...
                        alpha*(prev_state.phi+Sensors.p*dt);    % [rad]
% Pitch
StatesEstim.theta     = atan((-Sensors.a_x/sqrt(Sensors.a_y^2+Sensors.a_z^2))*0.5)...
                        *(1-alpha)+alpha*(prev_state.theta+Sensors.q*dt); % [rad]
% Yaw
% Normalize the Magnetometer
mag_norm = sqrt(Sensors.m_x^2+Sensors.m_y^2+Sensors.m_z^2);
if mag_norm == 0
    mag_norm = 1;
end
Sensors.m_x           = Sensors.m_x/mag_norm;
Sensors.m_y           = -Sensors.m_y/mag_norm;
Sensors.m_z           = Sensors.m_z/mag_norm;
% Tilt Compensate the Magnetometer
Mx = Sensors.m_x*cos(StatesEstim.phi)+Sensors.m_z*sin(StatesEstim.phi);
My = Sensors.m_x*sin(StatesEstim.theta)*sin(StatesEstim.phi)+...
     Sensors.m_y*cos(StatesEstim.theta)-...
    Sensors.m_z*sin(StatesEstim.theta)*cos(StatesEstim.phi);
StatesEstim.psi       = atan2(-My,Mx)*(1-alpha_psi)+...
                        alpha_psi*(prev_state.psi+Sensors.r*dt); % [rad]

%% Wrap Attitude Estimate to Pi and Eliminate NaNs

% Phi
if isnan(StatesEstim.phi)
    StatesEstim.phi = (prev_state.phi+Sensors.p*dt);
end
StatesEstim.phi =(mod((StatesEstim.phi+pi),(2*pi))-pi);

% Theta
if isnan(StatesEstim.theta)
    StatesEstim.theta = (prev_state.theta+Sensors.q*dt);
end
StatesEstim.theta = (mod((StatesEstim.theta+pi),(2*pi))-pi);

% Psi
if isnan(StatesEstim.psi)
    StatesEstim.psi = (prev_state.psi+Sensors.r*dt);
end
StatesEstim.psi = (mod((StatesEstim.psi+pi),(2*pi))-pi);
```

Snippet 5: Navigation system template for the flight simulator: Part 2.

```
%% Rotation Rates - Low-Pass Filter
alpha = 0.98; % alpha = tau / (tau + dt)
StatesEstim.dphi      = (1-alpha)*Sensors.p + alpha*(prev_state.dphi);    % [rad/sec]
StatesEstim.dtheta    = (1-alpha)*Sensors.q + alpha*(prev_state.dtheta);  % [rad/sec]
StatesEstim.dpsi      = (1-alpha)*Sensors.r + alpha*(prev_state.dpsi);    % [rad/sec]

%% Vertical Position Estimation - Direct Sensor Read and Direct Calculation
StatesEstim.z         = Sensors.ultra;                    % [m]
StatesEstim.dz        = (prev_state.z-Sensors.ultra)/dt; % [m/s]

%% Lateral Position Estimation - Kalman Filter
persistent P_lat xhat_lat F_lat B_lat H_lat Q_lat R_lat
meas_lat = [Sensors.dx;
            Sensors.dy];
u_lat    = [Sensors.a_x;
            Sensors.a_y];

if isempty(P_lat)
% First time through the code so do some initialization
   xhat_lat = zeros(4,1);
   P_lat = zeros(4);
   H_lat = [0 0 1 0;
            0 0 0 1];
   R_lat = 0.001^2*eye(2);
end

% Update dt for all matrices
F_lat = [1 0  dt 0;
         0 1  0 dt;
         0 0  1 0;
         0 0  0 1];
B_lat = [dt*dt/2 0;
         0       dt*dt/2;
         dt      0;
         0       dt];
Q_lat = [0.25*dt^4 0          0.5*dt^3  0;
         0         0.25*dt^4  0         0.5*dt^3;
         0.5*dt^3  0          dt^2      0;
         0         0.5*dt^3      0      dt^2]*0.01^2;
% Propagate the state estimate and covariance matrix:
xhat_lat = F_lat*xhat_lat + B_lat*u_lat;
P_lat = F_lat*P_lat*F_lat' + Q_lat;

% Calculate the Kalman gain
K = P_lat*H_lat'/(H_lat*P_lat*H_lat' + R_lat);

% Calculate the measurement residual
resid = meas_lat - H_lat*xhat_lat;

% Update the state and error covariance estimate
xhat_lat = xhat_lat + K*resid;
P_lat = (eye(size(K,1))-K*H_lat)*P_lat;

% Assign Kalman Filter Outputs to Correct States
StatesEstim.x         = xhat_lat(1);    % [m]
StatesEstim.y         = xhat_lat(2);    % [m]
StatesEstim.dx        = xhat_lat(3);    % [m/s]
StatesEstim.dy        = xhat_lat(4);    % [m/s]

%% Define Persistent Variables for Next Iteration

prev_state = StatesEstim;

end
```

Snippet 6: Navigation system template for the flight simulator: Part 3.

5.2.2 Flight Code Implementation

As with the guidance system implementation, the goal is to maximize accessibility to the flight code to people wihtout a strong programming background. From Snippet 1, we can see in the flight code set-up a line that begins with:

`std::list<Package> packages =`

This is where the navigation system is implemented in the main function. Succeeding the `std::list<Package> packages =` the user may simply input a list of estimator packages, which will be executed in the listed order. Estimator packages are merely a number of state estimators packaged together into a single unit, with each estimator providing an estimate for a given state/set of states and the package itself estimating the group of states. For example, an attitude estimator package may include three state estimators, each one working to estimate a given state such as pitch, roll, and yaw angles.

These estimator packages are then fed into the `Navigation` class on the next line to form the navigation system. The estimator packages are in turn made up of estimators which are defined in the files `ReadyFilters.hpp` and `ReadyFilters.cpp`. Each estimator has its functions named in the header file `ReadyFilters.hpp`, and then defined in the `ReadyFilters.cpp` file. A template estimator, which also functions to provide direct sensor readings, can be at the top of the respective files, and in Snippet 7. Additional estimators including low-pass (Snippet 27), high-pass (Snippet 28), angular complementary (Snippet 29) and Kalman filters (Snippets 30, 31, 32, and 33) can be found in Appendix I.

```
1  // From ReadyFilters.hpp
2  class Verbatim : public Estimator {
3      private:
4      sensor_field_ptr field_from;
5      state_field_ptr field_to;
6
7      public:
8      Verbatim(sensor_field_ptr field_from, state_field_ptr field_to);
9      void update(SensorReadings &readings, StateVector &state) override;
10 };
11
12 // From ReadyFilters.cpp
13 Verbatim::Verbatim(sensor_field_ptr field_from, state_field_ptr field_to) {
14     this->field_from = field_from;
15     this->field_to = field_to;
16 }
17
18 void Verbatim::update(SensorReadings &readings, StateVector &state) {
19     state.*field_to = readings.*field_from;
20 }
```

Snippet 7: Estimator template for the flight code.

Using the estimators, estimator packages can be defined in the file `EstPackages.hpp`.

A sample of estimator packages can be found in Snippet 8. Implementation of new

```
AngularComplementaryFilter _pitch = AngularComplementaryFilter(
    &SensorReadings::acc_pitch, &SensorReadings::gyro_q, &StateVector::pitch);
AngularComplementaryFilter _roll = AngularComplementaryFilter(
    &SensorReadings::acc_roll, &SensorReadings::gyro_p, &StateVector::roll);
AngularComplementaryFilter _yaw = AngularComplementaryFilter(
    &SensorReadings::magneto_yaw, &SensorReadings::gyro_r, &StateVector::yaw);
Package AttitudeComplementaryPackage = {_pitch, _roll, _yaw};

auto _kf = XY_KalmanFilter();
Package LateralPosition_Kalman = { _kf };

auto _kf_Alt = Altitude_KalmanFilter();
Package Altitude_Kalman = { _kf_Alt };

auto _p = Verbatim(&SensorReadings::gyro_p, &StateVector::p);
auto _q = Verbatim(&SensorReadings::gyro_q, &StateVector::q);
auto _r = Verbatim(&SensorReadings::gyro_r, &StateVector::r);
Package Verbatim_pqr = {_p, _q, _r};

auto _p = LowPass(&SensorReadings::gyro_p, &StateVector::p);
auto _q = LowPass(&SensorReadings::gyro_q, &StateVector::q);
auto _r = LowPass(&SensorReadings::gyro_r, &StateVector::r);
Package LowPass_pqr = {_p, _q, _r};
```

Snippet 8: Estimator packages examples for the flight code.

estimator packages is as simple as defining the estimator for each state, then grouping them into the package list.

5.3 Control

The control system provides the input signal to the actuators which control the dynamics of the vehicle. It does this by taking inputs from the navigation system, and comparing them with the inputs from the guidance system to generate a state error. The state error is then processed by the controllers which then output the results to the actuators(Figure 2.1). An explanation of each of the included controllers and control systems may be found in Appendix K. This section will focus on their implementation.

5.3.1 Simulator Implementation

The simulator implements control systems via `.m`-files. Any control files stored in the controller library folder (Figure 3.16) may be accessed by the system. Implementation of any existing control system can be achieved via selecting the control subsystem block (Figure 3.28) and selecting the desired system from the drop-down menu (Figure. 3.29).

Since the dynamics of the quadcopter are linearized around the hover point, they intrinsically include the adjustment for the equilibrium trim to linearized control (mg) in the thrust controller output while using the linear dynamics model in the simulator. This is not the case in the nonlinear dynamics model. To compensate for this, and

prevent the need to modify the controller when switching between dynamic model types, the simulator automatically adds the equilibrium trim when the nonlinear dynamics model is active. For nonlinear control strategies, this addition may be removed when defining the thrust control signal output.

Implementation of a new control system may be performed simple by adding a new `.m`-file to the navigation library. A template for the control system `.m`-file can be found in Snippets 9, and 10. The control system inputs are the state estimates and the sensor readings from the navigation system, and the reference signals from the guidance system. Three separate twelve state vectors are defined as a `persistent` variables, one to retain the previous state, the second to retain the previous reference signal, and the third to retain the previous error. These vectors are required for integration and derivation of the states, reference signals, and error signals in discrete time. The output of the control system is a four state signal bus, representing the thrust command, along with the commands for the desired rolling, pitching, and yawing motions. The controller template file displays a generic cascaded PID controllers (attitude and height inner-loop, and lateral position outer-loop). The standard control systems used in the simulator used are either a cascaded PID control system, or an LQR control system.

Once the control signal bus is defined, and the control signals have been determined, the control signals are sent to the actuator subsystem which converts the commands from thrust and moments to desired motor speeds. These speeds are then used to determine the generated thrusts and moments by the motors, which are then sent to the quadcopter dynamics subsystem to determine the dynamics and kinematics that the vehicle have in response.

Four control systems are included in the platform, two inner-loop only control systems and two control systems which control the full state. The two inner-loop control systems are a PID control system (Snippets 34, 35) and and LQR control system (Snippets 36, 37) can be found in Appendix L. Additionally a cascaded PID control system (Snippets 38, 39) and an full state LQR control system (Snippets 40, 41) can be found in Appendix L.

5.3.2 Flight Code Implementation

As with the navigation and guidance systems implementation, the goal is to maximize accessibility to the flight code to people without a programming background. From Snippet 1, we can see in the flight code set-up towards the botoom of the user setup section two lines which contain the variables `csys` and `recovery_csys`. This is where the control system is implemented in the main function. Proceeding the `csys` the user may simply input the name of the control system class which will run the quadcopter. The class proceeding the `recovery_csys` should be a control system that is known to work well for the quadcopter in order to enable recovery of the quadcopter if it begins to have unstable behaviour. The control system is made up of or more controllers. The

controllers are contained in the file `LibraryControllers.hpp`. Each controller is its own class, which can then be called by the control system. Two controller types can be found built into the platform, a PID controller (Snippet 42), and a full state feedback controller (Snippet 43). Both controllers can be found in Appendix J.

Using the controllers, control systems can be defined in the file `LibraryControlSystems.hpp`. An inner-loop PID control system is used as a template and can be found can be found in Snippet 11. Additional control systems including a cascading PID control system (Snippet 44), inner-loop LQR (Snippet 45), and full state LQR (Snippet 46) can be found in Appendix J.

```
function [u, comp_ref, comp_error] = NewController(dt, state, ref , Sensors)
%% Persistent Variables
persistent prev_state prev_ref prev_error %prev_comp_error

if isempty(prev_state)
    prev_state.x         = 0;   % [m]
    prev_state.y         = 0;   % [m]
    prev_state.z         = 0;   % [m]
    prev_state.phi       = 0;   % [deg]
    prev_state.theta     = 0;   % [deg]
    prev_state.psi       = 0;   % [deg]
    prev_state.dx        = 0;   % [m/s]
    prev_state.dy        = 0;   % [m/s]
    prev_state.dz        = 0;   % [m/s]
    prev_state.dphi      = 0;   % [rad/sec]
    prev_state.dtheta    = 0;   % [rad/sec]
    prev_state.dpsi      = 0;   % [rad/sec]
end
if isempty(prev_ref)
    prev_ref.x           = 0;   % [m]
    prev_ref.y           = 0;   % [m]
    prev_ref.z           = 0;   % [m]
    prev_ref.phi         = 0;   % [deg]
    prev_ref.theta       = 0;   % [deg]
    prev_ref.psi         = 0;   % [deg]
    prev_ref.dx          = 0;   % [m/s]
    prev_ref.dy          = 0;   % [m/s]
    prev_ref.dz          = 0;   % [m/s]
    prev_ref.dphi        = 0;   % [rad/sec]
    prev_ref.dtheta      = 0;   % [rad/sec]
    prev_ref.dpsi        = 0;   % [rad/sec]
end
if isempty(prev_error)
    prev_error.x         = 0;   % [m]
    prev_error.y         = 0;   % [m]
    prev_error.z         = 0;   % [m]
    prev_error.phi       = 0;   % [deg]
    prev_error.theta     = 0;   % [deg]
    prev_error.psi       = 0;   % [deg]
    prev_error.dx        = 0;   % [m/s]
    prev_error.dy        = 0;   % [m/s]
    prev_error.dz        = 0;   % [m/s]
    prev_error.dphi      = 0;   % [rad/sec]
    prev_error.dtheta    = 0;   % [rad/sec]
    prev_error.dpsi      = 0;   % [rad/sec]
end

%% Required Values
g = 9.81;
m = 0.0630;

%% Define Error Terms
error = structminus(ref,state);

%% Define Controllers

% Inner Loop Controllers
% Altitude Controller
C_alt.kp = 0.8;
C_alt.ki = 0;
C_alt.kd = 0.3;
% Roll Controller
C_r.kp = 0.01;
C_r.ki = 0.01;
C_r.kd = 0.0028;
```

Snippet 9: Control system template for the flight simulator: Part 1.

```
% Pitch Controller
C_p.kp = 0.013;
C_p.ki = 0.01;
C_p.kd = 0.002;
% Yaw Controller
C_y.kp = 0.004;
C_y.ki = 0;
C_y.kd = 0.012;

% Outer Loop Controllers
% X Positional Controller
% C_X is positive here instead of C_Y. Unsure why.
C_X.kp           = 0.24;
C_X.ki           = 0;
C_X.kd           = 0.1;

% Set Outer Loop Controller Output as Inner Loop Controller Reference
ref.theta        = (C_X.kp*error.x + C_X.ki*(prev_error.x+error.x*dt) +...
                    C_X.kd*((error.x-prev_error.x)/dt));
% Ensure the reference angle is wrapped to pi
ref.theta        = (mod((ref.theta+pi),(2*pi))-pi);
% Define Error Signal based off of new Reference Signal
error.theta      = ref.theta-state.theta;
% Wrap error angle to pi
error.theta      = (mod((error.theta+pi),(2*pi))-pi);

% Y Positional Controller
% NOTE: C_Y is negative here instead of C_X. Unsure why
C_Y.kp         = -0.24;
C_Y.ki         = 0;
C_Y.kd         = -0.1;

% Set Outer Loop Controller Output as Inner Loop Controller Reference
ref.phi        = (C_Y.kp*error.y + C_Y.ki*(prev_error.y+error.y*dt) +...
                  C_Y.kd*((error.y-prev_error.y)/dt));
% Ensure the reference angle is wrapped to pi
ref.phi        = (mod((ref.phi+pi),(2*pi))-pi);

% Define Error Signal based off of new Reference Signal
error.phi      = ref.phi-state.phi;
error.phi      = (mod((error.phi+pi),(2*pi))-pi); % Wrap error angle to pi


%% Controller Output

% Control Output
u.u1 = ( C_alt.kp*error.z   + C_alt.ki*(prev_error.z+error.z*dt)       +...
        C_alt.kd*(error.z-prev_error.z)/dt       ) + m*g;  % Thrust
u.u2 = ( C_r.kp*error.phi   + C_r.ki*(prev_error.phi+error.phi*dt)     +...
        C_r.kd*(error.phi-prev_error.phi)/dt     );        % Roll
u.u3 = ( C_p.kp*error.theta + C_p.ki*(prev_error.theta+error.theta*dt) +...
        C_p.kd*(error.theta-prev_error.theta)/dt );        % Pitch
u.u4 = ( C_y.kp*error.psi   + C_y.ki*(prev_error.psi+error.psi*dt)     +...
        C_y.kd*(error.psi-prev_error.psi)/dt     );        % Yaw

% Computed Reference Signals
comp_ref = ref;

% Computed Error Signals
comp_error = error;

prev_state = state;
prev_ref = ref;
prev_error = error;
end
```

Snippet 10: Control system template for the flight simulator: Part 2.

```cpp
class InnerLoopPID : public ControlSystem {
    private:
    struct pids {
        PID pitch, roll, yaw, thrust;
        /* data */
    } pids;

    public:
    InnerLoopPID();
    void Control(StateVector &current_state, StateVector &desired_state,
                 MomentThrustCommand &output_command) override;
};

InnerLoopPID::InnerLoopPID() {
    // float Kpz, Kiz, Kdz, Kpyaw, Kiyaw, Kdyaw;
    pids.thrust = PID(0.845, 0, 0.796);
    pids.roll = PID(0, 0, 0);
    pids.roll = PID(0.05, 0, 0.01);
    pids.pitch = PID(0.08, 0, 0);
    pids.yaw = PID(0.001, 0, 0.0001);
}

void InnerLoopPID::Control(StateVector &current_state, StateVector &desired_state,
                           MomentThrustCommand &output_command) {

    output_command.thrust = pids.thrust.update(desired_state.z - current_state.z) + mass;
    output_command.roll = pids.roll.update(desired_state.roll -
                                           current_state.roll, desired_state.p);
    output_command.pitch = pids.pitch.update(desired_state.pitch -
                                           current_state.pitch, desired_state.q);
    output_command.yaw = pids.yaw.update(desired_state.yaw - current_state.yaw);
}
```

Snippet 11: Control systems template for the flight code.

Chapter 6

Verification, Validation, and Results

Verification and validation of the platform are critical aspects for ensuring that the platform is suitable for research and design purposes. In this chapter, we will discuss the verification of the simulator models, validation of the model via comparison to hardware tests on a known vehicle, and the results of tests implemented on the platform via the implementation workflow.

6.1 Simulator Model Verification

To determine whether or not our simulator platform is suitable for use, the quadcopter dynamics model and the actuator models must first be validated. To this end, the system dynamics were verified on the linear and nonlinear models of the quadcopter.

6.1.1 Linear Model Verification

We begin with the verification of the linear model. As per (4.38), the quadcopter may be linearized around any x, y, and z coordinates and with any initial heading angle ψ. For these tests, we choose to set all of those parameters to zero. Additionally, it must be noted that while we are linearizing around an altitude of 0, that does *not* imply there is any sort of ground there. For purposes of these tests, the quadcopter exists in an endless void. This was done in two different manners. The first manner of verification was the direct input of forces and moments directly into the open-loop linearized quadcopter dynamics subsystem (Figure 6.1). It should be noted that for the figures representing each of tests in this section, the direction of the z-axis was inverted such that the positive direction show "up" rather than "down." This is to allow for a more intuitive understanding of the figures at a glance. In total, three classes of tests were performed in this manner. The first was an equilibrium test, in which no forces or moments where input into the system (Figure 6.2). Indeed, while it was not expected

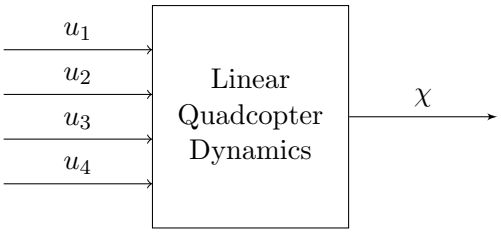
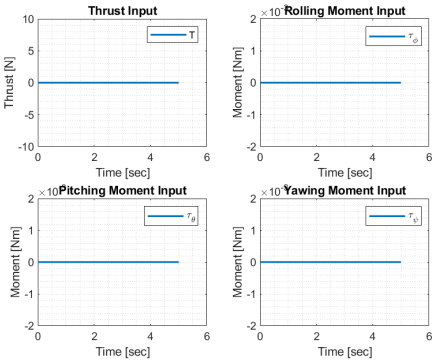
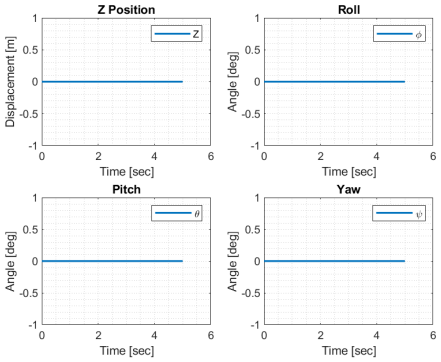


Figure 6.1: Simulated quadcopter linear dynamics validation using forces and moments inputs.

to show any results of particular interest as the quadcopter should simply maintain it's equilibrium state, it is critical for the test to show this to be true, otherwise the model would instantly be invalidated. The second class of test was one in which a



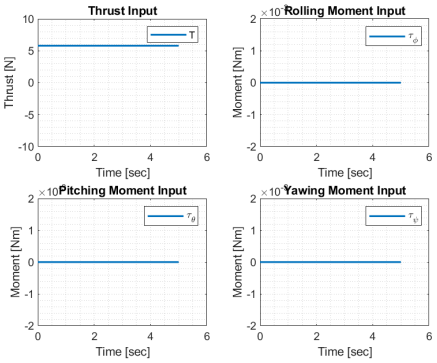
(a) Inputs.



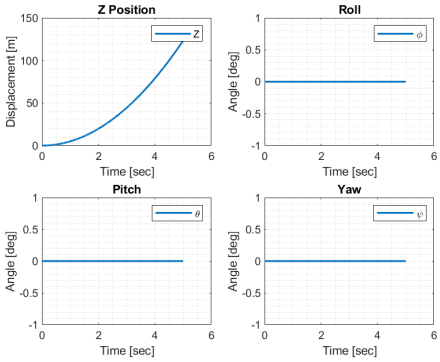
(b) States.

Figure 6.2: Model verification of the simulator's linear quadcopter dynamics with no inputs.

pure thrust input was input into the quadcopter's linearized dynamics. In Figure 6.3 a step input of $5.8[N]$ of thrust was input into the system while the moment commands remained as zero. It is worthwhile to note the significance of the value of the thrust input ($5.8[N]$). This value is just slightly higher than the required thrust for equilibrium trim of the quadcopter, allowing the quadcopter to rise both in the linear and nonlinear dynamic models. The third class of tests was one in which pure moment commands were input into the linearized quadcopter dynamics. Tests were performed to input pure rolling moment commands (Figure 6.4), pure pitching moment commands (Figure 6.5), and pure yawing moment commands (Figure 6.6). Each of these tests used a single moment step input of $0.001[Nm]$ around the given axis, with all other inputs set to zero. Inspecting the results of the aforementioned tests, and comparing them with our understanding of the quadcopter's model of linear dynamics (4.42), we found that the linear model reacted as anticipated to the given inputs.

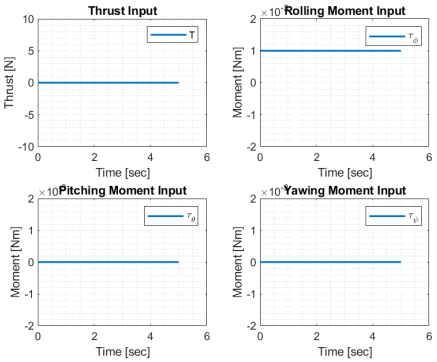


(a) Inputs.

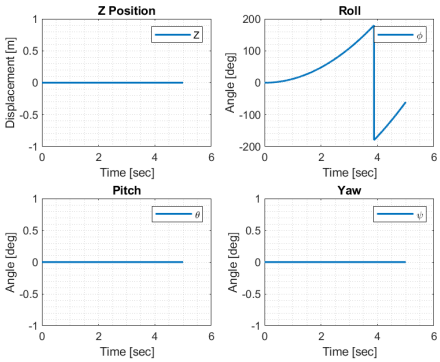


(b) States.

Figure 6.3: Model verification of the simulator's linear quadcopter dynamics via thrust step input.

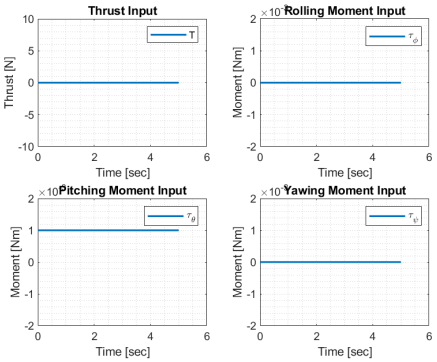


(a) Inputs.

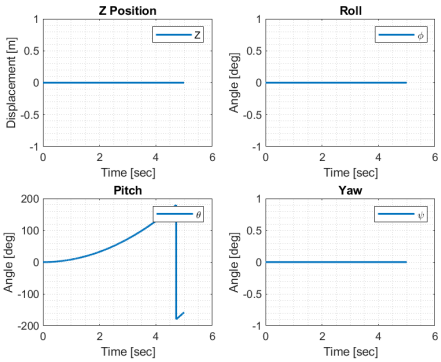


(b) States.

Figure 6.4: Model verification of the simulator's linear quadcopter dynamics via a rolling moment step input.

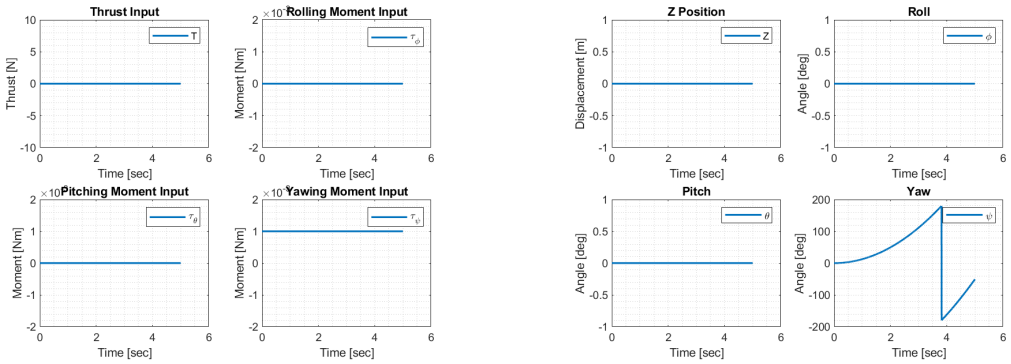


(a) Inputs.



(b) States.

Figure 6.5: Model verification of the simulator's linear quadcopter dynamics via a pitching moment step input.



(a) Inputs. (b) States.

Figure 6.6: Model verification of the simulator's linear quadcopter dynamics via a yawing moment step input.

6.1.2 Actuator Model Verification

Once we determined that the linear model of the quadcopter dynamics responds to given thrust and moment inputs, it was important to determine that the actuator generates said thrust and moments correctly to input to the dynamic model of the quadcopter. To this end, two classes of tests were performed. The first class of tests was to generate forces on a per-motor basis, and determine whether or not the appropriate thrust and moment inputs were sent to the linear quadcopter dyanmics (Figure 6.7). When all

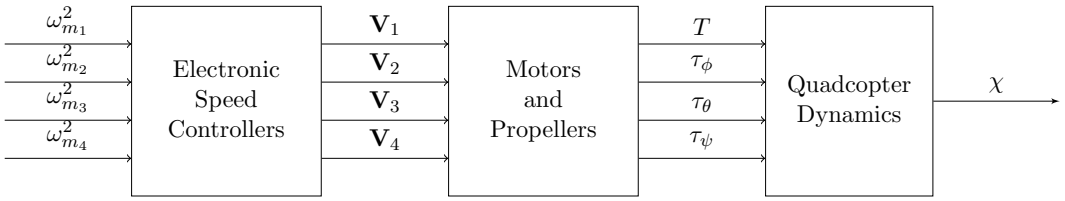
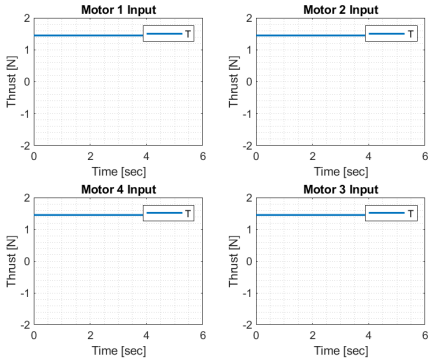


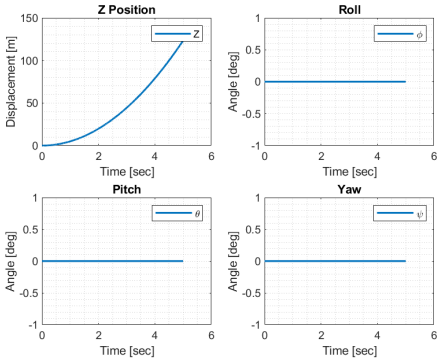
Figure 6.7: Simulated quadcopter linear dynamics validation using motor speed inputs.

four motors produce equal thrust (Figure 6.8) generates pure thrust in the z direction, as expected. To test the generation of a rolling moment, motors 2 and 3 were command to generate thrust, whereas motors 1 and 4 received no command input. (Figure 6.9). To generate a pitching moment, motors 1 and 2 were commanded to generate thrust, whereas motors 3 and 4 received no command input (Figure 6.10). Lastly, to generate a yawing moment, motors 1 and 3 were commanded to produce thrust while motors 2 and 4 were left idle (Figure 6.11). Numerous tests were run to verify these results and ensure that our plant dynamics were simulated properly.

With the results of the first class of test displaying the expected result, a second class of tests was performed. This class of tests implemented the full actuator block (Figure 6.12). To this end, pure thrust and moment commands were sent to the actuator block, and the output of the actuator block sent to the linear dynamic model of the

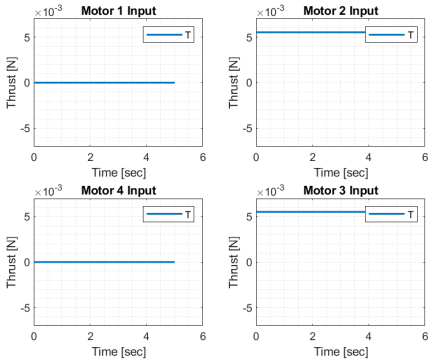


(a) Motor inputs.

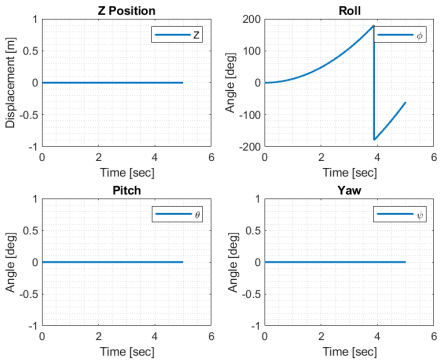


(b) States.

Figure 6.8: Model verification of the simulator's actuators on the linear quadcopter dynamics. All four motors generate equal thrust.

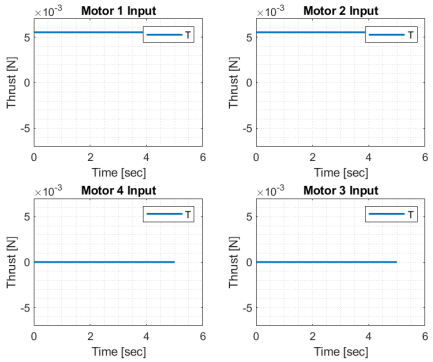


(a) Motor inputs.

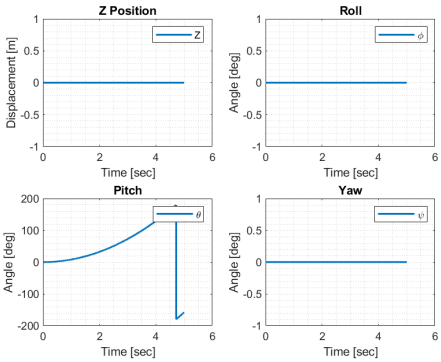


(b) States.

Figure 6.9: Model verification of the simulator's actuators on the linear quadcopter dynamics. Motors 2 and 3 produce increased thrust to generate a rolling moment.

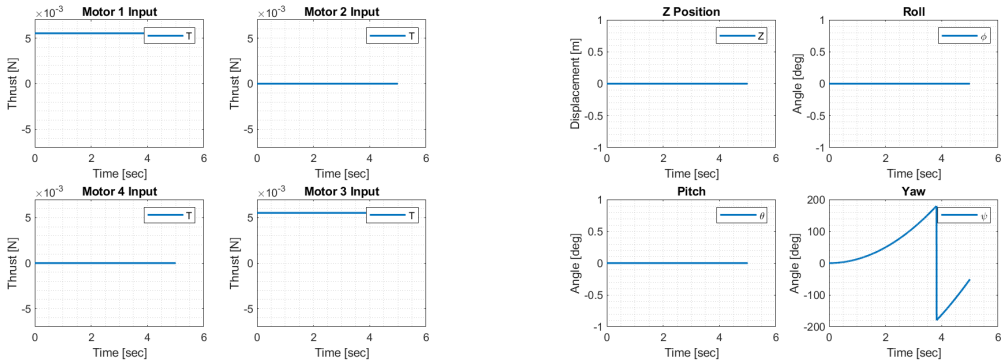


(a) Motor inputs.



(b) States.

Figure 6.10: Model verification of the simulator's actuators on the linear quadcopter dynamics. Motors 2 and 3 produce increased thrust to generate a rolling moment.



(a) Motor inputs.

(b) States.

Figure 6.11: Model verification of the simulator's actuators on the linear quadcopter dynamics. Motors 1 and 3 produce increased thrust to generate a yawing moment.

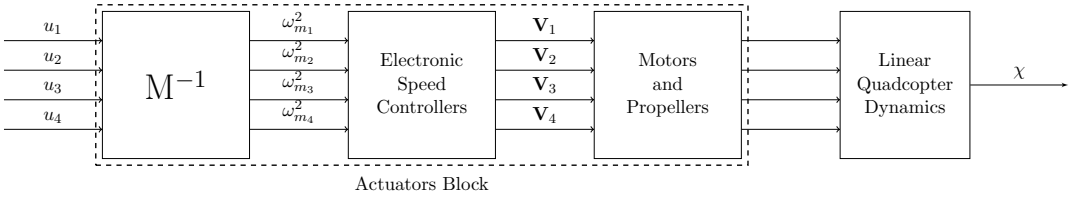


Figure 6.12: Simulated quadcopter linear dynamics validation using thrust and moments commands sent to the actuator block.

quadcopter. As the results from this class of tests were identical to those of the results where the thrust and moment commands were sent directly the linear model of the quadcopter dynamics, they will not be shown here.

6.1.3 Nonlinear Model Verification

Having verified the open-loop linear dynamics of the quadcopter, and verifying actuator, the open-loop nonlinear dynamics model were verified in the same manner. To this end, the nonlinear system was verified by sending direct thrust and moment inputs to the actuator block which were then sent to the nonlinear dynamics model (Figure 6.13). As before, a thrust of 5.8$[N]$ was sent to the actuator, while moments were set

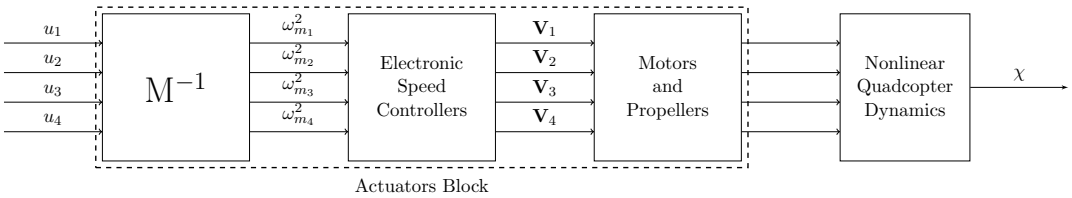
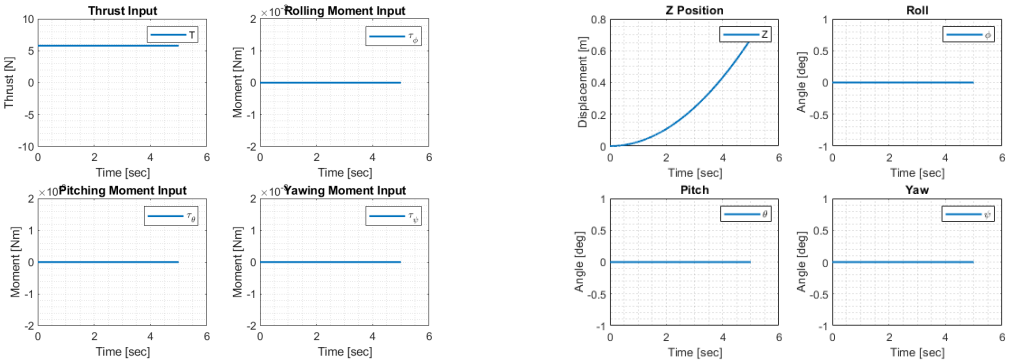


Figure 6.13: Simulated quadcopter nonlinear dynamics validation using thrust and moments commands sent to the actuator block.

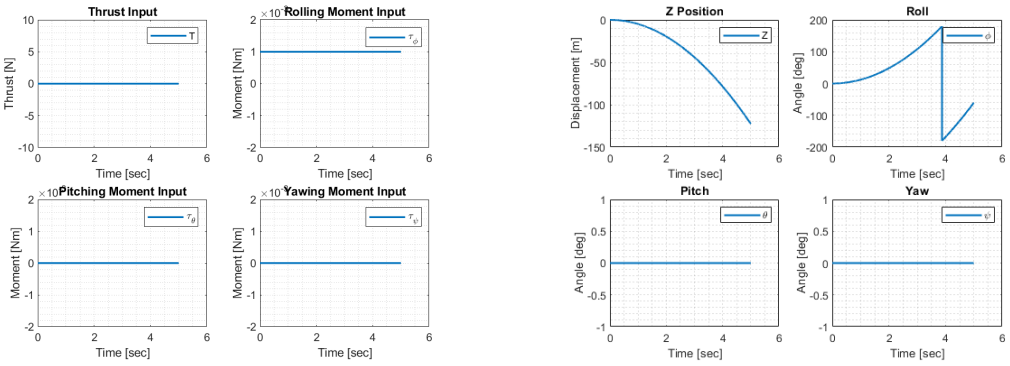
to zero (Figure 6.14). Next, tests of pure moment inputs were tested. To this end,



(a) Inputs. (b) States.

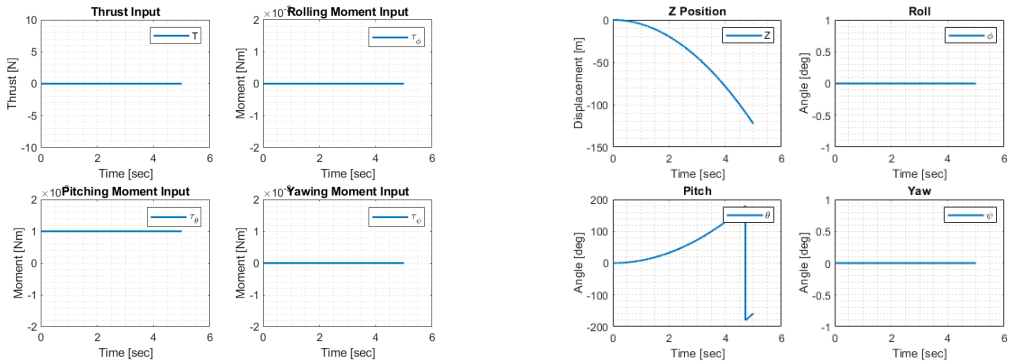
Figure 6.14: Model verification of the simulator's nonlinear quadcopter dynamics via a thrust step input.

as with the linear model verification tests, a moment step input of $0.001[Nm]$ around a given axis was sent to the actuator block, with all other inputs being zero. Tests were performed for pure rolling moment inputs (Figure 6.15), pure pitching moment inputs (Figure 6.16), and pure yawing moment inputs (Figure 6.15). It is worthwhile to note that unlike in the linear dynamics tests, where the model intrinsically includes the equilibrium trim, the nonlinear dynamics do not include this. Due to this, in all tests other than the pure thrust test, the quadcopter "falls" due to the gravity component of the model. Additionally, it is worth noting that as mentioned in the linear model verification section, that while the quadcopter begins at an altitude of zero, it is *not* on the ground, as there is no ground or environment implemented in the open-loop dynamics simulation.



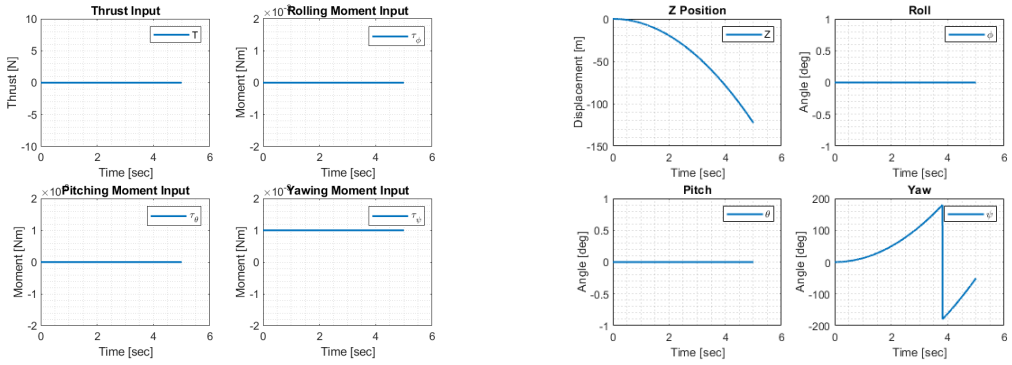
(a) Inputs. (b) States.

Figure 6.15: Model verification of the simulator's nonlinear quadcopter dynamics via a rolling moment step input.



(a) Inputs. (b) States.

Figure 6.16: Model verification of the simulator's nonlinear quadcopter dynamics via a rolling moment step input.



(a) Inputs. (b) States.

Figure 6.17: Model verification of the simulator's nonlinear quadcopter dynamics via a yawing moment step input.

6.2 Simulator Model Validation

While model verification is critical to see that the simulated vehicle behaves the way we expect it to, it is important to validate the model as well. Model validation was performed using an existing quadcopter with known mass properties and precise controller gains in a cascaded PID control system (Appendix K.3). In this instance, the existing quadcopter used was a PARROT Rolling Spider (Figure 6.18). The physical and mass properties of the Rolling Spider can be found in Table 6.1. Likewise, the controller gains used in the cascaded control system can be found in Table 6.2.

Test flights were implemented on the physical hardware and compared with the simulated results. In Figure 6.19, one such example is shown. In this example, an open-loop guidance law was implemented, commanding the quadcopter to hover at $0.7[m]$ for the duration of the flight, and then land after 35 seconds. Additionally, after 15 seconds, the quadcopter received a command to move 1 meter in the positive direction along the y-

Total Mass:	$0.0630[kg]$
Moment of Inertia:	$\begin{bmatrix} 0.00005829 & 0 & 0 \\ 0 & 0.00007169 & 0 \\ 0 & 0 & 0.0001 \end{bmatrix} [kg\ m^2]$
Arm Lengths:	$L_{x_1} = 44.1[mm]$ $L_{y_1} = 44.1[mm]$ $L_{x_2} = 44.1[mm]$ $L_{y_2} = 44.1[mm]$ $L_{x_3} = 44.1[mm]$ $L_{y_3} = 44.1[mm]$ $L_{x_4} = 44.1[mm]$ $L_{y_4} = 44.1[mm]$

Table 6.1: Physical properties of the Rolling Spider quadcopter.

Rolling Spider Cascaded PID Control System

X Controller:	K_p : 0.09 K_i : 0 K_d : 0.091	Y Controller:	K_p : 0.1 K_i : 0 K_d : 0.0818
Roll Controller:	K_p : 0.0034 K_i : 0 K_d : 0.00053414	Pitch Controller:	K_p : 0.0045 K_i : 0 K_d : 0.00070695
Yaw Controller:	K_p : 0.0004 K_i : 0 K_d : 0.00012	Height Controller:	K_p : 0.8862 K_i : 0 K_d : 0.25035

Table 6.2: Cascaded PID control system of the PARROT Rolling Spider quadcopter.

axis and remain there for 20 seconds. This test was then repeated in the simulator using the same guidance law and controllers, and the results compared. Because the internal sensors of the Rolling Spider are of unknown specifications, and the parameters of the estimators are unknown as well, the simulation is run assuming full state information. Needless to say, this introduces discrepancies into a comparison between the physical test and the simulated result, however at the low-speeds, short timescale and small displacements involved, we consider the error between the estimators and the true states to be small. Likewise, while real world flight conditions and environmental factors impact the flight, under those same assumptions, we consider their impact to be small

Figure 6.18: A PARROT Rolling Spider Quadcopter.

as well. After accounting for these factors and comparing the results (Figure 6.19), the simulation environment was considered to generate results within an acceptable range of fidelity, and the therefore was considered validated.

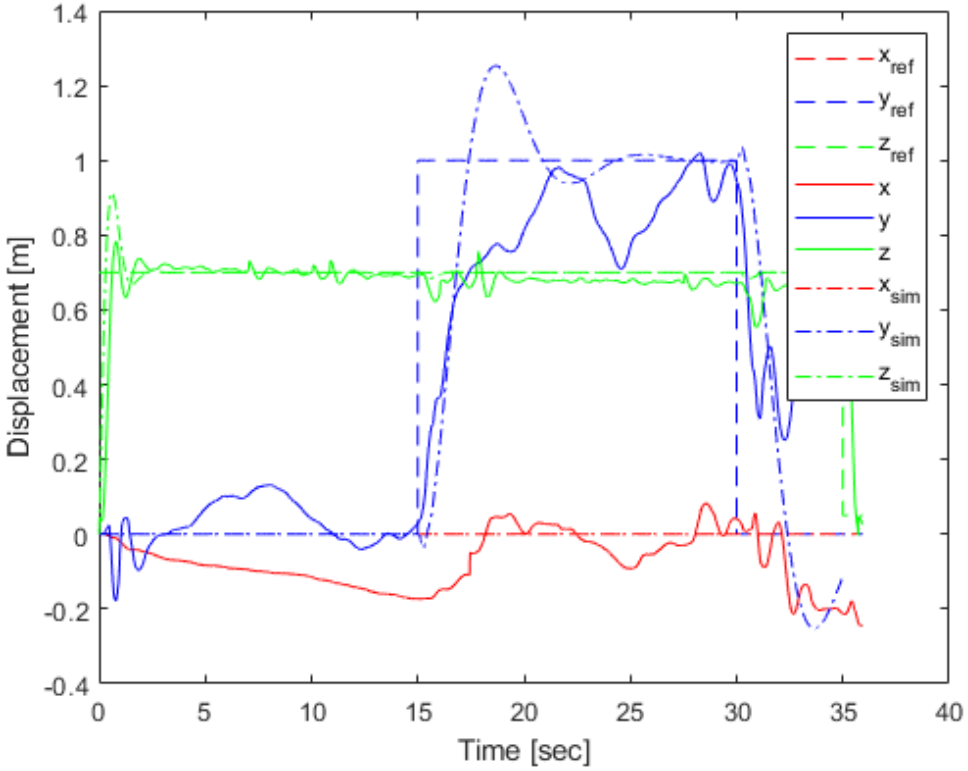


Figure 6.19: Model validation of the flight simulator.

Dynamics:	Linear
State Information:	Full State Information
Sensor Noise:	None
Motor Saturation:	Disabled
Environment:	Constant
Estimator Package:	Attitude Estimator: Complementary Filters Height Estimator: Kalman Filter Lateral Position Estimator: None
Guidance Package:	Open-loop Guidance
Controls Package:	Inner-loop PID Controllers
Simulation Time:	60 [Seconds]

Table 6.3: Representative Simulation 1

6.3 Results

6.3.1 Flight Simulator

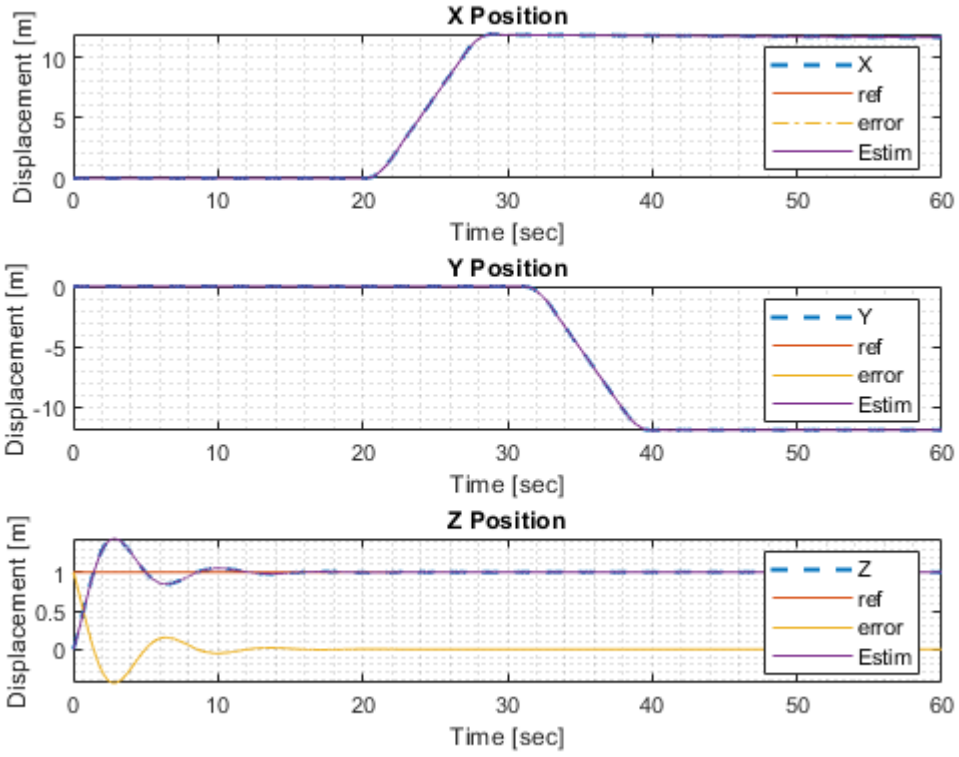
After validation of the simulation environment, testing of new GNC systems were initiated using the simulator workflow (Figure 2.3). While by no means comprehensive, the results of three representative examples are displayed below.

Simulation 1

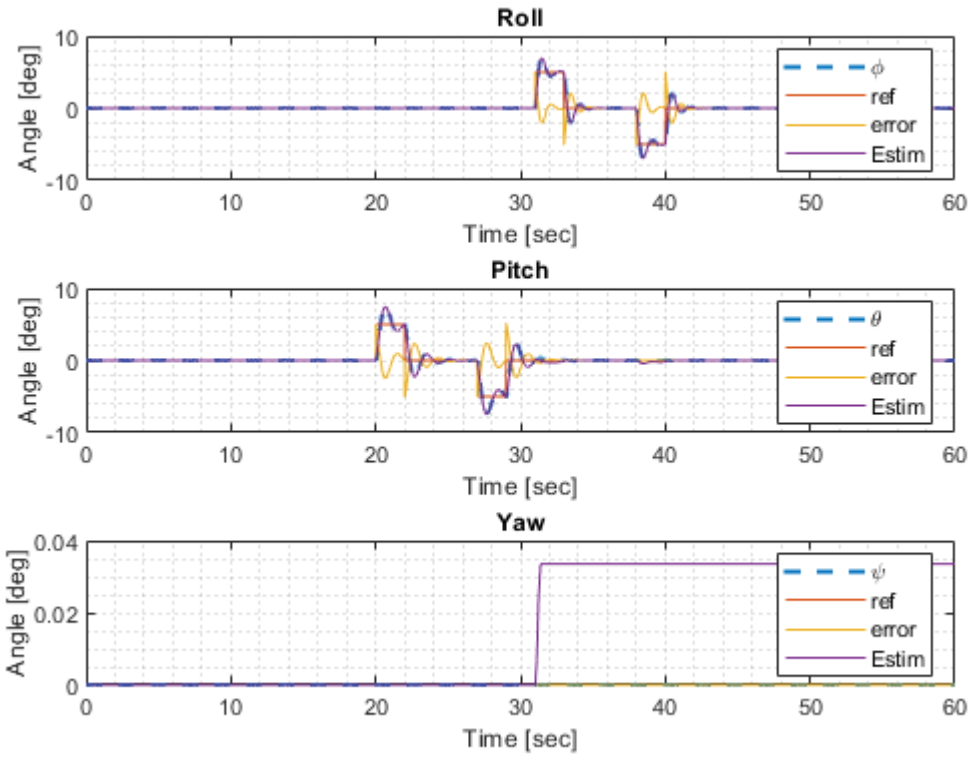
The simulation parameters for the first representative example can be found in Table 6.3. Plots of the resulting simulation can be found in Figure 6.20. Each plot shows the true state, the estimation of the state, the reference signal, and the error signal. Table 6.4 shows the parameters used in the flight controller. From the graphs, we can see that the the quadcopter responds quickly to the reference signals generated by the guidance law and pitches and rolls accordingly. Additionally, it maintains it's altitude well for the duration of the simulation. We can also see that, as expected, the changes to the pitch and roll angles generated x and y lateral velocities, which were then negated by pitching and rolling in the opposite directions for an equal amount of time.

Simulation 2

The simulation parameters for the second simulator example can be found in Table 6.5. The results of the simulation can be found in Figure 6.21. Each plot shows the true state, the estimation of the state, the reference signal, and the error signal. Table 6.6 shows the parameters used to determine the flight controller, where the the matrices Q_{LQ} and R_{LQ} were used to compute the gain matrix K for the inner-loop LQR controller using the matrices $\mathbf{A}_{inner}$ and $\mathbf{B}_{inner}$, describing the altitude and attitude dynamics, as

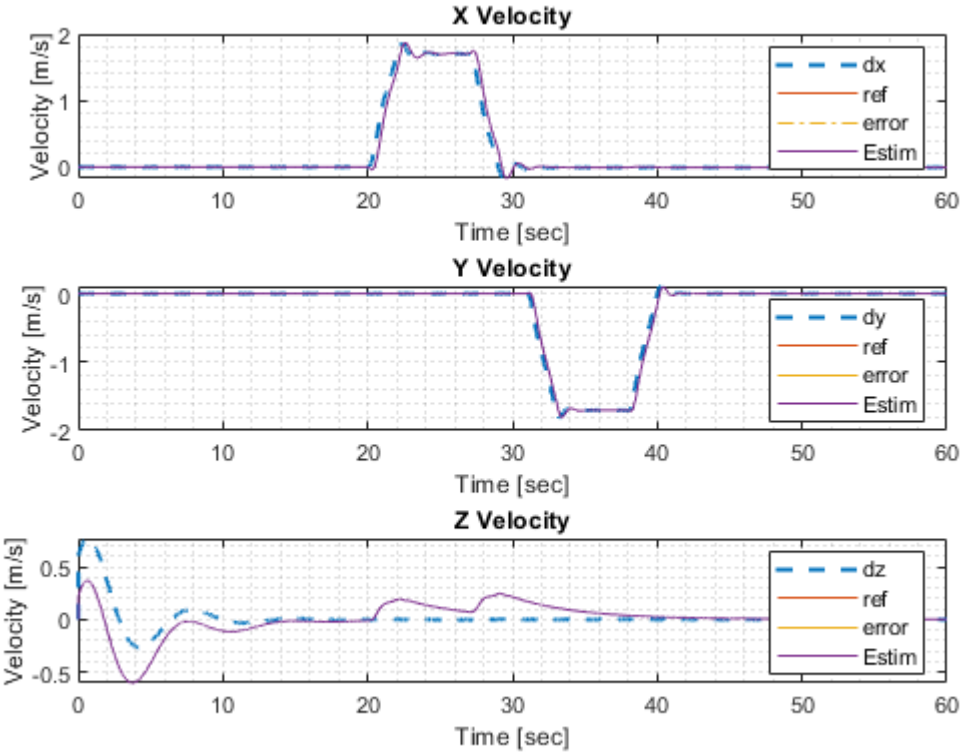


(a) Position Plot

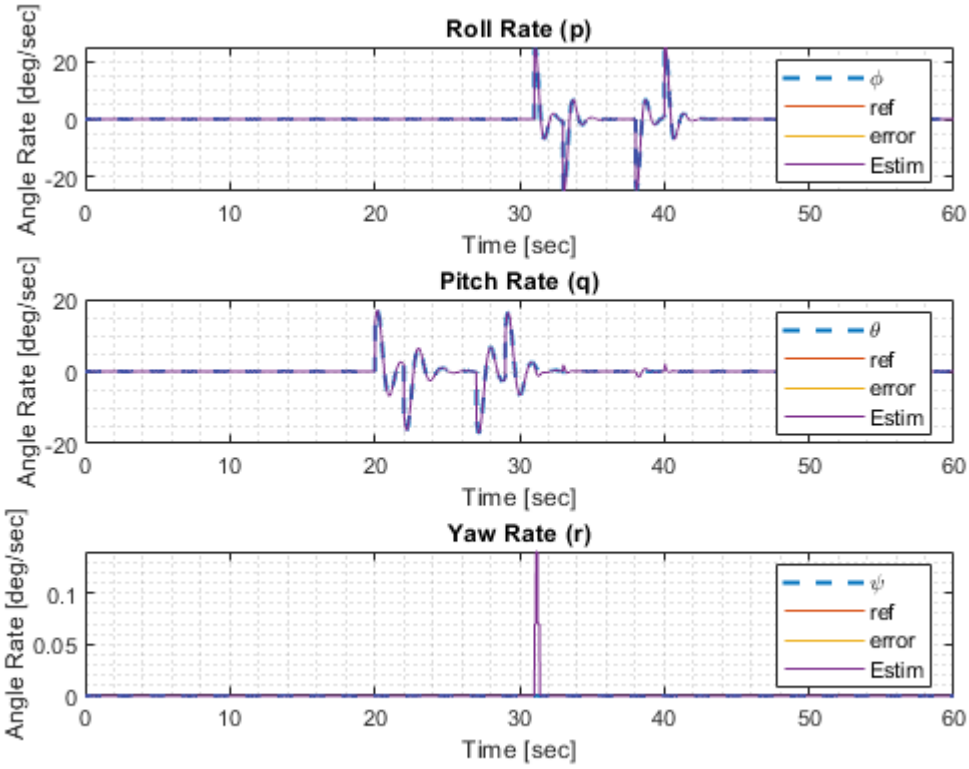


(b) Angle Plot

Figure 6.20: Results of simulation 1.



(c) Velocity Plot



(d) Angular Rate Plot

Figure 6.20: Results of simulation 1.

Inner-Loop Only PID Controller Parameters

Roll Controller:	$K_p:$ 0.065 $K_i:$ 0.005 $K_d:$ 0.01	Pitch Controller:	$K_p:$ 0.055 $K_i:$ 0.003 $K_d:$ 0.0085
Yaw Controller:	$K_p:$ 0.005 $K_i:$ 0 $K_d:$ 0.0015	Height Controller:	$K_p:$ 0.5 $K_i:$ 0 $K_d:$ 0.35

Table 6.4: Controller parameters used in representative simulation 1.

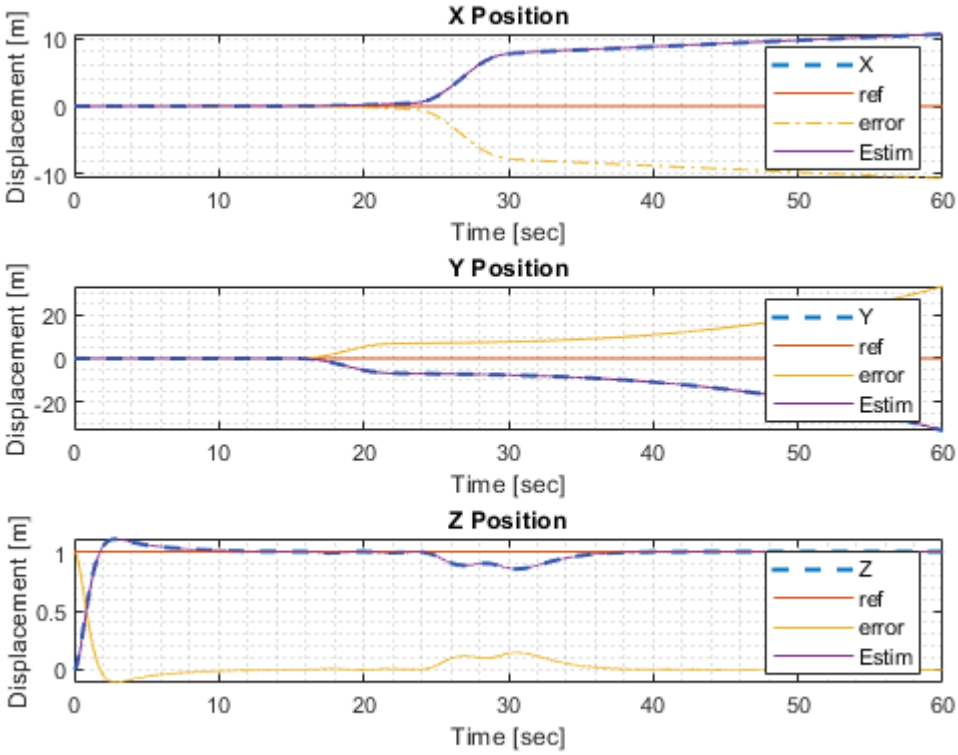
denoted below for the inner-loop only dynamics of the quadcopter,

$$\dot{\chi}_{inner} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{A}_{inner}} \underbrace{\begin{bmatrix} z \\ \phi \\ \theta \\ \psi \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}}_{\chi_{inner}} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_{xx}} & 0 & 0 \\ 0 & 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}}_{\mathbf{B}_{inner}} \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}. \tag{6.1}$$

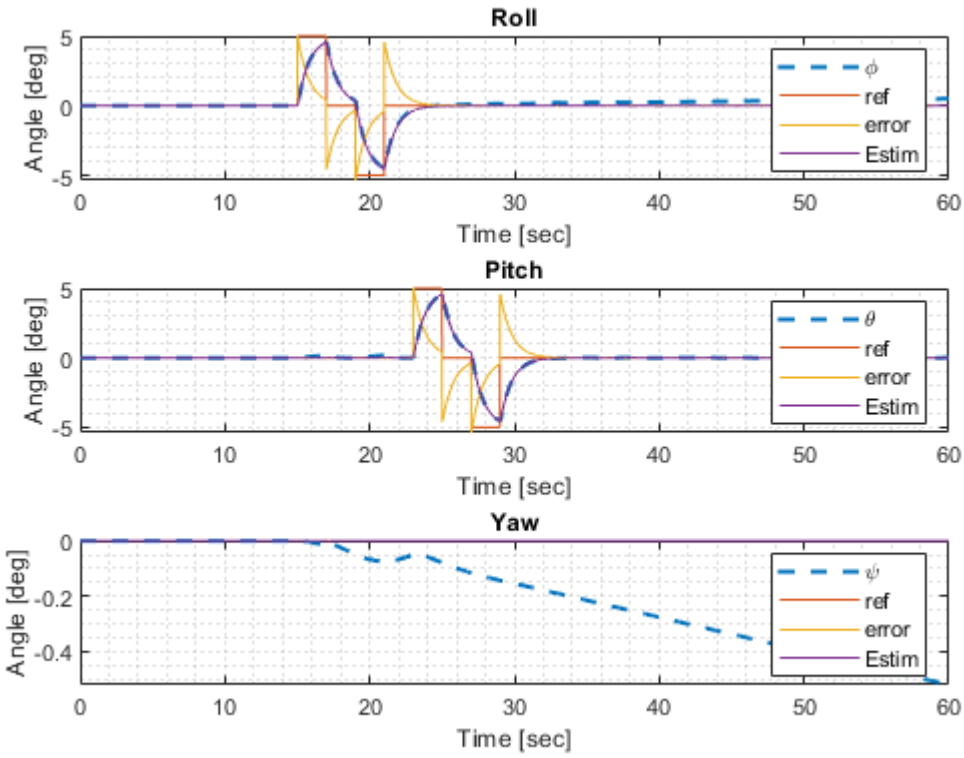
Table 6.7 shows the parameters of the estimators used in the example, where for the Kalman filters, the matrices Q_{KF} and R_{KF} were used to determine during simulation. From the graphs, we can see that the the quadcopter responds quickly to the reference signals generated by the guidance law and pitches and rolls accordingly, however not as rapidly as in the previous simulation. Additionally, it maintains it's altitude well for the duration of the simulation, although it noticeably drops during pitching and rolling motions before recovery. We can also see that, as expected, the changes to the pitch and roll angles generated x and y lateral velocities, which were then negated by pitching and rolling in the opposite directions for an equal amount of time. We can further see from this simulation that the estimators that while not perfect, perform admirably.

Simulation 3

The simulation parameters for the final representative example can be found in Table 6.8. The results of the simulation can be found in Figure 6.22. Each plot shows the true state, the estimation of the state, the reference signal, and the error signal. Table 6.9 shows the parameters used to determine the flight controller, where the the matrices

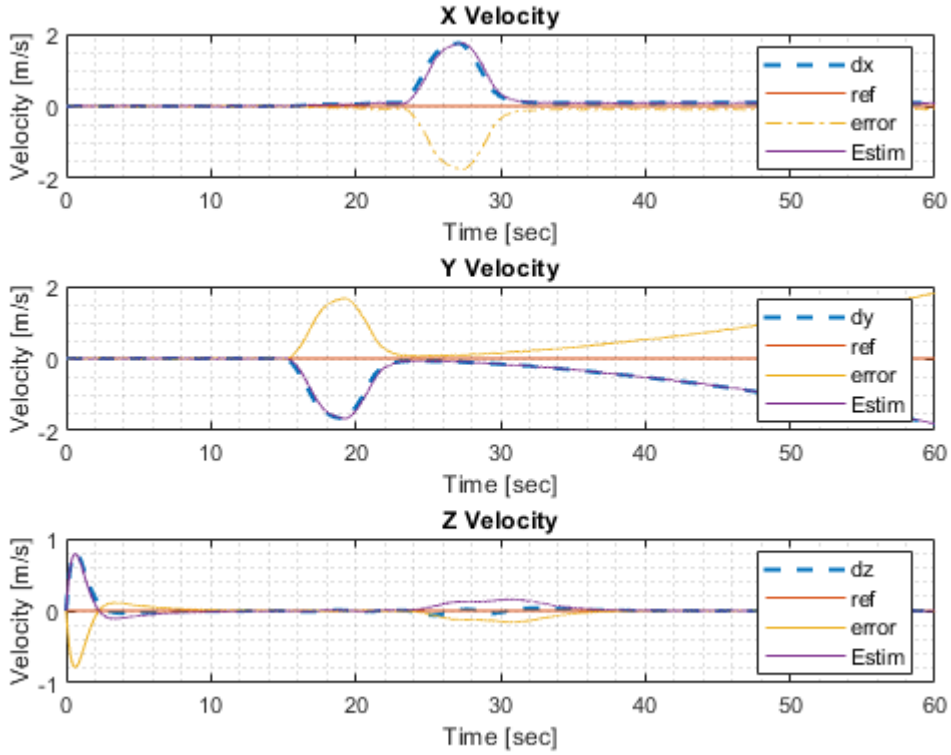


(a) Position Plot

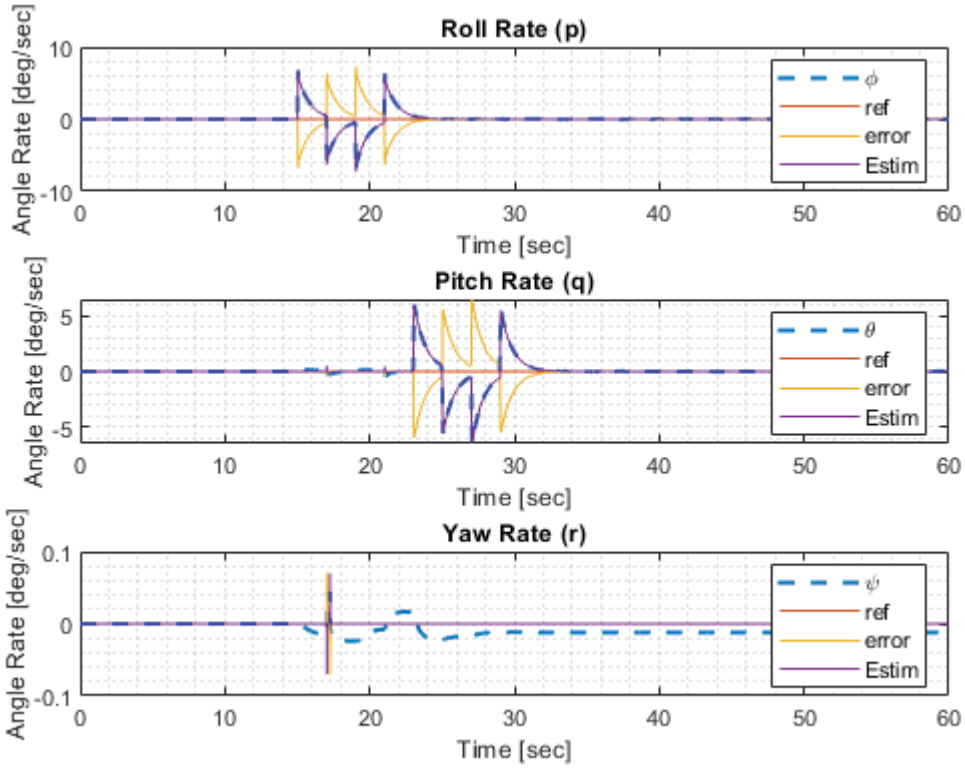


(b) Angle Plot

Figure 6.21: Results of simulation 2.



(c) Velocity Plot



(d) Angular Rate Plot

Figure 6.21: Results of simulation 2.

Dynamics:	Nonlinear
State Information:	Sensors and Estimators
Sensor Noise:	Dynamic Noise
Motor Saturation:	Disabled
Environment:	Constant
Estimator Package:	Attitude Estimator: Complementary Filters Height Estimator: Kalman Filter Lateral Position Estimator: None
Guidance Package:	Open-loop Guidance
Controls Package:	Inner-loop LQR Controller
Simulation Time:	60 [Seconds]

Table 6.5: Representative Simulation 2

Inner-Loop Only LQR Controller Parameters

$$Q_{LQ}: \begin{bmatrix} 0.3226 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.3080 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.3080 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3080 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.0161 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1935 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.1935 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1935 \end{bmatrix}$$

$$R_{LQ}: \begin{bmatrix} 0.1000 & 0 & 0 & 0 \\ 0 & 10.0000 & 0 & 0 \\ 0 & 0 & 10.0000 & 0 \\ 0 & 0 & 0 & 10.0000 \end{bmatrix}$$

Table 6.6: Controller parameters used in representative simulation 2.

Q_{LQ} and R_{LQ} were used to computer the gain matrix K for the LQR controller using the matrices $\mathbf{A}$ and $\mathbf{B}$ as developed in (4.42). Table 6.10 shows the parameters of the estimators used in the example, where for the Kalman filters, the matrices Q_{KF} and R_{KF} were used to determine the Kalman gain during simulation.To more easily display the waypoint guidance, an additional image depicting the trajectory in the XY-plane was added as well. From the graphs of this simulation, we can see that the the quadcopter responds quickly to the reference signals generated by the guidance law and accurately follows the desired waypoints. Additionally, it maintains it's altitude well for the duration of the simulation, although it noticeably drops during pitching and rolling motions before recovery. We can further see from this simulation that the estimators that while not perfect, the estimators perform admirably. One immediately noticeable aspect of the trajectory plot is that the quadcopter is highly active around

Estimator Parameters

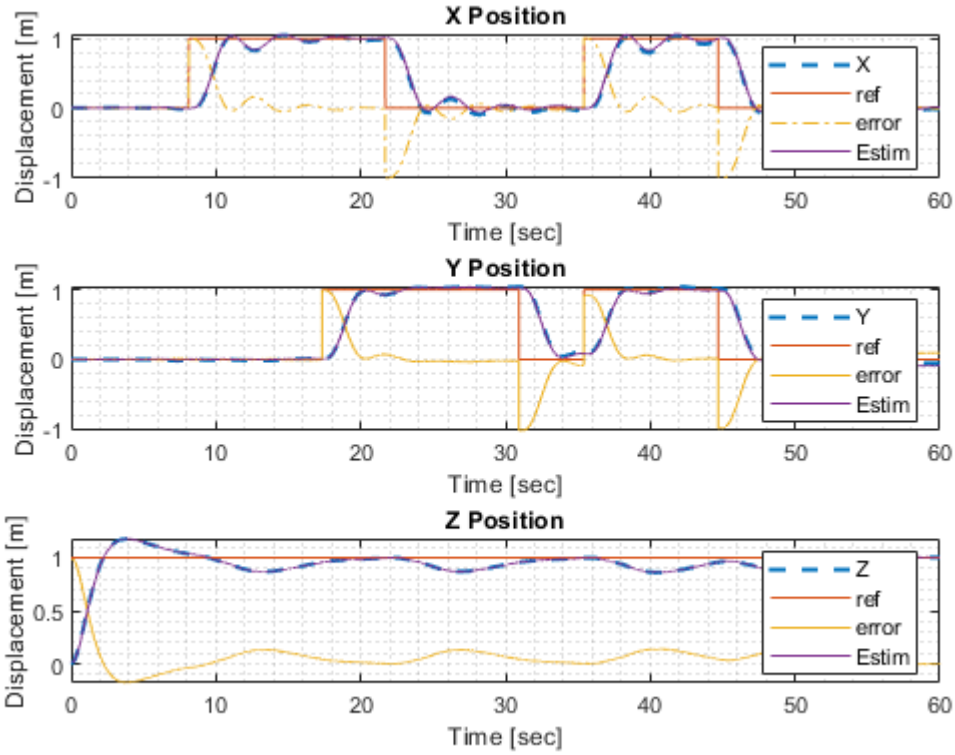
Roll Complementary	$\omega_c = 2.43$
Pitch Complementary	$\omega_c = 2.43$
Yaw Complementary	$\omega_c = 1.2$
Height Kalman	Q_{KF}: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ R_{KF}: $\begin{bmatrix} 1 & 0 \\ 0 & 0.0025 \end{bmatrix}$

Table 6.7: Estimator parameters used in representative simulation 2.

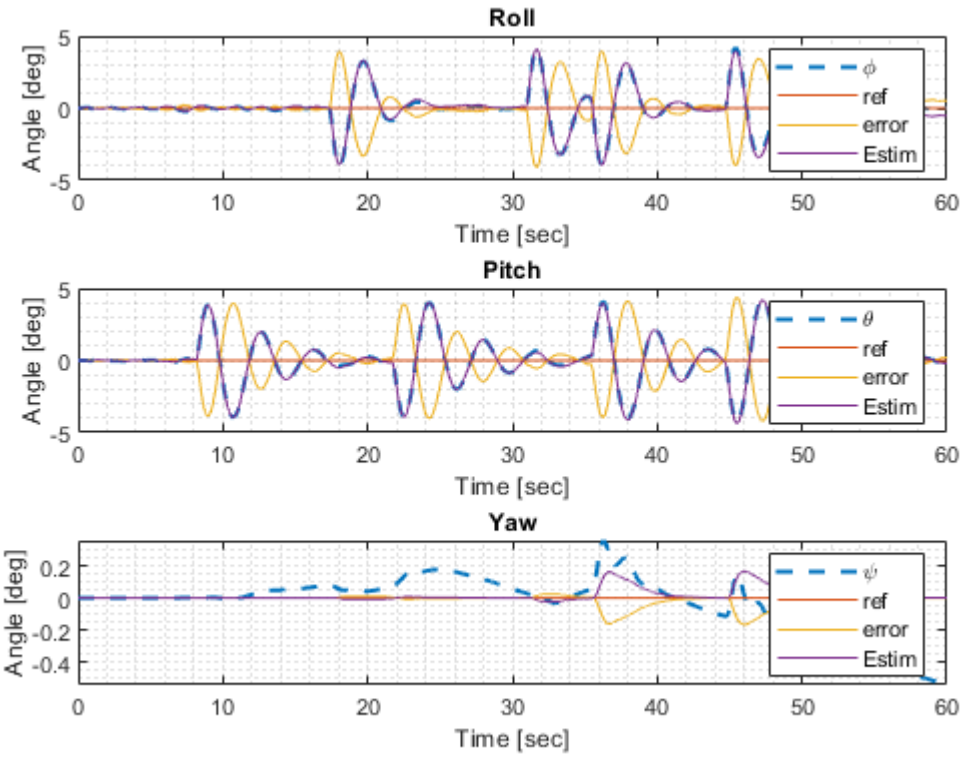
Dynamics:	Nonlinear
State Information:	Sensors and Estimators
Sensor Noise:	Dynamic Noise
Motor Saturation:	Enabled
Environment:	Constant
Estimator Package:	Attitude Estimator: Complementary Filters Height Estimator: Kalman Filter Lateral Position Estimator: Kalman Filter
Guidance Package:	Waypoint Guidance
Controls Package:	Full State LQR Controller
Simulation Time:	60 [Seconds]

Table 6.8: Representative Simulation 3

the "corners" of the trajectory. This is behaviour to be expected by the guidance law, as it requires the quadcopter to remain in the "neighborhood" of each waypoint for a given time before it is allowed to advance to the next waypoint. Furthermore, this is an aspect of how the controller and guidance law interact. As the controller is a linear controller controlling a highly nonlinear system, in the sections where the guidance law requires a trajectory which does not require any coupled motions, the system behaves nicely, however in the sections of the trajectory which require coupled motion (such as pitching and rolling simultaneously) the control is negatively impacted.

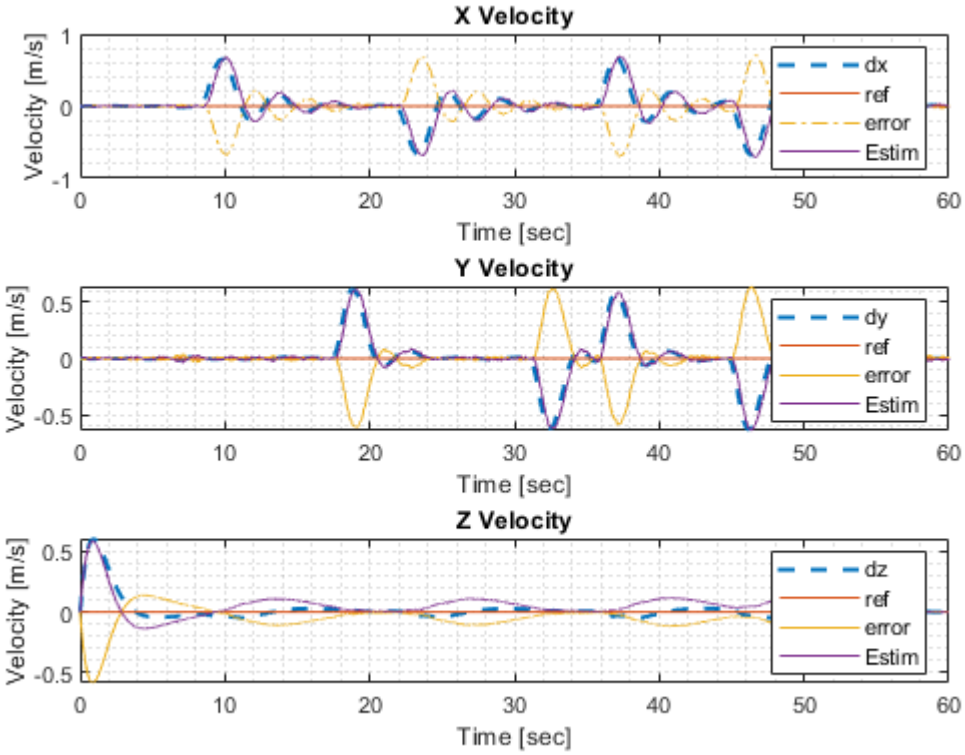


(a) Position Plot

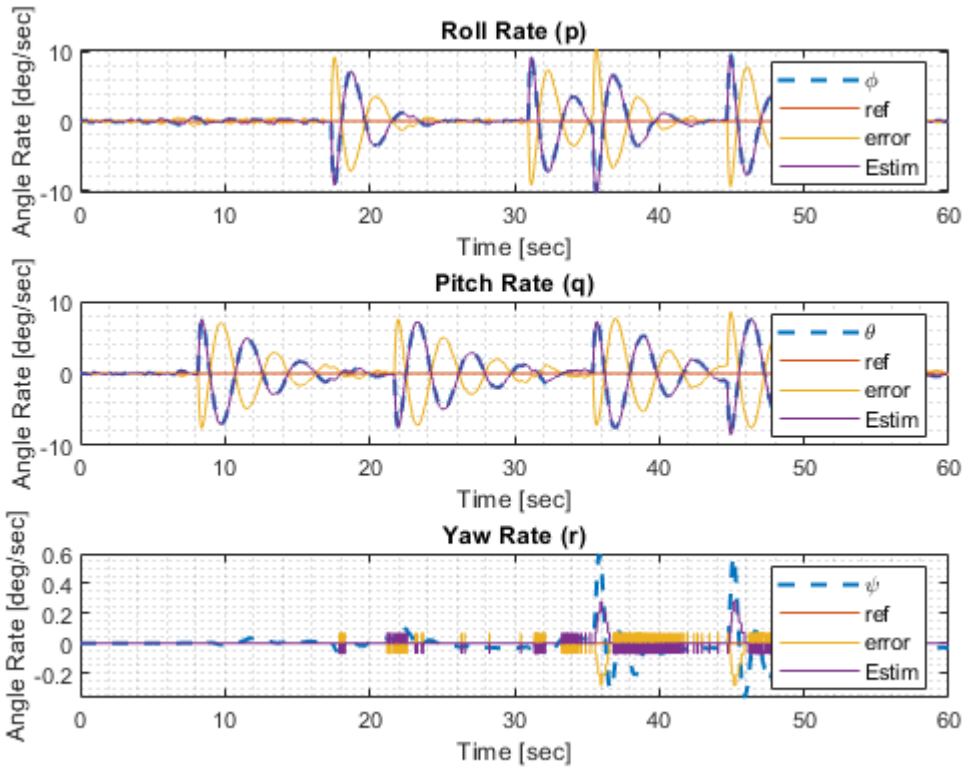


(b) Angle Plot

Figure 6.22: Results of simulation 3.

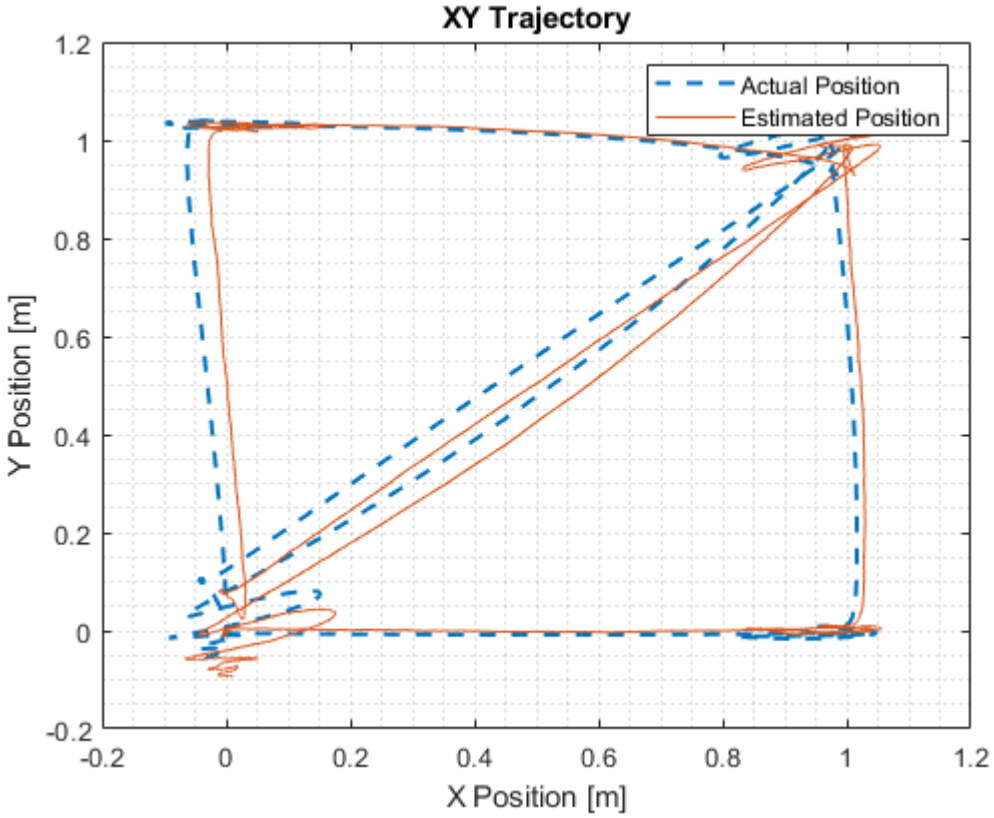


(c) Velocity Plot



(d) Angular Rate Plot

Figure 6.22: Results of simulation 3.



(e) XY-Plane Trajectory5 Plot

Figure 6.22: Results of simulation 3.

Full State LQR Controller Parameters

Q_{LQ}:	$\begin{bmatrix} 0.0035 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.0035 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.7699 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.0507 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.0507 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.0507 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0018 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0018 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0018 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0177 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0177 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0177 \end{bmatrix}$
R_{LQ}:	$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 200 & 0 & 0 \\ 0 & 0 & 200 & 0 \\ 0 & 0 & 0 & 200 \end{bmatrix}$

Table 6.9: Controller parameters used in representative simulation 3.

Estimator Parameters

Roll Complementary	$\omega_c = 2.43$
Pitch Complementary	$\omega_c = 2.43$
Yaw Complementary	$\omega_c = 1.2$
Height Kalman	Q_{KF}: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ R_{KF}: $\begin{bmatrix} 1 & 0 \\ 0 & 0.0025 \end{bmatrix}$
Lateral Kalman	Q_{KF}: $\begin{bmatrix} 0.00025dt^4 & 0 & 0.0005dt^3 & 0 \\ 0 & 0.00025dt^4 & 0 & 0.0005dt^3 \\ 0.0005dt^3 & 0 & 0.0001dt^2 & 0 \\ 0 & 0.0005dt^3 & 0 & 0.0001dt^2 \end{bmatrix}$ R_{KF}: $\begin{bmatrix} 0.00001 & 0 \\ 0 & 0.00001 \end{bmatrix}$

Table 6.10: Estimator parameters used in representative simulation 3.

6.3.2 Hardware Results

After the GNC systems were tested in the simulation environment, and acceptable parameters for the GNC systems were determined, hardware testing was able to be initiated.

Constrained Flight

During constrained flight testing, the quadcopter was attached to the gimbal (Figure 3.11) to test the attitude estimators and controllers. A four step process was initiated for constrained flight testing.

1. The quadcopter was powered on while on a flat surface.
2. After a 5 second wait, the LED blinked 3 times to indicate sensor calibration was complete. This was to ensure that the sensors are properly calibrated without any offsets due to the gimbal itself.
3. The quadcopter was placed into the gimbal, and adjusted to balance properly.
4. Using the BLE Live Logger, a start command was sent to the quadcopter to begin the test.

To demonstrate the controllers and estimators working in tandem in constrained flight, a gimbal test (Figure 6.23) was performed. As the GUI displays the controller output, to clearly demonstrate how the they work in response to a given reference signal the following steps were taken.

1. Open-loop guidance was chosen to require an attitude of $0°$.
2. The quadcopter was given a 5 seconds to maintain the set altitude.
3. *Carefully*, the quadcopter was manually moved away from a zero attitude and held at an angle to generate nonzero control signals for a few seconds.
4. The quadcopter was released and allowed to return to the it's zero attitude state.
5. Steps 3 and 4 were repeated multiple times and at larger angles and in various configurations.

From the results shown in the GUI (Figure 6.23) the correlation between the controller outputs and the attitude angle can be clearly seen.

Free Flight

After the completion of the constrained flight test, the quadcopter may be considered safe to fly in a free flight test. The quadcopter is designed to be flown in a laboratory environment. This entails flying the quadcopter inside a netted area free of obstacles for

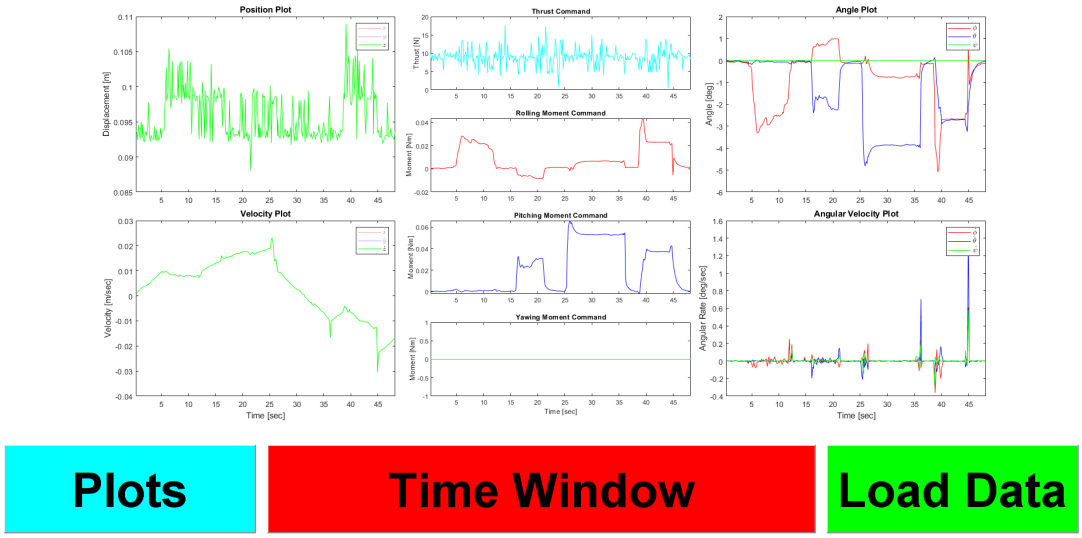


Figure 6.23: Results of the gimbal test.

safety reasons. During flight, the quadcopter should be the only thing inside the netted area with any observers should remain safely outside of the netted area. Additionally, during flight, the quadcopter operator should be ready to send the stop command via the BLE Live Logger to perform an emergency stop in the event of unsafe flight performance. Unfortunately, due to faulty electronic speed controllers, sustained free flight could not be achieved. Momentary hover was achieved, however once real world disturbances entered the system along the diagonal axis, the faulty ESC introduced a diagonal oscillation into the quadcopters motions, causing a shift in attitude exceeding 15° to trigger the recovery control system, quickly followed by the attitude exceeding 20°, triggering the emergency shutdown protocol. The recovery control system and emergency shutdown protocols were initiated successfully, and the quadcopter remained undamaged.

(a) Start flight.

(b) Hovering flight.

(c) First Disturbance and tilt.

(d) Recovery subsystem enabled.

(e) Overshoot zero attitude.

(f) Emergency shutdown initiated.

Figure 6.24: Test flight of the quadcopter.

(g) Quadcopter falls to the ground.

(h) Complete stop and end of test flight.

Figure 6.24: Test flight of the quadcopter.

Chapter 7

Conclusions and Future Work

In this chapter, we will summarize the work done in this thesis, and discuss potential future developments for the research platform.

7.1 Conclusion

The goal of this thesis was to develop a low-cost end-to-end platform for the development of guidance, navigation, and control systems. To this end, four primary components were developed.

Quadcopter

A 3D-printable quadcopter frame was designed and fabricated using an Ultimaker S3 Extended and Ultimaker S5 3D printer. The frame was designed to accommodate inexpensive, off-the-shelf sensors, a battery pack, electronic speed controllers, motors, and an Arduino microcomputer used as a a flight control system. The quadcopter was designed to be easy to assemble, and resistant to physical impacts. Testing of the quadcopter showed it to be highly durable and resistant to physical damage, and easy to use.

Testing Equipment

Physical testing equipment was developed for the quadcopter to allow for thrust profile determination of the motors, and to allow for constrained flight for attitude controller testing. The testing equipment was built using a combination of 3D components and inexpensive and commonly available parts.

Simulator Environment

A simulator environment was developed in `Simulink` to be highly modular and user-friendly. Additionally, a workflow was developed for implementing new systems from the conceptual stage through final simulations before applying said systems on hardware.

The simulator environment was verified through a series of input tests before being validated by simulating an existing quadcopter drone compared to its known real-world flight data.

Software

A suite of software was developed for the aerial platform. This software included software to determine the thrust profile of the motors, the actual flight code that ran the quadcopter, and BLE logging software. The software is fully open source, highly modular, simple to modify, and user-friendly. A minimal demo of the flight code running live on the quadcopter can be implemented by modifying as few as 5 lines of code.

7.2 Future Development

While the platform developed is capable, there are numerous avenues for improvement. Some examples of this are listed below.

The development of an open source simulation environment. One of the major advantages to the flight code is that it is fully open sourced. However, the simulator environment still relies on a proprietary program to run (MATLAB and Simulink). Open sourcing the simulator itself increases the accessibility of the platform as a whole.

The improvement the flight simulator fidelity. While the flight simulator is quite useful and effective at gauging the efficacy of various system, it still has room for improvement. Improving the fidelity of the model includes both improvements to the dynamics model of the quadcopter itself, as well as the internal actuator dynamics. The system model relies on numerous assumptions and simplifications, reducing the reliance on such can allow for more accurate simulation and allow the simulator to even more closely match the real world results.

The design of a testing equipment for height control and estimation. The design of a height test bench would allow for constrained flight along all four degrees of the inner-loop of the quadcopter. This would allow for an addition method of testing prior to free flight, and further reduce the risk of damage to the quadcopter or anyone in the lab when testing on hardware.

The refinement and redesign of the quadcopter frame. While the quadcopter as designed fulfills the design criteria, it has room for improvements. Refining or even redesigning the quadcopter frame can reduce the weight, improve structural integrity, and/or reduce the print time required to produce the frame. Additionally, new designs could allow for greater modularity of the sensors.

The design of a custom "all-in-one" flight controller board. Design of a custom "all-in-one" flight controller board which includes the microcomputer and associated sensors would further reduce required assembly by the user, and allow better placement of the sensors. Additionally, when combined with the refinement of the quadcopter design, this could lead to a drastic reduction in the size and weight of the quadcopter, and improve the performance of the platform.

A refinement of the communications protocol. Improving the communications protocol would allow for a higher throughput of data for logging purposes. This would allow the transfer of more data at a higher frequency providing additional insight into the flight of the quadcopter. This could be achieved for example by using a secondary microcomputer slaved to the flight controller board which can manage the communications without interrupting the flight code to transmit data.

While this list is not exhaustive, it provides an excellent starting points for potential improvements to the platform as a whole.

Appendix A

Quadcopter Frame Engineering Drawing

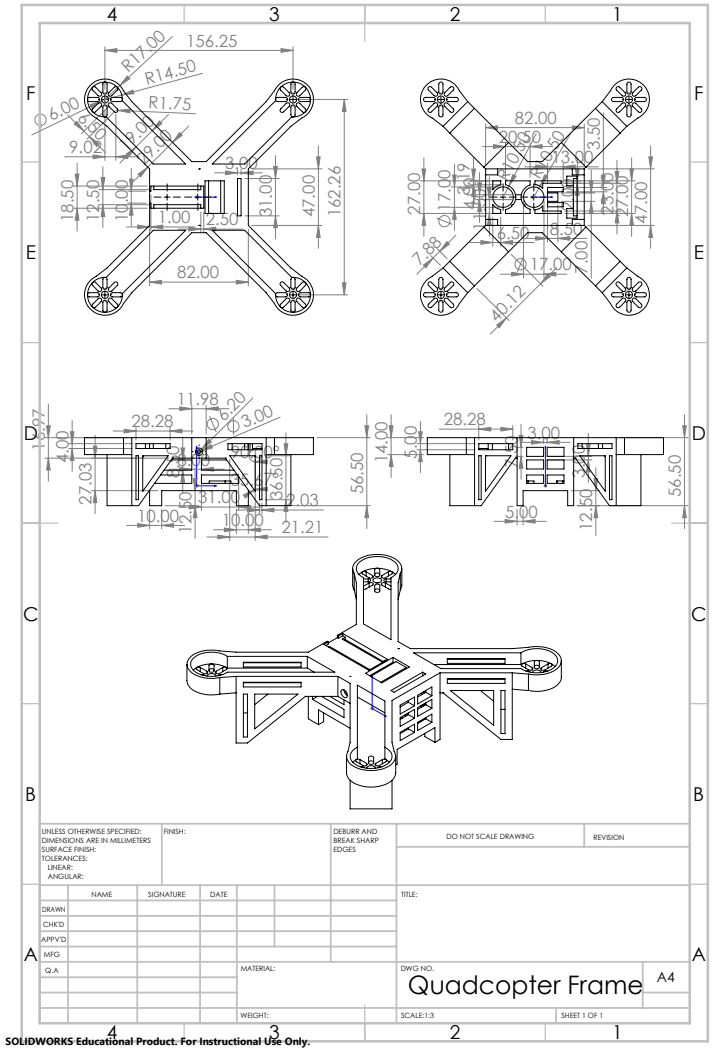


Figure A.1: Quadcopter Frame Engineering Drawing

Appendix B

Ultrasonic Sensor Code

The code responsible for the ultrasonic sensor can be found in `Ultra.hpp`.

```
#include <Arduino.h>
#include "Definitions.hpp"

using namespace std;

// Define Pins used for Ultrasonic Sensor
#define TRIG_PIN 4
#define ECHO_PIN 2
#define ULTRA_TIMEOUT 25000 // 25ms timout on pulse, roughty 9 meters
#define MINDIST 2
#define MAXDIST 4000
#define ALTITUDE_OUT_OF_BOUNDS -1

int handle_end_counter = 0;
int handle_start_counter = 0;
int emit_counter = 0;
int trampoline_counter = 0;

float range_safe(float altitude) {
    if (altitude < MINDIST || altitude > MAXDIST) return ALTITUDE_OUT_OF_BOUNDS;
    return altitude;
}
float echo_to_distance_mm(unsigned long ultra_duration) {
    return ((ultra_duration / 2) * 0.340); // TODO: add more precision for Mach
}

class Ultrasonic {
    private:
    volatile bool new_data;
    volatile bool is_active, in_pre_emit_phase;
    volatile unsigned long start_time;
    volatile float distance;

    public:
    Ultrasonic();
    void setup();
    void emit();
    void _handle_start();
    void _handle_end();
    void poll(SensorReadings &sensor_data, bool &logged);
};
```

Snippet 12: Ultrasonic sensors code: Part 1.

```cpp
Ultrasonic::Ultrasonic() {
    new_data = false;
    is_active = false;
    in_pre_emit_phase = false;
}

Ultrasonic* _ultrasonic_object;

void _ultrasonic_handler_trampoline() {
    trampoline_counter++;
    if (digitalRead(ECHO_PIN) == HIGH)
        _ultrasonic_object->_handle_start();
    else
        _ultrasonic_object->_handle_end();
}

void Ultrasonic::setup() {

    pinMode(TRIG_PIN, OUTPUT);
    pinMode(ECHO_PIN, INPUT);
    _ultrasonic_object = this;
    delayMicroseconds(20);
    attachInterrupt(digitalPinToInterrupt(ECHO_PIN), _ultrasonic_handler_trampoline,
                    CHANGE);
}

void Ultrasonic::emit() {
    emit_counter++;
    if (in_pre_emit_phase && micros() - start_time < ULTRA_TIMEOUT){
        return;
    }
    if (!is_active || micros() - start_time > ULTRA_TIMEOUT) {
        in_pre_emit_phase = true;
        is_active = true;
        digitalWrite(TRIG_PIN, LOW);
        delayMicroseconds(5);
        digitalWrite(TRIG_PIN, HIGH);
        delayMicroseconds(10);
        digitalWrite(TRIG_PIN, LOW);
    }
}

void Ultrasonic::_handle_start() {
    start_time = micros();
    in_pre_emit_phase = false;
}

void Ultrasonic::_handle_end() {
    unsigned long end_time = micros();
    unsigned long duration = end_time - start_time;
    distance = range_safe(echo_to_distance_mm(duration));
    new_data = (distance != ALTITUDE_OUT_OF_BOUNDS);
    is_active = false;
}

void Ultrasonic::poll(SensorReadings &sensor_data, bool &logged) {
    if (new_data) {
        new_data = false;
        sensor_data.ultrasonic_alt = distance / 1000;
        // Serial.println(sensor_data.ultrasonic_alt);
        logged = true;

    }
}
```

Snippet 13: Ultrasonic sensors code: Part 2.

Appendix C

Thrust Measurement Procedure

To use the thrust measurement test bench the following procedure should be followed:

1. Attach a chosen motor/propeller pair to the Individual Motor Thrust Test Bench.
2. Attach the motor to the ESC, and the ESC to the Arduino.
3. Connect the Arduino to a computer via USB.
4. Run the Thrust Measurement Software (Chapter 3.4.1).
5. Set motor output to maximum.
6. Connect the ESC to an appropriate power supply.
7. Wait for 2 beeps, then set motor output to minimum to calibrate the ESC.
8. Perform a manual test.
   (a) Starting at 0% increase power by 5%.
   (b) Record measurements in an spreadsheet.
   (c) Repeat until measurements are recorded up to 100%.
9. Repeat the manual test process $4-5$ times for each motor-propeller-ESC unit.
10. Attach all four ESCs and motors to the quadcopter frame.
11. Attach the Quadcopter Thrust Test Bench securely atop the Individual Motor Thrust Test Bench.
12. Attach the quadcopter securely to the Quadcopter Thrust Test Bench.
13. Perform a manual test on all 4 motors to confirm results taken from Step 6.
14. Determine thrust profiles for each motor-propeller-ESC unit.

Appendix D

Motor Command Code

To integrate measured motor thrust profiles into the flight code, the motor thrust profiles must be determined as per to process described in Appendix C. Once a thrust profile has been developed the following steps should be taken:

1. Convert the thrust profile into a function which determines desired ESC percentage as a function of the desired thrust. This is our Motor Thrust-ESC equation.
2. In the flight code directory, open the file `Motors.hpp` (Snippet 14).
3. For each motor, numbered as per Figure 3.3, add the Motor Thrust-ESC equation to the appropriate function.
    - `grf_to_pulse1` to Motor 1
    - `grf_to_pulse2` to Motor 2
    - `grf_to_pulse3` to Motor 3
    - `grf_to_pulse4` to Motor 4
4. Take the maximum value of the weakest motor and set `MINIMUM_MAX_MOTOR_THRUST` to that value in kgf. This sets the maximum thrust producible by any motor command to be no greater than the maximum thrust of the weakest motor.
5. Set `IDLE_PCT` to the lowest ESC percentage at which all motors are in motion.

```
#define IDLE_PCT 0.15 // [% of ESC ]
#define MINIMUM_MAX_MOTOR_THRUST .23 // [kgf]

int grf_to_pulse1(float x) {
    float gf = x * 1000; // kgf to grf (accel is measured in g's)
    if (gf > MINIMUM_MAX_MOTOR_THRUST*1000) gf = MINIMUM_MAX_MOTOR_THRUST*1000;
    float pct;

    pct = (gf+0.4455)/398.55; // 0% to 60% Range

    // Saturate Motor Commands to the Range of 10% - 100%
    if (pct > 1 || pct < IDLE_PCT){
        pct = (pct > 1) + (pct < IDLE_PCT)*IDLE_PCT;
    }

    int pulse = round(((MAX_PULSE_LENGTH - MIN_PULSE_LENGTH) * pct) + MIN_PULSE_LENGTH);
    return pulse;
}

int grf_to_pulse2(float x) {
    float gf = x * 1000; // kgf to grf (accel is measured in g's)
    if (gf > MINIMUM_MAX_MOTOR_THRUST*1000) gf = MINIMUM_MAX_MOTOR_THRUST*1000;
    float pct;

    pct = (gf-1.6841)/396.23; // 0% to 60% Range

    // Saturate Motor Commands to the Range of 10% - 100%
    if (pct > 1 || pct < IDLE_PCT){
        pct = (pct > 1) + (pct < IDLE_PCT)*IDLE_PCT;
    }

    int pulse = round(((MAX_PULSE_LENGTH - MIN_PULSE_LENGTH) * pct) + MIN_PULSE_LENGTH);
    return pulse;
}

int grf_to_pulse3(float x) {
    float gf = x * 1000; // kgf to grf (accel is measured in g's)
    if (gf > MINIMUM_MAX_MOTOR_THRUST*1000) gf = MINIMUM_MAX_MOTOR_THRUST*1000;
    float pct;

    pct = (gf-0.7114)/410.05; // 0% to 60% Range

    // Saturate Motor Commands to the Range of 10% - 100%
    if (pct > 1 || pct < IDLE_PCT){
        pct = (pct > 1) + (pct < IDLE_PCT)*IDLE_PCT;
    }

    int pulse = round(((MAX_PULSE_LENGTH - MIN_PULSE_LENGTH) * pct) + MIN_PULSE_LENGTH);
    return pulse;
}

int grf_to_pulse4(float x) {
    float gf = x * 1000; // kgf to grf (accel is measured in g's)
    if (gf > MINIMUM_MAX_MOTOR_THRUST*1000) gf = MINIMUM_MAX_MOTOR_THRUST*1000;
    float pct;

    pct = (gf+3.1727)/432.18; // 0% to 60% Range

    // Saturate Motor Commands to the Range of 10% - 100%
    if (pct > 1 || pct < IDLE_PCT){
        pct = (pct > 1) + (pct < IDLE_PCT)*IDLE_PCT;
    }

    int pulse = round(((MAX_PULSE_LENGTH - MIN_PULSE_LENGTH) * pct) + MIN_PULSE_LENGTH);
    return pulse;
}
```

Snippet 14: Motor thrust to ESC command functions for the flight code.

Appendix E

Quadcopter Wiring Diagram

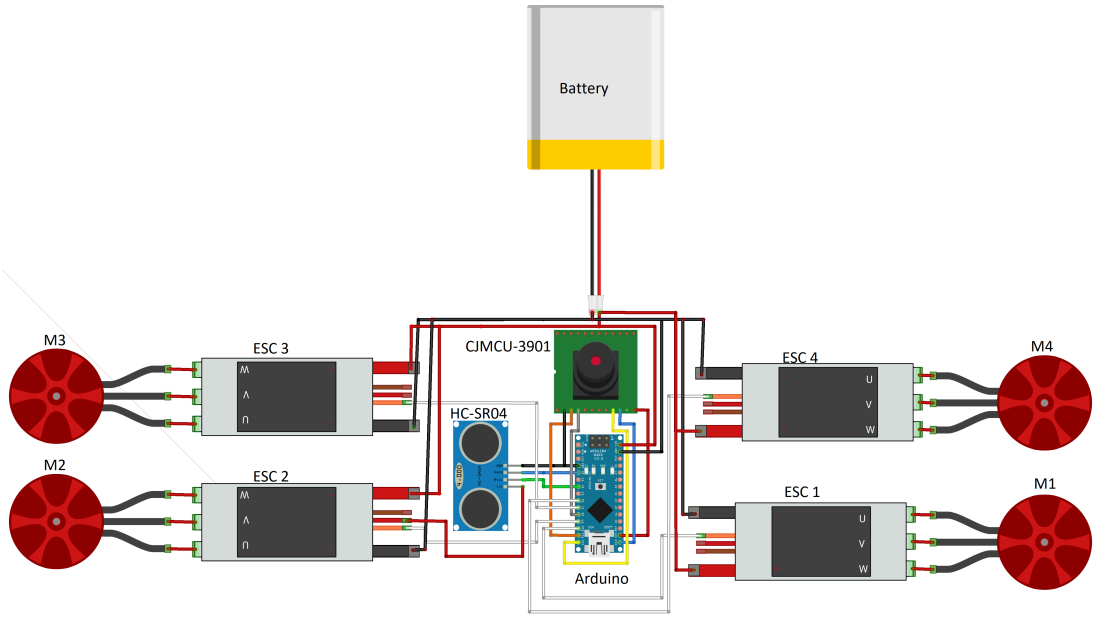


Figure E.1: Wiring diagram for the quadcopter.

Pin Connections

HC-SR04

Sensor	Board
GND	GND
Echo	D2
Trig	D4
VCC	ESC 2 Positive Wire

CJMCU

Sensor	Board
3v3	3v3
MOS	D11
MIS	D12
CLK	D13
CS	D8

ESC 1

ESC	Connectors
V_{in}	Battery Positive Lead
GND	Battery Ground Lead
1	Motor 1 Input 1
2	Motor 1 Input 2
3	Motor 1 Input 3

ESC 2

ESC	Connectors
V_{in}	Battery Positive Lead
GND	Battery Ground Lead
1	Motor 2 Input 3
2	Motor 2 Input 2
3	Motor 2 Input 1

ESC 3

ESC	Connectors
V_{in}	Battery Positive Lead
GND	Battery Ground Lead
1	Motor 1 Input 3
2	Motor 1 Input 2
3	Motor 1 Input 1

ESC 4

ESC	Connectors
V_{in}	Battery Positive Lead
GND	Battery Ground Lead
1	Motor 2 Input 1
2	Motor 2 Input 2
3	Motor 2 Input 3

Table E.1: Wiring connections of the quadcopter.

Appendix F

Thrust Measurement Code

The thrust measurement code (Snippets 16, 17, 18, 20, and 21) is an Arduino project comprised of multiple functions used to allow the user to give precise commands to the ESCs in order to measure the generated thrust for a given command.

```
// ------------------------------------------------------------------------
#include <Servo.h>
// ------------------------------------------------------------------------
// Customize here pulse lengths as needed
#define MIN_PULSE_LENGTH 1000 // Minimum pulse length in µs
#define MAX_PULSE_LENGTH 2000 // Maximum pulse length in µs
// ------------------------------------------------------------------------
Servo motA, motB, motC, motD;
char data;
int perc;
int on_off = 10;
bool motorA = true; bool motorB = false; bool motorC = false; bool motorD = false;
// ------------------------------------------------------------------------

/**
 * Initialisation routine
 */
void setup() {
    Serial.begin(9600);
    motA.attach(10, MIN_PULSE_LENGTH, MAX_PULSE_LENGTH);
    motB.attach(9, MIN_PULSE_LENGTH, MAX_PULSE_LENGTH);
    motC.attach(6, MIN_PULSE_LENGTH, MAX_PULSE_LENGTH);
    motD.attach(5, MIN_PULSE_LENGTH, MAX_PULSE_LENGTH);

    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);

    while(!Serial.available()){
      Serial.println("Press any key to begin");
    }
    displayInstructions();
}
```

Snippet 15: Setup for the thrust measurement Arduino code.

```
void loop() {
    if (Serial.available()) {
        data = Serial.read();

        switch (data) {
            // 0
            case 48 : Serial.println("Sending minimum throttle");
                      motA.writeMicroseconds(MIN_PULSE_LENGTH);
                      motB.writeMicroseconds(MIN_PULSE_LENGTH);
                      motC.writeMicroseconds(MIN_PULSE_LENGTH);
                      motD.writeMicroseconds(MIN_PULSE_LENGTH);
            break;

            // 1
            case 49 : Serial.println("Sending maximum throttle");
                      motA.writeMicroseconds(MAX_PULSE_LENGTH);
                      motB.writeMicroseconds(MAX_PULSE_LENGTH);
                      motC.writeMicroseconds(MAX_PULSE_LENGTH);
                      motD.writeMicroseconds(MAX_PULSE_LENGTH);
            break;

            // 2
            case 50 : Serial.print("Running test in 3");
                      delay(1000);
                      Serial.print(" 2");
                      delay(1000);
                      Serial.println(" 1...");
                      delay(1000);
                      test();
            break;

            // 3
            case 51 : Serial.print("Running manual test in 3");
                      delay(1000);
                      Serial.print(" 2");
                      delay(1000);
                      Serial.println(" 1...");
                      delay(1000);
                      manualtest();
            break;

            // 4
            case 52 : Serial.println("Enable/Disable Motors");
                      motorEnable();
            break;

            // 45
            case 53 : Serial.println("4 Motors Sequential Manual Test");
                      fourMotorsSequential();
            break;
        }
    }
}
```

Snippet 16: Main function for the thrust measurement Arduino code.

```
void test()
{
    for (int i = MIN_PULSE_LENGTH; i <= MAX_PULSE_LENGTH; i += 5) {
        Serial.print("Pulse length = ");
        Serial.println(i);

        motA.writeMicroseconds(i);
        motB.writeMicroseconds(i);
        motC.writeMicroseconds(i);
        motD.writeMicroseconds(i);

        delay(200);
    }

    Serial.println("STOP");
    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);
    delay(2000);
    displayInstructions();
}

void manualtest()
{
    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);
    while(1) {
      if (Serial.available() > 0) {
        perc = Serial.parseInt();
        data = Serial.read();
      }
      int i = MIN_PULSE_LENGTH + (MAX_PULSE_LENGTH-MIN_PULSE_LENGTH)*perc/100;
      if (i < MIN_PULSE_LENGTH){
        i = MIN_PULSE_LENGTH;
      } else if (i > MAX_PULSE_LENGTH) {
          i = MAX_PULSE_LENGTH;
      }
      Serial.print("Pulse length = ");
      Serial.println(i);

      if (perc == -1){
        break;
      }

      motA.writeMicroseconds(i*motorA+!motorA*MIN_PULSE_LENGTH);
      motB.writeMicroseconds(i*motorB+!motorB*MIN_PULSE_LENGTH);
      motC.writeMicroseconds(i*motorC+!motorC*MIN_PULSE_LENGTH);
      motD.writeMicroseconds(i*motorD+!motorD*MIN_PULSE_LENGTH);

      delay(200);
    }

    Serial.println("STOP\n");
    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);
    displayInstructions();
}
```

Snippet 17: Associated functions for the thrust measurement Arduino code: Part 1.

```
void motorEnable()
{
  Serial.println("Motor Status:");
  Serial.println("Press 1 to Start");
  while(1){
    if (Serial.available() > 0) {
        on_off = Serial.parseInt();
        data = Serial.read();
      }
      if (on_off == 1){
        on_off = -1;
        break;
      }
  }
  // Motor 1
  Serial.print("Motor 1: ");
  if (motorA){
    Serial.println("enabled");
  } else{
    Serial.println("disabled");
  }
  Serial.print("Set Motor 1 Status: ");
  while(1){
    if (Serial.available() > 0) {
        on_off = Serial.parseInt();
        data = Serial.read();
      }
    if (on_off == 1){
      motorA = true;
      Serial.println("enabled");
      on_off = -1;
      break;
    } else if (on_off == 0){
      motorA = false;
      on_off = -1;
      Serial.println("disabled");
      break;
    }
  }

  // Motor 2
  Serial.print("Motor 2: ");
  if (motorB){
    Serial.println("enabled");
  } else{
    Serial.println("disabled");
  }
  Serial.print("Set Motor 2 Status: ");
  while(1){
    if (Serial.available() > 0) {
        on_off = Serial.parseInt();
        data = Serial.read();
      }
    if (on_off == 1){
      motorB = true;
      Serial.println("enabled");
      on_off = -1;
      break;
    } else if (on_off == 0){
      motorB = false;
      on_off = -1;
      Serial.println("disabled");
      break;
    }
  }
```

Snippet 18: Associated functions for the thrust measurement Arduino code: Part 2.

```
  // Motor 3
  Serial.print("Motor 3: ");
  if (motorB){
    Serial.println("enabled");
  } else{
    Serial.println("disabled");
  }
  Serial.print("Set Motor 3 Status: ");
  while(1){
    if (Serial.available() > 0) {
        on_off = Serial.parseInt();
        data = Serial.read();
      }
    if (on_off == 1){
      motorC = true;
      Serial.println("enabled");
      on_off = -1;
      break;
    } else if (on_off == 0){
      motorC = false;
      on_off = -1;
      Serial.println("disabled");
      break;
    }
  }

  // Motor 4
  Serial.print("Motor 4: ");
  if (motorD){
    Serial.println("enabled");
  } else{
    Serial.println("disabled");
  }
  Serial.print("Set Motor 4 Status: ");
  while(1){
    if (Serial.available() > 0) {
        on_off = Serial.parseInt();
        data = Serial.read();
      }
    if (on_off == 1){
      motorD = true;
      on_off = -1;
      Serial.println("enabled");
      break;
    } else if(on_off == 0){
      motorD = false;
      on_off = -1;
      Serial.println("disabled");
      break;
    }
  }
  delay(1000);
  on_off = -1;
  displayInstructions();

}
```

Snippet 19: Associated functions for the thrust measurement Arduino code: Part 3.

```
void fourMotorsSequential()
{
    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);
    int j = 0;
    int k = 0;
    int last_perc = 0;
    while(1) {

      if (Serial.available() > 0) {
        perc = Serial.parseInt();
        if (perc != 1){
          last_perc = perc;
        }
        data = Serial.read();
      }
      int i = MIN_PULSE_LENGTH + (MAX_PULSE_LENGTH-MIN_PULSE_LENGTH)*perc/100;
      if (i < MIN_PULSE_LENGTH){
        i = MIN_PULSE_LENGTH;
      } else if (i > MAX_PULSE_LENGTH) {
          i = MAX_PULSE_LENGTH;
      }
      Serial.print("Pulse length = ");
      Serial.print(i);
      Serial.print(" sent to Motor ");
      Serial.println(j+1);

      if (perc == -1){
        break;
      }
      if (perc == 1){
        k = 0;
        if (j < 3){
          j++;
          perc = last_perc;
        } else {
          j = 0;
          perc = last_perc;
        }

      }
      if (perc == 4){
        k = 4;
      }

      motA.writeMicroseconds(i*(j==0 || k == 4)+!(j==0 || k == 4)*MIN_PULSE_LENGTH);
      motB.writeMicroseconds(i*(j==1 || k == 4)+!(j==1 || k == 4)*MIN_PULSE_LENGTH);
      motC.writeMicroseconds(i*(j==2 || k == 4)+!(j==2 || k == 4)*MIN_PULSE_LENGTH);
      motD.writeMicroseconds(i*(j==3 || k == 4)+!(j==3 || k == 4)*MIN_PULSE_LENGTH);
      delay(200);
    }

    Serial.println("STOP\n");
    motA.writeMicroseconds(MIN_PULSE_LENGTH);
    motB.writeMicroseconds(MIN_PULSE_LENGTH);
    motC.writeMicroseconds(MIN_PULSE_LENGTH);
    motD.writeMicroseconds(MIN_PULSE_LENGTH);
    displayInstructions();
}
```

Snippet 20: Associated functions for the thrust measurement Arduino code: Part 4.

```
/**
 * Displays instructions to user
 */
void displayInstructions()
{
    Serial.println("READY - PLEASE SEND INSTRUCTIONS AS FOLLOWING :");
    Serial.println("\t0 : Send min throttle");
    Serial.println("\t1 : Send max throttle");
    Serial.println("\t2 : Run test function");
    Serial.println("\t3 : Run manual test function");
    Serial.println("\t4 : Enable/Disable Motors");
    Serial.println("\t5 : 4 Motors Sequential Manual Test\n");

}
```

Snippet 21: Associated functions for the thrust measurement Arduino code: Part 5.

Appendix G

Guidance Systems

A guidance system is a system used to determine the trajectory of a vehicle from its current location to a designated target location. The guidance system is responsible for determining the trajectory itself, as well as changes in the vehicle's velocity, rotation, and/or acceleration for following the trajectory [30, 31]. The guidance system does this via processing the vehicle's current state vector and comparing it with the desired target state vector to generate a series of reference signals which it may then pass to the control system. A guidance system uses the output of the navigation system as its own input. The input from the navigation system is then processed through a guidance law to determine the reference signals which are then output to the control system. A guidance law is a mathematical or geometric construct used to determine what trajectory the vehicle should take. Guidance laws may be as simple as go to a set point or as complex as determining the optimal trajectory to intercept an evasive target.

A guidance system may be designed by first developing the geometric rule which defines it, and from there, the guidance law itself may be determined.

G.1 Open-Loop Guidance

Open-loop guidance, like open-loop control, is a system by which the reference signal generated by the guidance system is not influenced by the state of the system as state information is not fed back into the guidance system. At pre-determined times, preset reference signals are generated and transmitted to the control system for a predetermined amount of time.

G.1.1 Go-to-Point

One such open-loop guidance system is a "go-to-point" guidance system. This system had a single set point in space which is the desired location for the vehicle. This is one of the simplest forms of open-loop guidance as there is no calculation involved in the

determination of the reference signal for the controller. The final position of the vehicle is almost entirely reliant on the control system.

G.2 Closed-Loop Guidance

Closed-loop guidance, like closed-loop control, is a system by which the reference signal generated by the guidance system is influenced by the state of the system as state information. State information from the vehicle is fed back into the guidance system influencing the generation of the new reference signal to be sent to the control system.

G.2.1 Waypoint Guidance

Waypoint guidance is similar to go-to-point guidance as it too outputs directly the desired position of the quadcopter, however unlike the simple go-to-point guidance system, waypoint guidance iterates through a series of waypoints. Waypoint guidance is a closed-loop guidance system which relies on the current state of the vehicle to determine the desired reference signal. As the vehicle approaches the guidance system checks whether or not the waypoint has been reached. Determination of whether the waypoint has been reached can be achieved in multiple ways. The simplest manner is whether or not the vehicle passed either through the point itself, or within a small enough region around the point. This method increases the likelihood of the vehicle overshooting rather than stopping at the waypoint. An alternative method is to begin calculating how long the vehicle remains in a set region around the waypoint, resetting the clock if the vehicle leaves the region, and only register the waypoint as having been reached after a set amount of time has passed with the vehicle inside the region. Once reached, the current waypoint being used as a reference signal is replaced by the next waypoint stored in the queue. When the vehicle has exhausted the queue, the system will enact some sort of protocol to determine the continuation of the flight. These protocols include, but are not limited to, maintaining position at the final waypoint, shutting down, and enacting a cyclical protocol where it returns to the first waypoint and repeats the process until otherwise interrupted.

Appendix H

Guidance Systems Code

This appendix contains a number of code snippets required for the implementation of various guidance systems into both the simulator environment and the flight code running the quadcopter.

Snippet 22 provides the code required to implement an open-loop guidance system into the simulator environment.

```
function Ref = Guidance(time, state,y)

% State Estimates
StateEstim = state;

% Sensor Model
Sensors = y;

%% Guidance Law
% Reference State
Ref.x          = 0;    % [m]
Ref.y          = 0;    % [m]
Ref.z          = 1;    % [m]
Ref.phi        = 0;    % [rad]
Ref.theta      = 0;    % [rad]
Ref.psi        = 0;    % [rad]
Ref.dx         = 0;    % [m/s]
Ref.dy         = 0;    % [m/s]
Ref.dz         = 0;    % [m/s]
Ref.dphi       = 0;    % [deg/sec]
Ref.dtheta     = 0;    % [deg/sec]
Ref.dpsi       = 0;    % [deg/sec]

if time > 20 && time < 22
    Ref.theta = deg2rad(5);
end

if time > 27 && time < 29
    Ref.theta = deg2rad(-5);
end

end
```

Snippet 22: Open loop guidance system for the flight simulator.

Snippet 23 provides the code required to implement an waypoint guidance system into the simulator environment. Under this guidance system, the vehicle will advance from waypoint to waypoint, and upon arrival at the final waypoint, remain in place.

Snippet 24 provides the code required to implement an waypoint guidance system into the simulator environment. Under this guidance system, the vehicle will advance from waypoint to waypoint, and upon arrival at the final waypoint, the vehicle will return to the first waypoint and repeat the cycle.

Snippet 25 provides the code required to implement a go-to-point guidance system into the flight code.

The final snippet of code for the guidance systems is Snippet 26. This snippet provides the code required to implement a waypoint guidance system into the simulator environment. Setting the `bool` value for `cycle` to true or false determines whether or not the vehicle will cycle through all of the waypoints repeatedly, or remain stationary at the final waypoint.

```
function Ref = WaypointGuidance(time, state,y)

% State Estimates
StateEstim = state;

% Sensor Model
Sensors = y;

%% Define Reference Signal Structure
% Reference State
Ref.x          = 0;     % [m]
Ref.y          = 0;     % [m]
Ref.z          = 1;     % [m]
Ref.phi        = 0;     % [rad]
Ref.theta      = 0;     % [rad]
Ref.psi        = 0;     % [rad]
Ref.dx         = 0;     % [m/s]
Ref.dy         = 0;     % [m/s]
Ref.dz         = 0;     % [m/s]
Ref.dphi       = 0;     % [deg/sec]
Ref.dtheta     = 0;     % [deg/sec]
Ref.dpsi       = 0;     % [deg/sec]


%% Guidance Law
% Define Waypoints
Waypoints = [0 0 1;
             1 0 1;
             1 1 1;
             0 0 1];
neighborhood = 0.01; % Define Neighborhood of Waypoint
nTime = 2;           % Time the drone must stay in neighborhood to advance

% Identify Current Waypoint Number
persistent WaypointID nTimer
if isempty(WaypointID)
    WaypointID = 1;
    nTimer = 0;
end

% Set Reference Signal
Ref.x = Waypoints(WaypointID,1);
Ref.y = Waypoints(WaypointID,2);
Ref.z = Waypoints(WaypointID,3);

% Check if in neighborhood of Waypoint
range = sum((Waypoints(WaypointID,:)-[StateEstim.x StateEstim.y StateEstim.z]).^2);

if range <= neighborhood
    % If yes, check timer
    if nTimer == 0
        nTimer = time;
    end
    if (time - nTimer) >= nTime && WaypointID ~= length(Waypoints)
        WaypointID = WaypointID + 1;
    end
else
    % if no, set timer to zero
    if nTimer ~= 0
        nTimer = 0;
    end
end

end
```

Snippet 23: Waypoint guidance system for the flight simulator.

```
function Ref = WaypointGuidance_Cyclic(time, state,y)

% State Estimates
StateEstim = state;

% Sensor Model
Sensors = y;

%% Define Reference Signal Structure
% Reference State
Ref.x          = 0;    % [m]
Ref.y          = 0;    % [m]
Ref.z          = 1;    % [m]
Ref.phi        = 0;    % [rad]
Ref.theta      = 0;    % [rad]
Ref.psi        = 0;    % [rad]
Ref.dx         = 0;    % [m/s]
Ref.dy         = 0;    % [m/s]
Ref.dz         = 0;    % [m/s]
Ref.dphi       = 0;    % [deg/sec]
Ref.dtheta     = 0;    % [deg/sec]
Ref.dpsi       = 0;    % [deg/sec]


%% Guidance Law
% Define Waypoints
Waypoints = [0 0 1;
             1 0 1;
             0 0 1];
neighborhood = 0.01; % Define Neighborhood of Waypoint
nTime = 1.5;         % Time the drone must stay in neighborhood to advance

% Identify Current Waypoint Number
persistent WaypointID nTimer
if isempty(WaypointID)
    WaypointID = 1;
    nTimer = 0;
end

% Set Reference Signal
Ref.x = Waypoints(WaypointID,1);
Ref.y = Waypoints(WaypointID,2);
Ref.z = Waypoints(WaypointID,3);

% Check if in neighborhood of Waypoint
range = sum((Waypoints(WaypointID,:)-[StateEstim.x StateEstim.y StateEstim.z]).^2);

if range <= neighborhood
    % If yes, check timer
    if nTimer == 0
        nTimer = time;
    end
    if (time - nTimer) >= nTime
        WaypointID = WaypointID + 1;
        if WaypointID > length(Waypoints)
            WaypointID = 1;
        end
    end
else
    % if no, set timer to zero
    if nTimer ~= 0
        nTimer = 0;
    end
end

end
```

Snippet 24: Cyclic waypoint guidance system for the flight simulator.

```
class FixedPointGuidance : public Guidance {
    float x, y, z;

    public:
    FixedPointGuidance(float x, float y, float z) {
        this->x = x;
        this->y = y;
        this->z = z;
    }

    void update(SensorReadings &readings, StateVector &state, StateVector &desired_state) {
        desired_state.x = x;
        desired_state.y = y;
        desired_state.z = z;
    }
};
```

Snippet 25: Go-to-Point guidance for the flight code.

```
class WaypointGuidance : public Guidance {
    queue<Waypoint> waypoints;

    bool in_neighborhood;
    bool cycle;
    unsigned long start_ms;

    public:
    WaypointGuidance(queue<Waypoint> waypoints, bool cycle = false) {
        this->waypoints = waypoints;
        this->in_neighborhood = false;
        this->cycle = cycle;
    }

    void update(SensorReadings &readings, StateVector &state, StateVector &desired_state) {
        if (waypoints.size() == 1) {
            desired_state.x = waypoints.front().x;
            desired_state.y = waypoints.front().y;
            desired_state.z = waypoints.front().z;
        }

        else {
            // Neighborhood rule
            bool curr_n = (distance(state, waypoints.front()) < 0.3);

            if (!in_neighborhood && curr_n) {
                in_neighborhood = true;
                start_ms = millis();
            }

            else if (in_neighborhood && !curr_n) {
                in_neighborhood = false;
            }

            else if (in_neighborhood && curr_n) {
                if (millis() - start_ms > 100) {
                    if (cycle) waypoints.push(waypoints.front());
                    waypoints.pop();
                }
            }

            desired_state.x = waypoints.front().x;
            desired_state.y = waypoints.front().y;
            desired_state.z = waypoints.front().z;
        }
    }
};
```

Snippet 26: Waypoint guidance for the flight code.

Appendix I

State Estimators

A state estimator is a process by which we estimate the true state (χ) of the system, often from partial and noisy measurement data. Typically, we cannot directly observe the current state, as such, we may use a state estimator to convert sensor measurements of the inputs and outputs of a system, together with a model of the system dynamics, into an estimate of the true state ($\hat{\chi}$).

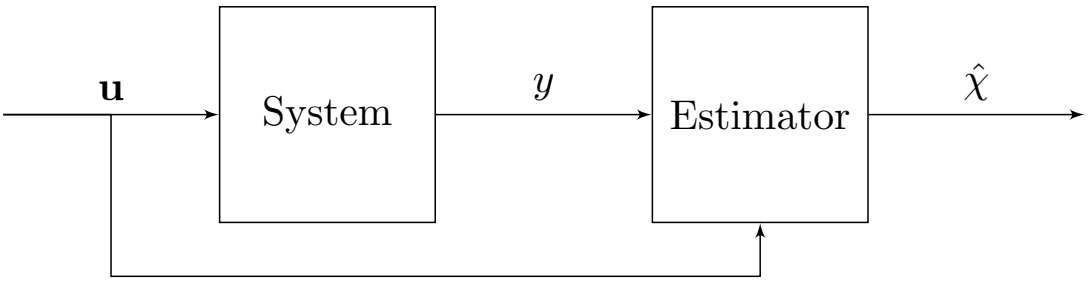


Figure I.1: A high level block diagram of a state estimator.

I.0.1 Complementary Filter

The *Complementary Filter* is an estimator which takes different measurements of a given signal and passes each measurement through a filter which is the *complement* of the other. If we define one filter as $H(s)$ then its complement would be $[1 - H(s)]$. A common form of complementary filter is a combination of a High-Pass Filter

$$H_{hpf}(s) = \frac{s}{s + \omega_c}, \tag{I.1}$$

with a Low-Pass Filter

$$H_{lpf}(s) = \frac{\omega_c}{s + \omega_c}, \tag{I.2}$$

using the same cut-off frequency (ω_c). If we take the standard form of the High-Pass Filter (I.1), we would find that its complement is,

$$1 - H_{hpf}(s) = 1 - \frac{s}{s+\omega_c} = \frac{(s+\omega_c) - s}{s+\omega_c} = \frac{\omega_c}{s+\omega_c}, \tag{I.3}$$

which we can immediately recognize as that it is indeed the standard form of a Low-Pass Filter (I.2). By choosing the correct sensors to use with each filter, and choosing an appropriate cut-off frequency, we can greatly increase the accuracy of our state estimate. Such an estimator can be used on-line in real-time applications using minimal computational effort while simultaneously providing high estimation accuracy when tuned properly and paired with adequate sensors.

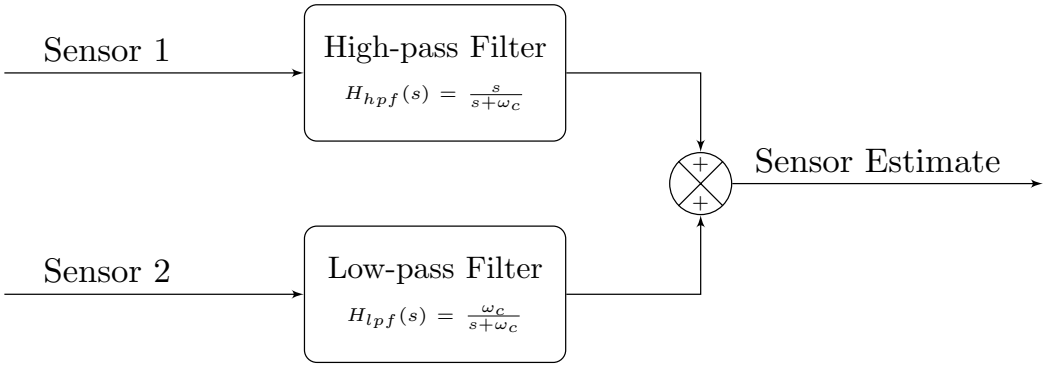


Figure I.2: Complementary filter model.

I.0.2 Kalman Filter

The Kalman Filter is an optimal linear estimator, described in R.E. Kalman's seminal paper from 1960 [32], which works using a two-step prediction/update process to recursively solve the discrete-data linear filtering problem. The Kalman Filter is Markov, in that its current estimate is reliant only on measurements and predictions from the current time-step and the previous state estimate. Two models are required for a Kalman Filter, the process model and the observation model, defined as:

$$\begin{aligned} \mathrm{x_k} &= \mathrm{F x_{k-1}} + \mathrm{B u_{k-1}} + \Gamma \mathrm{w_{k-1}} \\ \mathrm{z_k} &= \mathrm{H x_k} + \mathrm{v_k} \end{aligned} \tag{I.4}$$

Where x is our state vector, F is our state-transition matrix, B is our control-input matrix, u is our control vector, w is our process noise (assumed to be a zero-mean Gaussian with covariance Q), z is our observation vector, H is our observation matrix, v is the measurement noise vector (assumed to be a zero-mean Gaussian with covariance R).

The first step of the Kalman Filter estimation process is the prediction stage which relies on the dynamical model of the system and the command inputs to predict the state of the system.

$$\begin{aligned} \hat{x}_k^- &= F_k \hat{x}_{k-1}^+ + B_k u_{k-1} \\ P_k^- &= F_k P_{k-1}^+ F_k^T + \Gamma Q_k \Gamma^T \end{aligned} \tag{I.5}$$

The second step is the update stage which updates our estimates based on sensor measurements.

$$\begin{aligned} K_k &= P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \\ \hat{x}_k^+ &= \hat{x}_k^- + K_k (z_k - H_k \hat{x}_k^-) \\ P_k^+ &= (I - K_k H_k) P_k^- \end{aligned} \tag{I.6}$$

The hat operator ($\hat{}$) refers to the estimate of a variable, and the superscripts ($^-$) and ($^+$) refer to the prior and posterior estimates respectively. By iteratively repeating these steps, and provided that system being estimated using the Kalman Filter is observable (i.e.- the state may be determined by the outputs) and reachable (i.e.- there exists inputs which can drive it to any state), then the Kalman Filter is asymptotically stable and the estimate converges towards the true state.

I.1 Quadcopter Specific Estimators

Using the above estimators, we can determine estimators specific to our quadcopter with its embedded sensors.

Pitch and Roll Estimators

The pitch and roll estimators are used to determine the pitch and roll angles, θ and ϕ respectively, of the quadcopter. From (4.11) we can find that:

$$F^{\mathcal{B}} = \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ -T_i \end{bmatrix} + R^T \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} = \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ T_i \end{bmatrix} + \begin{bmatrix} -s_\theta \\ c_\theta s_\phi \\ c_\phi c_\theta \end{bmatrix} mg. \tag{I.7}$$

We can divide the forces felt ($F^{\mathcal{B}}$) by the mass of the quadcopter (m) to determine the accelerations felt in the body-frame of the quadcopter,

$$a = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \frac{1}{m} F^{\mathcal{B}}. \tag{I.8}$$

We can then use trigonometry to find the pitch and roll angles based on the accelerometer

readings using (I.9), and assuming no accelerations besides gravity,

$$\begin{cases} a_x = -g\sin(\theta) \Rightarrow \hat{\theta} = -\arcsin(\frac{a_x}{g}) \\ \frac{a_y}{a_z} = \tan(\phi) \Rightarrow \hat{\phi} = \arctan(\frac{a_y}{a_z}) \end{cases} . \tag{I.9}$$

Estimating the pitch and roll angles via the gyro is a simple integration of the angular rate by the elapsed time,

$$\begin{cases} \hat{\theta}(t) & = \int_0^t q(\tau)d\tau \\ \hat{\phi}(t) & = \int_0^t p(\tau)d\tau \end{cases} . \tag{I.10}$$

The two sensor estimates are excellent candidates for a Complementary Filter approach to the sensor fusion as they are improved by a high-pass and low-pass filter respectively. Over the short term, the gyro sensor is highly accurate in determining the attitude angle, over the long term, it is prone to drift. In contrast, the angle estimate generated by the accelerometers over the short term is noisy, while over the long term, it provides an accurate measurement of the angle. It is worth noting that the angle estimate determined by the accelerometers provides inaccurate results during acceleration of the quadcopter as the accelerometers read the acceleration of the quadcopter in addition to the acceleration due to gravity. Further, for an angular complementary filter using a gyro and acceleromter (Figure I.3) we can further simplify the filter structure to a single filter on the combined signal from the two sensors (Figure I.4).

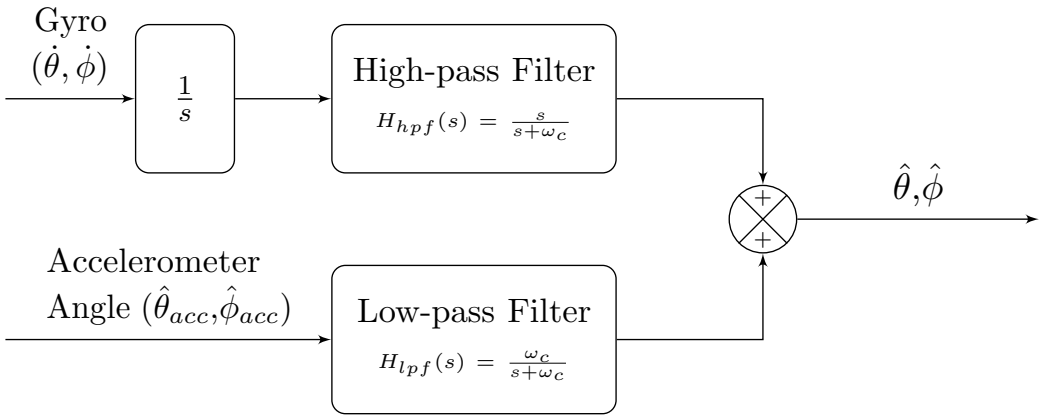


Figure I.3: Angular Complementary Filter.

In general, the cut-off frequency of the Complementary Filter is determined experimentally. Typical values for the cutoff frequency for Complementary Filters using MEMS gyros to determine pitch and roll angles are $6.28 \leq \omega_c \leq 8.84[rad/sec]$ [33].

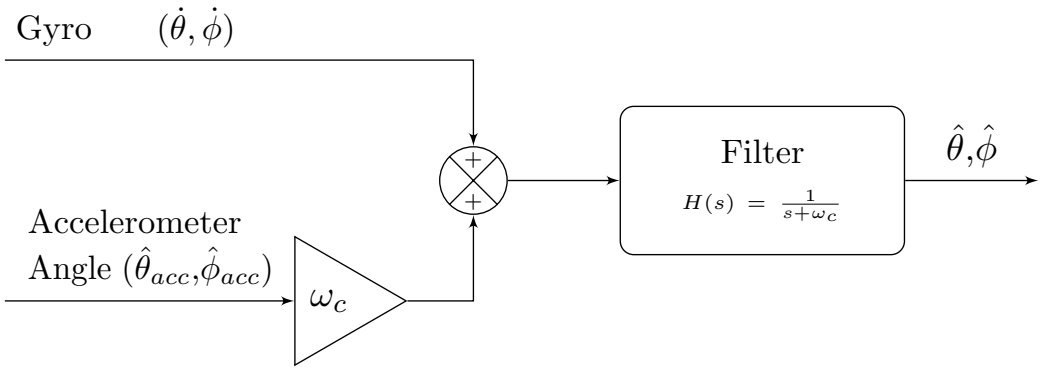


Figure I.4: Simplified Angular Complementary Filter.

Yaw Estimator

The yaw estimator works along similar principles to the pitch and roll estimators, however one major difference between the two is that the accelerometers cannot provide an estimate of the yaw angle. To this end, we can instead substitute the accelerometer reading for one from either the magnetometer, or from the optical flow sensor. Fortunately, as with the accelerometer, while both of the aforementioned sensors suffer from noise over the short term, over the long term, they produce accurate measurements of the yaw angle. If we assume no hard-iron interference with the magnetometer, from [34] we can find

$$\hat{\psi} = \arctan\left(\frac{m_z \sin\phi - m_z \cos\phi}{m_x \cos\theta + m_y \sin\theta + m_z \sin\theta \cos\phi}\right), \tag{I.11}$$

which translates our magnetometer readings to a compass heading angle. After calibrating to our initial compass heading angle, we can find the estimate of the yaw angle. Like the accelerometer, it is accurate over longer periods of time, but less accurate in the short term, and as such a Low-Pass Filter can help compensate. As mentioned, the optical flow sensor may be used to determine the yaw angle as well, however the methodology is not in the scope of this thesis. This can also be simplified as with the pitch and roll estimators to generate the simplified complementary filter found in Figure I.5

I.1.1 Height Estimator

The height estimator estimates the height of our quadcopter in the world frame. Height estimation is critical for maintaining an altitude hold of the quadcopter, about which point we are linearizing our system. The height estimator receives sensor data from any combination of the following sensors: IMU, barometric pressure sensor, and ultrasonic sensor. In general, a Kalman Filter is the standard height estimator used for the sensor fusion of height data. It is worth noting that while the IMU and ultrasonic sensors

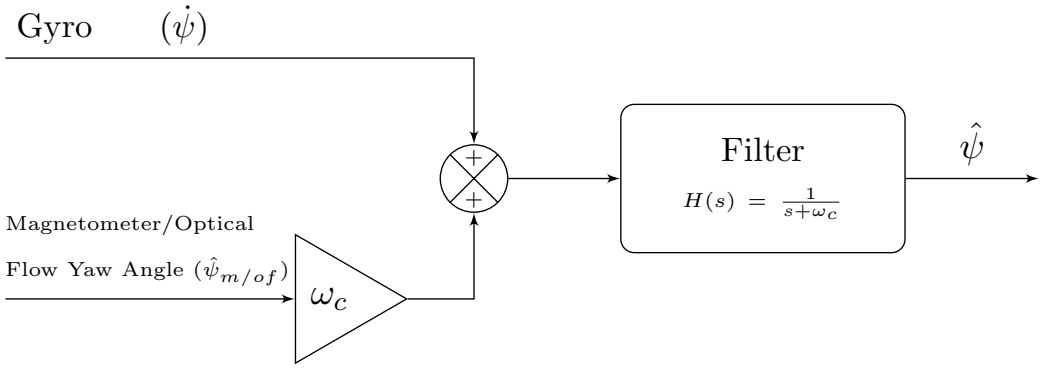


Figure I.5: Simplified Yaw Angle Complementary Filter.

may be used to directly estimate the height of the quadcopter, the barometric pressure sensors is used to measure the *altitude* and thus must be compensated for during the calculations. Here, we use a Kalman filter to fuse measurements from the ultrasonic sensor and the barometric pressure sensor.

The Kalman filter is based on a discretization of the altitude dynamics of the quadcopter. In this direction, let $h(t)$ denote the height of the quadcopter above the ground (this is equivalent to the inertial quadcopter state $-z(t)$). Then the alitude dynamics can be expressed simply as $\ddot{h}(t) = \frac{1}{m}u_T^{\mathcal{I}}(t) + w(t)$, where $u_1^{\mathcal{I}}(t)$ is the thrust input expressed in the inertial frame (including for simplicity also the gravitational term), and $w(t)$ is process noise associated with the motors. To enable estimation of the altitude from the barometric pressure sensor, we introduce a "phantom" state $b(t)$ with no dynamics, i.e. $\dot{b}(t) = 0$. Defining now $\dot{h}(t) = v_h(t)$, the complete state vector can be expressed as $\mathrm{x}(t) = \begin{bmatrix} h(t) & v_h(t) & b(t) \end{bmatrix}$. Considering a sample time of dt, we can discretize this model to obtain the difference equation

$$\mathrm{x}[k] = \underbrace{\begin{bmatrix} 1 & dt & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathrm{F}} \begin{bmatrix} h[k-1] \\ v_h[k-1] \\ b[k-1] \end{bmatrix} + \underbrace{\begin{bmatrix} \frac{1}{2m}dt^2 \\ \frac{1}{m}dt \\ 0 \end{bmatrix}}_{\mathrm{B}} u_T^{\mathcal{I}}[k-1] + \underbrace{\begin{bmatrix} \frac{1}{2}dt^2 \\ dt \\ 0 \end{bmatrix}}_{\Gamma} w[k-1] \tag{I.12}$$

$$\mathrm{z}[\mathrm{k}] = \underbrace{\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}}_{\mathrm{H}} \begin{bmatrix} h[k] \\ v_h[k] \\ b[k] \end{bmatrix} + v[k].$$

The measurement $h[k] + b[k]$ represents the correction of the barometric pressure height measurement which is calibrated with an initial condition corresponding to the ground height. The process noise is modeled as a zero-mean Gaussian signal with covariance

σ_a,

$$\begin{cases} w & \sim \mathcal{N}(0, Q) \\ Q & = \sigma_a^2 \end{cases} . \tag{I.13}$$

We assume for simplicity that the measurement noises are not correlated. The pressure sensor noise is zero-mean Gaussian with covariance σ_b. The ultrasonic sensor is highly accurate only within a certain range, while outside that range the sensor readings are relatively arbitrary and nonsensical. As such, we model this with a switching value for the noise covariance of the sensor that depends on the current height reading. For the ultrasonic sensor used with the quadcopter developed for this thesis, we define the switching function as

$$\sigma_u = \begin{cases} \sigma_{u,1}, & \texttt{if } 0.02[m] < h < 4[m] \\ \sigma_{u,2}, & \texttt{else} \end{cases},$$

thus

$$\begin{cases} v & \sim \mathcal{N}(0, R) \\ R & = \begin{bmatrix} \sigma_b^2 & 0 \\ 0 & \sigma_u^2 \end{bmatrix} \end{cases} .$$

With this model we are able to implement the Kalman filter outline in I.0.2.

For the practical implementation of the height Kalman filter, we make use of the accelerometers to input the value $u_T^{\mathcal{I}}$ based on the sensor readings rather than relying on a transformation of the controller output.

I.1.2 Lateral Position Estimator

A combination of the IMU and the optical flow sensor may be used to accurately estimate the lateral position of the quadcopter. The estimation of the lateral position is critical for trajectory tracking, a key feature of many guidance systems. The use of a Kalman Filter is the most common method of estimating lateral position. The Kalman Filter is based on the discretization of the planar dynamics of the quadcopter. To this end, we denote the lateral positions $x(t)$ and $y(t)$ as the displacement from the initial position, which for simplicity we denote as $(0, 0)$. The planar dynamics can be expressed as

$$\begin{cases} \ddot{x}(t) & = \mathrm{a_x}^{\mathcal{I}}(t) + w_x(t) \\ \ddot{y}(t) & = \mathrm{a_y}^{\mathcal{I}}(t) + w_y(t), \end{cases} \tag{I.14}$$

where $\mathrm{a_x}^{\mathcal{I}}$ and $\mathrm{a_y}^{\mathcal{I}}$ are the inertial frame lateral accelerations in the x and y directions

respectively. Notably, the inertial frame accelerations are a function of the Euler angles and thrust. To simplify we will retain the $\mathrm{a_x}^{\mathcal{I}}$ and $\mathrm{a_y}^{\mathcal{I}}$ notation. Further, we may define $\dot{x} = v_h(t)$ and $\dot{y} = v_y(t)$, which will allow us to define the complete state vector as $\mathrm{x} = \begin{bmatrix} x & y & v_x & v_y \end{bmatrix}$. Considering a sample time of dt, we can discretize the model to obtain the difference equation

$$\begin{aligned} \mathrm{x}[k] &= \underbrace{\begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathrm{F}} \begin{bmatrix} x[k-1] \\ y[k-1] \\ v_x[k-1] \\ v_y[k-1] \end{bmatrix} + \underbrace{\begin{bmatrix} \frac{1}{2}dt^2 & 0 \\ 0 & \frac{1}{2}dt^2 \\ dt & 0 \\ 0 & dt \end{bmatrix}}_{\mathrm{B}} \begin{bmatrix} \mathrm{a_x}^{\mathcal{I}} \\ \mathrm{a_y}^{\mathcal{I}} \end{bmatrix} + \underbrace{\begin{bmatrix} \frac{1}{2}dt^2 & 0 \\ 0 & \frac{1}{2}dt^2 \\ dt & 0 \\ 0 & dt \end{bmatrix}}_{\Gamma} w[k] \\ \mathrm{z}[k] &= \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathrm{H}} \begin{bmatrix} x[k] \\ y[k] \\ v_x[k] \\ v_y[k] \end{bmatrix} + v[k]. \end{aligned} \tag{I.15}$$

The process and measurement noises are modeled as a zero-mean Gaussian noises with covaraince matrices Q and R respectively,

$$\begin{cases} w & \sim \mathcal{N}(0, Q) \\ Q & = \begin{bmatrix} \sigma_{a_x}^2 & 0 \\ 0 & \sigma_{a_y}^2 \end{bmatrix} \end{cases} \tag{I.16}$$

$$\begin{cases} v & \sim \mathcal{N}(0, R) \\ R & = \begin{bmatrix} \sigma_{v_x}^2 & 0 \\ 0 & \sigma_{v_y}^2 \end{bmatrix}, \end{cases} \tag{I.17}$$

where $\sigma_{a_x}^2$ is the process noise variance for acceleration in the x direction, $\sigma_{a_y}^2$ is the process noise variance for the acceleration in the y, and $\sigma_{v_x}^2$ and $\sigma_{v_y}^2$ are the measurement noise variances for our optical flow sensor. For practical implementation purposes, rather than rely on transforming the controller output to acceleration output, and relying on the linearized assumptions involved, we choose to rely instead on the accelerometer readings to determine the acceleration. With this model we are able to implement the Kalman filter outline in I.0.2.

Appendix J

Flight Code Estimators

This appendix contains a number of code snippets required for the implementation of various filters and state estimators into the flight code.

Snippet 27 provides the code required to implement a simple low-pass filter into the flight code.

```
// From ReadyFilters.hpp
/* Low-pass Filter
 * y(n) = (1-alpha) * x(n) + alpha * y(n-1)
 * alpha = tau / (tau + dt)
 * */
class LowPass : public Estimator {
    private:
    float alpha;
    sensor_field_ptr lpf_sensor_field;
    state_field_ptr output_field;

    public:
    LowPass(sensor_field_ptr lpf_sensor_field,
                                state_field_ptr output_field, float alpha = DEFAULT_ALPHA);
    void update(SensorReadings &readings, StateVector &state) override;
    // ~LowPass();
};

// From ReadyFilters.cpp
LowPass::LowPass(sensor_field_ptr lpf_sensor_field,
                 state_field_ptr output_field, float alpha) {
    this->alpha = alpha; // alpha = tau / (tau + dt)
    // this->dt = dt;
    this->lpf_sensor_field = lpf_sensor_field;
    this->output_field = output_field;
}

void LowPass::update(SensorReadings &readings, StateVector &state) {
    float low_pass = readings.*lpf_sensor_field;
    float filtered_state = state.*output_field;
    filtered_state = (1-alpha) * low_pass + alpha*filtered_state;
    state.*output_field = filtered_state;
}
```

Snippet 27: Low-Pass Filter for the flight code.

Snippet 28 provides the code required to implement a simple high-pass filter into

the flight code.

```
// From ReadyFilters.hpp
/* High-pass Filter
 * y(n) = alpha * y(n-1) + alpha * (x(n) - x(n-1))
 * alpha = tau / (tau + dt)
 * */
class HighPass : public Estimator {
    private:
    float alpha;
    float prev_high_pass;
    sensor_field_ptr hpf_sensor_field;
    state_field_ptr output_field;

    public:
    HighPass(sensor_field_ptr hpf_sensor_field,
                                state_field_ptr output_field, float alpha = DEFAULT_ALPHA);
    void update(SensorReadings &readings, StateVector &state) override;
    // ~HighPass();
};

// From ReadyFilters.cpp
HighPass::HighPass(sensor_field_ptr hpf_sensor_field,
                  state_field_ptr output_field, float alpha) {
    this->alpha = alpha; // alpha = tau / (tau + dt)
    // this->dt = dt;
    this->prev_high_pass = 0;
    this->hpf_sensor_field = hpf_sensor_field;
    this->output_field = output_field;
}

void HighPass::update(SensorReadings &readings, StateVector &state) {
    float high_pass = readings.*hpf_sensor_field;
    float filtered_state = state.*output_field;
    filtered_state = alpha * filtered_state + alpha*(high_pass-prev_high_pass);
    prev_high_pass = high_pass;
    state.*output_field = filtered_state;
}
```

Snippet 28: High-Pass Filter for the flight code.

Snippet 29 provides the code required to implement an angular complementary filter into the flight code. This complementary filter is designed to use the gyro sensor along with either the accelerometer or the magnetometer to determine the attitude angle.

Snippets 30 and 31 provide the code required to implement a Kalman Filter for lateral position and velocity estimation into the flight code. This Kalman Filter relies on the accelerometers and the optical flow sensor measurements to generate the state estimates.

Snippets 32 and 33 provide the code required to implement a Kalman Filter for vertical position and velocity estimation into the flight code. This Kalman Filter relies on the accelerometers, the barometer, and the ultrasonic distance sensor measurements to generate the state estimates.

```cpp
// From ReadyFilters.hpp
/* Angular Complementary Filter
 * angle = (alpha)*(angle + gyro * dt) + (1-alpha)*(acc)
 * */
class AngularComplementaryFilter : public Estimator {
    private:
    float alpha;
    sensor_field_ptr low_pass_field, high_pass_field;
    state_field_ptr output_field;

    public:
    AngularComplementaryFilter(sensor_field_ptr low_pass_field,
                               sensor_field_ptr high_pass_field,
                               state_field_ptr output_field, float alpha = DEFAULT_ALPHA);
    void update(SensorReadings &readings, StateVector &state) override;
    // ~AngularComplementaryFilter();
};

// From ReadyFilters.cpp
AngularComplementaryFilter::AngularComplementaryFilter(sensor_field_ptr low_pass_field,
                                                       sensor_field_ptr high_pass_field,
                                                       state_field_ptr output_field,
                                                       float alpha) {
    this->alpha = alpha;
    // this->dt = dt;
    this->low_pass_field = low_pass_field;
    this->high_pass_field = high_pass_field;
    this->output_field = output_field;
}

void AngularComplementaryFilter::update(SensorReadings &readings, StateVector &state) {
    float low_pass = readings.*low_pass_field;
    float high_pass = readings.*high_pass_field;
    float angle = state.*output_field;
    angle = alpha * (angle + high_pass * dt) + (1 - alpha) * (low_pass);
    state.*output_field = angle;
}
```

Snippet 29: Angular complementary filter for the flight code.

```cpp
// From ReadyFilters.hpp
// Kalman Filter used to estimate the lateral position and velocity
class XY_KalmanFilter : public Estimator {
    private:
    BLA::Matrix<2> com;
    BLA::Matrix<2> obs;
    KalmanFilter<4, 2, 2> KF;
    float n_v = 100.0; // Velocity Sensor Noise Variance
    float n_a = 0.01; // Accelerometer Noise Variance

    float m_p = 0.001; // Process Position Noise Variance
    float m_v = 0.001; // Process Velocity Noise Variance

    public:
    XY_KalmanFilter();
    void update(SensorReadings &readings, StateVector &state) override;
    // ~XY_KalmanFilter();
};

// From ReadyFilters.cpp
XY_KalmanFilter::XY_KalmanFilter() {
    // Posterior Covariance Initialization
    // Knowing we start being static at the origin
    KF.P = {
        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  //
    };
    // time evolution matrix
    KF.F = {
        1.0, 0.0, 0.0, 0.0, // x
        0.0, 0.0, 1.0, 0.0, // y
        0.0, 1.0, 0.0, 0.0, // dx
        0.0, 0.0, 0.0, 1.0  // dy
    };
    // command matrix
    KF.B = {
        1.0, 0.0, //
        0.0, 1.0, //
        1.0, 0.0, //
        0.0, 1.0  //
    };
    // measurement matrix
    KF.H = {
        0.0, 0.0, 1.0, 0.0,  // dx measurement
        0.0, 0.0, 0.0, 1.0  // dy measurement
    };
    // measurement covariance matrix
    KF.R = {
        n_v * n_v, 0.0,       // dx measrement noise
        0.0,       n_v * n_v // dy measurement noise
    };
    // model covariance matrix
    KF.Q = {
        m_p * m_p, 0.0,       0.0,       0.0,       // x model position noise
        0.0,       0.0,       m_p * m_p, 0.0,       // y model position noise
        0.0,       m_v * m_v, 0.0,       0.0,       // dx model velocity noise
        0.0,       0.0,       0.0,       m_v * m_v // dy model velocity noise
    };
}
```

Snippet 30: Lateral Kalman Filter for the flight code: Part 1.

```
void XY_KalmanFilter::update(SensorReadings &readings, StateVector &state) {
    KF.F = {
        1.0, 0.0, dt, 0.0,  // x
        0.0, 1.0, 0.0, dt,  // y
        0.0, 0.0, 1.0, 0.0, // dx
        0.0, 0.0, 0.0, 1.0  // dy
    };

    KF.Q = {
        0.00025*pow(dt,4), 0.0,               0.0005*pow(dt,3), 0.0,
        0.0,               0.00025*pow(dt,4), 0.0,              0.0005*pow(dt,3),
        0.0005*pow(dt,3),  0.0,               0.0001*dt * dt,   0.0,
        0.0,               0.0005*pow(0.04,3), 0.0,              0.0001* dt * dt
    };

    KF.B = {
        dt * dt / 2, 0,           // x
        0,           dt * dt / 2, // y
        dt,          0,           // dx
        0,           dt           // dy
    };

    com(0) = readings.acc_x;
    com(1) = readings.acc_y;

    obs(0) = readings.camera_dx;
    obs(1) = readings.camera_dy;

    // APPLY KALMAN FILTER
    KF.update(obs, com);

    state.x = KF.x(0);
    state.y = KF.x(1);
    state.dx = KF.x(2);
    state.dy = KF.x(3);
}
```

Snippet 31: Lateral Kalman Filter for the flight code: Part 2.

```cpp
// From ReadyFilters.hpp
// Kalman Filter used to estimate altitude
class Altitude_KalmanFilter : public Estimator {
    private:
    BLA::Matrix<1> com;
    BLA::Matrix<2> obs;
    KalmanFilter<3, 2, 1> KF;

    float n_b = 1.0;  // Barometer Noise Variance
    float n_u = 0.05; // Ultrasonic Noise Variance

    float m_p = 0.05*dt*dt;  // Process Position Noise Variance
    float m_v = 0.05*dt;     // Process Velocity Noise Variance

    public:
    Altitude_KalmanFilter();
    void update(SensorReadings &readings, StateVector &state) override;
    // ~Altitude_KalmanFilter();
};

// From ReadyFilters.cpp
Altitude_KalmanFilter::Altitude_KalmanFilter() {
    /* Posterior Covariance Initialization
       Knowing we start being static at the origin
       Slight unknown as to barometer ground height
    */
    KF.P = {
        0.0, 0.0, 0.0, //
        0.0, 0.0, 0.0, //
        0.0, 0.0, 10.0 //
    };

    // time evolution matrix
    KF.F = {
        1.0, 0.0, 0.0, // z (height above the ground)
        0.0, 1.0, 0.0, // dz (vertical velocity)
        0.0, 0.0, 1.0, // barometer ground height
    };

    // command matrix
    KF.B = {
        1.0, //
        1.0, //
        0.0  //
    };

    // measurement matrix
    KF.H = {
        1.0, 0.0, 1.0, // barometer measurement
        1.0, 0.0, 0.0  // ultrasonic measurement
    };

    // measurement covariance matrix
    KF.R = {
        n_b * n_b, 0.0, // barometer measrement noise
        0.0, n_u * n_u  // ultrasonic measurement noise
    };

    // model covariance matrix
    KF.Q = {
        0.25*m_p * m_p, 0.5*m_v*m_v*dt, 0.0, // z model position noise
        0.5*m_v*m_v*dt, m_v * m_v,      0.0, // dz model position noise
        0.0             0.0             0.0  // barometer model noise
    };
}
```

Snippet 32: Vertical Kalman Filter for the flight code: Part 1.

```
void Altitude_KalmanFilter::update(SensorReadings &readings, StateVector &state) {
    KF.F = {
        1.0, dt,  0.0, // z (height above the ground)
        0.0, 1.0, 0.0, // dz (vertical velocity)
        0.0, 0.0, 1.0, // barometer ground height
    };

    KF.B = {
        dt * dt / 2, // z
        dt,          // dz
        0            // barometer ground height
    };

    // ENABLE THIS SECTION WHEN ULTRASONIC IS ENABLED
    com(0) = readings.acc_z; // Later change this to account for acceleration in the z
                             // axis of the word frame

    obs(0) = readings.barometer_alt;
    obs(1) = readings.ultrasonic_alt;

    // if (state.z <= 2/100 || state.z >= 3.5) {
    if (readings.ultrasonic_alt == -1) {
        KF.R = {
            n_b * n_b, 0.0,      // barometer measrement noise
            0.0,       10000.0   // Use this when Ultrasonic isn't enabled
        };
        obs(1) = 0;
    } else {
        KF.R = {
        n_b * n_b, 0.0,       // barometer measrement noise
        0.0,       n_u * n_u  // ultrasonic measurement noise
        };
    };

    // APPLY KALMAN FILTER
    KF.update(obs, com);

    state.z = KF.x(0);
    state.dz = KF.x(1);
}
```

Snippet 33: Vertical Kalman Filter for the flight code: Part 2.

Appendix K

Controllers

A control system is mechanism by which the future state of a system can be altered in a desired manner. In general, the control system generates a desired output to the state of the system by the application of tuned inputs via actuators [35, 36, 37]. Control systems may be broadly classified as open-loop control systems, and closed-loop control systems. In an open-loop control system (Figure K.1), the input signal is not compared with the output signal of the system. As the output of the system is not compared to the input, open-loop control systems are, by and large, considered inaccurate, and not reliable enough for many applications. As such, for purposes of this thesis, we will rely

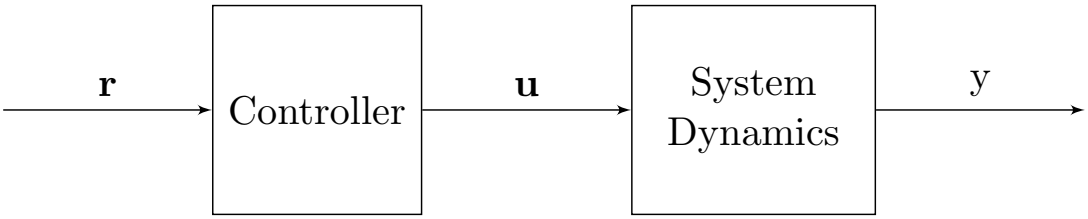


Figure K.1: A block diagram of a generalized open-loop controller.

on closed-loop control systems. A close-loop control system, unlike the open-loop, the output signal of the system is fed back into the system and compared to a reference signal. The difference between the two signals, the error signal, is then fed into the controller (Figure K.2). In a closed-loop control system, the purpose of the controller is to control the system in such a manner that the error signal is driven to zero, thereby bringing the system to the desired output state. This appendix will describe different types of controllers and control systems.

K.1 Proportional-Integral-Derivative (PID) Controller

A Proportional-Integral-Derivative Controller (PID) is a classic closed loop feedback controller. The PID controller is the most common controller used in industry, in part due to their functional simplicity, and in part due to their robust performance in a wide

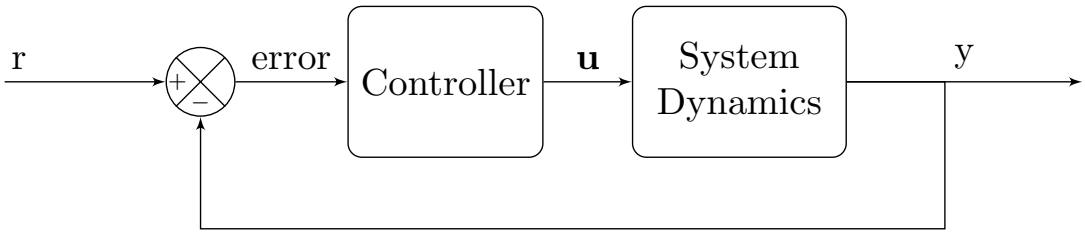


Figure K.2: A block diagram of a generalized closed-loop controller.

range of conditions. The PID controller consists of three terms whose sum constitutes the output of the controller, the proportional term, the integral term, and the derivative term. The equation which governs a PID controller is,

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}, \tag{K.1}$$

where K_p is the proportional gain, K_i is the integral gain, K_d is the derivative gain, $u(t)$ is the control signal, and $e(t)$ is the error signal. The block diagram form of this is shown in Figure K.3. A Laplace transformation of the aforementioned equation (K.1) is generally used to describe the standard form of the PID Controller,

$$C(s) = K_p + \frac{K_i}{s} + K_d s. \tag{K.2}$$

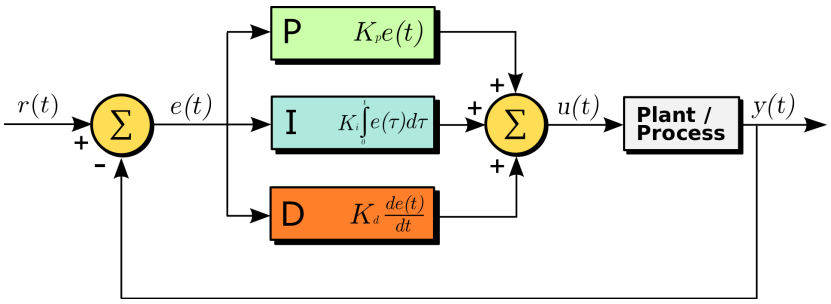


Figure K.3: Generalized architecture of a PID controller.

K.1.1 P - Proportional Term

The proportional term of the PID Controller multiplies the error signal by a set gain to generate a control signal. In general, the larger the proportional gain, the faster the control system response. However, too large of a proportional gain may cause the system to become unstable. Additionally, too small of a proportional gain will cause the control system response to be too slow to allow the system to react to disturbances, and in an unstable system, it will cause the system to diverge away from the reference signal. As the proportional term requires some error signal to drive the controller, a

proportional term alone will usually cause some steady-state error.

K.1.2 I - Integral Term

The integral term of the PID controller multiplies the integral of the error signal by a set gain to generate a control signal. Whereas the proportional term can is used to generate a rapid response in the system, the integral term is used to reduce or eliminate the steady-state error. This capability is due to the fact that the integral term is reliant on both the magnitude *and* duration of the error. However, because the integral term responds to the accumulation of the error in time, it can cause the system to overshoot the set-point, increase the transient response. As well, in reaction to large enough changes in the set-point, the integral term may cause *integral windup*.

Integral Windup

Integral windup is a phenomenon which occurs in physical systems due to the ideal output undergoing saturation. This causes the integral of the error signal to accumulate a larger error during the rise and will lead to an overshooting of the set-point. There are numerous methods [38, 39] to mitigate the integral windup such as choosing to give the set-point an appropriate ramp up to the desired value, conditionally integrating the error, or zeroing the integral value when the error is equal to zero.

K.1.3 D - Derivative Term

The derivative term of the PID controller multiplies the derivative of the error signal by a set gain to generate a control signal. The derivative term is used to "predict" the system behavior, and improves the settling time and stability of the system. It must be noted however, that an ideal derivative is non-causal, and as such cannot be implemented in reality. Approaches to compensate for this and allow for a derivative term to be used in the controller are varied, but include adding a pole 10 to 100 times further from the origin than then zero, and taking the derivative from the feedback path.In the laplace domain, rather than the derivative taking the form of s, a pseudo-derivative takes the form of

$$\frac{s}{\tau s + 1}. \tag{K.3}$$

K.2 Full State Feedback

A full state feedback (FSF) controller [40] is a linear controller, which uses the state vector to compute the control signal for the system. This method of control relies on the placement of closed-loop poles of the system in predetermined locations in the $s-$plane. Direct placement of the poles is advantageous because the location of the poles directly correspond to the stability and performance of the closed-loop system. The system must, of course, be considered controllable in order to implement this method. The full

state feedback controller is a proportional controller which works over the state vector χ,

$$\mathbf{u} = \mathrm{r} - \mathrm{K}\chi, \tag{K.4}$$

where $\mathbf{u}$ is the control signal, K is an appropriately sized gain matrix, and r is the reference signal. Substituting (K.4) into (4.42) we can derive the closed-loop system with full state feedback as

$$\dot{\chi} = (\mathbf{A} - \mathbf{B}\mathrm{K})\chi + \mathbf{B}\mathrm{r}. \tag{K.5}$$

The goal is to choose the values of K such that $(\mathbf{A} - \mathbf{B}\mathrm{K})$ has the desired properties. This may be accomplished by taking the characteristic equation of the closed loop system

$$\det(sI - (\mathbf{A} - \mathbf{B}\mathrm{K})), \tag{K.6}$$

and setting the values of K such that the value of (K.6) is equal to the desired characteristic equation. This method is called *pole placement*.

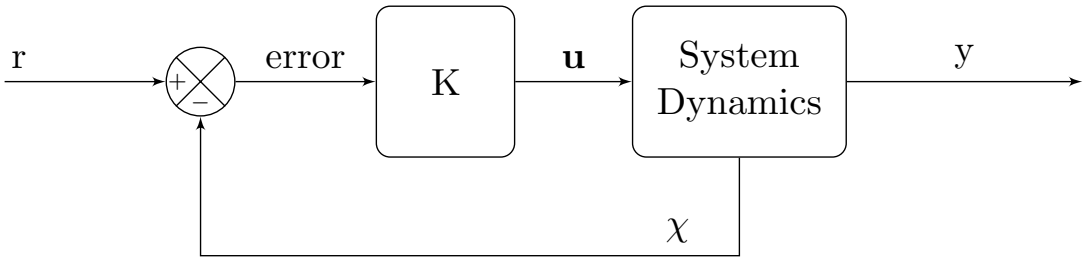


Figure K.4: Block diagram of a state feedback controller.

K.2.1 Linear Quadratic Regulator

A Linear Quadratic Regulator (LQR) is an *optimal* controller [41]. The LQR controller is a special case of a full state feedback controller, where the gain matrix K is optimized to give the desired performance at a desired control effort. The LQR works by minimizing the function

$$J = \int_0^\infty (\chi^{\mathrm{T}}\mathrm{Q}\mathrm{x} + \mathrm{u}^{\mathrm{T}}\mathrm{R}\mathrm{u})dt, \tag{K.7}$$

where Q is a positive semi-definite matrix and R is positive definite. This cost function represents that trade-off between the error and the cost of the control input. By choosing the matrices Q and R we can balance the rate of convergence of the solutions with the cost of the control effort. The solution to the LQR problem is given by,

$$\mathbf{u} = -\mathrm{R}^{-1}\mathbf{B}^{\mathrm{T}}\mathrm{P}\chi, \tag{K.8}$$

where $P \in \mathbb{R}^{n \times n}$ is a positive-definite, symmetric matrix that satisfies the algebraic Riccati equation,

$$PA + A^T P - PBR^{-1}B^T P + Q = 0. \tag{K.9}$$

It must be noted that in order to guarantee that a solution exists, we require that $Q \geq 0$ and $R > 0$. Choosing the specific values for Q and R is dependent on the system we are trying to control. The values of the two matrices determined how much the value of each state or input (squared) affects the overall cost. States and/or inputs that we require to be very small may have large weights attached to them, harshly penalizing them, whereas states or inputs that we don't have such a requirement on may have lower weights penalizing them less. Of note, the LQR problem may be modified to handle the reference tracking problem as well [42, 43]

K.3 Cascaded Control System

Cascaded control involves a mutli-loop control structure where the outer-loops provide the set-points for the inner-loops in a cascading manner from outer-most loop to the inner-most loop. The feedback for each controller is nested inside each controller above it in the cascade (Fig. K.5). Such a control system provides an improved response to

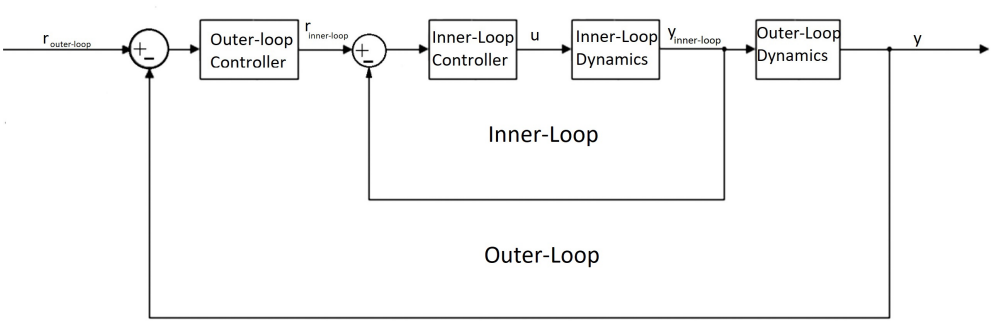


Figure K.5: Generalized block diagram of a cascaded control system.

disturbances when compared to a singular control loop. A cascaded controller is useful in situations where the inner-loop process has significantly faster dynamics than the outer-loop process. If the inner-loop is not sufficiently faster in response compared to the outer-loop, there is a risk of interaction between the loops which can lead to the instability of the system. Additionally, it is worth noting that the cascaded controller requires additional measurements to allow each level of the cascaded controller to function, and the control architecture is more complex than a singe controller. For quadcopters, as a general rule, the attitude and height controllers are viewed as the inner-loop controller, with the lateral position controller being the outer-loop controller (Figure K.6). Additionally, in some cases, there may be another layer to the cascaded controller with the attitude controllers feeding the set point to the angular rate controllers. It is worth noting that a cascading PID controller is the most popular control system

design for quadcopters in the industry, primarily due to its simplicity and relative ease of tuning.

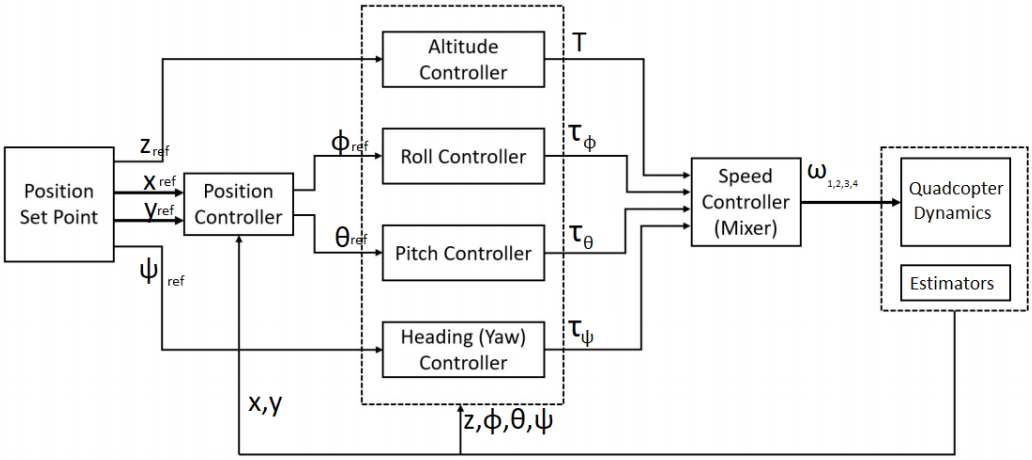


Figure K.6: Cascased control system for a quadcopter.

Appendix L

Control Systems Code

This appendix contains a number of code snippets required for the implementation of various controllers and control systems into both the simulator environment and the flight code running the quadcopter.

Snippets 34 and 35 provide the code required to implement a PID based control system which controls the inner-loop dynamics of the quadcopter in the simulator environment.

```
function [u, comp_ref, comp_error] = InnerLoop_PID(dt, state, ref , Sensors)
%% Persistent Variables
persistent prev_state prev_ref prev_error

if isempty(prev_state)
    prev_state.x         = 0;   % [m]
    prev_state.y         = 0;   % [m]
    prev_state.z         = 0;   % [m]
    prev_state.phi       = 0;   % [deg]
    prev_state.theta     = 0;   % [deg]
    prev_state.psi       = 0;   % [deg]
    prev_state.dx        = 0;   % [m/s]
    prev_state.dy        = 0;   % [m/s]
    prev_state.dz        = 0;   % [m/s]
    prev_state.dphi      = 0;   % [rad/sec]
    prev_state.dtheta    = 0;   % [rad/sec]
    prev_state.dpsi      = 0;   % [rad/sec]
end
if isempty(prev_ref)
    prev_ref.x           = 0;   % [m]
    prev_ref.y           = 0;   % [m]
    prev_ref.z           = 0;   % [m]
    prev_ref.phi         = 0;   % [deg]
    prev_ref.theta       = 0;   % [deg]
    prev_ref.psi         = 0;   % [deg]
    prev_ref.dx          = 0;   % [m/s]
    prev_ref.dy          = 0;   % [m/s]
    prev_ref.dz          = 0;   % [m/s]
    prev_ref.dphi        = 0;   % [rad/sec]
    prev_ref.dtheta      = 0;   % [rad/sec]
    prev_ref.dpsi        = 0;   % [rad/sec]
end
```

Snippet 34: Inner-Loop PID control system for the flight simulator: Part 1.

Snippets 36 and 37 provide the code required to implement an LQR based control system which controls the inner-loop dynamics of the quadcopter in the simulator environment.

Snippets 38 and 39 provide the code required to implement an PID based cascaded control system which controls the dynamics of the quadcopter in the simulator environment.

Snippets 40 and 41 provide the code required to implement a full state LQR control system which controls the dynamics of the quadcopter in the simulator environment.

Snippet 42 provides the code required to implement a general PID class in the flight code. When inputting an error signal, a single error signal can be input, or an error signal and its associated derivative taken from the state estimator may be input as well. This allows the controller to either determine the derivative of the error signal internally, or use a potentially more accurate sensor estimate to do so.

Snippet 43 provides the code required to implement a general full state feedback class in the flight code. This class can be used to develop other controllers and control systems.

Snippet 44 provides the code required to implement an PID based cascaded control system which controls the dynamics of the quadcopter in the flight code.

Snippet 45 provides the code required to implement an LQR based control system which controls the inner-loop dynamics of the quadcopter in the flight code.

Snippet 46 provides the code required to implement an LQR based control system which controls the dynamics of the quadcopter in the flight code.

```
if isempty(prev_error)
    prev_error.x         = 0;   % [m]
    prev_error.y         = 0;   % [m]
    prev_error.z         = 0;   % [m]
    prev_error.phi       = 0;   % [deg]
    prev_error.theta     = 0;   % [deg]
    prev_error.psi       = 0;   % [deg]
    prev_error.dx        = 0;   % [m/s]
    prev_error.dy        = 0;   % [m/s]
    prev_error.dz        = 0;   % [m/s]
    prev_error.dphi      = 0;   % [rad/sec]
    prev_error.dtheta    = 0;   % [rad/sec]
    prev_error.dpsi      = 0;   % [rad/sec]
end

%% Required Values
g = 9.81;
m = 0.53857;

%% Define Error Terms
error = structminus(ref,state);

%% Define Controllers
%% Inner Loop Controllers
% Altitude Controller
C_alt.kp = 0.5;
C_alt.ki = 0;
C_alt.kd = 0.35;

% Roll Controller
C_r.kp = 0.065;
C_r.ki = 0.0005;
C_r.kd = 0.01;

% Pitch Controller
C_p.kp = 0.055;
C_p.ki = 0.0003;
C_p.kd = 0.0085;

% Yaw Controller
C_y.kp = 0.005;
C_y.ki = 0;
C_y.kd = 0.015;
%% Controller Output

% Control Output
u.u1 = ( C_alt.kp*error.z   + C_alt.ki*(prev_error.z+error.z*dt)       +...
        C_alt.kd*(error.z-prev_error.z)/dt       );  % Thrust
u.u2 = ( C_r.kp*error.phi   + C_r.ki*(prev_error.phi+error.phi*dt)     +...
        C_r.kd*(error.phi-prev_error.phi)/dt     );  % Roll
u.u3 = ( C_p.kp*error.theta + C_p.ki*(prev_error.theta+error.theta*dt) +...
        C_p.kd*(error.theta-prev_error.theta)/dt );  % Pitch
u.u4 = ( C_y.kp*error.psi   + C_y.ki*(prev_error.psi+error.psi*dt)     +...
        C_y.kd*(error.psi-prev_error.psi)/dt     );  % Yaw

% Computed Reference Signals
comp_ref = ref;

% Computed Error Signals
comp_error = error;

prev_state = state;
prev_ref = ref;
prev_error = error;
end
```

Snippet 35: Inner-Loop PID control system for the flight simulator: Part 2.

```
function [u, comp_ref, comp_error] = LQR_Controller_InnerOnly(dt, state, ref , Sensors)
%% Persistent Variables
persistent prev_state prev_ref prev_error

if isempty(prev_state)
    prev_state.x         = 0;   % [m]
    prev_state.y         = 0;   % [m]
    prev_state.z         = 0;   % [m]
    prev_state.phi       = 0;   % [deg]
    prev_state.theta     = 0;   % [deg]
    prev_state.psi       = 0;   % [deg]
    prev_state.dx        = 0;   % [m/s]
    prev_state.dy        = 0;   % [m/s]
    prev_state.dz        = 0;   % [m/s]
    prev_state.dphi      = 0;   % [rad/sec]
    prev_state.dtheta    = 0;   % [rad/sec]
    prev_state.dpsi      = 0;   % [rad/sec]
end
if isempty(prev_ref)
    prev_ref.x           = 0;   % [m]
    prev_ref.y           = 0;   % [m]
    prev_ref.z           = 0;   % [m]
    prev_ref.phi         = 0;   % [deg]
    prev_ref.theta       = 0;   % [deg]
    prev_ref.psi         = 0;   % [deg]
    prev_ref.dx          = 0;   % [m/s]
    prev_ref.dy          = 0;   % [m/s]
    prev_ref.dz          = 0;   % [m/s]
    prev_ref.dphi        = 0;   % [rad/sec]
    prev_ref.dtheta      = 0;   % [rad/sec]
    prev_ref.dpsi        = 0;   % [rad/sec]
end
if isempty(prev_error)
    prev_error.x         = 0;   % [m]
    prev_error.y         = 0;   % [m]
    prev_error.z         = 0;   % [m]
    prev_error.phi       = 0;   % [deg]
    prev_error.theta     = 0;   % [deg]
    prev_error.psi       = 0;   % [deg]
    prev_error.dx        = 0;   % [m/s]
    prev_error.dy        = 0;   % [m/s]
    prev_error.dz        = 0;   % [m/s]
    prev_error.dphi      = 0;   % [rad/sec]
    prev_error.dtheta    = 0;   % [rad/sec]
    prev_error.dpsi      = 0;   % [rad/sec]
end

%% Required Values

%% Define Error Terms
error = structminus(ref,state);

error_innerOnly.z       = error.z;
error_innerOnly.phi     = error.phi;
error_innerOnly.theta   = error.theta;
error_innerOnly.psi     = error.psi;
error_innerOnly.dz      = error.dz;
error_innerOnly.dphi    = error.dphi;
error_innerOnly.dtheta  = error.dtheta;
error_innerOnly.dpsi    = error.dpsi;
```

Snippet 36: Inner-Loop LQR control system for the flight simulator: Part 1.

```
%% Define Controllers
K = [1.7768         0         0         0    1.4991         0         0         0
          0    0.1392         0         0         0    0.1133         0         0
          0         0    0.1493         0         0         0    0.1227         0
          0         0         0    0.1383         0         0         0    0.1125
          ];
C = K*structarray(error_innerOnly);

%% Controller Output

% Control Output
u.u1 = C(1);  % Thrust
u.u2 = C(2);  % Roll
u.u3 = C(3);  % Pitch
u.u4 = C(4);  % Yaw

% Computed Reference Signals
comp_ref = ref;

% Computed Error Signals
comp_error = error;

prev_state = state;
prev_ref = ref;
prev_error = error;
end
```

Snippet 37: Inner-Loop LQR control system for the flight simulator: Part 2.

```matlab
function [u, comp_ref, comp_error] = CascadedPID(dt, state, ref , Sensors)
%% Persistent Variables
persistent prev_state prev_ref prev_error

if isempty(prev_state)
    prev_state.x         = 0;   % [m]
    prev_state.y         = 0;   % [m]
    prev_state.z         = 0;   % [m]
    prev_state.phi       = 0;   % [deg]
    prev_state.theta     = 0;   % [deg]
    prev_state.psi       = 0;   % [deg]
    prev_state.dx        = 0;   % [m/s]
    prev_state.dy        = 0;   % [m/s]
    prev_state.dz        = 0;   % [m/s]
    prev_state.dphi      = 0;   % [rad/sec]
    prev_state.dtheta    = 0;   % [rad/sec]
    prev_state.dpsi      = 0;   % [rad/sec]
end
if isempty(prev_ref)
    prev_ref.x           = 0;   % [m]
    prev_ref.y           = 0;   % [m]
    prev_ref.z           = 0;   % [m]
    prev_ref.phi         = 0;   % [deg]
    prev_ref.theta       = 0;   % [deg]
    prev_ref.psi         = 0;   % [deg]
    prev_ref.dx          = 0;   % [m/s]
    prev_ref.dy          = 0;   % [m/s]
    prev_ref.dz          = 0;   % [m/s]
    prev_ref.dphi        = 0;   % [rad/sec]
    prev_ref.dtheta      = 0;   % [rad/sec]
    prev_ref.dpsi        = 0;   % [rad/sec]
end
if isempty(prev_error)
    prev_error.x         = 0;   % [m]
    prev_error.y         = 0;   % [m]
    prev_error.z         = 0;   % [m]
    prev_error.phi       = 0;   % [deg]
    prev_error.theta     = 0;   % [deg]
    prev_error.psi       = 0;   % [deg]
    prev_error.dx        = 0;   % [m/s]
    prev_error.dy        = 0;   % [m/s]
    prev_error.dz        = 0;   % [m/s]
    prev_error.dphi      = 0;   % [rad/sec]
    prev_error.dtheta    = 0;   % [rad/sec]
    prev_error.dpsi      = 0;   % [rad/sec]
end

%% Required Values
g = 9.81;
m = 0.53857;

%% Define Error Terms
error = structminus(ref,state);

%% Define Controllers
%% Inner Loop Controllers
% Altitude Controller
C_alt.kp = 1;
C_alt.ki = 0;
C_alt.kd = 1;

% Roll Controller
C_r.kp = 0.1023;
C_r.ki = 0;
C_r.kd = 0.02047;
```

Snippet 38: Cascaded PID control system for the flight simulator: Part 1.

```
% Pitch Controller
C_p.kp = 0.1423;
C_p.ki = 0;
C_p.kd = 0.02846;
% Yaw Controller
C_y.kp = 0.004;
C_y.ki = 0;
C_y.kd = 0.012;

%% Outer Loop Controllers
% X Positional Controller
% C_X is positive here instead of C_Y. Unsure why.
C_X.kp           = 0.24;
C_X.ki           = 0;
C_X.kd           = 0.1;

% Set Outer Loop Controller Output as Inner Loop Controller Reference
ref.theta        = (C_X.kp*error.x + C_X.ki*(prev_error.x+error.x*dt) +...
                    C_X.kd*((error.x-prev_error.x)/dt));
% Ensure the reference angle is wrapped to pi
ref.theta        = (mod((ref.theta+pi),(2*pi))-pi);
% Define Error Signal based off of new Reference Signal
error.theta      = ref.theta-state.theta;
% Wrap error angle to pi
error.theta      = (mod((error.theta+pi),(2*pi))-pi);

% Y Positional Controller
% NOTE: C_Y is negative here instead of C_X. Unsure why
C_Y.kp          = -0.24;
C_Y.ki          = 0;
C_Y.kd          = -0.1;

% Set Outer Loop Controller Output as Inner Loop Controller Reference
ref.phi        = (C_Y.kp*error.y + C_Y.ki*(prev_error.y+error.y*dt) +...
                  C_Y.kd*((error.y-prev_error.y)/dt));
% Ensure the reference angle is wrapped to pi
ref.phi        = (mod((ref.phi+pi),(2*pi))-pi);

% Define Error Signal based off of new Reference Signal
error.phi      = ref.phi-state.phi;
% Wrap error angle to pi
error.phi      = (mod((error.phi+pi),(2*pi))-pi);


%% Controller Output
% Control Output
u.u1 = ( C_alt.kp*error.z   + C_alt.ki*(prev_error.z+error.z*dt)       +...
        C_alt.kd*(error.z-prev_error.z)/dt       );  % Thrust
u.u2 = ( C_r.kp*error.phi   + C_r.ki*(prev_error.phi+error.phi*dt)     +...
        C_r.kd*(error.phi-prev_error.phi)/dt     );  % Roll
u.u3 = ( C_p.kp*error.theta + C_p.ki*(prev_error.theta+error.theta*dt) +...
        C_p.kd*(error.theta-prev_error.theta)/dt );  % Pitch
u.u4 = ( C_y.kp*error.psi   + C_y.ki*(prev_error.psi+error.psi*dt)     +...
        C_y.kd*(error.psi-prev_error.psi)/dt     );  % Yaw

% Computed Reference Signals
comp_ref = ref;
% Computed Error Signals
comp_error = error;

prev_state = state;
prev_ref = ref;
prev_error = error;
end
```

Snippet 39: Cascaded PID control system for the flight simulator: Part 2.

```
function [u, comp_ref, comp_error] = LQR_Controller(dt, state, ref , Sensors)
%% Persistent Variables
persistent prev_state prev_ref prev_error %prev_comp_error

if isempty(prev_state)
    prev_state.x         = 0;   % [m]
    prev_state.y         = 0;   % [m]
    prev_state.z         = 0;   % [m]
    prev_state.phi       = 0;   % [deg]
    prev_state.theta     = 0;   % [deg]
    prev_state.psi       = 0;   % [deg]
    prev_state.dx        = 0;   % [m/s]
    prev_state.dy        = 0;   % [m/s]
    prev_state.dz        = 0;   % [m/s]
    prev_state.dphi      = 0;   % [rad/sec]
    prev_state.dtheta    = 0;   % [rad/sec]
    prev_state.dpsi      = 0;   % [rad/sec]
end
if isempty(prev_ref)
    prev_ref.x           = 0;   % [m]
    prev_ref.y           = 0;   % [m]
    prev_ref.z           = 0;   % [m]
    prev_ref.phi         = 0;   % [deg]
    prev_ref.theta       = 0;   % [deg]
    prev_ref.psi         = 0;   % [deg]
    prev_ref.dx          = 0;   % [m/s]
    prev_ref.dy          = 0;   % [m/s]
    prev_ref.dz          = 0;   % [m/s]
    prev_ref.dphi        = 0;   % [rad/sec]
    prev_ref.dtheta      = 0;   % [rad/sec]
    prev_ref.dpsi        = 0;   % [rad/sec]
end
if isempty(prev_error)
    prev_error.x         = 0;   % [m]
    prev_error.y         = 0;   % [m]
    prev_error.z         = 0;   % [m]
    prev_error.phi       = 0;   % [deg]
    prev_error.theta     = 0;   % [deg]
    prev_error.psi       = 0;   % [deg]
    prev_error.dx        = 0;   % [m/s]
    prev_error.dy        = 0;   % [m/s]
    prev_error.dz        = 0;   % [m/s]
    prev_error.dphi      = 0;   % [rad/sec]
    prev_error.dtheta    = 0;   % [rad/sec]
    prev_error.dpsi      = 0;   % [rad/sec]
end

%% Required Values
```

Snippet 40: Full state LQR control system for the flight simulator: Part 1.

```
%% Define Error Terms
error = structminus(ref,state);

%% Define Controllers
% Thesis Drone
K = [    0    0    0.9370    0    0    0    0    0    1.0051    0    0    0
    0   -0.0041    0    0.0511    0    0    0   -0.0072    0    0.0181    0    0
    0.0042    0    0    0    0.0567    0    0.0075    0    0    0    0.0221    0
    0    0    0    0    0    0.0157    0    0    0    0    0    0.0126
    ];
C = K*structarray(error);

%% Controller Output

% Control Output
u.u1 = C(1);  % Thrust
u.u2 = C(2);  % Roll
u.u3 = C(3);  % Pitch
u.u4 = C(4);  % Yaw

% Computed Reference Signals
comp_ref = ref;

% Computed Error Signals
comp_error = error;

prev_state = state;
prev_ref = ref;
prev_error = error;
end
```

Snippet 41: Full state LQR control system for the flight simulator: Part 2.

```cpp
// PID Controller
class PID {
    private:
    float Kp, Ki, Kd;
    float integral, prev_error;

    public:
    PID(){};
    PID(float Kp, float Ki, float Kd);
    float update(float error);
    float update(float error, float derivative);
};

PID::PID(float Kp, float Ki, float Kd) {
    this->Kp = Kp;
    this->Kd = Kd;
    this->Ki = Ki;
    integral = 0;
    prev_error = 0;
}

float PID::update(float error) {
    integral += error * dt;
    float derivative = (error - prev_error) / dt;
    prev_error = error;
    float command = error * Kp + integral * Ki + derivative * Kd;
    return command;
}

float PID::update(float error, float derivative) {
    integral += error * dt;
    prev_error = error;
    float command = error * Kp + integral * Ki - derivative * Kd;
    // negative state derivative = erorr derivative for a constant reference signal
    return command;
}
```

Snippet 42: PID controller for the flight code.

```cpp
// Full State Feedback Controller
template <int num_states, int num_commands>
class FSF {
    private:
    public:
    FSF<num_states, num_commands>();
    BLA::Matrix<num_commands, num_states> K; // Gain Matrix
    BLA::Matrix<num_states, 1> x;            // State Vector
    BLA::Matrix<num_states, 1> r;            // Reference Signal Vector
    BLA::Matrix<num_commands, 1> u;          // Command Vector
    void update(BLA::Matrix<num_states, 1> r, BLA::Matrix<num_states, 1> x);
};

template <int num_states, int num_commands>
FSF<num_states, num_commands>::FSF() {
    this->K.Fill(0.0);
    this->r.Fill(0.0);
    this->x.Fill(0.0);
    this->u.Fill(0.0);
};

template <int num_states, int num_commands>
void FSF<num_states, num_commands>::update(BLA::Matrix<num_states, 1> r,
                                           BLA::Matrix<num_states, 1> x) {
    // this->K = K;
    this->x = x;
    this->r = r;

    this->u = K * (r - x);
};
```

Snippet 43: Full State Feedback Controller for the flight code.

```
class CascadingController : public ControlSystem {
    private:
    struct pids {
        PID x, y, pitch, roll, yaw, thrust;
    } pids;

    public:
    CascadingController();
    void Control(StateVector &current_state, StateVector &desired_state,
                 MomentThrustCommand &output_command) override;
};

CascadingController::CascadingController() {
    pids.x = PID(1, 0, 1);
    pids.y = PID(1, 0, 1);
    pids.thrust = PID(0.845, 0, 0.796);
    pids.roll = PID(0.05, 0, 0.01);
    pids.pitch =  PID(0.08, 0, 0.01);
    pids.yaw = PID(0.001, 0, 0.0001);
}

void CascadingController::Control(StateVector &current_state, StateVector &desired_state,
                                  MomentThrustCommand &output_command) {

    float pitch = pids.x.update(desired_state.x - current_state.x);
    float roll = pids.y.update(desired_state.y - current_state.y);

    float new_pitch = pids.pitch.update(pitch - current_state.pitch);
    float new_roll = pids.roll.update(roll - current_state.roll);
    float new_yaw = pids.yaw.update(desired_state.yaw - current_state.yaw);
    float new_thrust = pids.thrust.update(desired_state.z - current_state.z) + mass;
    output_command.pitch = new_pitch;
    output_command.roll = new_roll;
    output_command.yaw = new_yaw;
    output_command.thrust = new_thrust;
}
```

Snippet 44: Cascaded PID control system for the flight code.

```
class LQR_Controller : public ControlSystem {
    private:
    BLA::Matrix<12, 1> x;
    BLA::Matrix<12, 1> r;
    BLA::Matrix<2> obs;
    FSF<12, 4> LQR;

    public:
    LQR_Controller();
    void Control(StateVector &current_state, StateVector &desired_state,
                 MomentThrustCommand &output_command) override;
    // ~LQR_Controller();
};

LQR_Controller::LQR_Controller() {
    LQR.K = {
        0.0,      0.0,     4.4721, 0.0,      0.0,      0.0,
        0.0,      4.4721,  0.0,    102.7903, 0.0,      0.0,
        -4.4721,  0.0,     0.0,    0.0,      105.8913, 0.0,
        0.0,      0.0,     0.0,    0.0,      0.0,      14.1421,
    };
    LQR.x.Fill(0);
    LQR.r.Fill(0);
    LQR.u.Fill(0);
}

void LQR_Controller::Control(StateVector &current_state, StateVector &desired_state,
                             MomentThrustCommand &output_command) {
    pack_column_vector<12>(x, current_state);
    pack_column_vector<12>(r, desired_state);
    LQR.update(r, x);

    output_command.thrust = LQR.u(0) + mass;
    output_command.roll = LQR.u(1);
    output_command.pitch = LQR.u(2);
    output_command.yaw = LQR.u(3);
}
```

Snippet 45: Inner-Loop LQR control system for the flight code.

```cpp
class LQR_Controller : public ControlSystem {
    private:
    BLA::Matrix<12, 1> x;
    BLA::Matrix<12, 1> r;
    BLA::Matrix<2> obs;
    FSF<12, 4> LQR;

    public:
    LQR_Controller();
    void Control(StateVector &current_state, StateVector &desired_state,
                 MomentThrustCommand &output_command) override;
    // ~LQR_Controller();
};

LQR_Controller::LQR_Controller() {
    LQR.K = {
        0.0,      0.0,     4.4721, 0.0,      0.0,      0.0,
        0.0,      4.4721,  0.0,    102.7903, 0.0,      0.0,
        -4.4721,  0.0,     0.0,    0.0,      105.8913, 0.0,
        0.0,      0.0,     0.0,    0.0,      0.0,      14.1421,
    };
    LQR.x.Fill(0);
    LQR.r.Fill(0);
    LQR.u.Fill(0);
}

void LQR_Controller::Control(StateVector &current_state, StateVector &desired_state,
                             MomentThrustCommand &output_command) {
    pack_column_vector<12>(x, current_state);
    pack_column_vector<12>(r, desired_state);
    LQR.update(r, x);

    output_command.thrust = LQR.u(0) + mass;
    output_command.roll = LQR.u(1);
    output_command.pitch = LQR.u(2);
    output_command.yaw = LQR.u(3);
}
```

Snippet 46: LQR control system for the flight code.

Bibliography

[1] S. Dixon-Warren, "Motion sensing in the iphone 4: MEMS accelerometer," 2010, [Accessed 05-May-2020]. [Online]. Available: https://www.memsjournal.com/2010/12/motion-sensing-in-the-iphone-4-mems-accelerometer.html

[2] H. Wen, *Toward Inertial-Navigation-on-Chip: The Physics and Performance Scaling of Multi-Degree-of-Freedom Resonant MEMS Gyroscopes*. Springer International Publishing, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-25470-4_1

[3] D. Ren, L. Wu, M. Yan, M. Cui, Z. You, and M. Hu, "Design and analyses of a MEMS based resonant magnetometer," *Sensors (Basel, Switzerland)*, vol. 9, pp. 6951–66, 09 2009.

[4] "DJI p4 multispectral," DJI, 2021, [Accessed 17-March-2021]. [Online]. Available: https://www.dji.com/p4-multispectral

[5] "Skippy scout: Automated crop scouting system," DroneAG, 2020, [Accessed 17-March-2021]. [Online]. Available: https://droneag.farm/solutions/automated-crop-scouting-system/

[6] "e-agriculture in action: drones for agriculture," Food and Agriculture Organization of the United Nations, 2020, [Accessed 17-March-2021]. [Online]. Available: http://www.fao.org/3/i8494en/i8494en.pdf

[7] "Tzur: Idf's new drone," Ynet, 2016, [Accessed 17-March-2021]. [Online]. Available: https://www.ynetnews.com/articles/0,7340,L-4866692,00.html

[8] "Livesky sentry," Hoverfly Technologies, 2021, [Accessed 17-March-2021]. [Online]. Available: https://hoverflytech.com/livesky-sentry/

[9] "Flytrex delivery drone," Flytrex, 2021, [Accessed 17-March-2021]. [Online]. Available: https://www.flytrex.com/

[10] "Boeing: Cargo air vehicle completes first outdoor flight," Boeing, 2019, [Accessed 17-March-2021]. [Online]. Available: https://www.boeing.com/features/2019/05/cav-first-flight-05-19.page

[11] "Amazon prime air," Amazon, 2021, [Accessed 17-March-2021]. [Online]. Available: https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011

[12] M. Sørensen, M. Kjaergaard, and J. Bjørn, "Autonomous hover flight for a quad rotor helicopter: Linear quadratic controller, piecewise affine -hybrid systems controller," Master's thesis, Aalborg University, 2007.

[13] I. Kugelberg, "Black-box modeling and attitude control of a quadcopter," Master's thesis, Linköping University, 2016.

[14] "16.30 Feedback Control Systems," Massachusetts Institute of Technology, 2017, [Accessed 05-May-2020]. [Online]. Available: http://fast.scripts.mit.edu/dronecontrol/

[15] "085705 - Advanced Control Lab," Technion - Israel Institute of Technology, 2020, [Accessed 05-May-2020]. [Online]. Available: https://ug3.technion.ac.il/rishum/course/085705/202002

[16] *Arduino Nano 33 BLE Sense*, Arduino, 2020, [Accessed 06-May-2020]. [Online]. Available: https://store.arduino.cc/arduino-nano-33-ble-sense

[17] *iNEMO inertial module: 3D accelerometer, 3D gyroscope, 3D magnetometer*, STMicroelectronics, 2020, [Accessed 06-May-2020]. [Online]. Available: https://content.arduino.cc/assets/Nano_BLE_Sense_lsm9ds1.pdf

[18] J. Shabulinzenze, "Comparison of MEMS sensors in machine vibration monitoring," Master's thesis, Tampere University, 2020.

[19] "Cse176e/276e: Robotic system design and implementation," University of Califonia, San Diego, 2019, [Accessed 23-March-2021]. [Online]. Available: https://sites.google.com/a/eng.ucsd.edu/quadcopterclass/

[20] "CMPEN 473 Microcomputer Laboratory: Design of digital systems using microprocessors," Pennsylvania State University, 2017, [Accessed 23-March-2021]. [Online]. Available: http://www.cse.psu.edu/~kxc104/class/cmpen473/17s/

[21] *MEMS nano pressure sensor: 260-1260 hPa absolute digital output barometer*, STMicroelectronics, 2020, [Accessed 06-May-2020]. [Online]. Available: https://content.arduino.cc/assets/Nano_BLE_Sense_lps22hb.pdf

[22] *Ultrasonic Ranging Module HC - SR04*, Sparkfun Electronics, 2020, [Accessed 06-May-2020]. [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

[23] *PMW3901MB-TXQT: Optical Motion Tracking Chip*, PixArt Imaging Inc., 2020, [Accessed 06-May-2020]. [Online]. Available: https://wiki.bitcraze.io/_media/projects:crazyflie2:expansionboards:pot0189-pmw3901mb-txqt-ds-r1.00-200317_20170331160807_public.pdf

[24] M. V. Srinivasan, "How bees exploit optic flow: Behavioural experiments and neural models," *Philosophical Transactions: Biological Sciences*, vol. 337, no. 1281, pp. 253–259, 1992.

[25] ——, "An image-interpolation technique for the computation of optic flow and egomotion," *Biological Cybernetics*, vol. 71, pp. 401–415, 1994.

[26] T. S. R. Jerrold E. Marsden, *Wind Turbine Aerodynamics and Vorticity-Based Methods: Fundamentals and Recent Applications*. Springer International Publishing, 1999.

[27] M. Ben-Ari, "A tutorial on Euler angles and quaternions," 2017, [Accessed 06-May-2020]. [Online]. Available: https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/softwareAndLearningMaterials/quaternion-tutorial-2-0-1.pdf

[28] "Understanding Euler angles," CHRobotics, 2018, [Accessed 22-March-2021]. [Online]. Available: http://www.chrobotics.com/library/understanding-euler-angles

[29] E. Branlard, *Wind Turbine Aerodynamics and Vorticity-Based Methods: Fundamentals and Recent Applications*. Springer International Publishing, 2017.

[30] N. A. Shneydor, *Missile Guidance and Pursuit: Kinematics, Dynamics, and Control*. Horwood Publishing Limited, 1998.

[31] C. S. Draper, W. Wrigley, D. G. Hoag, R. H. Battin, J. E. Miller, D. A. Koso, A. L. Hopkins, and W. E. V. Velde, *Apollo Guidance and Navigation*. Massachusetts Institute of Technology, Instrumentation Laboratory, 1965. [Online]. Available: http://www.ibiblio.org/apollo/hrst/archive/1713.pdf

[32] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[33] S. A. E. H. Hegazy, A. M. Kamel, I. I. Arafa, and Y. Z. Elhalwagy, "MEMS gyro noise estimation and modeling for precise navigation simulation," in *2020 8th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, 2020, pp. 84–89.

[34] "Implementing a tilt-compensated ecompass using accelerometer and magnetometer sensors," Freescale Semiconductor, 17/03/2021, https://www.mikrocontroller.net/attachment/292888/AN4248.pdf.

[35] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 9th ed. USA: Prentice-Hall, Inc., 2000.

[36] Kiam Heong Ang, G. Chong, and Yun Li, "Pid control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology*, vol. 13, no. 4, pp. 559–576, 2005.

[37] Wikibooks, "Control systems — wikibooks, the free textbook project," 2020, [Online; accessed 31-March-2021]. [Online]. Available: https://en.wikibooks.org/w/index.php?title=Control_Systems&oldid=3793777

[38] H.-B. Shin and J.-G. Park, "Anti-windup PID controller with integral state predictor for variable-speed motor drives," *IEEE Transactions on Industrial Electronics*, vol. 59, pp. 1509–1516, 03 2012.

[39] A. Doroshenko, "Problems of modelling Proportional–Integral–Derivative controller in automated control systems," *MATEC Web of Conferences*, vol. 112, p. 05013, 01 2017.

[40] M. Fadali and A. Visioli, *Digital Control Engineering: Analysis and Design*, ser. Engineering professional collection. Academic Press, 2013. [Online]. Available: https://books.google.co.il/books?id=ooZHCxSUYjIC

[41] J. Z. Ben-Asher, *Optimal Control Theory with Aerospace Applications*. Cambridge University Press, 2010.

[42] J. F. Patarroyo-Montenegro, J. E. Salazar-Duque, and F. Andrade, "Lqr controller with optimal reference tracking for inverter-based generators on islanded-mode microgrids," in *2018 IEEE ANDESCON*, 2018, pp. 1–5.

[43] A. Budiyono and S. S. Wibowo, "Optimal tracking controller design for a small scale helicopter," *Journal of Bionic Engineering*, vol. 4, no. 4, pp. 271–280, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1672652907600419

עם מצבי קיצון ועוד. בנוסף, תבניות למשתמש גם נכללו על מנת ליעל את עבודת התכן.

תוכנת רישום נתונים בשידור חיי נכתבה ליצירת קשר דרך בלוטות' עם הרחפן. תוכנה זה מאפשרת מבט בזמן אמת על נתוני הרחפן, הפקודות של הבקרים למנועים, יציאות חוגי ההנחיה ,ניווט ובקרה ועוד. בנוסף התוכנה רושמת ושומרת את כל קריאות של המדידות מהסנסורים בקצב מוכתב מראש. תוכנת הרישום גם כן מאפשר שידור של פקודת עצירה מיידית לרחפן, למקרי חירום בזמן הטיסה.

הפלטפורמה השלמה משרתת כבסיס לתכן קצה לקצה כך שניתן לעבור משלב התכנון והפיתוח של מערכת חדשה ועד ליישום מעשי בפרקי זמן קצרים. התשתית מספקת מענה לפיתוח מודל מתמטי של המערכת הרצויה החדשה. לאחר פיתוח המודל, ניתן ליישמו בקלות בסביבת הסימולטור ולבדוקו בהרחבה. לאחר בדיקות מקיפות בתנאים מגוונים ומימוש בקרת הטיסה הרצויה, המערכת יכולה להיבדק על מערכת בדיקה פיזית במצבים נייחים וניידים כאחד באמצעות ציוד בדיקה מודפס בתלת מימד, זאת תוך כדי התבוננות בנתונים זמן אמת אשר שודרו מהרחפן. לאחר אימות המערכת, ניתן לבצע בדיקות טיסה בחינם עם הרחפן ובכך לסיים את תהליך התכן במידה והתוצאות מספקות.

תקציר

תזה זו מתארת את העבודה שנעשתה ליצירת פלטפורמת מחקר ותכן של מערכות הנחייה, ניווט, ובקרה. בפלטפורמה זו פותחה תשתית המשלבת חומרה ותוכנה בשיטה מודולרית בכדי לספק מגוון אפשרויות העומדות בפני תכנון החוקרים למערכות הנחייה, ניווט ובקרה. היישום העיקרי של פלטפורמה זו הינו לאפשר הרמת אב־טיפוס מהיר על מנת להפחית משמעותית בזמן ובעלות בפיתוח של המערכות אלו.

לצורך כך, רחפן מסוג קוודרוטור תוכנן בתור בסיס לפלטפורמה השלמה. הרחפן תוכנן כך שכל רכיביו ניתנים לרכישה מיידית מספקים קיימים, ובמחירים נוחים. בנוסף, סופקו קבצי תכן מכני על מנת לאפשר הדפסה במדפסת תלת מימדית של רוב הרכיבי הרחפן.

בוצע פיתוח של סביבת סימולציה מודולרית, מבוססת סימולינק, על מנת ליצור סביבת פיתוח פשוטה, יעילה ורב תכליתית עבור המתכנן. כחלק מסביבת סימולציה זו, נוצר סימולטור מודלורי הכולל בתוכו מודל מתמטי של הרחפן אשר פותח באמצעות חוקי ניוטון ואויילר. כחלק מהמודולריות קיים מודל לינארי מפושט נוח לתכן חוגי בקרה וניווט. בסימולטור זה, מתכנן יכול ליישם איזה חוג שירצה, ולשנותו בקלות, תוך כדי שינוי פרמטרים בסימולציה עצמה, כולל דינמיקת הרחפן, מצב טיסה (ריחוף, המראה) , מודל הסנסורים ואפיון רעשים, ועוד. חוגי הנחייה, ניווט, ובקרה ארוזים ובדוקים הוטמעו בסימולטור, כך שמתכנן יוכל לממש רק חוג הנחיה למשל ולעבוד עם חוגי בקרה וניווט בדוקים ומוכנים מראש.

פותח ציוד ומערכות בדיקה פיזיקליות בכדי לאפשר בדיקות של מנועים , בניית פרופילי דחף, בדיקות חוגי בקרה ובדיקת טיסה חופשית של הרחפן. ציוד זה כולל מערכת למדידת דחף של המנועים, בנפרד או ביחד, המבוססת על חלקים מודפסים שניתן לחבר אליהם מנוע אחד, או את הרחפן כולו עם ארבעת המנועים. בנוסף יוצר שולחן טיס 3 דרגות חופש המאפשר נעילת דרגות חופש של רחפן וזאת על מנת לאפשר בדיקות חוגי בקרה ,הנחיה וניווט, קבצי תכן מכני של מערכות אלו צורפו לעבודה זו כן שמתכנן מערכות יוכל להדפיס ציוד זה בהשקעת זמן ומשאבים מינימלית.

מערכת בקרת הטיסה קודדה בשפת C++ בצורה מודולרית גם כן. בקרת טיסה זו נבנתה כך שתוכל להיות בסיס פיתוחי למערכות מסוג זה. ניתן להפוך את הקוד לקוד גלוי" ולפתוח את הגישה לקוד לכל מי שירצה ליישם אותו. בדומה לסימולטור, גם כאן קיימות הספריות רבות אשר נכתבו ונארזו במערכת בקרת הטיסה לשימושו של המתכנן. ספריות אלו כוללות ספריות של בקרים, משערכי מצב, חוגי הנחייה, לוגיקת טיסה והתמודדות

המחקר בוצע בהנחייתו של פרופסור דניאל זלזו, בפקולטה להנדסת אווירונאוטיקה וחלל.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

פלטפורמה אווירית בעלות נמוכה לתכן מערכות הנחייה, ניווט, ובקרה

חיבור על פרויקט

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת אווירונאוטיקה וחלל

נתנאל דרעליך

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
ניסן תשפ"א חיפה מרץ 2021

פלטפורמה אווירית בעלות נמוכה לתכן מערכות הנחייה, ניווט, ובקרה

נתנאל דרעליך