Design of an Adaptive 3-Dimensional Display Enabled by a Swarm of Autonomous Micro Air Vehicles

by

Erich Mueller

B.S. Aeronautical and Astronautical Engineering
Massachusetts Institute of Technology, 2009

SUBMITTED TO THE DEPARTMENT OF AERONAUTICAL AND ASTRONAUTICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICAL AND ASTRONAUTICAL ENGINEERING AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

FEBRUARY 2012

Signature of Author: ______________________________

Department of Aeronautical and Astronautical Engineering
January 26, 2012

Certified by: ______________________________ [illegible]

Emilio Frazzoli
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: ______________________________ [illegible]

Eytan H. Modiano
Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Design of an Adaptive 3-Dimensional Display Enabled by a Swarm of Autonomous Micro Air Vehicles

by

Erich Mueller

Submitted to the Department of Aeronautical and Astronautical Engineering on January 26, 2012 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Aeronautical and Astronautical Engineering

ABSTRACT

This thesis is motivated by the concept of a system consisting of a swarm of small, automatically controlled air vehicles, each carrying a colour-controlled light source (payload), capable of executing coordinated maneouvres for the purpose of entertainment or data visualization. It focuses on a number of issues associated with the implementation of such a system, and specifically on the development of a non-linear, robust controller with feed-forward compensation, designed for control of a quad-rotor helicopter in the presence of aerodynamic interference generated by other vehicles in swarm-like operational conditions. System requirements and driving factors are identified, and various types of overall system architectures as well as vehicle designs are considered. The vehicle prototyping process is described, with specific emphasis on electronic design and hardware integration. Several problems associated with the flight of numerous vehicles in close proximity are addressed. Simulation and laboratory testing is described and results are presented and interpreted. Finally, consideration is given to the future development of such a system, as well as possible large-scale implementations and commercial opportunities.

Thesis Supervisor: Emilio Frazzoli
Title: Associate Professor of Aeronautics and Astronautics

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The type of system described in this thesis, known as Flyfire, was originally conceived by Carlo Ratti of MIT's Senseable City Laboratory as an adaptive 3-dimensional light display enabled by a swarm of small air vehicles carrying coloured light sources that can be controlled to follow a predetermined set of trajectories in space and colour which constitute the desired performance. Originally, the system was envisioned as consisting of up to 500 vehicles controlled in an indoor, windless environment, with minimum inter-vehicle spacing of approximately 50cm. No specific requirements on maneouvre bandwidth or accuracy were given, beyond those implied by common-sense performance expectations and non-collision conditions. These general requirements on the capabilities of the system were used to drive the development of both the overall system architecture and the individual vehicle design, with the expectation of earlier implementations utilizing significantly fewer vehicles.

1.1 Motivation

As envisioned, a system such as this would be used primarily for entertainment or artistic purposes. Originally conceived as a museum installation in which the reference trajectories of the vehicles would be generated to represent some data set relevant to the work done at the Senseable City Lab., considerable interest in the project was shown from the commercial sector, and specifically from companies intending to use the system for advertisement, such as in the production of a television commercial, as well as from entertainment groups intending to incorporate the system into existing live entertainment performances. While specific offers or companies should not be discussed in this document, interest in the project would strongly suggest that a concerted effort to develop such a system would be a financially viable investment.

Additionally, the effects of aerodynamic interference between helicopters flying in tight formations has not been well studied, and to the author's knowledge, development and testing of a feedback position controller for a swarm vehicle subject to aerodynamic interference, which accounts for other vehicles in the swarm with the explicit goal of rejecting wake related disturbances would be a valuable contribution from a scientific perspective. The implementation of such a system would enable the testing of algorithms for accurate trajectory tracking, collision avoidance, synchronization of trajectories, simultaneous state estimation, and macroscopically defined swarm control.

Though a number of general purpose flight testing platforms exist, this research also drives the development of a small, inexpensive vehicle capable of operating, through some overall system architecture, as part of a swarm, and such a vehicle could potentially serve as a multi-purpose platform for testing of various algorithms related to control, trajectory tracking and estimation.

1.2 Existing Technology

A number of systems exist which provide the capability of a adaptive, three-dimensional display. The Kinetic Sculpture [13], created by BMW, consists of an array of spheres, with each ball supported from above by a single cable, which may be released or retracted to manipulate the position of the ball. While the balls are not equipped with luminous elements of any kind, the system is capable of creating time-varying three dimensional forms of a type that have altitude uniquely defined by lateral coordinates, as in $z = f(x, y)$. Unlike the system described in this document, the spheres are restricted to motion along a vertical axis, and are therefore limited in the types of shapes they can create, as well as the overall volume of the space in which they can operate. Other existing three-dimensional displays such as the Nova [6], made by Creative Technology Germany, consist of a three dimensional array of LEDs fixed in space, each of which can be colour controlled to produce low-resolution moving forms similar to those that could be produced by the Flyfire system, without the additional capability of manipulating the locations of the colour points in space.

A significant amount of research has been committed to the development of small, autonomous air vehicles intended primarily for flight in indoor conditions with minimal environmental disturbances such as wind, as well as control and planning algorithms used to achieve accurate tracking of complex trajectories and execution of aggressive flight maneouvres for such vehicles [16], [19],[27]. Additionally, several implementations are known which enable the simultaneous flight of a number of small air vehicles in relatively close proximity [33],[8]. Much work has also been done developing controller for coordinated operation of other types of vehicles, including spacecraft [12], ground vehicles [23] and fixed wing aircraft in outdoor environments [14]. A large amount of other research has been done, and technology developed, which is relevant to the development of this system, and will be noted in subsequent sections when appropriate.

Chapter 2

System Architecture Considerations and Design

A number of technologies exist which would allow for different overall architectures and implementations of such a system. This chapter attempts to enumerate and assess the advantages and disadvantages of various implementation possibilities. System design options are described and considered, and choices are compared based on a number of criteria including cost, robustness, scalability and performance capabilities. Possible implementations can be roughly characterized by the degree of on-board autonomy of the vehicles, ranging from full on-board position feedback control to full off-board sensing with the vehicles themselves lacking sensors of any kind. In general the sensor and estimation architecture dictates requirements for the communication architecture. On-board position estimation may be enabled by a number of different sensor architectures, most of which rely on the sensing of signals of some kind from external beacons or markers placed in or around the operational volume. Existing systems such as Locata [4] utilize emulated GPS signals with modified commercial off-the-shelf GPS receiver modules, while others such as Cricket [24], or that used in the MIT Space Systems Laboratory for SPHERES [26], rely on ultrasonic time-of-flight (TOF) measurements from a number of optically synchronized ultrasonic transmitter beacons to make position and attitude measurements. A number of position control implementations have utilized on-board visual spectrum or IR cameras, with or without purpose-made external markers [2],[32],[22]. Positioning systems utilizing multiple on-board rangefinders based on laser parallax, ultrasonic TOF or RSSI methods are also conceivable, with more sophisticated scanning laser rangefinders having been used on aerial platforms for simultaneous localization and mapping in the past [1]. Position feedback control can also be enabled by off-board position sensors, used in combination with sufficient communication bandwidth to achieve stabilization and trajectory tracking. Many such systems use off-board arrays of cameras capable of quickly identifying one or more markers or features on each vehicle inside the operational volume and using this information to reconstruct vehicle positions and/or attitudes [3], [33].

Full control of position in general requires estimation of vehicle attitude, in order to achieve correct mixing of motor control signals. While for most vehicle designs the system dynamics are observable in the sense that all states may be stably estimated from position measurements alone, available position sensing bandwidth and accuracy preclude this type of estimation in practical application. More direct attitude sensing is therefore necessary, and may be achieved by a number of means. Often positioning systems such as those previously discussed in this chapter are capable of providing attitude measurements, essentially by measuring the positions of multiple points on a single vehicle. Such is the case for IR marker tracking systems such as RAVEN [33], or the ultrasonic system used for SPHERES [26]. With positioning systems that can only output position information, or in cases where it is otherwise advantageous to do so, the vehicles themselves may be equipped with attitude or attitude rate sensors such as magnetometers (assuming that the inherent magnetic field in the operational volume is temporally constant for the duration of operation), and rate 'gyros', in order to provide higher bandwidth estimation of attitude or feedback for stability augmentation. Most of the vehicle designs considered are rotorcraft, with varying degrees of passive pitch/roll attitude stability, which dictates the necessity of stability augmentation for attitude. In the case of a quad-rotor helicopter with fixed pitch propellers, passive stability of pitch/roll angles is low, or lacking completely, and the use of rate gyro based stability augmentation is effectively necessary for stabilization. In contrast, monocopter designs, or those utilizing propellers with stabilizing fly-bars have a large degree of passive attitude stability, tending to align themselves vertically without sensor feedback. In most cases it is advantageous to estimate the heading angle from more direct measurements than position, such as magnetometer output.

The selection of overall system architecture is also influenced by vehicle design, which is to be discussed in the next chapter, and in order for the discussion to remain as general as possible, I consider at this point the family of designs encompassing all types of flying vehicles. While the types of performances required of the system would suggest the use of a rotor-craft design, it is conceivable that fixed-wing or lighter-than-air designs might be implemented as well.

2.1 On-board Sensing

Systems for which the vehicles are each individually capable of determining their position within the operational volume, and thus capable of being independently feedback-controlled, have several advantages and disadvantages when compared with systems for which position sensing is wholly or partly enabled by off-board tracking. In general, systems of this type require more computation to be performed on-board, therefore requiring the vehicles to carry more massive and power-consuming electronics. Additional hardware required to enable position sensing, such as ultrasonic receeivers, a GPS module or camera(s) [21], [2], [22], must also be carried. It is possible to operate a system such as this without communication between vehicles, and with wireless communication between the vehicles and a central control computer limited only to a single initial synchronization, which could be enabled by an extremely small and lightweight sensor such as a single IR receiver, thereby offering a reduction in mass and energy consumption when compared to a system with more substantial communication requirements. Vehicles in such a system, however, would lack the ability to account for one another when executing maneouvres, making the system much less robust and more susceptible to vehicle collisions, as well as less capable of accurately tracking reference trajectories subject to strong aerodynamic interference. Allowing vehicles with only on-board position sensing to efficiently compensate for the effect of wake interference and avoid collisions would require relatively frequent two-way communication of some form between members of the swarm, which may be implemented in a number of ways. achieving this using the direct communication of position between vehicles alone becomes more challenging as the swarm size increases, due to the increasing number of nodes in the communication system, and effectively eliminates any options which rely on every vehicle maintaining knowledge of the positions of every other vehicle.One way to resolve this is to equip each vehicle with a set of sensors that could be used to determine the direction and range of omni-directional signals emitted by the other vehicles, and to add additional terms to the actuator commands that are derived directly from the output of these sensors. Such an implementation may act as an analog 'repulsion' law between members of the swarm, allowing them to avoid collisions without relying on a communication protocol that could not effectively be implemented for large swarms, however it would require additional hardware such as ultrasonic receivers and an omni-directional ultrasonic transmitter, as well as additional computation.

2.2 Off-board Sensing

Many existing systems capable of flying multiple small autonomous air vehicles indoors utilize off-board tracking systems which provide sensing of the positions and attitudes of all vehicles within the capture volume. While it is not strictly necessary to measure the attitude directly, it is effectively necessary for most vehicle designs, due to the bandwidth and accuracy of position measurement available for such systems. This is commonly achieved by placing multiple fiducial markers on each vehicle, and partially or wholly re-constructing the attitude of the vehicle relative to some initial orientation using measurement of the positions of each of the markers. For vehicles types with low passive attitude stability, such as quad-rotor helicopters, the pitch and roll components of attitude must be measured in addition to the heading component, which in general should be measured for any rotor-craft design. While other options exist, systems which utilize an off-board array of IR cameras, in combination with reflective markers or IR LEDs placed on each vehicle have been well tested and are highly suited for this task [20],[34]. In comparison to systems which rely on on-board position measurement, such systems generally require lighter and less power-hungry electronics to be carried, affording a reduction in the size and cost of each vehicle. However, because positions estimates are maintained by a central control system, communication requirements are more stringent. While vehicles determining their own position and attitude are capable of being stabilized with no communication system (though communication of some kind is required for robustness and trajectory tracking purposes as outlined in the previous section), those for which position is externally determined rely on sufficiently frequent communication of control commands, with sufficiently low latency, from an external control computer or bank of computers. This communication need not be two-way however, because it is in general only necessary for the central control system to broadcast commands to the vehicles. There exist a number of wireless communication protocols that would allow a moderately large swarm to be divided into a number of independent channels, with final discrimination of messages achieved by addressing, while maintaining sufficient bandwidth [11]. Control bandwidth may also be limited by the capabilities of the optical tracking system, or by the computational capacity of the central control computer, though the latter may be easily be increased beyond the limit of relevancy by introducing additional computers. For some vehicle designs that lack passive attitude stability, if no additional on-board attitude stability augmentation is implemented, the necessary control bandwidth may be prohibitively high for an optical tracking system such as that formerly described to stabilize even a modest number of vehicles. For other designs with more stable natural attitude dynamics or stability augmentation implemented through higher bandwidth on-board sensors, relatively low frequency position sensing would be sufficient to achieve adequate control. Additionally, because all position sensing and control computation is centralized, it is relatively easy to implement control laws which account for aerodynamic interference and attempt to mitigate collision risk, without any modification to the hardware of the system.

2.3 Hybrid Sensing

In general, optical motion capture systems such as those produced by NaturalPoint or Vicon, suffer reduced update bandwidth as the number of markers being tracked increases, so reducing the number of individual markers to be tracked by eliminating the need for external attitude sensing can offer a significant increase in performance capability, specifically maximum swarm size, without introducing too much additional mass or cost to the individual vehicles in many cases. Magnetometers may be placed on the vehicles and used to determine heading angle in cases where the operational volume is permeated by a magnetic field which remains approximately temporally constant over the time scales required to map the field and execute the performance. In situations where the magnetic field is not spatially constant, additional burden is placed on the communication system, but if the field is approximately spatially constant, no additional communication (or vehicle memory) is necessary to convey position-dependent correction terms. Magnetometers of sufficient precision and reliability for this purpose occupy approximately an additional square centimeter of PCB space, and increase the cost of each vehicle by approximately 10USD when purchased in volumes greater than one hundred units. Position measurement and control bandwidth requirements may also be significantly reduced, for a number of vehicle designs, by the implementation of an on-board stability augmentation system relying on high bandwidth measurements of angular rate, as provided by rate gyros, and optionally feedback from accelerometer readings. Microelectromechanical systems (MEMS) gyroscopes and accelerometers suitable for this task, are also extemely compact and cheap, coming in integrated circuits (ICs) measuring as little as 5mm square, and costing approximately 5USD per unit when purchased in reasonable volumes. Hybrid systems combine many of the advantages of on-board attitude control with those of off-board position control, allowing for relatively small, inexpensive vehicles while

Sensors	Advantages	Disadvantages
Emulated GPS and Magnetometer / Inertial	Low communication bandwidth, unlimited swarm size	Lack of existing technology, no collision avoidance, high vehicle cost
Ultrasonic 'GPS' with Attitude	Low communication bandwidth	Potential sonic interference problems, limited operational volume, no collision avoidance, high on-board computational load
Global On-board Positioning with Local Collision Avoidance	Low communication bandwidth, large swarm size, distributed collision avoidance	High vehicle cost, high on-board computational load, high vehicle cost.
Off-board Optical Tracking with Attitude	Centralized swarm control, collision avoidance possible, low vehicle cost, low vehicle size, very low on-board computational load	High communication bandwidth, limited swarm size, possible interference issues
Off-board Optical Tracking and Magnetometer / Inertial	Centralized swarm control, collision avoidance possible, low vehicle cost, very low vehicle size, acceptable swarm size	High communication bandwidth, possible interference issues

Table 2.1: Comparison of sensor selection options for full position feedback control of all vehicles in the swarm.

maintaining the ability to robustly control a relatively large swarm.

2.4 Summary of Proposed Implementation

A number of implementations have been considered, as summarized in table 2.1, and the final proposed system is described in this section.

Though there are several advantages to implementations using on-board position sensing, the lack of collision avoidance capability, low availability of technology, and increased vehicle cost, size and computational requirements seen by such architectures suggests that a system using off-board position tracking would provide good performance at a superior cost and availability. The main disadvantages of off-board position estimation are a limited swarm size and high communication bandwidth requirements, however, the maximum swarm size attainable would likely still be acceptable if single point tracking is used, and the unidirectional, broadcast nature of the communication structure would allow sufficient communication bandwidth to be allocated for coordinated flight of the swarm. Thus, the sensor architecture proposed for this system is one which utilizes an off-board optical motion capture system to track a single marker (IR LED or reflector) attached to each vehicle, and an on-board magnetometer and yaw-axis gyro for heading angle control and estimation. Depending on the specific vehicle design, additional rate gyros and an accelerometer may be required to provide stability augmentation that would ease off-board position control of the vehicles.

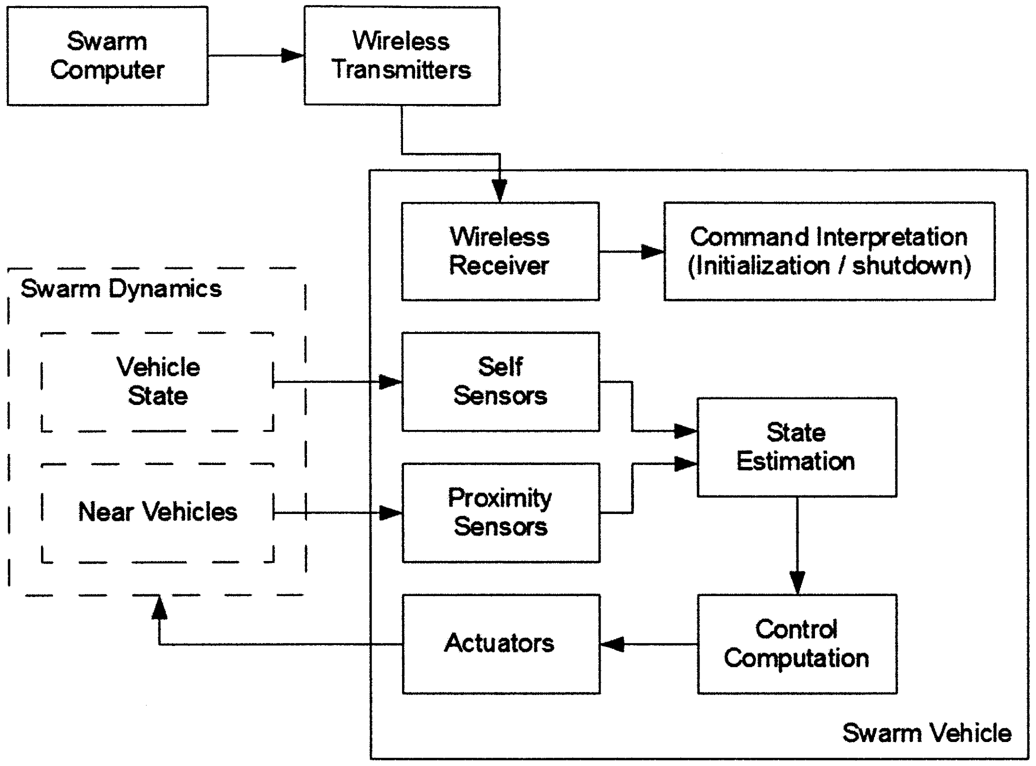


Figure 2.1: Block diagram representing the information flow and connections in a system with all position and attitude sensing and estimation performed on-board. Some communication is still required with a central control computer, for synchronization, initialization or system shut-down.

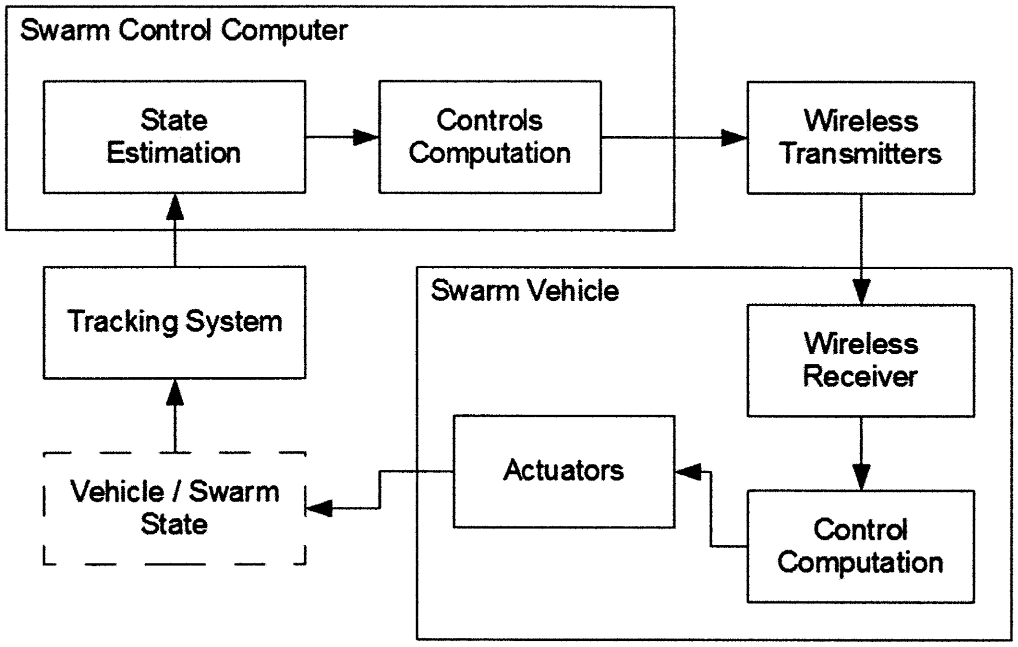


Figure 2.2: Block diagram representing the information flow and connections in a system with all position and attitude sensing and estimation performed off-board. All estimation, and the majority of control computation is performed off-board, while a minimal amount of control mixing may still be calculated on-board.

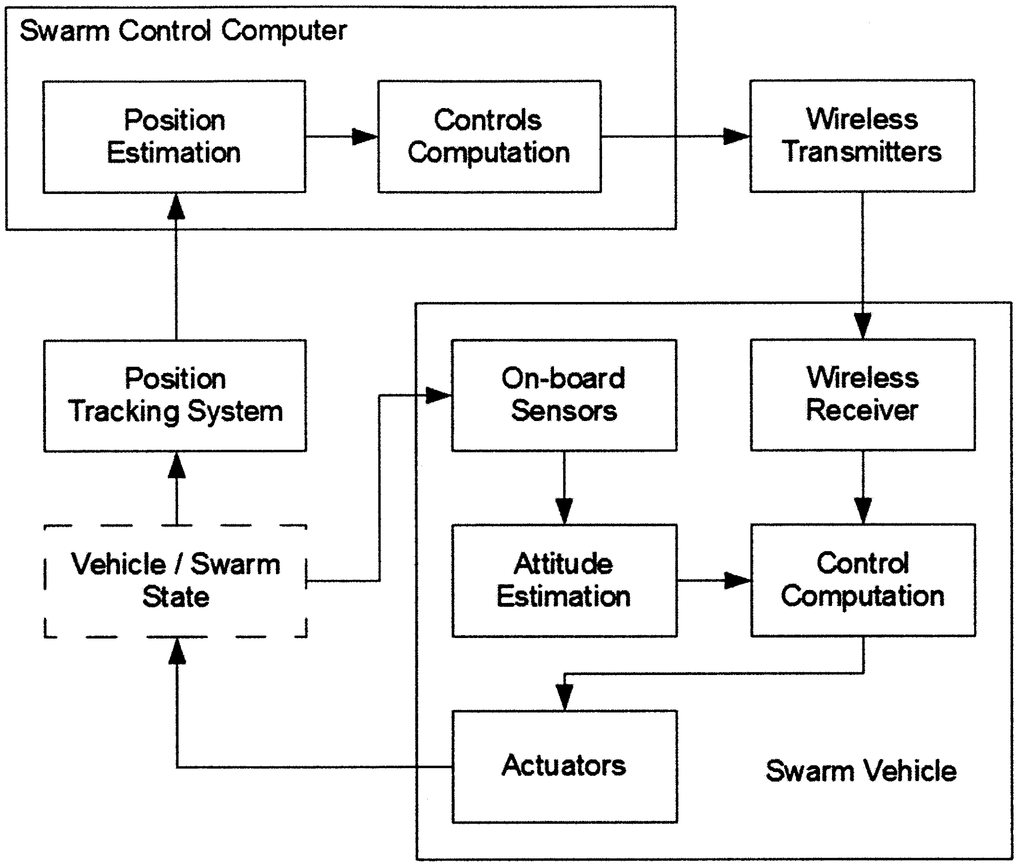


Figure 2.3: Block diagram representing the information flow and connections in a system with hybrid sensing. Each vehicle estimates its own attitude and attitude rates using on-board sensors such as magnetometers, rate gyros and accelerometers, and computes control commands based on these estimates as well as received control commands.

Chapter 3

Vehicle Design and Development

To the author's knowledge, while there are a large number of commercially available toy and research platform helicopters, none are acceptable off-the-shelf for this system. Many such products lack the ability to be modified for automatic computer control, are too large or expensive for the desired implementation, or lack the ability to control the additional coloured light source payload. A number of custom vehicles have also been developed in recent years in both academic settings [25],[15] and by independent enthusaists [30], however these vehicles are generally too large for the required tasks or are not readily available as commercial products for this task. Due to this lack of a readily available solution, over the course of this research a significant amount of effort was invested in the development and testing of a suitable vehicle possessing all required capabilities. This chapter first outlines the considerations and driving factors in the design of the vehicle, then describes the design process, and finishes with descriptions of the prototypes developed and the capabilities thereof. While some work has focused on the development of the physical platform, there exist commercial products which utilize acceptable actuator and structural implementations, so the majority of the vehicle prototyping has centred around the development of custom avionics that satisfy the vehicle requirements and can be integrated with inexpensive commercial hardware.

3.1 Requirements

This section outlines the flowdown of requirements driving the design of the individual unit vehicle used for the system, and describes considerations for different vehicle types and overall system architectures.

3.1.1 General Objectives

The system as a whole operates with the objective of facilitating a three dimensional volumetric display in which individual luminous elements, the vehicles, are controlled to execute coordinated maneouvres and change colour as specified by a pre-determined set of trajectories in physical and colour space (the performance). While it is clear that all types of vehicles will be limited in the performances they are capable of executing, and cannot be made to track arbitrary trajectories to arbitrary precision, so the set of possible performances must be restricted, there are loose requirements on the performance capabilities of the system. These expectations are derived from the more general operational objective that this system embodies, which is to provide an entertainment experience for viewers capable of generating sufficient revenue to recoup development, implementation and execution costs over a reasonable operational timeframe. This requirement of commercial viability is difficult to extend to specific requirements on the performance capabilities of the system, and is also influenced by factors such as implementation and operational costs - a system which is less costly to implement is required to perform less impressively, and a difficult to define continuum of possible cost+performance combinations exist which would satisfy commercial viability. Beyond simple commercial viability, the objective of producing a system which could reasonably be described as a swarm of coordinated, colour controlled, aerial vehicles demands that the individual vehicles are capable of sustaining flight for a reasonable period of time, perhaps several minutes, can be feedback controlled with some degree of accuracy, and can be manipulated in a way that is perceived as synchronous by a human observer. The required accuracy of trajectory tracking is dependent on the size of the vehicles and the minimum inter-vehicle spacing. For some aircraft, such as lighter than air vehicles, absolute collision avoidance is not a strict requirement, while for others for which collision is likely to result in catastrophic 'bicycle race' failure, the probability of collision over the course of a performance must be sufficiently low as to maintian commercial viability of the system. The swarm size is also not provided as a stringent requirement, however the overall objective is to produce a system capable of operating upwards of 200 vehicles at one time as a synchronized swarm.

3.1.2 Venues

The system is intended to operate, at this stage, only in indoor venues with relatively low disruptive airflow, and without access to GPS signals for position control. Operation is intended for moderate to large indoor spaces such as museum atriums, concert halls and stadiums, with the system additionally being required to be capable of relatively simple and swift relocation from venue to venue. Tolerance to weak aerodynamic disturbances such as natural convective airflow indoors, or the airflow generated by indoor air-conditioning systems is required. The system should also be tolerant of a variety of common indoor lighting conditions, as well as optical disruptions such as flash photography and reflections originating outside the operational volume. It is assumed that the operational volume can be manipulated such as to be free of large visual obstructions or surfaces with reflective properties that may cause interference with optical tracking systems, and that the venue can be made absent of illumination from sunlight, which may also cause interference with optical tracking systems. It is also assumed that the operational volume of any acceptable venue will be permeated by a magnetic field of sufficient strength and temporal stability, such as that of the earth, that magnetic field sensors capable of providing acceptable estimation of vehicle heading angle based on the earth's magnetic field will also be useable inside the volume. This requirement on the venue, which allows for magnetic sensing of heading angle, also requires the magnetic field inside the volume to, at all locations, be sufficiently horizontal, as a vertically aligned magnetic field cannot be used to glean heading angle information. Venues are also assumed to be restricted to those with ambient temperature and air pressure near common room temperature and sea level pressure, with sufficient visibility to allow for optical tracking.

3.1.3 Costs

Once a vehicle design is specified, the production cost of the system is dependent on a number of factors, including the individual vehicle cost, the number of vehicles in the swarm, and the costs of any external tracking or other hardware, such as battery charging systems, required for operation. Operational costs occur in addition to the cost required to manufacture the system, induced by the necessity to periodically replace damaged or defective vehicles as well as other hardware, and the logistics associated with the execution of the performances. A sufficiently sophisticated system would allow the deployment of a large swarm with very little human labour, however, most reasonable implementations would require significant labour to organize and execute the deployment of a large swarm, involving connection of vehicles to battery charging stations, initial placement of vehicles, setup of tracking systems, repeated on-site testing and debugging and other setup tasks which require labour time increasing significantly with swarm size.

The design of the vehicle can significantly influence the performance capabilities of the system, with vehicles such as quad-rotor helicopters equipped with high precision inertial measurement units (IMUs) capable of executing aggressive maneouvres with highly accurate trajectory tracking, and others such as blimps or highly attitude-stable monocopters being much slower to actuate. Tests performed with the Ascending Technologies Hummingbird [29], a robust quad-rotor helicopter with on-board attitude stability augmentation, indicate that such platforms are fully capable of executing high-speed, accurate maneouvres, can have the low thrust-to-weight ratios required for rapid vertical acceleration, and are well controlled indoors with IR optical motion capture systems [19],[33],[2]. Several casual tests conducted with a number of commercial toy helicopters have shown a range of capabilities, but in general the cost of the platform increases with robustness, power and size. Platforms which have good passive attitude stability also do not require the additional sensors necessary to implement attitude stability augmentation, and therefore offer a substantial reduction in cost when compared with less stable vehicles. The mass and overall size of the vehicle is also highly correlated with vehicle production cost, with motors, propellers and structural elements decreasing in cost as the size of the vehicle decreases, to a point, at which commercial availability of parts decreases, driving the cost to increase. Based on experience in developing the vehicle prototypes for this system, the mass of the electronics necessary to enable communication and control precludes the possibility of entering the vehicle size regime below that which minimizes cost. Robustness is also traded for cost in the design of the vehicle, with less expensive sensors, motors and structural elements being more likely to fail, and with functionality testing of electronics becoming more costly as the size of the swarm increases.

3.1.4 Collision Probabilities

In general it is desireable to avoid collisions between vehicles in the swarm during execution of a performance, because for many choices of vehicle design collision is likely to result in highly unacceptable behaviour or even catastrophic failure of a large subset of the swarm in a 'bicycle race' type collision cascade. While lighter-than-air vehicles may be designed to be robust to collisions, others such as quad-rotor helicopters, particularly in combination with some types of optical tracking schemes, are very unlikely to recover from collisions. The design of the vehicle must account for the need to avoid collisions while maintaining optimal performance capabilities, particularly as the vehicle design influences performance metrics such as trajectory tracking precision in a number of ways. While the design of the vehicle directly influences tracking performance through sensor and actuation accuracy and control bandwidth, the choice of vehicle design also effects the level and character of aerodynamic interference between members of the swarm, with heavier vehicles generating wakes of higher momentum flow, and smaller vehicles generating more concentrated wakes. Lateral actuation of vehicles also generates lateral airflow proportional in squared velocity to the acceleration of the vehicle. Consideration of aerodynamic interference is complex, as the overall control authority possessed by a vehicle is a reasonable measure of its ability to compensate for aerodynamic interference, and more agile, powerful vehicles in general will be less likely to suffer irreconcilable losses in performance as a result of wake interference. Much of the work presented in later chapters of this thesis focuses on exploration of the problem of compensating for aerodynamic interference.

3.1.5 Size and Mass

Requirements on the size and mass of the vehicles flow down from higher level requirements on the types of venues that the system is capable of operating in, along with the number of vehicles that are expected to comprise the swarm. It is also generally advantageous, as previously stated, with respect to cost, to reduce the size and mass of the vehicles as much as possible. Minimum inter-vehicle spacing is given as a function of trajectory tracking accuracy and vehicle size in light of the requirement to achieve negligible collision probability over the course of a performance. This relationship can be inverted for a given expected level of tracking accuracy, to give a required vehicle size based on the dimensions of the operational volume and swarm size, however, at this point in development, it seems advantageous to reduce the size and mass as much as possible in order to reduce cost, and to allow the operational volume to vary based on the degree to which the vehicles may be miniaturized.

3.2 Prototype Versions

This section gives a brief description of the different prototype designs constructed and tested. All versions constructed during the development of the system were rotor-craft designs, and the methods used to design and fabricate the electronics, as well as the physical vehicles, are described in subsequent sections.

3.2.1 Version 1.0

The first version of the swarm vehicle developed was a helicopter utilizing two concentric, contra-rotating propellers to provide the majority of lift, in combination with four small control propellers arranged with thrust axes vertical, symmetric about the centre of the vehicle. The electronics consisted of a combination of off-the-shelf products, specifically the Arduino Pro Mini and XBee radio transceiver unit [11], and a custom fabricated circuit board carrying a yaw-axis gyroscope, two-axis (planar) magnetometer and high speed power switches capable of controlling the motors by pulse-width modulation (PWM). These components were the STMicroelectronics LY5150ALH, Honeywell HMC6042, and Micrel MIC94067 respectively. The main rotor assembly was taken from a commercial toy helicopter, and connected to a small custom fabricated plastic frame to which the control motors were attached by thin carbon fibre reinforced polymer (CFRP) rods. This prototype was designed primarily to test the electronic design and programming, and was incapable of generating sufficient lift for flight.

3.2.2 Version 2.0

Following the development of the initial prototype, it became clear that a significant reduction in vehicle size could be achieved by integrating most of the electronics into a single, custom fabricated printed circuit board (PCB). The second version of the electronics was therefore developed, which eliminated the use of the Arduino Pro Mini in favour of a versatile custom PCB carrying a micro-controller (ATMEL ATMEGA168), three-axis accelerometer (Analog Devices ADXL345), three-axis magnetometer (Honeywell HMC5843), two gyroscope ICs providing three-axis rate measurement (STM LPR5150ALH and LY5150ALH), three two-channel high speed power switches for motor control (Micrel MIC94066), as well as power regulation circuitry, and connections for the XBee family of radio transceivers. This board is designed for use with a single cell (3.7V) lithium polymer (LiPO) battery pack, and is capable of providing six 2A PWM channels for motor control or any other purpose. A second version of this design was also fabricated, which eliminates the power switches, instead providing direct access to the PWM pins of the micro-controller, such as might be used for control of servo motors or brushless DC motors with separate controllers. The second prototype version was a quad-rotor design utilizing four Micro 2g Ready-to-Run brushless motors from www.gobrushless.com, and associated brushless motor drivers, each claiming 28g maximum thrust. The frame was fabricated as two pieces of CFRP laminated foam-core composite, cut precisely by water-jet, and connected by four aluminum standoffs. The propellers used were 2.5x2, direct drive, and due to a lack of commercially available counter-rotating propellers of this size, forward-rotating propellers were crudely modified using hot air and pliers. Despite the inaccuracy of this method, tests run with the counter-rotating propellers showed similar lift and power consumption properties as the forward-rotating versions. This version was also incapable of generating sufficient lift for takeoff, with the motors each producing less than 14g thrust at one hundred percent duty cycle. In addition, significant over-heating of the motors caused a further reduction of thrust over the course of operation, as well as reasonable concern for failure given that the motors often became too hot to comfortably touch after only about ten seconds of maximum duty cycle operation.

3.2.3 Version 3.0

Due to the failure of prototype version 2.0 to generate sufficient thrust for takeoff, measures were taken to reduce the mass of the vehicle by abandoning the robust construction afforded by aluminum and CFRP foam composite and adopting a construction consisting of CFRP rod, with thin spring-steel landing gear. The inadequate brushless motors and drivers were also replaced with smaller brushed motors controlled directly by the MIC94066 ICs on the main PCB, capable of providing approximately 12g of thrust each at full duty cycle with 2.5x2 propellers attached directly to the motor shafts. During the development of this prototype, a number of tests were conducted to characterize the thrust output and power consumption of the motors utilizing a number of different propllers, some of which had geared connections to the motors rather than direct attachment. For fixed thrust output, geared connections consumed significantly less power when used in combination with larger 3x2 or 3x3 propellers, however these larger propellers are also not readily available in counter-rotating versions, and modification of forward-rotating versions using hot air has proved to be much less effective than with the smaller versions. For this reason construction of the prototype proceeded as described, though it was noted that subsequent versions should seek to utilize counter rotating 3x3 propellers with geared connections to the motors for increased efficiency and hence flying time.

3.2.4 Version 3.1

This version was essentially very similar to version 3.0, but with the PCB redesigned physically to occupy less space, and specifically to fit inside the standard size table tennis ball which would serve as a container for the electronics, battery, and LED, as well as a diffusive covering for the LED. Electronically, the only difference was the elimination of one of the MIC94066 ICs, which was not utilized in version 3.0. This design reduced the overall radius of the vehicle as well as the mass, and provided a more aesthetically pleasing design with respect to LED diffusion and minimal structural blocking of the display LED. In both versions 3.0 and 3.1, the brushed DC motors used were lashed and epoxied to the ends of the CFRP cross-members of the frame, which were prepared by milling the ends to create vertically aligned grooves into which the motors partially fit, for alignment and securement purposes. Despite this construction procedure, adequate alignment of the motors was difficult to achieve, and tests with this version showed that, in order to achieve near zero angular acceleration in yaw, a significant constant offset yaw command was required. While it was still possible to generate sufficient thrust for take-off, tests with this version were abandonded in favour of development with future versions.

3.2.5 Version 3.2

The difficulties in achieving acceptable fabrication accuracy with the methods described for the previous versions lead to the consideration of commercially available hardware, and specifically the Walkera UFO8 quad-rotor was used to develop the next version of the quad-rotor design. Version 3.2 utilizes the frame, batteries, motors and propellers of the Walkera quad-rotor, but replaces the electronics with those used in prototype version 3.1, modified with a bolt-hole pattern matching that of the stock electronics used on the Walkera. The propellers used by this helicopter have a flexible linkage joining the blades to the shaft, which provides some degree of passive angular stability in pitch and roll, which can be further augmented by feedback from the on-board electronics. This version is shown in figure 3.4.

3.2.6 Version 4.0

In parallel with the development of version 3.2, an alternate, monocopter design was also pursued, which utilizes two brushed DC motors mounted with thrust axes horizontal to drive the entire vehicle to a substantial angular rate about the vertical axis, with lift being generated by two large wings attached to the body of the vehicle as it spins. Figure 3.5 shows an overhead view of the monocopter as it would fly, with the electronics exposed. The linkage between the wings and the hub of the vehicle allows the wings to hinge upwards, providing strong attitude stability in pitch and roll. Such a vehicle is actuated laterally by controlling the voltage (PWM duty cycle) difference between the two motors as the vehicle spins, such that time averaged lateral force applied by the propellers assumes a particular direction. In this implementation, the lateral actuation is effectively holonomic over time scales significantly longer than the yaw revolution period. This prototype utilized most of the hardware, including motors and mountings, wings, and some frame elements, from the comercially available Wowwee Bladestar monocopter, but with electronics replaced with a single custom PCB. The Bladestar produces very little excess thrust and the stock electronics weigh only approximately 2g, so a new circuit board had to be designed which eliminated the need for the 3.5g XBee for communication. In its place, the Nordic Semiconductor NRF24AP1 radio frequency (RF) transceiver IC was used, which requires only approximately one sqaure centimeter of PCB space, and adds negligible mass beyond that of the PCB substrate. The electronics used for this version consist of a single custom PCB weighing approximately 2.5g (with thick substrate), and carrying the RF transceiver, micro-controller (AMTEGA168), a two-axis magnetometer (Honeywell HMC6042), one MIC94066, and power regulation circuit, as well as a manual power switch. The PCB also has through-holes into which a RGB and an IR LED can be soldered, for direct control from the otherwise unused PWM pins on the ATMEGA168. A number of tests have been conducted with this prototype, but there are several issues with the current design that suggest that use of the Wowwee Bladestar is not a feasible option for the system. While communication is effective, sufficiently high bandwidth and sufficiently low latency, there is power line interference between the motors and the RF transceiver that causes the transceiver to inevitably enter an inoperable state that can only be resolved by cycling power to the device, when the motor duty cycle exceeds approximately thirty percent. This renders the vehicle completely incapable of being externally feedback controlled for flight, however, even using the original out-of-the-box Bladestar, the vehicle is incapable of being laterally actuated without suffering a reduction in thrust that causes downward acceleration. All tests performed with the modified prototype to date support the conclusion that it is not possible to achieve predictable lateral actuation using angle-based PWM control without an unacceptable loss of thrust - the vehicle cannot be actuated simultaneous laterally and upwards with sufficient authority to reject small wind or other disturbances. Tests using the on-board magnetometer to actuate the vehicle laterally along the ground have demonstrated the capability of the vehicle to be predictable actuated laterally, however this does not necessarily imply that precise stationary hovering is possible.

3.3 Electronics Design and Fabrication

This section describes the processes used to design and fabricate the electronics used in the various prototypes. It also describes a number of tests performed to validate the functionality of the electronics prior to deployment. Schematics of the various electronic designs are given, with explanations of the purposes of different components as well as justification of design decisions made. Over the course of prototype development several PCBs were developed which provide somewhat general functionality and could be implemented in a variety of other systems, and, to the author's knowledge, provide programmability, sensing and actuation equivalent to many commercially available products in a smaller, less massive packages.

3.3.1 Design and Layout Process

The circuit boards developed for this project were designed in the ExpressPCB custom software and the Eagle PCB Layout Editor. The online company ExpressPCB and their free, eponymous proprietary layout software allow users to draw and manipulate trace patterns and schematics, and order boards quickly and cheaply online. While several similar services exist which utilize industry standard output files such as those generated by the Eagle Layout Editor, learning to use such software is signficantly more time consuming, and for relatively simple, one-off prototype boards it was decided initially to use the less versatile ExpressPCB. As a qualitative comment, their service is exceptional, with all orders having been received within four business days of ordering, defect free and manufactured exactly to specifications. For larger scale production however, in which PCB assembly in addition to manufacture would be required, it would be necessary to use software capable of generating standardized output files in order to facilitate automated PCB assembly. For this reason, later versions of the electronics were also laid out, with formal attached schematics and lists of parts, in the Eagle Layout Editor. This editor is available for free online, with paid versions existing also that allow the layout of larger boards with more trace layers. Typically, for PCB assembly, a bill of materials (BOM) file, as well as a formatted list of component centroids and rotations is provided in addition to the trace, hole, silkscreen and ink patterns required to fabricate the PCB itself. The components may then be provided to the assembly company, or purchased by the assembly company for an additional fee. In general, PCB assembly is moderately expensive, particularly for relatively small production runs on the order of low hundreds of units - approximately 100USD per board, compared to 10USD per board for manufacturing alone.

Design of the PCBs was carried out based on the implementation circuits provided by the component data sheets, with components selected from online vendor catalogues (primarily DigiKey) based on price and fulfillment of requirements. The Atmel ATMEGA168 microcontroller was selected as the primary microcontroller for all designs, because of the large base of software libraries developed specifically for use with this chip through the Arduio project. In all implementations, the 8MHz internal clock is used for the microcontroller, rather than an external 20MHz clock, which is common in arduino implementations, in order to conserve board space and reduce complexity, and because several variants of the ATMEGA168 cannot operate at 20MHz with only 3.3V supplied power. Some of the layouts used, however, do have pads for an optional 20MHz resonator that would allow acceptable microcontroller variants to operate at the higher clock speed. The layout of the power regulation circuitry was also based on that of the Arduino boards, for which schematics are legally freely available online. All boards were designed to operate with the nominal 3.7V input power supplied by a single cell LiPO battery pack, and so utilize regulated 3.3V power and logic for the electronic components, provided by the Micrel MIC5205 low-dropout voltage regulator. In order to reduce noise caused by the switching of the motors and variable power consumption of the electronics, power supply decoupling capacitors were placed between the power lines and ground on both sides of the regulator, in all implementations. Due to non-linearities absent in the common theoretical models of capacitors, it is desireable to use multiple capacitors with capacitances separated by several orders of magnitude in capacitance in order to effectively reject noise across a range of frequencies. In all implementations developed, a 10μF on either side of the regulator, and multiple 0.1μF capcitors placed at different locations on the board were used. It is often suggested to utilize multiple decoupling capacitors in this way in order to separately filter the power for each of the sensitive electronic components, however it is uncertain whether this is entirely necessary or even offers a noticeable benefit. Nonetheless the capacitors add negligible cost and mass, and only a very moderate amount of production labour to each unit.

3.3.2 Electronics Structures

This section contains overviews, schematics, and trace patterns for the electronics developed in the course of this project, as well as explanations of the interfaces between the various chips, and the functions of other components on the boards. Versions of the electronics developed with reasonable similarity are grouped together, with documentation provided only for the final version, and any changes between design iterations described. This reduces the set of different electronic designs to three: the three-board stack developed for initial testing, the versatile board developed for control of a quad-rotor helicopter, and the small board designed for control of a mechanically attitude-stable monocopter.

Figures 3.6, 3.7, and 3.8 show simplified schematics of the three significantly different electronics stacks developed. Power regulation circuitry and power connections are omitted, as well as the discrete components such as resistors, capacitors and inductors used. The interfaces between the sensors, RF transceivers and switches, and the microcontroller are shown. The three-axis magnetometer and accelerometer used on the quad-rotor board communicate via the inter-integrated-circuite (I2C) bus protocol, while all other sensors have analog outputs that are multiplexed inside the microcontroller and read using the analog-to-digital converter (ADC) also in the controller. The MIC94066 high-side power switches receive digital signals from the microcontroller, and correspondingly open or close the solid-state 'switches' connecting the battery pack to the motors. The ATMEGA168 and ATMEGA328 have six digital pins that can be programmed for low-level hardware PWM ouput, which are used as the inputs to the power switches, and for direct control of the red, green and blue components of the payload LED on the monocopter design. Where three PWM channels are not available for LED control, a small commercial daughter board, the BlinkM [31] which allows control of a single RGB LED via I2C is used. The XBee radio module provides robust wireless communication capabilities along with a simple, nearly transparent serial interface - communication between the board and the central computer is achieved through the use of a pair of XBee modules, one of which connects to the computer via USB, and occurs for most purposes as if the board were connected directly to the computer through an RS232-USB converter. The monocopter design utilizes the NRF24AP1 RF transceiver IC, which communicates with the microcontroller via I2C, and must be reconfigured at each power-up.

3.3.3 Manufacturing Process

Following etching and drilling by ExpressPCB, all electronic prototypes were populated with components using a reflow solder assembly process in which solderpaste was applied to the pads on the board, the components placed, and the board placed under a fixed hot-air jet to vapourize the flux in the solderpaste and melt the solder. The fine pitch between the pads of many components necessitated the use of a fine syringe and microscope for the application of solderpaste, and because many of the ICs to be placed were in quad flat no-leads (QFN) packages, reflow soldering was the only option for assembly. In some cases larger components such as the sockets for the XBee radios or the power switches, which consist partially of plastic construction, were added at the end of the assembly and testing process, by manual soldering, to avoid melting, which can occur if hot air is used. Reflow soldering is also often accomplished by means of placing the boards into an oven which then executes a pre-determined temperature profile, normally matching the temperature profiles suggested on the component data sheets, if any. The machines used in the assembly process were all Zephyrtronics devices.

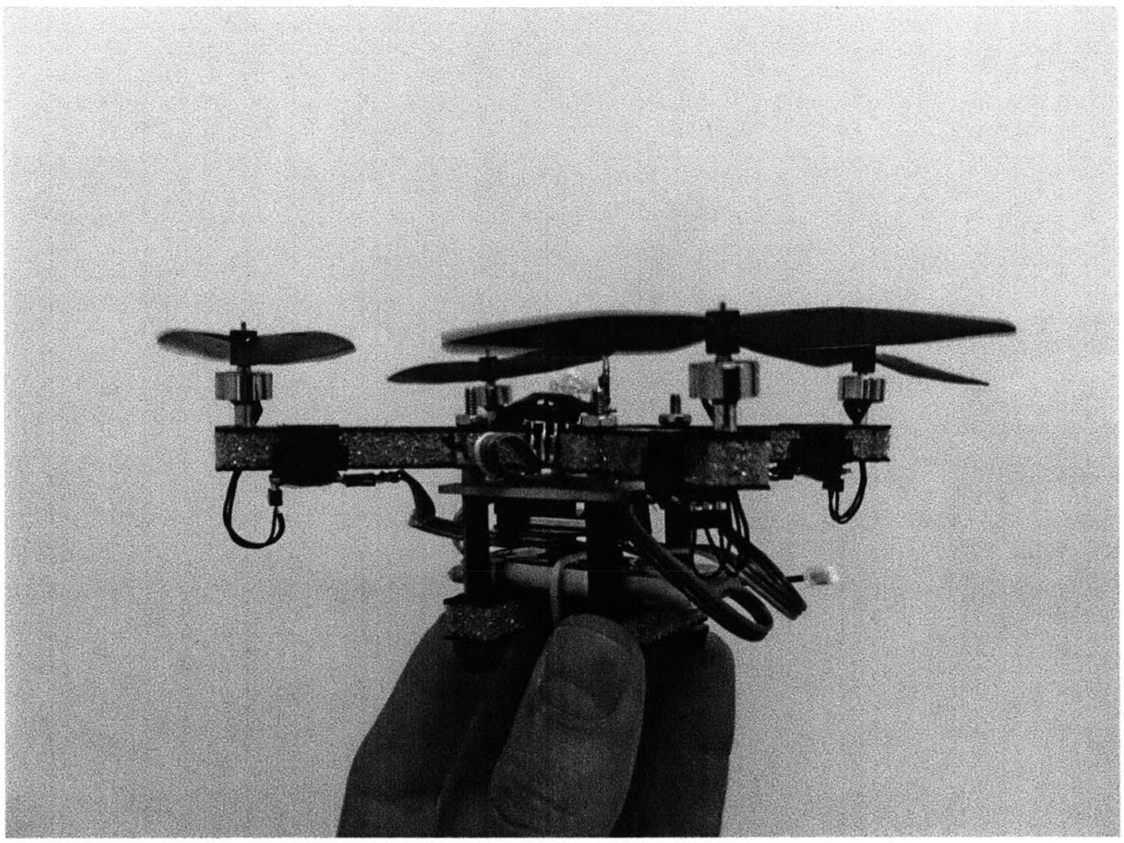


Figure 3.1: Photograph of quad-rotor version 2.0, showing CFRP reinforced, foam core and aluminum frame construction, as well as brushless motors with electronic speed controllers. These motors provide only approximately half of that stated on their website, and the vehicle is incapable of achieving lift-off.

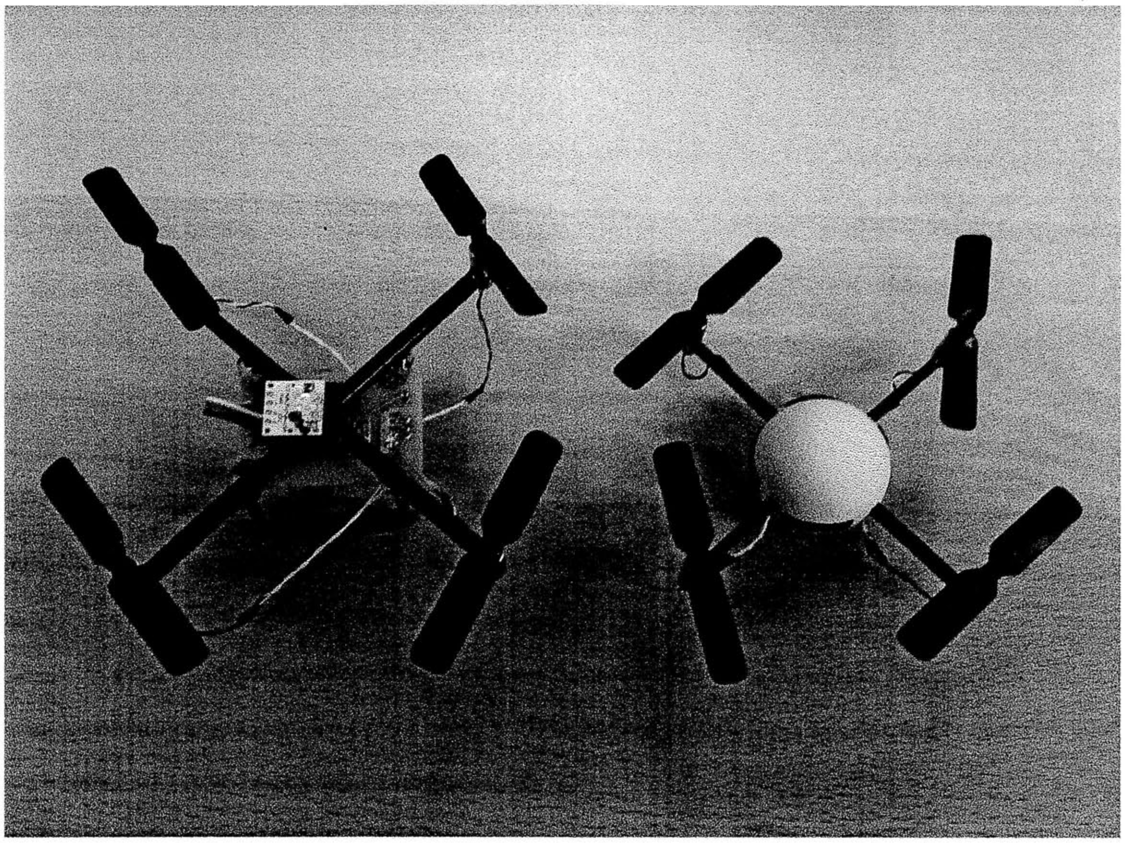


Figure 3.2: Photograph of quad-rotor versions 3.0 and 3.1 (left and right respectively). Though electronically similar, the electronics of V3.1 is reduced in size in order to fit within a standard ping-pong ball, as seen here.

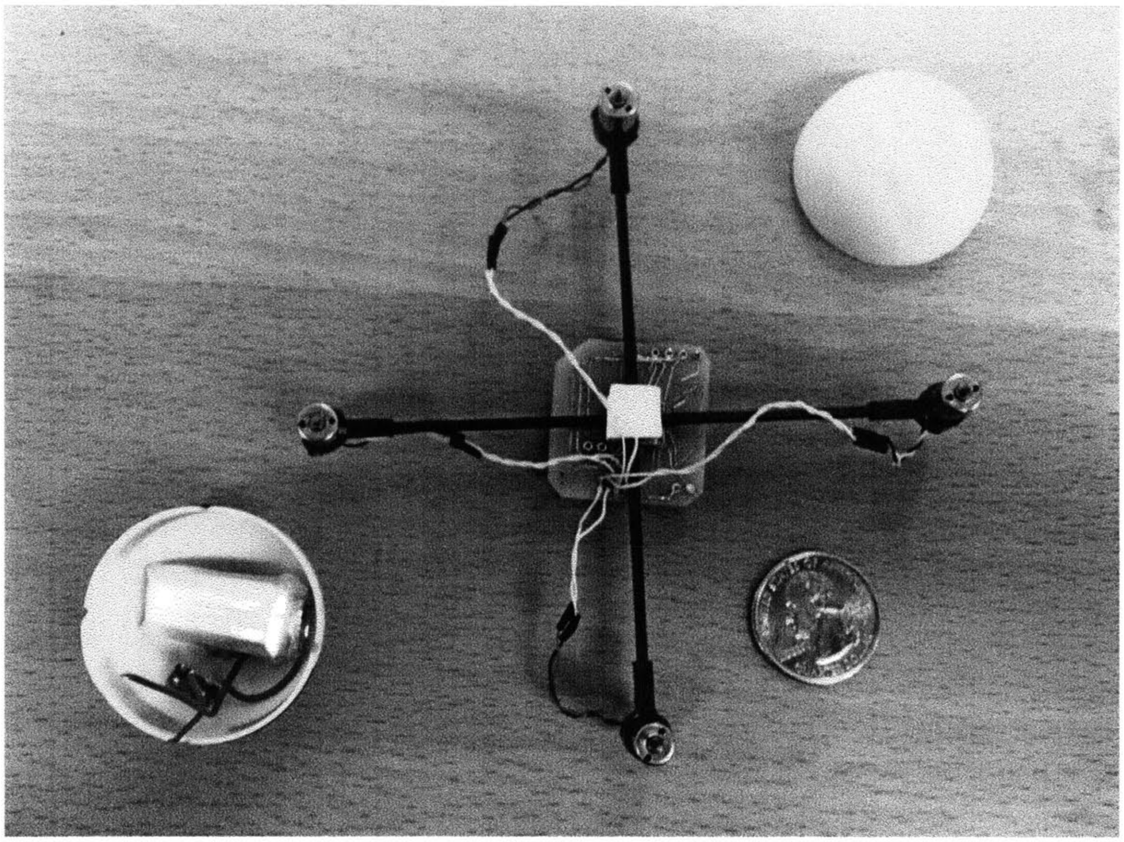


Figure 3.3: Photograph of quad-rotor version 3.1, showing the ping-pong ball casing with battery pack, and a quarter for scale. The electronics are bonded directly to the CFRP frame, with the table tennis ball supporting the vehicle while on the ground.

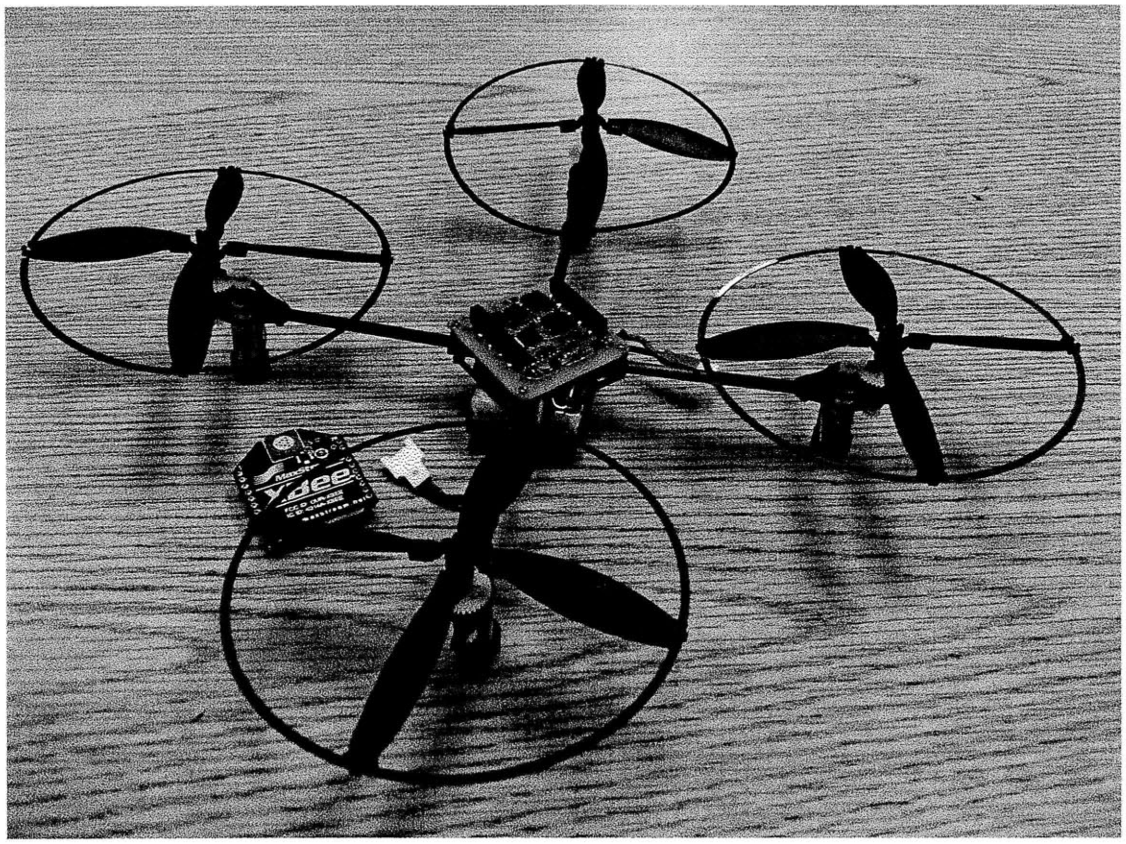


Figure 3.4: Photograph of the most recent quad-rotor helicopter developed for this project, with the XBee radio transceiver detached, showing the custom electronics above the battery pack. The frame, motors and propellers are taken from the Walkera UFO8, a commercially available toy.

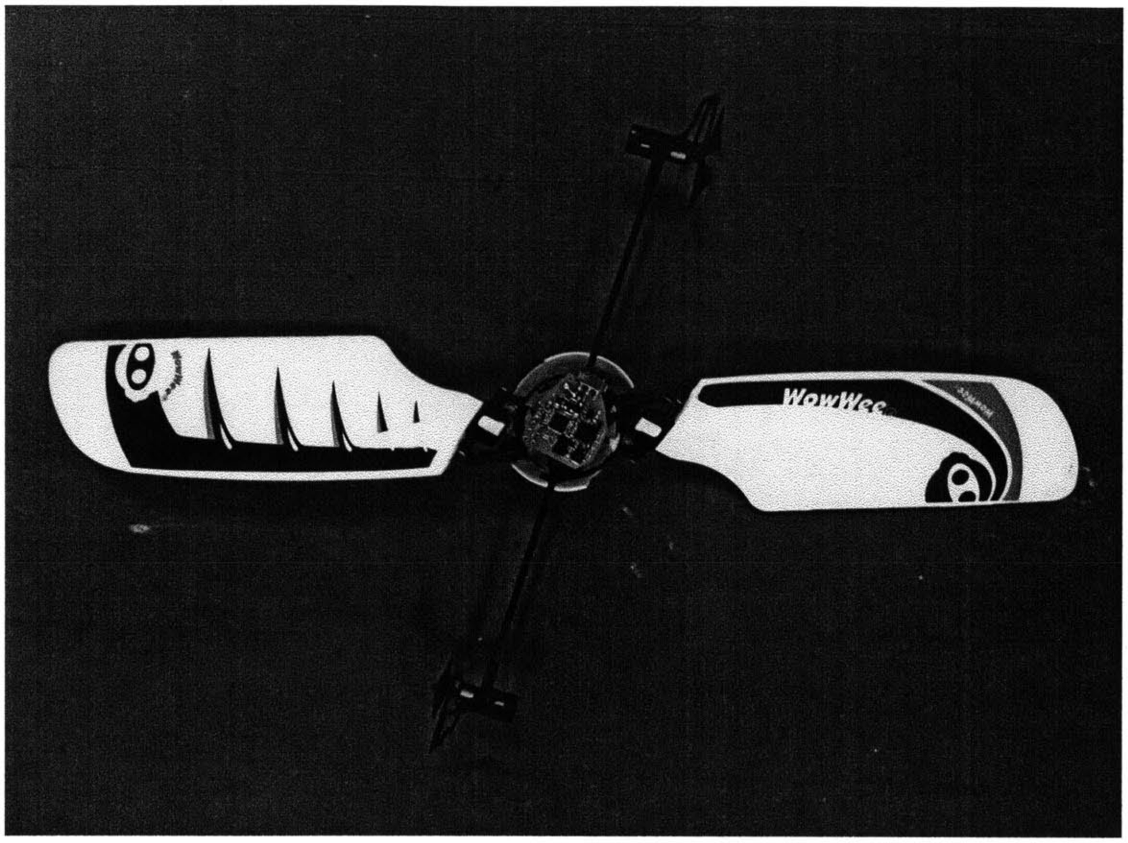


Figure 3.5: Photograph of the most recent monocopter design developed for this project, taken from above, as the monocopter would fly. The top casing of the body is removed, exposing the custom electronics. All other components on the vehicle are taken from the WowWee Bladestar, a commercial toy sold for approximately 20USD.

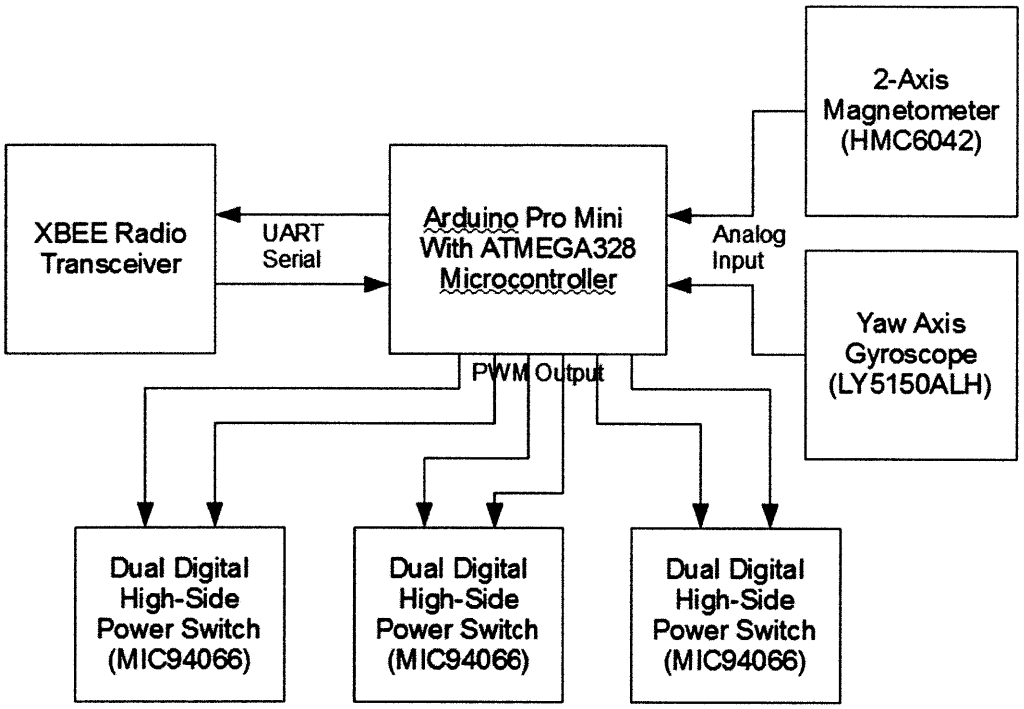

2-Axis Magnetometer (HMC6042)
XBEE Radio Transceiver
UART Serial
Arduino Pro Mini With ATMEGA328 Microcontroller
Analog Input
Yaw Axis Gyroscope (LY5150ALH)
PWM Output
Dual Digital High-Side Power Switch (MIC94066)
Dual Digital High-Side Power Switch (MIC94066)
Dual Digital High-Side Power Switch (MIC94066)

Figure 3.6: Simplified schematic of the first functional electronics stack developed for testing of interfaces with analog sensors, use of the MIC94066 high-side power switch for motor control, programming of the microcontroller, and use of the XBee radio for wireless communication. The three 2-channel power switches utilize the six PWM output pins on the microcontroller to allow independent 3.7V switched PWM control of up to six motors, with a maximum current draw of 2A each.

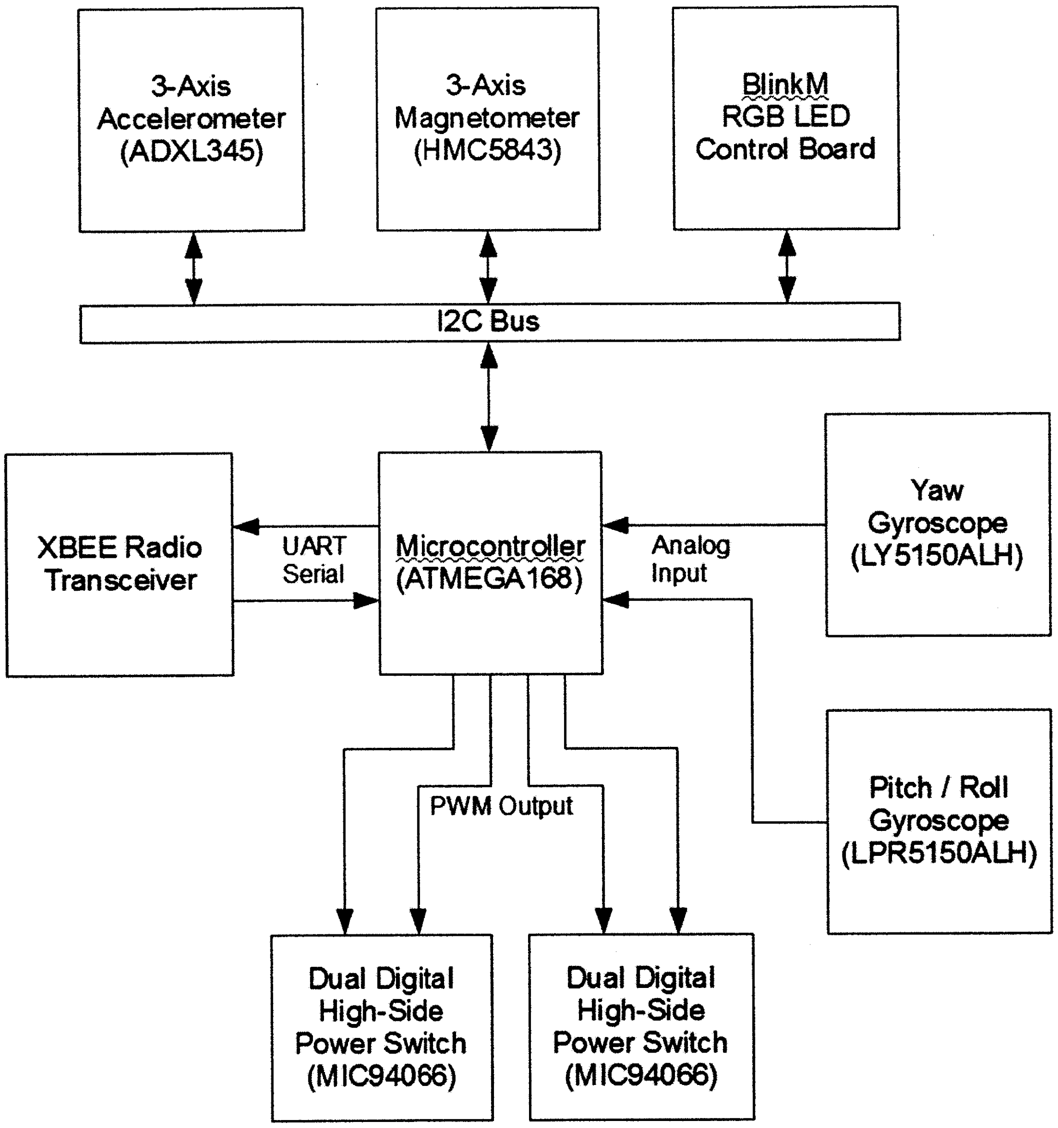

3-Axis Accelerometer (ADXL345)
3-Axis Magnetometer (HMC5843)
BlinkM RGB LED Control Board
I2C Bus
XBEE Radio Transceiver
UART Serial
Microcontroller (ATMEGA168)
Analog Input
Yaw Gyroscope (LY5150ALH)
Pitch / Roll Gyroscope (LPR5150ALH)
PWM Output
Dual Digital High-Side Power Switch (MIC94066)
Dual Digital High-Side Power Switch (MIC94066)

Figure 3.7: Simplified schematic of the final electronic design for quad-rotor control, showing general structure. Each MIC94066 chip switches direct battery power to two motors. All other chips rely on regulated 3.3V power supplied by a MIC5205 regulator. The XBee serial wireless module consists of a single separate board that connects via dual 10x2mm header plugs to the main board, which contains all other electronic components.

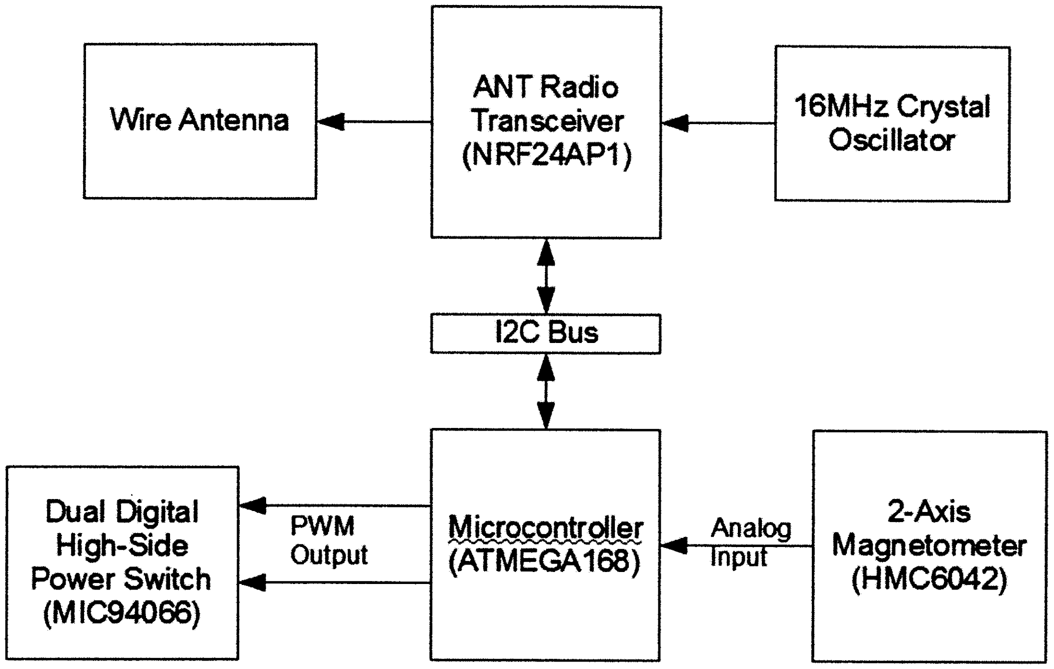

Wire Antenna
ANT Radio Transceiver (NRF24AP1)
16MHz Crystal Oscillator
I2C Bus
Dual Digital High-Side Power Switch (MIC94066)
PWM Output
Microcontroller (ATMEGA168)
Analog Input
2-Axis Magnetometer (HMC6042)

Figure 3.8: Simplified schematic of the final electronic design for monocopter control, showing general structure. The MIC94066 chip switches direct battery power to both motors. All other chips rely on regulated 3.3V power supplied by a MIC5205 regulator. This design consists of a single PCB containing all components, with wireless communication provided by the ANT NRF24AP1 wireless transceiver IC in place of the XBee module used in all other designs. Omitted here is the RGB payload LED, which is driven directly by three of the PWM outputs of the microcontroller.

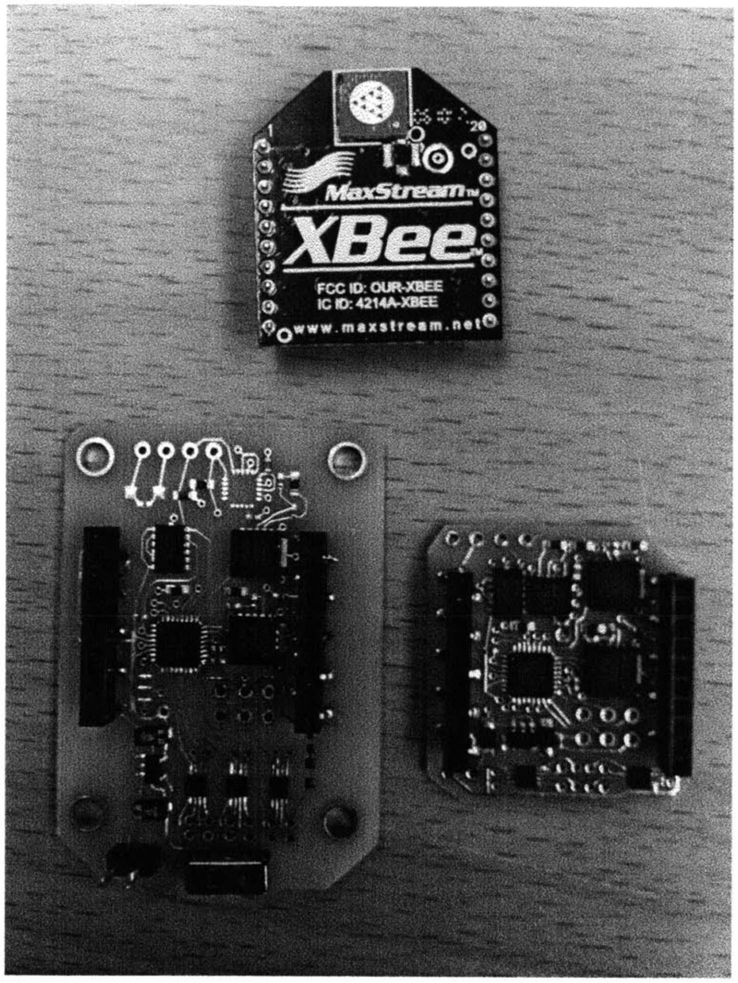


Figure 3.9: Photograph of the electronics developed for quad-rotor control, used in prototype versions 3.0 and 3.1 (left and right respectively), with the XBee radio transceiver shown as well. The transceiver plugs into the dual 10-hole plugs on each board, facilitating communication to the swarm computer.

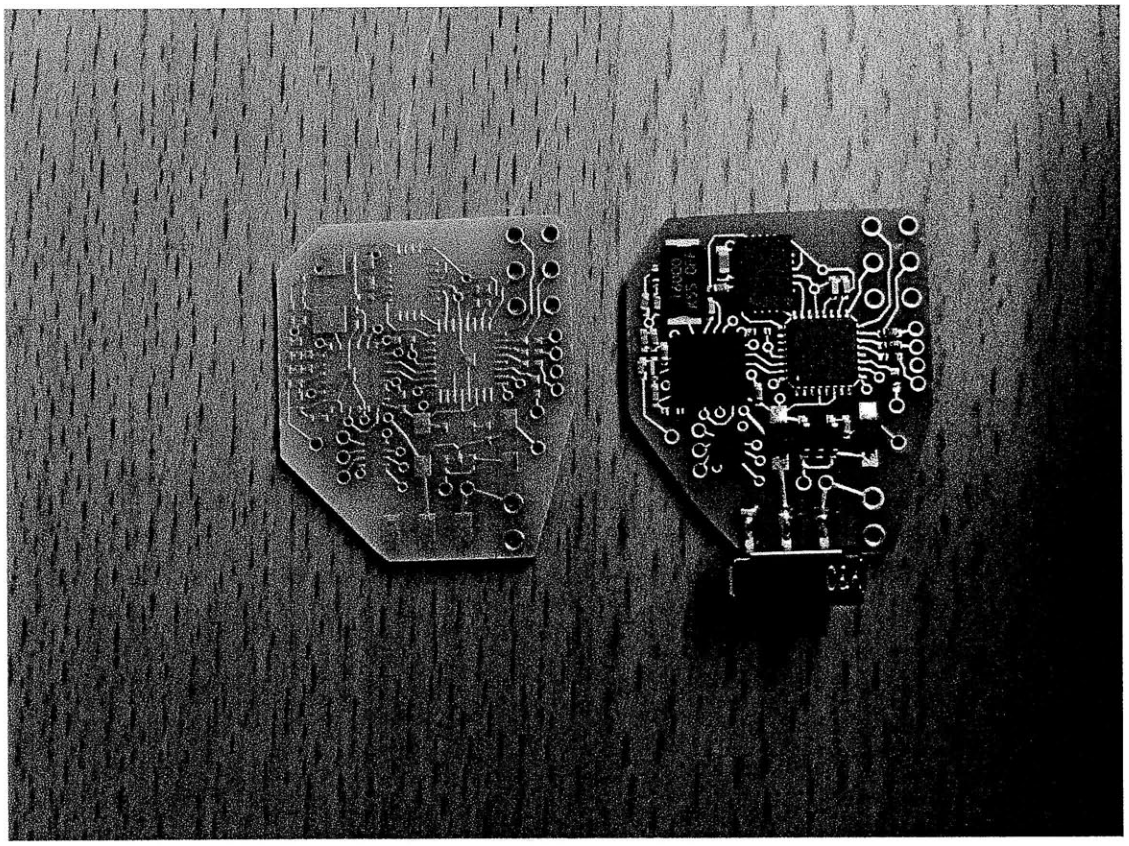


Figure 3.10: Photograph of the electronics used on the monocopter design, before and after population with components. The board is approximately 2cm in dimension, and weighs approximately 2.5g when fully populated.

Chapter 4

Software Design and Development

This chapter describes all aspects of software development and implementation relevant to this project, both the embedded implementations on the individual vehicles as well as all software developed to operate on the swarm control computer. Algorithms used or proposed for on-board stability augmentation and off-board control and estimation are described, as well as the proposed system communication architecture. Much of the software is based on tests conducted with the AscTech Hummingbird, which cannot itself be re-programmed but for which a significant amount of software has been developed for control in conjunction with the Optitrack motion capture system [20].

4.1 Embedded Programming and Software

The on-board software for all embedded systems designed for this project is run on the ATMEGA168 microcontroller, and serves a number of purposes including interfacing with sensors (initialization and reading), interfacing with radio communication chips, filtering of sensor signals, maintenance of on-board attitude estimates, computation and setting of actuator commands, setting of LED colour commands, and implementing a communication protocol consistent with that used by the swarm control computer to enable the flying of the vehicles by an external observer and the return of data from the vehicle to the computer for debugging purposes. All software developed for use on the microcontrollers was written in the Arduino development environment, included at least the base Arduino libraries, and was compiled using the compilers provided with the environment. However, much of the software required additional functionality beyond that of the Arduino libraries, leading to significant direct register access and custom interrupt implementations. This section describes the structure and purposes of the software implemented on the various versions of the electronics, and the general functionality provided by the programmed electronics. All versions of the electronics developed for this project were programmed using the Atmel AVR in-system programmer (AVRISP), which requires a six-pin direct connection to the microcontroller, that was enabled by the addition of a six-hole header to each of the prototype boards. During large-scale production, it would be possible to design the boards without this header, and pre-program the microcontrollers separately in order to conserve as much mass as possible, though the header itself requires only about an additional one square centimeter of board space, and likely adds less than half a gram of mass.

4.2 Swarm Computer Software

This section describes the structure of the software implemented, and suggested to be implemented, on the central swarm control computer. In implementations of the system consisting of a large number of vehicles, multiple, networked computers may be used to control the swarm, requiring a change in software structure. The overall structure of this software is reasonably modular, and has functionality that can be divided into four categories: interfacing with the optical tracking system to provide position measurements, executing estimation, control and mixing algorithms, interfacing with wireless communication modules to communicate to the vehicles in the swarm, and providing a graphical and command interface for the user. Figure 4.1 shows the program flow of the software used for flight of multiple vehicles during testing. In this case, control signals for all vehicles are computed sequentially by a single thread, on a single machine. Larger swarms would likely require control computation to be distributed across multiple machines, or potentially across multiple cores in a graphics processing unit (GPU) or other parallel architecture, which would in turn require a more complicated program flow.

4.2.1 Interface with OptiTrack

The proprietary software provided with the OptiTrack motion capture system allows for capture information including point cloud or rigid body data to be streamed across the network, and supports a number of different communication protocols. Both Unix Datagram Protocol (UDP) and Transmission Control Protocol / Internet Protocol (TCP/IP) are supported, and either could be used by control software to receive swarm measurements. In all current implementations in which this information is recieved, UDP is used, with a thread in the control program dedicated to listening continuously for UDP multicast packets, decoding the packets, and passing the information to other threads for estimation and control purposes. The user must ensure that the OptiTrack software is running with the correct configuration settings to enable broadcasting of the point cloud or rigid body data, depending on which packet decoding function is to be used.

4.2.2 Wireless Communication

Two different wireless transceivers were used over the course of development of different prototypes for this project: the XBee (or XBee Pro) ZigBee module, which offers nearly transparent wireless serial communication, and the ANT NRF24AP1, which must be operated at a lower level and communicates via I2C with the on-board micro-controller. The standard XBee module is exceptionally easy to use and provides reliable communication over the distances generally considered for this project, and more expensive and powerful versions offer communication over a significantly longer range (up to 1500m) with the same physical and electronic interface. The ANT RF transceiver is much weaker, and with simple quarter-wave wire antennas, offers reliable communication only over a range of approximately 5m, as well as a significantly lower data transfer rate. For both radio transceivers, there exist USB interface boards that can be connected, via USB, to the control computer, and used by control software running off-board to communicate with the vehicles. These devices can be purchased from a number of online electronics suppliers, including Sparkfun Electronics, from which they are called the XBee Explorer USB and Nordic USB ANT Stick. Communication between control software and these devices is achieved through the use of a Java-wrapped C library called RXTXcomm, or RXTXcomm.jar, which allows for the use of such USB¡-¿serial devices by facilitating the transfer of data to and from the computer's real and virtual COM ports. In existing software implementations, a single thread is typically devoted to interfacing with the vehicle(s) in the swarm in this way, taking information from the thread executing control computations and sending it to the wireless transceiver at approximately fixed intervals, as well as, in cases where information is read from the vehicles during operation, querying the vehicles for data at approximately fixed intervals and updating a data logging thread with received data.

While the XBee modules maintain configuration settings through powerdown, and can be configured simply, from any computer, using the Digi X-CTU software and an XBee USB board, the Nordic ANT modules must be re-configured each time they are powered-up, and operate somewhat differently from the XBee modules. In order to send information across the XBee wireless link, the data must be written across the serial (or USB virtual serial) line to the module, which then automatically transmits the data and optionally uses acknowledgements and re-sends to ensure successful transmission of all packets. The ANT transceiver can be configured to operate in a number of ways, however, for these applications only uni-directional broadcast communication was tested, and in this case, following configuration of ID parameters, transmission power and radio frequency, as well as packet transmission period, the broadcasting module sends a small packet (9 bytes data) at fixed intervals defined by the packet transmission period. A new packet is transmitted every frame, regardless of whether the host has changed the content of the packets, and thus packets are re-transmitted indefinitely until the host computer or micro-controller sends more data to the transceiver, at which time the new data is re-transmitted indefinitely at fixed intervals. For feedback control operation, the host computer is connected via USB to the ANT transceiver configured for uni-directional transmission, and the vehicle is programmed to configure the on-board transceiver with matching radio frequency and addressing settings, for uni-directional reception at powerup. It is often desireable for testing to perform the opposite configuration, in which the vehicle broadcasts on-board data such as sensor output to the host computer. It is also possible to use the ANT transceivers for bi-directional communication, however, because the final implementation of a large swarm would preclude such two-way transmission, only uni-directional broadcast communication was tested.

4.2.3 User Interface

Another important component of the software developed for control of multiple vehicles from a single central computer using a motion capture system for feedback is the user interface, which allows the user to operate the vehicles manually, select reference positions or load reference trajectories, change between different control modes, and monitor and record motion capture and estimator data during operation. Additionally, a number of user interfaces were developed for simple sensor and control testing purposes, which allowed the user to adjust motor or LED commands manually, and to see approximately real-time plots of on-board sensor output. Qualitative testing of on-board sensors was performed in this way, with accelerometer, gyro and magnetometer output plotted as the vehicle was rotated and accelerated. Magnetometer output was also used to draw a compass needle visualization of the magnetic field, which was used to assess calibration procedures and overall sensor viability.

The user interface implemented for operation of a small number of vehicles in synchrony provides approximately real-time plots of the helicopter position estimates, as well as attitude estimates if rigid-body tracking was performed by the motion capture system. In the case that only single point tracking is used, the interface allows the user to initialize the swarm by associating point cloud points with the individual helicopters, and in the case of rigid-body tracking, it allows the user to associate a rigid body ID (assigned by the motion capture software), with the helicopter ID used by the control program. This interface also allows the user to switch between manual and automatic control modes, to adjust the reference point tracked by each of the helicopters, and to begin data capture. The program thread responsible for maintaining the user interface is the core of the program, from which all other threads are spawned, but itself performs no additional tasks beyond operation of the user interface. Manual control of the AscTech Hummingbird allows the user to fly the helicopter using a joystick or video game controller.

4.2.4 Estimation and Control Software

The control program running on the central computer uses a dedicated thread to perform all control and estimation computations and updates. This thread receives data from the motion capture interface thread, and potentially also from the wireless communication thread, which it uses to the update estimates of the states of the vehicles in the swarm. These state estimates are used in combination with reference data stored by this thread and adjusted through the user interface to compute control commands, which are periodically queried and transmitted by the wireless communication thread. A number of different estimators and controllers were implemented over the course of this project, however most actual tests performed with the AscTech Hummingbirds utilized simple low-pass filtering for position, velocity and heading angle estimation, and PID, PD or robust sliding mode controllers for position and yaw stabilization.

4.3 Quad-rotor Helicopter Control Software

Current implementations developed for testing of various aspects of communication and control utilize the AscTech Hummingbird quad-rotor helicopter, marked with one or more IR reflectors, in combination with the OptiTrack motion capture system. These helicopters are not programmable, and communicate using the XBee wireless modules. The software used by the central control computer to fly them is as described in the previous section, and as they cannot be programmed, no on-board software was developed for these vehicles.

The custom quad-rotor helicopters built for this project all utlized the ATMEGA168 micro controller, which has 16kB of flash memory used to store the on-board program. This program memory is sufficient for the programming of a helicopter to recognize a substantial number of commands, interface with simple sensors and the XBee, execute control computations, and configure the micro-controller. The on-board program consists of an initialization function, in which all initialization tasks are performed, and a looping operational function, which executes indefinitely, performing some operations as frequently as possible and others with fixed periods. Initialization is required for a number of the functions of the micro-controller, specifically analog-to-digital conversion, for which the ADC and input pins must be properly configured, PWM output, for which the pins and timers must be configured, UART serial communication, and I2C serial communication. In addition, the I2C sensors (accelerometer and magnetometer) must be configured at each power-up to operate at the desired sampling frequencies with the desired in-chip filtering options set. In addition to low-level configuration, calibration and sensor zeroing also occurs during initialization. The gyros and accelerometer are zeroed by taking repeated readings with the vehicle sitting on a flat surface and at rest, while the magnetometer zeroing requires the vehicle to be aligned in the 'zero-heading' orientation. Following initialization, the program enters the operational loop, which executes indefinitely, at each iteration checking for new data from the wireless link, on the UART serial line, taking appropriate action if new data is present. Following this, the program checks whether any of periodic tasks need to be executed. Periodic tasks inlcude reading sensors and updating internal estimates, and computing motor commands and setting PWM duty cycles. In the current implementation, the magnetometer is read at approximately 50Hz, the accelerometer at 100Hz, and the rate gyros at 500Hz, however, an alternate implementation has also been tested which uses interrupts for reading of the rate gyros, eliminating waiting during the ADC conversion and allowing the gyros to be read at 2kHz. In this implementation, the 'ADC Conversion Complete' interrupt function updates rate estimates stored in the main program, and begins another ADC conversion. The ADC clock rate can be adjusted by the program during micro-controller configuration to achieve the desired read rate. If interrupts are not used, it is advisable to select the maximum ADC clock rate, however if interrupts are used in this fashion, the maximum rate may cause excessive lag if even moderate levels of computation (such as LPF updates) are performed within the interrupt vector, in which case a slower ADC clock speed may be selected. Interrupts could also be used to eliminate waiting during reading of UART buffers and I2C sensor readings, but this has not been tested as the implementation is significantly more complicated. Motor control commands (PWM duty cycles) are updated at 100Hz, while the PWM frequency used is 500Hz. The overall structure of the non-interrupt-based implementation is shown in figure 4.2.

Interpretation of commands received from the control computer through the wireless link is an integral function of the on-board software, and a number of control commands are recognized. Each command packet consists first of a specific byte identifying the packet as a command from the control computer, followed by a byte indicating the type of command, which dicatates the number of bytes to follow, which comprise the data associated with the command. Following this are a command-specific terminating byte and a general packet terminating byte, used as a weak assessment of the validity of the packet. Stronger assurement of packet validity can be achieved using checksums, but this does not offer a noticeable peformance improvement in this implementation, as communication robustness requirements are not strict. A partial list of the commands recognized by the helicopter, and data format, are given in table 4.1. Several other, development-related commands are also recognized, including commands which interpret data as encoding floating point values and allow the remote setting of stability augmentation gains.

Command	Data	Description
Direct Motor Voltage Set	(voltage 1, 2, 3, 4)	Sets the PWM output compare registers associated with the motors directly to the voltage values received, in the range 0-255.
PRTY Command Set	(pitch, roll, thrust, yaw)	Sets higher level commands which can be used with or without stability augmentation. Pitch, roll, and yaw are interpreted as signed, in the range -128 to 127, while thrust is in the range 0-255. These values are used to compute motor commands with fixed period, as indicated in figure 4.2.
Display LED Set	(voltage R, G, B)	Sets the red, green and blue colour components of the display LED. These values are used in the periodic LED update.
Stability Augmentation Mode Set	(SA mode)	Sets the mode of stability augmentation used. Current options include: none; yaw only; pitch, roll and yaw. Can be set to 'none' to stop motors and abort operation.
Magnetometer Data Query	-	Queries the vehicle for filtered magnetometer data. Upon receiving this packet, the vehicle transmits a return packet containing (magnetometer x, y, z).
Accelerometer Data Query	-	Queries the vehicle for filtered accelerometer data. Upon receiving this packet, the vehicle transmits a return packet containing (accelerometer x, y, z).
Gyro Data Query	-	Queries the vehicle for filtered rate gyro data. Upon receiving this packet, the vehicle transmits a return packet containing (rate x, y, z).
Heading Angle Query	-	Queries the vehicle for estimated heading angle. Upon receiving this packet, the vehicle transmits a return packet containing (heading angle mapped to the range 0-255).
Zero Sensors	-	Commands the vehicle to perform the sensor zeroing operation, allowing the user to set the 'zero attitude state' during operation. Upon receiving this packet, the vehicle runs the sensor zeroing functions and stores zero state magnetometer, accelerometer and gyro data. The vehicle then transmits a packet indicating successful or failed completion of zeroing.
Start Magnetometer Calibration	-	Commands the vehicle to execute magnetometer calibration. Upon receiving this packet, the vehicle begins reading magnetometer data for calibration, during which time the user must rotate the vehicle through a full range of attitudes. This procedure lasts 10 seconds, after which time the vehicle writes the calibration values to EEPROM, and transmits a packet indicating successful or failed calibration.

Table 4.1: A description of the commands recognized by the quad-rotor on-board software. Many of these commands would not be used during normal operation, but may provide

4.3.1 Magnetometer Calibration

In addition to initialization, which must be performed to set the 'zero-heading' angle by taking repeated measurements from the magnetometer while the vehicle is in the desired orientation, it is also beneficial to perform a one-time calibration of the magnetometer before operation. This calibration requires the user to rotate the vehicle through a large range of possible orientations while the magnetometer is read. The resulting data is used to determine the sensitivity and output offset for each axis, which, in the simplest representation consist of a range of possible output values for each axis. Specifically, the maximum and minimum value for each axis are recorded, with the offset set to the average of these, and the sensitivity set to the absolute value of the difference. These values are then written to non-volatile EEPROM in the micro-controller, to be recovered during subsequent initializations and used to shift and scale future magnetometer readings so that each axis' output varies across a zero-centered range of approximately the same size. Testing suggests that this calibration is acceptable for the duration of a session of testing, but should probably be performed again if temperature, humidity, or magnetic field strength or direction changes.

4.3.2 Stability Augmentation and Control

A considerable amount research has been done towards developing an accurate description of the dynamics of quad-rotor helicopters [9], [25], as well as designing controllers to meet specific performance requirements including stabilization of hover or trajectory tracking for this type of vehicle using a variety of different sensor architectures [17], [19], [10]. While much of this work has focused on full position control, it also extends to lower level stability augmentation and control using on-board inertial and magnetic sensors such as those with which this quad-rotor prototype is instrumented. The following overview of the stability augmentation system proposed for use on this vehicle is, however, not based on this research in any way, though it is likely similar to work which has been done in the past and is not claimed to be a novel contribution.

While no successful flight tests were performed with the final quad-rotor prototype, due to a number of issues including unexpected hardware malfunction, a working version would likely be a viable helicopter capable of achieving sufficient yaw stability for automatic flight with human or motion capture feedback control of position. The flexible rotor construction affords a reasonable amount of passive stability in pitch and roll, which could be further augmented by accelerometer and gyro output, and the original commercial product from which the hardware is taken is capable of manual flight with yaw stability augmentation performed with rate gyros alone. Control commands are computed in pitch, roll, thrust, yaw (PRTY) coordinates, which are linearly transformed by equation (4.1) to give motor commands. In this equation pitch, roll, thrust and yaw are denoted by u_0, u_1, u_2 and u_3 respectively. Figure 4.3 indicates motor numbering and axis system used. It should be noted that these are control coordinates, and not related to the common euler angle representation of attitude. Pitch corresponds to a voltage difference across motors 0 and 2, resulting in rotational actuation about +z. The roll command corresponds to actuation about +x, and the yaw command to +y. Thrust is interpreted as expected, as the sum of motor voltages, resulting in actuation in the +y direction. This set of input coordinates are used to simplify the control laws used for attitude stability augmentation, as each of the pitch, roll and yaw commands corresponds to rotational actuation about a specific axis.

$$\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 1 & \frac{1}{2} & -\frac{1}{2} \\ -1 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & -1 & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} \tag{4.1}$$

Filtered sensor outputs can be used directly for stability augmentation, eliminating the use of attitude and attitude rate estimation and reducing computational load on the micro-controller, as in the control law defined by equations (4.2). Under this control law, voltage differences are applied to the motors so as to drive the magnetic field vector, B, towards the zero-orientation magnetic field vector, B_0, and similarly for the accleration vectors A and A_0, by using the cross product between the initial, zero attitude vector and the vector at the current time. With simplified dynamics in which voltage control corresponds instantaneously to control of force, i.e. voltage has a static, monotonic relation to rotor thrust output, vector cross product control such as this, when combined with rate damping, can be analytically shown to stabilize the system to alignment of the vector with the reference. In addition to the terms related to these cross products, rate damping terms are also included, and the yaw axis gains are different from those of the other two axes. During flight, these pitch, roll and yaw commands (u_0, u_1 and u_3 respectively), are added to the commands provided across the wireless link, which may be provided directly by the user during manual flight, or computed for automatic position feedback control by the swarm computer.

$$\begin{aligned}
\vec{v}_{mag} &= \vec{B}_0 \times \vec{B} \\
\vec{v}_{acc} &= \vec{A}_0 \times \vec{A} \\
u_0 &= k_m v_{mag,z} + k_a v_{acc,z} - k_d \omega_z \\
u_1 &= k_m v_{mag,x} + k_a v_{acc,x} - k_d \omega_x \\
u_3 &= k_{m,yaw} v_{mag,y} - k_{d,yaw} \omega_y
\end{aligned} \tag{4.2}$$

It is important to note that during free flight, the accelerometer feedback used in equations (4.2) will not be as effective as if the position of the vehicle is constrained, because measurement is of net acceleration rather than the direction of gravity. This feedback control law will still provide some degree of stabilization, due to acceleration caused by aerodynamic drag, but the effect will likely be significantly reduced compared to that observed if the accelerometer was able to measure the true direction of gravity.

4.4 Monocopter Control Software

The on-board software developed to control the monocopter prototype is fundamentally very similar to that used to control the quad-rotor prototype. The general structure is the same, with an initialization function and an infinite loop operation function, however, because of the differences in electronic hardware, there are some differences in the software. Firstly, the ANT radio transceiver requires configuration that must be performed during initialization, following configuration of the I2C pins of the micro-controller. This initialization requires the setting of the network ID, the communication band (frequency), the transmission power, and the transmission period for the transceiver. There are no I2C sensors on this prototype, with the 3-axis I2C magnetometer used on the quad-rotor replaced with a 2-axis analog version, and all other sensor eliminated. This magnetometer does not require initialization, but utilizes the same ADC initialization as for the quad-rotor. Additionally, because the RGB LED is controlled directly from the pins on the microcontroller, 5 pins must be configured for PWM output as compared with 4 for the quad-rotor, and these pins are not necessarily the same.

4.5 Monocopter Lateral Actuation

The monocopter prototype developed for this project has only two control inputs: the PWM duty cycles of the voltage signals to the two motors. Because the PWM frequency is 500Hz, this can effectively be thought of as analog control of the voltages applied to the motors, ranging from 0V to the voltage provided by the single LiPO chemical cell, which is typically in the range of 3.7-4.2V. Vertical actuation of the monocopter can be easily achieved by controlling the sum of these motor voltages, and lateral actuation can be achieved using the difference between motor voltages. Instantaneously, the vehicle is non-holonomic, and can be actuated only along a single axis defined by alignment of the motors and the rotational rate. For simplicity this will be considered to be the axis nearest in alignment to the rotational axes of the two motors, orthogonal to the spar supporting the motors. As the vehicle rotates, the axis along which it can be actuated rotates, affording an effective holonomy over time scales significantly longer than the rotational period. The rotational period of the monocopter prototype was not precisely measured during the course of this project, but it estimated to have a rotational frequency of approximately 5Hz, from which one can reasonably assume that reference trajectories with spectral content faster than approximately 1Hz would not be well tracked by a controller which attempts to utilize this effective holonomy for actuation. However, in effect this is most likely made redundant by the extremely slow lateral dynamics of the physical design.

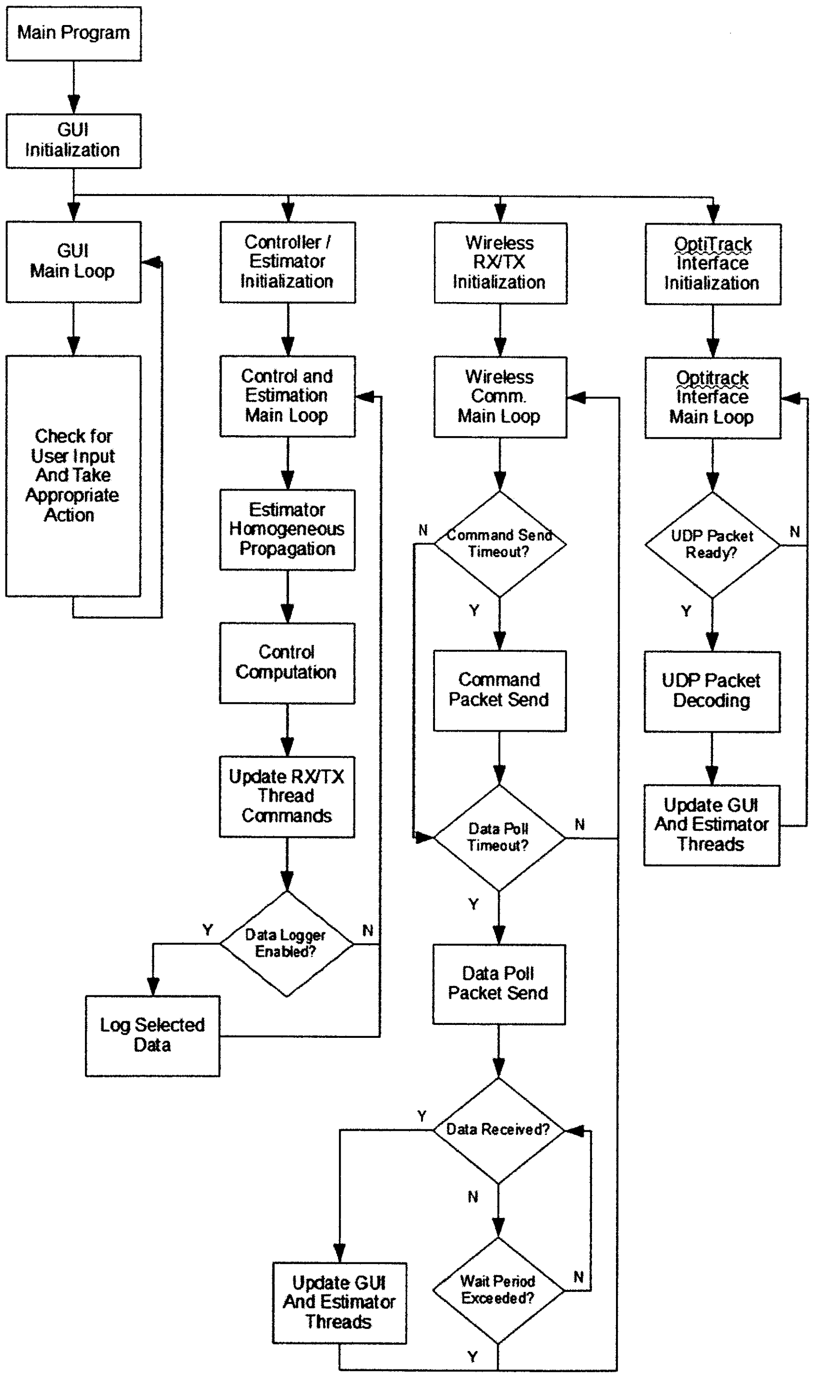
During preliminary testing, consistent lateral directional actuation has been observed under the action of the controller defined in equation (4.3). Quantities related to this equation are depicted in figure 4.4. While this result does demonstrate the ability of the vehicle to be laterally actuated, there are a number of issues that have not been addressed that may cause difficulties with lateral position control, including the rotation-rate-dependence of the dynamics transferring the motor voltage difference to the force seen by the helicopter. With the vehicle at zero yaw rate, the net lateral force can be reasonably assumed, by symmetry of the vehicle, to be aligned orthogonal to the spar on which the motors are held, but as the yaw rate increases, the dynamics between the voltage difference and the force direction becomes more significant in influencing the direction of the applied force. The approximately sinusoidal variation in voltage difference that arises from equation (4.3) can be shifted in phase by rotating the reference vector , W, by a fixed angle representative of the lag due to the rotational rate observed during typical operation, or perhaps by a time-dependent correction angle based on an open-loop estimate of the rotational rate. Multiplication of the original reference vector by the rotation matrix $R(\dot{\theta})$, as in equation (4.3), attempts to account for these dynamics in this way. In order to achieve point tracking, the reference vector must also vary in time, with $W = [-x, -y]^T$ for regulation to $x, y = 0$. Simulation results show that if the dynamics between the voltage difference and the force are known, and the vehicle operates in the presence of viscous damping, this control law will stabilize position. It should also be noted that the quantities $\sin\theta$ and $\cos\theta$ would be proportional to the ideal outputs of the magnetometer in x and y respectively, so it is possible to implement this controller directly from the magnetic field values, and in fact it is this law that has been implemented in testing.

$$W_1 = R(\dot{\theta})W \quad v_{diff} = k(\sin\theta W_{1,x} + \cos\theta W_{1,y}) \tag{4.3}$$

4.6 Swarm Positions Estimation

Simultaneously estimating the positions of all vehicles in the swarm from the measurement of three dimensional feature locations (the point cloud), has several issues that must be considered in order for any such system to operate effectively and robustly. Each vehicle is equipped with only a single IR LED or reflector, and as such the vehicles cannot be differentiated from one another based solely on the point cloud output. While it is possible to use multiple markers on each vehicle, arranged in a unique configuration that can be used by the proprietary motion capture software to identify the vehicles, this approach would likely impose unacceptable limitations on the maximum swamrm size, and additionally, experience would suggest that the reliability of such an implementation would be unacceptably low. Assuming continuous, relatively high bandwidth operation of the optical tracking system during performances, it is possible to automatically associate markers with vehicles, assuming some initial estimate of vehicle positions based on a semi-manual assignment of vehicles to markers. Initially, the vehicles are laid out on the ground, with the tracking system observing a number of markers. Assuming that at this point, each vehicle's marker is observed, the vehicles can be assigned to positions by manually associating each vehicle to a marker. As the system proceeds to operate, measurements are made by the motion capture system that provide the user software with a cloud of marker positions, and the problem then is to associate vehicles in the swarm with markers from each new frame of marker position data. It must be assumed that some extraneous markers are normally observed, corresponding to no vehicle's reflector, and it should also be assumed that there is some possibility of a reflector not being successfully observed by the motion capture system in every frame. One possible algorithm for this assignment would attempt to minimize the squared error, summed over all vehicles, between the new assigned marker the estimated position of the vehicle to which it is assigned, while disallowing any specific vehicle-marker pairings for which the error is above a pre-determined threshold (assuming that vehicle is not observed in that frame). In general this is a very difficult problem for even moderately sized swarms, due to the combinatorial explosion of the number of possible assignments, however, a desireable result could likely be achieved by first making the optimal assignment for each vehicle (i.e. assigning each vehicle to the nearest marker), and searching over possible resolutions to this assignment in the case that multiple vehicles are assigned to the same marker. While this approach will not guarantee better performance than an exhaustive search over all possible assignments, in an application such as this it will almost surely afford a large improvement in tractability, as it is likely that the initial assignment will be either globally optimal or require only a small number of reassignments to become so. Naturally if stochastic estimation is used in which the covariance or higher probability moments of the position estimate are maintained, the appropriate weighting of position error will generally be non-isotropic in space, and in some cases, such as with normally distributed stochastic position estimates, it may be possible to analytically compute the probability of each marker corresponding to a given vehicle, and make the assignment based on this.

Tests in which multiple quad-rotor helicopters were flown together using single point tracking with magnetometer based heading estimation used this position estimation scheme, in which an initial assignment is made manually and the high output frequency of the optical tracking system allows the assignment to be automatically, correctly maintained by simply assigning each vehicle to the nearest observed marker. The current implementation does not account for the possibility of multiple vehicles being assigned to the same marker, though this did not result in a detriment to functionality, as the inter-vehicle spacing maintained was always significantly higher than the maximum relative change in position achievable over the duration of a single motion capture frame.



Main Program
GUI Initialization
GUI Main Loop
Check for User Input And Take Appropriate Action
Controller / Estimator Initialization
Control and Estimation Main Loop
Estimator Homogeneous Propagation
Control Computation
Update RX/TX Thread Commands
Data Logger Enabled?
Y
N
Log Selected Data
Wireless RX/TX Initialization
Wireless Comm. Main Loop
Command Send Timeout?
N
Y
Command Packet Send
Data Poll Timeout?
N
Y
Data Poll Packet Send
Data Received?
Y
N
Update GUI And Estimator Threads
Wait Period Exceeded?
N
Y
OptiTrack Interface Initialization
Optitrack Interface Main Loop
UDP Packet Ready?
N
Y
UDP Packet Decoding
Update GUI And Estimator Threads

Figure 4.1: Program flow diagram illustrating the structure of the swarm control software used for flight testing, in which four threads are executed in an interleaved, approximately concurrent fasion. Information is shared between the different threads, with the Optitrack interface and radio communication threads sending updates to the GUI and control/estimation thread when new data is received, and the control/estimation thread updating the radio communication thread with new command data when new controls are calculated.

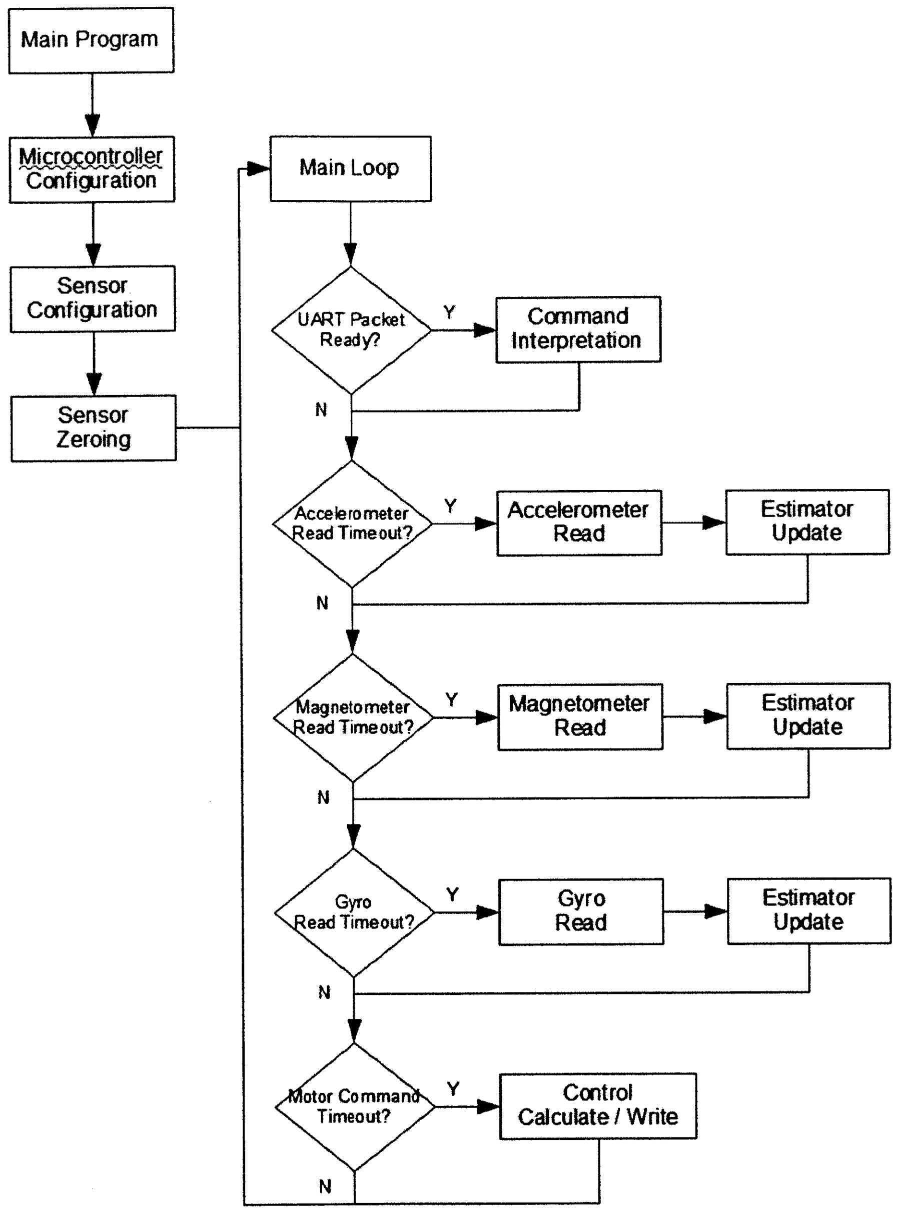


Figure 4.2: Program flow diagram illustrating the structure of the software run on the on-board ATMEGA168 micro-controller of the final quad-rotor helicopter prototype. In this structure, following initialization, the program loops indefinitely, checking for radio (UART) communication packets, and reading sensors and setting motor commands at approximately fixed intervals. Alternatively, interrupts may be used for sensor reading, with the I2C sensors (accelerometer and magnetometer) initiating the read procedure at fixed intervals but using interrupts to allow other operations during wait periods. The gyroscopes may also be read in this fashion, or the ADC clock set to the desired speed that the gyros may be read as quickly as possible - i.e. with the ADC conversion complete interrupt function used to start another conversion.

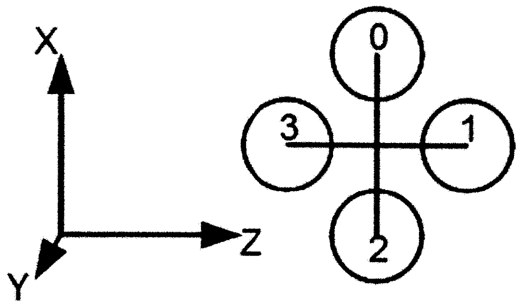


Figure 4.3: Quad-rotor axis definition and motor numbering used for automatic stability augmentation.

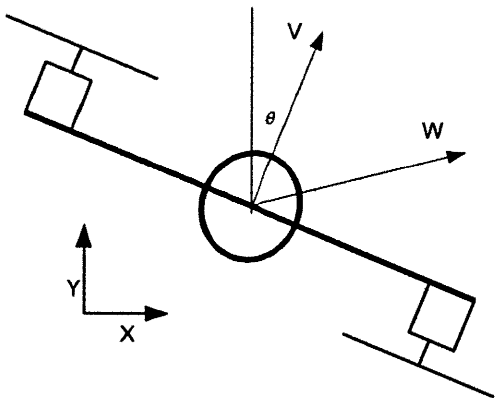


Figure 4.4: Definition of relevant quantities for lateral actuation of the monocopter design. Overhead view, indicating the body axis vector, V, orthogonal to the motor support spar, and the reference vector W.

Chapter 5

Aerodynamic Interference Modeling and Compensation

One of the most obvious and daunting challenges associated with the implementation of such a system is the effect of aerodynamic interference between different vehicles in the swarm. This effect is particularly noticeable when a helicopter attempts to stabilize its position directly in the wake of another, often resutling in highly erratic and undesireable behaviour. This chapter describes the development of a low-order wake disturbance model, and a controller that utilizes this model to account for the effect of the wake and improve control performance for a pair of quad-rotor helicopters flying in vertical alignment. This chapter also includes the presentation and discussion of simulation and experimental results relevant to this problem.

5.1 Wake Flight Problem Introduction

In general, the wake flight problem is that of designing a controller for a vehicle in a swarm, which utilizes information about the positions of other swarm vehicles to compensate for the effects of aerodynamic interference and improve trajectory tracking and position stabilization. While it is theoretically possible to design robust controllers that provide increased tolerance to unknown, bounded disturbances such as those caused by wake interference, the improvement offered by such controllers is inherently limited in implementation by effective gain limits caused by sensor and actuator noise. However, such a swarm implementation, with centralized estimation of the positions of all swarm vehicles, offers the opportunity to utilize information regarding other helicopters in the swarm to reject aerodynamic interference. The simplified version of this problem addressed in this thesis is that of designing a controller to stabilize the position of a single quad-rotor helicopter directly below another, identical helicopter which is itself hovering in place, stabilized by a similar controller. A simplified model of the quad-rotor dynamics is also used, which allows the thrust output of the rotors to be controlled directly, and assumes an isotropy that allows the 3-dimensional problem to be effectively resolved by the natural extension from a resolution to the 2-dimensional problem.

Previous tests with multiple quad-rotor helicopters controlled in synchrony using simple proportional-integral-derivative (PID) or sliding mode controllers, in which the helicopters were commanded to stabilize their positions in vertical alignment, have resulted in highly undesireable behaviour by the lower helicopter, which can be described qualitatively as an erratic lateral orbiting of the reference point in combination with vertial oscillation caused by entry and exit from the wake. Figure 5.1 shows the actual and reference position trajectories for a quad-rotor helicopter attempting to stabilize its position directly in the wake of another quad-rotor, without additional wake compensation, under the action of a simple PID position controller in addition to unknown on-board attitude stability augmentation.

Figures 5.2 and 5.3 depict the quad-rotor wake flight problem and variables associated with the simplified 2-dimensional dynamic model. Equations (5.1) describe a simple model of the dynamics of the quad-rotor in two dimensions, with inputs u_{diff} and u_{sum} denoting the difference and sum of rotor thrust outputs respectively. While angular rate damping due to aerodynamic drag is represented, velocity damping due to drag is neglected in this model for conciseness. The disturbances due to wake interference are represented by the functions $f_1(x, y)$ and $f_2(x, y)$, which are described in subsequent sections.

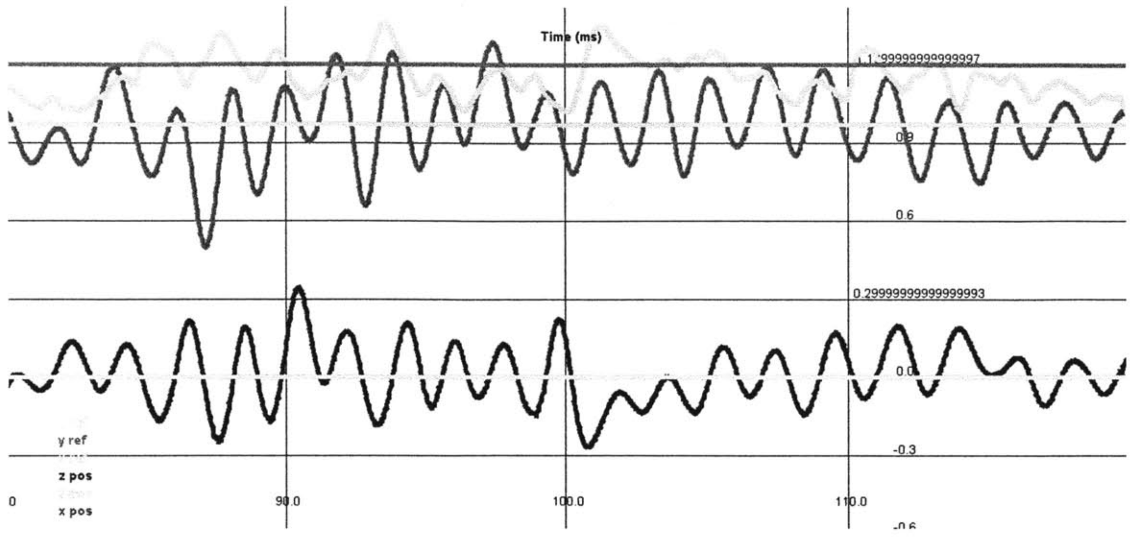


Figure 5.1: Position coordinates, in m, of an AscTech Hummingbird quad-rotor helicopter feedback controlled by the OptiTrack motion capture system and on-board magnetometer output, with PID control and no wake compensation, in the presence of wake interference, commanded to hover approximately 75cm directly below an identical quad-rotor.

$$\begin{aligned} J\ddot{\theta} &= u_{diff} - b_{\omega}\dot{\theta} + f_1(x, y) \\ m\ddot{x} &= (u_{sum} - f_2(x, y))\sin\theta \\ m\dot{y} &= (u_{sum} - f_2(x, y))\cos\theta - mg \end{aligned} \tag{5.1}$$

5.2 Quad-rotor Wake Disturbance Model

While a considerable amount of work has been done to characterize propeller downwash in fine temporal and spatial detail [18],[5],[28], the objective of this design is to obtain a controller requiring minimal computational power to implement, and it is hypothesized that a significant improvement in performance may be obtained by an extremely simple model of wake interference. It is also important to note that modelling of the overall effect of the interference is the goal, which may not correspond to accurate modelling of the helicopter downwash in isolation. The simple model used for feed-forward controller design computes

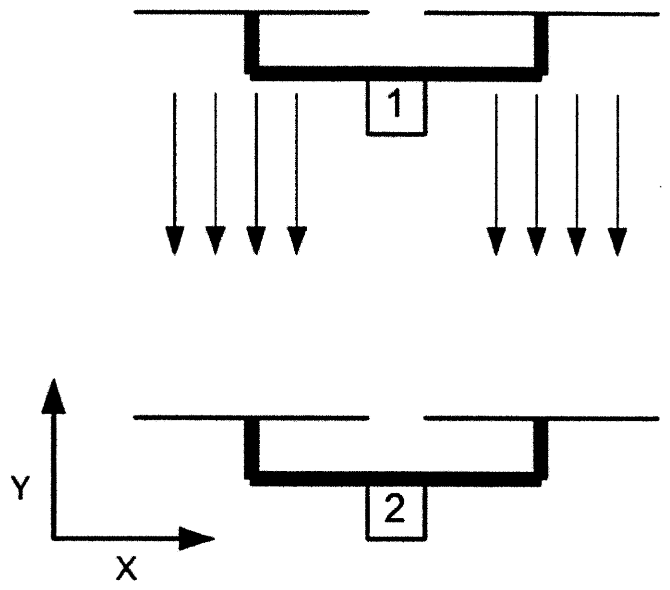


Figure 5.2: Depiction of the simplified wake interference condition and stabilization problem for quad-rotor helicopters. The objective of the problem is to design a controller which stabilizes vehicle 2 directly in the downwash of vehicle 1.

the downward velocity of a theoretical wake profile at the centre of each rotor, the effect of which is then modelled as a reduction in thrust of each of the propellers, proportional to the square of the corresponding downwash velocity. In this model, because the wake velocity effects the helicopter only at the propeller centers, rather than exerting an effect integrated over the area of each propeller, the wake profile used must approximate the effect observed by integration of the velocity profile over the propeller discs. An intuitive approximation of the actual wake profile may be the sum of four gaussian velocity profiles centred at each of the disturbing helicopter's propellers, or perhaps the sum of four square velocity profiles wherein the wake velocity takes on some value directly below each propeller and is zero otherwise, however, when modelled as effecting the disturbed helicopter only at the rotor centers, these models become less accurate, and the disruptive effect of the wake will not be observed. Alternatively, if the wake is envisioned as having a single gaussian velocity profile centered at

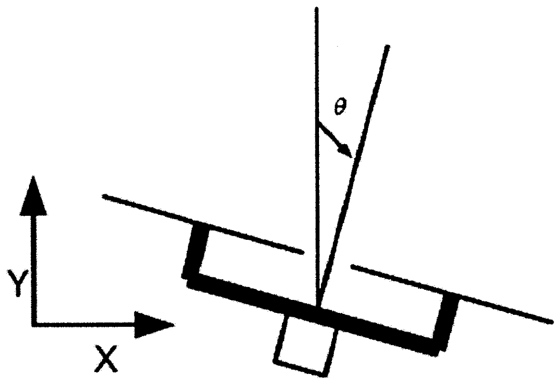


Figure 5.3: 2-Dimensional quad-rotor helicopter model variable definition diagram, showing the inclination angle θ, the vertical position coordinate y, and the lateral position coordinate x.

the geometric center of the disturbing vehicle, the effect of which on the disturbed vehicle is computed at the rotor centres, deviation from perfect vertical alignment of the vehicles causes a substantial torque disruption on the lower vehicle. Comparison of experimental observations with simulation results also suggests that this model captures the important aspects of the wake disturbance effect. Figure 5.4 shows the gaussian squared velocity profile used to model the disruptive effect of the wake.

In addition to the simple deterministic wake profile used in this model, there may be significant components of the wake velocity profile that cannot be determined from the information measured during operation, and are approximated as stochastic in nature. The presence of such random disturbances will undoubtedly limit the degree of improvement that can be achieved using the simple model, and it is one of the primary goals of this research to explore the nature of the wake disturbance effect, and approximately determine the proportion of the effect that can be compensated for using a disturbance model such as this.

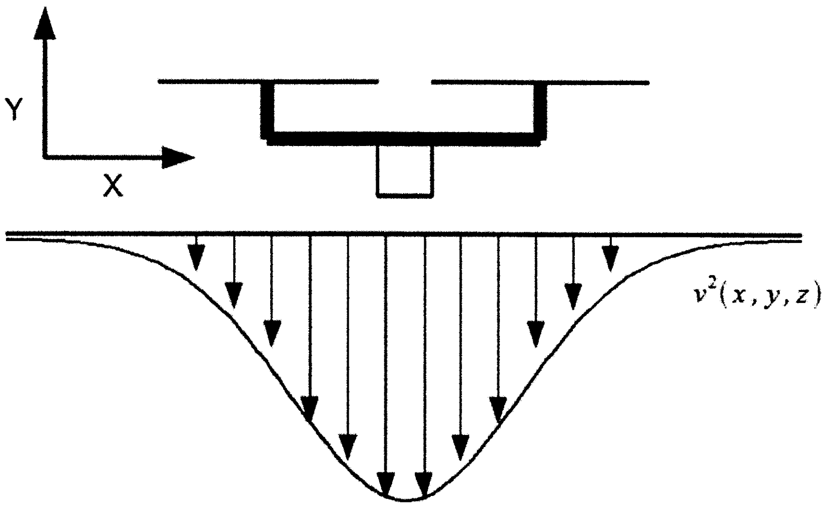


Figure 5.4: Wake disturbance downwash profile used to compute thrust distrubances at the rotor centres of the disturbed helicopter, modelled as a gaussian squared velocity profile.

$$\begin{aligned} v^2(x,y,z) &= \frac{mg}{2\pi\sigma\rho_{air}} e^{\frac{-x^2-z^2}{2\sigma}} \\ \sigma &= \sigma_0(1-k_\sigma y) \end{aligned} \tag{5.2}$$

$$F(x,y,z) = \frac{1}{2}\rho_{air} v^2(x,y,z) c_d \tag{5.3}$$

Equations (5.2) and (5.3) describe the effect of the upper helicopter's wake on the lower helicopter, with the force given in (5.3) acting on the disturbed helicopter at each of the rotor centres. The coordinates x, y and z are taken relative to the disturbing helicopter, and (5.2) gives a 'spread', σ, of the airflow, increasing linearly with distance. The squared velocity of the airflow is normalized by the applicaton of conservation of momentum, with the total downward momentum flow of the downwash equal to the force (mg) required for the disturbing helicopter to maintain hover. This model does not account for varying thrust, or differential rotor thrust from the upper vehicle.

5.3 Feed-forward Wake Compensation

This section outlines the development of a control law to provide adequate trajectory tracking capabilities for a quad-rotor helicopter with on-board attitude stability augmentation instrumentation and software, as well as rejection of aerodynamic disturbances caused by the wake of another identical quad-rotor. While basic PID control is sufficient to stabilize the position of such a platform, assuming sufficient stability augmentation, and to eliminate steady state position error caused by variations in motor thrust, deviation of the centre of mass or other constant disturbances, accurate trajectory tracking is more readily achieved by a robust, sliding mode controller, which also allows the natural incorporation of the partially known aerodynamic disturbance terms.

Equation (5.1) gives the disturbed dynamics of the vehicle, with a lateral torque disturbance denoted as $f_1(x,y)$. Assuming the exact disturbance is unknown, but an approximation, $\hat{f}_1(x,y)$ is used, for which the error is known to be bounded by the function $D_1(x,y)$, as in (5.4), which also gives the form of the disturbance approximation based on the downwash model previously described. In this equation, the quantity r denotes the distance from the geometric centre of the vehicle to the centre of a given rotor, assumed to be equal for all rotors.

$$\begin{aligned} &|f_1(x,y) - \hat{f}_1(x,y)| \leq D(x,y) \\ &\hat{f}_1 = \hat{c}_d(v^2(x+r,y) - v^2(x-r,y)) \end{aligned} \tag{5.4}$$

Proceeding with robust (sliding mode) controller derivation, with feedback linearization, the state is repeatedly differentiated until the desired form appears,

$$\begin{aligned} x^{[4]} &= \frac{u_{sum}}{m}(\ddot{\theta}\cos\theta - \dot{\theta}^2\sin\theta) \\ &= \frac{u_{sum}}{m}(\frac{u_{diff} - b\dot{\theta}}{J}\cos\theta - \dot{\theta}^2\sin\theta + \frac{f_1(x,y)\cos\theta}{J})) \end{aligned} \tag{5.5}$$

then choosing intermediate input u_0 and disturbance f_{11} gives the dynamics in a form suitable for robust controller design,

$$\begin{aligned} u_0 &= \frac{u_{sum}}{m}(\frac{u_{diff} - b\dot{\theta}}{J}\cos\theta - \dot{\theta}^2 \sin\theta) \\ f_{11}(x,y) &= \frac{u_{sum}}{mJ}\cos\theta f_1(x,y) \\ x^{[4]} &= u_0 + f_{11} \end{aligned} \tag{5.6}$$

Introducing the sliding variable s, for which $s = 0$ defines a surface of stable dynamics when $s = (\frac{d}{dt} + \lambda)^3$, with $\lambda > 0$, then choosing control input to drive the state of the system toward the sliding surface, ie. to satisfy $s\dot{s} \leq \eta|s|$, with the reference trajectory of the system defined by $x_r(t)$, gives the control law defined by equation (5.7). Here $v^2(x,y)$ is as defined in equation (5.2).

$$\begin{aligned} u_{diff} = \frac{J}{\cos\theta}(\frac{m}{u_{sum}}(\frac{\hat{c}_d u_{sum}\cos\theta}{mJ}(v^2(x+r,y) - v^2(x-r,y)) + x_r^{[4]} - 3\lambda\tilde{x}^{[3]} - 3\lambda^2\ddot{\tilde{x}} - \lambda^3\dot{\tilde{x}} \\ - (\frac{u_{sum}\cos\theta}{m}D(x,y) + \eta)sign((\frac{d}{dt} + \lambda)^3)\tilde{x}) + \dot{\theta}^2\sin\theta) + b\dot{\theta} \end{aligned} \tag{5.7}$$

Equation (5.7) requires $\eta, \lambda > 0$, and uses the quantity $\tilde{x} = x - x_r$. Note that this control depends on the quantity $u_s um$, the sum of control inputs, which is given by the control law in equation (5.8). While this derivation is not strictly applicable to the case where this quantity is variable, it can be assumed to be approximately equal to mg during much of operation, so the resulting controller would most likely provide acceptable performance. Though (5.7) appears relatively complicated, it is essentially the standard sliding mode controller, with the addition of a feed-forward term that attempts to counteract the effect of wake disturbance on the vehicle, as might be intuitively expected. Here the lateral directional control is considered in isolation, and a similar approach may be applied to vertical directional control, assuming simple double-integrator dynamics for the vertical direction. The result of this derivation is given as the control u_{sum} in equation (5.8).

$$u_{sum} = mg + \hat{c}_d(v^2(x+r,y) + v^2(x-r,y)) + m(\ddot{y}_r - \lambda_y \dot{\tilde{y}} - (\tfrac{D_y(x,y)}{m} + \eta_y) sign(\dot{\tilde{y}} + \lambda_y \tilde{y})) \tag{5.8}$$

The controllers shown in equations (5.7) and (5.8) give the general form of full-state feedback controllers used to drive the system described by equation (5.1), which, as written, assumes that the force output of the propellers can be controlled directly, and that the full state is available for measurement. In simulation these control laws are implemented exactly, while in experimental implementation there were a number of issues with the direct implementation of these controllers. Firstly, the dynamics do not allow for the direct control of motor thrust output, and have a number of unknown parameters, namely the inertia of the vehicle, J, and the lift-loss coefficient, c_d associated with wake-induced propeller lift loss. Additionally, the test platform used, the AscTech Hummingbird, has an onboard controller that provides attitude stability augmentation, and only position and velocity information is available to the external controller for feedback. Thus, while the general form of the sliding mode controller with feed-forward compensation was maintained in experimental testing, the lateral position controller was second order, similar to the vertical axis controller described here, and gains were tuned manually to achieve acceptable performance. Also, no tests were performed which required pre-determined, time-varying trajectories to be tracked, so all terms containing derivatives of position error were eliminated in implementation. Additionally, the $sign()$ function resulting from this derivation was replaced with a saturation function $sat()$, that varies linearly, continuously between -1 and +1 over some range. This serves the purpose of reducing chatter across the sliding surface, as the control input varies continuously in the state space, rather than having a discontinuity across the sliding surface that can cause undesireable, high frequency oscillations in the input.

5.4 Simulation Testing and Results

A number of simulation tests were conducted over the course of this research. Firstly, support for the wake disturbance model used was gleaned through simulation of the disturbed dynamics, which resulted in behaviour very similar to that observed in previous instances of experimental testing. Figure 5.6 shows simulation results from a simulated quad-rotor helicopter executing a step change in reference position, with the disturbing helicopter directly above the final reference location, and for comparison, the same controller and task is shown in figure 5.5, in the absence of wake interference. This result is significant, in that when compared to the performance of a similar controller (PID with unknown attitude stability augmentation), in the presence of aerodynamic interference, the result is a similar kind of oscillatory limit-cycling, as seen in 5.1. The effect of interference in the experimental test is shown by comparison to the case with the same controller in the absence of interference, shown in 5.10.

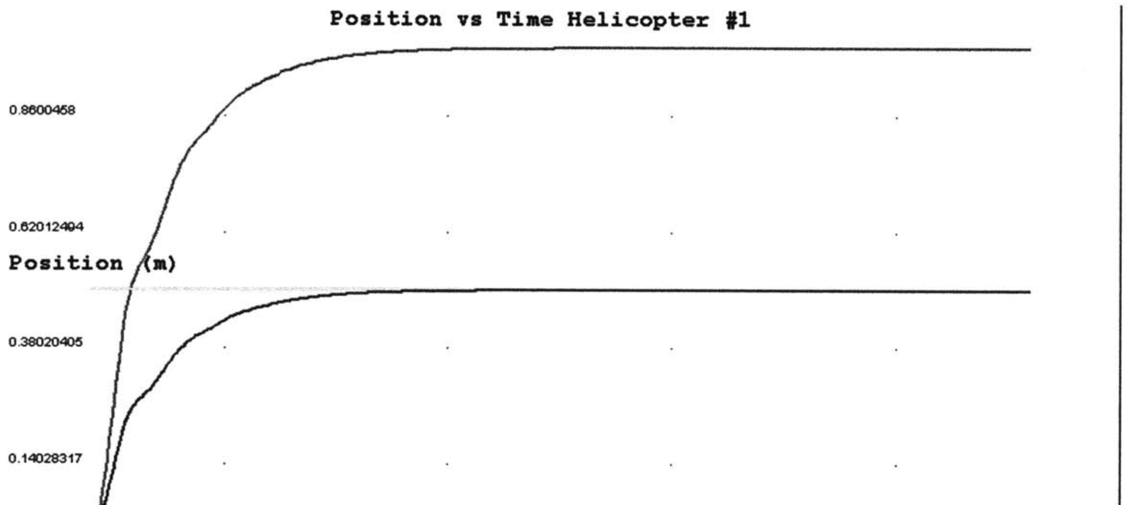


Figure 5.5: Step response of simulated automatically controlled quad-rotor helicopter with full state feedback. Control gains tuned manually through trial and error.

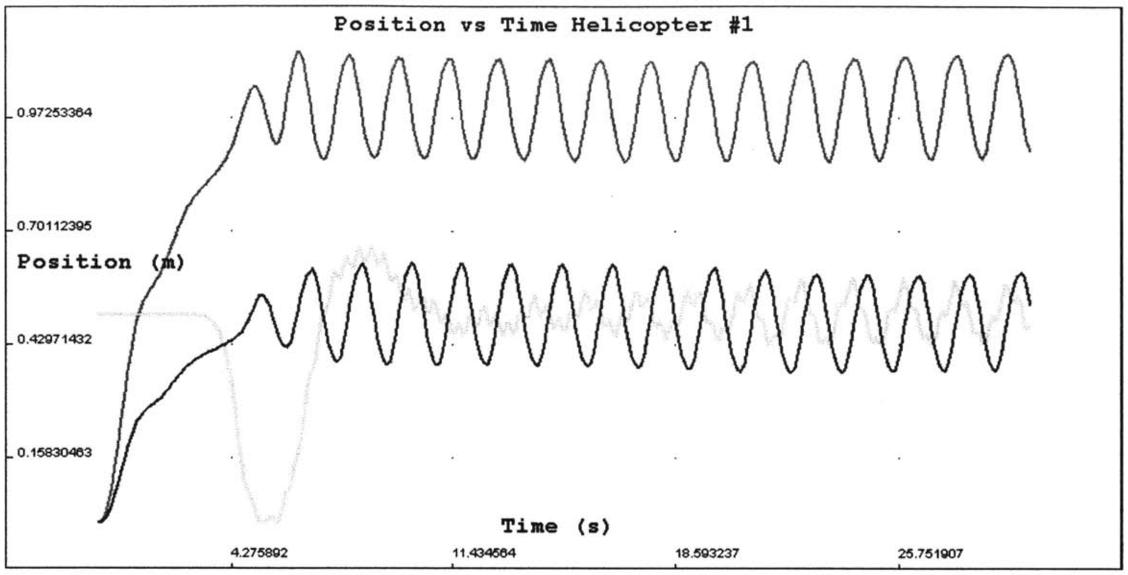


Figure 5.6: Step response of simulated automatically controlled quad-rotor helicopter with full state feedback, in the presence of aerodynamic interference, with the disturbance originating above the final reference point. Control gains are the same as those used to obtain 5.5

5.5 Experimental Testing and Results

A number of test flights with one or multiple quad-rotor helicopters were conducted, in order to test software structure, estimation and control algorithms, and to generate demonstration videos for promotional and sponsorship purposes. The setup used to test controllers designed for the rejection of wake interference is shown in figure 5.7, in which one of the quad-rotors was mounted to the ceiling at a height of approximately 2.2m, and the other quad-rotor commanded to hover directly in the wake. During testing, the thrust level of the disturbing helicopter was set to match that used to achieve hover.

5.5.1 OptiTrack Motion Capture System

All tests were conducted using the OptiTrack motion capture system, which, as implemented, utilizes an array of thirteen IR cameras, each equipped with a digital signal processor (DSP) to perform image processing tasks. The camera array interfaces to a host computer across USB, and provides frame information consisting of a list of pixel locations corresponding to identified markers for each camera. Proprietary software provided with OptiTrack, then utilizes calibration profile information to reconstruct a list of three dimensional points supposed to be associated with markers. This 'point cloud' of marker locations can then be broadcast over the local network, utilizing TCP/IP or UDP as selected by the user, by the OptiTrack software. The software also has the capability to track 'rigid bodies' consisting of at least 4 markers in a rigid configuration, and transmit only the positions and attitude quaternions of these bodies over the network, however, tests have shown that this is relatively error prone, and often unacceptable for automatically controlled flight. Figure 5.8 shows one of the thirteen IR cameras placed around the capture volume, that are capable of tracking small spherical reflectors placed on the vehicles.

Calibration of the tracking system is a relatively straightforward and fast procedure, though it is prone to failure and often requires a number of attempts using different camera and DSP parameters. Following placement of the cameras, a single marker is moved throughout the capture volume and used to generate calibration data. Once the relative positions and orientations of the cameras are known, a right triangle of precise dimensions, consisting of three markers is used to identify the ground plane and origin of the coordinate system.

5.5.2 Ascending Technologies Hummingbird

All flight testing with automatic position feedback control and aerodynamic interference compensation was conducted using the Ascending Technologies Hummingbirds [29] - quad-rotor helicopters approximately 40cm in diameter, including rotor discs. The Hummingbird is driven by four brushless DC motors controlled by individual electronic speed controllers connected to an electronics stack called the Autopilot, which contains 3-axis inertial and magnetic sensors, pressure altimeter and GPS module, as well as a micro-controller. It is capable of communicating using standard RC matched crystal signals, or through a transparent wireless serial link provided by an onboard XBee Pro. During operation through the XBee link, the helicopter must continue to receive valid commands above 10Hz, or automatic shutdown will be triggered, and the XBee link can only be used to operate the vehicle if a valid RC signal is present, with one channel dedicated to toggling of the control between XBee and RC radio. Figure 5.9 shows the Hummingbird next to the version 2 quad-rotor prototype developed for this project.

5.5.3 Testing and Results

A number of tests were run in which the helicopters were feedback controlled for hover either individually or as a pair. For initial testing of general system functionality, both quad-rotors were flown together in synchrony, and commanded to follow simple reference trajectories using controllers that did not account for these trajectories in advance. The problem of wake interference was then addressed using the test setup shown in figure 5.7, with which PID, sliding mode, and sliding mode with feed-forward compensation controllers were tested. Results are shown in figures 5.10, 5.11, 5.1, 5.12, and 5.13, which show baseline PID and sliding mode controllers without interference, the same controllers with interference, and the sliding mode controller with feed-forward compensation in the presence of interference, respectively. In the case where feed-forward wake compensation is implemented, the feed-forward gain was coarsely manually adjusted to optimize performance. In all tests, single point tracking was used, with magnetometer data retrieved from the on-board computer used to execute sliding mode control of the yaw angle as well. This process required a one-time manual yaw-angle offset to be adjusted immediately after takeoff, due to a tendency for a significant steady-state yaw error to appear at this point.

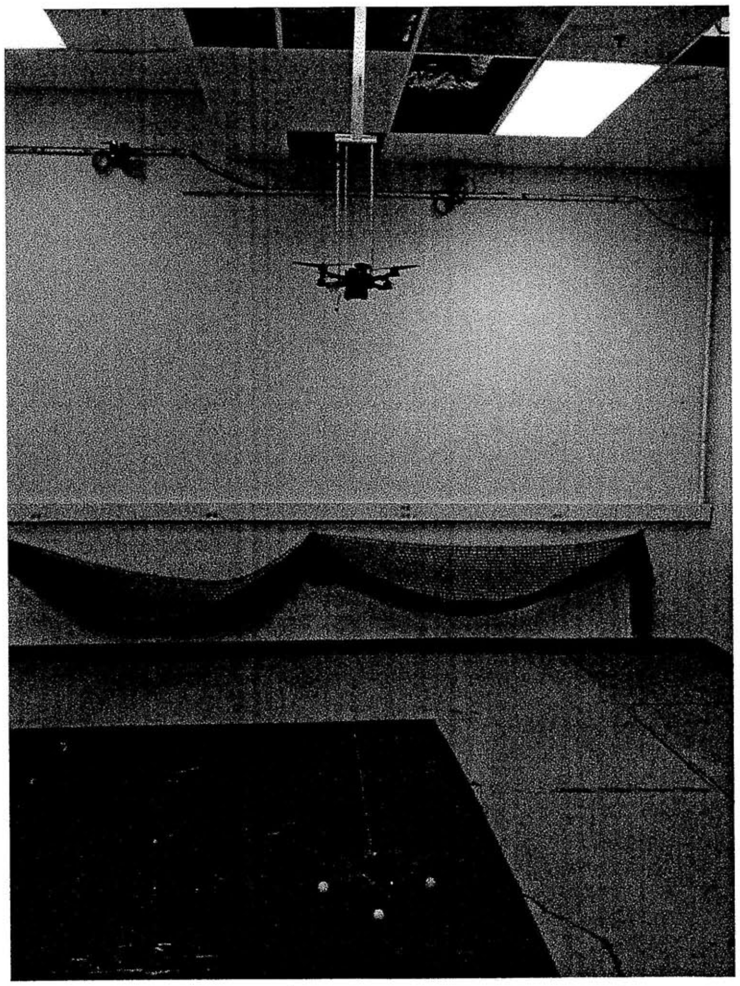


Figure 5.7: Setup used for testing of wake interference compensation, showing two Ascending Technologies Hummingbird quad-rotor helicopters, with the upper unit mounted at a height of 2.2m. Cameras used by the OptiTrack motion capture system are also seen along the top edge of the rear wall.

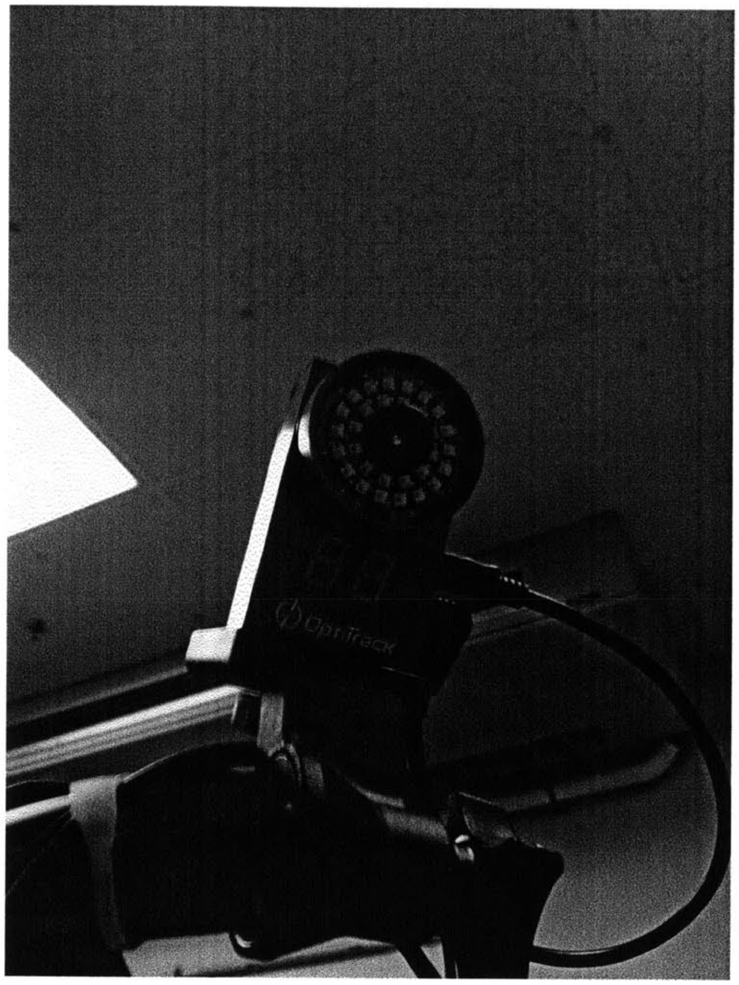


Figure 5.8: OptiTrack motion capture camera, mounted along the periphery of the capture volume. A system consisting of 13 such cameras provides position feedback control for vehicles in the test area.

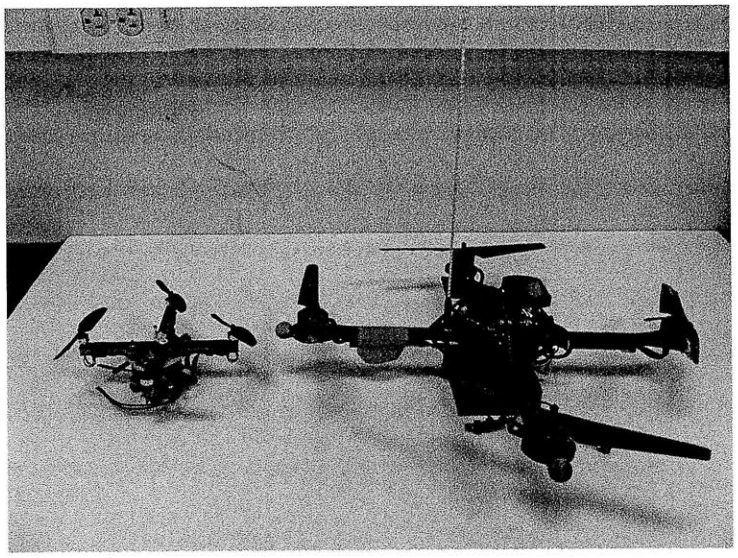


Figure 5.9: Ascending Technologies Hummingbird (right), with version 2 prototype developed for this project.

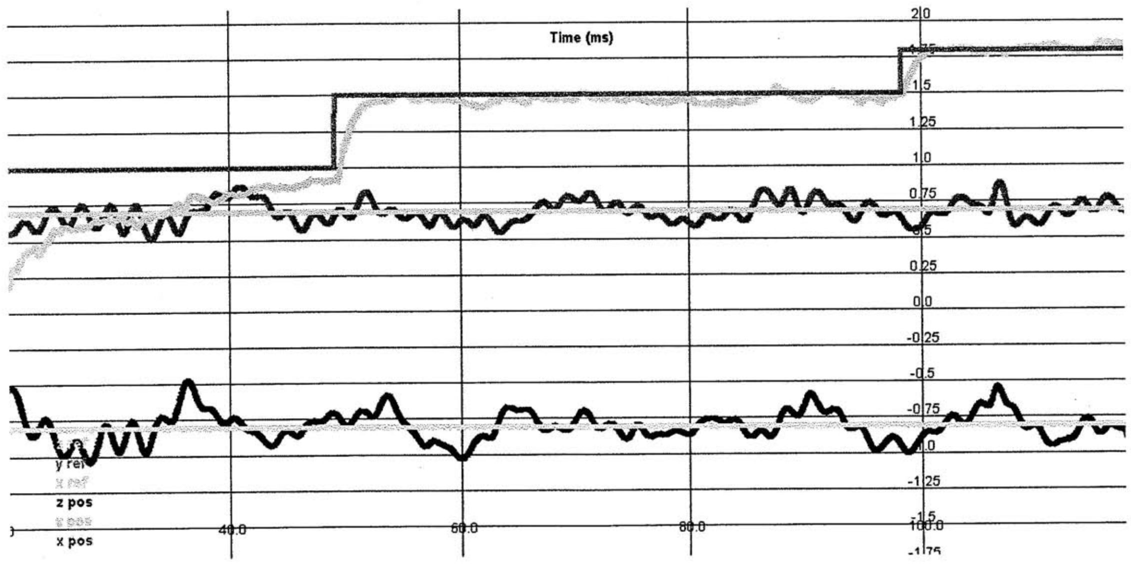


Figure 5.10: Performance of PID position control with unknown attitude stability augmentation, on the AscTech Hummingbird tracked by OptiTrack. Reference and measured positions are shown, in metres. Single point tracking is used, with yaw estimation and control based on on-board magnetometer output. Performance is relatively poor in comparison to that of sliding mode control.

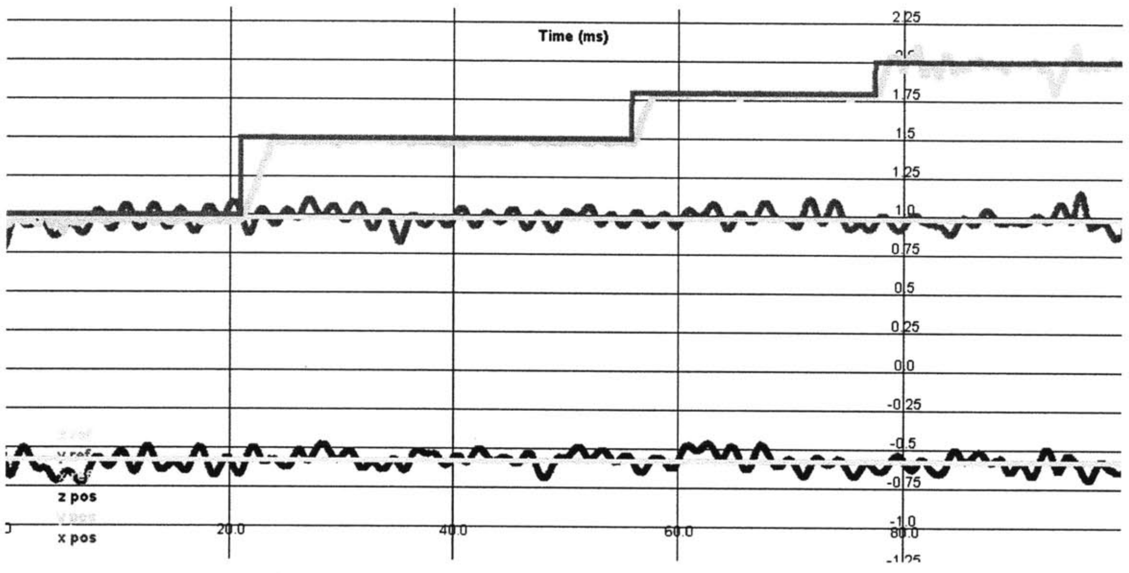


Figure 5.11: Performance of sliding mode position control with unknown attitude stability augmentation, on the AscTech Hummingbird tracked by OptiTrack. Reference and measured positions are shown, in metres. Single point tracking is used, with yaw estimation and control based on on-board magnetometer output. A significant improvement in performance compared to PID is observed.

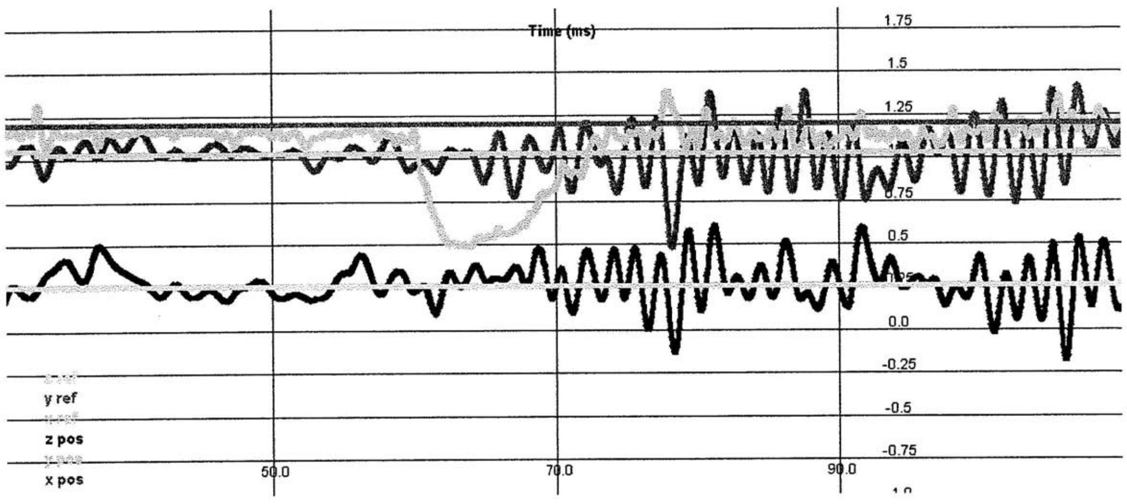


Figure 5.12: Performance of sliding mode position control with unknown attitude stability augmentation, on the AscTech Hummingbird tracked by OptiTrack, in the presence of aerodynamic interference from an identical helicopter placed at a height of 2.2m directly above the reference point. Wake interference introduced at t=60s. Reference and measured positions are shown, in metres. Single point tracking is used, with yaw estimation and control based on on-board magnetometer output.

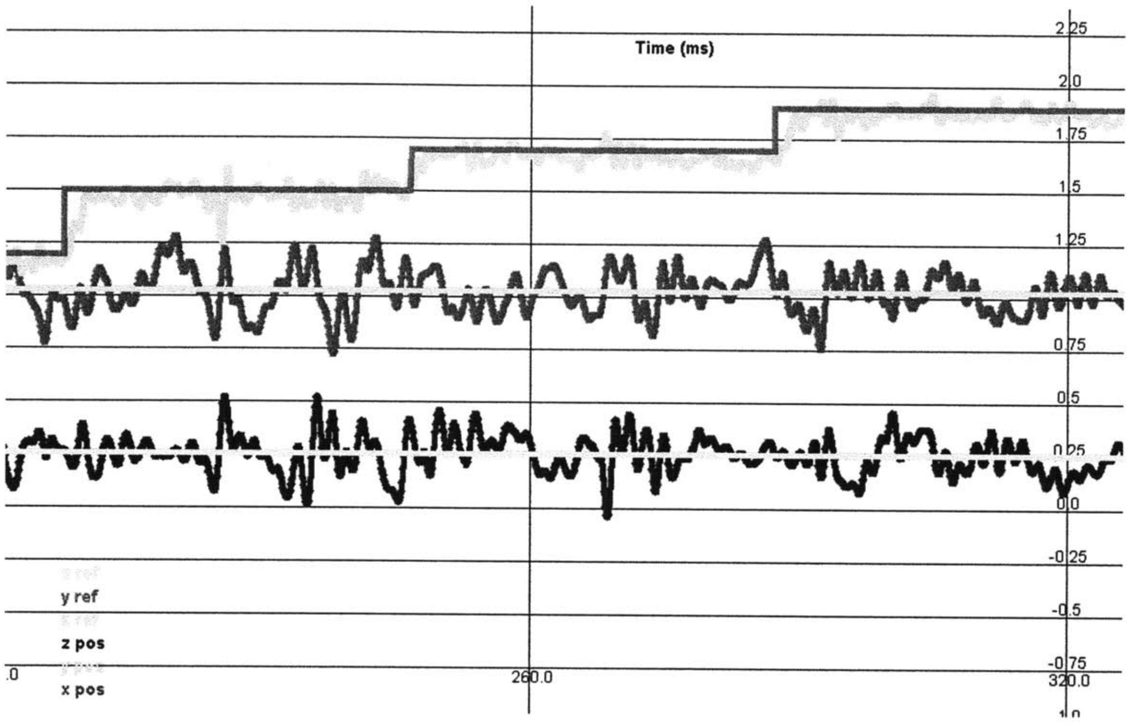


Figure 5.13: Performance of sliding mode position control with feed-forward wake compensation and unknown attitude stability augmentation, on the AscTech Hummingbird tracked by OptiTrack, in the presence of aerodynamic interference from an identical helicopter placed at a height of 2.2m directly above the reference point. Reference and measured positions are shown, in metres. Single point tracking is used, with yaw estimation and control based on on-board magnetometer output. Error bounds are similar, but the energy of the error signal is significantly reduced compared to that for the basic sliding mode controller without feed-forward compensation, as shown in 5.12.

Chapter 6

Conclusions and Future Work

6.1 Summary

This research has lead to a number of contributions, including the development of a very small, general purpose electronics stack for automatic flight, and the testing of feed-forward wake compensation for position stabilization of a quad-rotor helicopter in the wake of another, identical helicopter.

Versions 3.0, 3.1 and 3.2 of the electronics designed, offer a full suite of sensors (3-axis magnetometer, acclerometer, gyros) for stability augmentation, in system micro-controller programmability, six channel PWM control driving up to 2A per channel at 4V and robust wireless communication, in a package that is, to the author's knowledge, smaller and lighter than any commercially available equivalent. This electronics stack may be used for control of any vehicle operating on a single cell (3.7V minimum) lithium polymer battery, and very slight modifications allow servo motors or brushless motor electronic speed controllers to be commanded as well.

The type of feed-forward wake compensation described in this thesis has been shown to offer a noticeable improvement in the performance of a quad-rotor in hover below another, and while this improvement at this point is not substantial, implementation on a vehicle with direct motor control and appropriate tuning of gains, neither of which have been done, would most likely offer a significant improvement in performance. Simulation results and experimental testing suggest that the de-stabilizing nature of the wake can largely be explained by the presence of a constant downwash velocity profile, rather than as a result of stochastic temporal variations in flow, which in turn suggests that compensating in open-loop for this fixed flow profile could improve stability substantially.

6.2 Proposal for Future Work and Development

The continued development of this system would most likely prove to be a profitable financial investment, and if a reliable system of sufficient swarm size could be produced with the capability of operating in a variety of venues, a business based on the operation of such a system would, in the author's opinion, be likely to be profitable. However, there are still a number of development, implementation and operational issues that must be addressed. While the most recent quad-rotor version is almost surely capable of being flown, it has not been tested, and the overall swarm control and communication architecture is also untested. Because a reduction in vehicle mass and cost would generally be beneficial to the system, it would also likely be worthwhile to develop a quad-rotor prototype based on a small RF transceiver IC that allows lower-level control of the communication protocol as well. Currently, the monocopter prototype utilizes such an RF transceiver for communication, but is plagued with power interference issues that prevent its use for feedback control with high thrust levels. Additionally, the communication range of the transceiver/antenna pair used in this case is insufficient and additional work would be required to increase this range. From a business perspective, it would also be necessary to form a relationship with a company capable of mass-producing the unit vehicle reliably and inexpensively, and additional documents would have to be created which allow the PCBs to be populated automatically.

The logistics associated with operation of such a system, with even moderate swarm size, would also present a challenge to commercial operation, and most likely a team of technicians would have to be employed to move, set-up, and operate the system. While technology which enables autonomous docking with battery charging stations has beeen tested [7], it is not commercially available, and it would most likely be beneficial to invest as well in the development of such an automated charging system as this would relieve much of the load on human operators, as well as potentially allow a swarm with enough 'backup' helicopters to operate almost indefinitely, producing performances much longer than the flight duration of each individual vehicle.

Development of the feed-forward compensator for aerodynamic interference rejection that has been done in the course of this project is promising, as some improvement in performance has been observed using the simple wake model, with untuned gain coefficients and unknown internal stability augmentation dynamics. With direct motor speed control, and gain tuning, it seems likely that a dramatic improvement in performance may be seen using this model, and in the author's opinion it would be worthwhile to pursue testing in this direction in the future.

Appendix A

Custom Quad-rotor Flight Code

```
/*
Flight code for the Walkera-based custom quadrotor. In this implementation, the sensors are read
inefficiently (waiting during ADC conversion etc.), at fixed frequency, and used to update
low-pass estimates. Must be built with settings for Lilypad Arduino w/ ATMEGA168.
*/

#include <Wire.h>
#include <EEPROM.h>

#define TWI_ADDRESS_ACCEL 83
#define TWI_ADDRESS_MAG 30

#define MODE_MANUAL 0
#define MODE_AUGMENT 1
#define LPF_ENABLE 2
#define LPF_DISABLE 3

#define KD_PR 0
#define K_PA 1
#define KD_Y 2
#define K_PM 3
#define K_PMY 4
#define K_RATEDOT 5
#define K_RATEDOTY 6
#define K_LPF_GYROS 7
#define K_LPF_ACCEL 8
#define K_LPF_MAG 9
#define K_LPF_RATEDOT 10

#define K_LPF_GYROS_CAL 0
#define K_LPF_ACCEL_CAL 1
#define K_LPF_MAG_CAL 2
```

```
byte b[] = {0,0,0,0,0,0,0,0};
byte b_out[] = {0,0,0,0};
byte data[] = {0,0,0,0,0,0};
byte R,G,B;
float new_gain = 0;
byte motors[] = {0,0,0,0};
int v_user[] = {0,0,0,0}; //user input commands w/ stability augmentation
int v[] = {0,0,0,0}; //pitch,roll,yaw,thrust commands w/ stability augmentation
int v_temp[] = {0,0,0,0};
float v_m[] = {0,0,0};
//float m_len = 0;
//float mag0_len = 0;
int rate[] = {0,0,0};
float ratedot[] = {0,0,0};
int rate_temp[] = {0,0,0};
int accel[] = {0,0,0};
int mag[] = {0,0,0};
long mag_temp[] = {0,0,0};
int rate_raw[] = {0,0,0};
int accel_raw[] = {0,0,0};
int mag_raw[] = {0,0,0};
long mag0[] = {0,0,0};
int mag08[] = {0,0,0};


//periodic process parameters
//NOTE THAT THE PWM INITIALIZATION ROUTINE CHANGES PRESCALERS USED BY THE
//ARDUINO TIMING FUNCTION millis(), SO THAT THE RESULT
//OF CALLING millis() IS ACTUALLY IN UNITS OF 4ms RATHER THAN 1ms
unsigned long t_gyro = 0;
unsigned long T_gyro = 3;
unsigned long t_accel = 0;
unsigned long T_accel = 10;
unsigned long t_mag = 0;
unsigned long T_mag = 40;
unsigned long t_command = 0;
unsigned long T_command = 50;

unsigned long t_start_zeroing = 0;
unsigned long zeroing_duration = 5000; //milliseconds

unsigned long t;
unsigned long t2;
unsigned long t3;
unsigned long t4;
unsigned long t5;

int i = 0;
```

```
//status bits

boolean sa_enabled = false; //stability augmentation
boolean lpf_enabled = false; //low pass filtering
byte mode = MODE_MANUAL;

//calibration procedure parameters
int N_gyros = 600;
int N_accel = 100;
int N_mag = 50;
int n_gyros = 0;
int n_accel = 0;
int n_mag = 0;
int rate_offset[] = {0,0,0};
int accel_offset[] = {0,0,0};
int mag_offset[] = {0,0,0};
boolean zeroing = false;

//stability augmentation coefficients
float K[11] = {0}; //coefficients in use
//coefficients stored for use, initialized here to default values
float K0[11] = {0,0,0,0,0.001,0,0,0.3,0.5,0.5,0.5};
float K_lpf_zero[3] = {0.95,0.9,0.85}; //LPF coefficients used during sensor zeroing

float k_scale_ratedot = 20;

void setup(){
  Serial.begin(9600);
  Wire.begin();
  delay(20);
  initPWM();
  initAccel();
  initMagneto();
  readMagOffset(); //reads previous magnetometer calibration data from EEPROM
}

void loop(){
  t = millis()*4; //multiply time by 4 to account for changed prescalers in initPWM()

  //Serial receive code. All commands begin with 19, followed by a command identifier,
  //a number of data bytes, and possibly
  //termination / checking byte(s)
  if (Serial.available() > 0){
    b[0] = Serial.read();
    if (b[0] == 19){ //command packet

      while(Serial.available() < 1){}
      b[0] = Serial.read();
```

```
if (b[0] == 90){ //Kill motors, disable automatic control
  motors[0] = 0;
  motors[1] = 0;
  motors[2] = 0;
  motors[3] = 0;
  v_user[0] = 0;
  v_user[1] = 0;
  v_user[2] = 0;
  v_user[3] = 0;
  writeMotors();
  disableControlGains();
  Serial.write(90);
}
else if (b[0] == 94){
//Set external motor commands, added to stability augmentation computed commands
  while(Serial.available() < 5){}
  b[0] = Serial.read();
  b[1] = Serial.read();
  b[2] = Serial.read();
  b[3] = Serial.read();
  b[4] = Serial.read();
  if (b[4] == 84){
    v_user[0] = b[0];
    v_user[1] = b[1];
    v_user[2] = b[2];
    v_user[3] = b[3];
  }
}

else if (b[0] == 97){ //Set control mode or enable LPFs
  while(Serial.available() < 2){}
  b[0] = Serial.read();
  b[1] = Serial.read();
  if (b[1] == 87){
    switch(b[0]){
      case MODE_MANUAL:
        disableControlGains();
        break;
      case MODE_AUGMENT:
        enableControlGains();
        break;
      case LPF_ENABLE:
        enableLPFs();
        break;
      case LPF_DISABLE:
        disableLPFs();
        break;
      }
    Serial.write(b[0]+97);
  }
}
else if (b[0] == 99){ //Get rate estimate, deresolved into single byte
  Serial.write((rate[0]-rate_offset[0])/4);
  Serial.write((rate[1]-rate_offset[1])/4);
  Serial.write((rate[2]-rate_offset[2])/4);
  Serial.write(99);
}
else if (b[0] == 100){ //Get filtered accelerometer output
  Serial.write((accel[0]-accel_offset[0])/4);
  Serial.write((accel[1]-accel_offset[1])/4);
  Serial.write((accel[2]-accel_offset[2])/4);
  Serial.write(100);
}
else if (b[0] == 101){ //Get filtered magnetometer output
  Serial.write((mag[0]-mag_offset[0])/8);
  Serial.write((mag[1]-mag_offset[1])/8);
  Serial.write((mag[2]-mag_offset[2])/8);
  Serial.write(101);
}
else if (b[0] == 102){ //Get motor commands
  Serial.write(motors[0]);
  Serial.write(motors[1]);
  Serial.write(motors[2]);
  Serial.write(motors[3]);
  Serial.write(102);
}
else if (b[0] == 103){ //Get angular acceleration estimate
  Serial.write((byte)(k_scale_ratedot*ratedot[0]));
  Serial.write((byte)(k_scale_ratedot*ratedot[1]));
  Serial.write((byte)(k_scale_ratedot*ratedot[2]));
  Serial.write(103);
}
else if (b[0] == 110){ //Run magnetometer calibration routine
  Serial.write(110);
  delay(250); //actually a 1s delay
  calibrateMagneto();
  Serial.write(111);
}
else if (b[0] == 111){ //Read magnetometer offset data
  Serial.write(mag_offset[0]/8);
  Serial.write(mag_offset[1]/8);
  Serial.write(mag_offset[2]/8);
  Serial.write(111);
}
else if (b[0] == 112){ //Read gyro offset data
  Serial.write(rate_offset[0]/4);
  Serial.write(rate_offset[1]/4);
  Serial.write(rate_offset[2]/4);
```

```
      Serial.write(112);
    }
    else if (b[0] == 113){ //Read accelerometer offset data
      Serial.write(accel_offset[0]/4);
      Serial.write(accel_offset[1]/4);
      Serial.write(accel_offset[2]/4);
      Serial.write(113);
    }
    else if (b[0] == 114){ //Read magnetometer zero data
      Serial.write(mag0[0]/8);
      Serial.write(mag0[1]/8);
      Serial.write(mag0[2]/8);
      Serial.write(114);
    }
    else if (b[0] == 120){ //Begin sensor zeroing
      Serial.write(120);
      enableZeroingLPFs();
      zeroing = true;
      t_start_zeroing = t;
    }
  }

}

//Sensor read / estimate updates: periodic functions

if (t-t_gyro > T_gyro){
  readGyros();
  float k = K[K_LPF_GYROS];
  rate_temp[0] = (int)(k*rate[0] + (1-k)*rate_raw[0]);
  rate_temp[1] = (int)(k*rate[1] + (1-k)*rate_raw[1]);
  rate_temp[2] = (int)(k*rate[2] + (1-k)*rate_raw[2]);

  k = K[K_LPF_RATEDOT];
  ratedot[0] = k*ratedot[0] + (1-k)*(rate_temp[0]-rate[0])/(t-t_gyro);
  ratedot[1] = k*ratedot[1] + (1-k)*(rate_temp[1]-rate[1])/(t-t_gyro);
  ratedot[2] = k*ratedot[2] + (1-k)*(rate_temp[2]-rate[2])/(t-t_gyro);

  rate[0] = rate_temp[0];
  rate[1] = rate_temp[1];
  rate[2] = rate_temp[2];

  t_gyro = t;
}

if (t-t_accel > T_accel){
  readAccel();
  float k = K[K_LPF_ACCEL];
  accel[0] = (int)(k*accel[0] + (1-k)*accel_raw[0]);
  accel[1] = (int)(k*accel[1] + (1-k)*accel_raw[1]);
  accel[2] = (int)(k*accel[2] + (1-k)*accel_raw[2]);
  t_accel = t;
}

if (t-t_mag > T_mag){
  readMagneto();
  float k = K[K_LPF_MAG];
  mag[0] = (int)(k*mag[0] + (1-k)*mag_raw[0]);
  mag[1] = (int)(k*mag[1] + (1-k)*mag_raw[1]);
  mag[2] = (int)(k*mag[2] + (1-k)*mag_raw[2]);
  t_mag = t;
}

//Control command update. Writes commands to motors.

if (t-t_command > T_command){
  //compute control commands and update motors if pryt control enabled
  //note that gyro readings range in 0-1023, accel in -512-511, magneto in -2048-2047

  //compute non-normalized magnetic field cross product
  mag_temp[0] = (mag[0]-mag_offset[0]);
  mag_temp[1] = (mag[1]-mag_offset[1]);
  mag_temp[2] = (mag[2]-mag_offset[2]);

  v_m[0] = (float)(mag0[1]*mag_temp[2] - mag_temp[1]*mag0[2]);
  v_m[1] = (float)(mag0[2]*mag_temp[0] - mag_temp[2]*mag0[0]);
  v_m[2] = (float)(mag0[0]*mag_temp[1] - mag_temp[0]*mag0[1]);


  //compute stability augmentation commands in PCB frame, body fixed coordinate system
  //(pitch,roll,yaw,thrust)

  v[0] = -K[KD_PR]*(rate[2]-rate_offset[2]) + K[K_PA]*(accel[0]-accel_offset[0]) - K[K_PM]*v_m[2];
  v[1] = -K[KD_PR]*(rate[0]-rate_offset[0]) - K[K_PA]*(accel[2]-accel_offset[2]) + K[K_PM]*v_m[0];
  v[2] = -K[KD_Y]*(rate[1]-rate_offset[1]) + K[K_PMY]*v_m[1];// - K[K_RATEDOTY]*ratedot[1];
  v[3] = 0;

  //convert to motor commands, add user commands, should be scaled to the range 0-255

  v_temp[0] = v[0] - v[1] + v[2] + v[3] + v_user[0];
  v_temp[1] = -v[0] - v[1] - v[2] + v[3] + v_user[1];
  v_temp[2] = v[0] + v[1] - v[2] + v[3] + v_user[2];
  v_temp[3] = -v[0] + v[1] + v[2] + v[3] + v_user[3];

  //force outputs in range
  if (v_temp[0] > 255) v_temp[0] = 255;
  else if (v_temp[0] < 0) v_temp[0] = 0;
  if (v_temp[1] > 255) v_temp[1] = 255;
```

```
    else if (v_temp[1] < 0) v_temp[1] = 0;
    if (v_temp[2] > 255) v_temp[2] = 255;
    else if (v_temp[2] < 0) v_temp[2] = 0;
    if (v_temp[3] > 255) v_temp[3] = 255;
    else if (v_temp[3] < 0) v_temp[3] = 0;

    motors[0] = v_temp[0];
    motors[1] = v_temp[1];
    motors[2] = v_temp[2];
    motors[3] = v_temp[3];

    writeMotors();

    t_command = t;
  }


  //Used during sensor zeroing operation to store rest-state sensor outputs.
  //LPF gains are set to 0 upon completion.
  if  (zeroing && (t-t_start_zeroing > zeroing_duration)){ //zeroing finished
    rate_offset[0] = rate[0];
    rate_offset[1] = rate[1];
    rate_offset[2] = rate[2];
    accel_offset[0] = accel[0];
    accel_offset[1] = accel[1];
    accel_offset[2] = accel[2];
    mag0[0] = mag[0] - mag_offset[0];
    mag0[1] = mag[1] - mag_offset[1];
    mag0[2] = mag[2] - mag_offset[2];
    disableLPFs();
    Serial.write(121);
    zeroing = false;
  }

  //delay(5);

}

void readGyros(){
  //reads gyros to raw data vectors
  rate_raw[0] = analogRead(0);
  rate_raw[1] = -analogRead(2);
  rate_raw[2] = analogRead(1);
}

void readAccel(){
  //reads accelerometer to raw data vectors

  Wire.beginTransmission(TWI_ADDRESS_ACCEL);
  Wire.send(0x32);
  Wire.endTransmission();
  Wire.requestFrom(TWI_ADDRESS_ACCEL,6);
  i = 0;
  while(Wire.available()){
    data[i] = Wire.receive();
    i++;
  }

  accel_raw[0] = -((data[1] << 8) | data[0]);
  accel_raw[2] = -((data[3] << 8) | data[2]);
  accel_raw[1] = -((data[5] << 8) | data[4]);
}

void readMagneto(){
   //reads magnetometer to raw data vectors

  Wire.beginTransmission(TWI_ADDRESS_MAG);
  Wire.send(3);
  Wire.endTransmission();
  Wire.requestFrom(TWI_ADDRESS_MAG,6);
  i = 0;
  while(Wire.available()){
    data[i] = Wire.receive();
    i++;
  }

  mag_raw[2] = ((data[0] << 8) | data[1]);
  mag_raw[0] = -((data[2] << 8) | data[3]);
  mag_raw[1] = -((data[4] << 8) | data[5]);

}

void writeMotors(){
  OCR0A = motors[3];
  OCR1A = motors[0];
  OCR0B = motors[1];
  OCR1B = motors[2];
}

void calibrateMagneto(){
  //Offset calibration for magnetometer, performed while the vehilce
  //is rotated around into many possible orientations
  //Finds maximum and minimum values of magnetic field for each axis,
  //and computes offsets as the middle of this range

  int Bmax[] = {0,0,0};
  int Bmin[] = {0,0,0};
  t3 = millis()*4;
  while(true){

    t2 = millis()*4;
    if (t2-t_mag > T_mag){
      readMagneto();
      for(i=0; i<3; i++){
        if (mag_raw[i] != -4096){
          if (mag_raw[i] > Bmax[i]) Bmax[i] = mag_raw[i];
          else if (mag_raw[i] < Bmin[i]) Bmin[i] = mag_raw[i];
        }
      }
      t_mag = t2;
    }
    //10 seconds elapsed and at least a range of 400 covered in each dimension
    if ((t2-t3 > 15000) && (Bmax[0]-Bmin[0] > 400)
     && (Bmax[1]-Bmin[1] > 400) && (Bmax[2]-Bmin[2] > 400)) break;
  }
  mag_offset[0] = (Bmax[0]+Bmin[0])/2;
  mag_offset[1] = (Bmax[1]+Bmin[1])/2;
  mag_offset[2] = (Bmax[2]+Bmin[2])/2;

  //writes the result of the calibration to EEPROM
  EEPROM.write(8, ((mag_offset[0] & 0xFF00) >> 8));
  EEPROM.write(9, ((mag_offset[0] & 0x00FF)));
  EEPROM.write(10, ((mag_offset[1] & 0xFF00) >> 8));
  EEPROM.write(11, ((mag_offset[1] & 0x00FF)));
  EEPROM.write(12, ((mag_offset[2] & 0xFF00) >> 8));
  EEPROM.write(13, ((mag_offset[2] & 0x00FF)));
}

void readMagOffset(){
  //reads magnetometer offset data from EEPROM
  for(i=8; i<14; i++){
    data[i-8] = EEPROM.read(i);
  }
  mag_offset[0] = ((data[0] << 8) | data[1]);
  mag_offset[1] = ((data[2] << 8) | data[3]);
  mag_offset[2] = ((data[4] << 8) | data[5]);
}


void enableControlGains(){
  //enables stability augmentation by setting applied control vector K to gain vector K0
  K[KD_PR] = K0[KD_PR];
  K[K_PA] = K0[K_PA];
  K[KD_Y] = K0[KD_Y];
  K[K_PM] = K0[K_PM];
  K[K_PMY] = K0[K_PMY];
  K[K_RATEDOT] = KO[K_RATEDOT];
  sa_enabled = true;
}

void disableControlGains(){
  //disables stability augmentation by setting all control gains to 0
  K[KD_PR] = 0;
  K[K_PA] = 0;
  K[KD_Y] = 0;
  K[K_PM] = 0;
  K[K_PMY] = 0;
  K[K_RATEDOT] = 0;
  sa_enabled = false;
}

void enableLPFs(){
  //enables low pass filtering with gains in KO
  K[K_LPF_GYROS] = KO[K_LPF_GYROS];
  K[K_LPF_ACCEL] = KO[K_LPF_ACCEL];
  K[K_LPF_MAG] = KO[K_LPF_MAG];
  K[K_LPF_RATEDOT] = KO[K_LPF_RATEDOT];
  lpf_enabled = true;
}

void enableZeroingLPFs(){
  //enables slow low pass filtering used to obtain rest-state 'zero' sensor readings
  K[K_LPF_GYROS] = K_lpf_zero[K_LPF_GYROS_CAL];
  K[K_LPF_ACCEL] = K_lpf_zero[K_LPF_ACCEL_CAL];
  K[K_LPF_MAG] = K_lpf_zero[K_LPF_MAG_CAL];
  lpf_enabled = false;
}

void disableLPFs(){
  //disables low pass filtering
  K[K_LPF_GYROS] = 0;
  K[K_LPF_ACCEL] = 0;
  K[K_LPF_MAG] = 0;
  K[K_LPF_RATEDOT] = 0;
  lpf_enabled = false;
}

void initAccel(){
  Wire.beginTransmission(TWI_ADDRESS_ACCEL);
  Wire.send(0x2D);
  Wire.send(_BV(3));
  Wire.endTransmission();
  Wire.beginTransmission(TWI_ADDRESS_ACCEL);
  Wire.send(0x38);
  Wire.send(_BV(7));
  Wire.endTransmission();
}

void initMagneto(){
  Wire.beginTransmission(TWI_ADDRESS_MAG);
  Wire.send(0x02);
  Wire.send(0);
  Wire.endTransmission();

  delay(50);

  Wire.beginTransmission(TWI_ADDRESS_MAG);
  Wire.send(0);
  Wire.send((_BV(3) | _BV(4)));
  Wire.endTransmission();
}

void initPWM(){
  //configure timer/couters for fast PWM operation, prescaling system clock by 1/256.

  // pin 10/OCOA, OCROA determines duty cycle
  TCCR0A = _BV(COM0A1) | _BV(COM0B1) | _BV(WGM01) | _BV(WGM00);
  TCCR0B = _BV(CS02);

  //pins 11 and 12, OCR1A and OCR1B controlled, 8-bit PWM (supports up to 10 bit)
  TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM10);
  TCCR1B = _BV(WGM12) | _BV(CS12);
  //pin 13, OCR2A
  TCCR2A = _BV(COM2A1) | _BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS22) | _BV(CS21);
  //configure pins as outputs
  DDRB |= _BV(DDB1) | _BV(DDB2) | _BV(DDB3);
  DDRD |= _BV(DDD5) | _BV(DDD6);

  OCR0B = 0;
  OCR0A = 0;
  OCR1A = 0;
  OCR1B = 0;
  OCR2A = 0;

}
```

Appendix B

Monocopter Flight Code

```
#include <math.h>

byte channel_num = 1;
byte network_num = 1;
byte device_num = 1;
byte device_type = 1;
byte sync_tag = 164;

byte b_rec[] = {0,0,0,0,0,0,0,0};

int mag_x_pin = 0;
int mag_y_pin = 1;

int mag[] = {0,0};
float k_mag = 0.1;
int mag0[] = {0,0};
byte commands[] = {0,0,0}; //generalized direction commands

byte mag_max[] = {0,0};
byte mag_min[] = {255,255};
byte mag_avg[] = {0,0};

unsigned long t_lastcomm = 0;
unsigned long T_commtimeout = 2000;
unsigned long t = 0;


void setup(){

  initPWM();
  Serial.begin(19200);
  delay(200);
  initRadio();
  delay(500);
  commandLEDs(0,0,0);

  //zeroHeading();
}

void loop(){
  t = millis()*4; //multiply by 4 to account for change of prescaler in initPWM()

  if(Serial.available() > 12){
    b_rec[0] = Serial.read();
    if (b_rec[0] == sync_tag){

    b_rec[0] = Serial.read();
    b_rec[1] = Serial.read();
    b_rec[2] = Serial.read();
      if (b_rec[0] == 9 && b_rec[1] == 0x4E && b_rec[2] == channel_num){
        b_rec[0] = Serial.read();
        b_rec[1] = Serial.read();
        b_rec[2] = Serial.read();
        b_rec[3] = Serial.read();
        b_rec[4] = Serial.read();
        b_rec[5] = Serial.read();
        b_rec[6] = Serial.read();
        b_rec[7] = Serial.read();
        b_rec[8] = Serial.read();

        byte chk = 164^9^0x4E^channel_num^b_rec[0]^b_rec[1]^b_rec[2]
         ^b_rec[3]^b_rec[4]^b_rec[5]^b_rec[6]^b_rec[7];

        if (chk == b_rec[8] && b_rec[0] == 69 && b_rec[7] == 96){
          commands[0] = b_rec[1];
          commands[1] = b_rec[2];
          commands[2] = b_rec[3];
          commandLEDs(b_rec[4],b_rec[5],b_rec[6]);
          t_lastcomm = t;
        }

      }
    }
  }

  if (t-t_lastcomm > T_commtimeout){
    commands[0] = 0;
    commands[1] = 0;
    commands[2] = 0;
    commandLEDs(0,0,0);
    updateMotors();
    delay(100);
    initRadio();
  }

  magReading();
  updateMotors();

}

void initRadio(){
  int r = setRFFreq();
  delay(20);
  int r1 = assignChannel();
  delay(20);
  int r2 = setChannelID();
  delay(20);
  int r3 = setChannelPeriod();
  delay(20);
  int r4 = openChannel();

   if (r == 0 && r1 == 0 && r2 == 0 && r3 == 0)
    commandLEDs(0,90,0);
    else commandLEDs(90,0,0);
}

void magReading(){
   mag[0] = k_mag*mag[0] + (1-k_mag)*analogRead(0);
   //delay(5);
   mag[1] = k_mag*mag[1] + (1-k_mag)*analogRead(1);
}

//updates the actual commands to the motors
void updateMotors(){
  int c = commands[2];
  if (commands[2] > 127) c = -128+commands[2];
  float p = c*PI/180.0;
  int m[] = {mag[0]-mag_avg[0] , mag[1]-mag_avg[1]};
  int m0[] = {mag0[0]-mag_avg[0] , mag0[1]-mag_avg[1]};

  int hdif = commands[1]*((cos(p)*m[0] - sin(p)*m[1])*m0[0] + (sin(p)*m[0] + cos(p)*m[1])*m0[1]);

  commandMotors(commands[0]+hdif,commands[0]-hdif);
}

void commandLEDs(int r, int g, int b){
  OCR1A = r;
  OCR1B = g;
  OCR2A = b;
}
```

```
void commandMotors(int m0, int m1){
  OCR0A = m0;
  OCR0B = m1;
}

void magnetoCalibrate(){
  commandLEDs(0,0,120);
  delay(500);
  unsigned long mag_sum[] = {0,0};
  int n = 500;
  for(int i=0;i<n; i++){
    mag[0] = analogRead(0);
    mag[1] = analogRead(1);
    if (mag[0] > mag_max[0]) mag_max[0] = mag[0];
    if (mag[1] > mag_max[1]) mag_max[1] = mag[1];
    if (mag[0] < mag_min[0]) mag_min[0] = mag[0];
    if (mag[1] < mag_min[1]) mag_min[1] = mag[1];
    mag_sum[0] += mag[0];
    mag_sum[1] += mag[1];
    delay(5);
  }
  mag_avg[0] = mag_sum[0]/n;
  mag_avg[1] = mag_sum[1]/n;
  commandLEDs(0,90,0);
}

void zeroHeading(){
  unsigned long t0 = millis();
  unsigned long t = t0;
  while(true){
    t = millis();
    if (t-t0 > 1000) break;
    mag0[0] = 0.8*mag0[0] + 0.2*analogRead(0);
    mag0[1] = 0.8*mag0[1] + 0.2*analogRead(1);
  }
}


int setRFFreq(){
  Serial.write(sync_tag);
  Serial.write(2);
  Serial.write(0x45);
  Serial.write(channel_num);
  Serial.write(66);
  byte chk = sync_tag^2^0x45^channel_num^66;
  Serial.write(chk);

  return confirmMessage(0x45);
}
int assignChannel(){
  Serial.write(sync_tag);
  Serial.write(3);
  Serial.write(0x42);
  Serial.write(channel_num);
  Serial.write((byte)0); //bi-directional receive channel
  Serial.write(network_num);
  byte chk = sync_tag^3^0x42^channel_num^0^network_num;
  Serial.write(chk);

  return confirmMessage(0x42);
}
int setChannelID(){
  Serial.write(sync_tag);
  Serial.write(5);
  Serial.write(0x51);
  Serial.write(channel_num);
  Serial.write((byte)0);
  Serial.write(device_num);
  Serial.write(device_type);
  Serial.write(0x4E);
  byte chk = sync_tag^5^0x51^channel_num^0^device_num^device_type^0x4E;
  Serial.write(chk);

  return confirmMessage(0x51);
}
int setChannelPeriod(){
  Serial.write(sync_tag);
  Serial.write(3);
  Serial.write(0x43);
  Serial.write(channel_num);
  Serial.write((byte)0);
  Serial.write(2);
  byte chk = sync_tag^3^0x43^channel_num^2^0;
  Serial.write(chk);

  return confirmMessage(0x43);
}
int openChannel(){
  Serial.write(sync_tag);
  Serial.write(1);
  Serial.write(0x4B);
  Serial.write(channel_num);
  byte chk = sync_tag^1^0x4B^channel_num;
  Serial.write(chk);

  return confirmMessage(0x4B);
}
```

```
int confirmMessage(byte msg_id){
 while(Serial.available() < 7){}
  b_rec[0] = Serial.read();
  b_rec[1] = Serial.read();
  b_rec[2] = Serial.read();
  b_rec[3] = Serial.read();
  b_rec[4] = Serial.read();
  b_rec[5] = Serial.read();
  b_rec[6] = Serial.read();

  if (b_rec[0] == 164 && b_rec[1] == 3 && b_rec[2] == 64
   && b_rec[3] == channel_num && b_rec[4] == msg_id) return 0;
  else return 1;
}

void initPWM(){
  //configure timer/couters for fast PWM operation, prescaling system clock by 1/256.

  // pin 10/OC0A, OCR0A determines duty cycle
  TCCR0A = _BV(COM0A1) | _BV(COM0B1) | _BV(WGM01) | _BV(WGM00);
  TCCR0B = _BV(CS02);

  //pins 11 and 12, OCR1A and OCR1B controlled, 8-bit PWM (supports up to 10 bit)
  TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM10);
  TCCR1B = _BV(WGM12) | _BV(CS12);

  //pin 13, OCR2A
  TCCR2A = _BV(COM2A1) | _BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS22) | _BV(CS21);

  //configure pins as outputs
  DDRB |= _BV(DDB1) | _BV(DDB2) | _BV(DDB3);
  DDRD |= _BV(DDD5) | _BV(DDD6);

  OCR0B = 0;
  OCR0A = 0;
  OCR1A = 0;
  OCR1B = 0;
  OCR2A = 0;

}

void initPWMfast(){
  //configure timer/couters for fast PWM operation, prescaling system clock by 1/64.

  // pin 10/OC0A, OCR0A determines duty cycle
  TCCR0A = _BV(COM0A1) | _BV(COM0B1) | _BV(WGM01) | _BV(WGM00);
  TCCR0B = _BV(CS01) | _BV(CS00);

  //pins 11 and 12, OCR1A and OCR1B controlled, 8-bit PWM (supports up to 10 bit)
  TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM10);
  TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10);

  //pin 13, OCR2A
  TCCR2A = _BV(COM2A1) | _BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS22);

  //configure pins as outputs
  DDRB |= _BV(DDB1) | _BV(DDB2) | _BV(DDB3);
  DDRD |= _BV(DDD5) | _BV(DDD6);

  OCR0B = 0;
  OCR0A = 0;
  OCR1A = 0;
  OCR1B = 0;
  OCR2A = 0;

}
```

Appendix C

Hummingbird Off-board Flight Code

This appendix contains the core of the code used to operate the Hummingbird through the serial interface, with feedback from position (and possibly attitude) information provided by the Optitrack motion capture system. Many auxiliary classes are not included here for conciseness.

```
package Autopilot;

import java.awt.Color;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.ArrayList;

import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

import com.centralnexus.input.Joystick;
import com.centralnexus.input.JoystickListener;

/**
 * @author Erich Mueller
*/

/**
 *Frame which provides the graphical display of data read from the
 *IMU and Optitrack systems, and collects input from the joystick.
 *Contains the main method that should be called to start the program.
*/
public class MasterFrame extends JFrame{

private CommThread comm;
public IMUThread imu;
private Updater updater;
private int pitch, roll, yaw, thr;
private double t0;
private boolean first_data = true;
private Joystick joy;

private ArrayList<Graph> graphs;
private ArrayList<Controller> conts;
private ArrayList<VehicleGUI> displays;
private RelayThread relaythread;

private JPanel mainpanel;
private JMenuBar menubar;
private JMenu vehiclemenu;
private JMenu trajmenu;
private JMenu relaymenu;
private JMenu datamenu;
private JMenuItem add;
private JMenuItem add_ptcloud;
private JMenuItem startall;
private JMenuItem stopall;
private JMenuItem gotostartall;
private JMenuItem relayload;
private JMenuItem datastart;

private MasterFrame me = this;

public static void main(String[] args){
MasterFrame mf = new MasterFrame();
}

public MasterFrame(){
setDefaultCloseOperation(EXIT_ON_CLOSE);
setBounds(50,50,1000,700);
setTitle("Hummingbird Autonomous Aircraft Data Display");
joy = createJoy();
if (joy == null){
System.out.println("Joystick could not be connected");
//System.exit(0);
}
displays = new ArrayList<VehicleGUI>();
conts = new ArrayList<Controller>();
```

```
comm = new CommThread(this);
imu = new IMUThread(this);
updater = new Updater(this);

mainpanel = new JPanel();
menubar = new JMenuBar();
vehiclemenu = new JMenu("Vehicles");
trajmenu = new JMenu("Trajectories");
relaymenu = new JMenu("Wireless Relays");
datamenu = new JMenu("Data");
add = new JMenuItem("Add Vehicle");
add_ptcloud = new JMenuItem("Add Vehicle - 1pt Tracking");
startall = new JMenuItem("Start All Paths");
stopall = new JMenuItem("Stop All Paths");
gotostartall = new JMenuItem("Take All to Starts");
relayload = new JMenuItem("Load Relay Sequence");
datastart = new JMenuItem("Start Data Capture");
vehiclemenu.add(add);
vehiclemenu.add(add_ptcloud);
trajmenu.add(startall);
trajmenu.add(gotostartall);
trajmenu.add(stopall);
relaymenu.add(relayload);
datamenu.add(datastart);
menubar.add(vehiclemenu);
menubar.add(trajmenu);
menubar.add(relaymenu);
menubar.add(datamenu);
setJMenuBar(menubar);
Container cpane = getContentPane();
cpane.add(mainpanel);

if (joy != null){
joy.addJoystickListener(new JoystickListener(){
public void joystickButtonChanged(Joystick j){
updateJoystickInput(j);
}
public void joystickAxisChanged(Joystick j){
updateJoystickInput(j);
}
});
}
add.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
AddFrame f = new AddFrame(me,false);
}
});
add_ptcloud.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
```

```
AddFrame f = new AddFrame(me,true);
}
});
startall.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
for(int i=0; i<conts.size(); i++){
conts.get(i).startTrajectory();
}
if (relaythread != null) relaythread.start();
}
});
gotostartall.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
for(int i=0; i<conts.size(); i++){
conts.get(i).goToTrajStart();
}
}
});
stopall.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
for(int i=0; i<conts.size(); i++){
conts.get(i).stopTrajectory();
}
}
});
relayload.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
JFileChooser f =
new JFileChooser("C:\\Documents and Settings\\Erich\\My Documents\\eclipse\\Trajectories");
int returnVal = f.showOpenDialog(me);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
     relaythread = new RelayThread(f.getSelectedFile());
     System.out.println("Relay Program Loaded");
    }
}
});
datastart.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
LoggerFrame lf = new LoggerFrame(new DataLogger(conts),updater);
}
});

setVisible(true);


comm.start();
imu.start();
updater.start();
}
```

```
public void createGraphs(){
graphs.add(new Graph("Time", "Position"));
graphs.get(0).addSeries("X",Color.RED);
graphs.get(0).addSeries("Y",Color.BLUE);
graphs.get(0).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Velocity"));
graphs.get(1).addSeries("X",Color.RED);
graphs.get(1).addSeries("Y",Color.BLUE);
graphs.get(1).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Angular Rate"));
graphs.get(2).addSeries("X",Color.RED);
graphs.get(2).addSeries("Y",Color.BLUE);
graphs.get(2).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Commands"));
graphs.get(3).addSeries("Pitch",Color.RED);
graphs.get(3).addSeries("Roll",Color.BLUE);
graphs.get(3).addSeries("Yaw",Color.GREEN);
graphs.get(3).addSeries("Thrust",Color.CYAN);
graphs.add(new Graph("Time", "Quaternion"));
graphs.get(4).addSeries("Q1",Color.RED);
graphs.get(4).addSeries("Q2",Color.BLUE);
graphs.get(4).addSeries("Q3",Color.GREEN);
graphs.get(4).addSeries("Q4",Color.CYAN);
graphs.add(new Graph("Time", "Quat Internal"));
graphs.get(5).addSeries("Q1",Color.RED);
graphs.get(5).addSeries("Q2",Color.BLUE);
graphs.get(5).addSeries("Q3",Color.GREEN);
graphs.get(5).addSeries("Q4",Color.CYAN);
graphs.add(new Graph("Time", "Optitrack Angvel (rad/s)"));
graphs.get(6).addSeries("X",Color.RED);
graphs.get(6).addSeries("Y",Color.BLUE);
graphs.get(6).addSeries("Z",Color.GREEN);
}

/**
 * Updates the graphical state display of Rigid Body State data.
 * @param x
 * The new rigid body state data to plot.
 *//*
public void updateGUI(RB_State x){
if (first_data && x.time > 0.1) {
t0 = x.time;
first_data = false;
}
double t = x.time-t0;
graphs.get(0).pushDataPoints(t,x.pos.v);
graphs.get(1).pushDataPoints(t,x.vel.v);
graphs.get(4).pushDataPoints(t,x.quat.toList());
```

```
for(int i=0; i<graphs.size(); i++){
if (i==0 || i==1 || i==4)
graphs.get(i).autoFit();
}
repaint();
}
*//**
 * Updates the graphical state display of the vehicle rotational rate.
 * @param rate
 * The new rotational rate to plot.
 *//*
public void updateGUI(Vec3 rate, Quat q){
if (first_data_rate) {
t0_rate = Util.getTimeMils();
first_data_rate = false;
}
double t = (Util.getTimeMils()-t0_rate)/1000.0;

graphs.get(2).pushDataPoints(t,rate.v);
graphs.get(3).pushDataPoints(t,
new double[]{sent_commands[0],sent_commands[1],sent_commands[2],sent_commands[3]});
graphs.get(5).pushDataPoints(t,new double[]{q.q1,q.q2,q.q3,q.q4});

for(int i=0; i<graphs.size(); i++){
if (i==2 || i==3 || i==5)
graphs.get(i).autoFit();
}
repaint();
}*/

public void updateGUI(){
if (first_data) {
t0 = Util.getTimeMils();
first_data = false;
}
double t = (Util.getTimeMils()-t0)/1000.0;
for(int i=0; i<displays.size(); i++){
displays.get(i).updateGUI(t);
}
/*graphs.get(0).pushDataPoints(t,controller.getPosition().v);
graphs.get(1).pushDataPoints(t,controller.getVelocity().v);
graphs.get(2).pushDataPoints(t,controller.getIMURate().v);
graphs.get(3).pushDataPoints(t,controller.getSentCommands());
graphs.get(4).pushDataPoints(t,controller.getQuat().toList());
graphs.get(5).pushDataPoints(t,controller.getQuatOnboard().toList());
graphs.get(6).pushDataPoints(t,controller.getQuatRate().v);
for(int i=0; i<graphs.size(); i++){
graphs.get(i).autoFit();
}*/
}

/**
 * Handles changed input from the Joystick.
 * @param j
 * The Joystick object in use.
 */
public void updateJoystickInput(Joystick j){
pitch = (int)(-2046*j.getY());
roll = (int)(-2046*j.getX());
yaw = (int)(-2046*j.getR());
/*String s = Integer.toHexString(j.getButtons());
char c0 = s.charAt(0);
if (motor_state == false && (c0 == '1')){
motor_state = true;
imu.toggleMotors();
}
if (motor_state == true && (c0 == '0')){
motor_state = false;
imu.toggleMotors();
}*/
double zpct = (1-j.getZ())/2.0;
double upct = (1-j.getU())/2.0;

thr = (int)(upct*(4094-200)+zpct*200);
/*yref = (int)(zpct*200);
if (control_mode == MODE_STICK){
if ((c0 == '2')){
control_mode = MODE_AUTO;
controller.setIntegrating(true);
}
}
else if (control_mode == MODE_ALT_CONTROL || control_mode == MODE_AUTO){
if ((c0 == '1')){
control_mode = MODE_STICK;
controller.setIntegrating(false);
}
}*/
}

public Joystick createJoy(){
Joystick j = null;
try{
j = Joystick.createInstance();
}
catch(IOException e){
System.out.println("ERROR CREATING JOYSTICK "+e);
}
```

```
return j;
}
/**
 * Sets a record of the actual commands sent to the vehicle.
 * @param comm
 * The actual commands sent to the vehicle by the program.
 */
/*public void setSentCommands(int[] comm){
sent_commands = comm;
}*/
/**
 * Gets the commands collected from the Joystick.
 * @return
 * [Pitch,Roll,Yaw,Thrust] in manual control. [Pitch,Roll,Yaw,Y_Reference] in automatic
 * (pitch, roll and yaw are used to compute increments to reference later).
 */
public int[] getCommands(){
return new int[]{pitch,roll,yaw,thr};
}
public void addVehicle(int ID, int cport, Vec3 x0){
Controller c = new Controller(this,ID,cport,x0);
conts.add(c);
imu.addVehicle(c);
comm.addVehicle(c);
displays.add(new VehicleGUI(c));
mainpanel.setLayout(new GridLayout(1,displays.size()));
mainpanel.add(displays.get(displays.size()-1));
mainpanel.repaint();

}

}
```

```
package Autopilot;

import java.util.ArrayList;

/**
 * @author Erich Mueller
*/

/**
```

```
 *Thread which continually polls the onboard IMU for data, sends commands to the vehicle,
 *and distributes data to the GUI, the Controller and the DataWriter.
**/
public class IMUThread extends Thread{

private MasterFrame frame;
private ArrayList<Controller> conts;
private Rate_Filter filter;
private Quat quat;
private int command_update_period = 50;
private int imu_poll_period = 50;
private int gui_update_period = 200;
private double command_last = 0;
private double imu_last = 0;
private double gui_update_last = 0;
private boolean motor_flag = false;
private int motor_num = 0;
private boolean adding_flag = false;
private Controller adding_cont;
private boolean removing_flag = false;
private int removing_index = 0;
private int num_vehicles = 0;
private boolean first_command = true, pretraj = false;;
private long t0;
private Vec3 ratebias;


/**
 *
 * @param f
 * The frame containing the GUI to be updated by this thread.
 * @param c
 * The flight controller to be update by this thread.
 */
public IMUThread(MasterFrame f){
conts = new ArrayList<Controller>();
frame = f;

}

public void run(){

while(true){
double time = Util.getTimeMils();

if (time - imu_last > imu_poll_period){
for(int i=0; i<num_vehicles; i++){
conts.get(i).updateIMUData();
}
imu_last = time;
}
if(time - command_last > command_update_period){
command();
command_last = time;
}
if(motor_flag){
conts.get(motor_num).startMotors();
motor_flag = false;
}
if (adding_flag){
add();
adding_flag = false;
}
if (removing_flag){
remove();
removing_flag = false;
}
Util.pause(5);
}

}


/**
 * Function which gathers input from the user and controller,
 * and uses the current control mode to determine
 * what commands should be sent to the Hummingbird.
 */
public void command(){

for(int i=0; i<num_vehicles; i++){
conts.get(i).sendCommands();
}
}

/**
 * Sets a flag to turn the motors on or off the next time this thread executes it's run loop.
 */
public void toggleMotors(int vnum){
motor_num = vnum;
motor_flag = true;
}

public void addVehicle(Controller c){
adding_cont = c;
adding_flag = true;
}
public void removeVehicle(int index){
```

```
removing_index = index;
removing_flag = true;
}
private synchronized void add(){
conts.add(adding_cont);
num_vehicles++;
}
private synchronized void remove(){
conts.remove(removing_index);
num_vehicles--;
}
public int getNumVehicles(){
return num_vehicles;
}
}
```

```
package Autopilot;

import java.io.FileWriter;
import java.io.IOException;

/**
 * @author Erich Mueller
*/

/**
 *Implementation of the control law for autonomous flight of the Hummingbird.
  *Continually receives state updates from the CommThread and IMUThread,
  *and reference updates from the MasterFrame.
 *
**/

public class Controller {

//private Rate_Filter ratefilter = new Rate_Filter(0.2);
//private Position_Filter posfilter = new Position_Filter(0.1);

private int controlmode;
private MasterFrame frame;

private Vec3 x = new Vec3();
private Vec3 v = new Vec3();
private Vec3 accel = new Vec3();
private Vec3 rate = new Vec3();
private Vec3 quat_rate = new Vec3();
private Quat quat = new Quat();
private Quat quat_onboard = new Quat();
private Vec3 ref = new Vec3();
private int[] sent_commands = new int[4]; //pitch,roll,yaw,thrust
private double yawref;
private double yinteg = 0;
private double yawinteg = 0;
private double zinteg = 0;
private double xinteg = 0;

private boolean computing = false;
private boolean integrating = false;
private double time_last = 0;
private int vehicleID;
private SerialComm comm;
private Trajectory traj;
private double t_elapsed;
private double t0 = 0;
private double traj_time = -1;
private short battery0, battery1;

private boolean running_test = false;
private Vec3 test_origin = new Vec3();
private double test_time = -1;
private int test_num = 0;
private double v0_test = 0;
private double t0_test = -1;
private double test_freqs[] = { 0.8,0.6,0.5,0.4};
private double input_amp = 100; //units of pitch
private double test_duration_mils = 6000;
private double test_integral = 0;
private FileWriter testwriter;
private String logfile;

//parameters associated with open-loop wake compensation
private double sigma0 = 0.16*0.16/4.0; //r^2/4
private double wake_gain = 0;
private Vec3[] prop_centers =
{new Vec3(-0.16,0,0),new Vec3(0,0,-0.16),new Vec3(0.16,0,0), new Vec3(0,0,0.16)};
private Vec3 x_upper = new Vec3(1.03,2.10,0.26); //helicopter generating disruptive wake


private boolean zeroing_attitude = false;
private double t0_att0 = 0;
private double T_att0 = 2000;
private Quat quat_onboard_0 = new Quat();
```

```
public static final int MODE_NONE = 0, MODE_MANUAL = 1, MODE_AUTO_PID = 2, MODE_AUTO_SLIDE = 3;

public Controller(MasterFrame fr, int ID, int cport){
frame = fr;
vehicleID = ID;
comm = new SerialComm(cport);
controlmode = MODE_NONE;
}
public Controller(MasterFrame fr, int ID, int cport, Vec3 x0){
frame = fr;
vehicleID = ID;
comm = new SerialComm(cport);
controlmode = MODE_NONE;
x = x0;
}
public synchronized void sendCommands(){
int[] send = getCommands();
if (comm.portnum != 0) comm.command(send[0],send[1],send[2],send[3]);
}
/**
 * Computes the commands to send to the Hummingbird based on the current state of the system.
 * @return
 * [Pitch,Roll,Yaw,Thrust] commands.
 */
public synchronized int[] getCommands(){
if (time_last == 0) {
time_last = Util.getTimeMils();
t0 = time_last;
}
double time = Util.getTimeMils();
t_elapsed = time-t0;
if (traj_time > 0 && !traj.done()){
if (traj.getNextTime() <= time-traj_time){
setReference(traj.getNextPos(),traj.getNextYaw());
traj.increment();
}
}
int dt = (int)(time-time_last);
int pitch = 0,roll = 0,yaw = 0,thrust = 0;

quat = quat_onboard_0.inverse().times(quat_onboard);

if (zeroing_attitude){

//double ka = 0.2;
//quat_onboard_0 = quat_onboard_0.timesScalar(1-ka).plus(quat_onboard.timesScalar(ka));

if (time-t0_att0 > T_att0) {
```

```
zeroing_attitude = false;
System.out.println(quat_onboard_0);
}
}

quat_onboard_0.normalize();
quat_onboard.normalize();


/*RB_State filtered = posfilter.getFiltered();
x = filtered.pos;
v = filtered.vel;

Quat qnew = filtered.quat;
quat_rate = Quat.getRate(quat,qnew,dt);
quat = filtered.quat;
rate = ratefilter.getFiltered();*/

if (controlmode == MODE_MANUAL){
int[] user = frame.getCommands();
pitch = user[0];
roll = -user[1];
yaw = user[2];
thrust = user[3];
}
else if (controlmode == MODE_AUTO_PID){


//compute commands here
Vec3 err_body = quat.transformVector(ref.minus(x));
//err_body.testPrint();
//quat_onboard.transformVector(ref.minus(x)).testPrint();
Vec3 v_body = quat.transformVector(v);
double xerr = err_body.v[0];
double zerr = err_body.v[2];
double xdot = v_body.v[0];
double zdot = v_body.v[2];

//double kp = 350;
//double kd = 700;
//double ki = 0.2;

double kp = 200;//300
double kd = 400;//600
double ki = 0.1;//0.1

if(Math.abs(zerr) > 0.05 && Math.abs(zerr) < 0.75 && integrating){
zinteg += zerr*dt;
}
```

```
if(Math.abs(xerr) > 0.05 && Math.abs(xerr) < 0.75 && integrating){
xinteg += xerr*dt;
}

pitch = -(int)(+kp*zerr-kd*zdot+ki*zinteg);
roll = (int)(kp*xerr-kd*xdot+ki*xinteg);

Vec3 xunit = new Vec3(1,0,0);
Vec3 xabs = quat.transformVectorInverse(xunit);
Vec3 x1 = new Vec3(xabs.v[0],0,xabs.v[2]);
Vec3 xref = new Vec3(Math.cos(yawref),0,Math.sin(yawref));
Vec3 cr = x1.cross(xref);

double theta = Math.asin(x1.cross(xunit).length());
if (x1.v[0] < 0) theta = Math.PI-theta;
if (x1.v[2] < 0) theta = -theta;
double yawerr = yawref-theta;

yawerr = -cr.v[1];

double kp_yaw = 900;//500;
double ki_yaw = 0.04;


yaw = (int)(kp_yaw*yawerr+ki_yaw*yawinteg);


double yerr = ref.v[1]-x.v[1];
double yrate = v.v[1];
if (Math.abs(yerr) > 0.05 && integrating){
//dt in ms
yinteg += yerr*dt;
}
if (Math.abs(yawerr) > 0.03 && integrating){
yawinteg += yawerr*dt;
}
double offset = 1500;
double kpy = 600;
double kdy = 500;
double kiy = 0.05;

//double kpy = 300;
//double kdy = 200;
//double kiy = 0.03;

thrust = (int)(offset+yerr*kpy-yrate*kdy+yinteg*kiy);

if(thrust < 0) thrust = 0;
if (thrust > 3000) thrust = 3000;
```

```
}
else if (controlmode == MODE_AUTO_SLIDE){

//compute commands here
Vec3 err_body = quat.transformVector(ref.minus(x));
//err_body.testPrint();
//quat_onboard.transformVector(ref.minus(x)).testPrint();
Vec3 v_body = quat.transformVector(v);
double xerr = err_body.v[0];
double zerr = err_body.v[2];
double xdot = v_body.v[0];
double zdot = v_body.v[2];


double ki = 0.1;//0.1

if(Math.abs(zerr) > 0.05 && Math.abs(zerr) < 0.75 && integrating){
zinteg += zerr*dt;
}
if(Math.abs(xerr) > 0.05 && Math.abs(xerr) < 0.75 && integrating){
xinteg += xerr*dt;
}

//sliding controller for pitch and roll
double Lpr = 400;
double sp = zdot - Lpr*zerr;
double sr = xdot - Lpr*xerr;
double etapr = 100;
double phi = 75;

pitch = -(int)(-Lpr*zdot - etapr*Util.sat(sp/phi) + zinteg*ki);
roll = (int)(-Lpr*xdot - etapr*Util.sat(sr/phi) + xinteg*ki);


Vec3 xunit = new Vec3(1,0,0);
Vec3 xabs = quat.transformVectorInverse(xunit);
Vec3 x1 = new Vec3(xabs.v[0],0,xabs.v[2]);
Vec3 xref = new Vec3(Math.cos(yawref),0,Math.sin(yawref));
Vec3 cr = x1.cross(xref);

double theta = Math.asin(x1.cross(xunit).length());
if (x1.v[0] < 0) theta = Math.PI-theta;
if (x1.v[2] < 0) theta = -theta;
double yawerr = yawref-theta;

yawerr = -cr.v[1];

double kp_yaw = 900;//500;
double ki_yaw = 0.04;
```

```
yaw = (int)(kp_yaw*yawerr+ki_yaw*yawinteg);


double yerr = ref.v[1]-x.v[1];
double yrate = v.v[1];
if (Math.abs(yerr) > 0.05 && integrating){
//dt in ms
yinteg += yerr*dt;
}
if (Math.abs(yawerr) > 0.03 && integrating){
yawinteg += yawerr*dt;
}
double offset = 1500;
double kiy = 0.05;


//sliding mode controller for thrust
double Ly = 500;
double sy = yrate - Ly*yerr;
double etay = 200;
double phiy = 50;

thrust = (int)(offset - Ly*yrate - etay*Util.sat(sy/phiy) +yinteg*kiy);

if(thrust < 0) thrust = 0;
if (thrust > 3000) thrust = 3000;
}

time_last = time;

//add wake corrections
/* int[] c = getWakeCorrections();
pitch += c[0];
roll += c[1];
yaw += c[2];
thrust += c[3];

*/

sent_commands = new int[]{pitch,-roll,yaw,thrust};
return new int[]{pitch,-roll,yaw,thrust};

}

/**
 * Determines the pitch,roll,yaw and thrust corrections for feed-forward wake compensation.
 * @return Corrections to pitch, roll, yaw and thrust.
```

```
*/
public int[] getWakeCorrections(){
Vec3[] c = new Vec3[4];
double[] u = new double[4];
double sigma = sigma0;
for(int i=0; i<4; i++){
c[i] = x_upper.minus(x.plus(quat.transformVector(prop_centers[i]))); //position
u[i] = wake_gain*Math.exp(-(c[i].v[0]*c[i].v[0]
+ c[i].v[2]*c[i].v[2])/(2*sigma))/(2*Math.PI*sigma);
}
int[] p = new int[4];
p[0] = (int)(-u[1] + u[3]); //pitch
p[1] = (int)(u[0] - u[2]); //roll
p[2] = 0;//(int)(u[0] - u[1] + u[2] - u[3]); //yaw
p[3] = (int)(u[0] + u[1] + u[2] + u[3]); //thrust

return p;
}

public void incrementWakeGain(double inc){
wake_gain += inc;
System.out.println(wake_gain);
}

/**
 * Updates the controller's internal representation of the state of the system
 * not including angular rate.
 * @param s
 * The new rigid body state to be used by the controller.
 */
/*public void updateStateOptitrack(RB_State s){
if (!computing){
x = s.pos;
v = s.vel;
quat = s.quat;
}
}*/
/**
 * Updates the controller's internal representation of the angular rate of the Vehicle.
 * @param r
 * The new angular rate to be used by the controller.
 */
public void updateStateRate(Vec3 r){
if (!computing){
rate = r;
}
}
public void updateIMUData(){
newImuData(comm.pollIMUData());
}
public synchronized void newImuData(InternalData dat){
if (dat.raw_valid) {
double kr = 0.8;
rate = rate.times(1-kr).plus(dat.angvel.times(kr));
}
if (dat.est_valid){
if(zeroing_attitude){
double kq = 0.1;
quat_onboard_0 = quat_onboard_0.timesScalar(1-kq).plus(dat.getQuat().timesScalar(kq));
}
else{
double kq = 0.2;
quat_onboard = quat_onboard.timesScalar(1-kq).plus(dat.getQuat().timesScalar(kq));
}
}
if (dat.status_valid){
battery0 = dat.bat0;
battery1 = dat.bat1;
}

}
public synchronized void zeroAttitude(){
t0_att0 = Util.getTimeMils();
zeroing_attitude = true;
}
/**
 * Adjusts the zero-reference heading angle manually.
 * @param d The angle by which to adjust the zero-heading.
 */
public synchronized void adjustQuatOffset(double d){
Quat q = new Quat(0,Math.sin(d/2.0),0,Math.cos(d/2.0));
quat_onboard_0 = q.inverse().times(quat_onboard_0);
}

public synchronized void newOptitrackData(RB_Pos pos){
//posfilter.pushData(pos);
}

public synchronized void positionMeasurement(Vec3 xn, double dt){
Vec3 vn = xn.minus(x).divideBy(dt);
double kx = 0.8;
double kv = 0.8;
x = x.times(1-kx).plus(xn.times(kx));
v = v.times(1-kv).plus(vn.times(kv));
}
public synchronized void newCommands(int[] commands){
sent_commands = commands;
}
```

```
public synchronized void startMotors(){
comm.startMotors();
}

/**
 * Directly sets the reference position and yaw angle for the vehicle to track.
 * @param pos
 * Position in absolute (Lab) coordinates to be tracked.
 * @param yaw
 * Angle to track, in radians, between from the absolute x axis to the body frame x axis,
 * positive toward the absolute z axis (CCW from above).
 */
public synchronized void setReference(Vec3 pos, double yaw){
ref = pos;
yawref = yaw;
forceRefInRange();
}
/**
 * Sets the state of the integral controllers, whether or not integration of error should be active.
 * @param b
 * True if integration should be active.
 */
public void setIntegrating(boolean b){
integrating = b;
}
/**
 * Incrementally increases the reference position.
 * @param inc
 * The positional increment to change the reference by.
 * @param dyaw
 * The angular (radian) increment to reference yaw.
 */
public synchronized void incrementReference(Vec3 inc, double dyaw){
ref.plusEquals(inc);
yawref += dyaw;
if (yawref > Math.PI) yawref = -2*Math.PI+yawref;
if (yawref < -Math.PI) yawref = 2*Math.PI+yawref;
forceRefInRange();
}
/**
 * Directly sets the reference altitude.
 * @param y
 * The reference altitude to be tracked.
 */
public synchronized void setRefY(double y){
ref.v[1] = y;
forceRefInRange();
}
/**
```

```
 * Internal method used whenever a change to the reference is made
 * to ensure that the reference is within an acceptable range.
 */
public void forceRefInRange(){
/*if (ref.v[0] < 0.5) ref.v[0] = 0.5;
else if (ref.v[0] > 2.5) ref.v[0] = 2.5;
if (ref.v[1] < 0) ref.v[1] = 0;
else if (ref.v[1] > 1.5) ref.v[1] = 1.5;
if (ref.v[2] < -1.5) ref.v[2] = -1.5;
else if (ref.v[2] > 2) ref.v[2] = 2;
if (yawref > Math.PI) yawref = Math.PI;
else if (yawref < -Math.PI) yawref = -Math.PI;*/
}

public synchronized Vec3 getPosRef(){
return new Vec3(ref);
}
public synchronized double getYawRef(){
return yawref;
}
public synchronized Vec3 getPosition(){
return new Vec3(x);
}
public synchronized Vec3 getVelocity(){
return new Vec3(v);
}
public synchronized Vec3 getIMURate(){
return new Vec3(rate);
}
public synchronized Vec3 getAccel(){
return new Vec3(accel);
}
public synchronized Vec3 getQuatRate(){
return new Vec3(quat_rate);
}
public synchronized Quat getQuat(){
return new Quat(quat);
}
public synchronized double[] getSentCommands(){
return new double[]{sent_commands[0],sent_commands[1],sent_commands[2],sent_commands[3]};
}
public synchronized Quat getQuatOnboard(){
return new Quat(quat_onboard);
}
public synchronized short[] getBatteryCharges(){
return new short[]{battery0,battery1};
}
public synchronized int getID(){
return vehicleID;
}
public synchronized int getPort(){
return comm.portnum;
}
public synchronized void setControlMode(int mode){
controlmode = mode;
if (controlmode == MODE_AUTO_PID || controlmode == MODE_AUTO_SLIDE){
ref = x;
yawref = 0;
integrating = true;
}
else{
integrating = false;
yinteg = 0;
}
}
public synchronized void setTrajectory(Trajectory t){
traj = t;
}
public synchronized void startTrajectory(){
traj_time = Util.getTimeMils();
}
public synchronized void stopTrajectory(){
traj_time = -1;
}
public synchronized void runNextTest(){
running_test = true;
}
public synchronized void stopTest(){
running_test = false;
}
public synchronized boolean isTesting(){
return running_test;
}
public synchronized void goToTrajStart(){
setReference(traj.positions.get(0),traj.yaws.get(0));
}
public synchronized float getElapsedTime(){
return (float)t_elapsed;
}
public synchronized void setEpoch(double epoch){
t0 = epoch;
}
public synchronized void setLogFile(String s){
logfile = s;
}

}
```

```
package Autopilot;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.net.SocketException;
import java.util.ArrayList;

/**
 * @author Erich Mueller
*/

/**
 *Thread which listens on a UDP socket for optitrack state data broadcast by the
 *c++ application and distributes the data to the GUI, the Controller and the DataWriter.
**/
public class CommThread extends Thread{

private MulticastSocket socket;
private DatagramPacket packet;
private MasterFrame frame;
private ArrayList<Controller> conts;
private RB_Pos last_pos;
private Position_Filter pos_filter;
private int command_update_period = 20; //milliseconds
private int udp_comm_period = 1;
private int gui_update_period = 200;
private long command_last = 0;
private long udp_last = 0;
private long gui_update_last = 0;
private boolean adding_flag = false;
private Controller adding_cont;
private boolean removing_flag = false;
private int removing_index;

private int tracking_type;

private double dist_max = 0.3; //max distance in m for a point to still be considered for match

public static final int MODE_STICK = 0;
public static final int TRACK_RB = 0, TRACK_1PT = 1;

private DataWriter writer;

/**
 *
 * @param f
 * The frame containing the GUI to be updated by this thread.
 * @param c
 * The controller to be updated by this thread.
 */
public CommThread(MasterFrame f){
frame = f;
conts = new ArrayList<Controller>();
pos_filter = new Position_Filter(0.1);
tracking_type = TRACK_1PT;
}

public void run(){
try{

socket = new MulticastSocket(1511);
socket.joinGroup(InetAddress.getByName("239.255.42.99"));
packet = new DatagramPacket(new byte[256],256);

last_pos = new RB_Pos();
double t_last = 0;
while(true){
double time = Util.getTimeMils();
socket.receive(packet);
byte[] data = packet.getData();

int id = Util.bas(data[1],data[0]);
if (id == 7){ //NatNet Motion Capture Frame

if (tracking_type == TRACK_RB){ //rigid body tracking

int n_rbs = Util.bai(data[19],data[18],data[17],data[16]); //number of rigid bodies tracked

int p = 20;
for(int i=0; i<n_rbs; i++){
int rbid = Util.bai(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float x = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float y = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float z = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;

float qx = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float qy = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float qz = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float qw = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;


if (!(x*x + y*y + z*z + qx*qx + qy*qy + qz*qz + qw*qw < 0.000001)){ //valid frames
for(int j=0; j<conts.size(); j++){
if (conts.get(i).getID() == rbid){
RB_Pos newpos = new RB_Pos(new Vec3(x,y,z),new Quat(qx,qy,qz,qw),rbid,time);
conts.get(i).newOptitrackData(newpos);
last_pos = newpos;
break;
}
}
}

int n_marks = Util.bai(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
p += 4*(5*n_marks + 1); //skip associated marker data
}

}
else if (tracking_type == TRACK_1PT){ //Single point position tracking

int n_pts = Util.bai(data[15],data[14],data[13],data[12]);

Vec3[] pts = new Vec3[n_pts];

int p = 16;
for(int i=0; i<n_pts; i++){
float x = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float y = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
float z = Util.baf1(data[p+3],data[p+2],data[p+1],data[p]); p+=4;
pts[i] = new Vec3(x,y,z);
}
if (n_pts > 0){
Vec3[] locs = new Vec3[conts.size()];
Vec3 loc = new Vec3();
double dmin = 1E99;
int imin = 0;
for(int i=0; i<conts.size(); i++){
loc = conts.get(i).getPosition();

dmin = 1E99;
imin = -1;
for(int j=0; j<n_pts; j++){
double d = loc.minus(pts[j]).length();
if (d < dmin && d < dist_max){
dmin = d;
imin = j;
}
}
if (imin != -1) locs[i] = pts[imin];
else locs[i] = loc;
}

for(int i=0; i<conts.size(); i++){
//RB_Pos newpos = new RB_Pos(locs[i],new Quat(),conts.get(i).getID(),time);
//conts.get(i).newOptitrackData(newpos);
conts.get(i).positionMeasurement(locs[i],(time-t_last)/(1000.0));

}
}

}

}

if (adding_flag){
conts.add(adding_cont);
adding_flag = false;
}
if (removing_flag){
conts.remove(removing_index);
removing_flag = false;
}

t_last = time;
Util.pause(udp_comm_period);
}

}
catch(SocketException e){
System.out.println("Receiving UDP socket failed to initialize "+e);
}
catch(IOException e){
System.out.println("UDP packet reception failed "+e);
}
}
/**
 * Sets the DataWriter to be updated by this thread.
 * @param w
 * The DataWriter to be updated.
 */
public void setWriter(DataWriter w){
writer = w;
}
public void addVehicle(Controller c){
adding_cont = c;
adding_flag = true;
}
```

```
public void removeVehicle(int i){
removing_index = i;
removing_flag = true;
}
}
```

```
package Autopilot;

public class Updater extends Thread{

private MasterFrame frame;
private int frame_update_period = 200;
private int writer_update_period = 20;
private double time = 0;
private double tframe_last = 0;
private double twriter_last = 0;
private DataLogger logger;
private boolean writing = false;

public Updater(MasterFrame f){
frame = f;
}

public void run(){
while(true){
time = Util.getTimeMils();
if (time-tframe_last > frame_update_period){
frame.updateGUI();
tframe_last = time;
}
if (writing && time-twriter_last > writer_update_period){
logger.writeData();
twriter_last = time;
}
Util.pause(5);
}
}
public void addVehicle(Controller cont){
logger.addVehicle(cont);
}
public void startDataLogging(DataLogger l){
writing = true;
logger = l;
}
}
```

```
package Autopilot;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;

import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFileChooser;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class VehicleGUI extends JPanel{

private JPanel datapanel;
private JPanel controlpanel;
private JPanel graphpanel;
private ArrayList<Graph> graphs;
private ArrayList<JLabel> labels;
private Controller cont;

private JComboBox modebox;
private JTextField xref,yref,zref,yawref;
private JButton motorbutton;
private JButton trajbutton;
private JButton readybutton;
private JButton startbutton;
private JButton testbutton;
private JButton quat0button;
private JButton nudgeleft, nudgeright;

private VehicleGUI me = this;

public VehicleGUI(Controller c){
super();
cont = c;
setLayout(new BorderLayout());
createAll();
add(graphpanel,BorderLayout.CENTER);
add(datapanel,BorderLayout.PAGE_START);
add(controlpanel,BorderLayout.PAGE_END);
}

public void createAll(){
graphs = new ArrayList<Graph>();
labels = new ArrayList<JLabel>();

graphs.add(new Graph("Time", "Position"));
graphs.get(0).addSeries("X",Color.RED);
graphs.get(0).addSeries("Y",Color.BLUE);
graphs.get(0).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Velocity"));
graphs.get(1).addSeries("X",Color.RED);
graphs.get(1).addSeries("Y",Color.BLUE);
graphs.get(1).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Angular Rate"));
graphs.get(2).addSeries("X",Color.RED);
graphs.get(2).addSeries("Y",Color.BLUE);
graphs.get(2).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Optitrack Angvel (rad/s)"));
graphs.get(3).addSeries("X",Color.RED);
graphs.get(3).addSeries("Y",Color.BLUE);
graphs.get(3).addSeries("Z",Color.GREEN);
graphs.add(new Graph("Time", "Quaternion"));
graphs.get(4).addSeries("Q1",Color.RED);
graphs.get(4).addSeries("Q2",Color.BLUE);
graphs.get(4).addSeries("Q3",Color.GREEN);
graphs.get(4).addSeries("Q4",Color.CYAN);
graphs.add(new Graph("Time", "Quat Internal"));
graphs.get(5).addSeries("Q1",Color.RED);
graphs.get(5).addSeries("Q2",Color.BLUE);
graphs.get(5).addSeries("Q3",Color.GREEN);
graphs.get(5).addSeries("Q4",Color.CYAN);
graphs.add(new Graph("Time", "Commands"));
graphs.get(6).addSeries("Pitch",Color.RED);
graphs.get(6).addSeries("Roll",Color.BLUE);
graphs.get(6).addSeries("Yaw",Color.GREEN);
graphs.get(6).addSeries("Thrust",Color.CYAN);
```

```
graphpanel = new JPanel();

/*graphpanel.setLayout(new GridLayout(graphs.size(),1));
for(int i=0; i<graphs.size(); i++){
graphpanel.add(graphs.get(i));
}*/
graphpanel.setLayout(new GridLayout(3,1));
graphpanel.add(graphs.get(0));
graphpanel.add(graphs.get(4));
graphpanel.add(graphs.get(6));

datapanel = new JPanel();
datapanel.setPreferredSize(new Dimension(10,100));
datapanel.setLayout(new GridLayout(graphs.size()+1,1));
JLabel titlelabel = new JLabel("Rigid Body #: "+cont.getID()+" , COM"+cont.getPort());
datapanel.add(titlelabel);
for(int i=0; i<graphs.size(); i++){
labels.add(new JLabel("----"));
datapanel.add(labels.get(i));
}
modebox = new JComboBox();
modebox.addItem("None");
modebox.addItem("Joystick");
modebox.addItem("Automatic - PID");
modebox.addItem("Automatic - Sliding Mode");
modebox.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.setControlMode(modebox.getSelectedIndex());
}
});
xref = new JTextField(5);
yref = new JTextField(5);
zref = new JTextField(5);
yawref = new JTextField(5);

xref.setText("0");
yref.setText("0");
zref.setText("0");
yawref.setText("0");


xref.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.setReference(new Vec3(Double.parseDouble(xref.getText()),Double.parseDouble(yref.getText()),
Double.parseDouble(zref.getText())),Double.parseDouble(yawref.getText()));
}
});
yref.addActionListener(new ActionListener(){
```

```
public void actionPerformed(ActionEvent e){
cont.setReference(new Vec3(Double.parseDouble(xref.getText()),Double.parseDouble(yref.getText()),
Double.parseDouble(zref.getText())),Double.parseDouble(yawref.getText()));
}
});
zref.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.setReference(new Vec3(Double.parseDouble(xref.getText()),Double.parseDouble(yref.getText()),
Double.parseDouble(zref.getText())),Double.parseDouble(yawref.getText()));
}
});
yawref.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.setReference(new Vec3(Double.parseDouble(xref.getText()),Double.parseDouble(yref.getText())
,Double.parseDouble(zref.getText())),Double.parseDouble(yawref.getText()));
}
});
motorbutton = new JButton("Start/Stop Motors");
motorbutton.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.startMotors();
}
});
trajbutton = new JButton("Load Trajectory");
trajbutton.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
JFileChooser f = new JFileChooser(
"C:\\Documents and Settings\\Erich\\My Documents\\eclipse\\Trajectories");
int returnVal = f.showOpenDialog(me);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
     cont.setTrajectory(new Trajectory(f.getSelectedFile()));
     readybutton.setEnabled(true);
     System.out.println("Trajectory Loaded");
    }

}
});
readybutton = new JButton("Go To Trajectory Start");
readybutton.setEnabled(false);
readybutton.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.goToTrajStart();
startbutton.setEnabled(true);
System.out.println("Seeking Trajectory Start Location");
}
});
startbutton = new JButton("Start Trajectory");
startbutton.setEnabled(false);
startbutton.addActionListener(new ActionListener(){
```

```
public void actionPerformed(ActionEvent e){
cont.startTrajectory();
System.out.println("Beginning Trajectory");
}
});
quat0button = new JButton("Zero Attitude");
quat0button.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.zeroAttitude();
}
});
nudgeleft = new JButton("Left");
nudgeleft.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.adjustQuatOffset(0.1);
}
});
nudgeright = new JButton("Right");
nudgeright.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
cont.adjustQuatOffset(-0.1);
}
});

datapanel.addMouseListener(new MouseListener(){
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}
public void mouseReleased(MouseEvent e){}
public void mousePressed(MouseEvent e){
me.requestFocus();
me.requestFocusInWindow();
}
});
this.addKeyListener(new KeyListener(){
public void keyTyped(KeyEvent e){}
public void keyPressed(KeyEvent e){}
public void keyReleased(KeyEvent e){
switch(e.getKeyCode()){
case (KeyEvent.VK_UP):
cont.incrementWakeGain(5);
break;
case (KeyEvent.VK_DOWN):
cont.incrementWakeGain(-5);
break;
case (KeyEvent.VK_EQUALS):
cont.incrementWakeGain(1);
break;
case (KeyEvent.VK_MINUS):
```

```
cont.incrementWakeGain(-1);
break;
}
}
});

JPanel adjustpan = new JPanel();
adjustpan.add(new JLabel("Adjust Zero-heading Angle"));
adjustpan.add(nudgeleft);
adjustpan.add(nudgeright);

controlpanel = new JPanel();
controlpanel.setLayout(new GridLayout(8,2));
controlpanel.add(new JLabel("Control Mode "));
controlpanel.add(modebox);
controlpanel.add(new JLabel("X Ref"));
controlpanel.add(xref);
controlpanel.add(new JLabel("Y Ref"));
controlpanel.add(yref);
controlpanel.add(new JLabel("Z Ref"));
controlpanel.add(zref);
controlpanel.add(new JLabel("yaw Ref"));
controlpanel.add(yawref);
controlpanel.add(motorbutton);
controlpanel.add(trajbutton);
controlpanel.add(quat0button);
controlpanel.add(adjustpan);
controlpanel.add(readybutton);
controlpanel.add(startbutton);
}

public void updateGUI(double t){
Vec3 x = cont.getPosition();
Vec3 v = cont.getVelocity();
Vec3 imu = cont.getIMURate();
double[] sent = cont.getSentCommands();
Quat q = cont.getQuat();
Quat qo = cont.getQuatOnboard();
Vec3 qrate = cont.getQuatRate();
Vec3 xr = cont.getPosRef();
double yr = cont.getYawRef();


labels.get(0).setText("Position: "+x.toString());
labels.get(1).setText("Velocity: "+v.toString());
labels.get(2).setText("IMU Rate: "+imu.toString());
labels.get(3).setText("dq/dt Rate: "+qrate.toString());
labels.get(4).setText("OT Quat: "+q.toString());
labels.get(5).setText("Onboard Quat: "+qo.toString());
labels.get(6).setText("Commands: "+sent[0]+" , "+sent[1]+" , "+sent[2]+" , "+sent[3]);

if(!xref.isFocusOwner()){
xref.setText(xr.v[0]+"");
}
if(!yref.isFocusOwner()){
yref.setText(xr.v[1]+"");
}
if(!zref.isFocusOwner()){
zref.setText(xr.v[2]+"");
}
if(!yawref.isFocusOwner()){
yawref.setText(yr+"");
}


graphs.get(0).pushDataPoints(t,x.v);
graphs.get(1).pushDataPoints(t,v.v);
graphs.get(2).pushDataPoints(t,imu.v);
graphs.get(3).pushDataPoints(t,qrate.v);
graphs.get(4).pushDataPoints(t,q.toList());
graphs.get(5).pushDataPoints(t,qo.toList());
graphs.get(6).pushDataPoints(t,sent);
for(int i=0; i<graphs.size(); i++){
graphs.get(i).autoFit();
}
}


}
```

```
package Autopilot;

import gnu.io.CommPort;
import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
/**
 * @author Erich Mueller
 */

/**
 *Provides functions which allow the Hummingbird to be polled for IMU data
 * and commanded in serial-controlled mode.
**/
public class SerialComm {

private OutputStream out;
private InputStream in;
private int poll_mode;
public int portnum;

public static final int POLL_IMU_ONLY = 0, POLL_IMU_EST = 1, POLL_ALL = 2;

public SerialComm(int pnum){
poll_mode = POLL_IMU_EST;
portnum = pnum;
if (portnum != 0)initialize();
}

public void initialize(){
//initializes serial communication with the UFO

try{
CommPortIdentifier portIdentifier = CommPortIdentifier.getPortIdentifier("COM"+portnum);
        if (portIdentifier.isCurrentlyOwned()){
            System.out.println("Error: Port is currently in use");
        }
        else{
            CommPort commPort = portIdentifier.open("Hummingbird Comm",2000);
            SerialPort serialPort = (SerialPort) commPort;
            serialPort.setSerialPortParams(57600,SerialPort.DATABITS_8,
             SerialPort.STOPBITS_1,SerialPort.PARITY_NONE);
            in = serialPort.getInputStream();
            out = serialPort.getOutputStream();
        }
}
catch(Exception e){
System.out.println("ERROR ESTABLISHING SERIAL CONNECTION "+e);
}
}
public void command(int pitch, int roll, int yaw, int thrust){
if (pitch < -2047) pitch = -2047;
else if (pitch > 2047) pitch = 2047;
if (roll < -2047) roll = -2047;
else if (roll > 2047) roll = 2047;
```

```
if (yaw < -2047) yaw = -2047;
else if (yaw > 2047) yaw = 2047;
if (thrust < 0) thrust = 0;
else if (thrust > 4095) thrust = 4095;

writeCommand((short)pitch,(short)roll,(short)yaw,(short)thrust);
}

private void writeCommand(short pitch, short roll, short yaw, short thrust){
//writes a command to the outputstream for the serial communication link with the
//specified command data. pitch, roll and yaw range from -2047 to 2047
//thrust ranges from 0 to 4095

byte[] packet = new byte[17];
packet[0] = '>';
packet[1] = '*';
packet[2] = '>';
packet[3] = 'd';
packet[4] = 'i';
byte[] p = Util.sba(pitch);
byte[] r = Util.sba(roll);
byte[] y = Util.sba(yaw);
byte[] t = Util.sba(thrust);
short ct = 0x0F;
byte[] ctrl = Util.sba(ct);
short chk = (short)(pitch+roll+yaw+thrust+ct+0xAAAA);
byte[] ch = Util.sba(chk);
packet[5] = p[1];
packet[6] = p[0];
packet[7] = r[1];
packet[8] = r[0];
packet[9] = y[1];
packet[10] = y[0];
packet[11] = t[1];
packet[12] = t[0];
packet[13] = ctrl[1];
packet[14] = ctrl[0];
packet[15] = ch[1];
packet[16] = ch[0];

try{
out.write(packet);
out.flush();
}
catch(IOException e){
System.out.println("ERROR TRANSMITTING PACKET "+e);
}
}
public void startMotors(){
```

```
long t = System.nanoTime();
while(System.nanoTime()-t < 1*1000*1000*1000){
command(0,0,-2047,0);
Util.pause(50);
}
}

public InternalData pollIMUData(){
byte[] request = new byte[6];
request[0] = '>';
request[1] = '*';
request[2] = '>';
request[3] = 'p';
char c = (0x0002); //request imu data
if (poll_mode == POLL_IMU_EST) c = 0x0002 | 0x0004;
else if (poll_mode == POLL_ALL) c = 0x0001 | 0x0002 | 0x0004;

byte[] b = Util.cba(c);
request[4] = b[1];
request[5] = b[0];
//**** Note that the byte transmit order is little-endian

write(request);
Util.pause(10);
byte by[] = new byte[1];
by[0] = 0;
int i = 0;
InternalData imu = new InternalData();
imu.status_valid = false;
imu.raw_valid = false;
imu.est_valid = false;

while(i <= poll_mode){
while(by[0] != '>'){
read(by);
}
i++;
byte[] no = new byte[5];
read(no);
short len = Util.bas(no[3],no[2]);

//System.out.println(no[0] + ","+no[1]+","+no[2]+","+no[3]+","+no[4] + " , "+len);
if (len == 16){
//11 status data
imu.status_valid = true;
byte[] readin = new byte[16];
read(readin);
imu.bat0 = Util.bas(readin[1], readin[0]);
imu.bat1 = Util.bas(readin[3], readin[2]);
imu.status = Util.bas(readin[5], readin[4]);
imu.cpu_load = Util.bas(readin[7], readin[6]);
imu.compass_on = readin[8];
imu.checksum_error = readin[9];
imu.flying = readin[10];
imu.motors_on = readin[11];
imu.flight_mode = Util.bas(readin[13],readin[12]);
imu.uptime = Util.bas(readin[15],readin[14]);
read(new byte[5]);
}
else if (len == 28){
//imu raw data
imu.raw_valid = true;
byte[] readin = new byte[28];
read(readin);
imu.nanotime = System.nanoTime();
imu.pressure = Util.bai(readin[3],readin[2],readin[1],readin[0]);
imu.angvel.v[0] = Util.bas(readin[5],readin[4]);
imu.angvel.v[1] = Util.bas(readin[7],readin[6]);
imu.angvel.v[2] = Util.bas(readin[9],readin[8]);
imu.mag.v[0] = Util.bas(readin[11],readin[10]);
imu.mag.v[1] = Util.bas(readin[13],readin[12]);
imu.mag.v[2] = Util.bas(readin[15],readin[14]);
imu.acc.v[0] = Util.bas(readin[17],readin[16]);
imu.acc.v[1] = Util.bas(readin[19],readin[18]);
imu.acc.v[2] = Util.bas(readin[21],readin[20]);
imu.temp_gyro = Util.bai((byte)0,(byte)0,readin[23],readin[22]);
imu.temp_adc = Util.bai(readin[27],readin[26],readin[25],readin[24]);
imu.temp_adc = Util.bai(readin[27],readin[26],readin[25],readin[24]);

read(new byte[5]);
}

else if (len == 92){
//IMU calc data
imu.est_valid = true;
byte[] readin = new byte[92];
read(readin);
imu.pitch = Util.basi(readin[3],readin[2],readin[1],readin[0]);
imu.roll = Util.basi(readin[7],readin[6],readin[5],readin[4]);
imu.yaw = Util.basi(readin[11],readin[10],readin[9],readin[8]);
imu.pitchrate = Util.basi(readin[15],readin[14],readin[13],readin[12]);
imu.rollrate = Util.basi(readin[19],readin[18],readin[17],readin[16]);
imu.yawrate = Util.basi(readin[23],readin[22],readin[21],readin[20]);
imu.acc_xc = Util.bas(readin[25],readin[24]);
imu.acc_yc = Util.bas(readin[27],readin[26]);
imu.acc_zc = Util.bas(readin[29],readin[28]);
imu.acc_x = Util.bas(readin[31],readin[30]);
imu.acc_y = Util.bas(readin[33],readin[32]);
imu.acc_z = Util.bas(readin[35],readin[34]);
imu.pitch_acc = Util.basi(readin[39],readin[38],readin[37],readin[36]);
imu.roll_acc = Util.basi(readin[43],readin[42],readin[41],readin[40]);
imu.acc_abs = Util.basi(readin[47],readin[46],readin[45],readin[44]);
imu.Bx = Util.basi(readin[51],readin[50],readin[49],readin[48]);
imu.By = Util.basi(readin[55],readin[54],readin[53],readin[52]);
imu.Bz = Util.basi(readin[59],readin[58],readin[57],readin[56]);
imu.heading = Util.basi(readin[63],readin[62],readin[61],readin[60]);
imu.speed_x = Util.basi(readin[67],readin[66],readin[65],readin[64]);
imu.speed_y = Util.basi(readin[71],readin[70],readin[69],readin[68]);
imu.speed_z = Util.basi(readin[75],readin[74],readin[73],readin[72]);
imu.height = Util.basi(readin[79],readin[78],readin[77],readin[76]);
imu.dheight = Util.basi(readin[83],readin[82],readin[81],readin[80]);
imu.dheight_pressure = Util.basi(readin[87],readin[86],readin[85],readin[84]);
imu.height_pressure = Util.basi(readin[91],readin[90],readin[89],readin[88]);

imu.yaw -= 88000;
if (imu.yaw < 0) imu.yaw = 360000+imu.yaw;
Quat q = new Quat(imu.pitch*Math.PI/180000.0,imu.roll*Math.PI/180000.0,imu.yaw*Math.PI/180000.0);
Quat q1 = new Quat(0.707,0,0,0.707);
//q1 = q1.times(new Quat(0,0,-0.707,0.707));
q = q.inverse();
q = q1.inverse().times(q);
q = q.inverse();
Vec3 x = new Vec3(1,0,0);
x = q.transformVector(x);
//x = q1.transformVector(x);
//x.testPrint();
//System.out.println("Pitch: "+imu.pitch/1000.0+" Roll: "+imu.roll/1000.0+" Yaw: "+imu.yaw/1000.0);
read(new byte[5]);
}
}
return imu;
}
/**
 * Polls only the onboard filtered estimates.
 * @return InternalData with only estimated quantities valid.
 */
public InternalData pollInternalEstimate(){
byte[] request = new byte[6];
request[0] = '>';
request[1] = '*';
request[2] = '>';
request[3] = 'p';
char c = (0x0004); //request imuCalcData

byte[] b = Util.cba(c);
request[4] = b[1];
request[5] = b[0];
```

```
//**** Note that the byte transmit order is little-endian

write(request);
Util.pause(10);
byte by[] = new byte[1];
by[0] = 0;
int i = 0;
InternalData imu = new InternalData();
imu.status_valid = false;
imu.raw_valid = false;
imu.est_valid = false;

while(by[0] != '>'){
read(by);
}
i++;
byte[] no = new byte[5];
read(no);
short len = Util.bas(no[3],no[2]);
if (len == 92){
//IMU calc data
imu.est_valid = true;
byte[] readin = new byte[92];
read(readin);
imu.pitch = Util.basi(readin[3],readin[2],readin[1],readin[0]);
imu.roll = Util.basi(readin[7],readin[6],readin[5],readin[4]);
imu.yaw = Util.basi(readin[11],readin[10],readin[9],readin[8]);
imu.pitchrate = Util.basi(readin[15],readin[14],readin[13],readin[12]);
imu.rollrate = Util.basi(readin[19],readin[18],readin[17],readin[16]);
imu.yawrate = Util.basi(readin[23],readin[22],readin[21],readin[20]);
imu.acc_xc = Util.bas(readin[25],readin[24]);
imu.acc_yc = Util.bas(readin[27],readin[26]);
imu.acc_zc = Util.bas(readin[29],readin[28]);
imu.acc_x = Util.bas(readin[31],readin[30]);
imu.acc_y = Util.bas(readin[33],readin[32]);
imu.acc_z = Util.bas(readin[35],readin[34]);
imu.pitch_acc = Util.basi(readin[39],readin[38],readin[37],readin[36]);
imu.roll_acc = Util.basi(readin[43],readin[42],readin[41],readin[40]);
imu.acc_abs = Util.basi(readin[47],readin[46],readin[45],readin[44]);
imu.Bx = Util.basi(readin[51],readin[50],readin[49],readin[48]);
imu.By = Util.basi(readin[55],readin[54],readin[53],readin[52]);
imu.Bz = Util.basi(readin[59],readin[58],readin[57],readin[56]);
imu.heading = Util.basi(readin[63],readin[62],readin[61],readin[60]);
imu.speed_x = Util.basi(readin[67],readin[66],readin[65],readin[64]);
imu.speed_y = Util.basi(readin[71],readin[70],readin[69],readin[68]);
imu.speed_z = Util.basi(readin[75],readin[74],readin[73],readin[72]);
imu.height = Util.basi(readin[79],readin[78],readin[77],readin[76]);
imu.dheight = Util.basi(readin[83],readin[82],readin[81],readin[80]);
imu.dheight_pressure = Util.basi(readin[87],readin[86],readin[85],readin[84]);
imu.height_pressure = Util.basi(readin[91],readin[90],readin[89],readin[88]);

imu.yaw -= 88000;
if (imu.yaw < 0) imu.yaw = 360000+imu.yaw;
read(new byte[5]);
}

return imu;
}


public void read(byte[] b){
try{
in.read(b);
}
catch(IOException e){
System.out.println("ERROR READING TRANSMISSION "+e);
}
}
public void write(byte[] b){
try{
out.write(b);
out.flush();
}
catch(IOException e){
System.out.println("ERROR TRANSMITTING PACKET "+e);
}
}
public void close(){
try{
out.close();
in.close();
}
catch(IOException e){
System.out.println("ERROR CLOSING STREAMS "+e);
}
}
public void setPollMode(int i){
poll_mode = i;
}
}
```

Bibliography

[1] N. Roy A. Bachrach, R. He. Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, December 2009.

[2] Spencer G. Ahrens. Vision-based guidance and control of a hovering vehicle in unknown environments. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, Cambridge MA, May 2008.

[3] Erdinc Altug, James P. Ostrowski, and Robert Mahony. Control of a quadrotor helicopter using visual feedback. In *IEEE International Conference on Robotics and Automation*, Washington, DC, May 2002.

[4] J. Barnes, C. Rizos, J. Wang, D. Small, G. Voigt, and N. Gambale. Locata: the positioning technology of the future? In *6th International Symposium on Satellite Navigation Technology Including Mobile Positioning Location Services*, 22-25 July 2003.

[5] A. Cotroni, F. Di Felice, GP Romano, and M. Elefante. Investigation of the near wake of a propeller using particle image velocimetry. *Experiments in fluids*, 29:227–236, 2000.

[6] http://www.ctgermany.com/en/press/news/2007/NOVA+-+the+3D+LED+Innovation Creative Technology Germany. Nova - the 3d led innovation, 2007.

[7] Daniel R. Dale. Automated ground maintenance and health management for autonomous unmanned aerial vehicles. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge MA, June 2007.

[8] G. Hoffmann, D.G. Rajnarayan, S.L. Waslander, D. Dostal, J.S. Jang, and C.J. Tomlin. The stanford testbed of autonomous rotorcraft for multi agent control (starmac). In *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, volume 2, pages 12–E. IEEE, 2004.

[9] Gabriel M. Hoffmann, Haomiao Huang, Steven L. Waslander, and Claire J. Tomlin. Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment. In *AIAA Guidance, Navigation and Control Conference*, 20-23 August 2007.

[10] Haomiao Huang, G.M. Hoffmann, S.L. Waslander, and C.J. Tomlin. Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3277–3282, 12-17 May 2009.

[11] Digi International Inc. Xbee/xbee-pro zb rf module support documentation, http://ftp1.digi.com/support/documentation/90000976$_C$*.pdf*.

[12] Matthew M. Jeffrey. Closed-loop Control of Spacecraft Formations with Applications on SPHERES. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge MA, June 2008.

[13] Engaget Joshua Topolsky. The bmw museum's kinetic sculpture takes your brain to another dimension, 2008.

[14] E. King, Y. Kuwata, and J. P. How. Experimental demonstration of coordinated control for multi-vehicle teams. *International Journal of Systems Science*, 37(6):385–398, May 2006.

[15] E. Klavins, C. Matlack, J. Palm, A. Nelson, and A. Bradford. Quad-rotor uav project. 2010.

[16] I. Kroo, F. Prinz, M. Shantz, P. Kunz, G. Fay, S. Cheng, T. Fabian, and C. Partridge. The mesicopter: A miniature rotorcraft concept phase ii interim report, 2000.

[17] Daewon Lee, H. Jin Kim, and Shankar Sastry. Feedback linearization vs. adaptive sliding mode control for a quadrotor helicopter. *International Journal of Control, Automation and Systems*, 7:419–428, 2009. 10.1007/s12555-009-0311-8.

[18] G.W. Leese. Helicopter downwash data. Technical report, DTIC Document, 1974.

[19] Bernard Michini. Modeling and adaptive control of indoor unmanned aerial vehicles. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge MA, September 2009.

[20] Natrualpoint. Optitrack, http://www.naturalpoint.com/optitrack/.

[21] Eric A. Olsen. *GPS Sensing for Formation Flying Vehicles*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Cambridge MA, November 1999.

[22] Parrot. Parrot ar drone, http://ardrone.parrot.com/parrot-ar-drone/usa/.

[23] Nicholas Pohlman. Estimation and Control of a Multi-Vehicle Testbed Using GPS Doppler Sensing. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge MA, August 2002.

[24] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishna. The Cricket Location-Support System. In *6th ACM International Conference on Mobile Computing and Networking*, August 2000.

[25] O.A. Rawashdeh, H.C. Yang, R.D. AbouSleiman, and B.H. Sababha. Microraptor: A low-cost autonomous quadrotor system. In *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2009.

[26] Alvar Saenz-Otero. The SPHERES Satellite Formation Flight Testbed: Design and Initial Control. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge MA, August 2000.

[27] Frantisek M. Sobolic. Agile flight control techniques for a fixed-wing aircraft. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge MA, June 2009.

[28] A. Stella, G. Guj, and F. Di Felice. Propeller wake flowfield analysis by means of ldv phase sampling techniques. *Experiments in fluids*, 28(1):1–10, 2000.

[29] Ascending Technologies. Ascending technologies hummingbird researchpilot, http://www.asctec.de/asctec-hummingbird-researchpilot-5/.

[30] ThanhTran. Building a palm size quad-copter introducing a new simple flight controller, http://www.rcgroups.com/forums/showthread.php?t=1335765.

[31] ThingM. Blinkm, the smart led, http://thingm.com/products/blinkm.

[32] G. P. Tournier, M. Valenti, J. P. How, and E. Feron. Estimation and control of a quadrotor vehicle using monocular vision and moire patterns. In *AIAA Guidance, Navigation, and Control Conference (GNC)*, pages 21–24, Keystone, CO, August 2006 (AIAA-2006-6711).

[33] M. Valenti, B. Bethke, D. Dale, A. Frank, J. McGrew, S. Ahrens, J. P. How, and J. Vian. The MIT Indoor Multi-Vehicle Flight Testbed. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2758–2759, 10-14 April 2007.

[34] Vicon. Vicon motion capture systems, http://www.vicon.com/products.