SCHOOL OF ENGINEERING, AUT UNIVERSITY

Development of Autonomous Quadrotor Micro Aerial Vehicles

Daniel Reader

May 2013

Thesis Supervisor: Dr. Loulin Huang

A thesis submitted to Auckland University of Technology in partial fulfilment of the requirements for the degree of Master of Engineering (ME)

Table of Contents

List of Figures

List of Tables

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

[illegible] (Signed)

12 AUGUST 2013 (Date)

Acknowledgements

First thanks go to Dr. Loulin Huang who took a chance on an unknown Canadian ASIC designer with no background in robotics. You have been a mentor, an encourager, an advocate, and a friend. It has been a pleasure and an honour to work with you.

Thanks also to my beautiful wife, Sheri, and my children, Annalise and Everett, who came to the other side of the world with me and who provide me daily with love, hope, laughter, support, and encouragement. My home will be wherever we are together and I hope daily to be a little more worthy of all that I have been given with you in my life.

Little of what I have achieved would have been possible without the excellent work of Jian Huang who designed and assembled my circuit boards; thank you. You are a superlative engineer and your assistance, advice, and friendship over the course of my study has been invaluable. Thanks also to the rest of the electrical engineering technician team (Achala, Steve, Dave C., Jeremy, Daniel, Brett, David T.); you have all been exceedingly helpful and a real pleasure to work with.

Many thanks to Ross Jamieson for manufacturing custom test hardware for me and for converting my 3D model into the reality of the Jumbo QBot airframe. Thank you as well to the others on the mechanical engineering technician team that have lent assistance and advice along the way (Jim, Tim, and Sam).

Thanks to Josh and Vathnack who gave me a grounding in this work and who helped orient me at AUT. Additional thanks to Jeffrey who designed and built my first two airframes.

Thank you to AUT engineering management and administration (especially Kath) for providing assistance, space and resources.

Finally, all thanks to my creator who gave me the mind and the heart to do this work.

Abstract

While much has been written about quadrotor MAV (micro aerial vehicle) theory and operation, there is relatively little that communicates the practical steps necessary to build, program, and control these complex electromechanical systems. Research abounds that deals with the minutia of control and modelling, but the question of minimal sufficiency is left unanswered and unproven on real hardware. This thesis demonstrates that PID control and operational models that are simple in form but relevant to real time operations are all that is necessary to achieve stable autonomous flight. Furthermore, it presents a clear and practical approach to the development of real flying robots.

This work describes the construction of three functional MAVs that cover a range of size, complexity, and functionality. Details of airframe structure, component selection, firmware and software development, and tuning and testing are all included and they provide a reproducible framework for further research.

The culmination of this effort takes the form of a real physical (not simulation), fully autonomous quadrotor MAV of non-trivial mass (greater than 2kg), payload capacity (theoretically greater than 2kg), and computing power (running Linux on a processor capable of up to 1400 Dhrystone MIPS). Its successful operation is presented and serves to demonstrate the efficacy of the proposed straightforward, minimalist approach to design and development.

List of Abbreviations

ADC – Analog to Digital Converter

AHRS – Attitude and Heading Reference System

AUT – Auckland University of Technology

AVR – A family of microprocessors manufactured by Atmel®

BLDC – Brushless Direct Current (Motor)

CAD – Computer Aided Design

COM – Computer on Module

DDR – Double Data Rate

ESC – Electronic Speed Controller

GUI – Graphical User Interface

I2C – Inter-Integrated Circuit (Serial communication bus)

IMU – Inertial Measurement Unit

KB – Kilobyte

LiPo – Lithium Polymer (Battery)

LLH – Latitude, Longitude, and Height

LM345 – LORD Microstrain® 3DM®-GX3-45

MAV – Micro Aerial Vehicle

MB – Megabyte

OS – Operating System

PID – Proportional Integral Derivative (Control)

PWM – Pulse Width Modulation

RPM – Rotations per Minute

RS232 – Recommended Standard 232 (A serial communication standard)

GPS – Global Positioning System

SPI – Serial Peripheral Interface (Communication bus)

SDRAM – Synchronous Dynamic Random Access Memory

SRAM – Synchronous Random Access Memory

UAV – Unmanned Aerial Vehicle

USB – Universal Serial Bus

UART – Universal Asynchronous Receiver Transmitter

USART – Universal Synchronous Asynchronous Receiver Transmitter

Z-N Tuning – Ziegler-Nichols ultimate cycle tuning method

1. Introduction

1.1. Motivation

While much has been written about quadrotor theory and operation, there is relatively little that communicates the practical steps necessary to build, program, and control these complex electromechanical systems. The goal of this research has been to employ uncomplicated but effective mathematical models and basic control systems in the development of real physical (not simulation) autonomous quadrotor flying machines of non-trivial mass, payload capacity, and computing power. The corresponding output of this research was then to demonstrate a complete and practical approach to building and testing fully operational quadrotor robots, from initial system modelling through to fully autonomous flight.

It is understood that, in many ways, this work may appear to cover ground already well developed by researchers and private developers. Much of that work, however, is proprietary and what has been published tends to deal with complicated minutia of operation and possible improvements to it. It is proposed here that research sometimes must stand upon the collected body of knowledge, distil and reduce it to essential and sufficient components, and then present a practical, reproducible application of that knowledge to create a reliable platform for further study. That defines the core of this work and its significance to the research community.

In addition to that, the reality is that quadrotor MAV development is often seen as an end to itself; all of the capability and processing of a developed robot is tied up in its operation. The firmware is dedicated to the task of flying and processing avionic requirements and it is rarely considered to be a platform for developing further capability (e.g. image processing). In cases where it is, the development infrastructure is limited and, of necessity, an adjunct to the main task of flying. This research has instead pursued a goal of MAV operation under Linux. By doing so, the entire capability and development infrastructure of a well-known operating system is made available for further research. Programs can be developed to run on the MAV entirely independent of any other firmware, including the flight tasks. The multi-threaded nature of Linux and the power of an embedded computer (not just a microcontroller) define the uniqueness and the potential of what this research seeks to demonstrate.

1.2. Approach

In developing any complex system from first principles through to completion, there are many paths that could be taken, some longer and some shorter, some involving more work and some less. At the outset, it is difficult to know what challenges will need to be overcome, especially when the area of study is relatively novel to the researcher, as was the case for this study. With these considerations in mind, the approach taken was an evolutionary one; starting with a light simple airframe and processor on a testing stand and culminating in a relatively large and heavy airframe with a complex processor (and operating system) in free flight. In this way, degrees of success could be achieved and evaluated on a given system before tackling the greater challenges of the next. This has almost certainly resulted in more total work but is believed to have decreased the amount of wasted work that could have been spent wrestling with larger leaps of difficulty.

1.3. Document Structure

A review of some relevant pieces of current literature is presented immediately following this introductory section. Subsequent to that is a discussion of the theoretical models and control systems employed in the development of the flying robots that were built. That section and the ones following it are generally organized to align with the evolutionary approach taken; dependence (where one aspect of the system requires development of another), complexity/difficulty, and chronology (in that order but loosely) have been employed to determine the order. Following theory, then, is a chapter on the selection and implementation of the hardware systems developed (including component descriptions and physical parameters). Next is a presentation of the significant parts of the firmware (code running on the embedded processors) and base station software (application code running on a PC). A chapter on testing and tuning comes next and it essentially describes the whole of the achieved results for all of the systems involved. To wrap up, there is a chapter discussing the presented results which is followed by a final chapter that renders the conclusion and discusses the implications. References and an appendix (containing links to demonstration movies) complete the document.

2. Literature review

2.1. Media

Over the last decade, UAVs have increasingly captured media attention and have taken a place of interest for the general population. Most people are aware of the presence of these machines in our modern world but they are generally regarded with an attitude of concern and suspicion; their usefulness is in military applications of surveillance, spying, and destruction. If they have any place in civilian life, it is a place of hobbyists and fringe technical enthusiasts whose passion may be interesting, but not of real practical use. Around the world, however, media providers and journalists have begun to report on a broader base of usefulness and the world is starting to recognize the much more significant role these devices are likely to have in future society.

As early as 2006, the Asian Institute of Technology in Thailand captured a headline with their flying robot that was intended for agricultural purposes. The bold title of the article stated “Flying robot helps farmers avoid dangerous chemicals” (Sutharoj, 2006). That captures the sentiment behind the mechanization of many human tasks; personal health and safety can be a strong motivator for advancing robotic capability and increasing machine deployment.

In 2009, a quadrotor MAV became a news story because of its role in dealing with the forest fires that were raging in northern Greece at that time (European Commission, 2009). Such a story further highlights the key aspects of what makes multi-rotor MAVs so useful: they can be deployed quickly and easily in any situation, they keep people out of harm’s way when the aerial situation is dangerous, they can fly when and where larger aircraft cannot, and they are inexpensive to operate.

The areas of application, then, are nearly boundless and it is not only the military that is looking for ways to use them in solving common problems. Police forces are increasingly taking an interest in reducing their reliance on expensive conventional helicopters in favour of micro alternatives. One regional police force in Canada has integrated drone use into their operations and used a quadrotor MAV last year to find $744000 (CAD) worth of illegal marijuana growing in a farmer’s field (Rabyniuk, 2013).

There is also a growing fervour of capitalism that goes along with the increasing awareness of applications. The market potential is part of the reason for the growing media interest and for the proliferation of companies pursuing this technology. A quote printed by the National Geographic magazine recently stated "… the civilian market for drones – and especially small, low-cost, tactical drones – could soon dwarf military sales, which in 2011 totalled more than three billion dollars." (Horgan, 2013, pp. 125-128) This is further emphasized by a report released last year by an American aerospace and defence analyst group that generated this headline: "Worldwide UAV market to reach more than $94 billion in ten years" ("UAVs", 2012). The report further stated that "UAVs have been the most dynamic growth sector of the world aerospace industry this decade" ("UAVs") which again speaks to the relevance and importance of this technology in the changing world.

2.2. Academic Works

A number of theses and articles have been written about the dynamic control of helicopter robots and these have generally been demonstrated with software models and/or small prototypes. A good representative thesis detailing the theory and mathematics involved with a quadrotor robot's construction and its control was conducted by J. M. B. Domingues in Portugal (Domingues, 2009). Domingues' work relied on accelerometer and compass input (magnetometer) for determining robotic state and generating corresponding control. He concluded that, even with Kalman filtering, signal noise from the motors affected the accelerometers to an unacceptable degree and that gyroscopes should be employed to more accurately determine the robot's state (Domingues, 2009, pp. 75-76). For this reason, all of the IMUs involved with this research perform sensor fusion between gyroscopes, accelerometers, and magnetometers.

Domingues' thesis relied on linearized equations of motion and simple control processing; for improvement, he suggested changing the control method to proportional derivative (PD) (Domingues, 2009, p. 76). In fact, PD (and proportional integral derivative (PID)) control of a quadrotor robot has been explored by Katie Miller at Berkeley (Miller, 2008). Her conclusion was that linearization of the motion equations and control laws was adequate under "perfect" conditions but did not perform well in the presence of uncertainty (Miller, 2008, pp. 12-13). Miller's suggestion was to explore a more accurate non-linear representation of the system while still employing linearized control. While that is certainly a worthwhile aim, this study has focused on linearized equations and PID control, and attempts to show their sufficiency even in non-perfect (e.g. outdoor) conditions, at least on a MAV of non-trivial mass.

A decision to focus on PID control is somewhat contrary to the prevailing theme of research. Many papers have been written on theoretical improvements over PID and there is undoubtedly a broad range of potentially better approaches. For example, a paper was published by researchers at the Swiss Federal Institute of Technology in 2004 in which they attempted to demonstrate that linear quadratic (LQ) control was superior to PID for quadrotor applications (Bouabdallah, Noth, & Siegwart, 2004). In the end, the researchers were not able to perform a free flight with LQ control and stated: "Contrarily, using the classical approach (PID), the autonomous flight was a success." (Bouabdallah, Noth, & Siegwart, p. 6) They maintained, however, that the modern LQ technique was optimal and "should give better results" (Bouabdallah, Noth, & Siegwart, p. 6).

More recently, another group of researchers tried to enhance basic PID control by implementing fuzzy-logic based auto-tuning of the gain parameters (Sangyam, Laohapiengsak, Chonghcharoen, & Nilkhamhang, 2010). The approach they advocated was never tested on a real system and all of their analysis relied on simulation (it is worth noting that a quadrotor MAV in flight is extremely difficult to model completely; the paper doesn't discuss the nature or quality of the simulation). Nevertheless, they did determine that fuzzy auto-tuning can yield an improvement over static PID in the case of changing system parameters and an example they used is a sudden increase of payload by 7kg (from a starting point of 0.5 kg). That should be considered an extremely improbable scenario as there are very few (possibly zero for purely academic use) quadrotor UAVs that could support a 7kg payload at all, much less one that dynamically changes by that amount. Regardless, the following statement is made of both conventional PID and fuzzy based auto-tuning PID: "both control methodologies are capable of handling external disturbance force." (Sangyam, Laohapiengsak, Chonghcharoen, & Nilkhamhang, 2010, p. 531)

As there is a trend in seeking improvement over PID, there is a corresponding trend in demonstrating improvement by simulation.   Dierks and Jagannathan (2010) published a comprehensive article on performing quadrotor control using neural networks. Their paper is thorough and instructive but the control description alone is extremely complex and the only results obtained were simulated. Those results do show an improvement but this research aims to maintain simplicity of the control algorithms and to demonstrate that simplicity as viable for a real, non-simulated system.

Of course, not all papers written on quadrotor control theory focus on deficiencies of the control algorithm. Salih, Moghavvemi, Mohammed, and Gaeid (2010) published a paper in which they presume sufficiency of PID and set about to describe a system of quadrotor control around it. Once again, their results were limited to simulations based around simplifying assumptions. In many ways, however, their paper sets out a framework of equations and approach similar to that utilized for this research. Where they stopped at simulation, this study validates by practical application.

Beyond the control system, there is also published research tackling the modeling issues for quadrotor aerial vehicles (some of which were flagged by the Miller paper referenced earlier). The majority of the system nonlinearities are small enough in relative terms to be consistently removed by the general body of research (e.g. rotor gyroscopic effect). There are cases, however, where researchers have attempted to analyze the significant non-linear contributors. Sanca, Alsina, and Cerqueira (2008), for example, examined saturation and deadzone nonlinearities as well as aerodynamics and moments in axial flight. As others have done, they performed all of their analysis through simulation and found the effects of these issues to be measurable. A study of those results suggests, however, that while the added complexity is probably needed for better simulation, it is not really worthwhile for a sensor-equipped robot. This is because the deviations they found in simulation were small enough that they won’t significantly impact real flight on a robot able to detect accumulated error. For example, they present a graph that shows accumulated error in modeled height during ascent to be around 40 meters over 300 m total (the linear model suggests 300m while the more accurate model reflects around 260 m) (Sanca, Alsina, & Cerqueira, 2008, p. 148); the altitude sensors on a real flying robot have much better accuracy than that.

Having determined that simulation results will not be the primary pursuit of this research, it is worth noting that practical endeavor has its own pitfalls. For example, a relevant thesis in this area was undertaken by M.D. Schmidt at the University of Kentucky in 2011 (Schmidt, 2011). Schmidt contended that other quadrotor UAV research projects did not utilize the broad systems approach to design and implementation whereas his did with the expressed goals of robustness and ease of control. That is similar to the goals pursued by this research, but ultimately Schmidt's success was limited by component failures and a lack of resources. These are critical concerns for MAV development as these machines are complex electro-mechanical systems comprised of costly and relatively fragile parts. Every care in the course of this research has been taken to avoid damage and manage provided resources.

The goal of non-trivial mass and non-trivial payload capacity represents yet another departure from mainstream pursuit. Pounds, Mahoney, and Corke (2010) state that most quadrotor robots used for research are limited to a few hundred grams of payload capacity and then they go on to say:

"In the commercial sphere, several groups announced plans to market 4-6 kg devices, but these did not manifest in products, whereas numerous examples of sub-2 kg craft are now readily available. The rarity of quadrotor UAVs larger than 3 kg can be attributed to the numerous design challenges encountered as the weight of the vehicle increases, and to the attendant engineering rigour that must be exercised to safeguard proportionally more fragile hardware." (Pounds, Mahoney, & Corke, p. 692)

Their paper ultimately presents the successful flight of their 4kg airframe to a height of 2 m. It also maintained stable hover for 10 seconds without pilot correction (Pounds, Mahoney, & Corke, p. 22).

Although this research doesn't pursue a MAV exceeding 3kg, it does seek to present the successful development of a quadrotor robot with a mass well over 2kg and having a total weight plus payload capacity theoretically greater than 4kg. In addition, its flight goals are to exceed 9 meters of altitude and maintain consistent hover attitude at all times.

3. Modelling and Control Methods

A quadrotor MAV is a highly non-linear system that is subject to significant external sources of disturbance and influence. It is inherently under-actuated, having six degrees of freedom (three of orientation and three of position), but only four actuators (the motors). It is made up of mechanical, electrical, and software components that must continuously and reliably interact over tiny amounts of real time to keep the machine in the air and under control. It does this despite having been comprised of parts that exhibit varying (and variable) delays, parts that cause interference (mechanical, electrical, electromagnetic), other parts that respond negatively to the introduced interference, parts that bend (but hopefully never break), pieces that are unbalanced, and bits that are unaccounted for. The full characterisation of such a system would involve an immense amount of analysis, modelling, and computation. It is possible, however, to reduce the state space of the problem of quadrotor MAV flight dramatically by making simplifying assumptions that reduce the perceived complexity while still accurately reflecting the greatest part of the dynamics and interactions involved. To be clear, this research has endeavoured to achieve autonomous stable controlled flight using a set of modelling equations and control approaches that are beautiful not only for their simplicity but also for their efficacy.

3.1. Quadrotor Model

3.1.1. Coordinate Frames and Variables

The most basic quadrotor design involves four motors that are placed at each end of either a real or virtual arm arranged in the shape of a plus sign. Each motor is equidistant from the centre and from the two motors nearest it. In this way, motor symmetry is achieved and then the goal is to achieve similar symmetry with the rest of the components, arranging and distributing them to ensure balance and maintain the centre of gravity at the effective centre of the cross structure. A simple representation of the model is shown in Figure 1.

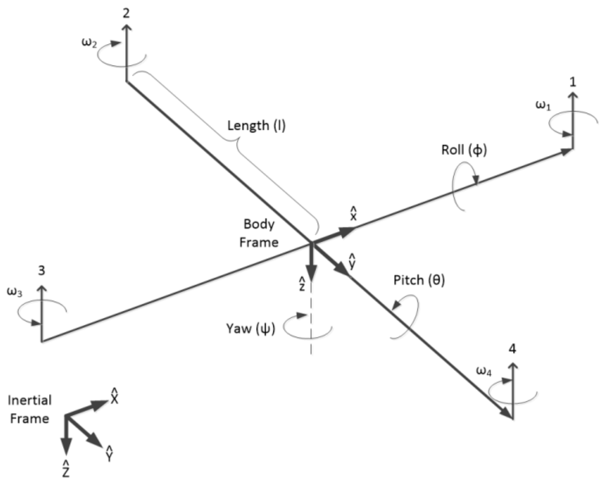


Figure 1 - Mathematical Model of Basic Quadrotor

Having established motor position at the ends of the cross frame, it is important to identify coordinate systems. In the first place, there is the body frame which is coincident with the airframe itself, has its origin at the centre (defined as the midpoint between the motors and on the same plane as them), and moves with the airframe as it changes position. Because of that relative nature of the body frame (from within it, the airframe is not seen to move; only its orientation changes), it is necessary to further define a reference or inertial frame that is fixed in space and from which the movement of the airframe can be observed and described.

All of the quadrotor variables of position and orientation can now be defined within those coordinate frames. The body frame is defined such that, for an all zero orientation, the x-axis is aligned with one arm and has a positive direction toward motor 1 (the arbitrarily chosen front or forward direction of the aircraft). The other motors are defined sequentially in a counter-clockwise direction around the airframe such that motor 2 is on the left (port) side, motor 3 is aft and motor 4 is on the right (starboard). The y-axis is then defined as having a positive value in the starboard direction while the z-axis is positive downward (or opposite the direction of motor thrust). The unit vectors for these axes are represented by $\hat{x}$, $\hat{y}$, and $\hat{z}$, respectively.

The inertial frame is then defined to align with the position of the Earth either at or near the point of launch. The X-axis is aligned with North/South, is positive in the direction of North, and has a unit vector represented by $\hat{X}$. The Y-axis runs East/West, points positively toward the East, and has a unit vector represented by $\hat{Y}$. Finally, the Z-axis is aligned with Up/Down as determined by gravity such that down is the direction of the force of gravity exerted by the Earth upon the aircraft (and any observer(s) at or near the point of origin). The unit vector for the Z-axis is represented by $\hat{Z}$ and is positive in the downward direction (i.e. gravity exerts force in the positive direction while motor lift thrust is exerted in the negative direction).

Orientation of the aircraft can then be defined in terms of a triplet of angles that represent its rotation around the axes of the body frame. These angles are known as Euler angles and, for an aircraft, are called roll, pitch, and yaw. The roll angle defines the airframe rotation about the $\hat{x}$ axis and, by the well-known right hand rule, is positive when the airframe is tilted to the starboard side and negative when it is tilted to port. The pitch angle defines rotation about the $\hat{y}$ axis and is positive when the aircraft is tilted aft (nose up) and negative when tilted forward (nose down). Yaw, then, is the rotation about the $\hat{z}$ axis and is positive when the aircraft is turned clockwise. These three Euler angles, roll, pitch, and yaw, are represented by the Greek symbols phi (ϕ), theta(θ), and psi (ψ) respectively and are sufficient to define the orientation of the quadrotor aircraft in all situations of concern for this research (for a quick description of the common issue with Euler angles known as Gimbal Lock, see Section 3.1.2).

The other significant variables of a quadrotor design are then the length of the arms, the mass of the airframe, and the rotation of the motors. Arm length (variable l) affects the relationship between thrust of the motors and the resultant torque on the airframe (in addition to size and mass, of course). The motor rotation naturally is the same as rotor rotation and therefore is the contributing variable to motor thrust. Each motor's rotation is described independently by the variables ω_1, ω_2, ω_3, and ω_4 where the subscripts define the associated motor. The mass of the airframe is denoted m and is directly related to the downward gravitational force on the airframe and the moment of inertia seen about the axes of rotation.

3.1.2. Gimbal Lock

Euler angles were chosen for orientation description because of their simplicity and their direct correlation to human intuition. When an airframe is observed in flight, it is fairly easy to understand its orientation in terms of its roll (tilted left or right), its pitch (nose up or down), and its yaw (heading). The fact that these natural observations can also be directly expressed in simple, clean mathematical terms as discovered and proven by Euler lends further motivation to their use. There is, however, a well-known problem with using Euler angles for representation of 3-dimensional orientation and it is known as gimbal lock. This issue will not be exhaustively described here, but at a high-level, there is an ambiguity (or singularity) of actual orientation that occurs when one angle in an Euler sequence is brought into alignment with another angle (i.e. when the former angle is +/- 90 degrees). The singularly exists for all systems that employ Euler angles but its manifestation changes with the order in which the rotational angles are applied. For the purposes of this research, yaw is always last in sequence and it is the only angle allowed to approach 90 degrees. Because this research focused on non-acrobatic flight, the airframe attitude was constrained to limit roll and pitch angles well away from 90 degrees.

It would, of course, have been possible to employ an alternative form of orientation representation that did not exhibit the Euler angle limitation. The most common alternative is quaternion representation that employs four values to describe the attitude of a body in space. In so doing, the quaternion approach eliminates the gimbal lock problem but it introduces a greater complexity of description (and mathematics) that is not easily conceptualized or intuited.

3.1.3. Body Torque and Thrust

Given the variables defined above, it is now possible to define the equations that will directly affect the orientation of the airframe and its movement in space. Within the overarching goal of simplicity and effectiveness sought for this research, rotation about the 3 airframe axes has been assumed to be decoupled. For example, only the torque induced by motors 1 and 3 on the forward and aft arms is considered to affect pitch. In the same way, only motors 2 and 4 contribute to roll torque. Torque about the $\hat{z}$ axis (yaw torque) is resultant from the rotation of all four motors but is considered to be entirely independent of the instantaneous roll and pitch orientation and torques. With these assumptions in place, then, the following equations define roll, pitch, and yaw torques experienced by the airframe:

$$\tau_{\hat{x}} = lb(\omega_2^2 - \omega_4^2) = I_{\hat{x}}\ddot{\phi} \tag{1}$$

$$\tau_{\hat{y}} = lb(\omega_1^2 - \omega_3^2) = I_{\hat{y}}\ddot{\theta} \tag{2}$$

$$\tau_{\hat{z}} = d(\omega_2^2 + \omega_4^2 - \omega_1^2 - \omega_3^2) = I_{\hat{z}}\ddot{\psi} \tag{3}$$

In these equations, τ is torque and the subscripts identify the affected axes; l is the length of the arms; b and d are the propeller thrust and drag constants, respectively; ω represents the rotational velocity of the motors and the subscripts identify the motor number; I is the moment of inertia and has subscripts for correlation to each axis; finally, ϕ, θ, and ψ represent roll, pitch, and yaw, respectively, and the superscript double dots are understood to identify the second derivative (angular acceleration in this case).

In these equations we see that further simplifying assumptions have been applied. In the first place, the effective torque on the airframe is related to motor speed through the use of constants that are considered valid for all angular rates of the propellers (it should be noted that the propellers cannot spin in reverse). This is not strictly the case and it has been shown that the static thrust (and drag) of a hovering aircraft does exhibit some variation over the range of motor speed (Brandt & Selig, 2011). In a given range, however, it is reasonably accurate to assume a linear relationship between the square of the propeller angular rates and the resultant force and torque induced in the airframe. Since a non-acrobatic quadrotor MAV generally operates within a narrow range centred on hover thrust, this assumption is adequate. The term ‘static thrust’ is also significant as it implies that the airframe velocity (and a corresponding aerodynamic value known as the advance ratio) is zero. It should be apparent that the airframe is intended to move, but its peak velocity goal is fairly small and the assumption of static thrust is representative of its state in almost all cases.

The thrust constant then provides the relationship between the square of the motor angular rate and the thrust force exerted on the airframe. The basic equation for this constant is shown in (4) where T is the thrust force.

$$b = \frac{T}{\omega^2} \tag{4}$$

Specifying the thrust constant this way is directly applicable to the problem here, but it is actually a reduction of the better known thrust coefficient (C_T) that is used in other literature. The thrust coefficient is a constant that also relates squared angular velocity to effective thrust, but it is defined independent of air density (ρ) and propeller radius (r). At any given time, propeller radius is a constant and its inclusion in b is obviously acceptable. Air density changes with temperature and pressure and is, therefore, clearly not constant, but the variation is considered small enough to be ignored. The relationship between b and C_T can be derived from formulas given by Domingues (2009, p. 15) as shown in (5).

$$b = \frac{T}{w^2} = C_T \frac{4\rho r^4}{\pi^2} \tag{5}$$

If a standard temperature (20 degrees C) and pressure (101325 Pa) are assumed, the air density (ρ) is equal to 1.204 kg per cubic meter (Moaveni, 2011, p. 631) and b can then be calculated directly from the thrust coefficient and propeller radius.

The drag constant, d, used here defines a linear ratio between motor torque and the square of angular velocity. This is again a reduction from a better known propeller constant: the power coefficient (C_P). Because propeller power (P_P) is related to torque (τ) as shown in equation (6), the relationship between d and C_P can again be derived from a corresponding Domingues equation (2009, p. 15), as shown in (7).

$$P_P = \tau\omega \tag{6}$$

$$d = \frac{\tau}{\omega^2} = \frac{P_P}{\omega^3} = C_P \frac{4\rho r^5}{\pi^3} \tag{7}$$

The final equality in (1), (2), and (3) starts with the moment of inertia and correlates the torques on the airframe to the resultant acceleration about the three body axes (i.e. angular acceleration of roll, pitch, and yaw). This is another simplification; it doesn't take into account the gyroscopic moment of the spinning propellers or the angular dynamic force of the body in motion. The impact of these effects on the rotation of the airframe is considered to be small enough relative to the direct contribution of propeller thrust that it is reasonable to omit them from the equation.

Using the same thrust constant presented above, the equation for total lift thrust (or, more accurately, the total thrust exerted on the airframe perpendicular to the plane of the body) generated by the rotors is shown below in equation (8).

$$F_{\hat{z}} = b(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \tag{8}$$

This total thrust then, combined with the effect of gravity (and wind, when present), determines the acceleration and corresponding displacement of the airframe in space.

For static hover above the ground, when the airframe is parallel to the ground and neither rising nor falling, the total thrust required will equal gravity, as per (9).

$$F_{\hat{z}} = m \cdot g \tag{9}$$

3.2. Proportional Integral Derivative (PID) Control

Having defined assumptions and arrived at basic equations for the forces acting upon the airframe, it is now possible to approach control of the aircraft. For many years (since the 1930s in its current form (Åström & Hägglund, 2006, p. v)), the proportional, integral, derivative (PID) approach to control has proven effective across systems of almost every imaginable type and application. From systems that are small and agile to those immense and ponderous, a PID control system can almost always provide a perfectly satisfactory (and sometimes effectively optimal) degree of management. Furthermore, a PID approach can often provide an excellent solution for systems that lack full characterization, whether that is by intention or by the practical constraints of time and equipment. In this case, it is fair to say that this research focused on an object of control (quadrotor MAV) that is not fully characterized and, indeed, that would be extremely difficult to characterize fully. There is also, once again, elements of simplicity and correlation to natural intuition that come along with PID control that add to its desirability of application in this case.

The intuitive nature of PID control can be seen from a simple description of its behaviour across the three components found in its name, along with the illustrative situation of a car's acceleration control (the accelerator or gas pedal). The proportional control response is, of course, directly related to the deviation between the desired value and the current state. If the difference to the desired value is large, the corresponding response is large, if small, the response is small. In the car illustration, this means the accelerator will be pushed harder when an increase of 30km/h is desired versus when an increase of only 5km/h is needed.

The integral term represents an accumulation of control response over time. If the system exhibits a resistance to achieving a desired setting exactly, the integral term will accumulate the offset over time and exert a stronger and stronger compensation response that corresponds to the delay in reaching the setpoint. In the scenario of controlling a car's velocity, now consider that the car in question encounters a hill to climb; as it turns upward, the car will decelerate and the proportional component will begin to push harder on the accelerator. In all likelihood, an equilibrium state will be achieved somewhere less than the desired value. A given amount of pressure to achieve 50 km/h on a level road may only achieve 40km/h on an uphill. In that case, the operator of the vehicle will perceive the deviation from the setpoint over time and will increase the displacement of the accelerometer by a growing amount until the setpoint is reached once again.

Finally, the derivative term is used as a check against unbounded increases in the control response; it exists as a damper that works against the proportional and integral terms to reduce the control strength as the system starts to move. The derivative can be accurately thought of as the velocity of the system response to the control input. The quicker the system begins to move toward its desired setting, the less the PI terms should be trying to push it. This acts as a check against the momentum gained in the path from the current state to the desired value and thereby helps to reduce the amount (or possibility of) overshoot when the setpoint is reached.

For the car accelerometer, this means that if the desired setpoint is 100km/h and it is currently only at 5, the proportional term will be significant (and the integral term will have a large value for accumulation), so the push on the accelerator will be correspondingly significant. If the car is powerful and begins to add 20km/h to its velocity every second, the momentum of acceleration will be great enough that, even though the force applied to the accelerator was decreased as it approached 100km/h (and the proportional amount reached zero at that point), the car would accelerate beyond the desired value. By employing a derivative term, however, the operator would factor in the rate of change of velocity and would stop pushing on the pedal much sooner, thereby reducing overshoot and achieving a smoother approach to the desired value.

Another way in which PID can be said to be intuitive is in the naturally understood nature of the terms with respect to time: the proportional term reflects the present difference between desired and actual, the integral term is a response to the accumulation of deviation in the past, and the derivative term attempts to anticipate the future. With all three components employed, reaching a specific desired value for the system can be achieved with alacrity and finesse.

This is how a PID system works in descriptive terms. Mathematically, the formula is well known, but takes several equivalent forms as shown in (10) and (11).

$$u(t) = K\left(e(t) + \frac{1}{T_i} \cdot \int_o^t e(\tau)d\tau + T_d \cdot \frac{de(t)}{dt}\right) \tag{10}$$

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + \frac{K_d de(t)}{dt} \tag{11}$$

The PID output variable is u(t) while the input is e(t) which is defined as the difference between the setpoint (desired) value and the current value at time t. Equation (10) utilizes a universal gain constant (K) that is applied to all terms but then modified by an inverse integral constant for the I term (T_i) and a multiplying term for the D term (T_d). Equation (11) is effectively the same but conceptually different in that it completely separates the PID gains from one another (hence K_p, K_i, and K_d).

The representation of equation (11) was chosen for this work largely because of the independence of the PID gain terms expressed therein. In terms of systems response, this is perhaps less intuitive than the alternative because it blurs the time nature of the integral and derivative factors. In any case, this is largely a matter of preference and it was deemed more desirable to decouple the factors for application herein.

A final note on PID control is this: not all terms are necessary at all times. By changing the gain factor to zero, any term may be eliminated. Although this effectively changes the definition of control to some degenerate form that is no longer PID (e.g. it becomes PI or PD), it is still nevertheless useful in a dynamic system to consider the possibility of dropping and re-introducing terms in response to changing situations. The control approaches for this research always implemented full PID but in several instances the I term had a gain of zero. The primary reason for doing so is that tuning of all three terms together can be challenging and often an intermediate approach to a solution involves only the proportional and derivative terms. If the system response was acceptable with PD-only, tuning of the I term was left for future endeavour (with the expectation that some (possibly insignificant) benefit would always be realized from adding it in at some point).

3.2.1. PID Tuning

In some cases, it was clear that all of the PID gain terms were necessary for correct operation. Specifically, the roll and pitch controls require fast achievement of the setpoint with minimal overshoot and absolute rejection of static offset (otherwise the airframe will drift). Initial attempts to manually tune the PID gains were made using small changes in individual terms and experimentally observing the response.

Ultimately, however, the effort to iteratively account for all of the system variables and nonlinearities was deemed either unlikely to achieve the desired result or to achieve it too slowly for practical purposes. What was desired was a clear and repeatable approach to tuning that could be applied to future systems. The path forward then involved a widely used approach to PID tuning called *the ultimate cycle method* that was first developed by John G. Ziegler and Nathaniel B. Nichols (Bolton, 1998, p. 238).

3.2.1.1. Modified Ziegler-Nichols Ultimate Cycle Tuning

The primary attraction of the Ziegler-Nichols ultimate cycle method (hereafter referred to as Z-N tuning) is twofold: first, it is practical and can be experimentally applied, and second, it reduces the PID tuning problem from three variables to two.

The approach to application of Z-N tuning involves first setting the PID integral and derivate terms to zero (i.e. P term only). Then the proportional term is gradually increased until oscillations of constant amplitude are observed. The proportional gain at that point is identified as the ultimate gain (K_u) while the period of oscillation is called the ultimate period (T_u). Tuned PID gain terms are then calculated as shown in equations (12), (13), and (14).

$$K_p = 0.6 \cdot K_u \tag{12}$$

$$K_i = \frac{2 \cdot K_p}{T_u} \tag{13}$$

$$K_d = \frac{K_p \cdot T_u}{8} \tag{14}$$

In theory, this is straightforward, but it quickly became apparent that this was not so simple for quadrotor tuning, especially for lightweight airframes, because almost any P-term value that had a measurable system effect seemed to cause oscillations of increasing magnitude. As the P-term value was slowly increased, the system would appear to remain stable until a disturbance occurred; then it would go increasingly unstable and it was essentially impossible to define any particular point at which constant amplitude oscillations could be said to reliably occur. This was less true of the heavier airframes and also less true when running the motors at higher nominal thrust, but a certain degree of instability (and inconsistency) always remained. Nevertheless, for all of the systems developed, some amount of correlation could be found between P-term only gain and predictable oscillation.

This, then, became the basis for a modified approach to Z-N tuning. Rather than requiring absolute values for K_u and T_u, both were approximated from experimental results. From those initial numbers, each term was then swept over a range of nearby values to determine possible refinement and improvement in system response. Altogether, this worked quite well and the results are discussed in chapter 6.

3.3. Orientation PID Control

3.3.1. Body Torque

Having established modelling equations for the quadrotor and identified our control approach, it is now possible to present the method of application to the system. The first goal of quadrotor development is attitude/orientation control. Having this desired target and intending to employ PID, it is still necessary to select the parameter that will be controlled in the system (the u(t) term in equation (11) must be linked to a physical parameter of the system). Ideally, the association will be as direct as possible between the sensed error component (e(t)) and the PID output. The error and desired terms for orientation are obviously with respect to roll, pitch, and yaw. The natural association for effecting a change in those variables is airframe torque. So then, torque about a given axis is associated with the corresponding Euler angle to give the orientation control equations seen below.

$$\tau_{x'} = K_{P_x} e_\phi + K_{I_x} \int_0^t e_\phi dt + K_{D_x} \dot{e}_\phi \tag{15}$$

$$\tau_{y'} = K_{P_y} e_\theta + K_{I_y} \int_0^t e_\theta dt + K_{D_y} \dot{e}_\theta \tag{16}$$

$$\tau_{z'} = K_{P_z} e_\psi + K_{I_z} \int_0^t e_\psi dt + K_{D_z} \dot{e}_\psi \tag{17}$$

The significance of the ′ (e.g. x′) in these equations versus the ^ (e.g. x̂) seen associated with previous torque equations is that these equations describe *desired* torque as an output of our control approach versus *actual* torque described previously. All of the outputs and the error term inputs are, of course, a function of time, but the (t) has been left off to give cleaner expression. The error terms are simply the difference, in radians, between the desired angle (e.g. 0 for hover) and the actual angle (current attitude of the airframe). The derivative error terms have the characteristic that the desired derivative is always zero (when the angle setpoint is achieved, no further angular velocity (rotation) is desired) and the error derivative terms can then be further broken down as follows:

$$\dot{e}_\phi = \dot{\phi}_d - \dot{\phi} = -\dot{\phi} \tag{18}$$

$$\dot{e}_\theta = \dot{\theta}_d - \dot{\theta} = -\dot{\theta} \tag{19}$$

$$\dot{e}_\psi = \dot{\psi}_d - \dot{\psi} = -\dot{\psi} \tag{20}$$

The outcome of this is that the derivative terms become simply the negative of the angular velocity experienced in the airframe (convenient because this comes directly out of orientation sensors).

All of this comes together such that, at any given point in time, we can calculate the desired airframe torques (from desired roll, pitch, yaw, and sensed (actual) roll, pitch, yaw, and corresponding angular velocities) and then use the desired total thrust (already identified as equal to the gravitational force as per (9) when hovering) to work out the necessary motor response.

3.3.2. Motor Response

At this point, we have the following four equations containing the four actuator variables of motor response (motor/propeller angular rate $\omega_{1\text{-}4}$) as related to the body torques and total thrust described above:

$$\omega_2^2 - \omega_4^2 = \frac{\tau_{x'}}{lb} \tag{21}$$

$$\omega_1^2 - \omega_3^2 = \frac{\tau_{y'}}{lb} \tag{22}$$

$$\omega_2^2 + \omega_4^2 - \omega_1^2 - \omega_3^2 = \frac{\tau_{z'}}{d} \tag{23}$$

$$\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2 = \frac{F_{z'}}{b} \tag{24}$$

Solving this equation set for the individual motor rotations gives:

$$\omega_1 = \sqrt{\frac{F_{z'}}{4b} + \frac{\tau_{z'}}{4d} + \frac{\tau_{y'}}{2lb}} \tag{25}$$

$$\omega_2 = \sqrt{\frac{F_{z'}}{4b} - \frac{\tau_{z'}}{4d} + \frac{\tau_{x'}}{2lb}} \tag{26}$$

$$\omega_3 = \sqrt{\frac{F_{z'}}{4b} + \frac{\tau_{z'}}{4d} - \frac{\tau_{y'}}{2lb}} \tag{27}$$

$$\omega_4 = \sqrt{\frac{F_{z'}}{4b} - \frac{\tau_{z'}}{4d} - \frac{\tau_{x'}}{2lb}} \tag{28}$$

There is an obvious pattern in these equations (unsurprisingly): each motor contributes a quarter of the total thrust, each contributes a quarter of the total yaw torque (motors 1 and 3 contribute positively while 2 and 4 are negative), and each contributes half of the torque for the axis on which it resides (roll for motors 2 (positive) and 4 (negative), pitch for 1 (positive) and 3 (negative)). Once again, this aligns well with intuition and it can be seen that if everything were ideal, having all of the motors matched in rotation would yield no airframe torques on any of roll, pitch, or yaw; the airframe would maintain its current orientation perfectly. In that situation, the total thrust would be the sum of the motor thrust and, if the airframe were parallel to the ground (perpendicular to the gravity vector), it would either hover or accelerate up or down depending on the total thrust's magnitude relative to gravity.

It may seem strange that total thrust is included as part of orientation control. It is true that it does not strictly contribute to orientation but is, rather, a part of position control as it relates to the up/down degree of freedom for the airframe. Including it here, however, is sensible, because the practical ability to control the airframe relies on the simplifying assumptions of linearity around the hover point and the overwhelming magnitude of the motor forces as compared to outside disturbances, as discussed in Section 3.1.3. For example, if the motors are off, the theoretical equations would suggest that no airframe torques should be observed, but in practical terms the outside forces would contribute significantly to changes in airframe attitude in that situation and there would be no control. For orientation control to be practically realizable, the motors must be running at a total thrust of some significance.

3.4. Position PID Control

The aircraft's ability to move through space defines the second triplet of its degrees of freedom; it can move forward or backward, left or right, and up or down. Controlling its movement up and down is relatively straightforward as the motor actuators are generally aligned with that direction (when at or near hover). The other directions present a problem, however, because there are no actuators (motors) generally aligned to move the aircraft forward/back or left/right. This, then, is the reason for quadrotor aircraft being described as under-actuated. Nevertheless, movement of the aircraft across space can be achieved simply by tilting it in the desired direction and thereby converting some of the motor thrust into a component of lateral directional thrust.

Determining a quadrotor's position in (outdoor) space is most easily defined in terms of the common navigational references of latitude, longitude, and height. These are fairly easily obtained in practice from sensors but there is some difficulty in achieving the precision required for autonomous flight. Latitude and longitude can be determined with reasonable precision from standard GPS sensors. Altitude/height is also available via GPS triangulation, but its accuracy is somewhat less. For that reason, air pressure sensors are typically employed in aircraft; they can detect the ambient atmospheric pressure with a high degree of accuracy and that can be converted into an altitude reading using standard formulas that are accepted and applied around the world.

An interesting paper that shows how to derive an accurate equation for altitude from pressure was published by the Portland State Aerospace Society (2004). In it, they quickly demonstrate the inadequacy of the isometric equation that is often applied and present a better solution and derivation of it. Ultimately, they present the quality of their derivation in light of the known gold standard equation from the CRC handbook (1996 edition) and that standard is the one applied for the purposes of this research. Equation (29) is taken directly from their paper (Portland State Aerospace Society, p. 4). The variable z is altitude in meters while P is determined pressure in pascals.

$$z == 44331.5 - 4946.62 \cdot P^{0.190263} \tag{29}$$

Whether perfectly achievable or not, the theoretical approach to outdoor position control (indoor position control was limited to height during the course of this research) presumes a high level of accuracy in the sensed positional state. At any given instant in time the quadrotor's current position in space is known. So also are its derivative terms of position which correspond to directional velocity in three directions: north, east, and down (with each term defining the positive direction of movement; south, west, and up are negative). These parameters then can be directly applied to form PID equations, but the decision as to which variable to associate with the PID output needs to be addressed first.

If the principal of most direct correlation were to be applied here as was done for body torque when dealing with orientation, the best parameter to associate would probably be directional thrust. Indeed, for height control this is simply and directly done to give a thrust offset (from the gravitational or hover thrust) out of the PID equation. For the lateral degrees of freedom, however, it was deemed simplest to directly manipulate the orientation variables (roll and pitch) and thereby achieve the translation desired.

The contribution of yaw to position control would naturally be to provide angular influence to the directional components of roll and pitch. For example, if the aircraft's nose is pointed east, then pitch affects east/west translation; if it is pointed north, pitch affects north/south translation. To limit the calculations involved, yaw is assumed to be always set to zero (causing the airframe to be pointed north) when the quadrotor is operating with autonomous position control engaged. The result is that a positive pitch angle will cause a component of thrust force to be applied in the southern direction (negative latitude). A positive roll angle will cause a resultant translational force upon the aircraft in the east direction (positive longitude). These, then, are the positional PID equations:

$$\theta_d = -K_{p_{lat}} e_{latm} - K_{i_{lat}} \int_0^t e_{latm} dt + K_{d_{lat}} v_{north} \tag{30}$$

$$\phi_d = K_{p_{long}} e_{longm} + K_{i_{long}} \int_0^t e_{longm} dt - K_{d_{long}} v_{east} \tag{31}$$

$$F_{\Delta_d} = K_{p_{alt}} e_{alt} + K_{i_{alt}} \int_0^t e_{alt} dt - K_{d_{alt}} v_{down} \tag{32}$$

The *d* subscript indicates the desired or new setpoint value; *lat* indicates latitude while *long* indicates longitude and *alt* indicates altitude; *latm* and *longm* then indicate latitude in meters and longitude in meters, respectively; finally, *north*, *east*, and *down* are subscripts for the velocity values in those directions. The error terms for latitude and longitude in meters are determined according to equations (33) and (34).

$$e_{latm} = (lat_{deg_d} - lat_{deg}) \cdot k_{lat} \tag{33}$$

$$e_{longm} = (long_{deg_d} - long_{deg}) \cdot k_{long} \tag{34}$$

For testing in Auckland, k_{lat} is approximated as 110974.88 meters per degree while k_{long} is approximately 89181.55 meters/degree (these values were determined using a web based calculator (Computer Support Group, Inc., 2011) and the approximate latitude for testing of -36.85399 degrees (which is the latitude of the building wherein most of this research was performed)).

It is not strictly correct to apply a conversion factor for latitude or longitude degrees to meters. This is due to a couple of factors: 1) degrees of latitude (slightly) and longitude (greatly) vary in corresponding distance between the equator and the poles (i.e. conversion to meters is not constant but depends on position on the earth), and 2) the surface of the earth is curved and translation between two points cannot generally be achieved in a straight line. For these reasons, a proper calculation of distance between two GPS points involves the computation of what is known as the great-circle distance and a reasonable approximation for short distances would involve the haversine formula (Veness, 2012). Because the distances involved in this study are relatively small (the aircraft, for the foreseeable future, is unlikely to travel more than a couple of kilometres) it was deemed acceptable to use a constant approximation as the relative loss of accuracy is extremely small (compared to GPS inaccuracy, for example).

The translational force that comes from a change of attitude can be easily worked out from simple trigonometry. As the angle of total thrust changes from zero (straight vertical) on either roll or pitch, the corresponding component of lateral thrust will be given by the computed tangent of the vertical thrust (which can be assumed to be constant and equal to mg, the weight of the quadrotor, when the robot is holding position). This relationship is illustrated using roll in Figure 2.

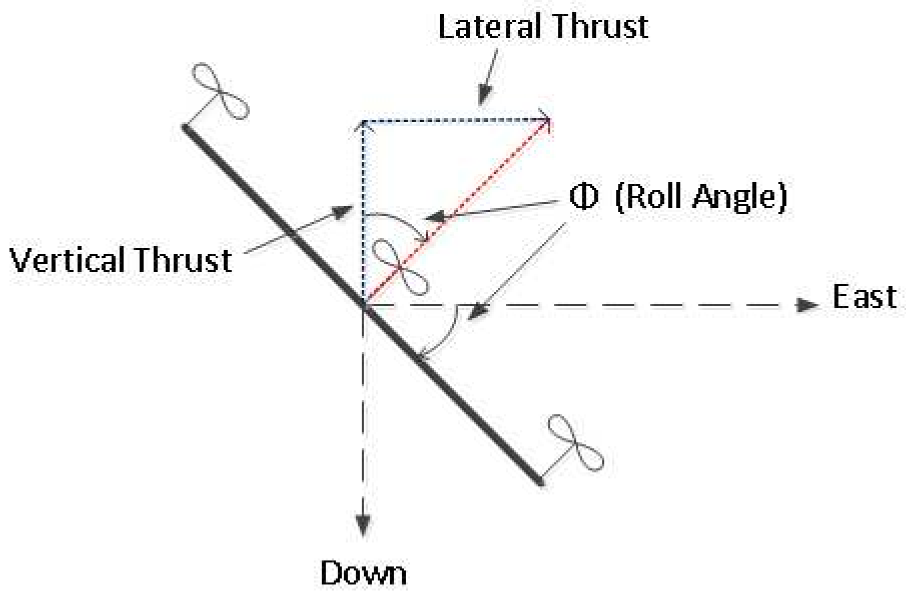


Figure 2 - Lateral Thrust Illustration

And the equation for lateral thrust with respect to the roll angle (i.e. eastward thrust) is then given in equation (35). The same approach is applied for northward thrust (equation (36)), but the pitch value is negative because a positive pitch angle (nose up) when the aircraft is facing north means that the lateral thrust will be oriented southward.

$$F_{east} = mg \cdot tan(\phi) \tag{35}$$

$$F_{north} = mg \cdot tan(-\theta) \tag{36}$$

The desired roll angle, pitch angle, and thrust offset outputs of the position PID calculations are directly applied to the orientation equations described is Section 3.3 to form a comprehensive whole for autonomous flight. In the simplest application, this is demonstrated by position hold; when the aircraft is in flight, it can be told to autonomously hold position by setting the desired latitude, longitude, and height to the currently sensed values. More complex autonomous flight patterns are achieved from the same starting point (setting the desired position to the current/starting position) and then applying increments or offsets according to a programmed pattern.

4. MAV Structure Design

MAVs can be built in all sorts of configurations and from a huge variety of materials and components. This research focused on basic quadrotor configurations with propellers parallel to the body frame and providing thrust directly perpendicular to it. The studied airframes were all of significant size (more than 0.5 meter across) and contained processing power and sufficient sensors to enable fully autonomous orientation control; one of them (Jumbo QBot) was also capable of autonomous position control.   The components of these developed airframes are described in the following sections

4.1. Mechanical

Over the course of this research, three complete MAVs were built. The first was a simple structure with no landing gear and no control board protection (it sat exposed on the top of the airframe). The next was a more complete design that was spider-like in inspiration and had landing gear and modular construction designed to properly house the control board, the battery, and other components. The final airframe was intentionally larger than the other two and was again purpose-built for the components and payload it houses and carries.

4.1.1. Basic Cross Frame (QBot1)

As this was the first airframe studied in this research, it was dubbed QBot1 (short for quadrotor robot 1). It was developed by another AUT student in the CAD software called Solidworks™ and was constructed out of a combination of parts, some of which were 3D printed (in AUT's rapid prototyping facility) and some of which were purchased. Figure 3 shows the mechanical representation of this airframe as it was designed in the CAD software.

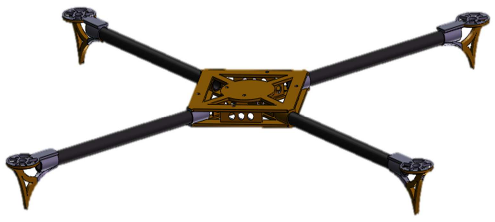


Figure 3 - QBot1 3D CAD Model

The motor mounts at the end of each arm and the central housing that ties the arms together and provides support for the control circuit board were all 3D printed. The arms are made from carbon fibre tubes (often used for building large kites) and all of the components are held together by friction and high-strength epoxy. Figure 3 depicts the motor holders as they were originally designed and they worked well for low motor speeds. When this research reached a point that the motors needed to be run at flight speed, however, it was seen that the mounts lacked enough rigidity to resist vibration. They were redesigned and replaced with reinforced mounts, as discussed in Section 6.1.2.3.

The fully assembled airframe with the final motor mounts is shown in Figure 4.

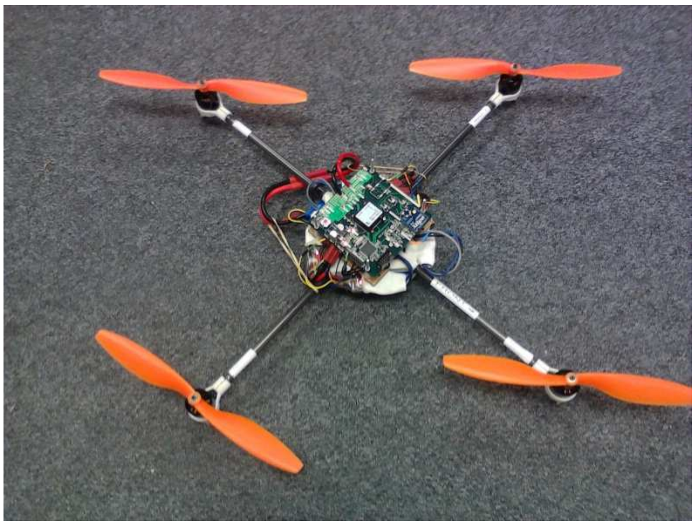


Figure 4 - Assembled QBot1

The important characteristics of this airframe are shown in Table 1.

Table 1 - Physical Parameters of QBot1 Airframe

Arm Length (l)	0.27m
Assembled mass without battery	660g

Assembled mass with battery (g)	1.01 kg

This airframe operated exclusively with plastic 10” EPP-style propellers having a 4.5” pitch. The corresponding thrust and drag constants were calculated using equations (5) and (7) with the values for C_T (0.1154) and C_P (0.0743) given for this propeller in Domingues’ (2009, p. 15) work. The calculated results are shown in Table 2.

Table 2 - Thrust and Drag Constants for Plastic 10x45 Propellers

Thrust Constant (b)	0.0000146 kg*m/rad^2
Drag Constant (d)	0.00000038 kg*m^2/rad^2

4.1.2. Spider-Inspired Airframe (Araqnobot)

The second airframe used for this research was designed by the same student who had created the QBot1 frame. The new design consisted of a much more extensive central housing that would better protect and support the control board and associated components. It was also made in a more modular fashion such that the battery holder and control board platform could be removed while leaving the rest of the airframe intact. Consideration was placed on providing proper mounting points for additional sensors underneath and at the end of each arm. In the end, the design took on a spider-like aspect and the assembled robot was given the name Araqnobot. The original CAD design is shown in Figure 5.

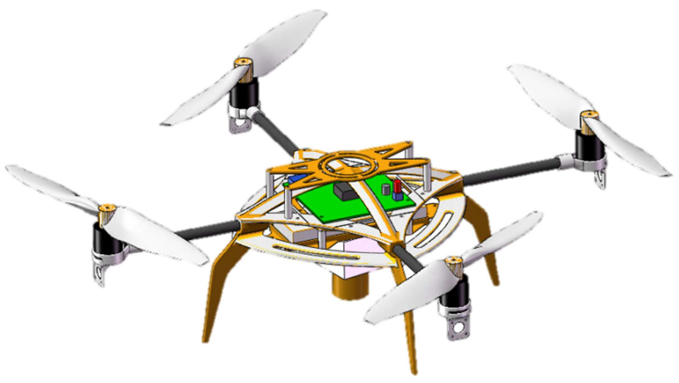


Figure 5 - Araqnobot 3D CAD Model

As with QBot1, this airframe had the central body and motor mounts manufactured on a 3D printer. The arms were again made out of tubular carbon fibre and glued in place with high-strength epoxy. The modular circuit board carrier and battery holder were also 3D printed and ultimately held in place by 3mm screws and nuts. The final assembly of the complete MAV is shown in Figure 6.

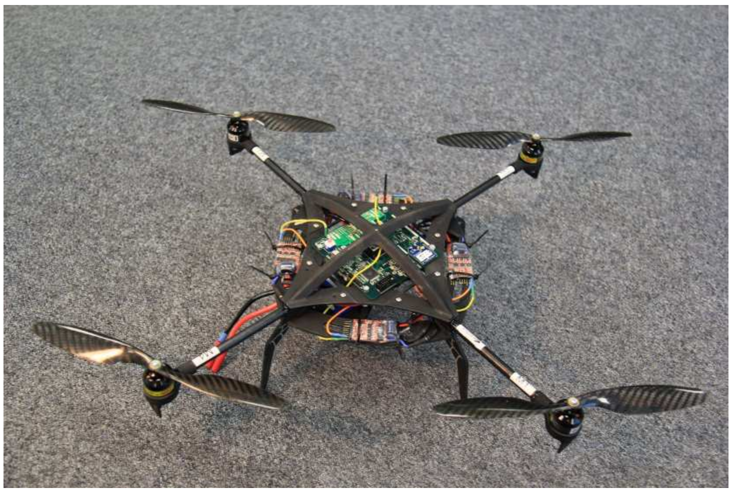


Figure 6 – Assembled Araqnobot

The relevant mechanical parameters for this airframe are contained in Table 3.

Table 3 - Physical Parameters of Araqnobot Airframe

Arm length (l)	0.265 m
Assembled mass without battery	780 g
Assembled mass with battery (g)	1.12 kg

At the outset, this airframe employed the same propellers as QBot1, but a decision was eventually made to transition away from plastic propellers and to use carbon-fibre ones instead. They have the same EPP-style shape and design, they are also 10” long and have a 4.5” pitch, but they are significantly more rigid and therefore behave closer to the ideal assumptions we have made (linear response during flight; not subject to significant flapping or distortion). The thrust and drag constants for this propeller were assumed to be the same as those for its plastic counterpart and once flight was achieved it was easy to experimentally see that the thrust component, at least, was essentially the same. This is accomplished by using the measured mass of the airframe to work out the thrust required to achieve static hover (F=mg, 10.98N in this case); if the thrust being computationally applied by the controller is significantly smaller, or greater, then the thrust constant must be correspondingly incorrect. In this case, the calculated thrust aligned very well and the previous number was carried forward. The drag constant is much harder to perceive or measure with precision and, for basic yaw control, that precision is considered to be unnecessary. As such, the value determined for the plastic propeller was applied to the carbon-fibre propeller and it yielded reasonable yaw control. The constants so determined are shown in Table 4.

Table 4 - Thrust and Drag Constants for Carbon-Fibre 10x45 Propellers

Thrust constant (b)	0.0000146 N per rad^2/sec^2
Drag constant (d)	0.00000038 N·m per rad^2/sec^2

4.1.3. Large Aluminium Airframe (Jumbo QBot)

The final quadrotor MAV developed entirely as part of this study was nicknamed Jumbo QBot. Its purpose was to move beyond the limited capability of the previous airframes and establish a platform of significant payload capacity (both in terms of mass and area) and computing power that could be used for research beyond the scope of the work described herein (hopefully for years to come). As a design effort, some principles were derived from the previous airframes but otherwise this airframe was to be entirely different. Something borrowed was the concept of hollow arms to house wiring and interconnect. The central and topmost position of the control board was also maintained, along with a protective frame for it as had been used on Araqnobot. Almost everything else was different. Rather that 3D printed nylon and carbon-fibre rods, the airframe material was all aluminium. This kept the airframe light (although heavier than the previous structures) and allowed assembly by screws and nuts rather than glue. It thereby has the advantage of easier repair and replacement of component pieces.

Another difference is that Jumbo QBot has no central body but rather relies on two sets of square aluminium bolted to the top and bottom of the arms. One set is placed near the centre of the airframe and the other is extended further out on the arms. These components keep the arms in alignment and provide torsional rigidity to the airframe. The arms themselves are made from rectangular extruded aluminium which was machined to provide access holes and bolt points in the desired locations. A pair of battery clips was incorporated to hold the larger battery by its ends rather than across its entire length. Landing gear is built from simple angled pieces and provided in various lengths to allow for the incorporation of larger objects (payload) underneath the airframe. To accommodate some of the foreseen payload, aluminium pieces were crafted to allow attachment of a laser scanner or an actuated camera platform. The entire airframe mechanical design was captured in Solidworks™ and is shown in Figure 7 (with the longest landing gear depicted).

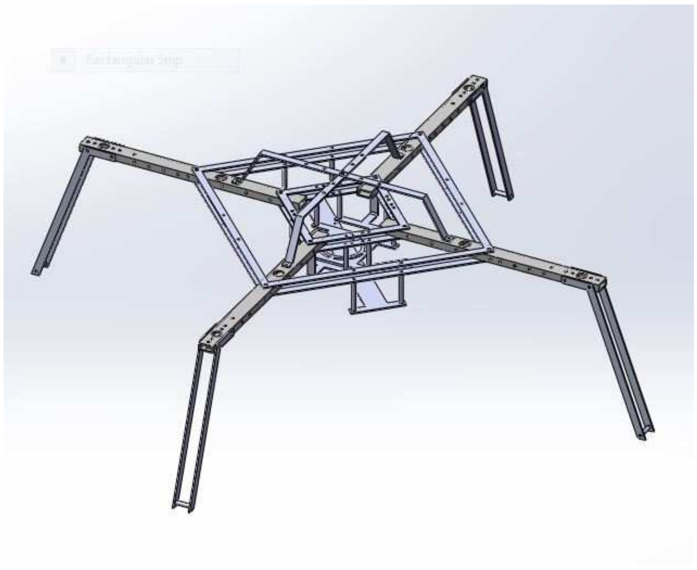


Figure 7 – Jumbo QBot 3D CAD Model

The design was completed as part of this research, but the machining of the pieces was accomplished in the mechanical development facility of AUT (by a technician); some of the construction was done by hand, but the majority was performed by computer-numerically-controlled (CNC) machines. The assembled bare airframe is shown in Figure 8 while the fully assembled MAV is shown in Figure 9.

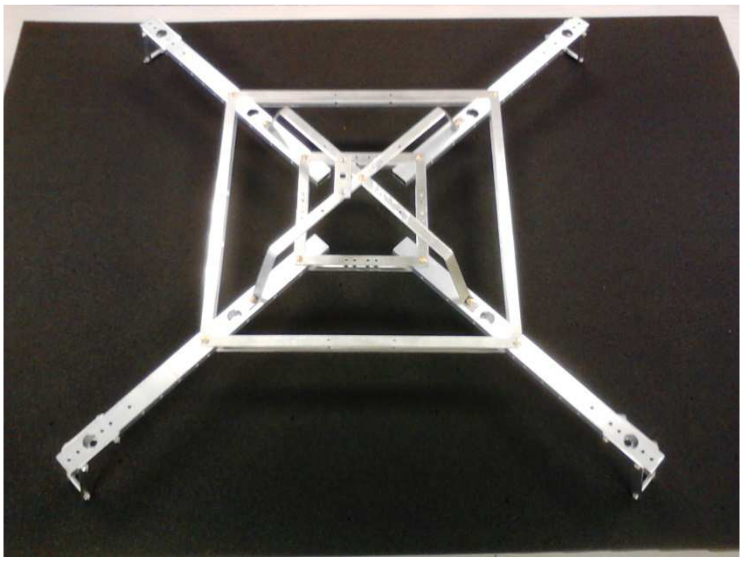


Figure 8 - Jumbo QBot Bare Frame

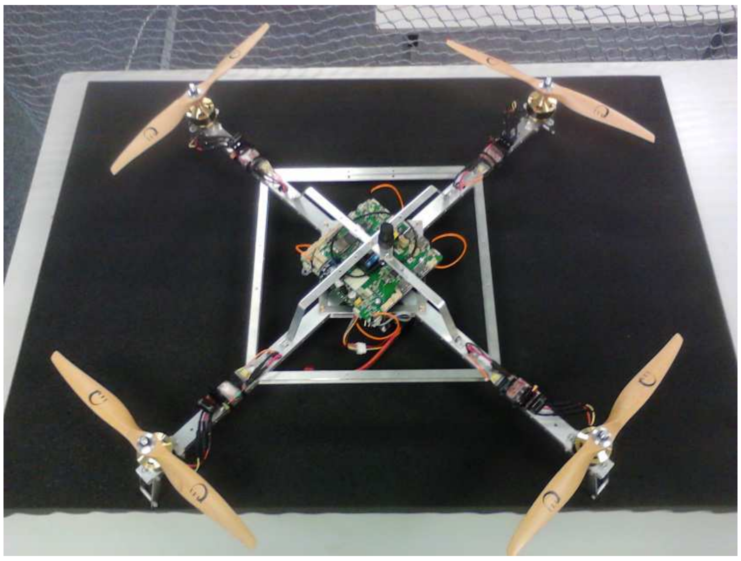


Figure 9 - Assembled Jumbo QBot

The physical parameters for this completed system are shown in Table 5.

Table 5 - Physical Parameters of Jumbo QBot Airframe

Arm length (l)	0.3265 m
Assembled mass without battery	1.72 kg
Assembled mass with battery (g)	2.39 kg

This airframe was a platform for significantly larger, more powerful motors, and the propellers increased along with them (and are made by a company called Xoar). Diameter was increased to 12" while pitch decreased slightly from 4.5 to 4". The construction material of the new propellers was changed again and this time beech wood was selected on the basis of its lightness, rigidity, and ability to withstand damage without failing entirely (several of these propellers were damaged over the course of this research and almost all of them would still be able to sustain flight, albeit in a reduced capacity).

The lift thrust constant was initially calculated with equation (4), using numbers posted on the motor manufacturer's website (Scorpion Power System Ltd., n.d.) for the Xoar propeller. It was later fine-tuned by experimental observation in the same manner as that applied to Araqnobot (stable hover effective lift thrust extrapolated back to thrust constant). The drag constant was approximated with equation (7) (d is equal to propeller power divided by propeller angular rate cubed) using the motor manufacturer's power and RPM numbers (Scorpion Power System Ltd.). This yielded perfectly acceptable performance and Table 6 shows the constants applied in the control algorithms.

Table 6 - Xoar 12x4 Propeller Constants

Thrust constant (b)	0.000018 kg*m/rad^2
Drag constant (d)	0.00000038 kg*m^2/rad^2

4.2. Electro-mechanical

4.2.1. Motors

The only components of the quadrotor MAV system that convert the control signals from the processor board and the electrical potential of the battery into mechanical movement are the motors. Each developed airframe had a different set of small but powerful brushless direct current (BLDC) motors. QBot1 employed MK2832/35 motors from a German company called Mikrokopter. Araqnobot used Robbe Roxxy 2827-35 motors while Jumbo QBot was powered by Scorpion SII-3008-1090KV(V2) motors. A comparison of the significant motor parameters is given in Table 7 (extracted from website information ("MK2832/35", 2009) ("Robbe ROXXY 2827-35", 2009) (Scorpion Power System, 2013)).

Table 7 - Motor Comparison

Parameter	MK2832/35	2827-35	SII-3008-1090KV
Weight	~68g with cable	~69g with cable	100g with cable
Max continuous current	9A	9A	26A
Max continuous power	110W	110W	370W
No-load speed	760 rpm/V	760 rpm/V	1090 rpm / V

It should be noted that the connector cables for the first two motors are quite long as compared to the Scorpion motor. Also, the Mikrokopter and Robbe motors are generally presented as appropriate for equivalent use. This is apparent in the chart as the parameters are nearly identical.

4.3. Electrical

There are many electrical components that make up a quadrotor system, from the simple (and increasingly tiny) resistors to the complex ARM processor used for processing in Jumbo QBot. The majority of the bits and pieces, the wires and the passive components, are necessary for correct operation, but insignificant for the purposes of this research. The major electrical components, however, are discussed in the following sections.

4.3.1. Batteries

As with motors, different batteries have different applications and there are always trade-offs to be made between battery capacity and corresponding mass. To have some consistency between the airframes, though, this research utilized 3-cell Lithium Polymer (LiPo) batteries exclusively. Individual LiPo cell voltage is nominally specified at 3.7V (4.2V fully charged) which gives 11.1V (12.6V) total for a 3-cell pack. The original battery used for QBot1 had a capacity of 3300 mAh and a mass of 260g. Araqnobot employed 4000mAh battery packs that weighed in at 340 grams. The high current requirements of Jumbo QBot called for a much larger battery and the one eventually employed weighed 670 grams and boasted a capacity of 11000 mAh.

4.3.2. Inertial Measurement Units

The inertial measurement unit (IMU) is the most critical sensor for quadrotor flight. In reality, IMUs are integrated devices that combine several sensors together, perform input filtering and sensor fusion calculations, and provide, at a minimum, useful orientation data as output. Two different types of IMU were employed in the course of this research; the UM6 from CH Robotics was used for both QBot1 and Araqnobot while the 3DM-GX3®-45 (referred to as LM345 herein) from LORD MicroStrain® was used for Jumbo QBot.

4.3.2.1. CH Robotics UM6

The UM6 provides basic IMU functionality and contains an accelerometer, a gyroscope, and a magnetometer to detect orientation and changes in orientation. It has an internal processor that performs Kalman filtering on the sensor inputs and computes both Euler Angles and Quaternion representation. It also offers the ability to perform filtering over GPS values input from an external device, but that feature was never used. The primary interface to the UM6 device is either SPI or UART and the serial communication can run at up to 115200 baud.

The communication protocol developed by CH Robotics is a fairly straightforward packet type that utilizes a fixed recognizable header and checksums for ensuring packet integrity in both directions. Different packet types are then defined for reading and writing registers and for an array of commands that provide access to all the necessary functionality of the sensor.

The source code that runs on the internal processor of the UM6 is also open source, which would be nice if changes were necessary. Overall, though, the sensor worked very well with no alteration.

4.3.2.2. LORD MicroStrain® 3DM-GX3®-45

The LM345 module is an advanced military-grade IMU with an extensive feature list including integrated GPS. It contains (at least) two processors and provides fully fused and filtered basic IMU data (derived from accelerometer, gyroscope, and magnetometer), as well as a set of computed (estimated) navigational data that is the result of a Kalman filter applied to both the IMU sensor data and the GPS module updates. Unprocessed GPS data is also available and these three sources of orientation and position data can be programmed independently to provide all of the information a mobile platform might desire.

Communication with the system processor can occur either via USB or serial RS232. The LM345 datasheet states that while the USB interface provides superior bandwidth (more than the device could theoretically use), it is limited in determinism (regular timing of updates) due to the nature of USB infrastructure in a modern processor system. The RS232 interface, on the other hand, can be strictly timed out of the LM345 and is advised for use in situations where regular deterministic timing is required or desired (as is the case for a quadrotor MAV application). Bandwidth on the RS232 interface must be actively managed in the system design to ensure that the requested data output of the LM345 does not exceed the capability of the interface. As it happens, that overrun situation was encountered in the course of this research and the data rate needed to be increased from the default of 115200 baud. The final data rate selected was 960000 baud, not necessarily because that much was needed but because it worked well and decreased the amount of time taken for data transfer.

The one element not integrated into the LM345 that is necessary for full functionality is the GPS antenna. Both a passive helical antenna and an active puck-style antenna were tested and, in open spaces, the performance of each was perfectly acceptable. It is expected, however, that in areas where GPS signal quality is diminished, the active antenna would provide better performance but at the cost of added mass (about 50g).

4.3.3. GPS

QBot1 was originally fitted with an LS20036 standalone GPS smart antenna module from Locosys Technology Inc. That device is a small form factor completely integrated GPS solution with an extensive feature set including the capability to track up to 32 satellites at a time. It communicates with a processor via a very basic serial interface (1 transmit, 1 receive wire) running at 9600bps. By default, it provides GPS updates at a rate of 1 Hz, but it can be programmed to provide them at higher rates, up to 10 Hz. At one point in time, the control board for QBot1 was upgraded and the LS20036 couldn't be transferred intact. Because outdoor flight was no longer planned for that platform, a replacement for the new board wasn't considered a priority and this component was dropped from the design (but could easily be added back in at any point in the future).

Araqnobot never had a GPS unit, but it has a location reserved on its control board for the LS20036, the same as QBot1.

A standalone GPS module wasn't included in the design of Jumbo QBot because the LM345 provided adequate GPS capability, as described above.

4.3.4. Wireless Communication

None of the quadrotor MAVs developed during the course of this study were intended to be remotely controlled in the sense commonly used by hobbyists. As such, standard remote control (RC) transmitters and receivers were not employed despite their obvious pertinence to this type of endeavour. Instead, a more conventional model of robotic operation was pursued that involved a base station sending commands to the robotic platform on an as-needed basis while the robot otherwise operates autonomously. Communication from the robot to the base is generally intended to be informational only and conveys telemetry and state data that is deemed useful to an operator/observer. In that sense, then, all that is needed is a basic digital link between a computer on the ground and the robot as it moves around. This was achieved in the case of all three platforms through the use of XBee® modules from Digi International.

XBee modules have small processors on them that can generally support a number of different protocols and communication paradigms (for example, point-to-point or mesh). This research exclusively used the XBee firmware supporting point-to-point communication employing the IEEE 802.15.4 communication standard. QBot1 and Araqnobot both used wire-antenna low power XBee modules than have an indoor range of 30m and an outdoor line-of-sight (LOS) range of up to 90m. Jumbo QBot has two sets of long-range modules available. One set operates at 900 MHz and could communicate up to 140m indoor and 3km with LOS outdoor (up to 10km with high-gain antennas). The other set operates at 2.4GHz and can reach 1.6km with LOS outdoor or up to 90m indoor. Testing started with the 2.4GHz setup but eventually the modules were switched to gain the greater range and avoid interference issues that had been encountered at 2.4GHz.

One of the greatest benefits of using technology like the XBee modules is that all of the complications of wireless communication are taken care of by the devices themselves. A small amount of initial setup is required, but after that the only interfaces to be dealt with are common UART serial ports that communicate end to end as if the intervening XBee infrastructure were simple wires. On the robot, the UART interface is connected to the processor (directly for QBot1 and Araqnobot, via USB for Jumbo QBot) and on the base station, the paired XBee is connected via a USB dongle that presents a serial interface to the operating system. What is sent serially from the robot is received directly by the base station, and vice versa. This keeps the interfaces and the protocols extremely clean and simple; perfect for reliable embedded communication.

4.3.5. Altitude Sensors

Three methods of detecting altitude (height above the ground might be more accurate in some cases) were explored during this study, each using different hardware. The first method applied indoor was sonar, next was air pressure, and the third for use outdoor was GPS. GPS position sensing has already been discussed and won’t be covered again here.

4.3.5.1. MB1200 Sonar Module

For sensing distance to the ground, an MB1200 (XL-MaxSonar®-EZ0™) from MaxBotix® Inc. was selected. These devices are small form factor ultrasonic modules that operate on a principal of echo location (like bats use). They have the ability to detect objects from 0 to 765 cm away and can theoretically provide distance information with 1cm accuracy for objects between 20cm and 765cm. Their beam width is reasonably narrow (about 1.8m across when 1.5m away from an object (e.g. the floor)); care would be necessary indoor to ensure clearance from nearby objects (like walls, desks, etc.). The processor interface provided is a reduced-amplitude RS232 that is tied to the device power for logic high (i.e. either 5V or 3.3V) and ground for logic low, rather than a standard RS232 voltage range (e.g. ±12V). The device communicates only in one direction and, when enabled, sends calculated range information every 99ms at a baud rate of 9600. The protocol is very basic and consists of a known packet arrangement (fixed header, trailing carriage return) that gets populated with the variable range information (to the nearest object detected). Its size and simplicity of use make it a good choice for MAV application.

4.3.5.2. Pressure Sensors

Only Jumbo QBot (to date) employs air pressure sensors to determine altitude. It accomplishes this in the same way that other aircraft do around the world; it obtains an ambient air pressure reading and applies a formula (equation (29)) to convert that to altitude. If a reading is taken at the point of takeoff, the relative difference provides the effective height of the aircraft above the ground (at least as long as the ground is level or the robot stays near the point of launch). Obtaining pressure readings of sufficient accuracy is somewhat challenging and two different pressure sensors were tested during this study. The first one is made by a company called Measurement Specialties and has a model number of MS5803-01BA (called just MS5803 hereafter). The second sensor tested is made by Bosch Sensortec and has model number BMP180. Both of these sensors are small and highly capable (although they look quite different physically). Their major attributes are summarized for side-by-side comparison in Table 8.

Table 8 - Pressure Sensor Comparison

Feature	MS5803-01BA	BMP180
Operating Pressure Range	300 – 1100 hPa	300-1100 hPa
Communication Interface	I^2C and SPI up to 20MHz	I^2C up to 3.4MHz
Maximum Output Resolution	0.012 hPa	0.01 hPa
Absolute Accuracy 0-50 °C, 300-1100 hPa	-1.0 to 1.0 hPa	-4.0 to 2.0 hPa (-1 +/- 1 hPa Typical)

Feature	MS5803-01BA	BMP180
Relative Accuracy 950…1050 hPa at 25 °C	Not Listed	+/- 0.12 hPa Typical
Typical RMS Noise Maximum Resolution Setting	0.012 hPa	0.03 hPa

These values are taken from the datasheets for the MS5803 (Measurement Specialties, 2013) and the BMP180 (Bosch Sensortec, 2012), respectively. Although the MS5803 seems to generally have better operating parameters, the lack of a relative accuracy specification was a cause for concern. In the quadrotor MAV application being studied, the relative accuracy was deemed of greater importance and so both sensors have been employed and, in practical terms, they have nearly identical performance on the Jumbo QBot platform. In the end, both are being used simultaneously and their data is averaged to give an overall improvement in useful observed accuracy.

Although these modules have processor communication interfaces, they do not directly provide pressure data over them. Instead, the output of internal ADC measurements for temperature and pressure are relayed (after appropriate conversion times) and must be processed according to the algorithms given by the manufacturer. Only at the end of that processing is the sensed pressure known.

As a further note on the current application, the interface for the BMP180 is I2C as listed, but it was actually determined that the module would be more useful with a USB interface so that it could be deployed either on the robot or the base station (to track and eliminate the impact of changes in the local barometric pressure). For that reason, the BMP180 is installed on a standalone circuit board with a small processor that does all of the I2C processing and subsequent calculation. The resulting pressure value is then relayed over the USB bus as a serial stream with known packet arrangement and checksum for maintaining integrity.

4.3.6. Processors

4.3.6.1. Atmel® ATmega2560

Both QBot1 and Araqnobot employ ATmega2560 microcontrollers for all of their processing. As 8-bit processors go, the ATmega2560 is reasonably full-featured and has proven an excellent choice for this type of application. It operates at a maximum frequency of 16MHz and has 256 KB flash memory, 8KB SRAM, and a 4KB EEPROM. In addition, it offers a large array of configurable I/O lines, 6 programmable timers (that support PWM output for motor control), 4 UARTs, a 16-channel 10-bit A/D converter, and a JTAG interface for programming and in-circuit debugging.

4.3.6.2. Gumstix® Overo FE Computer on Module

Gumstix is a company that specializes in making components for tiny intelligent systems. They are perhaps best known for making powerful but tiny computer modules that are roughly the size of a stick of gum (and presumably they selected their name based on this fact). For Jumbo QBot, an Overo FE COM was selected (hereafter referred to as simply the Gumstix or Gumstix Processor) and it was decided that the full embedded Linux operating system would serve as the platform for robotic operation. As the use of COM (Computer on Module) in the product name suggests, this is not simply a processor. Rather, it is a full computer system that incorporates processor, SRAM, (flash) disk drive, USB I/O, ADC capabilities, and networking and serial interfaces. It has prodigious processing capability and its feature list states that it is capable of achieving up to 1400 Dhrystone MIPS (Gumstix, Inc., 2012). Almost all that might be expected of a standalone personal computer is wrapped up in this tiny package that's capable of running Linux, Android™, and other operating systems.

The relevant high-level features of the Gumstix processor used for this research are summarized in Table 9.

Table 9 - Gumstix Overo FE COM Feature Summary

Processing Unit	Texas Instruments OMAP3530 Applications Processor
Processor Architecture	ARM Cortex-A8
Processor Speed	720 MHz
SDRAM	512MB DDR at 200 MHz
Flash	512MB built-in; microSD slot for more
Networking Capability	Wifi (IEEE 802.11 b/g) and Bluetooth
USB Support	Native Host port (USB 2.0 high-speed only) and On-The-Go port
Other Serial	SPI, I2C, UART (2 available channels)
PWM Outputs	6
ADC Channels	6

It should be apparent that this processor is in an entirely different class than the Atmel. Its USB capability was leveraged to offer even more functionality, and in that way 9 additional serial UART channels were added and four USB plugs were placed on the control board (with the potential to be used for external disk drives, a laser scanner, and almost anything else with a standard USB interface). Similarly, additional ADC channels were added on the I2C bus and the total ADC capability on the board was increased from 6 to 14.

4.3.7. Motor Speed Controllers

Aside from the physical structure (which is important and significant), the only major difference between QBot1 and Araqnobot are the motor speed controllers. These small devices are crucial to the effective use of the BLDC motors that actuate the aircraft and they are commonly referred to by several terms; electronic speed controllers (ESCs), BLDC motor controllers, and motor drivers, to name a few. The core function of these devices is to control the flow of power into a motor such that its speed is regulated in accordance with an electronic input signal. That input signal can theoretically be anything, but a digital signal employing pulse width modulation (PWM) is most common. As the name implies, PWM communicates information through the width of a digital pulse and this allows a single wire to communicate a control variable between a master and slave (communication is unidirectional). This enables the use of a very simple timer-based approach for both driving and sampling the wire in question. Common pulse widths for this type of control frequently range from 1ms to 2ms where 1ms conventionally corresponds to zero throttle while 2ms corresponds to full throttle. The specific values and the maximum update rate vary significantly between controllers, as does corresponding motor response to the output power.

Both QBot1 and Araqnobot employed PWM to convey the desired control (motor throttle), but, as stated previously, different ESCs were used. QBot1 used the Mikrokopter BL-Ctrl 2.0 while Araqnobot was equipped with the AutoQuad ESC32. Jumbo QBot also used the ESC32 but added serial UART communication as an option for conveying the motor control update information (it can use either PWM or UART but the current default is UART).

4.3.7.1. Mikrokopter BL-Ctrl 2.0

The BL-Ctrl 2.0 is an excellent ESC built around an Atmel ATMega168. It was designed specifically for use with Mikrokopter motors and airframes but offers a broad feature set and, thanks to its design intent, is generally a good choice for multi-rotor MAV use. Either PWM or I2C can be used for control updates and the I2C interface can also be used for extracting data monitored by the ESC (like temperature and voltage). Only the PWM connection was used on QBot1 (partly because there is no direct documentation for the communication protocol; only reference code for a Mikrokopter control board).

4.3.7.2. AutoQuad ESC32 2r1

The ESC32 from AutoQuad is an extremely powerful but still reasonably priced motor controller. Its central processor contains a 32-bit ARM Cortex core running at 72MHz. It boasts I2C, PWM, CAN bus, and UART ports. The firmware it runs is all open source and thereby available for analysis and modification. When using the UART port (or any other bidirectional port, presumably), control information is not limited to motor control alone. The default firmware offers a whole range of control parameters and also supports configurable telemetry broadcasts that can be used to provide automatic, regular updates of motor information back to the controlling processor (e.g. motor RPM, current draw, voltage…).

One of the most useful features of this ESC (and the developer’s software) is the ability for self-calibration and subsequent closed loop operation. When fitted with a propeller and installed on a fixed stand (or on the application airframe that is held in place), the software can be used to execute either of 2 calibration routines. The first is for associating voltage to RPM so that the motor controller can explicitly bring the motor to a desired rotation. This is a significant capability because the central controller of a MAV generally doesn’t care about throttle and really wants to be able to set motor rotation. Throttle is loosely assumed to be linear across the operating range and can thereby be used to approximate the desired rotation, but an explicit indication of RPM and the corresponding ability of the motor controller to actively achieve it is a definite improvement.

The second calibration routine is used for current limiting. In general, the motor controller, the motors and other wires and circuitry in a system have limits on the amount of current they can handle. Most motor controllers have built-in worst case assumptions to offer some protection in this regard, but greater efficiencies and response times can be achieved through explicit tuning of the motor and propeller set being used in a given application. With this routine, then, the motor controller performs a series of motor throttle steps to work out what rate of change and, ultimately, what maximum throttle is acceptable for a given current limit. For example, a direct step from 10% throttle to 90% is almost certainly going to exceed any normal current limit. The motor controller evaluates this and plots the required steps necessary to move from 10% to 90% without exceeding the limit.

With the closed-loop mode parameters calculated and applied, this motor controller offers optimal responsiveness and setpoint accuracy. Moving from PWM to serial communication further improves the clarity of the desired setpoint (it is explicitly specified, not encoded as a pulse width and then decoded again) while reducing the response time (i.e. no 2ms wait for PWM pulse width to be conveyed). Altogether, this controller was easy to work with and offered exceptionally good performance.

5. Software and Firmware Design

There are different definitions (and conceptions) for firmware and software (or at least the difference between them) around. For the purposes of this study, firmware is simply defined (delineated) as the code that runs on the embedded robot processor/microcontroller while software is defined as the code running on the base station PC. For all three quadrotor MAVs constructed for this research, the firmware they run is entirely contained on the robot itself in a non-volatile way. On startup, they contain all of the code necessary for flight and only operational commands are passed from the base station to the robot. In that sense, the delineation is appropriate and can be rigorously applied and understood.

All of the firmware developed as part of this research was written in C. While the structure and approach was largely portable, necessity required the use of Atmel AVR primitives that needed to be changed to Linux constructs as development moved to the Gumstix platform.

The base station software applications have been developed for Windows[1] (currently running on the Windows 7 operating system) and were written entirely in C# using Microsoft Visual Studio (2010 Professional and 2012 Ultimate).

A modular approach to firmware development was employed in developing the code for embedding in the quadrotor MAVs. The intention therein is to place code specific to a given peripheral in its own .c (code) and .h (header) files. The term ‘peripheral’ is used here to refer to all components and modules external to the central processing unit (even if they reside in the same chip) that require some form of programmed communication and control to be employed. The main processing loop also has its own file pair and it pulls in (using include statements) the header files of the peripherals that it uses. In this way, peripherals can be easily added to or dropped from the system and, as the firmware infrastructure was ported from one processing system to another, corresponding changes to peripheral handling were easily isolated. In the following sections, the root file name of the peripheral code (i.e. filename without extension) is included in parentheses for any description pertaining exclusively to that code set (to make the association easier for anyone studying this document and the developed code at the same time).

[1] Windows is a registered trademark of Microsoft Corporation in the United States and other countries

The following parts of this chapter are not intended to fully describe (and certainly not to recreate) the code that was written for this project, but rather to call out the major functional components and describe them in a useful way.

5.1. ATmega2560 Firmware

Development for a simple microcontroller has several decided advantages:

- What is written and compiled is generally all that will run; there are no other software routines vying for processor time
- Access to the hardware is direct and absolute; there are no intervening redirection or security mechanisms
- Analysis at the processing instruction level is reasonably easy and corresponding determinism (outside of interrupts from external sources) can be depended upon

These factors make a microcontroller like the ATmega2560 an excellent platform for the development of firmware targeting robotic applications. It can be simply and directly done with corresponding clarity of operation and responsiveness of the controlled system.

5.1.1. Peripherals

5.1.1.1. Wireless Communication (uart_XBee)

The code to interface with the XBee devices is relatively straightforward as it is, from the microcontroller point of view, simply UART communication. As such, the standard AVR UART initialization occurs in a routine called XBee_uart_init(). That routine sets the port baud rate and it must be the same as the value programmed into the connected XBee. It also initializes the receive channel interrupt and thereby places the microcontroller in a ready to receive state.

Bytes received from the XBee associated UART are stored in a 256-character ring buffer. The interrupt processing routine simply takes each byte received and places it in the next ring buffer index. When the index reaches 255, the next value is 0 which forms the ring. There is no checking of overflow as it is expected that bytes will be extracted from the buffer long before 256 could accumulate. If that ever fails to happen, the receive interrupt will simply overwrite data that has not yet been read.

The other significant functions contained in uart_XBee are a function to indicate how many characters (if any) are available in the ring (XBee_uart_bytes_available), and another for outputting a single character at a time (XBee_uart_putchar_printf). The character output function was declared in such a way as to allow association with a C file handle. This is significant because, in this way, the default C output construct, stdout, can be directly assigned to the XBee. That then allowed for simple code construction using generic printf statements to be used throughout all other source files.

5.1.1.2. *Attitude Sensor (um6_imu)*

Like the XBee, the UM6 IMU connects to the microcontroller via a UART interface. An initialization routine (um6_imu_uart_init) was written to setup the Atmel registers for correct operation, including receive interrupt processing. The exact same type of 256-byte ring buffer is applied in this case as was used for the XBee.

Sending information to the UM6 is always accomplished a single character at a time through the um6_imu_uart_putchar routine (no printf statements are employed in this case).

The most common interaction with the UM6 involves sending a command sequence that initiates the transfer of whatever parameters the device is currently programmed to relay. A standalone function called um6_imu_req_data is used for this request and it defines each character of the transmit string explicitly (including header and checksum characters).

Data from the UM6 to the microcontroller can be monitored either via the um6_imu_bytes_available routine (when the number of expected bytes is known) or the um6_imu_check_input_packet routine (when the incoming packet length is unknown or variable). Depending on the nature (or expectation) of the inbound data, it can be processed by several different routines:

- um6_imu_print_generic_packet will print any received IMU packet of known length
- um6_imu_get_triplet_raw will extract any returned set of 3 integers (most UM6 IMU information comes in triplets)
- um6_imu_get_roll_pitch_yaw_degrees will extract and properly scale (to degrees) returned UM6 Euler angles of roll, pitch, and yaw

To allow for the reading and writing of registers and for the posting of commands, three additional functions were created: um6_imu_generic_read, um6_imu_generic_write, and um6_imu_generic_command, respectively. Each of these routines will build up and transmit a packet as defined by the UM6 datasheet (CH Robotics, 2013) to accomplish the corresponding function.

5.1.1.3. Sonar Rangefinder (mb1200_sonar)

The MB1200 sonar module connects to yet another UART port on the ATMega microcontroller and the code developed for it uses exactly the same type of initialization routine (mb1200_sonar_uart_init). Apart from the initial sameness, however, it differs from the other UART peripherals in several ways:

- A multiplexer sits between the microcontroller and the physical sonar device. This allows up to 4 sonar modules to be connected at once (although only one can be communicating with the microcontroller at a time)
- The MB1200 only transmits data; it has no ability to receive commands or data from the microcontroller
- The data transmitted by the MB1200 is a set pattern that repeats (when enabled) at regular intervals to provide 4-byte updates of range to the nearest object; there is no need for a dynamic receive structure (like the ring buffers employed in other areas)

With these characteristics in mind, the code written for the MB1200 provided only two external functions beyond mb1200_sonar_uart_init. One is for initializing the I/O pins controlling the multiplexer and for establishing the data structure to receive sonar updates. It is named mb1200_sonar_control_init and it must actually be executed prior to mb1200_sonar_uart_init. The other routine is for enabling the sonar module output (mb1200_sonar_enable) and it takes a short integer argument to select between the multiplexed candidates.

5.1.1.4. GPS Module (ls20036_gps)

A fourth UART on the ATmega2560 was dedicated to the optional GPS module. Although this component was never used in flight, it was tested to verify proper connectivity and correct circuit board design. Use of a 256-byte ring buffer, receive interrupts, and initialization routine (ls20036_gps_uart_init) are all pretty much the same as for the XBee and the UM6. There were no routines written to parse incoming data and only the generic byte count (ls20036_gps_bytes_available) and character pop (ls20036_gps_get_next_char) functions are currently provided. A couple of data stream configuration routines were developed to test device configurability and extract useful state information; ls20036_gps_gga_rmc_only sets up the LS20036 to transfer two specific blocks of GPS information while ls20036_gps_gsv_only sets up transfer of only one (different) block of data. The definition of the GGA, RMC, and GSV data blocks can be found in the LS20036 datasheet (LOCOSYS Technology Inc., 2009).

5.1.1.5. Analog-to-Digital Converter (adc)

The ATmega2560 ADC is incorporated into the microcontroller and 7 of its 16 available channels were connected to monitoring points on the QBot1 and Araqnobot control boards. The seven monitored values included current draw by each of the four motors and voltage levels for the 3 cells of the attached LiPo battery. Initial setup of the ADC is accomplished by a basic initialization routine (adc_init) that selects the reference voltage (external input), programs the frequency, and enables the completion interrupt. It does not, however, actually start any conversion.

Although the ATmega2560 ADC has 16 channels, it only operates on one of them at a time. Routines were written that would allow monitoring of a selection of single channels: adc_check_motor_current could be used with a motor number integer to sample any single motor current; adc_check_7v4 would allow testing of the 7.4 volt output of the battery (which was selected because the control board is powered from that point). Alternatively, all seven connected channels could be iterated through using the adc_full_loop function. The sampled values of each channel would be stored in corresponding variables and the collective current state could be printed at any time using the adc_print_status command.

5.1.1.6. Motor Control (motor_control)

For QBot1 and Araqnobot, the motors were controlled entirely through the PWM outputs of the ATmega2560. The configuration of individual pulses is accomplished through internal timers that are configured to repeat and to output high or low logic signals as various thresholds or limits are encountered. The initialization of the timers is performed by the motor_control_init routine and it sets the starting pulse width parameters such that the motors are off. After initialization, motor_control_change_throttle can be used to assign different pulse widths (different timer on-to-off thresholds) to a specified motor.

Those two basic functions provide all that is necessary for the most basic motor control, but the motor_control codebase is more extensive than that. Because the motors themselves can be thought of individually as peripherals with different characteristics (even motors of the same make and model vary significantly), it was deemed useful to maintain the motor control variables alongside the controller interface code. From there, it also seemed reasonable that the PID code affecting motor control would be added, along with the conversion logic from thrust and torque to individual rotation and then from rotation to PWM. Because of the significance of these routines, they are discussed in greater detail below.

5.1.1.6.1. Roll, Pitch, and Yaw PID Control

Although the code to implement the orientation PIDs is little more than a rendering of the theoretical equations, it is included here in its entirety (Figure 10) because of the critical role it plays in the operation of the quadrotor MAVs. This block of code illustrates the simplicity and elegance of PID control. When the sensed information is accurate, nothing more than this is required to achieve stable orientation maintenance. For the most part, the written lines of C should be self-explanatory but a quick note should be made on the role of the yaw adjustment logic (to supplement the comments in-line with the code). For the non-aerobatic MAVs developed and studied by this research, only yaw will ever see a deviation from its setpoint that approaches 180 degrees. The issue with being oriented 180 degrees (π radians) away from the desired value is that there are two possible directions to take in resolving the discrepancy. This can correspondingly lead to indecision or erratic behaviour, especially if the robot has some momentum that might carry it from one side of 180 degrees to the other. The solution is to create a band (+/-10 degrees in this case) around 180 degrees within which the robot will only move to the setpoint directly (via the shortest path) if it does not already have some velocity taking it in the other direction.

```c
float motor_control_calculate_PID_rpy (uint8_t select, int current_value,
	int current_desired, float time_interval, int angular_velocity,
	float kp, float ki, float kd)
{
	int current_error;
	float current_error_scaled;
	float current_integral;
	float current_derivative;

	const float conversion_factor_angle = 0.0001917471;
	//This will convert the IMU euler angle output to radians
	const float conversion_factor_angular_velocity = 0.0010652652;
	//This will convert the IMU angular velocity to rad/s

	float PID_output;

	current_error = current_desired - current_value;

	if (select == 2) {
		//for now, only yaw will approach 180 degrees
		if (current_error > 16384) {
			//we're more than 180 degrees away from desired (which means
			//less than 180 in the opposite dir)
			if ((current_error > 17294) || (angular_velocity < 0))  {
				//within 10 degrees of inflection, only flip if we're
				//not moving in the right direction
				current_error = current_error - 32768;
			}
		}
		else if (current_error < -16384) {
			if ((current_error < -17294) || (angular_velocity > 0)) {
				current_error = 32767 + current_error;
				current_error++;
			}
		}
	}

	current_error_scaled = current_error * conversion_factor_angle;

	current_integral = last_rp_integral[select] +
			   current_error_scaled * time_interval;
	current_derivative = angular_velocity * conversion_factor_angular_velocity;

	PID_output = kp * current_error_scaled +
		     ki * current_integral -
		     kd * current_derivative;

	last_rp_integral[select] = current_integral;

	return PID_output;
}
```

Figure 10 – C Code for Roll, Pitch, Yaw PID Control

5.1.1.6.2. Converting Desired Thrust and PID Control Outputs (Torques) to Motor Rotations

When all of the airframe desired torques have been calculated according to the PID parameters and the current orientation of the airframe, it is possible to apply the equations given in (25), (26), (27), and (28) to work out the desired angular rate for each motor. Figure 11 shows the code that implements this operation and it should be easy to identify the rendering of the equations therein and to see that the theory has been implemented directly in practice.

```
Motor_Rotations motor_control_calculate_motor_rotations(float roll_pid_value,
       float pitch_pid_value, float yaw_pid_value, float total_thrust)
{
       float quarter_yaw_scaled, quarter_total_thrust_scaled;
       float half_quarter_total_thrust_scaled;
       float half_roll_scaled, half_pitch_scaled;
       float omega_squared[4];

       Motor_Rotations calculated_rotation;

       uint8_t idx;

       quarter_yaw_scaled = yaw_pid_value / (4 * drag_constant);
       quarter_total_thrust_scaled = total_thrust / (4 * thrust_constant);
       half_quarter_total_thrust_scaled = quarter_total_thrust_scaled / 2;
       //need to limit the yaw input
       if (quarter_yaw_scaled > half_quarter_total_thrust_scaled) {
              quarter_yaw_scaled = half_quarter_total_thrust_scaled;
       }
       if (quarter_yaw_scaled < - half_quarter_total_thrust_scaled) {
              quarter_yaw_scaled = - half_quarter_total_thrust_scaled;
       }
       half_roll_scaled = roll_pid_value / (2 * arm_length * thrust_constant);
       half_pitch_scaled = pitch_pid_value / (2 * arm_length * thrust_constant);

       omega_squared[0] = quarter_total_thrust_scaled + quarter_yaw_scaled +
                          half_pitch_scaled;
       omega_squared[2] = quarter_total_thrust_scaled + quarter_yaw_scaled –
                          half_pitch_scaled;
       omega_squared[1] = quarter_total_thrust_scaled - quarter_yaw_scaled +
                          half_roll_scaled;
       omega_squared[3] = quarter_total_thrust_scaled - quarter_yaw_scaled –
                          half_roll_scaled;

       for(idx = 0; idx < 4; idx++) {
              if (omega_squared[idx] < motor_idle_rotation_squared) {
                     //printf("\n\n*****************\nERROR ERROR ERROR ERROR\n
                     //MOTOR %d WANTS TO STALL\n*************\n\n",idx);
                     calculated_rotation.omega[idx] = 84;
              }
              else {
                     calculated_rotation.omega[idx] = sqrt(omega_squared[idx]);
              }
       }

       return calculated_rotation;
}
```

Figure 11 - Motor Rotation Calculation C Code

Naturally, there are some practical considerations that needed to be applied and that weren't apparent from the equations alone. In the first place, the large range of yaw (+/-180 degrees is possible) and the smallness of the drag constant contributing to yaw torque combined to yield a huge potential impact of yaw on the motor outputs. For a yaw deviation (from the desired value) of any significant magnitude, it was possible to see that the yaw factor contribution would effectively shut down two of the motors while doubling (or more) the output of the other two. In theory, that would be acceptable, but in practice, each balanced motor pair needs to operate at a significant thrust level in order to maintain balance. For that reason the impact of thrust on any given motor is limited to half of the total thrust that motor is currently intended to provide (i.e. yaw compensation can't cause the motor to drop or increase by more than half its current throttle setting).

The other practical code modification was to put a bound on the lowest possible motor angular velocity allowed. In a purely theoretical application, a motor's rotation can go to zero (and even negative), but real motors will stall at some point above zero (somewhere less than but near 84 rad/s in this case) (and they can't spin backwards without wiring modification). Shutdown in itself might not be too bad if the motor response remained consistent. Unfortunately, starting a BLDC motor from stall can take a significant amount of time (extremely inconsistent with normal speed changes) and it is, therefore, necessary to prevent stall and limit how slow the motors are allowed to go.

5.1.1.6.3. Motor Rotation to PWM

The final stage of the motor control process is to convert the motor rotational velocity (ω) into a PWM pulse width. Using the assumption of motor linear response to PWM (at least near the operating point), the conversion is straightforward and the corresponding C code is shown in Figure 12.

```
uint16_t motor_control_convert_rotation_to_pwm(uint8_t motor_num,
                                               uint16_t rotation_in)
{
       uint16_t calculated_pwm;

       calculated_pwm = motor_rotation_pwm_offset[motor_num] +
                        rotation_in * motor_rotation_pwm_constant[motor_num];
       return calculated_pwm;
}
```

Figure 12 - Motor Rotation to PWM Conversion C Code

What is not shown in the code is the declaration of the motor constants. For QBot1, those had to be worked out experimentally and were different for each motor. For Araqnobot, the ESC32 motor controller was designed to actively seek linear response and, after calibration, the corresponding constants were all the same.

5.1.2. Communication between Base Station and MAV

As with other aspects of this research, the communication infrastructure between the base station and the robot evolved over time. The first thing achieved was basic communication of status from the MAV to the base station by setting up printf statements to output to the XBee by default (as discussed in Section 5.1.1.1). Having printf capability to relay information allowed for a complete range of debugging and monitoring statements to be embedded in the firmware. Throughout the project, that remained the only method used for sending dynamic information from the MAV to the base. Some statements use a formatted string for automatic parsing by the receiver, but all statements are relayed in clear text with no checksums or other data integrity added.

That approach was taken because of an original desire to test and debug the robot without the need to develop both MAV firmware and base station software concurrently. With clear text updates, all that was needed to interact with the robot was a basic text terminal and those are readily available for any PC. For initial development then, robot commands took the form of single characters that were entered on the controlling terminal. The responses were observed by the operator and the firmware was gradually expanded. Eventually it became necessary to relay multi-character strings (e.g. motor throttle value) and a packet structure was defined that included framing, header characters, and packet length. That was sufficient for some time but the switch from human interaction to real base station software highlighted the need for some form of integrity protection on the base station commands. It was at that point that a trailing checksum was added and the base to bot commands then took the form shown in Figure 13. The checksums are automatically enabled by the base station software but they can be disabled to allow for the original terminal-mode operation.

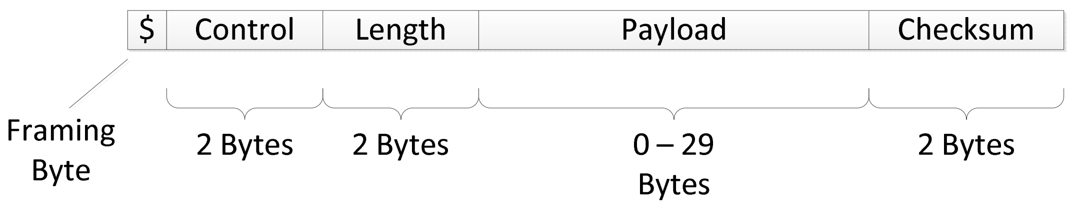


Figure 13 - Communication Packet Structure

The reason that checksums are only enabled one-way is due to the fact that communication from the MAV to the base station is only ever informational. Because the machines developed were all intended to operate without external processing, there is no reactive computation done on the base station (i.e. there is no case where the base station actively responds to changing state on the robot). As such then, there is no need to protect data relayed from the robot; if the information is corrupted, it cannot cause an operational fault and it will generally be corrected with the next status update. That said, there is still some sanity checking done by the system software; informational updates are parsed to extract their data and, in some cases, used to change displayed information in the GUI. The parsing routines rely on regular expressions that are reasonably targeted (e.g. only a string starting with '[status_mci] ' will be recognized as indicating motor controller initialization) and there is, therefore, a corresponding measure of protection built in.

This approach has the advantage of simplicity and clarity for direct human interaction, but it does lack the efficiency of binary encoding and the protection scheme is arguably weak. If it were to be designed again, a more robust full duplex packet protocol with better data integrity and the option for automatic retransmission would be developed.

5.1.2.1. Communication Processing Firmware (comm_processing)

At the outset of this research, all of the communication processing code was placed in the main processing loop. As the communication infrastructure grew, it dominated the file and obfuscated the code actually running the MAV. For that reason, it was extracted out and a large communication structure (C typedef) was created to hold all of the control parameters that could be updated by the base station. That structure was created as part of the main program and a handle was passed to the functions in comm_processing.c. There are only two significant routines: process_serial_command and process_serial_packet. Each accepts only a single character input because all receive communication processing on the MAV is limited to one character at a time; this ensures that the processing of large input strings never supersedes the time-critical processing of the PID loops.

The process_serial_command routine is stateless and used for those commands that need only a single character to identify the desired response (e.g. 'm' indicates that the base station is requesting motor initialization). Process_serial_packet is stateful and will accumulate characters matching the packet pattern until the identified length (from the header) is achieved. When checksums are enabled, all communication from the base station must be encapsulated in a packet with a valid checksum; single character commands are wrapped in the packet structure and, when extracted, processed by the process_serial_command function.

5.1.3. Main Program Loop

With all of the ancillary functions modularized, the main processing loop becomes quite simple. A high-level flow chart of its operation is shown in Figure 14.

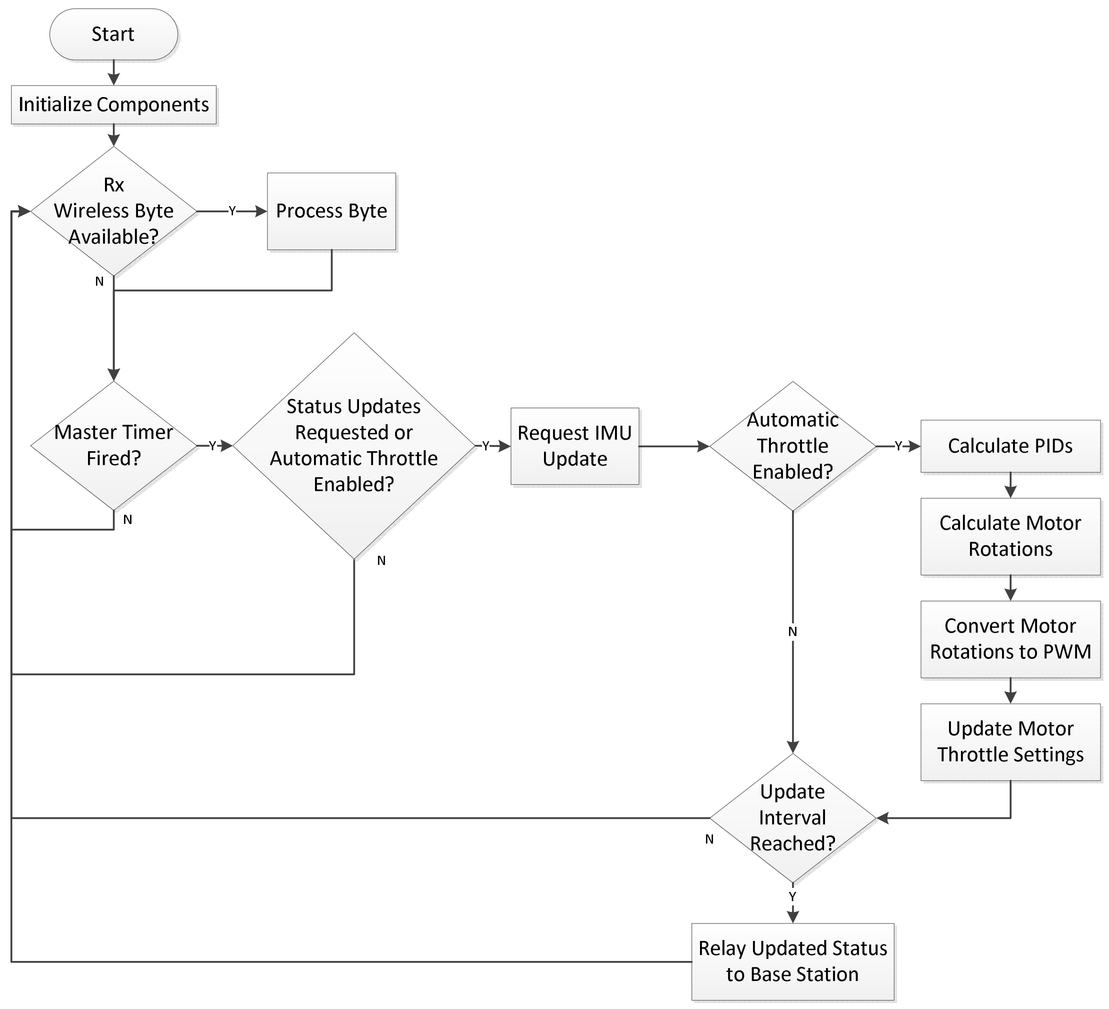

Start
Initialize Components
Rx Wireless Byte Available?
Y
Process Byte
N
Master Timer Fired?
Y
N
Status Updates Requested or Automatic Throttle Enabled?
Y
N
Request IMU Update
Automatic Throttle Enabled?
Y
N
Calculate PIDs
Calculate Motor Rotations
Convert Motor Rotations to PWM
Update Motor Throttle Settings
Update Interval Reached?
N
Y
Relay Updated Status to Base Station

Figure 14 - Main Processing Loop for ATMega Firmware

Processing begins by initializing the system components. This includes executing the initialization functions for each of the peripherals, by assigning stdout to the XBee UART output, and by starting the master timer (which has variable duration, but was always set to 10ms for the purposes of this research). After that, a free-running loop begins that starts by checking for any data that might have been received from the base station. If a byte (character) has been received, it will be processed (either as a packet byte or a single character command, depending on the communication state), but only a single byte is handled at a time, regardless of how many have been received. Following the communication processing is a check of the master timer. If it has not yet fired (e.g. less than 10ms have passed since the last firing), then the loop will return to the beginning and another byte, if one is available, will be processed.

When the timer fires, the MAV must be in a state wherein updates are relevant before any further processing occurs. If the base station has not requested perpetual updates and the robot is not in automatic throttle (orientation control) mode, nothing happens in response to a timer event and the loop begins again. If, however, either of those conditions is met, then there is a consumer of orientation data and the IMU will be queried. If the MAV is performing orientation control, the retrieved orientation data will be fed into the orientation PIDs and desired torques will be calculated. Those torque values are then combined and factored to yield the desired rotational velocity for each motor. Finally, the motor angular velocities are converted to PWM pulse widths and applied to the motors (via PWM timer updates). The detailed breakdown of that motor control process is given in Section 5.1.1.6. If automatic throttle control is not enabled, then those steps are bypassed, but either way, the logic reaches a point where it evaluates the need to update the base station.

From a desire to limit communication processing and reduce the needed bandwidth for the wireless link, updates are not relayed to the base station on every firing of the master timer. They are sent at a configurable reduced rate that currently defaults to once per second (or, more accurately, once for every 100 times the master timer has expired). That is sufficient for the base station to evaluate current MAV status but requires only a small amount of processing overhead and bandwidth. If the programmed update interval has not been reached, no data is sent to the host. If it has, then current telemetry (containing a variety of data that currently includes Euler angles, angular rates, battery status, etc.) is relayed over the wireless link. (For clarity here, this is sufficient to convey the concept, but the current firmware actually specifies two intervals: this allows for more frequent updating of variables with greater dynamism (e.g. the Euler angles) while still permitting judicious bandwidth management. (Both intervals currently default to one second.)) After the programmed data set has been relayed (or not), loop processing begins again.

5.2. Gumstix COM Firmware

The power of the Gumstix processor comes with overhead. While it is possible to imagine programming directly on the ARM processor without an operating system, the effort involved in properly configuring all the aspects of the system (including, for example SDRAM timing parameters and flash disk accesses) and then building drivers to use the networking modules, USB ports, ADC chip, etc. is simply too great. That work alone would be a complete research effort and certainly fell outside the goals of this study. Because an operating system was necessary, then, it is understood that it would come with issues (at minimum) of competition for CPU resources (in some cases from highest priority OS threads) and non-determinism (no guarantees of high-precision timing). A real-time operating system is a possible solution to these issues, but no free, well-supported OS of that type is readily available for the Gumstix. From the possible candidates, then, it was considered that, in a minimal configuration at least, Linux offered the lowest overhead and best potential for consistent responsiveness.

5.2.1. Embedded Linux Operating System

5.2.1.1. Building the Kernel

The operating system chosen to run on the Gumstix processor of Jumbo QBot is a distribution of Linux called Ângström and it is compiled using a build framework called OpenEmbedded. Pre-built images are available for the Gumstix processor, but it was considered desirable to build an operating system targeted to this application. Despite the fact that extensive instructions are available for this process on the Internet, it is still a challenging endeavour filled with hurdles.

The process starts with the selection of a build computer. An x86-based PC running Ubuntu 12.04 LTS (long term stability) was selected for this project. On that machine, the environment must be set up to include an array of software packages and tools that are required by the OpenEmbedded build process. With those in place, the desired kernel repository is targeted and clones of the current revision developer trees are brought local. The kernel revision of overo-2011.03 from the mainline repository was used for this research, along with revision 1.12.0 of the bitbake build environment.

With the build repository and command infrastructure in place, it is possible to launch the command to build the kernel that will be embedded: ‘bitbake omap3-console-image’. (Note that there are several kernel images that can be built, but the OS here is intended to run on a robot that has no means of displaying a graphical desktop, so the console image provides a small and efficient kernel intended for headless operation.)

That bitbake command will hopefully work without further intervention (as intended), but several hiccups were encountered at the time it was executed for the Jumbo QBot development. In the first place, the build process does rely on some standard code being available on the executing machine. All of the required code was present on the build machine used, but not all of it had the same revision as expected. As a result, the referring file (specifically: org.openembedded.dev/conf/distro/include/angstrom-2008-preferred-versions.inc) needed to be changed. Further along the process, Internet fetch issues were encountered. In addition to code being referenced on the build machine, most of the kernel-in-progress is built from code repositories on the Internet. Some of those repositories were either down or relocated (perhaps temporarily) and a number of times, the source code had to be manually retrieved. Thankfully, the build process did eventually complete, but it took several days of shepherding and coaxing.

The end result is twofold: a compiled kernel image is available for installation on the Gumstix, and cross-compilation tools are available on the build computer. The latter point is significant because it enables development and compilation of firmware on a PC with all of the processing power and graphical tools that come with it (although the Gumstix processor is quick, it's no match for a modern PC).

Two relatively short steps follow the kernel compilation and they simply build the files necessary for initialization and booting of the kernel on the Gumstix. With those files and the built kernel, everything is ready for installation on a Gumstix file system. That file system was built up on an 8GB Class 10 microSD card according to instructions given on the Gumstix development website (the file system has a specific structure and format) (Gumstix Inc., 2013). When the file system is ready and all boot and kernel files have been installed on it, it can be placed in the microSD slot on the Gumstix processor board.

The first time the Gumstix boots a new image, it will perform a one-time sequence of configuring all installed modules. That process takes quite a while but when it completes, the system will be fully functional and a command prompt will be presented.

5.2.1.2. Adding Tools to the Kernel

At this point, all that has been built is a stock image presented exactly as the developers defined it. That could be sufficient, but it is equivalent to a pre-built image that could have been downloaded. It is at this point, then, that modification and tailoring was applied for the purposes of this research. (As a side note, modification and tailoring could have been done prior to building the kernel, but it was deemed preferable to test and demonstrate a successful build prior to making modifications so that the fault could more easily be determined if any should arise.)

The first thing that was done for the Jumbo QBot OS was to build into the kernel several useful packages that would improve efficiency and usability of the system. These included vim (a text editor), screen (a console enhancement), and all the compiler tools (to allow development and recompilation on the robot itself). This is a fairly basic modification, but the process is still a bit obfuscated. First, in the root build directory, a new subdirectory must be created that is called ‘user.collection’. During the build process, bitbake searches for files in the local repository (that was originally cloned from the developers’ on the web) and it will check the user.collection directory first to see if the local user has overridden any of the files it needs. If it finds a user-defined file, it preferentially includes that one and ignores the developers’ clone. That is exactly what was needed here and a copy of the bitbake command file (called a bitbake recipe) for the omap3-console-image (built above) was copied over and placed in a parallel location (in this case, the copied file becomes user.collection/recipes/images/omap3-console-image.bb). That file can now be modified to add or remove high-level components of the kernel. The packages mentioned above were added to the “TOOLS_INSTALL” list and the kernel was ready to be rebuilt.

5.2.1.3. Modifying Kernel Configuration

With a kernel now running and built with the tools desired, testing of components began and it was quickly seen that the default configuration would not quite work for Jumbo QBot. A standard Gumstix release comes with touchscreen capability and a display driver enabled (even on the console image). Those components would never be used on Jumbo QBot and they reserve pins on the Gumstix that are needed for serial SPI communication. As a result, the root configuration file (in this case, org.openembedded.dev/recipes/linux/linux-omap3/overo/defconfig) needed to be modified. That file defines all of the significant parameterized options for a Gumstix application and it can either be modified directly (it is a basic text file), or a new one can be created using bitbake (command: “bitbake -c menuconfig virtual/kernel”). The process for doing this (and more) is well documented on the jumpnow website (Ellis, 2012b). The end result is that the defconfig file receives two changes:

- The line containing ‘CONFIG_TOUCHSCREEN_ADS7846=m’ is changed to ‘# CONFIG_TOUCHSCREEN_ADS7846 is not set’
- The line containing ‘CONFIG_PANEL_LGPHILIPS_LB035Q02=y’ is changed to ‘# CONFIG_PANEL_LGPHILIPS_LB035Q02 is not set’

When the kernel is rebuilt, the SPI chip selects will be available and the spidev driver for them should be automatically loaded on the running system (with corresponding file handles: /dev/spidev1.0 and /dev/spidev1.1).

5.2.1.4. Patching the Kernel

As the development effort began in earnest on the Gumstix, a few issues were encountered that needed to be dealt with. The 2011.03 revision was originally selected because it seemed to be stable and good instructions were available for working with it, but it is also being left behind as developers move to newer revisions. That means that although certain problems are known to the development community, they are no longer being actively resolved in that stream. This was the case for 3 encountered problems:

1. The wifi driver would often stall the boot process for several minutes
2. The Gumstix ADC Channels 2-6 were inactive
3. Loading the driver to use the PWM channels would cause a kernel taint message to appear

These problems were all resolved through the use of kernel patch files. A kernel patch is basically a file that describes changes that need to be made to source files in the current repository. It does this through explicit definition of the paths to the files that need to be changed and then it specifies line numbers and context lines around the desired changes. In this way, it is ensured that the patch will only be applied to files that exactly match the original specification; the whole patch will be rejected if any file fails to align with the changes requested. It is then possible to find patches online that have been created by someone in the open source community for the distribution and release that is being worked on. This was the case for the first two issues and the corresponding patch files were created/compiled by Scott Ellis and Ben Keane:

- Wifi issue patch file was named libertas-async-fwload.patch
  - Linked as part of a Gumstix discussion group thread and compiled from patches originally created by community contributor known as Donny3000 (Ellis, 2012a)
- ADC issue patch was named madc-adcin3-6.patch
  - Linked as part of a Gumstix discussion group thread (Keane, 2012)

The PWM kernel taint issue was fairly minor, but it was deemed worthwhile to explore the manual patch creation process to aid with understanding and thereby potentially allow more significant kernel modification in the future.

The warning stemmed from the fact that the PWM driver that was being used for Jumbo QBot had been developed out-of-tree (presumably outside the primary development tree for the kernel) and the kernel was incorrectly flagging that with a taint warning. The problem was fixed in a later kernel but required patching in the kernel for Jumbo QBot. That later kernel fix is found in a repository update that was written by Ben Hutchings (Hutchings, 2011); unfortunately, it couldn't be applied directly to the source for Jumbo QBot, but it did identify the file that needed to be patched (panic.c) and the flag at issue (TAINT_OOT_MODULE).

In the local build directory for this project, the file in question was found here: tmp/work/overo-angstrom-linux-gnueabi/linux-omap3-3.2-r103/git/kernel/panic.c. The process for building a patch file (albeit for a different file: board-overo.c) is described on the kernel development page already mentioned (Ellis, 2012b). The process begins by copying the file to be modified into a backup location (panic.c-orig). Then the changes are made; for the purposes of this effort, the modification was quite small and applied to panic.c as follows:

- `if (flag != TAINT_CRAP && flag != TAINT_WARN && __debug_locks_off())`
  was changed to
- `if (flag != TAINT_CRAP && flag != TAINT_WARN && flag != TAINT_OOT_MODULE && __debug_locks_off())`

After that, a patch file was created using the following git command:

```
git diff --no-prefix git/kernel/panic.c git/kernel/panic.c-orig >
my_pwm_kernel_taint.patch
```

At this point, three patch files were ready for application to the kernel. To include them in the build, they were first copied to the directory containing the kernel build recipe: org.openembedded.dev/recipes/linux. Then they were added to the recipe file (linux-omap3_git.bb) as additional sources by appending them to the SRC_URI variables as follows:

- Before:
  ```
  SRC_URI = "git://www.sakoman.com/git/linux-omap-2.6.git;branch=omap-
  3.2;protocol=git \
  ```

```
file://defconfig \
file://${BOOT_SPLASH} \
"
```

- After:

```
SRC_URI = "git://www.sakoman.com/git/linux-omap-2.6.git;branch=omap-
3.2;protocol=git \
file://defconfig \
file://${BOOT_SPLASH} \
file://my_pwm_kernel_taint.patch \
file://madc-adcin3-6.patch \
file://libertas-async-fwload.patch \
"
```

After a rebuild and installation on the Gumstix COM, the Linux operating system finally had all of the tools and functionality needed for Jumbo QBot operation and development.

5.2.2. Porting the Firmware from ATMega to Linux

With an operating system and corresponding build environment in place, it was possible to start developing the embedded firmware that would run on Jumbo QBot. The first task undertaken was to port the C code that had been developed for the ATMega to Linux. Parts of the code that were purely logical (e.g. arithmetic) could be used directly and it was only the parts tied directly to hardware (e.g. register writes for component initialization) that needed modification. The modular development approach provided easy isolation of those routines and the process, while somewhat time-consuming, was reasonably simple.

In this way, then, the initial firmware structure for the high-powered processor of Jumbo QBot was functionally identical to that employed on QBot1 and Araqnobot. The peripherals employed on those earlier airframes could easily be used by the new model and in several cases they were. In the section that follows, there is some redundancy with descriptions already given and that ground will not be covered again. Only the differences will be discussed here for the sake of brevity.

5.2.3. Peripherals

5.2.3.1. Wireless Communication (XBee)

Once operational, the functioning of a serial port on Linux is no different than operation on a microcontroller. Of course, a user level program in Linux doesn't have direct access to hardware registers and everything must be done by system calls. These are easy to employ, however, and any decent serial communication guide for Linux describes the process and functions necessary to achieve character communication as had been defined for use by this research. One significant difference that was immediately encountered is the necessity in Linux to use file handles for most hardware interaction (a file handle is an integer that unambiguously refers to an I/O device). Initialization of a device is not the first thing that occurs; instead, the file handle for the device must first be requested and then used for all further interactions. This meant that functions needed to be added to the XBee source code that would open the device (request the file handle from the OS) prior to first use and then close it again (release the file handle) prior to exit.

The hardware initialization process also had to be changed, but the intent and corresponding outcome were exactly the same. The primary difference is the scope of configurability of a serial port in Linux. The interface presented to a user application is the endpoint of a software stack that has developed over time (as opposed to a set of basic hardware registers) and now supports an array of functions covering a broad spectrum of intervening (between the application and the hardware) capability. Setting baud rate and character framing are directly analogous between the two systems, but higher-level aspects of communication exist that must be initialized properly to achieve the desired behaviour. As an example, serial ports in Linux can operate in ‘canonical’ mode where information is not relayed until a carriage return or newline is encountered. The communication protocol for this research relied on timely byte-by-byte communication and therefore initialization of the XBee serial port required ‘non-canonical’ mode to be explicitly set.

Because Linux provides its own buffering of serial port data, there was no need for a receive ring structure. The number of bytes in the buffer can be determined in a non-blocking way from a system call (specifically, the ioctl function was used) and then a simple read command is all that’s needed to fetch available data. There was also no need to convert the ATMega UART write routines as Linux inherently supports using printf to send data to a serial port file handle. Nothing more than the reassignment of the stdout variable was needed to enable correct handling of all the transmit communication statements that were embedded in the code.

5.2.3.1.1. XBee as Terminal

The preceding section defines the model of interaction with the XBee as it was originally defined, and it is a reasonable and sufficient model for interaction with an XBee as a peripheral. Linux has the capability, however, of launching a terminal and connecting it to a serial port at startup (or thereafter). This was the model eventually employed for Jumbo QBot and it has the advantage of allowing for complete system access (including configuration, recompiling, and anything else possible from a command terminal) over the wireless link. When the system boots, then, the base station is presented with a Linux login prompt for a command terminal, rather than simply the interface for the developed firmware.

Operating the developed MAV code from within that terminal then means that the XBee file handle does not need to be requested from the system. For receive, it is already identified by the STDIN system variable and, for transmit, there is no need to redirect STDOUT because the XBee is already the targeted output device. This has worked well and the current firmware reflects the expectation of operating in this mode (but routines for the alternative file handle request/reassignment/release approach are still in place).

5.2.3.2. Attitude and Position Sensor

The LM345 sensor comes with a complete software development kit (SDK) available from the manufacturer’s website. That kit includes a full range of C code and header files that provide access to almost all of the device’s functionality. Furthermore, it includes sample test software for Linux and the corresponding code to setup the Linux communication infrastructure to work with the device. As such, then, most of the effort to use this sensor for the purposes of this study involved extracting the needed functions and data structures from the SDK and the sample code and then modifying their use (not the content) to fit the Jumbo QBot application.

Using the high-level functions provided by the SDK, the device is initially setup according to the manufacturer recommendations (Microstrain, Inc., 2012, pp. 16-19), but with slight modification. The sequence is as follows:

1. Initialize the interface
2. Place sensor into idle mode
3. Setup NAV data-stream format (for data out of the internal navigation processor)

4. Setup AHRS data-stream format
5. Setup GPS data-stream format
6. Enable NAV, AHRS, and GPS data-streams and initialize data processing routines for incoming sensor data
7. Reset the filter
8. Setup the magnetometer as the heading source
9. Initialize the filter from the AHRS

These steps are carried out at the beginning of the Jumbo QBot firmware application and after that, the LM345 operates independently to sense its current state and provide regular updates for all of the parameters associated with the data-streams. Default update rates were used and the AHRS and NAV information sets are both relayed every 10ms (100 Hz) while the GPS information is limited to every 250ms (4 Hz). The data handling routines were copied directly from the Microstrain sample code and placed into the Jumbo QBot firmware file main.c (which contains the main operational loop). They were the only part of the software for this sensor that was modified and the change was slight: a global variable indicating completion of incoming packet processing was set at the end of each routine. This allows the main processing loop to reliably detect completed sensor updates and then perform its control loop calculations.

5.2.3.3. Altitude Sensors

5.2.3.3.1. MS5803 Pressure Sensor (ms_pressure_sensor)

The MS5803 is connected to the Gumstix COM via its SPI bus and the standard Linux SPI driver, spidev, was used to communicate with it. The manufacturer provides an application note on its website (Measurement Specialties, 2011) that includes C code for interfacing with and operating the sensor when connected to an Atmel microcontroller (which was conveniently familiar). That code was extracted and modified to work with the Linux spi driver, but was effectively unchanged in procedure and processing. The process of translation did, however, highlight one difficulty of this type of re-application. Where byte-wise operation was assumed for the Atmel device, it couldn’t be made to work on the Gumstix. It was eventually realized that the difference lies in the fact that the Atmel code controls the SPI chip select line external to the read and write commands. As a result, chip select remains asserted for an entire sequence of single-byte operations. The Linux spidev driver, on the other hand, asserts and releases chip select as part of its read and write operation. Performing single-byte accesses was therefore causing incomplete transactions on the SPI bus and nothing worked properly.

Eventually, the code was successfully adapted and the device began to return temperature and pressure values as expected.

5.2.3.3.2. USB connected BMP180 (usb_bmp180)

The BMP180 pressure sensor, as it was purchased for this project, has only an I2C interface. Furthermore, it wasn’t originally intended for incorporation onto the robot but was rather targeted for the base station (to provide ambient pressure compensation over time). For those reasons, it was assembled onto a small USB interface board that contained an Atmel ATMega48 as the primary point of interface and computation. The code written for that small board was technically part of this research effort, but falls outside the central topic and is therefore not discussed further here. What is important is that the output of the USB interface is a serial stream of temperature-compensated 16-bit pressure updates (in pascals). The Jumbo QBot firmware that interfaces with it then includes the following significant functions:

- usb_bmp180_open obtains the device file descriptor
- usb_bmp180_init initializes the serial interface
- usb_bmp180_process_line parses the next line received from the USB module
- bmp180_calculate calculates altitude and height from current pressure

5.2.3.3.3. Altitude Sensor Filtering

Both pressure sensors employed on Jumbo QBot exhibit a large amount of instantaneous noise on the sensed pressure (and corresponding calculated altitude). The obvious solution to that problem is to smooth out the readings and provide a more stable altitude reading by applying some form of low-pass filter. Selection of the filter and the corresponding filter parameters then poses a problem of its own as there is a huge array of possible options (and opinions surrounding them). The ideal filter would be simple to implement and effective in filtering unwanted noise while still tracking real changes in the sensed value. There are often trade-offs between those desired characteristics, but the approach eventually selected provides a good measure of all. It is called a complementary filter and is based upon the definition of that term given by Shane Colton in a presentation entitled ‘The Balance Filter’ (Colton, 2007). (His presentation is actually an excellent resource that covers some of the popular approaches to sensor fusion in a balancing robot application.)

The filters applied to the two pressure sensors on Jumbo QBot are then both implemented in the main processing loop. Whenever there is an updated pressure reading, a (generally small) percentage weighting of the update is summed with the remainder percentage (100% total) to give the filtered value. This is a straightforward implementation of a low pass digital filter that correlates well with intuition (a recurring theme over the course of this research). It should be apparent that sudden changes only contribute a small amount to the instantaneous value and will be offset by opposing sudden changes such as would be seen by high frequency oscillation (or noise). On the other hand, a static value or slowly moving change will be well tracked by the ongoing summation.

The same type of filter is applied to both sensors to give smoothed altitude values. A digital derivative (change divided by time delta since last update) for each sensor is then calculated from the smoothed value to give an approximation of vertical velocity. Because of the noisy nature of a digital derivative, that velocity value is filtered again by the same complementary technique.

5.2.3.3.4. Pressure Sensor Fusion

The two pressure sensors connected to Jumbo QBot were eventually combined to collectively give a single value for both altitude and vertical velocity. The approach was simple: each contributes 50% of the final values after all other calculations and filtering have been completed.

5.2.4. Analog-to-Digital Converter

Although test code was written for the AD7998 (which sits on the Gumstix I2C bus) and significant effort went into ensuring that the Gumstix built-in ADC channels worked (using polling software found online (Ellis, 2012c)), neither capability is currently being used in active Jumbo QBot firmware. The C code is available for incorporation into the main firmware program and there are many ways in which it will be useful in the future, but it wasn’t ultimately needed for this research.

5.2.5. Motor Control (motor_control and motor_scontrol)

The application of the theoretical equations for working out the orientation PIDs and calculating desired motor rotational velocity is exactly the same for Jumbo QBot as for QBot1 and Araqnobot. The details of these critical functions are presented and discussed in Section 5.1.1.6 (and contained in the firmware motor_control.c and motor_control.h files). For the first part of Jumbo QBot development, everything was kept the same, including the use of PWM. It was eventually determined, however, that the advantages of moving to serial control (e.g. immediate, explicit setting of desired rotational velocity) should be seized upon and that the capabilities of the Gumstix COM and the ESC32 motor controllers in this area should be utilized.

Each ESC32 motor controller on Jumbo QBot is attached to a USB serial UART channel. The interaction with the controllers is accomplished by a range of software routines and definition files that are provided online (Nesbitt, 2012) and distributed as open source software. The code downloaded for this research is actually the calibration routine software, but it provides a great resource for embedded development. With that software infrastructure in place, interface files (esc32.c and esc32.h) were created to wrap it all together and set up the environment as desired for Jumbo QBot.

As each motor controller is initialized (motor_scontrol_open), the corresponding serial port is opened and a processing thread is spawned to manage the receive data (the motor controllers can be configured to provide telemetry updates at regular intervals). This is done so that the main processing loop doesn't have to manage incoming data from yet another set of information sources. Instead, independent threads are managed by the operating system and each maintains its own set of received telemetry information that can be retrieved by the central loop at any time. Before completing the initialization function, communication with the motor controller is tested to ensure that all is well (a no-operation command is sent and an acknowledgement must be actively received).

There are then several functions provided that allow arming (must occur prior to starting) (motor_scontrol_arm), starting (motor_scontrol_start), stopping (motor_scontrol_stop), and rpm target setting (motor_scontrol_set_rpm) of the motors. Telemetry updates can also be initialized at any time (motor_scontrol_start_telemetry) but should be stopped prior to exit (to prevent overflowing the UART receive buffer) (motor_scontrol_cleanup). This is more complex and extensive than simple PWM setting, but it is also a significant improvement in capability.

It is hopefully apparent that the motor_scontrol file-set is not a replacement for motor_control. The motor_control .h and .c files still provide all of the functions (like the PID logic) needed to control the motors while motor_scontrol .h and .c simply provide the handling routines for serial port interaction.

5.2.6. Position Calculation Logic

While the firmware necessary for orientation hold is naturally associated with a particular peripheral (the motors (controllers)), the functions related to autonomous position tracking are somewhat more difficult to associate. The output of the position hold calculations is a set of Euler angles and not a direct feed to a particular actuator. Instead, those Euler angles take the place of manual updates that would come from the base station and are applied to make the robot change orientation in a way that will move it toward (or keep it at) the desired LLH coordinates.

For that reason, the code (position_calc) is discussed here. It is an adjunct to the main processing loop, referenced in the same way as peripheral code, but not directly tied to a given peripheral's function. The role it plays in the fully autonomous operation of Jumbo QBot is critical, though, and the core PID function is described below.

5.2.6.1. LLH PID Calculation

The LLH PID calculation logic is shown in Figure 15. The current error is first calculated and then, for latitude and longitude, converted to meters. The integral term is updated and then a proportional term limit is applied. As noted in the comments, the application of a P-term limit represents an effective bound on velocity (in the absence of an I-term; both would have to be limited if both were used and a velocity restriction is desired) which has proved helpful in maintaining control when actually flying. Finally, the PID formula is applied and the resultant output returned. For latitude and longitude, the output is in radians (for pitch and roll, respectively) while for height, the output is a thrust offset in Newtons (that will be applied to a static hover value).

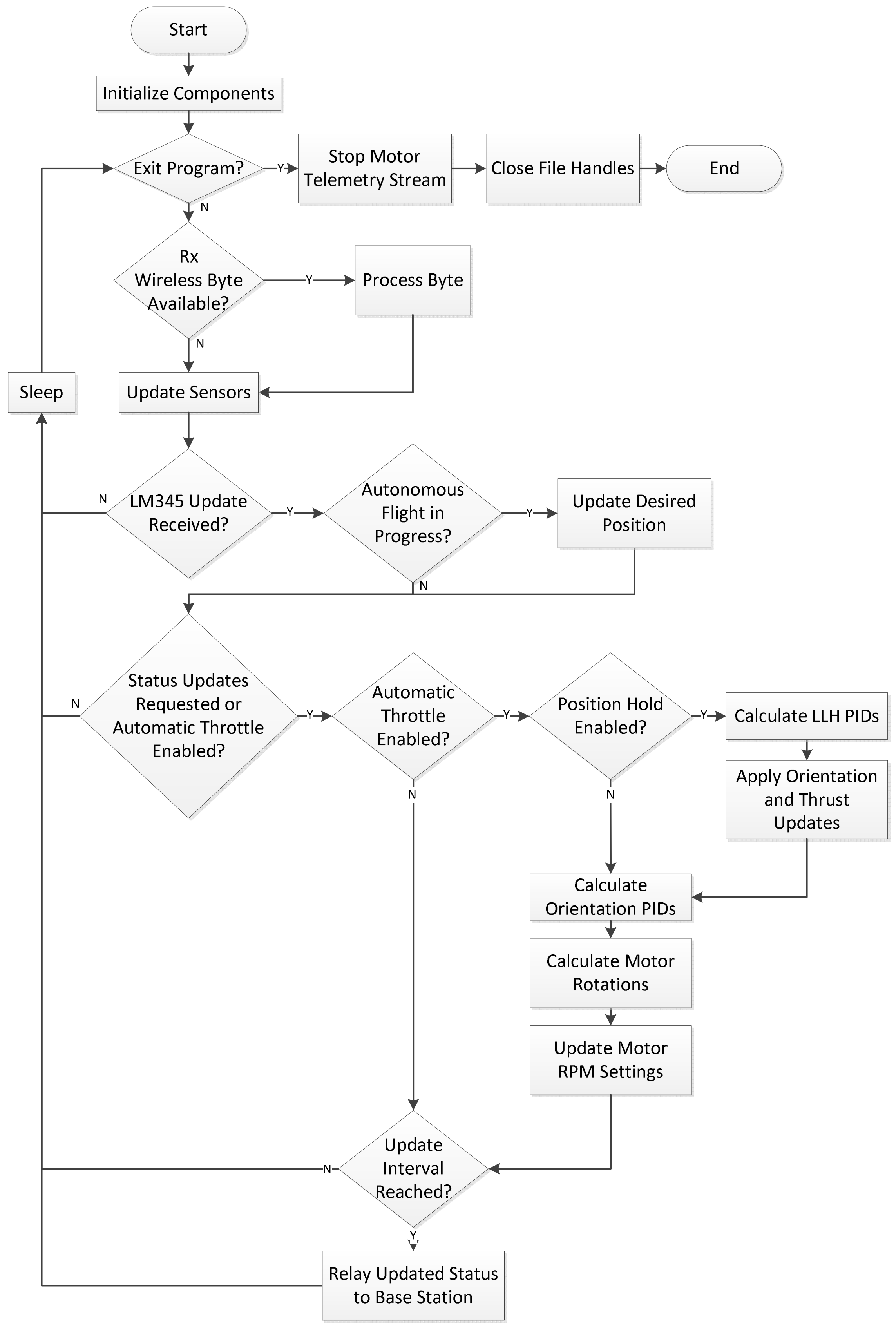
As with the orientation PID calculations, there is no special logic here; the theoretical formula has been directly applied and the code shown demonstrates the simplicity and clarity of the rendering.

```
float position_calc_PID_llh (
       uint8_t select, float p_term_limit, double current_value,
       double current_desired, float time_interval, float velocity,
       float kp, float ki, float kd)
{

  float current_error;
  float current_integral;
  float current_derivative;

  float p_term;
  float PID_output;

  current_error = (float) (current_desired - current_value);

  //first, convert degrees to meters
  switch(select) {
    case 0: //latitude
      current_error = current_error * latitude_deg_to_meters;
      break;
    case 1: //longitude
      current_error = current_error * longitude_deg_to_meters;
      break;
    default: //height is already in meters
      break;
  }

  current_integral = last_llh_integral[select] + current_error * time_interval;
  current_derivative = velocity;

  //Limiting the p_term means that we can control maximum velocity (for now
  //assuming that the i-term is 0 - which is the case on April 10, 2013)
  p_term = kp * current_error;
  if (p_term < 0) {
    if (p_term < -p_term_limit) p_term = -p_term_limit;
  }
  else {
    if (p_term > p_term_limit) p_term = p_term_limit;
  }

  PID_output = p_term +
               ki * current_integral -
               kd * current_derivative;

  last_llh_integral[select] = current_integral;

  return PID_output;

}
```

Figure 15 - LLH PID Calculation for Position Control

5.2.7. Main Program Loop

In terms of high-level functionality, the main processing loop for Jumbo QBot is only slightly more complex than that employed for QBot1 and Araqnobot. The macroscopic program flow is show in Figure 19.



Start
Initialize Components
Exit Program?
Y
Stop Motor Telemetry Stream
Close File Handles
End
N
Rx Wireless Byte Available?
Y
Process Byte
N
Sleep
Update Sensors
N
LM345 Update Received?
Y
Autonomous Flight in Progress?
Y
Update Desired Position
N
N
Status Updates Requested or Automatic Throttle Enabled?
Y
Automatic Throttle Enabled?
Y
Position Hold Enabled?
Y
Calculate LLH PIDs
Apply Orientation and Thrust Updates
N
N
Calculate Orientation PIDs
Calculate Motor Rotations
Update Motor RPM Settings
N
Update Interval Reached?
Y
Relay Updated Status to Base Station

Figure 16 - Jumbo QBot Main Processing Flowchart

When the program is first launched, it begins by initializing all of the components of the system. This involves opening the device file handles, initializing the I/O (including beginning the motor telemetry streams), and setting variables to startup values. Something not shown in the flow chart (due to its low-level nature) is that file handles for writing telemetry data to disk are also opened. Telemetry data from all of the sensors and the motors is stored in an array of dynamic variables (system memory) during flight and then written to disk once the flight completes. This conceptually happens in the background to the main program loop but is useful to understand the extent of analysis possible on a platform like this. In any case, if the exit flag is set by a command from the base station, the motor telemetry streams are stopped and then all of the file handles, both for devices and disk, are closed before the program exits. This ensures a clean return to the command prompt at the end of execution.

For as long as the exit flag is not set, the main program will perform a loop similar to that seen for the previous MAVs. First, an input character is processed (if one exists). Then the sensors are checked and any updates are processed. If the LM345 has provided a new set of position and orientation data (this currently happens every 10ms), then a check occurs to determine if the robot is in fully autonomous flight mode. If not, flow continues normally. Otherwise, the autonomous program is evaluated to determine and apply any required updates to desired position. (The autonomous update would arguably fit more logically within the bounds of automatic-throttle (flight) mode, because that is the only time it can be applied. It is separated out because of its role at a level above the usual flight mode; the autonomous operator is a virtual replacement of the real operator at the base station.)

In the case that the base station has requested updates but the robot is not in flight mode, the update interval will be directly checked and, if required, current airframe status will be relayed over the wireless link. If the robot is in flight mode, position lock is evaluated first. If position lock is engaged, yaw is set to zero degrees and latitude, longitude, and height PIDs are evaluated according to whether they are enabled, or not. If latitude lock is enabled, desired pitch will be overridden autonomously; if longitude lock is enabled, roll will be overridden; if height lock is enabled, thrust will be automatically adjusted. If no locks are enabled, then the desired orientation remains under manual control as dictated by the base station. (As a side note, latitude and longitude locks will only override manual roll and pitch if the LM345 has a valid GPS lock.)

The orientation PIDs are calculated next to give desired airframe torques. Those torques are converted to motor angular velocities and those values (converted from radians/sec to RPM) are communicated over the serial interfaces to the motor controllers. After the motor updates are completed during flight, the base station timer interval is evaluated and status is relayed in accordance with programmed settings.

Every iteration of the main program loop for Jumbo QBot is ended by a thread sleep function call. That sleep time (currently 500µs) is necessary to ensure that the firmware program operating on the Gumstix does not dominate the processor and cause a backlog of operating system processing. The concern is that the operating system may then degrade processing consistency as it starts to schedule high priority tasks in the middle of the execution of critical Jumbo QBot operations. By behaving in a way that is considerate of other OS threads and interrupts, it is believed that the flight program will always see a consistent level of CPU access. This has worked well to date and the Gumstix processing is fast enough that the sleep time is inconsequential.

5.3. Base Station (Windows Application)

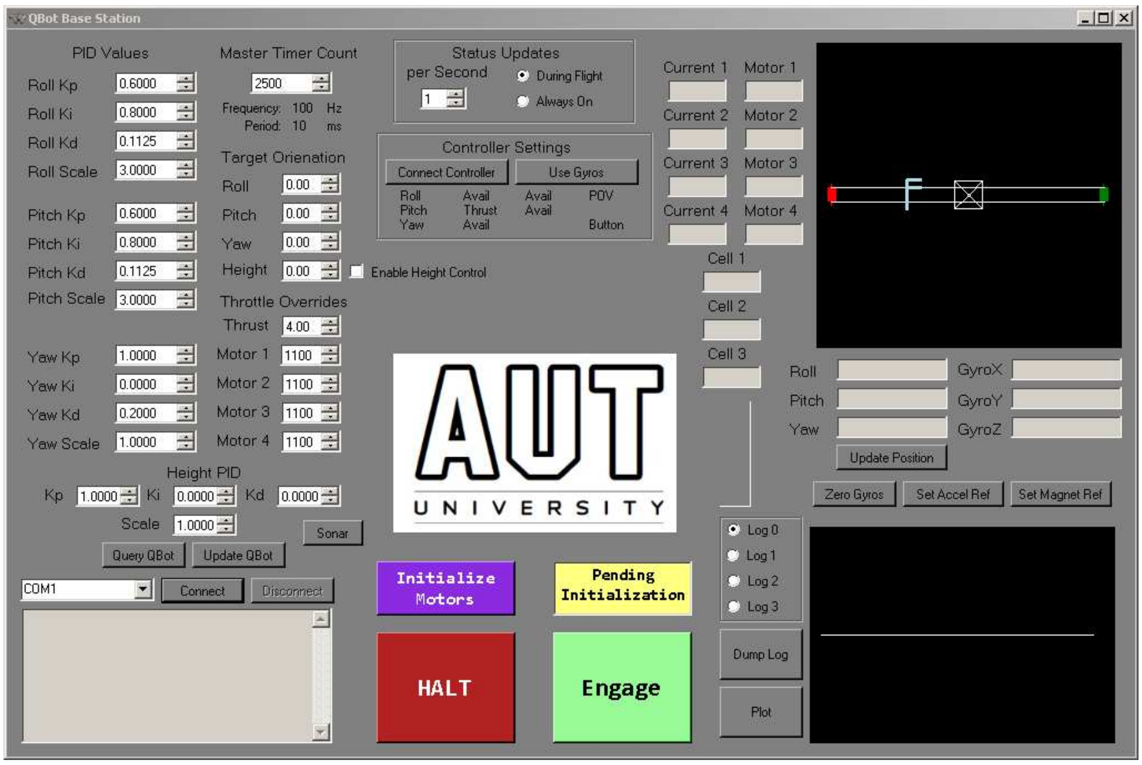
All of the base station software was designed in Microsoft Visual Studio 2010 and written in C#. Using that development suite for this research really highlighted the changing reality of GUI development for the modern world. Where a statement like ‘this code was written in C#’ used to be sufficient, a graphical application is now as much (or more) designed as written; a significant portion of the code is automatically generated by the drag-and-drop of design environment components onto a graphical framework. Adding a named event to a designed component then generates a functional stub in the source code to which event actions can be added, and it is at that point that a developer actually has to write something. As a result, the development process involves a combination of purely graphical design and conventional programming to yield a functional whole.

The base station application that was first developed for QBot1 was directly applicable to Araqnobot as well and they are now both controlled by the same software with no functional changes between them. Development for Jumbo QBot took that application as the starting point, but it eventually evolved into something more as the need for additional controls (for position hold, for example) and functionality was realized. The following section on the original base station will discuss the major components and functionality of the initial GUI application while the subsequent section focuses on the enhancements made for Jumbo QBot.

5.3.1. Controlling Application for QBot1 and Araqnobot

5.3.1.1. Main GUI

Figure 17 shows the GUI of the base station application developed for QBot1 and Araqnobot.



QBot Base Station

PID Values
Roll Kp 0.6000
Roll Ki 0.8000
Roll Kd 0.1125
Roll Scale 3.0000
Pitch Kp 0.6000
Pitch Ki 0.8000
Pitch Kd 0.1125
Pitch Scale 3.0000
Yaw Kp 1.0000
Yaw Ki 0.0000
Yaw Kd 0.2000
Yaw Scale 1.0000

Master Timer Count 2500
Frequency: 100 Hz
Period: 10 ms

Status Updates per Second 1
[✓ During Flight] Always On

Target Orienation
Roll 0.00
Pitch 0.00
Yaw 0.00
Height 0.00
Enable Height Control

Throttle Overrides
Thrust 4.00
Motor 1 1100
Motor 2 1100
Motor 3 1100
Motor 4 1100

Height PID
Kp 1.0000 Ki 0.0000 Kd 0.0000
Scale 1.0000

COM1

[✓ Log 0] Log 1 Log 2 Log 3

Figure 17 - Base Station GUI for QBot1 and Araqnobot

At the bottom left is a window for connecting the application to the serial port associated with the wireless transceiver (XBee). That window functions as a console that displays all messages received from the MAV and that sends any characters typed into it over the link. Most of the functionality of the GUI is disabled when no serial connection is active because the lack of serial connection implies the lack of a robot to receive information.

Along the left side of the GUI are all of the robot control variables that can be adjusted. All of the PID gains can be set here (the GUI includes height control terms that were never fully utilized and remain provisional), as can desired orientation, master timer period, thrust, and individual motor throttle settings (only usable when not in automatic flight mode). Some of these controls are dynamic (like thrust, orientation, and motor throttle settings) and updates are sent to the MAV as soon as they are changed. Other controls (like PID settings and master timer interval) require the ‘Update QBot’ button to be clicked before they will be collectively relayed. The ‘Query QBot’ button can be used to retrieve the current status of the control settings from the robot and the values retrieved will be placed in the appropriate GUI boxes (overriding the previous values). The sonar button will cause activation of an attached sonar module (if one exists) and the MAV will then include sensed height information in its status updates.

The centre of the GUI has status update control at the top and joystick interface settings beneath that. Much of the box labelled ‘Controller Settings’ is purely informational; it was created to monitor all of the available joystick controls (some of which are used and labelled accordingly) to determine what values were being seen by the software in response to certain movements and button presses. The most frequently used controller for the purposes of this research was a Playstation®3 DualShock® wireless controller that provides orientation and movement sensing in addition to conventional thumbstick input. The base station GUI then has a button (Use Gyros) to toggle between these input modes and thereby allow the robot to be controlled either by simply tilting the controller or by manipulating the thumbsticks (which is the default). Of course, the system can be operated without a controller and one must be actively connected (which is a software process, not a physical one) after communication with the MAV has been established. The default controller setting for this research has one thumbstick assigned to thrust, another one assigned to roll and pitch (with gyro input an alternative), and analog trigger buttons used for yaw.

Below the central AUT logo are a series of buttons and a macro status box (indicating ‘Pending Initialization’ in the Figure). The ‘Initialize Motors’ button will cause the MAV to execute its motor initialization routine and arm the motors. On initial startup, the MAV motor controls are not automatically configured (hence, ‘Pending Initialization’) and any command that might cause them to begin turning is essentially ignored. This provides a measure of safety that allows for initial sanity checking and clearance of the flight area before arming the motors. Once clicked, the MAV should indicate that initialization has been performed and the status box will turn green and indicate ‘Armed’. In that state, the MAV is ready to fly (or, alternatively, to have its motors controlled via the manual throttle settings). The ‘Halt’ button will set all of the motors to zero throttle and switch off the automatic throttle (if it was on). The ‘Engage’ button switches on the MAV’s automatic throttle and places it into full flight mode where it will begin using the PIDs to maintain desired orientation. When that occurs, the status box will switch to ‘Starting Motors’ as the MAV spins up the propellers and then to ‘Flying’ when the MAV hits desired thrust.

Informational output is the focus of the right side of the GUI. Text boxes are provided to relay instantaneous motor current and throttle status, battery cell levels, Euler angles, and angular rates. A graphical indicator of height is also included. Buttons are provided to assist with IMU calibration; the ‘Update Position’ button can be used to retrieve instantaneous values from the IMU at any time and can thereby be used to determine if any sensor drift or loss of reference has occurred. If the IMU does show calibration issues, the ‘Zero Gyros’, ‘Set Accel Ref’, and ‘Set Magnet Ref’ can be clicked to calibrate the corresponding internal sensor (assuming that the airframe is stationary, level, and pointing North, of course).

Four arrays to store logged variables from the MAV were incorporated into the GUI for post-flight analysis. The limited memory on QBot1 and Araqnobot only allows for about 5 seconds of full rate (100Hz) logging and those MAVs track roll, pitch, and yaw by default. When the ‘Dump Log’ button is clicked, the MAV will transmit its stored array and the base station software will then populate its internal values into the collection indicated by the Log 0-3 radio buttons. The Plot button will bring up a separate GUI window for data analysis, as described below in Section 5.3.1.2.

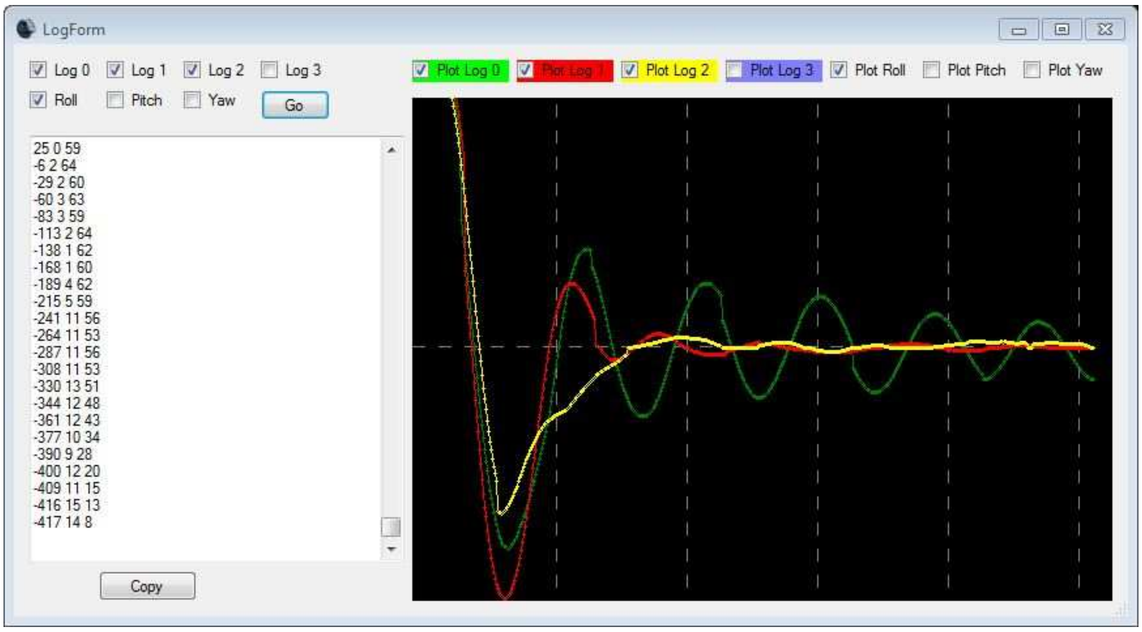
The two large black boxes in the right hand corners of the GUI are used for graphical display of dynamic status from the MAV. The bottom right box will display a plot of the Euler angles, each with its own line colour, as they vary over time. The top right box contains a graphical representation of the airframe that will tilt and turn in accordance with the indicated orientation from the robot. When the Euler angles are all zero, the airframe is depicted as pointing into the GUI with the observer standing directly behind. The coloured squares at the end of the arms are to indicate the port and starboard sides of the airframe while the ‘F’ in the image is located near the nose. These visual aids are necessary in an un-shaded symmetric wireframe model because the visual representation of an upside down or rotated object can be ambiguous (e.g. there is no difference in the 2D rendering of the model between facing directly away right side up and facing directly toward but upside down; the ‘F’ is what will indicate the inversion).

The code to properly render the 3D model is actually all based on Euler angles and Euler rotations and it provides an instructive correlation to the theory behind the real MAV orientation and the nature of its mathematical rotation in space. The software to accomplish it was all based around some basic Euler rotation code originally developed by a user called VCSKicks (VCSKicks, 2007). He provided source code for a simple C# project that would rotate a cube in response to sliders that each represented one Euler angle of rotation. For the purpose of building the base station GUI for this research, it formed the starting point, and so the C# namespace for the final application is still EulerRotation as defined by VCSKicks in his code.

5.3.1.2. Log Analysis GUI

When the "Plot" button is clicked on the main GUI, a subordinate window opens. When it first appears, none of the checkboxes will be selected and the text and plot windows will be empty. Figure 18 shows the window as it appears after some options are selected. The plot window on the right hand side is used for quick analysis and comparison of the telemetry recorded from the logged flights. Individual logs can be selected for rendering and any combination of roll, pitch, and yaw can be displayed. The horizontal dashed line on the display is placed at zero degrees while the vertical dashed lines mark every 100th data point (corresponding to 1 second at 100 Hz). Using this facility greatly enhanced the tuning process for the orientation control parameters.

The text box on the left can be used to selectively extract portions of the recorded information for transfer to other programs (e.g. spreadsheet). Again, the logs can be selected independently as can the Euler angles. When the 'Go' button is clicked, the text box will be populated with the requested information and it can all be copied to the clipboard by pressing the "Copy" button. When pasted into spreadsheet software as space delimited data, it will be allocated into rows and columns that can be analysed further.



```
25 0 59
-6 2 64
-29 2 60
-60 3 63
-83 3 59
-113 2 64
-138 1 62
-168 1 60
-189 4 62
-215 5 59
-241 11 56
-264 11 53
-287 11 56
-308 11 53
-330 13 51
-344 12 48
-361 12 43
-377 10 34
-390 9 28
-400 12 20
-409 11 15
-416 15 13
-417 14 8
```

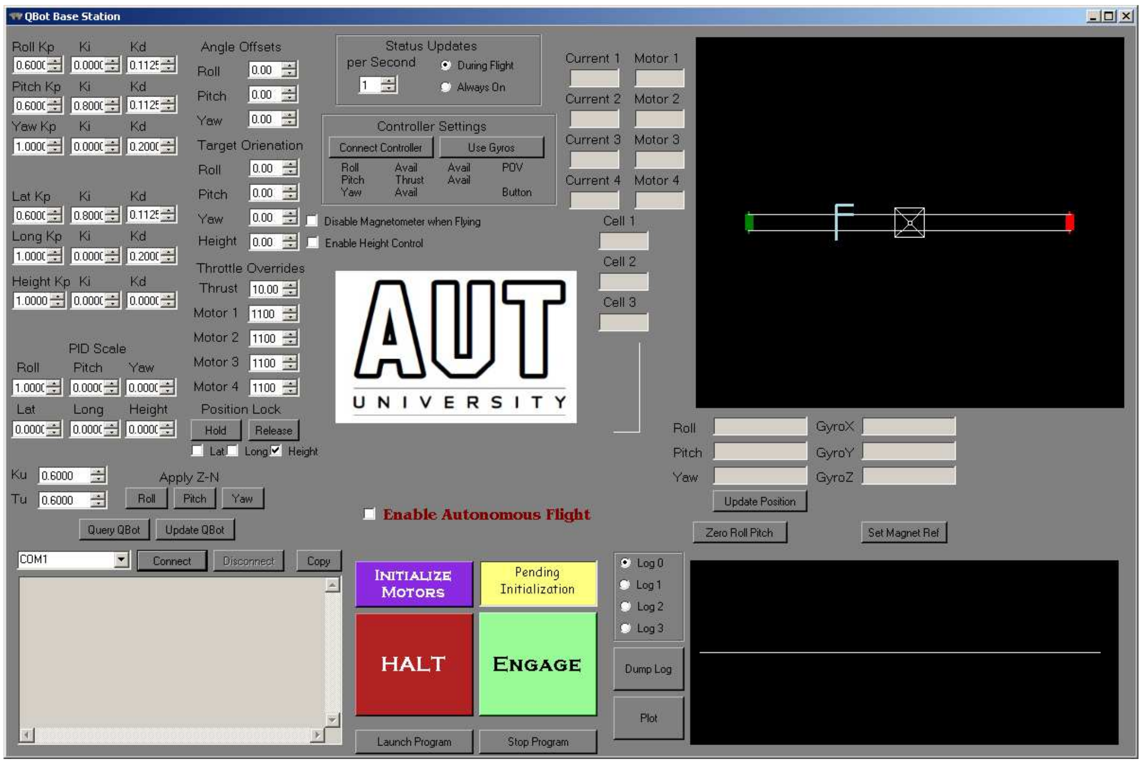
Figure 18 - Base Station Log Analysis GUI for QBot1 and Araqnobot

5.3.2. Controlling Application for Jumbo QBot

5.3.2.1. Main GUI

Figure 19 shows the modified base station application as it was used for Jumbo QBot development and operation. For the most part, it is nearly identical to the original version and only significant differences will be described here. The most obvious change is the addition of many more control parameters. This reflects the positional capability that was added to Jumbo QBot which required the addition of PID gain factors for latitude, longitude, and height. To aid with tuning, a group of controls were added that enabled the quick calculation and application of Ziegler-Nichols gains (discussed in Section 3.2.1.1) to the roll, pitch, and yaw PIDs. Selection checkboxes were added for the position lock degrees of freedom and then buttons were created to control the enabling and disabling of those locks. A series of explicit angle offsets was created to compensate for drift issues (largely due to flexing of the airframe when in flight versus on the ground) and allow for manual tuning of the hover attitude (effectively the same as trim compensation). The ‘Disable Magnetometer when Flying’ checkbox was intended to trigger that function on the MAV (for use when flying near sources of significant magnetic interference), but it hasn’t been fully tested and remains largely provisional.

Possibly the most significant addition (in conceptual terms, at least), is the ‘Enable Autonomous Flight’ checkbox. When that is checked and position lock is enabled, the MAV will begin executing an internal program to automatically adjust the desired position relative to the starting point (defined as the position of the aircraft at the point that position lock is enabled). In this way, the robot is able to execute a fully autonomous flight plan and the base station operator no longer has any direct input to its behaviour (control can always be returned to manual be disabling position lock).


QBot Base Station

	Kp	Ki	Kd
Roll	0.600(	0.000(	0.112
Pitch	0.600(	0.800(	0.112
Yaw	1.000(	0.000(	0.200(
Lat	0.600(	0.800(	0.112
Long	1.000(	0.000(	0.200(
Height	1.0000	0.000(	0.000(

PID Scale

Roll	Pitch	Yaw
1.000(	0.000(	0.000(

Lat	Long	Height
0.000(	0.000(	0.000(

Ku 0.6000
Tu 0.6000

Angle Offsets: Roll 0.00, Pitch 0.00, Yaw 0.00
Target Orienation: Roll 0.00, Pitch 0.00, Yaw 0.00, Height 0.00
Throttle Overrides: Thrust 10.00, Motor 1 1100, Motor 2 1100, Motor 3 1100, Motor 4 1100
Position Lock: [ ] Lat [ ] Long [✓] Height
Status Updates per Second: 1 — (●) During Flight ( ) Always On
[ ] Disable Magnetometer when Flying
[ ] Enable Height Control
[ ] Enable Autonomous Flight
COM1
(●) Log 0 ( ) Log 1 ( ) Log 2 ( ) Log 3

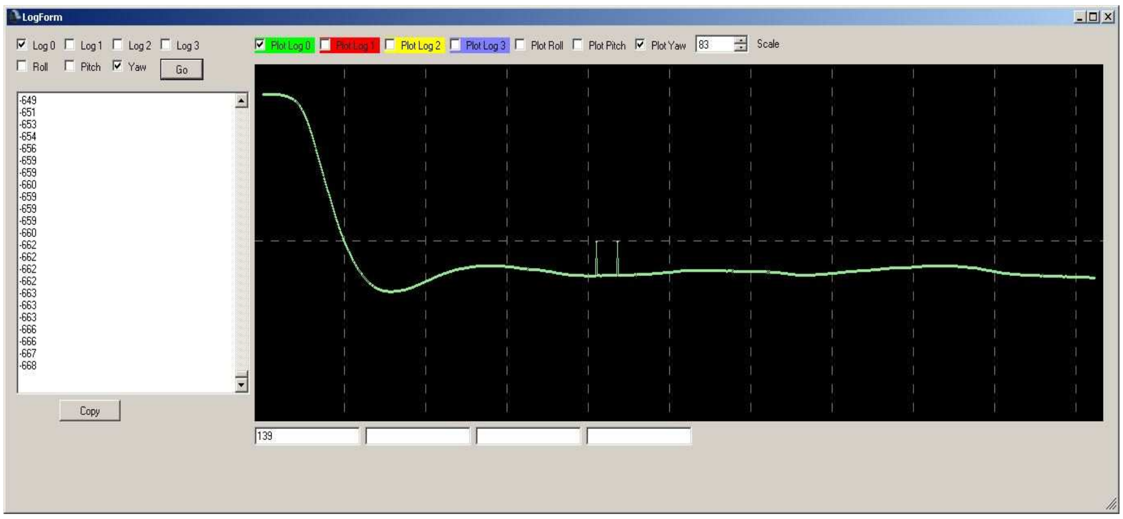
Figure 19 - Base Station Main GUI for Jumbo QBot

The two buttons located at the middle bottom of the GUI are provided to allow the launching of the MAV program from the Linux command line (on boot, Jumbo QBot presents a console terminal over the wireless interface) and the termination of execution (to return to the command prompt).

Finally, the LM345 orientation and position sensor has different capabilities than the IMU that was used for QBot1 and Araqnobot, so the ‘Zero Gyros’ and ‘Set Accel Ref’ buttons were removed. The added ‘Zero Roll Pitch’ button will cause the firmware on the robot to automatically work out the angle offsets for roll and pitch. This means that the orientation of the robot at the moment that button is clicked will be computed as zero thereafter (similar to ‘Set Accel Ref’ for the previous MAVs).

5.3.2.2.   Log Analysis GUI

The log analysis GUI for Jumbo QBot represents only an incremental evolution from the original design. It is shown in Figure 20. The plot window was expanded to accommodate the larger amount of data that can be tracked and relayed by Jumbo QBot (which thereby allows more extended analysis). All of the same buttons and checkboxes are available in the new design and only a couple of features have been added.



```
-649
-651
-653
-654
-656
-659
-659
-660
-659
-659
-659
-660
-662
-662
-662
-662
-663
-663
-663
-666
-666
-667
-668
```

Scale: 83

139

Figure 20 – Base Station Log Analysis GUI for Jumbo QBot

First is a ‘Scale’ number box that can be used to increase or decrease the magnitude of the plotted data points. Its function is a little different than normal scaling functions of this sort; rather than being a multiplier, it is a zoom factor subtracted from 100 and used as a divisor for the data points. A value in the zoom box of 99 will cause the data points to be rendered at unity (divide by 1); a value of 0 will cause each to be divided by 100 before rendering.

The second added feature is the series of text boxes underneath the plot window. Those boxes will be populated with the average max-to-min and min-to-max interval for each of the log arrays for whatever angle is currently selected (e.g. if roll is selected, the first box will contain a calculation of the average number of entries between the successive inflection points for the roll data in log 1; the second box will contain the calculated value for log 2, and so on). This provides a quick way to evaluate the oscillation period for a given waveform (which is useful for PID tuning).

6. Experimental Results

The firmware and software developed as part of this research are, unsurprisingly, the end result of a vast amount of testing and iteration. Despite the apparent simplicity, many things were tried and explored in the process of developing the operational systems that were produced. Some sensors worked in different ways than expected, communication issues were encountered, and even one compiler bug caused issues. Beyond the pitfalls of software and hardware development, extensive tuning of control parameters was required. All of this was done on physical systems using real data and the details of the process are presented below.

6.1. QBot1

The first goal defined for this research was to achieve orientation control of the basic QBot1 airframe on a 3 degree-of-freedom (DOF) test stand. Before the theory of operation could be applied and tested, however, each combination of motor, motor controller, and propeller required characterization.

6.1.1. Motor Testing

The theoretical formulas for orientation control rely on being able to set each motor to a known rotational velocity. Because the motor controllers for QBot1 had no automatic calibration capability, each one had to be tested to work out the relationship between PWM (throttle setting) and RPM (which can be directly converted to angular velocity). This was accomplished through the use of a PowerLog 6S device (made by Shenzen Junsi Electronic Co., LTD.) that uses a detector to count interruptions in detected light as they are caused by a spinning rotor. The detector is mounted below the spinning blade and a steady light source (not fluorescent) is positioned above. Then pulse width and RPM are logged as the throttle is swept through its operational range.

As an example, the data for motor 1 was recorded as shown in Table 10:

Table 10 - RPM to PWM Values for QBot1 Motor1

RPM [rpm]	Pos. Pulse Width [µs]
960	1200
1230	1220

RPM [rpm]	**Pos. Pulse Width [μs]**
1365	1240
1590	1260
1725	1280
1920	1300
2040	1320
2100	1340
2340	1360
2410	1380
2478	1400
2760	1420
2840	1440
3060	1460
3140	1480
3480	1500
3530	1520
3780	1540
3870	1560
3937	1580
4170	1600
4275	1620
4350	1640
4590	1660
4640	1680
4830	1700
4935	1720
4995	1740
5190	1760
5310	1780
5370	1800
5520	1820
5625	1840

Overall, that data is remarkably linear and indicates the quality of the motor and motor controller over the operational range. It also helps to justify the assumption of motor linearity for firmware calculations. With the data above, linearization parameters can be easily extracted. The slope can be calculated by taking the RPM delta and dividing by the pulse width delta: (5625 – 960) / (1840 – 1200) = 7.289 RPM/µs. This would be good enough, but a better value was desired (or, at least, an appreciation of the quality of this number was desired).

The approach to a better solution involved using a spreadsheet to first correlate slope to effective offset; namely the value that needs to be subtracted from pulse width * slope to give the expected RPM value. Because the airframe was intended to operate near hover thrust, only those values central to the operating range were used: 1920- 4830 RPM. The offset given by ‘offset = pulse width * slope – RPM’ was calculated for each data point and then the average was used for the constant offset. Those values were then used to convert the pulse width data points back to RPM (linearized RPM = pulse width * slope – constant offset) and the absolute difference between the real RPM and the linearized RPM was placed in a column of its own. The sum of the differences across the selected operational range was then used as a measure of quality of the linear approximation. An iterative process of adjusting the slope was employed until a minimum summed deviation was achieved. In this case, a slope value of 7.36 yielded the minimum total deviation (and 0.01 plus or minus made very little difference) and that value was selected for this motor and controller combination. The corresponding offset was calculated to be 7680.95. A plot of the original motor data versus the linearized data is shown in Figure 21.

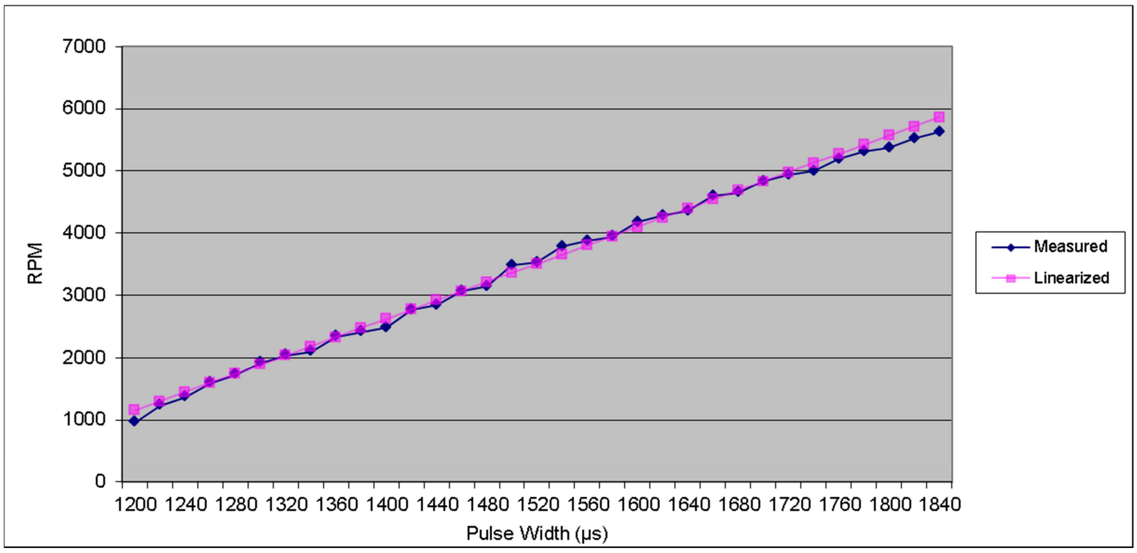


Figure 21 - QBot1 Motor1 Linearization Plot

RPM values are significant to motor manufacturers and are easily understood in conversation, but the theoretical formulas for this research require angular velocity in rad/s. Furthermore, the internal PWM values for QBot1 had ½ µs precision (meaning that a 1200µs pulse width had an internal representation of 2400) and the firmware solves for PWM pulse width from desired angular velocity, not the other way around. So then, a slope of 7.36 RPM per µs converts to 2.595 ½ µs per rad/s while the offset of 7680.95 RPM (804 rad/s) converts to a PWM offset of 2086 ½ us (i.e. a pulse width of 1.043 ms would correspond to 0 rad/s). The same process was applied to the other three motor & controller sets and the final values for slopes and offsets are given in Table 11.

Table 11 - QBot1 Motor Characterization Parameters

Motor	Slope (½ µs per rad/s)	Offset (½ us)
1	2.595	2086
2	2.709	2116
3	2.938	2300
4	2.616	2106

6.1.2. Orientation Control

6.1.2.1. Single Axis PID Tuning

With the motors characterized, it was possible to begin testing the control theory. An initial attempt was made to move directly to the 3-DOF test stand. That proved overly ambitious as there were too many variables involved and it was difficult to detect improvements (if there were any) amidst the general chaos. It was determined that the best approach would then be to limit the airframe to a single DOF at a time and demonstrate control for roll and pitch separately before attempting them altogether again.

Because QBot1 had round arms, a clamping system was devised for this research that holds two arms of the airframe between a set of roller bearings. The roller bearings allow unrestricted rotation about the axis held by the clamps while preventing rotation about the other axes. The roller-clamp concept was developed as part of this research, but the physical design and creation of the clamps was performed by a mechanical engineering technician at AUT. One of the clamps is shown disassembled in Figure 22.

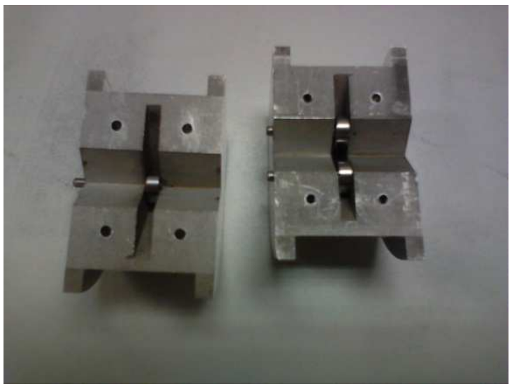


Figure 22 - Roller clamp used for single-axis testing

Even with only a single allowed axis of rotation, tuning the PID parameters proved a very challenging task. A lack of experience in this type of control system tuning no doubt contributed to the difficulty, but this system certainly exhibited a significant degree of complexity beyond a conventional machine with a first order type of response.

6.1.2.1.1. Experimental PID Tuning

The first PID tuning effort of this research employed simple guesswork and evaluation of system response to changing parameters. An example progression of roll testing is shown by the three figures below. In each, the horizontal axis is simply the sample number (100 per second) while the vertical axis represents the raw IMU output for the roll Euler angle (the UM6 IMU outputs 16-bit integers such that 16384 is equivalent to π radians (180 degrees); for simplicity, it can be considered that every 1000 steps corresponds roughly to 10 degrees). Figure 23 depicts the system response when the PID gains are set to 1, 2, and 0.2, respectively. Figure 24 shows the system response when the derivative gain constant is increased to 0.25 while Figure 25 shows the response to a derivative decrease to 0.18.

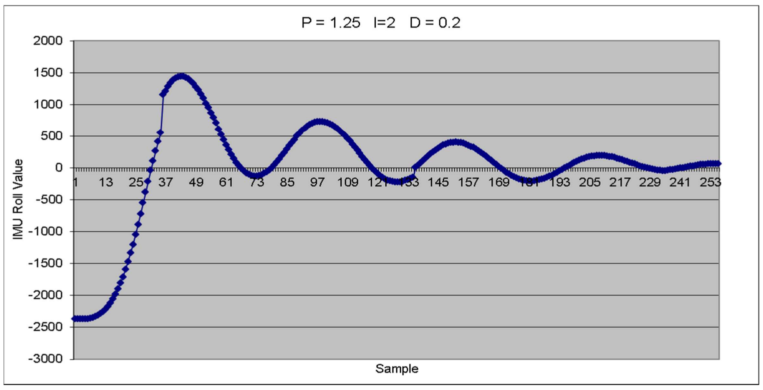


Figure 23 - Roll PID Experimentation K_p=1.25, K_i=2, K_d=0.2

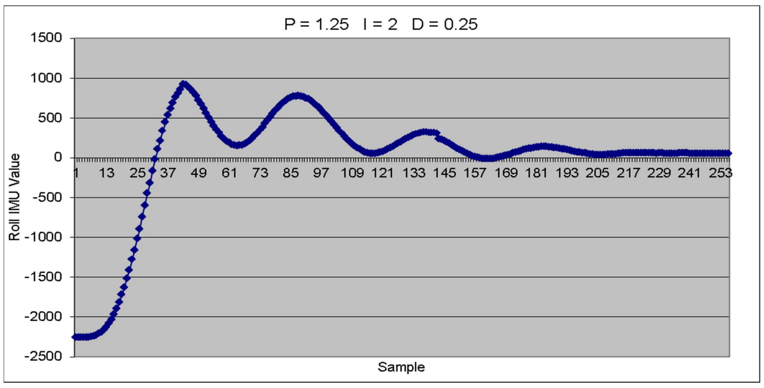


Figure 24 - Roll PID Experimentation $K_{p=}$1.25, K_i=2, K_d=0.25

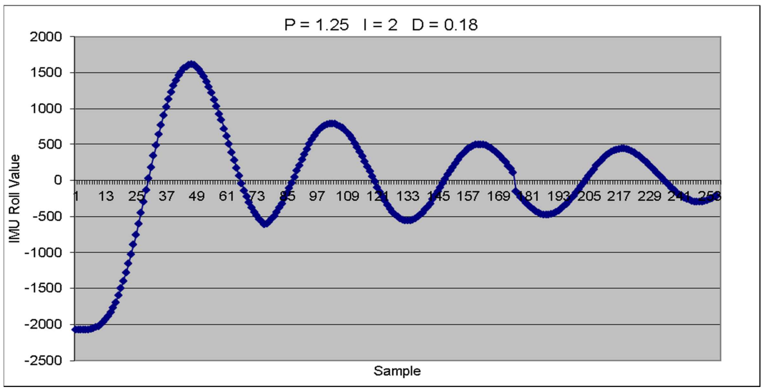


Figure 25 - Roll PID Experimentation K_p=1.25, K_i=2, K_d=0.18

At a high-level, these results are unsurprising as they reflect well known system response to PID derivative gain changes. When the derivative gain is increased, it can begin to over-damp and cause the system to approach the setpoint in steps rather than directly. When decreased, it can reach a point where it has little effect at all and the other terms dominate (in this case resulting in ongoing oscillation).

At a lower-level, the system response to a high-value derivative gain (Figure 24) warrants a bit more analysis. After initially crossing the setpoint, the approach back to it is taken in steps, as might be expected, but the normal derivative action is usually limited to reducing or stopping the velocity toward the setpoint (because the derivative term must go to zero when velocity is zero), not reversing it as is seen. The reversal could, of course, be attributed to integral windup that occurred prior to the first crossing and that is likely the case to a significant degree. Another contributor, however, could be the system delays and it may be that the detected velocity and corresponding response to it are time-shifted by a sufficient amount to cause acceleration overshoot.

6.1.2.1.2. Modified Ziegler-Nichols Tuning

It was at this point in testing that the weakness of a purely manual approach to tuning was appearing inefficient (at best) and an alternative was sought. The initial framework for the modified approach to Z-N tuning (discussed in Section 3.2.1.1) was applied and values for the Z-N tuning parameters, K_u and $T_{u,}$ were sought.

In the case of roll testing for QBot1, a proportional gain value of 1.0 gave a reasonable measure of system response and, while still ultimately unstable, oscillations remained fairly consistent for several cycles. The cycle period was seen to be at least 1 second and that became the first approximation. The Z-N Tuning PID equations ((12), (13), and (14)) were then applied to give $K_p = 0.6$, $K_i = 1.2$, and $K_d = 0.075$. With these parameters, the system achieved stable hold at a roll angle of zero, but it didn’t seem very well tuned. As a result, the approximated Z-N tuning values were evaluated experimentally using a number of different values for each. First, the K_u value was tested and the system response was captured for values of 0.1, 0.5, 1.0 (the original approximation), and 1.5. The plotted results are shown in Figure 26.

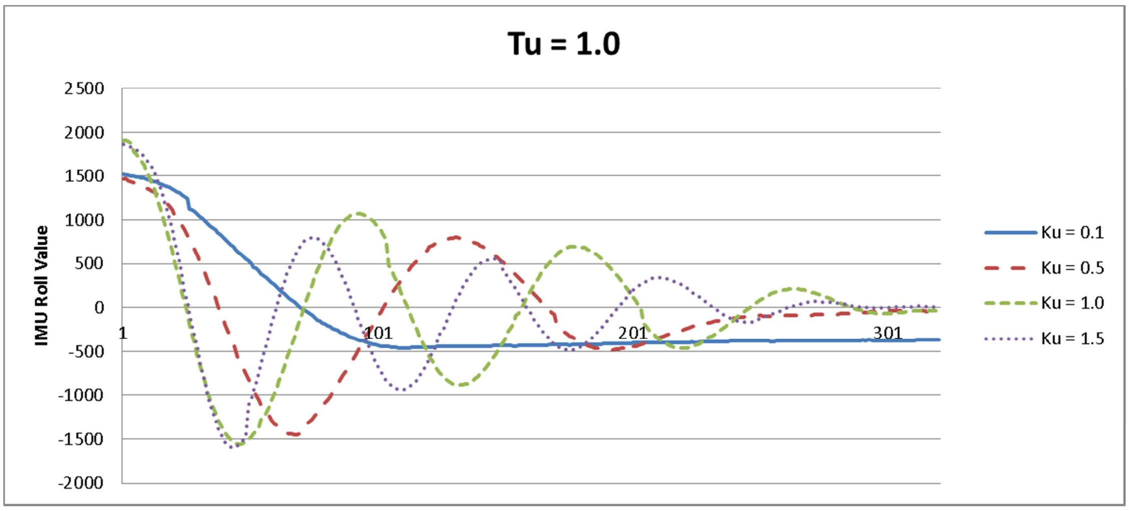


Figure 26 - QBot1 Z-N Tuning for Roll (T_u = 1.0)

It is apparent that a K_u of 0.1 is insufficient, and while the other plots do reach the setpoint and are stable there, none of them exhibit a response that would be considered well-tuned. As an observer of the physical system, it did seem that the K_u value of 1.0 was the most robust, but that was a subjective assessment only. It needed to be seen if that value could provide a reasonably tuned output with other values of T_u.

Setting K_u to 1.0 and sweeping T_u through several values was the next test undertaken. The values chosen were 0.5, 1.0, 1.5, and 2.0 and the plotted results are shown in Figure 27.

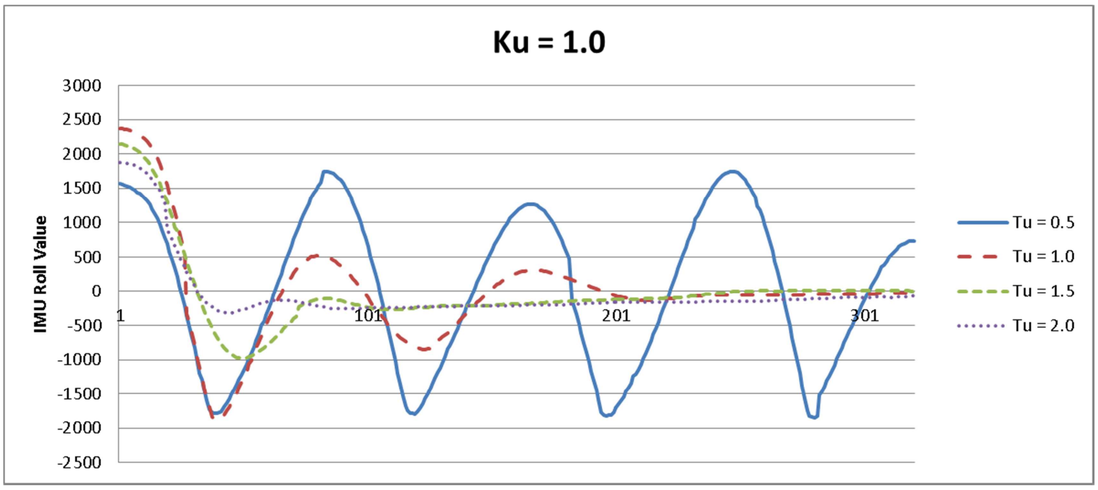


Figure 27 - QBot1 Z-N Tuning for Roll (K_u = 1.0)

A T_u value of 0.5 was obviously unstable, 1.0 (as previously tested) showed slow convergence, 1.5 and 2.0 both exhibited good system response. T_u = 1.5 exhibited more overshoot than 2.0, but it was quicker to achieve the setpoint and the system response waveform is very similar to that expected of a system tuned using the Z-N tuning method. Furthermore, it was very stable in the roll testing apparatus. As a result, the values of $T_u = 1.5$ and $K_u = 1.0$ were selected as reasonable approximations of the Z-N tuning parameters and the corresponding calculated PID gain values for application on the MAV were: $K_p = 0.6$, $K_i = 0.8$, $K_d = 0.1125$.

The same process was applied to pitch testing to ensure there were no major differences between the two axes. Although the intent of a quadrotor airframe design is to keep everything symmetrical about the centre, there are necessarily some deviations. For QBot1, the major difference between roll and pitch came from the orientation of its battery. The battery represents the largest single component of mass in the system and it is rectangular. It was aligned lengthwise with the x axis and thereby created a significantly higher moment of inertia for pitch than existed for roll. Because the moment of inertia is inversely proportional to the amount of acceleration caused by torque (which is the selected PID output for orientation control), it was possible that it would significantly impact the system response and corresponding tuning.

As with roll testing, the pitch testing exhibited similar instability and difficulties in achieving consistent oscillation. A K_u value of 1.0 was, however, once again selected as yielding something closest to what is desired for Z-N tuning. In this case, the observed system response had an approximated period of oscillation near 1.5 seconds. As those were the same values determined for roll, they made a reasonable starting point, but the same sweeping of T_u and K_u were employed again to double-check the approximations for improvement. The results are shown in Figure 28 (K_u sweep) and Figure 29 (T_u sweep).

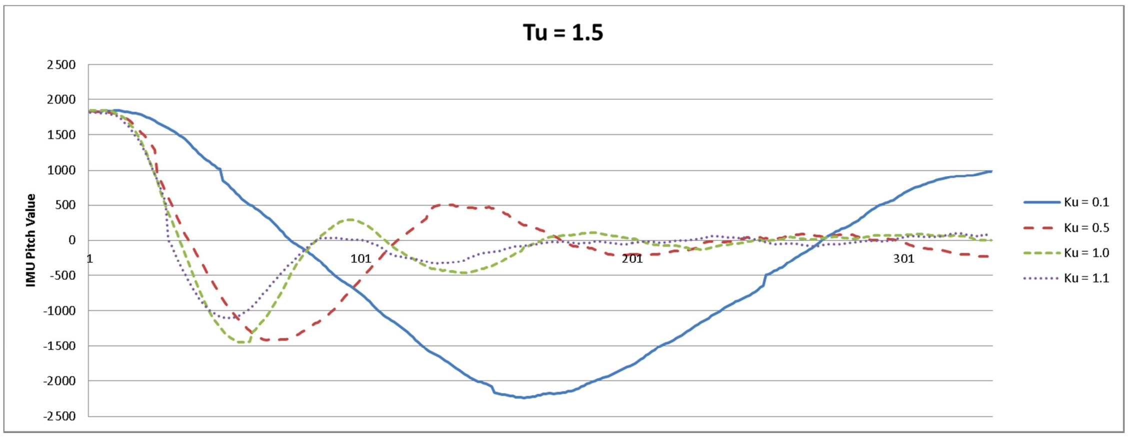


Figure 28 - QBot1 Z-N Tuning for Pitch ($T_u = 1.5$)

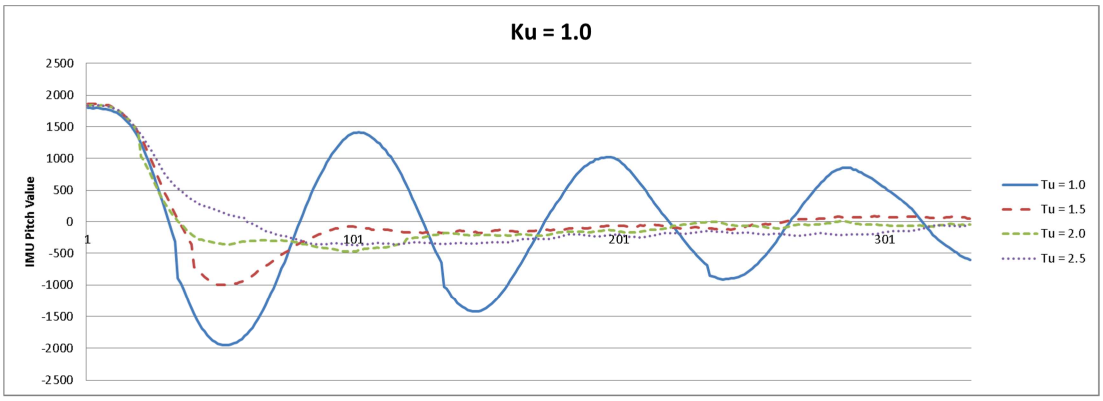


Figure 29 - QBot1 Z-N Tuning for Pitch (K_u = 1.0)

These results demonstrate that the approximations of T_u = 1.5 and K_u = 1.0 are reasonable for Z-N tuning of the pitch PID. The corresponding K_p, K_i, and K_d terms are the same as were calculated for roll: 0.6, 0.8, and 0.1125, respectively.

6.1.2.2. Three-Axis Testing

With pitch and roll orientation control working well independently, the QBot1 airframe was moved from the single DOF testing apparatus to the 3 DOF of freedom test stand. The assembled apparatus with QBot1 on it is shown in Figure 30.

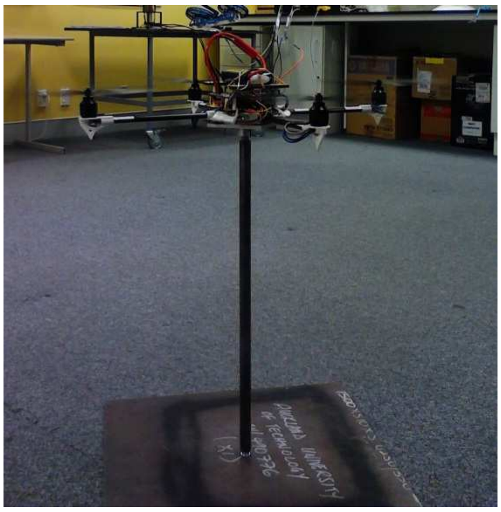


Figure 30 - QBot1 on 3DOF Test Stand

The tuned parameters worked perfectly and the robot was able to robustly hold itself in hover attitude. Induced disturbances (e.g. pulling on an arm by hand) were strongly rejected and the return to hover was consistently very quick and clean (little overshoot, no steps along the way).

It should be noted that the test stand causes a noticeable impact on the system and isn't truly representative of free flight. For PID response, it actually represents a worse-case scenario for several reasons:

1. The centre of mass and the pivot point are displaced from one another
2. The mounting plate effectively becomes an added element of mass to the airframe
3. The stand adds resistance to orientation changes due to resistance of the supporting bearing

With good bearing selection and lightweight mounting hardware, the first point is arguably the only one of significance. To expand on that in the case of the test stand shown in Figure 30, the airframe can only be rotated on the roll and pitch axes about the bearing at the bottom of the mounting plate. When the robot is in hover attitude, the difference in pivot point is hardly noticeable. When the robot leaves the level position, however, the force needed to restore it is significantly greater than would be needed for operation in free flight. This is exacerbated by mounting the battery above the airframe as was done for these initial tests. When out of level it is no longer just the moment of inertia that must be overcome to rotate the airframe, but also some portion of gravity; the airframe must effectively be lifted to return (arrive) at a hover position. These effects had been encountered during preliminary testing before application of the tuning process described above and it was seen that ‘takeoff’ represented the greatest challenge on a test stand like the one shown. After tuning, that startup process was no problem and the PID parameters were sufficient to bring the airframe from its idle position well away from level to hover attitude with alacrity.

Roll and pitch position control was therefore deemed successful. Not much testing or tuning went into yaw control. A couple of values were tested and it seemed that a PD approach was sufficient to reliably bring the airframe to a desired directional setting. Because of the relatively long times involved with yaw compensation (it takes a couple of seconds to spin the airframe 180 degrees), adding the integral term degraded system stability and it was removed (anti-windup would, perhaps, have been a better solution). In the end, using experimentally determined PID gain terms of 1.0, 0, and 0.2, respectively, gave desired system response and full orientation control was achieved.

6.1.2.3. Tethered Flight

Having achieved a stable hover attitude on the 3 DOF test stand, the next experiments involved actually flying the robot. The first issue encountered was that significantly more thrust was required to lift the MAV off the ground than had been required to hold it “hovering” on the stand and that resulted in excessive vibration of the motor mounts. The original design for QBot1 used lightweight wireframe style supports for the motor attachment points (seen on the left of Figure 31) that simply had too much flex when the motors were running at higher speeds. Two mounts broke and they needed to be redesigned; the new mounts (right of Figure 31) performed much better.

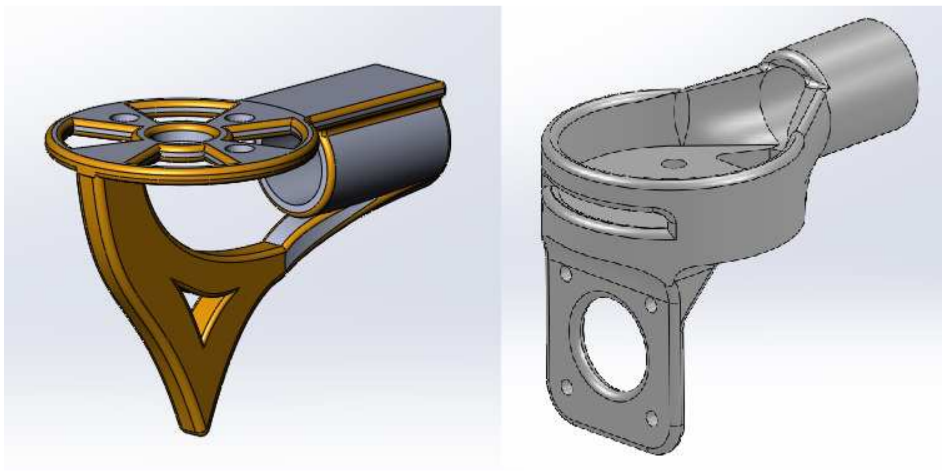


Figure 31 - QBot1 Motor Mount Comparison

At this point, the Base Station software was still in development and no controls for orientation adjustment had been implemented. The only operational control was thrust and that meant that QBot1 had to be constrained to prevent it drifting into other objects. This was accomplished by using lengths of string attached to a large square frame on the ground. The string lengths were set to allow the airframe to move about a half meter in any direction from the centre of the square. To achieve flight, the motors were started and thrust was increased until the airframe left the ground. The PID tuning parameters continued to work well and stable tethered flight was easily reproduced.

6.2. Araqnobot

Having achieved success with QBot1, the focus of this research shifted to assembling and testing Araqnobot with the intention of first repeating the approach employed to tune QBot1. The desired end goal for this platform was subsequently untethered flight with autonomous orientation hold and operator position compensation (i.e. manual thrust control and obstacle avoidance).

6.2.1. Motor Testing

Because it had been fitted with the new ESC32 motor controllers, the calibration approach for Araqnobot was significantly different from QBot1. The developers of the ESC32 device have created automatic calibration routines and have implemented closed loop modes of operation that provide internal linearization of response and a guard against overdrawing current. The process of calibration consists of the following steps:

1. Install the desired propeller on the motor
2. Fix the motor in place so that it cannot move
3. Connect the motor controller
4. Power up the motor and controller
5. Connect the motor controller to a serial port on a PC
6. Run the developer's calibration routines
7. Program the calibration parameters in the controller's non-volatile memory

Two calibration routines are provided: the first determines the motor's RPM achieved in response to voltage applied (motor speed vs. throttle) while the second evaluates the current draw seen by sudden changes in voltage (dynamic current draw from acceleration).

6.2.1.1. Voltage to RPM Calibration

To determine RPM vs. voltage, the calibration routine simply sweeps through the throttle settings (which are equivalent to applied voltage and duty cycle (at 100%, the full voltage of the battery is being applied)). For Araqnobot's motor 1, the collected data and corresponding calibration terms are shown in Figure 32 (which was automatically generated by the calibration software).

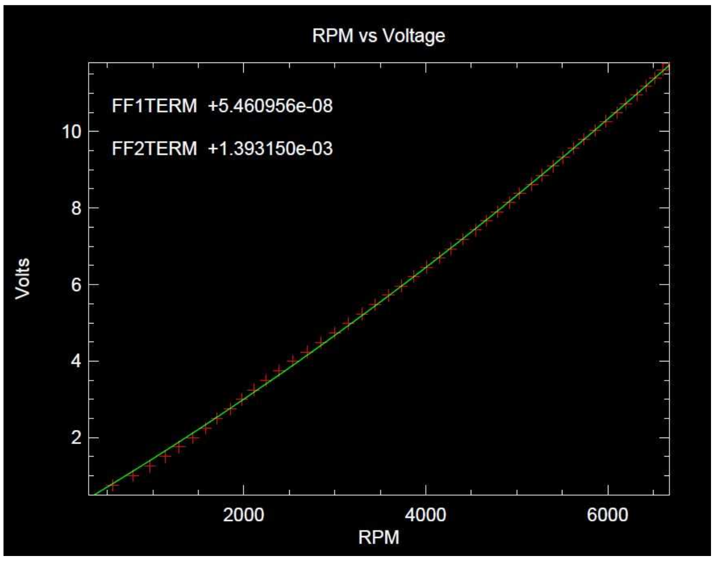


Figure 32 - Araqnobot Motor 1 RPM vs Voltage Calibration

This effectively takes the place of all of the manual characterization that was done for QBot1. The motor controller has worked out a best-fit linearization that will be internally utilized. When the ESC32 is programmed with the FF1TERM and FF2TERM parameters that have been calculated, it can be placed in closed-loop mode. At that point, the PWM inputs are no longer reflective of throttle value (0 to 100%), but rather of desired RPM value. The exact determination of the value desired is defined by the PWM_LO_VALUE, PWM_HI_VALUE, and PWM_SCALE parameters. These specify the linearized response of the motor, and this is the description given in the ESC32 datasheet table entry for PWM_SCALE: “The scale of the input PWM pulse length. In closed loop RPM run mode, PWM_LO_VALUE will indicate 0 RPM and PWM_HI_VALUE will indicate this RPM.” (AutoQuad, 2012, p. 11)

For Araqnobot, all of the motor controllers were set to have PWM_HI_VALUE of 2000 (2ms), PWM_LO_VALUE of 1000 (1ms), and PWM_SCALE of 6000 (RPM). As a result, a 1ms pulse width is expected to give 0 RPM, 2ms will yield 6000 RPM, and everything in between is linear (e.g. 1.5ms pulse is expected to give 3000 RPM). To make direct use of this relationship, the rotation to PWM conversion logic for Araqnobot was changed to use a static PWM value of 2000 ½ μs and then each rad/s of desired rotational velocity added another 3.1831 ½ μs.

6.2.1.2. *Current Limit Calibration*

The ESC32 current limit calibration routine works by iterating through duty cycle steps and tracking dynamic motor current draw (e.g. it steps from 5% throttle to 10%, then 5% to 15%, then 5% to 20%, and so on). BLDC motors generally draw peak current on motor acceleration rather than at a static setpoint (e.g. a jump from 5% to 100% will draw significantly more instantaneous current than static 100%). A quadrotor MAV generally exhibits continual acceleration and deceleration as the orientation logic is updated and the PIDs are applied; by limiting the dynamic current, efficiency can be increased and overheating can be avoided. The intent of the calibration, then, is for the motor controller to work out terms for its acceleration logic that will keep the dynamic current within the defined bounds (i.e. significant throttle increases can be executed in stages rather than all at once).

For the motors on Araqnobot, the current limit was set to 9 amps and the routine was executed. The results for motor 1, as generated by the software, are shown in the graph of Figure 33. A full explanation of the graphed results is beyond the scope of this research (the output values and corresponding expected operation were sufficient), but it is believed that the dots represent real sampled values while the lines superimposed reflect the applied software terms. As an example of analysis, consider the horizontal line of dots around 6 Volts; the leftmost values represent increased current draw as the motor is spinning less than it would at a constant value around 6V (it is therefore accelerating). The rightmost values are then more abundant because they indicate the more natural speed of the motor in response to an applied 6V (and correspondingly reduced steady-state current draw).

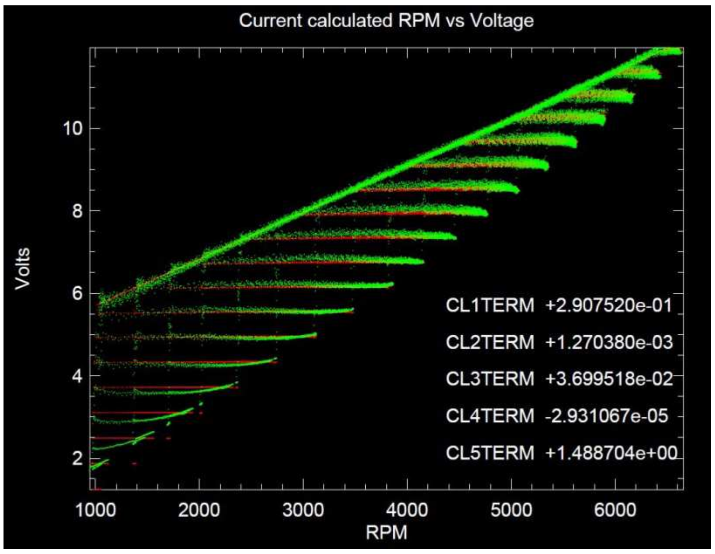


Figure 33 - Araqnobot Motor 1 Current Limit Calibration

Each motor underwent the same calibration and the CL1-CL5TERM values were programmed into the respective controllers.

6.2.2. Orientation Control

6.2.2.1. Single Axis PID tuning

Having a new airframe allowed an opportunity to test and refine the approach that was successful on QBot1. The round carbon fibre arms on Araqnobot allowed them to be clamped into the same roller bearing apparatus that had previously been employed. One aspect of testing that was changed was the amount of thrust applied on the test stand. For QBot1, only a fraction of lift thrust had been used on the stand because that was all that was needed to stabilize the airframe. It was later recognized, however, that it would be more accurate to perform tuning near the thrust level required for flight. The interesting thing about this approach is that it provided more consistency in system response for the Z-N tuning method. With only a P-term applied, a certain amount of system stability could be achieved and somewhat consistent oscillations could be recorded, as long as the deviation from the setpoint was minimized. This allowed a more conventional Z-N analysis and the output of several P-only iterations for pitch control is shown in Figure 34.

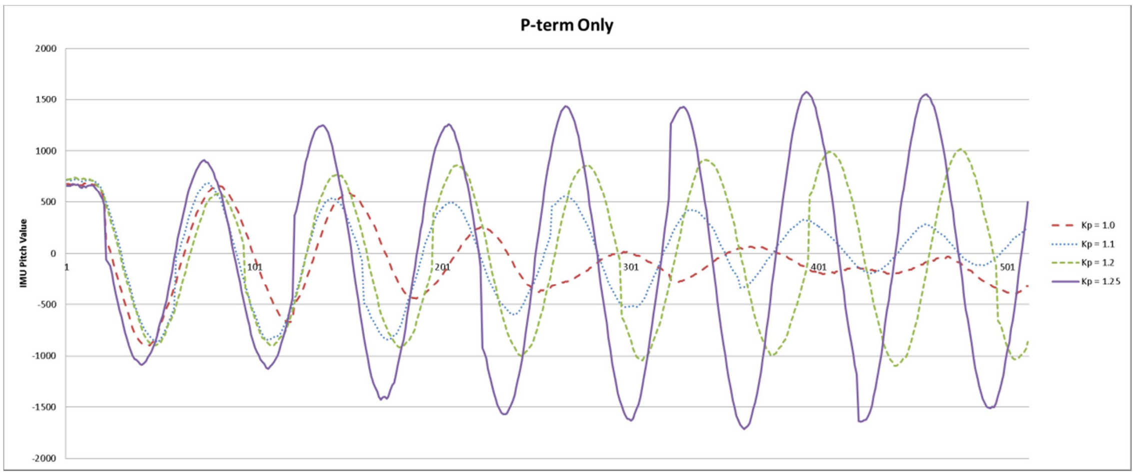


Figure 34 - Araqnobot Pitch Testing - P-term only

This data demonstrates that the system can give fairly stable oscillations with a K-term gain around 1.2. Although the amplitude appears to increase in the first 5 seconds of operation (the data capture window), it did stabilize and eventually the oscillations even diminished. Increasing the gain to 1.25, however, causes a measurable loss of stability and decreasing to 1.1 yields an increase. 1.2 was therefore selected as K_u and the waveform was analysed to determine an approximate T_u of 0.65 seconds.

These values yield PID gain terms of 0.72, 2.2154, and 0.0585. It was quickly apparent that, although the system would stabilize, these gain coefficients were not well tuned. As was done for QBot1, a sweep of T_u terms was performed to determine if better tuning could be achieved. The corresponding system response of a few selected values is shown in Figure 35.

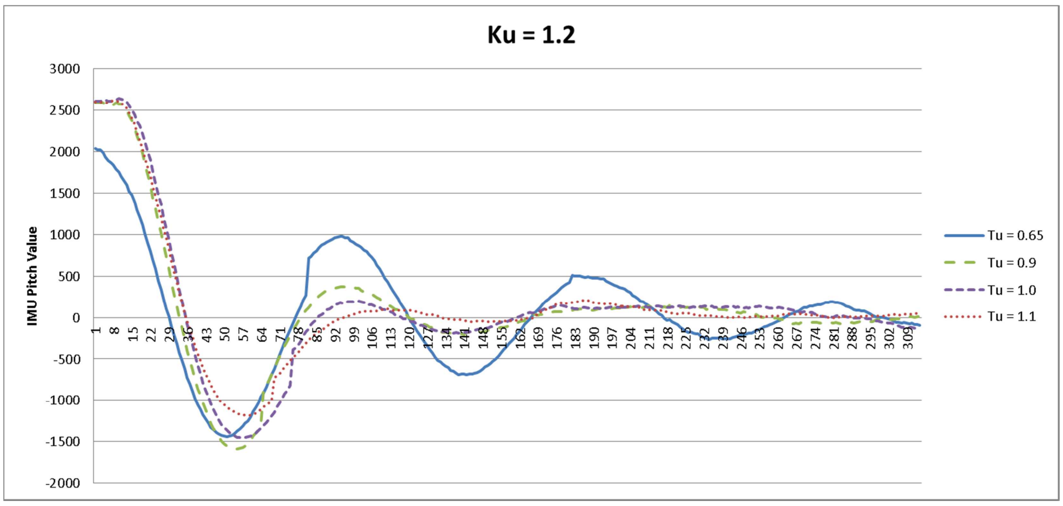


Figure 35- Araqnobot Z-N Tuning for Pitch (Ku = 1.2)

This demonstrates that a T_u value of 0.65 is clearly inadequate. 0.9 yielded a near-expected Z-N tuning response while 1.1 was starting to exhibit latency from the growing derivative term. 1.0 was therefore selected as the characteristic T_u and then the value of K_u was tested again to ensure its validity with the modified ultimate period. The results of a couple of tests on either side of 1.2 are shown in Figure 36.

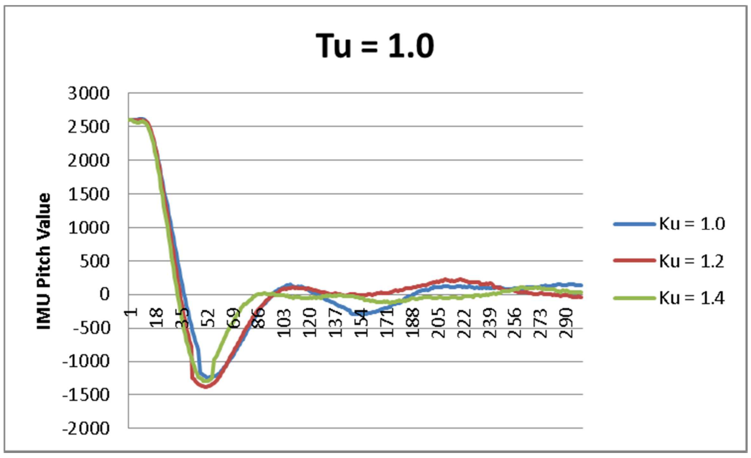


Figure 36 - Araqnobot Z-N Tuning for Pitch (Tu = 1.0)

The quality of these responses is largely a matter of subjective evaluation as there is little real difference. Although the data corresponding to a K_u value of 1.4 shows quicker convergence, it is a less conventional Z-N tuning response and the value of 1.2 can certainly be declared as sufficiently well-tuned. For this reason, final Z-N tuning parameters were set at $K_u = 1.2$ and $T_u = 1.0$. Application of the Z-N tuning equations thereby yields PID gain terms as follows: $K_p = 0.72$, $K_i = 1.44$, and $K_d = 0.09$.

With pitch reasonably well tuned, the airframe was switched in the apparatus so that roll could be tested. Because QBot1 had demonstrated fairly little difference in system response between roll and pitch, the same PID parameters were applied. Figure 37 shows the achieved result and it was determined that this was sufficient.

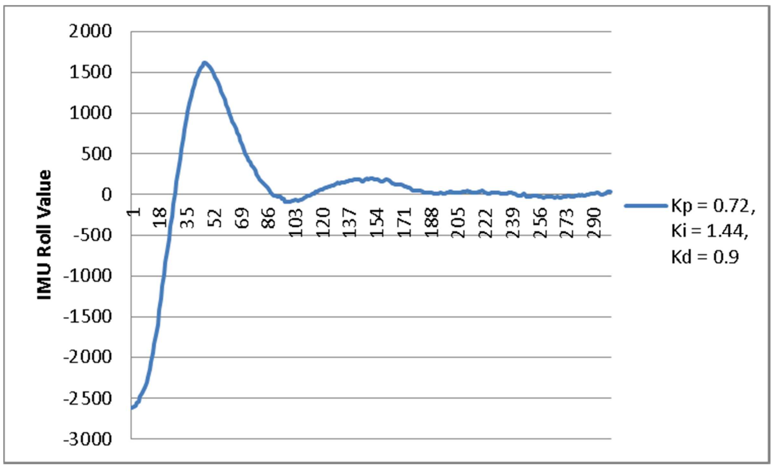


Figure 37 - Araqnobot Roll PID Parameter Check

6.2.2.2. Three-Axis Testing

Araqnobot received relatively little testing on a 3 DOF test stand. The roll and pitch tuning had worked well and stability was excellent. The yaw PID gain terms ($K_p = 1.0$, $K_i = 0$, $K_d = 0.2$) that worked well on QBot1 also yielded sufficient response for Araqnobot. In that area, all that was considered necessary for indoor operation was the ability to maintain heading and those values provided satisfactory response.

Remote control of thrust and orientation (for position compensation) had been implemented in the base station software by this point and those controls were exercised on the test stand in preparation for free flight.

6.2.2.3. Free Flight

The transition from constrained testing to free flight was a major step and several difficulties were encountered. As an example, the pointed design of Araqnobot's landing gear tended to catch on the carpet in the testing area and that would result in integral windup of roll and pitch and a loss of yaw control (the airframe would spin when a leg caught). These effects led to corresponding instability on takeoff and landing.

Another issue quickly realized was that tuning for a static setpoint on the test stand yielded insufficient responsiveness for the continuous adjustments of free flight. Any time the control changed, there was a corresponding wobble of the airframe and the response was not as crisp as needed for proper control. That led to an experimental adjustment of the scale factor for the PID (which is simply a multiplier applied to the PID output and has the same effect as increasing (or decreasing) the K_u for Z-N tuning). It was found that doubling or tripling the gain factor increased responsiveness and a series of tests (back on the 1 DOF test apparatus) were performed to evaluate the effect. The corresponding data is shown in Figure 38.

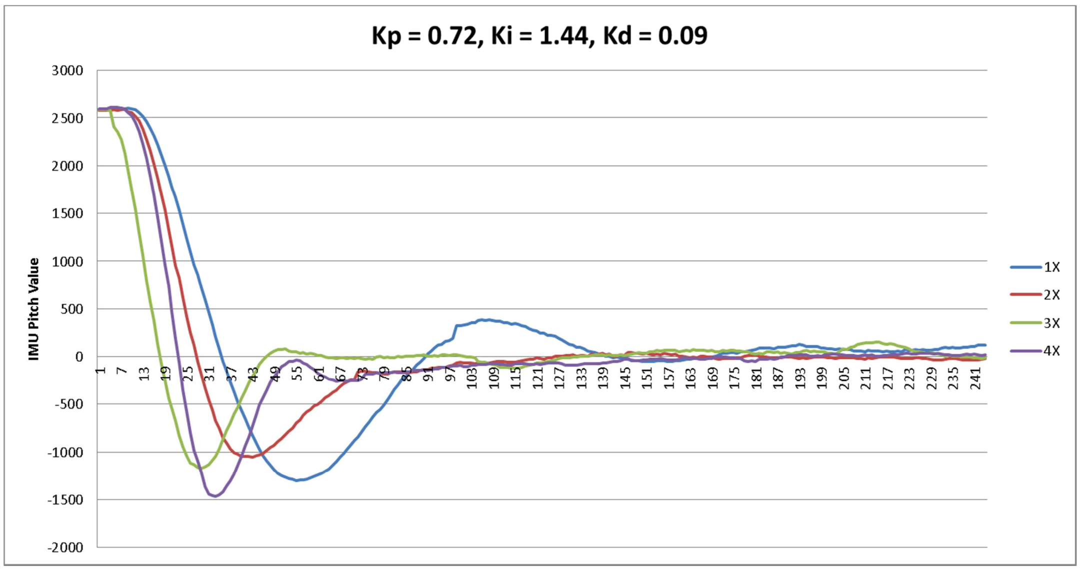


Figure 38 - Araqnobot PID Scale Factor Increase for Pitch

This data clearly shows that multiplying the PID output yields much stronger system response. As the multiplication factor is increased, however, the system begins to respond more strongly to sensed disturbances from the IMU and the amount of undesirable twitching is increased. In this case, a gain factor of 2 was selected for Araqnobot and that gave a good level of responsiveness without excessive twitching during flight.

The araqnobot airframe has been flown somewhat regularly indoor for many months. It was also tested outdoor and worked well but is significantly impacted by wind disturbance. Overall, the testing process has demonstrated the ability of this platform to execute reliable autonomous orientation maintenance in free flight and it will serve as a solid platform for the development of further capability.

6.2.3. Position Control

With a focus on indoor use, it was recognized that localization would be a challenge for Araqnobot. Echo location was considered to be the best option for initial testing due to the compactness of sonar rangefinders and their ease of use. The starting point for position control was altitude and an MB1200 sonar module was oriented to point downward from the bottom of Araqnobot’s battery holder. A few tests were made flying the airframe off the ground and significant variability was discovered. Over a number of readings, the height value could be reliably determined, but individual data points ranged from zero to maximum. Those extremes were next filtered out (by rejecting extreme jumps and carrying the previous sample forward), but the data output of the sonar module continued to be erratic. A record of sampled data through a short flight is shown in Figure 42.

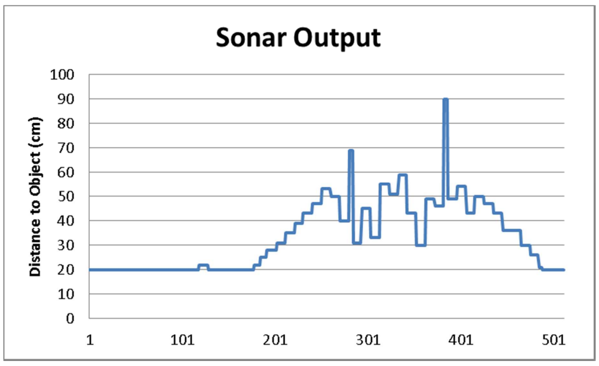


Figure 39 - Araqnobot Sonar Testing

The flight corresponding to this data consisted of take-off after about 2 seconds, followed by an ascent to more than half a meter, several seconds of hover (plus or minus a small amount), followed by descent back to the ground. The sonar data roughly correlates, but the instantaneous variation is far too great to allow autonomous control. Possible influences on the reading include vibration, noise, air turbulence, and echo material (carpet); some of these could perhaps be altered to improve accuracy. It would also be possible to filter the data for reliable determination of position, but the number of samples would have to be quite large (accumulated over a second, or more). That is probably a reasonable approach to take so long as the robot is intended to move at a correspondingly slow rate. For the purposes of this research, limited time was spent on this problem and no reliable solution was found.

6.3. Jumbo QBot

6.3.1. Motor Testing

Jumbo QBot uses the same motor controllers as Araqnobot but has larger motors and propellers. Once again, the intelligent calibration software of the ESC32 developers was employed in setting up the motor systems. The output graph for Motor 1 from the RPM to Voltage test routine is shown in Figure 40.

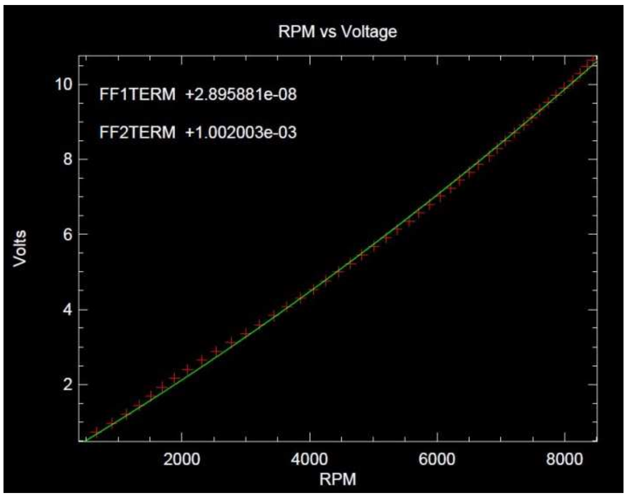


Figure 40 - Jumbo QBot Motor 1 RPM to Voltage Testing

The most significant difference between the motors for Jumbo QBot and the motors for Araqnobot (and QBot1) is the maximum RPM of around 8500 as compared to approximately 6500. Combined with a larger propeller and considering the squared relationship between angular velocity and static thrust, the Jumbo QBot power system is capable of more than twice the vertical thrust of its predecessors.

The extra output power naturally comes with a correspondingly dramatic increase in input power. The ESC32 current limiter calibration was run on each Jumbo QBot motor system and the graphed result for motor 1 is shown in Figure 41.

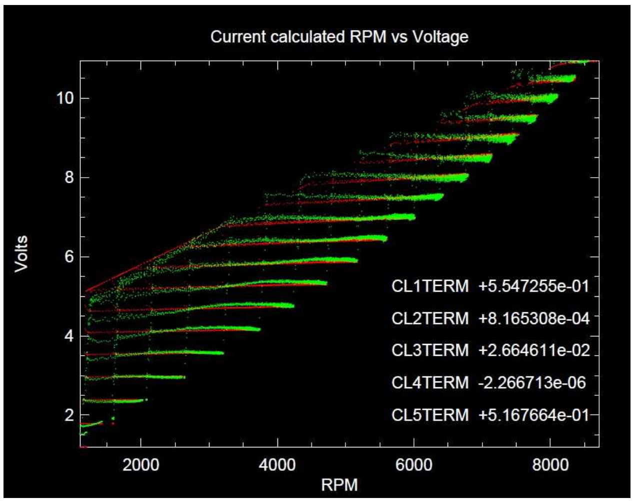


Figure 41 - Jumbo QBot Motor 1 Current Limiter Calibration

This chart was produced with the current limit setting equal to 30A and it looks very similar to the output of the same test for Araqnobot. One difference worth considering is that the Araqnobot motor test had a fairly solid sloped line forming the upper bound of its data points. Jumbo QBot does not and that suggests a difference apparent in the tests; the Araqnobot motor tested did not really need a current limiter while the Jumbo QBot motor tested certainly does. This is because the Araqnobot motor was able to quickly spin faster in response to increasing voltage and would thereby never allow a high voltage at a low speed to occur (which would have caused a corresponding significant increase in current draw). In other words, as the ESC32 started to increase the voltage duty cycle, the motor would respond almost as quickly as the increase would occur and the RPM to Voltage disparity remained small, along with the associated current (and many data points were accumulated along the acceleration line). The Jumbo QBot motor that was tested, however, could see a much larger differential and that was constrained only by the ESC32 current limiter. As soon as a spike over 30A was detected (voltage was rising, but RPM was staying low), the current limiter would engage and stop the progression. As a result, the acceleration line is much more sparse and characterized by steps representing each interval of the test program. If the current limit were increased, more data would be expected to appear in the top-left quadrant of Figure 40.

6.3.1.1. Motor Flight Data

The processing power and memory capacity of Jumbo QBot allowed for extensive monitoring of all system aspects. Voltages applied to the motor and corresponding current draw were tracked during flight to evaluate power consumption and the performance of the current limiter. A plot of some collected data (at a sample rate around 25 Hz) is shown in Figure 42.

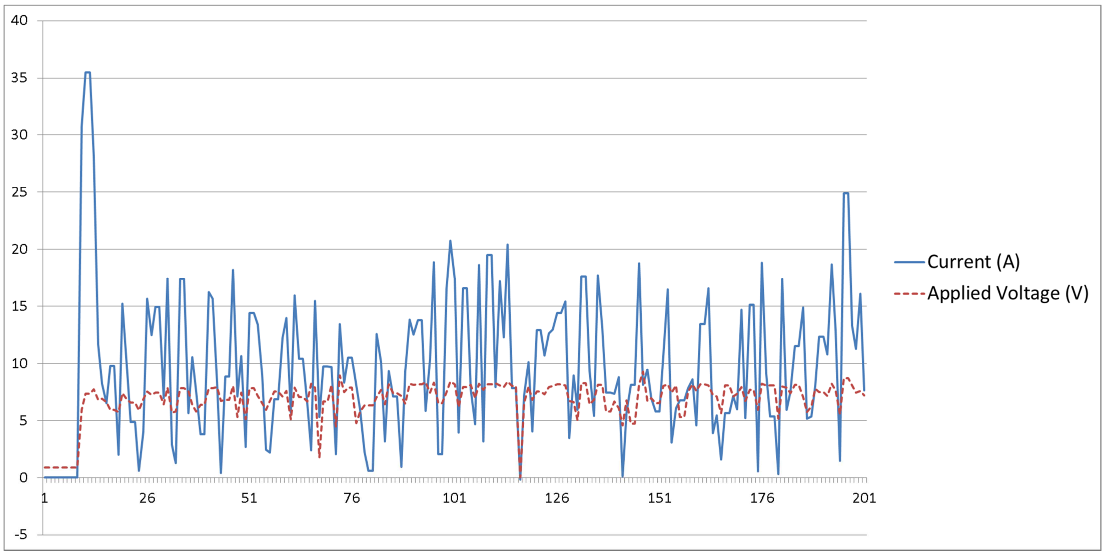


Figure 42 - Jumbo QBot Sample of Real Time Flight Data for Motor 1

The spike around point 10 corresponds to the first command to bring the motor up to hover speed. The current draw exceeds the limit of 30 amps instantaneously, but it would presumably be much higher without the stepping function in place. This chart demonstrates the highly dynamic nature of quadrotor flight as the motor voltage, current, and corresponding rotational velocity are constantly changing. Data from the other 3 motors was collected during the same flight (which continued beyond the data shown in Figure 40) and yielded further interesting information, as collected in Table 12.

Table 12 - Jumbo QBot Real Time Flight Motor Data

	Motor 1	Motor 2	Motor 3	Motor 4
Max Voltage	9.5 V	6.9 V	9.0 V	7.7 V
Max Current	35.5 A	33.3 A	33.2 A	30.7 A
Average Voltage	7.1 V	4.2 V	7.2 V	4.6 V
Average Current	9.8 A	4.5 A	11.9 A	4.2 A

Particularly interesting is that the motors affecting pitch (motor 1 and motor 3) draw significantly more power than the motors affecting roll. This is almost certainly due to the lack of an integral term in the PID for yaw; with a K_i value of zero, the equilibrium point of the yaw control system will have some static offset that, thanks to the non-zero K_p term, will cause slightly more thrust to be required on one of the roll/pitch axes while the other has less required. The end result is that two motors see higher loading than the others and the distribution is dependent upon the ongoing static yaw offset.

A further interesting data point is that the average total current draw for the airframe motors was found to be 30.3 amps. Assuming around 1A average for the control board and given the theoretical battery capacity of 11000mAh, this suggests that Jumbo QBot could remain in flight for over 21 minutes (with no payload).

6.3.2. Orientation Control

Orientation control of Jumbo QBot was approached in a similar manner to that employed for QBot1 and Araqnobot in that it began with Z-N tuning of the roll and pitch axes PIDs. After that, the transition was made directly to free flight (rather than to a three DOF test stand) and the steps of this approach are detailed below.

6.3.2.1. Single-Axis PID Tuning

Because the arms of Jumbo QBot had been constructed out of rectangular aluminium, the roller bearing apparatus couldn't be used for single-axis testing. An entirely different fixture was manufactured out of an inverted table and various hardware (including high-load fishing swivels). The assembled apparatus is shown in Figure 43.

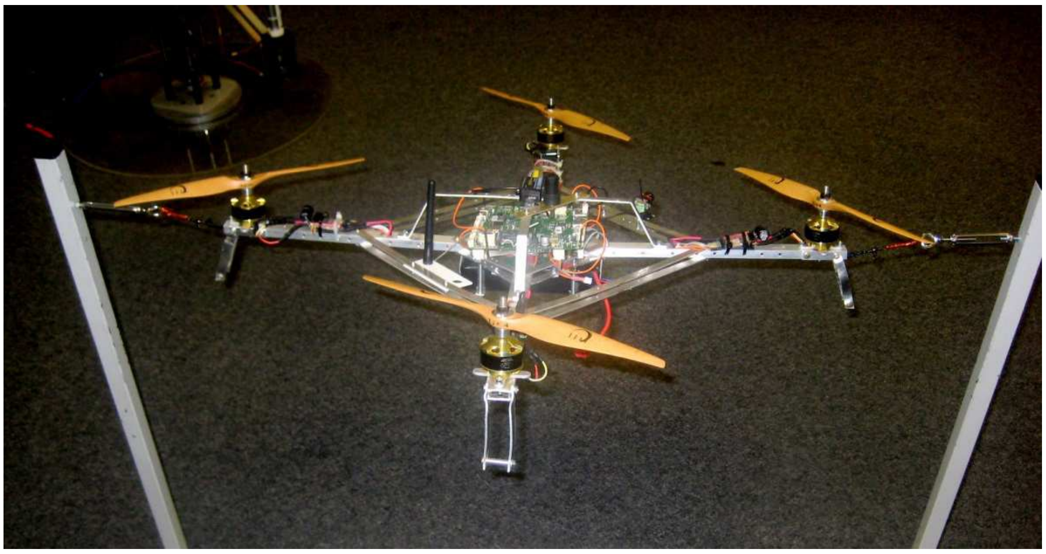


Figure 43 - Jumbo QBot on 1 DOF Test Stand

One advantage of this setup was that it allowed the fixed motors to be run at the same time as the motors on the axis being tuned. By running all four motors, the gyroscopic impact of the motors on the axis of rotation would be included in the tuning result. Gyroscopic effects are not being compensated for mathematically, but their (arguably small) impact will still affect the desired rotation and tuning in their presence was expected to yield results more representative of real flight.

Once again, the modified Z-N tuning approach was applied and the first step to find an approximation of K_u was taken by increasing the proportional gain while leaving K_i and K_d equal to zero. Jumbo QBot exhibited different behaviour from Araqnobot but it was again an improvement toward an expected Z-N tuning response. Consistency of oscillation was much more reliable and stable than had been experienced in testing either of the previous models. Because Jumbo QBot had significantly more memory, it was also possible to start the system near an angle of zero and observe the progression toward oscillation, whether stable or otherwise. Figure 44, Figure 45, and Figure 46 show the results of pitch testing with different K_p terms over 10 second intervals from startup. (Note: all single-axis testing for Jumbo QBot was done with a lift thrust setting of 20N.)

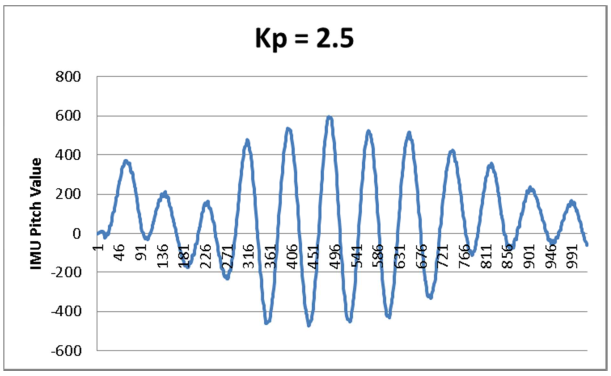


Figure 44 - Jumbo QBot P-term Only Pitch Testing (K_p = 2.5)

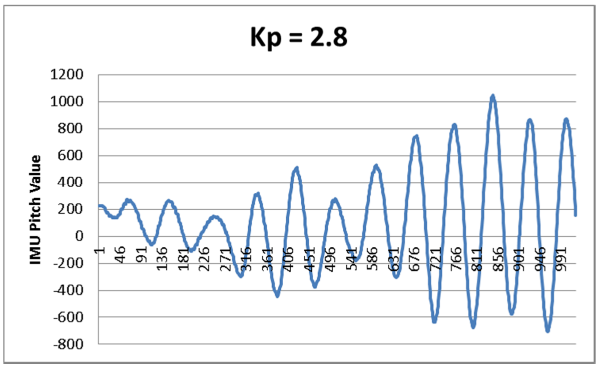


Figure 45 - Jumbo QBot P-term Only Pitch Testing (K_p = 2.8)

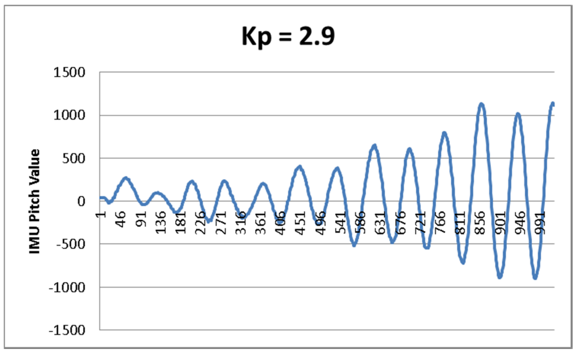


Figure 46 - Jumbo QBot P-term Only Pitch Testing (K_p = 2.9)

At a proportional gain of only 2.5, the airframe pitch would begin to oscillate then diminish and stabilize. 2.6, 2.7, and 2.8 all tended to yield fairly stable oscillations while a gain term of 2.9 caused oscillation increase and a loss of stability. 2.8 was therefore selected as K_u and the oscillations were analysed to yield an approximate period (starting point for T_u exploration) of 0.84 seconds.

Two iterations of ultimate period tuning were then employed; the first one used coarse steps around 0.84 while the second performed fine adjustments on the evaluated best response from the first iteration.

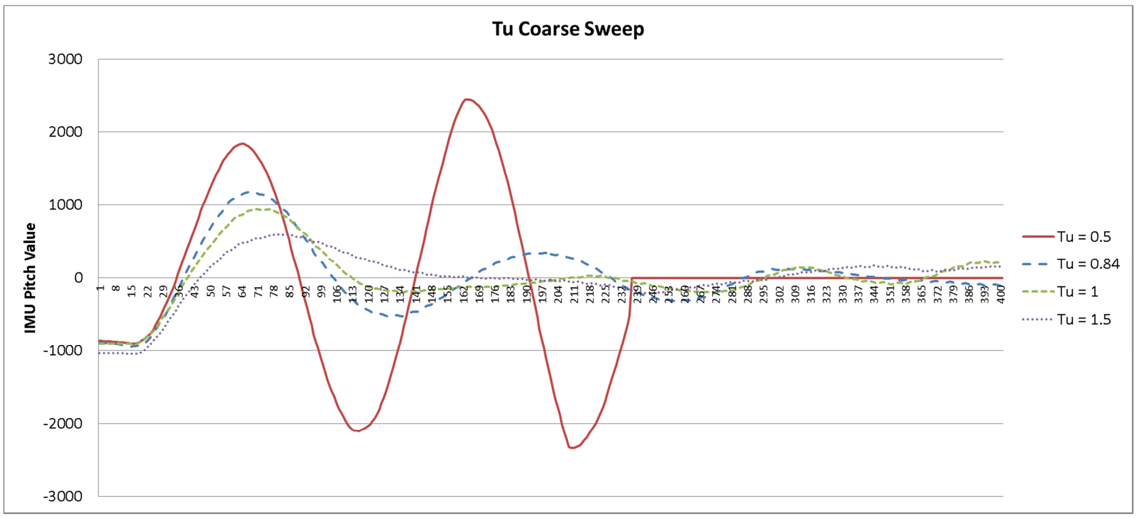


Figure 47 - Jumbo QBot Pitch Only Testing: T_u Coarse Sweep

Figure 47 shows that T_u values of 0.5 and 0.84 are insufficient while a value is 1 is pretty well tuned and 1.5 is too much. To explore the possibility of better tuning, 1 was then taken as a starting point and the effect of small increments was captured as shown in Figure 48.

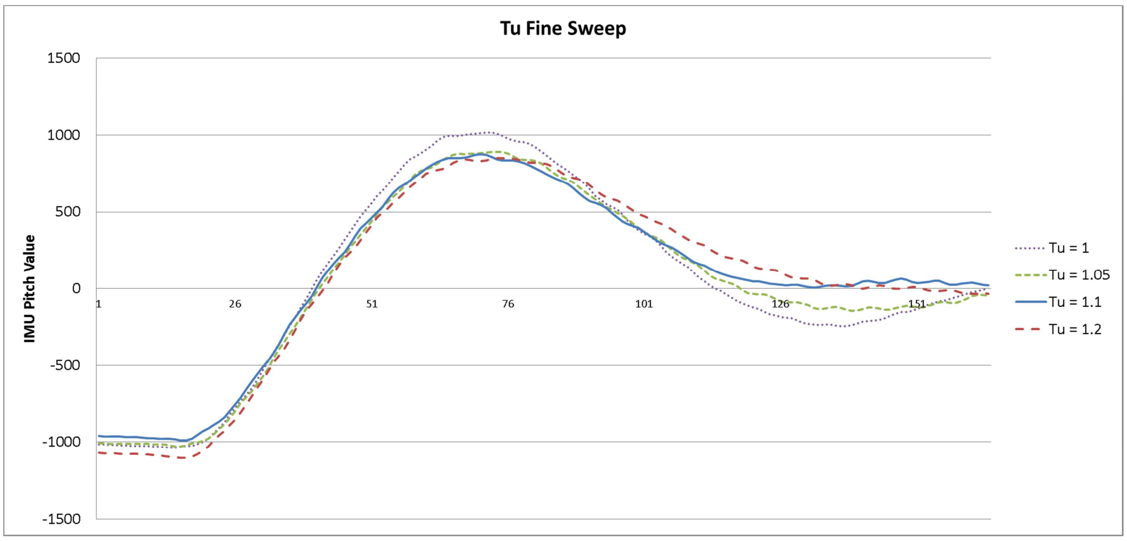


Figure 48 - Jumbo QBot Pitch Only: T_u Fine Sweep

At this point, a value of 1.05 was considered an optimal Z-N response as it had a small amount of characteristic overshoot on the second setpoint crossing, but it was slightly less than that observed for 1. With a K_u, then, of 2.8 and a T_u of 1.05, the PID gain terms were calculated as: $K_p = 1.68$, $K_i = 3.2$, and $K_d = 0.21$.

Having determined on Araqnobot that better stability when flying could be achieved through uniform amplification of all PID terms, an evaluation was done on Jumbo QBot to determine the optimum scale factor to apply. All integer terms from 1 to 6 were tested, but only the final three have been captured in Figure 49, below.

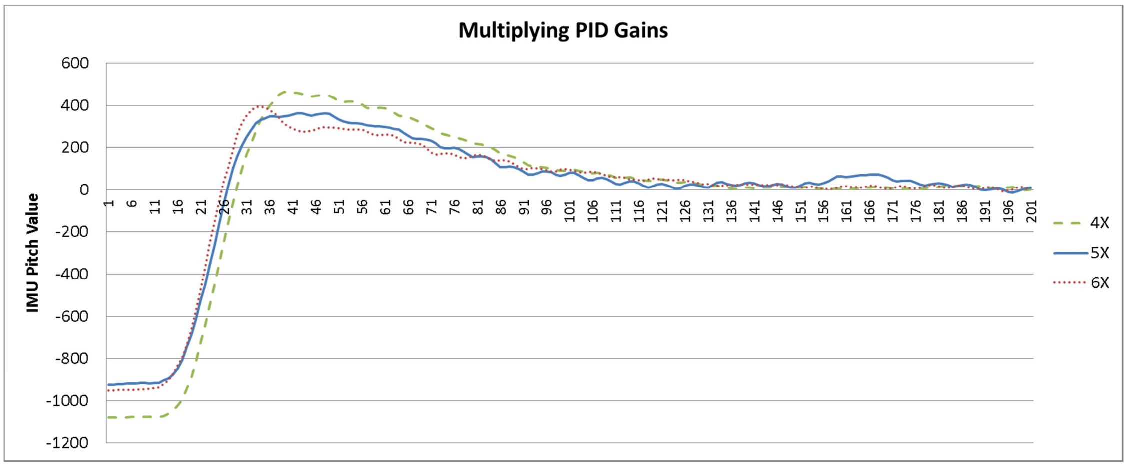


Figure 49 - Jumbo QBot Pitch Only PID Term Scaling

This seems to demonstrate a turning point between a multiplication factor of 5 and 6. Between 2, 3, 4 and 5, the trend was always toward a reduction in deviation from the setpoint and reduced area of overshoot. At 6, there is increased overshoot and slightly slower convergence. In the end, a multiplier of 4 was used as the default setting for Jumbo QBot flights. As was seen with Araqnobot, a strong multiplier does yield some twitchiness in response to IMU noise, but it gave excellent responsiveness when flying.

The same approach was rigorously applied to roll testing and, on the Jumbo QBot airframe, a measurable difference was encountered. The Z-N tuning parameters that were eventually selected as optimal were $K_u = 3.5$ and $T_u = 1.1$. The corresponding PID gain factors are $K_p = 2.1$, $K_i = 3.8182$, and $K_d = 0.28875$. The scaling multiplier applied for most flights was set at 3.

6.3.2.2. *Free Flight*

No yaw testing on a three DOF test stand was performed for Jumbo QBot as part of this research. The Jumbo QBot airframe was transitioned directly to free flight and the same yaw PID parameters successfully used on the previous airframes were applied again. They continued to work well and full orientation control was successfully realized.

This was well demonstrated through the achievement of a primary flight goal: ascent to over 9 meters (30 feet) above the ground under manual control. A plot of the altitude readings (after normalization) from the successful flight is shown in Figure 50. These readings were taken from the MS5803 pressure sensor discussed in the next section.

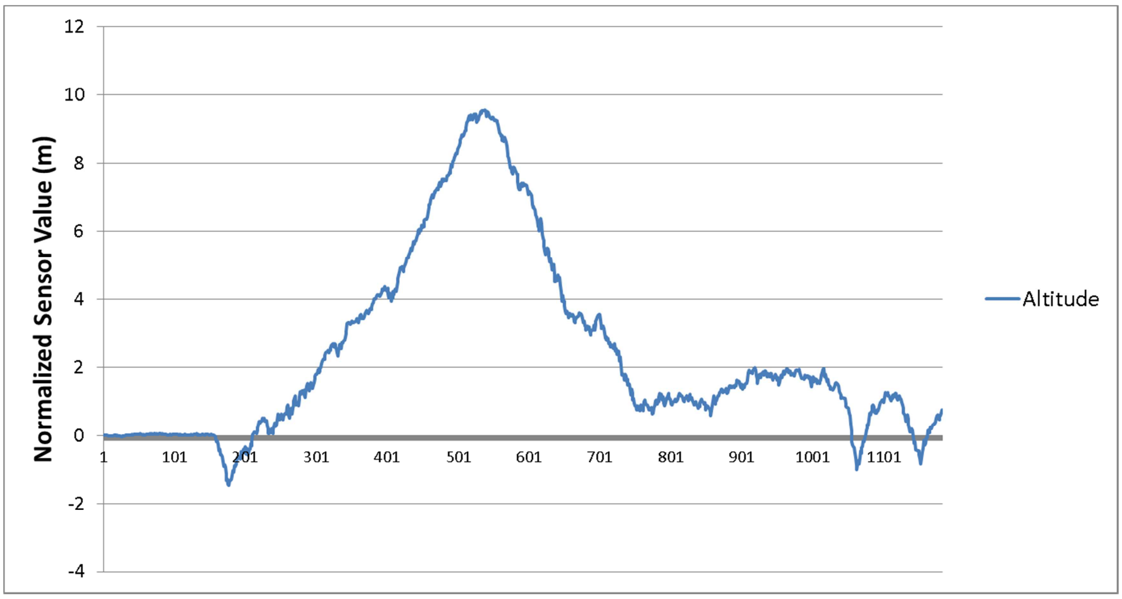


Figure 50 - Jumbo QBot High-Flight Data

6.3.3. Position Control

6.3.3.1. Position Detection Component Testing

6.3.3.1.1. Pressure Sensor Testing

Having observed the limitations of sonar range-finding with the Araqnobot MAV, a different approach was taken for altitude control on Jumbo QBot. Tiny pressure sensors were employed that can pick up minute variations in air pressure to the degree that they are able to discern elevation changes of less than a meter. Two different sensors were employed (as discussed in Section 4.3.5.2) and some testing data is presented in the following sections.

6.3.3.1.1.1. MS5803 Testing

The MS5803 was the first sensor to be tested and it yielded reasonably good results. Pressure sensors are known to generate readings with a significant amount of noise and an example of the raw data (after conversion from pressure to altitude) collected from the MS5803 is shown in Figure 51. The solution to noise is filtering and a complementary filter was applied to the sensor output, as described in the firmware Section 5.2.3.3.3. The result of a complementary filter with a 10% weighting for new data and 90% for the previous value is shown superimposed over the raw data in Figure 52.

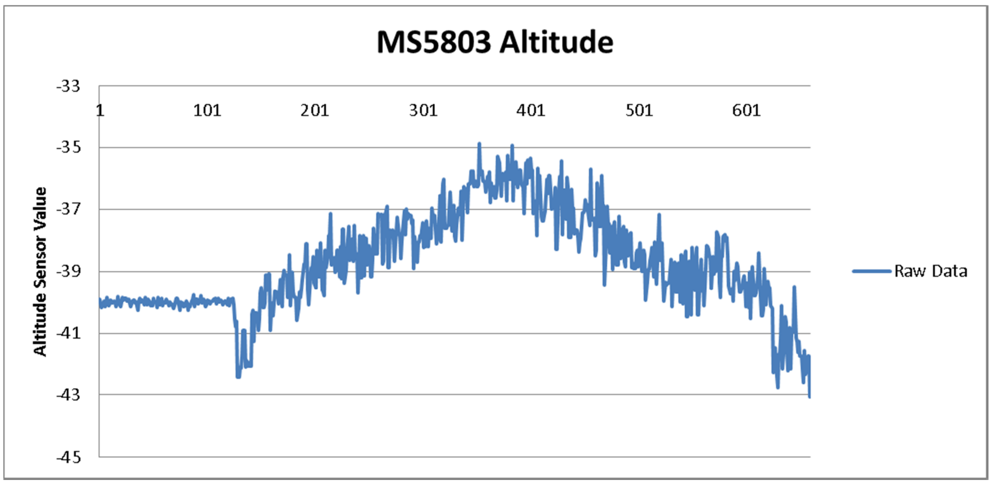


Figure 51 - MS5803 Pressure Sensor Raw Altitude Data

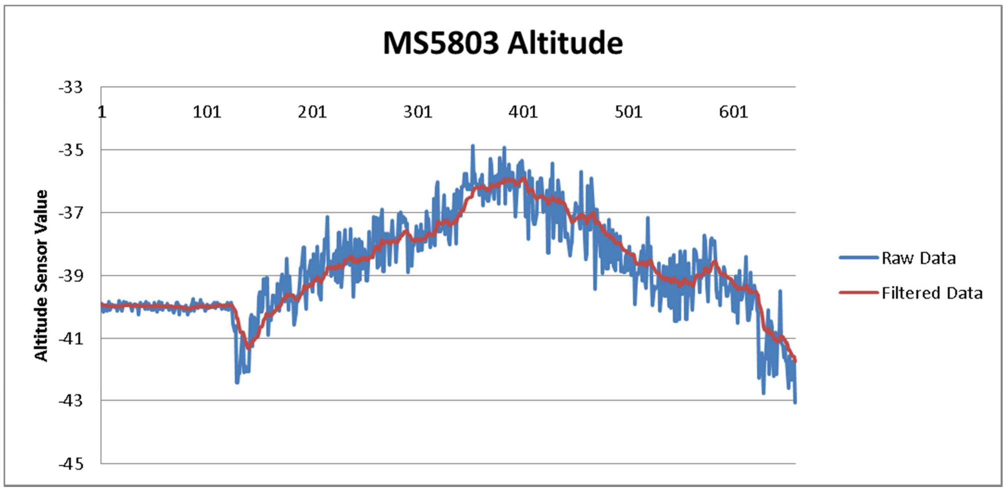


Figure 52 - MS5803 Pressure Sensor Filtered Altitude Data

The data shown in these figures was collected during a short flight that began by spinning up the motors at around data point 100. The air pressure disturbance caused by the propellers spinning at near hover velocity on the ground is seen by the sudden pressure increase (altitude drop) at that point. After achieving lift velocity, the MAV was flown to a height of around 4 meters before returning to the ground (where the rotors wash again drops the altitude reading below ground level). It should be noted that the absolute pressure was not significant for the purposes of this research. All autonomous operations (e.g. altitude hold) are performed relative to the sensed pressure/altitude at the point they are engaged and the recorded filtered data correlates well to the observed flight pattern.

6.3.3.1.1.2. BMP180 Testing

Although the MS5803 worked well, it was thought that another device might yield a small improvement and that would correspondingly enhance altitude hold performance. A USB module containing a Bosch BMP180 pressure sensor was attached to one of

Jumbo QBot's USB ports and the same filter was applied to its output as was used for the MS5803. A sample of the output results (from the same flight used for the MS5803 sample data) is shown in Figure 53.

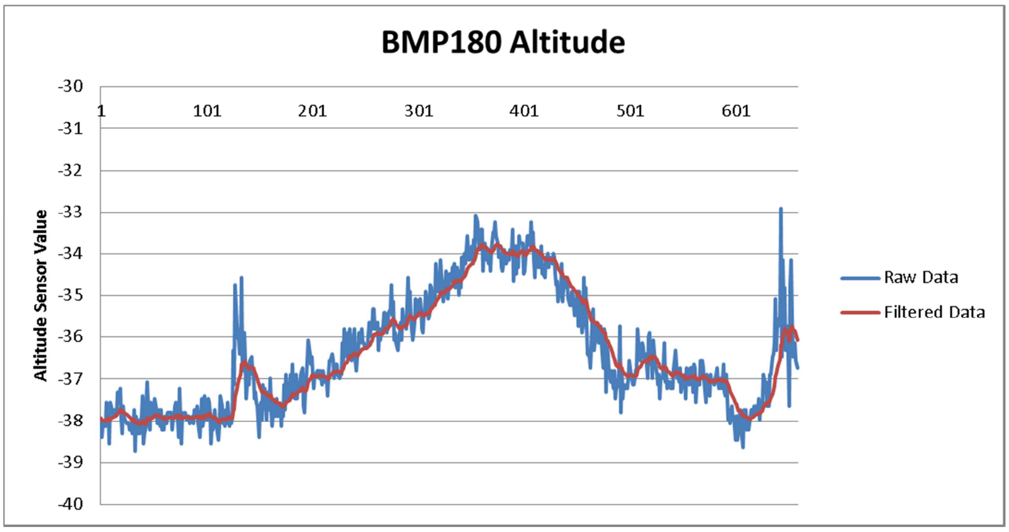


Figure 53 - BMP180 Pressure Sensor Filtered Altitude Data

These results are very similar to the MS5803 (again, absolute pressure/altitude is considered irrelevant), but it can be seen that the air disturbance near the ground causes a pressure drop for this sensor (at least in the position it was installed).

6.3.3.1.1.3. Pressure Sensor Fusion

Both pressure sensors worked well, but they had opposite responses to air disturbance near the ground and the advantage of averaging their values was apparent. A fusion ratio of 50% each was selected and the combined result is shown in Figure 54.

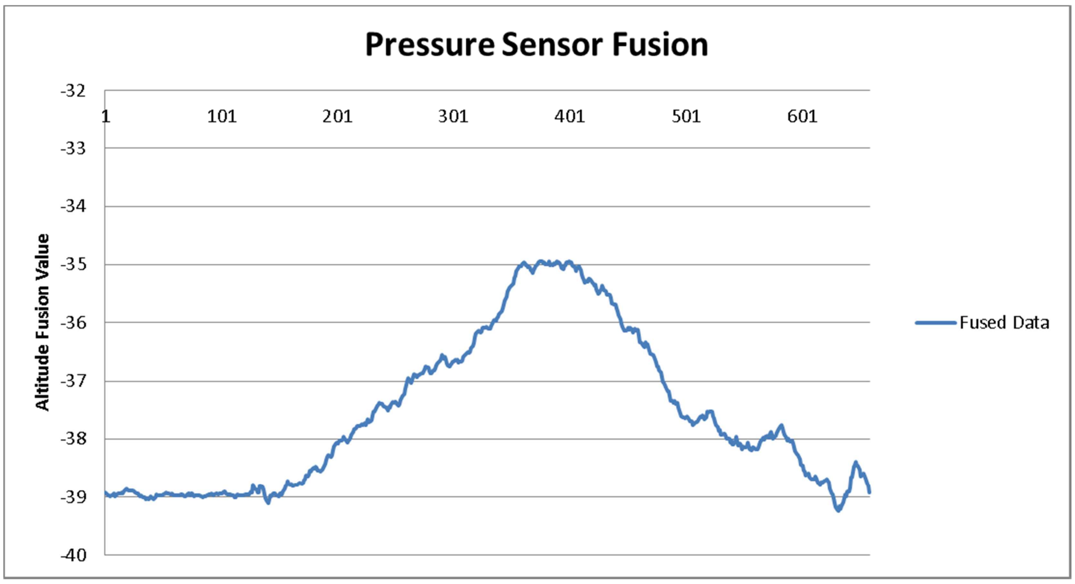


Figure 54 - Pressure Sensor Fusion Altitude Data

This fused data approach eventually became the default for altitude sensing in flight, but much of the testing described in the subsequent sections relied exclusively on the MS5803. This is because the BMP180 module wasn't available until relatively late in the research process. In the meantime, other approaches had been considered and extracting altitude from the GPS data was also explored.

6.3.3.1.2. GPS Testing

The LM345 offers two different sources of GPS information: raw data from the internal GPS module, and Kalman-filtered data from the navigation processor that can provide approximated position even in the event of GPS outage (provided it is of short duration). The latter was considered to be the most desirable and a large amount of testing was performed with the navigation source as the default. It was eventually discovered, however, that the raw GPS output provided data better correlated to the actual movement of the airframe and that the Kalman filtered values were quite different. An example of the disparity is shown with respect to latitude in Figure 55.

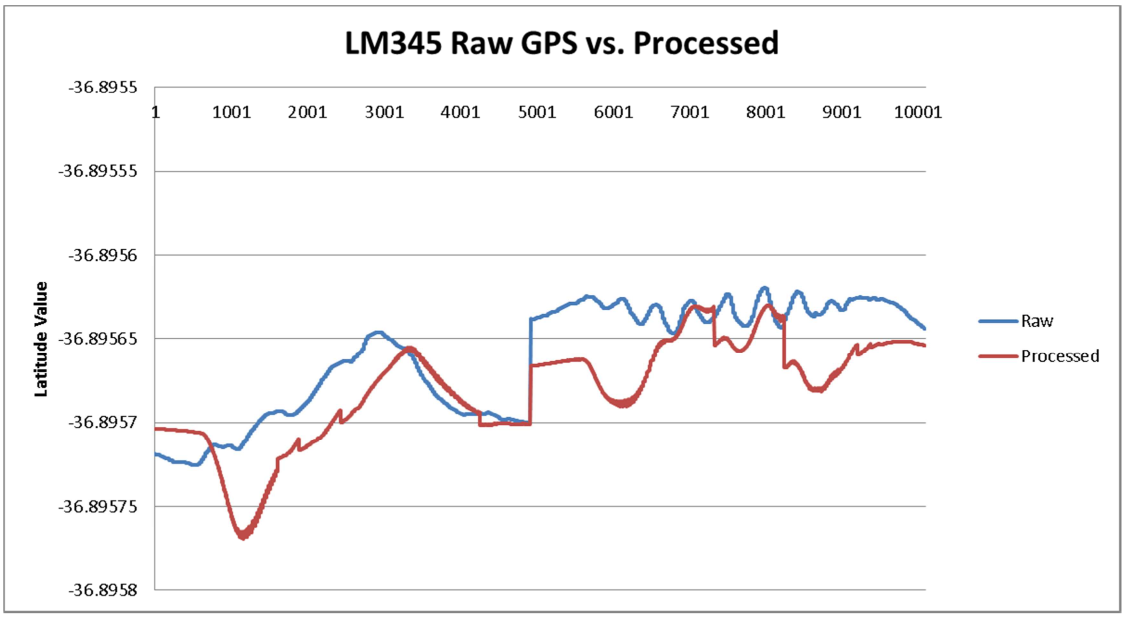


Figure 55 - LM345 GPS Raw Data vs. Data Processed by Kalman Filter

The differences were substantial enough during autonomous-mode testing that the decision was eventually made to rely on the raw GPS data only. This has proven sufficient for basic position holding and tracking.

Although it is well known that GPS altitude values lack accuracy, they are nonetheless provided and they were evaluated to determine if they could be usefully monitored. An example of recorded data from testing done on a level surface (i.e. no altitude change) is shown in Figure 56. This type of uncorrelated variation was typical and the GPS height value was considered too unreliable for altitude hold logic (but may be useful for terrain awareness in applications beyond the scope of this research).

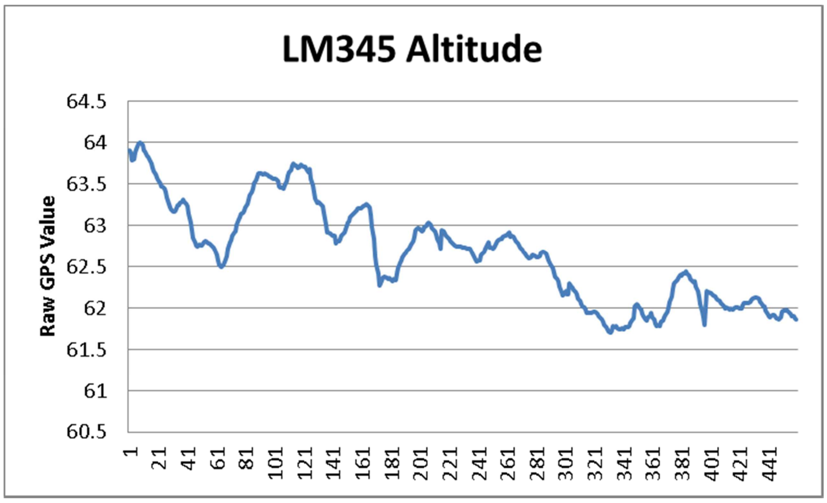


Figure 56 - LM345 GPS Altitude

6.3.3.2. Latitude and Longitude Control

6.3.3.2.1. Mathematical Simulation

Because the latitude and longitude PIDs generate angular orientation outputs, it is possible to easily calculate the corresponding lateral thrust as shown in equation (35). A quick mathematical simulation can then be performed using an assumption of perfect, instantaneous system response that is subject to no external forces. This is useful as a means of evaluating PID parameters in general terms and it helps to isolate reasonable relationships between the gains without risk to the physical MAV (the initial testing of autonomous operation is potentially dangerous because it means that the operator must relinquish control to the firmware; if the algorithms are flawed (or have the expected bugs) the airframe behaviour may become erratic (and potentially destructive)). Newton's basic laws of motion provide formulas for linear acceleration (a), velocity(v), and corresponding displacement (Δ_{position}) seen by an applied force (F) according to the equations below. (Other variables are as follows: m is mass, v0 is initial velocity, and t is time.)

$$a = \frac{F}{m} \tag{37}$$

$$v = at + v_0 \tag{38}$$

$$\Delta_{position} = vt + \frac{1}{2}at^2 \tag{39}$$

Combining these equations together, then, we have the components for a virtual simulation that can be worked out in a spreadsheet. Constant variables can be defined for desired position (the setpoint or endpoint), for the system update interval (correlating to, for example, the GPS update rate), for P term gain, I term gain (if used), and D term gain. The system response over time is then organized into rows having one column for position that starts at a set value. Another column contains velocity (and is initially zero). If the I term were employed (it wasn't for this study), the accumulated positional error could take another column. PD (or PID) output, resultant translational force, and corresponding acceleration then each have their own subsequent columns. From all of these values, then, subsequent rows of the table can be populated where each row is derived from the preceding one and uses equation (30) or (31) (latitude and longitude PID formulas) to update the simulated orientation and work out the corresponding positional changes. Simulated response is then very easy to see in a chart that plots the position changes over time and can be used to evaluate the gain terms of the control algorithm. Figure 57 shows a chart with changing P terms while Figure 58 depicts the effect of changing D terms.

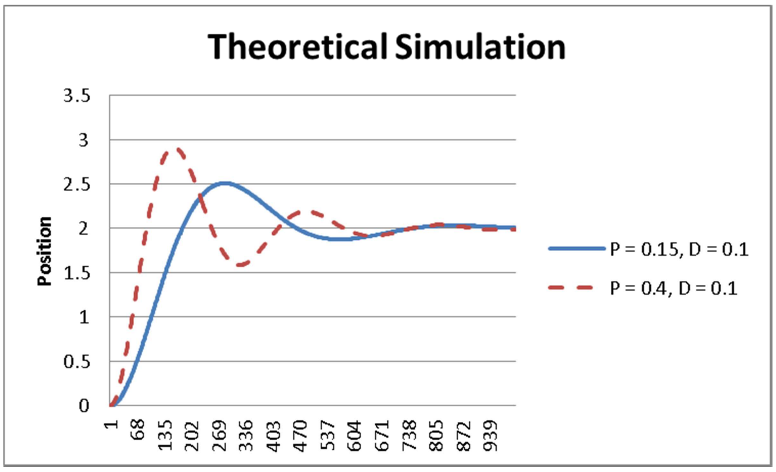


Figure 57 - Simple Simulation of P-term Increase for Latitudinal or Longitudinal Translation

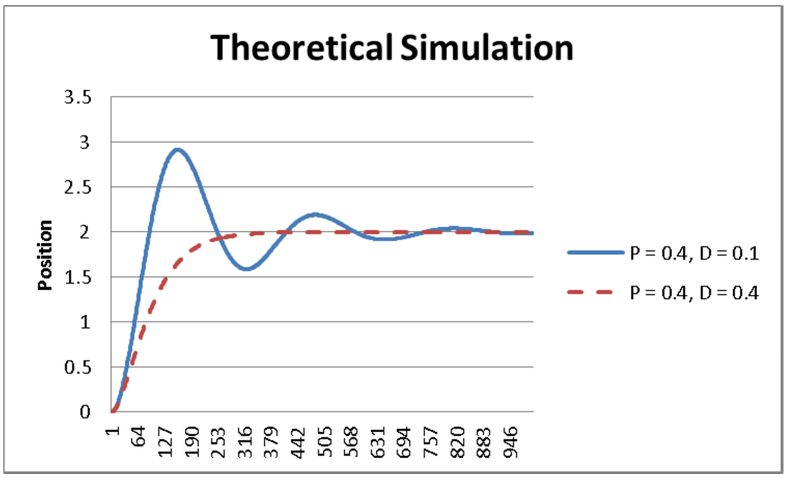


Figure 58 - Simple Simulation of D-term Increase for Latitudinal or Longitudinal Translation

This approach is very helpful in visualizing and evaluating the theoretical system response. It highlights, however, the difference between an ideal theoretical system and a real practical one. In the mathematical simulation, it can be demonstrated that larger and larger gain terms lead to quicker achievement of the setpoint and stability at it. The corresponding behaviour of the virtual system is to turn the virtual airframe more and more vertical and thereby increase total thrust without bound (vertical thrust has been defined as always equal to mg, so if the airframe is tilted toward 90 degrees, the total thrust required must approach infinity). It makes sense, then, that the virtual airframe can instantly move between points if its gains are set large enough.

Of course, in a practical system, this is not possible. Furthermore, it falls outside the defined characterization of the system as always remaining in a linear region near an attitude of hover. Placing a limit on the angle that can be output from the PID deals with that issue, but still leaves a practical concern; even with a moderate allowed angle, lateral acceleration can theoretically continue without bound. In reality, this will ultimately be limited by resistance of the air through which the airframe moves, but the achieved velocity would still be much higher than might be practically desired (for example, for operation near the ground, it may be desired that the MAV not be allowed to travel faster than a person can run). Limiting the achieved velocity (and thereby defining the point at which acceleration should stop) is fairly easily done in a PID calculation. The P and I terms must be collectively limited to a set amount. Because the D terms corresponds to negative velocity (i.e. counter to the PI terms), as the velocity approaches the PI limit, the PID output will approach zero.

In the airframe, this would be seen as an aggressive change of attitude (assuming significant P term) in response to a jump in desired position (e.g. the airframe will roll significantly in response to a 20m change in longitude). As the MAV gains velocity, the degree of roll will be less and less until it returns to a hover attitude. At such a point, however, it still maintains its lateral velocity (e.g. will continue to drift eastward) at a value equivalent to the PI limit because no force is being exerted to stop it. That will continue to be the case until the setpoint is approached and the PI terms are correspondingly diminished. This behaviour is illustrated in Figure 59.

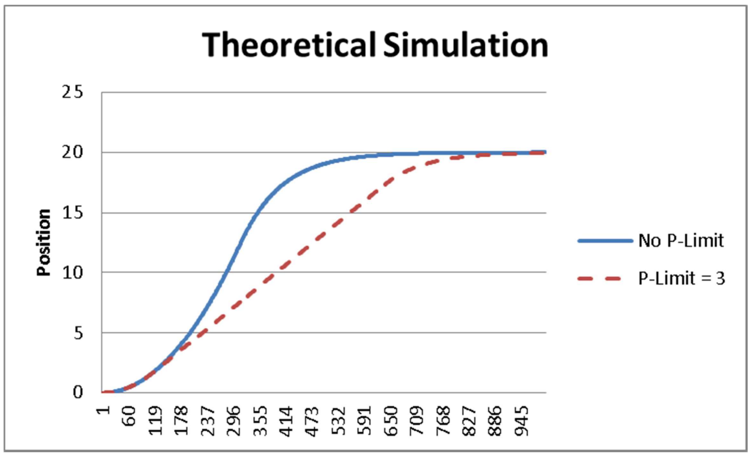


Figure 59 - Proportional Limiter Simulation for Latitudinal or Longitudinal Simulation

Without a P-term limit, the simulation airframe reaches a top speed over 28 km/h in response to a 20m setpoint change (note that the simulated maximum angle is 0.26 radians (~15 degrees)). With a P-term limit of 3 (effectively 3 m/s), the maximum velocity achieved is less than 11 km/h.

6.3.3.2.2. PID Tuning

There was limited time for position hold tuning during the course of this research. Several values were tested experimentally before the simulation approach described above was taken. The primary effect of the virtual simulation was to demonstrate that much larger derivative terms could be used for position control than had been used for orientation control. Even from a Ziegler-Nichols understanding, this makes sense as the position time constants are necessarily larger and the derivative component is directly proportional to the ultimate period in the Z-N tuning method. Using larger derivative terms quickly led to better stability and reasonable maintenance of positional latitude and longitude. The airframe still had a tendency to slowly oscillate, but it was bounded to a small area (approx. a couple meters of deviation) and the robot could be relied upon to track its position well. For both latitude and longitude, the gain terms that were eventually determined to be effective are: $K_p = 0.1$, $K_i = 0$, $K_d = 0.2$. These are quite small in comparison with the virtual simulated values, but they reflect the reality of GPS uncertainty. Because GPS values can vary significantly between samples, the system response (overreaction) is minimized through the use of smaller terms. A chart of sample latitudinal and longitudinal deviation values from a flight in which position hold was engaged is shown in Figure 61.

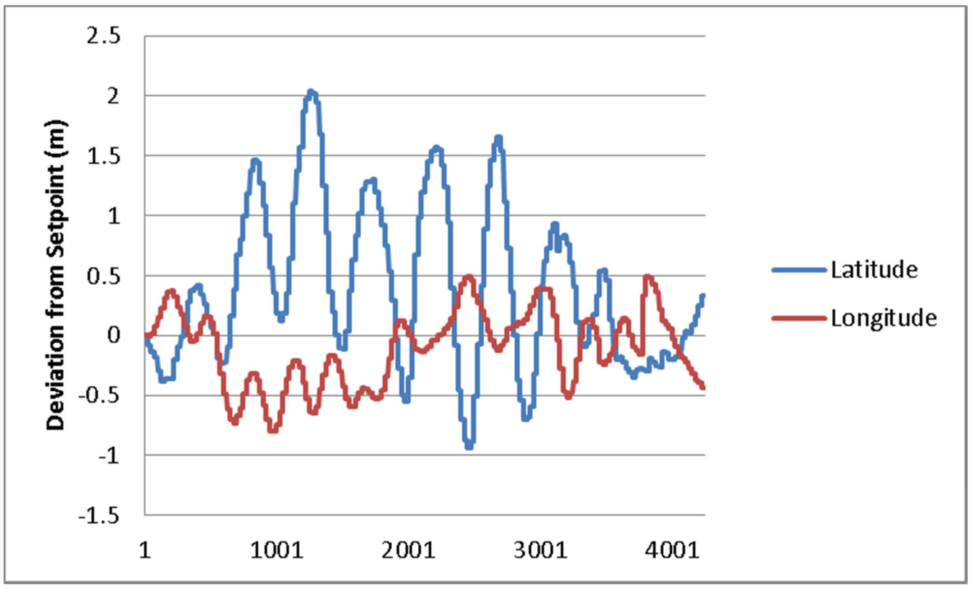


Figure 60 - Latitude and Longitude Position Hold Data

It is important to note that the samples for this plot are being taking at 100Hz and the total chart area spans about 40 seconds of flight. Interestingly, the robot exhibits significantly better stability for longitudinal maintenance of position versus latitudinal. The reasons for this may be a greater degree of accuracy in the longitude value from the GPS, variation in responsiveness of the airframe between pitch and roll, or perhaps external factors such as wind. In any case, this result demonstrates that autonomous position hold can be achieved with respect to GPS latitude and longitude readings.

6.3.3.3. Altitude Hold

6.3.3.3.1. Mathematical Simulation

A similar virtual simulation approach was applied for altitude hold as had been used for latitude/longitude hold (see Section 6.3.3.3.1). The primary difference between the other positional degrees of freedom and altitude is that relative thrust (around the hover point) is the PID output, rather than angle. Otherwise, the same concepts of proportional limiting (or PI limiting) (to bound velocity) and PID limiting (in this case, to bound thrust delta) are applied and the same type of spreadsheet setup is arranged. A plot of a few sample "simulations" is shown in Figure 61.

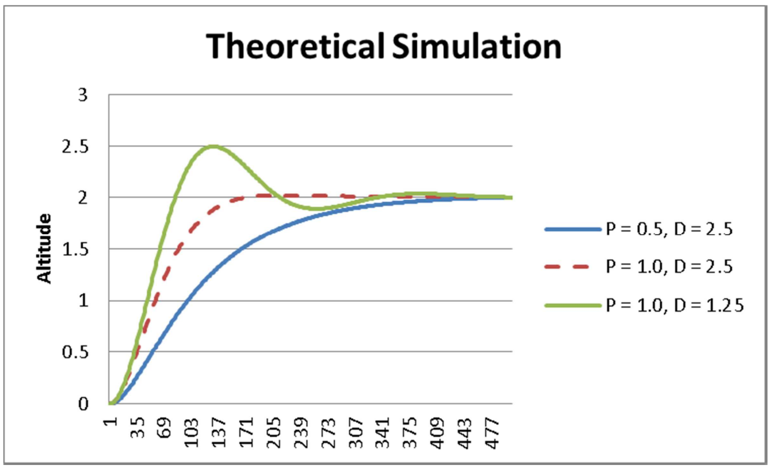


Figure 61 - Theoretical Simulation of Altitude Hold PD parameters

Once again, this chart exhibits the importance of the derivative term in achieving stability; if it is too low, more oscillatory behaviour is to be expected. The proportional term, on the other hand, determines the aggressiveness of the response to setpoint changes and it should be made as large as possible within the constraints of desired system response.

6.3.3.3.2. PID Tuning

The process for tuning the altitude hold PID was again primarily experimental in nature. The theoretical simulations applied above suggested strengthening the derivative term and this was tested. A number of other tests were performed and a selection of results is plotted in Figure 62, Figure 63, and Figure 64.

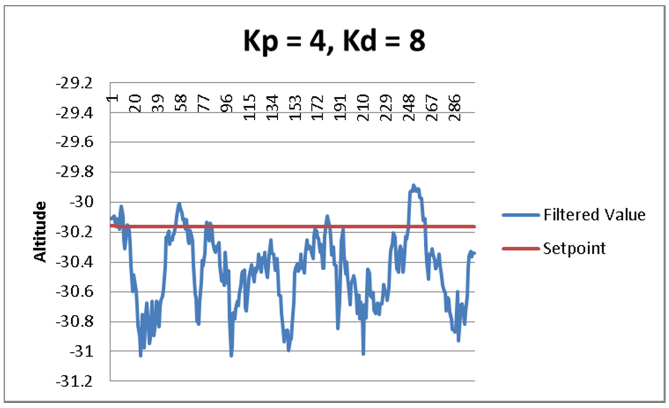


Figure 62 - Jumbo QBot Altitude Hold Testing (K_p = 4, K_d = 8)

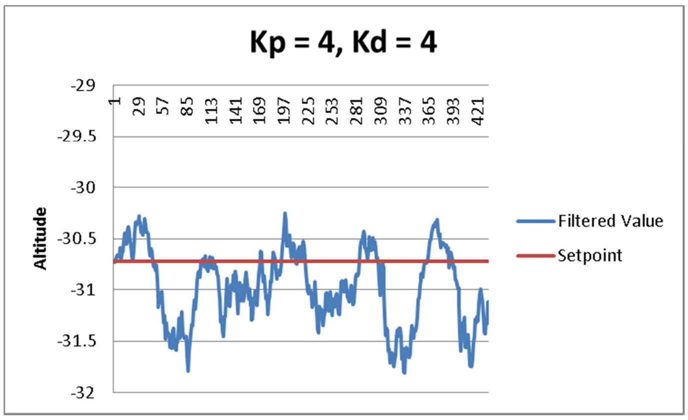


Figure 63 - Jumbo QBot Altitude Hold Testing (K_p = 4, K_d = 4)

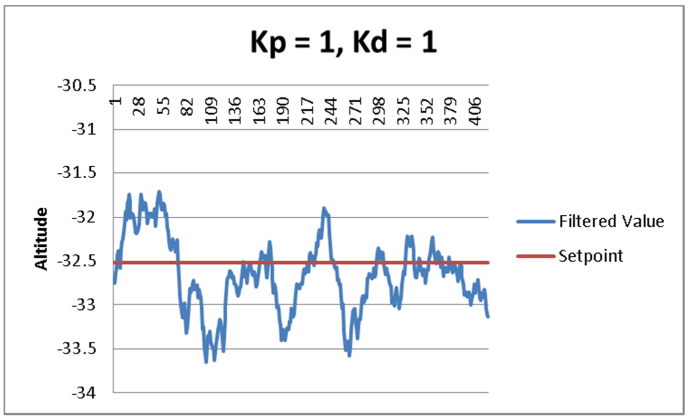


Figure 64 - Jumbo QBot Altitude Hold Testing (K_p = 1, K_d = 1)

These plots all look reasonably similar, but they do not tell the whole story. The filtered altitude reading is representative of the state of the airframe but tends to lag behind it to a significant degree. What is shown in the chart is, therefore, not the actual instantaneous altitude of the robot, but rather an averaged indication of its position. Unfortunately, it falls outside the scope of this research to provide a means of correlating the sensed value to the real state of the airframe. Observation by the controller is sufficient to state that, in the higher value cases (e.g. K_p = 4) the airframe would oscillate much more than indicated by the logged values. Essentially, the robot would drop and then surge back up again before the average reflected the degree of the drop. The lower values saw some up and down drift and compensation, but it was much slower and the filtered values that were captured better reflect the real world behaviour. In the end, the most stable results (from an observer's standpoint, at least) were achieved with values of 1 for both P and D gain terms and this was selected as the final (roughly) tuned parameter set.

6.3.3.4. Full Position Hold

With reasonable values for the position hold PID gain terms corresponding to latitude, longitude, and height/altitude, the robot was tested for autonomous operation. In a representative flight, Jumbo QBot was flown off the ground to a height around 1.5 meters and then autonomous position hold was engaged. Position hold is, of course, overlaid on the orientation hold operation and, when enabled, the robot performs all flight operations independently. A plot of the sensed values (after processing to normalize and, where necessary, convert to meters) from the representative flight is shown in Figure 65.

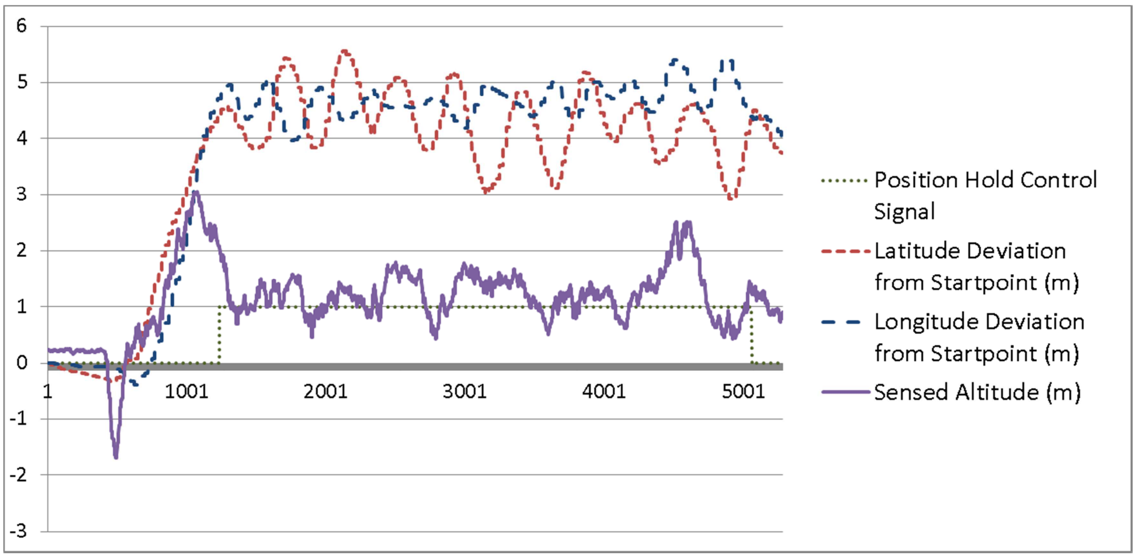


Figure 65 - Position Hold Flight Data

Those values were recorded at 100Hz and this flight, then, took over 50s during which automatic position hold was engaged for more than 35 seconds. It represents the first major achievement in truly independent operation of the developed MAVs.

6.3.4. Fully Autonomous Flight Plan Execution

Position hold is a sufficient form of fully autonomous operation. When the robot takes control of its own orientation and position, the controller and the base station no longer have any role in its flight. It is truly autonomous in all respects at that point, but it is not terribly useful. Simple firmware was therefore written to demonstrate mobility by incrementing position values in a programmable way. This was successfully tested outdoor via a program that increased desired altitude to 8 meters, hovered briefly, and then slowly decreased elevation to bring the robot carefully back to the ground. Figure 66 shows the data recorded during the flight (after normalization and unit conversion), along with the status of the control signal engaging autonomous operation.

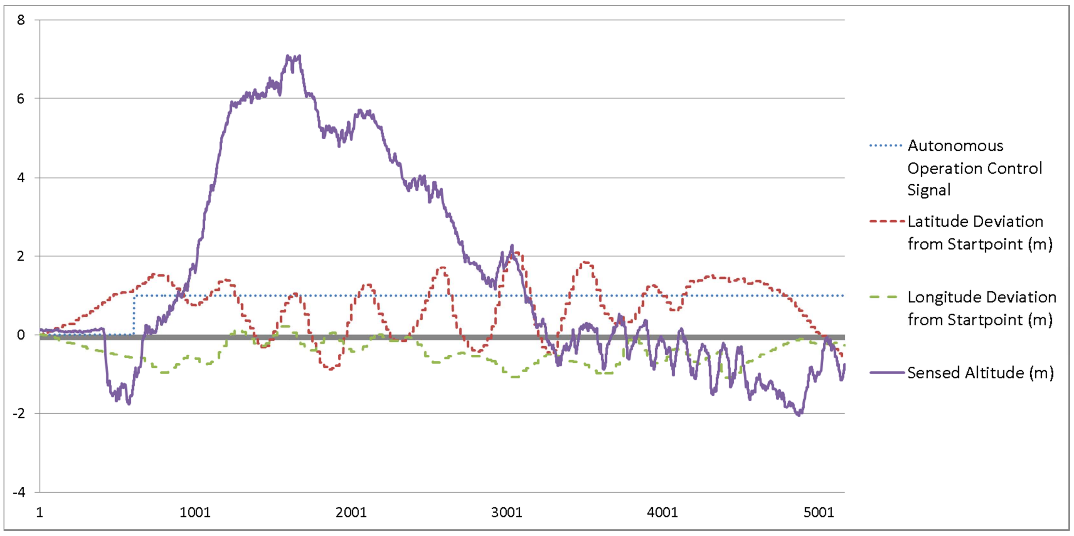


Figure 66 - Autonomous Flight Data

Because of the air pressure disturbance near the ground, the autonomous landing involved a bit of bouncing, but the airframe did eventually settle. That problem has been successfully resolved through indoor testing, but weather and time have limited further outdoor testing as part of this research.

7. Discussion

7.1. Results

Most of the results of this research have been described along with the presented data, but a quick summary is rendered here. At a high-level, three quadrotor MAVs (QBot1, Araqnobot, and Jumbo QBot) have been assembled with necessary electrical, mechanical, and electro-mechanical components and brought to flight readiness. One of those airframes (Jumbo QBot) was designed from the ground up and was constructed entirely as part of this research. A full firmware suite was developed for the 8-bit processors at the core of QBot1 and Araqnobot that includes modules for a number of different peripherals and sensors. Fully autonomous orientation control was implemented in that firmware such that the robots are able to maintain a given orientation without any external input. Base station software was created for the purpose of controlling and testing all three airframes. An embedded Linux image was compiled from online repositories for operation on the Jumbo QBot Gumstix COM; the build was customized and patches were applied to ensure adequate operation of all desired parts of the system. The 8-bit firmware was ported to Linux and expanded to include operational components that the more powerful system enabled (e.g. serial motor control). Finally, fully autonomous operation was achieved on the Jumbo QBot platform that started with position hold and culminated in automated takeoff and landing.

7.2. Issues

In general, successful operations and corresponding data have been discussed and described herein. Many real world issues have been encountered, however, and a few will be discussed here to further enhance the practical value of this research.

In the first place, lithium polymer (LiPo) batteries must be handled with great care. No explosions or fires were caused over the course of this study, but one connector set was partially melted simply through a momentary lack of attention. Furthermore, LiPo battery balancing is critical to longevity and endurance of any given pack. 3-cell batteries were used exclusively for all of the airframes involved in this study and it is important that each cell be charged and discharged at an equal rate to maintain equal voltage between the cells. The 8-bit ATMega processor boards, however, used 2 of the three cells as their primary power source and this caused imbalance over time that needed to be constantly corrected. Whenever possible, all battery cells should be drawn upon to power MAV components so that this imbalance may be avoided. Finally, one large battery pack exhibited imbalance without cause shortly after first use. The cells were rebalanced and the problem ignored, but eventually the plastic wrapping on the pack puffed out, indicating near critical failure of the battery. It was replaced, but the dangerous situation should have been avoided by returning the battery as soon as the first issue was detected.

A significant number of other problems have involved communication. There is an inherent weakness in digital communication (as opposed to the analog control typically employed by radio control hobbyists) in that it is typically all or nothing. Either a command is received correctly, or not at all, or possibly with some amount of corruption that renders it entirely different from its intended value. This can cause all kinds of havoc for the control of an aircraft with four high-speed rotors spinning in the air. Checksums were implemented from the host to the robot to help ensure accurate reception, but end-to-end integrity and guaranteed delivery were not built into the protocol. For the most part, this has worked to a sufficient degree, but remote control is not as robust as might be desired.

Another communication issue has been experienced with Jumbo QBot and it involves the wireless serial link randomly (and without warning) stopping operation. This has only happened twice in actual flight but it is currently unexplained (it is considered likely to be a kernel driver issue) and it caused airframe inversion and at least one broken propeller in both cases.

Safety has been an ongoing concern during the testing and development of these potentially dangerous machines. Of particular concern has been the lack of a failsafe to disconnect battery power in the event of an electrical short, a fire, motor controller insanity, or just total loss of control. There was one incident where such a failsafe would have been useful as one motor was running at high speed while its propeller was hitting a rigid wooden surface; the friction almost started a fire and the only solution was to manually pull the battery plug. This will be addressed in the future through the addition of a battery interface board that has interruption relays that can be triggered remotely. It will therefore be possible to cut battery power from the base station even if the control board has been damaged or gone crazy in some unexpected way.

QBot1 and Araqnobot were tested incrementally and in tightly controlled ways. As such, crashes rarely happened and no propellers were broken on those airframes over the course of this research. Jumbo QBot experienced a much higher rate of destructive accidents. This is largely due to 2 significant factors: increased complexity of the design, and a reduction in operator control as autonomous functions were developed. Among the problems encountered were a faulty serial communication cable, a motor controller that failed and caught fire, a motor controller transistor that went full open and burned out the motor, and an autonomous elevation into the ceiling. The aftermath of events like these is shown in Figure 67. Some of the accidents may have been avoidable, but the pursuit of autonomy with a large airframe hereby comes with two pieces of advice: 1) be very careful, and 2) stock up on propellers.

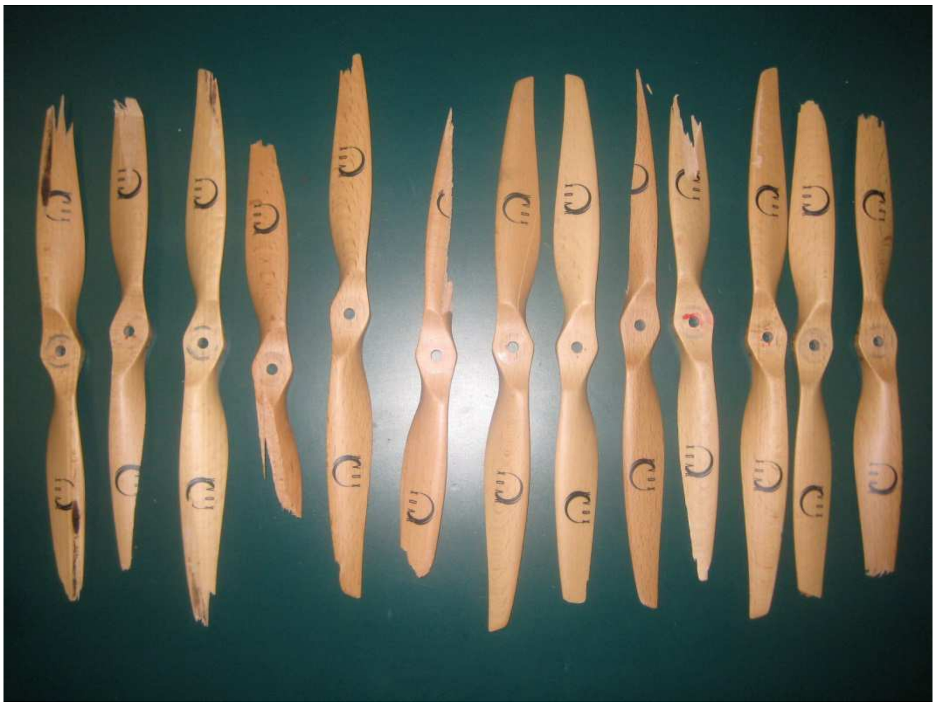


Figure 67 - Broken Jumbo QBot Propellers

Finally, battery life is the primary endurance issue for MAVs of this type. Range and flight duration are greatly determined by the availability of lightweight, long-life batteries. LiPo's have come a long way in recent years, but to see quadrotor flights exceeding an hour's duration with today's motors would require tripling battery life with no increased mass. For the purposes of this research, the duration of testing was continuously constrained by battery life and that's an ongoing reality for development in this field. A breakthrough in either motor or battery technology (or both) will be a factor in the further commercialization of multi-rotor MAVs.

7.3. Future Work

In many ways, the outcome of this research has been a platform for further research. There is so much that remains to be refined and so much more that the Jumbo QBot, especially, is capable of. Future work can be done to take advantage of its extensive computing power and an array of sensors and attachments can be tested thanks to its significant payload capacity. As such, there are many things that follow naturally from the successful concluding point of this research.

Firstly, the communication framework could be rebuilt to switch from clear text to binary. As the base station has evolved, the command line has been used less and less and it now exists almost exclusively for legacy reasons. In addition to binary data transfer, a more robust integrity system could be developed (expanding upon simple checksum and maybe employing inner and outer checksums, or something like that). Then explicit acknowledgement and automatic retries could be added to increase reliability of remote control and status messages.

All of the airframes exhibit some amount of twitchiness that comes from IMU noise. It would be useful to spend some time analysing the source of that noise and trying to isolate the IMUs from potential sources of disturbance like vibration. Other alternatives such as filtering of either the IMU input or the motor output could also be explored (smoothing the motor control could also result in an improvement of power efficiency).

Both QBot1 and Araqnobot should be capable of indoor altitude hold with a decent rangefinder sensor. Laser or infrared sensors should be tested to find a functional alternative (or, perhaps, enhancement) to sonar. Presuming a reliable sensor can be found, further localization could be added to all of the MAVs by mounting the sensors to detect objects horizontally and that would allow fully autonomous indoor operation (e.g. maintaining static position relative to walls).

Jumbo QBot already has mounting hardware and available USB ports that will allow the attachment of a 240 degree scanning laser rangefinder. That device should be installed and firmware developed to perform intelligent indoor navigation, obstacle avoidance, and environment mapping.

The position hold logic obviously works, but it is currently understood to be rudimentary. The approach and equations presented in Section 3.4 are acknowledged to be somewhat basic. A desire for minimal sufficiency should not supplant a rigorous approach to engineering and it was only due to time constraints that a primarily experimental approach was taken in this area. A more comprehensive modelling and control framework needs to be developed. Furthermore, the control parameters should be tested and tuned much more extensively and other control methods (such as feed forward) should be evaluated. It is believed to be possible to achieve smooth position hold without the constant oscillations or adjustments observed in the current system.

Finally, the base station software should be either reworked or set aside in favour of something more suitable to navigation and flight. The current framework is focused on PID tuning and evaluation which will become less important as the system evolves (PID tuning should perhaps be moved to a secondary C# form). A more conventional waypoint planning interface with graphical telemetry information should be developed for more extensive flying.

8. Conclusion

It can clearly be said that this research has demonstrated the construction of a fully autonomous quad-rotor MAV of non-trivial size (almost 1 meter from rotor-tip to rotor-tip and weighing over 2kg), payload capacity (theoretically up to 2kg), and computing power. This was achieved through an evolutionary approach that involved a progression of three physical robots that increased in complexity and capability at each stage. Fully autonomous orientation control was implemented on all three MAVs and the final MAV, Jumbo QBot, flew to a height over 9m and also implemented autonomous position control and flight plan execution from takeoff to landing. All theory and control was based on a set of equations and models that align well with intuition and that have both the elegance of minimal sufficiency and the validation of demonstrated effectiveness.   The theory, the development process, the components, and the experimental results have been practically and thoroughly presented herein.

The embedded computer system on Jumbo QBot runs a fully capable version of the Linux operating system and the flight program requires only a small amount of the available processing power (less than one third) and memory (less than 1%). Considering this and the available payload capacity of the flight system, it is evident that this research has established a platform for a broad range of future work, research, and exploration. The sky is not the limit; it is just the beginning.

9. Bibliography

Åström, K. J., & Hägglund, T. (2006). *Advanced PID control.* Research Triangle Park: ISA - Instrumentation, Systems, and Automation Society.

AutoQuad. (2012, June 25). *ESC32 2r1 user manual - version 0.0.* Retrieved from http://autoquad.org/wp-content/uploads/downloads/2012/07/esc32_v_0-Copy.pdf

Bolton, W. (1998). *Control engineering* (2nd ed.). New York, NY: Addison Wesley Longman Publishing.

Bosch Sensortec. (2012, January 27). *BMP180 digital pressure sensor.* Retrieved from http://ae-bst.resource.bosch.com/media/products/dokumente/bmp180/bst-bmp180-ds000-08.pdf

Bouabdallah, S., Noth, A., & Siegwart, R. (2004). PID vs LQ control techniques applied to an indoor micro quadrotor. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, *3*, pp. 2451-2456. Sendai, Japan. doi:10.1109/IROS.2004.1389776

Brandt, J. B., & Selig, M. S. (2011). Propeller performance data at low reynolds numbers. *AIAA Paper 2011-1255.* Orlando, FL: 49th AIAA Aerospace Sciences Meeting. doi:10.2514/6.2011-1255

CH Robotics. (2013, February 21). *UM6 ultra-miniature orientation sensor datasheet.* Retrieved from http://www.chrobotics.com/docs/UM6_datasheet.pdf

Colton, S. (2007, June 25). *The balance filter.* Retrieved from MIT - Massachusetts Institute of Technology: http://web.mit.edu/scolton/www/filter.pdf

Computer Support Group, Inc. (2011, June 29). *Length of a degree of latitude And longitude calculator*. Retrieved from http://www.csgnetwork.com/degreelenllavcalc.html

Dierks, T., & Jagannathan, S. (2010, January). Output feedback control of a quadrotor UAV using neural networks. *IEEE Transactions on Neural Networks*, *21*(1), 50-56. doi: 10.1109/TNN.2009.2034145

Domingues, J. M. (2009, October). *Quadrotor prototype.* Unpublished Master's Thesis, University Tecnicia de Lisboa, Mexico City. Retrieved from https://dspace.ist.utl.pt/bitstream/2295/574042/1/Tese_de_Mestrado.pdf

Ellis, S. (2012a, October 2). *Did they ever solve the slow bring up of wifi problem?* Retrieved April 27, 2013, from Gumstix Mailing List Archive: http://gumstix.8.x6.nabble.com/Did-they-ever-solve-the-slow-bring-up-of-wifi-problem-td4944197i20.html

Ellis, S. (2012b, March 23). *Gumstix kernel development*. Retrieved April 27, 2013, from http://www.jumpnowtek.com/?option=com_content&view=article&id=46&Itemid=54

Ellis, S. (2012c, October 17). *Using the gumstix onboard ADCs*. Retrieved April 27, 2013, from http://www.jumpnowtek.com/index.php?option=com_content&view=article&id=79&Itemid=91

European Commission. (2009, July 30). *Mega threats, micro solutions*. Retrieved from http://www.euronews.com/2009/07/30/mega-threats-micro-solutions/

Gumstix Inc. (2013). *Create a bootable microSD card*. Retrieved April 27, 2013, from Gumstix Developer Center: http://gumstix.org/create-a-bootable-microsd-card.html

Gumstix, Inc. (2012). *Overo FE COM*. Retrieved May 15, 2013, from https://www.gumstix.com/store/product_info.php?products_id=256

Horgan, J. (2013, March). The drones come home. *National Geographic*, pp. 122-135.

Hutchings, B. (2011, December 07). *Lockdep, bug: exclude TAINT_OOT_MODULE from disabling lock debugging*. Retrieved April 27, 2013, from https://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/commit/?id=9ec84acee1e221d99dc33237bff5e82839d10cc0

Keane, B. (2012, August 24). *Using ADCIN on 3.0 kernel - no readings for ADC3-6*. Retrieved April 27, 2013, from Gumstix Mailing List Archive: http://gumstix.8.x6.nabble.com/using-ADCIN-on-3-0-kernel-no-readings-for-ADC3-6-td640585.html

LOCOSYS Technology Inc. (2009, February 2). *Datasheet of stand-alone GPS smart antenna module, LS20036.* Retrieved from http://www.locosystech.com/download.php?fid=116

Measurement Specialties. (2011, August 09). *AN520 C-code example for MS56xx, MS57xx (except analog sensor), and MS58xx series pressure sensors.* Retrieved from http://www.meas-spec.com/downloads/C-Code_Example_for_MS56xx,_MS57xx_(except_analog_sensor)_and_MS58xx_Series_Pressure_Sensors.pdf

Measurement Specialties. (2013, February 19). *MS5803-01BA miniature variometer module.* Retrieved from http://www.meas-spec.com/downloads/MS5803-01BA.pdf

Microstrain, Inc. (2012, December 14). *3DM-GX3-45 data communications protocol.* Retrieved from http://files.microstrain.com/3DM-GX3-45-Data-Communications-Protocol.pdf

Miller, K. (2008, July 31). *Path tracking control for quadrotor helicopters.* Retrieved from http://www.eecs.berkeley.edu/Programs/ugrad/superb/papers%202008/Katie%20Miller.pdf

MK2832/35. (2009, November 16). Retrieved May 17, 2013, from Mikrokopter Shop: https://www.mikrocontroller.com/index.php?main_page=product_info&products_id=443&language=en

Moaveni, S. (2011). *Engineering Fundamentals: An Introduction to Engineering.* Stamford, CT: Cengage Learning.

Nesbitt, B. (2012, July 18). *Esc32 - 32 bit ARM brushless DC motor electronic speed controller - google project hosting*. Retrieved April 29, 2013, from http://code.google.com/p/esc32/source/browse#svn%2Ftrunk%2Fground

Portland State Aerospace Society. (2004, December 22). *A quick derivation relating altitude to air.* Retrieved from http://psas.pdx.edu/RocketScience/PressureAltitude_Derived.pdf

Pounds, P., Mahoney, R., & Corke, P. (2010). Modelling and control of a large quadrotor robot. *Control Engineering Practice, 18*(7), 691 - 699. doi:10.1016/j.conengprac.2010.02.008

Rabyniuk, C. (2013, April 29). *Halton police drone: high-tech surveillance.* Retrieved from http://oakvillenews.org/drones-in-canada/

Robbe ROXXY 2827-35. (2009, December 1). Retrieved May 17, 2013, from Mikrokopter Shop: https://www.mikrocontroller.com/index.php?main_page=product_info&cPath=73&products_id=452

Salih, A., Moghavvemi, M., Mohamed, H. A., & Gaeid, K. S. (2010, December 4). Flight PID controller design for a UAV quadrotor. *Scientific Research and Essays, 5*(23), 3660-3667. doi:10.1109/AQTR.2010.5520914

Sanca, A. S., Alsina, P. J., & Cerqueira, J. d. (2008). Dynamic modelling of a quadrotor aerial vehicle with nonlinear inputs. *Robotic Symposium, 2008. LARS '08. IEEE Latin American*, (pp. 143-148). Natal, Brazil. doi:10.1109/LARS.2008.17

Sangyam, T., Laohapiengsak, P., Chonghcharoen, W., & Nilkhamhang, I. (2010). Autonomous path tracking and disturbance force rejection of UAV using fuzzy based auto-tuning PID controller. *2010 International Conference on Electrical Engineering/Electronics Computer Telecommunications and Information Technology*, (pp. 528-531). Chiang Mai, Thailand.

Schmidt, M. D. (2011). *Simulation and control of a quadrotor unmanned aerial vehicle.* Unpublished Master's Thesis, University of Kentucky, Lexington, KY.

Scorpion Power System Ltd. (n.d.). *Scorpion SII-3008-1090 Prop Data*. Retrieved November 25, 2012, from http://www.scorpionsystem.com/files/i1,070_data_chart.htm

Scorpion Power System. (2013). *Scorpion SII-3008-1090KV (V2).* Retrieved May 17, 2013, from http://www.scorpionsystem.com/catalog/motors/s30_series_v2/SII-3008-1090KV/

Sutharoj, P. (2006, June). *Flying robot helps farmers avoid dangerous chemicals.* Retrieved from The Nation: http://www.nationmultimedia.com/2006/06/19/business/business_30006731.php

UAVs. (2012, April 12). Retrieved from http://www.homelandsecuritynewswire.com/dr20120412-worldwide-uav-market-to-reach-more-than-94-billion-in-ten-years

VCSKicks. (2007, December 31). *Euler rotation - codeproject*. Retrieved April 30, 2013, from http://www.codeproject.com/Articles/22577/Euler-Rotation

Veness, C. (2012). *Calculate distance, bearing and more between latitude/longitude points*. Retrieved May 14, 2013, from http://www.movable-type.co.uk/scripts/latlong.html

10. Appendix A – Video Links

- QBot1 Three DOF Test Stand – Tuned PID
    - http://youtu.be/jHiF4AiMh_8
- QBot1 Tethered Flight
    - http://youtu.be/mxvZeuJDQTo
- Araqnobot Indoor Flight
    - http://youtu.be/oqj76vawkwQ
- Araqnobot Outdoor Flight
    - http://youtu.be/EtJOVm9ToQo
- Introduction to Jumbo QBot
    - http://youtu.be/pgo3tFJHrus
- Jumbo QBot Early Indoor Flight
    - http://youtu.be/Pqh2zNGI2A4
- Jumbo QBot First High Flight
    - http://youtu.be/g4JKEWTOfws
- Jumbo QBot Over 9m Flight
    - http://youtu.be/p-8s4_YoEP4
- Jumbo QBot Autonomous Flight Testing
    - http://youtu.be/H0hsqsCxOUI
- Fully Autonomous Takeoff and Landing
    - http://youtu.be/74n0M5eEJBI
- Improved Autonomous Landing (Indoor)
    - http://youtu.be/dQgLupYWX-s