University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2011

SIMULATION AND CONTROL OF A QUADROTOR UNMANNED AERIAL VEHICLE

Michael David Schmidt
University of Kentucky, mdschm2@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Schmidt, Michael David, "SIMULATION AND CONTROL OF A QUADROTOR UNMANNED AERIAL VEHICLE" (2011). *University of Kentucky Master's Theses*. 93.
https://uknowledge.uky.edu/gradschool_theses/93

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

SIMULATION AND CONTROL OF A QUADROTOR UNMANNED AERIAL VEHICLE

The ANGEL project (Aerial Network Guided Electronic Lookout) takes a systems engineering approach to the design, development, testing and implementation of a quadrotor unmanned aerial vehicle. Many current research endeavors into the field of quadrotors for use as unmanned vehicles do not utilize the broad systems approach to design and implementation. These other projects use pre-fabricated quadrotor platforms and a series of external sensors in a mock environment that is unfeasible for real world use. The ANGEL system was designed specifically for use in a combat theater where robustness and ease of control are paramount. A complete simulation model of the ANGEL system dynamics was developed and used to tune a custom controller in MATLAB and Simulink®. This controller was then implemented in hardware and paired with the necessary subsystems to complete the ANGEL platform. Preliminary tests show successful operation of the craft, although more development is required before it is deployed in field. A custom high-level controller for the craft was written with the intention that troops should be able to send commands to the platform without having a dedicated pilot. A second craft that exhibits detachable limbs for greatly enhanced transportation efficiency is also in development.

Keywords: Quadrotor, ANGEL, Unmanned Aerial Vehicle, Control, Simulation

Michael David Schmidt

04/18/2011

SIMULATION AND CONTROL OF A QUADROTOR UNMANNED AERIAL VEHICLE

By

Michael David Schmidt

Dr. Bruce Walcott
Director of Thesis

Dr. Stephen Gedney
Director of Graduate Studies

04/18/2011

RULES FOR USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

THESIS

Michael David Schmidt

The Graduate School

University of Kentucky

2011

SIMULATION AND CONTROL OF A QUADROTOR UNMANNED AERIAL VEHICLE

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Electrical Engineering
in the College of Engineering
at the University of Kentucky

By

Michael David Schmidt

Lexington, Kentucky

Director: Dr. Bruce Walcott, Professor of Electrical Engineering

Lexington, Kentucky

2011

Copyright © Michael David Schmidt 2011

Table of Contents

List of Tables: vi
List of Figures: vii
Section I: Introduction 1
UAV Historical Perspective and Applications 1
Vertical Take-off and Landing (VTOL) Aircraft 2
The Quadrotor 4
Section II: Literature Review and Motivation 5
The Cutting Edge 5
Commercial Products 6
Research Motivation 7
Section III: ANGEL Simulation Model 8
Introducing the Aerial Network Guided Electronic Lookout (ANGEL) 8
Coordinate Systems 11
ANGEL System State 12
ANGEL Actuator Basics 13
Coordinate System Rotations 15
ANGEL Body Forces and Moments 15
ANGEL Moments of Inertia 20
ANGEL Kinematics and the Gimbal Lock Phenomenon 23
The Quaternion Method 25
MATLAB Simulation of ANGEL 26
Section IV: ANGEL Control Development 32
Control Fundamentals 32
Model Simplifications 35
Input Declarations 37
MATLAB Control Implementation 37
Controller Tuning and Response 41
Section V: Platform Implementation 46
ANGEL v1 Platform Basics 46
ANGEL v1 Power System 49
ANGEL Actuators (ESC/Motor/Propeller) 51

ANGEL Main Avionics 54
ANGEL v1 Avionics Loop Description 55
ANGEL Sensors 56
Sensor Fusion Algorithm and Noise 59
ANGEL User Control (Xbee and Processing GUI) 62
Control Library Implementation 65
ANGEL v2 Build Description 67
Section VI: Testing and Results 69
Testing and Results Introduction 69
Test Bench and Flight Harness Construction 69
Thrust Measurement 71
Pitch and Roll Test Data 72
Pitch and Roll Test Results 74
Avionics Loop Testing 74
Section VII: Concluding Remarks and Future Development 75
Simulation Conclusions and Future Work 75
Controller Conclusions and Future Work 75
Sensors and Fusion Algorithm Conclusions and Future Work 76
User Interface Conclusions and Future Work 77
Physical Build Conclusions and Future Work 78
Thesis Objective Conclusion 78
APPENDIX A – CODE 80
A-1 – System Dynamics MATLAB Code used in Simulation 80
A-2 – Code for Attitude Control in MATLAB Simulation 81
A-3 – Block to translate controller outputs to speed inputs 83
A-4 – Disabled Altitude Control Block 83
A-5-1 Arduino Motor Library (QuadMotor.h) 84
A-5-2 – Arduino Motor Library (QuadMotor.cpp) 85
A-6-1 – Arduino PID Library (SchmidPID.h) 87
A-6-2 – Arduino PID Library (SchmidtPID.cpp) 87
A-7-1 – Arduino IMU Sensor Library (IMU.h) 89
A-7-2 – Arduino IMU Sensor Library (IMU.cpp) 90

A-7-3 – Arduino IMU Sensor Library Example (Processing) 93
A-8 – Arduino Main Avionics Loop..... 95
A-9 – Processing Controller Code 102
APPENDIX B – CAD Schematics..... 112
B-1 – ANGEL v1 Junction Drawing 112
B-2 – Uriel Arm Junction Model (no dimensions) 113
B-3 – Motor Mount for Uriel (no dimensions) 113
B-3 – Large Uriel Assembly Diagram..... 114
REFERENCES 115
VITA..... 118

List of Tables:

Table 1: VTOL Vehicles 3
Table 2: Aircraft Comparisons..... 5
Table 3: Quadrotor differential thrust examples 14
Table 4: P, I, D values for Roll 42
Table 5: Properties of Thermoplastic and PVC..... 48
Table 6: Battery Parameters..... 50
Table 7: ESC Settings for ANGEL platform..... 54

List of Figures:

Figure 1: Perley's bomber in 1863 [1] 1
Figure 2: Nazi V-1 bomber [1] 1
Figure 3: Draganflyer X8 from Draganfly Innovations 6
Figure 4: Parrot AR Drone Quadrotor Toy 7
Figure 5: DoD ConOps for ANGEL system 9
Figure 6: ANGEL v1. Note the propellers are removed. 10
Figure 7: ANGEL v2 'Uriel' 10
Figure 8: Body and Earth frame axes with corresponding flight angles 12
Figure 9: Zero roll/pitch thrust force 17
Figure 10: 40° roll angle thrust force 17
Figure 11: Motor numbers and axes for inertia calculations 21
Figure 12: Gimbal Lock Phenomenon. Photo from HowStuffWorks.com 24
Figure 13: ANGEL Simulation Block Diagram 27
Figure 14: Actuator Signal Input for Pitch Forward Attempt 28
Figure 15: Flight path under single actuator change 28
Figure 16: Roll, Pitch and Yaw angles for single actuator simulation 29
Figure 17: New Yaw Compensating Input Signal 30
Figure 18: Correct flight path with updated signals 30
Figure 19: Roll/Pitch/Yaw graphs with updated input signals 31
Figure 20: Simulink model of ANGEL simulation 32
Figure 21: Centrifugal Governor 33
Figure 22: Generic Control System 34
Figure 23: Controller Sub Block 40
Figure 24: Roll axis ultimate gain oscillations 41
Figure 25: Parallel Form Gain Response 42
Figure 26: Standard Form Gain Response 43
Figure 27: Full simulation and controller model 44
Figure 28: Controller example input signals 44
Figure 29: Roll, Pitch and Yaw response to the input signals 45
Figure 30: Controller output signals 46
Figure 31: ANGEL v1 Hub 47
Figure 32: ANGEL v1 Subsystem Location 49
Figure 33: Power Distribution on ANGEL v1 51
Figure 34: Brushless Outrunner Motor 52
Figure 35: Motor Mount 52
Figure 36: Propeller Balancer 53
Figure 37: Example of how accelerometers measure force 57
Figure 38: ANGEL Controller GUI 64
Figure 39: CAD Diagram and 3D model of Uriel build 68
Figure 40: Dean-Y connectors made for Uriel 69
Figure 41: Roll and Pitch Axis Test Bench 70
Figure 42: Flight Harness 71
Figure 43: Thrust Stand 72
Figure 44: Roll Axis Test Results 73
Figure 45: Pitch Axis Test Results 73

Section I: Introduction

UAV Historical Perspective and Applications

Recent military conflicts have put the development of unmanned systems as combat tools in the global spotlight. The proliferation of unmanned aerial vehicles (UAVs) has been of particular interest to the mainstream media. While the impact of these systems may be new to some, their use has roots in conflict dating back to the Civil War. Pre-aviation UAVs, such as Perley's aerial bomber (Figure 1), were generally nothing more than floating payloads with timing mechanisms designed to drop explosives in enemy territory. With limited technological resources available at the time, most pre-aviation UAV endeavors proved too inaccurate to achieve widespread success.

In 1917, the combat potential of UAVs was finally realized with varying designs of aerial torpedoes. Although WWI ended before any deployable UAVs were used in theater, the push towards successful military integration had already begun. The British Royal Navy developed the Queen Bee in the 1930's for aerial target practice. During WWII, Nazi Germany extensively used the feared V-1 UAV (Figure 2) to bomb nonmilitary targets. The work towards eliminating the threat of the V-1 proved to be the beginnings of post-war UAV development for the United States. During the 1960s, surveillance drones were used for aerial reconnaissance in Vietnam, and the 1980s saw wide integration of several Israeli Air Force UAVs into the US fleet design [1].

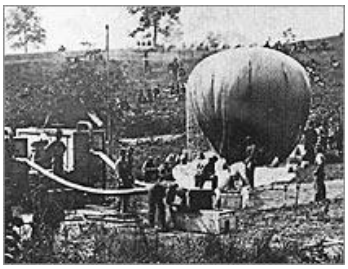


Figure 1: Perley's bomber in 1863 [1]

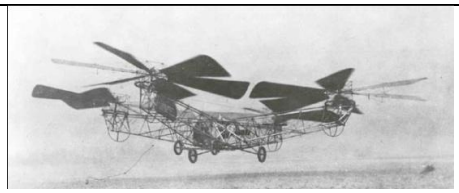
Figure 2: Nazi V-1 bomber [1]

After Operation Desert Storm, UAV development boomed in the United States. Current market studies estimate that worldwide UAV spending will more than double during the next 10 years, from $4.9 billion to $11.5 billion annually. This amounts to a total expenditure of just over $80 billion over the next decade [2]. While a large percentage of this spending will be for defense and aerospace applications, non-military use of UAVs has also increased. These include such practices as pipeline/powerline inspection, border patrol, search and rescue, oil/natural gas searches, fire prevention, topography and agriculture [3].

Vertical Take-off and Landing (VTOL) Aircraft

VTOL aircraft provide many benefits over conventional take-off and landing (CTOL) vehicles. Most notable are the abilities to hover in place and the small area required for take-off and landing. VTOL aircraft include conventional helicopters, other craft with rotors such as the tiltrotor, and fixed-wing aircraft with directed jet thrust capability. The two desirable benefits of VTOL aircraft make them especially useful for aerial reconnaissance, asset tracking, munitions delivery, etc. Table 1 shows a small sample of VTOL craft.

Table 1: VTOL Vehicles

	Westland Apache WAH-64D Longbow Helicopter Manned Vehicle Single Rotorcraft
	Schiebel Camcopter S-100 Unmanned Vehicle Single Rotorcraft
	McDonnell Douglas AV-8B Harrier II Manned Vehicle V/STOL (Vertical/Short Take-off and Landing) Directed Thrust Jet
	Bell-Boeing V-22 Osprey Manned Vehicle V/STOL Tiltrotor
	De Bothezat Quadrotor, 1923 Manned Vehicle VTOL Four rotor rotorcraft (Quadrotor)

The main disadvantage of VTOL vehicles, especially rotorcraft, are the increased complexity and maintenance that comes with the intricate linkages, cyclic control of the main rotor blade pitch, collective control of the main blade pitch, and anti-torque control of the pitch of the tail rotor blades.

The Quadrotor

The quadrotor is considered an effective alternative to the high cost and complexity of standard rotorcraft. Employing four rotors to create differential thrust, the craft is able to hover and move without the complex system of linkages and blade elements present on standard single rotor vehicles. The quadrotor is classified as an *underactuated* system. This is due to the fact that only four actuators (rotors) are used to control all six degrees of freedom (DOF). The four actuators directly impact z-axis translation (altitude) and rotation about each of the three principal axes. The other two DOF are translation along the x- and y-axis. These two remaining DOF are coupled, meaning they depend directly on the overall orientation of the vehicle (the other four DOF). Additional quadrotor benefits are swift maneuverability and increased payload. Drawbacks include an overall larger craft size and a higher energy consumption, which generally means lower flight time. [4] subjectively compares different types of VTOL miniature flying robots (MFR) in several categories.

Table 2: Aircraft Comparisons

Categories	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Power Cost	2	2	2	2	1	4	3	3
Control Cost	1	1	4	2	3	3	2	1
Payload/volume	2	2	4	3	3	1	2	1
Maneuverability	4	2	2	3	3	1	3	3
Mechanics Simplicity	1	3	3	1	4	4	1	1
Aerodynamics Complexity	1	1	1	1	4	3	1	1
Low Speed Flight	4	3	4	3	4	4	2	2
High Speed Flight	2	4	1	2	3	1	3	3
Miniaturization	2	3	4	2	3	1	2	4
Survivability	1	3	3	1	1	3	2	3
Stationary Flight	4	4	4	4	4	3	1	2
TOTAL	24	28	32	24	33	28	22	24

A=Single Rotor, B=Axial Rotor, C=Coaxial Rotor, D=Tandem Rotors, E=Quadrotor, F=Blimp, G=Bird-like, H=Insect-like. 1=Poor, 4=Excellent [4]

As is seen in Table 2, the quadrotor configuration provides many advantages in the quest for an achievable and usable UAV as a VTOL MFR. This thesis aims to further explore the modeling and simulation of a quadrotor vehicle with focus on good mechanical design and robust control system implementation.

Section II: Literature Review and Motivation

The Cutting Edge

Quadrotor research is a very popular area, especially in the academic setting. Arguably at the head of quadrotor research is UPenn's GRASP (General Robotics, Automation, Sensing and Perception) Lab. GRASP is currently pushing the envelope with aggressive quadrotor maneuvering and detection/avoidance algorithms, which allows the vehicle to accomplish such feats as autonomously flying through a moving object, such as a thrown

hoop. Their other research involves swarm-based task management, where individual quadrotor vehicles cooperate to lift heavy payloads. Additional areas of research include perching and landing algorithms, which allow the vehicles to grip onto abnormal landing surfaces [6]. It should be noted that most of the test bed vehicles from UPenn are bought commercially and tracked with an external Vicon® Motion Capture System to have complete position and orientation information for the vehicle.

Commercial Products

In addition to the highly scientific and technical research being performed on quadrotor systems, they have a strong footing in the commercial market as well. Draganfly Innovations [16] has a large section of the quadrotor market locked down for the industrial sector with their line of Draganflyer helicopter systems (comprised of tri-, quad-, and octo- rotor setups). Figure 3 shows the popular 8 rotor Draganflyer X8, used for aerial surveillance.

Figure 3: Draganflyer X8 from Draganfly Innovations

While this line of aerial surveillance robots offers many attractive features such as a folding frame, robust chassis design and high payload capacity for attaching several different camera or surveillance packages, they still require the use and training of a proprietary controller.

Where Draganfly is at the top of the list for the commercial/industrial market, the Parrot AR Drone [17] has a strong footing in the toy market (shown in Figure AB). This drone achieves an extremely light weight through its foam outer shell while maintaining good aerodynamics. It is controlled via an iPhone/iPod Touch using the built in accelerometers to deliver pitch and roll commands wirelessly. Two cameras feed back to the controller, allowing the user to navigate remotely. The light weight of the craft does not make it suitable for a medium or high disturbance environment, and the weight reductions mean a smaller battery capacity which directly affects the flight time of the platform.

Figure 4: Parrot AR Drone Quadrotor Toy

Research Motivation

The motivation for the research in this thesis builds on the previous work discussed above in quadrotor research. While each of these systems provide an important component of the bigger picture (high tech research, usable commercial product, fun and inexpensive toy), none of them provide a

full systems engineering approach to the problem of usability in a combat theater. The research presented here is the first step towards a more complete understanding of the quadrotor as a dynamic system. Although much of the work presented has been completed or overcome before, working through it personally while keeping in mind the end goal of a troop usable system has uncovered problems not addressed in the previous endeavors. Relying on external sensing systems or complex controllers and disregarding flight time and platform weight may still result in a usable system, as is evident from the commercial and academic successes listed previously. But by tackling the problem with a fresh set of objectives, this thesis aims to correct those inadequacies and offer solutions and alternatives in response to the development and testing of a new platform.

Section III: ANGEL Simulation Model

Introducing the Aerial Network Guided Electronic Lookout (ANGEL)

Before diving into the kinematics and simulation that describe how a quadrotor system acts in flight, a brief introduction to the specific system we are using is necessary. The ***A****erial* ***N****etwork* ***G****uided* ***E****lectronic* ***L****ookout* (***ANGEL***) platform was developed at the University of Kentucky with funding from a Department of Defense grant through the UK Center for Visualization. The platform was intended as a man portable, MAV (Micro Air Vehicle) capable of short range reconnaissance through a variety of sensor subsystems. Additionally, the vehicle was to be controllable only at a high level in order to allow ground forces to focus their attention elsewhere. This "set-and-forget" mentality is something majorly different than most UAVs deployed today, as they require constant attention from a ground station based pilot. See Figure 5 for the DoD concept of operations diagram. Two versions of the platform were developed and built. Regrettably, the funding cycle for the grant ended mid-build, and further platform development has been placed on hold until a suitable source of funding is found. This, however, has not impeded development of the simulation model or testing of

the control algorithms, which will be covered later in this paper. Figure 6 illustrates the first unnamed version of the ANGEL platform, and Figure 7 shows the much-improved second version, named 'Uriel'. More information on the builds and features are found in the Platform Builds section.

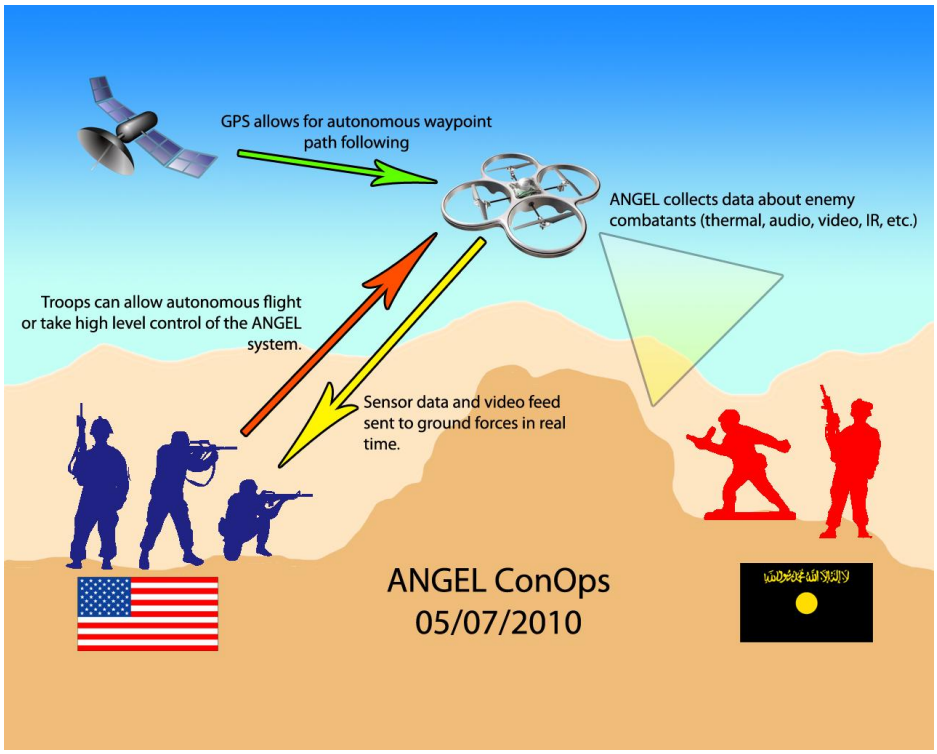


Figure 5: DoD ConOps for ANGEL system

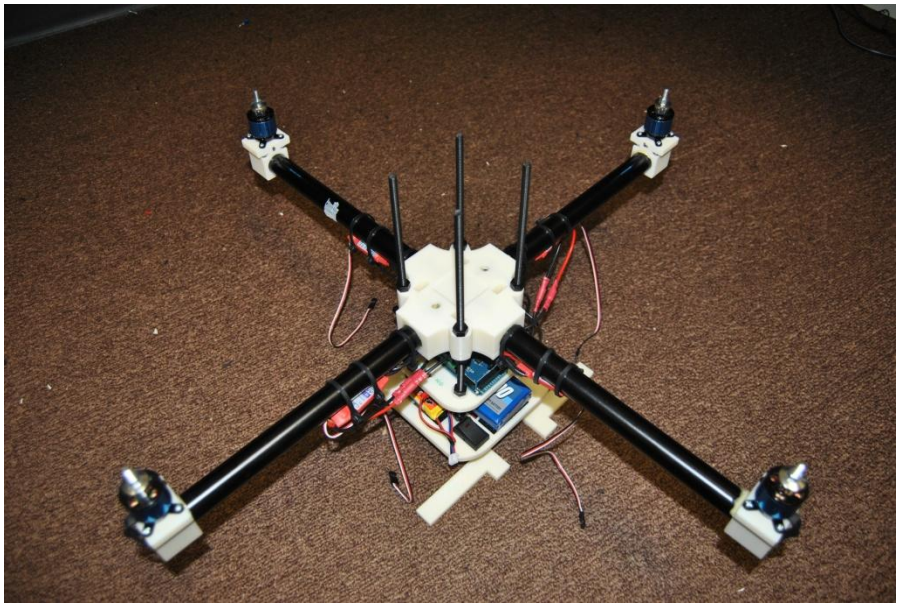


Figure 6: ANGEL v1. Note the propellers are removed.

Figure 7: ANGEL v2 'Uriel'

Coordinate Systems

Unlike conventional rotorcrafts that use complex mechanisms to change blade pitch to direct thrust and steer the craft, the quadrotor employs a much simpler differential thrust mechanism to control roll, pitch, and yaw. These three critical angles of rotation about the center of mass of the craft make up the overall *attitude* of the craft. In order to track these attitude angles and changes to them while the craft is in motion, the use of two coordinate systems is required. The *body frame* system is attached to the vehicle itself at its center of gravity. The *earth frame* system is fixed to the earth and is taken as an inertial coordinate system in order to simplify analysis. The North-East-Down convention will be used when describing the axes of the earth frame system to comply with standard aviation systems and to satisfy the right hand rule (as opposed to, for example, North-East-Up). The angular difference between these two coordinate systems is sufficient to define the platform attitude at any point in space. Specifically, starting with both systems parallel, the attitude of the system can be replicated by first rotating the body frame around its z-axis by the yaw angle, ψ, followed by rotating around the y-axis by the pitch angle, θ, and lastly by rotating around the x-axis by the roll angle, ϕ. Figure 8 illustrates the axes of both the body and earth frame, and how the flight attitude angles affect these axes. This rotation sequence is known as Z-Y-X rotation, as the order of axis rotation is of extreme importance.

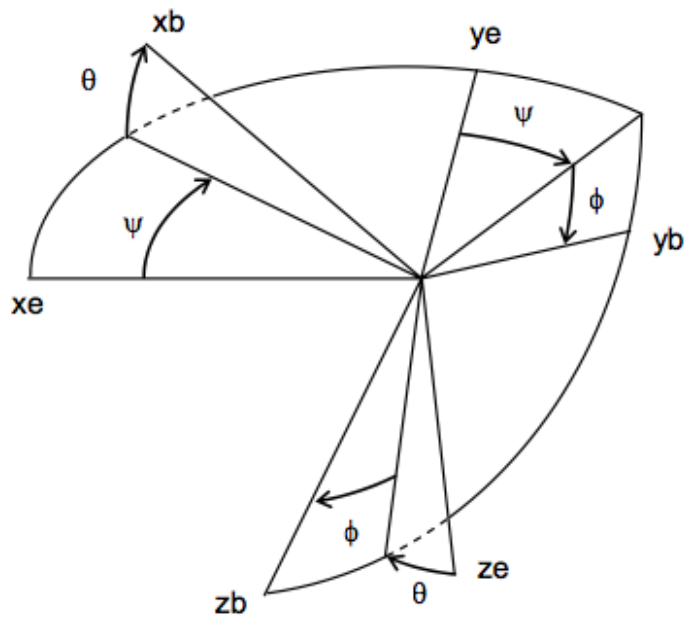


Figure 8: Body and Earth frame axes with corresponding flight angles

ANGEL System State

In defining the dynamic behavior of the ANGEL platform, we must have knowledge of the *state* of the craft. While more about the ANGEL state vector will be discussed later, knowledge of the parameters involved in defining the state describing the craft at any instant in time will help in understanding the derived dynamics.

The angles that make up the attitude of the craft with respect to the body coordinate system have already been discussed. The roll angle, Φ, the pitch angle, θ, and the yaw angle, Ψ, will all be represented in the state vector. Additionally, the angular velocities of these about each axis will be represented using dot notation, $\dot{\phi}, \dot{\theta}, \dot{\psi}$. These 6 states effectively define the attitude of the craft with respect to its own coordinate system. An additional 6 states are necessary to define the relationship of the craft with respect to the earth fixed coordinate system. These states include the physical location of the craft within the earth fixed system along each of its principal axes, denoted as X, Y, and Z. Additionally, the velocity of the craft in each of these directions is also necessary, and will be denoted as $\dot{X}, \dot{Y}, \dot{Z}$.

Together, these 12 state variables make up the state vector of the ANGEL platform. This state vector is provided in equation (1)

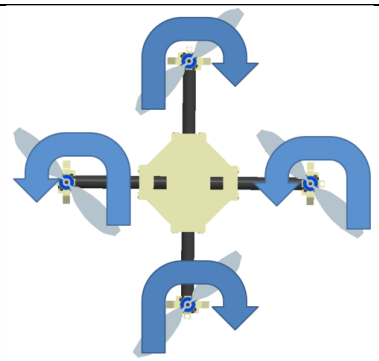
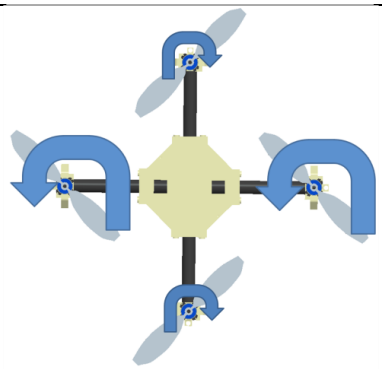
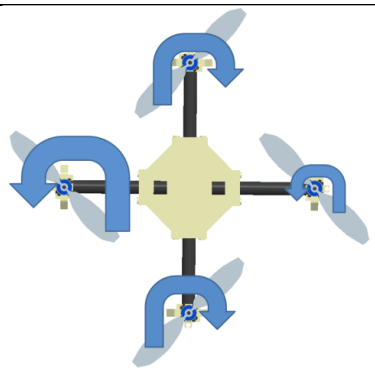
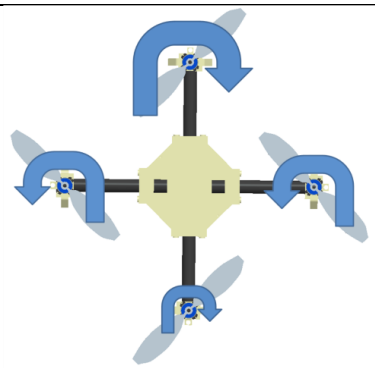
$$\mathrm{X} = \begin{bmatrix} \boldsymbol{\phi} & \boldsymbol{\theta} & \boldsymbol{\psi} & \dot{\boldsymbol{\phi}} & \dot{\boldsymbol{\theta}} & \dot{\boldsymbol{\psi}} & \boldsymbol{X} & \boldsymbol{Y} & \boldsymbol{Z} & \dot{\boldsymbol{X}} & \dot{\boldsymbol{Y}} & \dot{\boldsymbol{Z}} \end{bmatrix} \tag{1}$$

With this state now available, we can begin the overview of the platform dynamics, knowing exactly what parameters we need to define in order to have a complete model of the platform.

ANGEL Actuator Basics

With this basic review of aircraft attitude, it is now important to understand how the quadrotor is able to change the thrust output of each actuator to force a change in one or more of the attitude angles. It is important to remember that the quadrotor is by nature an underactuated system. This means that the vehicle is able to control all six DOF (three axes of translation and an angle of rotation about each translational axis) with only four input actuators. This underactuated state means that two DOF are coupled, in this case, the x- and y-axis translations. Translation on these axes depends directly on the attitude of the craft with respect to the other four degrees of freedom. The pictures in Table 3 illustrate the possible thrust configurations and the resulting angular shift. One of the simplifying principals of the quadrotor configuration over a single rotor configuration is the lack of an anti-torque rotor. By allowing two rotors to spin CW and the other two to spin CCW, as long as the ratio of thrust generated by CW to CCW actuators stays constant, the craft will not be subject to a non-zero torque resulting in a yaw deviation.

Table 3: Quadrotor differential thrust examples

	HOVER / ALTITUDE CHANGE When all actuators are at equal thrust, the craft will either hold in steady hover (assuming no disturbance) or increase/decrease altitude depending on actual thrust value.
	YAW RIGHT If the CW spinning actuators are decreased (or the CCW actuators increased), a net torque will be induced on the craft resulting in a yaw angle change. In this instance, a CW torque is induced.
	ROLL RIGHT If one of the actuators is decreased or increased on the roll axis as compared to the other actuator on the same axis, a roll motion will occur. In this instance, the craft would roll towards the right.
	PITCH UP Similar to the roll axis, if either actuator is changed on the pitch axis, the axis will rotate in the direction of the smaller thrust. In this instance, the craft nose would pitch up towards the reader (out of the page) due to the differential on the pitch axis.

Coordinate System Rotations

It was previously mentioned that two coordinate systems are needed to define the instantaneous state of the platform at any time. First, a body fixed system with the x-axis along the front of the craft, the y-axis to the right, and the z-axis down. Second, an earth fixed inertial system using the North-East-Down convention typical of aviation applications. The rotation of one frame relative to the other can be described using a rotation matrix, comprised of 3 independent matrices describing the craft rotation about each of the earth frame axes. These rotation matrices are given in equations (2) – (4).

$$R_\phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\ \phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \tag{2}$$

$$R_\theta = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3}$$

$$R_\psi = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\ \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Using these rotation matrices, the complete orientation of one coordinate system with respect to the other can be calculated [11]. The total rotation matrix equation is provided in equation (5).

$$\Theta = R_\phi R_\theta R_\psi \tag{5}$$

ANGEL Body Forces and Moments

In order to create an accurate model of the platform, the various forces and moments induced on the craft must be understood and accounted for. As these forces and moments are discussed, some assumptions are

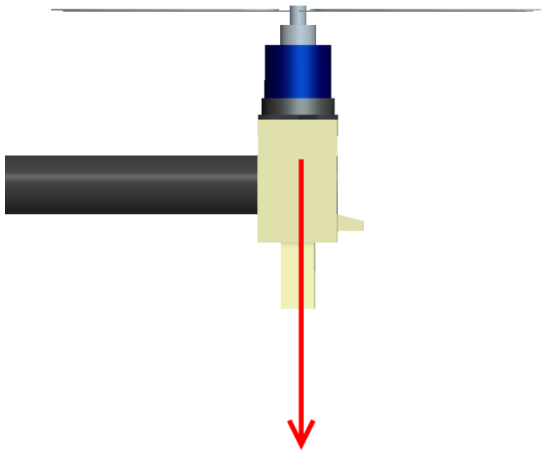
made in order to simplify analysis. These assumptions will be discussed in the appropriate areas.

The forces and moments induced on the craft are responsible for its movement and overall attitude. Each of the forces can be broken into an x, y, and z component. The following Newton-Euler form equation (6) defines the total influence of the net forces and moments on the craft. Using this equation with the individual forces and moments defined for each degree of freedom below, we can determine the full equations of motion for the craft.

$$\begin{bmatrix} mI_{3x3} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} \dot{V} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} \omega \times mV \\ \omega \times I\omega \end{bmatrix} = \begin{bmatrix} F \\ \tau \end{bmatrix} \tag{6}$$

The variables of concern in designing the control system are the $\dot{V}$ (change of body linear velocity) and the $\dot{\omega}$ (change of body angular velocity). Carrying the differential through to the sub variables that specify the various axes and degrees of freedom available to both velocities, we arrive at our state variables that will be used to specify the orientation of the craft to the control system.

Figure 9 and Figure 10 show how forces are interpreted differently based on which reference coordinate system is used. In Figure 9, where both the earth fixed system and the craft system are aligned in the Z-axis direction, the thrust generated by the actuator is the same for each coordinate system representation. As the craft undergoes a roll movement (for example, a 40 degree roll to the left shown in Figure 10), the alignment of the coordinate systems disappears. The full thrust force is still available to the craft fixed coordinate system, but only a portion of it is available in the z-axis of the earth fixed system. This illustrates the need of the rotation matrices previously developed, and will be useful in describing the craft in both systems for attitude estimation and translational movement.



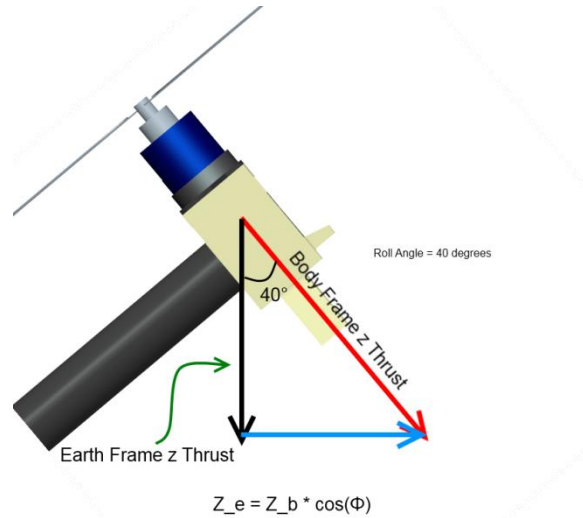


Figure 9: Zero roll/pitch thrust force **Figure 10: 40° roll angle thrust force**

From the reference of the onboard craft coordinate system, the thrusts generated by the motors/propellers are always in the crafts z-direction. The gravity vector, however, is always in the fixed frame z direction (towards the center of the earth). In this instance, it is important to utilize the rotation matrix from equation (5). We can therefore write the force of gravity as

$$F_g = mg \begin{bmatrix} -\sin\theta \\ \cos\theta \sin\phi \\ \cos\theta \cos\phi \end{bmatrix}_{body} \tag{7}$$

It is important to remember that this force is taken with respect to the *craft* coordinate system, affixed to the center of gravity of the ANGEL platform. Along with gravity, the only other forces to be considered are the forces generated by the propeller/motor combos. These forces combined with the force of gravity, allow us to solve equation (6) for the forces acting on the platform, and determine the acceleration of the craft in terms of the craft-fixed frame.

$$\begin{bmatrix}\ddot{X}\\ \ddot{Y}\\ \ddot{Z}\end{bmatrix} = -\frac{1}{m}\begin{bmatrix}0\\ 0\\ F_{thrust}\end{bmatrix} + g\begin{bmatrix}-\sin\theta\\ \cos\theta\sin\phi\\ \cos\theta\cos\phi\end{bmatrix} - \begin{bmatrix}\dot{\theta}\dot{Z} - \dot{\psi}\dot{Y}\\ \dot{\psi}\dot{X} - \dot{\phi}\dot{Z}\\ \dot{\phi}\dot{Y} - \dot{\theta}\dot{X}\end{bmatrix} \tag{8}$$

The last matrix containing rotational and translational velocities is the result of the cross product of the ω and V time derivatives in equation (6). Of special note is the fact that the only thrust component existing in the body frame is in the z-direction. To simplify the simulation model, the hub forces (horizontal forces on the blades) and friction/drag induced by the air on the blades will be ignored in the x- and y- directions.

At this point, another assumption should be noted. On take off and landing, there are significant aerodynamic changes due to a phenomenon known as the ground effect. While operating near the ground, a reduction in the induced airflow velocity provides greater efficiency from the rotor, and thus more thrust. Since autonomous take off and landing is not within the scope of this paper, the ground effect will be ignored when developing the simulation model of the ANGEL platform.

Next the moments will be considered in order to determine the acceleration rates of the various attitude angles. Each of the three angle accelerations is subject to the *Frame (or body) gyroscopic effect*. This is the moment induced by the angular velocity of the frame as a whole. The following equations illustrate the Frame Gyro Effect on each attitude angle.

Roll Angle Gyro Effect $$\dot{\theta}\dot{\psi}\left(I_y - I_z\right) \tag{9}$$

Pitch Angle Gyro Effect $$\dot{\phi}\dot{\psi}(I_z - I_x) \tag{10}$$

Yaw Angle Gyro Effect $$\dot{\phi}\dot{\theta}\left(I_x - I_y\right) \tag{11}$$

The equations show that the velocities at which the other angles are changing directly influence the acceleration of the target angle. A derivation and description of the moments of inertia of the three axes is given following the descriptions of all contributing moments. The next moment to discuss is

the moment generated by the rotor thrusts. This moment, known as the *Thrust-Induced Moment*, only affects the roll and pitch angles. The equations that follow illustrate this moment.

Roll Angle Thrust Induced Moment	$r(-T_2 + T_4)$	(12)
Pitch Angle Thrust Induced Moment	$r(T_1 - T_3)$	(13)

Although the yaw angle is not affected by the thrust-induced moment, it is still impacted by the various rotor thrusts due to imbalance in the counter rotating torques. While the thrusts are balanced, the yaw angle change should nearly be zero, neglecting any external noise or disturbance. Thrust imbalance controls yaw, which negates the need of a second anti-torque rotor. The equation for *Counter-rotating Thrust Imbalance* follows.

Counter-rotating thrust imbalance	$(T_1 + T_3 - T_2 - T_4)$	(14)

The last set of moments to consider are the individual moments from the propeller induced gyroscopic effects. These effects are based on the rotor inertia, rotor velocity, and changing attitude angle. The *Rotor gyroscopic effects* are summarized below.

Roll Rotor Gyro Effect	$J_r \dot{\theta} \Omega_r$	(15)
Pitch Rotor Gyro Effect	$J_r \dot{\phi} \Omega_r$	(16)

The inertial counter-torque moment on the z-axis is analogous to the rotor gyroscopic effects for the x- and y-axis and is given in [12].

Inertial counter-torque effect	$J_r \dot{\Omega}_r$	(17)

Equations (15), (16) and (17) comprise the total moment effects of the propeller itself. In comparison to the other moments, these gyroscopic

effects have very insignificant roles in the overall attitude of the craft. They are presented to provide a more accurate model, but will not be used in the simulation or implementation of the control system in order to reduce the overall complexity of the system [9].

Together, these moments determine the overall behavior of the principle attitude angles of the craft. The equations illustrating the acceleration of rotation around each axis are given in (18-20).

$$\ddot{\phi} = \sum \frac{\tau_x}{I_{xx}} = \left[\dot{\theta}\dot{\psi}\left(I_{yy} - I_{zz}\right) + r(-T_2 + T_4)\right]\left(\frac{1}{I_{xx}}\right) \tag{18}$$

$$\ddot{\theta} = \sum \frac{\tau_y}{I_{yy}} = \left[\dot{\phi}\dot{\psi}(I_{zz} - I_{xx}) + r(T_1 - T_3)\right]\left(\frac{1}{I_{yy}}\right) \tag{19}$$

$$\ddot{\psi} = \sum \frac{\tau_z}{I_{zz}} = \left[\dot{\phi}\dot{\theta}\left(I_{xx} - I_{yy}\right) + (T_1 + T_3 - T_2 - T_4)\right]\left(\frac{1}{I_{zz}}\right) \tag{20}$$

ANGEL Moments of Inertia

Calculating the moments of inertia about the various axes is the next step towards accurate modeling of the ANGEL system. While the notation (for example, I_{xx}) denotes the moment of inertia around the x-axis while the platform is rotation around the x-axis (or rolling), we will assume the rolling, pitching and yawing of the platform will not change the moment for any specific axis. The derivation of these principal moments of inertia (I_{xx}, I_{yy}, and I_{zz}) is adapted from [8]. For the remainder of the inertia discussion, refer to Figure 11 for the various axes and motors.

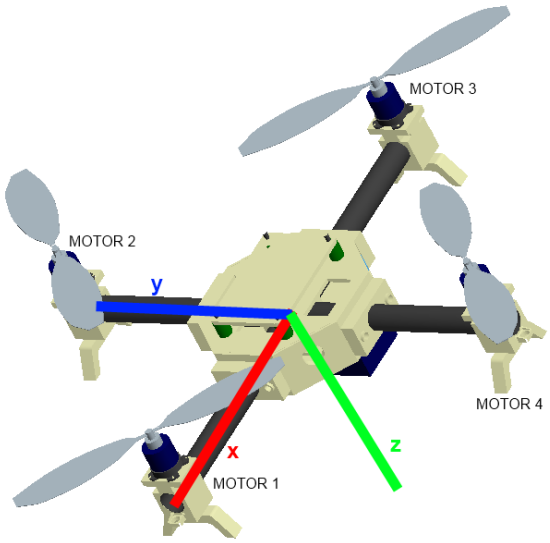


Figure 11: Motor numbers and axes for inertia calculations

Assuming perfect symmetry between the x- and y-axis, it is safe to assume the moments about each of these axes are numerically equivalent. To simplify the modeling process, all mass components of the platform will be modeled as solid cylinders attached by zero mass and frictionless arms. The moment of inertia of a cylinder rotating about an axis perpendicular to its body is given by

$$I = \frac{mr^2}{4} + \frac{mh^2}{12} \tag{21}$$

In (21), *m* refers to the cylinder mass, *r* to the cylinder radius, and *h* to the cylinder height. For this implementation, the cylinder includes the motor, motor bracket and landing gear. Taking the x-axis as the first effort, the moment of inertia due to the motors on either side of the axis (motors 2 and 4) is approximated by

$$I_{xx_1} = 2ml^2 \tag{22}$$

Again, *m* refers to the mass of a single cylinder and *l* refers to the arm length of one side of the craft. The last items of concern to the moment of inertia are the two motors in line with the x-axis (motors 1 and 3) and the central hub where the arms meet. The equation governing the effect of these objects on the moment of inertia is derived from (18).

$$I_{xx_2} = 2\left[\frac{mr^2}{4} + \frac{mh^2}{12}\right] + \frac{m_h r_h^2}{4} + \frac{m_h r_h^2}{12} \tag{23}$$

The first bracketed portion of the equation accounts for motors 1 and 3. The latter portion refers to the central hub, which includes all the electronic speed controllers for the motors, the avionics, sensors, and the batteries and power distribution system. Together, equations (22) and (23) form the overall moment of inertia approximation for the x- and y-axis.

$$I_{xx} = \frac{mr^2}{2} + \frac{mh^2}{6} + 2ml^2 + \frac{m_h r_h^2}{4} + \frac{m_h r_h^2}{12} \tag{24}$$

Due to symmetry, equation (24) applies to both the x- and y-axis. The moment of inertia for the z-axis rotation (yaw) can be attributed to all 4 motor/mount/gear cylinders and the central hub. The moment of the central hub modeled as a cylinder rotating about an axis through and parallel to its center is given by

$$I_{zz_1} = \frac{m_h r_h^2}{2} \tag{25}$$

For the 4 motors at an arm's length away from the axis of rotation, the total moment of inertia is

$$I_{zz_2} = 4ml^2 \tag{26}$$

Therefore, by combining equations (25) and (26), the total equation for the approximation of the moment of inertia about the z-axis is

$$I_{zz} = \frac{m_h r_h^2}{2} + 4ml^2 \tag{27}$$

ANGEL Kinematics and the Gimbal Lock Phenomenon

The kinematics of the ANGEL platform consider the movement of the body as a whole within its environment with no consideration paid to the forces or moments that actually induce these movements. Classically, this involves determining the velocity of the body from its position information through a time derivative. If we wish to determine the linear velocity of the craft, we can use the rotation matrix along with the time derivative of the position. This is shown in equation (28).

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix}_{craft} = \Theta \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix}_{earth} \tag{28}$$

From [11], it is shown that inverse of the total rotation matrix is equal to its transpose (orthogonal matrix), which means (28) can be rewritten as

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix}_{earth} = \begin{bmatrix} (\cos\theta\cos\psi)\dot{X}_{craft} + (\sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi)\dot{Y}_{craft} + (\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi)\dot{Z}_{craft} \\ (\cos\theta\sin\psi)\dot{X}_{craft} + (\cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi)\dot{Y}_{craft} + (\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi)\dot{Z}_{craft} \\ (-\sin\theta)\dot{X}_{craft} + (\sin\phi\cos\theta)\dot{Y}_{craft} + (\cos\theta\cos\phi)\dot{Z}_{craft} \end{bmatrix} \tag{29}$$

Also from [11], we know that the attitude (Euler) angles are not constant with time. Therefore, a relationship between the Euler angle rates (with respect to the earth fixed system) and the body axis rates (with respect to the craft fixed system) must be determined. At first glance, the Euler rates and body axis rates appear the same. However, under constant rotational velocity, the body axis rates are constant, but the Euler rates are not, due to

the direct dependence on the angular displacement of the coordinate systems. Equation (29) shows the derived Euler angle rates as a function of the body axis rates [11].

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{earth} = \begin{bmatrix} 1 & \tan\theta\sin\phi & \tan\theta\cos\phi \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{craft} \tag{30}$$

The use of these Euler rates is, however, not without disadvantages. While it is easy to immediately see the physical application of these Euler angles through visible rotations of the craft, their use opens the simulation model (and the physical system) to a phenomenon known as *gimbal lock*. Gimbal lock occurs when a craft capable of 3D rotation rotates such that two formerly exclusive axes of rotation coincide in the same plane. This phenomenon is illustrated in Figure 12.

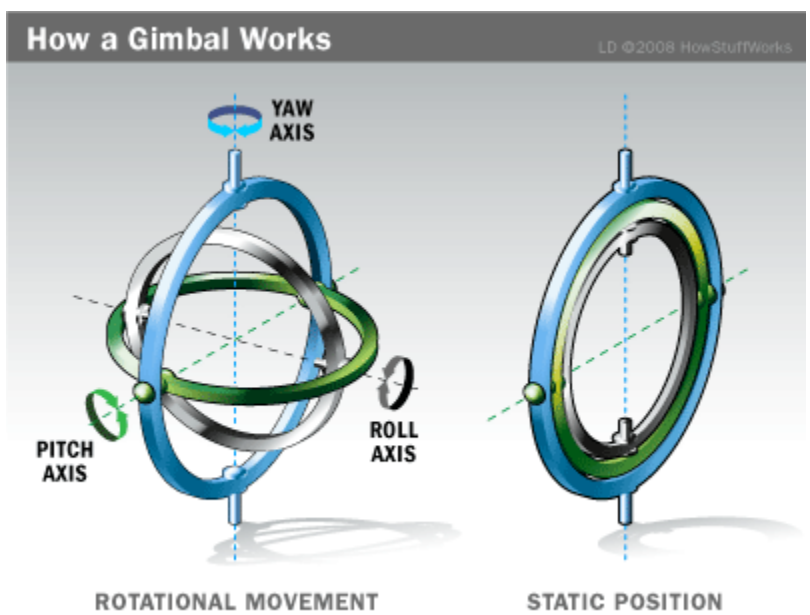


Figure 12: Gimbal Lock Phenomenon. Photo from HowStuffWorks.com

Using Euler angles, if the pitch angle rotates to pi/2, the independent axis to force a yaw rotation is lost. It is therefore general practice when dealing with

crafts that may experience these angles to use a different method of defining them. This is known as the quaternion method.

The Quaternion Method

While quaternions are not as visual as Euler angles (it is harder to imagine the implied craft movement when looking at the quaternion tuple), they offer a greatly simplified approach to 3D rotation. Where Euler angle rotation requires 3 successive angles of rotation (Z-Y-X, or Yaw-Pitch-Roll) to completely describe the craft orientation, the quaternion describes the rotation in a single move (rotate by θ degrees around the axis directed by the defined vector). When implemented, the quaternion used to define a rotation is a set of 4 numbers (s,x,y,z), such that

$$s^2 + x^2 + y^2 + z^2 = 1 \tag{31}$$

In application, the quaternion is constructed around a unit vector defining the axis of rotation (x_0, y_0, z_0) and the angle of rotation θ.

$$q = \begin{bmatrix} \cos\frac{\theta}{2} \\ x_0 \sin\frac{\theta}{2} \\ y_0 \sin\frac{\theta}{2} \\ z_0 \sin\frac{\theta}{2} \end{bmatrix} \tag{32}$$

From [11], the expression of these quaternion components [q_0, q_1, q_2, q_3] in terms of the Euler angles is given as

$$q = \begin{bmatrix} \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \\ \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \\ \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} \\ \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \end{bmatrix} \tag{33}$$

Alternatively, we can use an equivalent quaternion rotation matrix to derive the Euler angles back from the quaternion implementation.

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(\frac{2(q_2q_3 + q_0q_1)}{q_0^2 - q_1^2 - q_2^2 - q_3^2}\right) \\ \sin^{-1}(-2(q_1q_3 - q_0q_2)) \\ \tan^{-1}\left(\frac{2(q_2q_3 + q_0q_1)}{q_0^2 + q_1^2 - q_2^2 - q_3^2}\right) \end{bmatrix} \tag{34}$$

This relationship will allow us to redefine any equations of motion describing the behavior of the craft in terms of quaternions instead of Euler Angles.

MATLAB Simulation of ANGEL

In order to accurately simulate the behavior of the ANGEL system, we must build a loop through which an input command to the ANGEL can be applied and the resulting state space vector is updated. Figure 13 shows a block diagram of how this system should be implemented.

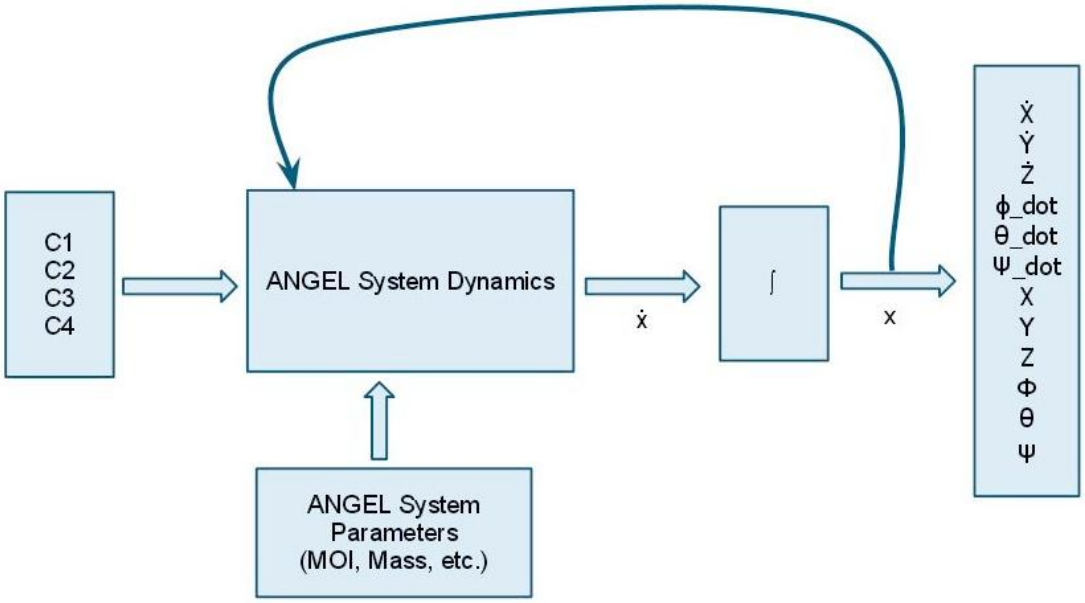


Figure 13: ANGEL Simulation Block Diagram

The input block in the simulation diagram allows us to change the commands (C1, C2, C3, C4) going to the motors. Thus, disregarding any external disturbances or noise from the physical implementation of the system, we can track how the system will react to changes in the actuator output. This will give us some idea of how the craft will react, and will allow us to design the control system based on desired performance parameters.

One of the most straightforward movements the craft can make is a simple pitch or roll in order to move either forward/back or left/right. To the novice user unfamiliar with the actuator interactions and coupling, the first attempt may involve changing the output speed of only one motor. For example, if a slight forward propagating pitch angle is desired, the first attempt may be to turn on all actuators to gain altitude, provide a negative pulse to the front motor momentarily in order to cause the craft to pitch forwards, travel forwards for a few seconds before providing a positive pulse to the front motor to kick the craft out of forward pitch. The net movement of the craft would presumably be in the positive x-direction of the earth fixed frame with no movement in the fixed y-direction. This however, is not the result that occurs. Figure 14 illustrates the input signal described in the preceding paragraph.

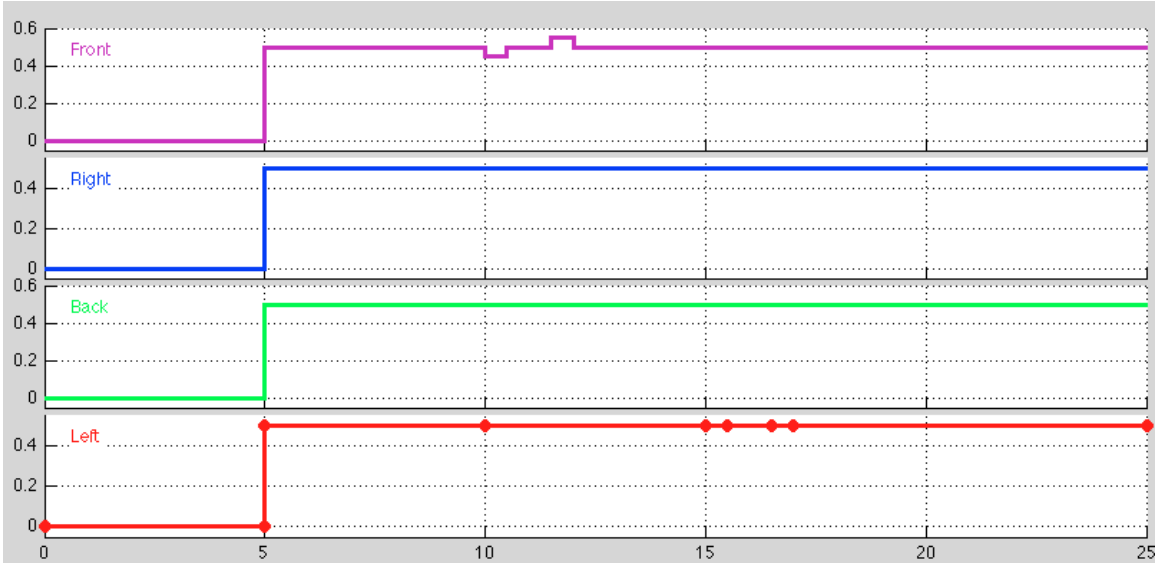


Figure 14: Actuator Signal Input for Pitch Forward Attempt

The actuators are powered on at 5s, steadily gaining in altitude. At 10s, a negative pulse is provided to the front motor, causing a drop in speed, which should cause the craft to pitch forward and move in the positive x-direction. At 12s, a positive pulse is applied to presumably bring the craft out of forward pitch and back into steady hover. This however, is not what occurs. Figure 15 illustrates the overall flight path of the craft in the earth XY plane.

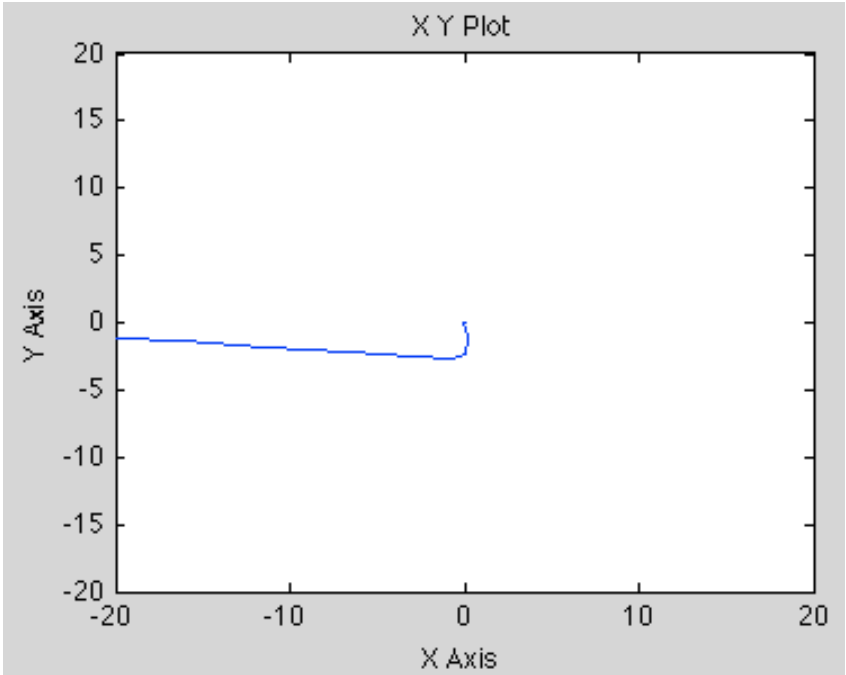


Figure 15: Flight path under single actuator change

As is evident, the craft does not exhibit the desired behavior. We can analyze what occurs by studying the roll, pitch, and yaw moments as a function of time. Figure 16 shows these values for this particular simulation.

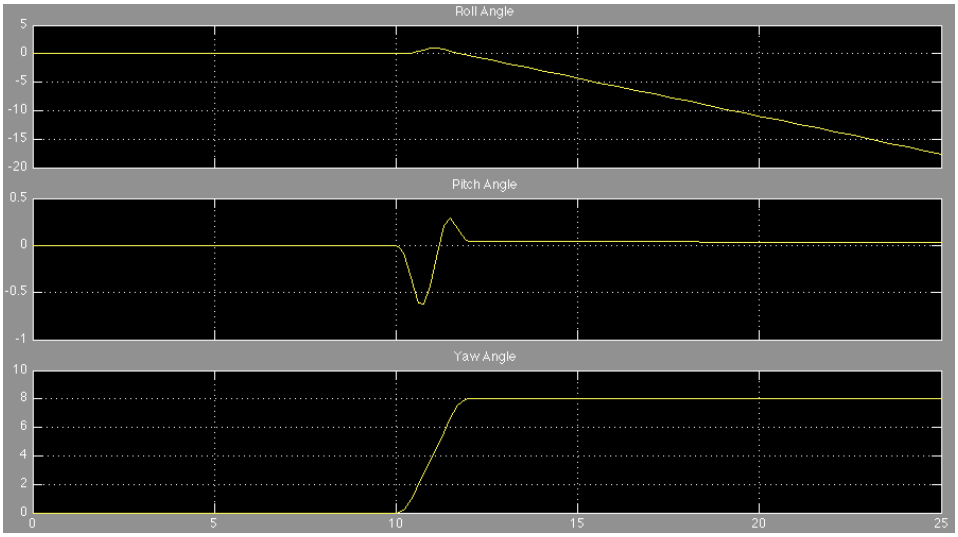


Figure 16: Roll, Pitch and Yaw angles for single actuator simulation

From the figure, we see the desired pitch angle response previously predicted. The actuators all turn on equally at 5s, and there is no deviation in the roll, pitch or yaw angles. At 10s, the effect of the short negative pulse on the front motor is evident in the pitch response curve, followed closely by the short positive pulse to bring the pitch back to a nearly zero offset. Thus, the pitch acts in accordance to the expectations. The yaw angle, however, does not look correct. There was no intended yaw movement in our signal description, and the presence of this deviation is entirely responsible for the odd trajectory of the craft. Due to the change in ratio of counter-clockwise propeller speed to clockwise propeller speed, an overall yaw moment was induced, as described by equations (14) and (20). As the front motor speed was decreased, the back motor speed should have increased simultaneously to compensate for the decreased overall clockwise thrust. Instead, the counter-clockwise spinning motors dominated the ratio, inducing a clockwise moment causing the craft to spin towards the negative y-direction in Figure 15. The pitching and yawing movements, when combined, changed the

thrust vector of the craft, similar to Figure 10. This caused the overall roll deviation, which accounts for the craft rolling off in the negative x-direction.

The correct method for implementing a forward movement will rectify the CCW/CW thrust ratio problem that caused the erratic behavior in the first simulation attempt. The new input signals for the actuators are shown in Figure 17.

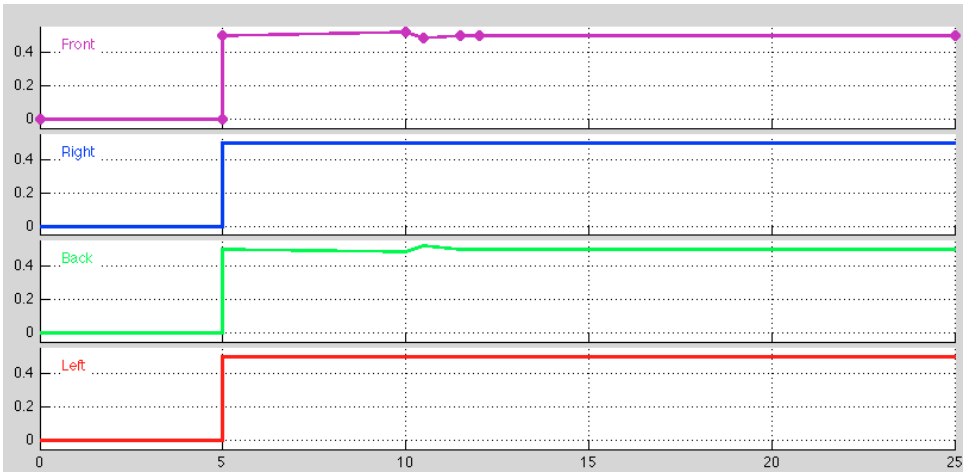


Figure 17: New Yaw Compensating Input Signal

While still not perfect, these signals provide something much closer to the intended behavior. The XY flight path plot and the Roll/Pitch/Yaw graphs are shown in Figure 18 and Figure 19, respectively.

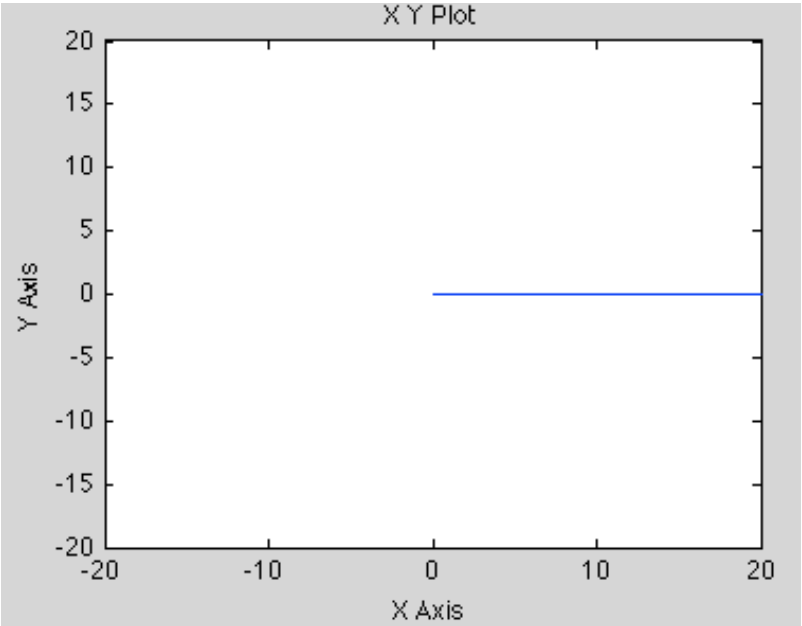


Figure 18: Correct flight path with updated signals

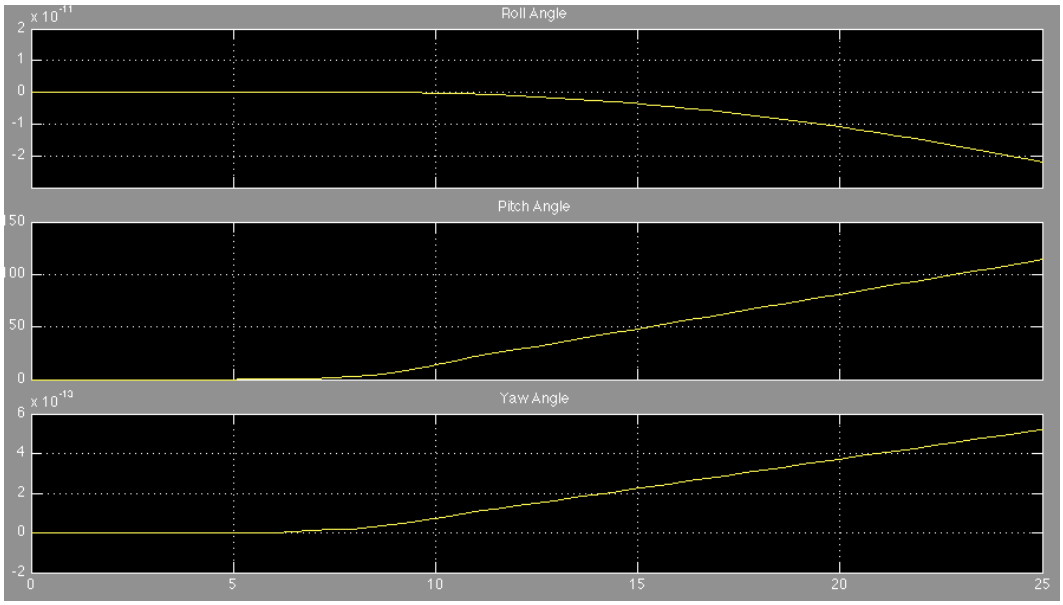


Figure 19: Roll/Pitch/Yaw graphs with updated input signals

Looking at the angle graphs, the first reaction may be that we did not solve anything by changing the input signals. The roll and yaw deviations, however, are much smaller in magnitude when compared to the pitch deviation. The presence of the roll/yaw changes is actually correct and not an error in the simulation. Referring to equations (18) and (20), the roll and yaw accelerations depend directly on the velocity of the changing pitch angle through the roll/yaw gyroscopic effects given in (9) and (11). Therefore, these minute deviations are part of the intended response, but do not play a significant role in the overall trajectory of the craft. For reference, the code used in the dynamics simulation of the platform is provided in Appendix A-1 and a diagram illustrating the Simulink Model used for the simulation is provided in Figure 20. The results of this simulation also verify the need for a control system to provide input to the motors. Notice the magnitude of the pitch angles. While we can manually actuate the motors, these results show that a very small change in motor input results in a change from 0 radians pitch to over 100 radians (almost 16 revolutions) in a matter of 25 seconds. In order to track a desired reference angle, a controller will need to be implemented and optimized.

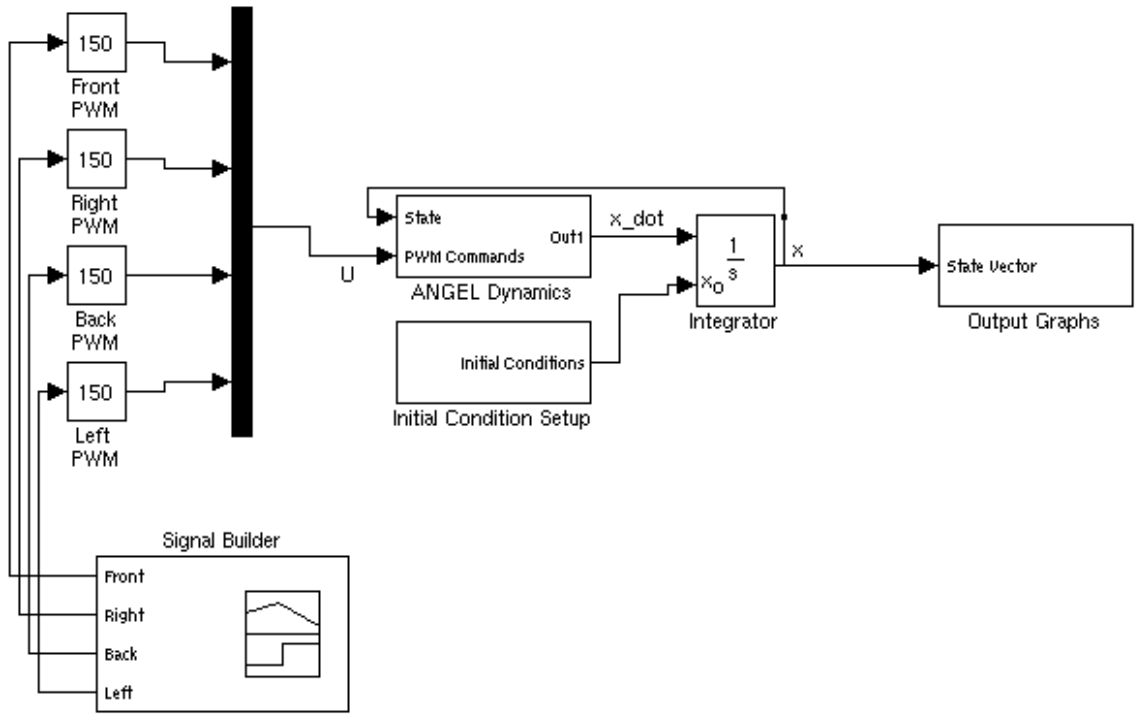


Figure 20: Simulink model of ANGEL simulation

From these two simulations, the complexity of by-hand control of the ANGEL platform should be clear. Each axis will need an independent control system implementation tuned to the specific characteristics and variables of the axis. The development and implementation procedure of the control systems are covered in Section IV.

Section IV: ANGEL Control Development

Control Fundamentals

A control system is an external architecture placed on any (controllable) dynamical system in order to maintain equilibrium determined by an input set point. By comparing the values that define the overall state or orientation of a system to the desired values, gaps and errors can be accounted for and rectified in order to achieve homeostasis. Control systems are present everywhere in nature. Biologically, the temperature of a human body is achieved through a complex process of thermoregulation. The

healthy bodily temperature is the set point, and the actual temperature of the body is constantly driven to that set point through organ heat generation and dissipation through evaporation (sweat) and vasodilatation.

Of importance to the study of unmanned vehicles and mechanical systems is the development of applicable control systems. Historically, one of the most popular examples of feedback control is the Centrifugal Flyball Governor engineered by J. Watt in 1788. In order to control the speed of steam engines, which exhibited rotary output, the speed of the rotation needed constant monitoring and control. The solution to this problem was to affix a device comprised of two rotating flyballs spun outwards by the centrifugal force generated by the rotary engine (Figure 21). As the engine speed increased, the rotational speed increased and the flyballs were forced up and out. This actuated a steam valve that slowed the engine, and the first version of automatic speed control was implemented.

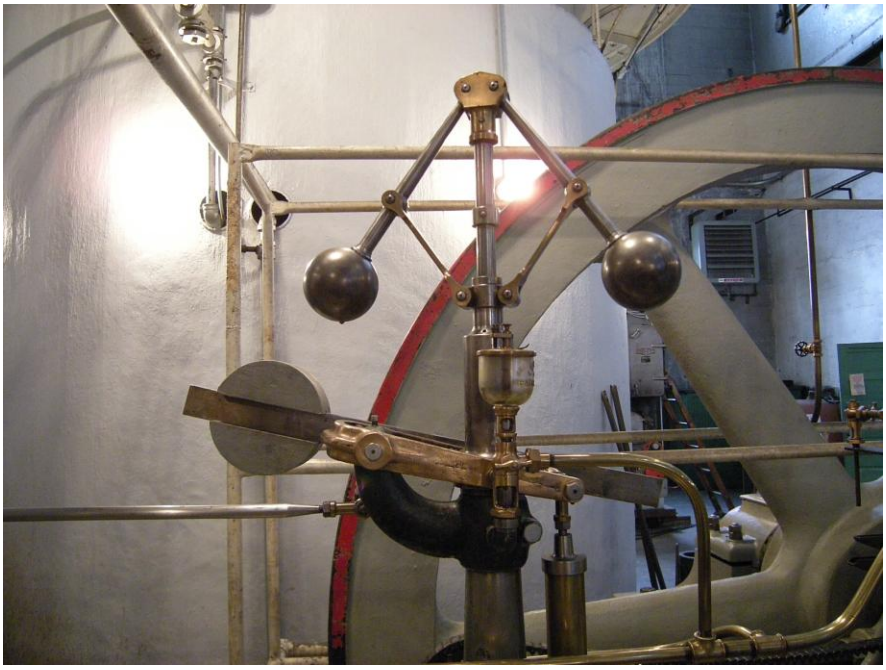


Figure 21: Centrifugal Governor[1]

This type of controller is known as "bang-bang" control, or On/Off control. If the speed needs to be increased or decreased, the valve is closed or opened.

[1] Photo by Joe Mabel

The discrete states of the system make implementation and testing very straightforward.

For more complex systems where a more continuous approach is necessary, linear feedback control may be used. One of the simplest implementations of linear feedback control is “Proportional Control”. In this sense, the control system actuates the system in proportion to the current error between the actual operation point (Process Variable) and the desired set point. However, the simplicity of proportional control is not without its drawbacks. If the proportional gain is set too low, the system becomes sluggish in its response, but is generally safer and more stable. Alternatively, if the gain is too high, the system will quickly respond to errors, but will experience oscillations around the set point.

Introducing two more gains, the derivative gain and integral gain, allows us to more completely define the desired behavior of the control system. The derivative portion controls the rate of change of the process variable, and as such can much more intuitively approach the set point if designed correctly. The integral portion looks at the global steady state error, and becomes more influential the longer the error is not zero. Together, these gains make up the extremely popular PID (Proportional, Integral, Derivative) controller. Figure 22 illustrates the controller architecture on a generic plant.

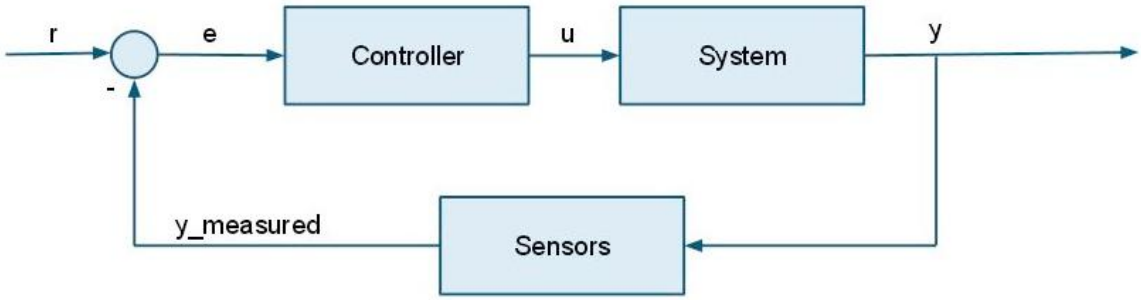


Figure 22: Generic Control System

The reference (denoted as r) is actually the desired set point for the process variable (system output) y. This reference signal is compared via negative feedback to the measured output from the sensor subsystem. The result of

this comparison is the error signal (e), which is the input to the controller. The controller is concerned with forcing this error signal to zero through the use of the previously discussed proportional, integral and derivative controls (or through other mechanisms if a different controller architecture is used). The controller then outputs an appropriate signal (u) as an input to the system with the idea that u will drive y more towards r, thus decreasing the magnitude of e.

With the advent and wide use of electronic controllers, the complexity of control system available to the average user has increased dramatically in recent years. While control systems can be implemented with a series of operational amplifiers and passive circuitry, the real power of adaptive, dynamic control comes in the form of microcontrollers. These control systems and corresponding gains can be changed based on sensor inputs and changing plant parameters. With a dynamic controller installed, some systems are even capable of tuning themselves, setting the appropriate gains in order to achieve the desired response characteristics for their current implementation.

In the following sub-sections, the model developed in Section III will be analyzed, and a control system will be selected and implemented based on a set of desired criteria. This system will be modeled in MATLAB in order to test the system response before it is implemented on the avionics platform.

Model Simplifications

In model-based control, it is typical to take the full simulation model and reduce it such that the control is applied to a specific behavior envelope. This makes the initial design of the system less intensive, and allows calibrating the system for the most significant effects before introducing minor effects that may add greatly to the complexity of the controller or sensor subsystems. Equations 35 – 37 below show the rotational equations of motion from the full simulation model developed in Section III.

$$\ddot{\phi} = \sum \frac{\tau_x}{I_{xx}} = \left[\dot{\theta}\dot{\psi}\left(I_{yy} - I_{zz}\right) + r(-T_2 + T_4) + J_r\dot{\theta}\Omega_r\right]\left(\frac{1}{I_{xx}}\right) \tag{35}$$

$$\ddot{\theta} = \sum \frac{\tau_y}{I_{yy}} = \left[\dot{\phi}\dot{\psi}(I_{zz} - I_{xx}) + r(T_1 - T_3) + J_r\dot{\theta}\Omega_r\right]\left(\frac{1}{I_{yy}}\right) \tag{36}$$

$$\ddot{\psi} = \sum \frac{\tau_z}{I_{zz}} = \left[\dot{\phi}\dot{\theta}\left(I_{xx} - I_{yy}\right) + (T_1 + T_3 - T_2 - T_4) + J_r\dot{\Omega}_r\right]\left(\frac{1}{I_{zz}}\right) \tag{37}$$

As previously noted in Section III, the rotor gyroscopic effects induced by the rotational motion of the propellers are not significant when compared to the moments induced by the actuator thrusts. For this reason, they are disregarded in both the simulation and the controller development process. To limit the envelope over which the controller is valid (thus reducing the complexity of the controller while accomplishing the most vital characteristics of the craft) a hover state will be the desired orientation of the craft. This means we are only concerned with the rotations of the craft near hover. As a result, we will only consider the rotational subsystem for the roll, pitch, and yaw angles. X,Y, and Z values, while important for obstacle avoidance and path following, all fall out of the angle subsystem due to coupling (with the general exception of the Z value, as this is determined by a separate input comprised of all the thrust values of the motors). Since the area around a steady hover has been selected as the appropriate envelope, the angular velocities for roll, pitch, and yaw will be very small. For this reason, we can also disregard the angle gyroscopic effects introduced in Section III. These reductions form the simplified control model used to develop the controller. These simplified equations of motion are provided in equations 38 – 40.

$$\ddot{\phi} = \sum \frac{\tau_x}{I_{xx}} = [r(-T_2 + T_4)]\left(\frac{1}{I_{xx}}\right) \tag{38}$$

$$\ddot{\theta} = \sum \frac{\tau_y}{I_{yy}} = [r(T_1 - T_3)]\left(\frac{1}{I_{yy}}\right) \tag{39}$$

$$\ddot{\psi} = \sum \frac{\tau_z}{I_{zz}} = [(T_1 + T_3 - T_2 - T_4)]\left(\frac{1}{I_{zz}}\right) \tag{40}$$

Input Declarations

With this simplified model defined, the next step is to define the U vector that will be used to control the system dynamics from the controller. In the simulation, the input to the system dynamics model was based on the relationship between the pulse-width modulation command send from the controller to the actuators and the actual thrust output. This relationship, however, is only linear for a small portion of the thrust curve. It is therefore beneficial to switch from PWM input to a more tangible rotational speed. From [4], it is shown that thrust generated by a motor-propeller combo is related to the square of the propeller speed when the flight regime is in hover and not translational movement, with a thrust constant factor and drag moment factor considered. Therefore, the inputs selected for the system can be formulated and are given in equations 41 – 43.

$$U_{roll} = rb(-\Omega_2^2 + \Omega_4^2) \tag{41}$$

$$U_{pitch} = rb(\Omega_1^2 - \Omega_3^2) \tag{42}$$

$$U_{yaw} = d(-\Omega_1^2 + \Omega_2^2 - \Omega_3^2 + \Omega_4^2) \tag{43}$$

There exists a fourth input, the altitude input, which is comprised of all the actuator inputs as a sum to fix the altitude of the craft. For this portion of the controller derivation, altitude is not a concern, so this input will be set to a constant value of craft mass multiplied by gravity in order to produce constant thrust.

MATLAB Control Implementation

In order to test and tune the controller before implementing it on the ANGEL platform itself, the decision was made to model it in MATLAB and tune

it using the simulation model developed in Section III (and reduced in the previous subsection).

The output of the controller serves as the input of the system dynamics model in the controller simulation. In the actual implementation of the controller, the outputs will be modified PWM commands sent to the motors in order to change the output speed (and consequently the thrust) of each of the 4 rotors. This will allow the ANGEL platform to track the desired attitude angles set by the user. These output commands (whether in the simulation or implementation) are comprised of the three gain correction terms, the sum of which creates the manipulated process variable (roll, pitch, or yaw). In PID controller theory, these correction factors are comprised of the gains (P, I, D for a PID controller) and the correct time form of the error signal.

For the proportional portion of the controller, the signal will be comprised of the P gain and the error function e(t), which in this case is simply the negative feedback function formed in the generic controller example in Figure 22. For the purposes of discussion, the examples will only be shown using the roll angle, although each attitude angle (roll, pitch and yaw) will have a separate controller tuned to their own individual dynamics. For example, the error signal for use in the proportional section of the roll controller would be the difference between the set point roll value (in hover, this would be 0 radians) and the actual roll value (from the state vector provided by the system dynamics model), such that

$$P_{out} = p_{roll} * e(t) = p_{roll} * (\phi_r - \phi) \tag{44}$$

The integral portion of the controller is comprised of the I gain and an integration of the error over time. This portion is responsible for looking at the instantaneous error as a sum over the entire implementation of the controller. This allows the controller to eliminate steady state errors and drive the output signal towards the desired reference signal. The introduction of the integration term also decreases the rise-time of the output signal but

increases the settling time. The error function in this instance includes an integral, which in practice is just the sum of the error between the reference signal and the actual state over a period of time. This is accomplished in the controller simulation by keeping a running sum of the instantaneous error in the controller subsystem. The equation governing the I portion of the modified process variable is given in equation (45).

$$I_{out} = i_{roll} * \int_0^t e(t)d\tau \tag{45}$$

Lastly, the D term of the modified signal deals with the speed with which the error signal is changing. This is used to reduce overshoot of the reference signal. It is comprised of the derivative gain and a derivative function of the error signal, and is shown in equation (46).

$$D_{out} = d_{roll} * {}^{d}/_{dt}\, e(t) \tag{46}$$

Each of these terms can be collected and summed to create the overall input to the system dynamics block (equation (47)). Recall that this example only deals with the roll controller, and that the total U vector will be comprised of the signals from each of the three controllers.

$$u_{roll} = p_{roll} * e(t) + i_{roll} * \int_0^t e(t)d\tau + d_{roll} * {}^{d}/_{dt}\, e(t) \tag{47}$$

The implementation of the controller was achieved using a custom-written block in MATLAB. Figure 23 shows this block along with its inputs and outputs.

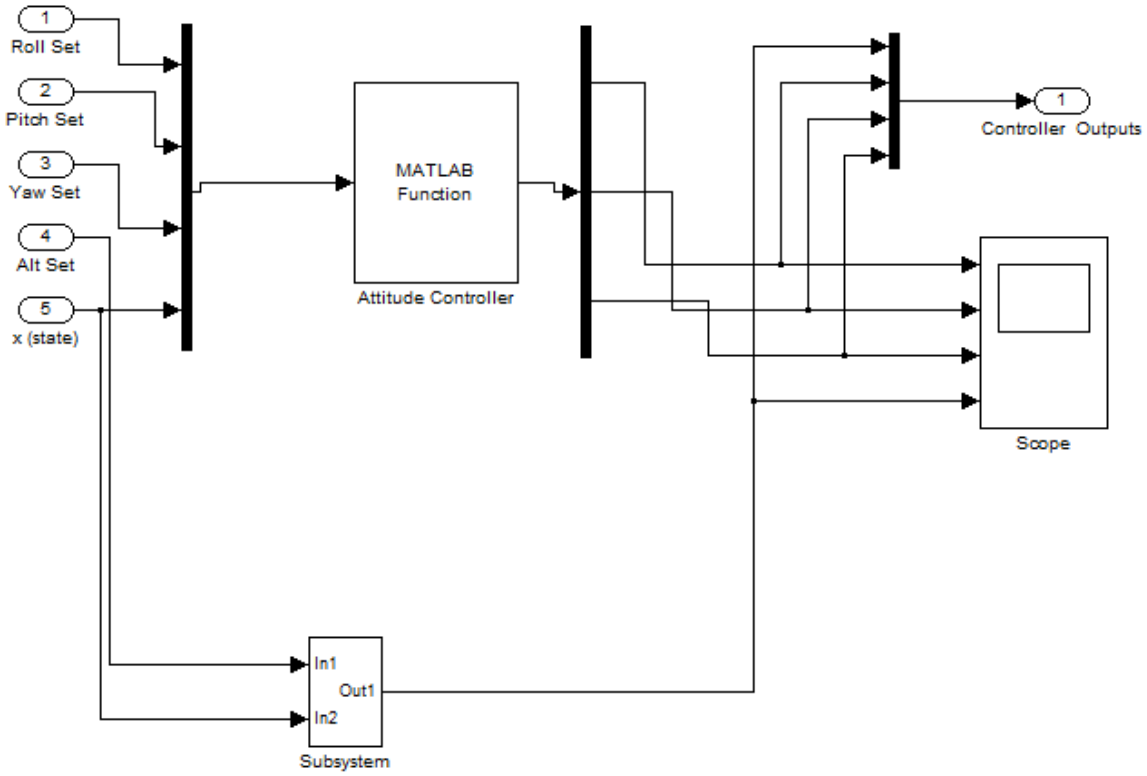


Figure 23: Controller Sub Block

The desired reference signals (Roll Set, Pitch Set, and Yaw Set) along with the state vector from the dynamics model are fed into the attitude controller sub block. The subsystem block towards the bottom of the figure which is fed the Altitude Set point along with the state vector is a disabled altitude controller. For the testing and tuning of the attitude controller, the altitude controller always outputs a signal that will equal the force of gravity on the craft. The outputs of the attitude controller are the roll, pitch and yaw signals, respectively. These (along with the constant signal from the altitude subsystem) are multiplexed together and fed to the controller block output. For debugging and tuning purposes, the roll, pitch and yaw signals are also sent to a scope for visual inspection. The MATLAB code for the attitude controller and a few other controller blocks is provided in Appendix A-2 through A-4.

Controller Tuning and Response

With the controller successfully implemented inside the MATLAB simulation environment, the next step was to tune the controller to the behavior of each attitude angle. Again, the roll axis will be used for this example, although the tuning method was applied to each axis individually. There are several tuning methods available to a controls engineer in order to fix the P, I and D gains appropriately to match a desired response. These include manual tuning (tweaking until the desired response is met), Ziegler-Nichols (tuning using a set algorithm), software tuning, and Cohen-Coon tuning (providing a step input, measuring the response, and setting parameters from this response). In the actual implementation of the platform, rejection of disturbances (wind, for example) is much more important than hitting the reference signal exactly every time. According to [12], the Ziegler-Nichols tuning method gives the loop exceptional disturbance rejection at the cost of slightly diminished reference tracking performance. For this reason, the Ziegler-Nichols tuning method was selected as a first pass algorithm. The method dictates that the I and D terms of the controller are zeroed out with the P term set such that loops output signal oscillates with a constant amplitude. For the roll axis, a P value of 1 resulted in the following output signal (Figure 24).

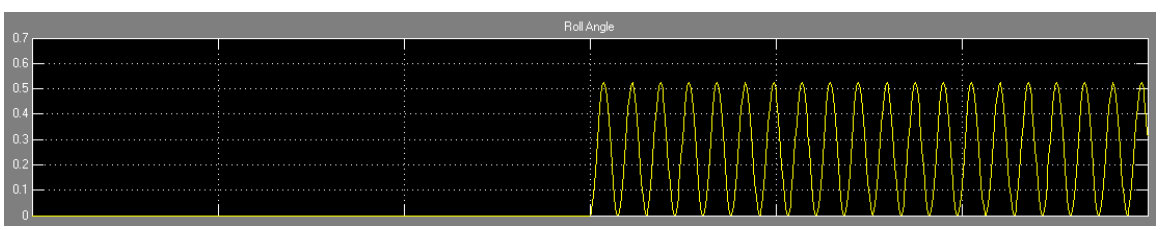


Figure 24: Roll axis ultimate gain oscillations

The oscillation period of this signal was determined to be 1.4 Hz. Using this signal along with the standard implementation of the Z-N method the following gains were set (Table 4):

Table 4: P, I, D values for Roll

P	0.5
I	0.8571
D	0.175 (standard)

As indicated in the table, the standard form (non-parallel) of the controller was utilized in selection of the I and D values. This only means that in the implementation of the output signal equation (46), the proportional gain value is actually applied to both the integral and derivative terms in addition to the proportional term. This standard method is widely encountered in industry, as opposed to the parallel ideal form which equation (46) currently illustrates. The slight differences of these two forms are shown in the figures that follow. Figure 25 shows the response under a slightly different set of parameters that would accommodate the parallel form.

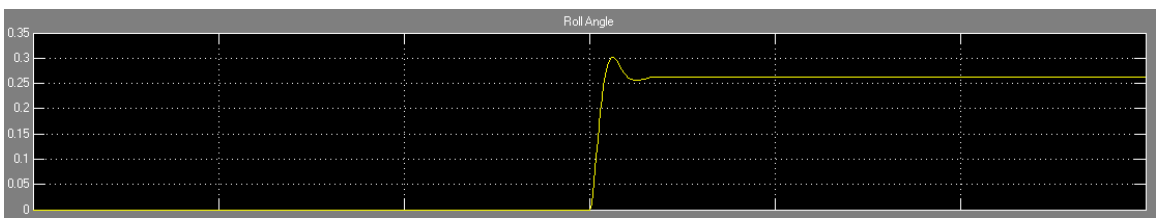


Figure 25: Parallel Form Gain Response

For a reference signal of 0.25 radians, the controller overshoots by 20% before quickly settling to the desired signal. While this is a decent response, overshoot should be avoided in most cases where non-acrobatic flight is required and a slower response time is permissible. This avoids fast oscillations to the actuators which may negatively impact the attitude of the craft depending on its dynamics. When tuned to the standard form of the modified process variable signal, the following response shown in Figure 26 is observed.

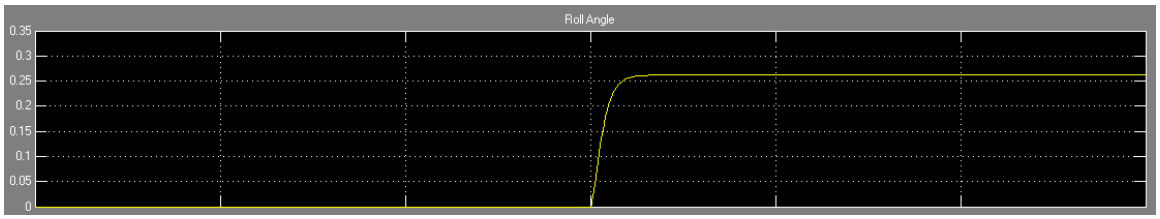


Figure 26: Standard Form Gain Response

In this response, no overshoot occurs at a slightly longer response time, which is the desired behavior. For these purposes, the Z-N standard method will be used, which utilizes the ultimate gain oscillation period only due already accounted for presence of the p-gain in the modified process variable signal.

With each of the independent attitude axes tuned separately, the controller can be tested with real values, and the response of each angle can be viewed and analyzed. For this example, the pitch and yaw angle reference signals will be set to 0 radians, and the roll angle reference signal will go from 0 radians to 0.2618 radians (15 degrees) after a time of 15s. Figure 27 shows the full controller merged with the simulation model, giving the experimenter control over whether or not to include the controller, and if included, how the controller gets its reference signals (whether through the signal builder or as a constant). Figure 28 shows the input reference signals as a function of time for this example.

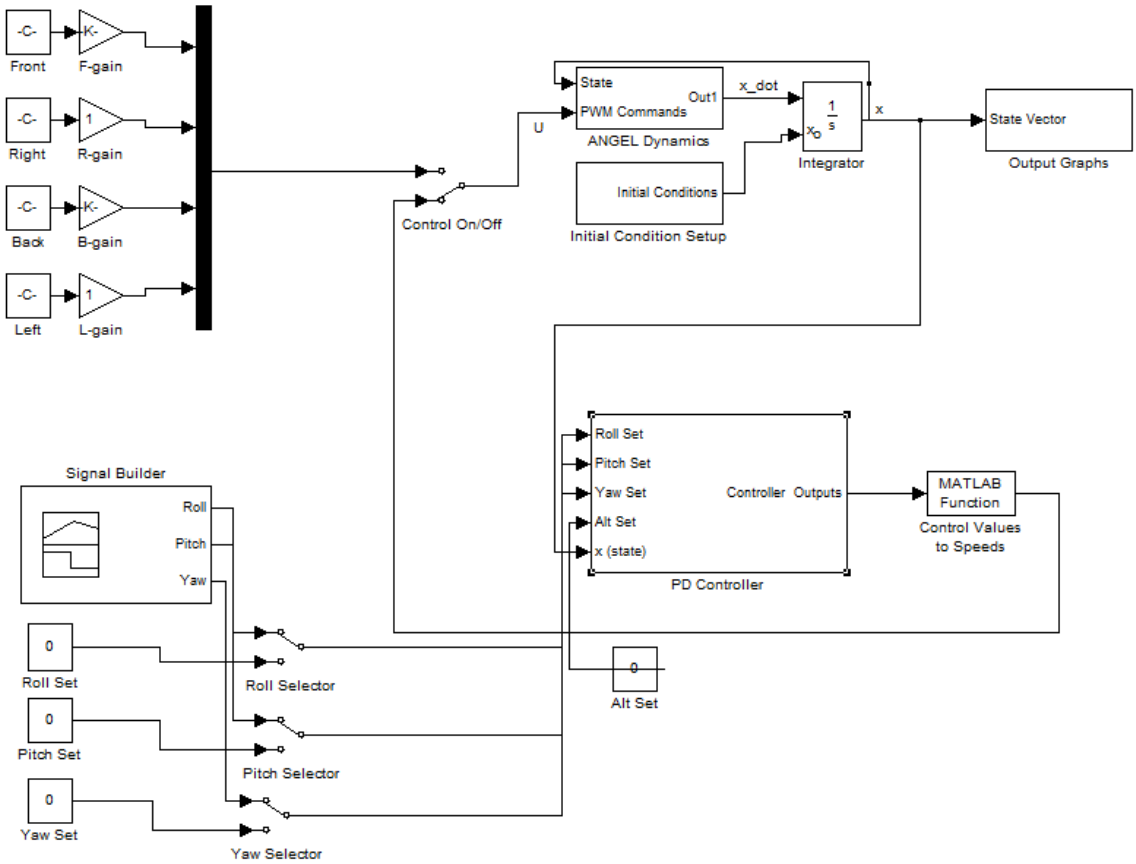


Figure 27: Full simulation and controller model

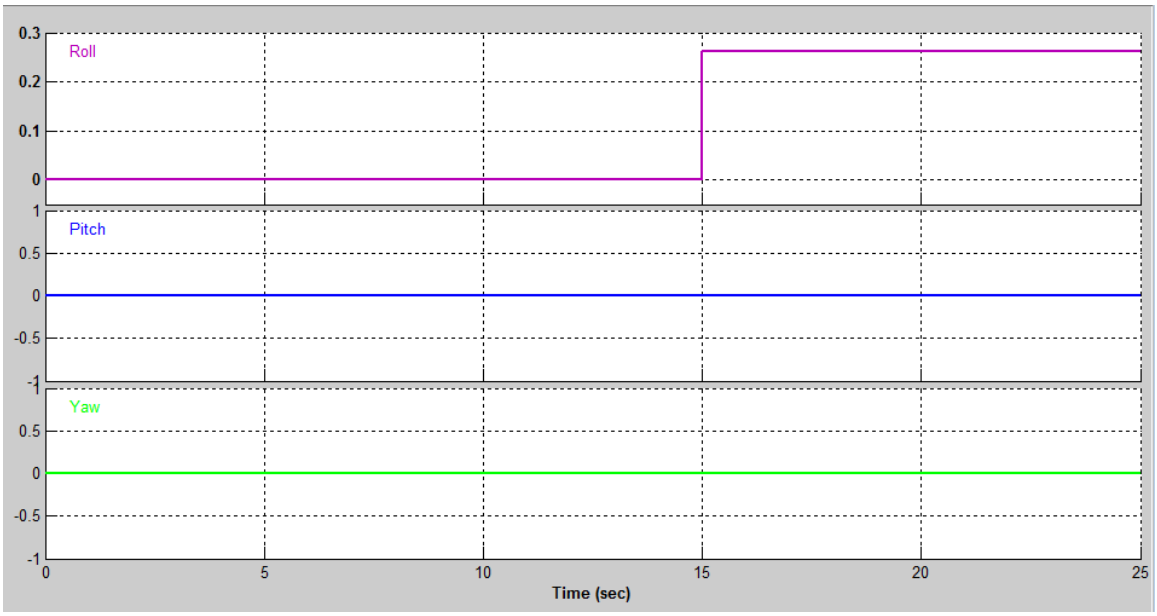


Figure 28: Controller example input signals

From 0 to 15 seconds, the system is in complete homeostasis, since the initial conditions are all set to 0. If this were not the case (if, for example, the initial condition for the roll angle had been set to pi/2) the controller would work to overcome this error from the start of the simulation. Figure 29 shows the actual roll, pitch and yaw angles as a function of time from the system dynamics block.

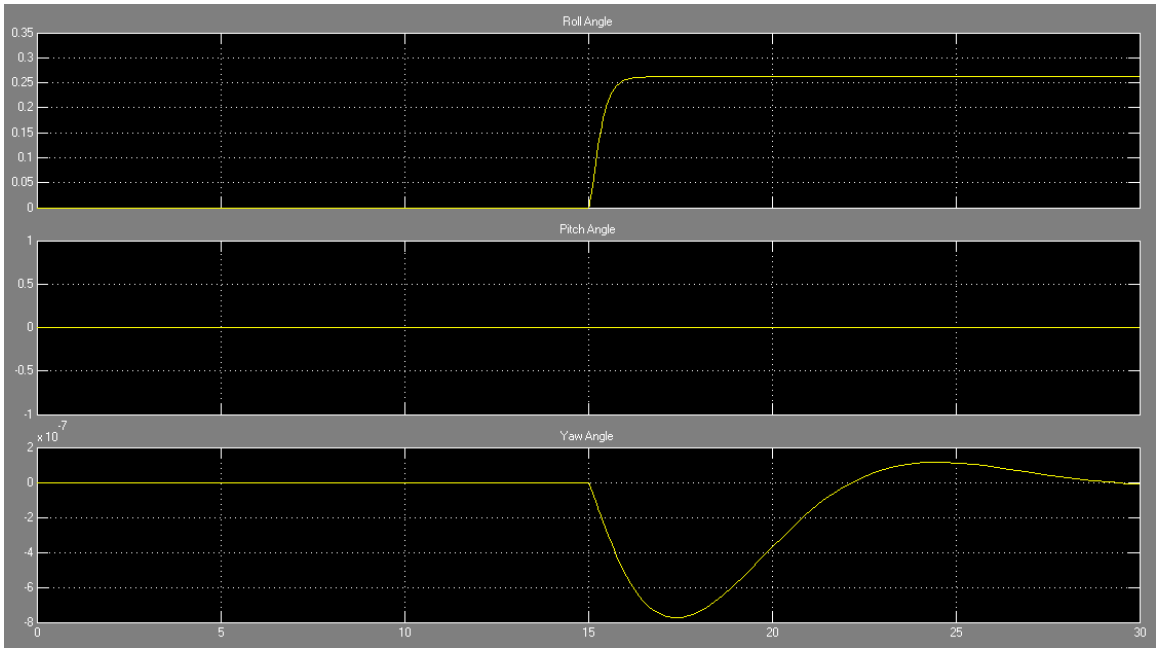


Figure 29: Roll, Pitch and Yaw response to the input signals

The roll angle response is exactly what was expected per the tuning of the controller. The angle matches the 15 degree reference signal change after the expected response time with no overshoot. The pitch angle, with a constant zero reference signal, stays at 0 for the duration of the simulation. The yaw angle exhibits a very small (8e-7) magnitude deviation in response to the controller shifting the thrust values for the roll axis signal. It corrects this minor deviation and returns to zero with little to no perceptible movement of the craft itself. This can be verified by studying the actual output of the controller (Figure 30).

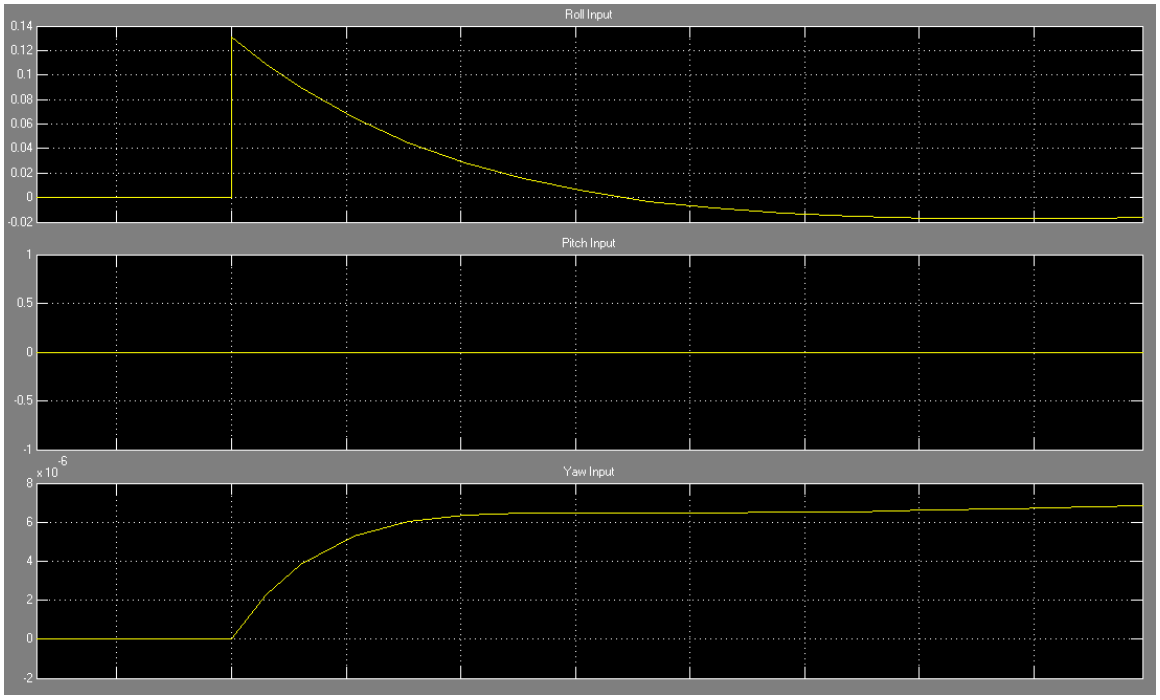


Figure 30: Controller output signals

From this graph, it is readily determined that the controller was swiftly responding to the changed input signal on the roll axis, exhibited by the sharp impulse in the output signal for the roll axis. Looking at the yaw output, the signal is not an impulse, verifying that the controller is changing in response to a shifting state vector value of the craft itself (error generated from state, not from reference).

Section V: Platform Implementation

ANGEL v1 Platform Basics

The first version of the ANGEL platform was designed and built over a period of roughly 2 months. While many basic design questions were answered by [10], the design itself, component placement, and all manufactured parts were completed by the principal author.

The first step in determining the layout of the platform was fixing a motor-to-motor distance. From the forces acting on the platform, it is evident that a small motor-to-motor distance would mean a larger force is necessary to actuate roll and pitch motions, while a larger motor-to-motor distance

would require much smaller input forces from the motor. In an effort to make the craft capable of capitalizing on small inputs while still being portable, an initial motor-to-motor distance of 24 inches was chosen. From this constraining dimension, a central hub was designed to join the 4 arms that would make up the platform body (Figure 31).

Figure 31: ANGEL v1 Hub

Please refer to Appendix B for CAD drawings of critical rapid-prototype parts. The hub (in addition to several other important parts on the ANGEL platform) was created using a Stratasys 3D printer. The choice to print the parts over building them from scratch provided several benefits during the design process. First, the parts could be designed precisely to within 0.0100” using CAD software, which yields tighter tolerances and closer approximations to the assumptions made in the simulations and modeling sections. Secondly, the printer prints using a thermoplastic material, which has similar strength properties to PVC. The downside to the printer approach is that the

thermoplastic is more brittle than conventionally used materials in aircraft (such as aluminum) and hence is more prone to fracture on impact. Table 5 illustrates the similarities between the ABS thermoplastic and conventional PVC.

Table 5: Properties of Thermoplastic and PVC

Properties	ABS Industrial Thermoplastic (Stratasys)	PVC (efunda.com) [2]
Tensile Strength (MPa)	37	41 – 45 @ yield
Tensile Modulus (MPa)	2320	2415 – 4140
Flex Strength (MPa)	53	69 – 110
Flex Modulus (MPa)	2250	2070 – 3450
Specific Gravity	1.04	1.3 – 1.58

From the properties listed, the thermoplastic is shown to have lower overall strength compared to the variants of PVC. However, the one property which is more desirable that the thermoplastic exhibits is a lower specific gravity. This weight reduction and fine detail available to the 3D printer made it a more desirable option.

All prints were done using a sparse-fill option. In this mode, the 3D printer creates a lattice structure in solid areas of the parts, reducing the weight of the part considerably without sacrificing too much strength. For large pieces that are not susceptible to direct impact force (such as the hub), this option was chosen.

The next design decision involved locating all the proper subsystems needed by the platform on its chassis. It was immediately determined that the majority of the weight should be distributed as low on the platform as

[2] The range of values for PVC is due to the presence of several variants.

possible, well below the plane of the rotors. Placement of the weight at a considerable distance would allow for a more stable platform that is better equipped to resist rolling or pitching by 180 degrees. With this in mind, a battery platform capable of holding the batteries and power distribution system was designed and placed just above the landing blades. Above this platform was a second custom designed shelf to hold the avionics needed by ANGEL. Figure 32 illustrates the subsystem location on the ANGEL chassis.

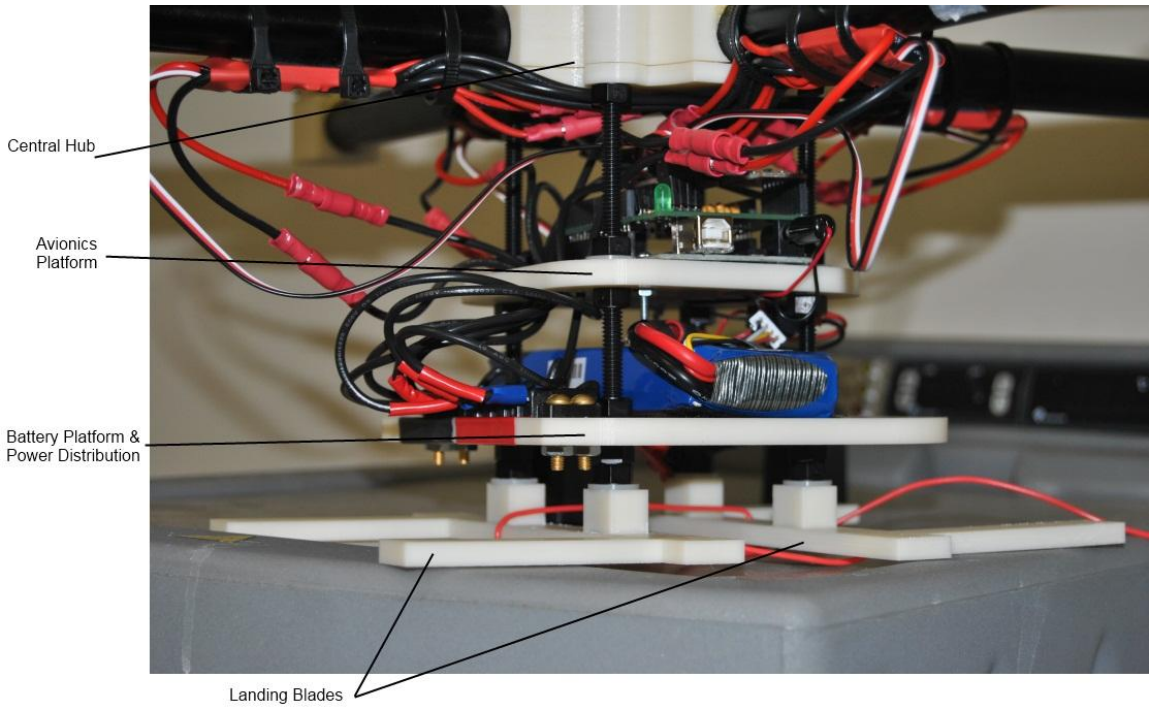


Figure 32: ANGEL v1 Subsystem Location

An additional advantage to this layout was the inherent protection provided to the avionics. The platforms were connected together by nylon all thread to decrease the overall weight of the system.

ANGEL v1 Power System

The power system employed in ANGEL v1 consisted of two separate lines. The first system line was used to power the electronic speed controllers (ESCs) and the motors. The second line was used to power the avionics. The decision to separate these lines was made in order to provide a cleaner,

steadier voltage to the avionics. Additionally, if avionics were powered from the main motor line, an expensive and heavy switching regulator would need to be used in order to efficiently bump the voltage down to a usable level. Providing the avionics with its own power source is a much cleaner solution.

For the motors/ESCs, a Zippy Flightmax battery was selected. For the avionics, a Rhino battery was selected. Parameters for both battery packs are listed in Table 6.

Table 6: Battery Parameters

Parameters	Zippy Flightmax	Rhino
Capacity	4000mAh	360mAh
Discharge	20C Constant/30C Burst	20C/30C
Voltage	3 Cell – 11.1V	2 Cell – 7.4V
Weight	306g	22.5g

The main motor battery (Zippy Flightmax) was chosen after looking into the requirements of the brushless outrunner motors selected as the actuators for the platform. The lithium polymer battery provides excellent energy density for the weight of the battery, and the 4000mAh capacity ensures adequate run time (depending on the all-up weight of the craft). Similarly, the Rhino LiPoly used for the Arduino exhibits the desired voltage for powering the avionics with enough capacity to outlast the motor battery at full charge.

The distribution system for the main 11.1V line is accomplished using an 8-position barrier strip. The battery supplied the power to the strip which was then distributed to the 4 ESCs used to control the motors. See Figure 33 for a wiring diagram.

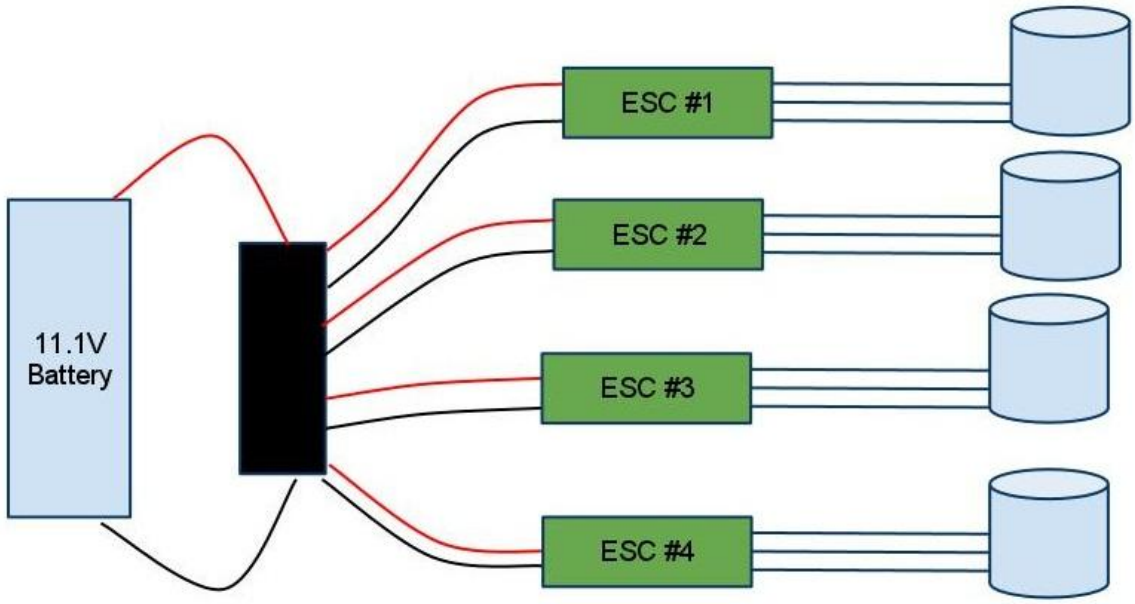


Figure 33: Power Distribution on ANGEL v1

The main drawback to the use of the barrier strip (shown as the black box in the figure) was its weight. A lighter solution was designed for the second ANGEL version, which will be discussed later.

ANGEL Actuators (ESC/Motor/Propeller)

The ANGEL platform has a total of 4 actuators that are responsible for orienting and translating the craft according to the outputs of the controller. The actuator lines are made up of the Electronic Speed Controller (ESC), the motors, and the propellers. The ESC gets power from the 11.1V line and an input signal in the form of a Pulse-Width-Modulation (PWM) command from the Arduino. The ESC then translates these PWM commands into the appropriate rotating magnetic field used to control the speed of the motors.

The motors selected for use on the ANGEL platform are Hacker KDA 20-22L brushless outrunner motors (pictured in Figure 34). These motors have a stationary internal core and windings with magnets on the outer cylinder. This outer cylinder is the portion that rotates. Since no brushes are involved and the only friction points are at the shaft, these motors are of much higher efficiency when compared to standard brushed motors. A

tertiary option would be in-runner motors, similar to brushed DC motors except with a stationary internal core. Again, since friction is minimized, in-runner motors have extremely good efficiency but do not have the desired torque output.

Figure 34: Brushless Outrunner Motor

These motors are attached to the ANGEL v1 platform with custom designed and rapid-prototyped motor mounts. These mounts (Figure 35) allow for easy wiring of the motor through the arms of the platform, and provide a flat standard mounting surface.

Figure 35: Motor Mount

The propellers selected for use on the ANGEL platform are a combination of slow flyer and slow flyer pusher APC propellers with a diameter of 10 inches and a pitch of 4.7”. The use of two different styles of propeller is important. If the same type of propeller were used for all 4 motors, 2 of the actuator combos would be highly inefficient due to the yaw requirement that 2 motors operate in a CW fashion and 2 operate in a CCW fashion. Essentially, two of the propellers would be spinning in a very inefficient manner, where the intended leading edge becomes the trailing edge (unless the propeller was flipped upside down). By combining regular propellers with pusher propellers along with the counter rotating motors, the efficiency of each actuator is maintained. The plastic molded propellers required careful balancing to ensure mitigation of vibration while running at high RPMs. This was achieved using a hobby prop balancer (Figure 36). Clear, low profile tape was applied to the propellers to help offset any inherent instability.

Figure 36: Propeller Balancer

The Turnigy Plush ESCs used on the ANGEL platform required programming before use to set operation characteristics matched to the ANGEL platform. Table 7 shows the parameters used.

Table 7: ESC Settings for ANGEL platform

Parameter	Value	Description
Electronic Brake	Disabled	Default for helicopters, saves battery
Batt Type	Li-XX	LiPo Battery is used
Low-Voltage Cut Off	Soft Cut	Slowly reduce motor speed when below voltage threshold
Cut Off Voltage	Med	Sets the level of the low voltage threshold. Guards against battery being discharged too low
Startup	Very Soft	Smooth startup (not racing)
Timing	Low	Depends on motor, good balance of power/efficiency.

ANGEL Main Avionics

The main avionics system for both implementations of the ANGEL platform is an Arduino MEGA. The Arduino platform was chosen due to its extreme ease of use, built in hardware, and wide range of add-on shields to expand its capabilities. The Arduino MEGA variant is based on the ATmega1280. It has 14 PWM pins (4 of which are used to send commands to the ESCs to control the motors), 16 analog inputs (each of which have 10 bits of resolution capable of analog values from ground to 5V), 128KB of flash memory, and a bootloader which allows for easy programming from a computer without the use of an external programmer.

The main function of the Arduino is to parse incoming sensor data, couple that data with the desired attitude sent by the user, and compute the proper commands to send to the motors to shift the craft accordingly. The Arduino therefore must be able to communicate with the sensors and the users controller, in addition to running the control loop to determine the correct orientation of the craft. Libraries to handle these job functions were written and implemented on the platforms and are available in Appendix A-5-1 through Appendix A-7-3.

ANGEL v1 Avionics Loop Description

The main avionics loop implemented on the Arduino is responsible for all control output and sensor aggregation input that together dictates the behavior and attitude of the craft. Once setup is complete the Arduino runs through an infinite loop, constantly comparing sensor values to reference signals and parsing user input.

The setup of the Arduino involves calls to the custom libraries written to interface with the sensor modules, the motors, and the control architecture. During setup, the craft is assumed to be on a flat level surface. When power is first applied to the Arduino, the gyroscopes and accelerometers responsible for reporting the attitude angle of the craft are zeroed with a self calibrating constant such that any acceleration measured should be due entirely to gravity. This calibration is necessary due to small, unavoidable movements in the sensors in between uses of the platform. The next step of the setup sequence is for the Arduino to establish a connection to the electronic speed controllers (ESCs) and the motors. By applying a prescribed pulse-width modulation command to the ESCs, successful control of the motors can be verified before entering the main loop. It is important the power be applied to the avionics system before the main motor power is applied. If the motors are powered before the system can send the initializing command, the motors will assume that something is incorrect with the controlling interface and will fail to start. This ensures that the correct type and range of command signal is being sent from the avionics to the motors. Following initialization of the motors, a series of instance creations for the objects defined in the libraries occurs. These include PID controller instance creations for each of the 3 principal rotation axes, and the complete inertial measurement unit instance creation where all controlling voltages are set so that calculations of attitude angles can be properly achieved. Once instance creation has been completed, a serial communication channel is opened through an Xbee interface in order for the user to send the platform commands from a computer wirelessly. With confirmation that this serial

communication has been successfully opened, the setup is complete and the ANGEL avionics system enters its main loop.

The first action of the main loop is to check the communications buffer in order to parse any waiting user commands sent through the serial interface. If a user command is present, the Arduino completes the required action and clears the buffer before proceeding with the rest of the loop. One main important feature of the ANGEL platform is a built in software safety that prohibits the motors from starting until a safety command has been issued from the user. This mitigates the possibility of an unintended motor start while the user is near the craft. If the safety and startup sequences have been satisfactorily cleared, the attitude angles are requested from the sensors (both the accelerometers and gyroscopes through a sensor fusion algorithm) and these values are passed to the respective axis PID instance for comparison to the reference signal. The controller instances each compute values necessary to force the craft towards the reference signal. These values are then combined together (for each axis and throttle) and sent through the motor controller instance to each of the four motors. This process repeats itself over again, with new sensor values being reported every loop. The main avionics loop is provided in Appendix A-8.

ANGEL Sensors

The subsystem which enables the ANGEL platform to perform as an *unmanned* aerial vehicle is the sensor network. Without this important subsystem, the craft would have no idea of its current orientation, and the controller would essentially be operating on an open loop with no feedback. The sensors serve the negative feedback to the controller that allows it to compare the current state vector to the desired state vector. The sensor subsystem on the first version of the ANGEL platform consists of an integrated five degree of freedom MEMS (Micro Electro-Mechanical System) IMU (Inertial Measurement Unit), which itself consists of the IDG500 dual-axis gyroscope and a the ADXL335 triple-axis accelerometer. This sensor allows the platform to track accelerations along the three principal axes of

translation (x, y and z) via the accelerometer and rotational speed around two of the principal axes of rotation (in this case, roll and pitch). In order to complete the IMU with a yaw rotation for tracking in all six degrees of freedom, the IXZ500 dual-axis gyroscope will be used in conjunction with the 5DOF system to give us a total of 6DOF available to the IMU with one axis on the added gyro left unused.

The 5DOF integrated MEMS IMU was selected for its small, low profile package and low power requirements. In order to derive meaningful data from the output of the accelerometer portion of the IMU, it is important to explain how it actually measures acceleration. Imaging a box with pressure sensitive walls, inside of which exists a sphere. As the box undergoes accelerations, the sphere impacts the walls, and these pressures are recorded and output by the sensor (Figure 37 [13]).

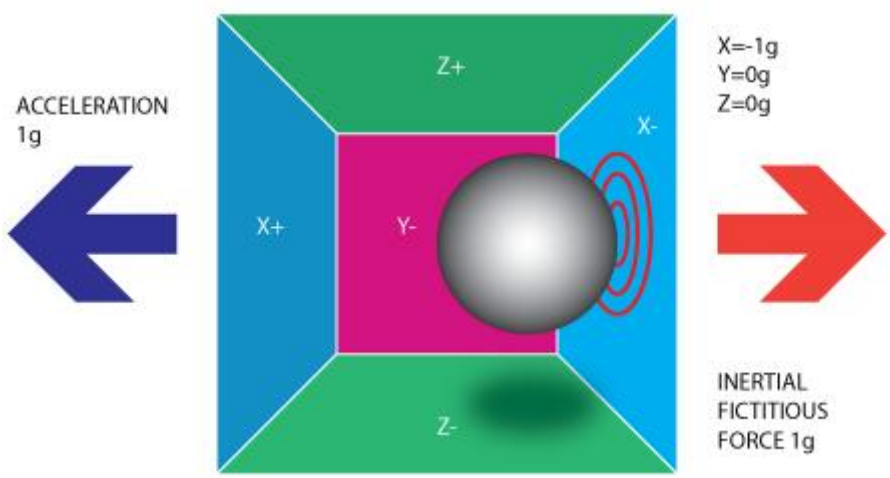


Figure 37: Example of how accelerometers measure force

In this manner, the accelerometer will detect a force in the opposite direction of the acceleration the craft is currently undergoing. It follows that accelerometers do not actually measure accelerations, just forces and pressure differentials against a bounding box. For example, due to the acceleration of gravity, the accelerometer would indicate a force in the –Z direction while the craft is sitting still on the ground.

The analog accelerometers in the chosen IMU provide the information about changing forces through varying voltage levels in a predefined range. For the ADXL335, this is anywhere between 270 and 330 mV per *g* of acceleration, with a typical value of 300mV/g for a given supply voltage. Additionally, the sensors have a prescribed zero-bias level, the voltage which is reported for each axis if no forces are detected. Using the actual reported value from the sensor after it has been converted by the Arduino's analog-to-digital converter (ADC) along with the zero bias level for the axis in question and the sensitivity, the voltage readings can be converted to acceleration vector components with units of *g*.

The Arduino ADC provides 10 bits of resolution, which means it will output digital values of 0 to 2^{10}-1, or 0 to 1023. For each channel of the accelerometer, the following equation can be used to convert the ADC values into useable voltage values.

$$Voltage = \frac{ADC_{out} * ADC_{ref}}{1023} \tag{48}$$

In the Arduino case, unless an external reference voltage is supplied, 3.3V is used as the ADC reference. ADC_{out} refers to the output value (0-1023) from the ADC. Once this voltage is determined, the next step is to remove the portion of the voltage that is reported when 0g is measured (the zero-bias). Combining this difference with the sensitivity of the accelerometer, the voltage is converted to meaningful data.

$$Acceleration_{channel} = \frac{\frac{ADC_{out} * ADC_{ref}}{1023} - V_{zero\ bias}}{Sensitivity} \tag{49}$$

This calculation is performed for each of the three channels to get acceleration values in the x, y and z directions. These values can optionally be fused together as a triplet to provide a Direction Cosine that indicates the resultant vector of the force.

The gyroscope (integrated IDG-500) measures the rate of change of rotation around an axis. The 5DOF IMU contains a 2-axis gyro, and an additional 2-axis gyro was added in order to track rotational velocities around

all three axes of rotation. The gyroscope reports the values of these rotational velocities in a similar manner as the accelerometer, and an equation must be derived to convert from the ADC values to the voltages and finally to the values of degrees/s that will provide a portion of the state vector to the controller. There are a few important differences between the sensors, aside from their measurement methodologies. The gyroscope actually provides two separate outputs per axis, one for standard measurements and one for high sensitivity measurements. The standard measurement axis provides a 500 degrees/s full scale range at a sensitivity of 2.0mV/deg/s. The high sensitivity output provides 110 degrees/s full scale output at a sensitivity of 9.1mV/deg/s. For the purposes of the ANGEL platform, where only the hover state is to be considered, the high sensitivity output can be used due to the diminished values of roll, pitch and yaw velocities. However, after further development into more aerobatic maneuvers, it will be necessary to switch to the standard output to take advantage of the larger full scale output range. Another important distinction has to do with the impact of the supply voltage on the sensor readings. Where they accelerometer outputs depended directly on the supply voltage (3.3V for the default Arduino case), the gyroscopes are not ratiometric to the supply voltage. This is an important distinction when changing the supply voltages. The accelerometer equations will need to be updated, because the sensitivities will change ratiometrically with the supply voltages, where the gyroscopic values for sensitivity will remain the same. An equation nearly identical to equation (49) can be used to determine the deg/s value for each axis as needed.

Sensor Fusion Algorithm and Noise

With an understanding of how the sensors report values to the main avionics processor and how the processor then converts the values to usable values, the next step is understanding how the processor uses the values from each of the three sensors together to make a meaningful and pertinent decision. There are typically three merging strategies used for combining

sensor data. The first is known as *correction*. In this instance, the data from one sensor is used to correct another. Second is *colligation*. This involves merging different parts of a sensor together and disregarding other parts. The last, and perhaps best, method for merging sensor data is *fusion*. In this manner, the values from each sensor are merged together in a weighted, statistical fashion to produce optimal results. For IMUs where the accelerometer is used to provide the direction cosine which dictates the overall direction of gravity and as a byproduct the attitude of the craft, the gyroscope plays a pivotal role. Due to the nature of acceleration measurement, the outputs fluctuate not only to changes in the gravitational vector (which is the desired output), but also to very small accelerations and disturbances (translational acceleration, thrust and altitude changes, wind, etc). This means the accelerometer outputs are inherently very noisy and prone to error. The gyroscopes are used to smooth out these errors to an extent. The gyros are, however, not without their own limitations. Although they suffer little from translational noise components (linear mechanical movements) due to their rotational measurement system, they tend to suffer heavily from drift and hysteresis. By statistically averaging the values from the sensors together, the attitude of the craft can be determined with a diminished noise and error component.

Typically, sensor fusion algorithms use a form of Kalman filtering to observe a noisy signal over time in order to produce a signal that is closer to the true value of the measurement. The Kalman filter approach typically uses the time domain principles of the noise with no regard to the signal transfer functions or frequency components. Alternatively, the Complementary filter approach concerns itself with analysis of the frequency domain with no consideration of the statistical description of the noise signal. The Complementary filter is actually simply a stationary (steady-state) Kalman filter. It has been determined that digital implementation of the steady-state Kalman filter is much simpler and efficient due to the assumption that the measurements are corrupted by stationary white noise [14]. A simplified version of the steady-state Kalman filter is derived and used in the sensor

library to provide the correct weights for each sensor. This algorithm is based on the technique covered in [13].

From the filter loop, an estimated resultant vector that accounts for measurement noise is expected as the output. The inputs available to the loop are the raw accelerometer values (Rx, Ry and Rz) and the raw gyroscope values (roll velocity, pitch velocity, yaw velocity). Using these values along with previous estimations (the previous output of the loop is fed back to the input), the corrected estimate is formed. For the first run through the loop, the accelerometers are assumed to be correct (zeroed and subjected to little or no noise) and the estimated value is set such that

$$R_{est}(n = 0) = R_{acc}(n = 0) \tag{50}$$

For step n, the previous estimate and the current accelerometer values are available to the loop as inputs, in addition to the rates reported from the gyroscope. Using the roll axis as an example, knowing the previous values of the estimate X and Z accelerometer values, the previous angle on the XZ angle of the resultant force vector can be determined as

$$\theta(n-1) = atan2(Rx_{est}(n-1), Rz_{est}(n-1)) \tag{51}$$

The atan2 function simply reports the angle in radians between a plane and the point provided by the function arguments. Using this previous roll angle, the new roll angle can be estimated from the gyroscope readings and the loop timing T.

$$\theta(n) = \theta(n-1) + \dot{\theta}(n) * T \tag{52}$$

Because the acceleration vector has been normalized to 1 and the individual vectors produced by these angles can be combined using the Pythagorean Theorem, the following x-axis direction vector can be derived from the gyroscopic readings. Similar values can be given for y and z directions.

$$Rx_{gyro} = \frac{\sin(\theta(n))}{\sqrt{1 + \cos\left(\theta(n)\right)^2 * \tan(\phi(n))^2}} \tag{53}$$

Now, for the nth step, the loop has the noisy $\mathbf{R_{acc}}$ vector and the computed $\mathbf{R_{gyro}}$ vector, both of which define the direction cosine of the overall acceleration on the craft. The loop finished by combining these values into $\mathbf{R_{est}}$(n) using a steady state weight, such that

$$R_{est}(n) = \frac{R_{acc} + R_{gyro} * weight_{gyro}}{1 + weight_{gyro}} \tag{54}$$

The gyro weight is an experimentally determined value that balances the emphasis put on the gyro measurements against the accelerometer measurements. It is essentially a measure of how much trust is placed in the gyro as compared to the accelerometer. In a normal Kalman filter, this weight is continuously updated based on the changing amount of measured noise. For the purposes of the ANGEL platform and with the limited computational power of the avionics controller, the constant weight will perform adequately. The last calculation the loops makes is to normalize the estimated R vector to 1. It will then repeat again using these newly calculated values as the (n-1) input for the next step. The functions for this estimation and user friendly reading of the sensor data are available in Appendix A-7-1 through A-7-3.

ANGEL User Control (Xbee and Processing GUI)

Early in the development phase of the ANGEL platform, it was decided that traditional user control methods would not be used. These methods generally include complicated custom controllers that either take too much devoted concentration to use, or too long to master for the craft to be used effectively. For the ANGEL platform to be used in a combat scenario successfully, it should respond to commands such as “Follow Me”, “Scout Ahead”, or “Follow this Path”. These high level commands are only achievable through either on board processing (feasible, but possibly cost or weight prohibitive) or through local processing of the high level commands

with low level actions sent to the platform by the controller. The latter approach was decided as the first past approach for user control.

With the delegation of processing determined, the next step was to determine the interface and the architecture of the controller. It was decided that a computer of some kind (in this case, a laptop) would be used to control and send commands to the ANGEL platform. This would also allow the user to view image and sensor feeds from the platform without designing a custom hardware controller. Next, the interface needed to be specified that would allow the computer (controller) to talk with the platform (Arduino avionics system). The following characteristics of the wireless network used to communicate with the platform were determined:

- High data rate is not necessary due to the nature of the commands being sent.
- Range should be large with relative high fidelity.
- Very low power consumption to conserver battery life on the platform
- Complexity should be minimized to mitigate in-field errors.

When considering each of these design requirements, the immediately defined candidate was the ZigBee wireless standard. While lower in data rate (Kbit range) when compared to Wi-Fi (11 or 54Mbits/s), Bluetooth (1 Mbits/s) or UWB (100-500 Mbits/s), ZigBee boasts incredible signal range (up to 1500m for the Xbee Pro line transceivers) at very low power consumption [15]. The ZigBee protocol is setup similar to a wireless serial link, and once paired, is capable of group communication. This would allow one controller to command several ANGEL platforms as a group or as individually addressable machines. The ability to expand off this protocol was another attractive feature of the standard.

With both the interface standard and the controller implementation defined, the next step was to design how the user would actually interact with the craft. As stated earlier, the end goal is to have the user issue high-level commands to the craft to minimize the focus controlling the craft demands. For the first implementation of the controller, however, it was determined that implementing low-level commands to control the craft and

tune it's parameters would make testing and debugging easier. A screenshot of ANGEL Controller v1 is shown in Figure 38.

Figure 38: ANGEL Controller GUI

The controller, even in its early alpha state, has several attractive features built in. The user starts by selecting the COM port through which the Xbee antenna will communicate. The action is recorded to an on-screen, live updating communications log. The ANGEL platform acknowledges the connection, which is also displayed in the log. From here, the user can choose to enter debug mode or stay in standard flight mode. Debug mode enables graphical outputs of all sensor data to ensure the proper signals are being received. In order to turn on the motors, the user must manually disengage the electronic safety. Only after this will the motors receive commands. Other functions of the GUI include a MOTOR PULSE commands to test that all motors are responding properly, a ZERO SENSORS command to re-zero the IMU in case the craft was powered on while in a non-zero position, a START command to turn the motors on to their lowest power setting, and a KILL command to shut power to the motors off and reengage the safety. For controller testing and tuning, the PID values of the axes (roll axis shown) can be updated directly from the controller. This allows the

debugger to carefully tune out slight oscillations. Although it is not visible in the above screenshot, the main control of the platform appears in the large blank area to the right. A square that tracks mouse positions allows the user to control roll and pitch by using the x- and y- axes of the track pad on the controlling laptop. This version of the controller was developed with Java and Processing. The source code for the controller appears in Appendix A-9.

Eventually, these functions will be removed and replaced by a high-level GUI that parses user commands into functions the platform can interpret. The controller could eventually be compressed into a more mobile friendly device, such as a smartphone or tablet. This would free the end user from wrestling with a cumbersome controller and allow them to focus on the operation at hand.

Control Library Implementation

The controller designed and tested in MATLAB from Section IV provided a great testing opportunity for the actual implementation of the controller within the on-board microcontroller. In this subsection, details of the implementation of the controller on the Arduino and corresponding libraries written to achieve this will be discussed.

Keeping with the object oriented approach to the design and architecture of the code used on the ANGEL platform, a PID controller class was written in C++. By writing the class and assigning it the necessary private variables and functions, a separate instance of the class could be declared for each axis. Thus, three controllers are easily created from a single class, while still maintaining the needed customization to account for inherent inaccuracies between the model and the implementation. The header and source code files are located in Appendix A-6-1 and A-6-2, but an explanation of the class and its methods is given below.

To declare a new PID instance, the following line is used:

$$SchmidtPID(float\ P, float\ I, float\ D, float\ windup) \tag{55}$$

This declaration allows the user to set a specific P, I and D value for the axis at hand. The windup value is a guard against massive accumulation of error in the integration term while the machine is "warming up". This windup is essentially the maximum negative and positive values for the accumulated integration error value. If the platform experiences high oscillations during take-off due to the aforementioned ground effects, where the platform experiences a thrust advantage, the windup guard minimizes the range of the integrated error, which would inaccurately bias the i-term for the moments just after take-off.

The class gives the user access to 6 methods used to interact with the controller. *updatePID(float target, float current)* is the main function of interest. Calling this method calculates the appropriate P, I and D factors using the various derivatives and integrations of error as outlined in Section IV, along with the P, I and D constant values defined in the class instance creation. The target is the desired angle offset (in the case for hover, this would be 0), and current is the main attitude of the craft. Both the target and current values are computed and reported by the avionics loop from the user data and the sensor data.

The second method made public is the *setValues(int P, int I, int D)* method. This allows the user to change the values for the P-gain, I-gain and D-gain after the class has been initialized. Through this method, in-flight adjustments can be made to the controllers for each axis individually through the GUI detailed in the previous subsection.

The third public method is the *zeroError()* method. This returns the accumulated error of the I-term to the initial value of 0. This is useful if the craft is being manipulated by hand for testing purposes.

The last three methods are *getP(), getI(),* and *getD()*. These methods simply return the currently set P, I and D gains. These functions were built in for the user to have constant awareness of the state of the controller, as well as for future development of an auto-tuning algorithm for the controller.

ANGEL v2 Build Description

After initial testing and implementation of the ANGEL v1 craft with limited success, it was determined to parallel development of a secondary craft to dynamically update and change craft parameters and test new build techniques. There were several changes to be implemented between the unnamed v1 craft and the v2 craft, named 'Uriel'.

1. While the low battery placement on v1 was ideal for stability, it negatively affected how the craft handled in more aggressive roll and pitch movements. It was decided that Uriel should be more compact while still maintaining a majority of its weight under the plane of the propellers.
2. The landing gear on the v1 craft was too fragile, and located too close to the central column, leaving the craft largely unbalanced on slightly uneven terrain. Updates to Uriel would include landing feet below each motor that could be easily changed if broken.
3. While the individual tiers of the v1 craft meant easy subsystem mounting, it was not friendly to modifications or battery replacement. This problem would be addressed in the Uriel build by reorienting the placement of the subsystems.
4. During crashes in the test flights of the v1 craft, it was apparent that the likely points of failure were the motor mounts (extremities of the craft). A main design change on the Uriel platform was removable arms to quicken the process of changing out bad motors or broken mounting equipment.
5. The v1 craft lacked a good location for mounting an ultrasonic altitude sensor. A position for this sensor was made in the Uriel design.

Figure 39 shows the drawing of the Uriel platform. The smaller overall size and detachable arms fell in line with the overall systems approach to the design of the craft as something that would be easily transported in a satchel or backpack.

ANGEL 2.0: "URIEL"

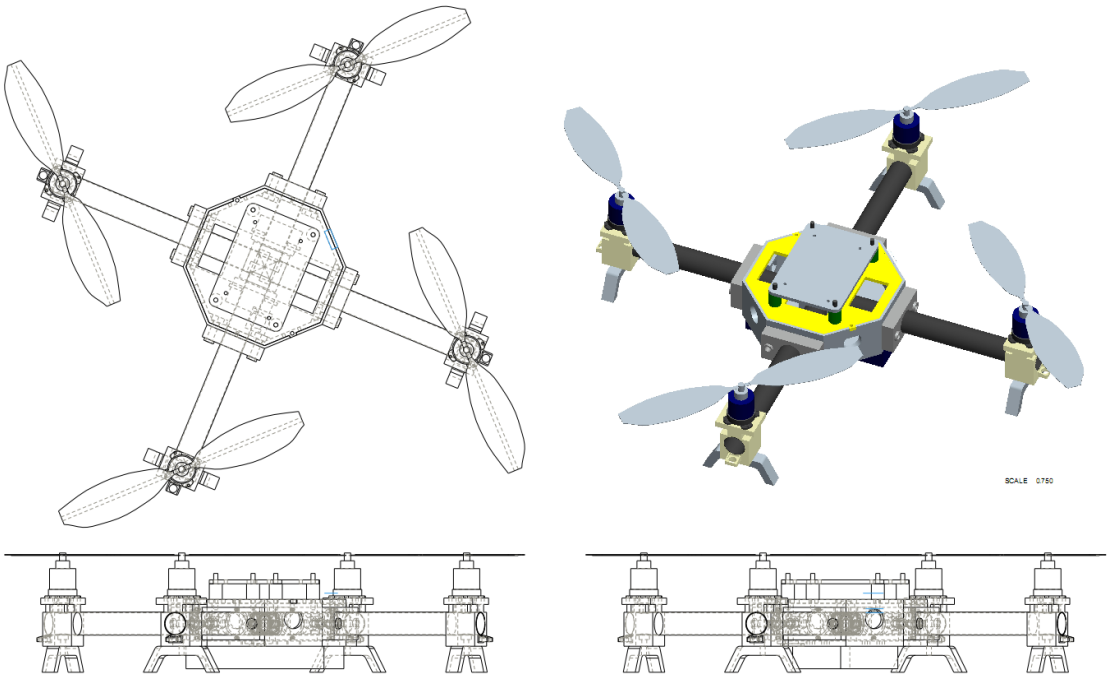


Figure 39: CAD Diagram and 3D model of Uriel build

The motor-to-motor distance on Uriel is smaller at 18” when compared to the 24” parameter on the v1 build. The battery is strapped to the underside of the main chassis, thus mitigating any increased roll/pitch behavior during aggressive maneuvering. This also facilitates in easy battery removal and charging.

A more advanced wiring interface was used on Uriel. Instead of relying on a heavy distribution hub, a series of custom-made Deans-Y connectors were constructed and used. This made it much easier to ensure proper wire connections and decreased the amount of excess wire needed. Figure 40 shows an example of the Deans-Y connectors made for the craft.

Figure 40: Dean-Y connectors made for Uriel

Vibration dampening bolts were added to the mount points for the avionics sensor board. It was determined that much of the inaccuracies of the accelerometers could be reduced by eliminating the vibration noise from the craft. Additionally, foam coated rapid-prototyped blocks will be added beneath the sensors themselves to keep them from vibrating at the pin out connection points on the sensor board.

The next section discusses some of the testing experiments and results for the v1 craft and further discusses how some negative results were overcome and updated with the modifications to the Uriel craft.

Section VI: Testing and Results

Testing and Results Introduction

In this section, some of the tests performed on the ANGEL platforms will be discussed. The data garnered from these tests will be shown and the ensuing analysis and decisions will be explained. The last section (Section VIII) will go into more depth about the conclusions of both ANGEL builds and the future development needed to make it a fully functioning platform.

Test Bench and Flight Harness Construction

To aid in the flight testing of the platform, two separate test benches were made to facilitate debugging while providing a buffer to catastrophic flight crashes where possible. The first test bench provides for no thrust and was only used to test the reaction of the main avionics loop to changes in the

sensor data. This test bench was vital for making minor adjustments to the PID controller instance in the avionics loop to tune out minor oscillations. It, however, did not allow for any testing of sustained random disturbances. The test bench is shown in Figure 41.

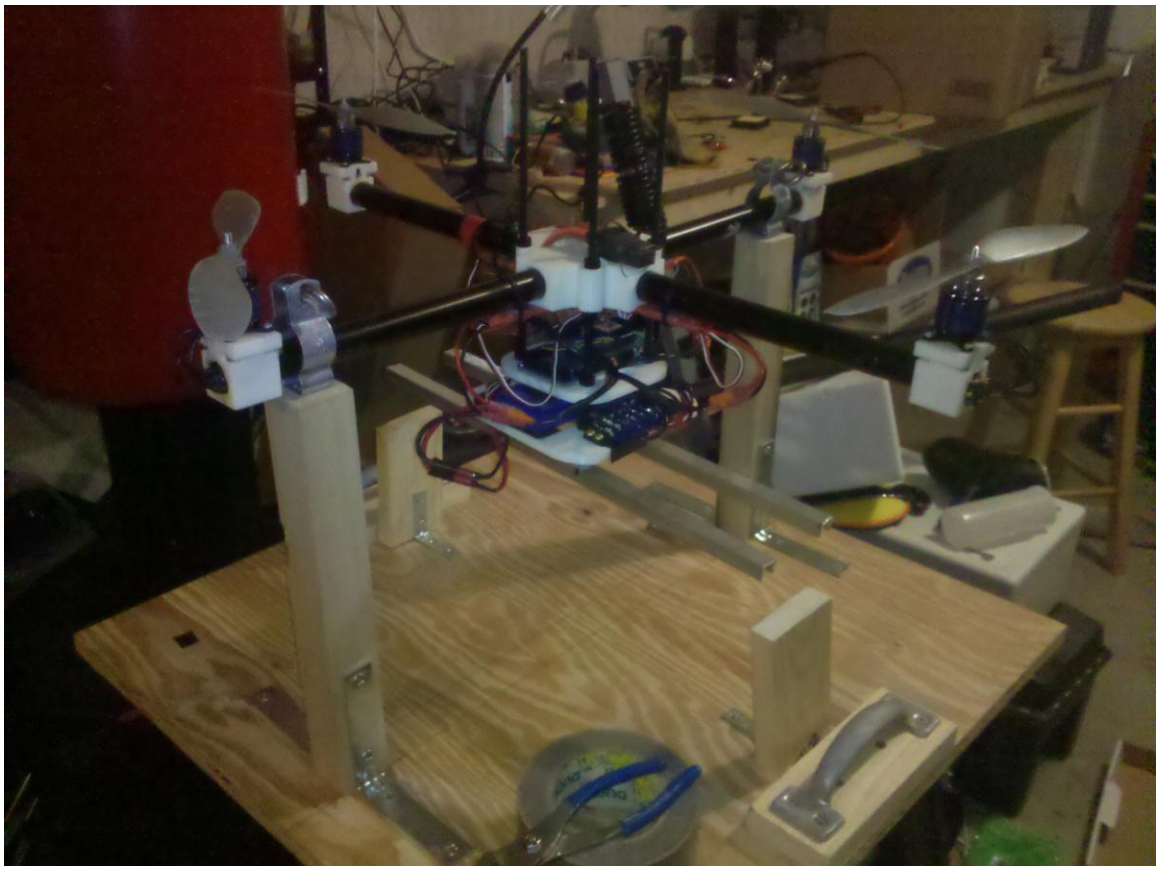


Figure 41: Roll and Pitch Axis Test Bench

This bench, however, did not allow for any yaw movement testing or testing without the stability provided by the bench.

The second bench that was constructed consisted of two guide wires rigidly mounted to the ceiling and pinned to the ground. Small eye hooks were added to the ANGEL platform and the guide wires were threaded through these hooks, allowing the platform to translate up and down with small roll and pitch movements but no yaw movement. It also kept the platform in a vertical column to keep it from translating in the earth fixed XY plane. This system is shown in Figure 42.

Figure 42: Flight Harness

Thrust Measurement

In the development of the controller, the input values for the simulation were specified in terms of the rotor speed (rotations per minute). This provides a number directly related to the thrust output in Newtons from the rotor/motor combo. However, this does not include such factors as battery power effects or inefficiencies in the actuator line (ESC-Motor-Propeller). To test a series of motor/propeller combos, a thrust stand was constructed to directly measure the force exerted by the actuator as a function of changing PWM input. Figure 43 shows this thrust stand. These figures were backed up with the use of a handheld tachometer to verify the relationship between output thrust and propeller speed. Future development

of a voltage monitor will scale the PWM-RPM relationship to changing values of the battery output.

Figure 43: Thrust Stand

Pitch and Roll Test Data

The main source of error in the testing process was making sure the errors and noise from the sensors were sufficiently ignored. The sensor fusion algorithm previously discussed did a good job of smoothing out these noisy values. A custom data logger was written in Java to grab the sensor data from the fusion function and the reported motor commands from the controller over the Xbee network link. Figure 44 shows the test results for the roll axis on the test bench, and Figure AC shows the test results for the pitch axis on the test bench. These results show the actual implementation of the controller as it drives the inputs to the motors in response to the sensor data and the set point of stable hover (zero radians) on both axes.

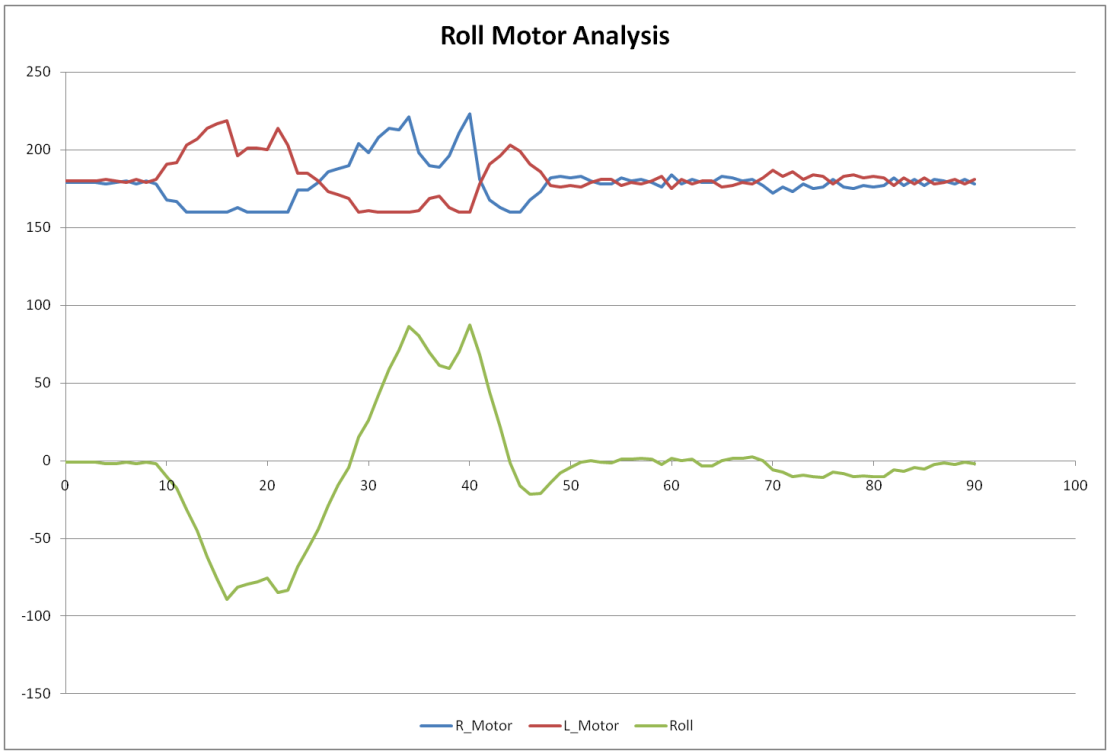


Figure 44: Roll Axis Test Results

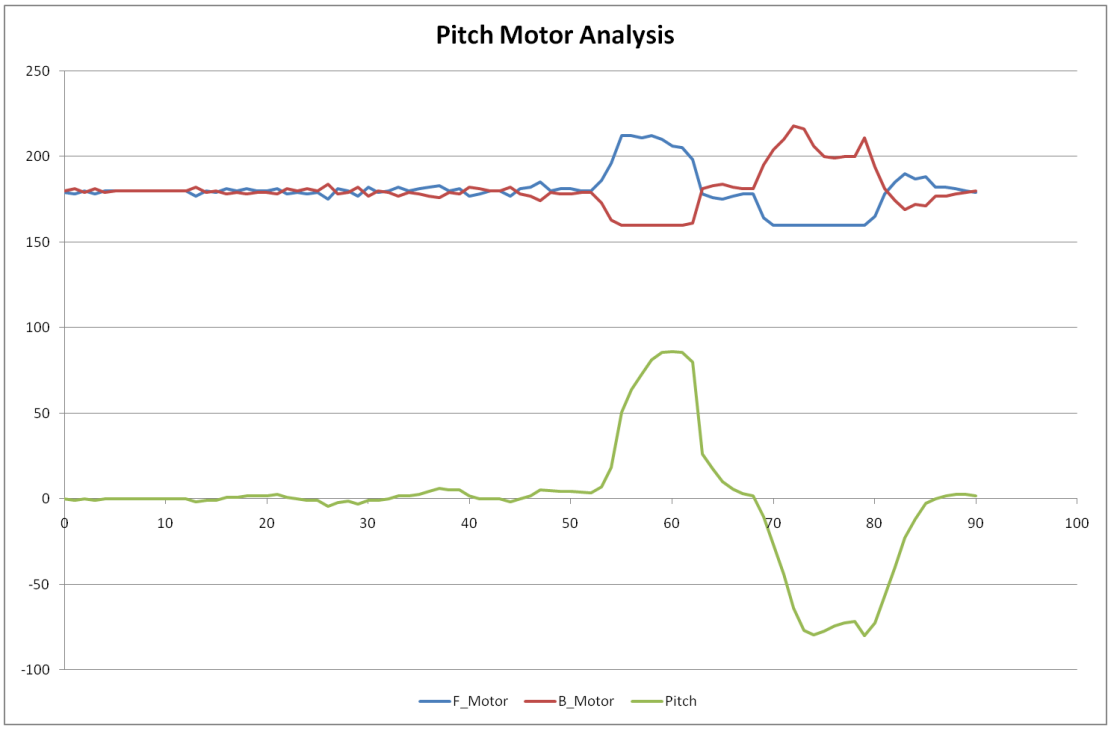


Figure 45: Pitch Axis Test Results

Pitch and Roll Test Results

Although the data indicates proper response of the control system to the reported sensor values, the actual flight test had much different results. A free test flight took place after the test bench check, which resulted in a "belly-up" scenario for the craft. Since the controller was not tuned to this portion of the flight envelope, the platform quickly plummeted to the ground. Further inspection of the actual platform response and testing with a tachometer uncovered a problem with the roll axis. At first, it seemed as if one of the motors was underpowered. The axis always seemed to favor one side over the other at higher speeds. Two replacement ESCs were interchanged with the roll axis ESCs and all the connections were checked. After an extended period of debugging to further isolate the issue to the ESCs, it was determined that one of the speed controllers was overpowering its motor beyond a certain input signal threshold. This ESC was replaced and testing of the platform continued.

Avionics Loop Testing

Another source of error during the testing process was the timing of the avionics loop. Originally, the loop was set to run as fast as possible, listening to commands at the beginning of the loop, then grabbing updated sensor data, and finally outputting the correct motor commands to the actuators. It was determine, however, that a loop timing mechanism was needed in order to section off time for receiving commands, grabbing new sensor data, and updating the motors in accordance with the individual subsystem update interval. The problem manifested itself when several commands over the Xbee network stacked themselves on the buffer, resulting in a large and unintended boost in thrust output of the actuators.

Section VII: Concluding Remarks and Future Development

Simulation Conclusions and Future Work

The simulation programmed in MATLAB and Simulink made the development and testing of the controller implemented on the platform much more straightforward. While the model worked sufficiently for the near-hover flight envelope, the hub forces and air friction forces neglected would need to be added in for the model to be even more accurate. Additionally, the gyroscopic effects ignored in the development of the equations of motion could be revisited in future revisions of the simulator.

In addition to adding in the removed assumptions, certain subsystems that were assumed 100% efficient would need to be modeled more realistically. For example, the motors are not 100% efficient, as they do not immediately output based on their signal input. This delay could be simulated with a first-order model of the DC motor and would allow for more testing and simulation of actuator limits and bandwidth. This model would be formed from a gain and a pole that is related to the timing constant for the motor, and it would be tuned to each of the actuators on the platform individually.

The sensors could also be modeled to improve the accuracy of the simulation. Instead of assuming that the sensors will report the actual true value of the craft based on the equations of motion, using a sensor model would account for the inherent inaccuracies of the inexpensive sensors used in the platform.

Lastly, environmental effects such as collisions, wind, temperature and precipitation could be added to an advanced simulation in order to fully test the platform in a variety of conditions.

Controller Conclusions and Future Work

The PID controller developed for the ANGEL platform is very straightforward, which made it (computationally) easier to implement and debug. However, as the platform is developed further, a more aggressive and capable controller will need to be used to handle aggressive movement, way

point tracking, and other higher level features. While the current implementation of the controller is actually comprised of 3 Single Input Single Output (SISO) systems for each of the principal attitude axes, a Linear Quadratic Regulator (LQR) could be implemented to treat the system as a whole as a Multiple Input Multiple Output (MIMO) system. From [4], it is shown that the LQR would be an improvement on the PID controller, but both implementations are poorly equipped to reject strong disturbances. An alternative attempt at correcting the disturbance rejection was to design the system based on a backstepping technique, where outlying unstable subsystems are progressively stabilized as new controller are developed while "stepping back" from the core stable system. Although this technique results in strong disturbance rejection, it loses the robustness of stabilization in near-hover flight. By combining these controllers with a special form of backstepping known as integral backstepping, [4] found the proper balance of disturbance rejection and autonomous stable hover. Updates to the ANGEL platform would push the control architecture more towards a system similar to the description in [4], which would allow the platform to function in a wider flight envelope, as well as open the possibilities of autonomous take-off and landing.

Sensors and Fusion Algorithm Conclusions and Future Work

The platform saw great improvement after the simplified steady-state Kalman filter was designed and included to merge data from the gyroscopes and accelerometers. This filter was able to account for the long-term drift present in the roll and pitch axes of the gyroscope by pairing the short term accelerometer data, and the noise of the accelerometer was smoothed out by the steady weighted inclusion of the gyroscope. However, the long term drift of the yaw axis is still unaccounted for. Inclusion of a magnetometer to provide a heading reading would allow the yaw drift to correct itself. While the filter implemented on the craft does a decent job of providing an estimation of the true sensor values, this can be improved. By adding functions to track the noise, the weight that each sensor system plays in the

overall fusion could be adjusted in real time, thus moving away from a steady state Kalman and more towards a standard Extended Kalman filter. Subsystem modifications planned for Uriel (the dampening bolts for the avionics platform and wedge pieces for the sensors) will also help mitigate the noisy sensor signals, making it easier for the filter to provide a true signal estimation.

Other sensors remain to be implemented on the platform. A downward facing ultrasonic sensor is planned for the further development of the Uriel platform. This will allow the system to have a reference of how far it is above the ground. This sensor alone is not sufficient for altitude determination above a certain threshold and would need to be combined with a barometer to estimate true altitude. Together, these sensors would allow for altitude control and autonomous take-off and landing, as well as perching movements. Additionally, integration of a GPS would be vital to waypoint following and true autonomous flight.

User Interface Conclusions and Future Work

Although the user interface currently used to send commands to the platform is sufficient for debugging and testing, it would need large improvements for use in the field. As it is used now, the interface is not ideal for set-and-forget flight. Some testing has gone into compressing the controller into a more user-friendly format such as a small touch-enabled device like an Android based phone or an iPhone/iPod Touch. This would allow the end user to strap the controller to his/her arm and give commands without pulling focus from the task at hand. Early testing shows connection and control of the platform from a small handheld touch enabled device is quite possible, with the ability to forward camera feeds and sensor data from the platform back to these devices so the user can make decisions about further mission parameters.

Although not necessary for the immediate development stage of the platform, the existing interface DOES support control of multiple crafts in tandem through the broadcast of the Xbee system. This feature would be

expanded to include individual addressable control and possible swarm tactics as the need for such methods arises.

Physical Build Conclusions and Future Work

The prototypes built around the development and research of this thesis have proven to be invaluable in testing how robust the simulation and controller models are in real flight scenarios. Several updates were made between the creation of the v1 system and the updated version of the ANGEL platform, the Uriel system. Unfortunately, the budget used for the ANGEL platform ended near October of 2010, which resulted in a halt on prototype development for the last 7 months. Personal funding of the Uriel subsystem resulted in the implementation of several good design changes, but until a more stable funding source is available, the needed prototyping development cannot proceed. On the v1 system, a two of the motors need to be replaced, and several sensors need to be added. On the Uriel system, an ESC needs to be replaced and the vibration dampening system needs some further work. Once more funding is secured, these parts and changes can be implemented and further testing can proceed. Until such time, development and testing will be limited to a simulation environment.

One constantly updatable parameter of the prototypes is reduction of weight. This will allow for longer flight times and more agility for in flight maneuvers. The weight and complexity of the prototype can be reduced by switching to a small Gumstix-like on-board computer with a more integrated sensor board. Additionally, updates to the power distribution system can further decrease the weight of the craft and make repairs more efficient.

Thesis Objective Conclusion

The overall objective of this research endeavor was to approach the design of a man-portable UAV from a systems engineering standpoint. The focus was to be on the system and its use as a whole, not isolating any one or two subsystems or using an external environment to facilitate the craft's

movement in the terrain. The derivation of the modeling equations and the raw implementation of a simulation model and controller allowed for the understanding of the physical characteristics that dictate the behavior of the craft. These models were tuned based on the specific craft parameters of the prototypes built to test the actual flight of the platform. The platform is comprised of several subsystems, each of which have been studied in-depth to understand how the various subsystems can work together for synergistic benefit. These systems include the avionics, sensors, actuators, chassis, and user control architecture. The test flight experiments performed using the v1 build prototype indicates that successful flight is possible with adequate funding. Although faulty sensors and subsystem components paired with a lack of continued funding kept this current prototype confined to the test bench, small tweaks to the avionics loop and the addition of new sensors and ESCs promise to make for a stable, autonomous platform.

APPENDIX A – CODE

A-1 – System Dynamics MATLAB Code used in Simulation

```
function [Xout] = sysdyn(Xin)
%SYSDYN This function computes ANGEL system response given a state vector and PWM
%input
%   Xout = [X_ddot Y_ddot Z_ddot Roll_ddot Pitch_ddot Yaw_ddot X_dot Y_dot
%   Z_dot Roll_dot Pitch_dot Yaw_dot]
%
%   Xin = [X_dot Y_dot Z_dot Roll_dot Pitch_dot Yaw_dot X Y Z Roll Pitch
%   Yaw F_PWM R_PWM B_PWM L_PWM]
%
%
 % this loads the necessary constants into the workspace
angel;
%---------STATE DEFINITIONS-------------------

X_dot=Xin(1); %X-axis Velocity (m/s)
Y_dot=Xin(2); %Y-axis Velocity (m/s)
Z_dot=Xin(3); %Z-axis Velocity (m/s)
Roll_dot=Xin(4); %Roll Velocity (rad.s^-1)
Pitch_dot=Xin(5); %Pitch Velocity (rad.s^-1)
Yaw_dot=Xin(6); %Yaw Velocity (rad.s^-1)
X=Xin(7); %X position (earth) (m)
Y=Xin(8); %Y position (earth) (m)
Z=Xin(9); %Z position (earth) (m)
Roll=Xin(10); %Roll Angle
Pitch=Xin(11); %Pitch Angle
Yaw=Xin(12); %Yaw Angle

%---------INPUT DEFINITIONS-----------

F = Xin(13); %Front Motor Speed
R = Xin(14); %Right Motor Speed
B = Xin(15); %Back Motor Speed
L = Xin(16); %Left Motor Speed

%--------THRUST CONVERSIONS----------

TF = b*F^2; % front thrust calculation (N)
TR = b*R^2; % right thrust calculation (N)
TB = b*B^2; % back thrust calculation (N)
TL = b*L^2; % left thrust calculation (N)
D = d*(-F+R-B+L);

%--------SYSTEM DYNAMICS------------

% X_ddot = -(1/craft_m)*(cos(Roll)*sin(Pitch)*cos(Yaw) + sin(Roll)*sin(Yaw))*(TF+TR+TL+TB);
% Y_ddot = -(1/craft_m)*(cos(Roll)*sin(Pitch)*sin(Yaw) + sin(Roll)*cos(Yaw))*(TF+TR+TL+TB);
% z_craft_component = (1/craft_m)*cos(Roll)*cos(Pitch)*(TF+TR+TL+TB)
% Z_ddot = g - z_craft_component;
% Roll_ddot = Pitch_dot*Yaw_dot*(Iyy-Izz)/Ixx + (arm_l/Ixx)*(-TR+TL);
% Pitch_ddot = Roll_dot*Yaw_dot*(Izz-Ixx)/Iyy + (arm_l/Iyy)*(TF-TB);
```

```
% Yaw_ddot = Roll_dot*Pitch_dot*(Ixx-Iyy)/Izz + (D)/Izz;

%---------SIMPLIFIED MODEL FOR CONTROL USE (LINEARIZED)
X_ddot = -(1/craft_m)*(cos(Roll)*sin(Pitch)*cos(Yaw) + sin(Roll)*sin(Yaw))*(TF+TR+TL+TB);
Y_ddot = -(1/craft_m)*(cos(Roll)*sin(Pitch)*sin(Yaw) + sin(Roll)*cos(Yaw))*(TF+TR+TL+TB);
z_craft_component = (1/craft_m)*cos(Roll)*cos(Pitch)*(TF+TR+TL+TB);
Z_ddot = g - z_craft_component;
Roll_ddot = (arm_l/Ixx)*(-TR+TL);
Pitch_ddot = (arm_l/Iyy)*(TF-TB);
Yaw_ddot = (D)/Izz;

%--------FUNCTION OUTPUT-------------

Xout = [X_ddot Y_ddot Z_ddot Roll_ddot Pitch_ddot Yaw_ddot X_dot Y_dot Z_dot Roll_dot
Pitch_dot Yaw_dot];


end
```

A-2 – Code for Attitude Control in MATLAB Simulation

```
function rc_out = rotationControl(rc_in)
%rotationControl PID controller for the attitude of the ANGEL craft
%   The input for the controller should be r_set, p_set, y_set, state,
%   i_error(3), last_pos(3)

%------------VARIABLE ASSIGNMENT-----------------

roll_set = rc_in(1);
pitch_set = rc_in(2);
yaw_set = rc_in(3);
% x_dot = rc_in(4);
% y_dot = rc_in(5);
% z_dot = rc_in(6);
roll_dot = rc_in(7);
pitch_dot = rc_in(8);
yaw_dot = rc_in(9);
% x = rc_in(10);
% y = rc_in(11);
% z = rc_in(12);
roll = rc_in(13);
pitch = rc_in(14);
yaw = rc_in(15);
% roll_i_error = rc_in(16);
% pitch_i_error = rc_in(17);
% yaw_i_error = rc_in(18);
% roll_last_pos = rc_in(19);
% pitch_last_pos = rc_in(20);
% yaw_last_pos = rc_in(21);


%-----------PID Values----------------
roll_p = 0.5; %0.5
```

```
roll_d = 0.175; %0.175

pitch_p = .5;
pitch_d = .175;

yaw_p = 5;
yaw_d = 10;

%-------------CONTROLLER LOOPS-------------

roll_error = roll_set - roll;
pitch_error = pitch_set - pitch;
yaw_error = yaw_set - yaw;

%roll_i_error = roll_i_error + (roll_error*0.02);
%pitch_i_error = pitch_i_error + (pitch_error);
%yaw_i_error = yaw_i_error + (yaw_error*0.02);


% roll_d_error = roll-roll_last_pos;
roll_d_error = roll_dot;
% pitch_d_error = pitch-pitch_last_pos;
pitch_d_error = pitch_dot;
% yaw_d_error = yaw-yaw_last_pos;
yaw_d_error = yaw_dot;


roll_return = (roll_p*roll_error)-(roll_d*roll_d_error);
pitch_return = (pitch_p*pitch_error)-(pitch_d*pitch_d_error);
yaw_return = (yaw_p*yaw_error)-(yaw_d*yaw_d_error);

% roll_return = (roll_p*roll_error)+(roll_i*roll_i_error)+(roll_d*roll_d_error);
%pitch_return = (pitch_p*pitch_error)+(pitch_i*pitch_i_error)+(pitch_d*pitch_d_error)
% yaw_return = (yaw_p*yaw_error)+(yaw_i*yaw_i_error)+(yaw_d*yaw_d_error);

% roll_last_out = roll;  %Stores the current position for use as last position
% pitch_last_out = pitch;
% yaw_last_out = yaw;

rc_out(1) = roll_return;
rc_out(2) = pitch_return;
rc_out(3) = yaw_return;

% rc_out(4) = roll_last_out;
% rc_out(5) = pitch_last_out;
% rc_out(6) = yaw_last_out;
% rc_out(7) = 0;
% rc_out(8) = 0;
% rc_out(9) = 0;
% rc_out(7) = roll_i_error;
% rc_out(8) = pitch_i_error;
% rc_out(9) = yaw_i_error;
end
```

A-3 – Block to translate controller outputs to speed inputs

```
function [out] = controlToSpeed(in)
%controlToSpeed Transforms the controller outputs to the speed inputs
%   Detailed explanation goes here
angel;
% Control inputs
U(1)=craft_m*g;
U(2)=in(2);
U(3)=in(3);
U(4)=in(4);

MM =    [1/(4*b),       0,        1/(2*arm_l*b),    -1/(4*d);
        1/(4*b),    -1/(2*arm_l*b),    0,          1/(4*d);
        1/(4*b),       0,        -1/(2*arm_l*b),   -1/(4*d);
        1/(4*b),     1/(2*arm_l*b),    0,          1/(4*d)];

MA=MM*U';

Omd = sqrt(abs(MA));

% outputs
out(1)=Omd(1); % [dec]
out(2)=Omd(2);
out(3)=Omd(3);
out(4)=Omd(4);

end
```

A-4 – Disabled Altitude Control Block

```
function alt_out = altitudeControl(alt_in)
%altitudeControl PID controller for the attitude of the ANGEL craft
%   The input for the controller should be alt_set, state,
%   i_error(1), last_pos(1)

%------------VARIABLE ASSIGNMENT-----------------

altitude_set = alt_in(1);
x_dot = alt_in(2);
y_dot = alt_in(3);
z_dot = alt_in(4);
roll_dot = alt_in(5);
pitch_dot = alt_in(6);
yaw_dot = alt_in(7);
x = alt_in(8);
y = alt_in(9);
z = alt_in(10);
roll = alt_in(11);
pitch = alt_in(12);
yaw = alt_in(13);
altitude_i_error = alt_in(14);
altitude_last_pos = alt_in(15);
```

```
%-----------PID Values----------------
altitude_p = 3;
altitude_i = 0;
altitude_d = .2;

%-------------CONTROLLER LOOPS-------------

altitude_error = altitude_set - z;

%altitude_i_error = altitude_i_error + (altitude_error*0.02);

altitude_d_error = z_dot;

altitude_return = (altitude_p*altitude_error)-(altitude_d*altitude_d_error);


altitude_last_out = z;  %Stores the current position for use as last position

alt_out(1) = altitude_return;
alt_out(2) = altitude_last_out;
alt_out(3) = 0;


end
```

A-5-1 Arduino Motor Library (QuadMotor.h)

```
/*
  QuadMotor.h - Library for controlling 4 quadrotor motors.
  Created by Michael D. Schmidt.
  Last Updated: 06/21/2010
*/

#ifndef QuadMotor_h
#define QuadMotor_h

#include "WProgram.h"


class QuadMotor
{
  public:
    QuadMotor(int FRONTPIN, int BACKPIN, int RIGHTPIN, int LEFTPIN);
    void initMotors();
      void kill();
      void setEach(int,int,int,int);
      void setAll(int);
      int getCommand(char);
      void pulseMotors(int);

  private:
    int _fpin;
```

```
      int _bpin;
      int _lpin;
      int _rpin;
      int _fcom;
      int _bcom;
      int _rcom;
      int _lcom;
};

#endif
```

A-5-2 – Arduino Motor Library (QuadMotor.cpp)

```
/****************************************
* INCLUDES
****************************************/
#include "WProgram.h"
#include "QuadMotor.h"

/****************************************
* CONSTRUCTORS
****************************************/
QuadMotor::QuadMotor(int FRONTPIN, int BACKPIN, int RIGHTPIN, int
LEFTPIN)
{
      _fpin = FRONTPIN;
      _bpin = BACKPIN;
      _lpin = LEFTPIN;
      _rpin = RIGHTPIN;
      _fcom = 0;
      _bcom = 0;
      _rcom = 0;
      _lcom = 0;
}

/****************************************
* METHODS
****************************************/


void QuadMotor::initMotors()
{
    pinMode(_fpin, OUTPUT);
    analogWrite(_fpin, 124);
      _fcom = 124;
    pinMode(_bpin, OUTPUT);
    analogWrite(_bpin, 124);
      _bcom = 124;
    pinMode(_rpin, OUTPUT);
    analogWrite(_rpin, 124);
      _rcom = 124;
    pinMode(_lpin, OUTPUT);
    analogWrite(_lpin, 124);
      _lcom = 124;
}

void QuadMotor::kill()
```

```
{
      analogWrite(_fpin, 124);
      _fcom = 124;
      analogWrite(_bpin, 124);
      _bcom = 124;
      analogWrite(_lpin, 124);
      _lcom = 124;
      analogWrite(_rpin, 124);
      _rcom = 124;
}

int QuadMotor::getCommand(char motor)
{
      if(motor == 'f' || motor == 'F'){
            return _fcom;
      }
      if(motor == 'b' || motor == 'B'){
            return _bcom;
      }
      if(motor == 'r' || motor == 'R'){
            return _rcom;
      }
      if(motor == 'l' || motor == 'L'){
            return _lcom;
      }
}

void QuadMotor::setEach(int f, int b, int r, int l)
{
      analogWrite(_fpin, f);
      _fcom = f;
      analogWrite(_bpin, b);
      _bcom = b;
      analogWrite(_lpin, l);
      _lcom = l;
      analogWrite(_rpin, r);
      _rcom = r;
}

void QuadMotor::setAll(int com)
{
      analogWrite(_fpin, com);
      _fcom = com;
      analogWrite(_bpin, com);
      _bcom = com;
      analogWrite(_lpin, com);
      _lcom = com;
      analogWrite(_rpin, com);
      _rcom = com;
}

void QuadMotor::pulseMotors(int q)
{
    for (int i = 0; i < q; i++) {
      setAll(165);
      delay(250);
      setAll(124);
```

```
      delay(250);
    }
}
```

A-6-1 – Arduino PID Library (SchmidPID.h)

```
/*
  SchmidtPID.h - Library for PID control.
  Created by Michael D. Schmidt.
  Last Updated: 06/22/2010
  Portions of this library were modified from Ted Carancho's AeroQuad
PID Controller (www.AeroQuad.com)
  and from
http://www.arduino.cc/playground/Main/BarebonesPIDForEspresso
*/

#ifndef SchmidtPID_h
#define SchmidtPID_h

#include "WProgram.h"


class SchmidtPID
{
  public:
    SchmidtPID(float P, float I, float D, float windup);
      float updatePID(float,float);
      void setValues(int,int,int);
      void zeroError();
      float getP();
      float getI();
      float getD();


  private:
  float _last;
  float _p;
  float _i;
  float _d;
  float _iError;
  float _guard;



};

#endif
```

A-6-2 – Arduino PID Library (SchmidtPID.cpp)

```
/**************************************
* INCLUDES
***************************************/
```

```
#include "WProgram.h"
#include "SchmidtPID.h"

/****************************************
* CONSTRUCTORS
****************************************/
SchmidtPID::SchmidtPID(float P, float I, float D, float windup)
{
	_p = P;
	_i = I;
	_d = D;
	_guard = windup;
	_last = 0;
	_iError = 0;
}

/****************************************
* METHODS
****************************************/

float SchmidtPID::updatePID(float target, float current)
{
	float instant_error;
	float dTerm;

	instant_error = target - current; //instant error between target
and current
	_iError += instant_error; //accumulated Error since PID creation
	if(_iError < -_guard){
		_iError = -_guard;
	}
	else if(_iError > _guard){
		_iError = _guard;
	}

	dTerm = _d * (current - _last);
	_last = current;
	return (_p * instant_error) + (_i * _iError) + dTerm;
}

void SchmidtPID::setValues(int P, int I, int D)
{
	_p = P;
	_i = I;
	_d = D;
}

void SchmidtPID::zeroError()
{
	_iError = 0;
}

float SchmidtPID::getP()
{
	return _p;
}
```

```
float SchmidtPID::getI()
{
      return _i;
}

float SchmidtPID::getD()
{
      return _d;
}
```

A-7-1 – Arduino IMU Sensor Library (IMU.h)

```
/*
  IMU.h - Library for reading IMU sensors.
  Created by Michael D. Schmidt.
  Last Updated: 03/31/10
*/

#ifndef IMU_h
#define IMU_h

#include "WProgram.h"
#include <math.h>


class IMU
{
  public:
    IMU(int XACCPIN, int YACCPIN, int ZACCPIN, int YRATEPIN, int
XRATEPIN, int ZRATEPIN, float vref, float vs, float gyroW);
    float getXAccel();
      float getYAccel();
      float getZAccel();
      float getRateAX();
      float getRateAY();
      float getRateAZ();
      void  zeroGyros();
      void  zeroAccels();
      float angleRad(char);
      float angleDeg(char);
      float estimate(char);
  private:
    int _xpin;
      int _ypin;
      int _zpin;
      int _yrpin;
      int _xrpin;
      int _zrpin;
      float _vref;
      float _vsup;
    float _zerog_x;
      float _zerog_y;
      float _zerog_z;
      float _accelsens;
      float _gyrosens;
      float _gyrozero_x;
      float _gyrozero_y;
```

```
	float _gyrozero_z;
	unsigned long _previousTime;
	char _firstSample;
	float _RxEst;
	float _RyEst;
	float _RzEst;
	float _RxGyro;
	float _RyGyro;
	float _RzGyro;
	float _gyroW;

};

#endif
```

A-7-2 – Arduino IMU Sensor Library (IMU.cpp)

```
/****************************************
* INCLUDES
****************************************/
#include "WProgram.h"
#include <math.h>
#include "IMU.h"

/****************************************
* CONSTRUCTORS
****************************************/
IMU::IMU(int XACCPIN, int YACCPIN, int ZACCPIN, int YRATEPIN, int
XRATEPIN, int ZRATEPIN, float vref, float vs, float gyroW)
{
	_xpin = XACCPIN;
	_ypin = YACCPIN;
	_zpin = ZACCPIN;
	_yrpin = YRATEPIN;
	_xrpin = XRATEPIN;
	_zrpin = ZRATEPIN;
	_vref = vref;
	_vsup = vs;
	_accelsens = _vsup*0.1;
	_gyrosens = 0.002;
	_gyrozero_x = 1.35;
	_gyrozero_y = 1.35;
	_gyrozero_z = 1.35;
	_zerog_x = _vsup/2;
	_zerog_y = _vsup/2;
	_zerog_z = _vsup/2;
	_previousTime = 0;
	_firstSample = 1;
	_gyroW = gyroW;
}

/****************************************
* METHODS
****************************************/

float IMU::getXAccel()
{
```

```
	int xa = analogRead(_xpin);
	float Rx = (((xa*_vref)/(1023))-_zerog_x)/_accelsens;
	return Rx;
}

float IMU::getYAccel()
{
	int ya = analogRead(_ypin);
	float Ry = (((ya*_vref)/(1023))-_zerog_y)/_accelsens;
	return Ry;
}

float IMU::getZAccel()
{
	int za = analogRead(_zpin);
	float Rz = (((za*_vref)/(1023))-_zerog_z)/_accelsens;
	return Rz;
}

float IMU::getRateAX()
{
	int axz = analogRead(_xrpin);
	float Raxz = (((axz*_vref)/1023)-_gyrozero_x)/_gyrosens;
	return Raxz;
}

float IMU::getRateAY()
{
	int ayz = analogRead(_yrpin);
	float Rayz = (((ayz*_vref)/1023)-_gyrozero_y)/_gyrosens;
	return Rayz;
}

float IMU::getRateAZ()
{
	int azz = analogRead(_zrpin);
	float Razz = ((((azz*_vref)/1023)-_gyrozero_z)/_gyrosens);
	return Razz;
}

void IMU::zeroGyros()
{
	delay(100);
	int rot_x = analogRead(_xrpin);
	int rot_y = analogRead(_yrpin);
	int rot_z = analogRead(_zrpin);
	float rvalue_x = ((rot_x*_vref)/1023)-(0*_gyrosens);
	float rvalue_y = ((rot_y*_vref)/1023)-(0*_gyrosens);
	float rvalue_z = ((rot_z*_vref)/1023)-(0*_gyrosens);
	_gyrozero_x = rvalue_x;
	_gyrozero_y = rvalue_y;
	_gyrozero_z = rvalue_z;
}

void IMU::zeroAccels()
{
	delay(100);
```

```
	int grav_x = analogRead(_xpin);
	int grav_y = analogRead(_ypin);
	int grav_z = analogRead(_zpin);
	float value_x = ((grav_x*_vref)/1023)-(0*_accelsens);
	float value_y = ((grav_y*_vref)/1023)-(0*_accelsens);
	float value_z = ((grav_z*_vref)/1023)-(1*_accelsens);
	_zerog_x = value_x;
	_zerog_y = value_y;
	_zerog_z = value_z;
}

float IMU::angleRad(char axis)
{
	float Ax = getXAccel();
	float Ay = getYAccel();
	float Az = getZAccel();
	if (axis == 'x') return atan2(Ax, sqrt(Ay * Ay + Az * Az));
	if (axis == 'y') return atan2(Ay, sqrt(Ax * Ax + Az * Az));
}

float IMU::angleDeg(char axis)
{
	return degrees(angleRad(axis));
}

float IMU::estimate(char axis)
{
	float Axz;
	float Ayz;
	char signRzGyro;
	unsigned long currentTime = millis();
	float RxAcc = getXAccel(); //pull raw data
	float RyAcc = getYAccel();
	float RzAcc = getZAccel();


	unsigned long deltaT = currentTime - _previousTime;
	_previousTime = currentTime;

	float RaccAbs = sqrt(RxAcc*RxAcc + RyAcc*RyAcc + RzAcc*RzAcc);
//find vector length
	RxAcc /= RaccAbs;
	RyAcc /= RaccAbs;
	RzAcc /= RaccAbs;


	//If this is the first time through the loop, let former
estimated angles be the Accel angels
	if(_firstSample){
		_RxEst = RxAcc;
		_RyEst = RyAcc;
		_RzEst = RzAcc;
	}
	else{
		if(abs(_RzEst) < 0.1){
			_RxGyro = _RxEst;
		_RyGyro = _RyEst;
```

```
            _RzGyro = _RzEst;
            }
            else{
                  float gyroX = getRateAX();   //get gyro data in deg/s
                  float gyroY = getRateAY();   //get gyro data in deg/s
            gyroX *= deltaT / 1000.0f;     //get angle change in deg
            gyroY *= deltaT / 1000.0f;     //get angle change in deg
            Axz = atan2(_RxEst,_RzEst) * 180 / 3.14159265358979f;
//get angle and convert to degrees
            Ayz = atan2(_RyEst,_RzEst) * 180 / 3.14159265358979f;
            Axz += gyroX;                   //get updated angle
according to gyro movement
            Ayz += gyroY;                   //get updated angle
according to gyro movement
            //estimate sign of RzGyro by looking in what qudrant the
angle Axz is,
                  //RzGyro is pozitive if  Axz in range -90 ..90 =>
cos(Awz) >= 0
                  signRzGyro = ( cos(Axz * 3.14159265358979f / 180) >=0
) ? 1 : -1;
                  _RxGyro = sin(Axz * 3.14159265358979f / 180);
            _RxGyro /= sqrt( 1 + (cos(Axz * 3.14159265358979f /
180))*(cos(Axz * 3.14159265358979f / 180)) * (tan(Ayz *
3.14159265358979f / 180))*(tan(Ayz * 3.14159265358979f / 180)) );
            _RyGyro = sin(Ayz * 3.14159265358979f / 180);
            _RyGyro /= sqrt( 1 + (cos(Ayz * 3.14159265358979f /
180))*(cos(Ayz * 3.14159265358979f / 180)) * (tan(Axz *
3.14159265358979f / 180))*(tan(Axz * 3.14159265358979f / 180)) );
            _RzGyro = signRzGyro * sqrt(1 - (_RxGyro*_RxGyro) -
(_RyGyro*_RyGyro));
            }
             _RxEst = (RxAcc + _gyroW * _RxGyro) / (1 + _gyroW);
             _RyEst = (RyAcc + _gyroW * _RyGyro) / (1 + _gyroW);
             _RzEst = (RzAcc + _gyroW * _RzGyro) / (1 + _gyroW);
             float R = sqrt(_RxEst*_RxEst + _RyEst*_RyEst +
_RzEst*_RzEst);
             _RxEst /= R;
             _RyEst /= R;
             _RzEst /= R;
    }
      _firstSample = 0;
    if (axis == 'x') return _RxEst;
      if (axis == 'y') return _RyEst;
      if (axis == 'z') return _RzEst;
}
```

A-7-3 – Arduino IMU Sensor Library Example (Processing)

```
/*
IMU Library Example
Created by: Michael D. Schmidt
Description: This IMU Library example file demonstrates the main
functionalities of the IMU Library. These functions include:
    getXAccel()
    getYAccel()
```

```
    getZAccel()
    getRateAX()
    getRateAY()
    getRateAZ()
    zeroGyros()
    zeroAccels()
    angleRad()
    angleDeg()

Last Updated: 06/18/2010 14:50
*/


#include <IMU.h>


/*Create IMU instance,
 XACCPIN = 0 - Pin for X axis acceleration
 YACCPIN = 1 - Pin for Y axis acceleration
 ZACCPIN = 2 - Pin for Z axis acceleration
 YRATEPIN = 3 - Pin for Y axis Gyro
 XRATEPIN = 4 - Pin for X axis Gyro
 ZRATEPIN = 5 - Pin for Z axis Gyro
 VREF = 5V  - Reference voltage (this is the ADC voltage)
 VS = 3.3V - Sensor Supply Voltage (this is the voltage the sensor
uses)
 */

IMU IMU(0,1,2,3,4,5,5,3.3); //create a new IMU instance

float pitch;
float roll;
float x;
float y;
float z;
float Rx;
float Ry;
float Rz;
int incomingByte;

void setup(){

  IMU.zeroGyros(); //Zero the Gyros
  IMU.zeroAccels(); //Zero the Accelerometers
  Serial.begin(9600);
  Serial.println("Initialized...");
}

void loop(){
  //Press Z during testing to zero all sensors
  if(Serial.available() > 0) {
    incomingByte = Serial.read();
    if(incomingByte == 'Z'){
      IMU.zeroGyros();
      IMU.zeroAccels();
      Serial.println("");
      Serial.println("SENSORS ZEROED");
```

```
    }
  }
  // getXAccel() reads the X acceleration value (will be between -1 and
1)
  // similar functions for the Y and Z axes
  x = IMU.getXAccel();
  y = IMU.getYAccel();
  z = IMU.getZAccel();

  // getRateAX() provides the deg/sec rotation rate around the X axis.
Limits depend on sensor.
  // similar functions for the Y and Z axes
  Rx = IMU.getRateAX();
  Ry = IMU.getRateAY();
  Rz = IMU.getRateAZ();
  // angleDeg('x') reads the angle of the x axis wrt ground (will be
between -90 and 90 degrees)
  // angleRad('x') also exists and allows for reading the value in as a
radian value
  // similar functions exist for Y and Z axes
  pitch = IMU.angleDeg('x');
  roll = IMU.angleDeg('y');
  Serial.print(pitch);
  Serial.print("\t");
  Serial.println(roll);
  delay(100);
}
```

A-8 – Arduino Main Avionics Loop

```
//Library Includes
#include <SchmidtPID.h>
#include <IMU.h>
#include <QuadMotor.h>

//Constant Definitions:
const int FRONTPIN = 3;
const int BACKPIN = 9;
const int RIGHTPIN = 10;
const int LEFTPIN = 11;
const int LEDPIN = 13;


//Variable Definitions:
float pitch;
float roll;
int incomingByte;
byte buffer[3];
float r_value;
float p_value;
int throttle = 150;
int i;
int L_command;
int R_command;
int F_command;
int B_command;
boolean safety = true;
boolean debug = false;
```

```
boolean start = false;
int roll_target = 0;
int pitch_target = 0;
float roll_p = 4; //default roll P value
float roll_i = 0;
float roll_d = -10;
float pitch_p = 4;
float pitch_i = 0;
float pitch_d = -10;

//Instance Creation:
IMU sensors(0,1,2,3,4,5,5,3.3,10); //create a new IMU instance
QuadMotor motors(FRONTPIN,BACKPIN,RIGHTPIN,LEFTPIN); //create a new
motor control instance
SchmidtPID rollPID(roll_p,roll_i,roll_d,1000);
SchmidtPID pitchPID(pitch_p,pitch_i,pitch_d,1000);

//Setup Loop
void setup(){
  sensors.zeroGyros(); //Zero the Gyros
  sensors.zeroAccels(); //Zero the Accelerometers
  motors.initMotors();
  Serial.begin(115200);
  delay(2000);
  Serial.print("Initialized...");
  Serial.print("\n");
  digitalWrite(LEDPIN, HIGH);
}

void loop(){
  //Press Z during testing to zero all sensors
  if(Serial.available() > 0) {
    incomingByte = Serial.read();
    //If Incoming = 126 (~), this means a throttle value of three
digits has been sent
    if(incomingByte == 126){
      //The following code is for using a 3 digit number, which is
currently buggy
      /*
      i = 0;
      Serial.flush();
      delay(10);
      while(Serial.available() > 0){
        buffer[i] = Serial.read();
        i++;
      }
      //throttle = (buffer[0]-48)*100 + (buffer[1]-48)*10 + (buffer[2]-
48);
      throttle = (buffer[0]*100 + buffer[1]*10 + buffer[2]);
      if(throttle <= 124){
        throttle = 124;
      }
      if(throttle >= 250){
        throttle = 250;
      }
      */
      throttle += 1;
```

```
      if(throttle <= 124){
          throttle = 124;
      }
      if(throttle >= 250){
          throttle = 250;
      }
      Serial.print("Throttle Duty Cycle: ");
      Serial.print(throttle);
      Serial.print("\n");
    }

    if(incomingByte == 36){
      throttle -= 1;
      if(throttle <= 124){
          throttle = 124;
      }
      if(throttle >= 250){
          throttle = 250;
      }
      Serial.print("Throttle Duty Cycle: ");
      Serial.print(throttle);
      Serial.print("\n");
    }


    //If Incoming = 33 (!), this means a Roll P value of two digits has
been sent
    if(incomingByte == 33){
      i = 0;
      Serial.flush();
      delay(10);
      while(Serial.available() > 0){
        buffer[i] = Serial.read();
        i++;
      }
      //throttle = (buffer[0]-48)*100 + (buffer[1]-48)*10 + (buffer[2]-
48);
      roll_p = (buffer[0]*10 + buffer[1]);
      if(roll_p <= 1){
        roll_p = 1;
      }
      if(roll_p >= 25){
        roll_p = 25;
      }
      rollPID.setValues(roll_p,roll_i,roll_d);
      Serial.print("Roll Axis [P]: ");
      Serial.print(roll_p);
      Serial.print("\n");
    }

    if(incomingByte == 'Z'){
      sensors.zeroGyros();
      sensors.zeroAccels();
      rollPID.zeroError();
      pitchPID.zeroError();
      Serial.print("SENSORS ZEROED");
      Serial.print("\n");
```

```
 }
if(incomingByte == 'I'){
   if(safety){
     Serial.print("Unable to Start, SAFETY ENABLED!");
     Serial.print("\n");
   }
   else if(!safety){
     Serial.print("STARTING, MOTORS ARMED!");
     Serial.print("\n");
     start = true;
   }

 }

 if(incomingByte == 'S'){
  if(safety){
    Serial.print("WARNING!! SAFETY OFF!");
    Serial.print("\n");
  }
  else{
    Serial.print("SAFETY ENABLED");
    Serial.print("\n");
  }
  safety = !safety;
 }
 if(incomingByte == 'X'){
   Serial.print("KILL");
   Serial.print("\n");
   motors.kill();
   safety = true;
   start = false;
 }
 if(incomingByte == 'G'){
   Serial.print("MOTOR VALUES:- ");
   Serial.print("F:");
   Serial.print(motors.getCommand('f'));
   Serial.print(",");
   Serial.print("B:");
   Serial.print(motors.getCommand('b'));
   Serial.print(",");
   Serial.print("R:");
   Serial.print(motors.getCommand('r'));
   Serial.print(",");
   Serial.print("L:");
   Serial.print(motors.getCommand('l'));
   Serial.print("\n");
  }
if(incomingByte == 'P'){
  if(safety){
    Serial.print("Unable to Pulse, SAFETY IS ENABLED");
    Serial.print("\n");
  }
  else{
    Serial.print("MOTOR PULSE");
    Serial.print("\n");
    motors.pulseMotors(3);
  }
```

```
}
if(incomingByte == 'D'){
  if(debug){
    Serial.print("EXITING DEBUG MODE...");
    Serial.print("\n");
  }
  else{
  Serial.print("ENTERING DEBUG MODE...");
  Serial.print("\n");
  }
  debug = !debug;
}
if(incomingByte == 'V'){
  Serial.print("ANGEL INITIALIZED AND READY FOR FLIGHT");
  Serial.print("\n");
  /*
  i = 0;
  Serial.flush();
  delay(10);
  while(Serial.available() > 0){
    buffer[i] = Serial.read();
    i++;
  }
  throttle = (buffer[0]*100 + buffer[1]*10 + buffer[2]);
  roll_p = (buffer[4]*10 + buffer[5]);
  roll_i = (buffer[7]*10 + buffer[8]);
  roll_d = (buffer[10]*10 + buffer[11]);
  pitch_p = (buffer[13]*10 + buffer[14]);
  pitch_i = (buffer[16]*10 + buffer[17]);
  pitch_d = (buffer[19]*10 + buffer[20]);
  if(buffer[3] == 0) {roll_p *= -1;}
  if(buffer[6] == 0) {roll_i *= -1;}
  if(buffer[9] == 0) {roll_d *= -1;}
  if(buffer[12] == 0) {pitch_p *= -1;}
  if(buffer[15] == 0) {pitch_i *= -1;}
  if(buffer[18] == 0) {pitch_d *= -1;}

  Serial.print("Values Initialized to:");
  Serial.print("Throttle Value:");
  Serial.print(throttle);
  Serial.print("\n");
  Serial.print("Roll PID:");
  Serial.print(roll_p);
  Serial.print(",");
  Serial.print(roll_i);
  Serial.print(",");
  Serial.print(roll_d);
  Serial.print("\n");
  Serial.print("Pitch PID:");
  Serial.print(pitch_p);
  Serial.print(",");
  Serial.print(pitch_i);
  Serial.print(",");
  Serial.print(pitch_d);
  Serial.print("\n");
  */
}
```

```
if(incomingByte == 'T'){
  roll_target -= 1;
  Serial.print("Roll Target set to: ");
  Serial.print(roll_target);
  Serial.print("\n");
}
if(incomingByte == 'Y'){
  roll_target += 1;
  Serial.print("Roll Target set to: ");
  Serial.print(roll_target);
  Serial.print("\n");
}

if(incomingByte == 42){
  pitch_target -= 1;
  Serial.print("Pitch Target set to: ");
  Serial.print(pitch_target);
  Serial.print("\n");
}
if(incomingByte == 43){
  pitch_target += 1;
  Serial.print("Pitch Target set to: ");
  Serial.print(pitch_target);
  Serial.print("\n");
}
if(incomingByte == 44){
  roll_p += 0.1;
  rollPID.setValues(roll_p,roll_i,roll_d);
  Serial.print("Roll P Increased to: ");
  Serial.print(roll_p);
  Serial.print("\n");
}
if(incomingByte == 45) {
  roll_p -= 0.1;
  rollPID.setValues(roll_p,roll_i,roll_d);
  Serial.print("Roll P Decreased to: ");
  Serial.print(roll_p);
  Serial.print("\n");
}
if(incomingByte == 46){
  roll_d += 0.1;
  rollPID.setValues(roll_p,roll_i,roll_d);
  Serial.print("Roll D Increased to: ");
  Serial.print(roll_d);
  Serial.print("\n");
}
if(incomingByte == 47) {
  roll_d -= 0.1;
  rollPID.setValues(roll_p,roll_i,roll_d);
  Serial.print("Roll D Decreased to: ");
  Serial.print(roll_d);
  Serial.print("\n");
}
if(incomingByte == 48){
  pitch_p += 0.1;
  pitchPID.setValues(pitch_p,pitch_i,pitch_d);
  Serial.print("Pitch P Increased to: ");
```

```
    Serial.print(pitch_p);
    Serial.print("\n");
  }
  if(incomingByte == 49) {
    pitch_p -= 0.1;
    pitchPID.setValues(pitch_p,pitch_i,pitch_d);
    Serial.print("Pitch P Decreased to: ");
    Serial.print(pitch_p);
    Serial.print("\n");
  }

  if(incomingByte == 50){
    pitch_d += 0.1;
    pitchPID.setValues(pitch_p,pitch_i,pitch_d);
    Serial.print("Pitch D Increased to: ");
    Serial.print(pitch_d);
    Serial.print("\n");
  }
  if(incomingByte == 51) {
    pitch_d -= 0.1;
    pitchPID.setValues(pitch_p,pitch_i,pitch_d);
    Serial.print("Pitch D Decreased to: ");
    Serial.print(pitch_d);
    Serial.print("\n");
  }

}
if(start){
roll = sensors.angleDeg('y');
pitch = sensors.angleDeg('x');
r_value = rollPID.updatePID(roll_target, roll);
p_value = pitchPID.updatePID(pitch_target, pitch);
r_value /= 10;
p_value /= 10;
//trying to switch values
L_command = int(throttle + r_value);
R_command = int(throttle - r_value);
F_command = int(throttle - p_value);
B_command = int(throttle + p_value);
if(L_command > 253) L_command = 253;
if(L_command < 160) L_command = 160;
if(R_command > 253) R_command = 253;
if(R_command < 160) R_command = 160;
if(F_command > 253) F_command = 253;
if(F_command < 160) F_command = 160;
if(B_command > 253) B_command = 253;
if(B_command < 160) B_command = 160;
if(!safety){
  motors.setEach(F_command,B_command,R_command,L_command);
}
if(debug){
  Serial.print(F_command);
  Serial.print(",");
  Serial.print(B_command);
  Serial.print(",");
  Serial.print(pitch);
  Serial.print(",");
```

```
    Serial.print(R_command);
    Serial.print(",");
    Serial.print(L_command);
    Serial.print(",");
    Serial.print(roll);
    Serial.print(",");
    delay(100);
    }
  }
}
```

A-9 – Processing Controller Code

```
import processing.serial.*; // serial library
Serial[] myPorts = new Serial[1]; // lets only use one port in this
sketch

// GUI variables
import controlP5.*; // controlP5 library
ControlP5 controlP5; // create the handler to allow for controlP5 items
Textlabel txtlblWhichcom; // text label displaying which comm port is
being used
Textlabel l_debug;
Textlabel r_debug;
Textlabel l_safety;
Textlabel r_safety;
ListBox commListbox; // list of available comm ports
ListBox commList;
Textfield pr_field;
Textfield ir_field;
Textfield dr_field;

int corner_x = 5;
int corner_y = 160; //130
int button_w = 90;
int button_h = 20;
int throt = 150; //default throttle value
int p_r_val = 5;
int i_r_val = 0;
int d_r_val = -10;
int[] arr = new int[3];
int[] dual_arr = new int[2];
int holder;
int i = 0;
int h = hour();
int m = minute();
int s = second();
PImage logo;
boolean fast_climb = false;

// setup
void setup() {
  size(800,600);
  frameRate(30);

  controlP5 = new ControlP5(this); // initialize the GUI controls
```

```
  println(Serial.list()); // print the comm ports to the debug window
for debugging purposes

  // make a listbox and populate it with the available comm ports
  commListbox = controlP5.addListBox("myList",width-180-10,30,180,120);
//addListBox(name,x,y,width,height)
  commListbox.captionLabel().toUpperCase(false);
  commListbox.captionLabel().set("COM Ports");
  commListbox.close();
  for(int i=0;i<Serial.list().length;i++) {
    commListbox.addItem("port: "+Serial.list()[i],i); //
addItem(name,value)
  }

  // text label for which comm port selected
  txtlblWhichcom = controlP5.addTextlabel("txtlblWhichcom","No Port
Selected",width-180-30,10); // textlabel(name,text,x,y)
  l_debug = controlP5.addTextlabel("l_debug","Debug Mode:
",corner_x,corner_y-25);
  r_debug = controlP5.addTextlabel("r_debug","OFF",corner_x +
99,corner_y + 12-28);
  l_safety = controlP5.addTextlabel("l_safety","SAFETY:
",corner_x,corner_y + 30 - 25);
  r_safety = controlP5.addTextlabel("r_safety","ON",corner_x +
103,corner_y + 30 + 12-29);

  //set label colors
  l_safety.setColorValue(0x13eb1c);

commList = controlP5.addListBox("commList",corner_x,corner_y +
250,400,150);
commList.setItemHeight(15);
commList.setBarHeight(20);
commList.captionLabel().toUpperCase(false);
commList.captionLabel().set("COMMUNICATIONS LOG");
addToLog("CONTROL: Welcome to ANGEL Controller. Please select a COM
port.");
commList.scroll(1);
i += 1;
commList.setColorBackground(color(71,71,71));
commList.setColorActive(color(128,133,72));

  // a button to send the letter a
  controlP5.addButton("INITIALIZE",6,corner_x,corner_y +
30,button_w,button_h);
  controlP5.addButton("MOTOR_PULSE",1,corner_x,corner_y +
60,button_w,button_h); // buton(name,value,x,y,width,height)
  controlP5.addToggle("DEBUG",false,corner_x + 70,corner_y - 30,10,10);
  controlP5.addToggle("SAFETY",true,corner_x + 70,corner_y - 20 + 30-
10,10,10);
  controlP5.addButton("ZERO_SENSORS",2,corner_x,corner_y +
90,button_w,button_h); // buton(name,value,x,y,width,height)
  controlP5.addButton("GET_MOTOR_VALUES",5,corner_x,corner_y +
120,button_w,button_h);
  controlP5.addButton("START",3,corner_x,corner_y +
150,button_w,button_h);
```

```
  controlP5.addButton("KILL",4,corner_x,corner_y +
180,button_w,button_h);
  logo = loadImage("logo.jpg");

  pr_field = controlP5.addTextfield("Roll_P",corner_x + 140,corner_y +
70,38,20);
  //p_field.setFocus(true);
  pr_field.setAutoClear(false);
  pr_field.setText("5");
  controlP5.addButton("DEC1",6,corner_x + 110,corner_y + 70,25,20);
  controlP5.addButton("INC1",6,corner_x + 190,corner_y + 70,25,20);

  ir_field = controlP5.addTextfield("Roll_I",corner_x + 140,corner_y +
110,38,20);
  //p_field.setFocus(true);
  ir_field.setAutoClear(false);
  ir_field.setText("0");
  controlP5.addButton("DEC2",6,corner_x + 110,corner_y + 110,25,20);
  controlP5.addButton("INC2",6,corner_x + 190,corner_y + 110,25,20);

  dr_field = controlP5.addTextfield("Roll_D",corner_x + 140,corner_y +
150,38,20);
  //p_field.setFocus(true);
  dr_field.setAutoClear(false);
  dr_field.setText("-10");
  controlP5.addButton("DEC3",6,corner_x + 110,corner_y + 150,25,20);
  controlP5.addButton("INC3",6,corner_x + 190,corner_y + 150,25,20);
}

// infinite loop
void draw() {
  background(95);
  image(logo,5,5);
}

// print the name of the control being triggered (for debugging) and
see if it was a Listbox event
public void controlEvent(ControlEvent theEvent) {
  // ListBox is if type ControlGroup, you need to check the Event with
if (theEvent.isGroup())to avoid an error message from controlP5
  if (theEvent.isGroup()) {
    // an event from a group
    if (theEvent.group().name()=="myList") {
      InitSerial(theEvent.group().value()); // initialize the serial
port selected
      //println("got myList"+"   value = "+theEvent.group().value());
// for debugging
    }
  }
  else {
    //println(theEvent.controller().name()); // for debugging
  }
}


// run this when buttonA is triggered, send an a
public void MOTOR_PULSE(int theValue) {
```

```
  addToLog("USER: Request Motor Pulse");
  myPorts[0].write('P');

}

// initialize the serial port selected in the listBox
void InitSerial(float portValue) {
  println("initializing serial " + int(portValue) + " in
serial.list()"); // for debugging

  String portPos = Serial.list()[int(portValue)]; // grab the name of
the serial port

  txtlblWhichcom.setValue("COM Initialized = " + portPos);
  addToLog("CONTROL: COM SELECTED - " + portPos);

  myPorts[0] = new Serial(this, portPos, 115200); // initialize the
port

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  //myPorts[0].bufferUntil('\n');
  //println("done init serial");
}

// serial event, check which port generated the event
// just in case there are more than 1 ports open
void serialEvent(Serial thisPort) {

  // read the serial buffer until a newline appears
  String myString = thisPort.readStringUntil(10);
  //myString = trim(myString); // ditch the newline
  if(myString != null){
  addToLog("ANGEL: " + myString); // print to debug window
  /*
  String[] match1 = match(myString,"999");
  if(match1 != null){
    int values[] = int(split(myString,','));
    p_r_val = values[2];
    println(p_r_val);
    p_field.setText(str(p_r_val));
  }
  */
  }



  // uncomment the following if you are getting streaming packets of
data that need to be parsed
/*
  // if you got any bytes other than the newline
  if (myString != null) {

    //myString = trim(myString); // ditch the newline

    // split the string at the spaces, save as integers
    int sensors[] = int(split(myString, ' '));
```

```
    // convert to x and y or whatever
    if ((sensors.length == 2)&&(portNumber==0)) { // hardcoded
portNumber==0 because only using one port in this sketch
      //float x = sensors[0]/100.0; // whatever conversion you need to
do
      //float y = sensors[1]/100.0;
    }
  }
*/
} // end serialEvent

void DEBUG(boolean theFlag) {
  if(theFlag==true) {
    println("DEBUG ON");
    addToLog("USER: Debug ON");
    r_debug.setValue("ON");

  } else {
    println("DEBUG OFF");
    addToLog("USER: Debug OFF");
    r_debug.setValue("OFF");
  }
  myPorts[0].write('D');
}

void SAFETY(boolean theFlag) {
  if(theFlag==true) {
    println("SAFETY ON");
    addToLog("USER: Safety ON");
    r_safety.setValue("ON");
    l_safety.setColorValue(0x13eb1c);

  } else {
    println("SAFETY OFF");
    addToLog("USER: Safety OFF");

    r_safety.setValue("OFF");
    l_safety.setColorValue(0xff0006);
  }
  myPorts[0].write('S');
}

void addToLog(String val){
  h = hour();
  m = minute();
  s = second();
  commList.addItem("("+h+":"+m+":"+s+")  "+ val,i);
  commList.scroll(1);
  i += 1;
}

public void ZERO_SENSORS(int theValue) {
  addToLog("USER: Request sensors be zeroed");
  myPorts[0].write('Z');
}
public void START(int theValue) {
  addToLog("USER: Request START");
```

```
  myPorts[0].write('I');
}
public void KILL(int theValue) {
  addToLog("USER: Request KILL");
  SAFETY(true);
  myPorts[0].write('X');
}
public void GET_MOTOR_VALUES(int theValue) {
  addToLog("USER: Request Motor Command Values");
  myPorts[0].write('G');
}
/*
public void P(String theText) {
 if(int(theText) < 1){
   p_r_val = 1;
   addToLog("CONTROL: MIN  Roll P = 1");
 }
 if(int(theText) > 25){
   p_r_val = 25;
   addToLog("CONTROL: Roll P = 25");
 }
 if(int(theText) >= 1 && int(theText) <= 25){
   throt = int(theText);
   addToLog("USER: Set Roll P to " + p_r_val);
 }
 sendCValue(p_r_val,'a');
 }
 */

/*
public void DEC1(int theValue){
  p_r_val -= 1;
  if(p_r_val <= 1){
     addToLog("CONTROL: MIN P for roll = 1");
     p_r_val = 1;
  }
  pr_field.setText(Integer.toString(p_r_val));
  addToLog("USER: Decrease Roll P Value to " + p_r_val);
  sendCValue(p_r_val,'a');
}

public void INC1(int theValue){
  p_r_val += 1;
  if(p_r_val >=25){
     addToLog("CONTROL: MAX P for roll = 25");
     p_r_val = 25;
  }
  pr_field.setText(Integer.toString(p_r_val));
  addToLog("USER: Increasae Roll P Value to " + p_r_val);
  sendCValue(p_r_val,'a');
}


public void DEC2(int theValue){
  i_r_val -= 1;
  if(i_r_val <= 0){
     addToLog("CONTROL: MIN I for roll = 0");
```

```
      i_r_val = 0;
   }
   ir_field.setText(Integer.toString(i_r_val));
   addToLog("USER: Decrease Roll I Value to " + i_r_val);
   sendCValue(i_r_val,'b');
}

public void INC2(int theValue){
   i_r_val += 1;
   if(i_r_val >=12){
      addToLog("CONTROL: MAX I for roll = 12");
      p_r_val = 12;
   }
   ir_field.setText(Integer.toString(i_r_val));
   addToLog("USER: Increasae Roll I Value to " + i_r_val);
   sendCValue(i_r_val,'b');
}

public void DEC3(int theValue){
   d_r_val -= 1;
   if(d_r_val == 0){
      addToLog("CONTROL: D != 0");
      d_r_val += 1;
   }
   dr_field.setText(Integer.toString(d_r_val));
   addToLog("USER: Decrease Roll D Value to " + d_r_val);
   sendCValue(d_r_val,'c');
}

public void INC3(int theValue){
   d_r_val += 1;
   if(d_r_val == 0){
      addToLog("CONTROL: D != 0");
      d_r_val -= 1;
   }
   dr_field.setText(Integer.toString(d_r_val));
   addToLog("USER: Increasae Roll D Value to " + d_r_val);
   sendCValue(d_r_val,'c');
}
*/
public void INITIALIZE(int theValue){
   addToLog("USER: Request ANGEL Initialization");
   myPorts[0].write('V');
}

//The following two functions are used to send multi digit nums,
currently bugguy
/*
void sendThrottle(int value){
   holder = value;
   for(int i = 0;i<3;i++){
     arr[i] = holder % 10;
     holder /= 10;
   }
   myPorts[0].write('~');
   myPorts[0].write(byte(arr[2]));
   //delay(5);
```

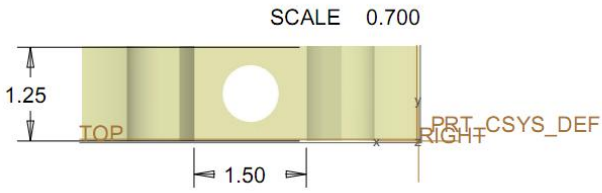
```
  myPorts[0].write(byte(arr[1]));
  //delay(5);
  myPorts[0].write(byte(arr[0]));
}

void sendCValue(int value, char ind){
  holder = value;
  holder = int(nf(holder,2));
  for(int i = 0;i<2;i++){
    dual_arr[i] = holder % 10;
    holder /= 10;
  }
  if(ind == 'a'){
  myPorts[0].write('!');
  }

  else if(ind =='b'){
    myPorts[0].write('@');
  }
  else if(ind == 'c'){
    myPorts[0].write('#');
  }
  myPorts[0].write(byte(dual_arr[1]));
  //delay(5);
  myPorts[0].write(byte(dual_arr[0]));
}
*/
void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP) {
      if(fast_climb){
        throt += 10;
        for(int i=0;i<10;i++){
          myPorts[0].write('~');
        }
      }
      else{
        throt += 1;
        myPorts[0].write('~');
      }
      if(throt > 250){
        throt = 250;
        addToLog("CONTROL: MAX Throttle = 250");
      }
      addToLog("USER: Increase Throttle to " + throt);
    } else if (keyCode == DOWN) {
      if(fast_climb){
        throt -= 10;
        for(int i=0;i<10;i++){
          myPorts[0].write('$');
          delay(10);
        }
      }
      else{
        throt -= 1;
        myPorts[0].write('$');
        delay(10);
```

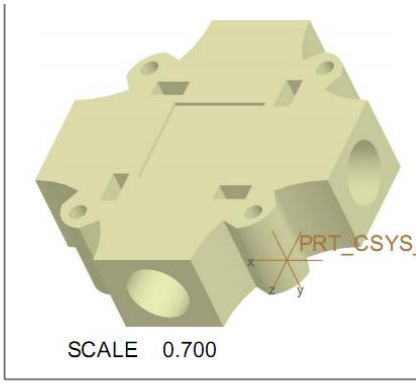
```
      }
      if(throt < 124){
        throt = 124;
        addToLog("CONTROL: MIN Throttle = 124");
      }
      addToLog("USER: Decrease Throttle to " + throt);
    } else if (keyCode == SHIFT) {
      fast_climb = !fast_climb;
    }
  }
  if(key == 'a'){
    if(fast_climb){
      for(int i=0;i<10;i++){
        myPorts[0].write('T');
        addToLog("USER: Roll LEFT");
        delay(10);
      }
    }
    else{
    addToLog("USER: Roll LEFT");
    myPorts[0].write('T');
    }
  }
  if(key == 'd'){
    if(fast_climb){
      for(int i=0;i<10;i++){
        myPorts[0].write('Y');
        addToLog("USER: Roll RIGHT");
        delay(10);
      }
    }
    else{
    addToLog("USER: Roll RIGHT");
    myPorts[0].write('Y');
    }
  }
  if(key == 'w'){
    if(fast_climb){
      for(int i=0;i<10;i++){
        myPorts[0].write('*');
        addToLog("USER: Pitch DOWN");
        delay(10);
      }
    }
    else{
    addToLog("USER: Pitch DOWN");
    myPorts[0].write('*');
    }
  }
  if(key == 's'){
    if(fast_climb){
      for(int i=0;i<10;i++){
        myPorts[0].write('+');
        addToLog("USER: Pitch UP");
        delay(10);
      }
    }
```

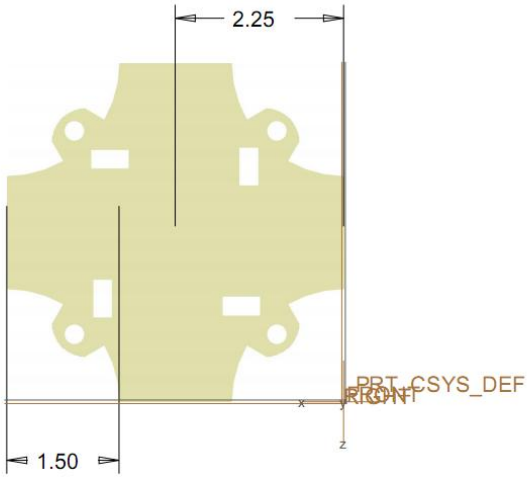
```
    else{
    addToLog("USER: Pitch UP");
    myPorts[0].write('+');
    }
  }
  if(key == '-'){
    myPorts[0].write('-');
    addToLog("USER: Roll P Decrease");
  }
  if(key == '='){
    myPorts[0].write(',');
    addToLog("USER: Roll P Increase");
  }
  if(key == '['){
    myPorts[0].write('/');
    addToLog("USER: Roll D Decrease");
  }
  if(key == ']'){
    myPorts[0].write('.');
    addToLog("USER: Roll D Increase");
  }

  if(key == ';'){
    myPorts[0].write('1');
    addToLog("USER: Pitch P Decrease");
  }
  if(key == '\''){
    myPorts[0].write('0');
    addToLog("USER: Pitch P Increase");
  }
  if(key == '.'){
    myPorts[0].write('3');
    addToLog("USER: Pitch D Decrease");
  }
  if(key == '/'){
    myPorts[0].write('2');
    addToLog("USER: Pitch D Increase");
  }
}
```

APPENDIX B – CAD Schematics

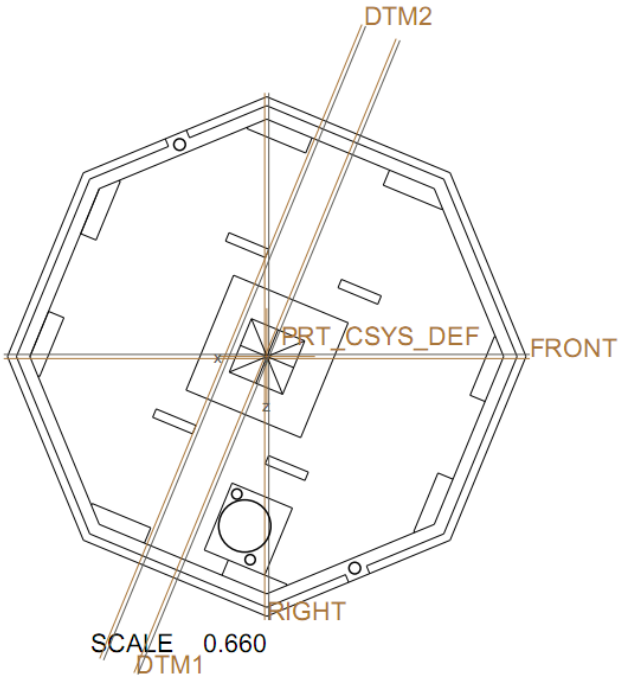
B-1 – ANGEL v1 Junction Drawing

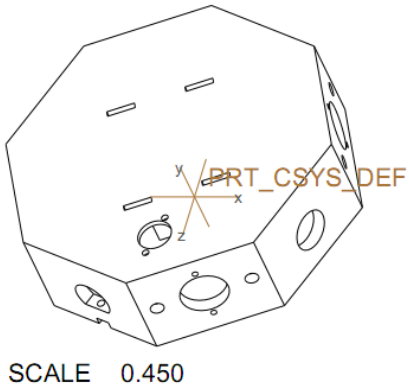




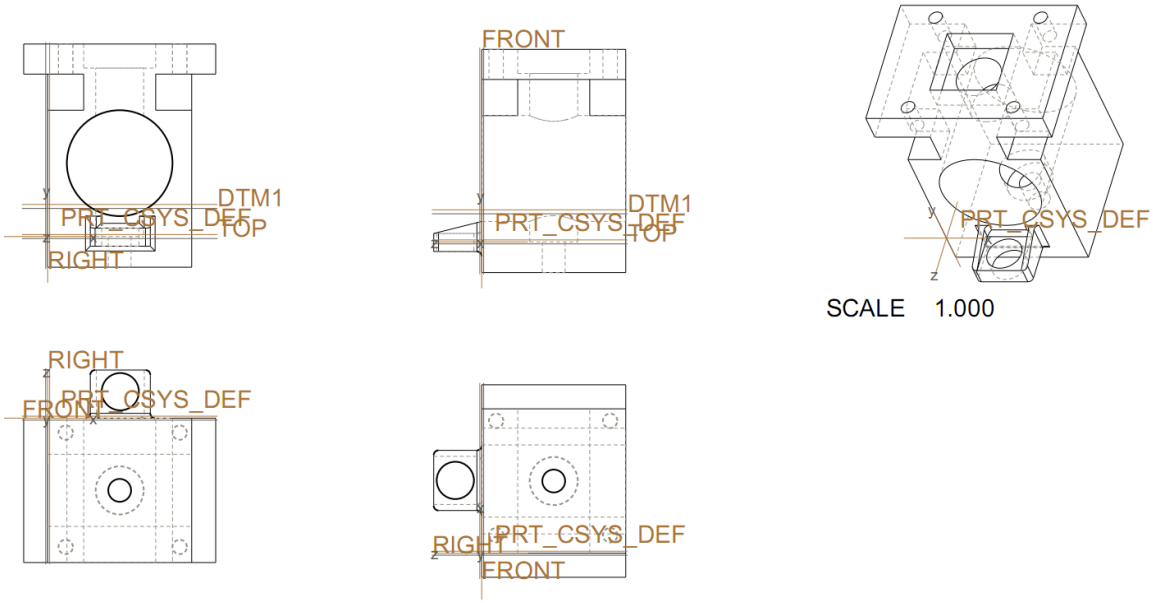


B-2 – Uriel Arm Junction Model (no dimensions)

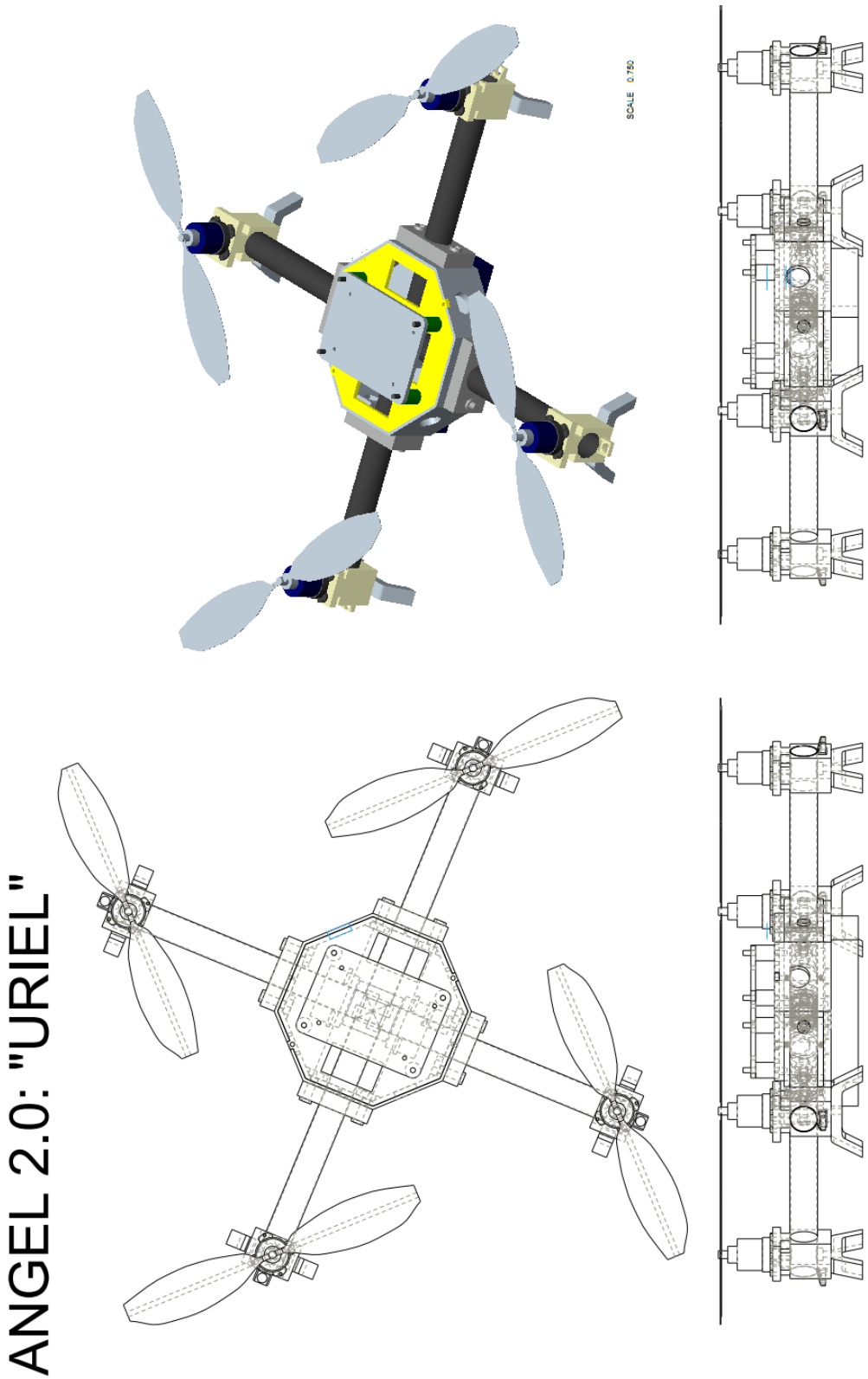




B-3 – Motor Mount for Uriel (no dimensions)



B-3 – Large Uriel Assembly Diagram



REFERENCES

[1] NOVA, "Battle Plan Under Fire – Time Line of UAVs". Available at: http://www.pbs.org/wgbh/nova/wartech/uavs.html)

[2] PRNewswire (Feb 1, 2010). Teal Group Predicts Worldwide UAV Market Will Total Over $80 Billion in its Just Released UAV Market Profile and Forecast. *PRNewswire*. January 2011 from http://www.prnewswire.com/news-releases/teal-group-predicts-worldwide-uav-market-will-total-over-80-billion-in-its-just-released-2010-uav-market-profile-and-forecast-83233947.html

[3] Valavanis, Kimon P., ed. *Advances in Unmanned Aerial Vehicles: State of the Art and the Road to Autonomy*. Dordrecht: Springer, 2007.

[4] Bouabdallah, S. & Siegwart, R. "Design and Control of a Miniature Quadrotor", *Advances in Unmanned Aerial Vehicles* (2007): 171-210.

[5] Premerlani, W. & Bizard, P. (May 17, 2009). *Direction Cosine Matrix IMU: Theory*. Available at: http://gentlenav.googlecode.com/files/DCMDraft2.pdf

[6] Mellinger, D., Shomin, M. & Kumar, V. "Control of Quadrotors for Robust Perching and Landing". Int. Powered Lift Conference, Philadelphia, PA, Oct 2010.

[7] Murray R., et al., *A Mathematical Introduction to Robotic Manipulation*, CRC, Boca Raton, FL 1994.

[8] Amir, M.Y.; Abbass, V.; , "Modeling of Quadrotor Helicopter Dynamics," *Smart Manufacturing Application, 2008. ICSMA 2008. International Conference on* , vol., no., pp.100-105, 9-11 April 2008
doi: 10.1109/ICSMA.2008.4505621
URL: http://ieeexplore.ieee.org.ezproxy.uky.edu/stamp/stamp.jsp?tp=&arnumber=4505621&isnumber=4505541

[9] Bouabdallah S. et al, "PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor", *Proceedings, IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

[10] The Aeroquad Open Source Project (Available at: http://www.aeroquad.com)

[11] Domingues, Jorge Miguel Brito, "Quadrotor Prototype", Universidade Tecnia de Lisboa, October 2009.

[12] Microstar Labs, "Ziegler-Nichols Tuning Rules for PID", Available at: http://www.mstarlabs.com/control/znrule.html

[13] Starlino, "A Guide to using IMU in Embedded Applications", Available at: http://www.starlino.com/imu_guide.html

[14] Higgins, W.T. "A Comparison of Complementary and Kalman Filtering," *Aerospace and Electronic Systems, IEEE Transactions on* , vol.AES-11, no.3, pp.321-325, May 1975
doi: 10.1109/TAES.1975.308081
URL: http://ieeexplore.ieee.org.ezproxy.uky.edu/stamp/stamp.jsp?tp=&arnumber=4101411&isnumber=4101405

[15] ZigBee Standard Comparisons, Available at: http://www.stg.com/wireless/ZigBee_comp.html

[16] Draganfly Innovations, Inc. Available at: http://www.draganfly.com/

[17] Parrot AR Drone. Available at: http://ardrone.parrot.com/parrot-ar-drone/usa/

VITA

Michael Schmidt was born in Honolulu, HI on July 28, 1986. He graduated summa cum laude from the University of Kentucky with Bachelor of Science degrees in Electrical and Mechanical Engineering in 2009. He was awarded the Robert L. Cosgriff Award at graduation by the College of Engineering and has passed the Fundamentals of Engineering exam. He has worked at the University of Kentucky Center for Visualization and Virtual Environments during his tenure as a student at the University. He wrote an award winning paper over his research on expanding a 2D image search algorithm to function in three dimensions. He also co-authored a paper currently awaiting publication on a new approach to Lean Management practices utilizing Latent Semantic Analysis.