

AA 449 Final Milestone Report
Blimp Project

June 11, 2010

Beth Boardman

Linh Bui,

Kyle Odland

Matthew Walker

Maggie Wintermute

Project Summary

The fundamental goal of the blimp project was to develop a controlled vehicle for the Distributed Space Systems Lab, which is attempting to assemble a test bed for assessing multi-vehicle control algorithms. It was decided that a neutrally buoyant blimp vehicle would provide a stable platform for testing control in three translational directions. The relatively slow reaction times of the blimp vehicle make it ideal for testing in an indoor environment. A group of AA/EE 449 students in a previous year had designed and flown a blimp vehicle; in order to improve upon the previous design, it was desired to lower the total weight of the vehicle (and thus decrease its volume) and improve upon the controller design.

To facilitate customization of the vehicle, it was decided to develop a new system rather than just modify the original. This included assembly of hardware, circuit board design, and software system integration. The ambitious initial project plan also included the construction of a second blimp vehicle and the development of algorithms to enable two vehicles to fly in various formations. The project goals were eventually revised once the challenges in implementing a single vehicle became apparent.

The most basic goal for the vehicle was to achieve way point tracking, the ability to respond accurately to position step commands. The project was The control system design approach focused on accurate and thorough plant modeling with the goal of creating a sophisticated and effective control system. The result of this was a set of linearized state space models that characterized the motion of the system in 4 degrees of freedom using two separate reference frames. The design approach for the hardware subsystem valued simplicity and low weight.

Table of Contents

1. Introduction and System Description.....	4
1.1. Project Overview	
1.1.1. Customer	
1.1.2. Goals	
1.1.3. Plan of Work	
1.2. System Description	
2. Related Work.....	8
2.1. Summary of Related Projects and Review of Literature	
3. Controls Subsystem.....	9
3.1. Plant Identification and System Modeling	
3.1.1. Reference Frames and Transformations	
3.1.2. Linearized System Model and Operating Regimes	
3.1.3. Controllability and Observability	
3.2. Control System Design	
3.2.1. Control System Architecture	
3.2.2. State Estimation	
3.2.3. Design Method	
3.2.4. Discrete Time Model	
3.3. Simulation	
3.3.1. Simulation Design	
3.3.2. Simulation Results	
3.4. Control System Implementation	
4. Software Subsystem.....	21
4.1. System Architecture	
4.2. Vicon Interface	
4.2.1. Signal Flow	
4.2.2. MATLAB Graphical User Interface	
4.3. MATLAB Controller Implementation	
4.4. Onboard Arduino Code	
5. Hardware Subsystem.....	25
5.1. Actuators	
5.2. Structure	
5.3. Blimp Envelope and Payload Attachment	
5.4. Microcontroller Components	
6. Power Subsystem.....	29
6.1. Introduction	
6.2. Design Procedure	
7. Results and Conclusions.....	34
8. References.....	38
8.1. Bibliography	
8.2. Use of References	
9. Appendices.....	39
9.1. Arduino Code	
9.2. GAL Code	
9.3. MATLAB Code	

1. Introduction and System Description

1.1 Project Overview

1.1.1 Customer

The Distributed Space Systems Laboratory researches self-organizing robotic distributed systems with application to aerospace missions. Notable examples of aerospace networked systems that are currently being studied in the Lab are formation flying space vehicles or self-organizing satellite constellations. The scope of this technology is much wider than space systems alone. One networked control system that is undergoing experimental verification is a collision avoidance algorithm, with obvious applications to many ground based systems.

To further develop and optimize these control systems, the Distributed Space Systems Lab is constructing a hardware testbed upon which these algorithms can be tested. This testbed will ultimately consist of a suite of small, autonomous robotic vehicles working in a cohesive network. At present, the Lab has two Lego Mindstorms Ground robots fully integrated with the tracking system and central computer. The fundamental goal of this project was to create two autonomously controlled, neutrally buoyant blimp vehicles to expand the capabilities of the Distributed Space Systems Laboratory testbed. A blimp model was chosen as the characteristically slower reaction time and inherent system stability make it an ideal vehicle for testing experimental control systems in an indoor facility.

The principal investigator for the Distributed Space Systems Laboratory is Professor Mehran Mesbahi of the Aeronautics and Astronautics Department. Professor Mesbahi and his graduate students defined the project goals, contributed the DSSL as a workspace for component testing and construction, and are supplying the funding for this project.

1.1.2 Goals

Specific project goals were outlined in order to fulfill the desires of the customer. These were modified slightly over the course of the project to reflect the results of unforeseen delays. The finalized goals established for the project are as follows:

1. Construct one working blimp vehicle that can complete simple waypoint tracking within the confines of the Distributed Space System Laboratory testbed facility, robust enough to be used for the future plans of the lab.
2. Incorporate the blimp vehicle into the existing lead-follow control algorithm employed by ground vehicles in the Distributed Space Systems Laboratory.

3. Construct a second working blimp vehicle that can complete simple waypoint tracking within the confines of the Distributed Space Systems Laboratory testbed facility, robust enough to be used for the future plans of the lab.
4. Derive and implement the control algorithms for a series of coordinated tasks for both operational blimp vehicles to conduct. This will start as a simple Lead/Follow scheme, and evolve to steadily more difficult tasks should project time allow.

At present, the first two goals have been completely met. The third goal, constructing and implementing the second blimp vehicle, can be completed in a relatively short time frame given that the first two have been achieved. Completing this objective, and developing the coordinated tasks, has been left to the future work of the DSSL.

1.1.3 Plan of Work

In order to ensure that all aspects of the project were completed on time, the team personnel were organized such that each team member was the manager of a particular subsystem of the project. Next, a plan of work outlining the tasks to be completed by each subsystem was detailed. The finalized plan of work, reflecting changes due to delays, is shown below.

Project Leader: Matthew Walker

Hardware Subsystem Lead: Beth Boardman

Software Subsystem Lead: Kyle Odland

Control Subsystem Lead: Maggie Wintermute

Power Subsystem Lead: Linh Bui

Subsystem	Task	Members	Deadline
Hardware			
1	Identify and order necessary structural components, motors and propellers	All	4/9
2	Finalize motor configuration	Boardman, Walker, Wintermute	4/9
3	Thrust test motors for plant model	Boardman, Walker, Wintermute	4/16
4	Develop blimp and gondola CAD model	Boardman, Walker	5/7
5	Final blimp and gondola design	Boardman, Odland, Wintermute	5/14
6	Construct blimps	All	5/21

Software	Task	Members	Deadline
1	Develop VICON camera system interface	Boardman, Odland, Wintermute	4/16
2	Research and define software system architecture between VICON camera system, central computer and microcontroller	All	4/23
3	Establish WiFi communication between Microcontroller and Central Computer, validate with simple task test	Boardman, Odland, Walker	4/30
4	Develop Microcontroller to circuit board communication	Bui, Odland, Wintermute	5/7
5	Develop control law code	Bui, Boardman, Odland	5/19
6	Implement control law onto microcontroller	Bui, Odland, Walker	5/21
Controls			
1	Develop Plant Model	Bui, Walker, Wintermute	4/16
2	Develop Waypoint Tracking Control Law	Bui, Walker, Wintermute	4/30
3	Develop coordinated task and formation control law	Boardman, Odland, Walker, Wintermute	5/7
4	Develop Simulink Model for control law validation	Odland, Walker, Wintermute	5/14
Power			
1	Identify and order necessary electrical components	Bui, Walker	4/9
2	Research Microcontroller– actuator interface, battery configurations	Bui, Boardman, Odland	4/30
3	Preliminary circuit board design, validate via breadboard	Bui, Walker	5/7
4	Final circuit board design, construct on prototyping board	Bui, Walker	5/14

1.1.4 Project Budget

As outlined in previous Milestone Reports, the exact components of the circuit board and system hardware have altered slightly over the course of the project. The following table details the cost of the components used for the finalized design and the dollar amount that the DSSL has been billed for.

Item	Cost per Item	Number	Total
1350 mAh 11.1 3 Cell LiPo Battery	35.00	2	70.00
Arduino Duemilanove	29.95	2	59.90
Circuit Board Development	-	50.00	50.00
Digi xBee Series 2 Pro Modules	42.00	2	84.40
L298n H-bridge	3	4.00	12.00
Misc. Structural	-	50.00	50.00
xBee Explorer Serial	30.00	1	30.00
xBee Explorer USB	25.00	1	25.00
Total			\$381.30

1.2 System Description

The blimp system itself consists of two main sections; the lifting balloon and the gondola. The gondola is the structure under the blimp which holds the electrical components in place. A diagram of the balloon and gondola is shown in Figure 1, which also includes variables representing the physical parameters of the system.

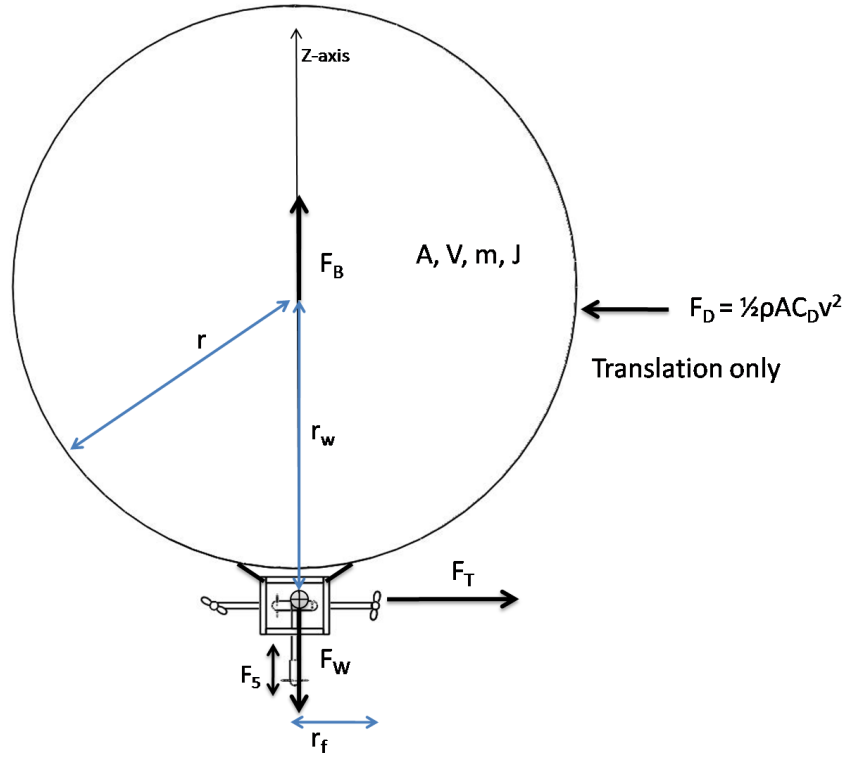


Figure 1. Diagram of blimp system

As Figure 1 shows, there are many parameters which relate to the system, most notably mass, moment of inertia, and the dimensions of the blimp. All of these parameters are tied together by the most basic force acting on the system: gravity. The blimp is inflated with helium to provide lift by displacing air. It is desired to keep the blimp system neutrally buoyant, therefore by estimating the likely payload mass, an approximate volume of helium needed for buoyancy was calculated.

Translational motion of the blimp will be provided by five DC motors. Four of the motors are placed on arms extending out horizontally from the gondola; these motors will be responsible for controlling the yaw of the blimp and translating it in the x and y directions. The fifth motor extends down below the gondola and will be used to facilitate translation in the z direction.

2. Body of Related Work

2.1 Summary of Previous Work and Review of Literature

As mentioned, the initial concept for a blimp vehicle for the DSSL was developed as part of an earlier AA/EE 449 Control System Design Course. The final Milestone report for this project, [4], was used as a starting point for the current project. In the previous project, the VICON camera system that currently makes up the sensor package of the DSSL testbed was not in place,

thus it was necessary for the older project to incorporate on-board sensors into the design. This drove different weight and power system constraints for that vehicle. Furthermore, without the external sensors, the earlier project was only able to control its orientation in yaw and was not able to achieve waypoint tracking about the lab. The control law developed on the older project for controlling orientation was a Ziegler-Nichols tuned PID design, which was determined to be insufficient for the goals of the current system.

In addition to the difference in sensor requirements, the Gumstix/Robostix microcontroller that was used for the earlier blimp vehicle was determined to not be the best option, per recommendations from the customer. Instead, an Arduino Duemilanove I/O board with an Atmel ATmega328P microcontroller was chosen for the current design. While the same motor configuration as the earlier project was used on this project, the sensor and controller changes necessitated an entirely new circuit board design.

Many resources were considered during the system modeling phase of the design process. [1] and [6] were particularly useful, given that the subjects of both papers were autonomous blimp vehicle projects. Heemstra's thesis, [3], was also reviewed given that the subject of that thesis was the control of another flying vehicle in the DSSL environment.

3. Controls Subsystem

3.1 Plant Identification and System Modeling

3.1.1 Reference Frames and Transformations

Modeling of the blimp system began as a very simply analysis of the forces on the vehicle. Much of the initial work on the plant model focused on the two distinct reference frames that may be used when working with a moving system. The first reference frame which was considered is the body centered frame, in which the axes of the frame are attached to the center of a moving body. Body centered frames are not inertial; however the blimp-centered reference frame allowed easy formulation of the forces on the vehicle by the motors. Since the body centered reference frame moves with the vehicle, it is impossible to use this frame along and track the motion of the blimp over time.

In order to facilitate tracking of the position of the blimp, it was necessary to work in an inertial reference frame. Commands to the system will necessarily be given in an earth centered reference frame, since positions commands are meaningless in a reference frame with a moving origin. For the purposes of this project, the DSSL lab space is considered to be an inertial frame.

The two reference frames are related rotationally by 3 Euler angles (typically called roll, pitch, and yaw). These angles represent the deviation of the body fixed axes from some fixed inertial

axes. Figure 1 shows the difference between the inertial reference frame and the blimp reference. A diagram demonstrating the forces and moments acting in the x-y plane in the blimp reference frame is shown in Figure 2.

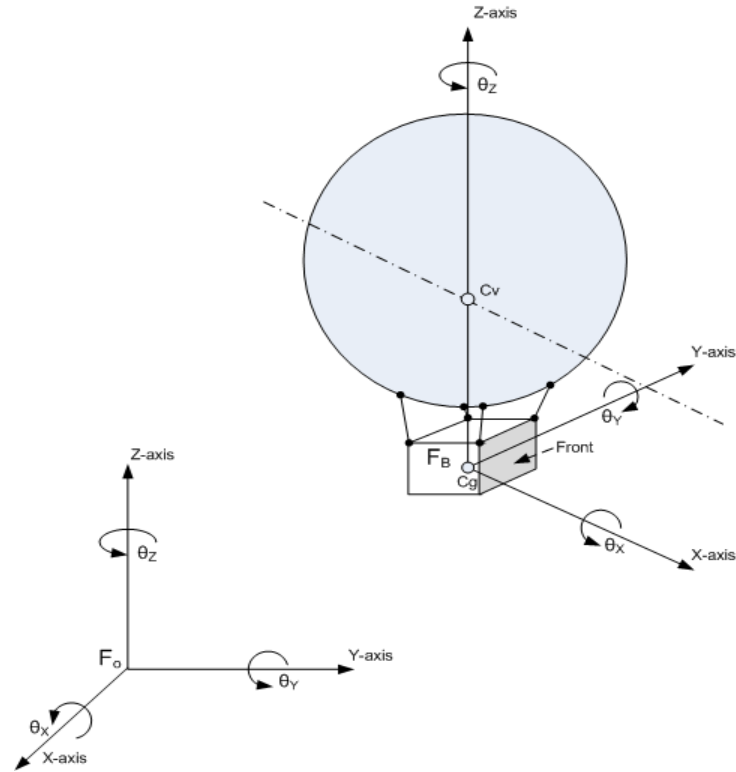


Figure 2. Blimp system reference frames

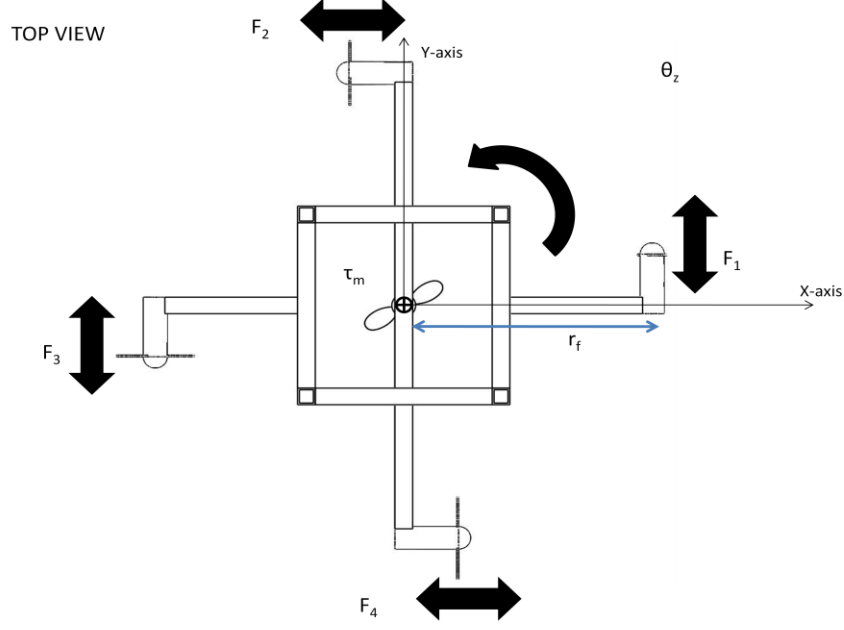


Figure 3. Top view of forces on blimp system, centered on body

As can be seen in Figure 2, the forces on the vehicle due to the firings of the motors are most easily accounted for in the body centered reference frame. Using this frame, the following equations of motion were developed for the blimp system. The subscript b indicates that the equations are formulated in the body-fixed frame. Initially all six degrees of freedom (three translational and three rotational) were considered.

$$\dot{x}_b = \dot{x}_b \quad (1)$$

$$\dot{y}_b = \dot{y}_b \quad (2)$$

$$\dot{z}_b = \dot{z}_b \quad (3)$$

$$F_{x,b} = F_2 - F_4 - D\dot{x}_b^2 + \dot{\theta}_z\dot{y}_b - \dot{\theta}_y\dot{z}_b \quad (4)$$

$$F_{y,b} = F_1 - F_3 - D\dot{y}_b^2 - \dot{\theta}_z\dot{x}_b + \dot{\theta}_x\dot{z}_b \quad (5)$$

$$\ddot{z}_b = F_5 - D\dot{z}_b^2 - \dot{\theta}_x\dot{y}_b + \dot{\theta}_y\dot{x}_b \quad (6)$$

$$\ddot{\theta}_{xb} = \tau_2 - \tau_4 + F_B r_B \cos \theta_y \sin \theta_x \quad (7)$$

$$\ddot{\theta}_{yb} = -\tau_1 + \tau_3 - F_B r_B \sin \theta_y \quad (8)$$

$$\ddot{\theta}_{zb} = -\tau_5 + r_f(F_1 + F_2 + F_3 + F_4) \quad (9)$$

The translational equations of motion that are formulated in the body frame include not only the identifiable forces due to the motor thrusts and drag (which is proportional to the square of velocity) but also a false force due to the Coriolis affect. This is a force which appears in rotating frames, due to non-inertial affects. The expression for the body frame accelerations is given below in equation 10. The cross product represents the Coriolis term, which crosses the rotational rates of the body frame with the velocities in the body frame.

$$\frac{d}{dt} \begin{bmatrix} \dot{x}_b \\ \dot{y}_b \\ \dot{z}_b \end{bmatrix} = \frac{1}{m} \vec{F}_b - \begin{bmatrix} \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix} \times \begin{bmatrix} \dot{x}_b \\ \dot{y}_b \\ \dot{z}_b \end{bmatrix} \quad (10)$$

The measurements which can be received from the VICON camera system are positions in the inertial frame. This means that output y of the state space model will not include body frame quantities. There are several ways to account for this in the system model. The first approach which was considered was to write the state vector of the system so that it included both the equations of motion in the body frame and the transforms to inertial frame in the A matrix of the state space model. The transformations between body and inertial frames are shown below in matrix form for both position and velocity. To transform velocities between the two frames, the rotation matrix for rotation about three axes must be used. It is given below in equation 11 for positions.

$$\begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = \begin{bmatrix} c\theta_y c\theta_z & c\theta_z s\theta_x s\theta_y - c\theta_x s\theta_z & c\theta_x c\theta_z s\theta_y + s\theta_x s\theta_z \\ c\theta_y s\theta_z & c\theta_x c\theta_z + s\theta_x s\theta_y s\theta_z & -c\theta_z s\theta_x + c\theta_x s\theta_y s\theta_z \\ -s\theta_y & c\theta_y s\theta_x & c\theta_x c\theta_y \end{bmatrix} \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} \quad (11)$$

Since the commanded quantities must match those in the y vector, it is also necessary to transform the angles if control over the vehicle yaw is desired. The Euler angles which define the rotation of the body frame with respect to the inertial frame also describe the rotation of the craft, since the body axes are fixed to the blimp center. The transformation between rotational rates in the two frames is shown in equation 12.

$$\begin{bmatrix} \dot{\theta}_{x,r} \\ \dot{\theta}_{y,r} \\ \dot{\theta}_{z,r} \end{bmatrix} = \begin{bmatrix} 1 & t\theta_y s\theta_x & t\theta_y c\theta_x \\ 0 & c\theta_x & -s\theta_x \\ 0 & \frac{s\theta_x}{c\theta_y} & \frac{c\theta_x}{c\theta_y} \end{bmatrix} \begin{bmatrix} \dot{\theta}_{x,b} \\ \dot{\theta}_{y,b} \\ \dot{\theta}_{z,b} \end{bmatrix} \quad (12)$$

The initial formulations of the state space blimp system involved many diverse combinations of the transformation equations shown above. In order to simplify the system, a system design was considered in which the state vector contained only body frame quantities. The A matrix held the dynamic equations of the system and the C matrix was used to transform to the inertial reference frame. Later in the controller design process, however, this approach presented many problems caused by issues of controllability. This is directly related to the form of B matrix, which is dictated by the physical actuators (motors) of the system. The actuators have direct control over translation in all three directions, but strong authority for rotation only about the z axis. Initially the system was modeled to include torques on the system due to drag of the propeller blades; however later testing concluded that these torques would be negligible. Since the system is passively stable in both roll and pitch, it was decided to remove these degrees of freedom from the state vector. This resulted in a reduced system which contained state variables in both reference frames. The state vector is shown in equation 13.

$$\vec{x} = \begin{bmatrix} x_r \\ y_r \\ z_r \\ \dot{x}_b \\ \dot{y}_b \\ \dot{z}_b \\ \theta_z \\ \dot{\theta}_{z,r} \end{bmatrix} \quad (13)$$

3.1.2 Linearized System Model and Operating Regimes

The state space representation using the above state vector contains both system dynamics and transformation equations in the A matrix, while the C matrix is composed entirely of ones and zeros. Since the transformation equations and the equations of motion are non-linear, it is necessary to linearize the system before a state space representation can be formulated. This is accomplished by selecting a single operating point, differentiating the nonlinear equations and holding all quantities constant at the operating point. In order to maintain proper operation of the vehicle for all yaw angles, multiple linearized system models are needed. Below is an example of one possible operation point, in which the system has been linearized about zero for all quantities except body frame velocities and yaw angle.

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -2DU_x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2DU_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2DU_z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (14)$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ a & 0 & -a & 0 & 0 \\ 0 & -a & 0 & a & 0 \\ 0 & 0 & 0 & 0 & a \\ 0 & 0 & 0 & 0 & 0 \\ al & al & al & al & 0 \end{bmatrix} \quad (15)$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (16)$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (17)$$

3.1.3 Controllability and Observability

The state space representation detailed in equations 14-17 models the blimp as a linear, time-invariant system about a single operating point. In the process of designing a control system, once a linear model has been achieved, classical linear control design techniques become applicable in this regime. However, additional verifications to the state space representation are necessary before control design can commence.

In general, a given state space representation of a linear, time-invariant system is not unique. In theory, any combination of states, inputs and outputs that characterizes the behavior of the system can modeled in the traditional state space form. Before control can be implemented on a system that is represented by a given state space model, this representation of the system must be completely state controllable and completely observable. [7]

A system is defined as completely state controllable if the unconstrained control inputs can transfer the system between any initial and final state. [8] Essentially, if a state variable exists in a system that cannot be affected by the inputs, the system is not controllable. The controllability of a system can be rapidly determined from the A and B matrices of a state space model by comparing the rank of the controllability matrix, which is shown in equation 18, to the number of state variables.[7] Rank is defined as the number of linear independent equations represented by a certain matrix. The system is controllable if the rank of the control matrix is equal to n, the order of the system, or the number of state variables. [8]

$$C_M = [B \quad AB \quad A^2B \quad \dots \quad A^{n-1}B] \quad (18)$$

A system is defined as completely observable if every state can be found from the system output. Fundamentally, a system is completely observable if all relations on the system state affect the system output. As with controllability, the observability matrix of a state space representation can be rapidly determined using the A and C matrices, as shown in equation 17. A system is observable if the rank of the observability matrix is equal to n, the order of the system.[8]

$$O_M = [C \quad CA \quad CA^2 \quad \dots \quad CA^{n-1}]^T \quad (19)$$

Initial iterations of the system model were not fully controllable, and as a result investigation was conducted into performing a minimum realization on the system. The final system model, however, is both fully controllable and observable.

3.2 Control System Design

3.2.1 Control System Architecture

The basic purpose of the control system is to guide the blimp to its desired reference position with minimal steady state error. The performance goals for this project included crossing the VICON camera space in under 30 seconds and arriving at the commanded reference position with negligible error. In order to fulfill the latter goal, it was determined that some sort of integral control would be needed. The basic control scheme which was selected for the blimp system combines full state feedback with integral control on the error. A block diagram of the controller architecture is shown in Figure 4.

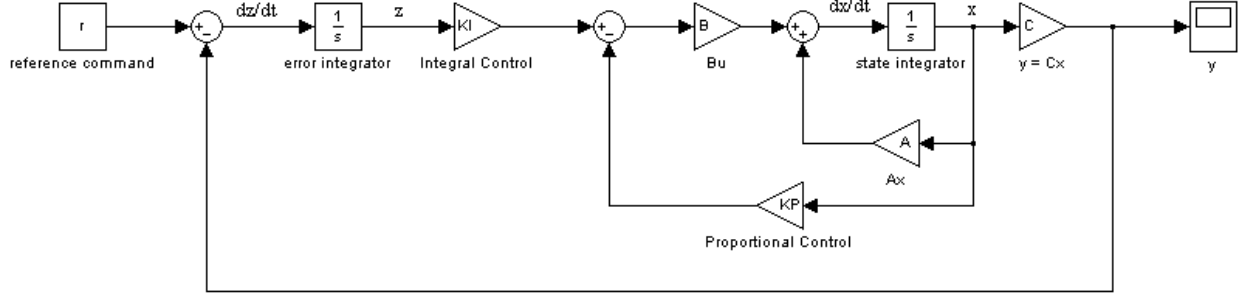


Figure 4. Blimp system architecture

Introducing an integral control component to the system essentially creates new states within the system; in order to determine the proper gain values for the controller matrices, it was necessary to formulate a new state space representation of the system. Using the diagram shown in Figure 3, the following augmented equations were formulated.

$$\frac{dx}{dt} = Ax + Bu \quad (20)$$

$$\frac{dz}{dt} = r - Cy \quad (21)$$

$$u = -K_p x - K_I z \quad (22)$$

$$\frac{dx}{dt} = Ax + B(-K_p x - K_I z) \quad (23)$$

$$\frac{d}{dt} \begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} A - BK_p & -BK_I \\ -C & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} - \begin{bmatrix} 0 \\ B \end{bmatrix} r \quad (24)$$

3.2.2 State Estimation

In order to use the controller design discussed above, the system must have access to all eight state variables. In the physical system the only states that can be physically observed are the x , y and z positions of the blimp, along with the yaw angle. One way of determining the state vector is to use an estimator, which the modeled dynamics of the system to calculate estimated states from the system output, y . One simple estimator design is the Luenberger observer, the dynamics of which are given in equation 25.

$$\dot{\hat{x}} = A\hat{x} + Bu + L(y - C\hat{x}) \quad (25)$$

Figure 5 shows a diagram of the Luenberger observer.

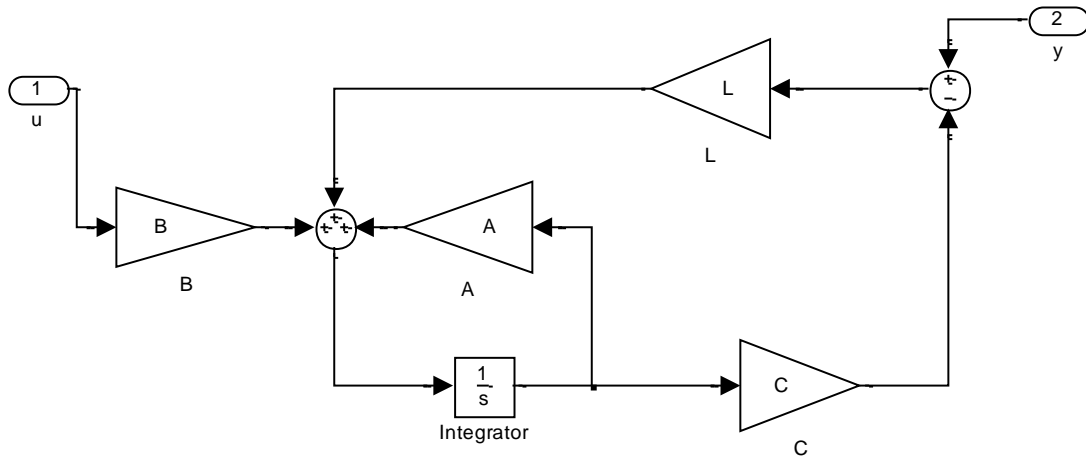


Figure 5. Luenberger observer

This observer uses a state space representation that is similar to the actual system, but includes a term that compares the actual output (y) to the output that is estimated by the observer ($C\hat{x}$). The observer gain matrix L is found by considering a new state, the error between the actual and estimated states. The

$$e = x - \hat{x} \quad (26)$$

$$\dot{e} = \dot{x} - \dot{\hat{x}} = Ax + Bu - (A\hat{x} + Bu + L(y - C\hat{x})) \quad (27)$$

$$\dot{e} = Ax - A\hat{x} + L(Cx - C\hat{x}) = A(x - \hat{x}) + LC(x - \hat{x}) \quad (28)$$

$$\dot{e} = (A - LC)e \quad (29)$$

By placing the poles of the $A-LC$ matrix to be in the left half plane, the error between the estimated states and the actual states will be driven to zero. The gain matrix L can thus be found by placing the poles of $A-LC$. The poles of the observer were chosen to be 10 times farther to the left of the imaginary axis than the system poles, so that the estimator error would go to zero much faster than the system response. Figure 6 shows the estimated position states (dotted line) of the blimp compared to the actual positions (solid line).

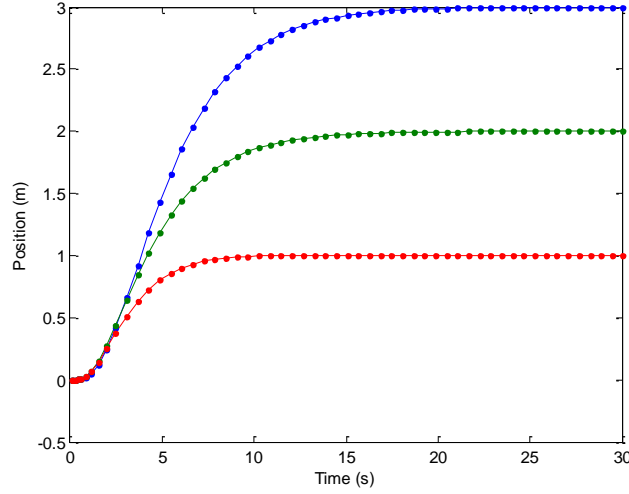


Figure 6: Comparison between estimated and actual states in simulation

The estimated position states are plotted as a function of time, and closely follow the actual states. This is expected, due to the fact that the position states are observed and can be directly compared to the estimates. Velocity estimates were less precise than position estimates due to the presence of finite noise due to drag which was included in the Simulink model.

Though the initial design for the control system used an observer to achieve full state feedback, it was eventually determined that the observer was not necessary, and that the non-observed states could be more accurately calculated by directly differentiating the position measurements and then transforming the velocities from reference to body centered frame. This was implemented using a second order accurate backwards differentiation scheme that makes use of multiple past measurements to avoid unwanted noise in the derivative. Once the reference frame velocities are known, body frame velocities are calculated using Coriolis equation, shown in equation 11.

3.2.3 Design Method

In order to place the pole of the system described in the system architecture system, the augmented A matrix was broken into a form that allowed the use of the MATLAB command “place.” The breakdown of the augmented A matrix is shown below.

$$A_{aug} = \begin{bmatrix} A & 0 \\ -C & 0 \end{bmatrix} - \begin{bmatrix} B \\ 0 \end{bmatrix} [K_p \quad K_I]$$

The poles used in the place command were determined with the intention of creating stability in the system; thus they were chosen to all have negative real components. The poles were also chosen to be fairly close to the imaginary axis, since the open loop poles of the system are all quite close to 0. The gain matrices K_I and K_P are shown in Appendix A. These matrices were generated for a linearization point of non-zero x and y velocities (this will be the nominal operating condition of the blimp when it is not rotating) with all other variables set to zero. The

code that was used to generate the controller matrices allows the linearization points for velocity and yaw angle to be changed. In order to allow the blimp to operate in all possible yaw angles, multiple controllers have been developed, each with a pre-determined range of valid operation.

3.2.4 Discrete Time Controller Model

In addition to the continuous time model described in the previous sections, a discrete time model was also created. This model used the same system dynamics as the continuous time model, transformed into a model with a discrete time step using the MATLAB command “c2d.” Controller design for discrete systems requires that all system poles be within the unit circle; this causes the system error to eventually converge to zero. Though a discrete model was developed, it was determined that the continuous controller was sufficient to operate the blimp system.

3.3 Simulation

3.3.1 Simulation Design

In order to model the open loop behavior of the system and test the effectiveness of various controller designs, a Simulink model of the system was developed. The closed loop system model is shown in Figure 7.

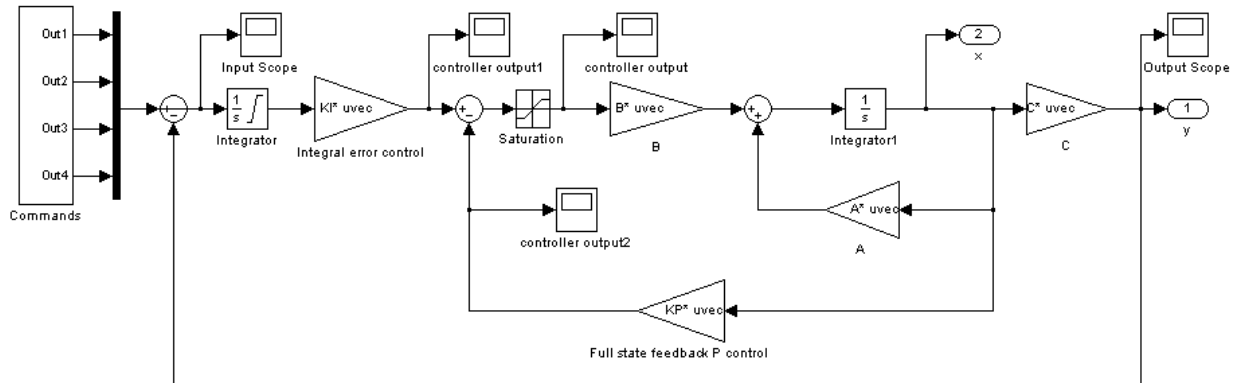


Figure 7. Simulink model of continuous blimp system

One of the most important aspects of the Simulink model is the block which saturates the output of the actuators. This block adds a needed degree of realism to the system; since the motors themselves cannot tolerate above 6 volts, it is unfeasible to design a controller which requires incredibly high control effort which can never be delivered in the real system. Though it cannot be seen directly in the model, there is also a saturation limit in place on the integration block which acts on the error. This prevents the system from winding up and overcompensating for remembered error.

3.3.2 Simulation Results

The model shown in Figure 7 was used to test many controller iterations until a continuous time controller was found that satisfied all of the performance requirements for the system. The step responses of the system using one such controller are shown below in Figure 8.

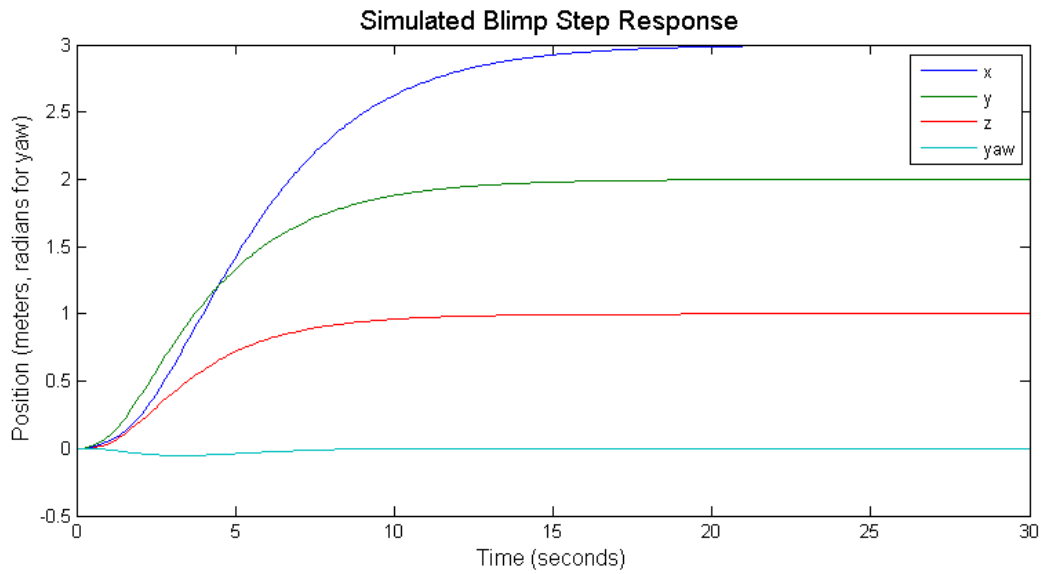


Figure 8. Step response of compensated blimp system

Though the compensator effort shown that produces the results in Figure 8 does allow the system to meet the desired performance goals, it does not act with reasonable use of control effort. The motor voltages which correspond to the step response in Figure 8 are shown in Figure 9.

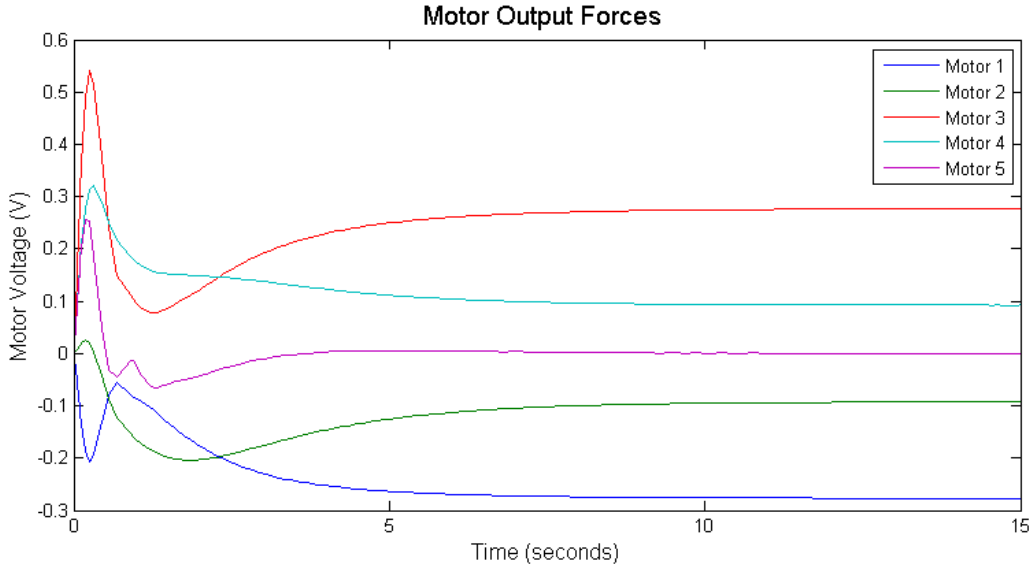


Figure 9. Motor output voltages in response to step commands

This problem of non-zero steady state control voltages was a direct result of the addition of drag into the Simulink model, and the chosen operating point of the linearized system. When the operating point velocities of the system were set to zero (meaning that no effects of drag were considered) the control voltages all converged to zero. This indicates that the problem of the control voltages is solvable in two ways; firstly, establishing a set of controllers for different operating points would allow the system to operate with zero set velocities when it approaches the commanded reference position, otherwise, it would be possible to introduce a number of logical statements into the controller code that command the motor voltages to zero once the steady state position has been achieved.

3.4 Control System Implementation

Implementation of the control system was handled in both software and hardware. The position of the blimp is sensed by the Vicon camera system, the data from which is read by a computer in the DSSL lab. Data from the Vicon computer is accessed through a client code which both interprets the position data and calculates the controller outputs. This information is communicated wirelessly to the blimp. The microcontroller and circuit on the blimp gondola interpret the control inputs and send duty cycle and direction commands to the actuators. The details of both the hardware and software aspects of controller implementation will be presented in subsequent sections of this report.

4. Software Subsystem

4.1 System Architecture

The current system architecture for the blimp project is shown in Figure 10. The initial plan for the software implementation was to run the collection of sensor data and the control loop in Simulink, and then send the output data to the blimp via serial connection. Data is gathered from the Vicon using a client programmed in MATLAB; the client is capable of polling data and storing it, or sending it via serial, but it was not able to interface in real time with Simulink. For this reason, we next attempted to utilize the C# Vicon interface which was created by the DSSL. Though the C# interface is fully functional, our team members are more comfortable working with MATLAB. The customer also stated that a working system using MATLAB would be desirable for future use by the lab. For these reasons, it was decided that the final system architecture would implement the MATLAB Vicon client to poll the marker data. The controller is also implemented in the MATLAB code. The position and angle data from the Vicon cameras is differentiated numerically to determine the full state of the system. Once this is known, the desired control is applied and the output voltage to each motor can be determined. This information will then be sent to the Arduino board, where the motors will be controlled.

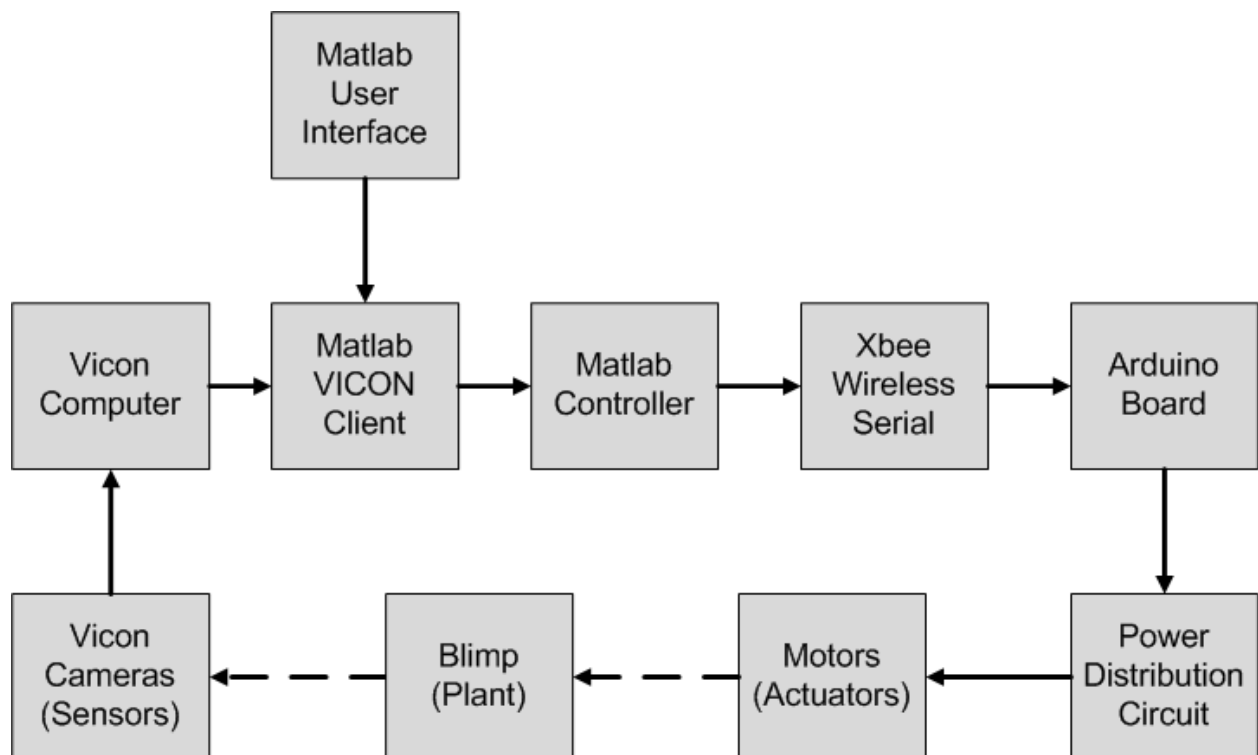


Figure 10. High Level Blimp System Architecture

4.2 Vicon Interface

4.2.1 Signal Flow

The interface with the Vicon camera system is provided through an existing Matlab client. The client runs through a file which continuously loops, polling data from the Vicon computer as it iterates. Figure 11 shows a flow chart detailing how the control software works.

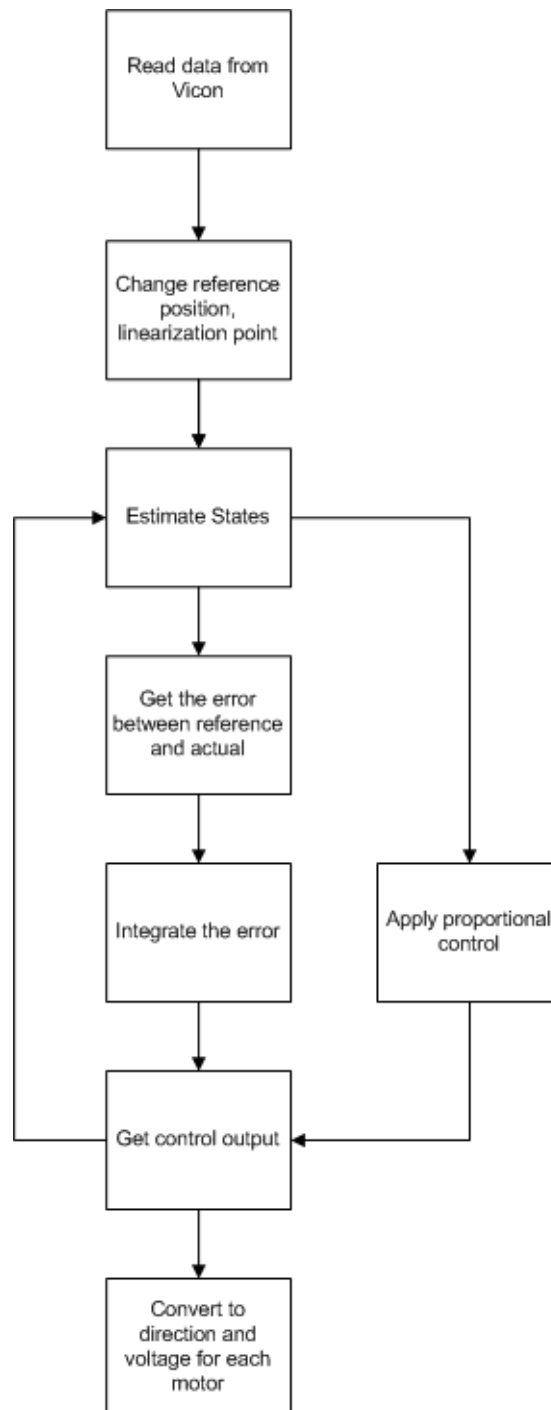


Figure 11: Schematic showing the flow of data in the Vicon interface code

The marker data received from the Vicon is then used to calculate the position of the blimp's center of gravity, as well as the heading angle. This is done with simple geometric relations between any two of the reflective markers. In order to be sensed by the Vicon, each marker must be in view of at least two of the six cameras. Since the larger envelope of the blimp is above the marker positions, markers are periodically lost from sight. In this case the Vicon system returns values of zero for all three positions. In order to protect against bad data from calculating position from lost markers, six separate cases were written so that the final position is always calculated from markers that are directly observed.

4.2.2 MATLAB Graphical User Interface

The reference position can be changed as the MATLAB code is running using a MATLAB user interface. The GUI allows the user to input a desired position in x, y, z, and yaw, which changes the variables in the looping file. A picture of the interface is shown in Figure 12.

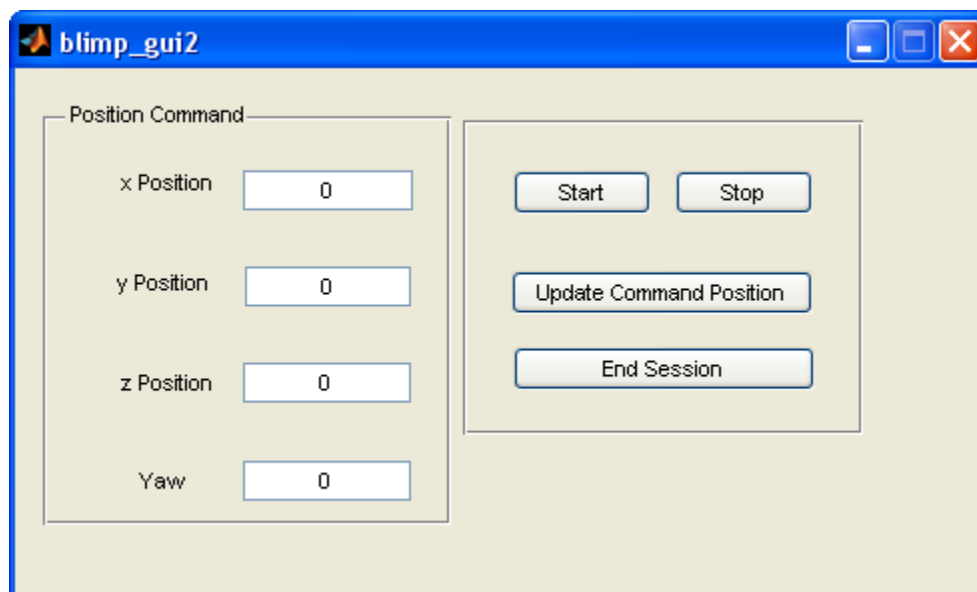


Figure 12. Graphical user interface for blimp reference position updates during flight

As seen in the figure, the GUI can also be used to start and stop the blimp control code.

4.3 MATLAB Controller Implementation

With the reference position and the position of the blimp established, the designed controller can be implemented in the code. The position and angle data is differentiated numerically, so that the full state of the system can be determined. The state vector used in the control system design includes inertial frame positions but body frame velocities. In order to provide the correct full state feedback data to the controller, the inertial frame velocities are transformed into the body frame using a vector rotation about the three Euler angles. Proportional gain K_p is applied to the full state of the system using a gain matrix established in the controller design.

The reference error of the system is found by subtracting the observed position of the blimp from the commanded position. This error is integrated using a simple trapezoid rule, and integral gain matrix K_I is applied. The proportional control voltages are then subtracted from the integral control voltages to get the command voltage to each motor. Due to the constraints of the selected motors, a saturation limit of 6 V is applied to the commanded voltages.

The command voltages must be converted into a direction command and a pulse width for each motor. The direction commands are determined by checking if the desired command voltage is positive or negative, and are set to bytes of 0 or 1. The pulse width is determined by comparing the command voltage to the nominal 12 V supplied by the battery. The max command PWM byte that can be sent to any motor is 255, so the actual PWM command for each motor is a single byte from 0-127 (half of the max). If the motor is firing in the positive direction, a factor is applied to the PWM command to account for the lower thrust generated when the motors are running backward. This will allow outputs from the controller in either direction to create the same net force with no dependence on directionality.

A 10 byte array is formed, consisting of 5 pulse width commands and 5 direction commands, and sent to the Arduino via serial communication over the Xbee modules.

4.4 Onboard Arduino Code

The code on the onboard microcontroller is responsible for reading in the direction and pulse width commands and for writing these commands to the desired motors. The 10 byte array is read in from the Xbee module using serial. The direction commands write a High or Low command to each of the 5 direction output pins, and the pulse width commands write a duty cycle to each of the 5 pulse width output pins.

The MATLAB code and the code on the Arduino are included in the Appendix.

5. Hardware Subsystem

5.1 Actuators

The actuators for the blimp system are the five motors. Before selecting the final motors for the blimp, a few experiments were run to determine the thrust, voltage and amperage of the motors. A ducted propeller and bi-directional propeller were both tested. After evaluating the motors it was determined that the ducted propellers would not be ideal for the blimp. Bi-directional propellers are the best for our situation. This would allow the blimp to rotate clockwise and counter-clockwise, hence rotating to the desired heading quickly. The ducted propellers were

removed from the motor and replaced by a bi-directional propeller. Upon doing a thrust test, the motor became over heated and was drawing too much current, consequently this burned out the motor. For this reason, it was decided to go with a motor that are better equipped to run the bi-directional propellers. The selected motor-propeller system chosen is the GWS EDP-50XC electric motor which runs at 6V drawing 1.35A and produces 0.5N of thrust.

In order to adequately control the system, it was necessary to determine the relationship between input voltage and the resulting thrusts and torques generated. As anticipated, the motors were found to be more efficient in one direction than in the other. Therefore, to achieve the equal thrust in both directions the voltage will need to be capped differently in each direction, and the duty cycle commands from the controller scaled accordingly. In the inefficient direction the cap will be 6V, the maximum the motor is rated for, this will produce a thrust of 0.25N. To achieve this same thrust in the efficient direction, the voltage must be capped at 4V. From these thrust voltage pairings, the coefficients used in the system model can be determined. Figure 13 shows the results of the thrust test. A motor torque test was also conducted. The toques were found to be sufficiently small enough to be negligible in the pitch and roll directions.

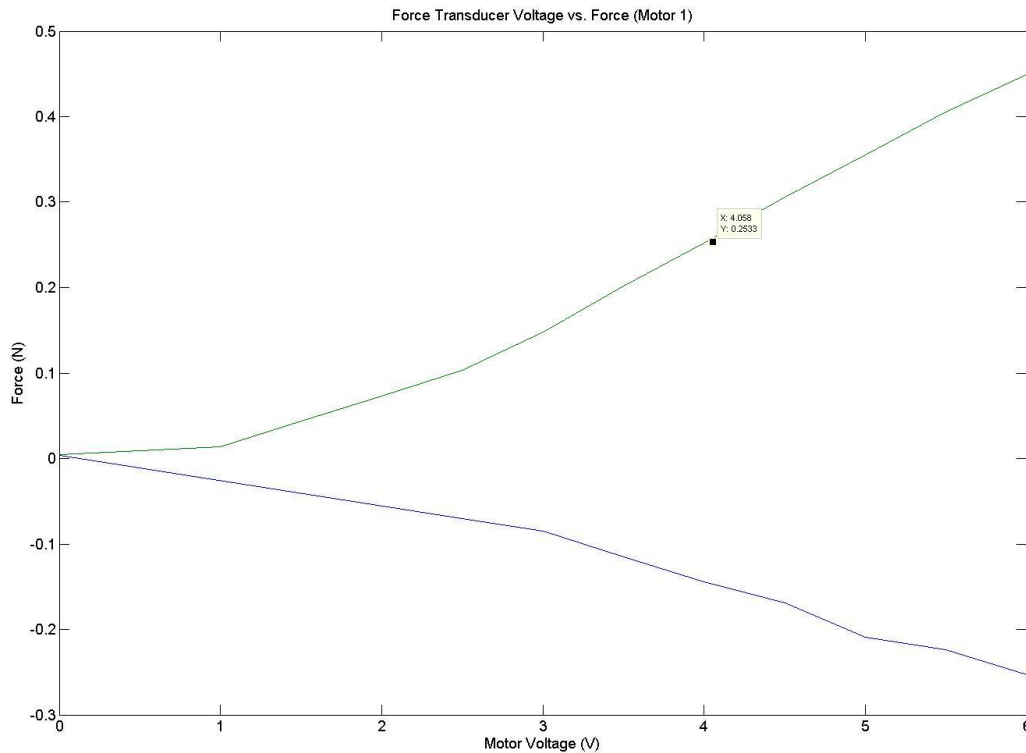


Figure 13. Results of motor thrust testing in two directions

5.2 Structure

The main function of the gondola is to house the onboard components. The primary design requirement was to have enough surface area for mounting all the components. The secondary design requirement is mass limitation. It has been estimated that the structure mass should be about 50 grams for a neutrally buoyant blimp. The next design requirement is to comply with the performance requirement of being able to rotate 360 degrees in 30 seconds. Also for performance, the motors must be far enough away from the main structure to avoid air inference. The final design requirement is to keep the manufacturing processing simple.

Keeping these design requirements in mind, a preliminary design was formed and built. To keep the gondola compact, a double deck configuration was chosen. For simplicity, the main structure was designed as a cube. Version1 of the gondola is in Figure 14, and has a mass of 92 grams. 92 grams is nearly twice that of the estimated mass, which could be too heavy for the blimp envelope to lift. The motor arm length of 20 cm allows the blimp to rotate 360 degrees in 30 seconds which meets the design requirements. However, the motor arms are attached poorly as when the motor is fired the arm swings back about half a centimeter.

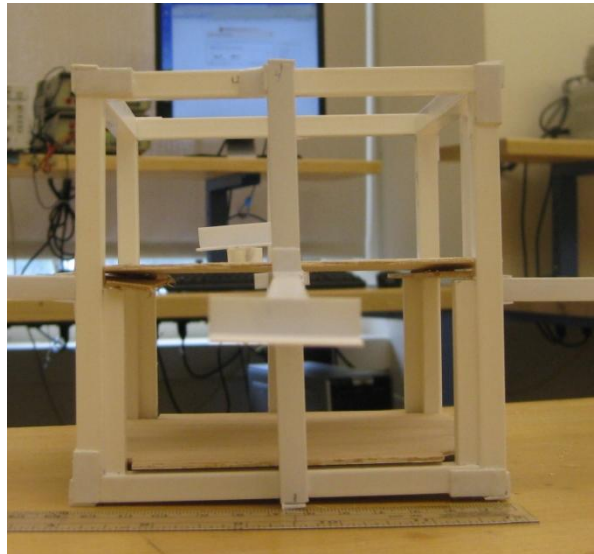


Figure 14. First version of blimp gondola

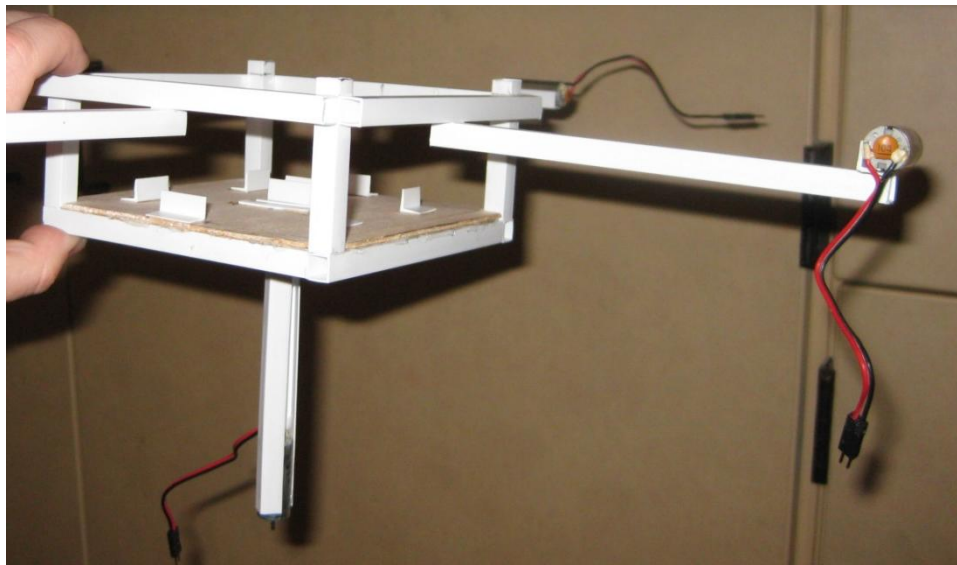


Figure 15. Final gondola design

A version2 was built to fix version1's problems. Version2, in Figure 15, is a squat double deck cube. The deck spacing is just enough that the Arduino and xBee hardware will fit comfortably. The mass of the version2 gondola is 49 grams, extremely close to what was estimated. The motor arm length was kept at 20 cm, to meet the design requirement. But their attachment was redone. By attaching the arms to a horizontal strut rather than a vertical one the arms are much more stable. The motor to move the blimp in the z-direction is 10 cm from the bottom of the blimp. This is enough space to mitigate any unwanted disturbances caused by air flow against the gondola. The main structure is composed of 7.9 mm square styrene tubing. Plastruct Bondene is used to attach the styrene tubing together. An epoxy will be used to attach the motor to the motor arms. Each motor is attached to the gondola at the end of the motor arm by sitting in a type of two sided cradle.

5.3 Blimp Envelope and Payload Attachment

Several envelopes were looked into before settling on the 4 foot diameter envelope purchased from Mobile Airships. This envelope is much larger than was initially desired. One of the earliest project goals was to reduce the payload weight of the blimp in order to reduce the overall envelope volume that would be needed. The initial calculations using the estimated weight of the gondola and power board indicated that an envelope with diameter just over 3 feet would be sufficient. In practice, however, finding durable and reusable envelopes of the correct diameter proved more difficult than was anticipated. The weather balloons that were tested proved to be either too small or too fragile. It was finally decided to use the larger envelope from Mobile

Airships due to its ease of use (helium can be added to the blimp through a plastic valve) and relative robustness.

The requirements for the attachment device are that it needs to be lightweight, rigid enough to turn the envelope and gondola together and it must not slip off the envelope. A wire frame was tested but was determined to be too heavy for the blimp to lift and slipped off the balloon. A net system which was fitted over the top of the blimp was also tested, but was determined to also be too heavy. The final configuration uses a Velcro attachment in which straps from the gondola are attached directly to strips fixed to the sides of the envelope. This configuration is both relatively lightweight and sturdy.

The blimp model used for the controls requires the blimp be neutrally buoyant and the center of mass be known. To meet these requirements variable weights are added. The weight system used is four screws glued to the corners of the gondola with washers added to vary the weight.

5.4 Microcontroller Components

The electronic hardware which is used on the blimp consists of three main components: the Arduino Duemilanove I/O board, the Xbee wireless communication module, and the Xbee shield. These three components work in tandem to receive serial data from the controller and distribute the appropriate pulse width and direction signals to the circuit board.

The Xbee module on the gondola is configured to receive data transmitted by an Xbee which is connected to the computer on which the control code is run. The Xbees have been configured to operate at a high baud rate of 115200 bits/second, which allows the onboard system to receive all the data transmitted from the Vicon with minimal delays. Data received by the Xbee module is passes to the I/O board through the shield, which facilitates the necessary electrical connections between the two smart components. Once the controller output data is passed to the Arduino, the onboard code interprets the input vector and passes duty cycles and directions to the appropriate output pins.

The main design criteria when selecting these components was ease of use. The Arduino board itself was selected for the relative simplicity involved in coding and forming electrical connections.

6. Power Subsystem

6.1 Introduction

A five bidirectional motor circuit design was constructed to meet these requirements:

- Operate from supply voltage of 11.1V
- Motor can turn in both directions with various speeds as commanded

- Minimal overall weight
- Minimal overall power consumption
- Minimal total part count
- Minimal total price

6.2 Design Procedure:

List of initial components:

- One 20C 3 cell 1350 mAh 11.1 V Lithium Polymer battery
- Five single 5 A H-Bridges
- One generic array logic GAL22V10D
- Two quad optocouplers
- One dual optocouplers
- Supporting diodes, resistors and capacitors

Battery: The whole circuit is powered by one 20C 1350mAh 11.1V 3 cell Lithium Polymer (LiPo) battery that is charged by a TP-610C charger. According to the datasheet of the Arduino, it has the normal operating voltage between 7V and 12V. Even though the absolute minimum operating voltage is 6V, there are no clear information on how Arduino is supposed to behave at this low range of 6V and 7V. The battery is very light-weight (94g), can last up to 30 minutes during normal .

Charger: The TP-610C charger can charge, discharge and balance any kind of 1 to 6 cell Lipo, A123 cells, and 14 NiCd/MH cells and 6 – 12V Pb batteries. It can check to see if the voltage of each cell is above the cutoff voltage and detect when the battery is fully charge to prevent damages to the battery.

H-Bridges: To accommodate the initial motors' drawing current (3.2A) and voltage (7V) at full thrust, five 5A single H-Bridge chips TLE52052 were chosen. These chips require two inputs and two outputs. Its different combinations of inputs would produce different combinations of outputs to make the motor turn clockwise, counter clockwise or stop as shown below in **Error! Reference source not found..** They also get logic supply from the output pin 5V of the Arduino.

Table 1: 5A H- Bridge TLE52052's Original Truth Table

Input 1 (PWM signal)	Input 2 (Direction)	Output 1	Output 2	Action
L	L	H	L	Motor turns clockwise
L	H	L	H	Motor turns counter clockwise
H	L	L	L	Brake
H	H	Z	Z	Open Circuit

Generic array logic chip: The above H-Bridges only accept CMOS/TTL compatible inputs and have an unusual truth table combination for the inputs as seen above in Table 1, so 1 chip GAL22V10D was used to give the TLE52052 chips their required inputs with the desired combinations as seen in Table 3. This chip has fourteen possible inputs with one input being a clock if needed and ten possible outputs. The truth table inside the GAL chip to change the input combinations is shown below in Table 2. Its voltage supply also comes from the 5V output pin of the Arduino since this is the typical logic supply for this chip to be able to operate normally.

Table 2: GAL22V10D's Truth Table for TLE52052

Input 1 (Old PWM signal)	Input 2 (Old Direction)	Output 1 (New PWM signal)	Output 2 (New Direction)	Supposed Action
L	L	H	H	Open Circuit
L	H	H	H	Open Circuit
H	L	L	L	Motor turns clockwise
H	H	L	H	Motor turns counter clockwise

Table 3: Circuit's Truth Table after Inserting GAL22V10D

Input 1 (PWM signal)	Input 2 (Direction)	Output 1	Output 2	Action
L	L	Z	Z	Open Circuit
L	H	Z	Z	Open Circuit
H	L	H	L	Motor turns clockwise
H	H	L	H	Motor turns counter clockwise

Optocouplers: 2 PS2501-4 and 1 PS2501-2 were initially used to provide electrical isolations between the Arduino board with the GAL chip and the five H-Bridges with the motors, and thus protect both sides from getting affected just in case something goes wrong on one of these two sides.

Due to time constraints and several changes to the circuit design, the circuit was soldered on a prototyping board rather than on a printed PCB. Due to the difficulty in placing all components into the confined space of the prototyping board, the optocouplers were ultimately removed from the circuit.

H-Bridges TLE52052: Despite the fact that TLE52052 H-Bridges are made to deliver up to 5 A, they had a tendency to overheat. Their over temperature protection would have shut them down after 5 – 10 minutes of running the motors continuously at average to maximum voltages.

Therefore, they were replaced by 4A dual H-Bridges L298N below that still get high temperature after the motors continuously ran for a while, but they do not get as hot as TLE52052.

L298N H-Bridges: These chips can stand up to 4A current going through them, require in total 6 inputs and produce 4 outputs for 2 motors. They can receive signals straight from the Arduino board and, with the right input combination according to their provided truth table as shown below in **Error! Reference source not found.**, turn directions consequently to given commands. There are 3 of them in the final design.

Table 4: 4 A Dual H-Bridge L298N's Original Truth Table

Enable (PWM)	Input 1 (Positive Input)	Input 2 (Negative Input)	Action
L	Z	Z	Power Off
H	L	L	Brake
H	H	L	Turns clockwise fast
H	L	H	Turns counter clockwise fast
H	H	H	Brake
P (Pulse)	H	L	Turns clockwise slow
P (Pulse)	L	H	Turns counter clockwise slow

GAL22V10D chip: As stated above, the new L298N can receive signals straight from the Arduino board; however, they require 3 inputs for each motor (instead of 2 like other H-Bridges that were considered), two of which determine the directions of the motors. However, the Arduino board does not have enough input pins for all 5 motors. So, the GAL22V10D was used to form a new truth table as shown below in Table 5. The new logic requires only 2 required inputs for each motor: 1 PWM input and 1 direction input. The truth table for the combined system is shown in Table 6.

Table 5: GAL22V10D's Truth Table for L298N

Input 1 (Direction)	Output 1 (Positive Input)	Output 2 (Negative Input)	Action
L	H	L	Turns clockwise
H	L	H	Turns counter clockwise

Table 6: Circuit's Truth Table after Putting New Code in GAL22V10D

Input 1 (PWM)	Input 2 (Direction)	First Output from 2 (Positive Input)	Second Output from 2 (Negative Input)	Action
L	Z	Z	Z	Power Off

H	L	H	L	Turns clockwise fast
H	H	L	H	Turns counter clockwise fast
P (Pulse)	H	L	H	Turns counter clockwise slow
P (Pulse)	L	H	L	Turns clockwise slow

Arduino board start-up: Each time the battery connected to the board is turned on, the Arduino board supplies the highest possible voltage to all motors for a brief period of time while the code initializes. Although, the period is approximately one second, this caused a current surge which presented a risk to the circuit and ultimately damaged the Arduino board. This problem was rectified by connecting the Arduino in parallel with the circuit board, and by implementing a 5V 1A regulator to protect the logic chip.

5V 1A voltage regulator: This regulator can stand up to a current of 1A and it converts the voltage of 11.1V coming from the battery to 5V and supplies the logic chip and the logic voltage required on the 3 L298N H-Bridges.

The final circuit board design only contains four main components along with some supporting resistors, capacitors and diodes:

- One 20C 3 cell 1350 mAh 11.1 V Lithium Polymer battery
- One 5V 1A voltage regulator
- One generic array logic GAL22V10D
- Three 4 A dual H-Bridges

Figure 16 depicts a schematic of the implemented circuit board.

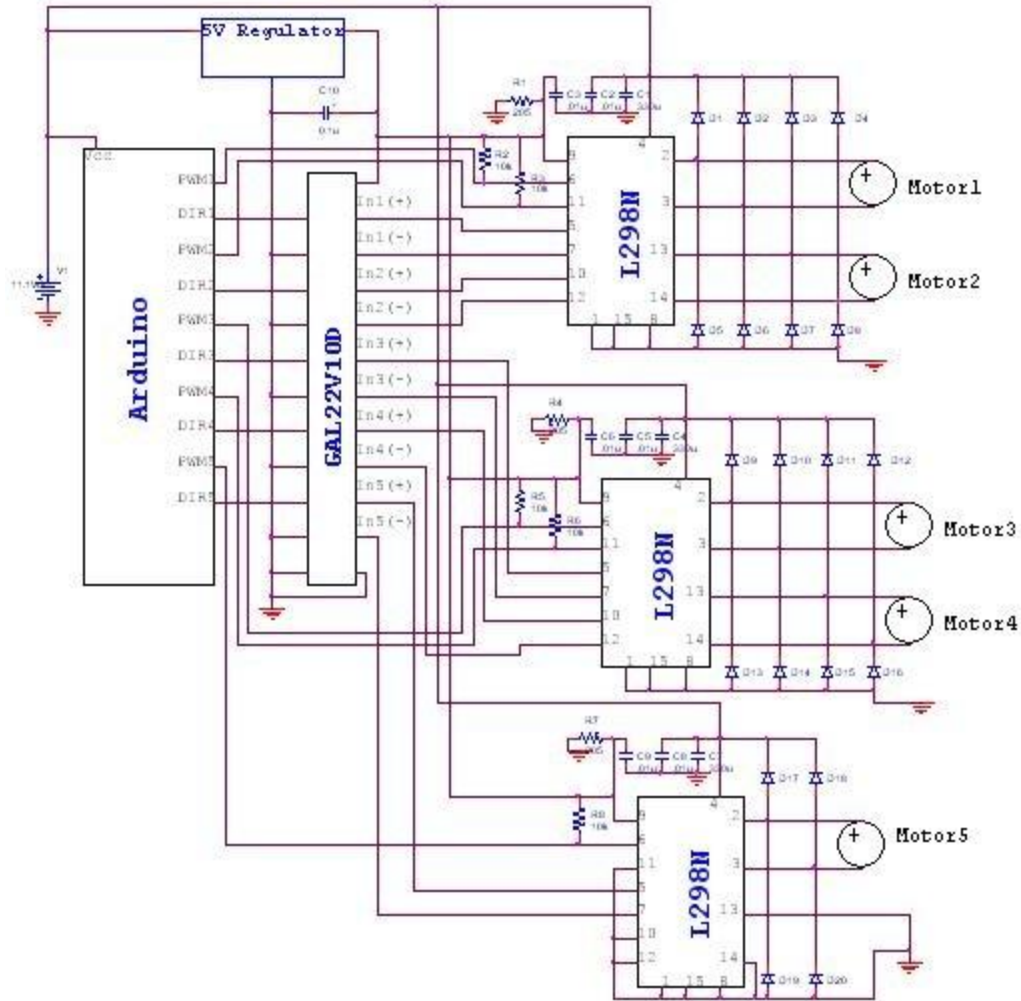


Figure 16: Final circuit board schematic

7. Results and Conclusions

Test flights of the blimp were incredibly useful in determining both the strengths and flaws of the overall project design. The results of one test of the position tracking control system are shown below. Figure 17 shows the coordinates of the blimp over time as it attempts to reach a commanded position of (1,1,1). As the figure shows, the blimp does not reach the final position with no steady state error, however the x and y positions do come quite close. Due to the buoyancy of the balloon, and the relatively limited authority of the z control motor, fine control of the vertical positions was difficult to achieve. The blimp was also highly responsive to disturbances in the air currents in the room; even though the air vents in the Vicon space were closed, drafts in the room had significant affect on the motion of the blimp.

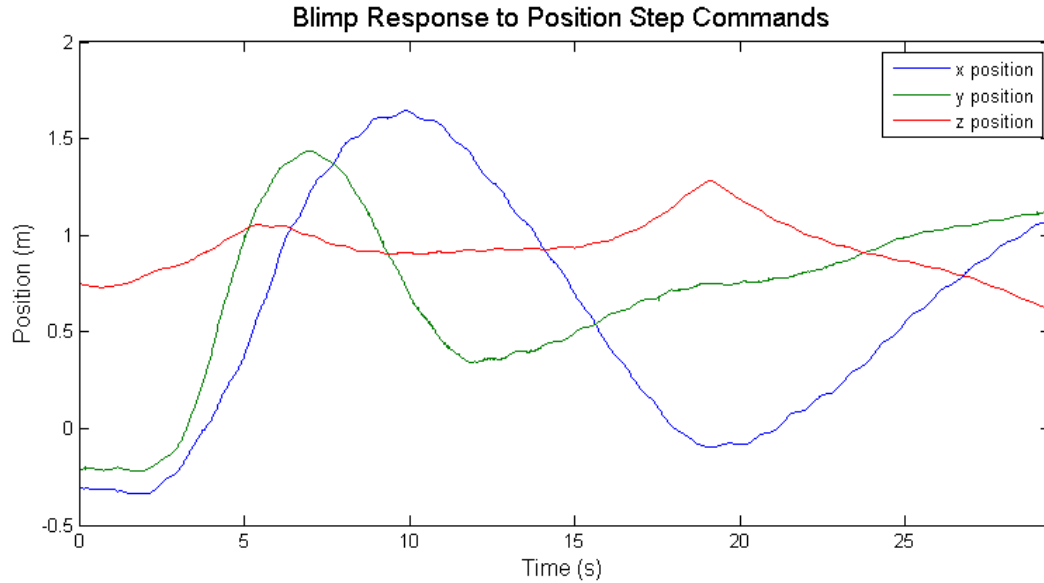


Figure 17. Position tracking of blimp system

As can be seen in the figure, the blimp experiences significant overshoot, which was not indicated in simulation. This is due to the fact that the simulated and carefully design control system was modified to achieve results in the laboratory setting. The motor actuators that are used to translate the blimp have a hard lower limit at which they may operate; below an input voltage of about 1 V they do not fire due to stiction. In order to guard against instability in the system, it was initially desired to design a controller with low control effort, which would also be useful in avoiding overshoot. When this controller design was implemented using the physical system, however, the control outputs to the blimp were all below the threshold created by friction; essentially, the selected actuators were not able to provide the fine control required by the system design. To compensate for this problem, the MATLAB controller was manually turned to output higher voltages, which significantly increased overshoot in the system, but did results in effective firings of the motors. The output control voltages for the above step command test are shown in Figure 18.

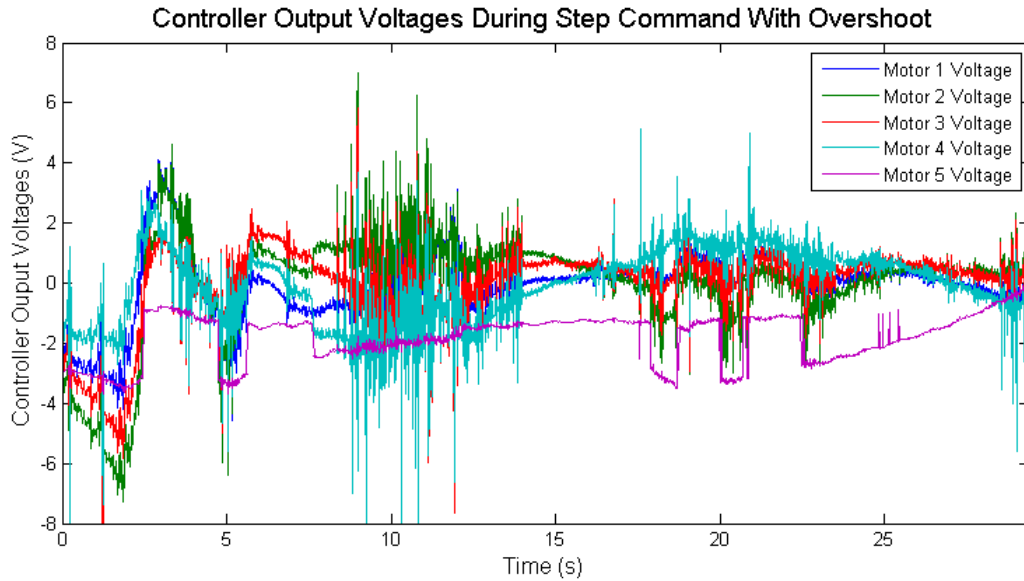


Figure 18. Controller output voltages for position step commands in three directions.

Further testing of the system produced similar results. Figure 19 shows the response of the blimp to a step position command of (1,-1,1).

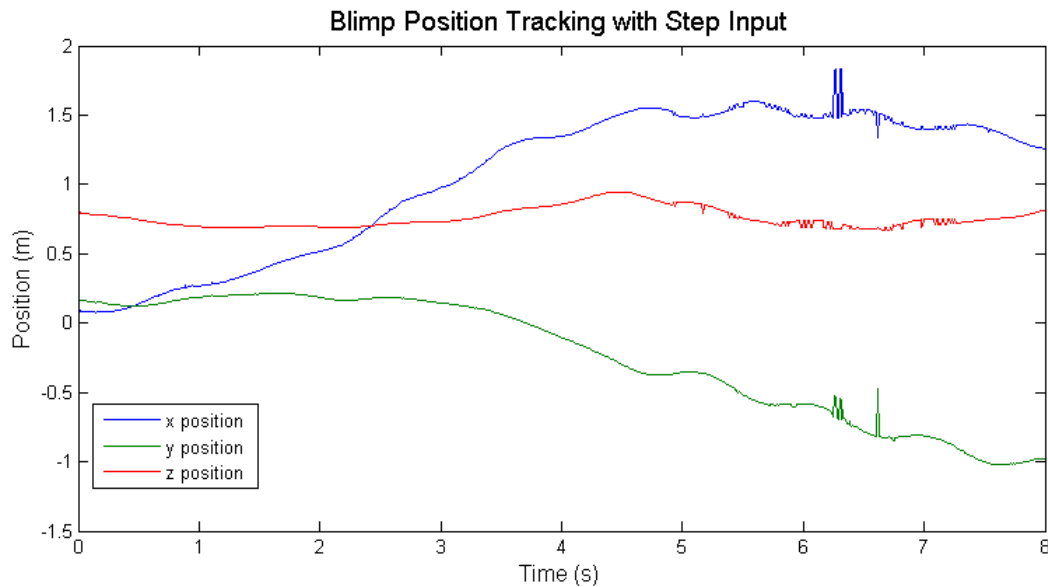


Figure 19. Blimp Step Test Results

The blimp controller also includes provisions for control of the heading angle, though this proved to be somewhat more problematic than even position control. The response of the system to a step command of pi radians is shown below. In order to produce the results below, the motors fired strongly for a short period of time, producing overshoot that was difficult for the

system to recover from. In order to remedy this problem, future commands will be implemented in smaller increments (way point tracking).

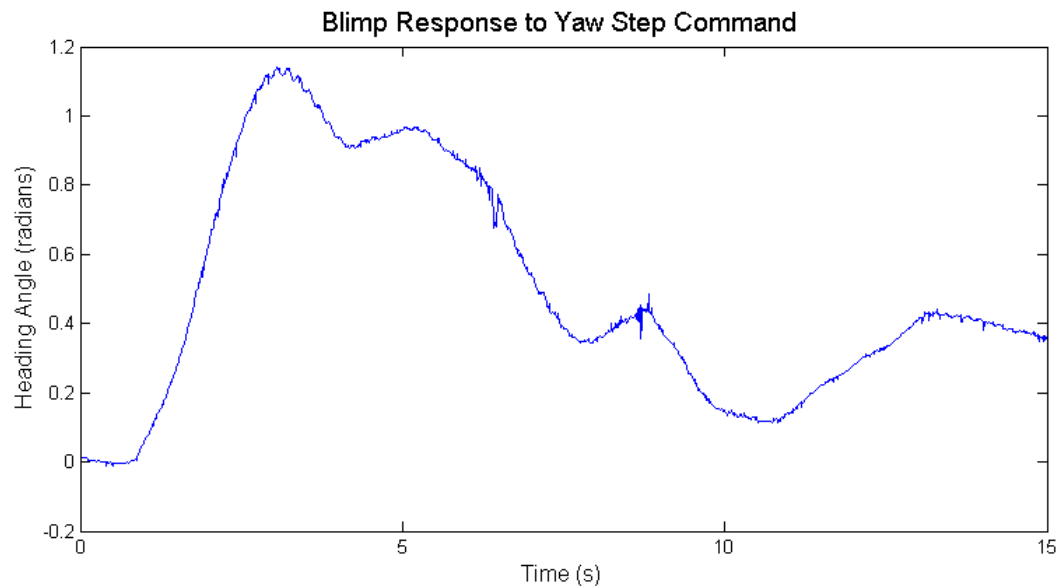


Figure 20. Response of Blimp to Yaw Step Command of Pi Radians

It is clear from testing of the system that many improvements could be made to the blimp. The most critical changes should be made to the actuator/controller system. As the above tests indicated, merely saturating the outputs of the motors in simulation did not accurately model what the true behavior of the motors would be. If fine control of the system is desired, the actuators must be able to supply lower thrust levels; improvements to the system could be made by physically changing the motors used on the blimp. It would also be possible to use patches in the controller code to solve the stiction issue; two proposed approaches are to pulse the motors above their stiction level (effectively introducing a second level of pulse width control) or to add a small sinusoidal signal on top of the pulse width output so as to continually create motion in the actuators and avoid static effects.

It is also clear from the noise in the control outputs that the control system itself requires refinement. Though the derivatives of position (used to calculate the proportional full state feedback) were filtered using a second order Butterworth filter, significant noise to the system is fed into the controller, which results in choppy outputs from the actuators.

Another refinement that will be made to the system involves finalizing and improving the blimp envelope. The final envelope which was used for testing is the balloon used by the previous 449 blimp team. This envelope was much larger than initially desired, but allowed easy attachment of the gondola. Reducing the size of the blimp while maintaining a sturdy design will be another area of future work. Finally, in order to continue reducing the weight of the system (and thus the volume) it is desired to order a printed circuit board.

8. References

8.1 Bibliography

1. Bestaoui, Yasmina and Hamel, Tarek. *Dynamic Modeling of Small Autonomous Blimps. Methods and Models in Automation and Robotics*, Miedzyzdroje, Pl, Aug. 2000, vol. 2, pp 579 – 584.
2. Dorf, Richard C. and Bishop, Robert H. *Modern Control Systems*. Pearson Prentice-Hall, Inc., Upper Saddle River, NJ, 2005.
3. Heemstra, Brian. *Linear Quadratic Methods Applied to Quadrotor Control*. Department of Aeronautics & Astronautics, University of Washington, 2010.
4. Hughes, Kyle, et al. *Distributed Space Systems Laboratory Blimp*. Department of Aeronautics & Astronautics, University of Washington, 2008.
5. Lewis, Frank L., and Stevens, Brian L. *Aircraft Control and Simulation*. Wiley, Hoboken, NJ, 2003.
6. Loo, van de Jasper. *Formation Flight of Two Autonomous Blimps: The Atalanta Wingman Project*. Master thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2007.
7. Nise, Norman S. *Control Systems Engineering*. Wiley, New York, 2007.
8. Ogata, Katsuhiko. *State Space Analysis of Control Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1967.
9. Schutter B. De, *Minimal State-Space Realization in Linear System Theory: an Overview*, Control Laboratory, Faculty of Information Technology and Systems, Delft University of Technology, Netherlands 30 January 2000

8.2 Use of References

1. Paper detailing the model determination portion of a project attempting a similar objective: autonomous waypoint tracking of a blimp vehicle. This was also used to validate our plant model with other research projects.
2. This textbook provided a comprehensive discussion for designing the state estimator.
3. Mr. Heemstra, a former Masters student in the Department of Aeronautics and Astronautics wrote his thesis on controlling a quadrotor vehicle for the DSSL. As the sensors for that project are identical to those used in for the blimp system, his thesis has been referenced over the course of the control design.

4. Mr. Hughes, a current Masters student in the Department of Aeronautics and Astronautics developed a similar blimp vehicle for this course in 2008. The final milestone report for that project has been a referenced over the course of the project.
5. This text book derived the transform matrices between body and inertial reference frames for vehicles in flight.
6. Thesis paper on similar project for formation flying blimp vehicles. This was used in validate our plant model with other research projects.
7. This was the textbook for our Introduction to Control Systems course, and provided the linearization procedure for state space representations, and the explicit equations for the controllability and observability matrices.
8. Used this reference to better understand the formal definitions of observability and controllability and the derivations behind the corresponding matrices. This source has been used throughout the course due the depth in its discussion of state space control methods.
9. This paper provided an overall discussion of the methods behind minimal realization of linear systems, and will be used when analyzing other operating points.

9. Appendix

9.1 Arduino Code

```
#define MOTOR1 10
#define DMOTOR1 12
#define MOTOR2 5
#define DMOTOR2 4
#define MOTOR3 3
#define DMOTOR3 2
#define MOTOR4 6
#define DMOTOR4 7
#define MOTOR5 9
#define DMOTOR5 8
#define LED 13
byte b[10];
byte key=255;

void setup()
{
  pinMode(MOTOR1,OUTPUT);
  pinMode(DMOTOR1,OUTPUT);
  pinMode(MOTOR2,OUTPUT);
  pinMode(DMOTOR2,OUTPUT);
  pinMode(MOTOR3,OUTPUT);
  pinMode(DMOTOR3,OUTPUT);
  pinMode(MOTOR4,OUTPUT);
  pinMode(DMOTOR4,OUTPUT);
```

```

pinMode(MOTOR5,OUTPUT);
pinMode(DMOTOR5,OUTPUT);

pinMode(LED,OUTPUT);
Serial.begin(115200);

}

void loop()
{
digitalWrite(LED,HIGH);
if(Serial.available()) {

    if(Serial.read() == key) {
        delay(5);
        for(int i = 0; i < 10; i++) {
            b[i] = Serial.read();
        }
        Serial.flush();
    }
}

//set the directions
if (b[5]==1) {
    digitalWrite(DMOTOR1, HIGH);
    digitalWrite(LED, LOW);
}
else {
    digitalWrite(DMOTOR1, LOW);
}

if (b[6]==1) {
    digitalWrite(DMOTOR2, HIGH);
}
else {
    digitalWrite(DMOTOR2, LOW);
}
if (b[7]==0) { //motors 3 and 4 have directions reversed?
    digitalWrite(DMOTOR3, HIGH);
}
else {
    digitalWrite(DMOTOR3, LOW);
}
if (b[8]==0) {
    digitalWrite(DMOTOR4, HIGH);
}
else {
    digitalWrite(DMOTOR4, LOW);
}
if (b[9]==1) {
    digitalWrite(DMOTOR5, HIGH);
}
else {
    digitalWrite(DMOTOR5, LOW);
}
}

```



```

//write the pwm commands
analogWrite(MOTOR1,b[0]);
analogWrite(MOTOR2,b[1]);
analogWrite(MOTOR3,b[2]);
analogWrite(MOTOR4,b[3]);
analogWrite(MOTOR5,b[4]);

}

```

9.2 GAL Code

```

//Title: Convert for H-Bridges
module logicGate(B5O2,B5O1,B4O2,B4O1,B3O2,B3O1,
    B2O2,B2O1,B1O2,B1O1,B5,B4,B3,B2,B1);
input B1/*synthesis LOC="2"*/,
    B2/*synthesis LOC="4"*/,
    B3/*synthesis LOC="6"*/,
    B4/*synthesis LOC="8"*/,
    B5/*synthesis LOC="10"*/;
output B1O1/*synthesis LOC="23"*/,
    B1O2/*synthesis LOC="22"*/,
    B2O1/*synthesis LOC="21"*/,
    B2O2/*synthesis LOC="20"*/,
    B3O1/*synthesis LOC="19"*/,
    B3O2/*synthesis LOC="18"*/,
    B4O1/*synthesis LOC="17"*/,
    B4O2/*synthesis LOC="16"*/,
    B5O1/*synthesis LOC="15"*/,
    B5O2/*synthesis LOC="14"*/;
    not not1(B1O1,B1);
buf buf1(B1O2,B1);

not not2(B2O1,B2);
buf buf2(B2O2,B2);

not not3(B3O1,B3);
buf buf3(B3O2,B3);

not not4(B4O1,B4);
buf buf4(B4O2,B4);

not not5(B5O1,B5);
buf buf5(B5O2,B5);

endmodule

```

9.3 MATLAB Code

9.3.1 Blimp Control GUI

```
function varargout = blimpGUI(varargin)
%BLIMPGUI M-file for blimpGUI.fig
%   BLIMPGUI, by itself, creates a new BLIMPGUI or raises the existing
%   singleton*.
%
%   H = BLIMPGUI returns the handle to a new BLIMPGUI or the handle to
%   the existing singleton*.
%
%   BLIMPGUI('Property','Value',...) creates a new BLIMPGUI using the
%   given property value pairs. Unrecognized properties are passed via
%   varargin to blimpGUI_OpeningFcn. This calling syntax produces a
%   warning when there is an existing singleton*.
%
%   BLIMPGUI('CALLBACK') and BLIMPGUI('CALLBACK',hObject,...) call the
%   local function named CALLBACK in BLIMPGUI.M with the given input
%   arguments.
%
%   *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only one
%   instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help blimpGUI

% Last Modified by GUIDE v2.5 26-May-2010 20:29:18

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',    mfilename, ...
    'gui_Singleton', gui_Singleton, ...
    'gui_OpeningFcn', @blimpGUI_OpeningFcn, ...
    'gui_OutputFcn', @blimpGUI_OutputFcn, ...
    'gui_LayoutFcn', [], ...
    'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before blimpGUI is made visible.
function blimpGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   unrecognized PropertyName/PropertyValue pairs from the
%            command line (see VARARGIN)
```

```

% Choose default command line output for blimpGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes blimpGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = blimpGUI_OutputFcn(hObject, eventdata, handles)
% varargout cell array for returning output args (see VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in start.
function start_Callback(hObject, eventdata, handles)
% hObject handle to start (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
start = 1; %if the start button is pressed, change value of 'start' and
assignin('base','start',start); %begin vicon data gathering while loop

% --- Executes on button press in update.
function update_Callback(hObject, eventdata, handles)
% hObject handle to update (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
xpos = str2double(get(handles.xpos,'String')); %get value of xpos and make
assignin('base','xpos',xpos) %it readable from workspace
ypos = str2double(get(handles.ypos,'String'));
assignin('base','ypos',ypos)
zpos = str2double(get(handles.zpos,'String'));
assignin('base','zpos',zpos)
yaw = str2double(get(handles.yaw,'String'));
assignin('base','yaw',yaw)

% --- Executes on button press in stop.
function stop_Callback(hObject, eventdata, handles)
% hObject handle to stop (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
stop = 1;
assignin('base','stop',stop);

```

```

function xpos_Callback(hObject, eventdata, handles)
% hObject    handle to xpos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of xpos as text
%        str2double(get(hObject,'String')) returns contents of xpos as a double

% --- Executes during object creation, after setting all properties.
function xpos_CreateFcn(hObject, eventdata, handles)
% hObject    handle to xpos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%        See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

```

```

function ypos_Callback(hObject, eventdata, handles)
% hObject    handle to ypos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ypos as text
%        str2double(get(hObject,'String')) returns contents of ypos as a double

```

```

% --- Executes during object creation, after setting all properties.
function ypos_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ypos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%        See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

```

```

function zpos_Callback(hObject, eventdata, handles)
% hObject    handle to zpos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of zpos as text
%        str2double(get(hObject,'String')) returns contents of zpos as a double

```

```

% --- Executes during object creation, after setting all properties.

```

```

function zpos_CreateFcn(hObject, eventdata, handles)
% hObject    handle to zpos (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

```

```

function yaw_Callback(hObject, eventdata, handles)
% hObject    handle to yaw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of yaw as text
%       str2double(get(hObject,'String')) returns contents of yaw as a double

```

```

% --- Executes during object creation, after setting all properties.
function yaw_CreateFcn(hObject, eventdata, handles)
% hObject    handle to yaw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

```

```

% --- Executes on button press in session.
function session_Callback(hObject, eventdata, handles)
% hObject    handle to session (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
session = 1;
assignin('base','session',session);

```

9.3.2 Controller Code

```

%At some point we could have it call the controllers from the workspace,
%but I think it might be easier just to have them all in this file

% %Assume that Vicon data can be parsed into Cg position and heading angle
%Vector y is gotten from client for loop

blimpGUI;
int_saturation=3;

```

```

max_n=255;
% s=serial('COM8');
% fopen(s);
y=zeros(1,4);
y2=zeros(1,4);
y3=zeros(1,4);
dt=.01; %set some sort of pause here
forward_coefficient=.546;
xhat=zeros(1,8);

afilt=[1 -1.433 .5615];
bfilt=[.0321 .0642 .0321];
r=zeros(1,4);
%reference from GUI
new_ref_error=zeros(1,4);
ref_error=zeros(1,4);
integralState=zeros(1,4);
traprule1=zeros(1,8);
traprule2=zeros(1,4);
n=zeros(5,1);
dir=n;

xf4=0;
xf3=0;
xf2=0;
zf4=0;
zf3=0;
zf2=0;
yf4=0;
yf3=0;
yf2=0;
yawf4=0;
yawf3=0;
yawf2=0;

v_4=zeros(1,4);
v_3=v_4;
v_2= v_4;
v_inertial=v_2;

u=zeros(1,5);
loadlibrary('CG.dll', 'CG.h');
calllib ('CG', 'Connect', 'localhost')
calllib('CG', 'EnableMarkerData');
calllib('CG', 'EnableSegmentData');

SubName='Blimp';
IVal=[0.0,0.0,0.0];
posx=(0);
posxyz = libpointer('doublePtrPtr',IVal);

IValOc=0;
Obpoint = libpointer('uint32Ptr',IValOc);
count=0;
start = 0;

```

```

stop = 0;
session = 0;
xpos = 0;
ypos = 0;
zpos = 0;
yaw = 0;

for number=1:15
    if(session==1)
        fwrite(s,uint8([255 0 0 0 0 0 0 0 0 0]));
        break
    end

    while(start==0)
        pause(.1)
        if(session==1)
            break
        elseif (start ==1)
            stop=0;
            break
        end
    end
end

for count=1:500000 %since it doesn't like while loops
    t(count)=count;
    if(stop==1)
        start=0;
        fwrite(s,uint8([255 0 0 0 0 0 0 0 0 0]));
        break
    end
    for i=1:5
        mkrname = calllib('CG', 'GetMarkerName', SubName, i);
        calllib('CG', 'GetFrame');
        calllib('CG','GetGlobalMarkerTranslation',SubName,mkrname,posxyz,Obpoint);
        column=i*3;
        Markers(count,column-2:column)=posxyz.value;
    end
    end
    %parse Marker data into Cg and heading angle
    %first get yaw
    if(Markers(count,1)~=0&&Markers(count,13)~=0)
        y(4)=atan2((Markers(count,2)-Markers(count,14)),(Markers(count,1)-Markers(count,13)));
    elseif(Markers(count,1)~=0&&Markers(count,7)~=0)
        y(4)=atan2((Markers(count,2)-Markers(count,8)),(Markers(count,1)-Markers(count,7)));
    elseif(Markers(count,13)~=0&&Markers(count,7)~=0)
        y(4)=atan2((Markers(count,14)-Markers(count,8)),(Markers(count,13)-Markers(count,7)));
    elseif(Markers(count,4)~=0&&Markers(count,10)~=0)
        y(4)=atan2(Markers(count,5)-Markers(count,11),Markers(count,4)-Markers(count,10))-pi/2;
    end

    %then get position of Cg
    if(Markers(count,1)~=0)
        y(1)=Markers(count,1)-190.5*cos(y(4));
        y(2)=Markers(count,2)-190.5*sin(y(4));
        y(3)=Markers(count,3);

```

```

elseif(Markers(count,4)~=0)
    y(1)=Markers(count,4)-190.5*cos(y(4)+pi/2);
    y(2)=Markers(count,5)-190.5*sin(y(4)+pi/2);
    y(3)=Markers(count,6);
elseif(Markers(count,7)~=0)
    y(1)=Markers(count,7)+190.5*cos(y(4));
    y(2)=Markers(count,8)+190.5*sin(y(4));
    y(3)=Markers(count,9);
elseif(Markers(count,10)~=0)
    y(1)=Markers(count,10)+190.5*cos(y(4)+pi/2);
    y(2)=Markers(count,11)+190.5*sin(y(4)+pi/2);
    y(3)=Markers(count,12);
elseif(Markers(count,13)~=0)
    y(1)=Markers(count,13)-165.1*cos(y(4));
    y(2)=Markers(count,14)-165.1*sin(y(4));
    y(3)=Markers(count,15);
end
ysave(count,1:4)=y(1:4);
%y(4)=0;
y(1:3)=y(1:3)/1000;
if(y(4)>-pi && y(4)<-3*pi/4 )

```

```

    K1=[0.5521 -1.0383 -0.0554 -0.0677 1.3518 -0.0344 1.9189 2.2304 0.1674 -0.2808 -0.0212
0.5277
    1.0740 0.1291 -0.0456 -1.2853 0.1755 -0.0204 1.8268 2.1732 0.2866 -0.0091 -0.0243
0.4916
    0.1128 0.5469 -0.0843 -0.2151 -0.9095 -0.0518 1.4999 1.9645 0.0832 0.0849 -0.0337
0.3666
    -0.4091 -0.6205 -0.0940 1.0025 0.2668 -0.0657 1.5921 2.0217 -0.0360 -0.1868 -0.0306
0.4028
    0.1440 0.0170 1.6476 -0.1100 0.0266 2.3090 -0.0783 -0.0532 0.0364 0.0004 0.3471 -
0.0277];
    op(count)=1;

```

```

elseif(y(4)>-3*pi/4 && y(4)<-pi/2)
    %here
    K1=[0.4950 -0.4748 0.0517 0.1606 1.0150 0.0407 1.9063 2.2248 0.0724 -0.1312 0.0123
0.5217
    0.1006 0.6986 -0.0493 -1.0566 -0.1186 -0.0269 1.8555 2.1942 -0.0080 0.1411 -0.0223
0.5011
    -0.6787 0.2840 -0.0978 -0.0116 -1.0765 -0.0610 1.9374 2.2421 -0.1445 0.0539 -0.0373
0.5351
    -0.2842 -0.8893 0.0032 1.2057 0.0571 0.0067 1.9882 2.2726 -0.0641 -0.2184 -0.0027
0.5557
    0.1405 -0.1337 1.6988 0.0410 0.1267 2.3411 -0.0148 -0.0047 0.0423 -0.0485 0.3652 -
0.0097];
    op(count)=2;

```

```

elseif(y(4)>-pi/2 && y(4)<-pi/4)
    %yeah
    K1=[0.6053 0.6408 0.1683 -0.2301 1.1130 0.0975 1.8162 2.1659 0.1191 0.1988 0.0710
0.4876

```



```

-0.2018  0.9580  0.1532 -1.2194  0.1121  0.0871  1.9368  2.2457 -0.0275  0.2528  0.0666
0.5316
-0.7658 -0.2558  0.0225 -0.0247 -1.1367  0.0052  1.8084  2.1725 -0.1671 -0.0373  0.0165
0.4753
0.0413 -0.5729  0.0376  0.9646 -0.1359  0.0155  1.6879  2.0927 -0.0205 -0.0913  0.0209
0.4314
0.1419  0.1700  1.7028 -0.0607  0.1277  2.3381  0.1017  0.0547  0.0482  0.0670  0.3724
0.0475];
    op(count)=3;

    elseif(y(4)>-pi/4 && y(4)<0)
        %is
        K1=[0.6382  0.4683 -0.0244  0.2663  1.0594 -0.0283  1.3537  1.8762  0.1880  0.0514 -0.0002
0.3075
-0.5352  0.0134 -0.1534 -0.9242 -0.1196 -0.1122  1.2635  1.8193 -0.0875 -0.0438 -0.0477
0.2712
-0.0669 -1.1493 -0.0709  0.2344 -1.3097 -0.0548  1.6357  2.0551  0.0326 -0.3186 -0.0195
0.4144
1.1065 -0.6944  0.0580  1.4249 -0.1307  0.0290  1.7259  2.1120  0.3081 -0.2233  0.0280
0.4507
0.2916 -0.0259  1.6515  0.1891  0.0569  2.2981 -0.0612 -0.0573  0.1013 -0.0075  0.3586 -
0.0097];
    op(count)=4;

    elseif(y(4)>0 && y(4)<pi/4)
        %do something
        K1=[-0.7565  1.3469  0.4307 -0.0801  1.5492  0.1884  2.9792  2.7844 -0.2780  0.4022  0.2401
1.0117
-1.7035 -0.7414  0.5521 -1.7136 -0.0321  0.2298  3.1257  2.8584 -0.5758 -0.2333  0.3267
1.0797
-0.2083 -1.3663  0.3416 -0.3674 -1.3915  0.1286  2.7918  2.6973 -0.1930 -0.3881  0.2188
0.9111
0.7387  0.7220  0.2202  1.2661  0.1899  0.0873  2.6454  2.6233  0.1048  0.2474  0.1322
0.8431
-0.2582  0.0197  3.4268 -0.0893  0.0659  3.2906  0.4719  0.1984 -0.1595 -0.0179  1.1068
0.2745];
    op(count)=5;

    elseif(y(4)>pi/4 && y(4)<pi/2)
        %and something else
        K1=[-1.0722  0.7140  0.2383  0.1305  1.4158  0.1029  3.3931  2.9684 -0.2706  0.2326  0.1353
1.2373
-0.2853 -1.4534  0.3739 -1.5178 -0.1560  0.1652  3.3606  2.9524 -0.0464 -0.4401  0.2072
1.2226
1.0313 -0.6829  0.3359 -0.1187 -1.3919  0.1459  3.3775  2.9617  0.2486 -0.2234  0.1890
1.2295
0.2443  1.4845  0.2003  1.5297  0.1800  0.0837  3.4100  2.9777  0.0244  0.4493  0.1171
1.2443
0.0480 -0.2166  2.9834 -0.0865 -0.0594  3.0867  0.3308  0.1433  0.0239 -0.1098  0.8761
0.1875];
    op(count)=6;

    elseif(y(4)>pi/2 && y(4)<3*pi/4)
        %almost
        K1=[-1.1032 -1.1694  0.1179 -0.3269  1.5334  0.0357  2.8273  2.7181 -0.2564 -0.4547  0.0883
0.9186

```

```

0.7967 -1.7370 0.3827 -1.7259 -0.0079 0.1634 3.2890 2.9352 0.3279 -0.5919 0.2210
1.1608
1.9462 0.2144 0.3437 -0.2528 -1.7128 0.1454 3.0122 2.7911 0.7181 -0.0149 0.1990
1.0382
0.0464 0.7820 0.0789 1.1462 -0.1714 0.0178 2.5505 2.5740 0.1338 0.1223 0.0663
0.7961
0.4473 -0.1663 2.9685 -0.1591 -0.1735 3.0762 0.2668 0.1002 0.2149 -0.0838 0.8716
0.1715];
    op(count)=7;

else
    %the last K!
    K1=[-0.2193 -1.7931 0.1134 -0.2204 1.6655 0.0347 3.0356 2.8136 0.0357 -0.6142 0.0858
1.0303
1.9287 -0.8361 0.2898 -1.8039 0.0210 0.1241 3.3421 2.9509 0.7264 -0.3102 0.1672
1.2016
1.3736 0.9929 0.1293 -0.4276 -1.5132 0.0485 2.8323 2.6999 0.5789 0.2187 0.0828
0.9519
-0.7744 0.0359 -0.0471 1.1560 0.1314 -0.0409 2.5258 2.5626 -0.1118 -0.0853 0.0014
0.7806
0.4721 -0.0885 2.9024 -0.2364 -0.0558 3.0534 0.1138 0.0276 0.2168 -0.0513 0.8259
0.0941];
    op(count)=8;
end

%this one works best of the ones we actually can generate
% K1=[-0.3185 3.3188 -0.0078 -0.3374 5.7522 0.0576 0.9198 1.6365 -0.0544 0.6222 -
0.0292 0.1965
% -3.0552 0.1377 0.1749 -5.6122 0.1046 0.2583 0.6228 1.6067 -0.5238 0.0560
0.0064 0.0312
% 0.3368 -3.0148 -0.2205 0.2436 -5.5276 -0.2408 0.7339 1.5967 0.0955 -0.5241 -
0.0340 0.0642
% 3.0735 0.1663 -0.4031 5.5185 0.1200 -0.4415 1.0310 1.6264 0.5649 0.0421 -0.0696
0.2295
% -0.1081 0.6645 6.6622 -0.2904 0.6881 11.8124 -0.3466 -0.2532 0.0206 0.1134
1.1111 -0.1185];

% this one works best, so far

% This one! This one here!
% K1=[-0.3009 4.8667 1.0192 -0.2861 5.5499 0.6017 5.4947 2.4162 -0.0372 1.0878
0.3776 0.4378
% -3.9514 0.5573 0.9774 -4.7613 0.4568 0.6957 4.8896 5.2672 -0.8627 0.1421
0.3187 0.3789
% 0.2084 -4.6210 -0.6668 0.2419 -5.3727 -0.3968 -3.2533 1.2305 -0.0023 -1.0058 -
0.2448 -0.0698
% 3.8590 -0.3117 -0.6250 4.7171 -0.2796 -0.4908 -2.6483 -1.6205 0.8231 -0.0600 -
0.1859 -0.0109
% -0.5215 0.0197 7.4442 -0.4307 -0.1344 12.5309 1.1455 0.5540 -0.2031 0.1316
1.2797 0.4269];

```

```

KI=K1(:,9:12);
KP=K1(:,1:8);

% Estimate States / Proportional Control
if(count<4)
    v_inertial =[0 0 0 0]';
else
    v_inertial=(3*y'-4*y2'+y3')/(2*dt);

end
yaw_dot=v_inertial(4);
%coriolis=[-1*v_inertial(4)*y(2) v_inertial(4)*y(1) 0 0]';
%v_body=v_inertial-coriolis;

% Undergo lengthy filtration process
xf1=v_inertial(1);
x_filt=filter(bfilt,afilt,[xf4 xf3 xf2 xf1]);
v_inertial_f(1)=x_filt(end);
xf4=xf3;
xf3=xf2;
xf2=xf1;

yf1=v_inertial(2);
y_filt=filter(bfilt,afilt,[yf4 yf3 yf2 yf1]);
v_inertial_f(2)=y_filt(end);
yf4=yf3;
yf3=yf2;
yf2=yf1;

zf1=v_inertial(3);
z_filt=filter(bfilt,afilt,[zf4 zf3 zf2 zf1]);
v_inertial_f(3)=z_filt(end);
zf4=zf3;
zf3=zf2;
zf2=zf1;

yawf1=v_inertial(4);
yaw_filt=filter(bfilt,afilt,[yawf4 yawf3 yawf2 yawf1]);
v_inertial_f(4)=yaw_filt(end);
yawf4=yawf3;
yawf3=yawf2;
yawf2=yawf1;

C_b2r=[cos(y(4)) sin(y(4)) 0
        -sin(y(4)) cos(y(4)) 0
        0 0 1];
v_body=C_b2r\ v_inertial_f(1:3)';
v_body(4)=v_inertial_f(4);
y3=y2;
y2=y;

xhat(1:3)=y(1:3);
xhat(7)=y(4);
xhat(4:6)=v_body(1:3);
xhat(8)=v_body(4);

```

```

u_prop=(KP*xhat)';
xhatplot(count,1:8)=xhat;

% Integral Control
r=[xpos ypos zpos yaw];
new_ref_error=r-y;

for i=1:4
    traprule2(i)=(new_ref_error(i)+ref_error(i))*dt*.5;
    integralState(i)=integralState(i)+traprule2(i);
    if(integralState(i)>int_saturation) %cap the integrator
        integralState(i)=int_saturation;
    elseif(integralState(i)<-1*int_saturation)
        integralState(i)=-1*int_saturation;
    end
end
ref_error=new_ref_error;
u_int=(KI*integralState)';
% Give Control Voltages to Arduino
u=u_int-u_prop;
usave(count,1:5)=u;

for m = 1:5
    if(u(m)>0)
        dir(m)=1;
        if(u(m)>6)
            u(m)=6;
            n(m)=max_n*(u(m)/12);
            n(m)=n(m)*(forward_coefficient);
            %n(m)=max_n;
        else
            n(m)=max_n*(u(m)/12);
            n(m)=n(m)*(forward_coefficient);
        end
    else
        dir(m)=0;
        if(u(m)<-6)
            u(m)=-6;
            n(m)=max_n*(u(m)/12);
            %n(m)=max_n;
        else
            n(m)=-max_n*(u(m)/12);
        end
    end
    if (n(m)<30&& n(m)>15)
        n(m)=30;
    elseif (n(m)<=15)
        n(m)=0;
    end
end

```

```

if (abs(ref_error(1))<=.1&&abs(ref_error(2))<=.1&&abs(ref_error(4)<=.25))
    Output=zeros(1,10);
    fwrite(s,uint8([255 Output]))
else
    n=n*2.5;
    for k=1:5;
        if n(k)>127
            n(k)=127;
        end
    end
    Output=[n',dir'];

    fwrite(s,uint8([255 Output]))
end

%Output
Outputsave(count,1:10)=Output;
pause(dt);

end
end
disp('Session Ended! Have a nice day!')
%Disconnect from the System and Unload the Library
calllib ('CG', 'Disconnect');
unloadlibrary CG
%fclose(s);

```

9.3.3 Linearized Model

```

clear all; close all; clc
Drag=.1; %random drag coefficient thing
rf=.3; %radius of motors from Cg
x=[0 0 0 0 0 7*pi/8 0]; %linearization point
k=[ 0 0 0 0 0]; %needed for linmod syntax but not used
[A,B,C,D]=linmod('Non_linear_u2x',x,k);
%linmod only makes valid A and B, so make C
C=zeros(4,8);
C(1,1)=1;
C(2,2)=1;
C(3,3)=1;
C(4,7)=1;

Alarge = [A,zeros(8,4);C,zeros(4,4)];
Blarge = [B;zeros(4,5)];
poles=1.5*[-2 -4 -3 -2.5 -6 -5.5 -7.5 -6+.2i -6-.2i -7 -8 -5];
K=place(Alarge,Blarge,poles)

KI=K(:,9:12);
KP=K(:,1:8);

```

9.3.4 State Space Model

```

% A_state=A+.1*(-1 + (2).*rand(8));

% B matrix
B=zeros(8,5);
B(4,:)=[0 -a/M 0 a/M 0];
B(5,:)=[a/M 0 -a/M 0 0];
B(6,5)=a/M;
B(8,:)=[a*rf/J a*rf/J a*rf/J a*rf/J 0];

% %C matrix
C=zeros(4,8);
C(1,1)=1;
C(2,2)=1;
C(3,3)=1;
C(4,7)=1;

D=zeros(4,5);

w=[0 0 0 Dr*Ux^2 Dr*Uy^2 Dr*Uz^2 0 0]';

M=ctrb(A,B);
rankct=rank(M);

% %Mo=obsv(A,C);
% %rankobs=rank(Mo);
% Clarge=eye(12);
% Clarge(1,1)=cos(tz);
% Clarge(2,1)=sin(tz);
% Clarge(2,2)=cos(tz);
% D=0;
%
% % %Controller Design
Alarge = [A,zeros(8,4);C,zeros(4,4)];
Blarge = [B;zeros(4,5)];

Mi=ctrb(Alarge,Blarge);
rankcti=rank(Mi);

%poles=1*[-2 -3 -10 -6 -4+1i -4-1i -11 -5+2i -5-2i -5 -9 -8];
%poles=.15*[-2 -4 -3 -2.5 -6 -5.5 -7.5 -6+.2i -6-.2i -7 -8 -5];
%poles=.25*[-2+1i -2-1i -3 -8-2i -8+2i -1 -4+1i -4-1i -7 -5+2i -5-2i -6];
K=place(Alarge,Blarge,poles);

KI=K(:,9:12);
KP=K(:,1:8);

poles_obs=real(poles)*5;

L=place(A',C',poles_obs(1:8));

% Sysc=ss(A,B,C,D)
% SYSD=c2d(Sysc,.1)
% [am,bm,cm,dm]=ssdata(SYSD)

```

