FLIGHT TESTING SMALL UAVS FOR AERODYNAMIC PARAMETER ESTIMATION

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Adam Chase

September 2013

COMMITTEE MEMBERSHIP

TITLE:	FLIGHT TESTING SMALL UAVS FOR DRAG POLAR ESTIMATION
AUTHOR:	Adam Chase
DATE SUBMITTED:	September 2013
COMMITTEE CHAIR:	Robert McDonald, Ph.D. Associate Professsor, Aerospace Engineering
COMMITTEE MEMBER:	Eric Mehiel, Ph.D. Associate Professor, Aerospace Engineering
COMMITTEE MEMBER:	Russell Westphal, Ph.D. Professor, Mechanical Engineering
COMMITTEE MEMBER:	Kurt Colvin, Ph.D. Professor, Industrial and Manufacturing Engineering

ABSTRACT

FLIGHT TESTING SMALL UAVS FOR AERODYNAMIC PARAMETER ESTIMATION

Adam Chase

ACKNOWLEDGMENTS

Thank you...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 Introduction

Aircraft designers often use model fits and "rules of thumb" to successfully complete designs, with many of these guidelines available in various design textbooks. [4],[5],[6] These models and practices have been established based on years of data analysis and validation that the designs perform as expected. However, the scope of these empirical design techniques is limited to the input of the regression model: an intelligent designer will not use a general aviation weight model for a transport category aircraft. To this end, there is a significant lack of small UAV-class guidelines for designing an aircraft.

Accurate estimations of a small-scale vehicle's lift and drag characteristics are extremely critical to the aircraft designer, and affect both point performance (turn rates, climb rates, stall speeds, etc.) and mission performance (range and endurance). Much of the prediction tools available in the classic lift and drag textbooks from Hoerner[7],[8] only apply to larger scale structures. In addition, drag prediction is extremely difficult on small vehicles, as some sources of drag are not easily modeled. For instance, actuator control horns and protruding screw heads are not typically included in the CFD analysis of aircraft. However, for small vehicles, these sources of "crud" drag can be a significant portion of the total vehicle's drag value.

One advantage of small UAVs is that they are fairly inexpensive, which makes building multiple fully-functioning prototypes a viable option. The authors chose to take advantage of this fact and develop a flight data acquisition system with a primary goal of measuring the lift and drag characteristics of a small UAV. This would enable vehicle designers to build a conceptual-design-level prototype, and both develop and validate predictive, regression-based models. The system would also allow designers to conduct quantitative trade studies, such as the trade between drag reduction technologies (wheel pants, retractable landing gear,

winglets, etc.) and the weight associated with them.

The authors also desired additional sensors to aid designers with more than just lift and drag characteristics. Some areas of interest are estimating stability and control derivatives, as well as possible control algorithm testing, in-flight thrust measurement, and payload integration capabilities. To accomplish these goals, sensors not directly necessary to lift and drag estimation were included in the overall system. The system was also designed in a manner that made it reconfigurable. This allows future designers to decide what combination of accuracy, risk, and sensing capabilities they need on a given flight test, and to then package the sensors in a manner that meets vehicle integration requirements.

For this paper, the authors chose to integrate the necessary sensors to estimate lift and drag forces, as well as those necessary for a basic INS, ambient temperature measurement, and servo and motor signals. The system will be validated by measuring the as-built drag polar of an R/C aircraft, as well as all other available states the system is capable of measuring.

2.0 Method

Some basic assumptions will apply throughout the modeling of dynamics in this paper. They are as follows:

1. The vehicle is a fixed mass.
2. Coriolis effects are negligible.
3. Thrust will be assumed to be 0.

Note that a stationary atmosphere is not assumed. The fixed mass assumption is consistent with the electric aircraft used to test the system. At the altitude and speed at which the vehicles will be tested, Coriolis effects can be ignored[9]. The zero-thrust assumption is in place to minimize drag error. Any error in a measured state will decrease the accuracy of the drag measurement, and the accuracy of a given drag measurement can be no better than the worst state measurement error. In-flight thrust is difficult to measure accurately, so a folding propeller will be used, and the motor will be turned off during data acquisition. This will allow the propeller to fold back, eliminating most of the wind-mill drag associated with a stalled propeller.

2.1 Reference Frames

For this thesis, the reference frames used will follow standard convention,[9] which will be repeated here for clarity.

North-East-Down (NED) Axes (x_{ned}, y_{ned}, z_{ned})

The NED axis system defines a local tangent plane on the Earth's surface, with the origin coinciding with the vehicle's center of gravity. The $\hat{i}$ vector points due north, the $\hat{j}$ vector

points due east, and the $\hat{k}$ vector points towards the center of the earth, in accordance with the right-hand rule. This coordinate system is vehicle carried, meaning the origin is fixed to the aircraft, but the axis directions are independent of vehicle orientation.

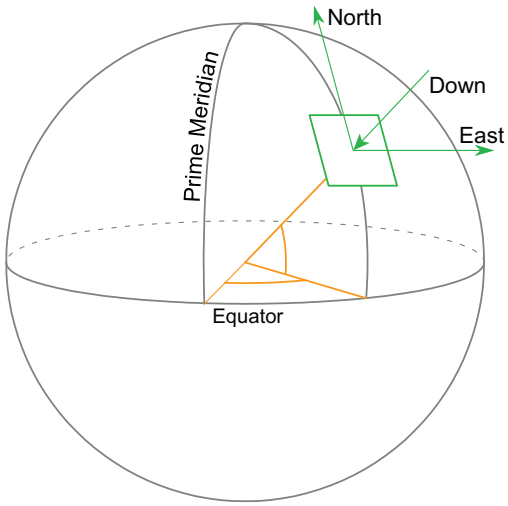


Figure 2.1: NED Frame of Reference[1]

Body Axes (x_b, y_b, z_b)

The body axis system has its origin at the vehicle's center of gravity, with the $\hat{i}$ direction pointing out the vehicle's nose, the $\hat{j}$ direction pointing out the right wing, and the $\hat{k}$ direction pointing out the belly of the aircraft, in accordance with the right-hand rule. This coordinate frame is fixed to the body, meaning the aircraft's spatial orientation does not change the direction of the axes.

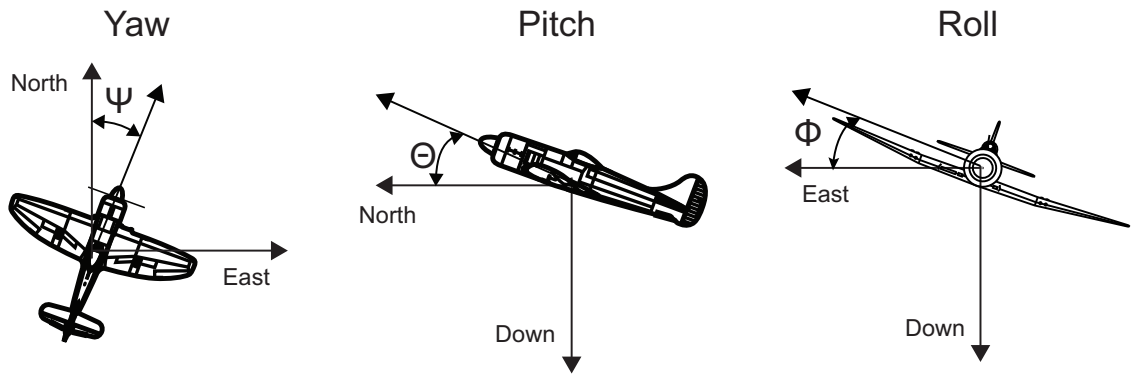


Figure 2.2: Body Axes Definition[2]

Stability Axes (x_s, y_s, z_s)

The stability axes are defined with its origin coinciding with the center of gravity of the vehicle. This axis system has essentially the same directions as the body axes, except rotated about the body axis $\hat{j}$ through an initial angle-of-attack, α_0. This inital angle-of-attack is defined at the beginning of a test maneuver and is then set for the remainder of the test, making it a body-fixed coordinate system. This system assumes no initial sideslip angle [10]. In Figure 2.3, only the $\vec{i}$ stability vector is shown, for clarity.

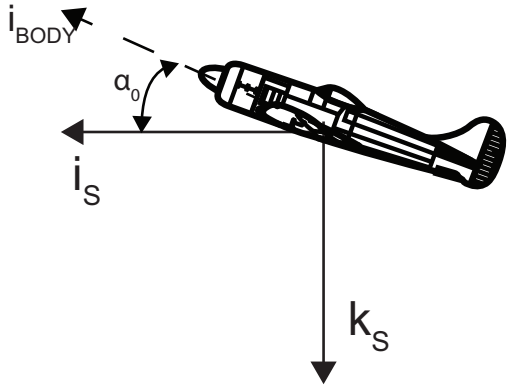


Figure 2.3: Stability Axes Definition

Wind Axes (x_w, y_w, z_w)

The wind axes are, again, a vehicle-carried coordinate system, meaning the origin of the wind axis also coincides with the center of gravity of the vehicle. However, the wind axes are not a body-fixed coordinate frame. The $\hat{i}$ direction points into the oncoming air, as seen from the vehicle. The $\hat{k}$ direction lies in the x-z plane of the body reference frame. The $\hat{j}$ direction is then defined to be out the right side of the vehicle, in order to follow the right hand rule.

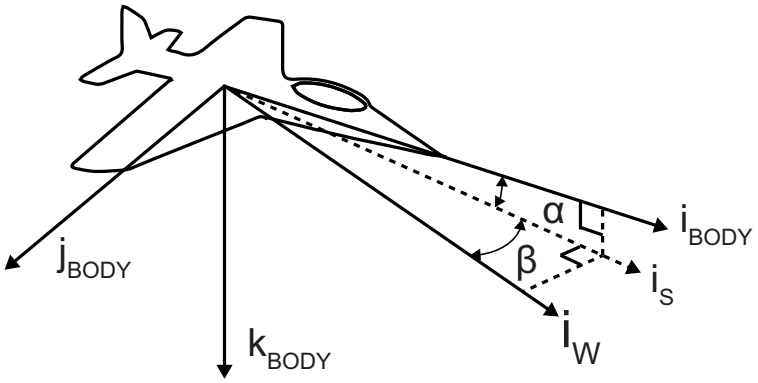


Figure 2.4: Wind Axes Definition

In Figure 2.4, only the $\vec{i}$ wind vector is shown, for clarity.

2.2 Equations of Motion

Newton's 2nd Law of Motion states

$$\Sigma\vec{F} = \frac{d}{dt}(m\vec{V}) \tag{2.2.1}$$

where $\Sigma\vec{F}$ is the sum of all applied forces, $\vec{m}$ is the mass of the vehicle, and $\vec{V}$ is the vehicle's velocity. Using the fixed mass assumption, this reduces to

$$\Sigma\vec{F} = m\frac{d\vec{V}}{dt} \tag{2.2.2}$$

$$= m\vec{a} \tag{2.2.3}$$

The applied forces on the vehicle for drag polar estimation are

$$\Sigma\vec{F} = \vec{F}_A + \vec{F}_G + \vec{F}_T \tag{2.2.4}$$

where $\vec{F_A}$ accounts for all aerodynamic forces acting on the vehicle, $\vec{F_G}$ is the force due to gravity, and $\vec{F_T}$ accounts for forces from the propulsion system, which are assumed to be zero. Aerodynamic forces can be described in the wind reference frame. In general, they are defined as

$$\vec{F}_{A_w} = D\hat{i}_w + Y\hat{j}_w + L\hat{k}_w \tag{2.2.5}$$

where D is drag force, Y is side force, and L is lift force.
The gravitational force on the vehicle acts in the $+z_{NED}$ direction and is equal in magnitude to the vehicle's weight W, leading to

$$\vec{F}_{G_{NED}} = 0\hat{i}_{NED} + 0\hat{j}_{NED} + W\hat{k}_{NED} \tag{2.2.6}$$

The aerodynamic and gravity forces can be combined and expressed in the body frame

$$m\vec{a}_b = \vec{F}_{G_b} + \vec{F}_{A_b} \tag{2.2.7}$$

which can then be expressed as

$$m\vec{a}_b - m\vec{g} = \vec{F}_{A_b} \tag{2.2.8}$$

$$m(\vec{a}_b - \vec{g}) = \vec{F}_{A_b} \tag{2.2.9}$$

Then, it is noted that a body mounted accelerometer will not measure $\vec{a}_b$, but will instead measure the quantity $\vec{a}_b - \vec{g}$. This means Equation 2.2.9 is really

$$\vec{F}_{A_b} = -m\vec{r}_b \tag{2.2.10}$$

where $\vec{r}_b$ is the reading from a body mounted accelerometer. The aerodynamic forces in wind axes, which is the goal, can then be calculated using a rotation matrix

$$\vec{F}_{A_w} = \bar{R}^b_w \vec{F}_{A_b} \tag{2.2.11}$$

Equations 2.2.10 and 2.2.11 are important, and show that the only sensors necessary for drag polar estimation during gliding flight are a body-mounted accelerometer and an air data system.

2.3 Kalman Filter Usage

This paper utilizes multiple Kalman filters to estimate both regression coefficients and improved states. As a brief overview, the Kalman filter combines the measured state with a predicted state to give an optimal[11] estimate of the actual system state.

Linear Kalman Filter

A linear Kalman filter can be applied[12] where the system in question can be described in the form

$$x_k = \bar{A}x_{k-1} + \bar{B}u_{k-1} + w_{k-1} \tag{2.3.1}$$

where $\bar{A}$ is the state transition matrix, x_{k-1} is the previous state, $\bar{B}$ is the input matrix, u_{k-1} is the input vector, and w_{k-1} is random process noise.

The measured state is then

$$z_k = \bar{H}x_k + v_k \tag{2.3.2}$$

where $\bar{H}$ is the output matrix and v_k is measurement noise.
The Kalman filter operates in a predictor-corrector manner, where the predictor step is often called the *a priori* estimate, and the corrector step is often called the *a posteriori* estimate. The *a priori* state estimate is calculated using prior states and inputs, while assuming no process noise

$$\hat{x}_k^- = \bar{A}\hat{x}_{k-1} + \bar{B}u_{k-1} \tag{2.3.3}$$

The *a priori* estimate of the covariance matrix is projected in a similar manner

$$\bar{P}_k^- = \bar{A}\bar{P}_{k-1}\bar{A}^T + \bar{Q} \tag{2.3.4}$$

where $\bar{P}$ is the covariance matrix and $\bar{Q}$ is the process noise matrix.

The Kalman gain is calculated by combining the predicted, *a priori* covariance matrix with the measurement noise covariance matrix $\bar{R}$

$$\bar{K}_k = \bar{P}_k^- \bar{H}^T (\bar{H}\bar{P}_k^- \bar{H}^T + \bar{R})^{-1} \tag{2.3.5}$$

This optimal Kalman gain is then used to estimate the *a posteriori* estimate of the state and covariance matrix

$$\hat{x}_k = \hat{x}_k^- + \bar{K}_k(z_k - y_k) \tag{2.3.6}$$

$$\bar{P}_k = (\bar{I} - \bar{K}_k\bar{H})\bar{P}_k^- \tag{2.3.7}$$

where y_k is the predicted value of z_k found using the output matrix $\bar{H}$ and the *a priori* state estimate

$$y_k = \bar{H}\hat{x}_k^-. \tag{2.3.8}$$

Note that Equation 2.3.6 is essentially a weighted average of a measured state and an expected state. The weighting is the Kalman gain, which is related to the ratio of confidence in the measured state and the expected state. For a 1-D case with equal confidence between the measured state and the expected state, the Kalman gain $\bar{K}_k = 0.5$, and the Kalman filter becomes a simple mean.

Extended Kalman Filter

The Extended Kalman filter is used for a non-linear system and is essentially a linearization of a nonlinear plant. A nonlinear system can be described as [12]

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.3.9}$$

$$z_k = h(x_k, v_k) \tag{2.3.10}$$

The process noise w_{k-1} and measurement noise v_k are not known (or the Kalman filter would not be necessary), so the states are approximated assuming both noise sources are zero.

$$\tilde{x}_k = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.3.11}$$

$$\tilde{z}_k = h(\tilde{x}_k, 0) \tag{2.3.12}$$

The actual states are related to the approximate states by

$$x_k \approx \tilde{x}_k + \bar{A}(x_k - \hat{x}_{k-1}) + \bar{W} w_{k-1} \tag{2.3.13}$$

$$z_k \approx \tilde{z}_k + \bar{H}(x_k - \tilde{x}_{k-1}) + \bar{V} v_k \tag{2.3.14}$$

where the matrices $\bar{A}$, $\bar{W}$, $\bar{H}$, and $\bar{V}$ represent the different Jacobian matrices:

$$\bar{A} = \frac{\partial f_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.3.15}$$

$$\bar{W} = \frac{\partial f_i}{\partial w_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.3.16}$$

$$\bar{H} = \frac{\partial h_i}{\partial x_j}(\hat{x}_k, 0) \tag{2.3.17}$$

$$\bar{V} = \frac{\partial h_i}{\partial v_j}(\hat{x}_k, 0). \tag{2.3.18}$$

The Extended Kalman filter uses these linearized equations to perform the same process as the linear Kalman filter. Again, the first step is to calculate the *a priori* estimate of the state and the covariance matrix

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.3.19}$$

$$\bar{P}_k^- = \bar{A}_k \bar{P}_{k-1} \bar{A}_{k-1}^T + \bar{W}_k \bar{Q}_{k-1} \bar{W}_k^T. \tag{2.3.20}$$

Next, the Kalman gain is calculated

$$\bar{K}_k = \bar{P}_k^- \bar{H}_k^T (\bar{H}_k \bar{P}_k^- \bar{H}_k^T + \bar{V}_k \bar{R}_k \bar{V}_k^T)^{-1}. \tag{2.3.21}$$

The Kalman gain is then used to calculate the *a posteriori* estimate of the state and covariance matrix

$$\hat{x}_k = \hat{x}_k^- + \bar{K}_k (z_k - y_k) \tag{2.3.22}$$

$$\bar{P}_k = (\bar{I} - \bar{K}_k \bar{H}_k) \bar{P}_k^- \tag{2.3.23}$$

where y_k is, as in the linear case, the predicted value of z_k, but calculated using the nonlinear output function and the *a priori* state estimate

$$y_k = h(\hat{x}_k^-, 0). \tag{2.3.24}$$

Unlike the linear Kalman filter, the Extended Kalman filter is not proven to be optimal. However, it has been utilized for a wide range of applications with excellent results.

3.0 Drag Meta-Modeling

Drag force comes from many different contributions, but can generally be split into drag that is independent of lift, and drag that is due to lift [13]. The summation of all sources of drag that are independent of lift is often called minimum drag ($C_{D_{min}}$ often used for the coefficient)[4], and is roughly constant, for a given Reynold's number and Mach number. The drag due to lift can be split into viscous drag-due-to-lift and inviscid drag-due-to-lift.

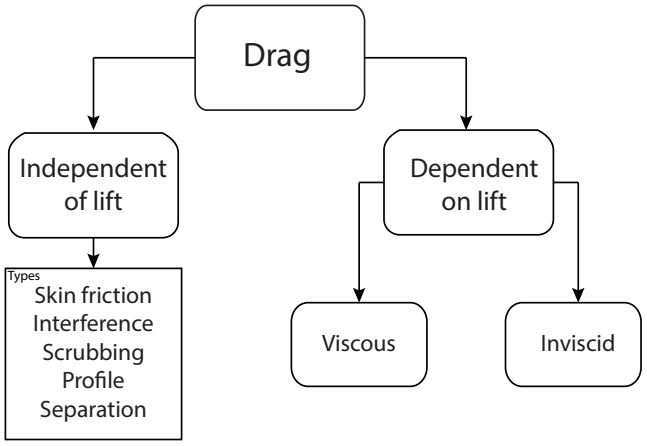


Figure 3.1: Drag Contribution Types

The viscous drag-due-to-lift is a type pressure drag that increases when the angle of attack of the wing increases, therefore generating more lift. It is a function of airfoil geometry, such as leading edge radius, thickness distribution, and camber. This type of drag is independent of finite wing vortices, and can be seen on two-dimensional airfoil data charts. To show an example of viscous drag-due-to-lift, a NACA 4412 was analyzed using XFOIL[14].

Figure 3.3 shows the airfoils viscous drag-due-to-lift. Since the shape is roughly parabolic through the linear region of the lift curve, the contribution to the total aircraft drag is

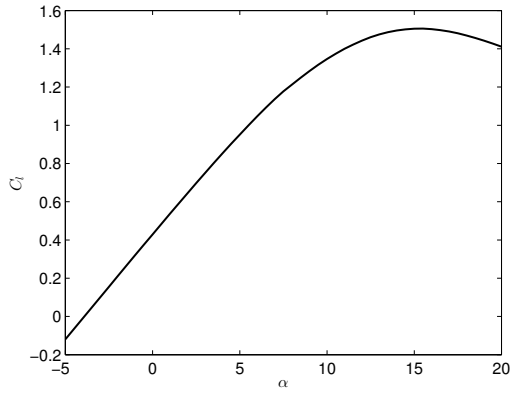


Figure 3.2: NACA 4412 Lift Curve

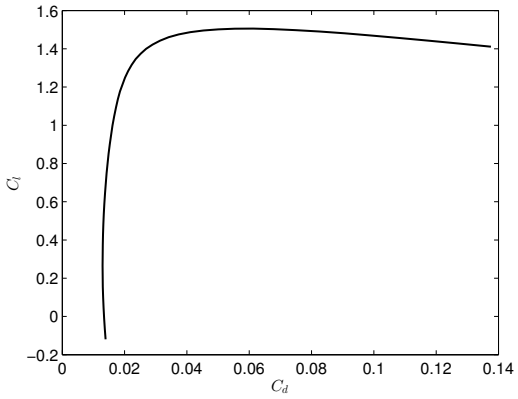


Figure 3.3: NACA 4412 Drag Polar

approximated using the Equation 3.0.1[13].

$$C_{D_{Visc.Lift}} = K_1 * (C_L - C_{L_{MinDrag}})^2 \tag{3.0.1}$$

where K_1 is the slope of the line shown in Figure 3.4, and $C_{L_{MinDrag}}$ is the lift coefficient at which minimum drag occurs (roughly 0.25 for this airfoil).

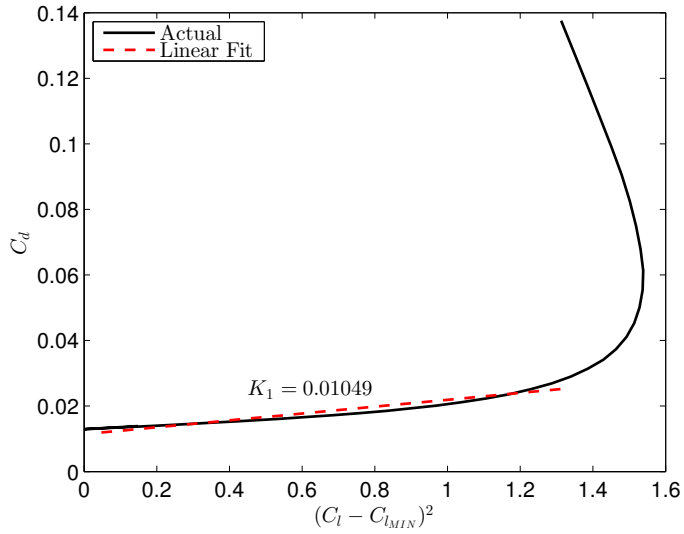


Figure 3.4: NACA 4412 K_1 Estimation

Inviscid drag-due-to-lift occurs because of the pressure difference between the top and bottom of a finite wing. This pressure difference causes the high pressure flow underneath

the wing to slip over to the top of the wing, causing a downwash velocity on the free stream velocity, shown in Figure 3.5.

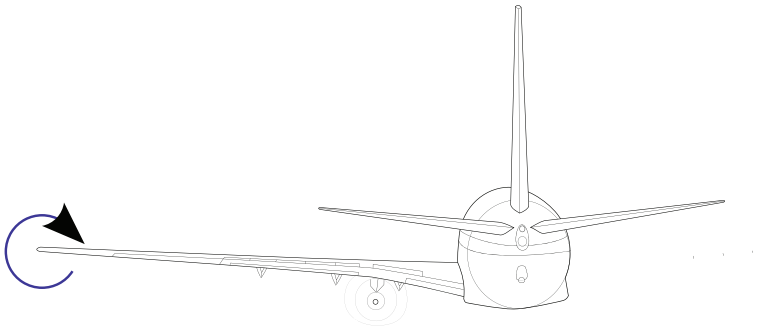


Figure 3.5: Downwash Caused By Wingtip Vortices[3]

This downwash changes the direction of the flow going into the airfoil. Since lift acts perpendicularly to the flow going into airfoil, there will be an induced angle between the free-stream lift vector (perpendicular to V_∞) and the airfoil's lift vector.

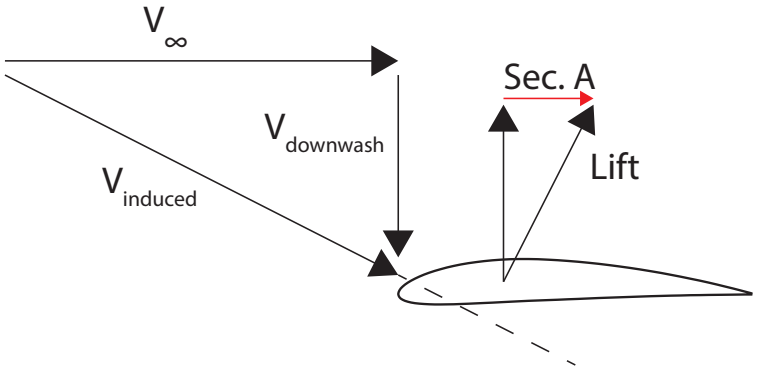


Figure 3.6: Induced Drag Free Body Diagram

The red section, labeled Sec. A in Figure 3.6, is a component of the airfoil's lift which is parallel to V_∞, meaning the airfoil's lift causes drag on the vehicle.

Induced drag is affected by the span lift distribution and the wing aspect ratio[15], and is governed by Equation 3.0.2.

$$C_{D_i} = \frac{C_L^2}{\pi e AR} = K_2 C_L^2 \tag{3.0.2}$$

These three coefficients (C_{D_0}, K_1, and K_2) are combined to result in a parabolic drag polar of the form

$$C_D = C_{D_{min}} + K_1(C_L - C_{L_{MIN}})^2 + K_2 C_L^2 \tag{3.0.3}$$

This drag polar gives valuable insight into how a vehicle will perform. The complete drag polar can be found using various regression techniques, discussed in the following sections.

3.1 Regression Model - Least Squares Fit

The standard model for a polynomial regression is

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + ... \tag{3.1.1}$$

When Equation 3.0.3 is expanded and like-terms of C_L are combined, equation 3.1.1 becomes

$$C_D = (C_{D_{min}} + K_1 C_{L_{min}}^2) - 2K_1 C_{L_{min}} C_L + (K_1 + K_2) C_L^2 \tag{3.1.2}$$

The value ($C_{D_{min}} + K_1 C_{L_{min}}^2$) is referred to as the parasite drag coefficient, and is represented by C_{D_0}[4]. These coefficients can be estimated using an Ordinary Least Squares fit. The Ordinary Least Squares problem statement is as follows

$$\bar{A}\vec{x} = \vec{b} \tag{3.1.3}$$

If $\bar{A}$ is an mxn matrix of measured state data, $\vec{x}$ is an nx1 vector of correlation coefficients, and $\vec{b}$ is an mx1 vector of measured function data, the solution to the Ordinary Least Squares problem is

$$\bar{A}\vec{x} = \vec{b} \tag{3.1.4}$$

$$\bar{A}^T\bar{A}\vec{x} = \bar{A}^T\vec{b} \tag{3.1.5}$$

$$(\bar{A}^T\bar{A})^{-1}(\bar{A}^T\bar{A})\vec{x} = (\bar{A}^T\bar{A})^{-1}\bar{A}^T\vec{b} \tag{3.1.6}$$

$$\bar{I}\vec{x} = (\bar{A}^T\bar{A})^{-1}\bar{A}^T\vec{b} \tag{3.1.7}$$

When applied to estimating a parabolic drag polar, the $\bar{A}$ matrix becomes

$$\bar{A}_{i,:} = \begin{bmatrix} 1 & C_{L_i} & C_{L_i}^2 \end{bmatrix} \tag{3.1.8}$$

the $\vec{b}$ vector becomes

$$\vec{b}_i = \begin{bmatrix} C_{D_i} \end{bmatrix} \tag{3.1.9}$$

and the $\vec{x}$ vector, which is the vector of interest, becomes

$$\vec{x} = \begin{bmatrix} C_{D_0} & -2K_1C_{L_{min}} & (K_1 + K_2) \end{bmatrix}^T \tag{3.1.10}$$

The coefficients C_{D_0}, K_1, and K_2 can then be found, assuming $C_{L_{min}}$ is known through wind tunnel testing, XFOIL, CFD, or other means.

3.2 Regression Model - Kalman Filter

The coefficients in question can also be estimated using an Extended Kalman filter. The system can again be described as

$$C_D = C_{D_0} - 2K_1C_{L_{min}}C_L + (K_1 + K_2)C_L^2 \tag{3.2.1}$$

For the Kalman filter regression, the state to be estimated are the coefficients C_{D_0}, $-2K_1C_{L_{min}}$, and $K_1 + K_2$. For ease of notation, substitute

$$C_1 = -2K_1C_{L_{min}} \tag{3.2.2}$$

$$C_2 = K_1 + K_2 \tag{3.2.3}$$

Since the regression coefficients should not change, the state transition matrix is an identity matrix, leading to

$$\hat{x}_k = \begin{bmatrix} C_{D_0} & C_1 & C_2 \end{bmatrix} \tag{3.2.4}$$

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2.5}$$

The measured data z_k is a vector containing the lift and drag coefficients at the k-th instant in time

$$z_k = \begin{bmatrix} C_D \\ C_L \end{bmatrix} = h(x_k, 0) \tag{3.2.6}$$

The vector y_k contains the estimates of C_D and C_L found using the *a priori* state vector $\hat{x}_k^-$ and is equal to

$$y_k = \begin{bmatrix} C_{D_{0_k}}^- + C_{1_k}^- C_L + C_{2_k}^- C_L^2 \\ C_L \end{bmatrix} = h(x_k^-, z_k, 0) \tag{3.2.7}$$

To implement into the Extended Kalman filter, the Jacobian of $h(x_k^-, z_k, 0)$ with respect to x_k needs to be calculated. Once done, the H matrix in the EKF becomes

$$H_k = \begin{bmatrix} 1 & C_L & C_L^2 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.2.8}$$

With the H matrix calculated, the EKF algorithim can be implemented as described in Section 2.3. The measurement noise covariance at each instant was calculated by doing error propagation as described in Section 4 and the process noise covariance was set to zero because the parabolic regression coefficients should be exactly constant.

4.0 Error Analysis

Without estimates of the error in data, the data itself is fairly meaningless. There are two main types of error in the system: model regression error and the error of a single data point. The error of a single data point comes from random noise in sensors, and can be decreased by improving the accuracy of the sensor or filtering the results. The model regression error comes from the fact that every aspect of the dynamics of the system are not precisely modeled. This type of error can be reduced by improving the accuracy of the dynamics being modeled, choosing a better regression model, or by increasing the number of data points collected, discussed later.

Random Error

The equations of motion can be used to propagate uncertainty in a signal, and they allow the uncertainty in the coefficients to be estimated based on sensor noise.

The error of a single point is assumed to be random and normally distributed. The first step in error propagation is to define y_i to be the i-th entry of the true function vector, $\hat{y}_i$ to be the i-th entry of the measured function vector, and to then do a Taylor series expansion about the operating point.

$$\hat{y}_i = y_i + \frac{\partial y_i}{\partial x_j} dx_j \tag{4.0.1}$$

where x_j is the j-th element of the state vector. Note that the term $\frac{\partial y_i}{\partial x_j}$ is the Jacobian matrix (J_{ij}) of the state transition function. The error can then be defined as

$$dy_i = \hat{y}_i - y_i = J_{ij} dx_j. \tag{4.0.2}$$

If the error is then interpreted as a discrete difference instead of a continuous difference, Equation 4.0.2 becomes

$$\Delta y_i = J_{ij} \Delta x_j. \tag{4.0.3}$$

If the Δ values are further assumed to represent standard deviations of normally distributed error, Equation 4.0.3 becomes

$$\sigma_{y_i} = J_{ij} \sigma_{x_j}. \tag{4.0.4}$$

For the purposes of this research, σ_{y_i} is a vector of the standard deviations of the aerodynamic force coefficients,

$$\sigma_{y_i} = \begin{bmatrix} \sigma_{C_{D_i}} & \sigma_{C_{Y_i}} & \sigma_{C_{L_i}} \end{bmatrix}^T \tag{4.0.5}$$

and σ_{x_j} is a vector of the standard deviations of the state values,

$$\sigma_{x_j} = \begin{bmatrix} \sigma_{r_{b_j}} & \sigma_{\alpha_j} & \sigma_{\beta_j} \end{bmatrix}^T. \tag{4.0.6}$$

For initial error estimation, the noise levels reported by instrument manufacturers was assumed to be one standard deviation of a normal distribution with mean equal to zero. The Jacobian matrix was calculated at each observed data point and combined with the sensors' standard deviation to produce estimates of the standard deviations of the aerodynamic coefficients.

One of the key findings of this section is that the error of the coefficient depends on the coefficient itself. This means that the error varies from data point to data point, which is called *heteroskedasticity* (as opposed to *homoskedasticity*, which means the error is independent of the state itself). To account for this, an error estimate function was created in MATLAB, which used the error propogation outlined in this section. This function was used whenever an estimate of point error was required, such as for the variance matrix P_k used in the Kalman filters utilized for state estimation. A plot of simulated flight data is shown in Figure 4.1,

where the error bounds shown were calculated using the error estimate function. Note that this figure also shows the heteroskedastic nature of the error.

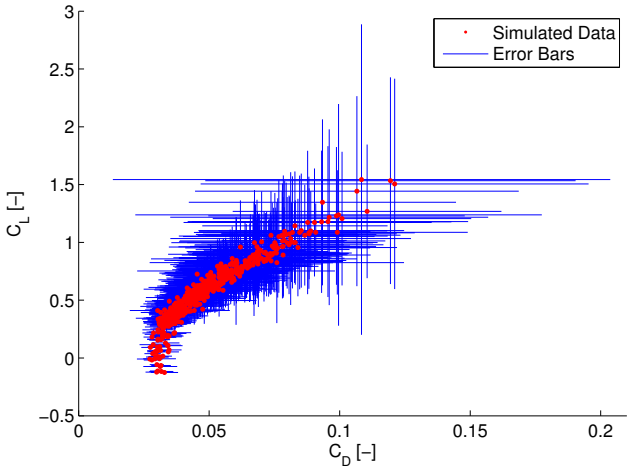


Figure 4.1: Heteroskedastic Error from Simulated Flight

While heteroskedasticity does not color the estimate of $\hat{\beta}_i$, it does color the confidence intervals. The method of dealing with this problem is discussed in the next section.

Least Squares Regression Error

The error of a single data point is not the main driving factor in the accuracy of a regression model. The important factor in the model is how accurately the coefficients are known, which is a function of the accuracy of each point, as well as the number of points sampled. The main parameter that describes the accuracy of the regression coefficients are the confidence intervals. A confidence interval is a range of values such that, if the experiment were repeated, the parameter calculated would be within the range some percentage of the time. A parameter can be represented as an estimated value, with a confidence bound

$$\beta = \beta_{EST} \pm t\frac{\sigma}{\sqrt{n}} \tag{4.0.7}$$

where β is the parameter in question, β_{EST} is the estimated value of the parameter, t is the Student's t value based on the number of samples and the desired confidence interval, σ

is the standard deviation of the sample, and n is the number of data points collected. Since the number of data points collected during flight will be large ($n > 100$), the t value will be taken as 1.96 for a 95% confidence interval.

As previously mentioned, one of the assumptions made in a Least Squares regression is homoskedasticity. However, the error using standard uncertainty propagation is heteroskedastic. This becomes a problem in estimating confidence intervals, because the standard error can be driven by outliers. If each data point had the same error, these outliers could be valid. However, if the data is heteroskedastic, the outlier may have larger error bounds (see Figure 4.1,) meaning the data point is not as likely as it first appears. This fact can drive the standard error estimate to be larger than is appropriate, which leads to a larger confidence interval and possibly a false lack of rejection of the confidence interval's hypothesis test. To account for this, the `robustfit` function in MATLAB was used to estimate both the coefficients and robust standard error estimates. The `robustfit` function calculates heteroskedastically-robust standard error estimates by doing a weighted average, where the weighting is based on a radial basis function. This means that the farther away a data point is from the estimated regression model, the less impact it has on the standard error of the model. The default weighting function used by `robustfit` is bisquare, and it was used for this research.

Kalman Filter Regression Error

Kalman filters are often used to propagate states. The filter does this by combining the system dynamics with a measured state. The variance is propagated using Equation 2.3.23. When estimating coefficients using the Kalman filter, the $\bar{A}$ state transition matrix is an identity matrix, which is due to the fact that the coefficients stay constant. When propagated through the filter, this means the state estimate x_k is essentially a variance-weighted-average of the coefficient estimates. The matrix P_k contains the variance of the coefficient estimates. Equation 4.0.7 calculates the confidence interval of regression coefficients and needs the

standard deviation of the mean, also called the standard error. The matrix P_k can be used to calculate the confidence intervals by noting

$$\bar{P}_k = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_i^2 \end{bmatrix} \tag{4.0.8}$$

The confidence interval for coefficient β_i can be calculated using σ_i in Equation 4.0.7.

5.0 Simulation

A 6-DOF flight simulator was used to validate the drag prediction method before hardware was purchased. The main utility of the simulator was to provide simulated flight test data with signals that contained no errors. The actual sensors used for flight testing contain noise, and this noise can be added onto the pure simulator signals to test the sensitivity of the drag polar regression to sensor accuracy.

5.1 Simulation Environment

The flight simulator used was a model of the de Haviland Beaver that comes as a demo in the Aerospace Toolbox of Simulink. The Simulink model was modified to output required signals to the workspace, which essentially created a sensor with zero noise. The mass, moments of inertia, and reference lengths were then scaled to those of a Zagi R/C aircraft[16]. The original Simulink model was already connected to a FlightGear Flight Sim, used as a visualization engine. This model was slightly altered to make flight gauges function properly.

5.2 Simulation Inputs

The engine forces and moments were set to zero in the simulator, to match the assumption of a folding propeller. The drag force calculation built into the Beaver Simulink model was replaced with a parabolic drag polar of the form

$$C_D = C_{D_0} + K_1(C_L(\alpha) - C_{L_{min}})^2 + \frac{(C_L(\alpha))^2}{\pi e AR} \tag{5.2.1}$$

Airfoil data, including K_1, $C_{L_{min}}$, and $C_L(\alpha)$, was taken from nonlinear aerodynamic data of a NACA 0012[17]. While this approximation to a drag polar does not capture the nonlinear

section of profile drag rise due to stall, it does represent the limited lifting capability of a real wing, making it more realistic than assuming the wing does not stall.

5.3 Simulation Results

The first goal of the simulation testing was to verify the drag polar equations were correct, and that the data analysis routines developed in MATLAB did in fact match inputs to outputs. The simulation was initialized with various initial states to ensure there was no dependency on initial conditions. The vehicle was then flown by an R/C aircraft pilot using a joystick attached to the simulation. It was noted early in the simulation testing that flying a sweep of speeds was beneficial, as a wider range of the drag polar was flown. This result was included in much of the flight test planning. After adequate data had been taken, the data was analyzed without adding simulated sensor noise. The results in Figure 5.1 show that the equations of motion used in the data analysis functions properly calculate the coefficients being passed into the system.

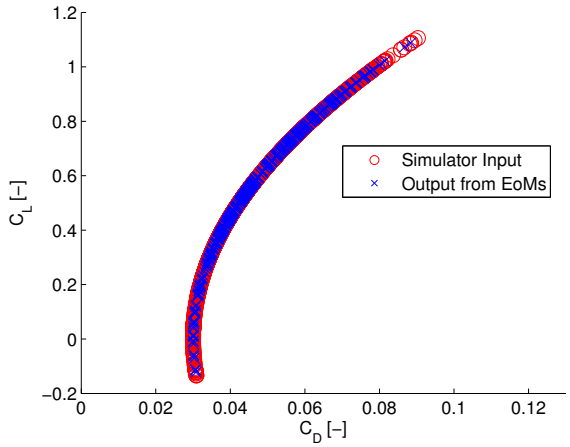


Figure 5.1: Data Analysis Verification (No Noise)

With this result, noise was added to the system to see how sensitive coefficient estimation was to noise in each sensor. This process was a balancing act between available sensor accuracy

and the desired accuracy of the final solution. The final result guided sensor selection to those discussed in Section 6. To check if the final sensors chosen were acceptable, Gaussian noise was added to each state, with a mean of zero and a standard deviation equal to the RMS error listed in the manufacturer's data sheet for each sensor.

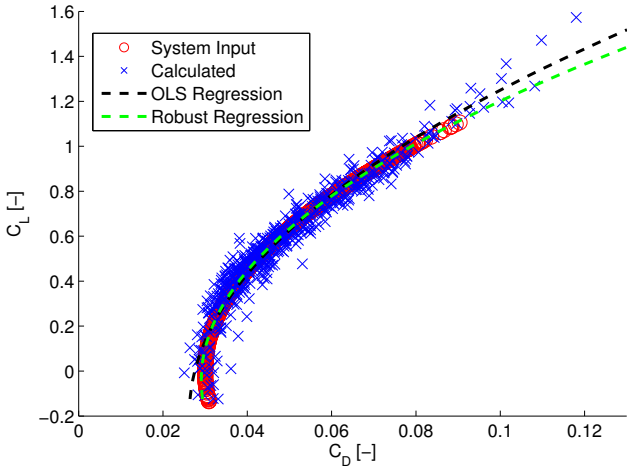


Figure 5.2: Drag Polar Prediction of Simulated Test Flight

For the particular simulated test flight shown in Figure 5.2, the estimated drag polar coefficients are shown in Table 5.1.

Table 5.1: Nonlinear Model Results

	C_{D_0}	K_1	K_2
System Inputs	0.0493	0	0.03
OLS Estimate	0.0355	-0.0136	0.0275
Robust LS Estimate	0.0460	-0.0037	0.0292

The results of this simulated flight test showed that the measurement system outlined in Section 6 predicted the simulated drag polar with a reasonable error. It also demonstrates the necessity of the heteroskedasticity correction, as the OLS regression has a 39% error on K_2 and a 9% error on C_{D_0}, while the robust regression has a 7% error on K_2 and a 3% error on C_{D_0}.

6.0 Hardware

One of the main goals of this research was to give the designer flexibility in choosing the appropriate sensors for a given flight test. To this end, hardware that is available in breakout boards was given preference, since it gives the aircraft designer more flexibility. The aircraft designer could utilize surface mount components if vehicle integration space is extremely limited, or can use the available breakout boards to make circuit-level integration easier if vehicle space is not a driving flight test concern.

6.1 Flight Computer

The flight computer chosen was an Arduino Due. This board has a 32-bit ARM processor, 54 digital I/O pins, 12 analog input pins, and 2 analog output pins. The main driver in the decision to use an Arduino-based platform was the vast support community, which allows quicker code development. The Arduino also offers a package that integrates well into most of the available airframes, and the stackable header pins allowed for easy integration with other boards. The Due in particular was chosen as it is (at the time of writing) the most advanced Arduino available. The main advantages it has over the comparable Arduino Mega is its increased clock speed (84 MHz for the Due[18] vs 16 MHz for the Arduino Mega[19]) and its 32-bit architecture (vs. 8-bit for the Arduino Mega).

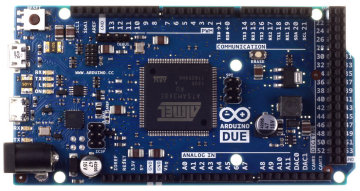


Figure 6.1: Arduino Due Flight Computer

The Arduino Due uses a 3.3V architecture instead of the usual Arduino architecture, which uses a 5V operating voltage. This was mainly beneficial, since most of the selected sensors used 3.3V as both supply and logic voltage. Logic level circuits, shown in Figure 6.2, were used to translate to 5V signals where required. The board is powered through the 3.5mm barrel jack, using a 3-cell LiPo battery, with a nominal voltage of 11.1V.

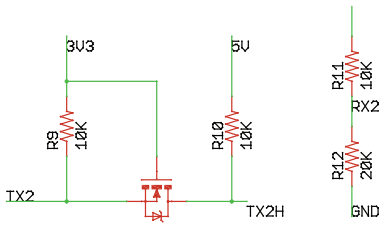


Figure 6.2: Logic Level Converter Circuit

6.2 Accelerometer

The accelerometer chosen for the data acquisiton system was the ADXL-362 from Analog Devices. It has a noise error of $175\mu G/\sqrt{Hz}$ and uses a 3.3V digital SPI interface[20]. The accelerometer is in an LGA package and was surface mounted to the main PCB.

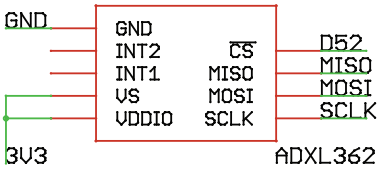


Figure 6.3: ADXL-362 Schematic

The accelerometer is calibrated in the field through an optimization routine using MATLAB's `fmincon` nonlinear constrained optimization function. For any given orientation, the

accelerometer's reading can be expressed as

$$\begin{bmatrix} r_x & 0 & 0 \\ 0 & r_y & 0 \\ 0 & 0 & r_z \end{bmatrix} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} + \begin{bmatrix} b_x & b_y & b_z \end{bmatrix} = \begin{bmatrix} (G_x - a_x) & (G_y - a_y) & (G_z - a_z) \end{bmatrix} \tag{6.2.1}$$

where the r terms are the bit readings from the accelerometer for each axis, the m terms are the slope of a linear fit for each axis, the b terms are the zero offset of a linear fit for each axis, and the a terms are the actual accelerations.

The slope and offset terms can be found through field calibration and a nonlinear optimization routine. The algorithm uses the fact that, while in a static orientation, the magnitude of the measured vector should be exactly 1G. The optimization problem is then

$$\underset{x}{\text{minimize}}\ f(x) = \sqrt{(1G - |(\vec{t})|)^2} \tag{6.2.2}$$

where $\vec{t}$ is the right hand side of Equation 6.2.1, and the variable of interest x is the vector of slopes and offsets in Equation 6.2.1. The slope terms in x are constrained to be positive. To be a deterministic system of equations, at least six static orientations are required. To avoid the noise of a single reading, the problem was expanded to take 100 data points in six different static orientations, and Equation 6.2.1 was expanded to become a least squares problem. The main benefit of this technique is that field calibration can be accomplished without needing precise knowledge of the orientation of gravity with respect to the sensor during the calibration routine. When tested, the results of the calibration were consistent between this algorithm and using known orientations. For error propagation, the noise during calibration was taken as the sensor's noise level.

6.3 Vehicle Mass

All test vehicles were weighed using a U-Line H-1650 counting scale. The scale has an accuracy of 0.001 lbs and a maximum capacity of 30 lbs. The minimum capacity of the scale is 10 grams [21].

6.4 Magnetometers

Two separate magnetometers were used for separate purposes. A Honeywell HMR-2300 3-D magnetometer, shown in Figure 6.4, is the main magnetometer. It is used when extremely accurate heading information is needed, or when GPS course is unavailable, such as during extremely slow or vertical flight.

Figure 6.4: Honeywell HMR-2300 3-D Magnetometer

This magnetometer provides a RMS error of 0.1 milliGauss for all axes, using the 1 Gauss full-scale setting[22]. The HMR-2300 can be supplied with power between 6V and 15V, so the 3-cell 11.1V nominal LiPo battery that powers the Arduino also passes through to power the magnetometer. The HMR-2300 operates using an RS-232 serial interface. To properly interface with the Arduino Due, which uses 3.3V TTL logic levels, a Max-3232 IC was used. This IC, when combined with charge pump capacitors, translates TTL levels between 3V and 5.5V to RS-232 logic levels of ±6V.

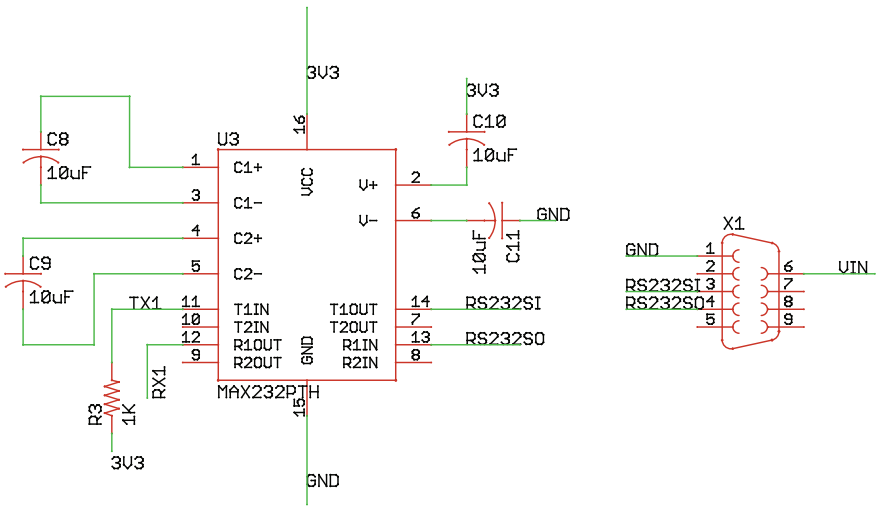


Figure 6.5: HMR-2300 Logic Level Circuit

The second magnetometer is a Honeywell HMC-5883L, which comes in an LCC package that was surface mounted to the main sensor board. It was added to the system for two main reasons: it is much smaller for applications where size is critical, and it is much less expensive for testing with unproven vehicles. It communicates with the Arduino using an I^2C interface and uses a 3.3V operating voltage[23]. The HMC-5883L has an accuracy of 2 milliGauss on each axis.

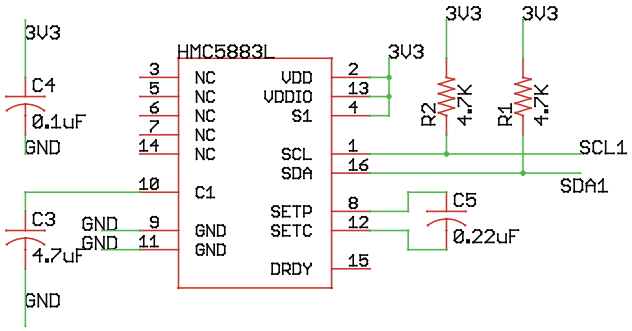


Figure 6.6: HMC-5883L Schematic

Both magnetometers were calibrated for both soft-iron and hard-iron effects[24]. To do this, data was acquired for 10 seconds with the magnetometer being swept through all directions. An ellipsoid was fit to the data using a ordinary least squares method available from the

MATLAB file exchange[25]. The least squares fit estimates the center and radius of each axis. The center values for each axis was subtracted from the readings to remove hard-iron effects. Each i-th axis is then scaled by $\frac{1}{R_i}$ to reshape the ellipse into a circle, which removes soft-iron effects.

The surface-mounted HMC-5883L was assumed to be aligned with the surface-mounted accelerometer. Since the two magnetometers both measured the North vector, a rotation matrix that describes the difference in alignment between the two sensors can be calculated by

$$\vec{N}_{HMC5883} = \bar{R}_b^{M_1} \vec{N}_{HMR2300}. \tag{6.4.1}$$

The rotation matrix $\bar{R}_b^{M_1}$ can then be used to align the HMR-2300's coordinate system with the body mounted accelerometers, using

$$\vec{N}_b = \bar{R}_b^{M_1} \vec{N}_{HMR2300}. \tag{6.4.2}$$

6.5 Gyroscope

A three-axis gyroscope was also included in the system. The gyroscope chosen was the Invensense ITG-3200, which comes in a QFN package. This gyroscope has a total error of 0.38°/s-rms[26], and uses a digital I^2C interface on a 3.3V operating voltage. This gyroscope has a full-scale span of ±2000°/s.

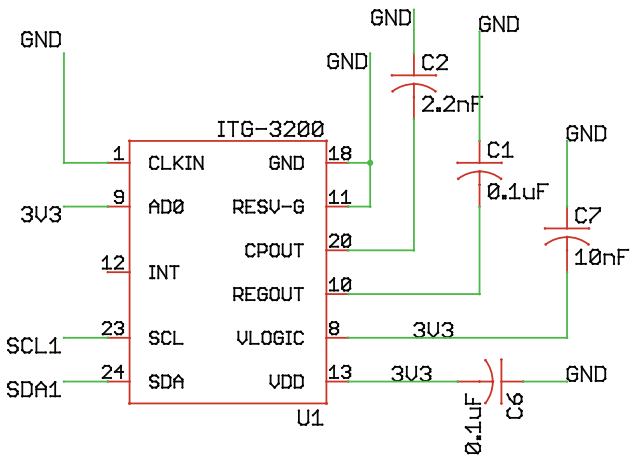


Figure 6.7: ITG-3200 Eagle Schematic

The gyroscope was calibrated in the same manner as the accelerometer. The device was placed in six orientations on a turn table which rotated at a constant 33 $\frac{1}{3}$ RPM. Slope and offset values for each axis were calculated using `fmincon`. Before each flight test, the offset values were re-calculated by taking 10 seconds of static readings.

6.6 Air Data System

A five-hole probe was chosen to measure aerodynamic angles as they do not contain moving parts and can provide very accurate, repeatable data. The five-hole probe selected was the Aeroprobe Air Data probe. It is 6 inches long, has a diameter of 1/8 inch, and uses a 0.25" hexagonal section as it's mounting section. The probe comes factory calibrated from angles to pressure readings, and was calibrated at an airspeed of 70 ft/s.

The probe was extended roughly one chord length in front of the leading edge of the wing by a 0.25" carbon fiber tube and an aluminum adapter which connected to the probe using set screws.

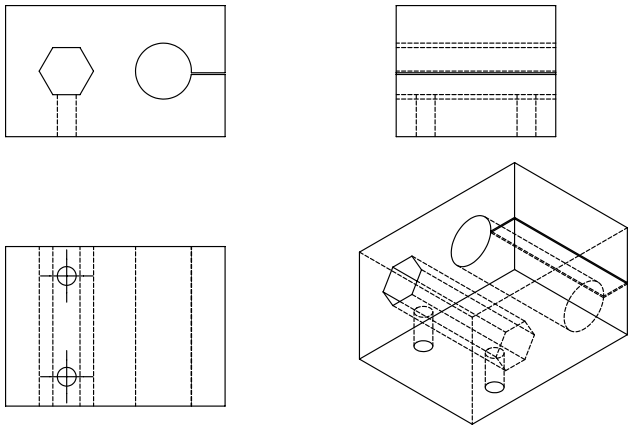


Figure 6.8: Five-Hole Probe Adapter

The adapter was manufactured to be square to itself and to have the reference flat of the probe's hexagonal section be parallel to one side of the adapter. An accelerometer was then glued to a side of the adapter, and the accelerometer was calibrated to the block using a level surface and the flat sides of the adapter. Before each flight test, three static readings are taken of the adapter's accelerometer and the main body-mounted accelerometer. Since the probe's orientation with respect to the adapter's accelerometer is known, this allows the wind angles measured by the probe to be calculated with respect to the body axes, and gives an accurate alignment of the wind reference frame to the body reference frame.

Each pair of lines of the five-hole air data probe is connected to an All Sensors digital differential pressure sensor with a full scale range of ± 5 in-H_2O[27]. The static port of the five-hole probe was connected to an All Sensors BARO-DO digital barometric pressure sensor,which has a range of 600 to 1100 mBar[28]. The barometric pressure sensor comes in the same package and uses the same communication protocol as the differential pressure sensors.

Figure 6.9: All Sensors 5-INCH-D-DO Pressure Sensor

The differential pressure sensors have a total error band of 0.25% FSO, and the barometric pressure sensor has a nominal error of 1 mBar. They use a UART serial interface that operates on a 5V logic level, so the logic levels were converted to the 3.3V levels of the Arduino Due. The serial interface includes addressable read commands, which allows multiple devices on a single bus, and ensures all devices record pressure at the same time. The sensor can output both a 14-bit pressure reading and a 12-bit temperature reading, which the device uses to correct its pressure measurement.

A digital temperature sensor was combined with the barometric pressure sensor to estimate the air density, which allowed air speed to be calculated.

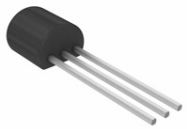


Figure 6.10: Dallas Semiconductors' DS18B20 Digital Temperature Sensors

The DS18B20 from Dallas Semiconductors was chosen for its relatively simple One-Wire interface. The device can be powered with the communication line and has a $\pm 0.5°C$ nominal accuracy[29].

The differential pressure sensors were calibrated for slope using a Reference Pressure Recorder from Crystal Engineering[30]. This calibration unit is capable of 0.025% of reading. The zero offset of each differential pressure sensor was removed before each flight by taking 100 data samples in a no-wind condition. The barometric pressure transducer and the temperature sensor were calibrated for offset using a Paroscientific Model 745 Pressure Standard, capable of 0.008% FSO accuracy[31].

Wind Angle Kalman Filter

To improve the accuracy of the wind angle estimation, a discrete Extended Kalman filter was used. The state transition functions are

$$\dot{\alpha} = \frac{1}{V\cos\beta}(-a_x \sin\alpha + a_z \cos\alpha) + q - (p\cos\alpha + r\sin\alpha)\tan\beta \tag{6.6.1}$$

$$\dot{\beta} = \frac{1}{V}(-a_x \cos\alpha \sin\beta + a_y \cos\beta - a_z \sin\alpha \sin\beta) + p\sin\alpha - r\cos\alpha. \tag{6.6.2}$$

These state transition equations come from solving for the vehicle forces in wind axes instead of body axes[9]. The equations for the Kalman filter for the wind angles are

$$\begin{bmatrix} \alpha_k \\ \beta_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_{k-1} \\ \beta_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta T & 0 \\ 0 & \Delta T \end{bmatrix} \begin{bmatrix} \dot{\alpha_k} \\ \dot{\beta_k} \end{bmatrix} + \hat{w}_{k-1} \tag{6.6.3}$$

$$z_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_k \\ \beta_k \end{bmatrix} + \hat{v}_{k-1}. \tag{6.6.4}$$

The process covariance matrix was calculated using the error propagation discussed in Section 4 and the standard deviation data from the zero offset calibration of each of the applicable sensors. The measurement noise covariance matrix was calculated based on the standard deviation data for the zero offset of each pressure transducer connected to the five-hole probe.

6.7 GPS Receiver

A uBlox LEA-6T GPS receiver was included in the data acquisition system. This model was selected for its ability to output raw timing data, which can be used to get an extremely accurate inertial velocity estimate[32]. The receiver itself was integrated onto a breakout board sold by CSG Shop, which has UART, USB, and I^2C interface options.

Figure 6.11: CGS Shop Board for uBlox LEA-6T

6.8 Data Acquisition System Integration

The sensors were packaged into a main shield for the Arduino. This shield plugs directly into the Arduino, eliminating the need to disconnect and reconnect wiring. Header pins capable of reading commanded PWM signals to servos were also added on the main board. Future work could include stability derivative estimation, and the header pins provide PWM measurement, which can map to servo angles, if it is assumed the servo is not stalled. The servo signal breakout pins also allowed the data to be easily split into sections with and without commanded throttle for drag polar estimation without thrust.

A second board was developed to integrate the air data system with the main sensor board. This pressure board can be located near a wing tip and provides expandability should additional sensors be desired in the future. It also interfaces with the temperature sensor,

which is located in the air flow. Finally, all data was saved to a microSD card attached to the main sensor board. Data was saved in binary format for both increased speed and file size reductions. Once on the ground, the data is converted to meaningful values using a custom MATLAB data parser.

7.0 Flight Test

Flight testing was done at Cal Poly's Educational Flight Range (EFR). The goal of flight testing was to validate the performance of the final system design. The drag polar estimation was chosen as the validation case, and was approached from each of the three coefficients. To this end, the final system was flown on multiple vehicles which each had a different role in the validation routine.

7.1 C_{D_0} Validation

Validation of the parasite drag coefficient was completed using a Finwing Universal Penguin FPV R/C aircraft[33].

This model was selected because it has a large internal payload bay which has plenty of room for the system and had enough excess power to overcome additional drag. The validation method involved flying the base model and measuring the drag polar. Then, parasite drag was added in the form of streamers, similar to those used in amateur rocketry recovery. The benefit of this technique is that it does not modify either of the other two drag polar coefficients. A power law fit to the expected drag coefficient of the streamers was given in [34]. The validation routine was flying the vehicle with and without streamers and seeing the horizontal shift in the drag polar.

7.2 K_1 Validation

The K_1 term of the drag polar is mainly driven by the C_L for minimum drag. To validate this coefficient, a custom test vehicle was manufactured with two different wings of equal area. One wing had a NACA 63-014 airfoil cross section, which is a symmetric airfoil, meaning the vehicles K_1 value should be zero. The next wing had a NACA 63-614 airfoil, which has a

design C_L for minimum drag of 0.6. The validation method was flying both of these wings on the same vehicle, and seeing the vertical shift in the drag polar.

7.3 K_2 Validation

The K_2 drag polar coefficient is the inviscid drag due to lift term. This form of drag comes from the wing tip vortices of a finite wing, and is affected by both wing aspect ratio and lift distribution. Prandtl showed experimentally[15] that, for rectangular wings with aspect ratios between one and seven, the drag coefficient corresponding to one aspect ratio can be scaled to that of a different aspect ratio using the equation

$$C_{D,1} = C_{D,2} + \frac{C_L^2}{\pi e}(\frac{1}{AR_1} - \frac{1}{AR_2}) \tag{7.3.1}$$

The K_2 validation uses this fact by flying the same test aircraft with two different sets of wings with the same area. The first wing has an aspect ratio of three, while the second wing has an aspect ratio of seven. The $AR = 3$ wing was then corrected to the $AR = 7$ wing and the resulting drag polar should fall on top of each other.

7.4 Results and Future Work

The system was built and tested to prove functionality. After functionality testing was complete, the system was integrated into a 0.40-size Piper Cub R/C aircraft.

Figure 7.1: System Integration into 0.40-size R/C Piper Cub

After integration, an initial test flight was conducted. Unfortunately, an electrical short-circuit caused the vehicle to crash and corrupted some data, so no aerodynamic force estimation has been conducted. However, the micro-SD card did survive the crash, and provided enough data to show the system was collecting data as expected before the short-circuit.

Figure 7.2: Pre-Flight System Checks

Figure 7.3: Crashed Vehicle in Water

The small amount of data collected was analyzed using a user interface developed to rapidly process the data acquisition system's files. This user interface will allow designers to analyze data immediately following a test flight and make corrections to the test flight program while still at the test flight location.

Due to the vehicle crash, the accuracy of the system developed still needs to be extensively validated. The immediate future work following this paper will be quantifying the accuracy of the system, as well as verifying reliability and safe integration into multiple unmanned systems. Most of the accuracy estimation will be accomplished by measuring changes to a vehicle's drag polar. The ability to measure parasite drag will be proven by adding payloads with a known drag coefficient to "dirty" the aircraft, and then measuring the change in the parasite drag coefficient of the test vehicle. To quantify the accuracy of drag-due-to-lift measurement, wings with different aspect ratios will be used on the vehicle, which should combine into a single equivalent curve[15]. Following the accuracy estimation, a sensor fusion algorithm will be developed to combine inertial sensors with the air data system and other available sensors in a manner similar to other current research,[35],[36] in order to give full situational awareness to the UAS. This situational awareness could allow stability and control derivative estimation, which the aircraft designer could use to size tail and control surfaces.

8.0 Summary

This is the summary chapter.

A.0 System Usage

This section documents the steps required for correct usage of the data acquisition system.

A.1 Integration

The main data acquisition system should be integrated into the flight test vehicle near the vehicle's center of gravity. The system should have one of its principal axes lined up roughly parallel to the vehicle's longitudinal axis. Servos should be connected to both the main board and the aircraft's receiver using a y-splitter with 2 male and one female connectors. If applicable, the servos should be connected according to the Table A.1.

Table A.1: Air Data System Setup

Servo	PWM Pinout
Throttle	D37
Elevator	D38
Rudder	D39
Aileron (1/2)	D40
Aileron (2/2)	D41
Gear	D42
Auxiliary (1/2)	D43
Auxiliary (2/2)	D44

If the additional digital pinouts are being use for measuring servo signals, the "+V" jumper should not have a jumper on it. If using the digital pinouts to run digital sensors or command servos, place the jumper on the appropriate voltage level setting.

The pressure board can be placed anywhere in the vehicle, as long as it can be connected to the main board. The pressure sensors should be attached to the five-hole probe as follows:

Table A.2: Air Data System Setup

Pressure Sensor	Port A	Port B	Measurement
0	Tube 4	Tube 5	β
1	Tube 1	Tube 6	q_∞
2	Tube 2	Tube 3	α
3	n/a	Tube 6	P_S

Port A and Port B refer to the ports as labeled in Figure A.1, and the tube number refers to the five-hole probe tubes. These tube numbers increase as the tube legth decreases: Tube 1 refers to the longest tube, Tube 6 refers to the shortest tube.

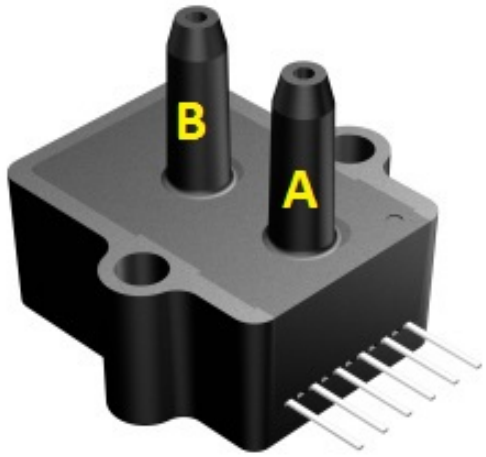


Figure A.1: Port Description for Pressure Sensors

The pressure tubing and temperature sensor line should be routed together to the air data boom and secured for flight. The temperature sensor's wiring is a standard servo extension. The air data boom's accelerometer wiring is a standard RJ-25 *patch* cable and should be routed in a separate bundle, as it will be removed before flight.

The air data boom itself should be integrated as far from aerodynamic effects as possible. For this thesis, the boom was mounted roughly one chord length in front of the leading edge of the wing and approximately halfway down the wing span. Other mounting locations could include out the aircraft's nose for a pusher vehicle, or on the vertical tail. The aluminum block accepts a 0.25" carbon fiber tube as it's mounting interface, and this tube should be mounted roughly in-line with the airflow angle expected during flight. The temperature sensor can be taped to the tube.

A.2 Pre-flight Procedure

The pre-flight procedure in this section must be completed in addition to all preparation listed in Section B.

1. With the transmitter turned on, power on the receiver.
2. Plug in the micro-USB cable into the main data acquisition board.
3. Ensure the micro-SD card is empty and formatted as FAT32, then insert it into the main data acquisition board.
4. Plug in the main data acquisition system's battery pack.
5. Plug the pressure sensor board into the main data acquisition board.
6. Plug in the air data system's RJ-25
7. Upload the calibration routine to the main board, and follow the prompts to calibrate the gyroscope, accelerometer, and magnetometers.
8. Place a wind blocker (Daisy/Solo cups work) over the five-hole probe and calibrate the pressure sensors, using the calibration routine.
9. Calibrate the air data system's orientation using the calibration routine.

10. Load the main data acquisition script onto the main data acquisition board.
11. Turn on serial output and check that all sensors are functioning properly.
12. Initialize the micro-SD card.
13. Turn off serial output.
14. Turn on data logging and verify the system is saving data.
15. Remove the micro-USB cable and the air data system's RJ-25 cable.

Talk about the pre-flight calibration process: air data boom orientation pressure offset calibration accel/gyro calibration magnetometer calibration

After the flight is complete, a second round of zero offset calibration data should be acquired. This set ensures that any drift that occurred during the flight is accounted for. Repeat the steps outlined in Section **??**.

A.3 Flight test plan

Yo bro, glide around for a while. change speeds and stuff.

A.4 Post-flight

Let's talk about how to use that sweet GUI of yours.

A.5 Software protocol

List of commands available for the mainScript and calib scripts, along with how they work.

B.0 Flight Test Procedure

This section documents proper flight test procedure, which has been learned from extensive flight testing experience and many crashed vehicles. It is split into three main time periods:

1. Pre-Flight Preparation
2. Flying Field Procedure
3. Post-Flight

The testing procedure is split into these time categories to ensure the testing is efficient and that any unforeseen circumstances can be dealt with quickly. Specifically , this guide applies to the Cal Poly Flight Lab testing a vehicle at the Cal Poly Educational Flight Range.

B.1 Pre-Flight Preparation

The preparation work required for a test flight is often overlooked, and this section will document how to effectively prepare to flight test a vehicle. The main purpose of pre-flight preparation is to minimize the possibility of problems that might force a test to be canceled after already going to the field.

B.1.1 Day Before Test

The days before a flight test are critical to ensuring the test is completed successfully. The following should be done at least one day before the the test is scheduled:

1. Charge all flight battery packs, including any receiver or auxiliary packs.
2. Charge the transmitter battery.

3. Ensure the airframe is structurally sound (wing tip test minimum.)

4. Verify radio system communicates properly, and the correct fail-safe is in place. **Important**: If the receiver has been used by Design/Build/Fly, the fail-safe must be changed to normal mode.

5. Verify control surface deflections matches desired directions, and all radio mixes work.

6. Verify the motor/propeller spin in the correct direction.

7. Pack a flight box, containing any necessary tools (recommend: screw drivers; masking, painter's, and strapping tape; razor blades; CA glue and kicker; spare propellers; paper and pencil; as a minimum.)

8. Check the weather. The closest monitoring station to the field is KCASANLU17.It is generally better to flight test early in the morning, since there will be less people at the field, winds will be calmer, and the sun won't be as harsh in the pilot's eyes.

9. Check the SLO Flyers' flight schedule. Some days are reserved for certain events (glider competitions, etc.) and these need to be worked around.

10. Make sure all required personnel know when and where to meet, and there is sufficient transportation to get to the field.

11. Create flight documentation that clearly lays out the test goals and how they will be accomplished. Print copies for all personnel.

B.1.2 Day Of Test

The morning of the flight test, the person responsible for the test should arrive early enough to accomplish the following, before leaving campus:

1. Pack flight batteries into flight box, including receiver and auxiliary packs.

2. Pack battery charging equipment, with adapters and leads, if necessary.

3. Pack transmitter into box.

4. Double check control surface deflections and radio link.

5. Do a full system check, potentially including a short taxi test in the quad.

6. Do a final check that all equipment made it into vehicles, before leaving the lab.

B.2 Flying Field Procedure

With the pre-flight preparation completed, the testing at the flying field should be fairly event free. Any problems that occur should be either fixable with the minimum supplies in the flight box, or the test should be canceled to minimize risk, and repairs done at the lab. Specific procedures at the flight field will depend on the test being conducted, but below are steps that apply to nearly all tests before flight.

1. Verify structural integrity using a wing tip test.

2. Verify motor/propeller are spinning in the correct direction.

3. Verify control surface deflections match desired directions.

4. Verify radio link and fail safe mode by completing a range check.

5. Verify center of gravity is at an appropriate location.

6. Create flight timer on the transmitter so the pilot knows how long the aircraft has been flying.

7. Document any required data before the flight, such as aircraft weight and geometry.

8. Document current weather conditions (wind speed and direction, temperature, humidity, barometric pressure) using the Kestrel portable weather station.

B.3 Post-Flight

Upon successful completion of a flight test, all data should be saved to a computer. The flight test cards, pre-flight data documentation, and any notes should be scanned and saved with the flight data. This data should be archived in a .zip file, with the date attached.

If the test flight resulted in a crash, any available data should still be saved. Any video or pictures of the flight should be saved as well, to aid in isolating what caused the crash. If useful, pictures should be taken of the crash site. Afterwards, all pieces of the aircraft should be returned to the lab, where the root cause of the accident should be determined.

After the root cause has been determined, remove all electronics from the vehicle, including batteries, speed controller, motor, servos, and receiver. If the vehicle was a complete loss or went into water, through these components away: **do not** return to lab supplies. If the vehicle was not a complete loss, verify all components work properly.

C.0 C_{D_0} Flight Test Card

D.0 K_2 Flight Test Card

E.0 Sample System Ouput

This appendix contains graphs representing typical outputs from the data acquisition system, shown in strip chart format.

E.1 Sample System Ouput - Raw Data

Figure E.1: accelX vs. Time

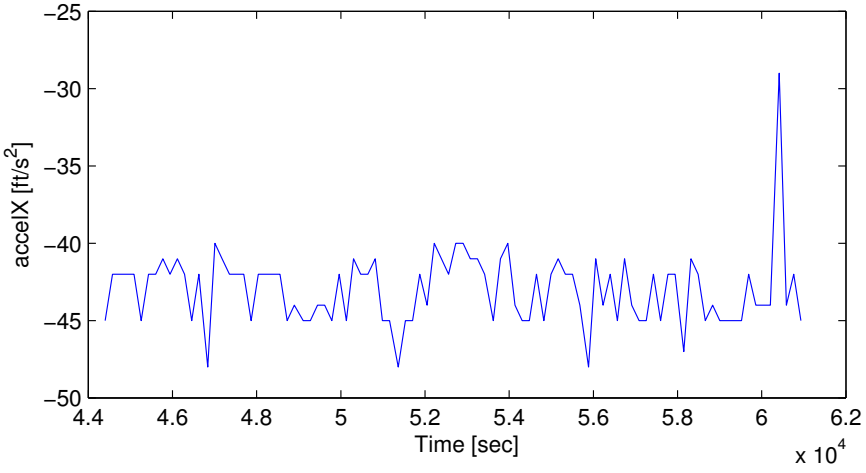


Figure E.2: accelY vs. Time

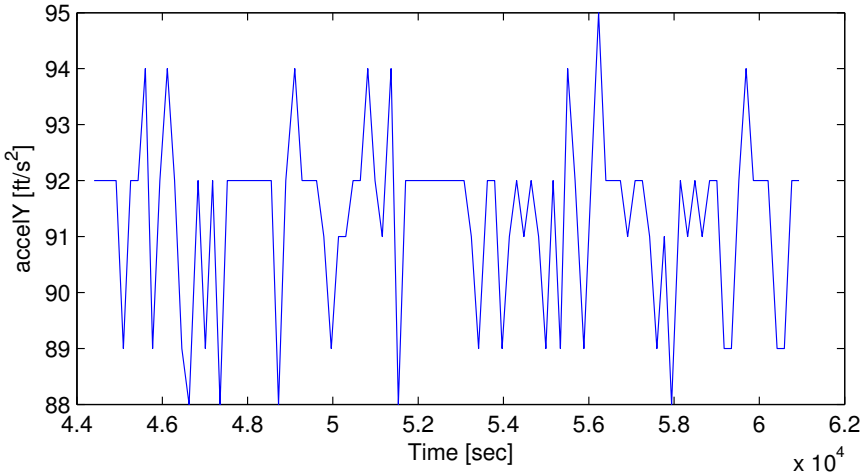


Figure E.3: accelZ vs. Time

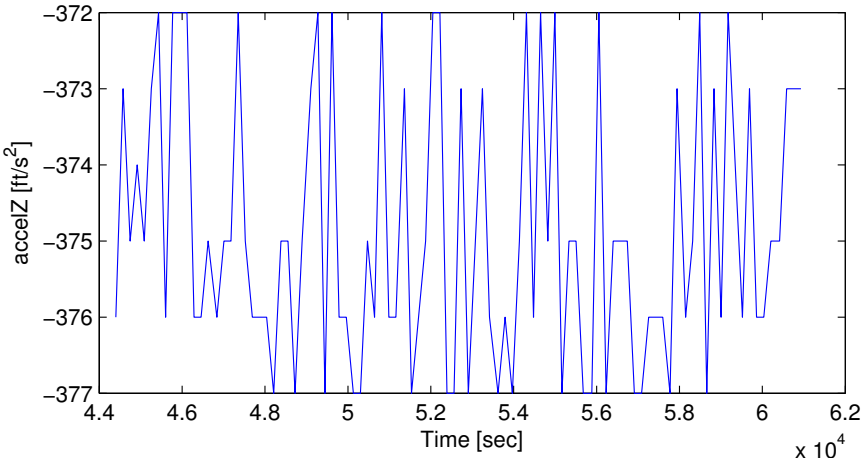


Figure E.4: gyroX vs. Time

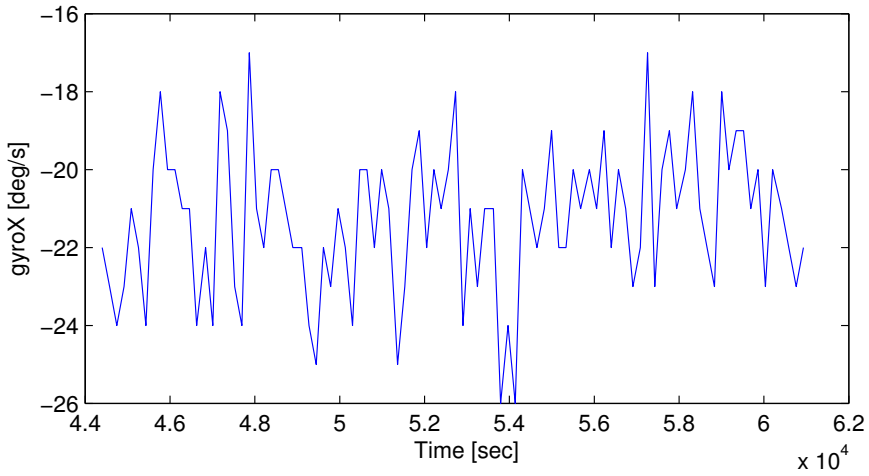


Figure E.5: gyroY vs. Time

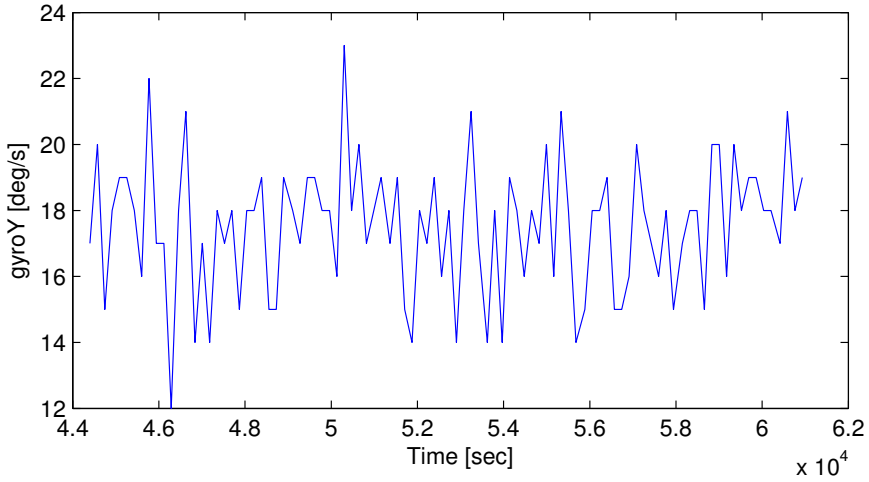


Figure E.6: gyroZ vs. Time

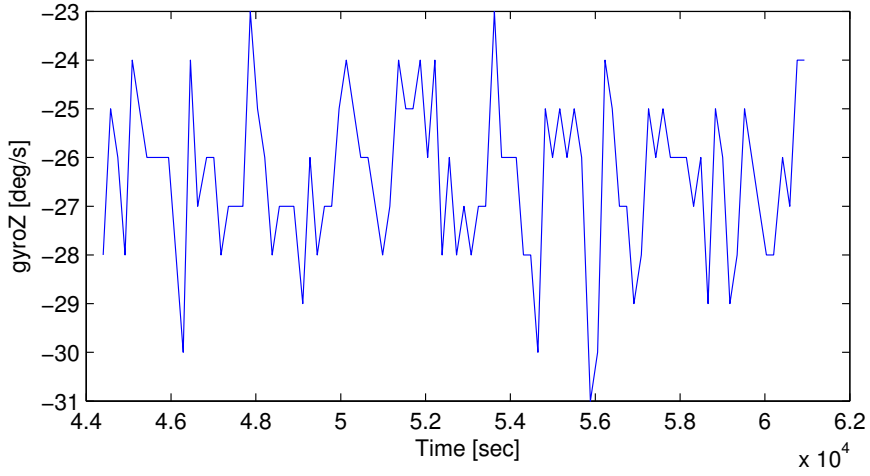
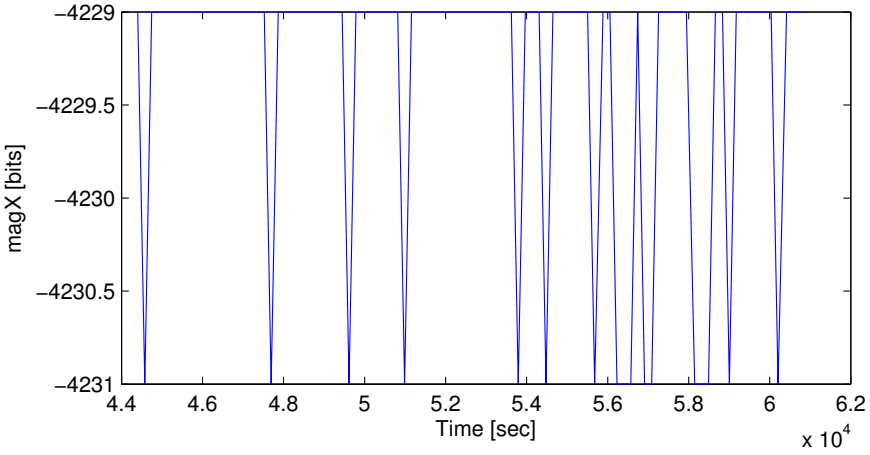


Figure E.7: magX vs. Time



E.2 Sample System Ouput - Units Data

Figure E.8: magY vs. Time

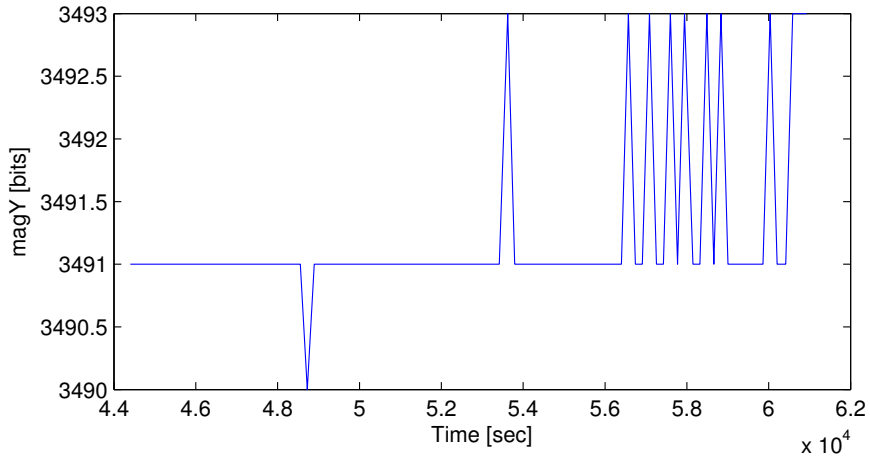


Figure E.9: magZ vs. Time

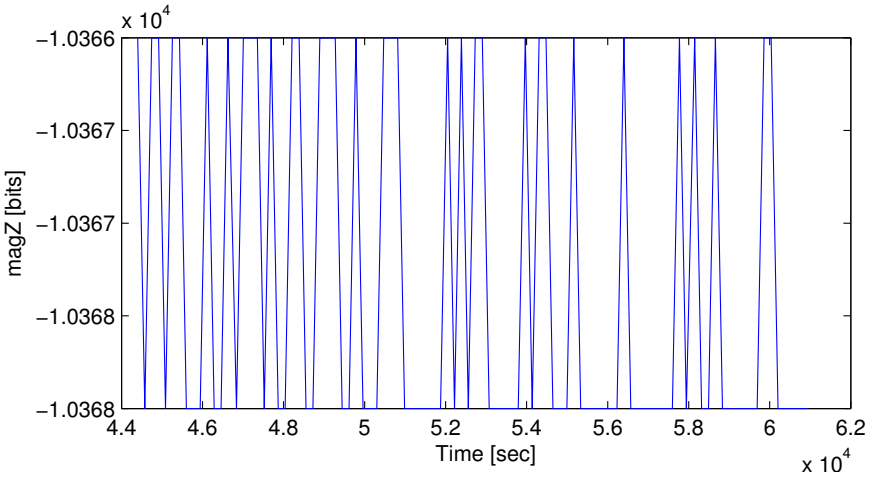


Figure E.10: press0 vs. Time

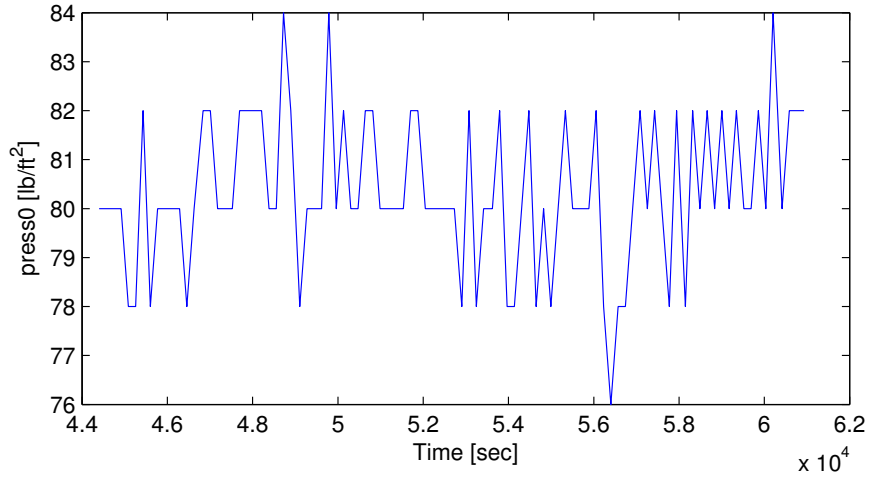


Figure E.11: press1 vs. Time

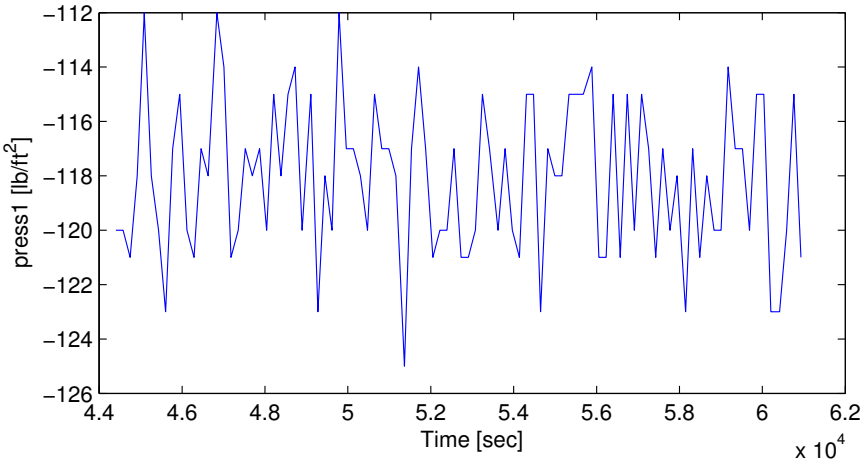


Figure E.12: press2 vs. Time

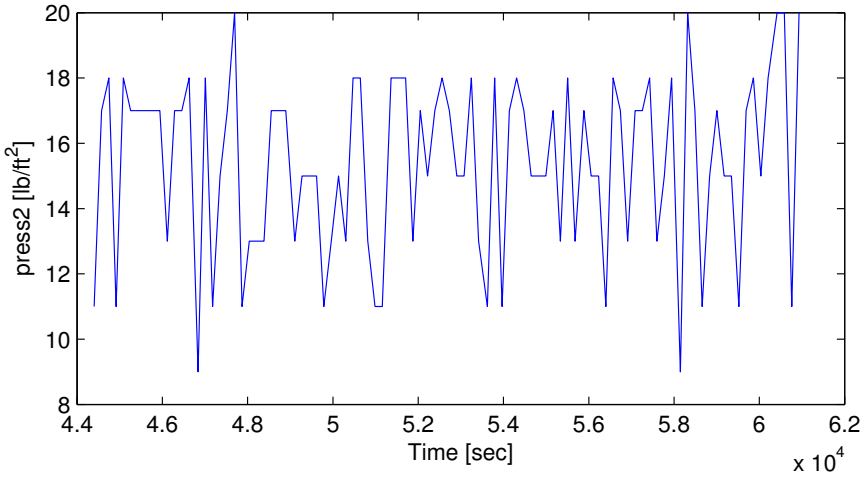


Figure E.13: press3 vs. Time

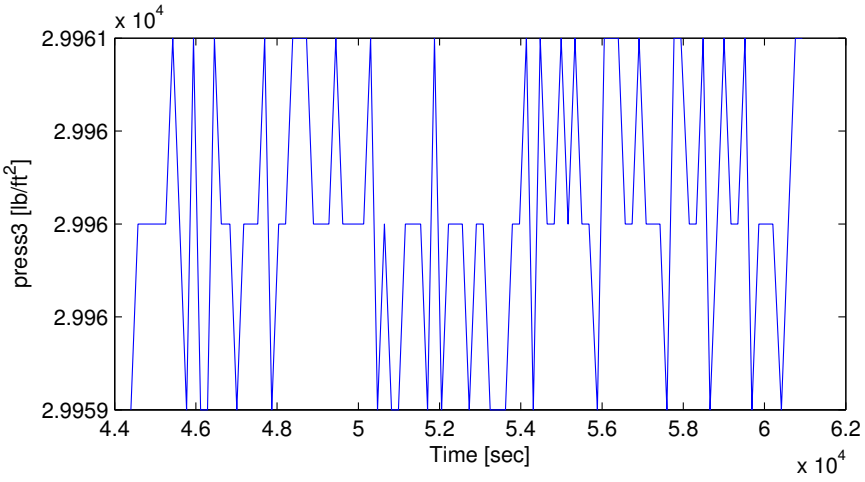


Figure E.14: gpsLat vs. Time

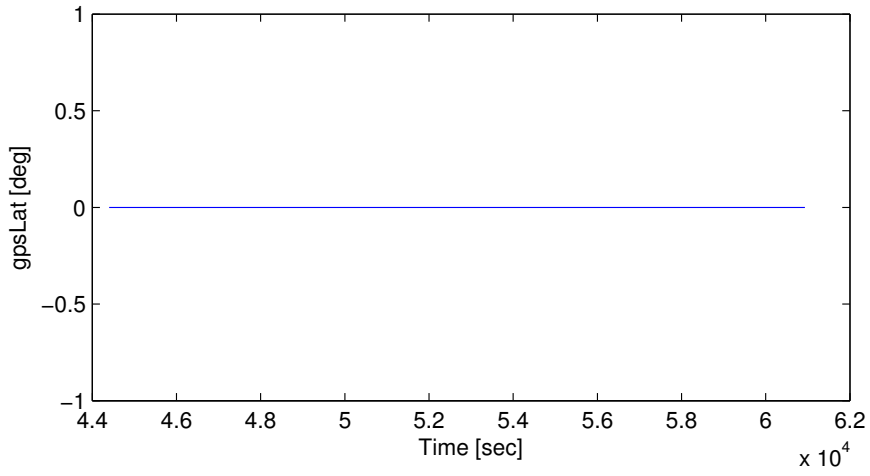


Figure E.15: gpsLong vs. Time

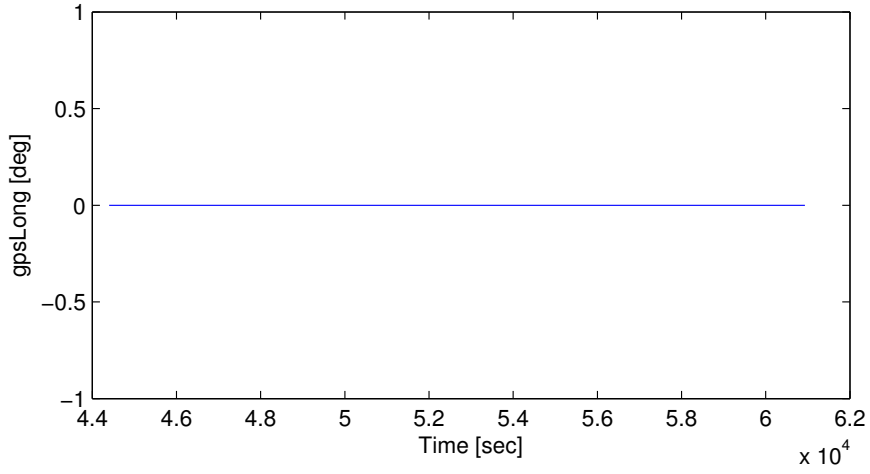


Figure E.16: gpsSpd vs. Time

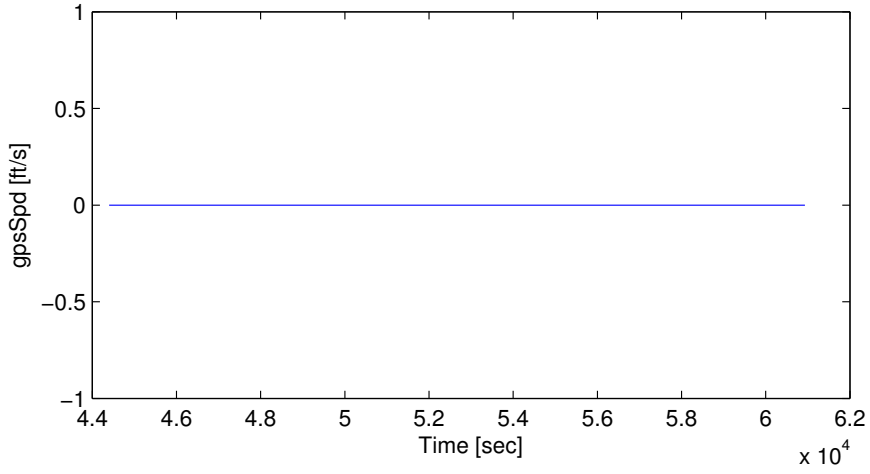


Figure E.17: gpsCrs vs. Time

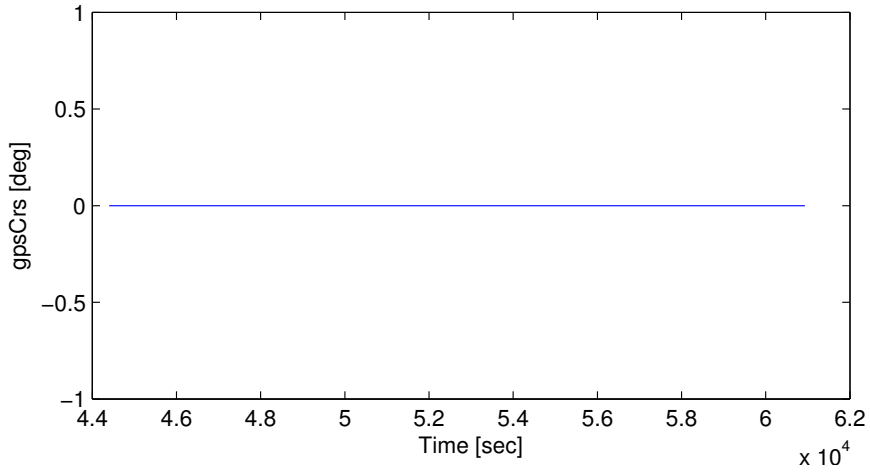


Figure E.18: date vs. Time

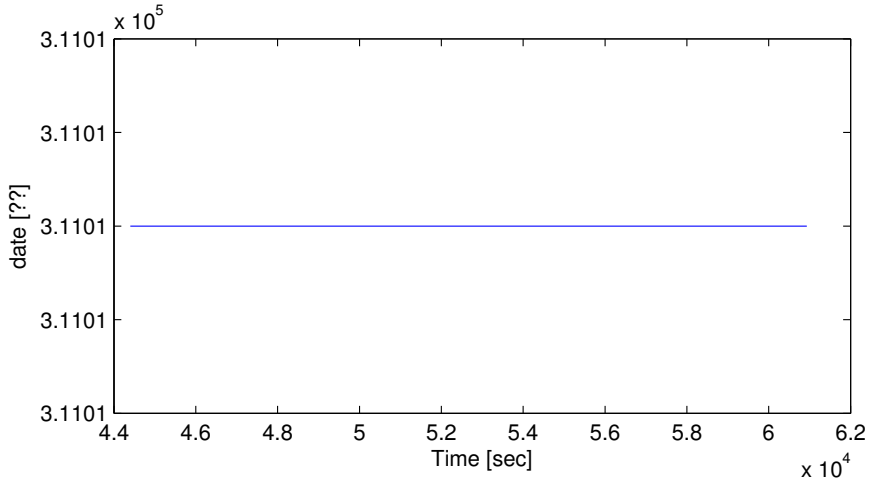


Figure E.19: CS vs. Time

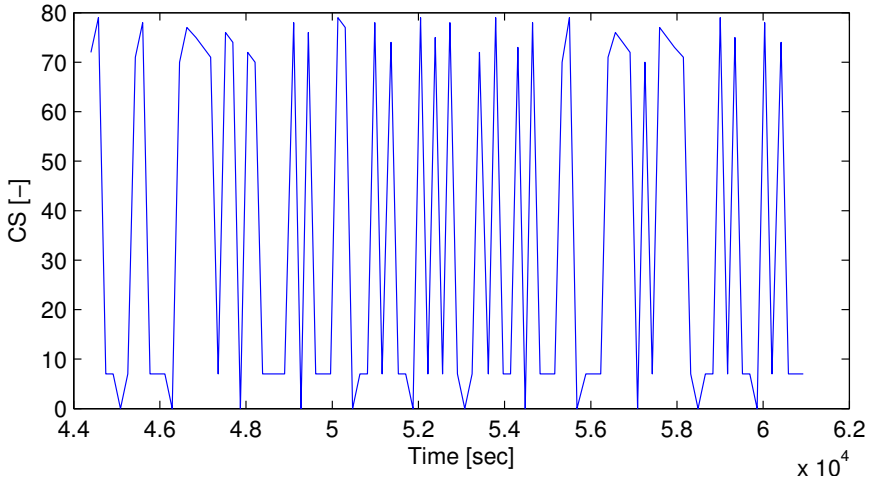


Figure E.20: temperature vs. Time

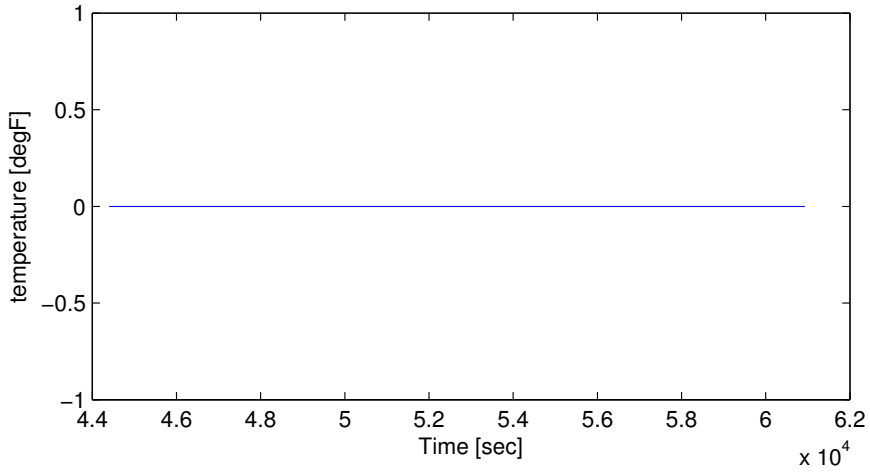


Figure E.21: deltaT vs. Time

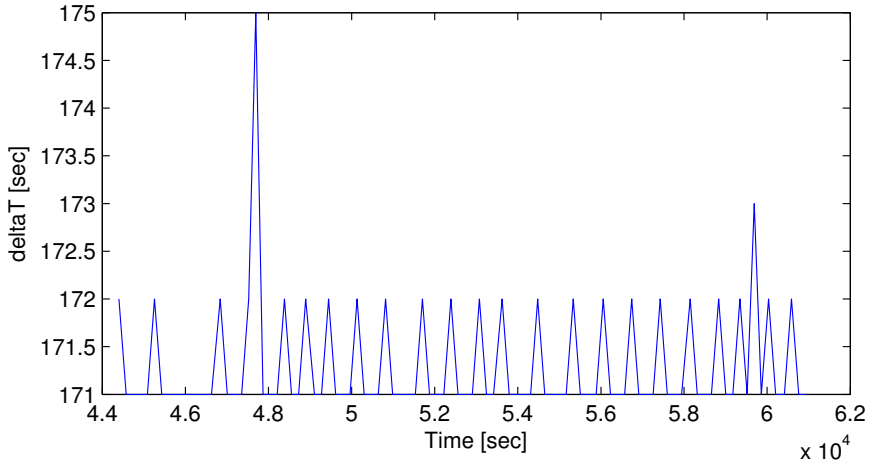


Figure E.22: rpmPwm vs. Time

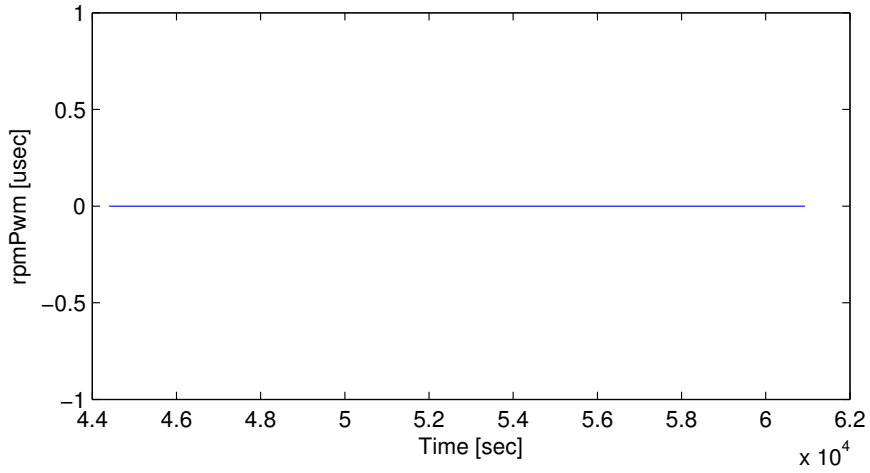


Figure E.23: accelX vs. Time

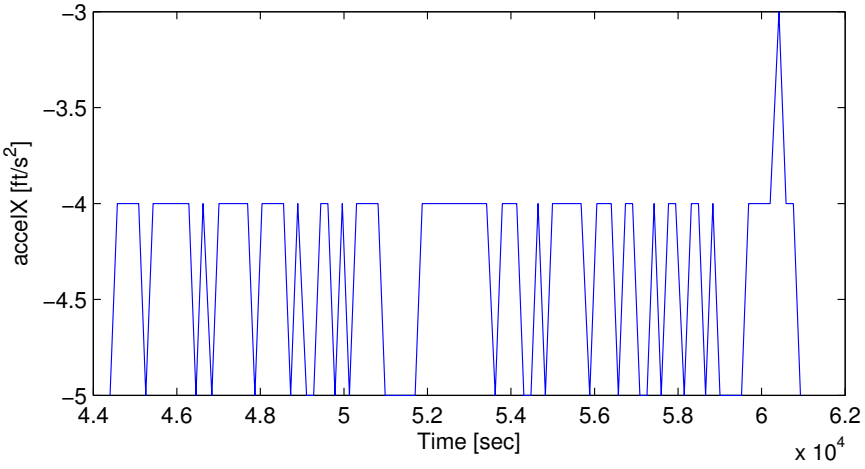


Figure E.24: accelY vs. Time

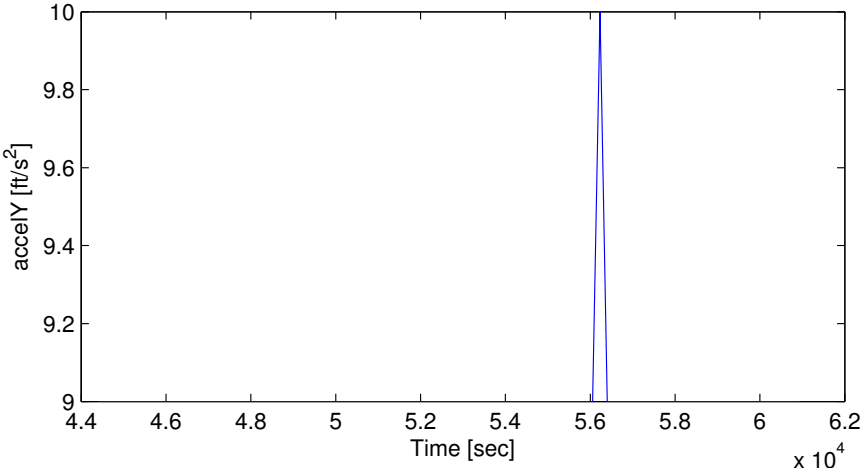


Figure E.25: accelZ vs. Time

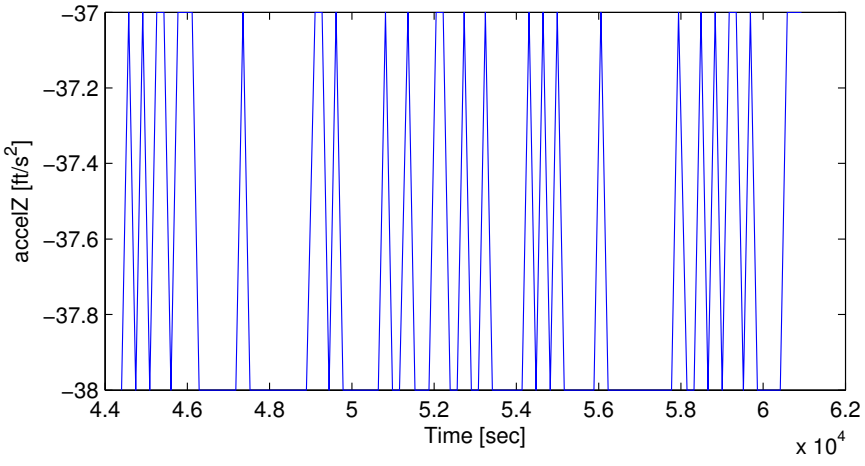


Figure E.26: gyroX vs. Time

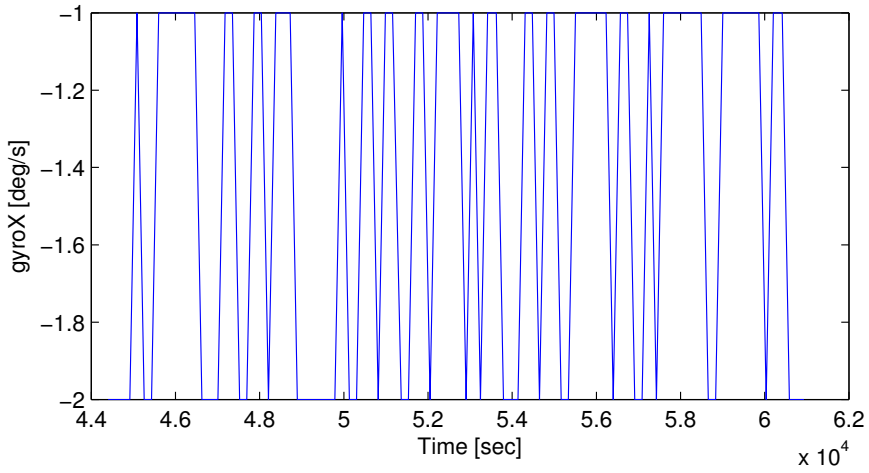


Figure E.27: gyroY vs. Time

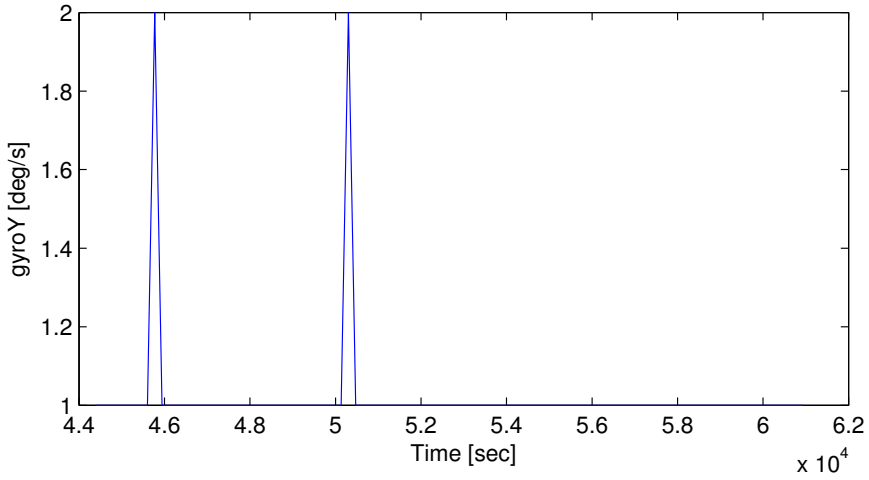


Figure E.28: gyroZ vs. Time

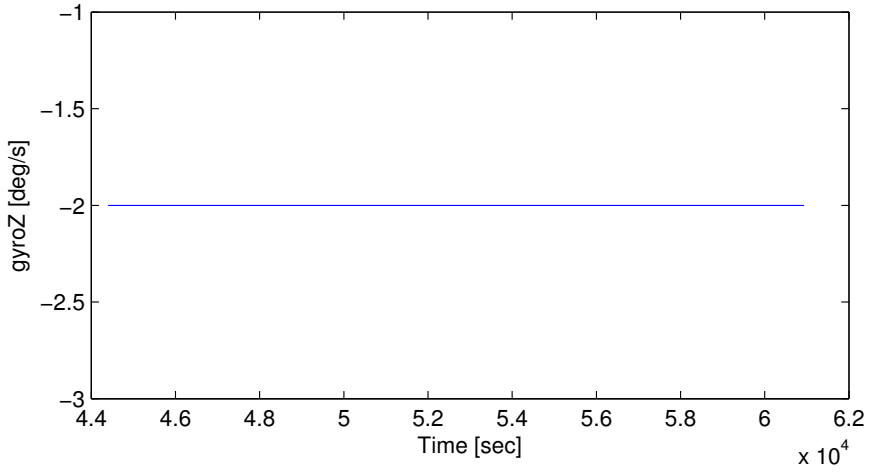


Figure E.29: magX vs. Time

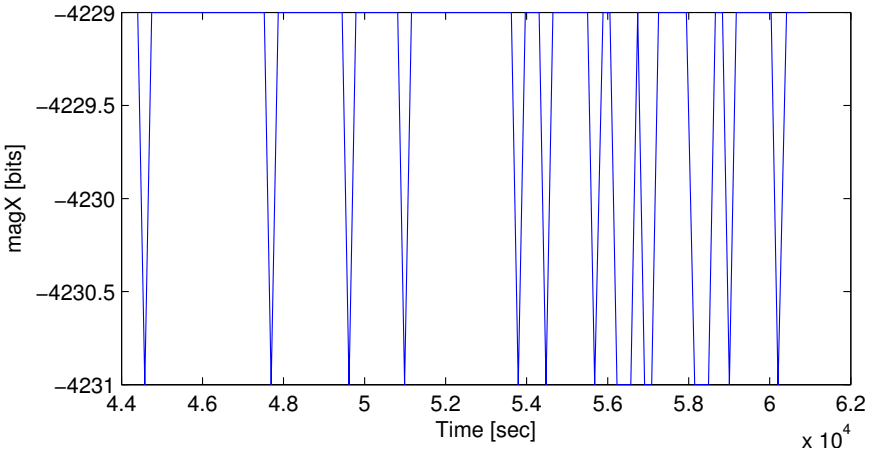
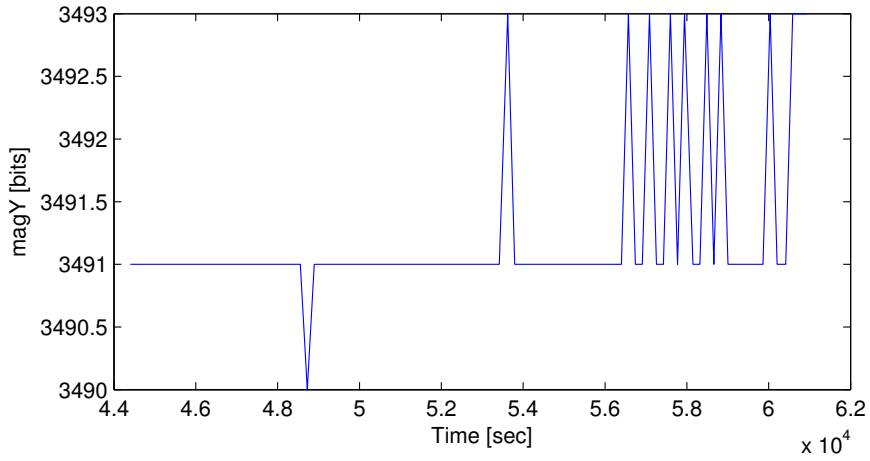


Figure E.30: magY vs. Time



E.3 Sample System Ouput - State Data

Figure E.31: magZ vs. Time

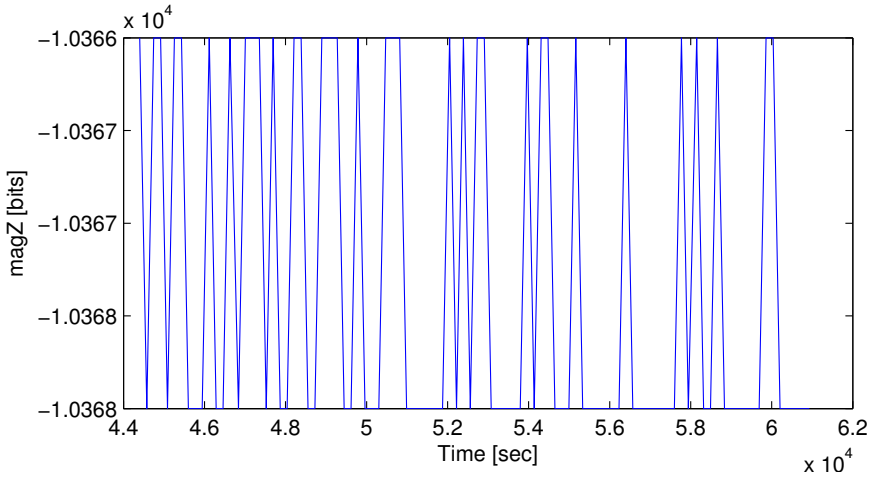


Figure E.32: press0 vs. Time

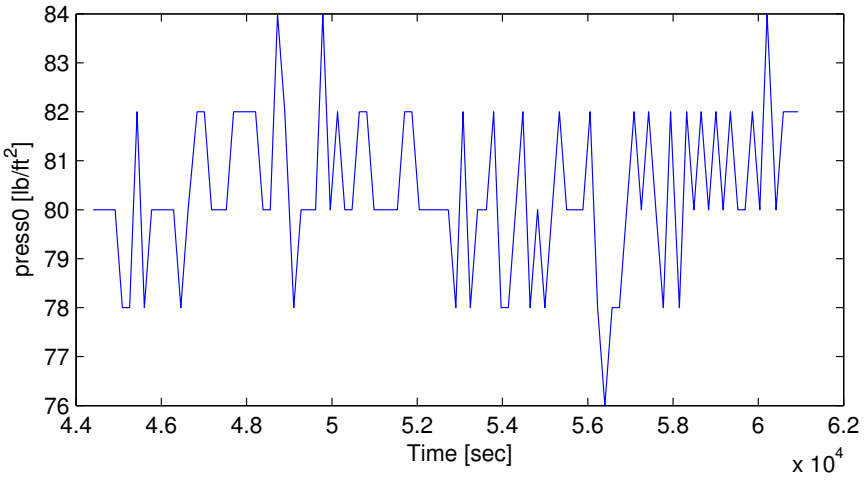


Figure E.33: press1 vs. Time

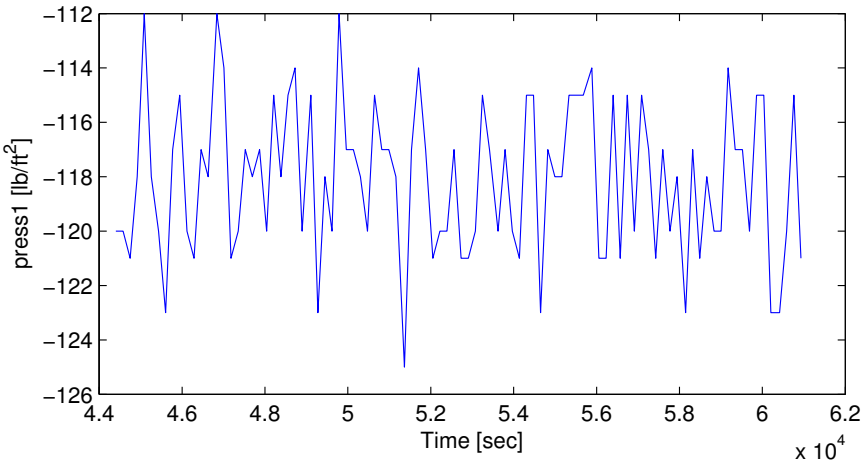


Figure E.34: press2 vs. Time

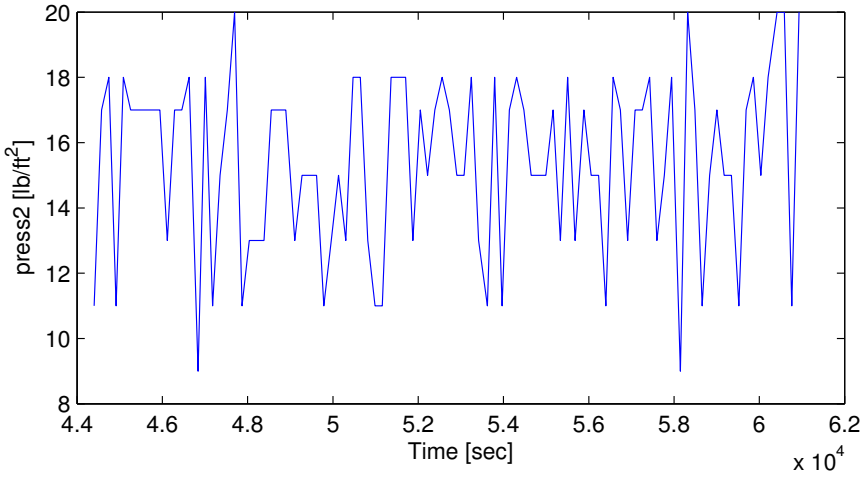


Figure E.35: press3 vs. Time

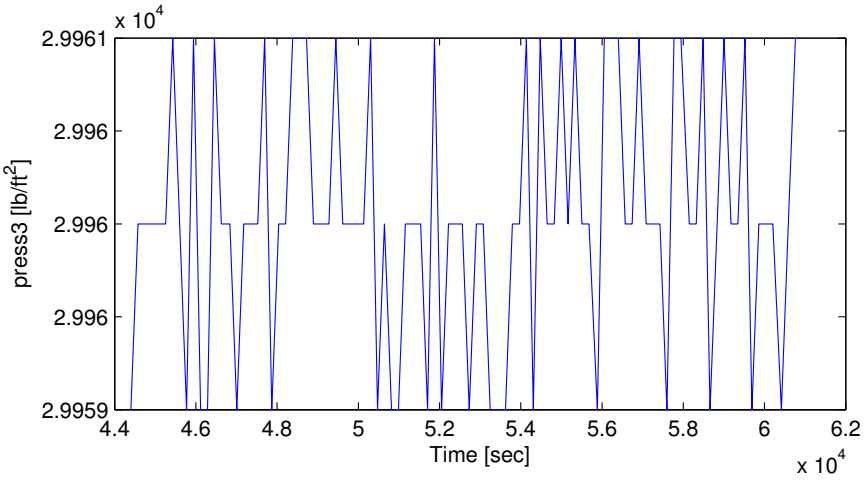


Figure E.36: gpsLat vs. Time

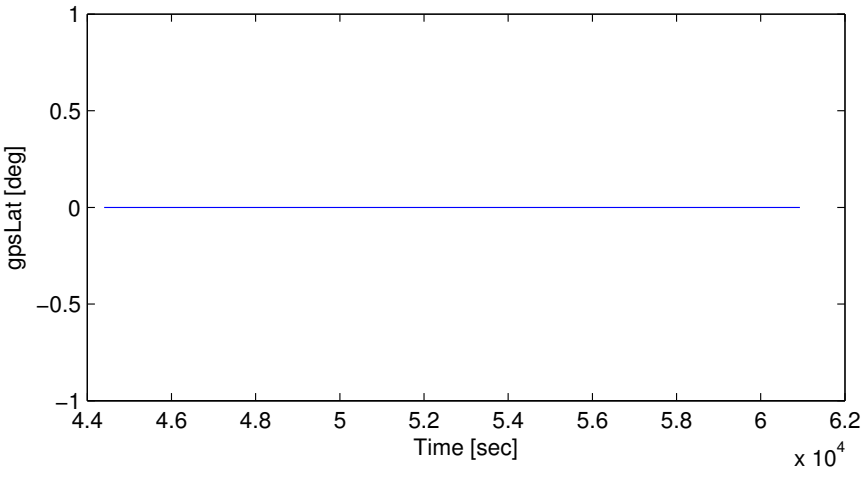


Figure E.37: gpsLong vs. Time

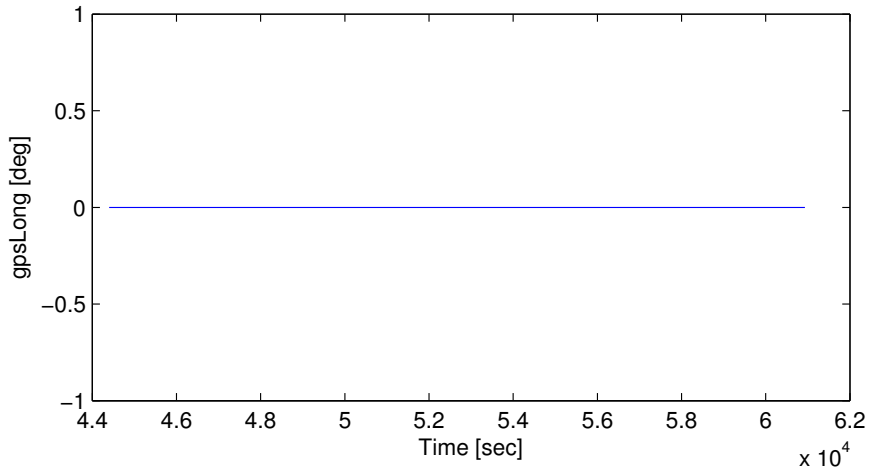


Figure E.38: gpsSpd vs. Time

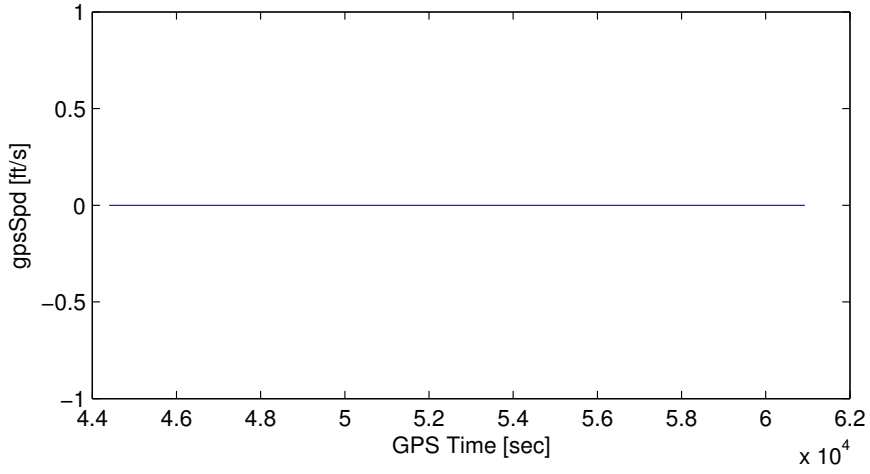


Figure E.39: gpsCrs vs. Time

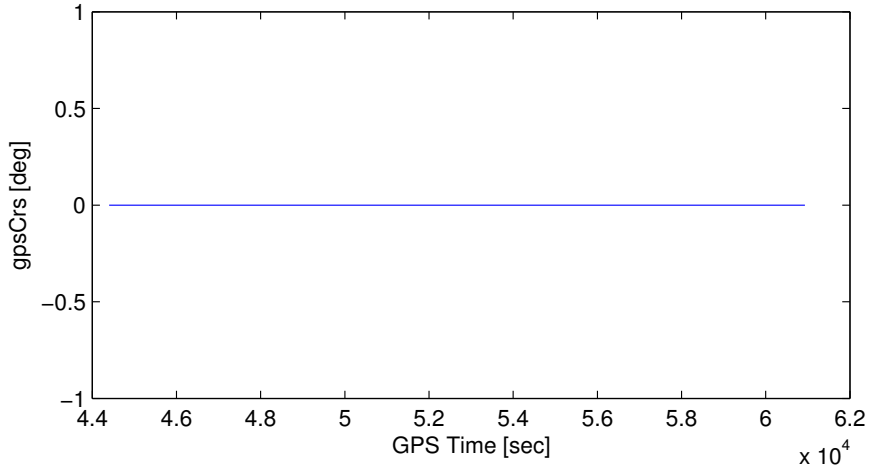


Figure E.40: date vs. Time

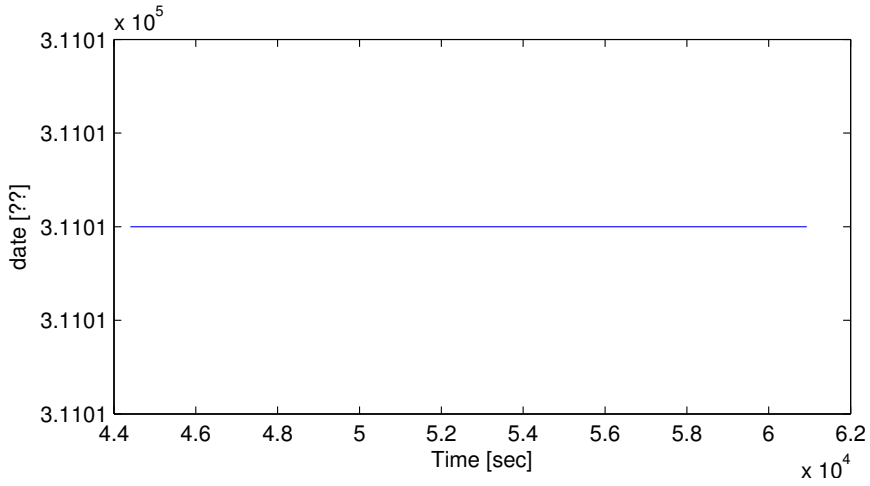


Figure E.41: CS vs. Time

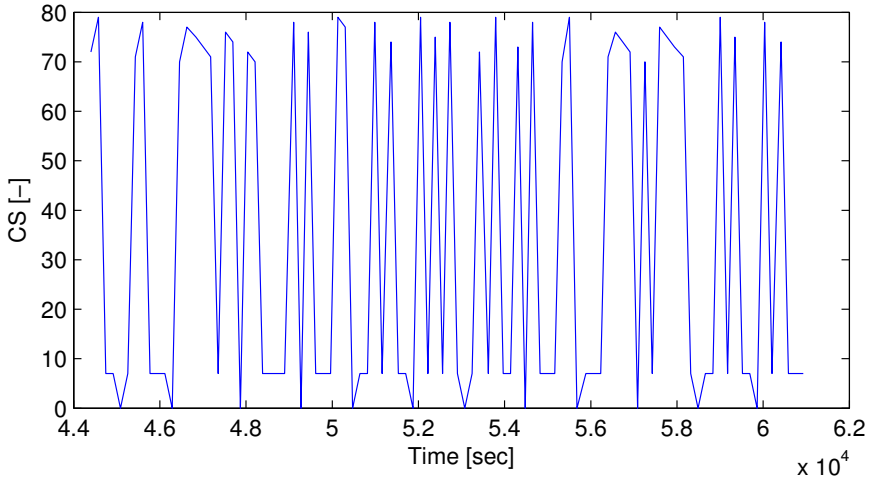


Figure E.42: temperature vs. Time

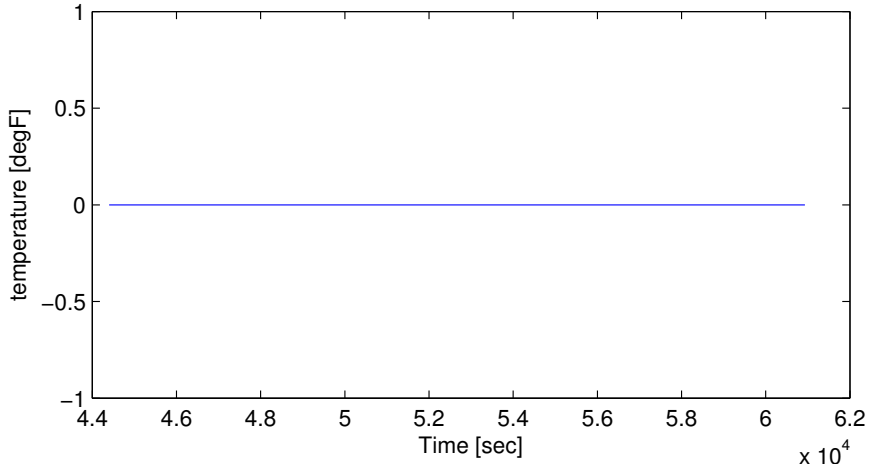


Figure E.43: deltaT vs. Time

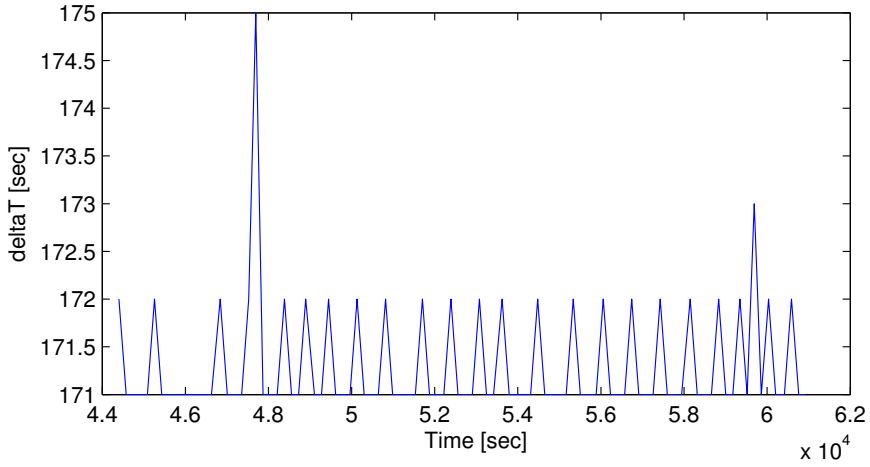


Figure E.44: rpmPwm vs. Time

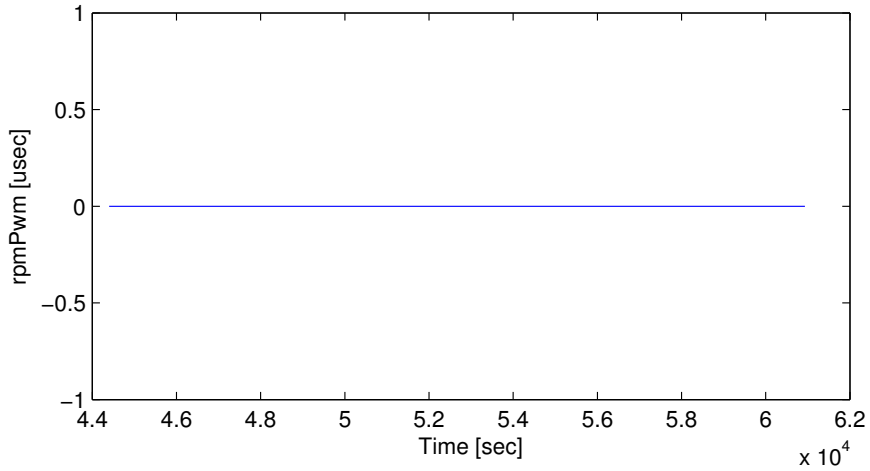


Figure E.45: ang2300X vs. Time

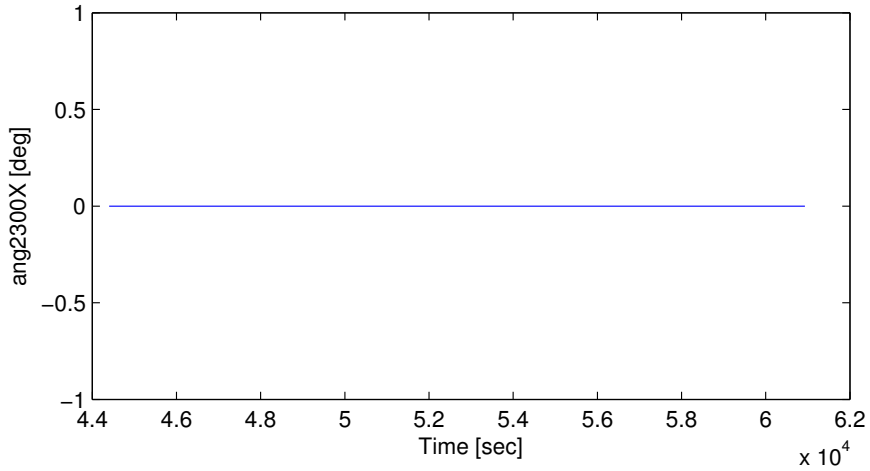


Figure E.46: ang2300Y vs. Time

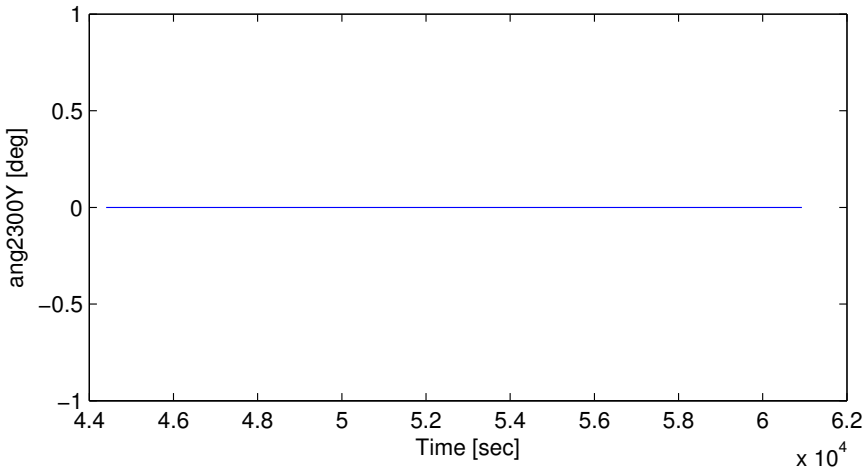


Figure E.47: ang2300Z vs. Time

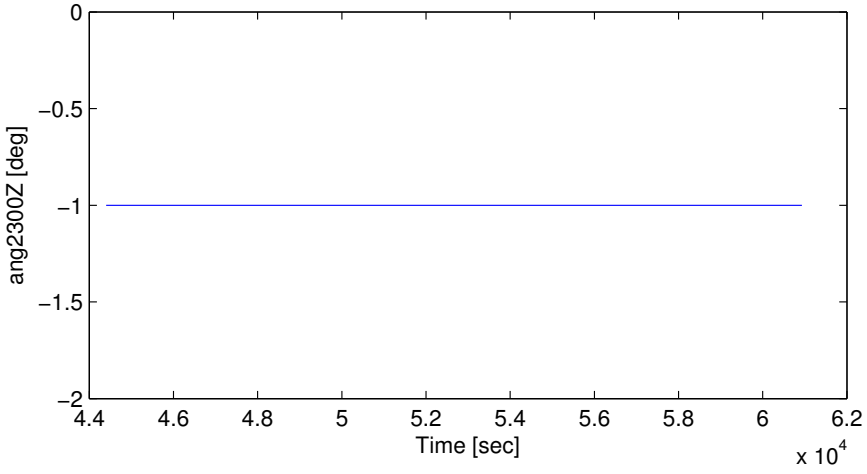


Figure E.48: ang5883X vs. Time

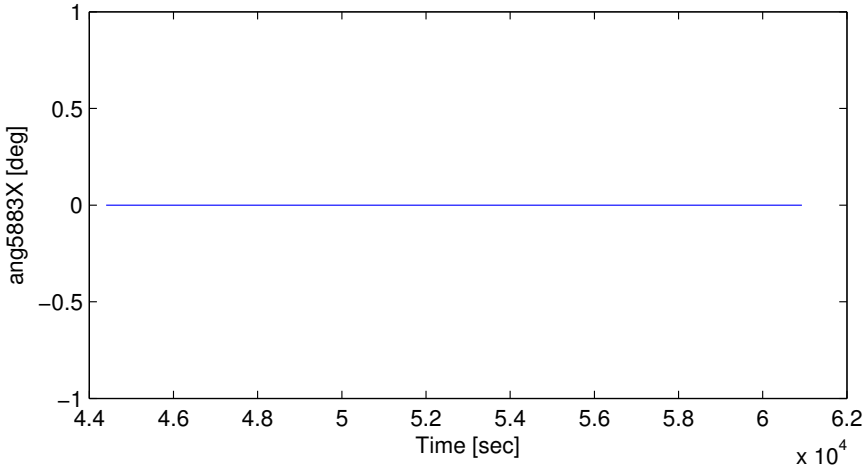


Figure E.49: ang5883Y vs. Time

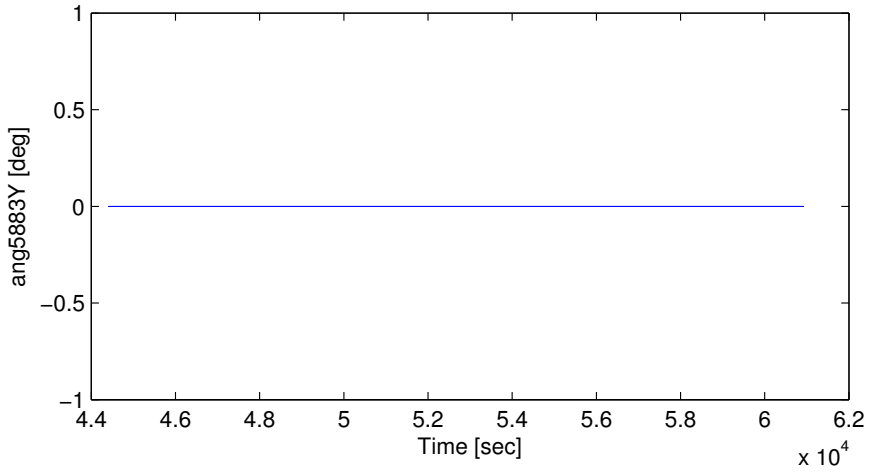


Figure E.50: ang5883Z vs. Time

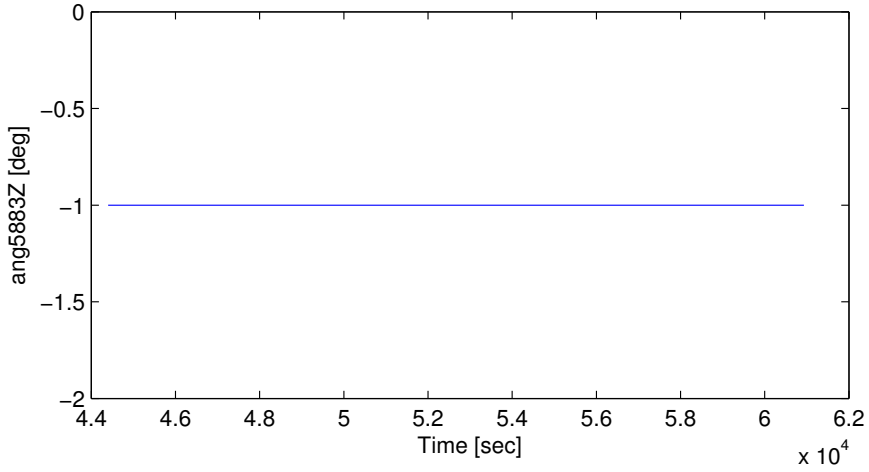


Figure E.51: qbar vs. Time

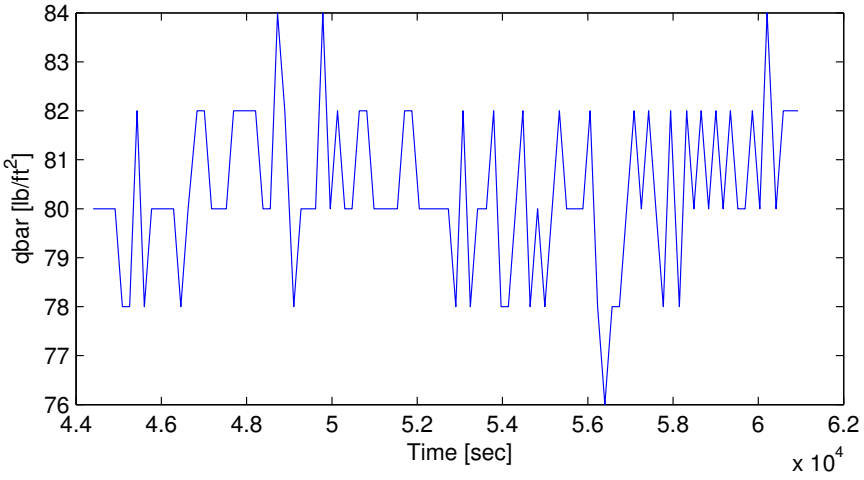


Figure E.52: rho vs. Time

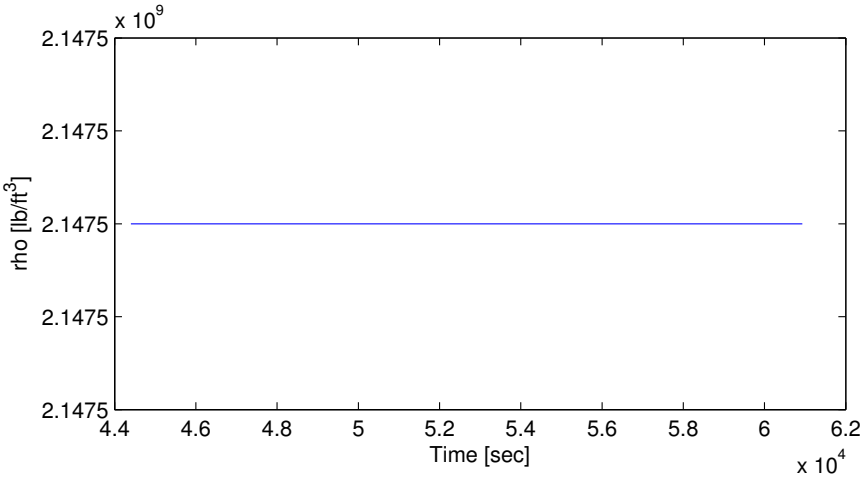


Figure E.53: alpha vs. Time

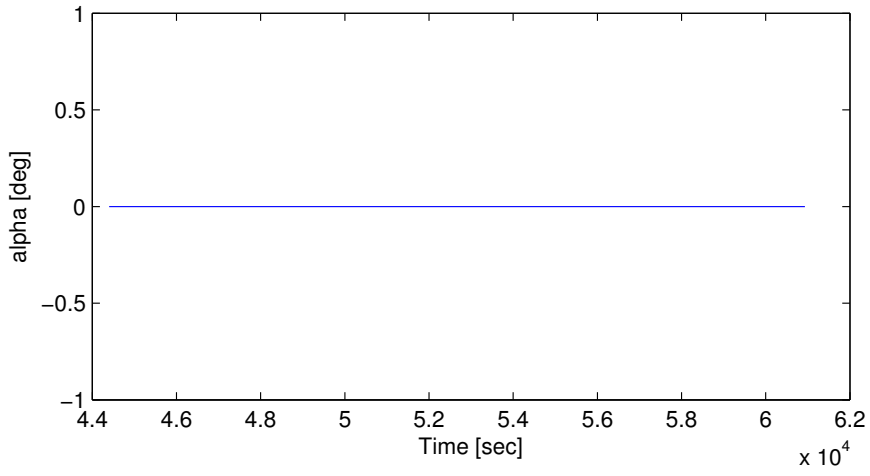
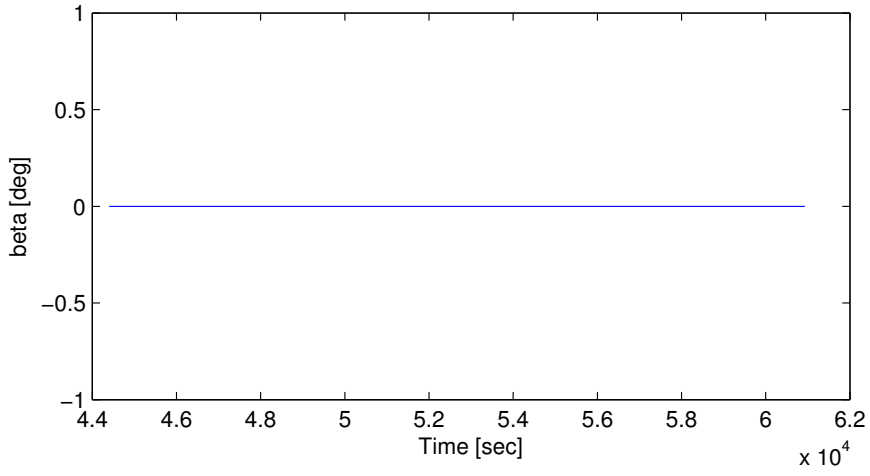


Figure E.54: beta vs. Time



E.4 Sample System Ouput - Filtered Data

Figure E.55: roll vs. Time

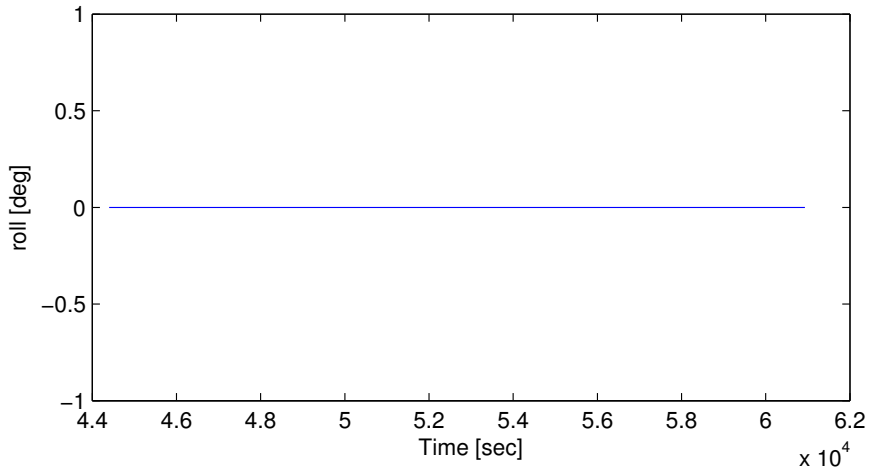


Figure E.56: pitch vs. Time

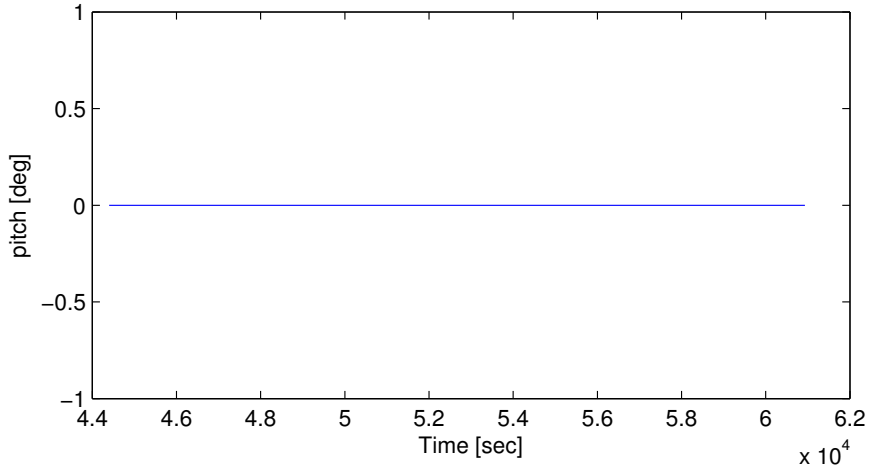


Figure E.57: yaw vs. Time

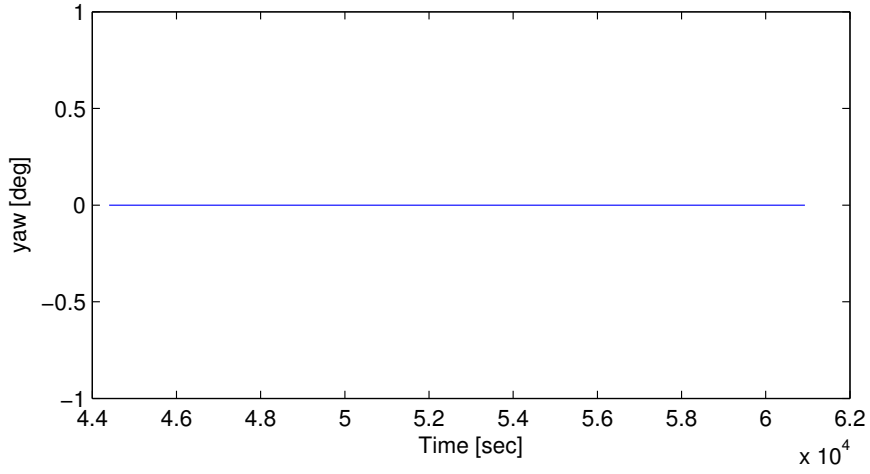


Figure E.58: rollRate vs. Time

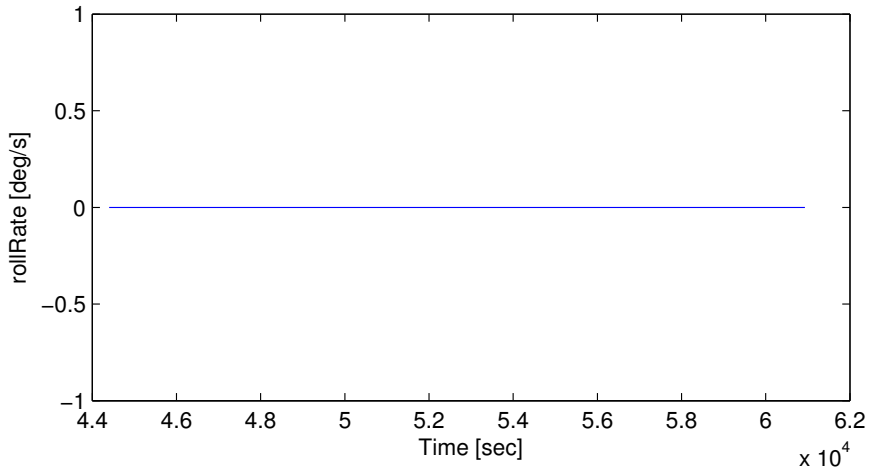


Figure E.59: pitchRate vs. Time

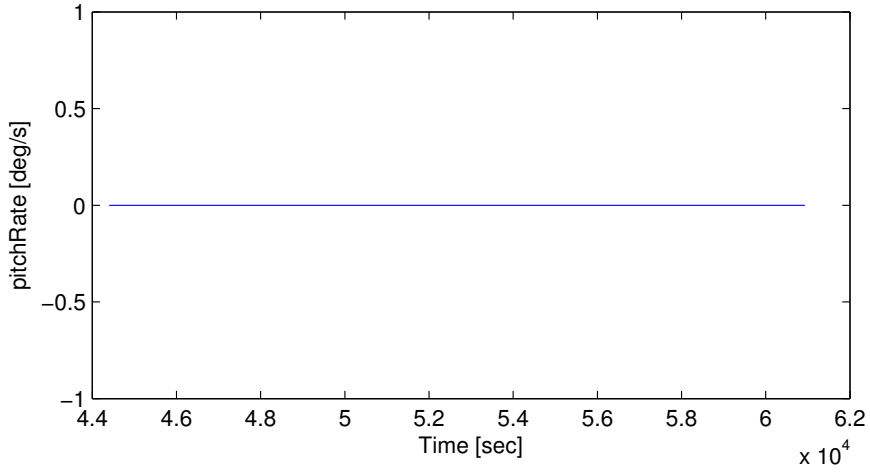


Figure E.60: yawRate vs. Time

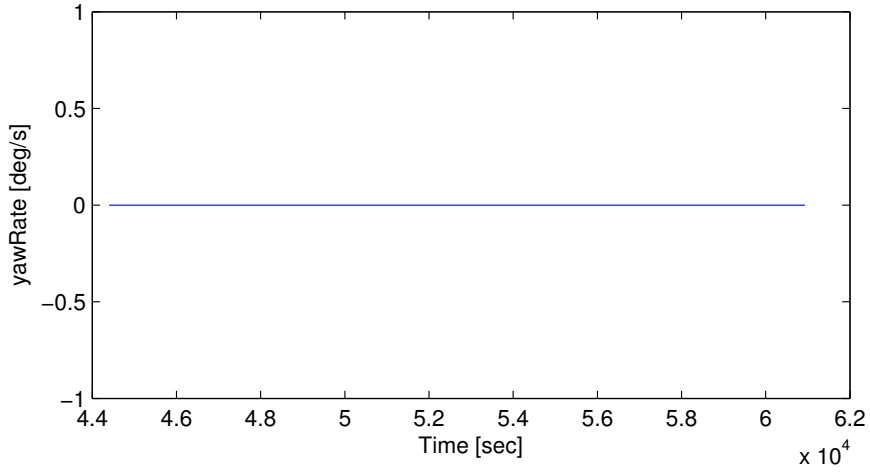


Figure E.61: accelX vs. Time

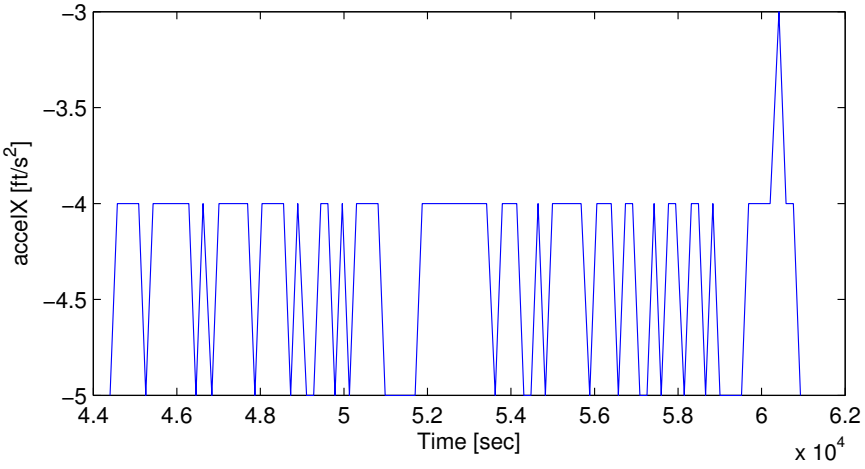


Figure E.62: accelY vs. Time

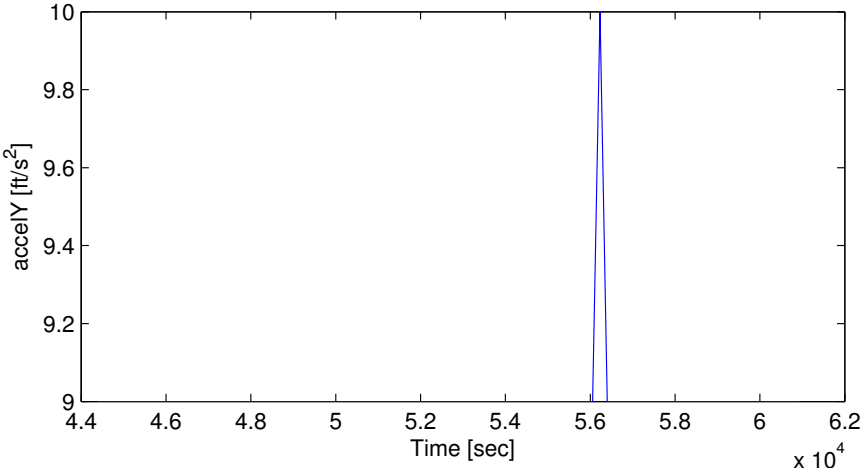


Figure E.63: accelZ vs. Time

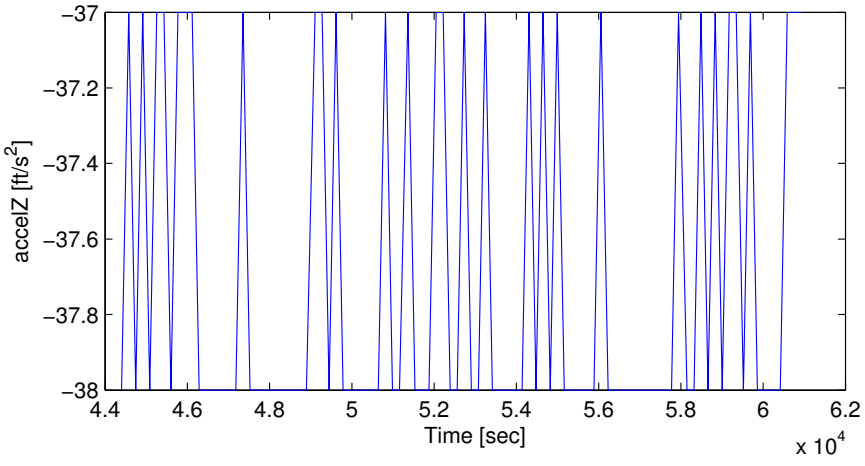


Figure E.64: qbar vs. Time

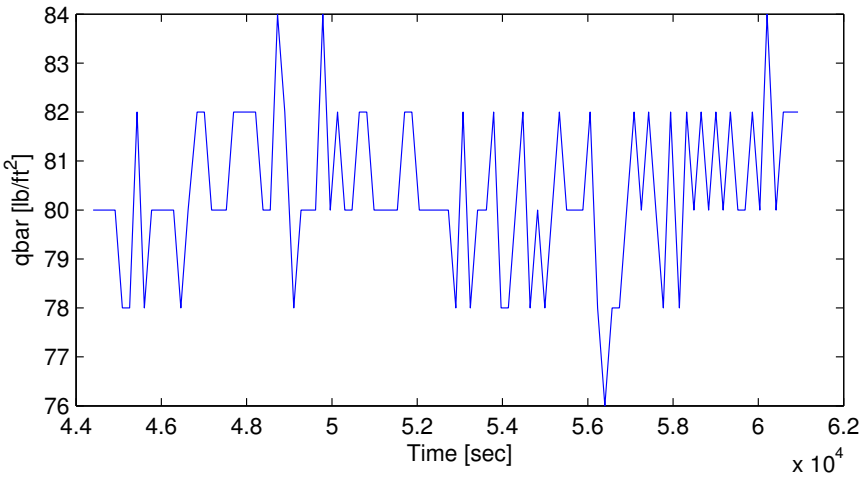


Figure E.65: rho vs. Time

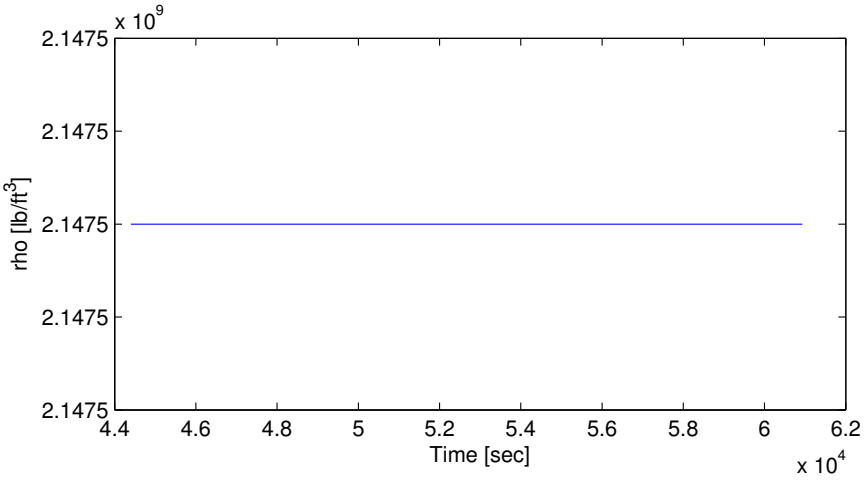


Figure E.66: alpha vs. Time

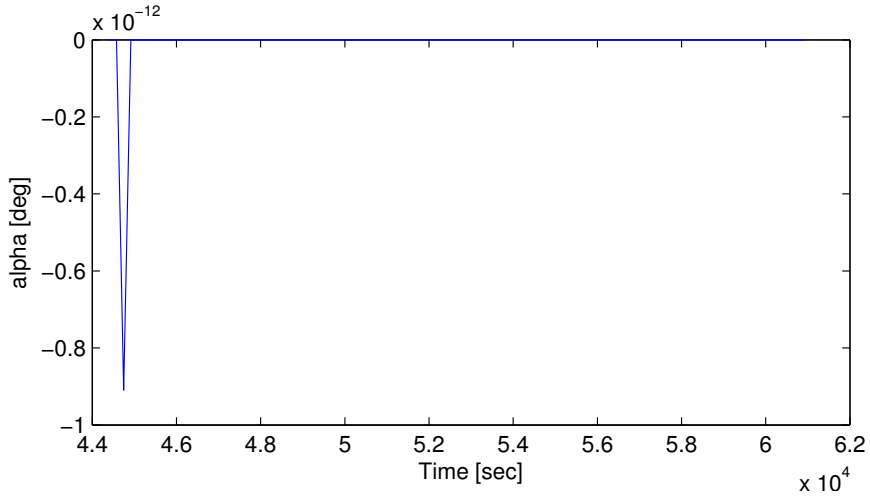


Figure E.67: beta vs. Time

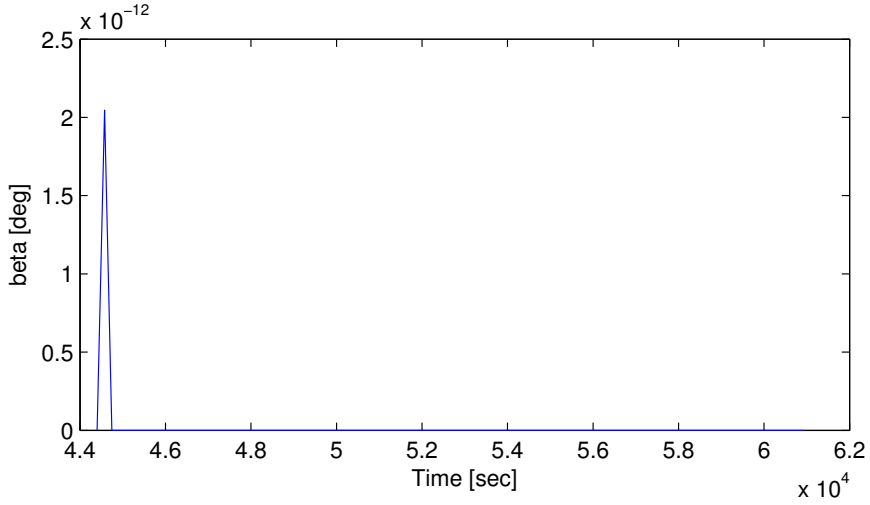


Figure E.68: roll vs. Time

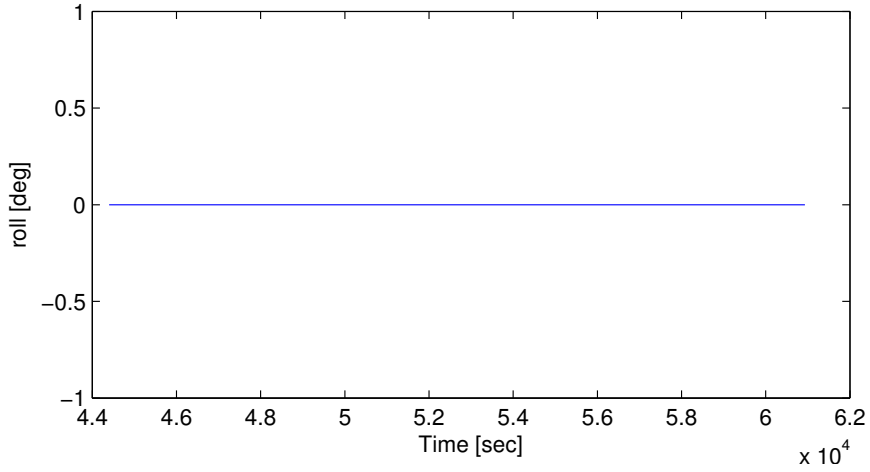


Figure E.69: pitch vs. Time

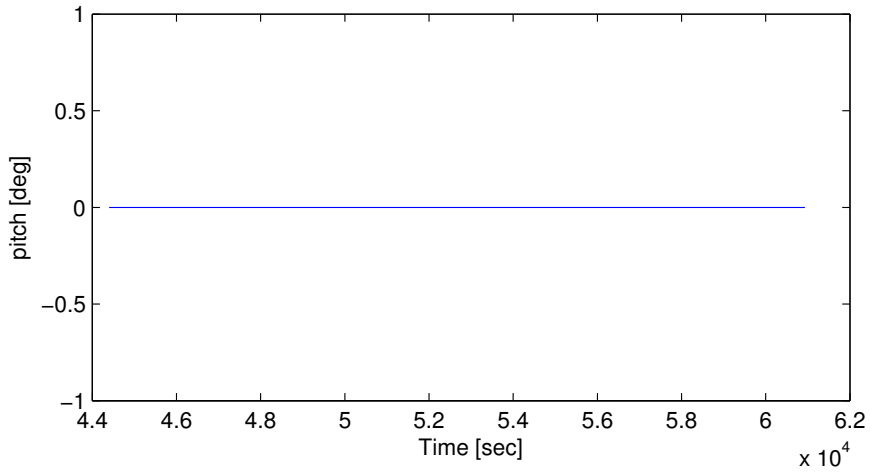


Figure E.70: yaw vs. Time

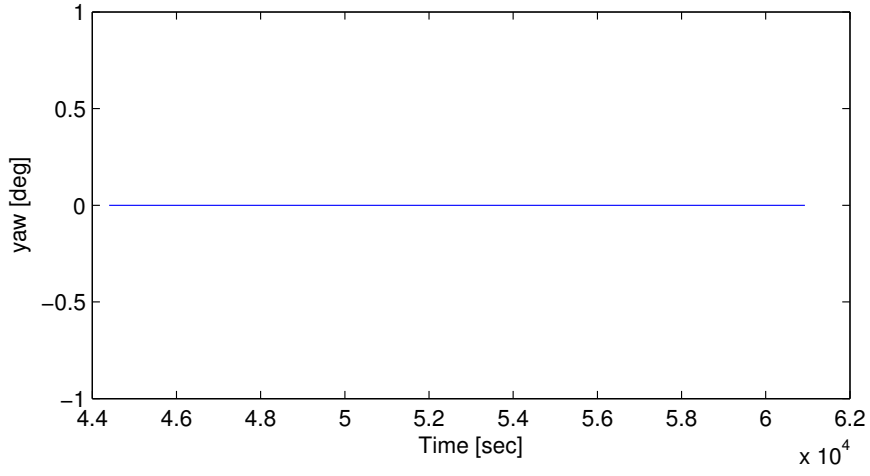


Figure E.71: rollRate vs. Time

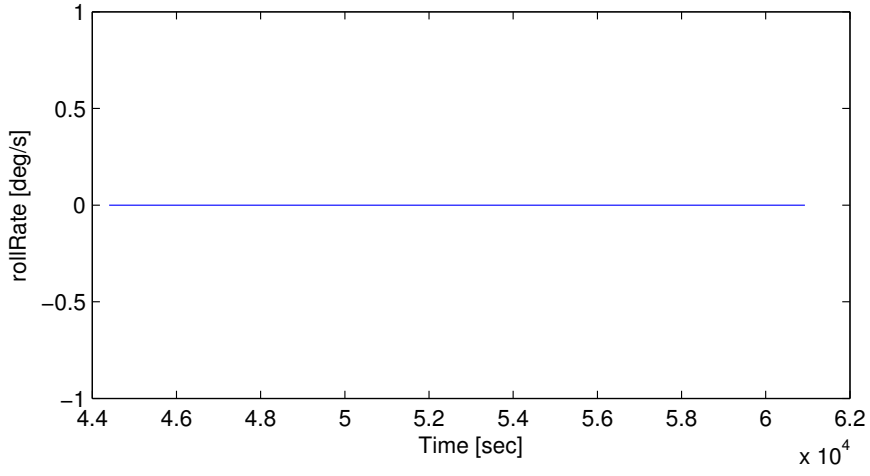


Figure E.72: pitchRate vs. Time

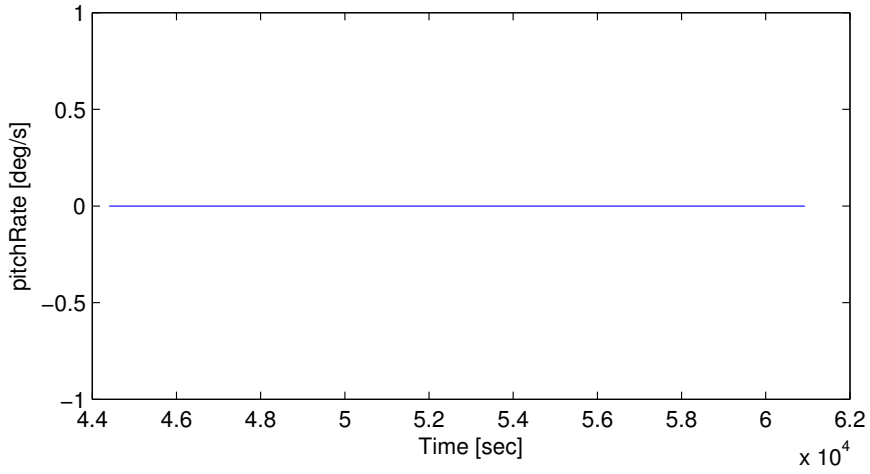


Figure E.73: yawRate vs. Time

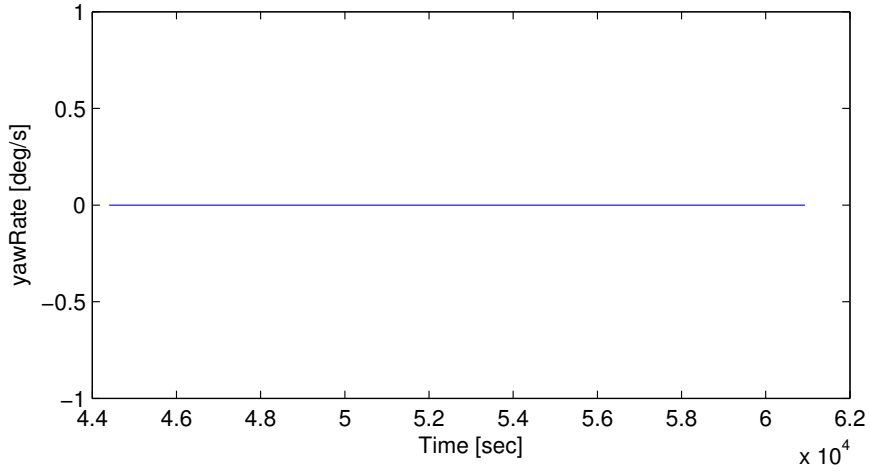


Figure E.74: accelX vs. Time

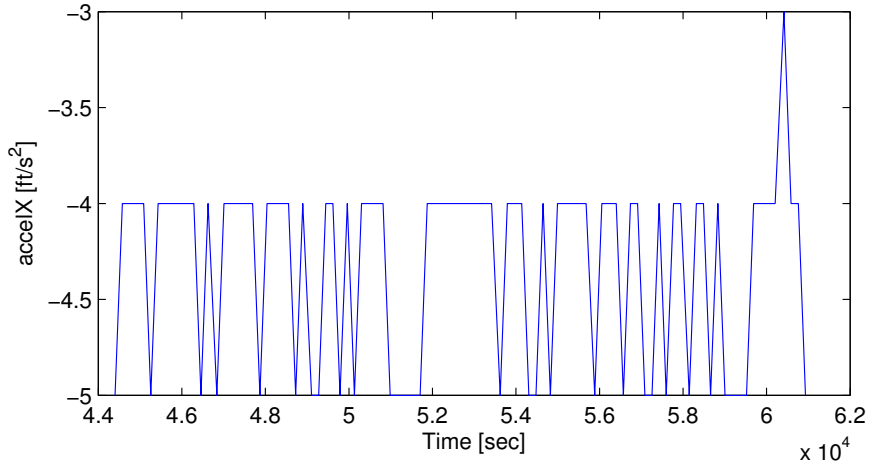


Figure E.75: accelY vs. Time

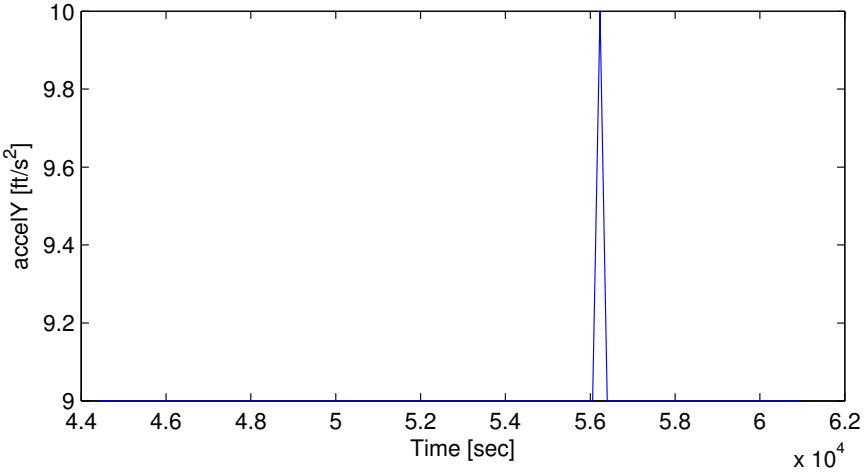
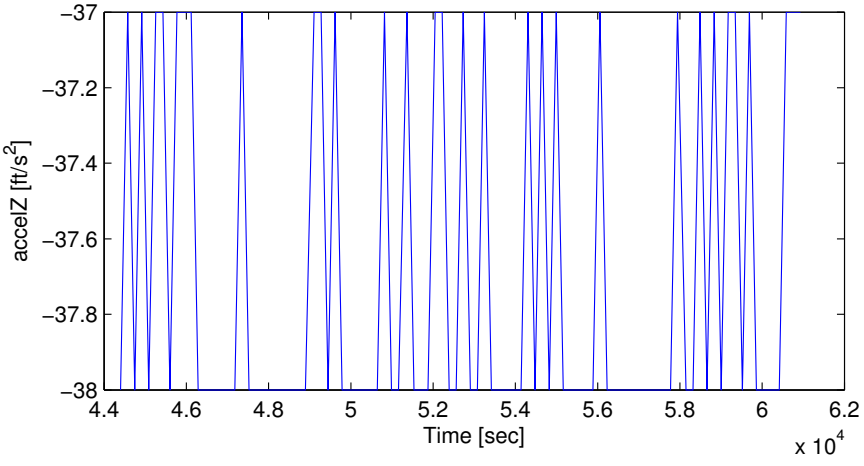


Figure E.76: accelZ vs. Time



F.0 Wiring Schematics

The following documents the wiring of the circuit boards. The actual Eagle files are available in `~/eagle/`

Figure F.1: Arduino Due Flight Data Recorder v3.20BOB Schematic

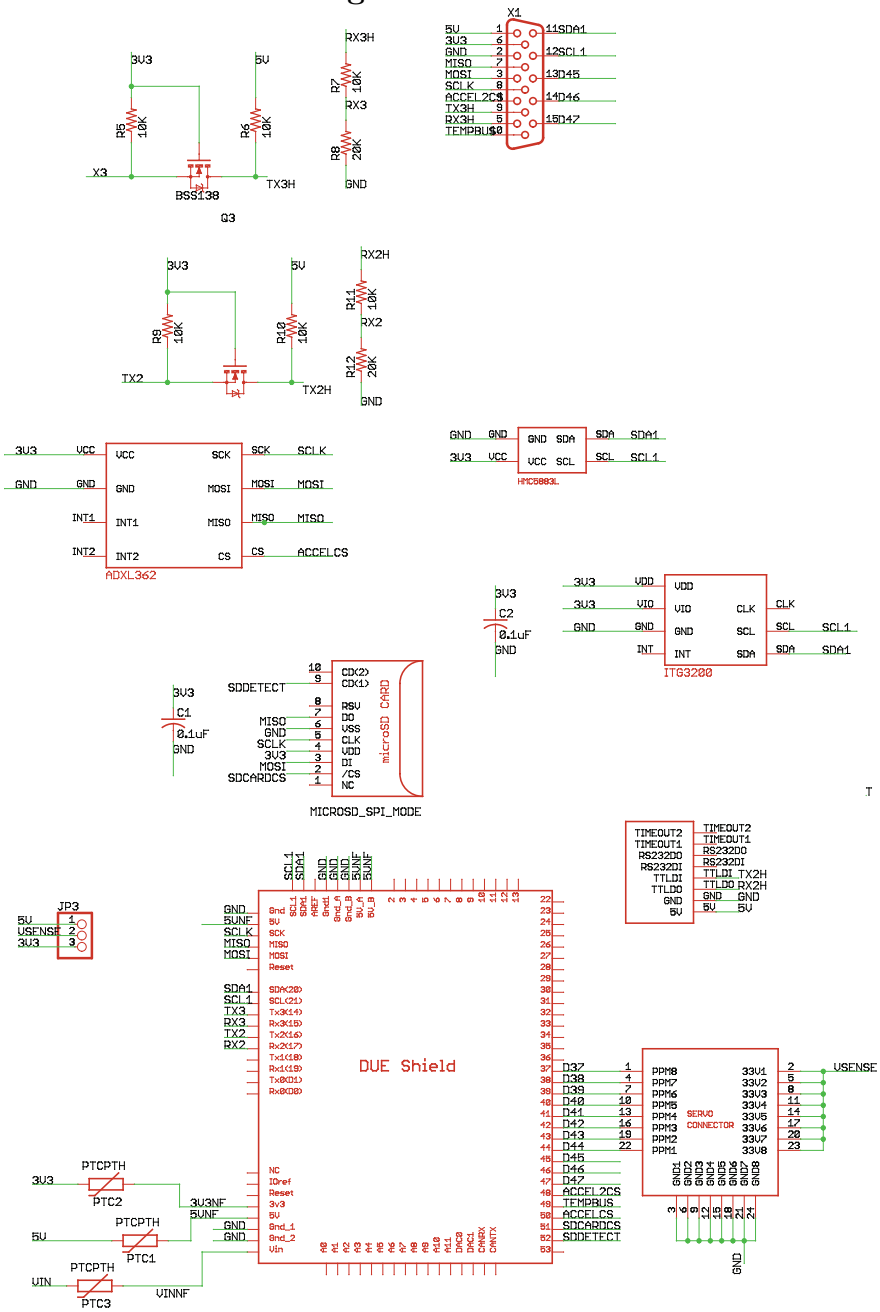


Figure F.2: Arduino Due Flight Data Recorder v3.20BOB Layout

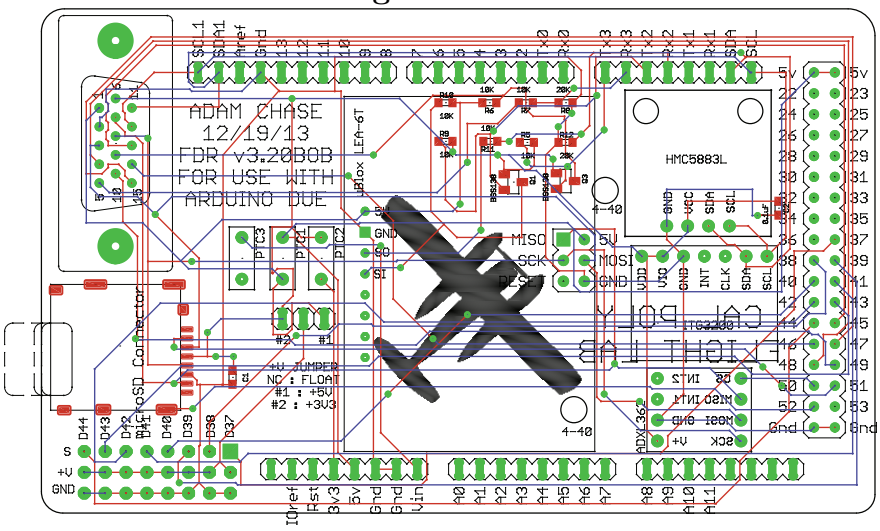


Figure F.3: Pressure Board v2.20 Schematic

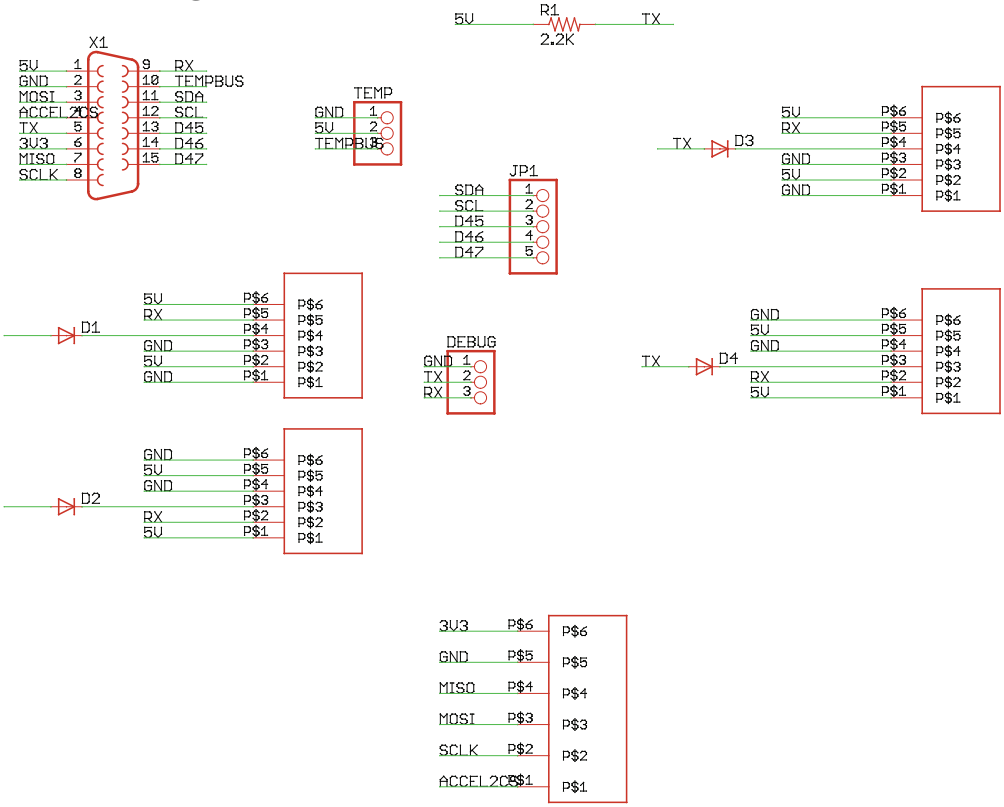
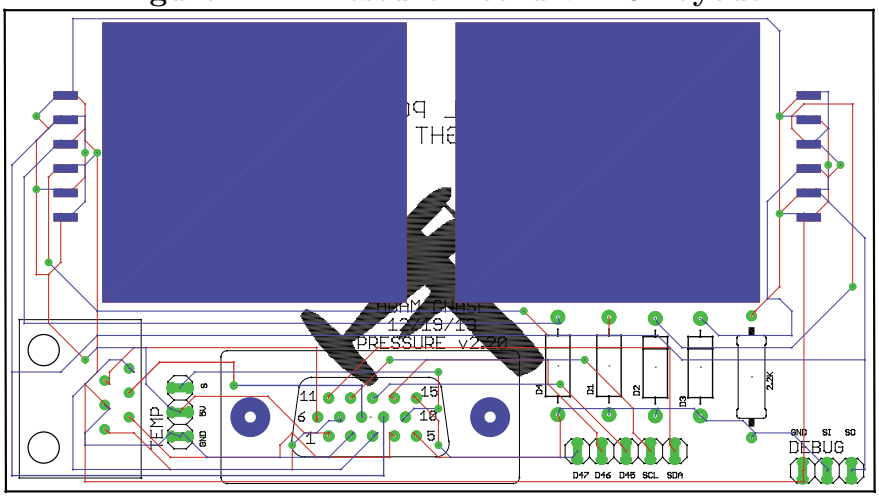


Figure F.4: Pressure Board v2.20 Layout



G.0 Source Code

G.1 Arduino Due Flight Data Recording

```
//TODO: add checking if data is actually available for all sensors, especially serial sensors
//todo:when initializing pressure, write to one and check if we get a response.
#include <Wire.h>
#include <ITG3200.h>
#include <SPI.h>
#include <SD.h>
#include <ADXL362.h>
#include <OneWire.h>
#include <DallasTemperature.h>


#define profiling 0
#define magInstalled 1
#define gpsInstalled 1
#define pressureInstalled 1
#define ONE_WIRE_BUS 49
#define tempInstalled 0
#define TEMPERATURE_PRECISION 9
#define hmcAddress 0x1E //0011110b, I2C 7bit address of HMC5883
#define accelSlaveSelect 52


#define serialBaud 19200
#define magBaud 19200
#define gpsBaud 57600
```

```
#define pressBaud 19200
#define sdChipSelect 53
#define pwmPin0 38
#define pwmPin1 39
#define pwmPin2 40
#define pwmPin3 41
#define pwmPin4 42
#define pwmPin5 43
#define pwmPin6 44
#define pwmPin7 45

 volatile  unsigned long trig0 , trig1 , trig2 , trig3 , trig4 , trig5 , trig6 , trig7=0;
 volatile  unsigned long pwm0,pwm1,pwm2,pwm3,pwm4,pwm5,pwm6,pwm7 = 0;

boolean printSerialOut = false;
boolean sdCardClosed = true;

char filename[80]={0};
File dataFile;
boolean logData=false;

char pressSN0[13] = "R11L07-20-A4";
char pressSN1[13] = "R10F30-04-A1";
char pressSN2[13] = "R11L07-20-A5";
char pressSN3[13] = "4F15-01-A213";

byte writeBuff[2048];
uint16_t writeBuffLoc=0;
```

```
const uint8_t nBytesPerSample = 111; //todo:check if this is correct

// set up communication with sensors
USARTClass &magSerial = Serial1;
USARTClass &gpsSerial = Serial2;
USARTClass &pressureSerial = Serial3;

//Constructors
ITG3200 gyro = ITG3200(); //construct gyro object
ADXL362 accel = ADXL362(accelSlaveSelect); //construct accel object

OneWire oneWire(ONE_WIRE_BUS);

// Pass oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

DeviceAddress tempDeviceAddress; // temperature sensor device address

void setup() {
        char msgID[6]="?????";
        char *gpsStatus,*nsInd,*ewInd,*mode={"?"};
        int32_t gpsLat,gpsLong,gpsSpd,gpsCrs=0;
        uint32_t utcTime,date,CS=0;
        unsigned long time=millis();

        Serial.begin(serialBaud); //begin serial communication for debugging
        Serial.println("Serial port initialized.");
```

```
        Serial.print("Data logging : ");
        if (logData)
                {
                Serial.println("on.");
                }
        else {
                Serial.println("off.");
                }
#if (magInstalled)
        Serial.println(" Initializing magnetometer...");
        serialDeviceInit(Serial, magSerial, magBaud,"mag"); //initialize magnetometer on
            Serial1
#endif

#if(gpsInstalled)
        Serial.println(" Initializing GPS...");
        serialDeviceInit(Serial, gpsSerial, gpsBaud,"gps"); //initialize gps on Serial2
        gpsSerial.setTimeout(50);
#endif

#if (pressureInstalled)
        Serial.println(" Initializing pressure sensors ... ");
        serialDeviceInit(Serial, pressureSerial, pressBaud,"pre"); // initialize pressure
            sensors on Serial3
        pressureSerial.setTimeout(50);
#endif

#if (gpsInstalled)
```

```
        while (!gpsSerial.available()>0)
                {
                if (millis()>time+5000)
                        {
                        break;
                        }
                }
        readGPS(gpsSerial,msgID,utcTime,&gpsStatus,
            gpsLat,&nsInd,gpsLong,&ewInd,gpsSpd,gpsCrs,date,&mode,CS);
        Serial.println(filename);
        char str[20];
        sprintf(str,"%d",utcTime);
        strcpy(filename,str);
        strcat(filename,".bin");
        Serial.println(filename);
#else
        Serial.print("No GPS installed. Filename : ");
        char str[20];
        sprintf(str,"%d",rand()%10000000);
        strcpy(filename,str);
        strcat(filename,".bin");
        Serial.println(filename);
#endif


        //set up gyro
        Wire.begin(); //begin wire transmission
        delay(1000); //todo:check if this is necessary
        gyro.init(ITG3200_ADDR_AD0_HIGH); //initialize gyro with address pulled high
```

```
        //set up accelerometer
        accel.begin();
        accel.beginMeasure();

        //set up HMC5883L magnetometer
        Wire.beginTransmission(hmcAddress); //open communication with HMC5883
        Wire.write(0x02); //select mode register
        Wire.write(0x00); //continuous measurement mode
        Wire.endTransmission();

#if (tempInstalled)
        //set up temp sensor
        sensors.begin();
        // Serial.println("Temperature sensor set up : ");

        if (sensors.getAddress(tempDeviceAddress, 0))
                {
                sensors.setResolution(tempDeviceAddress, TEMPERATURE_PRECISION);
                }
        else Serial.println("No temperature sensors found.");

#else
        Serial.println("Temperature sensor not installed.");
#endif

        pinMode(pwmPin0, INPUT);
        pinMode(pwmPin1, INPUT);
```

```
        pinMode(pwmPin2, INPUT);
        pinMode(pwmPin3, INPUT);
        pinMode(pwmPin4, INPUT);
        pinMode(pwmPin5, INPUT);
        pinMode(pwmPin6, INPUT);
        pinMode(pwmPin7, INPUT);

        attachInterrupt(pwmPin0,intHandler0,CHANGE);
        attachInterrupt(pwmPin1,intHandler1,CHANGE);
        attachInterrupt(pwmPin2,intHandler2,CHANGE);
        attachInterrupt(pwmPin3,intHandler3,CHANGE);
        attachInterrupt(pwmPin4,intHandler4,CHANGE);
        attachInterrupt(pwmPin5,intHandler5,CHANGE);
        attachInterrupt(pwmPin6,intHandler6,CHANGE);
        attachInterrupt(pwmPin7,intHandler7,CHANGE);

#if (gpsInstalled)
        //flush any data that has accumulated between turning on the device and starting the
            main loop.
                {
                while(gpsSerial.available()>0)
                        {
                        gpsSerial.read();
                        }
                }
#endif
                Serial.println("System initialized, ready for commands.");
        }
```

```
void loop() {
        byte buff4 [4];
        byte buff2 [2];
        int16_t pressure [4]; //pressure sensor data
        int16_t temperature=0;
        int16_t magReading[3]; //magnetometer data
        int16_t hmcReading[3]; //hmc5883L mag data
        int16_t gyroX, gyroY, gyroZ;
        int16_t accelX, accelY, accelZ, accelT;
        char msgID[6]="?????";
        char *gpsStatus={"?"},*nsInd={"?"},*ewInd={"?"},*mode={"?"};
        int32_t gpsLat,gpsLong,gpsSpd,gpsCrs=0;
        uint32_t utcTime,date,CS=0;

        if (Serial.available()>0) //check if there's serial data available
                {
                parseInput();
                }

        //start timer for tDiff data
        unsigned long time = millis();
        //read accel data
        readAccelData(accelX,accelY,accelZ);
        //read gyro data
        readGyroData(gyroX,gyroY,gyroZ);
        //read HMC5883L magnetometer
```

```
        readHMC(hmcReading);


        //read magnetometer data
#if (magInstalled)
        readMagnetometer(magSerial,magReading);
#endif


        //read pressure transducers
#if (pressureInstalled)
        readAllPress (pressureSerial ,pressSN0,pressSN1,pressSN2,pressSN3,pressure);
#endif


        //read GPS
#if (gpsInstalled)
        if (gpsSerial.available()>0)
                {
  //check ascii write time and speed, it might be better to bite the bullet on them
  //check if the gps message is a GPRMC message. If it isn't, don't write it.
                readGPS(gpsSerial,msgID,utcTime,&gpsStatus,
                    gpsLat,&nsInd,gpsLong,&ewInd,gpsSpd,gpsCrs,date,&mode,CS);
                }
#endif


#if (tempInstalled)
        sensors.requestTemperatures(); // Send the command to get temperatures
        temperature = int16_t (100*sensors.getTempF(tempDeviceAddress));
#endif
```

```
//format packet to be sent to SD card

unsigned long tDiff = millis()-time; //end tDiff timer

//write to SD card
if (logData) //if we're storing data
        {
        unsigned long profile = micros();
        if (writeBuffLoc + nBytesPerSample > sizeof(writeBuff)) //if there's not
            enough room in the buffer, right it to the card first
                {
                File dataFile = SD.open(filename,FILE_WRITE);
                int bytesWritten = dataFile.write(writeBuff,writeBuffLoc);
                Serial.println();
                if (bytesWritten>0)
                        Serial.println("Data buffer written to SD card.");
                else
                        Serial.println("Error : Data buffer NOT written to SD card.");
                Serial.println();
                dataFile.close();
                writeBuffLoc = 0;
                }
        parseToBinUInt32(writeBuff,time,writeBuffLoc); //split data into bytes for
            binary storage
        parseToBinInt16(writeBuff,accelX,writeBuffLoc);
        parseToBinInt16(writeBuff,accelY,writeBuffLoc);
        parseToBinInt16(writeBuff,accelZ,writeBuffLoc);
        parseToBinInt16(writeBuff,gyroX,writeBuffLoc);
```

```
parseToBinInt16(writeBuff,gyroY,writeBuffLoc);
parseToBinInt16(writeBuff,gyroZ,writeBuffLoc);
parseToBinInt16(writeBuff,magReading[0],writeBuffLoc);
parseToBinInt16(writeBuff,magReading[1],writeBuffLoc);
parseToBinInt16(writeBuff,magReading[2],writeBuffLoc);
parseToBinInt16(writeBuff,hmcReading[0],writeBuffLoc);
parseToBinInt16(writeBuff,hmcReading[1],writeBuffLoc);
parseToBinInt16(writeBuff,hmcReading[2],writeBuffLoc);
parseToBinInt16(writeBuff,pressure[0],writeBuffLoc);
parseToBinInt16(writeBuff,pressure[1],writeBuffLoc);
parseToBinInt16(writeBuff,pressure[2],writeBuffLoc);
parseToBinInt16(writeBuff,pressure[3],writeBuffLoc);
for (int i=0;i<5;i++)
        {
        writeBuff[writeBuffLoc++] = byte (msgID[i]);
        }
parseToBinUInt32(writeBuff,utcTime,writeBuffLoc);
writeBuff[writeBuffLoc++] = (byte) gpsStatus[0];
parseToBinInt32(writeBuff,gpsLat,writeBuffLoc);
writeBuff[writeBuffLoc++] = (byte) *nsInd;
parseToBinInt32(writeBuff,gpsLong,writeBuffLoc);
writeBuff[writeBuffLoc++] = (byte) *ewInd;
parseToBinInt32(writeBuff,gpsSpd,writeBuffLoc);
parseToBinInt32(writeBuff,gpsCrs,writeBuffLoc);
parseToBinUInt32(writeBuff,date,writeBuffLoc);
writeBuff[writeBuffLoc++] = (byte) *mode;
parseToBinUInt32(writeBuff,CS,writeBuffLoc);
parseToBinInt16(writeBuff,temperature,writeBuffLoc);
```

```
                parseToBinUInt32(writeBuff,pwm0,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm1,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm2,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm3,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm4,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm5,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm6,writeBuffLoc);
                parseToBinUInt32(writeBuff,pwm7,writeBuffLoc);
                parseToBinUInt32(writeBuff,tDiff,writeBuffLoc);
#if ( profiling )
                Serial.print("parsing into binary : ");
                Serial.print(micros()-profile);
                Serial.println(" usec");
                profile = micros();
#endif
                }

        else if(writeBuffLoc != 0) //we're not logging data, but theres data in the buffer.
            write it out to SD
                {
                unsigned long profile = micros();
                dataFile = SD.open(filename,FILE_WRITE);
#if ( profiling )
                Serial.print("opening SD : ");
                Serial.print(micros()-profile);
                Serial.println(" usec");
                profile = micros();
#endif
```

```
                int bytesWritten = dataFile.write(writeBuff,writeBuffLoc);
#if (profiling)
                Serial.print("writing to SD : ");
                Serial.print(micros()-profile);
                Serial.println(" usec");
                profile = micros();
#endif
                Serial.println();
                if (bytesWritten>0)
                        {
                        Serial.println("Data buffer written to SD card.");
                        }
                else
                        {
                        Serial.println("ERROR : Data buffer NOT written to SD card.");
                        }
                Serial.println();
                dataFile.close();
                writeBuffLoc = 0;
                }
        if (printSerialOut) //if the user wants to see the data (should be off by default for
            increased speed)
                {
                Serial.print(time);
                Serial.print('\t');
                Serial.print(accelX);
                Serial.print('\t');
                Serial.print(accelY);
```

```
Serial.print('\t');
Serial.print(accelZ);
Serial.print('\t');
Serial.print(gyroX);
Serial.print('\t');
Serial.print(gyroY);
Serial.print('\t');
Serial.print(gyroZ);
Serial.print('\t');
Serial.print(magReading[0]);
Serial.print('\t');
Serial.print(magReading[1]);
Serial.print('\t');
Serial.print(magReading[2]);
Serial.print('\t');
Serial.print(hmcReading[0]);
Serial.print('\t');
Serial.print(hmcReading[1]);
Serial.print('\t');
Serial.print(hmcReading[2]);
Serial.print('\t');
Serial.print(pressure[0]);
Serial.print('\t');
Serial.print(pressure[1]);
Serial.print('\t');
Serial.print(pressure[2]);
Serial.print('\t');
Serial.print(pressure[3]);
```

```
Serial.print('\t');
Serial.print(msgID);
Serial.print('\t');
Serial.print(utcTime);
Serial.print('\t');
Serial.print(gpsStatus);
Serial.print('\t');
Serial.print(gpsLat);
Serial.print('\t');
Serial.print(nsInd);
Serial.print('\t');
Serial.print(gpsLong);
Serial.print('\t');
Serial.print(ewInd);
Serial.print('\t');
Serial.print(gpsSpd);
Serial.print('\t');
Serial.print(gpsCrs);
Serial.print('\t');
Serial.print(date);
Serial.print('\t');
Serial.print(mode);
Serial.print('\t');
Serial.print(CS);
Serial.print('\t');
Serial.print(temperature);
Serial.print('\t');
Serial.print(tDiff);
```

```
                Serial.print('\t');
                Serial.print(pwm0);
                                Serial.print('\t');
                Serial.print(pwm1);
                                Serial.print('\t');
                Serial.print(pwm2);
                                Serial.print('\t');
                Serial.print(pwm3);
                                Serial.print('\t');
                Serial.print(pwm4);
                                Serial.print('\t');
                Serial.print(pwm5);
                                Serial.print('\t');
                Serial.print(pwm6);
                Serial.print('\t');
                Serial.println(pwm7);
                }
        }


void readMagnetometer(USARTClass &magSerial,int16_t *magReading)
        {
        unsigned long profile = micros();
        magSerial.print("*99P\r");
        if (magSerial.available()>0)
                {
                byte buff[6];
                for (int i=0;i<7;i++)
                        {
```

```
                    buff[i]=magSerial.read();
                    }
                magReading[0] = buff[1] + (buff[0] << 8);
                magReading[1] = buff[3] + (buff[2] << 8);
                magReading[2] = buff[5] + (buff[4] << 8);
                }
#if ( profiling )
        Serial.print("magnetometer : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
#endif
        }
void readHMC(int16_t *hmcReading)
        {
        unsigned long profile = micros();

        //Tell the HMC5883 where to begin reading data
        Wire.beginTransmission(hmcAddress);
        Wire.write(0x03); //select register 3, X MSB register
        Wire.endTransmission();



        //Read data from each axis, 2 registers per axis
        Wire.requestFrom(hmcAddress, 6);
        if(6<=Wire.available()){
                hmcReading[0] = Wire.read()<<8; //X msb
                hmcReading[0] |= Wire.read(); //X lsb
                hmcReading[1] = Wire.read()<<8; //Z msb
```

```
                hmcReading[1] |= Wire.read(); //Z lsb
                hmcReading[2] = Wire.read()<<8; //Y msb
                hmcReading[2] |= Wire.read(); //Y lsb
                }
#if ( profiling )
        Serial.print("HMC Mag : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
#endif
        }
void readGyroData(int16_t &gyroX,int16_t &gyroY,int16_t &gyroZ)
        {
        unsigned long profile = micros();
        if (gyro.isRawDataReady())
                {
                gyro.readGyroRaw(&gyroX,&gyroY,&gyroZ);
                }
        else{
                gyroX = NAN;
                gyroY = NAN;
                gyroZ = NAN;
                }
#if ( profiling )
        Serial.print("gyro : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
#endif
        }
```

```
void readAccelData(int16_t &accelX,int16_t &accelY,int16_t &accelZ)
        {
        unsigned long profile = micros();
        int16_t accelT = 0;
        accel.readXYZTData(accelX,accelY,accelZ,accelT);
#if (profiling)
        Serial.print("accel : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
#endif
        }
void readAllPress (USARTClass &pressureSerial,char add0[], char add1[], char add2[], char
    add3[], int16_t *pressure)
        {
        pressureSerial.print("WC\r");
#if (profiling)
        unsigned long profile = micros();
        Serial.print("writing to pressure : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
#endif
        pressure[0] = readUniquePress(pressureSerial,add0);
#if (profiling)
        Serial.print("press0 : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
```

```
#endif
        pressure[1] = readUniquePress(pressureSerial,add1);
#if (profiling)
        Serial.print("press1 : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
#endif
        pressure[2] = readUniquePress(pressureSerial,add2);
#if (profiling)
        Serial.print("press2 : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
#endif
        pressure[3] = readUniquePress(pressureSerial,add3);
#if (profiling)
        Serial.print("press3 : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
#endif
        }

int16_t readUniquePress(USARTClass &pressureSerial,char address[])
        {
        char bytesIn[80]={0x00};
        char readComm[80]={0x00};
```

```
        int nchars;
        strcat(readComm,"U");
        strcat(readComm,address);
        strcat(readComm,"RC\r");
        pressureSerial.print(readComm); //issue "read captured pressure" command to device 1
        nchars = pressureSerial.readBytesUntil('=',bytesIn,80);
        nchars = pressureSerial.readBytesUntil(0x20,bytesIn,20);
        int16_t pressure = int16_t (strtol(bytesIn,NULL,16));
        return pressure;
        }

void readGPS(USARTClass &gpsSerial,char *msgID,uint32_t &utcTime,char **gpsStatus,
    int32_t &gpsLat,char **nsInd,int32_t &gpsLong,char **ewInd,int32_t &gpsSpd,int32_t
    &gpsCrs,uint32_t &date,char **mode,uint32_t &CS)
        {
        unsigned long profile = micros();

        char bytesIn[200] = {0};
        int csCalc;
        int csSentHex;


        if (Serial2.available()>0)
                {
                //Serial.println("Was available");
                unsigned long now = millis();
                unsigned long timeOut = 500;
```

```
int i=0;
while(true) //read bytes until we get to a start byte or timeout
        {
        //Serial.println("in loop #1");
        if (gpsSerial.available()>0)
                {
                //bytesIn[i] = gpsSerial.read();
                if (gpsSerial.read() == '$')
                        {
                        //Serial.println("found '$'");
                        break;
                        }
                if (millis()-now>timeOut)
                        {
                        //Serial.println("timed out #1");
                        break;
                        }
                }
        }
now = millis();
while(true) //read data until a stop byte or until timeout
        {
        //Serial.println("in loop #2");
        if (gpsSerial.available()>0)
                {
                bytesIn[i] = gpsSerial.read();
                if (bytesIn[i++] == '\n')
                        {
```

```
                        //Serial.println("found '\n'");
                        break;
                        }
                    if (millis()-now>timeOut)
                        {
                        //Serial.println("timed out #2");
                        break;
                        }
                    }
            }
    //if we've read a start
    //byte through an end byte, or timed out twice, just
    //flush everything on there and try again next time
    now = millis();
    while(gpsSerial.available()>0)
            {
            //Serial.println("in loop #3");
            gpsSerial.read();
            }

    csCalc = getCheckSum(bytesIn);
    int j=i-4;
    char csSent[3] = {0};
    //Serial.println("Next char is bytesIn[j]");
    for (int k=0;k<2;k++)
            {
            //Serial.println(bytesIn[j]);
            csSent[k] = bytesIn[j++];
```

```
            }
        //Serial.println("next char is csSent");
        //Serial.println(csSent);
        csSentHex = strtol(csSent,NULL,HEX);
        }
if (csSentHex == csCalc)
        {
        //process gps string
        char *s1=bytesIn; //copy to a new variable so we don't mess the data up;
        char *pt; //create container to store separated variable into
        pt = strsep(&s1,",*"); //pull of first delimted strings, based on either a ,
            or a *

        for (int j=0;j<5;j++) //we know it's the msg field, so parse it into
            characters so we can write binary
                {
                msgID[j] = (char )pt[j];
                }

        int i=0; //variable for which section of delimited code we're in
        while (pt = strsep( &s1,",*")) //loop over string, delimiting it as we go
                {
                //Serial.println(pt);
                switch (i++)
                        {
                        case 0:
                                {
                                utcTime = 100*atof(pt);
```

```
            break;
            }
        case 1:
            {
            *gpsStatus = pt;
            break;
            }
        case 2:
            {
            gpsLat = 100000*atof(pt);
            break;
            }
        case 3:
            {
            *nsInd = pt;
            break;
            }
        case 4:
            {
            gpsLong = 100000*atof(pt);
            break;
            }
        case 5:
            {
            * ewInd = pt;
            break;
            }
        case 6:
```

```
            {
            gpsSpd = 1000*atof(pt);
            break;
            }
        case 7:
            {
            gpsCrs = 1000*atof(pt);
            break;
            }
        case 8:
            {
            date = atoi(pt);
            break;
            }
        case 9:
            {
            //magnetic variation, not available
            break;
            }
        case 10:
            {
            //magnetic variation indicator, not available
            break;
            }
        case 11:
            {
            *mode = pt;
            break;
```

```
                                        }
                                case 12:
                                        {
                                        CS = atoi(pt);
                                        break;
                                        }
                                }
                        }
                }
        // if the check sum wasn't correct, empty all data
        else
                {
                //Serial.println("bad CS, flushing gps buffer");
                //Serial.print("Calc-ed CS : ");
                //Serial.println(csCalc);
                //Serial.print("Calc-ed received CS : ");
                //Serial.println(csSentHex);
                //Serial.write(bytesIn);
                //Serial.println();
                while (gpsSerial.available()>0)
                        gpsSerial.read();
                }
#if (profiling)
        Serial.print("writing to pressure : ");
        Serial.print(micros()-profile);
        Serial.println(" usec");
        profile = micros();
#endif
```

```
        }

void serialDeviceInit (UARTClass& mainSerial, USARTClass& deviceSerial, int
    deviceBaud,char ID[])
        {
        deviceSerial.begin(deviceBaud); //begin magnetometer serial communication
        if (ID == "mag")
                {
                deviceSerial.print("*99ID\r"); //send ID command to check if serial port is
                    open
                }
        else if (ID == "gps")
                {
                }
        else if (ID == "pre")
                {
                deviceSerial.print("RS\r"); //send "Read Serial Number" command and wait
                    for a response
                }

        else
                {
                mainSerial.println("Incorrect device ID for serial port initialization .");
                }

        int start = millis();
        int time = start;
        boolean deviceSerialOpen = true; //assume it will open
```

```
while (!deviceSerial.available())
        {
        time = millis(); //wait for device to respond

        if (time > start + 10000)
                {
                deviceSerialOpen = false; // if it doesn't open, set to false
                break;
                }
        }
if (deviceSerialOpen)
        {
        delay(20); //wait for all serial communication to come across before flushing
        while (deviceSerial.available()>0)
                {
                deviceSerial.read();
                }
        mainSerial.print(ID);
        mainSerial.println(" serial port initiliazed .");
        }
else{
        mainSerial.print("ERROR : ");
        mainSerial.print(ID);
        mainSerial.println(" serial port not responding.");
        }
}
```

```
void initSDCard(UARTClass &serial,uint8_t sdCardChipSelect)
        {
        unsigned long time = millis();
        while (millis() < time+10000)
                {
                boolean didWork = SD.begin(sdCardChipSelect);
                if (didWork)
                        {
                        sdCardClosed = false;
                        break;
                        }
                }
        if (sdCardClosed)
                {
                serial.println("ERROR : SD Card not responding.");
                }
        else
                {
                Serial.println("SD card initialized .");
                }
        }


void parseToBinInt16(byte buff[], int16_t var, uint16_t &loc)
        {
        int i=0;
        for (i;i<2;i++)
                {
                buff[i+loc] = byte (var >> (8*i));
```

```
                }
        loc=loc+i;
        }

void parseToBinInt32(byte buff[], int32_t  var, uint16_t  &loc)
        {
        int  i=0;
        for  (i;i<4;i++)
                {
                buff[i+loc] = byte (var >> (8*i));
                }
        loc=loc+i;
        }

void parseToBinUInt32(byte buff[],uint32_t  var, uint16_t  &loc)
        {
        int  i=0;
        for  (i;i<4;i++)
                {
                buff[i+loc] = byte (var >> (8*i));
                }
        loc=loc+i;
        }



void parseInput()
        {
        char comm[80] = {0};
```

```
if (Serial.read() == '#')
        {
        int i = 0;
        while(Serial.available()>0)
                {
                comm[i++] = Serial.read();

                if (comm[i-1] == '&')
                        {
                        break;
                        }
                }
        if (!strcmp(comm,"dataOn&"))
                {
                if (!sdCardClosed)
                        {
                        logData = true;
                        Serial.println("Data logging : on.");
                        }
                else
                        {
                        Serial.println("Please initialize SD card first.");
                        }
                }
        else if (!strcmp(comm, "dataOff&"))
                {
                logData = false;
                Serial.println("Data logging : off.");
```

```
            }
        else if (!strcmp(comm,"serialOn&"))
            {
            printSerialOut = true;
            }
        else if (!strcmp(comm,"serialOff&"))
            {
            printSerialOut = false;
            }
        else if (!strcmp(comm,"initSD&"))
            {
            // initialize SD card
            Serial.println(" Initializing SD Card...");
            initSDCard(Serial,sdChipSelect);
            }
        else if (!strcmp(comm,"?&"))
            {
            Serial.println("Help menu");
            Serial.println("List of commands : ");
            Serial.println("dataOn :\tturn on data logging to SD card.");
            Serial.println("dataOff :\tturn off data logging to SD card.");
            Serial.println("initSD :\ tinitialize SD card.");
            Serial.println("serialOn :\tturn on serial output.");
            Serial.println(" serialOff :\tturn off serial output.");
            Serial.println("? :\t help menu.");
            }
        }
    else
```

```
                {
                Serial.print("Improprer serial command. Flushing data... ");
                while (Serial.available()>0)
                        {
                        Serial.read();
                        }
                Serial.println("Serial data flushed.");
                }

        }


void intHandler0()
        {
        if(digitalRead(pwmPin0))
                {
                trig0 = micros();
                }
        else
                {
                pwm0 = micros()-trig0;
                }
        }

void intHandler1()
        {
        if(digitalRead(pwmPin1))
                {
```

```
                trig1  = micros();
                }
        else
                {
                pwm1 = micros()−trig1;
                }
        }

void intHandler2()
        {
        if (digitalRead(pwmPin2))
                {
                trig2  = micros();
                }
        else
                {
                pwm2 = micros()−trig2;
                }
        }

void intHandler3()
        {
        if (digitalRead(pwmPin3))
                {
                trig3  = micros();
                }
        else
                {
```

```
                pwm3 = micros()-trig3;
                }
        }

void intHandler4()
        {

        if(digitalRead(pwmPin4))
                {
                trig4  = micros();
                }
        else
                {
                pwm4 = micros()-trig4;
                }
        }

void intHandler5()
        {
        if(digitalRead(pwmPin5))
                {
                trig5  = micros();
                }
        else
                {
                pwm5 = micros()-trig5;
                }
        }
```

```
void intHandler6()
        {
        if(digitalRead(pwmPin6))
                {
                trig6 = micros();
                }
        else
                {
                pwm6 = micros()-trig6;
                }
        }

void intHandler7()
        {
        if(digitalRead(pwmPin7))
                {
                trig7 = micros();
                }
        else
                {
                pwm7 = micros()-trig7;
                }
        }

// Calculates the checksum for a given string returns as integer
int getCheckSum(char *string) {
        int i;
```

```
int XOR;
int c;
// Calculate checksum ignoring any $'s in the string
for (XOR = 0, i = 0; i < strlen(string); i++) {
        c = (unsigned char)string[i];
        if (c == '*') break;
        if (c != '$') XOR ^= c;
        }
return XOR;
}
```

G.2 Future Work

The accuracy of the system developed still needs to be extensively validated. The immediate future work following this paper will be quantifying the accuracy of the system, as well as verifying reliability and safe integration into multiple UASs. Most of the accuracy estimation will be accomplished by measuring changes to a vehicle's drag polar. Parasite drag will be proven by adding a known quantity and measuring the change in the parasite drag coefficient of the test vehicle. To quantify the accuracy of drag-due-to-lift, wings with different aspect ratios will be used on the vehicle, which should combine into a single equivalent curve[15]. Following the accuracy estimation, a sensor fusion algorithm will be developed to combine inertial sensors with the air data system and other available sensors in a manner similar to other current research,[35],[36] in order to give full situational awareness to the UAS. This situational awareness could allow stability and control derivative estimation, which the aircraft designer could use to develop and simulate new control laws using actual stability derivatives.

Bibliography

[1] Mike1024. Ecec enu longitude latitude relationships. Wikipedia, February 2010.

[2] MLWatts. Hawker tempest mark ii three view. Wikipedia, September 2012.

[3] Olivier Clynen. 737 ng winglet effect simplified. Wikipedia, June 2012.

[4] Daniel P Raymer et al. *Aircraft design: a conceptual approach*, volume 3. American Institute of Aeronautics and Astronautics, 1999.

[5] Leland Malcolm Nicolai and Grant Carichner. *Fundamentals of aircraft and airship design*, volume 1. Amer Inst of Aeronautics &, 2010.

[6] Jan Roskam. *Airplane design*. DARcorporation, 1985.

[7] Sighard F Hoerner. *Fluid-dynamic drag: practical information on aerodynamic drag and hydrodynamic resistance*. Hoerner Fluid Dynamics, 1965.

[8] Sighard F Hoerner and Henry V Borst. Fluid-dynamic lift: Practical information on aerodynamic and hydrodynamic lift. *NASA STI/Recon Technical Report A*, 76:32167, 1975.

[9] Vladislav Klein and Eugene A Morelli. *Aircraft system identification: theory and practice*. American Institute of Aeronautics and Astronautics Reston, VA, USA, 2006.

[10] Jan Roskam. Airplane flight dynamics and automatic flight controls, design. *Analysis and Research Corporation, Lawrence, KS*, 2001.

[11] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[12] Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995.

[13] Leeland Nicolai. Estimating r/c model aerodynamics and performance. White paper, Society of Automotive Engineers, 2009.

[14] Mark Drela. Xfoil: An analysis and design system for low reynolds number airfoils. In *Low Reynolds number aerodynamics*, pages 1–12. Springer, 1989.

[15] Ludwig Prandtl. *Applications of modern hydrodynamics to aeronautics.* National Advisory Committee for Aeronautics, 1923.

[16] Brian L Stevens and Frank L Lewis. Aircraft control and simulation. 2003.

[17] Stephen R Osborne. *Transitions between hover and level flight for a tailsitter uav.* PhD thesis, Citeseer, 2007.

[18] Atmel. Sam3x-sam3a series summary. Data Sheet 11057BSATARM13-Jul-12, Atmel, 2012.

[19] Atmel. 8-bit atmel microcontroller with 64k/128k/256k bytes in-systemprogrammable flash. Data Sheet 2549PAVR10/2012, Atmel, 2012.

[20] Analog Devices. Micropower, 3-axis,2 g/4 g/8 g digital output mems accelerometer. Technical report, 2012.

[21] U-Line. Easy-count counting scale data sheet. Technical report, U-Line.

[22] Honeywell Magnetic Sensors. Smart digital magnetometer hmr2300. Technical report, Honeywell, 2006.

[23] Honeywell. 3-axis digital compass ic hmc5883l. Technical report, Honeywell.

[24] Talat Ozyagcilar. *Calibrating an eCompass in the Presence of Hard and Soft-Iron Interference.* Freescale Semiconductors, rev 3.0 edition, 04 2013.

[25] Yury Petrov. Ellipsoid fit. Matlab File Exchange, 08 2013.

[26] InvenSense. Itg-3200 product specification revision 1.4. Technical report, 2010.

[27] All Sensors. Ds-0012 rev a. Technical report, All Sensors.

[28] All Sensors. Ds-0010. Technical report, All Sensors.

[29] Maxim Integrated. Ds18b20 programmable resolution 1-wire digital thermometer. Technical report, Maxim Integrated, 2008.

[30] Crystal Engineering, 708 Fiero Lane, Suite 9, San Luis Obispo, California 93401. *nVision Operation Manual for Reference Recorder*, 2013.

[31] Paroscientific. Model 745 high accuracy portable pressure standard data sheet. Technical report, Paroscientific.

[32] Doug Weibel. Proof of concept test - extremely accurate 3d velocity measurement with a ublox 6t module., December 2012.

[33] Finwing Hobby. Finwing universal penguin fpv airplane. Website, 2012.

[34] Dahlke Auman and CW Dahlke. Drag characteristics of ribbons. *AIAA Paper*, 2011, 2001.

[35] Matthew B. Rhudy; Trenton Larrabee; Haiyang Chao; Yu Gu; Marcello Napolitano. Uav attitude, heading, and wind estimation using gps/ins and an air data system. 2013.

[36] Seung-Min Oh and Eric N Johnson. Development of uav navigation system based on unscented kalman filter. In *AIAA Guidance, Navigation and Control Conference*, 2006.