MODELING, IDENTIFICATION AND CONTROL OF A QUAD-ROTOR DRONE USING LOW-RESOLUTION SENSING

BY

YUE SUN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Geir E. Dullerud

Abstract

This thesis focuses on the modeling and identification, control and filter design, simulation and animation, and experiments of an electrical-motor drive model-scale quadrotor --- the AR.Drone. Equations of Motion of drone's model were derived from Kinematics and Dynamics of common quadrotors. The identification was conducted thoroughly including its low-resolution on-board sensors, such as rate gyro and altimeter. Control targets are composed of two stages --- local references following and global position tracking. PID algorithm is used by both controllers with various filters designs, such as low/high pass filter, Complementary Filter and Kalman Filter. Simulation is also divided to two stages with two different simulators ---- MATLAB and C++. The first stage MATLAB simulation is intended to only test the controllers with no disturbances or noises. The second stage high fidelity C++ simulation contains everything including animation. Experiments results are presented and correlated to simulation to evaluate the identification and modeling.

This thesis also includes modeling and identification of a low-resolution camera sensor --- Kinect. The model is included in global position tracking simulation. Some experiments videos and animation videos are available at http://www.youtube.com/user/sunyue89/videos.

The author hopes this thesis is helpful to researchers and amateurs who would like to develop the AR.Drone or any other small scale quadrotors using low-resolution sensing for autonomous control.

Acknowledgement

The author would like to express sincere appreciation to Professor Geir E. Dullerud, for the opportunity to access to all the hardware, and his illuminating instructions along the research. Special thanks to Jan Vervoorst and his advisor Professor Naira Hovakimyan, who generously agreed to share his outstanding MATLAB Simulink model and provide all his studies in quadrotor, which had significantly speeded the research. Thanks to Dan Block for his AR.Drone's components and tools which repaired our broken drone very timely. The author also owes many thanks to the lab partners Steve Granda and Richard Otap, who had provided brilliant software support and hardware preparation. Thanks to Qing Xu for his great skills in setting up identification experiments of the drone, Sangeetha Chandrasekaran for sharing her work in identification of the drone's moment of inertia, Seungho Lee for his assistance in modeling, Peter Maginnis for his help in Kalman Filter, and Kan Chen's foundation work on the animation. The author would like to sincerely appreciate the patience and intelligence of Bicheng Zhang, in taking data, implementing the filters and tuning gains. Also many thanks to Weijia Luo's great work in development of CAD drawings and animation of the drone. Finally, I would like to thank my family and girlfriend who supported my work and helped me through the hard time. This thesis would never have been completed without their encouragement and devotion.

Table of Contents

Chapter 1 Introduction

Robots have been extensively developed and utilized these days. In various industries, robots are widely used to replace humans for dangerous, dirty and boring work. Among these robots, unmanned aerial vehicles (UAVs) are one of the most important families, because of the capability to conduct many military, transportation and scientific research tasks that are difficult or costly for manned aircrafts to accomplish [1].

Structures of UAVs are never unique. Very common ones are single rotor helicopters, fixed-wing aircrafts and quadrotors. Even though fixed-wing aircrafts are the most common large-scale UAVs, quadrotors have their own advantages such as vertical taking-off and landing, stationary and low-speed capability, as well as simple mechanics, good maneuverability and robustness [2]. These advantages have made it the best choice for this research, which is part of networked communication and control study of several small-scale autonomous vehicles. Section 1 gives the description of the configuration and flight control of a quadrotor.

Thanks to the rapid growth of semiconductor and information technology, processors and sensors are getting less expensive with more functionalities and accuracy. This has decently reduced the cost of embedded system, and has given birth to various model-scale robots. One intersection product between these model-scale robots and UAVs is the AR-Drone. It was chosen as the experiment tester for its low cost, powerful processor, multiple sensors, common operating system, Wi-Fi capability and robustness. The drawback of the AR-drone is the system protection and the low-resolution of on board sensors. Section 2 contains brief introduction of the product.

To capture and study the motion of AR-Drone, Kinect was used as the camera sensor. It is a recent launched motion sensing device by Microsoft for Xboxes and PCs. It is able to track movement of objects and individuals in three dimensions in a wide angle of view. Section 3 introduces the basic technology and application of the Kinect

Section 4 provides the overview of this paper after introduction.

1.1 Quadrotor

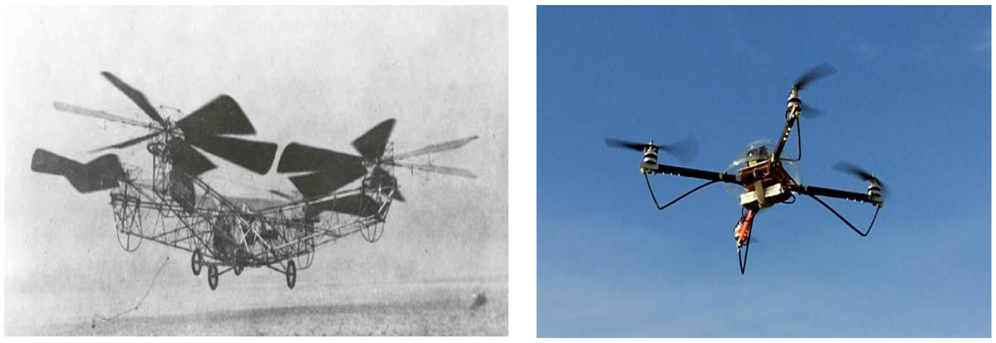


Figure 1.1 Large-scale and Small Scale Quadrotors

Quadrotor is a multi-copter with four propellers and a fixed cross structure. The propellers usually have identical pitch blades and are symmetric about the central of the cross. A sample of large scaled engine quadrotor and a model-scaled quadrotor is shown in Figure 1.1.

Quadrotor has six degrees of freedom (DOF), i.e., three translational (X, Y, Z) and three rotational (Roll, Pitch, Yaw) components, as shown in Figure 1.2.

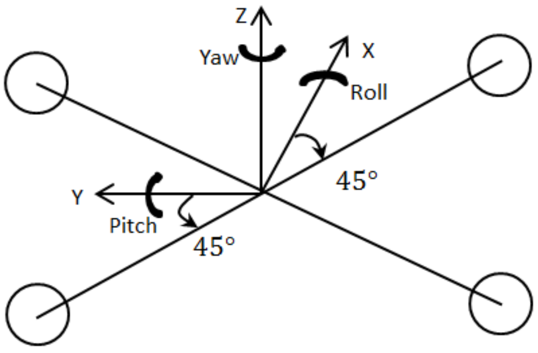


Figure 1.2 Degrees of Freedom of Quadrotor

However, with just four propellers, only a maximum of four desired set-points of DOF can be achieved at one time. Thus, four basic movements are necessary for the quadrotor to reach a desired position and angle. They are throttle, roll, pitch and yaw. Throttle lifts the quadrotor up to a desired Z position, while Roll, Pitch and Yaw control the rotational status of the quadrotor. These movements will be discussed in details in *2.4 Basic Movements*. They are results of a series of mechanisms including engines (electrical motors for small-scale), rotors, gears and propellers, which are all the basic components of a typical quadrator.

1.2 AR.Drone

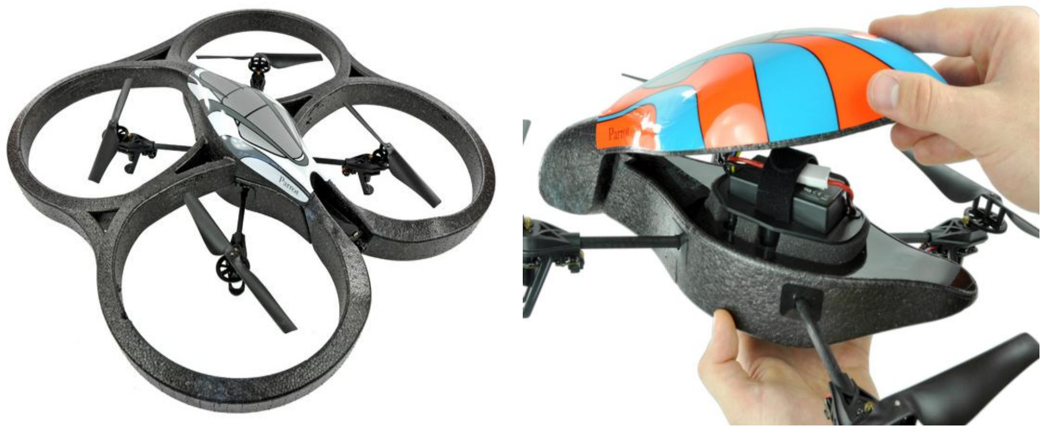


Figure 1.3 AR.Drone Appearance and the Interior Layout

AR.Drone is a quadrotor helicopter built by French company Parrot as seen in Figure 1.3. It is mainly composed of a cross beam, four electrical motors and propellers, two electrical boards, two cameras, a base house that protects and connects all the above components, and a top cover that seals the electrical boards. The cross beam is made of plastic and the body (base and cover) is mostly made of foam. Each beam is about 40 cm long, and the body is about 30 cm long. Important components and their technical specifications are listed below [3].

- 15 W brushless electric motor
- High-efficiency customized propeller
- ARM9 468 MHz embedded microcontroller
- 128 MG of RAM
- Linux Operating System
- Wi-Fi and USB communication
- MEMS 3-axis accelerometer
- 2-axis gyro and a 1-axis yaw precision gyro
- Ultrasonic altimeter with range of 6 m
- Two fitted wide-angle cameras (93 degrees)

1.3 Kinect

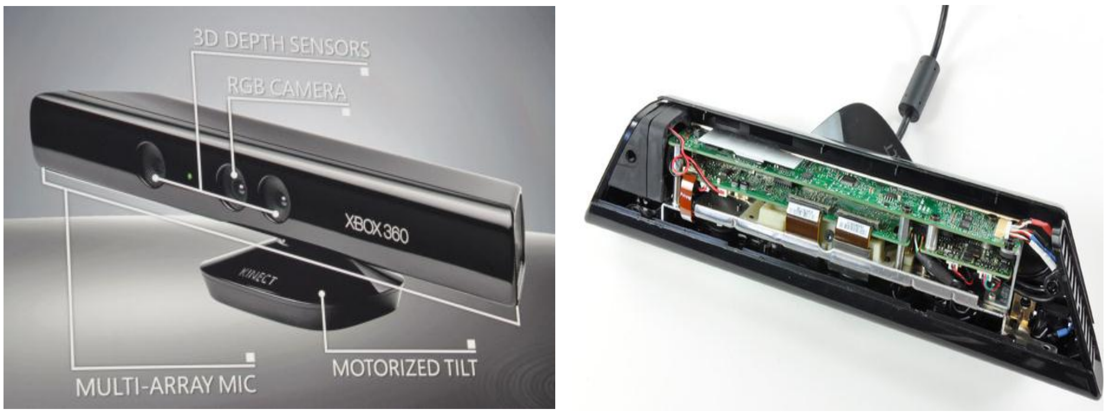


Figure 1.4 Kinect Appearance and Interior Layout

Kinect is a motion sensing device, with the shape of a horizontal bar connected to a small base by a pivot, which provides a tiling angle of around 27d up and down. It is connected to Xbox or PC through USB, and user-interface software is available for several applications, such as motion caption, video chat and facial recognition.

Infrared laser projector, camera and special microchip are used to track motions, especially the depth motion, in a very wide range, approximately 57d horizontally and 43d vertically. Range of 1.2-3.5 [m] gives the most accuracy.

Kinect outputs video with frame rate of about 30 Hz. The RGB video stream has 8-bit 640x480 pixels with a Bayer color filter, while the depth sensing video stream is 11-bit. The horizontal minimum viewing distance is about 87cm, and the vertical one is about 63cm, resulting in a resolution of about 1.3mm per pixel.

1.4 Overview

With basic introduction of quadrotor, this paper starts studying the kinemics and dynamics of quadrotor in Chapter 2, such that differential equations of motion (EOM) are set up.

Chapter 3 focuses on techniques and results of identification of the specific quadrotor – AR.Drone. Noise level of its onboard sensors and of the Kinect was also measured or estimated. With identified data, it discusses ways to simplify EOMs and to build up the dynamical model for the AR.Drone. It also suggests models for various sensors.

Chapter 4 explains the control algorithm and filter design. Both controller and filter design is composed of two stages. The local controller is designed to follow quadrotor height and angle reference commands with swift response by only on-board sensors. The global controller, on the other hand, is designed to achieve automatic global position control with the assistance of external camera sensor --- the Kinect. Local filter section represents three different ways to estimate angle from gyroscope, including a high pass filter, a Complementary Filter and an open-source Kalman Filter design. The global filter section discusses a low pass filter, a Complementary Filter and a Kalman filter design for more accurate global position and velocity estimation based on Kinect data.

Chapter 5 is where transition from theories to practice occurs. It first introduces simulation designs that are used to test and tune the controller and filter, along with the quadrotor and sensor models, in two different simulators – MATLAB and C++. Chapter 5 then gives detailed procedure of the experimental set up and data acquired from experiments with the tuned controller and different filters. These data was used to compare the filter design and select the best. Then the controllers and filters are re-evaluated in the model for validation by correlating simulation and experiments results.

Chapter 6 then introduces the open-source C++ library, Irrilicht Engine, to visualize the simulation. This step is to alive the data and to make modeling more fun. It also gives direct visualized 3D comparison between simulation and experiments.

Chapter 7 concludes this paper and suggests future work.

Chapter 2 Kinemics and Dynamics

This chapter derives EOMs of a quadrator. It starts with a generic 6 DOF rigid body. Section 2.1 and section 2.2 introduce the kinemics and dynamics of the rigid body respectively. Section 2.3 describes the force and torque input such that EOMs are completely introduced.

Several assumptions have been made to simplify the dynamics. The inertia matrix is assumed to be time-invariant. It is also diagonal by assuming origin of body frame coincides with center of mass, and axes of body frame coincide with principle axes of inertia. The dynamic equations are derived by assuming propellers the only force and torque source. The gyroscopic effect produced by propeller rotation is also neglected by assuming their contribution much less than thrust and drag due to propeller rotation.

2.1 Kinemics

Kinemics studies the motion of an object or particle without active forces and torques. For a 6 DOF rigid body, it is common to define two reference frames [4], i.e., the earth frame and body-fixed frame, to describe its motion. Kinemics of a 6 DOF rigid body then studies the translational and rotational relationship between these frames. We shall set up the reference frames as below, so that it is consistent with the AR.Drone. The rotational and translational vectors are illustrated in Figure 2.1 below.

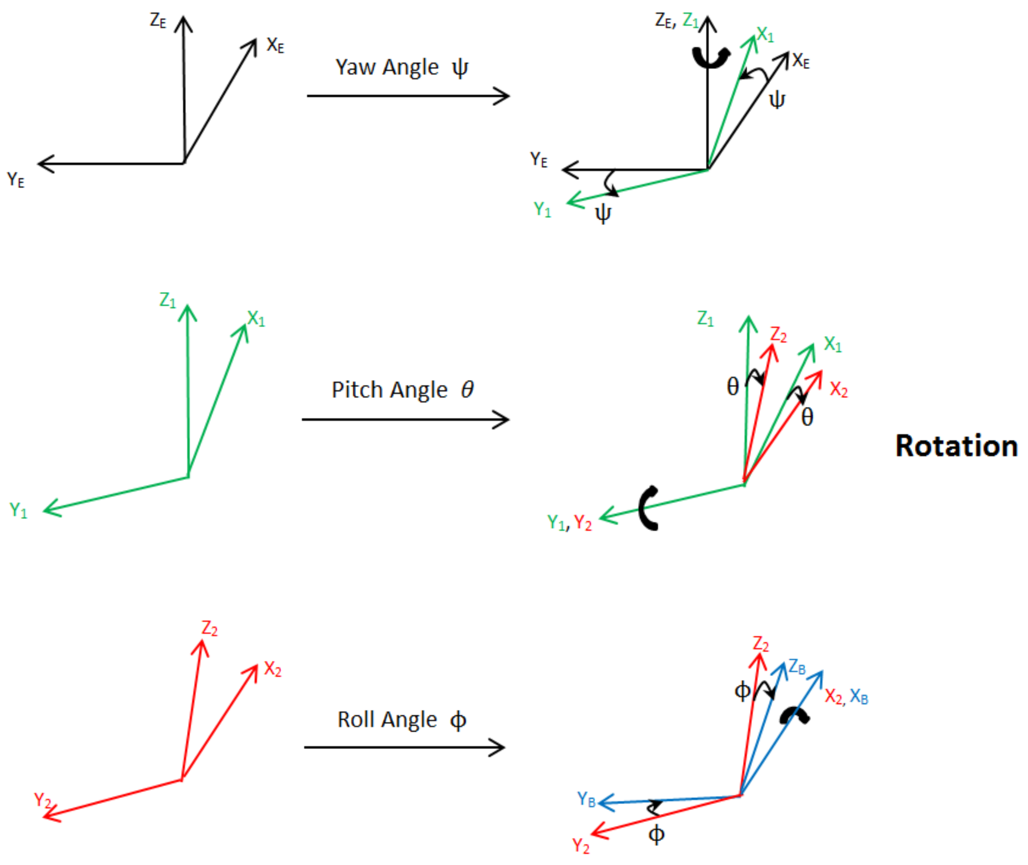


Figure 2.1 (cont. on next page)

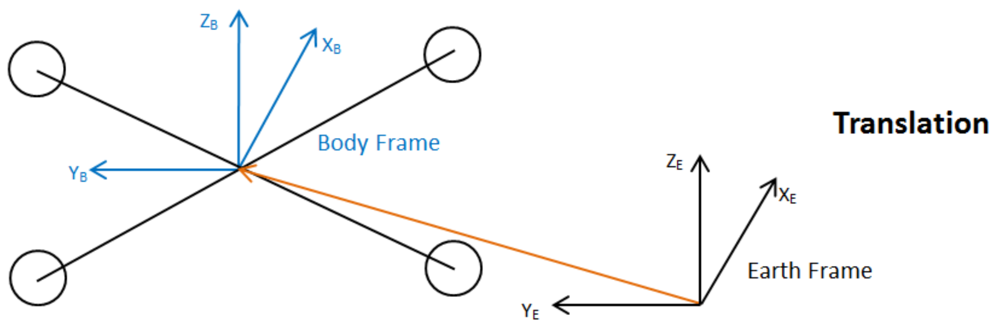


Figure 2.1 Rotational and Translational Vectors

From the figure, the translation and rotation position of the body frame respect to the earth frame, could be expressed by the vectors

$$\mathrm{S}^E = [x\ y\ z]^T \tag{2.1}$$

$$\Theta^E = [\phi\ \theta\ \psi]^T \tag{2.2}$$

respectively. The "E" means that they are defined the in the earth frame. In aerial terminology ϕ, θ and ψ are called roll, pitch and yaw angle, respectively.

In the body frame, however, it is meaningless to define translation and rotation position vectors, because the body frame itself is moving. However, it is necessary to define velocity vectors, since force and torque are acting directly on the frame. Thus, we define linear and angular velocity vectors respect to body frame

$$\mathrm{V}^B = [u\ v\ \omega]^T \tag{2.3}$$

$$\Omega^B = [p\ q\ r]^T \tag{2.4}$$

respectively. The "B" indicates them being defined in the body frame. Components of V^B and Ω^B are defined in the same order as those of S^E and Θ^E, i.e., *x, y, z*, ϕ, θ and ψ.

The rotational matrix, R_θ, relates translation position S^E in earth frame to linear velocity V^B in body frame by

$$\dot{\mathrm{S}}^E = \mathrm{V}^E = R_\theta \mathrm{V}^B \tag{2.5}$$

The expression of the rotational matrix R_θ is given by

$$R_\theta = \begin{bmatrix} cos\psi cos\theta & cos\psi sin\theta sin\phi - sin\psi cos\phi & cos\psi sin\theta cos\phi + sin\psi sin\phi \\ sin\psi cos\theta & sin\psi sin\theta sin\phi + cos\psi cos\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi \\ -sin\theta & cos\theta sin\phi & cos\theta cos\phi \end{bmatrix} \tag{2.6}$$

Please refer to Appendix A for derivation of Eq2.5 and Eq2.6.

The relationship between rotational position Θ^E in earth frame and angular velocity Ω^B in body frame involves the translational matrix, T_θ, by

$$\dot{\Theta}^E = \Omega^E = T_\theta \Omega^B \tag{2.7}$$

The expression of the translational matrix T_θ is given by

$$T_\theta = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix} \tag{2.8}$$

Please refer to Appendix A for derivation of Eq2.7 and Eq2.8.

2.2 Dynamics

For every vector p defined in body frame, its time derivative as seen in the earth frame is given by

$$\frac{d}{dt^E} p = \frac{d}{dt^B} p + \omega_{B/E} \times p \tag{2.9}$$

where $\omega_{B/E}$ is the angular velocity of body frame with respect to the earth frame. This is called the Equation of Coriolis and is derived in [5].

According to Newton's law, the force acting on the quadrotor is related to the derivative of the linear velocity vector V^B, as seen in the earth frame, by

$$f^B = m \frac{d\mathrm{V}^B}{dt^E} \tag{2.10}$$

where m is the mass of quadrotor. Notice that both the force and velocity is defined in body frame, only the derivative is respect to earth frame.

Combing the Coriolis equation and Newton's law, we can obtain the dynamic equation for V^B in body frame, as

$$f^B = m\left(\frac{d\mathrm{V}^B}{dt^\mathrm{B}} + \omega_{B/E} \times \mathrm{V}^B\right) \tag{2.11}$$

The same rule applies to torque and angular velocity vector Ω^B. The Newton's law for it is given by

$$\tau^B = \mathrm{I}\frac{d\Omega^B}{dt^E} \tag{2.12}$$

where I is the moment of Inertia matrix. Combining with equation of Coriolis it becomes

$$\tau^B = \mathrm{I}\left(\frac{d\Omega^B}{dt^B} + \omega_{B/E} \times \Omega^B\right) \tag{2.13}$$

Being aware that $\omega_{B/E}$ is identical to the angular velocity vector Ω^B because $\omega_{B/E}$ is measured in body frame. Decompose Eq2.11 and Eq2.13 with $f^B = (f_x, f_y, f_z)^T$, $\tau^B = (\tau_\phi, \tau_\theta, \tau_\psi)^T$, $V^B = [u\ v\ \omega]^T$, $\Omega^B = [p\ q\ r]^T$,the dynamic equations of a quadrator are

$$\begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} = m \begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{\omega} \end{pmatrix} + m \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} u \\ v \\ \omega \end{pmatrix} \tag{2.14}$$

$$\begin{pmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{pmatrix} = \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} + \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \cdot \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \end{pmatrix} \tag{2.15}$$

2.3 Forces and Moments

As we expect, the force vector f^B and moment vector τ^B primarily depends the propeller speed, the gravity, and the status (roll, pitch and yaw angle) of the quadrator. We shall start from the net force and torque due to spinning speed of each propeller defined in Figure 2.2 and Figure 2.3 below. Figure 2.2 shows the ID of each motor, their rotational directions and distances to the center, from the top view. Figure2.3 then defines the forces and torque generated by each motor. Note that the reference frame is consistent to the body frame defined for AR.Drone in Figure1.2.

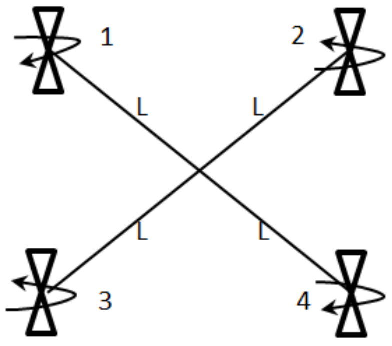


Figure 2.2 Spinning Direction of Motors

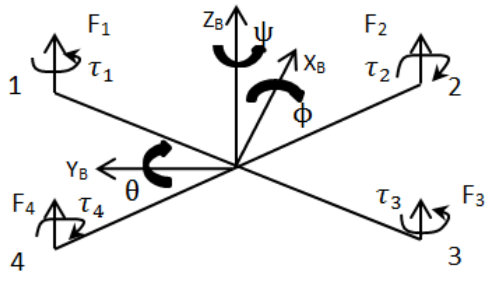


Figure2.3 Generated Forces and Torques

The force and torque generated by each propeller is related to rotational speed by [6]

$$F_i = b\Omega_i^2 \tag{2.16}$$

$$\tau_i = d\Omega_i^2 \tag{2.17}$$

where i denotes motor number from 1 to 4, *b* [Ns^2] is the aerodynamic thrust and *d* [Nms^2] is the aerodynamic drag.

Combination of these forces and torques gives net thrust force, roll, pitch and yaw moment by

$$\begin{cases} F = F_1 + F_2 + F_3 + F_4 \\ \tau_r = \frac{1}{\sqrt{2}} l(F_1 + F_4 - F_2 - F_3) \\ \tau_p = \frac{1}{\sqrt{2}} l(F_3 + F_4 - F_1 - F_2) \\ \tau_y = \tau_1 + \tau_3 - \tau_2 - \tau_4 \end{cases} \tag{2.18}$$

Since the propellers are fixed to the body, all these forces and torques are in body frame. Thus, the torque vector τ^B is simply the combination of τ_r, τ_p and τ_y above. The force vector f^B, however, has to include gravity mapped to body frame by the rotational matrix R_θ.

$$f_B = \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ F \end{pmatrix} - R_\theta \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix} = \begin{pmatrix} gsin\theta \\ -gcos\theta cos\phi \\ -gcos\theta cos\phi + F/\mathrm{m} \end{pmatrix} \tag{2.19}$$

Plug force vector and torque vector into Eq2.14 and Eq2.15, rearrange equations in the state space form, we can summarize the dynamic equations in the state space form as

$$\begin{cases} \begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} rv - q\omega \\ p\omega - ru \\ qu - pv \end{pmatrix} + \begin{pmatrix} gsin\theta \\ -gcos\theta cos\phi \\ -gcos\theta cos\phi + \frac{F}{m} \end{pmatrix} \\ \begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{I}_{yy} - \mathrm{I}_{zz}}{\mathrm{I}_{xx}} qr \\ \frac{\mathrm{I}_{zz} - \mathrm{I}_{xx}}{\mathrm{I}_{yy}} pr \\ \frac{\mathrm{I}_{xx} - \mathrm{I}_{yy}}{\mathrm{I}_{zz}} pq \end{pmatrix} + \begin{pmatrix} \frac{\tau_\phi}{\mathrm{I}_{xx}} \\ \frac{\tau_\theta}{\mathrm{I}_{yy}} \\ \frac{\tau_\psi}{\mathrm{I}_{zz}} \end{pmatrix} \end{cases} \tag{2.20}$$

It is a good idea to also recall and decompose the kinemics equation here

$$\begin{cases} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = R_\theta \begin{pmatrix} u \\ v \\ \omega \end{pmatrix} = \begin{bmatrix} cos\psi cos\theta & cos\psi sin\theta sin\phi - sin\psi cos\phi & cos\psi sin\theta cos\phi + sin\psi sin\phi \\ sin\psi cos\theta & sin\psi sin\theta sin\phi + cos\psi cos\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi \\ -sin\theta & cos\theta sin\phi & cos\theta cos\phi \end{bmatrix} \begin{pmatrix} u \\ v \\ \omega \end{pmatrix} \\ \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = T_\theta \begin{pmatrix} p \\ q \\ r \end{pmatrix} = \begin{bmatrix} 1 & sin(\phi)\, tan(\theta) & cos(\phi)\, tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi)\, sec(\theta) & cos(\phi)\, sec(\theta) \end{bmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \end{cases} \tag{2.21}$$

Equation Set 2.20 and 2.21 give a complete description of motion of a quadrotor.

2.4 Basic Movements

Recall that in Chapter 1 Section 1 we introduced four basic independent movements of a quadrotor. Now we could directly match the propeller speeds to each of the movement, and study what force and torque is generated in each scenario, based on Equation 2.16, 2.17 and 2.18. The scenarios are categorized to:

- ***Throttle***

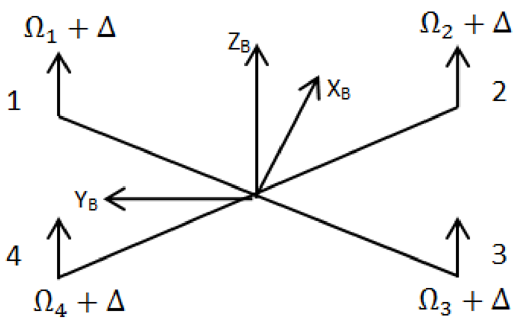


Figure 2.4 Throttle Movement

The net roll and pitch and yaw moment shall be maintained, while the thrust force changes. Thus all propeller speeds should be increased by the same amount in order to generate a positive throttle force.

- ***Roll***

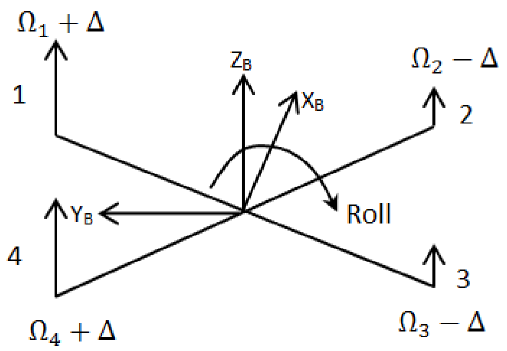


Figure 2.5 Roll Movement

The net force, pitch and yaw moment shall be maintained, while roll moment is not. Thus speed of propeller 1 and 4 should be increased the same amount as that of propeller 2 and 3 being decreased, in order to generate a positive roll moment.

- ***Pitch***

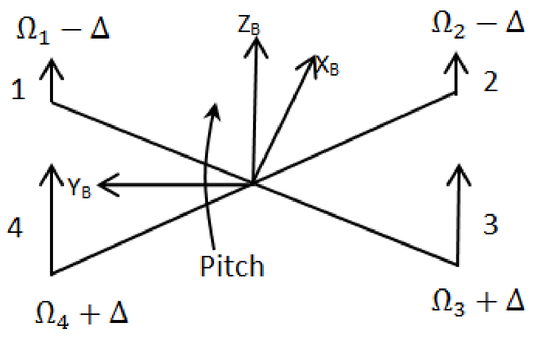


Figure 2.6 Pitch Movement

The net force, roll and yaw moment shall be maintained, while pitch moment is not. Thus speed of propeller 3 and 4 shall be increased the same amount as that of propeller 1 and

4 being decreased, in order to generate a positive pitch moment.

- ***Yaw***

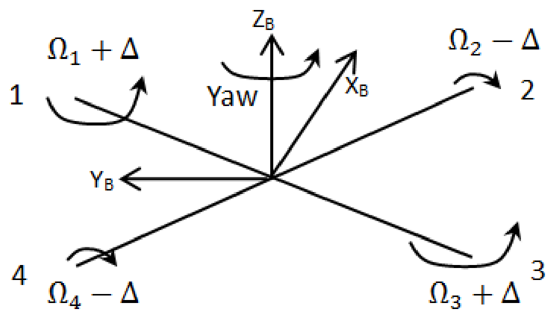


Figure 2.7 Yaw Movement

The net force, roll and pitch moment shall be maintained, while yaw moment is not. Thus speed of propeller 1 and 3 shall be increased the same amount as that of propeller of 2 and 4 being decreased, in order to generate a positive yaw moment.

The study of four basic movements gives us an intuitive scope of how to control AR-Drone, and will be used later again, in discussion of how to translate movement channel's control signals into motor RPMs.

Chapter 3 Modeling and ID

Continuing from the equations derived in Chapter 2, the first section of this chapter gives a detailed dynamic model of the drone. To quantize the model it deals with procedures and results of identification of AR.Drone's key parameters in Section 2. To improve the accuracy of model, various sensor noises shall be included. Section 3 suggests models for the Kinect, the rate gyro and the altimeter. Section 4 then shows the results of identification of these sensors.

3.1 AR.Drone Model

EOMs of the quadrotor have already been developed in equation set 2.20 and 2.21. However, the acceleration vectors $\dot{V^B}$ and $\dot{\Omega^B}$ are all in body frame. To better understand the global position control it is very important to observe the acceleration vectors in earth fame. Taking derivative of the Kinemics equation 2.5, plug in 2.20, neglecting the cross product due to Coriolis, we have

$$\ddot{S}^E = \dot{R}_\theta V^B + R_\theta \dot{V^B} = 0 + R_\theta \begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} (sin\psi sin\phi + cos\psi sin\theta cos\phi)\dfrac{F}{m} \\ (-cos\psi sin\phi + sin\psi sin\theta cos\phi)\dfrac{F}{m} \\ -g + cos\theta cos\psi \dfrac{F}{m} \end{pmatrix} \tag{3.1}$$

Define a new state vector, $X = (u, v, w, p, q, r, x, x_{dot}, y, y_{dot}, z, z_{dot}, \phi, \theta, \psi)^T$, the dynamical equation can be then expressed in the state space form by $\dot{X} = f(X, U)$, where U is the combination of force and torque input vector. The equation set is listed below.

$$\left\{ \begin{array}{c} \dot{u} = rv - q\omega + gsin\theta \\ \dot{v} = p\omega - ru - gcos\theta sin\phi \\ \dot{\omega} = qu - pv - gcos\theta cos\phi + \dfrac{F}{m} \\ \dot{p} = \dfrac{\mathrm{I}_{yy} - \mathrm{I}_{zz}}{\mathrm{I}_{xx}} qr + \dfrac{\tau_\phi}{\mathrm{I}_{xx}} \\ \dot{q} = \dfrac{\mathrm{I}_{zz} - \mathrm{I}_{xx}}{\mathrm{I}_{yy}} pr + \dfrac{\tau_\theta}{\mathrm{I}_{yy}} \\ \dot{r} = \dfrac{\mathrm{I}_{xx} - \mathrm{I}_{yy}}{\mathrm{I}_{zz}} pq + \dfrac{\tau_\psi}{\mathrm{I}_{zz}} \\ \dot{x} = x_{dot} \\ \ddot{x} = (sin\psi sin\phi + cos\psi sin\theta cos\phi)\dfrac{F}{m} \\ \dot{y} = y_{dot} \\ \ddot{y} = (-cos\psi sin\phi + sin\psi sin\theta cos\phi)\dfrac{F}{m} \\ \dot{z} = z_{dot} \\ \ddot{z} = -g + cos\theta cos\psi \dfrac{F}{m} \\ \dot{\phi} = p + \sin(\phi)\tan(\theta)\, q + \cos(\phi)\tan(\theta)\, r \\ \dot{\theta} = \cos(\phi)\, q - \sin(\phi)\, r \\ \dot{\psi} = \dfrac{sin\phi}{cos\theta} q + \dfrac{cos\phi}{cos\theta} r \end{array} \right. \tag{3.2}$$

The dynamical equations are nonlinear with all the triangular forms and second order terms (rv, for example). Linearization could have been conducted around point $\Theta^E = [\phi\ \theta\ \psi]^T\ = 0$ and neglecting the second order terms, assuming their contribution is small. However linearization becomes less accurate as Θ^E and accelerations (r, v, etc) become large. Since I did not intend to design a linear controller with optimization, I kept the nonlinear forms in the model.

3.2 AR.Drone ID

Mass and moments of inertia come naturally as the key parameters of a 6 DOFs rigid body. Besides those, we also need to identify the map from propeller speed to aerodynamic force and drag for a quadrotor.

3.2.1 Mass ID

Scale is used to measure the mass of the AR.Drone. By taking average of each five measurements of two different drones with hood, the average mass of each is 430.5 [g] and 436.5 [g].

Table 3.1 Mass measurement of AR.Drone

	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
AR.Drone 1 Mass [g]	430.6	430.5	430.5	430.6	430.5
AR.Drone 2 Mass [g]	436.5	436.5	436.6	436.4	436.6

3.2.2 Moment of Inertia ID

Modern CAD software, SolidWorks, is used to estimate the moment of inertia. Since we could measure the mass and geometric parameters of the cross beam, the body and the battery, we can draw a CAD model in SolidWorks with the same shape and mass. Four motors and propellers are then estimated by four cylinders in the specific position with the same mass. Figure 3.1 below shows the CAD drawings.

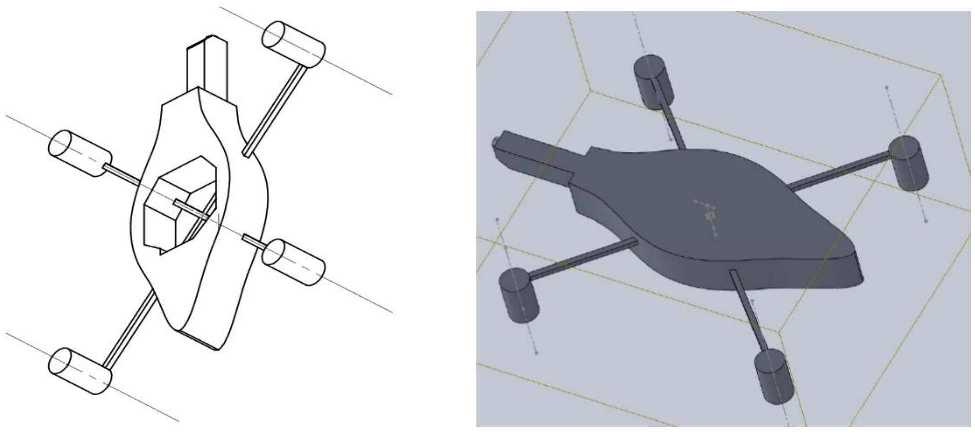


Figure 3.1 Simplified CAD Drawing for Moment of Invertia ID

The moment of inertia matrix is estimated to be

$$\mathrm{I} = \begin{pmatrix} \mathrm{I}_{xx} & 0 & 0 \\ 0 & \mathrm{I}_{yy} & 0 \\ 0 & 0 & \mathrm{I}_{zz} \end{pmatrix} = \begin{pmatrix} 2.04016 & 0 & 0 \\ 0 & 1.56771 & 0 \\ 0 & 0 & 3.51779 \end{pmatrix} \times 10^4 \ \mathrm{g} * \mathrm{cm}^3 \qquad (3.3)$$

3.2.3 Aerodynamic thrust and drag coefficients ID

It is generally hard to measure aerodynamics force and torque. Fortunately, engineering reverse methodology can be used to estimate the coefficients. For example, we could make the AR.Drone hover with different masses to estimate the net force generated by propellers. We could also provide a known torque to the yaw channel, and make it balance to estimate the net torque from propellers. Yes these methods will require good control of the drone first, but we could estimate these coefficients with data available from existing experiments, build up a raw model, design and tune controller that stabilizes the flying, and then apply engineering reverse to obtain an accurate model.

Thanks to Steve Granda and Richard Otap's work on the software side, the drone's software was reconfigured by us to achieve an open architecture environment in which to carry out development. The reconfiguration was as low as sending commands to motors and obtaining data from onboard sensors. The commands for motors, however, are in the range from 0 to 0x1ff, in format of hex. Thus, experiments were conducted to study the relationship between motor command and propeller RPM.

Hall Effect sensor, magnet and oscilloscope were used in the experiment. The mini-magnet, with diameter 6[mm] and height 2[mm], was fixed onto the bottom of the propeller gear. One side of the Hall Effect sensor board is then attached onto the side of motor, while the other side wired into the oscilloscope. When the magnet is rotated to the position right above the sensor, a high voltage signal will be recorded in the scope. Then measurement of the time difference between two neighbor high signals gave the period of one revolution. Then Revolution per Minute (RPM) can be easily calculated. The experiment set-up is shown as below:

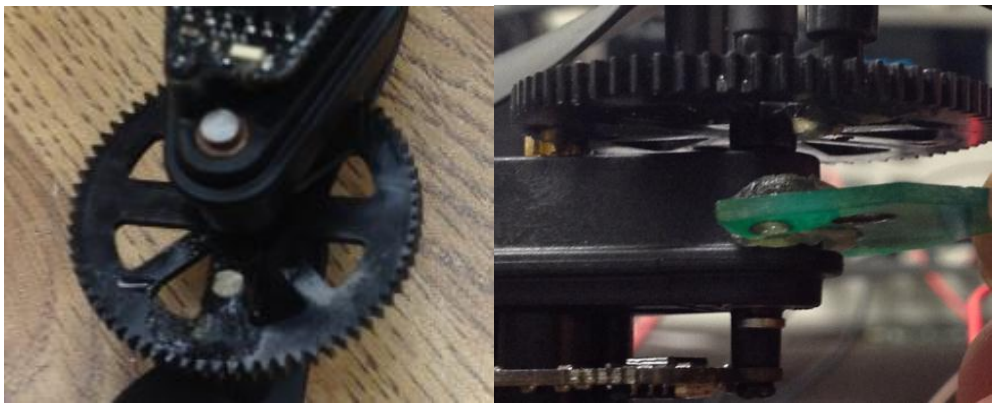


Figure 3.2 (cont. on next page)

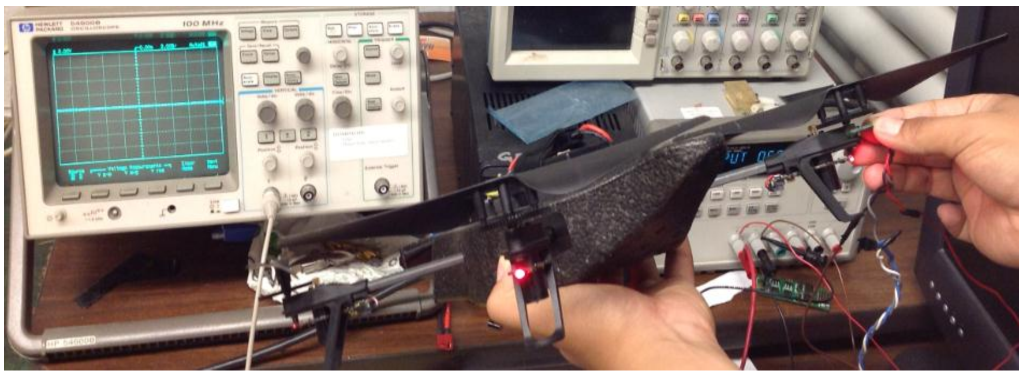


Figure 3.2 Experiment setup of propeller speed measurement

The experiments were conducted four times, with each of the motors and propellers, by sending motor command from 56 to 511 (0x38 to 0x1ff in Hex). The final linear fit data with trend line is shown below.

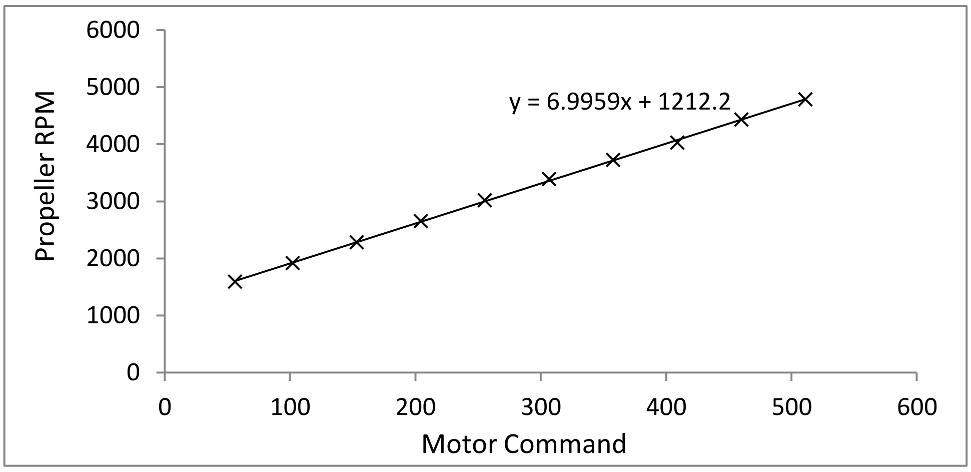


Figure 3.3 Linear Fit of Propeller RPM and Motor Command

Given the gear ratio from motor gear to propeller gear 8:68, we could also estimate the motor's speed. The motor speed was calculated to be in the range of 13527 [RPM] to 40690 [RPM], respective to the input range of 56 to 511. The start speed could be found at the intersection with propeller RPM axis, which corresponds to motor speed of 10340 [RPM]. The start and maximum motor speed are very close to those provided in Parrot's official site (starts at 10350 RPM and goes to 41000 RPM) [7], which proves the identification successful.

The importance of this experiment lies on the fact that RPM of propellers are very difficult to measure when flying, but not the motor command. By recording four motor commands, RPM of the propellers could be estimated by the linear fit trend line. The total mass, then, is supported by the sum of square of each motor's speed, assuming thrust is linear to speed square. The chart below shows the map from propeller RPM square to mass

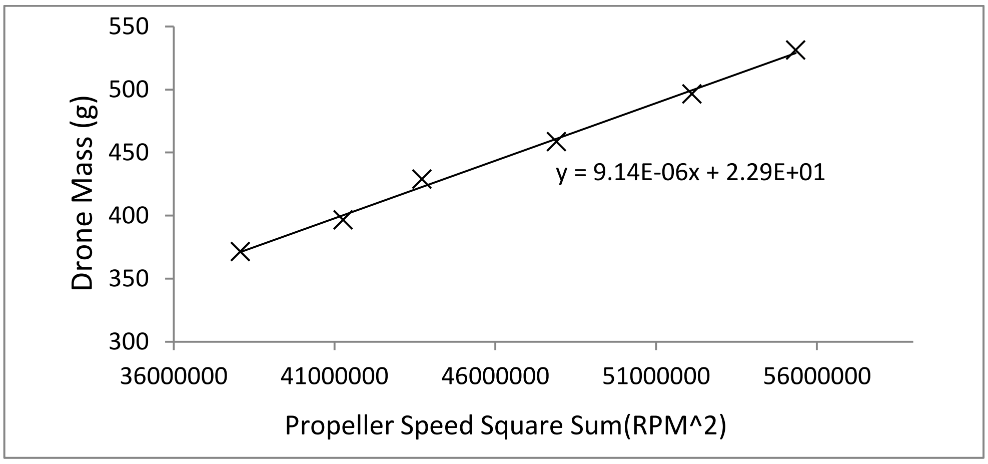


Figure 3.4 Linear Fit of Drone Mass and Propeller Speed Square Sum

The aerodynamics thrust coefficient for the AR.Drone, from propeller speed square to thrust, is estimated by the slope of the map to be $9.14 \times 10^{-6}\ [g/RPM^2]$. With a gear ratio 8:68, it corresponds to $1.27 \times 10^{-7} [g/RPM^2]$ from motor speed square to thrust. The experiment set up is shown in the following figure.

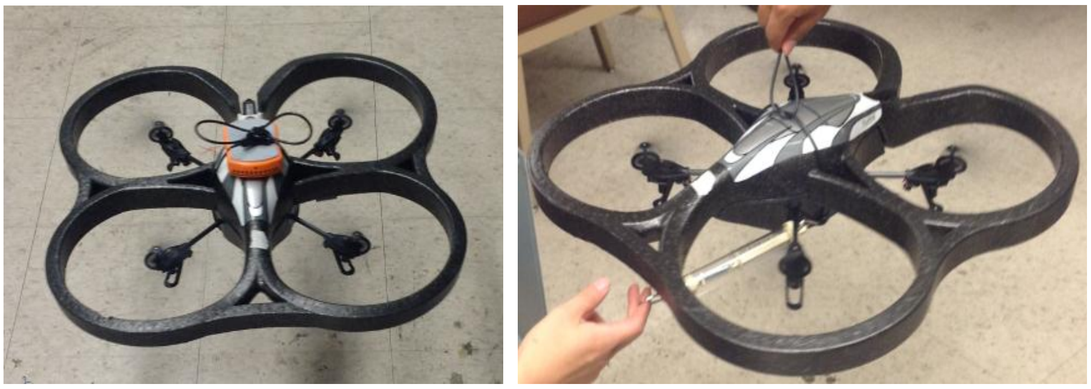


Figure 3.5 Experiment Setup for ID of aerodynamic thrust and drag coefficient

The same method is used to estimate the aerodynamics drag coefficient as shown in the figure above. Applying a known force that produces a negative yaw torque with a fixed distance to the center, the torque generated is then balanced by the torque due to $(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2)$. The char below shows the map from propeller RPM square to torque

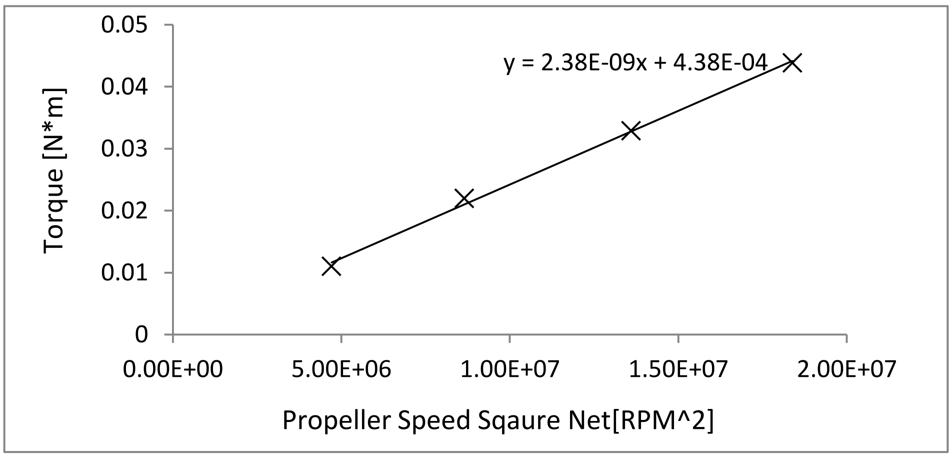


Figure 3.6 Linear Fit of Yaw Torque and Net Propeller Speed Square

The aerodynamics drag coefficient for the AR.Drone from propeller speed square to torque is then estimated to be $2.38 \times 10^{-9}\ [N \cdot m/RPM^2]$. It corresponds to $3.29 \times 10^{-11}\ [N \cdot m/RPM^2]$ from the motors, with a gear ratio of 8:68.

3.3 Sensor Model

The most common model for sensors is the output of nominal data plus white noise. However specific sensors have their own noise source. Kinect model is introduced first, followed by the MEMS gyroscope model and ultrasonic altimeter model. They are used to measure position, Euler angular speed and height, respectively. Accelerometers are not used, and the reason will be explained in *5.2 Local Filter Design*. The other on board sensor such as the camera is not used at this stage.

3.3.1 The Kinect model

Kinect, as a regular camera sensor, has a quantized error and measuring noise. It may also have offset or scale error to real position. To simplify the model we assume the output is only composed of the scaled nominal data with scaled offset and Gaussian white noise. The assumption is valid if the noise dominates the quantized error. The model can be described by the equation

$$k = p + \alpha \cdot o + n \tag{3.4}$$

where p is the true position, o is the offset, n is the noise and α is the scale factor.

3.3.2 The Gyroscope model

The gyroscope is used to be integrated to obtain angle, thus needs to be modeled in more details. Common model of gyro assumes that the gyro output g, is result of the true rotational rate ω, plus a constant bias b_c, a walking bias b_ω, and wide band sensor noise n [8].

$$g = \omega + b_c + b_\omega + n \tag{3.5}$$

The constant bias, b_c, is the average output of the gyro when no rotation has occurred. It can be measured by taking long time average of the gyro output.

The walking bias, b_ω, is mainly due to flicker noise in the electronics and other components. It dominates at low frequencies [9]. The flicker noise can be modeled by a random walk whose standard deviation

$$\sigma_b[°/s] = BS[°/s]\sqrt{\delta t/\tau} \tag{3.6}$$

where $BS[°/s]$ is so called Bias Stability/Bias Instability or Bias Variation, defined by manufacturer with the lowest point on Allan Variation analysis. It evaluates the gyro's walking bias instability within the Allan averaging time τ, with a sampling period δt.

The wide band sensor noise, n, is due to thermo-mechanical noise which fluctuates at higher frequencies than sampling rate. It thus can be modeled by white noise with zero mean and standard deviation $\sigma_n[°/s]$. Integrated over time, it produces angel random walk noise with standard deviation $\sigma_\theta\left[°/\sqrt{s}\right]$ satisfies

$$\sigma_\theta(t)[°/\sqrt{s}] = \sigma_n[°/s]\sqrt{\delta t \cdot t[s]} \tag{3.7}$$

where δt again is the sampling time. Manufacturer usually define Angle Random Walk (ARW) by [10]

$$ARW = \sigma_\theta(1) = \sigma_n\sqrt{\delta t} \tag{3.8}$$

Thus we can estimate the standard deviation of the white noise given ARW and sampling period.

3.3.3 Altimeter model

Altimeter is modeled by a nominal output with offset and noise. The model thus can be

$$a = h + \alpha \cdot o + n \tag{3.9}$$

where a is the altimeter output, h is the true height, α is the scale factor, o is the offset and n is the white noise.

3.4 Sensors ID

Based on the models of each sensor, Identification experiments were conducted. The results of gyro are also compared with some low-cost low-resolution consumer gyro sensors on market.

3.4.1 Kinect ID

Kinect is identified by placing the drone on a thin shelf in front of it, and recording the measured data. We started from the point in the middle front of the Kinect, measured the distances from the point to the origin (somewhere in our lab's corner) defined in our global coordinates, and calibrated the Kinect with the positions measured. Then the drone was measured in the following locations surrounding the center.

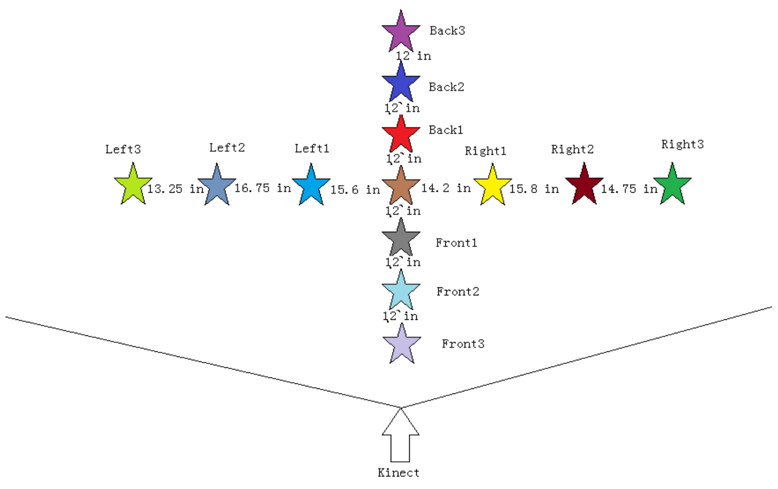
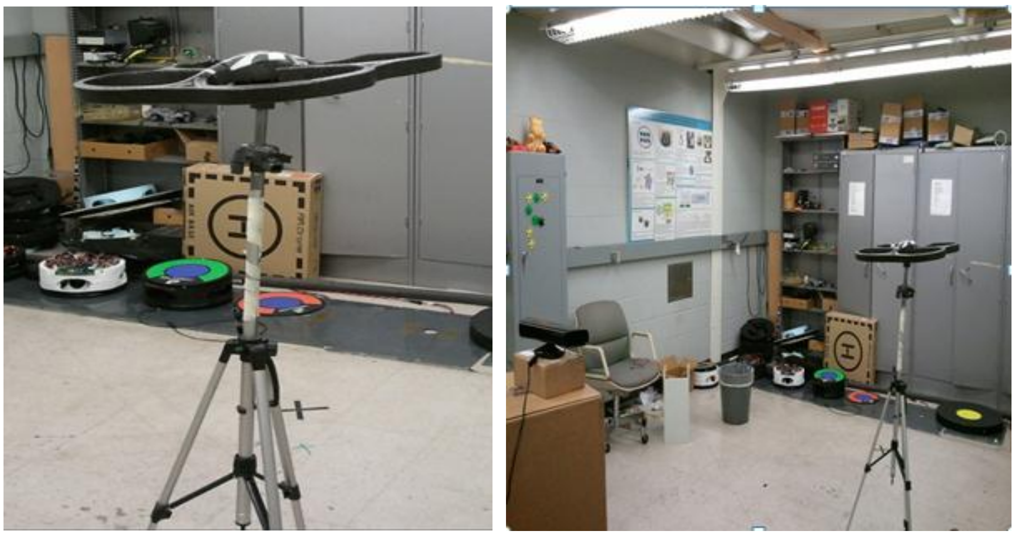


Figure 3.7 Test locations of the Kinect

The experiment set up was shown below, along with the results.



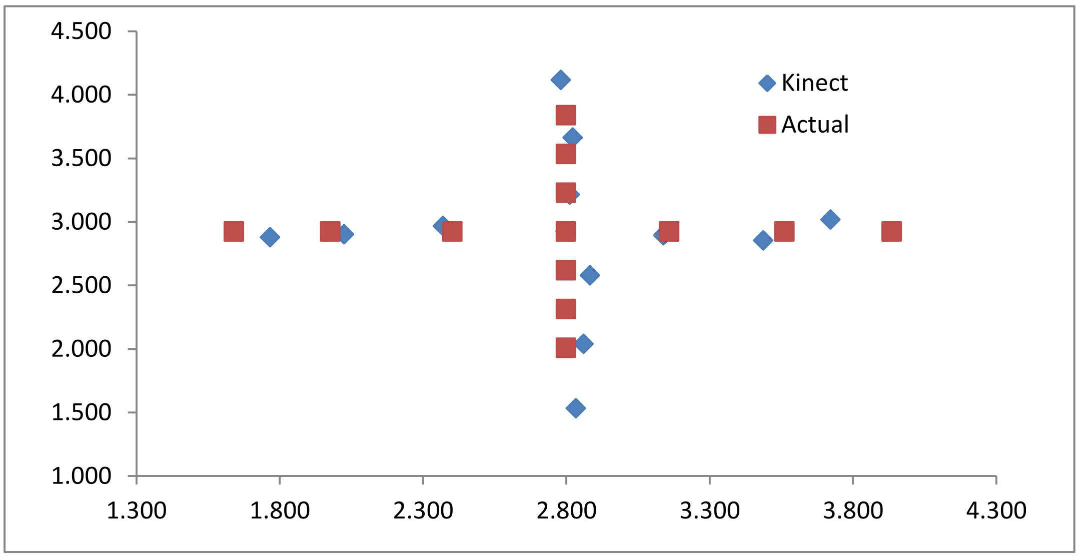


Figure 3.8 Experiments Setup and Result of Kinect ID

It is obviously shown that Kinect gives a better measurement around the middle than sides, and in x-direction than y-direction. The Kinect sometimes focuses on the body edge instead of the center of the drone, which can cause the offset as large as 0.25 [m] at some locations, given the drone's hull has a diameter of 0.50[m]. This offset tends to be more obvious at the sides than the middle and along the y axis than x. The exact position, position detected by the Kinect and the standard deviation is quantized in the table below.

Table 3.2 Kinect Measurement Data

	Real X	Real Y	Kinect X	Kinect Y	X_Std	Y_Std
Mid	2.799	2.923	2.799	2.923	0.038	0.012
Left1	2.403	2.923	2.371	2.967	0.029	0.014
Left2	1.977	2.923	2.025	2.901	0.012	0.028
Left3	1.641	2.923	1.767	2.878	0.019	0.067
Right1	3.160	2.923	3.139	2.893	0.013	0.020
Right2	3.561	2.923	3.487	2.853	0.019	0.051

Table 3.2 (cont.)

Right3	3.936	2.923	3.722	3.017	0.046	0.128
Back1	2.799	3.228	2.813	3.214	0.029	0.054
Back2	2.799	3.533	2.822	3.661	0.013	0.025
Back3	2.799	3.837	2.782	4.117	0.007	0.040
Front1	2.799	2.618	2.884	2.577	0.002	0.010
Front2	2.799	2.313	2.861	2.039	0.005	0.006
Front3	2.799	2.009	2.833	1.531	0.005	0.036

The experiment results also showed that X and Y channels are not independent. For example, the farther Y is from the center, the more inaccurate X is along the same line. So is X. Thus the most accurate model would be a two-dimensional look-up table. However as a start I assumed the two channels are independent. With the assumption the linear fit of the Kinect's X position and real X can be linear fitted with the data from left to right as.

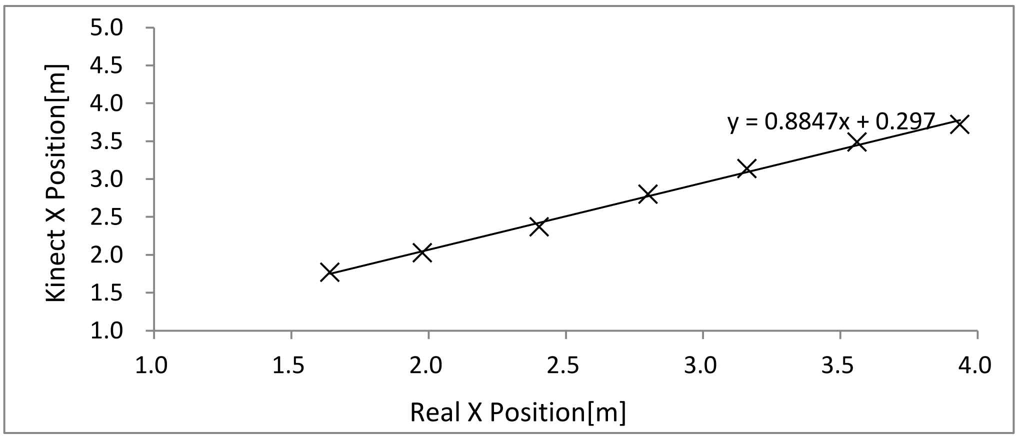


Figure 3.9 Linear Fit of Kinect X Reading and Real X Position

Apply the same method on Y from back to front, the linear fit curve is

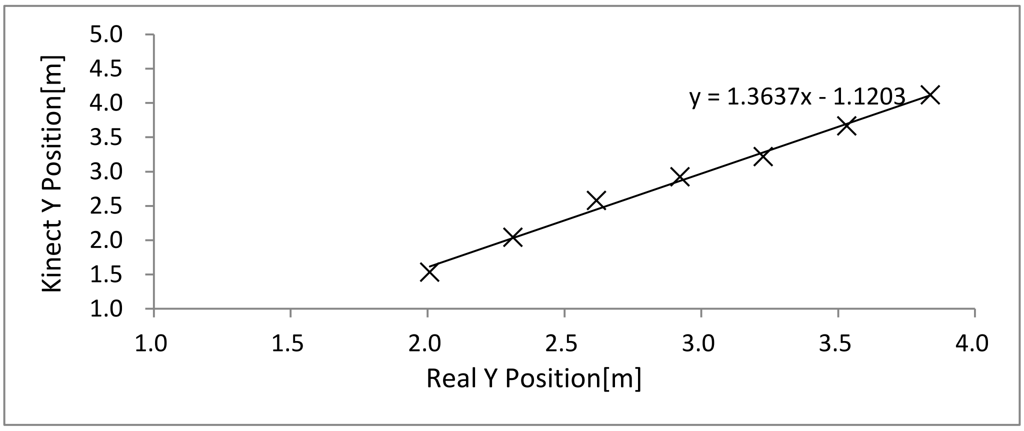


Figure 3.10 Linear Fit of Kinect Y Reading and Real Y Position

Again assuming the noise is Gaussian independent of X and Y coordinates, then each channel's noise's standard deviation is estimated by the average of all standard deviations of the corresponding channel. The modeling equations for Kinect X and Y position in the collecting range are given by

$$\begin{cases} X = P_x - 0.1153P_x + 0.297 + n(\sigma = 0.025) & (1.641 < X < 3.936) \\ Y = P_y + 0.3637P_y - 1.1203 + n(\sigma = 0.007) & (2.009 < Y < 3.837) \end{cases} \tag{3.10}$$

3.4.2 Gyroscope ID

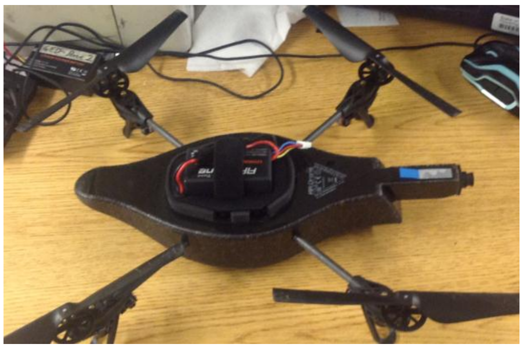


Figure 3.11 Experiment Setup for Gyro ID

Three axis gyros were identified by setting still and recording their outputs every 0.005 [s] (see experiment figure above). The constant bias could be found by taking average of the outputs over time, while the Angular Random Walk (ARW) and Bias Stability (BS) are obtained by running Allan Variance analysis. The following figures illustrate the output data along with short term average or linear fit, and Allan Variance of each gyro.

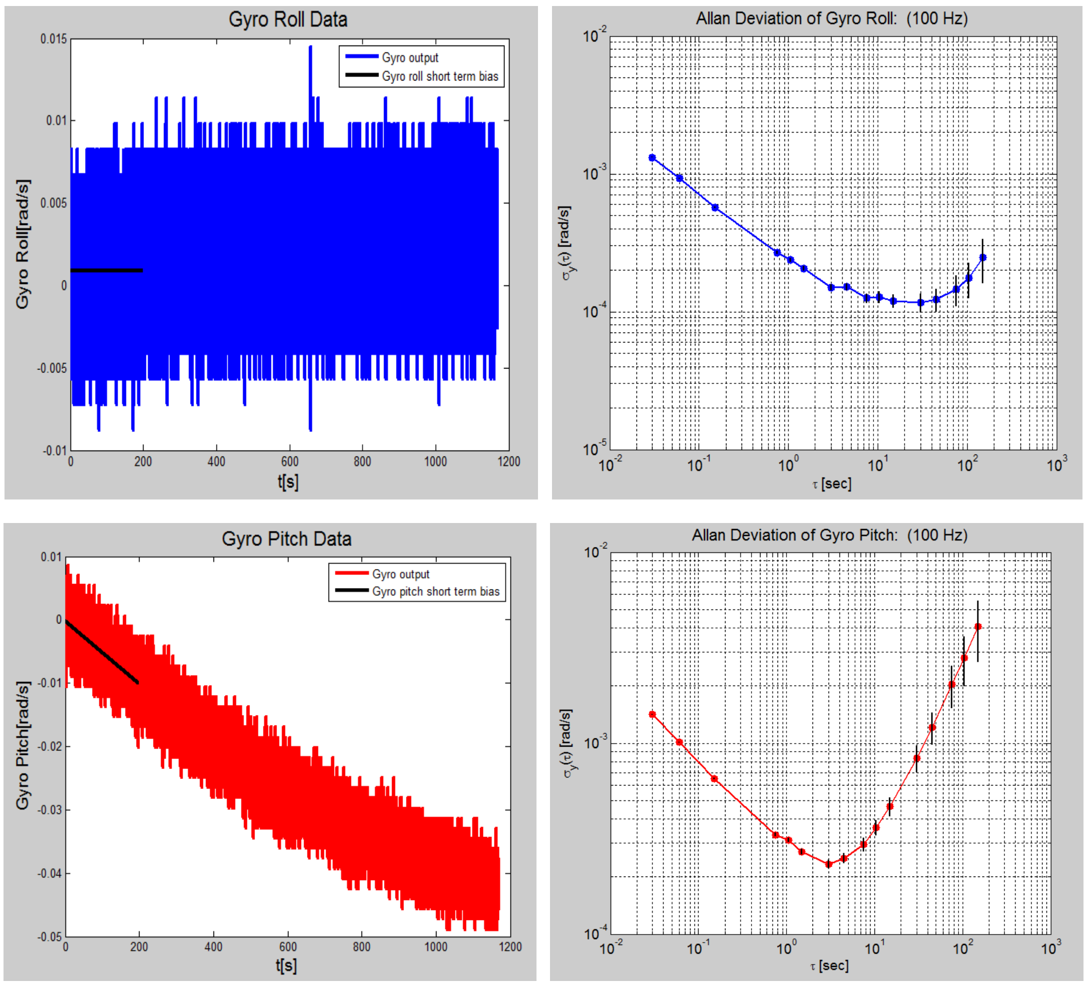


Figure 3.12 (cont. on next page)

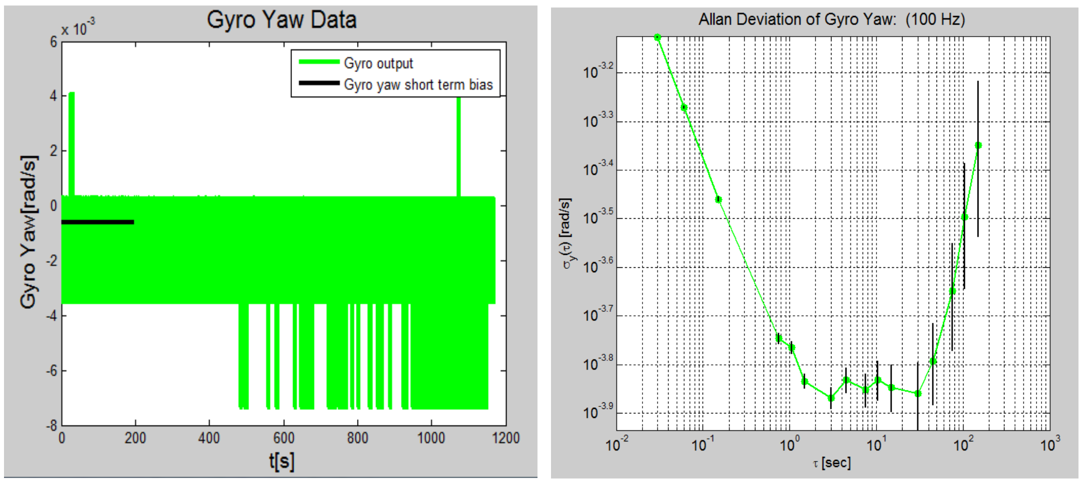


Figure 3.12 Output Gyro Data and Allan Variation Analysis

Since most of experiments will only be run in the first three minutes, only short term bias was evaluated by taking average of the first three minutes gyro output. The figures showing gyro outputs and bias are on the left side. From the figures, while Gyro Roll and Yaw data show the trend of a constant bias b_c, Gyro Pitch's bias is more likely to be linear with time. Thus, a first order varying bias model vs. time will be more accurate for pitch gyro.

The Angular Random Walk (ARW) could be found by the Allan deviation value at averaging time τ = 1[s]. From the figure it's clearly seen that white noise with a slop of -0.5 dominates at low τ. The standard deviation σ_n of the white noise n for each gyro can be found by

$$\sigma_n[°/s] = \frac{ARW}{\sqrt{\delta t}} \tag{3.11}$$

The Bias Stability/Bias Instability (BS) is found by the lowest point of Allan Variance curve. With more average time τ, the deviation tends to be caused by the bias instead of noise. With even larger average time τ the deviation shows the trend of increasing because of bias random walk. This phenomenon is very obvious in the Allan Variance curve of the pitch gyro, where the Allan deviation to the right of lowest has much larger slope than those of roll and yaw gyro, which indicates that a constantly increasing bias dominates the random walk, which validates the varying bias model. The standard deviation of random walk bias b_ω for each gyro then can be found by

$$\sigma_b[°/s] = BS[°/s]\sqrt{\delta t/\tau} \tag{3.12}$$

Table below lists the identified constant/varying bias b_c , ARW, BS, and standard deviation of white noise σ_n and of bias random walk σ_b for each gyro at sampling rate 25 Hz.

Table 3.3 Indentified Parameters of Gyros

	$b_c[rad/s]$	ARW$[rad/\sqrt{s}]$	BS$[rad/s]$ at $\tau[s]$	$\sigma_n[rad/s]$	$\sigma_b[rad/s]$
Gyro Roll	8.83×10^{-4}	0.00025	0.00012 at 30	0.00125	9.13×10^{-6}
Gyro Pitch	$(-4.97 - 6.22t) \times 10^{-5}$	0.0003	0.00023 at 3	0.0015	2.66×10^{-5}
Gyro Yaw	-5.67×10^{-4}	0.00018	0.00013 at 3	0.0009	1.50×10^{-5}

The following table lists the bias and ARW of several low-cost consumer gyros. ARW are calibrated to sampling rate 25 Hz. Prices are listed on official site if provided. Unfortunately BS is usually not listed for these gyros, but rather for more precise ones. According to the table, the identified parameters of AR.Drone's gyros are close to those of the low-cost consumer gyros.

Table 3.4 Specification of Selected Low Cost Gyros

	Type	**Bias$[rad/s]$**	**ARW$[rad/\sqrt{s}]$**	**Price($)**	**Datasheet**
Analog Devices ADXRS610	Yaw	N/A	0.0009	30	adxrs610.pdf
Analog Devices ADXRS450	One-Axis	N/A	0.00026	53.23	adxrs450.pdf
Analog Devices ADIS16250	Yaw	0.00028	0.0002	N/A	adis16250.pdf
InvenSense IDG-500	Dual-Axis	N/A	0.00023	N/A	IDG500.pdf

To further validate the model of gyros with identified parameters, the outputs are integrated over time in a simulation, and the result angles are compared to the integration of gyro in real system. Figure Set 3.9 illustrates the comparison of simulated and real angle. The time span is from 0 to 180 [s] (3 [min]). From the figure it can be seen that the model predicts real system within 10%.

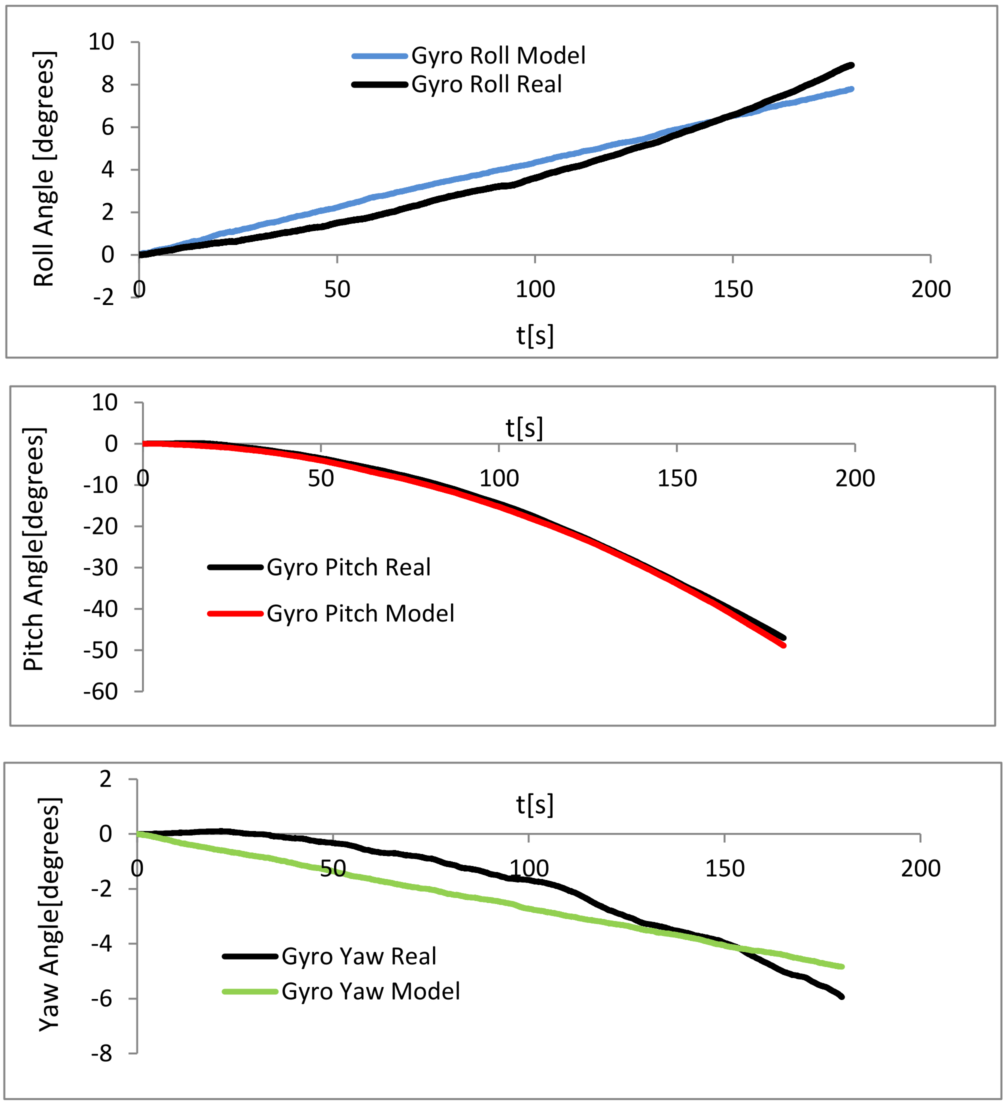


Figure 3.13 Comparisons of Simulated and Experimental Angles

Even though the model predicts the real sensor very well, it's important to mention that the gyros tend to behave quite differently in different runs. For example, we have observed the drift to be totally the opposite sign in even two neighbor tests. Thus, the model will not be able to present the real system in every situation. However, it gives a very good approach to how "bad" the real system can be, and can be utilized to test the robustness of controller and filter.

3.4.3 Altimeter ID

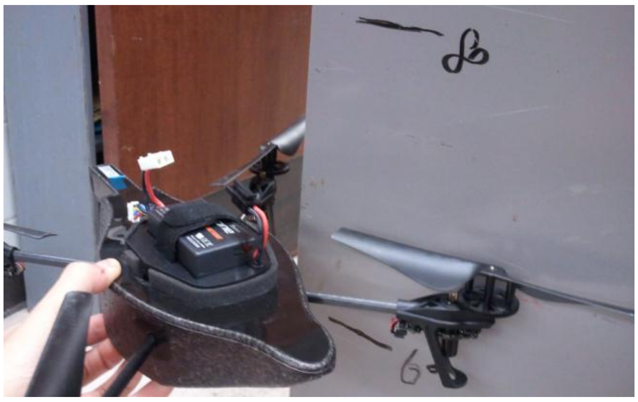


Figure 3.14 Altimeter ID

Altimeter was identified by placing at specific height and recording the data. The table below gives the true height and the average output of altimeter with standard deviation. The red color height gives the threshold of the altimeter.

Table 3.5 Altimeter ID Results

True Height[m]	Altimeter Average Output[m]	Standard Deviation[m]
0	0.2603	0.00045
0.2	0.2719	0.00019
0.26(Threshold)	0.2650	0.0002
0.4	0.4544	0.0032
0.6	0.7014	0.0014
0.8	0.9261	0.0007
1	1.1826	0.0013

Altimeter's output is around 0.27[m] when the true height is less than or equal to the threshold 0.26[m]. After the threshold, as the true height goes up the altimeter reading also increases, but also with an increasing offset. Figure below shows the linear fit trend line of the altimeter output with true height when true height is above 0.26[m].

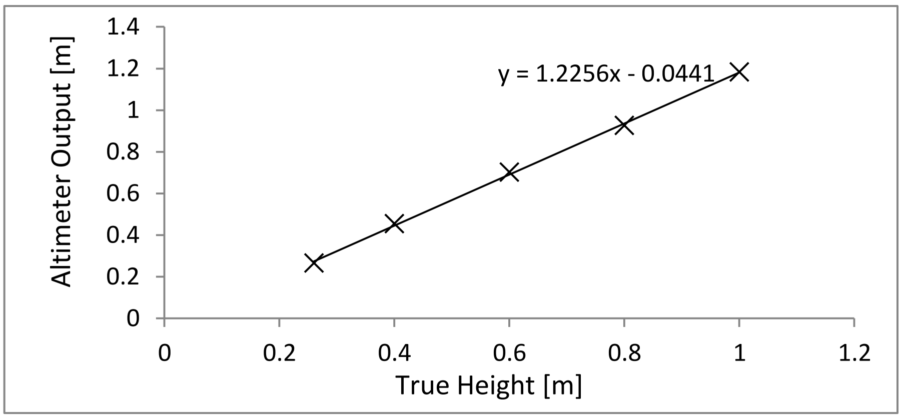


Figure 3.15 Linear Fit of Altimeter Output and True Height

Taking the average of noise standard deviation, assuming that the noise is Gaussian and independent of height, the modeling equation can be summarized as

$$\begin{cases} a = 0.27 + n(\sigma = 0.0002) & (h < 0.26) \\ a = h + 0.2256h - 0.0441 + n(\sigma = 0.0017) & (h > 0.26) \end{cases} \tag{3.13}$$

Chapter 4 Controllers and Filters Design

Based on dynamical models of the drone and sensors, local and global controller and filters are designed. Here “local” refers to the basic movements control accomplished by “on-board” sensors, while “global” refers to the position control sensed by the Kinect.

The four basic movement targets, throttle, roll, pitch and yaw, introduced in *2.4 Basic Movements*, are achieved by four independent “local” PID controllers designed in Section 1. However, angles and vertical speed cannot be read directly from sensor, which indicates that filter design is necessary. Thus Section 2 introduces three different ways to estimate angle from the rate gyro. These include integral of gyro through high pass filter, a complementary filter design combines the gyro and accelerometer, and an open-source Kalman filter design to determine angle from rate gyro and accelerometer. It also introduces an open-source technique “linear-regression” to determine the vertical velocity.

The “global” position controllers are also PID controllers built upon the “local” controllers. Their designs are included in Section 3. The Kinect can only sense global positions with noise, but the best PID controller performance requires speeds and smooth positions, which again requires a filter. The global filter designs, including two first order low pass filters, a complementary filter and a Kalman filter design is introduced in section 4.

4.1 Local Controller

The control targets correspond to throttle, roll, pitch and yaw movements are height, roll angle, pitch angle and yaw angle. Since four channels are independent, four separate PID controllers are designed for each channel. The following figure shows the block diagram architecture of four feedback PID controllers, sensors and filters.

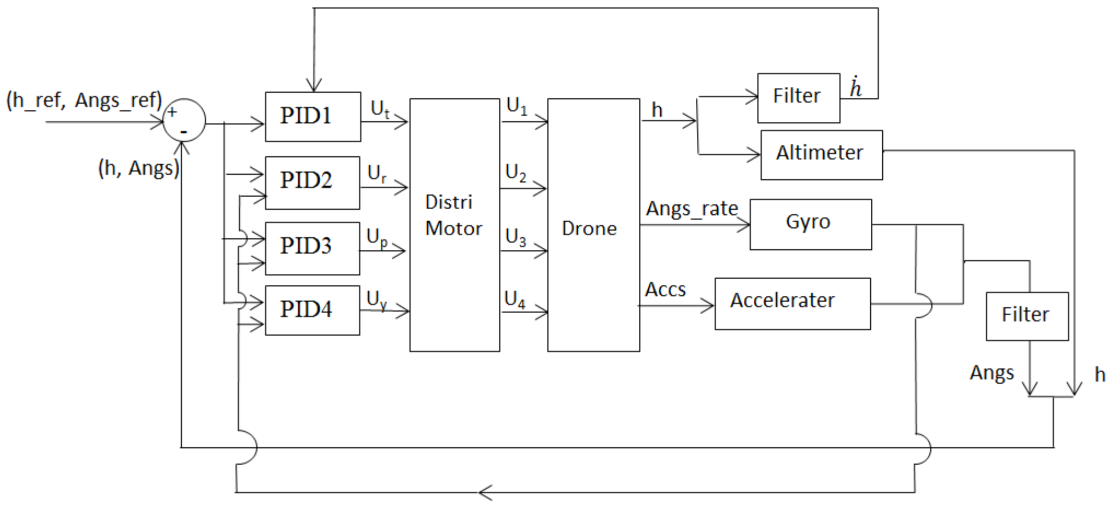


Figure 4.1 Local Closed-loop Feedback

The controllers’ outputs, i.e., control signals, shall have the same unit of the channels. For example, control signal of throttle should be in the unit of N, and that of roll should be in the unit of N.m. However, inputs to the AR.Drone are in the range of 0xff to 0x1ff for each motor, which corresponds to some RPM reference value, as determined in the AR.Drone's identification. Thus, we could make the control signals of each channel has the unit of RPM, and let the PID gains carry the unit. It efficiently saves the effort of relating force and torque to RPMs.

The control signals for each channel, in the unit of RPM, are calculated as following:

- ***Throttle***

  Given a constant reference height target h_{ref}, according to the architecture of PID controller, the control signal is given by

$$u_t = \left[k_{Pt}\left(h_{ref} - h\right) + k_{It}\int\left(h_{ref} - h\right) + k_{Dt}\left(\dot{h_{ref} - h}\right) + RPM_{hov}\right] / \sqrt{(cos\theta cos\phi)}$$

$$= \left[k_{Pt}\left(h_{ref} - h\right) + k_{It}\int\left(h_{ref} - h\right) + k_{Dt}(-\dot{h}) + RPM_{hov}\ \right] / \sqrt{(cos\theta cos\phi)} \quad (4.1)$$

  where k_{Pt}, k_{It} and k_{Dt} are the proportional, integral and derivative gain correspondingly, RPM_{hov} is the average speed of motors at hovering, h is the feedback from ultrasonic altimeter, θ and ϕ are the pitch and roll angle repectively . They are included to ensure balance in earth frame Z direction. Estimation of $\dot{\mathrm{h}}$ from h is by linear regression and will be discussed in next section.

  Extra consideration of a PID design, such as saturation of the error signal and control signal is also included. For constant position target tracking with force(RPM) as control input, integral term is usually not necessary, or much less compared to the other terms, because the transfer function from acceleration to position has a s^2 term that cancels the denominator s of the Laplace transform of a step reference input, which satisfies the internal modal principle. In other words, integration is not necessary to obtain zero steady state error. Thus, anti-windup is not a necessary part.

  To be used by the drone the controller has to be discrete with specific sampling period. It also needs to be written by some programming language, such that it can be built by the operating system's compiler as an executable program. Thus it's necessary to translate the design to some pseudo code. Given the sampling period T_s the pseudo code could be

```
Error = h_ref-h
Saturate Error
Integral = Integral + Error*$T_s$
Saturate Integral
$\mathrm{U} = k_P \cdot Error + k_I \cdot Integral + k_D \cdot (-\dot{h})$
Saturate U
$\mathrm{U = U/cos\theta cos\psi}$
```

- ***Roll/Pitch/Yaw***

  The controller of roll, pitch and yaw channel follows the same idea and will be discussed together. To track a constant reference angle Ang_{ref}, again following the architecture of PID controller, the control signal is given by

$$u_{Ang} = k_{P\_Ang}(Ang_{ref} - Ang) + k_{I\_Ang} \int (Ang_{ref} - Ang) + k_{D\_Ang}(-\dot{Ang}) \qquad (4.2)$$

  where k_{P_Ang}, k_{I_Ang} and k_{D_Ang} corresponds to the proportional, integral and derivative gain on the specific Angle, and $\dot{Ang}$ is the feedback from the rate gyro. Estimation of roll, yaw and pitch angle will be discussed in the next section. Saturation of signals is again included in the design, while anti-windup is not. The pseudo code is similar to that of throttle.

Recall that in Chapter 2 Section 4, how four basic movements were effected by each motor speed was introduced. Now we are interested in the way to quantize their relationship, i.e., a set of equations from control commands of each channel to those of each motor. Since they have the same unit of RPMs, one way is simply a geometric match by

$$\begin{cases} u_t = \frac{1}{4}(u_1 + u_2 + u_3 + u_4) \\ u_r = \frac{1}{4}\left(\frac{1}{\sqrt{2}}(u_1 + u_4) - \frac{1}{\sqrt{2}}(u_2 + u_3)\right) \\ u_p = \frac{1}{4}\left(\frac{1}{\sqrt{2}}(u_3 + u_4) - \frac{1}{\sqrt{2}}(u_1 + u_2)\right) \\ u_y = \frac{1}{4}(u_1 + u_3 - u_2 - u_4) \end{cases} \qquad (4.3)$$

where $u_1,\ u_2,\ u_3,\ u_4$ are the control commands of each motor, and u_t, u_r, u_p, u_y are those of throttle, roll, pitch and yaw channel correspondingly. Rearrange the equations to express $u_1,\ u_2,\ u_3,\ u_4$ by u_t, u_r, u_p, u_y:

$$\begin{cases} u_1 = u_t + \frac{1}{\sqrt{2}}(u_r - u_p) + u_y \\ u_2 = u_t + \frac{1}{\sqrt{2}}(-u_r - u_p) - u_y \\ u_3 = u_t + \frac{1}{\sqrt{2}}(-u_r + u_p) + u_y \\ u_4 = u_t + \frac{1}{\sqrt{2}}(u_r + u_p) - u_y \end{cases} \qquad (4.4)$$

The controller code for throttle, roll, pitch and yaw channels, along with the control signal distribution from channels to motors finish the local controller design.

4.2 Local Filter

Filters may not be necessary to run a successful model, however, is crucial to experiments because a lot of noise sources might have been underestimated in modeling. Filters give a more accurate and more importantly, much smoother results, which is especially important to PID control technique.

4.2.1 Linear Regression

Linear Regression is widely used by fitting dependent and independent variables with a straight line by ordinary least square [11]. In our application it is used to determine vertical velocity from height and time. This thesis paper will not go through the theories, but rather use the results directly and focus on the open-source algorithm.

Linear regression states that given a linear relationship between independent variable x and dependent variable y by

$$y = kx + b \tag{4.5}$$

Then with N number of data acquired, the least square linear fit coefficients are

$$\begin{cases} k = \dfrac{N\sum x_i y_i - \sum x_i \sum y_i}{N\sum x_i^2 - (\sum x_i)^2} \\ b = \dfrac{\sum y_i \sum x_i^2 - \sum x_i \sum x_i y_i}{N\sum x_i^2 - (\sum x_i)^2} \end{cases} \tag{4.6}$$

The open source code we used is the “AR.Drone attitude estimation driver” by Hugo Perquin, with GNU General Public License published by the Free Software Foundation [12]. The program is designed for AR.Drone attitude estimation, including angles and angle rates, height and height rate, and accelerations. The main algorithm contains linear regression and the Kalman filter design by Tom Pycke. The linear regression pseudo code can be concluded as

```
Note: comment is after //

//Initialize
Specify N number of data
OffIndex=0
for i=0 to N-1
    Sum_x=Sum_x+i
    Sum_xx=Sum_xx+i*i
    Y[i]=0
end
Denominator= Sum_xsquare- Sum_x* Sum_x/N

//Calculation, called every sample time T
Y[OffIndex]=y_new        //update
StartIndex=(OffIndex+1)Mod N
for i=0 to N-1
    Moving Index=(i+StartIndex)Mod N
    Sum_y=Sum_y+Y[MovingIndex]
    Sum_xy=Sum_xy+i*Y[MovingIndex]
end
OffIndex=StartIndex
Numerator=Sum_xy-Sum_x*Sum_y/N
K=numerator/denominator * T
```

4.2.2 Integral with High Pass Filter

The most direct way to obtain angles from onboard sensors is integral of gyros. It is fast and accurate in short term. However as we have seen in the identification of the gyros, due to the bias the integrated result become very inaccurate at long run. The most straight forward way is then to manually subtract the bias from the gyro outputs, however the bias itself vary in different runs. Thus, the most robust way is high pass the gyro reading and then integrates. The following figure gives the transfer blocks from rate gyro output to angle through a first order high pass filter and integrator.

$$\begin{tikzcd} g \arrow[r] & \boxed{\frac{\tau s}{\tau s+1}} \arrow[r] & \boxed{\frac{1}{s}} \arrow[r] & \theta \end{tikzcd}$$

Figure 4.2 Low Pass Filter Structure

The overall transfer function is then

$$\Theta(s) = \frac{\tau}{\tau s+1} \cdot G(s) \tag{4.7}$$

Notice the result transfer function is actually a low pass filter applied on gyro. To write the transfer function in digital form, apply bilinear transform with sampling period T_s [13]

$$s = \frac{2}{T_s} \cdot \frac{z-1}{z+1} \tag{4.8}$$

and rearrange the difference equation, the digital filter can be

$$\Theta_k = \left(\frac{2\tau - T_s}{2\tau + T_s}\right)\Theta_{k-1} + (\frac{2T_s}{2\tau + T_s})\tau g \tag{4.9}$$

where Θ_{k-1} is the last step angle, g is the current step gyro output, Θ_k is the filtered angle and τ is the pole to be tuned. It should be very carefully chosen such that the bias is effectively filtered instead of the fast response of gyro. The less τ is, the more active the high pass filter is, however too small τ will make the gyro ineffective by filtering out all changes.

4.2.3 Complementary Filter

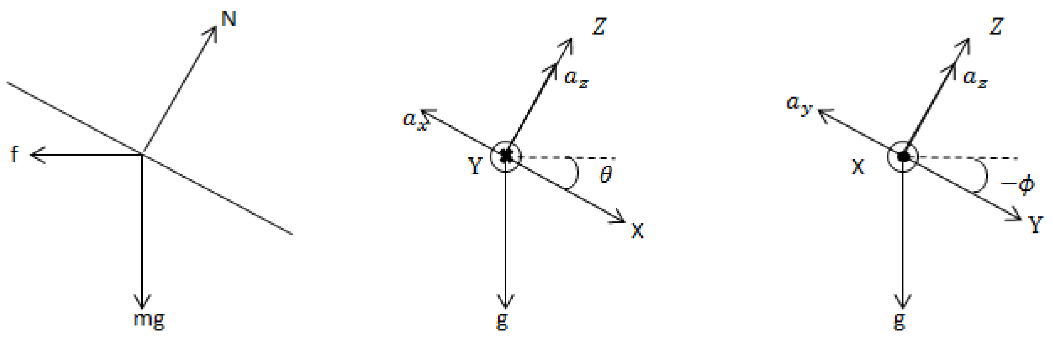


Figure 4.3 Free Body Diagram of Drone without Acceleration

The figure above gives the free body diagram (FDB) of the drone flying in constant speed with a positive static pitch angle theta (y axis pointing in). The forces are balanced with the thrust acting on positive body z axis, friction acting on the opposite direction of velocity, and gravity acting downward. At this stage, the accelerometer will sense the acceleration produced by every force except the gravity. However, since the sum of all the other forces is equal to gravity, the resultant acceleration on body frame x-axis and z-axis will be $g * sin(\theta)$ and $-g * cos(\theta)$, respectively. Thus, the pitch angle can be estimated by accelerometers by

$$\theta = arctan\left({}^{-a_x}/_{a_z}\right) \tag{4.10}$$

The same idea applies to the roll angle. A static negative roll angle $-\phi$ (x axis pointing out) balance produces a positive acceleration $g * \sin(\phi)$ on z-axis and $-g * \cos(\phi)$ on y-axis. Thus, the roll angle can be estimated through accelerometers by

$$\phi = arctan\left({}^{a_y}/_{a_z}\right) \tag{4.11}$$

However accelerometers are usually very dangerous to use, because of the complicated dynamics during a run. In other words, the accelerometer reading is usually very different from expected while rotational and linear acceleration and vibration comes in. However, in the long run it gives non-drifting reading that can be used to correct the drift from gyro. The complementary filter [14] then combines the two by high pass the integrated angle from gyro, and low pass the angle estimated from accelerometers, such that their sum would be a fast and accurate angle output [15]. The figures below illustrate the idea of the complementary filter and its realization.

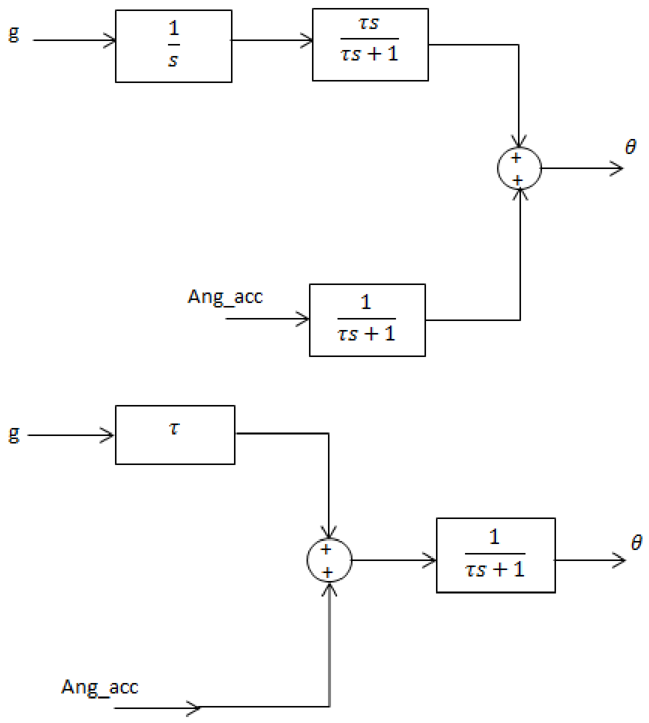


Figure 4.4 Local Complementary Filter and its Realization

The overall transfer function from g and Ang_acc to Θ is given by

$$\Theta(s) = \frac{\tau}{\tau s + 1} \cdot (\tau G(s) + Ang\_acc(s)) \tag{4.12}$$

Again apply bilinear transform and rearrange the difference equation we have

$$\Theta_k = \frac{2T}{2\tau + T} \cdot Ang\_acc + \frac{2\tau - T}{2\tau + T}(\Theta_{k-1} + \frac{T}{1 - \frac{T}{2\tau}} g) \tag{4.13}$$

Usually τ is chosen to be much larger than sampling period T to actively high and low pass signals. The equation can be simplified as

$$\Theta_k = \alpha \cdot Ang\_acc + (1 - \alpha)(\Theta_{k+1} + Tg) \tag{4.14}$$

where α is some value close to but less than 1, and is equal to

$$\alpha = \frac{2T}{2\tau + T} \tag{4.15}$$

4.2.4 Kalman Filter

Another way to combine angle estimation from gyro and accelerometer is the Kalman Filter design in sensor fusion [16][17][18]. The filter we used is the open source Kalman Filter design by Tom Pycke [19]. It sets up the structure with a state vector $x^T = (\theta, b)$, where θ is the angle and *b* is the gyro bias, an input *g* which is the gyro reading, and an output *z* which is the angle measured by the accelerometer. Then following the structure of a discrete Kalman filter

$$\begin{cases} X_k = F_k X_{k-1} + B_k U_k + W_k & W_k \sim N(O, Q_k) \\ Z_k = H_k X_k + V_k & V_k \sim N(O, R_k) \end{cases} \tag{4.16}$$

The state space equations are written as

$$\begin{cases} \begin{pmatrix} \theta_k \\ b_k \end{pmatrix} = \begin{pmatrix} 1 & -\delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \theta_{k-1} \\ b_{k-1} \end{pmatrix} + \begin{pmatrix} \delta t \\ 0 \end{pmatrix} (g_k \quad 0) + W_k & W_k \sim N(O, Q_k) \\ Z_k = (1 \quad 0) \begin{pmatrix} \theta_k \\ b_k \end{pmatrix} + V_k & V_k \sim N(O, R_k) \end{cases} \tag{4.17}$$

Now the problem is to evaluate this Kalman filter design by relating the equations to the sensor models, suggesting and measure possible covariance matrices, translating the design to executable language and testing its performance through experiments.

In the predict step, for the first component, angle, of the state vector we have

$$\theta_k = \theta_{k-1} + \omega \cdot \delta t \tag{4.18}$$

where ω is the exact angle rate and δt is the sampling period. Recall the model of rate gyro

$$\mathrm{g} = \omega + \mathrm{b} + n(\sigma = \sigma_n) \tag{4.19}$$

where ω is the exact angle rate, n is the gyro noise with standard deviation σ_n and b is the bias. Replace ω in Equation 4.19, it can be rewritten as

$$\theta_k = \theta_{k-1} - b \cdot \delta t + g \cdot \delta t - n(\sigma = \sigma_n) \cdot \delta t \tag{4.20}$$

The second component, bias, of the state vector, can be estimated by

$$b_k = b_{k-1} + n_b(\sigma = \sigma_b) \tag{4.21}$$

where n_b is a random walk noise whose standard deviation is equal toσ_b. Combining 4.20 and 4.21, the state space equation can be written as

$$\begin{pmatrix} \theta_k \\ b_k \end{pmatrix} = \begin{pmatrix} 1 & -\delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \theta_{k-1} \\ b_{k-1} \end{pmatrix} + \begin{pmatrix} \delta t \\ 0 \end{pmatrix} (g \quad 0) + \begin{pmatrix} -n\delta t \\ n_b \end{pmatrix} \tag{4.22}$$

Assuming gyro white noise and bias random walk are independent, the predict covariance matrix, Q, can be written as

$$Q = \begin{pmatrix} Cov(-n\delta t, -n\delta t) & Cov(-n\delta t, n_b) \\ Cov(n_b, -n\delta t) & Cov(n_b, n_b) \end{pmatrix} = \begin{pmatrix} \sigma_n^2 \delta^2 t & 0 \\ 0 & \sigma_b^2 \end{pmatrix} \tag{4.23}$$

The update step involves the measured output by

$$Z_k = \arctan\left({}^{a_1}/_{a_2}\right) = (1 \quad 0) \begin{pmatrix} \theta_k \\ b_k \end{pmatrix} + n_m \tag{4.24}$$

where a_1 and a_2 are the true accelerations depending on what angle is being measured, and n_m is the measurement noise. Even though we could estimate the noise level of each accelerometer channel, the noise level of $\arctan\left({}^{a_1}/_{a_2}\right)$ is still difficult to be estimated. Thus, it's a good idea to tune the standard deviation in experiments.

Following the steps of discrete Kalman Filter the pseudo code could be as following

```
//Initialize
    $x = \begin{pmatrix} \theta \\ b \end{pmatrix} = 0$,  $P = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$,  $Q = \begin{pmatrix} \sigma_n^2 \delta^2 t & 0 \\ 0 & \sigma_b^2 \end{pmatrix}$,    $R = \sigma_z^2$
//Predict, called every $\delta t$
                    $x = Fx + Bu$
                    $P = FPF^T + Q$
//Update, called after predict
                    $y = z - Hx$
                    $S = HPH^T + R$
                    $K = PH^T S^{-1}$
                    $x = x + Ky$
                    $P = (I - KH)P$
```

4.3 Global Controllers

Autonomous position controller is currently designed to be an outer loop upon local controllers. The main control targets are earth frame (global) X and Y positions, which could be sensed by the Kinect. The control is achieved by fixing height and yaw references, altering pitch and roll references according to positions by two independent PID controllers. The whole architecture is shown below:

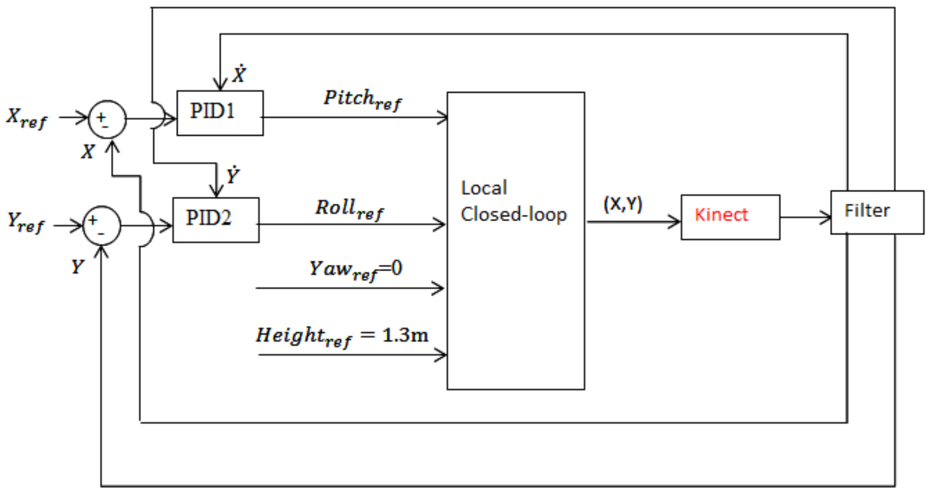


Figure 4.5 Global Closed-loop Feedback

Note that Kinect is colored red because it is running outside of the AR.Drone's operating system. The filter could also be run outside, but currently not because some of the filter's design requires the local angle inputs. Fortunately the Drone's CPU is powerful enough to run everything together.

The data communication between AR.Drone, Kinect and PC is achieved by

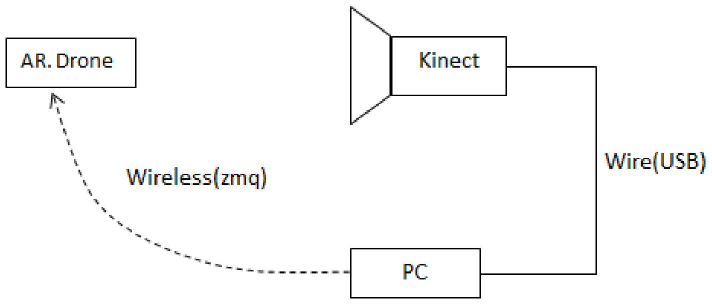


Figure 4.6 Data Communication between Drone and Kinect

Given a fixed reference input Y_{ref}, following PID architecture, the control signal ϕ_{ref} generated by the global controller roll channel can be

$$\phi_{ref} = k_{P1}(Y_{ref} - Y) + k_{I1}\int(Y_{ref} - Y) + k_{D1}(\dot{Y_{ref} - Y}) \quad (4.25)$$

$$= k_{P1}(Y_{ref} - Y) + k_{I1}\int(Y_{ref} - Y) - k_{D1}\dot{Y}$$

The controller's pitch channel works the same way, but tracks reference input X_{ref} and generates control signal θ_{ref} by

$$\theta_{ref} = k_{P2}(X_{ref} - X) + k_{I2}\int(X_{ref} - X) + k_{D2}\dot{(X_{ref} - X)} \quad (4.26)$$

$$= k_{P2}(X_{ref} - X) + k_{I2}\int(X_{ref} - X) - k_{D2}\dot{X}$$

The pseudo code is similar to it of the local controller.

4.4 Global Filter

To obtain velocity from Kinect positions, and smooth the Kinect position feedbacks, a low pass filter, Complementary Filter and Kalman Filter have been designed. Following the idea of the local angle filters, low pass filter only utilizes the Kinect position data, while the Complementary Filter and Kalman Filter design also need an acceleration input besides the position feedback.

4.4.1 Low Pass Filter

Position signal can be smoothed by going through a first order transfer function as

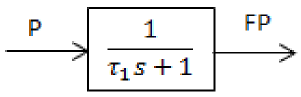


Figure 4.7 Low Pass Filter of Position

where P denotes the position feedback from Kinect and FP is the filtered position. Apply bilinear transform and rewrite the difference equation, we have

$$FP_k = \frac{2\tau_1 - T}{2\tau_1 + T}FP_{k-1} + \frac{2T}{2\tau_1 + T}P \quad (4.27)$$

where FP_k denotes the current step filtered position and FP_{k-1} is the last step filtered position, T is the sampling period, and $1/\tau_1$ is the pole to be tuned. The more τ_1 is, the closer the pole is to the origin, which results in a more active low pass filter. However, too large τ_1 will over filter the Kinect position which result in a very slow update of position. It is usually more convenient to rewrite the equation as

$$FP_k = \alpha_1 FP_{k-1} + (1 - \alpha_1)P \quad (4.28)$$

and tune α_1. The tuning direction of α_1 follows the same way as τ_1, except that α_1 has to be within range of [0, 1].

Velocity signal is similarly filtered by putting Kinect measured position through a first order low pass filter and a derivative as

$$\begin{tikzcd} {} \arrow[r, "P"] & \boxed{\frac{1}{\tau_2 s+1}} \arrow[r] & \boxed{s} \arrow[r, "V"] & {} \end{tikzcd}$$

Figure 4.8 Low Pass Filter of Velocity

Bilinear transform is applied to replace s in the overall transfer function, the result difference equation is

$$\mathrm{V}_k = \left(\frac{2\tau_2 - T}{2\tau_2 + T}\right) \cdot \mathrm{V}_{k-1} + \left(\frac{2T}{2\tau_2 + T}\right) \cdot \frac{P_k - P_{k-1}}{T} \qquad (4.29)$$

which is really similar to that of position low pass filter. Here $\cdot \mathrm{V}_{k-1}$ is the last step velocity, $(P_k - P_{k-1})/T$ gives the updated velocity by dividing the difference of current step and last step position over sampling time. τ_2 functions the same as τ_1 in terms of efficiency of filtering. The equation can be simplified as

$$\mathrm{V}_k = \alpha_2 \mathrm{V}_{k-1} + (1 - \alpha_2)\frac{P_k - P_{k-1}}{T} \qquad (4.30)$$

which again can be tuned by playing with α_2 in the similar way of α_1.

4.4.2 Complementary Filter

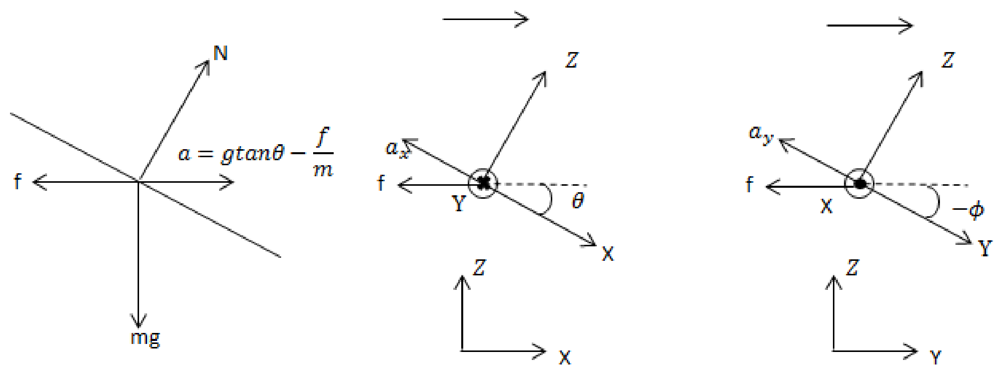


Figure 4.9 Free Body Diagram of Drone with Translational Acceleration

Advance filter involves another signal to be used to estimate position. We decided to utilize the global accelerations, which could be estimated from the flying angles and accelerometer's reading. Again get back to the FBD but during the transient response when forces are not balanced, a positive pitch angle theta will result in a positive acceleration in global X direction, while the resistance force prevents it by producing an opposite acceleration. Assuming global Z axis balancing, the net acceleration could be calculated by

$$a = gtan(\theta) - \frac{f}{m} \qquad (4.31)$$

The acceleration due to resistance force could be estimated from the accelerometer reading.

Since the accelerometer only senses forces that compensate the gravity, but the thrust force F has no components acting on the body x-axis, neglecting angular acceleration, the accelerometer only reads the air resistance force's component on the body x-axis. In equation

$$a_x = -\frac{f}{m} \cdot cos(\theta) \tag{4.32}$$

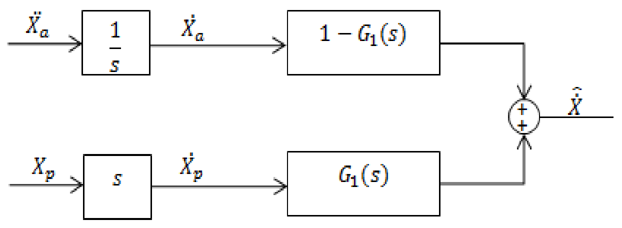
When θ is small $tan(\theta)$ could be estimated by theta and $cos(\theta)$ is close to 1. Thus the net global X acceleration could be estimated by

$$a_X = g\theta + a_x \tag{4.33}$$

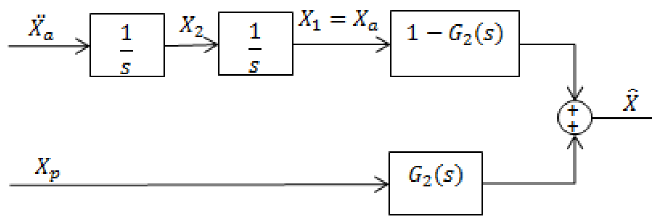
Similarly for global Y channel, a negative roll angle produces a positive global acceleration that compensated by the resistance force, which produces a negative reading on y-axis accelerometer. The net global Y acceleration thus could be estimated by

$$a_Y = -g\phi + a_y \tag{4.34}$$

The acceleration could be used to estimate global positions through integration, while the measured position is available for correction. The double integration would produce huge drifts on position estimation, while the measured position is very noisy. This again produces an idea of applying a Complementary Filter that high passes the integration and low passes the measured position to obtain smooth and accurate position. The same idea works on the velocity, where we could high pass the integration of acceleration and low pass the derivative of position. The idea is illustrated by the block structures below:



$$G_1(s) = \frac{b}{s^2 + as + b} \qquad 1 - G_1(s) = \frac{s^2 + as}{s^2 + as + b}$$



$$1 - G_2(s) = \frac{s^2}{s^2 + as + b} \qquad G_2(s) = \frac{as + b}{s^2 + as + b}$$

Figure 4.10 Complementary Filter on Position and Velocity

One way of its realization was introduced in [20] and shown below. It is much easier for application.

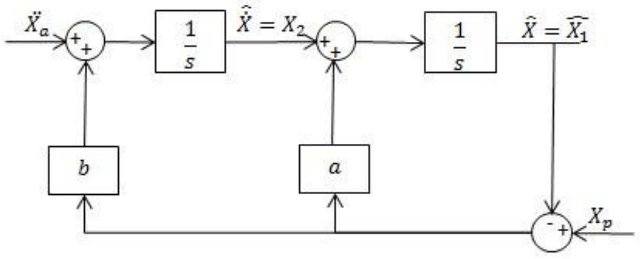


Figure 4.11 Realization of Global Complementary Filter

4.4.3 Kalman Filter

Even though the drone has high nonlinear complicated dynamics, the global position control itself only involves the global accelerations. Thus, a simplified Kalman Filter could be designed, focusing only on how the global accelerations affect the global positions. A linear Kalman Filter could be set up with a state vector $x^T = (P, V)$ where P denotes position and V is the velocity, a known input a which denotes the acceleration, and a measurable output Z which denotes the position sensed by the Kinect. Then following the discrete Kalman Filter setup

$$\begin{cases} X_k = F_k X_{k-1} + B_k U_k + W_k & W_k \sim N(O, Q_k) \\ Z_k = H_k X_k + V_k & V_k \sim N(O, R_k) \end{cases} \tag{4.35}$$

the discrete state space equations are

$$\begin{cases} \begin{bmatrix} P_k \\ V_k \end{bmatrix} = \begin{bmatrix} 1 & \delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} P_{k-1} \\ V_{k-1} \end{bmatrix} + \begin{bmatrix} \delta t^2/2 \\ \delta t \end{bmatrix} a_k + W & W \sim N(0, Q) \\ Z_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_k \\ V_k \end{bmatrix} + \alpha \cdot o + V & V \sim N(0, R) \end{cases} \tag{4.36}$$

where α and o are the scale factor and offset of the Kinect, obtained from the identification. If they are dependent of P_k instead of constants, the H matrix should be changed accordingly. The covariance matrices Q and R could be estimated by

$$\begin{cases} Q = \begin{bmatrix} \delta t^4/4 & \delta t^3/2 \\ \delta t^3/2 & \delta t^2 \end{bmatrix} \sigma_a^2 \\ R = \sigma_z^2 \end{cases} \tag{4.37}$$

where σ_a and σ_z is the standard deviation of input acceleration disturbance and measurement noise, respectively. While σ_z could be obtained from Kinect's ID, it is very difficult to measure the input disturbance σ_a, especially when the way of estimating acceleration is a combination of angle and accelerometer. Thus it will be left to be tuned during experiments. The initialize, predict and update process is similar to that of the local Kalman Filter.

Chapter 5 Simulation and Experiments

This chapter puts together all previous content.

The first stage of simulation utilized the model designed and conducted by Jan Vervoorst in MATLAB Simulink, for his own-built quadrotor and AR.Drone. The model he designed is a very thorough one that includes quadrotor dynamics, sensor noise, motor dynamics and his own controller. Thanks to his great work and generosity, I was able to save a large amount of time to initiate simulation of the AR.Drone model and controller. However, since the first stage was only to find a preliminary controller that stabilizes the AR.Drone model with height and angle references, and obtain a set of gains for global controller, I have changed the Simulink model accordingly, such that it only has the drone dynamics, the local controllers and the global controllers, but not any of other dynamics or noise sources. This "perfect" simulation design is briefly introduced in Section 1.

Once the preliminary model and controller was validated, C++ was used to achieve more "realistic" simulation with all sensor models and filters design. C language is used primarily because of its easiness of development and convenience of interface. For instance, it's able to be run in real time by simply including the timing library. The controller code written in C easily interfaces with the AR.Drone's driver code, in the way that they can be compiled and built together as an executable program for Arm operating system. It also provides a platform for animation discussed in the next chapter. This convenience is prominent during experiments, when one has to frequently go between simulation and experiments.

Section 3 includes all experiments on the AR.Drone. It first evaluates the local filters design by comparing their estimated angle outputs when drone is at static and roll reference following, such that the best filter is selected. Then with the selected filter it tests all local references tracking, i.e., specific height, roll, pitch and yaw references, by tuning controller gains obtained from simulation. The last part contains the global position tracking from point to point, and a specific trajectory tracking, by tuning gains from simulation and comparing the global filters design.

Section 4 evaluates the model by comparing the simulation results to experiments. It compares the simulated response of local references to experiments, by the same local filter and controller obtained from experiments. Notice that the simulation gains have been adjusted accordingly to the ones used in experiments. The results are used to evaluate drone and sensor's identification and modeling. Section 4 also contains the comparison between simulated global position tracking results and those of experiments. The comparison evaluates the overall modeling, including the Kinect model and global controller.

5.1 "Perfect" Simulink Model

The first stage model is designed to only test the drone's dynamical model and local controller algorithm. None of the sensor's model and filter's algorithm is included. The closed loop structure is shown in Figure 5.1 below

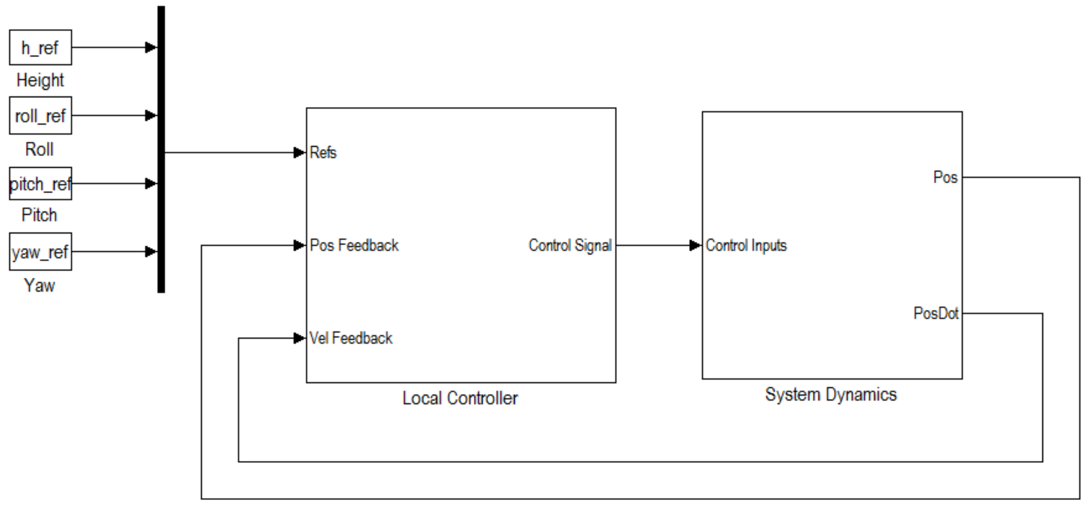


Figure 5.1 Simulink Block of Local Closed-loop Feedback

Following references inputs, the "Local Controller" subsystem contains four discrete PID controllers that take in the positions and velocities feedback of each channel (throttle, roll, pitch and yaw), and generate channel control signals. It also distributes the channel control signals to each motor. The figure below contains the details of the block.

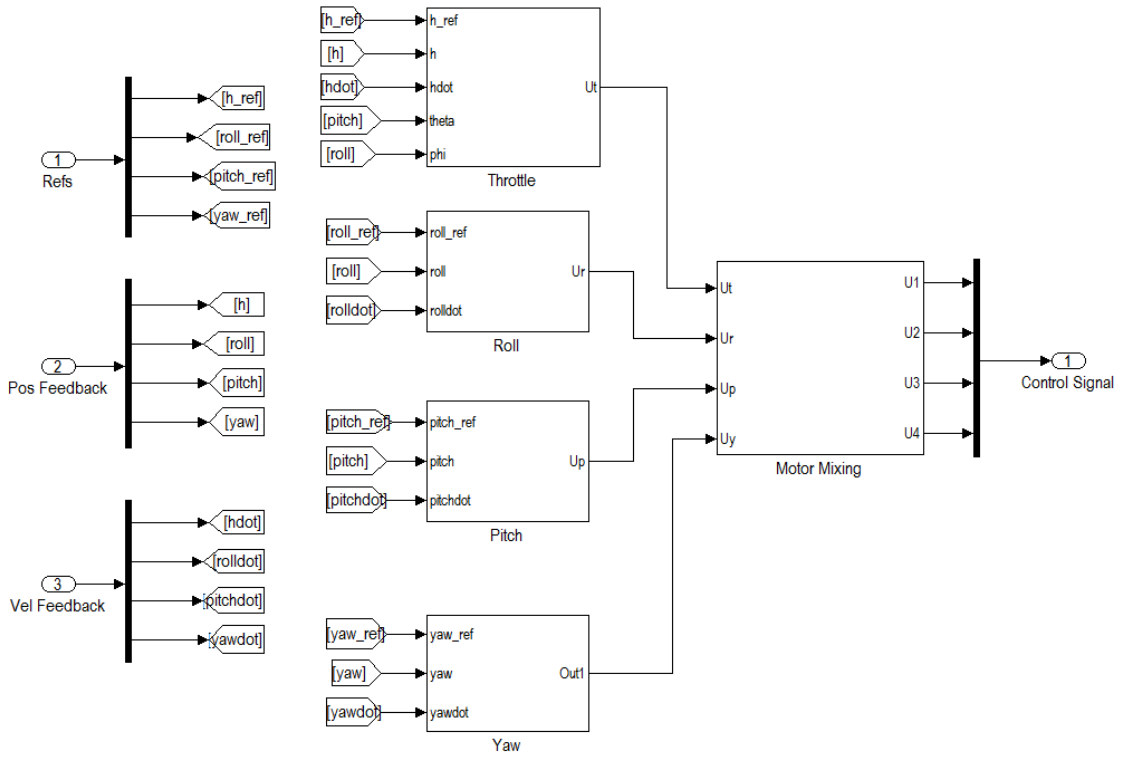


Figure 5.2 Simulink Block of Local Controller

The "System Dynamics" subsystem contains the map from control signal (motor RPMs) to thrust and torque, based on the identified thrust and drag parameters. It also contains the dynamical equations written in embedded MATLAB function, where the inputs are the state vector and force vector, and the output is the state_dot vector. The state_dot vector is then integrated continuously with an initial value to be returned to the dynamics. Two selectors are used to pick up the position (height, roll, pitch, yaw angles) and velocity (vertical speed, roll, pitch, yaw speed) vectors to feedback to the control system.

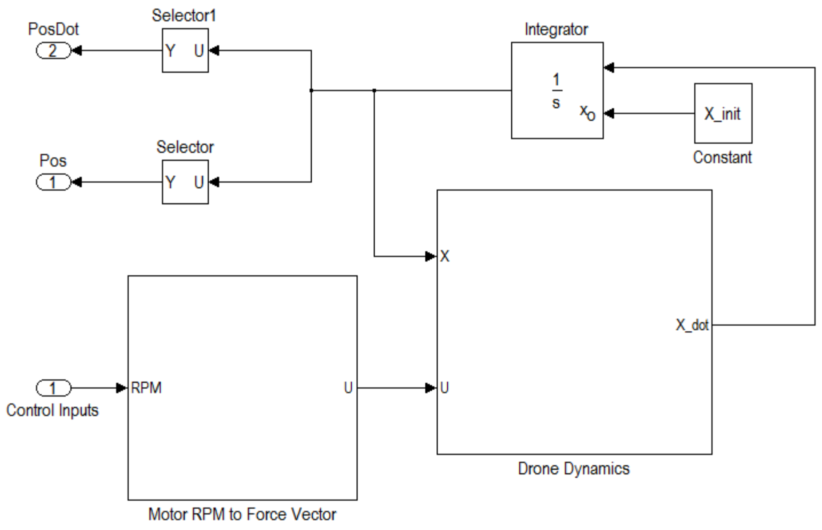


Figure 5.3 Simulink Block of Drone Dynamics

Please refer to Appendix B for details of PID controller of each channel and “motor mixing” in Local Controller subsystem, and details of the “motor RPM to Force Vector” and “Drone Dynamics” in System Dynamics subsystem. They are simply the Simulink realization of the corresponding equations.

As introduced in Chapter 4 the global controller is built on the local closed loop. The Kinect model and global filters are again not considered at this stage. The following figure illustrates the overall structure.

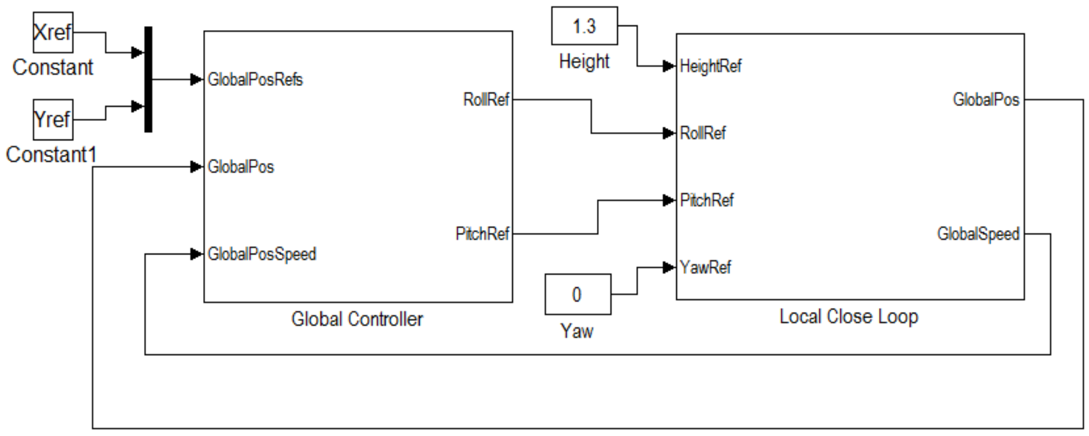


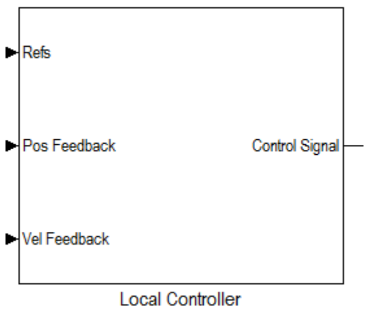
Figure 5.4 Simulink Block of Global Closed-loop Feedback

The “Global Controller” subsystem takes in the global position references, the current global position and velocity and executes the PID control algorithm. The output control signals are reference roll and pitch angles that go into the local closed loop shown in Figure 5.1 with fixed yaw and height references. The detailed blocks are very similar to those of local PID controllers.

5.2 “Realistic” C++ Model

The first step is to translate existing MATLAB Simulink blocks to C. Each main Simulink subsystem (Local Controller, System Dynamics and Global Controller) is written as a c file, the sub-blocks within the subsystem are translated to sub-functions belong to the c file, and the wire connections between sub-blocks are translated to a higher level function that contains all the sub-functions.

For example, the “Local Controller” subsystem is translated to “ConDyn.c”, the sub-blocks “Throttle, Roll, Pitch and Yaw” and “MotorMixing” are translated to functions “PICon()” and “MotorMixing()”, and the connections from inputs to sub-blocks and outputs are translated to the higher level function “ConDyn()”. The translation will result in a C file that contains all the information of the subsystem “Local Controller”. The following figure illustrates the translational process. The ConDyn.c block on the right only gives the pseudo code. Please refer to Appendix C for the detailed source code.



$\longrightarrow$

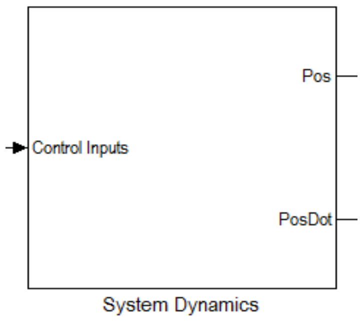
```
ConDyn(Ref, Pos, PosDot, RPM)
{
    PICon(Ref, Pos, ConCommand)
    MotorMixing(ConCommand, RPM)
}
```

Note: __ as input and ‾ as output

ConDyn.c

Figure 5.5 Translation of Local Controller from Simulink to C

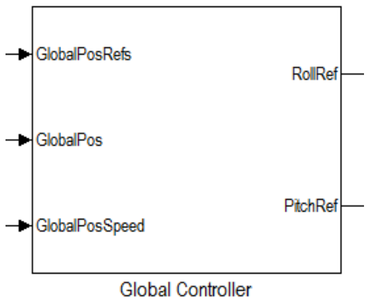
Following the same way we could also translate the “System Dynamics” subsystem and “Global Controller” subsystem by



$\longrightarrow$

```
SysDyn(MotorRPM, X̄, X̄dot)
{
    RPMtoForce(MotorRPM, V̄)
    QuadDyn(X, V, X̄dot)
    Integral(Xdot, X̄)
}
```

SysDyn.c



$\longrightarrow$

```
GloConDyn(RefPos, GloPos, GolPosDot,
Angref)
    {
        //PID Control Algorithm
    }
```

GloConDyn.c

Figure 5.6 Translations of System Dynamics and Global Controller from Simulink to C

The next step is to include the sensor models and all kinds of filters. They can be two different C files that contain various functions as follows. The detailed model or algorithm for each model and filter was introduced in previous chapters, and some of the source code is included in Appendix C.

```
Gyro($\underline{X}$, $\bar{G}$)

Altimeter($\underline{X}$, $\bar{h}$)

Kinect($\underline{X}$, $\overline{GloPos}$)
```

SenDyn.c

```
LR($\underline{h}$, $\overline{hdot}$)    //Linear Regression

HGI($\underline{G}$, $\overline{Ang}$)    //High Pass Gyro Integral

LP($\underline{RawGloPos}$,   $\overline{GloPos}$,  $\overline{GloPosDot}$)
//Low Pass Global Position and Velocity
```

FilDyn.c

Figure 5.7 C Structure of SenDyn.c and FilDyn.c

To combine all these subsystems (C-files) and run a simulation, header files that correspond to each c-file and functions need to be generated. There should also be a header file that contains all the parameters. More importantly, to translate the whole closed loop structure, the "main.c" file needs to come in, which combines everything in a loop that runs in specific sampling rate. It also shall print interested variables to terminal and files such that they could be observed and compared, just as the" scope" and "save to command window" functions in the Simulink. The following C pseudo structure of the "main.c" file put everything together:

```
Include ConDyn.h SynDyn.h GloConDyn.h SenDyn.h FilDyn.h

main(){
    while(Forever){
        If every T_sys{
                SysDyn($\underline{MotorRPM}$, $\bar{X}$, $\overline{Xdot}$)
        }
        If every T_sensor{
                Gyro($\underline{X}$, $\bar{G}$)
                Altimeter($\underline{X}$, $\bar{h}$)
        }
        If every T_local{
                LR($\underline{h}$, $\overline{hdot}$)
                HGI($\underline{G}$, $\overline{Ang}$)
                Pos=Combine(h, Ang)
                PosDot=Combine(hdot, G)
                ConDyn($\underline{Ref}$, $\underline{Pos}$, $\underline{PosDot}$, $\overline{MotorRPM}$)
         }
         If every T_global{   //Call only if global position tracking is active
                Kinect($\underline{X}$, $\overline{RawGloPos}$)
                LP($\underline{RawGloPos}$,  $\overline{GloPos}$, $\overline{GloPosDot}$)
                GloConDyn($\underline{RefPos}$, $\underline{GloPos}$, $\underline{GloPosDot}$, $\overline{AngRef}$)
                Ref=Combine(h_ref, Angref, yaw_ref)
        }
    }
}
```

Figure 5.8 C Structure of main.c

Even though the C/C++ structure is more complicated and less straight forward than Simulink, it moves a big step forward, in the way that the controllers' and filters' sources files (ConDyn.c, GloConDyn.c, Filter.c) can now be directly interfaced with the AR.Drone's sensor-board data acquisition code and motor-board driver code. The advantage of easy interfacing is outstanding when new designs of filters or controllers are to be tested. Besides, it gives a more realistic real-time simulation by including the sensor's models and animation. The results of this model will be compared with experiments data in Section 4.

5.3 Experiments Procedure and Results

A lot of experiments were conducted to test the controllers and filters design. They can be categorized to three sets: local filter selection, local controller testing, global controller and filter testing. Filter and controller gains were first obtained from modeling, and have been extensively tuned to achieve better performance. The experiments results only illustrate and analyze the best tuning results.

5.3.1 Local Filter Angle Estimation

Direct integral, high pass integral, Complementary Filter and Kalman Filter were first evaluated when the drone is at static. The following figure illustrates the pitch angle output of each method. Pitch angle was chosen because it drifts more than roll and yaw, such that the comparisons are more obvious.

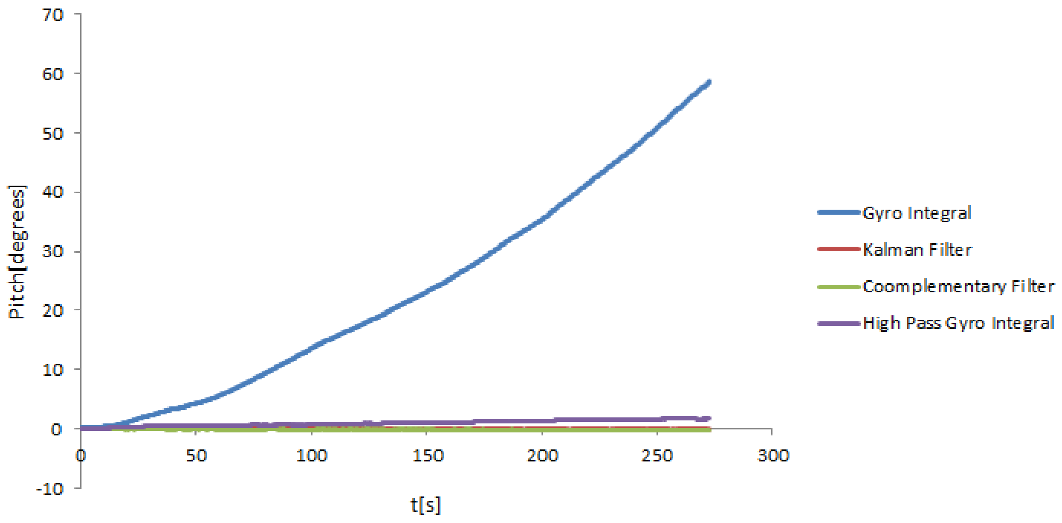


Figure 5.9 Comparisons of Local Filters at Static

From the figure it's clearly seen that high-pass rate gyro has successfully reduced the drift, however, not as well as the Kalman Filter and Complementary Filter. The accelerometer helps corrects the drift so well such that even no drift is seen over time.

To further evaluate each filter's performance the drone was made to execute balancing for the first 4.5 seconds, and follow 15° roll angle reference afterwards. Roll angle was chosen because it has less drift than pitch and yaw, such that the integral gives the best "true angle" estimation in short time run, since the true angle is very difficult to be measured during a free run. Figure below compares the roll angle estimated by each method.

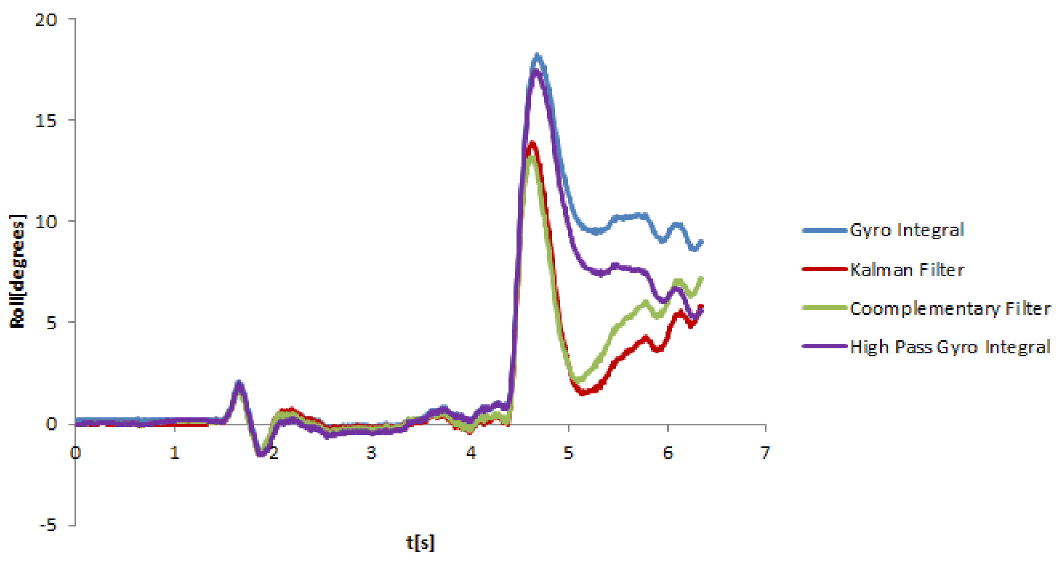


Figure 5.10 Comparison of Local Filters at Roll Reference 15° following

The high pass filter, Kalman Filter and Complementary Filter gave very close estimation of angle during the free run at the first the four seconds, when the direct gyro integral was also close. However, once the drone was subject to transient process, the filters behaved quite differently. While the high pass filter still tracked the gyro integral from 4.5 [s] to 5[s], the Kalman Filter and Complementary Filter tended to give 5° to 8° less estimation of the gyro integral, which could no way be because of the drift in such a short time with so fast response. However, after 5 [s] three filters tended to get close with steady state response while the gyro integral became alone.

During the experiments we did observe a large angle in transient process and an obvious angle reduction while the drone had reached steady-state. The reduction was not expected because of the active controller, but not nonsense since resistance torque generated by air viscosity could be very large at high velocity. Unfortunately there was no other source that we could utilize to measure the true angle, however we believe the high pass gyro reading gave the best angle estimation from 4.5[s] to 5.5[s] compared to the direct integral and the other two filters. The decision was made also by studying the accelerometer data during the period, shown in the following figure.

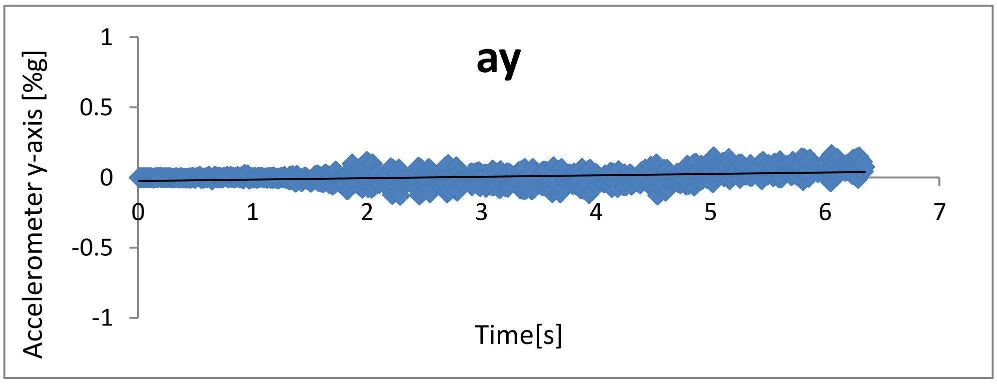


Figure 5.11 (cont. on next page)

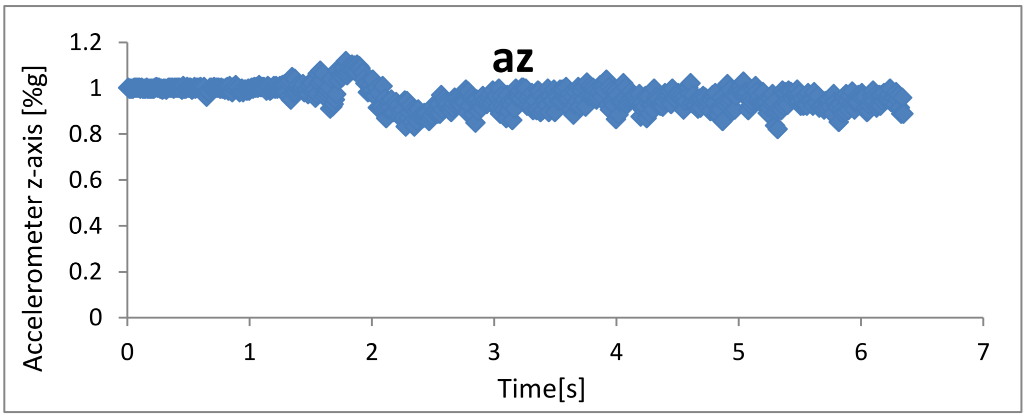


Figure 5.11 Noisy Accelerometer Output

As we can see the accelerometer data was very noisy along the way, even though a_y and a_z showed the trend of going up and down, respectively, after 4.5[s]. Besides, during the transient process from 4.5[s] to 5 [s], neither of the accelerometers showed significant changes. The angle estimated from accelerometer at this period would be very close to 0° as before, while the true angle had a prompt rise from 0° to 17°. This is the reason why the roll angle estimated by Kalman Filter and Complementary Filter had a significant drop from 4.7[s] to 5[s]. However during the steady state, the angles output from Kalman Filter and Complementary Filter had the trend of going up, which matched the trend of the accelerometers.

Because of the room limit of the lab, roll and pitch angle references will change very frequently during global position tracking. Thus, the transient response would dominate steady state, which had led us to use the high-pass gyro integration method to estimate angle.

5.3.2 Local Reference Following

With the high pass filter and linear regression, all four channel's local references following experiments were conducted. The drone was first made to follow several height references. Then after reaching a specific height the drone was made to follow roll, pitch and yaw angle references. The following figures illustrate the experimental setups and results.

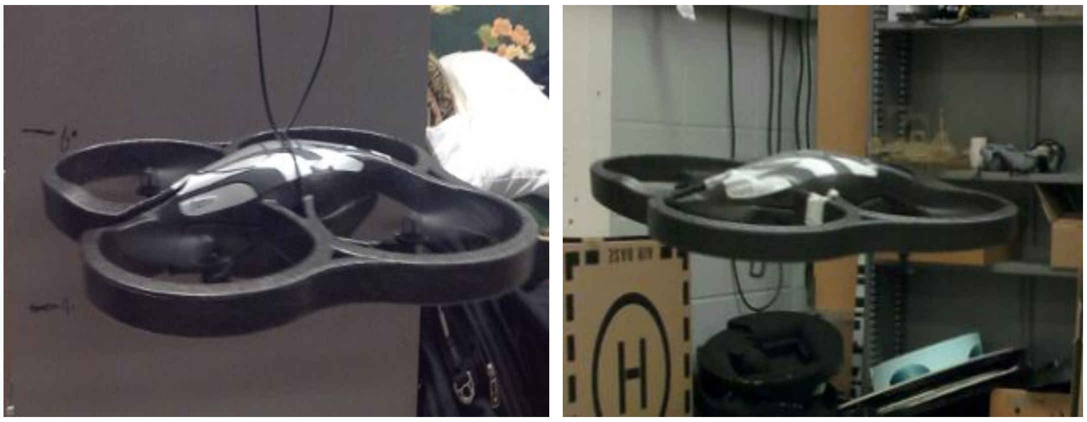
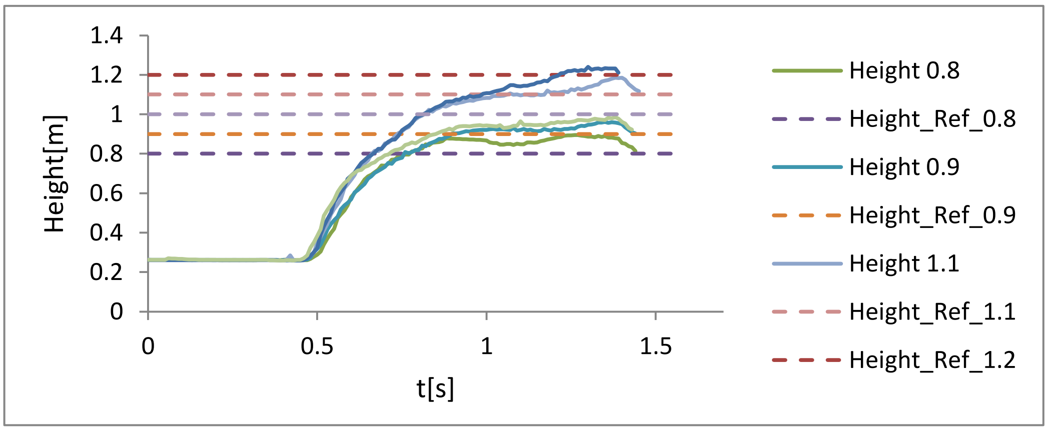
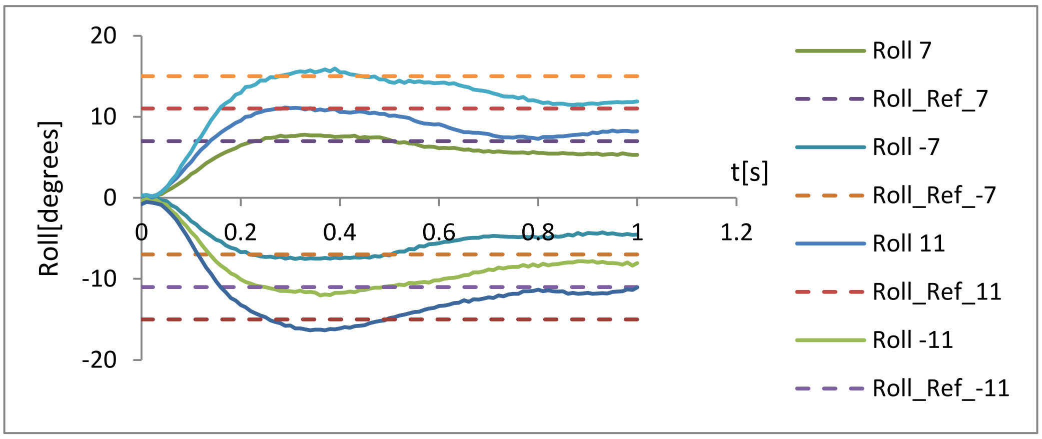


Figure 5.12 (cont. on next page)





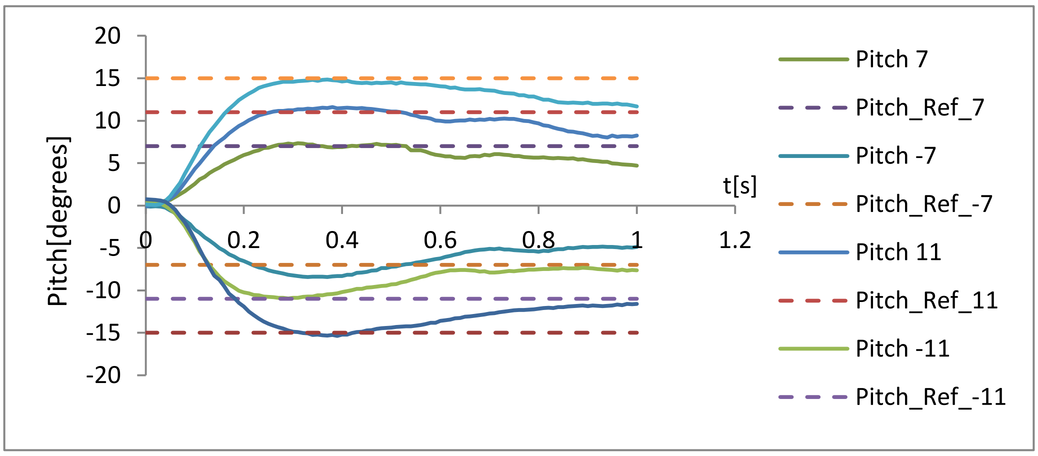


Figure 5.12 (cont. on next page)

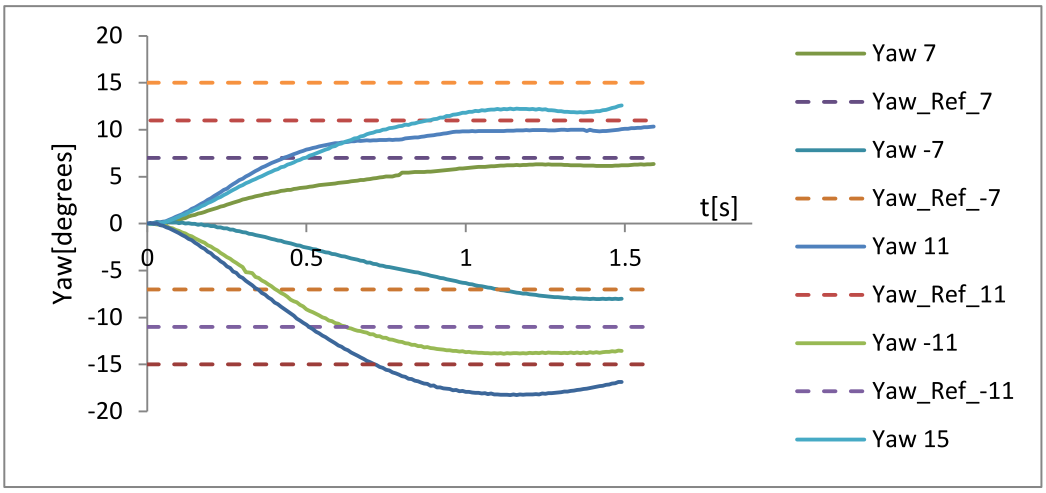


Figure Set 5.12 Experiments Setup and Results of Local Reference Following

These results proved the local controllers and filters to be very successful. The drone followed all references very well in the transient process. The pitch and roll references following did have some angle deficit (around 3° to 4°) at steady state which was not compensated by the controller, due to large air resistance torque at high speed. However, this is not very critical to global position tracking application.

5.3.3 Global Position Tracking

The first stage of global position tracking was from point to point. It was used to evaluate the global controller and compare the designs of global filters. The drone was made to fly to a specific height with all zero angle references until steady state. Then it was made to go from one point to another point, with the same height reference and zero yaw reference. Some of the experimental figures are listed below.

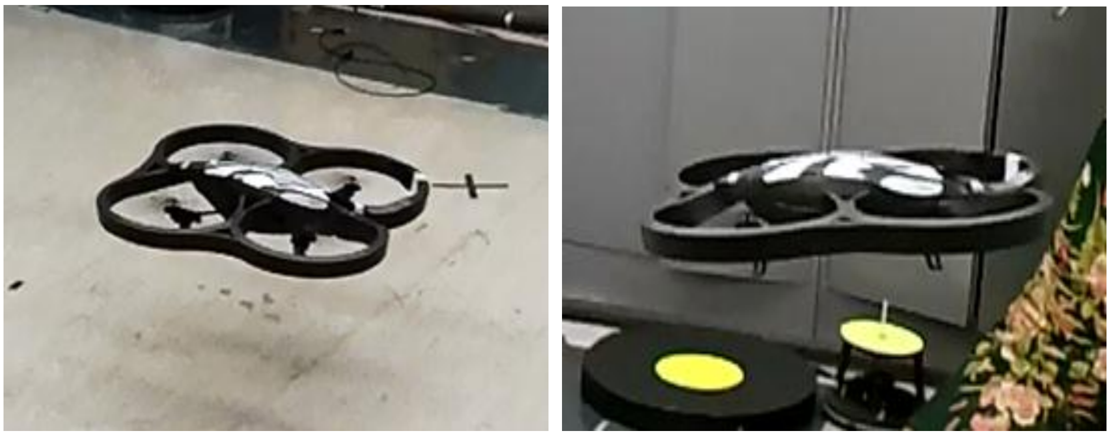


Figure 5.13 Experiment Setup for Point to Point Global Position Tracking

The experiment was conducted by the global low pass filters, however Complementary Filter and Kalman Filter was also running as comparison. The following figure displays the positions feedback from Kinect, the low pass filter, the Complementary Filter and Kalman Filter.

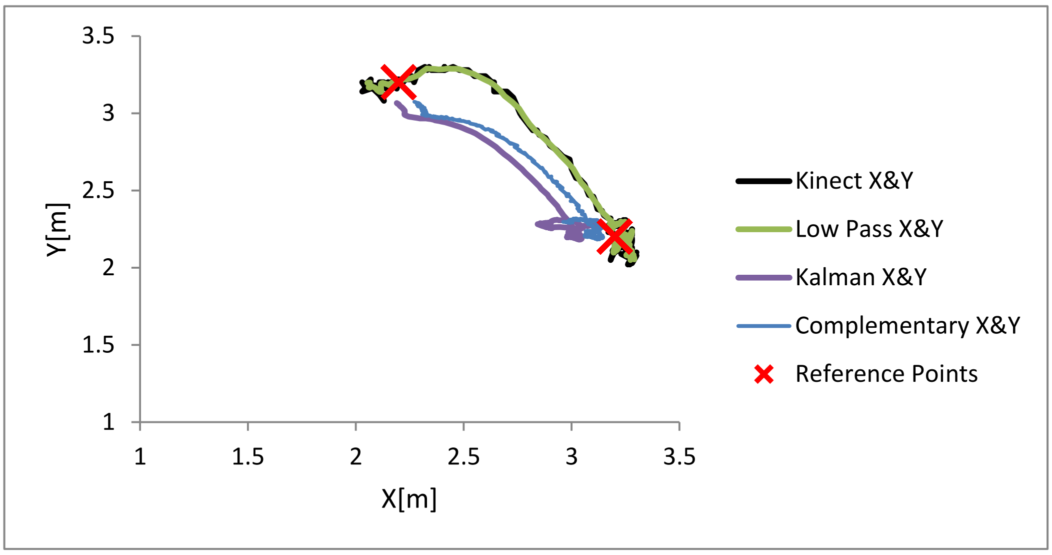
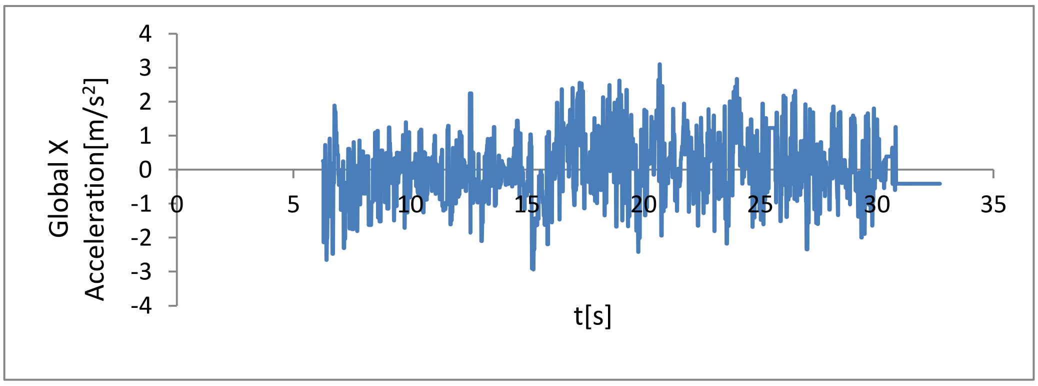


Figure 5.14 Point to Point Global Position Tracking Results

Even though the output positions of Complementary Filter and Kalman Filter were smoother, and showed the trend of position following, they did not correctly present the real location of the drone. This is not unexpected due to the way of calculating acceleration. Recall the way of calculating global X acceleration by

$$a_X = g\theta + a_x$$

The pitch angle estimated from high pass gyro integration already has some error; the combination of it with the noisy accelerometer simply makes the estimation much worse. Take a look at the global X and Y acceleration



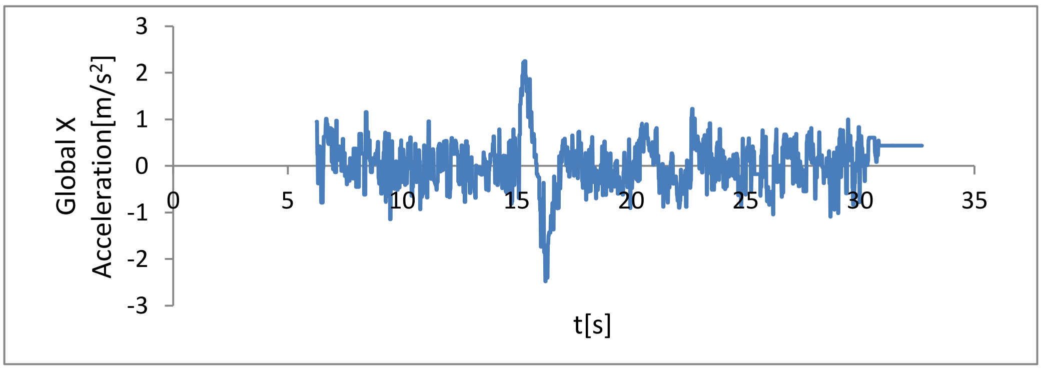


Figure5.15 Noisy Acceleration Estimation

They are simply too noisy to show any trends. Integration and double integration of such a noisy signal will cause huge drift on velocities and positions, which are not possible for correction in a short time. However, if the accelerometer data were not included, the positions predicted by Kalman Filter and Complementary Filter had a lot of overshoot that worsen the performance.

Thus, Kalman Filter and Complementary Filter are abandoned in global position control. However, if the acceleration signal could be estimated more accurately, or if there are any other accurate signals than can be used to estimate position, Kalman Filter and Complementary Filter will generated provide a more accurate and robust signals than the low pass filter.

Once the filter was selected, the drone was made to follow a more complicated geometry. The following figure shows the reference circle and the drone's path. The drone started at point (3.2,2,5) where the reference circle intersects with Kinect feedback.

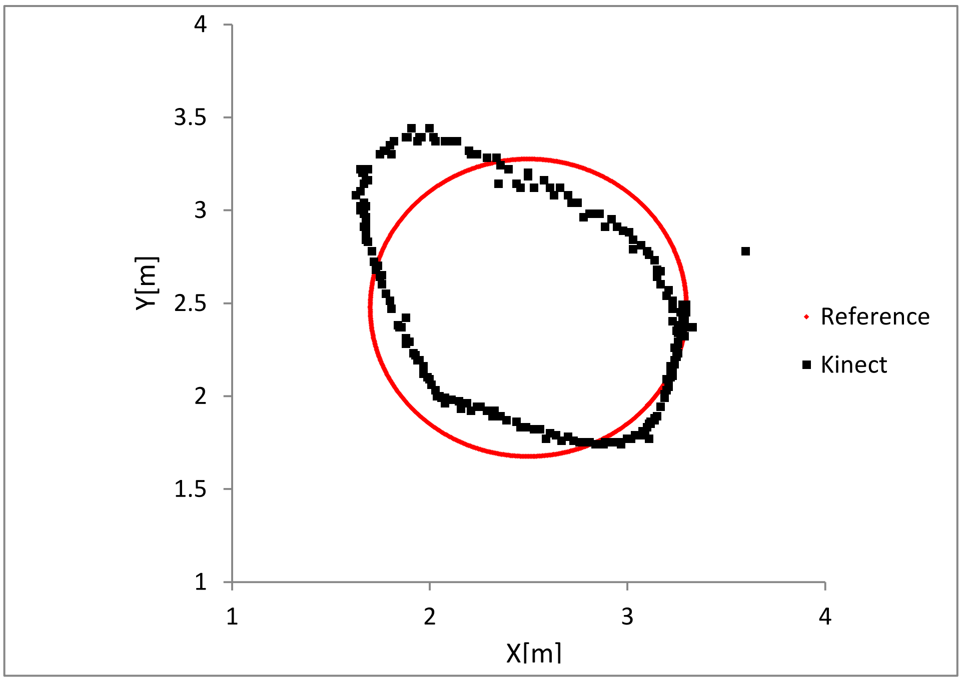


Figure 5.16 Circle Tracking Results

Considering that drone's hull diameter is 0.50[m], and the Kinect often focuses on different parts of the drone, the circle following is precise within 10%.

5.4 Model and Experiments Comparison

The preliminary controllers and filters in the model were updated with the ones tuned from experiments. Then both local and global responses from the simulation were compared to those from experiments.

5.4.1 Local References Following

Figures below illustrate the comparison of simulation and experiments with different local control targets references. The simulation results are very close to those of the experiments, especially in transient process. However, since the model does not consider air resistance or other disturbance in a real fly, simulation tends to be off experiments at steady state.

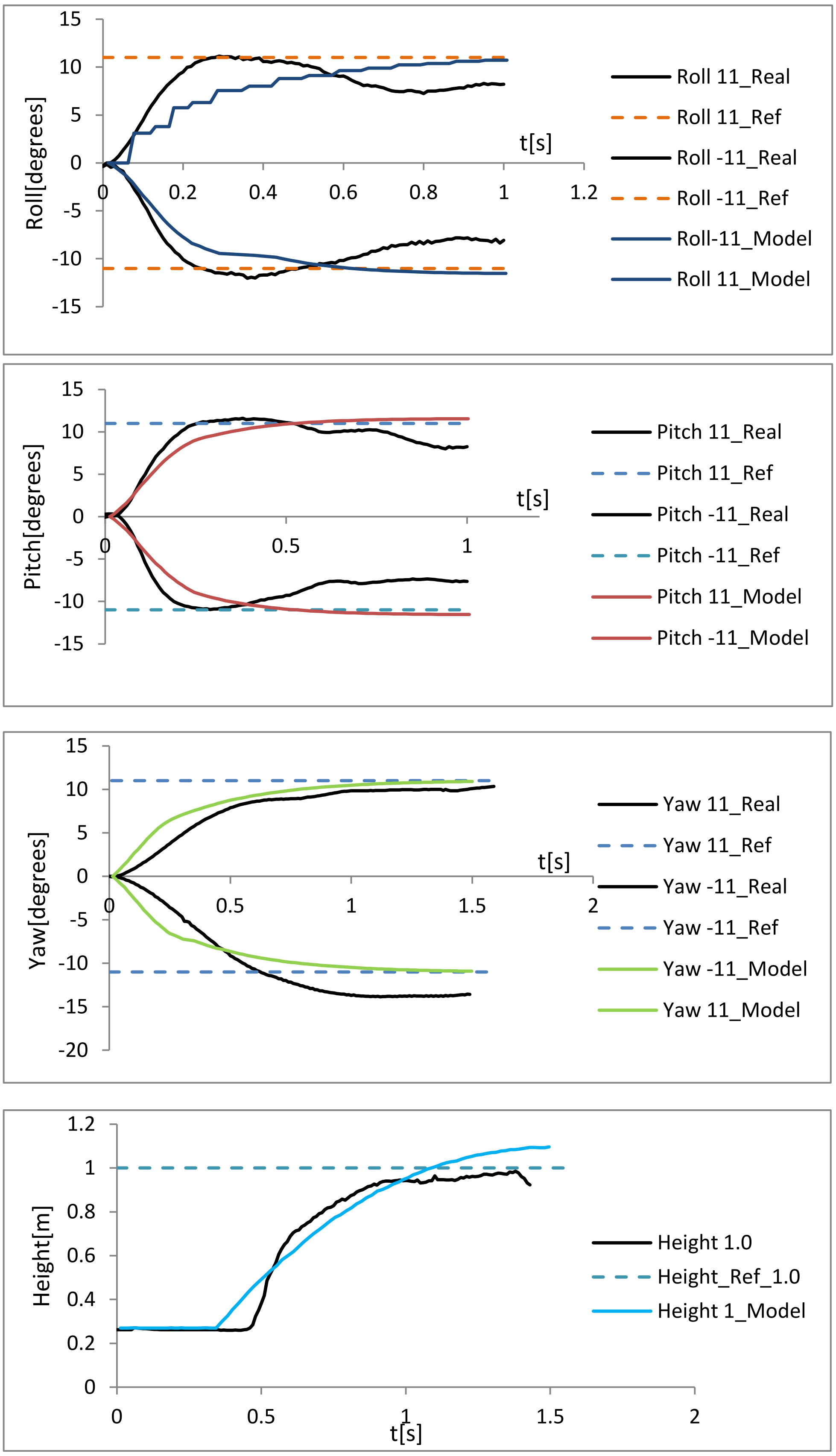


Figure Set 5.17 Comparisons of Simulation and Experiment Results at Local Reference Following

5.4.2 Global Position Tracking

Figure below illustrates the comparison between simulated and real positions from point to point. The simulation is a good estimation of the real system, however not perfect. The real system tends to lag the simulation, which is similar to the local responses where real pitch and roll angles deficits the ones from simulation. One of the possible reasons is that in a real run, air resistance has prevented the drone going as fast as the model, where aerodynamic forces are not simulated. Another possible reason is the Kinect model is oversimplified, such that it doesn't truly represent the Kinect's reading noise. The real Kinect tends to focus on different parts of the drone's body, while the model only assumes it has a fixed bias with some noise.

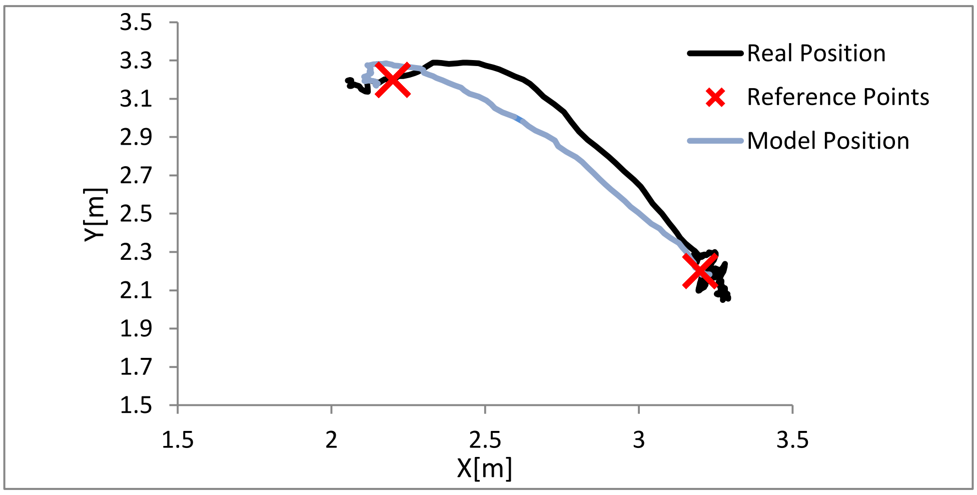


Fig 5.18 Comparison of Simulation and Experiments Results at Point to Point

The figure below compares circle tracking between Simulation and Experiments. The simulation is again a very good estimation of real system.

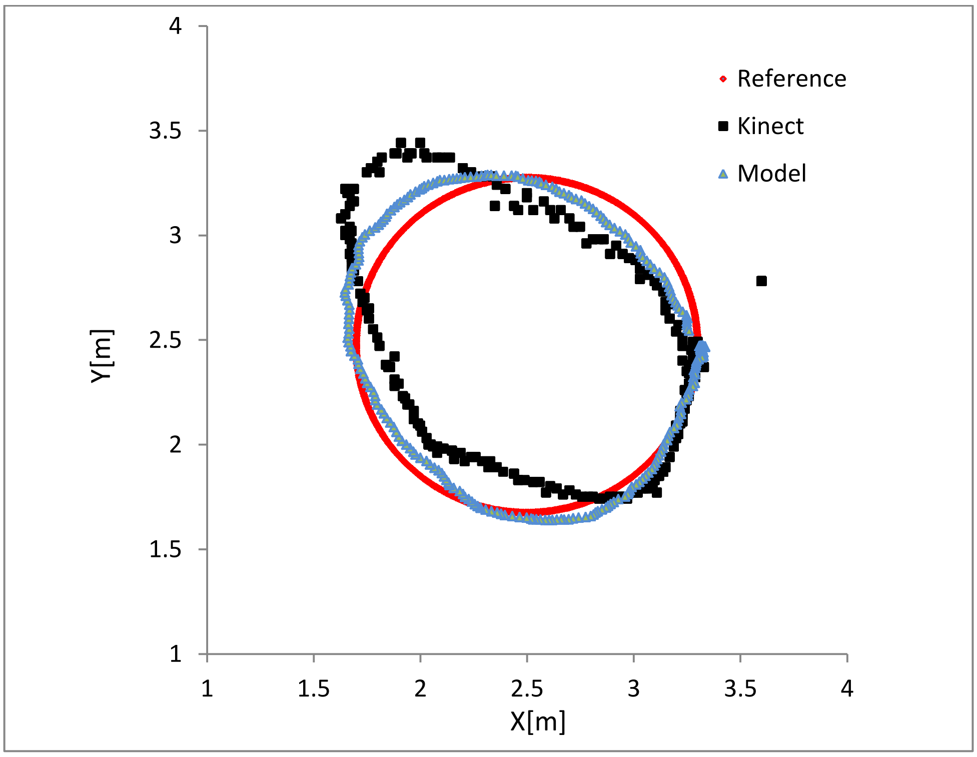


Figure 5.19 Comparison of Simulation and Experiments Results at Circle Tracking

Chapter 6 Animation

Animation visualizes movement by rapid displaying a sequence of the motion images. In our application it produces an intuitive virtual effect on the drone's dynamics, by printing the AR.Drone's 3D picture according to the translational and rotational vectors obtained from simulation. The animation could be used to compare with the experiment videos directly to evaluate the modeling.

6.1 Animation Tool

Irrlicht Engine is an open source real time 3D engine with high rendering performance written in C++. It is selected as the animation tool because it is free, it can interface with our model easily since they are both written in C/C++, and it has a lot of example tutorials within its package. Besides, it is very widely used with a lot of available examples and supports online. Its good rendering effect can be seen at the following figures

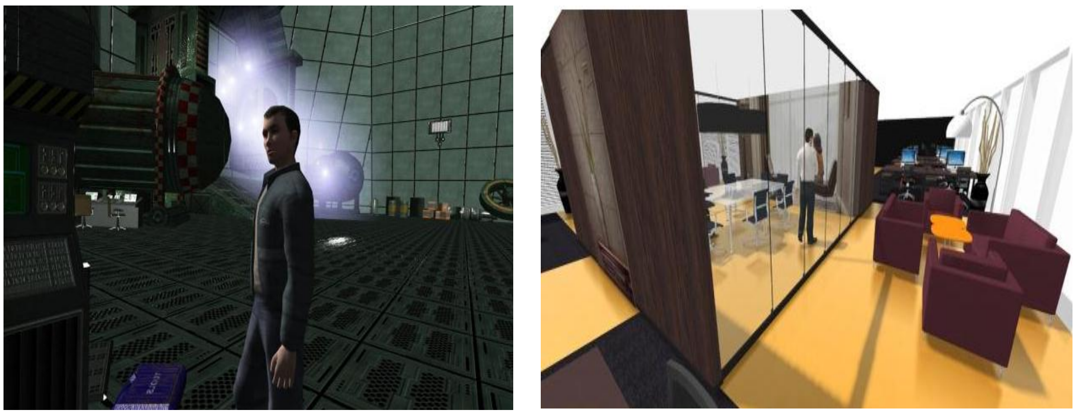


Figure 6.1 Rendering Effect of Irrlicht Engine

Here lists some of its features that were very helpful while we developed our animation. Please refer to its official website [21] for more information.

- Real-time 3D rendering using Direct3D and OpenGL
- Runs on Windows, Linux, OSX and others
- Well documented API with lots of examples and tutorials
- Direct import of mesh files such as Maya(.obj)
- Direct import of textures such as Windows Bitmap(.bmp) and JPEG(.jpg)
- Powerful library with a lot of useful functions

6.2 Animation Procedure

The nominal steps to create an animation by Irrlicht Engine is to initialize an Irrlicht environment, upload the personalized mesh and texture files, and execute a run that moves the mesh pictures. To interface with Drone's modeling, the run of Engine shall be in parallel with run of simulation, such that the movement of the mesh pictures is synchronized with simulated results.

6.2.1 Environment Initialization

Following the nominal procedures of initialization of Irrlicht Engine, we put everything in a function called “IrrInit()” which outputs the initialized Irrlicht device. The pseudo code is concluded as below.

```
(Note: X denotes a variable named X)
Create an Irrlicht device with specification
Create driver for the device, select driver type (Open GL, DirectX, etc.)
Create scene manager scene for the device

Create camera for scene from scene manager, specify its position
Create meshes for scene and upload personalized meshes by scene manager, specify their location
Create texture for meshes and use driver to get corresponding textures
Create light for scene from scene manager, specify its location
```

IrrInit($\overline{device}$)

6.2.2 Personalized mesh and texture files

The animation gives the best visualization effect with the AR.Drone’s body. The first stage to create such a body is to draw the AR.Drone in modern CAD software. Thanks to Weijia Luo, who created a very realistic AR.Drone assembly in ProE. Views of the CAD model are shown in the figure below.

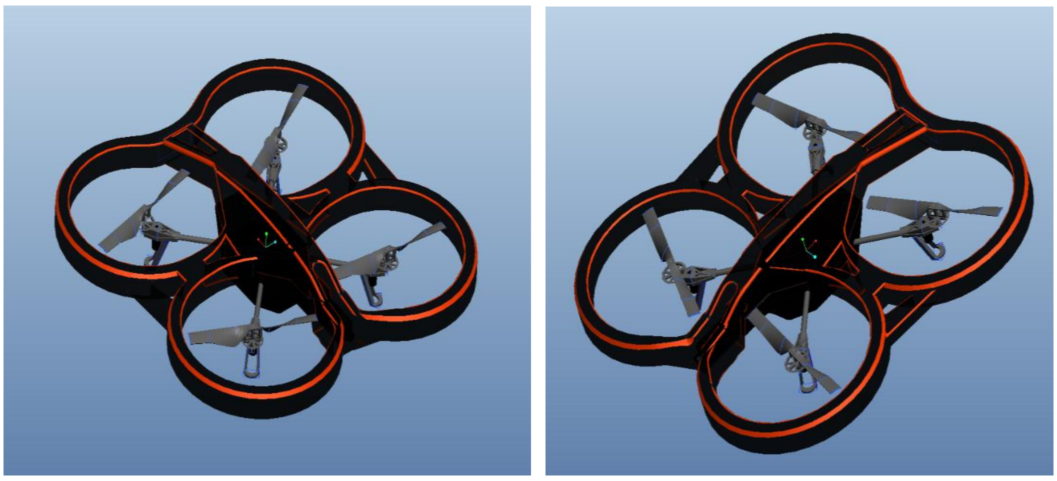


Figure 6.2 Personalized CAD Drawings of the AR.Drone

The CAD model was then outputted as a mesh file by ProE along with its textures. Autodesk Maya was then used to modify the textures and colors such that they are close to the real AR.Drone. Following the same technique we could also personalize the backgrounds to be similar to our lab. The mesh files and textures are then uploaded to Irrlicht Engine by the scene manager. The following figures are from the Irrlicht Engine where the AR.Drone and the lab’s floor have been updated.

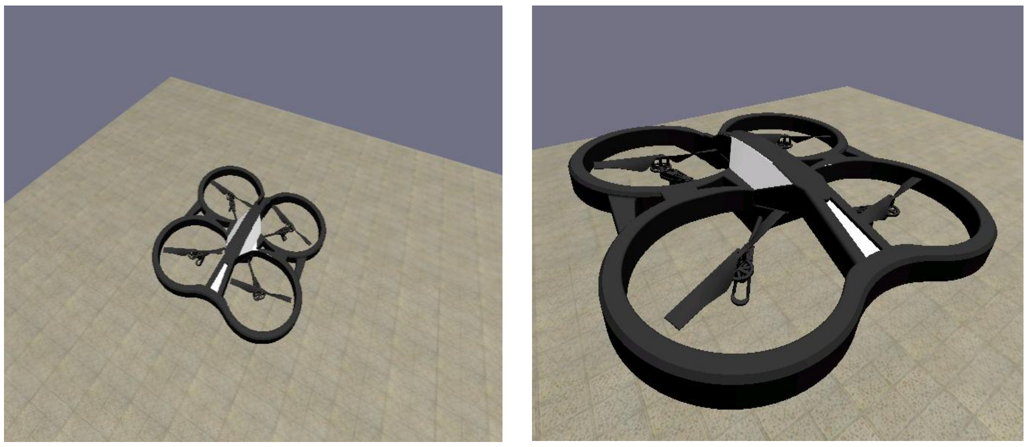


Figure 6.3 Personalized AR.Drone Scene in Irrlicht Engine

6.2.3 Movement of the mesh pictures

The movements are achieved by frequently drawing the scene and updating the mesh's positions. One cycle of the drawing and updating could be executed by function IrrPrint() with the following pseudo code.

```
Set positions of the drone's mesh according to position vector
Set rotation of the drone's mesh according to the rotational vector
Begin scene by the driver
Use scene manager to draw everything
End scene by the driver
```

IrrPrint(Pos)

6.2.4 Interface with model

A C++ file, Irr.cpp, was created to combine the two functions. Then by linking to Irr.cpp, the functions could be used by the main.cpp file to interface with the dynamics. Note all the other C files have to be formatted to C++ to be compiled. The following pseudo code is added to main.cpp.

```
Include Irr.h
main{
    IrrInit($\overline{device}$)
    While(FOREVER){
        …
        if device is running and every T_ani{ //Execute this every animation period
            Pos=Select(X)
            IrrPrint(Pos)
        }
        …
    }
}
```

main.cpp

Chapter 7 Conclusion

This paper mainly contains the kinemics and dynamics, modeling, identification, controller and filter design, simulation, experiments and animation on an electric-motor drive small-scale quadrotor --- AR.Drone. It also suggests a model, conducts identification of a motion sensor camera Kinect. The camera's model along with AR.Drone's on board sensor models improved the fidelity of simulation.

While the kinemics, dynamics and modeling of AR.Drone is similar to that of other quadrotors, the identification utilized model CAD software to identify the moment of inertia matrix of the drone, and applied engineering reverse methodology to determine the aerodynamic thrust and drag. Modeling of AR.Drone's on board sensors such as the rate gyro and altimeter was also very detailed with thorough identifications.

While the controllers' design follows common PID algorithm, this paper discusses and compares several sophisticated designs of the filters, which is very important to conduct a successful and accurate run during experiments, especially with those low-cost and low-precision sensors and complicated aerodynamics during fly.

The simulation was conducted in two stages in two different simulators. A first stage MATLAB simulation was designed with perfect sensors and no filters, to only test the controllers and obtain a preliminary set of gains. The second stage C/C++ simulation also includes the sensor models and filter designs to enhance the fidelity. Animation is also included to visualize the model. The simulation-tuned controllers and filters' gains were used as a preliminary start for experiments.

Numbers of experiments have been conducted to test the controllers and compare the filters. The best filters were selected based on the flying performance. The controller gains were further tuned to achieve the best flying. Then experiment results were compared to those form simulation with the same filters and controllers, to evaluate the modeling and identification. The simulation correlated well with experiments at both local references following and global position tracking. The model and identification was proved to be successful.

A lot of improvement could have been made to the modeling, control design and experiments. The model could have been more accurately identified, including the air resistance force and torque. The global position tracking controller could have directly commanded the motors, instead of a outer loop to command angle, to improve the global tracking performance. Local Kalman Filter and Complementary Filter could have been used not based on accelerometers readings, but on the net angular acceleration from input torque and air resistance torque. The acceleration estimation method of global Kalman Filter and Complementary Filter could have been improved by combining with air resistance force instead of accelerometer reading. The local angles and global positions could have been exactly measured by precise motion camera sensors such as the Vicon. These ideas either have been tried but failed due to inaccurate identification, or have not been tried due to time constraint. They are listed as suggested future work.

References

[1] Haulman, D. L.,2003: *U.S. Unmanned Aerial Vehicles In Combat*, 1991-2003, Air Force Historical Research Agency

[2] Huang, H.,Hoffmann, G. M., Waslander, S. L., Tomlin, C. J., 2009: *Aerodynamics and Control of Autonomous Quadrotor Helicopters in Aggressive Maneuvering*, Stanford University

[3] Wikipedia, *AR.Drone,* http://en.wikipedia.org/wiki/Parrot_AR.Drone

[4] [5] Beard, R. W.,2008: *Quadrotor Dynamics and Control.* Brigham Young University. pp 1-10

[6] Bresciani, T., 2008: *Modeling, Identification and Control of a Quadrotor Helicopter.* Lund University. pp.132-134

[7] AR.Drone Official Website, *motor description,* http://ardrone.parrotshopping.com/us/p_ardrone_product.aspx?i=199962

[8] Flenniken, W. S.,2005: *Modeling Inertial Measurement Units And Analyzing The Effect Of Their Errors In Navigation Applications.* Auburn University, Alabama. pp.7-8

[9][10] Woodman, O. J.,2007: *An introduction to inertial navigation*. University of Cambridge. pp.10-12

[11] Gu, A. and Zakhor A.,2007: *Optical Proximity Correction with Linear Regression,* University of California at Berkeley. pp.3-4

[12]Perquin H., 2001: *AR.Drone Attitude Estimate Driver,* GNI General Public License, http://blog.perquin.com

[13] Scott,D.M., 1994: *A Simplified Methods for the Bilinear s-z Transformation*, Industrial and Management Systems Engineering Faculty Publications, Paper 73.

[14] Euston, M., Coote, P. & Mahony, R., etc, 2008: *A Complementary Filter for Attitude Estimation of a Fixed-Wing UAV.* Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference. pp. 340-345.

[15] Colton, S.,2007: *The Balance Filter.* Massachusetts Institute of Technology, Boston.

[16] Digana,T.,2004: *Kalman Filtering in Multi-sensor Fusion,* Helsinki University of Technology

[17] Welch, G. & Bishop, G.,2006: *An Introduction to the Kalman Filter.* University of North Carolina at Chapel Hill. NC. pp. 2-6

[18] Kalman, R. E.,1960: *A New Approach to Linear Filtering and Prediction Problems.* Transaction of the ASME—Journal of Basic Engineering.

[19] Pycke,T.,2007, *Kalman Filtering of IMU data,* http://tom.pycke.be/mav/71/kalman-filtering-of-imu-data

[20] Higgins, W. T., 1975: A Comparison of Complementary and Kalman Filtering. Trans. Aerospace and Electronic Systems, IEEE, Vol. AES-li, No. 3. pp. 321-325

[21] Irrlicht Engine, *Features*, http://irrlicht.sourceforge.net/features/

Appendix A Rotational and Translational Matrix

A.1 Rotational Matrix

The rotational matrix that maps a vector from earth frame to body-fixed fame is a multiplication of three basic rotation matrices.

1. *The yaw rotation about the Z_E axis transforms the earth frame to Quadrotor 1 frame by*

$$R_E^1 = \begin{bmatrix} cos\psi & -sin\psi & 0 \\ sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.1}$$

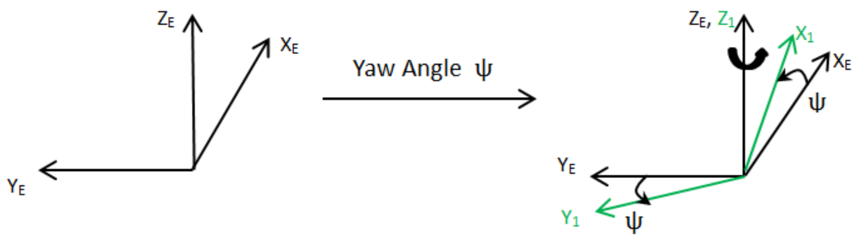


Figure A.1

2. *The pitch rotation about the Y_2 axis transforms the Quadrotor 1 frame to Quadrotor 2 frame by*

$$R_1^2 = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix} \tag{A.2}$$

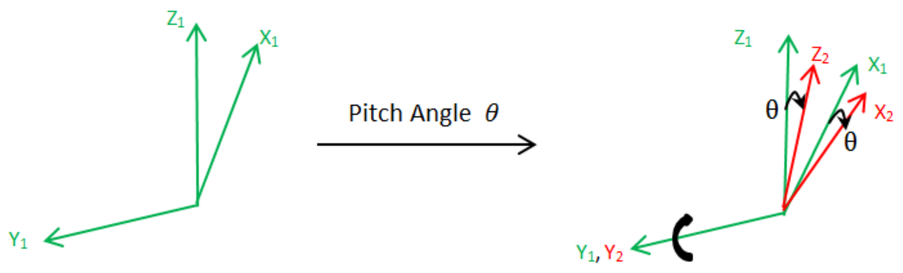


Figure A.2

3. *The roll rotation about the X_1 axis transforms the Quadrotor 2 frame to fixed-body frame by*

$$R_2^B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & -sin\phi \\ 0 & sin\phi & cos\phi \end{bmatrix} \tag{A.3}$$

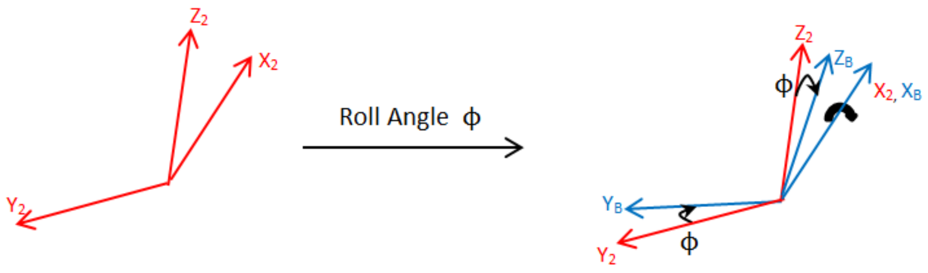


Figure A.3

Then the rotational matrix R_θ is calculated by

$$R_\theta = R_2^B \cdot R_1^2 \cdot R_E^1 = \begin{bmatrix} cos\psi cos\theta & cos\psi sin\theta sin\phi - sin\psi cos\phi & cos\psi sin\theta cos\phi + sin\psi sin\phi \\ sin\psi cos\theta & sin\psi sin\theta sin\phi + cos\psi cos\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi \\ -sin\theta & cos\theta sin\phi & cos\theta cos\phi \end{bmatrix}$$

(A.4)

A.2 Translational Matrix

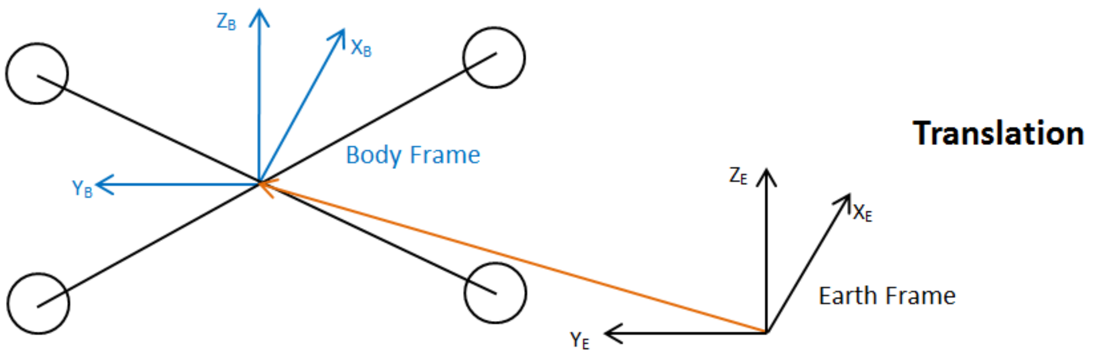


Figure A.4

The translational matrix can be found by resolving earth frame angular velocity to body-fixed frame angular velocity

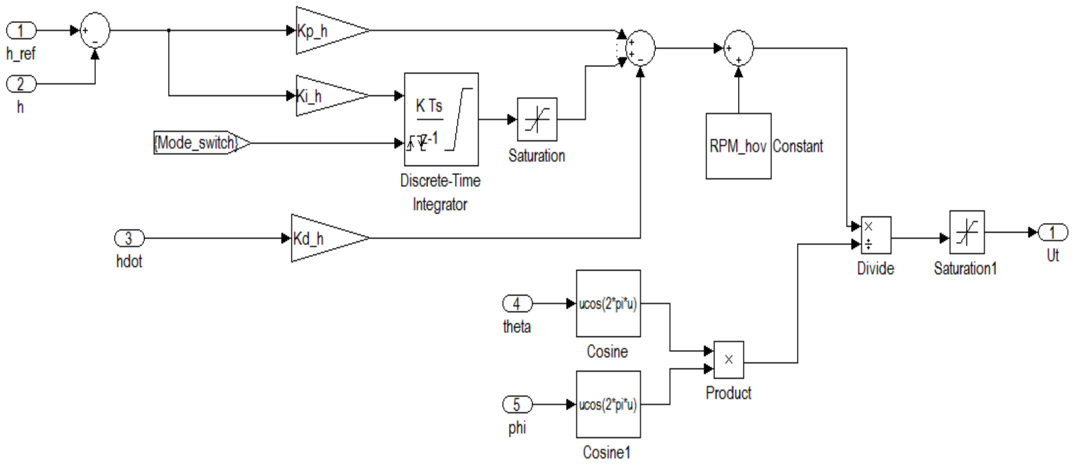
$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + R_2^B \cdot \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + R_2^B \cdot R_1^2 \cdot \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \quad (A.5)$$

$$= \begin{bmatrix} 1 & 0 & -sin\theta \\ 0 & cos\phi & sin\phi cos\theta \\ 0 & -sin\phi & cos\phi cos\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = T_\theta^{-1} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

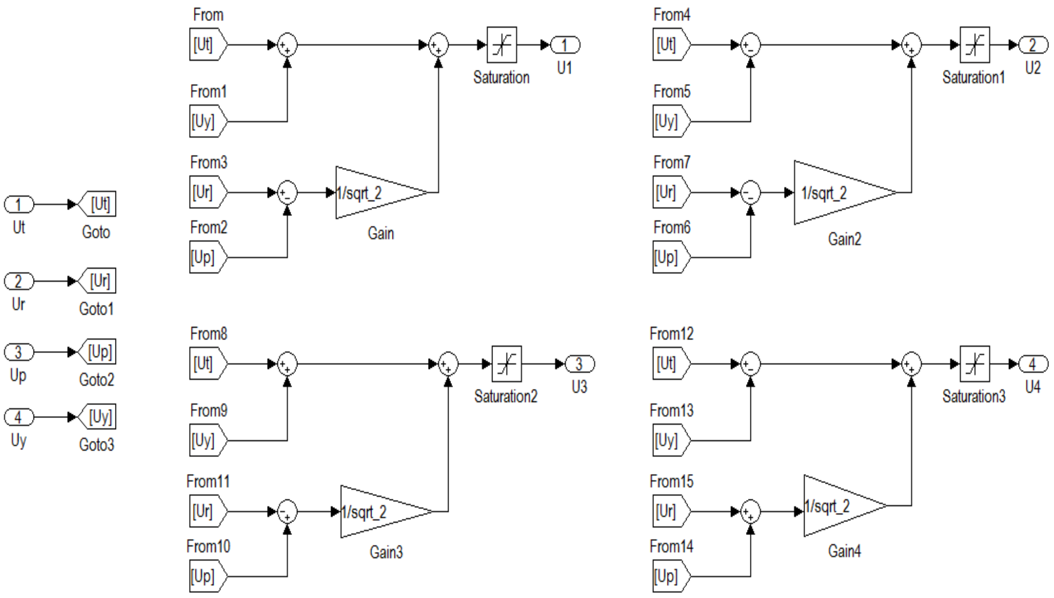
Inverting we have

$$\dot{\theta}^E = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (A.6)$$

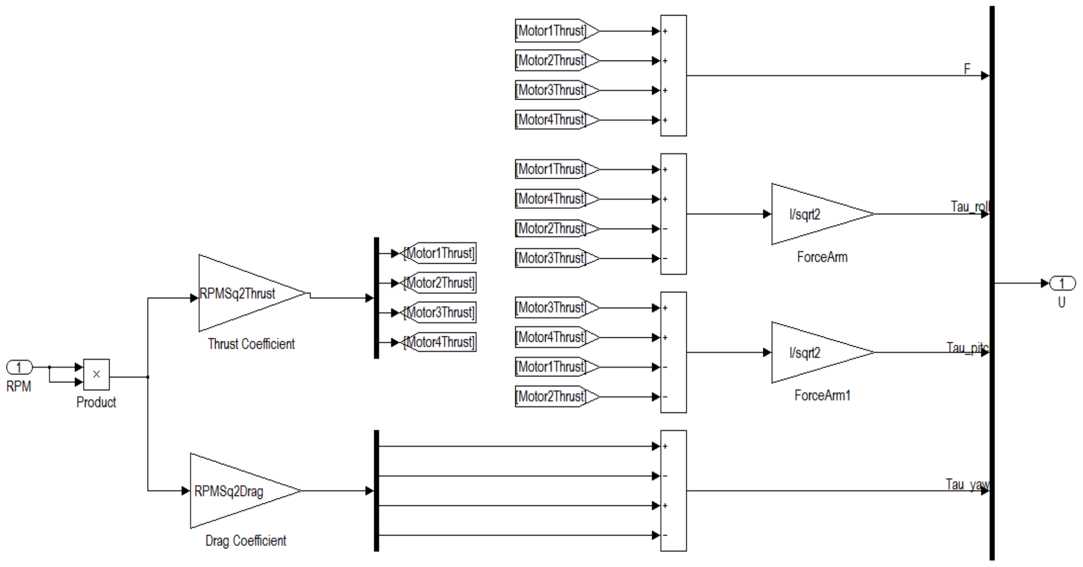
Appendix B   MATLAB Block Details



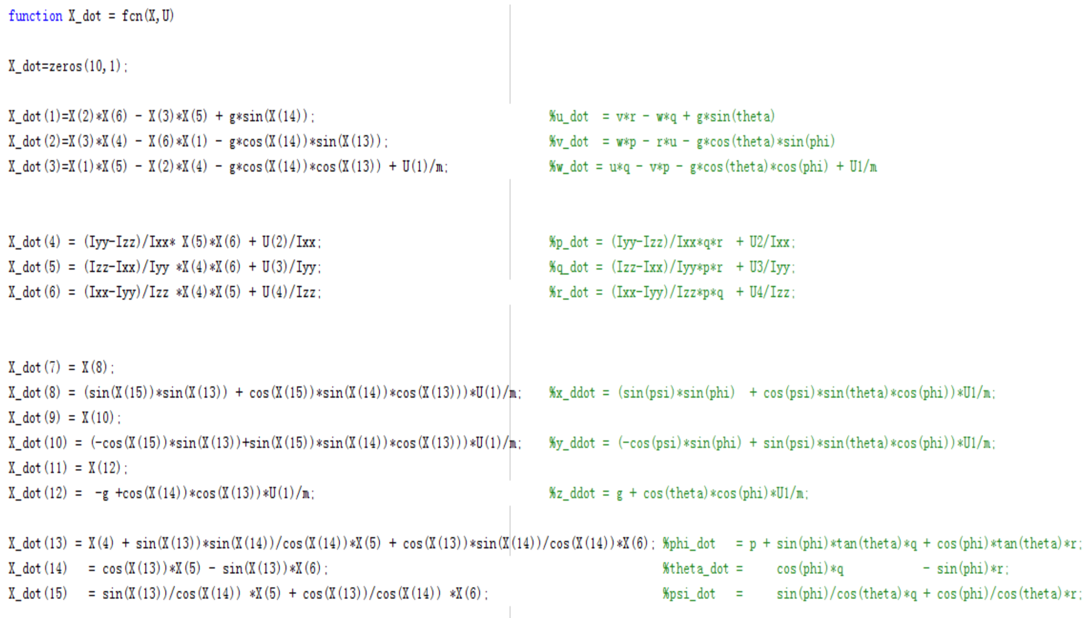
Detail Block 1. Throttle PID Control



Detail Block 2. Motor Mixing



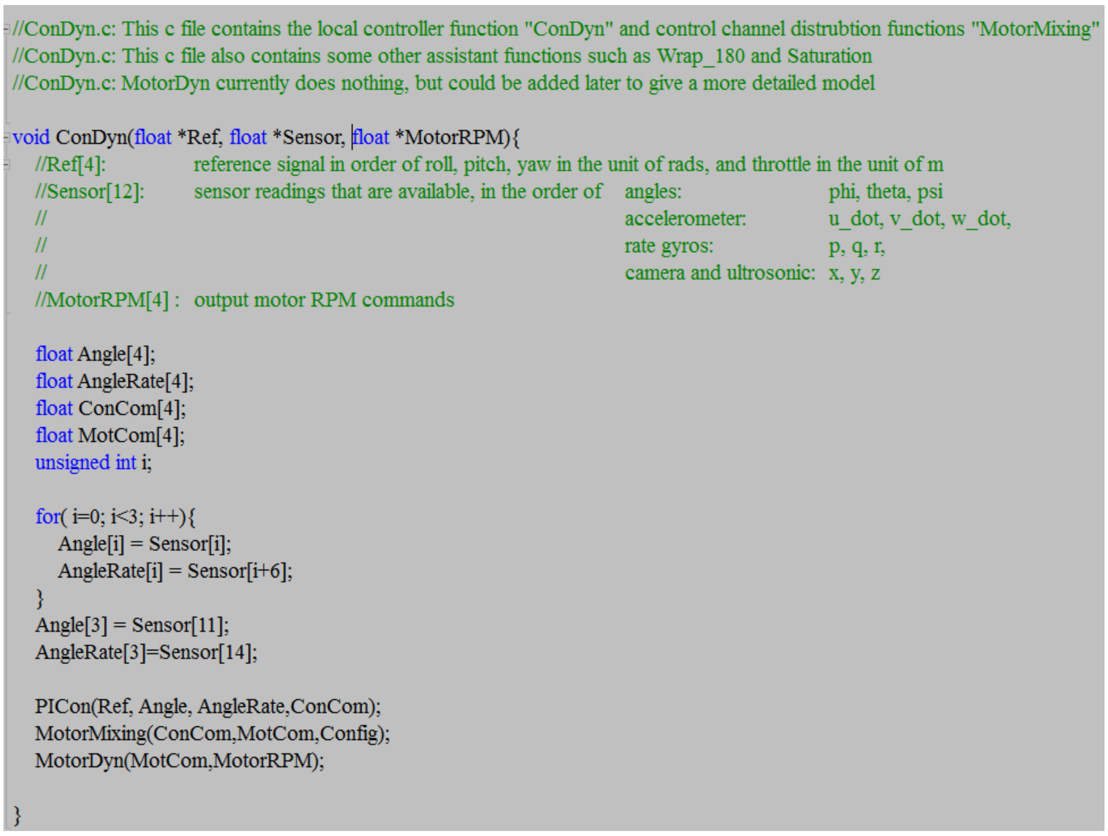
Detail Block 3. Force and Torque Calculation



```
function X_dot = fcn(X,U)

X_dot=zeros(10,1);

X_dot(1)=X(2)*X(6) - X(3)*X(5) + g*sin(X(14));                              %u_dot  = v*r - w*q + g*sin(theta)
X_dot(2)=X(3)*X(4) - X(6)*X(1) - g*cos(X(14))*sin(X(13));                   %v_dot  = w*p - r*u - g*cos(theta)*sin(phi)
X_dot(3)=X(1)*X(5) - X(2)*X(4) - g*cos(X(14))*cos(X(13)) + U(1)/m;          %w_dot = u*q - v*p - g*cos(theta)*cos(phi) + U1/m


X_dot(4) = (Iyy-Izz)/Ixx* X(5)*X(6) + U(2)/Ixx;                             %p_dot = (Iyy-Izz)/Ixx*q*r  + U2/Ixx;
X_dot(5) = (Izz-Ixx)/Iyy *X(4)*X(6) + U(3)/Iyy;                             %q_dot = (Izz-Ixx)/Iyy*p*r  + U3/Iyy;
X_dot(6) = (Ixx-Iyy)/Izz *X(4)*X(5) + U(4)/Izz;                             %r_dot = (Ixx-Iyy)/Izz*p*q  + U4/Izz;


X_dot(7) = X(8);
X_dot(8) = (sin(X(15))*sin(X(13)) + cos(X(15))*sin(X(14))*cos(X(13)))*U(1)/m;   %x_ddot = (sin(psi)*sin(phi)  + cos(psi)*sin(theta)*cos(phi))*U1/m;
X_dot(9) = X(10);
X_dot(10) = (-cos(X(15))*sin(X(13))+sin(X(15))*sin(X(14))*cos(X(13)))*U(1)/m;   %y_ddot = (-cos(psi)*sin(phi) + sin(psi)*sin(theta)*cos(phi))*U1/m;
X_dot(11) = X(12);
X_dot(12) =  -g +cos(X(14))*cos(X(13))*U(1)/m;                                 %z_ddot = g + cos(theta)*cos(phi)*U1/m;

X_dot(13) = X(4) + sin(X(13))*sin(X(14))/cos(X(14))*X(5) + cos(X(13))*sin(X(14))/cos(X(14))*X(6); %phi_dot   = p + sin(phi)*tan(theta)*q + cos(phi)*tan(theta)*r;
X_dot(14)   = cos(X(13))*X(5) - sin(X(13))*X(6);                                                 %theta_dot =     cos(phi)*q             - sin(phi)*r;
X_dot(15)   = sin(X(13))/cos(X(14)) *X(5) + cos(X(13))/cos(X(14)) *X(6);                         %psi_dot   =     sin(phi)/cos(theta)*q + cos(phi)/cos(theta)*r;
```

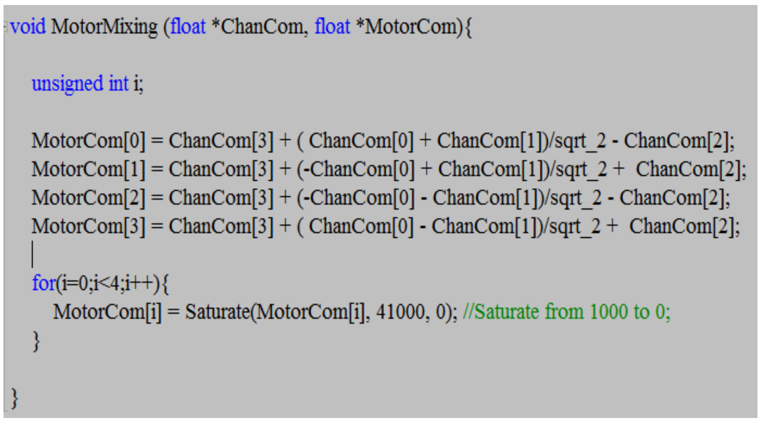
Detail Block 4. Dynamics embedded MATLAB function

Appendix C   Example Source Code



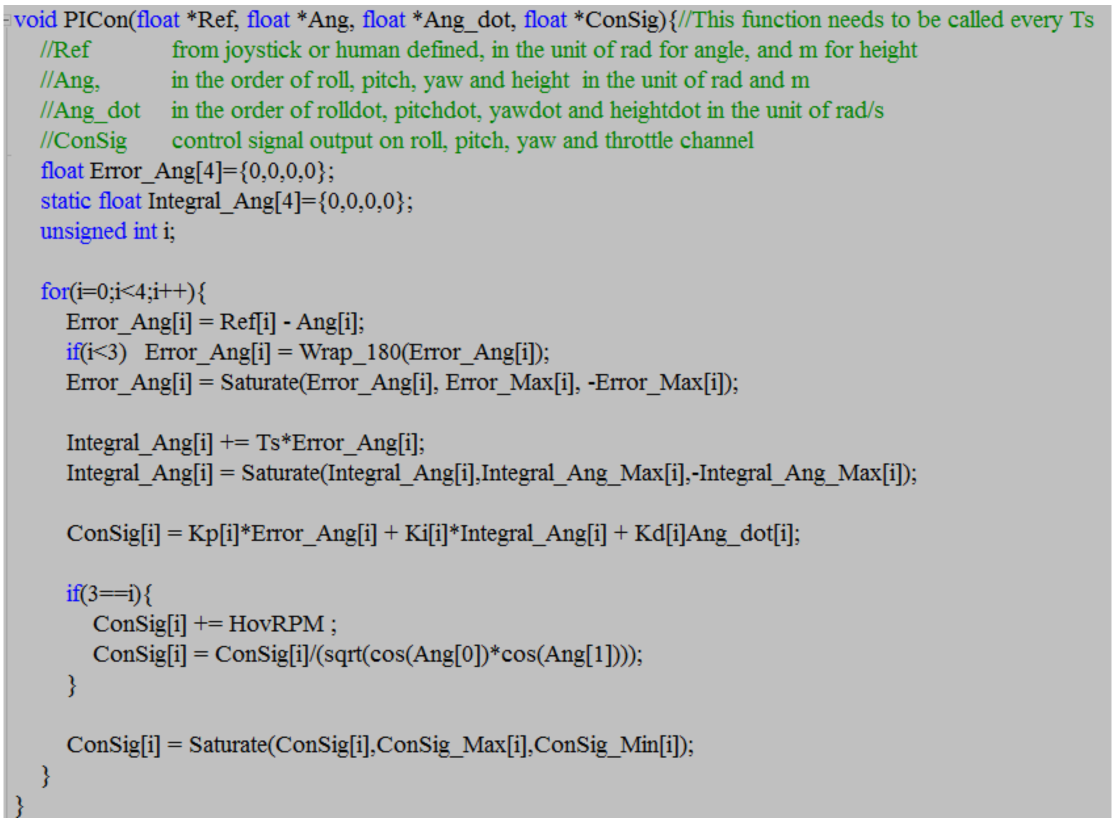
```c
//ConDyn.c: This c file contains the local controller function "ConDyn" and control channel distrubtion functions "MotorMixing"
//ConDyn.c: This c file also contains some other assistant functions such as Wrap_180 and Saturation
//ConDyn.c: MotorDyn currently does nothing, but could be added later to give a more detailed model

void ConDyn(float *Ref, float *Sensor, float *MotorRPM){
    //Ref[4]:           reference signal in order of roll, pitch, yaw in the unit of rads, and throttle in the unit of m
    //Sensor[12]:       sensor readings that are available, in the order of   angles:                      phi, theta, psi
    //                                                                        accelerometer:               u_dot, v_dot, w_dot,
    //                                                                        rate gyros:                  p, q, r,
    //                                                                        camera and ultrosonic:       x, y, z
    //MotorRPM[4] :  output motor RPM commands

    float Angle[4];
    float AngleRate[4];
    float ConCom[4];
    float MotCom[4];
    unsigned int i;

    for( i=0; i<3; i++){
        Angle[i] = Sensor[i];
        AngleRate[i] = Sensor[i+6];
    }
    Angle[3] = Sensor[11];
    AngleRate[3]=Sensor[14];

    PICon(Ref, Angle, AngleRate,ConCom);
    MotorMixing(ConCom,MotCom,Config);
    MotorDyn(MotCom,MotorRPM);

}
```

Detail Source Code 1. High level function ConDyn() of ConDyn.c



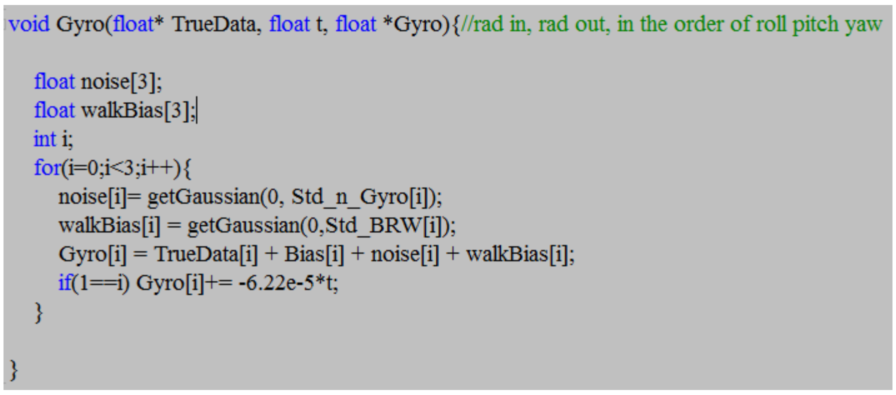
```c
void MotorMixing (float *ChanCom, float *MotorCom){

    unsigned int i;

    MotorCom[0] = ChanCom[3] + ( ChanCom[0] + ChanCom[1])/sqrt_2 - ChanCom[2];
    MotorCom[1] = ChanCom[3] + (-ChanCom[0] + ChanCom[1])/sqrt_2 +  ChanCom[2];
    MotorCom[2] = ChanCom[3] + (-ChanCom[0] - ChanCom[1])/sqrt_2 - ChanCom[2];
    MotorCom[3] = ChanCom[3] + ( ChanCom[0] - ChanCom[1])/sqrt_2 +  ChanCom[2];

    for(i=0;i<4;i++){
        MotorCom[i] = Saturate(MotorCom[i], 41000, 0); //Saturate from 1000 to 0;
    }

}
```

Detail Source Code 2. Function MotorMixing() in ConDyn.c



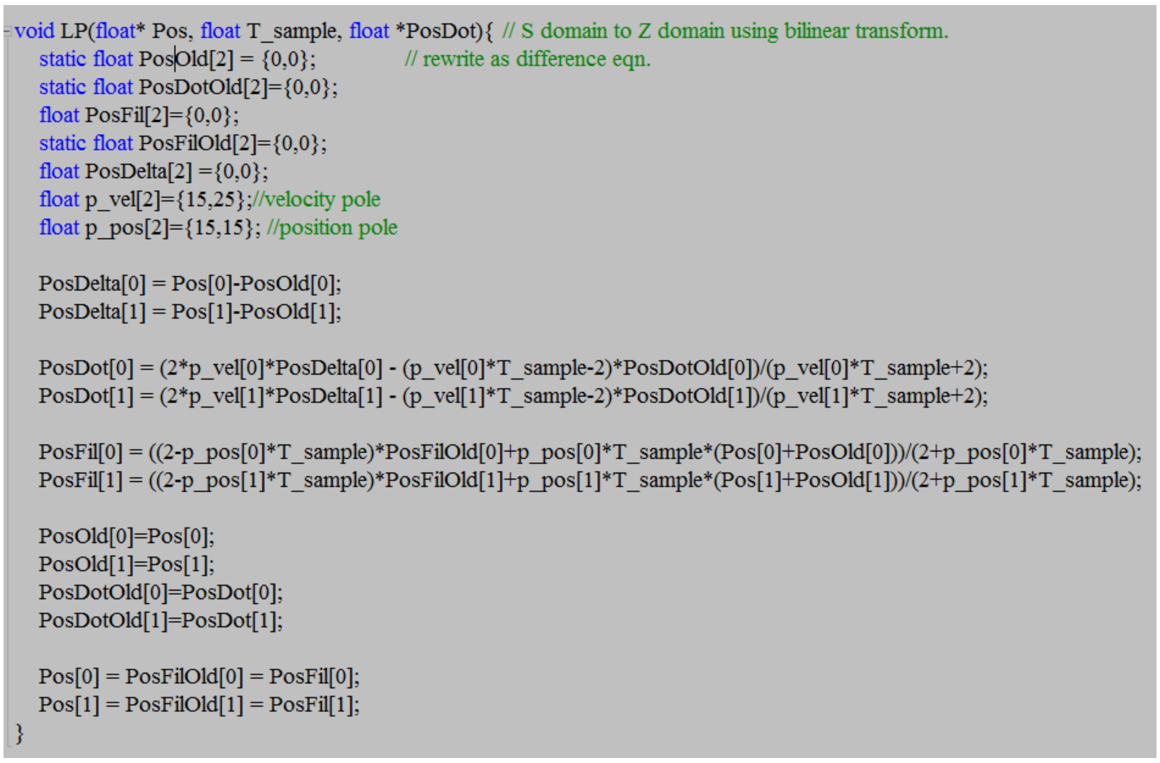
```
void PICon(float *Ref, float *Ang, float *Ang_dot, float *ConSig){//This function needs to be called every Ts
  //Ref          from joystick or human defined, in the unit of rad for angle, and m for height
  //Ang,         in the order of roll, pitch, yaw and height  in the unit of rad and m
  //Ang_dot      in the order of rolldot, pitchdot, yawdot and heightdot in the unit of rad/s
  //ConSig       control signal output on roll, pitch, yaw and throttle channel
  float Error_Ang[4]={0,0,0,0};
  static float Integral_Ang[4]={0,0,0,0};
  unsigned int i;

  for(i=0;i<4;i++){
    Error_Ang[i] = Ref[i] - Ang[i];
    if(i<3)  Error_Ang[i] = Wrap_180(Error_Ang[i]);
    Error_Ang[i] = Saturate(Error_Ang[i], Error_Max[i], -Error_Max[i]);

    Integral_Ang[i] += Ts*Error_Ang[i];
    Integral_Ang[i] = Saturate(Integral_Ang[i],Integral_Ang_Max[i],-Integral_Ang_Max[i]);

    ConSig[i] = Kp[i]*Error_Ang[i] + Ki[i]*Integral_Ang[i] + Kd[i]Ang_dot[i];

    if(3==i){
      ConSig[i] += HovRPM ;
      ConSig[i] = ConSig[i]/(sqrt(cos(Ang[0])*cos(Ang[1])));
    }

    ConSig[i] = Saturate(ConSig[i],ConSig_Max[i],ConSig_Min[i]);
  }
}
```

Detail Source Code 3. Function MotorMixing() in ConDyn.c



```
void Gyro(float* TrueData, float t, float *Gyro){//rad in, rad out, in the order of roll pitch yaw

  float noise[3];
  float walkBias[3];
  int i;
  for(i=0;i<3;i++){
    noise[i]= getGaussian(0, Std_n_Gyro[i]);
    walkBias[i] = getGaussian(0,Std_BRW[i]);
    Gyro[i] = TrueData[i] + Bias[i] + noise[i] + walkBias[i];
    if(1==i) Gyro[i]+= -6.22e-5*t;
  }

}
```

Detail Source Code 4. Function Gyro() in SenDyn.c



```c
void LP(float* Pos, float T_sample, float *PosDot){ // S domain to Z domain using bilinear transform.
    static float PosOld[2] = {0,0};             // rewrite as difference eqn.
    static float PosDotOld[2]={0,0};
    float PosFil[2]={0,0};
    static float PosFilOld[2]={0,0};
    float PosDelta[2] ={0,0};
    float p_vel[2]={15,25};//velocity pole
    float p_pos[2]={15,15}; //position pole

    PosDelta[0] = Pos[0]-PosOld[0];
    PosDelta[1] = Pos[1]-PosOld[1];

    PosDot[0] = (2*p_vel[0]*PosDelta[0] - (p_vel[0]*T_sample-2)*PosDotOld[0])/(p_vel[0]*T_sample+2);
    PosDot[1] = (2*p_vel[1]*PosDelta[1] - (p_vel[1]*T_sample-2)*PosDotOld[1])/(p_vel[1]*T_sample+2);

    PosFil[0] = ((2-p_pos[0]*T_sample)*PosFilOld[0]+p_pos[0]*T_sample*(Pos[0]+PosOld[0]))/(2+p_pos[0]*T_sample);
    PosFil[1] = ((2-p_pos[1]*T_sample)*PosFilOld[1]+p_pos[1]*T_sample*(Pos[1]+PosOld[1]))/(2+p_pos[1]*T_sample);

    PosOld[0]=Pos[0];
    PosOld[1]=Pos[1];
    PosDotOld[0]=PosDot[0];
    PosDotOld[1]=PosDot[1];

    Pos[0] = PosFilOld[0] = PosFil[0];
    Pos[1] = PosFilOld[1] = PosFil[1];
}
```

Detail Source Code 5. Function LP() in FilDyn.c