Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, master's theses and master's reports - Open

Dissertations, master's theses and master's reports

2014

REAL TIME FUZZY CONTROLLER FOR QUADROTOR STABILITY CONTROL

Pranav S. Bhatkhande
Michigan Technological University

Copyright 2014 Pranav S. Bhatkhande

Recommended Citation
Bhatkhande, Pranav S., "REAL TIME FUZZY CONTROLLER FOR QUADROTOR STABILITY CONTROL", Master's Thesis, Michigan Technological University, 2014.
http://digitalcommons.mtu.edu/etds/758

Follow this and additional works at: http://digitalcommons.mtu.edu/etds

Part of the Electrical and Computer Engineering Commons

REAL TIME FUZZY CONTROLLER FOR QUADROTOR STABILITY CONTROL

By

Pranav S. Bhatkhande

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Electrical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2014

© 2014 Pranav S. Bhatkhande

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Electrical Engineering.

Department of Electrical & Computer Engineering

Report Advisor: *Dr. Timothy C. Havens*

Committee Member: *Dr. Jeffery Burl*

Committee Member: *Dr. Yushin Ahn*

Department Chair: *Dr. D. Fuhrmann*

For my family and friends

Contents

List of Figures

Acknowledgments

Thanks to my parents, without their support, this would not have been possible.

I would like to thank my adviser, Dr. Timothy C. Havens for providing the guidance, advice and resources necessary to complete this report. Without his encouragement and support, this would have been very difficult. Special thanks to everyone at the Intelligent Robotics Lab for all the fun times. I would also like to thank my committee members, Dr. Jeffery Burl and Dr. Yushin Ahn for providing excellent inputs in the making of this report.

Lastly, I would like to thank all the members of the open-source UAV community.

Abstract

In this report, we develop an intelligent adaptive neuro-fuzzy controller by using *adaptive neuro fuzzy inference system* (ANFIS) techniques. We begin by starting with a standard *proportional-derivative* (PD) controller and use the PD controller data to train the ANFIS system to develop a fuzzy controller. We then propose and validate a method to implement this control strategy on *commercial off-the-shelf* (COTS) hardware.

An analysis is made into the choice of filters for attitude estimation. These choices are limited by the complexity of the filter and the computing ability and memory constraints of the of the micro-controller. Simplified Kalman filters are found to be good at estimation of attitude given the above constraints.

Using model based design techniques, the models are implemented on an embedded system. This enables the deployment of fuzzy controllers on enthusiast-grade controllers. We evaluate the feasibility of the proposed control strategy in a model-in-the-loop simulation. We then propose a rapid prototyping strategy, allowing us to deploy these control algorithms on a system consisting of a combination of an ARM-based microcontroller and two Arduino-based controllers. We then use a combination of the code generation capabilities within MATLAB/Simulink in combination with multiple open-source projects in order to deploy code to an ARM Cortex M4 based controller board.

We also evaluate this strategy on an ARM-A8 based board, and a much less powerful Arduino based flight controller. We conclude by proving the feasibility of fuzzy controllers on *Commercial-off the shelf* (COTS) hardware, we also point out the limitations in the current hardware and make suggestions for hardware that we think would be better suited for memory heavy controllers.

Chapter 1

Introduction

1.1 Quadrotor hardware and design

In Chapter. 2, we describe the quadrotor hardware and design. Quadrotors (also called quadcopters) are flying vehicles with four vertically-mounted rotors that are typically found in a “plus” or a “X” frame. The four arm-mounted motors provide four thrust vectors to the system. The mechanical simplicity of the system is contrasted by the complexity of the problem of controlling these systems. Being under-actuated, and having no redundancy, the control problem is of utmost importance. Quadrotors have multiple uses; they’re highly maneuverable and have the ability to reach places that might be dangerous to humans. Quadrotors can also be used as remote sensor pods. In disaster areas, quadrotors can provide a high quality information bridge between the disaster zone and the rescue teams. They can also help in automated inspection of infrastructure. These platforms can also provide soldiers with high-quality, timely information in a combat situation. The low cost, high maneuverability, and ease of manufacturing make these machines very interesting. Quadrotors are under-actuated, i.e., they have four motors to control six *degrees of freedom* (DOF). This makes the control difficult. In this report, we first briefly discuss the dynamics of the quadrotor. For a detailed description of the system dynamics consult reference [1]. It must be noted that the quadrotor dynamics here do not consider the coupling in very high speed maneuvers—we are primarily in interested stable platforms for remote-sensing use.

There are two main issues in creating controllers for quadrotors. Unlike ground platforms, there is a limitation on the amount of processing capability you can have on-board when flying. This limit is addressed to some degree by many *Commerical Off The Shelf* (COTS) controllers on the market today. Programming control algorithms within these limits of memory space and processing capability is a challenge. The second issue is the highly complex system dynamics involved when flying. *Inertial Measurement Unit* (IMU) noise characteristics also introduce some challenges in the control design.

1.2 Quadrotor dynamics

In Chapter 3 the dyanmics of the quadrotor are explained. This chapter defines the dynamic model used to create the fuzzy controller developed later. This model of for the quadrotor dynamics is developed in[1]. This chapter describes and develops the mathematical relation and also explains some challenges in create control algorithms for this platform.

1.3 State estimation filters

In Chapter. 4 we describe the filters we use to estimate the attitude of the vehicle. After we look at the dynamics of the quadrotor system, we move on to figuring out the best possible choice of attitude estimation algorithms. The APM 2.5 controller has an accelerometer and a gyroscope. We investigate a simple implementation of the Kalman filter for estimating the attitude of the device. We explain why the simple implementation is required to have minimum impact on memory. The Pixhawk flight controller[2], operating on the NuttX operating system h,as an open source implementation of an extended Kalman filter. We also look at this implementation when the constraints on the memory are less strict.

1.4 Control system design

In Chapter 5 introduce our proposed control strategy. We discuss how we developed this strategy using an ANFIS system[3]. We show that the ANFIS system is very useful in creating fuzzy controllers when the dynamics of the system are well known. With the vast amount of theory available for tuning PID controllers, a PD controller tuned to control the dynamic system can be used to initially train the fuzzy controller. Tuning a quadrotor is a difficult process when using PID controllers. In our experience with many commercially available platforms, tuning the PID controller loop within the controller to achieve desired performance is difficult. We hope to reduce the tuning problems with the proposed fuzzy control strategy.

We make use of the ANFIS [3] in order to create our required fuzzy rule base. We use the PD controller as the base for creating our initial controller rule base, however, we propose that one any available and relevant data can be used. We hope, that by publishing these results, people will be able to build their own controllers for a variety of custom configurations with relative ease. In order to create a proof of concept of the strategy working on embedded hardware, we decided to modify existing open-source projects and tools and build our system on top of them.

1.5 Hardware implementation

Chapter 6 describes a prototype hardware implementation for a fuzzy controller developed using the technique we developed in Chapter 5 This report also investigates a hardware implementation of our proposed fuzzy controller on an ARM-based microcontroller. To cut development time, we propose a method to rapidly develop fuzzy control algorithms and then implement these algorithms on COTS ARM-based components. We show that an enthusiast grade controller—the APM 2.5/APM 2.6—can be augmented to incorporate the more complicated controller. A system is developed where these Arduino-Mega based controllers can communicate over *user datagram protocol* (UDP) to ARM-based boards. The ARM-based chips handle the heavy processing, while the Arduinos are used as end actuators that provide the *pulsewidth modulation* (PWM) control signals.

We conclude by showing results of the fuzzy control strategy and comparing it with a PD controller. We then show the viability of the hardware implementation and hardware-in-the-loop simulator testing [4]. We perform testing on the Pixhawk flight controller and state our results.

1.6 Motivation

The Intelligent Robotics Laboratory at Michigan Tech deals with a number of applications of multirotors, where multiple sensor packages are taken into flight to collect data. It was observed that with commercial off the shelf parts, putting together a flying configuration is relatively easy; however when you start modifications to the frame design, or start modifying the payloads, the dynamics of the vehicle change. Modeling these dynamics becomes a challenging task. One approach is to design an optimal controller, however doing so requires a full knowledge of the system dynamics. The other philosophy is to design an adaptive controller, which can perform controlling action without knowing the exact dynamic model of the plant. An ideal solution would probably use a hybrid of these two approaches. The tuning problems that we faced when assembling our quadrotors was the primary motivation of this fuzzy control design.

Chapter 2

Quadrotor hardware and design

In our experiments, we decided to use off the shelf parts to assemble our quadrotor. In this section we document the decision process for our experiments.

2.1 Mechanical Parts and Motors

The mechanical parts required for the quadrotor are minimal, this makes the quadrotor an ideal and low cost flying platform suitable for many purposes. We had to decide upon the frame, propellers, speed controllers and other small parts in order to assemble the system. We purchased a fiber glass frame from an online retailer (450mm wide, 55mm high), this seemed to be a popular choice among the enthusiast quadrotor community. The frame is made from glass fiber and is of reasonably strong construction, its low cost makes it an ideal choice of experimental purposes. The propellers are chosen to be 8x4.5 inch propellers, driven by Turnigy Aerodrive SK3 motors. The speed controllers are Turnigy Multistar.

2.2 Electronics and software

2.2.1 Controller Boards

In this report, we evaluate a number of controller boards for the purpose of implementing a fuzzy controller. The boards are listed here in the order in which we evaluated them

- *APM 2.5 Controller board* [5]– The APM 2.5 controller is in essence an Arduino Mega 2560 board fused with an IMU, and the ability to output PWM signals. On board is a 3 axis gyroscope, 3 axis accelerometer, and has an additional barometric sensor for sensing altitude, however we realized that barometers do not work well at low altitude. This controller board has 32KB of memory, and 0.5 KB is used by the system's bootloader, it also has 2 KB of SRAM and 1KB EEPROM. This memory limitation is crucial to note.

- *Pixhawk PX4*– The Pixhawk controller is an incremental upgrade from two boards, the PX4FMU and PX4IO [2]. This board has a more powerful Cortex M4 processor fused together with an IO board like the APM 2.5 controller capable of PWM signal generation. A 6 axis accelerometer and gyroscope are also on-board.

- *Gumstix Overo Firestorm*– The Gumstix is much closer to smart-phone hardware rather than a flight controller board. It runs a dual core ARM-Cortex A8 chip, 512 Mb of RAM. We used the Gumstix in some hardware testing.

2.2.2 Batteries and Power

Two options are possible for delivering power to the quadrotor.

- Tethered Power– A tether is created between the vehicle and a power supply, this solution is ideal for bench testing and flying indoors.
- Batteries– 3S, 2200mAH batteries power the quadrotor. The flying time possible with these batteries is about 20 minutes.

2.2.3 Wireless and radio-frequency units

- Wireless Transmitter– A Futaba radio transmitter, receiver pair is used as the remote control.
- Wireless data transmission– Wireless data transmission is possible over MAVlink[6].

The base station receives this data over serial at a baud rate of 57600bps

2.2.4 Ground-station software

- Qgroundcontrol[7]– Developed as an open-source project, Qgroundcontrol implements MAVlink on the ground station. This tool helps calibrate sensors, controller settings, and also provides options to debug sensor data.
- Mission Planner– Mission planner is similar to QgroundControl, we use this software package mainly to perform tests with the stock Ardupilot software on the APM 2.5 flight controller.

Chapter 3

Quadrotor Dynamics

In this chapter, the dynamics of the quadrotor are explained. This model is based on the model developed in [1]. We begin by deciding upon a dynamic system model for the quadrotor system. The frame that is taken into consideration for developing our control strategy is shown in Fig. 3.1. It must be noted that the z-axis is taken in the downward direction—toward the ground or into the paper. This is especially important since it follows the aerospace convention. The directions for the motors are also shown in Fig. 3.1. Reference [1] explains the dynamics frame in more detail. We follow the same expressions in [1] for rolling torque and pitching torque.

Consider the vehicle as shown in Fig. 3.1, the vehicle has a thrust in the upward direction, the negative direction of the z axis. Let us denote the motor thrusts as T_i, the speed of each

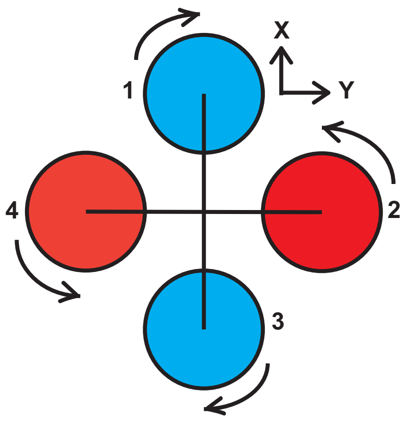

Figure 3.1: 'Plus' quadrotor frame—*x*-axis points forward, *y*-axis to the right, and *z*-axis points down toward ground (into the paper in this figure)

motor as ω_i, b is the lift constant, and $i \in \{1,2,3,4\}$ represents the labels for the motors. Hence, the thrust from each motor can be calculated as

$$T_i = b\omega_i^2,\ i = 1,\ldots,4.$$

This upward thrust is opposed by the force of gravity acting in the downward direction, i.e., $F_g = mg$. So, for a vehicle of mass m, this is given by

$$F_t = mg - \sum_{i=1}^{4} T_i.$$

In order to *rotate* or yaw the vehicle, the controller uses a pairwise difference in the thrust of motors 1 and 3. In order to *roll* the vehicle, a correspondingly difference of force is input to motors 2 and 4.

Consider r is the distance between the center of the airframe, as seen in Fig. 3.1. We now define two torque values, τ_x and τ_y, as the *rolling torque* and *pitching torque* acting along the x and y axis respectively:

$$\tau_x = rb(\omega_4^2 - \omega_2^2); \tag{3.1a}$$

$$\tau_y = rb(\omega_1^2 - \omega_3^2). \tag{3.1b}$$

Now, we consider the *aerodynamic drag*, denoted as D, that acts to oppose thrust. The drag component corresponding to every T_i is denoted as D_i. The factor k depends on factors similar to the lift constant b. Thus, *aerodynamic drag* is defined as

$$D_i = k\omega_i^2.$$

This aerodynamic drag creates a reaction torque that acts to oppose the intended motion of each of the motors. This reaction torque is given by

$$\tau_z = D_1 - D_2 + D_3 - D_4. \tag{3.2}$$

As can be seen, (3.1) and (3.2) describe the torque along each of the three axes of the vehicle as a function of the motor speeds for each motor.

Given a torque vector

$$\xi = (\tau_x, \tau_y, \tau_z)^T,$$

the rotational equations of motion

$$I\dot{\mathbf{a}} + \mathbf{a} \times I\mathbf{a} = \xi, \tag{3.3}$$

where I is the inertia matrix and $\mathbf{a}$ is the angular velocity vector around each axis. I is diagonal for an ideal quadcopter model,

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}.$$

Now consider ω_T as the motor speed vector, we define the matrix A as

$$A = \begin{pmatrix} -b & -b & -b & -b \\ 0 & -rb & 0 & rb \\ rb & 0 & -rb & 0 \\ k & -k & k & -k \end{pmatrix}.$$

We define ω_T as

$$\omega_T = \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix},$$

and γ as the thrust/torque vector

$$\gamma = \begin{pmatrix} \sum_{i=1}^4 T_i \\ \tau_x \\ \tau_y \\ \tau_z \end{pmatrix}.$$

We now combine A, γ and ω_T in this relation,

$$\omega_T = A^{-1}\gamma. \tag{3.4}$$

The position is x, y and z. The pitch, roll and yaw angles are denoted as θ_r θ_p and θ_y. The vehicle is under-actuated: it needs to generate a pitch angle θ_p to create a forward velocity. Control over θ_p and θ_r enables control of the quadrotor. Modern enthusiast controllers like APM 2.5 (*Arducopter*) output a state vector. Note that, in order to calculate the x, y, z positions one has to calculate the appropriate rotation matrix $\Re$. More can be read about the dynamics of this vehicle in [1]. The final state vector of the vehicle is

$$\mathbf{x} = (x, y, z, \theta_r, \theta_p, \theta_y, \dot{x}, \dot{y}, \dot{z}, \dot{\theta}_r, \dot{\theta}_p, \dot{\theta}_y), \tag{3.5}$$

where $(x, y, z, \theta_r, \theta_p, \theta_y)$ is the 6 DOF pose of the vehicle (i.e., position and rotation) and $(\dot{x}, \dot{y}, \dot{z}, \dot{\theta}_r, \dot{\theta}_p, \dot{\theta}_y)$ are the rates of change in each of the 6 DOF pose variables. In our real-world system, the state vector is provided to the controller by an *inertial measurement unit* (IMU) or some other collection of pose-estimate sensors[8].

Chapter 4

State estimation filters

4.1 Introduction

Kalman filters are widely used in top of the line enthusiast controllers. Thanks to improving performance in low-powered micro-controller devices, it is now possible to implement these filters on these power limited devices. Kalman filters significantly improve our estimation of attitude, compared to raw sensor information. We will be looking at two implementations of the Kalman filter algorithm[9]. [9] and the associated github page sheds some light on how one could create an efficient Kalman filter algorithm for an IMU unit. [10] has a good overview on how one could improve the low precision issues with commercial, off the shelf, cheap IMUs. Kalman filters seem to be used extensively in attitude estimation problems. In our experiments using the Kalman filter greatly improved our estimate of attitude. The Pixhawk micro-controller toolchain [11] implements an Kalman filter. We make some trade-offs between the memory efficiency of the filter and its performance. We briefly describe a simplified version of the Kalman filter , and then go over some performance metrics of this filter on embedded hardware.

4.2 Simplified two-state Kalman filter

4.2.1 Terms and definitions

We use a simplified version of the Kalman filter here. The idea is to avoid huge matrices. We try to find a balance between performance with respect to accuracy in attitude estimates and computation requirements on embedded hardware. Note here that with the Pixhawk, it might be possible to implement an extended Kalman filter; however, this leaves us with a very little memory left for the fuzzy controller. We use a simplified version of the filter, implemented as published by tj-electronics [9], an open-source library [12].

We now briefly look at our simplied Kalman filter. The state here is defined by $\mathbf{x}_k$,

$$\mathbf{x}_k = \begin{pmatrix} \theta \\ \dot{\theta}_b \end{pmatrix},$$

The term θ represents the angle, while the term $\dot{\theta}_b$ represents the gyroscopes' drift over time (bias). Assume that $\hat{x}_{k-1|k-1}$ is the previous state estimate, $\hat{x}_{k|k-1}$ is the *a priori* state estimate, $\hat{x}_{k|k}$ is the *a posteriori* error estimate.

F is our state transition model, F is applied to the previous state estimate, $\hat{x}_{k-1|k-1}$.

$$F = \begin{pmatrix} 1 & -\delta t \\ 0 & 1 \end{pmatrix},$$

B is the input matrix, and is defined as,

$$B = \begin{pmatrix} \delta t \\ 0 \end{pmatrix},$$

B is applied to the input of to the system, in this case, it is our measurement in *deg/sec* from the gyroscope, this is expressed as $\dot{\theta}_k$.

Now we assume that the true state of the system is defined by $\mathbf{x}_k$, where F is our state transition matrix, B is our input matrix and $\dot{\theta}_k$ is the input and w_k is the process noise. The process noise w_k is considered to have zero mean and co-variance Q_k. Q_k is defined as

$$Q_k = \begin{pmatrix} Q_\theta & 0 \\ 0 & Q_{\dot{\theta}_b} \end{pmatrix} \delta t.$$

Process noise is thus defined as,

$$w_k \sim N(0, Q_k). \tag{4.1}$$

We now define the state $\mathbf{x}_k$ of the system.

$$\mathbf{x}_k = F\mathbf{x}_{k-1} + B\dot{\theta}_k + w_k, \tag{4.2}$$

We cannot observe the state $\mathbf{x}_k$, but we can make a measurement. z_k is the measurement made at a time k. True space now has to be mapped into the observed space, we do this with H, our observation model. H is defined as

$$H = \begin{pmatrix} 1 & 0 \end{pmatrix}.$$

When we make a measurement, we can only do so with a certain degree of certainty, this is where we factor in the measurement noise. This measurement noise is considered to be normal, with mean zero and variance L. The co-variance L is equal to the variance of v_k.

This is defined as

$$L = var(v_k). \tag{4.3}$$

The measurement noise is thus defined as,

$$v_k \sim N(0, L). \tag{4.4}$$

z_k, the measurement made at time step k is thus expressed as

$$z_k = Hx_k + v_k. \tag{4.5}$$

Now let us define the *a priori* error estimate and *a posteriori* state estimate

$$e_k^- = x_k - \hat{x_k}^-, \tag{4.6}$$

$$e_k = x_k - \hat{x_k}. \tag{4.7}$$

Corresponding to these, the *a priori* and *a posteriori* error co-variance is then

$$P_k^- = E[e_k^- e_k^{-T}], \tag{4.8}$$

$$P_k = E[e_k e_k^T]. \tag{4.9}$$

These equations are defined as described in [13].

We then define the Kalman gain matrix, K. The use of the Kalman gain matrix will be made clear later. Put briefly the Kalman gain determines the amount of trust we place on our predicted state, or our measurement.

$$K = \begin{pmatrix} K_0 \\ K_1 \end{pmatrix}.$$

4.2.2 Kalman filter Equations

The Kalman filter equations can be divided into two parts, the first stage is called the predict stage, and this followed by the update stage.

In equation. 4.10, we predict the state at a further time-step.

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + B\dot{\theta}_k, \tag{4.10}$$

we now project the error variance ahead in equation. 4.11

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q_k. \tag{4.11}$$

The equations 4.11 and 4.10 together represent the prediction part of the process. We now explain the update equations.

The measurement z_k is made, this enables us to calculate the innovation, denoted by $\tilde{y_k}$. The observation matrix H maps the *a priori* state into the observed state, this is subtracted from the measurement from the accelerometer or gyroscope, which results into a single value for $\tilde{y_k}$. Equation 4.12 defines this innovation,

$$\tilde{y_k} = z_k - Hx_{k|\hat{k}-1}. \tag{4.12}$$

We now determine a quantity which defines how much we trust our measurement. If the measurement noise increases, our innovation covariance increases, this makes our measurement seem unreliable. The observation model, simply maps the *a priori* state into observed space. We now update the innovation co-variance in equation 4.13.

$$S_k = HP_{k|k-1}H^T + L. \tag{4.13}$$

The next step is to figure out the Kalman gain. When we have a very large value for S_k it would mean that our measurements vary a lot, this would reduce our trust in our measurements, and put more trust into our predicted state. P_k tells how much we expect the state $\mathbf{x}_k$ is expected to change. In order to change the estimate by a larger amount, one would need a larger Kalman gain. This is more formally noted in equation4.14

$$K_k = P_{k|k-1} H^T S_k^{-1}. \tag{4.14}$$

We then estimate the next state:

$$x_{\hat{k|k}} = x_{k|\hat{k}-1} + K_k \tilde{y_k}. \tag{4.15}$$

The error co-variance is updated as follows

$$P_{k|k} = (I - K_k H) P_{k|k-1}. \tag{4.16}$$

Equations 4.15 and 4.16 are called the update equations. This process is repeated thrice, to find the roll, pitch and yaw for our system.

The open-source community contributes tremendously. For example, a certain set of values for variances that work well for certain IMU's are available [9]. After some testing and experimentation, this filter is found to perform satisfactorily on the APM 2.5 flight controller, and then Pixhawk flight controller[14]. [14] has published a modified version of the same filter, and has included that in a experimental control system for the APM 2.5 flight controller. This serves as a good reference for building better filters.

4.3 Filter processing challenges

All filters on an embedded hardware need to be optimized for memory efficiency, huge matrices declared as float's or double's consume memory that would be consider unacceptable in many applications. The Gumstix Firestorm COM is capable of running an extended Kalman filter and a fuzzy controller at the same time, without any performance impact, the ARM Cortex-A8 architecture is found to be extremely adept to deal with these challenges. The ARM Cortex-M4 processor and the ARM Cortex-M3 failsafe co-processor on the Pixhawk boards can also run a extended Kalman filter, but these reduce the amount of memory and processing capability available to other parts of the system, most importantly, the control system. As it stands right now, the Pixhawk runs best with a simple , less memory intense control system and a Kalman filter for attitude estimation. We implement a fuzzy-attitude control system on the Pixhawk while using the Kalman filter, we did this reducing the amount of membership functions in the input space, and using the takagi-Sugeno inference system allowing us to eliminate membership functions in the output space.

Chapter 5

Control System Design

5.1 Control Strategy Design

5.1.1 Traditional control strategy

To stabilize the quadrotor system, the typical strategy is to have three PID control loops that continuously measure the current pitch, roll and yaw; given by (θ_r, θ_p, θy) and the change in the respective quantities ($\dot{\theta}_r$, $\dot{\theta}_p$, $\dot{\theta}_y$) relative to some desired pose. The request for the change in attitude is by the user in the form of remote control commands, by a radio, or predefined flight-plan[15]. Tuning the parameters is a very difficult task, for this under-actuated system. Although it might be theoretically possible to analytically tune

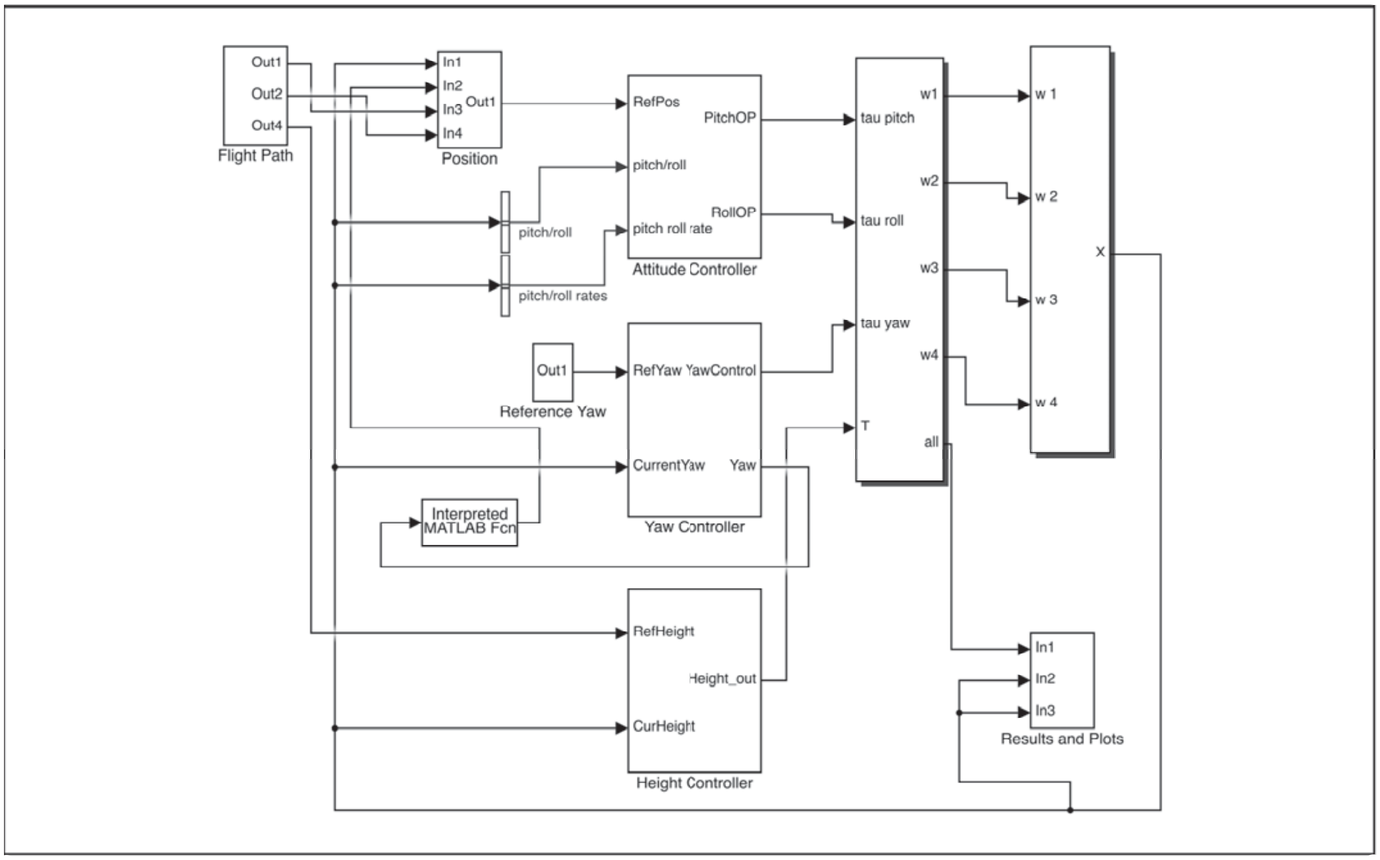

Figure 5.1: Overall Control Loop

the gains of the PD controller for the quadrotor, reforming this analysis for every new configuration of the quadrotor becomes difficult and tedious. Modified tuning techniques can also be used to tune the PD controller [16]. In our application, the PD controller is tuned using classical tuning methods for optimal response as described in [1]. Figure. 5.1 shows the overall control loop.

5.1.2 PD Controller

As noted in chapter 3, section 3, we need to control the *roll, pitch, yaw*; stated as $(\theta_r, \theta_p, \theta_y)$, and $(\dot{\theta}_r, \dot{\theta}_p, \dot{\theta}_y)$. We define the *Proportional gain values* for *roll, pitch, yaw* as $(K_{p_r}, K_{p_p}, K_{p_y})$ and derivative gain values as $(K_{d_r}, K_{d_p}, K_{d_y})$. Note that a feedforward constant C is added to the altitude controller to balance the weight of the quadrotor against the force of gravity given as

$$C = \sqrt{\frac{mg}{4b}}. \tag{5.1}$$

The control equations are

$$\tau_x = K_{p_r}(\hat{\theta}_r - \theta_r) + K_{d_r}(\hat{\dot{\theta}}_r - \dot{\theta}_r); \tag{5.2a}$$

$$\tau_y = K_{p_p}(\hat{\theta}_p - \theta_p) + K_{d_p}(\hat{\dot{\theta}}_p - \dot{\theta}_p); \tag{5.2b}$$

$$\tau_z = K_{p_y}(\hat{\theta}_y - \theta_y) + K_{d_y}(\hat{\dot{\theta}}_y - \dot{\theta}_y); \tag{5.2c}$$

$$T = K_{pZ}(\hat{Z} - Z) + K_{d_Z}(Z - \dot{Z}) + C. \tag{5.2d}$$

5.1.3 Control splitting

The outputs generated by the four controllers above are split among the four motors. This is called control splitting. Let the contribution of each be denoted by f_r, f_p, f_y and f_z respectively for roll, pitch, yaw and altitude.

$$\omega_1 = f_p + f_y + f_z \tag{5.3a}$$

$$\omega_3 = -f_p + f_y + f_z \tag{5.3b}$$

$$\omega_2 = -1(-f_r - f_y + f_z) \tag{5.3c}$$

$$\omega_4 = -1(f_r - f_y + f_z) \tag{5.3d}$$

Note that the output of the *altitude controller* is added equally to all motors. This allows the roll and pitch of the vehicle. This control splitting block is the same for both the PD controllers and the Fuzzy controller desribed in section 5.1.4. In Fig. 5.1 we show the placement of the control splitting block.

5.1.4 Fuzzy control strategy

In this section, we develop a fuzzy control strategy to control the quadrotor described in Sec. 3. We propose a strategy based on the ANFIS system[3]. We first set up an

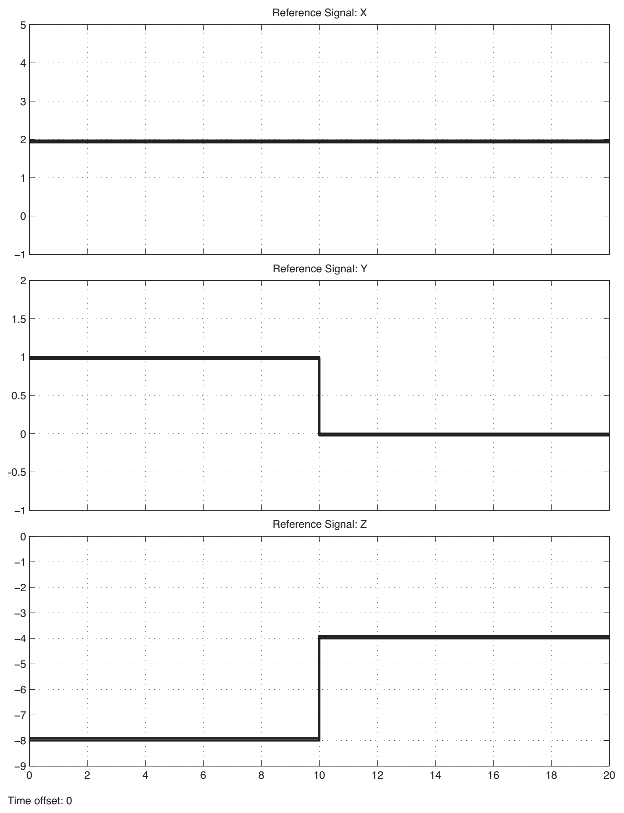

Figure 5.2: Experimental (x, y, z) signals

experiment, collect data from this experiment and then create a controller from the training data obtained. The derived controller is used to control the quadrotor.

5.1.4.1 Experimental Setup

The goal of the experiment is to create a closed loop scenario, in which we can test control algorithms against the approximate dynamics model described in Sec. 3. We define x, y and z coordinates, the (x, y, z) is where we could like our quadrotor to go. In the absence of *Radio Control* (RC) commands, these serve as a good replacement. For illustration, consider Fig. 5.2; here we keep the value of *x* constant and request changes in the *y* and *z* coordinates. Various input conditions are simulated.

5.1.4.2 Generating training data

We first log data from the experiment set up above. The experiment is first run for the *z* controller, in this case, we first train the system to go from a height of 0 to a maximum step height, thus simulating the step response. We log data for *z*, *dz* and *rpm change* due to the *z* controller. A similar process is repeated for the attitude control and yaw control of the vehicle. Data logged from this process is then fed into the ANFIS system.

5.1.4.3 Learning controller from training data

ANFIS combines a neural network with fuzzy logic and thus achieves a learning mechanism for a fuzzy rule base. It is widely regarded as an universal estimator [17]. We propose that the controller only has to learn once, in a simulation or a hardware-in-loop test, and the code deployed to the embedded hardware would perform well compared to a PD or PID controller. [18] follows a similar training procedure on similar training data. In this effort, we collect training data from the above experiment and feed it into the ANFIS system [3]. The follow parameters are used in the ANFIS system:

- number of inputs: 2
- number of outputs: 1
- number of rules: 25
- type of membership functions: Gaussian Bell functions
- fuzzy inference system: Sugeno
- intersection: product
- union: max
- defuzzification : weighted average

Figures 5.3–5.6 show the surface views for the four learned controllers. The ANFIS system has the ability to leverage neural networks and fuzzy rules to create a fuzzy inference system. We do this for all our sets of the training data, and create the rule bases for our controllers.

While ANFIS systems are very good at producing high quality fuzzy rule bases (as a universal estimator), they are computationally complex. However, hybrid learning algorithms [19] could be used to produce good control rule bases more efficiently.

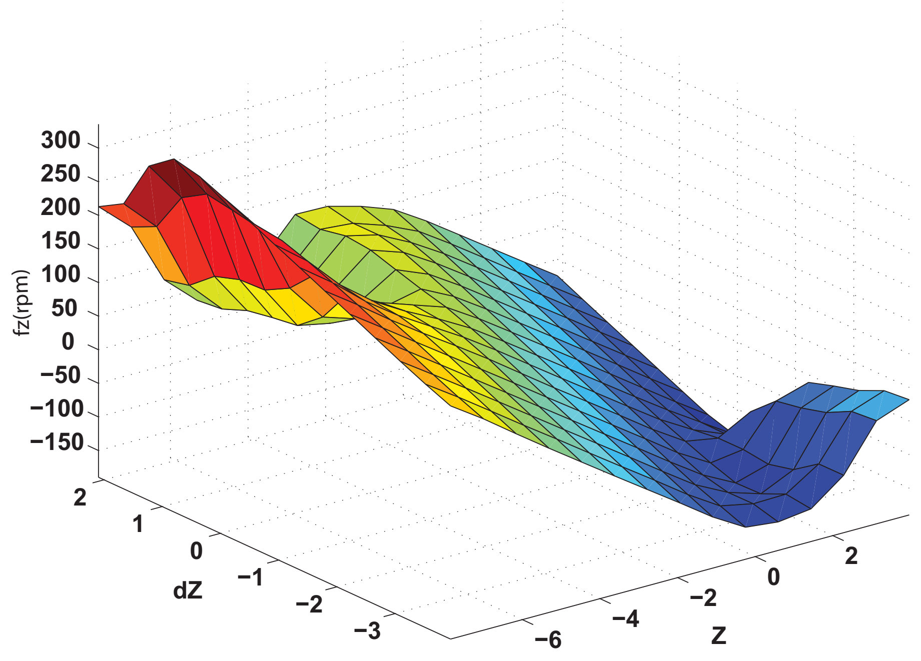

Figure 5.3: Surface view: Height controller

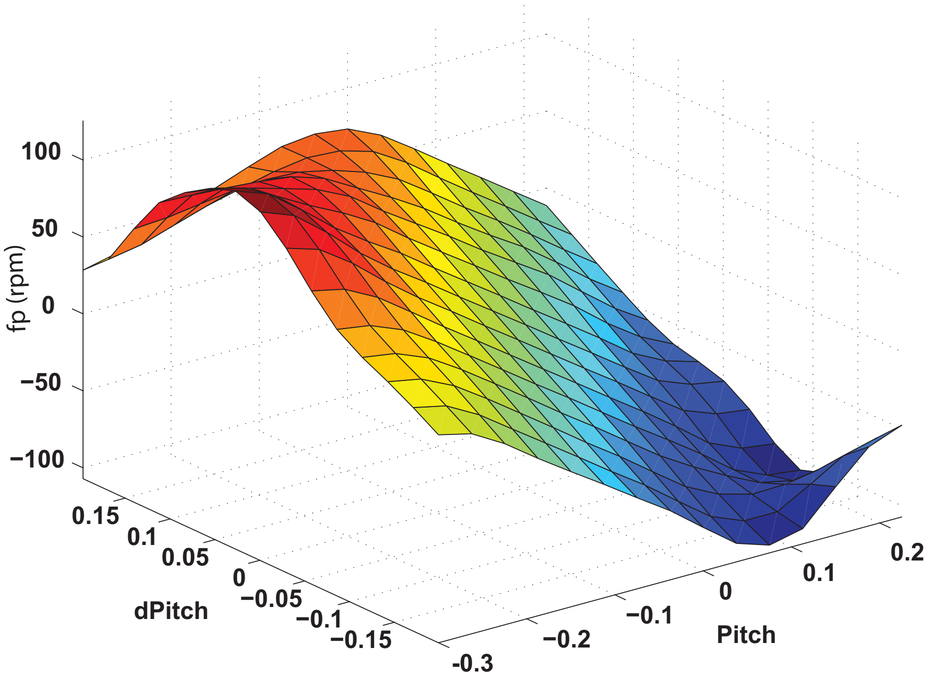

Figure 5.4: Surface view: Pitch controller

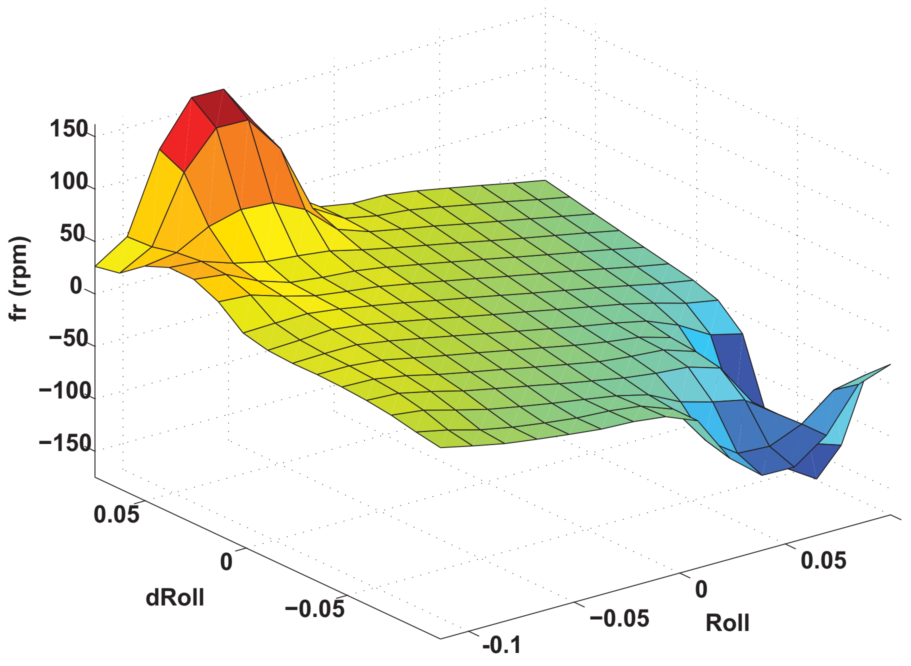

Figure 5.5: Surface view: Roll controller

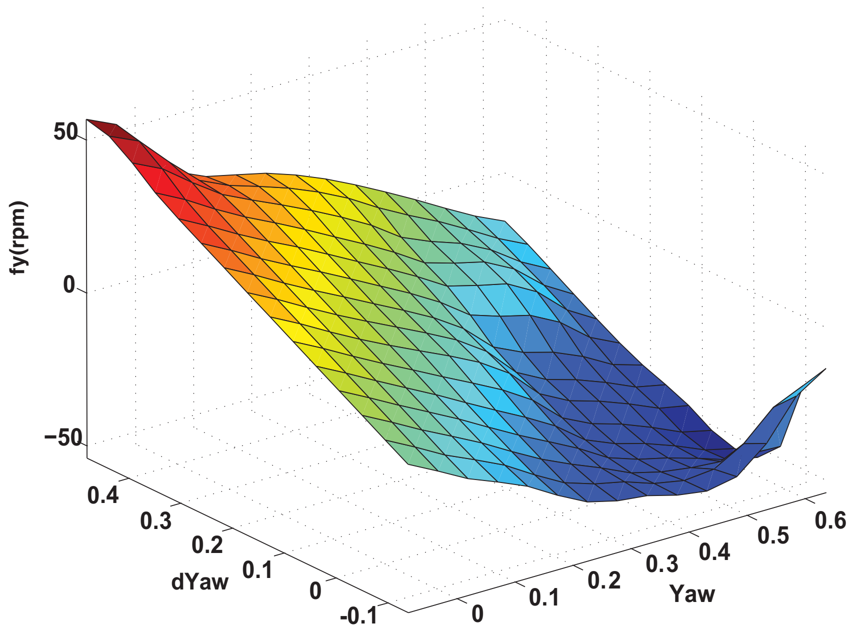

Figure 5.6: Surface view: Yaw controller

5.2 Generalization of the controller training process

As we explain in the previous section, we developed a fuzzy control strategy to stabilize the vehicle. We used the PD controller data as our training data, and validated this idea by giving new sets of goals not present in the training data. This proves the viability of this method. However, in the absence of control data, we could still hypothesize that developing an ANFIS derived fuzzy controller is possible simply by manually creating training data. We illustrate this idea further in the report when we create a realization of a 'proof-of-concept' controller on an ARM-based micro-controller. We will be developing a controller that takes two inputs, the current roll, and the change in roll request; the output of this controller will be the attitude command sent to the vehicle.

Chapter 6

Hardware Implementation

6.1 Overview

In this chapter, we discuss the hardware implementation of the fuzzy controller we derived earlier in the report. We attempted the implementation on a variety of micro-controllers, we summarize the implementation, and then suggest our best case implementation. We go further and then suggest a hardware specification that we think would be optimal for this fuzzy controller design. Fuzzy controller design is an important aspect of this report.

6.2 Challenges in hardware implementation

Fuzzy controllers are fundamentally more complex to implement, most enthusiast grade microcontrollers work on Arduino-based boards. Top of the line Arduino Boards do not have the ability to implement a fuzzy controller; this limitation is due to the memory and the processor architecture. Our initial experiments on the APM 2.5 (Arduino-Mega derived flight controller) showed that the APM 2.5 board failed to implement the fuzzy controller due to heavy memory restrictions and the lack of hardware floating point.

To address this, we first attempted an implementation on the Raspberry Pi. The Pi performs well with a single fuzzy controller (e.g., the height controller), but struggles to keep up when all four controllers are implemented. The Gumstix system was found to be better performing; this is due to it being a dual core chip, in addition to that it is also clocked higher.

Due to lack of direct interfacing between Arduino Uno and the Gumstix, a method of communication must be decided upon. The method of communication between the Arduino and the Gumstix is decided upon to be *User Datagram Protocol* (UDP). The reasons for this is as follows. The GPIO pins of the Gumstix can only read and write logical values, they are not useful for sending IMU information. The MATLAB implementation for both microcontrollers supports UDP; hence, UDP is decided upon for its universal nature and fast processing.

We also used the Pixhawk controller for validating our models. To use the Pixhawk, we had to deploy code to the Pixhawk controller using model based design techniques. This process, however was slightly more involved. Pixhawk [2], has developed a toolchain [11]. We then used a wrapper [14] originally written for the px4fmu version 1.x boards, and ported that to work with version 2.x boards. This wrapper code is then added to the simulink model, and the generated code is built using *make*, cross-compiled using arm-gcc, and then uploaded to the flight controller. We discuss this solution in depth further.

Programming various types of microcontrollers in various languages leads to a huge development overhead. This time can be cut down by a rapid prototyping strategy. We use a strategy based on that proposed in [20]. Similar strategies are used in automobiles for programming Electronic Control Units [21].

6.3 Implementation results on various platforms

In this section, we test implementation of the fuzzy control algorithm on various hardware platforms.

6.3.1 Combination of APM 2.5 and Gumstix Overo Firestorm

6.3.1.1 Hardware and software architecture

The Gumstix overo micro-controller is an ARM-Cortex A8 based system. This micro-controller is supported from Simulink as a Simulink target. This allows us to follow a strategy similar to [21]. The APM 2.5 controller is a modified version of the Arduino-Mega 2560 micro-controller platform. The Gumstix can run multiple operating systems, including a minimal implementation of Linux. The APM 2.5 controller runs a modified Arduino environment.

6.3.1.2 Development process

A workflow [21] is developed to implement the control algorithm on hardware. First, we develop the control algorithm in Matlab/Simulink. We generate C/C++ code using code generation capabilities within Matlab. After the code is generated, the native code is exported to the microcontrollers. The build system for Arduinos required modification to be used with Arducopter [22]. The Gumstix code was generated directly from Matlab. Note that with this process, three microcontrollers are programmed to perform various tasks. These are listed as below and illustrated in Fig. 6.1. More can be read about such a

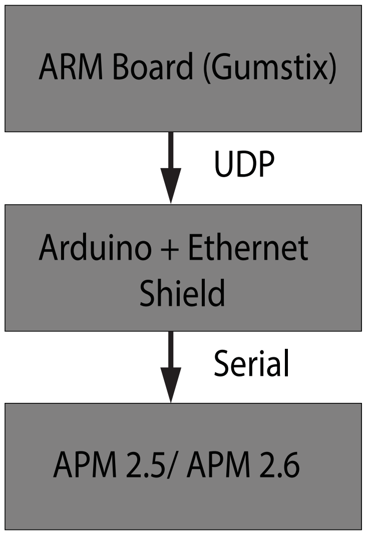

Figure 6.1: Hardware Implementation

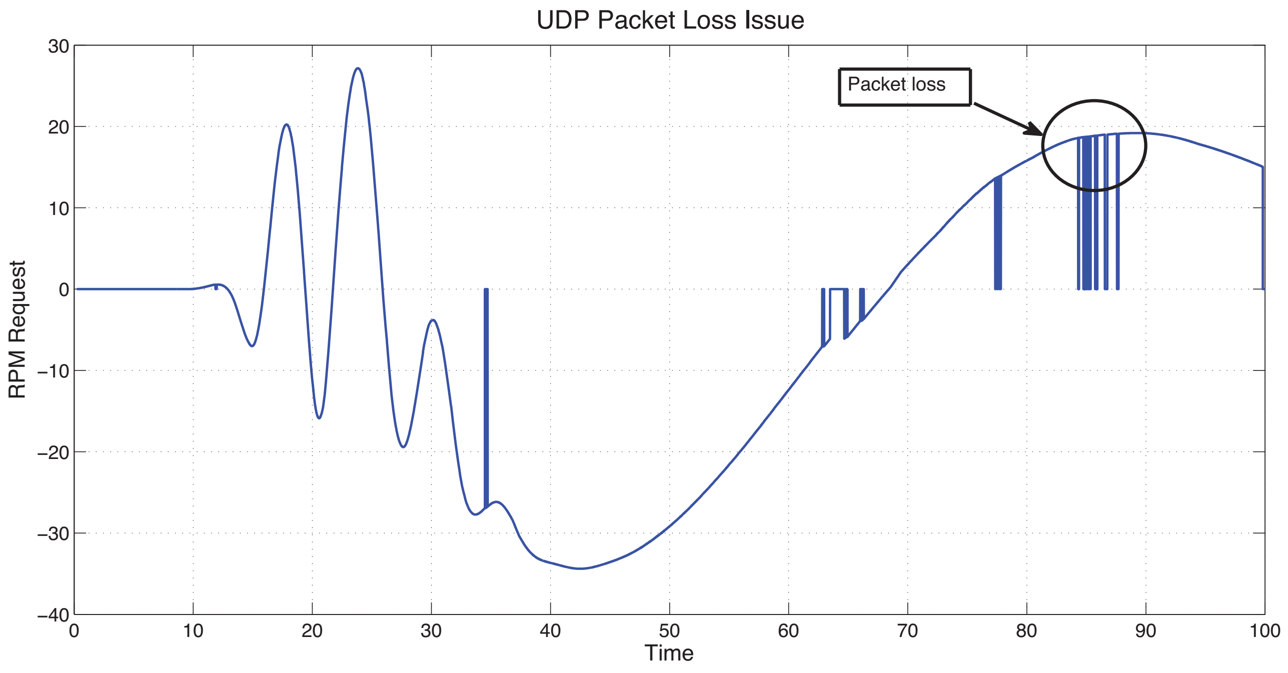

Figure 6.2: Gumstix UDP Result

strategy in [20].

1. Gumstix Overo FIRESTORM– heavy processing, filtering, processing fuzzy

controller;

2. Arduino UNO– basic relay between APM 2.5 [22] and Gumstix microcontroller;

3. Arduino Ethernet Shield– UDP packet bridge between Gumstix and Arduino.

A custom built Arduino Board with an onboard IMU (APM 2.5) [22] generates the Yaw, Roll, and Pitch, and the difference in all those quantities per time step. These data are sent to the Arduino UNO board, and the Arduino UNO board acts as a relay between the APM controller and the Gumstix Microcontroller. We use the Arduino Ethernet Shield to transfer the data from the APM to the Arduino. The Arduino then sends the data via UDP to the Gumstix microcontroller. The Gumstix microcontroller returns the control data via the reverse loop, enabling us to send commands to the APM controller. The APM controller is connected to the speed controllers. Figure 6.1 shows this process.

6.3.1.3 Results: Gumstix with UDP

In order to get an estimate of performance of this controller, we set up a model in the loop test. Fuzzy controllers for roll, pitch, yaw and height were first deployed to the Gumstix board. The host computer was set up to simulate the Quadrotor plant. The model used for this plant is the same model as described here [1]. Zero-order hold blocks were used to carry out the necessary analog to digital conversion in *Simulink*. The controller performance was found to be very close to the performance achieved using a continuous time simulation. However, UDP introduces packet-loss. This packet loss in shown in Fig. 6.2. The more powerful dual core ARM Cortex A8 processor however is able to process the four fuzzy controllers without any issues. The processor usage in this case is always less than ten percent. This result is expected, as the Gumstix has support for hardware floating point, and is clocked around 1GHz.

6.3.1.4 Primary issues with this method

Quadrotors need a very high rate of PWM channel update, the inner loop of many commercial quadrotors, also called the rate controller, operates at 250Hz or above. The transport delay introduced by UDP between the ARM processor, and the APM 2.5 controller do not make this a viable option. This points to the requirement of a custom hardware module, we discuss this later.

6.3.2 Pixhawk PX4 Controller Board

6.3.2.1 Hardware and software architecture

The Pixhawk system relies heavily on open-source software contributions and open-source hardware. Pixhawk, combines two previous generation components into one single fused

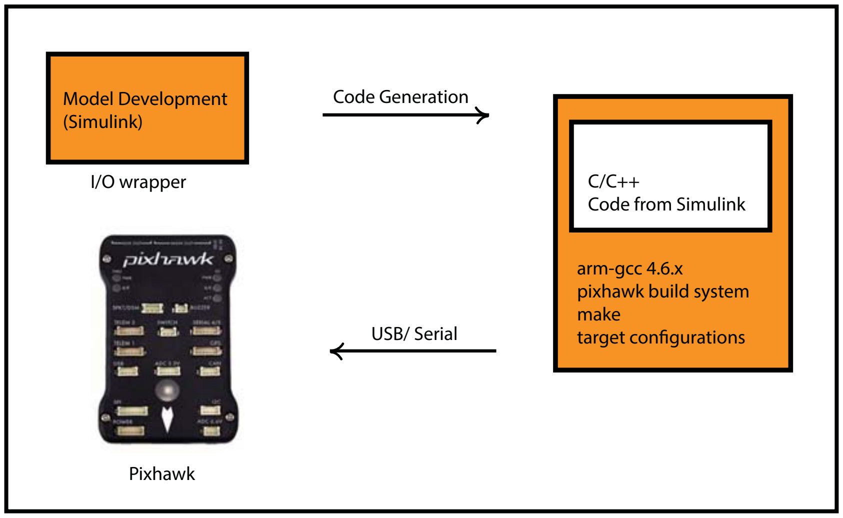

Figure 6.3: Development Workflow - Pixhawk

product. The Pixhawk system thus consists of one *Pixhawk flight management unit* (px4fmu), and one *Pixhawk input output board* (px4io). The px4fmu consists of a 32-bit ARM Cortex M4 processor running at 168MHz, supplemented by a secondary ARM Cortex M3 failsafe co-processor unit. The px4io unit consists of interfacing circuitry between the ARM-M4 module and the PWM channels, LED and other miscellaneous hardware interfaces and parts. We now describe the software stack for this board briefly. The board runs on *Real Time Embedded Operating System* (RTOS)- NuttX [23][24]. This is an operating system with a small data footprint and provides some important features required here. Firstly, it provides for the standard C/C++ library implementation on the Pixhawk hardware, in addition to this, it also provides a minimal *Virtual File System* (VFS) and support for multiple hardware devices.

6.3.2.2 Development process

In Fig. 6.3, we illustrate our work flow in developing models for this particular controller. The process is equivalent to hand-writing C++ code on top of the Pixhawk toolchain, however, we speed up the process by developing our models in Simulink. A proper build system with a Simulink supported target does exist at the time of this writing for deploying code directly to the controller. The process is used as a workaround method to custom writing control algorithms on the Pixhawk controller.

In [2], we note the process used to develop wrapper code in Simulink for the Pixhawk. We then employ a process similar to [21] to develop our fuzzy control algorithm. The limitation imposed by available memory space, and the lack of useful altitude feedback, permits only the attitude controller to be implemented on the Pixhawk controller.

The wrapper allows correct port mapping between the controller and the tags in Simulink. The control algorithm has two main parts to it.

- Rate Controller– Type: Proportional-Integral-Derivative (PID), this controller operates as that 'fast inner loop' and operates 250Hz. This rate controller consists of three-parallel PID controllers. Using the ANFIS technique, it would be possible to make this controller as a fuzzy controller, however, we observed that the processing overhead of executing the fuzzy controller at over 250Hz significantly reduces system performance.

- Attitude Controller– Type: ANFIS derived fuzzy controller, Takagi Sugeno, this controller operates as the 'outer loop', its purpose is to maintain a set attitude for the vehicle. We illustrate the performance of this controller further in this section.

Input, output signals for the fuzzy attitude controllers are listed below,

- Control inputs for roll controller– (θ_r, ch1).
- Control inputs for pitch controller– (θ_p,ch2).
- Controller output– For both controllers, normalized roll and pitch request.

We mimic the strategy we developed earlier in order to create the training data for this attitude controller. We setup a simulation of the dynamic system, sensors and control system with the basic open-source PD controller available from [11]. In order to create this proof of concept, we set one input to be the roll request channel (channel 1) on the remote transmitter, and the second input to the be the current roll of the vehicle. A similar process is developed for controlling the pitch. The inputs to the pitch controller are the pitch request channel (channel 2) and the pitch obtained from the IMU. We follow the *North, East, Down* (NED) frame as before. The roll and pitch are provided by the 6DoF accelerometer within the Pixhawk controller.

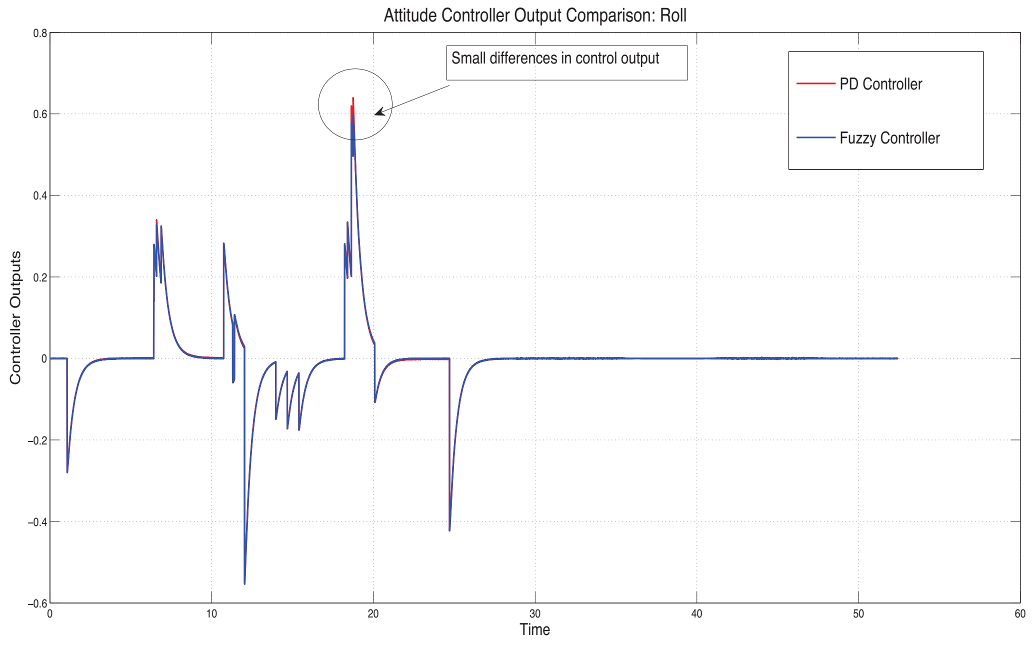

Figure 6.4: Pixhawk: Roll controller comparison

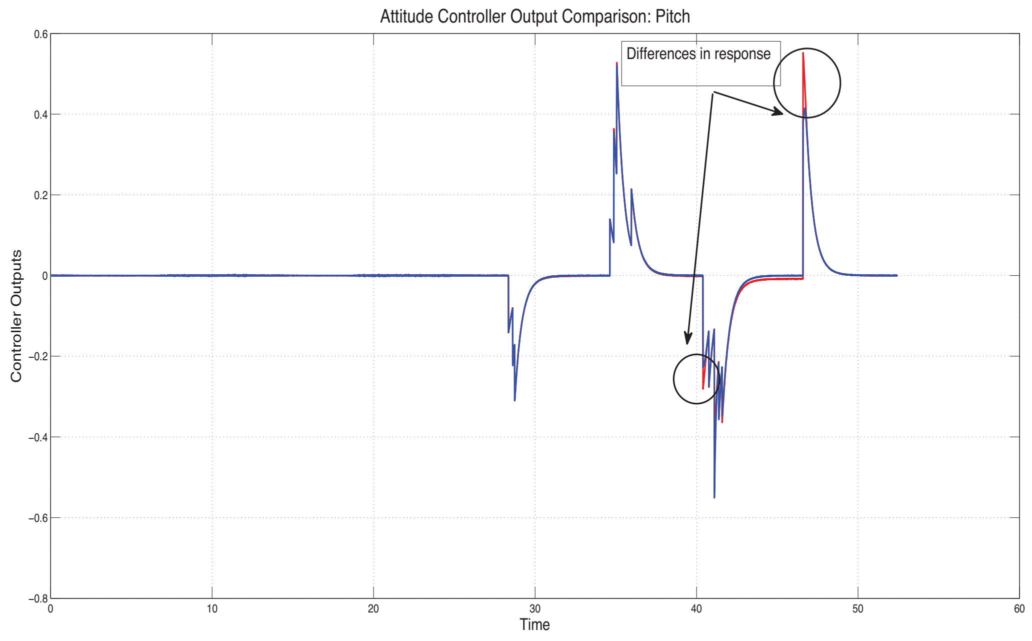

Figure 6.5: Pixhawk: Pitch controller comparison

6.3.2.3 Results: Pixhawk

Results are indicated in figure Fig. 6.5 and Fig 6.4. These figures indicate the differences in the attitude request from each of the parallel controllers. The comparison here is between the PD controller found in the open-source variant of the control algorithm found in [11]. The attitude controller seen here sends commands to the rate controller. The rate controller is connected to the control splitting block, similar to the model developed earlier. The only difference being that the output of this control splitting block is now sent a PWM signal generation mixer block on-board the Pixhawk controller.

The results indicate the satisfactory performance of the controller.

Deviations for the desired response are denoted. This data is in response to manually manipulating the radio channels for roll and pitch on the Futaba transmitter. The vehicle is first made to roll in the positive and negative direction, then followed by a positive pitch and a negative pitch.

A similar method can be followed for creating a rate-controller and then a rate-attitude controller processing multiple fuzzy controllers. Evaluating these at 250Hz caused issues. This is one the bottlenecks that we should see go away with improving controller hardware, the hardware already exists (for example: Gumstix Overo), however, an initiative like Arduino Tre[25] (combining a fast 1GHz processor with the Arduino eco-system) is necessary to successfully implement memory intensive fuzzy controllers.

6.3.3 Alternative implementation method

Another method proposed here is the implementation of the fuzzy controller as a look-up table. It could be argued that in essence, post the tuning process, the fuzzy controller behaves like a look-up table. Some literature survey leads has some conclusive evidence that this will work[26][27]. A post survey analysis leads to the conclusion that a non-linear mapping is created between the input space and the output space, this seems to be a viable solution if implemented correctly. Our results of trying to implement the look-up table based fuzzy controller on the Pixhawk lead us to some interesting results. Due to the complexity of a look-up table, the implementation becomes more memory intensive, which slows the controller down. In our experiments, we saw that the ARM Cortex M4 processor is able to process fuzzy controllers well. We thus decided not to implement the controller using multiple look-up tables. However, this method is worth mentioning. Cases that have a weaker processor, but more memory, might benefit from it.

References

[1] P. I. Corke, *Robotics, Vision & Control: Fundamental Algorithms in Matlab*. Springer, 2011.

[2] pixhawk.org, “Pixhawk developer homepage.” `http://pixhawk.org/dev/start`.

[3] J.-S. Jang, “Anfis: adaptive-network-based fuzzy inference system,” *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 23, no. 3, pp. 665–685, 1993.

[4] Z. Bo, X. Bin, Z. Yao, and Z. Wei, “Hardware-in-loop simulation testbed for quadrotor aerial vehicles,” in *Control Conference (CCC), 2012 31st Chinese*, pp. 5008–5013, 2012.

[5] A. Team, “Apm 2.5 specifications and board overview.” `http://copter.ardupilot.com/wiki/apm25board_overview/`.

[6] Mavlink, “Mavlink source page.” `https://github.com/mavlink/mavlink`.

[7] "Qgroundcontrol source page." https://github.com/mavlink/qgroundcontrol.

[8] S. Weiss, M. Achtelik, M. Chli, and R. Siegwart, "Versatile distributed pose estimation and sensor self-calibration for an autonomous mav," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 31–38, May 2012.

[9] TKJElectronics, "Tkj-electronics, blog post on kalman filters." http://copter.ardupilot.com/wiki/apm25board_overview/.

[10] P. Zhang, J. Gu, E. Milios, and P. Huynh, "Navigation with imu/gps/digital compass with unscented kalman filter," in *Mechatronics and Automation, 2005 IEEE International Conference*, vol. 3, pp. 1497–1502 Vol. 3, 2005.

[11] pixhawk.org, "Pixhawk toolchain." http://pixhawk.org/dev/toolchain_installation.

[12] TKJElectronics, "Kalman filter library, tkjelectronics." https://github.com/TKJElectronics/KalmanFilter.

[13] G. B. G. Welsh, "An introduction to kalman filter." http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html, July 2006.

[14] A. Polak, "Px4 development kit for simulink." http://polakiumengineering.org/, 2014 Feb.

[15] I. Dikmen, A. Arisoy, and H. Temeltas, “Attitude control of a quadrotor,” in *Recent Advances in Space Technologies, 2009. RAST ’09. 4th International Conference on*, pp. 722–727, 2009.

[16] P. M. Meshram and R. Kanojiya, “Tuning of pid controller using ziegler-nichols method for speed control of dc motor,” in *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on*, pp. 117–122, 2012.

[17] O. Lutfy, S. B. M. Noor, and M. Marhaban, “A genetically trained simplified anfis controller to control nonlinear mimo systems,” in *Electrical, Control and Computer Engineering (INECCE), 2011 International Conference on*, pp. 349–354, 2011.

[18] G. E. M. Mahfouz, M. Ashry, “Design and control of quad-rotor helicopters based on adaptive neuro-fuzzy inference system,” *International Journal of Engineering Research and Technology*, vol. 2, December 2013.

[19] A. Al-Hmouz, J. Shen, R. Al-Hmouz, and J. Yan, “Modeling and simulation of an adaptive neuro-fuzzy inference system (anfis) for mobile learning,” *Learning Technologies, IEEE Transactions on*, vol. 5, no. 3, pp. 226–237, 2012.

[20] R. Toulson, “Advanced rapid prototyping in small research projects with matlab/simulink,” in *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pp. 1–7, 2008.

[21] S.-H. Seo, S.-W. Lee, S.-H. Hwang, and J. W. Jeon, "Development of platform for rapid control prototyping technique," in *SICE-ICASE, 2006. International Joint Conference*, pp. 4431–4435, 2006.

[22] P. Wallich, "Arducopter parenting," *Spectrum, IEEE*, vol. 49, no. 12, pp. 26–28, 2012.

[23] "Nuttx." `http://nuttx.org/`.

[24] "Nuttx source page." `https://github.com/PX4/NuttX`.

[25] Arduino, "Arduino tre." `http://arduino.cc/en/Main/ArduinoBoardTre#.Uysc7dxWqzA`.

[26] Y. Jiang, X. Zhang, T. Zou, and G. Cao, "A novel 3-d fuzzy logic controller design using table look-up scheme," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 4, pp. V4–388–V4–393, Oct 2010.

[27] P. Albertos, M. Olivares, and A. Sala, "Fuzzy logic based look-up table controller with generalization," in *American Control Conference, 2000. Proceedings of the 2000*, vol. 3, pp. 1949–1953 vol.3, 2000.

[28] TKJElectronics, "Arduino example sketch, kalman filter." `https://github.com/TKJElectronics/Example-Sketch-for-IMU-including-Kalman-filter/blob/master/IMU6DOF/MPU6050/MPU6050.ino`.

[29] A. Visioli, "Tuning of pid controllers with fuzzy logic," *Control Theory and Applications, IEE Proceedings* -, vol. 148, no. 1, pp. 1–8, 2001.

[30] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor," *Robotics Automation Magazine, IEEE*, vol. 19, no. 3, pp. 20–32, 2012.

[31] M. Santos, V. López, and F. Morata, "Intelligent fuzzy controller of a quadrotor," in *Intelligent Systems and Knowledge Engineering (ISKE), 2010 International Conference on*, pp. 141–146, 2010.

[32] B. Yu, X. Dong, Z. Shi, and Y. Zhong, "Formation control for quadrotor swarm systems: Algorithms and experiments," in *Control Conference (CCC), 2013 32nd Chinese*, pp. 7099–7104, 2013.

Appendix A

List of signals

Signal Name	Description	Units
ch1	Control Input - Roll	normalized, unitless
ch2	Control Input - Pitch	normalized, unitless
ch3	Control Input - Thrust	normalized, unitless
ch4	Control Input - Yaw	normalized, unitless
ch5	Control Input- Mode Selector	normalized, unitless
roll	Attitude - Roll	degrees
pitch	Attitude - Pitch	degrees

This list is continued on the next page.

yaw	Attitude - Yaw	degrees
p	Angular Velocity	deg/sec
q	Angular Velocity	deg/sec
r	Angular Velocity	deg/sec
phi	Euler Angle	rad
theta	Euler Angle	rad
psi	Euler Angle	rad