AQuaH – Autonomous Quad-Propeller Helicopter

Jonathan Mejias

Rydon Samaroo

Sayyid Khan

Chapter 1 - Introduction

Abstract

The purpose of this report is to show the methods taken to achieve hover from our Quad-Copter and to maintain stability while doing so.

Overview

This project is being continued from ED2. The first time around we approached the design with the standpoint of a hobbyist. We knew some of the engineering perspectives that needed to be taken into account, but overall we used the trial and error method. This time around we are required to work from a top down perspective meaning we have to solve this like engineers and put everything down on paper first. Only after this is completed can we start to work on the actual Quad-Copter.

Chapter 2 - Components

Mystery 5000 KV Outrunner Brushless Motors (x4)

- No. Of cells: 3 (11.1V) Li-Poly
- RPM/V (kv): 5000 RPM/V
- No load current: 2.8A
- Loaded current: 25A
- Loaded speed: 55300 RPM
- Weight: 37g
- Pull force: 600g
- Recomended prop without gearbox: 6 inches
- Max eff. 96%
- Total length (motor & shaft): 40.1mm
- Shaft diameter: 2.26mm
- Shaft length: 13.9mm
- Recommended input voltage: 6-18 Volts

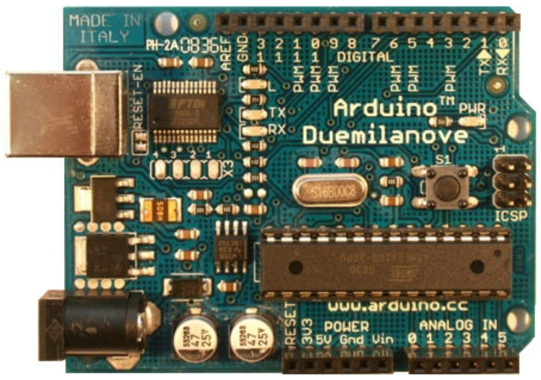
Mystery 30A ESC Burshless Motor Speed Controller (x4)

- Constant Current: 30A
- Input Voltage: 5.6V - 16.8V
- Max Current: 40A
- Low Voltage Cutoff: Shut Off / Power Reduce / Ignore
- Inside Impedance: 0.0044
- Reversion: Yes
- Size: 57mm X 25mm X 8mm
- Weight: 18g (Net Weight)
- Low Resistance
- High rate PWM
- Max 16 Cells (With BEC Disabled)
- User programmable brake
- Low voltage auto setting based on battery
- Self adjusting Throttle Range
- Soft start ramp up
- Auto motor cutoff with reset
- Turn motor in forward or reverse direction
- Low torque start
- PPm Input

Pulse position modulation (PPM) is a modulation technique that uses pulses that are of uniform height and width, but displaced in time from some base position according to the amplitude of the signal at the instant of sampling. In short, the period of the PPM signal varies from 1ms to 2ms. The closer the PPM signal is to 2ms the faster the motor would go. For testing purposes we used the pulse generator in the lab. Once we obtained all the data we could program our microcontroller to achieve PPM signals as well.

Arduino Duemilanove Microcontroller

- Microcontroller: ATmega328
- Operating Voltage: 5V
- Input Voltage (recommended): 7-12V
- Input Voltage (limits): 6-20V
- Digital I/O Pins: 14 (of which 6 provide PWM output)
- Analog Input Pins: 6
- DC Current per I/O Pin: 40 mA
- DC Current for 3.3V Pin: 50 mA
- Flash Memory: 32 KB (ATmega328) of which 2 KB used by bootloader
- SRAM: 2 KB (ATmega328)
- EEPROM: 1 KB (ATmega328)
- Clock Speed: 16 MHz



Memsic 2125 Dual Axis Accelerometer

- Measures ±3 g on each axis
- Simple pulse output of g-force for each axis
- Convenient 6-pin 0.1" spacing DIP module
- Analog output fo temperature (Tout pin)
- Low current at 3.3 or 5 V operation: less than 4 mA at 5 VDC
- Dual-axis tilt sensing for autonomous robotics applications
- Single-axis rotational position sensing
- Movement/Lack-of-movement sensing for alarm systems
- R/C hobby projects such as autopilots
- Power requirements: +3.3 to +5 VDC
- Communication: TTL/CMOS compatible 100 Hz PWM output signal with duty cycle proportional to acceleration
- Dimensions: 0.42 x 0.42 x 0.45 in (10.7 x 10.7 x 11.8 mm)
- Operating temp range: 32 to +158 °F (0 to +70 °C)

A Pulse Width Modulated (PWM) signal is outputted on two pins, each corresponding to a axis. When a duty cycle of 50% is outputted that means the axis is leveled or 0 deg. This is the same for both pins. However when the accelerometer is rotated 90 deg then the PWM outputted is ±12.5% from the original 50%, meaning at -90 deg the output is 37.5% and at 90 deg the output is 62.5%.

Chapter 3 - Collecting Data

3.1 Calculating Lift

The first approach we took was figuring out the necessary lift equations for our helicopter which will indicate the amount of force needed to attain lift. We needed to find out how much our helicopter weighed altogether. From there we could estimate the amount of force needed to give us the necessary lift. We were referred to a Dr. Gaonkar from the Mechanical Engineering Department. Dr. Gaonkar specializes in helicopter dynamics and he referred us to a book on helicopter theory, in which lift equations were given. These equations involve complex statics and dynamics.

$L=12\rho V2SrefCL$
$CL=CT=T\sigma\rho\pi R2(\Omega R)2$
$\sigma=w*b\pi R$
L=Lift force $N=kg*ms2$
ρ=Air density $kgm3$
V=Rotor RPM ms
$Sref$=Rotor reference area (m2)

$CL=CT$=Coefficient of lift

T=Thrust $N=kg*ms2$

R=Rotor radius (m)

Ω=Angular velocity rads

σ=solidity

w=blade width (m)

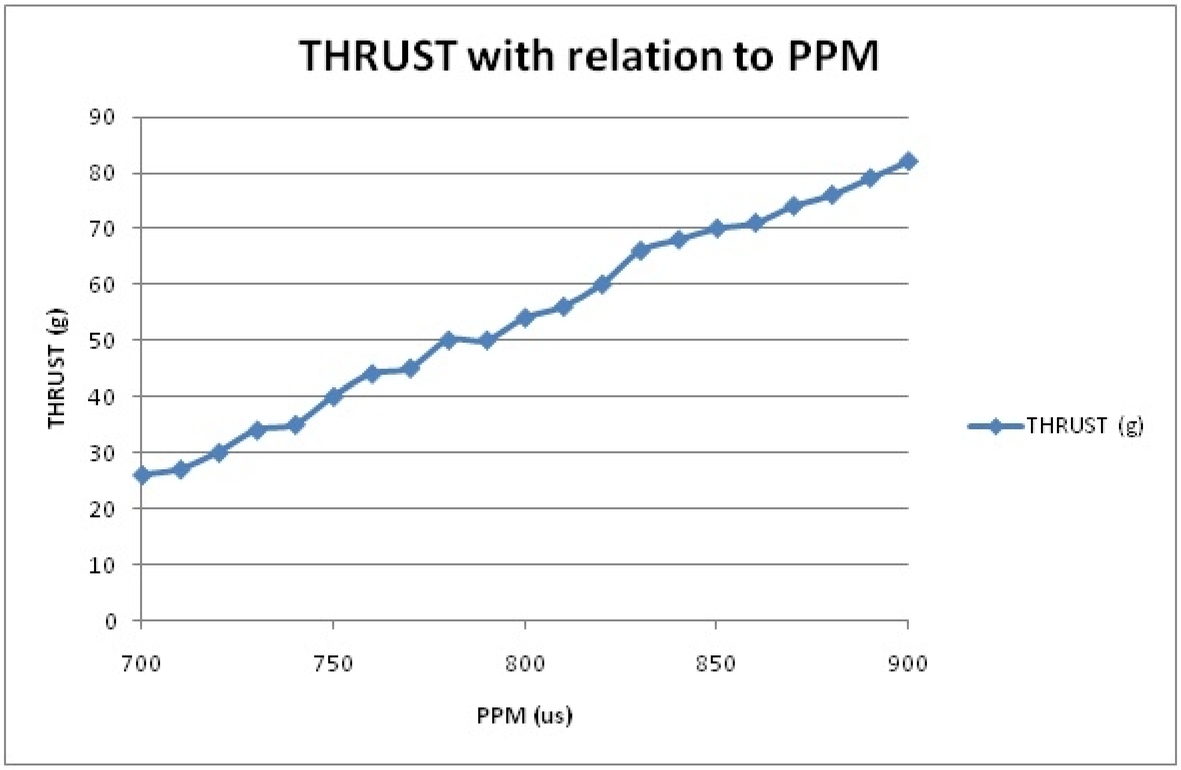
b=# of blades

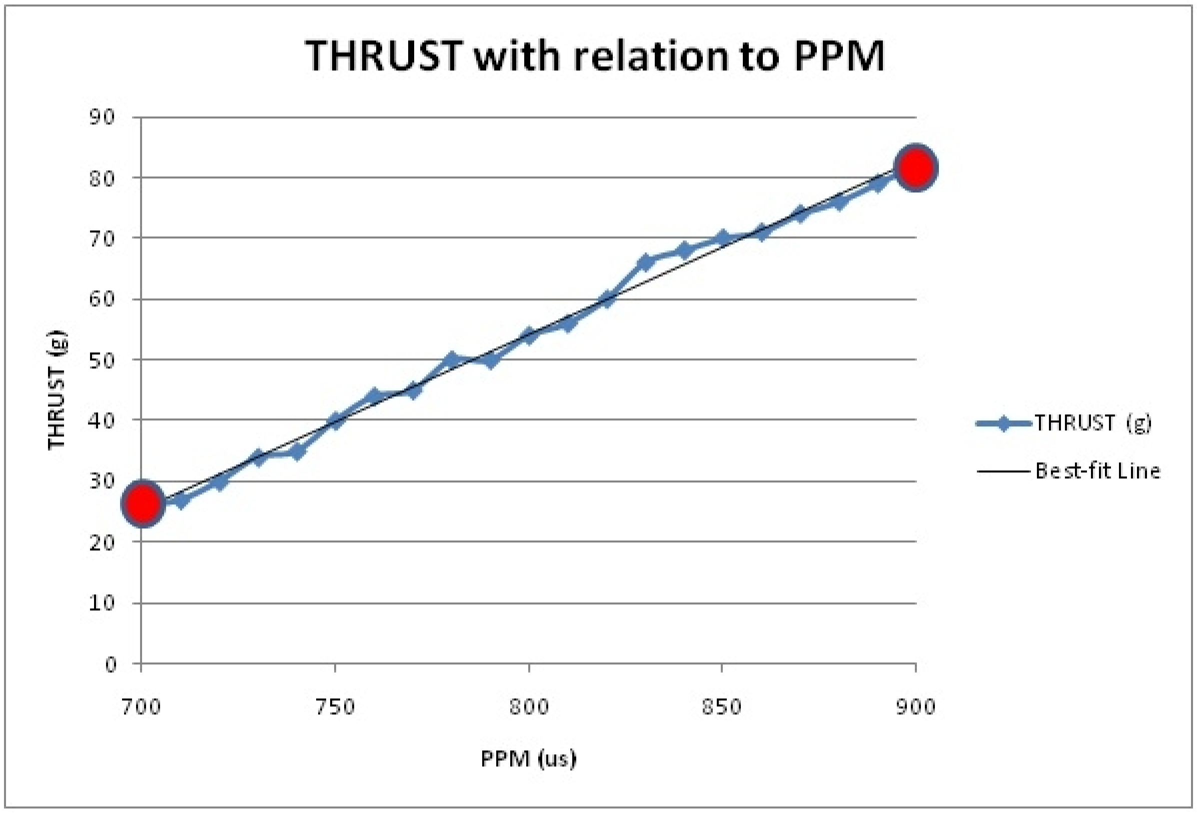
From the equations given in Table 1 above, almost all of the variables are constant. The most difficult variable to calculate is the coefficient of lift. After studying the equations extensively we learned that the lift coefficient is not easily calculated. This coefficient is normally calculated through experiments using equipment that is not accessible to us.

We then decided to go to plan B which was creating a look up table. This will be done by using a tachometer to measure RPM's from the propellers and the air mass created by the rotors. This air mass will be measured with a gram scale which needs to be as close as possible to the rotors. From here we will be able to create graphs and equations using points taken from our experiments. The equations were generated by using the point slope form equation and the slope formula. Not only will be able to generate equations that relate RPM's to thrust, we can also generate equations that relate PPM to thrust as well, which will be extremely helpful when programming the outputs of the microcontroller.

PPM	**RPMx2**	**RPM**	**THRUST**	**FORCE**	**CURRENT**
700.00	7870.00	3935.00	26.00	0.25506	2.34
710.00	8208.00	4104.00	27.00	0.26487	2.54

720.00	8660.00	4330.00	30.00	0.2943	2.73
730.00	8970.00	4485.00	34.00	0.33354	2.93
740.00	9350.00	4675.00	35.00	0.34335	3.24
750.00	9680.00	4840.00	40.00	0.3924	3.37
760.00	9950.00	4975.00	44.00	0.43164	3.62
770.00	10250.00	5125.00	45.00	0.44145	4.01
780.00	10750.00	5375.00	50.00	0.4905	4.23
790.00	10950.00	5475.00	50.00	0.4905	4.47
800.00	11250.00	5625.00	54.00	0.52974	4.69
810.00	11550.00	5775.00	56.00	0.54936	4.93
820.00	11800.00	5900.00	60.00	0.5886	5.17
830.00	12150.00	6075.00	66.00	0.64746	5.57
840.00	12550.00	6275.00	68.00	0.66708	5.85
850.00	12750.00	6375.00	70.00	0.6867	6.08
860.00	12900.00	6450.00	71.00	0.69651	6.31
870.00	13150.00	6575.00	74.00	0.72594	6.52
880.00	13650.00	6825.00	76.00	0.74556	6.88
890.00	13700.00	6850.00	79.00	0.77499	6.89
900.00	14000.00	7000.00	82.00	0.80442	7.20





In order to bring this into an equation that we can use later on we must first convert grams to Newton's. This is down by changing the grams to kilogram (multiplying by 0.001) and then multiplying the kilograms by gravity (9.8 m/s^2). After this conversion we use point slope form from basic algebra to get our lift equation.

Lift Equation

3.2 Body Dynamics

Now that we have modeled lift into an equation based on PPm we needed to come up with dynamic equations for our Quad-Copter. We began to work closely with an undergraduate Mechanical Engineering student (Dan Rodriguez). He was able to provide the equations needed that will model the Quad-Copter dynamics. These equations will be used on our model and algorithm so that we may predict the behavior of the system and correct it in due time if needed. Given below are the dynamic equations used in our project.

1.$F_x=m*a*x \Rightarrow -F_1\sin\Theta-F_2\sin\Theta=m*\ddot{x}$, m=mass, a=accelration

2.$F_y=m*a*y \Rightarrow F_1\cos\Theta+F_2\cos\Theta-w=m*\ddot{y}$, $w=mg$

3.$F_2*r-F_1*r=J_o*\ddot{\theta}$, r=radius, Jo=polar moment of inertia

With the event of determining the body dynamic equations we now have all the necessary equation to solve the control aspect of our Quad-Copter.

Chapter 4 - Controls

When designing our quad-copter we had a simple goal in mind, which was to make the quadcopter hover on its own without any human help, this is why we are calling it autonomous quadcopter. Although this goal seemed simple enough it actually took a lot of work to understand and apply the theory of control systems to our quadcopter. It should also be noted that this was the main aspect of the project and without the controls working properly we could not have attained our goal of stable flight. In addition to this, once we figured out the control systems solution it was also necessary to implement it in programming our arduino micro controller.

The schematics and block diagrams in this chapter are highly detailed because in order to develop a control system using control theory we need to have all the equations and relations that govern the way are Quad-copter will respond to different stimuli and how our sensor can will read how it is affected. Once all of these equations are determine we can use computer programs to find what's the best theoretical values for our PID controller.

4.1 - Understanding PID

Figure 4.1 - Block diagram of PID controller (Wikipedia.org)

The type of controller we pick to stabilized and the most of the work was the PID controller. This stand for Proportional-Integrator-Derivative controller and even though the theory behind this requires plenty of complicated math, the actual concept and what it tries to achieve is straight forward. An explanation of PID controllers in a nutshell goes something like this, the proportional part tries to bring a measure values closer to a desired value, the integrator part eliminates any residual error from the proportional part as well as look at past trends, and the derivative speeds out the process in addition to look at future trends. The concept will become more clear by looking at the example below, this example will only covers the concepts of a simple proportional controller with a minimal amount of math but once this is understood you can just think of a PID controller as a upgrade to a Proportional controller.

Example Summary:

The goal of this system is to keep water in the bucket at a constant temperature which is determined by the user. The main components of this system is a heating coil which changes the water temperature, a thermometer which measures the water temperature, and a micro-controller which process the information and gives instruction. The systems works by comparing the desired temperature (DT) to the actual temperature (AT). To demonstrate how the system works and explain the concept of a P Controller we will look at three scenario, DT>AT, DT=AT, and DT<AT. (*Note: DT-AT=Error*)

Scenario 1: DT>AT

If the desired temperature is greater than the actual temperature then our error will be positive, this error value is feed into the P controller and multiplied by a number K. The product of Error*K gives us a values for Pout which corresponds to certain voltage and this voltage is sent to the heating coil.

Numerically:

Let K = 2, DT=80C, AT= 60C, Voltage=Pout/10

DT-AT=Error   80-60=20

*K*Error=Pout   2*20=40*

Pout/10=Voltage   40/10=4V

4 volts is now sent to the heating coil which begin to raise the actual temperature and bring it closer to the desired temperature. From this we can also see that as the Error becomes greater, Pout and ultimately the voltage sent to the coil increases <u>proportionally</u>.

Scenario 2: DT=AT

If both the desired and actual temperature are the same then the Error is now zero, which means we want nothing to be done since everything is perfect.

Numerically:

Let K = 2, DT=80C, AT= 80C, Voltage=Pout/10

DT-AT=Error   80-80=0

*K*Error=Pout   2*0=0*

Pout/10=Voltage   0/10=0V

The coil will be turned off and will remain off until the system comes out of equilibrium.

Scenario 3: DT<AT

If the actual temperature become greater than our desired temperature then we can really do anything but turn the heating coil off and wait because we only have a heating element and no cooling element in our system. Also if you think about this numerically we will get negative values, and this is no good to us, so what we will have to end up doing is to put a clause in the program of the micro-controller to turn off the heating coil if we get a negative number for our Error.

After reading through these three scenarios it should become clear how the P controller work and how they can be use. It should also be noted that the values and calculation can and will be continuously updated many times a second which will make the system respond to any outside effects like a change in room temperature quickly.

4.2 Quad-Copter Specifics

Figure 4.2

Figure 4.3

This is the block diagram we have determined for our Quad-Copter. At this point we have abstracted it out to just the tasks that needs to be done, however in the following sections all these tasks will be replaced by mathematical equation and the block diagram will be a mathematical model that will look similar those who have taking a controls course. The remaining part of this section will be devoted to explaining what each of these blocks on the block diagram does, and why they are needed in the block diagram in the first place.

Before we even get into the block diagram, the first thing to explain would be the "System of X axis" and "System of Y axis" titles. As shown in figure 4.2 what we basically have is two perpendicular line and on the ends of the lines we have some type of force. Although these two lines are connected the angles created by the differences in forces are independent of each other, meaning a change in angle on the X axis won't change the angle on the Y axis. By noticing this relation we can break the system into two simpler block diagrams as opposed to one complex block diagram. *It should be noted that the angle are measure with respect to the Z axis.*

Block **Summary**

<u>Accelerometer</u>

Restating what we said previously about the Memsic 2125, what the Memsic does is output a PWM signal that is proportional to the G-forces experienced by it. So it terms of interfacing this with the arduino we will measure it as a pulse-in and this time high (TH) reading will give us a value relating to the G-force. A (TH) reading of 5000 will be considered leveled.

<u>Adder and Flow Director</u>

Since the goal of this project is to achieve stable flight we need to compare the accelerometer value of stable flight to that of the actual accelerometer value. After we find this difference, e(t), we need to determine what motor this will go to because the Memsic 2125 outputs both values higher and lower than 5000 (TH). What ends up happening is negative values will correspond to one motor and positive values will correspond to the other.

<u>Controller</u>

After the path is determined the e(t) value will go into the PID controller, which we have talked about slightly already and we will go into more depth later. The value that will come out of the

PID controller will be based on e(t) and the value of e(t) is based on TH number, so we need to have a block that can relate the TH based output of the PID controller to PPM. The reason we have to add this block to the controller is most importantly because our motors use a PPM signal as its input and the equation that will govern this block will be affected by both e(t) coming out and the expressions that make up the PID controller.

Lift Model

Once we find out the values of the PPM signal that needs to be sent to the motor, we have to determine how this PPM will relate to RPMs and how the RPMs will relate to force. The way how these relation were determined was cover in the a previous section.

Body Dynamics

After we find the forces exerted by each motor, these two paths will converge into a single block. This block will convert this two forces into a single angle, the angle of the frame with respect to the Z axis. Once this is found, in order to simulate our accelerometer properly we need to convert this angel in to a G-force. This G-force is then sent back into the accelerometer and the loop will continue.

4.3 - Mathematical Equation

Each one of the above blocks can be described by an equation, these equations were either found through experimentation (Chapter 3), given in the data sheets, basic physics equations or common sense. The only blocks that can't be accounted for is the PID controller and the TH to PPM. The reason why we don't have to PID controller block is because that what we are ultimately trying to find and the reason why we don’t have to TH to PPM block is because that is directly dependent on the value of the PID controller.

Time Domain	Data Sheet
Laplace Domain	
Time Domain	Common Sense
Laplace Domain	
N/A	
Time Domain	Common Sense
Go to Path 1	
Go to Path 2	
Laplace Domain	
N/A	
Time Domain	Experimentation
Laplace Domain	
Time Domain	Physic Equation

(Sum of Forces)

Laplace Domain

Time Domain

Experimentation

Laplace Domain

.

4.4 Mathematical Model

Figure 4.4 - System model on X axis

Figure 4.5 - System model on Y axis

Above we have shown both the layout of the motors and our overall system for both the x and y axis. After speaking to Dr. Raviv (Control Systems Professor) from the Department of Electrical Engineering about our project and what our transfer function would look like. He had told us that normally a complex system such as the one we are working with are usually solved by a team of Control Engineers. He advised us that the best thing we could do to solve our problem was to simplify the system as much as possible, meaning that we should neglect all disturbances coming into our system and that we should solve for one motor at a time. Below we will show this implementation.

Figure .5 – System model for x-axis (neglecting disturbances)

Figure 4.6 – System model for y-axis (neglecting disturbances)

Figure 4.7 – Simplest model of single rotor

4.5 P and PD Controller

As stated earlier on in this chapter when we decide to use a controller we also need to have a block that will convert the TH based value of the controller to a PPm based value. Here we are also going to use the point slope method, and map the values to each other.

P controller:

TH to PPm

Open Loop Transfer Function (Multiply out)

Closed Loop Transfer Function

Let the numerator equal to 1 and plot in root locus in MATLAB

As seen on the root locus above for the P controller our system can never be stable for any values of K. From basic control systems we know that the root locus always begins at the poles (X) and end at a zero (not shown) in this case ± infinity running up and down the jw-axis. We can see that the for small K’s the root locus begins where the X is shown on the plot and for very large K’s the root locus runs up and down the jw-axis. For our system to be stable the root locus needs to fall on the left hand side of the plot. Otherwise, the system will be unstable.

PD controller:

Open Loop Transfer Function (Multiply out)

Closed Loop Transfer Function

Let . , be 1 and neglect 235, now plot the root locus in MATLAB

Show above is the root locus for our PD controller. From basic control systems we can see that the system is always stable for any values of K. From the plot above we observe once again that the root locus begins at the (X) and ends at a zero. In this case one of the pole goes to the zero (o) shown above in blue and the other pole goes to negative infinity on the real axis show in green.

Chapter 5 – PD Algorithm

Theory and research are crucial in order to achieve a top-down design, but without practical application, the project just becomes a series of formulas and numbers. Theory alone isn't enough to claim the desired and expected results – this is where coding and algorithms come into play. The code gives the intangible theory a physical feel, allowing others to see progress and results. Before delving into the details of the code, it's important to understand the overall structure of the program. The flow chart below shows how the basic flow of the program:

For 5.1 though 5.4 the code snippets described reflects the program for only one axis (X-axis) since the demonstration is done using only two motors. This is why the code shows only Rotor1 and Rotor2. When the full code is posted is 5.5, all 4 motors will be accounted for.

5.1 Warm Up

The code below is used for the Warm Up sequence:

```
// Warmup routine from 500 to 850 PPM

  Rotor1.write(500);

  Rotor2.write(500);

  delay(3000);

  Rotor1.write(550);

  Rotor2.write(550);

  delay(3000);


  while (PPMCounter < 850)

  {

    PPMCounter = PPMCounter + 5;

    Rotor1.write(PPMCounter);

    Rotor2.write(PPMCounter);
    delay(200);

  }
```

The warm up sequence is the first thing program runs. The purpose of the warm up is to slowly raise the speed of the motors to prevent them from suddenly going to full speed and causing further instability. The warm up sequence pulses 500 us (microseconds) to both motors for 3 seconds and then pulses 550 us to both motors for 3 seconds. The motors are armed at a PPM signal of 550 us. The, starting at 550 us, the PPM signal is incremented by 5 until the motors reach a value of 850 us – then the main loop begins.

5.2 Read Accelerometer and Calculate Error

The code below is used to read the accelerometer and calculate the error:

```
// Measures length of pulse in micro-seconds

  // Typical range is from 3700 to 6250 us

  PulseX = pulseIn(X, HIGH);

  PulseY = pulseIn(Y, HIGH);



  // Convert PULSE to Duty Cycle value

  DutyX = PulseX/100;

  DutyY = PulseY/100;
  // Calculate error of both axis

  // 50 = level, reference point

  CurrentErrorX = 50 - DutyX;

  CurrentErrorY = 50 - DutyY;
```

Since the accelerometer is a digital device, it gives a digital output. The output is a pulse width modulated signal that can be measured by finding the length of the pulse in microseconds. Typically, the length of the pulse will range from 4000 us to 6000 us (with a variance of plus or minus 250 us). When the accelerometer is level, it will generally output a pulse of 5000 us. Depending on the direction of the tilt, the accelerometer will output a pulse either below or greater than 5000 us. After measuring the pulse, it is divided by 100 to gain a variable referred to as the "Duty". This Duty variable is used to calculate the error by subtracting the Duty from 50 (the reference of a level, stable position). The error is known as the "CurrentError" and can either be negative or positive.

5.3 Applying P-algorithm to the appropriate motor

The code below is used to apply the P-algorithm to the appropriate motor:

```
// Decide which rotor to adjust

  // Rotor1 on X-axis
```

```
if (CurrentErrorX < 0)

{

  Abs = -(CurrentErrorX);

  Pout1 = Kp*Abs;


  PPM1 = ((4/Kp)*Pout1)+850;


  Rotor1.write(PPM1);

}

// Rotor2 on X-axis

if (CurrentErrorX > 0)

{

  Pout2 = Kp*CurrentErrorX;


  PPM2 = ((4/Kp)*Pout2)+865;


  Rotor2.write(PPM2);

}

if (CurrentErrorX == 0)

{

  Rotor1.write(PPM1);

  Rotor2.write(PPM2);

}
```

Since the direction of the tilt gives either a negative or a positive Current Error value, the CurentError is what is used to decide which motor to apply the P-algorithm to. In this case, the algorithm is only being run on one axis, so a negative value will represent one motor while a positive value represents the next. Once it is decided which motor will be adjusted, the P-algorithm takes effect. The algorithm gives a Pout value which is calculated by multiplying a constant Kp by the CurrentError. Then that Pout value is used in a formula (derived in earlier chapters) to find a PPM value. That PPM value is the final value used to adjust the motor(s).

5.4 Applying PD-algorithm to the appropriate motor

The code below is used to apply the PD-algorithm to the appropriate motor:

```
// Decide which rotor to adjust

   // Rotor1 on X-axis

   if (CurrentErrorX < 0)

   {

     Abs = -(CurrentErrorX);

     PDout1 = Kp*Abs + Kd*((CurrentErrorX-PreviousErrorX)/Tau);

     PPM1 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout1 + 850;

     Rotor1.write(PPM1);

   }

   // Rotor2 on X-axis

   if (CurrentErrorX > 0)

   {

     PDout2 = Kp*CurrentErrorX + Kd*((CurrentErrorX-PreviousErrorX)/Tau);

     PPM2 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout2 + 892;

     Rotor2.write(PPM2);

   }
   if (CurrentErrorX == 0)

   {

     Rotor1.write(PPM1);

     Rotor2.write(PPM2);

   }
```

The only changes made from the P-algorithm to the PD-algorithm were done in the algorithm section of the code, the remaining structure of the program was unchanged. In the PD-algorithm, a new variable PDout is defined. PDout is calculated by combining the value of Pout and Dout to find a more fine tuned result. Furthermore, the PPM value is calculated by using this PDout value in a formula (derived in previous chapters).

5.5 Full Code

The code below is used to appropriately apply the PD-algorithm to all 4 motors:

```
// PID Control Loop for AUTONOMOUS QUAD-PROPELLER HELICOPTER

// Written by Rydon Jake Samaroo

// Additional team members: Sayyid Khan, Jonathan Mejias

// Latest Revision on 11.27.2010
// Creating Servo objects
#include <Servo.h>
Servo Rotor1;
Servo Rotor2;
Servo Rotor3;
Servo Rotor4;

// Pins for X and Y inputs from ACC
const int X = 12;
const int Y = 13;

int PPMCounter = 550;
int PPMCounter1 = 580;

// Variables for Duty Cycle
double DutyX;
double DutyY;

// Pins for four PPM outputs to motors
const int PID1 = 3;
const int PID2 = 6;
const int PID3 = 7;
const int PID4 = 8;

// Variables for error calculations
double PulseX, PulseY;
double CurrentErrorX, CurrentErrorY;
double PPM1, PPM2, PPM3, PPM4, Abs;
```

```
// Variables for PID calculations
double Kp = 20.0;
double Pout1, Pout2, Pout3, Pout4, PDout1, PDout2, PDout3, PDout4;
int DELAY = 250;
double Tau = DELAY*1000;
double Kd = 20000;
double PreviousErrorX, PreviousErrorY = 0;

void setup()
{
  // Baud-rate for serial monitor
  Serial.begin(9600);

  // Attach motors to corect pins
  Rotor1.attach(3);
  Rotor2.attach(6);

  // Set accelerometer as inputs
  pinMode(X, INPUT);
  pinMode(Y, INPUT);

  // Warmup routine from 500 to 750 PPM
  Rotor2.write(500);
  Rotor1.write(500);
  Rotor3.write(500);
  Rotor4.write(500);
  delay(3000);
  Rotor2.write(550);

  Rotor1.write(550);

  Rotor3.write(550);

  Rotor4.write(550);

  delay(3000);


  while (PPMCounter < 850)

  {

    PPMCounter = PPMCounter + 5;

    PPMCounter1 = PPMCounter1 + 5;

    Rotor1.write(PPMCounter);

    Rotor2.write(PPMCounter1);

    Rotor3.write(PPMCounter);

    Rotor4.write(PPMCounter);

    delay(250);

  }

}


void loop()

{

  // Measures length of pulse in micro-seconds

  // Typical range is from 3700 to 6250 us

  PulseX = pulseIn(X, HIGH);

  PulseY = pulseIn(Y, HIGH);


  // Convert PULSE to Duty Cycle value

  DutyX = PulseX/100;

  DutyY = PulseY/100;
```

```
// Calculate error of both axis
// 50 = level, reference point
CurrentErrorX = 50 - DutyX;
CurrentErrorY = 50 - DutyY;

// Decide which rotor to adjust
// Rotor1 on X-axis
if (CurrentErrorX < 0)
{
  Abs = -(CurrentErrorX);
  PDout1 = Kp*Abs + Kd*((CurrentErrorX-PreviousErrorX)/Tau);
  PPM1 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout1 + 850;
  Rotor1.write(PPM1);
}
// Rotor2 on X-axis
if (CurrentErrorX > 0)
{
  PDout2 = Kp*CurrentErrorX + Kd*((CurrentErrorX-PreviousErrorX)/Tau);
  PPM2 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout2 + 892;
  Rotor2.write(PPM2);
}
// Rotor3 on Y-axis
if (CurrentErrorY < 0)
{
  Abs = -(CurrentErrorY);
  PDout3 = Kp*Abs + Kd*((CurrentErrorY-PreviousErrorY)/Tau);
  PPM3 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout3 + 850;
  Rotor3.write(PPM3);

}

// Rotor4 on Y-axis

if (CurrentErrorY > 0)

{

  PDout4 = Kp*CurrentErrorY + Kd*((CurrentErrorY-PreviousErrorY)/Tau);

  PPM4 = ((8*Tau)/(25*Kp*Tau+25*Kd))*PDout4 + 892;

  Rotor4.write(PPM4);

}

if (CurrentErrorX == 0)

{

  Rotor1.write(PPM1);

  Rotor2.write(PPM2);

}


Serial.print(DutyX);

Serial.print("\t");

Serial.print(CurrentErrorX);

Serial.print("\t");


// Sets previous error to be used in next cycle

PreviousErrorX = CurrentErrorX;

CurrentErrorY = PreviousErrorY;


// Prints values to serial monitor

Serial.print(PreviousErrorX);

Serial.print("\t");

Serial.print(PPM1);
```

```
  Serial.print("\t");

  Serial.print(PPM2);

  Serial.println();



  delay(DELAY);

}
```