بسم الله الرحمن الرحيم

Sudan University of Sciences and Technology

Faculty of Engineering

Aeronautical Engineering Department

Implementation of a Quadcopter Control System

A project submitted in partial fulfillment for the requirements of the Degree of B.Sc. (Honor) in aeronautical engineering

By:

Al-Baraa Omer Mohammed Abu Arki

Mohammed Saif El-Deen Ibraheem Idrees

Muzammil Mohammed Muzammil Abbas

Supervised By:

Dr. Khidir Tay Allah Yousif

October, 2017

الآيــــــــــــة

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

اقْرَأْ بِاسْمِ رَبِّكَ الَّذِي خَلَقَ (1) خَلَقَ الْإِنْسَانَ مِنْ عَلَقٍ (2) اقْرَأْ وَرَبُّكَ الْأَكْرَمُ (3) الَّذِي عَلَّمَ بِالْقَلَمِ (4) عَلَّمَ الْإِنْسَانَ مَا لَمْ يَعْلَمْ (5)

صدق الله العظيم
سورة العلق – الأيات 1 الي 5

Abstract

The control system of a quadcopter is the most important part. Control system governing quadcopter stability and control movement by correcting measurement errors and comparing to the desired values achieving pilot desired and safe flight.

This thesis concerned with the implementation of a quadcopter control system that is tested through visual simulation with real physics then hardware capabilities test that describes the capabilities of the hardware mounted on the quadcopter after the construction is over.

This work was divided into two subsections: simulation and construction; the simulation was conducted using Protues circuit simulation software that failed and was replaced by unity 3D due to its limitation to simulate electronic speed controller (ESCs) and Inertial Measurement Unit (IMU) which are essential to simulate the quadcopter system, Unity 3D simulation software provided 3D visual simulation of the quadcopter depending only on the code and no components simulation was need only the mass and drag properties of the frame.

The construction of the quadcopter consisted of choosing a suitable frame to carry the load of the quadcopter that was plastic foam due its light weight and flexibility casted with fiber glass to reinforce to ensure strength. The quadcopter components were mounted on it with distributed load to ensure equilibrium then the transmitter was setup to determine what position of the transmitter sticks belonged to which flight movement and all the ESCs were calibrated to operate at the same speed then the Proportional-Integral-Derivative (PID) controller code was uploaded and operation was successful which gives us the time to add auto leveling.

Auto leveling of the quadcopter were possible by taking the readings of the gyroscope and applying correction when there is no user input received; the Proportional-Integral-Derivative (PID) applies gyroscope correction to stabilize the aircraft which is zero gyroscope orientation in all axes.

التجريد

نظام التحكم في الطائرات بدون طيار يعد اهم جزء منها لانه يحكم استقرارية و حركة الطائرة عن طريق تصحيح اخطاء قياسها ومقارنتها بالقيم المراده لطيران آمن حسب رغبة الطيار.

هذا البحث يهتم بتضمين نظام تحكم للطائرة رباعية المراوح بدون طيار والتي تم اختبارها عن طريق المحاكاة البصرية بقيم وقوي فيزيائية حقيقة محوسبة واختبار مقدرات اجزاءها التي سوف تركب علي الطائرة. قسم البحث الي جزئين جزء المحاكاة وجزء البناء, جزء المحاكاة تم عن طريق برنامج Protues وunity 3D لتوفر محاكاة بصرية للطائرة كما تمت محاكاة الاجزاء الحسية للطائرة عن طريق برنامج يحدد خصائص الطائرة بعد تركيب اجزائها.

تطلب بناء الطائرة وجود هيكل يجمع بين المرونة وخفة الوزن الذي تم التحصل عليه بالجمع بين مادتي الفلين والفايبر جلاس الذي منح الهيكل صلابة الفايبر ومرونة وخفة الفلين. بعد صناعة الهيكل تم تثبيت الاجزاء الالكترونية عليه وتمت تهيئة المرسل لمعرفة اي حركة من حركات الطيران تنتمي الي اي وضع من اواع حركة ذراع المرسل ثم تمت تهيئة متحكمات السرعة لتعمل كلها في نفس الوقت بنفس السرعة ثم تم تحميل برنامج الطيران علي الطائرة مما اعطانا بعض الوقت لاضافة خاصية التوازن الذاتي.

التوازن الذاتي للطائرة يعتمد علي قراءات الgyroscope الذي يقوم بتصحيح حركة الطيران الي وضع الاتزان حينما لا يكون المستخدم يتحكم في الطائرة.

Acknowledgement

Many thanks and appreciation to our supervisor and everyone helped make this thesis see the light of day and to all who dedicated their time, knowledge and resources to grow our knowledge and skills.

Dedication

So much love to our families and friends whom stood by our side when times were hard and those whom were with us every step of our way and those who loved us and believed in us more than others.

Table of Contents

Contents

الآيـــــــــــة I

Abstract II

التجريد III

Acknowledgement IV

Dedication V

Table of Contents VI

List of Figures IX

List of Tables X

Glossary XI

Symbols XIII

Chapter One 1

Introduction 1

1.1 Overview 1

1.2 Aim & Objectives 1

1.2.1 Aim 1

1.2.2 Objectives 1

1.3 Problem Statement 1

1.4 Proposed Solution 2

1.5 Methodology 2

1.6 Thesis Outline 2

Chapter Two 3

Literature Review and Background 3

2.1 Introduction 3

2.2 History of Quadcopter 4

2.3 Dynamic Model 5

2.3.1 Quad-rotor Characteristics 5

2.3.2 System Description 7

2.4 Quadcopter Block Diagram 12

2.4.1 Inertial Measurement Unit (IMU) 12
2.4.2 RF Receiver 13
2.4.3 Brushless DC Motor 13
2.4.4 Electronic Speed Controller (ESC) 13
2.4.5 Flight Controller 14
2.5 Tools 14
2.5.1 Arduino 14
2.5.2 Proteus 8 Labcenter 14
2.5.3 Unity 3D 14
2.5.4 eCalc – xcopterCalc 14
Chapter Three 15
Modeling and Simulation 15
3.1 Mathematical Modeling 15
3.1 PID Controller 17
3.1.1 Theory of operation 18
3.1.2 PID Tuning 19
3.1.3 Classic PID Equations 19
3.2 Simulation 20
3.2.1 Unity 20
3.2.3 eCalc 22
Chapter Four 25
Construction 25
4.1 Overall hardware connection to the microcontroller 25
4.2 Hardware Components 26
4.2.1 The Frame of the Quadcopter 26
4.2.2 The Microcontroller – Arduino Uno 26
4.2.3 Electronic Speed Controllers 27
4.2.4 Inertial Measurement Unit 29
4.2.5 IMU interface with ARDUINO 30
4.2.6 The Battery Pack 31
4.2.7 The Brushless Motors 33

4.2.8 Propellers 35
4.3 Software Implementation 35
4.3.1 Quadcopter Flowchart 35
4.3.2 Transmitter and Receiver 36
4.3.3 Gyroscope 37
4.3.4 ESCs connection to Arduino 39
Chapter Five 41
Results and Discussion 41
Chapter Six 42
Conclusion and Recommendations 42
6.1 Conclusion 42
6.2 Recommendations 42
References 43
Appendix A: eCalc Hardware Analysis I
Appendix B: Quadcopter Simulation Code II
B.1 PID Controller Simulation Code II
B.2 Quadcopter simulation code IV
Appendix C: Quadcopter Code XI

List of Figures

Figure 1: Shows the x and + structure configurations of a quadcopter 3
Figure 2: shows the quadcopter rotorcraft of Bothezat.. 4
Figure 3: shows the Quad-rotor rotorcraft 5
Figure 4: shows throttle control input 6
Figure 5: (a) Pitch (b) Roll (c) Yaw 6
Figure 6: shows the quadcopter in an inertial frame 8
Figure 7: Quadcopter Block Diagram 12
Figure 8: Quadcopter control loop 17
Figure 9: Quadrotor 3D model 20
Figure 10 :Rigid body component attached to the quadrotor 3D model..................................... 21
Figure 11: Quadcopter controller parameters 22
Figure 12: Hardware Specifications passed into eCalc for evaluation 23
Figure 13 :Range Estimation 23
Figure 14: Motor characteristics at full throttle 24
Figure 15: Overall Quadcopter hardware connection 25
Figure 16: Plastic foam frame incase in fiber glass 26
Figure 17: Arduino UNO Microcontroller Board 27
Figure 18: 30A Brushless ESC 28
Figure 19: MPU6050 IMU used in our quadcopter 30
Figure 20: MPU6050 Interface with Arduino 30
Figure 21: 3S LiPo Battery 32
Figure 22: A2212/13T 1000 KV BLDC (Brushless DC Motor) 34

List of Tables

Table 1: Effects of each of controllers Kp, Ki, and Kd on a closed-loop system....................... 19
Table 2 : Specification for 30A Brushless ESC.. 29
Table 3 : LiPo batteries 3S 11.1V 2600MAH 30C packs.. 33
Table 4 : Specifications of A2212 / 920 KV out runner motor 34

Glossary

ADC Analog to digital converter

BEC Battery Eliminator Circuit

BLDC Brushless DC Motor

BT Bluetooth

DAC Digital to Analog Converter DLPF Digital Low Pass Filter

EMF Electro Motive Force

ESCs Electronic Speed Controllers

FET Field Effect Transistors

FHSS Frequency-Hopping Spread Spectrum

FSK Frequency Shift Keying

GFSK Gaussian Frequency Shift-Keying I2C Inter-Integrated Circuit

IDE Integrated Development Environment

IMU Inertial Measurement Unit

LAN Local Area Network

LQ Linear Quadratic

MAV Micro Aerial Vehicle

MEMS Micro Electronic Mechanical System

PAN Personal Area Network

PCB Printed Circuit Board

PID Proportional Integral Derivative

PPM Pulse Width Modulation

PWM Pulse Position Modulation

RC Remote Control

RF Radio Frequency

RPM Revolution Per Minute SCL Serial Clock Line

UAV Unmanned Aerial Vehicles SDA Serial Data Line

USB Universal Serial Bus

Symbols

Symbol	Unit	Description
Θ	rad	Pitch angle
$\theta \cdot$	rad.s −1	Pitch angle rate
ρ	kg.m−3	Air density at sea level and 20◦C
τs	-	Time constant
∅	rad	Roll angle
∅ ·	rad.s −1	Roll angle rate
Φ	rad	Yaw angle
φ ·	rad.s −1	Yaw angle rate
ω	rad.s −1	Propeller angular velocity
ω	− rad.s −1	Quadrotor's angular velocities (P Q R)
DP	m	Propeller diameter
Fnet	N	Combination of all the forces acting on the quadrotor
		Fnet= (FxFyFz)
FP	N	Total thrust generated by the propellers FP = (Fpx FPy FPz)
G	m.s −2	Earth's gravity (constant value of 9.81m.s −2)
I	kg.m2	Inertia matrix of the Quadrotor
Mnet	-	Sum of all the moments acting on the Quadrotor
		Mnet = (Mx My Mz)
m	kg	Mass of the Quadrotor
P	rad.s −1	Angular speed around the ux0 axis of the Quadrotor
$P \cdot$	rad.s −2	Angular acceleration of the Quadrotor along the ux0 axis of

		Inertial reference frame
Q	rad.s^{-1}	Angular speed around the uy0 axis of the Quadrotor.
$\dot{Q}$	rad.s^{-2}	Angular acceleration of the Quadrotor along the uy0 axis of the Inertial reference frame
R	rad.s^{-1}	Angular speed around the uz0 axis of the Quadrotor
$\dot{R}$	rad.s^{-2}	Angular acceleration of the Quadrotor along the uz0 axis of the Inertial reference frame
S	-	Rotation matrix (also known as direction cosine matrix)
T	N	Propeller thrust
$\dot{V}$	m. s^{-2}	Linear acceleration of the Quadrotor along the uy axis of the Inertial reference frame
$\dot{W}$	m.s^{-2}	Linear acceleration of the Quadrotor along the uz axis of the Inertial reference frame
$\dot{U}$	m. s^{-2}	Linear acceleration of the Quadrotor along the ux axis of the Inertial reference frame

Chapter One

Introduction

1.1 Overview

Study and development of unmanned aerial vehicles (UAV) and micro aerial vehicles (MAV) are getting high encouragement nowadays, since the application of UAV and MAV can apply to variety of areas such as rescue mission, military, film making, agriculture and others [1].

Quadcopter has advantages over the conventional helicopter where the mechanical design is simpler. Besides that, Quadcopter changes direction by manipulating the individual propeller's speed and does not require cyclic and collective pitch control [2].

1.2 Aim & Objectives

1.2.1 Aim

This work aim to fly the quadcopter using hand gestures on a Leap Motion hand gesture sensor; each hand gesture has a unique instruction programmed on the motion sensor transmitted to the quadcopter flight controller with Wi-Fi wireless signals.

1.2.2 Objectives

- Run a successful visual simulation of the controller in Unity3D software.
- Implementation of the PID controller to the Arduino microcontroller as flight controller unit
- Construct the quadcopter and record a successful flight time of at least 1 minute

1.3 Problem Statement

The stability and control of a quadcopter is a challenging matter and the most fundamental feature in UAVs to sustain a balanced well controlled flight when building it instead of ready manufactured flight controllers.

1.4 Proposed Solution

PID controller allows you to change the UAVs flight characteristics, including how it responds to user input, how well and how quickly it stabilizes.

1.5 Methodology

The method applied on choosing the proper PID controller gains is trial and error method to set the values that results in balanced, stable and controlled flight. A collection of hardware components was used to build the quadcopter model providing the hardware to implement the PID flight controller.

1.6 Thesis Outline

Chapter 2 is the literature review and background discussing the UAVs historically and providing a background of the components that are essential in the operation of the quadcopter; relative reports that discussed building a quadcopter controller are also included in the literature review.

Chapter 3 includes the modeling of the quadcopter and the simulation of implementing the PID controller to a quadcopter flight controller.

Chapter 4 shows the steps that were followed in constructing the hardware of the quadcopter model that was used to implement the PID controller.

Chapter 5 discusses the result and discusses those results that were observed during building the hardware and implementing the software.

Chapter 6 is the conclusion of the thesis that includes the recommendations for our successors to solve the problems we could not and start their work where we finished.

Chapter Two

Literature Review and Background

2.1 Introduction

A quad copter flying machine also known as quad rotor is a rotary wing aircraft powered by four motors mounted on each edge of the structure in a an x or + formation depending on the formation.

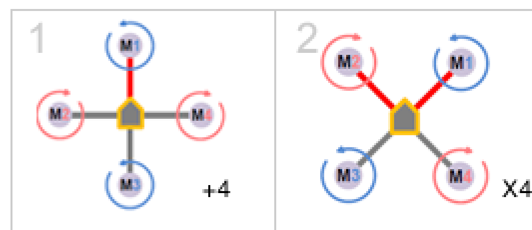


Figure 1: Shows the x and + structure configurations of a quadcopter

However the quadcopter concept isn't introduced recently considering that it existed since 1921; in January 1921 a US Army Corps. Contract of developing a vertical flying Machine was awarded to Dr. George de Bothezat and Ivan Jerome. The 1678 kg X-shaped structure supported 8.1 m diameter 6 blade rotor at each end of the 9 m arms and a 180 hp Le Rohne radial engine. At the ends of the lateral arms, two small propellers with variable pitch were used for thrusting and yaw control; each rotor had individual collective pitch control to produce differential thrust through vehicle inclination for translation.

Figure 2: shows the quadcopter rotorcraft of Bothezat

On the aircraft first flight in October 1922, the rotor craft weighed 1700 kg at take-off; the engine was soon upgraded to a 220 hp Bentley BR-2 rotary, about 100 flights were made by the end of 1923. Although the contract called for a 100 m hover, the highest it ever reached was 5 m. After expanding $200,000 de Bothezat demonstrated that his vehicle could be quite stable and that the practical helicopter, it was however unpowered, unresponsive, mechanically complex, susceptible to reliability problems and pilot work load was too high during hover to attempt lateral motion [3].

2.2 History of Quadcopter

Only few works were reported in the literature of a helicopter having four rotors. Young et al [4]. Sponsored by the Directorate Aerospace in NASA Ames Research Center present new configuration s of mini-drones and their applications among which the helicopter with four rotors called the Quad-rotor Tail-Sitter.

Pounds et al [5]; Conceived and developed a control algorithm for a prototype of an aerial vehicle having four rotors; they considered using an MIU (Measurement Inertial Unit) to measure the speed and acceleration. They use a linearization of the dynamic model to conceive the control algorithm; the result of the control law was tested in the simulation.

Altug et al [6]; Proposed a control algorithm to stabilize the quad-rotor using vision as principal sensor. They studied two methods, the first uses a control algorithm of linearization and the other uses the technique of back-stepping. They have tested the control laws in the simulation; they also present an experience using vision to measure yaw angle and the altitude.

The main reason there’s few works of literature taking quad-rotors as case study or research area is that the interest in quadcopters has increased recently and more researchers and aeronautics specialists are looking into the matter and conducting research.

2.3 Dynamic Model

2.3.1 Quad-rotor Characteristics

Consider Figure 3 below; the front and rear motors rotate counter-clockwise while the other two rotate clockwise, gyroscopic effects and aerodynamic torques tend to cancel in trimming flight.

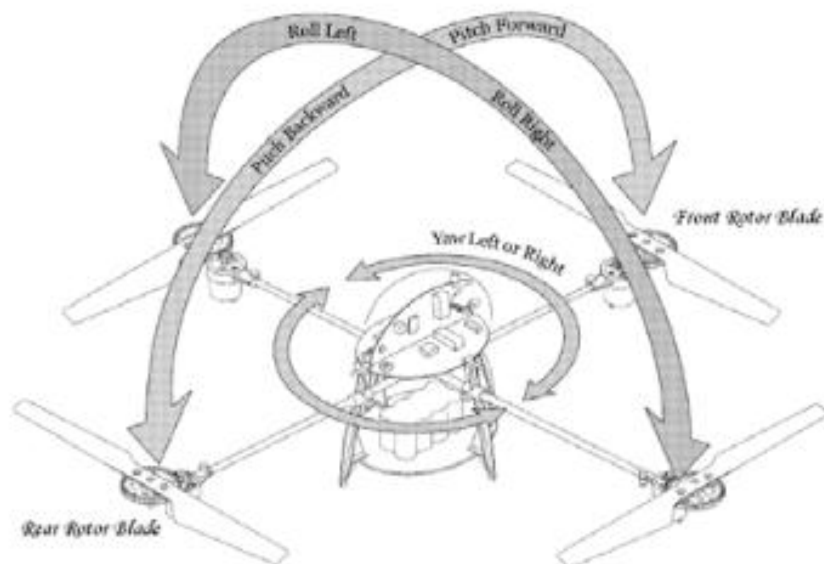


Figure 3: shows the Quad-rotor rotorcraft

This four-rotor rotor-craft does not have a swash plate; in fact, it doesn't need any blade pitch control. The collective input or throttle input is the sum of the thrusts of each motor (Figure 4).

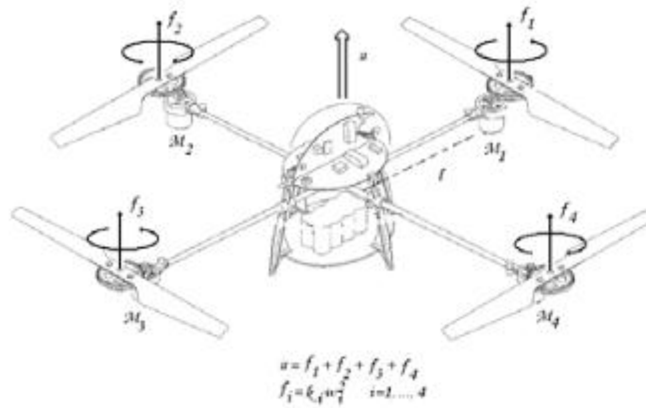


Figure 4: shows throttle control input

Pitch movement is obtained by increasing/reducing the speed of the rear motor while reducing/increasing the speed of the front motor. The roll movement is obtained similarly using the lateral motors. The yaw movement is obtained by increasing/decreasing the speed of the front and rear motors while decreasing/increasing the speed of the lateral motors; this should be done while keeping the total thrust constant.

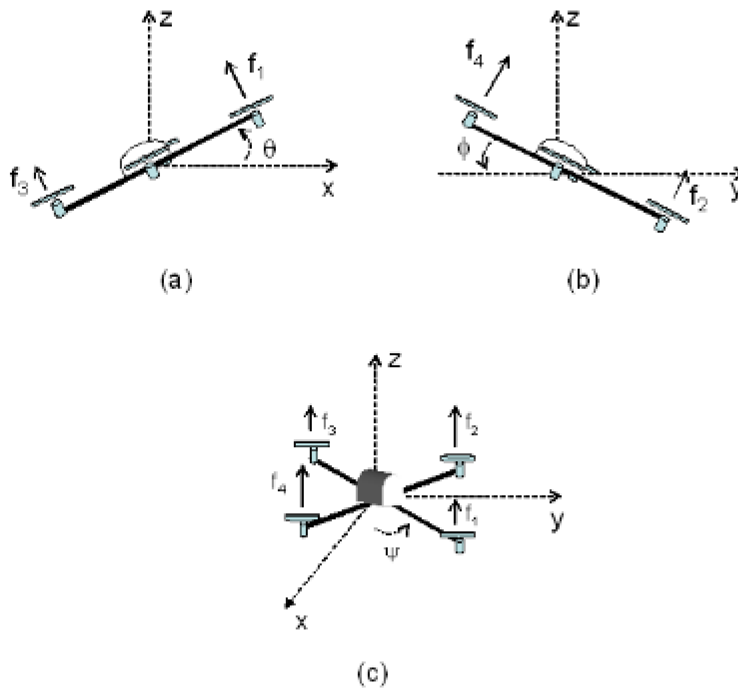


Figure 5: (a) Pitch (b) Roll (c) Yaw

2.3.2 System Description

The dynamic model of the quadcopter is presented simply by regarding it as a solid body developed in the three dimensions experiencing one force and three moments; the electric motors' dynamics are neglected along with its blades flexibility due to its relatively fast speed.

The generalized coordinates of the rotorcraft are:

$$q = (x, y, z, \varphi, \theta, \emptyset) \in R^6 \qquad \text{.......2.1}$$

Where:

(x, y, z) ≡ The position of the center of mass of the quadcopter

$(\varphi, \theta, \emptyset)$ ≡ Euler angles- angles of pitch, yaw and roll- of the quadcopter

Hence the model is naturally divided into translational and rotational coordinates:

$$\xi = (x, y, z) \in \Re^3, \quad \eta = (\varphi, \theta, \emptyset) \in S^3 \qquad \text{.......2.2}$$

The translational kinetic energy of the rotorcraft is

$$T_{trans} \triangleq \frac{m}{2} \xi \, T\xi \qquad \text{.......2.3}$$

Where m denotes the mass of the rotorcraft. The rotational kinetic energy is:

$$T_{rot} \triangleq \frac{1}{2} \eta^T \mathbb{J}_\eta \qquad \text{.......2.4}$$

The matrix $\mathbb{J}$ acts as the inertia matrix for the full rotational kinetic energy of the rotorcraft expressed directly in terms of generalized coordinates η. The only potential energy which needs to be considered is the gravitational potential given by

$$U = mgz \qquad \text{.......2.5}$$

The Langrangian is

$$L(q, q^{\cdot}) = T_{trans} + T_{rot} - U = \frac{m}{2}\, \xi\, T\xi + \frac{1}{2}\, \eta^{T} \mathbb{J}_{\eta} - mgz \qquad \text{.......2.6}$$

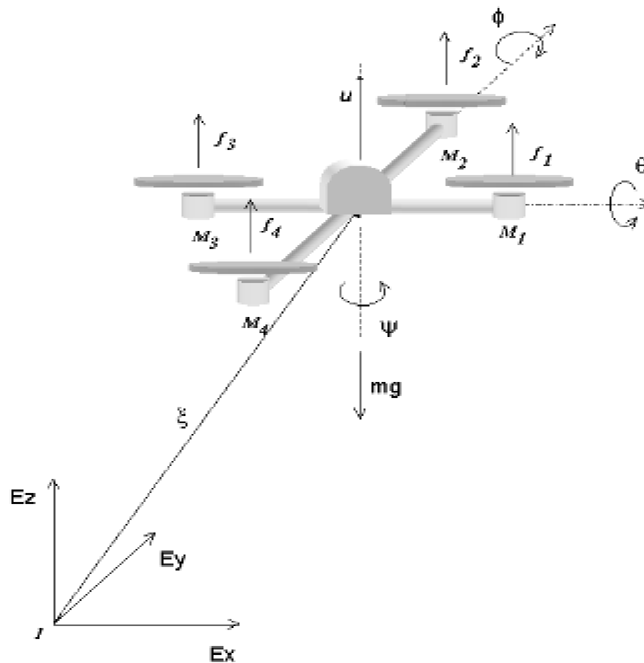


Figure 6: shows the quadcopter in an inertial frame

The dynamic model of the quadcopter is obtained from the Euler Lagrange equations with external generalized force

$$\frac{d}{dt}\frac{\partial L}{\partial q^{\cdot}} - \frac{\partial L}{\partial q} = F \qquad \text{…….2.7}$$

$F = (F_{\xi}, \tau)$ Where τ are the generalized moments and F_{ξ} is the translational force applied to the rotorcraft due to control inputs, we ignore the small body forces because they are generally of a much smaller magnitude than the principal control inputs u andτ, then we write

$$\mathcal{F} = \begin{pmatrix} 0 \\ 0 \\ u \end{pmatrix} \qquad \text{…….2.8}$$

See figure 4:

$$u = f1 + f2 + f3 + f4 \qquad \text{…….2.9}$$

$$f_i = k_i {\omega_i}^2 \quad i = 1, \dots, 4 \qquad \text{…….2.10}$$

Where $k_i > 0$ is a constant and ω_i is the angular speed of motor i then

$$F_{\xi} = R\mathcal{F} \qquad \text{…….2.11}$$

Where R is the transformation matrix representing the orientation of the rotorcraft, we use c_{θ} for cosθ and s_{θ} for $sin(\theta)$

$$R = \begin{pmatrix} c_\theta c_\psi & s_\psi s_\theta & -s_\theta \\ c_\psi s_\theta s_\phi & -s_\psi c_\phi s_\psi s_\theta s_\phi & +c_\psi c_\phi c_\theta s_\phi \\ c_\psi s_\theta c_\phi & +s_\psi s_\phi s_\psi s_\theta c_\phi & -c_\psi s_\phi c_\theta c_\phi \end{pmatrix} \qquad \ldots\ldots.2.12$$

The generalized moments on the η variables are

$$\tau \triangleq \begin{pmatrix} \tau_\psi \\ \tau_\theta \\ \tau_\phi \end{pmatrix} \qquad \ldots\ldots.2.13$$

Where

$$\tau_\psi = \sum_{i=1}^{4} \tau M_\ell \quad \tau_\theta = (f2 - f4)\ell \quad \tau_\phi = (f3 - f1)\ell \qquad \ldots\ldots.2.14$$

Where ℓ is the distance from the motors to the center of gravity and τ_{M_ℓ} is the couple produced by motor M_i.

Since the Langrangian contains no cross-terms in the kinetic energy combining $\dot{\xi}$ and $\dot{\eta}$, the Euler-Langrange equation may be divided to the $\dot{\xi}$ dynamics and η dynamics.

$$m\ddot{\xi} + \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix} = \mathcal{F}_\xi \qquad \ldots\ldots.2.15$$

$$\mathbb{J}\ddot{\eta} + \dot{\mathbb{J}}\dot{\eta} - \frac{1}{2}\frac{\partial}{\partial \eta}\left(\dot{\eta}^T \mathbb{J}\dot{\eta}\right) = \tau \qquad \ldots\ldots.2.16$$

Defining the Coriolis/centripetal vector

$$\bar{V}(\eta, \dot{\eta}) = \dot{\mathbb{J}}\dot{\eta} - \frac{1}{2}\frac{\partial}{\partial \eta}\left(\dot{\eta}^T \mathbb{J}\dot{\eta}\right) \qquad \ldots\ldots.2.17$$

We may write

$$\mathbb{J}\ddot{\eta} + \bar{V}(\eta, \dot{\eta}) = \tau \qquad \text{.......2.18}$$

But we can rewrite $\bar{V}(\eta, \dot{\eta})$

$$\bar{V}(\eta, \dot{\eta}) = \left(\dot{\mathbb{J}} - \frac{1}{2}\frac{\partial}{\partial \eta}(\dot{\eta}^T \mathbb{J})\dot{\eta}\right) = C(\eta, \dot{\eta})\dot{\eta} \qquad \text{.......2.19)}$$

Where $C(\eta, \dot{\eta})$ is referred to as the Coriolis terms and contains the gyroscopic and centrifugal terms associated with the η dependence of $\mathbb{J}$. Finally:

$$m\ddot{\xi} = u\begin{pmatrix} -sin\theta \\ cos\theta\ sin\phi \\ cos\theta\ cos\phi \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} \qquad \text{.......2.20}$$

$$\mathbb{J}\ddot{\eta} = -C(\eta, \dot{\eta})\dot{\eta} + \tau \qquad \text{.......2.21}$$

In order to simplify let us propose a change of input variables.

$$\tau = C(\eta, \dot{\eta})\dot{\eta} + \mathbb{J}\tilde{\tau} \qquad \text{.......2.22}$$

$$\tilde{\tau} = \begin{pmatrix} \tilde{\tau}_\psi \\ \tilde{\tau}_\theta \\ \tilde{\tau}_\phi \end{pmatrix} \qquad \text{.......2.23}$$

Are the new inputs, then $\ddot{\eta} = \tilde{\tau}$, rewriting the equations

$$m\ddot{x} = -u\ sin\theta \qquad m\ddot{y} = u\ cos\theta\ sin\phi \qquad m\ddot{z} = u\ cos\theta\ cos\phi - m \qquad \text{.......2.24}$$

$$\ddot{\psi} = \tilde{\tau}_\psi \qquad \ddot{\theta} = \tilde{\tau}_\theta \qquad \ddot{\phi} = \tilde{\tau}_\phi \qquad \text{.......2.25}$$

Where x and y are the coordinates of the horizontal plane and z is the vertical position. ψ is the yaw angle around the z-axis, θ is the pitch angle around (new) y-axis and ϕ is the roll angle around the (new) x-axis. The control inputs u,$\tilde{\tau}_\psi$, $\tilde{\tau}_\theta$ and $\tilde{\tau}_\phi$ are the total thrust or collective input directed from the bottom of the aircraft and the new angular moments.

2.4 Quadcopter Block Diagram

The quadcopter rotorcraft consists of an Inertial Measurement Unit (IMU), flight control unit, Electric Speed Controllers (ESC) for the motors and a Radio Frequency (RF) receiver; as shown on figure 7 below the IMU consists of an accelerometer, gyroscope and a magnetometer.

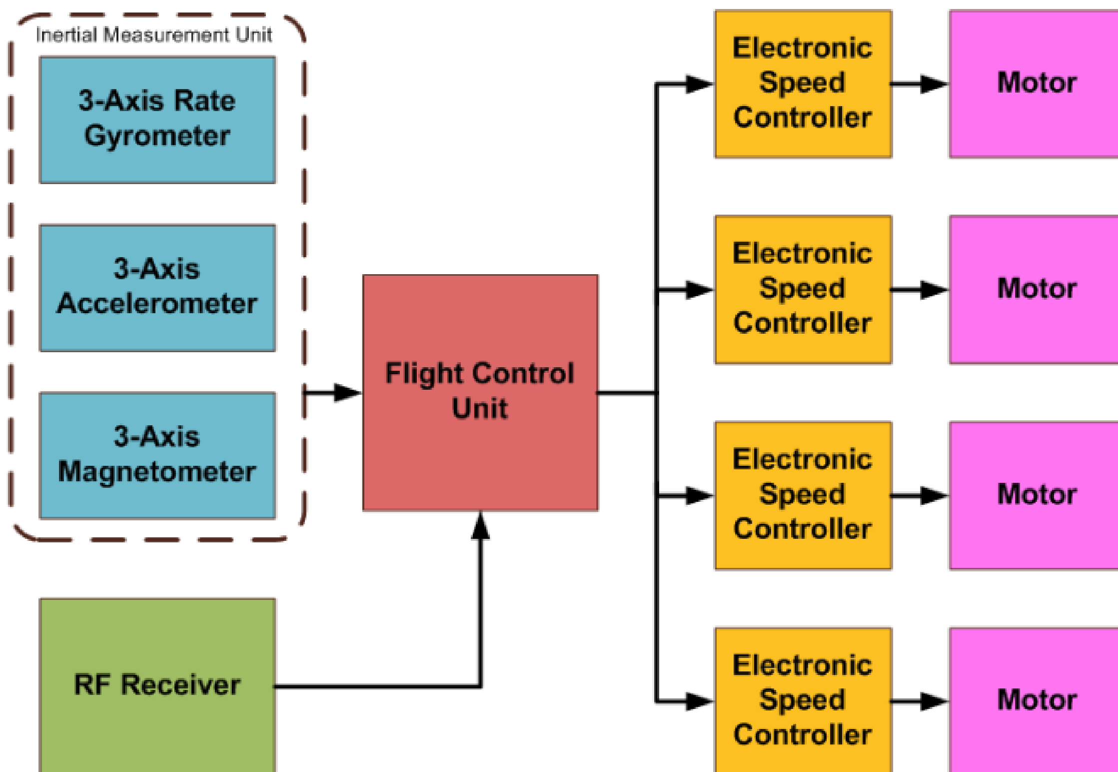


Figure 7: Quadcopter Block Diagram

2.4.1 Inertial Measurement Unit (IMU)

Accelerometers are devices that measure acceleration. A gyroscope is a device consisting of a wheel or disk m0ounted so that it can spin rapidly about an axis that is itself free to alter in direction. The orientation of the axis is not affected by tilting of the mounting; so, gyroscopes can be used to provide stability or maintain a reference direction in navigation systems,

automatic pilots, and stabilizers. A magnetometer is an instrument used for measuring magnetic forces, especially the earth's magnetism.

2.4.2 RF Receiver

A quadcopter consists of a communication system to transmit pilot commands to the copter flight controller to carry out a pitch, roll or a yaw; this system consists of a transmitter which is the R/C controller the pilot uses to control the rotorcraft and a Radio Frequency receiver on the quad copter (RF) to receive information signals sent by the R/C controller.

2.4.3 Brushless DC Motor

BLDC motors are a type of synchronous motor. This means the magnetic field generated by the stator and the magnetic field generated by the rotor rotates at the same frequency. The stator of a BLDC motor consists of stacked steel laminations with windings placed in the slots that are axially cut along the inner periphery, the stator resembles that of an induction motor; however, the windings are distributed in a different manner. Most BLDC motors have three stator windings connected in star fashion. Each of these windings is constructed with numerous coils interconnected to form a winding. One or more coils are placed in the slots and they are interconnected to make a winding. Each of these windings is distributed over the stator periphery to form even numbers of poles.

The rotor is made of permanent magnet and can vary from two to eight pole pairs with alternate North (N) and South (S) poles. Based on the required magnetic field density in the rotor, the proper magnetic material is chosen to make the rotor. Ferrite magnets are traditionally used to make permanent magnets. As the technology advances, rare earth alloy magnets are gaining popularity. The ferrite magnets are less expensive but they have the disadvantage of low flux density for a given volume. In contrast, the alloy material has high magnetic density per volume and enables the rotor to compress further for the same torque. Also, these alloy magnets improve the size-to-weight ratio and give higher torque for the same size motor using ferrite magnets. Neodymium (Nd), Samarium Cobalt (SmCo) and the alloy of Neodymium, Ferrite and Boron (NdFeB) are some examples of rare earth alloy magnets. Continuous research is going on to improve the flux density to compress the rotor further.

2.4.4 Electronic Speed Controller (ESC)

An electronic speed controller or ESC is an electronic circuit that vary an electric motor's speed, its direction and possibly also to act as a dynamic brake. ESCs most often used for brushless motors essentially providing an electronically generated three-phase electric power low voltage source of energy for the motor.

2.4.5 Flight Controller

A Flight Controller Unit is the block responsible of receiving the flight commands, stabilizing the quad copter, executing pilot commands, controlling the speed of the motors and performing flight movements.

2.5 Tools

2.5.1 Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. [7]

2.5.2 Proteus 8 Labcenter

Protues software by Labcenter enabling powerful features to design, test and layout professional PCB layouts and supports the schematics and simulation of 800 microcontrollers.

2.5.3 Unity 3D

Unity3D is a powerful cross-platform 3D engine and a user-friendly development environment for developing 3D projects and simulations equipped with graphical and programmatic documentation and scripting guide to simulate real world physics and variables making it easy for the user to run their simulations and see the result visually rather than tables and figures.

2.5.4 eCalc – xcopterCalc

eCalc is an online tool for simulating real-life quadrotor parameters by providing quadcopter parameters such as model weight, number of rotors, frame size, elevation, … etc. and other parameters concerning the flight controller, Motors and propellers then assessing and providing suggestion to get the best performance of the quadcopter.

Chapter Three

Modeling and Simulation

3.1 Mathematical Modeling

To mathematically write the movement of an aircraft we must employ Newton's second law of motion. As such, the equations of the net force and moment acting on the Quadrotor's body (respectively F_{net} and M_{net}) are provided:

$$F_{net} = \frac{d}{dt}(mv)_b + \omega^- \times (mv)_b \text{.................................} \mathbf{3.1}$$

$$M_{net} = \frac{d}{dt}(I\omega^-)_b + \omega^- \times (I\omega^-)_b \text{...............................} \mathbf{3.2}$$

Where **I** is the inertia matrix of the Quadrotor, **v** is the vector of linear velocities andω^- is the vector of angular velocities. If the equation of Newton's second law is to be as complete as possible, we should add extra terms such as the force of gravity (F$_{g}$) which is too significant to be neglected, thus it is defined by

$$F_g = mS[0\ \ 0\ \ g]^T = mg[-sin\theta\ cos\theta sin\emptyset\ \ cos\theta cos\emptyset]^T{}_b \text{..............} \mathbf{3.3}$$

Where **S** is the rotation matrix

$$S=\begin{bmatrix} \cos\theta\cos\varphi & \cos\theta\sin\varphi & -\sin\theta \\ \sin\varphi\sin\theta\cos\varphi - \cos\emptyset\sin\varphi & \cos\emptyset\ \cos\ \varphi + \sin\emptyset\ \sin\theta\sin\varphi & \sin\emptyset\cos\theta \\ \cos\emptyset\sin\theta\cos\varphi + \sin\emptyset\sin\varphi & \sin\theta\cos\emptyset\sin\varphi - \sin\emptyset\cos\varphi & \cos\theta\cos\emptyset \end{bmatrix}$$

The force of gravity together with the total thrust generated by the propellers (F$_{P}$) have therefore to be equal to the sum of forces acting on the Quadcopter:

$F_{net} = F_p + F_g$**..................................3.4**

Combine equations 1, 3, 4. We can write the vector of linear accelerations acting on the vehicle's body:

$$\begin{bmatrix} U^{\cdot} \\ V^{\cdot} \\ W^{\cdot} \end{bmatrix} = \begin{bmatrix} 0 & -R & Q \\ R & 0 & P \\ -Q & P & 0 \end{bmatrix} \begin{bmatrix} U \\ V \\ W \end{bmatrix} + \frac{1}{m} \begin{bmatrix} F_{px} \\ F_{py} \\ F_{pz} \end{bmatrix} + \begin{bmatrix} -sin\theta \\ cos\theta sin\emptyset \\ cos\theta cos\emptyset \end{bmatrix} g \text{......................3.5}$$

Where $[F_{P\,x}\,F_{P\,y}\,F_{P\,z}]$ are the vector elements of $\mathbf{F}_P$.

The forces and moments acting on Quadcopter of (×) congigrations

$F_{pz} = -(T_1 + T_2 + T_3 + T_4)$ 3.6

$M_x = l(-T_1 - T_2 + T_3 + T_4)$ 3.7

$M_y = l(T_1 - T_2 + T_3 - T_4)$ 3.8

$M_z = K_{T\,M}(T_1 + T_2 - T_3 + T_4)$.....................3.9

where

L is the distance to the aircrafts COG, and *KT M* is a constant that relates moment and thrust of a propeller

Assuming the Quadcopter is a rigid body with constant mass and axis aligned with the principal axis of inertia, then the tensor I becomes a diagonal matrix containing only the principal moments of inertia:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \text{..............................3.10}$$

Combine equation 9 and 10 result:

$$M_{net} = \begin{bmatrix} I_{xx}P^{\cdot} \\ I_{yy}Q^{\cdot} \\ I_{zz}R^{\cdot} \end{bmatrix} + \begin{bmatrix} (I_{zz} - I_{yy})QR \\ (I_{xx} - I_{zz})PR \\ (I_{yy} - I_{xx})PQ \end{bmatrix} \ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.11$$

$$M_{net} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \begin{bmatrix} P^{\cdot} \\ Q^{\cdot} \\ R^{\cdot} \end{bmatrix} + \begin{bmatrix} P \\ Q \\ R \end{bmatrix} \times \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} \ldots\ldots\ldots 3.12$$

Then

$$\begin{bmatrix} P^{\cdot} \\ Q^{\cdot} \\ R^{\cdot} \end{bmatrix} = \begin{bmatrix} \frac{M_x}{I_{xx}} \\ \frac{M_y}{I_{yy}} \\ \frac{M_z}{I_{zz}} \end{bmatrix} - \begin{bmatrix} \frac{(I_{zz}-I_{yy})QR}{I_{xx}} \\ \frac{(I_{xx}-I_{zz})PR}{I_y} \\ \frac{(I_{yy}-I_{xx})PQ}{I_{zz}} \end{bmatrix} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.13$$

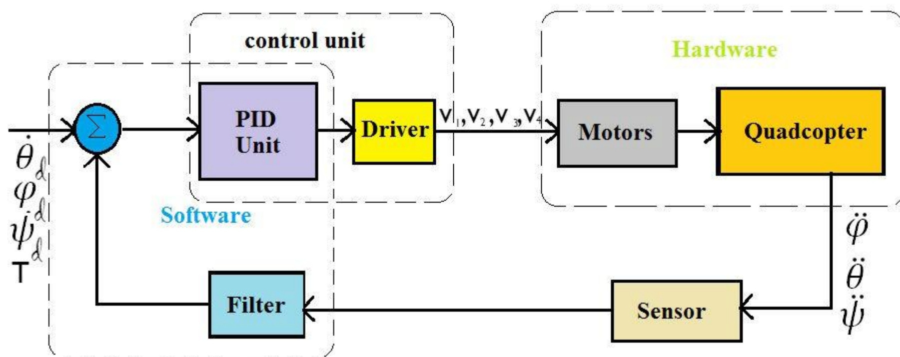


Figure 8: Quadcopter control loop

3.1 PID Controller

Proportional-Integral-Derivative controller is a closed feedback loop system used in applications requiring continuous modulated control by continuously calculating an error value $e(t)$ representing a difference between a desired set point and a measured process variable then applies correction based on proportional, integral and derivative terms

3.1.1 Theory of operation

The sum of the three PID terms produces a Manipulated Variable (MV) that is used to correct the error of the system:

$$u(t) = MV(t) = K_p\, e(t) + K_i \int_0^t \tau\, d\tau + K_d \frac{de(t)}{dt} \quad \text{.......................} \; 3.14$$

Where

K_p is proportional gain

K_i is the integral gain

K_d is the derivative gain

$e(t)$ is the error

t is the continuous time

τ is the variable of integrations

The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant K_p, called the proportional gain constant.

The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain (K_i) and added to the controller output.

The derivative term of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain K_d. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gainK_d.

The present, past and future errors are dependent on the terms of the PID respectively meaning the present error depends on P, past error accumulates the I and future error is forecasted by the D term.

The proportional controller K_P will reduce the rise time and the steady state error but will not eliminate the steady state error; the integral controller K_I will eliminate the steady state error however it may worsen the transient error; a derivative controller K_D will increase the stability of the system by reducing the overshoot and improving the transient response.

Table 1: Effects of each of controllers K_p, K_i, and K_d on a closed-loop system

Closed Loop Response	Rise Time	Overshoot	Settling Time	Steady State Error
K_P	Decrease	Increase	Small Change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	Small Change	Decrease	Decrease	Small Change

3.1.2 PID Tuning

Tuning a control loop is the adjustment of its control parameters (proportional band/gain, integral gain/reset, derivative gain/rate) to the optimum values for the desired control response. Stability (no unbounded oscillation) is a basic requirement, but beyond that, different systems have different behavior, different applications have different requirements, and requirements may conflict with one another. [8]

3.1.3 Classic PID Equations

Proportional controller

$$A(t) = K_P * e(t) \quad \text{……………………3.15}$$

Integral Controller

$$A(t) = K_I * \int_0^t e(t)dt \quad \text{……………………3.16}$$

Derivative Controller

$$A(t) = K_D * \frac{d\,e(t)}{dt} \quad \text{……………………3.17}$$

The total equation of the classic controller

$$u(t) = K_P * e(t) + K_I \int_0^t e(t)dt + K_D \frac{d\,e(t)}{dt} \quad \text{……………………3.18}$$

3.2 Simulation

Simulation of the quadcopter was conducted over three software's each of which completing the other's limitation Proteus was used to design the circuit and add the code simulating the operation of the quadrotor; unity provided a graphic simulation of the quadrotor operating using a PID controller simulated using unity scripting and documentation references; the hardware capabilities of the quadrotor were tested using eCalc to define which parameters needed adjusting until an appropriate hardware status was obtained.

3.2.1 Unity

Unity documentation provided a better ground to build the simulation of the PID controllers through code only existing inside the unity environment written for the purpose of simulating real life physics subjecting the PID controller to an environment similar to real life parameters.

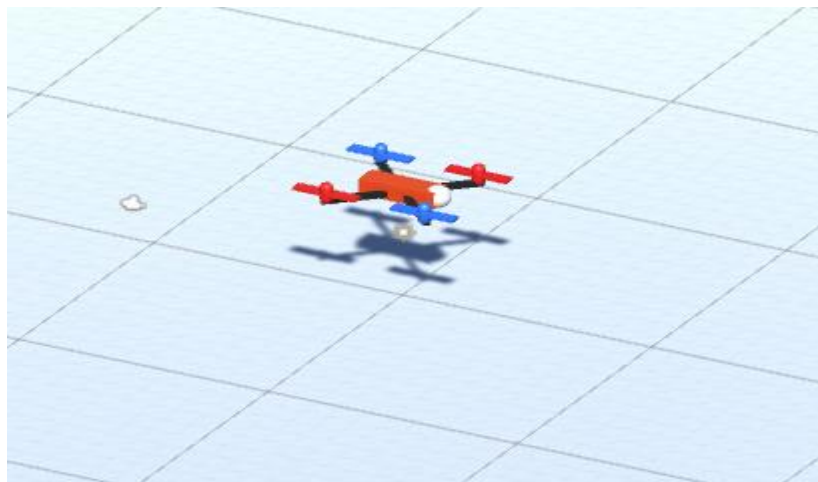


Figure 9: Quadrotor 3D model

From figure 10 above illustrates the three-dimensional model of the quadrotor, the physics forces applied on the motors to rotate them is supplied to the four motors simultaneously, hence the motors are named individually as Front Right (FR), Front Left (FL), Back Right (BR) and Back Left (BL).

To simulate the quadrotor as a rigid body unity offers a physics component called a rigid body providing all the physical specifications of a real rigid body. Consider figure 11 below:

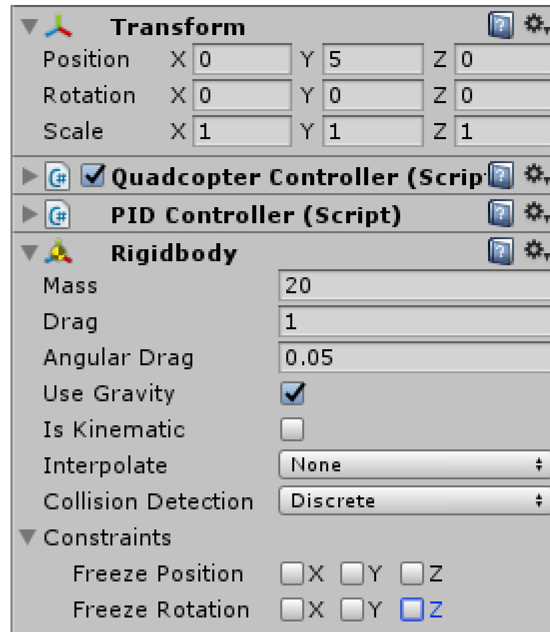


Figure 10 :Rigid body component attached to the quadrotor 3D model

The Transform tab describes the coordinates- x, y and z coordinates system- of the quadrotor position, rotation and scale in the simulation; position (0, 5, 0) means the quadrotor is in the origin of the simulation plane with an elevation of 5 units from the ground and (0, 0, 0) rotation means that the quadrotor is perfectly balanced and there is no rotation around any axes and the scale is one unit cubed representing height, width and length.

A rigid body in unity is a component simulating real objects physics qualities such as mass, drag, angular drag, etc.; the use gravity check box allows you to use real gravity which is going to affect the object as soon as the simulation starts and is kinematic specifies whether the object will remain at rest the whole time the simulation is running and the collision detection is set to discrete that will update it every frame of the simulation.

Now when the simulation runs the gravity will pull the quadrotor to the ground and it will remain in this state if no force is applied to counter the gravity two pieces of code are associated with this the quadcopter controller which is the implementation of the PID controller to the quadcopter that is simulated in another script.

The simulation will require constant variables to be initialized by the quadcopter PID controller; maximum propeller force, maximum torque, throttle and move factor are set as shown in figure below, the mass and drag of the quadcopter is set in the rigid body component.

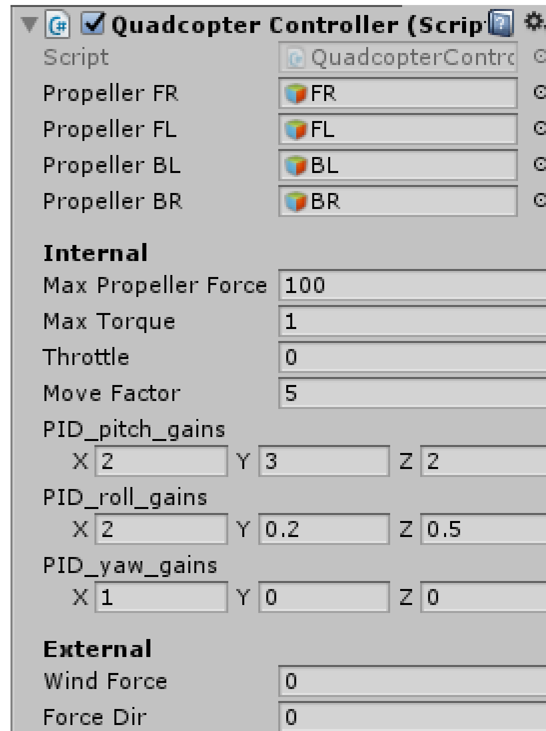


Figure 11: Quadcopter controller parameters

Once the simulation starts the quadrotor will be affected by gravity and instantly pulled to the ground waiting for user input as it orientation is balanced; the craft controller has two sticks the one on the left controls the steering forward and back, left and right and the right stick controls the throttle and the turning-yawing – left and right which pass the user input to another function to execute the command that is set as desired point for the controller.

Pitch, roll and yaw errors will be calculated to be passed to the PID controller assuming no errors at all when the simulations starts then adapting the PID variables to the throttle and calculating the force that must be added to the propellers based on the PID output to translate the quadrotor from its measured position to the user desired point.

3.2.3 eCalc

The hardware chosen for the quadrotor was passed as input with detailed specifications to the eCalc tool to determine its ability to withstand the load and power consumption then adjusting the hardware based on the suggestions provided to ensure best performance possible.

General	Model Weight: 800 g incl. Drive; 28.2 oz	# of Rotors: 4; flat	Frame Size: 625 mm; 24.61 inch	FCU Tilt Limit: no limit		Field Elevation: 500 m ASL; 1640 ft ASL	Air Temperature: 35 °C; 95 °F	Pressure (QNH): 1013 hPa; 29.91 inHg
Battery Cell	Type (Cont. / max. C) - charge state: LiPo 2200mAh - 25/35C - normal	Configuration: 3 S 1 P	Cell Capacity: 2200 mAh; 2200 mAh total	max. discharge: 85%	Resistance: 0.0095 Ohm	Voltage: 3.7 V	C-Rate: 25 C cont.; 35 C max	Weight: 55 g; 1.9 oz
Controller	Type: max 30A	Current: 30 A cont.; 30 A max	Resistance: 0.008 Ohm	Weight: 40 g; 1.4 oz		**Accessories**	Current drain: 0 A	Weight: 0 g; 0 oz
Motor	Manufacturer - Type (Kv) - Cooling: ProTronik - 2810-1000 (1000); good; search...	KV (w/o torque): 1000 rpm/V; Prop-Kv-Wizard	no-load Current: 1.6 A @ 8.4 V	Limit (up to 15s): 230 W	Resistance: 0.05 Ohm	Case Length: 31.5 mm; 1.24 inch	# mag. Poles: 14	Weight: 88 g; 3.1 oz
Propeller	Type - yoke twist: select... - 0°	Diameter: 10 inch; 254 mm	Pitch: 4.5 inch; 114.3 mm	# Blades: 2	PConst / TConst: 1.2 / 1.0	Gear Ratio: 1 : 1		calculate

Figure 12: Hardware Specifications passed into eCalc for evaluation

As shown in Figure above the specification of hardware pieces are passed onto to the input fields of the tool providing a detailed description about the important hardware operating the quadcopter such as battery cell, controller, motor and propellers also general specifications of weight, number of rotors, frame size, … etc., are required in order to provide specific performance graphs.

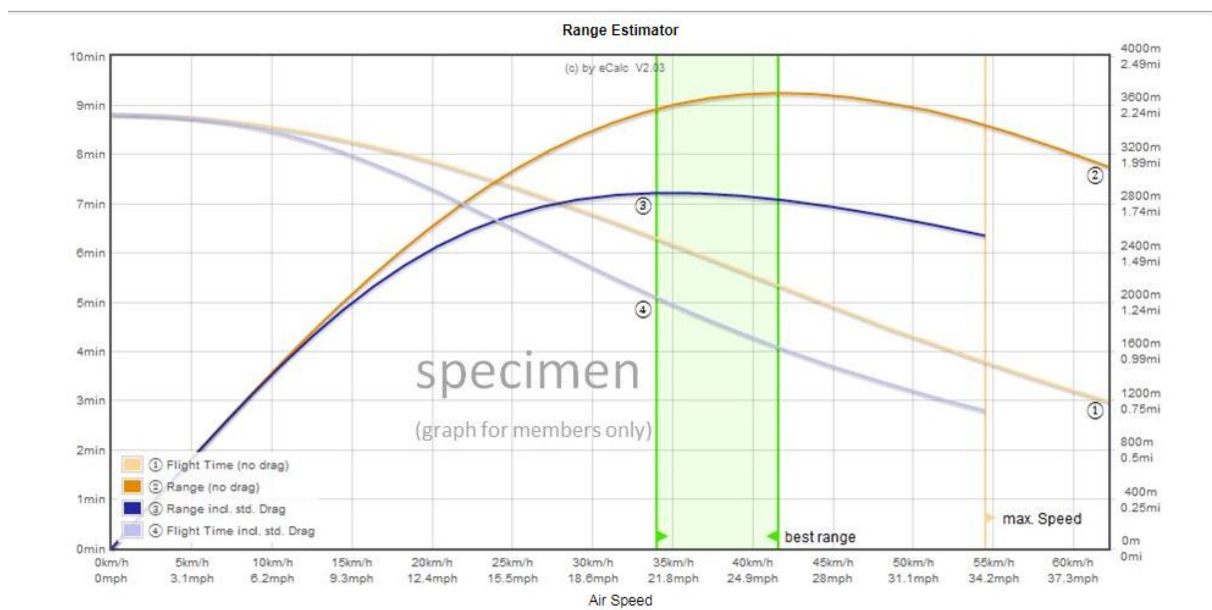


Figure 13 :Range Estimation

Figure above shows the analysis made by eCalc for the range that the quadcopter with the specifications entered possesses. Flight time estimated by eCalc is 9 minutes with a maximum speed of 54 km/h (33.2 mph); the green area denotes the best flight operation range.

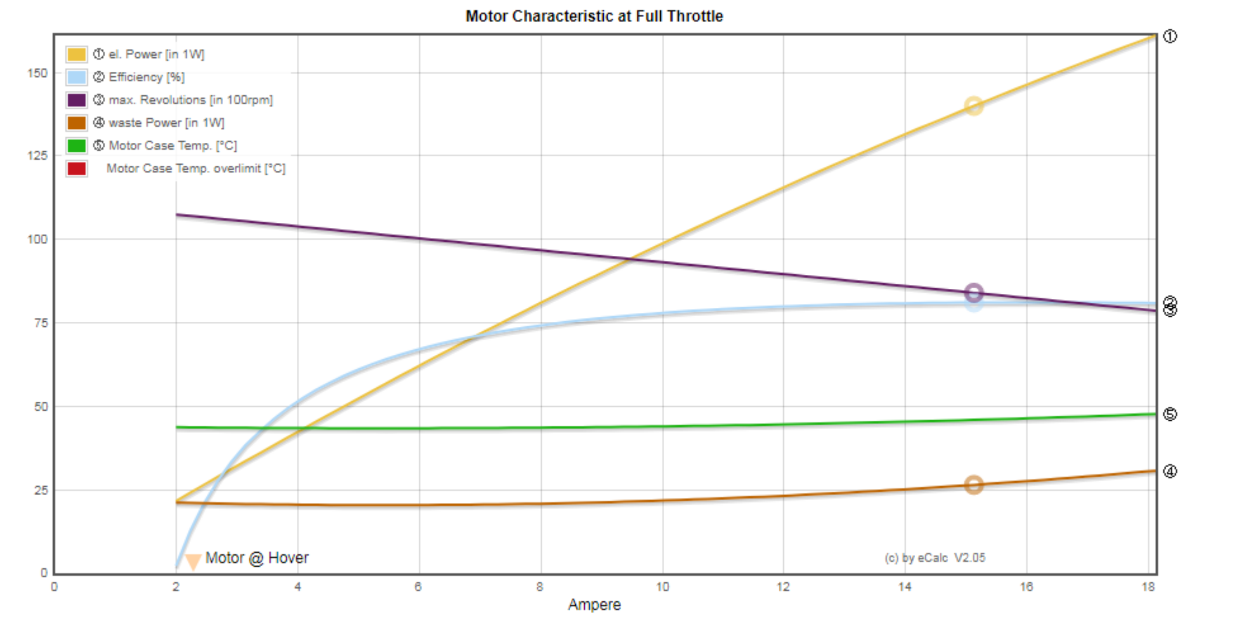


Figure 14: Motor characteristics at full throttle

Figure above denotes the characteristics of the motors at full throttle, the best characteristics are a power of 137.5 W, 75% efficiency, a maximum RPM of 85 rpm and a wasted power of 25 W shown by the circles.

Chapter Four

Construction

4.1 Overall hardware connection to the microcontroller

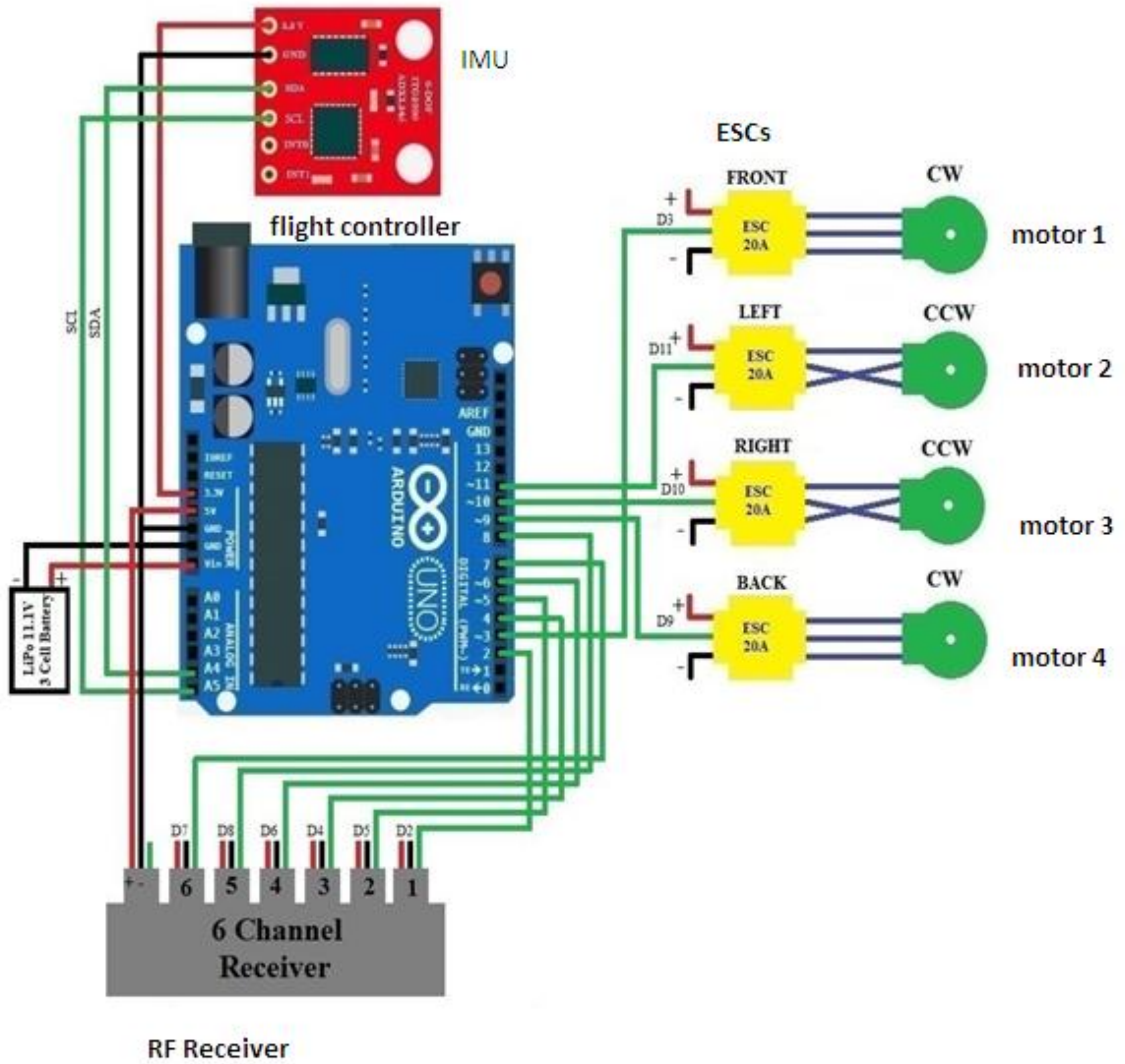


Figure 15: Overall Quadcopter hardware connection

4.2 Hardware Components

4.2.1 The Frame of the Quadcopter

Typical quad-rotors utilize a four-spar method, with each spar anchored to the central hub. The frame of the quad copter is composed of a combination of materials chosen for their strength, weight and flexibility.

When designing an autonomous quad-rotor, there are several material options which must be considered. When designing a machine capable of flight, weight must be greatly well thought-out.

The airframe is the mechanical structure of an aircraft that supports all the components, much like a “skeleton” in Human Beings. Designing an airframe from scratch involves important concepts of physics, aerodynamics, materials engineering and manufacturing techniques to achieve certain performance, reliability and cost criteria.

Figure 16: Plastic foam frame incase in fiber glass

4.2.2 The Microcontroller – Arduino Uno

Arduino Uno is a microcontroller board based on the ATmega328P It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz quartz crystal, a USB connection, a power jack, an ICSP header and a reset button.

It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. "Uno" means one in Italian and was chosen to mark the release of Arduino Software (IDE) 1.0. The

Uno board and version 1.0 of Arduino Software (IDE) were the reference versions of Arduino, The Uno board is the first in a series of USB Arduino boards, and the reference model for the Arduino platform; for an extensive list of current, past or outdated boards see the Arduino index of boards.

Figure 17: Arduino UNO Microcontroller Board

4.2.3 Electronic Speed Controllers

An electronic speed control or ESC is a circuit with the purpose to control an electric motor's speed, its direction and possibly also to act as a dynamic brake in some cases. ESCs are often used on electrically powered brushless motors essentially providing an electronically-generated three phase electric power, with a low voltage source.

An ESC interprets control information in a way that varies the switching rate of a network of field effect transistors (FETs), not as mechanical motion as would be the case of a servo. The quick switching of the transistors is what causes the motor itself to emanate its characteristic high-pitched whine, which is especially noticeable at lower speeds. It also allows much smoother and more precise variation of motor speeds in a far more efficient manner than the mechanical type with a resistive coil and moving arm once in common use.

The ESC generally accepts a nominal 50 Hz Pulse Width Modulation (PWM) servo input signal whose pulse width varies from 1ms to 2ms. When supplied with a 1ms width pulse at 50 Hz, the ESC responds by turning off the DC motor attached to its output. A 1.5ms pulse-width input signal results in a 50% duty cycle output signal that drives the motor at approximately 50% speed. When presented with 2.0ms input signal, the motor runs at full speed due to the 100% duty cycle (on constantly) output.

The correct phase varies with the motor rotation, controlled and monitored by the ESC. The orientation of the motor is determined by the back EMF (Electromotive Force). The back EMF is the voltage induced in a motor wire by the magnet spinning past its internal coils. Finally, a PID algorithm in the controller adjusts the PWM to maintain a constant RPM.

Reversing the motor's direction may also be accomplished by switching any two of the three leads from the ESC to the motor.

Ideally the ESC controller should be paired to the motor and rotor craft with the following considerations.

1. Temperature and thermal characteristics.

2. Max Current output and Impendence.

3. Needs to be Equipped with a BEC (Battery Eliminator Circuit) to eliminate the need of a second battery.

4. Size and Weight properties.

5. Magnet Rating.

Figure 18: 30A Brushless ESC

Additionally, the speed controller has fixed throttle settings so that the "stop" and "full throttle" points of all the various modes which can be cut through cleanly. The controller produces audible beeps to assist in navigating through the program modes and troubleshooting logs.

Table 2：Specification for 30A Brushless ESC

30A Brushless ESC Output	Continuous 30A, burst 40A up to 10 Sec
Input voltage	2-4 cells lithium battery or 5-12 cells NiCd/NiMH battery
BEC	2A / 5V (Linear mode).
Max speed	210,000rpm for 2 poles BLM, 70,000rpm for 6 poles BLM, 35,000rpm for 12 poles BLM. (BLM: Brushless Motor
Size	45 * 24 * 11mm / 1.8 * 0.9 * 0.4in
Weight	25g / 0.9oz
Item total weight	480g / 1.06Lbs

4.2.4 Inertial Measurement Unit

Precision and accuracy is important when it comes to Accelerometer and gyroscope measurement. We require a 3-axis accelerometer and gyroscope that provides reliable and accurate data. It is also an advantage if they can be on the same chip. For this reason, we went with the MPU-600, which is a small, thin, ultralow power, 3-axis accelerometer and gyroscope. The device is very accurate, as it contains 16-bis analog to digital conversion hardware for each channel. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. The sensor has a "Digital motion processor" which can be programmed with firmware and is able to do complex calculations with the sensor values.

Figure 19: MPU6050 IMU used in our quadcopter

4.2.5 IMU interface with ARDUINO

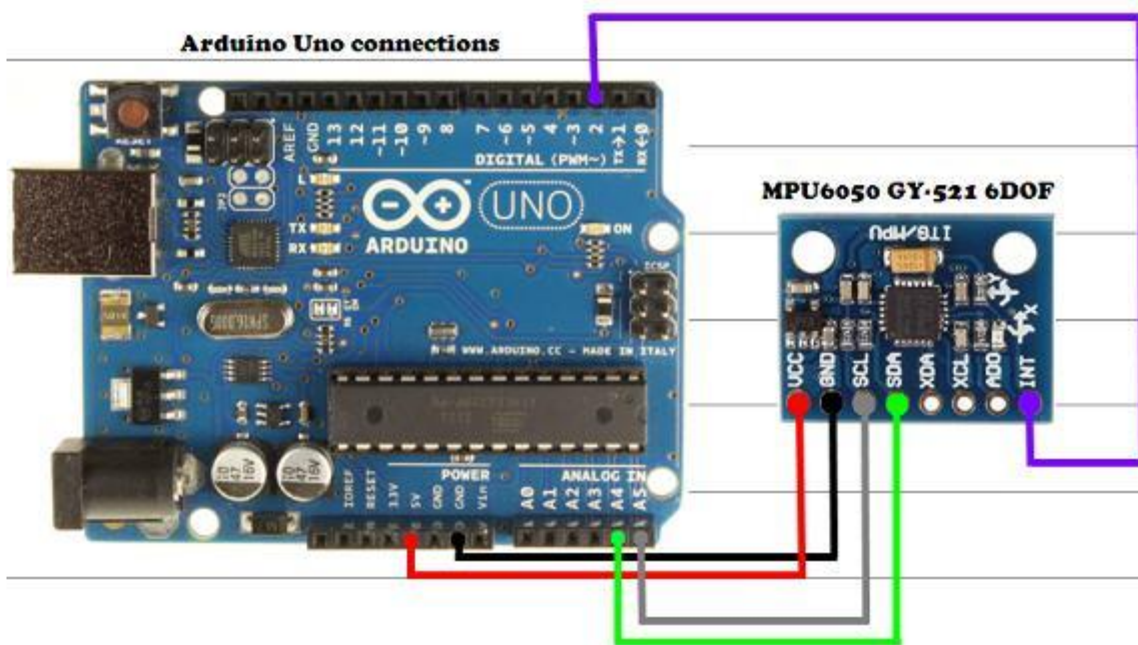


Figure 20: MPU6050 Interface with Arduino

4.2.6 The Battery Pack

Selecting the proper battery for our rotor copter was a challenging task. Nickel Cadmium (NiCd), Nickel Metal Hydride (NiMH), and Lithium Polymer (LiPo) were common choices with the advantages and disadvantages of each battery pack.

NiCd batteries are reasonably inexpensive, but they have a number of negatives. NiCd batteries need to be fully discharged after each use. If they aren't, they will not discharge to their full potential (capacity) on following discharge cycles, causing the cell to develop what's commonly referred to as a memory. Additionally, the capacity per weight (energy density) of NiCd cells is commonly less than NiMH or LiPo cell types as well. Finally, the Cadmium that is used in the cell is quite destructive to the environment, making disposal of NiCd cells an issue.

NiMH cells have many advantages over their NiCd counterparts. NiMH cell manufacturers are able to offer significantly higher capacities in cells approximately the same size and weight of equivalent NiCd cells. NiMH cells have an advantage when it comes to cell memory as well, as they do not develop the same issues as a result of inappropriate discharge care.

Lithium Polymer (LiPo) cells are one of the newest and most revolutionary battery cells Available. LiPo cells maintain a more consistent voltage over the discharge curve when compared to NiCd or NiMH cells. The higher nominal voltage of a single LiPo cell (3.7V vs. 1.2V for a typically NiCd or NiMH cell); making it possible to have an equivalent or even higher total nominal voltage in a much smaller package LiPo cells typically offer very high capacity for their weight, delivering upwards of twice the capacity for ½ the weight of comparable NiMH cells.

Lastly, a LiPo cell battery needs to be carefully monitored during charging since overcharging and the charging of a physically damaged or discharged cell can be a potential fire hazard and possibly even fatal.

LiPo Pro's:

- Highest power/weight ratio.
- Very low self-discharge.

- Less affected by low temperatures than some.

LiPo Con's:

- Intolerant of over-charging.
- Intolerant of over-discharging Battery.
- significant fire risk

Figure 21: 3S LiPo Battery

Table 3 : LiPo batteries 3S 11.1V 2600MAH 30C packs

Capacity	**2600mAh**
Configuration	3S1P
Dimensions	116X34X26mm
Weight	200g
Constant Discharge	30C
Burst Discharge	60C
Balance connector	ST-XHR
Discharge plug	T plug
Use	Vehicles & Remote-Control Toys
Material	EVA

4.2.7 The Brushless Motors

Each of the four rotors comprises of a Brushless DC Motor attached to a propeller. The Brushless motor differs from the conventional Brushed DC Motors in their concept essentially in that the commutation of the input voltage applied to the armature's circuit is done electronically, whereas in the latter, by a mechanical brush. As any rotating mechanical device, it suffers wear during operation, and as a consequence it has a shorter nominal life time than the newer Brushless motors.

In spite of the extra complexity in its electronic switching circuit, the brushless design offers several advantages over its counterpart, to name a few: higher torque/weight ratio, less operational noise, longer lifetime, less generation of electromagnetic interference and much more power per volume. Virtually limited only by its inherent heat generation, whose transfer to the outer environment usually occurs by conduction.

Figure 22: A2212/13T 1000 KV BLDC (Brushless DC Motor)

Table 4 : Specifications of A2212 / 920 KV out runner motor

No. of Cells:	2 - 3 Li-Poly 6 - 10 NiCd/NiMH
Kv:	1000 RPM/V
Max Efficiency:	80%
Max Efficiency Current:	4 - 10A (>75%)
No Load Current:	0.5A @10V
Resistance:	0.090 ohms
Max Current:	13A for 60S
Max Watts:	150W
Weight:	52.7 g / 1.86 oz.
Size:	28 mm diameter x 28 mm bell length
Shaft Diameter:	3.2 mm
Poles:	14
Model Weight:	300 - 800g / 10.5 - 28.2 oz.

4.2.8 Propellers

Propeller is a set of rotating blades design to convert the power (torque) of the Engine in to thrust.

The Quadrotor consists of four propellers coupled to the brushless motor. Among These four propellers, two clockwise and the remaining other two are counter clockwise. Clockwise and anticlockwise propellers cancel their torque from each other. Propellers are specified by their diameter and pitch. The propeller used is 1045 Fixed-pitch, symmetric, tapered Normal Rotation Carbon Fiber Propeller, shown in (figure):

Figure 23: 1045 fixed-pitch, Carbon fiber Propeller

4.3 Software Implementation

4.3.1 Quadcopter Flowchart

The operation flow of the quadcopter is illustrated in figure below demonstrating steps at which quadcopter flows in order to fly and satisfy pilot commands.

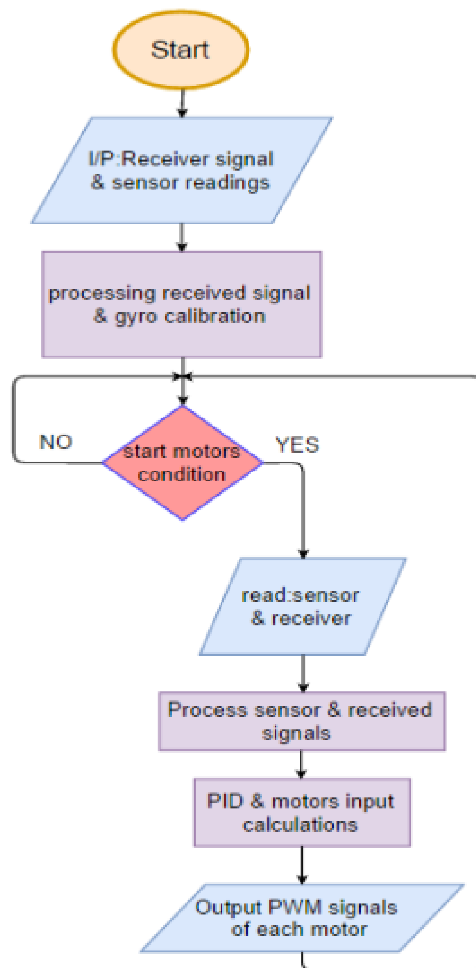


Figure 24: Quadcopter flow chart

4.3.2 Transmitter and Receiver

A four channel RC transmitter is used for the purpose of giving freedom to control throttle, pitch, roll and yaw individually. To obtain an accurate response set points and minimum and maximum ranges must be determined before transmission execution.

Since the main loop of the code executes sequentially - one line at a time- an interrupt needs to occur enabling receiving signals transmitted from the RC; Arduino allows pins to allow interrupt only if the interrupt for a specific pin was declared in the code.

Before declaring the interrupt pins, interrupt mode must be activated through the following syntax:

```
PCICR |= (1 << PCIE0);
```

After enabling interrupt mode four pins are declared as receiver interrupt pins each for each channel of the transmitter, the pins being Arduino pins 8, 9,10 and 11 declared as following:

```
PCMSK0 |= (1 << PCINT0);

PCMSK0 |= (1 << PCINT1);

PCMSK0 |= (1 << PCINT2);

PCMSK0 |= (1 << PCINT3);
```

4.3.3 Gyroscope

In order to determine the error, the actual quadcopter readings and the received signal needs to be compared with each other; the gyroscope is interfaced with an I2C interface- pronounced Isquared-C, is a multi-master, multi-slave, single-ended, serial computer bus invented- typically used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

To connect to the gyroscope the Wire Library is included in code allowing the Arduino to use the I2C; communications start by the master (Arduino) sending a tart bit followed by the 7-bit address of the slave (GY-85 with address 0x68) so that only gyroscope is chosen using the following statement:

```
Wire.beginTransmission (0x68);
```

Referring to the Gyroscope data-sheet set the gyro output scale to ±2000 deg/s by writing the value (3) decimal to the 3th and 4th bits of (22) gyroscope register.

```
Wire.write (22);              // calling the register of Full Scale

Wire.write (3<< 3);           // write 3 then shift it to left of Full Scale register

Wire.endTransmittion ();      // necessary to end each call to register
```

250 readings per second has to be obtained, we do so by setting the sampling rate of the gyroscope sampling rate register; the first three bits of register (22) are used for setting internal sampling rate with either 1KHz or 8KHz.

Register (21) can also be used for setting the sampling rate and is called Sampling Rate Divider Register which output is set to any values satisfying the equation:

$$F_{sample} = F_{Internal}/(divider + 1) \quad (31)$$

F_{sample} is the sample rate

F_{internal} is the internal rate determined by register (22) which is either (1KHz,8KHz)

Divider is determined by register (21)

Hence, to get 250 reading out of that gyro, reg. (22) need to be set to zero which is already the default value, the divider register (21) need to be set to 31 decimals, simply be writing that value to that register.

```
Wire.begingTransmittion (0x68);
Wire.write (21);
Wire.write (31);
Wire.endTransmittion ();
```

After configuring sensor register, the gyro is ready to provide readings through the registers (29-34) readings are ready in registers to be picked by the microcontroller any time. The readings collected usually have consistent off-set errors which differ in the value from an axis to another which are eliminated by calculating the average value for a fair amount of readings storing this into variables (gyro_roll_cal, gyro_pitch_cal, gyro_yaw_cal) these values are subtracted from each reading taken hence eliminating the gyro offset error.

The calibrated reading taken from gyroscope includes noise-measurement noise from propellers and motors- which can't be eliminated by eliminating the offset error but with the use of a very simple filter that was proven to provide accurate results.

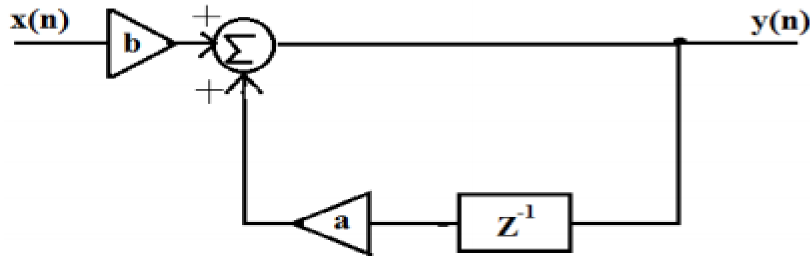


Figure 25: Recursive Filter Block Diagram

$$y(n) = ay(n-1) + bx(2) \qquad (32)$$

Where a = b – 1

The filter takes the input x (n) and sums it with the feedback y (n – 1); a and b are gains to be tuned to get the required output response. Applying filter to the three inputs of system (roll, pitch, yaw) will eliminate the noise and obtain the required response.

4.3.4 ESCs connection to Arduino

So far, the RC signal was transmitted, received and processed by the Arduino then the Gyro sensor provided the angular accelerations the Quadcopter which were filtered, calibrated and processed in the Arduino.

The Arduino output ports connected to the ESCs need to be declared before computing the total received input signal.

```
DDRD |= B11110000;
```

The output of the ESCs controls the motors depending on the PID output however the basic movement of the quadcopter is satisfied by the following equations passed as output values of the ESCs.

$$esc_1 = throttle - PID_{out_pitch} + PID_{out_roll} - PID_{out_yaw} \quad (33)$$

$$esc_2 = throttle + PID_{out_pitch} + PID_{out_roll} + PID_{out_yaw} \quad (34)$$

$$esc_3 = throttle + PID_{out_pitch} - PID_{out_roll} - PID_{out_yaw} \quad (35)$$

$$esc_4 = throttle - PID_{out_pitch} - PID_{out_roll} + PID_{out_yaw} \quad (36)$$

The positive and negative signs for the PID outputs in the ESCs equations are set according to the basic movements of the Quadcopter. Lastly tuning the PID gains (K_P, K_I, K_P) to provide a smooth and stable response of the Quadcopter.

Chapter Five

Results and Discussion

The results that will be discussed in this chapter will include notice of unnatural behavior by the quadcopter during construction and after unnatural here is defined as any error or fault that can endanger the safety of the quadcopter.

It has been noticed that the motors had a variation of speed resulting from the ESCs; the voltage supplied from the battery to the ESCs- with only the battery connected to the ESCs without any software code or even Arduino connected- vary from one ESC to the other, this was resolved by ESC calibration meaning all ESCs start the motors at the same time with the same speed on the condition that all ESCs has the same current rating, if the ESCs had different current rating overheat will if the rating of ESC is less than the others due because it will try to compensate the difference by operating the motor at a higher speed.

A power regulator must be used and the mounting, isolation and soldering of the components must be accurate and tight, at one point one of the ESCs experienced excessive overheat without and obvious reason however when the isolation was removed it was found that the soldering was loose; if the mounting of the propellers and motor is not tight a high degree of vibration occurs in the quadcopter and will also cause the propellers to detach itself from the quadcopter body.

A high vibration was clearly noticed in the propellers that was perceived at first to be a vibration problem so the propeller was re-mounted at the center of the propeller shaft and a square piece of duct tape was added to them as load to reduce vibration which reduced the vibration to a minimal value measured by computer code implemented to the quadcopter (Appendix-X).

The trial and error method in choosing the PID parameters that result in the stability of the quadcopter; the ease of control was noticed in the quadcopter however a negative roll angle kept occurring however a successful flight time of 53 seconds was recorded in the process of finding the suitable PID gains for a stable flight without any drifts.

Chapter Six

Conclusion and Recommendations

6.1 Conclusion

The research phase of the thesis aided in understanding the mathematical model of the quadcopter which is a step required before control and a background of the PID controller and it operation theory in order to transform its equations to equations that are applicable to the quadcopter and were implemented in the Arduino microcontroller.

The choice of the unity software was made upon the fact that it provided an environment with no limitation on executing and constructing a PID controller with code based on its basic equations and theory of operation.

In conclusion the construction of the circuit that connected the hardware components and implementation of the software was the initial work however troubleshooting and tuning the PID gains was challenging and the trial and error method proved to be a failure in choosing the proper PID parameters that provide stable flight without any offsets or deviations.

6.2 Recommendations

1. Another method should be used to choose the proper PID gains for a more stable flight
2. Wires can be connected with jacks instead of soldering wires together.
3. Power distribution boards are more helpful in mounting the IMU properly on the frame beneath the microcontroller and placing the battery safely.
4. Replacing the RC transmitter with a motion sensor that captures hand gestures transmitted as movement commands with a wireless Wi-Fi communication.

References

[1] Allison Ryan and J. Karl Hedrick (2005). “A mode-switching path planner for UAV-assisted search and rescue.” 44th IEEE Conference on Decision and Control, and the European Control Conference 2005.

[2] Kong Wai Weng (2011). “Quadcopter” Robot Head To Toe Magazine September 2011 Volume 3, pp. 1-3.

[3] Modeling and Control of mini flying machines, springer

[4] Young L. A., Aiken E. W., Johnson J. L., Demblewski R., Andrews J. and Klem J., “ New concepts and perspectives on micro-rotorcraft and small autonomous rotary-wing vehicles”, Proceedings of the 20th AIAA Applied Aerodynamics Conference, St. Louis, MO, 2002

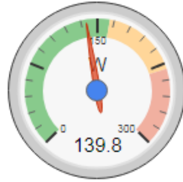
[5] Pounds P., Mahony R., Hynes P. and Roberts J., “Design of a four rotor aerial robot”, Proceedings of the Australasian Conference on Robotics and Automation, Auckland, Australia, 2002.

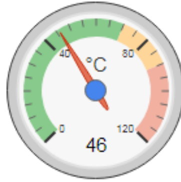
[6] Altu˘g E., Ostrowski J. P. and Mahony R., “Control of a quadrotor helicopter using visual feedback”, Proceedings of the 2002 IEEE International Conference on Robotics and Automation, ICRA 2002, May 11–15, 2002, Washington, DC

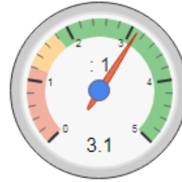
[7] https://www.arduino.cc/en/Guide/Introduction

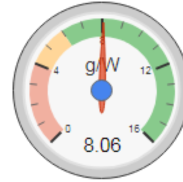
[8] en.wikipedia.org/wiki/PID_controller#PID_controller_theory

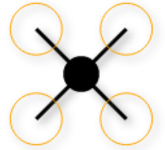
Appendix A: eCalc Hardware Analysis











Remarks:

Battery		
Load:	27.46	C
Voltage:	9.38	V
Rated Voltage:	11.10	V
Energy:	24.42	Wh
Total Capacity:	2200	mAh
Used Capacity:	1870	mAh
min. Flight Time:	1.9	min
Mixed Flight Time:	6.9	min
Hover Flight Time:	12.2	min
Weight:	165	g
	5.8	oz

Motor @ Optimum Efficiency		
Current:	15.69	A
Voltage:	9.19	V
Revolutions*:	8299	rpm
electric Power:	144.2	W
mech. Power:	117.0	W
Efficiency:	81.2	%

Motor @ Maximum		
Current:	15.10	A
Voltage:	9.26	V
Revolutions*:	8405	rpm
electric Power:	139.8	W
mech. Power:	113.4	W
Power-Weight:	699.0	W/kg
	317.1	W/lb
Efficiency:	81.1	%
est. Temperature:	46	°C
	115	°F

Wattmeter readings		
Current:	60.4	A
Voltage:	9.38	V
Power:	566.6	W

Motor @ Hover		
Current:	2.29	A
Voltage:	10.82	V
Revolutions*:	4050	rpm
Throttle (log):	30	%
Throttle (linear):	40	%
electric Power:	24.8	W
mech. Power:	18.0	W
Power-Weight:	127.3	W/kg
	57.7	W/lb
Efficiency:	72.4	%
est. Temperature:	38	°C
	100	°F
specific Thrust:	8.06	g/W
	0.28	oz/W

Total Drive		
Drive Weight:	745	g
	26.3	oz
Thrust-Weight:	3.1	: 1
Current @ Hover:	9.18	A
P(in) @ Hover:	101.8	W
P(out) @ Hover:	71.8	W
Efficiency @ Hover:	70.5	%
Current @ max:	60.41	A
P(in) @ max:	670.5	W
P(out) @ max:	453.5	W
Efficiency @ max:	67.6	%

Multicopter		
All-up Weight:	800	g
	28.2	oz
add. Payload:	1350	g
	47.6	oz
max Tilt:	68	°
max. Speed:	41	km/h
	25.5	mph
est. rate of climb:	8.8	m/s
	1732	ft/min
Total Disc Area:	20.27	dm²
	314.19	in²
with Rotor fail:	✖	

Appendix B: Quadcopter Simulation Code

B.1 PID Controller Simulation Code

```
using UnityEngine;
using System.Collections;

public class PIDController : MonoBehaviour
{
    float error_old = 0f;
    //The controller will be more robust if you are using a further ba
ck sample
    float error_old_2 = 0f;
    float error_sum = 0f;
    //If we want to average an error as input
    float error_sum2 = 0f;

    //PID parameters
    public float gain_P = 0f;
    public float gain_I = 0f;
    public float gain_D = 0f;
    //Sometimes you have to limit the total sum of all errors used in
the I
    private float error_sumMax = 20f;

    public float GetFactorFromPIDController(float error)
    {
        float output = CalculatePIDOutput(error);

        return output;
    }

    //Use this when experimenting with PID parameters
    public float GetFactorFromPIDController(float gain_P, float gain_I
, float gain_D, float error)
    {
        this.gain_P = gain_P;
        this.gain_I = gain_I;
        this.gain_D = gain_D;
```

```
        float output = CalculatePIDOutput(error);

        return output;
    }

    //Use this when experimenting with PID parameters and the gains ar
e stored in a Vector3
    public float GetFactorFromPIDController(Vector3 gains, float error
)
    {
        this.gain_P = gains.x;
        this.gain_I = gains.y;
        this.gain_D = gains.z;

        float output = CalculatePIDOutput(error);

        return output;
    }

    private float CalculatePIDOutput(float error)
    {
        //The output from PID
        float output = 0f;


        //P
        output += gain_P * error;


        //I
        error_sum += Time.fixedDeltaTime * error;

        //Clamp the sum
        this.error_sum = Mathf.Clamp(error_sum, -
error_sumMax, error_sumMax);

        //Sometimes better to just sum the last errors
        //float averageAmount = 20f;

        //CTE_sum = CTE_sum + ((CTE - CTE_sum) / averageAmount);

        output += gain_I * error_sum;


        //D
```

```
        float d_dt_error = (error - error_old) / Time.fixedDeltaTime;

        //Save the last errors
        this.error_old_2 = error_old;

        this.error_old = error;

        output += gain_D * d_dt_error;


        return output;
    }
}
```

B.2 Quadcopter simulation code

```
using UnityEngine;
using System.Collections;

public class QuadcopterController : MonoBehaviour
{
    //The propellers
    public GameObject propellerFR;
    public GameObject propellerFL;
    public GameObject propellerBL;
    public GameObject propellerBR;

    //Quadcopter parameters
    [Header("Internal")]
    public float maxPropellerForce; //100
    public float maxTorque; //1
    public float throttle;
    public float moveFactor; //5
    //PID
    public Vector3 PID_pitch_gains; //(2, 3, 2)
    public Vector3 PID_roll_gains; //(2, 0.2, 0.5)
    public Vector3 PID_yaw_gains; //(1, 0, 0)

    //External parameters
    [Header("External")]
    public float windForce;
    //0 -> 360
    public float forceDir;
```

```
Rigidbody quadcopterRB;


//The PID controllers
private PIDController PID_pitch;
private PIDController PID_roll;
private PIDController PID_yaw;

//Movement factors
float moveForwardBack;
float moveLeftRight;
float yawDir;

void Start()
{
    quadcopterRB = gameObject.GetComponent<Rigidbody>();

    PID_pitch = new PIDController();
    PID_roll = new PIDController();
    PID_yaw = new PIDController();
}

void FixedUpdate()
{
    AddControls();

    AddMotorForce();

    AddExternalForces();
}

void AddControls()
{
    //Change throttle to move up or down
    if (Input.GetKey(KeyCode.UpArrow))
    {
        throttle += 3f;
    }
    if (Input.GetKey(KeyCode.DownArrow))
    {
        throttle -= 3f;
    }

    throttle = Mathf.Clamp(throttle, 0f, 200f);
```

```
        //Steering
        //Move forward or reverse
        moveForwardBack = 0f;

        if (Input.GetKey(KeyCode.W))
        {
            moveForwardBack = 1f;
        }
        if (Input.GetKey(KeyCode.S))
        {
            moveForwardBack = -1f;
        }

        Mathf.Clamp (moveForwardBack, 0, 45f); //Clamping rot

        //Move left or right
        moveLeftRight = 0f;

        if (Input.GetKey(KeyCode.A))
        {
            moveLeftRight = -1f;
        }
        if (Input.GetKey(KeyCode.D))
        {
            moveLeftRight = 1f;
        }

        Mathf.Clamp (moveLeftRight, 0, 45f); //Clamping rot

        //Rotate around the axis
        yawDir = 0f;

        if (Input.GetKey(KeyCode.LeftArrow))
        {
            yawDir = -1f;
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            yawDir = 1f;
        }
    }

    void AddMotorForce()
    {
        //Calculate the errors so we can use a PID controller to stabi
lize
```

```
        //Assume no error is if 0 degrees

        //Pitch
        //Returns positive if pitching forward
        float pitchError = GetPitchError();

        //Roll
        //Returns positive if rolling left
        float rollError = GetRollError() * -1f;

        //Adapt the PID variables to the throttle
        Vector3 PID_pitch_gains_adapted = throttle > 100f ? PID_pitch_
gains * 2f : PID_pitch_gains;

        //Get the output from the PID controllers
        float PID_pitch_output = PID_pitch.GetFactorFromPIDController(
PID_pitch_gains_adapted, pitchError);
        float PID_roll_output = PID_roll.GetFactorFromPIDController(PI
D_roll_gains, rollError);

        //Calculate the propeller forces
        //FR
        float propellerForceFR = throttle + (PID_pitch_output + PID_ro
ll_output);

        //Add steering
        propellerForceFR -= moveForwardBack * throttle * moveFactor;
        propellerForceFR -= moveLeftRight * throttle;


        //FL
        float propellerForceFL = throttle + (PID_pitch_output -
 PID_roll_output);

        propellerForceFL -= moveForwardBack * throttle * moveFactor;
        propellerForceFL += moveLeftRight * throttle;


        //BR
        float propellerForceBR = throttle + (-
PID_pitch_output + PID_roll_output);

        propellerForceBR += moveForwardBack * throttle * moveFactor;
        propellerForceBR -= moveLeftRight * throttle;
```

```
        //BL
        float propellerForceBL = throttle + (-PID_pitch_output -
 PID_roll_output);

        propellerForceBL += moveForwardBack * throttle * moveFactor;
        propellerForceBL += moveLeftRight * throttle;


        //Clamp
        propellerForceFR = Mathf.Clamp(propellerForceFR, 0f, maxPropel
lerForce);
        propellerForceFL = Mathf.Clamp(propellerForceFL, 0f, maxPropel
lerForce);
        propellerForceBR = Mathf.Clamp(propellerForceBR, 0f, maxPropel
lerForce);
        propellerForceBL = Mathf.Clamp(propellerForceBL, 0f, maxPropel
lerForce);

        //Add the force to the propellers
        AddForceToPropeller(propellerFR, propellerForceFR);
        AddForceToPropeller(propellerFL, propellerForceFL);
        AddForceToPropeller(propellerBR, propellerForceBR);
        AddForceToPropeller(propellerBL, propellerForceBL);

        //Yaw
        //Minimize the yaw error (which is already signed):
        float yawError = quadcopterRB.angularVelocity.y;

        float PID_yaw_output = PID_yaw.GetFactorFromPIDController(PID_
yaw_gains, yawError);

        //First we need to add a force (if any)
        quadcopterRB.AddTorque(transform.up * yawDir * maxTorque * thr
ottle);

        //Then we need to minimize the error
        quadcopterRB.AddTorque(transform.up * throttle * PID_yaw_outpu
t * -1f);
    }

    void AddForceToPropeller(GameObject propellerObj, float propellerF
orce)
    {
        Vector3 propellerUp = propellerObj.transform.up;

        Vector3 propellerPos = propellerObj.transform.position;
```

```
        quadcopterRB.AddForceAtPosition(propellerUp * propellerForce,
propellerPos);

        //Debug
        //Debug.DrawRay(propellerPos, propellerUp * 1f, Color.red);
    }

    //Pitch is rotation around x-axis
    //Returns positive if pitching forward
    private float GetPitchError()
    {
        float xAngle = transform.eulerAngles.x;

        //Make sure the angle is between 0 and 360
        xAngle = WrapAngle(xAngle);

        //This angle going from 0 -> 360 when pitching forward
        //So if angle is > 180 then it should move from 0 to 180 if pi
tching back
        ///note: xAngle > 180f && xAngle < 360f
        if (xAngle > 30f && xAngle < 60f)
        {     //xAngle = 60 - xAngle
            = 60f - xAngle;

            //-1 so we know if we are pitching back or forward
            xAngle *= -1f;
        }

        return xAngle;
    }

    //Roll is rotation around z-axis
    //Returns positive if rolling left
    private float GetRollError()
    {
        float zAngle = transform.eulerAngles.z;

        //Make sure the angle is between 0 and 360
        zAngle = WrapAngle(zAngle);

        //This angle going from 0-> 360 when rolling left
        //So if angle is > 180 then it should move from 0 to 180 if ro
lling right
        ///note: zAngle > 180f && zAngle <c> 360f
        if (zAngle > 30f && zAngle < 60f)
```

```
        {
            zAngle = 360f - zAngle;

            //-1 so we know if we are rolling left or right
            zAngle *= -1f;
        }

        return zAngle;
    }

    //Wrap between 0 and 360 degrees
    float WrapAngle(float inputAngle)
    {
        //The inner % 360 restricts everything to +/- 360
        //+360 moves negative values to the positive range, and positi
ve ones to > 360
        //the final % 360 caps everything to 0...360
        return ((inputAngle % 360f) + 360f) % 360f;
    }

    //Add external forces to the quadcopter, such as wind
    private void AddExternalForces()
    {
        //Important to not use the quadcopters forward
        Vector3 windDir = -Vector3.forward;

        //Rotate it
        windDir = Quaternion.Euler(0, forceDir, 0) * windDir;

        quadcopterRB.AddForce(windDir * windForce);

        //Debug
        //Is showing in which direction the wind is coming from
        //center of quadcopter is where it ends and is blowing in the
direction of the line
        Debug.DrawRay(transform.position, -windDir * 3f, Color.red);
    }
}
```

Appendix C: Quadcopter Code

```
#include <Wire.h>                    //Include the Wire.h library so we can communicate with the
gyro.

#include <EEPROM.h>                  //Include the EEPROM.h library so we can store
information onto the EEPROM


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//PID gain and limit settings

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

float pid_p_gain_roll = 1.3;               //Gain setting for the roll P-controller

float pid_i_gain_roll = 0.04;              //Gain setting for the roll I-controller

float pid_d_gain_roll = 18.0;              //Gain setting for the roll D-controller

int pid_max_roll = 400;                    //Maximum output of the PID-controller (+/-)


float pid_p_gain_pitch = pid_p_gain_roll;  //Gain setting for the pitch P-controller.

float pid_i_gain_pitch = pid_i_gain_roll;  //Gain setting for the pitch I-controller.

float pid_d_gain_pitch = pid_d_gain_roll;  //Gain setting for the pitch D-controller.

int pid_max_pitch = pid_max_roll;          //Maximum output of the PID-controller (+/-)


float pid_p_gain_yaw = 4.0;                //Gain setting for the pitch P-controller. //4.0

float pid_i_gain_yaw = 0.02;               //Gain setting for the pitch I-controller. //0.02

float pid_d_gain_yaw = 0.0;                //Gain setting for the pitch D-controller.

int pid_max_yaw = 400;                     //Maximum output of the PID-controller (+/-)


boolean auto_level = true;                 //Auto level on (true) or off (false)
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Declaring global variables

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

byte last_channel_1, last_channel_2, last_channel_3, last_channel_4;

byte eeprom_data[36];

byte highByte, lowByte;

volatile int receiver_input_channel_1, receiver_input_channel_2, receiver_input_channel_3,
receiver_input_channel_4;

int counter_channel_1, counter_channel_2, counter_channel_3, counter_channel_4,
loop_counter;

int esc_1, esc_2, esc_3, esc_4;

int throttle, battery_voltage;

int cal_int, start, gyro_address;

int receiver_input[5];

int temperature;

int acc_axis[4], gyro_axis[4];

float roll_level_adjust, pitch_level_adjust;


long acc_x, acc_y, acc_z, acc_total_vector;

unsigned long timer_channel_1, timer_channel_2, timer_channel_3, timer_channel_4, esc_timer,
esc_loop_timer;

unsigned long timer_1, timer_2, timer_3, timer_4, current_time;

unsigned long loop_timer;

double gyro_pitch, gyro_roll, gyro_yaw;

double gyro_axis_cal[4];
```

```
float pid_error_temp;

float pid_i_mem_roll, pid_roll_setpoint, gyro_roll_input, pid_output_roll, pid_last_roll_d_error;

float pid_i_mem_pitch, pid_pitch_setpoint, gyro_pitch_input, pid_output_pitch,
pid_last_pitch_d_error;

float pid_i_mem_yaw, pid_yaw_setpoint, gyro_yaw_input, pid_output_yaw,
pid_last_yaw_d_error;

float angle_roll_acc, angle_pitch_acc, angle_pitch, angle_roll;

boolean gyro_angles_set;


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Setup routine

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void setup(){

  //Serial.begin(57600);

  //Copy the EEPROM data for fast access data.

  for(start = 0; start <= 35; start++)eeprom_data[start] = EEPROM.read(start);

  start = 0;                                                    //Set start back to zero.

  gyro_address = eeprom_data[32];                                       //Store the gyro address in the
variable.


  Wire.begin();                                                  //Start the I2C as master.


  TWBR = 12;                                                     //Set the I2C clock speed to 400kHz.


  //Arduino (Atmega) pins default to inputs, so they don't need to be explicitly declared as inputs.

  DDRD |= B11110000;                                             //Configure digital poort 4, 5, 6 and 7
as output.
```

```
  DDRB |= B00110000;                                           //Configure digital poort 12 and 13 as
output.

  //Use the led on the Arduino for startup indication.
  digitalWrite(12,HIGH);                                       //Turn on the warning led.

  //Check the EEPROM signature to make sure that the setup program is executed.
  while(eeprom_data[33] != 'J' || eeprom_data[34] != 'M' || eeprom_data[35] != 'B')delay(10);

  //The flight controller needs the MPU-6050 with gyro and accelerometer
  //If setup is completed without MPU-6050 stop the flight controller program
  if(eeprom_data[31] == 2 || eeprom_data[31] == 3)delay(10);

  set_gyro_registers();                                        //Set the specific gyro registers.

  for (cal_int = 0; cal_int < 1250 ; cal_int ++){              //Wait 5 seconds before
continuing.
    PORTD |= B11110000;                                        //Set digital poort 4, 5, 6 and 7 high.
    delayMicroseconds(1000);                                   //Wait 1000us.
    PORTD &= B00001111;                                        //Set digital poort 4, 5, 6 and 7 low.
    delayMicroseconds(3000);                                   //Wait 3000us.
  }

  //Let's take multiple gyro data samples so we can determine the average gyro offset
(calibration).
  for (cal_int = 0; cal_int < 2000 ; cal_int ++){              //Take 2000 readings for
calibration.
```

```
    if(cal_int % 15 == 0)digitalWrite(12, !digitalRead(12));               //Change the led status to
indicate calibration.
    gyro_signalen();                                              //Read the gyro output.
    gyro_axis_cal[1] += gyro_axis[1];                                     //Ad roll value to gyro_roll_cal.
    gyro_axis_cal[2] += gyro_axis[2];                                     //Ad pitch value to gyro_pitch_cal.
    gyro_axis_cal[3] += gyro_axis[3];                                     //Ad yaw value to gyro_yaw_cal.
    //We don't want the esc's to be beeping annoyingly. So let's give them a 1000us puls while
calibrating the gyro.
    PORTD |= B11110000;                                                 //Set digital poort 4, 5, 6 and 7 high.
    delayMicroseconds(1000);                                            //Wait 1000us.
    PORTD &= B00001111;                                                  //Set digital poort 4, 5, 6 and 7 low.
    delay(3);                                                  //Wait 3 milliseconds before the next loop.
  }
  //Now that we have 2000 measures, we need to devide by 2000 to get the average gyro offset.
  gyro_axis_cal[1] /= 2000;                                          //Divide the roll total by 2000.
  gyro_axis_cal[2] /= 2000;                                          //Divide the pitch total by 2000.
  gyro_axis_cal[3] /= 2000;                                          //Divide the yaw total by 2000.


  PCICR |= (1 << PCIE0);                                              //Set PCIE0 to enable PCMSK0 scan.
  PCMSK0 |= (1 << PCINT0);                                              //Set PCINT0 (digital input 8) to
trigger an interrupt on state change.
  PCMSK0 |= (1 << PCINT1);                                              //Set PCINT1 (digital input 9)to
trigger an interrupt on state change.
  PCMSK0 |= (1 << PCINT2);                                              //Set PCINT2 (digital input 10)to
trigger an interrupt on state change.
  PCMSK0 |= (1 << PCINT3);                                              //Set PCINT3 (digital input 11)to
trigger an interrupt on state change.
```

```
  //Wait until the receiver is active and the throtle is set to the lower position.
  while(receiver_input_channel_3 < 990 || receiver_input_channel_3 > 1020 ||
receiver_input_channel_4 < 1400){
    receiver_input_channel_3 = convert_receiver_channel(3);                //Convert the actual
receiver signals for throttle to the standard 1000 - 2000us
    receiver_input_channel_4 = convert_receiver_channel(4);                //Convert the actual
receiver signals for yaw to the standard 1000 - 2000us
    start ++;                                                    //While waiting increment start whith every
loop.
    //We don't want the esc's to be beeping annoyingly. So let's give them a 1000us puls while
waiting for the receiver inputs.
    PORTD |= B11110000;                                          //Set digital poort 4, 5, 6 and 7 high.
    delayMicroseconds(1000);                                     //Wait 1000us.
    PORTD &= B00001111;                                          //Set digital poort 4, 5, 6 and 7 low.
    delay(3);                                                    //Wait 3 milliseconds before the next loop.
    if(start == 125){                                            //Every 125 loops (500ms).
      digitalWrite(12, !digitalRead(12));                        //Change the led status.
      start = 0;                                                 //Start again at 0.
    }
  }
  start = 0;                                                     //Set start back to 0.

  //Load the battery voltage to the battery_voltage variable.
  //65 is the voltage compensation for the diode.
  //12.6V equals ~5V @ Analog 0.
  //12.6V equals 1023 analogRead(0).
```

```
  //1260 / 1023 = 1.2317.
  //The variable battery_voltage holds 1050 if the battery voltage is 10.5V.
  battery_voltage = (analogRead(0) + 65) * 1.2317;


  loop_timer = micros();                                                     //Set the timer for the next loop.


  //When everything is done, turn off the led.
  digitalWrite(12,LOW);                                                      //Turn off the warning led.
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Main program loop
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void loop(){


  //65.5 = 1 deg/sec (check the datasheet of the MPU-6050 for more information).
  gyro_roll_input = (gyro_roll_input * 0.7) + ((gyro_roll / 65.5) * 0.3);   //Gyro pid input is deg/sec.
  gyro_pitch_input = (gyro_pitch_input * 0.7) + ((gyro_pitch / 65.5) * 0.3);//Gyro pid input is deg/sec.
  gyro_yaw_input = (gyro_yaw_input * 0.7) + ((gyro_yaw / 65.5) * 0.3);      //Gyro pid input is deg/sec.
```

```
  //Gyro angle calculations
  //0.0000611 = 1 / (250Hz / 65.5)
  angle_pitch += gyro_pitch * 0.0000611;                    //Calculate the traveled pitch angle and add this to the angle_pitch variable.
  angle_roll += gyro_roll * 0.0000611;                      //Calculate the traveled roll angle and add this to the angle_roll variable.

  //0.000001066 = 0.0000611 * (3.142(PI) / 180degr) The Arduino sin function is in radians
  angle_pitch -= angle_roll * sin(gyro_yaw * 0.000001066);  //If the IMU has yawed transfer the roll angle to the pitch angel.
  angle_roll += angle_pitch * sin(gyro_yaw * 0.000001066);  //If the IMU has yawed transfer the pitch angle to the roll angel.

  //Accelerometer angle calculations
  acc_total_vector = sqrt((acc_x*acc_x)+(acc_y*acc_y)+(acc_z*acc_z));  //Calculate the total accelerometer vector.

  if(abs(acc_y) < acc_total_vector){                        //Prevent the asin function to produce a NaN
    angle_pitch_acc = asin((float)acc_y/acc_total_vector)* 57.296;   //Calculate the pitch angle.
  }
  if(abs(acc_x) < acc_total_vector){                        //Prevent the asin function to produce a NaN
    angle_roll_acc = asin((float)acc_x/acc_total_vector)* -57.296;   //Calculate the roll angle.
  }

  //Place the MPU-6050 spirit level and note the values in the following two lines for calibration.
```

```
  angle_pitch_acc -= 0.0;                                                       //Accelerometer calibration value for pitch.
  angle_roll_acc -= 0.0;                                                        //Accelerometer calibration value for roll.

  angle_pitch = angle_pitch * 0.9996 + angle_pitch_acc * 0.0004;            //Correct the drift of the gyro pitch angle with the accelerometer pitch angle.
  angle_roll = angle_roll * 0.9996 + angle_roll_acc * 0.0004;               //Correct the drift of the gyro roll angle with the accelerometer roll angle.

  pitch_level_adjust = angle_pitch * 15;                                    //Calculate the pitch angle correction
  roll_level_adjust = angle_roll * 15;                                      //Calculate the roll angle correction

  if(!auto_level){                                                          //If the quadcopter is not in auto-level mode
    pitch_level_adjust = 0;                                                 //Set the pitch angle correction to zero.
    roll_level_adjust = 0;                                                  //Set the roll angle correcion to zero.
  }



  //For starting the motors: throttle low and yaw left (step 1).
  if(receiver_input_channel_3 < 1050 && receiver_input_channel_4 < 1050)start = 1;
  //When yaw stick is back in the center position start the motors (step 2).
  if(start == 1 && receiver_input_channel_3 < 1050 && receiver_input_channel_4 > 1450){
    start = 2;

    angle_pitch = angle_pitch_acc;                                          //Set the gyro pitch angle equal to the accelerometer pitch angle when the quadcopter is started.
```

```
    angle_roll = angle_roll_acc;                                    //Set the gyro roll angle equal to the accelerometer roll angle when the quadcopter is started.
    gyro_angles_set = true;                                         //Set the IMU started flag.


    //Reset the PID controllers for a bumpless start.
    pid_i_mem_roll = 0;
    pid_last_roll_d_error = 0;
    pid_i_mem_pitch = 0;
    pid_last_pitch_d_error = 0;
    pid_i_mem_yaw = 0;
    pid_last_yaw_d_error = 0;
  }
  //Stopping the motors: throttle low and yaw right.
  if(start == 2 && receiver_input_channel_3 < 1050 && receiver_input_channel_4 > 1950)start = 0;


  //The PID set point in degrees per second is determined by the roll receiver input.
  //In the case of deviding by 3 the max roll rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).
  pid_roll_setpoint = 0;
  //We need a little dead band of 16us for better results.
  if(receiver_input_channel_1 > 1508)pid_roll_setpoint = receiver_input_channel_1 - 1508;
  else if(receiver_input_channel_1 < 1492)pid_roll_setpoint = receiver_input_channel_1 - 1492;


  pid_roll_setpoint -= roll_level_adjust;                           //Subtract the angle correction from the standardized receiver roll input value.
```

```
  pid_roll_setpoint /= 3.0;                                    //Divide the setpoint for the PID roll controller by 3 to get angles in degrees.


  //The PID set point in degrees per second is determined by the pitch receiver input.
  //In the case of deviding by 3 the max pitch rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).
  pid_pitch_setpoint = 0;
  //We need a little dead band of 16us for better results.
  if(receiver_input_channel_2 > 1508)pid_pitch_setpoint = receiver_input_channel_2 - 1508;
  else if(receiver_input_channel_2 < 1492)pid_pitch_setpoint = receiver_input_channel_2 - 1492;

  pid_pitch_setpoint -= pitch_level_adjust;                    //Subtract the angle correction from the standardized receiver pitch input value.
  pid_pitch_setpoint /= 3.0;                                   //Divide the setpoint for the PID pitch controller by 3 to get angles in degrees.

  //The PID set point in degrees per second is determined by the yaw receiver input.
  //In the case of deviding by 3 the max yaw rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).
  pid_yaw_setpoint = 0;
  //We need a little dead band of 16us for better results.
  if(receiver_input_channel_3 > 1050){ //Do not yaw when turning off the motors.
    if(receiver_input_channel_4 > 1508)pid_yaw_setpoint = (receiver_input_channel_4 - 1508)/3.0;
    else if(receiver_input_channel_4 < 1492)pid_yaw_setpoint = (receiver_input_channel_4 - 1492)/3.0;
```

```
  }


  calculate_pid();                                          //PID inputs are known. So we can
calculate the pid output.


  //The battery voltage is needed for compensation.
  //A complementary filter is used to reduce noise.
  //0.09853 = 0.08 * 1.2317.
  battery_voltage = battery_voltage * 0.92 + (analogRead(0) + 65) * 0.09853;


  //Turn on the led if battery voltage is to low.
  if(battery_voltage < 1000 && battery_voltage > 600)digitalWrite(12, HIGH);




  throttle = receiver_input_channel_3;                           //We need the throttle signal as a
base signal.


  if (start == 2){                                           //The motors are started.
    if (throttle > 1800) throttle = 1800;                        //We need some room to keep full
control at full throttle.
    esc_1 = throttle - pid_output_pitch + pid_output_roll - pid_output_yaw; //Calculate the pulse
for esc 1 (front-right - CCW)
    esc_2 = throttle + pid_output_pitch + pid_output_roll + pid_output_yaw; //Calculate the pulse
for esc 2 (rear-right - CW)
    esc_3 = throttle + pid_output_pitch - pid_output_roll - pid_output_yaw; //Calculate the pulse
for esc 3 (rear-left - CCW)
    esc_4 = throttle - pid_output_pitch - pid_output_roll + pid_output_yaw; //Calculate the pulse
for esc 4 (front-left - CW)
```

```
    if (battery_voltage < 1240 && battery_voltage > 800){                   //Is the battery connected?
      esc_1 += esc_1 * ((1240 - battery_voltage)/(float)3500);             //Compensate the esc-1
pulse for voltage drop.
      esc_2 += esc_2 * ((1240 - battery_voltage)/(float)3500);             //Compensate the esc-2
pulse for voltage drop.
      esc_3 += esc_3 * ((1240 - battery_voltage)/(float)3500);             //Compensate the esc-3
pulse for voltage drop.
      esc_4 += esc_4 * ((1240 - battery_voltage)/(float)3500);             //Compensate the esc-4
pulse for voltage drop.
    }

    if (esc_1 < 1100) esc_1 = 1100;                                //Keep the motors running.
    if (esc_2 < 1100) esc_2 = 1100;                                //Keep the motors running.
    if (esc_3 < 1100) esc_3 = 1100;                                //Keep the motors running.
    if (esc_4 < 1100) esc_4 = 1100;                                //Keep the motors running.

    if(esc_1 > 2000)esc_1 = 2000;                                  //Limit the esc-1 pulse to 2000us.
    if(esc_2 > 2000)esc_2 = 2000;                                  //Limit the esc-2 pulse to 2000us.
    if(esc_3 > 2000)esc_3 = 2000;                                  //Limit the esc-3 pulse to 2000us.
    if(esc_4 > 2000)esc_4 = 2000;                                  //Limit the esc-4 pulse to 2000us.
  }

  else{
    esc_1 = 1000;                                             //If start is not 2 keep a 1000us pulse for
ess-1.
```

```
    esc_2 = 1000;                                           //If start is not 2 keep a 1000us pulse for
ess-2.

    esc_3 = 1000;                                           //If start is not 2 keep a 1000us pulse for
ess-3.

    esc_4 = 1000;                                           //If start is not 2 keep a 1000us pulse for
ess-4.

  }


  if(micros() - loop_timer > 4050)digitalWrite(12, HIGH);                  //Turn on the LED if the
loop time exceeds 4050us.


  //All the information for controlling the motor's is available.

  //The refresh rate is 250Hz. That means the esc's need there pulse every 4ms.

  while(micros() - loop_timer < 4000);                                //We wait until 4000us are passed.

  loop_timer = micros();                                         //Set the timer for the next loop.


  PORTD |= B11110000;                                           //Set digital outputs 4,5,6 and 7 high.

  timer_channel_1 = esc_1 + loop_timer;                             //Calculate the time of the faling
edge of the esc-1 pulse.

  timer_channel_2 = esc_2 + loop_timer;                             //Calculate the time of the faling
edge of the esc-2 pulse.

  timer_channel_3 = esc_3 + loop_timer;                             //Calculate the time of the faling
edge of the esc-3 pulse.

  timer_channel_4 = esc_4 + loop_timer;                             //Calculate the time of the faling
edge of the esc-4 pulse.


  //There is always 1000us of spare time. So let's do something usefull that is very time
consuming.
```

```
  //Get the current gyro and receiver data and scale it to degrees per second for the pid calculations.

  gyro_signalen();



  while(PORTD >= 16){                                                       //Stay in this loop until output 4,5,6 and 7 are low.

    esc_loop_timer = micros();                                              //Read the current time.

    if(timer_channel_1 <= esc_loop_timer)PORTD &= B11101111;                //Set digital output 4 to low if the time is expired.

    if(timer_channel_2 <= esc_loop_timer)PORTD &= B11011111;                //Set digital output 5 to low if the time is expired.

    if(timer_channel_3 <= esc_loop_timer)PORTD &= B10111111;                //Set digital output 6 to low if the time is expired.

    if(timer_channel_4 <= esc_loop_timer)PORTD &= B01111111;                //Set digital output 7 to low if the time is expired.

  }

}



ISR(PCINT0_vect){

  current_time = micros();

  //Channel 1=========================================

  if(PINB & B00000001){                                                     //Is input 8 high?

    if(last_channel_1 == 0){                                                //Input 8 changed from 0 to 1.

      last_channel_1 = 1;                                                   //Remember current input state.

      timer_1 = current_time;                                               //Set timer_1 to current_time.

    }

  }
```

```
  else if(last_channel_1 == 1){                                    //Input 8 is not high and changed from 1 to 0.
    last_channel_1 = 0;                                            //Remember current input state.
    receiver_input[1] = current_time - timer_1;                    //Channel 1 is current_time - timer_1.
  }
  //Channel 2=========================================
  if(PINB & B00000010 ){                                           //Is input 9 high?
    if(last_channel_2 == 0){                                       //Input 9 changed from 0 to 1.
      last_channel_2 = 1;                                          //Remember current input state.
      timer_2 = current_time;                                      //Set timer_2 to current_time.
    }
  }
  else if(last_channel_2 == 1){                                    //Input 9 is not high and changed from 1 to 0.
    last_channel_2 = 0;                                            //Remember current input state.
    receiver_input[2] = current_time - timer_2;                    //Channel 2 is current_time - timer_2.
  }
  //Channel 3=========================================
  if(PINB & B00000100 ){                                           //Is input 10 high?
    if(last_channel_3 == 0){                                       //Input 10 changed from 0 to 1.
      last_channel_3 = 1;                                          //Remember current input state.
      timer_3 = current_time;                                      //Set timer_3 to current_time.
    }
  }
```

```
  else if(last_channel_3 == 1){                                  //Input 10 is not high and changed
from 1 to 0.

    last_channel_3 = 0;                                          //Remember current input state.

    receiver_input[3] = current_time - timer_3;                  //Channel 3 is current_time -
timer_3.


  }

  //Channel 4=========================================

  if(PINB & B00001000 ){                                         //Is input 11 high?

    if(last_channel_4 == 0){                                     //Input 11 changed from 0 to 1.

      last_channel_4 = 1;                                        //Remember current input state.

      timer_4 = current_time;                                    //Set timer_4 to current_time.

    }

  }

  else if(last_channel_4 == 1){                                  //Input 11 is not high and changed
from 1 to 0.

    last_channel_4 = 0;                                          //Remember current input state.

    receiver_input[4] = current_time - timer_4;                  //Channel 4 is current_time -
timer_4.

  }

}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Subroutine for reading the gyro

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void gyro_signalen(){

  //Read the MPU-6050
```

```
if(eeprom_data[31] == 1){

   Wire.beginTransmission(gyro_address);                                    //Start communication with the gyro.

   Wire.write(0x3B);                                                 //Start reading @ register 43h and auto increment with every read.

   Wire.endTransmission();                                               //End the transmission.

   Wire.requestFrom(gyro_address,14);                                      //Request 14 bytes from the gyro.


   receiver_input_channel_1 = convert_receiver_channel(1);                 //Convert the actual receiver signals for pitch to the standard 1000 - 2000us.

   receiver_input_channel_2 = convert_receiver_channel(2);                 //Convert the actual receiver signals for roll to the standard 1000 - 2000us.

   receiver_input_channel_3 = convert_receiver_channel(3);                 //Convert the actual receiver signals for throttle to the standard 1000 - 2000us.

   receiver_input_channel_4 = convert_receiver_channel(4);                 //Convert the actual receiver signals for yaw to the standard 1000 - 2000us.


   while(Wire.available() < 14);                                          //Wait until the 14 bytes are received.

   acc_axis[1] = Wire.read()<<8|Wire.read();                               //Add the low and high byte to the acc_x variable.

   acc_axis[2] = Wire.read()<<8|Wire.read();                               //Add the low and high byte to the acc_y variable.

   acc_axis[3] = Wire.read()<<8|Wire.read();                               //Add the low and high byte to the acc_z variable.

   temperature = Wire.read()<<8|Wire.read();                               //Add the low and high byte to the temperature variable.

   gyro_axis[1] = Wire.read()<<8|Wire.read();                              //Read high and low part of the angular data.

   gyro_axis[2] = Wire.read()<<8|Wire.read();                              //Read high and low part of the angular data.
```

```
    gyro_axis[3] = Wire.read()<<8|Wire.read();                             //Read high and low part of the
angular data.

  }


  if(cal_int == 2000){

    gyro_axis[1] -= gyro_axis_cal[1];                                     //Only compensate after the
calibration.

    gyro_axis[2] -= gyro_axis_cal[2];                                     //Only compensate after the
calibration.

    gyro_axis[3] -= gyro_axis_cal[3];                                     //Only compensate after the
calibration.

  }

  gyro_roll = gyro_axis[eeprom_data[28] & 0b00000011];                     //Set gyro_roll to the
correct axis that was stored in the EEPROM.

  if(eeprom_data[28] & 0b10000000)gyro_roll *= -1;                        //Invert gyro_roll if the
MSB of EEPROM bit 28 is set.

  gyro_pitch = gyro_axis[eeprom_data[29] & 0b00000011];                    //Set gyro_pitch to the
correct axis that was stored in the EEPROM.

  if(eeprom_data[29] & 0b10000000)gyro_pitch *= -1;                       //Invert gyro_pitch if the
MSB of EEPROM bit 29 is set.

  gyro_yaw = gyro_axis[eeprom_data[30] & 0b00000011];                      //Set gyro_yaw to the
correct axis that was stored in the EEPROM.

  if(eeprom_data[30] & 0b10000000)gyro_yaw *= -1;                         //Invert gyro_yaw if the
MSB of EEPROM bit 30 is set.


  acc_x = acc_axis[eeprom_data[29] & 0b00000011];                         //Set acc_x to the correct
axis that was stored in the EEPROM.

  if(eeprom_data[29] & 0b10000000)acc_x *= -1;                            //Invert acc_x if the MSB of
EEPROM bit 29 is set.
```

```
  acc_y = acc_axis[eeprom_data[28] & 0b00000011];                      //Set acc_y to the correct axis that was stored in the EEPROM.

  if(eeprom_data[28] & 0b10000000)acc_y *= -1;                         //Invert acc_y if the MSB of EEPROM bit 28 is set.

  acc_z = acc_axis[eeprom_data[30] & 0b00000011];                      //Set acc_z to the correct axis that was stored in the EEPROM.

  if(eeprom_data[30] & 0b10000000)acc_z *= -1;                         //Invert acc_z if the MSB of EEPROM bit 30 is set.

}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Subroutine for calculating pid outputs

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void calculate_pid(){

  //Roll calculations

  pid_error_temp = gyro_roll_input - pid_roll_setpoint;

  pid_i_mem_roll += pid_i_gain_roll * pid_error_temp;

  if(pid_i_mem_roll > pid_max_roll)pid_i_mem_roll = pid_max_roll;

  else if(pid_i_mem_roll < pid_max_roll * -1)pid_i_mem_roll = pid_max_roll * -1;


  pid_output_roll = pid_p_gain_roll * pid_error_temp + pid_i_mem_roll + pid_d_gain_roll * (pid_error_temp - pid_last_roll_d_error);

  if(pid_output_roll > pid_max_roll)pid_output_roll = pid_max_roll;

  else if(pid_output_roll < pid_max_roll * -1)pid_output_roll = pid_max_roll * -1;


  pid_last_roll_d_error = pid_error_temp;
```

```
  //Pitch calculations
  pid_error_temp = gyro_pitch_input - pid_pitch_setpoint;
  pid_i_mem_pitch += pid_i_gain_pitch * pid_error_temp;
  if(pid_i_mem_pitch > pid_max_pitch)pid_i_mem_pitch = pid_max_pitch;
  else if(pid_i_mem_pitch < pid_max_pitch * -1)pid_i_mem_pitch = pid_max_pitch * -1;

  pid_output_pitch = pid_p_gain_pitch * pid_error_temp + pid_i_mem_pitch + pid_d_gain_pitch
* (pid_error_temp - pid_last_pitch_d_error);
  if(pid_output_pitch > pid_max_pitch)pid_output_pitch = pid_max_pitch;
  else if(pid_output_pitch < pid_max_pitch * -1)pid_output_pitch = pid_max_pitch * -1;

  pid_last_pitch_d_error = pid_error_temp;

  //Yaw calculations
  pid_error_temp = gyro_yaw_input - pid_yaw_setpoint;
  pid_i_mem_yaw += pid_i_gain_yaw * pid_error_temp;
  if(pid_i_mem_yaw > pid_max_yaw)pid_i_mem_yaw = pid_max_yaw;
  else if(pid_i_mem_yaw < pid_max_yaw * -1)pid_i_mem_yaw = pid_max_yaw * -1;

  pid_output_yaw = pid_p_gain_yaw * pid_error_temp + pid_i_mem_yaw + pid_d_gain_yaw *
(pid_error_temp - pid_last_yaw_d_error);
  if(pid_output_yaw > pid_max_yaw)pid_output_yaw = pid_max_yaw;
  else if(pid_output_yaw < pid_max_yaw * -1)pid_output_yaw = pid_max_yaw * -1;

  pid_last_yaw_d_error = pid_error_temp;
}
```

```
//This part converts the actual receiver signals to a standardized 1000 – 1500 – 2000 microsecond value.

//The stored data in the EEPROM is used.

int convert_receiver_channel(byte function){

  byte channel, reverse;                                                //First we declare some local variables

  int low, center, high, actual;

  int difference;


  channel = eeprom_data[function + 23] & 0b00000111;                              //What channel corresponds with the specific function

  if(eeprom_data[function + 23] & 0b10000000)reverse = 1;                      //Reverse channel when most significant bit is set

  else reverse = 0;                                                   //If the most significant is not set there is no reverse


  actual = receiver_input[channel];                                            //Read the actual receiver value for the corresponding function

  low = (eeprom_data[channel * 2 + 15] << 8) | eeprom_data[channel * 2 + 14];  //Store the low value for the specific receiver input channel

  center = (eeprom_data[channel * 2 - 1] << 8) | eeprom_data[channel * 2 - 2]; //Store the center value for the specific receiver input channel

  high = (eeprom_data[channel * 2 + 7] << 8) | eeprom_data[channel * 2 + 6];   //Store the high value for the specific receiver input channel


  if(actual < center){                                                 //The actual receiver value is lower than the center value

    if(actual < low)actual = low;                                      //Limit the lowest value to the value that was detected during setup
```

```
    difference = ((long)(center - actual) * (long)500) / (center - low);       //Calculate and scale the
actual value to a 1000 - 2000us value
    if(reverse == 1)return 1500 + difference;                              //If the channel is reversed
    else return 1500 - difference;                                         //If the channel is not reversed
  }
  else if(actual > center){                                                //The actual receiver value
is higher than the center value
    if(actual > high)actual = high;                                        //Limit the lowest value to the value
that was detected during setup
    difference = ((long)(actual - center) * (long)500) / (high - center);      //Calculate and scale the
actual value to a 1000 - 2000us value
    if(reverse == 1)return 1500 - difference;                              //If the channel is reversed
    else return 1500 + difference;                                         //If the channel is not reversed
  }
  else return 1500;
}

void set_gyro_registers(){
  //Setup the MPU-6050
  if(eeprom_data[31] == 1){
    Wire.beginTransmission(gyro_address);                                  //Start communication with
the address found during search.
    Wire.write(0x6B);                                                      //We want to write to the
PWR_MGMT_1 register (6B hex)
    Wire.write(0x00);                                                      //Set the register bits as 00000000 to
activate the gyro
    Wire.endTransmission();                                                //End the transmission with the gyro.
```

```
  Wire.beginTransmission(gyro_address);                                    //Start communication with
the address found during search.
  Wire.write(0x1B);                                                        //We want to write to the
GYRO_CONFIG register (1B hex)
  Wire.write(0x08);                                                        //Set the register bits as 00001000
(500dps full scale)
  Wire.endTransmission();                                                  //End the transmission with the gyro

  Wire.beginTransmission(gyro_address);                                    //Start communication with
the address found during search.
  Wire.write(0x1C);                                                        //We want to write to the
ACCEL_CONFIG register (1A hex)
  Wire.write(0x10);                                                        //Set the register bits as 00010000 (+/-
8g full scale range)
  Wire.endTransmission();                                                  //End the transmission with the gyro

  //Let's perform a random register check to see if the values are written correct
  Wire.beginTransmission(gyro_address);                                    //Start communication with
the address found during search
  Wire.write(0x1B);                                                        //Start reading @ register 0x1B
  Wire.endTransmission();                                                  //End the transmission
  Wire.requestFrom(gyro_address, 1);                                       //Request 1 bytes from the gyro
  while(Wire.available() < 1);                                             //Wait until the 6 bytes are received
  if(Wire.read() != 0x08){                                                 //Check if the value is 0x08
    digitalWrite(12,HIGH);                                                 //Turn on the warning led
    while(1)delay(10);                                                     //Stay in this loop for ever
  }
```

```
    Wire.beginTransmission(gyro_address);                              //Start communication with the address found during search
    Wire.write(0x1A);                                                  //We want to write to the CONFIG register (1A hex)
    Wire.write(0x03);                                                  //Set the register bits as 00000011 (Set Digital Low Pass Filter to ~43Hz)
    Wire.endTransmission();                                            //End the transmission with the gyro

  }
}
```