IMPLEMENTATION OF DUCTED FANS ON A LOW COST
SEMI-AUTONOMOUS LIGHTER-TAN-AIR UNINHABITED AERIAL
VEHICLE

A Thesis in
Aerospace Engineering
by
Ilker Oruc

© 2013 Ilker Oruc

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2013
The thesis of Ilker Oruc was reviewed and approved* by the following:

Cengiz Camci
Professor of Aerospace Engineering
Thesis Advisor

Savas Yavuzkurt
Professor of Mechanical Engineering

George A. Lesieutre
Head of the Department of Aerospace Engineering

*Signatures are on file in the Graduate School.
Abstract

This thesis aims to develop an uninhabited air vehicle for surveillance and monitoring missions, using electric ducted fans, Arduino on-board controller and several low cost sensor. The platform design is based on a small lighter-than-air vehicle. The system is expected to provide the ability of autonomous flight with several control features such as Altitude Controller, Direction Holder and Speed Controller. A 10 foot(3.0 m) buoyant blimp is chosen with a configuration of tilt-able ducted fans, one horizontal and vertical tail with control surfaces(elevator and rudder) and a gondola to carry main board, sensors and batteries. A telemetry system is added in order to send the real-time sensory input to the ground station and receive manual control commands from the ground station during the flight. The micro-controler on the board is programmed using the Arduino programming language, based on Wiring and several open source libraries are written with object oriented C++ programming language. PID controller method with an auto-tune feature is implemented to the code for a smooth desirable trajectory. The controller output is used to adjust tilt-able ducted fans parameters such as thrust and angular position. Kalman filter is applied to get rid of noisy data from the motion sensors. A small prototype is manufactured and several indoor flight test results are presented.
Table of Contents

List of Figures vii
List of Tables viii
Acknowledgments ix

Chapter 1
Introduction 1
1.1 History of Blimps ........................................ 2
1.2 Similar Projects ........................................ 2
  1.2.1 Research and Development Projects ............... 2
  1.2.2 Open Source Projects ............................. 4
    1.2.2.1 ArduCopter .................................. 4
    1.2.2.2 ArduBlimp ................................... 5
    1.2.2.3 Paparazzi ................................... 5
1.3 Project Objectives ................................... 5
1.4 Methodology ........................................ 5

Chapter 2
System Description and Prototyping 7
2.1 Vehicle Configuration .................................. 7
  2.1.1 Two Rotors Configuration .......................... 8
  2.1.2 Four Rotors Configuration ......................... 8
2.2 Vehicle Manufacture & Assembly ...................... 12
  2.2.1 Small Scale Prototype Manufacture ............... 12
    2.2.1.1 Blimp ...................................... 12
    2.2.1.2 Gondola .................................... 13
    2.2.1.3 Stabilizer Tail Fins ......................... 13
  2.2.2 Electronic Board Design & Manufacture .......... 14
    2.2.2.1 Motor Controller Circuit .................... 14
    2.2.2.2 Sensor Circuit for Small Scale Prototype .... 15
# List of Figures

1.1 Hindenburg disaster in 1937 ............................................. 3  
1.2 System components of the Project AURORA ........................ 4  
1.3 A view from ArduCopter Mission Planner User Interface ......... 4  
2.1 Two Rotors Vehicle Configuration .................................... 9  
2.2 Four Rotors Vehicle Configuration .................................. 10  
2.3 A picture of target blimp that is chosen ............................. 11  
2.4 A picture of blimp that is chosen for small prototype .......... 13  
2.5 A picture of the gondola that is manufactured ................... 14  
2.6 Stabilizer Tail Fins ..................................................... 14  
2.7 Picture of the small scale envelope chosen for prototype ...... 15  
2.8 TIP120 NPN Power Darlington Transistor Scheme ............... 15  
2.9 Circuit Board Wiring Scheme ........................................ 16  
2.10 Sensor Board wiring scheme ......................................... 16  
3.1 Arduino Mega 2560 ....................................................... 18  
3.2 BMP085 Barometric Pressure Sensor ................................ 20  
3.3 Triple Axis Accelerometer Breakout - ADXL335 ................. 20  
3.4 Triple-Axis Digital-Output Gyro ITG-3200 Breakout ............. 21  
3.5 Triple-Axis Tilt Compensated Compass- LsM303 Breakout Board 22  
3.6 Air Speed Sensor Kit .................................................. 23  
3.7 20 Channel EM-406A SiRF III Receiver with Antenna ........... 24  
3.8 XBee Pro 60mW Wire Antenna - Series 1 (802.15.4) ............. 25  
4.1 Vehicle Body Coordinates ............................................. 27  
4.2 Attitude Estimation Algorithm ...................................... 28  
4.3 Heading vector representation w/o tilt angle .................... 32  
4.4 Heading vector representation w/ tilt angle ..................... 33  
4.5 Compass Heading System ............................................. 34  
4.6 Altitude Change in Standard Atmosphere .......................... 34  
4.7 Altitude Calculation .................................................. 35  
4.8 Airspeed Sensor Calibration Result .................................. 36  
4.9 Altitude Holder Control Diagram ................................... 39  
4.10 Direction Holder Block Diagram ................................... 41  
4.11 Speed Controller Block Diagram ................................... 42  
5.1 Direction Holder Calibration Test - Throttle Position: % 30, Target Offset: $\pm 5^\circ$ 45  
5.2 Direction Holder Calibration Test - Throttle Position: % 30, Target Offset: $\pm 1^\circ$ 46  

vi
5.3 Direction Holder Calibration Test - Throttle Position: % 50, Target Offset: ±5° . 47
5.4 Direction Holder Calibration Test - Throttle Position: % 50, Target Offset: ±1° . 47
5.5 Direction Holder Calibration Test - All Cases . . . . . . . . . . . . . . . . . . . . . 48
List of Tables

2.1 Two Rotors Configuration ................................................. 8
2.2 Four Rotors Vehicle Configuration ...................................... 8
2.3 Specifications of Envelope .................................................. 11
2.4 Gas Properties at Sea Level ............................................... 12
2.5 Target Blimp Configuration Weight Comparison ...................... 12

3.1 Technical Specifications of Arduino Mega 2560 ..................... 19
3.2 Total Cost of Avionics ....................................................... 25
Acknowledgments

I would like to thank my thesis advisor Dr. Cengiz Camci for his academic guidance and advice. I could not have completed this thesis without his support and opportunities that he gave me.

Also I would like thank all my friends, colleagues and my family for their supports during writing this thesis.
Dedication

To my family...
Chapter 1

Introduction

Blimp, commonly known as a non-rigid airship without an internal solid structure. Non-existence of a solid internal structure differs the blimps from the semi rigid and rigid airship like zeppelin. Like hot air balloons, blimps fly using the lifting force of a special gas, lighter than air usually helium, rather than hydrogen. But unlike a hot balloon, blimp has a shape and structure so it can move forward through the air under their own power, like an aircraft[1]. They can also hover like a helicopter.

In the past, blimps had been used in many applications such as commercial, military and cargo transportation. The advantage of high payload capability and long range & endurance flights made blimps very popular during World War I. But since they were so vulnerable to enemy aircraft, blimps lost their popularity on military applications. Especially, developments on heavier-than-air aircraft had been the most important reason to lost their popularity. Although, increasing energy costs forced the aerospace industry to return their attentions to these lighter-than-air vessels once again. Specifically, recent developments on Uninhabited Aerial Vehicles(UAV) systems, brought out the idea of using blimps as an UAV platform.

Today, UAVs have a wide spectrum of application potential for commercial and civilian purposes such as observation and data acquisition, besides their use of military surveillance platforms[2]. All these applications requires an air vehicle with low/high altitude and long endurance flight capabilities. With today’s advanced robust flight technology and light weight & more durable materials, blimps are a powerful alternative as an UAV platform. They have high altitude and long endurance flight capabilities compared to the heavier-than-air platforms such as helicopters and fixed wings. Also blimps don’t need a lift generator like an engine to fly, to hover and loiter. Flight efficiency can be improved by adding short fixed wings which may generate a lot of lift support in forward flight with measurable velocity and tilt-able ducted fans. Blimps are safer, easy to fly and have a higher stability than other air vehicles.

In recent years, usage ducted fans for propulsion has increased by both model and full-scale light aircraft builders[3]. Especially, high efficiency of ducted fans in vectored thrust makes it
an ideal propulsion system for lighter-than-air platforms. To take off statically heavy and to land light an airship, the ability of vector thrust is needed\[3\]. Thus, most of the airships use ducted fans to improve operating economics. Besides these advantages, ducted fans are much more quieter than the conventional propulsion systems such as turbojets, turbofans or propellers. Also the shroud grants a safer operation on the ground.

All these features of blimps and ducted fans have forwarded us to the idea of developing an autopilot system for a small scale blimp using electric ducted fans. Developing such an autonomous system requires a comprehensive engineering tasks, embracing knowledge as diverse as aerodynamics, vehicle dynamics, electronics, programming, signal processing and prototyping\[4\]. So, we have started our research with the awareness of this project’s complexity.

1.1 History of Blimps

The history of the blimps goes to the first invention of the balloons. In 1784, Jean-Pierre Blanchard had tried to control a balloon by mounting a hand-operated six blade propeller to the basket but his attempt resulted with a failure\[5\]. In the same year, he published a design for a cigar shaped steerable airship, that was to become a standard for the airships later\[5\]. Later in 1851, Dr. William Bland came up with the idea of designing a steam engine powered blimp\[6\]. The idea of supplying a power to airships by a steam engine was highly appreciated by the experts and in 1852 Henri Giffard flew 27 km on a steam powered airship\[7\]. Paul Haenlein developed an internal combustion engine powered airship in 1872 but in the first public demonstration, blimp damaged by a fire before it flew\[8\]. Later, scientist developed an electric powered blimp. All these developments made airships and blimps so popular among the people. Especially, twentieth century has been the best era of the blimps and airships. In 1900, the first rigid airship was invented by Count Ferdinand von Zeppelin\[8\], which was to become the most popular airship in the world later. Various models of Zeppelin were built and used for military and civilian purposes. But all these popularity was lost with the crash of the most famous Zeppelin, Hindenburg, in 1937 with 35 fatalities.

1.2 Similar Projects

A rich and varied literature exists in the field of UAV autonomous systems for different types of air vehicles. However, most of these systems are designed for conventional airplanes, helicopters or multi-copters. There are several studies and projects on the field about the blimps or airships.

1.2.1 Research and Development Projects

Fukao, developed an autonomous blimp for a surveillance system using a camera, which uses the image processing information to circle around a target object using Lucas-Kanade algorithm\[10\]. Rottman presented an approach, which applies Monte Carlo reinforcement learning and utilizes
Gaussian process for dealing with the continuous state-action space, to control the height of a blimp[11]. Anderson developed a low-cost, light-weight, low-g IMU for an indoor blimp[12]. Bestaoui, derived the dynamic motion equations of a small autonomous blimp using the Newton-Auler approach. They ignore the aero-elastic effects, so all the equations derived with the rigid body assumption. They did not discussed the case of a partially inflated blimp[13]. Besides these partial blimp works, LAAS/CNRS started a whole autonomous blimp project in 2001[14]. A 25 foot radio controlled outdoor blimp with two 7.5 $cm^3$ engines was used in their research. The blimp was equipped with several sensors and with computing and communications capabilities in order to transform it from a radio controlled machine to a robot. They mainly concentrated on two objectives: the definition of blimp trajectory control laws and on environment modeling issues using low altitude imaginary. On the way of first objective, a complete model of their airship which takes into account the aerodynamic, dynamic and aerostatic wrenches was developed. From this model, they developed a control strategy which was based on lateral and longitudinal state decoupling[14]. They used three different image processing algorithm (stereo-vision, motion estimation and map building) to build various digital elevation maps integrating tens of stereo images. Another autonomous blimp project, Project AURORA (Autonomous Unmanned Remote Monitoring Robotic Airship), held at the Information Technology Institute of Campinas, Brazil, which aims at the development of unmanned robotic airships capable of autonomous flight over pre-defined navigation points for aerial inspection and environmental monitoring missions[2]. A 35 foot non-rigid airship, powered with two propellers driven by two-stroke engines is used as a platform. AURORA Autonomous Robotic Airship Project, was considered as one of the first successful controlled flights of an outdoor airship reported in the literature[2]. Other important research related to outdoor autonomous blimps in the world are: the Lotte Project[15] at
Germany, the DIVA Project in Portugal and the partnership of STWing-SEAS of University of Pennsylvania and EnviroBLIMP at Carnegie Mellon University[16].

Figure 1.2. System components of the Project AURORA[2]

1.2.2 Open Source Projects

Recent developments in MEMS technology has cheapen the sensor prices. This, let the researchers, developers and amateur hobbyist to develop their autonomous projects easier than before. Today, it is possible to design your own control board with several sensors that you will need, with a cost of under $100. There are several successful open source projects on the market. These are:

1.2.2.1 ArduCopter

Arducopter is an opensource UAV platform for multi-rotors and helicopters[17]. Besides its basic manual flight control capability for RC multi-copters, it is a complete UAV solution with its autonomous flight control and mission planning feature with a well designed user interface[Figure 1.3]. However, Arducopter doesn’t supports blimps or airship models.

Figure 1.3. A view from ArduCopter Mission Planner User Interface[17]
1.2.2.2 ArduBlimp

ArduBlimp is low cost open source autonomous flight kit, designed for small scale indoor blimps[18]. It uses Arduino as a main flight controller. Using an infrared and ultrasonic sensor, it navigates around a pre-defined object. The project is still being developed.

1.2.2.3 Paparazzi

Paparazzi is a free and open-source hardware and software project aims to develop an autopilot system for multiple platforms such as fixed wing and multi-copters[19]. It is being developed at ENAC University. A robust and accurate altitude estimate, provided by its unique combination of inertial measurement and/or infrared thermo-piles for attitude sensing, are the main features of the Paparazzi autopilot[19].

1.3 Project Objectives

The main goal of this thesis is to develop a low cost semi-autopilot system for a small scale blimp, using electric driven ducted fans and low cost avionics. This system should provide an autonomous flight control to a blimp, from one pre-defined navigation point to another. This robust attitude can be provided using direction hold, altitude hold and speed hold features during flight. Also control system have to be robust enough to withstand unexpected weather conditions, up to some limit. System should also able to be controlled manually, if an unexpected event happens or the operator demands it.

The size of the blimp should be big enough to carry all mandatory sensors, to estimate the attitude and position of the blimp, on-board control system and flight components like servos and fans. Tilt-able electric ducted fans should be used for vertical take-off and landing. Blimp should also be equipped with two horizontal and vertical tails with control surfaces(elevator and rudder) for steering and a gondola to place main board, sensors and batteries. System should have a telemetry system, so the real time sensor values can be transferred to the ground station and received commands from the ground station during the flight. Detailed description of the air vehicle and flight conditions are given in the Chapter 2.

This thesis does not aim primarily on advancing the vehicle dynamics of a blimp or making an invention on UAV technologies, but rather on the development of autopilot system for a blimp with some basic autonomous features with cost of under $300.

1.4 Methodology

In this project, Arduino micro-controller system is chosen as an on-board microprocessor board. Arduino will be responsible of processing all information gathered from the sensor. Arduino is chosen because it has the flexibility of various sensor integration and provides an user friendly GUI(graphical user interface) for easy programming. Also since the programming language of
Arduino is C++ based, the learning curve is not steep for an user who has a C/C++ programming language background. Another advantage of using Arduino platform is finding and integrating several sensors is so easy. Since, it is the most popular on-board platform on the market, it is easy to find several online resources.

For the estimation of the vehicle attitude and position, several sensors will be used. For the direction control, a digital magnetometer will be used. Vehicle will also have an air speed sensor. The altitude information will be obtained from both altitude/pressure and GPS sensor for the best accuracy. Temperature information will be used to estimate the density of the air to calculate airspeed. As we described before, at the end of this project, the system should be able of navigate from one pre-defined navigation point to another. In order to obtain vehicle position information, a GPS sensor will be used. All these sensory outputs will be filtered using Kalman method and Low-pass filters. PID controller will send the output signals to the servos and motors to keep the blimp on the desired trajectory.

Before outdoor flight tests, all the software will be tested indoor to reduce the risk of crash during outdoor flight tests.
In this chapter the general description of the vehicle is given in details. Moreover, manufacturing process of the circuit board, small prototype and test stands are also described.

2.1 Vehicle Configuration

As declared in the previous chapters the aim of this project is to develop a semi-autonomous system for a small scale outdoor blimp. Before starting to develop such an autonomous system, vehicle configuration should be defined.

Basically a blimp consists several parts: envelope or hull, gondola, thrusters and tail surfaces. Envelope is a large thin bag, generally made of a durable, lightweight fabric (polyester composites), that stores helium gas. Usually envelope is cigar-shaped in order to reduce the drag force. Since the characteristic of the blimp is highly influenced by the envelope, sizing is an important part in this project. The gondola is a rigid shape box or platform where the electronics, batteries and miscellaneous components are usually kept. In some blimp configurations, thrusters are also attached to the gondola. The thrusters are the main components that controls the blimp. Their shapes and the specifications closely related with the size of the envelope. Several calculations needs to be done before selecting. Tail surfaces are another important components of a blimp. Usually, in small sized blimps, tail surfaces are used as non-active surfaces for stabilization purposes. But in larger blimps, tail surfaces can also be used to as active control surfaces to guide the vehicle. In some configurations, a small fan is embedded in one of the vertical tail surfaces to provide an extra moment force for steering.

Before deciding the exact vehicle configuration, a trade off study had been carried out. After this study, several sketches were drawn. After eliminating some of the sketches, two different vehicle configuration came forward.
2.1.1 Two Rotors Configuration

This configuration includes two ducted fans as main thrusters, located under the gondola. Both of these fans work at the same speed and both fixed on a rod which is actively driven by an actuator. As it can be seen in Figure 2.1, two vertical and two horizontal tails are located on the blimp. These tails are active control surfaces and controlled by two different digital actuators. The active control surfaces can only work efficiently when enough air flows around them. One brush-less rotor is embedded in the lower vertical tail to generate required torque force to steer the blimp in hover condition or at slow speeds. Weight is an important criteria in selecting the right size of the blimp. The envelope should be big enough to generate required lift force to provide buoyancy. So the components required with this configuration are listed in Table 2.1.1 with their weights.

Roughly, this configuration needs:

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Weight (g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brushless Ducted Fans</td>
<td>2</td>
<td>320</td>
</tr>
<tr>
<td>Brushless Tail Rotor</td>
<td>1</td>
<td>36</td>
</tr>
<tr>
<td>Electric Speed Controller</td>
<td>3</td>
<td>210</td>
</tr>
<tr>
<td>Digital Servo</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>590</strong></td>
</tr>
</tbody>
</table>

Table 2.1. Two Rotors Vehicle Configuration

2.1.2 Four Rotors Configuration

In this configuration four main rotors as a main thruster are located under the gondola. As it can be seen in Figure 2.2, two vertical and two horizontal tails are located on the blimp. Compared to the first configuration, these tails are non-active tail surfaces and there is no brushless rotor in the lower vertical tail. All of the main thrusters has their own electronic speed controller, so each can run at a different speed. Table 2.3 shows the components needed in this configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Weight (g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brushless Ducted Fans</td>
<td>4</td>
<td>640</td>
</tr>
<tr>
<td>Electric Speed Controller</td>
<td>4</td>
<td>280</td>
</tr>
<tr>
<td>Digital Servo</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>936</strong></td>
</tr>
</tbody>
</table>

Table 2.2. Four Rotors Vehicle Configuration

Both configurations has their own specific advantages and disadvantages. First, it can be easily said that the first configuration needs a smaller envelope. As given in the Table 2.2 and Table 2.3, the basic components, without any sensor, microprocessor and battery, weight approximately 550 and 960 grams respectively. Since the avionics will be the same at both configuration, vehicle configuration can be decided with the basic components weight comparison. Moreover, the first configuration needs only two ducted fans. Ducted fans are the heaviest and
Two Rotors Configuration

- 2 tilt-able ducted fans as a main thruster
- 1 brush-less motor on lower vertical tail
- Two active vertical and horizontal tails
- Needs 2 different ESCs
- Needs 3 actuators

Figure 2.1. Two Rotors Configuration
Advanced Configuration

Four Rotors Configuration

- 4 tilt-able ducted fans as a main thruster
- Fixed vertical and horizontal tails
- Needs 4 different ESCs
- Needs 2 actuators

Figure 2.2. Four Rotors Configuration
the most energy consuming parts of this vehicle. More ducted fans will need more batteries to have the same flight duration. Since weight is the most important criteria in blimp design, it is better to use as little as possible battery. Also, the first aim of this project is development of an autopilot system rather than designing a vehicle. So, the vehicle configuration should be as common as possible.

After a market research, a 10 foot envelope was chosen from Aerial Products Corporation. The choice of the blimp was made after making an estimate of the weight of avionic components that is to be used. The blimp comes with two ducted fans, an extra rotor on the lower vertical tail and gondola. The net payload is given as 227g in the technical data sheet provided by the company. This payload will be used for the avionics.

<table>
<thead>
<tr>
<th>Specifications:</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Panels:</td>
</tr>
<tr>
<td>Length:</td>
</tr>
<tr>
<td>Diameter:</td>
</tr>
<tr>
<td>Helium Req.:</td>
</tr>
<tr>
<td>Net Lift:</td>
</tr>
<tr>
<td>Engines:</td>
</tr>
</tbody>
</table>

**Table 2.3.** Specifications of Envelope

Before deciding blimp configuration, calculating the buoyancy will give information about the payload capacity of different configuration alternatives. Blimp buoyancy can be calculated multiplying the density difference between air and helium gas at Sea Level with envelope volume. Table 2.4 shows the gas properties of Helium and Air at Sea Level.

Using Eq. 2.1 and Eq. 2.2, net lift forces can be calculated.

$$\Delta \rho = \rho_{air} - \rho_{He}$$  \hspace{1cm} (2.1)

$$Bouyancy = Volume.\Delta \rho$$  \hspace{1cm} (2.2)
Table 2.4. Gas Properties at Sea Level

\[
Bouyancy = (1204.2 - 166.25) \times 2.9 = 3.010\, kgf
\] (2.3)

The envelope that is chosen gives 3.010 kgf buoyancy.

<table>
<thead>
<tr>
<th>Components</th>
<th>2 Rotors</th>
<th>4 Rotors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Components</td>
<td>590</td>
<td>936</td>
</tr>
<tr>
<td>Avionics</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>Envelope</td>
<td>1204</td>
<td>1204</td>
</tr>
<tr>
<td>Batteries</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>Gondola</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td><strong>Total Weight</strong></td>
<td>2404</td>
<td>2750</td>
</tr>
<tr>
<td><strong>Bouyancy</strong></td>
<td>3010</td>
<td>3010</td>
</tr>
<tr>
<td><strong>Net Lift</strong></td>
<td>606</td>
<td>260</td>
</tr>
</tbody>
</table>

Table 2.5. Target Blimp Configuration Weight Comparison

Table 2.5 shows the weight comparison of two different vehicle with all of the components. Two rotors configuration has much more payload capacity compared to the four rotors one. Payload capacity can be used for carrying extra sensors, visual devices such as cameras. This capacity can be used in future improvements also. So it is decided to continue with "Two Rotors Configuration" and all the calculations are done with using this configuration.

2.2 Vehicle Manufacture & Assembly

2.2.1 Small Scale Prototype Manufacture

2.2.1.1 Blimp

After a market research, it is decided to manufacture a small scale prototype of the blimp for testing. In this purpose, 44 x 20 x 18 inch, metalized nylon (mylar) envelope with a resealable valve is chosen from Interactive Toy Concepts LTD. This blimp balloon holds up to 5 cubic feet of helium. Before manufacturing of the gondola, lift capacity of a fully filled envelope is measured.
Figure 2.4. A picture of blimp that is chosen for small prototype[21]

with electronic scale. First measurements shows that blimp can lift only 75 grams. Figure 2.4 shows the view of envelope.

2.2.1.2 Gondola

Gondola is a rigid box, where the avionics and the payloads are stored. A five by five inch rectangular prism shaped gondola was manufactured from 2 mm thick balsa wood. Balsa is chosen because of its lightweight. To attach the motors to the gondola, a 10 inch long stick is passed through the cg(center of gravity) point of the gondola. Two 9V brushed engines are put on the both ends of the stick. The top side of the gondola is left open to remove the avionics easily.

At first, the gondola is attached to the one envelope with Arduino board and motors. It has seen that one envelope can not produce enough lift. So as a first assumption, it was thought that two blimps will be enough.

Two blimps are attached to each other with Velcro. Velcro is chosen because its light-weight makes it easier to unattached the blimps when it is required. Since the gondola is designed for one blimp configuration at first, an extra structure is needed to attach it to the both blimps. This structure should be lightweight but strong enough to hold two blimps in a rigid shape. Figure 2.5 shows the manufacturing process of the connection structure and the gondola.

2.2.1.3 Stabilizer Tail Fins

Stabilizer tail fins are important components of an air vehicle for the flight stability. It has seen that blimp itself drifts without stabilization components. In order to fix that problem six tail fins, made of balsa, are manufactured. And four them are sealed as a vertical stabilizer and two of them are sealed as a horizontal stabilizer. The design of tail fins are supplied from the manufacturer company. Figure 2.6 shows manufacture process of tail fins.
2.2.2 Electronic Board Design & Manufacture

2.2.2.1 Motor Controller Circuit

Arduino is a micro-controller that has several digital and analog outputs. It also has several power output pins, support 3.3V and 5V, these can be used as power source for sensors, small actuators and motors. Arduino is a powerful control board that comes with several features but it is not a tool for heavy lifting. It can handle small sensors, LEDs and small LCDs but not with high voltage, high current devices such as a DC motor. Arduino can not supply a power higher than 5 volts but it can send signals to an external switch to decide how much power it should pass.

TIP120 is an NPN(Negative-Positive-Negative) Power Darlington Transistor, that can handle with high amperage, but Arduino can’t. In this project, Arduino micro-controller is used to tell to the TIP120 transistor, how much power to pass from external power source to the motors.

The working principle is, Arduino gives orders through one of its PWM pins to the TIP120
Figure 2.7. Picture of the small scale envelope chosen for prototype

Figure 2.8. TIP120 NPN Power Darlington Transistor Scheme

Base pin and TIP120 transistor maps the voltage value from 0 to maximum value that comes from the battery and drives motors. In order to protect the Arduino from high voltage, 2.5K resistor is put between Arduino’s PWM pin and TIP120 Base pin. Also DC motors generate lots of harmful sparks and stray electricity that needs to be blocked. So a diode is put to protect the transistor from the reverse voltage that spikes back. The last product can be seen in Figure 2.9.

2.2.2.2 Sensor Circuit for Small Scale Prototype

As given in Chapter 1, six different sensors are used in this project. These sensors establish communication between the blimp and the environment. Some of them just give attitude infor-
All these sensors are sensitive devices and can be affected by other devices easily. So location of these sensors is an important decision. For example, magnetometer is a device that measures the earth’s magnetic field in 3 axis. It gives three vector values in 3 axis and these values can be used to define the heading of the vehicle. But since it is so sensitive, its measurements can be affected by metals, electronic devices or any object that generates a magnetic field. So it’s important to locate this sensor in a place where should be far away from the motors and actuators which are powerful magnetic field generators.

In this purpose a three by three breadboard is used to put all the sensors together and this circuit is located above the gondola. Figure 2.10 shows the schematic of the circuit design.
An autopilot system requires a main board with an on-board processor and several sensors to fly an air vehicle to control. Sensors obtain all the information (aircraft attitude, geo-localization, altitude, airspeed etc.) required for the flight and pass it to the main board, which makes real-time decisions during the flight. The number of the sensors on an autopilot system depends on advancement of the control system. However, this number is seriously constrained by the load, energy consumption and weight [22], especially on small UAV systems. This situation is more serious on small airships.

As established in the Objectives section, it is intended to develop a control system with several features such as Air Speed Holder, Direction Holder and the Altitude Holder. To design such a controller, several sensors are needed. Direction holder can be supplied by using a magnetometer sensor. To measure the air speed of the vehicle, total pressure and the static pressure should be measured. Altitude information can be supplied using a pressure sensor. Vehicle attitude estimation is critical in indoor flights, where GPS (Global Positioning Systems) data is unavailable. So the estimation of the attitude degree can be calculated by combining the sensor data from both the accelerometer and gyroscope. Autonomous navigation can be provided by using the coordinate information received from GPS sensor.

This chapter addresses detailed technical information about the sensors chosen for this project.

3.1 Main Controller

The main controller is the unit where all the sensor information processed and flight control decisions are made. Integration of the several sensors needs fast data processing and transfer rates to pass the decisions to the required targets.

This section gives information about the main controller device used in this project.
3.1.1 Arduino

Recent developments in MEMS (Microelectromechanical systems) technology has reduced the sensor prices. This reduction, let the researchers, developers and amateur hobbyist to develop autonomous projects easier than before. Today, it is possible to design your own control board with several sensors that you will need with a cost under $100.

Arduino platform is the most popular Micro Controller Unit (MCU) on the market. Its developers define Arduino project as an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It has been a cheap and easy solution as single-board processor (state estimator and flight controller) who wants to develop autonomous systems since it was commercialized.

Arduino platform has several versions with different processors types and memory capacity for different applications. In this project we preferred Arduino Mega 2560, which is based on the ATmega2560 chipset. It has 54 digital input/output pins (which 14 can be used as PWM (Pulse-with-modulated) outputs, 16 analog inputs, 4 UARTs (hardware serial ports). It also supports I²C bus which is a multi-master serial single-ended computer bus invented by Philips that is used to attach low-speed peripherals to a motherboard, embedded system, cell-phone, or other electronic device. I²C provides connect sensors using the same pins. Technical specifications of Arduino Mega 2560 is given below in Table 3.1.
### Table 3.1. Technical Specifications of Arduino Mega 2560

<table>
<thead>
<tr>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>ATmega2560</td>
</tr>
<tr>
<td>Operating Voltage</td>
<td>5V</td>
</tr>
<tr>
<td>Input Voltage (recommended)</td>
<td>7-12V</td>
</tr>
<tr>
<td>Input Voltage (limits)</td>
<td>6-20V</td>
</tr>
<tr>
<td>Digital I/O Pins</td>
<td>54 (of which 15 provide PWM output)</td>
</tr>
<tr>
<td>Analog Input Pins</td>
<td>16</td>
</tr>
<tr>
<td>DC Current per I/O Pin</td>
<td>40 mA</td>
</tr>
<tr>
<td>DC Current for 3.3V Pin</td>
<td>50 mA</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>256 KB of which 8 KB used by bootloader</td>
</tr>
<tr>
<td>SRAM</td>
<td>8 KB</td>
</tr>
<tr>
<td>EEPROM</td>
<td>4 KB</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>16 Mhz</td>
</tr>
</tbody>
</table>

3.2 Sensors

A sensor is a converter that measures a physical quantity and converts it into a signal which can be read by an observer or an instrument[26]. Sensors work as a human sense organ and make devices to interact with their environment. As mentioned before, an autopilot system needs several sensors to understand its environment during flight and make real-time decisions to fly using these sensor values. In this project, six different sensors and a telemetry breakout board for wireless communication with ground station are used.

#### 3.2.1 Barometric Pressure and Temperature Sensor

Pressure sensor is an electronic device measures pressure of the ambient air. It uses a transducer to convert pressure force to electricity. In this project, BMP085 Breakout board is used, which has the Bosch BMP085 high-precision, low-power barometric pressure sensor. It works within a range of 300 to 1100 hPa with an accuracy of 0.03 hPa, in ultra high resolution mode which corresponds to 0.25m in sea-level altitude[27]. Besides its pressure sensor, it also has a temperature sensor on it. Both information are transmitted by I²C serial bus which allows easy system integration with a micro-controller. Sensor is fully calibrated by the manufacturer, so it doesn’t need an extra calibration process.

Sensor provides pressure and temperature data at the measurement location. Using the measured pressure and the pressure at sea level, the altitude in meters is calculated using the international barometric formula given below:

\[
\text{altitude} = 44330 \times \left(1 - \left( \frac{P}{P_o} \right)^{\frac{1}{5.256}} \right)
\]

where

- \( P_o \) is the sea level pressure which changes day by day. So, on the flight day, this value will be obtained from the closest local weather station.
• $p$ is the measured pressure.

Figure 3.2. BMP085 Barometric Pressure and Temperature Sensor[27]

3.2.2 Accelerometer

An accelerometer is an electro-magnetical device that measures the change in velocity in an unit of time. It can measure the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion, shock, or vibration[28]. Most of the accelerometers on the market today, use differential capacitors to measure g-force, which is then converted into bits or volts and then passed to a microprocessor to perform action[29].

In this project, ADXL345 breakout board is chosen to determine the vehicle attitude information. The ADXL335 is a small, thin, low power, complete 3-axis accelerometer with high resolution (13-bit)[28]. The product measures acceleration with a minimum full-scale range of ±3 g up to ±16g, which is much more sensitive than that we require in this project. The sensor uses the $I^2C$ serial interface to communicate with the main board. Besides this feature, it has a single and double tap detection feature. It also has free falling feature, which can be used to determine if the blimp falls. The acceleration information supplied from this sensor will be used to determine the vehicle orientation in 3D space combined with the information that received from magnetometer and gyroscope.

Figure 3.3. Triple Axis Accelerometer Breakout - ADXL335 [28]

Features:

• 2.0-3.6VDC Supply Voltage
• Ultra Low Power: 40µA in measurement mode, 0.1µA in standby@ 2.5V
• Tap/Double Tap Detection
• Free-Fall Detection
• SPI and I²C interfaces

Detailed vector calculation and vehicle attitude detection will be given on Chapter 5.

3.2.3 Gyroscope

A gyroscope is a device for measuring or maintaining the orientation of a system, form on the principles of angular momentum [30]. The application area of the gyroscope is considerably wide. It varies from a palm-size smart phones to large airplane. Especially, recent developments in MEMs technology reduced the gyroscope sensor costs.

In this project, ITG-3200 Breakout board offered by Sparkfun company is chosen. The ITG-3200 features three 16-bit analog to- digital converters(ADCs) for digitizing the gyro outputs, a user low-pass filter bandwith ad a Fast Mode I²C(400kHz) interface[31]. It can be powered by a power source between 2.1 and 3.6V. The angle rate information provided by the sensor is used in vehicle attitude calculation. Detailed information about the calculation will be given in Chapter 5.

![Figure 3.4. Triple-Axis Digital-Output Gyro ITG-3200 Breakout [31]](image)

**Features:**[31]

• Digital-output X-, Y-, and Z-Axis angular rate sensors (gyros) on one integrated circuit
• Digitally-programmable low-pass filter
• Low 6.5mA operating current consumption for long battery life
• Wide VDD supply voltage range of 2.1V to 3.6V
• Standby current: 5µA
• Digital-output temperature sensor
• Fast Mode I²C (400kHz) serial interface
• Optional external clock inputs of 32.768kHz or 19.2MHz to synchronize with system clock

• Pins broken out to a breadboard friendly 7-pin 0.1” pitch header

3.2.4 Magnetometer

Magnetometer is a device that measures the strength and the direction of magnetic fields. Today, magnetometers are generally used for measuring the magnetic field of the Earth to determine the direction of an object, as a compass. They can also be used for measuring magnetic abnormalities and detecting magnetic metals (ferros) [32]

A compass is a navigation instrument that provides a heading information parallel to the surface of the earth. The strength of the earth’s magnetic field is about 0.5 to 0.6 gauss and has a component parallel to the earth’s surface that always points toward the magnetic north pole. A compass consists a magnetic needle that always forwards itself with the magnetic field of the earth. The main working principle of a magnetometer is also so similar with the traditional compasses. The magneto-resistor inside the magnetometer adjusts the current flow in three axis with the change of the magnetic field.

In this project, to calculate the heading for the blimp, LSM303 Breakout Board is chosen as a magnetic field information provider. The LSM303 Breakout Board is a triple axis accelerometer combined with a triple axis magnetic sensor.

![Figure 3.5. Triple-Axis Tilt Compensated Compass- LSM303 Breakout Board][33]

Features:[33]

• ±2/4/8 g dynamically selectable full-scale

• ±1.3 to ±8.1 gauss magnetic field full-scale

• 16-bit data out

• I²C interface

• Embedded self-test
3.2.5 Airspeed Sensor

The airspeed sensor kit consists of a MPXV7002DP Differential pressure sensor, a pitot tube with two ports and a plastic tube. MPXV7002DP piezoresistive transducer is state-of-the-art monolithic silicon pressure sensors designed for a wide range of applications, but particularly those employing a micro-controller or microprocessor with A/D inputs[34].

The sensor outputs an analog signal proportional with the differential pressure value between the total and the static ports of the pitot tube. The output signal is between a range of 0.5V to 4.5V. This output can be used to calculate true air velocity. Vehicle speed can also be measured by GPS but especially at low speed, GPS speed accuracy can be really low. Since blimps is a low speed air vehicle it is better to use a sensor like this.

3.2.6 GPS Sensor

The Global Positioning System (GPS) is a space-based satellite navigation system that provides location and time information in all weather, anywhere on or near the Earth, where there is an unobstructed line of sight to four or more GPS satellites[36]. GPS are used in lots applications today. Today’s advanced GPS systems provide latitude, longitude and altitude information with an accuracy of less than 1 meters.

In this project, EM-408 GPS module[Figure 3.7] from USGlobalSat based on the SiRF Star III Chipset is selected. It is a low cost GPS with high reliability and accuracy. It has a built-in patch antenna. Here are some features of the module:

Features:[37]

• 20-Channel Receiver
Figure 3.7. 20 Channel EM-406A SiRF III Receiver with Antenna [37]

- 10m Positional Accuracy / 5m with WAAS
- Hot Start : 1s
- Warm Start : 38s
- Cold Start : 42s
- 70mA at 4.5-6.5V
- Outputs NMEA 0183 and SiRF binary protocol

The outputs the data through the TTL serial line and supports several GPS output message protocols such as, NMEA 0183 GGA, GSA, GSV, RMC (VTG, GLL optional)[38].

3.2.7 Wireless Communication

The wireless communication is an important part of autopilot system. It needs to be reliable and fast enough to support the real-time communication between the air vehicle and the ground station. Having a wireless communication gives the operator the ability monitoring sensor readings, updating mission objectives and taking the control of vehicle in case of an emergency.

In this project, wireless data link is granted by two XBee Pro embedded RF modules [Figure 3.8]. This is a very popular 2.4GHz Xbee XBP24-AWI-001 module from Digi. It allows a very reliable and simple communication between the micro-controller and the ground station via serial port. It supports 128-bit encryption with a data rate of up to 250kbps. It allows a secure and reliable communication in a range up to 1 mile(1600 meters).

Some of the features of the XBee module are:

Features:[39]

- 3.3V @ 215mA
- 250kbps Max data rate
- 60mW output (+18dBm)
- 1 mile (1500m) range
- Built-in antenna
- Fully FCC certified
- 6 10-bit ADC input pins
- 8 digital IO pins
- 128-bit encryption
- Local or over-air configuration
- AT or API command set

Figure 3.8. XBee Pro 60mW Wire Antenna - Series 1 (802.15.4) [39]

3.2.8 Low Cost Sensors

The total cost of the avionics used in this project is given on Table 3.2. The total cost is $306. All of the sensors used in this project is used with a breakout board. So, the actual price of sensor chips are much less than this price. The breakout board doubles the sensors’ real price. As if a PCB board is designed, the breakout board will not be needed to be used. So it can be said that the total cost will be approximately half of this.

<table>
<thead>
<tr>
<th>Components:</th>
<th>Unit(g)</th>
<th>Price($)(w/ breakout board)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arduino Mega</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>Accelerometer - ADXL345</td>
<td>1</td>
<td>27.50</td>
</tr>
<tr>
<td>Gyroscope - ITG3200</td>
<td>1</td>
<td>27.50</td>
</tr>
<tr>
<td>Magnetometer - LSM303</td>
<td>1</td>
<td>49.50</td>
</tr>
<tr>
<td>Airspeed Sensor</td>
<td>1</td>
<td>24.50</td>
</tr>
<tr>
<td>GPS - 20 channel EM-408 SiRF III Receiver</td>
<td>1</td>
<td>65.45</td>
</tr>
<tr>
<td>Pressure &amp; Temperature Sensor - BMP085</td>
<td>1</td>
<td>8.50</td>
</tr>
<tr>
<td>XBee Pro 60mW wire Antenna-Series 1</td>
<td>2</td>
<td>44.00</td>
</tr>
<tr>
<td><strong>Total Cost</strong></td>
<td></td>
<td><strong>306.4</strong></td>
</tr>
</tbody>
</table>

Table 3.2. Total Cost of Avionics
Data Acquisition & Control

Algorithms

Developing a control system for an autonomous flight needs several complex data acquisition and control algorithms. The complexity of the algorithm depends on the situations that the UAV has to deal with. However, there is a minimum level of simplicity of the algorithm to have autonomous flight. Data acquisition is one the essential part of the algorithm that determines the complexity. Most of the sensors provide raw information like pressure, force, temperature or voltage. This information should be processed before used by controllers for autonomous flight.

This chapter addresses the data acquisition and the algorithm of the controllers. Firstly, the block diagrams will be given to show the logic of the data acquisition from the sensors and then the algorithms used to control the UAV will be presented.

4.1 Data Acquisition

In this section the algorithms developed to process the raw sensor information is described.

4.1.1 Attitude Estimation

The attitude estimation is critical for autonomous flight. The orientation of the vehicle can be learned from the attitude estimation. The attitude estimation is a result of sensor reading or processing[4], usually from both the accelerometer and gyroscope, sometimes also magnetometer. This sensor combination is commonly called Internal Measurement Unit (IMU).

Accelerometers are incapable of measuring the gravitational force but they can be used to measure the upwards acceleration that counters gravity at rest. This acceleration is measured 1g on the z-axis when both the pitch and roll axis are zero, but when the sensor is tilted, an upward component exits on x-axis or y-axis. These components can be used to calculate pitch and roll
Positive and negative attributes of accelerometers can be listed as[41]:

**Positive**

- If not moving, accelerometer will give accurate reading of tilt angle

**Negative**

- Accelerometers are slower to respond than Gyro’s
- Accelerometers are prone to vibration/noise

To start the calculation, the reference coordinate system and the rotation axes should be defined. The convention of aeronautics for the reference axis \((x \rightarrow y \rightarrow z)\) is used to define the attitude of the blimp. The linear displacements\((x, y, z)\), and the orientation angles\((\phi, \theta, \psi)\) -(roll, pitch and yaw respectively) are defined using the aerospace convention. Figure 4.1 shows the positive directions used this thesis.

\[\begin{align*}
\tan \phi_{xyz} & = \left( \frac{G_{py}}{G_{pz}} \right) \\
\tan \theta_{xyz} & = \left( \frac{-G_{px}}{G_{py} \sin \phi + G_{pz} \cos \phi} \right) = \frac{-G_{px}}{\sqrt{G_{py}^2 + G_{pz}^2}}
\end{align*}\]  

(4.1)

Here \(G_{px}, G_{py}\) and \(G_{pz}\) are the upward acceleration in \(x, y, z\) axis respectively.

These equations have an infinite number of solutions at multiples of 360. Restricting the range of the roll and pitch angles to lie in the range -180 to 180 helps but solutions still gives two unique solutions for roll and pitch angles. The solution is to constrain either the roll or the pitch angle (but not both) to lie between -90 and +90. The convention used in aerospace sequence is that roll angle can range between -180 and +180 but the pitch angle is restricted to -90 and +90. This restriction eliminates the two duplicate solution and gives a single solution. Also the roll equation is undefined when both \(G_{py}\) and \(G_{pz}\) are equal to zero. These problems can be
solved in code by using the function \( \text{atan2}() \), which eliminates the angle calculation ambiguity by taking into account the quadrant\[42\].

To calculate angle values from the accelerometer, the methods given in Listing 4.1.

**Listing 4.1. Accelerometer tilt angle calculation**

```c
double getXangle() {
    double accXval = (double)readAcc(0) - zeroValue[0];
    double accZval = (double)readAcc(2) - zeroValue[2];
    double angle = (atan2(accXval, accZval) + PI) * RAD_TO_DEG;
    return angle;
}

double getYangle() {
    double accYval = (double)readAcc(1) - zeroValue[1];
    double accZval = (double)readAcc(2) - zeroValue[2];
    double angle = (atan2(accYval, accZval) + PI) * RAD_TO_DEG;
    return angle;
}
```

where

- \( \text{readAcc}(0) \), \( \text{readAcc}(1) \) and \( \text{readAcc}(2) \): are the functions which send required address values to the sensor and return acceleration data in X, Y, and Z axes respectively.
- \( \text{zeroValue}[0], \text{zeroValue}[1], \text{zeroValue}[2] \): are the initial average readings at leveled position in X, Y and Z axes respectively.

![Vehicle Attitude Estimation Block Diagram](image)

**Figure 4.2. Vehicle Attitude Estimation Block Diagram**

The gyro can also be used to calculate the current tilt angle by taking a reading at a set frequency, calculating how many degrees it had turned in that period and then summing these values up. This is called integration by time. It has several positive and negative attributes\[41\]:

- **Positives**:
  - It is robust against drift in the accelerometer.
  - It can provide accurate data in stationary conditions.

- **Negatives**:
  - It accumulates errors over time.
  - It requires high sampling rates to be effective.

By combining the data from both the accelerometer and the gyro, a more accurate estimation of the vehicle’s attitude can be achieved.
Positive

• Gyro’s respond fast so they are good at producing a quick response to a change in angle

Negative

• We are only taking readings at certain time intervals. We don’t know what’s happening between these periods.

• For better tilt angle accuracy we need to take more gyro samples which takes more processing time

• Due to the inaccuracy of each gyro reading, the tilt angle calculated will drift over time

The gyroscope measures degrees per second ($\circ/s$) while the accelerometer measures acceleration (g) in three dimensions. The gyro drifts over time and output is too noisy. That means, it can not be trusted for a longer time-span, but it is very precise for a short time. The accelerometer does not have any drift because the accelerometer spits real time data rather than a time average from the Gyro, but it is too unstable for shorter time span. So, both accelerometer and the gyroscope output data can be merged to obtain more precise pitch and roll angles instead of using each of them. This can be done by using Kalman filter or Complimentary Filter [41].

**Complimentary Filter** is designed to combine signals with different content and accuracy characteristics into a composite signal with the ”best” characteristics of each input signal.[43]

Some positive and negative attributes can be listed as[44]:

Positive

• Can help fix noise, drift, and horizontal acceleration dependency.

• Fast estimates of angle, much less lag than low-pass filter alone

• Not very processor-intensive.

Negative

• More theory to understand than the simple filters.

**Kalman Filter** (KF) is defined as ”a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error” by Welch[45]. Predicting the past, present and the future states of the system and achieving even there is an unknown in the absolute character of the modeled system is the most strong features of the filter[45].

Some of the advantages and the disadvantages of Kalman filter are[44]:

Positive

• It is the theoretically most suitable filter for noise filtering.

• The physical properties of the system such as mass, inertia, etc. are also taken into account)
Negative

- Compared to the other filter, it is mathematically complex.
- It needs more knowledge to code.
- Would need more processor time.

Kalman Filter is an advanced design tool for obtaining clear, accurate and reliable data from noisy sensors. Most of the IMU applications uses Kalman Filter. A Complementary Filter is easy to understand and code, however its accuracy is less than Kalman Filter. Thus, an open source Kalman filter library is found and it is embedded to our main code [Appendix B].

The method written for angle calculation from gyroscope output is given in Listing 4.2. The basic process of the code is integrating the rotational rate values by time.

**Listing 4.2. Gyroscope tilt angle calculation**

```c
void calculateGyroAngle()
{
    // Create variables to hold the output rates.
    static double gyroXrate, gyroYrate, gyroZrate;
    // Read the x, y and z output rates from the gyroscope.
    gyroXrate = -(double)(readX() / 14.375);
    gyroXangle += gyroXrate * ((double)(micros() - timer) / 1000000);
    gyroYrate = -(double)(readY() / 14.375);
    gyroYangle += gyroYrate * ((double)(micros() - timer) / 1000000);
    gyroZrate = -(double)(readZ() / 14.375);
    gyroZangle += gyroZrate * ((double)(micros() - timer) / 1000000);

    timer = micros();
    // Wait 10ms before reading the values again.
    // (Remember, the output rate was set to 100hz and 1 reading per 10ms = 100hz.)
    delay(10);
}
```

where

`readX()`, `readY()` and `readZ()` are the methods that send required address values the to the sensor and subtract this value from initial rate of the setting, which is set during initialization.
micros(): is the function which returns the time value passed since the start up of the Arduino device in micro seconds.

After obtaining tilt angle values from both accelerometer and gyroscope, merging can be done by using Kalman Filter. The code given below passes the angle values to the Kalman filter function. The detailed code of Kalman Filter calculation can be found in Appendix B.

**Listing 4.3. Filtered tilt angle calculation**

```c
void calculateRotation()
{
    double accXangle = getXangle();
    double accYangle = getYangle();

    // Complementary filter calculation
    compAngleX = (0.93*(compAngleX+(gyroXrate*(double)(micros()-timer)/1000000)))+(0.07*accXangle);
    compAngleY = (0.93*(compAngleY+(gyroYrate*(double)(micros()-timer)/1000000)))+(0.07*accYangle);

    // Passing unfiltered tilt angle values from accelerometer and tilt rates from gyroscope

    // calculate the angle using a Kalman filter
    double xAngle = kalmanX(accXangle, gyroXrate, (double)(micros()-timer));

    // calculate the angle using a Kalman filter
    double yAngle = kalmanY(accYangle, gyroYrate, (double)(micros()-timer));

    delay(10);
}
```

### 4.1.2 Compass Heading System

Magnetometer supplies the strength of earth’s magnetic field forces in three axis. Heading is defined as the angle between the X axis (ref: reference coordinate system) and the magnetic north on the horizontal plane measured in a clockwise direction when viewing from the top of the blimp (or device)[46]. Since the earth’s magnetic field components are parallel to the earths field, when the magnetometer is at leveled position, pitch and roll angles are 0. Then the heading
angle can be determined as shown in Figure 4.3. Here $X_b, Y_b$ and $Z_b$ axes are the device(or blimp) body coordinates and $Y_m$, which are named as $X_h$ and the $Y_h$ in here, are the components measured by the magnetic sensor. Then the heading angle can be calculated as:

![Figure 4.3. Heading vector representation w/o tilt angle on body coordinates](image)

This calculation is valid while the roll and pitch angles $0$. Otherwise the magnetic sensor measurements $X_m, Y_m$ and $Z_m$ need to be compensated to obtain $X_h, Y_h$. This components can be compensated using this equation:

$$X_h = X_M \cos \theta + X_M \sin \theta$$  \hspace{1cm} (4.2)

$$Y_h = X_M \sin \phi \sin \theta + Y_M \cos \phi - Z_M \sin \phi \cos \theta$$  \hspace{1cm} (4.3)

and then heading can be calculated with:

$$\text{Heading} = \arctan \left( \frac{Y_h}{X_h} \right)$$  \hspace{1cm} (4.4)

Although the magnetic heading is obtained, it may not necessarily be the correct heading. The reason for this is that there are two types of poles on the Earth: geographic and magnetic. The northern and southern-most points of the globe are the geographic poles (true north). The magnetic poles are the origin of the Earth’s magnetic field and the pull is the strongest at these locations (magnetic north). Since the earth covered with lands and seas, at some places on the Earth, the magnetic fields change. The magnetic field is affected by the soils ferromagnetic properties at lands. So this causes a declination angle in a range of $\pm 20$ depending on the geographic location[46]. And this has to be taken into account in our calculation.
The code given in Listing 4.4 is written to calculate the tilt compensated heading. Here:

- `magValue[]` is the array where the raw magnetic field magnitude is stored.
- `accelValue[]` is the array where the gravitational force magnitude is stored.

The block diagram in Figure 4.5 explains the data acquisition logic in the code. Here heading calculations are the equations given above.

### 4.1.3 Altitude Calculation

Altitude information is one of the crucial parameters for autonomous flight. Altitude information can be obtained from both GPS and barometer. Each of these has some constraints and advantages. GPS can provide a high accuracy altitude information up to centimeter, depending on the number of satellite signals received. However, it cannot be used at indoor applications or when the GPS signal is unavailable. This is a critical limitation. Barometer is a more convenient device...
Figure 4.5. Vehicle Heading Block Diagram

compared to the GPS. It is a device which measures the static pressure of the air so this data can be used to calculate the altitude of the vehicle. The sensor that we chose, BMP085, offers a measuring range of -1000m to 9000m with an absolute accuracy of down to 1m (Figure 4.6)

Figure 4.6. Altitude Change in Standard Atmosphere

For the best result, averaged GPS altitude smoothed by pressure readings passed through a Kalman Filter should give a more accurate altitude estimate when weather conditions are stable. But because of time constrains, it is decided to use only barometer sensor for altitude calculation.

BMP085 sensor provide pressure and temperature information as an output. With the measured pressure $P$, and the pressure at sea level $P_0$ eg. 1013.hPa, the altitude in meters can be calculated with the international barometric formula:

$$h(\text{m}) = \text{altitude} = 44330 \times (1 - P/P_0)^{(1/5255)}$$

(4.5)
Since the sensor outputs real-time pressure, calculation can be coded as given in Listing 4.5

### Listing 4.5. Altitude Calculation

```c
currentAltitude = 44330*(1-pow((currentPressure/100.0/SeaLevelP), (1/5.255)));
```

The block diagram in Figure 4.7 shows the data acquisition process of the altitude reading. Here altitude calculation block represents the international barometric formula. Since the data is noisy because of the a Low-by-pass data filter is used for more reasonable estimate.

![Figure 4.7. Vehicle Altitude Calculation](image)

### 4.1.4 Speed Information

The air speed can be measured by comparing the total pressure(static+dynamic) and ambient pressure(static). To measure the these pressures, a pitot tube with pointing directly into fluid flow and another tube pointing perpendicular to the flow can be used.

The speed sensor provides real-time total and static pressure data. These values can be used to calculate indicated air speed with these formula:

\[ V_{in} = 2\sqrt{(P_t - P_s)/\rho_o} \]  \hspace{1cm} \text{(4.6)}

where

- \(\rho\) is 1.225 \text{ kg/m}^3, the air density at sea level and 15 degrees Celsius

- \(P_t\) stagnation or total pressure inside the pitot tube, in Pascal.

- \(P_s\) static pressure, the pressure measured on the side of the airplane/tube, Pascal.

Indicated airspeed is the true air speed at sea level, which is directly related to the pressure in front of aircraft. True speed is the vehicle’s forward ground speed combined with wind speed.

\[ V_{true} = V_{in}\sqrt{\rho_o/\rho} \]  \hspace{1cm} \text{(4.7)}

where,

\(\rho_o\) is the air density at the current altitude (in \text{ kg/m}^3), can be roughly calculated from GPS altitude and/or barometric absolute pressure using this formula:
\[ \rho = \frac{p}{R_{\text{specific}} T} \]  

(4.8)

where

- \( R_{\text{specific}} \) is the the specific gas constant, is 287.058 J/(kgK) for dry air in SI units.
- \( T \) is absolute temperature which can be obtained from barometer sensor.

\[ \Delta P = \text{sensorValue} \times 4.8798393554 - 2493.6289146 \]  

(4.9)

**Figure 4.8.** Airspeed Calibration Graph

Airspeed sensor provides a digital 10-bit output value within a range zero to 1023. In order to define which output value corresponds to which differential pressure value, calibration is done using a Pressure Calibrator. The graph obtained from the calibration is given in Figure 4.8.

The trend of the graph gives the equation (4.9) from the sensor output value to the differential pressure value.
Listing 4.6. Airspeed Calculation

```c
float calculateAirSpeed()
{
    static float sensorValue = analogRead(A0);
    static float cur_pressure = sensorValue*4.8798393554 - 2493.62891461;
    if ((millis() - time) >= 20)
    {
        time = millis();
        air_pressure = cur_pressure*0.25 + air_pressure*0.75;
        if (air_pressure >= ref_pressure)
        {
            pressure_diff = air_pressure - ref_pressure;
        }
        else
        {
            pressure_diff = 0.0;
        }
        airspeed = sqrt(pressure_diff*airspeed_ratio);
        delay(1000);
    }
    return airspeed;
}
```

The code given in Listing 4.6 calculates the airspeed using the pressure difference value obtained from airspeed sensor. To convert analog signal to a pressure value, calibration is done.

4.1.5 Navigation Information

The navigation information is the most important part of the sensory data that is used to calculate bearing and distance of the target way point. GPS sensor uses Global Positioning Signals(GPS) to determine the device's real-time location on the Earth. It provides real-time longitude, latitude and the altitude information, which can be used to calculate the distance and the bearing between two different way points.

Since the Earth has a sphere shape, distance can be calculated just by subtracting latitude and the longitude information. The distance and the bearing information can be calculated using several different navigation methods. One of the most popular method is using the Haversine formula. Haversine formula is an equation important in navigation, giving great-circle distances between two points on sphere from their longitudes and latitudes[47].

For any two points on a sphere, the haversine of the central angle between them is given by: (All the equations taken from [48])
\[ haversin(d/r) = haversine(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\haversin(\lambda_2 - \lambda_1) \quad (4.10) \]

where

- \( haversin(\theta) = \sin^2(\theta/2) = \frac{(1 - \cos(\theta))}{2} \)
- \( r \) is the radius of the Earth (mean radius = 6,371 km)
- \( \phi_1, \phi_2 \): latitude of point 1 and latitude of point 2
- \( \lambda_1, \lambda_2 \): longitude of point 1 and longitude of point 2

If Eq. 4.10 is solved for \( d \) by applying the inverse \( haversine() \) or by using \( \arcsin() \) function

\[ d = rhaversin^{-1}(h) = 2r \arcsin(\sqrt{h}) \quad (4.11) \]

where \( h \) is \( haversin(d/r) \), or more explicitly:

\[ d = 2r \arcsin \left( \sqrt{haversine(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\haversin(\lambda_2 - \lambda_1)} \right) \]
\[ = 2r \arcsin \left( \sqrt{\sin^2 \left( \frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1)\cos(\phi_2)\sin^2 \left( \frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (4.12) \]

Eq. 4.12 shows the calculation of distance between two points located on earth using longitude and latitude information. These information will be provided by the GPS sensor.

Bearing is the term used in navigation, which is regularly defined as the direction (relative to true north) in which your destination lies. It can be calculated using latitude and longitude information of two points.

This formula [Eq. 4.13] is for the initial bearing which will take vehicle from the start point to the end point, if followed in a straight line along a great-circle arc:[48]

\[ bearing = \arccos \left( \frac{\sin(lat_2) - \sin(lat_1)\cos(d)}{\cos(lat_1)\sin(d)} \right) \quad (4.13) \]

The bearing information will be used as a Setpoint for the Direction Holder controller. When the navigation way points are defined, distance and the bearing between two way points will be calculated using Eq 4.12 and 4.13 and these values will be send to Direction Holder and the Navigation Controller to keep the vehicle on desired trajectory.

### 4.2 Control Algorithms

This section addresses the control algorithm used in autonomous part of the code. As mentioned before, this project aims to develop an autonomous system with several features such as: Altitude Holder, Direction Holder, Speed Holder and the Waypoint Controller. In the previous section,
data acquisition is explained, which forms the structures of the controller algorithms. Here, the control algorithm will be explained with block diagrams, which shows how the data is used as an input and which controller uses which controller components (fans, servos, tails).

### 4.2.1 Altitude Holder

Altitude Controller is the part of the code which keeps the blimp at desired altitude defined by user. This controller uses altitude information provided by pressure sensor as an input. PID controller uses predefined target altitude value as a Setpoint and sends angle value to the servos as an output. This servo drives the tilt angle of the ducted fans. For example if the target altitude is at a higher point than the current blimp’s altitude, PID controller increase the tilt angle of the ducted fans and the blimp gains altitude.

The block diagram in Figure 4.9 shows processing steps of the controller.

![Altitude Holder Control Diagram](image)

**Figure 4.9.** Altitude Holder Control Diagram

Here, Target Altitude is the one set by the user. This value can be set directly by defining an altitude Error is the difference between the actual current altitude value and the Setpoint. PID controller tries to decrease this error value to the zero. It does this by sending Output commands to the servos, which is a PWM signal, and the feedback (current altitude value) is sent back to the beginning of the loop and the new error value is calculated.
Listing 4.7. Altitude Holder Algorithm

do{
    Input=currentAlt;
    if(targetAlt > currentAlt) //if target is at a higher altitude
    {
        servo[1].write(0); // Set ducted fans to takeoff position
        altPID.SetControllerDirection(DIRECT);
        if (delh > 10.0)
        {
            altPID.SetTunings(aggKp, aggKi, aggKd);
        }
    else if (delh < 10.0)
    {
        altPID.SetTunings(consKp, consKi, consKd);
    }
    }
    else if(targetAlt < currentAlt)
    {
        servo[1].write(180);
        altPID.SetControllerDirection(DIRECT);
        if (delh > 10.0)
        {
            altPID.SetTunings(aggKp, aggKi, aggKd);
        }
    else if (delh <10.0)
    {
        altPID.SetTunings(consKp, consKi, consKd);
    }
    }
    altPID.Compute();
    Output=map(Output,0,255,0,100);
    Serial.print("PID Motor Speed: ");Serial.println(Output);
    for (int i=0; i<MOTORS; i++)
    {
        motor[i].SetSpeed(Output); //Set motor speed using pid output
    }
    delay(100);
}while(delh!=0.0);
4.2.2 Direction Holder

Direction controller is the part of the code which keeps the blimp on the desired heading. Direction Holder algorithm uses bearing information that is calculated from the navigation points defined by the user.

Listing 4.8. Direction Controller Algorithm

```c
if (Setpoint <= 358 && Setpoint > 1) {
    if (gap >= -1 && gap <= 180) // target direction is on the right
        turnRight();
    else if (gap > 180) // if the distance is greater than 180 degrees, turn reverse direction
        turnLeft();
    else if (gap <= -180 && gap >= -180) // target is on the left
        turnLeft();
    else if (gap < -180) // if the distance is greater than 180 degrees, turn reverse direction
        turnRight();
    else
        stopMotors();
}
else if (Setpoint >= 0 && Setpoint <= 1)
    if (gap <= -1)
        turnLeft();
    else if (gap >= 1)
        if (Input >= 180 && Input <= 358)
            turnRight();
        else
            turnLeft();
    else if (Setpoint >= 358 && Setpoint <= 359)
        if (gap >= 1)
```
4.2.3 Speed Controller

Speed controller is the part of the code which keeps the blimp on desired speed defined by user. Speed information received from air speed sensor is used as Input for the PID controller. The PID controller calculates the differences between the Setpoint and the Input and adjust RPM values of the ducted fans to control the speed of the blimp.

The block diagram in Figure 4.11 shows the processing steps of the controller.

![Figure 4.11. Speed Controller Block Diagram](image)

Listing 4.9. Airspeed Controller Algorithm

```java
Input = currentSpeed;
if (targetSpd > currentSpd) // if vehicle is faster than targetSpeed
{
    spdPID.SetControllerDirection(DIRECT);
    if (dels > 5.0)
    {
        spdPID.SetTunings(aggKp, aggKi, aggKd);
    }
    else if (dels < 5.0)
    {
        spdPID.SetTunings(consKp, consKi, consKd);
    }
}
```

```c

```
```c
spdPID.Compute();
Output=map(Output,0,255,−10,10);
for (int i=0; i<MOTORS; i++)
{
    value=motor[i].speed+Output;  // Increase motor speed using pid output
    motor[i] = motor[i].Setspeed(value); // Write new motor speed.
}
value=0;
delay(1000); // refresh motor speed every 1 sec
}while(delS!=0.0);
```
Chapter 5

Flight Tests

This chapter addresses the first indoor flight test results. All the tests, in this part are performed using small 44 inch blimp prototype and the all the tests are performed indoor, to compensate exterior disturbances.

5.1 Direction Holder Tests

These test series consists of several test cases, in order to determine the best throttle position and target point offset value. PID controller is a common control method to stabilize a system. The PID function uses system feedback to continuously control a dynamic process. The purpose of PID control is to keep a process running as close as possible to a desired Set Point. But some systems doesn’t need to use a complex controller like this. Before deciding a PID controller test, another common type of control can be tested, which is On-Off control. The first flight tests are performed to determine that if the system needs to be controlled by a PID controller for Direction Holder. In this purpose the blimp is tethered via a thin soft cable in order to compensate the side drifts during direction corrections, as much as possible.

In order to obtain more comparable results, a simple test code is written to automate the test flights. The code, given in Listing 5.1 assigns the new randomly defined Setpoint value on every 60 seconds. Real time flight data is monitored through the wireless sensor.

Listing 5.1. Airspeed Controller Algorithm

```c
void testdirection()
{
    if(millis()>30000) // start 3 seconds after you turned on arduino
    {
        currentTime=millis();
    }
}  
```
if (currentTime >= (loopTime+60000)) // change Setpoint on every 60 sec.
{
    Setpoint = test[i];
    loopTime = currentTime;
    i = i + 1;
}
}

5.1.1 Case 1: Throttle Position: %30, Target Offset: ±5°

In this case, throttle position of the both fans are limited to a fixed rpm speed which is %30 under 9V power supply. The target direction offset is set to ±5 degrees. The controller will keep steering until the vehicle real-time heading value reaches the range between the Setpoint ± 5. When the blimp heading gets in this range, all the motors stops running. Having a target offset is required in since the blimp tends to keep steering when the fans stop running because of its inertia.

Real-time heading variation is given in Figure 5.1. Setpoint is changed by the code automatically in every 60 seconds.

![Figure 5.1. Direction Holder Calibration Test - Throttle Position: %30, Target Offset: ±5°](image)

The Figure 5.1 shows that vehicle oscillates between a range of ±30 degrees around the Setpoint. And also it has some overshooots when the Setpoint is changed. Bigger changes in Setpoint causes more overshoots and vehicle doesn’t tend to stabilize.
The frequency of oscillation is:

\[ f = \frac{4.5}{60} = 0.075 \text{ hertz} \]  

(5.1)

5.1.2 Case 2: Throttle Position: %30, Target Offset: ±1°

In this case motors speeds are set to %30 of full throttle value (9V) and the target offset value is set to ±1 degrees.

![Figure 5.2. Direction Holder Calibration Test - Throttle Position: %30, Target Offset: ±1°](image)

Figure 5.2 shows that vehicle oscillates between a range of ±20 degrees around the Setpoint. It still has some overshoots when the Setpoint is changed.

The frequency of oscillation is:

\[ f = \frac{6}{60} = 0.1 \text{ hertz} \]  

(5.2)

5.1.3 Case 3: Throttle Position: %50, Target Offset: ±5°

In this case motors speeds are set to %50 of full throttle value (9V) and the target offset value is set to ±1 degrees.

The Figure 5.3 shows that vehicle oscillates between a range of ±25 degrees around the Setpoint. The oscillation range is not as high as the first case but bigger than the second case. It still has some overshoots when the Setpoint is changed. Oscillation period is faster than the second case.
The frequency of oscillation is:

\[ f = \frac{5}{60} = 0.0833 \text{ hertz} \] (5.3)

5.1.4 Case 4: Throttle Position: %50, Target Offset: ±1°

In this case, motors speeds are set to %50 of full throttle value(9V) and the target offset value is set to ±1 degrees.
Figure 5.4 shows that vehicle oscillates between a range of ±15 degrees around the Setpoint. It still has some overshoots when the Setpoint is changed.

\[ f = \frac{6}{60} = 0.1 \text{ hertz} \]  

(5.4)

Figure 5.5 shows the all case results on the same graph. At every throttle position blimp has a tendency to oscillate. It can be seen that, in Case 4 (green line), blimp oscillates in a smaller range than the other cases. but when the Setpoint is changed it overshoots more than the other cases.

![Direction Controller Flight Test - Fixed Setpoint, Variable Throttle and Target Offset Value](image)

**Figure 5.5.** Direction Holder Calibration Test - All Cases

The reason of the oscillation is that, the lag in the system response time causes the blimp to overshoot and oscillate around the Set Point. Despite several throttle positions and target offset ranges are tested, the system keep oscillating. This oscillations can be compensated using PID controller methods with suitable coefficients. To find suitable PID coefficients several flight tests should be performed.

All of the PID controllers(Altitude Holder, Direction Holder, Speed Controller) of these project are coded. However, it could not be able to present the result within the time schedule of this study. All of these flight tests are planned as a future study. The altitude holder should be during vertical take off and should be monitored if the blimp can keep the desired altitude by tilting the ducted fans. Since the blimp is tethered to the ground during the first Direction Holder tests, the torque of the twisted cable used for holding the blimp affects the dynamics of the system. Thus, a test stand with one degree of freedom on z axis, can be designed for Direction Holder tests. Speed Holder tests could be tested indoor, in big room.
Chapter 6

Conclusions

This thesis aims to develop an uninhabited air vehicle for surveillance and monitoring missions, using electric ducted fans, Arduino on-board controller and several low cost sensors. As an air vehicle platform, a 10 foot envelope is chosen. To find the optimal vehicle configuration, several sketches are drawn and a configuration with two main electric driven ducted fans as main thruster and a small tail rotor, embedded to the lower vertical tail, is chosen. Vehicle weight distributions are calculated.

In order to reach the project objectives, a low cost, open source on-board micro-controller (Arduino), equipped with six different low cost, high resolution sensors. To estimate the vehicle attitude, ITG3200 Gyroscope and ADXL345 Accelerometer sensory outputs are merged with Kalman Filter to obtain more reliable and un-noisy data. To determine the vehicle heading, LSM303 tilt compensated magnetometer is used. Altitude information is provided by the BMP085 Pressure and Temperature sensor. 20 Channels EM408 GPS modem is used to determine the coordinate information for outdoor flights. An airspeed speed sensor equipped with MPXV7002DP Differential Pressure sensor is used to measure vehicle speed during flight. A Low-pass filter is applied to the all sensory output to get rid of the noisy data. The wireless communication is granted by two 2.4 GHz XBee Pro embedded RF modules.

The micro-controller on the board is programmed using the Arduino programming language, based on Wiring. For the future applications several open source libraries are written with object oriented C++ programming language such as, MotorController, ServoController. Sensor data acquisition are provided using several open source libraries supplied by the sensor companies and open source communities. A command-based user interface with several options is developed for monitoring the sensory output and to control the vehicle manually. Three different controllers, Altitude Controller, Direction Holder, Speed Controller, are written using PID method for smooth trajectory.

For the proof of concept, a small 44 inch prototype is manufactured. This prototype is equipped with a gondola which is manufactured with balsa wood, to carry all the mandatory
sensors for vehicle dynamics, the on-board micro-processor unit, flight components such as 40 mm diameter ducted fans and mini-digital servos and batteries. In order to drive 9V small ducted fans, a motor controller circuit [Figure 2.9] is manufactured using MOSFET type transistors. Since all the digital sensors that are used supports \(I^2C\) communication protocol, a sensor circuit[2.10] is designed and all the sensor are put on this board. This board gives the easiness of communicating with all sensor from the same communication pins.

The implementation of ducted fans as vectoring thrusters allowed to improve the maneuverability of the UAV. Previous studies, done in the Turbomachinery Lab., on ducted fans shows that ducted fans are much more efficient and easy to integrate into air vehicles, especially airships. In the time schedule of this thesis, we couldn’t have chance to compare the performance of ducted fans on an application of a blimp , but the specific features, such as safety and low noise operation granted by ducted fans, are observed.

After the preparation process is completed, we started to perform indoor tests using small scale prototype, in order to determine if the system can be controlled by a On-Off control method instead of PID control method. First tests results are listed in Chapter 5. Results showed that, On-Off control method is not being enough to reach a stable flight regime and it is seen that blimp keeps oscillating forever. System should controlled by a complex control method such as a PID controller.

6.1 Future Works

Development of an autonomous vehicle requires a comprehensive set of engineering tasks, embracing knowledge as diverse as aerodynamics, vehicle dynamics, electronics, programming, signal processing and prototyping.[4] Each of these tasks takes considerable time to complete. Especially designing such a complex system from the beginning requires a steep learning curve.

For the future study, improvements and complements can be separated into several topics:

Software:

- The controller algorithm that is developed in this project takes account the system dynamics as decoupled system. However, a fully autonomous system dynamics works as coupled system. So for the future study, all of the control algorithms could be redesigned as coupled.

- The Navigation Controller algorithm has not been completed yet. This controller could be completed.

- GPS sensor data acquisition is missing in the code right now. It could be completed.

- The software has a capability of manual flight via the keyboard commands. These can be improved to use a joystick instead of keyboard.

- Several emergency cases could be developed in case of losing the link between the ground station and the blimp.
• Micro-controller processing performance could be tested and optimal of processing times could be improved.

Hardware:
• Motor Controller board could be tested and voltage spikes could be prevented.
• A PCB board could be designed and all the electronic components could be put together.
• A voltage drop warning system could be developed.
• All the wiring could be redesigned.

Vehicle:
• All the autopilot system could be tested with actual size target blimp.
• Ducted fans locations could be optimized for better flight efficiency.
Appendix A

Attitude Estimation Equations

This part gives the derivation of equations related with the calculation of roll and pitch angles from accelerometer output. The convention will be adopted that the accelerometer output is negated to give value +1g in any axis aligned with the earth’s downward gravitational field.

With this assumption, a three-axis accelerometer oriented in the earth’s gravitational field $\mathbf{g}$ and undergoing linear acceleration $\mathbf{a}_r$ measured in the earth’s reference frame $\mathbf{r}$, will have output $\mathbf{G}_p$ given by: (All the equations are obtained from:[49])

$$\mathbf{G}_p = \begin{pmatrix} G_{px} \\ G_{py} \\ G_{pz} \end{pmatrix} = R(g - a_r) \tag{A.1}$$

where $R$ is the rotation matrix describing the orientation of the air vehicle relative to the earth’s and $\mathbf{G}$ is the accelerometer output(measured in the native accelerometer units of g).

A.1 Pitch and Roll Estimation

The orientation of the air vehicle can be defined by its roll, pitch and yaw rotations from an initial position. The roll, pitch and yaw rotation matrices, which transform a vector (such as the earth’s gravitational field vector $\mathbf{g}$) under a rotation of the coordinate system of Figure 4.1 by angles $\phi$ in roll, $\theta$ in pitch and $\psi$ in yaw about the x, y and z axes respectively, are:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix} \tag{A.2}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \tag{A.3}$$
\[
R_z(\psi) = \begin{pmatrix}
\cos\psi & \sin\psi & 0 \\
-sin\psi & \cos\psi & 0 \\
0 & 0 & 1
\end{pmatrix}
\] (A.4)

where \(R\) is the rotation matrix describing the orientation of the vehicle relative to the earth’s coordinate frame.

There are six possible orderings of these three rotation matrices and, in principle, all are equally valid. These are \(R_{xyz}, R_{yzx}, R_{yxz}, R_{yzx}, R_{zxy}, R_{zyx}\). However four of these rotation sequences \((R_{yxz}, R_{yxz}, R_{yzx}, R_{zxy})\) can be immediately rejected as being unstable for determining the vehicle orientation. The rotation sequence \(R_{xyz}\) is widely used in the aerospace industry and is termed the ‘aerospace rotation sequence’. So in this part we will only derive \(R_{xyz}\) vector sequence.

The rotation matrices do not, however, commute meaning that the composite rotation matrix \(R\) depends on the order in which the roll, pitch and yaw rotations are applied. It is instructive to compute the values of the six possible composite rotation matrices \(R\) and to determine their effect on the earth’s gravitational field of \(1g\) initially aligned downwards along the \(z\)-axis.

\[
R_{xyz} \begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix} = R_x(\phi)R_y(\theta)R_z(\psi) \begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix}
\] (A.5)

\[
= \begin{pmatrix}
\cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\
\cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \cos\phi\cos\psi + \sin\theta\sin\phi\sin\psi & \cos\theta\sin\phi \\
\cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \cos\psi\sin\phi & \cos\phi\cos\psi
\end{pmatrix}
\] (A.6)

\[
= \begin{pmatrix}
-\sin\theta \\
\cos\theta\sin\phi \\
\cos\theta\cos\phi
\end{pmatrix}
\] (A.7)

It can be readily seen from Equations A.5 to A.7 that composite rotation matrix and values of the measured gravitational vector are all different. A consequence is that roll, pitch and yaw rotation angles are meaningless without first defining the order in which these rotations are to be applied.

Equation A.7 can be rewritten in the form of Equation A.8 relating the roll \(\phi\) and pitch \(\theta\) angles to the normalized accelerometer reading \(G_p\):

\[
\frac{G_p}{\lVert G_p \rVert} = \begin{pmatrix}
-\sin\theta \\
\cos\theta\sin\phi \\
\cos\theta\cos\phi
\end{pmatrix} \Rightarrow \frac{1}{\sqrt{G_{px}^2 + G_{py}^2 + G_{pz}^2}} \begin{pmatrix}
G_{px} \\
G_{py} \\
G_{pz}
\end{pmatrix} = \begin{pmatrix}
-\sin\theta \\
\cos\theta\sin\phi \\
\cos\theta\cos\phi
\end{pmatrix}
\] (A.8)

Solving for the roll and pitch angles from Equation A.8, and using the subscript \(xyz\) to denote that the roll and pitch angles are computed according to the rotation sequence \(R_{xyz}\), gives:
\[ \tan \theta_{xyz} = \frac{G_{py}}{G_{pz}} \]  \hspace{1cm} (A.9)

\[ \tan \theta_{xyz} = \left( \frac{-G_{pz}}{G_{py} \sin \phi + G_{pz} \cos \phi} \right) = \frac{-G_{pz}}{\sqrt{G_{py}^2 + G_{pz}^2}} \]  \hspace{1cm} (A.10)
Source Code

B.1 Source Code

MainCode.pde

Listing B.1. Main Code

```cpp
#include <ADXL345.h>
#include <Barometer.h>
#include <Servo.h>
#include <MotorControl.h>
#include <HMC.h>
#include <Wire.h>
#include <SoftwareSerial.h>
#include <TinyGPS.h>
#include <PID_v1.h>
#include <SignalFilter.h>

// Configuration libraries
#include "defines.h"
#include <commons.h>

/////////////////////////Create objects////////////////
MotorControl motor[MOTORS];
Servo servo[SERVOS];
Barometer BarSensor;
ADXL345 adxl; // variable adxl is an instance of the ADXL345 library
TinyGPS gps;

SoftwareSerial nss(3, 4);
static int motorPins[MOTORS] = {MotorInitPin};
static int servoPins[SERVOS] = {ServoInitPin};
static int commandFlag=0;
static int MotorSet=0;
static int ServoSet=0;
```
static char ReadInput;
int ModeChange=0;

///////////SerialCom Variables/////////
static int value;
static int id=-1;
static int i;
static int CurrentValue;

////////Barometer Variables////////
short currentTemp;
float currentPres;
short currentAlt;
static short SeaLevelP = 1024.0;
short groundTemperature;
float groundPressure;
short groundAltitude;

////////AirSpeedSensor/////////
static float ref_pressure, air_pressure, pressure_diff;
static float airspeed_ratio = 1.5191;
static float airspeed;
float time;

////////Compass////////
unsigned int heading;

////////Accelometer////////
unsigned long timer;

////////FlightMode////////
static int flightMode = 0;

//PID Libraries
SignalFilter filter;

void manualFlight();

void setup()
{
  Serial.flush();
  pinMode(A5, LOW);
  pinMode(A4, LOW);
  Serial.begin(BAUD);
  Wire.begin();
  initMotor();
  initServo();
  initAirSpeed();
initCompass();
initBarometer();
initGyro();
initGPS();
initAccelometer();
filter.begin();
filter.setFilter('b');
filter.setOrder(1);
Serial.println("Sensors Setup completed with Success!");
}

void loop()
{
    // User command based interface. Waits for predefined characters to enter menu.
    if (Serial.available())
    {
        switch (readSerial()){
            // To enter Flight Mode menu, input 'f' character
            case 'f':
                Serial.println("************FlightMode**********\n");
                Serial.print("Current Flight Mode: ");
                if(flightMode == false)
                    Serial.println("Manual");
                else
                    Serial.println("Autopilot");
                commandFlag=3;
                flightMode=true;
                autoPilot();
                Serial.println("**********************************************************************\n");
                break;

            // To enter Motor Control menu, input 'm' character
            case 'm':
                Serial.println("Motor is chosen.");
                if(MotorSet==0)
                    Serial.println("Set the motor pins first! Enter: 'S'");
                commandFlag=1;
                ModeChange=1;
                break;

            // To enter Servo menu, input 's' character
            case 's':
                Serial.println("Servo is chosen.");
                if(ServoSet==0)
                    Serial.println("Set the servo pins first! Enter: 'S'");
                commandFlag=2;
                ModeChange=1;
                break;

            // To enter Altitude Monitoring menu, input 'p' character

case 'p':
    Serial.println("************Altitude Information**********
    do{
        printAltitude();
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("***************************************
    break;

//To enter Altitude Monitoring menu, input 'h' character

    case 'h':
    Serial.println("************Heading**********
    do{
        printHeading();
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("***************************************
    break;

//To enter Acceleration Monitoring menu, input 'g' character

    case 'g':
    Serial.println("************Accelerometer**********
    do{
        printAccAngles(ReadInput);
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("***************************************
    break;

//To enter Gyroscope Monitoring menu, input 'r' character

    case 'r':
    Serial.println("************Gyro**********
    do{
        printGyroAngles();
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("***************************************
    break;

//To enter Attitude Monitoring menu, input 'o' character

    case 'o':
    Serial.println("************Rotation -(Comp, Acc, Gyro)**********
    do {
        calculateRotation();
        commandFlag=0;
        readSerial();
        delay(50);
    }while(ReadInput!='q');
// To enter Airspeed Monitoring menu, input 'h' character
case 'v':
    Serial.println("************ AirSpeed **********
    do{
        calculateAirSpeed();
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("****************************************");
    break;

// To enter GPS Monitoring menu, input 'h' character
case 't':
    Serial.println("************ GPS Coordinates **********
    do{
        calculateCoord();
        commandFlag=0;
        readSerial();
    }while(ReadInput!='q');
    Serial.println("****************************************");
    break;
}

// This part checks the commandFlag and transmit
// the character if suitable commandFlag is defined.
switch(commandFlag){
    case 1:
        controlMotor(ReadInput);
        break;
    case 2:
        controlServo(ReadInput);
        break;
    case 3:
        autoPilot();
}
}

Serial.flush();

Accelerometer.pde

Listing B.2. Accelerometer
double zeroValue[3];

// Initialize at leveled position
void initAccelometer()
{
    Serial.println("Initializing Accelerometer Sensor...");
adxl.powerOn();
}
adxl.writeTo(0x31, 0x09);
// set sensitivity to 16 avail: 2.4.8.16
    adxl.setRangeSetting(16);

    timer = micros();
    // store 100 value and find average
    int valueX, valueY, valueZ;
    int x, y, z;
    for (int i = 0; i < 100; i++)
    {
        adxl.readAccel(&x, &y, &z);
        valueX += x;
        // Serial.println(valueX);
        valueY += y;
        // Serial.println(valueY);
        valueZ += z;
        // Serial.println(valueZ);
        delay(10);
    }
    // Store leveled position sensor reading in an array
    zeroValue[0] = valueX / 100;
    zeroValue[1] = valueY / 100;
    zeroValue[2] = valueZ / 100;
    Serial.println("Accelometer Sensor Initialized.");
    Serial.println("----------------------------------");
}

double getXangle()
{
    double accXval = (double)readAcc(0) - zeroValue[0];
    double accZval = (double)readAcc(2) - zeroValue[2];
    double angle = (atan2(accXval, accZval) + PI) * RAD_TO_DEG;
    return angle;
}

double getYangle()
{
    double accYval = (double)readAcc(1) - zeroValue[1];
    double accZval = (double)readAcc(2) - zeroValue[2];
    double angle = (atan2(accYval, accZval) + PI) * RAD_TO_DEG;
    return angle;
}

do
{
    Serial.print("X angle: "); Serial.print(getXangle()); Serial.print("\t");
}
Serial.print("Y angle: "); Serial.print(getYangle());
Serial.print("\n");
delay(100);
}while(ch != '*');

// Read sensor
int readAcc(int i)
{
    int val[3];
    adxl.readAccel(&val[0],&val[1],&val[2]);
    return val[i];
}

AltitudeHolder.pde

Listing B.3. AltitudeHolder

boolean altitudeControl(int targetAlt)
{
    double Setpoint, Input, Output;  // define required variables for PID
    double aggKp=2, aggKi=0.5, aggKd=0.1;
    double consKp=5, consKi=0.02, consKd=0.01;

    PID altPID(&Input, &Output, &Setpoint,consKp,consKi,consKd, DIRECT);

    // PID Setup
    Setpoint=targetAlt;
    altPID.SetMode(AUTOMATIC);

    currentAlt=calculateAltitude();
    int delh = abs(targetAlt - currentAlt);
    Serial.print("Distance to Reach Target: ");Serial.println(delh);
    Serial.print("Current Altitude (m): ");Serial.println(currentAlt);
    Serial.print("Target Altitude (m): ");Serial.println(targetAlt);

    do
    {
      Input=currentAlt;
      if(targetAlt > currentAlt) // if target is at a higher altitude
      {
        servo[1].write(0);  // Set ducted fans to takeoff position
        altPID.SetControllerDirection(DIRECT);
        if (delh > 10.0)
        {
          altPID.SetTunings(aggKp, aggKi, aggKd);
        }
      }
      else if (delh < 10.0)
      {
        altPID.SetTunings(consKp, consKi, consKd);
      }
    }
else if (targetAlt < currentAlt) {
    servo[1].write(180);
    altPID.SetControllerDirection(DIRECT);
    if (delh > 10.0) {
        altPID.SetTunings(aggKp, aggKi, aggKd);
    } else if (delh < 10.0) {
        altPID.SetTunings(consKp, consKi, consKd);
    }
    altPID.Compute();
    Output = map(Output, 0, 255, 0, 100); 
    Serial.print("PID Set Motor Speed: "); Serial.println(Output);
    for (int i = 0; i < MOTORS; i++) {
        motor[i].SetSpeed(Output); // Set motor speed using pid output
    }
    delay(100);
} while (delh != 0.0);

Barometer.pde

Listing B.4. Barometer

// create altitude variables
short currentTemperature;
float currentPressure;
short currentAltitude;

// print the altitude screen for monitoring
static void printAltitude()
{
    currentTemperature = 0.1 * BarSensor.bmp085GetTemperature(BarSensor.
        bmp085ReadUT());
    currentPressure = BarSensor.bmp085GetPressure(BarSensor.bmp085ReadUP());
    currentAltitude = 44330*(1-pow((currentPressure/100.0/SeaLevelP),(1/5.255))); 
    // altitude = pressure / SeaLevelP;
    Serial.print("Temperature: ");
    Serial.print(currentTemperature, DEC);
    Serial.println(" deg C");
    Serial.print("Pressure: ");
    Serial.print(currentPressure, DEC);
    Serial.println(" hPa");
Serial.print("Relative Altitude to SL: ");
Serial.print(currentAltitude, DEC);
Serial.println(" m");
Serial.println();
delay(1000);
}

// save ground information before take off
static void saveGroundAltitude()
{
groundTemperature = BarSensor.bmp085GetTemperature(BarSensor.bmp085ReadUT());
groundPressure = BarSensor.bmp085GetPressure(BarSensor.bmp085ReadUP());
groundAltitude = 44330*(1-pow((groundPressure/100.0/SeaLevelP),(1/5.255)));
}

// Calculate altitude relative to the ground
static short calculateAltitude()
{
currentTemperature = BarSensor.bmp085GetTemperature(BarSensor.bmp085ReadUT());
currentPressure = BarSensor.bmp085GetPressure(BarSensor.bmp085ReadUP());
currentAltitude = 44330*(1-pow((currentPressure/100.0/SeaLevelP),(1/5.255)));
delay(1000);
return abs(currentAltitude-groundAltitude);
}

// Initialize pressure sensor
void initBarometer()
{
Serial.println("Initializing Pressure and Temperature Sensor...");
BarSensor.bmp085Calibration();
saveGroundAltitude();
Serial.println("Pressure Sensor Initialized.");
Serial.println("/////////////////////////////////////////");
}

gpsSensor.pde

Listing B.5. GPS

// Create required variables for GPS sensor reading
static void gsdump(TinyGPS &gps);
static bool feedgps();
static void print_float(float val, float invalid, int len, int prec);
static void print_int(unsigned long val, unsigned long invalid, int len);
static void print_date(TinyGPS &gps);
static void print_str(const char *str, int len);

// Calculate coordinates
void calculateCoord()
{
  bool newdata = false;
  unsigned long start = millis();

  // Every second we print an update
  while (millis() - start < 1000)
  {
    if (feedgps())
      newdata = true;
  }
  gpsdump(gps);
}

// Initialize GPS sensor
void initGPS()
{
  Serial.println("Initializing GPS Sensor...");
  nss.begin(4800);
  Serial.println("GPS Sensor Initialized.");
  Serial.println("///////////////////////////////////");
}

// Read sensor device
static void gpsdump(TinyGPS &gps)
{
  float flat, flon;
  unsigned long age, date, time, chars = 0;
  unsigned short sentences = 0, failed = 0;
  static const float LONDON_LAT = 51.508131, LONDON_LON = -0.128002;

  print_int(gps.satellites(), TinyGPS::GPS_INVALID_SATELLITES, 5);
  print_int(gps.hdop(), TinyGPS::GPS_INVALID_HDOP, 5);
  gps.f_get_position(&flat, &flon, &age);
  print_float(flat, TinyGPS::GPS_INVALID_F_ANGLE, 9, 5);
  print_float(flon, TinyGPS::GPS_INVALID_F_ANGLE, 10, 5);
  print_int(age, TinyGPS::GPS_INVALID_AGE, 5);

  print_date(gps);

  print_float(gps.f_altitude(), TinyGPS::GPS_INVALID_F_ALTITUDE, 8, 2);
  print_float(gps.f_course(), TinyGPS::GPS_INVALID_F_ANGLE, 7, 2);
  print_float(gps.f_speed_kmph(), TinyGPS::GPS_INVALID_F_SPEED, 6, 2);
  print_str(gps.f_course() == TinyGPS::GPS_INVALID_F_ANGLE ? "*** " : TinyGPS::cardinal(gps.f_course()), 6);
  print_int(flat == TinyGPS::GPS_INVALID_F_ANGLE ? 0UL : (unsigned long)TinyGPS::distance_between(flat, flon, LONDON_LAT, LONDON_LON) / 1000, 0xFFFFFFF, 9);
  print_float(flat == TinyGPS::GPS_INVALID_F_ANGLE ? 0.0 : TinyGPS::course_to(flat, flon, 51.508131, -0.128002), TinyGPS::GPS_INVALID_F_ANGLE, 7, 2);
// Print out with sensory data in int
static void print_int(unsigned long val, unsigned long invalid, int len)
{
    char sz[32];
    if (val == invalid)
        strcpy(sz, "*******");
    else
        sprintf(sz, "%ld", val);
    sz[len] = 0;
    for (int i=strlen(sz); i<len; ++i)
        sz[i] = ' ';  
    if (len > 0)
        sz[len-1] = ' ';
    Serial.print(sz);
    feedgps();
}

// Print out with sensory data in float
static void print_float(float val, float invalid, int len, int prec)
{
    char sz[32];
    if (val == invalid)
    {
        strcpy(sz, "*******");
        sz[len] = 0;
        if (len > 0)
            sz[len-1] = ' ';
        for (int i=7; i<len; ++i)
            sz[i] = ' ';
        Serial.print(sz);
    }
    else
    {
        Serial.print(val, prec);
        int vi = abs((int)val);
        int flen = prec + (val < 0.0 ? 2 : 1);
        flen += vi >= 1000 ? 4 : vi >= 100 ? 3 : vi >= 10 ? 2 : 1;
        for (int i=flen; i<len; ++i)
            Serial.print(" ");
    }
}
//print out the date
static void print_date(TinyGPS &gps)
{
  int year;
  byte month, day, hour, minute, second, hundredths;
  unsigned long age;
  gps.crack_datetime(&year, &month, &day, &hour, &minute, &second, &hundredths, &age);
  if (age == TinyGPS::GPS_INVALID_AGE)
    Serial.print("******* ******* ");
  else
    {
    char sz[32];
    sprintf(sz, "%02d/%02d/%02d %02d:%02d:%02d ",
            month, day, year, hour, minute, second);
    Serial.print(sz);
    }
  print_int(age, TinyGPS::GPS_INVALID_AGE, 5);
  feedgps();
}

static void print_str(const char *str, int len)
{
  int slen = strlen(str);
  for (int i = 0; i < len; ++i)
    Serial.print(i < slen ? str[i] : ' ');
  feedgps();
}

static bool feedgps()
{
  while (nss.available())
    {
      if (gps.encode(nss.read()))
        return true;
    }
  return false;
}

Gyroscope.pde

Listing B.6. Gyroscope

//define device addresses
char WHO_AM_I = 0x00;
char SMPLRT_DIV = 0x15;
char DLPF_FS = 0x16;
char GYRO_XOUT_H = 0x1D;
char GYRO_XOUT_L = 0x1E;
char GYRO_YOUT_H = 0x1F;
char GYRO_YOUT_L = 0x20;
char GYRO_ZOUT_H = 0x21;
char GYRO_ZOUT_L = 0x22;

// This is a list of settings that can be loaded into the registers.
// DLPF, Full Scale Register Bits
// FS_SEL must be set to 3 for proper operation
// Set DLPF_CFG to 3 for 1kHz Fint and 42 Hz Low Pass Filter
char DLPF_CFG_0 = 1<<0;
char DLPF_CFG_1 = 1<<1;
char DLPF_CFG_2 = 1<<2;
char DLPF_FS_SEL_0 = 1<<3;
char DLPF_FS_SEL_1 = 1<<4;

// I2C devices each have an address. The address is defined in the data sheet
// for the device. The ITG-3200 breakout board can have different address
// depending on how
// the jumper on top of the board is configured. By default, the jumper is
// connected to the VDD pin. When the jumper is connected to the VDD pin the
// I2C address
// is 0x69.
char itgAddress = 0x69;

static double del[3];

double gyroXangle = 180;
double gyroYangle = 180;
double gyroZangle = 180;

static double gyroXrate, gyroYrate, gyroZrate;

// The loop section of the sketch will read the X, Y and Z output rates from the
// gyroscope and output them in the Serial Terminal
void calculateGyroAngle()
{

    // Create variables to hold the output rates.

    // Read the x, y and z output rates from the gyroscope.
    gyroXrate = -((double)readX()/14.375);
    gyroXangle += gyroXrate*((double)(micros()-timer)/1000000);

    gyroYrate = -((double)readY()/14.375);
    gyroYangle += gyroYrate*((double)(micros()-timer)/1000000);

    gyroZrate = -((double)readZ()/14.375);
gyroZangle += gyroZrate * ((double)(micros()-timer)/1000000);

timer = micros();

// Print the output rates to the terminal, separated by a TAB character.
// Serial.print("X: ");
Serial.print(gyroXangle);
Serial.print(‘	’);
// Serial.print("Y: ");
Serial.print(gyroYangle);
Serial.print(‘	’);
// Serial.print("Z: ");
Serial.println(gyroZangle);

// Wait 10ms before reading the values again. (Remember, the output rate was
// set to 100Hz and 1 reading per 10ms = 100Hz.)
delay(10);

}

// Print gyro angle to the screen for monitoring
void printGyroAngles()
{
    calculateGyroAngle();
    Serial.print(gyroXangle);
    Serial.print(’	’);
    // Serial.print("Y: ");
    Serial.print(gyroYangle);
    Serial.print(’	’);
    // Serial.print("Z: ");
    Serial.println(gyroZangle);
}

// This function will write a value to a register on the ITG-3200.
// Parameters:
// char address: The I2C address of the sensor. For the ITG-3200 breakout the
// address is 0x69.
// char registerAddress: The address of the register on the sensor that should
// be written to.
// char data: The value to be written to the specified register.
void itgWrite(char address, char registerAddress, char data)
{
    // Initiate a communication sequence with the desired i2c device
    Wire.beginTransmission(address);
    // Tell the I2C address which register we are writing to
    Wire.write(registerAddress);
    // Send the value to write to the specified register
    Wire.write(data);
    // End the communication sequence
// This function will read the data from a specified register on the ITG-3200 and return the value.
// Parameters:
// char address: The I2C address of the sensor. For the ITG-3200 breakout the address is 0x69.
// char registerAddress: The address of the register on the sensor that should be read
// Return:
// unsigned char: The value currently residing in the specified register

unsigned char itgRead(char address, char registerAddress)
{
    // This variable will hold the contents read from the i2c device.
    unsigned char data = 0;

    // Send the register address to be read.
    Wire.beginTransmission(address);
    // Send the Register Address
    Wire.write(registerAddress);
    // End the communication sequence.
    Wire.endTransmission();

    // Ask the I2C device for data
    Wire.beginTransmission(address);
    Wire.requestFrom(address, 1);

    // Wait for a response from the I2C device
    if(Wire.available()){
        // Save the data sent from the I2C device
        data = Wire.read();
    }

    // End the communication sequence.
    Wire.endTransmission();

    // Return the data read during the operation
    return data;
}

// This function is used to read the X-Axis rate of the gyroscope. The function returns the ADC value from the Gyroscope
// NOTE: This value is NOT in degrees per second.
// Usage: int xRate = readX();

int readX(void)
{
    int data = 0;
    data = itgRead(itgAddress, GYRO_XOUT_H) << 8;
    data |= itgRead(itgAddress, GYRO_XOUT_L);
return ((data-del[0]));
}

// This function is used to read the Y-Axis rate of the gyroscope. The function returns the ADC value from the Gyroscope
// NOTE: This value is NOT in degrees per second.
// Usage: int yRate = readY();
int readY(void)
{
    int data = 0;
    data = itgRead(itgAddress, GYRO_YOUT_H)<<8;
    data |= itgRead(itgAddress, GYRO_YOUT_L);
    return ((data-del[1]));
}

// This function is used to read the Z-Axis rate of the gyroscope. The function returns the ADC value from the Gyroscope
// NOTE: This value is NOT in degrees per second.
// Usage: int zRate = readZ();
int readZ(void)
{
    int data = 0;
    data = itgRead(itgAddress, GYRO_ZOUT_H)<<8;
    data |= itgRead(itgAddress, GYRO_ZOUT_L);
    return ((data-del[2]));
}

// Sensor initialization
void initGyro(void)
{
    Serial.println("Initialization Gyro Sensor...");

    // Configure the gyroscope
    // Set the gyroscope scale for the outputs to +/-2000 degrees per second
    itgWrite(itgAddress, DLPF_FS, (DLPF_FS_SEL_0 | DLPF_FS_SEL_1 | DLPF_CFG_0));
    // Set the sample rate to 100 Hz
    itgWrite(itgAddress, SMPLRT_DIV, 9);
    itgCalibrate();

    Serial.println("Gyro Sensor Initialized.");
    Serial.println("/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n

// Device calibration
```cpp
void itgCalibrate()
{
    double valueX , valueY , valueZ ;
    for ( int i=0; i<100; i++)
    {
        valueX += readX();
        //Serial.println(valueX);
        valueY += readY();
        //Serial.println(readY());
        valueZ += readZ();
        //Serial.println(valueZ);
        delay(10);
    }

del[0]=valueX/100;
del[1]=valueY/100;
del[2]=valueZ/100;
}
```

**KalmanX.pde**

Listing B.7. KalmanX

```cpp
/* Kalman filter variables and constants */
const double Q_angleX = 0.001; // Process noise covariance for the
    // accelerometer - Sw
const double Q_gyroX = 0.003; // Process noise covariance for the gyro - Sw
const double R_angleX = 0.03; // Measurement noise covariance - Sv

double angleX = 180; // The angle output from the Kalman filter
double biasX = 0; // The gyro bias calculated by the Kalman filter
double PX_00 = 0, PX_01 = 0, PX_10 = 0, PX_11 = 0;
double dtX , yX , SX;
double KX_0 , KX_1 ;

double kalmanX(double newAngle , double newRate , double dtime) {
    // See also http://www.x-firm.com/?page_id=145
    // with slightly modifications by Kristian Lauszus
    // See http://academic.csuohio.edu/simond/courses/eec644/kalman.pdf and
    // http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf for more information
    dtX = dtime / 1000000; // Convert from microseconds to seconds

    // Discrete Kalman filter time update equations - Time Update ("Predict")
    // Update xhat - Project the state ahead
    angleX += dtX * (newRate - biasX);
    ```
// Update estimation error covariance - Project the error covariance ahead
PX_00 += -dtX * ( PX_10 + PX_01 ) + Q_angleX * dtX;
PX_01 += -dtX * PX_11;
PX_10 += -dtX * PX_11;
PX_11 += + Q_gyroX * dtX;

// Discrete Kalman filter measurement update equations - Measurement Update
("Correct")
// Calculate Kalman gain - Compute the Kalman gain
SX = PX_00 + R_angleX;
KX_0 = PX_00 / SX;
KX_1 = PX_10 / SX;

// Calculate angle and resting rate - Update estimate with measurement zk
yX = newAngle - angleX;
angleX += KX_0 * yX;
biasX += KX_1 * yX;

// Calculate estimation error co variance - Update the error co variance
PX_00 -= KX_0 * PX_00;
PX_01 -= KX_0 * PX_01;
PX_10 -= KX_1 * PX_00;
PX_11 -= KX_1 * PX_01;

return angleX;
}

KalmanY.pde

Listing B.8. KalmanY

/* Kalman filter variables and constants */
const double Q_angleY = 0.001; // Process noise covariance for the accelerometer - Sw
const double Q_gyroY = 0.003; // Process noise covariance for the gyro - Sw
const double R_angleY = 0.03; // Measurement noise covariance - Sv

double angleY = 180; // The angle output from the Kalman filter
double biasY = 0; // The gyro bias calculated by the Kalman filter
double PY_00 = 0, PY_01 = 0, PY_10 = 0, PY_11 = 0;
double dtY, yY, SY;
double KY_0, KY_1;

double kalmanY(double newAngle, double newRate, double dtime) {
  // KasBot V2 - Kalman filter module - http://www.arduino.cc/cgi-bin/yabb2/
  YaBB.pl?num=1284738418
  // See also http://www.z-firm.com/?page_id=145
  // with slightly modifications by Kristian Lauszus
  // See http://academic.csuohio.edu/simond/courses/ee644/kalman.pdf and
  // http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf for more
  // information
  // KasBot V2 - Kalman filter module - http://www.arduino.cc/cgi-bin/yabb2/
  YaBB.pl?num=1284738418
  // See also http://www.z-firm.com/?page_id=145
  // with slightly modifications by Kristian Lauszus
  // See http://academic.csuohio.edu/simond/courses/ee644/kalman.pdf and
  // http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf for more
  // information
dtY = dtime / 1000000;  // Convert from microseconds to seconds

// Discrete Kalman filter time update equations - Time Update ("Predict")
// Update xhat - Project the state ahead
angleY += dtY * (newRate - biasY);

// Update estimation error covariance - Project the error covariance ahead
PY_00 += -dtY * (PY_10 + PY_01) + Q_angleY * dtY;
PY_01 += -dtY * PY_11;
PY_10 += -dtY * PY_11;
PY_11 += +Q_gyroY * dtY;

// Discrete Kalman filter measurement update equations - Measurement Update
("Correct")
// Calculate Kalman gain - Compute the Kalman gain
SY = PY_00 + R_angleY;
KY_0 = PY_00 / SY;
KY_1 = PY_10 / SY;

// Calculate angle and resting rate - Update estimate with measurement zk
yY = newAngle - angleY;
angleY += KY_0 * yY;
biasY += KY_1 * yY;

// Calculate estimation error covariance - Update the error covariance
PY_00 -= KY_0 * PY_00;
PY_01 -= KY_0 * PY_01;
PY_10 -= KY_1 * PY_00;
PY_11 -= KY_1 * PY_01;

return angleY;

Magnetometer.pde

Listing B.9. Magnetometer

// Read magnetic field forces
// Calculate heading
static unsigned int getHeading()
{
    int heading;
    int x,y,z;
    float xr,yr, tanxyr;
    delay(100);  // There will be new values every 100ms
    HMC.getValues(&x,&y,&z);
    // Serial.print("z:");
    // Serial.print(x);
    // Serial.print(y);
    // Serial.print(" z:");
// Serial.println(z);
xr = x * DEG_TO_RAD;
yr = y * DEG_TO_RAD;
tanxyr = atan(xr/yr);

// Heading = tanxyr * RAD_TO_DEG;
if(y > 0)
    heading = 90 - tanxyr*180/PI;
else if ( y < 0)
    heading = 270 - tanxyr *180/PI;
else if (y == 0 && x<0)
    heading = 180;
else
    heading = 0;
heading = filter.run(heading-10);

if (heading<10)
    { 
        heading=360+(heading-10);
    }
else
    { 
        heading=heading-10;
    }
return (heading); //subtract declination angle respet to location -10 for PA

//Print the heading information on the screen for monitoring
static void printHeading()
{
    heading=filter.run(getHeading());

    Serial.print(" Magnitude =");
    Serial.print(heading-10);// subtract
    if (heading>337.5 && heading<=359 || heading>=0 && heading<=22.5)
        Serial.println(" N");
    if (heading>22.5 && heading<=67.5)
        Serial.println(" NE");
    if (heading>67.5 && heading<=112)
        Serial.println(" E");
    if (heading>112 && heading<=157.5)
        Serial.println(" SE");
    if (heading>157.5 && heading<=202.5)
        Serial.println(" S");
    if (heading>202.5 && heading<=247.5)
        Serial.println(" SW");
    if (heading>247.5 && heading<=292.5)
        Serial.println(" S");
    if (heading>292.5 && heading<=337.5)
        Serial.println(" NW");
// Initialize compass
static void initCompass()
{
    Serial.println("///////////////////////////////////");
    Serial.println("Initializing Compass Sensor...");
    HMC.init();
    delay(1000);
    Serial.println("Compass Sensor initialized.");
    Serial.println("///////////////////////////////////");
}

Motors.pde

Listing B.10. Motors

#include "defines.h"

//Motor pins initialization
void initMotor()
{
    Serial.println("///////////////////////////////////");
    Serial.println("Initializing Motor(s)...");
    for (int i = 0; i < MOTORS; i++)
    {
        motorPins[i+1]=motorPins[i]+1;
        motor[i].Pin(motorPins[i]);
        Serial.print("Motor "); Serial.print(i+1); Serial.print(" is attached to the ");
        Serial.println(motorPins[i]);
        motor[i].MinMax(min_val, max_val);
        // if (ArmOrBrake==1)
        // motor[i].SetArm();
        // else if (ArmOrBrake==2)
        // motor[i].SetBrake();
        motor[i].SetSpeed(0);
        delay(100);
    }
    Serial.println("Motors setup completed!");
    commandFlag=1;
    MotorSet=1;
    Serial.println("///////////////////////////////////");
}

//Main motor controller function. Waits for the throttle setting
void controlMotor(char ch)
{
    if (motorType == 0){
        if (commandFlag == 1 && MotorSet == 1)
```c
if (ch >= '0' && ch <= '9')
{
    value = (value * 10) + (ch - '0');
    ModeChange = 0;
}
else if (ch == '-')
{
    for (i = 0; i < MOTORS; i++)
    {
        value = motor[i].speed - 2;
    }
}
else if (ch == '+')
{
    for (i = 0; i < MOTORS; i++)
    {
        value = motor[i].speed + 2;
    }
}
else if (ch == '*')
{
    for (int i = 0; i < MOTORS; i++)
    {
        motor[i].SetSpeed(0);
        Serial.println("---------------- EMERGENCY STOP! ----------------");
    }
}
else if (ch >= 'a' && ch <= ('a' + MOTORS - 1))
{
    id = ch - 'a';
}
else if (ch == 10)
{
    if (id == -1)
    {
        for (i = 0; i < MOTORS; i++)
        {
            value = constrain(value, 0, 100);
            if (ModeChange == 0)
            {
                motor[i].speed = value;
                motor[i].SetSpeed(motor[i].speed);
            }
            Serial.print("Motor ");
            Serial.print(i);
            Serial.print(" Speed: ");
            Serial.print(motor[i].speed);
            Serial.println("%");
        }
    }
```
else
{
    value = constrain(value, 0, 100);
    motor[id].speed = value;
    if(ModeChange == 0)
    {
        motor[id].SetSpeed(motor[id].speed);
    }
    Serial.print("Motor ");
    Serial.print(id);
    Serial.print(" Speed: ");
    Serial.println(motor[id].speed);
}
value = 0;
id = -1;
}
}
Listing B.11. Orientation

static void SetDirection()
{
    if(flightMode == 'A')
    {
        // Define Variables we'll be connecting to
        double Setpoint, Input, Output;
        // Define the aggressive and conservative Tuning Parameters
        double aggKp = 2, aggKi = 0.05, aggKd = 0.01;
        double consKp = 4, consKi = 0.1, consKd = 0.02;
        // Specify the links and initial tuning parameters
        PID myPID(&Input, &Output, &Setpoint, consKp, consKi, consKd, DIRECT);

        unsigned long serialTime; // this will help us know when to talk with processing
        Input = filter.run(getHeading());
        double gap = (Setpoint - Input); // distance away from setpoint
        if(gap < 10)
        { // we're close to setpoint, use conservative tuning parameters
            myPID.SetTunings(consKp, consKi, consKd);
        }
    }
}
myPID.SetOutputLimits(10,90);
}
else{
    myPID.SetTunings(consKp, consKi, consKd);
    myPID.SetOutputLimits(90,170);
}

myPID.Compute();
servo[rudder].write(int(Output)); //control rudder servo
// delay(100);
}

Rotation.pde

Listing B.12. Rotation

double compAngleX = 180;
double compAngleY = 180;

void calculateRotation()
{
    double accXangle = getXangle();
    double accYangle = getYangle();

    compAngleX = (0.93*(compAngleX+(gyroXrate*(double)(micros()-timer)/1000000)))
        +(0.07*accXangle);
    compAngleY = (0.93*(compAngleY+(gyroYrate*(double)(micros()-timer)/1000000)))
        +(0.07*accYangle);

    double xAngle = kalmanX(accXangle, gyroXrate, (double)(micros() - timer)); //
        calculate the angle using a Kalman filter
    double yAngle = kalmanY(accYangle, gyroYrate, (double)(micros() - timer)); //
        calculate the angle using a Kalman filter

    Serial.print(gyroXangle); Serial.print("\t");
    Serial.print(gyroYangle); Serial.print("\t");

    Serial.print(accXangle); Serial.print("\t");
    Serial.print(accYangle); Serial.print("\t");

    Serial.print(compAngleX); Serial.print("\t");
    Serial.print(compAngleY); Serial.print("\t");
    //
    Serial.print(xAngle); Serial.print("\t");
    Serial.print(yAngle); Serial.print("\t");

    Serial.print("\n");
    delay(10);
SerialCom.pde

Listing B.13. SerialCom

```cpp
class SerialCom {
    char readSerial() {
        ReadInput = Serial.read();
        return ReadInput;
    }
}
```

Servo.pde

Listing B.14. Servo

```cpp
void initServo() {
    Serial.println("Servo setup started!");
    for (int i=0; i<SERVOS; i++)
    {
        servoPins[i+1]=servoPins[i]+1;
        servo[i].attach(servoPins[i]);
        Serial.print(" Servo"); Serial.print(i+1); Serial.print(" is attached to the ");
        Serial.println(servoPins[i]);
    }
    commandFlag=2; // Read input for servo
    ServoSet=1; // Servo is armed. Close setup warning.
    i=0;
    Serial.println("Servo Setup completed!");
    Serial.println("/////////////////////////////////");
}

static void controlServo(char ch) {
    if(commandFlag == 2 && ServoSet == 1)
    {
        if (ch >='0' && ch <= '9')
        {
            value = (value *10 )+ (ch -'0');
            ModeChange=0;
        }
        else if (ch == '-')
        {
            for (int i=0; i<SERVOS; i++)
            {
                value=servo[i].angle-2;
            }
        }
        else if (ch == '+')
        {
            for (int i=0; i<SERVOS; i++)
            {
                value=servo[i].angle+2;
            }
        }
    }
```
```cpp
{ 
    for (int i=0; i<SERVOS; i++)
    {
        value=servo[i].angle+2;
    }
}
else if (ch == '*')
{
    for (int i=0; i<SERVOS; i++)
    {
        servo[i].write(0);
        Serial.println("--------------- EMERGENCY STOP! ---------------");
    }
}
else if (ch>= 'a' && ch <=('a'+ SERVOS-1))
{
    id=ch-'a';
}
else if(ch == 10)
{
    if (id== -1)
    {
        for (i=0; i<SERVOS; i++)
        {
            value=constrain(value,0,180);
            if(ModeChange == 0)
            {
                servo[i].angle=value;
                servo[i].write(servo[i].angle);
            }
            Serial.print("Servo ");
            Serial.print(i);
            Serial.print(" Angle: ");
            Serial.println(servo[i].angle);
        }
    }
    else
    {
        value= constrain(value,0,180);
        if(ModeChange == 0)
        {
            servo[id].angle=value;
            servo[id].write(servo[id].angle);
        }
        Serial.print("Servo ");
        Serial.print(id);
        Serial.print(" Angle: ");
        Serial.println(servo[id].angle);
    }
}
value=0;
```
defines.h

Listing B.15. defines

```c
#define DEFINES_H
#endif

#define MOTORS 2 //define number of motor
#define MotorMin 20 //Motor min and max PWM signal range
#define MotorMax 150 //Motor min and max PWM signal range
#define MotorInitPin 8 //Motor initial pin number starts from this number.
#define motorType 1 //Motor type 0: brushed 1: brushless
#define ArmOrBrake 1 // enter 1 for normal Setup enter 2 for Brake function on/off
#define min_val 30 //Motor min and max PWM signal range
#define max_val 150

#define SERVOS 3 //define number of servo
#define ServoInitPin 5
enum {elevator, motorrotate, rudder};

#define BAUD 115200 //define serial communication baud rate

#define altitudeSensivity 5
```

```c
id=-1;
}
}
}
```
Bibliography

URL http://science.howstuffworks.com/transport/flight/modern/blimp.htm

URL http://repositorio.cti.gov.br/repositorio/handle/10691/179

[3] de Piolenc & George E. Wright, F. Ducted Fan Design, Volume 1 (Revised), Marc de Piolenc.
URL http://books.google.com/books?id=YcAjcSSP4HMC

URL http://ria.ua.pt/handle/10773/2515


URL http://ezinearticles.com/


URL http://www.lakehurst.navy.mil/nlweb/images/1213d.gif


g IMU for the Navigation of an Indoor Blimp,” .
URL http://etd.auburn.edu/etd/handle/10415/443


URL https://www.sparkfun.com/products/9801

URL www.ctsystems.eu/support/history-mag.html

URL https://www.sparkfun.com/products/9810


URL http://www.robotshop.com/content/images/airspeed-sensor-kit-large.jpg


URL https://www.sparkfun.com/products/8234


URL https://www.sparkfun.com/products/8742

URL https://www.sparkfun.com

URL http://www.hobbtronics.co.uk/accelerometer-gyro

URL http://en.wikipedia.org/wiki/Atan2


URL http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1696358

URL http://www.movable-type.co.uk/scripts/latlong.html

Vita

Ilker Oruc

Ilker Oruc was born on November 25th 1985 in Hatay, Turkey. He graduated with a Bachelor of Science in Aeronautical Engineering from Istanbul Technical University (ITU) in May 2010. He started to work in the first National Light Commercial Project of Turkey in Rotorcraft Center of Excellence at ITU, during the senior of his studies and continued to work in this project until August 2011. In 2011, he started his Master’s of Science degree in Pennsylvania State University and completed his Master’s on development of control system for uninhabited lighter-than-air vehicles in 2013 at Turbomachinery Aero-Heat Laboratory.