REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

JAVA BASED QUADCOPTER SIMULATION AND MODELLING

17011083 — Omer Faruk Yavuz

COMPUTER PROJECT

Advisor
Assist. Prof. Dr. Ferkan YILMAZ

January, 2020

ACKNOWLEDGEMENTS

This thesis is written as a computer project thesis and we have started from scratch. We chose to work with the quadcopters because in the future, these devices will be one of the key technologies for many industries and bring new jobs to new entrepreneurs. We want to give a special thanks to Dr. Ferkan Yilmaz for his guidance regarding this report and unceasing belief in our capabilities. We would also like to thank the department of computer engineering (Yildiz Technical University) for the opportunity to conduct this project.

Omer Faruk Yavuz

TABLE OF CONTENTS

LIST OF SYMBOLS

b	Refered to BODY frame
n	Refered to NED frame
ϕ	Euler angle roll
θ	Euler angle pitch
ψ	Euler angle yaw
p	Euler angle roll rate
q	Euler angle pitch rate
r	Euler angle yaw rate

LIST OF ABBREVIATIONS

İHA	Insansız Hava Aracı
NED	North East Down
DOF	Degrees of Freedom
CAD	Computer Aided Design
IMU	Inertial Measurement Unit
ESC	Electro Speed Controlle
BCS	Body Coordinate System
ECEF	Earth Centered Earth Fixed
RPM	Revolution Per Minute

LIST OF FIGURES

ÖZET

JAVA TABANLI QUADCOPTER SİMULASYONU VE MODELLEMESİ

Omer Faruk Yavuz
Bilgisayar Mühendisliği Bölümü
Bilgisayar Projesi

Danışman: Dr. Öğretim Üyesi Ferkan YILMAZ

On yıldan daha uzun bir süre boyunca, quadcopterlerin popülerligi son derece artmıstır. Günümüzde, egitimden is sektörüne kadar birçok kisi bu quadcopterleri kendi amacı dogrultusunda kullanmaktadır. Bu cihazlar, herhangi biri tarafından uygun fiyatla satın alınabildigi gibi hükümetler tarafından askeri amaçlı olarak da kullanılabilir. Yayımlanan bu tez, bir quadcopterin bünyesinde barındırdıgı kinematik denklemleri, bir quadcopterin hareket simülasyonunu, Arduino aracılıgı ile sensör verilerinin okunmasını ve bu verilerin APC-220 Wi-Fi modülü ile Arduino COM-port ekranından okunmasını amaçlar

Anahtar Kelimeler: Quadcopter, simülasyon, bilgisayar destekli tasarım, Arduino, sensör

ABSTRACT

JAVA BASED QUADCOPTER SIMULATION AND MODELLING

Omer Faruk Yavuz
Department of Computer Engineering
° Project

Advisor: Assist. Prof. Dr. Ferkan YILMAZ

For more than a decade, the popularity of the quadcopters is increasing. From the purpose of the education to business, everyone can use these quadcopters for their own purpose. These mini-UAV's can be bought with cheap prices as well as they can be bought by the governments for military purposes. Our thesis aims to show the equations of motion of a quadcopter's, also the simulation of it's movement, sensor readings with Arduino and sending these sensor datas to our laptop via APC-220 Wi-Fi module.

Keywords: quadcopter, simulation, CAD, Arduino, sensor

1
Introduction

Today, the technologies that we use in our daily life comes from the adaptation of military needings to civil industry. The internet, UAV's, GPS, Radar etc. can be given as an example of that adaptation. Israel and US Military Forces are the first forces that understands the importance of UAV's [1]. With the research and development that has been made in this industry, those vehicles have become smaller, and the technologies that incorporate (sensors etc.) is improved in that time period. Smaller sizes of those UAV's are used both in military and civil industry [2]. This project aims to understand the structure of a quadcopter. Quadcopters have four motors and they have a unique design. With the changing of the current that goes to these motors, a quadcopter can move to different directions and it contains 6-DOF. The exploration of this technology is making charming with the mathematical and kinematic equations, which it contains the basis of aerospace knowledge, with it's low-cost equipment and small sizes.

2
Literature Review

The project that we have done aims to simulate a quadcopter's motion. During our research, we have looked at the master thesis based on the quadcopter modellings. Andrew Gibiansky's MATLAB based quadcopter simulation can be given as an example. Our simulation is written in Java, not a popular platform for simulation design comparing to Unity. Lack of usage of Java language as a simulation platform is the main motivation of this project. The correctness of the equations is made with MATLAB language. In our simulation, we have used air density formula which is not used before in any similar to this thesis and modelling. In our research, we have found that in most of the papers related to quadcopter modelling, authors take the air density as a constant value. However, air density changes in places where the quadcopter hovers. So, we have implemented the air density formula in our simulation.

As mentioned in the introduction section, quadcopters are becoming a challenging technology in the world, thus understanding this technology would even bring some revolution to some industries. Understanding the motion capability and theoretical study would be helpful for engineers.

Andrew Gibinasky, an Russian engineer which also influences this thesis and other master thesis in the world designs a powerful modelling. Joakim Brobakk Lehn also brings us very good explanation. A similar thesis is written by Cristina Pupaza, which she specifically explains the equations detailly. We have also referenced to her.

We have used Wi-Fi modules to send and receive sensor datas. This thesis will be a good starting for the enthusiasts who will make this project go for the further.

3
Feasibility

This project has been successfully accomplished. The programming language we used is one of the widely used language in the world, Java. For the students who might develop this project for the further in the future or code another simulation, Java programming language would be a good choice.

- **Technical Feasibility:**

  The radio communication module (APC-220) we used is widely used in the industry, thus using this module and sending sensor datas through the computer via these modules is simple enough. It includes an embedded high speed microprocessor and high performance IC that creates a transparent UART/TTL interface, and eliminates any need for packetizing and data encoding. It transmits data with a line of sight 1000m with 9600 bps. It has 256 bytes data buffer and has a GFSK modulation. Due to the low in price, we have chosen to use it.

  Arduino Uno is an open-source microcontroller based on Microchip ATmega328P. Uno is a microcontroller that is used in many different robotics projects, so any tutorials and documentations on any project can be found easily on the internet. They are also cheap in price. In order to read sensor values, we used this microcontollers.

  MATLAB is a multi-paradigm numerical computing environment and proprietary programming language developed by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages. MATLAB is a powerful mathematical language which enabled us to validate our mathematical equations results from our Java based simulation.

  Open Graphics Library is a cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics. The API

is typically used to interact with a graphics processing unit, to achieve hardware-accelerated rendering.

The Lightweight Java Game Library is an open-source Java software library for video game developers. It exposes high performance cross-platform libraries commonly used in developing video games and multimedia titles, such as Vulkan, OpenGL, OpenAL and OpenCL.

In our research, we have found that using OpenGL library and LWJGL will become important for visualization. An open source 3D graphic design program, Blender, will also be used to draw quadcopter in 3D for modelling for our simulation.

Java is chosen for this project. Java is one of the mostly used programming language in the world. When we look at the similar to the quadcopter simulations, we have seen that none of them use Java as their programming language. We have chosen Java, however we have seen during this process that using C and Unity would be easier for visualization and using MATLAB would be better for calculating mathematical equations. Due to the uniqeness of this project, we have chosen to use Java language.

In order to run this simulation, the user needs at least 8GB RAM, Intel HD Graphics 620 and Windows 10 operating system.

- **Workforce and Time Planning:**

  The workforce of this project will be one per month. The academic study of an unmanned vehicle is difficult, thus we have been working for this project for nearly two years. In order to code a simulation, we needed to know computer graphics, we have studied computer graphics theoretically. We have used LWJGL and OpenGL for simulation. In order to use them, we need to do some practice before implementing them. We have spend one year for learning and implementing it. We have spend one week to learn and practice. Author of this project had experience on using Arduino and sensors.

  The timesheet of this project is given in the System Analysis section.

- **Legal Feasibility:**

  This simulation is coded by the author of this thesis. We have used all open-source softwares (as mentioned in the section technical feasibility) and implemented all the codes by ourselves. This project does not violate any rules under the Turkish Republic laws and does not violate any patent rights. Thus this simulation can be used freely by anyone who owns personal computers that require at least these spesifications which mentioned above.

- **Economic Feasibility:**

  Total expenses for modules: APC220: 116TL, Arduino Microcontroller: 20TL, Wiring: 15TL, HMC-5883L: 26TL, MPU-6050: 8TL, HC-SR04: 5TL.

  This project does not violate any rules and does not need external expenses due to the legal feasibility.

  Using a simulation for any device reduces the cost of using it comparing to using a device without simulation. An example can be given for pilots flying a plane. The same example can be given for quadcopters also. Using this simulation will reduce the amount of money that people spend on using quadcopters, because these devices are sensitive. Using them for hobby might not be so important but using them in business industry, it is important.

4
System Analysis

This project targets the explaination of a quadcopter movement with mathematical equations. In the future, we expect to do quadcopter localization, for that we needed to get the sensor readings. With the sensor readings, we also expected to some filterings in the future. We haven't implemented PID controller however a theoretical explanation is made in this thesis.

In order to study the quadcopter's movement, the enthusiast has to read lots of articles, some of them are referenced already in this article. Reading theses was an improving solution for us. Before we start the project, Assoc. Prof. Dr. Ferkan Yilmaz has seen the vacancy of unused language, Java, for simulation developments. Due to it has a powerful optimization, we chose the Java language. So that we implemented the codes via Java language.

We expected to design a GUI for output values of mathematical equations, however we have seen that using keyboard would be a better option. The output values of the equations can be seen in the console screen.

Studying	Dates	Day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Quadcopter Theory Studying	16.01.2018 - 16.12.2019	700																							
Simülasyon Prestudy	16.01.2018 - 16.01.2019	365																							
Simülasyon Study	16.09.2019 - 16.12.2019	90																							
Blender Design	08.11.2019 - 16.12.2019	3																							
Wireless Module Implementation	12.11.2019 - 16.12.2019	30																							
Sensors and Arduino Implementation	12.11.2019 - 16.12.2019	30																							

Figure 4.1 Timesheet of the project

5
System Design

5.0.0.1 Input-Output Design

Our quadcopter simulation takes of its input's with the keyboard buttons. When the user pushes "Up" (arrow) button, the quadcopter lifts and gains "thrust" force. When the user stops pushing down the "Up" button, quadcopter hovers, not looses of its altitude's. When the user pushes "Down" button, quadcopter looses of its altitude, thus lands. When the user presses "A" button, quadcopter moves to the left, when pressed "D", quadcopter moves to the right, when pressed "W", the quadcopter moves away from the user, when pressed "S", the quadcopter comes towards to the user. Finally, when the user pressed "R" and "V", the quadcopter rolls, when pressed "P" and "H" it gains pitch movement, when pressed "Y", the quadcopter rotates around itself.

The quadcopter owns a predefined angular velocity. The calculation of total thrust is done for every second. For each rotor, simulation also calculates their thrust forces. When the quadcopter changes of its own direction, the angular velocities for each rotor are changed proportionally. The position of the quadcopter in X, Y and Z axes, torques, momentums, air density, angular accelerations and angular velocities in the body frame, drag forces in each axes, roll, pitch and yaw angles are calculated for every second.

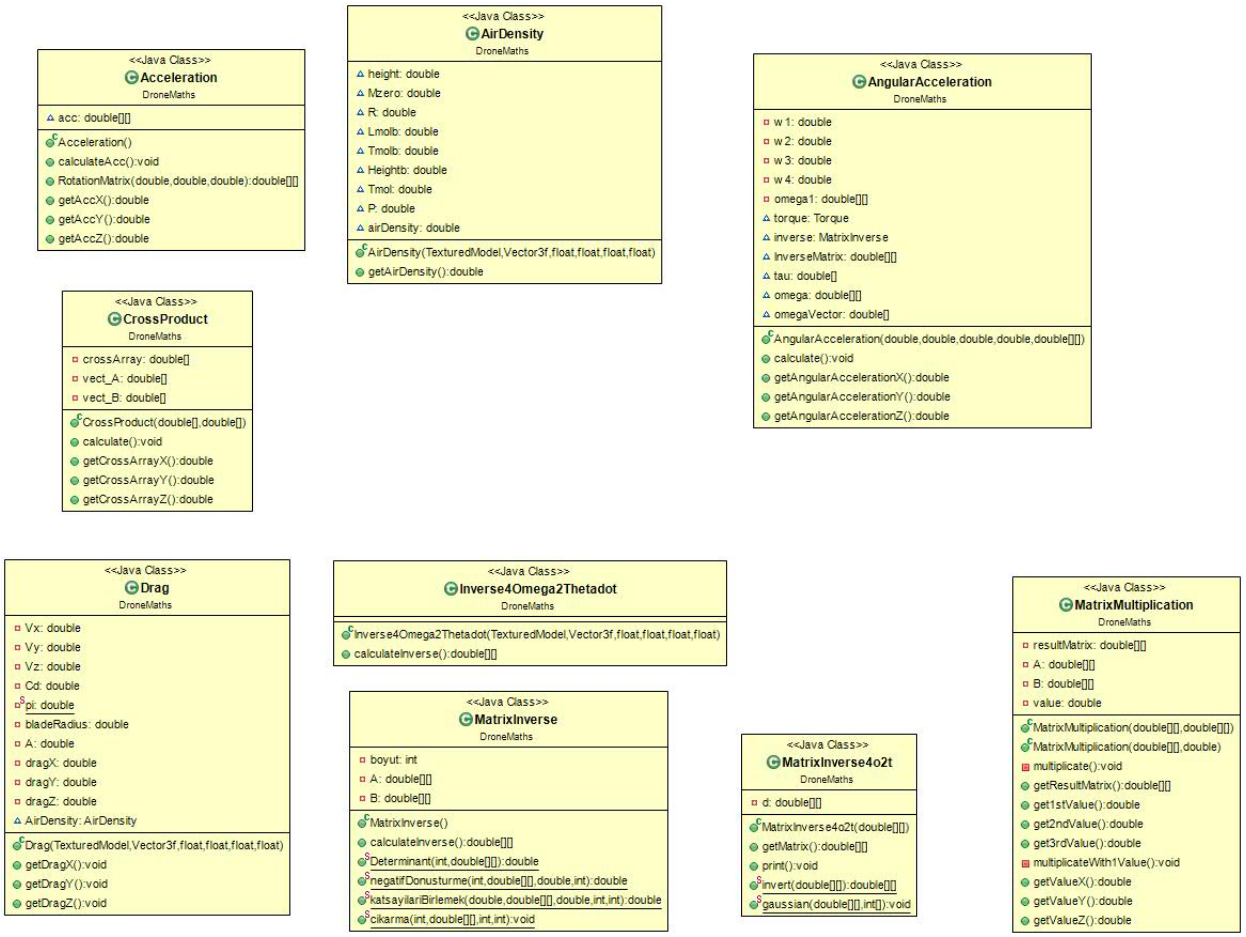
The correctness of the results is made with MATLAB language. For simplification, we have taken some formulas as constant or made some simplifications on some formulas, which all are described in the Physics section.

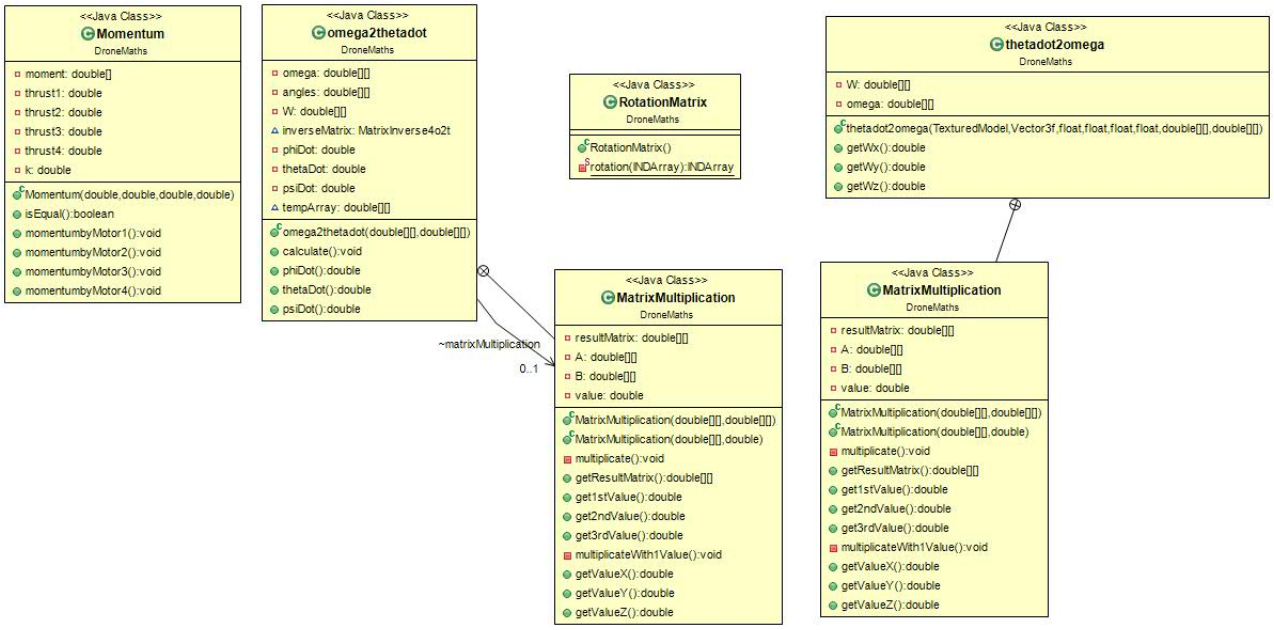
5.0.0.2 Software Design

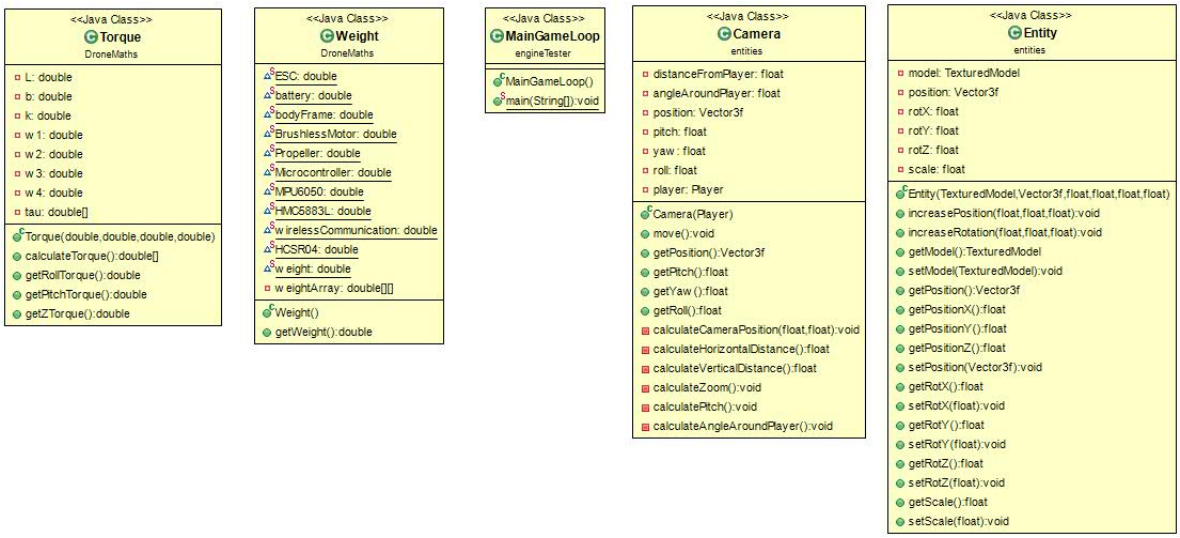
The simulation is written with Java programming language. Coding via Java programming language is the primary way to produce code that will be deployed as Java bytecode, though there are compilers available for other languages such as JavaScript, Python and Ruby, and a native Java scripting language called Groovy. Java

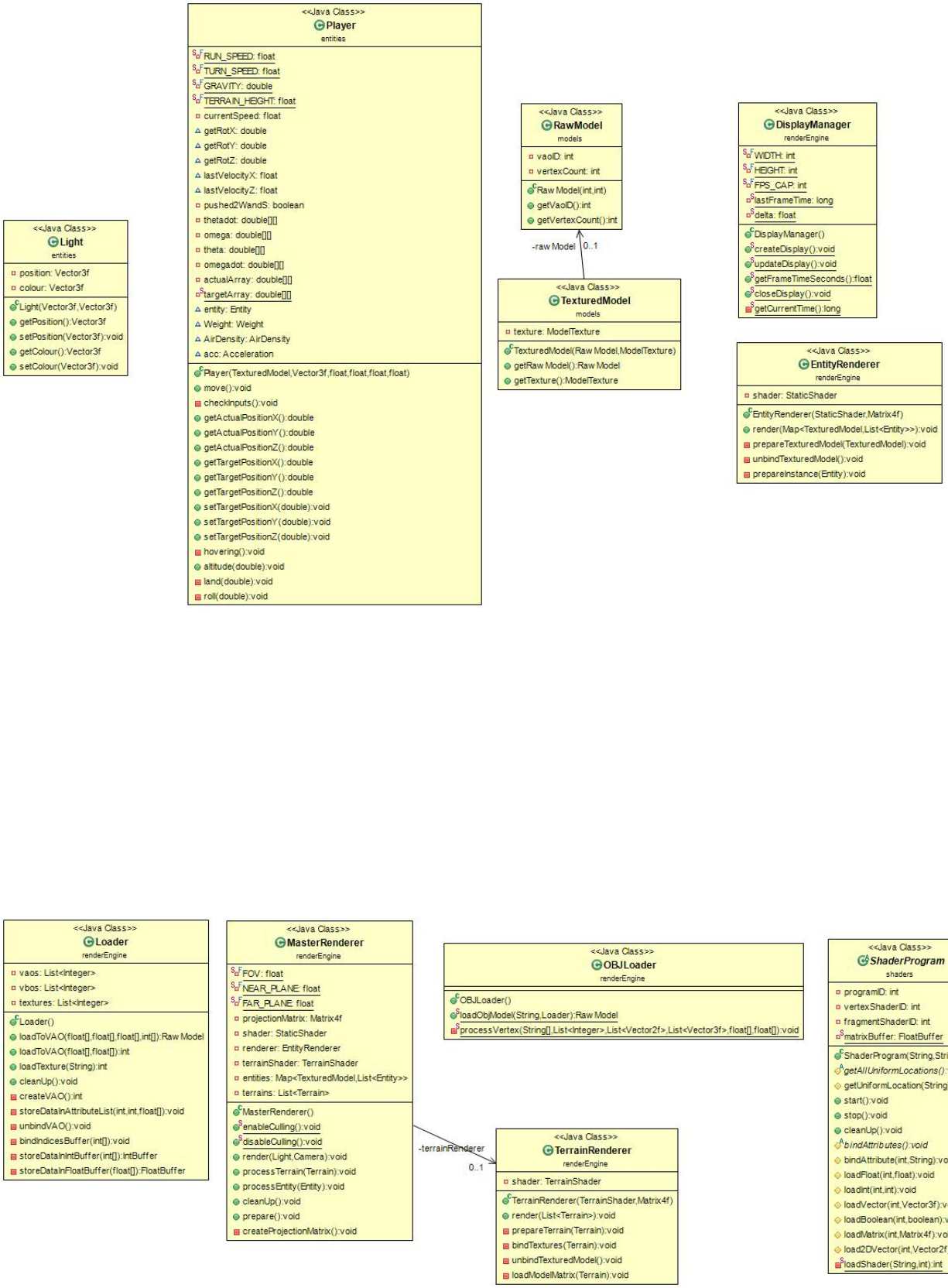
syntax borrows heavily from C and C++ but it eliminates certain low-level constructs such as pointers and has a very simple memory model where every object is allocated on the heap and all variables of object types are references. Memory management is handled through integrated automatic garbage collection performed by the Java Virtual Machine (JVM).

The UML schemas of the simulation are given below




<<Java Class>>
Momentum
DroneMaths
moment: double[]
thrust1: double
thrust2: double
thrust3: double
thrust4: double
k: double
Momentum(double,double,double,double)
isEqual():boolean
momentumbyMotor1():void
momentumbyMotor2():void
momentumbyMotor3():void
momentumbyMotor4():void
<<Java Class>>
omega2thetadot
DroneMaths
omega: double[][]
angles: double[][]
W: double[][]
inverseMatrix: MatrixInverse4o2t
phiDot: double
thetaDot: double
psiDot: double
tempArray: double[][]
omega2thetadot(double[][],double[][])
calculate():void
phiDot():double
thetaDot():double
psiDot():double
~matrixMultiplication
0..1
<<Java Class>>
RotationMatrix
DroneMaths
RotationMatrix()
rotation(INDArray):INDArray
<<Java Class>>
MatrixMultiplication
DroneMaths
resultMatrix: double[][]
A: double[][]
B: double[][]
value: double
MatrixMultiplication(double[][],double[][])
MatrixMultiplication(double[][],double)
multiplicate():void
getResultMatrix():double[][]
get1stValue():double
get2ndValue():double
get3rdValue():double
multiplicateWith1Value():void
getValueX():double
getValueY():double
getValueZ():double
<<Java Class>>
thetadot2omega
DroneMaths
W: double[][]
omega: double[][]
thetadot2omega(TexturedModel,Vector3f,float,float,float,float,double[][],double[][])
getWx():double
getWy():double
getWz():double
<<Java Class>>
MatrixMultiplication
DroneMaths
resultMatrix: double[][]
A: double[][]
B: double[][]
value: double
MatrixMultiplication(double[][],double[][])
MatrixMultiplication(double[][],double)
multiplicate():void
getResultMatrix():double[][]
get1stValue():double
get2ndValue():double
get3rdValue():double
multiplicateWith1Value():void
getValueX():double
getValueY():double
getValueZ():double


<<Java Class>>
Torque
DroneMaths
L: double
b: double
k: double
w 1: double
w 2: double
w 3: double
w 4: double
tau: double[]
Torque(double,double,double,double)
calculateTorque():double[]
getRollTorque():double
getPitchTorque():double
getZTorque():double
<<Java Class>>
Weight
DroneMaths
ESC: double
battery: double
bodyFrame: double
BrushlessMotor: double
Propeller: double
Microcontroller: double
MPU6050: double
HMC5883L: double
wirelessCommunication: double
HCSR04: double
weight: double
weightArray: double[][]
Weight()
getWeight():double
<<Java Class>>
MainGameLoop
engineTester
MainGameLoop()
main(String[]):void
<<Java Class>>
Camera
entities
distanceFromPlayer: float
angleAroundPlayer: float
position: Vector3f
pitch: float
yaw : float
roll: float
player: Player
Camera(Player)
move():void
getPosition():Vector3f
getPitch():float
getYaw ():float
getRoll():float
calculateCameraPosition(float,float):void
calculateHorizontalDistance():float
calculateVerticalDistance():float
calculateZoom():void
calculatePitch():void
calculateAngleAroundPlayer():void
<<Java Class>>
Entity
entities
model: TexturedModel
position: Vector3f
rotX: float
rotY: float
rotZ: float
scale: float
Entity(TexturedModel,Vector3f,float,float,float,float)
increasePosition(float,float,float):void
increaseRotation(float,float,float):void
getModel():TexturedModel
setModel(TexturedModel):void
getPosition():Vector3f
getPositionX():float
getPositionY():float
getPositionZ():float
setPosition(Vector3f):void
getRotX():float
setRotX(float):void
getRotY():float
setRotY(float):void
getRotZ():float
setRotZ(float):void
getScale():float
setScale(float):void


«Java Class»
Player
entities
RUN_SPEED: float
TURN_SPEED: float
GRAVITY: double
TERRAIN_HEIGHT: float
currentSpeed: float
getRotX: double
getRotY: double
getRotZ: double
lastVelocityX: float
lastVelocityZ: float
pushed2WandS: boolean
thetadot: double[][]
omega: double[][]
theta: double[][]
omegadot: double[][]
actualArray: double[][]
targetArray: double[][]
entity: Entity
Weight: Weight
AirDensity: AirDensity
acc: Acceleration
Player(TexturedModel,Vector3f,float,float,float,float)
move():void
checkInputs():void
getActualPositionX():double
getActualPositionY():double
getActualPositionZ():double
getTargetPositionX():double
getTargetPositionY():double
getTargetPositionZ():double
setTargetPositionX(double):void
setTargetPositionY(double):void
setTargetPositionZ(double):void
hovering():void
altitude(double):void
land(double):void
roll(double):void
«Java Class»
Light
entities
position: Vector3f
colour: Vector3f
Light(Vector3f,Vector3f)
getPosition():Vector3f
setPosition(Vector3f):void
getColour():Vector3f
setColour(Vector3f):void
«Java Class»
RawModel
models
vaoID: int
vertexCount: int
Raw Model(int,int)
getVaoID():int
getVertexCount():int
-raw Model
0..1
«Java Class»
TexturedModel
models
texture: ModelTexture
TexturedModel(Raw Model,ModelTexture)
getRaw Model():Raw Model
getTexture():ModelTexture
«Java Class»
DisplayManager
renderEngine
WIDTH: int
HEIGHT: int
FPS_CAP: int
lastFrameTime: long
delta: float
DisplayManager()
createDisplay():void
updateDisplay():void
getFrameTimeSeconds():float
closeDisplay():void
getCurrentTime():long
«Java Class»
EntityRenderer
renderEngine
shader: StaticShader
EntityRenderer(StaticShader,Matrix4f)
render(Map<TexturedModel,List<Entity>>):void
prepareTexturedModel(TexturedModel):void
unbindTexturedModel():void
prepareInstance(Entity):void
«Java Class»
Loader
renderEngine
vaos: List<Integer>
vbos: List<Integer>
textures: List<Integer>
Loader()
loadToVAO(float[],float[],float[],int[]):Raw Model
loadToVAO(float[],float[]):int
loadTexture(String):int
cleanUp():void
createVAO():int
storeDataInAttributeList(int,int,float[]):void
unbindVAO():void
bindIndicesBuffer(int[]):void
storeDataInIntBuffer(int[]):IntBuffer
storeDataInFloatBuffer(float[]):FloatBuffer
«Java Class»
MasterRenderer
renderEngine
FOV: float
NEAR_PLANE: float
FAR_PLANE: float
projectionMatrix: Matrix4f
shader: StaticShader
renderer: EntityRenderer
terrainShader: TerrainShader
entities: Map<TexturedModel,List<Entity>>
terrains: List<Terrain>
MasterRenderer()
enableCulling():void
disableCulling():void
render(Light,Camera):void
processTerrain(Terrain):void
processEntity(Entity):void
cleanUp():void
prepare():void
createProjectionMatrix():void
-terrainRenderer
0..1
«Java Class»
TerrainRenderer
renderEngine
shader: TerrainShader
TerrainRenderer(TerrainShader,Matrix4f)
render(List<Terrain>):void
prepareTerrain(Terrain):void
bindTextures(Terrain):void
unbindTexturedModel():void
loadModelMatrix(Terrain):void
«Java Class»
OBJLoader
renderEngine
OBJLoader()
loadObjModel(String,Loader):Raw Model
processVertex(String[],List<Integer>,List<Vector2f>,List<Vector3f>,float[],float[]):void
«Java Class»
ShaderProgram
shaders
programID: int
vertexShaderID: int
fragmentShaderID: int
matrixBuffer: FloatBuffer
ShaderProgram(String,String)
getAllUniformLocations():void
getUniformLocation(String):int
start():void
stop():void
cleanUp():void
bindAttributes():void
bindAttribute(int,String):void
loadFloat(int,float):void
loadInt(int,int):void
loadVector(int,Vector3f):void
loadBoolean(int,boolean):void
loadMatrix(int,Matrix4f):void
load2DVector(int,Vector2f):void
loadShader(String,int):int

6
Implementation

6.0.1 Coordinate Systems for Navigation

Coordinate systems can be used in many areas and we will use them in our flight dynamics modelling. Geodetic coordinate system is a system that characterizes a coordinate point near the Earth's surface in terms of longitude, latitude and height. ECI coordinate system is a mixed inertial system and it is oriented with respect to sun. Its origin is fixed at the center of earth. ECEF coordinate system rotates with the Earth around its spin axis. Its origin is fixed at the center of the Earth. NED coordinate system (inertial frame) is a coordinate frame fixed to the earth's surface and it can be used by small-scale UAV rotorcraft. The x-axis points towards true North, the y-axis towards East, while the z-axis points downwards normal to the Earth's surface. BODY coordinate system is vehicle-carried and is directly defined on the body of the flying vehicle. Due to frequent using in aerospace engineering, we will use the ZYZ Euler angles(notation). The origin of the NED coordinate system is usually defined as the starting point on the ground where the quadcopter takes off. The position of the quadcopter is defined as the position of the BODY origin with respect to the NED origin.

At this point, we will make two working hypotheses. First the curvature of the Earth will be neglected. This is a valid hypothesis when the airborne vehicle operates over a small area. The second notable assumption is that the atmosphere (the air mass over the Earth) is not moving in relation to the surface.

6.0.2 The Rotation Matrix

A rotation matrix R is any matrix that

$$RR^T = R^T R = I, det(R) = 1 \tag{6.1}$$

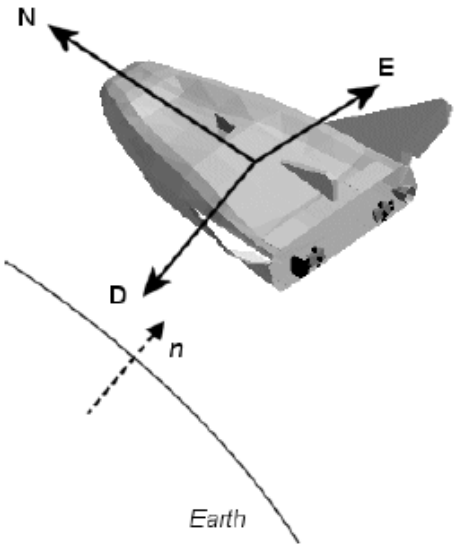


Figure 6.1 NED coordinate system on a plane.

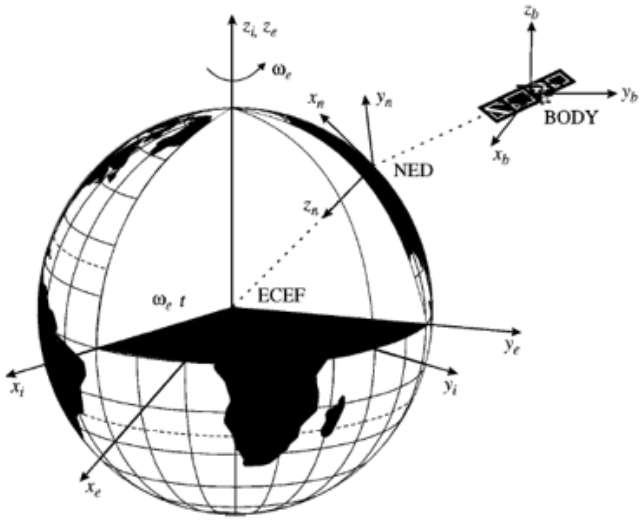


Figure 6.2 The ECEF is rotating with angular rate e with respect to an ECI fixed in space.

This implies that R is orthogonal and as a consequence the inverse rotation matrix is given by

$$R^{-1} = R^T \tag{6.2}$$

Hence

$$R_b^a = (R_a^b)^{-1} = (R_a^b)^T \tag{6.3}$$

6.0.3 Rotation Between BODY and NED

To transform between the BODY and NED coordinate systems, we need the rotation matrix (R_n^b).

In aerospace engineering the convention most widely used is yaw, pitch, roll (zyz-convention), noted ψ; θ; ϕ respectively. This implies that the first rotation corresponds to a yaw rotation, the second a pitch rotation and the last a roll rotation. Each independent rotation can be described by the following rotation matrices

$$R_{z,\psi} = \begin{pmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_{y,\theta} = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix}$$

$$R_{x,\phi} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{pmatrix}$$

To derive the rotation matrix from BODY frame to NED frame, we need to multiply all those three matrices. Thus, it gives us the equation[3]

$$R_b^n(\theta_{nb}) = R_{z,\psi} R_{y,\theta} R_{x,\phi}$$

$$\mathrm{R}_b^n(\theta_{nb}) = \begin{pmatrix} c(\psi)c(\theta) & -s(\psi)c(\phi)+c(\psi)s(\theta)s(\phi) & s(\psi)s(\phi)+c(\psi)c(\phi)s(\theta) \\ s(\psi)c(\theta) & c(\psi)c(\phi)+s(\phi)s(\theta)s(\psi) & -c(\psi)s(\phi)+s(\theta)s(\psi)c(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{pmatrix}$$

where c stands for cosine and s for sine.

In the following, we will also use the transformation from NED frame to BODY frame. To transform in the opposite direction above, we had the equation $R_n^b = (R_b^n)^T$ which gives

$$\mathrm{R}_n^b(\theta_{nb}) = \begin{pmatrix} c(\psi)c(\theta) & s(\psi)c(\theta) & -s(\theta) \\ -s(\psi)c(\phi)+c(\psi)s(\theta)s(\phi) & c(\psi)c(\theta)+s(\phi)s(\theta)s(\psi) & c(\theta)s(\phi) \\ s(\psi)s(\phi)+c(\psi)c(\phi)s(\theta) & -c(\psi)s(\phi)+s(\theta)s(\psi)s(\phi) & c(\theta)c(\phi)) \end{pmatrix}$$

6.0.4 Transformation Between Geodetic and NED

NED, ECEF and Geodetic coordinates are all related to each other. In order to work on NED coordinate system, we have to transform from Geodetic coordinate system to ECEF coordinate system. A geodetic position given as

$$p_g = [\phi, \lambda, h]^T \tag{6.4}$$

the ECEF coordinates are given by

$$p^e = \begin{pmatrix} (N+h)cos(\lambda)cos(\phi) \\ (N+h)cos(\lambda)sin(\phi) \\ (\frac{r_p^2}{r_e^2}N+h)sin(\lambda) \end{pmatrix} \tag{6.5}$$

λ is the geodetic latitude, ϕ is the longitude, h is the ellipsoidal height and N is the radius of curvature in the prime vertical. N is calculated by

$$N = \frac{r_e^2}{\sqrt{r_e^2 cos^2(\lambda) + r_p^2 sin^2(\lambda)}} \tag{6.6}$$

where the equatorial and polar earth radius, r_e and r_p, are the semi-axes of the ellipsoid with $r_e = 6378.137$km and $r_p = 6356.752$km.

Thus, we can transform into NED coordinates through the rotation matrix

$$R_e^n(\phi, \lambda) = \begin{pmatrix} -sin(\lambda)cos(\phi) & -sin(\lambda)sin(\phi) & cos(\lambda) \\ -sin(\phi) & cos(\phi) & 0 \\ -cos(\lambda)cos(\phi) & -cos(\lambda)sin(\phi) & -sin(\lambda) \end{pmatrix}$$

The position in NED coordinates is then given by

$$\hat{p}^n = R_e^n(\phi, \lambda)(\hat{p}^e - \hat{p}_{ref}^e) \tag{6.7}$$

where p_{ref}^e is the ECEF coordinates of the NED origin.

6.0.5 Physics

The quadcopter is navigating in a three dimensional space. The BODY coordinate system is a moving coordinate frame fixed to the quadcopter. The quadcopter structure is represented in Figure 1, where it shows the angular velocities, forces and torques on the system. As seen below, the sum of angular velocities with respect to their directions $(\omega_1, \omega_2, \omega_3, \omega_4)$ are equal to zero.

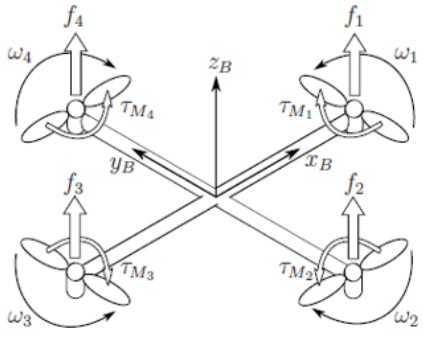


Figure 6.3 Directions of blades on a quadcopter.

In order to design a quadcopter, we need to have an understanding of its physical properties. This theses does not aim to design quadcopter mechanically however

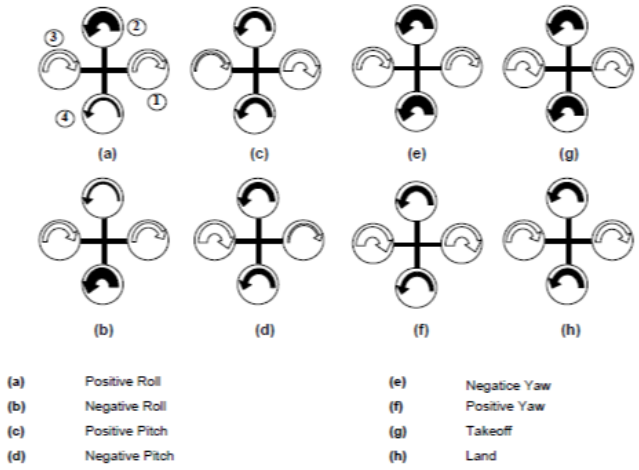


Figure 6.4 Rotation of a quadcopter.

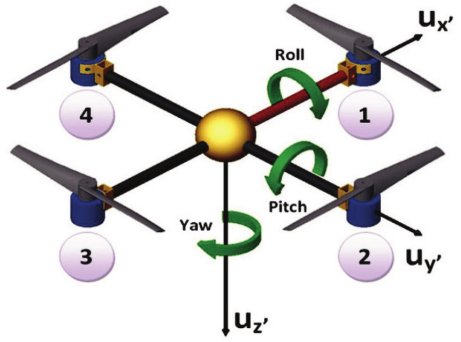


Figure 6.5 Roll, pitch and yaw angles.

a general understanding of DC motors and the forces generated by them would be helpful.

6.0.6 Motor Dynamics

It is important to know how DC motors can accelerate the quadcopter. DC motors consist of one set of coils, called armature winding, inside another set of coils or a set of permanent magnets, called the stator. The rotator in DC motors rotates because the current flowing through a coil produces a magnetic field repelled by permanent magnets mounted in the armature.

Stator is the stationary outside part of a motor. The stator of a permanent magnet DC motor is composed of two or more permanent magnet pole pieces. Rotor is the inner part which rotates. The rotor is composed of windings (called armature windings) which are connected to the external circuit through a mechanical commutator. Both stator and rotor are made of ferromagnetic materials. The two are separated by air-gap. A winding is made up of series or parallel connection of coils. Armature

winding is the winding through which the voltage is applied or induced. Field winding is the winding through which a current is passed to produce flux (for the electromagnet). Windings are usually made of copper. If electrical energy is supplied to a conductor lying perpendicular to a magnetic field, the interaction of current flowing in the conductor and the magnetic field will produce mechanical force.

There are two conditions which are necessary to produce a force on the conductor. The conductor must be carrying current, and must be within a magnetic field. When these two conditions exist, a force will be applied to the conductor, which will attempt to move the conductor in a direction perpendicular to the magnetic field. This is the basic theory by which all DC motors operate.

The force exerted upon the conductor can be expressed as follows.

$$F = B \times I \times L \tag{6.8}$$

where B is the density of the magnetic field, L is the length of conductor, and I the value of current flowing in the conductor. The direction of motion can be found using Fleming's Left Hand Rule.

Consider a coil in a magnetic field of flux density B. When the two ends of the coil are connected across a DC voltage source, current I flows through it. A force is exerted on the coil as a result of the interaction of magnetic field and electric current. The force on the two sides of the coil is such that the coil starts to move in the direction of force.

In an actual DC motor, several such coils are wound on the rotor, all of which experience force, resulting in rotation. The greater the current in the wire, or the greater the magnetic field, the faster the wire moves because of the greater force created.

When the motor is rotating, the voltage V applied to the motor is diminished by the voltage E produced by the rotator[4]. When V is increased also the E is increased. We can formulate voltage E

$$E = k_e \omega \tag{6.9}$$

where k_e is constant, and ω is the angular velocity of the motor. When the motor

reaches its maximum operating speed E will be generated at a constant rate. When E comes close to V, the motor is rotating fast but the current flowing through the coil is lower.

When eventually $V–E = 0$, the current flowing through the coil drops to zero and there is no torque acting on the motor. The DC motor has reached its final velocity.

All motors are assumed identical in our modelling for simplicity. In quadcopter dynamics, we use brushless motors. The torque on the rotator is proportional to the current in the coil

$$\tau = k_t I \tag{6.10}$$

where k_t is constant. Since $I = \frac{(V-E)}{R}$, where R is the resistance of the coil, we obtain

$$\tau = \frac{k_t}{R}(V - E) = \frac{k_t}{R}(V - k_e\omega) \tag{6.11}$$

The torque also can be represented as

$$\tau = k_t(I - I_0) \tag{6.12}$$

The voltage across the motor is the sum of the back electromotive force (voltage, or electromotive force, that pushes against the current which induces it) and some resistive loss

$$V = IR_m + k_v\omega \tag{6.13}$$

where V is the voltage drop across the motor, R_m is the motor resistance, ω is the angular velocity of the motor, and k_v is a proportionality constant (indicating the back electromotive force generated per RPM). We can use this description of our motor to calculate the power it consumes

$$P = IV = \frac{(\tau + k_t I_o)(k_t I_o R_m + \tau R_m + k_t k_v \omega)}{k_t^2} \tag{6.14}$$

6.0.7 Thrust Force

The power generated is used to keep the quadcopter aloft. By conservation of energy, we know that the energy the motor produces in a given time period is equal to the force generated on the propeller times the distance that the air it displaces moves ($Pdt = Tdx$). Equivalently the power is equal to the thrust times the air velocity ($P = T\frac{d_x}{d_t}$).

$$P = Tv_h \tag{6.15}$$

We assume that the free stream velocity (v_∞) is zero (the air in the surrounding environment is stationary relative to the quadcopter). Momentum theory gives us the equation for hover velocity as a function of thrust[5]

$$v_h = \sqrt{\frac{T}{2\rho A}} \tag{6.16}$$

where ρ is the density of the surrounding air and A is the area swept out by the rotor. Using the simplified equation for power, we can then write

$$P = \frac{K_v}{K_t}\tau\omega = \frac{K_v K_t}{K_t}T\omega = \frac{T^{3/2}}{\sqrt{2\rho A}} \tag{6.17}$$

In the general case, we got that τ, $\tau = \overrightarrow{r} \times \overrightarrow{F}$ but in this case the torque is proportional to the thrust T by some constant radio k_t determined by the blade configuration parameters. Solving the thrust magnitude T

$$T = (\frac{K_v K_t \sqrt{2\rho A}}{K_t}\omega)^2 \tag{6.18}$$

We find that the total thrust of the quadcopter (in the BODY frame) is given by

$$T^b = \sum_{i=1}^{4} T_i = \begin{bmatrix} 0 \\ 0 \\ T_i \end{bmatrix} \tag{6.19}$$

A real-time air-density reading is necessary because the density of air is correlated to external environmental factors that are not always constant. Cross-sectional areas of

all the propellers are constant during any state of the flight. The area of the propellers could also contribute to the amount of total thrust generated.

The vector component magnitudes are with respect to the inertial frame is given by

$$|T_x| = \sqrt{-|T|^2 \cos(\theta)^2 (1 - \frac{1}{\cos(\theta)^2})} \tag{6.20}$$

$$|T_y| = |T| \cos(\theta) \sin(\phi) \tag{6.21}$$

The thrust mg onto the inertial z axis

$$T_{proj} = |T_z| = |T| \cos(\theta) \cos(\phi) \tag{6.22}$$

Thus we have the equation

$$T = \frac{mg}{\cos(\theta) \cos(\phi)} = k \sum \omega_i^2 \tag{6.23}$$

6.0.8 Torque

If the quadcopter reaches high speeds, the air drag will also have a considerable impact. The drag can then be modeled as

$$F_D = \begin{bmatrix} -k_d \dot{x} \\ -k_d \dot{y} \\ -k_d \dot{z} \end{bmatrix} \tag{6.24}$$

where k_d is a constant and $\dot{x}$, $\dot{y}$, $\dot{z}$ are the linear velocities.

Each rotor contributes some torque about the yaw axis and either the roll or pitch axes. The drag equation from fluid dynamics gives us the frictional force[6]

$$F_d = \frac{1}{2} \rho C_D A v^2 \tag{6.25}$$

where ρ is the surrounding air density, A is the reference area(propeller cross-section, not area swept out by the propeller), and C_d is a dimensionless constant. This implies

that the torque due to drag is given by

$$\tau_D = \frac{1}{2} R\rho C_d A v^2 = \frac{1}{2} R\rho C_D A(\omega R)^2 = b\omega^2 \tag{6.26}$$

where ω is the angular velocity of the propeller, R is the radius of the propeller, and b is some appropriately dimensioned constant. We assumed that all the force is applied at the tip of the propeller, which is inaccurate. However, the only result that matters for our purposes is that the drag torque is proportional to the square of the angular velocity. This is in fact the same relationship we found for the thrust. We can then write the complete torque generated about the yaw axis for the motors as

$$\tau_\psi = b\omega^2 + I_M \dot{\omega} \tag{6.27}$$

where I_m is the moment of inertia about the motor yaw axis, $\dot{\omega}$ is the angular acceleration of the propeller, and b is our drag coefficient. When the quadcopter hovers at a constant height we will have $\dot{\omega} = 0$, since the propellers will spin with approximately constant thrust, only slightly varied in order to keep a stable attitude. Thus ignoring this term, we simplify the expression to

$$\tau_\psi = (-1)^{i+1} b {\omega_i}^2 \tag{6.28}$$

where the $(-1)^{i+1} b\omega_i^2$ term is positive for the ith propeller if the propeller is spinning clockwise and negative if it is spinning counterclockwise. The total torque generated about the yaw axis is given by the sum of all the torques generated by each propeller.

$$\tau_\psi = b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \tag{6.29}$$

The roll and pitch torques are derived from standard mechanics. We can arbitrarily choose the i = 1 and i = 3 motors to be on the roll axis

$$\tau_\phi = \sum r \times T = L(k\omega_1^2 - k\omega_3^2) = Lk(\omega_1^2 - \omega_3^2) \tag{6.30}$$

Correspondingly, the pitch torque is given by a similar expression

$$\tau_\theta = \sum r \times T = L(k\omega_2^2 - k\omega_4^2) = Lk(\omega_2^2 - \omega_4^2) \tag{6.31}$$

where L is the distance from the center of the quadcopter to any of the four propellers. All together, we can describe the torques in the body reference frame as

$$\tau^b = \begin{bmatrix} Lk(\omega_1^2 - \omega_3^2) \\ Lk(\omega_2^2 - \omega_4^2) \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix} \tag{6.32}$$

The thrust and torque forces we have derived so far are highly simplified. We have made assumptions and simplifications, and ignored a multitude of advanced effects that contribute to the highly nonlinear dynamics of the quadcopter. We have ignored the rotational drag forces (our rotational velocities are relatively low), blade flapping (deformation of the propeller blades due to high velocities and the fact that the propeller consist of a thin flexible material), surrounding fluid velocities (wind and other disturbances created by both the quadcopter and its environment) to mention a few.

6.0.9 Equations of Motion

In the NED frame, the acceleration of the quadcopter is due to thrust generated by the motors, gravity and air drag. We can obtain the thrust vector in the NED frame by using our rotation matrix R_b^n to map the thrust vector from the body frame to the NED frame. Thus, the linear motion can be summarized as

$$m\ddot{x} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_b^n T^b + F_D \tag{6.33}$$

where $\overrightarrow{x}$ is the position of the quadcopter, g is the acceleration due to the gravity, F_D is the drag force, and T^b is the thrust vector in the body frame. While it is convenient to have the linear equations of motion in the inertial frame, the rotational equations of motion are useful to us in the BODY frame so that we can express rotations about the center of the quadcopter instead of NED frame (typical initial position for the quadcopter).

We derive the rotational equations of motion from Euler's equations for rigid body

dynamics. Expressed in vector form, Euler's equations are written as

$$I\dot{\omega} + \omega \times (I\omega) = \tau \tag{6.34}$$

where ω is the angular velocity vector, I is the inertia matrix, and τ is a vector of external torques. We can rewrite this as

$$\dot{\omega} = \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = I^{-1}(\tau - \omega \times (I\omega)) \tag{6.35}$$

We can model our quadcopter as two thin uniform rods crossed at the origin with a point mass (motor) at the end of each, and ignore the various components located close to the center of the quadcopter, including the battery. With this simplification in mind, it's clear that the symmetries result in a diagonal inertia matrix of the form

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{6.36}$$

Using this we obtain our final result for the BODY frame rotational equations of motion

$$\dot{\omega} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} \\ \tau_\theta I_{yy}^{-1} \\ \tau_\psi I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{xx}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{xx}}\omega_x\omega_y \end{bmatrix} \tag{6.37}$$

6.0.10 Degrees of Freedom - DOF

In our modelling, we will work with 6-DOF. 6-DOF means that the specific number of axes that a rigid body is able to move in three-dimensional space. The rigid-body can move in X-Y-Z axes as well as change its rotation in roll, pitch and yaw angles. The IMU that we used in our modelling owns 6-DOF, it combines 3-axis gyroscope and 3-axis accelerometer.

6.0.11 Propeller

Blade element theory relies on two key assumptions:

1. There are no aerodynamic interactions between different blade elements.
2. The forces on the blade elements are solely determined by the lift and drag coefficients

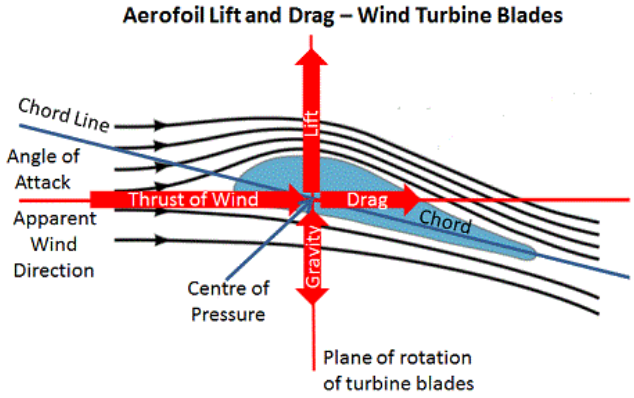


Figure 6.6 Forces acting on blade

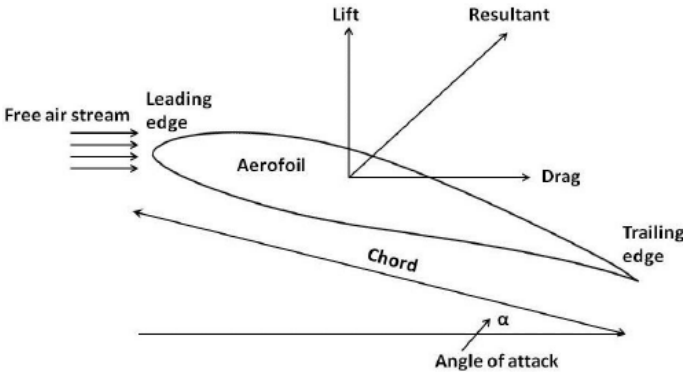


Figure 6.7 Angle of attack

Consider a blade divided up into N elements. Each of the blade elements will experience a slightly different flow as they have a different rotational speed (), a different chord length () and a different twist angle (). Blade element theory involves dividing up the blade into a sufficient number (usually between ten and twenty) of elements and calculating the flow at each one. Overall performance characteristics are determined by numerical integration along the blade span. However for simplification of this thesis, we will ignore dividing it up.

6.0.12 Air Density

In our research, we have found that in most of the papers related to quadcopter modelling, authors take the air density as a constant value. However, air density changes in places where the quadcopter hovers. So, we have implemented the air density formula in our simulation

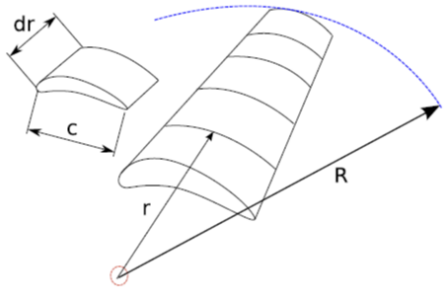


Figure 6.8 Dividing up into N pieces of a blade.

$$\rho = \frac{(PM_0)}{(R^* T_M)} \tag{6.38}$$

where is the air density, T_M is the molecular-scale temperature $T_M = T_{M,b} + L_{M,b}(H - H_b)$, P is the pressure

$$P = \begin{cases} P_b[\frac{T_{M,b}}{T_{M,b}+L_{M,b}(H-H_b))}](\frac{(g_0' M_0)}{R^* L_{M,b}}) \\ P_b e^{\frac{-g_0' M_o (H-H_b)}{R^* T_{M,b}}} \end{cases} \tag{6.39}$$

for $L_{M,b} \neq 0$.

R^* is the universal gas constant $R^* = 8314.32N\frac{m}{(kmol(K))}$, $M_0 = 28.9644\ \frac{kg}{kmol}$ is the mean molecular weight of air. For simplicity, we have taken P as a constant.

6.0.13 Sensors

In our century, all technological devices work with the data taken from sensors. By using them, we estimate our errors and recalculate our positions, velocities and angles. In this project, we used the Arduino Uno as our microcontroller.

6.0.13.1 Magnetometer

Magnetometers measure the magnetic field around them. They can detect the strength and the direction of magnetic field around them. Magnetometers are unable to distinguish various magnetic fields. By using magnetometer, the quadcopter can determine its trajectory in space. The sensor can therefore use the earth's magnetic field as a reference. The magnetometer sensor readings are also dependent on roll and pitch angles. Most of IMU sensors use magnetometers to sense earth magnetic field direction and provide an absolute measurement of the yaw angle. The local magnetic

field is dependent on the Earth's magnetic field, but also dependent on the magnetic field that caused by beside of nearby objects.

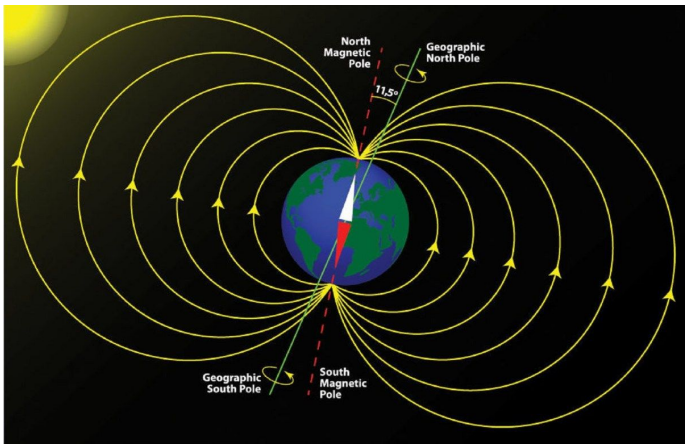


Figure 6.9 Magnetic Field of the Earth

The Earth's magnetic field is a magnetic dipole with ends at the North and South poles. The magnetic strength varies across the earth, however for simplicity we will consider the magnetic field constant. The magnetic north is divided into two different centers close to the north. The effetiveness of these centers depends on where the quadcopter is located. Thus the angle between these centers is approximately 2. At each location on the Earth, the field lines intersect the Earth's surface at a specific angle of inclination. Near the equator, the field lines are approximately parallel to the Earth's surface and thus the inclination angle in this region is 0. As one travels North from the equator the field lines become progressively steeper. At the magnetic pole, the field lines are directed almost straight down into the Earth and the inclination is 90. Consequently, the inclination angle varies with latitude.

Magnetometer is affected by two different distortions, mainly from hard irons. Hard iron term is used for nearby objects of quadcopter, while soft irons is caused by ferromagnetic objects in the vicinity of the sensor. These objects will distort earth magnetic field itself. The distortion is particularly strong inside buildings with metallic structures. These two effects will increase the error, especially in presence of variable speed motors, like we use on quadcopters and will cause error in yaw and yaw angular rate. Soft iron distortions are considered deflections or alterations in the existing magnetic field. These distortions will stretch or distort the magnetic field depending upon which direction the field acts relative to the sensor. This type of distortion is commonly caused by metals such as nickel and iron.

When we imagine a magnetic field with no distortions, a plot of m_x versus m_y will simply be a circle with a center at the origin. When there is a hard iron distortion, the center of the circle will be shifted and when there is a soft iron distortion, the shape

of the circle becomes like ellipsoid. This kind of shapes can be seen by looking at the Earth's magnetic fields.

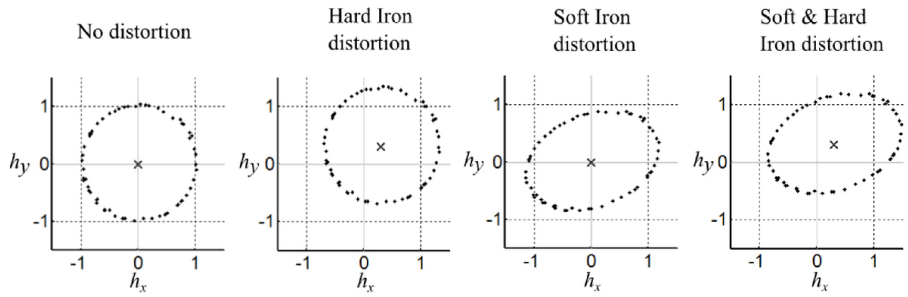


Figure 6.10 Hard and soft iron distortions

The hard and soft iron distortions should be removed from the magnetometer measurements. When we take out the magnetic effects on the magnetometer, we can estimate yaw angle accurately.

The measurements from the three-axis rate magnetometer is expressed as

$$m^b_{mag} = D^{-1}(R^b_n(\theta)m^n + b^b_{mag}) + \omega^b_{mag} \tag{6.40}$$

where $R^b_n(\theta)$ is the rotation matrix from NED to BODY with respect to $\theta = [\phi, \theta, \psi]^T$, b^b_{mag} is the magnetometer bias from hard iron distortions (local magnetic disturbance), ω^b_{mag} is a measurement noise vector. The vector m^n is the output from the magnetometer in NED coordinates when all angles are zero, and the x-axis of the NED coordinate system is pointing towards magnetic north.

Hard iron distortions can be estimated by sampling data from almost all attitudes and taking the maximum and minimum from each measurement axis before taking the average, resulting in

$$b_x = \frac{1}{2}(min_i(m^b_{mag}(1,i)) + max_i(m^b_{mag}(1,i)))$$

$$b_y = \frac{1}{2}(min_i(m^b_{mag}(2,i)) + max_i(m^b_{mag}(2,i)))$$

$$b_z = \frac{1}{2}(min_i(m^b_{mag}(3,i)) + max_i(m^b_{mag}(3,i)))$$

Instead of determining all the elements in the soft iron distortion matrix due to the

hardness of founding them, we can obtain an estimate by approximating D as a diagonal matrix. The diagonal elements can then be estimated as

$$d_{11} = \frac{\overline{d}}{\overline{d}_x}$$

$$d_{22} = \frac{\overline{d}}{\overline{d}_y}$$

$$d_{33} = \frac{\overline{d}}{\overline{d}_z}$$

where $\overline{d}$ is the average distance from the center to the data points representing the circle, and $\overline{d}_i$ is the half distance between the maximum and minimum sensor values on the i axis. Then we can write

$$\overline{d}_x = \frac{1}{2}(min_i(m^b_{mag}(1,i)) + max_i(m^b_{mag}(1,i)))$$

$$\overline{d}_y = \frac{1}{2}(min_i(m^b_{mag}(2,i)) + max_i(m^b_{mag}(2,i)))$$

$$\overline{d}_z = \frac{1}{2}(min_i(m^b_{mag}(3,i)) + max_i(m^b_{mag}(3,i)))$$

$$\overline{d} = \frac{\overline{d_x} + \overline{d_y} + \overline{d_z}}{3}$$

It is necessary to perform filtering and calibration of the magnetometer to remove the bias b^b_{mag} and noise ω^b_{mag}. The calibrated measurements are denoted

$$\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = m^b_{imu} - b^b_{mag} \tag{6.41}$$

If the magnetometer is sitting in a local horizontal plane leveled to the surface of the Earth such that $\phi = \theta = 0$, the magnetic heading angle ψ_m is recognized as the

direction planar with the surface of the Earth satisfying

$$tan(\psi_m) = \frac{m_y}{m_x} \tag{6.42}$$

If the roll and pitch angles are known, the magnetic readings m_x, m_y, m_z can be transformed to the horizontal plane according to

$$\begin{bmatrix} m_x^h \\ m_y^h \\ m_z^h \end{bmatrix} = R_{y,\theta} R_{x,\phi} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} \tag{6.43}$$

or, can be written like

$$\begin{bmatrix} m_x^h \\ m_y^h \\ m_z^h \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} \tag{6.44}$$

The sign of the arguments m_x^h and m_y^h must be taken into account when computing the magnetic heading. This can be done by using the following mapping:

$$\psi_m = \begin{cases} 180 - \frac{180}{\pi} arctan(\frac{h_y}{h_x}), & \text{if } h_x < 0 \\ -\frac{180}{\pi} arctan(\frac{h_y}{h_x}), & \text{if } h_x > 0, h_y < 0 \\ 360 - \frac{180}{\pi} arctan(\frac{h_y}{h_x}), & \text{if } h_x > 0, h_y > 0 \\ 90 & \text{if } h_x = 0, h_y < 0 \\ 270 & \text{if } h_x = 0, h_y > 0 \end{cases}$$

To determine true North heading, ψ, the appropriate declination angle (2°) which depends on the latitude must be added or subtracted.

There is magnet inside DC motor, thus it creates magnetic field, so our magnetometer can be effected heavily. The more power that goes to the motors, the deflection and the interference gets larger. When we design of our quadcopter, we have paid attention to leave as much room as possible between magnetometer and the lines that are carrying signals.

The magnetometer we used in our modelling is HMC-5883L.

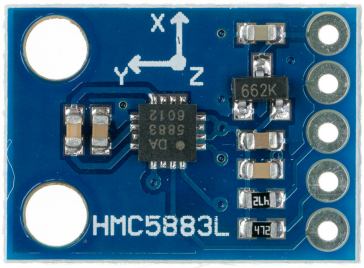


Figure 6.11 HMC5883L Module.

6.0.13.2 Ultrasonic Sensor

A small ultrasonic sensor can be used to measure the distance between of our quadcopter and, for example a barrier. This sensor works by sending out sound waves and detecting the return of that wave.

The distance estimation can be found with this equation:

$$d_{dist} = \frac{(t_{end} - t_{start})}{2} 340 m/s \tag{6.45}$$

where t_{start} is the time when a sound wave is sent out, t_{end} is the time when the return of the sound wave is detected, d_{dist} is the distance from the sensor to the barrier. 340 is approximately the speed of sound in air at 20°.

When we receive signals from the outside, we may receive unwanted high frequency noise. In order to split up those signals, we can use low-pass filter. A low-pass filter is a filter that passes signals with a frequency lower than selected cutoff frequency and attenuates signals with frequencies higher than the cutoff frequency.

The ultrasonic sensor we used in our modelling is HC-SR04.

6.0.13.3 Inertial Measurement Unit

The IMU measurement model is only valid for low-speed applications such as quadcopters, marine craft moving etc. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock.

Figure 6.12 HC-SR04 Module.

The IMU we used in our quadcopter is MPU-6050. MPU-6050 contains three accelerometers and three gyroscopes within inside. Three accelerometers are mounted at right angles to each other, so that acceleration can be measured in three axes; x, y and z. It is very accurate, as it contains 16-bits analog to digital conversion hardware for each channel. Therefore it captures the x, y, and z channel at the same time. The sensor uses the I2C-bus to interface with the Arduino microcontroller. Gyroscope measures the rotational acceleration of the sensor while accelerometers are used to determine position and orientation of the quadcopter in flight. The MPU 6050 features four different sensitivity ranges for the accelerometer and the gyroscope with four resulting scale factors. To transform the sensor data into units of gravity and units of degrees per second, the accelerometer and gyroscope measurements must be divided by 16384 and 131 respectively. In order to obtain the acceleration in meters per second, the accelerometer sensor data must be multiplied by 9.81 m/s. The accelerometer is very sensitive to vibrations and contains a fair amount of high frequency noise. An improvement to the estimate can thus be made by filtering data using the low-pass filter. Also the gyroscopes suffer from low frequency random walk noises, which accumulates when integrated in the angle estimates as will represented later in this article. To decrease this effect, the data then can be filtered using the high-pass filter. However, this thesis does not aim to filter designing.

Three accelerometers and three gyroscopes are mounted at right angles to each other, so that acceleration and angular rate are measured independently in three axes.

Gravity has a nominal acceleration of 9.81 m/s^2, but if the measurement is biased, the IMU may report 9.75 m/s^2. The real value between the real value and the output is bias.

For each power-up of the IMU, the initial bias is different. A very repeatable bias allows for better “tuning” of IMU parameters by the INS to quickly reach a good estimate of the bias.

While the IMU is powered on, the initial bias changes over time. This change in bias is often related to temperature, time and/or mechanical stress on the system.

The three gyroscopes and three accelerometers are mounted orthogonal to each other. The mountings, however, have errors and so are not perfectly 90 degrees. This leads to a correlation between sensors. If the two axes were perfectly orthogonal, they do not measure any of the effect of gravity. If there is a non-orthogonality, the other axes also measure the other's acceleration, which leads to a correlation in the measurements. Continuous estimation and correction during system operation is also an approach used to minimize this effect.

The gravity of Earth is modeled as a constant vector:

$$g^n = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{6.46}$$

Gravity increases from 9.789 m/s^2 at the equator to 9.832 m/s^2 at the poles. The nominal "average" value at the surface of the Earth, known as standard gravity, is g = 9.80665 m/s^2.

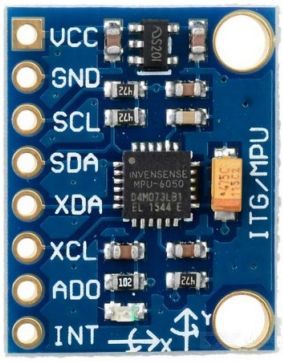


Figure 6.13 MPU6050 Module

6.0.14 APC-220 Module

The APC220 radio module provides a simple and economic solution to wireless data communications. Integrates an embedded high speed microprocessor and high performance IC that creates a transparent UART/TTL interface, and elminates any need for packetizing and data encoding.

It has a transmit distance up to 1000m (line of sight) @9600 bps, 256 bytes data buffer,

high sensitivity (-112dbbm @9600 bps), GFSK modulation and UART/TTL interface. We have used this module in order to send the sensor values to our Arduino COM-port screen.

Figure 6.14 APC-220 Module

6.0.15 PD Controller

In quadcopter, we have 3-positions and 3-angles but only 4 control inputs, which are ω_i. Torque τ_ϕ affects the the angle ϕ, τ_θ affects the the angle θ, τ_ψ affects the the angle ψ.

The derivatives of the measured angles, $\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$, will give the derivative of error, and their integral will provide us give us the actual error. The desired velocities and angles will be all zero. Thus we get

$$\tau_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} Lk(\omega_1^2 - \omega_3^2) \\ Lk(\omega_2^2 - \omega_4^2) \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix} = \begin{bmatrix} -I_{xx}(K_d\dot{\phi} + K_p\int_0^T \dot{\phi}dt) \\ -I_{yy}(K_d\dot{\theta} + K_p\int_0^T \dot{\theta}dt) \\ -I_{zz}(K_d\dot{\psi} + K_p\int_0^T \dot{\psi}dt) \end{bmatrix}$$

The correct angular velocities are given as

The controller aims to drive the angles to zero. The angles are not completely driven to zero. This is a common problem with using PD controllers for mechanical systems.

6.0.16 PID Controller

Proportional, derivative and integral terms describe how the error term is handled. If the proportional gain is too low, the amount of time that the quadcopter reaches the correct position will be more. If the proportional gain coefficient is too high, the quadcopter will probably overshoot and become unstable.

The integral gain can remove the steady state error which can be seen by using only both the proportional and derivative gains. If the integral gain value gets too high the quadcopter will begin to show signs of slow reaction and a decreased effect of the proportional gain as consequence, it will also start to oscillate with a low frequency.

In signal processing, control theory, electronics, and mathematics, overshoot is the occurrence of a signal or function exceeding its target. It arises especially in the step response of bandlimited systems such as low-pass filters. K_d is important for preventing overshoot.

The block diagram of a PID controller is given below

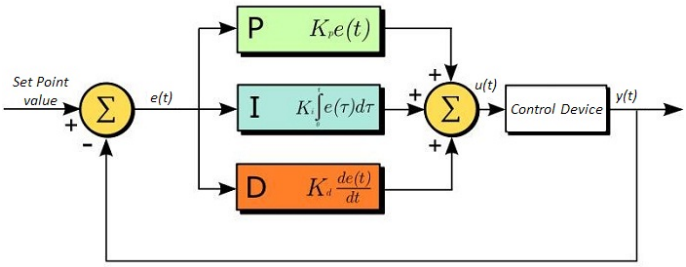


Figure 6.15 PID controller diagram

We can write the error and control input as

$$e(t) = x_d(t) - x(t)$$

$$u(t) = K_p e(t) + K_I \int e(\tau)d\tau + K_D \frac{de(t)}{dt}$$

where $u(t)$ is the control input, $e(t)$ is the difference between the desired state $x_d(t)$ and the present state $x(t)$, and K_P , K_I and K_D are the parameters for the proportional, integral and derivative elements of the PID controller. The equations are

$$e_\phi = K_d\dot{\phi} + K_p \int_0^T \dot{\phi}dt + K_i \int_0^T \int_0^T \dot{\phi}dtdt$$

$$e_\theta = K_d\dot{\theta} + K_p \int_0^T \dot{\theta}dt + K_i \int_0^T \int_0^T \dot{\theta}dtdt$$

$$e_\psi = K_d\dot{\psi} + K_p \int_0^T \dot{\psi}dt + K_i \int_0^T \int_0^T \dot{\psi}dtdt$$

In the frequency-domain we had the following equation

$$\frac{C(s)}{E(s)} = \frac{K_d s^2 + K_p s + K_I}{s} = \frac{K_d(s^2 + \frac{K_p}{K_D})s + \frac{K_I}{K_D}}{s}$$

as seen, there are two zeros and one pole.

However, PID controller has its integral wind-up problem. For a control system with a wide range of operating conditions, it may happen that the control variable reaches the actuator limits. When this happens the feedback loop is broken and the system runs as open loop because the actuator will remain at its limit independently of these process output. If a controller with integrating action is used, the error will continue to be integrated. This means that the integral term may become very large or, it "winds up". It is then required that the error has opposite sign for a long period before things return to normal. The consequence is that any controller with integral action may give large transients when the actuator saturates.

In some cases, integral wind-up may looses to settling rather than oscillatons, and also in some cases, it may loose the control of reaching the steady-state.

If there is a large disturbance in the process variable, this large disturbance is integrated over time, becoming a still larger control signal (due to the integral term). However, even once the system stabilizes, due to the integral is still large, thus causing the controller to overshoot its target. It may begin to become unstable, or simply take an incredibly long time to reach a steady state. In order to avoid this, we disable the integral function until we reach something close to the steady state. Once we are in a controllable region near the desired steady state, we turn on the integral function, which pushes the system towards a low steady-state error.

6.0.16.1 Zieger-Nichols Method

There is a heuristic method found by Zieger and Nichols for controller parameters in the 1940's. The parameters are given below

At first, we set the K_i and K_d values to zero, the K_p is increased until it oscillates. So that we find K_u value and then we can find K_d and K_i values with the oscillation period T_u.

We have simulated the Ziegler-Nichols method in Simulink. The simulation results are given below

Zieger-Nichols method			
Control type	K_p	K_i	K_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$1.2K_p/T_u$	-
PD	$0.8K_u$	-	$K_pT_u/8$
Classic PID	$0.6K_u$	$2K_p/T_u$	$K_pT_u/8$
Pessen Integral Rule	$0.7K_u$	$2.5K_p/T_u$	$K_pT_u/20$
Some overshoot	$0.33K_u$	$2K_p/T_u$	$K_pT_u/3$
No overshoot	$0.2K_u$	$2K_p/T_u$	$K_pT_u/3$

Figure 6.16 Ziegler-Nichols method table.

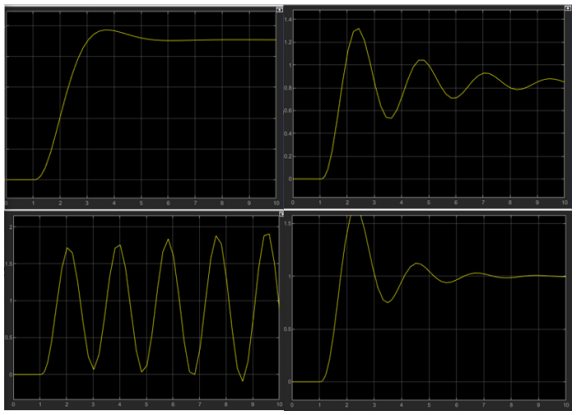


Figure 6.17 The Simulink results of Ziegler-Nichols method. We start from setting the K_p, K_d, K_i values to zero. Then we start to increase the value K_p so that the oscillation can be seen effectively. After that, we take the T_u parameter and fill the table given above. After all, the PID output can also be seen in this figure.

6.0.17 Mechanics

We have seen that using Arduino Uno would be useful in our sensor readings. An accelerometer is an electromechanical device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic - caused by moving or vibrating the accelerometer. IMU is a chip where 3-axis accelerometer and gyroscope are all in two. It contains 16-bits analog to digital conversion hardware for each channel. It calculates both the static acceleration and dynamic acceleration. We can measure both roll and pitch using accelerometer and gyroscope (IMU) and estimate yaw. We can use the data acquired from the IMU to detect any drift in yaw angle, but this estimation is useless if we want to turn the quadcopter towards a specific yaw angle. In order to suitable measurements for yaw, we can use Magnetometer. To get measurement of hovering height, range sensor(HC-04) can be used in combination with the GPS measurements. Sensor technology in quadcopters is enhanced and later on will be introduced. Radio communication is essential for controlling the quadcopter, thus we have used WiFi

module APC-220.

6.0.18 Simulation

We have coded this quadcopter simulation with Java programming language. We have used OpenGL and LWJGL for rendering, and we used Blender for a simple quad-design. Our simulation uses mathematical formulas to represent a quadcopter's equation of motions.

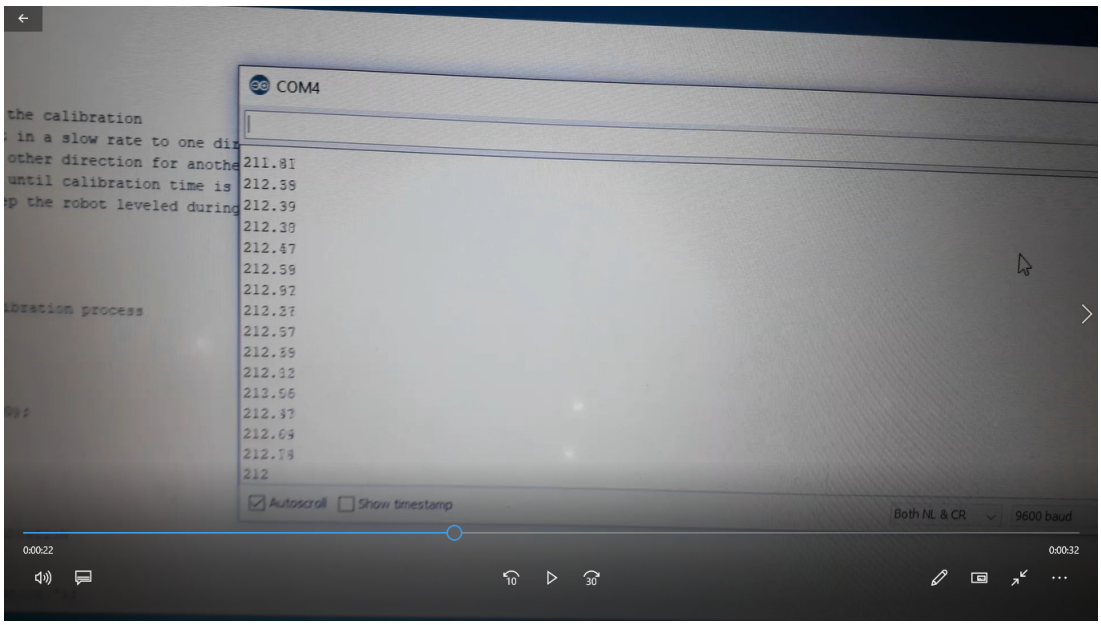


Figure 6.18 HMC5883L Sensor Readings

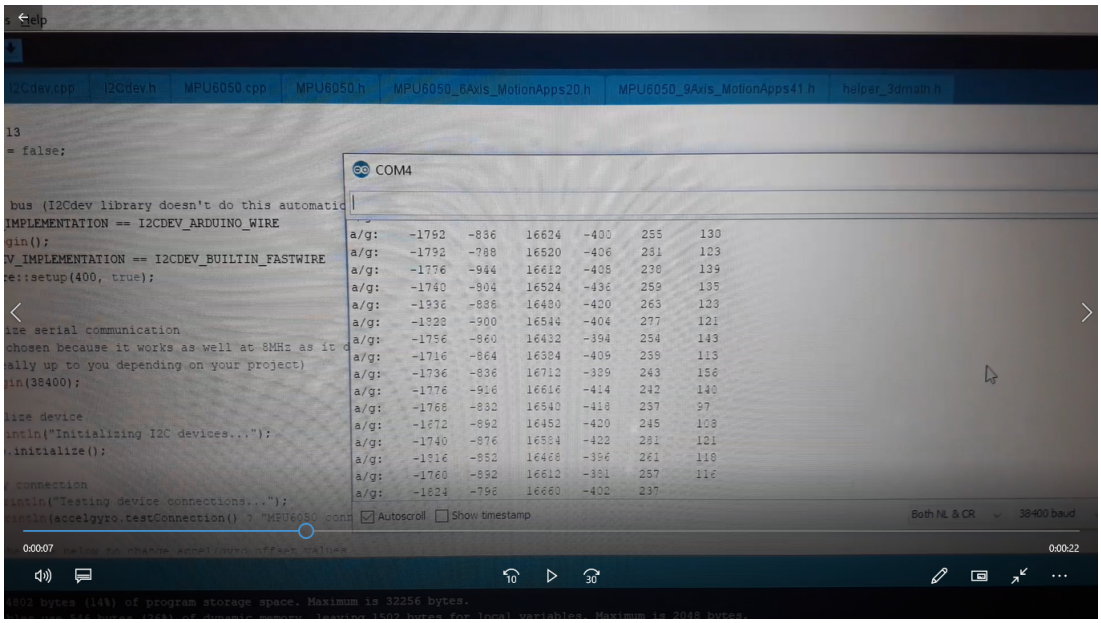


Figure 6.19 MPU6050 Sensor Readings

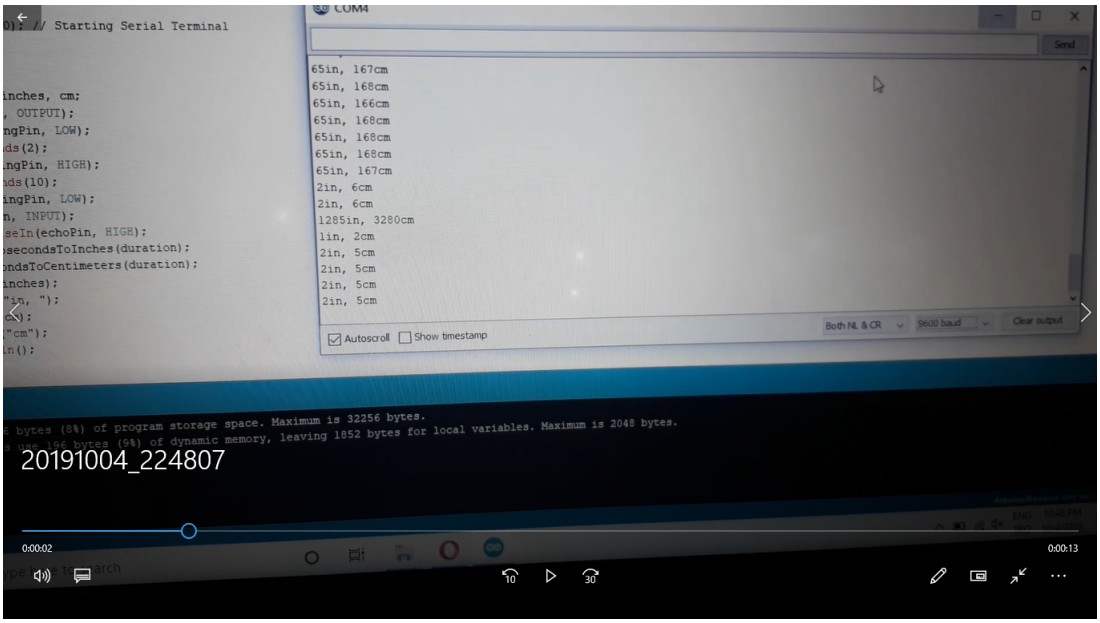


Figure 6.20 HC-S04 Sensor Readings

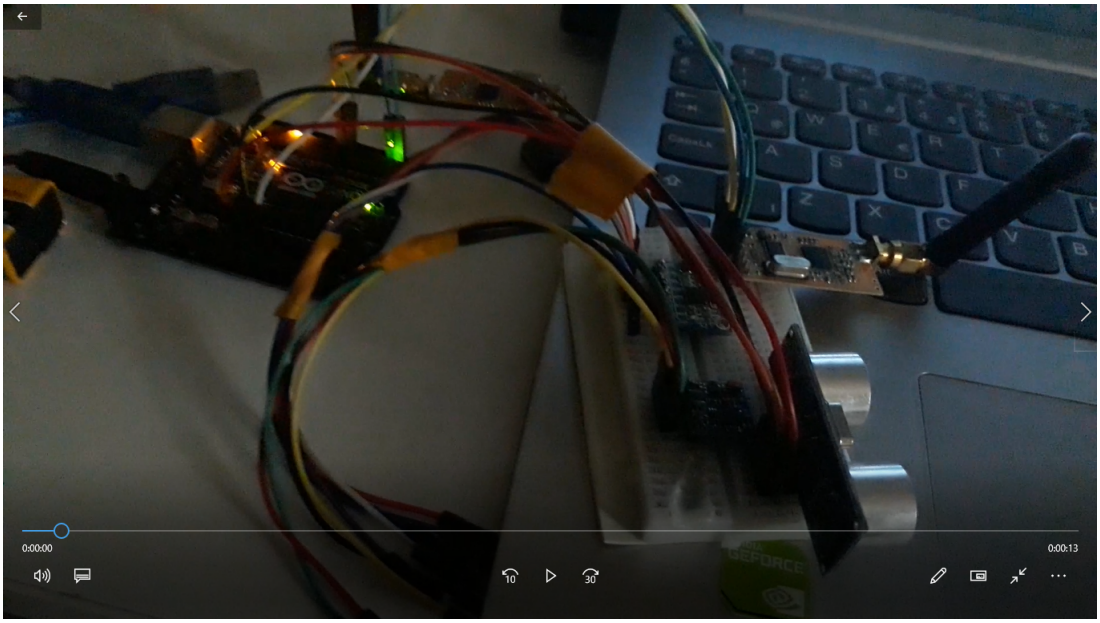


Figure 6.21 All modules we used.

Figure 6.22 A screenshot of our quadcopter simulation.

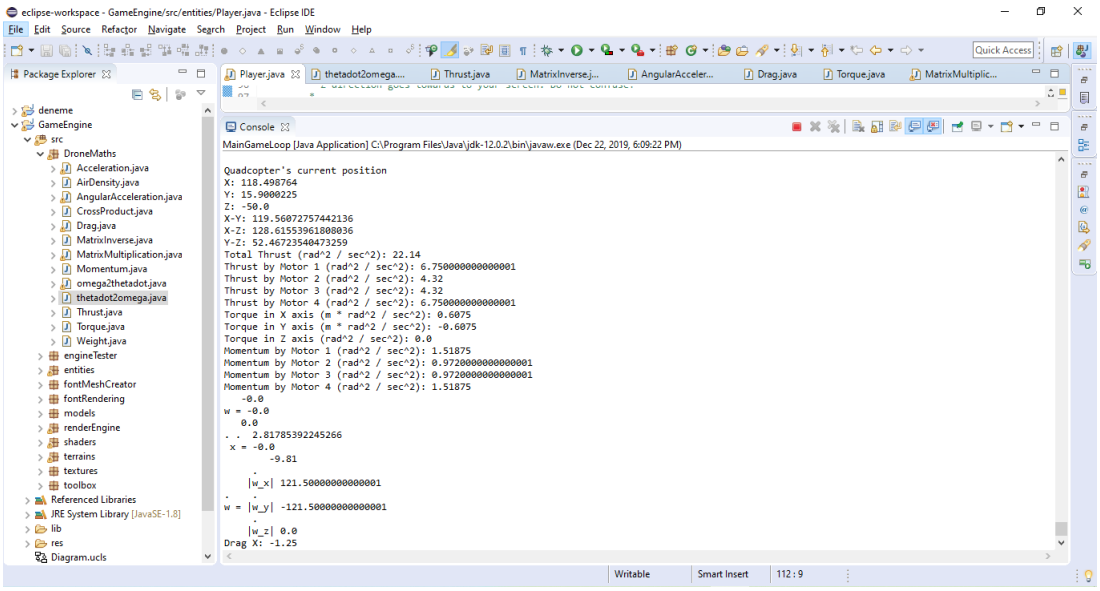


Figure 6.23 A screenshot of our quadcopter simulation results.

7
Experimental Results

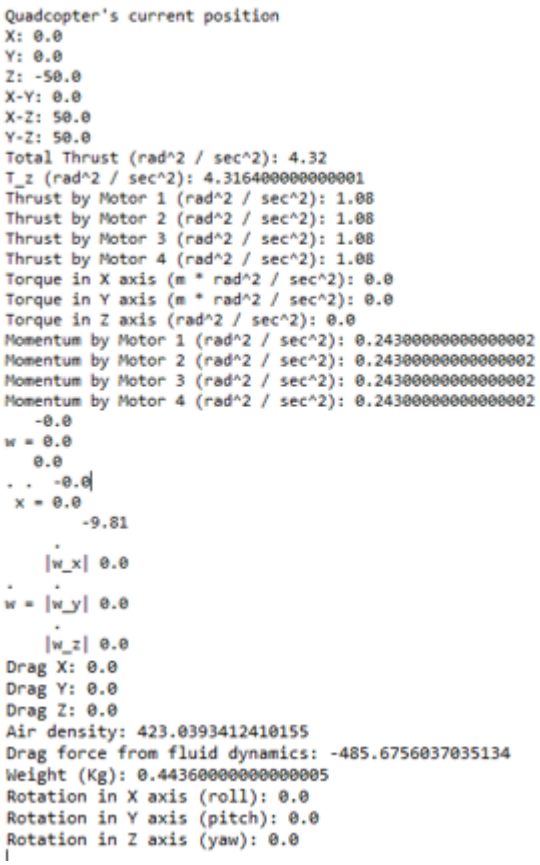


Figure 7.1 A simulation result of the quadcopter when hovering.

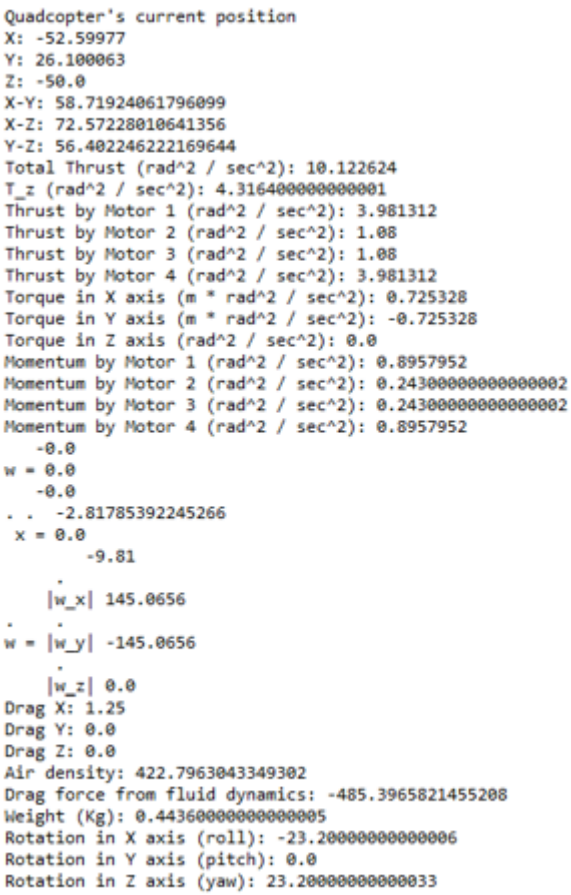


Figure 7.2 A simulation result of the quadcopter when rolling.

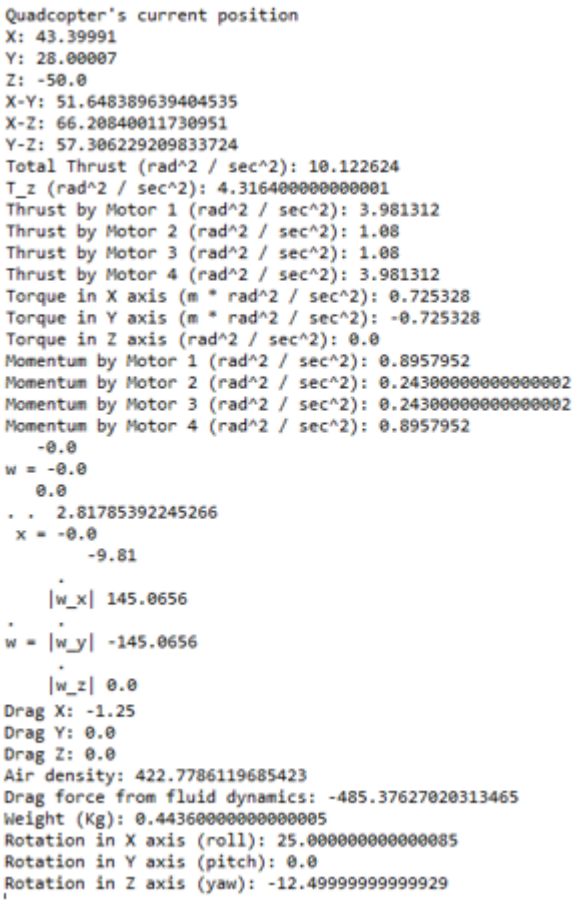


Figure 7.3 Simulation results of the quadcopter when pitching.

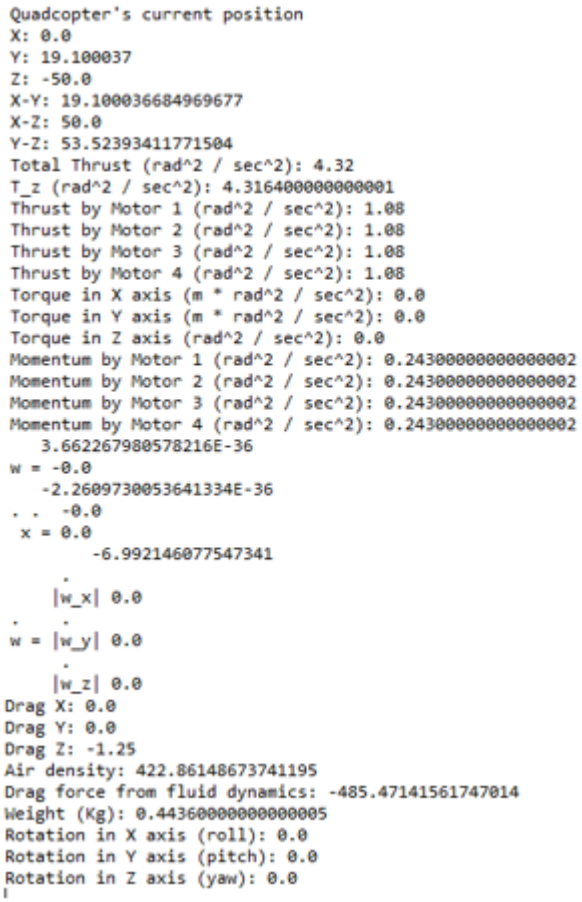


Figure 7.4 Simulation results of the quadcopter when landing.

8
Performance Analysis

The quadcopter simulation we have designed works properly. The calculations we have made is printed on the console. However we had a thinking of printing them on the simulation screen itself, rather than the console. But we have failed to do it due to the amount of time we have spent for correctness of the calculation results of the equations of motions. We have wanted to use ND4J library, but for some reason, the simulation failed to use it so we had to write of our own matrix library. It is one of our failures. When we have started to design the quadcopter simulation, we had an also thinking of designing it autonomously, however it seemed more reasonable to use the quadcopter via keyboard buttons rather than the GUI console.

9
Conclusion

As the conclusion, we have seen that the biggest problem of the quadcopters is the battery problem. By solving this problem, the quadcopter industry will be changed significantly. Due to time limit we have, we weren't fully able to design the quadcopter mechanically. This thesis will be a basis to swarm technology in the future.

References

[1] P.-l. Limited. (2019). 28 ways military tech changed our lives, [Online]. Available: https://www.pocket-lint.com/gadgets/news/143526-how-military-tech-changed-our-lives (visited on 05/31/2019).

[2] J. B. Lehn, "Model, design and control of a quadcopter," 2015.

[3] T. W. M. Randal W. Beard, *Small Unmanned Aircraft*. Princeton University Press, 2012.

[4] R. Rojas, "Models for dc motors,"

[5] A. Gibiansky, "Quadcopter dynamics, simulation, and control," 2012.

[6] C. Pupaza, "Mathematical model of a multi-rotor drone prototype and calculation algorithm for motor selection, proceedings in manufacturing systems," 2018.

Curriculum Vitae

FIRST MEMBER

Name-Surname: Omer Faruk Yavuz
Birthdate and Place of Birth: 01.01.1999, Eskişehir
E-mail: l1117083@std.yildiz.edu.tr
Phone: 0534 696 32 85
Practical Training: Telcomobil Inc. Software Department

Project System Informations

System and Software: Windows Operating System, Java, MATLAB, OpenGL, Arduino, Wireless module, Light Weight Java Game Library, C programming language
Required RAM: 8GB
Required Disk: 256MB