Autonomous Flight of Quadcopter using Visual Odometry

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY
IN
MECHANICAL ENGINEERING

Submitted by:

PRASHAM PATEL (17BME087)

PURNA PATEL (17BME089)

MECHANICAL ENGINEERING DEPARTMENT
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY

Year: 2020-2021

Declaration

This is certify to that

- The thesis comprises my original work towards the degree of Bachelor of Technology in Mechanical Engineering at Nirma University and has not been submitted elsewhere for degree.
- Due acknowledgement has been made in the text to all other material used

Sign

Name: Prasham Patel
Roll No: 17BME087

Sign

Name: Purna Patel
Roll No: 17BME089

Undertaking for Originality of the Work

We, Prasham Patel(17bme087) & Purna Patel (17bme089) gives undertaking that the Major Project entitled “Autonomous Flight of Quadcopter using visual Odometry“ submitted by us, towards the partial fulfillment of the requirements for the degree of Bachelor of Technology in Mechanical Engineering of Nirma University, Ahmedabad, is the original work carried out by us. We give assurance that no attempt of plagiarism has been made. We understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Signature of Student

Prasham Patel:

Purna Patel:

Date:

Place: Ahmedabad

Endorsed by:

Certificate

TO WHOMSOEVER IT MAY CONCERN

This is to certify that, **Mr. Prasham Patel** and **Mr. Purna Patel**, students of B.Tech (Mechanical engineering), VIIIth Semester of, Institute of Technology, Nirma University has satisfactorily completed the project report titled **Autonomous Flight of Quadcopter using Visual odometery**.

Date:

Guide and Co-guide Name(s)
Prof. Jatin Dave
Guide, Assistant Professor,
Department of Mechanical Engineering,
Institute of Technology
Nirma University, Ahmedabad.

Co-guide Name
Prof. Akash Mecwan
Guide, Assistant Professor,
Department of Electronica and Communication Engineering,
Institute of Technology
Nirma University, Ahmedabad.

Dr. V.J.Lakhera
Head and Professor,
Department of Mechanical Engineering,
Institute of Technology
Nirma University, Ahmedabad

Approval Sheet

The Project entitled **Autonomous Flight of Quadcopter using Visual Odometry** by **Prasham Patel (17BME087)**, **Purna Patel (17BME089)** is approved for the degree of Bachelor of Technology in Mechanical Engineering

Examiners

_____

_____

_____

Date: _____

Place: _____

Acknowledgments

We feel obliged to thank Nirma University to provide such an excellent hands-on learning experience by providing us with the best equipments and facilities necessary for completion of this project. Secondly, a special thanks to our Guide Dr. Jatin Dave and Co-Guide Dr. Akash Mecwan for their enthusiastic and pro-active support and guidance .

We would also like to thank Nirma's Team ROBOCON as well as Team Arrow and their Faculty Advisers Dr. Mihir Chauhan and Dr. Absar Lakdawala for their timely assistance, without which the completion of this project would not have been possible.

Finally, we are grateful to our friends and family for their constant support.

Prasham Patel:

Purna Patel:

Date:

Place:

Abstract

Autonomous navigation of aerial vehicles has always been more challenging than that of ground vehicles due to the limitations of the type of sensors which could be used as well as the amount sensors it could carry due to limited payload. However, as the technology as evolved the powerful onboard processors have made it possible to use Camera and LiDAR sensors which are computationally expensive to use in such dynamic real time systems. By exploiting the high speed processing power of the commonly available onboard processor we have developed a prototype Quadcopter equipped with cameras and SONAR sensors controlled by a three level system architecture.

The main aim of the project was to develop a platform compatible with variety of sensors and capable of running in real time. The path planning and visual odometry algorithms are discussed in the initial section of this report, followed by the system architecture of the prototype. And finally the testing of the system is discussed. Each algorithm is tested separately and communication between the different levels as well as with that of the ground station is also tested. The final prototype can successfully maintain its orientation while following a given path and is also able to keep an account of its current position with respect to target with use of visual odometry.

Key words: Aerial Robotics, Quadrotor VTOL Aircraft, Quadcopter, Aerial vehicle, Autonomous Navigation, Visual Odometry, Trajectory Generation

Content

Table of Figures

CHAPTER 1

Introduction

The Industrial Revolution 4.0 has noticed major advancements in the field of autonomous navigation. Autonomous systems are taking over the field of technology rapidly due to its wide applications and its accuracy of operation than manual controlled systems. In this world of autonomous systems, flight automation is one of the most important field. In fact one of the most widely used automation system is the auto pilot in the modern airliners. Aerial robotics is a field of robotics which deals with the problems of autonomous flights and make them perform various tasks.

In this report we will present and discuss the different stages of automation required in order to automate the flight of a quadcopter. This project is an extension of our previous projects, “Design and Development of Quadrotor VTOL Aircraft” and “Dynamic Modelling of Quadrotor VTOL Aircraft”, in which we have designed, developed and studied the dynamics of the quadrotor VTOL aircraft. In this project, we dug deeper into the part of automation. Further in the Report we will discuss about PID equations for the quadcopter control, autonomous trajectory generation and use of visual odometry in order to determine the current position of the quadcopter.

Autonomous systems are taking over the field of technology rapidly, pre-dominantly in wheeled mobile robots which have been successfully deployed for warehouse management, patrolling and even in space exploration. However, same cannot be said about aerial robots. This is mainly due to the fact that aerial robots have limited amount of payload that they can carry and thus greatly reduce the number of sensors which can be attached to it. The second issue is that an aerial robot is a system with 6 degrees of freedom and thus contact based sensors are useless unlike in the wheeled robots. Due to such limitations or rather constraints, vision based navigation systems are gaining popularity.

Camera can provide information rich data to accurately estimate the environment. Deep-learning methods can also be used to identify obstacles, objects of interest or landmarks for position estimation. Besides it, camera coupled with onboard IMU can be used for visual odometry, which is essential for processes like SLAM.

After the flight computer provides the position and target position, it has to decide the path to follow in order to get to the location. The algorithm of trajectory generation comes into picture in this case. This algorithm is studied and further derived in a useful way in this project. After the trajectory generator provides trajectory, the quadcopter need to follow this trajectory smoothly. In order to achieve this, PID controllers are used. The PID equations are derived in this report ahead and applied successfully in the prototype.

In this report development of a vision based navigation system has been discussed. Functions such autonomous hover, autonomous position holding and object tracking are also developed and tested on the prototype. Other than this, the program of trajectory generation and tracking are also developed in MATLAB environment and simulated successfully. Chiefly, platforms system architecture and its ability to generate a trajectory and follow it using a camera and sonars, identifying and tracking of object of interest, and calculating its distance from the object of interest is discussed. The system developed is expected to be used as a platform for further research in autonomous flight at Nirma's Center for robotics.

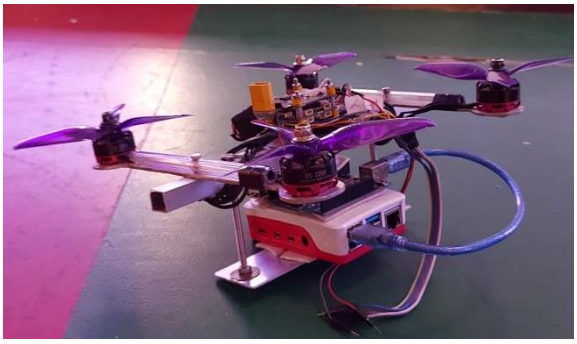

Figure 1.1 Prototype

CHAPTER 2

Previous Work

Previous work includes designing, prototyping, testing and calculation of the dynamic model of the quadrotor VTOL aircraft. It is a Vertical Landing and Take-off (VTOL) aircraft with minimal moving parts and no control surfaces. The aircraft changes its attitude by varying the angular velocities of the four motors. The following subsections discusses the progress of the previous work in brief.

2.1 Design

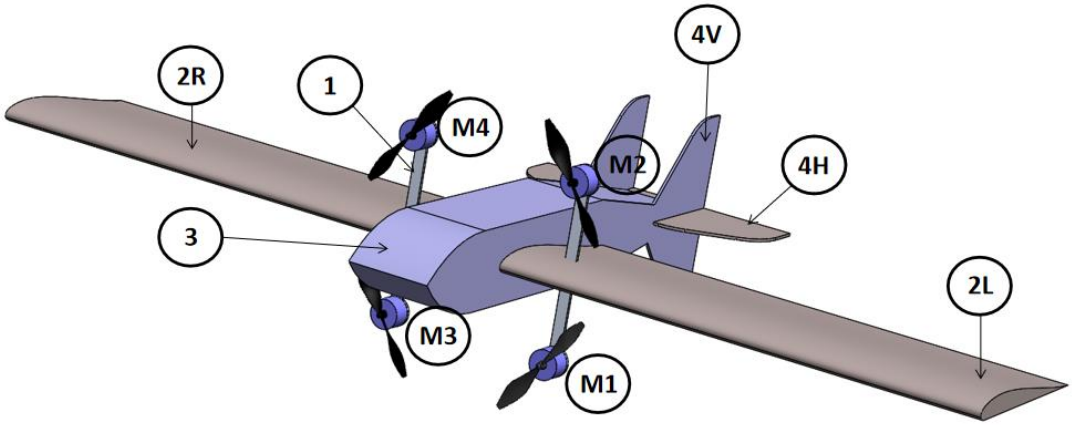

Figure 2.1: Isometric View

The aircraft has 4 BLDC (Brushless DC) motors (M1, M2, M3 and M4), which are used to provide thrust and to control its attitude. Unlike other aircrafts, it does not have any control surfaces. The motors (M1, M2, M3 and M4) are mounted on the frame (1) in a quadrotor configuration. Solid wings (2R and 2L) are mounted such that they remain perpendicular to the frame (1). During take-off and hover, the aircraft will be aligned such that frame (1) remains parallel and the wings (2R and 2L) remain perpendicular to the ground. During glide, the wings (2R and 2L) will remain parallel to the ground. Same 4 BLDC motors (M1, M2, M3 and M4) will provide thrust for the aircraft to glide and will be used to maneuver the aircraft.

2.2 Controls and Maneuverability

Unlike other aerial vehicles which uses aerofoils, this aircraft has no control surfaces. This makes its design very simple to manufacture and to maintenance. This also makes the structure rigid as there are no delicate servo motor mountings used to actuate the control surfaces. This new design has a unique methods to maneuver in gliding mode. As to all VTOL aircrafts, this design have two modes of operation: Hovering and Gliding. Thus the dynamic model of the aircraft is divided into two modes: Hovering mode and Gliding mode.

2.2.1 Controls in Hovering Mode

In figure 2.2, the first image shows the configuration of motors, their direction of rotation and the orientations of axis for hovering mode and the second image shows the direction of rotation and the configuration of four motors when viewed from the top.

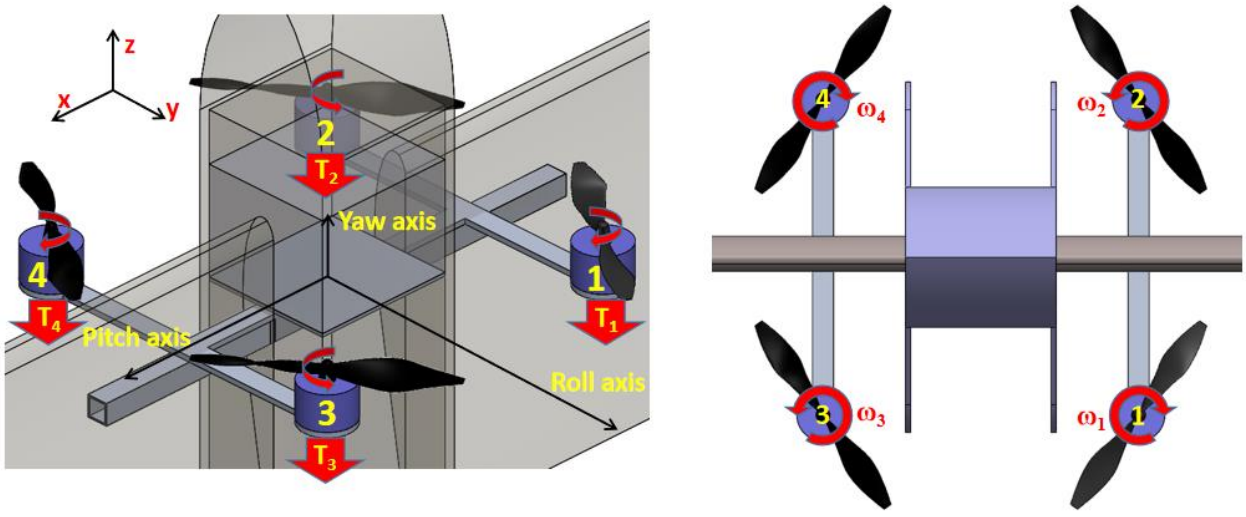

Figure 2.2: Configuration of motors, its direction of rotation and orientation of axis for hovering mode

2.2.1.1 For steady hover:

$T_1 + T_2 + T_3 + T_4 = mg$

$\omega_1 + \omega_2 + \omega_3 + \omega_4 = 0$

2.2.1.2 For motion in only z direction:

$$\omega_1 + \omega_2 + \omega_3 + \omega_4 = 0$$

$$T_1 + T_2 + T_3 + T_4 - mg = m * a_z$$

2.2.1.3 For pitching motion:

$$(\mathrm{T}_1+\mathrm{T}_2+\mathrm{T}_3+\mathrm{T}_4)*\cos\phi - mg = m*a_z$$

$$(\mathrm{T}_1+\mathrm{T}_2+\mathrm{T}_3+\mathrm{T}_4)*\sin\phi = -m*a_y$$

$$\mathrm{T}\,m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{pitch}\begin{bmatrix} 0 \\ 0 \\ T_1+T_2+T_3+T_4 \end{bmatrix} \quad (1)$$

$$I_{pitch}*\ddot{\phi} = ((\mathrm{T}_1+\mathrm{T}_3)-(\mathrm{T}_2+\mathrm{T}_4))*\mathrm{L}_1 \quad (2)$$

2.2.1.4 For rolling motion:

$$(\mathrm{T}_1+\mathrm{T}_2+\mathrm{T}_3+\mathrm{T}_4)*\cos\theta - mg = m*a_z$$

$$(\mathrm{T}_1+\mathrm{T}_2+\mathrm{T}_3+\mathrm{T}_4)*\sin\theta = m*a_x$$

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{roll}\begin{bmatrix} 0 \\ 0 \\ T_1+T_2+T_3+T_4 \end{bmatrix} \quad (3)$$

$$I_{roll}*\ddot{\theta} = ((\mathrm{T}_1+\mathrm{T}_2)-(\mathrm{T}_3+\mathrm{T}_4))*\mathrm{L}_2 \quad (4)$$

2.2.1.5 For Yaw motion:

$$\mathrm{T}_1+\mathrm{T}_2+\mathrm{T}_3+\mathrm{T}_4 = mg$$

$$(\omega_1+\omega_2+\omega_3+\omega_4)*\mathrm{I}_\mathrm{m} = -\omega_{yaw}*I_{yaw}$$

$$I_{yaw}*\ddot{\psi} = -(\mathrm{M}_1+\mathrm{M}_2+\mathrm{M}_3+\mathrm{M}_4) \quad (5)$$

2.2.1.6 General dynamic model:

For linear accelerations:

Total rotation of the drone can be defined as

$$\mathrm{R} = \mathrm{R}_{\mathrm{yaw}}*\mathrm{R}_{\mathrm{pitch}}*\mathrm{R}_{\mathrm{roll}}$$

$$R = \begin{bmatrix} C\psi C\theta - S\phi S\psi S\theta & -C\phi S\psi & C\psi S\theta + C\theta S\phi S\psi \\ C\theta S\psi + C\psi S\phi S\theta & C\phi C\psi & S\psi S\theta - C\psi C\theta S\phi \\ -C\phi S\theta & S\phi & C\phi C\theta \end{bmatrix} \quad (6)$$

General dynamic equation of drone for linear accelerations in hovering mode can be described as follows.

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R\begin{bmatrix} 0 \\ 0 \\ T_1+T_2+T_3+T_4 \end{bmatrix} \tag{8}$$

For angular accelerations:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} C\theta & 0 & -C\phi S\theta \\ 0 & 1 & S\phi \\ S\theta & 0 & C\phi C\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{9}$$

General dynamic equation of drone for angular accelerations in hovering mode can be described as follows.

$$I\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} ((\mathrm{T}_1+\mathrm{T}_3)-(\mathrm{T}_2+\mathrm{T}_4))*\mathrm{L}_1 \\ ((\mathrm{T}_1+\mathrm{T}_2)-(\mathrm{T}_3+\mathrm{T}_4))*\mathrm{L}_2 \\ -(\mathrm{M}_1+\mathrm{M}_2+\mathrm{M}_3+\mathrm{M}_4) \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I\begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{10}$$

2.2.2 Controls in Gliding Mode

In figure 2.3, the first image shows the configuration of motors, their direction of rotation and the orientations of axis for gliding mode and the second image shows the direction of rotation and the configuration of four motors when viewed from the front.

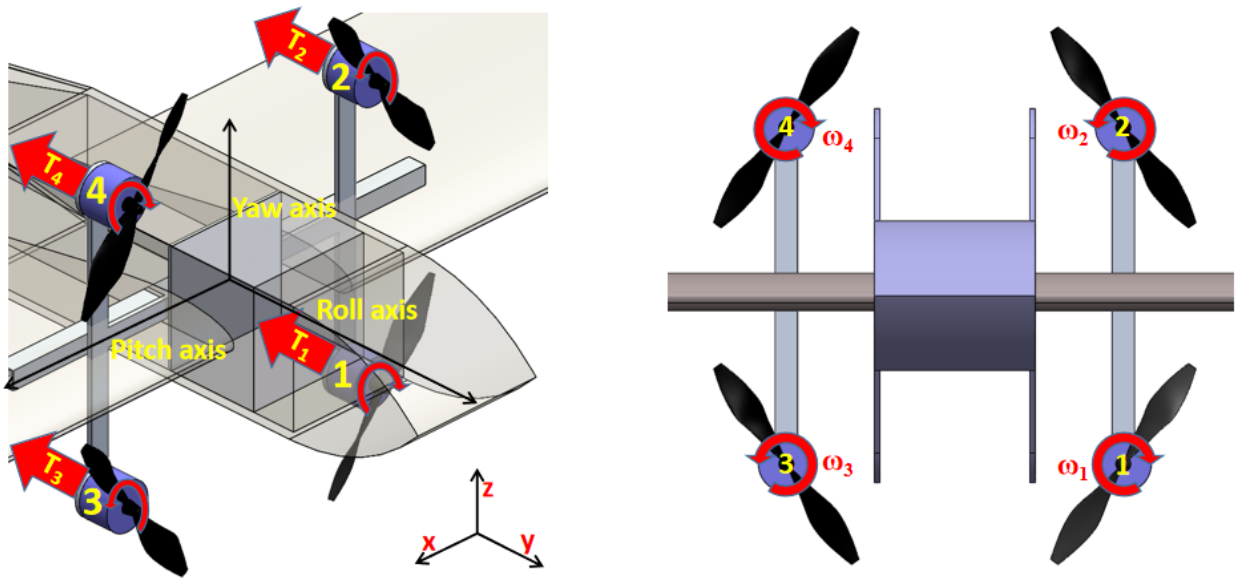

Figure 2.3: Configuration of motors, its direction of rotation and orientation of axis for gliding mode

2.2.2.1 Lift and drag forces on the wings:

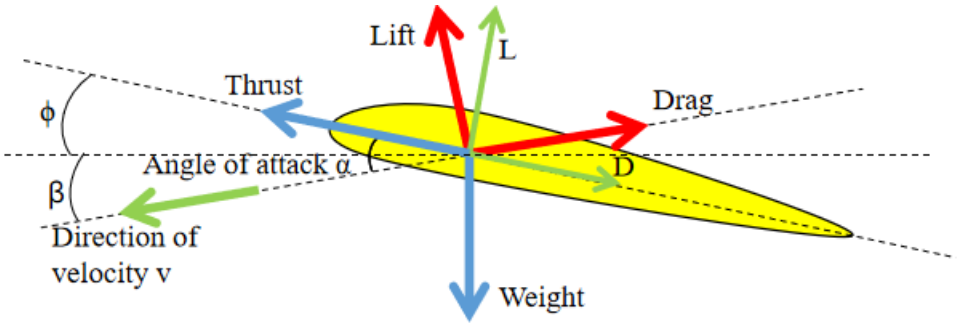

Figure 2.4: Free body diagram of wing

The force generated in the direction of Yaw axis in body fixed frame can be given as follows.

$$L = \frac{1}{2}\rho A v^2 \left(C_l(\alpha) * \cos\alpha + C_d(\alpha) * \sin\alpha\right) \quad (11)$$

And, the force generated in the direction of Roll axis in body fixed frame can be given as follows.

$$D = \frac{1}{2}\rho A v^2 \left(C_d(\alpha) * \cos\alpha - C_l(\alpha) * \sin\alpha\right) \quad (12)$$

Moment generated due to horizontal stabilizer.

$$M_h = -L_h r_h \tag{13}$$

Moment generated due to vertical stabilizer.

$$M_v = -L_v r_v \tag{14}$$

Moment generated on pitch axis due to eccentricity between the center of gravity (CG) and center of lift (CL).

$$M_g = -mgr_g \sin(\phi + 90) \tag{15}$$

2.2.2.2 For cruising at constant velocity:

$$T_1 + T_2 + T_3 + T_4 = D$$

$$L = mg$$

Moment on Pitch axis = $((T_1 + T_3) - (T_2 + T_4))*L_1 + M_g = 0$

2.2.2.3 For motion in only y direction:

$$T_1 + T_2 + T_3 + T_4 - D = m * a_y$$

$$L = mg$$

2.2.2.4 For pitching motion:

$$(T_1 + T_2 + T_3 + T_4 - D) * \cos\phi - L * \sin\phi = m * a_y$$

$$(T_1 + T_2 + T_3 + T_4 - D) * \sin\phi + L * \cos\phi - mg = m * a_z$$

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{pitch}\begin{bmatrix} 0 \\ T_1 + T_2 + T_3 + T_4 - D \\ L \end{bmatrix} \tag{16}$$

$$I_{pitch} * \ddot{\phi} = ((T_1 + T_3) - (T_2 + T_4)) * L_1 + M_h + M_g \tag{17}$$

2.2.2.5 For rolling motion:

$$T_1 + T_2 + T_3 + T_4 = D$$

$$L * \cos\theta - mg = m * a_z$$

$$L * \sin\theta = m * a_x$$

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{roll}\begin{bmatrix} 0 \\ T_1+T_2+T_3+T_4-D \\ L \end{bmatrix} \quad (18)$$

$$(\omega_1+\omega_2+\omega_3+\omega_4)*\mathrm{I_m} = -\omega_{roll} * I_{roll}$$

$$I_{roll} * \ddot{\theta} = -(\mathrm{M_1}+\mathrm{M_2}+\mathrm{M_3}+\mathrm{M_4}) \quad (19)$$

2.2.2.6 For Yaw motion:

$$(\mathrm{T_1}+\mathrm{T_2}+\mathrm{T_3}+\mathrm{T_4}-\mathrm{D})*\sin\psi = -\mathrm{m}*\mathrm{a_x}$$

$$(\mathrm{T_1}+\mathrm{T_2}+\mathrm{T_3}+\mathrm{T_4}-\mathrm{D})*\cos\psi = \mathrm{m}*\mathrm{a_y}$$

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_{yaw}\begin{bmatrix} 0 \\ T_1+T_2+T_3+T_4-D \\ L \end{bmatrix} \quad (20)$$

$$I_{yaw} * \ddot{\psi} = ((\mathrm{T_3}+\mathrm{T_4})-(\mathrm{T_1}+\mathrm{T_2}))*\mathrm{L_2}+M_v \quad (21)$$

2.2.2.6 General dynamic model:

For linear accelerations:

Total rotation of the drone can be defined as

$$\mathrm{R} = \mathrm{R_{yaw}} * \mathrm{R_{pitch}} * \mathrm{R_{roll}}$$

$$R = \begin{bmatrix} C\psi C\theta - S\phi S\psi S\theta & -C\phi S\psi & C\psi S\theta + C\theta S\phi S\psi \\ C\theta S\psi + C\psi S\phi S\theta & C\phi C\psi & S\psi S\theta - C\psi C\theta S\phi \\ -C\phi S\theta & S\phi & C\phi C\theta \end{bmatrix} \quad (22)$$

General dynamic equation of drone for linear accelerations in gliding mode can be described as follows.

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R\begin{bmatrix} 0 \\ T_1+T_2+T_3+T_4-D \\ L \end{bmatrix} \quad (23)$$

For angular accelerations:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} C\theta & 0 & -C\phi S\theta \\ 0 & 1 & S\phi \\ S\theta & 0 & C\phi C\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{24}$$

General dynamic equation of drone for angular accelerations in gliding mode can be described as follows.

$$I\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} ((T_1 + T_3) - (T_2 + T_4)) * L_1 + M_h + M_g \\ -(M_1 + M_2 + M_3 + M_4) \\ ((T_3 + T_4) - (T_1 + T_2)) * L_2 + M_v \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{25}$$

2.3 Prototyping and Testing

The design of the drone presented in this project was manufactured and tested successfully. The manufacturing of the drone was based on a quadcopter. The components used were selected from a racing quadcopter of similar weight.

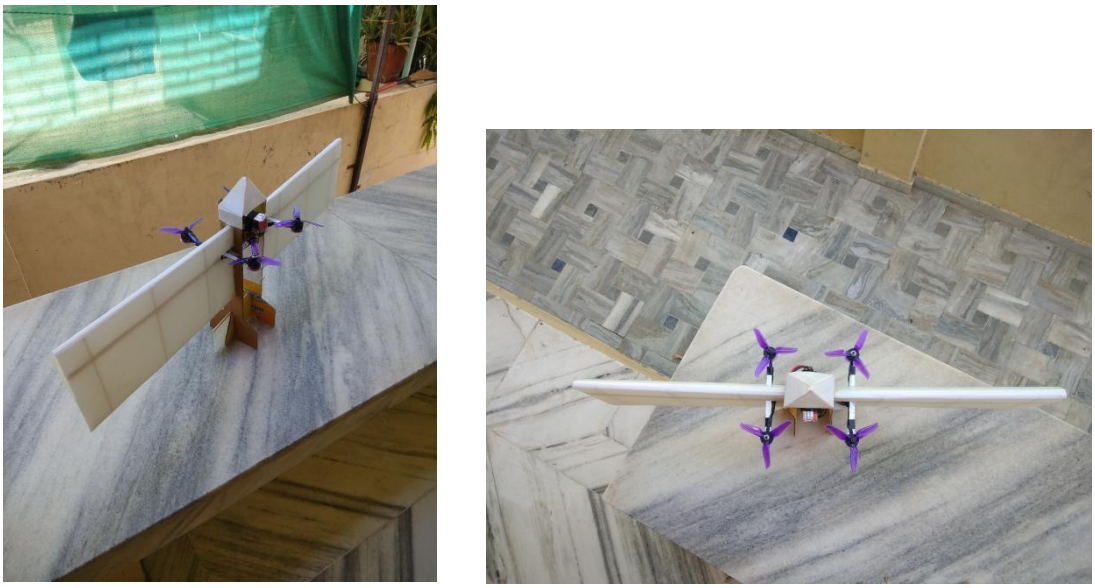

Figure 2.5: VTOL drone Prototype

The drone was tested in both hovering and gliding modes. During hovering mode, drone was successfully able to maneuver on all the axis as expected. The test flight of gliding mode was very important for this project as the concept of maneuvering was totally new. During the test flight in gliding mode, drone performed very well. The sensitivity of the controls was even better than an airplane.

Figure 2.6: VTOL drone Testing

CHAPTER 3

Literature Review (Drone Automation)

Many ongoing researches are working on complete automation of the flight. Many consumer drones mainly used for aerial photo and videography uses some level of automation in order to ease the controls of the drone, avoiding hitting obstacles in the path, automatic return to home or to track and follow an object. Such automation requires processing of various information at different level in the architecture of the drone. Processing the information in one level will provide boundary conditions for other level and so on. The upcoming sub sections will describe the flow of information and the processing work done by each level in the drone architecture.

3.1 General flow of information

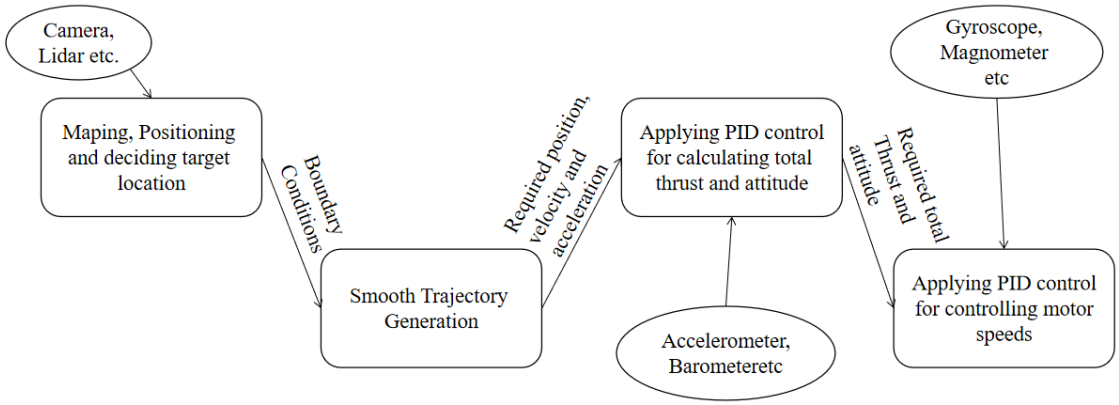

Figure 3.1: Flow Chart of Information

The figure 3.1 shows the flow chart of the information in the drone. The flow of information takes place from higher level to lower level.

The highest level in the drone architecture takes input form sensors like cameras and LiDAR and generates a crude map of the surrounding. It then locates itself in that map by implementing various mapping and positioning algorithms. It also decides based on the information available, the target positions, target velocities and target accelerations which serve as the required boundary conditions for the lower level, the trajectory generator.

In this level, it receives input in form of boundary conditions and based on the type of trajectory to be generated, it generates a set of equations as a function of time by simultaneously solving different matrices. These equations provides required position, velocity and acceleration at every point of time to the lower level which is a PID controller.

This level is a PID controller which constantly tries to reduce the errors in position, velocity and acceleration with respect to the trajectory provided by the trajectory generator. By taking the input of required position, velocity and acceleration from the higher level and of current value of position, velocity and acceleration from accelerometer and gyroscope, it calculates and give required thrust and attitude to reduce the error in the position, velocity and acceleration. This values of thrust and attitude becomes the required boundary conditions for other PID controller which finally controls the speed of the motors.

This is the lowest level of control in the quadcopter. It actively controls the RPM of four motors in order to reduce the errors in thrust and attitude. It takes input of required thrust and attitude from the higher level and current attitude from gyroscope and magnetometer. It constantly vary the RPM of the four motors in order to attain the required attitude and thrust.

Now let us analyse the whole control process from top to bottom. A quadcopter using cameras and/or LiDAR maps the surrounding and positions itself. It has some command of direction to follow and based on such commands it decides couple of points in the space from where it must go to reach the destination safely. Now using these points the trajectory generator generates a smooth trajectory that passes through all the given points. This trajectory as a function of time provides the desired values of position, velocity and acceleration at every time to the PID controller. Using this values and the current values of position, velocity and acceleration from various sensors, it constantly ties to minimize the error by updating the value of thrust and attitude at every point of time. As the required values keep constantly updating, the quadcopter keeps following the updated values and so keeps tracking the trajectory. Using this values of required thrust and attitude, and the current values obtained by the sensors, it constantly controls the speed of the four motors and hence the quadcopter reaches the target destination autonomously.

3.2 PID Controller

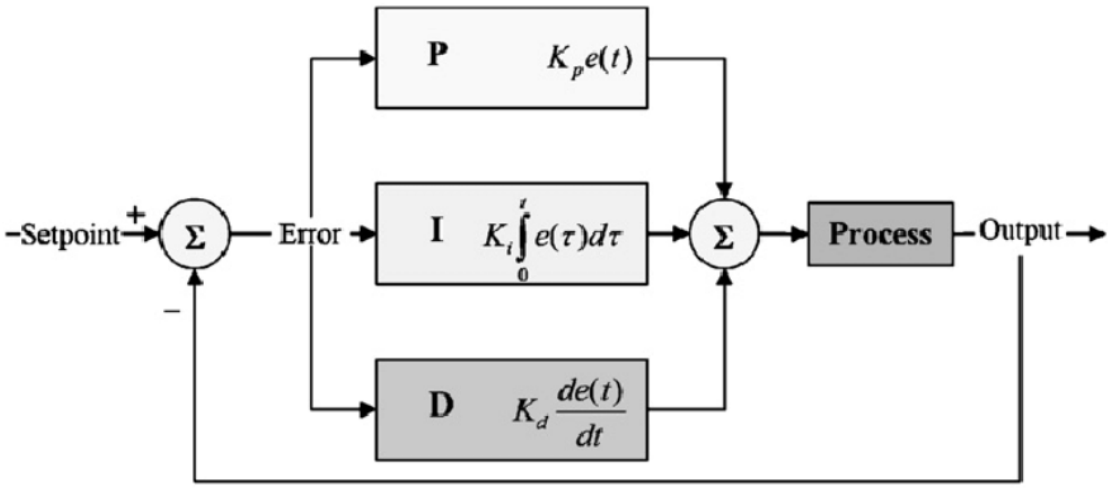

Figure 3.2: PID Control loop

The two lower most stages of the quadcopter control are PID controllers. PID stands for Proportional-Integral-Derivative control. It continuously tries to minimize the error such that the oscillations in the system is close to zero. PID controller is a closed loop system. For PID control to work, it constantly needs feedback from various sensors. PID for a system are calibrated to minimize the error in least possible time and with no oscillations. General PID equation is shown below.

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{d(e(t))}{dt} \tag{26}$$

In the above equation, u(t) is the PID controlled variable. e(t) denotes the error to be minimized. K_p is called proportional gain. It controls the value of controlled variable directly proportional to the error value. K_d is termed as derivative gain. It controls the value of controlled variable based on the rate of change of the error. It normally acts as the brakes in the equation. Without the derivative gain, the system will oscillate infinitely. K_i is termed as integral gain. Its valve is normally negligible but if the value of error remains constant with the time, its value increases until the error approaches zero. So its value increase with error and the effective time of the error.

The combination of these three parts of the PID equation gives an equation which actively reduces the error in the least possible time, with no oscillations and no steady state error.

PID control is the best and fairly simple option to control the quadcopter. For automation two PID control loops are required. In the first control loop, there are total three PID equations with control variable be thrust, roll and pitch respectively. This values is then fed to the last level which tries to minimize the error with respect to these values. This PID loop has total four equations each gives values of four control variables. These control variables correspond to four equations with four variables. Solving these equations, we get four values which are the required RPM of each motors.

3.3 Trajectory Generator

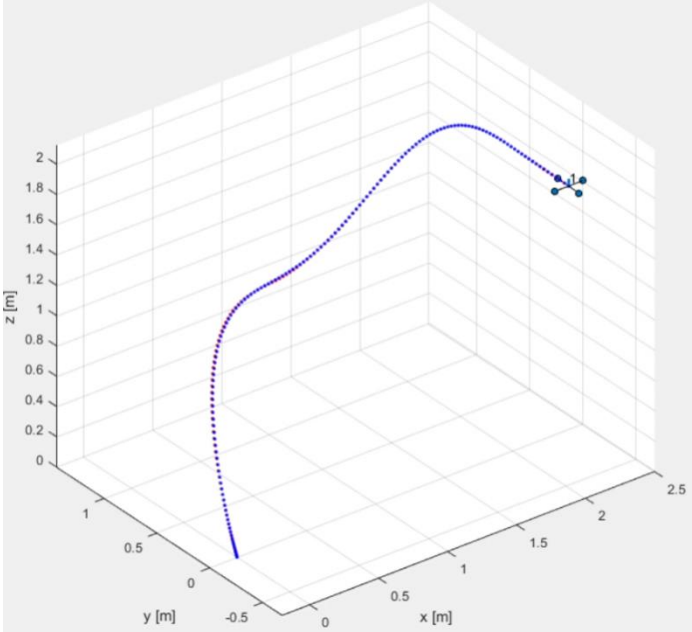

Figure 3.3: Trajectory generation Simulation

Trajectory is a function of time defined such that it passes through a set of points. Trajectory provides information various parameters defined in the equation at a particular time. A trajectory must pass through every point provided and at any point it should not provides undefined or infinite value of the set parameters. In a possible set of trajectories

between, the trajectory with minimum rate of change of parameters is termed as smooth trajectory. The smooth trajectory of order 1 must follow the following condition.

$$x^*(t) = \arg\min_{x(t)} \int_0^T L(\dot{x}, x, t)dt \tag{27}$$

The objective is to find the best x(t) to minimize the function L. The necessary condition to satisfy the optimal function x(t) in order to minimize the function L can be derived by Eular-Lagrange Equation. Following equation is the Eular-Lagrange equation to minimize the above mentioned function L of order 1.

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0 \tag{28}$$

In order to generate smooth trajectory of n^{th} order, function L is normally defined as follows. $x^{(n)}$ is termed as n^{th} derivative of x.

$$L(x^{(n)}, x^{(n-1)}, x^{(n-2)} \ldots\ldots \dot{x}, x, t) = \left(x^{(n)}\right)^2 \tag{29}$$

$$x^*(t) = \arg\min_{x(t)} \int_0^T \left(x^{(n)}\right)^2 dt \tag{30}$$

Eular Lagrange equation to minimize the above mentioned argument.

$$\frac{\partial L}{\partial x} - \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) + \frac{d^2}{dt^2}\left(\frac{\partial L}{\partial \ddot{x}}\right) - \ldots\ldots\ldots + (-1)^n \frac{d^n}{dt^n}\left(\frac{\partial L}{\partial x^{(n)}}\right) = 0 \tag{31}$$

Using the method discussed above we can find smooth trajectory of any order. Based on there order, trajectories can be classified as follows:

- n = 1, Minimum Velocity trajectory (Shortest distance trajectory)

- n = 2, Minimum Acceleration trajectory
- n = 3, Minimum Jerk trajectory
- n = 4, Minimum Snap trajectory

General equations of different trajectories:

- Minimum Velocity: $x = c_1t + c_0$

  2 boundary conditions: 1) initial position 2) final position

- Minimum Acceleration: $x = c_3t^3 + c_2t^2 + c_1t + c_0$

  4 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity

- Minimum Jerk: $x = c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0$

  6 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity 5) initial acceleration 6) final acceleration

- Minimum Snap: $x = c_7t^7 + c_6t^6 + c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0$

  8 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity 5) initial acceleration 6) final acceleration 7) initial jerk 8) final jerk

Trajectory generator uses a predefined algorithm in order to generate a smooth trajectory. In order to derive the equation using the algorithm, boundary conditions must be provided. As the order of the trajectory increases, the requirement of the boundary conditions increases and so the complexity of the matrices to be solved increases. The following subsections corresponds to the topmost stage of the drone automation in which the drone maps the environment and determines its current position and destination position. Visual odometry is the most common approach. We will discuss different algorithms used to perform this task.

3.4 Epipolar Geometry and camera calibration

3.4.1 Pin Hole Camera Model

A simple pin hole camera model is the most common model used for image processing and camera calibration. It defines the mathematical relationship between the co-ordinates of the object in the real world against the co-ordinates of the object in the image[16]. The formulation of the model can be seen below with reference to Figure 3.4.

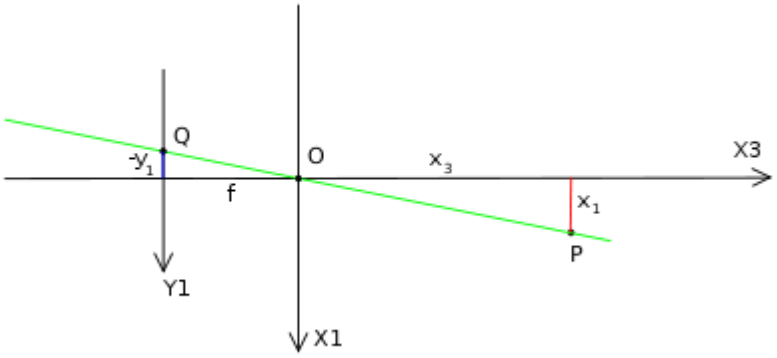

Figure 3.4: Pin Hole Camera Model

$$y1 = \frac{f \times x_1}{x_3} \tag{32}$$

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{33}$$

Here, ‘u’ and ‘v’ are the co-ordinates of the object point in the projection plane, while ‘X’, ‘Y’, ‘Z’ are the real world co-ordinates. f_x and f_y represent the focal length along x and y axis respectively, and c_x and c_y are the projection center. And finally r_{11} to r_{33} and t_1, t_2, t_3 are the elements of rotational and translational matrix.

3.4.2 Camera Calibration Matrix

The lens of the camera is not perfect and thus the image formed will have some radial and tangential distortion[15][16]. The formulation for calculating the undistorted image is given below, and the code for calibrating a camera with help of 8×8 chessboard can be referred from appendix. Here, ‘R’ and ‘t’ represent the rotational and translational matrix. And k_1 to k_6, and p_1, p_2 are the calibration parameters which are to be found.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{34}$$

$$x' = {}^{x}/_{z}$$

$$y' = {}^{y}/_{z}$$

$$r^2 = x'^2 + y'^2$$

$$x'' = x' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1x'y' + p_2(r^2 + 2x'^2) \quad (35)$$

$$y'' = y' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_2x'y' + p_1(r^2 + 2y'^2) \quad (36)$$

$$u = f_x \times x'' + c_x \quad (37)$$

$$v = f_y \times y'' + c_y \quad (38)$$

Figure 3.5: Camera Distortion

3.4.3 Epipolar Geometry

Considering to camera projection plane oriented as shown in Figure 3.6. An epipole is defined as a point on the projection plane from where the line connecting two center of projection passes[15][16]. A plane constructed by the epipoles and the point 'p', we get epiplane. This simple geometry shows that projection of point 'p' must lie on a single line called epipolar line. The fact that a points projection can be found on corresponding epilines helps us to reduce our search for matching pixel as explained in further sections.

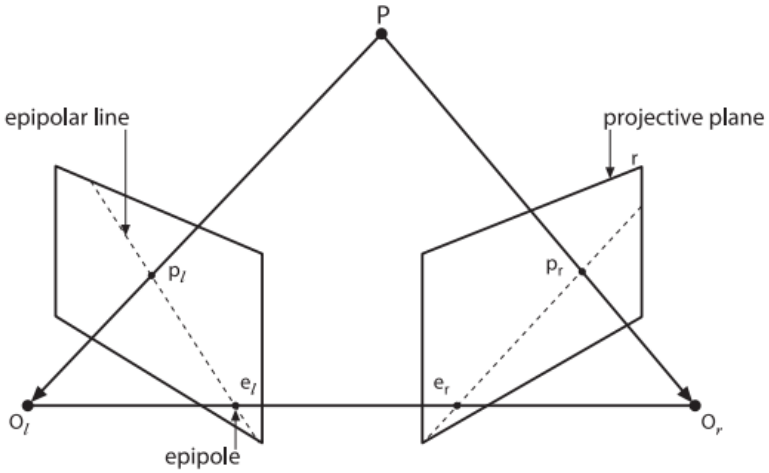

Figure 3.6: Epipolar Geometry

3.4.4 Stereo Calibration

3.5 Stereo Vision

Stereovision techniques uses two cameras to view the same object and are ideally oriented as shown in the Figure 3.6. However, there will always be some variation and thus a stereo-calibration using the epipolar geometry as discussed above gets necessary. The difference in the position of the pixel helps us to sense the depth in the image taken by the cameras. This difference is mapped for each pixel and forms a disparity map and further triangulation is used to estimate the actual depth.

3.5.1 Disparity Map

Disparity map is created by calculating difference in the location of the identical pixels in the left and the right image along the epipolar lines. The map shows unscaled depth of the objects with respect to the position of the camera[15][16]. And to get a scaled value we need to triangulate the position of the object. In the Figure 3.7 brighter points are nearer to the camera and darker points are further away. The code for constructing disparity map can be referred from the appendix.

Figure 3.7: Disparity Map

3.5.2 Triangulation

Triangulation is the process of getting the real co-ordinates of the object from two images focused on the same object[15][16]. Its should be kept in mind that before triangulation camera calibration and stereo-calibration are must to ensure accurate results. The formulation for calculating depth is given below:

$$\frac{T-(x^l-x^r)}{Z-f} = \frac{T}{Z} \Rightarrow \mathrm{Z} = \frac{fT}{(x^l-x^r)} \tag{39}$$

Here, 'Z' is the distance of the object from the projection plane. And 'T' is the distance of between the two cameras, While x_l and x_r are the pixel locations in the left and right images respectively.

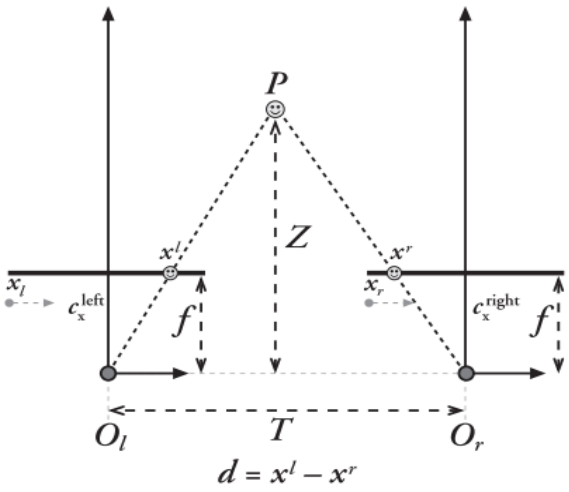

Figure 3.8: Triangulation

3.6 Template Matching

Template matching is process of finding an object in the image by scanning the image with a fixed sized window. The window is generally of the same size as template which is to be found. The match is found by calculating similarities in intensity values of the pixels between the template and the match window[15][16]. The formula below calculates the squared difference in the intensity of the pixel in template at a given location to the intensity of the pixel in image.

$$R(x,y) = \sum x',y'(T(x',y') - I(x+x',y+y'))^2 \quad (40)$$

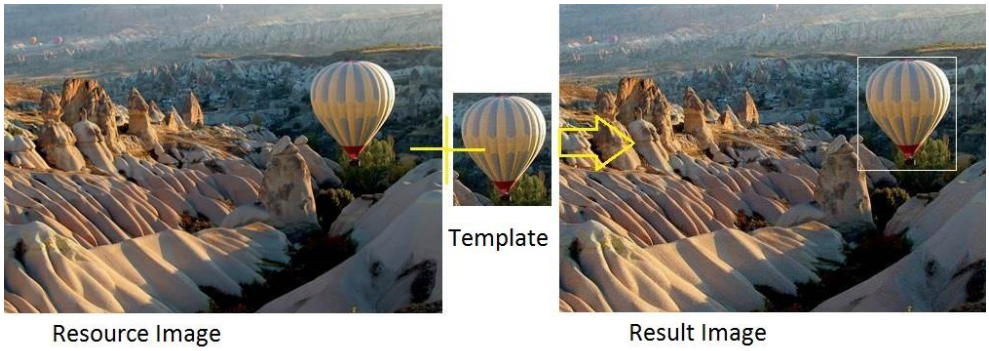

Figure 3.9: Template Matching

CHAPTER 4

PID Equations generation

Major step in drone automation is PID equation generation. Two of four stages in the drone architecture are PID controllers. Proper PID equations and its effective implementation are very important for automation of drone. The stability of drone flight totally depends on the PID calibration. In our case there are two PID controlled variable to be stabilized, u_1 and u_2. u_1 is the summation of the thrust force from individual motor. Where as u_2 is a 3x1 matrix which provides the moment on each axis. The feedback closed loop of the lower most two stages is shown in following image.

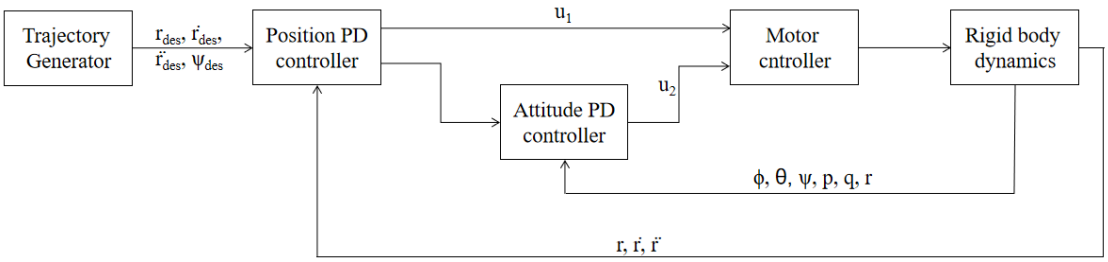

Figure 4.1: Closed control loops of the drone PID controllers

To generate the PID equations, we need the dynamic equations of the drone which was derived in our previous project. Following are the dynamic equations in hovering mode.

$$m\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R\begin{bmatrix} 0 \\ 0 \\ T_1+T_2+T_3+T_4 \end{bmatrix} \Bigg\} u_1 \tag{41}$$

$$I\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \underbrace{\begin{bmatrix} ((T_1+T_3)-(T_2+T_4))*L_1 \\ ((T_1+T_2)-(T_3+T_4))*L_2 \\ -(M_1+M_2+M_3+M_4) \end{bmatrix}}_{u_2} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I\begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{42}$$

For simplicity, the integral variable is not considered in the upcoming calculations. General form of PD equation can be stated as below.

$$(\ddot{r}_{des} - \ddot{r}) + Kd(\dot{r}_{des} - \dot{r}) + Kp(r_{des} - r) = 0 \tag{43}$$

PD equations for acceleration in x, y and z direction.

$$\ddot{z} = \ddot{z}_{des} + Kd(\dot{z}_{des} - \dot{z}) + Kp(z_{des} - z) \tag{44}$$

$$\ddot{x} = \ddot{x}_{des} + Kd(\dot{x}_{des} - \dot{x}) + Kp(x_{des} - x) \tag{45}$$

$$\ddot{y} = \ddot{y}_{des} + Kd(\dot{y}_{des} - \dot{y}) + Kp(y_{des} - y) \tag{46}$$

From the equation (41) an (44) (considering $a_z = \ddot{z}$ and angles on all the axis almost zero) we get following equation for u_1.

$$u_1 = m(g + \ddot{z}) \tag{47}$$

Considering acceleration in x and y directions and its required yaw position, we can find required pitch and roll angles.

$$\phi_{des} = (\ddot{x}\sin\psi_{des} - \ddot{y}\cos\psi_{des})/g \tag{48}$$

$$\theta_{des} = (\ddot{x}\cos\psi_{des} + \ddot{y}\sin\psi_{des})/g \tag{49}$$

Knowing destination roll, pitch and yaw angles, we can apply PD equation on it to calculate angular accelerations in roll, pitch and yaw axis.

$$\alpha_{Pitch} = Kp_{Pitch}(\phi_{des} - \phi) + Kd_{Pitch}(0 - p) \tag{50}$$

$$\alpha_{Roll} = Kp_{Roll}(\theta_{des} - \theta) + Kd_{Roll}(0 - q) \tag{51}$$

$$\alpha_{Yaw} = Kp_{Yaw}(\psi_{des} - \psi) + Kd_{Yaw}(0 - r) \tag{52}$$

Where p, q and r are angular velocity with respect to environmental fixed frame (i.e. X, y and z axis) and its relation with angular velocities on roll, pitch and yaw axis can be given by equation (9).

Considering equations (42), (50), (51), (52) and the inertia tensor I, u_2 can be stated as follows.

$$u_2 = I * \begin{bmatrix} Kp_{Pitch}(\phi_{des} - \phi) + Kd_{Pitch}(0 - p) \\ Kp_{Roll}(\theta_{des} - \theta) + Kd_{Roll}(0 - q) \\ Kp_{Yaw}(\psi_{des} - \psi) + Kd_{Yaw}(0 - r) \end{bmatrix} \quad (53)$$

u_1 and u_2 provides the required thrust and moment for the drone to maintain its position, velocity and acceleration according to the trajectory provided. So in order to calculate the required rpm of each motor, we have to solve extra equations that will provides us the required rpm of four motors. These equations are generated using the dynamic model of the drone and the relationship between the thrust and angular velocity of the motor propeller assembly.

To establish the relationship between the thrust, angular velocity and the motor torque, we need to consider the lift and drag forces on the propeller.

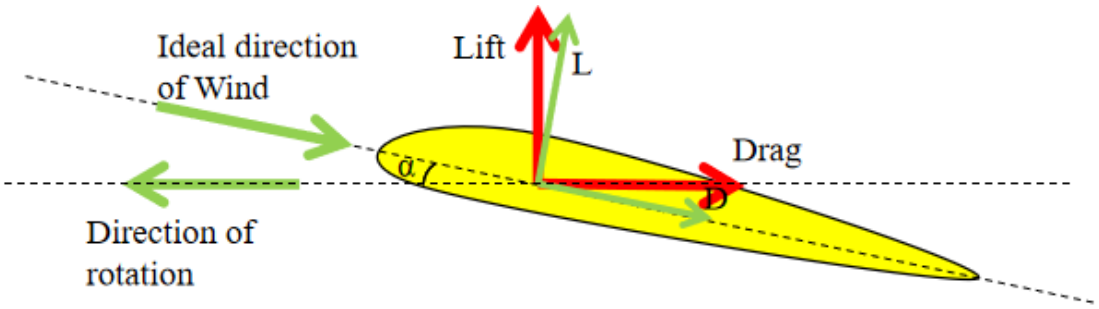

Figure 4.2: Lift and drag forces on a propeller blade

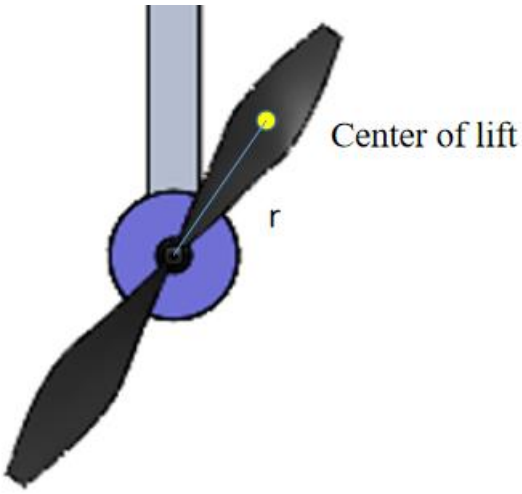

Figure 4.3: Center of lift and r

Figures 4.2 shows the lift and drag forces on a cross section of propeller blade and the figure 4.3 shows the center of lift and its distance from the center of rotation.

Let,

L and D be lift and drag forces on the propeller blade.

Lift and Drag be the forces perpendicular and parallel to the plane of rotation.

T be the thrust of the propeller.

α be the angle of attack of the propeller blade.

n be the number of blades on the propeller.

r be the distance between center of propeller and the center of lift of the blade.

A be the area of the wing.

ρ be the density of air.

v be the tangential velocity of the center of lift.

ω be the angular velocity of the propeller.

L and D generated on the propeller wing can be stated as follows.

$$L = \frac{1}{2}\rho A v^2 * C_l \qquad D = \frac{1}{2}\rho A v^2 * C_d \tag{54}$$

Now, v = ω*r

$$L = \frac{1}{2}\rho A\omega^2 r^2 * C_l \qquad D = \frac{1}{2}\rho A\omega^2 r^2 * C_d \tag{55}$$

Lift and drag forces on the propeller in total can be stated as follows.

$$T = \frac{1}{2}n\rho A\omega^2 r^2 \left(C_l * \cos\alpha - C_d * \sin\alpha\right) \tag{56}$$

$$Drag = \frac{1}{2}\rho A\omega^2 r^2 \left(C_l * \sin\alpha + C_d * \cos\alpha\right) \tag{57}$$

Torque = Fr*sinθ

θ in case of propeller is 90º.

There is no mathematical difference between moment and torque but there is a basic difference between them in definition. But for mathematical modelling purpose torque can be considered as same as moment so we can relate the equation with the dynamic model of the drone.

So the moment on the motor can be stated as follows.

$$M = \frac{1}{2}n\rho A\omega^2 r^3 \left(C_l * \sin\alpha + C_d * \cos\alpha\right) \tag{58}$$

From equations (56) and (58), we can consider that thrust and moment of the motors are proportional to the square of angular velocity. Every thing else are constants.

$T \alpha \omega^2$ and $M \alpha \omega^2$

Therefore, T is directly proportional to M

$$M = K * T \tag{59}$$

We have the values of u_1 and u_2 from the equations (47) and (53). And from the dynamic model ((41) and (42)) we have the values of the following equations.

$$
\begin{aligned}
&T_1 + T_2 + T_3 + T_4 = u_1 \\
&T_1 - T_2 + T_3 - T_4 = u_2(1,1)/L_1 \\
&T_1 + T_2 - T_3 - T_4 = u_2(2,1)/L_2 \\
&M_1 - M_2 - M_3 + M_4 = -u_2(3,1)
\end{aligned}
\tag{60}
$$

Note: In the above equation of moment, the moments are considered always positive and so that effect of different directions is compensated by adding necessary negative signs in the equation.

Considering the relation between M and T from equation (59), we get following four equations for thrust.

$$
\begin{aligned}
&T_1 + T_2 + T_3 + T_4 = u_1 \\
&T_1 - T_2 + T_3 - T_4 = u_2(1,1)/L_1 \\
&T_1 + T_2 - T_3 - T_4 = u_2(2,1)/L_2 \\
&T_1 - T_2 - T_3 + T_4 = -u_2(3,1) * K
\end{aligned}
\tag{61}
$$

The values of the individual thrust forces can be calculated by solving following matrix equation.

$$
\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2(1,1)/L_1 \\ u_2(2,1)/L_2 \\ u_2(3,1) * K \end{bmatrix}
\tag{62}
$$

Now using equation (56) we can find the corresponding angular velocity for the motor and supply it to corresponding ESC and thus the motor speed is controlled.

The method shown above is derived mathematically in order to find the motor speed. We have assumed that the conditions will always be ideal. But in reality, there are many variations in the system. Even the coefficient of lift and drag changes with the propeller

speed. Ironically, it is observed in some research that for a certain range of propeller speed, the angular velocity is in fact directly proportional to the thrust and the moment. Plagiarism

So to reduce the complexity, the actual control of the speed is handed directly to the bottom most PID controller. So u_1 and u_2, instead of varying thrust, directly varies the angular velocities of the motors. The variations in the system is handled by the PID equation itself and hence the complexities of the mathematical calculations in the code are reduced drastically.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \begin{bmatrix} K(\ddot{z}_{des} + g) + Kd(\dot{z}_{des} - \dot{z}) + Kp(z_{des} - z) \\ Kp_{Pitch}(\phi_{des} - \phi) + Kd_{Pitch}(0 - p) \\ Kp_{Roll}(\theta_{des} - \theta) + Kd_{Roll}(0 - q) \\ Kp_{Yaw}(\psi_{des} - \psi) + Kd_{Yaw}(0 - r) \end{bmatrix} \quad (63)$$

Equation (63) is the simplified version of the PID equations derived above. All the constants and system variables are taken care by the PID constants itself. Now to handle the drone smoothly, these PID constants must be calibrated. This is the first stage of the drone automation. Next stage is the automated trajectory generation which is discussed in next chapter.

CHAPTER 5

Automated Trajectory Generation

Another major stage in the drone automation is automated trajectory generation. By developing the PID equations, we can make the quadcopter follow a predefined trajectory. But in order to go from point A to point B through some intermediate points, the flight computer must be capable of generating a smooth trajectory through that points. A smooth trajectory provide the values of destination coordinates (position, velocity, acceleration, jerk etc.) at every point of time. The number of outputs the trajectory provides depends on the degree of the trajectory.

We have earlier discussed the conditioned to be satisfied in order to generate smooth trajectory and we have derived general equations of smooth trajectories for 1, 2, 3 and 4 degrees using Eular-Lagrange equation. In this chapter we will further discuss the algorithm used to derive an exact equation of the trajectory between two points and also through more than two points. Following shows the general equations of different trajectories and the boundary conditions required in order to define them.

- Minimum Velocity: $x = c_1t + c_0$

  2 boundary conditions: 1) initial position 2) final position

- Minimum Acceleration: $x = c_3t^3 + c_2t^2 + c_1t + c_0$

  4 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity

- Minimum Jerk: $x = c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0$

  6 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity 5) initial acceleration 6) final acceleration

- Minimum Snap: $x = c_7t^7 + c_6t^6 + c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0$

8 boundary conditions: 1) initial position 2) final position 3) initial velocity 4) final velocity 5) initial acceleration 6) final acceleration 7) initial jerk 8) final jerk

In minimum velocity trajectory, the value of acceleration does not changes and in minimum acceleration trajectory, the change in acceleration remains constant. So this leads to some points where the values of output goes out of the defined range and thus these two trajectories are not preferred. For a smooth quadcopter flight, minimum jerk and minimum snap trajectories are preferred. Now we will discuss the algorithms to derive these two trajectories between two points.

5.1 Minimum Jerk trajectory

In minimum jerk trajectory, initial and final jerk are not taken into consideration, and it leads to minimum jerk through out the trajectory. So it is called minimum jerk trajectory. Following is the general equation for smooth minimum jerk trajectory derived using Eular-Lagrange equation.

$$x = c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0 \quad (64)$$

Now, differentiating equation (64) with respect to time, we get equation for velocity with respect to time.

$$\dot{x} = 5c_5 t^4 + 4c_4 t^3 + 3c_3 t^2 + 2c_2 t + c_1 \quad \text{w} \quad (65)$$

Now, differentiating equation (65) with respect to time, we get equation for acceleration with respect to time.

$$\ddot{x} = 20c_5 t^3 + 12c_4 t^2 + 6c_3 t + 2c_2 \quad (66)$$

Now, in order to find constants c_5, c_4, c_3, c_2, c_1 and c_0, we need six boundary conditions. The boundary conditions we know are initial and final positions, initial and final velocities, and initial and final accelerations.

Let,

At initial condition (i.e. at t = 0),
position = x_i, velocity = v_i and acceleration = a_i

At final condition (i.e. at t = T),
position = x_f, velocity = v_f and acceleration = a_f

Considering, above boundary conditions, we get 6 equations as follows.

$$
\begin{aligned}
&c_0 = x_i \\
&c_5T^5 + c_4T^4 + c_3T^3 + c_2T^2 + c_1T + c_0 = x_f \\
&c_1 = v_i \\
&5c_5T^4 + 4c_4T^3 + 3c_3T^2 + 2c_2T + c_1 = v_f \\
&2c_2 = a_i \\
&20c_5T^3 + 12c_4T^2 + 6c_3T + 2c_2 = a_f
\end{aligned}
\tag{67}
$$

These equations can be easily solved by solving the following matrix equation.

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 \\
T^5 & T^4 & T^3 & T^2 & T & 1 \\
0 & 0 & 0 & 0 & 1 & 0 \\
5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\
0 & 0 & 0 & 2 & 0 & 0 \\
20T^3 & 12T^2 & 6T & 2 & 0 & 0
\end{bmatrix}
\begin{bmatrix} c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}
=
\begin{bmatrix} x_i \\ x_f \\ v_i \\ v_f \\ a_i \\ a_f \end{bmatrix}
\tag{68}
$$

Solving the above matrix, we get the values of all six constants. Now by putting the values of these constants in the equations (64), (65) and (66), we get the actual equations for position, velocity and acceleration which will provide destination coordinates to the PID controller.

Now by solving the above matrix, we will get trajectory in only one of three directions. So in order to find 3-dimensional trajectory, we need to repeat the above mentioned algorithm three times. After this, we will end up with total nine equations (three in each direction) for a trajectory between the points in 3-D space.

5.2 Minimum Snap trajectory

Snap is defined as rate of change of jerk. In minimum snap trajectory, initial and final snap are not taken into consideration, and thus it leads to minimum snap throughout the trajectory. So it is called minimum snap trajectory. These type of trajectories are generally smoother and more efficient than the minimum jerk trajectory. These extra pros comes with its own cons. This trajectory is way more complicated to solve than minimum jerk trajectory and it also requires extra two boundary conditions to be defined. So, if the flight computer is able to handle the complexity of the matrix calculations, this trajectory is used. Following is the general equation for smooth minimum snap trajectory derived using Eular-Lagrange equation.

$$x = c_7 t^7 + c_6 t^6 + c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0 \tag{69}$$

Now, differentiating equation (69) with respect to time two times, we get equations for velocity and acceleration with respect to time.

$$\dot{x} = 7c_7 t^6 + 6c_6 t^5 + 5c_5 t^4 + 4c_4 t^3 + 3c_3 t^2 + 2c_2 t + c_1 \tag{70}$$

$$\ddot{x} = 42c_7 t^5 + 30c_6 t^4 + 20c_5 t^3 + 12c_4 t^2 + 6c_3 t + 2c_2 \tag{71}$$

Further differentiating equation (71), we get equation for jerk with respect to time.

$$\dddot{x} = 210c_7 t^4 + 120c_6 t^3 + 60c_5 t^2 + 24c_4 t + 6c_3 \tag{72}$$

Now, in order to find constants c_7, c_6, c_5, c_4, c_3, c_2, c_1 and c_0, we need eight boundary conditions. The boundary conditions we know are initial and final positions, initial and final velocities, initial and final accelerations, and initial and final jerk.

Let,

At initial condition (i.e. at t = 0),
position = x_i, velocity = v_i, acceleration = a_i and jerk = j_i

At final condition (i.e. at t = T),
position = x_f, velocity = v_f, acceleration = a_f and jerk = j_f

Considering, above boundary conditions, we get 8 equations as follows.

$$
\begin{aligned}
&c_0 = x_i \\
&c_7T^7 + c_6T^6 + c_5T^5 + c_4T^4 + c_3T^3 + c_2T^2 + c_1T + c_0 = x_f \\
&c_1 = v_i \\
&7c_7T^6 + 6c_6T^5 + 5c_5T^4 + 4c_4T^3 + 3c_3T^2 + 2c_2T + c_1 = v_f \\
&2c_2 = a_i \\
&42c_7T^5 + 30c_6T^4 + 20c_5T^3 + 12c_4T^2 + 6c_3T + 2c_2 = a_f \\
&6c_3 = j_i \\
&210c_7T^4 + 120c_6T^3 + 60c_5T^2 + 24c_4T + 6c_3 = j_f
\end{aligned}
\tag{73}
$$

These equations can be solved by solving the following matrix equation.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ T^7 & T^6 & T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 7T^6 & 6T^5 & 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 42T^5 & 30T^4 & 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 210T^4 & 120T^3 & 60T^2 & 24T & 6 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} x_i \\ x_f \\ v_i \\ v_f \\ a_i \\ a_f \\ j_i \\ j_f \end{bmatrix} \quad (74)$$

Solving the above matrix, we get the values of all eight constants. Now by putting the values of these constants in the equations (69), (70) and (71), we get the actual equations for position, velocity and acceleration which will provide destination coordinates to the PID controller.

Now by solving the above matrix, we will get trajectory in only one of three directions. So in order to find 3-dimensional trajectory, we need to repeat the above mentioned algorithm three times. After this, we will end up with total nine equations (three in each direction) for a trajectory between the points in 3-D space.

5.3 Trajectory generation through more than two points

Until now in this chapter we have discussed about trajectory generation between two points. But in real conditions, the trajectory must pass through more than two points. The most common approach for this is to split the trajectory among the points. So if there are n points through which the trajectory must pass, there will be (n-1) sections of the trajectory.

Each section of the trajectory can be treated as an individual trajectory. So for each section, we need to define six or eight boundary conditions depending on the trajectory. Normally we know the boundary conditions of the initial and final points. But for the intermediate points, we need to make valid assumptions to define boundary conditions. Though there is one fact that helps us making this process easy. The initial boundary conditions of n^{th} equation will be equivalent to the final boundary conditions of $(n - 1)^{th}$

equation. This fact reduces the number of boundary conditions to half. We also need to make assumption of the final time (T) of each trajectory part.

The approach we followed is to assume a constant velocity of the drone. Now we initially assume that the drone moves in straight line between two points at constant velocity. We know velocity and the distance between two points through which we can calculate final time (T) of the trajectory. Now to determine the velocity at final point, the value is considered as the constant we assumed in the direction of the final point of the next trajectory. The acceleration is normally considered zero at intermediate points. For minimum snap trajectory, jerk is also assumed zero.

So, for n number of points, there will be n-1 number of trajectories. Each trajectory will contain 3 set of equations for 3 dimensions and set will have 3 equations for position, velocity and acceleration respectively. So there will be total (n-1)*3*3 number of equations the trajectory generator need to derive. This becomes possible when we use nested for loops in the code. But the flight-computer must be capable enough to handle this many computations. If we implement minimum snap trajectory, the computations becomes even harder. The MATLAB code of the trajectory generator we wrote and simulated is stated in the Appendix.

CHAPTER 6

Visual Odometry

Visual Odometry refers to the process of estimating the current position of the system with respect to the initial position by means of Visual sensors such as RGB Cameras. Here, a simple webcam is used for this purpose. The process is divided into three parts :

1) Detection of features
2) Matching features
3) Calculation of relative change in position of Camera

6.1 ORB Feature detection

ORB stands for Oriented FAST and Rotated BRIEF, it is a fusion between the FAST features and the BRIEF features. For intuition, assume there is a pixel ‘p’ and 16 pixel surrounding pixel ‘p’, as shown in the Figure 6.1. Then the algorithm computes intensity for each of the pixel and if more than 8 pixel surrounding have intensity higher than ‘p’, pixel ‘p’ is registered as key point or a feature[18].

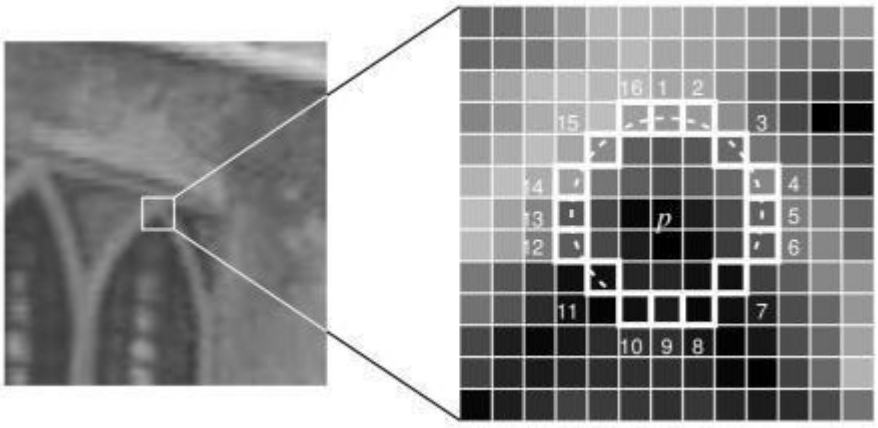

Figure 6.1: Key-point

ORB algorithm also finds such features in multiple scaled copies of the image as shown in Figure 6.2. The reason behind this is to make it scale invariant and thus is particularly useful in our case. As the object moves far away or get near its scale on the projection plane changes. Thus a algorithm which is scale invariant will give us very robust features for real time tracking[18].

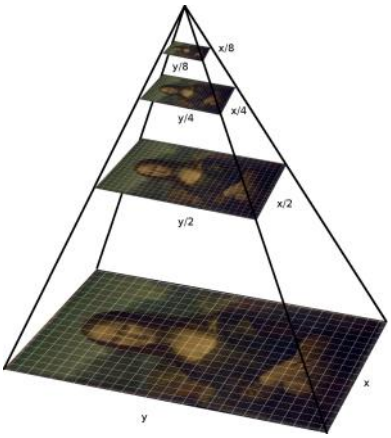

Figure 6.2: Scaled Images for finding features

6.2 Matching features in consecutive frames

Matching of the pixels between two frames is done with help of a brute force matcher based on Norm-Hamming Distance. Norma-Hamming distance is just counting the number of ones after applying XOR operation between two arrays. The search for the feature in the corresponding frame is done with the help of K-nearest neighbor algorithm[17]. And further a D.Lowe's ratio test is applied to further increase the accuracy of matches.

Figure 6.3: KNN Matcher result

6.3 Calculation of relative change in position of Camera

To calculate the unscaled relative position of camera with respect to object, we need to calculate the center of features. This is simply done by calculating average of value of position along x-axis and along y-axis. Here, we assume that the object is always at the center of the camera projection screen. Next we need to calculate the orientation of the object along the axis of camera and its distance from center of projection. The visual output of this algorithm is shown in the Figure 6.4.

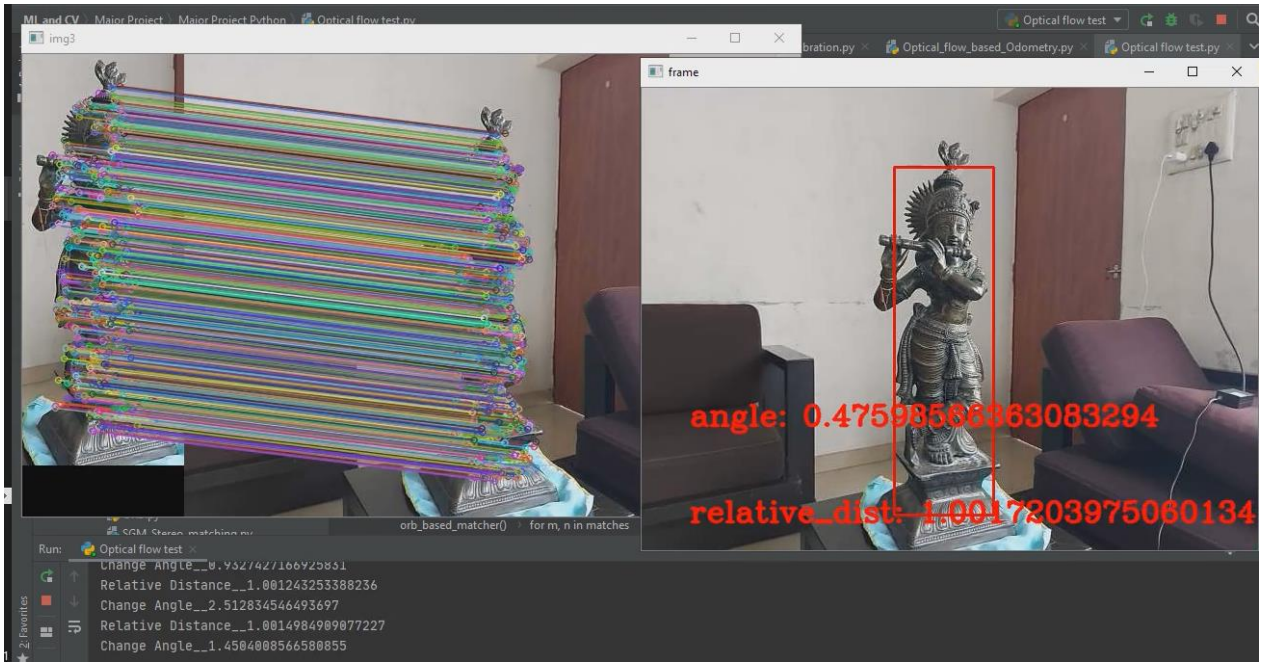

Figure 6.4: Visual output of algorithm

6.3.1 Orientation of camera with respect to object

To calculate change in angle of orientation with respect to initial frame we calculate the slope of every point with respect to the center of features in both initial and current frame. Then we subtract the initial slope from the current slope and divide by the number of feature points. Here, ‘xi’ and ‘yi’ are the pixel locations co-ordinates, and ‘n’ is the number of the features detected. ‘Cx’ and ‘Cy’ are the co-ordinates of the center. ‘θ’ is the angle of orientation.

$$C_x, C_y = \left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right) \tag{75}$$

$$S_I = \frac{\sum_{i=0}^{n} \frac{y_I - y_i}{x_I - x_i}}{n} \qquad S_c = \frac{\sum_{i=0}^{n} \frac{y_c - y_i}{x_c - x_i}}{n} \tag{76}$$

$$\Delta\theta = \mathrm{Tan}^{-1}(S_c) - \mathrm{Tan}^{-1}(S_I) \tag{77}$$

6.3.2 Unscaled distance

To get an intuition, consider a object with features as shown in the Figure 6.5. If distance between the features increase we can say that object is closure to the camera with respect to the initial position. And similarly if the distance between the feature decreases we can say that the object has moved further far away from the camera with respect to its initial position. But it should be kept in mind that the actual distance of the object from the center line of camera still remains unchanged. Now we can exploit the similar constraint to get relative distance or the ratio of the initial and current distance. Here, 'Y' is the distance of the object point in real world from the axis of the camera, and 'f' is the focal length. 'D' and 'D'' are the previous and current distances of object point from the center of projection. While, 'x' and 'x'' are the previous and current pixel location. The formulation for the same is given below:

$$\frac{Y}{D} = \frac{x}{f} \qquad \frac{Y}{D\prime} = \frac{x\prime}{f} \qquad D' = \frac{x}{x\prime} \tag{78}$$

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2} \tag{79}$$

$$C_x, C_y = \left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right) \tag{80}$$

$$D = \frac{\sum_{i=1}^{n} \frac{d_i^o}{d_i^c}}{n} \tag{81}$$

Here, it should be noted that pixel location along the y axis are also calculated to finally calculate the distance between the center of the features and the given feature. In the end, 'D' which is the relative distance is given by the sum of ration of distance between the current

feature and center of feature denoted by 'd_i^c' and previous center of features and given feature denoted by 'd_i^o'.

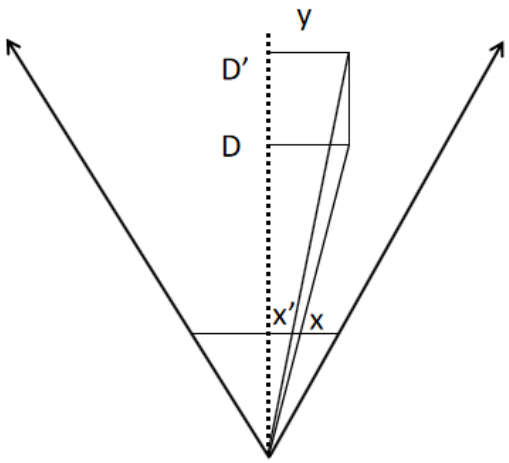

Figure 6.5: Geometry for Relative Distance

However, the accuracy of such method decrease when an object is too small or too far. And the reason behind it is again the epipolar geometry. As can be seen from Figure 6.6, the rate of change of distance of feature from the center axis of the camera decrease as the features gets closer to the axis. Thus, this method is limited by the scale of the object on the projection screen.

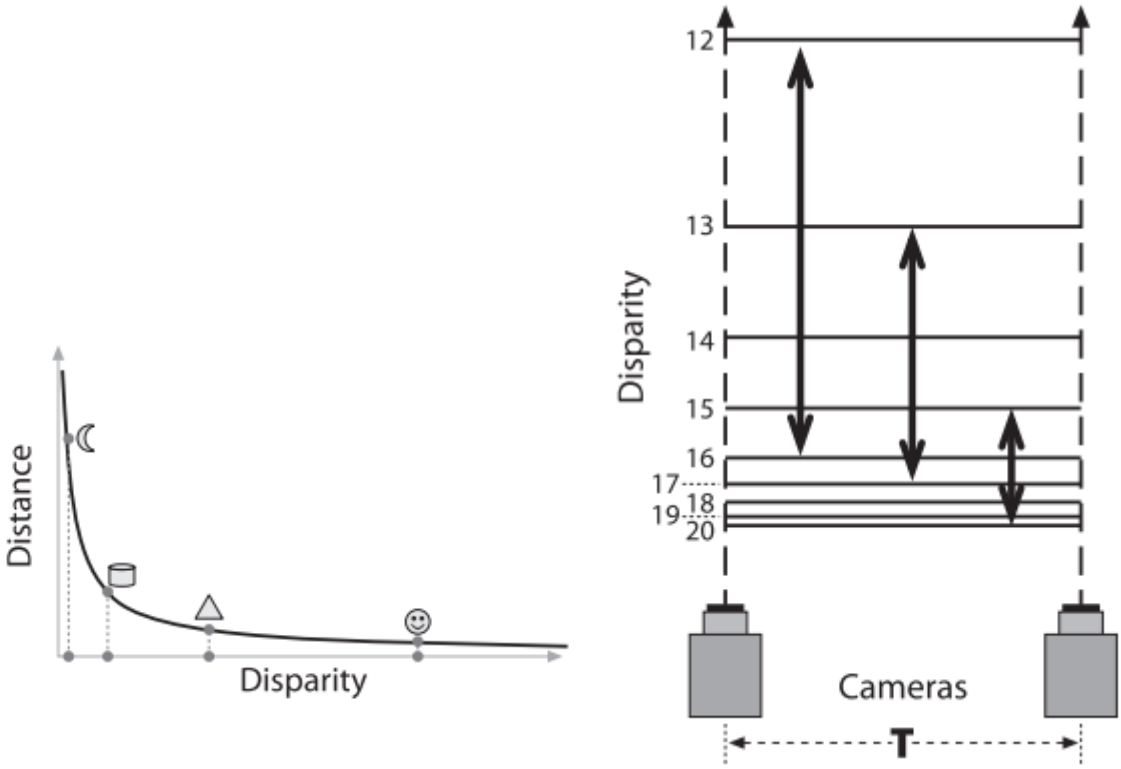

Figure 6.6: Disparity Visualization

CHAPTER 7

System Architecture

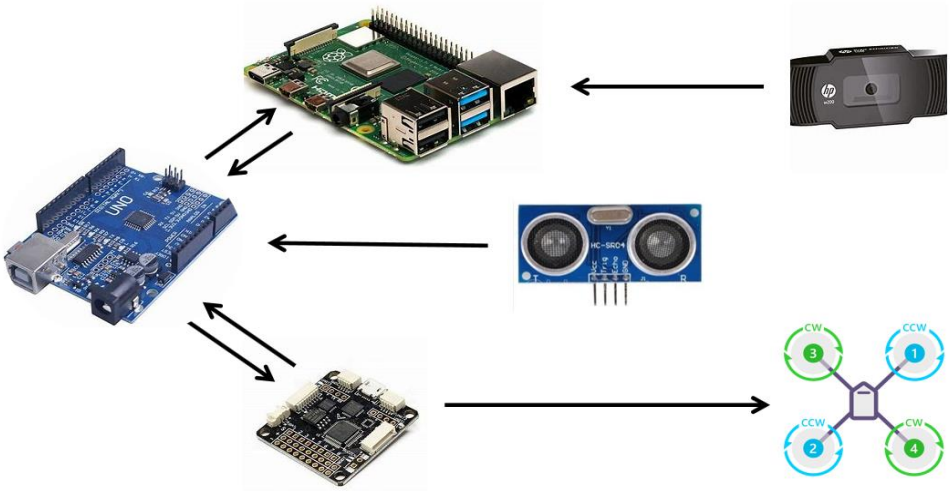

Figure 7.1: System architecture

The architecture of the platform is required to be able to perform computationally expensive calculation for visual odometry and trajectory generation, while also simultaneous taking inputs from different sensors and controlling the speed of the motor. And thus a architecture divided into three levels, namely:

1) Flight Controller
2) Micro Controller
3) Flight computer

A direct advantage of such a system is increase numbered of compatible sensors and connectivity to variety of wireless networks for transmitting data to ground base. A diagram of the current system architecture is shown in Figure 7.1.

7.1 Flight Controller

A F3-flight controller is used on the platform. It has inbuilt 6-axis IMU, thus we can get a telemetry data for acceleration for all three translational axis as well as the orientation along all three rotation axis of the drone. This flight controller handles the last PID stage. It is

designed to communicate with different sensors and receivers. It receives input from sensors and the pilot via radio receiver and controls the RPM of four motors of the quadcopter accordingly. The stable flight of quadcopter totally depends on the flight controller and its PID calibration. Following figure 7.2 shows the flight controller used in this project.

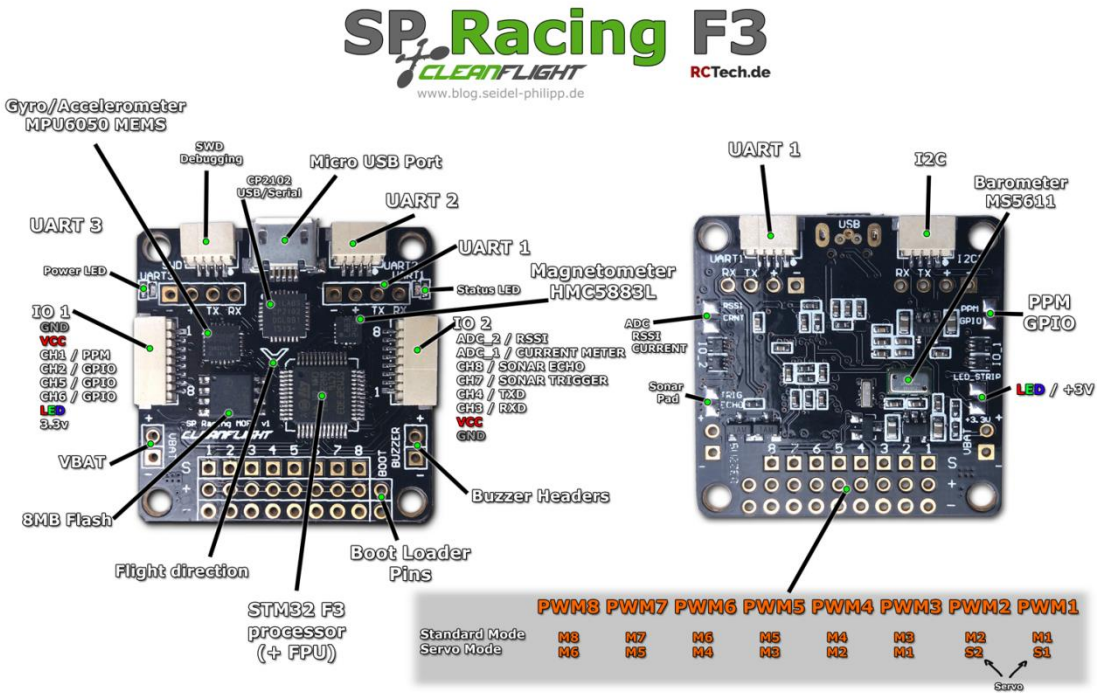

Figure 7.2: SP F3 Flight Controller

The figure 7.2 shows the hardware components of the flight controller. This flight controller runs on an open source software called CleanFlight. The software is designed to use the onboard sensors to undergo a stable flight.

7.2 Flight Computer

A Raspberry Pi-4 Model B 8Gb acts as a flight computer. It takes input directly from the camera and processes it for further use. Apart from performing computationally expensive programs, it acts as a server. It mainly transmits data to :

1) Ground Station (Laptop over here), done via WiFi
2) Arduino, done via UART

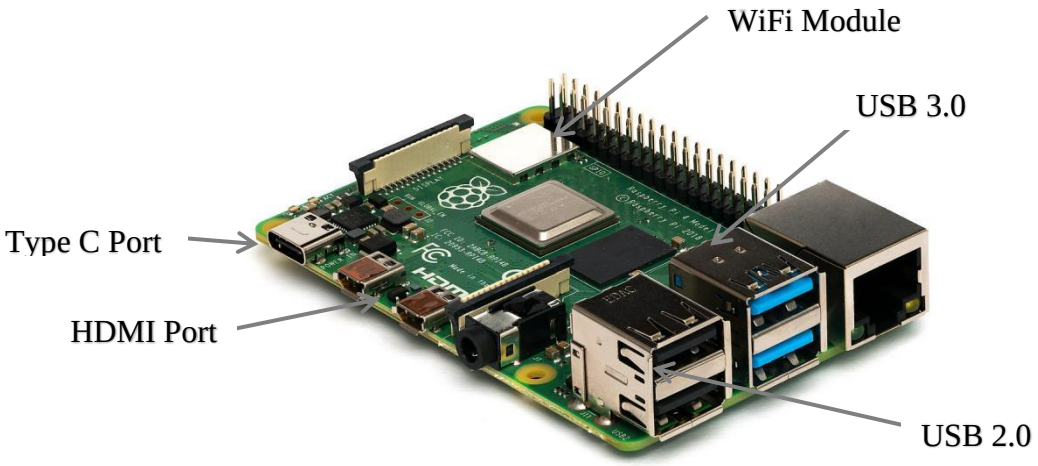

Figure 7.3: Raspberry Pi 4 Model B

A Figure 7.3 shows all the hardware component of the RPi that are currently under use. The Wi-Fi however, is not a ideal choice as the live video feed lags as well as it may even slow down other processes. For UART we use a USB 2.0 port of the RPi to transmit data over USB A/B cable with Arduino. The USB 3.0 is reserved to be used with cameras as they are faster than USB 2.0.

7.3 Micro-controller

An Arduino UNO is used in this project. The main purpose of the Arduino is to take data from variety of sensors which use different protocols. It should be noted that even though RPi has PWM pins, they are not as accurate as the PWM signal is software generated while Arduino has hardware generated PWM which is much more accurate and reliable.

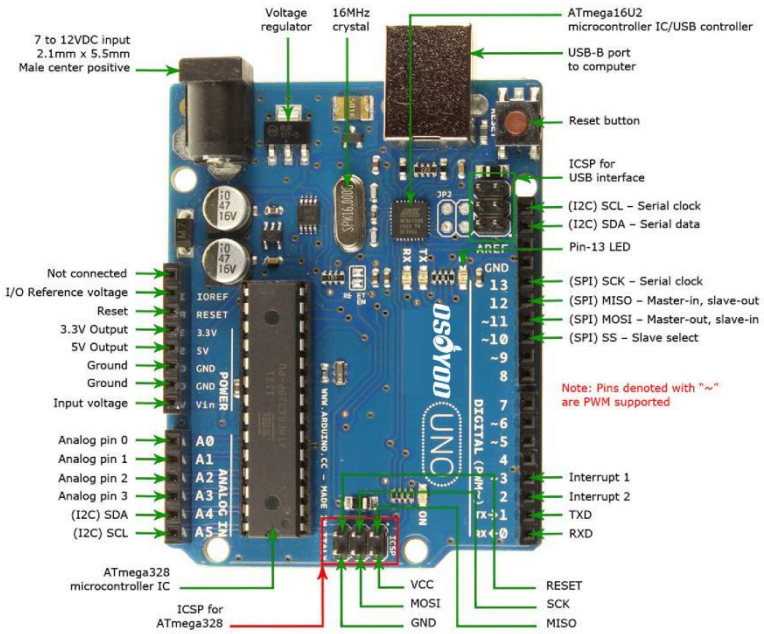

Figure 7.4: Arduino Pin Diagram

Here, Arduino takes data from 3 SONAR Sensors using digital pins and uses 6 PWM signal pins to communicate with the flight controller. Out of this 6 channels available for use 4 are used for Throttle, Roll, Yaw and Pitch control while remaining 2 are auxiliary channels. Apart from the GPIO pins the USB Type B port in UNO is used for UART communication with flight controller. The below given Figure 7.4 shows all the ports and GPIO pins currently used.

CHAPTER 8

Prototyping, Testing and Simulation

To achieve complete automation of the quadcopter, we need to calibrate and run efficiently all the four stages of control. So, following pragmatic approach, each stage is calibrated and tested separately and then communication is established between different stages in order to combine the result and achieve complete automation. In this chapter we will discuss the prototypes testing and simulations we carried out in order to calibrate each stages. We will also discuss the method adopted to establish communication between different controllers. The code and algorithms we developed are stated in appendix below the report.

8.1 Testing and calibrating the PID controller

Microcontroller board was used to control this loop. Actually the lower most PID loop is controlled by the flight controller, which in our case is SP F3 flight controller shown in figure 7.2. Arduino UNO (figure 7.4) is used to send commands to the flight controller. It takes input from ultrasonic sensors and controls various parameters using PID equations.

8.1.2 Communication between Arduino and Flight controller

This controller is designed to communicate with the standard receivers available in the market. One of the most basic and simple transmitter receiver protocol in PWM. The receiver sends PWM signals to the flight controller which provides it the necessary input for the control.

In order to replace the receiver with Arduino, we need to generate PWM signals and send it to the flight controller. For this, we can either generate PWM signals via simply varying the delay between pin HIGH and pin LOW, or we can use servo library in Arduino. Using servo library is the most efficient way to send PWM signals as it do not interrupts with the running code. The code we developed is stated in the appendix.

8.1.3 Steady hover

Now after establishing the communication, first and most simple automation task for quadcopter is steady hover. In order to achieve this, PID control in Z direction (as discussed in chapter 4) is used by taking feedback from ultrasonic sensors. I this the first step was to use run ultrasonic sensor.

Ultrasonic sensor works on the principle that sound travels at a constant velocity in a given medium. The sensor have a ultrasonic sound transmitter and receiver. It generates ultrasonic sound when signalled and gives a HIGH in echo pin when it receives the ultrasonic echo. Now by counting the time difference between these signal and echo, we can calculate the distance by the formula below.

$$d = v_{sound} * t / 2$$

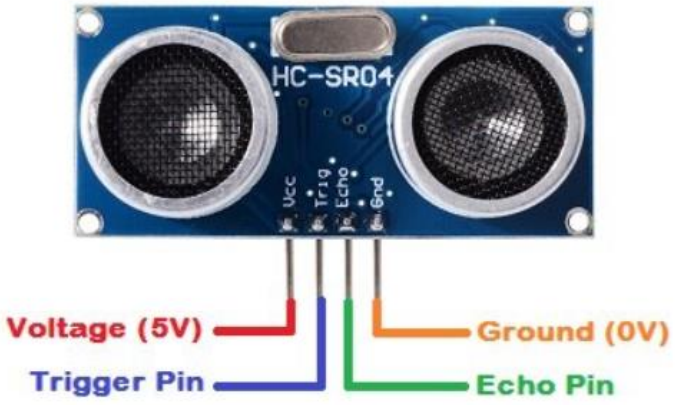

Figure 8.1: Ultrasonic Sensor

The values received from ultrasonic sensor oscillates constantly. In order to reduce these fluctuations, a filter need to be applied. We have use a method which uses trust factor in order to filter out the variations in the values. Following is the equation of the filter. Here, T is the trust factor for the sensor which in case of ultrasonic sensor is normally 1.25.

$$x = T \times x_{ultra} + (1 - T) \times x_{previous} \tag{82}$$

After receiving the value of the distance, we need to calculate the velocity. In order to do this, two values of distances are taken some time apart and rate of change with respect to time is calculated. These values of distance and velocity is then fed to the PID equation. By calibrating PID gains we can attain a steady hover flight. Following is the image of the testing of steady hover function.

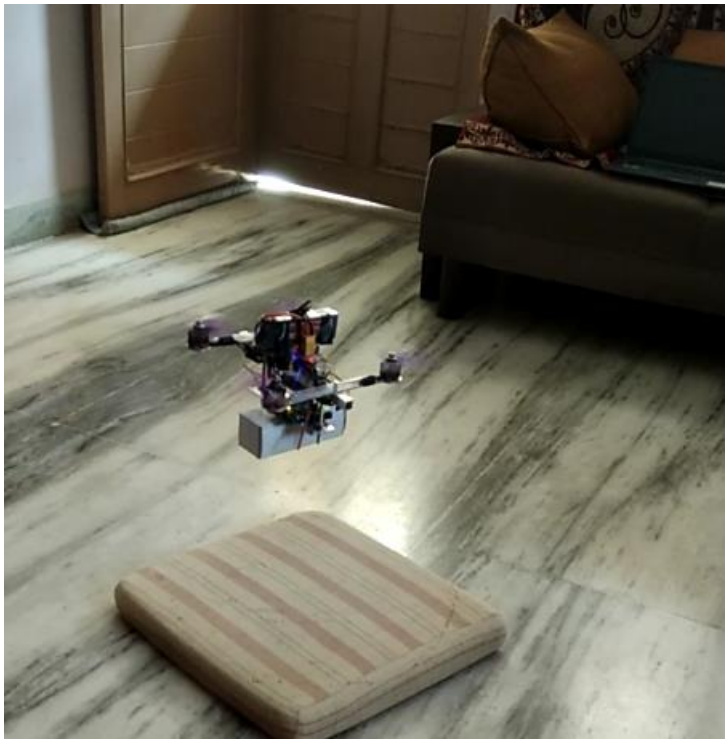

Figure 8.2: Steady Hover

8.1.4 Controlling all the three Directions

Adding extra two more sensors for calculating distance and velocities on x and y axis, we can control the position of quadcopter in x and y direction. The code for total control of the quadcopter on Arduino is stated in the appendix. During testing, the quadcopter was able to maintain its position for some until it receives incorrect value of distance or velocity from the sensor. In order to achieve this goal, more accurate and precise sensors are required. Once we achieve this task, and the quadcopter is able to correct its error accurately and quickly, we can feed the trajectory to the PID controller and it must follow it smoothly. Following is the image of quadcopter prototype trying to hold its position in all three direction.

Figure 8.3: Total Control

8.2 Automated trajectory generator simulation

The drone must be able to generate trajectories by its own in order to achieve complete automation. The algorithm used for generating trajectories is discussed in chapter 5 in detail. Using that algorithm, we have developed a code for automated trajectory generation in MATLAB environment. The code is designed to make trajectories passing through 2 to 4 number of points in 3-D space. The code was tested successfully in MATLAB simulation. The code is stated in appendix. Following are the screen shots of the simulation conducted.

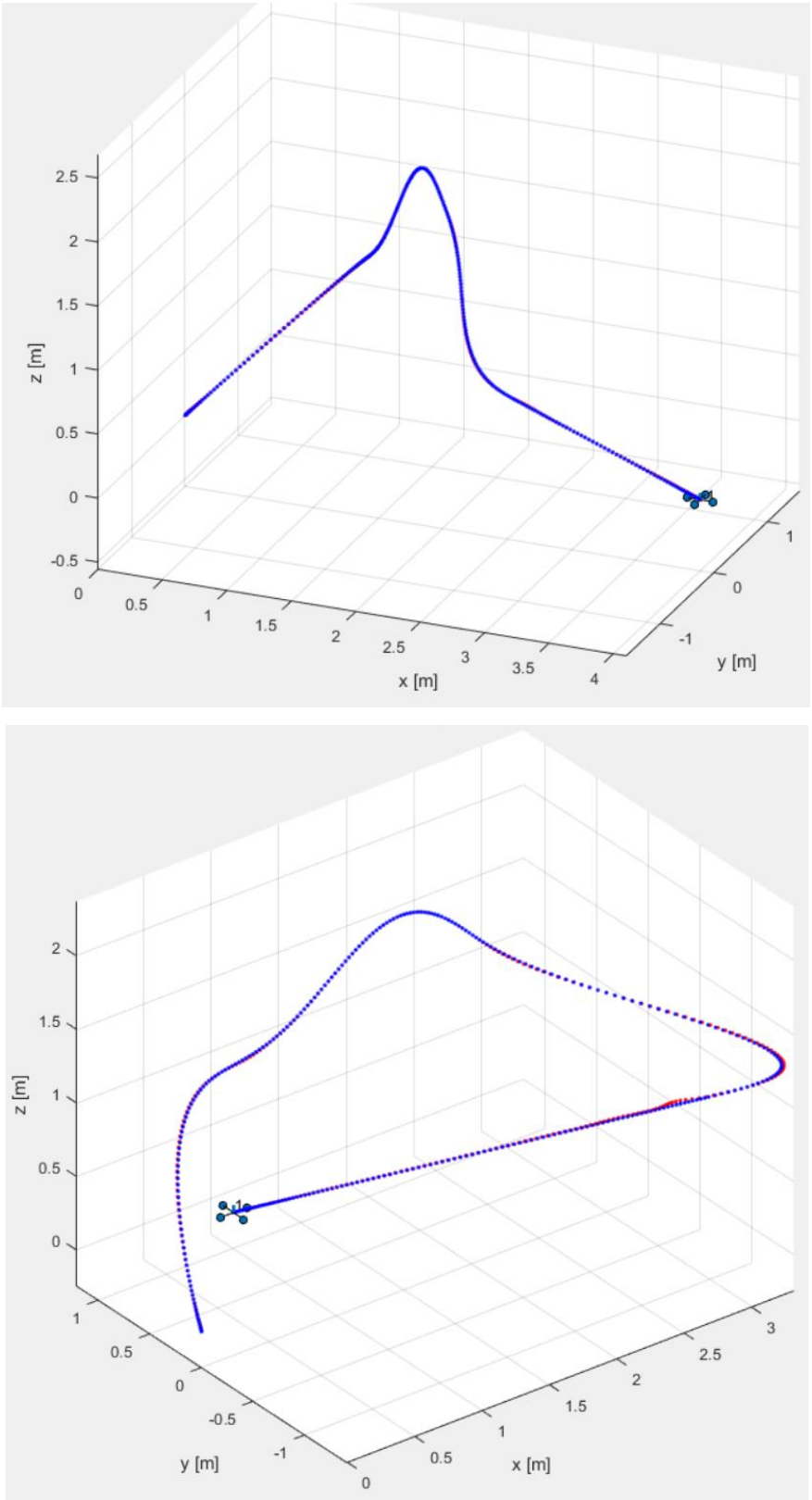

Figure 8.4: Trajectory generator simulation

8.2.1 Communication between RPi and Arduino

The communication between RPi and Arduino is done via UART using an USB A/B cord. The UART signal is transmitted through a USB 2.0 port of the RPi with a baudrate of 9600 and for the same we have used PySerial Library. In particular we have encode data into a string of 24 digits. It is done such that indices of the string which are a multiple of 4 represent the channel number and 3 digits following it is the value that is to be passed into the channel. The code used in the testing can be referred from the appendix.

8.3 Visual Odometry Testing

The Algorithm was tested to track distance and angle of orientation from a human like statue. And the image input was taken form a smart phone camera which transmitted live data via Wi-Fi.

To initial the testing a camera is placed in a steady position and the current distance and angle are taken as the reference distance (taken as 1) and reference angle (taken as 0) with respect to which other positions will be calculated. Then the object of interest which is the statue over here was selected by drawing a bounding box in the first image sent by the camera.

Then the camera was moved along a straight line such that the object was always in the front of it. In the first test the camera was moved further away from the object and the value of the relative distance was expected to be greater than 1 with no change in angle. For the second test the camera was moved nearer to the object and the value of the relative distance was expected to be lesser than 1. For the third and fourth test distance of the object was varied in a similar fashion to the first and second test, however, this time the angle of orientation was changed in both clockwise as well as anti-clock wise.

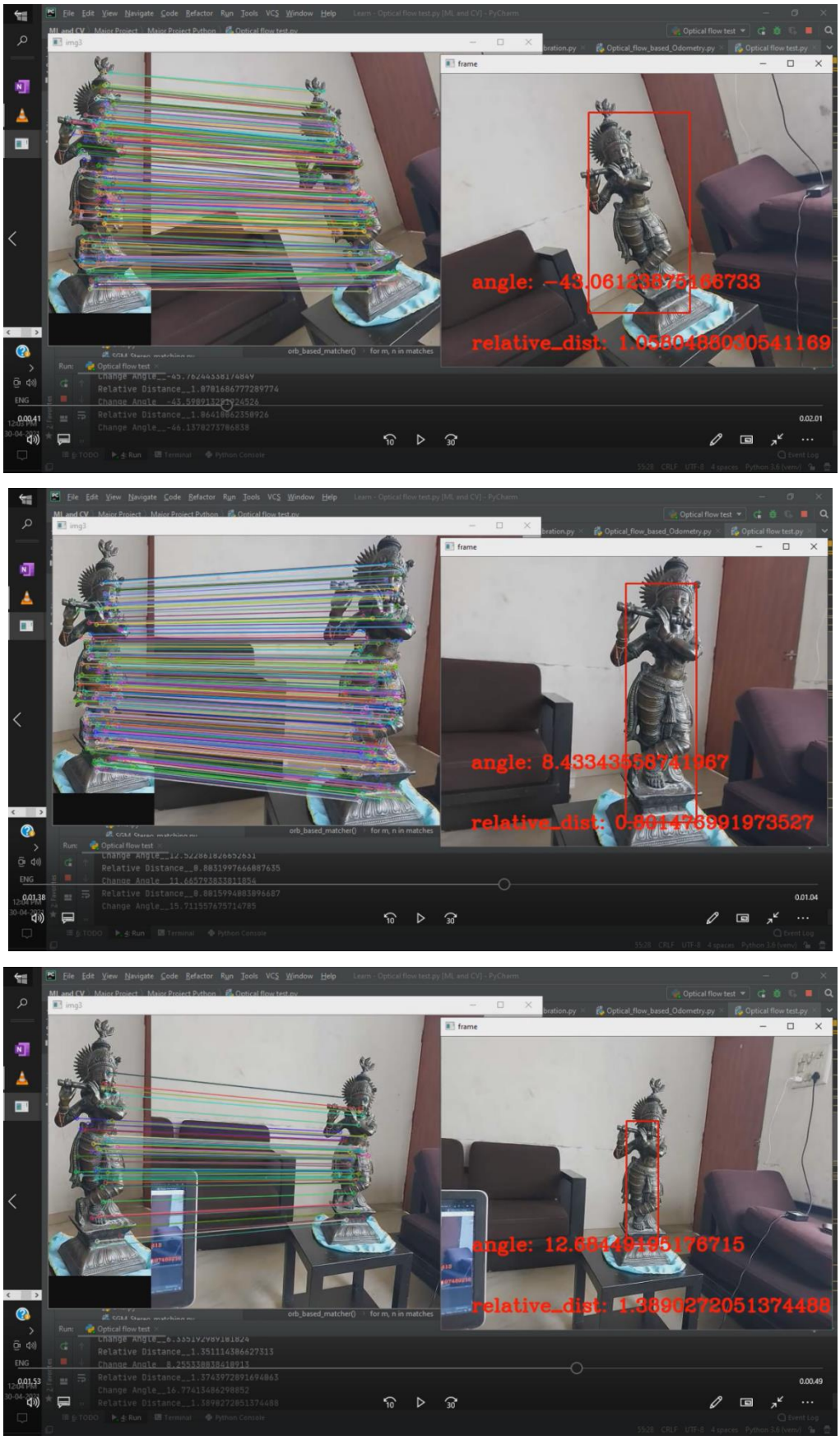

Figure 8.5: Visual Odometry testing

CHAPTER 9

Conclusion

In this project, we studied developed and tested various concepts and algorithms vital for drone automation. As discussed above, PID equations governing quadcopter were derived. It was also tested to an extend in a working prototype of the drone. The equations for automated trajectory generation were studied, and a viable algorithm was designed in MATLAB environment and simulated successfully. The algorithm can be easily converted into another programming language and applied to the actual hardware.

Using different algorithms and OpenCV library, a program was developed to follow a selected object and was tested on the hardware. Other than that, a program was developed which provided relative change in the position of the object only using a single camera. We were unable to test run the codes for steriovision due to lack of apparatus. But the algorithms of disparity mapping and distance measurement were studied in detail during the project.

In this project attempts were made to automate the drone. The different stages of automation were discussed and tested. For complete and accurate automation, accurate measurements of the current state of the drone are utmost necessary. In this project we have developed and tested algorithms for individual stages of automation. But for complete automation, all the systems need to work accurately in communication with one another. The system we used for testing included Raspberry PI as flight computer, Arduino UNO for PID Controller and SP F3 Racing as flight controller. We have also used a high definition camera for image sensing purpose. Following shows the image of the hardware.

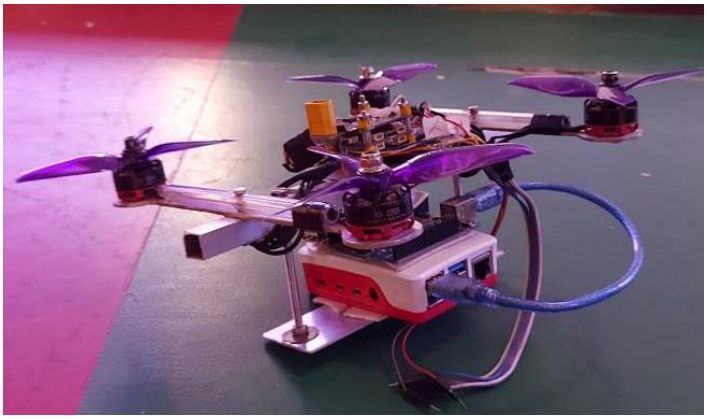

Figure 9.1: Hardware

This prototype weighted 830 grams which was tested for the object tracking purpose. The weight of the prototype was higher than the normal weight supported by the drone frame and motors. This might cause the motors to draw more current from the ESC and eventually bricking it. So to use this prototype with all the hardware, more powerful motors and higher amps ESC is required, or overall lowering the weight of the drone is required.

Other than that, Raspberry Pi was only able to handle mildly complicated algorithms. As the complication increased, the program used more time to react to the environment and hence lag in the code was observed. So for more complicated algorithms, it is recommended to use flight computer with better computing power.

The flight controller used was optimized and programmed for manual control and to communicate with standard receivers. Due to this, an extra Arduino UNO was required for the upper stage of the PID control. The reason we decided to go for this setup was our willingness to learn more on the PID control of the drone and its application. For the process of the automation to simplify, one can use better flight controllers specifically designed for flight automation like Pixhawk. The software ArduPilot run by Pixhawk have is programmed for all type of configuration and type of aircraft. This will help simplify the process of automation.

CHAPTER 10

Future Scope

Another motivation behind the project was to develop a platform and a base for the further research on the areal robotics. The prototype built can be used for the purpose of general research on quadcopter or can also be used for research on Quadrotor VTOL aircraft.

The further research involves improving the algorithms of the PID controller and image sensing to work in a dynamic environment. The algorithms were only tested in a controlled indoor environment. Other than that more accurate sensors can be implemented in order to increase the viability of the system.

Research on concepts of mapping and positioning of can be carried out. Algorithms like SLAM, RRT, A* etc. Can be further developed and tested on the platform. The dynamics of Quadrotor VTOL aircraft, discussed in the previous project can be used to develop the PID equations for its upper PID control loop. This will help further automate the aircraft.

Aerial robotics will be an important field in the near future. We believe our project might be able to plant a sapling of the research in this field in our University. There are vast applications of areal robotics in drone deliveries, military applications, agricultural applications, photo and videography, aerial survey, mapping rescue etc. Thus the scope of further research in this field is vast and very vital for further technological development.

REFERENCES

[1] Design, Implementation and Verification of a Quadrotor Tail-sitter VTOL UAV Ya Wang, Ximin Lyu, Haowei Gu, Shaojie Shen, Zexiang Li and Fu Zhang.

[2] A. Frank, J. McGrew, M. Valenti, D. Levine, and J. How, ”Hover transition and level flight control design for a single-propeller indoor airplane”, In AIAA Guidance, Navigation and Control Conference and Exhibit, p. 6318. 2007.

[3] B. W. McCormick, Aerodynamics of V/STOL flight[M]. Courier Corporation, 1967.

[4] N. Michael, D. Mellinger, Q. Lindsey and V. Kumar, "The GRASP Multiple Micro-UAV Testbed," in IEEE Robotics & Automation Magazine, vol. 17, no. 3, pp. 56-65, Sept. 2010, doi: 10.1109/MRA.2010.937855.

[5] V. Ghadiok, J. Goldin and W. Ren, "Autonomous indoor aerial gripping using a quadrotor," 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Francisco, CA, 2011, pp. 4645-4651, doi: 10.1109/IROS.2011.6094690.

[6] J. P. F. Guimarães et al., "Fully Autonomous Quadrotor: A Testbed Platform for Aerial
[7] Robotics Tasks," 2012 Brazilian Robotics Symposium and Latin American Robotics Symposium, Fortaleza, 2012, pp. 68-73, doi: 10.1109/SBR-LARS.2012.18.

[8] http://www.airfoiltools.com/

[9] John J. Craig, Introduction to Robotics: Mechanics and Control, 3rd edition, Presona Education, 2005.

[10] X. Lyu, H. Gu, Y. Wang, Z. Li, S. Shen and F. Zhang, "Design and implementation of a quadrotor tail-sitter VTOL UAV," 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 2017, pp. 3924-3930, doi: 10.1109/ICRA.2017.7989452.

[11] [10]D. Kubo, and S. Suzuki, Tail-Sitter Vertical Takeoff and Landing Unmanned Aerial Vehicle: Transitional Flight Analysis, Journal of Aircraft, Vol.45, No.1, pp. 292?297, 2008.

[12] R. H. Stone, "Control architecture for a tail-sitter unmanned air vehicle," 2004 5th Asian Control Conference (IEEE Cat. No.04EX904), Melbourne, Victoria, Australia, 2004, pp. 736-744 Vol.2.

[13] S. Bouabdallah, “Design and control of quadrotors with application to autonomous flying,” Ph.D. dissertation, Ecole Polytechnique F´ ´ ed´erale de Lausanne, Dec. 2007.

[14] A. Oosedo, S. Abiko, A. Konno, T. Koizumi, T. Furui and M. Uchiyama, "Development of a quad rotor tail-sitter VTOL UAV without control surfaces and experimental verification," 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, 2013, pp. 317-322, doi: 10.1109/ICRA.2013.6630594.

[15] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

[16] OpenCV. (2015). *Open Source Computer Vision Library*.

[17] Bradski, G., & Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc."

[18] Rublee, Ethan & Rabaud, Vincent & Konolige, Kurt & Bradski, Gary. (2011). ORB: an efficient alternative to SIFT or SURF. Proceedings of the IEEE International Conference on Computer Vision. 2564-2571. 10.1109/ICCV.2011.6126544.

APPENDIX

The following codes have been used for various processes through out the Project. And they requires additional open CV, PySerial and Numpy Libraries as dependencies. Link for further details and insights are given below under the title of resources.

Camera Calibration Code to be run in RPI (PYTHON)

```
import cv2
import numpy as np
import os


class StereoCalibration(object):

    def __init__(self, filepath):

        self.objpoints = []
        self.imgpoints_l = []
        self.imgpoints_r = []

    def img_point(self, folder, nc, nr, resize=False, sx=1, sy=1, left = True, right = False):

        objp = np.zeros((nc * nr, 3), dtype=np.float32)
        objp[:, :2] = np.mgrid[0:nc, 0:nr].T.reshape(-1, 2)
        images = []
        for img in os.listdir(folder):
            img = cv2.imread(os.path.join(folder, img))
            if img is not None:
                images.append(img)
        termination_criteria = (cv2.TERM_CRITERIA_EPS +
cv2.TERM_CRITERIA_COUNT, 30, 0.001)
        for img in images:
            if resize:
                img = cv2.resize(img, None, fx=sx, fy=sy,
interpolation=cv2.INTER_AREA)

            ret, corners = cv2.findChessboardCorners(img, (nc, nr), None)

            if (ret and left):
                self.objpoints.append(objp)
                corner2 = cv2.cornerSubPix(img, corners, (11, 11), (-1, -1),
criteria=termination_criteria)
                self.imgpoints_l.append(corner2)

                ret, self.mtx_l, self.dist_l, self.rvecs_l, self.tvecs_l =
cv2.calibrateCamera(self.objpoints,

self.imgpoints_l,
```

```
images[0].shape[::-1], None, None)
                        h, w = images[0].shape[:2]
                        self.n_mtx_l, roi = cv2.getOptimalNewCameraMatrix(self.mtx_l,
self.dist_l, (w, h), 1, (w, h))
                        values_left =    dict({'n_mtx_l': self.n_mtx_l, 'dist_l': self.dist_l,
                                                'rvecs_l': self.rvecs_l, 'tvecs_l': self.tvecs_l,
'roi_l': self.roi})
                        return values_left

                elif (ret and right):
                        self.objpoints.append(objp)
                        corner2 = cv2.cornerSubPix(img, corners, (11, 11), (-1, -1),
criteria=termination_criteria)
                        self.imgpoints_r.append(corner2)

                        ret, self.mtx_r, self.dist_r, self.rvecs_r, self.tvecs_r =
cv2.calibrateCamera(self.objpoints,

self.imgpoints_r,

images[0].shape[::-1],

None, None)
                        h, w = images[0].shape[:2]
                        self.n_mtx_r, roi = cv2.getOptimalNewCameraMatrix(self.mtx_r,
self.dist_r, (w, h), 1, (w, h))
                        values_right =    dict({'n_mtx_r': self.n_mtx_r, 'dist_r': self.dist_r,
'rvecs_r': self.rvecs_r, 'tvecs_r': self.tvecs_r, 'roi_r': roi})
                        return values_right

        def stereo_calib(self, values_left, values_right, img_size):
                R, T, E, F = cv2.stereoCalibrate(self.objpoints, self.imgpoints_l, self.imgpoints_r,
self.n_mtx_l, self.dist_l, self.n_mtx_r, self.dist_r, imageSize= img_size)


def undistort(img, mtx, dist, n_mtx, roi):
        img = cv2.undistort(img, mtx, dist, None, newCameraMatrix=n_mtx)
        x, y, w, h = roi
        img = img[y:y + h, x:x + w]
        return img
```

Stereo Pair Matching Code to be run in RPI (PYTHON)

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from matplotlib import patches

img1 = cv2.imread(' Path to left image')
img2 = cv2.imread('Path to right image')

def compute_left_disparity_map(img_left, img_right):
    ### START CODE HERE ###

    # Parameters
    num_disparities =   7*16
     block_size = 11

     min_disparity = 0
     window_size = 6

     img_left = cv2.cvtColor(img_left, cv2.COLOR_BGR2GRAY)
     img_right = cv2.cvtColor(img_right, cv2.COLOR_BGR2GRAY)

     left_matcher_BM = cv2.StereoBM_create(
          numDisparities=num_disparities,
          blockSize=block_size
     )

     # Stereo SGBM matcher
    left_matcher_SGBM = cv2.StereoSGBM_create(
          minDisparity=min_disparity,
          numDisparities=num_disparities,
          blockSize=block_size,
          P1=8 * 3 * window_size ** 2,
          P2=32 * 3 * window_size ** 2,
          mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
     )

     # Compute the left disparity map
    disp_left = left_matcher_BM.compute(img_left, img_right).astype(np.float32) / 16

     return disp_left

disp_left = compute_left_disparity_map(img1, img2)
cv2.imshow('disp_left', disp_left )
cv2.waitKey(
```

Object Tracking Algorithm Code to be run in RPI (PYTHON)

```
import cv2
import numpy as np
import multiprocessing

def get_relative_dist(orignal_point, current_point, focus):
    dist = ( (focus - 1) * (orignal_point - current_point) )/ current_point
    return dist


def get_template(q1, q2):
    url = "http://192.168.183.89:8080/video"
    cap = cv2.VideoCapture(url)
    ref, frame = cap.read()
    roi = cv2.selectROI(windowName="ROI", img=frame, showCrosshair=True,
fromCenter=False)
    x, y, w, h = roi
    template = frame[y:y + h, x:x + w]
    cv2.imshow("img", template)
    cv2.waitKey(3000)
    cv2.destroyAllWindows()
    q2.send(template)

    while True:
        _, frame = cap.read()
        res = cv2.matchTemplate(frame, template, cv2.TM_SQDIFF)
        q1.send([frame, res, w, h])
        print("track done")


def template_matcher(q1, q3, q5):
    while True:
        frame, res, w, h = q1.recv()
        min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
        top_left = min_loc
        bottom_right = (top_left[0] + w, top_left[1] + h)
        object = frame[top_left[1]:bottom_right[1], top_left[0]:bottom_right[0]]
        cv2.imshow("im", object)
        cv2.waitKey(1)
        q3.send(object)
        q5.send([frame, top_left, bottom_right])


def ORB_matcher(q2, q3, q4):
    template = q2.recv()
    w = template.shape[0]
    orb = cv2.ORB_create(nfeatures=1000)
    kp1, des1 = orb.detectAndCompute(template, None)
```

```
        while True:
            try:
                object = q3.recv()
                kp2, des2 = orb.detectAndCompute(object, None)
                bf = cv2.BFMatcher(cv2.NORM_HAMMING)
                matches = bf.knnMatch(des1, des2, k=2)
                dist_template = 0
                dist_object = 0
                good =[]

                for m, n in matches:
                    if m.distance < 0.65 * n.distance:
                        good.append([m])
                        dist_template = dist_template - kp1[m.queryIdx].pt[0] + (w / 2)
                        dist_object = dist_object - kp2[m.trainIdx].pt[0] + (w / 2)

                img3 = cv2.drawMatchesKnn(template, kp1, object, kp2, good, None, flags=2)
                cv2.imshow("img3", img3)
                cv2.waitKey(1)
                #imgkp1 = cv2.drawKeypoints(img1, kp1, None)
              #imgkp2 = cv2.drawKeypoints(img2, kp2, None)

              q4.send([dist_template, dist_object])

            except Exception as e:
                print('knn error', e)


def get_co_ord(q4, q5):
        while True:
            axis = q5.recv()
            frame = axis[0]
            top_left = axis[1]
            bottom_right = axis[2]
            dist = q4.recv()
            dist_template = dist[0]
            dist_object = dist[1]

            cv2.rectangle(frame, top_left, bottom_right, (0, 0, 225), 2)
            #cv2.imshow("img", frame)
          #cv2.waitKey(1)
          if (dist_object > dist_template):
                print("object Near")
                cv2.putText(frame, "object Near", (50, 300),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA, False)

            else:
                print("object Far")
                cv2.putText(frame, "object Far", (50, 300),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA, False)
```

```
            mx = (bottom_right[0] + top_left[0]) / 2
            my = (bottom_right[1] + top_left[1]) / 2

            print(mx, my)

            if mx < 290:
                print("turn left")
                cv2.putText(frame, "turn left", (50, 200),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA, False)
            elif mx > 350:
                print("turn right")
                cv2.putText(frame, "turn right", (50, 200),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA, False)

            cv2.imshow("img1", frame)
            cv2.waitKey(1)

if __name__ == '__ma' \
                   'in__':

        # Defined pipe line parent-child objects

       parent_cam1, child_cam1 = multiprocessing.Pipe()
        parent_cam2, child_cam2 = multiprocessing.Pipe()
        parent_object, child_object = multiprocessing.Pipe()

        parent_track, child_track = multiprocessing.Pipe()
        parent_orb, child_orb = multiprocessing.Pipe()

        # Start multiple Processes simultaneously

      cam_process = multiprocessing.Process(target=get_template, args=(parent_cam1,
parent_cam2,))
        cam_process.start()

        track_process = multiprocessing.Process(target=template_matcher, args=(child_cam1,
parent_object, parent_track,))
        track_process.start()

        orb_process = multiprocessing.Process(target=ORB_matcher, args=(child_cam2,
child_object, parent_orb,))
        orb_process.start()

        show_process = multiprocessing.Process(target= get_co_ord, args=(child_orb,
child_track))
        show_process.start()
```

Position Estimation Algorithm Code to be run in RPI (PYTHON)

```
import cv2
import numpy as np
import multiprocessing
import math

cap = cv2.VideoCapture(0)
orb = cv2.ORB_create(nfeatures=1000000)

def get_dist(cx, cy, x, y):
	dist = ((cx-x)**2+(cy-y)**2)**0.5
	return dist

def get_angle(ox, cx, oy, cy, ocx, ocy, ccx, ccy):
	slope_current = (ccy-cy)/(ccx-cx)
	slope_orignal = (ocy-oy)/(ocx-ox)

	angle = math.atan(slope_orignal)-math.atan(slope_current)
	return angle

def get_relative_dist(ox, oy, cx, cy, ocx, ocy, ccx, ccy):
	dist_c = get_dist(ccx, ccy, cx, cy)
	dist_o = get_dist(ocx, ocy, ox, oy)
	dist = dist_o/dist_c
	return dist

def get_object_feat():
	ref, frame = cap.read()
	roi = cv2.selectROI(windowName="ROI", img=frame, showCrosshair=True,
fromCenter=False)
	x, y, w, h = roi
	template = frame[y:y + h, x:x + w]
	cv2.imshow("img", template)
	cv2.waitKey(3000)
	kp1, des1 = orb.detectAndCompute(template, None)
	return kp1, des1, template, x, y

def orb_based_matcher(kp1, des1, template, x, y):
	ref, frame = cap.read()
	kp2, des2 = orb.detectAndCompute(frame, None)
	bf = cv2.BFMatcher(cv2.NORM_HAMMING)
	matches = bf.knnMatch(des1, des2, k=2)

	min_w = 0
	min_h = 0
	max_w = 1200
	max_h = 1200

	good = []
```

```
        orignal_points_x = []
        current_points_x = []
        orignal_points_y = []
        current_points_y = []

        for m, n in matches:
            if m.distance < 0.55 * n.distance:
                good.append([m])

                if (min_w <    kp2[m.trainIdx].pt[0]):
                    min_w =    kp2[m.trainIdx].pt[0]
                elif (max_w >    kp2[m.trainIdx].pt[0]):
                    max_w = kp2[m.trainIdx].pt[0]

                if (min_h <    kp2[m.trainIdx].pt[1]):
                    min_h =    kp2[m.trainIdx].pt[1]
                elif (max_h >    kp2[m.trainIdx].pt[1]):
                    max_h =    kp2[m.trainIdx].pt[1]

                orignal_points_x.append(kp1[m.queryIdx].pt[0] + x)
                current_points_x.append(kp2[m.trainIdx].pt[0])
                orignal_points_y.append(kp1[m.queryIdx].pt[1] + y)
                current_points_y.append(kp2[m.trainIdx].pt[1])

        ocx = sum(orignal_points_x) / len(orignal_points_x)
        ccx = sum(current_points_x) / len(current_points_x)

        ocy = sum(orignal_points_y)/len(orignal_points_y)
        ccy = sum(current_points_y)/len(current_points_y)

        point_dist = []
        point_angle = []

        for i in range(len(orignal_points_x)):
            dist = get_relative_dist(orignal_points_x[i], orignal_points_y[i], current_points_x[i],
current_points_y[i], ocx, ocy, ccx, ccy)
            point_dist.append(dist)
            angle = get_angle(orignal_points_x[i], current_points_x[i], orignal_points_y[i],
current_points_y[i], ocx, ocy, ccx, ccy)
            point_angle.append(angle)

        relative_dist = ((sum(point_dist)) / len(point_dist))
        change_angle = ((sum(point_angle))/len(point_angle))

        max_w = int(max_w)
        max_h = int(max_h)
        min_w = int(min_w)
        min_h = int(min_h)

        img3 = cv2.drawMatchesKnn(template, kp1, frame, kp2, good, None, flags=2)
```

```
        print("Relative Distance__" + str(relative_dist))
        print("Change Angle__" + str(math.degrees(change_angle)))

        cv2.putText(frame, "relative_dist: " + str(relative_dist), (50, 450),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2,
                    cv2.LINE_AA, False)
        cv2.putText(frame, "angle: " + str(math.degrees(change_angle)), (50, 350),
cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA, False)
        cv2.rectangle(frame, (min_w, min_h), (max_w, max_h), (0, 0, 225), 2)

        cv2.imshow("frame", frame)
        cv2.imshow("img3", img3)
        cv2.waitKey(1)

if __name__ == '__main__':
        kp1, des1, template, x, y = get_object_feat()
        cv2.destroyAllWindows()
        while True:
            orb_based_matcher(kp1, des1, template, x, y)
```

UART Communication Algorithm Code to be run in RPI (PYTHON)

```
import serial
import cv2

def drawBox(img,bbox):
        x,y,w,h = int(bbox[0]),int(bbox[1]),int(bbox[2]),int(bbox[3])
        cv2.rectangle(img,(x,y),((x+w),(y+h)),(255,0,255),3,1)
        cv2.putText(img, "Tracking", (75, 75), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255,
0), 2)

Throttle = 40
Roll = 93
Yaw = 93
Pitch = 93
Aux1 = 40
Aux2 = 40
data = str(Throttle + 1000) + str(Roll + 2000) + str(Yaw + 3000) + str(Pitch + 4000) +
str(Aux1 + 5000) + str(
        Aux2 + 6000) + '\n'
#ser = serial.Serial('/dev/ttyUSB0')
ser = serial.Serial()
ser.baudrate = 115200
ser.port = "/dev/tty"
ser.open()

while True:
        print("Entre the control value:")
```

```
value = input()
if value == 'w':
	Throttle = Throttle + 1
elif value == 's':
	Throttle = Throttle - 1
elif value == 'a':
	Yaw = Yaw - 1
elif value == 'd':
	Yaw = Yaw + 1
elif value == 'i':
	Pitch = Pitch + 1
elif value == 'k':
	Pitch = Pitch - 1
elif value == 'j':
	Roll = Roll - 1
elif value == 'l':
	Roll = Roll + 1
elif value[0:2] == 'th':
	Throttle = int(value[2:])
elif value[0:2] == 'ya':
	Yaw = int(value[2:])
elif value[0:2] == 'ro':
	Roll = int(value[2:])
elif value[0:2] == 'pi':
	Pitch = int(value[2:])
elif value[0:2] == 'a1':
	Aux1 = int(value[2:])
elif value[0:2] == 'a2':
	Aux2 = int(value[2:])
elif value == 'arm':
	Throttle = 40
	Yaw = 93
	Roll = 93
	Pitch = 93
	Aux1 = 140
	Aux2 = 40
elif value == 'track': #object tracking loop
	cap = cv2.VideoCapture(0)
	tracker = cv2.TrackerMIL_create()
	success, img = cap.read()
	bbox = cv2.selectROI("Tracking", img, False)
	tracker.init(img, bbox)
	while True:
		timer = cv2.getTickCount()
		success, img = cap.read()
		success, bbox = tracker.update(img)
		if success:
			drawBox(img, bbox)
		else:
			cv2.putText(img, "Lost", (75, 75), cv2.FONT_HERSHEY_SIMPLEX,
```

```
0.7, (0, 0, 255),          2)

                    fps = cv2.getTickFrequency() / (cv2.getTickCount() - timer)
                    cv2.putText(img, str(int(fps)), (75, 50), cv2.FONT_HERSHEY_SIMPLEX,
0.7, (0, 0,             255), 2)
                    cv2.imshow("tracking", img)
                    mx = int(bbox[0]) + (int(bbox[2]) / 2)
                    my = int(bbox[1]) + (int(bbox[3]) / 2)
                    if mx < 290:
                          Yaw = 96
                    elif mx > 350:
                          Yaw = 90
                    else:
                          Yaw = 93
                    if my < 190:
                          Pitch = 95
                    elif my > 285:
                          Pitch = 91
                    else:
                          Pitch = 93
                    data = str(Throttle + 1000) + str(Roll + 2000) + str(Yaw + 3000) + str(Pitch +
4000) + str(
                          Aux1 + 5000) + str(Aux2 + 6000) + '\n'
                    ser.write(data.encode("utf-8"))
                    print(data)
                    if cv2.waitKey(1) & 0xff == ord('q'):
                          break
              Yaw = 93
              Pitch = 93
        elif value == "end":
              break
        else:
              print("Invalid data:\n")
        if 40 <= Throttle <= 150 and 40 <= Roll <= 150 and 40 <= Yaw <= 150 and 40 <= Pitch
<=   150 and 40 <= Aux1 <= 150 and 40 <= Aux2 <= 150:
              data = str(Throttle + 1000) + str(Roll + 2000) + str(Yaw + 3000) + str(Pitch + 4000)
+     str(Aux1 + 5000) + str(
                    Aux2 + 6000) + '\n'
        else:
              print("Value out of range:")
        ser.write(data.encode("utf-8"))
        print(data)
ser.close()
```

Arduino Code for 3-D PID control:

```
#include <Servo.h>

#define SERVOS 4
```

```
#include <NewPing.h>
#define trigger_pin    2
#define echo_pinZ        4    // Arduino pin tied to echo pin on the ultrasonic sensor.
#define echo_pinX        7
#define echo_pinY        8
#define max_distance 400 // Maximum distance we want to ping for (in centimeters).
Maximum sensor distance is rated at 400-500cm.
#define pwm1      13
#define pwm2      12
#define pwm3      11
#define pwm4      10
unsigned long t, t2;
float aZ, aX, aY, distanceZ, distanceX, distanceY, Height, set_distX, set_distY;
float velocityZ, velocityX, velocityY, thrust, trust_const;
int pwm_value1, pwm_value2, pwm_value3, pwm_value4, Roll=93, Yaw=93, Pitch=93,
Aux1, Aux2=40, Throttle=45;
NewPing sonarZ(trigger_pin, echo_pinZ, max_distance);
NewPing sonarX(trigger_pin, echo_pinX, max_distance);
NewPing sonarY(trigger_pin, echo_pinY, max_distance);
Servo myservo[SERVOS];
int servo_pins[SERVOS] = {3,5,6,9};
float Kp1, Kv1, Kp2, Kv2, Kp3, Kv3;
float mass = 0.600; //in kilograms

void setup() {
   Serial.begin(115200); // Initialize serial with a baud rate of 115200 bps
   for(int i = 0; i < SERVOS; i++) {
      myservo[i].attach(servo_pins[i]);
   }
   pinMode(pwm1, INPUT);
   pinMode(pwm2, INPUT);
   pinMode(pwm3, INPUT);
   pinMode(pwm4, INPUT);
}
```

```
void loop() {

  //Reading PWM Values

  pwm_value1 = pulseIn(pwm1, HIGH);
  pwm_value2 = pulseIn(pwm2, HIGH);
  pwm_value3 = pulseIn(pwm3, HIGH);
  pwm_value4 = pulseIn(pwm4, HIGH);

  //Reading Sonar values and calculating distances and velocities

  //delay(50);                          // Wait 50ms between pings (about 20 pings/sec).
29ms should be the shortest delay between pings.

  trust_const = 0.2;
  distanceZ = ((sonarZ.ping_cm())*trust_const)+((1-trust_const)*aZ);
  delay(10);
  distanceX = ((sonarX.ping_cm())*trust_const)+((1-trust_const)*aX);
  delay(10);
  distanceY = ((sonarY.ping_cm())*trust_const)+((1-trust_const)*aY);
  t2 = millis() - t;
  velocityZ = ((distanceZ - aZ)*1000)/t2;
  velocityX = ((distanceX - aX)*1000)/t2;
  velocityY = ((distanceY - aY)*1000)/t2;
  aZ = distanceZ;
  aX = distanceX;
  aY = distanceY;
  t = millis();

  Serial.print("DistanceZ: ");
```

```
    Serial.println(distanceZ); // Send ping, get distance in cm and print result (0 = outside set distance range)
    Serial.print("DistanceX: ");
    Serial.println(distanceX);
    Serial.print("DistanceY: ");
    Serial.println(distanceY);
    Serial.print("VelocityZ: ");
    Serial.println(velocityZ);
    Serial.print("VelocityX: ");
    Serial.println(velocityX);
    Serial.print("VelocityY: ");
    Serial.println(velocityY);

    //Height maintaining PID Control

    if (pwm_value1 > 1500){

        Kp1 = 2; //set for 10cm hovering height
        Kv1 = 0.3;
        Height = 10;

        thrust = (9.81+(Kv1*(0-velocityZ)/100)+(Kp1*(Height-distanceZ)/100))*950/9.81; //PID equation
        //thrust = thrust*1000/9.81; //in grams
        Serial.println(thrust);
        if (thrust > 1200){
            Throttle = (1.2*40)+45;
        }
        else if (thrust <= 0){
            Throttle = 45;
        }
        else{
        Throttle = (thrust*40/1000)+45;
        }
```

```
    }
    else {
       Throttle = 45;
    }

    //X distance maintaining PID control

    if (pwm_value2 > 1500){

       Kp2 = 1;
       Kv2 = 6;
       set_distX = 100; //(pwm_value3-1000)/5;


Pitch=(-180/(9.81*3.14))*(0+(Kv2*(0-velocityX)/100)+(Kp2*(set_distX-distanceX)/100));
       Pitch=Pitch+93;
       Serial.print("Pitch: ");
       Serial.println(Pitch);
       if (Pitch > 113){
          Pitch = 113;
       }
       else if (Pitch < 73){
          Pitch = 73;
       }
    }
    else{
       Pitch = 93;
    }

    //Y distance maintaining PID control

    if (pwm_value2 > 1500){

       Kp3 = 1;
```

```
      Kv3 = 6;
      set_distY = 100; //(pwm_value3-1000)/5;


Roll=(-180/(9.81*3.14))*(0+(Kv3*(0-velocityY)/100)+(Kp3*(set_distY-distanceY)/100));
      Roll=Roll+93;
      Serial.print("Roll: ");
      Serial.println(Roll);
      if (Roll > 113){
         Roll = 113;
      }
      else if (Roll < 73){
         Roll = 73;
      }
   }
   else{
      Roll = 93;
   }

   //Pitch failsafe

   if (pwm_value3 > 1550 || pwm_value3 < 1450){
      Pitch = (pwm_value3*45)/500 + 93;
   }

   //Roll failsafe

   if (pwm_value4 > 1550 || pwm_value4 < 1450){
      Roll = (pwm_value4*45)/500 + 93;
   }


    Serial.println(Pitch);
    myservo[0].write(Throttle); //Throttle
```

```
    delay(10);
    myservo[1].write(Roll); //Roll
    delay(10);
    myservo[2].write(Yaw); //Yaw
    delay(10);
    myservo[3].write(Pitch); //Pitch
    delay(10);

}
```

Arduino code for serial communication between Rpi and Arduino

```
#include <Servo.h>
#define SERVOS 6
Servo myservo[SERVOS];
int servo_pins[SERVOS] = {3,5,6,9,10,11};
void setup() {
   Serial.begin(115200); // Initialize serial with a baud rate of 115200 bps
   for(int i = 0; i < SERVOS; i++) {
      myservo[i].attach(servo_pins[i]);
}
}



void loop() {

   int angle = 0;

   while(Serial.available() > 0) {

      String input = Serial.readStringUntil('\n');
      String iThrottle = input.substring(1,4);
      String iRoll = input.substring(5,8);
      String iYaw = input.substring(9,12);
      String iPitch = input.substring(13,16);
      String iAux1 = input.substring(17,20);
      String iAux2 = input.substring(21,24);
      int Throttle = iThrottle.toInt();
      int Roll = iRoll.toInt();
      int Yaw = iYaw.toInt();
      int Pitch = iPitch.toInt();
      int Aux1 = iAux1.toInt();
      int Aux2 = iAux2.toInt();
```

```
    Serial.println(input);
    Serial.println(Throttle);
    Serial.println(Roll);
    Serial.println(Yaw);
    Serial.println(Pitch);
    Serial.println(Aux1);
    Serial.println(Aux2);
  myservo[0].write(Throttle); //Throttle
  delay(10);
  myservo[1].write(Roll); //Roll
  delay(10);
  myservo[2].write(Yaw); //Yaw
  delay(10);
  myservo[3].write(Pitch); //Pitch
  delay(10);
  myservo[4].write(Aux1); //Aux1
  delay(10);
  myservo[5].write(Aux2); //Aux2
  delay(10);



}
```

MATLAB code for automated minimum jerk trajectory generation. Here waypoints0 is a matrix providing set of points in 3 dimensions through which the trajectory must pass.

```
s=size(waypoints0);
traj=zeros((s(1,1)-1),3);
traj_vel=zeros((s(1,1)-1),3);
traj_acc=zeros((s(1,1)-1),3);
des_t=zeros((s(1,1)-1),1);
for a=1:(s(1,1)-1)

T=2*(sqrt((waypoints0(a+1,1)-waypoints0(a,1))^2+(waypoints0(a+
1,2)-waypoints0(a,2))^2+(waypoints0(a+1,3)-waypoints0(a,3))^2)
);

    A=[0 0 0 0 0 1;
       T^5 T^4 T^3 T^2 T 1;
       0 0 0 0 1 0;
       5*T^4 4*T^3 3*T^2 2*T 1 0;
       0 0 0 2 0 0;
       20*T^3 12*T^2 6*T 2 0 0];

    for b=1:3
       if (a==1)
          vi=0;
       else
          vi=(waypoints0(a+1,b)-waypoints0(a,b))/(T);
       end
       if (a==s(1,1)-1)
          vf=0;
       else
          vf=(waypoints0(a+2,b)-waypoints0(a+1,b))/(T);
       end
       D=[waypoints0(a,b);waypoints0(a+1,b);vi;vf;0;0];
       C=inv(A)*D;

traj(a,b)=C(1,1)*t^5+C(2,1)*t^4+C(3,1)*t^3+C(4,1)*t^2+C(5,1)*t
+C(6,1);

traj_vel(a,b)=C(1,1)*5*t^4+C(2,1)*4*t^3+C(3,1)*3*t^2+C(4,1)*2*
t+C(5,1);

traj_acc(a,b)=C(1,1)*20*t^3+C(2,1)*12*t^2+C(3,1)*6*t+C(4,1)*2;

    end
end
    if (t_index<=2)
       desired_state.pos = [traj(1,1);traj(1,2);traj(1,3)];
       desired_state.vel =
[traj_vel(1,1);traj_vel(1,2);traj_vel(1,3)];
       desired_state.acc =
[traj_acc(1,1);traj_acc(1,2);traj_acc(1,3)];
```

```
        desired_state.yaw = 0;
        desired_state.yawdot = 0;
    else
        x = t_index - 1;
        desired_state.pos = [traj(x,1);traj(x,2);traj(x,3)];
        desired_state.vel =
[traj_vel(x,1);traj_vel(x,2);traj_vel(x,3)];
        desired_state.acc =
[traj_acc(x,1);traj_acc(x,2);traj_acc(x,3)];
        desired_state.yaw = 0;
        desired_state.yawdot = 0;
    end

end
end
```