MODELING, SIMULATION AND COMPLETE CONTROL OF A QUADCOPTER

ME – 440 Major Project
Semester Report

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY in
MECHANICAL ENGINEERING

By

ABID SULFICAR 13ME107

HARIKRISHNAN SURESH 13ME141

ARAVIND VARMA 13ME216

ARJUN RADHAKRISHNAN 13ME217

Under the guidance of:

Prof. VIJAY DESAI

DEPARTMENT OF MECHANICAL ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL, MANGALORE – 575025

May, 2017

DEPARTMENT OF MECHANICAL ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
KARNATAKA
SURATHKAL, MANGALORE- 575025

CERTIFICATE

This is to certify that the U.G. project work report titled "**Modeling, Simulation, and Control of a Quadcopter**", submitted by

Abid Sulficar 13ME107

Harikrishnan Suresh 13ME141

Aravind Varma 13ME216

Arjun Radhakrishnan 13ME217

is the record of the work carried out by them, and is accepted as the *U.G. Project Work Report* submission in partial fulfilment of the requirements for the award of Bachelor of Technology Degree by the Department of Mechanical Engineering, National Institute of Technology Karnataka, Surathkal.

Prof. Vijay Desai	**Prof. Narendra Nath S**
Department of Mechanical Engineering	**Head, Department of Mechanical Engineering**
Project Guide	

DECLARATION

We hereby declare that the report of the U.G. project work titled "**Modeling, Simulation, and Control of a Quadcopter**", which is being submitted to the National Institute of Technology Karnataka Surathkal, for the award of Bachelor of Technology in Mechanical Engineering is a bonafide report of the work carried out by us. The material contained in this report has not been submitted to any University of Institution for the award of any degree.

Abid Sulficar	13ME107
Harikrishnan Suresh	13ME141
Aravind Varma	13ME216
Arjun Radhakrishnan	13ME217

Place: NITK, Surathkal

Date:

ACKNOWLEDGEMENT

We wish to express our sincere thanks to the people who extended their help during the course of this project.

We express our deepest gratitude to our guide Prof. Vijay Desai, Department of Mechanical Engineering, NITK Surathkal for providing us the opportunity to do our major project under his guidance. We thank him for the constant technical support and guidance which has immensely contributed to the successful completion of this project.

We thank our Head of the Department for providing the facilities necessary to undertake this project.

Contents

List of figures 7

List of tables 8

Nomenclature 9

Chapter 1 - Introduction 10

Chapter 2 - Literature Survey 12

Chapter 3 – Methodology 17

3.1 Mathematical model of quadcopter 17

3.2 Mathematical model of quadcopter with one failed rotor 22

3.3 Attitude controller 24

3.3.1 PID controller 24

3.3.2 Feedback linearization controller 26

3.3.3 Linear Quadratic Regulator 30

3.4 Trajectory planning using image processing 32

3.4.1 Feasible landing point 32

3.4.2 Trajectory planning algorithm 34

3.5 Trajectory Controller 35

Chapter 4 - Simulation 37

4.1. Simulation for attitude controller comparison 38

4.1.1. Attitude commands 38

4.1.2. Control gains 38

4.2. Simulation for trajectory following – feasible landing point 39

4.2.1. Path commands 39

4.2.2. Control gains 40

4.3. Simulation for trajectory following – using a trajectory planner 40

4.3.1. Path commands 40

4.3.2. Control gains 41

4.3.3. VR Simulation 41

Chapter 5 - Results and Discussions 43

5.1 Attitude controller comparison 43

5.2 Trajectory planner 47
5.2.1 Selecting the most feasible landing point 47
5.2.2 Trajectory Planning algorithm 48
5.3 Trajectory tracking 49
5.3.1 Trajectory tracking – Most feasible landing point 49
5.3.2. Trajectory tracking – traversing the maze 51
Further work 54
References 55
Appendix 56
Appendix A1: Quadcopter plant 56
Appendix A2: Quadcopter plant with a failed rotor 59
Appendix A3: Layout for PID controller 62
Appendix A4: Layout for FBL+PD controller 62
Appendix A5: Inside FBL blocks 63
Appendix A6: Layout of LQR 64
Appendix A7: Linriz.m function 64
Appendix A8: Inside image processing block 65
Appendix A9: Pathgen.m function 66
Appendix A10: Circle2.m function 67
Appendix A11: dijkstra.m function 69
Appendix A12: Functions called by Dijkstra.m 70
Appendix A13: pathcr.m function 71
Appendix A14: Trajectory controller 75
Appendix A15: Complete layout for trajectory control simulation 1 76
Appendix A16: Complete layout for trajectory control simulation 2 76

List of figures

Figure 1: Control architecture (taken from [3]) 13
Figure 2: Comparison between controllers on the basis of total error (taken from [4]) 14
Figure 3: Inertial and body frames of quadcopter (taken from [1]) 17
Figure 4: Quad plant Simulink Block 21
Figure 5: Block diagram of Attitude Controller 24
Figure 6: Desired path with switching 40
Figure 7: A VR model of quadcopter during simulation (viewpoint 1) 42
Figure 8: A VR model of quadcopter during simulation (viewpoint 2) 42
Figure 9: Step responses of controllers – Φ 43
Figure 10: Step responses of controllers – θ 44
Figure 11: Step responses of controllers – Ψ 45
Figure 12: Step responses of controllers –Z 46
Figure 13: Image containing different shapes in different colors 47
Figure 14: Black circle given as output 47
Figure 15 : Maze input to the trajectory planner 48
Figure 16 : Path generated by the trajectory planner 48
Figure 17: Actual path followed 49
Figure 18: Coordinate wise comparison of desired and actual paths 50
Figure 19: Plots of attitude variables vs Time 50
Figure 20: Desired path generated by trajectory planner 51
Figure 21: Actual path followed by quadcopter 52
Figure 22: Coordinate wise comparison of desired path and actual path 52
Figure 23: Plots of Attitude variables vs Time 53

List of tables

Table 1: Parameter values for quad plant 37
Table 2: Initial conditions for trajectory control simulation 1 37
Table 3: Initial conditions for trajectory control simulation 2 38
Table 4: Gain values for attitude controller 1 39
Table 5: Gain values for attitude controller 2 39
Table 6: Gain values for attitude controller 3 39
Table 7: Gain values for trajectory controller 40
Table 8: Gain values for trajectory controller 41
Table 9: Characteristic parameters to a step input for Φ 44
Table 10: Characteristic parameters to a step input for θ 44
Table 11: Characteristic parameters to a step input for Ψ 45
Table 12: Characteristic parameters to a step input for Z 46
Table 13: Computational time of controllers 46

Nomenclature

$\{O\}(O,X,Y,Z)$	Inertial frame
$\{B\}(O_B,X_B,Y_B,Z_B)$	Body frame
ε	Quadcopter position w.r.t {O}, m
η	Quadcopter Euler angles w.r.t {O}, rad
Φ	Roll angle, rad
θ	Pitch angle, rad
Ψ	Yaw angle, rad
P,Q,R	Angular velocities about X_B,Y_B,Z_B respectively, rad/s
V_X,V_Y,V_Z	Linear velocities about X_B,Y_B,Z_B respectively, m/s
C_D	Thrust coefficient of the motor
ω_i	Rotational speed of i^{th} rotor, rad/s
A	Cross-sectional area of the propeller's rotation, m^2
r	Radius of rotor, m
T_i	Thrust given by i^{th} rotor, N
A_x, A_y, A_z	Linear drag coefficients in the X,Y,Z respectively, N.s/m
M_Φ	Rolling moment, N.m
M_θ	Pitching moment, N.m
M_ψ	Yawing moment, N.m
L	Distance between the center of propeller and the center of quadcopter, m
B	Torque coefficient of motor
I_R	inertia moment of rotor, $kg.m^2$
A_r	Rotational drag coefficient, N.m.s
I	Inertia matrix, $kg.m^2$
m	Mass of quadcopter, kg
g	Acceleration due to gravity, m/s^2

Chapter 1 - Introduction

A Quadcopter is a rotor-based, unmanned aerial vehicle. Quad copters are becoming increasingly popular because of their small size and high maneuverability and find applications in diverse fields. The dynamics of a quadcopter is highly non-linear. Furthermore, it is an under-actuated system with six degrees of freedom and four control inputs. The thrust as well as the torques required for tilting the quadcopter are the control inputs which determine the motion of the vehicle. The thrust as well as torques are generated by adjusting the rotor speeds. The thrust generated by the rotor blades is always in the direction of the central axis of the quadcopter. Therefore, to achieve propulsion in a particular direction, the axis of quadcopter should be tilted with respect to the vertical. The translational motion of a quadcopter is hence coupled with its angular orientation, making quadcopter dynamics and control very complex.

Quad copters have applications in many fields, some of which are listed below.

- Reconnaissance – used to gather military intelligence by scouting enemy territory. Because of their small size and minimal noise generated, they can move undetected.
- Aerial surveillance – road patrol, home security, law and order. The ease of motion between points through air and large visibility of the surroundings, especially with a strong camera makes quad copters a prime candidate for aerial surveillance needs.
- Used in motion picture film making and photography for aerial shots and views.
- Assistance for search and rescue operations in disaster struck areas or in case of fire.
- Used in automation systems in industries for material handling purposes.
- Delivery of goods and items.
- Used for 3D modelling of terrains or large structures as well as thermal imaging.

Failure of quadcopter may occur due to many reasons such as

- Electronic Speed Control (ESC) burn out – ESC may burn out if the current exceeds the maximum permissible current. This causes the propeller associated with the ESC to stop spinning and can lead to failure.
- Damage to motor, whether due to physical damage or exceeding the maximum current value can cause the motor to stop functioning mid-flight or may cause loss in efficiency.
- Any physical damage to the propeller blades such as dents, nicks, cuts etc. can cause vibrations and may cause the propeller blade to come off mid-flight.

- Bending of propeller blades can lead to a decrease in lift and increased noise during operation leading to a decrease in efficiency.

The loss of quadcopter propeller blades can cause the quadcopter to crash. Apart from the monetary loses associated with the damage to quadcopter parts, it can have many negative consequences. Loss of a quadcopter used for reconnaissance work can lead to loss of valuable military intelligence and causes the risk of it being discovered by the enemy. In motion picture film making, thermal imaging photography etc., the equipment mounted on the quadcopter are very costly and propeller failure can lead to the damage of valuable equipment. In search and rescue operations in disaster affected regions, the failure of the quadcopter can lead to possible delays, increasing the risk on the life of affected people. In material handling systems, failure may lead to damage of costly parts. Added to all this, there is also the risk of the quadcopter crashing on to people and causing injuries especially in public spaces.

The potential risk of loss of a propeller is high in many of the situations in terms of cost as well as other factors. Hence, a mechanism for the control of a quadcopter against the possibility of failure is a necessity. Quadcopter control, even with four propellers functioning, is a complex problem because it is an under-actuated and highly non-linear system. Various control algorithms like PID and feedback linearization are used for the purpose of control. The control problem becomes more complex when there is complete loss of one or more propellers. The first stage in devising an algorithm for such a case is fault detection. If left unchecked, the fault leads to the failure of the system. In this case, the loss of a propeller is the fault. After detecting the nature of the fault and the component in which the fault has occurred, the next stage is replacing the model of the system with a new model which takes into account the effect of the fault. The original model does not accurately represent the behavior of the system in the event of a fault. Finally, a controller should be devised to control the system represented by the new model. The controller used for this purpose should ensure that the quadcopter stays in flight regardless of the failure of a rotor and that its motion could be controlled sufficiently enough to land it safely on a desired location in the vicinity.

The quadcopter has to maneuver through obstacles at times. Image processing is used for this purpose in this project. One variant of the model involves generating a path through a maze. The image (resolution: 50x50) of the maze is fed into MATLAB as a 50x50x3 array. Each pixel containing an obstacle is assigned a high cost. The cost assigned to the pixel reduces as the quadcopter move away from the obstacles. The aim is to move from the source to the destination while minimizing the cost. The path thus obtained would not only be devoid of obstacles, but also be at a safe distance from them.

Another variant involves scanning the image for safe landing points in case failure occurs. Given the landing points, the one nearest to the quadcopter at the time of failure could be found out using a simple distance formula. But this may not always be the most suitable landing point. The velocity of the quadcopter when the failure occurs should also be taken into account. Instead of going to the nearest landing point, a better alternative is to move to the one which can be reached in the shortest time.

Chapter 2 - Literature Survey

The employment of quadcopters in challenging applications like rescue, surveillance comes from its ability to perform aggressive maneuvers and follow complex trajectory in 3D space. For these applications, precise angle handling of quadcopters is important. This calls for a clear understanding of the system dynamics before designing a controller to achieve the purpose.

Mathematical modeling is the first and most critical step towards understanding the system dynamics and monitoring the response. The differential equations governing the quadcopter dynamics derived using the two most popular approaches (Newton-Euler equations and Euler-Lagrange equations) is presented in [1]. While the derivations are listed for a simplified model, the paper also presents a more realistic model for the quadcopter with the inclusion of the drag force caused by air resistance. Other complex dynamic interactions like aerodynamic effects and blade flapping have been neglected due to challenges in modeling. A matrix approach for the derivation of governing equations is elaborated in [2]. The concept of mixed frame of reference for describing the state variables along with its convenience in developing the mathematical model is described. A more detailed study of the dominant aerodynamic effects on the quadcopter is addressed in [3]. The derivation for exogenous forces on the body of quadcopter, considering even the complex aerodynamic phenomena neglected in the simplified models used in [1] and [2] is also given. The steady state thrust and reaction torque (due to rotor drag) for a hovering rotor in free air is modelled using momentum theory. The lumped parameter approximation used for thrust and reaction torque expressions is shown here, with the constant value obtained from static thrust tests. The dynamic model considering flapping dynamics and rotor stiffness for induced drag is presented here, though these terms are minor considerations from robotics perspective. It is mentioned that high gain control can dominate all the secondary aerodynamic effects, and high performance control can be achieved using the simple static thrust model.

Different control methods for attitude stabilization have been researched, including PID controllers ([1], [3], [5] and [8]), back-stepping controller [4], sliding mode controller ([4] and [12]), linear quadratic regulator ([8] and [10]) and feedback linearization control ([9], [10], [11] and [12]). A hierarchical control approach is implemented in [3], with nested feedback loops as shown in the figure. The control problem is decoupled into position controller and attitude controller, with the position controller providing the set-points to the attitude controller.

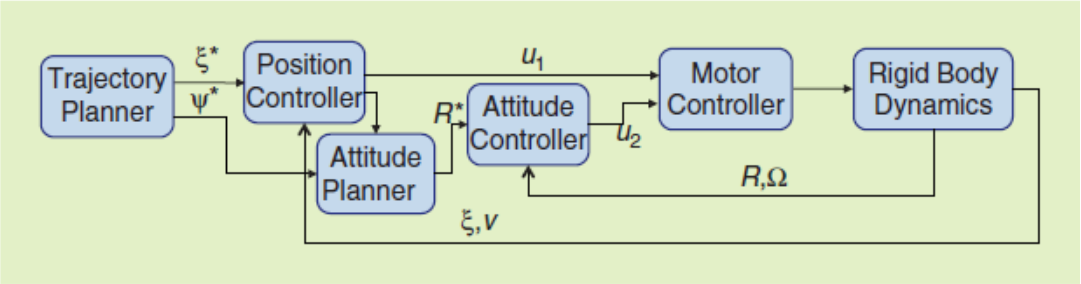


Figure 1: Control architecture (taken from [3])

An exponentially converging attitude controller is presented considering the measure of error in rotations. A skew-symmetric matrix is generated to go from actual attitude vector to the desired attitude vector. For small deviations from the hover position, the error matrix is linearized and a PD controller gives satisfactory performance. The control law for large deviations is also given, where the error matrix is not linearized. A much simpler PD controller is implemented in [1], where the expression for thrust and torque components are obtained from the error in attitude values, and the individual rotor speeds are calculated from the thrust and torque commands. Here, the motor dynamics is neglected while deriving the force and torque expressions from the motor-propeller system. A PID controller with an additional term for angular acceleration feedback is used for attitude control in [5]. This additional term allows the gains to significantly increase, thereby yielding higher bandwidth. A first order time delay in thrust is also included in the model for controlling each angle.

Among the nonlinear controllers, the most popular approach is feedback linearization control which in turn has two approaches. Both the approaches are discussed in [10] - Exact linearization and non-interacting control via dynamic feedback and Dynamic inversion with zero-dynamics stabilization. The former approach involves the use of dynamic feedback control law, and the nonlinear system cannot be solved using static feedback control. With the position variables chosen as output function, the thrust input is delayed till its second derivative and the system is extended to include the thrust input and its first derivative as the system states. The extended system fulfills the condition for feedback linearization and can be transformed via dynamic feedback into a system which is fully Linear and controllable. The latter approach uses attitude variables as the output variables, and dynamic inversion is carried out with small angle approximation. Here, the attitude variables are differentiated till the input terms appear and the system is not extended. The paper also shows the implementation of Linear Quadratic Regulator (LQR) on the quadcopter, where model linearization is performed using small angle approximation, and the LQR function in Matlab is used to solve the algebraic equation using Riccati's method and obtain the control gains.

Feedback linearization with position variables and yaw angle as output terms of interest is also shown in [12], where the equations are differentiated twice till the input terms appear. Repeated differentiation of dynamic equations causes it to be very sensitive to noise, along with high cost of

computation. Here, the computation is reduced by assuming small angle approximation which also lowers the extent of nonlinearity in the system. Feedback linearization by dynamic inversion is discussed in [10] and [11], where the attitude variables are considered as outputs of interest. A two layer architecture is adopted for structured tracking, with a dynamic inversion inner loop and an internal dynamics stabilization outer loop. Dynamic inversion is carried out with small angle approximations for the Euler angles, which gives a simplified expression for the matrix to be inverted. A back stepping approach is used in [10] to design the linear controller for the linearized dynamics and the residual dynamics. The paper also presents a detailed stability analysis for the two controllers. However, in [11] the traditional PD controller is used to provide the linear control inputs in the inner loop and a PID is designed to perform trajectory following along with internal dynamics stabilization.

A comparative study between these different controllers for attitude stabilization and control is addressed in [4]. The controllers are applied to the system and analyzed separately to find the optimum controller for the quadcopter. As seen in the figure where total error is used to evaluate the performance of different controllers, the sliding mode technique proves to be the superior to other techniques. But the PD controller shows reasonable performance compared to the sliding mode technique, and as its implementation is much easier most of the literature contains PD controller in the model. Inverse control based on feedback linearization is also easy to implement, with better settling time compared to PID and sliding mode controllers though the response is slower.

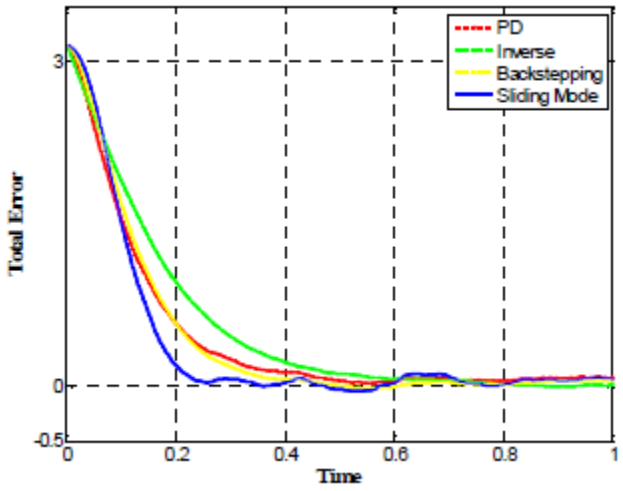


Figure 2: Comparison between controllers on the basis of total error (taken from [4])

A Comparison between PID control and LQR control is given in [8]. Here, multiple PID controllers are designed for a near hover condition neglecting the gyroscopic effects. In order to implement the LQR, the system is linearized around each state. Linearization around the equilibrium point ignoring the gyroscopic effects causes a huge drift from reality, and is avoided.

Feedback linearization controller and sliding mode controllers have superior performance compared to PID as seen in [4], and a detailed analysis between these two controllers is performed in [12]. While feedback linearization controller is simpler to implement, uncertainty in the dynamic model can severely affect performance, and even cause instability. In addition, the dependence on higher derivative terms of states makes it highly sensitive to external disturbances. The sliding mode controller is a more robust approach which compensates for model uncertainties and external disturbances. However, handling these uncertainties causes very high input gains and is a serious problem in power-limited systems like mini quadcopters. Feedback linearization controllers also use more efficient inputs without chattering, compared to sliding mode controller.

All the above papers that implemented feedback linearization (both approaches) have used small angle approximation. This may hold well in near hover conditions, but for a quadcopter involved in trajectory following or sometimes complex motions will have the Euler angles reaching high values. So, a more general approach for feedback linearization is required, which will be addressed through this project.

In addition to the controller, the motor dynamics and their interactions with the drag forces on the propellers is also modelled in [3], with a first order linear approximation. As rotor speed drives the dynamic model, high-quality control of the motor speed is critical for the overall control of the vehicle. Direct voltage control is sufficient in most cases, as the steady state motor speed is directly proportional to the voltage supplied. The performance of the controller is limited by the amount of current that can be supplied by the batteries. Assuming that the battery is able to supply the required current for all rotor speeds and current levels do not exceed the limiting value at any stage, the motor controller can be removed from the system.

Trajectory tracking is a widely studied problem for quadcopters. A lot of literature has focused on nonlinear methods like input-output linearization using differential flatness theory and back-stepping control which enables acrobatic maneuvers. For normal trajectory tracking, such approaches are not necessary. Trajectory control in [1] is done using a heuristic approach to generate a symmetric function for jounce to control the acceleration and in turn determine the control inputs to achieve the desired trajectory. A PD controller is integrated into the heuristic method for better response to the disturbances and to stabilize the quadcopter during its trajectory tracking. In [3], the dynamics is linearized about the desired trajectory. An acceleration vector command is computed to minimize the error in the trajectory. The commands for roll and pitch are then calculated based on this acceleration vector and desired yaw angle and fed to the attitude controller. A position controller without linearization is also addressed, where the position error is projected along the yaw axis. A trajectory planning algorithm is discussed in [3], where the problem is simplified by assuming differential flat theory for quadcopters that respect the dynamics of the under actuated system. The trajectory is generated by minimizing a cost function derived from jounce and yaw accelerations. Hence, real time planning is carried out to generate the optimal path. Another trajectory planning algorithm is illustrated in [5]. Here, a sequence of desired

waypoints is inputted and a dynamically feasible trajectory is generated that traverses the waypoints in minimum time while satisfying acceleration and velocity constraints. The controller consists of a piecewise PI control in the along direction, and PID in the cross track direction which provides the respective control inputs to follow the desired path. The controllers mentioned above require derivatives of the desired path to be given as input, which limits aggressive maneuvers of the quadcopter. This calls for a trajectory controller that does not depend on the higher order derivatives and can enable complex motions of the quadcopter.

Based on the literature survey, the following objectives were established for this project:

1. Implement a linear and nonlinear controller for attitude control of a quadcopter, and perform a comparative study for the same.
2. Integrate a trajectory controller into the attitude controller, to follow the path commands given by the trajectory planner to the nearest landing point. Determine the coordinates of the landing points in the inputted map using image processing and identify the landing site nearest to the point of failure.
3. Develop a trajectory planning algorithm that helps traverse a maze without hitting the walls and test the trajectory controller for this complex trajectory.
4. Develop a failure detection module for the quadcopter to enable switching of controllers in order to ensure stability of the system.

Chapter 3 – Methodology

3.1 Mathematical model of quadcopter

The kinematics and dynamics of a quadcopter can be clearly understood by considering two frames of reference: earth inertial frame { O } and body fixed frame { B }. The earth inertial frame is defined with gravity pointing in the negative z direction, and the coordinate axes of the body frame are along the arms of the quadcopter. 4 DC motors are placed at the extremities of all the arms, with a propeller mounted on them to provide the required thrust. In the structure shown in figure 3, Motors 1 and 3 rotate in the counter-clockwise direction with angular velocities ω_1 and ω_3, whereas motors 2 and 4 rotate in the clockwise direction ω_2 and ω_4.

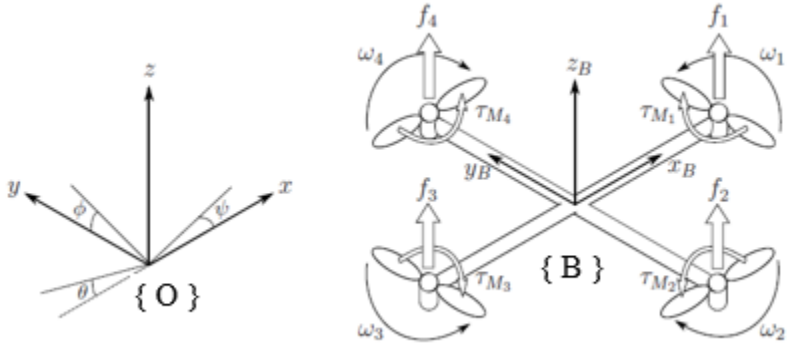


Figure 3: Inertial and body frames of quadcopter (taken from [1])

The absolute position of the center of mass of the quadcopter is expressed in the inertial frame as $\varepsilon = [\, X \;\; Y \;\; Z \,]^{\,T}$. The attitude or the angular position is defined in the inertial frame with the 'roll-pitch-yaw' Euler angles $\eta = [\, \Phi \;\; \theta \;\; \Psi \,]^{\,T}$. The linear velocities $V_B = [\, V_X \;\; V_Y \;\; V_Z \,]^{\,T}$ and angular velocities $v = [\, P \;\; Q \;\; R \,]^{T}$ are defined in the body frame.

The relationship between these two frames is expressed using the rotation matrix R_1

$$R_1 = \begin{bmatrix} C_\Psi C_\theta & C_\Psi S_\theta S_\Phi - S_\Psi C_\Phi & C_\Psi S_\theta C_\Phi + S_\Psi S_\Phi \\ S_\Psi C_\theta & S_\Psi S_\theta S_\Phi + C_\Psi C_\Phi & S_\Psi S_\theta C_\Phi - C_\Psi S_\Phi \\ -S_\theta & C_\theta S_\Phi & C_\theta C_\Phi \end{bmatrix} \qquad (3.1)$$

Where $C_\theta = cos(\theta)$ and $S_\theta = sin(\theta)$. R_1 is orthogonal, which implies $R^{-1} = R^T$ which is the rotation matrix from { B } to { O }.

Since all the motors are identical, the derivation is explained for a single one. The thrust acting on the quadcopter by a single motor-propeller system is given by momentum theory:

$$T_i = C_D \rho A r^2 \omega_i^2 \quad (3.2)$$

Where C_D is thrust coefficient of the motor, ρ is the density of air, A is the cross-sectional area of the propeller's rotation, r is the radius of rotor and ω_1 is the angular speed of the rotor. For simple flight motion, a lumped parameter approach is considered to simplify the above equation to

$$T_i = K\omega_i^2 \quad (3.3)$$

Combining the thrust from all the 4 motor-propeller system, the net thrust in the body frame Z direction is given by:

$$T = K \sum \omega_i^2 \quad (3.4)$$

Therefore, the net thrust acting on the quadcopter in the body frame is:

$$F^B = [0 \quad 0 \quad T]^T \quad (3.5)$$

In addition to thrust, a drag force also acts on the quadcopter which is a resisting force. It has components along the coordinate axes in the inertial frame directly proportional to the corresponding velocities. The drag force is given in the component form as

$$F^D = \begin{bmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \quad (3.6)$$

where A_x, A_y and A_z are the drag coefficients in the x, y and z directions.

If all the rotor velocities are equal, the quadcopter will experience a force in z direction will move up, hover or fall down depending on the magnitude of the force relative to gravity. The moments acting on the quadcopter cause pitch, roll and yaw motion. Pitching moment M_Φ occurs due to difference in thrust produced by motors 2 and 4. Rolling moment M_θ occurs due to difference in thrust produced by motors 1 and 3.

$$M_\Phi = L(T_4 - T_2) \quad (3.7)$$

$$M_\theta = L(T_3 - T_1) \quad (3.8)$$

in which L is the distance between the center of propeller and the center of quadcopter.

Yawing moment $M\Psi$ is caused by the drag force acting on all the propellers and opposing their rotation. Again from the lumped parameter approach,

$$\tau_{Mi} = B\omega_i^2 + I_R\dot{\omega}_i \qquad (3.9)$$

where τ_{M1} is the torque produced by motor 1, B is the torque constant, I_R is the inertia moment of rotor. The effect of $\dot{\omega}_1$ is very small and can be neglected.

$$M_\psi = B(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2) \qquad (3.10)$$

The rotational moment acting on the quadcopter in the body frame is:

$$M^B = [M_\Phi \quad M_\theta \quad M_\psi]^T \qquad (3.11)$$

There is also a rotational drag which is a resistive torque that acts on the body frame which is proportional to the body from angular velocities. The rotational drag is given by:

$$M^R = [A_r P \quad A_r Q \quad A_r R]^T \qquad (3.12)$$

where A_r is the rotational drag coefficient.

The model presented here has been simplified by ignoring several complex effects like, blade flapping (deformation of blades at high velocities and flexible materials), surrounding wind velocities etc.

Newton-Euler formulation is used to derive the dynamic equations of motion for the quadcopter. The quadcopter is assumed to have a symmetrical structure, so the Inertia matrix is diagonal and time-invariant, with $I_{XX} = I_{YY}$.

$$I = \begin{bmatrix} I_{XX} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \qquad (3.13)$$

In the body frame, the force producing the acceleration of mass $m\,\dot{V}_B$ and the centrifugal force $v \times (m\,V_B)$ are equal to the gravity $R^T G$ and the total external thrust F^B and the aero dynamical drag force $R^T F^D$.

$$m\,\dot{V}_B + v \times (m\,V_B) = R^T G + F^B - R^T F^D \qquad (3.14)$$

In the case of a quadcopter, it is convenient to express the dynamics with respect to a mixed frame { M } with the translational dynamics with respect to the inertial frame and { O } and the rotational dynamics with respect to the body frame { B }.

In the inertial frame, centrifugal effects are negligible. The only forces coming into play are the gravitational force, thrust, drag and acceleration of the mass of quadcopter.

$$m\,\ddot{\varepsilon} \;=\; G + R\,F^{B} - F^{D} \tag{3.15}$$

Rewriting this,

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \begin{bmatrix}0\\ 0\\ -g\end{bmatrix} + R\,\frac{F^{B}}{m} - \frac{F^{D}}{m} \tag{3.16}$$

Making the following substitution and taking the component form gives the dynamic equation for translational motion:

$$\begin{bmatrix}X\\ \dot{Y}\\ \dot{Z}\end{bmatrix} = \begin{bmatrix}U\\ V\\ W\end{bmatrix} \tag{3.17}$$

$$\dot{U} = \;(\sin\Phi\sin\Psi + \;\cos\Phi\sin\theta\cos\Psi)\frac{T}{m} - \frac{A_x}{m}U \tag{3.18 a}$$

$$\dot{V} = \;(-\sin\Phi\cos\Psi + \;\cos\Phi\sin\theta\sin\Psi)\frac{T}{m} - \frac{A_y}{m}V \tag{3.18 b}$$

$$\dot{W} = \;-g + (\cos\Phi\cos\theta)\frac{T}{m} - \frac{A_z}{m}W \tag{3.18 c}$$

Again, considering the rotational dynamics in the body frame, the angular acceleration of the inertia $I\,\dot{\nu}$, the centripetal forces $\nu \times (I\,\nu)$ and the gyroscopic forces τ are equal to the external torque M^{B} and the torque generated due to aero dynamic drag.

$$I\,\dot{\nu} + \;\nu \times (\,I\,\nu\,) \;+\; \tau \;=\; M^{B} - M^{D} \tag{3.19}$$

Rewriting this equation,

$$\dot{\nu} = I^{-1}\left(-\begin{bmatrix}P\\ Q\\ R\end{bmatrix} \times \begin{bmatrix}I_{XX}P\\ I_{YY}Q\\ I_{ZZ}R\end{bmatrix} - I_{R}\begin{bmatrix}P\\ Q\\ R\end{bmatrix} \times \begin{bmatrix}0\\ 0\\ 1\end{bmatrix}\Omega + \;M^{B} - M^{D}\right) \tag{3.20}$$

Where Ω = - ω_1 + ω_2 - ω_3 + ω_4 and I_R is the rotational inertia of each motor.

Writing in component form,

$$\dot{P} = \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) QR - \frac{I_R}{I_{XX}} Q\Omega + \frac{M_\Phi}{I_{XX}} - \frac{A_r}{I_{xx}} P \quad (3.21\text{ a})$$

$$\dot{Q} = \left(\frac{I_{ZZ} - I_{XX}}{I_{YY}}\right) PR - \frac{I_R}{I_{YY}} P\Omega + \frac{M_\theta}{I_{YY}} - \frac{A_r}{I_{yy}} Q \quad (3.21\text{ b})$$

$$\dot{R} = \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) PQ + \frac{M_\Psi}{I_{ZZ}} - \frac{A_r}{I_{zz}} R \quad (3.21\text{ c})$$

The transformation of angular velocities from body frame to inertial frame is given by:

$$\begin{bmatrix} \dot{\Phi} \\ \dot{\theta} \\ \dot{\Psi} \end{bmatrix} = \begin{bmatrix} 1 & sin\,\emptyset\,tan\,\theta & cos\,\emptyset\,tan\,\theta \\ 0 & cos\,\emptyset & -sin\,\emptyset \\ 0 & \frac{sin\,\emptyset}{cos\,\theta} & \frac{cos\,\emptyset}{cos\,\theta} \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} \quad (3.22)$$

Using the complete equations of motion describing the dynamics of the system, a quadcopter plant is created in Simulink using the Level-2 S-Function block. For the plant, the 4 rotor speeds $\{\omega_1, \omega_2, \omega_3, \omega_4\}$ are considered as the inputs and the 12 states in the mixed frame $\{U, V, W, P, Q, R, \Phi, \theta, \Psi, X, Y, Z\}$ as the outputs.

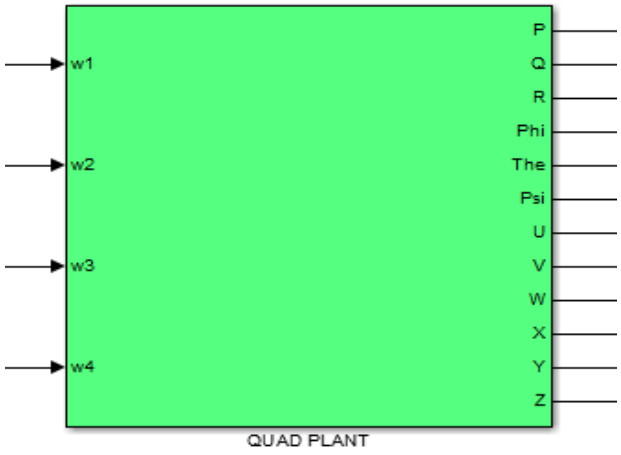


Figure 4: Quad plant Simulink Block

Using the equation of thrust and torque mentioned in dynamics of quadcopter and the above four equations of thrust and torques, the values of the angular velocities at each of the four rotors are obtained and shown in Equation 3.25.

$$\begin{pmatrix} T \\ m_\phi \\ m_\theta \\ m_\psi \end{pmatrix} = \begin{pmatrix} K & K & K & K \\ 0 & KL & 0 & -KL \\ -KL & 0 & KL & 0 \\ -B & B & -B & B \end{pmatrix} \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix} \tag{3.23}$$

Taking inverse of the above matrix, the following equations are obtained for individual rotor speeds:

$$\omega_1^2 = \frac{T}{4k} - \frac{m_{\boldsymbol{\theta}}}{2kl} - \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.24 a}$$

$$\omega_2^2 = \frac{T}{4k} - \frac{m_{\boldsymbol{\phi}}}{2kl} + \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.24 b}$$

$$\omega_3^2 = \frac{T}{4k} + \frac{m_{\boldsymbol{\theta}}}{2kl} - \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.24 c}$$

$$\omega_4^2 = \frac{T}{4k} + \frac{m_{\boldsymbol{\phi}}}{2kl} + \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.24 d}$$

These ω are then used to calculate the current states of the quadcopter as mentioned before. This model is used to develop both the attitude and trajectory controller along with implementation of the trajectory planner. The model described in the following session is a special case, and is used only to test the failure detection module (Section 3.6).

3.2 Mathematical model of quadcopter with one failed rotor

The dynamics of the quadcopter remains almost the same in the case of failure of one rotor. However, the control problem becomes increasingly complex. It becomes impossible to control the full attitude of a quadcopter with only three functional rotors. Without loss of generality, it can be assumed that the failed rotor is rotor number 2. This means that the torque control input M_Φ is lost as the torque can now be provided only in one direction. So the spinning of rotor 4 now creates an unbalanced torque. In order to avoid the toppling of the quad, the rotor 4 velocity must be minimized and this creates an unbalance in yaw. Any attempt to maintain the quad in hover in such a case implies that the yaw control must be relinquished. The total number of control inputs reduces to three. This changes only the right-hand side of the dynamical equations which deals with external driving forces and torques which are the control inputs.

With rotor 2 encountering failure, $\omega_2 = 0$. Therefore net thrust is given by

$$T = K(\omega_1^2 + \omega_3^2 + \omega_4^2) \tag{3.25}$$

and thrust due to each rotor is $T_i = K\omega_i^2$, i = 1,3,4, and the torques are given by

$$M_\Phi = L * T_4 \tag{3.26}$$

$$M_\theta = L(T_3 - T_1) \tag{3.27}$$

$$M_\psi = B(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2) \tag{3.28}$$

As M_Φ is no longer a control input, the equation for *P* is modified to express M_Φ in terms of the other control inputs. The remaining equations stay the same. The modified equations governing the rotational dynamics are given below

$$\dot{P} = \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) QR - \frac{I_R}{I_{XX}} Q\Omega + \frac{0.5l\left(T - \frac{M_\psi}{d}\right)}{I_{XX}} - \frac{A_r}{I_{xx}} P \tag{3.29 a}$$

$$\dot{Q} = \left(\frac{I_{ZZ} - I_{XX}}{I_{YY}}\right) PR - \frac{I_R}{I_{YY}} P\Omega + \frac{M_\theta}{I_{YY}} - \frac{A_r}{I_{yy}} Q \tag{3.29 b}$$

$$\dot{R} = \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) PQ + \frac{M_\Psi}{I_{ZZ}} - \frac{A_r}{I_{zz}} R \tag{3.29 c}$$

The angular velocities of rotors required for desired values of control inputs are given by:

$$\begin{bmatrix} T \\ M_\theta \\ M_\psi \end{bmatrix} = K \begin{bmatrix} 1 & 1 & 1 \\ -l & -l & 0 \\ -d & -d & d \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{3.30}$$

Taking the inverse, we get the equations for the three rotor speeds as

$$\omega_1^2 = \frac{T}{4k} - \frac{m_{\boldsymbol{\theta}}}{2kl} - \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.31 a}$$

$$\omega_3^2 = \frac{T}{4k} + \frac{m_{\boldsymbol{\theta}}}{2kl} - \frac{m_{\boldsymbol{\psi}}}{4b} \tag{3.31 b}$$

$$\omega_4^2 = \frac{T}{2k} + \frac{m_{\boldsymbol{\psi}}}{2b} \tag{3.31 c}$$

3.3 Attitude controller

A quadcopter consists of mainly six outputs of interest ($\Phi, \theta, \Psi, X, Y, Z$) with only four control inputs. This is solved by decoupling it into two distinct control loops (figure 5), inner loop dealing with the attitude variables and the outer variable dealing with the position variables. The angular motion of the quadcopter does not depend on the translational components, whereas the translational motion depends on the Euler angles. So the aim is to first control the rotational behavior due to its independence and then control the translational behavior.

Controlling vehicle attitude requires sensors to measure vehicle orientation, actuators to apply the torques needed to re-orient the vehicle to a desired attitude, and algorithms to command the actuators based on (1) sensor measurements of the current attitude and (2) specification of a desired attitude. Once the attitude control is designed and optimized, it can be integrated with the trajectory controller.

The block diagram for attitude controller is as shown below:

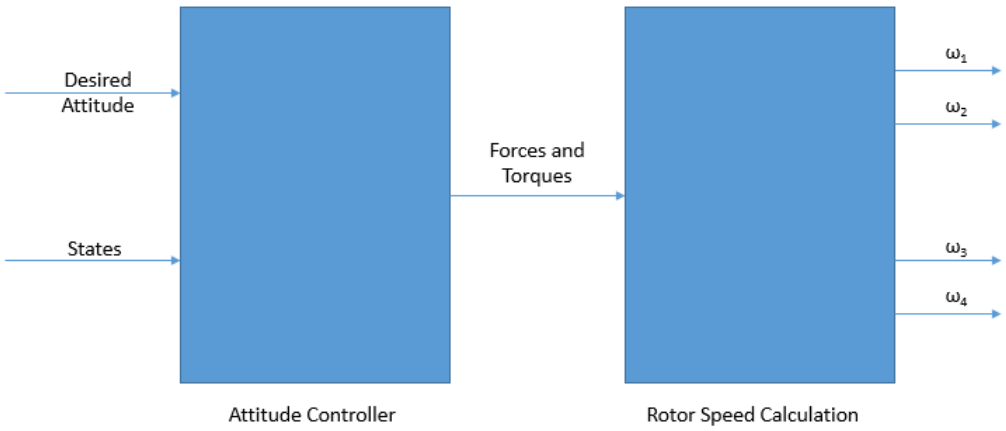


Figure 5: Block diagram of Attitude Controller

A lot of different methods have been studied to achieve autonomous flight, from which three methods (both linear and nonlinear) are discussed below. A comparative study is carried out to identify the most optimal controller for attitude stabilization.

3.3.1 PID controller

Of all the controllers, a PID Controller is the easiest to implement. The general form of PID controller is

$$e(t) = x_d(t) - x(t) \tag{3.32 a}$$

$$u(t) = K_P e(t) + K_I \int e(\tau)d\tau + K_D \frac{d}{dt} e(t) \tag{3.32 b}$$

Where $u(t)$ is the control input and $e(t)$ is the error between desired state $x_d(t)$ and present state $x(t)$, and K_P, K_I and K_D are the parameters for the proportional, integral and derivative elements of the PID controller. The desired values of attitude are fed from an attitude command block.

The standard PID control technique is applied on the nonlinear system directly, with an individual PID block for each attitude variable to control it independently. This does not require the model to be linearized about the hover condition, and can thus stabilize the quadcopter on the advent of strong perturbations.

A phi controller is basically an attitude controller which is used to control the attitude of the quadcopter about the X axis. Using this controller, the ϕ angle is stabilized. It can also be used to set ϕ to a particular value, which would help in the motion of the quadcopter in Y direction. Similarly, the theta controller and psi controllers are used to stabilize θ and ψ attitudes of the quadcopter. The corresponding torques are calculated using the following equations:

$$m_\phi = I_{xx} \left(K_{\phi,D} \dot{e_\phi}(t) + K_{\phi,P} e_\phi(t) + K_{\phi,I} \int e_\phi(t)dt \right) \tag{3.33 a}$$

$$m_\theta = I_{yy} (K_{\theta,D} \dot{e_\theta}(t) + K_{\theta,P} e_\theta(t) + K_{\theta,I} \int e_\theta(t)dt) \tag{3.33 b}$$

$$m_\psi = I_{zz} (K_{\psi,D} \dot{e_\psi}(t) + K_{\psi,P} e_\psi(t) + K_{\psi,I} \int e_\psi(t)dt) \tag{3.33 c}$$

Where $e_\phi(t) = \phi_d(t) - \phi(t)$, $e_\theta(t) = \theta_d(t) - \theta(t)$ and $e_\psi(t) = \psi_d(t) - \psi(t)$.

The Z controller is used to stabilize the altitude of the quadcopter to a desired value. Similar to the angular attitude controllers, this controller also employs a PID to control the altitude. Here, the thrust required is calculated using the following equation:

$$T = mC_\theta C_\phi \left[g + K_{z,D} \dot{e_z}(t) + K_{z,P} e_z(t) + K_{z,I} \int e_z(t)dt \right] \tag{3.34}$$

Where $e_z(t) = z_d(t) - z(t)$.

Since thrust is calculated in the body frame while g and other PID terms are in the inertial frame, a rotational matrix is applied to the terms in the inertial frame which is given by $C_\theta C_\phi$ where C stands for cos function.

3.3.2 Feedback linearization controller

Feedback linearization control is a popular nonlinear control approach, where the nonlinear system is algebraically transformed into (fully or partly) linear ones by cancelling the nonlinearities. Most feedback linearization techniques are based either on input-output linearization or input-state linearization. We have adopted the input-output linearization in our work. Input-Output linearization involves the repeated differentiation of the output variables till the input term appears, the last derivative being the r^{th} one. This will help in obtaining a mapping between the transformed inputs and the outputs. We use the concept of dynamic inversion to the system given by [], which yields the inner loop that feedback linearizes the system from the control input to the output. The output variables not considered above is called residual or internal dynamics. Dynamic inversion need not necessarily yield the internal dynamics stable, which will then require another outer stabilizing loop.

Consider a SISO (Single Input Single Output) system with state x, input u and output y whose dynamics are given by

$$\dot{x} = f(x) + g(x)u \tag{3.35}$$

$$y = h(x) \tag{3.36}$$

The derivative of the output y can be expressed as

$$\dot{y} = \frac{\partial h}{\partial x}[f(x) + g(x)u] \tag{3.37 a}$$

The derivative of h along the trajectory of the state x is known as the Lie Derivate and equation (3.41) can be written in terms of lie derivative as

$$\dot{y} = \frac{\partial h}{\partial x}[f(x) + g(x)u] = L_f h(x) + L_g h(x)u \tag{3.37 b}$$

The output y is differentiated continuously until input terms appear in the differential equation. Generally, the i^{th} derivative of y is expressed in terms of lie derivative as

$$y^{(i)} = L_f^i h(x) + L_g L_f^{i-1} h(x) u \tag{3.38}$$

The above equation can be linearized through dynamic inversion, choosing u as

$$u = \frac{1}{L_g L_f^{i-1} h(x)} \left[-L_f^i h(x) + v \right] \tag{3.39}$$

This yields the simple output equation

$$y^i = v \tag{3.40}$$

The concepts used for the SISO systems can be extended to MIMO systems. In particular, we consider square systems having the same number of inputs and outputs. Suppose that an input term first appears in the rth derivative of the ith output. Then, the equation for the ith output is expressed as

$$y_i^{(r_i)} = L_f^{r_i} h_i(x) + \sum_{j=1}^{m} L_{g_j} L_f^{r_i - 1} h_i(x) u_j \tag{3.41}$$

The set of differential equations that corresponds to the input – output relations may be expressed in the matrix form as

$$\begin{bmatrix} y_i^{(r_i)} \\ \vdots \\ y_m^{(r_m)} \end{bmatrix} = \begin{bmatrix} L_f^{r_1} h_1(x) \\ \vdots \\ L_f^{r_m} h_m(x) \end{bmatrix} + \boldsymbol{E}(x) \begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix} \tag{3.42}$$

Where E(x) is an m x m coefficient matrix of the inputs.

This set of equations can be converted to simple linear equations for the outputs by the following input transformation

$$\boldsymbol{u} = -\boldsymbol{E}^{-1} \left[\begin{bmatrix} L_f^{r_1} h_1(x) \\ \vdots \\ L_f^{r_m} h_m(x) \end{bmatrix} + \boldsymbol{v} \right] \tag{3.43}$$

This yields equations of the form

$$y_i^{(r_i)} = v_i \tag{3.44}$$

Thus, the inversion-based control law has the capability in shaping the output response by simply designing the new controls v_i to get the desired output.

To implement this controller, we assume the system is decomposed into two sub models – *M1* with states $\boldsymbol{X_1} = [Z, \Phi, \theta, \Psi, W, P, Q, R]$ and *M2* with states $\boldsymbol{X_2} = [X, Y, U, V]$.

Consider the inner loop with the sub model *M1* and output variables

$$Y_1 = [Z\ \Phi\ \theta\ \Psi]^T \tag{3.45}$$

The first derivative w.r.t time does not contain input terms, as evident from the following equations. The expressions are obtained from Equations 3.17 and 3.22.

$$\begin{bmatrix} \dot{Z} \\ \dot{\Phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & \sin\emptyset\tan\theta & \cos\emptyset\tan\theta \\ 0 & 0 & \cos\emptyset & -\sin\emptyset \\ 0 & 0 & \frac{\sin\emptyset}{\cos\theta} & \frac{\cos\emptyset}{\cos\theta} \end{bmatrix} \begin{bmatrix} W \\ P \\ Q \\ R \end{bmatrix} \tag{3.46}$$

The transformation matrix is denoted by $MatW$.

Differentiating this again gives:

$$\ddot{Y}_1 = \begin{bmatrix} \ddot{Z} \\ \ddot{\Phi} \\ \ddot{\theta} \\ \ddot{\Psi} \end{bmatrix} = \frac{d}{dt}\,(MatW) * \begin{bmatrix} W \\ P \\ Q \\ R \end{bmatrix} + MatW * \begin{bmatrix} \dot{W} \\ \dot{P} \\ \dot{Q} \\ \dot{R} \end{bmatrix} \tag{3.47}$$

The expressions of $\dot{W}$, $\dot{P}$, $\dot{Q}$, and $\dot{R}$ contain input terms are obtained from Equations 3.18 c and 3.21.

$$\begin{bmatrix} \dot{W} \\ \dot{P} \\ \dot{Q} \\ \dot{R} \end{bmatrix} = \begin{bmatrix} -g \\ \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) QR - \frac{I_R}{I_{XX}} Q\Omega \\ \left(\frac{I_{ZZ} - I_{XX}}{I_{YY}}\right) PR - \frac{I_R}{I_{YY}} P\Omega \\ \left(\frac{I_{XX} - I_{YY}}{I_{ZZ}}\right) PQ \end{bmatrix} + \begin{bmatrix} (\cos\Phi \cos\theta)\frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_{XX}} & 0 & 0 \\ 0 & 0 & \frac{1}{I_{YY}} & 0 \\ 0 & 0 & 0 & \frac{1}{I_{YY}} \end{bmatrix} \begin{bmatrix} T \\ M_{\Phi} \\ M_{\theta} \\ M_{\Psi} \end{bmatrix} = MatC + MatD * U \tag{3.48}$$

From the above equations, we obtain the general form for $\ddot{Y}_1$

$$\ddot{Y}_1 = A_1(\boldsymbol{X_1}) + B_1(\boldsymbol{X_1}) * U \tag{3.49}$$

Where $A_1(\boldsymbol{X_1}) = MatA = \frac{d}{dt} MatW * \begin{bmatrix} W \\ P \\ Q \\ R \end{bmatrix} + MatW * MatC$ and

$$B_1(\boldsymbol{X_1}) = MatB = MatW * MatD$$

The relative degree of the system is calculated as eight, whereas number of states of the system is twelve. To ensure the stability of the whole system, the remaining internal dynamics must be stabilized. But if we consider the sub model that has eight states, the sub model is stabilized.

Based on the general form, the input to the system can be written as:

$$U = \alpha(\boldsymbol{X_1}) + \beta(\boldsymbol{X_1}) * \vartheta \tag{3.50}$$

where $\alpha(\boldsymbol{X_1}) = -B_1(\boldsymbol{X_1})^{-1} * A_1(\boldsymbol{X_1})$

and $\beta(\boldsymbol{X_1}) = B_1(\boldsymbol{X_1})^{-1}$

This on simplification yields $\ddot{Y}_1 = \vartheta$, which is a linear system.

The new input $\vartheta = [\vartheta_1\ \vartheta_2\ \vartheta_3\ \vartheta_4]^T$ can be designed using any of the standard linear control techniques.

In the first approach, we used PD controller to design the 4 linear control inputs, where the error term is given by $e = Z_d - Z$ and so on. The control gains are tuned manually to obtain the desired response.

3.3.3 Linear Quadratic Regulator

In the second approach, the control inputs are designed using the Linear Quadratic Regulator (LQR) approach. LQR is an optimal control technique used to determine the control signal which drives the system states to the desired value along with minimizing a cost function. Hence, the control effort in case of LQR is the least.

Consider a dynamic system of the form:

$$\dot{\boldsymbol{X}} = A\,.\boldsymbol{X} + B\,.U \tag{3.51 a}$$

$$\boldsymbol{Y} = C\,.\boldsymbol{X} \tag{3.51 b}$$

The cost function for this optimal problem is given by:

$$J = \int_{t_0}^{\infty} \{U(t)^T\,.R\,.\,U(t) + [\boldsymbol{X}(t) - \boldsymbol{X_d}(t)]^T\,.Q\,.[\boldsymbol{X}(t) - \boldsymbol{X_d}(t)]\}\,dt \tag{3.52}$$

Where *R* is the cost of actuators and *Q* is the cost of the state.

The control input U that minimizes the cost function is a static linear feedback as:

$$U = -K\,.[\boldsymbol{X}(t) - \boldsymbol{X_d}(t)] \tag{3.53}$$

The value of K is obtained by solving the Riccati's algebraic equation, performed by Matlab using the LQR function:

$$K = LQR(A, B, Q, R) \tag{3.54}$$

The decomposition of the dynamic model into sub models *M1* and *M2* is done in this case too, with the controller being designed for the sub model *M1*. The equations in state variable form for the eight states is the same as Equations 3.18 c, 3.21 and 3.22.

The sub model *M1* is linearized around the equilibrium point (near hover condition) using the Jacobian approach ($A_r = \frac{\partial}{\partial x} A(0)$ and $B_r = B(0)$).

The linearized plant dynamics is given by

$$\dot{\boldsymbol{X_1}} = A_r \boldsymbol{X_1} + B_r U \tag{3.55 a}$$

$$Y_1 = C\boldsymbol{X_1} \tag{3.55 b}$$

where $A_r = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -\frac{A_z}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{A_r}{I_{XX}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{A_r}{I_{YY}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{A_r}{I_{ZZ}} \end{bmatrix}$,

$B_r = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_{XX}} & 0 & 0 \\ 0 & 0 & \frac{1}{I_{YY}} & 0 \\ 0 & 0 & 0 & \frac{1}{I_{ZZ}} \end{bmatrix}$ and

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A function called linriz.m (Appendix A7) was written in Matlab to solve the algebraic equation using Riccati's method, where the LQR in-built function was employed.

$$K = LQR(A_r, B_r, Q, R) \tag{3.56}$$

The Q matrix is then modified manually, the above equation is again solved to find the control gains until the desired response is achieved.

3.4 Trajectory planning using image processing

Processing of visual information from the environment is desirable for an unmanned aerial vehicle. This becomes a necessity when the vehicle is autonomous. This visual information is fed into the quadcopter processor in the form of images. Obtaining useful information from the image data is necessary for taking decisions in case of an autonomous vehicle. This might involve determining the shortest path between two locations, identifying no-fly zones or traversing an environment full of obstacles. Therefore, image processing has an important role to play in the future of quadcopters.

3.4.1 Feasible landing point

Safe landing locations are represented by black circles in the input image. The objective of this algorithm is to calculate the coordinates of the centres of all the black circles in this image. The most suitable landing point is then found out based on the current position and velocity of the quadcopter.

The inputted image is converted to grayscale and then to a black and white image. A very low threshold, while converting to black and white, ensures that only the darkest pixels are retained and the rest are converted to white.The features left in the image, that are less than a nominal

number of pixels, are considered to be disturbances and hence filtered out. The current image consists of only the most prominent black features.

The number of pixels lying on the perimeter of each of these features and the total number of pixels contained by each of these features (can be considered as area) can be calculated using in-built MATLAB functions. The metric used to decide if a given feature is a circle or not is given by:

$$Metric = \frac{4 * \pi * Area}{(Perimeter)^2} \tag{3.57}$$

It can be easily observed that the value of the above parameter equals 1 when the geometry is a circle. No two-dimensional geometry is possible for which the metric exceeds the value of 1. As the value of the metric approaches 1, geometry of the feature approaches that of a circle. Hence the metric is calculated for all the features and and the ones that surpass the threshold (0.9 in our case) qualify as a circle.

The mere proximity of a landing point to the quadcopter at the time of failure is not the best criterion for its selection. Minimizing the time taken to reach the landing point serves the purpose better. The distance from the landing point to the quadcopter and velocity of the quadcopter, at the time of failure, both, are taken into account. Using this, a landing point is suitably selected.

The algorithm for the above task is explained below:-

Let θ_1 be the inclination of the line joining point of failure and landing point being considered with respect to the x-axis in the global coordinate system.

Let θ_2 be the inclination of the velocity vector of quadcopter at point of failure with respect to the x-axis in the global coordinate system.

Then,

$\theta = \theta_1 - \theta_2$, is the angle between the two lines.

Now, v_a is the velocity of the quadcopter along the line joining point of failure and landing point, and v_p is the velocity of the quadcopter perpendicular to it.

$$v_a = |v * cos\,\theta| \tag{3.58 a}$$

$$v_p = |v * \sin\theta| \tag{3.58 b}$$

where v is the velocity of the quadcopter at the time of failure.

The quadcopter has a maximum limit of acceleration. To reach the landing point in the shortest time, it is ensured that the quadcopter's perpendicular displacement from the line joining the point

of failure and the landing point, reaches zero in the same time as it reaches the landing point along the line.

$$0 = ut + \frac{1}{2}a_p t^2 \tag{3.59}$$

$$d = ut + \frac{1}{2}a_a t^2 \tag{3.60}$$

Here d is the distance between the landing point and point of failure.

Solving for t,

$$\frac{(-2 * v_a * v_p)}{u} + \frac{(2 * v_p^2 * a_p)}{a_a^2} = d \tag{3.61}$$

Also

$$a_p^2 + a_a^2 = a^2, \tag{3.62}$$

which is the maximum acceleration possible for the quadcopter.

On solving these two equations using MATLAB function, the value of a_p and a_a is obtained.

With the obtained accelerations along and perpendicular to the quadcopter, the shortest time required to move to the landing point is calculated using the equation,

$$t = \left| \frac{-2 * v_p}{a_p} \right| \tag{3.63}$$

Similarly, the time required for the quadcopter to move to the other landing points are also calculated. Then, the landing point with the shortest time required is chosen for safe landing of the quadcopter.

3.4.2 Trajectory planning algorithm

For maneuvering through the maze, a trajectory is devised from the image using a shortest path algorithm, called Dijkstra's algorithm. Each pixel in the image is considered a node. Each node is assigned a weight based on its proximity to an obstacle. The pixels containing the obstacles have the highest weight and these nodes are not a part of the network. The node at the top left corner of

the image is called the initial node. Let the weight of node Y be the cost assigned to that node. The algorithm is as follows:

1. The initial node is set to current. All the other nodes are marked unvisited. A set of all the unvisited nodes called the unvisited set is created.
2. For the current node, all of its unvisited neighbors are considered and their tentative weights are calculated. The newly calculated tentative weight is compared to the current assigned value and the smaller one is assigned.
3. After considering all of the neighbors of the current node the current node is marked as visited and is removed from the unvisited set. A visited node is never checked again.
4. If the destination node has been marked visited, the algorithm is stopped.
5. Otherwise, the unvisited node that is marked with the smallest tentative weight is selected, and is set as the new "current node", and the algorithm is repeated from step 2.

Once the trajectory from the source node to the destination node is obtained, it is traced by colored lines using MATLAB functions.
A time series data of the trajectory is to be fed to the quadcopter model for trajectory tracking. For this, the time taken to move from one node to the neighboring node is assumed to be of the ratio $1:\sqrt{2}$ for horizontal/vertical nodes and diagonal nodes, i.e. the time taken by a quadcopter to move to a diagonally adjacent node is $\sqrt{2}$ times the time it would take to move to a vertically or horizontally adjacent node.

Using this assumption, the units of time required to move from the source node to the destination node is calculated, by adding$\sqrt{2}$, if the next adjacent node is diagonal and 1, if it is horizontal or vertical, to the current node in the path obtained by Dijkstra's algorithm.

Once the total time is computed, the actual time of flight of the quadcopter is divided by this total time, to get the equivalent of 1 unit in seconds. Now, having the X, Y and time data of each node in the path available, a time series data of the path from source node to destination node is created.

This time series data is taken from the workspace by the Simulink block of the quadcopter model when the simulation is started.

3.5 Trajectory Controller

In this controller, the deviation from the desired path is calculated (in the body frame) at every instant and is fed to the succeeding blocks as the desired velocity. Using this, the desired roll and pitch angles are calculated. This is the underlying principle of this controller.

The errors in X and Y positions are transformed from the inertial frame to the body frame.

$$Error\ in\ X\ in\ body\ frame = (X_d - X)\cos(\psi) + (Y_d - Y)\sin(\psi) \quad (3.64\ a)$$

$$Error\ in\ Y\ in\ body\ frame = (Y_d - Y)\cos(\psi) - (X_d - X)\sin(\psi) \quad (3.64\ b)$$

These errors are taken as the desired velocities in body frame.

Therefore,

$$U_d = Error\ in\ X\ in\ body\ frame \quad (3.65\ a)$$

$$V_d = Error\ in\ Y\ in\ body\ frame \quad (3.65\ b)$$

The next step is calculation of desired attitudes in order to feed it to the attitude controller.

$$\theta_d = K_{p,\theta}(U_d - U) - K_{d,\theta}(\dot{U}) \quad (3.66\ a)$$

$$\Phi_d = -\{K_{p,\Phi}(V_d - V) - K_{d,\Phi}(\dot{V})\} \quad (3.66\ b)$$

The above equations are devoid of $\dot{V}_d$ and $\dot{U}_d$ terms. The absence of these terms permits the presence of non-differentiable points in the path function.

Chapter 4 - Simulation

The dynamic model, controllers and the trajectory planners are implemented in Matlab/Simulink. Separate simulations are carried out for attitude control comparison, trajectory following to the nearest landing point and trajectory following based on the planner. The complete Simulink model including the path commands, trajectory controller, attitude controller and the quadcopter plant is shown in Appendix A15.

The values for the parameters in the quad plant are given in Table 1.

Parameter	Value	Unit
g	9.81	m/s^2
L	0.225	m
m	0.468	kg
K	$2.98*10^{-6}$	
d	0.0382	
B	$0.114*10^{-6}$	
I_{XX}	$4.856*10^{-3}$	kg m^2
I_{YY}	$4.856*10^{-3}$	kg m^2
I_{ZZ}	$8.801*10^{-3}$	kg m^2
I_R	$3.357*10^{-5}$	kg m^2

Table 1: Parameter values for quad plant

The initial conditions given to the quadcopter states for the simulation with feasible landing point is mentioned in Table 2.

State	Value	State	Value
X	0 m	Φ	10 rad
Y	0 m	θ	12 rad
Z	2 m	Ψ	10 rad
U	0 m/s	P	0 rad/s
V	0 m/s	Q	0 rad/s
W	0 m/s	R	0 rad/s

Table 2: Initial conditions for trajectory control simulation 1

The initial conditions given to the quadcopter states for the simulation with trajectory planner is mentioned in Table 3.

State	Value	State	Value
X	1 m	Φ	0 rad
Y	1 m	θ	0 rad
Z	2 m	Ψ	0 rad
U	0 m/s	P	0 rad/s
V	0 m/s	Q	0 rad/s
W	0 m/s	R	0 rad/s

Table 3: Initial conditions for trajectory control simulation 2

4.1. Simulation for attitude controller comparison

The three controllers – Feedback linearization with PD controller (FBL+ PD), Feedback linearization with LQR (FBL+LQR), and PID controller are implemented as separate Simulink models and simulated. The Simulink block layout is shown in Appendices A3- A6.

A simulation time of 50s with a variable step size is given, and Ode-45 Dormand-Prince method is used to solve the numerical problem.

4.1.1. Attitude commands

The attitude commands are provided from a block, which contains step functions for each of the attitude variables. The step function starts with the initial condition as shown in Table 2, and falls to 0 with a step time of 5s for the 3 angles and rises to 3 with the same step time for height command.

4.1.2. Control gains

For the FBL+ PD controller combination, the values of all control gains are determined through manual tuning.

Controller	K_P	K_D
Roll	3	2
Pitch	3	2
Yaw	2	2
Height	3	2

Table 4: Gain values for attitude controller 1

For LQR, the values of all control gains are determined through inbuilt LQR Matlab function, as shown in linriz.m (Appendix A7).

Controller	K_1	K_2
Roll	10	0.1703
Pitch	10	0.1703
Yaw	10	.0267
Height	10	2.8195

Table 5: Gain values for attitude controller 2

For the PID controller, the values of all 3 gains are determined through manual tuning.

Controller	K_P	K_I	K_D
Roll	6	1.5	1.75
Pitch	5	3	3
Yaw	6	1.5	1.75
Height	15	10	10

Table 6: Gain values for attitude controller 3

4.2. Simulation for trajectory following – feasible landing point

4.2.1. Path commands

The path commands are provided by two blocks, one that guides the quadcopter to the final goal location and the other that guides it to the nearest landing point determined through the image processing module. The quadcopter's mission is to follow the specified trajectory and stop at its destination with the desired attitude values. A switch is used to change the source of path commands, which is to be activated by the failure detection module. Here, a clock input is given and switching at the end of 50s is performed assuming that the failure occurs then.

For the initial path commands, *X* and *Y* are fed as a ramp input such that the final goal having coordinates (100,200) is reached within 100s. Z and Ψ are given as a constants 2m and 0 rad respectively.

For the alternative path commands block, the current state is taken as input. The image processing module returns back the coordinates of the nearest landing location. Based on these two points, a ramp is created till the landing point for both *X* and *Y*. Once the actual *X* or *Y* reaches within 0.005% of the desired value, the ramp is replaced by a step with the desired value as magnitude.

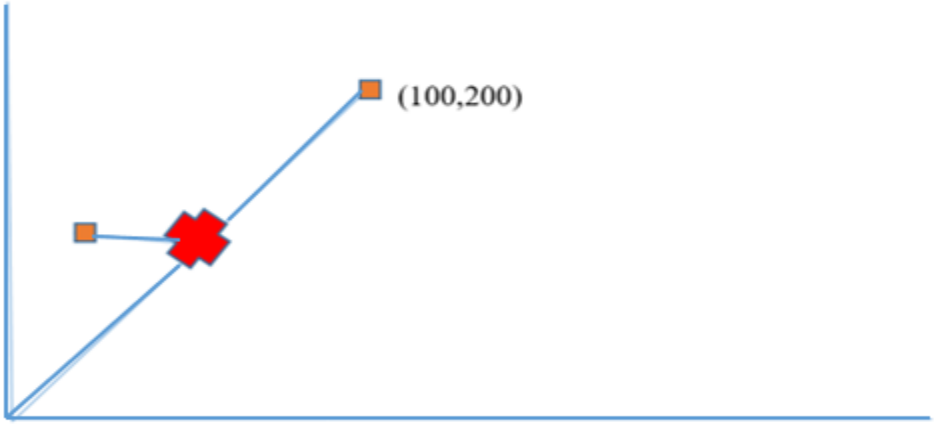


Figure 6: Desired path with switching

4.2.2. Control gains

For the position PD controller, the values for proportional and derivative gains are determined through manual tuning till the desired performance is achieved.

Controller	K_P	K_D
Φ command	0.5	0.4
Θ command	0.36	0.45

Table 7: Gain values for trajectory controller

4.3. Simulation for trajectory following – using a trajectory planner

4.3.1. Path commands

The pathcr.m file, which performs the trajectory planning, is first run to obtain the path command data. While the desired *X* and *Y* commands are obtained as a time series data, the Z and Ψ values

are given as a constant function. Compared to the previous simulation, the quadcopter is made to follow a more complex trajectory.

4.3.2. Control gains

For the position PD controller, the values for proportional and derivative gains are determined through manual tuning till the desired performance is achieved.

Controller	K_P	K_D
Φ command	0.5	0.4
Θ command	0.36	0.45

Table 8: Gain values for trajectory controller

4.3.3. VR Simulation

The Simulink 3D Animation package provides apps for linking Simulink models and MATLAB algorithms to 3D graphics objects. This package can be used to visualize and verify dynamic system behavior in a virtual reality environment. Objects are represented in the Virtual Reality Modeling Language (VRML), a standard 3D modeling language. A 3D world can be animated by changing position, rotation, scale, and other object properties during desktop or real-time simulation.

A VR simulation model of the quadcopter was created using 3D World Editor in MATLAB. This model could be rotated or translated about any axes. It was then implemented into the various SIMULINK models of the quadcopter using a VR Sink block. The phi, theta and psi states of the quadcopter were connected to the rotation control of the VR model and the X, Y and Z states were connected to the translation control. Once the SIMULINK model is run, the VR model achieves the corresponding motion in a given terrain which can be seen through a VR simulator.

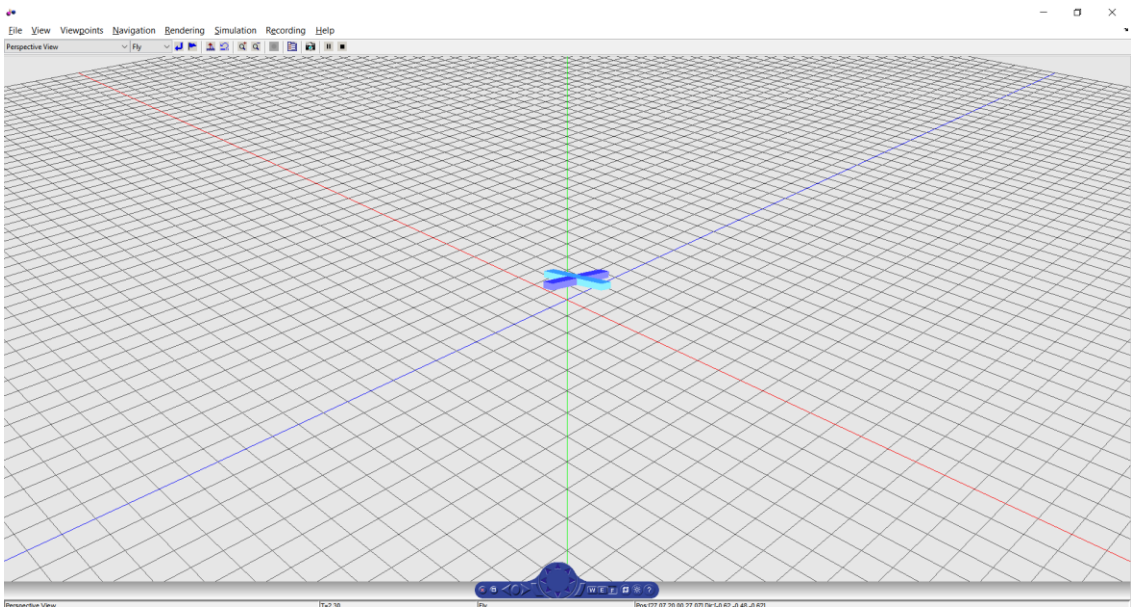


Figure 7: A VR model of quadcopter during simulation (viewpoint 1)

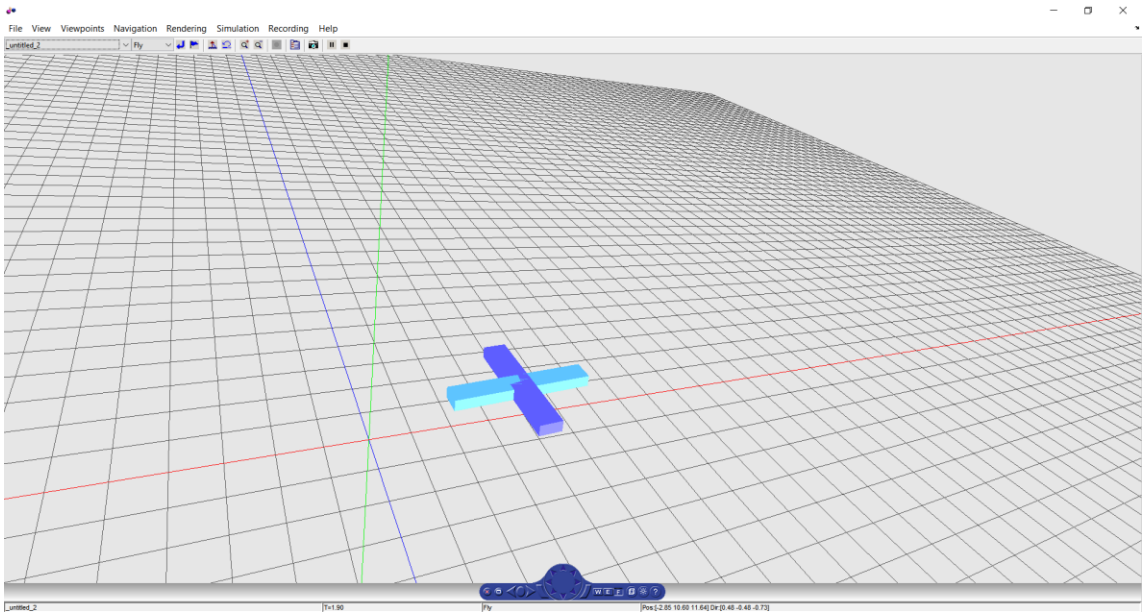


Figure 8: A VR model of quadcopter during simulation (viewpoint 2)

Chapter 5 - Results and Discussions

5.1 Attitude controller comparison

To compare the three different controllers used for the attitude control, a step input is provided as the desired value for each of the attitude variables. The comparison is done based on parameters like rise/fall time and percentage overshoot/undershoot.

Looking at the step response for Φ as seen in figure 9, it is evident that LQR controller shows the best performance as the fall time is least and there is no significant undershoot. The combination of FBL and PD controller shows a comparatively slower response with some undershoot. PID controller shows comparatively poor performance as the fall is very gradual and it takes a long time to settle, though with no oscillations. All the three controllers reach the steady state value with no oscillations, and so settling time is not considered here.

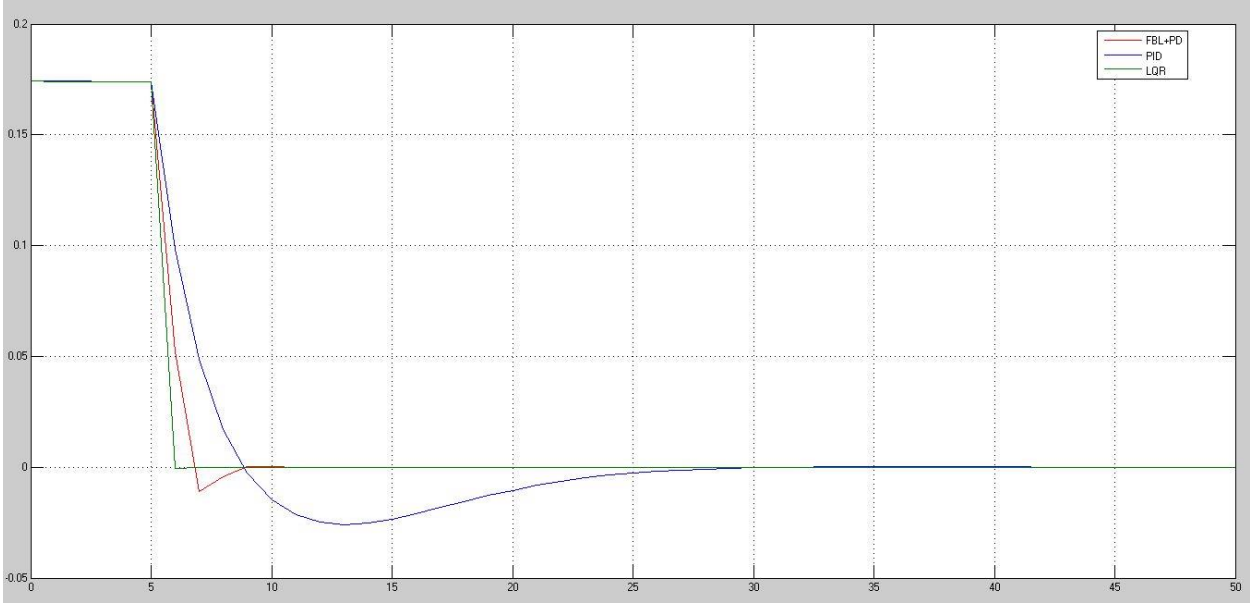


Figure 9: Step responses of controllers – Φ

Table 9 gives a quantitative comparison between the parameters mentioned above, which clearly shows the superior performance of LQR.

Φ(t)	FBL + PD	PID	LQR
Fall time (s)	1.076	2.792	0.059
Undershoot (m) %	5.851	14.368	0.556

Table 9: Characteristic parameters to a step input for Φ

Consider the step response for θ, where LQR again shows the better performance. The combination of FBL and PD controller has almost similar fall time, but shows some undershoot. The PID controller shows a large undershoot and some oscillations leading to a poor response. Based on settling time, both LQR and FBL+PD combination show similar performance but PID lags far behind.

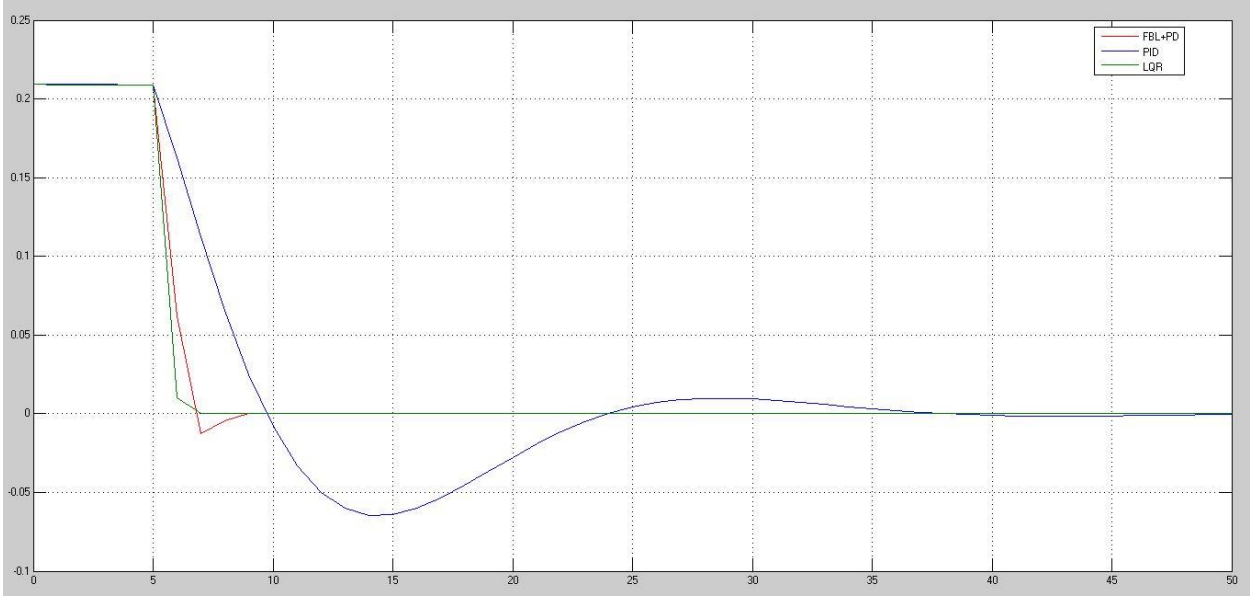


Figure 10: Step responses of controllers – θ

Table 10 gives a clearer picture about the θ response of the three controllers for a step input.

θ(t)	FBL + PD	PID	LQR
Fall time (s)	1.076	3.612	0.074
Undershoot (m) %	5.851	30.921	0.505

Table 10: Characteristic parameters to a step input for θ

Figure 11 shows the step response for Ψ for the three controllers. Here, the combination of FBL and PD controller show the best performance with very small undershoot and a decent fall time. LQR shows the least fall time, but has a large undershoot and rapidly varying response before settling. PID also shows a very undershoot and the slowest response among the three controllers.

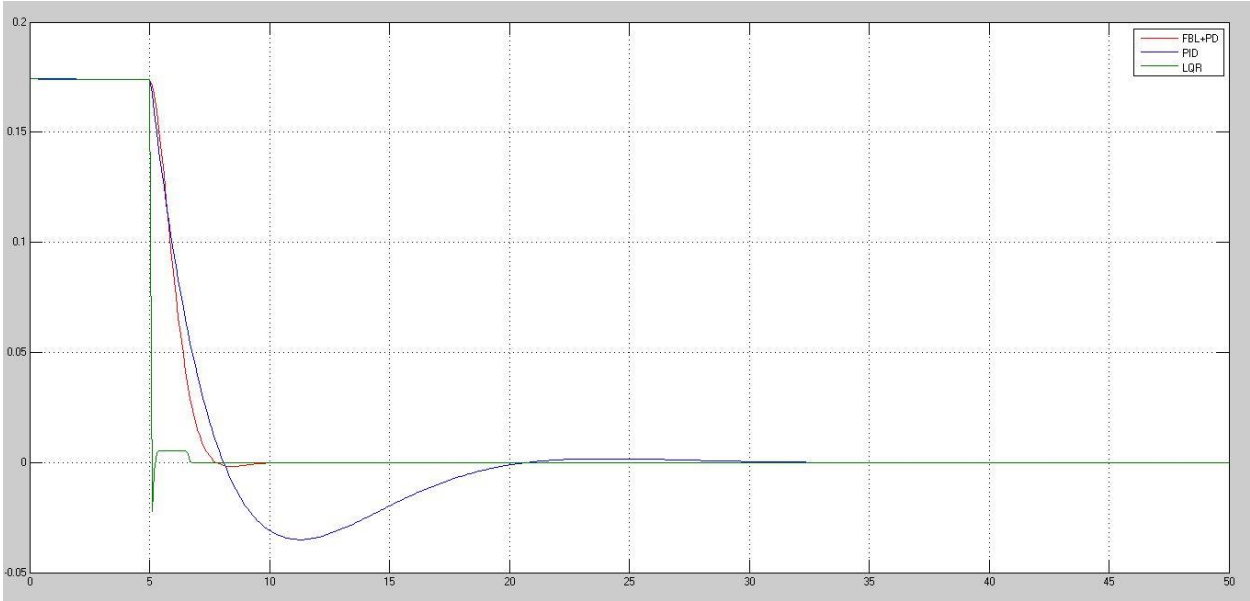


Figure 11: Step responses of controllers – Ψ

Table 11 contains the values of the characteristic parameters used for comparison, and clearly shows that the combination of FBL and PD controller has the best performance.

Ψ(t)	FBL + PD	PID	LQR
Fall time (s)	1.561	2.32	0.053
Undershoot (m) %	1.531	19.88	15.698

Table 11: Characteristic parameters to a step input for Ψ

For *Z* control, since a positive step was given as input the comparison is made on the basis of rise time and percentage of overshoot. Figure 12 shows the response of the three controllers, from which it is evident that the combination of FBL and PD controller has the best performance. It shows the least overshoot and remains steady before the step input is given. The PID controller has a faster rise, but higher overshoot due to the step input and a deviation before the step is provided. LQR shows a very poor performance in this case, as it saturates below the desired height. It also shows deviations before the step is given.

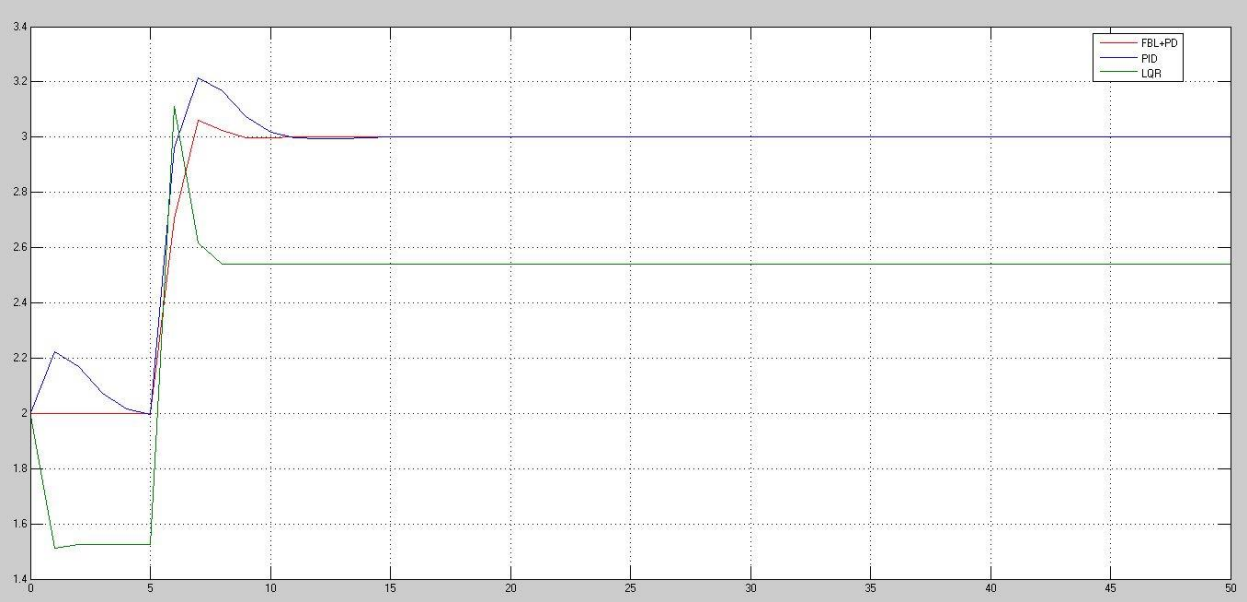


Figure 12: Step responses of controllers –*Z*

From table 12, it is clear that the combination of FBL and PD controller has the best performance.

Z(t)	FBL + PD	PID	LQR
Rise time (s)	1.07	0.788	0.33
Overshoot (m) %	5.851	21.341	58.871

Table 12: Characteristic parameters to a step input for *Z*

Finally, a comparison can also be made based on the computational time. All the 3 controllers were simulated for 50s in Matlab. As seen in the table below, the computational time is least for the combination of FBL and PD controller.

Controller	Simulation time (sec)
FBL + PD	2.98
PID	4.08
LQR	5.10

Table 13: Computational time of controllers

From the comparative study presented above, it is very evident that the combination of Feedback Linearization and PD controller shows the best performance. Hence, a trajectory controller is integrated with this to achieve the desired trajectory tracking.

5.2 Trajectory planner

5.2.1 Selecting the most feasible landing point

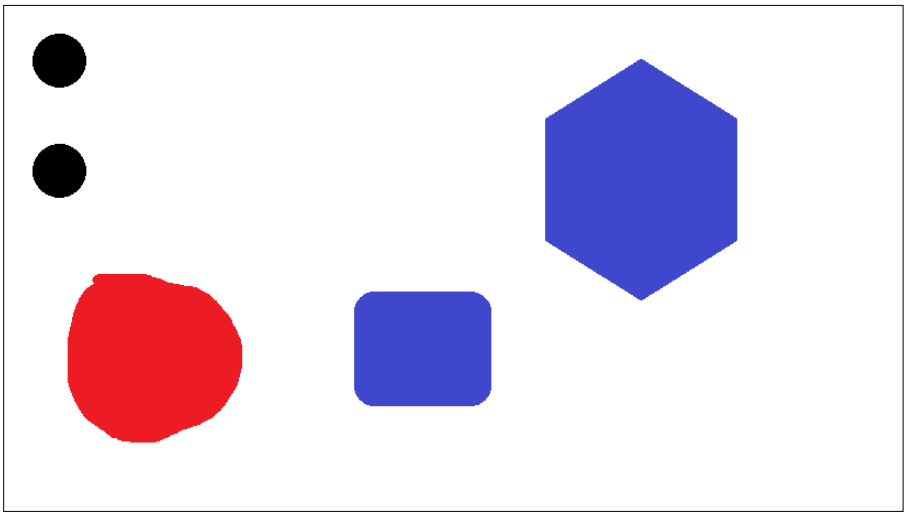


Figure 13: Image containing different shapes in different colors

The input image consists of various shapes in various colors. The final image consists of only the black circles. The centroids of these circles are the feasible landing points.

Figure 14: Black circle given as output

5.2.2 Trajectory Planning algorithm

Figure 15 : Maze input to the trajectory planner

The input maze consists of walls which are no fly zones shown in Figure 15. The Figure 16 shows the path generated by the trajectory planner.

Figure 16 : Path generated by the trajectory planner

5.3 Trajectory tracking

5.3.1 Trajectory tracking – Most feasible landing point

A comparison can be made between the desired trajectory and the actual trajectory using Figure 17. It can be inferred that the quadcopter is able to follow the trajectory with reasonable accuracy throughout. At the point of switching, the inertia of the quadcopter prevents it from making a sharp change in the trajectory. So it consumes some time before again following the desired trajectory. When it approaches the end point, the change in input from ramp to step occurs, but the quadcopter due to its inertia oscillates about that point before finally resting there. The final location of the quadcopter is exactly in the specified landing point, which can be concluded from Figure 18 where *X* and *Y* comparisons are plotted.

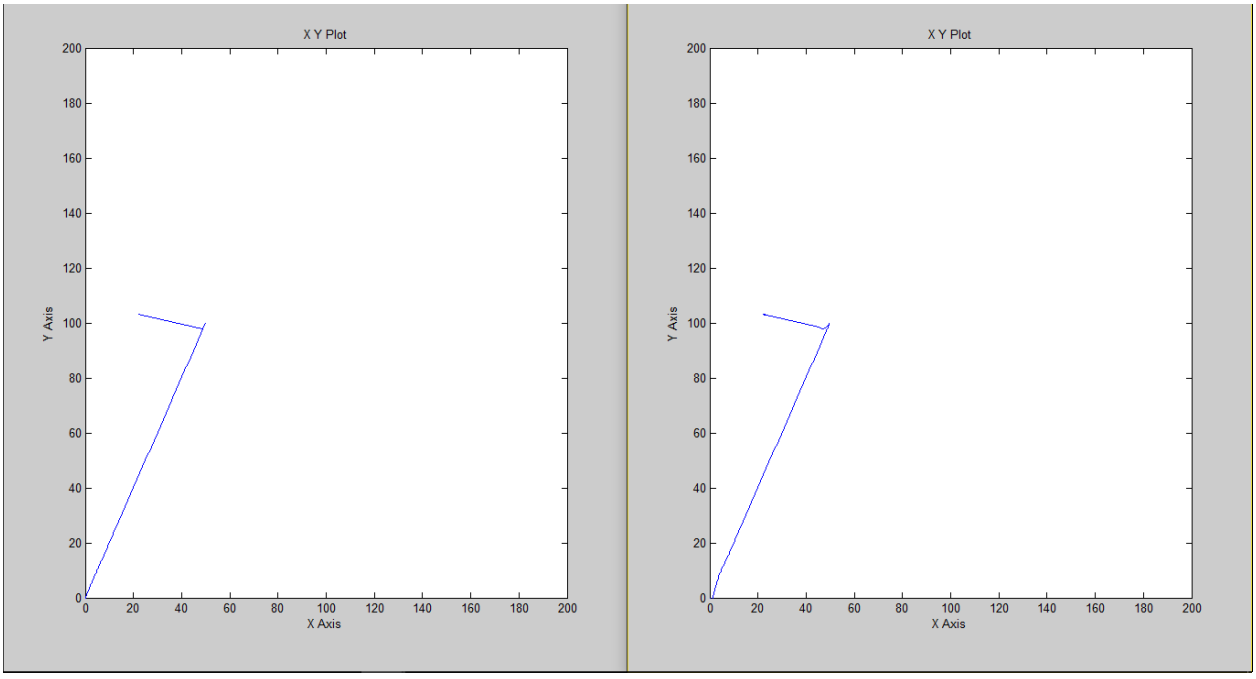


Figure 17: Actual path followed

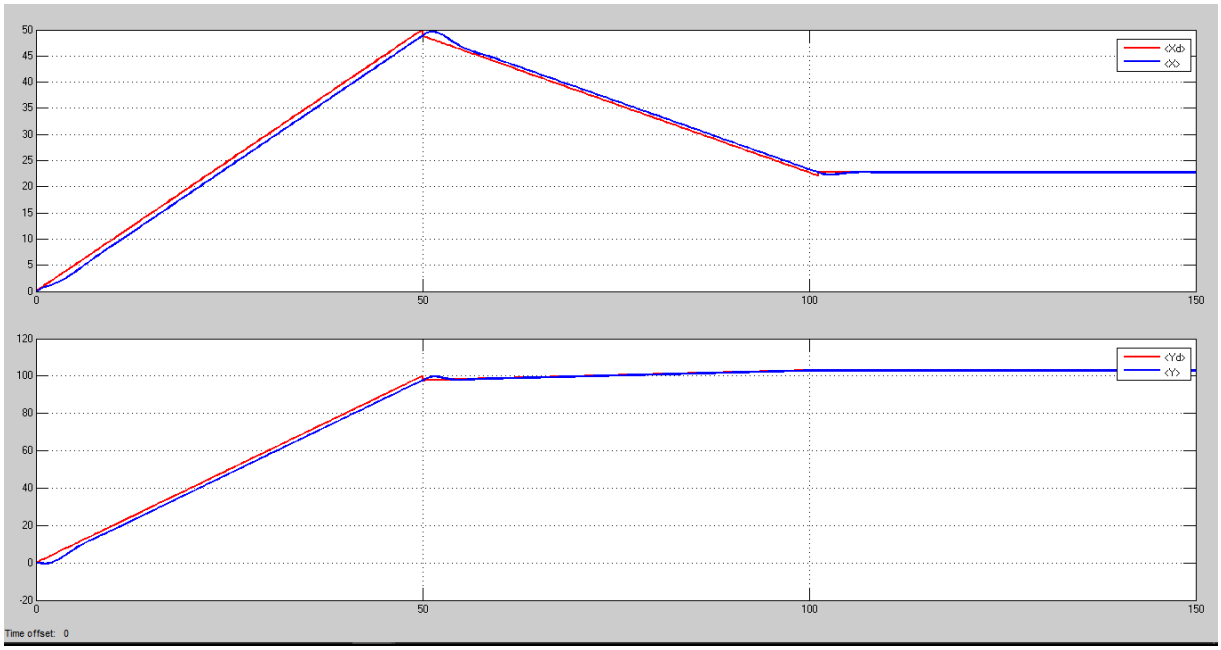


Figure 18: Coordinate wise comparison of desired and actual paths

The plots of attitude variables vs time for the above simulation is given in Figure 19. There are three regions of interest – the starting point when a path command is given, during switching and finally after reaching the landing point. The attitude variables show changes only in these three regions.

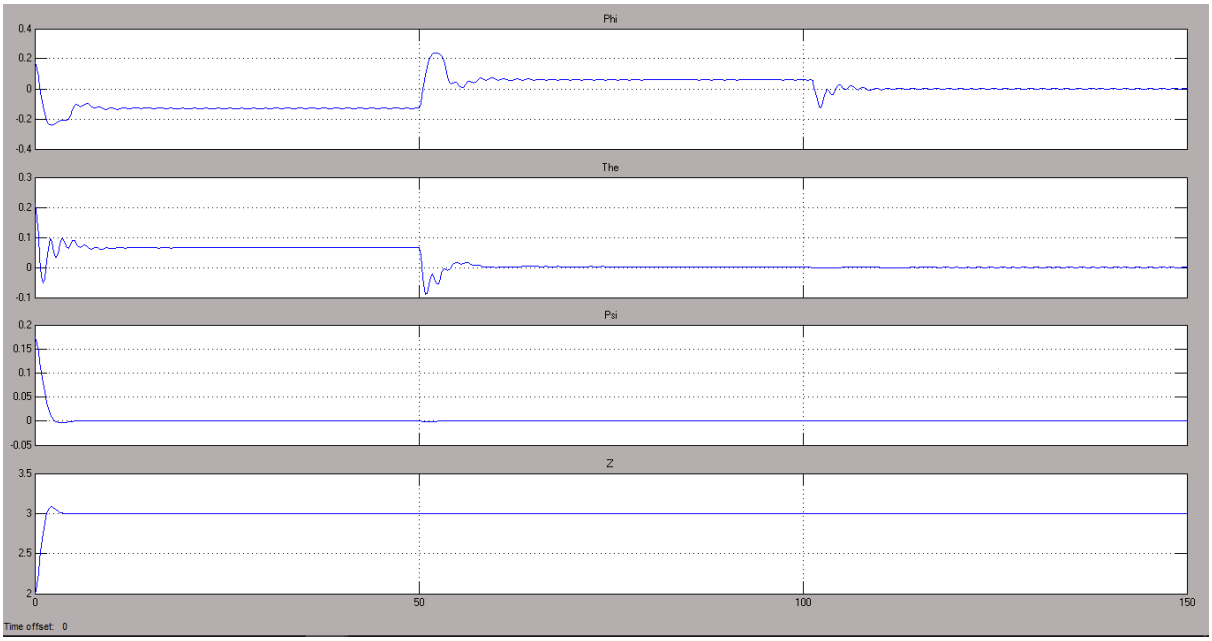


Figure 19: Plots of attitude variables vs Time

5.3.2. Trajectory tracking – traversing the maze

In this simulation, the quadcopter is traversing a complex maze. The path planning is done based on the image of surroundings. The image is processed to identify obstacles and safe flying zones. The Dijkstra's algorithm is employed to determine the shortest path from the starting point to destination. Then, trajectory tracking is used to closely follow this path.

The shortest path from the left top corner of the image to the right bottom corner of the image is generated using Dijkstra's Algorithm. The path is efficient and is devoid of any obstacles. This path can be seen in Figure 20.

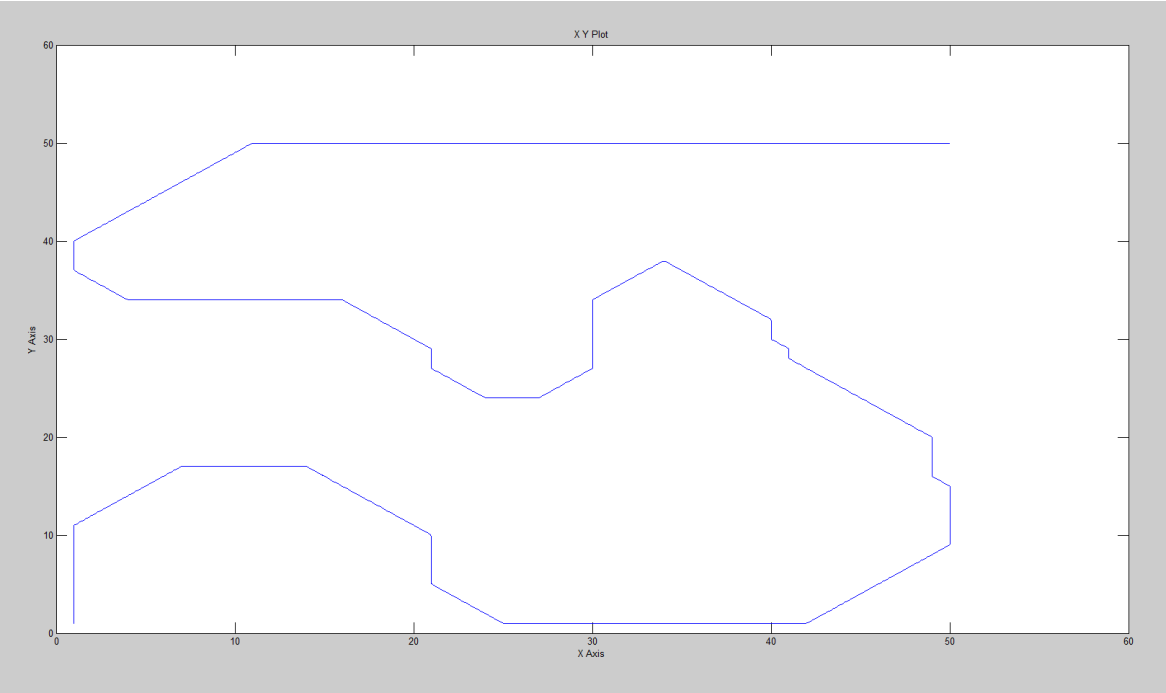


Figure 20: Desired path generated by trajectory planner

This trajectory is fed in the form of a time series data to the controller. The trajectory is followed with reasonable accuracy. It is visible in Figure 21 that all the sharp turns in the desired trajectory are transformed to smooth curves in the actual path due to the inertia of the quadcopter.

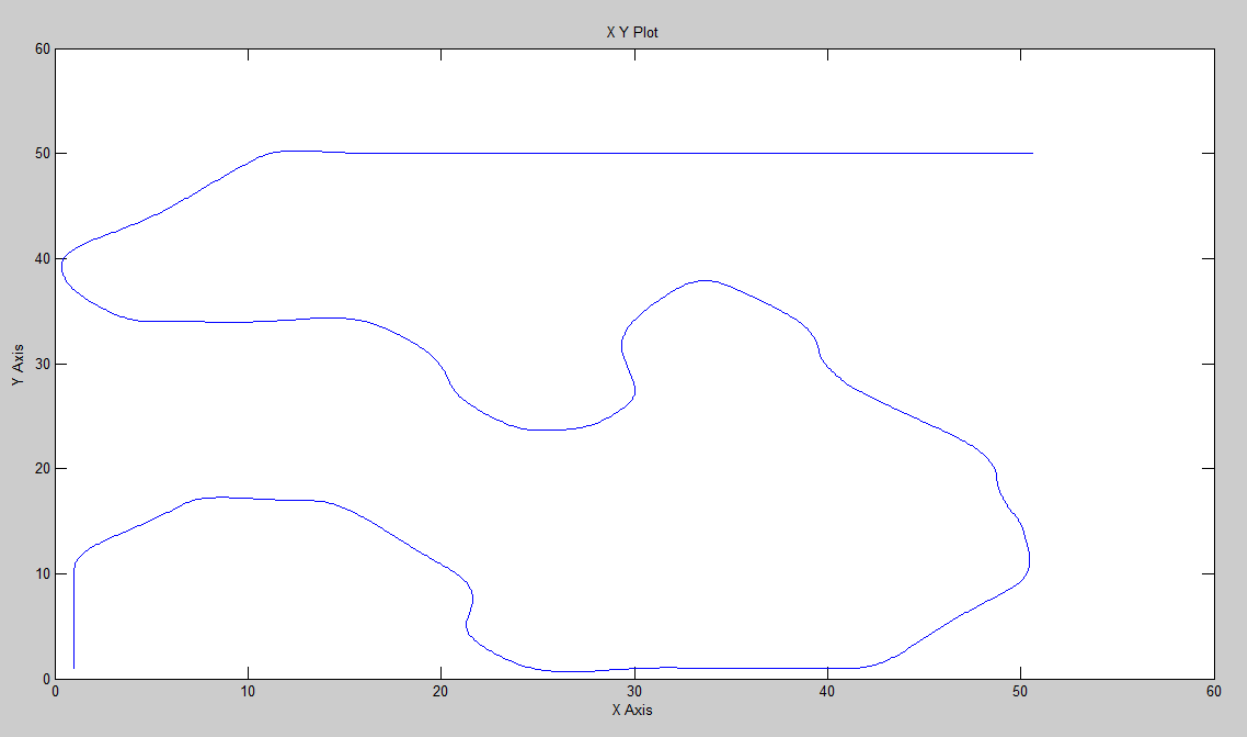


Figure 21: Actual path followed by quadcopter

The trajectory controller is successful in guiding the quadcopter to traverse such a complex path, which is clearly seen in Figure 22 through a coordinate wise comparison. Although there is a slight lag between the desired and actual paths due to the slow response of the trajectory controller, it finally ceases at the end when the destination is reached.

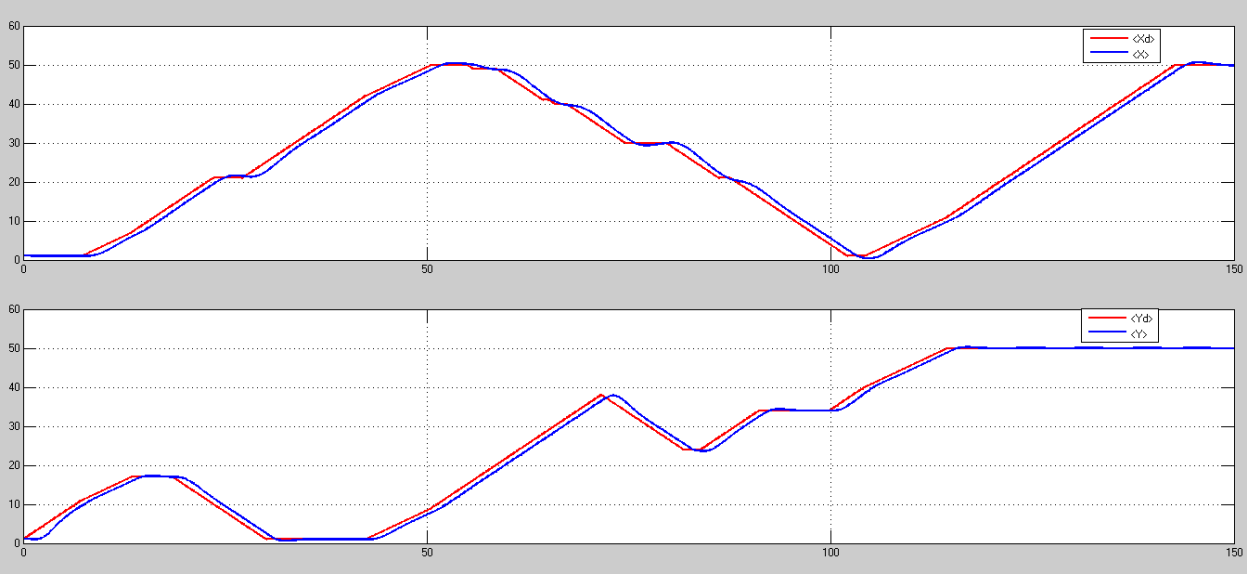


Figure 22: Coordinate wise comparison of desired path and actual path

The attitude plots, especially the roll and pitch angles show changes throughout the course of the quadcopter motion because of the complexity in path. Theta and x are correlated whereas Phi and y are correlated. Z settles at 3m soon after the start of the simulation.

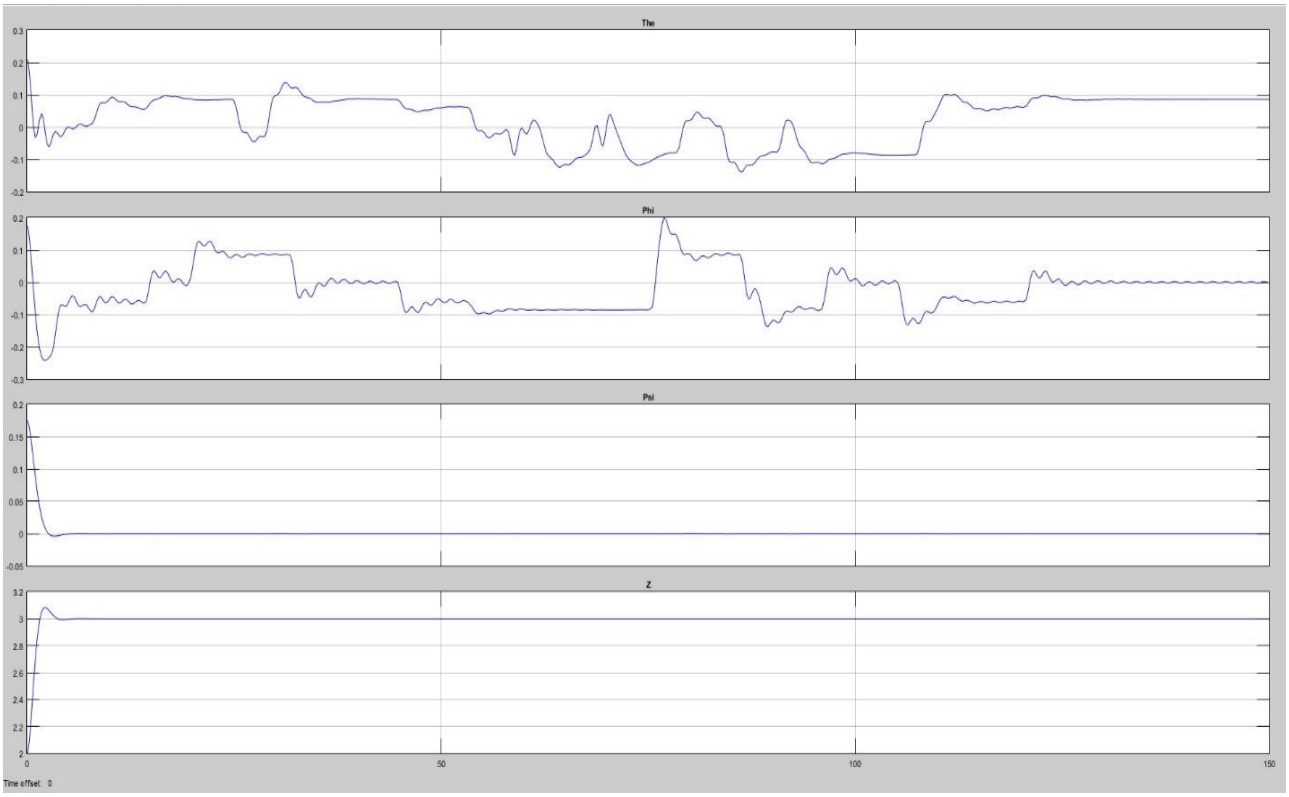


Figure 23: Plots of Attitude variables vs Time

Further work

Miniature Unmanned Aerial Vehicles (UAVs) with ability to vertically take-off and land (as in quadrotors) exhibit advantages and features in maneuverability that has recently gained strong interest in the research community. All the simulations were carried out ensuring that the quadcopters entire motion occurs at sufficient height above the ground, and take-off and landing are not performed by the quadcopter. The phenomenon of ground effect has also not been considered at low altitudes. Incorporating the ability to perform vertical take-off and landing overcoming ground effects is a key issue which will be addressed in the future.

Reliability of control systems require robustness and fault tolerance capabilities in presence of anomalies and unexpected failures in actuators, sensors or subsystems. Designing a controller that can combat the failure of one or more rotors is a further step in this project. Based on the inputs from the FDI module, the quadcopter must be equipped to switch to the failsafe controller on the onset of failure. Implementing an adaptive control strategy of this kind is another key issue to be addressed.

References

[1] Luukkonen, Teppo. "Modelling and control of quadcopter." *Independent research project in applied mathematics, Espoo* (2011).

[2] Hossain, M. Raju, D. Geoff Rideout, and D. Nicholas Krouglicof. "Bond graph dynamic modeling and stabilization of a quad-rotor helicopter." *Proceedings of the 2010 Spring Simulation Multiconference*. Society for Computer Simulation International, 2010.

[3] Mahony, Robert, Vijay Kumar, and Peter Corke. "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor." *IEEE robotics & automation magazine* 19.3 (2012): 20-32.

[4] Dikmen, I. Can, Aydemir Arisoy, and Hakan Temeltas. "Attitude control of a quadrotor." *Recent Advances in Space Technologies, 2009. RAST'09. 4th International Conference on*. IEEE, 2009.

[5] Hoffmann, Gabriel M., Steven L. Waslander, and Claire J. Tomlin. "Quadrotor helicopter trajectory tracking control." *AIAA guidance, navigation and control conference and exhibit*. 2008.

[6] S. Bouabdallah, A. Noth, and R. Siegwart, "PID vs LQ control techniques applied to an indoor micro quadrotor," IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, pp. 2451–2456, 2004.

[7] Zhou, Qing-Li, et al. "Design of feedback linearization control and reconfigurable control allocation with application to a quadrotor UAV." *Control and Fault-Tolerant Systems (SysTol), 2010 Conference on*. IEEE, 2010.

[8] Sabatino, Francesco. "Quadrotor control: modeling, nonlinear control design, and simulation." (2015).

[9] Das, Abhijit, Kamesh Subbarao, and Frank Lewis. "Dynamic inversion with zero-dynamics stabilisation for quadrotor control." *IET control theory & applications* 3.3 (2009): 303-314.

[10] Lee, Daewon, H. Jin Kim, and Shankar Sastry. "Feedback linearization vs. adaptive sliding mode control for a quadrotor helicopter." *International Journal of control, Automation and systems* 7.3 (2009): 419-428.

[11] Slotine, Jean-Jacques E., and Weiping Li. *Applied nonlinear control*. Vol. 199. No. 1. Englewood Cliffs, NJ: prentice-Hall, 1991.

Appendix

Appendix A1: Quadcopter plant

```
function quadplant(block)
setup(block);

function setup(block)

  block.NumInputPorts  = 4 ;

  block.NumOutputPorts = 12;

  for i = 1:4; % These are the motor inputs
  block.InputPort(i).Dimensions        = 1;
  block.InputPort(i).DirectFeedthrough = false;
  block.InputPort(i).SamplingMode      = 'Sample';
  end

  for i = 1:12;
  block.OutputPort(i).Dimensions       = 1;
  block.OutputPort(i).SamplingMode     = 'Sample';
  end

  % Register the parameters.
  block.NumDialogPrms     = 0; %fromtemplate

  % Set up the continuous states.
 block.NumContStates = 12; %notintemplate

  block.SampleTimes = [0 0];

  block.SetAccelRunOnTLC(false);

  block.SimStateCompliance = 'DefaultSimState';

  block.RegBlockMethod('InitializeConditions', @InitializeConditions);

  block.RegBlockMethod('Outputs', @Outputs);

  block.RegBlockMethod('Derivatives', @Derivatives);
  block.RegBlockMethod('Terminate', @Terminate); % Required


function InitializeConditions(block)

% P, Q, R are in rad/s
P=0; Q=0; R=0;

% Phi, The, Psi are in rads
```

```
Phi=10*pi/180; The=12*pi/180; Psi=10*pi/180;

U=0; V=0; W=0;
X=0; Y=0; Z=2;

init = [P,Q,R,Phi,The,Psi,U,V,W,X,Y,Z];

for i=1:12
block.OutputPort(i).Data = init(i);
block.ContStates.Data(i) = init(i);
end

function Outputs(block)
for i = 1:12;
  block.OutputPort(i).Data = block.ContStates.Data(i);
end

function Derivatives(block)

% P Q R in units of rad/sec
P = block.ContStates.Data(1);
Q = block.ContStates.Data(2);
R = block.ContStates.Data(3);
% Phi The Psi in radians
Phi = block.ContStates.Data(4);
The = block.ContStates.Data(5);
Psi = block.ContStates.Data(6);
% U V W in units of m/s
U = block.ContStates.Data(7);
V = block.ContStates.Data(8);
W = block.ContStates.Data(9);
% X Y Z in units of m
X = block.ContStates.Data(10);
Y = block.ContStates.Data(11);
Z = block.ContStates.Data(12);
% w values in rev/min! NOT radians/s!!!!
w1 = block.InputPort(1).Data;
w2 = block.InputPort(2).Data;
w3 = block.InputPort(3).Data;
w4 = block.InputPort(4).Data;
w  = [w1; w2; w3; w4];

% CALCULATE MOMENT AND THRUST FORCES

%find k,d,l
k=2.98e-06; d=.0382; l=0.225;

%find m,Ixx,Iyy,Izz,Ir
m=0.468; Ixx=4.856e-03;Iyy=4.856e-03;Izz=8.801e-03;Ir=3.357e-05;
Ax=.3; Ay=0.3; Az=0.25; Ar=0.2;
T1= k*w1^2;
T2= k*w2^2;
T3= k*w3^2;
T4= k*w4^2;
```

```
T = T1+T2+T3+T4; %total thrust
Mphi= l*(T4-T2); %torques
Mthe= l*(T3-T1);
Mpsi= d*(-T1+T2-T3+T4);

Omega=w1-w2+w3-w4;

dP= ((Iyy-Izz)/Ixx)*Q*R - Ir/Ixx * Q*Omega + Mphi/Ixx - Ar/Ixx*P;
dQ= ((Izz-Ixx)/Iyy)*P*R + Ir/Iyy * P*Omega + Mthe/Iyy - Ar/Iyy*Q;
dR= ((Ixx-Iyy)/Izz)*P*Q + Mpsi/Izz -Ar/Izz*R;

dPhi= P+ sin(Phi)*tan(The)*Q + cos(Phi)*tan(The)*R;
dTheta= cos(Phi)*Q - sin(Phi)*R;
dPsi= sin(Phi)/cos(The)*Q + cos(Phi)/cos(The)*R;

dU= ( sin(Phi)*sin(Psi) + cos(Phi)*sin(The)*cos(Psi) )*T/m - Ax/m*U;
dV= ( -sin(Phi)*cos(Psi) + cos(Phi)*sin(The)*sin(Psi) )*T/m - Ay/m*V;
dW= -9.8 + cos(Phi)*cos(The)*T/m - Az/m*W;

vb = [U;V;W];

dX = U;
dY = V;
dZ = W;

f = [dP dQ dR dPhi dTheta dPsi dU dV dW dX dY dZ].';

  %This is the state derivative vector
block.Derivatives.Data = f;

function Terminate(block)

%endfunction
```

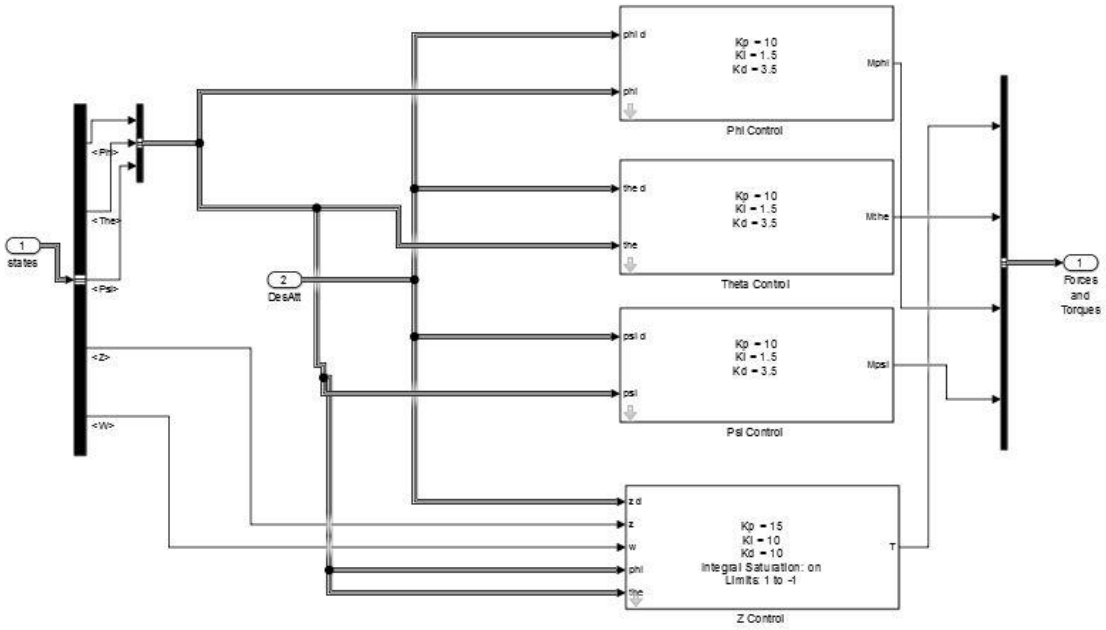
Appendix A2: Quadcopter plant with a failed rotor

```
 function quadplant2(block)
setup(block);

function setup(block)

  block.NumInputPorts  = 3;

  block.NumOutputPorts = 12;


  for i = 1:3; % These are the motor inputs
  block.InputPort(i).Dimensions        = 1;
  block.InputPort(i).DirectFeedthrough = false;
  block.InputPort(i).SamplingMode      = 'Sample';
  end

  for i = 1:12;
  block.OutputPort(i).Dimensions       = 1;
  block.OutputPort(i).SamplingMode     = 'Sample';
  end

  % Register the parameters.
  block.NumDialogPrms     = 0; %fromtemplate

  % Set up the continuous states.
 block.NumContStates = 12; %notintemplate

  block.SampleTimes = [0 0];

  block.SetAccelRunOnTLC(false);

  block.SimStateCompliance = 'DefaultSimState';

  block.RegBlockMethod('InitializeConditions', @InitializeConditions);

  block.RegBlockMethod('Outputs', @Outputs);

  block.RegBlockMethod('Derivatives', @Derivatives);
  block.RegBlockMethod('Terminate', @Terminate); % Required


function InitializeConditions(block)
% P, Q, R are in rad/s
P=0; Q=0; R=0;

% Phi, The, Psi are in rads
Phi=10*pi/180; The=12*pi/180; Psi=10*pi/180;

U=0; V=0; W=0;
X=0; Y=0; Z=2;
```

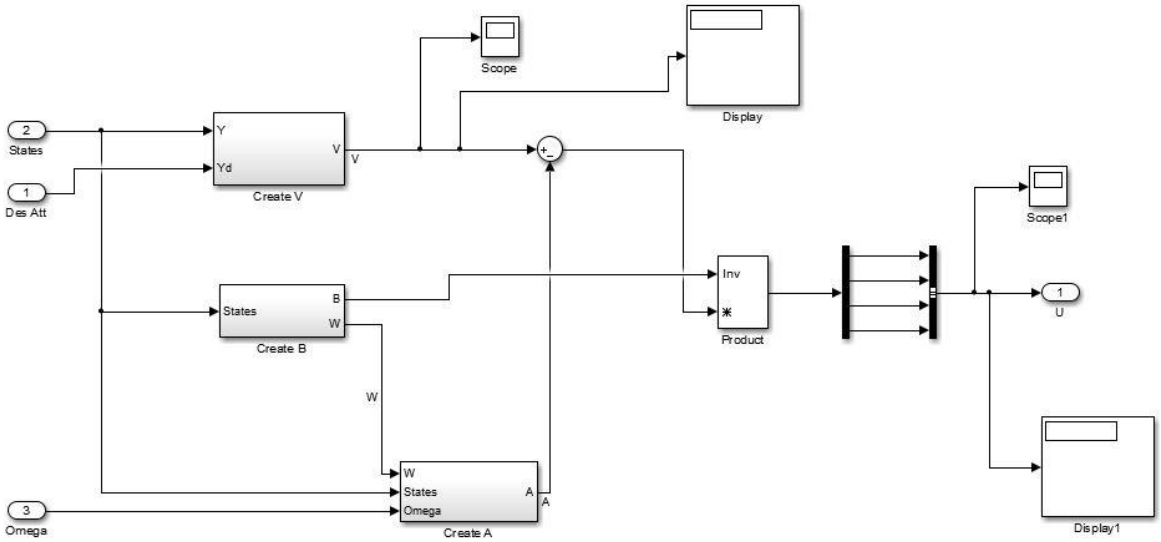
```
init = [P,Q,R,Phi,The,Psi,U,V,W,X,Y,Z];

for i=1:12
block.OutputPort(i).Data = init(i);
block.ContStates.Data(i) = init(i);
end

function Outputs(block)
for i = 1:12;
  block.OutputPort(i).Data = block.ContStates.Data(i);
end

function Derivatives(block)

% P Q R in units of rad/sec
P = block.ContStates.Data(1);
Q = block.ContStates.Data(2);
R = block.ContStates.Data(3);
% Phi The Psi in radians
Phi = block.ContStates.Data(4);
The = block.ContStates.Data(5);
Psi = block.ContStates.Data(6);
% U V W in units of m/s
U = block.ContStates.Data(7);
V = block.ContStates.Data(8);
W = block.ContStates.Data(9);
% X Y Z in units of m
X = block.ContStates.Data(10);
Y = block.ContStates.Data(11);
Z = block.ContStates.Data(12);
% w values in rev/min! NOT radians/s!!!!
w1 = block.InputPort(1).Data;
w3 = block.InputPort(2).Data;
w4 = block.InputPort(3).Data;
w  = [w1; w3; w4];

%find k,d,l
k=2.98e-06; d=.03825; l=0.225;

%find m,Ixx,Iyy,Izz,Ir
m=0.468; Ixx=4.856e-03;Iyy=4.856e-03;Izz=8.801e-03;Ir=3.357e-05;
Ax=.3; Ay=0.3; Az=0.25; Ar=0.1;

T1= k*w1^2;
%T2= k*w2^2;
T3= k*w3^2;
T4= k*w4^2;

Fmat= [ 1 1 1; -l l 0; -d -d d;];
Fmat1=inv(Fmat);

mat1= [T1;T3;T4];
mat2= Fmat*mat1;
```

```
T = mat2(1); %total thrust
Mthe= mat2(2);%torques
Mpsi= mat2(3);

%Mphi is not used as control input but appears later in eq
%Substitute for Mphi
Mphi= 0.5*l*(T-Mpsi/d);
Omega=w1+w3-w4; %or opp signs check.

dP= ((Iyy-Izz)/Ixx)*Q*R - Ir/Ixx * Q*Omega + Mphi/Ixx - Ar/Ixx*P;
dQ= ((Izz-Ixx)/Iyy)*P*R + Ir/Iyy * P*Omega + Mthe/Iyy - Ar/Iyy*Q;
dR= ((Ixx-Iyy)/Izz)*P*Q + Mpsi/Izz -Ar/Izz*R;

dPhi= P+ sin(Phi)*tan(The)*Q + cos(Phi)*tan(The)*R;
dTheta= cos(Phi)*Q - sin(Phi)*R;
dPsi= sin(Phi)/cos(The)*Q + cos(Phi)/cos(The)*R;

dX = U;
dY = V;
dZ = W;

dU= ( sin(Phi)*sin(Psi) + cos(Phi)*sin(The)*cos(Psi) )*T/m - Ax/m*U;
dV= ( -sin(Phi)*cos(Psi) + cos(Phi)*sin(The)*sin(Psi) )*T/m - Ay/m*V;
dW= -9.8 + cos(Phi)*cos(The)*T/m - Az/m*W;


vb = [U;V;W];
Rib = [cos(Psi)*cos(The) cos(Psi)*sin(The)*sin(Phi)-sin(Psi)*cos(Phi)
cos(Psi)*sin(The)*cos(Phi)+sin(Psi)*sin(Phi);
       sin(Psi)*cos(The) sin(Psi)*sin(The)*sin(Phi)+cos(Psi)*cos(Phi)
sin(Psi)*sin(The)*cos(Phi)-cos(Psi)*sin(Phi);
       -sin(The)         cos(The)*sin(Phi)
cos(The)*cos(Phi)];
% i_dp = Rib*vb;
%dX = i_dp(1);
%dY = i_dp(2);
%dZ = i_dp(3);
f = [dP dQ dR dPhi dTheta dPsi dU dV dW dX dY dZ].';

  %This is the state derivative vector
block.Derivatives.Data = f;

function Terminate(block)

%endfunction
```

Appendix A3: Layout for PID controller

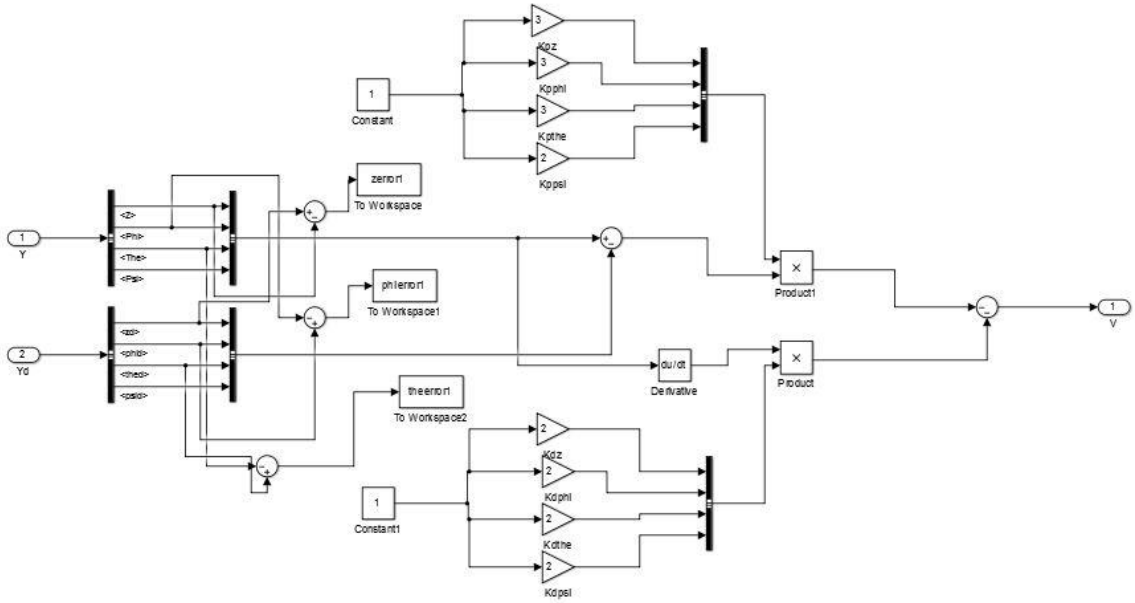


Appendix A4: Layout for FBL+PD controller



Appendix A5: Inside FBL blocks

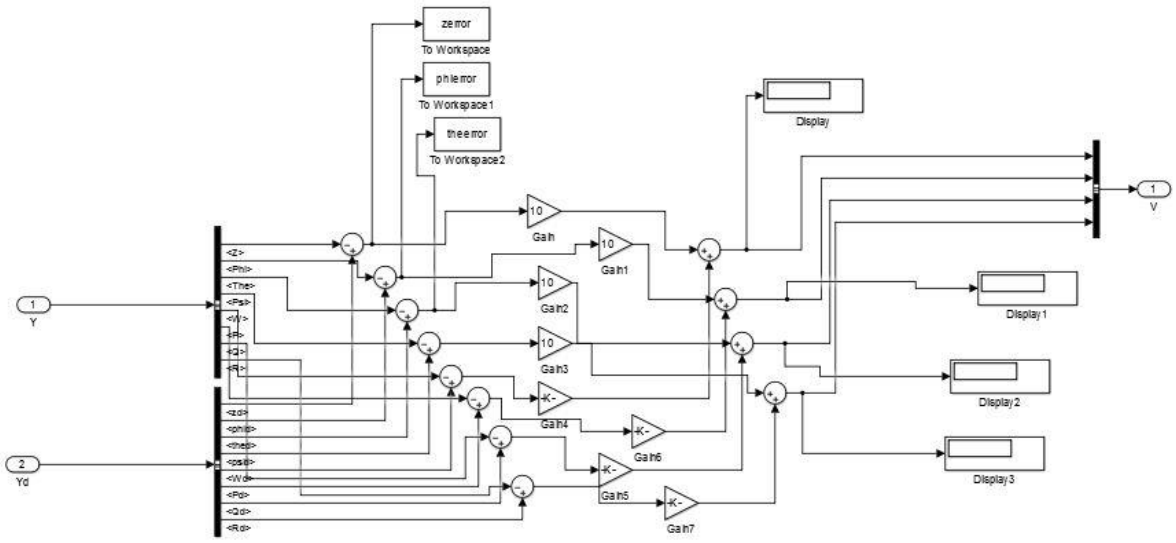
Inside Create V block



Matrices used for Feedback linearization

```
function B = MatW(phi,the)
B=[ 1 0 0 0; 0 1 sin(phi)*tan(the) cos(phi)*tan(the); 0 0 cos(phi) -sin(phi);
0 0 sin(phi)/cos(the) cos(phi)/cos(the)];
end


function B = MatD(phi,the)
m=0.468; Ixx=4.856e-03;Iyy=4.856e-03;Izz=8.801e-03;
B=[ 1/m*cos(phi)*cos(the) 0 0 0; 0 1/Ixx 0 0; 0 0 1/Iyy 0; 0 0 0 1/Izz];
end


function B = MatC(P,Q,R,Omega,W)
m=0.468; Ixx=4.856e-03;Iyy=4.856e-03;Izz=8.801e-03;Ir=3.357e-05;Ar=0.2;
Az=.25;
B=[-9.8 - Az/m*W; (Iyy-Izz)/Ixx*Q*R - Ir/Ixx*Q*Omega - Ar/Ixx*P; (Izz-
Ixx)/Iyy*P*R + Ir/Iyy*P*Omega - Ar/Iyy*Q; (Ixx-Iyy)/Izz*P*Q - Ar/Izz*R ];
end


function B = MatA(W,Wdot,C,Q);
B = W*C+Wdot*Q;
end
```

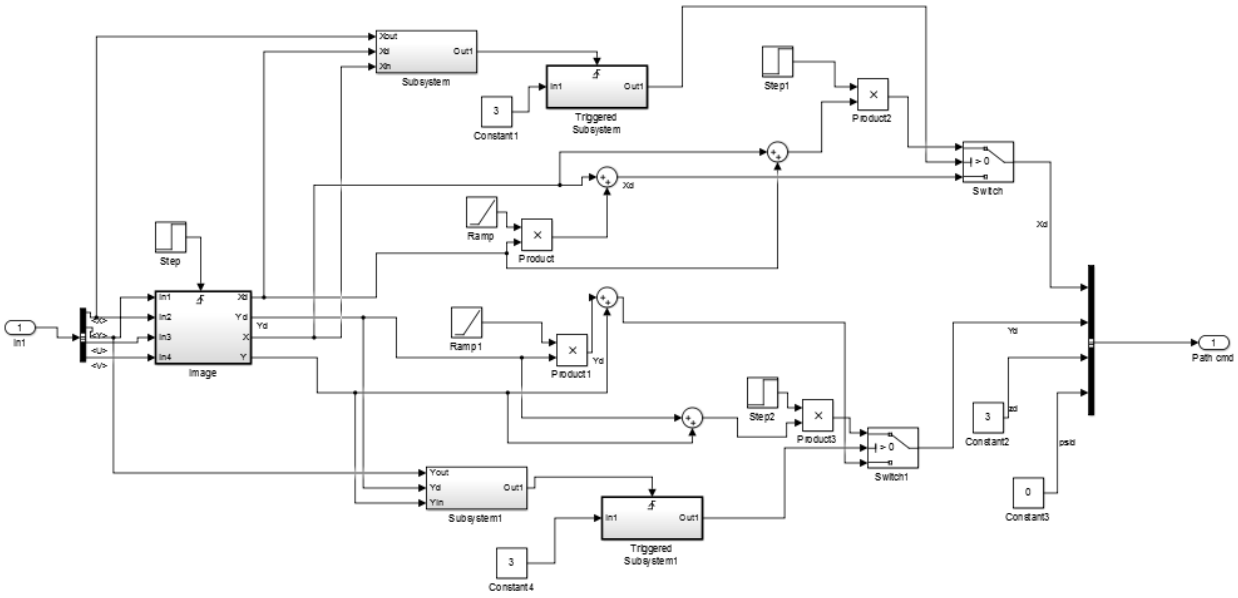
Appendix A6: Layout of LQR



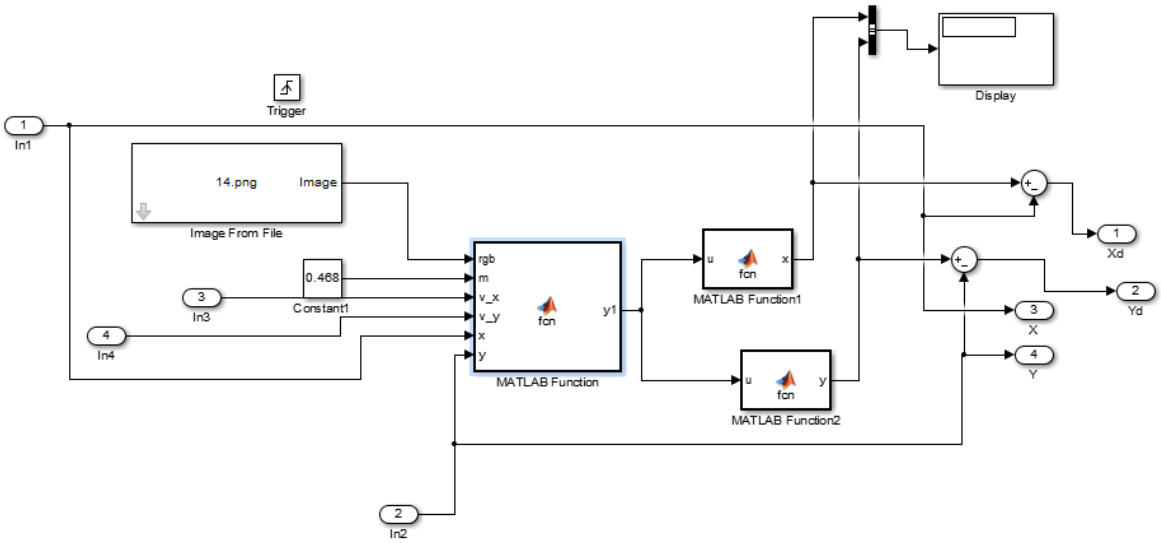
Appendix A7: Linriz.m function

```
A=[0 0 0 0 1 0 0 0;0 0 0 0 0 1 0 0;0 0 0 0 0 0 1 0;0 0 0 0 0 0 0 1;0 0 0 0 -
.5341 0 0 0;0 0 0 0 0 -41.186 0 0;0 0 0 0 0 0 -41.186 0;0 0 0 0 0 0 0
22.725];
B=[0 0 0 0;0 0 0 0;0 0 0 0;0 0 0 0;2.137 0 0 0;0 205.93 0 0;0 0 205.93 0;0 0
0 29788.5];
C=[1 0 0 0 0 0 0 0;0 1 0 0 0 0 0 0;0 0 1 0 0 0 0 0;0 0 0 1 0 0 0 0];
D=zeros(4);
sys_ss = ss(A,B,C,D);
co = ctrb(sys_ss);
controllability = rank(co);
Q = C'*C;
R=eye(4);
K = lqr(A,B,Q,R);

%Increasing the weights to improve performance
Q(1,1)=100;
Q(2,2)=100;
Q(3,3)=100;
Q(4,4)=100;

K = lqr(A,B,Q,R);
```

Appendix A8: Inside image processing block



Inside the image block



Appendix A9: Pathgen.m function

```
function y= pathgen(L,m,v_x,v_y,x,y)
omega=1000;
F=4*2.98*10^(-6)*omega^2;
theta=pi/4;
% m=0.468;
a=F*sin(theta)/m;
% v_x=10;
% v_y=10;
s=sqrt(v_x^2+v_y^2);
% x=150;
% y=150;

k=1;
rows=size(L);
while k<=rows(1,1)
    theta1=atan2(L(k,2)-y,L(k,1)-x);
    theta2=atan2(v_y,v_x);
    theta=theta1-theta2;
     v_a=abs(s*cos(theta));
     v_p=abs(s*sin(theta));

d=sqrt((x-L(k,1))^2 + (y-L(k,2))^2);

if theta == 0 || s ==0
    t(k)=(1/a)*(sqrt(v_a^2+2*a*d)-v_a);

elseif theta == pi || theta == -pi
    t(k)=(1/a)*(sqrt(v_a^2+2*a*d)+v_a);


else
syms u v
[solv, solu] = solve(u^2 + v^2 == a^2, (-2*v_a*v_p)/u + (2*v_p^2*v)/(u^2) ==
d);

p=1;

while p<=length(solv)
if isreal(solv(p))==1
 if (L(k,1)-x)*solv(p)>=0
    a_x=solu(p);
    a_y=solv(p);
 end
end
p=p+1;
end

t(k)=abs(-2*v_p/a_y);
end
k=k+1;
end
```

```
min = t(1);r=1;
l=2;
while l<=length(t)
    if t(l)<=min
        min=t(l);
        r=l;
    end
    l=l+1;
end
y=L(r,:);
```

Appendix A10: Circle2.m function

```
function y= circle2(RGB)
% imshow(RGB);

I= rgb2gray(RGB);
% bw = imbinarize(I);
bw=im2bw(I,0.3);

 bw2 = bwmorph(~bw, 'dilate',2);
 bw = bwareaopen(bw2,500);
 se = strel('disk',2);
bw = imclose(bw,se);
bw = imfill(bw,'holes');

[B,L] = bwboundaries(bw,'noholes');



% Display the label matrix and draw each boundary
imshow(label2rgb(L, @jet, [.5 .5 .5]))
hold on
for k = 1:length(B)
  boundary = B{k};
  plot(boundary(:,2), boundary(:,1), 'w', 'LineWidth', 2)
end

stats = regionprops(L,'Area','Centroid');
threshold = 0.99;
centroid=zeros(length(B),2);


% loop over the boundaries
i=1;
for k = 1:length(B)

  % obtain (X,Y) boundary coordinates corresponding to label 'k'
  boundary = B{k};
```

```
  % compute a simple estimate of the object's perimeter
  delta_sq = diff(boundary).^2;
  perimeter = sum(sqrt(sum(delta_sq,2)));

  % obtain the area calculation corresponding to label 'k'
  area = uint32(stats(k).Area);


  % compute the roundness metric
  metric =(4*pi*area)/perimeter^2;

  % display the results
  metric_string = sprintf('%2.2f',metric);

  % mark objects above the threshold with a black circle
  if metric > threshold
    centroid(i,:)=stats(k).Centroid;
     plot(centroid(1),centroid(2),'ko');


  end
  i=i+1;

  text(boundary(1,2)-35,boundary(1,1)+13,metric_string,'Color','y',...
       'FontSize',14,'FontWeight','bold');

end
% disp(centroid);
p=1;
q=0;
while p<=length(B)

    if centroid(p,1)~=0

        q=q+1;
    end
    p=p+1;

end
x=zeros(q,2);
p=1;
q=1;
while p<=length(B)

    if centroid(p,1)~=0
        x(q,:)=centroid(p,:);
        q=q+1;
    end
    p=p+1;

end
y=x;
% title(['Metrics closer to 1 indicate that ',...
```

```
%        'the object is approximately round']);
%   imshow(bw);
```

Appendix A11: dijkstra.m function

```
%---------------------------------------------------
% Dijkstra Algorithm
% author : Dimas Aryo
% email : mr.dimasaryo@gmail.com
%
% usage
% [cost rute] = dijkstra(Graph, source, destination)
%
% example
% G = [0 3 9 0 0 0 0;
%      0 0 0 7 1 0 0;
%      0 2 0 7 0 0 0;
%      0 0 0 0 0 2 8;
%      0 0 4 5 0 9 0;
%      0 0 0 0 0 0 4;
%      0 0 0 0 0 0 0;
%      ];
% [e L] = dijkstra(G,1,7)
%---------------------------------------------------
function [e L] = dijkstra(A,s,d)

if s==d
    e=0;
    L=[s];
else

A = setupgraph(A,inf,1);

if d==1
    d=s;
end
A=exchangenode(A,1,s);

lengthA=size(A,1);
W=zeros(lengthA);
for i=2 : lengthA
    W(1,i)=i;
    W(2,i)=A(1,i);
end

for i=1 : lengthA
    D(i,1)=A(1,i);
    D(i,2)=i;
end

D2=D(2:length(D),:);
L=2;
```

```
while L<=(size(W,1)-1)
    L=L+1;
    D2=sortrows(D2,1);
    k=D2(1,2);
    W(L,1)=k;
    D2(1,:)=[];
    for i=1 : size(D2,1)
        if D(D2(i,2),1)>(D(k,1)+A(k,D2(i,2)))
            D(D2(i,2),1) = D(k,1)+A(k,D2(i,2));
            D2(i,1) = D(D2(i,2),1);
        end
    end

    for i=2 : length(A)
        W(L,i)=D(i,1);
    end
end
if d==s
    L=[1];
else
    L=[d];
end
e=W(size(W,1),d);
L = listdijkstra(L,W,s,d);
end
```

Appendix A12: Functions called by Dijkstra.m

(save as separate files)

```
function L = listdijkstra(L,W,s,d)

index=size(W,1);
while index>0
    if W(2,d)==W(size(W,1),d)
        L=[L s];
        index=0;
    else
        index2=size(W,1);
        while index2>0
            if W(index2,d)<W(index2-1,d)
                if W(index2,1)==s
                    L = [L 1];
                else
                    L=[L W(index2,1)];
                end
                L=listdijkstra(L,W,s,W(index2,1));
                index2=0;
            else
                index2=index2-1;
            end
            index=0;
```

```
        end
    end
end


function G = exchangenode(G,a,b)

%Exchange element at column a with element at column b;
buffer=G(:,a);
G(:,a)=G(:,b);
G(:,b)=buffer;

%Exchange element at row a with element at row b;
buffer=G(a,:);
G(a,:)=G(b,:);
G(b,:)=buffer;


function G = setupgraph(G,b,s)

if s==1
    for i=1 : size(G,1)
        for j=1 :size(G,1)
            if G(i,j)==0
                G(i,j)=b;
            end
        end
    end
end
if s==2
    for i=1 : size(G,1)
        for j=1 : size(G,1)
            if G(i,j)==b
                G(i,j)=0;
            end
        end
    end
end
```

Appendix A13: pathcr.m function

```
% function d_path = pathcr(m)
m=imread('path5.png');

n=rgb2gray(m);
a=im2bw(n);
p=ones(50,50);
for i=1:1:50
for j=1:1:50
if a(i,j)==0
    p(i,j)=5;
```

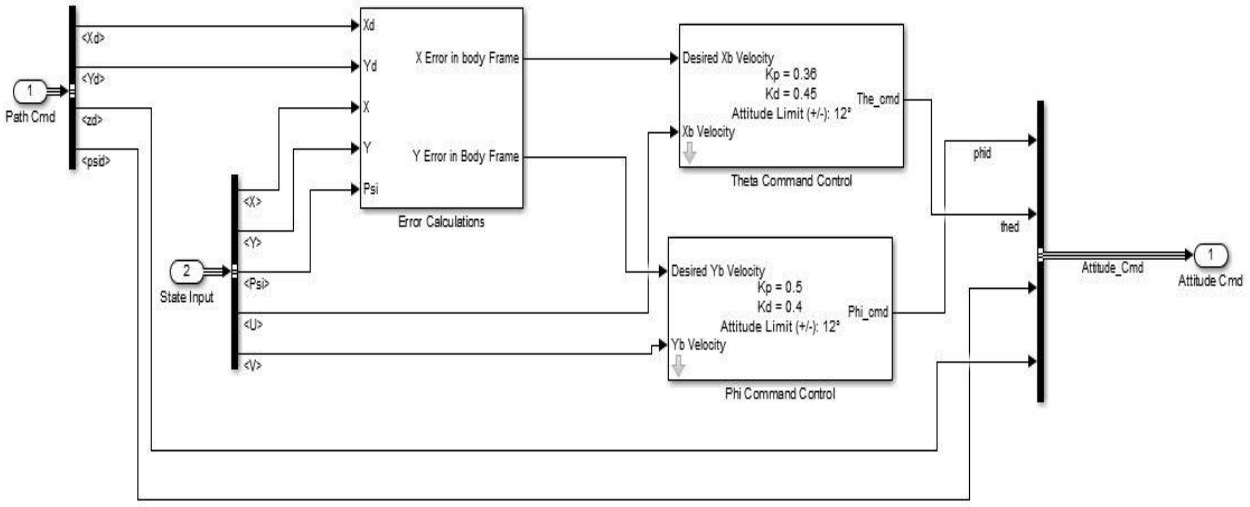
```
else
    p(i,j)=1;
end
end
end
%for l=1:1:100
for i=1:1:50
for j=1:1:50

    if p(i,j)~=5

if i==1 && j==1
        p(i,j)=(p(i,j+1)+p(i+1,j))/2;
elseif i==1 && j==50
        p(i,j)=(p(i,j-1)+p(i+1,j))/2;
    elseif i==50 && j==50
        p(i,j)=(p(i,j-1)+p(i-1,j))/2;
    elseif i==50 && j==1
        p(i,j)=(p(i,j+1)+p(i-1,j))/2;
    elseif i==1 && j~=50 && j~=1
        p(i,j)=(p(i,j-1)+p(i+1,j)+p(i,j+1))/3;
    elseif i==50 && j~=50 && j~=1
        p(i,j)=(p(i,j-1)+p(i-1,j)+p(i,j+1))/3;
elseif j==1 && i~=50 && i~=1
        p(i,j)=(p(i-1,j)+p(i,j+1)+p(i+1,j))/3;
elseif j==50 && i~=50 && i~=1
        p(i,j)=(p(i-1,j)+p(i,j-1)+p(i+1,j))/3;
    else
        p(i,j)=(p(i-1,j)+p(i,j-1)+p(i+1,j)+p(i,j+1))/4;
end
    end

end
end
%end
q=1;
k=1;
n=zeros(50,50);
for i=1:1:50
    for j=1:1:50
        n(i,j)=q;
        q=q+1;
    end
end
A=zeros(2500,2500);
for i=1:1:50
    for j=1:1:50

            if i==1 && j==1
       if p(i,j+1)~=5
           A(n(i,j),n(i,j+1))=p(i,j+1); end
       if p(i+1,j)~=5
           A(n(i,j),n(i+1,j))=p(i+1,j); end
       if p(i+1,j+1)~=5
           A(n(i,j),n(i+1,j+1))=p(i+1,j+1); end
            elseif i==1 && j==50
```

```
if p(i,j-1)~=5
    A(n(i,j),n(i,j-1))=p(i,j-1); end
if p(i+1,j)~=5
    A(n(i,j),n(i+1,j))=p(i+1,j); end
if p(i+1,j-1)~=5
    A(n(i,j),n(i+1,j-1))=p(i+1,j-1); end
     elseif i==50 && j==50
if p(i,j-1)~=5
    A(n(i,j),n(i,j-1))=p(i,j-1); end
if p(i-1,j)~=5
    A(n(i,j),n(i-1,j))=p(i-1,j); end
if p(i-1,j-1)~=5
    A(n(i,j),n(i-1,j-1))=p(i-1,j-1); end
     elseif i==50 && j==1
if p(i,j+1)~=5
    A(n(i,j),n(i,j+1))=p(i,j+1); end
if p(i-1,j)~=5
    A(n(i,j),n(i-1,j))=p(i-1,j); end
if p(i-1,j+1)~=5
    A(n(i,j),n(i-1,j+1))=p(i-1,j+1); end
     elseif i==1 && j~=50 && j~=1
if p(i,j+1)~=5
    A(n(i,j),n(i,j+1))=p(i,j+1); end
if p(i+1,j)~=5
    A(n(i,j),n(i+1,j))=p(i+1,j); end
if p(i,j-1)~=5
    A(n(i,j),n(i,j-1))=p(i,j-1); end
if p(i+1,j-1)~=5
    A(n(i,j),n(i+1,j-1))=p(i+1,j-1); end
if p(i+1,j+1)~=5
    A(n(i,j),n(i+1,j+1))=p(i+1,j+1); end
     elseif i==50 && j~=50 && j~=1
if p(i,j+1)~=5
    A(n(i,j),n(i,j+1))=p(i,j+1); end
if p(i-1,j)~=5
    A(n(i,j),n(i-1,j))=p(i-1,j); end
if p(i,j-1)~=5
    A(n(i,j),n(i,j-1))=p(i,j-1); end
if p(i-1,j-1)~=5
    A(n(i,j),n(i-1,j-1))=p(i-1,j-1); end
if p(i-1,j+1)~=5
    A(n(i,j),n(i-1,j+1))=p(i-1,j+1); end
     elseif j==1 && i~=50 && i~=1
if p(i+1,j)~=5
    A(n(i,j),n(i+1,j))=p(i+1,j); end
if p(i,j+1)~=5
    A(n(i,j),n(i,j+1))=p(i,j+1); end
if p(i-1,j)~=5
    A(n(i,j),n(i-1,j))=p(i-1,j); end
if p(i-1,j+1)~=5
    A(n(i,j),n(i-1,j+1))=p(i-1,j+1); end
if p(i+1,j+1)~=5
    A(n(i,j),n(i+1,j+1))=p(i+1,j+1); end
     elseif j==50 && i~=50 && i~=1
if p(i+1,j)~=5
    A(n(i,j),n(i+1,j))=p(i+1,j); end
if p(i,j-1)~=5
```

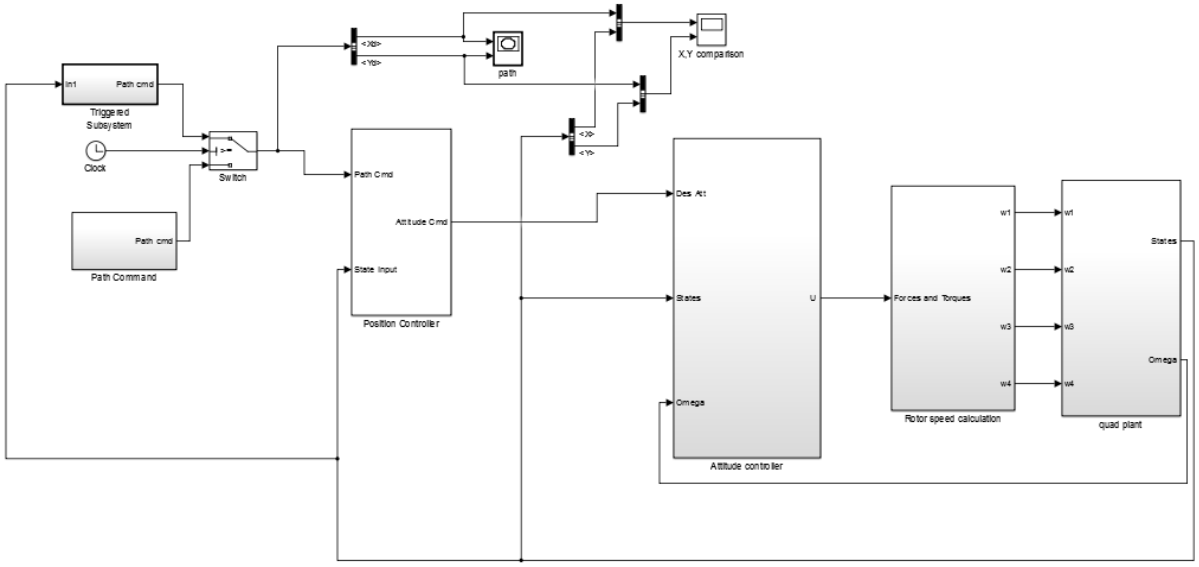
```
            A(n(i,j),n(i,j-1))=p(i,j-1); end
        if p(i-1,j)~=5
            A(n(i,j),n(i-1,j))=p(i-1,j); end
        if p(i-1,j-1)~=5
            A(n(i,j),n(i-1,j-1))=p(i-1,j-1); end
        if p(i+1,j-1)~=5
            A(n(i,j),n(i+1,j-1))=p(i+1,j-1); end
             else
        if p(i,j+1)~=5
            A(n(i,j),n(i,j+1))=p(i,j+1); end
        if p(i+1,j)~=5
            A(n(i,j),n(i+1,j))=p(i+1,j); end
        if p(i,j-1)~=5
            A(n(i,j),n(i,j-1))=p(i,j-1); end
        if p(i+1,j-1)~=5
            A(n(i,j),n(i+1,j-1))=p(i+1,j-1); end
         if p(i+1,j+1)~=5
            A(n(i,j),n(i+1,j+1))=p(i+1,j+1); end
         if p(i-1,j)~=5
            A(n(i,j),n(i-1,j))=p(i-1,j); end
         if p(i-1,j-1)~=5
            A(n(i,j),n(i-1,j-1))=p(i-1,j-1); end
         if p(i-1,j+1)~=5
            A(n(i,j),n(i-1,j+1))=p(i-1,j+1); end
             end
    end
end
[cost, t_route] = dijkstra(A,1,2500);
j=ones(50,50);
l=length(t_route);
x=zeros(1,l);
y=zeros(1,l);
for i=1:1:l
    j(t_route(i))=0;
    x(i)=mod(t_route(i),50);
    if x(i)==0
        x(i)=50;
    end
    y(i)=ceil(t_route(i)/50);
end
x=fliplr(x);
y=fliplr(y);
for i=1:1:10
    x(l+i)=x(l);
    y(l+i)=y(l);
end

w=m;
for i=1:1:l+10
    m(y(i),x(i))=0;
end
p=0;
tim=zeros(1,l+10);
for i=2:1:l+10
    key_x=0;
    key_y=0;
    if x(i)-x(i-1)==1
```

```
        key_x=1;
    end
    if y(i)-y(i-1)==1
        key_y=1;
    end
        if key_x==1&&key_y==1
            p=p+sqrt(2);
        else
            p=p+1;
        end
        tim(i)=p;
end
  time = 150*tim/tim(1+10) ;
 imshow(m);
    ts_x = timeseries(x,time);
    ts_y = timeseries(y,time);
    path = struct('x',ts_x,'y',ts_y);
    d_path=path;
% save('x_time.mat',path);
```

Appendix A14: Trajectory controller



Appendix A15: Complete layout for trajectory control simulation 1



Appendix A16: Complete layout for trajectory control simulation 2