TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

B-05-BAS-2019/24

MODEL PREDICTIVE CONTROL-BASED OPTIMAL TRAJECTORY TRACKING IN AUTONOMOUS QUADROTOR SYSTEM

By:
Dipesh Simkhada (076/BAS/012)
Hemant Bhatta (076/BAS/014)
Krishna Bahadur Karki (076/BAS/016)
Prasanna Pratap Thapa (076/BAS/028)

A PROJECT REPORT SUBMITTED TO THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE BACHELOR'S DEGREE IN AEROSPACE ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

March, 2024

COPYRIGHT

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled **'MODEL PREDICTIVE CONTROL-BASED OPTIMAL TRAJECTORY TRACKING IN AUTONOMOUS QUADROTOR SYSTEM'** submitted by **Dipesh Simkhada, Hemant Bhatta, Krishna Bahadur Karki and Prasanna Pratap Thapa** in partial fulfillment of the requirements for the Bachelor's Degree in Aerospace Engineering.

Supervisor: **Dr. Mahesh Chandra Luintel**, Professor
Department of Mechanical and Aerospace Engineering
Institute of Engineering, Pulchowk Campus

External Examiner: **Er. Biman Rimal**, Assistant Professor,
Department of Mechanical Engineering
Institute of Engineering, Thapathali Campus

Tribhuvan University
Institute of Engineering
Department of Mechanical and Aerospace
Pulchowk Campus

Head of Department : **Dr. Sudip Bhattarai**, Assistant Professor
Department of Mechanical and Aerospace Engineering
Institute of Engineering, Pulchowk Campus

DATE OF APPROVAL: 3/12/2024

ABSTRACT

Optimal trajectory tracking through nonlinear optimization is a computational process that aims to track the reference path for a system, considering the dynamics and objectives of system along with the constraints that are provided. Among current approaches in trajectory tracking formulation as an optimization problem, this paper describes Model Predictive Control(MPC), an approach that provides a continually evolving optimized trajectory tracking system that also respects system dynamics and constraints. The project showcases both the numerical simulation for the MPC implemented for reference trajectory tracking followed by an implementation on the actual quadcopter. However, in the actual hardware implementation flight test have been conducted in the quadrotor where only the attitude has been controlled in the PX4 firmware directly using Model Predictive Control. In this paper, the MPC-based control technique is used for trajectory tracking of the quadrotor model in different types of trajectories. For numerical simulation, non linear model has been developed and analyzed using the Finite-length time horizon control referred to as Model Predictive Control whereas, for SITL(Software-in-The-Loop) simulation, linear model of MPC have been used. For the flight test, to achieve efficient computational cost for the Pixhawk Flight Controller, the linear model has been implemented to analyze the attitude control shown by the quadrotor system. Finally, the system's performance for the MPC control are studied and discussed.

Keywords: MPC, Reference trajectory, Trajectory tracking, constraints, optimization, Numerical Simulation, Pixhawk Flight Controller

ACKNOWLEDGEMENT

This project is prepared in partial fulfilment of the requirement for for the the Bachelor's degree in Aerospace Engineering. First and foremost, we would like to express our sincere gratitude towards Prof. Dr. Mahesh Chandra Luintel, our project supervisor for his guidance, inspiring lectures and precious encouragement. His useful suggestions for this whole work and cooperative behaviour are sincerely acknowledged.

We would like to thank the Department of Mechanical and Aerospace Engineering, Institute of Engineering, Pulchowk Campus for providing us opportunity of collaborative undertaking which has helped us to implement the knowledge gained over these years as major project for fourth year, and develop a major project of our own that has greatly enhanced our knowledge and provided us a new experience of teamwork.

We are extremely grateful for the crucial support and guidance of Er. Rishav Mani Sharma whose kind words and constant guidance helped us navigate this project paying careful attention to intricate details. His invaluable insights greatly helped us take the project to the next level.

We extend our sincere gratitude to Research Engineer-UAS Nabin Bhandari for helping us by providing his valuable ideas during the course of our project. We would also like to thank all of our friends who have directly and indirectly helped us in doing this project. Last but not the least, we place a deep sense of appreciation to our family members who have been constant source of inspiration for us.

Any kind of suggestion or criticism will be highly appreciated and acknowledged.

TABLE OF CONTENTS

TITLE PAGE i

COPYRIGHT ii

ABSTRACT iv

ACKNOWLEDGEMENT v

TABLE OF CONTENTS ix

LIST OF FIGURES xiv

LIST OF TABLES xv

LIST OF ABBREVIATIONS xvi

1 INTRODUCTION 1

1.1 Background . . . 1

1.1.1 Brief history of MPC . . . 1

1.2 Motivation . . . 4

1.3 Objectives . . . 4

1.3.1 Main Objective . . . 4

1.3.2 Specific Objectives . . . 5

1.4 Problem statement . . . 5

1.4.1 Problem Description . . . 5

1.4.2 Proposed Solution . . . 5

1.5 Scope of Project . . . 5

1.5.1 In medicine transportation . . . 6

1.5.2 In delivery drones . . . 6

1.5.3 Emergency response . . . 6

1.6 System Requirements . . . 6

1.6.1 Hardware requirements . . . 6

1.6.2 Software requirements . . . 9

2 LITERATURE REVIEW 11

2.1 State Estimation and Localization . . . 12

3 THEORETICAL BACKGROUND 13

3.1 Non-Linear Model Predictive Control . . . 13

3.2 System modelling . . . 14

3.2.1 Reference frames . . . 14
3.2.2 Translation Kinematics . . . 15
3.2.3 Rotational Kinematics . . . 16
3.2.4 Translation Dynamics . . . 16
3.2.5 Rotational dynamics . . . 17
3.3 State-Space Model . . . 19
3.3.1 Non-linear model of system dynamics . . . 19
3.3.2 Linear model of system dynamics . . . 20
3.3.3 Augmented state space matrices . . . 21
3.3.4 Controllability . . . 21
3.4 Trajectory Generation . . . 22

4 METHODOLOGY 23
4.1 Development of MPC Model . . . 24
4.2 Implementation of MPC . . . 25
4.3 Receding Horizon control . . . 26
4.4 Solver selection . . . 26
4.5 Problem Definition for Nonlinear MPC . . . 27
4.5.1 System Architecture of control . . . 27
4.5.2 Overall Structure . . . 28
4.5.3 Altitude Controller . . . 28
4.5.4 Position Controller . . . 28
4.5.5 Attitude Controller . . . 28
4.6 Interaction and Feedback . . . 29
4.7 Cost Function . . . 29
4.8 Validation in Implementaton of mathematical model . . . 31
4.9 Pixhawk Architecture . . . 36
4.10 PX4 flight Stack . . . 37
4.11 Problem Definition for Linear MPC . . . 38
4.11.1 State matrices and vectors . . . 38
4.12 Constraints . . . 39
4.13 Cost Function(J) . . . 41
4.14 Hildreth's quadratic programming method . . . 41
4.15 Experimental Setup . . . 42

5 RESULTS AND DISCUSSION 43
5.1 Numerical Simulation of Non-linear MPC . . . 43
5.1.1 Trajectory tracking simulation . . . 43
5.1.2 Circular trajectory . . . 43
5.1.3 Square shaped Trajectory . . . 46

5.1.4 Linear Trajectory . . . 50
5.1.5 Triangular Helix shaped Trajectory . . . 53
5.1.6 Spiral Trajectory simulated for T = 100 seconds . . . 56
5.2 SITL and Hardware implementation of Linear MPC . . . 57
5.2.1 Software in the Loop (SITL) . . . 57
5.2.2 SITL Test MPC Based PX4 Firmware . . . 58
5.2.3 Building PX4 Firmware . . . 61
5.2.4 Flight Tests . . . 62
5.2.5 Unmodelled Disturbance Rejection Test Flight Results . . . 74
5.3 Limitations . . . 77
5.4 Problems Faced . . . 77
5.5 Budget Expenses . . . 78

6 CONCLUSION AND FUTURE ENHANCEMENT 79
6.1 Conclusion . . . 79
6.2 Scope for Future Enhancement . . . 79

REFERENCES 81

A APPENDIX A 83
A.1 Parameter Definition . . . 83
A.1.1 Mass of quadcopter . . . 83
A.1.2 Dimensions of quadcopter . . . 83
A.1.3 Moments of Inertia of quadcopter . . . 83
A.1.4 Drag Coefficient . . . 86
A.1.5 Load cell . . . 86
A.1.6 Thrust Coefficient . . . 86

B APPENDIX B 88
B.1 MATLAB Program . . . 88
B.1.1 main.m . . . 88
B.1.2 quadcopter.m . . . 91
B.1.3 RK4.m . . . 93
B.2 Python Program for non linear optimization . . . 93
B.2.1 main.py . . . 93
B.2.2 Quadrotor.py . . . 102
B.2.3 MPCController.py . . . 105
B.2.4 Plotting.py . . . 112

C APPENDIX C 113
C.1 PX4 Firmware . . . 113

C.1.1 mc_att_control.cpp 113
C.1.2 CMakeLists.txt 132

List of Figures

1.1 Trend in use of Model Predictive Control(MPC) as system control strategy (Source: Google Books Ngram Viewer) 2

1.2 Percentage of survey participants who indicated whether a control technology had shown to have a significant impact in practice (”Current Impact”) or was anticipated to do so during the following five years (”Future Impact”)[1] 2

1.3 Sinusoidal trajectory tracking result for PID, NMPC, and NNMPC controllers in three axes[2] 3

1.4 IMU 7

1.5 Barometer 7

1.6 GPS Module 8

1.7 Pixhawk 9

3.1 Principle of MPC[3] 13

3.2 Basic NMPC control loop[3] 13

3.3 Earth and Body frame of Reference [2] 14

4.1 Methodology Flowchart 23

4.2 Development of MPC Model 24

4.3 Implementation of MPC 25

4.4 System architecture of control 27

4.5 Reference control inputs 32

4.6 Simulated Control inputs 33

4.7 Reference linear position 33

4.8 Simulated linear Position 34

4.9 Reference angular position 34

4.10 Simulated angular positions 35

4.11 PX4 Architecture 36

4.12 PX4 FLIGHT STACK 37

4.13 Cascaded control architecture 38

4.14 Assembled qudcopter 42

5.1 Circular Trajectory Followed by Quadcopter 44

5.2 Position of Quadcopter 45

5.3 Attitude of Quadcopter 45

5.5 Total Thrust and Torques 45

5.4 Linear velocities along the x,y,z axis 46

5.6 Angular velocities 46

5.7 Square Reference path followed by quadcopter 47

5.8 Square Reference path followed by quadcopter(side view) 47

5.9 Attitude of quadcopter 48

5.10 Position of quadcopter along x,y and z axis 48

5.11 Linear velocities of quadcopter 49

5.12 Angular velocities of quadcopter 49

5.13 Total Thrust and torque on quadcopter 49

5.14 Linear Reference path followed by quadcopter 50

5.15 Positions of quadcopter 51

5.16 Attitude of quadcopter 51

5.17 Angular velocities of quadcopter 52

5.18 Linear velocities of quadcopter 52

5.19 Total Thrust of quadcopter 52
5.20 Triangular Helix Trajectory for quadcopter 53
5.21 Position of quadcopter 54
5.22 Attitude of quadcopter 54
5.23 Linear velocities 55
5.24 Angular velocities 55
5.25 Total Thrust of quadcopter 56
5.26 Helical Trajectory 56
5.27 Gazebo simulation 58
5.28 Circular Trajectory 58
5.29 Roll Angle 59
5.30 Roll Angular Rate 59
5.31 Pitch angle 59
5.32 Pitch Angular Rate 60
5.33 Yaw Angle 60
5.34 Yaw Rate 60
5.35 Build window for the PX4 file 61
5.36 Tracking performance for Mission path 1 62
5.37 Roll Angle 63
5.38 Roll Angular Rate 63
5.39 Pitch Angle 63
5.40 Pitch Angular Rate 64
5.41 Yaw Angle 64

5.42 Yaw Angular Rate 64
5.43 Mission path 65
5.44 Roll Angle 65
5.45 Roll Angular Rate 66
5.46 Pitch Angle 66
5.47 Pitch Angular Rate 66
5.48 Yaw Angle 67
5.49 Yaw Angular Rate 67
5.50 Mission path 3 67
5.51 Roll Angle 68
5.52 Roll Angular Rate 68
5.53 Pitch Angle 68
5.54 Pitch Angular Rate 69
5.55 Yaw Angle 69
5.56 Yaw Angular Rate 69
5.57 Trajectory tracking performance 70
5.58 MPC Roll Angle 70
5.59 PID Roll Angle 70
5.60 MPC Roll Angular Rate 71
5.61 PID Roll Angular Rate 71
5.62 MPC Pitch Angle 71
5.63 PID Pitch Angle 72
5.64 MPC Pitch Angular Rate 72

5.65 PID Pitch Angular Rate 72

5.66 MPC Yaw Angle 73

5.67 PID Yaw Angle 73

5.68 MPC Yaw Angular Rate 73

5.69 PID Yaw Angular Rate 74

5.70 Roll Angle With Wind 75

5.71 Roll Angle Without Wind 75

5.72 Pitch Angle With Wind 75

5.73 Pitch Angle Without Wind 76

5.74 Yaw Angle With Wind 76

5.75 Yaw Angle Without Wind 76

A.1 Bifilar Pendulum Experiment for MOI determination 84

List of Tables

4.1 Minimum and Maximum Values for the constraints 30

4.2 Parameter values for simulation 31

4.3 Parameter values for simulation 42

5.1 Budget Expenditure 78

A.1 Time for oscillations about X-axis 84

A.2 Time for oscillations about Z-axis 84

A.3 Setup specifications 85

LIST OF ABBREVIATIONS

MPC	Model Predictive Control
UAV	Unmanned Aerial Vehicle
NMPC	Non-Linear Model Predictive Control
ESC	Electronic Speed Controller
ROS	Robot Operating System
LIDAR	Light Detection and Ranging
GPS	Global Positioning System
IPOPT	Interior Point Optimizer
IMU	Inertial Measurement Unit
SITL	Software-In-The-Loop
MOI	Moment Of Inertia
CG	Center of Gravity

CHAPTER 1: INTRODUCTION

1.1. Background

In recent years, the development of autonomous UAVs has seen a surge of interest in its effectiveness in addressing a wide range of logistical challenges and research and development as well. These systems have a huge potential to transform sectors including e-commerce, emergency response, medical commodities delivery, and humanitarian aid through efficient provision and timely supply of goods in remote or inaccessible locations. However, guaranteeing safe and dependable mission completion for these UAVs involves substantial operational and technical difficulties. The modern-day age requires fast and efficient operation of UAVs so that they can reach the desired place within time or with the shortest of routes or any other constraint for that matter and that is where advanced control techniques like Model Predictive Control chime in.

1.1.1. Brief history of MPC

The concept of Model Predictive control also sometimes known as Receeding Horizon Control, originated in the 1970s when researchers, prominently from the process control industry started exploring the avenues of optimizing future control inputs over a finite prediction horizon to improve closed-loop system performance. With the evolution of MPC, its applicability in handling complex, multivariable, constrained systems also increased and eventually, it was also adopted by the aerospace industry to solve challenges pertaining to the guidance and control of aircraft and spacecraft systems. MPC started to appear in aircraft applications in the 1980s and 1990s, with a primary focus on navigation, guiding, and trajectory tracking. Using developments in processing power and optimization algorithms, the 2000s saw even further expansion. MPC became widely used in aircraft, extending to unmanned systems and autonomous aerial vehicles, proving to be an effective instrument for improving the functionality and management of aerospace systems.

The figure below is the result of a survey conducted by the IFAC Industry Committee[1]. When compared to other crucial control technologies like robust control, adaptive control, and nonlinear control—often referred to as the ”crown jewels” of control theory—the survey indicates a clear recognition of the significant impact of model predictive control (MPC), both currently and in the near future. What sets MPC apart from other theoretical breakthroughs is that its roots are in industrial implementation rather than academic study. The survey emphasizes how vital system identification and estimation are to maximum principle correction (MPC) and other feedback control methods.

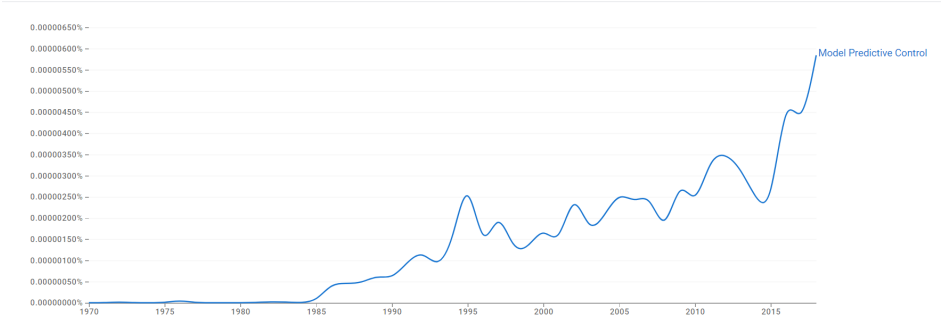


Figure 1.1: Trend in use of Model Predictive Control(MPC) as system control strategy (Source: Google Books Ngram Viewer)

	Current Impact	Future Impact
Control Technology	%High	%High
PID control	91%	78%
System Identification	65%	72%
Estimation and filtering	64%	63%
Model-predictive control	62%	85%
Process data analytics	51%	70%
Fault detection and identification	48%	78%
Decentralized and/or coordinated control	29%	54%
Robust control	26%	42%
Intelligent control	24%	59%
Discrete-event systems	24%	39%
Nonlinear control	21%	42%
Adaptive control	18%	44%
Repetitive control	12%	17%
Hybrid dynamical systems	11%	33%
Other advanced control technology	11%	25%
Game theory	5%	17%

Figure 1.2: Percentage of survey participants who indicated whether a control technology had shown to have a significant impact in practice (”Current Impact”) or was anticipated to do so during the following five years (”Future Impact”)[1]

Implementation of predictive control for optimal trajectory tracking is one of the areas where MPC has shown significant promise. If it can track down the provided trajectory efficiently with as little deviation from the intended path as possible then we could replace traditional control techniques and save time as well as increase the efficiency of our system. The following figure shows that the performance of MPC in trajectory tracking is visibly better than traditional control techniques like PID which is prominently used. Motion planning and op-

timization of trajectories for effective and obstacle-free navigation is a crucial component of autonomous UAV systems. The intricacy and dynamic nature of real-world conditions often prove difficult for traditional methods to tackle. Model Predictive Control (MPC), therefore, emerges as a viable strategy for overcoming these difficulties since it can successfully optimize trajectories while accounting for the dynamics, constraints, and uncertainties of the system, including its ability to handle multiple inputs and outputs.

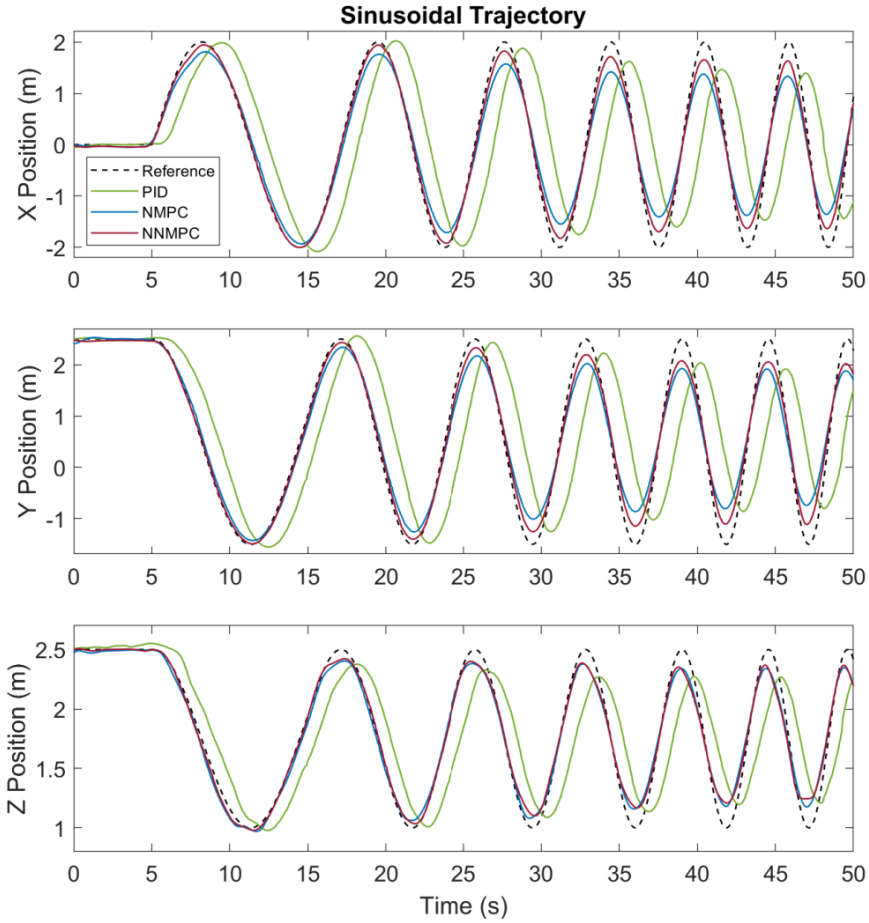


Figure 1.3: Sinusoidal trajectory tracking result for PID, NMPC, and NNMPC controllers in three axes[2]

MPC, as a control technique, closely depends on dynamic models to predict the future behavior of the system. These models and real-time sensor data enable the system to continuously adapt and optimize the trajectory in response to changing external conditions within the specified constraints. By incorporating accurate models and predictive capabilities, MPC offers a superior alternative to traditional open-loop or rule-based methods, allowing for more precise and efficient trajectory planning.

1.2. Motivation

Unmanned aerial vehicles (UAVs) have opened up intriguing possibilities for changing the fields of logistics and delivery operations. However, in order to fully realize the potential of autonomous UAV systems while accounting for the nonlinear system behavior, considerable technical obstacles must be overcome. Particularly when it comes to accessing rural or challenging-to-reach regions, traditional control methods frequently experience inefficiencies. As a result, costs rise, mission times increase, and the supply of necessities is constrained.

A need for robust control technologies is even more essential given the recent rising development towards the field of delivery services that are quicker and more effective, as well as the rise in e-commerce and consumer expectations. These control technologies must streamline delivery routes, cut down on delivery times, and guarantee the highest levels of dependability and safety, which can be achieved by the development of robust optimal trajectory generation techniques like MPC.

Additionally, current trajectory planning and optimization methods for autonomous UAV systems frequently rely on planned routes or simplistic waypoint navigation, which has a limited ability to adapt and respond to real-time impediments and environmental circumstances. To overcome these drawbacks, Model Predictive Control (MPC) emerges as a reliable control method that can take complicated system dynamics, restrictions, and uncertainties into account in real time. By utilizing MPC-based trajectory optimization, we can create delivery trajectories that are ideal and adaptable to changing conditions. By anticipating and avoiding hazards like obstructions, bad weather, and other potential dangers, this strategy will increase operational effectiveness, save energy use, and increase safety.

1.3. Objectives

The primary and secondary objectives of this project are mentioned below:

1.3.1. Main Objective

To implement Model Predictive Control (MPC)-based optimal trajectory tracking for autonomous quadrotor systems.

1.3.2. Specific Objectives

1. To develop a quadrotor system.
2. To obtain optimal control actions using the MPC algorithm.
3. To implement the MPC-based control system in the quadrotor.
4. To test the robustness of the MPC-controlled quadcopter to unmodeled disturbances.

1.4. Problem statement

Detailed problem description and its solution are mentioned below:

1.4.1. Problem Description

Autonomous UAVs with traditional linear control techniques suffer from inefficient trajectory tracking and their inability to address the system's nonlinear behavior and add input and output constraints as an integral part of the control system. Hence, they cannot produce the best control actions. Further, current solutions have limitations in terms of flight efficiency and accurate trajectory tracking.

1.4.2. Proposed Solution

Autonomous UAVs using the MPC-based control approach has the potential to incorporate a dynamic model of the system and can explicitly account for nonlinear behaviors and constraints on variables, making it suitable for controlling complex and constrained processes. The approach will optimize metrics like time and trajectory smoothness while satisfying constraints like dynamic limits.

1.5. Scope of Project

The scope of this project includes the development, implementation, and evaluation of UAVs by Model Predictive Control-Based Trajectory Optimization. The implication of this approach can have various practical applications in different domains which some are described below:

1.5.1. In medicine transportation

The developed quadcopter can be used in uphill and downhill terrain to transport medicines in hilly parts which will not only save time but also manpower that would otherwise be used for transportation.

1.5.2. In delivery drones

MPC-based approach has not been prominently researched and developed for the specific purpose of autonomous delivery systems. The project can provide a framework for future research on this topic.

1.5.3. Emergency response

MPC-based quadcopter can quickly provide medical supplies to disaster-affected areas or accident sites. The outcomes have the potential to unlock widespread adoption of drone-based logistics by achieving more efficient, safer, and scalable autonomous aerial delivery operations through advanced control and optimization techniques. Further research directions include enabling beyond visual line-of-sight deliveries, integrating drones into larger autonomy systems, and fully unlocking collision avoidance systems among others.

1.6. System Requirements

The system requirements to carry out this project involve:

1.6.1. Hardware requirements

Sensing Units

The Pixhawk is equipped with a variety of sensors to record orientation, altitude, and quadcopter condition information.

1. **IMU**

   IMU is a sensor package that typically includes accelerometers and gyroscopes(sometimes magnetometers). Accelerometers determine linear acceleration in three dimensions (x, y, and z) and give information about the direction and speed changes of the quadcopter.

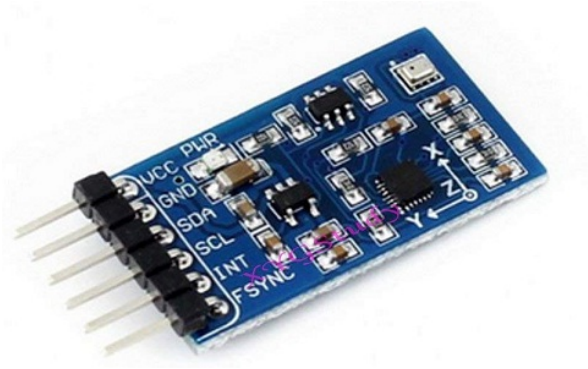


Figure 1.4: IMU

Gyroscopes are used to measure the angular velocity or rotational rate around the axes of a quadcopter. They aid in identifying angular motion or changes in orientation.IMU frequently employ sensor fusion methods, like as sensor fusion algorithms or Kalman filtering, to combine input from many sensors and get precise quadcopter location, orientation, and velocity.

2. **Magnetometer**

   A Pixhawk's magnetometer, which is installed in quadcopters, is essential for determining out the quadcopter heading or orientation. The sensors in the magnetometer are designed to identify variations in the magnetic field surrounding the quadcopter. It uses the X, Y, and Z axes to quantify the field's strength and direction. The flight controller can determine the orientation of the quadcopter with respect to the Earth's magnetic field using these measurements.

3. **Barometer**

Figure 1.5: Barometer

The barometer is the sensor that measures atmospheric pressure to determine the altitude of the quadcopter. The pressure variations are converted by the sensor into altitude values that the flight controller uses for stabilization and navigation. Accurate readings from the barometer assist in controlling the quadcopter's vertical speed, ensuring smooth and controlled ascents and descents.

4. **GPS Module**

Figure 1.6: GPS Module

The Global Positioning System, or GPS, employs satellites to transmit location and time information around the world. In order to pinpoint a receiver's exact location, it triangulates signals from several satellites. Every satellite sends out a signal that includes its position and the time it was sent. A GPS receiver determines the distance to each satellite by measuring the time interval between the transmission and receiving of signals from various satellites.GPS enables the "return-to-home" function, which allows the quadcopter to find its way back to the takeoff point on its own in the event of a signal outage or emergency.

Controller

1. **Pixhawk**

   Pixhawk 4 is a well-liked and sophisticated autopilot system . It acts as the drone's brain, managing all of its functions including flight and navigation. It has a high-performance 32bit STM32F427 Cortex M4 core with FPU having a speed of 168MHz along with 256KB RAM and a flash memory of 2MB. It has an interface of 5x UART (serial ports), one high-power capable, 2x with HW flow control,2x CAN, Spektrum DSM / DSM2 / DSM-X Satellite compatible input, Futaba S.BUS compatible input and output, PPM sum signal input, RSSI (PWM or voltage) input, I2C,SPI,3.3 and 6.6V ADC inputs and internal micro USB port and external micro USB port extension.3-Axis Gyrometer, Accelerometer, High-performance Barometer rand Magnetometer are the sensors installed in it.

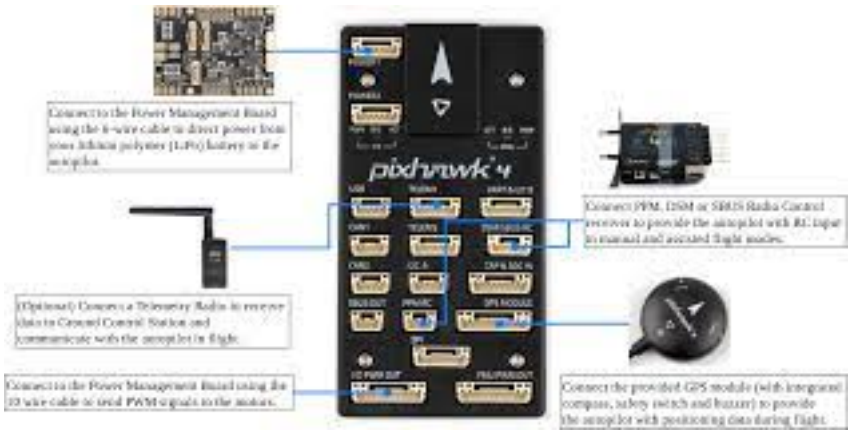


Figure 1.7: Pixhawk

1.6.2. Software requirements

These are software that we are using for doing simulations and validations:

1. **MATLAB:**

   MATLAB is a powerful software package widely used for simulation, modeling, and numerical computing. Modeling the dynamics and observing behavior of quadcopters has been possible with MATLAB's dynamic simulation system. It allows us to visually show the outcomes of simulations. Plots, graphs, and three-dimensional visualizations can be used to show sensor readings, control signals, the behavior of the quadcopter, and more. Debugging and understanding system performance is helped by this visual feedback. After the formulation of the mathematical model equation, the validation of the quadcopter mathematical model and its behavior is done using the Matlab.

2. **Python Libraries:**

   Many libraries in Python are useful for problems involving trajectory optimization. Basic mathematical and scientific computing skills are offered by NumPy and SciPy. We can create and solve optimization issues for trajectory optimization using the optimization modeling frameworks Pyomo and CasADi. Dynamic simulations and testing can be done using the physics simulation engine PyBullet. Cvxpy library is also used as an optimization framework for solving the optimal control problem for the linear model.

3. **PX4 Autopilot:**

   PX4 Autopilot is an open source flight control software which is mainly used for autonomous UAV operation. It provies an efficient and robust platform for control od UAVs along with the facility of using custom firmware. It has been widely used due to its large range of UAV configurations and user friendly ground control software for

mission planning and operations. PX4 firmware supports C++ programming language which was necessary for the hardware implementation phase in this project.

4. **QGroundcontrol:**

   QGroundcontrol is an open-source software that is used as a software interface for the control of Unmanned vehicles such as drones, UAVs, rovers, etc. It is popularly used as a go-to software for drones because of its user-friendly design. It provides a simple interface for any type of vehicle operation like mission planning, telemetry monitoring, sensor calibration, Remote Control, Battery calibration, etc. QGroundcontrol has been used in our project to upload the build file of the modified PX4 firmware and subsequently assign flight operations as mentioned above.

5. **Gazebo:**

   Gazebo is also an open-source simulation tool quite popular in the robotics field. It is also used in conjunction with PX4. Gazebo functions as a high-fidelity physics-based simulator within PX4, giving developers a virtual environment to test and validate their hardware designs, control schemes, and UAV algorithms before putting them into practice in real-world circumstances.

CHAPTER 2: LITERATURE REVIEW

D. Sotelo et al.(2020) [4] present a control strategy designed for an unmanned aerial vehicle for its automatic carrier landing system based on an LMPC scheme. In this paper, to reduce the online computational time, all the state trajectories are included in the optimal control problem as the constraints in the prediction horizon, then only a quadratic programming problem is solved and a major part of the computation is shifted offline for LMPC. We intend to integrate offline computation for faster trajectory optimization for NMPC as well.

B. Lindqvist et al.(2020) [5] present an approach in which the trajectory calculation is done from an initial condition, and fed to the NMPC as an additional input. The solver used is the nonlinear, non-convex solver Proximal Averaged Newton for Optimal Control (PANOC) and its associated software OpEn (Optimization Engine), in which a penalty method was applied to properly consider the obstacles and other constraints during navigation. We intend to use penalty in the cost function for better optimization. The use of Feedback Linearization(FBL) Control law in Nonlinear Model Predictive Horizon (NMPH).NMPC and NMPH differ in that NMPC provides inputs to a nonlinear plant[6] while NMPH computes reference trajectory to be tracked. We will explore the use of FBL in NMPC for better convergence. When designing the MPC compared to classic PID, the limitations due to the physical properties of the Load Transporting Systems on the UAV were considered. It has been seen that the performance of the MPC according to PID is visibly better[7].

K. Subbarao et al.(2015)[8] focus on the development of the Nonlinear Model Predictive Control (NMPC) formulation for an optimal controller and feasible trajectories. The study explores the use of a state-dependent coefficient (SDC) form and state-dependent Riccati equation (SDRE) based controller, combined with Quadratic Programming (QP) optimization. The research addresses the need to constrain thrust forces and angle magnitudes in quadcopters using an integrated equation. Simulation studies compare different control algorithms, and it is observed that the NMPC controller, with its faster reaction and better tracking performance, outperforms other controllers. The findings highlight the effectiveness of the NMPC approach in achieving accurate and efficient trajectory control for autonomous aerial systems.

T.lukkenon et al.(2011)[9] focus on the modelling of the quadcopter dynamics model for the design of control of quadcopter.The differential equations for the quadcopter dynamics mathematical model were obtained using the Euler-Lagrange and Newton-Euler equations.Matlab was used to simulate a quadcopter's flight in order to validate the model.

The position controller and attitude controller of the quadcopter is presented in [10]. The quadcopter's roll, pitch, and yaw angles, as well as its altitude, are all controlled by the at-

titude controller, while the position controller controls the quadcopter's translational motion in space. The control loops for X- and Y-positions were coupled together, as were the control loops for altitude, roll, pitch, and yaw. PID controllers were used to control altitude, and PD controllers were used to control roll, pitch, yaw, X, and Y positions.

There has also been development of a novel approach of control for a quadrotor helicopter using Model Predictive Control (MPC) technique[11]. The MPC technique is employed with a linear prediction model obtained by linearizing the dynamics of the plant. The control strategy focuses on stabilizing the system and achieving position control with a single-layer predictive controller. Constraints, to which the MPC is subjected to in this paper, are imposed on the roll and pitch angles to limit their amplitudes. The paper compares the proposed MPC approach with traditional PID and backstepping control techniques, demonstrating its effectiveness in stabilization and path tracking. Overall, the study suggests that the MPC-based approach can be a viable alternative for controlling unstable and underactuated systems like quadrotor helicopters.

The feasibility of the implementation of Model Predictive Control technique in the Pixhawk flight controller[12] has been studied. Pixhawk 1 was used as the flight controller in the thesis where the implementtion of MPC was verified in the actual quadcopter module for angular rate control. Hildereth'S Quadratic Programming algorith was used for solving the optimization problem. The findings highlight that the angular rate control was able to closely follow the reference setpoint of the angular rates given to the quadcopter vehicle. For the hardware implementation phase of our project, we took this thesis as our reference since the feasibility of implementation of MPC in Pixhawk Flight Controller has been studied in this paper. For matrix and vector computation the imported Eigen C++ library is used by the author over Armadillo C++ library due to its compatibility with PX4 Autopilot[12]. We, on the other hand, have opted to use PX4-based inbuilt matrix library instead of Eigen in our project.

2.1. State Estimation and Localization

Jansen.(2014)[13] used Kalman Filter for autonomous localization and tracking of UAV. The Kalman Filter is made to address linear issues, although other approaches have been created for nonlinear situations. Several issues with the Kalman Filter are addressed by a different filtering technique known as the Extended Kalman Filter, which is described by George and Sukkarieh [14], Mao et al.[15]. The EKF can be employed in situations where systems are nonlinear rather than the Kalman Filter, which needs a linear equation for the measurement and state-transition models. Antonov et al. (2011)[16] suggested using an Unscented Kalman Filter (UKF), which directly approximates the probability density function.

CHAPTER 3: THEORETICAL BACKGROUND

3.1. Non-Linear Model Predictive Control

A finite horizon open-loop optimum control problem subject to system dynamics and constraints including states and controls is how the model predictive control issue is typically formulated. The fundamentals of model predictive control are depicted in Figure 3.1. Based on measurements made at time t, the controller predicts the dynamic behavior of the system in the future over a prediction horizon Tp and chooses the input so that an open-loop performance objective functional is optimum over a control horizon $Tc \leq Tp$

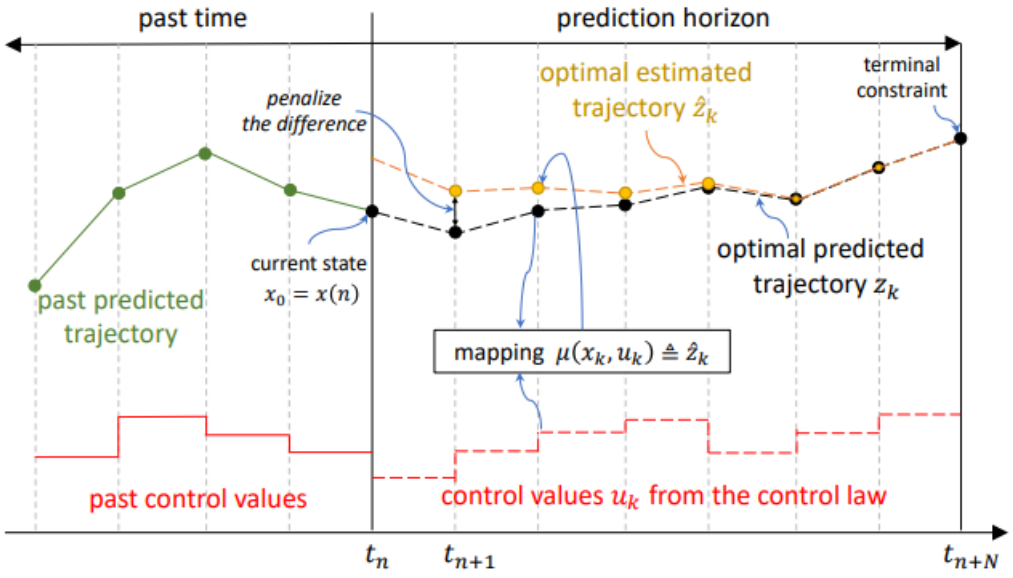


Figure 3.1: Principle of MPC[3]

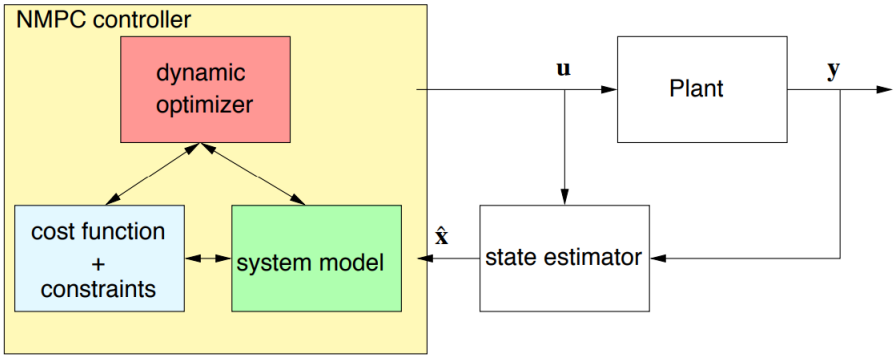


Figure 3.2: Basic NMPC control loop[3]

3.2. System modelling

In this section, the kinematics and dynamics of a quadrotor is derived based on a Newton-Euler method with the following assumptions:

. The structure is rigid and symmetrical.

. The center of gravity of the quadrotor coincides with the body fixed frame origin.

. The propellers are rigid.

. Thrust and drag are proportional to the square of propeller's speed.

3.2.1. Reference frames

To characterize the position, structure, and motion of the vehicle, frames and reference coordinates must first be established before moving on to mathematical modeling. The kinematics and dynamics of a quadcopter can be clearly understood by considering two frames of reference:

. Earth inertial frame (Inertial frame of reference).

. Body fixed frame (Non-Inertial frame of reference).

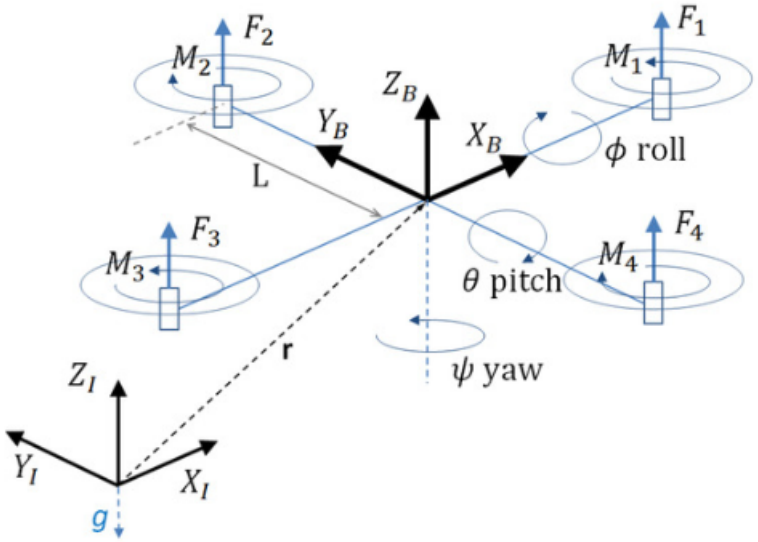


Figure 3.3: Earth and Body frame of Reference [2]

3.2.2. Translation Kinematics

Kinematic equations are obtained by geometric relationships between axes on both inertial and body frames.

Inertial Frame of reference

Inertial frames of reference are those coordinate system where the Newton's first law of motion is satisfied i.e. the body with zero net force does not accelerate. The positions of quadcopter that is x,y and z are always taken with reference to inertial frame. Similarly,angular rates p, q, and r respectively,along the x, y, and z axes are taken with reference to inertial frame. However,the roll, pitch, and yaw angles(or the Euler angles) which are represented as φ, θ, and ψ, respectively, are taken with reference to body frame.

Non-Inertial frame of reference

Non inertial frames are those which accelerates with respect to the inertial frame of references.The calculations of state variables are reasonable and more useful when they are in respect with the inertial frame.Hence ,state variables with reference to body frame are transformed to corresponding variables of inertial frames and vice-versa.

The rotation matrix from the body frame to the inertial frame is found below :
R= Rz × Ry × Rx

$$R = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta\sin\phi & \sin\theta\cos\phi & \cos\theta \end{bmatrix} \quad (3.1)$$

Since R is an orthogonal rotation matrix, the rotation matrix from the inertial frame to the body frame is $R^{-1} = R^T$.

3.2.3. Rotational Kinematics

The x, y, and z axes correspond to the angles on the body frame, which are designated as p, q, and r, respectively.Trigonometric relationships between the Euler rates $\dot{\varphi}, \dot{\theta}, \dot{\psi}$ and angular rates of body $\dot{p}$, $\dot{q}$, $\dot{r}$ is given below:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} \tag{3.2}$$

3.2.4. Translation Dynamics

Translation motion equations are described by Newton's Second Law. While in reference to body frame centrifugal force prevails,but in reference to inertial frame , centrifugal force is nullified. Thus , in inertial frame, only gravitational force,aerodynamical forces(the drag forces) and thrust affects the translation dynamics.

Hence, the translation dynamics simply becomes:

$$m\ddot{\xi} = G + R*.T_B \tag{3.3}$$

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

$$T_B = K\sum \omega_i^2 \tag{3.4}$$

We may also write

$$T_B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{3.5}$$

Linear Acceleration matrix=

$$\ddot{\xi} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \tag{3.6}$$

The same equation elaborated in matrix form is presented below:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = -g \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \frac{T}{m} \begin{bmatrix} C\psi S\theta C\varphi + S\psi S\varphi \\ S\psi S\theta C\varphi - C\psi S\varphi \\ C\theta C\varphi \end{bmatrix} \tag{3.7}$$

3.2.5. Rotational dynamics

The Euler method is used to derive the rotational equations of motion in the body frame, and the general formalism is as follows:

$$I\dot{\nu} + \nu \times (I\nu) + \Gamma = \tau \tag{3.8}$$

The external torque τ is equal to the angular acceleration of inertia I, centripetal forces , and gyroscopic forces Γ in the body frame.

The roll torque component τ_ϕ and the pitch torque component τ_θ are obtained from standard mechanics where i_2 and i_4 motors are arbitrarily chosen to be on the roll-axis while i_1 and i_3 are arbitrarily chosen to be on the pitch-axis.

$$\tau_\phi = l * k(-\omega_2^2 + \omega_4^2) \tag{3.9}$$

$$\tau_\theta = l * k(-\omega_1^2 + \omega_3^2) \tag{3.10}$$

The total torque about the z-axis is given by the sum of all the torques from each propeller as follows:

$$\tau_\psi = b * (\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \tag{3.11}$$

Therefore, the torque matrix can be written as follows:

$$\tau_B = \begin{bmatrix} l * k\left(-\omega_2^2 + \omega_4^2\right) \\ l * k\left(-\omega_1^2 + \omega_3^2\right) \\ b\left(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2\right) \end{bmatrix} \tag{3.12}$$

where,

l and b are the digonal length between two motors and drag coffiecient respectively.

Inertia matrix is given as;

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{3.13}$$

The equation (1) can be written as

$$\dot{v} = \dot{\nu} = I^{-1}\left(-\left(\begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} - I_r \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\right) \omega_\Gamma\right) + \tau \tag{3.14}$$

Therefore, rotational dynamic equations become:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{(I_{yy} - I_{zz})qr}{I_{xx}} \\ \frac{(I_{zz} - I_{xx})pr}{I_{yy}} \\ \frac{(I_{xx} - I_{yy})pq}{I_{zz}} \end{bmatrix} - I_r \cdot \begin{bmatrix} \frac{q}{I_{xx}} \\ -\frac{p}{I_{yy}} \\ 0 \end{bmatrix} \omega_\Gamma + \begin{bmatrix} \frac{\tau_\phi}{I_{xx}} \\ \frac{\tau_\theta}{I_{yy}} \\ \frac{\tau_\psi}{I_{zz}} \end{bmatrix} \tag{3.15}$$

where, $\omega_\Gamma = \omega_1 - \omega_2 + \omega_3 - \omega_4$.

The transformation matrix W^{-1} and its time derivative are then used to attract the angular accelerations in the inertial frame from the body frame accelerations.

3.3. State-Space Model

Formulating the acquired mathematical model for the quadrotor into a state space model helps in making the control problem easier to tackle.

3.3.1. Non-linear model of system dynamics

Details of a non linear model for the quadrotor system that represents its dynamics and formulation of the control problem are explained below. **Formulation of the optimal control problem**

To formulate an optimal control problem involving a state vector and its associated state equations;

We first define the state vector as

$$\begin{bmatrix} x & y & z & \phi & \theta & \psi & \dot{x} & \dot{y} & \dot{z} & \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix} \tag{3.16}$$

and then we defined the state equation as:

$$\dot{x} = f(t,x,u) = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ (\cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi)\frac{T_B}{m} \\ (\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi)\frac{T_B}{m} \\ -g + (\cos\theta\cos\phi)\frac{T_B}{m} \\ \frac{I_{YY}-I_{ZZ}}{I_{XX}}\dot{\theta}\dot{\psi} + \frac{\tau_\phi}{I_{XX}} \\ \frac{I_{ZZ}-I_{XX}}{I_{YY}}\dot{\psi}\dot{\phi} + \frac{\tau_\theta}{I_{YY}} \\ \frac{I_{XX}-I_{YY}}{I_{ZZ}}\dot{\phi}\dot{\theta} + \frac{\tau_\psi}{I_{ZZ}} \end{bmatrix} \tag{3.17}$$

where, T_B,τ_ϕ,τ_θ and τ_ψ are control inputs which are represented as U_1,U_2,U_3 and U_4 respectively below in the code.

3.3.2. Linear model of system dynamics

The linear model for the quadcopter dynamics is described in this section. The dynamical behaviour of a quadrotor is highly non-linear. But for model prective control implementation the non linear dyanamics can be discretized into a linear model which also represents the actual behaviour closely.The development of the linear model for quadcopter dynamics is introduced in this section, which precedes Model Predictive Control (MPC) implementation. Although a quadrotor exhibits much non-linearity in its inherent dynamics, approximating these characteristics into a linear model provides a practical way that closely mimics how the system behaves thus fostering ease of MPC deployment. This leads to nonlinear dynamics discretization; hence resulting to a linearized mathematical model describing the behavior of the quadrotor within discrete-time domain. This shift allows us to apply linear control methods like MPC that are suitable for real-time applications and have computational efficiency. Nonetheless, through simplification by linearizing the system retains most important behaviors of the model within operating regime of interest. The selection of discretization parameters such as careful choice of time steps and effecting other non-linearities ensure a good approximation between this simple theory and actual behavior, especially when used in control purposes at their operating limits.

The rotational motion of the quadcopter will be represented using a linear time invariant (LTI) state space model. The state space model for the rotational motion of the quadcopter is derived from equations 3.15 The state space matrices A, B and C are given below as:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 1/I_{xx} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1/I_{yy} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1/I_{zz} \end{bmatrix} \quad C = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The input vector u and the state vector x, which came from equation 3.15, are shown below.

For the MPC angular rates controller design to be implemented digitally at a certain sample

rate, the state space model must be discretized and linearized. The linear discrete model utilised in this thesis is obtained from [12] in which Taylor series expansion is used to linearise the model about the hover position of the quadcopter.

The following equations simply represent the state vector, state matrices, and input vector, indicated by the subscript m.

$$x_m(k+1) = A_m x_m(k) + B_m u(k) \tag{3.18}$$

$$y(k) = C_m x_m(k) \tag{3.19}$$

No direct feedthrough signal exists from the input to the output, thus excluding the state matrix D from equation 3.19

3.3.3. Augmented state space matrices

A linear MPC controller is created using the linear discrete state space model defined in the above equations. Equations 3.18 and 3.19 of the quadcopter model integrate integral action in order to provide offset-free tracking by applying the augmentation strategy described in The augmented state space model is

$$\begin{bmatrix} \Delta x_m(k+1) \\ y(k+1) \end{bmatrix} = \begin{bmatrix} A_m & O_{q\times n}^T \\ C_m A_m & I_{q\times q} \end{bmatrix} \begin{bmatrix} \Delta x_m(k) \\ y(k) \end{bmatrix} + \begin{bmatrix} B_m \\ C_m B_m \end{bmatrix} \Delta u(k)$$

$$\text{y(k)} = \begin{pmatrix} O_{q\times n} & I_{q\times q} \end{pmatrix} \begin{pmatrix} \Delta x_m(k) \\ y(k) \end{pmatrix}$$

where, Iq×q is the identity matrix with dimensions q × q, where q is the number of outputs. The matrix $O_{q\times n}$ is a zero matrix of size $q \times n$, where n represents the number of states or the dimensions of the state space.A, B, and C represent the augmented state matrices as described in the preceding equation.

3.3.4. Controllability

The augmented state space model's controllability needs to be confirmed. If a system has full rank (rank(CO) = n), where n is the number of states, then its controllability matrix, CO, is said to be controllable.The augmented state space model's controllability needs to be confirmed.

If a system has full rank (rank(CO) = n), where n is the number of states, then its controllability matrix, CO, is said to be controllable. The controllability matrix, CO, is as follows:

$$CO = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix}$$

where A and B are the state space matrices.

3.4. Trajectory Generation

The process of choosing the desired path or motion for a system, such as a vehicle or a robot, to follow is called trajectory generation. It entails specifying a series of stages or waypoints that the system must pass through in order to accomplish a particular goal or adhere to a set of constraints. Model Predictive Control (MPC) trajectory generation for a quadcopter entails creating an optimization problem to produce a series of control inputs that steer the quadcopter along a desired trajectory while taking its dynamics and limits into consideration. Different techniques for trajectory creation are possible, including path planning followed by smoothing and trajectory planning. Rapidly-Exploring Random Trees (RRT*) and B-spline smoothing and an optimization-based trajectory planning method where the trajectory is defined by a set of polynomials methods.

CHAPTER 4: METHODOLOGY

The methodology flowchart for this project is shown below.

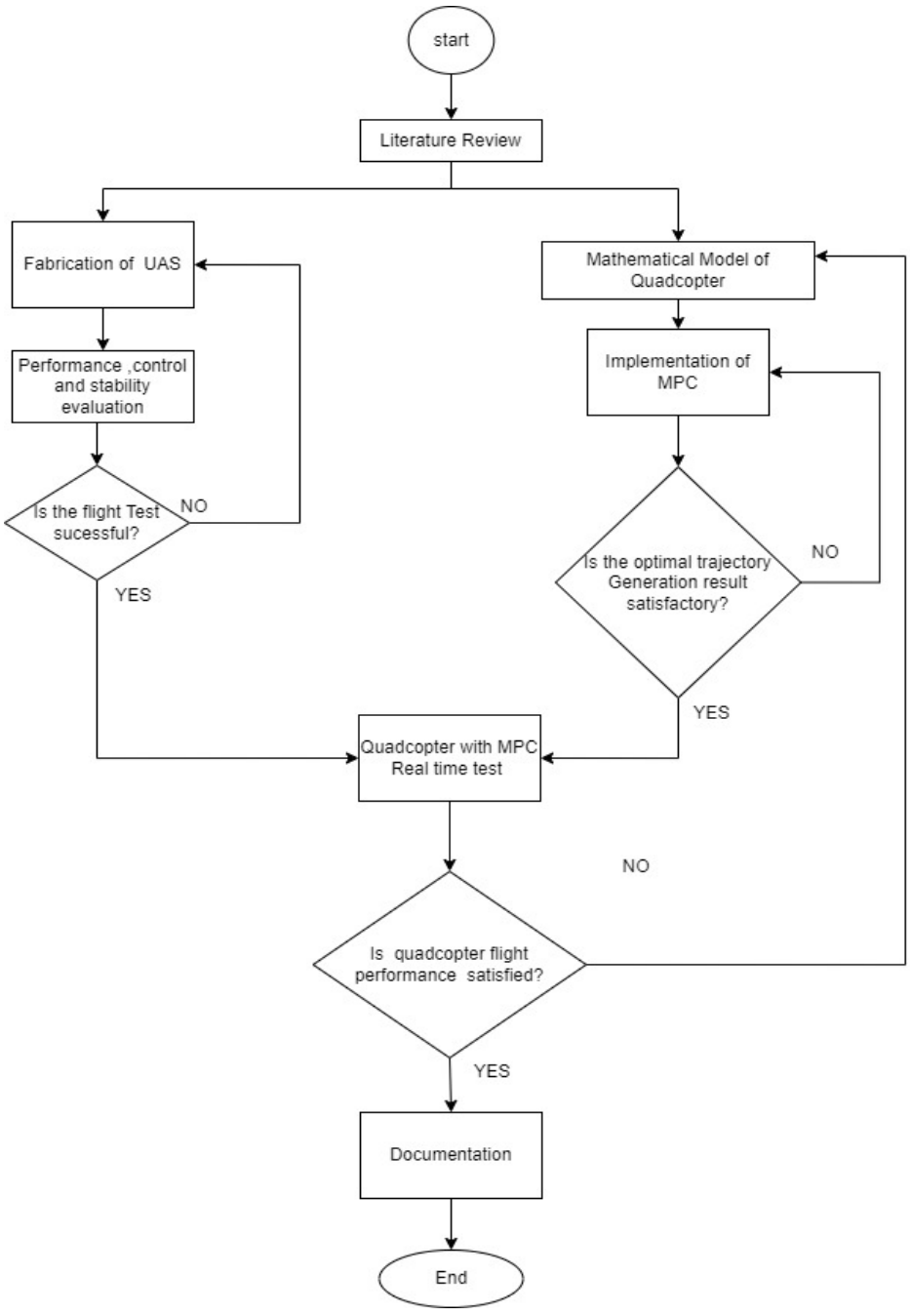


Figure 4.1: Methodology Flowchart

4.1. Development of MPC Model

Flowchart for the development of MPC model is as shown in the figure below.

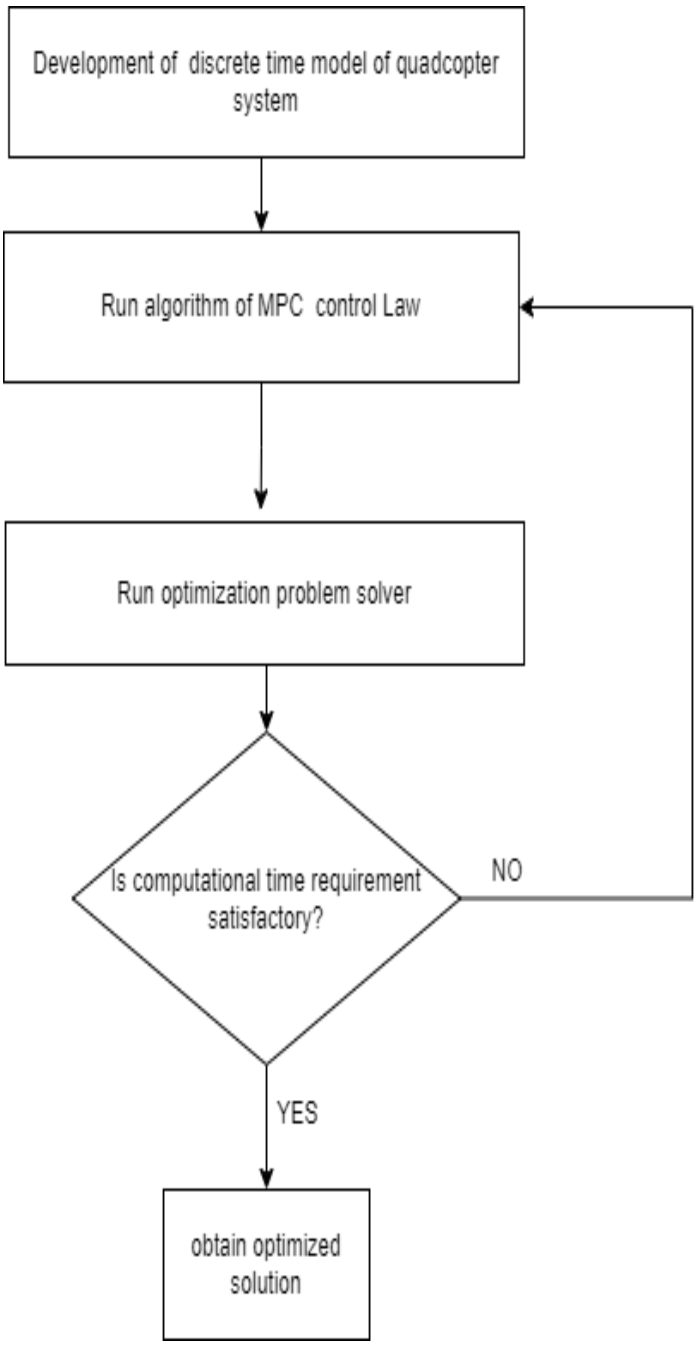


Figure 4.2: Development of MPC Model

4.2. Implementation of MPC

The flowchart for the implementation of MPC is shown below.

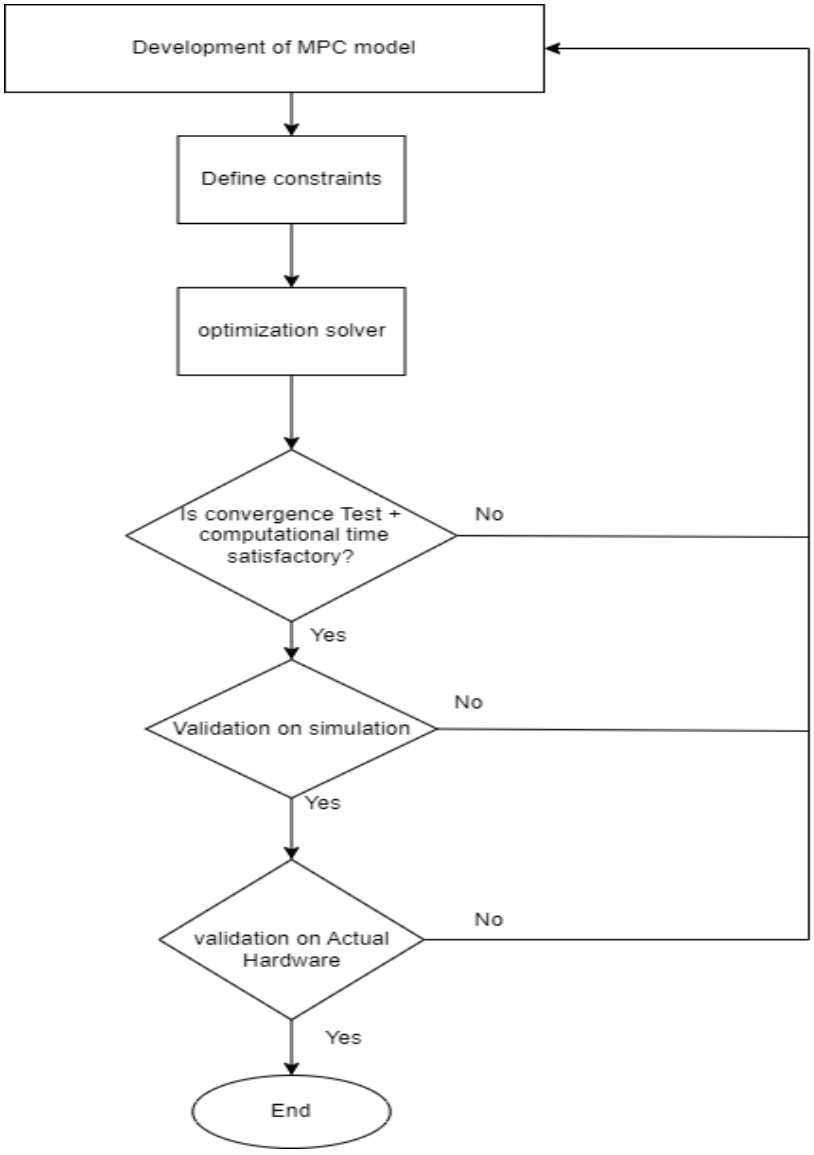


Figure 4.3: Implementation of MPC

4.3. Receding Horizon control

Receding Horizon Control, also referred to as Model Predictive Control, involves forecasting future states and control inputs through the repeated solution of a nonlinear programming problem over a shifting finite time horizon. The nonlinear programming problem is addressed within a finite time horizon (N), with implementation occurring in the current time slot($\triangle t$), followed by iterative optimization. This approach, reliant on real-time information updates, establishes a feedback control system that enhances robustness to noise and uncertainties.

4.4. Solver selection

The equation that defines the relationship between the various quantities in a quadcopter model is non-linear. For instance, several variables have to remain within certain intervals. These constraints are put in place to establish the physical capabilities such as speed, to ensure the meaningfulness of the variable, for example, mass being positive, or to provide specific limitations or constraints on Torque and thrust. These types of nonlinear programs are found in different classes of problem,among them dynamic optimization is one where we have to solve the optimization problem where constraints and objective functions are non linear. Nonlinear Model Predictive Control (NMPC) is an advanced control strategy that extends the principles of Model Predictive Control (MPC) to systems with nonlinear dynamics. In the context of tracking the path of a quadcopter, NMPC plays a crucial role in achieving accurate and adaptive control.

The significance of NMPC in tracking the path of a quadcopter lies in its ability to adaptively predict and adjust control inputs in real-time. This adaptability is crucial for handling uncertainties, disturbances, and changes in the quadcopter's dynamics during flight.

To solve the non-linear optimization problem, the optimization problem cost function along with constraints was formulated in Casadi, an open-source tool for the non-linear optimization problem. The Interior Point Optimizer (IPOPT) solver was used to solve the non-linear optimization problem. It is used to solve the non-linear optimization problems of the form:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g^L \leq g(x) \leq \hat{g} \\ & X^L \leq x \leq \hat{X} \end{aligned} \tag{4.1}$$

where $x \in \mathbb{R}$ are the optimization variables with lower and upper bounds of x^L and x^U. f :

$\mathbb{R} \to \mathbb{R}^n$ is the objective function with non-linear constraints $g(x)$. IPOPT solver can be used to solve the optimization problem with both $f(x)$ and $g(x)$ being either linear or non-linear, convex or non-convex, as long as they are twice continuously differentiable. IPOPT is a powerful tool for handling large-scale nonlinear optimization problems. When dealing with a quadcopter's complex nonlinear dynamics and the real-time nature of control strategies, it becomes crucial to have a solver that can efficiently manage the computational demands of optimization tasks. Additionally, the robustness and reliability of IPOPT are essential considerations in this regard. In an environment where uncertainties and disturbances are inherent, having a solver that consistently converges to optimal solutions ensures the stability and effectiveness of the control system.

4.5. Problem Definition for Nonlinear MPC

The problem definition for the silulation of non linear model predictive control for the quadrotor is explained in this section.

4.5.1. System Architecture of control

The quadrotor tracking system incorporates three distinct subcontrollers, namely the Altitude Controller, Position Controller, and Attitude Controller. The specifics of each sub-controller are illustrated in the accompanying figure.

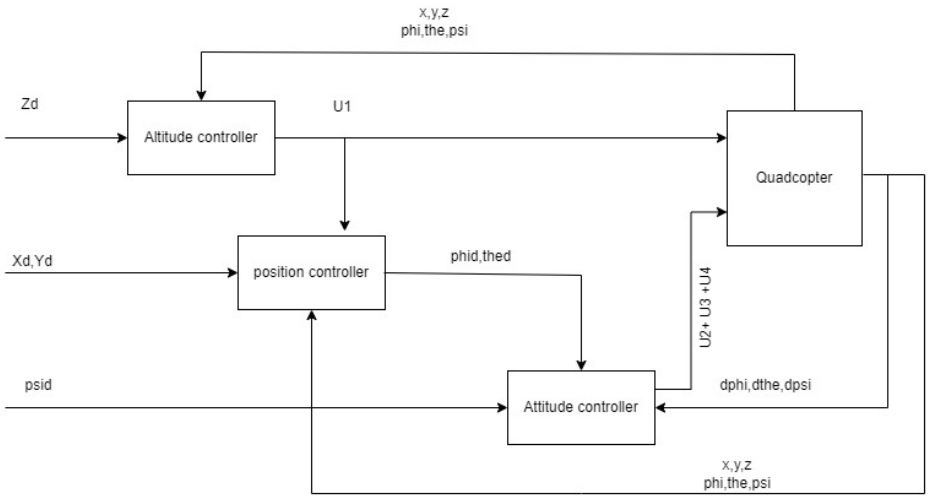


Figure 4.4: System architecture of control

4.5.2. Overall Structure

The control architecture employs a cascaded structure with three hierarchical controllers: Altitude, Position, and Attitude. This structure facilitates a top-down control flow, where higher-level controllers generate desired values for lower-level ones.

4.5.3. Altitude Controller

Goal: Regulate the quadrotor's altitude (z-axis position).
Inputs:

- Desired altitude (z_d)
- Quadrotor's current linear and angular position.

Output: Thrust command (U_1) to adjust vertical motion.

4.5.4. Position Controller

Goal: Control the quadrotor's horizontal position (x, y).
Inputs:

- Desired horizontal positions (x_d, y_d)
- Quadrotor's current linear and angular position and the thrust command.

Outputs: Desired roll and pitch angles (ϕ_d, θ_d) to achieve the desired horizontal movement.

4.5.5. Attitude Controller

Goal: Maintain the quadrotor's desired attitude (orientation angles: ϕ, θ, ψ).
Inputs:

- Desired roll, pitch, and yaw angles (ϕ_d, θ_d, ψ_d) from Position.
- Quadrotor's current angular rates ($\dot{\phi}, \dot{\theta}, \dot{\psi}$)

Outputs: Torque commands (u_2, u_3, u_4) to control rotation rates around the three axes.

4.6. Interaction and Feedback

The Altitude controller directly instructs the quadrotor by providing a thrust command (u_1). Meanwhile, the Position controller with the thrust command, shares the desired roll and pitch angles (ϕ_d, θ_d) with the Attitude controller where we have already provided the desired yaw angle (psi_d). Consequently, the Attitude controller generates torque commands (u_2, u_3, u_4) based on both the desired and current attitude angles, sending them to the quadrotor for execution. To maintain precision in the quadrotor's movement, continuous measurements of its current position and attitude ($x, y, z, \phi, \theta, \psi$) are actively collected and fed back to the controllers, enabling real-time adjustments and fine-tuning of control strategies. The division of control jobs into easier-to-manage subtasks is made possible by the cascaded structure. It is possible to effectively control various parts of the quadrotor's motion since each controller has its own time scale. The quadrotor's accuracy in following the intended trajectories is guaranteed by the feedback loops

4.7. Cost Function

To optimize control inputs across a finite time horizon, Model Predictive Control (MPC) uses a cost function. The system's performance is measured by the cost function using control input and state deviations from the expected states. It takes constraints given to our system into account and seeks to minimize these variances and offers a versatile and adaptive control approach for dynamic systems by iteratively solving the optimization issue at each time step.

Next, we design the cost function(J) for the following set of problems for our quadrotor system.

- The trajectory to be followed by the quadrotor is $X(t = t_f)$ = X_f, meaning that the reference trajectory must be followed as closely as possible
- The control effort U is to be kept as low as possible.

Following cost function is designed to incorporate the above objective:

$$Minimize : J(u,x) = \sum_{k=0}^{N-1} \left[(x_k - x_{ref,k})^T Q (x_k - x_{ref,k}) + (u_k - u_{ref,k})^T R (u_k - u_{ref,k}) \right] \quad (4.2)$$

where N is the prediction horizon

Constraints

$\dot{x} = f(x,u)$

$X(t = t_0) = X_0$

$z_{\min} \leq z_k \leq z_{\max}$

$\phi_{\min} \leq \phi \leq \phi_{\max}$

$\theta_{\min} \leq \theta \leq \theta_{\max}$

$\dot{x}_{\min} \leq \dot{x}_k \leq \dot{x}_{\max}$

$\dot{y}_{\min} \leq \dot{y}_k \leq \dot{y}_{\max}$

$\dot{z}_{\min} \leq \dot{z}_k \leq \dot{z}_{\max}$

$\dot{\phi}_{\min} \leq \dot{\phi} \leq \dot{\phi}_{\max}$

$\dot{\theta}_{\min} \leq \dot{\theta} \leq \dot{\theta}_{\max}$

$\dot{\psi}_{\min} \leq \dot{\psi} \leq \dot{\psi}_{\max}$

$thrust_{\min} \leq thrust_k \leq thrust_{\max}$

$\tau_{\phi_{\min}} \leq \tau_\phi \leq \tau_{\phi_{\max}}$

$\tau_{\theta_{\min}} \leq \tau_\theta \leq \tau_{\theta_{\max}}$

$\tau_{\psi_{\min}} \leq \tau_\psi \leq \tau_{\psi_{\max}}$

Relevent values in radian, radian/sec, meter, meter/sec, N/m

Variable	Min Value	Max Value
z_k	0	inf
ϕ	$-\frac{\pi}{2}$	$\frac{\pi}{2}$
θ	$-\frac{\pi}{2}$	$\frac{\pi}{2}$
$\dot{x}_k$	-20	20
$\dot{y}_k$	-20	20
$\dot{z}_k$	-20	20
$\dot{\phi}$	-1	1
$\dot{\theta}$	-1	1
$\dot{\psi}$	-0.15	0.15
$_k$	0	15
τ_ϕ	-10	10
τ_θ	-10	10
τ_ψ	-10	10

Table 4.1: Minimum and Maximum Values for the constraints

Define the Reference Trajectory

The objective of the quadcopter is to follow the reference trajectory. An example reference trajectory is given as:

$$p_{\text{ref}}(t) := \begin{bmatrix} X_{\text{ref}}(t) \\ Y_{\text{ref}}(t) \\ Z_{\text{ref}}(t) \end{bmatrix} = \begin{bmatrix} \sin 2t \\ 1 - \cos 2t \\ z_0 + 2\sin t \end{bmatrix} \tag{4.3}$$

4.8. Validation in Implementaton of mathematical model

The mathematical model uses governing equations defined in Translation Dynamics section and Rotational Dynamics section.[9] The values of constants required in governing equations are taken as given in Table 4.1 below.

Table 4.2: Parameter values for simulation

Parameter	**Value**	**Unit**
g	9.81	m/s^2
m	0.468	kg
l	0.225	m
k	2.980×10^{-6}	
b	1.140×10^{-7}	
Im	3.357×10^{-5}	kg m^2
Ixx	4.856×10^{-3}	kg m^2
Iyy	4.856×10^{-3}	kg m^2
Izz	8.801×10^{-3}	kg m^2
Ax	0.25	kg/s
Ay	0.25	kg/s
Az	0.25	kg/s

For validation in implementation of mathematical model, a Matlab code was written and named main, which is provided in annex.This code calls other two functions :quadcopter , which is mathematical model , and RK4, which uses a numerical method called Runge-Kutta 4 to calculate lower order derivatives from given higher order derivatives.

The quadcopter is initially in a stable state in which the values of all positions and angles are zero, the body frame of the quadcopter is congruent with the inertial frame. The total thrust is equal to the hover thrust, the thrust equal to gravity. The simulation progresses at 0.0001 second intervals to total elapsed time of two seconds.

For the first 0.25 seconds the quadcopter ascended by increasing all of the rotor velocities from the hover thrust. Then, the ascend is stopped by decreasing the rotor velocities significantly for the following 0.25 seconds. Consequently the quadcopter ascended 0.1 meters in the first 0.5 seconds. After the ascend the quadcopter is stable again.

Next the quadcopter is put into a roll motion by increasing the velocity of the fourth rotor and decreasing the velocity of the second rotor for 0.25 seconds. The acceleration of the roll motion is stopped by decreasing the velocity of the fourth and increasing the velocity of the second rotor for 0.25 seconds. Thus, after 0.5 seconds in roll motion the roll angle had increased approx. 25 degrees. Because of the roll angle the quadcopter accelerated in the

direction of the negative y-axis.

Then, similar to the roll motion, a pitch motion is created by increasing the velocity of the third rotor and decreasing the velocity of the first. The motion is stopped by decreasing the velocity of the third rotor and increasing the velocity of the first rotor. Due to the pitch movement, the pitch angle had increased approximately 22 degrees. The acceleration of the quadcopter in the direction of the positive x-axis is caused by the pitch angle.

Finally, the quadcopter is turned in the direction of the yaw angle by increasing the velocities of the first and the third rotors and decreasing the velocities of the second and the fourth rotors. The yaw motion is stopped by decreasing the velocities of the first and the third rotors and increases the velocities of the second and the fourth rotors. Consequently, the yaw angle increases approximately 10 degrees

The graph below is plotted with control input in y-axis and time in x-axis.Fig 4.3 is figure given as reference[9] while fig 4.4 is obtained from simulation of our mathematical model.

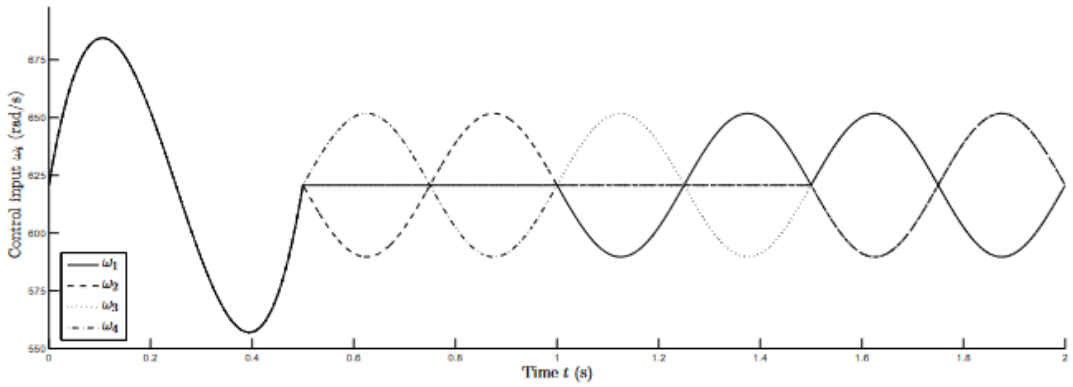


Figure 4.5: Reference control inputs

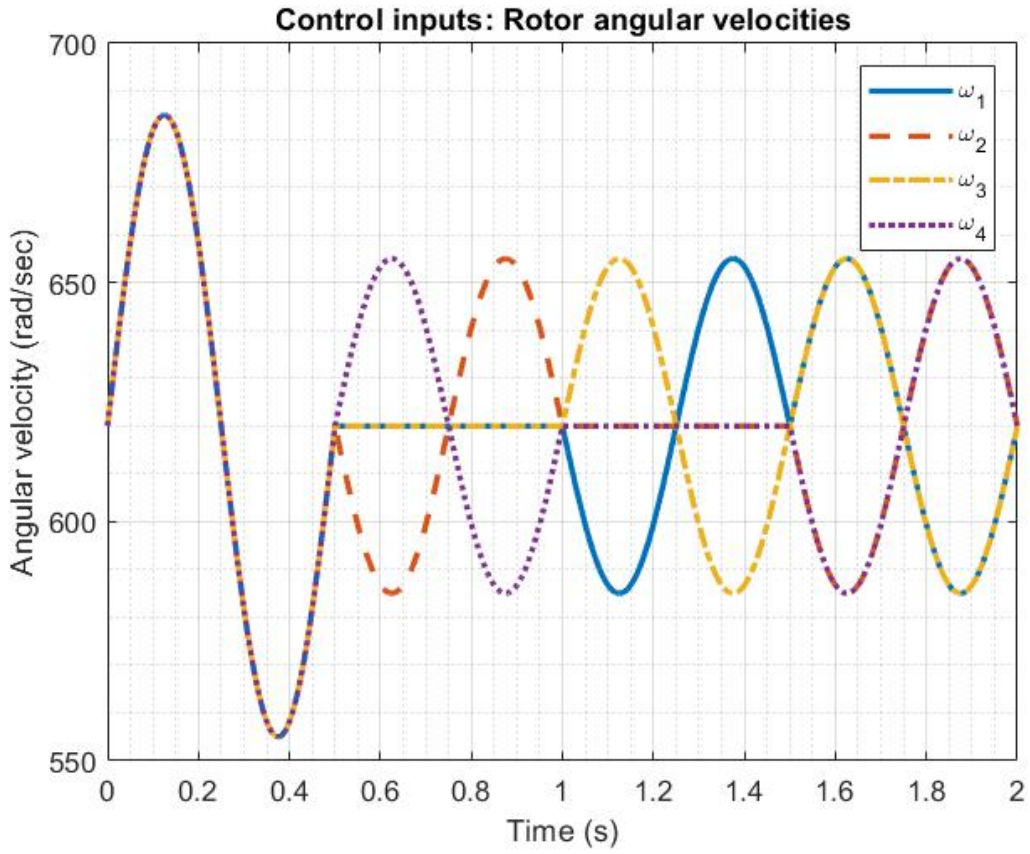


Figure 4.6: Simulated Control inputs

The graph below is plotted with position in y-axis and time in x-axis.Fig 4.5 is figure given as reference[9] while fig 4.6 is obtained from simulation of our mathematical model.

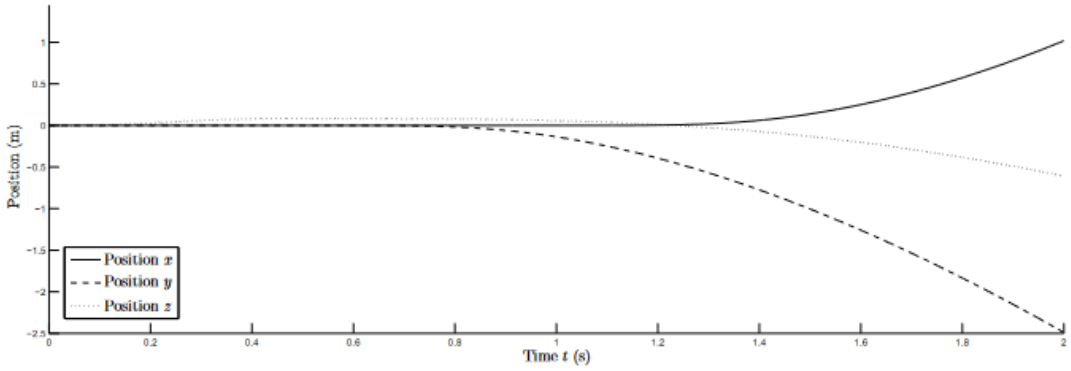


Figure 4.7: Reference linear position

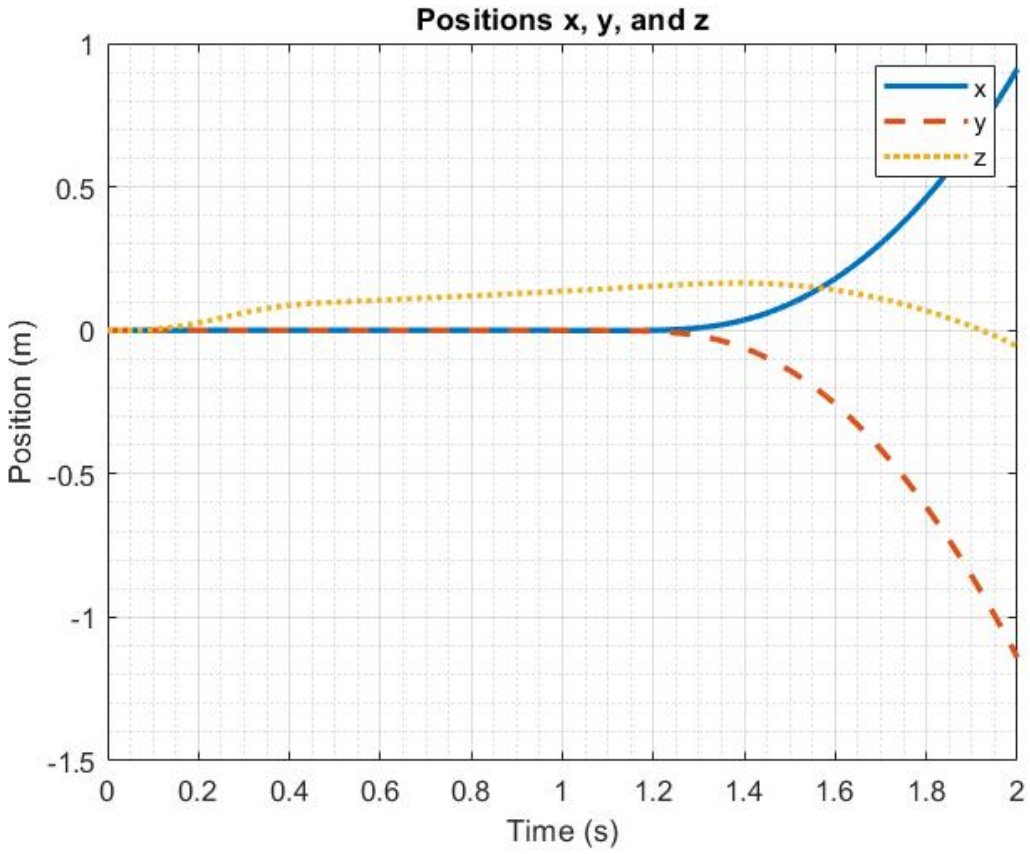


Figure 4.8: Simulated linear Position

The graph below is plotted with Angles in y-axis and time in x-axis.Fig 4.7 is figure given as reference[9] while fig 4.8 is obtained from simulation of our mathematical model.

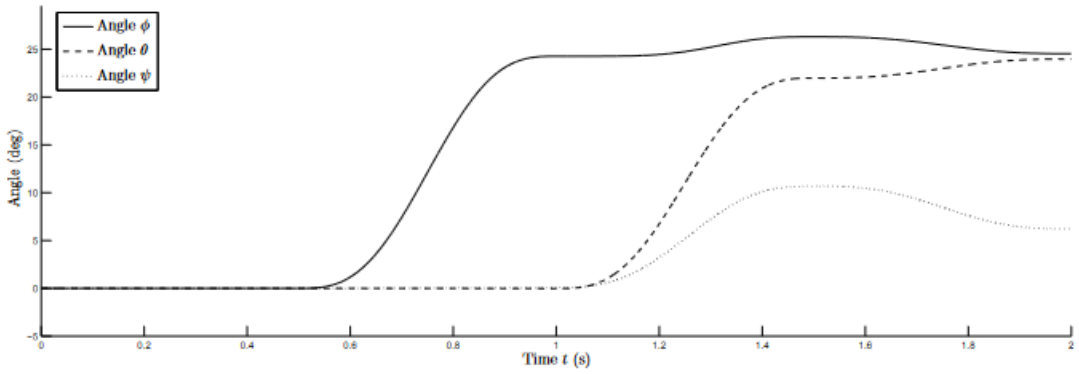


Figure 4.9: Reference angular position

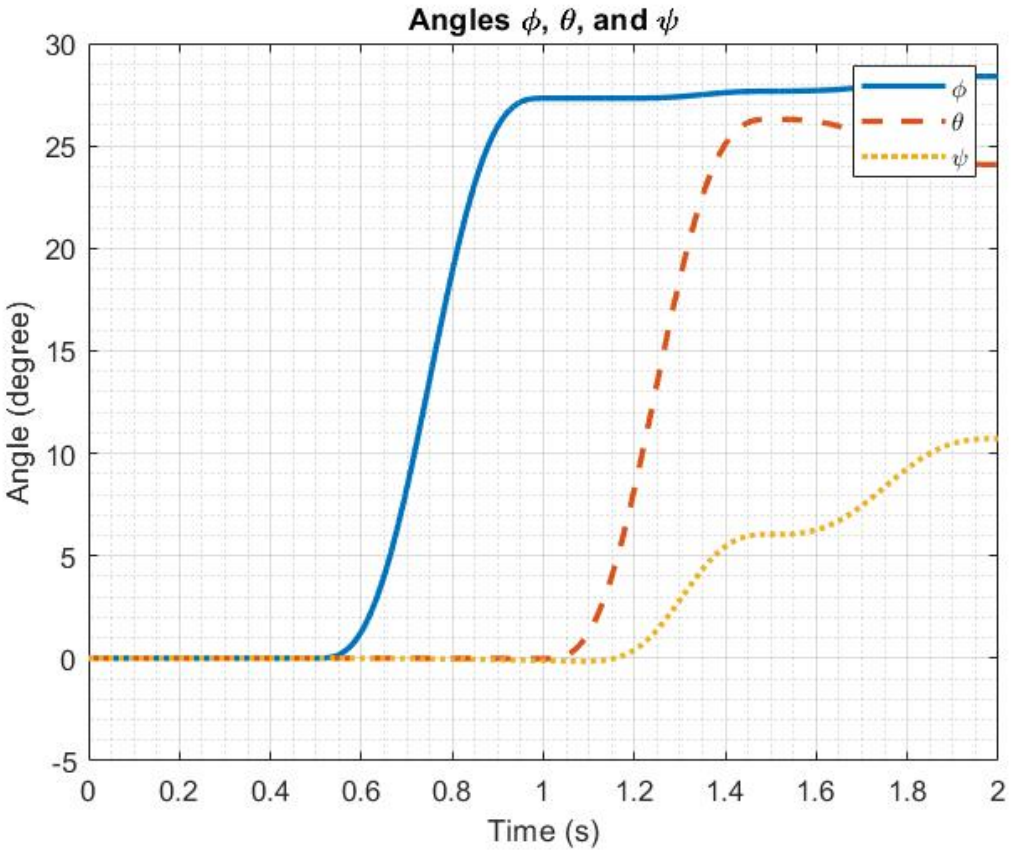


Figure 4.10: Simulated angular positions

The graphs obtained from simulation of our mathematical model closely resembles the graph given in reference paper[9]. This validates implementation of mathematical model.

4.9. Pixhawk Architecture

The PX4 architecture[17] is basically divided into two parts:**Middleware** and **Flight Stack**.

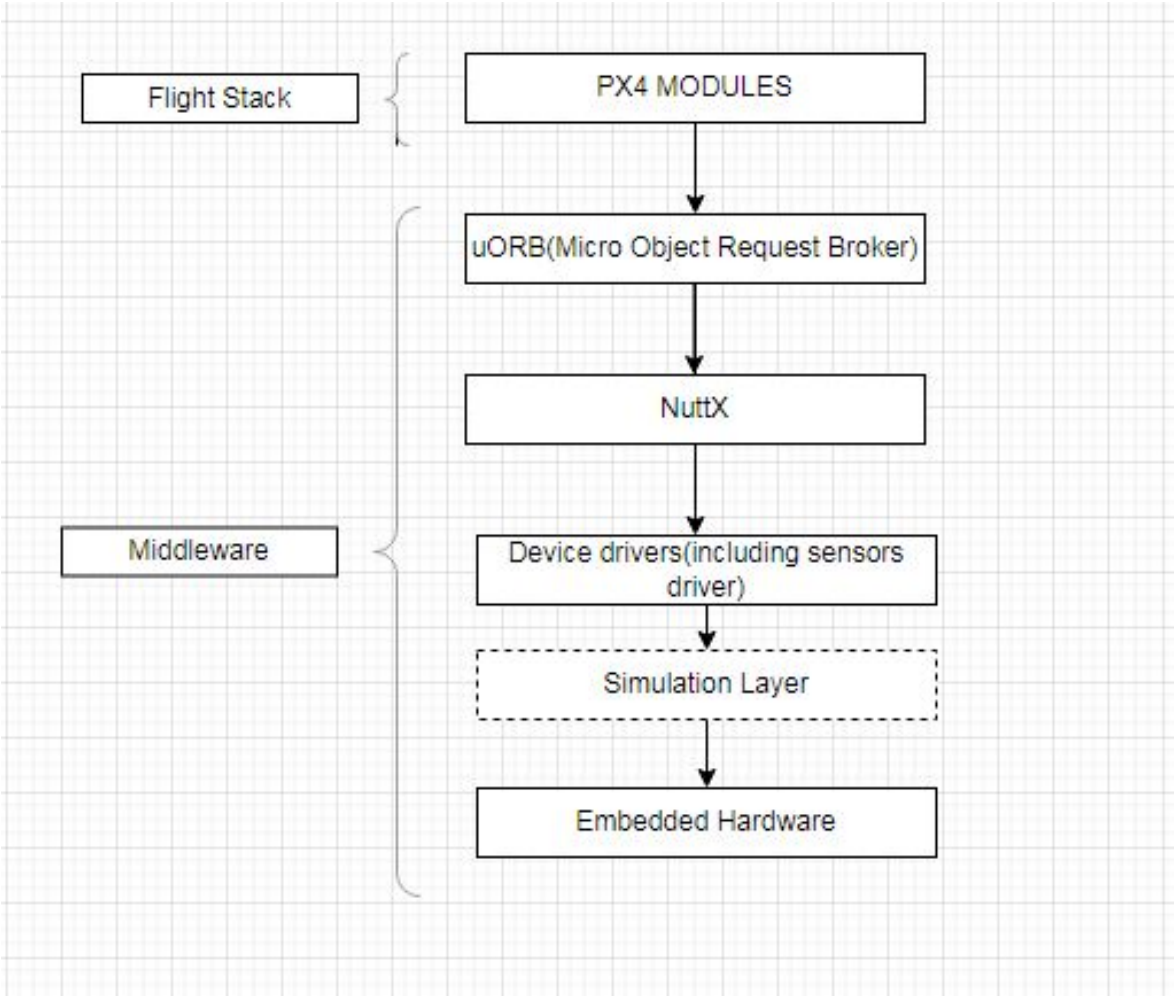


Figure 4.11: PX4 Architecture

The basic architecture of PX4 is shown in the fig.4.11. In the middleware, PX4 mainly includes embedded hardware(microcontroller integrated with sensors). Similarly, sensor drivers and actuator drivers are also an integral part of middleware.A simulation layer is included in middleware that allowed us to run SITL in a simulation environment.
NuttX is used as RTOS in PX4.It is a real-time embedded system and highly efficient in complex environments like that of PX4.uORB stands for micro-Object Request Broker and is a synchronous type Application Programming Interface. It works under the publish and subscribe data bus interface and acts as a data bridge between different modules as well as between modules and other middleware blocks.

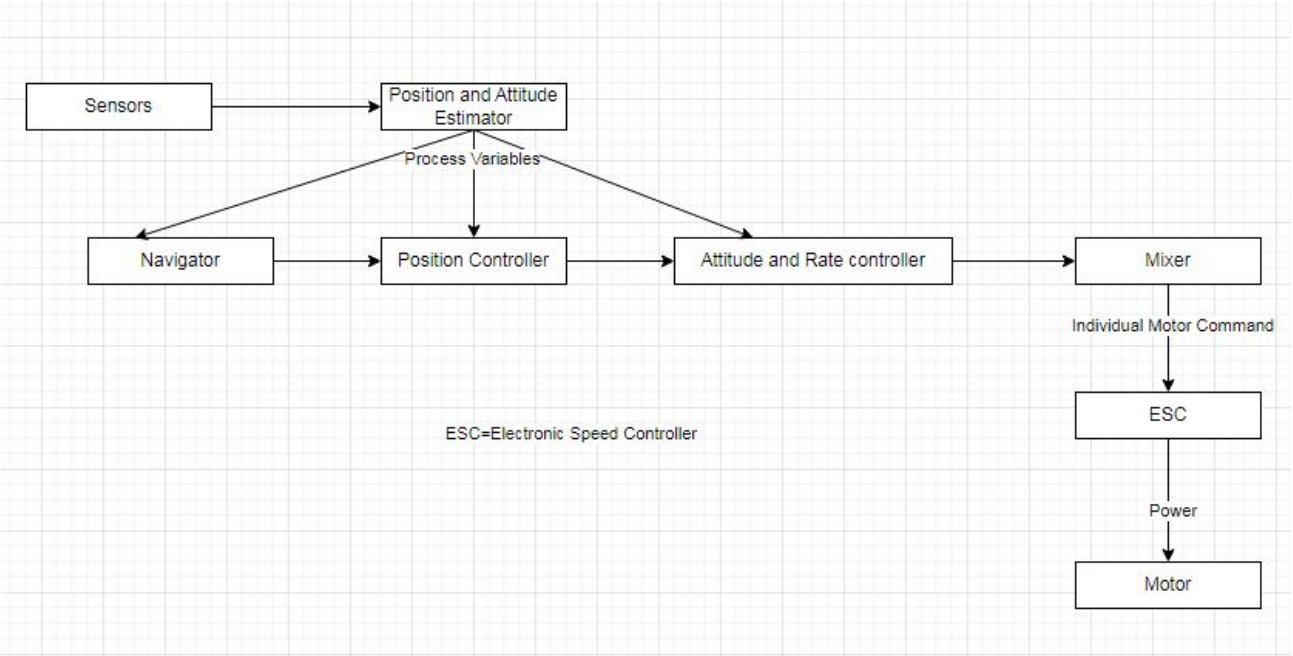


Figure 4.12: PX4 FLIGHT STACK

4.10. PX4 flight Stack

We have used our MPC implementation to be exactly synonymous with the PX4 flight stack.Therefore,it was necessary for us to understand the PX4 Flight Stack architecture.
The PX4 flight stack architecture can be summarised as in fig 4.12. The sensors of Pixhawk include barometric sensors, IMU sensors, GPS, etc which provide data to their respective estimators. The estimators directly in line with trajectory tracking are position and attitude estimators. In each estimator, the data fed through real-time embedding with sensors are filtered through EKF. The outputs of these estimators are called estimate states or process variables.
The process variables are passed in a cascaded fashion to the navigator, position controller, and then attitude controller. The mixer is an application in PX4 that is responsible for mixing action. The mixing action refers to the transformation of force commands into motor commands. Further, each motor command is passed to respective ESCs.The ESCs are responsible for controlling the angular velocity of motors thereby, controlling all four control inputs of the quadcopter.

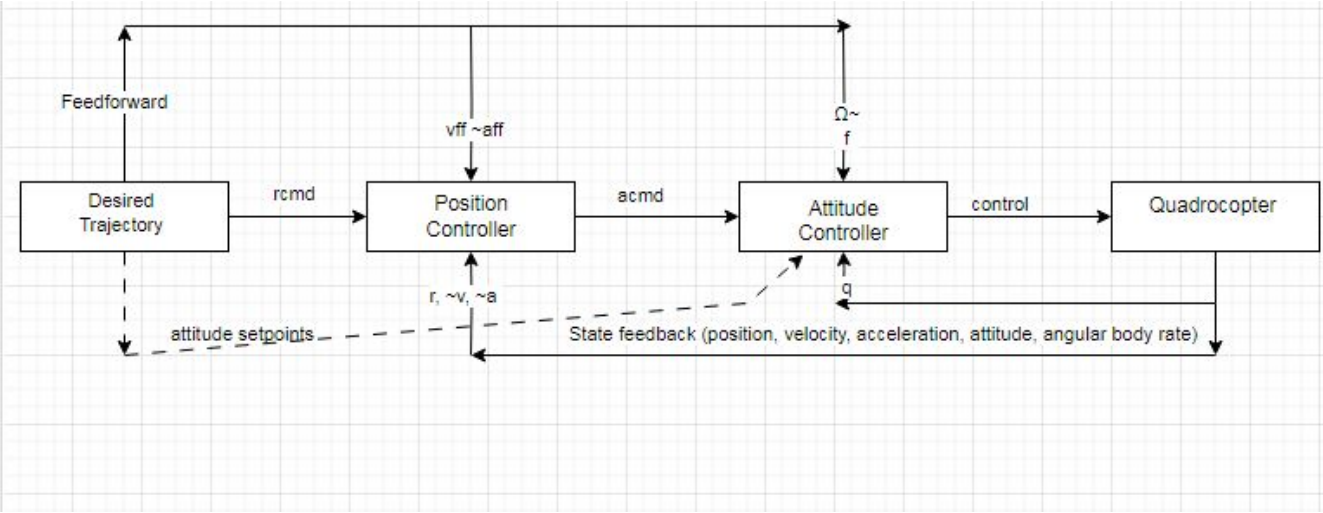


Figure 4.13: Cascaded control architecture

4.11. Problem Definition for Linear MPC

4.11.1. State matrices and vectors

A sample time of 0.2 seconds was used to obtain these equations, with the values of the moments of inertia substituted in state equations here, m in suffix denotes for model.

$$Am = \begin{bmatrix} 1 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Bm = \begin{bmatrix} 1.7395 & 0 & 0 \\ 1.7395 & 0 & 0 \\ 0 & 1.7395 & 0 \\ 0 & 1.7395 & 0 \\ 0 & 0 & 12.658 \\ 0 & 0 & 12.658 \end{bmatrix}$$

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

4.12. Constraints

The constraints on MPC are constraints of the quadcopter in terms of the maximum and minimum needed speed of the motor. The constraint on the speed of the motor reflects the constraint on torques .

$$\text{weight} = 1.158 \times 9.81 = 11.359\,\text{N}$$

Thrust of the quadcopter was calculated as follows:

$$\text{Thrust} = b\sum_{i=1}^{4}\omega_i^2$$

where, b - thrust coefficient, N/rpm^2, ω_i - speed for each motor, rpm.

The thrust coefficient is 2.35089×10^{-7} N/rpm^2. The speed for each quadcopter motor was calculated as follows:

$$11.359 = b\sum_{i=1}^{4}\omega_i^2$$

Solving for ω_i:

$$\omega_i = \sqrt{\frac{11.359}{4 \times 2.35089 \times 10^{-7}}} \approx 3475\,\text{rpm}$$

The input for the rotational motion of the quadcopter is torqued in the roll, pitch, and yaw axes which are listed below

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} d_b(\omega_4^2 - \omega_2^2) \\ d_b(\omega_1^2 - \omega_3^2) \\ k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2) \end{bmatrix}$$

where U_1, U_2, and U_3 are torques τ_ϕ, τ_θ, and τ_ψ respectively. ω_1, ω_2, ω_3, and ω_4 represent the angular velocities in the roll, pitch, and yaw axes.The maximum rotational speeds of mtor is taken as 6000 rpm.

$$\tau_{\phi_{\max}} = 0.235 \times 2.35089 \times 10^{-7}(6000^2 - 3475^2) = 1.3217$$

$$\tau_{\phi_{\max}} = 0.235 \times 2.35089 \times 10^{-7}(6000^2 - 3475^2) = 1.3217$$

$$\tau_{\psi_{\max}} = 4.83379 \times 10^{-8}(6000^2 + 6000^2 - 3475^2 - 3475^2) = 2.3129$$

These are expressed in vector form as follows:

$$U_{\max} = \begin{bmatrix} 1.3217 \\ 1.3217 \\ 2.3129 \end{bmatrix}$$

The minimum torques were obtained by replacing the motor speed values of 4 with 2, replacing 1 with 3 and replacing motor speed values 1 and 3 with 2 and 4 respectively in the matrix of U_1, U_2, and U_3 . This results in negated values of $[\ \mathrm{U}_{\max}].U_{\min} = \begin{bmatrix} -1.3217 \\ -1.3217 \\ -2.3129 \end{bmatrix}$

A rate of input change of 60 percent of the input was found to guarantee stable performance.[12] Therefore the maximum rate of input change is taken as:

$$\Delta U_{\max} = \begin{bmatrix} 0.793 \\ 0.793 \\ 1.388 \end{bmatrix}$$

The minimum rate of input change, U min are the negated values of the maximum rate of input change.

At a particular sample instant, k, the current rate of input change, u(ki) of each input variable was bounded by the maximum and minimum rates of input change as shown in the equations below,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta u_1(k) \\ \Delta u_2(k) \\ \Delta u_3(k) \\ \Delta u_1(k+1) \\ \Delta u_2(k+1) \\ \Delta u_3(k+1) \end{bmatrix} \leq \begin{bmatrix} 0.793 \\ 0.793 \\ 1.388 \\ 0.793 \\ 0.793 \\ 1.388 \\ -0.793 \\ -0.793 \\ -1.388 \\ -0.793 \\ -0.793 \\ -1.388 \end{bmatrix}$$

Expanding the above equations ,we have

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta u_1(k) \\ \Delta u_2(k) \\ \Delta u_3(k) \\ \Delta u_1(k+1) \\ \Delta u_2(k+1) \\ \Delta u_3(k+1) \end{bmatrix} \leq \begin{bmatrix} 1.3217 - u_1(k-1) \\ 1.3217 - u_3(k-1) \\ 2.3129 - u_3(k-1) \\ 1.3217 - u_1(k-1) \\ 1.3217 - u_2(k-1) \\ 2.3129 - u_3(k-1) \\ -1.3217 + u_1(k-1) \\ -1.3217 + u_2(k-1) \\ -2.3129 + u_3(k-1) \\ -1.3217 + u_1(k-1) \\ -1.3217 + u_2(k-1) \\ -2.3129 + u_3(k-1) \end{bmatrix}$$

or,

$$CC\Delta U \leq d$$

4.13. Cost Function(J)

The Cost Function(J) has been referenced from [12].

$$J = (Rs - Px(k))^T (Rs - Px(k)) - 2\Delta U^T H^T (Rs - Px(k)) + \Delta U^T (H^T H + W)\Delta U$$

where Output prediction matrix P and H ,and Input weight W. The quadratic programming problem to be minimized is:

$$J = \frac{1}{2}\Delta U^T E \Delta U + \Delta U^T F$$

subject to $CC\Delta U \leq d$, where $E = 2(H^T H + W)$ and $F = -2H^T (R_s - Px(k))$.

4.14. Hildreth's quadratic programming method

Hildreth's algorithm solves the QP problem by iteratively updating the decision variables using a set of linear equations derived from the KKT (Karush-Kuhn-Tucker) optimality conditions.The main steps of Hildreth's quadratic programming algorithm are as follows:

1. Initialize the decision variables and construct the linear equations based on the KKT conditions and Solve the linear equations to obtain the updated values of the decision variables.

2. Check for convergence. If the convergence criteria are not satisfied, go back to previous step ; otherwise, terminate the algorithm.

4.15. Experimental Setup

Since our project is the implementation of the mpc algorithm on the real quadcopter. so, we have assembled the full-functioning quadcopter from scratch. We had a fight test regarding the functionality of the quadcopter. Initially, we had a problem related to the thrust generated by the motor. Among the four motors, 2 were not working properly. And then replacing all those four motors with the new motors, the quadcopter is working well.

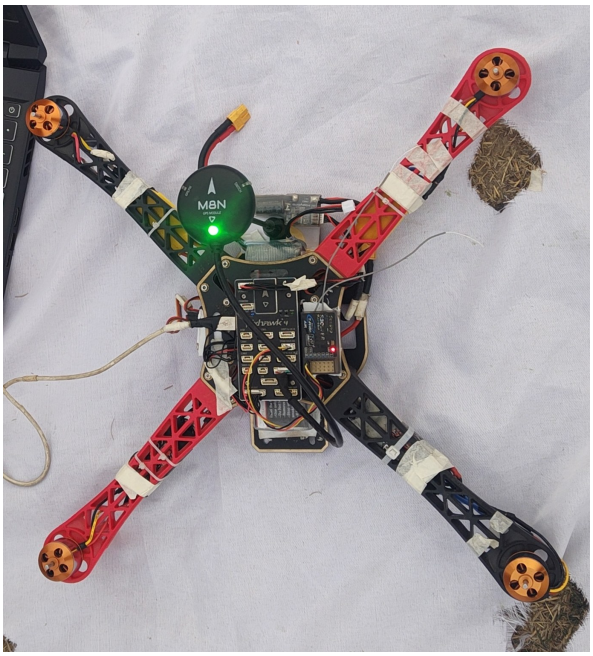


Figure 4.14: Assembled qudcopter

Parameter values for simulation of MPC based model:

Table 4.3: Parameter values for simulation

Parameter	**Value**	**Unit**
g	9.8	m/s^2
m	1.0	kg
l	0.225	m
thrust coeff(b)	2.90×10^{-6}	
drag coeff(d)	1.14×10^{-8}	
Ixx	4.0×10^{-3}	kg m^2
Iyy	4.0×10^{-3}	kg m^2
Izz	8.4×10^{-3}	kg m^2

CHAPTER 5: RESULTS AND DISCUSSION

5.1. Numerical Simulation of Non-linear MPC

Prior to hardware implementation, we conducted comprehensive numerical simulations to evaluate the trajectory tracking performance of the quadrotor under the non-linear Model Predictive Control (NMPC) framework. Our analysis revealed that the proposed system architecture for control, consisting of three distinct MPC controllers for altitude, attitude, and position control, exhibited significant computational cost intensity. Upon closer examination, it became evident that the computational demands of this architecture might not align optimally with hardware constraints. Therefore, a reassessment of our control strategy was necessary to ensure a more suitable approach for hardware implementation.

5.1.1. Trajectory tracking simulation

The methodology discussed was carried out for simulation in Python. The trajectory tracking performance of the quadrotor model was tested out in several trajectories and the results are shown below where the parameters for nonlinear optimization in each case are:

- **Simulation Time:** 10 seconds
- **Prediction Horizon (N):** 50 seconds
- **Interval Between Iterations (dt):** 0.02 seconds
- **Iteration Range:** i ranges from 0 to $\frac{\text{sim-time}}{\text{dt}} = 2500$
- **Yaw:** $\frac{2\pi i\, dt}{10}$ for all trajectories

5.1.2. Circular trajectory

Figure 5.1 shows the tracking performance for the circular reference trajectory that has been provided to the quadcopter. It takes around 3 seconds for the quadrotor to closely follow the reference circular path which is at a height of 3 meters from the ground. The figure illustrates

that the quadrotor is following the reference path satisfactorily after 4 seconds. In Figure 5.2 the simulated graph for linear position trajectory is illustrated. For around 4 seconds, the quadcopter is not following the desired trajectory but is converging towards it. The reason behind the initial discrepancy is that the quadcopter can not produce high enough velocity at the start since it realizes that it has to move directly to a height of 3 meters at the first instant which is clearly not possible which is also apparent from the Figures 5.4 and 5.5 with high linear and angular velocity demands initially. Figure 5.6 shows the corresponding values of thrust and torques across the three axes.

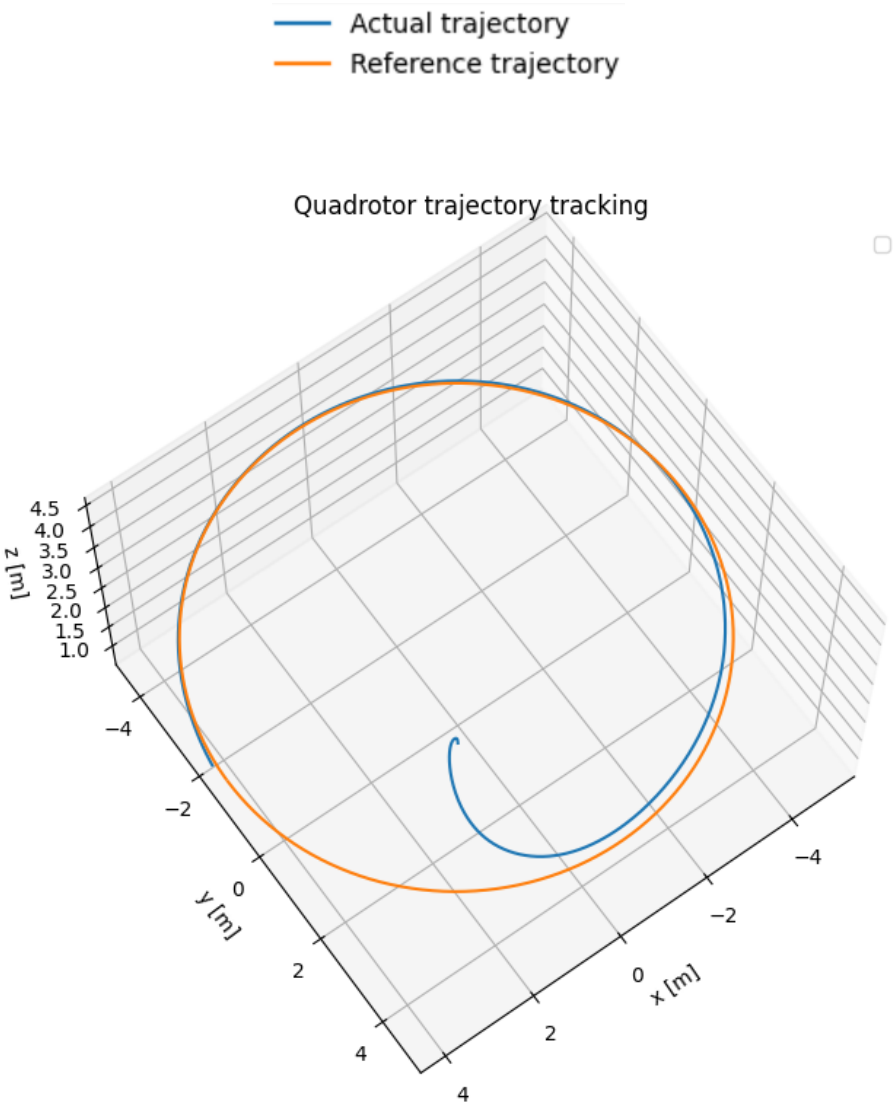


Figure 5.1: Circular Trajectory Followed by Quadcopter

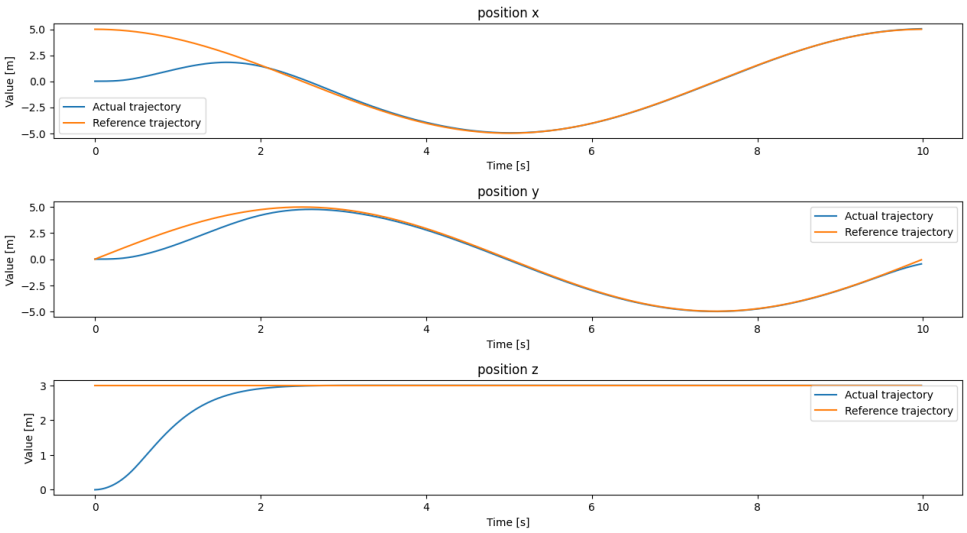


Figure 5.2: Position of Quadcopter

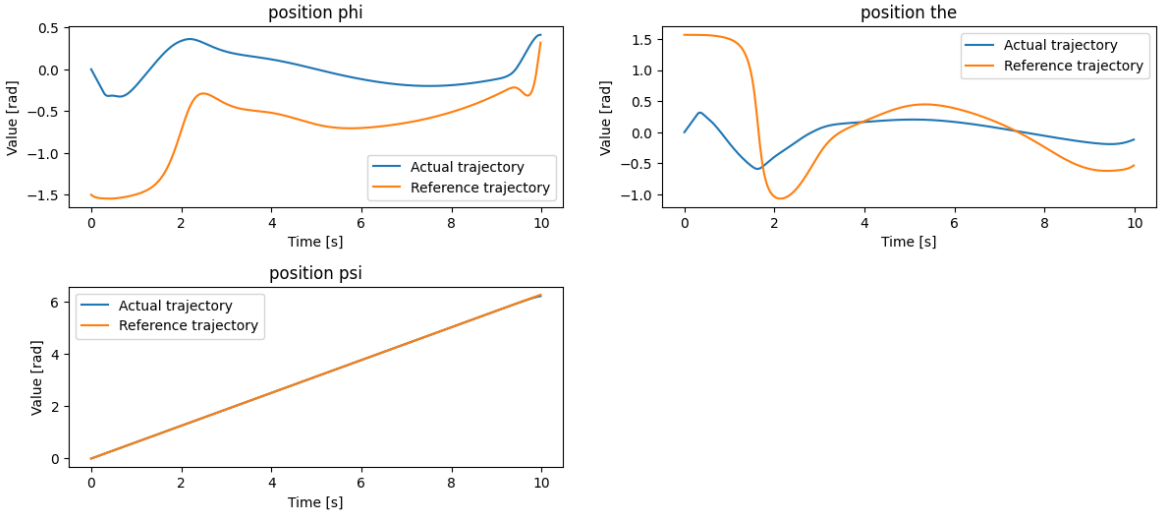


Figure 5.3: Attitude of Quadcopter

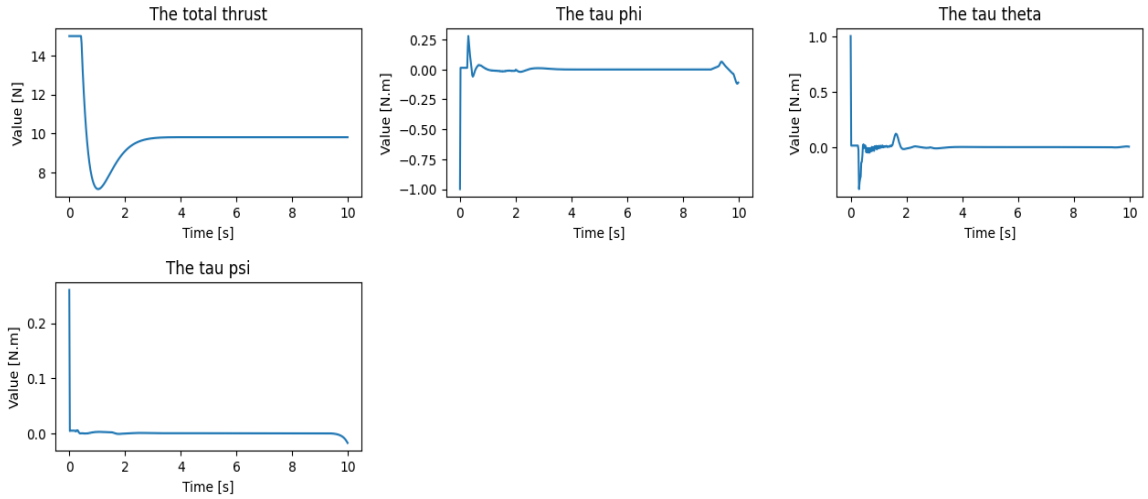


Figure 5.5: Total Thrust and Torques

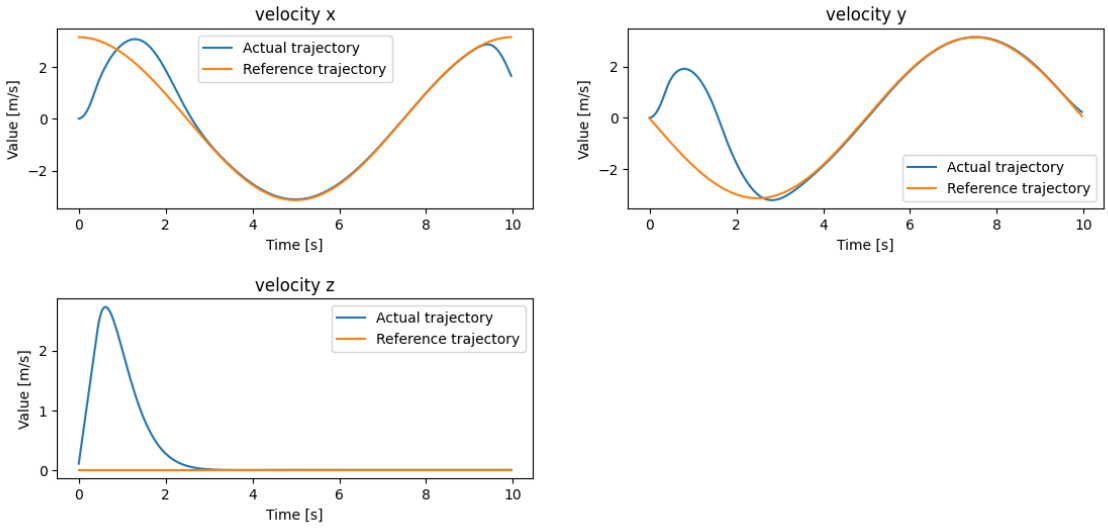


Figure 5.4: Linear velocities along the x,y,z axis

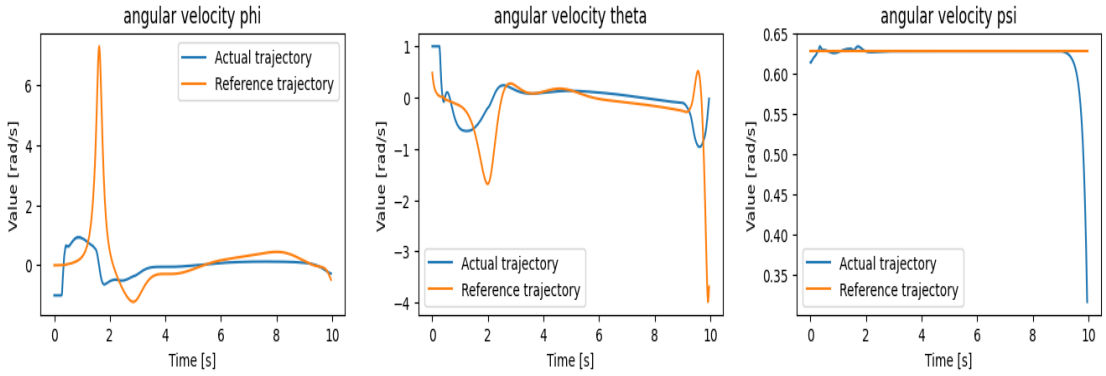
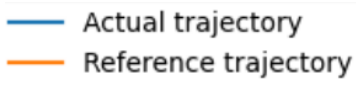


Figure 5.6: Angular velocities

5.1.3. Square shaped Trajectory

Figure 5.1 shows the tracking performance for the square-shaped reference trajectory that has been provided to the quadcopter. As can be seen in the figure the quadcopter is not able to take a sharp right-angle turn as required in the corners of the square path. Instead, the quadcopter has smoothly followed a trajectory allowed by its system dynamics limits. What is interesting is that if we observe the angular velocities plot we can see drastic requirements of changes in velocities as seen in Figure 5.11, meaning that there will be some angular acceleration around such ”bumps” which will generate some torques in the direction of the corresponding body frame angles which is seen in Figure 5.12



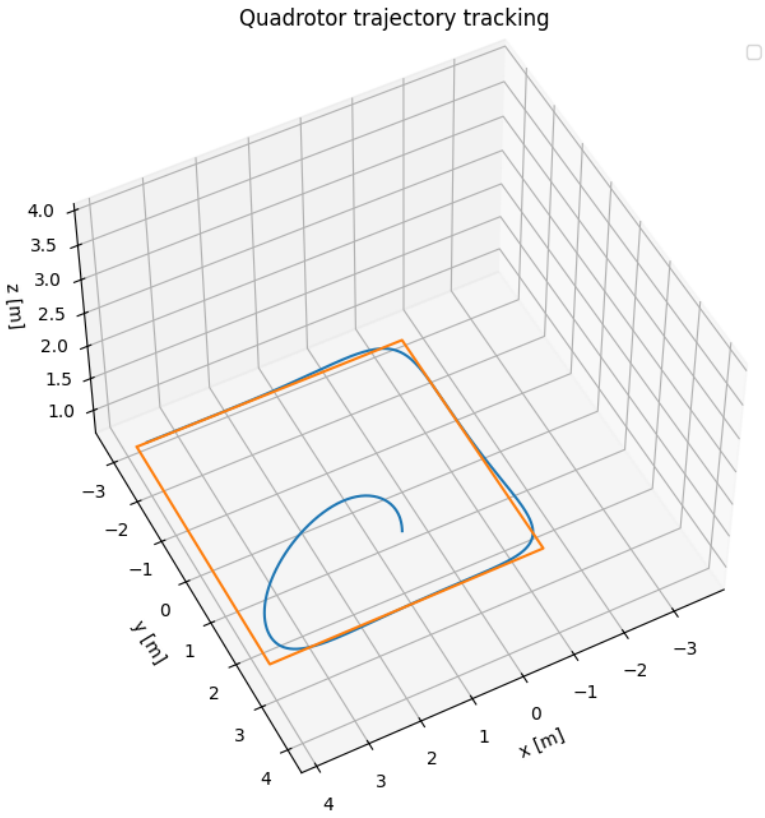


Figure 5.7: Square Reference path followed by quadcopter

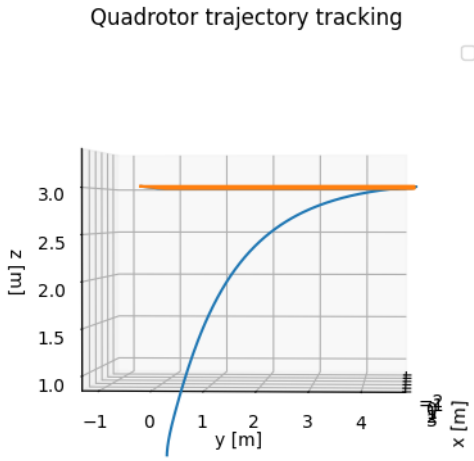


Figure 5.8: Square Reference path followed by quadcopter(side view)

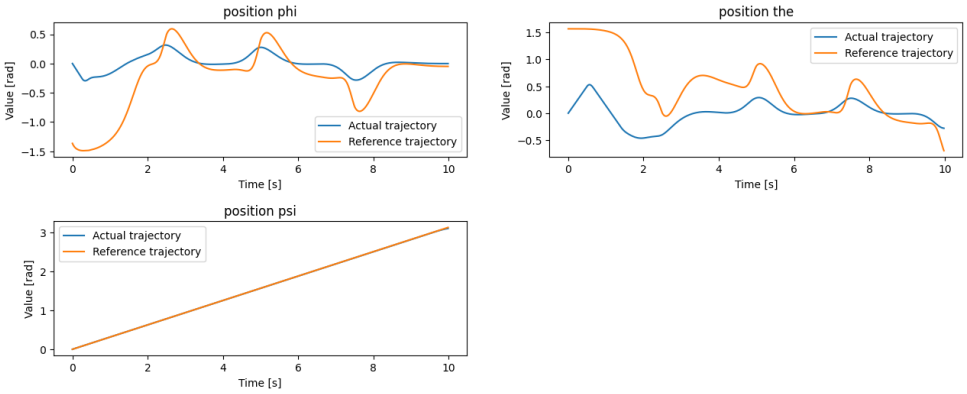


Figure 5.9: Attitude of quadcopter

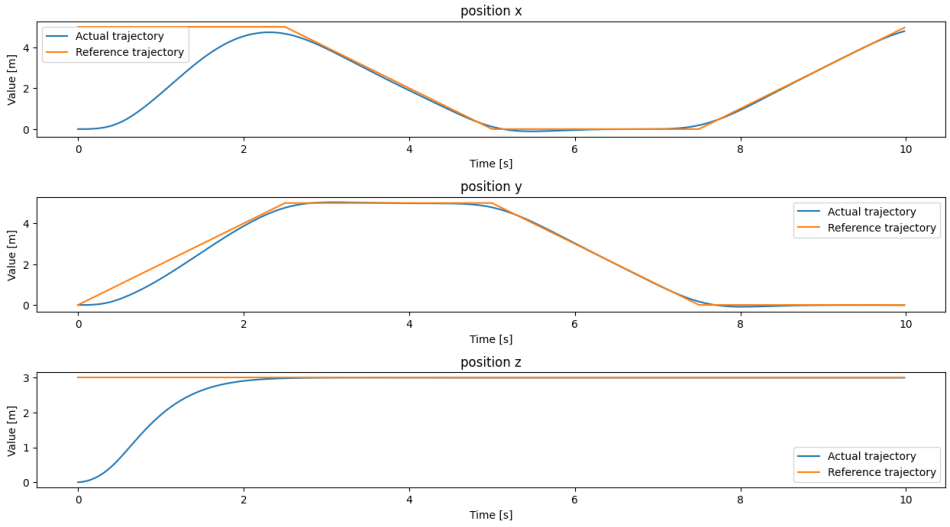


Figure 5.10: Position of quadcopter along x,y and z axis

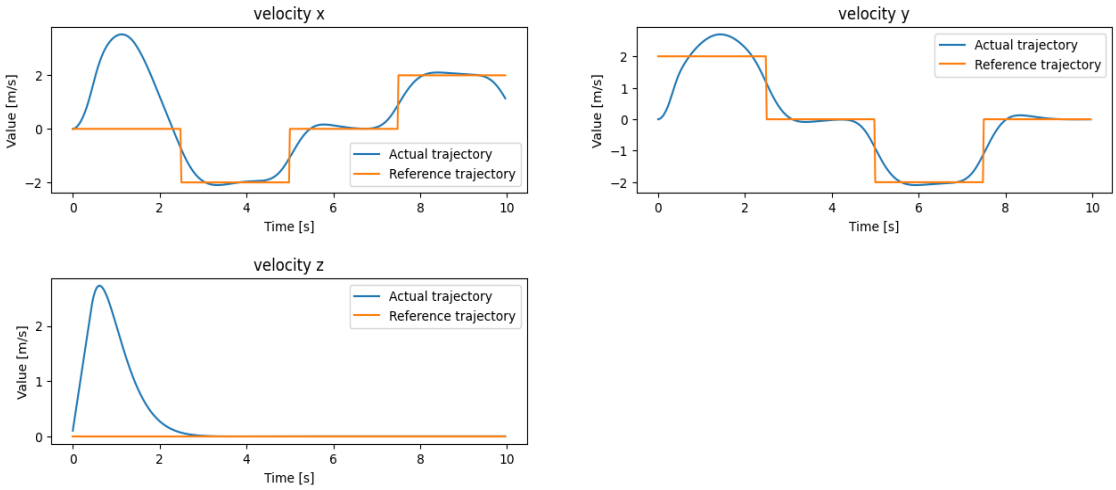


Figure 5.11: Linear velocities of quadcopter

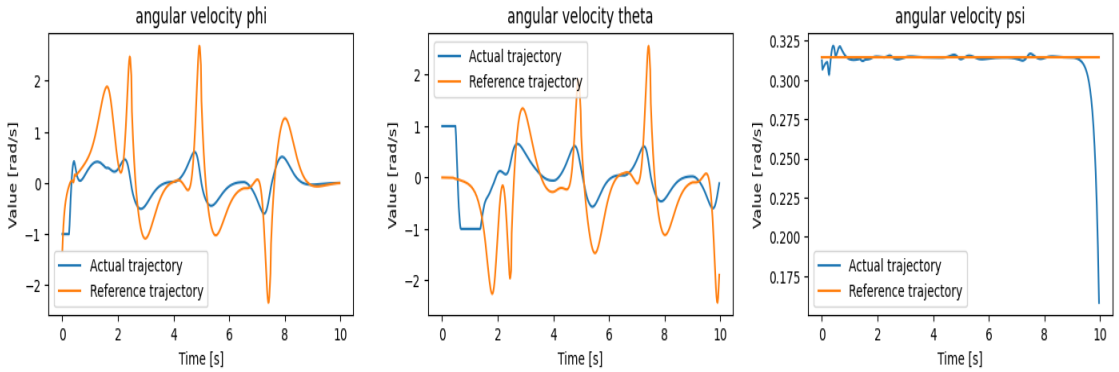


Figure 5.12: Angular velocities of quadcopter

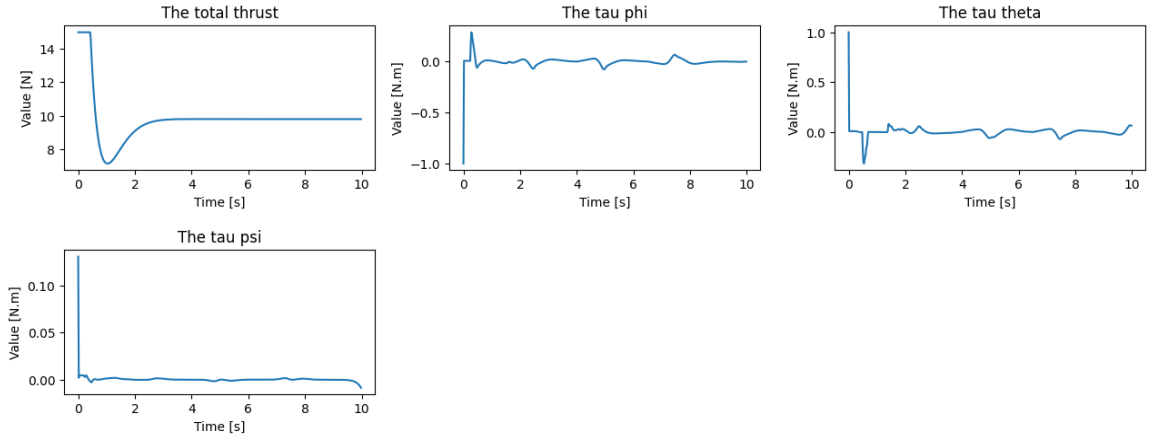


Figure 5.13: Total Thrust and torque on quadcopter

5.1.4. Linear Trajectory

A Linear reference trajectory has been provided to the quadcopter. Figure 5.1 shows the tracking performance for the same linear reference trajectory. We can see that the quadcopter follows the reference trajectory pretty closely after roughly 2 seconds. Since the reference has also been given starting from the origin (the origin position for the quadcopter), the convergence towards the reference path is faster than the previously mentioned square and circular trajectories.

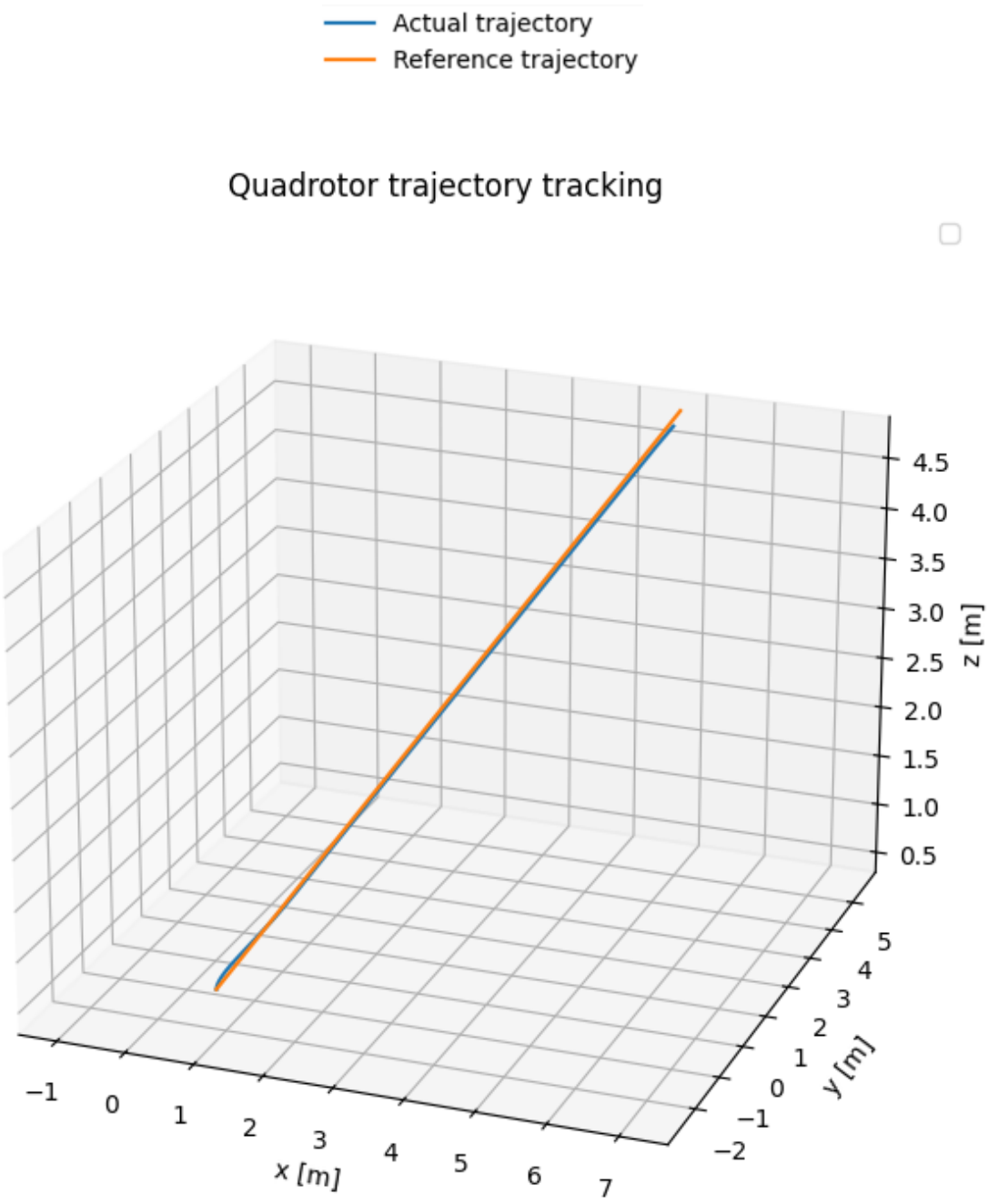


Figure 5.14: Linear Reference path followed by quadcopter

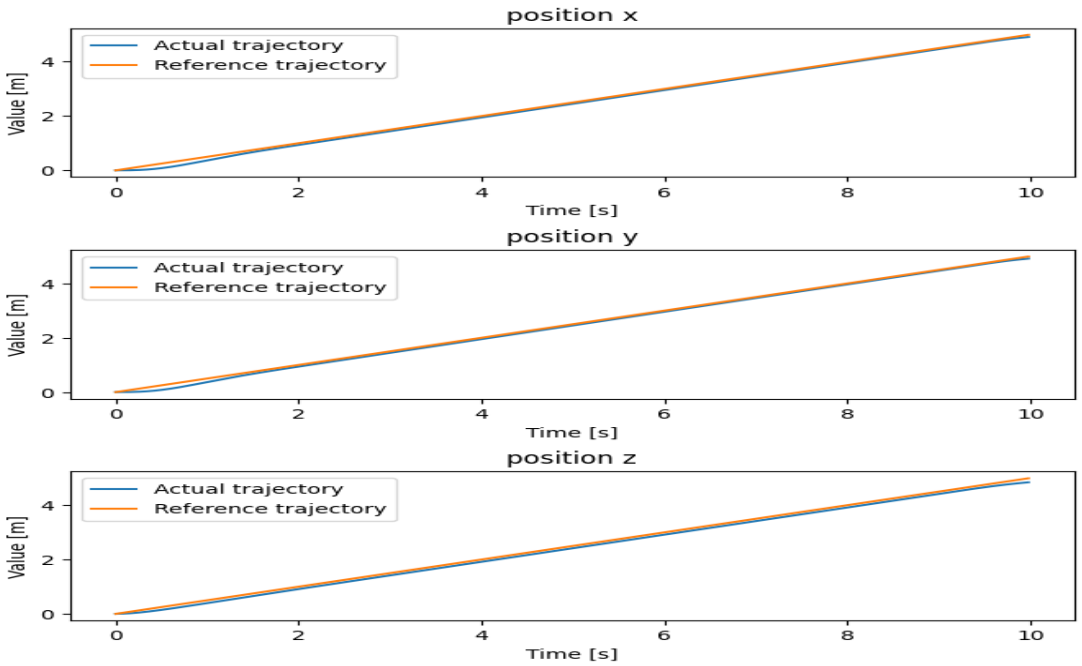


Figure 5.15: Positions of quadcopter

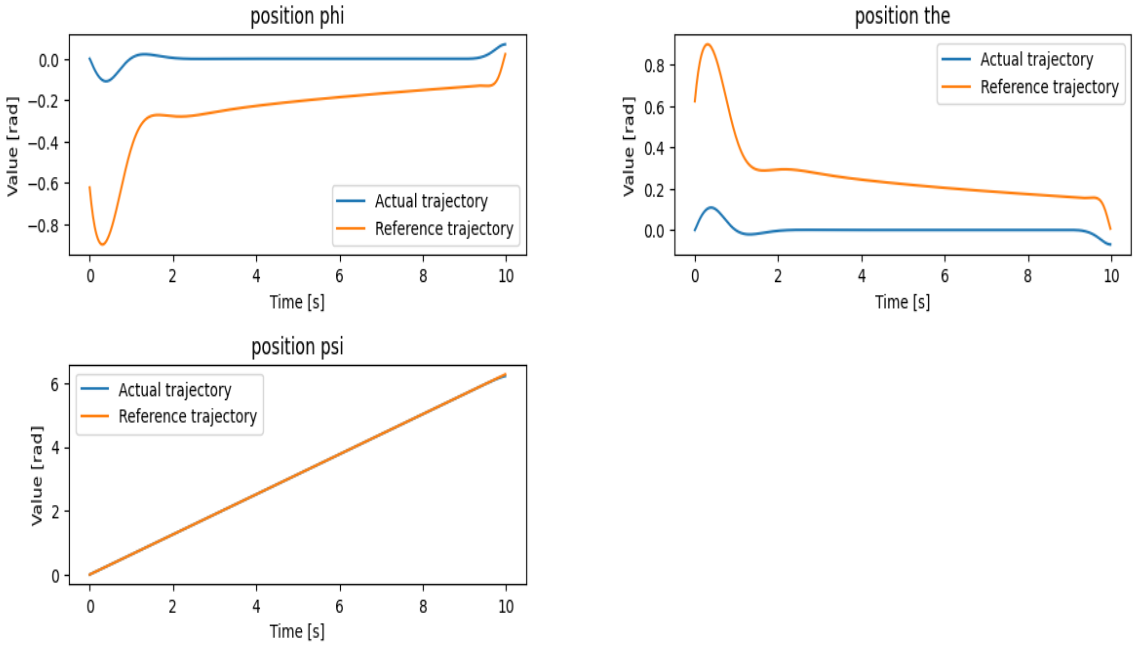


Figure 5.16: Attitude of quadcopter

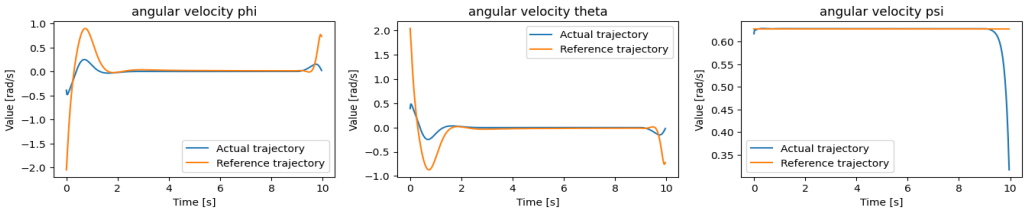


Figure 5.17: Angular velocities of quadcopter

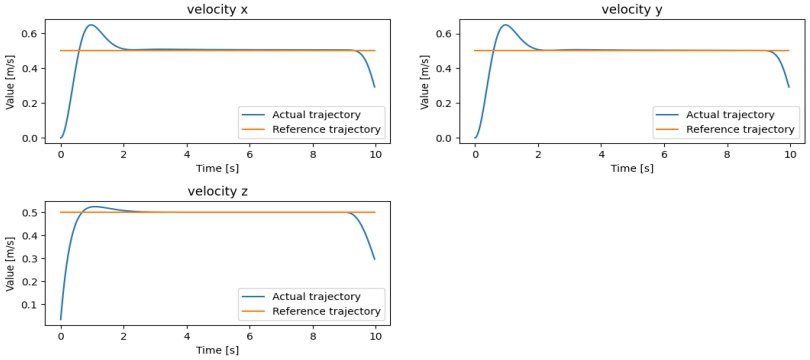


Figure 5.18: Linear velocities of quadcopter

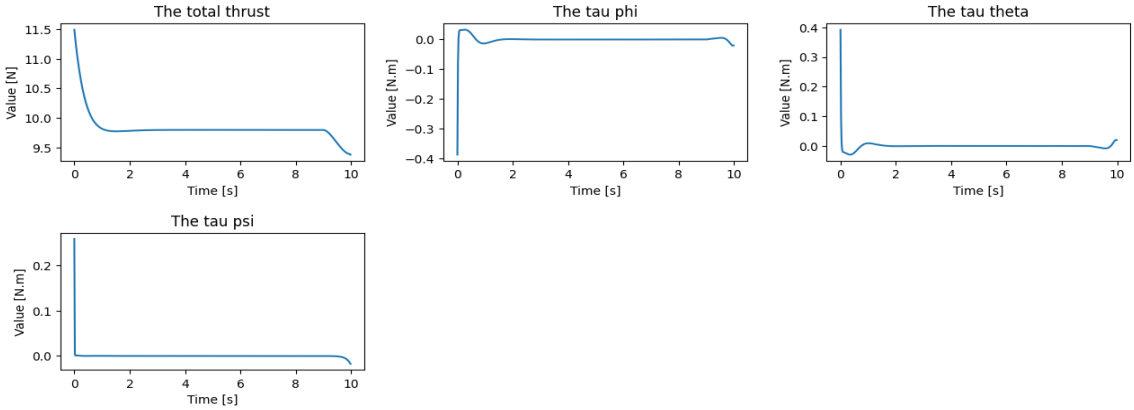


Figure 5.19: Total Thrust of quadcopter

5.1.5. Triangular Helix shaped Trajectory

A triangular helix reference trajectory has been provided to the quadcopter and the tracking performance of the quadcopter can be observed in Figure 5.19. Towards the corner where the quadcopter has to take an extremely sharp turn, we can see that the actual path is far from the reference. It is because such sharp turns are limited by our system dynamics which is carefully subjected as constraints that we have provided for the nonlinear optimization so that we do not get unreasonable values of velocity or acceleration requirements that in no way can be practically achieved in real life. The corresponding twelve states of linear and angular position and their derivatives are shown in Figure 5.20-5.23.

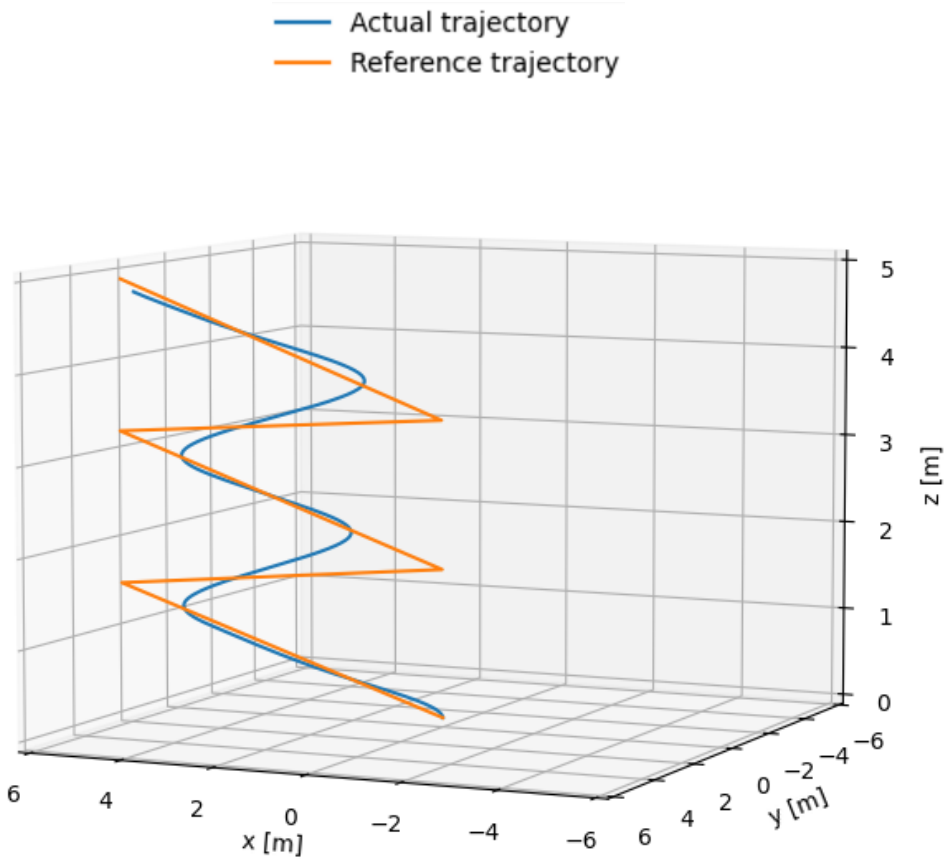


Figure 5.20: Triangular Helix Trajectory for quadcopter

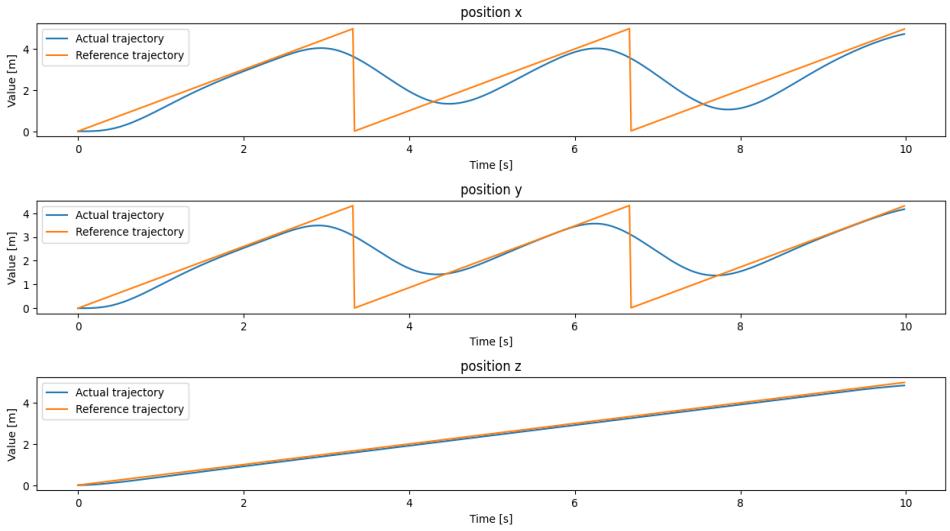


Figure 5.21: Position of quadcopter

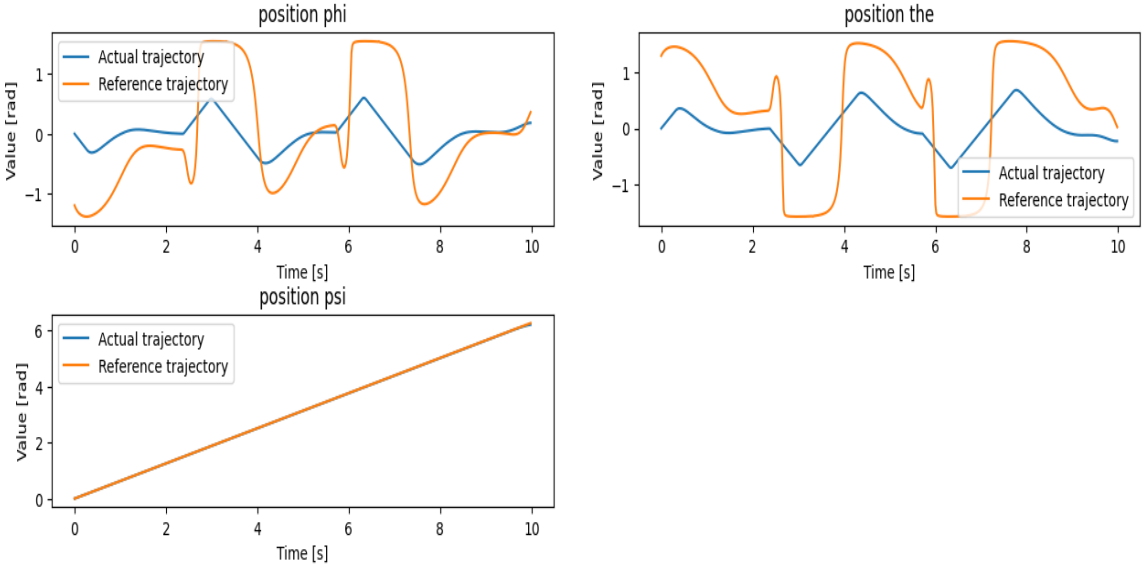


Figure 5.22: Attitude of quadcopter

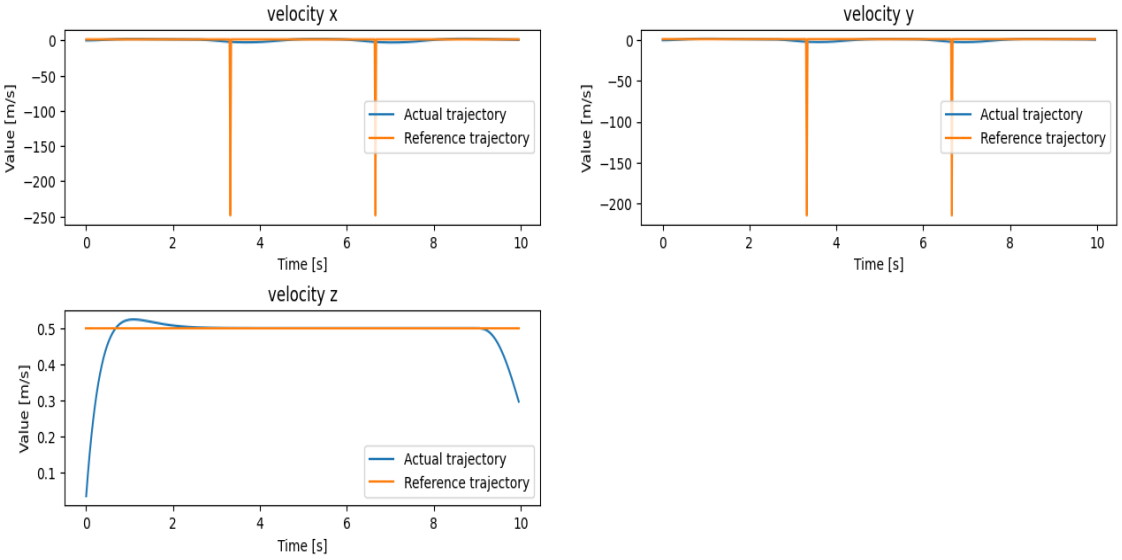


Figure 5.23: Linear velocities

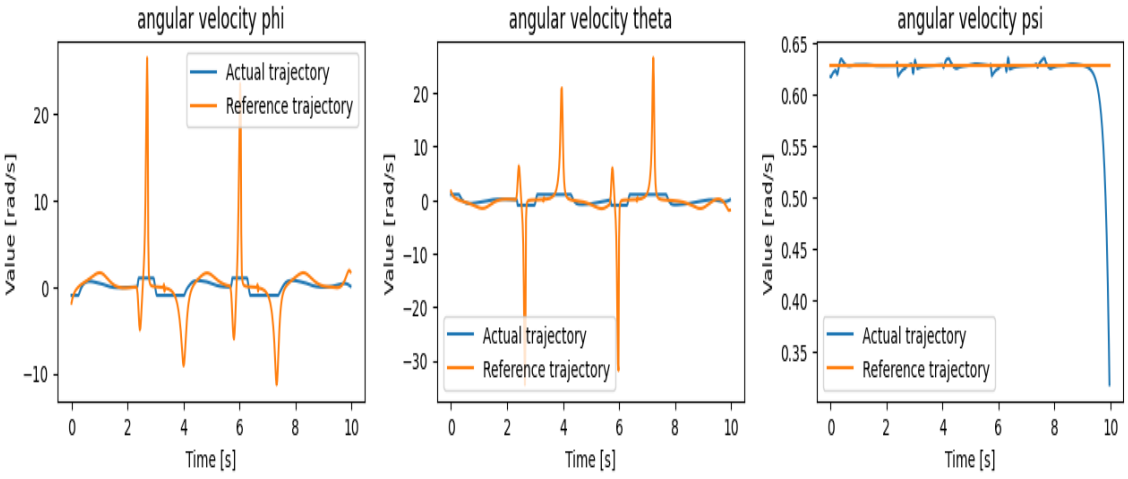


Figure 5.24: Angular velocities

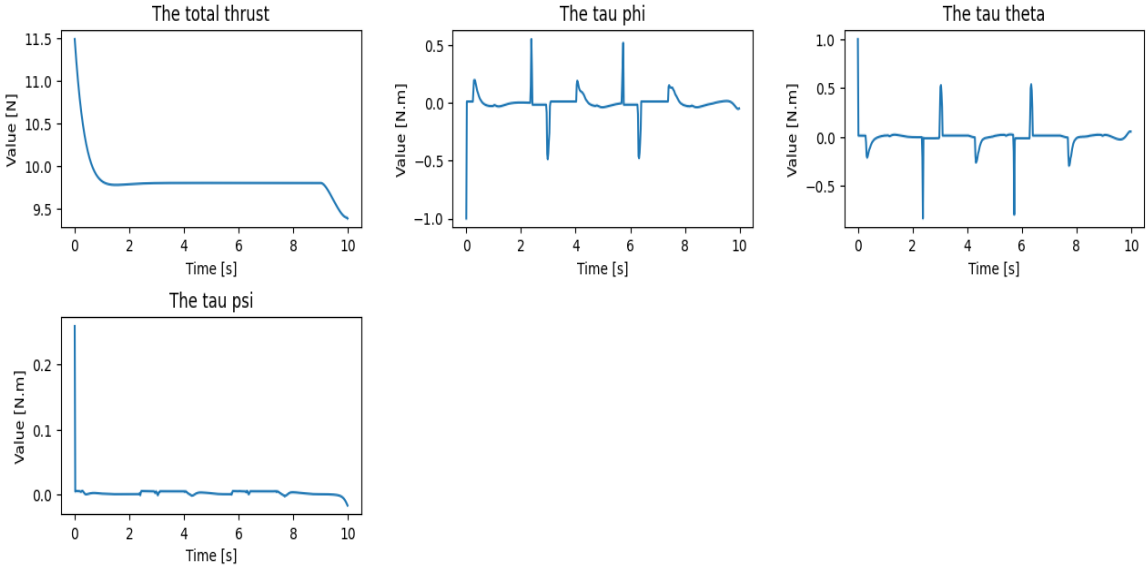


Figure 5.25: Total Thrust of quadcopter

5.1.6. Spiral Trajectory simulated for T = 100 seconds

A spiral trajectory was given as the reference trajectory for a quadcopter to track. The simulation was carried out for 100 seconds and the result is shown in Figure 5.25. This was done in order to verify that the numerical optimization can run for longer periods of time as well without any errors in simulation.

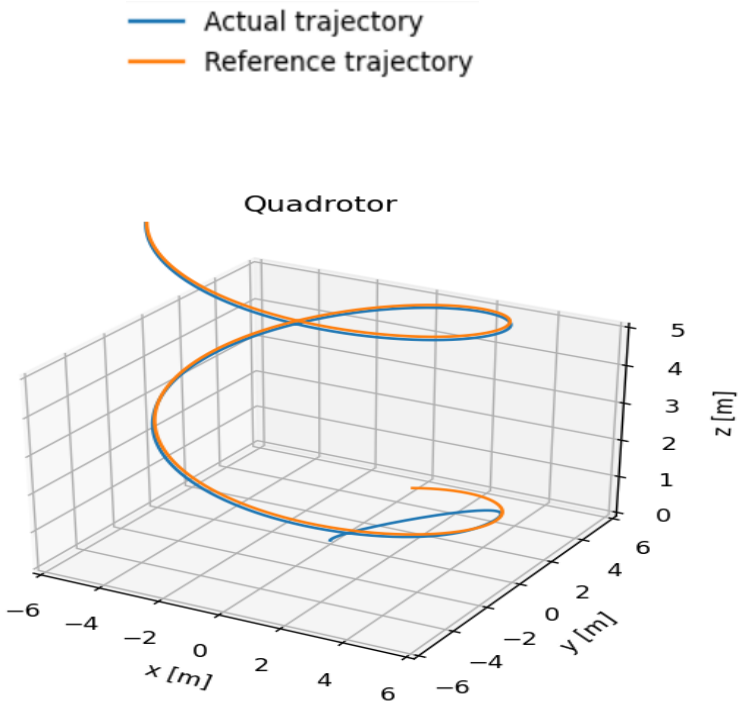


Figure 5.26: Helical Trajectory

5.2. SITL and Hardware implementation of Linear MPC

The integration of Model Predictive Control (MPC) into the Pixhawk flight controller involved several strategic decisions to optimize computational efficiency while ensuring effective control performance. To accomplish this, we transitioned from the computationally heavy non-linear MPC architecture to a more lightweight linear model tailored to meet the Pixhawk's specifications. This transition not only reduced computational costs but also ensured compatibility with the Pixhawk's hardware capabilities.

To seamlessly integrate MPC into the Pixhawk framework, we leveraged Software-in-the-loop (SITL) simulations in Gazebo. By directly embedding the linear MPC model within the attitude control module of the PX4 firmware, we could simulate and validate the control algorithm's performance in a virtual environment. This allowed us to fine-tune and optimize the MPC implementation before deployment onto the actual quadrotor.
Once the SITL simulations demonstrated satisfactory results, we proceeded to build and execute the modified PX4 firmware on the physical quadrotor for flight testing. This real-world validation phase confirmed the efficacy and reliability of the linear MPC implementation in controlling the quadrotor's attitude.

The prediction horizon was taken as five seconds and control horizon was taken as two seconds. Similarly, sampling interval of 0.2 seconds was considered for both SITL and hardware implementation of Linear MPC.

5.2.1. Software in the Loop (SITL)

Software-in-the-Loop (SITL) simulation of PX4 firmware within Gazebo presents a pivotal aspect of UAV (Unmanned Aerial Vehicle) development and testing. SITL offers a virtual environment where the PX4 firmware can be executed and analyzed without the need for physical hardware. Gazebo, with its advanced physics engine and realistic modeling capabilities, serves as the ideal platform for simulating UAV dynamics and interactions with the environment. By integrating PX4 with Gazebo, developers can accurately replicate real-world scenarios, including varying weather conditions, terrain features, and sensor inputs.

The build system makes it very easy to build and start PX4 on SITL, launch a simulator, and connect them. The syntax for build in SITL looks like this:

```
1 make px4_sitl
```

Similarly, the following code was run in Ubuntu Terminal for running Gazebo simulation:

```
make px4_sitl_default gazebo
```

Figure 5.27: Gazebo simulation

5.2.2. SITL Test MPC Based PX4 Firmware

The SITL simulation results were obtained by flying the simulated quadcopter— running the modified PX4 firmware — in Gazebo through flight mission waypoints created in QGroundControl.The SITL flight mission a was executed with a control horizon of two and a prediction horizon of five.

A circular trajectory, as shown in figure 5.28 , was given and simulated in Gazebo.

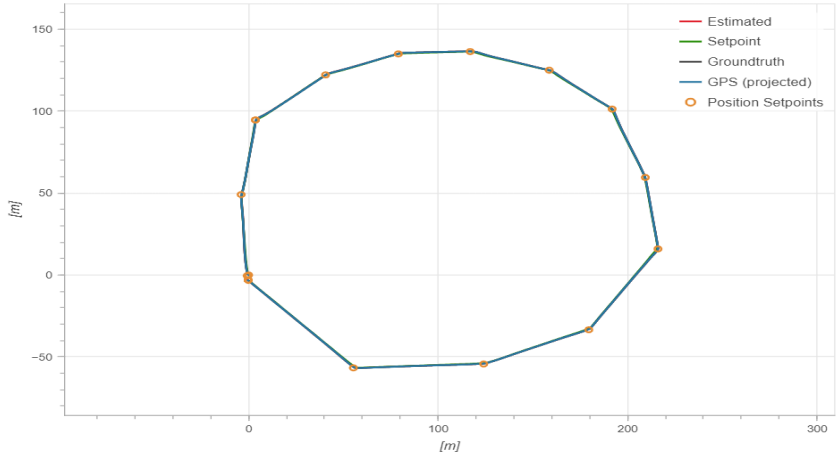


Figure 5.28: Circular Trajectory

Attitude Results

The plots for roll ,pitch and yaw angles along with respective rates for this trajectory are

given below :

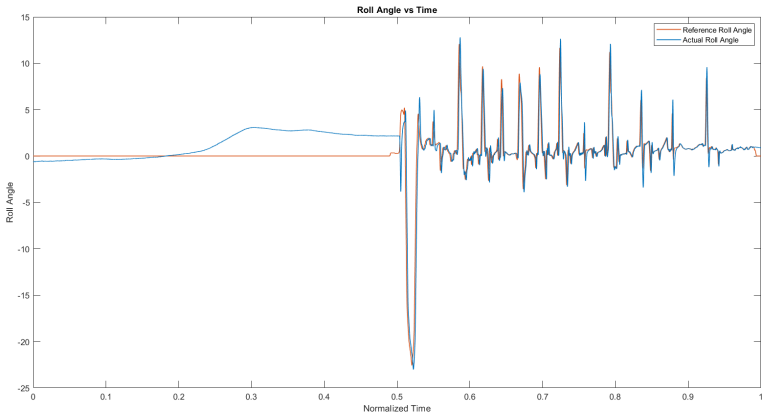


Figure 5.29: Roll Angle

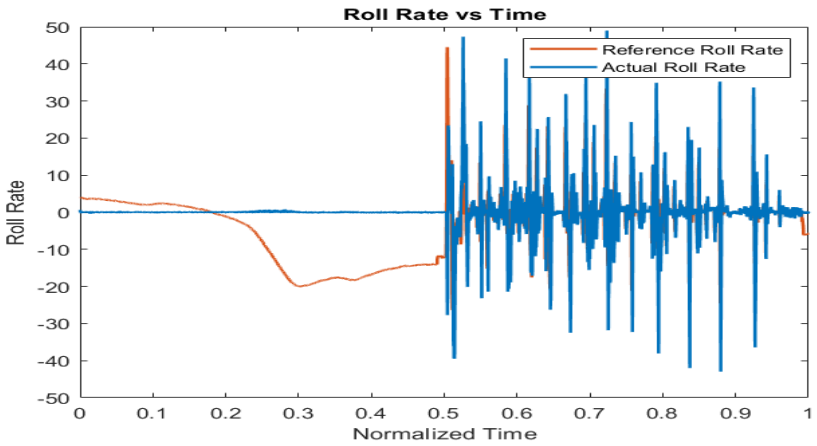


Figure 5.30: Roll Angular Rate

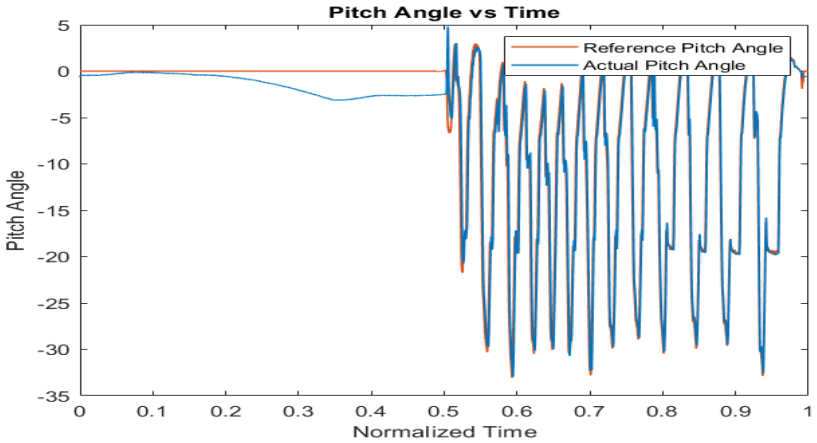


Figure 5.31: Pitch angle

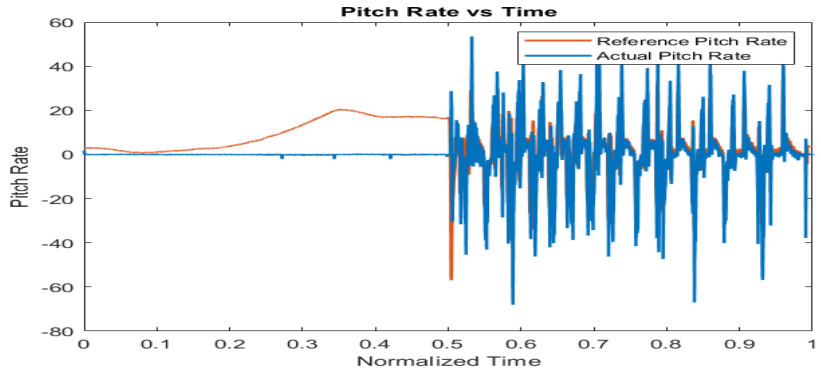


Figure 5.32: Pitch Angular Rate

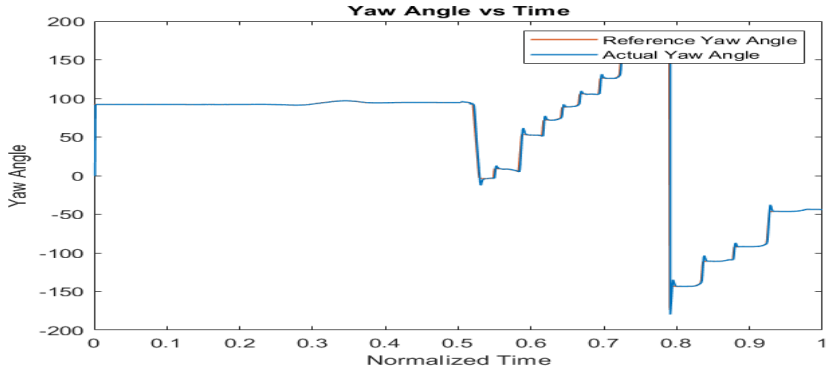


Figure 5.33: Yaw Angle

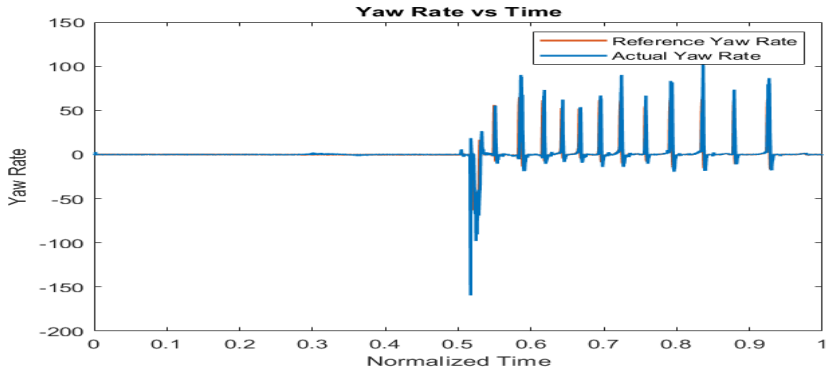


Figure 5.34: Yaw Rate

Before integration of moving horizon control known as MPC, SITL test was conducted. As shown by the results above a successful run of the Gazebo simulation for the MPC implemented in the PX4 firmware ensured that we could proceed to the actual implementation of the contol in our assembled quadrotor.

5.2.3. Building PX4 Firmware

After completion of SITL simulation of linear MPC enforced for the attitude control . Firstly, we needed to built the PX4 firmware with the MPC code to get a .px4 extension file which can only be obtained after a successful build. So, the Pixhawk was connected to the computer via a USB cable. A terminal window was opened and directory containing pixhawk code was navigated by commanding following code in Ubuntu terminal:

```
cd PX4-Autopilot
```

Then, the code was built using following command in same terminal and directory.

```
make px4_fmu-v5_default
```

The Code was uploaded in Pixhawk through QGroundControl.
Various problems were encountered during the building process. One of the main problems was Stack Overflow due to the limited storage of Pixhawk. Hence, space usage was increased from CMakeLists. Further, other problems(errors in Ubuntu Terminal) were treated by changing other various parameters within attitude control modules.
After completion of the build, a successful run ended with the output:

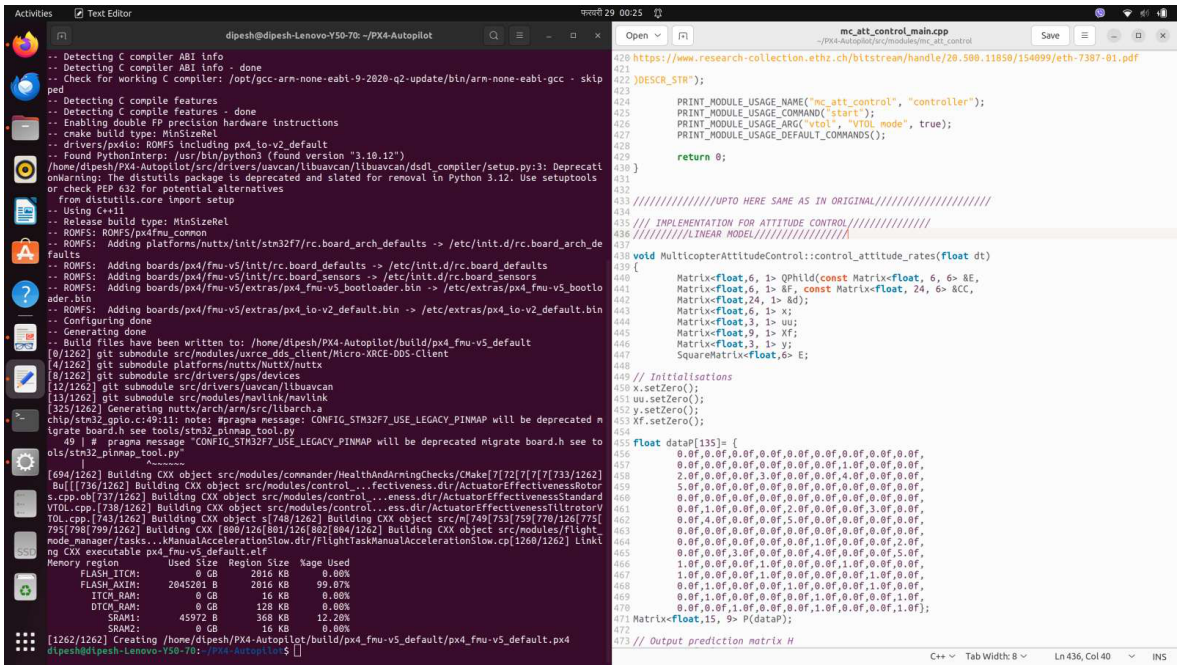


Figure 5.35: Build window for the PX4 file

5.2.4. Flight Tests

The Flight test for trajectory was performed with MPC implemented on the PX4 Autopilot firmware whereby the attitude of the quadrotor is controlled. The performance for the model predictive control for attitude control was implemented for different trajectories. The results are shown and discussed below.

Model Predictive Control based Trajectory tracking
In this section, the actual flight performance for the MPC implementation for different trajectories are discussed. The estimated entities (shown in blue color in plots) are actual variables of the quadcopter during the mission. Similarly, the setpoint entities(shown in red color in plots) are calculated variables of the quadcopter during a mission.

1. **Mission Path 1: W shaped trajectory**
   The mission path here replicated the W shaped trajectory.

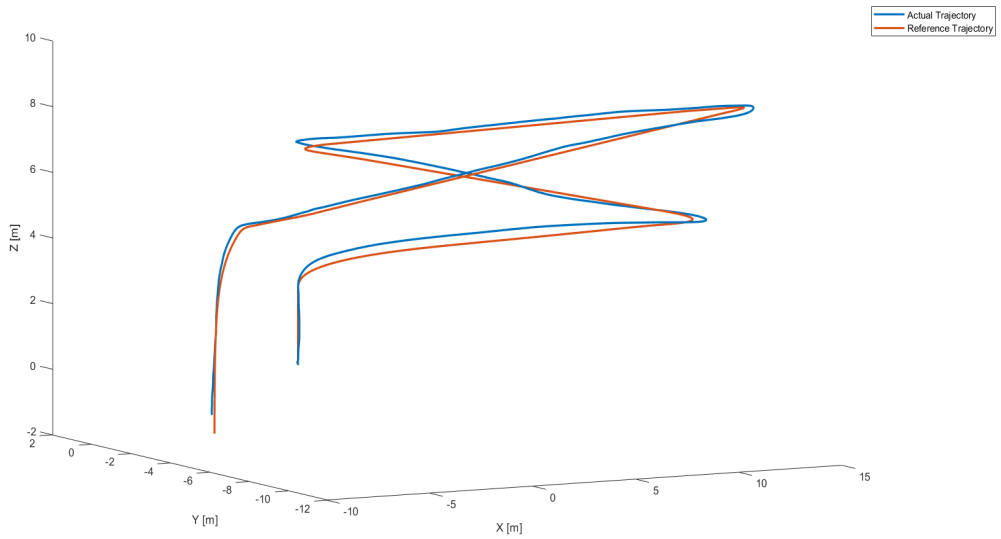


Figure 5.36: Tracking performance for Mission path 1

The plots for roll, pitch and yaw angles along with respective rates for this trajectory are given below :

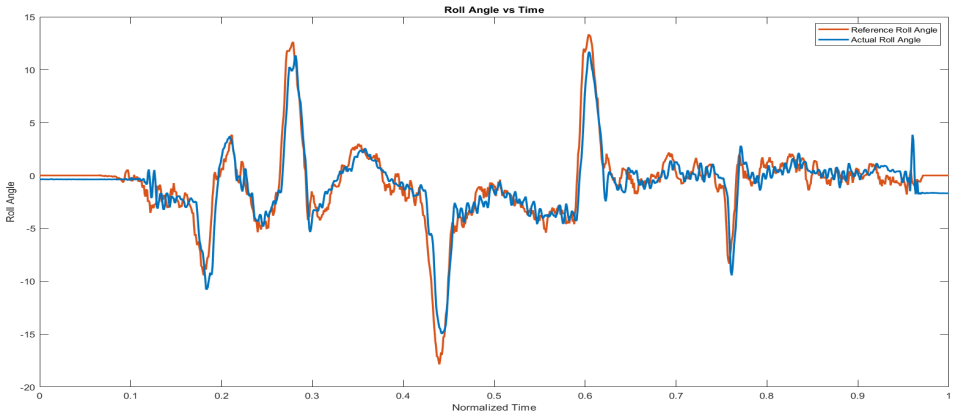


Figure 5.37: Roll Angle

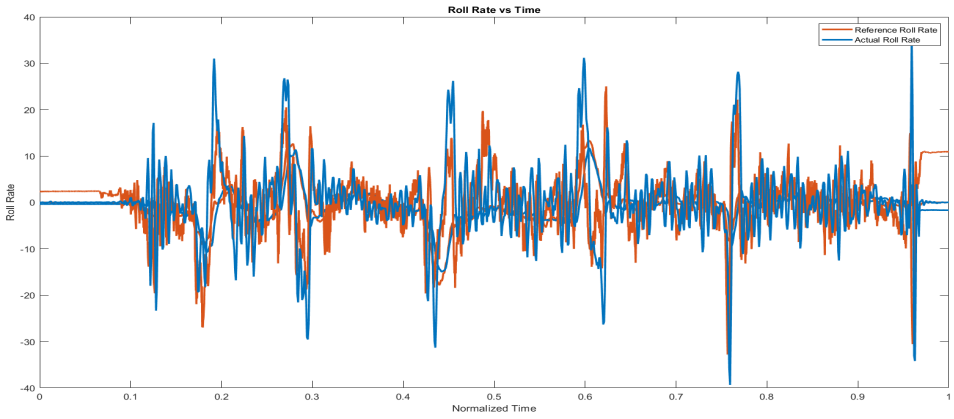


Figure 5.38: Roll Angular Rate

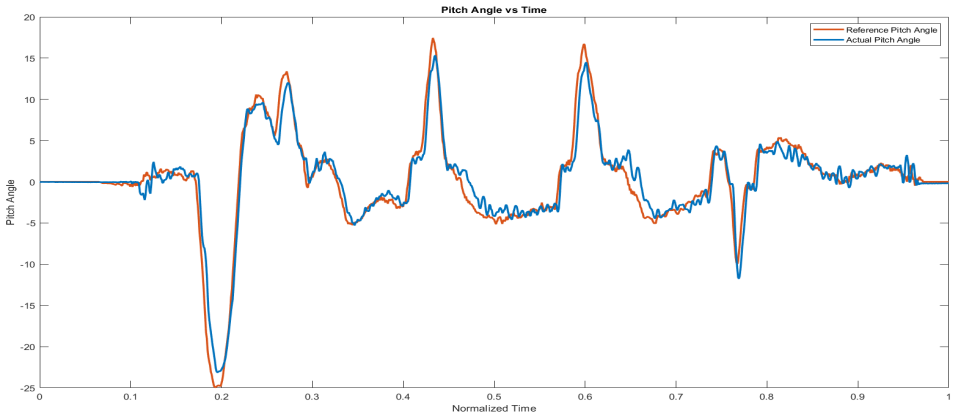


Figure 5.39: Pitch Angle

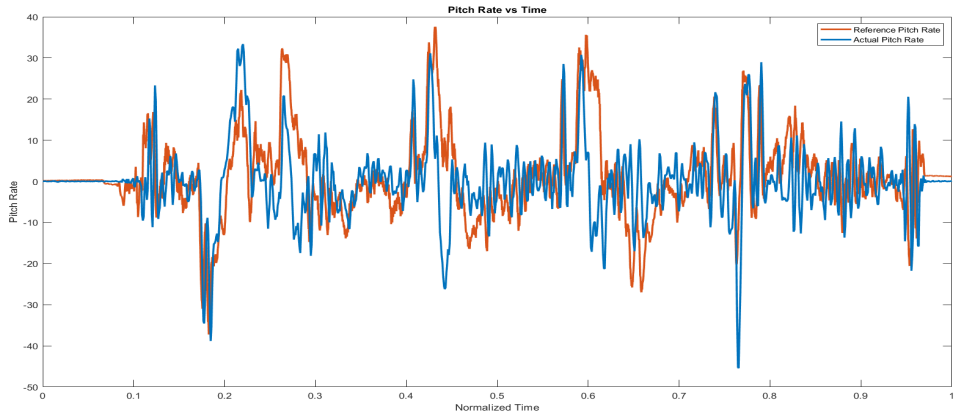


Figure 5.40: Pitch Angular Rate

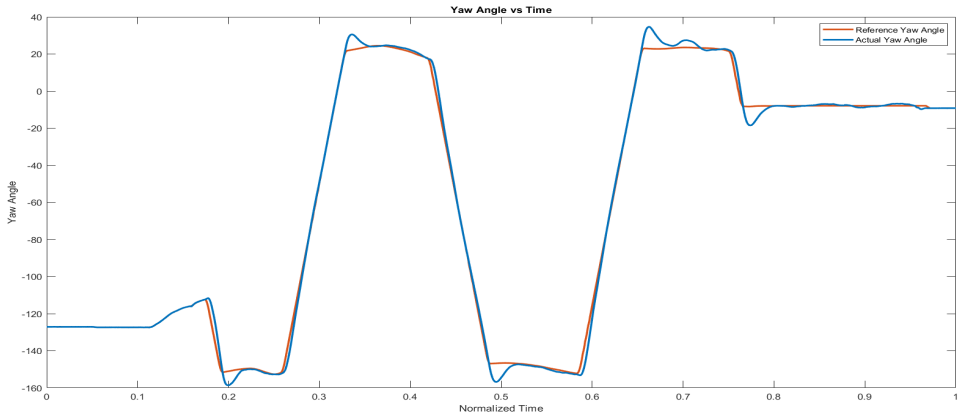


Figure 5.41: Yaw Angle

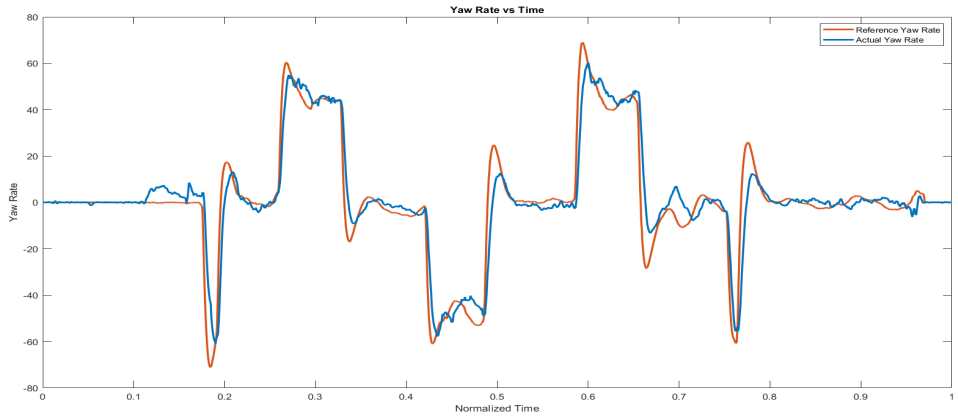


Figure 5.42: Yaw Angular Rate

2. **Mission Path 2:**

   This mission path resembled an approximate replica of Nepal's flag. This type of path was choosen to observe the performance of the attitude controller in sharp turns.

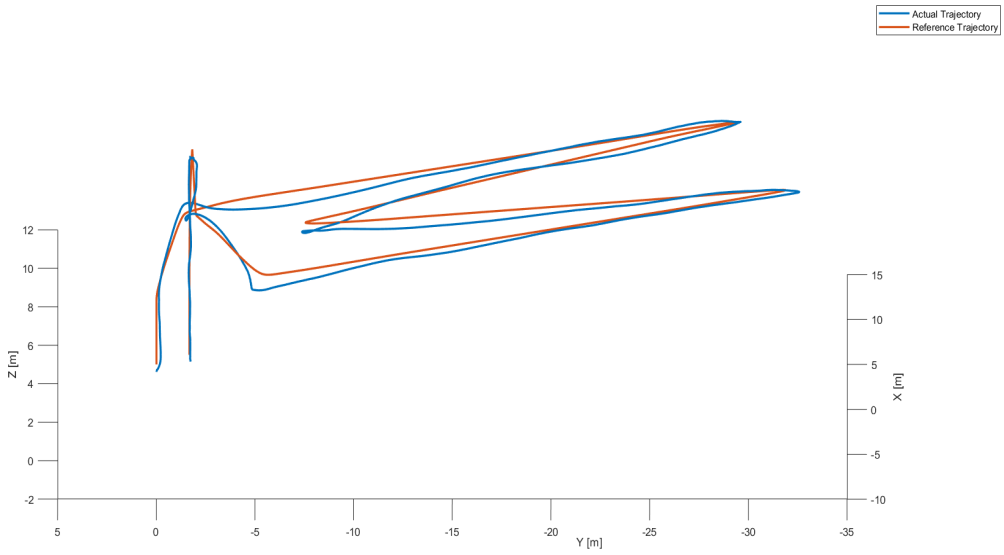


Figure 5.43: Mission path

The plots for roll ,pitch and yaw angles along with respective rates for this trajectory are given below :

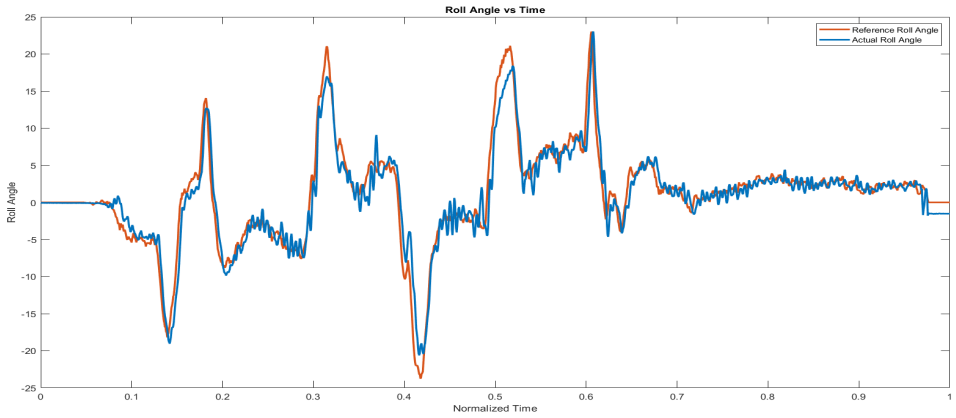


Figure 5.44: Roll Angle

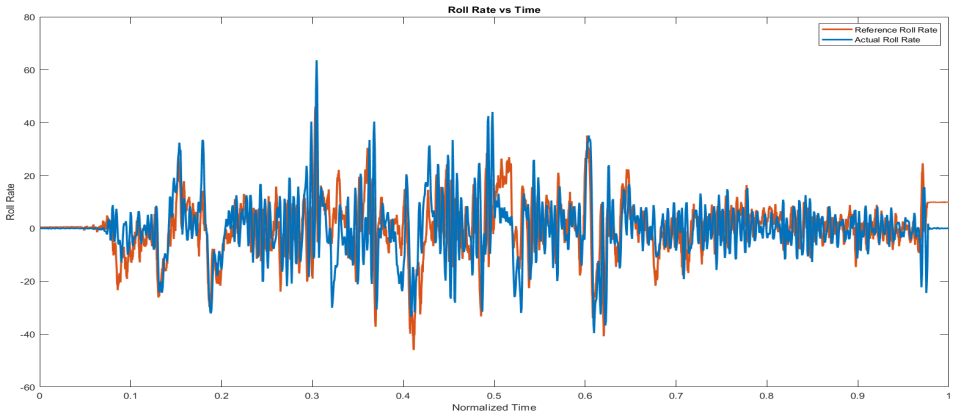


Figure 5.45: Roll Angular Rate

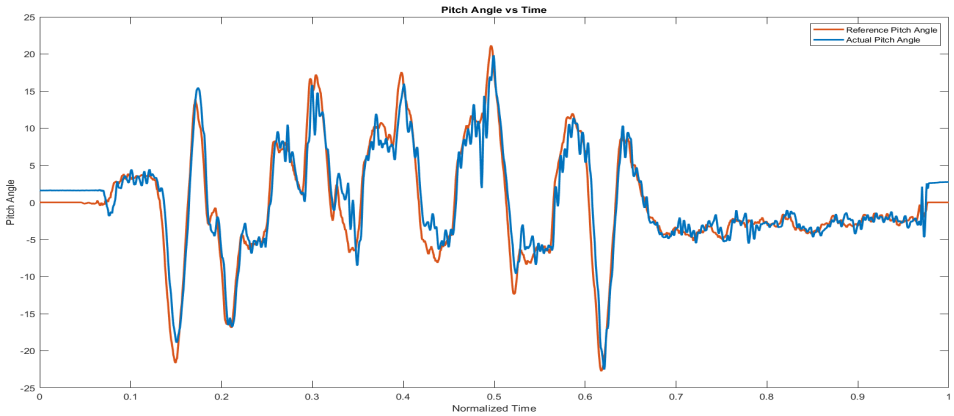


Figure 5.46: Pitch Angle

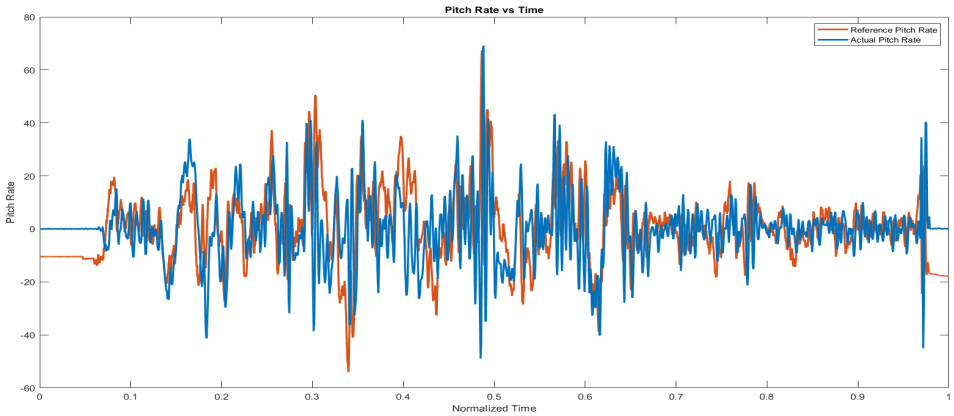


Figure 5.47: Pitch Angular Rate

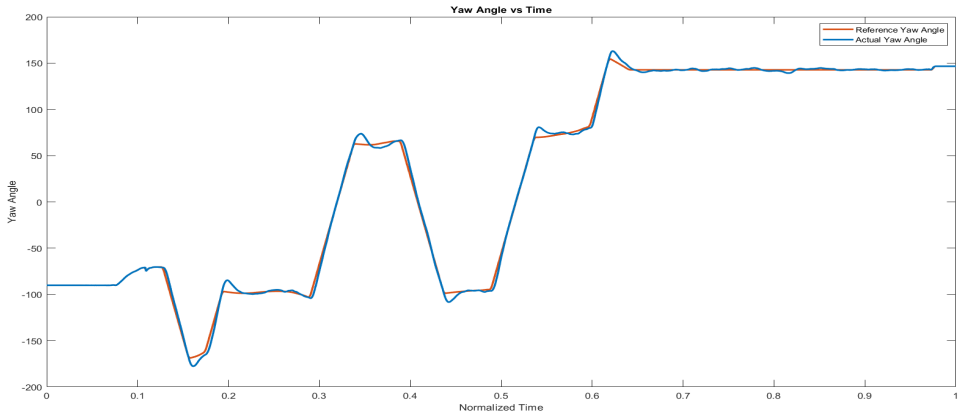


Figure 5.48: Yaw Angle

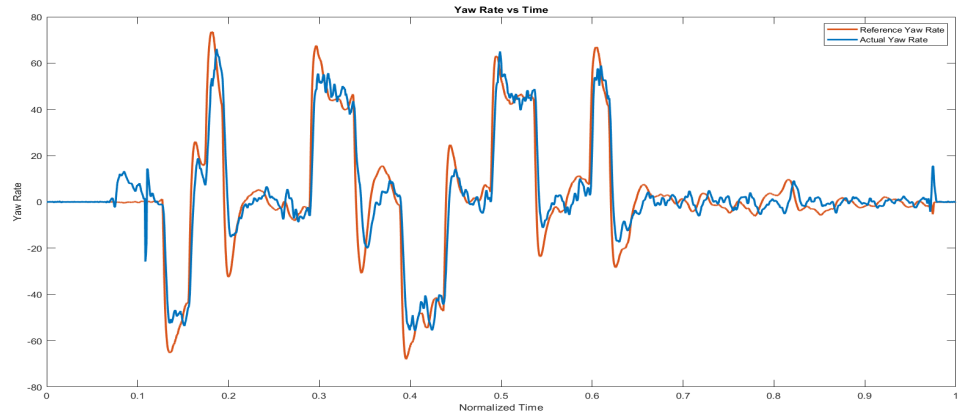


Figure 5.49: Yaw Angular Rate

3. **Mission Path 3**

The mission path provided replicated shape as shown in fig 5.50.

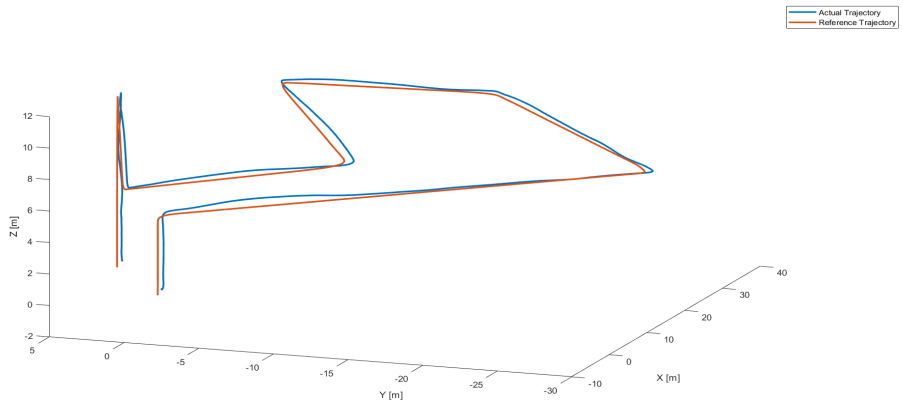


Figure 5.50: Mission path 3

The plots for roll ,pitch and yaw angles along wit respective rates for this trajectory

are given below :

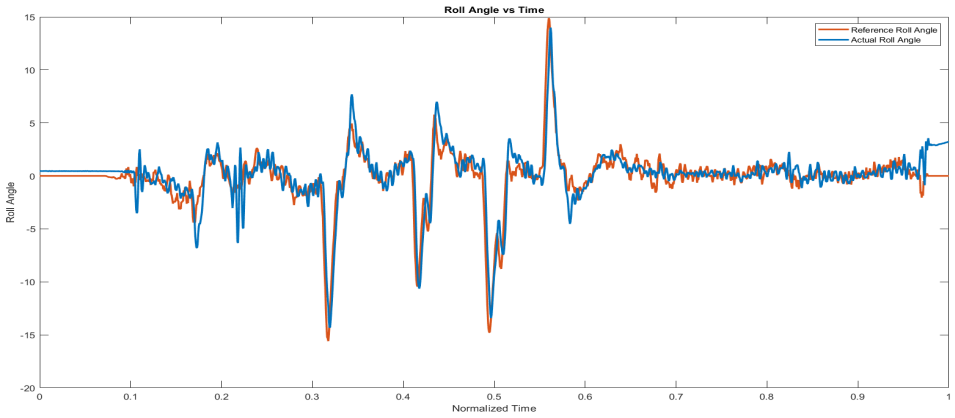


Figure 5.51: Roll Angle

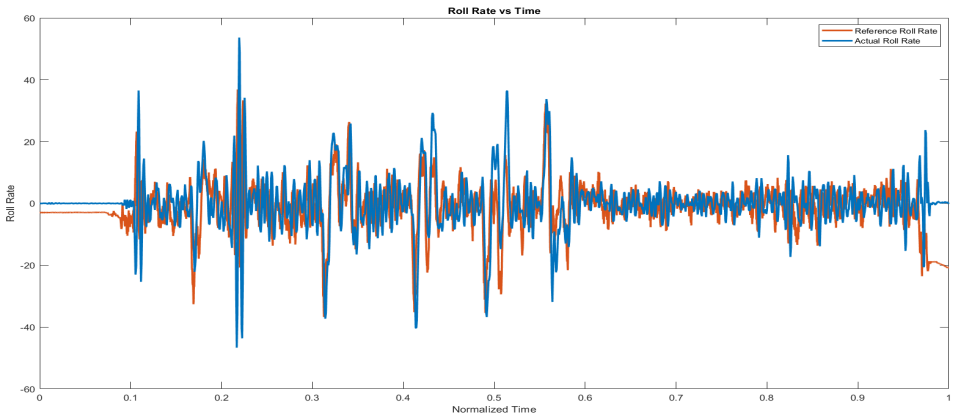


Figure 5.52: Roll Angular Rate

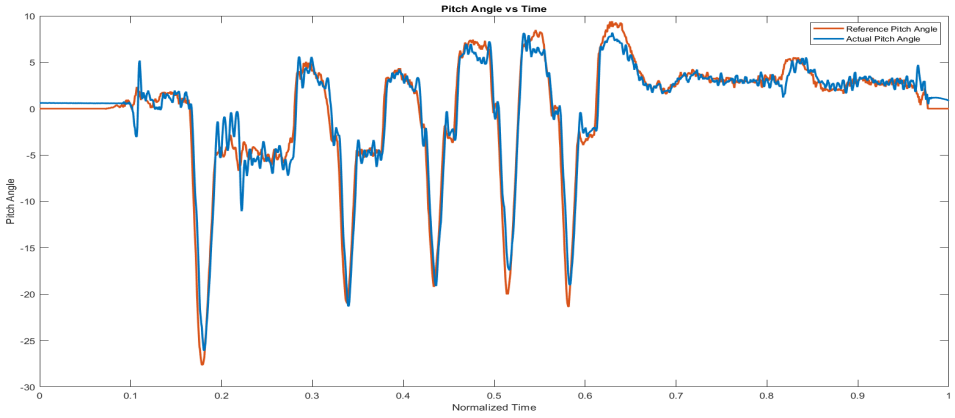


Figure 5.53: Pitch Angle

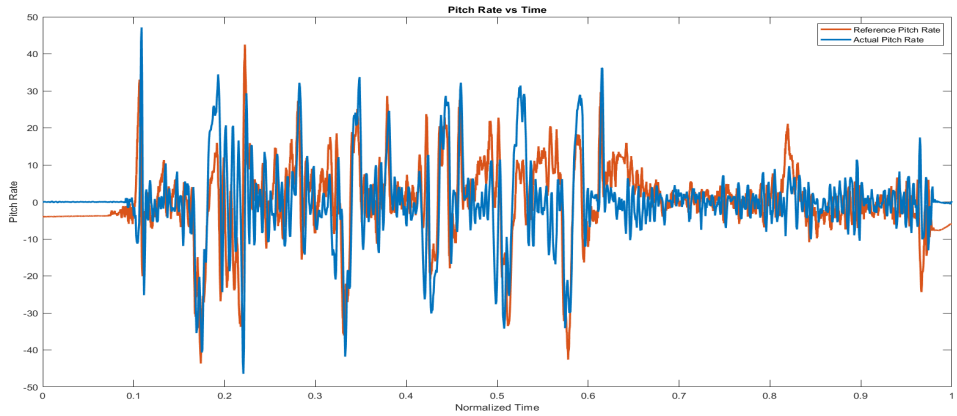


Figure 5.54: Pitch Angular Rate

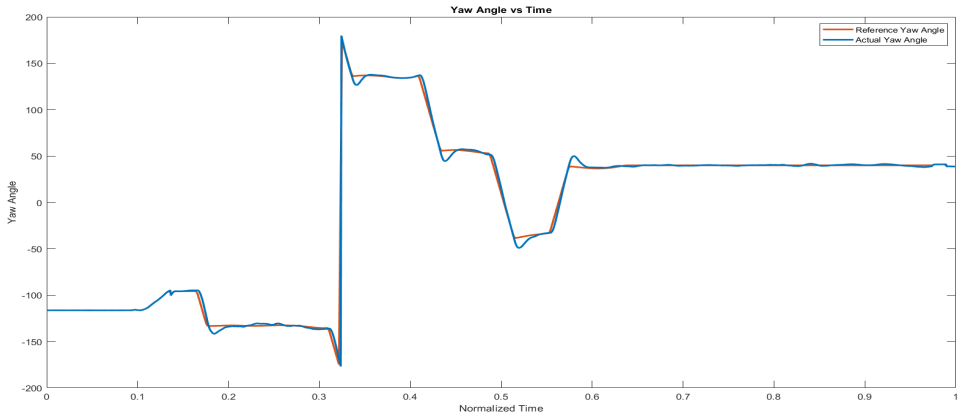


Figure 5.55: Yaw Angle

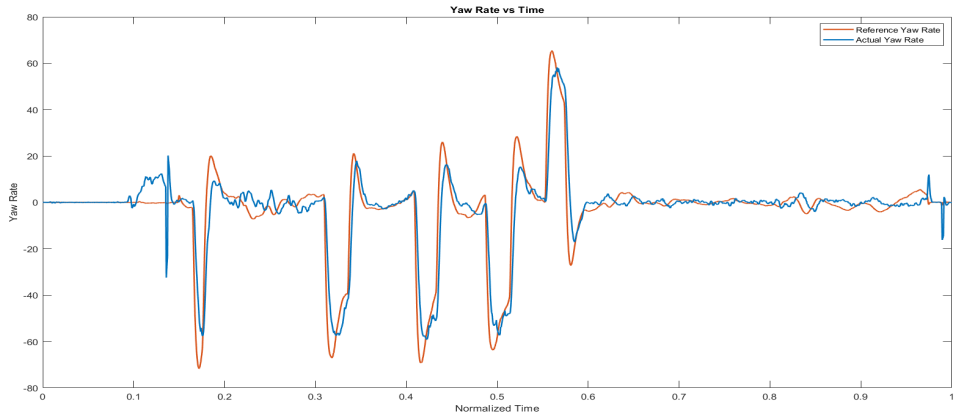
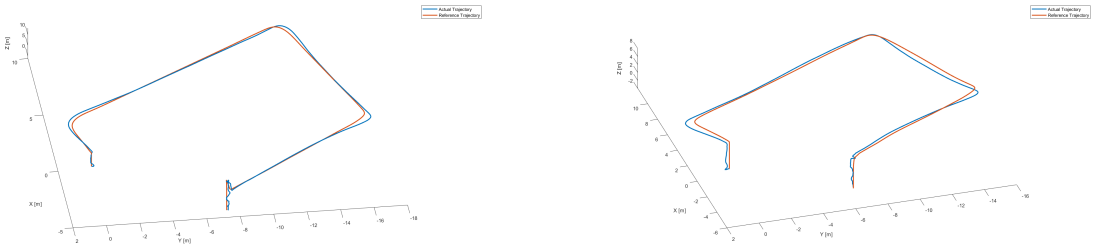


Figure 5.56: Yaw Angular Rate

4. **MPC and PID PX4-Autopilot Test Flight Results**

   Attitude results of the mission were obtained in the csv file format from ulog file by using the pyulog.

In this section, MPC and PID are visualized in the same trajectory .The variables of attitude control are plotted for MPC followed by PID for each separate entity.



(a) MPC implemented trajectory tracking (b) PID implemented trajectory tracking

Figure 5.57: Trajectory tracking performance

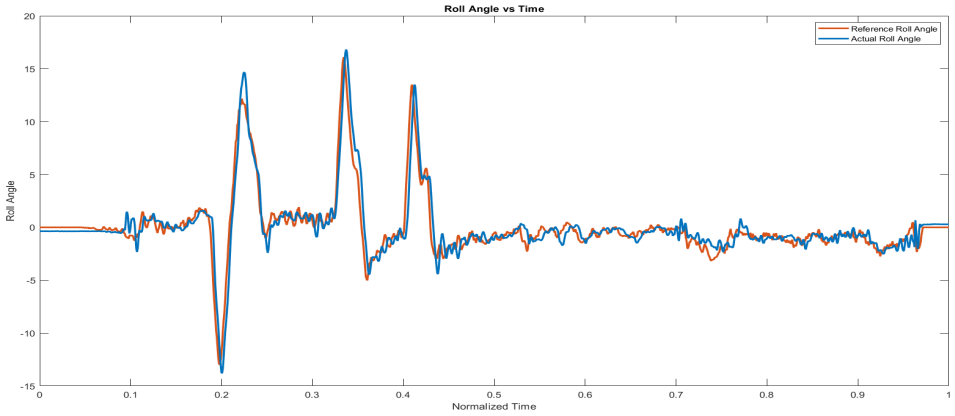


Figure 5.58: MPC Roll Angle

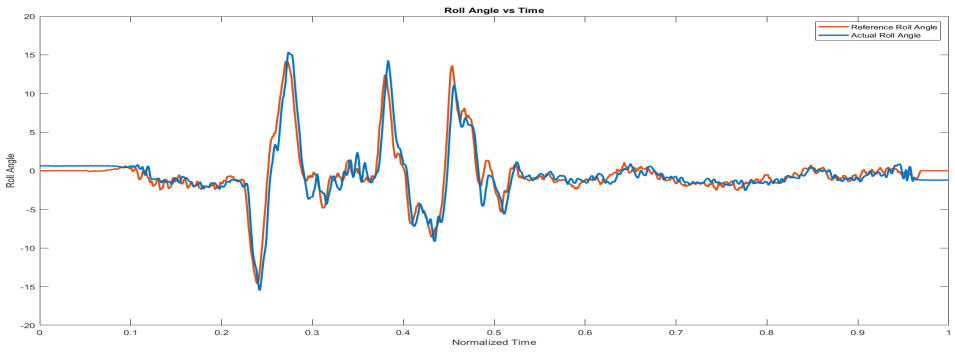


Figure 5.59: PID Roll Angle

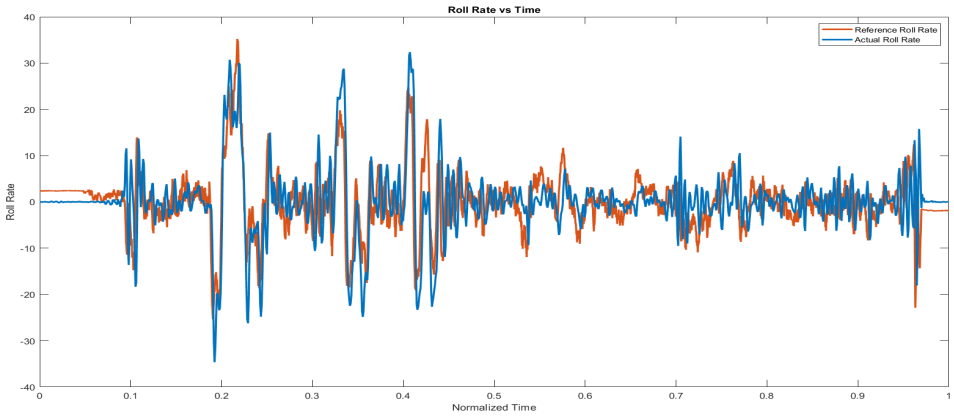


Figure 5.60: MPC Roll Angular Rate

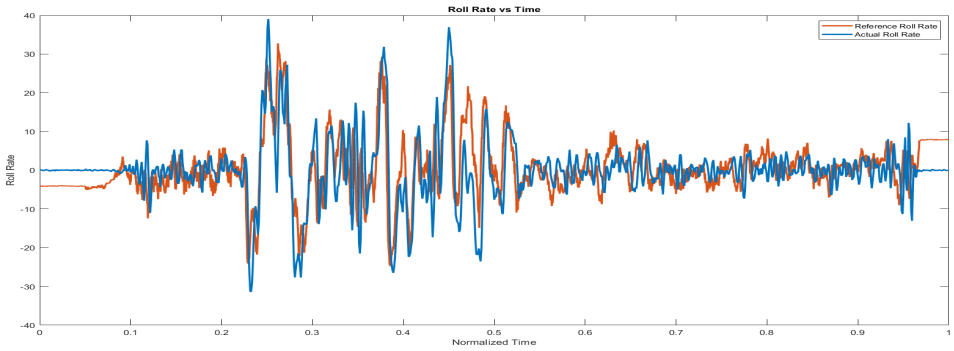


Figure 5.61: PID Roll Angular Rate

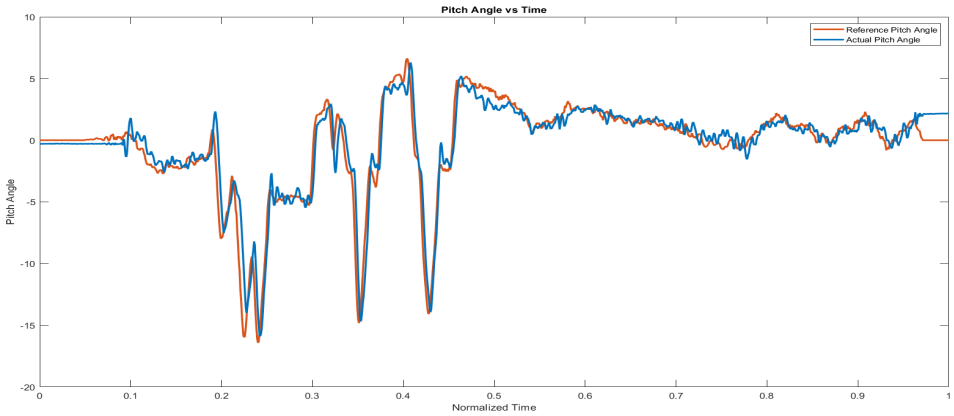


Figure 5.62: MPC Pitch Angle

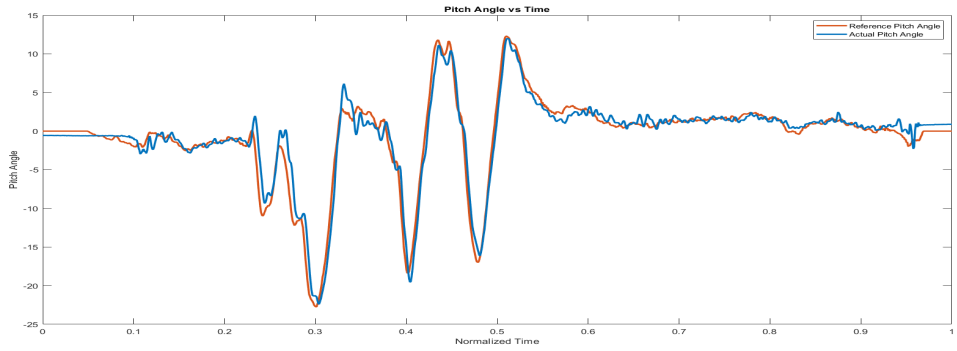


Figure 5.63: PID Pitch Angle

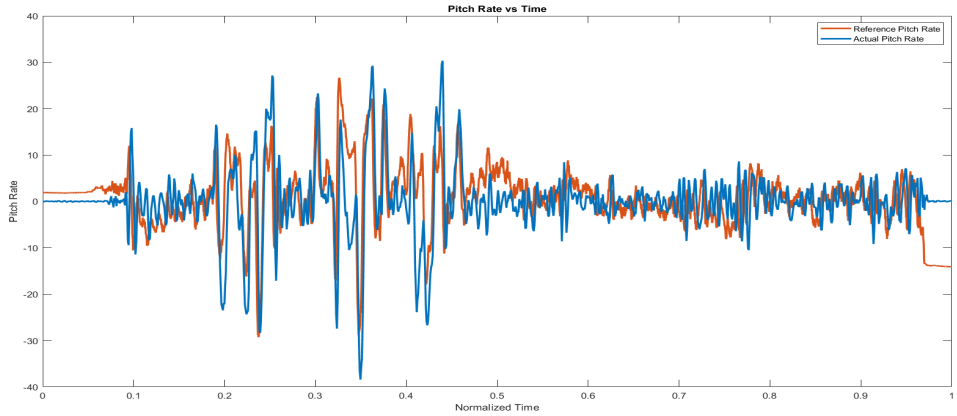


Figure 5.64: MPC Pitch Angular Rate

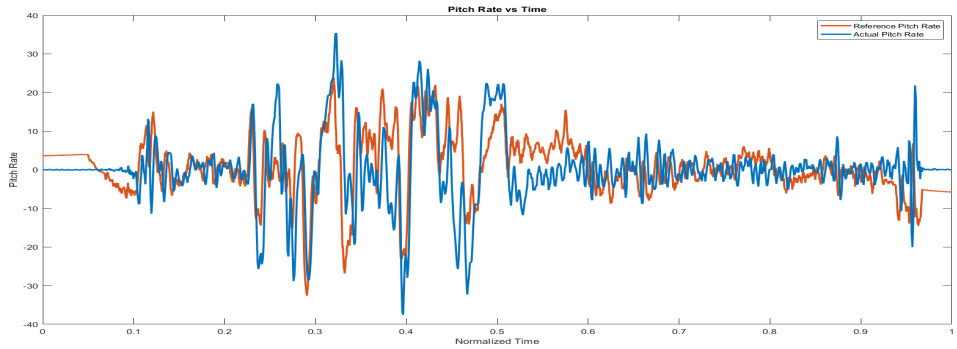


Figure 5.65: PID Pitch Angular Rate

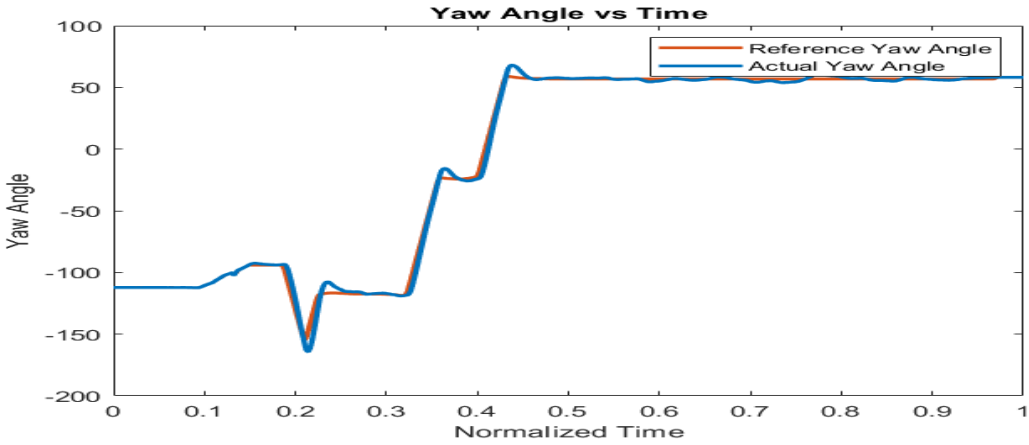


Figure 5.66: MPC Yaw Angle

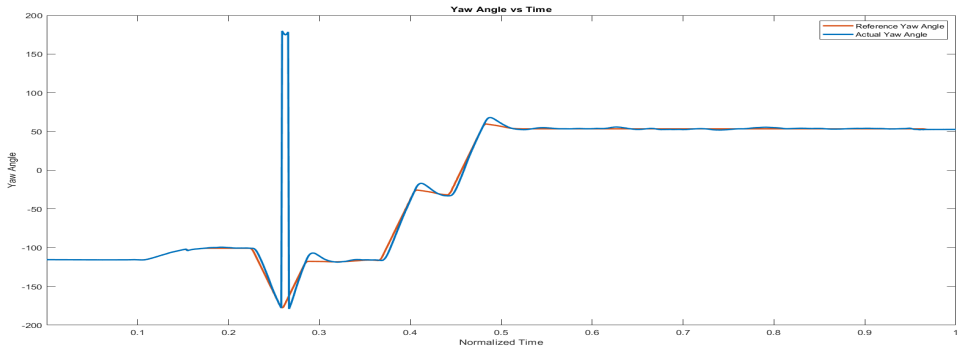


Figure 5.67: PID Yaw Angle

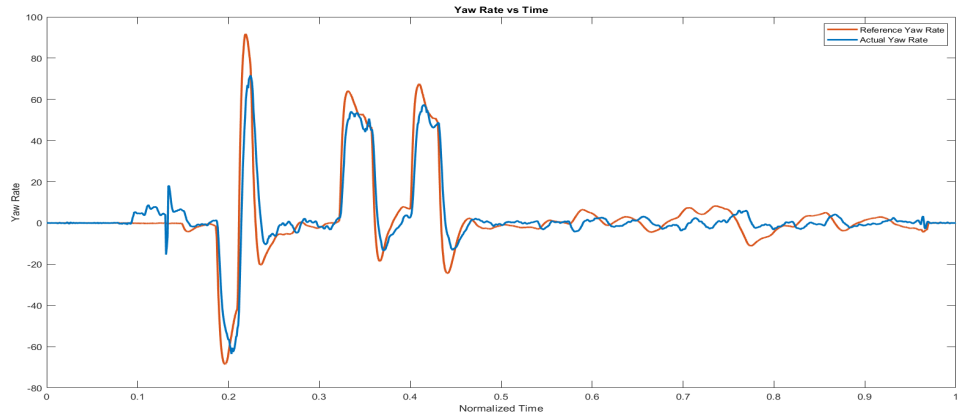


Figure 5.68: MPC Yaw Angular Rate

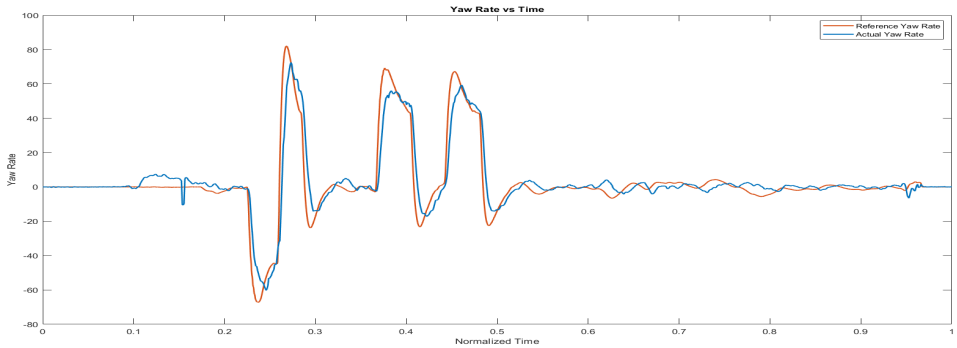


Figure 5.69: PID Yaw Angular Rate

The hardware integration of MPC in real quadrotor is inflicted by various disturbances like wind and deviation of numerical model from real-world model . Regardless of all this interferences, the angular controller MPC Based PX4 Firmware showed rewarding tracking of setpoints.

5.2.5. Unmodelled Disturbance Rejection Test Flight Results

At the end of the test flight, we intended to test the robustness (disturbance rejection capability in this case) of the quadrotor to unmodelled disturbances. The disturbance in this case was a directed gust of approximately 3 m/s to which the quadrotor are subjected to.

Firstly, the quadcopter was flown in hold mode without any wind(without turning on the wind tunnel) .Afterwards, the quadcopter was flown in test section of wind tunnel with a wind of 3.08 m/s.To test the hold capacity of MPC based PX4 firmware,the quadcopter was put in hold mode manually inside wind tunnel lab of Pulchowk Campus.The attitude response plots associated with this test are provided in this section.

The MPC based PX4 firmware showcased adequate hold capacity even in a wind of 3.08m/s. With further improvements, it depicted the potential of quadcopter to be used in real-time environment along with wind disturbance-negating features.

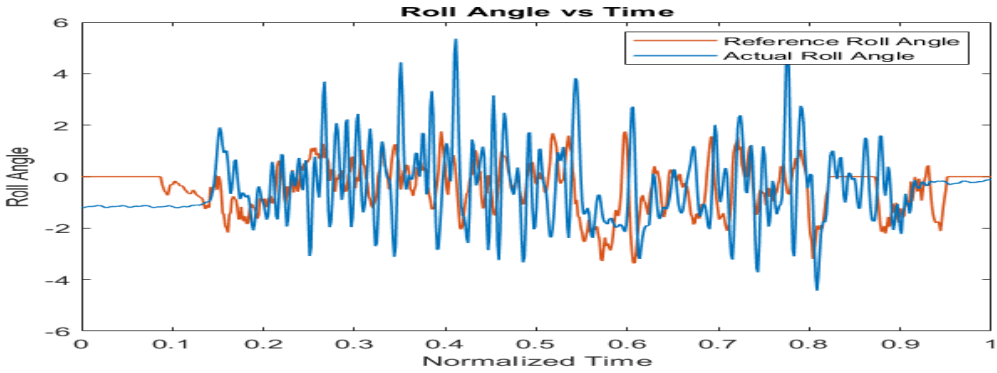


Figure 5.70: Roll Angle With Wind

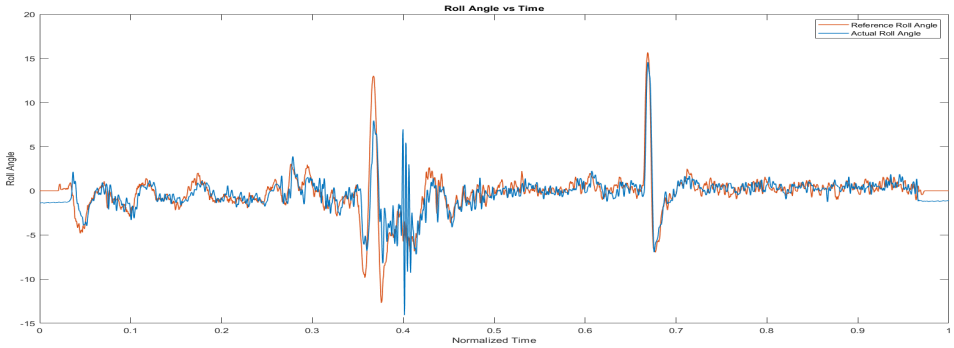


Figure 5.71: Roll Angle Without Wind

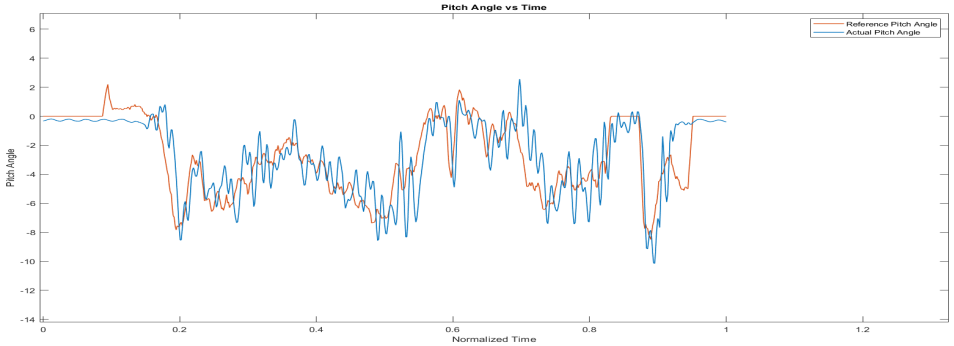


Figure 5.72: Pitch Angle With Wind

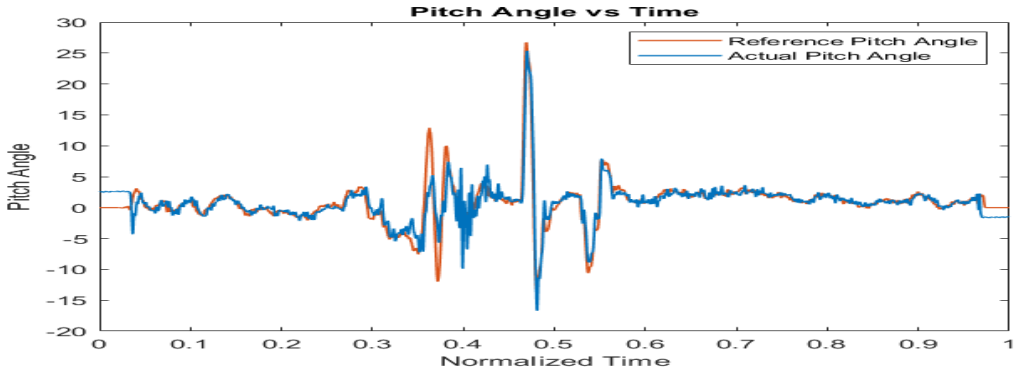


Figure 5.73: Pitch Angle Without Wind

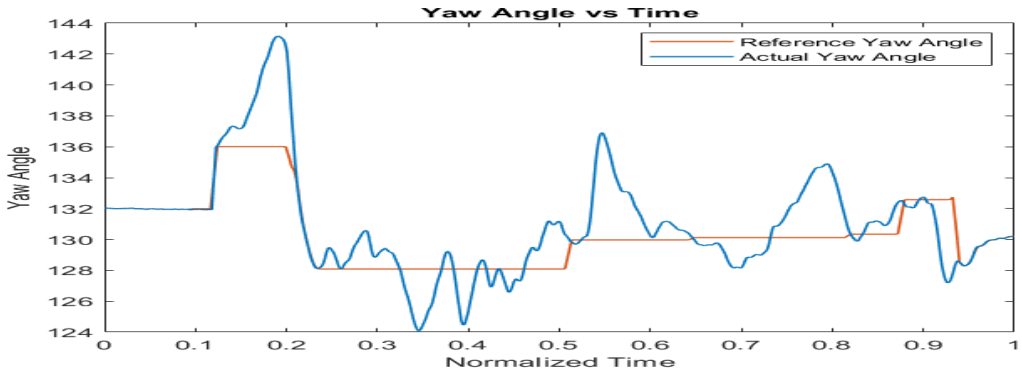


Figure 5.74: Yaw Angle With Wind

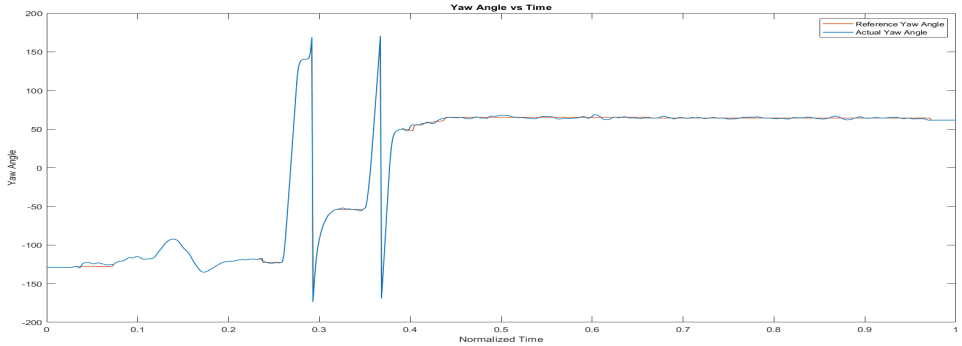


Figure 5.75: Yaw Angle Without Wind

5.3. Limitations

The proposed project will have the following limitations:

1. Collision detection and avoidance system is not included because of a significant increase in the complexity of modeling and computation.
2. There are three sub-controllers to control position, attitude, and altitude. For hardware implementation having such three controllers all running nonlinear optimization might not be the best practical approach or an approach that would result in the highest-performing platform.
3. Optimization will be limited by dynamic limits of the quadrotor due to limits in the power of motor and structures.
4. The experiment will be limited to the VLOS domain.
5. The proposed system is based on the assumption that the computational time required to run the MPC algorithm is lesser than the sampling interval so that the control input can be executed without delay.
6. The quadrotor will not facilitate long-duration flights.
7. GPS alone might not provide good navigation in congested areas. Such cases may need to rely on other systems like LIDAR, stereo vision cameras, etc.
8. The hardware implementation for the optimal control is limited to the attitude control only.

5.4. Problems Faced

Budget restriction did not allow us to use latest and more efficient accessories like a new GPS module, or oneshot ESCs. Besides it,some major problems faced by the team is listed below:

1. The inherent memory allocation of Pixhawk modules created a nuisance in building MPC code in the .px4 extension.
2. Due to the unavailability of large indoor open spaces for testing of the quadcopter, we could not do indoor testing that had to be done before outdoor testing.
3. The GPS module did not show satisfactory signal channels initially during the test of the robustness of the quadcopter to unmodeled disturbance (directed gust in this case) which created some inherent errors in position and attitude estimation.

5.5. Budget Expenses

An approximation of the budget expended is mentioned below:

Particulars	**Quantity**	**Rate(NRs.)**	**Net Amount(NRs.)**
GPS Module	1	3,000	3,000
Pixhawk - 4	1	14000	14000
Raspberry pi	1	12500	12,500
Brushless Motor	4	1,000	4,000
Drone Frame	1	2,000	2,000
Battery(3s cell Li-Po	1	3000	3000
Propeller	4	250	1000
ESC	4	900	3,600
Telemetry	1	4000	4000
cables and connectors			1500
Miscellaneous			5,000
Total			54,600

Table 5.1: Budget Expenditure

CHAPTER 6: CONCLUSION AND FUTURE ENHANCEMENT

6.1. Conclusion

The project aimed to validate the implementation of the mathematical model for a quadcopter, implement a basic Model Predictive Control (MPC) model for reference path tracking, and assemble a fully functional quadcopter. Initially, we developed a non linear Model Predictive Control based model involving non linear optimization of the cost function for reference path tracking keeping in check the constraints provided to the quadcopter for optimally tracking various reference paths. Upon finding that the non-linear MPC-based control was much more computationally intensive we opted to implement a linear model Predictive control strategy in the hardware integration phase. We transitioned from the computationally heavy non-linear MPC architecture to a more lightweight linear model tailored to meet the Pixhawk's specifications. The linear MPC-based control was implemented only to the attitude control module within the PX4 firmware. To seamlessly integrate MPC into the Pixhawk framework, Simulation through SITL was done for the PX4 firmware modified to accommodate MPC-based control in Gazebo.

While building the PX4 file, stack overflow was one of the major issue which was rectified by increasing the stack size of the attitude control module in the CMakeLists.txt file. After a successful build of the custom PX4 firmware with the optimal control implemented for the attitude, it was tested in a real environment through hardware implementation in quadcopter. The disturbance rejection of the quadcopter towards the gust, treated as an unmodelled disturbance in this project, can be further improved by incorporating more comprehensive dynamics, including environmental disturbances like gusts during the formulation of the optimal contol problem. Hence, the linear MPC model, through various flight tests, demonstrated satisfactory reference setpoint tracking capabilities for the attitude.

6.2. Scope for Future Enhancement

There are several areas for future enhancement and research that can further improve the quadcopter system. Some of them are enlisted below.

- Explore advanced control algorithms like adaptive or hybrid control strategies for improved trajectory tracking and disturbance rejection.
- Incorporate obstacle avoidance system in the quadcopter's model.

- Extend implementation of optimal control beyond attitude control module within PX4 firmware.
- Improve the formulation of the the objective function by including incorporating constraints related to environmental factors like wind gusts, Aerodynamic effects, etc.

listings

References

[1] T. Samad, M. Bauer, S. Bortoff, S. Di Cairano, L. Fagiano, P. F. Odgaard, R. R. Rhinehart, R. Sánchez-Peña, A. Serbezov, F. Ankersen *et al.*, "Industry engagement with control research: Perspective and messages," *Annual Reviews in Control*, vol. 49, pp. 1–14, 2020.

[2] B. Jiang, B. Li, W. Zhou, L.-Y. Lo, C.-K. Chen, and C.-Y. Wen, "Neural network based model predictive control for a quadrotor uav," *Aerospace*, vol. 9, no. 8, 2022. [Online]. Available: https://www.mdpi.com/2226-4310/9/8/460

[3] R. Findeisen and F. Allgöwer, "An introduction to nonlinear model predictive control," in *21st Benelux meeting on systems and control*, vol. 11. Veldhoven, 2002, pp. 119–141.

[4] D. Sotelo, A. Favela-Contreras, V. V. Kalashnikov, and C. Sotelo, "Model predictive control with a relaxed cost function for constrained linear systems," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–10, 2020.

[5] B. Lindqvist, S. S. Mansouri, A.-a. Agha-mohammadi, and G. Nikolakopoulos, "Nonlinear mpc for collision avoidance and control of uavs with dynamic obstacles," *IEEE robotics and automation letters*, vol. 5, no. 4, pp. 6001–6008, 2020.

[6] Y. Al Younes and M. Barczyk, "Nonlinear model predictive horizon for optimal trajectory generation," *Robotics*, vol. 10, no. 3, p. 90, 2021.

[7] A. Altan, Ö. Aslan, and R. Hacıoğlu, "Model predictive control of load transporting system on unmanned aerial vehicle (uav)," in *Fifth international conference on advances in mechanical and robotics engineering*. Institute of Research Engineers and Doctors Rome, Italy, 2017, pp. 1–4.

[8] K. Subbarao, C. Tule, and P. Ru, "Nonlinear model predictive control applied to trajectory tracking for unmanned aerial vehicles," 06 2015.

[9] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, vol. 22, no. 22, 2011.

[10] S. Bhattarai, K. Poudel, N. Bhatta, S. Mahat, S. Bhattarai, and K. S. Thapa Magar, "Modeling and development of baseline guidance navigation and control system for medical delivery uav," in *2018 AIAA Information Systems-AIAA Infotech@ Aerospace*, 2018, p. 0508.

[11] R. V. Lopes, "Model predictive control applied to tracking and attitude stabilization of a vtol quadrotor aircraft," 2011.

[12] C. Amadi, "Design and implementation of a model predictive control on a pixhawk flight controller." 2018.

[13] J. Jansen, "Autonomous localization and tracking for uavs using kalman filtering," Master's thesis, NTNU, 2014.

[14] M. George and S. Sukkarieh, "Tightly coupled ins/gps with bias estimation for uav applications," in *Proceedings of Australiasian Conference on Robotics and Automation (ACRA)*, 2005, pp. 1–7.

[15] G. Mao, S. Drake, and B. D. Anderson, "Design of an extended kalman filter for uav localization," in *2007 Information, Decision and Control*. IEEE, 2007, pp. 224–229.

[16] S. Antonov, A. Fehn, and A. Kugi, "Unscented kalman filter for vehicle state estimation," *Vehicle System Dynamics*, vol. 49, no. 9, pp. 1497–1520, 2011.

[17] PX4 Development Team. (2023) PX4 Autopilot. [Online]. Available: https://px4.io/

[18] J. Habeck and P. Seiler, "Moment of inertia estimation using a bifilar pendulum," 2016.

[19] C. Luis and J. L. Ny, "Design of a trajectory tracking controller for a nanoquadcopter," *arXiv preprint arXiv:1608.05786*, 2016.

[20] G. Ononiwu, A. Okoye, J. Onojo, and N. Onuekwusi, "Design and implementation of a real time wireless quadcopter for rescue operations," *American Journal of Engineering Research*, vol. 5, no. 9, pp. 130–138, 2016.

CHAPTER A: APPENDIX A

A.1. Parameter Definition

The specifications and parameters foe the quadcopter are explained below.

A.1.1. Mass of quadcopter

The mass of quadcopter ,along with three cell battery and all accessories , was found to be 1158 grams in a digital weighing machine.

A.1.2. Dimensions of quadcopter

The lateral length of quadcopter(straight distance between opposite arms) was found to be 0.49 m when measured using a measuring tape.The moment arm was calculated by measuring distance between two opposite motor centers(along lateral distance) and diving by two. The moment arm was found to be 0.235 m .
The clearance between base of battery and a levelled ground was calculated as 0.015m , while The height of motor base from ground was found to be 0.055 m.

A.1.3. Moments of Inertia of quadcopter

Oxford Dictionary defines Moments of Inertia as "a quantity expressing a body's tendency to resist angular acceleration, which is the sum of the products of the mass of each particle in the body with the square of its distance from the axis of rotation."
A bifilar pendulum consists of suspending an aircraft from two parallel wires, or filars, that allow it to rotate freely about a given axis. The experiment is to measure the moment of inertia for the axis of rotation parallel to the filars. A small moment is then applied to the aircraft to measure its period of oscillation, which allows further calculation of its angular frequency[18].The experiment typically involves the following steps:

1. Suspend the object using the two parallel wires.
2. Displace the object from its equilibrium position and release it.
3. Measure the time taken for the object to complete a certain number of oscillations (T).
4. Repeat the measurement for different numbers of oscillations to improve accuracy.
5. Calculate the moment of inertia using the formula

$$I = \frac{mgT^2d^2}{16\pi^2L} \quad \text{(C.3.1)}$$

where, m - mass of quadcopter (with battery connected), kg
g - acceleration due to gravity, ms^{-2}

T - period for one oscillation, s

d - distance between strings, m

L - length of string, m

The experiment for the determination of moment of intertia about X, Y and Z axes were conducted. The setup for the experiment is also shown in the image below.

Figure A.1: Bifilar Pendulum Experiment for MOI determination

The results from the experiments are presented in tables below:

Table A.1: Time for oscillations about X-axis

Run	**Time taken for 10 Oscillations (s)**
1	29.0
2	28.72
3	28

Table A.2: Time for oscillations about Z-axis

Run	**Time taken for 10 Oscillations (s)**
1	68.0
2	68.1
3	68.0

Moment of inertia for x-axis, Ixx

From values of table A.1, we have

Average oscillation:

$$= \frac{29.0 + 28.72 + 28.0}{3} = 28.57\,\text{s}$$

Period for one oscillation:

$$= \frac{28.57}{10} = 2.857\,\text{s}$$

For this experimental setup, the following values were measured:

$$L = 1.00\,\text{m}$$
$$d = 0.14\,\text{m}$$
$$m = 1.158\,\text{kg}$$

The moment of inertia is calculated below:

$$I_{xx} = \frac{1.158 \times 9.81 \times 2.857^2 \times 0.14^2}{16 \times \pi^2 \times 1.00} = 0.011497\,\text{kgm}^2$$

Moment of inertia for z-axis, Izz

From values of table A.2, we have

Average oscillation:

$$= \frac{68.0 + 68.1 + 68.0}{3} = 68.0\,\text{s}$$

Period for one oscillation:

$$= \frac{68.0}{10} = 6.80\,\text{s}$$

For this experimental setup, the following values were measured:

$$L = 0.89\,\text{m}$$
$$d = 0.065\,\text{m}$$
$$m = 1.158\,\text{kg}$$

The moment of inertia is calculated below:

$$I_{zz} = \frac{1.158 \times 9.81 \times 6.80^2 \times 0.065^2}{16 \times \pi^2 \times 0.89} = 0.0158\,\text{kgm}^2$$

Parameters	Specification
Mass	1.158 kg
Armlength	23.5 cm
Propeller diameter	10 inch
Blade pitch	11.43 cm
Ground clearance	5.5 cm
ESC	30 Amp
Motor	3542-1000 kv
Battery 3 cell Li-Poly	

Table A.3: Setup specifications

A.1.4. Drag Coefficient

The formulas used in this section to determine the drag coefficient were obtained from Luis and Ny [19].

The torque created by a propeller is calculated with the formula below,

$$Q = Cd\rho n^2 D^5$$

where,

Cd - non-dimensional drag coefficient

n - propeller speed, rps

D - rotor diameter, m

ρ - air density, kg/m^3

The formula for the drag coefficient of the propeller is presented in equation C.4.2 below,

$$k = \frac{Cd\rho D^5}{3600}$$

Taking the air density to be $1.225\,\text{kg/m}^3$.

$$k = 0.099 \times 1.225 \times 0.27^5 \div 3600 = 4.83379 \times 10^{-8}\,\text{Nm/rpm}^2$$

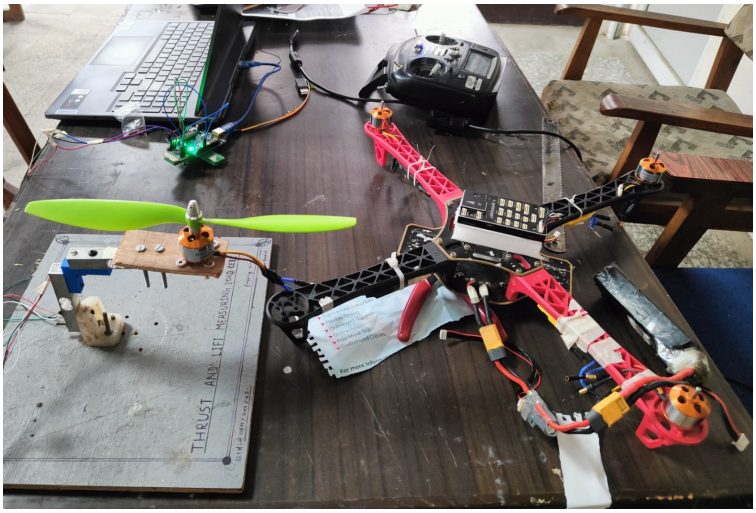
A.1.5. Load cell

Since our overall weight of quadrotor was estimated to be 1200 grams, minimum thrust of its four motors and propellers have to be at least 1200 grams in order to just fly. As we have 4 motors, each one has to produce 1200 ÷ 4= 300 grams of thrust (with assumptions that all motors/propellers produce equal thrust).As it is general rule to have 2:1 thrust/weight ratio for a standard quadcopter [20], for the weight of 1200 grams ,we required the standard thrust for each motor to be : 300 X 2 = 600 grams. From the above estimate for the thrust, 1000KV Brushless Motor was chosen.
Later, after assembly of quadrotor ,weight was found to be 1158 grams which was below our estimated weight of 1200. The motor was tested in Load Cell and found to produce 722 grams of thrust when connected to 3 cell LiPo battery.The thrust/weight ratio for our quadcopter was calculated as 5:2. Therefore the choice of motor is justified.

A.1.6. Thrust Coefficient

From Luis and Ny [19] , the formula for calculating the thrust of a propeller is as follows:

$$\text{Thrust, } T, \quad T = Ct\rho n^2 D^4$$



(a) Maximum Thrust Calculation using Load cell

where,

Ct - non-dimensional thrust coefficient

n - propeller speed, rps

D - rotor diameter, m

ρ - air density, kg/m^3

For uniformity, the propeller speed is converted to rpm. Using 1 rev/s = 60 rev/min, we have:

$$T = Ct\rho\left(\frac{\omega}{60}\right)^2 D^4$$

where ω is the propeller speed in rpm.

Using the diameter and air density values from the previous section, the thrust coefficient, b, is calculated as follows:

$$b = \frac{Ct\rho D^4}{3600} = 0.13 \times 1.225 \times 0.27^4 \div 3600 = 2.35089 \times 10^{-7}\,\text{N/rpm}^2$$

CHAPTER B: APPENDIX B

B.1. MATLAB Program

The main MATLAB code is written in main section. Each section other than main should be treated as a function.The MATLAB program describing dynamics of quadrotor is quadrotor.m and RK4 numerical method is RK4.m

B.1.1. main.m

```
% Numerical simulation of quadrotor dynamics with open loop control
% Reference: Modelling and control of quadcoptor (2011) by Teppo Luukkonen

clc
clear
close all

fprintf("Simulation start...\r\n");

% Time params
dt = 0.001;
stop_time = 2;
time = 0:dt:stop_time;

% Physical params
g = 9.81;         % Gravity's acceleration, m/(s*s)
m = 1;        % Mass, kg
l = 0.225;        % Rotor to COM distance, m
k = 2.980e-6;     % Lift coefficient
b = 1.140e-7;     % Drag constant
Im = 3.357e-5;    % MOI of rotor, kg*m*m
Ixx = 4.856e-3;   % Quadrotor MOI about X-axis, kg*m*m
Iyy = 4.856e-3;   % Quadrotor MOI about Y-axis, kg*m*m
Izz = 8.801e-3;   % Quadrotor MOI about Z-axis, kg*m*m

% Aerodynamic drag coefficients
Ax = 0.25; % kg/s
Ay = 0.25; % kg/s
Az = 0.25; % kg/s

phy_params = [g, m, l, k, b, Im, Ixx, Iyy, Izz];
aero_params = [Ax, Ay, Az];
```

```
% Initial conditions
xi = [0, 0, 0];
xi_dot = [0, 0, 0];
eta = [0, 0, 0];
eta_dot = [0, 0, 0];

% Rotor command profile params
t1s = 0;
t2s = 0.5;
t3s = 1;
t4s = 1.5;
t4e = 2;

amp_t1 = 65;
amp_t2 = 35;
amp_t3 = 0;
amp_t4 = -35;
offset = 620;

% Generate rotor command profile
w = zeros(4, length(time));
for i = 1:length(time)
  t = time(i);
  if t >= t1s && t < t2s
    w(1, i) = amp_t1 * sin(2 * pi * (t - t1s) / (t2s - t1s)) + offset;
    w(2, i) = w(1, i);
    w(3, i) = w(1, i);
    w(4, i) = w(1, i);
  elseif t >= t2s && t <= t3s
    w(1, i) = offset;
    w(2, i) = amp_t4 * sin(2 * pi * (t - t2s) / (t3s - t2s)) + offset;
    w(3, i) = offset;
    w(4, i) = amp_t2 * sin(2 * pi * (t - t2s) / (t3s - t2s)) + offset;
  elseif t >= t3s && t <= t4s
    w(1, i) = amp_t4 * sin(2 * pi * (t - t3s) / (t4s - t3s)) + offset;
    w(2, i) = offset;
    w(3, i) = amp_t2 * sin(2 * pi * (t - t3s) / (t4s - t3s)) + offset;
    w(4, i) = offset;
  elseif t >= t4s && t <= t4e
    w(1, i) = amp_t2 * sin(2 * pi * (t - t4s) / (t4e - t4s)) + offset;
    w(2, i) = amp_t4 * sin(2 * pi * (t - t4s) / (t4e - t4s)) + offset;
    w(3, i) = w(1, i);
    w(4, i) = w(2, i);
  end
```

```
end

% Memory allocation
state = zeros(12, length(time));
state(:,1) = [xi, xi_dot, eta, eta_dot]';

% Numerical integration
for t = 1:length(time)-1
  fn = @(y)quadcoptor(y, phy_params, aero_params, w(:,t));
  state(:,t+1) = RK4(fn, state(:,t), dt);
end

% Control input
figure;
plot(time, w(1,:), 'LineStyle', '-', 'LineWidth', 2); hold on;
plot(time, w(2,:), 'LineStyle', '--', 'LineWidth', 2);
plot(time, w(3,:), 'LineStyle', '-.', 'LineWidth', 2);
plot(time, w(4,:), 'LineStyle', ':', 'LineWidth', 2);
xlabel('Time (s)');
ylabel('Angular velocity (rad/sec)');
title('Control inputs: Rotor angular velocities');
legend('\omega_{1}', '\omega_{2}', '\omega_{3}', '\omega_{4}');
grid on; grid minor;

% Position
figure;
plot(time, state(1,:), 'LineStyle', '-', 'LineWidth', 2); hold on;
plot(time, state(2,:), 'LineStyle', '--', 'LineWidth', 2);
plot(time, state(3,:), 'LineStyle', ':', 'LineWidth', 2);
legend('x', 'y', 'z');
title('Positions x, y, and z');
ylabel('Position (m)');
xlabel('Time (s)');
grid on; grid minor;

% Orientation
figure;
plot(time, rad2deg(state(7,:)), 'LineStyle', '-', 'LineWidth', 2); hold on;
plot(time, rad2deg(state(8,:)), 'LineStyle', '--', 'LineWidth', 2);
plot(time, rad2deg(state(9,:)), 'LineStyle', ':', 'LineWidth', 2);
legend('\phi', '\theta', '\psi');
title('Angles \phi, \theta, and \psi');
xlabel('Time (s)');
ylabel('Angle (degree)');
grid on; grid minor;
```

```
fprintf("Simulation complete!\r\n");
```

B.1.2. quadcopter.m

```
function [state_dot] = quadcoptor(state, phy_params, aero_params, w)
  % Unpack state variables
  xi = state(1:3);
  xi_dot = state(4:6);
  n = state(7:9);
  n_dot = state(10:12);

  % Unpack physical params
  g = phy_params(1);
  m = phy_params(2);
  l = phy_params(3);
  k = phy_params(4);
  b = phy_params(5);
  Im = phy_params(6);
  Ixx = phy_params(7);
  Iyy = phy_params(8);
  Izz = phy_params(9);

  % Unpack aero-drag params
  Ax = aero_params(1);
  Ay = aero_params(2);
  Az = aero_params(3);

  % Angle transedental functions
  cr = cos(n(1)); % roll
  sr = sin(n(1));
  cp = cos(n(2)); % pitch
  sp = sin(n(2));
  tp = tan(n(2));
  cy = cos(n(3)); % yaw
  sy = sin(n(3));

  % *** Rotational dynamics *** %

  % Transformation matrix, its inverse, and derivative of the inverse
  W = [1, 0, -sp; 0, cr, cp * sr; 0, -sr, cp * cr];
  Winv = [1, sr * tp, cr * tp; 0, cr, -sr; 0, sr / cp, cr / cp];
  dWinv = zeros(3, 3);
```

```
  dWinv(1,2) = n_dot(1)*cr*tp + n_dot(2)*sr/cp^2;
  dWinv(1,3) = -n_dot(1)*sr*cp + n_dot(2)*cr/cp^2;
  dWinv(2,2) = -n_dot(1)*sr;
  dWinv(2,3) = -n_dot(1)*cr;
  dWinv(3,2) = n_dot(1)*(cr  + sr*tp)/cp;
  dWinv(3,3) = (-n_dot(1)*sr + n_dot(2)*cr*tp)/cp;

  % Angular velocity in body frame
  v = W * n_dot;

  % External and gyro torques in body frame from w
  tau = zeros(3, 1);
  tau(1) = l * k * (w(4)^2 - w(2)^2 );
  tau(2) = l * k * (w(3)^2 - w(1)^2);
  tau(3) = b * (-w(1)^2 + w(2)^2 - w(3)^2 + w(4)^2);
  tau_gyro = Im * (w(1) - w(2) + w(3) - w(4)) * [v(2), -v(1), 0]';

  % Angular acceleration in body frame
  I = diag([Ixx, Iyy, Izz]);
  v_dot = I \ (-cross(v, I * v) + tau - tau_gyro);

  % Double derivative of inertial angle
  n_ddot = dWinv * v + Winv * v_dot;

  % *** Translational dynamics *** %

  % Body Z-axis thrust from rotor angular speed
  T = k*(w(1)^2 + w(2)^2 + w(3)^2 + w(4)^2);

  % Forces in body frame acting on quadcoptor
  f_gravity = -g * m * [0, 0, 1]';
  f_thrust = T * [cy*sp*cr + sy*sr; sy*sp*cr - cy*sr; cp*cr];
  f_aero = -diag([Ax, Ay, Az]) * xi_dot;

  % Double derivative of inertial position
  xi_ddot = (f_gravity + f_thrust + f_aero) / m;

  % Differential of the state vector
  state_dot = [xi_dot; xi_ddot; n_dot; n_ddot];
end
```

B.1.3. RK4.m

```
  function y_update = RK4(f,y,dt)
  k1 = f(y);
  k2 = f(y + 0.5 * dt * k1);
  k3 = f(y + 0.5 * dt * k2);
  k4 = f(y + dt * k3);
  K = (1 / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
  y_update = y + K * dt;
end
```

B.2. Python Program for non linear optimization

B.2.1. main.py

```
import numpy as np
import math
import time
import matplotlib.pyplot as plt

from Quadrotor import Quadrotor
from Plotting import Plotting
from MPCController import AltitudeMPC, AttitudeMPC, PositionMPC

class Trajectory:
    def __init__(self, sim_time=10.0, dt = 0.02):
        self.sim_time = sim_time
        self.dt = dt
        self.ref = self.desiredTrajectory()

        self.x_ref = np.array(self.ref)[:,0]
        self.y_ref = np.array(self.ref)[:,1]
        self.z_ref = np.array(self.ref)[:,2]
        self.psi_ref = np.array(self.ref)[:,3]
```

```
def desiredTrajectory(self):

   ########LINEAR PATH#############
    ref = []
    for i in range(int(self.sim_time/self.dt)):
        t = i*self.dt
        x = 0.5*t
        y = 0.5*t
        z = 0.5*t
        yaw = 2*math.pi*t/10
        ref.append([x,y,z,yaw])
    return ref
    ############################
    ########CIRCULAR PATH#############
    ref = []
    for i in range(int(self.sim_time/self.dt)):
        t = i*self.dt
        x = 5*math.sin(2*math.pi*t/10)
        y = 5*math.cos(2*math.pi*t/10)
        z = 3
        yaw = 2*math.pi*t/10
        ref.append([x,y,z,yaw])
    return ref
    ############################
    ########SQUARE PATH#############
    ref = []
    square_side_length = 5
    altitude_rate = 0.5  # Rate at which altitude increases per unit of time
    helix_period = 20

    for i in range(int(self.sim_time / self.dt)):
        t = i * self.dt

        # Generate square helical trajectory
        side_time = self.sim_time / 4
        if 0 <= t < side_time:
            x = square_side_length
            y = square_side_length * (t / side_time)
            z = 3
        elif side_time <= t < 2 * side_time:
            x = square_side_length - square_side_length * ((t - side_time) / side_time)
            y = square_side_length
            z =- altitude_rate * t
        elif 2 * side_time <= t < 3 * side_time:
```

```
            x = 0
            y = square_side_length - square_side_length * ((t - 2 * side_time) / side_time)
            z = 3
        else:
            x = square_side_length * ((t - 3 * side_time) / side_time)
            y = 0
            z = 3
        yaw = 2 * math.pi * t / helix_period
        ref.append([x, y, z, yaw])
    return ref
    ############################
    #######TRIANGULR HELIX##############
     ref = []
    triangle_side_length = 5
    altitude_rate = 0.5  # Rate at which altitude increases per unit of time
    helix_period = 10

    for i in range(int(self.sim_time / self.dt)):
        t = i * self.dt

        # Generate triangular helical trajectory
        x = triangle_side_length * (t % (self.sim_time / 3)) / (self.sim_time / 3)
        y = x * math.sqrt(3) / 2  # Equilateral triangle
        z = altitude_rate * t
        yaw = 2 * math.pi * t / helix_period

        ref.append([x, y, z, yaw])
    return ref
    ########################################
    ########################################

def desired_altitude(self, quad, idx, N_):
    # initial state / last state
    x_ = np.zeros((N_+1, 2))
    u_ = np.zeros((N_, 1))

    z_ref_ = self.z_ref[idx:(idx+N_)]
    length = len(z_ref_)
    if length < N_:
        z_ex = np.ones(N_ - length)*z_ref_[-1]
        z_ref_ = np.concatenate((z_ref_, z_ex), axis=None)

    dz_ref_ = np.diff(z_ref_)
    dz_ref_ = np.concatenate((quad.dpos[2], dz_ref_), axis=None)
```

```
    ddz_ref_ = np.diff(dz_ref_)
    ddz_ref_ = np.concatenate((ddz_ref_[0], ddz_ref_), axis=None)

    thrust_ref_ = (quad.g + ddz_ref_)*quad.mq

    x_ = np.array([z_ref_, dz_ref_]).T
    x_ = np.concatenate((np.array([[quad.pos[2], quad.dpos[2]]]), x_), axis=0)
    u_ = np.array([thrust_ref_]).T
    # print(x_)
    # print(u_)
    return x_, u_

def desired_position(self, quad, idx, N_, thrust):
    # initial state / last state
    x_ = np.zeros((N_+1, 4))
    u_ = np.zeros((N_, 2))

    x_ref_ = self.x_ref[idx:(idx+N_)]
    y_ref_ = self.y_ref[idx:(idx+N_)]
    length = len(x_ref_)
    if length < N_:
        x_ex = np.ones(N_ - length)*x_ref_[-1]
        x_ref_ = np.concatenate((x_ref_, x_ex), axis=None)

        y_ex = np.ones(N_ - length)*y_ref_[-1]
        y_ref_ = np.concatenate((y_ref_, y_ex), axis=None)

    dx_ref_ = np.diff(x_ref_)
    dx_ref_ = np.concatenate((quad.dpos[0], dx_ref_), axis=None)
    dy_ref_ = np.diff(y_ref_)
    dy_ref_ = np.concatenate((quad.dpos[1], dy_ref_), axis=None)

    ddx_ref_ = np.diff(dx_ref_)
    ddx_ref_ = np.concatenate((ddx_ref_[0], ddx_ref_), axis=None)
    ddy_ref_ = np.diff(dy_ref_)
    ddy_ref_ = np.concatenate((ddy_ref_[0], ddy_ref_), axis=None)

    the_ref_ = np.arcsin(ddx_ref_*quad.mq/thrust)
    phi_ref_ = -np.arcsin(ddy_ref_*quad.mq/thrust)

    x_ = np.array([x_ref_, y_ref_, dx_ref_, dy_ref_]).T
    x_ = np.concatenate((np.array([[quad.pos[0],...
    quad.pos[1], quad.dpos[0], quad.dpos[1]]]), x_), axis=0)
    u_ = np.array([phi_ref_, the_ref_]).T
```

```
    # print(x_)
    # print(u_)
    return x_, u_

def desired_attitude(self, quad, idx, N_, phid, thed):
    # initial state / last state
    x_ = np.zeros((N_+1, 6))
    u_ = np.zeros((N_, 3))

    phi_ref_ = phid
    the_ref_ = thed

    psi_ref_ = self.psi_ref[idx:(idx+N_)]
    length = len(psi_ref_)
    if length < N_:
        psi_ex = np.ones(N_ - length)*psi_ref_[-1]
        psi_ref_ = np.concatenate((psi_ref_, psi_ex), axis=None)

    dphi_ref_ = np.diff(phi_ref_)
    dphi_ref_ = np.concatenate((quad.dori[0], dphi_ref_), axis=None)
    dthe_ref_ = np.diff(the_ref_)
    dthe_ref_ = np.concatenate((quad.dori[1], dthe_ref_), axis=None)
    dpsi_ref_ = np.diff(psi_ref_)
    dpsi_ref_ = np.concatenate((quad.dori[2], dpsi_ref_), axis=None)

    ddphi_ref_ = np.diff(dphi_ref_)
    ddphi_ref_ = np.concatenate((ddphi_ref_[0], ddphi_ref_), axis=None)
    ddthe_ref_ = np.diff(dthe_ref_)
    ddthe_ref_ = np.concatenate((ddthe_ref_[0], ddthe_ref_), axis=None)
    ddpsi_ref_ = np.diff(dpsi_ref_)
    ddpsi_ref_ = np.concatenate((ddpsi_ref_[0], ddpsi_ref_), axis=None)

    tau_phi_ref_ = (quad.Ix*ddphi_ref_ - dthe_ref_*dpsi_ref_*(quad.Iy-quad.Iz))/quad.la
    tau_the_ref_ = (quad.Iy*ddthe_ref_ - dphi_ref_*dpsi_ref_*(quad.Iz-quad.Ix))/quad.la
    tau_psi_ref_ =  quad.Iz*ddpsi_ref_ - dphi_ref_*dthe_ref_*(quad.Ix-quad.Iy)

    x_ = np.array([phi_ref_, the_ref_, psi_ref_, dphi_ref_, dthe_ref_, dpsi_ref_]).T
    x_ = np.concatenate((np.array([[quad.ori[0], quad.ori[1],...
    quad.ori[2], quad.dori[0], quad.dori[1], quad.dori[2]]]), x_), axis=0)
    u_ = np.array([tau_phi_ref_, tau_the_ref_, tau_psi_ref_]).T

    # print(x_)
    # print(u_)
    return x_, u_
```

```
# quad = Quadrotor()
# traj = Trajectory()
# traj.desired_altitude(quad, 495, np.array([1,2]), 30)

# exit()

if __name__ == "__main__":
    quad = Quadrotor()

    dt = 0.02
    N = 50
    sim_time = 10.0
    iner = 0

    traj = Trajectory(sim_time, dt)

    al = AltitudeMPC(quad, T=dt, N=N)
    po = PositionMPC(quad, T=dt, N=N)
    at = AttitudeMPC(quad, T=dt, N=N)

    his_thrust = []; his_tau_phi = []; his_tau_the = []; his_tau_psi = []
    his_time = []
    his_phids = []
    his_theds = []

    while iner - sim_time/dt < 0.0:
        # print(iner)
        # Solve altitude -> thrust
        next_al_trajectories, next_al_controls = traj.desired_altitude(quad, iner, N)
        thrusts = al.solve(next_al_trajectories, next_al_controls)

        # Solve position -> phid, thed
        next_po_trajectories, next_po_controls = traj.desired_position(quad, iner, N, thrusts)
        phids, theds = po.solve(next_po_trajectories, next_po_controls, thrusts)

        # Solve attitude -> tau_phi, tau_the, tau_psi
        next_at_trajectories,next_at_controls=traj.desired_attitude(quad, iner, N, phids,theds)
        tau_phis, tau_thes, tau_psis = at.solve(next_at_trajectories, next_at_controls)

        quad.updateConfiguration(thrusts[0], tau_phis[0], tau_thes[0], tau_psis[0], dt)



        # Store values
        his_thrust.append(thrusts[0])
        his_tau_phi.append(tau_phis[0])
```

```
    his_tau_the.append(tau_thes[0])
    his_tau_psi.append(tau_psis[0])
    his_time.append(iner*dt)
    # Make sure phids and theds have length 500
    his_phids.append(phids)
    his_theds.append(theds)


    iner += 1
# Convert quad.path to a NumPy array
quad_path_array = np.array(quad.path)
print(np.array(quad.path))
his_phids = np.array(his_phids)
his_theds = np.array(his_theds)

# Plot Drone
plot = Plotting("Quadrotor trajectory tracking")
plot.plot_path(quad_path_array)
plot.plot_path(traj.ref)
plt.legend()

plt.figure()
plt.subplot(311)
plt.plot(his_time, quad_path_array[:500, 0], label="Actual trajectory")
plt.plot(his_time, traj.x_ref[:500], label="Reference trajectory")
plt.title("position x")
plt.xlabel("Time [s]")
plt.ylabel("Value [m]")
plt.legend()


plt.subplot(312)
plt.plot(his_time, quad_path_array[:500, 1], label="Actual trajectory")
plt.plot(his_time, traj.y_ref[:500], label="Reference trajectory")
plt.title("position y")
plt.xlabel("Time [s]")
plt.ylabel("Value [m]")
plt.legend()


plt.subplot(313)
plt.plot(his_time, quad_path_array[:500, 2], label="Actual trajectory")
plt.plot(his_time, traj.z_ref[:500], label="Reference trajectory")
plt.title("position z")
plt.xlabel("Time [s]")
```

```
plt.ylabel("Value [m]")
plt.legend()

plt.figure()
plt.subplot(321)
plt.plot(his_time, quad_path_array[:500, 3], label="Actual trajectory")
plt.plot(his_time, his_phids[:, 0], label="Reference trajectory")
plt.title("position phi")
plt.xlabel("Time [s]")
plt.ylabel("Value [rad]")
plt.legend()


plt.subplot(322)
plt.plot(his_time, quad_path_array[:500, 4], label="Actual trajectory")
plt.plot(his_time, his_theds[:, 0], label="Reference trajectory")
plt.title("position the")
plt.xlabel("Time [s]")
plt.ylabel("Value [rad]")
plt.legend()


plt.subplot(323)
plt.plot(his_time, quad_path_array[:500, 5], label="Actual trajectory")
plt.plot(his_time, traj.psi_ref[:500], label="Reference trajectory")
plt.title("position psi")
plt.xlabel("Time [s]")
plt.ylabel("Value [rad]")
plt.legend()

# Plot control
plt.figure()
plt.subplot(331)
plt.plot(his_time, his_thrust)
plt.title("The total thrust")
plt.xlabel("Time [s]")
plt.ylabel("Value [N]")

plt.subplot(332)
plt.plot(his_time, his_tau_phi)
plt.title("The tau phi")
plt.xlabel("Time [s]")
plt.ylabel("Value [N.m]")

plt.subplot(333)
```

```
plt.plot(his_time, his_tau_the)
plt.title("The tau theta")
plt.xlabel("Time [s]")
plt.ylabel("Value [N.m]")

plt.subplot(334)
plt.plot(his_time, his_tau_psi)
plt.title("The tau psi")
plt.xlabel("Time [s]")
plt.ylabel("Value [N.m]")

# Plot derivatives of linear position (x, y, z)
plt.figure()
plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 0]) / dt, label="Actual trajectory")
plt.plot(his_time[:-1], np.diff(traj.x_ref[:500]) / dt, label="Reference trajectory")
plt.title("velocity x")
plt.xlabel("Time [s]")
plt.ylabel("Value [m/s]")
plt.legend()

plt.figure()
plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 1]) / dt, label="Actual trajectory")
plt.plot(his_time[:-1], np.diff(traj.y_ref[:500]) / dt, label="Reference trajectory")
plt.title("velocity y")
plt.xlabel("Time [s]")
plt.ylabel("Value [m/s]")
plt.legend()

plt.figure()
plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 2]) / dt, label="Actual trajectory")
plt.plot(his_time[:-1], np.diff(traj.z_ref[:500]) / dt, label="Reference trajectory")
plt.title("velocity z")
plt.xlabel("Time [s]")
plt.ylabel("Value [m/s]")
plt.legend()

# Plot derivatives of angular position (phi, theta, psi)
plt.figure()

plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 3]) / dt, label="Actual trajectory")
plt.plot(his_time[:-1], np.diff(his_phids[:, 0]) / dt, label="Reference trajectory")
plt.title("angular velocity phi")
plt.xlabel("Time [s]")
plt.ylabel("Value [rad/s]")
plt.legend()
```

```
    plt.figure()
    plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 4]) / dt, label="Actual trajectory")
    plt.plot(his_time[:-1], np.diff(his_theds[:, 0]) / dt, label="Reference trajectory")
    plt.title("angular velocity theta")
    plt.xlabel("Time [s]")
    plt.ylabel("Value [rad/s]")
    plt.legend()

    plt.figure()
    plt.plot(his_time[:-1], np.diff(quad_path_array[:500, 5]) / dt, label="Actual trajectory")
    plt.plot(his_time[:-1], np.diff(traj.psi_ref[:500]) / dt, label="Reference trajectory")
    plt.title("angular velocity psi")
    plt.xlabel("Time [s]")
    plt.ylabel("Value [rad/s]")
    plt.legend()

    plt.show()
```

B.2.2. Quadrotor.py

```
import numpy as np
import math
import time
import matplotlib.pyplot as plt

class Quadrotor:
    def __init__(self, pos=[0,0,0], ori=[0,0,0], dpos=[0,0,0], dori=[0,0,0]):
        # The configuration of quadrotor
        self.pos = np.array(pos)
        self.ori = np.array(ori)
        self.dpos = np.array(dpos)
        self.dori = np.array(dori)

        # The paths
        self.path = [np.append(self.pos, self.ori)]

        # The constant parameters of quadrotor
        self.mq = 1         # Mass of the quadrotor [kg]
        self.g = 9.8         # Gravity [m/s^2]
        self.Ix = 4e-3       # Moment of inertia about Bx axis [kg.m^2]
        self.Iy = 4e-3       # Moment of inertia about By axis [kg.m^2]
```

```
        self.Iz = 8.4e-3     # Moment of inertia about Bz axis [kg.m^2]
        self.la = 0.2        # Quadrotor arm length [m]
        self.b = 2.9e-6       # Thrust coefficient [N.s^2]
        self.d = 1.1e-6      # Drag coefficient [N.m.s^2]

        # The constraints of the quadrotor
        self.max_z = math.inf; self.min_z = 0
        self.max_phi = math.pi/2; self.min_phi = -self.max_phi
        self.max_the = math.pi/2; self.min_the = -self.max_the

        self.max_dx = 20.0; self.min_dx = -self.max_dx
        self.max_dy = 20.0; self.min_dy = -self.max_dy
        self.max_dz = 20.0; self.min_dz = -self.max_dz
        self.max_dphi = 1; self.min_dphi = -self.max_dphi
        self.max_dthe = 1; self.min_dthe = -self.max_dthe
        self.max_dpsi = 1; self.min_dpsi = -self.max_dpsi

        self.max_thrust = 15.0; self.min_thrust = 0.0
        self.max_tau_phi = 10.0; self.min_tau_phi = -self.max_tau_phi
        self.max_tau_the = 10.0; self.min_tau_the = -self.max_tau_the
        self.max_tau_psi = 10.0; self.min_tau_psi = -self.max_tau_psi

    def body2inertial(psi,theta,phi):
        Rx = np.array([[1, 0, 0],
                       [0, np.cos(phi), -np.sin(phi)],
                       [0, np.sin(phi), np.cos(phi)]])

        Ry=np.array([[np.cos(theta),0,np.sin(theta)],
                     [0,1,0],
                     [-np.sin(theta),0,np.cos(theta)]])
        Rz = np.array([[np.cos(psi), -np.sin(psi), 0],
                       [np.sin(psi), np.cos(psi), 0],
                       [0, 0, 1]])
        Rot_matrix=np.dot(Rx,np.dot(Ry,Rz))
        return Rot_matrix

    def correctControl(self, thrust, tau_phi, tau_the, tau_psi):
        thrust = min(max(thrust, self.min_thrust), self.max_thrust)
        tau_phi = min(max(tau_phi, self.min_tau_phi), self.max_tau_phi)
        tau_the = min(max(tau_the, self.min_tau_the), self.max_tau_the)
        tau_psi = min(max(tau_psi, self.min_tau_psi), self.max_tau_psi)
        return thrust, tau_phi, tau_the, tau_psi

    def correctDotState(self):
        self.dpos[0] = min(max(self.dpos[0], self.min_dx), self.max_dx)
```

```
        self.dpos[1] = min(max(self.dpos[1], self.min_dy), self.max_dy)
        self.dpos[2] = min(max(self.dpos[2], self.min_dz), self.max_dz)

        self.dori[0] = min(max(self.dori[0], self.min_dphi), self.max_dphi)
        self.dori[1] = min(max(self.dori[1], self.min_dthe), self.max_dthe)
        self.dori[2] = min(max(self.dori[2], self.min_dpsi), self.max_dpsi)

    def correctState(self):
        self.pos[2] = min(self.pos[2], self.max_z)

        self.ori[0] = min(max(self.ori[0], self.min_phi), self.max_phi)
        self.ori[1] = min(max(self.ori[1], self.min_the), self.max_the)

    def updateConfiguration(self, thrust, tau_phi, tau_the, tau_psi, dt):
        # Prepare the current state
        phi = self.ori[0]
        the = self.ori[1]
        psi = self.ori[2]

        dphi = self.dori[0]
        dthe = self.dori[1]
        dpsi = self.dori[2]

        # The dynamic equations
         thrust, tau_phi, tau_the, tau_psi =
                self.correctControl(thrust, tau_phi, tau_the,
                   tau_psi)
         ddx = thrust/self.mq*(np.cos(phi)*np.sin(the)*np.cos(psi) + np.sin(phi)*np.sin(psi))
         ddy = thrust/self.mq*(np.cos(phi)*np.sin(the)*np.sin(psi) - np.sin(phi)*np.cos(psi))
         ddz = self.g - thrust/self.mq*(np.cos(phi)*np.cos(the))

         ddpos = np.array([ddx, ddy, ddz])

        ddphi = (dthe*dpsi*(self.Iy - self.Iz) + tau_phi*self.la)/self.Ix
        ddthe = (dphi*dpsi*(self.Iz - self.Ix) + tau_the*self.la)/self.Iy
        ddpsi = (dphi*dthe*(self.Iz - self.Iy) + tau_psi)/self.Iz
        ddori = np.array([ddphi, ddthe, ddpsi])

        # Update the quadrotor states
        self.dpos = self.dpos + ddpos*dt
        self.dori = self.dori + ddori*dt
        self.correctDotState()

        self.pos = self.pos + self.dpos*dt
        self.ori = self.ori + self.dori*dt
```

```
        self.correctState()

        # Add current configuration to paths
        self.path.append(np.append(self.pos, self.ori))

    def updateConfigurationViaSpeed(self, o1, o2, o3, o4, dt):
        # Compute the control vector through angular speed
        thrust = self.b*(o1 + o2 + o3 + o4)
        tau_phi = self.b*(-o2 + o4)
        tau_the = self.b*(o1 - o3)
        tau_psi = self.d*(-o1 + o2 - o3 + o4)


        self.updateConfiguration(thrust, tau_phi, tau_the, tau_psi, dt)
```

B.2.3. MPCController.py

```
import casadi as ca
import numpy as np
import math
# import time
# import matplotlib.pyplot as plt

# from Quadrotor import Quadrotor

def shift(u, x_n):
    u_end = np.concatenate((u[1:], u[-1:]))
    x_n = np.concatenate((x_n[1:], x_n[-1:]))
    return u_end, x_n

class AltitudeMPC:
    def __init__(self, quad, T=0.02, N=30, Q=np.diag([40.0, 1.0]), R=np.diag([1.0])):
        self.quad = quad
        self.T = T  # time step
        self.N = N  # horizon length

        # weight matrix
        self.Q = Q
        self.R = R

        # The history states and controls
        self.next_states = np.zeros((self.N+1, 2))
        self.u0 = np.zeros((self.N, 1))
```

```
    self.setupController()

def setupController(self):
    self.opti = ca.Opti()
    # the total thrust
    self.opt_controls = self.opti.variable(self.N, 1)
    thrust = self.opt_controls

    # state variable: altitude position
    self.opt_states = self.opti.variable(self.N+1, 2)
    z = self.opt_states[:,0]
    dz = self.opt_states[:,1]

    # create model
    f = lambda x_, u_: ca.vertcat(*[
        x_[1],
        -self.quad.g + u_/self.quad.mq,
    ])

    # parameters, these parameters are the reference trajectories of the pose and inputs
    self.opt_u_ref = self.opti.parameter(self.N, 1)
    self.opt_x_ref = self.opti.parameter(self.N+1, 2)

    # initial condition
    self.opti.subject_to(self.opt_states[0, :] == self.opt_x_ref[0, :])
    for i in range(self.N):
        x_next = self.opt_states[i, :] + f(self.opt_states[i, :],...
        self.opt_controls[i, :]).T*self.T
        self.opti.subject_to(self.opt_states[i+1, :] == x_next)

    # cost function
    obj = 0
    for i in range(self.N):
        state_error_ = self.opt_states[i, :] - self.opt_x_ref[i+1, :]
        control_error_ = self.opt_controls[i, :] - self.opt_u_ref[i, :]
        obj = obj + ca.mtimes([state_error_, self.Q, state_error_.T]) \
                    + ca.mtimes([control_error_, self.R, control_error_.T])
    self.opti.minimize(obj)

    # boundary and control conditions
    self.opti.subject_to(self.opti.bounded(self.quad.min_z, z, self.quad.max_z))
    self.opti.subject_to(self.opti.bounded(self.quad.min_dz, dz, self.quad.max_dz))

    self.opti.subject_to(self.opti.bounded(self.quad.min_thrust thrust,...
```

```
        self.quad.max_thrust))

        opts_setting = {'ipopt.max_iter':2000,
                        'ipopt.print_level':0,
                        'print_time':0,
                        'ipopt.acceptable_tol':1e-8,
                        'ipopt.acceptable_obj_change_tol':1e-6}

        self.opti.solver('ipopt', opts_setting)

    def solve(self, next_trajectories, next_controls):
        ## set parameter, here only update initial state of x (x0)
        self.opti.set_value(self.opt_x_ref, next_trajectories)
        self.opti.set_value(self.opt_u_ref, next_controls)

        ## provide the initial guess of the optimization targets
        self.opti.set_initial(self.opt_states, self.next_states)
        self.opti.set_initial(self.opt_controls, self.u0.reshape(self.N, 1))

        ## solve the problem
        sol = self.opti.solve()

        ## obtain the control input
        u_res = sol.value(self.opt_controls)
        x_m = sol.value(self.opt_states)
        self.u0, self.next_states = shift(u_res, x_m)
        return u_res

class PositionMPC:
    def __init__(self, quad, T=0.02, N=30, Q=np.diag([40.0, 40.0, 1.0, 1.0])
     , R=np.diag([1.0, 1.0])):
        self.quad = quad
        self.T = T  # time step
        self.N = N  # horizon length

        # weight matrix
        self.Q = Q
        self.R = R

        # The history states and controls
        self.next_states = np.zeros((self.N+1, 4))
        self.u0 = np.zeros((self.N, 2))

        self.setupController()
```

```
def setupController(self):
    self.opti = ca.Opti()
    # the phi and theta refernece
    self.opt_controls = self.opti.variable(self.N, 2)
    phid = self.opt_controls[:,0]
    thed = self.opt_controls[:,1]

    # state variable: position (x,y)
    self.opt_states = self.opti.variable(self.N+1, 4)
    x = self.opt_states[:,0]
    y = self.opt_states[:,1]

    dx = self.opt_states[:,2]
    dy = self.opt_states[:,3]

    # create model
    f = lambda x_, u_, t_: ca.vertcat(*[
        x_[2], x_[3],                       # dx, dy
        ca.sin(u_[1])*t_/self.quad.mq,  # ddx
        -ca.sin(u_[0])*t_/self.quad.mq,  # ddy
    ])

    # parameters, these parameters are the reference trajectories of the pose and inputs
    self.thrust = self.opti.parameter(self.N, 1)
    self.opt_u_ref = self.opti.parameter(self.N, 2)
    self.opt_x_ref = self.opti.parameter(self.N+1, 4)

    # initial condition
    self.opti.subject_to(self.opt_states[0, :] == self.opt_x_ref[0, :])
    for i in range(self.N):
    x_next = self.opt_states[i, :] + f(self.opt_states[i, :], self.opt_controls[i, :],
          self.thrust[i,:]).T*self.T
    self.opti.subject_to(self.opt_states[i+1, :] == x_next)

    # cost function
    obj = 0
    for i in range(self.N):
        state_error_ = self.opt_states[i, :] - self.opt_x_ref[i+1, :]
        control_error_ = self.opt_controls[i, :] - self.opt_u_ref[i, :]
        obj = obj + ca.mtimes([state_error_, self.Q, state_error_.T]) \
                    + ca.mtimes([control_error_, self.R, control_error_.T])
    self.opti.minimize(obj)

    # boundary and control conditions
    self.opti.subject_to(self.opti.bounded(self.quad.min_dx, dx, self.quad.max_dx))
```

```
        self.opti.subject_to(self.opti.bounded(self.quad.min_dy, dy, self.quad.max_dy))

        self.opti.subject_to(self.opti.bounded(self.quad.min_phi, phid, self.quad.max_phi))
        self.opti.subject_to(self.opti.bounded(self.quad.min_the, thed, self.quad.max_the))

        opts_setting = {'ipopt.max_iter':2000,
                        'ipopt.print_level':0,
                        'print_time':0,
                        'ipopt.acceptable_tol':1e-8,
                        'ipopt.acceptable_obj_change_tol':1e-6}

        self.opti.solver('ipopt', opts_setting)

    def solve(self, next_trajectories, next_controls, thrust):
        ## set parameter, here only update initial state of x (x0)
        self.opti.set_value(self.opt_x_ref, next_trajectories)
        self.opti.set_value(self.opt_u_ref, next_controls)
        self.opti.set_value(self.thrust, thrust)

        ## provide the initial guess of the optimization targets
        self.opti.set_initial(self.opt_states, self.next_states)
        self.opti.set_initial(self.opt_controls, self.u0.reshape(self.N, 2))

        ## solve the problem
        sol = self.opti.solve()

        ## obtain the control input
        u_res = sol.value(self.opt_controls)
        x_m = sol.value(self.opt_states)
        self.u0, self.next_states = shift(u_res, x_m)
        return u_res[:,0], u_res[:,1]

class AttitudeMPC:
    def __init__(self, quad, T=0.02, N=30, Q=np.diag([40.0, 40.0, 40.0, 1.0, 1.0, 1.0]),
    R=np.diag([1.0, 1.0, 1.0])):
        self.quad = quad
        self.T = T  # time step
        self.N = N  # horizon length

        # weight matrix
        self.Q = Q
        self.R = R

        # The history states and controls
        self.next_states = np.zeros((self.N+1, 6))
```

```
        self.u0 = np.zeros((self.N, 3))

        self.setupController()

    def setupController(self):
        self.opti = ca.Opti()
        # the torques of all axis
        self.opt_controls = self.opti.variable(self.N, 3)
        tau_phi = self.opt_controls[:,0]
        tau_the = self.opt_controls[:,1]
        tau_psi = self.opt_controls[:,2]

        # state variable: orientation
        self.opt_states = self.opti.variable(self.N+1, 6)
        phi = self.opt_states[:,0]
        the = self.opt_states[:,1]
        psi = self.opt_states[:,2]

        dphi = self.opt_states[:,3]
        dthe = self.opt_states[:,4]
        dpsi = self.opt_states[:,5]

        # create model
        f = lambda x_, u_: ca.vertcat(*[
            x_[3],x_[4],x_[5], # dotphi, dotthe, dotpsi
            (x_[4]*x_[5]*(self.quad.Iy-self.quad.Iz) + self.quad.la*u_[0])/self.quad.Ix,
            (x_[3]*x_[5]*(self.quad.Iz-self.quad.Ix) + self.quad.la*u_[1])/self.quad.Iy,
            (x_[3]*x_[4]*(self.quad.Ix-self.quad.Iy) + u_[2])/self.quad.Iz,
        ])

        # parameters, these parameters are the reference trajectories of the pose and inputs
        self.opt_u_ref = self.opti.parameter(self.N, 3)
        self.opt_x_ref = self.opti.parameter(self.N+1, 6)

        # initial condition
        self.opti.subject_to(self.opt_states[0, :] == self.opt_x_ref[0, :])
        for i in range(self.N):
            x_next = self.opt_states[i, :] + f(self.opt_states[i,.....

            :], self.opt_controls[i, :]).T*self.T
            self.opti.subject_to(self.opt_states[i+1, :] == x_next)

        # cost function
        obj = 0
        for i in range(self.N):
```

```
            state_error_ = self.opt_states[i, :] - self.opt_x_ref[i+1, :]
            control_error_ = self.opt_controls[i, :] - self.opt_u_ref[i, :]
            obj = obj + ca.mtimes([state_error_, self.Q, state_error_.T]) \
                        + ca.mtimes([control_error_, self.R, control_error_.T])
        self.opti.minimize(obj)

        # boundary and control conditions
        self.opti.subject_to(self.opti.bounded(self.quad.min_phi, phi, self.quad.max_phi))
        self.opti.subject_to(self.opti.bounded(self.quad.min_the, the, self.quad.max_the))

    self.opti.subject_to(self.opti.bounded(self.quad.min_dphi, dphi, self.quad.max_dphi))
    self.opti.subject_to(self.opti.bounded(self.quad.min_dthe, dthe, self.quad.max_dthe))
    self.opti.subject_to(self.opti.bounded(self.quad.min_dpsi, dpsi, self.quad.max_dpsi))

self.opti.subjectto(self.opti.bounded(self.quad.min_tau_phi, tau_phi, self.quad.max_tau_phi))
self.opti.subjectto(self.opti.bounded(self.quad.min_tau_the, tau_the, self.quad.max_tau_the))
self.opti.subjectto(self.opti.bounded(self.quad.min_tau_psi, tau_psi, self.quad.max_tau_psi))

        opts_setting = {'ipopt.max_iter':2000,
                        'ipopt.print_level':0,
                        'print_time':0,
                        'ipopt.acceptable_tol':1e-8,
                        'ipopt.acceptable_obj_change_tol':1e-6}

        self.opti.solver('ipopt', opts_setting)

    def solve(self, next_trajectories, next_controls):
        ## set parameter, here only update initial state of x (x0)
        self.opti.set_value(self.opt_x_ref, next_trajectories)
        self.opti.set_value(self.opt_u_ref, next_controls)

        ## provide the initial guess of the optimization targets
        self.opti.set_initial(self.opt_states, self.next_states)
        self.opti.set_initial(self.opt_controls, self.u0.reshape(self.N, 3))

        ## solve the problem
        sol = self.opti.solve()

        ## obtain the control input
        u_res = sol.value(self.opt_controls)
        x_m = sol.value(self.opt_states)
        self.u0, self.next_states = shift(u_res, x_m)
        return u_res[:,0], u_res[:,1], u_res[:,2]
```

B.2.4. Plotting.py

```
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits import mplot3d

class Plotting:
    def __init__(self, name, xlim=[-6,6], ylim=[-6,6], zlim=[0,10], is_grid=True):
        self.fig = plt.figure()
        self.ax = plt.axes(projection ='3d')
        self.ax.set_title(name)
        self.ax.grid(is_grid)
        self.ax.set_xlim(xlim)
        self.ax.set_ylim(ylim)
        self.ax.set_zlim(zlim)
        self.ax.set_xlabel('x [m]')
        self.ax.set_ylabel('y [m]')
        self.ax.set_zlabel('z [m]')

    def plot_path(self, path):
        path = np.array(path)
        self.ax.plot(path[:,0], path[:,1], path[:,2])
```

CHAPTER C: APPENDIX C

C.1. PX4 Firmware

The MPC attitude control C++ code is present in this section. The changes are done in the attitude control module within the PX4 firmware which is implemented with the PX4 based matrix library. The changes are done within PX4-Autipilot/src/modules/mc_att_control.

C.1.1. mc_att_control.cpp

We added MPC code within the inbuilt main code of attitude control module of PX4.This section depicts the overall program required for attitude control with MPC formulation. In the main .cpp file of the original PX4 firmware a class has been created for the implementation of the optimal control.

```
{
/****************************************************************************
 *
 *   Copyright (c) 2013-2018 PX4 Development Team. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. Neither the name PX4 nor the names of its contributors may be
 *    used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
```

```
29  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
30  * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
31  * POSSIBILITY OF SUCH DAMAGE.
32  *
33  ****************************************************************************
    */
34
35 /**
36  * @file mc_att_control_main.cpp
37  * Multicopter attitude controller.
38  *
39  * @author Lorenz Meier         <lorenz@px4.io>
40  * @author Anton Babushkin      <anton.babushkin@me.com>
41  * @author Sander Smeets        <sander@droneslab.com>
42  * @author Matthias Grob        <maetugr@gmail.com>
43  * @author Beat K ng            <beat-kueng@gmx.net>
44  *
45  */
46
47 #include "mc_att_control.hpp"
48
49 #include <drivers/drv_hrt.h>
50 #include <mathlib/math/Limits.hpp>
51 #include <mathlib/math/Functions.hpp>
52
53 #include "AttitudeControl/AttitudeControlMath.hpp"
54
55 using namespace matrix;
56
57 MulticopterAttitudeControl::MulticopterAttitudeControl(bool vtol) :
58         ModuleParams(nullptr),
59         WorkItem(MODULE_NAME, px4::wq_configurations::nav_and_controllers)
            ,
60         _vehicle_attitude_setpoint_pub(vtol ? ORB_ID(
            mc_virtual_attitude_setpoint) : ORB_ID(
            vehicle_attitude_setpoint)),
61         _loop_perf(perf_alloc(PC_ELAPSED, MODULE_NAME": cycle")),
62         _vtol(vtol)
63 {
64         parameters_updated();
65         // Rate of change 5% per second -> 1.6 seconds to ramp to default
            8% MPC_MANTHR_MIN
66         _manual_throttle_minimum.setSlewRate(0.05f);
67         // Rate of change 50% per second -> 2 seconds to ramp to 100%
68         _manual_throttle_maximum.setSlewRate(0.5f);
```

```
69  }
70
71  MulticopterAttitudeControl::~MulticopterAttitudeControl()
72  {
73          perf_free(_loop_perf);
74  }
75
76  bool
77  MulticopterAttitudeControl::init()
78  {
79          if (!_vehicle_attitude_sub.registerCallback()) {
80                  PX4_ERR("callback registration failed");
81                  return false;
82          }
83
84          return true;
85  }
86
87  void
88  MulticopterAttitudeControl::parameters_updated()
89  {
90          // Store some of the parameters in a more convenient way &
               precompute often-used values
91          _attitude_control.setProportionalGain(Vector3f(_param_mc_roll_p.
               get(), _param_mc_pitch_p.get(), _param_mc_yaw_p.get()),
92                                                  _param_mc_yaw_weight.get());
93
94          // angular rate limits
95          using math::radians;
96          _attitude_control.setRateLimit(Vector3f(radians(
               _param_mc_rollrate_max.get()), radians(_param_mc_pitchrate_max.
               get()),
97                                                  radians(
                                                       _param_mc_yawrate_max.
                                                       get())));
98
99          _man_tilt_max = math::radians(_param_mpc_man_tilt_max.get());
100 }
101
102 float
103 MulticopterAttitudeControl::throttle_curve(float throttle_stick_input)
104 {
105         float thrust = 0.f;
106
107         switch (_param_mpc_thr_curve.get()) {
```

```
108         case 1: // no rescaling to hover throttle
109                 thrust = math::interpolate(throttle_stick_input, -1.f, 1.f
                        ,
110                                 _manual_throttle_minimum.getState(),
                                        _param_mpc_thr_max.get());
111                 break;
112
113         default: // 0 or other: rescale such that a centered throttle
                stick corresponds to hover throttle
114                 thrust = math::interpolateNXY(throttle_stick_input, {-1.f,
                         0.f, 1.f},
115                 {_manual_throttle_minimum.getState(), _param_mpc_thr_hover
                        .get(), _param_mpc_thr_max.get()});
116                 break;
117         }
118
119         return math::min(thrust, _manual_throttle_maximum.getState());
120 }
121
122 void
123 MulticopterAttitudeControl::generate_attitude_setpoint(const Quatf &q,
        float dt, bool reset_yaw_sp)
124 {
125         vehicle_attitude_setpoint_s attitude_setpoint{};
126         const float yaw = Eulerf(q).psi();
127
128         attitude_setpoint.yaw_sp_move_rate = _manual_control_setpoint.yaw
                * math::radians(_param_mpc_man_y_max.get());
129
130         // Avoid accumulating absolute yaw error with arming stick gesture
                 in case heading_good_for_control stays true
131         if ((_manual_control_setpoint.throttle < -.9f) && (
                _param_mc_airmode.get() != 2)) {
132                 reset_yaw_sp = true;
133         }
134
135         // Make sure not absolute heading error builds up
136         if (reset_yaw_sp) {
137                 _man_yaw_sp = yaw;
138
139         } else {
140                 _man_yaw_sp = wrap_pi(_man_yaw_sp + attitude_setpoint.
                        yaw_sp_move_rate * dt);
141         }
142
```

```
143         /*
144          * Input mapping for roll & pitch setpoints
145          * ----------------------------------------
146          * We control the following 2 angles:
147          * - tilt angle, given by sqrt(roll*roll + pitch*pitch)
148          * - the direction of the maximum tilt in the XY-plane, which also
                 defines the direction of the motion
149          *
150          * This allows a simple limitation of the tilt angle, the vehicle
                flies towards the direction that the stick
151          * points to, and changes of the stick input are linear.
152          */
153         _man_roll_input_filter.setParameters(dt, _param_mc_man_tilt_tau.
                get());
154         _man_pitch_input_filter.setParameters(dt, _param_mc_man_tilt_tau.
                get());
155
156         // we want to fly towards the direction of (roll, pitch)
157         Vector2f v = Vector2f(_man_roll_input_filter.update(
                _manual_control_setpoint.roll * _man_tilt_max),
158                               -_man_pitch_input_filter.update(
                                    _manual_control_setpoint.pitch *
                                    _man_tilt_max));
159         float v_norm = v.norm(); // the norm of v defines the tilt angle
160
161         if (v_norm > _man_tilt_max) { // limit to the configured maximum
                tilt angle
162                 v *= _man_tilt_max / v_norm;
163         }
164
165         Quatf q_sp_rp = AxisAnglef(v(0), v(1), 0.f);
166         // The axis angle can change the yaw as well (noticeable at higher
                tilt angles).
167         // This is the formula by how much the yaw changes:
168         //   let a := tilt angle, b := atan(y/x) (direction of maximum
                tilt)
169         //   yaw = atan(-2 * sin(b) * cos(b) * sin^2(a/2) / (1 - 2 * cos
                ^2(b) * sin^2(a/2))).
170         const Quatf q_sp_yaw(cosf(_man_yaw_sp / 2.f), 0.f, 0.f, sinf(
                _man_yaw_sp / 2.f));
171
172         if (_vtol) {
173                 // Modify the setpoints for roll and pitch such that they
                        reflect the user's intention even
174                 // if a large yaw error(yaw_sp - yaw) is present. In the
```

```
                presence of a yaw error constructing
175             // an attitude setpoint from the yaw setpoint will lead to
                    unexpected attitude behaviour from
176             // the user's view as the tilt will not be aligned with
                the heading of the vehicle.
177
178             AttitudeControlMath::correctTiltSetpointForYawError(
                q_sp_rp, q, q_sp_yaw);
179         }
180
181         // Align the desired tilt with the yaw setpoint
182         Quatf q_sp = q_sp_yaw * q_sp_rp;
183
184         q_sp.copyTo(attitude_setpoint.q_d);
185
186         // Transform to euler angles for logging only
187         const Eulerf euler_sp(q_sp);
188         attitude_setpoint.roll_body = euler_sp(0);
189         attitude_setpoint.pitch_body = euler_sp(1);
190         attitude_setpoint.yaw_body = euler_sp(2);
191
192         attitude_setpoint.thrust_body[2] = -throttle_curve(
            _manual_control_setpoint.throttle);
193
194         attitude_setpoint.timestamp = hrt_absolute_time();
195         _vehicle_attitude_setpoint_pub.publish(attitude_setpoint);
196 }
197
198 void
199 MulticopterAttitudeControl::Run()
200 {
201         if (should_exit()) {
202                 _vehicle_attitude_sub.unregisterCallback();
203                 exit_and_cleanup();
204                 return;
205         }
206
207         perf_begin(_loop_perf);
208
209         // Check if parameters have changed
210         if (_parameter_update_sub.updated()) {
211                 // clear update
212                 parameter_update_s param_update;
213                 _parameter_update_sub.copy(&param_update);
214
```

```
215                 updateParams();
216                 parameters_updated();
217         }
218
219         // run controller on attitude updates
220         vehicle_attitude_s v_att;
221
222         if (_vehicle_attitude_sub.update(&v_att)) {
223
224                 // Guard against too small (< 0.2ms) and too large (> 20ms
                        ) dt's.
225                 const float dt = math::constrain(((v_att.timestamp_sample
                        - _last_run) * 1e-6f), 0.0002f, 0.02f);
226                 _last_run = v_att.timestamp_sample;
227
228                 const Quatf q{v_att.q};
229
230                 /* check for updates in other topics */
231                 _manual_control_setpoint_sub.update(&
                        _manual_control_setpoint);
232                 _vehicle_control_mode_sub.update(&_vehicle_control_mode);
233
234                 if (_vehicle_status_sub.updated()) {
235                         vehicle_status_s vehicle_status;
236
237                         if (_vehicle_status_sub.copy(&vehicle_status)) {
238                                 _vehicle_type_rotary_wing = (
                                        vehicle_status.vehicle_type ==
                                        vehicle_status_s::
                                        VEHICLE_TYPE_ROTARY_WING);
239                                 _vtol = vehicle_status.is_vtol;
240                                 _vtol_in_transition_mode = vehicle_status.
                                        in_transition_mode;
241                                 _vtol_tailsitter = vehicle_status.
                                        is_vtol_tailsitter;
242
243                                 const bool armed = (vehicle_status.
                                        arming_state == vehicle_status_s::
                                        ARMING_STATE_ARMED);
244                                 _spooled_up = armed && hrt_elapsed_time(&
                                        vehicle_status.armed_time) >
                                        _param_com_spoolup_time.get() * 1_s;
245                         }
246                 }
247
```

```
248                 if (_vehicle_land_detected_sub.updated()) {
249                         vehicle_land_detected_s vehicle_land_detected;
250
251                         if (_vehicle_land_detected_sub.copy(&
                               vehicle_land_detected)) {
252                                 _landed = vehicle_land_detected.landed;
253                         }
254                 }
255
256                 if (_vehicle_local_position_sub.updated()) {
257                         vehicle_local_position_s vehicle_local_position;
258
259                         if (_vehicle_local_position_sub.copy(&
                               vehicle_local_position)) {
260                                 _heading_good_for_control =
                                    vehicle_local_position.
                                    heading_good_for_control;
261                         }
262                 }
263
264                 bool attitude_setpoint_generated = false;
265
266                 const bool is_hovering = (_vehicle_type_rotary_wing && !
                    _vtol_in_transition_mode);
267
268                 // vehicle is a tailsitter in transition mode
269                 const bool is_tailsitter_transition = (_vtol_tailsitter &&
                     _vtol_in_transition_mode);
270
271                 const bool run_att_ctrl = _vehicle_control_mode.
                    flag_control_attitude_enabled && (is_hovering
272                                         || is_tailsitter_transition);
273
274                 if (run_att_ctrl) {
275
276                         // Generate the attitude setpoint from stick
                            inputs if we are in Manual/Stabilized mode
277                         if (_vehicle_control_mode.
                            flag_control_manual_enabled &&
278                             !_vehicle_control_mode.
                                flag_control_altitude_enabled &&
279                             !_vehicle_control_mode.
                                flag_control_velocity_enabled &&
280                             !_vehicle_control_mode.
                                flag_control_position_enabled) {
```

```
281
282                         generate_attitude_setpoint(q, dt,
                               _reset_yaw_sp);
283                         attitude_setpoint_generated = true;
284
285                 } else {
286                         _man_roll_input_filter.reset(0.f);
287                         _man_pitch_input_filter.reset(0.f);
288                 }
289
290                 // Check for new attitude setpoint
291                 if (_vehicle_attitude_setpoint_sub.updated()) {
292                         vehicle_attitude_setpoint_s
                               vehicle_attitude_setpoint;
293
294                         if (_vehicle_attitude_setpoint_sub.copy(&
                               vehicle_attitude_setpoint)
295                             && (vehicle_attitude_setpoint.
                                   timestamp > _last_attitude_setpoint
                                   )) {
296
297                                 _attitude_control.
                                       setAttitudeSetpoint(Quatf(
                                       vehicle_attitude_setpoint.q_d),
                                        vehicle_attitude_setpoint.
                                       yaw_sp_move_rate);
298                                 _thrust_setpoint_body = Vector3f(
                                       vehicle_attitude_setpoint.
                                       thrust_body);
299                                 _last_attitude_setpoint =
                                       vehicle_attitude_setpoint.
                                       timestamp;
300                         }
301                 }
302
303                 // Check for a heading reset
304                 if (_quat_reset_counter != v_att.
                       quat_reset_counter) {
305                         const Quatf delta_q_reset(v_att.
                               delta_q_reset);
306
307                         // for stabilized attitude generation only
                                extract the heading change from the
                               delta quaternion
308                         _man_yaw_sp = wrap_pi(_man_yaw_sp + Eulerf
```

```
                        (delta_q_reset).psi());
309
310                     if (v_att.timestamp >
                        _last_attitude_setpoint) {
311                         // adapt existing attitude
                            setpoint unless it was
                            generated after the current
                            attitude estimate
312                         _attitude_control.
                            adaptAttitudeSetpoint(
                            delta_q_reset);
313                     }
314
315                     _quat_reset_counter = v_att.
                        quat_reset_counter;
316                 }
317
318                 Vector3f rates_sp = _attitude_control.update(q);
319
320                 const hrt_abstime now = hrt_absolute_time();
321                 autotune_attitude_control_status_s pid_autotune;
322
323                 if (_autotune_attitude_control_status_sub.copy(&
                    pid_autotune)) {
324                     if ((pid_autotune.state ==
                        autotune_attitude_control_status_s::
                        STATE_ROLL
325                       || pid_autotune.state ==
                          autotune_attitude_control_status_s
                          ::STATE_PITCH
326                       || pid_autotune.state ==
                          autotune_attitude_control_status_s
                          ::STATE_YAW
327                       || pid_autotune.state ==
                          autotune_attitude_control_status_s
                          ::STATE_TEST)
328                      && ((now - pid_autotune.timestamp) < 1
                         _s)) {
329                         rates_sp += Vector3f(pid_autotune.
                            rate_sp);
330                     }
331                 }
332
333                 // publish rate setpoint
334                 vehicle_rates_setpoint_s rates_setpoint{};
```

```
335                         rates_setpoint.roll = rates_sp(0);
336                         rates_setpoint.pitch = rates_sp(1);
337                         rates_setpoint.yaw = rates_sp(2);
338                         _thrust_setpoint_body.copyTo(rates_setpoint.
                               thrust_body);
339                         rates_setpoint.timestamp = hrt_absolute_time();
340
341                         _vehicle_rates_setpoint_pub.publish(rates_setpoint
                               );
342                 }
343
344                 if (_landed) {
345                         _manual_throttle_minimum.update(0.f, dt);
346
347                 } else {
348                         _manual_throttle_minimum.update(
                               _param_mpc_manthr_min.get(), dt);
349                 }
350
351                 if (_spooled_up) {
352                         _manual_throttle_maximum.update(1.f, dt);
353
354                 } else {
355                         _manual_throttle_maximum.setForcedValue(0.f);
356                 }
357
358                 // reset yaw setpoint during transitions, tailsitter.cpp
                       generates
359                 // attitude setpoint for the transition
360                 _reset_yaw_sp = !attitude_setpoint_generated || !
                       _heading_good_for_control || (_vtol &&
                       _vtol_in_transition_mode);
361         }
362
363         perf_end(_loop_perf);
364 }
365
366 int MulticopterAttitudeControl::task_spawn(int argc, char *argv[])
367 {
368         bool vtol = false;
369
370         if (argc > 1) {
371                 if (strcmp(argv[1], "vtol") == 0) {
372                         vtol = true;
373                 }
```

```
374         }
375
376         MulticopterAttitudeControl *instance = new
               MulticopterAttitudeControl(vtol);
377
378         if (instance) {
379                 _object.store(instance);
380                 _task_id = task_id_is_work_queue;
381
382                 if (instance->init()) {
383                         return PX4_OK;
384                 }
385
386         } else {
387                 PX4_ERR("alloc failed");
388         }
389
390         delete instance;
391         _object.store(nullptr);
392         _task_id = -1;
393
394         return PX4_ERROR;
395 }
396
397 int MulticopterAttitudeControl::custom_command(int argc, char *argv[])
398 {
399         return print_usage("unknown command");
400 }
401
402 int MulticopterAttitudeControl::print_usage(const char *reason)
403 {
404         if (reason) {
405                 PX4_WARN("%s\n", reason);
406         }
407
408         PRINT_MODULE_DESCRIPTION(
409                 R"DESCR_STR(
410 ### Description
411 This implements the multicopter attitude controller. It takes attitude
412 setpoints (‘vehicle_attitude_setpoint‘) as inputs and outputs a rate
       setpoint.
413
414 The controller has a P loop for angular error
415
416 Publication documenting the implemented Quaternion Attitude Control:
```

```
417 Nonlinear Quadrocopter Attitude Control (2013)
418 by Dario Brescianini, Markus Hehn and Raffaello D'Andrea
419 Institute for Dynamic Systems and Control (IDSC), ETH Zurich
420
421 https://www.research-collection.ethz.ch/bitstream/handle
       /20.500.11850/154099/eth-7387-01.pdf
422
423 )DESCR_STR");
424
425         PRINT_MODULE_USAGE_NAME("mc_att_control", "controller");
426         PRINT_MODULE_USAGE_COMMAND("start");
427         PRINT_MODULE_USAGE_ARG("vtol", "VTOL mode", true);
428         PRINT_MODULE_USAGE_DEFAULT_COMMANDS();
429
430         return 0;
431 }
432 ////////////////UPTO HERE SAME////////////
433 /// START THE MODEL/////////
434 void MulticopterAttitudeControl::control_attitude_rates(float dt)
435 {
436         Matrix<float,6, 1> QPhild(const Matrix<float, 6, 6> &E,
437         Matrix<float,6, 1> &F, const Matrix<float, 24, 6> &CC,
438         Matrix<float,24, 1> &d);
439         Matrix<float,6, 1> x;
440         Matrix<float,3, 1> uu;
441         Matrix<float,9, 1> Xf;
442         Matrix<float,3, 1> y;
443         SquareMatrix<float,6> E;
444
445 // Initialisations
446 x.setZero();
447 uu.setZero();
448 y.setZero();
449 Xf.setZero();
450
451 float dataP[135]= {
452         0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,
453         0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,
454         2.0f,0.0f,0.0f,3.0f,0.0f,0.0f,4.0f,0.0f,0.0f,
455         5.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,
456         0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,
457         0.0f,1.0f,0.0f,0.0f,2.0f,0.0f,0.0f,3.0f,0.0f,
458         0.0f,4.0f,0.0f,0.0f,5.0f,0.0f,0.0f,0.0f,0.0f,
459         0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,
460         0.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,2.0f,
```

```
        0.0f,0.0f,3.0f,0.0f,0.0f,4.0f,0.0f,0.0f,5.0f,
        1.0f,0.0f,0.0f,1.0f,0.0f,0.0f,1.0f,0.0f,0.0f,
        1.0f,0.0f,0.0f,1.0f,0.0f,0.0f,0.0f,1.0f,0.0f,
        0.0f,1.0f,0.0f,0.0f,1.0f,0.0f,0.0f,1.0f,0.0f,
        0.0f,1.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,1.0f,
        0.0f,0.0f,1.0f,0.0f,0.0f,1.0f,0.0f,0.0f,1.0f};
Matrix<float,15, 9> P(dataP);

// Output prediction matrix H
float dataH[90] = {
        17.395f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 17.395f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 12.658f, 0.0f, 0.0f, 0.0f,
        34.79f, 0.0f, 0.0f, 17.395f, 0.0f, 0.0f,
        0.0f, 34.79f, 0.0f, 0.0f, 17.395f, 0.0f,
        0.0f, 0.0f, 25.316f, 0.0f, 0.0f, 12.658f,
        52.185f, 0.0f, 0.0f, 34.79f, 0.0f, 0.0f,
        0.0f, 52.185f, 0.0f, 0.0f, 34.79f, 0.0f,
        0.0f, 0.0f, 37.974f, 0.0f, 0.0f, 25.316f,
        69.58f, 0.0f, 0.0f, 52.185f, 0.0f, 0.0f,
        0.0f, 69.58f, 0.0f, 0.0f, 52.185f, 0.0f,
        0.0f, 0.0f, 50.632f, 0.0f, 0.0f, 37.974f,
        86.975f, 0.0f, 0.0f, 69.58f, 0.0f, 0.0f,
        0.0f, 86.975f, 0.0f, 0.0f, 69.58f, 0.0f,
        0.0f, 0.0f, 63.29f, 0.0f, 0.0f, 50.632f};
Matrix<float,15, 6> H(dataH);

// Input weight
float dataW[36] = {
0.055f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.055f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.095f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.055f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.055f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.095f};
Matrix<float,6, 6> W(dataW);

// Transpose of H output matrix
//H_t = H.transpose();
Matrix<float,6, 15> H_trans = H.transpose();
// quadratic programming variable, E
E =  (H_trans*H + W );
E = E * 2;
// Constraint matrix
float dataCC[144] = {1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
```

```
0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
-1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
-1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f,
-1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,
0.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f,
0.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.0f};
Matrix<float,24, 6> CC(dataCC);
// Constraint vector of input and rate of input change
float datadd[24] ={
0.793f,
0.793f,
1.388f,
0.793f,
0.793f,
1.388f,
0.793f,
0.793f,
1.388f,
0.793f,
0.793f,
1.388f,
1.3217f,
1.3217f,
2.31291,
1.3217f,
1.3217f,
2.31291,
1.3217f,
```

```
1.3217f,
2.31291,
1.3217f,
1.3217f,
2.31291};
Matrix<float,24, 1> dd(datadd);
// Past rate of input change matrix
float datadupast[72] ={0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,
-1.0f, 0.0f, 0.0f,
0.0f, -1.0f, 0.0,
0.0f, 0.0f, -1.0f,
-1.0f, 0.0f, 0.0f,
0.0f, -1.0f, 0.0f,
0.0f, 0.0f, -1.0f,
1.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 1.0f,
1.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 1.0f};

Matrix<float,24, 3> dupast(datadupast);
float dataAd[36] = {1.0f, 0.2f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.2f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.2f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f};
// State space matrices
static Matrix<float, 6, 6> Ad(dataAd);

float dataBd[18] = {17.395f, 0.0f, 0.0f,
17.395f, 0.0f, 0.0f,
0.0f, 17.395f, 0.0f,
```

```
0.0f, 17.395f, 0.0f,
0.0f, 0.0f, 12.651f,
0.0f, 0.0f, 12.651f};
static Matrix<float, 6, 3> Bd(dataBd);


float dataCd[18] = {0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f};
static Matrix<float, 3, 6> Cd(dataCd);

// Desired attitude rates/ rates setpoint
float datades[3] = {_rates_sp(0),_rates_sp(1),_rates_sp(2)};
// Reference adjusting matrix
Matrix<float,3, 1> des(datades);

float dataRs[45] ={ 1, 0, 0,
0, 1, 0,
0, 0, 1,
1, 0, 0,
0, 1, 0,
0, 0, 1,
1, 0, 0,
0, 1, 0,
0, 0, 1,
1, 0, 0,
0, 1, 0,
0, 0, 1,
1, 0, 0,
0, 1, 0,
0, 0, 1};
static Matrix<float, 15, 3> Rs(dataRs);
// disturbance

float datadist[3] = {1,1.1,0.9};
Matrix<float,3, 1> dist(datadist);
dist = dist*0.06;
// current rates

float datax_curr[3]={_rates_sp(0),
_rates_sp(1),
_rates_sp(2)};

Matrix<float,3, 1> x_curr(datax_curr);
// Defining previous state vector
```

```
641
642 float datax_prev[6] = {0,
643         _rates_sp(0),
644         0,
645         _rates_sp(1),
646         0,
647         _rates_sp(2)};
648 Matrix<float,6, 1> x_prev(datax_prev);
649
650 int i = 0;
651 do{
652 Matrix<float, 6, 1> F =  (H_trans)*(Rs*des - P*Xf);
653 F = F * -2;
654 //Matrix<float, 24, 1> d = dd + dupast*uu;
655 ////////////////////Function starts here////////////////////////////////
656
657 static SquareMatrix<float, 6> E_cholesky = cholesky(E);
658 static Matrix<float, 6, 24> CC_transd = CC.transpose();
659 LeastSquaresSolver<float, 6, 6> qrd = LeastSquaresSolver<float, 6, 6>(
       E_cholesky);
660
661 Matrix<float, 24, 1> K = (CC*(qrd.solve(F)) + d);
662
663 int k_row = 24;
664 Matrix<float, 24, 1> lambda;
665 lambda.setZero();
666 float al = 3.0f;
667 int km = 0;
668
669 do
670 {
671 Matrix<float, 24, 1> lambda_p = lambda;
672 // loop to determine lambda values for respective iterations
673 int j = 0;
674 do
675 {
676 /// solved upto this
677 float Tjj = T(j, j);
678 T(j, j) = 0;
679 float la = -(T.col(j).dot(lambda) + K(j)) / Tjj;
680 T(j, j) = Tjj;
681 if (la < 0.0f) lambda(j) = 0.0f;
682 else lambda(j) = la;
683 j++;
684 } while (j < k_row);
```

```
al = (lambda - lambda_p).norm_squared();

if (al < 0.001f) break;
km++;
} while (km < 15);

Matrix<float, 6, 1> DeltaU = -qrd.solve(F); (qrd.solve(CC_transd))*lambda;

//////////////////Function ends here//////////////////////////
float DeltaU_1data[6] = {DeltaU(0, 0), DeltaU(1, 0), DeltaU(2, 0),
DeltaU(3, 0), DeltaU(4, 0), DeltaU(5, 0)};
Matrix<float, 2, 3> DeltaU_1(DeltaU_1data);
Matrix<float, 1, 3> newDeltaU_1(DeltaU_1.row(0));
Matrix<float, 3, 1>deltau_tran=newDeltaU_1.transpose();
//Matrix<float, 3, 1>deltau_tran=(DeltaU_1.row(0)).transpose();
uu = uu + deltau_tran;
x = Ad*x_prev + Bd*uu;
y = Cd*x;//+ dist;
Matrix<float, 6, 1> xx = (x-x_prev);
Xf(1,0) = xx(1,0);Xf(2,0) = xx(0,0);Xf(3,0) = xx(3,0);
Xf(4,0) = xx(4,0);Xf(5,0) = xx(0,0);Xf(6,0) = xx(6,0);
Xf(7,0) = y(1,0); Xf(8,0) = y(2,0);Xf(9,0) = y(3,0);
x_prev = x;
i++;
} while(i < 2);
Matrix<float,3,1> _att_control;
_att_control(0,0) = uu(0,0);
_att_control(1,0) = uu(1,0);
_att_control(2,0) = uu(2,0);
float umaxx_data[3] = {1.258f, 1.258f, 0.2147f};
Vector<float,3> umaxx(umaxx_data);

Vector<float,3> uminn = -umaxx;

for (int k = 0; k < 3; k++)
{
_att_control(k,0) = (2 * ((_att_control(k,0) - uminn(k))
/(umaxx(k) - uminn(k)))-1);
}
}
////Solved upto this point
/////////////////////////////////////////

extern "C" __EXPORT int mc_att_control_main(int argc, char *argv[])
```

```
{
        return MulticopterAttitudeControl::main(argc, argv);
}
//int mc_att_control_main(int argc, char *argv[])
//{
        //return MulticopterAttitudeControl::main(argc, argv);
//}
mc_att_control_main.cpp
Displaying mc_att_control_main.cpp.;
}
```

C.1.2. CMakeLists.txt

This section contains CMake file of attitude control module,which is required for MPC attitude control.

```
{
############################################################################
#
#   Copyright (c) 2015-2019 PX4 Development Team. All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# 1. Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
# 2. Redistributions in binary form must reproduce the above copyright
#    notice, this list of conditions and the following disclaimer in
#    the documentation and/or other materials provided with the
#    distribution.
# 3. Neither the name PX4 nor the names of its contributors may be
#    used to endorse or promote products derived from this software
#    without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
# OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
# AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
```

```
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
############################################################################

add_subdirectory(AttitudeControl)
add_link_options(--stack=1000000) # 1MB stack size
# Add compiler flags
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-frame-larger-than")

# Add linker options (if necessary)
# set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,--stack,0x2000
   ")

px4_add_module(
        MODULE modules__mc_att_control
        MAIN mc_att_control
        COMPILE_FLAGS
                ${MAX_CUSTOM_OPT_LEVEL}
                -Wframe-larger-than=4096
        SRCS
                mc_att_control_main.cpp
                mc_att_control.hpp
        DEPENDS
                AttitudeControl
                mathlib
                px4_work_queue
        )

}
```