Design and Implementation of Model Predictive Control on Pixhawk Flight Controller

by
Chinedu Amata Amadi

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Mechatronic) in the Faculty of Engineering at Stellenbosch University

UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

Supervisor: Dr. Willie. J. Smit

December 2018

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2018

Plagiarism Declaration

1. Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.
2. I agree that plagiarism is a punishable offence because it constitutes theft.
3. I also understand that direct translations are plagiarism.
4. Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
5. I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

18899293 Student number	 Signature
CA. Amadi Initials and surname	6/09/2018 Date

Abstract

Quadcopters have undergone a steady rise in popularity in the last decade. They have been adopted in the military, fire and rescue missions, security systems and photography, just to list a few. The rate of adoption of quadcopters is on the rise as more applications for their use are discovered. At the Solar Thermal Energy Research Group (STERG), Stellenbosch University, quadcopters are used in the calibration and inspection of heliostats and to improve point focusing of the heliostats. Therefore, it is necessary to use quadcopters with excellent performance to achieve these objectives.

STERG uses the Pixhawk autopilot, one of the most popular open source flight controllers available, for quadcopter research. The Pixhawk runs on the PX4 firmware comprised of modules used for state estimation, position- and angular control and others. A Proportional Derivative (PD) controller is implemented on the PX4 firmware to control the angular rates of a quadcopter. However, previous studies show that this controller is inadequate and necessitates a need for an alternative. Model Predictive Control (MPC) was chosen as the alternative, due to its ability to generate a sequence of inputs needed to control a system by minimising the error between reference values and predicted outputs and also its ability to handle constraints. Nevertheless, MPC has not been implemented on the PX4 firmware before, as it requires a mathematical model of the specific quadcopter to be used. Thus, the aim of this thesis is to evaluate the feasibility of implementing MPC on the Pixhawk, running the PX4 firmware, to control the angular rates of a quadcopter.

The MPC angular rates controller was designed and implemented in MATLAB. The controller was then programmed in C++ for compatible inclusion in the relevant PX4 module. A multicopter simulator was used to run the modified PX4 firmware on a simulated quadcopter to control its angular rates. Subsequently, the PX4 firmware was uploaded onto the Pixhawk. Several challenges were encountered in this stage, with the most prominent, being the size of the memory on the Pixhawk. Measures such as code optimisation, stack size adjustment and disabling unused modules were necessary to ensure a successful firmware upload. A quadcopter running the modified PX4 firmware on the Pixhawk was flight tested and thereafter, the angular rates flight data was plotted and analysed. The plots show that the MPC angular rates controller is able to achieve close reference tracking of angular rates.

The findings from this novel approach demonstrate the feasibility of implementing model predictive control on the PX4 firmware, and proposes using a Pixhawk with a larger memory in order to integrate MPC into other PX4 control modules.

Uittreksel

Hommeltuie het die afgelope dekade meer gewildheid geraak. Hulle word gebruik in die weermag, brand- en reddingsmissies, sekuriteitsisteme en fotografie, om net 'n paar te noem. Hommeltuie word al meer gebruik soos meer toepassings vir hul gebruik ontdek word. By die Son Termiese Energie Navorsingsgroep (STERG), Universiteit Stellenbosch, word hommeltuie gebruik in die kalibrasie en inspeksie van heliostate en om die puntfokus van die heliostate te verbeter. Daarom is dit nodig om hommeltuie te gebruik met uitnemende prestasie om hierdie doelwitte te bereik.

STERG gebruik die Pixhawk outoloots, een van die gewildste oopbron vlugbeheerders beskikbaar vir hommeltuignavorsing. Die Pixhawk gebruik die PX4 fermatuur wat bestaan uit modules vir toestandafskatting, posisie- en hoekbeheer en ander. 'n Proporsionele afgeleide (PD) beheerder is ge´implementeer op die PX4-fermatuur om die hoektempo's van die hommeltuig te beheer. Vorige studies toon egter dat hierdie beheerder ontoereikend is en 'n alternative beheerder noodsaak. Modelvoorspellende beheer (MVB) is gekies as 'n alternatief, omdat dit 'n reeks intree-waardes kan genereer om 'n stelsel te beheer deur die fout tussen die verwysingswaardes en die voorspelde uitree te minimeer, asook omdat dit die vermoë het om begrensings te hanteer. Nietemin is die MVB nog nie voorheen op die PX4-fermatuur geïmplementeer nie, aangesien dit 'n wiskundige model benodig van die spesifieke hommeltuig wat gebruik word. Dus, die doel van hierdie proefskrif is om die uitvoerbaarheid van MVB wat uitvoer op 'n Pixhawk met PX4-fermatuur te evalueer, waar die MVB die hoektempo van 'n hommeltuig beheer.

Die MBV-hoektempo beheerder is ontwerp en geïmplementeer in MATLAB. Die beheerder is in C++ geprogrammeer vir aanpasbare insluiting in die betrokke PX4-module. 'n Hommeltuig-simulator is gebruik om die gewysigde PX4-fermatuur op 'n gesimuleerde hommeltuig uit te voer en so die hoektempo te beheer. Daarna is die PX4-fermatuur op die Pixhawk gelaai. Verskeie uitdagings is in hierdie stadium ondervind, die mees prominente was die grootte van Pixhawk se geheue. Maatreëls soos kodeoptimering, stapelgrootte aanpassing en om ongebruikte modules af te skakel was nodig om 'n suksesvolle fermatuuroplaai te verseker. 'n Hommeltuig wat die gewysigde PX4-fermatuur op die Pixhawk uitvoer, is in vlug getoets en daarna is die hoektempo vlugdata geteken en ontleed. Die plotte toon dat die MVB-hoektempo beheerder in staat

is om die verwysing vir hoektempo's goed te volg.

Die bevindinge van hierdie nuwe benadering demonstreer die haalbaarheid om modelvoorspellende beheer op die PX4 fermatuur te ïmplementeer, en stel voor dat 'n Pixhawk met 'n groter geheue gebruik moet word om MVB te integreer met ander PX4 beheer modules.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Dr. Willie Smit for his guidance and support during my research. Our meetings were always productive and either resulted in suggestions towards my research, advice on my personal affairs or just general discussions that I found uplifting. He is an amazing mentor.

I would also like to thank the Solar Thermal Energy Research Group (STERG) for the research office space and other facilities and resources that aided me in my research.

I want to acknowledge the Centre for Renewable and Sustainable Energy Studies (CRSES) for their financial support during the second half of the second year of my programme.

My smooth adoption of LaTeXwould not have been possible without the guidance of Abbas M. Sherif (PhD in Mathematics candidate at University of Kwazulu-Natal, Master of Science in Theoretical Physics from Stellenbosch University). His advice on research and general encouragement were invaluable during this journey.

Finally, but by no means the least, I want to thank my family and friends for their unwavering support — emotionally, spiritually and financially — during the course of my programme. I am grateful for being blessed which such incredible people in my life.

To Amata and Chiaka Amadi

Contents

List of Figures

List of Tables

Nomenclature

Roman Letters

A	State matrix
B	Input matrix
b	Thrust coefficient
C	Output matrix
D	Feedforward matrix
d	Moment arm
g	Acceleration due to gravity
I	Moment of inertia matrix
I_{xx}	Moment of inertia about x-axis
I_{yy}	Moment of inertia about y-axis
I_{zz}	Moment of inertia about z-axis
k	Aerodrag coefficient
m	Mass
n	Number of states
nu	Control horizon
ny	Prediction horizon
R	Rotation matrix
u	Input vector
W	Earth-fixed frame
x	State vector
y	Output vector

Greek Letters

B	Body-fixed frame
η	Angular velocity vector about earth-fixed axis
θ	Pitch angle about y body axis
$\dot{\theta}$	Angular velocity about y body-fixed axis

$\ddot{\theta}$	Angular acceleration about y body-fixed axis
ν	Angular velocity vector about body-fixed axis
τ	Torque acting in body-fixed frame
ϕ	Roll angle about x body-fixed axis
$\dot{\phi}$	Angular velocity about x body-fixed axis
$\ddot{\phi}$	Angular acceleration about x body axis
ψ	Yaw angle about z body-fixed axis
$\dot{\psi}$	Angular velocity about z body-fixed axis
$\ddot{\psi}$	Angular acceleration about z body-fixed axis
ω_i	Motor speed for each motor

Subscripts

m	Model
p	Prediction
xx	X axis
yy	Y axis
zz	Z axis

Abbreviations

ASM	Active Set Method
ESC	Electronic Speed Controller
EKF	Extended Kalman Filter
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPU	Floating Point Units
FTDI	Future Technology Devices International
GCS	Ground Control Station
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
IPM	Interior Point Method
KKT	Karush Kuhn and Tucker
LQR	Linear Quadratic Regulator
LTI	Linear Time Invariant
MATLAB	Matrix Laboratory
MIMO	Multiple Input Multiple Output
MPC	Model Predictive Control
NED	North East Down

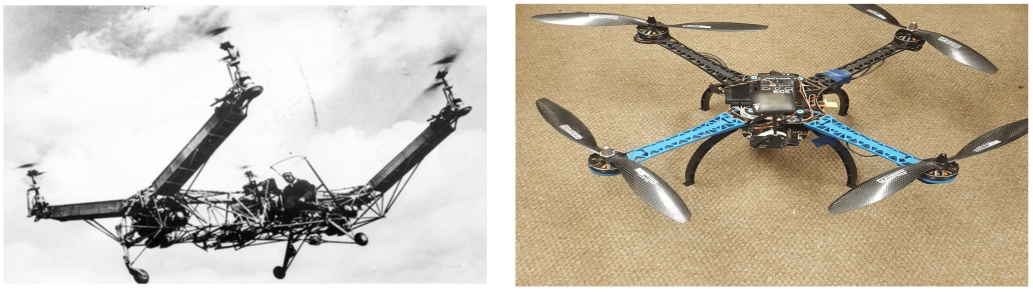
PCM	Pulse Control Modulation
PD	Proportional Derivative
PI	Proportional Integral
PID	Proportial Integral Derivative
PPM	Pulse Position Modulation
PWM	Pulse Width Modulation
QP	Quadratic Program
RC	Remote Control
SBUS	Serial Bus
SISO	Single Input Single Output
SITL	Software In The Loop
STERG	Solar Thermal Energy Research Group
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
VTOL	Vertical Take Off and Landing

Chapter 1

Introduction

1.1 Background

In the last three decades, significant advancements in miniaturisation of mechanical and electronic devices (Hsu, 2002) have led to the reemergence of multicopters. The preceding multicopters were large, complex, unstable and required a highly skilled pilot to operate them. The modern multicopters are significantly smaller, simpler, more reliable and maneuvarable (Villbrandt, 2011). Multicopters are classified as Unmanned Aerial Vehicles (UAVs) because they are either controlled remotely by a radio transmitter or from a computer ground station. Multicopters equipped with four rotors are called **quadcopters** or **quadrotors**. Figure 1.1 shows the 1956 Convetawing Quadrotor in the left panel and a typical modern quadcopter in the right panel.



(a) (b)

Figure 1.1: (a) Convertawing Quadrotor, 1956 (San Diego Air and Space Museum Archives) (b) Quadcopter (STERG, Stellenbosch University)

Currently quadcopters are being used in the military and defence industries, search and rescue operations, power plant inspections, logistics and freight, transportation, surveillance, livestock monitoring, aerial imagery, wild fire monitoring, and photography to list a few (Luukkonen, 2011). This rise in

potential sectors for the application of quadcopters motivates research interests to improve and address current challenges quadcopters encounter. Some of these challenges include obstacle avoidance for autonomous navigation, increasing quadcopter flight time and improving the control system to reject disturbances and track reference commands.

There are two main components required in controlling a quadcopter: a microcontroller, also termed as the flight controller, and sensors connected to the microcontroller. The microcontroller runs a control algorithm that uses data from the sensors and commands sent from a radio transmitter or groundstation to control the quadcopter by varying the speed of the motors (Leong *et al.*, 2012). The microcontroller has to compute control commands very quickly in order to keep the quadcopter airborne; consequently, the algorithm running on the microcontroller has to be fast.

There are various flight controllers commercially available that differ according to the specific use of the quadcopter. All flight controllers fall under two categories: closed and open source flight controllers. Open source flight controllers enable the user to make modifications to both the hardware and software of the flight controller, while closed source flight controllers do not. This freedom of controller customisation is one of the main reasons open source flight controllers are used. Some of the popular open source flight controllers are the Arducopter, Openpilot, Paparazzi, Pixhawk, Mikrokopter, Kkmulticopter, Multiwii and Aerocopter (Lim *et al.*, 2012). Figure 1.2 below only shows the Arducopter, Openpilot, Paparazzi and Pixhawk flight controllers.

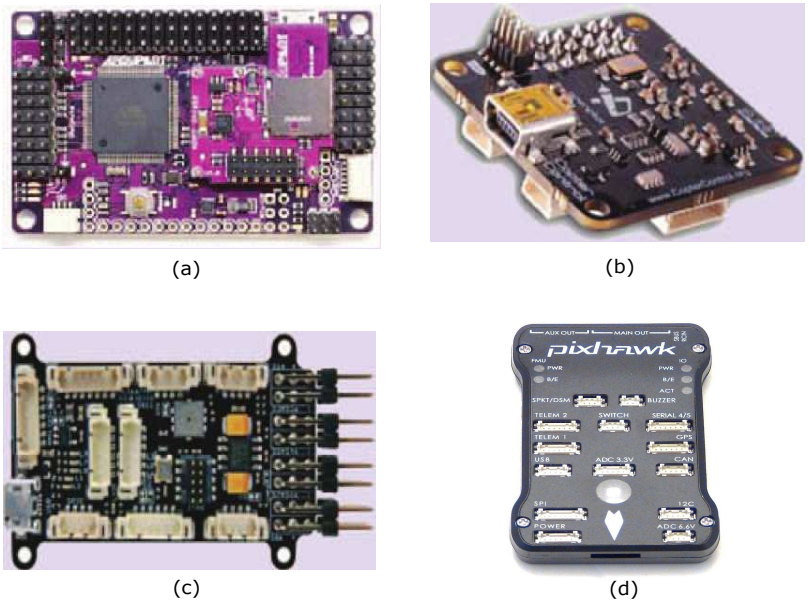


Figure 1.2: (a) Arducopter (b) Openpilot (c) Paparazzi (d) Pixhawk autopilot (Adapted from Lim *et al.* (2012))

On this list, the Pixhawk flight controller is considered to be one of the most versatile and advanced flight controllers. In addition to being used as a flight controller for quadcopter control, the Pixhawk can be utilised in controlling other multicopters, fixed-winged aircraft and Unmanned Ground Vehicles (UGVs) like rovers. Autonomous control of a marine vehicle was realised by Risqi *et al.* using the Pixhawk. With a large community of developers and users, the PX4 firmware that runs on the Pixhawk is constantly being improved upon to release more stable versions for the respective vehicles.

At the Solar Thermal Energy Research Group (STERG) at Stellenbosch University, the Pixhawk autopilot is used in quadcopter research. Quadcopters are used in the calibration and inspection of heliostats, and to improve point focusing of the heliostats. Currently the group utilises the default PX4 firmware running on the Pixhawk to control position and attitude. The attitude controller incorporates a P controller for angular error and Proportional Derivative (PD) controller for angular rate error.

The use of Proportional Integral Derivative (PID) controllers and its variants (such as a Proportional Integral (PI), PD or proportional controllers) ensures simplicity and flexibilty in control. Thus, a solid background in control theory is not essential for users of the Pixhawk. However, the results obtained from quadcopter research at STERG strongly indicate that these methods prove to be inadequate especially in controlling the angular rates (or velocity) of the quadcopter. Therefore, a more advanced control method is required.

Advanced control techniques have been implemented on quadcopters, and there have been a number of quadcopter controller comparisons between more advanced controllers and the PID controller. These advanced controllers are either implemented on a quadcopter, or simulated, and have proven to perform better than the PID controller or any of its variants. In Bhatkhande and Havens (2014), a Fuzzy logic controller is implemented to control a quadcopter. The controller is compared with a PD and it was confirmed that the Fuzzy logic controller performed better. Tosun *et al.* (2015) achieved better attitude control using Linear Quadratic Regulator (LQR) controller in comparison to a PID controller. Adaptive control is implemented by Palunko and Fierro (2011) in investigating stability when there are dynamic changes in the center of gravity of the quadcopter. A PD controller failed to achieve stability under such conditions whereas the adaptive controller did. In Ganga and Dharmana (2017), a PID controller is used as a benchmark to assess its trajectory tracking ability in comparison with a Model Predictive Controller (MPC). Simulations were run which demonstrated that the MPC performed better than the PID controller. Other controllers that have been implemented on a quadcopter but only these ones have been considered for brevity.

1.2 Research Problem

As established in the previous section, the necessity for an enhanced control method to control the angular rates of the quadcopter is beneficial to researchers at STERG in calibrating and inspecting heliostats and improving the point focusing of the heliostats. The control method chosen as the replacement to the default PX4 angular rates controller is Model Predictive Control (MPC).

Model predictive control uses a mathematical model of a system to predict its output (response) over a prediction horizon. A cost function comprised of the error between these predicted outputs and reference values and a weighted control input term are minimised to generate a sequence of control inputs to be sent to the system to track the reference. The length of the sequence is equivalent to a predetermined parameter known as the control horizon, however only the first element in this sequence is implemented on the system, as reference and sensors data are updated in subsequent sample times.

Model predictive control has yet to be implemented on the PX4 firmware as it requires a mathematical model of the quadcopter, as the Pixhawk can be used on vehicles of different types and sizes.

1.3 Aim of Thesis

The aim of this thesis is to evaluate the feasibility of implementing a model predictive controller on the Pixhawk flight controller to control the angular rates of a quadcopter.

1.4 Objectives

The objectives of this thesis are:

1. To design and simulate a model predictive controller in MATLAB.
2. To program the controller in C++ and simulate its performance with a software-in-the-loop (SITL) simulator compatible with the Pixhawk.
3. To assess the feasibility of running the controller on the Pixhawk hardware.
4. To conduct flight tests if implementing the MPC controller on the Pixhawk was realised.

1.5 Thesis Outline

Chapter 2 is the literature review which introduces the quadcopter structure and its working principle. Coordinate frames required in mathematical modelling are covered in a section. Quadcopter dynamics and mathematical modelling are reviewed in the subsequent section. A state space representation of the quadcopter mathematical model is presented subsequently. In the following section, MPC is introduced and explained in detail. The section that follows briefly explains the state observer used in determining the states of the quadcopter that are needed for rates control of the quadcopter. A concise discussion on the history of the Pixhawk hardware and a description of its firmware architecture concludes this chapter.

Chapter 3 covers the methodology of designing and implementing the MPC angular rates controller first in MATLAB, then for SITL and finally on the Pixhawk autopilot for flight testing. The steps carried out in the methodology are visualised in the figure below.

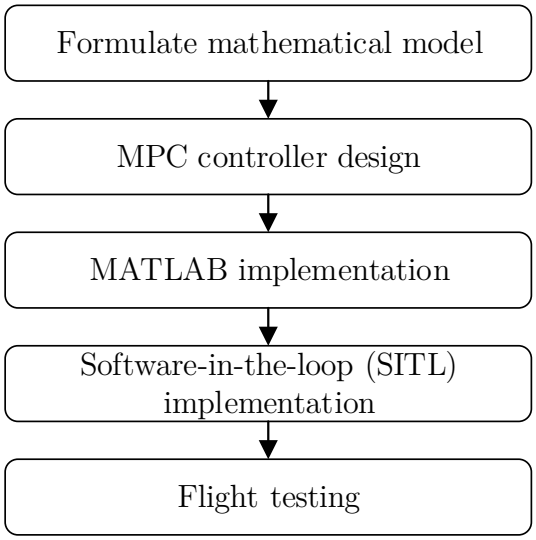


Figure 1.3: Methodology

The simulations obtained from running the MPC angular rates controller in MATLAB and SITL are presented and discussed in chapter 4. This chapter concludes with the presenting and analysing the results obtained from the implementing the controller on the Pixhawk for outdoor flight testing.

Chapter 5 concludes the thesis with a discussion of the significance of the results obtained, challenges faced and recommendations for future work.

In the Appendices, appendix A presents an example to better explain parts of the model predictive controller. The Pixhawk specifications are listed in appendix B. Appendix C details the procedures employed in determining the

physical parameters of the quadcopter such as its mass, mass moment of inertias for each cartesian axis and the thrust and drag motor coefficients. Appendix D shows the MATLAB code used in designing an MPC controller. The C++ code used in SITL and flight testing is provided in appendix E. In appendix F, the commands used in setting up the the simulator used for SITL implementation are outlined. Finally, additional results obtained from MATLAB and SITL simulations, and flight testing are presented in appendix G.

Chapter 2

Literature Review

A detailed description of the quadcopter, its structure, its mode of operation and onboard sensors are presented in section 2.1. Section 2.2 defines the coordinate frames required to describe the translational and rotational motion of the quadcopter. Section 2.3 covers the derivation of the equations of motion that describe the rotational dynamics of the quadcopter and its mathematical model. The derived mathematical model of the quadcopter is represented in state-space form in section 2.4. In section 2.5, a brief history of Model Predictive Control (MPC) is given, followed by the principle of MPC and a detailed description of its components. This section is concluded with the motivation for choosing this control method. Section 2.6 presents the state observer used by the Pixhawk to estimate the states of the quadcopter. Finally in section 2.7, the brief history of the Pixhawk autopilot and a description of its software architecture are presented.

2.1 Quadcopter Structure

A quadcopter is a multirotor Unmanned Aerial Vehicle (UAV) with four rotors, which are varied in speed to change the altitude and/or angular position of the vehicle (Thorat, 2015).

Quadcopters are configured with propellers that are connected either in an "X" or "+" configuration (Giernacki *et al.*, 2017) as shown in figure 2.1 below; each propeller is powered by an electric motor. The propellers are connected in counter-rotating pairs in order to cancel out the torques created by a set of propellers rotating in one direction (Patel and Barve, 2014). In figure 2.1, the propellers labeled 1 and 2 rotate in a counter-clockwise direction, while the propellers labeled 3 and 4 rotate in a clockwise direction.

With each component of the quadcopter securely fastened, ensuring that no component moves relative to another, the quadcopter can be classified as a rigid body which has six degrees of freedom (Greenwood, 2003). Three of the degrees of freedom are transitional, typically represented by cartesian

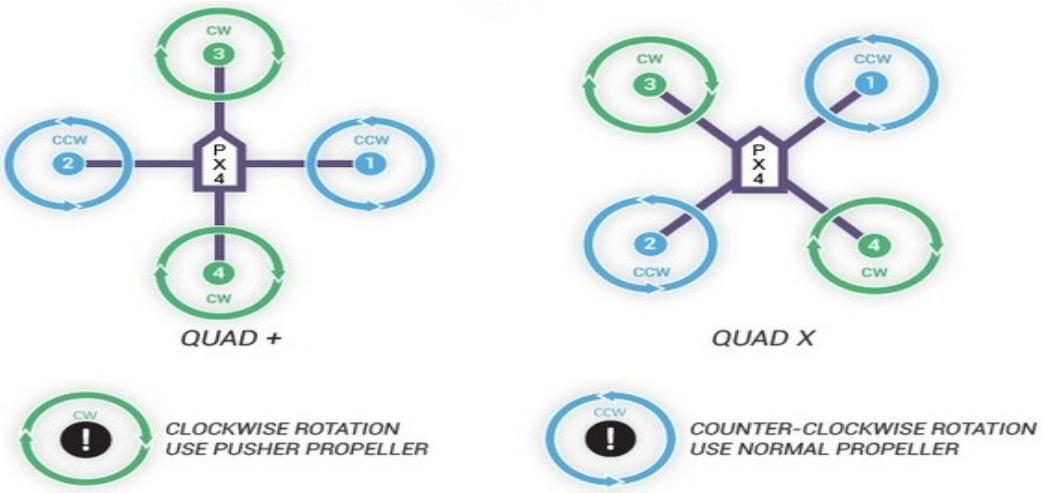


Figure 2.1: Quadcopter configuration (Ardupilot, 2016)

coordinates, the other three are rotational and are typically represented by euler angles.

The six degrees of freedom are illustrated in fig 2.2 below.

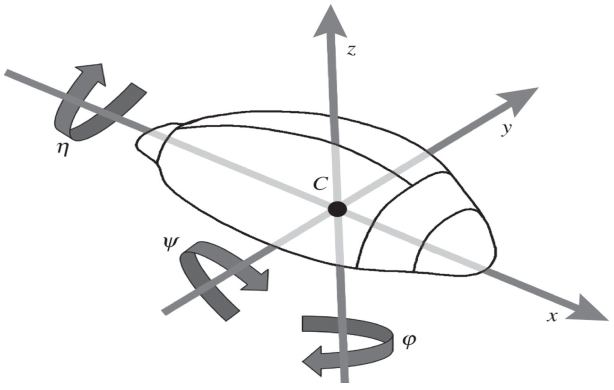


Figure 2.2: Six degrees of freedom of a rigid body (Voise *et al.*, 2011)

The quadcopter is underactuated as it possesses only four motors for a six degree of freedom rigid body (Magnussen and Skjønhaug, 2011). This necessitates the need for a control system to compensate for this underactuation, in order to stabilize the quadcopter when it is airborne.

As stated in the introductory chapter, the microcontroller to be used to implement the MPC angular rates controller is the Pixhawk autopilot. The Pixhawk is embedded with an accelerometer, gyroscope, barometer and magnetometer.

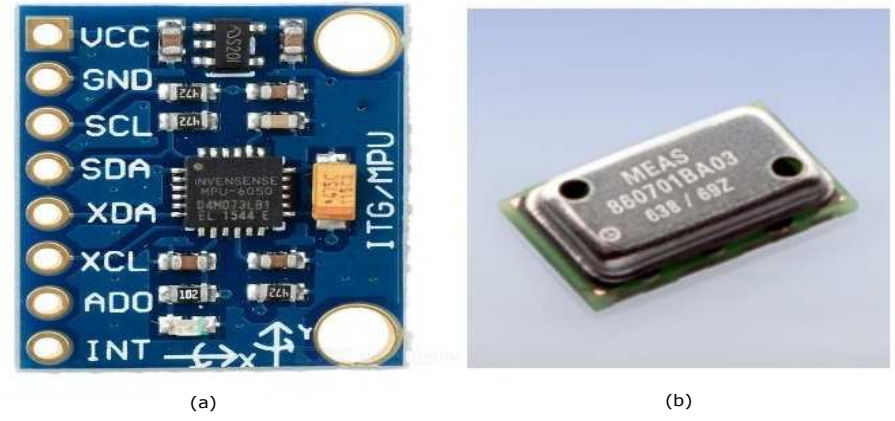
A gyroscope is a device that senses and measures the angular velocity of a body or system. There are gyroscopes that measure angular velocity in one, two or three orthogonal axes. The gyroscope on the Pixhawk is a 3-axis

gyroscope measuring the angular velocity of the quadcopter in the x (roll), y (pitch) and z (yaw) axes.

Accelerometers are electromechanical devices that measure acceleration. Similar to the gyroscope, the acceleration of a body or system can be measured in either one, two or three orthogonal axes.

A 3-axis gyroscope can be coupled with a 3-axis accelerometer to provide a full six degree of freedom motion tracking system. This coupled device is known as an **Inertial Measurement Unit (IMU)** (Kim and Golnaraghi, 2004). The IMU embedded in the Pixhawk is the MPU6050 and is shown in figure 2.3(a) below.

A barometer is a device used to measure atmospheric pressure. Atmospheric pressure decreases with increasing altitude (Cavcar, 2000). This relation is used to determine the altitude of the body or system equipped with a barometer. The MEAS barometer is used in the Pixhawk and is displayed in figure 2.3(b) below.



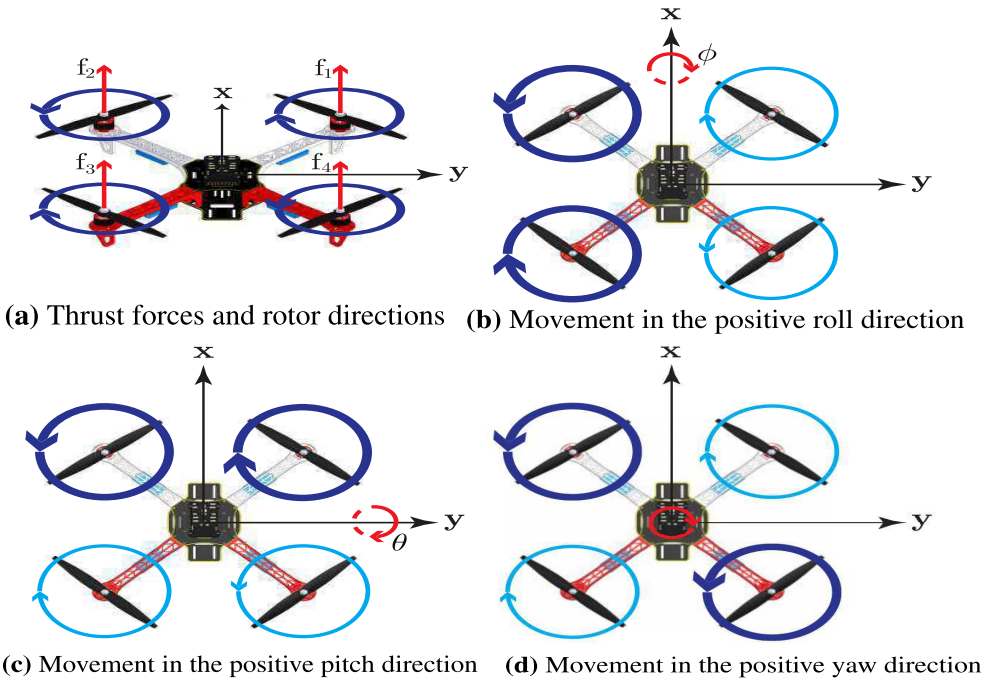
(a) (b)

Figure 2.3: (a) MPU6050 Inertial Measurement Unit (IMU) (b) MEAS barometer

Magnetometers are devices used to measure the ambient magnetic field surrounding them. Magnetometers are used in quadcopters to determine its spatial orientation by providing readings in three orthogonal axes.

The sensors above provide the microcontroller with data about the current position and rotational orientation of the quadcopter. A radio transmitter (or remote controller) is used to send desired position and orientation commands to the microcontroller via a radio receiver. The error between the desired and current position and orientation is sent as an input to a control algorithm running on the microcontroller. The control algorithm outputs motor pulse width signals that are sent to the four electronic speed controllers (ESCs) which convert these signals to power settings to control the speed of each motor in order to drive the current position and orientation of the quadcopter to what is desired (Quan, 2017).

Figure 2.4 displays the different movements that can be achieved by varying the speed of specific motors on the quadcopter, where the thickness of the arrows around the propeller corresponds to the speed of the respective motor. The mode of operation of the quadcopter is shown in figure 2.5.



(a) Thrust forces and rotor directions **(b)** Movement in the positive roll direction

(c) Movement in the positive pitch direction **(d)** Movement in the positive yaw direction

Figure 2.4: Quadcopter movements (Adapted from Grujic and Nilsson (2016))

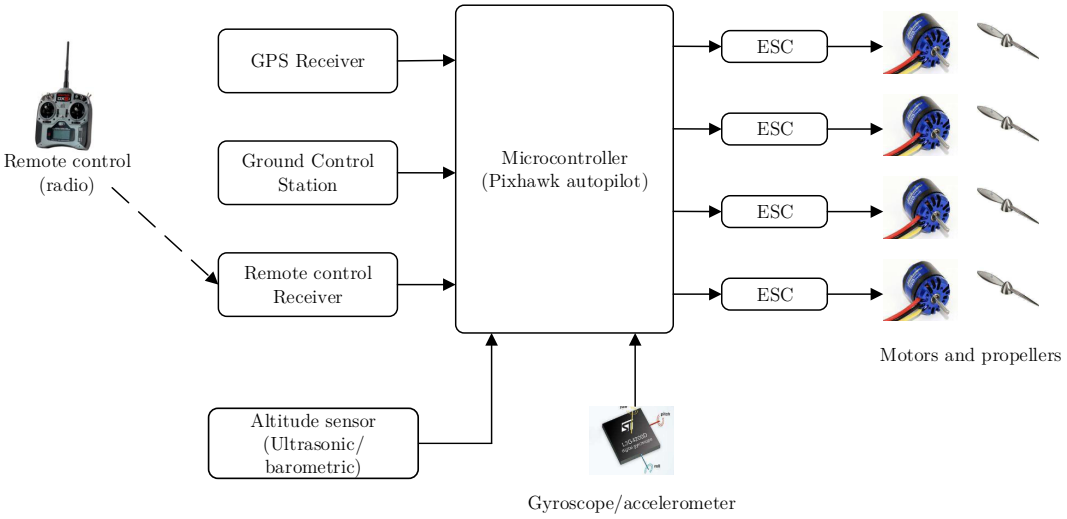


Figure 2.5: Quadcopter mode of operation (Adapted from Santoro, 2014)

2.2 Coordinate Frames

Coordinate frames are needed to describe the motions of the quadcopter before the quadcopter is mathematically modeled. Two coordinate frames are used to achieve this purpose and these are the earth-fixed frame, $\boldsymbol{W}$, and body-fixed frame, $\boldsymbol{B}$. With two coordinate frames, parameters that are measured or observed in the earth-fixed frame can be related in the body-fixed frame and vice versa. These coordinate frames are illustrated in figure 2.6 with the quadcopter in the body-fixed frame.

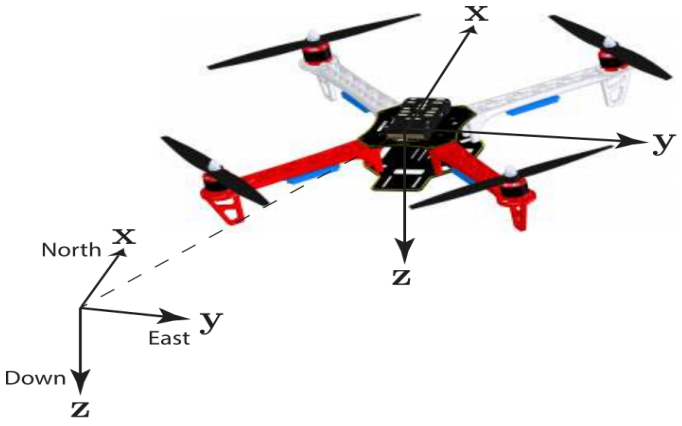


Figure 2.6: Earth-fixed and body-fixed coordinate frames (Adapted from Grujic and Nilsson (2016))

The earth-fixed frame is taken as the reference frame using the NED (North East Down) convention where the x-axis of the frame is pointed to the north, y-axis pointed to the east and the z-axis pointed down (Nebylov and Watson, 2016). The body-fixed frame, $\boldsymbol{B}$, has its origin at the center of mass of the quadcopter. The orientation of the quadcopter, known as its **attitude**, is expressed in the body-fixed frame by euler angles $\boldsymbol{\phi}$, $\boldsymbol{\theta}$ and $\boldsymbol{\psi}$ which correspond to the roll, pitch and yaw angles. The angles are also referred to as the angular positions of the quadcopter and are shown in fig 2.7 below.

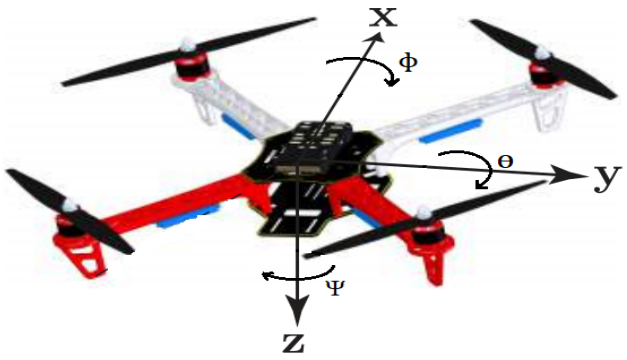


Figure 2.7: Quadcopter euler angles (Adapted from Grujic and Nilsson (2016))

In order to relate the orientation of the quadcopter in the earth-fixed frame, a rotation matrix, $\boldsymbol{R}$, is required. The rotation matrix, $\boldsymbol{R}$ from the body-fixed frame to the earth-fixed frame is given as:

$$\boldsymbol{R} = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \tag{2.2.1}$$

where $S_\phi = \sin(\phi)$ and $C_\phi = \cos(\phi)$. Equation 2.2.1 was obtained from (Carrillo *et al.*, 2013) by taking the rotation order roll, pitch then yaw.

The IMU on the Pixhawk is used to obtain the angular velocity of the quadcopter in the body-fixed frame. A transformation is needed to relate euler rates, $\nu = \begin{bmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T$, that are measured in the earth-fixed frame and the angular body-fixed rates, $\eta = \begin{bmatrix} p & q & r \end{bmatrix}^T$. The transformation is obtained from Luukkonen (2011) and is as follows,

$$\eta = W_\eta^{-1}\nu, \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & S_\phi T_\theta & C_\phi T_\theta \\ 0 & C_\phi & -S_\phi \\ 0 & S_\phi / C_\theta & C_\phi / C_\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.2.2}$$

where W_η is the transformation matrix and $T_x = tan(x)$

Subsequently transforming angular velocities in the body-fixed frame to the earth-fixed frame is achieved by the equation below,

$$\nu = W_\eta \eta, \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & S_\phi C_\theta \\ 0 & -S_\phi & C_\phi C_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{2.2.3}$$

The relations in equations 2.2.2 and 2.2.3 are obtained from Alderete (1995).

Taking small angle assumptions, the transformation matrix can be reduced to an identity matrix (Nagaty *et al.*, 2013). Therefore,

$$\nu = \eta \tag{2.2.4}$$

2.3 Quadcopter Dynamics

This thesis is focused on designing and implementing a model predictive controller on a Pixhawk, running the PX4 firmware, to control the angular rates of a quadcopter. Therefore, only the rotational dynamics of the quadcopter are taken into consideration. The following assumptions were made before a mathematical model was derived from the quadcopter's dynamics (Habib *et al.*, 2014):

- The rotational motion of the quadcopter is independent of its translational motion.
- The centre of gravity coincides with the origin of the body-fixed frame.
- The structure of the quadcopter is rigid and symmetrical with the four arms coinciding with the body x- and y-axes.
- Drag and thrust forces are proportional to the square of the propeller's speed.
- The propellers are rigid.

2.3.1 Rotational Equations of Motion

The rotational motion of the quadcopter is described by Euler's equation of rotation in the equation 2.3.1 below.

$$I\dot{\nu} + \nu \times I\nu = \tau \tag{2.3.1}$$

where,
I - inertia matrix of quadcopter
ν - angular velocity vector in body-fixed frame
τ - torque/moment vector acting on the quadcopter in the body-fixed frame

The angular velocites in the body-fixed frame are put into vector ν as shown below,

$$\nu = \begin{bmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T \tag{2.3.2}$$

The symmetrical structure of the quadcopter reduces the inertia matrix, I, to a diagonal matrix where the diagonal elements correspond to the mass moment of inertia for each cartesian axis; the off-diagonal elements are products of inertia. The inertia matrix is shown below,

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.3.3}$$

The elements of the torque vector, τ, represent the torques about the respective body-fixed frame axes and is expressed in equation 2.3.4 below (Luukkonen, 2011).

$$\tau = \begin{bmatrix} db(\omega_4^2 - \omega_2^2) \\ db(\omega_1^2 - \omega_3^2) \\ k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2) \end{bmatrix} \tag{2.3.4}$$

where,
b - thrust coefficient
k - aerodrag coefficient
d - moment arm
ω - motor speed

The torque τ_ϕ, results in a rolling torque or moment about the x-axis by varying the speeds of the second and fourth motors labeled in the figure 2.8.

Similarly a pitch torque τ_θ , moment about the y-axis, is achieved by varying the speeds of the first and third motors. In achieving a yaw torque τ_ψ, the first and third motor are paired together while the second and fourth are paired together. Varying the speeds for each motor pair results in rotation about the z-axis.

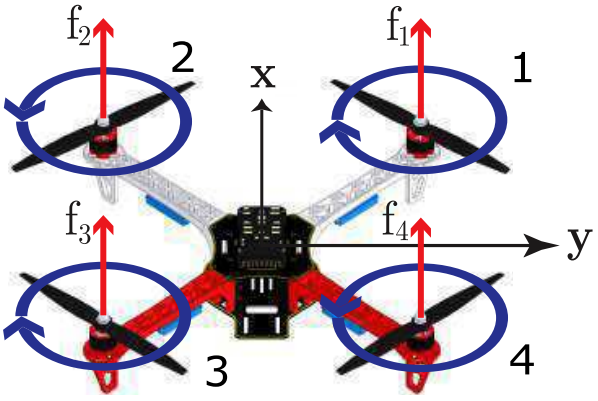


Figure 2.8: Motor labels (Adapted from Grujic and Nilsson (2016))

Substituting equations 2.3.3 and 2.3.4 into equation 2.3.1,

$$\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \times \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} db(\omega_4^2 - \omega_2^2) \\ db(\omega_1^2 - \omega_3^2) \\ k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2) \end{bmatrix}$$

let,
$U_1 = db(\omega_4^2 - \omega_2^2)$
$U_2 = db(\omega_1^2 - \omega_3^2)$, and
$U_3 = k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2)$

Now,

$$\begin{bmatrix} I_{xx}\ddot{\phi} \\ I_{yy}\ddot{\theta} \\ I_{zz}\ddot{\psi} \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \times \begin{bmatrix} I_{xx}\dot{\phi} \\ I_{yy}\dot{\theta} \\ I_{zz}\dot{\psi} \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

$$\begin{bmatrix} I_{xx}\ddot{\phi} \\ I_{yy}\ddot{\theta} \\ I_{zz}\ddot{\psi} \end{bmatrix} + \begin{bmatrix} I_{zz}\dot{\theta}\dot{\psi} - I_{yy}\dot{\theta}\dot{\psi} \\ -I_{zz}\dot{\phi}\dot{\psi} + I_{xx}\dot{\phi}\dot{\psi} \\ I_{yy}\dot{\phi}\dot{\theta} - I_{xx}\dot{\phi}\dot{\theta} \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

Next, make $\begin{bmatrix} \ddot{\phi} & \ddot{\theta} & \ddot{\psi} \end{bmatrix}^T$ the subject of the formula,

$$\begin{bmatrix} I_{xx}\ddot{\phi} \\ I_{yy}\ddot{\theta} \\ I_{zz}\ddot{\psi} \end{bmatrix} = \begin{bmatrix} I_{yy}\dot{\theta}\dot{\psi} - I_{zz}\dot{\theta}\dot{\psi} \\ I_{zz}\dot{\phi}\dot{\psi} - I_{xx}\dot{\phi}\dot{\psi} \\ I_{xx}\dot{\phi}\dot{\theta} - I_{yy}\dot{\phi}\dot{\theta} \end{bmatrix} + \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{(I_{yy} - I_{zz})\dot{\theta}\dot{\psi} + U_1}{I_{xx}} \\ \frac{(I_{zz} - I_{xx})\dot{\phi}\dot{\psi} + U_2}{I_{yy}} \\ \frac{(I_{xx} - I_{yy})\dot{\phi}\dot{\theta} + U_3}{I_{zz}} \end{bmatrix}$$

This can be written out as,

$$\ddot{\phi} = \frac{(I_{yy} - I_{zz})\dot{\theta}\dot{\psi}}{I_{xx}} + \frac{U_1}{I_{xx}} \tag{2.3.5}$$

$$\ddot{\theta} = \frac{(I_{zz} - I_{xx})\dot{\phi}\dot{\psi}}{I_{yy}} + \frac{U_2}{I_{yy}} \tag{2.3.6}$$

$$\ddot{\psi} = \frac{(I_{xx} - I_{yy})\dot{\phi}\dot{\theta}}{I_{zz}} + \frac{U_3}{I_{zz}} \tag{2.3.7}$$

2.4 State Space Representation

In this thesis, a linear time invariant (LTI) state space model will be used in representing the rotational motion of the quadcopter. Other linear model types include transfer function models and Finite Impulse Response (FIR) models (Rossiter, 2003). State space models are preferred as they are best suited for Multi-input Multi-output (MIMO) systems, model analysis and numerical calculations (Rossiter, 2003).

The state space model for the rotational motion of the quadcopter is derived from equations 2.3.5, 2.3.6 and 2.3.7. The state space matrices A, B and C are given in the equation 2.4.1 below,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{I_{xx}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}, C = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4.1}$$

The state vector x, and the input vector u, obtained from equation 2.3.4 are shown below,

$$x = \begin{bmatrix} \dot{\phi} \\ \ddot{\phi} \\ \dot{\theta} \\ \ddot{\theta} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix}, u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} db(\omega_4^2 - \omega_2^2) \\ db(\omega_1^2 - \omega_3^2) \\ k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2) \end{bmatrix} \tag{2.4.2}$$

In order to design the MPC angular rates controller, the state space model is linearised and then discretised. The motivation for discretisation is to enable the model to be implemented digitally at a specified sample rate (Grujic and Nilsson, 2016). The linear discrete model utilised in this thesis is obtained from Grujic and Nilsson (2016) in which Taylor series expansion is used to linearise the model about the hover position of the quadcopter.

The state vector, state matrices and input vector are represented compactly in the equations below where the subscript m stands for model.

$$x_m(k+1) = A_m x_m(k) + B_m u(k) \tag{2.4.3}$$

$$y(k) = C_m x_m(k) \tag{2.4.4}$$

There is no direct feedthrough signal from the input to the output therefore the state matrix D is excluded from equation 2.4.4.

The next section introduces model predictive control.

2.5 Model Predictive Control

This section begins with an overview of the history behind Model Predictive Control (MPC). The principle of MPC is discussed in full in subsection 2.5.2. Subsection 2.5.3 motivates the reasons for choosing MPC as the alternative to the current Pixhawk angular rates controller. A linear discrete state space

model is augmented for integral control in subsection. The controllability of the LMPC model is discussed in subsection 2.5.5. Output predictions, constraints, optimization and quadratic programming formulation are covered in subsections 2.5.6, 2.5.7, 2.5.8 and 2.5.9 respectively.

2.5.1 Historical background

Model predictive control was introduced by Richalet *et al.* (1978) in their paper, *'Model Predictive Heuristic Control - Application to Industrial Processes'* in which the effectiveness of this digital control method was attributed to its ease of implementation and robustness to structural perturbations.

In 1980, at the Joint Automatic Control Conference (JACC), C.R. Cutler and B.L.Ramaker presented their paper on *'Dynamic Matrix Control (DMC) - a Computer Control Algorithm'*(Cutler and Ramaker, 1980). Their research illustrates how the dynamics of a process are incorporated into the design of DMC, which integrates both feedforward and multivariable control. DMC was successfully implemented in process computer applications at Shell Oil Company Texas, USA, six years prior to the publication of this paper.

The following years saw further research in variations of MPC, applications of MPC and characteristics of MPC such as its robustness and stability. The evolution of MPC is outlined and discussed in Holkar and Waghmare (2010). In 2015, M.G. Forbes et al presented their paper, *'Model Predictive Control in Industry: Challenges and Opportunities'* in which they examined industrial practices and emerging research trends towards providing sustained MPC performance (Forbes *et al.*, 2015).

Advancements in processing power and microelectronics have enabled the MPC algorithm, that was mainly utilised in industries on processes with slow transient response, to be implemented on microcontrollers for processes with fast transient responses. Commercialisation and research in the UAV sector benefited significantly from this progression. Bemporad *et al.* (2009) proposed a hierarchal hybrid MPC approach for stabilisation and autonomous navigation of quadcopters. Mueller and D'Andrea (2013) published a paper on *'Model Predictive Control for Quadrocopter State Interception'*. Bangura and Mahony (2014) implemented an unconstrained MPC algorithm on a Pixhawk for position and trajectory tracking. Wang *et al.* (2017) presented a paper on *'Nonlinear Model Predictive Control with Constraint Satisfaction for a Quadcopter'*. These are just a few research papers involving MPC implemented on UAVs.

2.5.2 Principle of model predictive control

In model predictive control, a mathematical model of the process to be controlled is used to predict the model output, $y_m(k+i)$, of the process over

a horizon known as the **prediction horizon**, n_y (Rossiter, 2003). The chosen sample time is i for each prediction step, where the number of steps is equivalent to the size of the prediction horizon.

A sequence of control inputs is obtained by minimising a cost function of the error between the predicted outputs and the reference or set point values $r(k+i)$, and a weighted control input term $u(k+i)$ over a **control horizon**, n_u (Kwon and Han, 2006). The number of control inputs in this sequence is equivalent to the size of the control horizon. These sequence of control inputs are the control actions needed to drive the model and process to the reference values. Only the first element of the control input sequence is implemented, both in the model and the process, as the process measurement and reference values are constantly being updated in subsequent sampling instants.

This procedure of minimisation and implementation is repeated in successive sampling instants, and at each instant, it is assumed that the reference value remains constant over the prediction horizon (Rossiter, 2003). The block diagram in figure 2.9 provides a visual representation of the principle of MPC, and figure 2.10 illustrates the prediction horizon in MPC with a sample time of one.

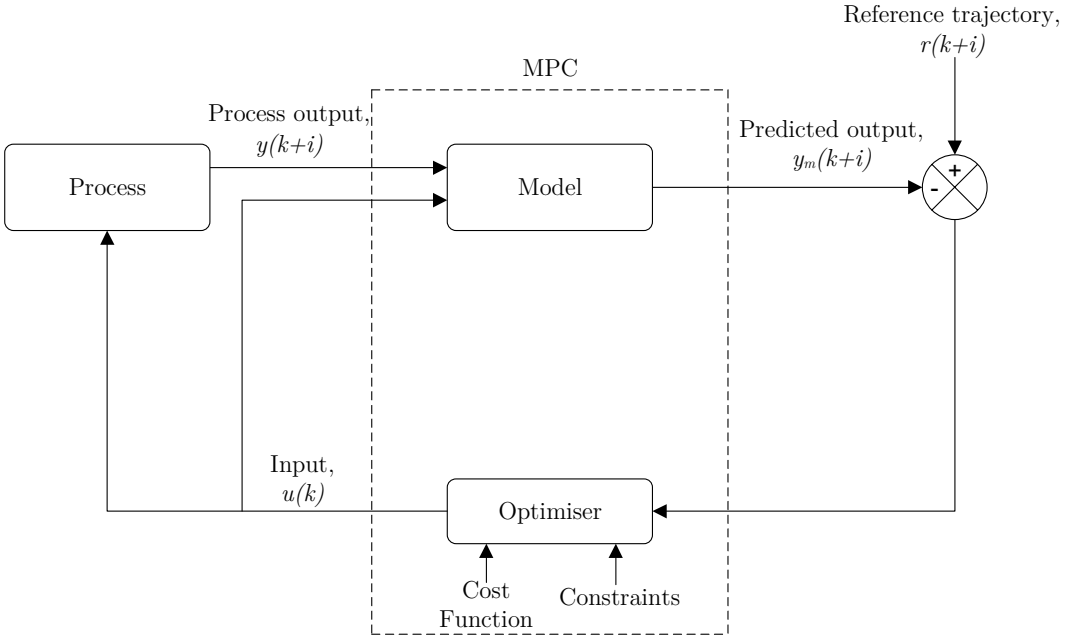


Figure 2.9: Principle of MPC

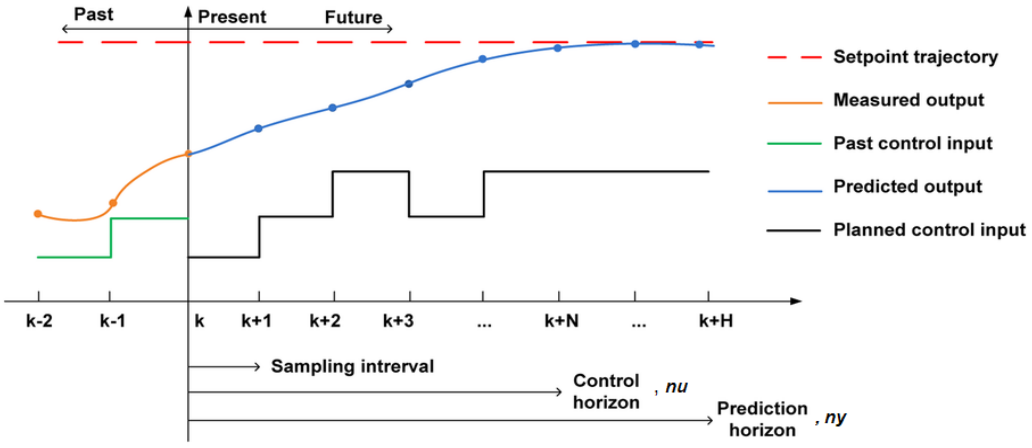


Figure 2.10: Prediction horizon (Adapted from Moradzadeh *et al.* (2014))

The current measured states or the outputs of the process are used in updating the future predictions to ensure that a more accurate model of the process is in use. The terminology **receding horizon** is often applied to predictive control because the prediction horizon is constantly moving away at subsequent sampling instants (Rossiter, 2003). At the current sample time, points that were previously beyond the prediction horizon are taken into account (Rossiter, 2003). Table 2.1 illustrates the receding horizon concept.

Table 2.1: Illustration of receding horizon (Rossiter, 2003)

Sampling instant	Horizon window
	0 1 2 3 4 5 6 7 8
0	→
1	→
2	→
3	→
⋮	⋮

One of the main attractions of MPC is its ability to systematically satisfy physical constraints on control inputs, rate of input change and/or outputs (Rossiter, 2003). These constraints are handled directly in the optimisation process. The optimisation process is captured in figure 2.9 in the optimiser block, which takes in constraints and cost function as its inputs.

2.5.3 Motivation for MPC

As briefly stated in the introductory chapter, there are different control methods that can be implemented on a quadcopter. MPC and PID were considered as replacements for the current angular rates controller on the Pixhawk.

The previous subsection established that MPC uses a model to predict the response of the system over a chosen prediction horizon, and the minimisation of the error between the output predictions and reference generates a control trajectory needed to drive the current states of the model to the reference states; but only the first element of the trajectory is applied on the system. Constraints on the system are taken into account in the controller formulation to ensure that control actions do not violate these constraints.

PID runs the risk of actuator saturation (Hu and Lin, 2001) as the physical limits of the system are not explicitly taken into account in determining control actions. Integral desaturation is one of the techniques employed to deal with these limits (or constraints). Control actions are assigned the maximum or minimum control values for controller demands that exceed that of the constraints on the system (Rossiter, 2003). With prediction and systematic constraint handling, MPC is better equipped to dealing with actuator saturation.

The current states of the system are typically used as the initial states in MPC. Predictions are made using these states and this introduces feedback into the MPC controller while concurrently compensating for modelling assumptions, uncertainties and disturbances (Dani *et al.*, 2017).

Model predictive control is also well suited for Multiple-Input Multiple-Output (MIMO) systems; the quadcopter constitutes as one. This is another advantage as MPC is able to systematically incorporate a model, constraints and cost function with relative ease (Wang, 2009).

In spite the benefits MPC has over PID, MPC is more computationally intensive as it involves a lot of matrix and vector operations. PID does not require a model of the system and there are no matrix and vector operations to be executed. However, the benefits MPC offers motivate the reason to investigate the feasibility of implementing the method on the PX4 firmware, for quadcopter angular rates control.

The table below shows a comparison of the different control methods.

Table 2.2: Comparison of control methods with constraint handling as a criteria

Control method	Model requirement	Constraint handling	Manual tuning
PID	No	No	Yes
MPC	Yes	Yes	Yes

2.5.4 Augmented state space matrices

The linear discrete state space model defined at the end of subsection 2.4 is used in formulating a linear MPC controller. The state space equations are listed again below for ease of reference.

$$x_m(k+1) = A_m x_m(k) + B_m u(k) \tag{2.5.1}$$

$$y(k) = C_m x_m(k) \tag{2.5.2}$$

where the subscript m stands for model.

In order to achieve off-set free tracking, integral action needs to be embedded by modifying the quadcopter model in equation 2.5.1 and 2.5.2. The model is augmented using the formulation described in Wang (2009).

Taking a difference operation on equations 2.5.1 and 2.5.2,

$$x_m(k+1) - x_m(k) = A_m(x_m(k) - x_m(k-1)) + B_m(u(k) - u(k-1)) \tag{2.5.3}$$

$$y(k+1) - y(k) = C_m(x_m(k+1) - x_m(k)) \tag{2.5.4}$$

$$y(k+1) = C_m(x_m(k+1) - x_m(k)) + y(k) \tag{2.5.5}$$

Let,

$$\Delta x_m(k+1) = x_m(k+1) - x_m(k) \tag{2.5.6}$$

$$\Delta u(k) = u(k) - u(k-1) \tag{2.5.7}$$

$$\Delta x_m(k) = x_m(k) - x_m(k-1) \tag{2.5.8}$$

Make these substitutions into equations 2.5.3 and 2.5.5,

$$\Delta x_m(k+1) = A_m \Delta x_m + B_m \Delta u(k) \tag{2.5.9}$$

$$y(k+1) = C_m(A_m \Delta x_m + B_m \Delta u(k)) + y(k)$$

$$y(k+1) = C_m A_m \Delta x_m + C_m B_m \Delta u(k) + y(k) \tag{2.5.10}$$

$\Delta u(k)$ is now the input to the state space model. A new state variable vector that relates Δx_m and $y(k)$ is chosen to be

$$x(k) = [\Delta x_m(k)^T y(k)]^T \tag{2.5.11}$$

where the subscript T indicates the matrix transpose.

The augmented state space model, after putting equations 2.5.9, 2.5.10 and 2.5.11 together, is

$$\begin{bmatrix} \Delta x_m(k+1) \\ y(k+1) \end{bmatrix} = \begin{bmatrix} A_m & O_{q\times n}^T \\ C_m A_m & I_{q\times q} \end{bmatrix} \begin{bmatrix} \Delta x_m(k) \\ y(k) \end{bmatrix} + \begin{bmatrix} B_m \\ C_m B_m \end{bmatrix} \Delta u(k) \tag{2.5.12}$$

$$y(k) = \begin{bmatrix} O_{q\times n} & I_{q\times q} \end{bmatrix} \begin{bmatrix} \Delta x_m(k) \\ y(k) \end{bmatrix} \tag{2.5.13}$$

where $I_{q\times q}$ is the identity matrix with dimensions $q \times q$, where q is the number of outputs. $O_{q\times n}$ is zero matrix with dimensions $q \times n$, where n is the number of states of the system or the state space dimensions.

Equations 2.5.12 and 2.5.13 can be simplified as follows:

$$x(k+1) = Ax(k) + B\Delta u(k) \tag{2.5.14}$$

$$y(k) = Cx(k) \tag{2.5.15}$$

with A, B and C corresponding to the augmented state matrices in the equation above.

2.5.5 Controllability

With the modification of the initial discrete state space model to an augmented one, it is necessary to check the controllability of the augmented state space model. A system is controllable if there exists a control input that transfers any state of the system to zero in finite time (Kalman *et al.*, 1960). An LTI system is shown to be controllable if and only if its controllability matrix, CO, has full rank, that is `rank(CO) = n`, where `rank` is a `MATLAB` command and n is the number of states (Golnaraghi and Kuo, 2010).

The controllability matrix, CO, is as follows:

$$CO = \begin{bmatrix} B & AB & A^2B ... A^{n-1}B \end{bmatrix} \tag{2.5.16}$$

where A and B are the state space matrices.

2.5.6 Output predictions

The previous subsection established the controllability of the augmented quadcopter model. The next step in designing the model predictive controller is to calculate the predicted plant output with the future control signal as the adjustable variable(s). This prediction is described within a preselected prediction horizon (Wang, 2009).

Taking k as the current sample time, the future control trajectory is denoted by,

$$\Delta U = \begin{bmatrix} \Delta u(k), & \Delta u(k+1), & \Delta u(k+2), & \cdots, & \Delta u(k+n_u-1) \end{bmatrix}^T \tag{2.5.17}$$

The control horizon is chosen to be less than or equal to prediction horizon.

The future state variables are required in order to calculate the predicted plant output. The future state variables are denoted as

$$x(k+1), \quad x(k+2), \quad x(k+3), \quad \cdots, \quad x(k+n_y) \tag{2.5.18}$$

The process used in calculating the future state and output predictions is as described by Rossiter (2003). Using the augmented quadcopter model in equations 2.5.14 and 2.5.15, the future state and output predictions are calculated recursively as follows,

at k+1,

$$x(k+1) = Ax(k) + B\Delta u(k) \tag{2.5.19}$$

$$y(k+1) = Cx(k+1) \tag{2.5.20}$$

at k+2,

$$x(k+2) = Ax(k+1) + B\Delta u(k+1) \tag{2.5.21}$$

$$y(k+2) = Cx(k+2) \tag{2.5.22}$$

Substituting equations 2.5.19 and 2.5.20 into 2.5.21 and 2.5.22 to eliminate x(k+1) results in

$$x(k+2) = A^2x(k) + AB\Delta u(k) + B\Delta u(k+1) \tag{2.5.23}$$

$$y(k+2) = C(A^2x(k) + AB\Delta u(k) + B\Delta u(k+1)) \tag{2.5.24}$$

at k+3,

$$x(k+3) = Ax(k+2) + B\Delta u(k+2) \tag{2.5.25}$$

$$y(k+3) = Cx(k+3) \tag{2.5.26}$$

Substituting equations 2.5.23 and 2.5.24 into equations 2.5.25 and 2.5.26,

$$x(k+3) = A^2x(k+1) + AB\Delta u(k+1) + B\Delta u(k+2) \tag{2.5.27}$$

$$y(k+3) = C(A^2x(k+1) + AB\Delta u(k+1) + B\Delta u(k+2)) \tag{2.5.28}$$

Similarly, equations 2.5.19 and 2.5.20 are substituted into equations 2.5.27 and 2.5.28 in order to eliminate x(k+1),

$$x(k+3) = A^3x(k) + A^2B\Delta u(k) + AB\Delta u(k+1) + B\Delta u(k+2) \tag{2.5.29}$$

$$y(k+3) = C(A^3x(k) + A^2B\Delta u(k) + AB\Delta u(k+1) + B\Delta u(k+2)) \tag{2.5.30}$$

This process continues recursively to give n_y-step ahead predictions. Hence, the output for n_y-step ahead predictions can be expressed as,

$$y(k+n_y) = C[A^{n_y}x(k) + A^{n_y-1}B\Delta u(k) + A^{n_y-2}B\Delta u(k+2) + \cdots + B\Delta u(k+n_y-1)] \tag{2.5.31}$$

The future output predictions is denoted in vector form below,

$$Y = \left[y(k+1),\quad y(k+2),\quad y(k+3), \cdots, y(k+n_y)\right]^T \tag{2.5.32}$$

This vector of output predictions is expressed compactly as follows,

$$\begin{bmatrix} y(k+1) \\ y(k+2) \\ y(k+3) \\ \vdots \\ y(k+n_y) \end{bmatrix} = \begin{bmatrix} CA \\ CA^2 \\ CA^3 \\ \vdots \\ CA^{n_y} \end{bmatrix} x(k) + \begin{bmatrix} CB & 0 & 0 & \cdots & 0 \\ CAB & CB & 0 & \cdots & 0 \\ CA^2B & CAB & CB & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ CA^{n_y-1}B & CA^{n_y-2}B & CA^{n_y-3}B & \cdots & CA^{n_y-n_u}B \end{bmatrix} \Delta U$$

This is further simplified as

$$Y = Px(k) + H\Delta U \tag{2.5.33}$$

where,

$$P = \begin{bmatrix} CA \\ CA^2 \\ CA^3 \\ \vdots \\ CA^{n_y} \end{bmatrix}; H = \begin{bmatrix} CB & 0 & 0 & \cdots & 0 \\ CAB & CB & 0 & \cdots & 0 \\ CA^2B & CAB & CB & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ CA^{n_y-1}B & CA^{n_y-2}B & CA^{n_y-3}B & \cdots & CA^{n_y-n_u}B \end{bmatrix}$$

and ΔU is as defined in equation 2.5.17.

2.5.7 Constraints

The final step before setting up the cost function is to define the system constraints. In practice, all systems are subject to operational constraints such as restricted dimensions and limited control capacity. In many situations, these constraints are purposely imposed and are intended to be made as tight as possible in order to reduce energy consumption, to minimise the utilisation of resources or to merely reduce the size of a certain device (Hu and Lin, 2003).

Constraints can be imposed on the control variable (also known as the input), *u(k)*, rate of input change variable, $\Delta u(k)$ and also the output, *y(k)* or state variables, *x(k)*. Considering the case of the quadcopter, which achieves control by varying the angular speeds, ω, of each motor, the speeds vary from a minimum value of zero to a maximum value that is dependent on the motor specification.

$$0 < \omega_i < \omega_{max} \tag{2.5.34}$$

where $i = 1, ..., 4$; for each motor.

Only constraints on the control variables, *u(k)* and their rates of change $\Delta u(k)$ will be considered in the MPC design. The constraints formulations used in this subsection are derived from Wang (2009). Constraints on the control variable are of the form,

$$u_{min} \leq u(k) \leq u_{max} \tag{2.5.35}$$

As there are three control variables necessary to control the rotational motion of the quadcopter, each control variable is subjected to constraints as shown below,

$$\begin{aligned} u_1^{min} &\leq u_1(k) \leq u_1^{max} \\ u_2^{min} &\leq u_2(k) \leq u_2^{max} \\ u_3^{min} &\leq u_3(k) \leq u_3^{max} \end{aligned} \tag{2.5.36}$$

This procedure is also applicable to the rates of input change. The rate of change constraints are as follows,

$$\begin{aligned} \Delta u_1^{min} &\leq \Delta u_1(k) \leq \Delta u_1^{max} \\ \Delta u_2^{min} &\leq \Delta u_2(k) \leq \Delta u_2^{max} \\ \Delta u_3^{min} &\leq \Delta u_3(k) \leq \Delta u_3^{max} \end{aligned} \tag{2.5.37}$$

The constraints on the rates of input change can be grouped into one variable, ΔU in order to take subsequent sample times into consideration. Therefore at sample time, k,

$$\Delta U(k)^{min} \leq \Delta U(k) \leq \Delta U(k)^{max}$$

where,

$$\Delta U(k)^{min} = \begin{bmatrix} \Delta u_1^{min} \\ \Delta u_2^{min} \\ \Delta u_3^{min} \end{bmatrix}, \Delta U(k) = \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \Delta u_3 \end{bmatrix}, \Delta U(k)^{max} = \begin{bmatrix} \Delta u_1^{max} \\ \Delta u_2^{max} \\ \Delta u_3^{max} \end{bmatrix}$$

Furthermore, these constraints can be split into separate inequalities. Taking sample times into consideration, the following expression is used to represent maximum rate of input change,

$$\begin{bmatrix} \Delta U(k) \leq \Delta U^{max} \\ \Delta U(k+1) \leq \Delta U^{max} \\ \vdots \\ \Delta U(k+n_u-1) \leq \Delta U^{max} \end{bmatrix}$$

For minimum rate of input change,

$$\begin{bmatrix} -\Delta U(k) \leq -\Delta U^{min} \\ -\Delta U(k+1) \leq -\Delta U^{min} \\ \vdots \\ -\Delta U(k+n_u-1) \leq -\Delta U^{min} \end{bmatrix}$$

These constraints are generalised in the following equation,

$$\begin{bmatrix} -\Delta U \leq -\Delta U^{min} \\ \Delta U \leq \Delta U^{max} \end{bmatrix} \tag{2.5.38}$$

where ΔU, ΔU^{min}, ΔU^{max} are vectors with the number of elements corresponding to the size of the control horizon, n_u $\times$ number of rate of input change constraints.

In matrix form,

$$\begin{bmatrix} -I \\ I \end{bmatrix} \Delta U \leq \begin{bmatrix} -\Delta U^{min} \\ \Delta U^{max} \end{bmatrix}$$

The input constraints can be written in a form that incorporates the rates of input change. This is shown in equation 2.5.39 below.

$$\begin{bmatrix} u(k) \\ u(k+1) \\ \vdots \\ u(k+n_u-1) \end{bmatrix} = \begin{bmatrix} I \\ I \\ \vdots \\ I \end{bmatrix} u(k-1) + \begin{bmatrix} I & 0 & 0 & \dots & 0 \\ I & I & 0 & \dots & 0 \\ \vdots & & & & \\ I & I & \dots & I & I \end{bmatrix} \begin{bmatrix} \Delta u(k) \\ \Delta u(k+1) \\ \vdots \\ \Delta u(k+n_u-1) \end{bmatrix} \tag{2.5.39}$$

Equation 2.5.39 is expressed compactly below,

$$-(C_1 u(k-1) + C_2 \Delta U) \leq -U^{min}$$
$$(C_1 u(k-1) + C_2 \Delta U) \leq -U^{max}$$

where,

$$C_1 = \begin{bmatrix} I \\ I \\ \vdots \\ I \end{bmatrix}, C_2 = \begin{bmatrix} I & 0 & 0 & \dots & 0 \\ I & I & 0 & \dots & 0 \\ \vdots & & & & \\ I & I & \dots & I & I \end{bmatrix}$$

The control input and rates of input change can be grouped as follows,

$$\begin{bmatrix} C_u \\ C_{\Delta u} \end{bmatrix} \Delta U \leq \begin{bmatrix} d_u \\ d_{\Delta u} \end{bmatrix} \tag{2.5.40}$$

where,

$$C_u = \begin{bmatrix} -C_2 \\ C_2 \end{bmatrix}; d_u = \begin{bmatrix} -U^{min} + C_1 u(k-1) \\ U^{max} - C_1 u(k-1) \end{bmatrix}; C_{\Delta u} = \begin{bmatrix} -I \\ I \end{bmatrix}; d_{\Delta u} = \begin{bmatrix} -\Delta U^{min} \\ \Delta U^{max} \end{bmatrix}$$

Equation 2.5.40 is compacted further in the equation below,

$$CC\Delta U \leq d \tag{2.5.41}$$

where,

$$CC = \begin{bmatrix} C_{\Delta u} \\ C_u \end{bmatrix}, d = \begin{bmatrix} d_{\Delta u} \\ d_u \end{bmatrix}$$

2.5.8 Optimisation

As stated in subsection 2.5.2, the goal of model predictive control is to drive the predicted outputs to the desired reference values. This goal is achieved by minimising a cost function that is comprised of the error between predicted outputs and reference values, subjected to constraints on the system. The result from this optimisation is a series of inputs, ΔU, used to drive the predicted outputs to the reference.

At sample time, k, output predictions are generated according to the size of the prediction horizon (Wang, 2009). For instance, for a system with a prediction horizon of eight, a series of output predictions totalling eight in number is generated. Given that at any sample time, only one set of reference values, $r(k)$, are available and the cost function relates the errors between the reference values and predicted outputs, it is assumed that for each optimisation window, the reference values are constant (Wang, 2009).

This is represented in the equation 2.5.42 below, obtained from Wang (2009), where I is an identity matrix with dimensions corresponding to the number of outputs. The number of identity matrices is equivalent to size of the prediction horizon.

$$R_s = \begin{bmatrix} I \\ I \\ \vdots \\ I \end{bmatrix} r(k) \tag{2.5.42}$$

At sample time, k, the reference value(s), $r(k)$, is transformed to a reference matrix R_s, before being used in the cost function in equation 2.5.43 below.

$$J = (R_s - Y)^T (R_s - Y) + \Delta U^T W \Delta U \tag{2.5.43}$$

The cost (or objective) function is obtained from Wang (2009). The first term in the cost function minimises the errors between the predicted output and the reference value while the second term is linked with minimising the size of ΔU.

W is the diagonal weight matrix for control inputs. A weight matrix with low values signifies that the focus of minimising the cost function above is on the error between the reference and the predicted output values. This results

in large magnitude in the ΔU values and this equates to rapid control as the ΔU values diminish rapidly. But a weight matrix with high diagonal values places the importance on minimising the values of ΔU, and this results in slower control as the ΔU values decrease more slowly (Wang, 2009).

Substituting $Y = P(x) + H\Delta U$ from equation 2.5.33 into equation 2.5.43,

$$J = (R_s - Px(k))^T(R_s - Px(k)) - 2\Delta U^T H^T(R_s - Px(k)) + \Delta U^T(H^T H + W)\Delta U \tag{2.5.44}$$

2.5.9 Quadratic programming formulation

Taking the first derivative of the cost function in equation 2.5.44, the following equation is obtained,

$$J = \frac{1}{2}\Delta U^T E \Delta U + \Delta U^T F \tag{2.5.45}$$

with constraints $CC\Delta U \leq d$

where $E = 2(H^T H + W)$, $F = -2H^T(R_S - Px(k))$ and R_S is the reference matrix

Taking the number of matrix and vector operations to be carried out in MPC into consideration, it is important to choose an efficient method to solve the quadratic programming problem in equation 2.5.45, to maximise the limited computational resources on the microcontroller.

Linear MPC generally leads to structured convex quadratic programs to be solved (Diehl, 2015). The convexity of the quadratic program ensures that a global solution to the problem is obtainable. Two methods are commonly used in solving quadratic programming problems (Lau *et al.*, 2009) — Interior Point Method (IPM) and Active Set Method (ASM).

These methods differ in their approach in handling linear inequalities, (Diehl, 2015) which in this case are the constraints. Before elaborating on these methods, the necessary conditions that must be satisfied in minimising objective functions with inequality constraints (Snyman, 2005) will be listed. These conditions are known as the **Karush Kuhn and Tucker (KKT)** conditions. For notational simplicity, the following objective function in equation 2.5.46 will be used.

$$J = \frac{1}{2}x^T E x + x^T F \tag{2.5.46}$$

such that $Mx \leq \gamma$
The KKT conditions are as follows,

$$Ex + F + M^T\lambda = 0$$

$$\begin{aligned} Mx - \gamma &\leq 0 \\ \lambda^T(Mx - \gamma) &= 0 \\ \lambda &\geq 0 \end{aligned} \tag{2.5.47}$$

where λ is a vector of Lagrange multipliers. Taking S_{act} to denote the index set of active constraints (Wang, 2009), the conditions are now expressed as

$$Ex + F + \sum_{i \in S_{act}} \lambda_i M_i^T = 0 \tag{2.5.48}$$

$$M_i x - \gamma_i = 0 \qquad i \in S_{act} \tag{2.5.49}$$

$$M_i x - \gamma_i < 0 \qquad i \notin S_{act} \tag{2.5.50}$$

$$\lambda_i \geq 0 \qquad i \in S_{act} \tag{2.5.51}$$

$$\lambda_i = 0 \qquad i \notin S_{act} \tag{2.5.52}$$

where M_i is the ith row of the M matrix. $M_i x_i - \gamma_i = 0$ is an equality constraint and signifies that the constraint is active. However, a constraint $M_i x_i - \gamma_i < 0$ indicates that the constraint is satisfied (Wang, 2009). If a Lagrange multiplier is zero, the constraint is inactive whereas a non-negative Lagrange multiplier signifies that the constraint is active (Wang, 2009).

Interior point methods approach the KKT conditions for the quadratic program inequality problem using successive descent steps (Lau *et al.*, 2009). Each descent step is obtained by the Newton's method for optimisation, which constitutes a linear system to be factored and solved — these iterations are expensive to compute (Diehl, 2015). Lau *et al.* (2009) performed a comparison of interior point method and active set methods for Field Programmable Gate Array (FPGA) implementation of MPC. In this paper, they compared computational complexity, storage, speed of convergence and some issues in practical implementation of these methods on the FPGA.

They discovered that ASM performs better than IPM when the number of variables and constraints are small. This validates Nocedal and Wright (2006) where they stated that IPM was well suited for larger problems. With the concentration on rates control in this thesis, the number of variables and constraints is considered small. Therefore, the active set method was chosen to solve the linear MPC quadratic programming problem.

Active set methods define a set of constraints called the **working set**, that is taken as the active set. The working set is a subset of the constraints that are active at the current point. The current point is a local solution to the original main problem if all the Lagrange multipliers, $\lambda_i \geq 0$. Otherwise, if there exists $\lambda_i < 0$, the corresponding constraint is deleted from the constraint equation (Wang, 2009).

Active set methods belong to a group of methods known as **primal methods**. As active set methods require active constraints to be identified, this

can constitute a large computational load if the constraints are a lot (Wang, 2009). Wang (2009) suggests tackling this problem by using a **dual method** to systematically eliminate constraints; the derivation of this method is outlined below.

The dual problem derivation of the original primal problem is given below,

$$\max_{\lambda \geq 0} \min_{x} [\frac{1}{2} x^T E x + x^T F + \lambda^T (Mx - \gamma)] \tag{2.5.53}$$

Minimisation of x is obtained by

$$x = -E^{-1}(F + M^T \lambda) \tag{2.5.54}$$

Substituting this into equation 2.5.53 results in

$$\max_{\lambda \geq 0} (-\frac{1}{2} \lambda^T T \lambda - \lambda^T K - \frac{1}{2} \gamma^T E^{-1} \gamma) \tag{2.5.55}$$

where the matrices T and K are

$$T = M E^{-1} M^T \tag{2.5.56}$$

$$K = \gamma + M E^{-1} F \tag{2.5.57}$$

Maximising an objective function is equivalent to minimising the negative of that objective function. Thus equation 2.5.55 becomes,

$$\min_{\lambda \geq 0} (\frac{1}{2} \lambda^T T \lambda + \lambda^T K + \frac{1}{2} \gamma^T E^{-1} \gamma) \tag{2.5.58}$$

An algorithm known as the **Hildreth's quadratic programming** procedure is used in solving the dual programming problem in equation 2.5.58. The steps of the algorithm are as outlined in (Wang, 2009) and are shown below using `MATLAB` syntax.

Algorithm 1: Hildreth's quadratic programming procedure

```
% Initialisation
[rowM, colM] = size(M);
x = −E \ F;
j = 0;
for i = 1 : rowM do
  % Constraint violation check
  if M(i, :) × x > γ(i) then
    j = j + 1;
  else
    j = j + 0;
  end if
end for
if j == 0 then
  return
end if
% Setup matrices for dual quadratic programming
T = M(E \ M^T);
K = γ + (ME \ F);
[rowK, colK] = size(K);
% Initialise lambda
λ = zeros(rowK, colK);
al = 10;
for y = 1 : 40 do
  λ_prev = λ;
  for i = 1 : rowK do
    w = T(i, :) ∗ λ − T(i, i) ∗ λ(i, 1);
    w = w + K(i, 1);
    la = −w/T(i, i);
    λ(i, 1) = max(0, la);
  end for
  al = (λ − λ_prev)^T (λ − λ_prev);
  if al < 10^−4 then
    break;
  end if
end for
x = −E \ F − E \ M^T ∗ λ;
```

The Hildreth's quadratic programming algorithm does not require matrix inversion, it is based on an element-by-element search, therefore there will not be any interruption in computation (Wang, 2009). Another advantage of this approach is its ability to recover from an ill-conditioned constrained problem, ensuring safe operation of the plant as observed by Wang (2009).

An example is solved in appendix A to give a better understanding of objective function, constraints and optimisation.

2.6 State Observer

Information on the current states of the quadcopter is needed in order to control the angular rates of the quadcopter. A state observer is used to estimate the required states of the quadcopter from the sensor measurements. The sensors onboard the Pixhawk were described in subsection 2.1. The state observer implemented on the Pixhawk is the Extended Kalman Filter (EKF). The structure of the EKF is described in subsection 2.6.1 and its implementation is detailed in subsection 2.6.2.

2.6.1 EKF structure

A Kalman filter is an iterative mathematical process that uses a set of equations and consecutive data inputs to quickly estimate the true value of the states of a system being measured; which contain random error or uncertainty (Julier and Uhlmann, 1997).

A linear Kalman filter is used to estimate states of a linear system. An extended Kalman filter extends this estimation ability to non-linear systems (Wan and Van Der Merwe, 2000). The linear Kalman filter and EKF approach state estimation in a similar way (Henriques, 2011). The difference lies in the formulation of certain equations. Therefore, the structure of the linear Kalman filter will be considered in this subsection. The equations required for estimation by EKF are outlined subsequently.

The linear Kalman filter requires a model of the system that has the states to be estimated. Taking the quadcopter as an example of a system, the linear discrete state space model in the equation below is used.

$$x(k+1) = A_d x(k) + B_d u(k) + \omega(k) \tag{2.6.1}$$

$$y(k) = C_d x(k) + \nu(k) \tag{2.6.2}$$

where $\omega(k)$ and $\nu(k)$ are the process or system and measurement noise respectively. It is assumed that there is no feedthrough signal in this model, therefore state matrix D is omitted in equation 2.6.2.

The Kalman filter formulation outlined in this section is obtained from Choset (2005), with modification of variables to ensure consistency with the state space matrix variables used hitherto in this thesis. To commence the linear Kalman filtering process, an initial state vector, $x_o(k)$ and error covariance matrix, $P_o(k)$ are declared. The error covariance matrix represents the error in the system or state estimates. $x_o(k)$ and $P_o(k)$ become the past predicted state, $x_p(k-1)$ and past predicted error covariance matrix, $P_p(k-1)$ respectively.

Equation 2.6.3 is used to predict the new state vector and equation 2.6.4 is used to predict the new error covariance matrix.

$$x_p(k) = A_d x_p(k-1) + B_d u(k) + \omega(k) \tag{2.6.3}$$

$$P_p(k) = A_d P_p(k-1) A_d^T + Q(k) \tag{2.6.4}$$

where Q_k denotes the noise covariance matrix and the subscript p indicates a prediction of the accompanying variable.

Next the Kalman gain, $K_{al}(k)$, is computed. The Kalman gain weighs the error between predictions and the measurements obtained from sensor readings. The Kalman gain is computed using the equation below,

$$K_{al}(k) = \frac{P_p(k)H}{HP_p(k)H^T + R(k)} \tag{2.6.5}$$

where $R(k)$ is the measurement noise covariance error. H is an identity matrix used to ensure equivalence of matrix dimensions between two or more matrices; in this instance, equivalence between matrices $P_p(k)$ and $R(k)$.

The next step is to compute the estimated state vector and error covariance matrix. These variables are computed using equations 2.6.6 and 2.6.7 respectively.

$$\hat{x}(k) = x_p(k) + K_{al}[y(k) - Hx_p(k)] \tag{2.6.6}$$

where $y(k)$ is the measurement obtained from the sensor(s).

$$\hat{P}(k) = (I - K_{al}H)P_p(k) \tag{2.6.7}$$

$\hat{}$ signifies that the corresponding variable is an estimate.

The obtained estimates of the state vector and error covariance matrix become the past predicted state vector, $x_p(k-1)$ and past predicted error covariance matrix, $P_p(k-1)$ for the next iteration of the Kalman filter. This estimation procedure is carried out for each time step. With each iteration, the Kalman filter output states that are closer to their actual value. Figure 2.11 illustrates the linear Kalman filter estimation process.

EKF linearises a non-linear system about the current state estimate, $\hat{x}$ (Henriques, 2011). As stated earlier, EKF differs from the linear Kalman filter in particular equation formulations. Equations 2.6.1 and 2.6.2 are replaced by equations 2.6.8 and 2.6.9 respectively below.

$$x(k) = f(x(k-1), \omega(k)) \tag{2.6.8}$$

$$y(k) = h(x(k), \nu(k)) \tag{2.6.9}$$

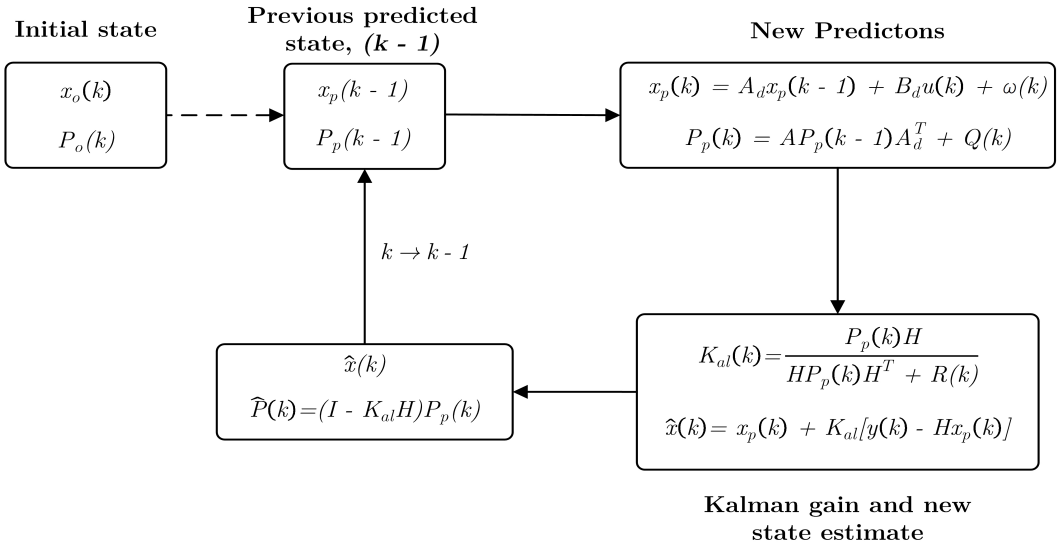


Figure 2.11: Flow diagram of Kalman filter

The partial derivates of functions f and h, with respect to the state vector, $x(k)$, and noise vectors $\omega(k)$ and $\nu(k)$, are derived by computing the Jacobian as shown in the equations below. EKF proceeds as a linear Kalman filter after effecting these changes.

$$F(k-1) = \frac{\partial f}{\partial x}|_{\hat{x}(k-1)} \tag{2.6.10}$$

$$H(k) = \frac{\partial h}{\partial x}|_{\hat{x}(k)} \tag{2.6.11}$$

2.6.2 EKF implementation

As the focus of this thesis is on the quadcopter rates control, state estimation is achieved using the default EKF algorithm programmed on the Pixhawk software. The EKF on the Pixhawk uses an IMU, magnetometer and GPS to estimate the following states:

- Quaternion defining rotation from North, East, Down world-fixed frame to x, y, and z body-fixed frame.
- Velocity at the IMU - North, East, Down (m/s)
- Position at the IMU - North, East, Down (m)
- IMU delta velocity bias estimates - x, y, and z (m/s)
- Earth magnetic components - North, East, Down (gauss)

- Vehicle body frame magnetic field bias - x, y and z (gauss)
- Wind velocity - North, East (m/s)

The EKF has different modes of operation that enables a variation of combinations of sensor measurements. The sensors listed above provide the minimum data required for all the different modes to run. Further information about the EKF implemented on the Pixhawk is available on the Pixhawk developer website.

2.7 Pixhawk Autopilot

The Pixhawk autopilot is an independent, open-hardware project aimed at providing high-end autopilot hardware to academics, hobbyists and industries at low cost and high availability (Pixhawk, 2013). The project originated from the Pixhawk Project of the Computer Vision and Geometry Lab of ETH Zurich (Swiss Federal Institute of Technology) and Autonomous Systems Lab (Meier *et al.*, 2015).

The Pixhawk autopilot runs on the PX4 open source software. The structure outlined in this section is a summary of research paper from Meier *et al.* (2015). The software structure is illustrated in figure 2.12 below. The software structure is divided into two halves with two layers each, with the NuttX RTOS layer serving as the divider. The lower half manages device drivers required for a particular microcontroller or bus type. In addition, this half includes a simulation layer that enables the PX4 flight code to run on a desktop operating system and control a computer modeled vehicle in a simulated world. This simulation procedure is known as **software-in-the-loop (SITL)**.

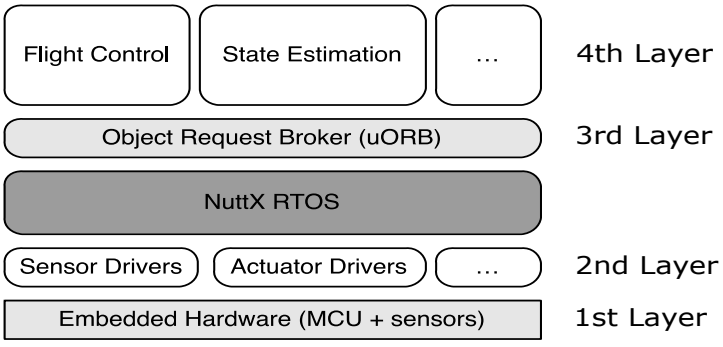


Figure 2.12: Software structure (Adapted from Meier *et al.* (2015))

The third layer, in the upper half, is the micro object request broker, μORB. μORB provides a data structure that handles data distribution between applications. Examples of some applications on the PX4 are the position controller, attitude controller and state estimator. The broker uses a publish-subscribe

data protocol. A publisher advertises a topic which contains information to be shared (Meier *et al.*, 2015). Topics are communication messages that contain structured data.

For example, the estimator application advertises state estimates as a topic and publishes it to μORB for other applications. Applications subscribe to this topic via μORB to access the state estimate data. Once subscription to this topic is established, the relevant state estimate data is copied into a variable to be used directly in the application. Applications can be both a publisher and subscriber. An application that subscribes to a topic can modify data within that topic, then advertise the topic with modified data for other applications to subscribe and copy. A subscriber can request new data from a topic at its own pace with a polling function or receive new data the instant it is available (Meier *et al.*, 2015).

Figure 2.13 below illustrates how inter-application messaging is handled by the micro object request broker, where the labeled lines are the functions required to execute the respective topic actions.

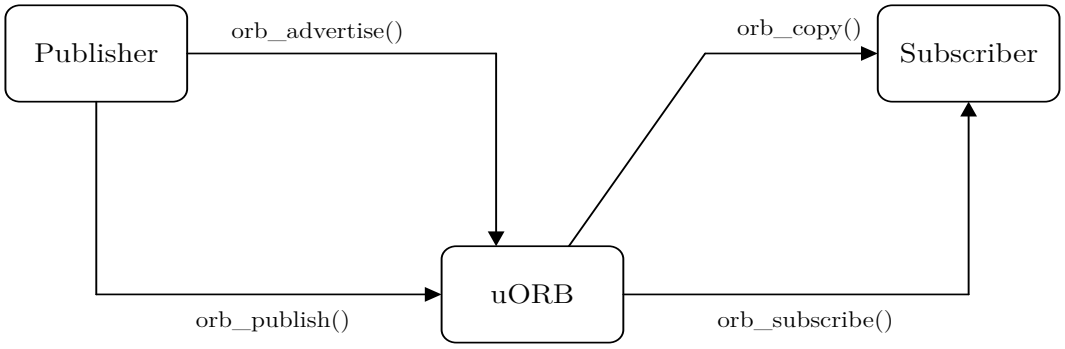


Figure 2.13: Micro object request broker, μORB

The fourth layer in the software structure are the PX4 applications, such as state estimators, controllers and more. These applications are executed as independently (Meier *et al.*, 2015). In the PX4 software, all the applications are classified as modules.

The specifications of the Pixhawk are listed in appendix B.

Chapter 3

Controller Design and Implementation

This chapter commences by expanding on the PX4 code architecture and how the attitude module is structured in the PX4 firmware. The matrices and vectors required to design and set up the MPC controller are formulated in section 3.2. In section 3.3, the MPC controller is implemented in MATLAB. In section 3.4, the MPC controller is implemented in C++ in a software-in-the-loop (SITL) simulator. Finally, section 3.5 details the procedure for implementing the MPC controller on the Pixhawk hardware for flight testing.

3.1 PX4 Architecture

The PX4 software is divided into two main layers: the flight stack and the middleware. The middleware consists of drivers for the embedded sensors, communication with the external world (computer, ground control station (GCS) et cetera), the uORB publish-subscribe message bus and the simulation layer. The flight stack is elaborated on in the following subsection.

3.1.1 PX4 flight stack

The PX4 flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. The flight stack includes controllers for multirotor, fixed wing and Vertical Take Off and Landing (VTOL) vehicles in addition to attitude and position estimators. Figure 3.1 provides a visual representation of the PX4 flight stack where each block is a module or application.

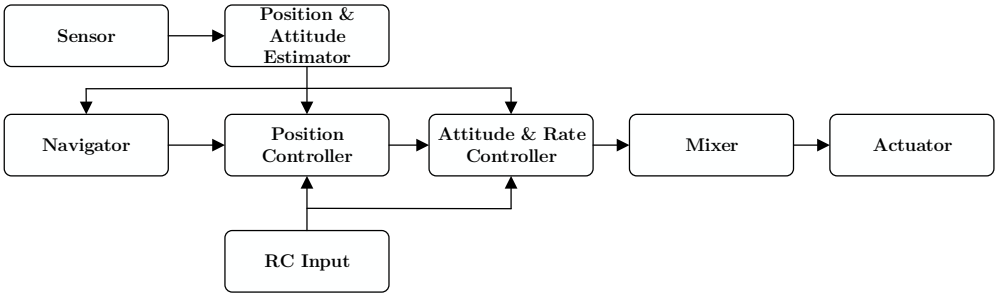


Figure 3.1: PX4 flight stack

The estimators receive sensor readings, combine them and use an algorithm — the extended kalman filter (EKF) — to estimate the vehicle states. These estimated states are used by the navigator, an algorithm for autonomous flight control, the position controller and the attitude and rates controller. The position and attitude (and rates) controllers receive remote control (RC) input of the vehicle position and/or orientation desired by the user, via the radio receiver connected to the Pixhawk. The RC and the receiver are connected wirelessly through radio protocols at a particular radio frequency. The rates controller outputs commands, like turning left, to the mixer block which converts them into motor commands for each motor. Finally, these motor commands are sent to the ESCs to regulate the speed of each motor (actuator) as needed.

3.1.2 PX4 attitude control module

Before designing the MPC controller for angular rates control, it is necessary to understand how the current attitude module on the Pixhawk is structured. The attitude module is comprised of two loops: Proportional (P) controller loop for angular error and a Proportional Derivative (PD) controller loop for angular rate error; at the time of writing this thesis, the angular rate error was controlled by a PD controller. In figure 3.2 below, the P controller and PD controllers are used in the attitude and rates controller blocks respectively.

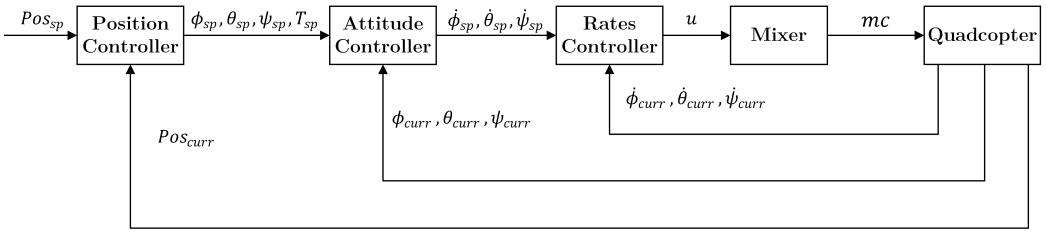


Figure 3.2: PX4 control block diagram

where,
curr - current
mc - motor commands
Pos - position
sp - set point
T - thrust

The `mc_att_control` module on the PX4 flight stack is responsible for attitude and rates control of the quadcopter. The flow chart in fig 3.3 shows the function call hierarchy and conditionals of the `mc_att_control` module where the rectangular symbols represent program functions.

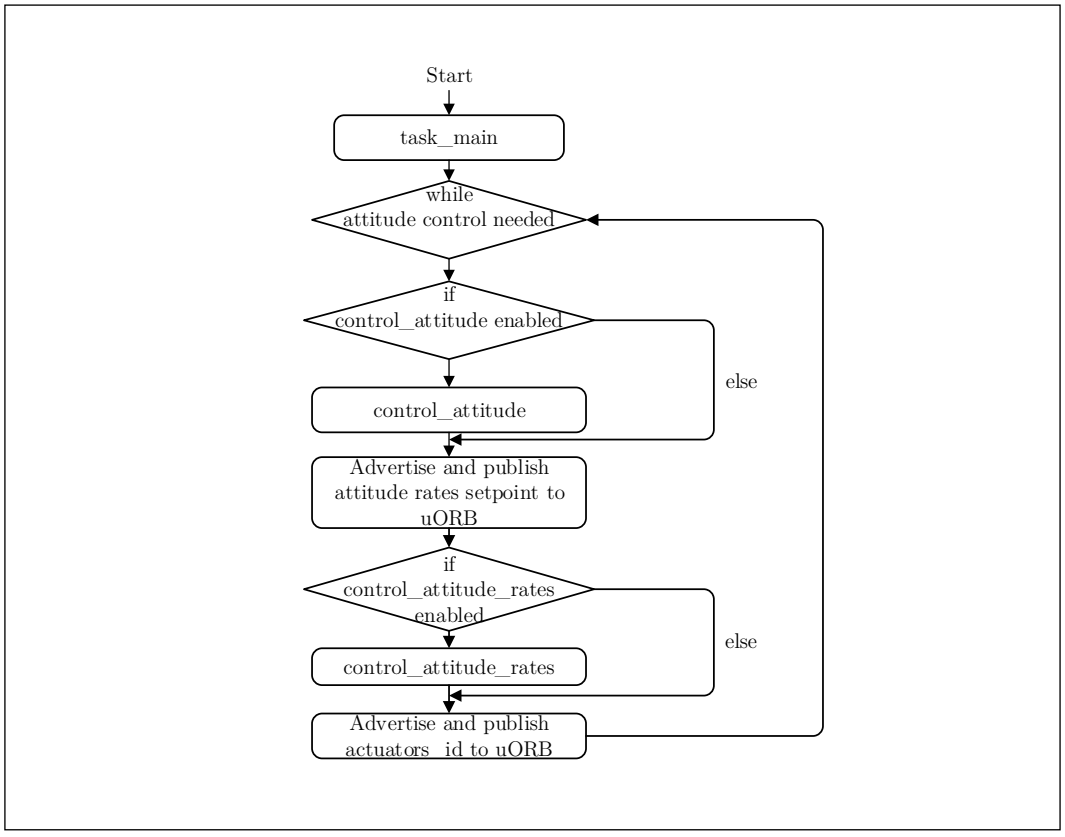


Figure 3.3: PX4 attitude controller flow chart

3.2 Model Predictive Controller

The proposed model predictive controller to control the angular rates of the quadcopter is represented in the block diagram in figure 3.4 below. In the MPC block, the current angular rates are augmented with the quadcopter state space model before the start of each optimisation loop. This is done to acquire more

accurate quadcopter state predictions to be used in the optimiser. Employing this approach ensures that model uncertainties and assumptions, made in the modelling process, are compensated for.

Rate setpoints are sent to the optimiser block from the attitude controller. The optimiser also receives the cost function and system constraints. The Hildreth's quadratic programming procedure is used in the optimiser block to minimise the cost function subject to predefined quadcopter constraints.

The output of the MPC controller is the control or input vector, u. This vector is scaled between -1 and 1 before being advertised and published to uORB; scaling in this form is a requirement of the PX4 control architecture. The scaled input vector is then sent to the mixer module to generate motor commands for quadcopter.

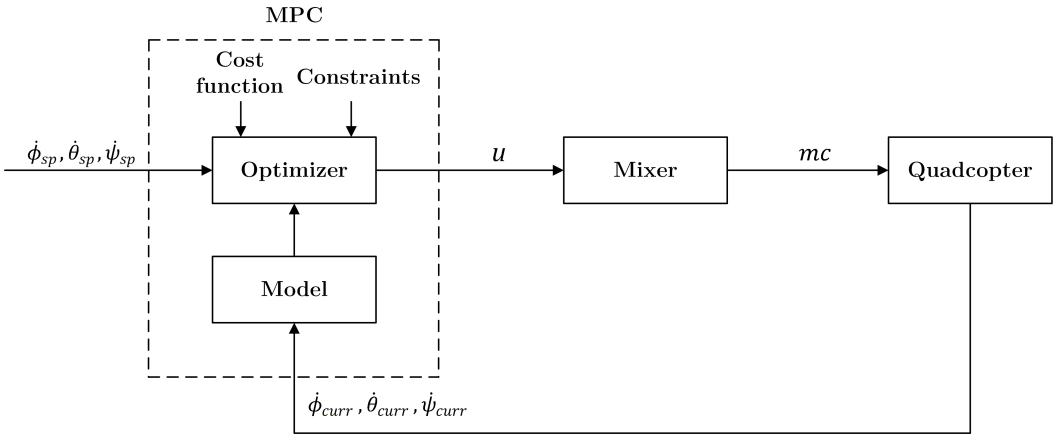


Figure 3.4: Modified MPC block diagram

In designing the model predictive controller, particular physical properties of the quadcopter were required. A summary of the physical properties obtained by experimentation or instrumentation are presented in the table 3.1 below. Appendix C provides the full details of how these properties were obtained.

Table 3.1: Quadcopter parameters

Quadcopter Parameters	
Mass, m	1.587 kg
Moment arm, d	0.243 m
Thrust coefficient, b	4.0687 $\times 10^{-7}$N/rpm^2
Drag coefficient, k	8.4367 $\times 10^{-9}$Nm/rpm^2
Moment of inertia about x-axis, I_{xx}	0.0213 kgm^2
Moment of inertia about y-axis, I_{yy}	0.02217 kgm^2
Moment of inertia about z-axis, I_{zz}	0.0282 kgm^2

In the following subsections, the matrix and vector parameters used in the MPC controller were formulated.

3.2.1 State matrices and vectors

As mentioned in subsection 2.5.4, the MPC controller makes use of a linear discrete state space model. The MATLAB function `cd2m()` was used to convert the continuous state space model into a discrete one. This procedure is shown below,

```
[A_m, B_m, C_m, D_m] = c2dm(A, B, C, D, ts);
```

The function takes the continuous state matrices and sample time as arguments and returns discrete state space matrices.

As described in subsection 2.5.4, the Wang (2009) approach was used to formulate the augmented state space matrices.

Equations 3.2.1, 3.2.2 and 3.2.3 show the linear discrete model state space model and equations 3.2.4, 3.2.5 and 3.2.6 show the augmented state space model matrices. A sample time of 0.2 seconds was used in obtaining these equations, with the values of the moments of inertia from table 3.1 substituted in.

$$A_m = \begin{bmatrix} 1 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2.1}$$

$$B_m = \begin{bmatrix} 0.9390 & 0 & 0 \\ 9.3897 & 0 & 0 \\ 0 & 0.9021 & 0 \\ 0 & 9.0212 & 0 \\ 0 & 0 & 0.7092 \\ 0 & 0 & 7.0922 \end{bmatrix} \tag{3.2.2}$$

$$C_m = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2.3}$$

The augmented state space matrices of the quadcopter are given below:

$$A = \begin{bmatrix} 1 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{3.2.4}$$

$$B = \begin{bmatrix} 0.9390 & 0 & 0 \\ 9.3897 & 0 & 0 \\ 0 & 0.9021 & 0 \\ 0 & 9.0212 & 0 \\ 0 & 0 & 0.7092 \\ 0 & 0 & 7.0922 \\ 9.3897 & 0 & 0 \\ 0 & 9.0212 & \\ 0 & 0 & 7.0922 \end{bmatrix} \tag{3.2.5}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2.6}$$

The state and input/control vectors used were defined in section 2.5.4. These vectors are presented again in equation 3.2.7 below.

$$x = \begin{bmatrix} \Delta x_m \\ y \end{bmatrix}, \Delta u = \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \Delta u_3 \end{bmatrix} \tag{3.2.7}$$

where

$$\Delta x_m = \begin{bmatrix} \Delta\phi & \Delta\dot{\phi} & \Delta\theta & \Delta\dot{\theta} & \Delta\psi & \Delta\dot{\psi} \end{bmatrix}^T$$

The augmented state space equations are placed in compact form in equations 3.2.8 and 3.2.9 below.

$$x(k+1) = Ax(k) + B\Delta U(k) \tag{3.2.8}$$

$$y(k) = Cx(k) \tag{3.2.9}$$

3.2.2 Constraints

The constraints on the quadcopter system are the maximum speeds of the motors. The quadcopter used in this thesis is fitted with four brushless EMAX MT3506 650 kv motors.

The maximum speed rating of the motors was obtained from the EMAX website by cross-referencing the propeller type, propeller size, voltage and current of the battery used in the quadcopter with the data available on the website. The maximum speed rating was found to be 4720 rpm.

The constraints employed in the MPC formulation were constraints on the input and the rate of input change. In order to determine these constraints, the total thrust generated the motors needed to be calculated. The mass of the quadcopter (battery included) was read from a mass scale to be 1.587 kg.

Taking acceleration due to gravity to be 9.81 m/s^2,

$$\text{weight} = 1.587 \times 9.81 = 15.57\,\text{N}$$

Thrust of the quadcopter was calculated as follows,

$$\text{Thrust} = b\sum_{i=1}^{4}\omega_i^2 \tag{3.2.10}$$

where,

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

From table 3.1, the thrust coefficient is 4.0687×10^{-7}N/rpm^2. The speed for each quadcopter motor was calculated as follows,

$$15.57 = b\sum_{i=1}^{4}\omega_i^2$$

$$\omega_i = \sqrt{\frac{15.57}{4 \times 4.0687 \times 10^{-7}}}$$

$$\approx 3093\,\text{rpm}$$

As established in subsection 2.3.1 of the literature review, the input for the rotational motion of the quadcopter are torques in the roll, pitch and yaw axes. These torque equations are listed below,

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} db(\omega_4^2 - \omega_2^2) \\ db(\omega_1^2 - \omega_3^2) \\ k(\omega_1^2 + \omega_3^2 - \omega_2^2 - \omega_4^2) \end{bmatrix} \tag{3.2.11}$$

where U_1, U_2 and U_3 are torques τ_ϕ, τ_θ and τ_ψ respectively.

The input constraints are the maximum and minimum torques and are calculated below. For instance, in calculating the maximum rolling torque, $\tau_{\phi_{max}}$, the maximum motor speed was substituted into ω_4 and the minimum motor speed substituted into ω_2. The maximum and minimum motor speeds are 4720 rpm and 3093 rpm respectively.
The maximum torques are calculated below,

$$\tau_{\phi_{max}} = 0.243 \times 4.0687 \times 10^{-7}(4720^2 - 3093^2) = 1.257 \tag{3.2.12}$$

$$\tau_{\theta_{max}} = 0.243 \times 4.0687 \times 10^{-7}(4720^2 - 3093^2) = 1.257 \tag{3.2.13}$$

$$\tau_{\psi_{max}} = 8.4367 \times 10^{-9}(4720^2 + 4720^2 - 3093^2 - 3093^2) = 0.2145 \tag{3.2.14}$$

These are expressed in vector form as follows,

$$U^{max} = \begin{bmatrix} 1.257 & 1.257 & 0.2145 \end{bmatrix}^T \tag{3.2.15}$$

The minimum torques were obtained by replacing the motor speed values of ω_4 with ω_2 in equation 3.2.12, replacing ω_1 with ω_3 in equation 3.2.13 and replacing motor speed values ω_1 and ω_3 with ω_2 and ω_4 respectively in equation 3.2.14. This results in negated values of equation 3.2.15.

Therefore the vector of minimum input constraints is as follows,

$$U^{min} = \begin{bmatrix} -1.257 & -1.257 & -0.2145 \end{bmatrix}^T \tag{3.2.16}$$

The rate of input change was obtained iteratively in the MATLAB simulations run in section 3.3. A rate of input change of 60% of the input was found to guarantee stable performance.

Therefore the maximum rate of input change is,

$$\Delta U^{max} = \begin{bmatrix} 0.7542 & 0.7542 & 0.1287 \end{bmatrix}^T \tag{3.2.17}$$

The minimum rate of input change, ΔU^{min} are the negated values of the maximum rate of input change.

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

$$\Delta u_1^{min} \leq \Delta u_1(k) \leq \Delta u_1^{max} \tag{3.2.18}$$

$$\Delta u_2^{min} \leq \Delta u_2(k) \leq \Delta u_2^{max} \tag{3.2.19}$$

$$\Delta u_3^{min} \leq \Delta u_3(k) \leq \Delta u_3^{max} \tag{3.2.20}$$

Equation 3.2.21 expresses the above equations in a compact form.

$$\Delta u_i^{min} \leq \Delta u_i(k) \leq \Delta u_i^{max} \tag{3.2.21}$$

where $i = 1$, ..., total number of inputs.

For instance, choosing a control horizon of three, the future control trajectory at sample instant, k, is denoted by

$$\begin{bmatrix}\Delta U(k) & \Delta U(k+1) & \Delta U(k+2)\end{bmatrix}^T$$

Each element of this control trajectory is comprised of three rates of input change variables and each of them are bound in a similar way to equations 3.2.18 - 3.2.20 above. Therefore, the constraints on the control trajectory were imposed as follows,

$$\Delta u_i^{min} \leq \Delta u_i(k) \leq \Delta u_i^{max} \tag{3.2.22}$$

$$\Delta u_i^{min} \leq \Delta u_i(k+1) \leq \Delta u_i^{max} \tag{3.2.23}$$

$$\Delta u_i^{min} \leq \Delta u_i(k+2) \leq \Delta u_i^{max} \tag{3.2.24}$$

The equations above are expressed compactly as

$$\Delta U^{min} \leq \Delta U \leq \Delta U^{max} \tag{3.2.25}$$

In subsection 2.5.7, the compact inequality in equation 3.2.26 was defined to represent the rate of input change constraints.

$$C_{\Delta u}\Delta U \leq d_{\Delta u} \tag{3.2.26}$$

where

$$C_{\Delta u} = \begin{bmatrix}-I \\ I\end{bmatrix}, d_{\Delta u} = \begin{bmatrix}-\Delta U^{min} \\ \Delta U^{max}\end{bmatrix}$$

where the identity matrix, I is a diagonal square matrix with dimension equivalent to the product of nu $\times$ number of inputs.

Using a control horizon of two, in the MPC model, the dimension of the identity matrix, I is six. With the constraints defined earlier in this subsection, equation 3.2.26 is expanded below,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta u_1(k) \\ \Delta u_2(k) \\ \Delta u_3(k) \\ \Delta u_1(k+1) \\ \Delta u_2(k+1) \\ \Delta u_3(k+1) \end{bmatrix} \leq \begin{bmatrix} 0.7542 \\ 0.7542 \\ 0.1287 \\ 0.7542 \\ 0.7542 \\ 0.1287 \\ -0.7542 \\ -0.7542 \\ -0.1287 \\ -0.7542 \\ -0.7542 \\ -0.1287 \end{bmatrix}$$

Similarly, the input constraints expressed in subsection 2.5.7 is presented in equation 3.2.27 below,

$$C_u \Delta U \leq d_u \tag{3.2.27}$$

where,

$$C_u = \begin{bmatrix} -C_2 \\ C_2 \end{bmatrix}, d_u = \begin{bmatrix} -U^{min} + C_1 u(k-1) \\ U^{max} - C_1 u(k-1) \end{bmatrix}$$

With a control horizon of two, matrices C_1 and C_2 become,

$$C_1 = \begin{bmatrix} I \\ I \end{bmatrix}, C_2 = \begin{bmatrix} I & 0 \\ I & I \end{bmatrix}$$

Substituting the constraints obtained in equations 3.2.15 and 3.2.16, the input constraint in equation 3.2.27 was expanded as follows,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \Delta u_1(k) \\ \Delta u_2(k) \\ \Delta u_3(k) \\ \Delta u_1(k+1) \\ \Delta u_2(k+1) \\ \Delta u_3(k+1) \end{bmatrix} \leq \begin{bmatrix} 1.257 - u_1(k-1) \\ 1.257 - u_3(k-1) \\ 0.2145 - u_3(k-1) \\ 1.257 - u_1(k-1) \\ 1.257 - u_2(k-1) \\ 0.2145 - u_3(k-1) \\ -1.257 + u_1(k-1) \\ -1.257 + u_2(k-1) \\ -0.2145 + u_3(k-1) \\ -1.257 + u_1(k-1) \\ -1.257 + u_2(k-1) \\ -0.2145 + u_3(k-1) \end{bmatrix}$$

Putting input and rate of input change equations 3.2.26 and 3.2.27 into one inequality, the equation below was obtained,

$$CC\Delta U \leq d \tag{3.2.28}$$

where

$$CC = \begin{bmatrix} C_{\Delta u} \\ C_u \end{bmatrix}, d = \begin{bmatrix} d_{\Delta u} \\ d_u \end{bmatrix}$$

3.2.3 Quadratic programming parameters

The quadratic programming problem to be minimised is

$$J = \frac{1}{2}\Delta U^T E \Delta U + \Delta U^T F \tag{3.2.29}$$

subject to $CC\Delta U \leq d$

where $E = 2(H^T H + W)$, $F = -2H^T(R_s - Px(k))$.

The H, P and R_s matrices are all dependent on the size of the prediction horizon. The formulation of these matrices are elaborated on in the following section.

3.3 MATLAB Implementation

The MATLAB simulation started with initialising parameters required in the continuous state space model and the controller. These include the moments of inertia for each axis, quadcopter thrust b and drag coefficients k, maximum and minimum motor speeds, control horizon n_u, prediction horizon n_y, number of inputs, number of outputs, number of states, initial state x, initial output y, reference to be tracked, adjusted reference matrix R_s, disturbance and number of simulations N_{sim}.

A flow chart of the MATLAB implementation is shown on the following page. The processes (or steps) in the flow chart are explained in the subsequent subsections.

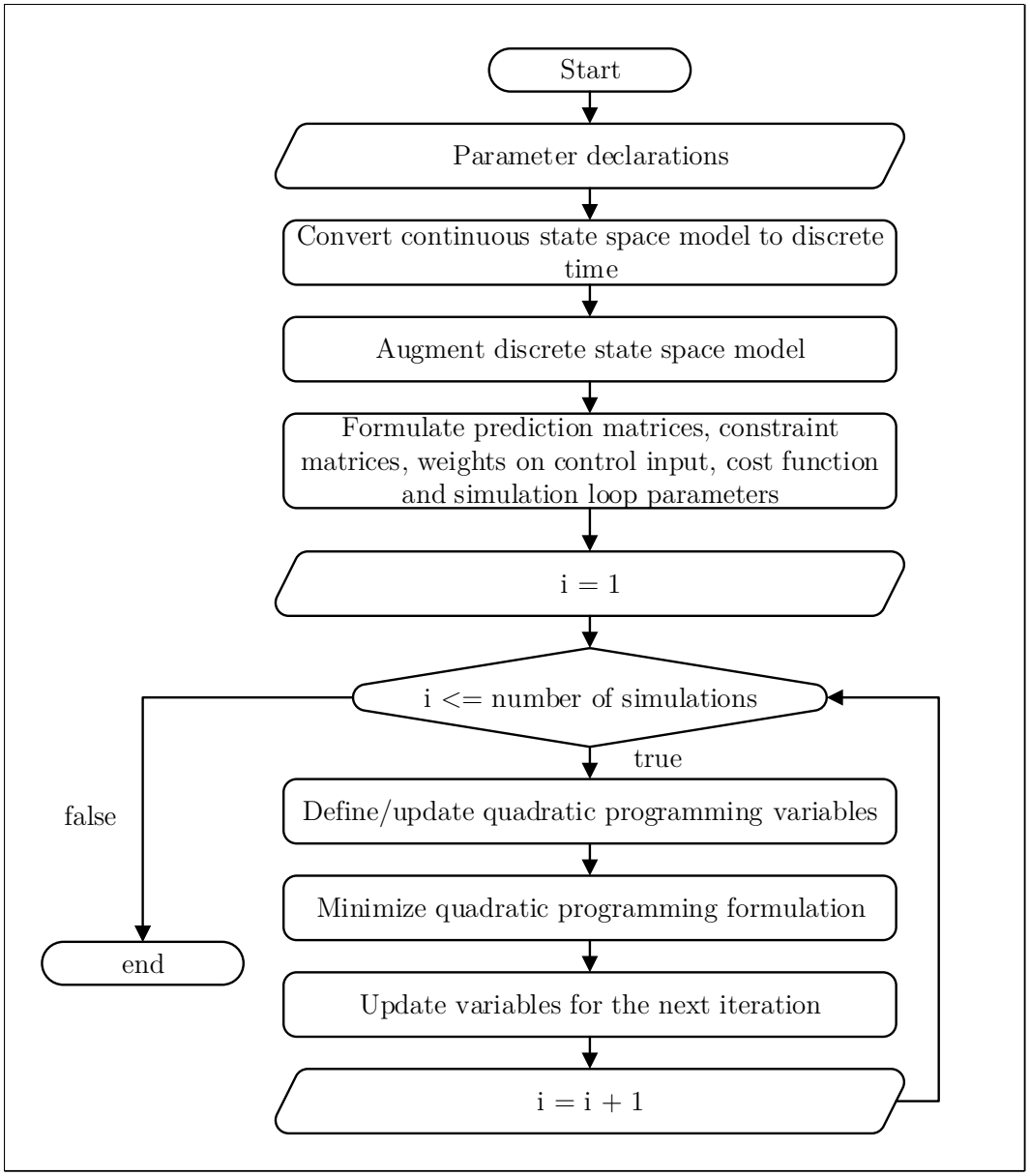


Figure 3.5: Flow chart of MATLAB Implementation

3.3.1 Continuous to discrete time conversion

The MATLAB function `cd2m()` was used to convert the continuous state space model into a discrete one. The use of this function is shown below,

```
[A_d, B_d, C_d, D_d] = c2dm(A, B, C, D, ts);
```

The function takes the continuous state matrices and sample time as arguments and returns discrete state matrices. Matrix `D` is a zero square matrix.

3.3.2 Augmented state space model

The next step was augmenting the discrete state space model. A custom function, `augment_mimo()`, was written to perform this conversion. The function arguments and return values are shown in the equation below.

```
[A_aug, B_aug, C_aug]= augment_mimo(A_d, B_d, C_d, no_of_states,
  no_of_inputs, no_of_outputs);
```

Afterwards, the controllability of the augmented model was confirmed.

3.3.3 Prediction matrices and constraints

The prediction matrices were obtained by using another custom `MATLAB` function, `mpc_predictions_output()`. It takes the augmented state space matrices, control and prediction horizon as arguments. The function returns the output matrices `H` and `P` as defined in subsection 2.5.6.

```
[H, P] = mpc_predictions_output(A_aug, B_aug, C_aug, ny, nu);
```

The constraints `umax`, `umin` and `dumax` were calculated as done in subsection 3.2.2. The constraint matrices and vectors were formulated using the function below,

```
[CC, dd, dupast] = constraints_mimo(Dumax,umax,umin,
no_of_inputs,nu);
```

3.3.4 Input weights and simulation loop parameters

Recall the quadratic programming problem to be minimised is,

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

subject to $CC\Delta U \leq d$

where $E = 2(H^T H + W)$, $F = -2H^T(R_s - Px(k))$.

The input weight W is a square diagonal matrix with its dimension in correspondence with the size of the control horizon. For example, with three inputs and a control horizon of two, W will be a 6×6 matrix with the last three diagonal elements being a repetition of the first three.

E and F were the final parameters required to perform the quadratic programming minimisation. E is defined before the minimisation loop while matrix F is in the loop as it is updated at the start of each loop iteration.

The augmented state vector `Xf`, defined in subsection 2.5.4, was also initialised before the minimisation loop as a zero vector with the number of rows equivalent to the number of rows of either augmented the state matrix `A_aug` or `B_aug`.

3.3.5 Minimisation loop

The `for` loop variable, `i`, was initialised with the value of 1. Thereafter, the quadratic programming variable `F` and the constraint vector `d` were updated before being passed into the Hildreth's programming function `QPhild()` defined below.

```
[DeltaU, Unconst] = QPhild(E, F, CC, d);
```

`DeltaU` was the solution to the quadratic programming problem. `Unconst` was the unconstrained solution to the problem — this was added for comparison purposes. `DeltaU` was the control trajectory of input actions where the number of columns of this matrix is equivalent to the size of the control horizon; only the first column of input values is implemented on the system.

After extracting the first column from `DeltaU`, the next step was to update the model for the next loop iteration. This was executed in the following lines,

```
xh(:,i+1) = A_aug*xh(:,i) + B_aug*deltau';
yh(:,i) = C_aug*xh(:,i+1) + dist(:,i);
Xf = xh(:,i+1);
```

where `deltau` was the first column of input values and `xh` was a state vector with the same dimensions as the `Xf`, and `dist` was a disturbance added to the output `yh`.

In order to obtain the current input vector to be sent to the quadcopter, `deltau` is added to the previous input vector as shown below,

```
u = u + deltau';
```

The variables needed to plot the graphs were updated as well. This process was repeated until the loop condition was false. The graphs of the simulation are presented in the next chapter. The full `MATLAB` code is included in appendix D.

The software-in-the-loop (SITL) implementation is discussed in the following section.

3.4 SITL Implementation

3.4.1 Microsoft Visual Studio

Before the MPC controller was implemented on a simulated quadcopter, the controller had to be programmed in C++; the programming language that the PX4 firmware is written in.

The C++ code was written from scratch, opting against using the MATLAB coder toolbox to convert the MATLAB code into C++. This approach offers better comprehension and flexibility in code debugging. To verify that both the MATLAB and C++ MPC controllers were functionally identical, both controllers were initialised with the same variables: state space matrices, vectors, references, constraints and disturbances. After obtaining the same output with the MATLAB controller — giving tolerance for differences in the number of significant figures — the C++ MPC controller was moved onto the PX4 firmware to make it compatible to run the SITL simulation.

The Microsoft Visual Studio 2015 Integrated Development Environment (IDE) was used in programming, running and testing the controller in C++.

3.4.2 Eigen C++ library

From the subsections 2.5.4 and 3.2, it is evident that the model predictive controller is matrix based. The Visual Studio IDE does not have an inbuilt matrix library to perform matrix and vector definitions and computations fundamental to MPC, therefore an external library had to be included.

Two linear algebra C++ libraries were considered for this purpose: Armadillo C++ library and Eigen C++ library. Eigen was chosen over Armadillo because of its compatibility with the PX4 firmware.

The PX4 developers implemented a basic matrix library that does not extend to defining large matrices (and vectors) and performing complex matrix decompositions; this further strengthens the motivation for including eigen. The Eigen C++ library is a header only library which implies that the user is only required to download the eigen source files and ensure that these files are discoverable by the compiler.

Eigen version 3.3.4 was used in this thesis.

3.4.3 Matrix and vector definitions

The Visual Studio IDE was installed on a 64-bit Dell Latitude E640 that runs on an Intel Core i5 CPU M520 at 2.40 GHz $\times$ 4 with 4GB RAM. There were no memory issues implementing the MPC angular rates controller on the Visual Studio IDE, but with controller target being the Pixhawk — with 256kb RAM/2mb Flash memory — it was necessary to take steps to optimise the C++ MPC controller code before implementation on the Pixhawk.

The following subheadings discuss these steps.

3.4.3.1 Float vs double precision

The Pixhawk autopilot runs on a 32-bit ARM Cortex M4 core processor with FPU. FPU stands for Floating Point Unit and it is the part of the processor used for performing floating point calculations. This FPU is used for single precision floating point values and not double precision floating point operations. Therefore all the floating point operations used in the controller were declared as type **float** and not double. The code snippet below shows how matrices (and vectors) other than the default two, three or four dimensional matrices were defined. This snippet defines a 10×5 float matrix.

```
Matrix<float, 10, 5> A;
```

3.4.3.2 Fixed vs dynamic matrix and vector sizes

In Eigen, matrices and vectors are classified to either have a dynamic or fixed size. An example of how these types are defined in eigen is shown below,

```
Matrix3d A;
VectorXf B;
```

Matrix A is of type double with a fixed size as the 3 stands for a dimension of 3×3. Meanwhile vector B is float dynamic sized vector where the dimension is unknown until runtime. Using fixed sizes is very beneficial to performance, as it enables the Eigen library to avoid dynamic memory allocation and to unroll loops. Dynamic memory allocation entails reserving memory as needed at program runtime as opposed to reserving a fixed amount of memory ahead of time. Loop unrolling is a loop transformation technique that aims to optimise the execution time of a program (Carminati *et al.*, 2017).

All the matrices and vectors used in the controller were declared as float and fixed.

3.4.3.3 Passing matrices and vectors by reference

Due to the potentially large matrices and vectors that are used in MPC, the preferable approach of passing arguments to a C++ function was by reference. Employing this approach ensures greater time and memory efficiency compared to passing arguments by value. Because these arguments are not copied, their values were used and altered directly.

3.4.3.4 Optimisation flags

Visual studio has optimisation flags that can be added to inform the compiler to optimise the respective code for minimum size, maximum speed, general optimization and more. As computation time is important, the `-O2` flag, used to optimise the code for maximum speed, was added.

The C++ code snippet below demonstrates how a matrix is declared, initialised and how a matrix element is accessed in Eigen. This procedure is also applicable to vector declarations and initialisations.

```
#include <iostream>
#include <Eigen/Dense>
#include <unsupported/Eigen/MatrixFunctions>

using namespace std;

int main()
{
    Matrix3f m;
    m << 1, 2, 3,
         4, 5, 6,
         7, 8, 9;

    /* Displaying the matrix element on the
     second row and third column on the
     console */

    cout << m(1,2);

    return 0;
}
```

3.4.4 Building the PX4 firmware

On successfully implementing the MPC angular rates controller in C++, in Microsoft Visual Studio, the next step was to move the relevant variables, matrices, vectors and functions to the attitude control module on the PX4 firmware.

This phase of the controller implementation was done in a Linux environment using the Ubuntu 16.04 LTS operating system dual booted with Windows 10 on the same laptop.

The default firmware was cloned from the PX4 firmware GitHub repository to a directory on the laptop. Qt Creator was the IDE used in editing the source file of the attitude module in order to include the matrices, vectors, functions and other variables needed to run the MPC angular rates controller. These

parameters were declared in the appropriate class to ensure their accessibility within the file.

As established earlier in this chapter, the PX4 firmware is made up of various modules which are integrated together. CMake is an open-source, cross-platform family of tools used to generate definition files (CMake) for each module which are built by a toolchain — a set of tools used in compiling, building, debugging (and more) a project — into the PX4 firmware. Each module folder contains its source code and a cmake definition file that informs the toolchain on how to build the module; the cmake definition file is always saved as CMakeLists.txt. The code snippet below is an example of the contents of a cmake file for creating a PX4 firmware module.

```
px4_add_module(
MODULE      modules_test_app
MAIN        test_app
STACK_MAIN  2000
STACK_MAX
COMPILER_FLAGS
SRCS
            test_app.cpp
DEPENDS
            platforms_common
)
```

where,
`MODULE` - unique name of the module
`MAIN` - entry point of the module on the flight stack, if not given it is assumed to be library
`STACK_MAIN` - size of stack for main function
`STACK_MAX` - maximum stack size of any frame
`COMPILE_FLAGS` - compile flags for compiler
`SRCS` - module source files
`DEPENDS` - targets which the module depends on

If no `STACK_MAX` is defined, the `STACK_MAX` is set as `STACK_MAIN`.

Building the PX4 firmware was done from the Ubuntu terminal window. The following commands were executed in the order shown below to build the firmware:

```
cd Firmware
make px4fmu-v2_default
```

The first command was used to navigate to the directory where the firmware folder is located. The second command built the firmware. `px4fmu-v2_default`

is the build target for the type of Pixhawk board in use. The terminal window outputted error messages if the build was unsuccessful.

The next step was to run the PX4 firmware on a simulated quadcopter.

3.4.5 jMAVSim

jMAVSim is an open-source java based multicopter simulator, that enables a user to fly a vehicle running the PX4 firmware in a simulated environment.

The setup procedure for using jMAVSim is explained in appendix F. After jMAVSim was successfully setup, the termimal running the simulator was closed. A new terminal was reopened and directory commands were used to navigate to the directory containing the firmware folder. The following command was executed in the terminal to run SITL using jMAVSim:

```
make posix_sitl_default jmavsim
```

On successful run, the PX4 shell console was opened which enables the user to type in commands to take off and land the simulated quadcopter. Specific keys on the keyboard were used to steer and control the airborne quadcopter. The command below was entered into the console to take off the quadcopter:

```
pxh> commander takeoff
```

The ground station software, QGroundControl (QGC), was connected to jMAVSim to fly a mission with the quadcopter or use a joystick instead of the keyboard to control the quadcopter.

These flight missions in QGC were waypoints that were setup for the simulated quadcopter to fly through to monitor the controller performance as the quadcopter flies through each point. QGroundControl is discussed in more detail in the following subsection.

Figure 3.6 shows a quadcopter in the jMAVSim simulator environment.

Figure 3.6: Quadcopter in jMAVSim environment

3.4.6 QGroundControl (QGC)

QGroundControl is a cross-platform application — supported on Windows, Linux, Mac OS, Android, iOS — that provides full flight control and mission planning for drones that communicate using the MAVLink protocol. It is also used to setup vehicles powered by PX4 or ArduPilot firmwares (QGroundControl). Setting up includes uploading firmware to the board, choosing the vehicle configuration (airframe), calibrating onboard sensors, establishing a connection between the radio transmitter and receiver, and more. The QGC homescreen is shown below.

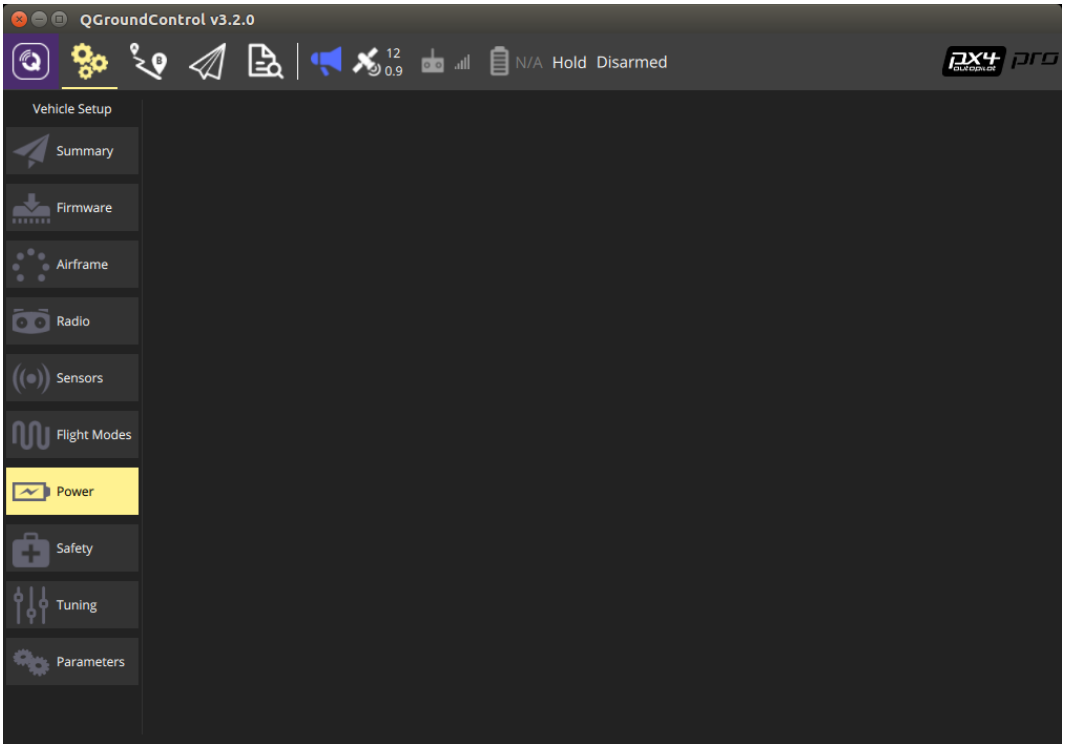


Figure 3.7: QGroundControl homescreen

The SITL simulation of the PX4 firmware with the MPC angular controller was carried out using jMAVSim, to simulate the quadcopter running the firmware, and QGC to create waypoints for the simulated quadcopter to fly through. The flight logs from the SITL simulation were logged and downloaded using QGC.

jMAVSim and QGC were both setup and used in Ubuntu 16.04 LTS operating system on the same laptop - Dell Latitude E640. The MPC C++ code for the SITL implementation is available in appendix E.

3.5 Flight Testing

After successful SITL simulations, the next step was to upload the modified PX4 firmware onto the Pixhawk for flight testing. The Pixhawk was connected to the computer via a USB cable. A terminal window was opened and the directory containing the PX4 firmware is navigated to. Finally, the following command was executed in order to build and upload the firmware to the Pixhawk:

```
make px4fmu-v2_default upload
```

Running this command in the Ubuntu terminal immediately resulted in an error message stating that the stack had overflowed by a certain number of bytes and a subsequent crash of the Pixhawk. Stack overflow occurs when a program tries to use more memory space than is available on the call stack. In this scenario, overflow occured when a function or variables in the attitude control module tried to use more memory than was allocated in its CMakeLists.txt file.

The following subsections explain the steps taken to rectify this error.

3.5.1 FTDI cable

The Pixhawk has a system console that grants low-level access for debugging output and analysis of the system boot process (Console). Connection to the system console is achieved using a Future Technology Devices International (FTDI) cable where one end of the cable is connected to the serial port on the Pixhawk and the other to a USB port on the computer.

The FTDI cable enables the user to view details of the modules running on the flight stack, on the system console in the Ubuntu terminal. These details include respective stack sizes of each module and their usage, in addition to displaying the available memory on the Pixhawk.

The FTDI cable was mainly used to reboot the Pixhawk after crashing from an attempted firmware upload. Reboot was achieved by typing **reboot** in the terminal and pressing the enter key. The errors encountered in uploading the firmware were also visible on the terminal screen after executing:

```
make px4fmu-v2_default upload
```

3.5.2 Stack size

The intuitive first step in addressing the stack overflow error message was to increase the stack size of the module causing the error. The stack size was increased by incrementing the size of the `STACK_MAIN` variable in the attitude control CMakeLists.txt file. Memory is limited, therefore there is a constraint on how much memory can be allocated to the attitude control module. It was noticed that the error persisited even after an increase in the stack size. Therefore, attention was shifted from stack size to speed of module compilation and code efficiency. These approaches are discussed in the following subsections.

3.5.3 Optimisation flags

Similar to the visual studio IDE, the compiler on the Pixhawk has provisions for certain flags to be added to maximise speed, minimise code size, general

optimisation and more. The `-O3` flag, used to maximise speed, was added as a compiler flag in the cmake definition file for the attitude module.

3.5.4 Hardcoding output matrices, constraints matrices and vectors

The Dell Latitude laptop has a significantly larger memory capacity compared to that on the Pixhawk hardware. Therefore, running custom functions used to generate output matrices, constraint matrices and vectors were executed smoothly. In order to reduce the computational load on the pixhawk, these matrices and vectors were hardcoded, after the desired control and prediction horizons were chosen.

3.5.5 Static keyword

In addition to hardcoding matrices and vectors, these variables were preceded with the keyword **static**. The static keyword ensures that the accompanying variable is declared and stored only once in memory. The initialised value remains constant until it is modified within its function or class. The keyword ensures that the defined values persist until the program is terminated — when the Pixhawk is powered off.

3.5.6 Disable unused modules and libraries

The PX4 firmware can also be used to control VTOL vehicles, fixed wing aircraft and rovers. Some of the modules associated with these vehicles are activated by default. These modules were disabled to conserve more memory on the Pixhawk. Furthermore, other unused modules, libraries and device drivers were disabled.

3.5.7 Size of control and prediction horizons

Two crucial parameters in designing an MPC controller are the control and prediction horizons, as the matrices and vectors used are influenced by the size of these horizons. After several iterations, a sample time of 0.2 s was chosen for the discrete model for the MPC angular rates controller, to be implemented on the Pixhawk. Initially, a control horizon of three was used with a prediction horizon of eight. Building the firmware with matrices and vectors in correspondence with these horizons resulted in build errors. Decreasing the prediction horizon down to two also resulted in an unsuccessful build, and a horizon of one underutilises the predictive ability of the MPC controller. Reducing the control horizon to two with a prediction horizon of five finally resulted in a successful firmware build.

The next chapter presents the results obtained from varying the size of the prediction horizons.

All of these steps contributed in successfully uploading the modified PX4 firmware, with the MPC angular rates controller, onto the Pixhawk. Before flight testing outdoors, the quadcopter was flown in the Structures laboratory at the Mechanical Engineering department, Stellenbosch University, with a net over the area to be flown, in order to dampen any quadcopter crashes as parameters such as the input weights and the rate of input change constraints were being tuned and tested.

Figure 3.8 below shows the quadcopter used in the laboratory.

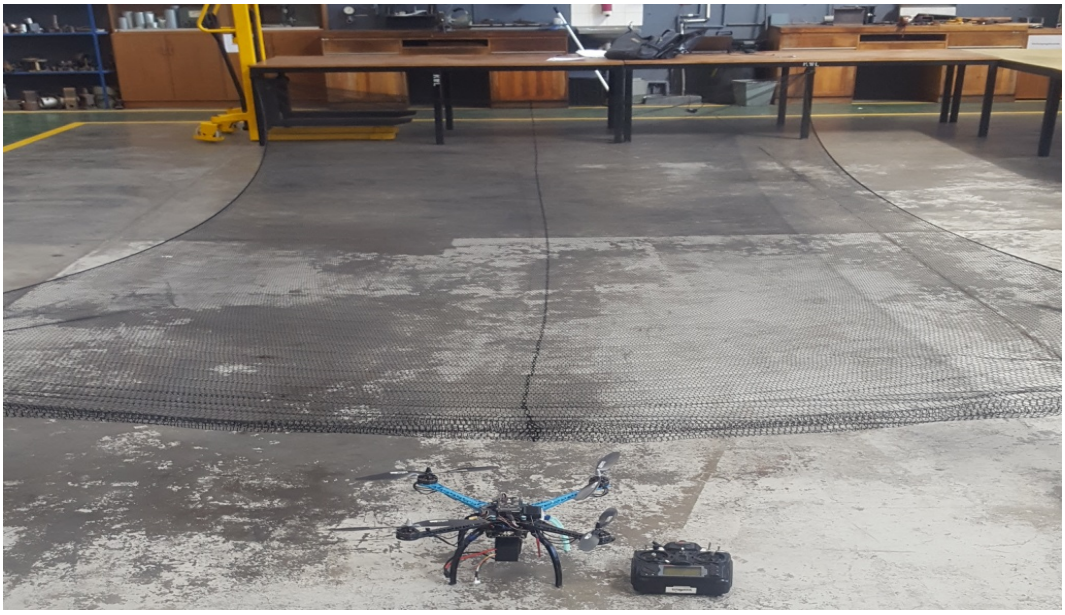


Figure 3.8: Indoor flight testing

Before flying the quadcopter — indoor or outdoor — the following checklist had to be ticked off in the order below,

- ☒ Upload PX4 firmware onto Pixhawk.
- ☐ Charge quadcopter batteries.
- ☐ Calibrate sensors using QGroundControl.
- ☐ Calibrate radio transmitter to receiver via QGC.
- ☐ Ensure the right propellers are used and are tightly fastened.
- ☐ Connect the Pixhawk power module cable to the battery.
- ☐ Turn off the safety switch.

☐ Arm quadcopter for flight via the radio transmitter.

The table below lists the hardware components of the quadcopter used for flight testing.

Table 3.2: Summary of quadcopter hardware

Quadcopter hardware components		
S/N	Part	Component used
1	Frame	s500
2	Propellers	12×3.8 Quanum carbon fiber
3	Motors	EMAX MT3506 650kv
4	ESCs	Hobbywing Platinum 30A
5	Battery	14.8v 5000 mah LIPO Turnigy
6	Controller	Pixhawk
7	Transmitter	Spektrum DX7 radio
8	Receiver	Spektrum DSMX receiver

The results obtained from the controller implementations discussed in this chapter are presented in the following one.

Chapter 4

Simulation and Experiments

The results obtained from running the simulations in MATLAB are presented in section 4.1.Section 4.2 presents the results obtained from the SITL simulations. Finally, the flight test results from flying the quadcopter running the modified PX4 firmware are plotted and discussed in section 4.3.

4.1 MATLAB Simulations

The MPC angular rates controller designed in the previous chapter was simulated in MATLAB in order to test its performance. Sinusoidal functions were used as the references to be tracked by the controller. Its performance was observed by plotting the model output (or response) together with the references, by varying the control and prediction horizons. The weights on the control inputs were tuned to adjust to these variations. Furthermore, the applicable matrices and vectors, dependent on the size of the control and prediction horizons, were modified for compatible matrix and vector computations.

The sinusoidal references tracked by the MPC angular rates controller were the fourier series below,

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} sint + \frac{cos(3t)}{2} \\ sint + \frac{cos(3t)}{2} \\ sint + \frac{cos(2t)}{2} + \frac{sin(3t)}{3} \end{bmatrix}$$

The variables $\dot{\phi}$, $\dot{\theta}$ and $\dot{\psi}$ are the angular rates for the roll, pitch and yaw axes respectively measured in rad/s. t is the simulation time step which is bounded between 0 and 20; with increments of 0.2.

The table below shows the initialisations of the parameters needed for the MATLAB simulations, where $\begin{bmatrix} p & q & r \end{bmatrix}^T = \begin{bmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T$.

Table 4.1: MATLAB MPC simulation parameters for $n_u = 3$, $n_y = 6$

n_u	n_y	**p, q**	**r**	**disturbance**
3	6	sint + $\frac{cos(3t)}{2}$	sint + $\frac{cos(2t)}{2}$ + $\frac{sin(3t)}{3}$	(rand(3,101)*2 -1)*0.5

The results obtained from the simulations are plotted below.

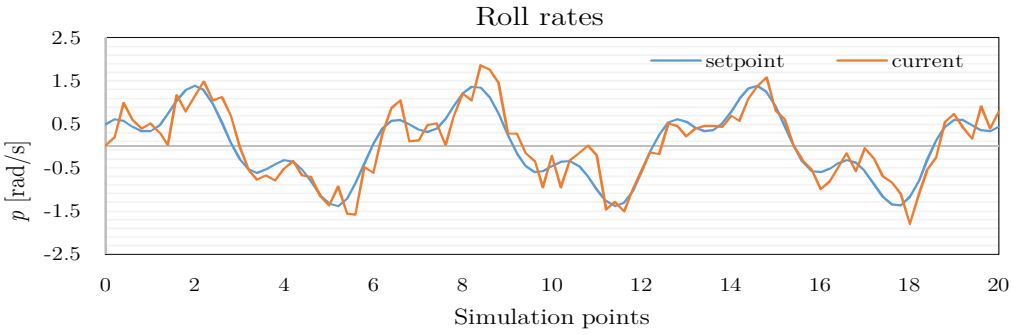


Figure 4.1: MATLAB roll rates for $n_u = 3$, $n_y = 6$

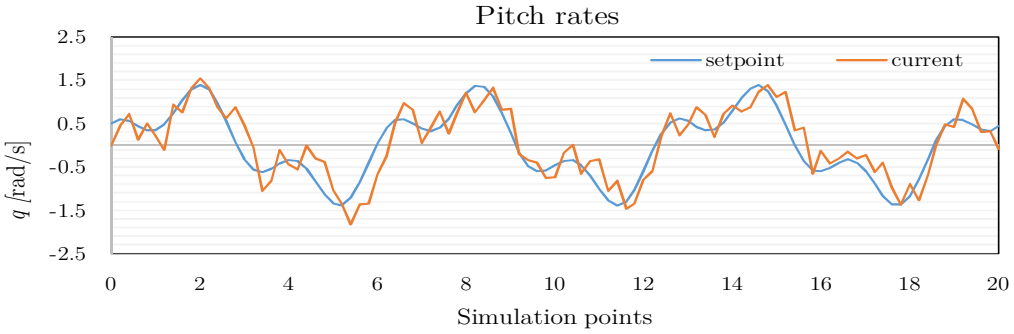


Figure 4.2: MATLAB pitch rates for $n_u = 3$, $n_y = 6$

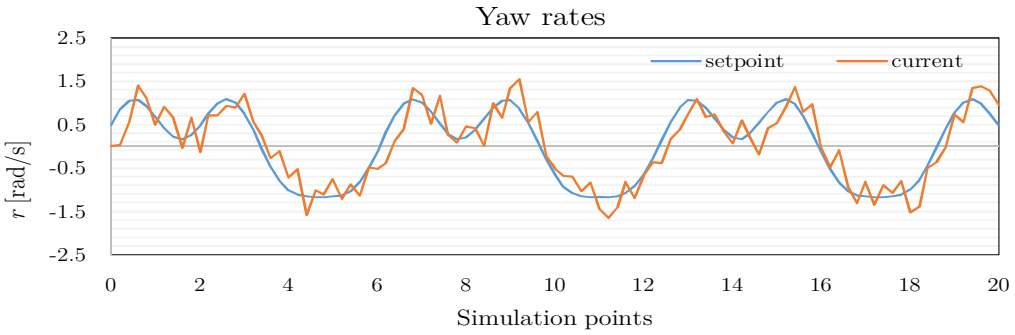


Figure 4.3: MATLAB yaw rates for $n_u = 3$, $n_y = 6$

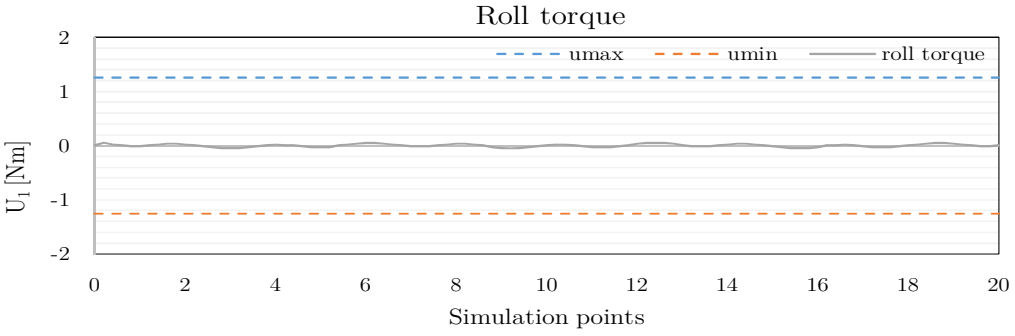


Figure 4.4: MATLAB roll torque for $n_u = 3$, $n_y = 6$

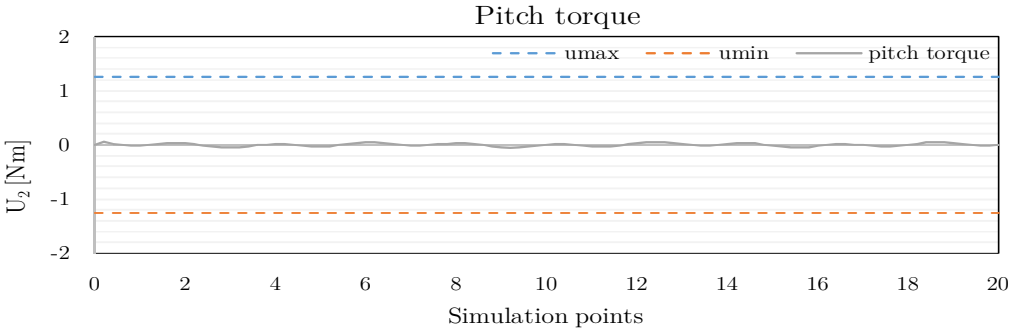


Figure 4.5: MATLAB pitch torque for $n_u = 3$, $n_y = 6$

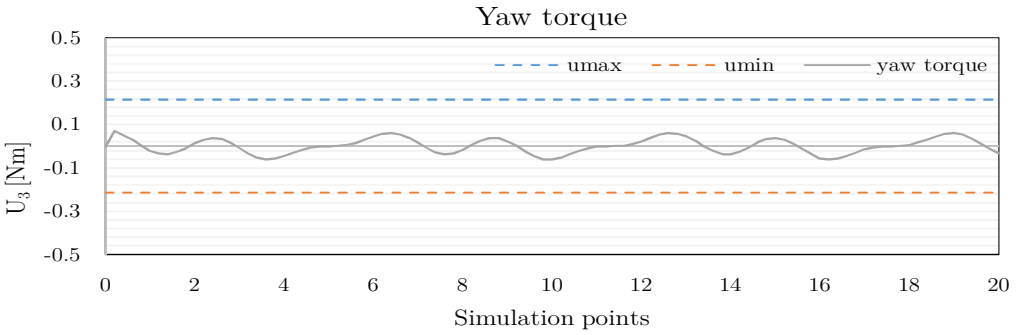


Figure 4.6: MATLAB yaw torque for $n_u = 3$, $n_y = 6$

In table 4.1, a disturbance parameter was initialised. It is a weighted randomised value, added to the output vector of the quadcopter model, to replicate ambient disturbance and sensor noise present in real world quadcopter flights.

It is apparent from the rates plots that the controller was able to attain close reference tracking despite the additional disturbance parameter.

The dotted horizontal lines in each of the torque (or input) plots represent the constraints that bound the respective input control signals. By tuning the input weight matrix W, favourable reference tracking was achieved, with the torque signals within their respective constraints.

Further MATLAB simulations were carried out and the results of these simulations are presented in appendix G.

4.2 Software-in-the-loop (SITL)

The SITL simulation results were obtained by flying the simulated quadcopter — running the modified PX4 firmware — in jMAVSim through flight mission waypoints created in QGroundControl. In figure 4.7 below, the left panel shows the quadcopter in the simulated jMAVSim environment and the right panel shows the flight mission waypoints in QGC.

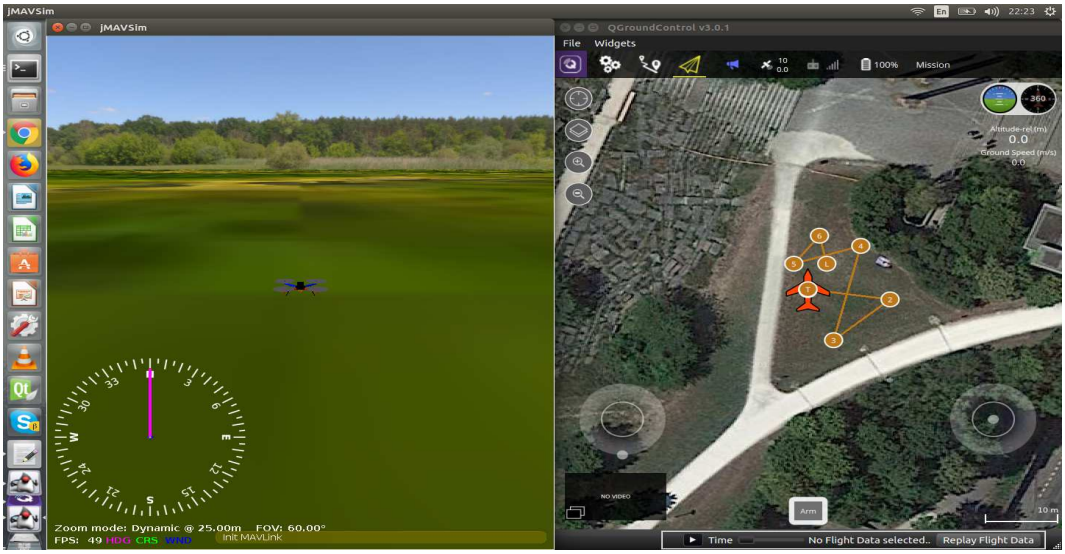


Figure 4.7: Left panel: Quadcopter in jMAVSim, right panel: Flight mission waypoints in QGC

The label **T**, in the right panel of the figure above, indicates the take-off position and **L**, the landing position. Between take-off and landing, the quadcopter flew in ascending numerical order of the points shown above. The flight mission was initiated by clicking the **Arm** button at the bottom of the QGC screen.

The initial SITL flight mission above was executed with a control horizon of two and a prediction horizon of five. After take off, it was noticed that the quadcopter oscillated with high frequency, about its centre of gravity, as it flew to the first waypoint and eventually crashed. The tentative explanation for this response was that the sample time of the controller was faster than the response time of the quadcopter.

To test this conjecture, the minimisation procedure was slowed down by enclosing the Hildreth's quadratic programming function and state space model parameters, within a loop to run at least twice before outputting the input (or torque) commands to the mixer module. Running the loop three times resulted in the quadcopter drifting off course, after taking off, and subsequently crashing. Whereas, a loop with two iterations resulted in a relatively smooth flight through each mission waypoint, and the mission ended with a safe landing.

The SITL flight simulations were resumed after making this modification and the flight data was accordingly plotted. QGC was used to download the flight data for each flight mission executed.

The control and prediction horizons used in the flight data plotted below were two and four respectively.

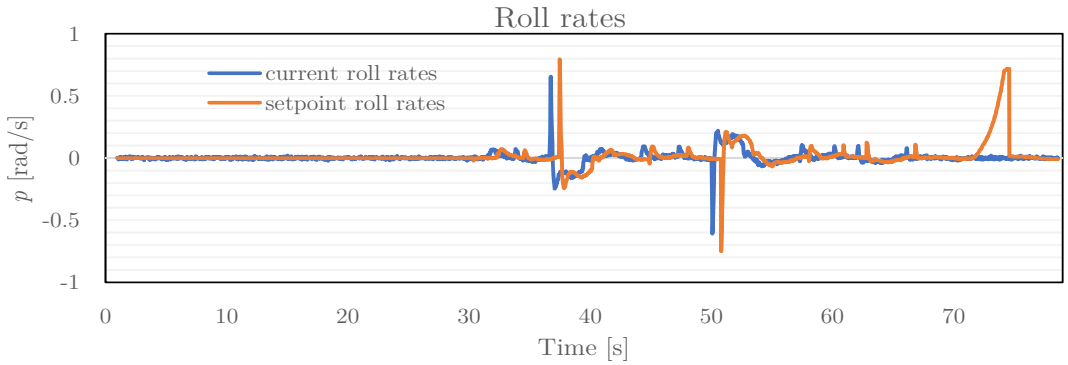


Figure 4.8: SITL roll rates for $n_u = 2$, $n_y = 4$

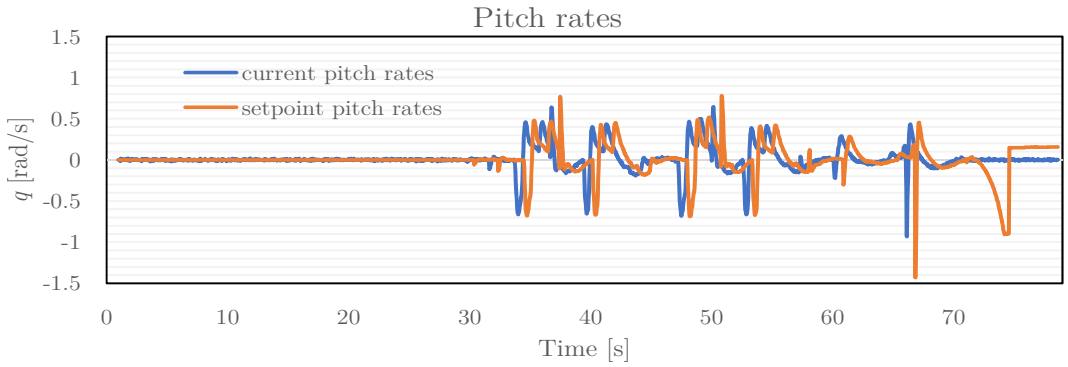


Figure 4.9: SITL pitch rates for $n_u = 2$, $n_y = 4$

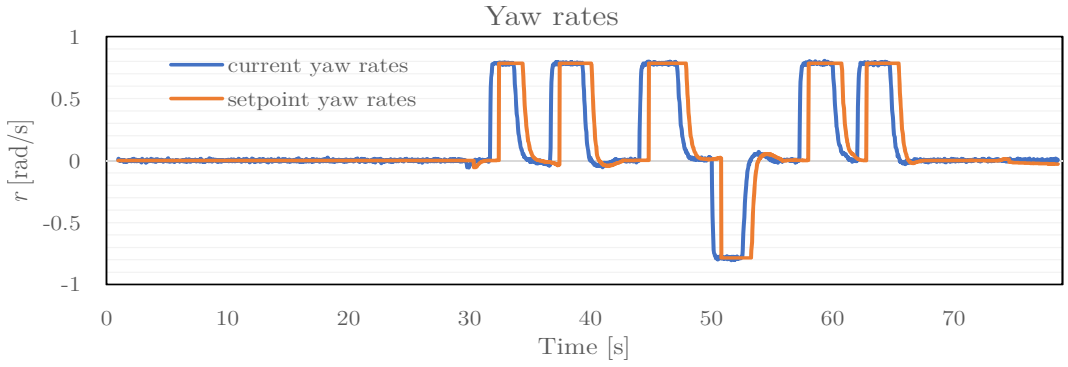


Figure 4.10: SITL yaw rates for $n_u = 2$, $n_y = 4$

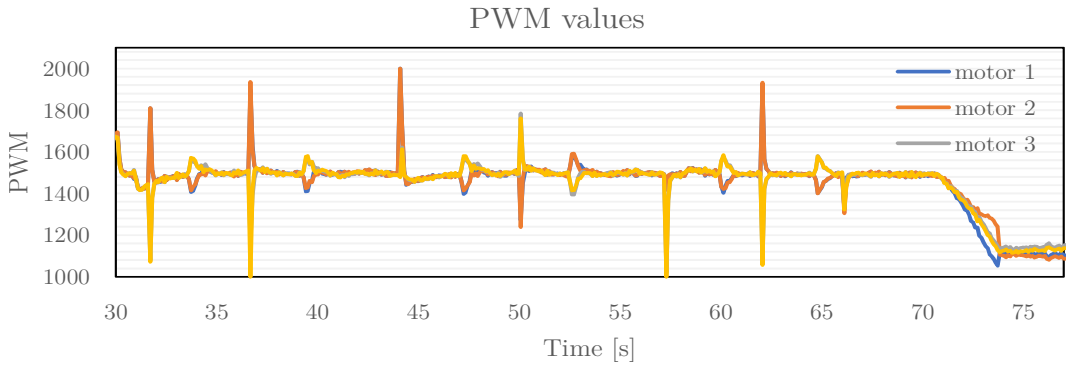


Figure 4.11: SITL PWM values for $n_u = 2$, $n_y = 4$

Similarly, jMAVSim includes disturbance in the simulator, to replicate real world flight conditions and it is evident from the rate plots that the MPC rates controller was able to achieve close reference tracking of the rates setpoint.

As established in subsection 3.1.1, the output from the attitude and rates controller is sent to the mixer module. The mixer module converts these signals into motor commands to be sent to the quadcopter; for real flight, these commands are sent to the ESCs to regulate the speed of each quadcopter motor. The motor commands from the mixer module are Pulse Width Modulation (PWM) values which vary between 1000 (the minimum) and 2000 (the maximum). In figure 4.11, it can be observed that these PWM values are well within the PWM limits from take-off to landing.

Appendix G contains plots of other flight missions that were executed.

4.3 Flight Tests

The measures outlined in section 3.5 were effected to ensure a successful firmware upload onto the Pixhawk. Furthermore, the loop modification discussed in the previous section was made, and the checklist listed at the end of section 3.5 was ticked off before indoor flight testing commenced. The input weights and rates of input change constraints were tuned indoors, in the Structures laboratory at the Stellenbosch University Mechanical engineering department, before the quadcopter was flown outside.

The following results were obtained for an outdoor flight using a control horizon of two and prediction horizon of two.

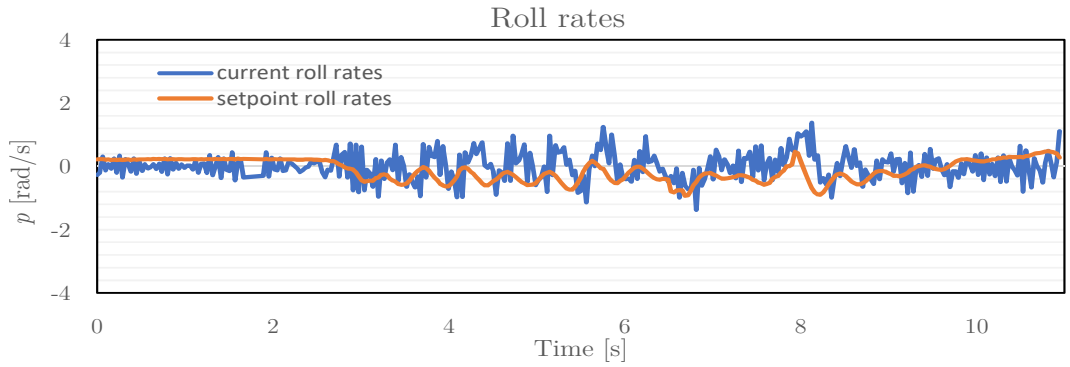


Figure 4.12: Flight roll rates for $n_u = 2$, $n_y = 2$

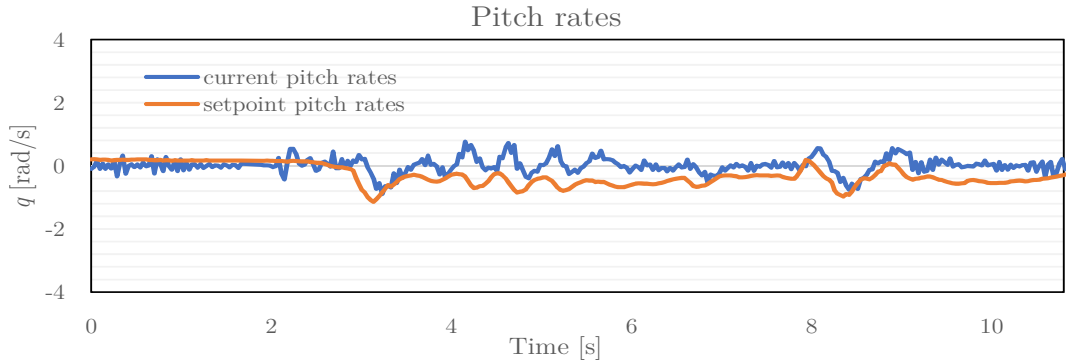


Figure 4.13: Flight pitch rates for $n_u = 2$, $n_y = 2$

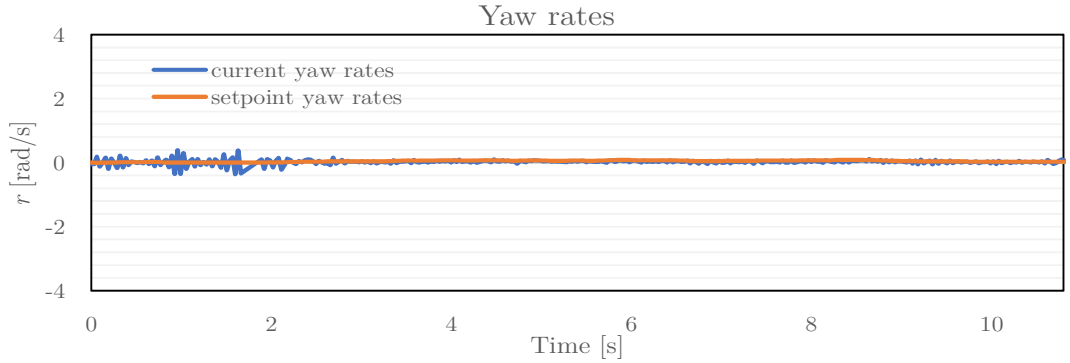


Figure 4.14: Flight yaw rates for $n_u = 2$, $n_y = 2$

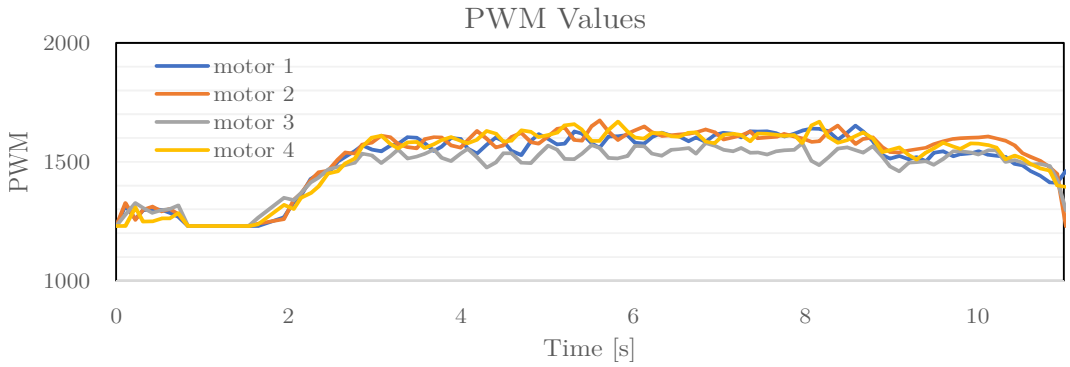


Figure 4.15: Flight PWM values for $n_u = 2$, $n_y = 2$

Real world quadcopter flights are subjected to external disturbances, such as wind, sensor noise, model uncertainty from assumptions made in the modelling process and more. Despite these flight conditions, it can be observed that the MPC angular rates controller, running on the Pixhawk, was able to closely track the setpoint rates. It is also evident that the PWM values sent to the motors were well within the minimum and maximum PWM boundaries.

Additional flight testing results are available in appendix G. The following chapter concludes this thesis.

Chapter 5

Conclusion

This chapter concludes the thesis with a summary of the methodology employed in achieving the aim and objectives outlined in chapter 1, and finally, recommendations are made for future research.

Summary

The PX4 firmware, which runs on the Pixhawk autopilot, is categorised into modules that control different vehicles such as multicopters, VTOL vehicles, fixed wing aircrafts and rovers. To accommodate vehicles of different sizes, PID (or any of variants) controllers are used to control these vehicles. These controllers can be implemented on any vehicle, that the PX4 firmware supports, irrespective of size; albeit the user will have to tune the control parameters. Model predictive control requires a mathematical model of the specific vehicle (or system) to be controlled. Model predictive control has yet to be implemented on the PX4 firmware, the specificity required by MPC is one of the main reasons why.

Thus, the aim of this thesis was to evaluate the feasibility of implementing model predictive control on the PX4 firmware, running on the Pixhawk autopilot, to control the angular rates of a quadcopter. The methodology, visualised in the following figure, shows the approach carried out in this evaluation.

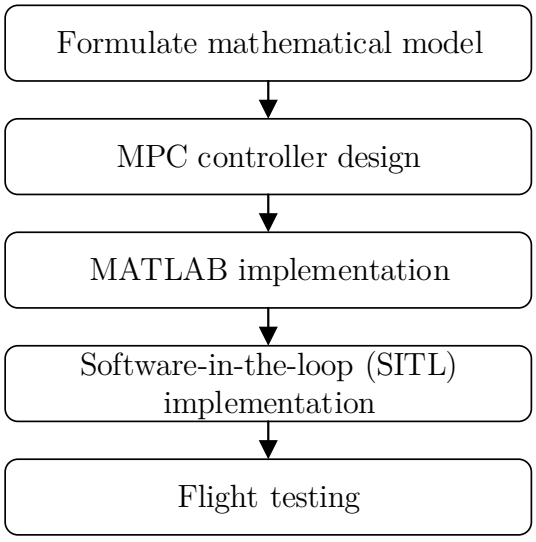


Figure 5.1: Methodology

The challenges encountered in the MPC angular rates implementation was most prominent in the flight testing stage. Stack overflow, among other error messages, was the reoccurring message outputted on the Ubuntu terminal window for each unsuccessful firmware upload attempt. Increasing the stack size of the attitude control module, which contains the angular rates controller, proved ineffective in rectifying the stack overflow error message. Therefore, focus was shifted towards code optimisation, in order ensure a successful firmware upload onto the Pixhawk.

During this code optimisation process, it was realised that a maximum control horizon of two was crucial to a successful upload of the firmware onto the Pixhawk. A quadcopter running the modified PX4 firmware on the Pixhawk was flight tested. The results obtained illustrate that the MPC angular rates controller is able to attain close reference tracking of the setpoint angular rates. But more significantly, these results demonstrate the feasibilty of implementing model predictive control on the PX4 firmware.

Recommendation for Future Research

The results presented in the previous chapter, substantiate the feasibility of implementing model predictive control on the PX4 firmware, to control the angular rates of a quadcopter. Considering the implementation challenges encountered, expanding MPC to other PX4 modules and/or using a control horizon larger than two will require a Pixhawk with a larger memory.

The Pixhawk 1 was the hardware used in this thesis with a processor running at 168 MHz with 256 Kb RAM and 2 Mb flash memory. There are two new versions of the Pixhawk, which are the Pixhawk 3 pro and Pixhawk 4.

The Pixhawk 3 pro has a processor with a flash size of 2 Mb, 384 Kb in RAM running at 180 MHz. Whereas the Pixhawk 4 has a processor speed of 216 MHz, 2 Mb flash memory and 512 kB RAM.

These versions of the Pixhawk are superior to the Pixhawk 1, and will be a good starting point for implementing larger control and prediction horizons, for performance and comparative analysis. In addition, the feasibility of integrating model predictive control into other PX4 control modules can be explored.

Appendices

Appendix A

Optimisation example

Consider the quadratic objective function, J, and its inequality constraints in equation A.0.1

$$J = 0.5x_1^2 - 4x_1 + x_2^2 - 5x_2 + 30 \tag{A.0.1}$$

such that,

$$x_2 - x_1 \geq 2$$
$$0.5x_1 + x_2 \geq 9$$
$$0 \leq x_1 \leq 10$$
$$0 \leq x_2 \leq 10$$

The objective is to minimise J to obtain the x_1 and x_2 values. Two cases will be considered: one with no constraints on the objective function and another where constraints are applied.

Case 1: No constraints

To minimise equation A.0.1, the partial derivates of J are taken for the variables x_1 and x_2. The optimal points of x_1 and x_2 are obtained by equating the partial derivative equations to zero and solving for the corresponding variable. This is shown below.

$$\frac{\partial J}{\partial x_1} = x_1 - 4 = 0$$
$$\Rightarrow x_1 = 4$$
$$\frac{\partial J}{\partial x_2} = 2x_2 - 5 = 0$$
$$\Rightarrow x_2 = 2.5$$

Thus, $x_1^* = 4$, $x_2^* = 2.5$

Figure A.1 below shows a contour plot of the objective function with the optimal x_1 and x_2 values indicated by the dot where the contour lines represent different values of the objective function.

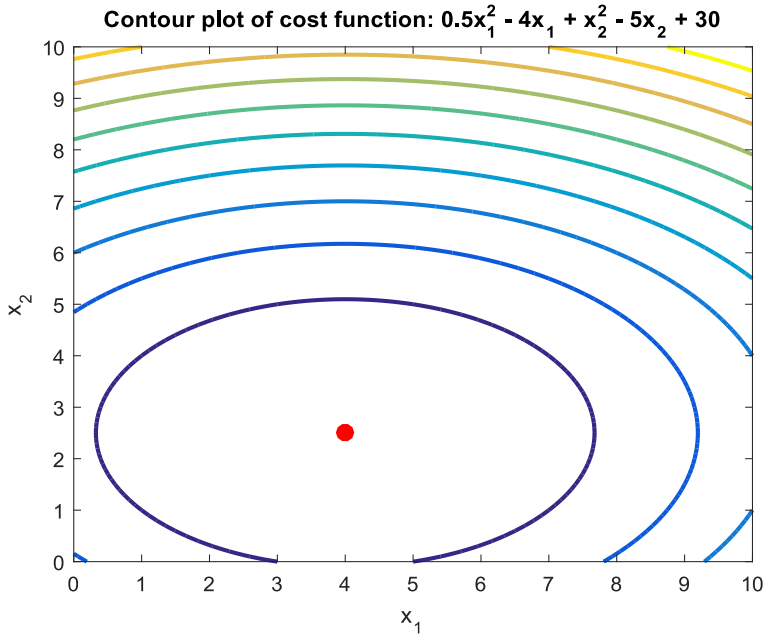


Figure A.1: Contour plot of objective function without constraints

Case 2: With constraints

In order to use the Hildreth's quadratic programming procedure to minimise the objective (or cost) function in equation A.0.1 with constraints taken into account, it is necessary to first set up the objective function in matrix and vector format to satisfy the form in the equations below and then formulate a dual problem as detailed in subsection 2.5.9.

$$J = \frac{1}{2}x^T Ex + x^T F$$

such that,

$$Mx \leq \gamma$$

Thus,

$$E = 2 * \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix}, F = \begin{bmatrix} -4 \\ -5 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & -1 \\ -0.5 & -1 \end{bmatrix}, \gamma = \begin{bmatrix} -2 \\ -9 \end{bmatrix}$$

Figure A.2 shows the contour plot of the objective function. The constraints are represented by the lines on the plot. The solution to minimising

the objective function is restricted to the feasibility region as labelled in the figure. The optimal point is represented by the dot.

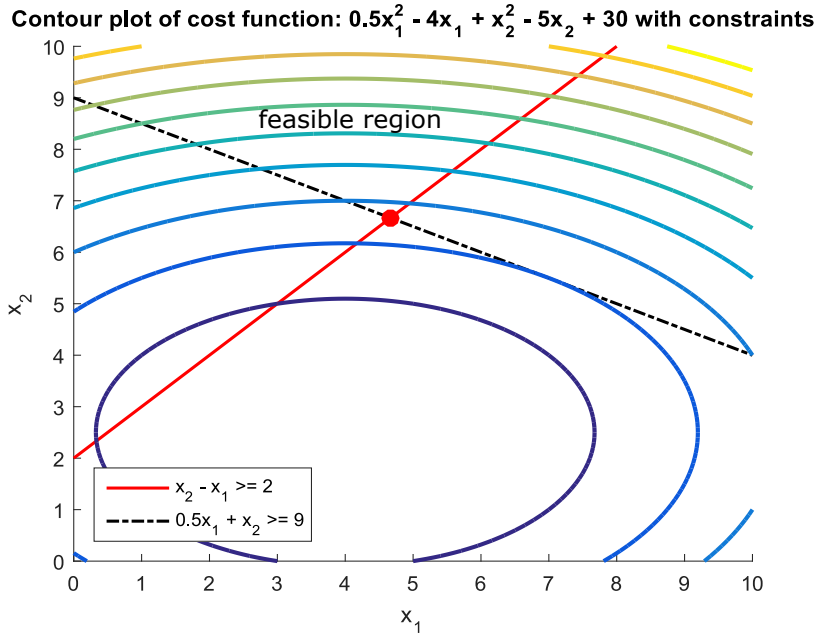


Figure A.2: Contour plot of objective function with constraints

Appendix B

Pixhawk Autopilot Specifications

Figures B.1 and B.2 show the top and side view of Pixhawk 1 flight controller respectively.

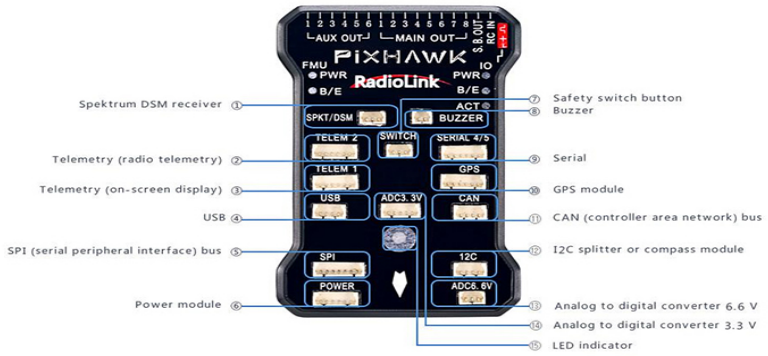


Figure B.1: Top view of Pixhawk hardware with labeled ports (Pixhawk, 2013)

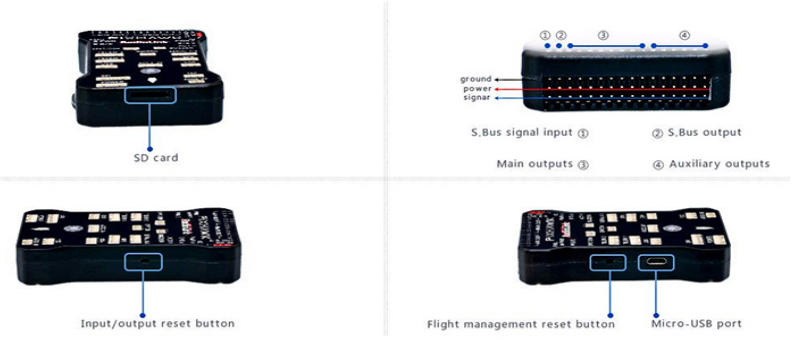


Figure B.2: Labeled side view of Pixhawk (Pixhawk, 2013)

The specifications of the Pixhawk board are given below and are obtained from Pixhawk (2013).

Processor

- 32bit STM32F427 Cortex M4 core with FPU
- 168 MHz
- 256 Kb RAM
- 2 Mb Flash
- 32 bit STM32F103 failsafe co-processor

Sensors

- MPU6000 as main accel and gyro
- ST Micro 16-bit gyroscope
- ST Micro 14-bit accelerometer/compass (magnetometer)
- MEAS barometer

Appendix C

Parameter Determination

The physical properties of the quadcopter are determined in this chapter and the summary of these properties are tabulated at the end.

C.1 Mass

The mass, m, of the quadcopter (with the battery connected) was measured using a digital mass scale. It was found to be approximately 1.587 kg.

C.2 Moment Arm

The moment arm, d, of the quadcopter was determined by measuring the distance from the centre of any propeller to the centre of the quadcopter frame with a measuring tape. The obtained moment arm is labelled in Figure C.1 below.

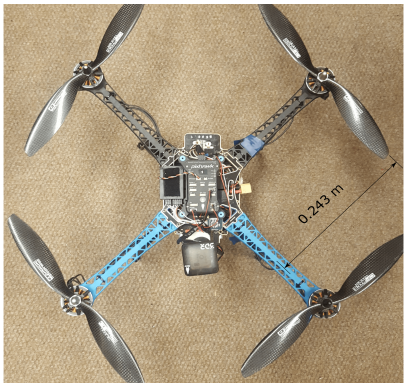


Figure C.1: Moment arm of quadcopter

C.3 Moments of Inertia

Moment of inertia is a measure of resistance to rotate a body about a particular axis (Jardin and Mueller, 2007). An experiment known as the **bifilar pendulum** experiment (Habeck and Seiler, 2016) was carried out as the first step in determining the moment of inertia for the three axes of the quadcopter.

In this experiment, the quadcopter is suspended from a horizontal beam using two strings (filars); where the mass, tension and elasticity of the string are assumed to be negligible. The two strings are of equal length. The length of the string and the distance between them were measured and recorded.

Once the quadcopter has been suspended and motionless, the quadcopter is displaced slightly from its equilibrium position. A stopwatch is started as soon as the quadcopter is displaced and the number of seconds it takes for the quadcopter to complete ten oscillations is recorded. This procedure is repeated two more times in order to calculate an average.

The quadcopter is suspended in the other two axes and the method described above is repeated for each subsequent setup. The experimental setup for the bifilar pendulum for the respective axes is displayed in figure C.2 below. The results from the experiments are presented in tables C.1, C.2 and C.3 below.

Figure C.2: (a) Rotation about x-axis (b) Rotation about y-axis (c) Rotation about z-axis

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

Rotation about X-axis	
Run	Time taken for 10 Oscillations
1	13.09 s
2	12.93 s
3	13.06 s

Table C.2: Time for oscillations about Y-axis

Rotation about Y-axis	
Run	Time taken for 10 Oscillations
1	13.28 s
2	13.44 s
3	13.12 s

Table C.3: Time for oscillations about Z-axis

Rotation about Z-axis	
Run	Time taken for 10 Oscillations
1	11.71 s
2	11.41 s
3	11.68 s

The next step in determining the moment of inertia is to calculate the period for each axis from the oscillations recorded; thereafter this value is substituted into equation C.3.1 obtained from Habeck and Seiler (2016). The period is the time needed to complete one oscillation cycle (Tongue, 2002). This is calculated by dividing the average oscillations in tables C.1, C.2 and C.3 above by 10.

Moment of inertia,

$$I = \frac{mgT^2d^2}{16\pi^2L} \tag{C.3.1}$$

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

C.3.1 Moment of inertia for x-axis, I_{xx}

From table C.1, the average period is calculated as follows,
Average oscillation,

$$= \frac{13.09 + 12.93 + 13.06}{3} = 13.027\,\text{s}$$

Period for one oscillation,

$$= \frac{13.037}{10} = 1.3027\,\text{s}$$

For this experimental setup the following values were measuerd,
$L = 0.928$ m
$d = 0.344$ m
$m = 1.587$ kg

The moment of inertia is calculated below,

$$I_{xx} = \frac{1.587 \times 9.81 \times 1.3027^2 \times 0.344^2}{16 \times \pi^2 \times 0.928}$$

$$I_{xx} = 0.0213\,\text{kgm}^2$$

C.3.2 Moment of inertia for y-axis, I_{yy}

From table C.2, the average period is calculated as follows,
Average oscillation,

$$= \frac{13.28 + 13.44 + 13.12}{3} = 13.28\,\text{s}$$

Period for one oscillation,

$$= \frac{13.28}{10} = 1.328\,\text{s}$$

The values of L, d and m are the same as those used in the x-axis pendulum setup.
The moment of inertia is calculated below,

$$I_{yy} = \frac{1.587 \times 9.81 \times 1.328^2 \times 0.344^2}{16 \times \pi^2 \times 0.928}$$

$$I_{yy} = 0.02217\,\text{kgm}^2$$

C.3.3 Moment of inertia for z-axis, I_{zz}

From table C.3, the average period is calculated as follows,

Average oscillation,

$$= \frac{11.71 + 11.41 + 11.68}{3} = 11.6\,\text{s}$$

Period for one oscillation,

$$= \frac{11.605}{10} = 1.1\,\text{s}$$

For this experimental setup the following values were measuerd,
$L = 1.064$ m
$d = 0.4755$ m
$m = 1.587$ kg

The moment of inertia is calculated below,

$$I_{xx} = \frac{1.587 \times 9.81 \times 1.16^2 \times 0.4755^2}{16 \times \pi^2 \times 1.064}$$

$$I_{zz} = 0.0282\,\text{kgm}^2$$

C.4 Drag Coefficient

The formulas used in this section to determine the drag coefficient and the thrust coefficient in the following section were obtained from Luis and Ny (2016).

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

$$Q = C_d \rho n^2 D^5 \tag{C.4.1}$$

where,

C_d - 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,

$$k = \frac{C_d \rho D^5}{3600} \tag{C.4.2}$$

APC carbon fiber propeller data for 12 inches × 3.8 inches was used in the absence of Quanum carbon fiber propeller data to obtain the value of C_d. This value is available on the APC propeller website. The propeller diameter is 0.304 m. Taking the air density to be 1.225 kg/m^3.

$$k = \frac{0.095493 \times 1.225 \times 0.304^5}{3600}$$

$$= 8.4367 \times 10^{-9}\,\text{Nm/rpm}^2$$

C.5 Thrust Coefficient

From Luis and Ny (2016), the formula for calculating the thrust of a propeller is as follows,

Thrust, T,

$$T = C_t \rho n^2 D^4$$

where,

C_t - 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,

$$1\,\text{rev/s} = 60\,\text{rev/min}$$

Therefore,

$$T = C_t \rho (\frac{\omega}{60})^2 D^4 \tag{C.5.1}$$

where ω is the propeller speed in rpm

C_t is obtained from the APC propeller website. Using the diameter and air density values from the previous section, the thrust coefficient, b, is calculated as follows,

$$b = \frac{C_t \rho D^4}{3600} \tag{C.5.2}$$

$$= \frac{0.14 \times 1.225 \times 0.304^4}{3600} = 4.0687 \times 10^{-7}\,\text{N/rpm}^2$$

Appendix D

MATLAB Code

The MATLAB code used in designing the MPC controller is presented in this chapter. Each section corresponds to a different function required by the main MATLAB script file.

D.1 Main Program

```
% This script initializes the quadcopter parameters and matrices
    needed to
% run the MPC control algorithm.

% In addition, the main control algorithm is initialized here.

%% Parameter Declarations

tic % initialize timer

% Quadcopter properties

m = 1.587;                % Mass                      [kg]
g = 9.81;                 % Accel due to gravity      [m/(s^2)]
Ixx = 0.0213;             % x-axis moment of inertia  [kg*m^2]
Iyy = 0.02217;            % y-axis moment of inertia  [kg*m^2]
Izz = 0.0282;             % z-axis moment of inertia  [kg*m^2]
b = 4.0687e-7;            % Thrust coefficient        [N/(rpm)^2]
k = 8.4367e-9;            % Drag coefficient          [N*m/(rpm)^2]
d_arm = 0.243;            % Moment arm                [m]
max_speed = 4720;         % Maximum motor speed       [rpm]
min_speed = 3093;         % Minimum take-off speed    [rpm]

% Controller design parameters

no_of_states = 6;         % Number of states
no_of_inputs = 3;         % Number of inputs
no_of_outputs = 3;        % Number of outputs
```

```
nu = 3;                     % control horizon
ny = 6;                     % prediction horizon
N_sim = 20;                 % Number of simulations
x = zeros(6,1);             % Initial state values
y = zeros(3,1);             % Initial output values
u = [0;0;0];                % Initial control values
t = 0:0.2:N_sim;            % Steps for sinusoidal reference
    functions

% Reference to be tracked
r = [sin(t) + cos(3*t)/2; sin(t) + cos(3*t)/2; sin(t) + cos(2*t)/2
     + sin(3*t)/3];

Rs = ref_adj(no_of_outputs, ny); % Reference Adjusting Matrix
dist = (rand(3,101)*2 -1)*0.5; % Disturbance

%% State Space Matrices

A = [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];

B = [0 0 0;
    1/Ixx 0 0;
    0 0 0;
    0 1/Iyy 0;
    0 0 0;
    0 0 1/Izz];

C = [0 1 0 0 0 0;...
    0 0 0 1 0 0;...
    0 0 0 0 0 1];

D = zeros(3,3);

%% Converting from Continuous to Discrete Time

[Ad,Bd,Cd,Dd] = c2dm(A,B,C,D,0.2);

%% Augmented Model

[A_aug,B_aug,C_aug] = augment_mimo(Ad,Bd,Cd,no_of_states,
    no_of_inputs,no_of_outputs);

% Controllability check
CO = [B_aug A_aug*B_aug A_aug^2*B_aug A_aug^3*B_aug A_aug^4*B_aug
    A_aug^5*B_aug];

rank(CO);
```

```
76
77 %% Prediction Matrices
78
79 [H,P] = mpc_predictions_output(A_aug,B_aug,C_aug,ny,nu); % Output
       Predictions
80
81
82 %% Constraints
83
84 % Constraint calculations
85
86 u_1 = d_arm * b*(max_speed^2 - min_speed^2);
87 u_2 = u_1;
88 u_3 = k * (max_speed^2 + max_speed^2 - min_speed^2 - min_speed^2);
89
90 umax = [u_1; u_2; u_3];
91 umin = -umax;
92 Dumax = 0.6*umax;
93
94 % Constraint matrices
95
96 [CC, dd, dupast] = constraints_mimo(Dumax,umax,umin,no_of_inputs,
       nu);
97
98 %% Cost Function & its Parameters
99
100 % Weights on control inputs
101
102 % Works
103 % W = [7.5e-2 0 0 0 0 0  ;...
104 %       0 7.5e-2 0 0 0 0  ;...
105 %       0 0 4.5e-2 0 0 0 ;...
106 %       0 0 0 7.5e-2 0 0 ;...
107 %       0 0 0 0 7.5e-2 0  ;...
108 %       0 0 0 0 0 4.5e-2];
109
110 % Also works
111 % W = [7.5e1 0 0 0 0 0  ;...
112 %       0 7.5e1 0 0 0 0  ;...
113 %       0 0 4.5e1 0 0 0 ;...
114 %       0 0 0 7.5e1 0 0 ;...
115 %       0 0 0 0 7.5e1 0  ;...
116 %       0 0 0 0 0 4.5e1];
117
118   W = [7.5e-2 0 0 0 0 0 0 0 0 ;...
119        0 7.5e-2 0 0 0 0 0 0 0 ;...
120        0 0 4.5e-2 0 0 0 0 0 0;...
121        0 0 0 7.5e-2 0 0 0 0 0 ;...
122        0 0 0 0 7.5e-2 0 0 0 0 ;...
123        0 0 0 0 0 4.5e-2 0 0 0 ;...
124        0 0 0 0 0 0 7.5e-2 0 0 ;...
125        0 0 0 0 0 0 0 7.5e-2 0 ;...
```

```
126       0 0 0 0 0 0 0 0 4.5e-2];
127
128 E = 2*(H'*H + W);
129
130 % Simulation loop parameters
131
132 Xf = zeros(size(B_aug,1),1);
133 xh = zeros(size(B_aug,1),1);
134 yh = y;
135
136 % Simulation loop
137
138 for i = 1:100
139
140       F = -2*H'*(Rs*r(:,i) - P*(Xf));
141
142       d = dd + dupast*u;
143
144       % Quadratic Progamming
145
146       [DeltaU, Uncons] = QPhild(E,F,CC,d);
147
148       % For control horizon of 2
149 %       DeltaU = [DeltaU(1,1) DeltaU(2,1) DeltaU(3,1);...
150 %                 DeltaU(4,1) DeltaU(5,1) DeltaU(6,1)];
151
152       % For control horizon of 3
153       DeltaU = [DeltaU(1,1) DeltaU(2,1) DeltaU(3,1);...
154                 DeltaU(4,1) DeltaU(5,1) DeltaU(6,1);...
155                 DeltaU(7,1) DeltaU(8,1) DeltaU(9,1)];
156
157       deltau = DeltaU(1,:);
158       u = u + deltau'; % Input
159
160       % Model
161
162       xh(:,i+1) = A_aug*xh(:,i) + B_aug*deltau';
163       yh(:,i) = C_aug*xh(:,i+1) + dist(:,i);
164       Xf = xh(:,i+1);
165
166 %     % Same result with model above
167
168 %     x(:,i+1) = Ad*x(:,i) + Bd*u;
169 %     y(:,i) = Cd*x(:,i+1) + dist(:,i);
170 %     Xf = [x(:,kk+1) - x(:,kk); y(:,kk)]
171
172       % Plot variables
173
174       u1(:,i) = u;
175       delt(:,i) = deltau;
176
177 %       Scaled input
```

```
178 %       u_s = (2 * ((u(:,1) - umin(:,1))./(umax(:,1) - umin(:,1)))
        - 1);
179
180 end
181
182 toc % terminate timer
183
184 %% Plots
185
186 % i=1:N_sim;
187 i = 1:100;
188
189 % Output 1
190 subplot(3,1,1)
191 plot(i,r(1,i),'k',i,yh(1,i),'r')
192 legend('reference trajectory','actual output')
193 xlabel('Time {ms}')
194 ylabel('\phi {rad}')
195
196 % Output 2
197 subplot(3,1,2)
198 plot(i,r(2,i),'k',i,yh(2,i),'r')
199 legend('reference trajectory','actual output')
200 xlabel('Time {ms}')
201 ylabel('\theta {rad}')
202
203 % Output 3
204 subplot(3,1,3)
205 plot(i,r(3,i),'k',i,yh(3,i),'r')
206 legend('reference trajectory','actual output')
207 xlabel('Time {ms}')
208 ylabel('\psi {rad}')
209
210 figure (2)
211 subplot(3,1,1)
212 plot(i, umax(1)*ones(1,100), '--k', i, umin(1)*ones(1,100), '--k',
        i, u1(1,i), '--g')
213 title('Plots of input over a sample period')
214
215 subplot(3,1,2)
216 plot(i, umax(2)*ones(1,100), '--k', i, umin(2)*ones(1,100), '--k',
        i, u1(2,i), '--g')
217
218 subplot(3,1,3)
219 plot(i, umax(3)*ones(1,100), '--k', i, umin(3)*ones(1,100), '--k',
        i, u1(3,i), '--g')
220
221 figure (3)
222 subplot(3,1,1)
223 plot(i, Dumax(1)*ones(1,100), '--k', i, -Dumax(1)*ones(1,100), '--
      k', i, delt(1,i), '--g')
224 title('Change in input over a sample period')
```

```
225
226 subplot(3,1,2)
227 plot(i, Dumax(2)*ones(1,100), '--k', i, -Dumax(2)*ones(1,100), '--
      k', i, delt(2,i), '--g')
228
229 subplot(3,1,3)
230 plot(i, Dumax(3)*ones(1,100), '--k', i, -Dumax(3)*ones(1,100), '--
      k', i, delt(3,i), '--g')
```

D.2 Augment State Space Matrices

```
1 function [A_aug,B_aug,C_aug] = augment_mimo(Ad,Bd,Cd,no_of_states,
      no_of_inputs,no_of_outputs)
2
3 A_aug = [Ad zeros(no_of_states,no_of_inputs); Cd*Ad eye(
      no_of_outputs)];
4 B_aug = [Bd; Cd*Bd];
5
6 C_aug = [zeros(no_of_inputs, no_of_states) eye(no_of_inputs)];
```

D.3 Constraint Matrices and Vectors

```
1 % Constraint matrices for mimo with state/output constraints
2
3 % CC*du(future) - dd + du*u(k-1) + dy*ypast   ypast is the past
      state vector
4
5 % Dumax is the change in input
6 % umax and umin are the input limits
7 % ymax and ymin are the output/state limits
8 % nu is the control horizon
9 % ny is the output/state horizon
10 % no_of_inputs is the number of inputs
11
12 function [CC, dd, dupast] = constraints_mimo(Dumax,umax,umin,
      no_of_inputs,nu)
13
14 CC = [];
15 dd = []; % vector of limits
16 dupast = []; % matrix coefficient of past input, u(k-1)
17 % dypast = []; % matrix/vector coefficient of past output, y(k-1)
18 dim = nu*no_of_inputs;
19
20 %% CC
21
22 % C Delta U
23
24 CC(1:dim,1:dim) = eye(dim);
25 CC(dim+1:2*dim,1:dim) = -eye(dim);
```

```
26
27 % CuE
28 % E
29
30 s = 0; % row counter
31
32 E = zeros(dim,dim);
33
34 % Rows
35 for j=1:nu
36     E(1+s:no_of_inputs*j,1:no_of_inputs) = eye(no_of_inputs);
37
38     s = s + no_of_inputs;
39 end
40
41 % Columns
42
43 p = no_of_inputs + 1;
44
45 for counter = 1:nu-1
46     i = counter;
47     for k = (no_of_inputs*counter)+1:no_of_inputs:dim-no_of_inputs
           +1
48         E(k:no_of_inputs*(i+1),p:no_of_inputs*(counter+1)) = eye(
               no_of_inputs);
49         i = i + 1;
50     end
51     p = p + no_of_inputs;
52 end
53
54 % Cu
55 Cu = [eye(dim); -eye(dim)];
56 CuE = Cu*E;
57
58 % Putting CuE in CC
59 CC(2*dim+1:4*dim,1:dim) = CuE;
60
61 %% dd vector of limits
62
63 % d deltaU limits
64
65 ddeltaU = zeros(2*dim,1);
66 t = 1;
67 for i=1:no_of_inputs:size(ddeltaU,1)
68
69     ddeltaU(i:t*no_of_inputs,1) = Dumax;
70   t = t+1;
71 end
72
73 % du limits on input
74 du = zeros(2*dim,1);
75
```

```
    % Upper Limits
    t = 1;
    for i=1:no_of_inputs:size(du,1)/2
        du(i:t*no_of_inputs,1) = umax;
        t = t + 1;
    end

%     % Lower Limits
    for i = 1 + size(du,1)/2: no_of_inputs: size(du,1)
        du(i:t*no_of_inputs,1) = -umin;
        t = t+1;
    end

    dd = [ddeltaU; du];

%% dupast past inputs

 % L
 L = zeros();
 t = 1;
  for i=1:no_of_inputs:(no_of_inputs*nu)
    L(i:no_of_inputs*t,1:no_of_inputs) = eye(no_of_inputs);
    t = t+1;
  end

  % -CuL
  CuL = Cu*L;

    dupast = [zeros(size(ddeltaU,1),no_of_inputs);-CuL];
```

D.4 Hildreth's Quadratic Programming Function

```
function [DeltaU, Uncons] = QPhild(E,F,CC,d)

[n1,m1] = size(CC);
DeltaU = -E\F;  % Global solution without constraints
Uncons = DeltaU;
kk = 0;

for i=1:n1
    if(CC(i,:)*DeltaU > d(i))
        kk = kk + 1;
    else
        kk = kk + 0;
    end
end

if (kk == 0)
    return;
```

```
18 end
19
20 % Dual quadratic programming matrices
21
22 T = CC*(E\CC'); % Same as H matrix = M*inv(E)*M'
23
24 K = (CC*(E\F) + d);% This is the K matrix = gamma + M*inv(E)*F ..
       from their examples
25
26 % backslash(\) is used instead of the inverse function, inv()
27
28 [n, m] = size(K);
29 lambda= zeros(n,m);
30 al = 3;
31
32 for km = 1:15 % Number of iterations
33     % find the elelments in the solution vector one by one
34     % km could be larger i the Lagrange multiplier has a slow
35     % convergence rate
36     lambda_p = lambda; % previous lambda
37
38     for i=1:n % Loop to determine lambda values for respective
           iters
39         w = T(i,:)*lambda - T(i,i)*lambda(i,1);
40         w = w + K(i,1);
41         la = -w/T(i,i);
42         lambda(i,1) = max(0,la);
43     end
44
45     al = (lambda-lambda_p)'*(lambda-lambda_p);
46     if (al < 10e-5)
47         break;
48     end
49 end
50
51 DeltaU = -E\F - E\CC'*lambda;
```

Appendix E

C++ Code

The MPC angular rates C++ code in this appendix is separated in sections according to how the code is typed in the PX4 attitude control module. A control horizon of two and a prediction horizon of five was utilised in the controller code snippets in the following sections.

E.1 Includes

These are the headers included at the top of the `mc_att_control.cpp` file in addition to the default headers.

```
#include <px4_eigen.h>
#include <eigen/Eigen/Dense>
#include <eigen/unsupported/Eigen/MatrixFunctions>
#include <eigen/Eigen/Core>
#include <eigen/Eigen/Cholesky>
```

E.2 Multicopter Attitude Control Class

The code snippet below shows the definitions for the matrices and vectors used for the MPC controller in the multicopter attitude class.

```

// Hildreth's Quadratic programming function
Matrix<float, 6, 1> QPhild(const Matrix<float, 6, 6> &E,
Matrix<float, 6, 1> &F, const Matrix<float, 24, 6> &CC,
Matrix<float, 24, 1> &d);
Matrix<float, 6, 1> x;          // State vector
Matrix<float, 6, 1> x_prev;     // Previous state vector
Matrix<float, 3, 1> des;        // Reference/desired rates vector
Matrix<float, 3, 1> u;          // Input/control vector
Matrix<float, 9, 1> Xf;         // Augmented state vector
```

```
Matrix<float, 3, 1> y;          // Output vector
Matrix<float, 3, 1> x_curr;     // Current rates vector
Matrix<float, 15, 9> P;         // Output prediction matrix, P
Matrix<float, 15, 6> H;         // Output prediction matrix, H
Matrix<float, 6, 15> H_trans;   // Transpose of H matrix
Matrix<float, 3, 1> dist;       // Randomised disturbance vector
Matrix<float, 6, 6> E;          // Quadratic programming matrix, E
Matrix<float, 6, 6> W;          // Input weight
Matrix<float, 24, 6> CC;        // Constraint matrix
Matrix<float, 24, 1> dd;        // Constraint vector
Matrix<float, 24, 3> dupast;    // Past constraint matrix
```

E.3 Constructor

Some of the vectors and matrices defined in the multicopter attitude class are initialised in the constructor. These initialisations are shown below.

```
MulticopterAttitudeControl::MulticopterAttitudeControl() :
{
    // Initialisations
    x.setZero(6,1);
    u.setZero(3,1);
    y.setZero(3,1);
    Xf.setZero(9,1);

    // Output prediction matrix P
    P <<
        0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 2.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 2.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 2.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 3.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 3.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 3.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 4.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 4.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 4.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 5.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 5.0f, 0.0f, 0.0f, 1.0f;

    // Output prediction matrix H
    H << 9.3897f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 9.0212f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 7.0922f, 0.0f, 0.0f, 0.0f,
        18.7793f, 0.0f, 0.0f, 9.3897f, 0.0f, 0.0f,
        0.0f, 18.0424f, 0.0f, 0.0f, 9.0212f, 0.0f,
        0.0f, 0.0f, 14.1844f, 0.0f, 0.0f, 7.0922f,
```

```
34          28.169f, 0.0f, 0.0f, 18.7793f, 0.0f, 0.0f,
35          0.0f, 27.0636f, 0.0f, 0.0f, 18.0424f, 0.0f,
36          0.0f, 0.0f, 21.2766f, 0.0f, 0.0f, 14.1844f,
37           37.5587f, 0.0f, 0.0f, 28.1690f, 0.0f, 0.0f,
38           0.0f, 36.0848f, 0.0f, 0.0f, 27.0636f, 0.0f,
39           0.0f, 0.0f, 28.3688f, 0.0f, 0.0f, 21.2766f,
40          46.9484f, 0.0f, 0.0f, 37.5587f, 0.0f, 0.0f,
41           0.0f, 45.1060f, 0.0f, 0.0f, 36.0848f, 0.0f,
42           0.0f, 0.0f, 35.4610f, 0.0f, 0.0f, 28.3688f;
43
44      // Input weight
45      W << 0.065f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
46           0.0f, 0.065f, 0.0f, 0.0f, 0.0f, 0.0f,
47           0.0f, 0.0f, 0.085f, 0.0f, 0.0f, 0.0f,
48           0.0f, 0.0f, 0.0f, 0.065f, 0.0f, 0.0f,
49           0.0f, 0.0f, 0.0f, 0.0f, 0.065f, 0.0f,
50           0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.085f;
51
52      // Transpose of H output matrix
53      H_trans = H.transpose();
54
55      // quadratic programming variable, E
56      E = 2 * (H_trans*H + W);
57
58      // Constraint matrix
59      CC << 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
60           0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
61           0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
62           0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
63           0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
64           0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
65           -1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
66           0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
67           0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f,
68           0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,
69           0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f,
70           0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
71           1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
72           0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
73           0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
74           1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
75           0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
76           0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
77           -1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
78           0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
79           0.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f,
80           -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,
81           0.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f,
82           0.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.0f;
83
84
85
```

```
86      // Constraint vector of input and rate of input change
87      dd <<   0.7548f,
88              0.7548f,
89              0.1288f,
90              0.7548f,
91              0.7548f,
92              0.1288f,
93              0.7548f,
94              0.7548f,
95              0.1288f,
96              0.7548f,
97              0.7548f,
98              0.1288f,
99              1.258f,
100             1.258f,
101             0.2147f,
102             1.258f,
103             1.258f,
104             0.2147f,
105             1.258f,
106             1.258f,
107             0.2147f,
108             1.258f,
109             1.258f,
110             0.2147f;
111
112     // Past rate of input change matrix
113     dupast << 0.0f, 0.0f, 0.0f,
114             0.0f, 0.0f, 0.0f,
115             0.0f, 0.0f, 0.0f,
116             0.0f, 0.0f, 0.0f,
117             0.0f, 0.0f, 0.0f,
118             0.0f, 0.0f, 0.0f,
119             0.0f, 0.0f, 0.0f,
120             0.0f, 0.0f, 0.0f,
121             0.0f, 0.0f, 0.0f,
122             0.0f, 0.0f, 0.0f,
123             0.0f, 0.0f, 0.0f,
124             0.0f, 0.0f, 0.0f,
125             -1.0f, 0.0f, 0.0f,
126             0.0f, -1.0f, 0.0,
127             0.0f, 0.0f, -1.0f,
128             -1.0f, 0.0f, 0.0f,
129             0.0f, -1.0f, 0.0f,
130             0.0f, 0.0f, -1.0f,
131             1.0f, 0.0f, 0.0f,
132             0.0f, 1.0f, 0.0f,
133             0.0f, 0.0f, 1.0f,
134             1.0f, 0.0f, 0.0f,
135             0.0f, 1.0f, 0.0f,
136             0.0f, 0.0f, 1.0f;
137 }
```

E.4 Hildreth's Quadratic Programming Function

The Hildreth's quadratic programming function is defined in the following code snippet.

```
// Hildreth's Quadratic Programming function

Matrix<float, 6, 1> MulticopterAttitudeControl::
QPhild(const Matrix<float, 6, 6> &E, Matrix<float, 6, 1> &F,
const Matrix<float, 24, 6> &CC, Matrix<float, 24, 1> &d)
{
    // Compute decomposition of E
    static LLT<Matrix<float, 6, 6>> lltOfE(E);
    static Matrix<float, 6, 24> CC_transd = CC.transpose();
    Matrix<float, 24, 24> T = CC*(lltOfE.solve(CC_transd));
    Matrix<float, 24, 1> K = (CC*(lltOfE.solve(F)) + d);

    int k_row = 24;
    Matrix<float, 24, 1> lambda;
    lambda.setZero(24, 1);
    float al = 3.0f;
    int km = 0;

    do
    {
        Matrix<float, 24, 1> lambda_p = lambda;
    // loop to determine lambda values for respective iterations
        int i = 0;
        do
        {
            float Tii = T(i, i);
            T(i, i) = 0;
            float la = -(T.col(i).dot(lambda) + K(i)) / Tii;
            T(i, i) = Tii;
            if (la < 0.0f)  lambda(i) = 0.0f;
            else lambda(i) = la;
            i++;
        } while (i < k_row);
        al = (lambda - lambda_p).squaredNorm();

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

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

    return DeltU;
}
```

E.5 Inside Attitude Control Rates Function

In this final section, the MPC angular rates controller code is presented.

```
void MulticopterAttitudeControl::control_attitude_rates(float dt)
{
    // State space matrices
    static Matrix<float, 6, 6> Ad;
    Ad << 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;

    static Matrix<float, 6, 3> Bd;
    Bd << 0.9390f, 0.0f, 0.0f,
        9.3897f, 0.0f, 0.0f,
        0.0f, 0.9021f, 0.0f,
        0.0f, 9.0212f, 0.0f,
        0.0f, 0.0f, 0.7092f,
        0.0f, 0.0f, 7.0922f;

    static Matrix<float, 3, 6> Cd;
    Cd << 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;

    // Desired attitude rates/ rates setpoint
    des << _rates_sp(0),
            _rates_sp(1),
            _rates_sp(2);

    // Reference adjusting matrix
    static  Matrix<float, 15, 3> Rs;
    Rs << 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;

```

```
48      // disturbance
49      dist.setRandom(3,1);
50      dist*0.06;
51
52      // current rates
53      x_curr << rates(0),
54              rates(1),
55              rates(2);
56
57      // Defining previous state vector
58      x_prev <<  0,
59              x_curr(0),
60              0,
61              x_curr(1),
62              0,
63              x_curr(2);
64
65      // QPhild section to determine torque vector,
66      // that is _att_control vector
67
68      int i = 0;
69      do{
70
71          Matrix<float, 6, 1> F = -2 * (H_trans)*(Rs*des - P*Xf);
72          Matrix<float, 24, 1> d = dd + dupast*u;
73          Matrix<float, 6, 1> DeltaU = QPhild(E, F, CC, d);
74
75          Matrix<float, 2, 3> DeltaU_1;
76
77          DeltaU_1 << DeltaU(0, 0), DeltaU(1, 0), DeltaU(2, 0),
78                      DeltaU(3, 0), DeltaU(4, 0), DeltaU(5, 0);
79
80          Matrix<float, 3, 1>deltau_tran=(DeltaU_1.row(0)).transpose
                ();
81          u = u + deltau_tran;
82
83          x = Ad*x_prev + Bd*u;
84          y = Cd*x;//+ dist;
85
86          Xf << x - x_prev,
87              y;
88
89          x_prev = x;
90          i++;
91      } while(i < 2);
92
93          _att_control(0) = u(0);
94          _att_control(1) = u(1);
95          _att_control(2) = u(2);
96
97          math::Vector<3> umaxx(1.258f, 1.258f, 0.2147f);
98          math::Vector<3> uminn = -umaxx;
```

```
99
100     // Scale inputs or attitude control vector between -1 and 1
101     // before sending to mixer module
102
103     for (int k = 0; k < 3; k++)
104         {
105         _att_control(k) = (2 * ((_att_control(k) - uminn(k))
106         /(umaxx(k) - uminn(k)))-1);
107         }
108     }
```

Appendix F

jMAVSim

The following lines of code below are executed in the Ubuntu terminal; from cloning the repository that contains the code for the multirotor simulator to installing the required programs and finally to running the software.

Clone repository and initialise submodule:

```
git clone https://github.com/DrTon/jMAVSim
git submodule init
git submodule update
```

Compile:

```
cd jMAVSim
ant
```

Run:

```
java -cp lib/*:out/production/jmavsim.jar me.drton.jmavsim.
    Simulator
```

More information can be found on the jMAVSim GitHub page.

The QGroundControl software is downloaded from the QGC website.

Appendix G

Additional Results

G.1 MATLAB Simulations

The parameters used for each MATLAB simulation are shown in tables G.1 and G.2.

Table G.1: MATLAB MPC simulation parameters for $n_u = 2$, $n_y = 2$

n_u	n_y	**p, q**	**r**	**disturbance**
2	2	$\text{sint} + \frac{cos(3t)}{2}$	$\text{sint} + \frac{cos(2t)}{2} + \frac{sin(3t)}{3}$	`(rand(3,101)*2 -1)*0.5`

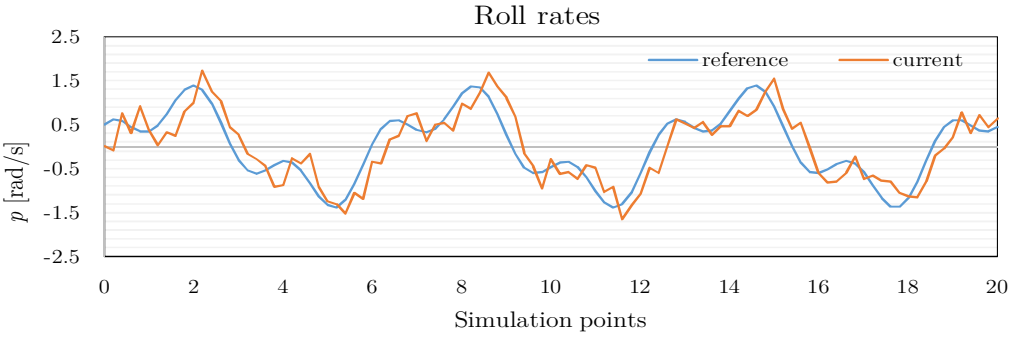


Figure G.1: MATLAB roll rates for $n_u = 2$, $n_y = 2$

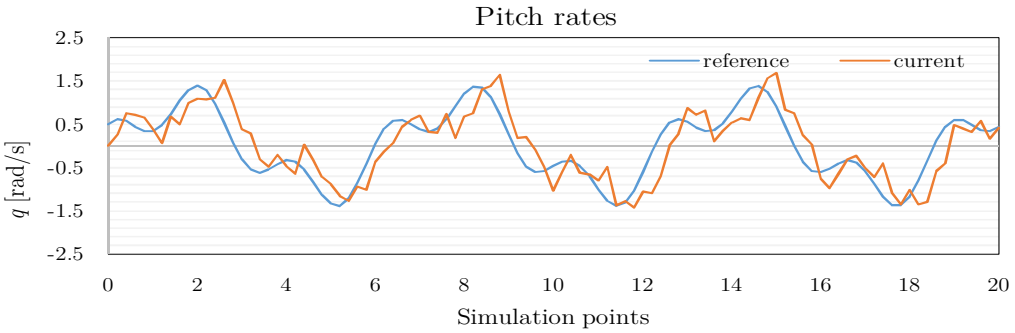


Figure G.2: MATLAB pitch rates for $n_u = 2$, $n_y = 2$

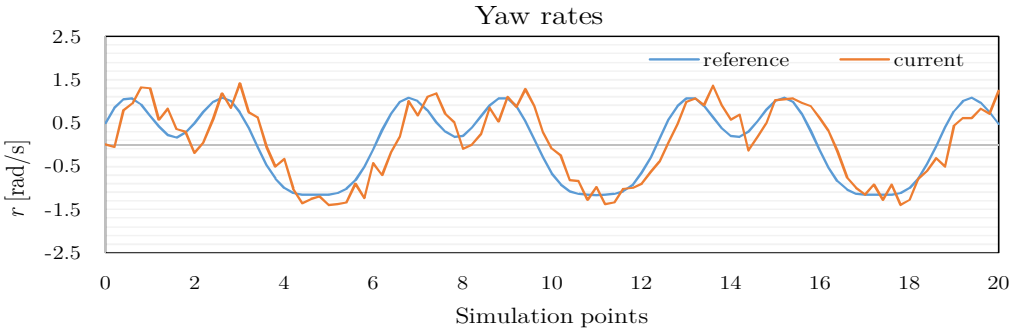


Figure G.3: MATLAB yaw rates for $n_u = 2$, $n_y = 2$

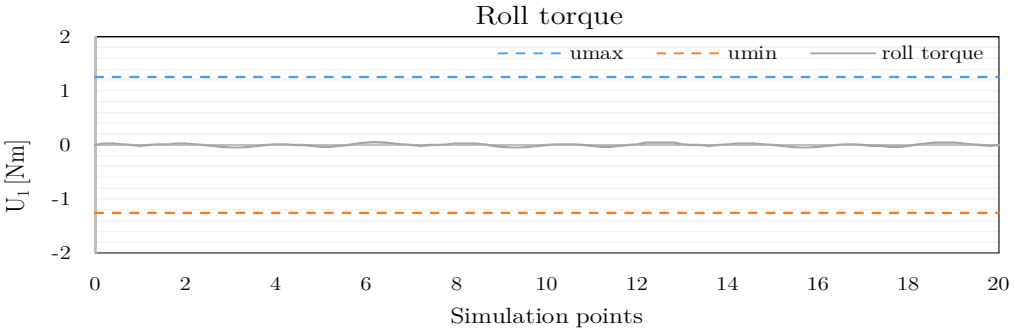


Figure G.4: MATLAB roll torque for $n_u = 2$, $n_y = 2$

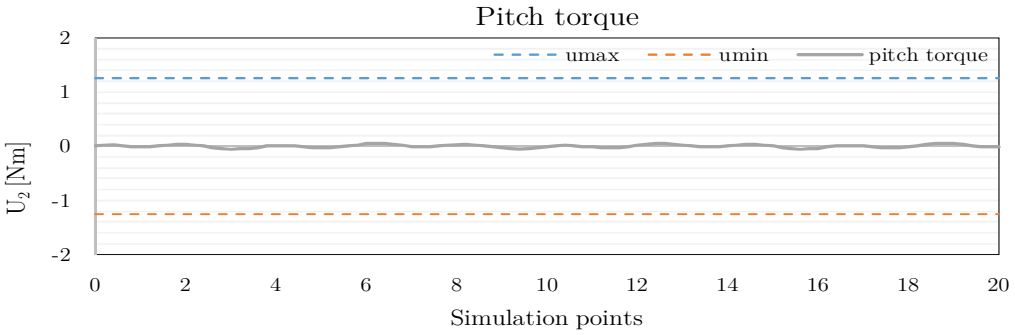


Figure G.5: `MATLAB` pitch torque for $n_u = 2$, $n_y = 2$

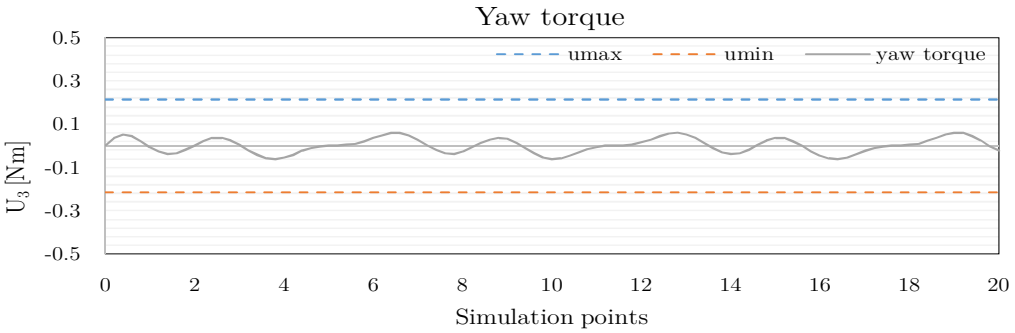


Figure G.6: `MATLAB` yaw torque for $n_u = 2$, $n_y = 2$

Table G.2: `MATLAB` MPC simulation parameters for $n_u = 2$, $n_y = 4$

n_u	n_y	**p, q**	**r**	**disturbance**
2	4	$\text{sint} + \frac{cos(3t)}{2}$	$\text{sint} + \frac{cos(2t)}{2} + \frac{sin(3t)}{3}$	`(rand(3,101)*2 -1)*0.5`

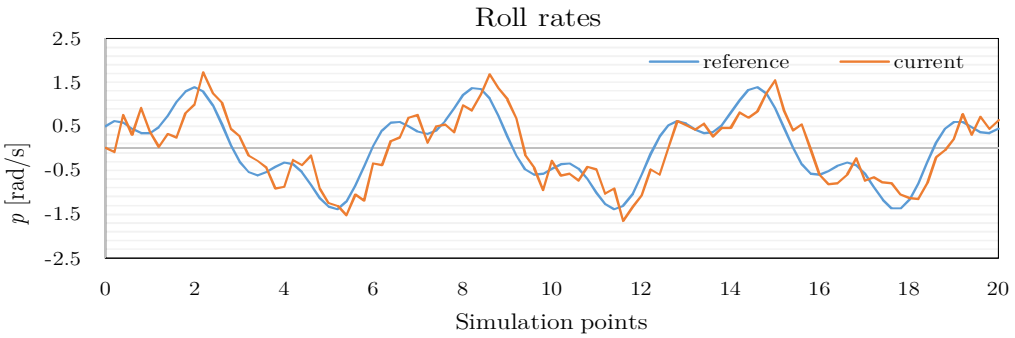


Figure G.7: MATLAB roll rates for $n_u = 2$, $n_y = 4$

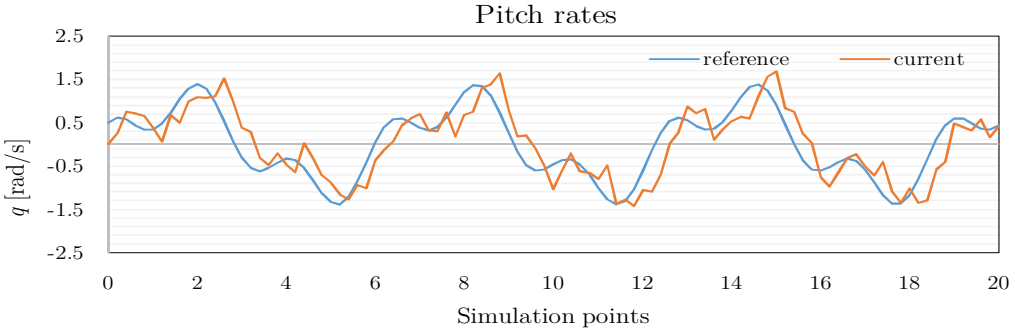


Figure G.8: MATLAB pitch rates for $n_u = 2$, $n_y = 4$

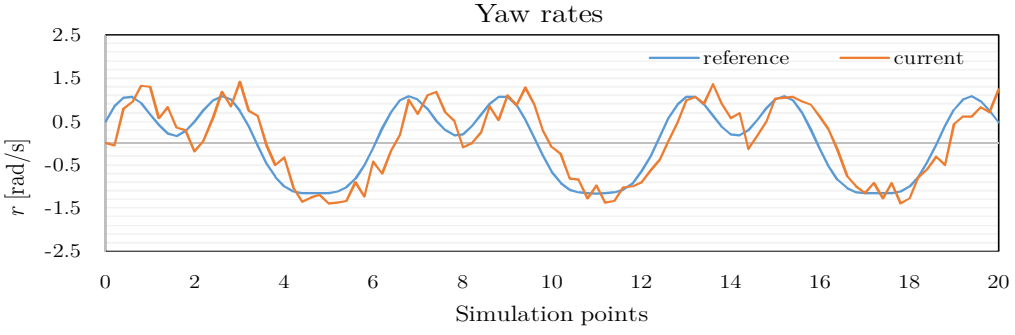


Figure G.9: MATLAB yaw rates for $n_u = 2$, $n_y = 4$

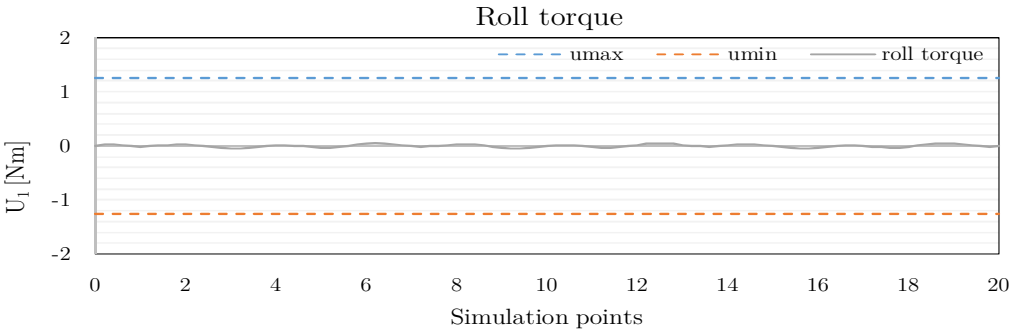


Figure G.10: MATLAB roll torque for $n_u = 2$, $n_y = 4$

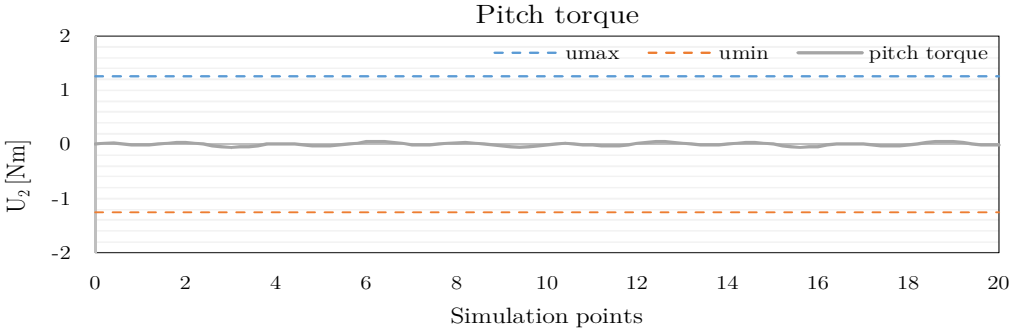


Figure G.11: MATLAB pitch torque for $n_u = 2$, $n_y = 4$

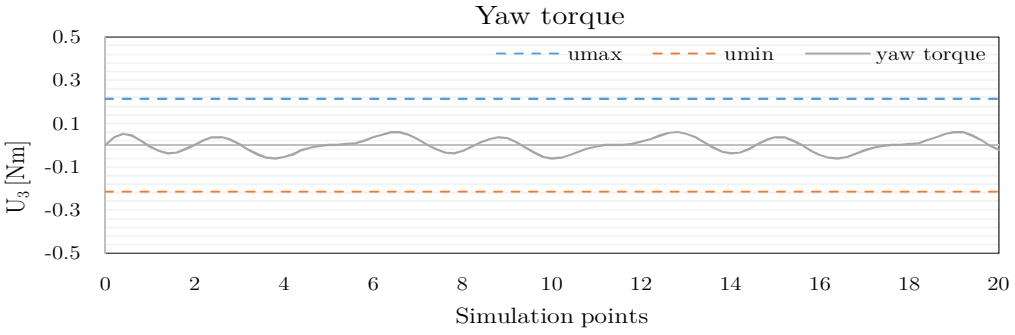


Figure G.12: MATLAB yaw torque for $n_u = 2$, $n_y = 4$

G.2 Software-in-the-loop Simulations

The flight data from SITL flight missions executed in section 4.2 are plotted in the following subsections.

The results displayed below were obtained from executing the SITL flight mission in section 4.2 for $n_u = 2$ and $n_y = 2$.

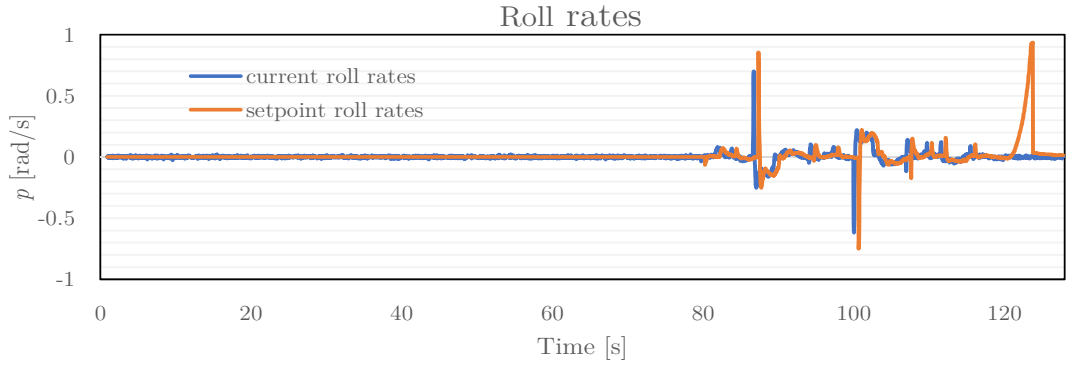


Figure G.13: SITL roll rates for $n_u = 2$, $n_y = 2$

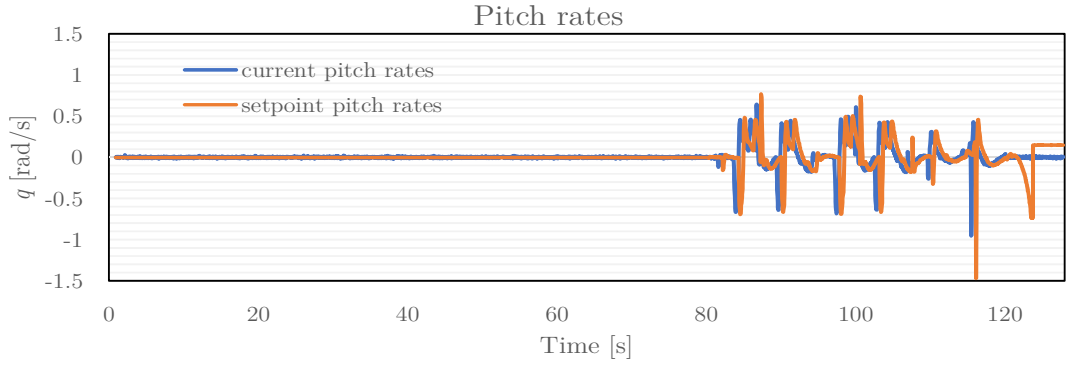


Figure G.14: SITL pitch rates for $n_u = 2$, $n_y = 2$

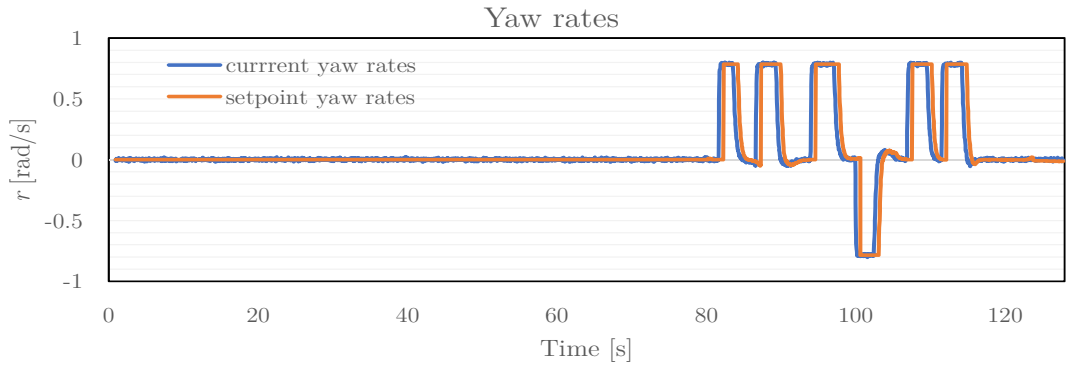


Figure G.15: SITL yaw rates for $n_u = 2$, $n_y = 2$

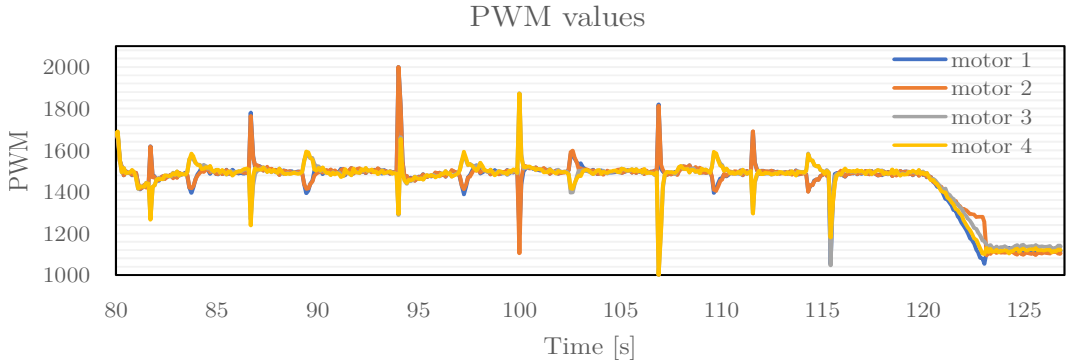


Figure G.16: SITL PWM values for $n_u = 2$, $n_y = 2$

G.3 Flight Tests

Additional results from obtained from outdoor flight testing are plotted in this section with a control horizon of two and a prediction horizon of five.

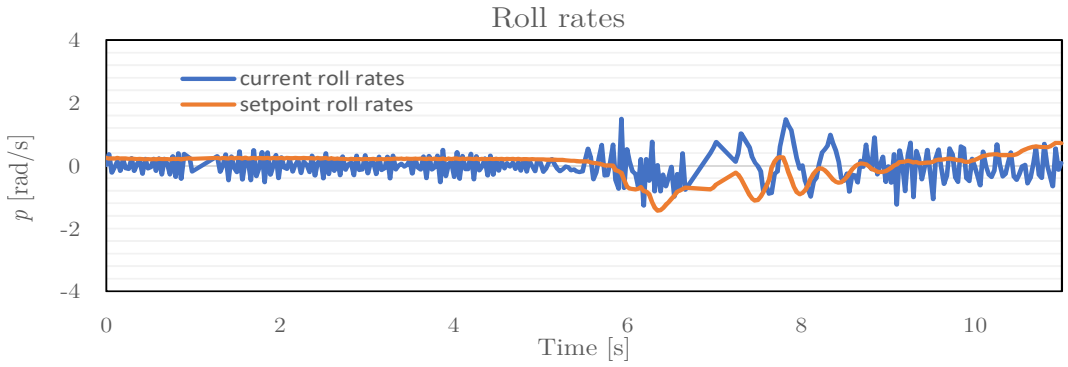


Figure G.17: Flight roll rates for $n_u = 2$, $n_y = 5$

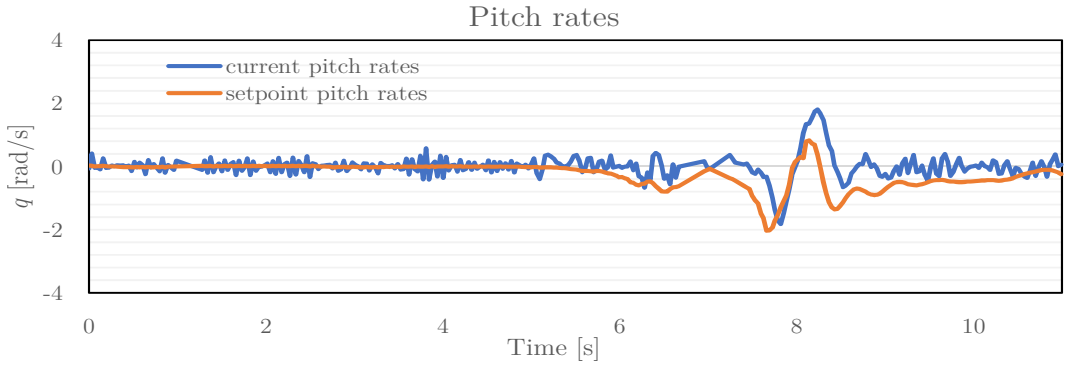


Figure G.18: Flight pitch rates for $n_u = 2$, $n_y = 5$

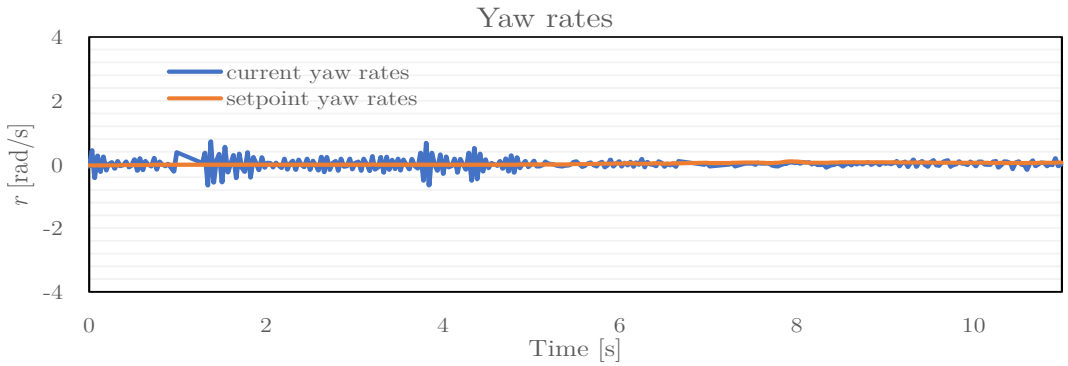


Figure G.19: Flight yaw rates for $n_u = 2$, $n_y = 5$

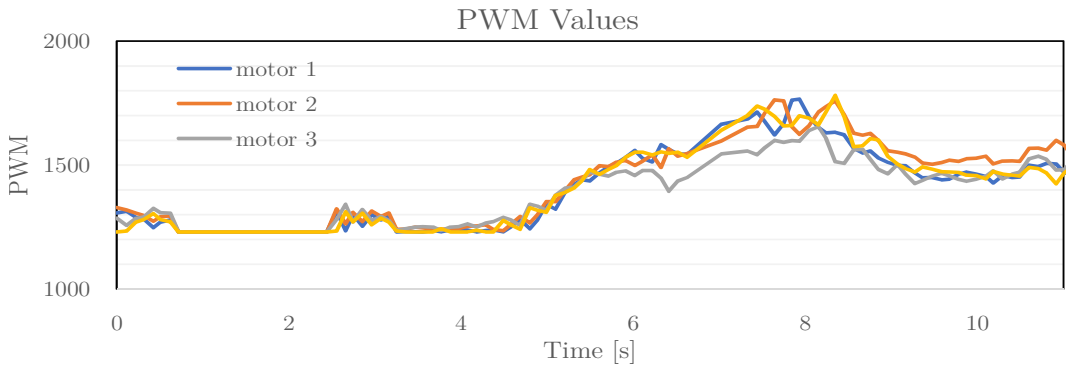


Figure G.20: Flight PWM values for $n_u = 2$, $n_y = 5$

List of References

Alderete, T.S. (1995). Simulator aero model implementation. *NASA Ames Research Center, Moffett Field, California*. (Cited on page 12.)

Bangura, M. and Mahony, R. (2014). Real-time model predictive control for quadrotors. *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 11773–11780. (Cited on page 17.)

Bemporad, A., Pascucci, C.A. and Rocchi, C. (2009). Hierarchical and hybrid model predictive control of quadcopter air vehicles. *IFAC Proceedings Volumes*, vol. 42, no. 17, pp. 14–19. (Cited on page 17.)

Bhatkhande, P. and Havens, T.C. (2014). Real time fuzzy controller for quadrotor stability control. In: *Fuzzy Systems (FUZZ-IEEE), 2014 IEEE International Conference on*, pp. 913–919. IEEE. (Cited on page 3.)

Carminati, A., Starke, R.A. and de Oliveira, R.S. (2017). Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software. *Applied Computing and Informatics*, vol. 13, no. 2, pp. 184 – 193. ISSN 2210-8327.
Available at: `http://www.sciencedirect.com/science/article/pii/S2210832716300564` (Cited on page 52.)

Carrillo, L.R.G., López, A.E.D., Lozano, R. and Pégard, C. (2013). Modeling the quad-rotor mini-rotorcraft. In: *Quad Rotorcraft Control*, pp. 23–34. Springer. (Cited on page 12.)

Cavcar, M. (2000). The international standard atmosphere (isa). *Anadolu University, Turkey*, vol. 30, p. 9. (Cited on page 9.)

Choset, H.M. (2005). *Principles of robot motion: theory, algorithms, and implementation*. MIT press. (Cited on page 32.)

CMake (). Cmake. `https://cmake.org/`. (Cited on page 54.)

Console, P.S. (). System console Â· px4 developer guide. `https://dev.px4.io/en/debug/system_console.html`. (Cited on page 58.)

Cutler, C.R. and Ramaker, B.L. (1980). Dynamic matrix control?? a computer control algorithm. In: *Joint automatic control conference*, 17, p. 72. (Cited on page 17.)

Dani, S., Sonawane, D., Ingole, D. and Patil, S. (2017). Performance evaluation of pid, lqr and mpc for dc motor speed control. In: *Convergence in Technology (I2CT), 2017 2nd International Conference for*, pp. 348–354. IEEE. (Cited on page 20.)

Diehl, M. (2015). Optimization algorithms for model predictive control. *Encyclopedia of Systems and Control*, pp. 989–997. (Cited on pages 28 and 29.)

Forbes, M.G., Patwardhan, R.S., Hamadah, H. and Gopaluni, R.B. (2015). Model predictive control in industry: Challenges and opportunities. *IFAC-PapersOnLine*, vol. 48, no. 8, pp. 531–538. (Cited on page 17.)

Ganga, G. and Dharmana, M.M. (2017). Mpc controller for trajectory tracking control of quadcopter. In: *Circuit, Power and Computing Technologies (ICCPCT), 2017 International Conference on*, pp. 1–6. IEEE. (Cited on page 3.)

Giernacki, W., Skwierczyński, M., Witwicki, W., Wroński, P. and Kozierski, P. (2017). Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In: *Methods and Models in Automation and Robotics (MMAR), 2017 22nd International Conference on*, pp. 37–42. IEEE. (Cited on page 7.)

Golnaraghi, F. and Kuo, B.C. (2010). Automatic control systems. *Complex Variables*, vol. 2, pp. 1–1. (Cited on page 22.)

Greenwood, D.T. (2003). *Advanced Dynamics*. Cambridge University Press. (Cited on page 7.)

Grujic, I. and Nilsson, R. (2016). Model-based development and evaluation of control for complex multi-domain systems: attitude control for a quadrotor uav. *Technical Report Electronics and Computer Engineering*, vol. 4, no. 23. (Cited on pages 10, 11, 12, 14, and 16.)

Habeck, J. and Seiler, P. (2016). Moment of inertia estimation using a bifilar pendulum. (Cited on pages 81 and 82.)

Habib, M.K., Abdelaal, W.G.A., Saad, M.S. *et al.* (2014). Dynamic modeling and control of a quadrotor using linear and nonlinear approaches. (Cited on page 13.)

Henriques, B.S.M. (2011). Estimation and control of a quadrotor attitude. *PhD diss., Technical University of Lisbon*. (Cited on pages 32 and 33.)

Holkar, K. and Waghmare, L. (2010). An overview of model predictive control. *International Journal of Control and Automation*, vol. 3, no. 4, pp. 47–63. (Cited on page 17.)

Hsu, T.-R. (2002). Miniaturization–a paradigm shift in advanced manufacturing and education. In: *Internat ional conference on Advanced Manufacturing Technologies and Education in the 21st Century*. (Cited on page 1.)

Hu, T. and Lin, Z. (2001). *Control systems with actuator saturation: analysis and design*. Springer Science & Business Media. (Cited on page 20.)

Hu, T. and Lin, Z. (2003). Composite quadratic lyapunov functions for constrained control systems. *IEEE Transactions on Automatic Control*, vol. 48, no. 3, pp. 440–450. (Cited on page 24.)

Jardin, M.R. and Mueller, E.R. (2007). Optimized measurements of uav mass moment of inertia with a bifilar pendulum. In: *AIAA Guidance, Navigation and Control Conference and Exhibit, Hilton Head, SC, USA*. (Cited on page 81.)

Julier, S.J. and Uhlmann, J.K. (1997). A new extension of the kalman filter to nonlinear systems. In: *Int. symp. aerospace/defense sensing, simul. and controls*, vol. 3, pp. 182–193. Orlando, FL. (Cited on page 32.)

Kalman, R.E. *et al.* (1960). Contributions to the theory of optimal control. *Bol. Soc. Mat. Mexicana*, vol. 5, no. 2, pp. 102–119. (Cited on page 22.)

Kim, A. and Golnaraghi, M. (2004). Initial calibration of an inertial measurement unit using an optical position tracking system. In: *Position Location and Navigation Symposium, 2004. PLANS 2004*, pp. 96–101. IEEE. (Cited on page 9.)

Kwon, W.H. and Han, S.H. (2006). *Receding horizon control: model predictive control for state models*. Springer Science & Business Media. (Cited on page 18.)

Lau, M.S., Yue, S., Ling, K. and Maciejowski, J. (2009). A comparison of interior point and active set methods for fpga implementation of model predictive control. In: *Control Conference (ECC), 2009 European*, pp. 156–161. IEEE. (Cited on pages 28 and 29.)

Leong, B.T.M., Low, S.M. and Ooi, M.P.-L. (2012). Low-cost microcontroller-based hover control design of a quadcopter. *Procedia Engineering*, vol. 41, pp. 458–464. (Cited on page 2.)

Lim, H., Park, J., Lee, D. and Kim, H.J. (2012). Build your own quadrotor: Open-source projects on unmanned aerial vehicles. *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 33–45. (Cited on page 2.)

Luis, C. and Ny, J.L. (2016). Design of a trajectory tracking controller for a nanoquadcopter. *arXiv preprint arXiv:1608.05786*. (Cited on pages 84 and 85.)

Luukkonen, T. (2011). Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*. (Cited on pages 1, 12, and 14.)

Magnussen, Ø. and Skjønhaug, K.E. (2011). *Modeling, design and experimental study for a quadcopter system construction*. Master's thesis, Universitetet i Agder/University of Agder. (Cited on page 8.)

Meier, L., Honegger, D. and Pollefeys, M. (2015 may). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. (Cited on pages 35 and 36.)

Moradzadeh, M., Boel, R. and Vandevelde, L. (2014). Anticipating and coordinating voltage control for interconnected power systems. *Energies*, vol. 7, no. 2, pp. 1027–1047. (Cited on page 19.)

Mueller, M.W. and D'Andrea, R. (2013). A model predictive controller for quadrocopter state interception. In: *Control Conference (ECC), 2013 European*, pp. 1383–1389. IEEE. (Cited on page 17.)

Nagaty, A., Saeedi, S., Thibault, C., Seto, M. and Li, H. (2013). Control and navigation framework for quadrotor helicopters. *Journal of intelligent & robotic systems*, pp. 1–12. (Cited on page 12.)

Nebylov, A.V. and Watson, J. (2016). *Aerospace Navigation Systems*. John Wiley & Sons. (Cited on page 11.)

Nocedal, J. and Wright, S. (2006). Numerical optimization, series in operations research and financial engineering. *Springer, New York, USA, 2006*. (Cited on page 29.)

Palunko, I. and Fierro, R. (2011). Adaptive control of a quadrotor with dynamic changes in the center of gravity. *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 2626–2631. (Cited on page 3.)

Patel, K. and Barve, J. (2014). Modeling, simulation and control study for the quad-copter uav. In: *Industrial and Information Systems (ICIIS), 2014 9th International Conference on*, pp. 1–6. IEEE. (Cited on page 7.)

Pixhawk (2013). Home - pixhawk flight controller hardware project. `https://pixhawk.org/`. (Cited on pages 35, 78, and 79.)

QGroundControl (). Qgc - qgroundcontrol - drone control. `http://qgroundcontrol.com/`. (Accessed on 07/24/2018). (Cited on page 56.)

Quan, Q. (2017). *Introduction to multicopter design and control*. Springer. (Cited on page 9.)

Richalet, J., Rault, A., Testud, J. and Papon, J. (1978). Model predictive heuristic control: Applications to industrial processes. *Automatica*, vol. 14, no. 5, pp. 413–428. (Cited on page 17.)

Risqi, A.N., Hermanudin, A.A., Reksoprodjo, A.A., Muharram, A.P., Sunar, A., Baskoro, F.A.S., Muhamad, F., Fernanda, H., Utama, I.B.K.Y., Fitriani, L. *et al.* (). Makara 07-autonomous surface vehicle. (Cited on page 3.)

Rossiter, J.A. (2003). *Model-based predictive control: a practical approach*. CRC press. (Cited on pages 15, 18, 19, 20, and 23.)

Snyman, J. (2005). Practical mathematical optimization: Basic theory and gradient-based algorithms. (Cited on page 28.)

Thorat, S.R. (2015). *Quadcopter Flight Control using Modular Spiking Neural Networks*. Ph.D. thesis, Indian Institute of Technology, Bombay Mumbai. (Cited on page 7.)

Tongue, B. (2002). *Principles of Vibration*. Oxford University Press. ISBN 9780195142464.
Available at: `https://books.google.co.za/books?id=wAGqXVImUjYC` (Cited on page 82.)

Tosun, D., Isik, Y. and Korul, H. (2015). Comparison of pid and lqr controllers on a quadrotor helicopter. *International Journal of Systems Applications, Engineering and Development*, vol. 9. (Cited on page 3.)

Villbrandt, J. (2011). The quadrotors coming of age. *Illumin Magazine*, vol. 12. (Cited on page 1.)

Voise, J., Schindler, M., Casas, J. and Raphaël, E. (2011). Capillary-based static self-assembly in higher organisms. *Journal of The Royal Society Interface*, p. rsif20100681. (Cited on page 8.)

Wan, E.A. and Van Der Merwe, R. (2000). The unscented kalman filter for nonlinear estimation. In: *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pp. 153–158. Ieee. (Cited on page 32.)

Wang, L. (2009). *Model predictive control system design and implementation using MATLAB®*. Springer Science & Business Media. (Cited on pages 20, 21, 22, 24, 27, 28, 29, 30, 31, and 41.)

Wang, Y., Ramirez-Jaime, A., Xu, F. and Puig, V. (2017). Nonlinear model predictive control with constraint satisfactions for a quadcopter. In: *Journal of Physics: Conference Series*, vol. 783, p. 012025. IOP Publishing. (Cited on page 17.)