



PES University

(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560085, Karnataka, India

Report on
**Design and Implementation of
Robust Drone Control**

Submitted by

Aravind S (PES1201801734)
Pranay Mundra (PES1201801166)
Sanjay DV (PES1201801314)
Kavin Vignesh (PES1201800612)

January - August 2021

Under the guidance of

Dr. TS Chandar
Professor
Department of ECE

Faculty of Engineering
Department of Electronics and Communication, B.Tech



Certificate

This is to certify that the report entitled

Design and Implementation of Robust Drone Control

is a bonafide work carried out by

Aravind S (PES1201801734)
Pranay Mundra (PES1201801166)
Sanjay DV (PES1201801314)
Kavin Vignesh (PES1201800612)

In partial fulfillment for the completion of the course work in the program of study B.Tech in Electronics and Communication Engineering, under rules and regulations of PES University, Bengaluru during the period Jan - Sep 2021. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The report has been approved as it satisfies the academic requirements in respect of Capstone project work.

Dr. T. S. Chandar
Internal Guide

Dr. Anuradha M
Chairperson

Dr. B. K. Keshavan
Dean - Faculty of Engg. & Technology

Name and signature of the examiners:

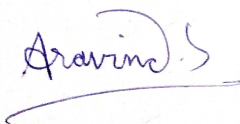
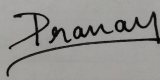
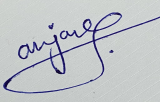

- 1.
- 2.

Declaration

We, **Aravind S, Pranay Mundra, Sanjay DV** and **Kavin Vignesh**, hereby declare that the report entitled *Design and Implementation of Robust Drone Control* is an original work done under the guidance of **Dr. TS Chandar**, Professor in the Dept of Electronics and Communication, and is being submitted as a partial requirement for completion of the course work in the B.Tech ECE Program of study during Jan-May 2021.

Place: PES University Ring Road, Bengaluru

Date: 21/05/2021

Name of the student	Signature
Aravind S	
Pranay Mundra	
Sanjay DV	
Kavin Vignesh	

Contents

1	Introduction	2
1.1	Objective	2
1.2	Methodology	2
2	Hardware and Software	4
2.1	Hardware	4
2.2	Software	4
3	Block Diagram	5
4	Literature Survey	6
5	Drone Dynamics and Mathematical Modelling	8
5.1	Quadrotor system model	8
5.2	Convention of the angles	9
5.3	The state space representation	10
6	Overview of Controllers	11
6.1	Proportional-Integral-Derivative Controller(PID)	11
6.1.1	Mathematical form of PID	12
6.1.2	Simulation results on MATLAB:	12
6.2	Linear Quadratic regulator	12
6.2.1	General description of LQR	12
6.2.2	Mathematical model	13
6.3	Model Predictive Control	13
6.3.1	Idea Behind MPC	13
6.3.2	Principles of MPC	14
6.4	Sliding Mode Control	15
6.5	Uncertainty and disturbance Estimator	15
6.5.1	Derivation of the UDE Control Law for a second order system	16
7	Design of Cascaded Control for a Quadrotor	17
7.1	Small Angle Control	17



7.1.1	Description	17
7.1.2	Derivation of Small Angle Controller	17
7.1.3	Calculation of desired Roll and Pitch	18
7.1.4	Thrust force controller	18
7.1.5	Controller for torque about X-axis	18
7.1.6	Controller for torque about Y-axis	19
7.1.7	Controller for torque about Z-axis	19
7.2	Virtual Forces	19
7.2.1	Description	19
7.2.2	Derivation of Virtual Forces based Controller	20
7.2.3	X-axis virtual force controller	20
7.2.4	Y-axis virtual force controller	21
7.2.5	Z-axis virtual force controller	21
7.2.6	Controller for torque about X-axis	21
7.2.7	Controller for torque about Y-axis	21
7.2.8	Controller for torque about Z-axis	22
8	Simulink implementation and Results	23
8.1	Physical Parameters of PlutoX	23
8.2	Small angle controller	24
8.3	Virtual force controller	29
9	Hardware implementation	34
9.1	Whycon marker	34
9.2	ROS - Robot Operating System	34
9.3	Python-based controllers	36
9.3.1	PID controller with <code>rospy</code>	36
9.3.2	UDE controller with <code>rospy</code>	36
9.3.3	Hardware Implementation of UDE controller through ROS interface	37
9.3.4	UDE controller with Cygnus IDE	37
10	Analysis and Conclusion	39
10.1	Analysis	39
10.2	Conclusion	39
A	MATLAB scripts	42
A.1	Small Angle controller	42
A.1.1	<code>init.m</code>	42
A.1.2	<code>mfnu_outer.m</code>	43
A.1.3	<code>aux1.m</code>	43

A.1.4	<code>mfnu_inner.m</code>	44
A.1.5	<code>aux2.m</code>	44
A.2	Virtual force controller	45
A.2.1	<code>init.m</code>	45
A.2.2	<code>mfnu_outer.m</code>	45
A.2.3	<code>aux1.m</code>	46
A.2.4	<code>mfnu_inner.m</code>	47
A.2.5	<code>aux2.m</code>	47
A.3	Utilities	47
A.3.1	<code>plant.m</code>	47
A.3.2	<code>clamper.m</code>	48
A.3.3	<code>graphResults.m</code>	48
B	Python modules	50
B.1	PID Controller	50
B.1.1	<code>pluto_utils.py</code>	50
B.1.2	<code>pluto_pid_control.py</code>	51
B.2	UDE Controller	52
B.2.1	<code>main.py</code>	52
B.2.2	<code>outer_loop_controller.py</code>	54
B.2.3	<code>inner_loop_controller.py</code>	57
B.2.4	<code>params.py</code>	60
B.3	Utilities	60
B.3.1	<code>integral.py</code>	60
B.3.2	<code>android_cam.py</code>	61
B.3.3	<code>logger.py</code>	62
B.3.4	<code>plotter.py</code>	63
C	C++ scripts	64
C.1	UDE based Attitude Controller	64
C.1.1	<code>PlutoPilot.h</code>	64
C.1.2	<code>PlutoPilot.cpp</code>	64
C.1.3	<code>UDE.h</code>	66
C.1.4	<code>UDE.cpp</code>	66
C.1.5	<code>params.h</code>	68

Abstract

This report describes the design and implementation of robust drone control using the Uncertainty and Disturbance Estimator (UDE) to a lightweight, commercial quadrotor (UAV/drone). Application of the concept of virtual forces along with small angle assumptions are augmented by a robust UDE controller. The approach is validated using MATLAB/Simulink simulations and drone flight tested indoors. The controller is found to offer increased robustness to parametric uncertainties, disturbances and varying reference points. ROS1 source codes using `rospy` are supplemented to support hardware implementation with the PlutoX drone and WhyCon marker setup. We finally discuss the comparative results of both small angle and virtual force based controllers. It is inferred that lightweight implementations of this control scheme could increase the performance of typical PID controllers on-board commercial drones.

Chapter 1

Introduction

A quadrotor is a miniaturized Unmanned Aerial Vehicle (UAV) that may operate autonomously. The configuration has recently become common for small-scale unmanned aerial vehicles (drones) in recent decades. Quadcopter research has risen in response to the need for aircraft with greater maneuverability and hovering ability. Quadcopters can be relatively easy while still being highly reliable and maneuverable thanks to the four-rotor design.

Over the years, the quadcopter template has become widespread for small-scale drones. The necessity for aircraft with greater maneuverability and hovering ability has led to a rise in quadcopter research. The four-rotor design allows quadcopters to be relatively simple in design yet highly reliable and maneuverable. Technological developments in onboard electronics have allowed the production of cheap lightweight flight controllers, accelerometers (IMU), gyroscopes, GPS, and cameras. With their miniature form factor and agility, quadrotors can be flown indoors as well as outdoors. [1]

The ability to responsively control a drone during its flight is a prerequisite towards autonomous control and higher-level API commands such as directing it from point to point. This API forms the basis for trajectory planning algorithms that work by giving the drone a set of points to traverse in a well-defined order. Robust control in this regard is needed between two given points to minimize flight time, increase stability and achieve path planning.

Commercial Controllers used today depend on a linearized model that does not consider model uncertainties and external disturbances such as wind gusts. Stability is lost as well when the roll and pitch take values outside a certain interval. Reliable and smooth control maximizes total flight time and saves power.

1.1 Objective

Our objective is to design and implement a robust control architecture on a drone (quadcopter) for stabilization and movement between two given points in space. The design will be carried out in two phases, the first of which is on MATLAB under idealized circumstances with noise generated from a suitable probability distribution. The errors and costs incurred during software simulation will be computed and inspected to determine the controller to take forward to the next phase. The second phase involves porting the controller with the best response characteristics onto hardware via the ROS interface. The final response curves of the controller output will be benchmarked against similar drone models with different control architectures.

1.2 Methodology

1. Upon selecting a suitable quadcopter model to carry out our research, we will begin to model its state, kinematics, and dynamics using concepts from physics such as laws of motion and rotational dynamics. Once a system model is formalized, it will be implemented in MATLAB as a nonlinear model. Future states of the model are predicted by using a solver such as `ode4`



which solves differential equations of the system model. Utilities to represent the simulation shall be included such as graph plotters and logs.

2. To familiarize ourselves at the outset of classical techniques in control theory, we will be implementing PID feedback loops in MATLAB for both inner and outer loops of the system. Understanding the sensitivity of the response towards constants in these feedback loops will help us gain insight into the control problem as a whole.
3. Following this, nonlinear control strategies such as UDE, MPC, and SMC will be reviewed extensively and implemented on MATLAB either on their own or in conjunction with earlier classical techniques to deal with the effects of added noise and disturbance on the system's state. Effectiveness in this regard would classify such a control method as robust which is our ultimate aim here. Additional modifications to the core control algorithm may be added for increased performance.
4. The controller with the most ideal response characteristics under noisy circumstances will be chosen for implementation on hardware. This will be achieved via the ROS interface (language agnostic) or the Cygnus IDE for Pluto (C++ scripts). Communication with the drone itself is done over WiFi using an onboard ESP12F.
5. The finalized error and cost plots will be benchmarked against drone models with similar specifications of motors and weight.

Chapter 2

Hardware and Software

2.1 Hardware

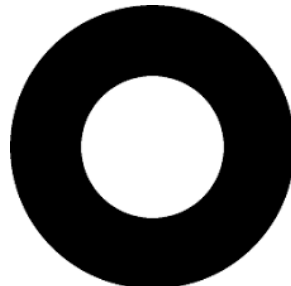
1. The quadrotor model used is the **PlutoX** model with WhyCon marker integration. PlutoX is an open-source, DIY, crash-resistant drone kit designed by IIT Bombay-based startup DronaAviation.
2. **WhyCon** is a low-cost vision-based localization device that achieves millimeter precision and good performance. The marker is used for the indoor localization of the drone. [2]
3. **Desktop computer** with mid-range consumer performance.

2.2 Software

1. The reviewed robust control approaches are simulated using the **MATLAB/SIMULINK** ecosystem during the first phase of simulations.
2. **ROS** is a meta operating system designed for interaction between robots and robotic applications. It is open source and offers features such as low level hardware abstraction, device management, and message passing between processes/nodes.
3. **Cygnus IDE** is an environment in which the on-board microcontroller of the PlutoX can be programmed using C++. The user can write programs to implement a custom controller and flash the firmware.



(a) PlutoX Drone



(b) WhyCon Marker

Figure 2.1: Hardware used

Chapter 3

Block Diagram

The desired x, y, z position and yaw(ψ) values are provided as a reference input to the outer loop position controller. This high-level position controller takes these values and converts them to the desired values of θ, ϕ and ψ (roll, pitch, and yaw) for the attitude controller needed to tilt the drone in the direction of the desired coordinates.

The IMU onboard the drone give the estimated values of the angular velocity and acceleration. The WhyCon marker placed on the drone gives an estimated value of the position of the drone. This is used as feedback to the nonlinear controller in the inner attitude loop which finds optimal values of the torques about the x, y and z axes to reach the desired goal coordinates. [1]

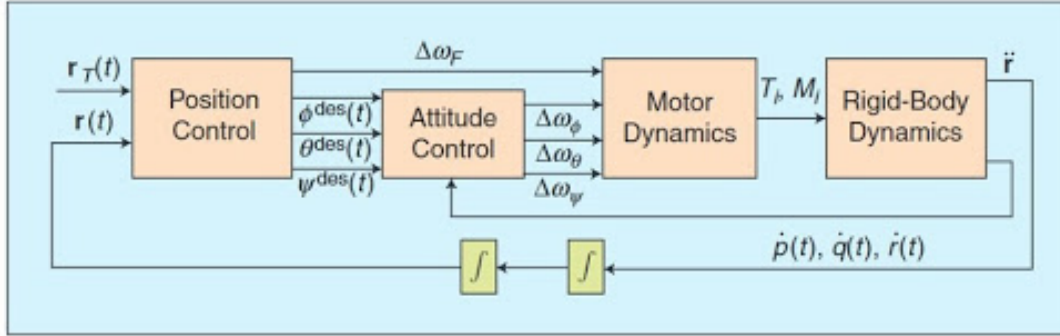


Figure 3.1: Cascaded control system for position and attitude control

Chapter 4

Literature Survey

The following papers were surveyed from which certain conclusions were drawn:

1. N.Michael, D.Mellinger, V.Kumar, **The GRASP Multiple Micro-UAV Test Bed**, IEEE Robotics, and Automation Magazine, 2010 [3]

An extensive and detailed overview of the dynamic model for the quadrotor was discussed. Testbed design considerations and choice of drone models were also highlighted. A nominal PID control was used to track a set point whereas a nonlinear SMC (Sliding Mode Control) was used for trajectory control. The importance of 3D trajectory generators such as spline and the minimum snap was also discussed.

2. Andrew Gibiansky, **Quadcopter Dynamics, Simulation and Control**, Personal Blog, 2012

An in-depth review of quadcopter dynamics, kinematics, rotational dynamics, and equations of motion governing the structure's motion in three dimensions was conducted. Rudimentary PD and PID control loops were designed and simulated on MATLAB. Additionally, a regression-based approach for online tuning of PID constants was also given.

3. M. A. Garratt, S. G. Anavatti, I. A. P. Santos, **Robust Hybrid Nonlinear Control Systems for the Dynamics of a Quadcopter Drone**, IEEE Transactions on Systems, MAN and Cybernetics, 2018 [4]

A discussion on the importance of robust control followed by nonlinear drone dynamics and MIMO transfer functions was done. An MPC/fuzzy feedforward control architecture used for trajectory tracking was shown to be substantially better than a conventional PD control. This paper highlighted the importance of methods such as Model Predictive Control in minimizing errors.

4. A. L'Afflitto, R. B. Anderson, K. Mohammadi, **An Introduction to Nonlinear Robust Control for Unmanned Quadrotor Aircraft**, IEEE Control Systems Magazine, 2018 [1]

This paper showcases a Sliding Mode Control approach to stabilizing a drone and movement between points with the desired yaw angle. In sliding mode control, trajectories of the system are steered to a subspace of the state space in finite time. Once on this sliding surface, the system can asymptotically converge to the origin without regard to matched uncertainties. It is understood to be a fairly straightforward controller with no optimization for the cost involved, just plain matrix operators.

5. Y. Wang, S. Boyd, **Fast Model Predictive Control Using Online Optimization**, IEEE Transactions on Control Systems Technology, 2010 [5]

Model predictive control generally uses an interior point method-based solver which can drastically decrease the real-time response of the controller. This paper seeks to increase performance by describing a few methods using online optimization. Infeasible Newton Start,



Fast computation of Newton step, and Warm start are nifty optimizations that exploit the structure of the MPC problem. An implementation in C was supplemented to showcase results on a simple and random system model.

6. D.D. Dhadekar, P.D. Sanghani, S.E. Talole, **Robust Control of Quadrotor using Uncertainty and Disturbance Estimation**, Journal of Intelligent and Robotic Systems, 2021 [6]

Explanation of the design of a nonlinear dynamic inversion controller for drones which was made robust using an Uncertainty and Disturbance Estimator (UDE) approach in both inner and outer loops was given. The design did not need a priority knowledge of the bounds in uncertainty and small-angle assumptions of roll and pitch were relaxed. The desired roll and pitch angle trajectories were generated using UDE. Monte Carlo simulations as well as experimental validations revealed that the proposed design seemed viable for robust control.

7. Karthik PB, Keshav Kumar, **Reinforcement Learning for Altitude Hold and Path Planning on a Quadrotor**, IEEE Intl. Conference on Control, Automation and Robotics, 2020 [7]

This paper showed the use of Reinforcement Learning techniques for altitude holding and path planning in a quadrotor. A simulation in the VRep simulator was performed followed by implementation on a Pluto X drone which is the same model that we have chosen.

8. M. Kumar, P. Sharma, **Trajectory Planning of Unmanned Aerial Vehicle using Invasive Weed Optimization**, IEEE ICESC, 2020 [8]

This paper used an Invasive Weed Optimization method for planning out trajectories for quadrotors in a war field scenario. The algorithm was run on a self-generated problem set and showed promising results in obtaining the optimal trajectories.

Chapter 5

Drone Dynamics and Mathematical Modelling

5.1 Quadrotor system model

This section explains the formulation of the quadrotor system model which is used in the MATLAB simulation and also for the creation of the controllers.

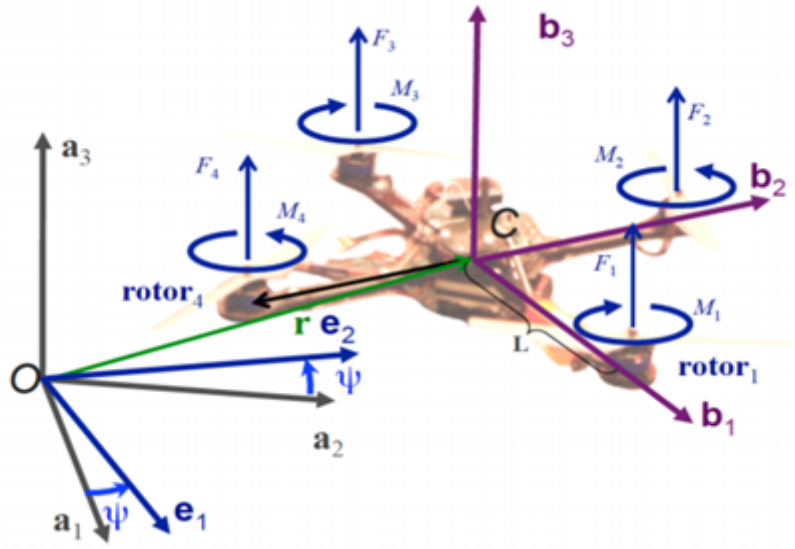


Figure 5.1: Quadrotor model with the body-fixed and inertial reference frames

The system here has two frames - The inertial frame, attached to the starting point of the drone and the body frame, attached to the center of mass of the drone. Gravity points in the negative z-direction.

- **Frame A** - a_1, a_2, a_3 is the inertial frame (Earth fixed frame)
- **Frame B** - b_1, b_2, b_3 is the body-fixed frame of the drone

The origin of the body-fixed frame coincides with the drone's center of mass. The direction along axis b_1 is the preferred forward direction. Axis b_3 is always perpendicular to the plane of the rotors, pointing vertically upwards during perfect hover. [3]



5.2 Convention of the angles

[9] Pitch (θ), roll (ϕ), and yaw (ψ) are referred to as the Euler angles.

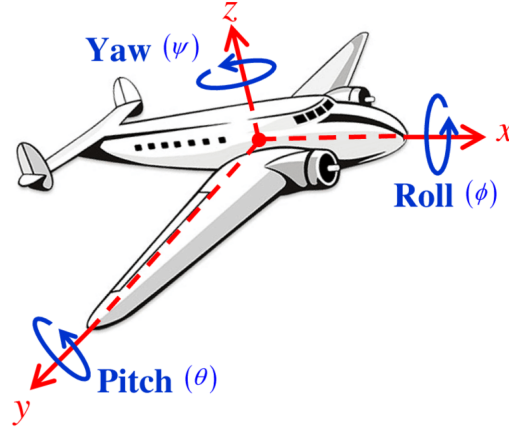


Figure 5.2: Representation of Euler angles

Euler angles provides us with a way to represent the 3D orientation in space of an object using a combination of 3 rotations about different orthonormal axes.

These angles are defined with respect to the body frame. Body and inertial frames are related by a rotation matrix \mathbf{R} . The rotation matrix maps vectors in the body-fixed frame to the inertial frame. This is useful when modeling the state of the system. The rotation matrix is calculated using the Z-Y-X convention, where the first rotation is about the Z axis called *yaw*, this is followed by a rotation about the Y axis called *pitch* and finally a rotation about the X axis called *roll*. All the rotations here are done with respect to the intermediate frames and not the inertial frame. The final rotation matrix is calculated as the product of individual rotation matrices about each axis.

- The rotation matrix for Z-Y-X convention is given as:

$${}^A_B\mathbf{R} = \begin{bmatrix} \cos \psi \cos \theta - \sin \phi \sin \theta \sin \psi & -\cos \phi & \cos \psi \sin \theta + \cos \theta \sin \phi \sin \psi \\ \cos \theta \sin \psi + \cos \phi \sin \theta \sin \psi & \cos \phi \cos \psi & \sin \psi \sin \theta - \cos \psi \cos \theta \sin \phi \\ -\cos \phi \sin \theta & \sin \phi & \cos \phi \cos \theta \end{bmatrix}$$

- From the Newton's Equations of motion we get:

$$m\ddot{\mathbf{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + {}^A_B\mathbf{R} \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix}$$

Where \mathbf{r} is the position vector of the drone.

F_1, F_2, F_3, F_4 indicates the upward thrust from each of the rotors, and $-mg$ indicates the weight of the drone itself acting downwards.

- From Euler's equations of Rotation we get:

$$\mathbf{I} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \mathbf{I} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$



This equation relates the torque to the angular velocity, angular acceleration, and moment of inertia matrix and is obtained from the following general expression:

$$\tau = \mathbf{I}\dot{\omega} + \omega \times \mathbf{I}\omega$$

Here p , q and r are the angular velocities of the drone in the body-fixed frame. I is the matrix for the moment of inertia. L is the length of the arm of the quadrotor. M_i is the momentum about the z-axis due to motor i .

The angular velocity of the drone in the body frame is related to velocity in the inertial frame by the following equation:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & -\cos \phi \sin \theta \\ 0 & 1 & \sin \phi \\ \sin \theta & 0 & \cos \phi \cos \theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

The moment of inertia matrix is given by

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

This matrix is diagonal because the drone is symmetric and no cross-coupling assumption has been made.

5.3 The state space representation

The state vector is given by

$$\mathbf{x} = [x \ y \ z \ \dot{x} \ \dot{y} \ \dot{z} \ \phi \ \theta \ \psi \ p \ q \ r]^\top$$

The derivative of the same is a non-linear function of the current state and the control input.

$$\dot{\mathbf{x}} = f(\phi, \theta, \psi)x + g(\phi, \theta, \psi)u$$

The control input is defined by

$$\mathbf{u} = [F_t \ \tau_x \ \tau_y \ \tau_z]^\top$$

Chapter 6

Overview of Controllers

6.1 Proportional-Integral-Derivative Controller(PID)

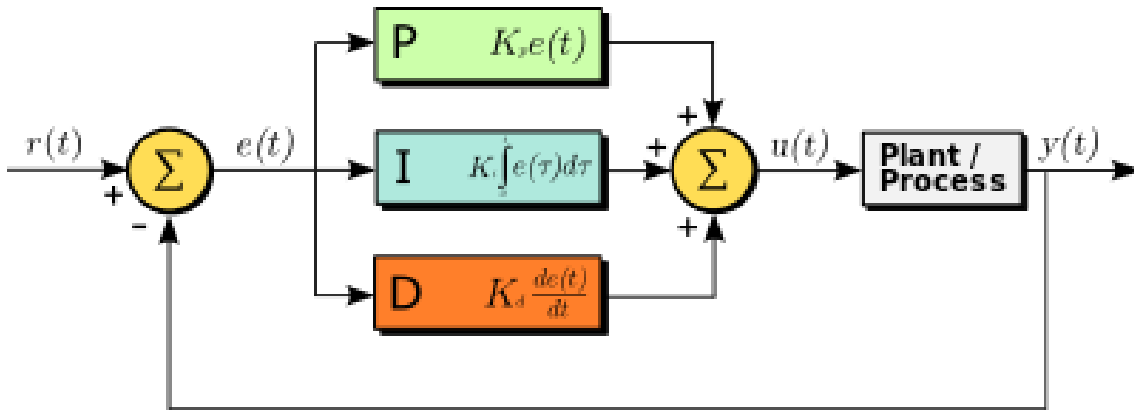


Figure 6.1: PID

PID is short for Proportional, Integral, and Derivative. It is a common control loop technique with a feedback mechanism and is used widely in applications that require continuous and modulated control. It automatically applies a correction mechanism to correct a control function from deviations. The main and distinctive feature of the PID controller is the ability to use its proportional, integral, and derivative effects in an abstract manner on the controller output to apply accurate and optimal control. The block diagram above shows a PID controller which continuously calculates the error, $e(t)$. The error term is the difference between the reference output and the actual output as obtained from the PID controller, i.e. $e(t) = r(t) - y(t)$, and attempts to decrease it based on the proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjustment of a control variable $u(t)$ to a new value determined by a weighted sum of the control terms. [3, 10, 9, 11]

- Term **P** is proportional to the current value of the error weighted by K_p . The proportional controller alone will result in an error between the actual value and the desired set value because an error is always required by the controller to generate a proportional and apt response, hence if there is no error then there is no corrective response. Therefore, if the error is large and non-negative then the output is proportionally large and non-negative.
- The integral term **I** is used to attenuate the steady-state error by keeping a track of the cumulative error of the past. It adds a control effect to any residual error after the operation of the proportional control.
- The **D** term is used to obtain the best estimate of the future trend of the error term based on the rate of change of the current value of error. The greater the rate of change, the greater is the damping effect applied by the term.



6.1.1 Mathematical form of PID

- The overall function is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(t) dt + K_d \frac{de(t)}{dt}$$

K_i , K_p , K_d are all non-negative values denoting the coefficients for the integral, proportional, and derivative terms respectively.

6.1.2 Simulation results on MATLAB:

The PID controller was implemented on MATLAB for the drone model obtained in section 5.3. The following response graphs indicate one-dimensional control using a nominal PD controller. The drone model assumed does not involve cross-coupling and has a mass of about 180g with an arm length of 86mm. The weights of K_p and K_d were tuned to give a response whose rise time was slightly over 1 second and settling time to be about 3 seconds. The overshoot obtained was around 10% which could be further improved by finer tuning of the derivative term.

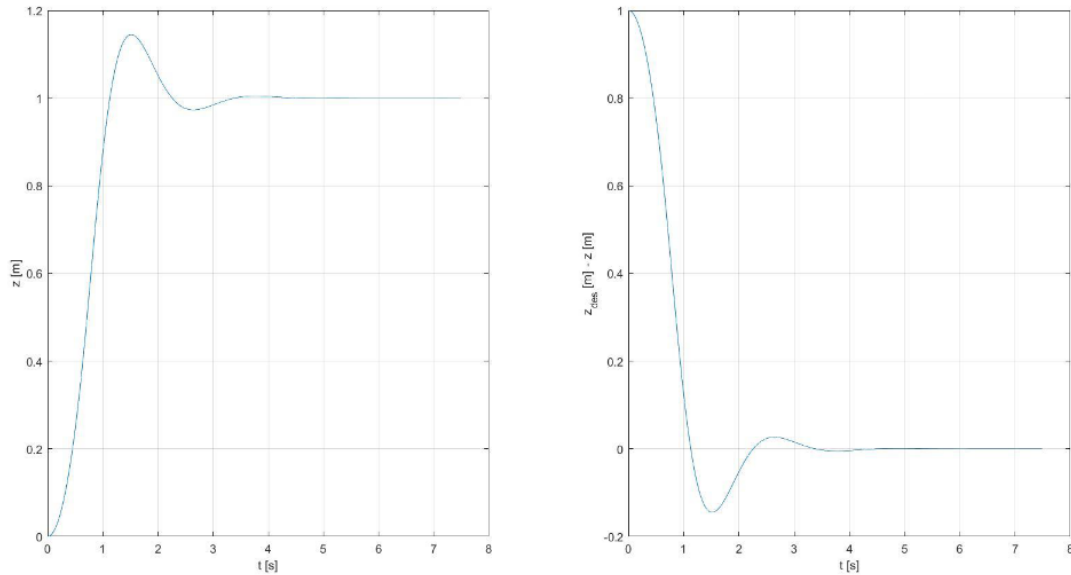


Figure 6.2: 1D PD Output

6.2 Linear Quadratic regulator

LQR stands for linear quadratic regulator. Optimal control mainly is concerned with operating a system dynamically and minimizing the cost. A system whose dynamics are described by a set of linear differential equations where the cost is a quadratic function is said to be an LQ problem. The solution to an LQ problem is provided by LQR.

6.2.1 General description of LQR

The settings of a regulating controller controlling a process are found by a mathematical process that minimizes the cost function of the process. The weights are manually computed before hand and are not a part of the algorithm. The cost function is most often the sum of all the errors of deviations between the desired output and the actual output. The algorithm thus helps in finding the necessary control settings to minimize the undesired deviations from the desired value. The engineer still has to specify the parameters for the cost functions and compare the results obtained with the desired goal to modify the parameters to get the closest possible output iteratively. The main drawback of LQR is that it's very difficult to find a set of proper weights. [12]

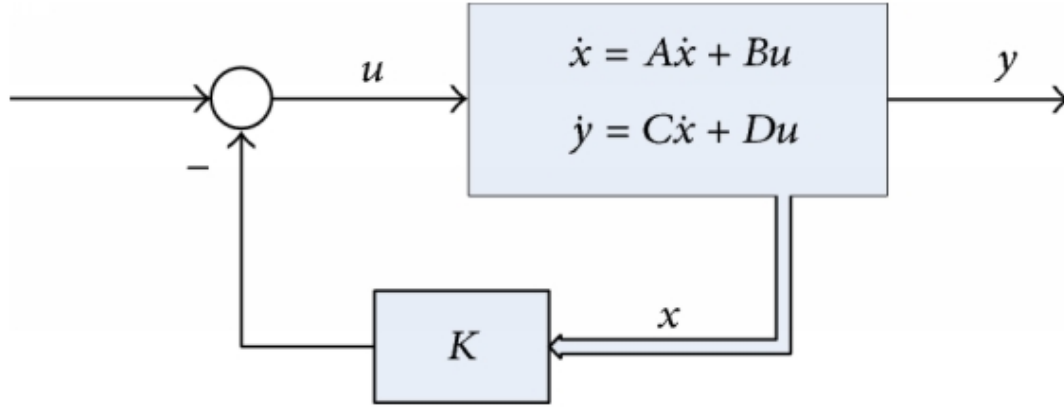


Figure 6.3: LQR scheme

6.2.2 Mathematical model

For a continuous-time linear system, defined on $t \in [t_0, t_1]$ described by $\dot{x} = Ax + Bu$ with the quadratic cost function defined as:

$$J = x^T(t_1)F(t_1)x(t_1) + \int_{t_0}^{t_1} (x^T Q x + u^T R u + 2x^T N u) dt$$

The control law which is fed back that minimizes the value of the cost is,

$$u = -Kx$$

Where K is given by,

$$K = R^{-1} (B^T P(t) + N^T)$$

6.3 Model Predictive Control

Model predictive control (MPC) is a form of control technique for MIMO systems that uses a series of constraints to control a process while simultaneously applying an optimizer at each step. Model predictive controllers are based on dynamic and changing models of the mechanism, which are typically linearized models obtained by perturbing about a fixed point. MPC can foresee future events and take appropriate control measures, unlike PID. [13]

The control algorithm is based on :

- Optimization of the problem is done numerically at each step
- Constrained optimization - typically Linear programming(LP) or Quadratic programming.
- Receding Horizon Control.

6.3.1 Idea Behind MPC

MPC focuses on optimization based on a plant model, finite horizon in nature, and iterative methods . The sampling of current plant state is done at time t and a control strategy to minimize cost for a short duration into the future is computed . An online calculation is employed to state trajectory exploration that originates from the current state and a control strategy to minimize cost is found until time $(t + T)$. The sampling of the plant state is done after the first step of the control strategy is applied and using the new current state the calculations will be repeated from which we can deduce the expected direction of the state. MPC is also known as receding horizon regulation since the prediction horizon is constantly shifted forward. [5]

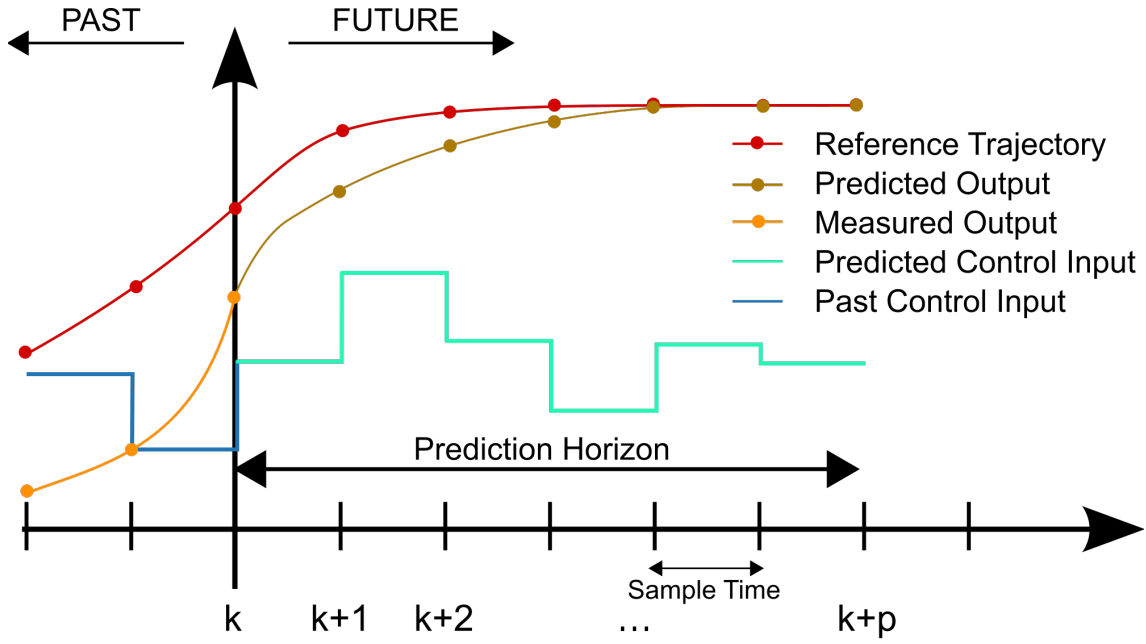


Figure 6.4: Typical MPC Graph

6.3.2 Principles of MPC

Model Predictive Control (MPC) is a multivariable control algorithm that employs the following techniques,

- A dynamic internal model of the process.
- Over the receding horizon, a cost function J will be computed.
- An optimization procedure that uses the control input u to minimise the cost function J .

An example of quadratic cost function for optimization is given by [5]

$$J = \sum_{i=1}^N w_{x_i} (r_i - x_i)^2 + \sum_{i=1}^N w_{u_i} \Delta(u_i)^2 \quad (6.1)$$

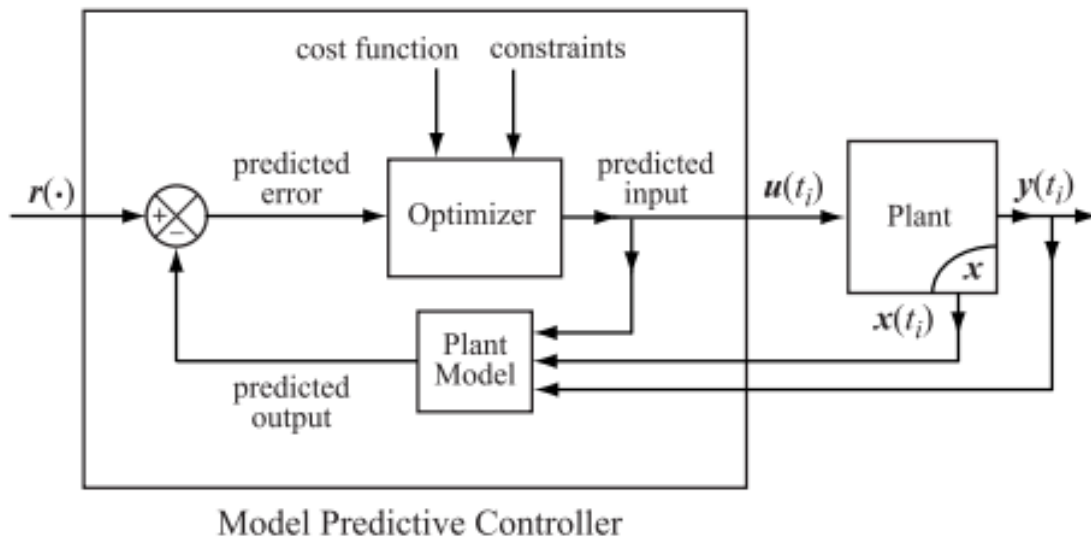


Figure 6.5: Block diagram of MPC



6.4 Sliding Mode Control

Sliding mode control (SMC) is a control architecture, nonlinear in nature, that involves applying a piece-wise continuous control signal to a nonlinear system and forcing it to "slide" over a cross-section of the system's regular behaviour. The state-feedback control law is not a time-dependent function. Instead, depending on the current state space trajectory, it can move from one continuous structure to another. The many control structures are arranged in such a way that trajectories always travel toward a neighbouring region with a different control structure, and the eventual trajectory will not be totally contained within one control structure. Rather, it will slide along the boundaries or edges of the numerous control structures. The system's motion as it slides along these boundaries is referred to as sliding mode. [14, 12]

In Shaik [12], four scalar sliding surfaces are defined as,

$$s_\alpha = \dot{e}_\alpha + \lambda_\alpha e_\alpha = 0 \quad (6.2)$$

where, λ_α is the tuning gain and the relevant tracking error is defined as,

$$e_\alpha = \alpha^d - \alpha \quad (6.3)$$

The variable α is used as a placeholder for one of the relevant states: ϕ , θ , ψ or z . The time derivative of the sliding manifold is

$$\dot{s}_\alpha = -\dot{f} - u + \ddot{\alpha}^d + \lambda_\alpha \dot{e}_\alpha \quad (6.4)$$

The sliding mode controller's goal is to select an input u for each option of α such that the system trajectories are driven in limited time to the sliding manifold s_α , which is subsequently driven asymptotically to zero. This is accomplished by selecting the following control inputs, each with a scalar tuning gain of k_α :

$$u_1 = -\frac{m}{\cos \phi \cos \theta} \left[-g + \ddot{r}_r^d + \lambda_{r_z} \dot{e}_{r_z} + k_{r_z} \text{sgn}(s_{r_z}) \right] \quad (6.5)$$

$$u_2 = J_x \left[-\frac{J_y - J_z}{J_x} qr + \ddot{\phi}^d + \lambda_\phi \dot{e}_\phi + k_\phi \text{sgn}(s_\phi) \right] \quad (6.6)$$

$$u_3 = J_y \left[-\frac{J_z - J_x}{J_y} pr + \ddot{\theta}^d + \lambda_\theta \dot{e}_\theta + k_\theta \text{sgn}(s_\theta) \right] \quad (6.7)$$

$$u_4 = J_z \left[-\frac{J_x - J_y}{J_z} pq + \ddot{\psi}^d + \lambda_\psi \dot{e}_\psi + k_\psi \text{sgn}(s_\psi) \right] \quad (6.8)$$

6.5 Uncertainty and disturbance Estimator

The primary concept underlying this method is to 'estimate' the uncertainty and disturbance as a whole, then cancel their impacts. Another distinguishing aspect of this method is that it does not necessitate any prior knowledge of the uncertainty and/or disruption, such as magnitude or boundaries. Only the frequency distribution of the disturbance is necessary. The estimation is done online dynamically and compensated. Most of the other famous robust control strategy such as 'sliding mode control' require both upper and lower bounds in the uncertainty/disturbance.

Uncertainties in plant dynamics are estimated dynamically. The controller is designed to compensate for the effects of these uncertainties and unknown external disturbances. The required dynamics can be obtained by any simple controller (like, PID, LQR). It can be used for linear and non-linear systems. [6]



6.5.1 Derivation of the UDE Control Law for a second order system

Consider a second order system with uncertainties in plant parameters defined as:

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= (a_1 + \Delta a_1)x_1 + (a_2 + \Delta a_2)x_2 + (b + \Delta b)u + d\end{aligned}\quad (6.9)$$

Where $\Delta a_1, \Delta a_2$ and Δb are the uncertainties and d is an external unknown disturbance.

Lumping the uncertainties and disturbance into D , we can write

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= a_1x_1 + a_2x_2 + bu + D\end{aligned}\quad (6.10)$$

Assume \hat{D} to be the estimate of D after passing it through a first order filter $G_f(s)$. We get,

$$\hat{D} = G_f(s)D \quad (6.11)$$

$$G_f(s) = \frac{1}{1 + s\tau} \quad (6.12)$$

Redefining the control u as:

$$u = \frac{1}{b}(u_a + \nu + u_d) \quad (6.13)$$

where,

- b is the coefficient of input of the system model.
- u_a compensates for the internal model dynamics
- ν is another control law (like PD, LQR)
- u_d compensates for uncertainties and disturbances

Since \hat{D} is the estimate of D , we define $u_d = -\hat{D}$. Applying the control with the definition of u_d , would result in the dynamics as

$$\ddot{x}_1 + k_1\dot{x}_1 + k_0x_1 = D - \hat{D} \quad (6.14)$$

The lumped term D can also be written as,

$$D = \dot{x}_2 - a_1x_1 - a_2x_2 - bu \quad (6.15)$$

Define ν for a tracking controller as follows:

$$\nu = \dot{x}_{2ref} + k_1(x_{2ref} - x_2) + k_2(x_{1ref} - x_1) \quad (6.16)$$

Substituting for u (6.5) in the expression for D (6.7) as follows,

$$D = \dot{x}_2 - u_d - \nu \quad (6.17)$$

Substitute for \hat{D} (6.3) and u_d in the above expression and solve for u_d ,

$$u_d = -\frac{1}{\tau}(x_2 - \int_0^t \nu dt) \quad (6.18)$$

Thus with these equations for ν (6.8) and u_d (6.10), we are ready to derive the equations for the control inputs in 7

Chapter 7

Design of Cascaded Control for a Quadrotor

This chapter discusses the design of the cascaded control shown in [3](#). An Uncertainty and Disturbance Estimator(UDE) based controller was chosen for detailed analysis due to its ease of implementation and low computation cost.

Two different approaches for controller design using UDE were tested:

- Small Angle Control
- Virtual Forces based control

7.1 Small Angle Control

7.1.1 Description

This approach is based on [\[6\]](#). Here, a linearized model of the system is used. The linearization is achieved by making the following assumptions:

- The roll, pitch and yaw of the drone, that is ϕ , θ and ψ , are assumed to be very small
- The angular rates, p , q , and r are approximated to 0

The above assumptions are reasonable when the trajectory is simple (non-curved) of the drone requires no aggressive maneuvers.

7.1.2 Derivation of Small Angle Controller

Due to the small angle assumption, the following trigonometric relations hold true:

$$\sin \theta \approx \theta \quad \cos \theta \approx 1$$

$$\sin \phi \approx \phi \quad \cos \phi \approx 1$$

$$\sin \psi \approx \psi \quad \cos \psi \approx 1$$

Also, the angular rates are approximated to 0 i.e. $\dot{\phi} = \dot{\theta} \approx 0$.

Using the above relations, the UDE controller expression in section 6.5 and the dynamics derived in Chapter 5, the controller expressions for the torques of the drone and the desired values of roll and pitch can be derived.



7.1.3 Calculation of desired Roll and Pitch

The desired roll, ϕ , is calculated as:

$$\phi_{des} = \frac{-1}{g}(\nu_x + u_{dx})$$

Where,

$$\nu_x = \ddot{x}_{ref} + k_1(\dot{x}_{ref} - \dot{x}) + k_2(x_{ref} - x)$$

$$u_{dx} = -\frac{1}{\tau}(\dot{x} - \int \nu_x dt)$$

The desired value of pitch, θ , is calculated as:

$$\theta_{des} = \frac{1}{g}(\nu_y + u_{dy})$$

where,

$$\nu_y = \ddot{y}_{ref} + k_1(\dot{y}_{ref} - \dot{y}) + k_2(y_{ref} - y)$$

$$u_{dy} = -\frac{1}{\tau}(\dot{y} - \int \nu_y dt)$$

7.1.4 Thrust force controller

Using the UDE equations derived earlier for a second order system and the dynamics for the virtual force in z , we obtain

$$F_t = m [g + \nu_z + u_{dz}]$$

where,

$$\nu_z = \ddot{z}_{ref} + k_1(\dot{z}_{ref} - \dot{z}) + k_2(z_{ref} - z)$$

$$u_{dz} = -\frac{1}{\tau}(\dot{z} - \int \nu_z dt)$$

7.1.5 Controller for torque about X-axis

The dynamics for the angular velocity about the x-axis is given by:

$$\dot{p} = \frac{\tau_x}{I_{xx}} - \frac{qr}{I_{xx}}(I_{zz} - I_{yy}) + D$$

Using the UDE equations derived earlier for a second system, the control input τ_x is defined as:

$$\tau_x = I_{xx} \left[\frac{qr}{I_{xx}}(I_{xx} - I_{yy}) \right] + \nu_{\tau_x} + u_{d\tau_x}$$

Where,

$$\nu_{\tau_x} = \dot{p}_{ref} + k_1(p_{ref} - p) + k_2(\phi_{ref} - \phi)$$

$$u_{d\tau_x} = -\frac{1}{\tau}(p - \int \nu_{d\tau_x} dt)$$



7.1.6 Controller for torque about Y-axis

The dynamics for the angular velocity about the y-axis is given by:

$$\dot{q} = \frac{\tau_y}{I_{yy}} - \frac{pr}{I_{yy}}(I_{xx} - I_{zz}) + D$$

Using the UDE equations derived earlier for a second order system, the control input τ_y is defined as:

$$\tau_y = I_{yy} \left[\frac{pr}{I_{yy}}(I_{xx} - I_{zz}) + \nu_{\tau_y} + u_{d\tau_y} \right]$$

Where,

$$\begin{aligned} \nu_{\tau_y} &= \dot{q}_{ref} + k_1(q_{ref} - q) + k_2(\theta_{ref} - \theta) \\ u_{d\tau_y} &= -\frac{1}{\tau}(q - \int \nu_{\tau_y} dt) \end{aligned}$$

7.1.7 Controller for torque about Z-axis

$$\dot{r} = \frac{\tau_z}{I_{zz}} - \frac{pq}{I_{zz}}(I_{yy} - I_{xx}) + D$$

Using the UDE equations derived earlier for a second order system, the control input τ_z is defined as:

$$\tau_z = I_{zz} \left[\frac{pq}{I_{zz}}(I_{xx} + \nu_{\tau_y} + u_{d\tau_y}) \right]$$

Where,

$$\begin{aligned} \nu_{\tau_z} &= \dot{r}_{ref} + k_1(r_{ref} - r) + k_2(\phi_{ref} - \phi) \\ u_{d\tau_z} &= -\frac{1}{\tau}(r - \int \nu_{\tau_z} dt) \end{aligned}$$

7.2 Virtual Forces

7.2.1 Description

The idea here is to control the drone using the concept of virtual forces. The control problem can be thought of as trying to drag a point mass at (x_1, y_1, z_1) in free space to (x_2, y_2, z_2) . The drone here is assumed to be a point mass in space with a center of mass given by the entire weight of the model.

The required 3D vector (virtual force) originating from the object's center of mass at (x_0, y_0, z_0) towards the goal point is to be found. This desired vector can be found using the control laws shown previously such as UDE with extremely simple dynamics given by:

$$\begin{aligned} \ddot{x} &= \frac{F_x}{m} \\ \ddot{y} &= \frac{F_y}{m} \\ \ddot{z} &= -g + \frac{F_z}{m} \\ \mathbf{F}_{\text{virtual}} &= [F_x \quad F_y \quad F_z]^\top \end{aligned}$$

Once the virtual force, F_{virtual} , is calculated, the Z axis of the body frame of the drone, Z_b , should be aligned with it. The remaining axes body axes of the drone, X_b and Y_b can be



calculated in the inertial frame using vector algebra. A rotation matrix is constructed from these axis vectors and the desired angles of roll, pitch and yaw can be calculated using trigonometric relations obtained from the general rotation matrix in section 5.2.

7.2.2 Derivation of Virtual Forces based Controller

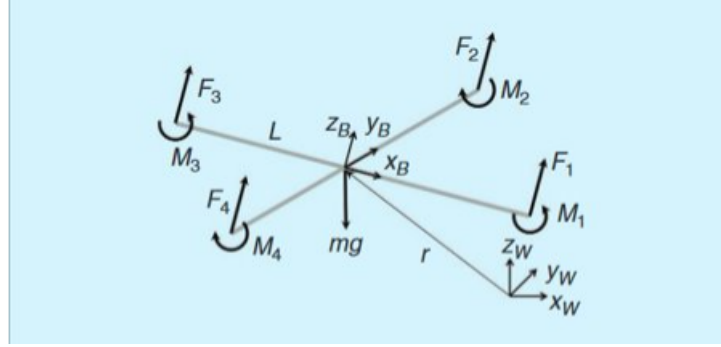


Figure 7.1: Virtual forces

$$\mathbf{Z}_b = \frac{\mathbf{F}_{\text{virtual}}}{\|\mathbf{F}_{\text{virtual}}\|}$$

$$\mathbf{X}_c = [\cos \psi_{des} \quad \sin \psi_{des}]^T$$

$$\mathbf{Y}_b = \frac{\mathbf{Z}_b \times \mathbf{X}_c}{\|\mathbf{Z}_b \times \mathbf{X}_c\|}$$

$$\mathbf{X}_b = \mathbf{Y}_b \times \mathbf{Z}_b$$

$${}^w\mathbf{R}_b = [\mathbf{X}_b \quad \mathbf{Y}_b \quad \mathbf{Z}_b]$$

$$\phi_{des} = \arctan \frac{\mathbf{R}_{32}}{\sqrt{\mathbf{R}_{31}^2 + \mathbf{R}_{33}^2}}$$

$$\theta_{des} = \arctan \frac{-\mathbf{R}_{31}}{\mathbf{R}_{33}}$$

$$\psi_{des} = \psi_{ref}$$

7.2.3 X-axis virtual force controller

Using the UDE equations derived earlier for a second order system and the dynamics for the virtual force in x , we obtain

$$F_x = m [v_x + u_{d_x}]$$

Where,

$$v_x = \ddot{x}_{ref} + k_1(\dot{x}_{ref} - \dot{x}) + k_2(x_{ref} - x)$$

$$u_{d_x} = -\frac{1}{\tau}(\dot{x} - \int \nu_x dt)$$



7.2.4 Y-axis virtual force controller

Using the UDE equations derived earlier for a second order system and the dynamics for the virtual force in y , we obtain

$$F_y = m [v_y + u_{d_y}]$$

where,

$$v_y = \ddot{y}_{ref} + k_1(\dot{y}_{ref} - \dot{y}) + k_2(y_{ref} - y)$$

$$u_{d_y} = -\frac{1}{\tau}(\dot{y} - \int \nu_y dt)$$

7.2.5 Z-axis virtual force controller

Using the UDE equations derived earlier for a second order system and the dynamics for the virtual force in z , we obtain

$$F_z = m [g + \nu_z + u_{d_z}]$$

where,

$$\nu_z = \ddot{z}_{ref} + k_1(\dot{z}_{ref} - \dot{z}) + k_2(z_{ref} - z)$$

$$u_{d_z} = -\frac{1}{\tau}(\dot{z} - \int \nu_z dt)$$

7.2.6 Controller for torque about X-axis

The dynamics for the angular velocity about the x-axis is given by:

$$\dot{p} = \frac{\tau_x}{I_{xx}} - \frac{qr}{I_{xx}}(I_{zz} - I_{yy}) + D$$

Using the UDE equations derived earlier for a second system, the control input τ_x is defined as:

$$\tau_x = I_{xx}[\frac{qr}{I_{xx}}(I_{xx} - I_{yy})] + \nu_{\tau_x} + u_{d_{\tau_x}}$$

Where,

$$\nu_{\tau_x} = \dot{p}_{ref} + k_1(p_{ref} - p) + k_2(\phi_{ref} - \phi)$$

$$u_{d_{\tau_x}} = -\frac{1}{\tau}(p - \int \nu_{d_{\tau_x}} dt)$$

7.2.7 Controller for torque about Y-axis

The dynamics for the angular velocity about the y-axis is given by:

$$\dot{q} = \frac{\tau_y}{I_{yy}} - \frac{pr}{I_{yy}}(I_{xx} - I_{zz}) + D$$

Using the UDE equations derived earlier for a second order system, the control input τ_y is defined as:

$$\tau_y = I_{yy}[\frac{pr}{I_{yy}}(I_{xx} - I_{zz})] + \nu_{\tau_y} + u_{d_{\tau_y}}$$



Where,

$$\begin{aligned}\nu_{\tau_y} &= \dot{q}_{ref} + k_1(q_{ref} - q) + k_2(\theta_{ref} - \theta) \\ u_{d_{\tau_y}} &= -\frac{1}{\tau}(q - \int \nu_{\tau_y} dt)\end{aligned}$$

7.2.8 Controller for torque about Z-axis

$$\dot{r} = \frac{\tau_z}{I_{zz}} - \frac{pq}{I_{zz}}(I_{yy} - I_{xx}) + D$$

Using the UDE equations derived earlier for a second order system, the control input τ_z is defined as:

$$\tau_z = I_{zz}[\frac{pq}{I_{zz}}(I_{xx} + \nu_{\tau_y} + u_{d_{\tau_y}})]$$

Where,

$$\begin{aligned}\nu_{\tau_z} &= \dot{q}_{ref} + k_1(q_{ref} - q) + k_2(\phi_{ref} - \phi) \\ u_{d_{\tau_z}} &= -\frac{1}{\tau}(r - \int \nu_{\tau_z} dt)\end{aligned}$$

Chapter 8

Simulink implementation and Results

The Simulink platform was chosen because of the ease with which we can model, simulate and analyze dynamical systems. MATLAB scripting within Simulink allows us to visualize our results by exporting variables from one workspace to another.

8.1 Physical Parameters of PlutoX

The drone model used reflects the commercially available Pluto X sold by DronaAviation, IIT Bombay. The physical properties are listed in SI units as follows:

Property	Value
Mass	56g
Arm length	65mm
Max RPM	30000
I_{xx}	$6.5 \times 10^{-5} \text{ kg m}^2$
I_{yy}	$6.21 \times 10^{-5} \text{ kg m}^2$
I_{zz}	$1.18 \times 10^{-4} \text{ kg m}^2$
Force coefficient k_F	$2.83 \times 10^{-8} \text{ N s}^2$
Moment coefficient k_M	$1.55 \times 10^{-10} \text{ N m s}^2$
Maximum thrust F_z	1.1172 N
Maximum torque $\tau_{x/y}$	$1.81 \times 10^{-2} \text{ N m}$
Maximum torque τ_z	$3.1 \times 10^{-3} \text{ N m}$

Table 8.1: Physical parameters of Pluto X

Two controllers for our control problem have been designed, namely a *small-angle* controller and a *virtual force* based controller. The former linearizes higher order terms arising from trigonometric functions in model whereas the latter uses the concept of virtual forces. Both structures use a form of cascaded control consisting of an outer positional controller and an inner attitude or angle controller. The position controller is responsible for calculating the required total thrust and the desired angles of roll (ϕ) and pitch (θ). These values are then passed to the attitude controller which is responsible for calculating the torque τ needed for attaining the desired angle.

The system model block contains the nonlinear dynamics of the quadrotor. It finds the second order derivative of the system state based on current inputs, passes this to two consecutive integrator blocks and provides the state information to each controller as required. Noise and disturbance can also be added to the simulation. Triggers on a particular subsystem are used to set the operating speeds of the inner and outer loop. In this form of cascaded control, the best performance is obtained when the inner loop runs at least 10 times faster than the outer loop.

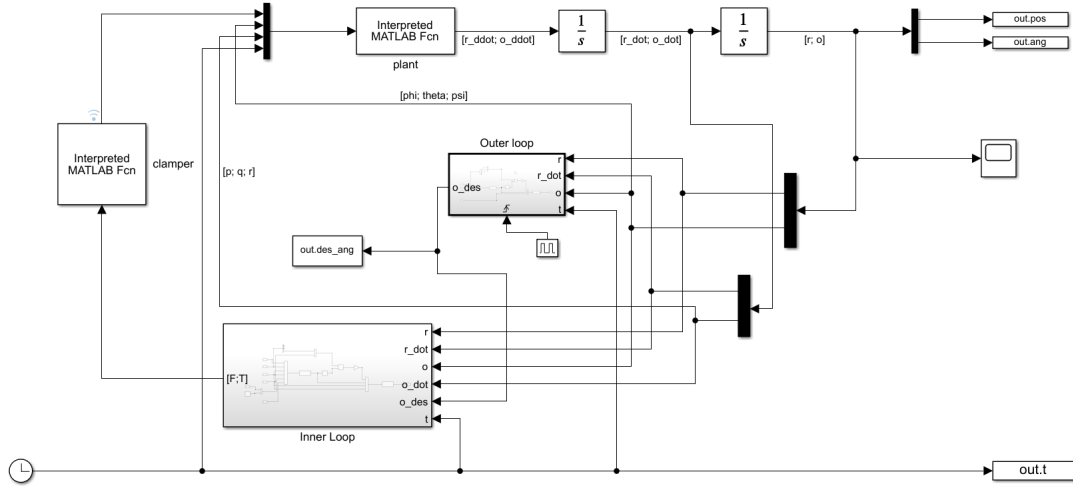


Figure 8.1: High level Simulink model

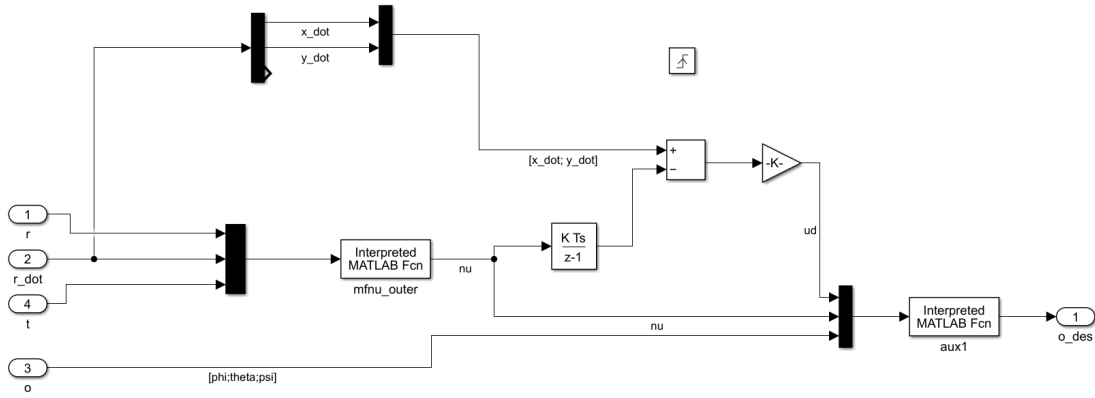


Figure 8.2: Small angle controller - Outer loop

8.2 Small angle controller

The controllers are derived by linearizing the equations of motion and motor models at an operating point that corresponds to the nominal hover state, $r = r_0$, $\theta = \phi = 0$, $\psi = \psi_0$, $\dot{r} = 0$, and $\dot{\phi} = \dot{\theta} = \dot{\psi} = 0$, where the roll and pitch angles are small. At this hover state, the nominal thrusts from the propellers must satisfy

$$F_{i,0} = \frac{mg}{4}$$

and the motor speeds are given by

$$\omega_{i,0} = \omega_h = \sqrt{\frac{mg}{4k_F}}$$

Variable	Controller	T_s	ζ	k_1	k_2	τ
x	UDE	5.0	0.702	1.6	1.3	1.25
y	UDE	5.0	0.702	1.6	1.3	1.25
z	UDE	2.0	0.8	4	6.25	1.25
ϕ	UDE	0.4	1.0	20	100	0.001
θ	UDE	0.4	1.0	20	100	0.001
ψ	UDE	0.4	1.0	20	100	0.001

Table 8.2: Parameters for small angle control

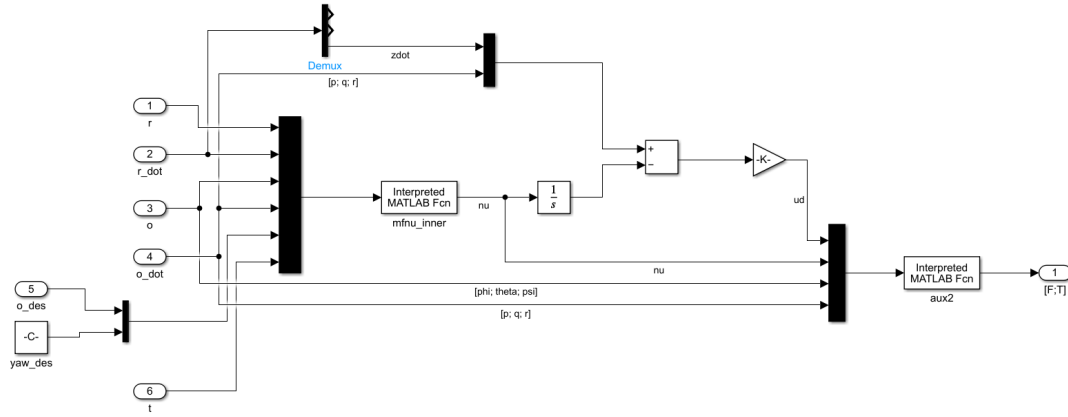


Figure 8.3: Small angle controller - Inner loop

Simulation for a step input

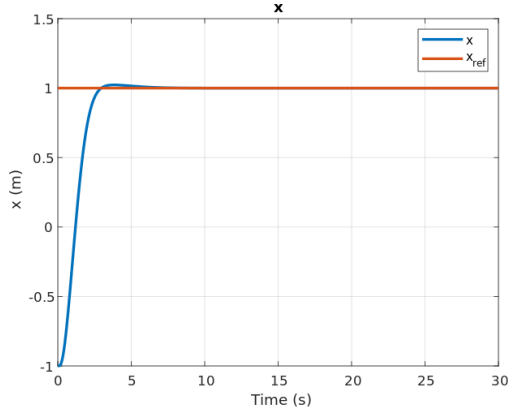
The step input is essentially a constant goal/reference point to be reached by the drone starting from an initial point.

The following simulation parameters were used:

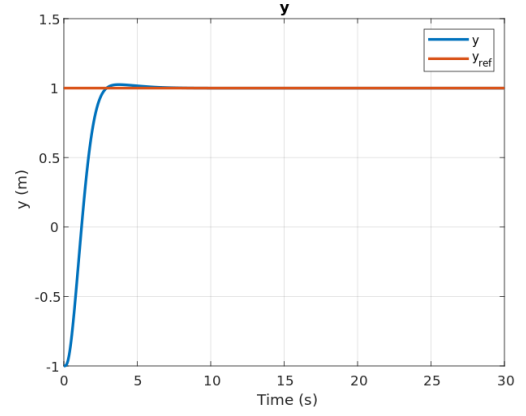
1. The simulation is run for 30 seconds with $tstep = 0.001$ seconds
2. The solver used is Runge-Kutta (`ode4`)
3. The initial position is set to $[x = -1, y = -1, z = -1]^T$
4. The initial attitude is set to $[\phi = 0, \theta = 0, \psi = 0]^T$
5. The desired goal point and yaw is $[x = 1, y = 1, z = 1, \psi = 0]^T$
6. No added *disturbance*
7. The thrust and torques are clamped to permissible values based on the motors of the PlutoX drone



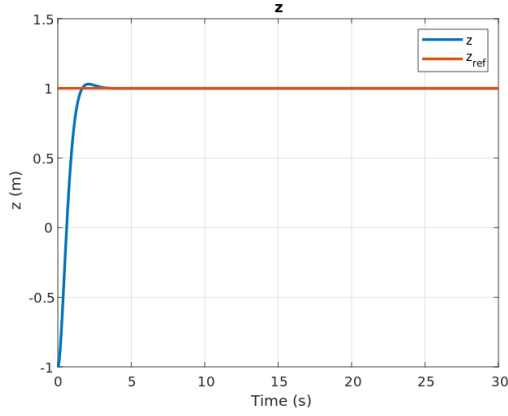
Results



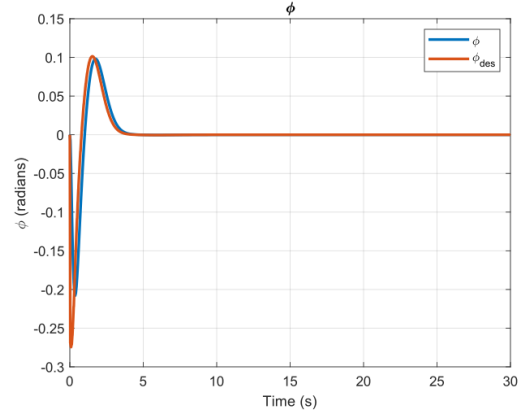
(a) Response in x for $x_{ref} = 1$



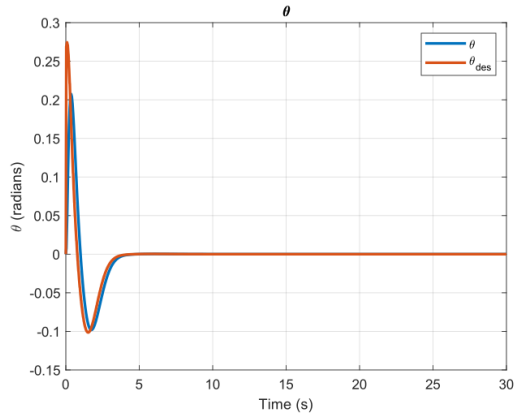
(b) Response in y for $y_{ref} = 1$



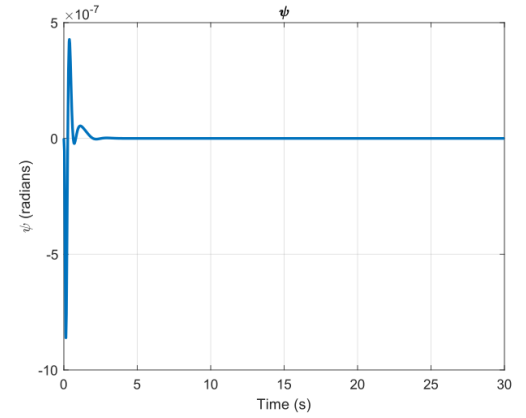
(c) Response in z for $z_{ref} = 1$



(d) Response of ϕ with ϕ_{des} decided by the outer loop



(e) Response of θ with θ_{des} decided by the outer loop



(f) Response of ψ with $\psi_{des} = 0$

Figure 8.4: Response of Small Angle Controller to Step Input



Simulation for a tracking reference with added disturbance

The desired goal point varies in time as a sinusoidal signal. The disturbance added to the plant model also varies in a similar fashion

The following simulation parameters were used:

1. The simulation is run for 30 seconds with `tstep` = 0.001 seconds
2. The solver used is Runge-Kutta (`ode4`)
3. The initial position is set to $[x = -1, y = -1, z = -1]^\top$
4. The initial attitude is set to $[\phi = 0, \theta = 0, \psi = 0]^\top$
5. The desired goal point and yaw is $[x = 1, y = 1, z = 1, \psi = 0]^\top$
6. The time varying disturbance in the plant model is

$$\vec{\mathbf{r}}_{disturbance} = mg \begin{bmatrix} \frac{\sin 2\omega t}{6} \\ \frac{\sin 2\omega t}{5} \\ \frac{\sin 2\omega t}{3} \end{bmatrix}$$

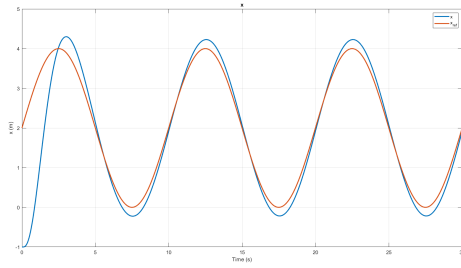
7. The tracking reference used is

$$\vec{\mathbf{r}}_{tracking} = \vec{\mathbf{r}}_{const} + \begin{bmatrix} 2 \sin \omega t \\ \sin \omega t \\ \frac{\sin \omega t}{2} \end{bmatrix}$$

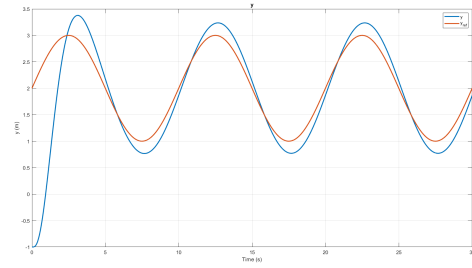
8. The thrust and torques are clamped to permissible values based on the motors of the PlutoX drone



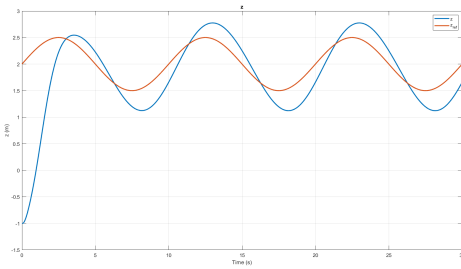
Results



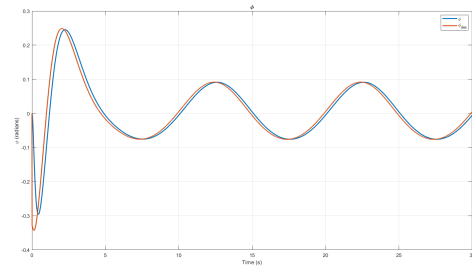
(a) Response in x with varying x_{ref}



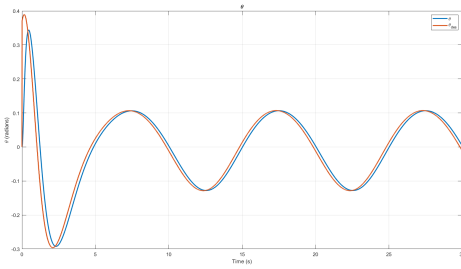
(b) Response in y with varying y_{ref}



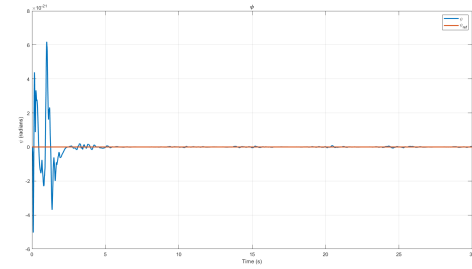
(c) Response in z with varying z_{ref}



(d) Response of ϕ with ϕ_{des} decided by the outer loop



(e) Response of θ with θ_{des} decided by the outer loop



(f) Response of ψ with $\psi_{des} = 0$

Figure 8.5: Response of Small Angle Controller to sinusoidal input

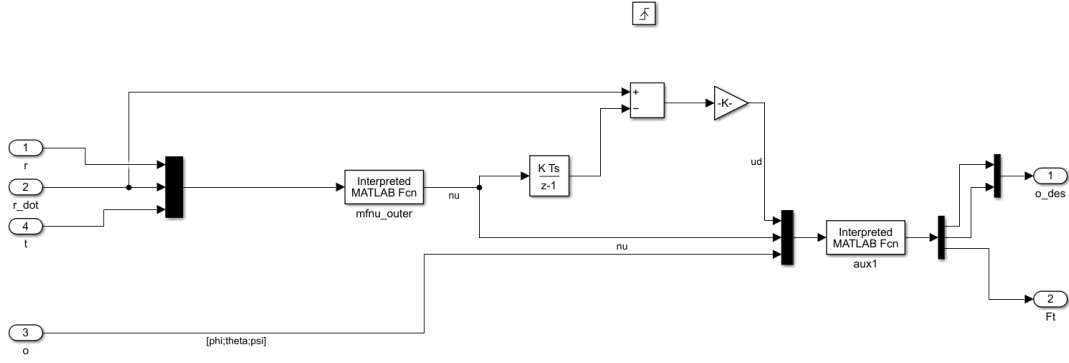


Figure 8.6: Virtual force controller - Outer loop

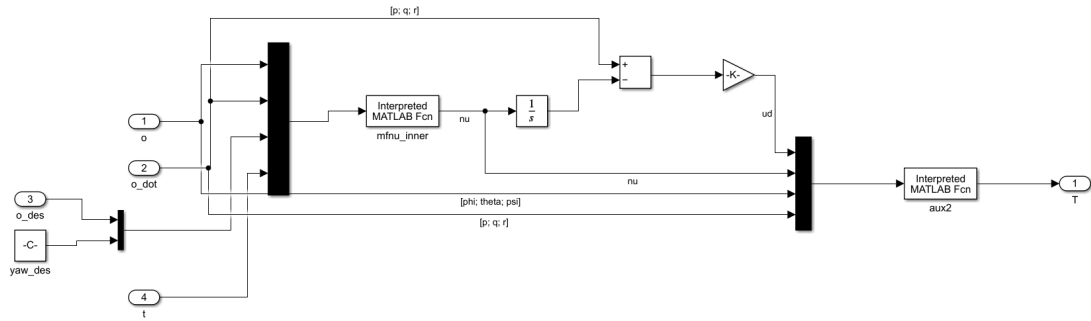


Figure 8.7: Virtual force controller - Inner loop

8.3 Virtual force controller

The theory describing virtual forces and controlling them is described in 7.2. The parameters in Table 8.3 were used for the controllers and simulation was performed for a step input and a sinusoidal input.

Variable	Controller	T_s	ζ	k_1	k_2	τ
x	UDE	5.0	0.702	1.6	1.3	0.5
y	UDE	5.0	0.702	1.6	1.3	0.5
z	UDE	5.0	0.702	1.6	1.3	0.5
ϕ	UDE	0.4	1.0	20	100	0.1
θ	UDE	0.4	1.0	20	100	0.1
ψ	UDE	0.4	1.0	20	100	0.1

Table 8.3: Parameters for virtual force controller

Simulation for a step input

The step input is essentially a constant goal/reference point to be reached by the drone starting from an initial point.

The following simulation parameters were used:

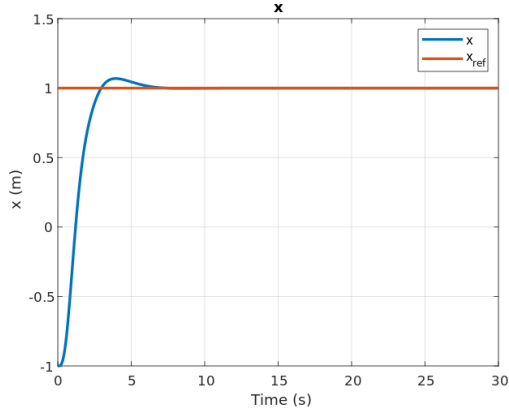
1. The simulation is run for 30 seconds with $t_{step} = 0.001$ seconds
2. The solver used is Runge-Kutta (`ode4`)
3. The initial position is set to $[x = -1, y = -1, z = -1]^T$
4. The initial attitude is set to $[\phi = 0, \theta = 0, \psi = 0]^T$



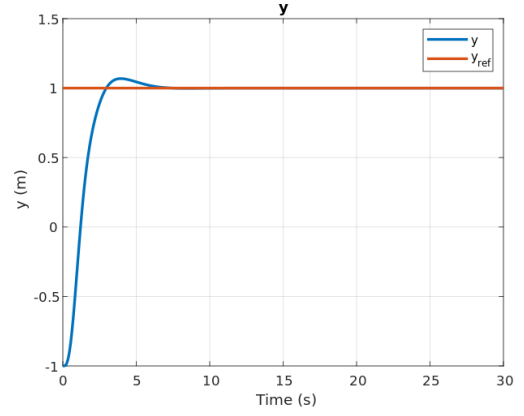
5. The desired goal point and yaw is $[x = 1, y = 1, z = 1, \psi = 0]^\top$
6. No added *disturbance*
7. The thrust and torques are clamped to permissible values based on the motors of the PlutoX drone



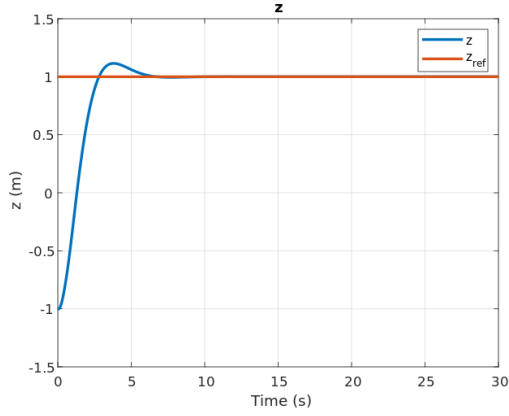
Results



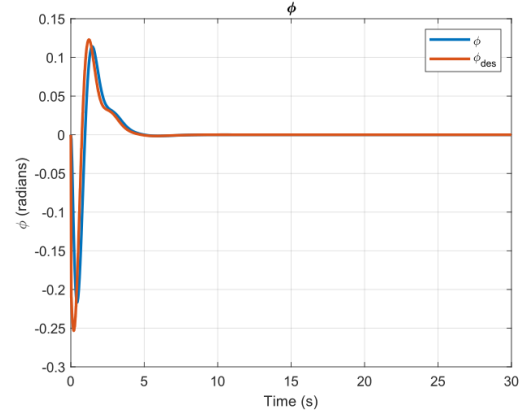
(a) Response in x for $x_{ref} = 1$



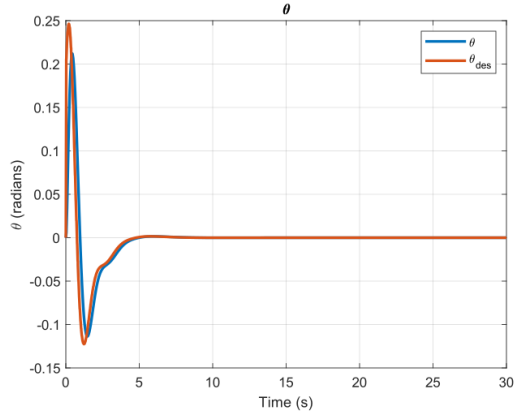
(b) Response in y for $y_{ref} = 1$



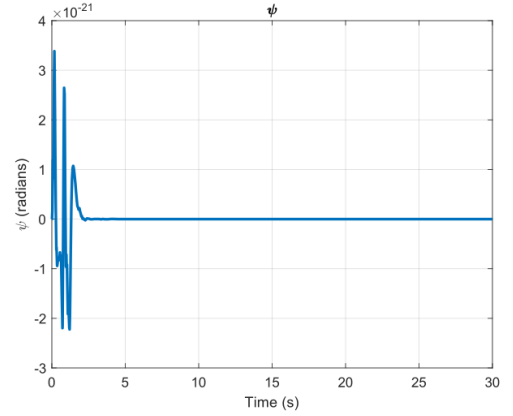
(c) Response in z for $z_{ref} = 1$



(d) Response of ϕ with ϕ_{des} decided by the outer loop



(e) Response of θ with θ_{des} decided by the outer loop



(f) Response of ψ for $\psi_{ref} = 0$

Figure 8.8: Response of Virtual Force Controller to Step input



Simulation for a tracking reference with added disturbance

The desired goal point varies in time as a sinusoidal. The disturbance added to the plant model also varies in a similar fashion

The following simulation parameters were used:

1. The simulation is run for 30 seconds with $\text{tstep} = 0.001$ seconds
2. The solver used is Runge-Kutta (`ode4`)
3. The initial position is set to $[x = -1, y = -1, z = -1]^\top$
4. The initial attitude is set to $[\phi = 0, \theta = 0, \psi = 0]^\top$
5. The desired goal point $\vec{\mathbf{r}}_{const}$ and yaw is $[x = 2, y = 2, z = 2, \psi = \pi/4]^\top$
6. The time varying disturbance in the plant model is

$$\vec{\mathbf{r}}_{disturbance} = mg \begin{bmatrix} \frac{\sin 2\omega t}{6} \\ \frac{\sin 2\omega t}{5} \\ \frac{\sin 2\omega t}{3} \end{bmatrix}$$

7. The tracking reference used is

$$\vec{\mathbf{r}}_{tracking} = \vec{\mathbf{r}}_{const} + \begin{bmatrix} 2 \sin \omega t \\ \sin \omega t \\ \frac{\sin \omega t}{2} \end{bmatrix}$$

8. The thrust and torques are clamped to permissible values based on the motors of the PlutoX drone

Results

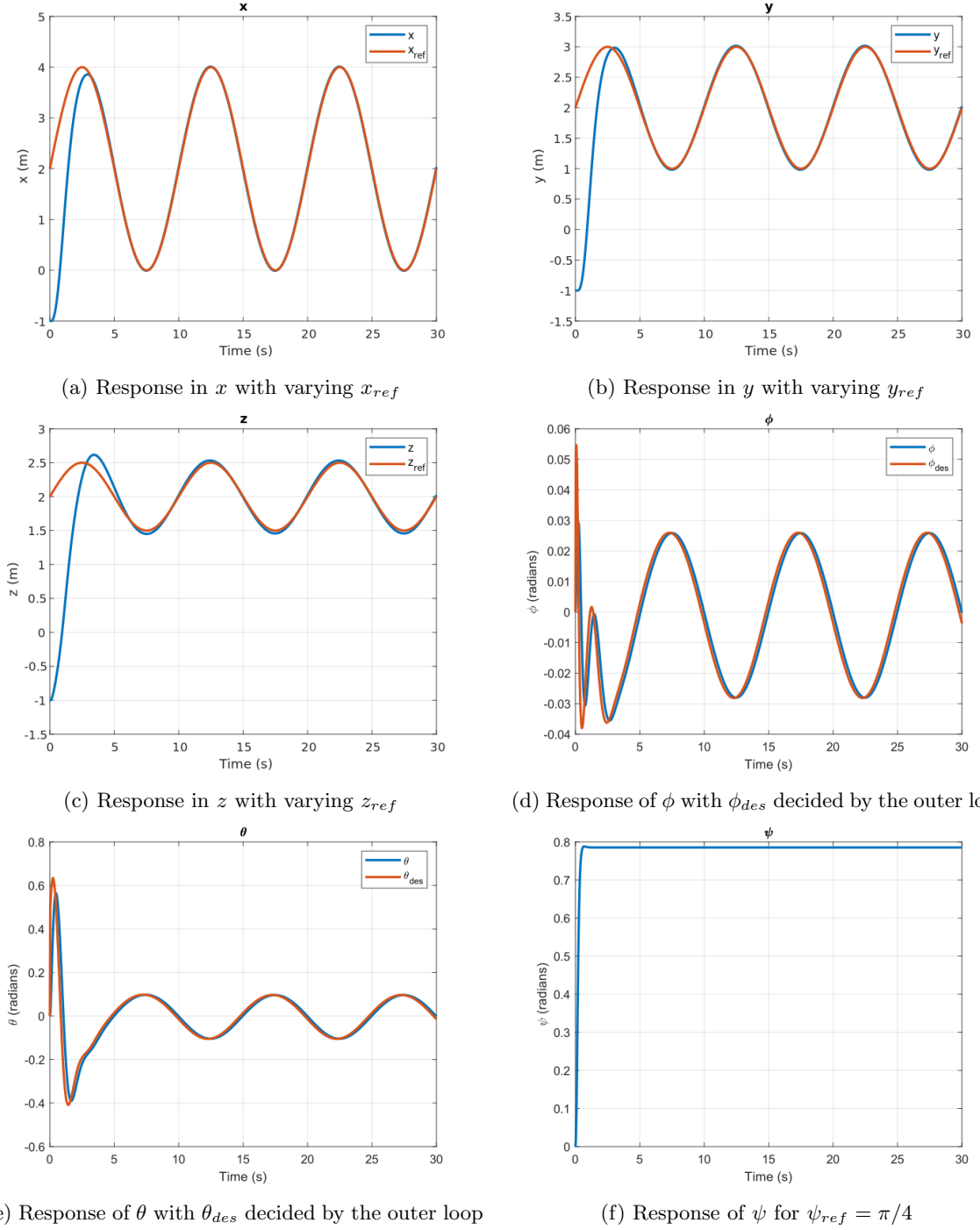


Figure 8.9: Response of Virtual Force Controller to sinusoidal input

Chapter 9

Hardware implementation

9.1 Whycon marker

WhyCon is a visual localisation system. It can find the position and orientation of the drone with the marker placed on with an accuracy of $\pm 2\text{cm}$. The marker needs preliminary calibration before it can be used reliably.

Brief working: The WhyCon marker based localisation system is built on the easy identification of a black and white roundel with known dimensions. In the first stage, the algorithm searches for contiguous regions of dark pixels. If the test is passed, the procedure begins looking for another continuous area of bright pixels around the segment's centroid, and checks if the number of pixels matches the value determined from the bounding box dimensions. The pattern's 3D position is determined based on its known dimensions, camera characteristics, and the user-defined coordinate system once the segments identified pass this test.

Once the camera is calibrated, the marker can be used to estimate the linear position, linear velocity and angular position of the drone. The video feed is sent by the camera over to the computer where the appropriate transformations are applied to the bounding boxes and centroid. The resulting information about the estimated position and it's rate of change is published to a topic called `/whycon/poses`.

9.2 ROS - Robot Operating System

ROS is an adaptable framework for developing robot software. It provides developers with the libraries and tools to create robot applications and support robust collaborative robotics software development from scratch. ROS is not an operating system in the traditional sense but rather a meta operating system where each process can be thought of as a *node*. The tools and libraries bootstrapped aim to ease the task of creating complex robot behavior across a wide variety of robotic platforms and products.

ROS is distributed as well as modular so that the users can decide how much of the framework they want to use in their project. Some of the philosophical goals of ROS are:

- Peer-to-peer
- Multi-lingual
- Tool based
- Lightweight
- Open Source and free

Design and concepts of ROS:



1. **Computation Graph Model** - The processes running in ROS are considered as *nodes* in a graph structure. Each node is connected by edges called *topics*. The nodes communicate with each other using these topics. The *ROS master* enables node to node communication by registering all the nodes associated to itself.
2. **Nodes** - A single process running in a graph is represented by a node. Each node has a unique name and it will be registered with the ROS master. Other nodes with same name must be registered under different namespaces. Nodes are at the core of ROS programming. All types of communication/requests are enabled through nodes as almost all of the ROS client source is in form of a node or action.
3. **Topics** - Topics are channels used the by nodes for communication. To send a message to a topic, the sender node must publish to the required topic while the receiver node must subscribe to it. Hence it follows the *Publisher-Subscriber* model. The Pub-Sub is anonymous i.e. other than the sending and the receiving nodes all the other nodes will be unaware of the communication taking place.
4. **Services** - They are an actions taken by a node that produces a single result. Services are frequently used for actions that have a consistent behaviour from start to finish. These services are frequently advertised by nodes, and nodes can also invoke services from other nodes.
5. **Parameter server** - A shared database among nodes that provides for shared access to static or configurable data.

ROS also provides us with certain tools which makes working a lot easier.

Tools improve the core functionality of ROS. These tools assist ROS developers in seeing, recording, and navigating ROS package topologies. The availability of these tools enhances the system's capabilities by making it easier to solve typical development difficulties. These tools are distributed as packages that are included by default in most ROS installs.

- (a) **rviz** - **rviz** is a 3-D visualizer for visualising robots, their working environs, and onboard sensor data. It's a very versatile tool with a variety of visuals and plugins to choose from.
- (b) **roscap** - **roscap** is a cmd line tool which is used to store and replay ROS message data. This logs ROS messages by listening to topics and capturing messages using a file format called bags. It's almost as good as data being produced by original nodes i.e by ROS Computation graph.
- (c) **catkin** - **catkin** is the ROS build system. **catkin** is comparable to CMake in that it is open sourced cross-platform, and is independent of the language used.
- (d) **roscpp** - **roscpp** is a package which provides with a set of tools for augmenting the function of ***sh** shells. The suite of tools include **rosls**, **roscd** and **roscp** which have similar functionality as **ls**, **cd** and **cp** respectively. It also includes **roscat** which runs ROS executables in packages.
- (e) **roslaunch** - **roslaunch** is a tool used for launching nodes both remotely and locally and also to set parameters on the parameter server. The configuration files are in YAML whereas launch files are written in XML and can be used to combine multiple commands into a single command. **roslaunch** can start nodes on specified machines and also restart processes that stop working amidst executions.

Rospy

rospy is a Python client library for ROS. Python programmers interface with ROS topics, Services and Parameters using the **rospy** client API. **rospy** is designed in a way that favours implementation speed over runtime performance for prototyping algorithms quickly within ROS. Many ROS tools like **rosservice** and **rostopic** are written on top of **rospy**. It's also easier to use this for code initialization and configuration.



9.3 Python-based controllers

9.3.1 PID controller with rospy

The code can be found [here](#) in this repository written by us on GitHub.

- We import libraries `rospy` and `numpy`. The latter is a library for numerical computation. Support for multidimensional arrays will be added, as well as an extensive set of high-level mathematical functions to manipulate these arrays. The former is used as described before.
- A class called `DroneFly` is implemented for controlling the drone with a simple PID loop and provides utilities for arming/disarming the drone.
- The class also includes a callback which subscribes to `whycon/poses` for positional data.
- The control input calculated by PID in function `calcPID` is linearly mapped to an `rc` range and published on the `/drone_command` topic. The Pluto node subscribed to this topic translates the `rc` values into PWM values and sends it over WiFi to the drone.
- The constructor takes a 3x1 vector as an argument which corresponds to the desired x, y, and z coordinates. The position vector is by default initialized to (0, 0, 0).
- `rospy.sleep(0.1)` is used to account for the delay of communication (over hotspot) between the drone and the host. This is used to avoid queueing of instructions while the system is performing tasks on the previous instruction. It also gives time for arming and disarming the motors of the drone.
- `isThere()` is used for calculating the error between the desired and actual position of the drone i.e. the absolute difference between the two values and testing whether it is greater than the threshold specified.
- `calcPID()` calculates the error, uses a simple PID and finds the updated control input to drive the output of the drone to the desired goal point.
- `clamp()` is used to restrict the control input values between the specified upper and lower limits.
- `position-hold()` wraps the logic of `calcPID()` around an indefinite ROS while loop running at a specified rate. The control input is published to `/drone_command` in each iteration. The current values of position are logged to the console as well.

9.3.2 UDE controller with rospy

The code can be found [here](#) in this repository written by us on GitHub.

Four scripts containing classes for each component are of interest here:

- `main.py`
- `outer_loop_controller.py`
- `inner_loop_controller.py`
- `ude_base_controller.py`

The structure of this controller is more complicated given that we have to run two loops at differing speeds. The ROS graph can be summarized as follows:

The node `pluto_ude` is the main commander for the thrust and torques which publishes `rc` values to `/drone_command`. It receives the desired control inputs from `outer_loop_controller` and `inner_loop_controller` nodes. These nodes in turn subscribe to topics which contain sensor data onboard the drone such as `/whycon/poses` and `/Pluto_sensor_data`. The `inner_loop_controller` receives the desired angles for calculating the torques from `/commander/desired_attitude` which is published on by `outer_loop_controller`.



1. `main.py` - The plumbing for this script is similar to the structure used in section 7.2. It includes clamping to restrict the control input in a valid range as well as lambda functions which linearly map u to `rc` values for PWM onboard the drone. Subscribers include `/commander/thrust` and `/commander/torques` whose values are mapped and published onto `/drone_command`.
2. `outer_loop_controller.py` - The class includes three controllers of the UDE base class, one for displacement in each axis. The callbacks receive linear positional data from the WhyCon and linear acceleration from the onboard IMU. The velocity is estimated by using a discrete integrator at each step. The computed values of thrust and desired attitude are published on topics `/commander/thrust` and `/commander/desired_attitude` respectively. Other details mirror the outer loop block used from the Simulink model in section xx.
3. `inner_loop_controller.py` - The class includes three controllers of the UDE base class, one for the angle about each of the X, Y and Z axes i.e. ϕ , θ and ψ . The class also subscribes to `/commander/desired_attitude` and receives desired values for the attitude via a callback. Sensor data from the gyroscope such as current attitude and angular velocity are similarly obtained from the `/Pluto_sensor_data` topic. The needed torques are published to `/commander/torques`.
4. `ude_base_controller.py` - Base class for the UDE controller which includes lower level functions used to compute terms in the expressions for u_a , ν , u_d and \mathbf{u} .

9.3.3 Hardware Implementation of UDE controller through ROS interface

Hardware implementation through the ROS interface was unsuccessful due to failure of the WiFi communication module - ESP12F. As a result, the controller was written in C++ using the Cygnus IDE and flashed onto the drone as explained in section 9.3.4

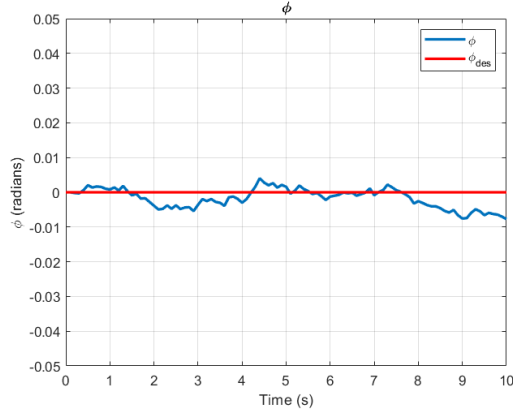
9.3.4 UDE controller with Cygnus IDE

The Cygnus IDE is an IDE provided by Drona Aviation to allow for tinkering with the PlutoX drone. This IDE was used to flash a custom controller onto the microcontroller (STM32) present on-board the drone due to issues present in the communication module (ESP12f).

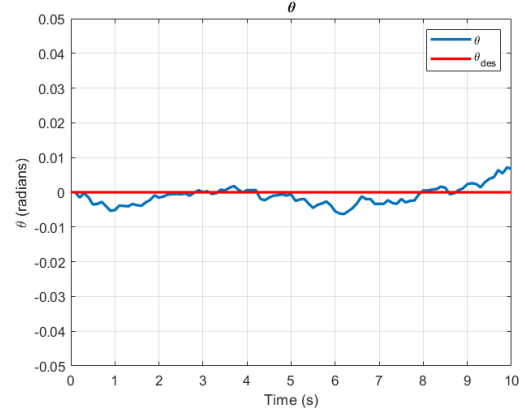
The structure of the UDE controller mirrors that used in the `rospy` implementation, with the only difference being that the entire logic is imperative without the use of additional callbacks and a Pub-Sub architecture to fetch sensor and positional data as compared to the ROS approach.

The drawback with this approach is that the WhyCon data is unavailable to us for use within the script. Due to this, the position control was not implemented on hardware. Attitude control based on UDE was successfully implemented to stabilize the drone. The position has to be estimated using a combination of Kalman filters and integrators. The noise present in raw sensor data has to be attenuated before it can be integrated to give the position.

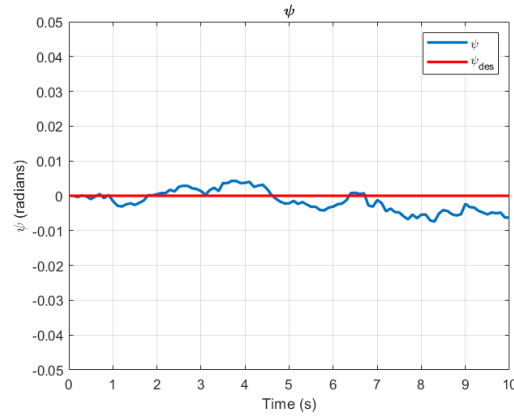
Results of Attitude Control on Hardware



(a) Variation of ϕ on hardware with $\phi_{des} = 0$



(b) Variation of θ on hardware with $\theta_{des} = 0$



(c) Variation of ψ on hardware with $\psi_{des} = 0$

Figure 9.1: Attitude Control based on UDE on PlutoX

From the above graphs, it can be seen that the values of roll, pitch and yaw are oscillating with small error about the desired value

Chapter 10

Analysis and Conclusion

10.1 Analysis

The performance benefits of the UDE control approach augmented with the concept of virtual forces can be clearly seen. Steady state error, ability to reject disturbances and perform under a constant step input is compared as:

Metric	Tuned PID	UDE with virtual forces
Rise time T_r in x	1.12	1.47
Rise time T_r in y	1.13	1.47
Rise time T_r in z	∞^1	1.41
Settling time T_s in x	2.76	3.33
Settling time T_s in y	2.77	3.34
Settling time T_s in z	∞^2	3.31
Maximum overshoot in x	12.01	6.88
Maximum overshoot in y	12.01	6.83
Maximum overshoot in z	84.65	11.52

Table 10.1: Comparative study

¹ - The final value is never reached in z so the rise time can be assumed to be infinite

² - The final value is never reached in z so the settling time can be assumed to be infinite

The simulation run with the same parameters as in 8.3 without the use of a UDE controller but a simple PID scheme gives us the following results:

- It can be inferred that the UDE controller augmented with virtual forces clearly outperforms a highly tuned PID in terms of maximum overshoot at a slight cost of decrease in rise and settling times.
- The PID controller never manages accurately control the altitude of the drone. The final value of the height never converges within 98% of the reference value.
- The PID controller also fails when the input is changed from step to a varying reference as can be seen in 10.1. Comparing this to the results seen in 8.9, it is obvious that the UDE approach is clearly superior in terms of overall performance and its ability to track a sinusoidal reference.

10.2 Conclusion

In this project, different control strategies like PID, MPC, SMC etc, were studied. Out of these, the UDE controller was found to be optimal in terms of ease of implementation and computational complexity. Two controllers based on UDE were designed for a quadrotor, namely *Small Angle Controller* and *Virtual Force Controller*. These controllers were simulated using MATLAB/SIMULINK

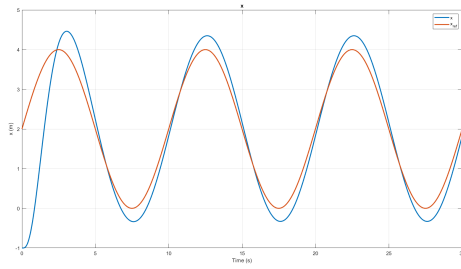
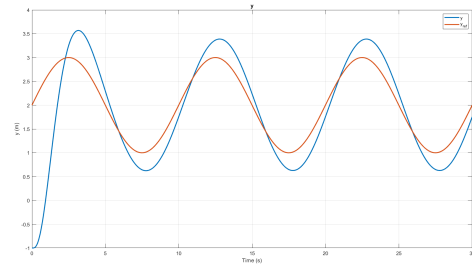
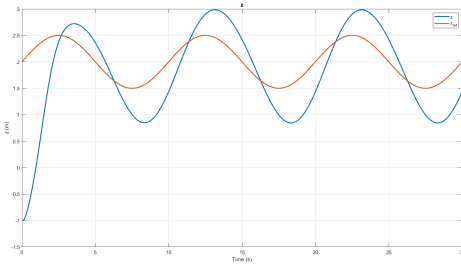
(a) Response in x with varying x_{ref} (b) Response in y with varying y_{ref} (c) Response in z with varying z_{ref}

Figure 10.1: Response of PID to sinusoidal input

and their results were validated. The virtual force controller was not successfully implemented on the PlutoX drone through the ROS interface due to hardware failure. A UDE based attitude controller was implemented using the Cygnus IDE and flashed onto the drone which successfully stabilized the drone with a minimal error of ± 0.5 degrees.

References

- [1] A. L'afflitto, R. B. Anderson, and K. Mohammadi, "An introduction to nonlinear robust control for unmanned quadrotor aircraft: how to design control algorithms for quadrotors using sliding mode control and adaptive control techniques [focus on education]," *IEEE Control Systems Magazine*, vol. 38, no. 3, pp. 102–121, 2018.
- [2] E. Altug, J. P. Ostrowski, and R. Mahony, "Control of a quadrotor helicopter using visual feedback," in *Proceedings 2002 IEEE international conference on robotics and automation (Cat. No. 02CH37292)*, vol. 1. IEEE, 2002, pp. 72–77.
- [3] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "The grasp multiple micro-uav testbed," *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 56–65, 2010.
- [4] F. Santoso, M. A. Garratt, S. G. Anavatti, and I. Petersen, "Robust hybrid nonlinear control systems for the dynamics of a quadcopter drone," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 8, pp. 3059–3071, 2018.
- [5] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," *IEEE Transactions on control systems technology*, vol. 18, no. 2, pp. 267–278, 2009.
- [6] D. D. Dhadekar, P. D. Sanghani, K. Mangrulkar, and S. Talole, "Robust control of quadrotor using uncertainty and disturbance estimation," *Journal of Intelligent & Robotic Systems*, vol. 101, no. 3, pp. 1–21, 2021.
- [7] P. Karthik, K. Kumar, V. Fernandes, and K. Arya, "Reinforcement learning for altitude hold and path planning in a quadcopter," in *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*. IEEE, 2020, pp. 463–467.
- [8] M. Kumar, P. Sharma, and P. Kumar, "Trajectory planning of unmanned aerial vehicle using invasive weed optimization," in *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*. IEEE, 2020, pp. 467–472.
- [9] S. Bouabdallah and R. Siegwart, "Full control of a quadrotor," in *2007 IEEE/RSJ international conference on intelligent robots and systems*. Ieee, 2007, pp. 153–158.
- [10] X. Zhou, C. Yang, and T. Cai, "A model reference adaptive control/pid compound scheme on disturbance rejection for an aerial inertially stabilized platform," *Journal of Sensors*, vol. 2016, 2016.
- [11] E. Kuantama, T. Vesselenyi, S. Dzitac, and R. Tarca, "Pid and fuzzy-pid control model for quadcopter attitude with disturbance parameter," *International journal of computers communications & control*, vol. 12, no. 4, pp. 519–532, 2017.
- [12] M. K. Shaik and J. F. Whidborne, "Robust sliding mode control of a quadrotor," in *2016 UKACC 11th International Conference on Control (CONTROL)*. IEEE, 2016, pp. 1–6.
- [13] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.
- [14] S. Bouabdallah and R. Siegwart, "Backstepping and sliding-mode techniques applied to an indoor micro quadrotor," in *Proceedings of the 2005 IEEE international conference on robotics and automation*. IEEE, 2005, pp. 2247–2252.

Appendix A

MATLAB scripts

A.1 Small Angle controller

A.1.1 init.m

```
1 clc; clear; close all;
2
3 %%%%%% Global Variables %%%%%%
4 global params
5 global r_init o_init
6 global k1_outer k2_outer tau_outer
7 global k1_inner k2_inner tau_inner
8 global yaw_des
9 global A w
10
11 global tstep tstop simout
12
13 %%%%%% Simulation Parameters %%%%%%
14 tstep = 0.001;
15 tstop = 30;
16
17 %%%%%% Plant Parameters %%%%%%
18 params.mass = 56e-3; %kg
19 params.gravity = 9.81; %m/s^2
20 params.length = 65e-3; %m
21 params.I = [6.5e-5,0,0;
22             0,6.21e-5,0;
23             0,0,1.18e-4]; % Kg m^2
24 params.force_coeff = 2.83e-8;
25 params.moment_coeff = 1.55e-10;
26 params.max_rpm = 30000;
27
28
29 %%%%%% Controller Gains %%%%%%
30 %% Ts_inner_z = 2s Ts_inner_att = 0.2s or 1s; Ts_outer = 5 or 10s; zeta_outer =
    0.8;
31 %% zeta_inner = 1 or 0.8 or 0.707
32 k1_outer = 1.6*eye(2); k2_outer = 1.3*eye(2); tau_outer = 1.25;
33 % k1_outer = 4*eye(2); k2_outer = 6.25*eye(2); tau_outer = 1.25;
34 k1_inner = [4, 0, 0, 0;
35             0, 20, 0, 0;
36             0, 0, 20, 0;
37             0, 0, 0, 20];
38 k2_inner = [6.25, 0, 0, 0;
39             0, 100, 0, 0;
40             0, 0, 100, 0;
41             0, 0, 0, 100];
42 tau_inner = 0.001;
43
44 %%%%%% Initial COnditions %%%%%%
45 % 1st col - Position; 2nd col - Velocity
46 r_init = [-1,0;
47           -1,0;
```




```

48     -1,0];
49     o_init = [0,0;
50              0,0;
51              0,0];
52     yaw_des = 0;
53
54     %%%%% Tracking values %%%%%
55     A = 1; w = 0.2*pi;
56
57     simout = sim('UDE_small_angle.mdl');
58     graphResults();

```

A.1.2 mfnu_outer.m

```

1  function y = mfnu_outer(u)
2      global k1_outer k2_outer
3      global A w
4
5      y = zeros(2,1);
6      r = u(1:2);
7      r_dot = u(4:5);
8      t = u(7);
9      %{
10     r_ref = [1;1];
11     r_ref_dot = [0;0];
12     r_ref_ddot = [0;0];
13     %}
14
15     r_ref = [2*A*sin(w*t); A*sin(w*t)];
16     r_ref_dot = [2*A*w*cos(w*t); A*w*cos(w*t)];
17     r_ref_ddot = [-2*A*w*w*sin(w*t); -A*w*w*sin(w*t)];
18
19     re = r - r_ref;
20     re_dot = r_dot - r_ref_dot;
21
22     y = r_ref_ddot - k1_outer*re_dot - k2_outer*re;
23
24 end

```

A.1.3 aux1.m

```

1  function y = aux1(u)
2      global params
3      g = params.gravity;
4
5      y = zeros(2,1);
6      clamp = @(x, lb, ub) max(min(x, ub), lb);
7
8      ud = u(1:2); nu = u(3:4);
9      phi = u(5); theta = u(6); psi = u(7);
10
11     ua = zeros(2,1);
12     gains = eye(2);
13
14     %x
15     ua(1) = -g*sin(phi)*sin(psi);
16     gains(1,1) = 1/(g*cos(psi));
17
18     %y
19     ua(2) = -g*sin(psi)*sin(theta);
20     gains(2,2) = -1/(g * cos(psi) * cos(theta));
21
22     y = gains*(ua + nu + ud);
23

```



```

24     y(1) = clamp(y(1), -pi/4, pi/4);
25     y(2) = clamp(y(2), -pi/4, pi/4);
26
27     y = flip(y);
28
29 end

```

A.1.4 mfnu_inner.m

```

1 function y = mfnu_inner(u)
2     global k1_inner k2_inner
3     global A w
4
5     y = zeros(4,1);
6     r = u(3);
7     r_dot = u(6);
8     o = u(7:9); o_dot = u(10:12); o_des = u(13:15); t = u(16);
9
10    ref = [2*A*sin(w*t); o_des];
11    ref_dot = [2*A*w*cos(w*t); 0;0;0];
12    ref_ddot = [-2*A*w*w*sin(w*t); 0;0;0];
13
14    re = [r; o] - ref;
15    re_dot = [r_dot; o_dot] - ref_dot;
16
17    y = ref_ddot - k1_inner*re_dot - k2_inner*re;
18 end

```

A.1.5 aux2.m

```

1 function y = aux2(u)
2     global params
3     I = params.I; m = params.mass; g = params.gravity;
4     Ixx = I(1,1); Iyy = I(2,2); Izz = I(3,3);
5
6     y = zeros(4,1);
7
8     ud = u(1:4); nu = u(5:8);
9     phi = u(9); theta = u(10); psi = u(11);
10    p = u(12); q = u(13); r = u(14);
11
12    ua = zeros(4,1);
13    gains = eye(4);
14
15    %z
16    ua(1) = g;
17    gains(1,1) = m;
18
19    %phi
20    ua(2) = q*r*(Izz - Iyy) / Ixx;
21    gains(2,2) = Ixx;
22
23    %theta
24    ua(3) = p*r*(Ixx - Izz) / Iyy;
25    gains(3,3) = Iyy;
26
27    %psi
28    ua(4) = p*q*(Iyy - Ixx) / Izz;
29    gains(4,4) = Izz;
30
31    y = gains*(ua + nu + ud);
32
33 end

```



A.2 Virtual force controller

A.2.1 init.m

```

1  clc; clear; close all;
2
3  %%%%%%%%% Global Variables %%%%%%%%%
4  global params
5  global r_init o_init
6  global k1_outer k2_outer tau_outer
7  global k1_inner k2_inner tau_inner
8  global yaw_des
9  global A w
10
11 global tstep tstop simout
12
13 %%%%%%%%% Simulation Parameters %%%%%%%%%
14 tstep = 0.001;
15 tstop = 30;
16
17 %%%%%%%%% Plant Parameters %%%%%%%%%
18 params.mass = 56e-3; %kg
19 params.gravity = 9.81; %m/s^2
20 params.length = 65e-3; %m
21 params.I = [6.5e-5,0,0;
22             0,6.21e-5,0;
23             0,0,1.18e-4]; % Kgm^2
24 params.force_coeff = 2.83e-8;
25 params.moment_coeff = 1.55e-10;
26 params.max_rpm = 30000;
27
28
29 %%%%%%%%% Controller Gains %%%%%%%%%
30 %% Ts_inner_z = 2s Ts_inner_att = 0.2s or 1s; Ts_outer = 5 or 10s; zeta_outer =
    0.8;
31 %% zeta_inner = 1 or 0.8 or 0.707
32
33 k1_outer = 1.6*eye(3); k2_outer = 1.3*eye(3); tau_outer = 0.5;
34 k1_inner = 20*eye(3); k2_inner = 100*eye(3); tau_inner = 0.1;
35
36 %%%%%%%%% Initial COnditions %%%%%%%%%
37 % 1st col - Position; 2nd col - Velocity
38 r_init = [-1,0;
39           -1,0;
40           -1,0];
41 o_init = [0,0;
42           0,0;
43           0,0];
44 yaw_des = 0;
45
46 %%%%%%%%% Tracking values %%%%%%%%%
47 A = 1; w = 0.2*pi;
48
49 simout = sim('ude_virtual_force.mdl');
50 graphResults();

```

A.2.2 mfnu_outer.m

```

1  function y = mfnu_outer(u)
2      global k1_outer k2_outer
3      global A w
4

```



```

5     y = zeros(3,1);
6     r = u(1:3);
7     r_dot = u(4:6);
8     t = u(7);
9
10    r_ref = [1;1;1];
11    r_ref_dot = [0;0;0];
12    r_ref_ddot = [0;0;0];
13
14    %{
15    r_ref = [2*A*sin(w*t); A*sin(w*t)];
16    r_ref_dot = [2*A*w*cos(w*t); A*w*cos(w*t)];
17    r_ref_ddot = [-2*A*w*w*sin(w*t); -A*w*w*w*sin(w*t)];
18    %}
19
20    re = r - r_ref;
21    re_dot = r_dot - r_ref_dot;
22
23    y = r_ref_ddot - k1_outer*re_dot - k2_outer*re;
24
25 end

```

A.2.3 aux1.m

```

1 function y = aux1(u)
2     global params yaw_des
3     g = params.gravity;
4     m = params.mass;
5
6     y = zeros(3,1);
7     clamp = @(x, lb, ub) max(min(x, ub), lb);
8
9     ud = u(1:3); nu = u(4:6);
10    phi = u(7); theta = u(8); psi = u(9);
11
12    ua = zeros(3,1);
13    gains = eye(3);
14
15    %x
16    ua(1) = 0;
17    gains(1,1) = m;
18
19    %y
20    ua(2) = 0;
21    gains(2,2) = m;
22
23    %z
24    ua(3) = g;
25    gains(3,3) = m;
26
27    f_virtual = gains*(ua + nu + ud);
28    Zb = f_virtual/norm(f_virtual);
29
30    Xc = [cos(yaw_des); sin(yaw_des); 0];
31    Yb = cross(Zb,Xc)/norm(cross(Zb,Xc));
32    Xb = cross(Yb,Zb);
33
34    R = [Xb, Yb, Zb];
35    phi_des = atan2(R(3,2), sqrt(R(3,1)^2 + R(3,3)^2));
36    theta_des = atan2(-R(3,1), R(3,3));
37
38    phi_des = clamp(phi_des, -pi/4, pi/4);
39    theta_des = clamp(theta_des, -pi/4, pi/4);
40
41    y = [phi_des; theta_des; f_virtual(3)];
42
43 end

```



A.2.4 mfnu_inner.m

```

1 function y = mfnu_inner(u)
2     global k1_inner k2_inner
3     global A w
4
5     y = zeros(3,1);
6     o = u(1:3); o_dot = u(4:6); o_des = u(7:9); t = u(10);
7
8     ref = o_des;
9     ref_dot = [0;0;0];
10    ref_ddot = [0;0;0];
11
12    re = o - ref;
13    re_dot = o_dot - ref_dot;
14
15    y = ref_ddot - k1_inner*re_dot - k2_inner*re;
16 end

```

A.2.5 aux2.m

```

1 function y = aux2(u)
2     global params
3     I = params.I; m = params.mass; g = params.gravity;
4     Ixx = I(1,1); Iyy = I(2,2); Izz = I(3,3);
5
6     y = zeros(3,1);
7
8     ud = u(1:3); nu = u(4:6);
9     phi = u(7); theta = u(8); psi = u(9);
10    p = u(10); q = u(11); r = u(12);
11
12    ua = zeros(3,1);
13    gains = eye(3);
14
15    %phi
16    ua(1) = q*r*(Izz - Iyy) / Ixx;
17    gains(1,1) = Ixx;
18
19    %theta
20    ua(2) = p*r*(Ixx - Izz) / Iyy;
21    gains(2,2) = Iyy;
22
23    %psi
24    ua(3) = p*q*(Iyy - Ixx) / Izz;
25    gains(3,3) = Izz;
26
27    y = gains*(ua + nu + ud);
28
29 end

```

A.3 Utilities

A.3.1 plant.m

```

1 % Drone model
2 function y = plant(u)
3
4 global params w

```



```

5 y = zeros(6,1);
6 m = params.mass; g = params.gravity; I = params.I;
7
8 Ft = u(1); Tx = u(2); Ty = u(3); Tz = u(4);
9 phi = u(5); theta = u(6); psi = u(7);
10 p = u(8); q = u(9); r = u(10); t = u(11);
11
12
13 %Rotation Matrix
14 R = [cos(psi)*cos(theta)-sin(phi)*sin(psi)*sin(theta), -cos(phi)*sin(psi), cos(psi)
      *sin(theta)+cos(theta)*sin(phi)*sin(psi);
      cos(theta)*sin(psi)+cos(psi)*sin(phi)*sin(theta), cos(psi)*cos(phi), sin(psi)
      )*sin(theta)-cos(psi)*cos(theta)*sin(phi);
      -cos(phi)*sin(theta), sin(phi), cos(phi)*cos(theta)];
17
18 % get acceleration in the 3 axes
19 accel = [0; 0; -g] + R*[0; 0; Ft/m];% + [m*g*sin(w*t);m*g*sin(w*t);m*g*sin(w*t)/3];
20
21 % get angular accelerations
22 omega = I\[Tx;Ty;Tz] - I\(\cross([p; q; r],I*[p; q; r]));
23
24 y = [accel;omega];
25
26 end

```

A.3.2 clamper.m

```

1 function y = clamper(u)
2     global params
3     kf = params.force_coeff; km = params.moment_coeff;
4     rpm = params.max_rpm; l = params.length;
5
6     y = zeros(4,1);
7
8     F = u(1); Tx = u(2); Ty = u(3); Tz = u(4);
9
10    clamp = @(x, lb, ub) max(min(x, ub), lb);
11    w_max = rpm*2*pi/60;
12    Fmax = kf*w_max*w_max;
13    Tmax = kf*w_max*w_max * l;
14
15    F = clamp(F, 0, 4*Fmax);
16    Tx = clamp(Tx, -Tmax, Tmax);
17    Ty = clamp(Ty, -Tmax, Tmax);
18    Tz = clamp(Tz, -km*2*w_max^2, km*2*w_max^2);
19
20    y = [F; Tx; Ty; Tz];
21 end

```

A.3.3 graphResults.m

```

1 function graphResults()
2     global simout A w
3
4     t = simout.t;
5     x = simout.pos(:,1); y = simout.pos(:,2); z = simout.pos(:,3);
6     phi = simout.ang(:,1); theta = simout.ang(:,2); psi = simout.ang(:,3);
7     phi_des = simout.des_ang(:,1); theta_des = simout.des_ang(:,2);
8
9     % x
10    figure;
11    set(gcf,'DefaultLineLineWidth',2);
12    set(gca,'FontSize',18,'FontWeight','bold');
13    plot(t,x); hold on;

```



```

14     plot(t,2*A*sin(w*t)); hold off;
15     title('x'); xlabel('Time (s)'); ylabel('x (m)');
16     grid on;
17
18     % y
19     figure;
20     set(gcf,'DefaultLineLineWidth',2);
21     set(gca,'FontSize',18,'FontWeight','bold');
22     plot(t,y); hold on;
23     plot(t,A*sin(w*t)); hold off;
24     title('y'); xlabel('Time (s)'); ylabel('y (m)');
25     grid on;
26
27     % z
28     figure;
29     set(gcf,'DefaultLineLineWidth',2);
30     set(gca,'FontSize',18,'FontWeight','bold');
31     plot(t,z); hold on;
32     plot(t,2*A*sin(w*t)); hold off;
33     title('z'); xlabel('Time (s)'); ylabel('z (m)');
34     grid on;
35
36     % phi
37     figure;
38     set(gcf,'DefaultLineLineWidth',2);
39     set(gca,'FontSize',18,'FontWeight','bold');
40     plot(t,phi);
41     hold on; plot(t,phi_des); hold off;
42     legend('\phi', '\phi_{des}');
43     title('\phi'); xlabel('Time (s)'); ylabel('\phi (radians)');
44     grid on;
45
46     % theta
47     figure;
48     set(gcf,'DefaultLineLineWidth',2);
49     set(gca,'FontSize',18,'FontWeight','bold');
50     plot(t,theta);
51     hold on; plot(t,theta_des); hold off;
52     legend('\theta', '\theta_{des}');
53     title('\theta'); xlabel('Time (s)'); ylabel('\theta (radians)');
54     grid on;
55
56     % psi
57     figure;
58     set(gcf,'DefaultLineLineWidth',2);
59     set(gca,'FontSize',18,'FontWeight','bold');
60     plot(t,psi);
61     title('\psi'); xlabel('Time (s)'); ylabel('\psi (radians)');
62     grid on;
63
64
65 end

```

Appendix B

Python modules

B.1 PID Controller

B.1.1 pluto_utils.py

```
1 #!/usr/bin/env python
2
3  import rospy
4  from plutodrone.msg import PlutoMsg
5  from geometry_msgs.msg import PoseArray
6  from std_msgs.msg import Int32
7  from std_msgs.msg import Float32
8  from std_msgs.msg import Float64
9  import numpy as np
10
11 class DroneFly():
12     """Class for controlling the PlutoX Quadrotor"""
13     def __init__(self, des_pos=np.array([0,0,20])):
14         self.pluto_cmd = rospy.Publisher('/drone_command', PlutoMsg, queue_size=10)
15
16         rospy.Subscriber('whycon/poses', PoseArray, self.get_pose)
17         # rospy.Subscriber('/drone_yaw', Float64, self.get_yaw)
18
19
20         self.cmd = PlutoMsg()
21
22         # position as x,y,z,yaw
23         self.des_pos = np.array([0,0,20,0])
24         self.curr_pos = np.array([0,0,0,0])
25
26         self.prev_error = np.array([0,0,0,0])
27
28         # kp and kd - pitch, roll, throttle, yaw
29         self.kp = np.array([0,0,200,0])
30         self.kd = np.array([0,0,75,0])
31
32         # input as pitch, roll, throttle, yaw
33         self.input = np.array([0,0,0,0])
34
35         self.cmd.rcRoll = 1500
36         self.cmd.rcPitch = 1500
37         self.cmd.rcYaw = 1500
38         self.cmd.rcThrottle = 1500
39         self.cmd.rcAUX1 = 1500
40         self.cmd.rcAUX2 = 1500
41         self.cmd.rcAUX3 = 1500
42         self.cmd.rcAUX4 = 1000
43         self.cmd.plutoIndex = 0
44
45         self.i = 0
46         rospy.sleep(.1)
47
48     def isThere(self):
```




```

49     if (abs(self.des_pos - self.curr_pos) > 0.1).all():
50         return True
51     else:
52         return False
53
54     def position_hold(self):
55         while not rospy.is_shutdown():
56             self.calcPID()
57             self.cmd.rcRoll = self.input[1] + 1500
58             self.cmd.rcPitch = self.input[0] + 1500
59             self.cmd.rcYaw = self.input[3] + 1500
60             self.cmd.rcThrottle = self.clamp(-self.input[2] + 1500, 1350, 2100)
61             print("Applied Throttle =", self.cmd.rcThrottle)
62
63             print('Reached =', self.isThere())
64             print('Current Position =', self.curr_pos)
65
66             #Publish the values
67             self.pluto_cmd.publish(self.cmd)
68             rospy.sleep(.01)
69
70
71     def calcPID(self):
72         err = self.des_pos - self.curr_pos
73         self.input = self.kp*err + self.kd*(self.prev_error - err)
74         self.prev_error = err
75
76     def clamp(self, x, lb, ub):
77         return max(min(x,ub),lb)
78
79     def arm(self):
80         self.cmd.rcAUX4 = 1500
81         self.cmd.rcThrottle = 1000
82         self.pluto_cmd.publish(self.cmd)
83         rospy.sleep(.1)
84
85     def disarm(self):
86         self.cmd.rcAUX4 = 1100
87         # self.cmd.rcThrottle = 1600
88         self.pluto_cmd.publish(self.cmd)
89         rospy.sleep(.1)
90
91     def get_pose(self, pose):
92
93         #This is the subscriber function to get the whycon poses
94         #The x, y and z values are stored within the drone_x, drone_y and the drone_z
95         #variables
96
97         x = pose.poses[0].position.x
98         y = pose.poses[0].position.y
99         z = pose.poses[0].position.z
100         self.curr_pos = np.array([x,y,z,0])
101
102     # def get_yaw(self, yaw):
103     #
104     #     #This is the subscriber function to get the whycon poses
105     #     #The x, y and z values are stored within the drone_x, drone_y and the drone_z
106     #     #variables
107     #
108     #     self.currentyaw = yaw.data

```

B.1.2 pluto_pid_control.py

```

1  #!/usr/bin/env python
2  import rospy
3  from pluto_utils import DroneFly
4  import numpy as np
5
6  def control_drone():
7      print("Initializing drone....")

```



```

8     drone = DroneFly(des_pos=np.array([0,0,20]))
9     rospy.sleep(1)
10    drone.disarm()
11    rospy.sleep(1)
12    print("Arming Drone.....")
13    drone.arm()
14    rospy.sleep(1)
15    drone.position_hold()
16    print("\nDisarming drone.....")
17    drone.disarm()
18    rospy.sleep(1)
19    print("Exiting....")
20
21    if __name__ == '__main__':
22        rospy.init_node('pluto_pid', anonymous=False)
23        print("Starting.....")
24        control_drone()

```

B.2 UDE Controller

B.2.1 main.py

```

1  #!/usr/bin/env python
2  '''basically this subscribes to topics published on by the inner and outer
   controller scripts
3  and publishes those required values onto /drone_command'''
4
5  import rospy
6  import numpy as np
7
8  from geometry_msgs.msg import PoseArray
9  from plutodrone.msg import PlutoMsg
10 from pluto_controller.msg import ThrustMsg
11 from pluto_controller.msg import TorqueMsg
12
13 class Commander:
14     '''Class that commands the drone with required thrust and torques.
15     Uses the outer and inner loop controllers to find the desired control.
16     '''
17
18     # TODO: set up and subscribe to a topic for ref trajectory
19     def __init__(self, des_pos=None):
20         self._outer_controller_sub = rospy.Subscriber('/commander/thrust',
21                                                         ThrustMsg,
22                                                         self.thrust_callback)
23         self._inner_controller_sub = rospy.Subscriber('/commander/torques',
24                                                         TorqueMsg,
25                                                         self.torque_callback)
26
27         # TODO: setup subscribers and publishers, add proper args to above
28         statements
29         self.pluto_cmd = rospy.Publisher('/drone_command', PlutoMsg, queue_size=10)
30
31         self._pos_sub = rospy.Subscriber('whycon/poses', PoseArray, self.get_pose)
32         self.curr_pos = np.array([0, 0, 0], dtype=np.float64)
33
34         self.cmd = PlutoMsg()
35         self._req_thrust = 0
36         self._req_torques = [0, 0, 0]
37
38         self.cmd.rcRoll = 1500
39         self.cmd.rcPitch = 1500
40         self.cmd.rcYaw = 1500
41         self.cmd.rcThrottle = 1500
42         self.cmd.rcAUX1 = 1500
43         self.cmd.rcAUX2 = 1500
44         self.cmd.rcAUX3 = 1500
45         self.cmd.rcAUX4 = 1000

```



```

45     self.cmd.plutoIndex = 0
46
47     self.i = 0
48
49     self.des_pos = des_pos
50     rospy.sleep(.1)
51
52     def __clamp(self, x, lb, ub):
53         '''Clamp values'''
54         return max(min(x, ub), lb)
55
56     def isThere(self):
57         '''Function to check if drone is at desired position'''
58         if (abs(self.des_pos - self.curr_pos) > 0.1).all():
59             return True
60         else:
61             return False
62
63     def control(self):
64         '''Function to run the controller'''
65         # Linear mapping lambdas
66         thrust_map = lambda x: round(1000 * (1 + x/1.1172))
67         torque_xy_map = lambda x: round(11049.72 * x + 1500)
68         torque_z_map = lambda x: round(64516.13 * x + 1500)
69
70         while not rospy.is_shutdown():
71             self.cmd.rcRoll = torque_xy_map(self._req_torques[0])
72             self.cmd.rcPitch = torque_xy_map(self._req_torques[1])
73             self.cmd.rcYaw = torque_z_map(self._req_torques[2])
74             self.cmd.rcThrottle = thrust_map(self._req_thrust)
75
76             print("Applied Throttle =", self.cmd.rcThrottle)
77
78             print('Reached =', self.isThere())
79             print('Current Position =', self.curr_pos)
80
81             #Publish the commanded values
82             self.pluto_cmd.publish(self.cmd)
83             rospy.sleep(.01)
84
85     def thrust_callback(self, msg):
86         '''Callback for the thrust'''
87         self._req_thrust = msg.data.thrust_z
88
89     def torque_callback(self, msg):
90         '''Callback for the torque'''
91         self._req_torques = [msg.data.torque_x,
92                             msg.data.torque_y,
93                             msg.data.torque_z]
94
95     def arm(self):
96         '''Function to arm the drone'''
97         self.cmd.rcAUX4 = 1500
98         self.cmd.rcThrottle = 1000
99         self.pluto_cmd.publish(self.cmd)
100         rospy.sleep(.1)
101
102     def disarm(self):
103         '''Function to disarm the drone'''
104         self.cmd.rcAUX4 = 1100
105         # self.cmd.rcThrottle = 1600
106         self.pluto_cmd.publish(self.cmd)
107         rospy.sleep(.1)
108
109     def get_pose(self, msg):
110         self.curr_pos[0] = msg.poses[0].position.x
111         self.curr_pos[1] = msg.poses[0].position.y
112         self.curr_pos[2] = msg.poses[0].position.z
113
114     def main():
115         rospy.init_node('pluto_ude', anonymous=False)
116
117         print("Initializing drone....")
118
119         commander = Commander(des_pos=np.array([0, 0, 10], dtype=np.float64))
120         commander.control()

```



```

121
122     rospy.sleep(1)
123     commander.disarm()
124     rospy.sleep(1)
125
126     print("Arming Drone.....")
127     commander.arm()
128     rospy.sleep(1)
129     commander.control()
130
131     print("\nDisarming drone.....")
132     commander.disarm()
133     rospy.sleep(1)
134     print("Exiting.....")
135
136
137 if __name__ == "__main__":
138     try:
139         main()
140     except rospy.ROSInterruptException:
141         pass

```

B.2.2 outer_loop_controller.py

```

1  #!/usr/bin/env python
2  '''Outer Loop Position Controller for Drone'''
3
4  import rospy
5  import numpy as np
6
7  from geometry_msgs.msg import PoseArray
8  from plutodrone.msg import PlutoSensorMsg
9  from pluto_controller.msg import AttitudeMsg
10 from pluto_controller.msg import ThrustMsg
11
12 from ude_base_controller import UDEBaseController
13 from integral import ForwardEuler
14 from params import DroneParams
15
16 class OuterLoopController:
17
18     '''Class for Outer Loop Controller. Implements the UDEBaseController'''
19     def __init__(self, rate=100):
20         '''Constructor'''
21         self._x_controller = UDEBaseController()
22         self._y_controller = UDEBaseController()
23         self._z_controller = UDEBaseController()
24
25         self._f_virtual = np.array([0, 0, 0], dtype=np.float64)
26
27         self._phi_des = 0
28         self._theta_des = 0
29         self._yaw_des = 0
30         self._thrust = 0
31
32         # Subscribe to the positional data from the whycon
33         self._pose_sub = rospy.Subscriber('whycon/poses', PoseArray, self.
34         __pose_callback)
35         self._cur_pose = np.array([0, 0, 0], dtype=np.float64)
36
37         # Subscribe to the sensor data which gives the accelerations
38         self._acc_sub = rospy.Subscriber('/Pluto_sensor_data', PlutoSensorMsg, self
39         __acc_callback)
40         # Initialize 3 integrators for velocities in x, y and z
41         self._acc_integral = [ForwardEuler(IC=0, Ts=0.01)] * 3
42         self._cur_vel = np.array([0, 0, 0], dtype=np.float64)
43
44         # To use along with rospy.sleep() or other ways to control rate of
45         publishing
46         self._rate = rate

```



```

44
45     # Static ref trajectory, can be changed by subscriber to another topic
46     # TODO setup a topic for reference trajectory
47     self._ref = np.array([0, 0, 10], dtype=np.float64)
48     self._ref_dot = np.array([0, 0, 0], dtype=np.float64)
49
50
51     def __initialize_x_controller(self, model_params, gains, init_states, tau):
52         '''Function to initialize the x controller'''
53         self._x_controller.set_states(init_states[0], init_states[1])
54         self._x_controller.set_gains(gains[0], gains[1])
55         self._x_controller.set_model_params(model_params[0], model_params[1],
56 model_params[2])
57         self._x_controller.set_tau(tau)
58         # Hardcoded 0 as ua for x
59         self._x_controller.set_const_ua(0)
60
61     def __initialize_y_controller(self, model_params, gains, init_states, tau):
62         '''Function to initialize the y controller'''
63         self._y_controller.set_states(init_states[0], init_states[1])
64         self._y_controller.set_gains(gains[0], gains[1])
65         self._y_controller.set_model_params(model_params[0], model_params[1],
66 model_params[2])
67         self._y_controller.set_tau(tau)
68         # Hardcoded 0 as ua for y
69         self._y_controller.set_const_ua(0)
70
71     def __initialize_z_controller(self, model_params, gains, init_states, tau):
72         '''Function to initialize the z controller'''
73         self._z_controller.set_states(init_states[0], init_states[1])
74         self._z_controller.set_gains(gains[0], gains[1])
75         self._z_controller.set_model_params(model_params[0], model_params[1],
76 model_params[2])
77         self._z_controller.set_tau(tau)
78         # Hardcoded gravity as ua for z
79         self._z_controller.set_const_ua(9.8)
80
81     def __set_reference(self, ref, ref_dot):
82         '''Function to update the reference values the base controllers
83 are supposed to track
84 '''
85         self._x_controller.set_reference([ref[0], ref_dot[0]])
86         self._y_controller.set_reference([ref[1], ref_dot[1]])
87         self._z_controller.set_reference([ref[2], ref_dot[2]])
88
89     def __clamp(self, x, lb, ub):
90         '''Clamp the input'''
91         return max(min(x, ub), lb)
92
93     def __calculate_angles(self):
94         '''Function to calculate the angles to be output'''
95         self._f_virtual = np.array([self._x_controller.get_input(),
96                                     self._y_controller.get_input(),
97                                     self._z_controller.get_input()])
98
99         Zb = self._f_virtual / np.linalg.norm(self._f_virtual)
100
101         Xc = np.array([np.cos(self._yaw_des), np.sin(self._yaw_des), 0])
102         Yb = np.cross(Zb, Xc) / np.linalg.norm(np.cross(Zb, Xc))
103         Xb = np.cross(Yb, Zb)
104
105         R = np.transpose(np.array([Xb, Yb, Zb]))
106
107         cosine_phi = np.linalg.norm(R[[2, 2], [0, 2]])
108         self._phi_des = np.arctan2(R[2, 1], cosine_phi)
109         self._theta_des = np.arctan2(-R[2, 0], R[2, 2])
110
111         self._phi_des = self.__clamp(self._phi_des, -np.pi/4, np.pi/4)
112         self._theta_des = self.__clamp(self._theta_des, -np.pi/4, np.pi/4)
113         self._thrust = self._f_virtual[2]
114
115     def set_x_states(self, x_pos, x_vel):
116         '''Set the states of the x axis UDE controller'''
117         self._x_controller.set_states(x_pos, x_vel)
118
119     def set_y_states(self, y_pos, y_vel):

```



```

117         '''Set the states of the y axis UDE Controller'''
118         self._y_controller.set_states(y_pos, y_vel)
119
120     def set_z_states(self, z_pos, z_vel):
121         '''Set the states of the z axis UDE Controller'''
122         self._z_controller.set_states(z_pos, z_vel)
123
124     def set_yaw_des(self, yaw_des):
125         '''Set the desired yaw value'''
126         self._yaw_des = yaw_des
127
128     def get_phi_des(self):
129         '''Get the value of phi'''
130         return self._phi_des
131
132     def get_theta_des(self):
133         '''Get the value of theta'''
134         return self._theta_des
135
136     def get_psi_des(self):
137         '''Get the desired value of psi'''
138         return self._psi_des
139
140     def get_thrust(self):
141         '''Get the value of thrust'''
142         return self._thrust
143
144     def initialize_outer_loop_controller(self, model_params, gains, init_states,
145                                         tau):
146         '''Initialize the controllers'''
147         self.__initialize_x_controller(model_params[0], gains[0], init_states[0],
148                                       tau[0])
149         self.__initialize_y_controller(model_params[1], gains[1], init_states[1],
150                                       tau[1])
151         self.__initialize_z_controller(model_params[2], gains[2], init_states[2],
152                                       tau[2])
153
154     def update_outer_loop_controller(self):
155         '''Update the outer loop controller'''
156         # TODO update the states before calling update_controller
157
158         self.__set_reference(self._ref, self._ref_dot)
159         self._x_controller.update_controller()
160         self._y_controller.update_controller()
161         self._z_controller.update_controller()
162
163         self.__calculate_angles()
164
165     def __pose_callback(self, msg):
166         self._cur_pose = np.array([msg.poses[0].position.x,
167                                    msg.poses[0].position.y,
168                                    msg.poses[0].position.z])
169
170     def __acc_callback(self, msg):
171         self._cur_vel[0] = self._acc_integral[0](msg.accX)
172         self._cur_vel[1] = self._acc_integral[1](msg.accY)
173         self._cur_vel[2] = self._acc_integral[2](msg.accZ)
174
175     def main():
176         """Compute the desired attitudes and commanded thrust
177         and publish them to suitable topics.
178         """
179         des_att_pub = rospy.Publisher('/commander/desired_attitude', AttitudeMsg,
180                                       queue_size=15)
181         des_att_msg = AttitudeMsg()
182
183         thrust_pub = rospy.Publisher('/commander/thrust', ThrustMsg, queue_size=15)
184         thrust_msg = ThrustMsg()
185
186         rospy.init_node('outer_loop_controller', anonymous=True)
187
188         physical_params = DroneParams()
189         model_params = [[0, 0, 1/physical_params.mass],
190                        [0, 0, 1/physical_params.mass],
191                        [0, 0, 1/physical_params.mass]]

```



```

188     # str8 from sim
189     gains = [[1.6, 1.3],
190             [1.6, 1.3],
191             [1.6, 1.3]]
192     init_states = [[0, 0],
193                  [0, 0],
194                  [0, 0]]
195     tau = [0.5,
196           0.5,
197           0.5]
198
199     OLC = OuterLoopController()
200     OLC.initialize_outer_loop_controller(model_params, gains, init_states, tau)
201
202     # Run the outer controller at 100Hz
203     rate = rospy.Rate(100)
204
205     while not rospy.is_shutdown():
206         OLC.update_outer_loop_controller()
207
208         thrust_msg.thrust_z = OLC.get_thrust()
209
210         des_att_msg.phi = OLC.get_phi_des()
211         des_att_msg.theta = OLC.get_theta_des()
212         des_att_msg.psi = OLC.get_psi_des()
213
214         thrust_pub.publish(thrust_msg)
215         des_att_pub.publish(des_att_msg)
216
217         rate.sleep()
218
219
220 if __name__ == "__main__":
221     try:
222         main()
223     except rospy.ROSInterruptException:
224         pass

```

B.2.3 inner_loop_controller.py

```

1  #!/usr/bin/env python
2  '''Inner loop attitude controller using UDE'''
3
4  import rospy
5  import numpy as np
6
7  from geometry_msgs.msg import PoseArray
8  from plutodrone.msg import PlutoSensorMsg
9  from pluto_controller.msg import AttitudeMsg
10 from pluto_controller.msg import TorqueMsg
11
12 from ude_base_controller import UDEBaseController
13 from params import DroneParams
14
15 class InnerLoopController():
16     def __init__(self, params, rate=1000):
17         self._rate = rate
18         self._phi_controller = UDEBaseController()
19         self._theta_controller = UDEBaseController()
20         self._psi_controller = UDEBaseController()
21
22         self._torques = np.zeros((3, 1))
23         self._I = np.array(params.moments_of_inertia)
24
25         self._MAX_TORQUE_XY = params.MAX_TORQUE_XY
26         self._MAX_TORQUE_Z = params.MAX_TORQUE_Z
27
28         # Subscriber for raw sensor IMU data
29         self._att_sub = rospy.Subscriber('/Pluto_sensor_data',
30                                         PlutoSensorMsg,

```



```

31         self.__sensor_callback)
32     self._cur_angles = np.array([0, 0, 0], dtype=np.float64)
33     self._cur_omega = np.array([0, 0, 0], dtype=np.float64)
34
35     # Subscriber for desired attitude
36     self._desired_att_sub = rospy.Subscriber('/commander/desired_attitude',
37                                             AttitudeMsg,
38                                             self.__desired_att_callback)
39     self._ref_att = np.array([0, 0, 0], dtype=np.float64)
40
41
42     def __clamp(self, x, lb, ub):
43         return max(min(x, ub), lb)
44
45     def __initialize_phi_controller(self, gains, init_states, tau):
46         '''Function to initialize the phi controller'''
47         self._phi_controller.set_states(init_states[0], init_states[1])
48         self._phi_controller.set_gains(gains[0], gains[1])
49         self._phi_controller.set_tau(tau)
50
51     def __initialize_theta_controller(self, gains, init_states, tau):
52         '''Function to initialize the theta controller'''
53         self._theta_controller.set_states(init_states[0], init_states[1])
54         self._theta_controller.set_gains(gains[0], gains[1])
55         self._theta_controller.set_tau(tau)
56
57     def __initialize_psi_controller(self, gains, init_states, tau):
58         '''Function to initialize the psi controller'''
59         self._psi_controller.set_states(init_states[0], init_states[1])
60         self._psi_controller.set_gains(gains[0], gains[1])
61         self._psi_controller.set_tau(tau)
62
63     def __calculate_torques(self):
64         '''Function to find the torques to be commanded to the drone'''
65         self._torques = np.array([self._phi_controller.get_input(),
66                                   self._theta_controller.get_input(),
67                                   self._psi_controller.get_input()])
68
69         self._torques[0] = self.__clamp(self._torques[0],
70                                         -self._MAX_TORQUE_XY, self._MAX_TORQUE_XY)
71         self._torques[1] = self.__clamp(self._torques[1],
72                                         -self._MAX_TORQUE_XY, self._MAX_TORQUE_XY)
73         self._torques[2] = self.__clamp(self._torques[2],
74                                         -self._MAX_TORQUE_Z, self._MAX_TORQUE_Z)
75
76
77     def set_theta_states(self, theta, theta_vel):
78         '''Set the states of the theta axis UDE controller'''
79         self._theta_controller.set_states(theta, theta_vel)
80
81     def set_phi_states(self, phi, phi_vel):
82         '''Set the states of the y axis UDE Controller'''
83         self._phi_controller.set_states(phi, phi_vel)
84
85     def set_psi_states(self, psi, psi_vel):
86         '''Set the states of the z axis UDE Controller'''
87         self._psi_controller.set_states(psi, psi_vel)
88
89     def get_torques(self):
90         '''Get the torques to be commanded to the drone'''
91         return self._torques
92
93     def initialize_inner_loop_controller(self, gains, init_states, tau):
94         '''Initialize the controllers'''
95         self.__initialize_phi_controller(gains[0], init_states[0], tau[0])
96         self.__initialize_theta_controller(gains[1], init_states[1], tau[1])
97         self.__initialize_psi_controller(gains[2], init_states[2], tau[2])
98
99     def __sensor_callback(self, msg):
100         self._cur_angles = [msg.roll, msg.pitch, msg.yaw]
101         self._cur_omega = [msg.gyroX, msg.gyroY, msg.gyroZ]
102
103     def __desired_att_callback(self, msg):
104         self._ref_att[0] = msg.phi
105         self._ref_att[1] = msg.theta
106         self._ref_att[2] = msg.psi

```




```

107
108     def update_inner_loop_controller(self):
109         '''Update the outer loop controller'''
110
111         self._phi_controller.set_states(self._cur_angles[0], self._cur_omega[0])
112         self._theta_controller.set_states(self._cur_angles[1], self._cur_omega[1])
113         self._psi_controller.set_states(self._cur_angles[2], self._cur_omega[2])
114
115         p, q, r = self._cur_omega
116
117         phi_a2 = q*r*(self._I[2] - self._I[1]) / (p*self._I[0])
118         theta_a2 = p*r*(self._I[0] - self._I[2]) / (q*self._I[1])
119         psi_a2 = p*q*(self._I[1] - self._I[0]) / (r*self._I[2])
120
121         self._phi_controller.set_model_params(0, phi_a2, 1/self._I[0])
122         self._theta_controller.set_model_params(0, theta_a2, 1/self._I[1])
123         self._psi_controller.set_model_params(0, psi_a2, 1/self._I[2])
124
125         self._phi_controller.update_controller()
126         self._theta_controller.update_controller()
127         self._psi_controller.update_controller()
128
129         self._calculate_torques()
130
131
132 def main():
133     """Compute and publish values of commanded torques to a suitable topic.
134     The reference values of desired angles are subscribed from
135     /commander/desired_attitude
136     """
137
138     torque_pub = rospy.Publisher('/commander/torques', TorqueMsg, queue_size=15)
139     torque_msg = TorqueMsg()
140     rospy.init_node('inner_loop_controller', anonymous=True)
141
142     # str8 from sim
143     gains = [[20, 100],
144             [20, 100],
145             [20, 100]]
146     init_states = [[0, 0],
147                   [0, 0],
148                   [0, 0]]
149     tau = [0.1,
150           0.1,
151           0.1]
152
153     ILC = InnerLoopController(DroneParams())
154     ILC.initialize_inner_loop_controller(gains, init_states, tau)
155
156     # Inner loop runs at 1k Hz
157     rate = rospy.Rate(1000)
158
159     while not rospy.is_shutdown():
160         ILC.update_inner_loop_controller()
161
162         t = ILC.get_torques()
163         torque_msg.torque_x = t[0]
164         torque_msg.torque_y = t[1]
165         torque_msg.torque_z = t[2]
166
167         torque_pub.publish(torque_msg)
168
169         rate.sleep()
170
171 if __name__ == "__main__":
172     try:
173         main()
174     except rospy.ROSInterruptException:
175         pass
176

```



B.2.4 params.py

```

1  #!/usr/bin/env python
2
3  class DroneParams:
4      '''Class to store hardcoded physical parameters of the drone'''
5      def __init__(self):
6          self.mass = 56e-3
7          self.length = 65e-3
8          self.MAX_RPM = 3e4
9          self.gravity = 10
10         self.moments_of_inertia = [6.5e-5, 6.21e-5, 1.18e-4]
11         self.force_coeff = 2.83e-8
12         self.moment_coeff = 1.55e-10
13         self.MAX_THRUST = 1.1172
14         self.MAX_TORQUE_XY = 1.81e-2
15         self.MAX_TORQUE_Z = 3.1e-3

```

B.3 Utilities

B.3.1 integral.py

```

1  #!/usr/bin/env python
2
3  import numpy as np
4
5  class Integral(object):
6      def __init__(self, IC, Ts, gain):
7          self.IC = np.array(IC) if isinstance(IC, list) else IC
8          self.x = self.IC
9          self.Ts = Ts
10         self.K = gain
11         self.n = 0
12         self.y = 0
13
14         def getState(self):
15             return self.x
16
17         def __call__(self):
18             self.n += 1
19             return self.y
20
21         def reset(self, newIC=False):
22             self.n = 0
23             self.y = 0
24             if not newIC:
25                 self.x = self.IC
26             else:
27                 self.x = np.array(newIC)
28
29         def update(self):
30             pass
31
32     class ForwardEuler(Integral):
33         def __init__(self, IC, Ts, gain=1):
34             super().__init__(IC, Ts, gain)
35             self.y = self.IC
36
37         def update(self, u):
38             self.x = self.x + self.K * self.Ts * u
39             self.y = self.x
40
41         def __call__(self, u):
42             output = self.y
43             self.update(u)
44             self.n += 1

```



```

45         return output
46
47 class BackwardEuler(Integral):
48     def __init__(self, IC, Ts, gain=1):
49         super().__init__(IC, Ts, gain)
50
51     def update(self, u):
52         self.y = self.x + self.K * self.Ts * u
53         self.x = self.y
54
55     def __call__(self, u):
56         self.update(u)
57         self.n += 1
58         return self.y
59
60 class Trapezoidal(Integral):
61     def __init__(self, IC, Ts, gain=1):
62         super().__init__(IC, Ts, gain)
63
64     def update(self, u):
65         self.y = self.x + self.K * self.Ts/2 * u
66         self.x = self.y + self.K * self.Ts/2 * u
67
68     def __call__(self, u):
69         self.update(u)
70         self.n += 1
71         return self.y
72
73
74 if __name__ == "__main__":
75     # Unit testing
76     import matplotlib.pyplot as plt
77
78     # f(t) = 2*pi*sin(2*pi*t)
79     # f'(t) = -cos(2*pi*t)
80     t = np.linspace(0, 1, 101)
81     f = 2*np.pi * np.sin(2*np.pi*t)
82     f_drv = -np.cos(2*np.pi*t)
83
84     i1 = ForwardEuler(-1, 0.01)
85     i2 = BackwardEuler(-1, 0.01)
86     i3 = Trapezoidal(-1, 0.01)
87     i1_out = np.zeros(101)
88     i2_out = np.copy(i1_out)
89     i3_out = np.copy(i1_out)
90
91     for i in range(t.size):
92         i1_out[i] = i1(f[i])
93         i2_out[i] = i2(f[i])
94         i3_out[i] = i3(f[i])
95
96     fig, (ax1, ax2, ax3) = plt.subplots(3)
97     fig.suptitle("Integral tests")
98     ax1.plot(t, f_drv, label="f'(t)")
99     ax1.plot(t, i1_out, label="forward euler")
100    ax2.plot(t, f_drv, label="f'(t)")
101    ax2.plot(t, i2_out, label="backward euler")
102    ax3.plot(t, f_drv, label="f'(t)")
103    ax3.plot(t, i3_out, label="trapezoidal")
104    ax1.legend()
105    ax2.legend()
106    ax3.legend()
107    plt.show()
108
109    # diff1 = np.array(i1_out) - f_drv
110    # diff2 = np.array(i2_out) - f_drv
111    # diff3 = np.array(i3_out) - f_drv
112    # print(sum(diff1), sum(diff2), sum(diff3), sep='\n')

```

B.3.2 android_cam.py



```

1  #!/usr/bin/env python
2  import cv2
3  import rospy
4  from cv_bridge import CvBridge
5  from sensor_msgs.msg import Image
6  from sensor_msgs.msg import CameraInfo
7  import camera_info_manager
8
9
10 URL = "http://192.168.43.1:8080/video"
11 bridge = CvBridge()
12 rospy.init_node('android_image', anonymous=True)
13 img_publisher = rospy.Publisher('/usb_cam/image_raw', Image, queue_size=10)
14 cam_info_publisher = rospy.Publisher('/usb_cam/camera_info', CameraInfo, queue_size
    =10)
15
16 camera = cv2.VideoCapture(URL)
17
18 manager = camera_info_manager.CameraInfoManager(cname='android_cam')
19
20 manager.loadCameraInfo()
21
22 while True:
23     _, img = camera.read()
24     time = rospy.Time.now()
25     pub_img = bridge.cv2_to_imgmsg(img, encoding="bgr8")
26     pub_img.header.stamp = time
27     img_publisher.publish(pub_img)
28
29     cam_info = manager.getCameraInfo()
30     cam_info.header.stamp = time
31
32     print(cam_info)
33     cam_info_publisher.publish(cam_info)
34
35     cv2.imshow("Android cam", img)
36
37     if cv2.waitKey(1) == 27:
38         break
39 cv2.destroyAllWindows()

```

B.3.3 logger.py

```

1  #!/usr/bin/env python
2  '''Subscribes to the WhyCon position and logs it into a file'''
3
4  import rospy
5  from geometry_msgs.msg import PoseArray
6
7  LOG = open('log.txt', 'w')
8
9  def get_pose(pose):
10     '''Callback for the position data'''
11     x = pose.poses[0].position.x
12     y = pose.poses[0].position.y
13     z = pose.poses[0].position.z
14     time = rospy.get_time()
15
16     to_write = str(time) + ',' + str(x) + ',' + str(y) + ',' + str(z) + '\n'
17     LOG.write(str(to_write))
18
19 rospy.init_node('logger', anonymous=True)
20 rospy.Subscriber('whycon/poses', PoseArray, get_pose)

```



B.3.4 plotter.py

```
1  '''Reads the logs written and plots the x, y, and z positions'''
2  import matplotlib.pyplot as plt
3
4  LOG = open('log.txt', 'r')
5  time = []
6  x = []
7  y = []
8  z = []
9  for line in LOG:
10     line = line.rstrip('\n')
11     data = line.split(',')
12     data = [float(x) for x in data]
13     time.append(data[0])
14     x.append(data[1])
15     y.append(data[2])
16     z.append(data[3])
17  LOG.close()
18
19  plt.figure()
20  plt.plot(time, x)
21  plt.xlabel('Time')
22  plt.ylabel('Position')
23  plt.title('x')
24
25  plt.figure()
26  plt.plot(time, y)
27  plt.xlabel('Time')
28  plt.ylabel('Position')
29  plt.title('y')
30
31  plt.figure()
32  plt.plot(time, z)
33  plt.xlabel('Time')
34  plt.ylabel('Position')
35  plt.title('z')
36
37  plt.show()
```

Appendix C

C++ scripts

C.1 UDE based Attitude Controller

C.1.1 PlutoPilot.h

```
1 // Only modify this file to include
2 // - function definitions (prototypes)
3 // - include files
4 // - extern variable definitions
5 // In the appropriate section
6
7 #ifndef _PlutoPilot_H_
8 #define _PlutoPilot_H_
9
10 //add your includes for the project UDEControl here
11
12 double deciDeg2rad = 1.74533e-3;
13 double deg2rad = 0.0174533;
14 //end of add your includes here
15
16
17 //add your function definitions for the project UDEControl here
18
19
20
21
22 //Do not add code below this line
23 #endif /* _UDEControl_H_ */
```

C.1.2 PlutoPilot.cpp

```
1 // Do not remove the include below
2 #include "PlutoPilot.h"
3 #include "Utils.h"
4 #include "User.h"
5 #include "Estimate.h"
6 #include "Sensor.h"
7 #include "Motor.h"
8
9 #include "math.h"
10
11 #include "UDE.h"
12 #include "Localisation_custom.h"
13
14 Localisation_custom localiser;
15 UDE controller;
```



```

16 Interval T5;
17 //The setup function is called once at PlutoX's hardware startup
18 void plutoInit()
19 {
20 //Add your hardware initialization code here
21 LED.flightStatus(DEACTIVATE);
22 setUserLoopFrequency(5); // Runs loop at 200Hz
23
24 }
25
26
27
28
29 //The function is called once before plutoLoop() when you activate developer mode
30 void onLoopStart()
31 {
32 //Do your one time stuff here
33 localiser.resetLocaliser();
34 localiser.caliberateLocaliser();
35 Motor.init(M5); Motor.init(M6); Motor.init(M7); Motor.init(M8);
36 T5.reset();
37
38 }
39
40
41
42 //The loop function is called in an endless loop
43 void plutoLoop()
44 {
45 if (!App.isArmSwitchOn()) {
46     Motor.set(M5, 0); Motor.set(M6, 0);
47     Motor.set(M7, 0); Motor.set(M8, 0);
48 } else {
49     bool flag = T5.set(50, true);
50     if(flag) {
51         double velZ = Velocity.get(Z) * 1e-2;
52
53         controller.UpdateLinPosData(4);
54         controller.UpdateLinVelData(velZ);
55         controller.UpdatePosController();
56     }
57     double phi = Angle.get(AG_ROLL) * deciDeg2rad;
58     double theta = Angle.get(AG_PITCH) * deciDeg2rad;
59     double psi = Angle.get(AG_YAW);
60     if (psi > 180) psi -= 360;
61     psi *= deg2rad;
62
63     double p = Rate.get(X) * deciDeg2rad;
64     double q = Rate.get(Y) * deciDeg2rad;
65     double r = Rate.get(Z) * deciDeg2rad;
66
67     controller.UpdateAngPosData(phi, theta, psi);
68     controller.UpdateAngVelData(p, q, r);
69     controller.UpdateAttController();
70
71     Motor.set(M5, controller.motor_5_pwm);
72     Motor.set(M6, controller.motor_6_pwm);
73     Motor.set(M7, controller.motor_7_pwm);
74     Motor.set(M8, controller.motor_8_pwm);
75 }
76 }
77
78
79
80 //The function is called once after plutoLoop() when you deactivate developer mode
81 void onLoopFinish()
82 {
83 //Do your cleanup stuff here
84 LED.set(RED, OFF);
85 LED.set(BLUE, OFF);
86 LED.set(GREEN, OFF);
87 Motor.set(M5, 0);
88 Motor.set(M6, 0);
89 Motor.set(M7, 0);
90 Motor.set(M8, 0);
91 }

```



C.1.3 UDE.h

```

1  /*
2  * UDE.h
3  *
4  * Created on: 13-Sep-2021
5  * Author: Aravind S
6  */
7
8  #ifndef SRC_MAIN_UDE_H_
9  #define SRC_MAIN_UDE_H_
10
11 #include "Utils.h"
12 #include "params.h"
13 #include "stdint.h"
14
15 class UDE {
16 public:
17     UDE();
18     void UpdatePosController();
19     void UpdateAttController();
20     void UpdateLinPosData(double z);
21     void UpdateLinVelData(double z_dot);
22     void UpdateAngPosData(double phi, double theta, double psi);
23     void UpdateAngVelData(double p, double q, double r);
24     int16_t motor_5_pwm, motor_6_pwm, motor_7_pwm, motor_8_pwm;
25
26 private:
27
28     void UpdateMotorPWM();
29     void clamp(double* x, double lb, double ub);
30     void clamp(int16_t* x, int lb, int ub);
31
32     static const double k1_pos_ = 1.6, k2_pos_ = 1.3, k1_att_ = 20, k2_att_ = 100;
33     double Thrust_, Tx_, Ty_, Tz_;
34     static const double tau_pos_ = 0.4, tau_att_ = 0.01;
35
36     double prev_nu_z_;
37     double nu_z_integral_;
38     double prev_nu_phi_, prev_nu_theta_, prev_nu_psi_;
39     double nu_phi_integral_, nu_theta_integral_, nu_psi_integral_;
40
41     double z_, z_dot_;
42     double phi_, theta_, psi_, p_, q_, r_;
43
44     static const double z_ref_ = 10;
45     double z_ref_dot_;
46     double phi_ref_, theta_ref_, psi_ref_;
47
48     uint32_t prev_time_pos_, prev_time_att_;
49     params params_;
50     max_values max_values_;
51
52 };
53
54
55
56
57 #endif /* SRC_MAIN_UDE_H_ */

```

C.1.4 UDE.cpp

```

1  /*
2  * UDE.cpp

```



```

3  *
4  *   Created on: 13-Sep-2021
5  *       Author: Aravind S
6  */
7  #include "UDE.h"
8  #include "math.h"
9  #include "Utils.h"
10
11 UDE::UDE() {
12     motor_5_pwm = 1000; motor_6_pwm = 1000; motor_7_pwm = 1000; motor_8_pwm = 1000;
13     Thrust_ = 0; Tx_ = 0; Ty_ = 0; Tz_ = 0;
14     psi_ref_ = 0;
15     z_ref_dot_ = 0;
16     prev_time_pos_ = 0; prev_time_att_ = 0;
17     prev_nu_z_ = 0;
18     prev_nu_phi_ = 0; prev_nu_theta_ = 0; prev_nu_psi_ = 0;
19     nu_z_integral_ = 0;
20     nu_phi_integral_ = 0; nu_theta_integral_ = 0; nu_psi_integral_ = 0;
21     phi_ = 0; theta_ = 0; psi_ = 0;
22     p_ = 0; q_ = 0; r_ = 0;
23 }
24
25 void UDE::UpdateLinPosData(double z) {
26     z_ = z;
27 }
28
29 void UDE::UpdateLinVelData(double z_dot) {
30     z_dot_ = z_dot;
31 }
32
33 void UDE::UpdateAngPosData(double phi, double theta, double psi) {
34     phi_ = phi;
35     theta_ = theta;
36     psi_ = psi;
37 }
38
39 void UDE::UpdateAngVelData(double p, double q, double r) {
40     p_ = p;
41     q_ = q;
42     r_ = r;
43 }
44
45 void UDE::UpdatePosController() {
46     double z_err = z_ - z_ref_; double z_dot_err = z_dot_ - z_ref_dot_;
47
48     double nu_z = 0 - k1_pos*z_dot_err - k2_pos*z_err;
49
50     uint32_t now = micros();
51     double dt = (now - prev_time_pos_)*1e-6;
52     prev_time_pos_ = now;
53     nu_z_integral_ += (nu_z - prev_nu_z_)*dt;
54     prev_nu_z_ = nu_z;
55     double ud_z = (-1/tau_pos_) * (z_dot_ - nu_z_integral_);
56     double f_virtual_z = params_.mass * (params_.gravity + nu_z + ud_z);
57
58     phi_ref_ = 0;
59     theta_ref_ = 0;
60     Thrust_ = f_virtual_z;
61
62     clamp(&Thrust_, 0, max_values_.thrust);
63 }
64
65 void UDE::clamp(double* x, double lb, double ub) {
66     if (*x > ub) *x = ub;
67     if (*x < lb) *x = lb;
68 }
69
70 void UDE::clamp(int16_t* x, int lb, int ub) {
71     if (*x > ub) *x = ub;
72     if (*x < lb) *x = lb;
73 }
74
75 void UDE::UpdateAttController() {
76     double phi_err = phi_ - phi_ref_;
77     double theta_err = theta_ - theta_ref_;
78     double psi_err = psi_ - psi_ref_;

```



```

79
80 double nu_phi = 0 - k1_att_ * p_ - k2_att_ * phi_err;
81 double nu_theta = 0 - k1_att_ * q_ - k2_att_ * theta_err;
82 double nu_psi = 0 - k1_att_ * r_ - k2_att_ * psi_err;
83
84 uint32_t now = micros();
85 double dt = (now - prev_time_att_)*1e-6;
86 prev_time_att_ = now;
87
88 nu_phi_integral_ += (nu_phi + prev_nu_phi_)/2 * dt;
89 nu_theta_integral_ += (nu_theta + prev_nu_theta_)/2 * dt;
90 nu_psi_integral_ += (nu_psi + prev_nu_psi_)/2 * dt;
91
92 prev_nu_phi_ = nu_phi; prev_nu_theta_ = nu_theta; prev_nu_psi_ = nu_psi;
93
94 double ud_phi = (-1/tau_att_) * (p_ - nu_phi_integral_);
95 double ud_theta = (-1/tau_att_) * (q_ - nu_theta_integral_);
96 double ud_psi = (-1/tau_att_) * (r_ - nu_psi_integral_);
97
98 // phi
99 double ua = q_*r*(params_.Izz - params_.Iyy) / params_.Ixx;
100 Tx_ = params_.Ixx * (ua + nu_phi + ud_phi);
101 // theta
102 ua = p_*r*(params_.Ixx - params_.Izz) / params_.Iyy;
103 Ty_ = params_.Iyy * (ua + nu_theta + ud_theta);
104 // psi
105 ua = p_*q*(params_.Iyy - params_.Ixx) / params_.Izz;
106 Tz_ = params_.Izz * (ua + nu_psi + ud_psi);
107
108 clamp(&Tx_, -max_values_.Txy, max_values_.Txy);
109 clamp(&Ty_, -max_values_.Txy, max_values_.Txy);
110 clamp(&Tz_, -max_values_.Tz, max_values_.Tz);
111
112 UpdateMotorPWM();
113 }
114
115 void UDE::UpdateMotorPWM() {
116 // Convert Thrust and torque to Motor PWM
117 double x1 = Thrust_/max_values_.thrust;
118 double x2 = Tx_/max_values_.Txy;
119 double x3 = Ty_/max_values_.Txy;
120 double x4 = Tz_/max_values_.Tz;
121
122 double a1 = static_cast<int>(1050 + 750*x1);
123 double a2 = static_cast<int>(-200 + 200*(x2+1));
124 double a3 = static_cast<int>(-200 + 200*(x3+1));
125 double a4 = static_cast<int>(-200 + 200*(x4+1));
126
127 motor_6_pwm = a1 - a2 - a3 - a4;
128 motor_5_pwm = a1 - a2 + a3 + a4;
129 motor_7_pwm = a1 + a2 + a3 - a4;
130 motor_8_pwm = a1 + a2 - a3 + a4;
131
132 clamp(&motor_6_pwm, 1050, 2000);
133 clamp(&motor_8_pwm, 1050, 2000);
134 clamp(&motor_7_pwm, 1050, 2000);
135 clamp(&motor_5_pwm, 1050, 2000);
136
137 }

```

C.1.5 params.h

```

1 /*
2  * params.h
3  *
4  * Created on: 14-Sep-2021
5  * Author: Aravind S
6  */
7
8 #ifndef SRC_MAIN_PARAMS_H_

```



```
9  #define SRC_MAIN_PARAMS_H_
10
11  struct params {
12      static const double mass = 56e-3; //kg
13      static const double gravity = 9.81; //m/s^2
14      static const double length = 65e-3; //m
15      static const double Ixx = 6.5e-5;
16      static const double Iyy = 6.21e-5;
17      static const double Izz = 1.18e-4;
18      static const double force_coeff = 2.83e-8;
19      static const double moment_coeff = 1.55e-10;
20      static const double max_rpm = 30000;
21  };
22
23  struct max_values {
24      static const double angle = 0.785; //radians
25      // static const double thrust = 1.1172 ;//N
26      static const double thrust = 1.0 ;//N
27      static const double Txy = 1.81e-2;
28      static const double Tz = 3.1e-3;
29  };
30
31  static const double rad_s2rpm = 9.549297;
32
33
34  #endif /* SRC_MAIN_PARAMS_H_ */
```
