FLIGHT EVALUATION OF DEEP MODEL REFERENCE ADAPTIVE CONTROL

BY

JASVIR VIRDI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Assistant Professor Girish Chowdhary

ABSTRACT

This thesis presents flight test results for a new neuroadaptive architecture: Deep Neural Network based Model Reference Adaptive Control (DMRAC). This architecture utilizes the power of deep neural network representations for modeling significant nonlinearities while marrying it with the boundedness guarantees that characterize MRAC based controllers. Through experiments on a real quadcopter platform, it is shown that DMRAC can outperform state of the art controllers in different flight regimes while having long-term learning abilities. This makes DMRAC a highly powerful architecture for high-performance control of nonlinear systems.

To my family, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my family, especially my parents, for their sacrifices and my brother who has given me constant support during tough times. To my advisor, Dr. Girish Chowdhary for his enthusiasm, encouragement, support and for giving me the opportunity to work on this project. To Girish Joshi, for his candid advise regarding different control algorithms and debugging. To John Hart, for getting me access to Intelligent Robotics Lab (IRL). To Sri Theja Vupalla, for advising me regarding equipment for conducting experiments. And finally, to all the people who made my stay in Champaign memorable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

DMRAC	Deep Model Reference Adaptive Control
DNN	Deep Neural Networks
DOF	Degree of Freedom
FIFO	First In First Out
FIR	Finite Impulse Response
GP-MRAC	Gaussian Process Model Reference Adaptive Control
IIR	Infinite Impulse Response
IMU	Inertial Measurement Unit
IRL	Intelligent Robotics Lab
MRAC	Model Reference Adaptive Control
PID	Proportional Integral Derivative
SGD	Stochastic Gradient Descent
SVD	Singular Value Decomposition
UDP	User Datagram Protocol

CHAPTER 1

INTRODUCTION

Creating adaptive controllers for mobile robots that can learn online to handle a large variety of disturbances and operating environments while ensuring stability has been challenging problem. The key challenge is that robot dynamics can significantly change during operation due to different operating conditions, degradation, or failures. When such changes happen, heuristic, hand-crafted, or model-based controllers can fail. Even controllers that learn from experience, such as reinforcement learning (RL) can fail when the robot dynamics changes beyond what the RL agent was trained on.

Adaptive controllers that can learn online and in real-time to adapt to such changes have long been part of classical controls [1, 2, 3]. More recently, Model Reference Adaptive Controllers (MRAC) using shallow networks as the learning element have become a leading method for adaptive flight control, including for highly unstable rotorcraft [4, 5, 6, 7, 8, 9]. Methods such as Gaussian Process Model Reference Adaptive Control [10, 6], L1 adaptive control [11], and single hidden layer neural network based adaptive control [4, 12] have demonstrated quite a bit of success in adapting to disturbances during flight. However, a key drawback of these existing methods has been the lack of long-term learning: The shallow networks in these methods updated with Lyapunov theory derived gradient based rules can instantaneously adapt to mitigate the disturbance, but do not generalize to similar disturbances or operating conditions [12, 5]. Deep Neural Networks (DNNs) trained with dropouts and batch updates could certainly help alleviate these short-term learning issues [13, 14], but it has been difficult to train and update these networks in real-time on aerial robots with limited onboard computing while guaranteeing stability.

1.1 Motivation

Recently, a new neuroadaptive control architecture [13, 14] called Deep Model Reference Adaptive Control (DMRAC) has been shown theoretically as a possible way of integrating DNN's with MRAC to ensure long term learning while guaranteeing boundedness. In this thesis, flight evaluation of DMRAC is performed on a quadcopter platform in different flight regimes and comparison is shown with other existing algorithms such as PID and MRAC. Moreover, experiments have been conducted to demonstrate long term learning properties and generalisability of the above controller.

1.2 Parrot Mambo Mini Drone

Figure 1.1: Parrot mambo mini drone fitted with vicon markers

In order to perform flight tests, parrot mambo mini drone was chosen due to its small size, low cost, and relative ease of implementing and testing of algorithms. The algorithms were implemented using Simulink in an on-board, off-board architecture which will be discussed later. The above drone platform contains sensors such as IMU (3 axis accelerometer and 3 axis gyroscope), ultrasound, camera etc. The weight of the drone is about 63 grams and it has a 550mAh LiPo battery which can give about 9 minutes of flight time. The cross section dimensions of the drone are 18 x 18 cm.

1.3 Test Facility

Figure 1.2: Vicon Arena

The experiments for conducting flight evaluation of different controllers was conducted in CSL Studio VICON facility at the University of Illinois at Urbana Champaign. The vicon arena is equipped with a motion capture system that gives millimeter accuracy. The system is used for tracking and position feedback in an indoor facility where GPS is unavailable.

1.4 Overview

This thesis is organised as follows: In chapter 2, a brief overview of the equations of motion and dynamics of quadcopter are discussed. Chapter 3 focuses on state estimation using different onboard sensors and position information provided by Vicon system, which are subsequently used to build state feedback control laws. Chapter 4 contains experimental results where a comparison is shown between different controller's performance on a variety of tasks with varying disturbances. Also, experimental results pertaining to the notion of generalisability of DMRAC is shown along with performance improvement via Transfer Learning.

CHAPTER 2

KINEMATICS AND DYNAMICS OF QUADCOPTER

2.1 Introduction

In this chapter, a brief summary of the equations of motion of quadcopter are presented [1]. The convention followed here is in accordance with standard aeronautics literature. The inertial frame and the body frame are oriented in NED (North-East-Down) position with the frames' x axis pointing towards north, y axis points towards east and z axis pointing downward. The inertial frame is denoted as F^i, body frame as F^b.

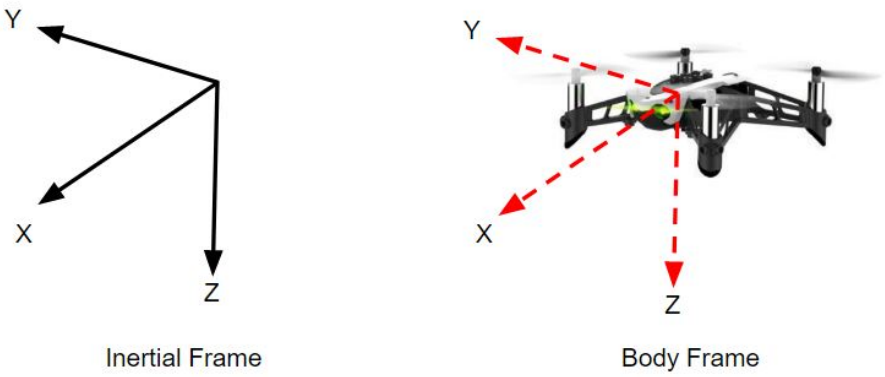

Figure 2.1: Representation of inertial and body frame. Both the frames are related via a rotation matrix which can be found if Euler angles are known

2.2 Quadrotor state representation and Equations of motion

The state of a quadcopter is defined as a vector comprising of 12 components as shown below:

X,Y Z = Position components pointing along NED directions in F^i

u,v,w = Body frame velocity components along NED directions

ϕ, θ, ψ = Euler angles (roll,pitch and yaw respectively) relating F^b and F^i

p,q,r = Body frame angular velocity components along NED directions

The derivation of equations of motion can be found in detail in [15]. Here, the final summary of equations is presented:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.1}$$

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & sin\phi tan\theta & cos\phi tan\theta \\ 0 & cos\phi & -sin\phi \\ 0 & \frac{sin\phi}{cos\theta} & \frac{cos\phi}{cos\theta} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{Jy-Jz}{Jx} qr \\ \frac{Jz-Jx}{Jy} pr \\ \frac{Jz-Jy}{Jz} pq \end{bmatrix} + \begin{bmatrix} \frac{\tau_\phi}{J_x} \\ \frac{\tau_\theta}{J_y} \\ \frac{\tau_\psi}{J_z} \end{bmatrix} \tag{2.4}$$

In equation (2.1), "sin" and "cos" are represented as "s" and "c" respectively. In equation (2.2), f_x, f_y, f_z represent body forces acting at center of mass. The thrust force and gravity are both incorporated in f_z. In equation (2.4), $\tau_\phi, \tau_\theta, \tau_\psi$ represent rotational torques generated by the propellers whereas J_x, J_y, J_z represent moments of inertia along x,y and z axes of the attached body frame.

CHAPTER 3

STATE ESTIMATION

3.1 Introduction

State estimation is vitally important for developing state feedback control laws. This is achieved in the case of quadrotor by fusing data from different onboard sensors and the Vicon system. In the first half of this chapter, a brief discussion about interfacing with Vicon to get position information and designing of estimator is done. In the latter half of the chapter, a detailed analysis is presented on how control laws were developed for the quadrotor.

3.2 Interfacing with Vicon System

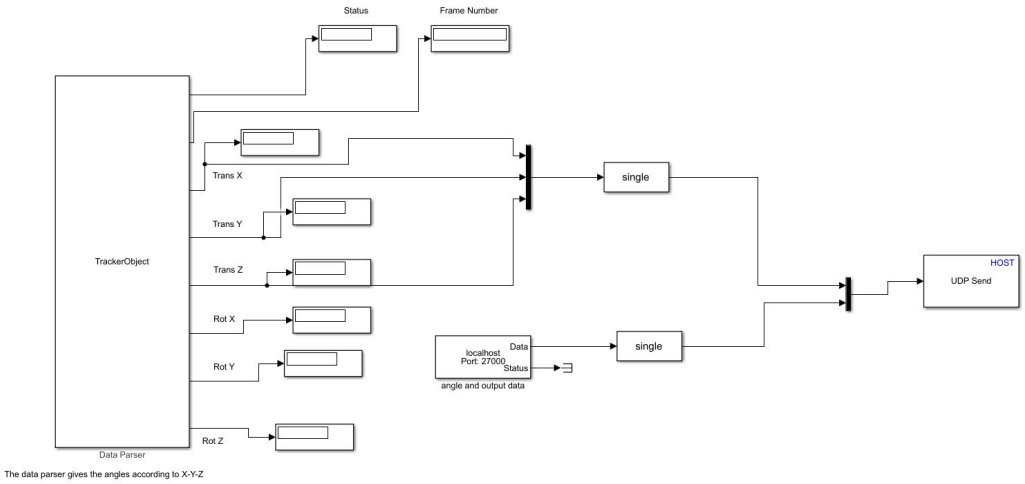

Figure 3.1: Vicon to drone communication system

The block labelled as "Data Parser" is an S function which interfaces Vicon with Simulink. There is a SDK (Software Development Kit) provided by Vicon [16] which is used in this block to access position and orientation

information of the drone inside the Vicon arena in real time. However, in this particular case, only position information is used in the estimator for correcting position and velocity estimates. The information is sent to the drone as a datapacket via UDP protocol.

3.3 State estimator architecture

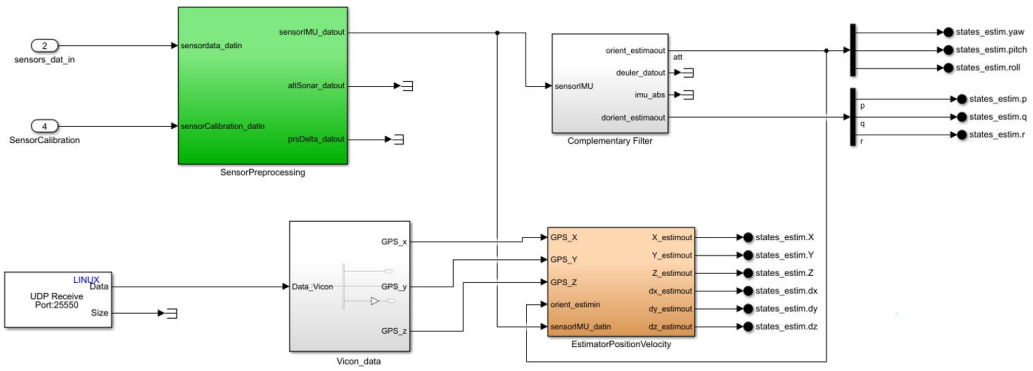

Figure 3.2: Overall estimator architecture for quadrotor system

The estimator can be divided into 5 main sub systems. The "UDP Receive" block receives position data of the quadrotor from the Vicon system. The "Vicondata" block is responsible for converting data into proper NED representation. The "SensorPreprocessing" block is used for filtering IMU data to remove noise. The "Complementary Filter" block houses a complementary filter which uses filtered IMU data to estimate the Euler angles and different angular velocity components. The "EstimatorPositionVelocity" block has a Kalman filter where filtered IMU accelerometer data is used as the prediction step and the data from the Vicon system is used as the correction step. The output from this system is position and velocity of the quadrotor in the inertial frame. (Note[17]: The above filter was derived from the work done by Sertak Karaman and Fabian Riether on parrot mini drones). In the subsequent subsections, each block is explained in more detail.

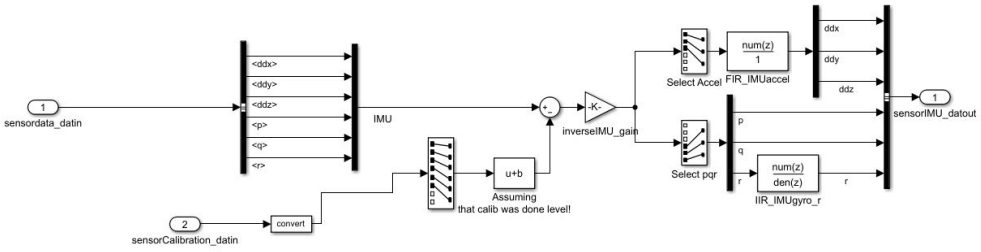

Figure 3.3: Sensor Preprocessing subsystem

3.3.1 Sensor Preprocessing block

The sensor preprocessing subsystem involves a calibration step where acceleration due to gravity term is subtracted from the z component of accelerometer reading. This is essential as accelerometers only measure proper acceleration values. The sensor data after the calibration step is passed through a FIR (Finite Impulse Response) and an IIR (Infinite Impulse Response) filter to remove noise from it. The FIR filter used here is a fifth order filter of the form:

$$y[n] = \sum_{i=0}^{5} b_i x[n-i] \tag{3.1}$$

In the above equation, y[n] denotes the output signal at n^{th} time instance, x[n] denotes the input signal and b_i denote the coefficients. The IIR filter used here is of direct form II and is of fifth order. The equation for it is given below:

$$v[n] = x(n) - \sum_{i=1}^{5} a_i v[n-i] \tag{3.2}$$

$$y[n] = \sum_{j=0}^{5} b_j v[n-j] \tag{3.3}$$

Here, y[n] and x[n] denote the same meanings as was seen in FIR filter.

3.3.2 Vicon data block

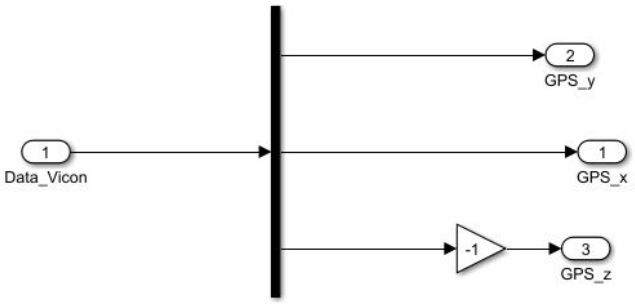

Figure 3.4: Vicon data conversion to NED inertial frame

The Vicon data block is responsible for converting the position information sent by the Vicon system to the correct NED inertial frame.

3.3.3 Complementary filter

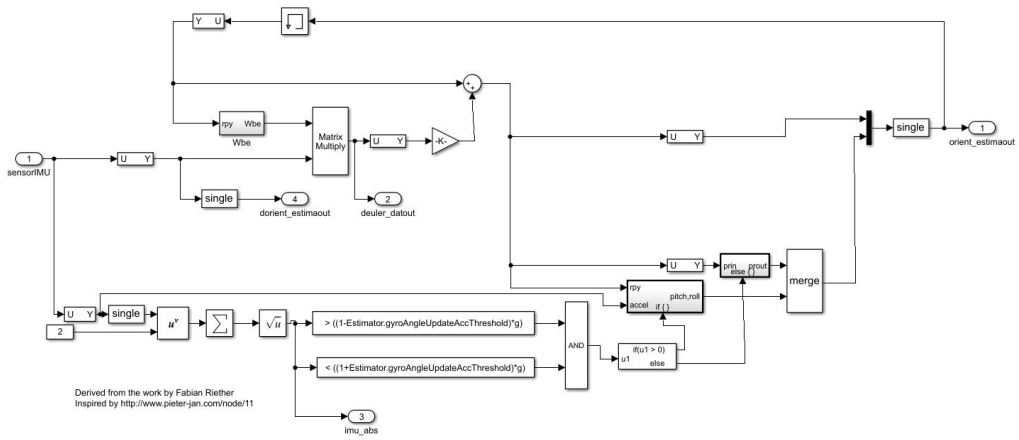

Figure 3.5: Complementary filter

The complementary filter's main role is to give correct estimates of roll and pitch angles from filtered IMU data. It works on the principle of combining fast changing data like angular velocity which is measured by gyroscopes with the slow changing accelerometer data. The way it works is as follows: if

$| u |< (1-\epsilon)g$ or $| u |> (1+\epsilon)g$ where u denotes the accelerometer reading, ϵ is a constant greater than 0 and g denotes the acceleration due to gravity, the angular velocity is integrated to get Euler angle readings. As soon as $| u |$ comes within the above range, it implies that the drone is relatively static and the absolute value of acceleration is around g. This is used to correct the values of Euler angles that are just based on simple integration of angular velocities . The current pitch reading, θ is changed to $(1-\gamma)\theta + (\gamma)asin(\frac{a_x}{g})$ and the current roll reading, ϕ is changed to $(1-\gamma)\phi + (\gamma)atan(\frac{a_y}{a_z})$. Here, γ is a constant.

3.3.4 Kalman Filter

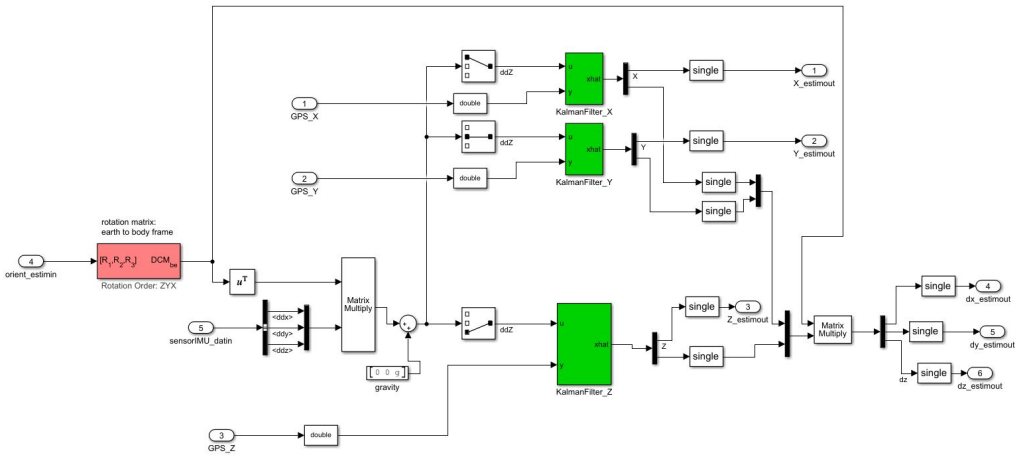

Figure 3.6: Kalman filtering for determining position and velocity

In order to determine position and velocity of the drone inside the Vicon Arena, position information provided by Vicon system and IMU accelerometer measurements are combined together using Kalman filters. Kalman filters used in this model are of discrete type that are already provided in Simulink as ready to use blocks. Since acceleration readings are in body frame whereas Vicon readings are in inertial frame, before fusing them together, acceleration readings are converted to inertial frame readings by multiplying it with a rotation matrix that relates body frame with the inertial frame.

CHAPTER 4

CONTROL OF QUADROTOR

4.1 Introduction

In this chapter, different control techniques are discussed which are used to make Quadrotor fly autonomously to follow a given reference trajectory. This chapter starts with the basic architecture of controller and subsequently details about implementation of different controllers are discussed in more detail.

4.2 Control System Architecture

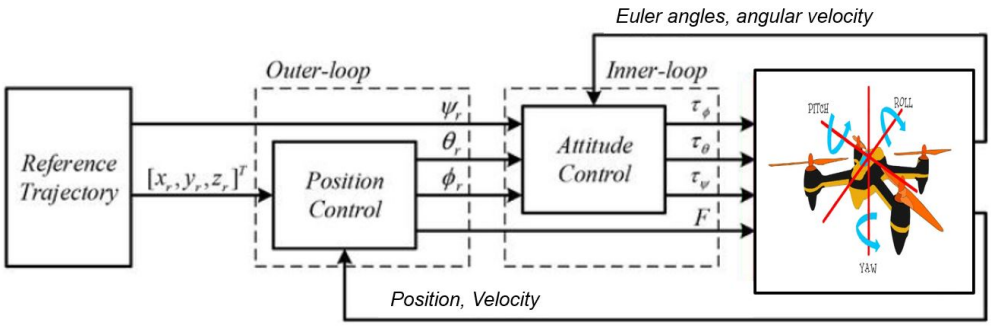

Figure 4.1: Overall control architecture diagram

The entire controller logic can be divided into 4 main categories as shown above.The reference trajectory module is where the trajectory that the drone is required to follow is specified. Position control/Outer loop control module is responsible for generating thrust force,reference roll and pitch angles by using reference trajectory, position and velocity of the drone as inputs. The third module is the inner loop/attitude controller which generates ap-

propriate roll,pitch and yaw torques to ensure roll,pitch and yaw reference angles are tracked as closely as possible. Finally, the thrust force and output torques generated by outer and inner loop controller respectively are converted to motor RPMs using a mixer model, which are then fed to the drone. The internal onboard sensors and Vicon system send state information in real time to each controller and the entire loop is complete.

4.2.1 Position Control-Outerloop Controller

In outerloop control, reference roll and pitch angles are calculated approximately by linearising equations of motion about the hover state. The acceleration in north and east direction in inertial frame can be rewritten as:

$$\ddot{X} = -\theta cos\psi - \phi sin\psi \tag{4.1}$$

$$\ddot{Y} = -\theta sin\psi + \phi cos\psi \tag{4.2}$$

The main goal of the outerloop controller is to make [X,Y] follow [X_{ref},Y_{ref}] as closely as possible. This is done by using a PID controller whose input is the current [X,Y] position and reference signal is [X_{ref},Y_{ref}]. After, solving the above equations for reference roll and pitch angles:

$$\theta_{ref} = -\text{PID}(X, X_{ref})cos\psi - \text{PID}(Y, Y_{ref})sin\psi \tag{4.3}$$

$$\phi_{ref} = -\text{PID}(X, X_{ref})sin\psi + \text{PID}(Y, Y_{ref})cos\psi \tag{4.4}$$

where PID function is expressed in the following form:

$$\text{PID}(a, a_{ref}) = K_P(a_{ref} - a) + K_I(\int_0^t (a_{ref} - a)dt) + K_D(\frac{d}{dt}(a_{ref} - a)) \tag{4.5}$$

In the above equation, K_P,K_I and K_D denote proportional, integral and derivative constants.

4.3 Inner Loop Control Algorithm

The inner loop controller/attitude controller is responsible for generating appropriate torques so that the quadrotor follows the reference roll,pitch and yaw angles as closely as possible. This can be done in various ways. However, for these experiments,three main control algorithms were considered: PID, MRAC and DMRAC.

4.3.1 PID

In PID control, the three torque inputs to the system namely yaw, pitch and roll torques are calculated based on the attitude angles found by the estimator and reference angles which are calculated by outer loop controller. The algorithm is given as follows:

$$\tau_{in}(x, x_{ref}) = K_P(x_{ref} - x) + K_I(\int_0^t (x_{ref} - x)dt) + K_D(\frac{d}{dt}(x_{ref} - x)) \quad (4.6)$$

Here, x=$\{\phi, \theta, \psi\}$, x_{ref}=$\{\phi_{ref}, \theta_{ref}, \psi_{ref}\}$ and τ_{in}=$\{\tau_\phi, \tau_\theta, \tau_\psi\}$

4.3.2 Model Reference Adaptive Control

In Model Reference Adaptive Control and Deep Model Reference Adaptive Control which will be discussed in the subsequent section, the control system architecture is modified slightly.

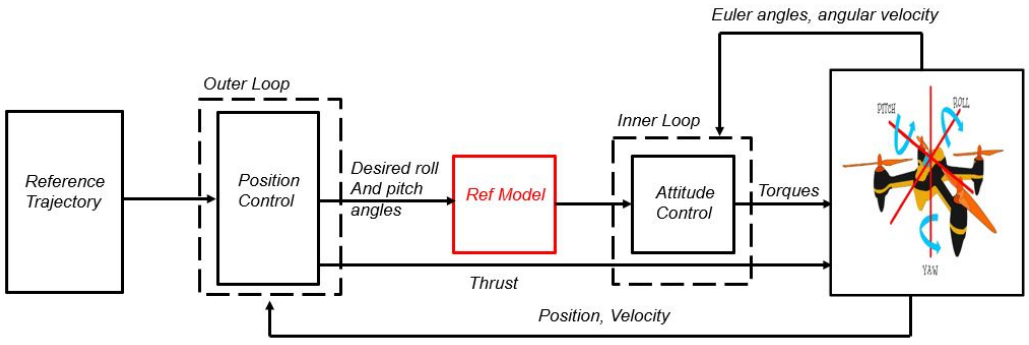

Figure 4.2: Overall Control System Architecture for MRAC and DMRAC

The red block as shown in the above figure is a reference model. Here, the

goal of the attitude controller is to follow the output of the reference model as closely as possible. The reference model is a system which has some desired characteristics and for these experiments, it is modeled as a second order system. The reference model's equations are as follows:

$$\dot{X_{rm}} = A_{rm}X_{rm} + B_{rm}r(t) \tag{4.7}$$

$$X_{rm} = \begin{bmatrix} \phi_{rm} \\ \dot{\phi_{rm}} \\ \theta_{rm} \\ \dot{\theta_{rm}} \\ \psi_{rm} \\ \dot{\psi_{rm}} \end{bmatrix} B_{rm} = \begin{bmatrix} 0 & 0 & 0 \\ w_n^2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & w_n^2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & w_n^2 \end{bmatrix} r(t) = \begin{bmatrix} \phi_{ref} \\ \theta_{ref} \\ \psi_{ref} \end{bmatrix} \tag{4.8}$$

$$A_{rm} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -w_n^2 & -2\gamma w_n & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -w_n^2 & -2\gamma w_n & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -w_n^2 & -2\gamma w_n \end{bmatrix} \tag{4.9}$$

Here, r(t) is the reference signal whose first two values, that is reference roll and pitch angles are outputs from outer loop controller and the third value, reference yaw, is output from reference trajectory module. The natural frequency w_n and damping γ are chosen according to the desired characteristics.

In order to build MRAC for the quadrotor, following dynamics model is considered:

$$\dot{X} = AX + B(u(t) + \Delta(x)) \tag{4.10}$$

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} X = \begin{bmatrix} \phi \\ \dot{\phi} \\ \theta \\ \dot{\theta} \\ \psi \\ \dot{\psi} \end{bmatrix} B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} u(t) = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{4.11}$$

Here $\Delta(x)$ captures uncertainity which could be unmodeled dynamics such as disturbances etc. This term is not known apriori. The control effort u(t) can be split into 3 main parts:

$$u(t) = u_{pid}(t) - v_{ad}(t) \tag{4.12}$$

$u_{pid}(t)$ is responsible for ensuring the current states match the reference states when there are no uncertainities inside the system. This is achieved through standard PID control in this case. The adaptive control effort's main objective is to negate the uncertainity inside the system. If the uncertainity was known perfectly well apriori, then $v_{ad}(t) = \Delta(x)$. However, realistically, since it is not known, $v_{ad}(t) = \hat{\Delta}(x)$. Using equation (4.7) and (4.10), the error dynamics can be computed as follows:

$$e(t) = X_{rm}(t) - X(t) \tag{4.13}$$

$$\dot{e(t)} = A_{rm}e(t) + B(v_{ad} - \Delta(x)) \tag{4.14}$$

Here, uncertainty, $\Delta(x)$, is modeled as an unstructured uncertainity which is defined as a continuous function over a compact set as follows:

$$\Delta(x) = W^{*T}\phi(x) + \epsilon_1(x), \forall x(t) \in D_x \subset R^n \tag{4.15}$$

$\phi(x)$ is basis function of a Neural Network adaptive element [18], or Gaussian Basis Function Network [19]. W^* is a set of ideal weights corresponding to that basis function. The modelling error $\epsilon_1(x)$ is upper bounded, s.t, $\tilde{\epsilon_1} = sup_{x \in D_x}||\epsilon_1(x)||$ can be made arbitratily small given a large number of basis functions. In this work for implementing MRAC on the quadrotor, Gaussian radial basis functions are used. The Gaussian radial basis functions are expressed as :

$$\phi(x) = e^{-(\frac{x-\mu}{\sigma})^2} \tag{4.16}$$

where μ is center,σ is the standard deviation which are assumed apriori.

The adaptive control elements is given as :

$$v_{ad}(x) = W^T\phi(x) \tag{4.17}$$

Using Lyapunov theory and Barabalet's lemma, one can show that for the given weight update law:

$$\dot{W} = -\gamma\phi(x)e^T PB \tag{4.18}$$

where $\gamma > 0$ is the adaptive gain, P>0 is a solution to the Lyapunov equation: $A_{rm}^T P + PA_{rm} + Q = 0$ for any Q>0, all signals in the closed loop are bounded for the above type of uncertainity.

4.3.3 Deep Model Reference Adaptive Control

DMRAC is a new neuroadaptive based controller that incorporates deep learning within the MRAC framework. It is a learning based controller that combines advantages of deep nets at representing complex non linearities with stability guarantees associated with MRAC. Details regarding stability proof and uniform ultimate boundedness of DMRAC under unstructured uncertainity is provided in [13, 14]. The general architecture for this controller

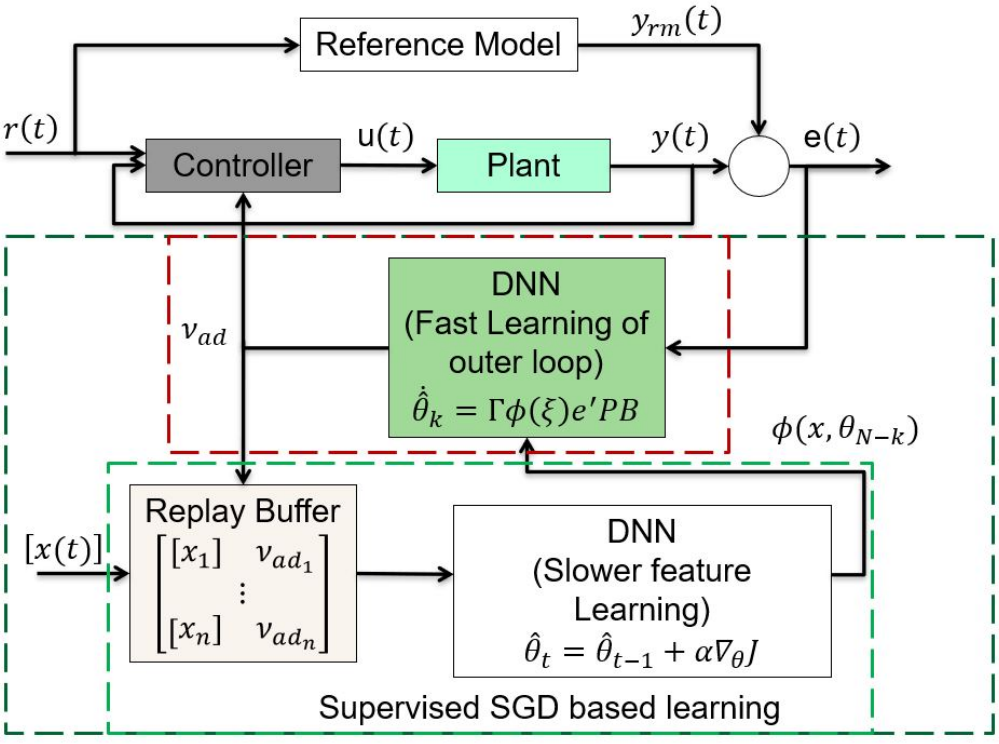

Figure 4.3: Architecture of DMRAC

is given in figure 4.3. The entire controller can be split into 3 main modules: the DNN (Fast Learning Outer Loop) module, replay buffer module and DNN (Slower feature learning) module. DNN(Fast Learning Outer Loop) module is where a standard MRAC control algorithm is used with basis function $\phi(x)$. Its output $\Delta'(X)$ (the estimated uncertainity) is stored together with the state at that time inside a replay buffer as a datapoint. This buffer is of a fixed size and a kernel independence test is done to ensure only those datapoints are retained that give a sufficiently rich representation of operating domain, once buffer's capacity is reached. Random batches of data are drawn from this replay buffer to train a neural network using stochastic gradient descent method, which maps state to estimated uncertainity values. The basis function $\phi(x)$ is computed as the output of the second last layer of this trained neural network by doing a forward pass using state value at current time and the information is then passed onto DNN(Fast Learning Outer Loop) module which completes the loop.

Concisely, DMRAC algorithm can be stated as follows:

Algorithm 1 DMRAC Controller Training

Input: $\gamma, \eta, \zeta_{tol}, p_{max}$ {γ=Adaptive gain,η=SGD learning rate, ζ_{tol}=Kernel independence test coefficient[20], p_{max}=Max buffer size }
while New measurements are available **do**
  Update the DMRAC weights W using Eq:(4.18)
  Compute $y_{\tau+1} = \hat{W}^T \Phi(x_{\tau+1})$
  Given $x_{\tau+1}$ compute $\gamma_{\tau+1}$ [20].
  if $\gamma_{\tau+1} \geq \zeta_{tol}$ **then**
    Update $\mathcal{B} : \boldsymbol{Z}(:) = \{x_{\tau+1}, y_{\tau+1}\}$ and $X : \Phi(x_{\tau+1})$
    if $|\mathcal{B}| > p_{max}$ **then**
      Delete element in $\mathcal{B}$ by SVD maximization [20]
    end if
  end if
  if $|\mathcal{B}| \geq M$ **then**
    Sample a mini-batch of data $\boldsymbol{Z}^M \subset \mathcal{B}$
    Train the DNN network over mini-batch data using SGD
    Update the feature vector Φ for D-MRGeN network
  end if
end while

In DMRAC, the key idea is that training of neural network is based on the output labels generated by MRAC. This is essential since apriori, there

is no information about the actual disturbance values. Hence, MRAC which is operating in the fast learning outer loop module is acting as a generative network. It produces estimates of uncertainities which are samples from the same distribution as of actual disturbance[21]. Another important aspect is that the time scales of neural network training and weight learning using MRAC weight update rule are different. The weight updates are performed in real time whereas the neural network training is done after collecting some samples of state-estimated uncertainity data. During successive training iterations of the neural network, the basis $\phi(x)$ provided by the neural network is used as the fixed feature vector for the MRAC weight update rule.

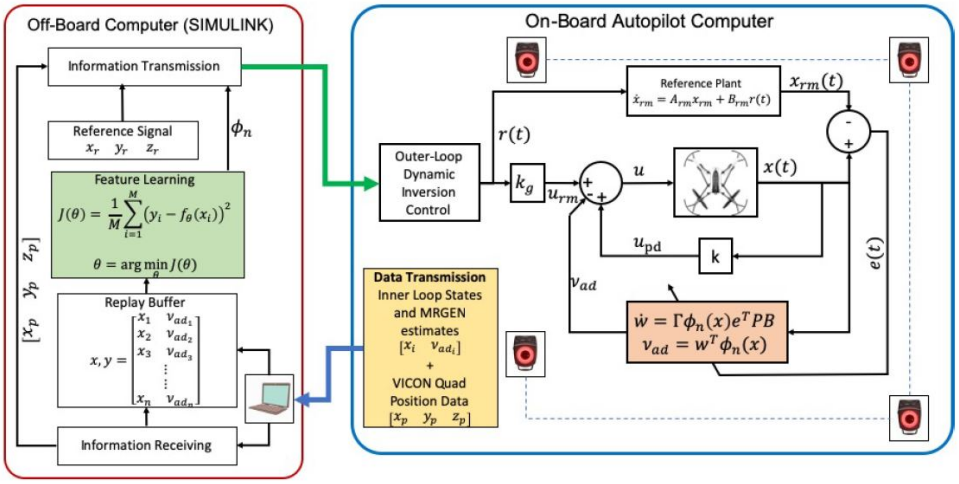

Figure 4.4: On-board - Off-board Implementation of Deep Model reference Adaptive controller for Quadrotor control.

On the quadrotor, implementation of DMRAC was done in an offline-online manner as shown in Figure 4.4. This was necessary as the onboard computational memory was limited and hence, running entire controller onboard was not possible. The offline part was run on a computer whereas the online part was run completely on the drone. The offline part comprised of replay buffer and DNN (slower feature learning) module whereas the online part had DNN(Fast Learning of outer loop) module. The communication between the drone and the computer was done using UDP protocol. The information sent by the drone is labelled state-adaptive torque data which is used for training of the neural network. After training is completed on the computer and a new basis $\phi(x)$ is evaluated, this information is sent back to the drone by the computer.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Introduction

In this section, flight test results compiled by running different control algorithms on a quadrotor inside Vicon arena are presented. The results are divided into 3 main sections namely a) Performance comparison between PID, MRAC and DMRAC, b) Generalisability of DMRAC and c) Evaluating Transfer Learning with DMRAC.

5.2 Performance comparison between PID, MRAC and DMRAC

In this section, results are presented that compare and contrast the performance of the DMRAC algorithm over control algorithms such as MRAC and PID in a variety of different flight operating conditions. The experiments were done under different amounts of wind bias, when rotor chipping occurs during mid flight etc.

5.2.1 Flight test results on a figure of 8 trajectory (Base Case)

In this experiment, each controller's performance is evaluated on tracking a figure of 8 reference trajectory without any disturbance. The feedback, feedforward gains, learning rates etc are tuned on this case to ensure each controller performs equally well. The parameters obtained from this experiment are kept fixed throughout the remainder of the other experiments. Fig-5.1 and Fig-5.2 show the comparison between each controllers' performance on the nominal baseline task with no external disturbance or faults.

Since each controller is tuned to achieve best performance over the given task, the difference between the controller is negligible, and all three controllers perform equally well.

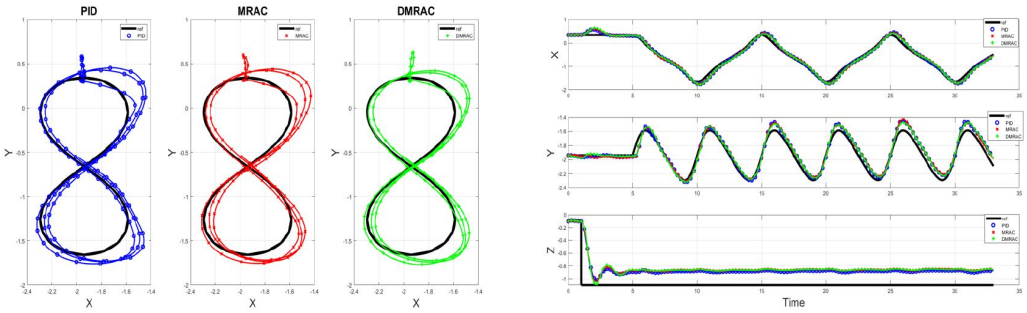

Figure 5.1: Tracking performance on a simple figure of 8 trajectory

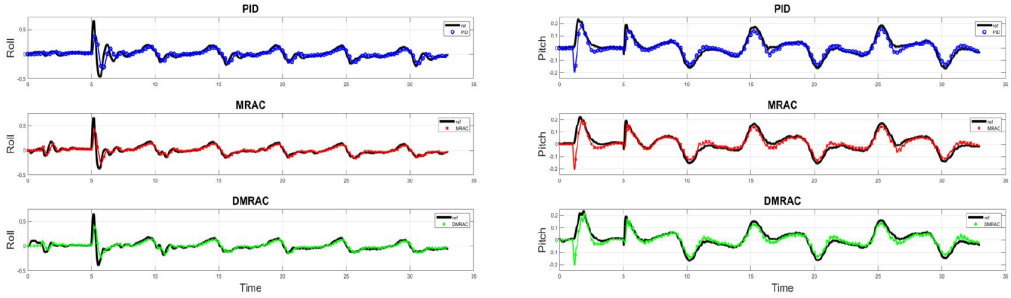

Figure 5.2: Tracking of reference model's roll and pitch signal for a figure of 8 trajectory

5.2.2 Reference trajectory tracking with wind bias

The second task used to evaluate the controllers' performance is reference tracking on a figure of 8 trajectory with wind bias. This task is designed to assess the performance of all three controllers in the case of external disturbance. A wind bias disturbance is simulated, using a fan placed near the drone's initial position, and oriented to cause the disturbance along the X-axis. Fig-5.3 and Fig-5.4 show a comparison of each controllers' performance on this task. It can be seen that DMRAC is more robust and achieves much better tracking compared to other two algorithms. The tracking error for the DMRAC controller for the inner loop reference roll and pitch states is also much lower compared to MRAC or PID refer Figure-5.4. The results also

clearly demonstrate that the proposed controller can handle abrupt changes in reference commands. At around 5 secs mark, the reference signal changes from step input for height control in the z-axis to figure of 8 trajectory command in the x-y plane. The PID controller experiences high oscillation, whereas MRAC and DMRAC handle the switch much more smoothly. Further the above experiment is repeated with medium and high wind bias. The following results on a tracking task under external disturbance demonstrates that DMRAC outperforms MRAC and PID refer Figure-5.5-5.8.

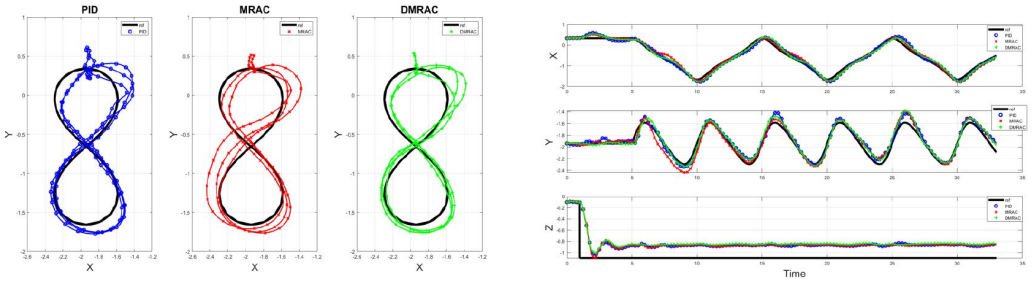

Figure 5.3: Tracking performance under low wind bias

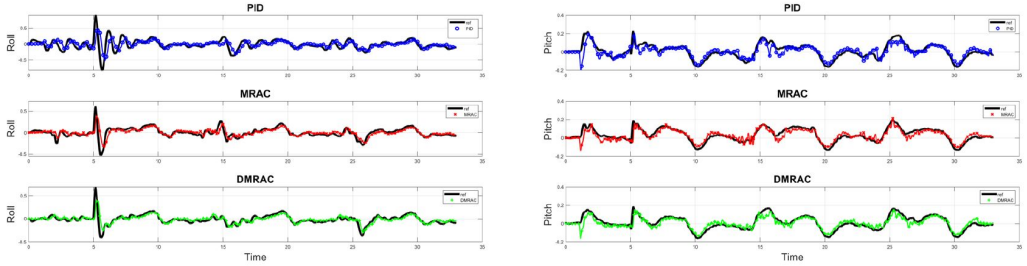

Figure 5.4: Tracking of reference model's roll and pitch signal under low wind bias

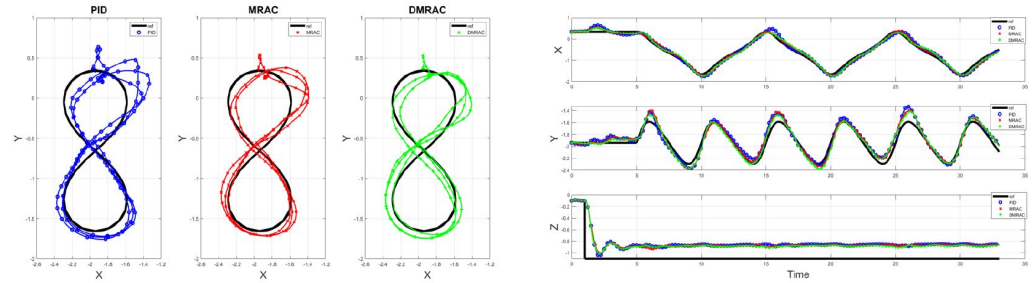

Figure 5.5: Tracking performance under medium wind bias

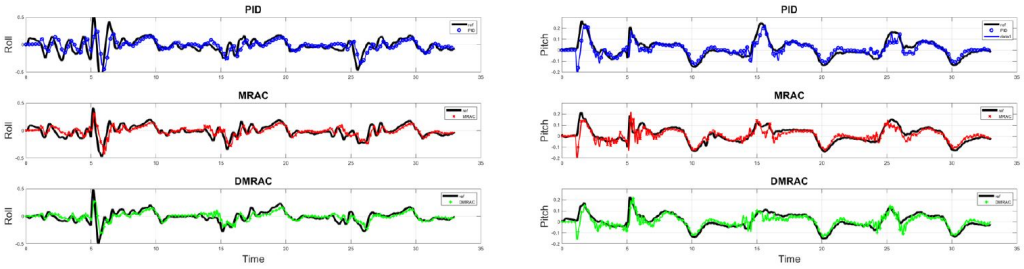

Figure 5.6: Tracking of reference model's roll and pitch signal under medium wind bias

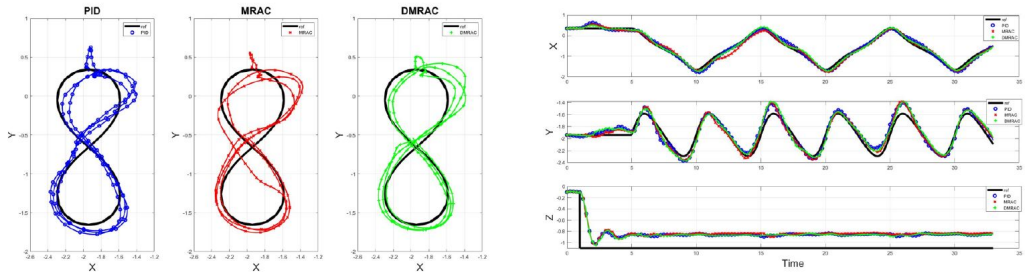

Figure 5.7: Tracking performance under high wind bias

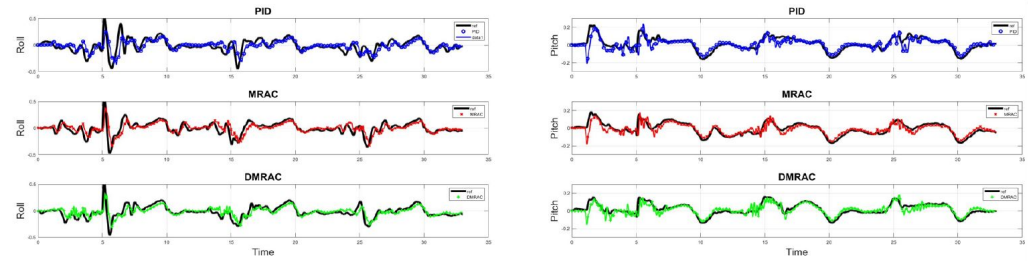

Figure 5.8: Tracking of reference model's roll and pitch signal under high wind bias

5.2.3 Reference trajectory tracking under a highly nonlinear disturbance

In this experiment, to simulate a highly nonlinear and unpredictable disturbance, a piece of cloth is attached underneath the frame of the quadrotor and the entire setup is subjected to high wind bias. This causes an erratic flapping of the cloth which produces unpredictable disturbance torques and forces. The experiment was designed to push each controller to its limits and was repeated three times to demonstrate repeatability. It was observed that PID failed in all the three experiments, whereas both MRAC and DM-

RAC gave a stable performance. However, the tracking error observed for MRAC was relatively higher compared to DMRAC. Below, results for the best case tracking performance observed for all the three controllers is presented. Figures-5.9 and Fig-5.10 clearly show that PID fails at around the end of the flight with high oscillations, whereas we observe DMRAC tracking under severe disturbance forces appears to be the best followed by MRAC. Similar results are provided for circular reference trajectory with high wind bias. We observe the PID fails very early in the flight. However adaptive controllers are successful in completing the task, we observe DMRAC outperforms the MRAC with much tighter tracking. Refer Fig:5.11-5.12. The Fig-5.13 plots the control torques generated in Roll and pitch to achieve the trajectory tracking.

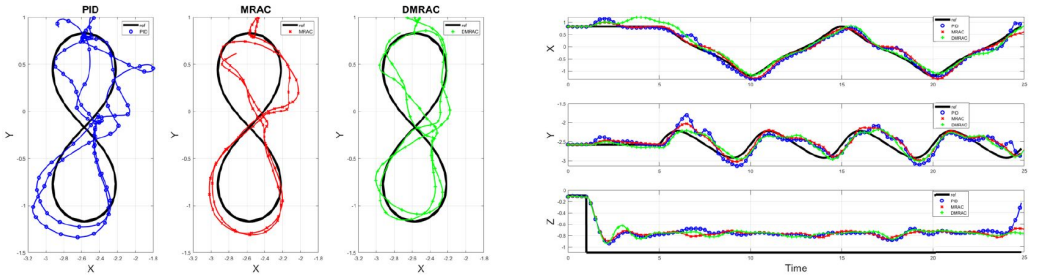

Figure 5.9: Tracking performance under high wind bias with cloth attached underneath the quadcopter

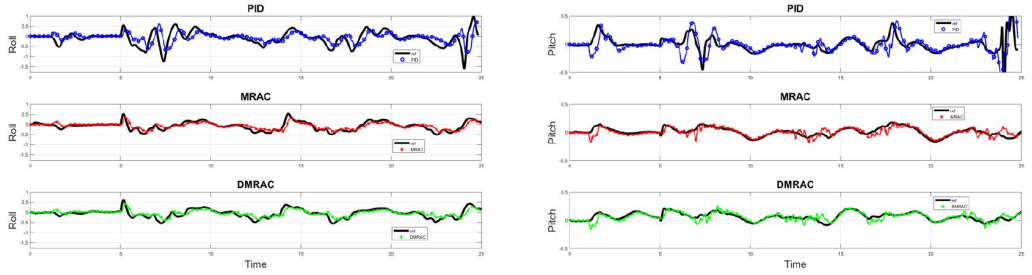

Figure 5.10: Tracking of reference model's roll and pitch signal under high wind bias with cloth attached underneath

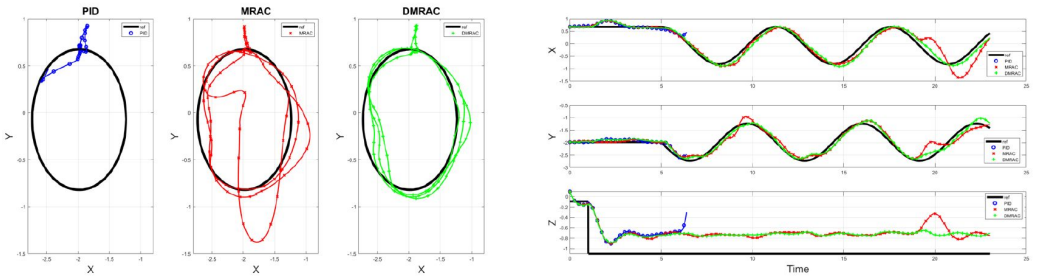

Figure 5.11: Tracking performance for a circular trajectory under high wind bias with cloth attached on drone

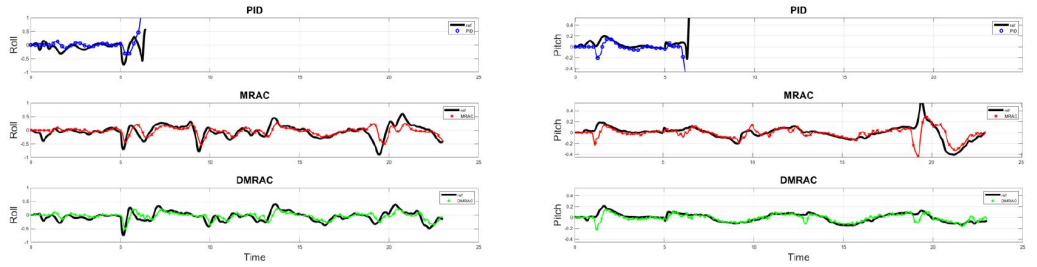

Figure 5.12: Tracking of reference model's roll and pitch signal under high wind bias

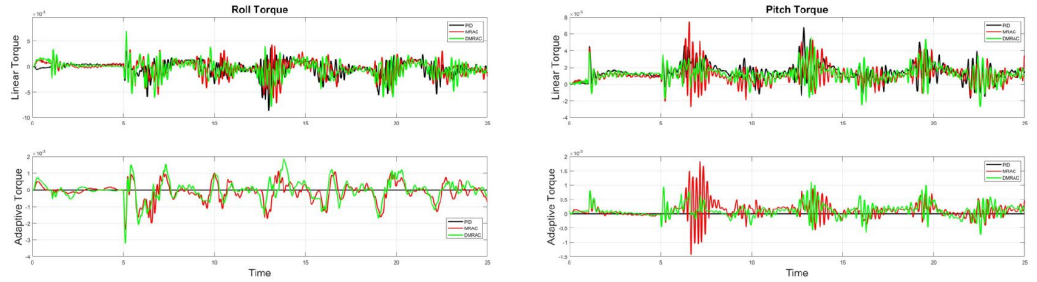

Figure 5.13: Linear and adaptive control torque for PID, MRAC and DMRAC

5.2.4 Fault tolerance: Rotor blade chipping in mid-flight

In these experiments, fault-tolerance capability of the controllers is tested in case of rotor chipping during mid-flight. One of the rotor blades is cut in half and is attached back using tape, as shown in Fig-1.1. The quadrotor is commanded to hover at 1m above the ground. Due to centrifugal forces, the chipped blade breaks off and causes the fault into the system at an undertermined time. Since this is not a controlled fault,in order to ensure the reliability of the results, each controller is made to perform on the above

task over multiple runs. The results presented in Fig- 5.15 clearly shows that DMRAC outperforms PID and MRAC. In the case of PID, only two runs were carried out, since in both cases, the drone underwent severe oscillation and crashed. Tests conclusively demonstrated that PID is not capable of handling sever faults in the system even with extensive tuning. In the case of MRAC and DMRAC, eight flight tests are carried out. Out of eight test runs, failures were seen twice for MRAC, whereas no failure were observed in the case of DMRAC. Also, on comparing flights where no crash occurs, one can see that MRAC produces poor reference tracking when compared to DMRAC. Refer Figure-5.14 and 5.15 for more detailed results. The figures show mean and variance plots for reference tracking in the x-y-z position for each algorithm.

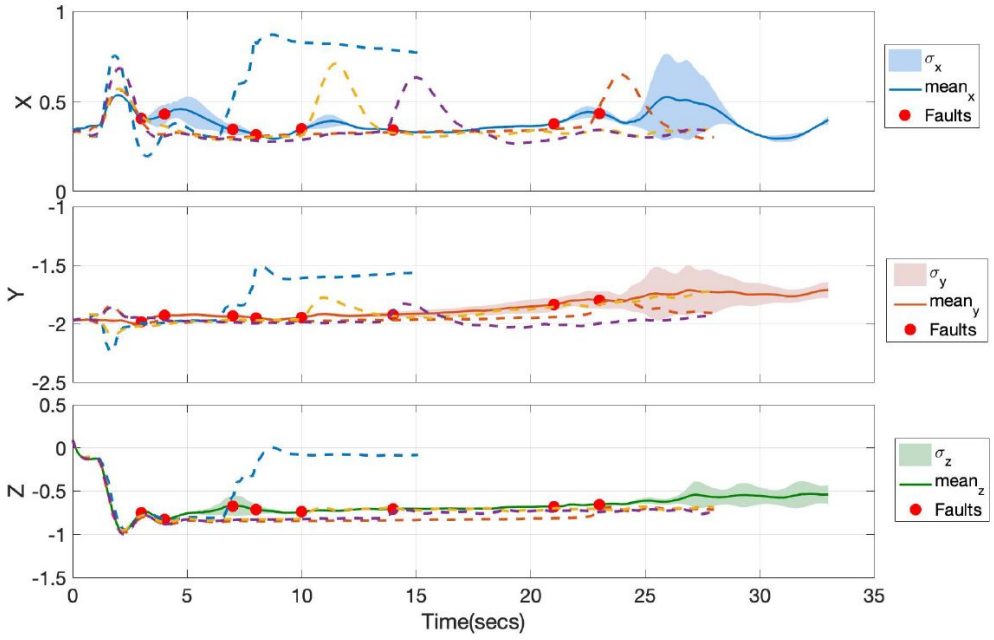

Figure 5.14: MRAC Trajectory tracking performance in X-Y-Z under system fault for eight flight test. Out of eight flights we observe four times the quadrotor either crashed or produced bad tracking (Red dot: Time at which Fault occurred)

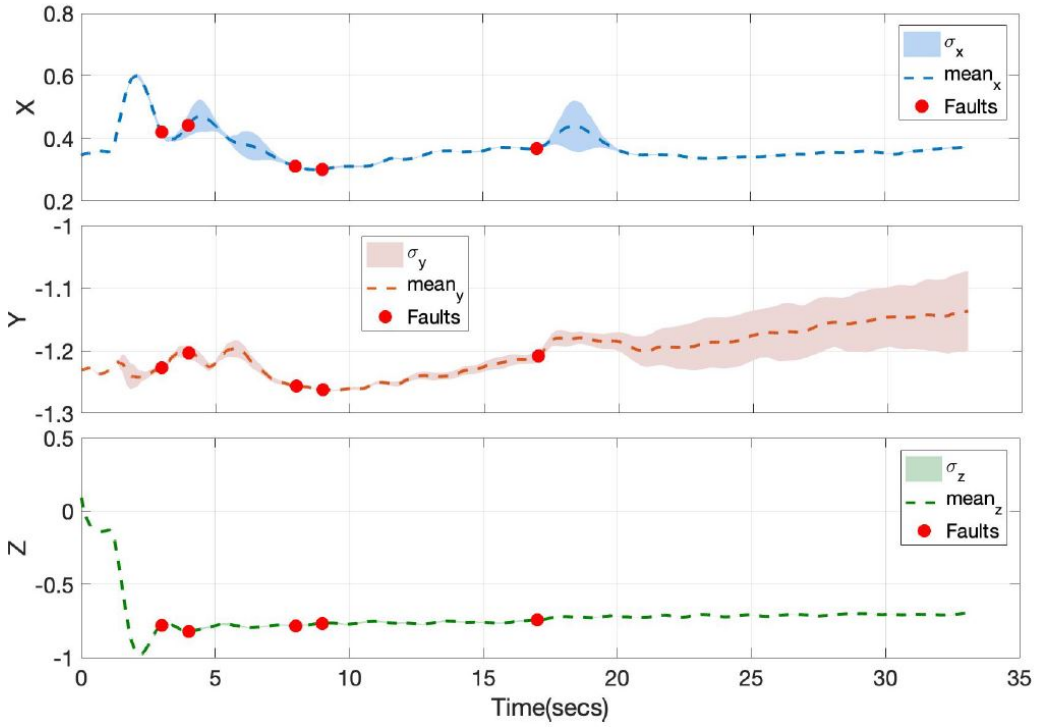

Figure 5.15: DMRAC Trajectory tracking performance in X-Y-Z under system fault for Eight flight test (Red dot: Time at which Fault occurred)

5.3 Generalisability of DMRAC

In this section, results related to learning retention due to Deep architecture in model reference adaptive controller are presented. This experiment aims to test the memory associated with deep neural networks in the context of an adaptive controller, i.e, generalizing capability of DMRAC. Initially, DMRAC is trained on the labeled pair of input and output data generated using model reference generative network. The trained network is then used as a feed-forward function approximator to estimate the adaptive control for reference tracking in a new but similar task. In this experiment, training of the DMRAC neural network is performed on flight data collected from the quadrotor going in circles both clockwise and anti-clockwise with and without wind bias. The test case is that the quadrotor is made to fly in a figure of 8 with and without wind disturbance which is an unseen task and has not been used in the training process. Further, in order to test this controller's performance, comparison is made against a tuned PID controller on the same test case. The main hypothesis is that since the DMRAC controller

is trained over clockwise(cw) and anti-clockwise(ccw) trajectories, this controller should be able to generalize to any trajectory formed combining the cw and ccw turns. Since the DMRAC neural network weights are trained off-line before the test flight, the entire controller is hardcoded onto the onboard computer. The parameters, such as PID gains, learning rate etc are kept unchanged from previous experiments.

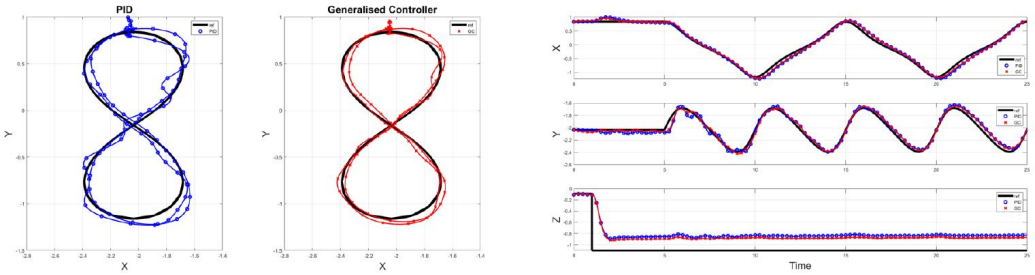

Figure 5.16: DMRAC generalizing without active learning: Tracking performance on a figure of 8 trajectory

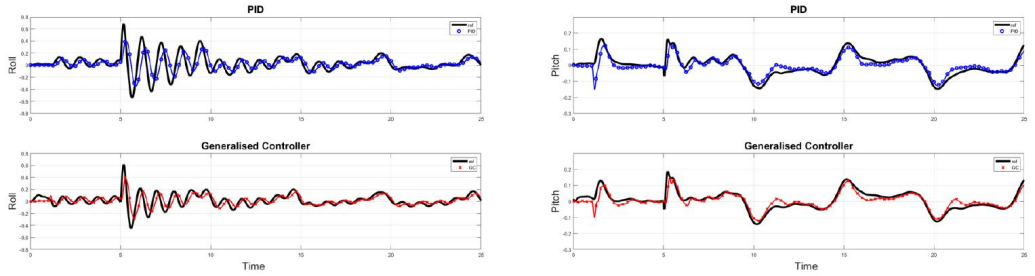

Figure 5.17: DMRAC generalizing without active learning: Tracking of reference model's roll and pitch for a figure of 8 reference trajectory

In Figure:5.16-5.17, one can observe that DMRAC controller generalizes well to a previously unseen reference signal. On comparing to the PID controller's performance on the same task, one can see that the tracking is much better as well as the oscillation in roll is much lower than PID. Thereby, DMRAC not only generalizes well but also proves to be robust. In Figure:5.18-5.19, generalization of DMRAC is tested in the windy case. Here, one can see that the oscillation observed for the generalized controller in tracking is far lower than PID, and it gives much better tracking overall. These experiments demonstrate clearly that DMRAC retains the memory of both windy and non-windy cases in form of deep features, and can counter both wind and no wind cases reasonably well even without active online adaptation

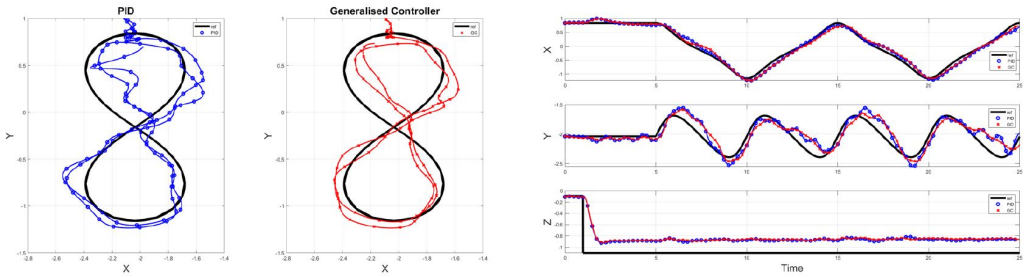

Figure 5.18: DMRAC generalizing without active learning: Controller performance for clockwise tracking of figure of 8 trajectory under wind bias

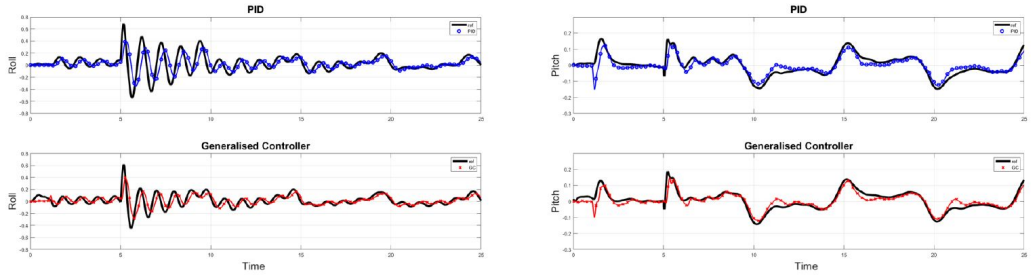

Figure 5.19: DMRAC generalizing without active learning: Tracking of reference model's roll and pitch for clockwise tracking of figure of 8 under wind bias

5.4 Evaluating Transfer Learning with DMRAC

Lately, Transfer learning (TL) has been a much-researched topic in machine learning and reinforcement learning. In similar lines in these experiments, the aim is to test the advantages of representation transfer in an adaptive control setting. Here, transfer learning is tested through sharing network parameters between tasks. TL is performed by first running DMRAC on related tasks and learning the network weights, which incorporate some feature knowledge. These learned weights are then used to initialize a fresh DMRAC network executing a new unseen task. In these experiments, flight test of a drone executing a basic figure of 8 trajectory is used as a source task for representation transfer through the warm-start of the networks. The target task is an unseen but related task for which an initialized network is used for the drone executing figure of 8 trajectories under high wind bias, refer Fig-5.20. A clear improvement of controller performance in achieving smaller transients and better steady state tracking is observed with warm-started DMRAC. The deep network weights learnt over the quadrotor executing figure of 8 trajec-

tories encodes the feature knowledge about modeling uncertainties. When this learning is transferred to a new drone executing figure of 8 with wind bias, it is able to adapt faster and also quickly learn features corresponding to wind bias.

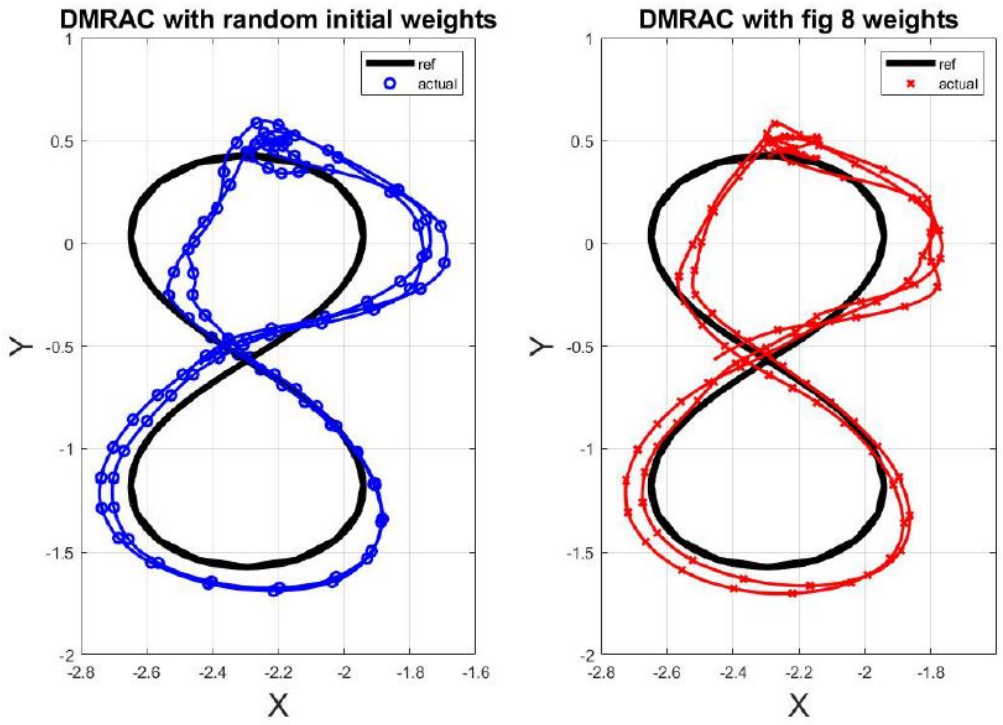

Figure 5.20: Figure of 8 Trajectory tracking under wind bias with random initialization vs. Feature transfer in DMRAC

5.5 Simulation to Real-World Transfer Learning

The following experiments are similar to one in the previous section. Here, network representation transfer from simulation to the real world is investigated. In this experiment, DMRAC is run in a simulation environment, where the network is trained over data collected through the simulated drone follow a figure of 8 trajectory without any disturbance. These trained network weights are then used as initialization weights for the case where DMRAC is experimented on the actual physical quadrotor. The controller performance is compared between the randomly initialized DMRAC vs. DMRAC initialized with network weights from the simulated quadrotor. The two controllers

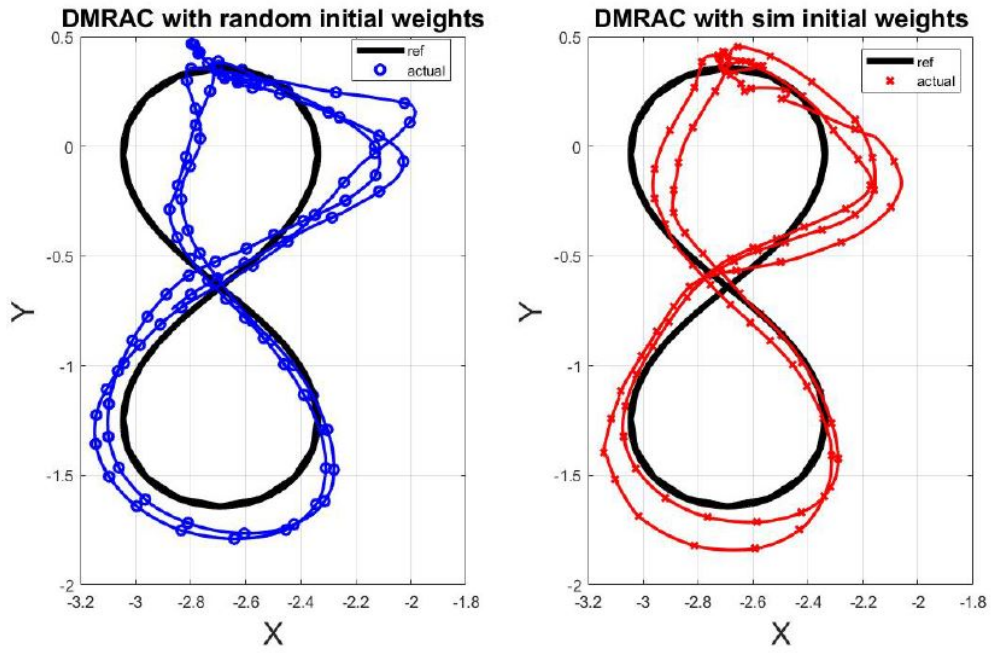

Figure 5.21: Figure of 8 Trajectory tracking under wind bias with random initialization vs. Feature transfer from simulation to Real in DMRAC

are tested on performing a figure of 8 trajectory maneuver under high wind bias. Figure-5.21 shows the improvement in DMRAC's performance when the initial weights are from simulation rather than being initialized entirely randomly.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, successful implementation of DMRAC adaptive controller using model reference generative network architecture was shown on an actual quadrotor. It was demonstrated that the fast-slow architecture utilizing asynchronous onboard and off-board processing can be used to incorporate deep learning in the closed-loop for high-bandwidth flight control of unstable aircraft in the presence of significant disturbances. The results clearly show that when utilized in the closed loop in this manner DMRAC can provide significant performance and generalization benefits over shallow MRAC and PIDs. The results obtained are significant, not only for flight control, but for other robotic control applications involving deep learning as well. This is because this approach of separating the learning in asynchronous manner can be adopted to other learning based controllers, including learning based MPC and reinforcement learning.

As far as possible directions of future work are concerned, testing of this adaptive controller on bigger quadrotors outside the Vicon arena could be a good step in further validating it's real world performance. Moreover, a different test platform like fixed wing aircrafts with significant wing damage could be used to show how this controller would perform in different type of safety critical application. This, in fact, would be a good test to prove whether DMRAC is ready to be put on-board an actual aircraft for autonomous flights. Further, notion of transfer learning from quadrotor experiences to fixed wing aircraft could be explored. Another area of improving DMRAC could be using contextual information along with the system states to extract relevant features. This contextual information could be relevant model information not captured in system states. An example of this in aircrafts could be parameters like angle of attack, engine thrust etc. These states can extract features which can help in decision making in case of faults.

CHAPTER 7

REFERENCES

[1] P. Ioannou and J. Sun, "Theory and design of robust direct and indirect adaptive-control schemes," *International Journal of Control*, vol. 47, no. 3, pp. 775–813, 1988.

[2] G. Tao, *Adaptive control design and analysis*. John Wiley & Sons, 2003, vol. 37.

[3] J.-B. Pomet and L. Praly, "Adaptive nonlinear regulation: estimation from the Lyapunov equation," *Automatic Control, IEEE Transactions on*, vol. 37, no. 6, pp. 729–740, 6 1992.

[4] E. N. Johnson and S. Kannan, "Adaptive Trajectory Control for Autonomous Helicopters," *Journal of Guidance Control and Dynamics*, vol. 28, no. 3, pp. 524–538, 5 2005. [Online]. Available: http://doi.aiaa.org/10.2514/1.6271

[5] G. Chowdhary, M. Mühlegg, and E. N. Johnson, "Exponential parameter and tracking error convergence guarantees for adaptive controllers without persistency of excitation," *International Journal of Control*, vol. 87, no. 8, pp. 1583–1604, 2014. [Online]. Available: http://dx.doi.org/10.1080/00207179.2014.880128

[6] R. Grande, G. Chowdhary, and J. P. How, "Experimental Validation of Bayesian Nonparametric Adaptive Control using Gaussian Processes," *AIAA Journal of Aerospace Information Systems (formerly JACIC)*, 1 2014.

[7] I. Gregory, C. Cao, E. Xargay, N. Hovakimyan, and X. Zou, "L1 adaptive control design for nasa airstar flight test vehicle," in *AIAA guidance, navigation, and control conference*, 2009, p. 5738.

[8] G. Chowdhary, E. N. Johnson, R. Chandramohan, M. S. Kimbrell, and A. Calise, "Guidance and control of airplanes under actuator failures and severe structural damage," *Journal of Guidance, Control, and Dynamics*, vol. 36, no. 4, pp. 1093–1104, 2013.

[9] Z. Zuo and P. Ru, "Augmented l 1 adaptive tracking control of quadrotor unmanned aircrafts," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, no. 4, pp. 3090–3101, 2014.

[10] G. Chowdhary, H. A. Kingravi, J. P. How, and P. A. Vela, "Bayesian Nonparametric Adaptive Control Using Gaussian Processes," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2014.

[11] B. Michini and J. How, "L1 adaptive control for indoor autonomous vehicles: Design process and flight testing," in *AIAA Guidance, Navigation, and Control Conference*, 2009, p. 5754.

[12] G. Chowdhary and E. N. Johnson, "Theory and Flight Test Validation of a Concurrent Learning Adaptive Controller," *Journal of Guidance Control and Dynamics*, vol. 34, no. 2, pp. 592–607, 3 2011.

[13] G. Joshi and G. Chowdhary, "Deep model reference adaptive control," 2019.

[14] G. Joshi, J. Virdi, and G. Chowdhary, "Design and flight evaluation of deep model reference adaptive controller," in *AIAA Scitech 2020 Forum*, 2020, p. 1336.

[15] Randal.W.Beard, "Quadrotor dynamics and control," *BYU Scholars Archive*, 2008.

[16] "Accessing vicon tracker data from simulink," Software development kit can be found at https://docs.vicon.com/display/Tracker33/Access +Vicon+Tracker+ data+ from+Simulink (2020/04/15).

[17] Sertac Karaman and Fabian Riether, "Rosmat: Rolling spider matlab toolbox," https://github.com/Parrot-Developers/RollingSpiderEdu/tree/master/MIT_MatlabToolbox, starting guide for Parrot Mambo mini drone is also available at https://www.mathworks.com/help/supportpkg/parrot.

[18] F. L. Lewis, "Nonlinear Network Structures for Feedback Control," *Asian Journal of Control*, vol. 1, pp. 205–228, 1999.

[19] R. M. Sanner and J.-J. Slotine, "Gaussian networks for direct adaptive control," *Neural Networks, IEEE Transactions on*, vol. 3, no. 6, pp. 837–863, 11 1992.

[20] G. Chowdhary and E. Johnson, "A singular value maximizing data recording algorithm for concurrent learning," in *Proceedings of the 2011 American Control Conference*, June 2011, pp. 3547–3552.

[21] G. Joshi and G. Chowdhary, "Adaptive control using gaussian-process with model reference generative network," in *2018 IEEE Conference on Decision and Control (CDC)*. IEEE, 2018, pp. 237–243.

APPENDIX A

PID CODE FOR QUADCOPTER CONTROL

The PID quadrotor control class is given as follows:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 30 00:48:05 2020

@author: Jasvir
"""
import math

class PID_Control():
    def __init__(self):
        # state_vec is (X,Y,Z,yaw,pitch,roll,dx,dy,dz,p,q,r)
        # (x,y,z,yaw,pitch,roll,dx,dy,dz) in earth frame
        # (p,q,r) in body frame
        self.delt = 0.005 # time step
        self.mass = 0.063
        self.g = 9.81

        self.P_yaw = 0.004 # Parrot Mambo
        self.D_yaw = 0.3*0.004;

        self.P_pitch = 0.013
        self.I_pitch = 0.01
        self.D_pitch = 0.002

        self.P_roll = 0.01
        self.I_roll = 0.01
        self.D_roll = 0.0028

        self.P_x = -0.44
        self.I_x = 0
        self.D_x = -0.35

```

```
33          self.P_y = -0.44  # when P_y at -0.9, PID does not
    work, unstable (fig_8)
34          self.I_y = 0      # MRAC gives stable results with
    tracking which
35          self.D_y = -0.35 # improves over time, gain =0.005
36
37          self.P_z = 0.8
38          self.D_z = 0.3
39
40          self.integration_val_pitch = 0
41          self.integration_val_roll = 0
42          self.integration_val_x = 0
43          self.integration_val_y = 0
44
45
46      def assign_states(self,state):
47          self.X, self.Y, self.Z = state[0], state[1], state[2]
48          self.yaw, self.pitch, self.roll = state[3], state[4],
     state[5]
49          self.dx, self.dy, self.dz = state[6], state[7], state
    [8]
50          self.p, self.q, self.r = state[9], state[10], state
    [11]
51
52
53      def integral(self,prev_integral_value,error):
54          # 0.001 is antiwindup gain
55  #         if prev_integral_value>=2: # For modeling saturation
56  #             prev_integral_value=2
57  #         elif prev_integral_value<=-2:
58  #             prev_integral_value=(-2)
59          error_anti_windup = error-0.001*prev_integral_value
60          integral_new = prev_integral_value +
    error_anti_windup*self.delt
61          return integral_new
62
63      def PID_control(self, Kp, Ki, Kd, error, int_error,
    d_error):
64          PID_control_value = Kp*error + Ki*int_error - Kd*
    d_error
65          return PID_control_value
66
67      def outer_loop_control(self,ref_traj):
```

```
68          # ref_traj is (Xref, Yref, Zref, Yaw_ref)
69          Xref, Yref, Yaw_ref = ref_traj[0], ref_traj[1],
    ref_traj[3]
70
71          error_x = Xref-self.X
72          self.integration_val_x = self.integral(self.
    integration_val_x,error_x)
73
74          error_y = Yref-self.Y
75          self.integration_val_y = self.integral(self.
    integration_val_y,error_y)
76
77          PID_x = self.PID_control(self.P_x, self.I_x, self.D_x
    , error_x, self.integration_val_x, self.dx)
78          PID_y = self.PID_control(self.P_y, self.I_y, self.D_y
    , error_y, self.integration_val_y, self.dy)
79
80          pitch_ref = PID_x*math.cos(self.yaw)+PID_y*math.sin(
    self.yaw) # Approx Model Inversion
81
82          roll_ref = PID_x*math.sin(self.yaw)-PID_y*math.cos(
    self.yaw) # Approx Model Inversion
83
84          return [Yaw_ref, pitch_ref, roll_ref]
85
86
87      def Thrust_force(self,ref_traj):
88
89          Zref= ref_traj[2]
90          error_z = Zref-self.Z
91
92          Thrust_force_PID = self.PID_control(self.P_z, 0, self
    .D_z, error_z, 0, self.dz)
93          #Thrust_force_total = (-self.mass*self.g +
    Thrust_force_PID)/(math.cos(self.pitch)*math.cos(self.roll
    ))
94
95          return Thrust_force_PID
96
97
98      def inner_loop_control(self,yaw_pitch_roll_ref):
99
100         error_yaw = yaw_pitch_roll_ref[0]-self.yaw
```

```
101
102         error_pitch = yaw_pitch_roll_ref[1]-self.pitch
103         self.integration_val_pitch = self.integral(self.
    integration_val_pitch,error_pitch)
104
105         error_roll = yaw_pitch_roll_ref[2]-self.roll
106         self.integration_val_roll = self.integral(self.
    integration_val_roll,error_roll)
107
108         torque_yaw = self.PID_control(self.P_yaw, 0, self.
    D_yaw, error_yaw, 0, self.r)
109         torque_pitch = self.PID_control(self.P_pitch, self.
    I_pitch, self.D_pitch, error_pitch, self.
    integration_val_pitch, self.q)
110         torque_roll = self.PID_control(self.P_roll, self.
    I_roll, self.D_roll, error_roll, self.integration_val_roll
    , self.p)
111
112         return [torque_roll, torque_pitch, torque_yaw]
```

In order to test the above code, following test script can be used:

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Tue Mar 31 15:57:34 2020
4
5  @author: me112
6
7  Test Script
8  """
9  from PID_control import PID_Control
10 import matplotlib.pyplot as plt
11 import numpy as np
12 import math
13
14 def drone_dynamics_with_control():
15     A = PID_Control()
16     A.assign_states([0,0,0,0,0,0,0,0,0,0,0,0])
17     time_step=A.delt
18     u,v,w = 0,0,0
19     Jx = 0.0000582857
20     Jy = 0.0000716914
21     Jz = 0.0001
22
```

```
    X_vals=[]
    Y_vals=[]
    Z_vals=[]
    ref_traj_x=[]
    ref_traj_y=[]

    T_fig_8 = 20

    for time in range(1,5000): #Number of time steps

        ref_traj = [math.cos(2*math.pi*time*0.005/T_fig_8),
    math.sin(4*math.pi*time*0.005/T_fig_8),-2,0] # Xref(t),
    Yref(t),Zref(t),Yawref(t)

        [Yaw_ref, pitch_ref, roll_ref] = A.outer_loop_control
    (ref_traj)

        Thrust = A.Thrust_force(ref_traj)

        [torque_roll, torque_pitch, torque_yaw] = A.
    inner_loop_control([Yaw_ref, pitch_ref, roll_ref])

        theta = A.pitch
        psi = A.yaw
        phi = A.roll

        first_row = [math.cos(theta)*math.cos(psi), \
                     math.sin(phi)*math.sin(theta)*math.cos(
    psi)-math.cos(phi)*math.sin(psi),\
                     math.cos(phi)*math.sin(theta)*math.cos(
    psi)+math.sin(phi)*math.sin(psi)]

        second_row = [math.cos(theta)*math.sin(psi), \
                     math.sin(phi)*math.sin(theta)*math.sin(
    psi)+math.cos(phi)*math.cos(psi),\
                     math.cos(phi)*math.sin(theta)*math.sin(
    psi)-math.sin(phi)*math.cos(psi)]

        third_row = [-math.sin(theta), \
                     math.sin(phi)*math.cos(theta),\
                     math.cos(phi)*math.cos(theta)]

```

```
57          R_body_to_veh = np.array([first_row,second_row,
    third_row])
58
59
60          ## x_(t+1)=f(x_t,u_t)
61          [X_next] = np.array(A.X) + time_step*(np.dot(
    first_row,[[u],[v],[w]]))
62          [Y_next] = np.array(A.Y) + time_step*(np.dot(
    second_row,[[u],[v],[w]]))
63          [Z_next] = np.array(A.Z) + time_step*(np.dot(
    third_row,[[u],[v],[w]]))
64
65          u_next = u + time_step*(A.r*v-A.q*w - 9.81*math.sin(
    theta))
66          v_next = v + time_step*(A.p*w-A.r*u + 9.81*math.cos(
    theta)*math.sin(phi))
67          w_next = w + time_step*(A.q*u-A.p*v + 9.81*math.cos(
    theta)*math.cos(phi) + Thrust/A.mass)
68
69          phi_next = phi + time_step*(A.p + A.q*math.sin(phi)*
    math.tan(theta) + A.r*math.cos(phi)*math.tan(theta))
70          theta_next = theta + time_step*(A.q*math.cos(phi) - A
    .r*math.sin(phi))
71          psi_next = psi + time_step*((A.q*math.sin(phi) + A.r*
    math.cos(phi))/math.cos(theta))
72
73          p_next = A.p + time_step*(((Jy-Jz)/Jx)*A.q*A.r +
    torque_roll/Jx)
74          q_next = A.q + time_step*(((Jz-Jx)/Jy)*A.p*A.r +
    torque_pitch/Jy)
75          r_next = A.r + time_step*(((Jx-Jy)/Jz)*A.p*A.q +
    torque_yaw/Jz)
76
77
78          [dx_next],[dy_next],[dz_next] = np.dot(R_body_to_veh
    ,[[u_next],[v_next],[w_next]])
79
80          u,v,w = u_next,v_next, w_next
81          A.assign_states([X_next, Y_next, Z_next, psi_next,
    theta_next, phi_next, dx_next, dy_next,dz_next ,p_next,
    q_next, r_next])
82
83          X_vals.append(A.X)
```

```
84          Y_vals.append(A.Y)
85          Z_vals.append(A.Z)
86          ref_traj_x.append(ref_traj[0])
87          ref_traj_y.append(ref_traj[1])
88
89      plt.plot(Y_vals,X_vals)
90      plt.plot(ref_traj_y,ref_traj_x,'k')
91
92
93  drone_dynamics_with_control()
```

APPENDIX B

MRAC CODE FOR QUADCOPTER CONTROL

Note MRAC controller incorporates the above PID control class:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 30 00:48:05 2020

@author: Jasvir
"""
import numpy as np
from PID_control import PID_Control

class MRAC_Control(PID_Control):
    def __init__(self):
        PID_Control.__init__(self)
        self.ref_model_states = np.array
    ([[0],[0],[0],[0],[0],[0]])
        self.BW = 2
        self.number_centers = 25
        self.adaptive_gain = 0.01 # 0.01 best for the given
    distrubance, after that failure

        equal_spacing = np.linspace(-2,2,self.number_centers)

        self.centers = equal_spacing*np.ones((6,self.
    number_centers))
        self.basis = np.zeros((self.number_centers,1));
        self.output_weight = np.zeros((self.number_centers,3)
    )

    def reference_model(self,yaw_pitch_roll_ref_OL):
        yaw_OL, pitch_OL, roll_OL = yaw_pitch_roll_ref_OL[0],
     yaw_pitch_roll_ref_OL[1], yaw_pitch_roll_ref_OL[2]
        wn = 20
        damping = 0.1
```

```
28          Arm = np.array([[0,1,0,0,0,0],[-wn**2,-2*damping*wn
    ,0,0,0,0],\
29                          [0,0,0,1,0,0],[0,0,-wn**2,-2*damping*
    wn,0,0],\
30                          [0,0,0,0,0,1],[0,0,0,0,-wn**2, -2*
    damping*wn]])
31
32          Brm = np.array([[0,0,0],[wn**2,0,0],[0,0,0],\
33                          [0,wn**2,0],[0,0,0],[0,0,wn**2]])
34
35          ref = np.array([[roll_OL],[pitch_OL],[yaw_OL]])
36
37          Bm_rt_pdt = np.dot(Brm,ref)
38
39          k1 = np.dot(Arm, self.ref_model_states) + Bm_rt_pdt
40          k2 = np.dot(Arm,(self.ref_model_states + k1*self.delt
    /2)) + Bm_rt_pdt
41          k3 = np.dot(Arm, (self.ref_model_states + k2*self.
    delt/2)) + Bm_rt_pdt
42          k4 = np.dot(Arm, (self.ref_model_states + k3*self.
    delt)) + Bm_rt_pdt
43
44          self.ref_model_states = self.ref_model_states + (self
    .delt/6)*(k1+2*k2+2*k3+k4) # roll,d_roll,pitch,d_pitch,yaw
    ,d_yaw
45
46      def mrac_weight_update(self,ref_model_states):
47          current_rpy_state = np.array([[self.roll],[self.
    D_roll],[self.pitch],[self.D_pitch],[self.yaw],[self.D_yaw
    ]])
48
49          for i in range(0,self.number_centers):
50              expression_1 = self.centers[:,i].reshape(-1,1)-
    current_rpy_state
51
52              expression = (-(np.linalg.norm(expression_1)**2))
    /(2*self.BW)
53
54              self.basis[i] = np.exp(expression)
55
56
57          self.basis[0] = 1
58
```

```
        error = ref_model_states-current_rpy_state
        P = np.array
    ([[50.13,0.0013,0,0,0,0],[0.0013,0.1253,0,0,0,0],\

    [0,0,50.13,0.0013,0,0],[0,0,0.0013,0.1253,0,0],\

    [0,0,0,0,50.13,0.0013],[0,0,0,0,0.0013,0.1253]])

        B = np.array([[0,0,0],[1,0,0],[0,0,0],\
                      [0,1,0],[0,0,0],[0,0,1]])

        self.output_weight = self.output_weight + (-self.delt
    )*(self.adaptive_gain)*np.dot(self.basis,np.dot(error.T,np
    .dot(P,B)))


    def mrac_torque(self):
        vad = np.dot(self.output_weight.T, self.basis)

        u_net = -vad

        return u_net
```

MRAC controller can be tested in a similar way as PID controller:

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 31 15:57:34 2020

@author: me112

Test Script
"""
from MRAC_control import MRAC_Control
import matplotlib.pyplot as plt
import numpy as np
import math

def drone_dynamics_with_control():
    A = MRAC_Control()
    A.assign_states([0,0,0,0,0,0,0,0,0,0,0,0])
    time_step=A.delt
    u,v,w = 0,0,0
    Jx = 0.0000582857
```

```
    Jy = 0.0000716914
    Jz = 0.0001

    X_vals=[]
    Y_vals=[]
    Z_vals=[]
    ref_traj_x=[]
    ref_traj_y=[]

    roll_vals=[]
    roll_refm_list =[]
    roll_OL_list =[]

    T_fig_8 = 20

    for time in range(1,5000): #Number of time steps
        # [math.cos(2*math.pi*time*0.005/T_fig_8),math.sin(4*math.pi*time*0.005/T_fig_8),-2,0] (fig 8 trajectory)


        ref_traj = [math.cos(2*math.pi*time*0.005/T_fig_8),math.sin(4*math.pi*time*0.005/T_fig_8),-2,0] # Xref(t),Yref(t),Zref(t),Yawref(t)



        [yaw_ref_OL, pitch_ref_OL, roll_ref_OL] = A.outer_loop_control(ref_traj)

        Thrust = A.Thrust_force(ref_traj)

        #A.reference_model([yaw_ref_OL, pitch_ref_OL, roll_ref_OL])

        [refm_roll], [refm_pitch], [refm_yaw] = A.ref_model_states[0], A.ref_model_states[2], A.ref_model_states[4]

        [torque_roll_pid, torque_pitch_pid, torque_yaw_pid] = A.inner_loop_control([refm_yaw, refm_pitch, refm_roll])

        [torque_roll_ad], [torque_pitch_ad], [torque_yaw_ad] = A.mrac_torque()
```

```
54
55
56          torque_roll = torque_roll_pid + torque_roll_ad
57          torque_pitch = torque_pitch_pid + torque_pitch_ad
58          torque_yaw = torque_yaw_pid + torque_yaw_ad
59
60          theta = A.pitch
61          psi = A.yaw
62          phi = A.roll
63
64          first_row = [math.cos(theta)*math.cos(psi), \
65                       math.sin(phi)*math.sin(theta)*math.cos(
    psi)-math.cos(phi)*math.sin(psi),\
66                       math.cos(phi)*math.sin(theta)*math.cos(
    psi)+math.sin(phi)*math.sin(psi)]
67
68          second_row = [math.cos(theta)*math.sin(psi), \
69                       math.sin(phi)*math.sin(theta)*math.sin(
    psi)+math.cos(phi)*math.cos(psi),\
70                       math.cos(phi)*math.sin(theta)*math.sin(
    psi)-math.sin(phi)*math.cos(psi)]
71
72          third_row = [-math.sin(theta), \
73                       math.sin(phi)*math.cos(theta),\
74                       math.cos(phi)*math.cos(theta)]
75
76          R_body_to_veh = np.array([first_row,second_row,
    third_row])
77
78
79          ## x_(t+1)=f(x_t,u_t)
80          [X_next] = np.array(A.X) + time_step*(np.dot(
    first_row,[[u],[v],[w]]))
81          [Y_next] = np.array(A.Y) + time_step*(np.dot(
    second_row,[[u],[v],[w]]))
82          [Z_next] = np.array(A.Z) + time_step*(np.dot(
    third_row,[[u],[v],[w]]))
83
84          u_next = u + time_step*(A.r*v-A.q*w - 9.81*math.sin(
    theta))
85          v_next = v + time_step*(A.p*w-A.r*u + 9.81*math.cos(
    theta)*math.sin(phi))
```

```
86          w_next = w + time_step*(A.q*u-A.p*v + 9.81*math.cos(
    theta)*math.cos(phi) + Thrust/A.mass)
87
88          phi_next = phi + time_step*(A.p + A.q*math.sin(phi)*
    math.tan(theta) + A.r*math.cos(phi)*math.tan(theta))
89          theta_next = theta + time_step*(A.q*math.cos(phi) - A
    .r*math.sin(phi))
90          psi_next = psi + time_step*((A.q*math.sin(phi) + A.r*
    math.cos(phi))/math.cos(theta))
91
92          if A.X>0: # Disturbance model
93              wind_x = 0#0.5*50*(1.5**2+math.sin(time))
94              wind_y = 0
95          else:
96              wind_x = 0
97              wind_y = 0#0.5*50*(1.5**2+math.sin(time))
98
99          p_next = A.p + time_step*(((Jy-Jz)/Jx)*A.q*A.r +
    torque_roll/Jx) +  time_step*wind_x
100         q_next = A.q + time_step*(((Jz-Jx)/Jy)*A.p*A.r +
    torque_pitch/Jy) + time_step*wind_y
101         r_next = A.r + time_step*(((Jx-Jy)/Jz)*A.p*A.q +
    torque_yaw/Jz)
102
103
104         [dx_next],[dy_next],[dz_next] = np.dot(R_body_to_veh
    ,[[u_next],[v_next],[w_next]])
105
106         u,v,w = u_next,v_next, w_next
107
108         # (t+1) steps
109         A.assign_states([X_next, Y_next, Z_next, psi_next,
    theta_next, phi_next, dx_next, dy_next,dz_next ,p_next,
    q_next, r_next])
110
111         A.reference_model([yaw_ref_OL, pitch_ref_OL,
    roll_ref_OL])
112
113         A.mrac_weight_update(A.ref_model_states)
114
115
116
117
```

```
118
119
120
121         X_vals.append(A.X)
122         Y_vals.append(A.Y)
123         Z_vals.append(A.Z)
124
125         roll_OL_list.append(roll_ref_OL)
126         roll_vals.append(A.roll)
127         roll_refm_list.append(refm_roll)
128
129         ref_traj_x.append(ref_traj[0])
130         ref_traj_y.append(ref_traj[1])
131
132
133     #print(np.linalg.norm(np.array(roll_vals)-np.array(
    roll_refm_list)))
134     #print(np.linalg.norm(np.array(ref_traj_x)-np.array(
    ref_traj_y)))
135     #plt.plot(X_vals)
136     #plt.plot(ref_traj_x,'k',linewidth=2)
137     plt.plot(Y_vals,X_vals)
138     plt.plot(ref_traj_y,ref_traj_x,'k')
139     #plt.plot(ref_traj_y,ref_traj_x,'k')
140     #plt.plot(roll_OL_list,'r',linewidth=4)
141
142     #plt.plot(roll_refm_list)
143     #plt.plot(roll_vals,'m')
144
145
146
147
148 drone_dynamics_with_control()
```

APPENDIX C

DMRAC CODE FOR QUADCOPTER CONTROL

Note, DMRAC controller incorporates the above PID control class. Here, when the entire buffer gets filled, the oldest entry gets replaced by the newest datapoint (FIFO). There is another way of choosing datapoints for the buffer using SVD maximisation highlighted in [20]. Although, it can improve DMRAC's performance even further, it wasn't considered here due to limited onboard computational power.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 30 00:48:05 2020

@author: Jasvir
"""

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from PID_control import PID_Control
import time

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.HL1 = nn.Linear(6,20)
        self.HL2 = nn.Linear(20,10)
        self.OL = nn.Linear(10,3)

        self.optimizer = optim.Adam(self.parameters(), lr
   =0.0005, weight_decay=0)
```

```
26
27
28          self.optimizer.zero_grad()
29
30          self.loss_fn = nn.MSELoss()
31
32      def forward(self, x):
33          OL_1 = torch.tanh(self.HL1(x))
34          OL_2 = torch.tanh(self.HL2(OL_1))
35          OL_3 = self.OL(OL_2)
36          return (OL_2,OL_3)
37
38
39
40  class Deep_MRAC_Control(PID_Control):
41      def __init__(self):
42          PID_Control.__init__(self)
43
44          self.dev = "cpu"#torch.device("cuda" if torch.cuda.
    is_available() else "cpu")
45          self.network = Net().to(self.dev)
46
47
48
49          self.ref_model_states = np.array
    ([[0],[0],[0],[0],[0],[0]])
50
51          self.adaptive_gain = 0.4 # PID- make this 0 and lr=0,
     MRAC with nn, make this like whatever for dist case
52          # for DMRAC, make this 0.4 for dist case and lr=0.001
53          # Note: MRAC with gaussian RBFs won't perform as well
     as the neural network MRAC case
54          # Best MRAC with NN performance with gain=0.9, after
    that high oscillation
55          # DMRAC with 0.4 adaptive gain and lr=0.001
    outperforms
56          # PID (put 0 inboth lr and adaptive gain)
57
58          self.last_layer_weight = np.zeros((10,3))
59
60          self.vad = np.zeros((3,1))
61
62          self.buffer_size = 250
```

```
63
64          self.input_training_data = np.zeros((6,self.
    buffer_size))
65
66          self.output_training_data = np.zeros((3,self.
    buffer_size))
67
68      def reference_model(self,yaw_pitch_roll_ref_OL):
69          yaw_OL, pitch_OL, roll_OL = yaw_pitch_roll_ref_OL[0],
     yaw_pitch_roll_ref_OL[1], yaw_pitch_roll_ref_OL[2]
70          wn = 20
71          damping = 0.1
72          Arm = np.array([[0,1,0,0,0,0],[-wn**2,-2*damping*wn
    ,0,0,0,0],\
73                          [0,0,0,1,0,0],[0,0,-wn**2,-2*damping*
    wn,0,0],\
74                          [0,0,0,0,0,1],[0,0,0,0,-wn**2, -2*
    damping*wn]])
75
76          Brm = np.array([[0,0,0],[wn**2,0,0],[0,0,0],\
77                          [0,wn**2,0],[0,0,0],[0,0,wn**2]])
78
79          ref = np.array([[roll_OL],[pitch_OL],[yaw_OL]])
80
81          Bm_rt_pdt = np.dot(Brm,ref)
82
83          k1 = np.dot(Arm, self.ref_model_states) + Bm_rt_pdt
84          k2 = np.dot(Arm,(self.ref_model_states + k1*self.delt
    /2)) + Bm_rt_pdt
85          k3 = np.dot(Arm, (self.ref_model_states + k2*self.
    delt/2)) + Bm_rt_pdt
86          k4 = np.dot(Arm, (self.ref_model_states + k3*self.
    delt)) + Bm_rt_pdt
87
88          self.ref_model_states = self.ref_model_states + (self
    .delt/6)*(k1+2*k2+2*k3+k4) # roll,d_roll,pitch,d_pitch,yaw
    ,d_yaw
89
90
91
92
93      def DMRAC_last_layer_weight_update(self,ref_model_states,
    second_last_layer_output_basis):
```

```
94
95          current_rpy_state = np.array([[self.roll],[self.
    D_roll],[self.pitch],[self.D_pitch],[self.yaw],[self.D_yaw
    ]])
96
97
98          error = ref_model_states-current_rpy_state
99          P = np.array
    ([[50.13,0.0013,0,0,0,0],[0.0013,0.1253,0,0,0,0],\
100
    [0,0,50.13,0.0013,0,0],[0,0,0.0013,0.1253,0,0],\
101
    [0,0,0,0,50.13,0.0013],[0,0,0,0,0.0013,0.1253]])
102
103         B = np.array([[0,0,0],[1,0,0],[0,0,0],\
104                       [0,1,0],[0,0,0],[0,0,1]])
105
106
107
108         self.last_layer_weight = self.last_layer_weight + (-
    self.delt)*(self.adaptive_gain)*np.dot(
    second_last_layer_output_basis,np.dot(error.T,np.dot(P,B))
    )# (8x3)
109
110
111
112     def deep_mrac_torque(self,second_last_layer_output_basis)
    :
113         self.vad = np.dot(self.last_layer_weight.T,
    second_last_layer_output_basis) #(3x8),(8x1)
114
115         u_net = self.vad
116
117         return u_net
118
119     def buffer_fill_simple(self,current_iter):
120         iter_number = current_iter%(self.buffer_size-1)
121         self.input_training_data[:,iter_number] = np.array([
    self.roll,self.D_roll,\
122
    self.pitch,self.D_pitch,\
123
    self.yaw,self.D_yaw])
```

```
124
125         self.output_training_data[:,iter_number] = self.vad
    [:,0]
126
127
128
129     def DMRAC_training(self,current_iter):
130
131
132         if current_iter > self.buffer_size:
133
134             random_numbers = np.random.randint(0,self.
    buffer_size,100)
135             random_input_data_for_training =  self.
    input_training_data.T[random_numbers]
136
137             random_output_data_for_training = self.
    output_training_data.T[random_numbers]
138             start = time.time()
139             for epoch in range(10):
140
141                 pred = self.network.forward(torch.Tensor(
    random_input_data_for_training).to(self.dev))[1]
142
143                 loss_vals = self.network.loss_fn(pred,torch.
    Tensor(random_output_data_for_training).to(self.dev))
144
145                 loss_vals.backward()
146                 self.network.optimizer.step()
147                 self.network.optimizer.zero_grad()
148                 print(current_iter,loss_vals) # For printing
    loss in each epochs
149
150             end = time.time()
151             print(end - start)
```

DMRAC controller can be tested in a similar way as PID controller:

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue Mar 31 15:57:34 2020
4
5 @author: me112
6
```

```
Test Script
"""
from DMRAC_control_gpu import Deep_MRAC_Control
import matplotlib.pyplot as plt
import numpy as np
import math
import torch



def drone_dynamics_with_control():
    torch.manual_seed(0) # For getting repeatable results
    A = Deep_MRAC_Control()
    device = "cpu"#torch.device("cuda" if torch.cuda.
   is_available() else "cpu")

    A.assign_states([0,0,0,0,0,0,0,0,0,0,0,0])
    time_step=A.delt
    u,v,w = 0,0,0
    Jx = 0.0000582857
    Jy = 0.0000716914
    Jz = 0.0001

    X_vals=[]
    Y_vals=[]
    Z_vals=[]
    ref_traj_x=[]
    ref_traj_y=[]

    roll_vals=[]
    roll_refm_list =[]
    roll_OL_list =[]
    T_fig_8=20

    for time in range(1,10000): #Number of time steps

        ref_traj = [math.cos(2*math.pi*time*0.005/T_fig_8),
   math.sin(4*math.pi*time*0.005/T_fig_8),-2,0] # Xref(t),
   Yref(t),Zref(t),Yawref(t)


```

```
46          [yaw_ref_OL, pitch_ref_OL, roll_ref_OL] = A.
    outer_loop_control(ref_traj)
47
48          Thrust = A.Thrust_force(ref_traj)
49
50          #A.reference_model([yaw_ref_OL, pitch_ref_OL,
    roll_ref_OL])
51
52          [refm_roll], [refm_pitch], [refm_yaw] = A.
    ref_model_states[0], A.ref_model_states[2], A.
    ref_model_states[4]
53
54          [torque_roll_pid, torque_pitch_pid, torque_yaw_pid] =
     A.inner_loop_control([refm_yaw, refm_pitch, refm_roll])
55
56          second_last_layer_output = A.network.forward(torch.
    Tensor([A.roll,A.D_roll,\
57                                                          A.
    pitch,A.D_pitch,\
58                                                          A.yaw,
    A.D_yaw]).to(device))[0]
59
60          second_lat_layer_output_cpu = torch.Tensor.cpu(
    second_last_layer_output)
61
62          second_lat_layer_output_detached =
    second_lat_layer_output_cpu.detach().numpy().T
63
64          second_last_layer_output_final_form = np.reshape(
    second_lat_layer_output_detached,(10,1))
65
66          [[torque_roll_ad], [torque_pitch_ad], [torque_yaw_ad
    ]] = A.deep_mrac_torque(
    second_last_layer_output_final_form)
67
68          A.buffer_fill_simple(time)
69
70
71          torque_roll = torque_roll_pid - torque_roll_ad
72          torque_pitch = torque_pitch_pid - torque_pitch_ad
73          torque_yaw = torque_yaw_pid - torque_yaw_ad
74
75
```

```
76
77          theta = A.pitch
78          psi = A.yaw
79          phi = A.roll
80
81          first_row = [math.cos(theta)*math.cos(psi), \
82                       math.sin(phi)*math.sin(theta)*math.cos(
    psi)-math.cos(phi)*math.sin(psi),\
83                       math.cos(phi)*math.sin(theta)*math.cos(
    psi)+math.sin(phi)*math.sin(psi)]
84
85          second_row = [math.cos(theta)*math.sin(psi), \
86                       math.sin(phi)*math.sin(theta)*math.sin(
    psi)+math.cos(phi)*math.cos(psi),\
87                       math.cos(phi)*math.sin(theta)*math.sin(
    psi)-math.sin(phi)*math.cos(psi)]
88
89          third_row = [-math.sin(theta), \
90                       math.sin(phi)*math.cos(theta),\
91                       math.cos(phi)*math.cos(theta)]
92
93          R_body_to_veh = np.array([first_row,second_row,
    third_row])
94
95
96          ## x_(t+1)=f(x_t,u_t)
97          [X_next] = np.array(A.X) + time_step*(np.dot(
    first_row,[[u],[v],[w]]))
98          [Y_next] = np.array(A.Y) + time_step*(np.dot(
    second_row,[[u],[v],[w]]))
99          [Z_next] = np.array(A.Z) + time_step*(np.dot(
    third_row,[[u],[v],[w]]))
100
101         if A.X>0: # Disturbance model
102             wind_x = 0.5*50*(1.5**2+math.sin(time))
103             wind_y = 0
104         else:
105             wind_x = 0
106             wind_y = 0.5*50*(1.5**2+math.sin(time))
107
108         u_next = u + time_step*(A.r*v-A.q*w - 9.81*math.sin(
    theta))
```

```
109         v_next = v + time_step*(A.p*w-A.r*u + 9.81*math.cos(
    theta)*math.sin(phi))
110         w_next = w + time_step*(A.q*u-A.p*v + 9.81*math.cos(
    theta)*math.cos(phi) + Thrust/A.mass)
111
112         phi_next = phi + time_step*(A.p + A.q*math.sin(phi)*
    math.tan(theta) + A.r*math.cos(phi)*math.tan(theta))
113         theta_next = theta + time_step*(A.q*math.cos(phi) - A
    .r*math.sin(phi))
114         psi_next = psi + time_step*((A.q*math.sin(phi) + A.r*
    math.cos(phi))/math.cos(theta))
115
116         p_next = A.p + time_step*(((Jy-Jz)/Jx)*A.q*A.r +
    torque_roll/Jx) + time_step*wind_x
117         q_next = A.q + time_step*(((Jz-Jx)/Jy)*A.p*A.r +
    torque_pitch/Jy) +  time_step*wind_y
118         r_next = A.r + time_step*(((Jx-Jy)/Jz)*A.p*A.q +
    torque_yaw/Jz)
119
120
121         [dx_next],[dy_next],[dz_next] = np.dot(R_body_to_veh
    ,[[u_next],[v_next],[w_next]])
122
123         # (t+1) steps
124         u,v,w = u_next,v_next, w_next
125
126         A.DMRAC_last_layer_weight_update(A.ref_model_states,
    second_last_layer_output_final_form)
127
128         A.assign_states([X_next, Y_next, Z_next, psi_next,
    theta_next, phi_next, dx_next, dy_next,dz_next ,p_next,
    q_next, r_next])
129
130         A.reference_model([yaw_ref_OL, pitch_ref_OL,
    roll_ref_OL])
131
132
133
134         # DMRAC training (need to uncomment this, whenever
    running DMRAC)
135
136         if time%200 == 0:
137             A.DMRAC_training(time)
```

```
138
139
140
141
142
143         # For making graphs
144
145         X_vals.append(A.X)
146         Y_vals.append(A.Y)
147         Z_vals.append(A.Z)
148
149         roll_OL_list.append(roll_ref_OL)
150         roll_vals.append(A.pitch)
151         roll_refm_list.append(refm_pitch)
152
153         ref_traj_x.append(ref_traj[0]) # Put ref_traj[0] here
     for graph plotting
154         ref_traj_y.append(ref_traj[1]) # Put ref_traj[1] here
     for graph plotting
155
156
157     #print(np.linalg.norm(np.array(roll_vals)-np.array(
    roll_refm_list)))
158     #print(np.linalg.norm(np.array(ref_traj_x)-np.array(
    ref_traj_y)))
159
160
161     plt.plot(Y_vals,X_vals)
162
163     plt.plot(ref_traj_y,ref_traj_x,'k',linewidth=2)
164
165     #plt.plot(ref_traj_y,ref_traj_x,'k')
166     #plt.plot(roll_OL_list,'r',linewidth=4)
167
168     #plt.plot(roll_refm_list)
169     #plt.plot(roll_vals,'m')
170
171
172
173
174 drone_dynamics_with_control()
```