

# Experiments on 2-DOF Helicopter Using Approximate Dynamic Programming

by

Anthony Birge and Andrew Fandel  
Advisor: Dr. Suruz Miah

Department of Electrical and Computer Engineering  
Caterpillar College of Engineering and Technology  
Bradley University

© Anthony Birge and Andrew Fandel, Peoria, Illinois, 2018

## Abstract

In recent years, the use of unmanned aerial vehicles has seen significant growth in commercial and military sectors. Motion control of such vehicles remains difficult due to precessional torques causing erratic movements. Conventional linear control techniques are widely used. This project aims to experiment with the use of an adaptive motion control strategy. Specifically, a model-based reinforcement learning strategy, approximate dynamic programming, was examined for use on a 2 degree-of-freedom (2-DOF) helicopter, the Quanser AERO.

The adaptive motion control strategy for a 2-DOF helicopter was implemented following electrical engineering methodologies. The proposed strategy was modified for the specific system which required modeling and mathematical analysis. Simulations and experiments were conducted in certain operating conditions. The motion control strategy was implemented to the Quanser AERO using Simulink, a Raspberry Pi, and a Raspberry Pi/smart phone. With successful implementation, a proof of concept can be attained for use in other applications using embedded systems.



## Acknowledgements

The authors would like to thank everyone who made this research possible. This includes the Electrical and Computer Engineering Department and Quanser. Quanser provided a free demonstration license, so we could complete the research without any additional cost. We would also like to thank our lab director, Mr. Christopher Mattus, for assisting in configuring our local wireless network. Finally, we would like to thank Mr. Lyall Randell for his extensive research in V-REP and quadcopters.

# Table of Contents

List of Tables	viii
List of Figures	ix
Nomenclature	1
<b>1 Introduction</b>	<b>2</b>
1.1 Literature Review . . . . .	2
1.2 Problem Statement . . . . .	3
1.3 Report Organization . . . . .	3
<b>2 Model-Based Reinforcement Learning</b>	<b>5</b>
2.1 ADP . . . . .	6
2.2 ADP Conditions . . . . .	12
<b>3 Quanser AERO System Modeling</b>	<b>13</b>
3.1 Half-Quadcopter . . . . .	14
3.2 2-DOF Helicopter . . . . .	17
<b>4 MATLAB Simulations</b>	<b>24</b>
4.1 Half-Quadcopter . . . . .	25
4.1.1 Results . . . . .	25
4.1.2 Analysis . . . . .	25
4.2 2-DOF Helicopter . . . . .	29
4.2.1 Results . . . . .	29
4.2.2 Analysis . . . . .	29

<b>5</b>	<b>Real-Time Simulation</b>	<b>33</b>
5.1	Required Software . . . . .	33
5.2	Communication . . . . .	34
5.3	Designing Quanser AERO . . . . .	34
5.3.1	Joint Control . . . . .	35
5.3.2	Motor Control . . . . .	36
5.4	Simulation Results . . . . .	40
5.4.1	LQR . . . . .	41
5.4.2	ADP . . . . .	41
5.4.3	PID . . . . .	41
5.4.4	Analysis . . . . .	41
<b>6</b>	<b>Hardware Implementation</b>	<b>54</b>
6.1	ADP Algorithm in Simulink . . . . .	56
6.2	QFLEX 2 USB . . . . .	58
6.2.1	Required Software . . . . .	59
6.2.2	Quanser AERO Simulink Model . . . . .	59
6.2.3	Results . . . . .	60
6.3	QFLEX 2 Embedded/Raspberry Pi . . . . .	77
6.3.1	Raspberry Pi 3 . . . . .	77
6.3.2	Required Software . . . . .	77
6.3.3	SPI Communication . . . . .	78
6.3.4	SPI Communication in Simulink . . . . .	80
6.3.5	Results . . . . .	82
6.4	Raspberry Pi/Android . . . . .	84
6.4.1	Required Software . . . . .	85
6.4.2	Simulink Model for Android Applications . . . . .	85
6.4.3	Results . . . . .	87
<b>7</b>	<b>Conclusion and Future Work</b>	<b>88</b>
7.1	Conclusion . . . . .	88
7.2	Future Directions . . . . .	89
	<b>APPENDICES</b>	<b>91</b>

<b>A</b>	<b>MATLAB</b>	<b>92</b>
A.1	ADP Simulation (MATLAB)	92
<b>B</b>	<b>Running V-REP</b>	<b>104</b>
B.1	Running a Simulation	104
B.1.1	File Setup	104
B.1.2	Running the Simulation	105
B.2	Skeletal Framework	105
B.2.1	MATLAB Framework	105
B.2.2	LUA Framework	107
B.3	Calculate State-Space	108
B.4	Joint Control	112
B.4.1	LUA 2-DOF Helicopter	112
B.4.2	LUA 2-DOF Half-Quadcopter	114
B.4.3	LQR Half-Quadcopter	116
B.4.4	LQR 2-DOF Helicopter	120
B.4.5	ADP Half-Quadcopter	125
B.4.6	ADP 2-DOF Helicopter	137
B.5	Motor Approach	148
B.5.1	Quanser AERO Base	148
B.5.2	Quanser AERO Motor#0	150
B.5.3	Quanser AERO Motor#1	152
B.5.4	Quanser AERO Motor PID Control	154
<b>C</b>	<b>QFLEX 2 USB</b>	<b>160</b>
C.1	QFLEX 2 USB Panel (Tutorial)	160
C.2	Initialization Code (MATLAB)	165
C.3	Update ADP Gain (MATLAB)	168
C.4	Base Color (MATLAB)	172
<b>D</b>	<b>Raspberry Pi 3</b>	<b>173</b>
D.1	Raspberry Pi 3 (Tutorial)	173
D.2	Initialization Code (MATLAB)	178
D.3	SPI Communication (MATLAB)	182

<b>E</b>	<b>Android Application</b>	<b>201</b>
E.1	Android Application (Tutorial) . . . . .	201
<b>F</b>	<b>Quanser AERO Data Sheets</b>	<b>205</b>
F.1	Quanser AERO Product Specifications . . . . .	205
F.2	Quanser AERO User Manual . . . . .	208
F.3	Quanser AERO Embedded Data Sheet . . . . .	224
	<b>References</b>	<b>229</b>

# List of Tables

3.1	Parameters calculated and measured by Quanser. . . . .	17
3.2	Parameters calculated and measured by Quanser. . . . .	23
5.1	Comparing V-REP and Quanser AERO Moments of Inertia . . . . .	35
5.2	File Sets . . . . .	36
5.3	Parameter values shown in Figure 5.5 . . . . .	39
5.4	Experimental V-REP LQR angular RMSE measurements. . . . .	41
5.5	Experimental V-REP ADP angular RMSE measurements. . . . .	41
6.1	Experimental Simulink angular RMSE measurements. . . . .	61
6.2	Experimental Simulink angular velocity RMSE measurements. . . . .	76
6.3	Experimental Simulink power RMS measurements. . . . .	76

# List of Figures

2.1	Generic state-feedback block diagram. . . . .	5
2.2	Critic neural network. . . . .	9
2.3	ADP timing flowchart. . . . .	10
3.1	Quanser AERO. (a) Quanser AERO configured as a 2-DOF helicopter. (b) A simplistic view of the Quanser AERO's free body diagram and its orientation.	13
3.2	(a) Rotor arm dynamics (top view) and (b) blade rotations ( $V_0 = V_1$ ). . . . .	14
3.3	(a) Force diagram (top view) and (b) force diagram (side view). . . . .	15
3.4	Gravitational torque exerted on Quanser AERO in $\theta$ plane. . . . .	16
3.5	General free body diagram of the Quanser AERO. . . . .	18
3.6	(a) Forces acting on the propeller controlling the pitch of the Quanser AERO. (b) Forces acting on the propeller controlling the yaw of the Quanser AERO. . . . .	19
4.1	Constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	26
4.2	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angu- lar position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	27
4.3	Sinusoidal desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	28
4.4	Constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	30
4.5	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) An- gular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	31

4.6	Sinusoidal desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	32
5.1	Communication scheme between MATLAB (acting as controller) and V-REP (acting as environment). . . . .	34
5.2	V-REP joint model flowchart. . . . .	36
5.3	Quanser AERO developed for joint control. (a) 2-DOF helicopter mode. (b) Half-quadcopter mode. . . . .	36
5.4	Measured currents and estimate function. . . . .	38
5.5	Circuit modeling Quanser AERO motors. . . . .	38
5.6	V-REP motor model flowchart. . . . .	40
5.7	Quanser AERO developed for motor control. (a) Half-quadcopter mode (side). (b) Half-quadcopter mode (top). . . . .	40
5.8	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	42
5.9	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	43
5.10	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	44
5.11	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	45
5.12	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	46
5.13	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	47
5.14	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	48
5.15	Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	49



5.16	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	50
5.17	Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	51
5.18	Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs. . . . .	52
5.19	Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs. . . . .	52
5.20	Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs. . . . .	53
6.1	Raspberry Pi 3. . . . .	55
6.2	From top to bottom. 1) User controls the Quanser AERO from their laptop via Wi-Fi through the Raspberry Pi 3. 2) User controls the Quanser AERO from their laptop via Ethernet through the Raspberry Pi 3. 3) User controls the Quanser AERO from their cell phone via Wi-Fi through the Raspberry Pi 3. 4) User controls the Quanser AERO via a USB connection using the QFLEX 2 USB panel. . . . .	55
6.3	ADP algorithm in the form of a Simulink model. . . . .	56
6.4	User-defined function to update the state-feedback gain in the ADP Simulink model. . . . .	57
6.5	User-defined function to change the base LED color. . . . .	58
6.6	Simulink subsystem used to communicate with the QFLEX 2 USB panel. . . . .	60
6.7	Constant desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	62
6.8	Constant desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	63
6.9	Step desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	64
6.10	Step desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	65
6.11	Sinusoidal desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	66

6.12	Sinusoidal desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	67
6.13	Constant desired pitch and sinusoidal desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	68
6.14	Constant desired pitch and sinusoidal desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	69
6.15	Sinusoidal desired pitch and constant desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	70
6.16	Sinusoidal desired pitch and constant desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	71
6.17	Sawtooth desired pitch and constant desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	72
6.18	Sawtooth desired pitch and constant desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	73
6.19	Square desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	74
6.20	Square desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs. . . . .	75
6.21	Overview of the SPI protocol. Image provided by <a href="https://www.mathworks.com/help/supportpkg/raspberrypi/ug/support-spi-communication.html">https://www.mathworks.com/help/supportpkg/raspberrypi/ug/support-spi-communication.html</a> . . . . .	79
6.22	Simulink model used to generate the code for the Raspberry Pi 3. . . . .	81
6.23	Simulink subsystem used for the ADP algorithm. . . . .	82
6.24	Simulink subsystem used specifically by the Raspberry Pi 3. . . . .	83
6.25	Simulink subsystem used to establish communication between the Raspberry Pi 3 and the Quanser AERO. . . . .	84
6.26	Simulink model used to communicate with the Android smart phone. . . . .	86
6.27	Simulink model used to create the Android smart phone application. . . . .	87
C.1	QFLEX 2 USB panel installed on the Quanser AERO. . . . .	160
C.2	Connections from your laptop to the Quanser AERO. . . . .	161

C.3	Build the Simulink model. . . . .	161
C.4	Connect to the target. . . . .	162
C.5	Run the Simulink model. . . . .	163
C.6	Observe the Quanser AERO. . . . .	163
C.7	Stop the Simulink model. . . . .	164
D.1	QFLEX 2 Embedded panel installed on the Quanser AERO. . . . .	173
D.2	Connect the Raspberry Pi to your laptop and power it on. . . . .	174
D.3	Connect to the Raspberry Pi using PuTTY. . . . .	174
D.4	Login to the Raspberry Pi. . . . .	175
D.5	Version of Linux installed on SD card by MATLAB. . . . .	176
D.6	Deploy model to hardware. . . . .	176
D.7	Connect the Raspberry Pi to the Quanser AERO. . . . .	177
D.8	Execute the code on the Raspberry Pi. . . . .	178
E.1	Connect your Android smart phone to your laptop. . . . .	202
E.2	Android smart phone application. . . . .	203
E.3	Android smart phone application. . . . .	204

---

## Abbreviations

**2-DOF** - 2 Degrees-Of-Freedom

**ADP** - Approximate Dynamic Programming

**LQR** - Linear Quadratic Regulator

**PID** - Proportional-Integral-Derivative

**RMSE** - Root Mean Squared Error

**SPI** - Serial Peripheral Interface

**V-REP** - Virtual Robot Experimentation Platform

## Mathematical Symbols

$J_p(J_y)$  - Total moment of inertia about pitch (yaw) axis

$J_\theta(J_\psi)$  - Total moment of inertia about pitch (yaw) axis

$D_p(D_y)$  - Damping constant about pitch (yaw) axis

$K_{sp}$  - Stiffness about pitch axis

$K_{pp}(K_{py})$  - Torque thrust gain acting on pitch from pitch (yaw) rotor

$K_{yy}(K_{yp})$  - Torque thrust gain acting on yaw from yaw (pitch) rotor

$V_p(V_y)$  - Applied voltage to pitch (yaw) motor

$V_0(V_1)$  - Applied voltage to motor #0 (#1)

$\theta(t)[\psi(t)]$  - Pitch [yaw] angle at time  $t \geq 0$

# Chapter 1

## Introduction

The area of unmanned aerial vehicles has seen significant growth over the past few years. Popular configurations of unmanned aerial vehicles include quadcopters and helicopters. The difficulty with such unmanned aerial vehicles is the nonlinear nature of the system due to coupling between the various rotors in addition to the stochastic nature of the environment that these vehicles operate in.

One such platform for testing control techniques of such unmanned aerial vehicles is the Quanser AERO. Essentially, the Quanser AERO is a testbed developed by Quanser Inc. The Quanser AERO offers accurate measurements of the system's pitch angle, yaw angle, as well as a variety of other velocity and acceleration data.

Frequently, the Quanser AERO is used by researchers to implement test algorithms for validation on a physical system that bear resemblance to a quadcopter or helicopter. Those techniques often involve expansive linearization; however, most techniques are not adaptive. The proposed model-based reinforcement learning technique known as approximate dynamic programming, ADP, uses an approximate system model and measured state error data to update the state-feedback gain of the system.

### 1.1 Literature Review

The Quanser AERO has been studied extensively for a variety of control methods, but Quanser also provides instructional workbooks and lab guides. Quanser documentation uses the linearized system model to calculate LQR state-feedback gain for teaching purposes. Quanser documentation also implements the use of linear-quadratic Gaussian (LQG) control, but more advanced control techniques have been proposed as well.

More extensive control techniques have been studied outside of Quanser Inc. LQR has been used by [17], but an additional adaptive controller was implemented. Sliding mode control has been studied by [1], but sliding mode control can become difficult to implement in comparison to other control techniques.

Fuzzy model-based control was studied by [5, 12]. [4] developed a fuzzy stability controller by using a PD controller to train an adaptive neural fuzzy inference system. Fuzzy control relies on two controllers until the fuzzy control takes complete action making the task of design more intensive considering two controllers.

[6] uses an ADP approach, but the system parameters are relied upon heavily. Neural networks have been proposed in [9], but the use of high-order neural networks are computationally tedious. Reinforcement learning is used in [16], but stabilization is controlled first then parameter estimation of the system.

The problem with the aforementioned techniques is they either involve considerable mathematics not easily implemented in embedded systems or known parameters which may be difficult to calculate. That is not to say that understanding the dynamics of quadcopters and their control in general is easy. Dynamics of quadcopters were discussed in [7, 14, 11]. [14] implemented PD stabilization, and [11] developed position control on a quadcopter. [3] implemented quadcopter control via a PID controller tuned with an LQR loop. [15] developed a real-time, adaptive high-gain EKF, extended Kalman filter, which regulates system angles suitable for inertial navigation.

The Quanser AERO and quadcopters in general have seen significant research over the past few years, but not many approaches prove adaptable to the system and environment.

## 1.2 Problem Statement

As discussed briefly earlier, the difficulty with controlling a nonlinear system is determining the optimal state-feedback gain. ADP is one such method that hopes to make this difficulty possible. In order to implement ADP on a physical system, standard engineering practices must be followed to have successful implementation. The steps leading to implementation include analyzing the method, modeling the system, simulating the method on the system, and finally implementing the method on the physical system.

In our case, the method, ADP, was analyzed and modified to control our specific system, the Quanser AERO. Because ADP utilizes an approximate system model, the Quanser AERO's system model needed to be derived. In order to see the effectiveness of ADP, we must perform simulations verifying our method and system. These simulations were conducted in both MATLAB and V-REP. With promising simulation results, physical implementation could begin. Implementation included implementing ADP in Simulink directly, on a Raspberry Pi, and using a Raspberry Pi/Android smart phone.

## 1.3 Report Organization

This report is organized as follows. Chapter 1 gives an introduction to the project and problem at hand. Chapter 2 discusses the model-based reinforcement learning approach

known as approximate dynamic programming, or ADP. Chapter 3 discusses the modeling of the system for ADP to be applied to, the Quanser AERO. Chapter 4 discusses the MATLAB simulations conducted to verify that the use of ADP for the Quanser AERO is possible. Chapter 5 extends the simulations conducted in Chapter 4 to provide a virtual representation of the system. Chapter 6 discusses how ADP can be physically implemented onto hardware. Finally, Chapter 7 discusses the conclusions found throughout this project.

Further resources are attached as appendices. Appendix A provides the MATLAB simulation code. Appendix B provides a tutorial and code pertaining to the real-time simulations. Appendix C provides a tutorial and MATLAB code pertaining to controlling the Quanser AERO via the QFLEX 2 USB panel. Appendix D provides a tutorial and MATLAB code pertaining to controlling the Quanser AERO via the QFLEX 2 Embedded panel and a Raspberry Pi. Appendix E provides a tutorial of implementing an Android smart phone application.

# Chapter 2

## Model-Based Reinforcement Learning

Reinforcement learning is a type of machine learning where an algorithm tries to determine proper actions in order to minimize or maximize some sort of performance measurement. The question is how can reinforcement learning be applied to the area of control theory? A perfect example is a simple state-feedback diagram shown in Figure 2. For most cases

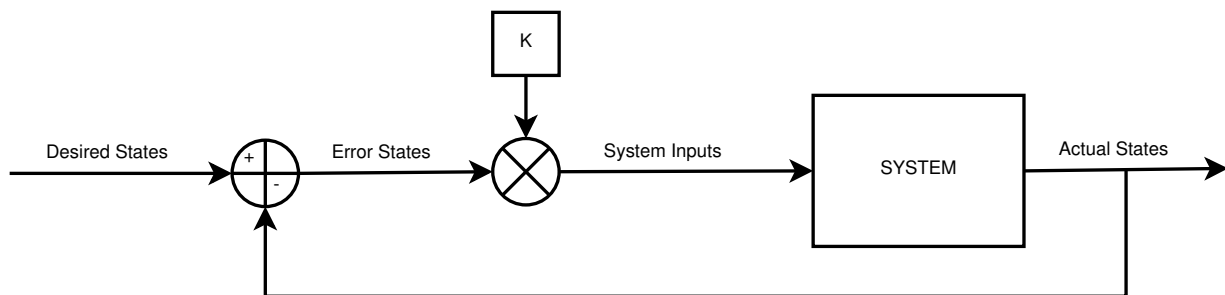


Figure 2.1: Generic state-feedback block diagram.

studied in an academic environment, the system in Figure 2 would be a linear system. The problem with this assumption is that most physical systems are in fact nonlinear systems. For a linear system, the state-feedback gain,  $K$ , is easily computed using conventional control techniques. For the nonlinear systems on the other hand, an accurate calculation for the state-feedback gain can be either difficult or impossible. This is where approximate dynamic programming, ADP, comes into play.

The main objective of ADP is to update the state-feedback gain in Figure 2 by using a model-based reinforcement learning approach. ADP accomplishes this task by utilizing measured state error data and an approximate system model. State error data is collected for  $T$  seconds at sample times of  $\tau$  seconds. Every  $T$  seconds the state-feedback gain is updated by using ADP. ADP uses the measured state error data as initial conditions to the approximate system model. ADP predicts future state errors as the initial conditions



would be propagated through system model. By adjusting the value of  $K$ , ADP can try to minimize or maximize a performance measurement to determine how the updated  $K$  values would affect the system. This seems like a rather simple approach, but ADP can become more complex mathematically.

The rest of this chapter is organized as follows. Section 2.1 goes into more depth concerning the mathematics of ADP. Section 2.2 discusses some situations which may deter ADP and how they can be overcome.

## 2.1 ADP

Let us denote the state and control variables of the system to be controlled with  $\mathbf{x}^T(t)$  and  $\mathbf{u}^T(t)$  at time  $t \geq 0$ , respectively. In order to utilize ADP, an approximate system model is needed which can be in the form of a typical state-space model seen in Equation 2.1.

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad (2.1)$$

We suppose that a more exact nonlinear system model could be used, but in this project, an approximated linear system model was used in ADP. Suppose that the system is supposed to reach a predefined desired (reference) state  $\mathbf{x}^{[d]}$  yielding its state error denoted by  $\mathbf{e}(t) = \mathbf{x}^{[d]} - \mathbf{x}(t)$ . Because all simulations and implementation must be considered in discrete-time, a sampling time must be defined for data collection and input update. Let  $t = k\tau$ , where  $k = 0, 1, 2, \dots$  is the discrete-time index, and  $\tau > 0$  is the sampling time between data collection and model update. The sampling time in which ADP updates the state-feedback gain is  $T$ .

Using the first-order Euler integration, the discrete-time error model of the system can then be written as

$$\mathbf{e}[k+1] = \mathbf{f}(\mathbf{e}[k]) + \mathbf{G}\mathbf{u}[k] \quad (2.2)$$

where  $\mathbf{f}(\mathbf{e}[k])$  and matrix  $\mathbf{G}$  are given as

$$\mathbf{f}(\mathbf{e}[k]) = \mathbf{e}[k] + \tau\mathbf{A}\mathbf{e}[k] - \tau\mathbf{A}\mathbf{x}^{[d]} \quad (2.3a)$$

$$\mathbf{G} = -\tau\mathbf{B}. \quad (2.3b)$$

The problem is to now find the optimal  $\mathbf{u}^*[k]$  such that the system's state error  $\mathbf{e}[k]$  asymptotically converges to zero, *i.e.*,  $\mathbf{e}[k] \rightarrow \mathbf{0}$  as  $k \rightarrow \infty$ .

The model-based reinforcement learning approach is based on the conventional dynamic programming technique which has the ability to determine optimal control/action inputs by considering possible future system states without actually experiencing them [8]. In

other words, the task is to find the sequence of actuator inputs  $\{\mathbf{u}(k)\}_{k=0}^{\infty}$  of the system that minimizes the cost function defined by

$$J(\mathbf{u}) = \sum_{k=0}^{\infty} (\mathbf{e}^T(k)\mathbf{Q}\mathbf{e}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k)) \quad (2.4)$$

subject to (2.2), where  $\mathbb{R}^{4 \times 4} \ni \mathbf{Q} = \mathbf{Q}^T \geq \mathbf{0}$  and  $\mathbb{R}^{2 \times 2} \ni \mathbf{R} = \mathbf{R}^T > \mathbf{0}$  penalize the system's error deviation from its desired state and its control effort, respectively. It should be noted that these dimensions are with respect to our specific system which will be discussed in the next chapter.

In addition, the cost-to-go function (value function) of the system is defined as

$$V(\mathbf{e}(k)) = \sum_{\kappa=k}^{\infty} (\mathbf{e}^T(\kappa)\mathbf{Q}\mathbf{e}(\kappa) + \mathbf{u}^T(\kappa)\mathbf{R}\mathbf{u}(\kappa)). \quad (2.5)$$

Note that the second term of the value function (2.5) takes into account the asymptotic energy consumption of the system. Following [13, Ch. 11], the value function (2.5) can be rewritten in the form:

$$V(\mathbf{e}(k)) = \mathbf{e}^T(k)\mathbf{Q}\mathbf{e}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k) + V(\mathbf{e}(k+1)). \quad (2.6)$$

Therefore, the optimal control input  $\mathbf{u}(k)$  that drives the system using (2.1) to its desired configuration can be obtained by solving the following minimization problem:

$$\mathbf{u}^*(k) = \operatorname{argmin}_{\mathbf{u}(k)} \left( \mathbf{e}^T(k)\mathbf{Q}\mathbf{e}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k) + V^*(\mathbf{e}(k+1)) \right). \quad (2.7)$$

A standard solution to the minimization problem (2.7) is obtained by solving the discrete-time Hamilton–Jacobi–Bellman equation given by

$$V^*(\mathbf{e}(k)) = \min_{\mathbf{u}(k)} \left( \mathbf{e}^T(k)\mathbf{Q}\mathbf{e}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k) + V^*(\mathbf{e}(k+1)) \right). \quad (2.8)$$

Following [2], calculating the gradient of the right side of (2.8) and setting it equal to zero yields

$$\frac{\partial}{\partial \mathbf{u}(k)} (\mathbf{e}(k)^T \mathbf{Q} \mathbf{e}(k) + \mathbf{u}^T(k) \mathbf{R} \mathbf{u}(k)) + \left( \frac{\partial \mathbf{e}(k+1)}{\partial \mathbf{u}(k)} \right)^T \nabla V^*(\mathbf{e}(k+1)) = \mathbf{0} \quad (2.9)$$

where  $\nabla$  is the gradient operator which defines  $\nabla V^*(\mathbf{e}(k+1)) = \frac{\partial V^*(\mathbf{e}(k+1))}{\partial \mathbf{e}(k+1)}$  as a column vector. Using (2.2), the optimal control action of the system at time instant  $k$  is given by

$$\mathbf{u}^*(k) = -\frac{1}{2} \mathbf{R}^{-1} \mathbf{G}^T \nabla V^*(\mathbf{e}(k+1)). \quad (2.10)$$

Note that the solution  $\mathbf{u}^*(k)$  in (2.10) is straight forward given the fact that  $\nabla V^*(\mathbf{e}(k+1))$  is defined. Since the value function (2.5) constitutes an infinite summation that takes into account the changes of the system's state-space, this function is to be approximated based on the data collected along the system trajectory (2.2).

Similar to the conventional LQR-based optimal control technique, the cost-to-go function (2.5) can be expressed as a quadratic function, *i.e.*,

$$V(\mathbf{e}(k)) = \mathbf{e}^T(k)\mathbf{P}\mathbf{e}(k) \quad (2.11)$$

for some  $\mathbf{P} \in \mathbb{R}^{4 \times 4}$ . It should be noted that these dimensions are for our specific system which will be discussed in the next chapter. The cost-to-go function (2.11) can be approximated by

$$V(\mathbf{e}(k)) = (\text{vec}(\mathbf{P}))^T(\mathbf{e}(k) \otimes \mathbf{e}(k)) \equiv \mathbf{w}_c^T \phi(\mathbf{e}(k)) \quad (2.12)$$

where  $\otimes$  is the Kronecker product operator, and the weight  $\mathbf{w}_c = \text{vec}(\mathbf{P})$ , where the operator  $\text{vec}(\mathbf{P})$  forms the vector by stacking columns of the matrix  $\mathbf{P}$ . The weight vector  $\mathbf{w}_c$  can be approximated using the critic neural network which will be discussed shortly. The vector-valued function  $\phi(\mathbf{e}(k)) = \mathbf{e}(k) \otimes \mathbf{e}(k)$  is the quadratic polynomial vector containing all possible pairwise products of the two components of  $\mathbf{e}(k)$ . According to the Weierstrass approximation theorem, the function  $V(\mathbf{e}(k))$  can be approximated using basis functions which are polynomials of the combination of state elements ( $e^{(1)}(k), e^{(2)}(k), e^{(3)}(k), e^{(4)}(k)$ ). It should be noted again that these dimensions are for our specific system which will be discussed in the chapter. Here we choose the basis functions up to the second order. Therefore,  $\phi(\mathbf{e}(k)) = [\phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6, \phi_7, \phi_8, \phi_9, \phi_{10}, \phi_{11}, \phi_{12}, \phi_{13}, \phi_{14}]^T$  with

$$\phi_1(\mathbf{e}(k)) = e^{(1)}(k) \quad (2.13a)$$

$$\phi_2(\mathbf{e}(k)) = e^{(2)}(k) \quad (2.13b)$$

$$\phi_3(\mathbf{e}(k)) = e^{(3)}(k) \quad (2.13c)$$

$$\phi_4(\mathbf{e}(k)) = e^{(4)}(k) \quad (2.13d)$$

$$\phi_5(\mathbf{e}(k)) = (e^{(1)}(k))^2 \quad (2.13e)$$

$$\phi_6(\mathbf{e}(k)) = e^{(1)}(k)e^{(2)}(k) \quad (2.13f)$$

$$\phi_7(\mathbf{e}(k)) = e^{(1)}(k)e^{(3)}(k) \quad (2.13g)$$

$$\phi_8(\mathbf{e}(k)) = e^{(1)}(k)e^{(4)}(k) \quad (2.13h)$$

$$\phi_9(\mathbf{e}(k)) = (e^{(2)}(k))^2 \quad (2.13i)$$

$$\phi_{10}(\mathbf{e}(k)) = e^{(2)}(k)e^{(3)}(k) \quad (2.13j)$$

$$\phi_{11}(\mathbf{e}(k)) = e^{(2)}(k)e^{(4)}(k) \quad (2.13k)$$

$$\phi_{12}(\mathbf{e}(k)) = (e^{(3)}(k))^2 \quad (2.13l)$$

$$\phi_{13}(\mathbf{e}(k)) = e^{(3)}(k)e^{(4)}(k) \quad (2.13m)$$

$$\phi_{14}(\mathbf{e}(k)) = (e^{(4)}(k))^2 \quad (2.13n)$$

being the basis functions of the critic neural network, where the value function (2.12) can be approximated using a linear-in-weight critic neural network structure modeled by

$$\hat{V}(\mathbf{e}(k)) \cong \mathbf{w}_c^T \phi(\mathbf{e}(k)) \quad (2.14)$$

where  $\mathbf{w}_c \in \mathbb{R}^{10 \times 1}$  is the weight vector for the critic neural network. The output node of the critic neural network produces the approximate output of the cost-to-go function using (2.14). The critic neural network can be seen in Figure 2.2.

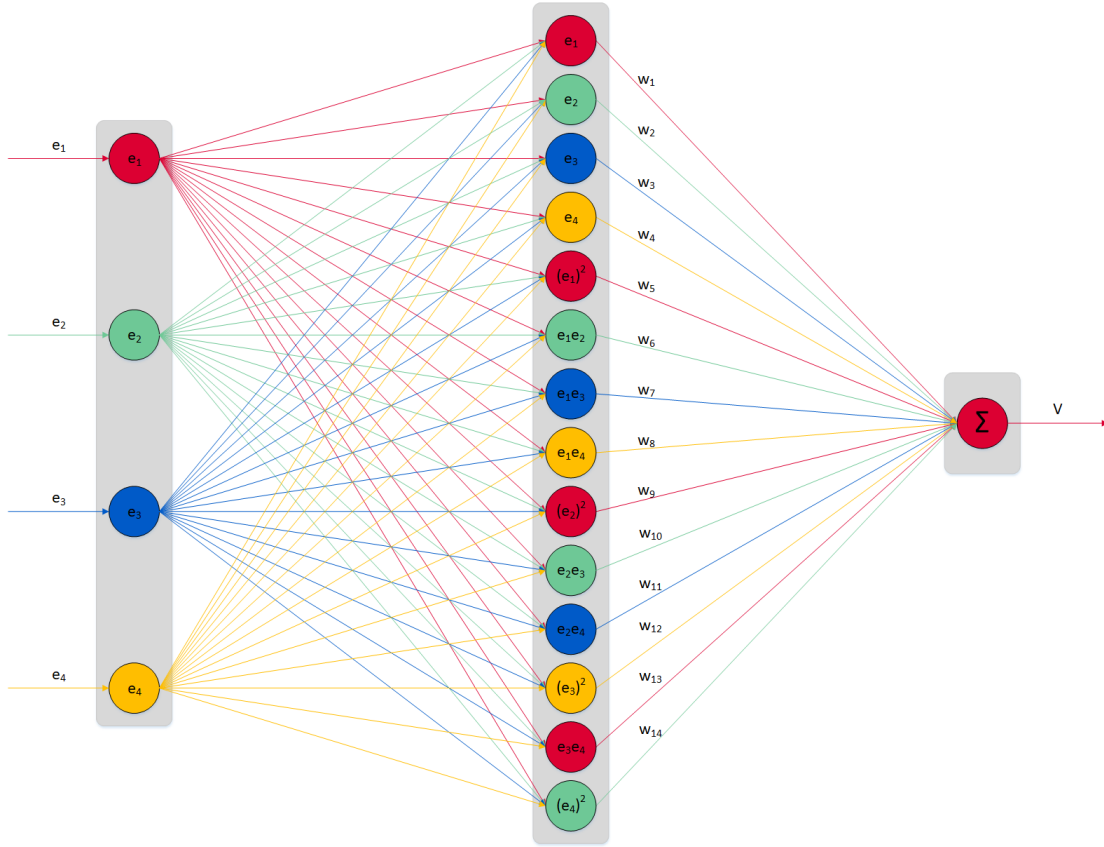


Figure 2.2: Critic neural network.

Similar to [10], the critic weight vector,  $\mathbf{w}_c$ , is solved using the least square technique, except that the system's data (state error) is collected online along its trajectory. For that, we introduce another sampling time  $\tau$  such that the sampling time between state-feedback gain updates is  $T = \bar{n}\tau$ , with  $\bar{n} > 14$  being a scalar parameter that quantifies the number of training samples. Note that training samples are simply a sequence of data points  $\{\mathbf{e}(k + \kappa), \mathbf{e}(k + \kappa + 1), \dots\}_{\kappa=0}^{\bar{n}-1}$  collected by the system along its trajectory, where  $V(\mathbf{e}(k + \kappa))$  is the desired (reference) value function at time instant  $k + \kappa$  with  $\kappa = 0, 1, \dots, \bar{n} - 1$ . To solve for the weight vector  $\mathbf{w}_c = [w_1, w_2, \dots, w_{14}]^T$  using the batch least square technique, the number of training samples of the critic neural network is chosen such that  $\bar{n} > 14$ . The target value function is determined using

$$V(\mathbf{e}(k)) = \mathbf{e}^T(k)\mathbf{Q}\mathbf{e}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k) + \mathbf{w}_c^T\phi(\mathbf{e}(k+1)), \quad (2.15)$$

and the estimated value function is the output of the critic neural network given by (2.14). The least square error is then defined as

$$\delta_c = \frac{1}{2} \sum_{\kappa=0}^{\bar{n}-1} [V(\mathbf{e}(k)) - \hat{V}(\mathbf{e}(k))]^2. \quad (2.16)$$

The weight vector,  $\mathbf{w}_c$ , that minimizes the sum-of-square error  $\delta_c$  is given by

$$\mathbf{w}_c = (\mathbf{\Lambda}^T \mathbf{\Lambda})^{-1} \mathbf{\Lambda}^T \mathbf{V} \quad (2.17)$$

where the matrix  $\mathbb{R}^{14 \times \bar{n}} \ni \mathbf{\Lambda} = [\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{\bar{n}-1}]^T$  with  $\mathbf{a}_\kappa = \phi^T(\mathbf{e}(k + \kappa))$  and  $\mathbf{V} = [v_0, v_1, \dots, v_{\bar{n}-1}]^T$  with  $v_\kappa = V(\mathbf{e}(k + \kappa))$  for  $\kappa = 0, 1, \dots, \bar{n} - 1$ .

The key steps of the ADP technique implemented in this work are given in Algorithm 1. A timing flowchart of ADP can be seen in Figure 2.3.

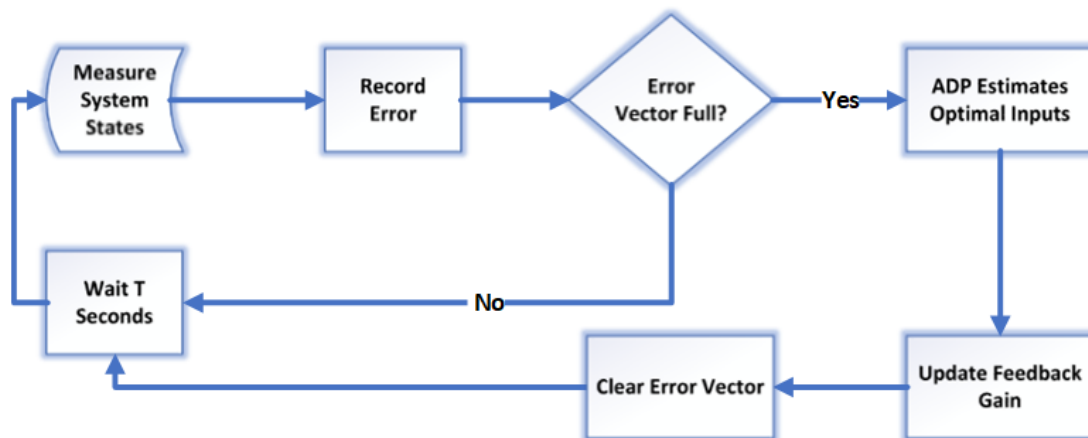


Figure 2.3: ADP timing flowchart.

**Algorithm 1:** ADP Outline

---

**Input:**  $\mathbf{x}^{[d]}$   
**Output:**  $\mathbf{x}$

```

1 begin
2   repeat
3     •  $t = k\tau$ 
4     • Apply  $\mathbf{u}[k]$  to system model  $\rightarrow \mathbf{x}[k]$ 
5     • Apply Equation 2.1 to  $\mathbf{x}[k] \rightarrow \mathbf{x}[k+1]$ 
6     • Constrain  $\psi$  on  $[-180^\circ, 180^\circ]$  and  $\theta$  on  $[-90^\circ, 90^\circ]$ 
7     • Calculate error  $\rightarrow \mathbf{e}[k] = \mathbf{x}^{\text{ref}}[k] - \mathbf{x}[k]$ 
8     • Calculate updated  $\mathbf{w}_c$  if  $t = T$ 
9     repeat
10      •  $\mathbf{w}_{last} = \mathbf{w}_c, i = 0$ 
11      repeat
12        •  $\mathbf{u}[i] = [0, 0]^T, j = 0$ 
13        repeat
14          •  $\mathbf{u}_{last}[i] = \mathbf{u}[i]$ 
15          • Find  $\mathbf{e}[i+1]$  using collected error data and Equation 2.2
16          • Compute new  $\mathbf{u}[i]$  using Equation 2.10
17          •  $j = j + 1$ 
18        until  $\|\mathbf{u}[i] - \mathbf{u}_{last}[i]\| < \epsilon$  or  $j = j_{max}$ 
19        Output:  $\mathbf{u}[i]$ 
20        • Find  $V(i)$  using Equation 2.15
21        •  $\Lambda(i) = \phi(\mathbf{e}(i))$ 
22        •  $i = i + 1$ 
23      until  $i = \bar{n} - 1$ 
24      Output:  $\Lambda, V$ 
25      • Update  $\mathbf{w}_c$  using Equation 2.17
26    until  $\|\mathbf{w}_c - \mathbf{w}_{last}\| < \epsilon_c$ 
27    Output:  $\mathbf{w}_c$ 
28    • Calculate state-feedback gain using Equation 2.10 and Equation 2.11
29    • Constrain optimal inputs on  $[-18, 18]$ 
30    •  $k = k + 1$ 
31  until  $t = t_f$ 

```

---

## 2.2 ADP Conditions

ADP utilizes a critic neural network to approximate the value function. The weights of this critic neural network are determined recursively. The problem with this approach, and ADP in general, is that these weights can diverge. It was initially suspected that the weights would converge on almost all instances, but it turns out this is not the case. Initial MATLAB simulations, that will be discussed Chapter 4, showed that the weights always seemed to diverge for a linear model. It appears that the neural network cannot handle the error data of a linear system. The cause may be due to the error data not being random enough because of a truly linear system. This divergence of the weights make ADP useless if it cannot support simple linear systems.

After much research and trial-and-error with the MATLAB simulations, it was determined that several conditional cases should be considered when performing the algorithm. Research showed that if random error data was inputted to ADP, the weights would converge on almost all instances. This findings still baffle us as to why the random error data causes convergent weights while the linear system error data causes divergent weights. Further research will be needed to determine the cause of this phenomenon.

The conditional cases implemented in ADP utilize the fact that the weights converge for random error data. Before ADP is used, it is ran with random error data to obtain convergent weights; these weights are then saved. When ADP is called again, several types of errors can cause divergent weights. Because the algorithm is recursive, we can reach the end of one of the recursive loops. If this is the case, the initial convergent weights should be used to save time instead of continuing for further loop iterations. The weights may also not be found because the least-squares regression calculation cannot be performed. If this is the case, the initial convergent weights should be used again as well. The last possible failure would be if the weights actually begin to diverge to infinity. If this is the case, the weights will be assigned a very large number which is user-defined in the algorithm.

These conditional cases in ADP algorithm have been shown effective, and they guarantee that ADP will output some type of weights instead of failing.

# Chapter 3

## Quanser AERO System Modeling

In order to test the effectiveness of the ADP algorithm, the ideal testing platform would be a nonlinear system that is very difficult to control. This almost chaotic system would then highlight the effectiveness of ADP in its adaptive learning capabilities. Our problem is we do not have such a system readily available for testing, and it is also difficult to simulate such a system. The closest physical system we have is the Quanser AERO (<https://www.quanser.com/products/quanser-aero/>) seen in Figure 3.1(a).

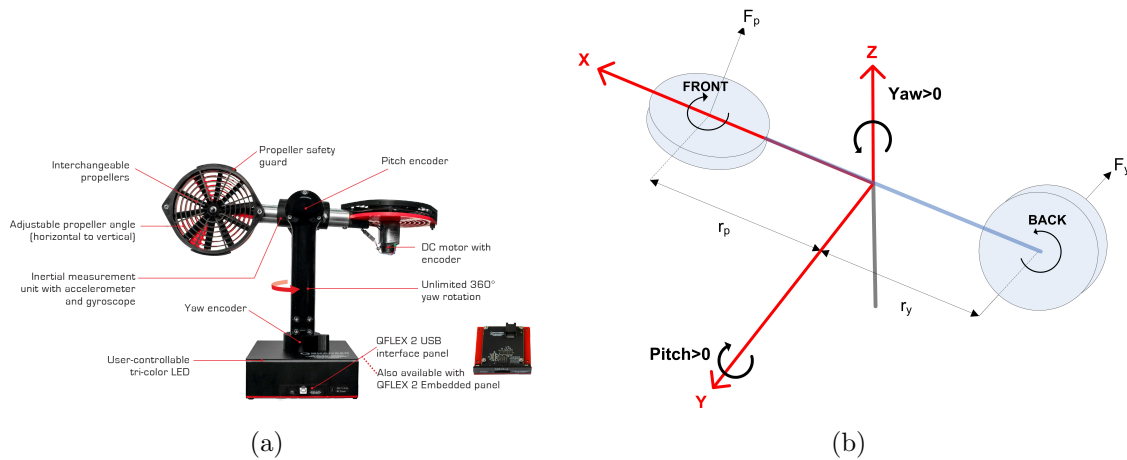


Figure 3.1: Quanser AERO. (a) Quanser AERO configured as a 2-DOF helicopter. (b) A simplistic view of the Quanser AERO's free body diagram and its orientation.

The Quanser AERO is a configurable testing platform that utilizes two fans to mimic a 2-DOF helicopter. It should be noted that this system does not actually model a physical helicopter, but it does model something similar. The Quanser AERO can either be configured as a 2-DOF helicopter or a half-quadcopter both of which will be discussed. This reconfiguration can be done by unscrewing various screws and rotating the fan assemblies. Control of the Quanser AERO can be achieved by applying the proper input voltages  $V_p$  and  $V_y$  to the main and tail DC motors, respectively. Determining the values of these motor input voltages is the task of ADP.



In order to be able use ADP, we need to know the system model of our particular Quanser AERO configuration. This system model can be approximated, but a relatively accurate model compared to that of the actual system model is needed for ADP. In order to utilize state-space modeling, we need to determine what states we will be considering. The Quanser AERO utilizes many sensors for measurement, but we will only focus on the sensors measuring pitch ( $\theta$ ) and yaw ( $\psi$ ). Let us denote the state and control (DC motor input voltages) variables of the state-space models as  $\mathbf{x}^T(t) \equiv [\theta(t), \psi(t), \dot{\theta}(t), \dot{\psi}(t)]$  and  $\mathbf{u}^T(t) \equiv [V_p(t), V_y(t)]$  at time  $t \geq 0$ , respectively. It should be noted that  $\psi > 0$  when the Quanser AERO moves in a counter-clockwise direction, and  $\theta > 0$  when the Quanser AERO raises above the horizontal plane.

This chapter examines the state-space models of the Quanser AERO configured as a half-quadcopter and 2-DOF helicopter. This chapter is organized as follows. Section 3.1 shows the derivation of the Quanser AERO configured as a half-quadcopter. Section 3.2 shows the derivation of the Quanser AERO configured as a 2-DOF helicopter.

### 3.1 Half-Quadcopter

The propeller-arm system comprises half of the Quanser AERO, shown in Figure 3.2. The movement of the arm is characterized by a thrust generated by the propeller and a resultant force, perpendicular to the thrust, generated as a result of Newton's Third Law by virtue of the motor accelerating the propeller and the propeller driveshaft.

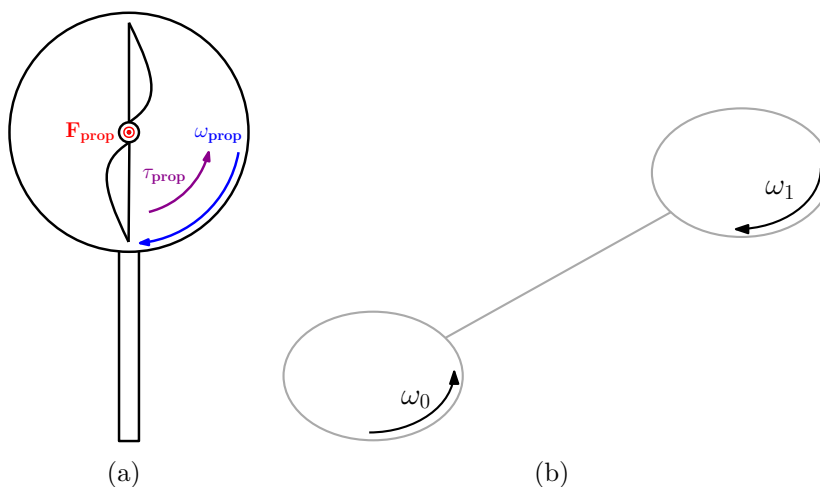


Figure 3.2: (a) Rotor arm dynamics (top view) and (b) blade rotations ( $V_0 = V_1$ ).

We'll assume that each rotor produces an upward force, or *thrust*.  $T_i$ , that is proportional to the square of the angular velocity of the propeller times an empirically derived constant,  $c_t$ .  $c_t$  is a constant empirically derived from various characteristics of the surrounding air

[14].

$$F_{prop_i} = c_t \omega_i^2, \quad \text{for } i = 0, 1 \quad (3.1)$$

The thrust force will be applied at radius,  $R$ , creating a torque on the body in the  $\theta$  plane.

$$\tau_{\theta_i} = R F_{prop_i}, \quad \text{for } i = 0, 1 \quad (3.2)$$

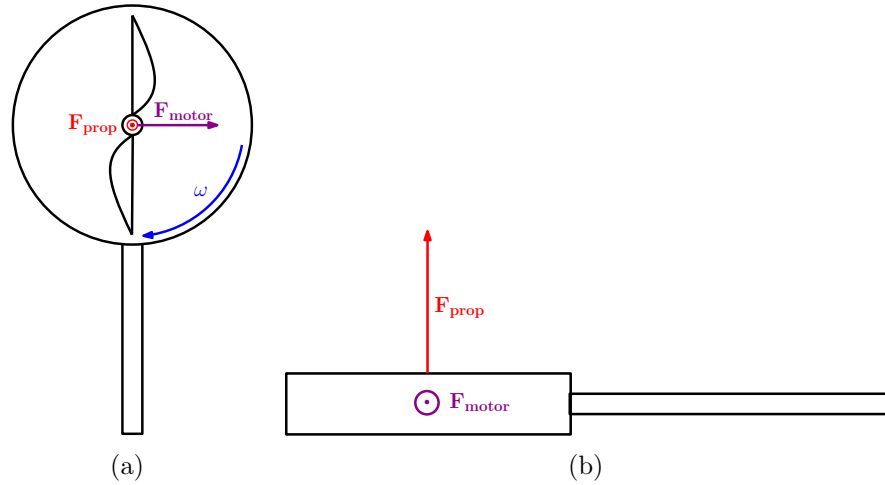


Figure 3.3: (a) Force diagram (top view) and (b) force diagram (side view).

Gyroscopic effects from the propeller on the body and torque on body from the drag of the propeller generate torque opposite of the direction of  $\omega_{prop}$ . The torque will be a function of the squared angular speed of the propeller multiplied by a constant,  $b$ .  $b$  is empirically determined and understood to be a function of the propeller cross sectional area and surrounding air density.

The gyroscopic effects on the body are defined as torques  $\tau_i$  and will articulate the Quanser AERO perpendicular to the thrust causing movement in the  $\psi$  plane:

$$\tau_{\psi_i} = b\omega_i^2 + J_{prop}\dot{\omega}_i, \quad \text{for } i = 0, 1 \quad (3.3)$$

Damping torques exist in the Quanser AERO in the form of bearing stiffness.  $D_\theta$  and  $D_\psi$  are constants empirically determined for the system. These torques,  $\tau_{\theta_d}$  and  $\tau_{\psi_d}$ , are expressed as follows

$$\tau_{\theta_d} = -D_\theta \dot{\theta} \quad (3.4a)$$

$$\tau_{\psi_d} = -D_\psi \dot{\psi} \quad (3.4b)$$

Finally, in the pitch plane, a gravitational torque exists on the Quanser AERO. The gravitational torque is found at the center of mass which is  $R_c$  from the pivot. The gravitational

force is then the mass of the body and the two propellers.  $g$  is the acceleration due to gravity.

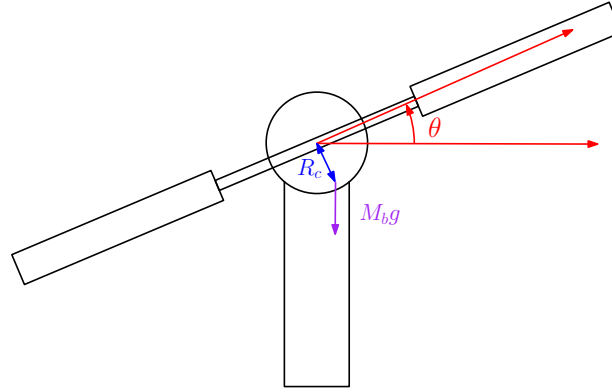


Figure 3.4: Gravitational torque exerted on Quanser AERO in  $\theta$  plane.

$$\tau_{\theta_g} = R_c M_b g \sin(\theta) \quad (3.5)$$

Next, we use a definition of torque,  $\tau$ :

$$\sum_n \tau = J_{\text{total}} \alpha_{\text{total}} = J_{\theta} \ddot{\theta} + J_{\psi} \ddot{\psi} \quad (3.6)$$

We use components of torque to generalize:

$$\tau_{\theta} = \tau_{\theta_0} + \tau_{\theta_1} + \tau_{\theta_d} + \tau_{\theta_g} \quad (3.7a)$$

$$\tau_{\psi} = (\tau_{\psi_0} + \tau_{\psi_1} + \tau_{\psi_d}) \cos(\theta) \quad (3.7b)$$

Next, we consider the coordinates of our system to finalize the equations of motion:

$$\tau_{\theta} = J_{\theta} \ddot{\theta} = R_c t (\omega_0^2 - \omega_1^2) - R_c M_b g \sin(\theta) - D_{\theta} \dot{\theta} \quad (3.8a)$$

$$\tau_{\psi} = J_{\psi} \ddot{\psi} = b \omega_0^2 - b \omega_1^2 + J_{prop} \dot{\omega}_0 - J_{prop} \dot{\omega}_1 - D_{\psi} \dot{\psi} \quad (3.8b)$$

The linearized form of the controls are made with the following assumptions:

$$R_c M_b g \sin(\theta) \approx K_{sp} \theta \quad (3.9a)$$

$$\cos(\theta) \approx 1 \quad (3.9b)$$

$$J_{prop} \dot{\omega}_{0,1} \approx 0 \quad (3.9c)$$

$$R_c t \omega_{0,1}^2 \propto K_{pp} V_{0,1} \quad (3.9d)$$

$$b \omega_1^2 \propto K_{yy} V_{0,1} \quad (3.9e)$$

where  $K_{yy}$  and  $K_{pp}$  are thrust gains, and  $K_{sp}$  is the stiffness constant.

$$\tau_\theta = J_\theta \ddot{\theta} = K_{pp}V_0 - K_{pp}V_1 - K_{sp}\theta - D_\theta \dot{\theta} \quad (3.10a)$$

$$\ddot{\theta} = \frac{K_{pp}V_0 - K_{pp}V_1 - K_{sp}\theta - D_\theta \dot{\theta}}{J_\theta} \quad (3.10b)$$

$$\tau_\psi = J_\psi \ddot{\psi} = K_{yy}V_0 - K_{yy}V_1 + -D_\psi \dot{\psi} \quad (3.11a)$$

$$\ddot{\psi} = \frac{K_{yy}V_0 - K_{yy}V_1 + -D_\psi \dot{\psi}}{J_\psi} \quad (3.11b)$$

Next, the linearized equations of motion are realized in a state-space representation with Table 3.1 representing the constant values:

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -K_{sp}/J_\theta & 0 & -D_\theta/J_\theta & 0 \\ 0 & 0 & 0 & -D_\psi/J_\psi \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ K_{pp}/J_\theta & -K_{pp}/J_\theta \\ K_{yy}/J_\psi & -K_{yy}/J_\psi \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} \quad (3.12)$$

Table 3.1: Parameters calculated and measured by Quanser.

Parameter	Value	Unit
$J_\theta$	0.0215	[kg m <sup>2</sup> ]
$J_\psi$	0.0237	[kg m <sup>2</sup> ]
$D_\theta$	0.00711	[N m s rad <sup>-1</sup> ]
$D_\psi$	0.0220	[N m s rad <sup>-1</sup> ]
$K_{sp}$	0.0375	[N m rad <sup>-1</sup> ]
$K_{pp}$	0.0011	[N m V <sup>-1</sup> ]
$K_{yy}$	0.0022	[N m V <sup>-1</sup> ]

Using Equations 3.12 and Table 3.1, the model can explicitly be stated as:

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1.7442 & 0 & -0.3307 & 0 \\ 0 & 0 & 0 & -0.9283 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.05116 & -0.05116 \\ 0.09282 & -0.09282 \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} \quad (3.13)$$

## 3.2 2-DOF Helicopter

In order to utilize ADP for the 2-DOF helicopter, the state-space model of the system must be derived. Quanser documentation has provided this derivation, but we started

from the beginning in order to verify their results and better understand the system. It should be noted that the system model derived here contains many assumptions and linearizations. These same assumptions and linearizations were assumed by Quanser in their documentation. It should also be noted that this derivation is very different from the one derived in Section 3.1 where Quanser documentation was not provided.

To begin the state-space model derivation, let us draw a free body diagram of the Quanser AERO which can be seen in Figure 3.2. We can then look at the free body diagram in terms of the two propellers seen in Figures 3.6(a) and 3.6(b). This also divides the forces in terms of horizontal and vertical planes.

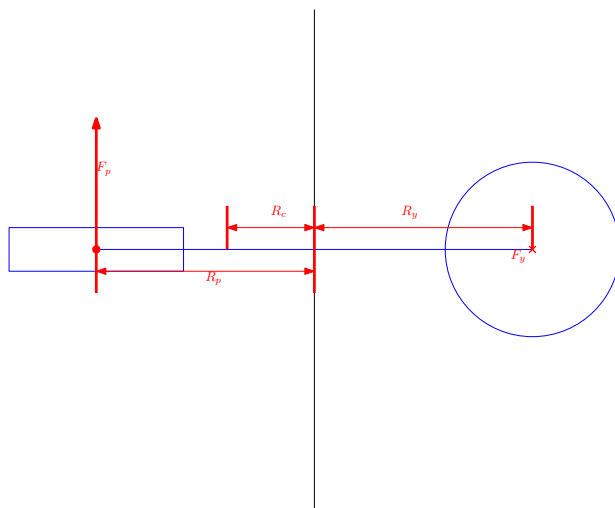


Figure 3.5: General free body diagram of the Quanser AERO.

The symbols represented in Figure 3.2 can be described as follows.

- $R_p$  is the distance from the fork of the Quanser AERO to the center of the main propeller.
- $R_y$  is the distance from the fork of the Quanser AERO to the center of the tail propeller.
- $R_c$  is the distance from the fork of the Quanser AERO to the center of mass of the body. The body is the structure connecting and including the main and tail propellers.
- $F_p$  is the force produced by the main rotor causing lift. We will assume that when the helicopter is rising,  $F_p$  is positive.
- $F_y$  is the force produced by the tail rotor causing thrust. We will assume that when the helicopter is rotating counter-clockwise,  $F_y$  is positive.

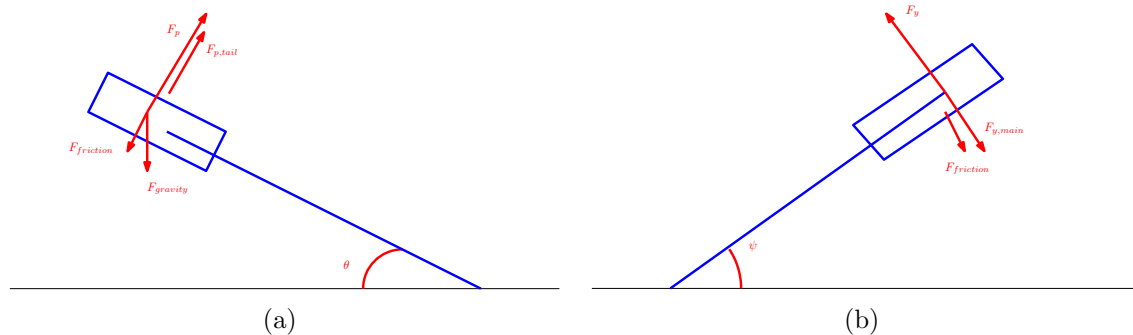


Figure 3.6: (a) Forces acting on the propeller controlling the pitch of the Quanser AERO. (b) Forces acting on the propeller controlling the yaw of the Quanser AERO.

We can look at the forces specifically associated with the main rotor as in Figure 3.6(a). The symbols represented in Figure 3.6(a) can be described as follows.

- $F_p$  is the force produced by the main rotor causing lift. We will assume that when the helicopter is rising,  $F_p$  is positive.
- $F_{p,tail}$  is the coupled force generated by the tail rotor. To visualize this force, think of the torques on the tail rotor. As the tail propeller spins, the propeller is causing torque on the actual motor shaft. This torque is translated to the pitch axis. We assume this coupling force is aiding the main rotor force as in [1].
- $F_{friction}$  is the frictional force opposing pitch.
- $F_{gravity}$  is the gravitational force exerted on the main rotor. This force only exists in the vertical plane.

We can look at the forces specifically associated with the main rotor as in Figure 3.6(b). The symbols represented in Figure 3.6(b) can be described as follows.

- $F_y$  is the force produced by the tail rotor causing thrust. We will assume that when the helicopter is rotating counter-clockwise,  $F_y$  is positive.
- $F_{y,main}$  is the coupled force generated by the main rotor. Again, we used the directions used in [1].
- $F_{friction}$  is the frictional force opposing yaw.

We know that the forces in each of the planes are tangential to the path of motion except the force of gravity. This means we can find the torques associated with each of the forces. Let the torques be defined as follows.

- $\tau_{p,main}$  is the torque associated with the force generated by the main rotor on the main rotor.

- $\tau_{p,tail}$  is the torque associated with the force generated by the tail rotor on the main rotor.
- $\tau_{p,friction}$  is the frictional torque opposing a change in pitch.
- $\tau_{p,gravity}$  is the torque associated with the force of gravity on the main rotor.
- $\tau_{y,tail}$  is the torque associated with the force generated by the tail rotor on the tail rotor.
- $\tau_{y,main}$  is the torque associated with the force generated by the main rotor on the tail rotor.
- $\tau_{y,friction}$  is the frictional torque opposing a change in yaw.
- $\tau_{p,net}$  is the net torque on the main rotor.
- $\tau_{y,net}$  is the net torque on the tail rotor.

We also know that the net torque is the rotational inertia times the tangential angular acceleration. Let the rotational inertia for the the main rotor be  $J_p$ , and let the rotational inertia for the tail rotor be  $J_y$ . The angular acceleration for the pitch and yaw are  $\ddot{\theta}$  and  $\ddot{\psi}$ , respectively.

$$\tau_{p,net} = J_p \ddot{\theta} \quad (3.14a)$$

$$\tau_{y,net} = J_y \ddot{\psi} \quad (3.14b)$$

Now we can calculate the individual torques because we know that torque is the tangential force times the distance from the pivot. Let us also make one assumption here. Let us assume that the torque generated on the main (tail) rotor from the main (tail) rotor is directly proportional to  $V_p$  and  $V_y$ , respectively. The proportional gains here will be classified as thrust gains.

$$\tau_{p,main} = R_p F_p = K_{p,main} V_p \quad (3.15a)$$

$$\tau_{p,main} = G_p \quad (3.15b)$$

$$\tau_{p,main} = R_p \beta_p \dot{\theta} \quad (3.15c)$$

$G_p$  is the torque coupling which we have yet determined. The tangential frictional force is the equal to a damping coefficient,  $\beta_p$ , times the velocity,  $\dot{\theta}$ .

$$\tau_{p,main} = R_y F_y = K_{y,tail} V_y \quad (3.16a)$$

$$\tau_{p,main} = G_y \quad (3.16b)$$

$$\tau_{p,main} = R_y \beta_y \dot{\psi} \quad (3.16c)$$

$$\tau_{p,gravity} = R_c m_{body} g \sin(\theta) \quad (3.16d)$$

$G_y$  is the torque coupling which we have yet determined. The tangential frictional force is equal to a damping coefficient,  $\beta_y$ , times the velocity,  $\dot{\psi}$ . The gravitational torque is found at the center of mass which is  $R_c$  from the pivot. The gravitational force is then the mass of the body and the two propellers.  $g$  is the acceleration due to gravity.

We can now combine Equations 3.14, 3.15, and 3.16 using the same sign conventions as used in Figures 3.6(a) and 3.6(b).

$$J_p \ddot{\theta} = K_{p,main} V_p + G_p - R_p \beta_p \dot{\theta} - R_c m_{body} g \sin(\theta) \quad (3.17a)$$

$$J_y \ddot{\psi} = K_{y,tail} V_y - G_y - R_y \beta_y \dot{\psi} \quad (3.17b)$$

We can rewrite Equations 3.17a and 3.17b with the second derivative state variables set alone.

$$\ddot{\theta} = \frac{K_{p,main} V_p}{J_p} + \frac{G_p}{J_p} - \frac{R_p \beta_p \dot{\theta}}{J_p} - \frac{R_c m_{body} g \sin(\theta)}{J_p} \quad (3.18a)$$

$$\ddot{\psi} = \frac{K_{y,tail} V_y}{J_y} - \frac{G_y}{J_y} - \frac{R_y \beta_y \dot{\psi}}{J_y} \quad (3.18b)$$

We can now write Equation 3.18 in state-space form.

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{R_p \beta_p}{J_p} & 0 \\ 0 & 0 & 0 & -\frac{R_y \beta_y}{J_y} \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{K_{p,main}}{J_p} & 0 \\ 0 & \frac{K_{y,tail}}{J_y} \end{bmatrix} \begin{bmatrix} V_p \\ V_y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{G_p}{J_p} - \frac{R_c m_{body} g \sin(\theta)}{J_p} \\ -\frac{G_y}{J_y} \end{bmatrix} \quad (3.19)$$

At this point, we shall make a few more assumptions to obtain a linearized model of the 2-DOF helicopter. ADP, in theory, should be able to withstand the inconsistencies between



the linearized model and the physical system. Let us assume that we do not the directions of the coupling forces acting on each of the rotors. This means we do not know the direction of  $G_p$  and  $G_y$ . Let us also assume that these torques are proportional to the voltages that generate them,  $V_p$  and  $V_y$ . This means we can set up two more equations in terms of thrust gains as we did in Equations 3.15a and 3.16a.

$$G_p = K_{p,tail}V_y \quad (3.20a)$$

$$G_p = K_{y,main}V_p \quad (3.20b)$$

Let us also use the small angle approximation method to get rid of the sinusoidal term. This means  $\sin(\theta) = \theta$  for small angles. Combining assumptions, we can rewrite the state-space model in a linear form.

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{R_c m_{body} g}{J_p} & 0 & -\frac{R_p \beta_p}{J_p} & 0 \\ 0 & 0 & 0 & -\frac{R_y \beta_y}{J_y} \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{K_{p,main}}{J_p} & \frac{K_{p,tail}}{J_p} \\ \frac{K_{y,main}}{J_y} & \frac{K_{y,tail}}{J_y} \end{bmatrix} \begin{bmatrix} V_p \\ V_y \end{bmatrix} \quad (3.21)$$

We can calculate the rotational inertias because we know the Quanser AERO configuration.

$$J_p = I_{body} + 2I_{prop} = \frac{m_{body}L_{body}^2}{12} + 2m_{prop}r_{prop}^2 \quad (3.22a)$$

$$J_y = I_{body} + 2I_{prop} + I_{yoke} = \frac{m_{body}L_{body}^2}{12} + 2m_{prop}r_{prop}^2 + \frac{m_{yoke}r_{fork}^2}{2} \quad (3.22b)$$

Finally, we can add some notations to make the state-space model a little more concise. Let us define the stiffness about the pitch axis as  $K_{sp} = R_c m_{body} g$ . Let us define the damping constant about the pitch axis as  $D_p = R_p \beta_p$ . Let us define the damping constant about the yaw axis as  $D_y = R_y \beta_y$ .

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{K_{sp}}{J_p} & 0 & -\frac{D_p}{J_p} & 0 \\ 0 & 0 & 0 & -\frac{D_y}{J_y} \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{K_{p,main}}{J_p} & \frac{K_{p,tail}}{J_p} \\ \frac{K_{y,main}}{J_y} & \frac{K_{y,tail}}{J_y} \end{bmatrix} \begin{bmatrix} V_p \\ V_y \end{bmatrix} \quad (3.23)$$

For conciseness, we will also rename the thrust gains. The first term of the subscript will be the rotor that is affected [pitch (main) or yaw (tail)], and the second term of the subscript will be the rotor causing the effect [pitch (main) or yaw (tail)].

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{K_{sp}}{J_p} & 0 & -\frac{D_p}{J_p} & 0 \\ 0 & 0 & 0 & -\frac{D_y}{J_y} \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{K_{pp}}{J_p} & \frac{K_{py}}{J_p} \\ \frac{K_{yp}}{J_y} & \frac{K_{yy}}{J_y} \end{bmatrix} \begin{bmatrix} V_p \\ V_y \end{bmatrix} \quad (3.24)$$

The derived state-space model is the same as the one derived by Quanser documentation provided in the licensed software package resources. The only item left is to determine the constants used in the state-space model. Some of the constants can be calculated and others need to be measured. The rotational inertias and the stiffness can be calculated by provided product features, but the damping constants and the thrust gains need to be measured. Quanser documentation provides experimental procedures to measure these values including the signs for the thrust gains. The constants calculated and measured by Quanser for the Quanser AERO can be seen in Table 3.2. With the parameters known, we

Table 3.2: Parameters calculated and measured by Quanser.

Parameter	Value	Unit
$J_p$	0.0215	[kg m <sup>2</sup> ]
$J_y$	0.0237	[kg m <sup>2</sup> ]
$D_p = R_p \beta_p$	0.00711	[N m s rad <sup>-1</sup> ]
$D_y = R_y \beta_y$	0.0220	[N m s rad <sup>-1</sup> ]
$K_{sp} = R_c m_{\text{body}} g$	0.0375	[N m rad <sup>-1</sup> ]
$K_{pp}$	0.0011	[N m V <sup>-1</sup> ]
$K_{py}$	0.0021	[N m V <sup>-1</sup> ]
$K_{yy}$	0.0022	[N m V <sup>-1</sup> ]
$K_{yp}$	-0.0027	[N m V <sup>-1</sup> ]

can calculate the numeric state-space model of the Quanser AERO configured as a 2-DOF helicopter as defined the Quanser measurements.

$$\begin{bmatrix} \ddot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1.7442 & 0 & -0.3307 & 0 \\ 0 & 0 & 0 & -0.9283 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.0512 & 0.0977 \\ -0.1139 & 0.0928 \end{bmatrix} \begin{bmatrix} V_p \\ V_y \end{bmatrix} \quad (3.25)$$

# Chapter 4

## MATLAB Simulations

In any electrical engineering project, one must perform theoretical research which includes a mathematical analysis before they can continue to implementation. That research was performed in the previous chapters. The next step is combining the theoretical aspect with the defined system, the Quanser AERO. We cannot go straight to implementation because we need to verify that the ADP algorithm does, in fact, behave as intended for the Quanser AERO. In our case, the most viable option was to perform MATLAB simulations using the derived models discussed in Chapter 3 and the ADP algorithm discussed in Chapter 2.

The MATLAB simulation code can be seen in Appendix A.1. The MATLAB simulation can be ran for both the half-quadcopter and the 2-DOF helicopter configurations. Both ADP and LQR are ran in a single simulation to compare the results of the two control methods. The desired trajectory can be set, and then both control approaches are ran for the state-space model.

A few assumptions were made when writing the MATLAB simulation code, and these assumptions can be seen in the code provided. Parameters discussed in Chapter 3 were used to describe the state-space models. No measurements were used from our physical Quanser AERO. Quanser documentation also provided reasonable matrices for  $\mathbf{R}$  and  $\mathbf{Q}$  used in the LQR control approach. These matrices were adjusted slightly to make them positive semi-definite and then used for both ADP and LQR. Sampling times of  $\tau = 0.01$  and  $T = 0.2$  seconds were used for timing. These constraints were provided by Dr. Miah at the beginning of the project.

The disadvantage of using MATLAB is the state-space models used for the system are linear. We derived linear state-space models in Chapter 3. We cannot simulate a truly nonlinear system for the Quanser AERO because the coupling is described as linear. In essence, the MATLAB simulations only provide a comparison of how ADP and LQR control a linear system. One would suspect both of them to perform well given the fact that LQR is designed for linear systems. The real question is whether ADP will out-perform LQR for a linear system. To try to mimic a nonlinear system, the MATLAB simulation code has a

nonlinear adjustment factor. This factor adds a random few degrees to the output of the system every sampling time. The idea of this factor is to try to mimic some pseudo-random noise or nonlinearity to the system. This factor is adjustable, but it is currently set to zero in Appendix A.1.

For both the half-quadcopter and 2-DOF helicopter, three MATLAB simulations are provided. The initial conditions are the same for all of the MATLAB simulations. The first MATLAB simulation is setting the desired pitch and yaw to a constant value. The second MATLAB simulation imposes time-varying constant trajectories for both the desired pitch and yaw. The last MATLAB simulation provided imposes sinusoidal desired trajectories for both the pitch and yaw. The desired angular velocities are always zero.

This chapter is organized as follows. Section 4.1 provides and analyzes the MATLAB simulation results for the half-quadcopter. MATLAB simulation results for the 2-DOF helicopter are provided and analyzed in Section 4.2.

## 4.1 Half-Quadcopter

### 4.1.1 Results

The results of the MATLAB simulations for the half-quadcopter can be seen in Figures 4.1, 4.2, and 4.3.

### 4.1.2 Analysis

The MATLAB simulation results for constant desired pitch and yaw can be seen in Figure 4.1. The angular positions for both ADP and LQR can be seen in Figure 4.1(a). The MATLAB simulation shows that both control techniques have convergent pitch and yaw values. It does appear that the ADP technique does oscillate initially slightly more than LQR, but ADP does reach the desired values. The same results can be seen with the angular velocities shown in Figure 4.1(b). The voltage inputs seen in Figure 4.1(e) show similar traits for ADP and LQR, but no technique has a distinct advantage.

For the time-varying constant desired trajectories, the MATLAB simulations can be seen in Figure 4.2. The angular position seen in Figure 4.2(a) shows faster convergence for LQR and large initial oscillations for ADP. The angular velocities seem to have difficulty converging to zero because of the changing trajectories, but both ADP and LQR have similar traits as seen in Figure 4.2(b). Notice that the voltage inputs of opposite motors mirror each other. Again for the voltage inputs, there is no clear advantage for either ADP or LQR.

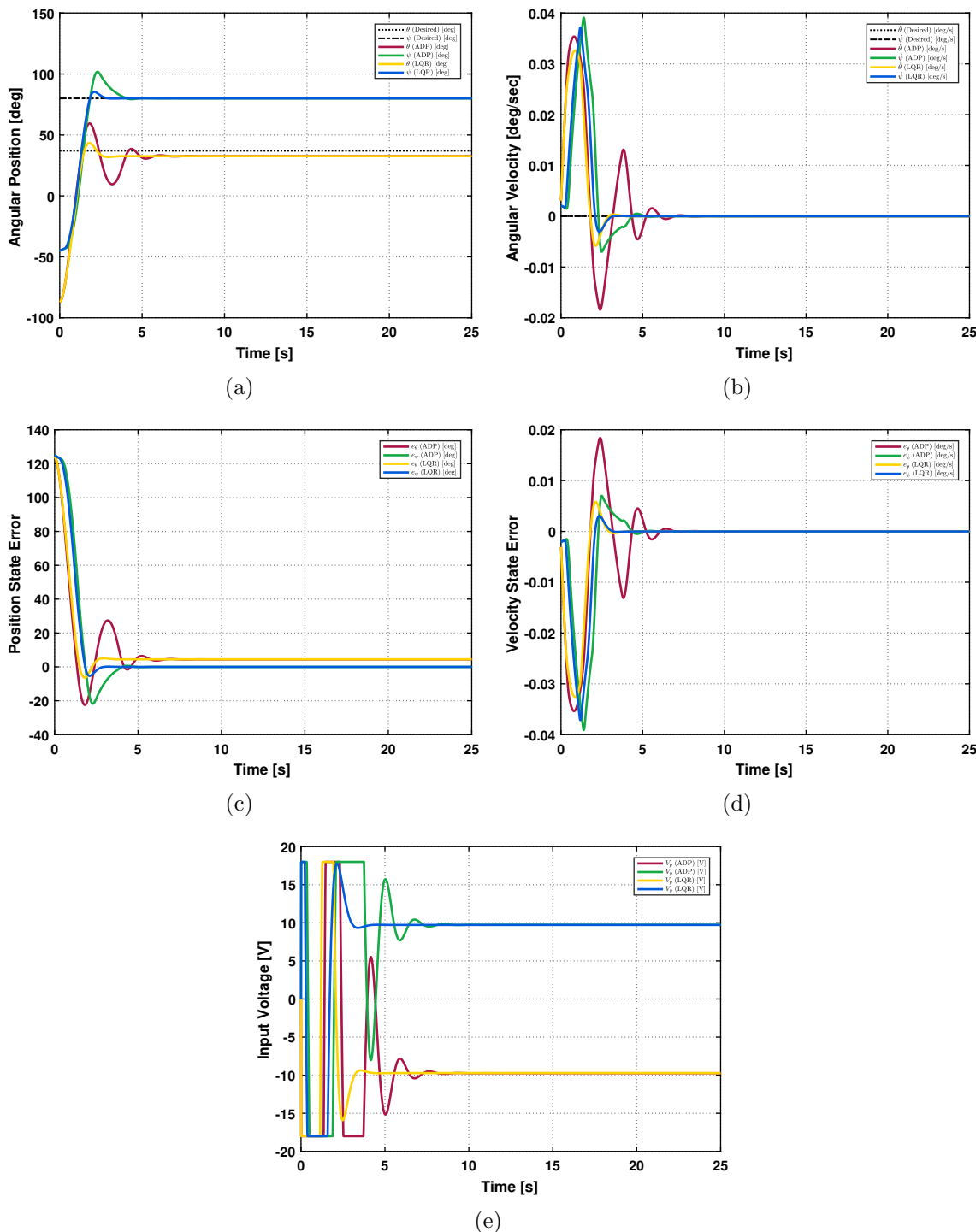


Figure 4.1: Constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

The most surprising MATLAB simulation is the sinusoidal desired trajectory as seen in Figure 4.3. For the angular position, we see the initial oscillations from ADP, while LQR

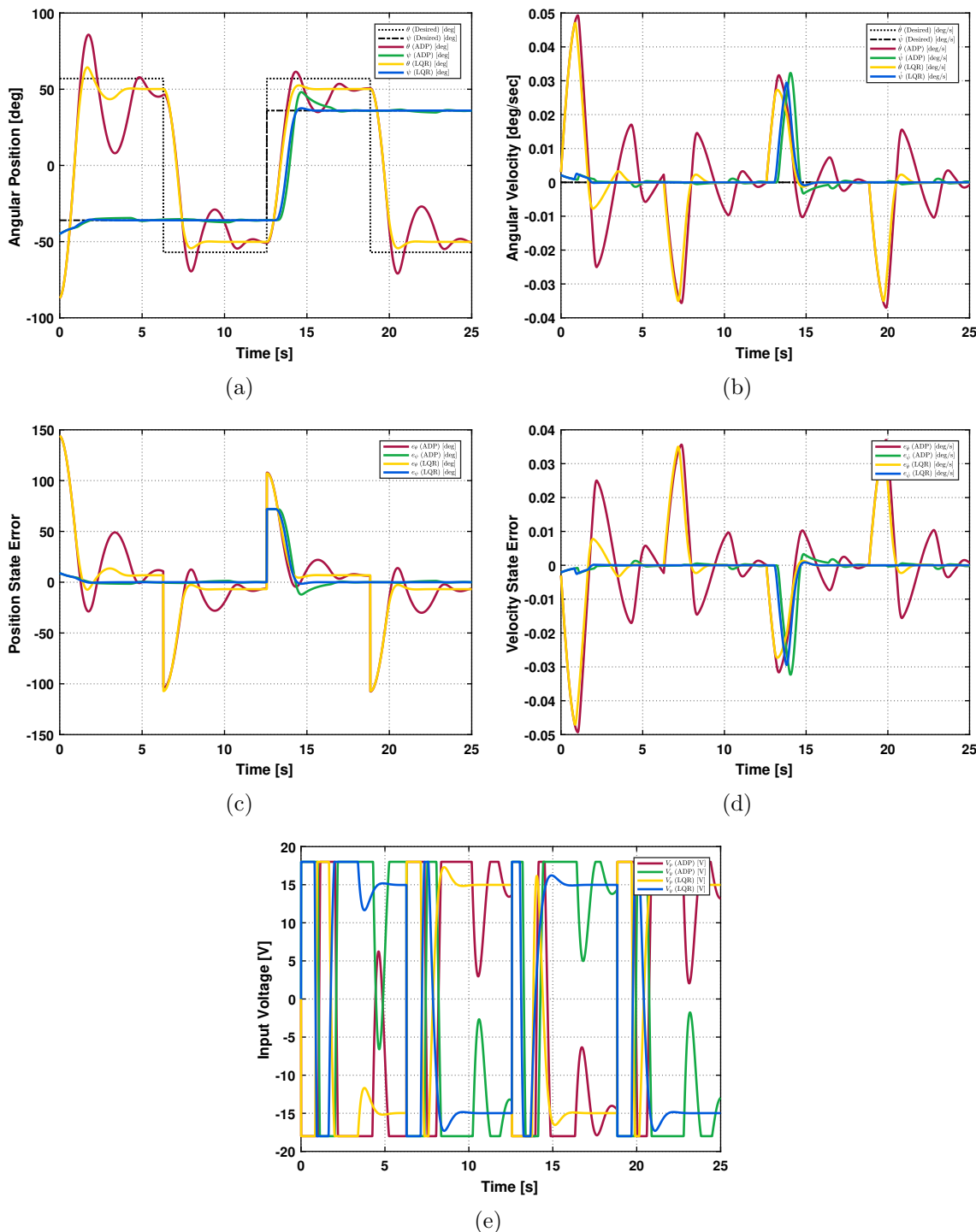


Figure 4.2: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

seems to converge quicker which can be seen in Figure 4.3(a). The difference with this MATLAB simulation is that ADP actually appears closer to the desired trajectory than

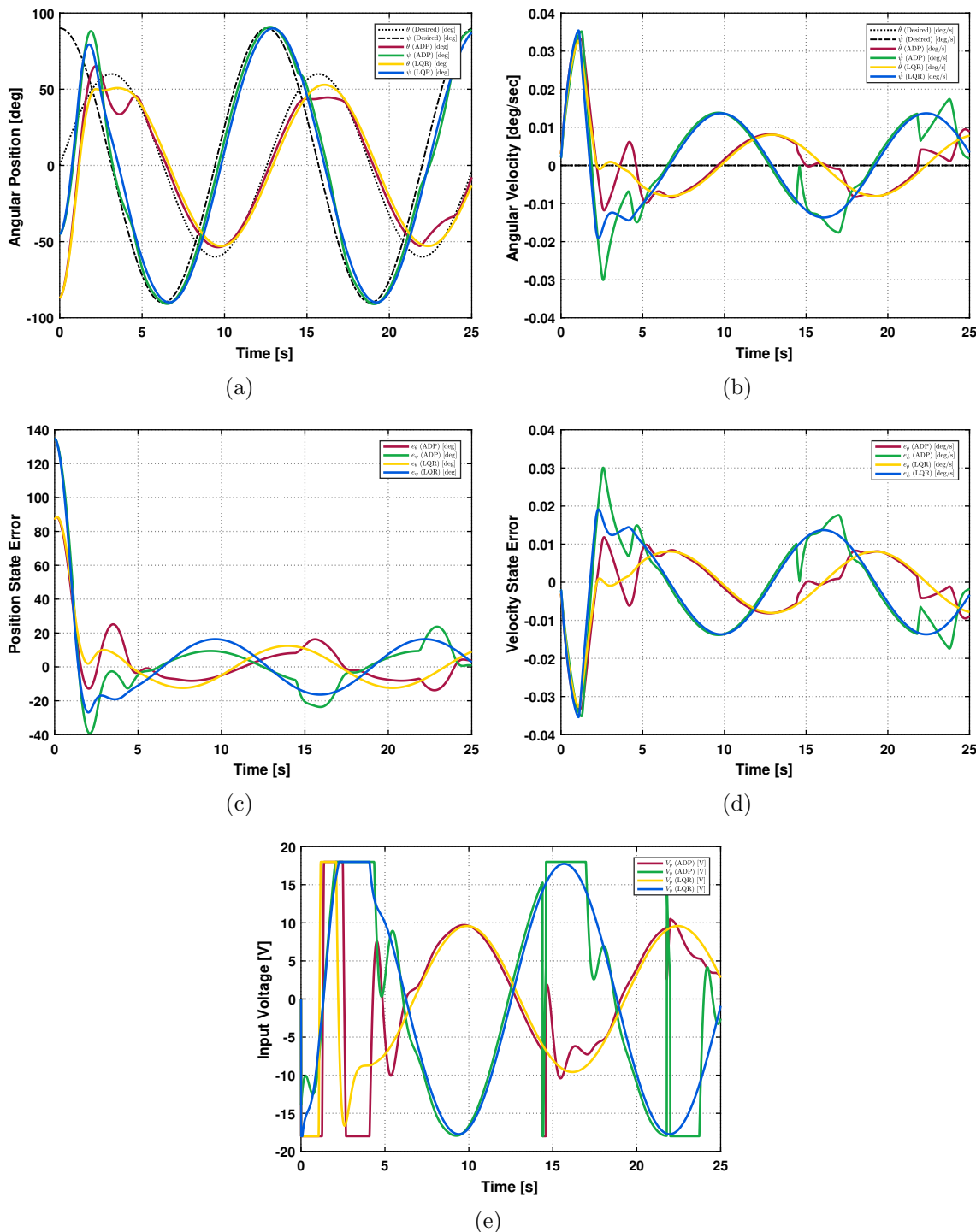


Figure 4.3: Sinusoidal desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

LQR. This can be verified in the angular position error plot seen in Figure 4.3(c). As expected, the angular velocity is no where close to zero because we have a changing desired

trajectory. Also, there is again no clear advantage for either ADP or LQR in terms of input voltage as seen in Figure 4.3(e).

Overall, both ADP and LQR perform well for the linear system of the half-quadcopter as seen in the MATLAB simulation results. We usually see more initial oscillation from ADP than LQR, but this might be the algorithm learning the system. Both control techniques, however, converge to the desired trajectories. On the down side, there is no clear advantage to ADP over LQR for a linear system as hoped for in the beginning of this project.

## 4.2 2-DOF Helicopter

### 4.2.1 Results

The results of the MATLAB simulations for the 2-DOF helicopter can be seen in Figures 4.4, 4.5, and 4.6.

### 4.2.2 Analysis

The MATLAB simulation results for constant desired pitch and yaw can be seen in Figure 4.4. The angular positions for both ADP and LQR can be seen in Figure 4.4(a). The MATLAB simulation shows that both control techniques have convergent pitch and yaw values. It does appear that the ADP technique does oscillate initially much more than LQR, but ADP does reach the desired values. The same results can be seen with the angular velocities shown in Figure 4.4(b). The voltage inputs seen in Figure 4.4(e) show similar traits for ADP and LQR, but no technique has a distinct advantage.

For the time-varying constant desired trajectories, the MATLAB simulations can be seen in Figure 4.5. The results of the time-varying trajectories are very similar to those of the constant desired trajectories. The angular position seen in Figure 4.5(a) shows faster convergence for LQR and slight initial oscillations for ADP. This is exactly what we saw in the constant desired trajectories. The angular velocities seem to have difficulty converging to zero because of the changing trajectories, but both ADP and LQR have similar traits as seen in Figure 4.5(b). Again for the voltage inputs, there is no clear advantage for either ADP or LQR.

The most surprising MATLAB simulation is the sinusoidal desired trajectory as seen in Figure 4.6. For the angular position, we see the initial oscillations from ADP, while LQR seems to converge quicker which can be seen in Figure 4.6(a). The difference with this MATLAB simulation is that ADP actually appears closer to the desired trajectory than LQR. This can be verified in the angular position error plot seen in Figure 4.6(c). As expected, the angular velocity is no where close to zero because we have a changing desired trajectory. Also, there is again no clear advantage for either ADP or LQR in terms of input voltage as seen in Figure 4.6(e).



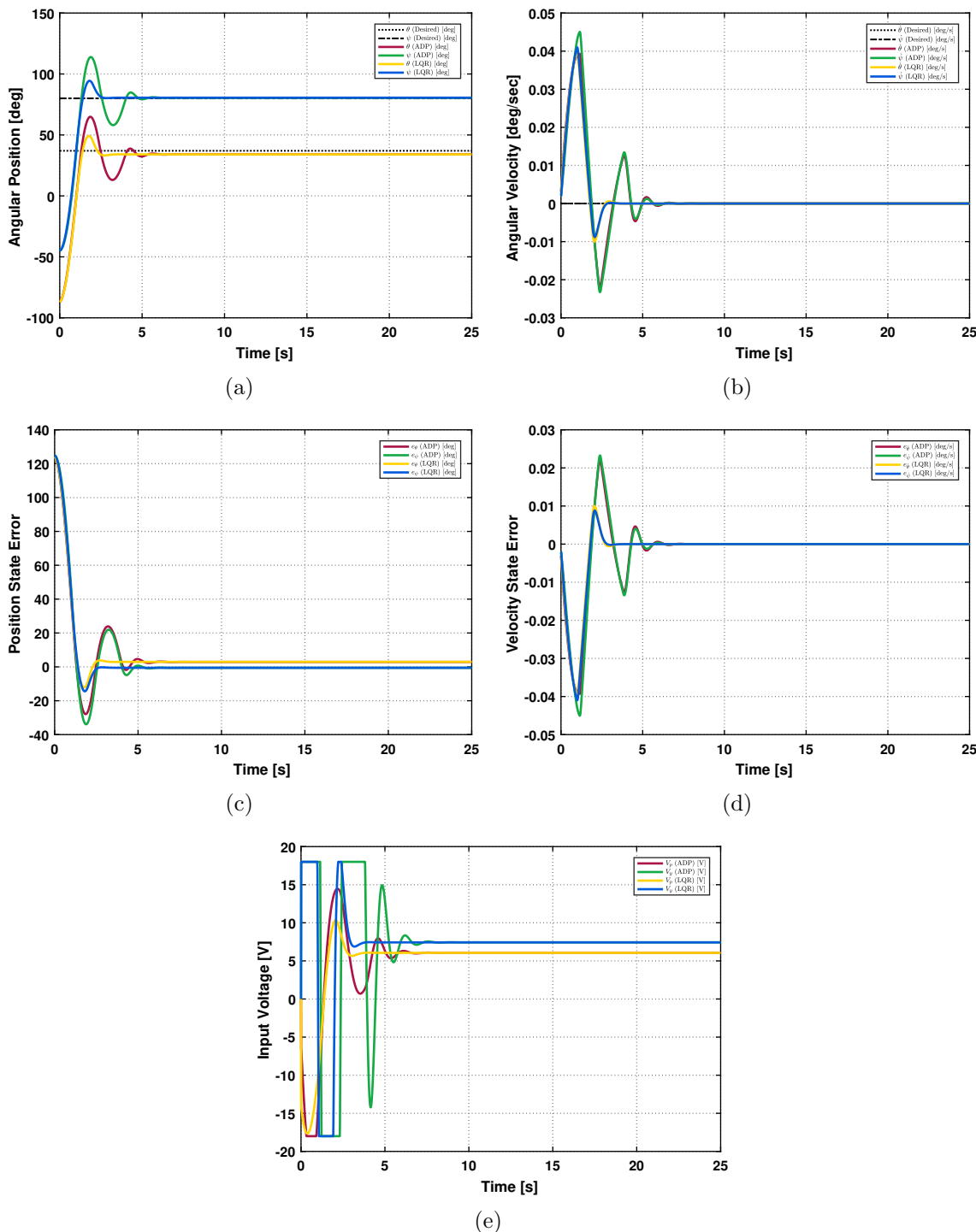


Figure 4.4: Constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

Overall, both ADP and LQR perform well for the linear system of the 2-DOF helicopter as seen in the MATLAB simulation results. We usually see more initial oscillation from ADP

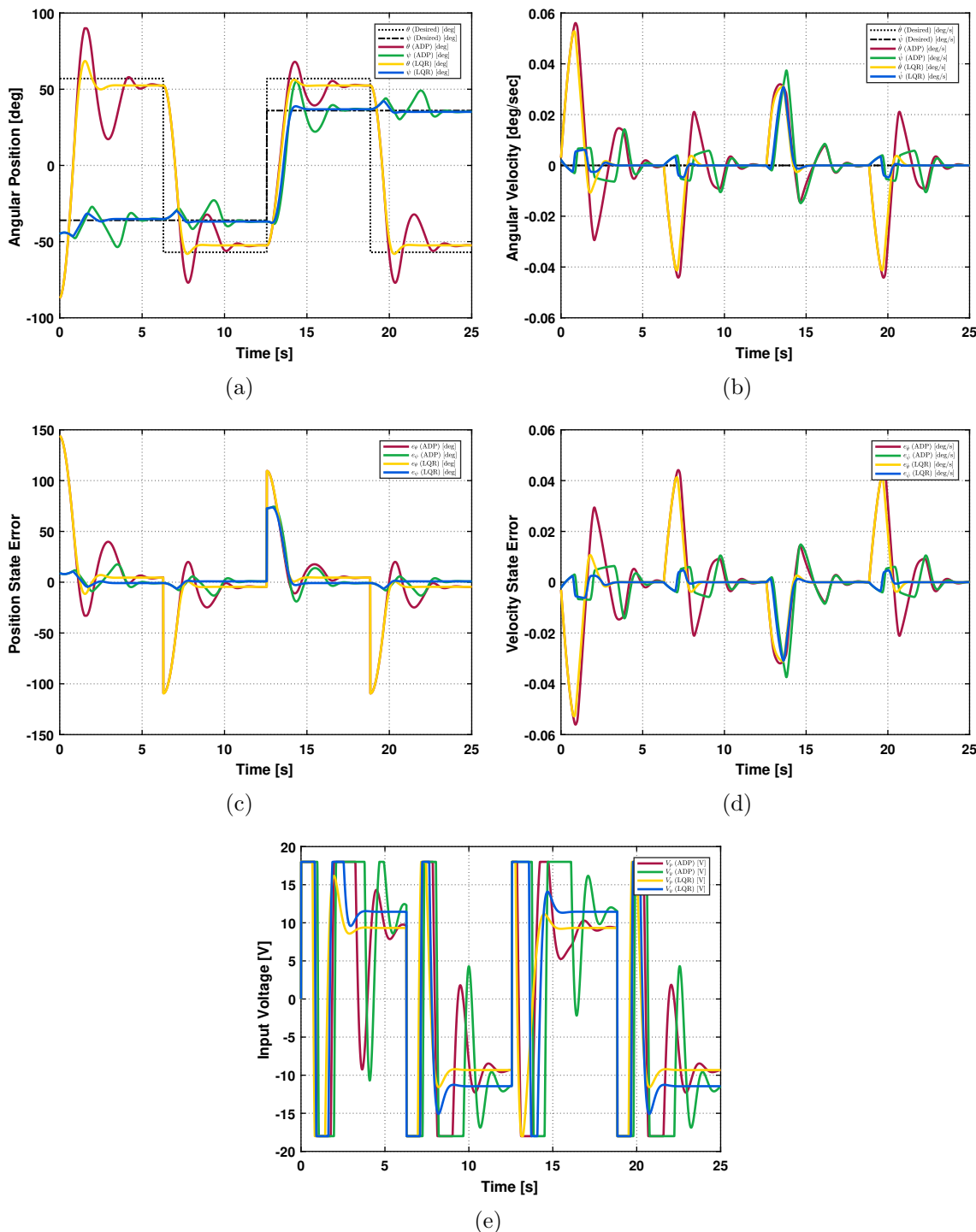


Figure 4.5: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

than LQR, but this might be the algorithm learning the system. Both control techniques, however, converge to the desired trajectories. On the down side, there is no clear advantage

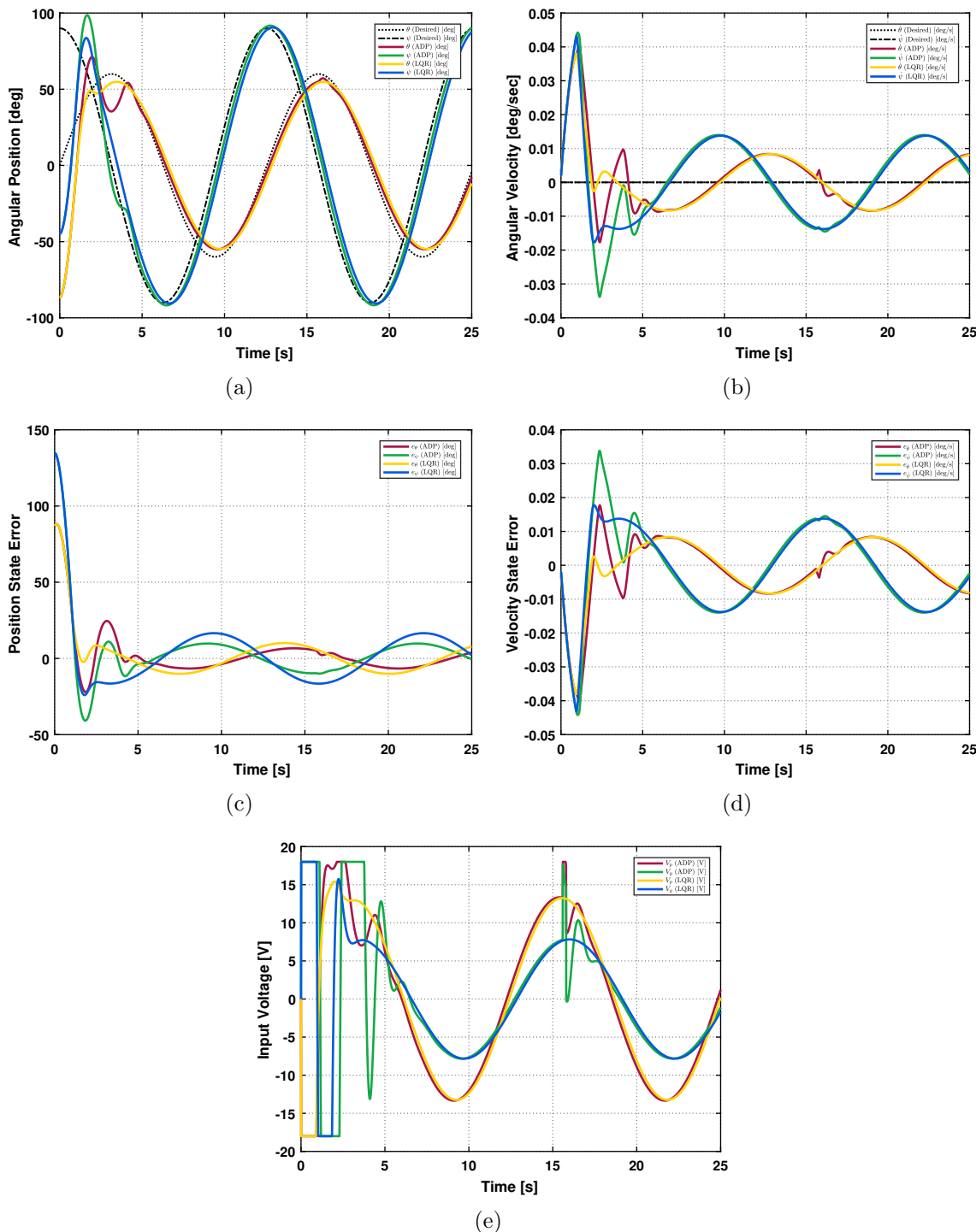


Figure 4.6: Sinusoidal desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

to ADP over LQR for a linear system as hoped for in the beginning of this project.

# Chapter 5

## Real-Time Simulation

V-REP, Virtual Robot Experimentation Platform, is a program made by Coppelia Robotics that offers an integrated development environment for prototyping robots and testing control algorithms on different robots among many other capabilities.

On the most basic level, V-REP offers an environment, called a “scene,” in which the user places components of a robot, such as joints, sensors, grabbers, and basic shapes. When implemented, these items are called “scene objects.” Each scene object can be assigned code called by V-REP at each time step in a simulation. The code, written in the LUA language, allows the user to use V-REP’s built-in API to have scene objects interact, governed by the different physics engines available within V-REP.

V-REP can be run solely by internal LUA code, and many of the provided robots come with some short script attached to each robot demonstrating functionality. One of the deliverables for this project is to have V-REP be controlled remotely by MATLAB.

This chapter is organized as follows. Section 5.1 discusses the software used throughout this project pertaining to V-REP. Section 5.2 discusses the data transfer between MATLAB and V-REP. Section 5.3 discusses the design of the Quanser AERO in V-REP. Section 5.3.1 discusses the design of a working Quanser AERO model, and Section 5.3.2 discusses a model still under development. Section 5.4 discusses the results including RMSE measurements and the plots from simulations performed throughout the project.

### 5.1 Required Software

At the time of this publication, the software simulations were performed using V-REP PRO EDU V3.5.0.4 and MATLAB 2016B. It is not required, but it is recommended to use the Vortex Physics engine to control the V-REP simulation.

**NOTE:** Different versions of V-REP may affect model functionality.

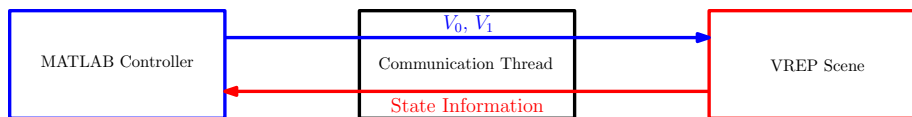


Figure 5.1: Communication scheme between MATLAB (acting as controller) and V-REP (acting as environment).

## 5.2 Communication

V-REP and MATLAB communicate through a communication thread defined at port 19999. This process is shown in Figure 5.1. Next, the V-REP API offers a function to package a MATLAB vector and send that through the port for V-REP to read. Once V-REP reads the signal, it clears the signal, reading it for the next MATLAB assertion. A simple MATLAB and LUA setup are provided in Appendices B.2.1 and B.2.2.

## 5.3 Designing Quanser AERO

The V-REP design started with considering the deliverables. The initial design was to use the V-REP quadcopter model provided. To adapt this model to the Quanser AERO system, the center was to be set at a fixed point, while two of the four propellers were to be used to actuate the system. This plan was modified to a two pronged approach at designing and implementing a Quanser AERO base, each utilizing different API function calls within V-REP. The first design will make use of the linearized model and exert control in V-REP via API calls concerning the pitch and yaw joints, configured as motors. The second model will make use of the motor circuit to calculate thrust and resultant torques.

Clearly, factors such as the weight, size, and location of each shape that comprise the V-REP model are important to accurately define the  $\mathbf{A}$  and  $\mathbf{B}$  matrices of the state-space model. The model uses only cylinders and rectangular prisms such that algebraic moments of inertia could be utilized for convenience. To compute the moment of inertia of a cylinder, Equation 5.1 is used. To compute the moment of inertia of a rectangular prism, Equation 5.2 is used. To compute the moment of inertia of a cylinder or rectangular prism that is off of its axis of rotation by a distance  $d$ , the Parallel Axis Theorem, Equation 5.3, is used.

$$I_{cyl} = m * r^2 \quad (5.1)$$

$$I_{prism} = \frac{m}{12}(a^2 + b^2) \quad (5.2)$$

$$I_{PAX} = I + md^2 \quad (5.3)$$

Table 5.1 displays the similarity of the moment of inertia about  $\theta$  and  $\psi$  for V-REP and the Quanser documentation values. Code to calculate the moment of inertia for the V-REP Quanser AERO in yaw and pitch is included in Appendix B.3.

Table 5.1: Comparing V-REP and Quanser AERO Moments of Inertia

Parameter	Source	Value	Unit
$J_\theta$	Quanser	0.215	[kg m <sup>2</sup> ]
$J_\theta$	V-REP	0.0214	[kg m <sup>2</sup> ]
$J_\psi$	Quanser	0.237	[kg m <sup>2</sup> ]
$J_\psi$	V-REP	0.0244	[kg m <sup>2</sup> ]

### 5.3.1 Joint Control

In this approach, the V-REP model makes use of the model constants provided by Quanser. MATLAB is configured to send the optimized motor voltages decided by the control algorithm. V-REP then receives this signal, and sets the corresponding motor's input voltage in the simulation. LUA script then estimates the current angular velocities numerically by trapezoidal integration. The angular velocities are applied to the scene joints. The design of the LUA script attached to the Quanser AERO is shown in Figure 5.2.

In this design, the joints are configured as motors capable of receiving angular velocities from the V-REP API's `setJointTargetVelocity` function. In this configuration, V-REP will command the physics engine running the scene to apply torque to the selected joint until the target velocity is reached. V-REP will continue adjusting this torque to maintain the target velocity.

For the 2-DOF helicopter, we can write the angular accelerations as:

$$\ddot{\theta} = \frac{-K_{sp}}{J_\theta} \theta + \frac{-D_\theta}{J_\theta} \dot{\theta} + \frac{K_{pp}}{J_\theta} V_0 + \frac{K_{py}}{J_\theta} V_1 \quad (5.4a)$$

$$\ddot{\psi} = \frac{-D_\psi}{J_\psi} \dot{\psi} + \frac{K_{yp}}{J_\psi} V_0 + \frac{K_{yy}}{J_\psi} V_1 \quad (5.4b)$$

For the half-quadcopter, we can write the angular accelerations as:

$$\ddot{\theta} = \frac{-K_{sp}}{J_\theta} \theta + \frac{-D_\theta}{J_\theta} \dot{\theta} + \frac{-K_{pp}}{J_\theta} V_0 + \frac{K_{pp}}{J_\theta} V_1 \quad (5.5a)$$

$$\ddot{\psi} = \frac{-D_\psi}{J_\psi} \dot{\psi} + \frac{-K_{yy}}{J_\psi} V_0 + \frac{K_{yy}}{J_\psi} V_1 \quad (5.5b)$$

The LUA script for the half-quadcopter is included as Appendix B.4.1, and the 2-DOF helicopter script is included as Appendix B.4.2. Each of these scripts are attached to the base of the Quanser AERO in the V-REP scene. Table 5.2 provides the reader with information about the MATLAB script that runs each V-REP scene.

Table 5.2: File Sets

Model	MATLAB File	V-REP Scene
Half-quadcopter	B.4.3	AeroHalfquadcopter
2-DOF Helicopter	B.4.4	AeroHelicopter
Half-quadcopter	B.4.5	AeroHalfquadcopter
2-DOF Helicopter	B.4.6	AeroHelicopter

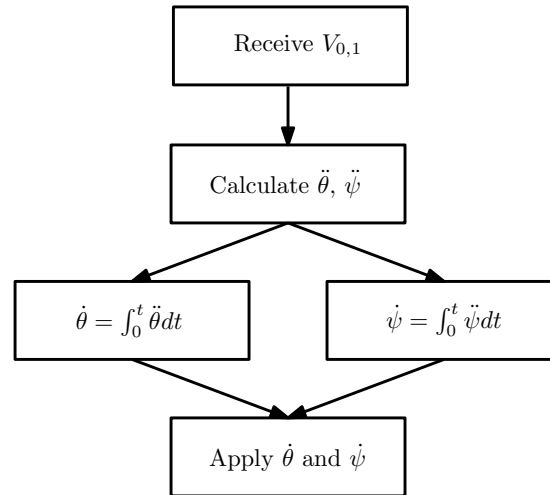


Figure 5.2: V-REP joint model flowchart.

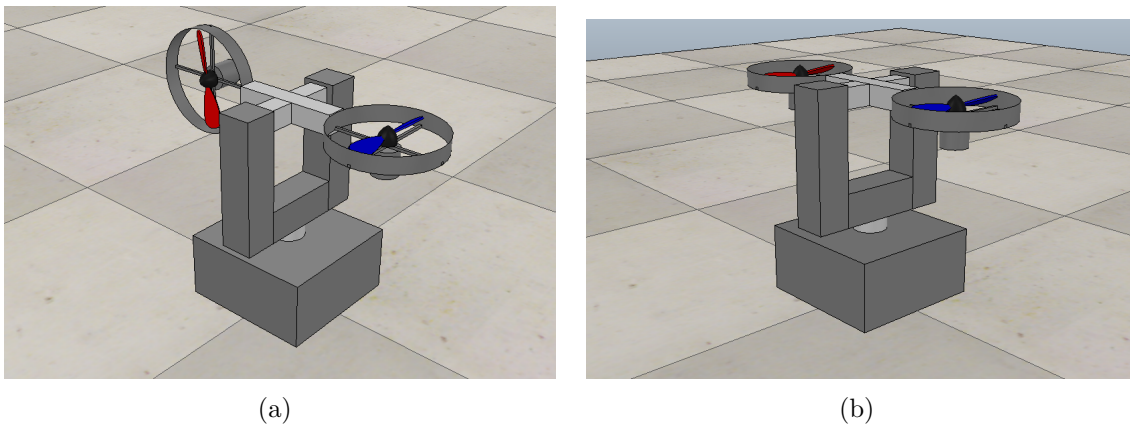


Figure 5.3: Quanser AERO developed for joint control. (a) 2-DOF helicopter mode. (b) Half-quadcopter mode.

### 5.3.2 Motor Control

For the motor control approach, the LUA script applies a force and a torque on the Quanser AERO at the location of the motor. This will simulate the propeller's lift and the motor's counter-torque on the Quanser AERO through V-REP's selected physics engine.

V-REP joints in this model must be configured as free spinning connections and contain

no internal friction. Therefore,  $D_\theta = D_\psi = 0$ . Further, the model is balanced about the pitch joint, therefore,  $K_{sp} = 0$ . Using these values, the state-space matrices for the 2-DOF helicopter become:

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.0515 & 0.0983 \\ -0.1105 & 0.0901 \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} \quad (5.6)$$

And for the half-quadcopter, the state-space matrices become:

$$\begin{bmatrix} \dot{\theta} \\ \dot{\psi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -0.0450 & 0.0450 \\ -0.0901 & 0.0901 \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} \quad (5.7)$$

Figure 5.5 shows the modeling circuit of the electric motors driving each propeller in the Quanser AERO. Both the ADP and LQR will decide an optimal voltage to regulate the error states to zero, so the goal will be to calculate the lift and torque from equations provided in Quanser literature.

First, we calculate the lifting force created by the propeller. We express  $\tau_\theta$ , assuming 90° application:

$$\tau_\theta = R \cdot F_0 \sin 90 - R \cdot F_1 \sin 90 = K_{pp}V_0 - K_{pp}V_1 \quad (5.8)$$

Therefore:

$$F_{prop0,1} = \frac{K_{pp}}{R} V_{0,1} \quad (5.9)$$

Next, the torque generated against the motor by the propeller dynamics is considered. V-REP provides a local coordinate system for the motor, and the torque is applied about the Z-axis. The model defines Motor#0 as having upwards thrust with a counterclockwise spin, and Motor#1 as having upwards thrust with a clockwise spin. Likewise, when the thrust changes direction as the polarity of voltage changes, the torque vector has to switch directions.

Quanser literature expresses drag torque as:

$$\tau_{drag} = k_d \omega_m \quad (5.10)$$

And gyroscopic torque as:

$$\tau_{gyro} = k_m \dot{i}_m \quad (5.11)$$

Creating a total torque about the motor's body frame Z-axis as:

$$\tau_z = k_d \omega_m + k_m \dot{i}_m \quad (5.12)$$

Using Equation 5.12, the input voltage from the control algorithm, and the motor current,  $i_m$ , the torque created by the motor can be modelled. However, we don't have an expression for the motor current,  $i_m$ , or the propeller speed,  $\omega_m$ .



To model the current in V-REP, a sweep was performed on the actual Quanser AERO with the Simulink model developed in Section 6.1. For this test, we enabled the yaw and pitch locks on the Quanser AERO, applied voltages in steps of 2V from -22 to 22 Volts, and logged the currents. A cubic function estimating ( $R^2 = 0.997$ ) the motor current, given the input voltage, was generated shown in Equation 5.13. The recorded data points and the current plot are shown in Figure 5.4.

$$i(v_{0,1}) = 2.77 \times 10^{-5}v^3 + -9.66 \times 10^{-6}v^2 + 2.13 \times 10^{-2}v + -3.19 \times 10^{-3} \quad (5.13)$$

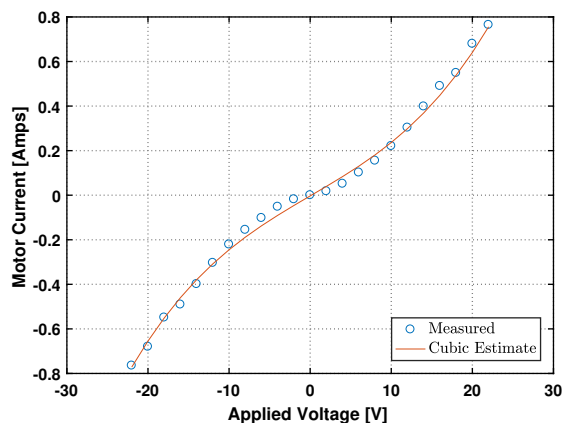


Figure 5.4: Measured currents and estimate function.

Using the current model allows for calculation of the torque from the drag force of the propeller and from the motor accelerating the propeller. Referencing Quanser literature, Equation 5.14 represents the back EMF, a relationship between the voltage across the motor and the motor speed.

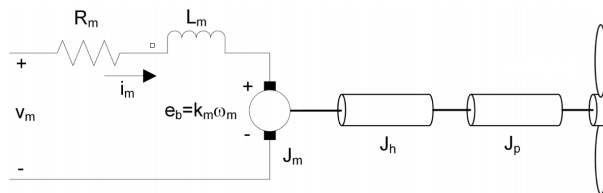


Figure 5.5: Circuit modeling Quanser AERO motors.

$$e_b = k_m \omega_m \quad (5.14)$$

Using Figure 5.5 and KVL:

$$v_m - R_m i_m - L_m \frac{di_m}{dt} - k_m \omega_m = 0 \quad (5.15)$$

Table 5.3: Parameter values shown in Figure 5.5

Parameter	Description	Value	Unit
$R_m$	Terminal Resistance	8.4	$[\Omega]$
$k_t$	Torque Constant	0.042	$[\text{N m A}^{-1}]$
$k_m$	Back EMF Constant	0.042	$[\text{V rad}^{-1} \text{s}^{-1}]$
$J_m$	Rotor Inertia	$4 \times 10^{-6}$	$[\text{kg m}^2]$
$L_m$	Rotor Inductance	0.00116	$[\text{H}]$
$k_d$	Drag Coefficient	$1 \times 10^{-5}$	$[\text{N m rad}^{-1} \text{s}^{-1}]$

Leading to:

$$\omega_m = \frac{v_m - R_m i_m - L_m \frac{di_m}{dt}}{k_m} \quad (5.16)$$

Equation 5.16 represents the model to find the propeller speed, given the input voltage, and the current. Therefore, using Equations 5.13, 5.15, 5.10, and 5.11 allows for estimated forces and torques to be calculated. Each propeller's LUA script is described by the flowchart in Figure 5.3.2.

The LUA script is implemented as follows. The Quanser AERO base receives Appendix B.5.1, then each motor gets its own script, Motor#0: Appendix B.5.2, Motor#1: Appendix B.5.3. For now, the only implementation of MATLAB code is provided as an untuned PID controller, provided as Appendix B.5.4. Thus far, LQR and ADP algorithms have not been applicable to the system.

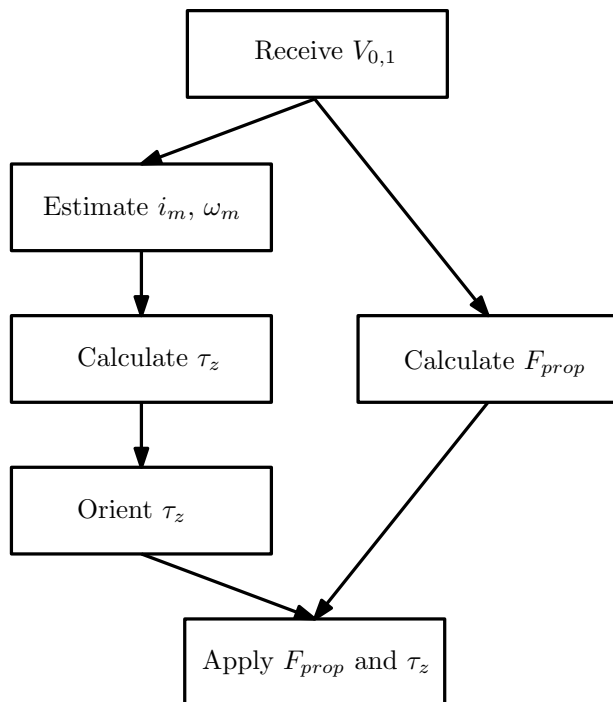


Figure 5.6: V-REP motor model flowchart.

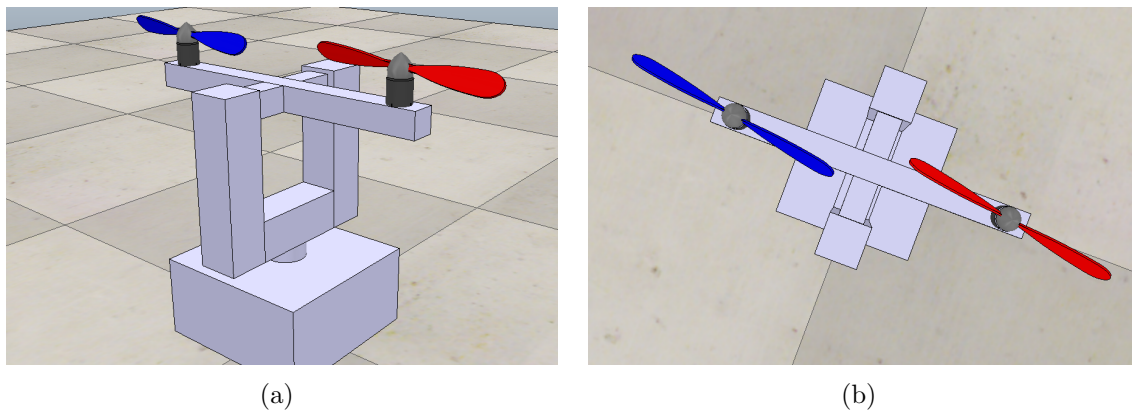


Figure 5.7: Quanser AERO developed for motor control. (a) Half-quadcopter mode (side). (b) Half-quadcopter mode (top).

## 5.4 Simulation Results

Figures 5.8, 5.9, 5.10, 5.11, 5.12, and 5.13 are presented to demonstrate the V-REP joint control model performing in both 2-DOF helicopter and half-quadcopter mode with the LQR algorithm acting as the controller. The results of the RMSE of the measurements are summarized in Table 5.4.

Figures 5.14, 5.15, 5.16, and 5.17 are presented to demonstrate the V-REP joint control model performing in both 2-DOF helicopter and half-quadcopter mode with ADP acting

as the controller.

We can see that LQR performs well for this particular system. ADP under-performs LQR most notably in smoothness; however, ADP does track the referenced angular trajectory. The results are similar to the MATLAB simulations, but there are more disturbances. The results of the RMSE measurements are summarized in Table 5.5.

Figures 5.18, 5.19 and 5.20 are provided as proof of concept for the motor modeled V-REP scene. In these simulations,  $K_p = 8$ ,  $K_i = 10$ , and  $K_d = 3$ .

### 5.4.1 LQR

Table 5.4: Experimental V-REP LQR angular RMSE measurements.

Figure	Configuration	$\theta$ LQR [deg]	$\psi$ LQR [deg]	$\dot{\theta}$ LQR [deg /s]	$\dot{\psi}$ LQR [deg /s]
5.8	2-DOF Helicopter	3.60063	1.42257	0.00034832	0.00082065
5.9	2-DOF Helicopter	0.984427	1.75581	0.000235371	0.000358396
5.10	2-DOF Helicopter	0.433482	0.309782	0.000207048	0.000369757
5.11	Half-quadcopter	5.03313	3.18276	0.000334748	0.00081096
5.12	Half-quadcopter	1.25609	1.15357	0.000226184	0.000364947
5.13	Half-quadcopter	0.643793	0.170658	0.000205815	0.000367207

### 5.4.2 ADP

Table 5.5: Experimental V-REP ADP angular RMSE measurements.

Figure	Configuration	$\theta$ ADP [deg]	$\psi$ ADP [deg]	$\dot{\theta}$ ADP [deg /s]	$\dot{\psi}$ ADP [deg /s]
5.14	2-DOF Helicopter	3.94776	1.655	0.000159409	0.00091622
5.15	2-DOF Helicopter	0.449811	1.58411	0.000406199	0.00139641
5.16	Half-quadcopter	6.58232	3.27046	0.00096847	0.00108425
5.17	Half-quadcopter	0.742075	1.95259	0.000410448	0.00106001

### 5.4.3 PID

### 5.4.4 Analysis

From the simulation results provided in Sections 5.4.1, 5.4.2, and 5.4.3, we can see that the V-REP models seem to be working correctly. There is, however, one difficulty associated with ADP compared to both LQR and PID control. Because of the computational

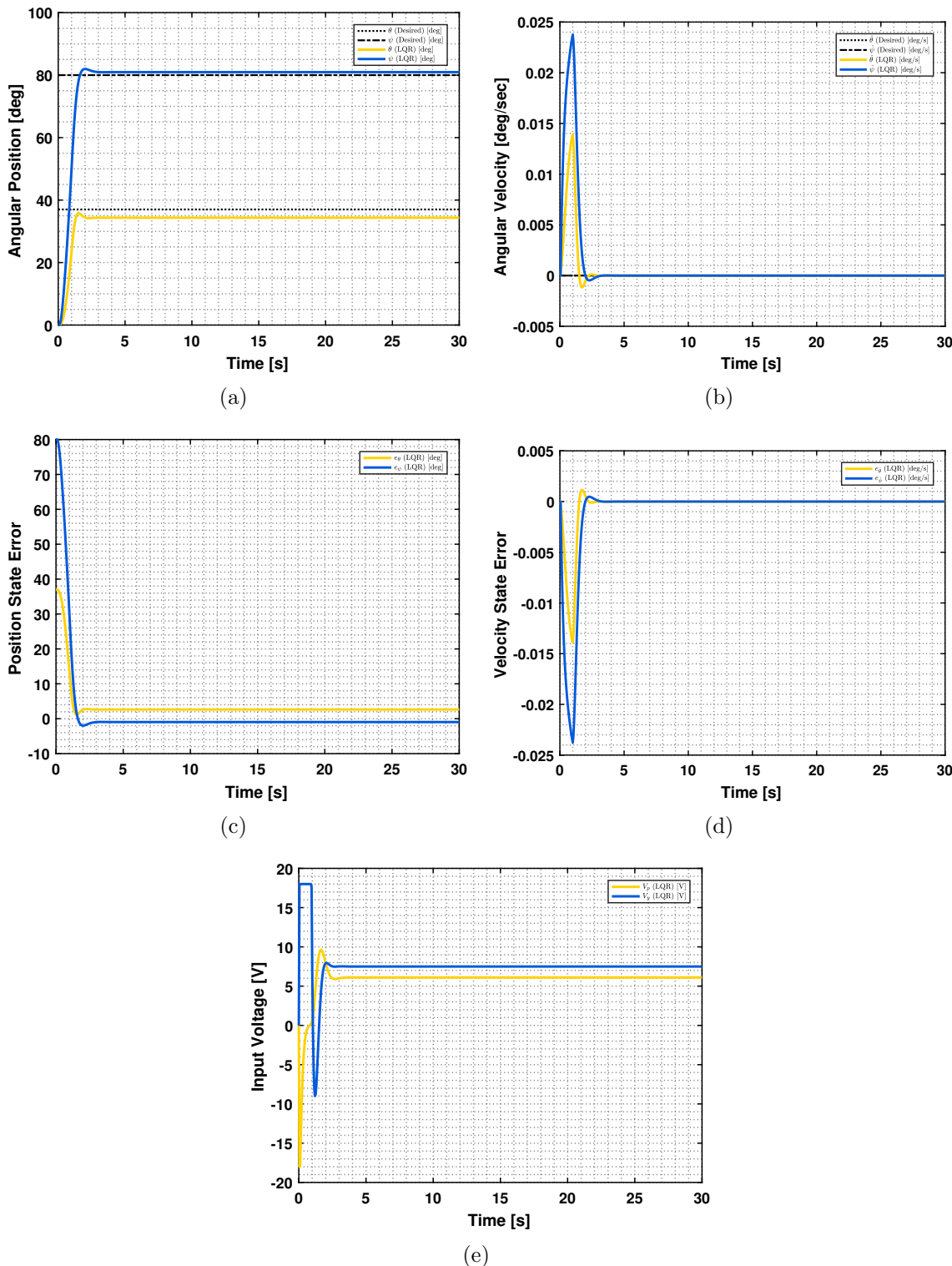


Figure 5.8: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

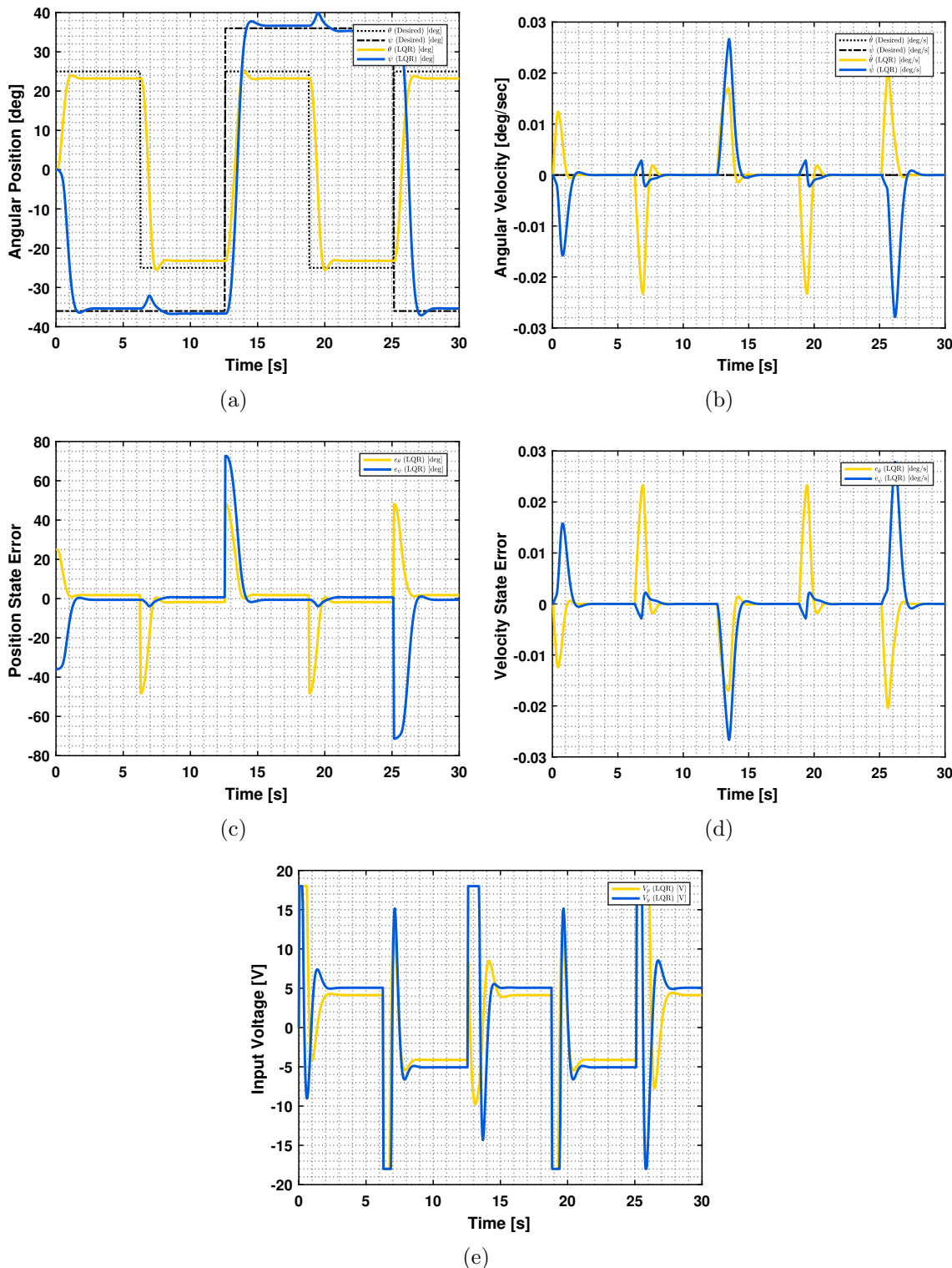


Figure 5.9: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

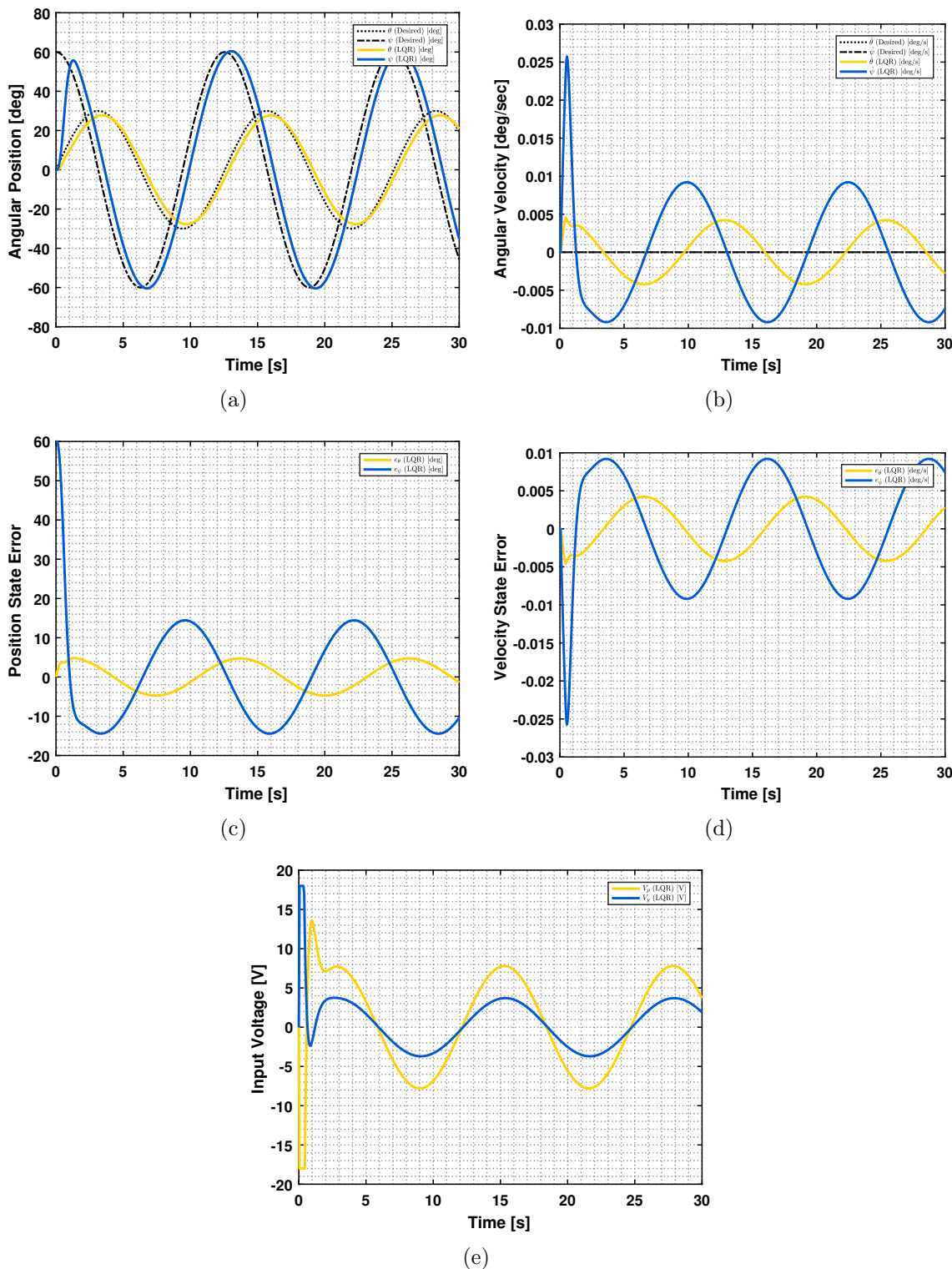


Figure 5.10: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.



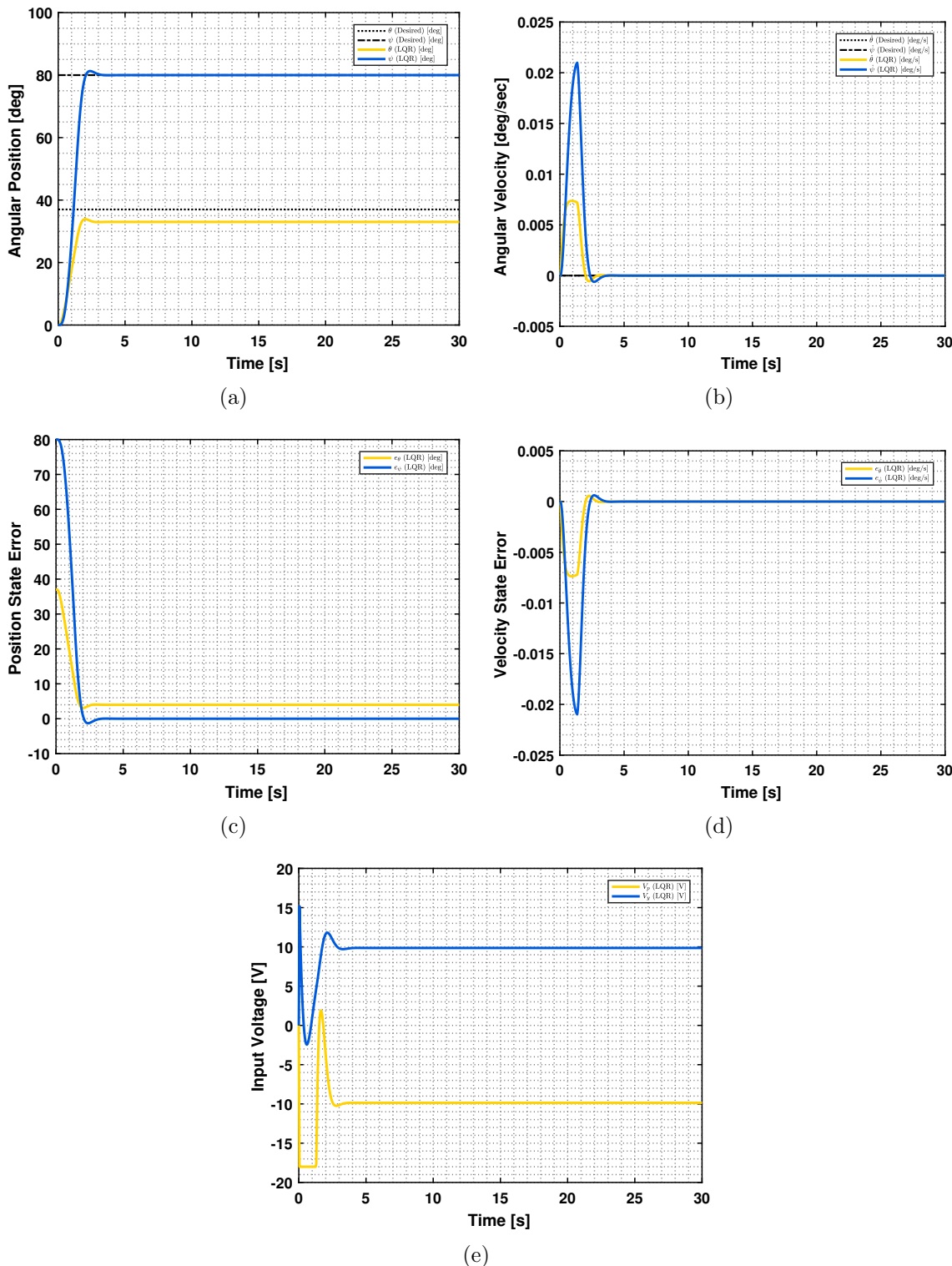


Figure 5.11: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.



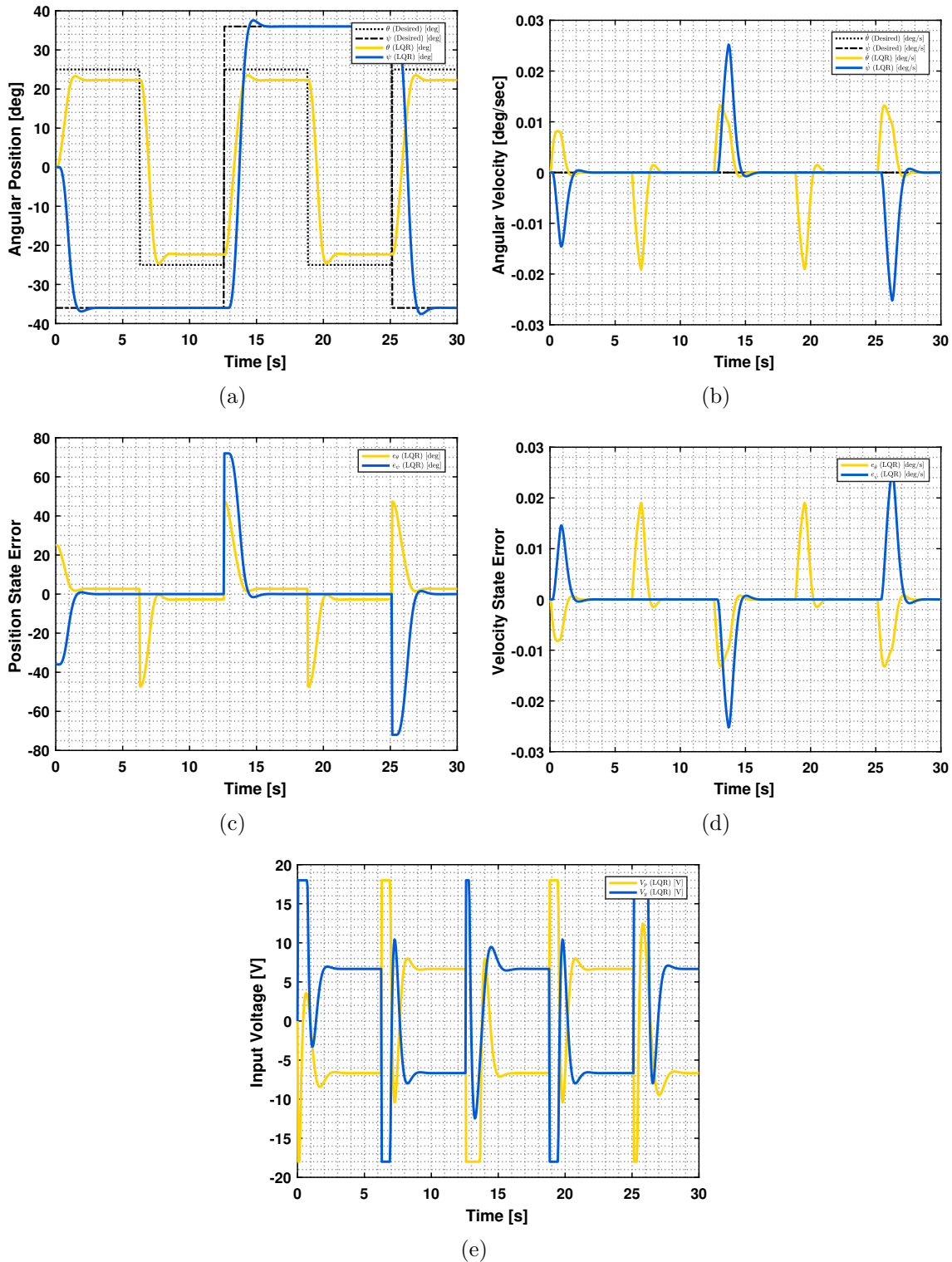


Figure 5.12: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

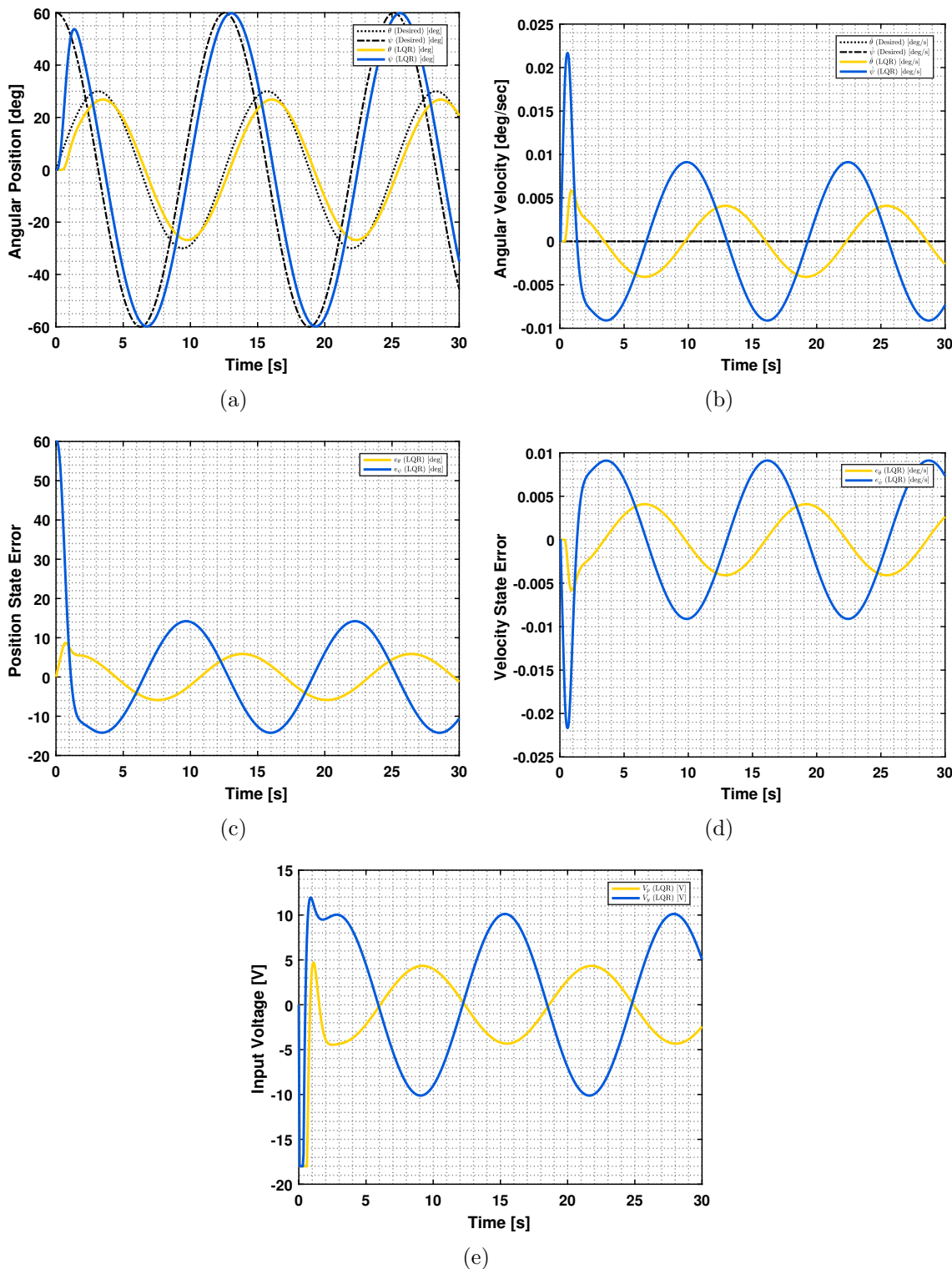


Figure 5.13: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

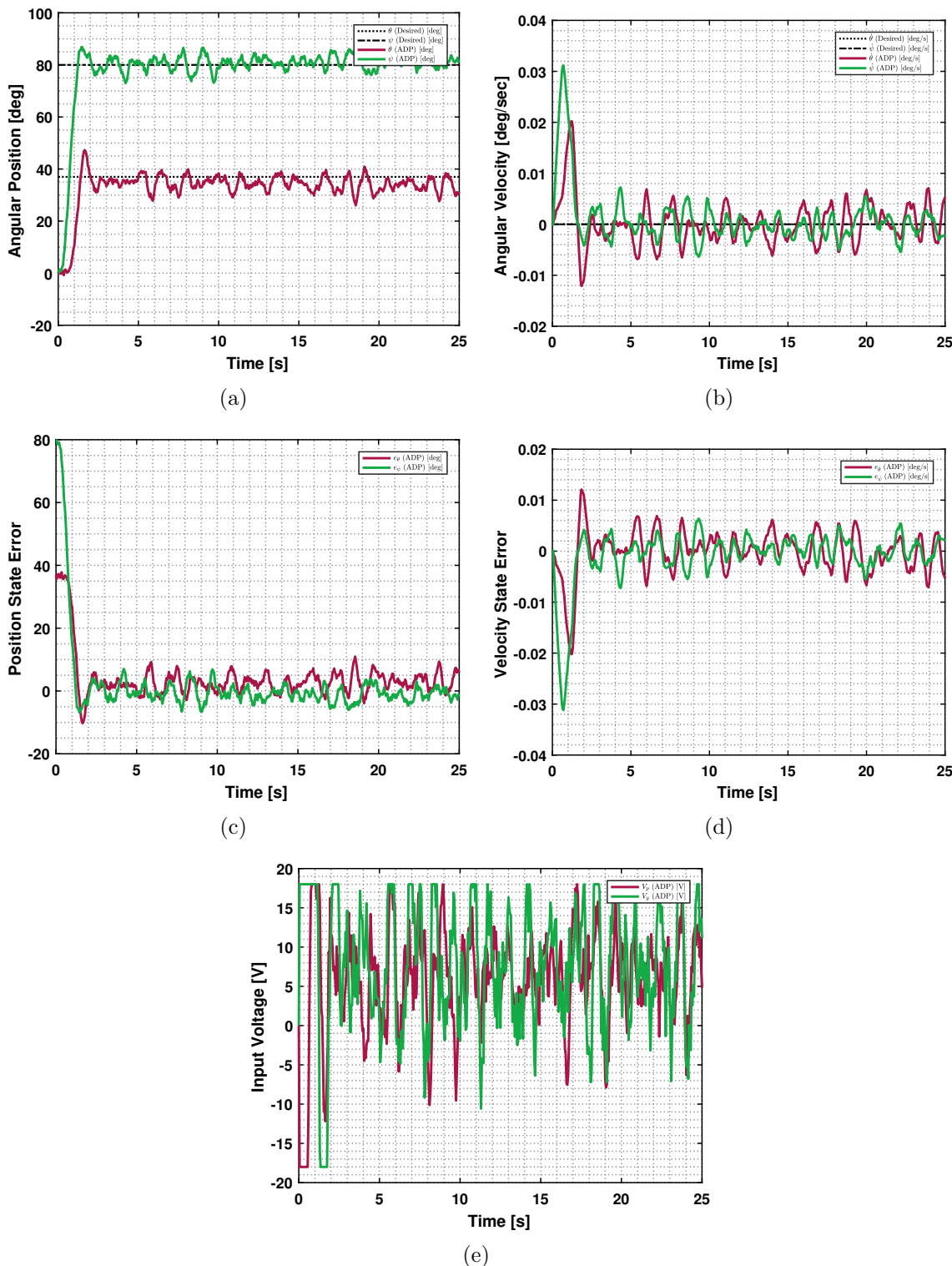


Figure 5.14: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

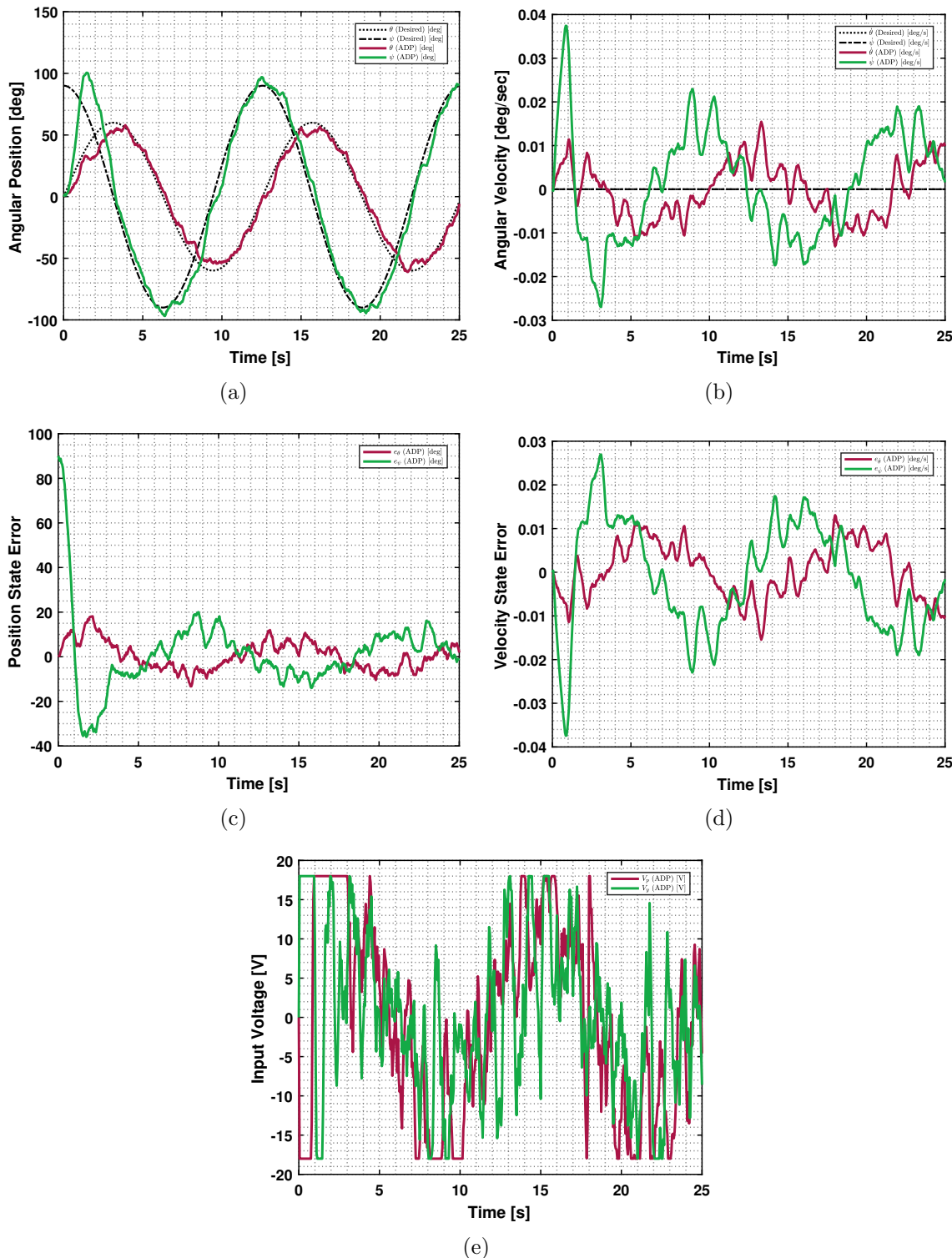


Figure 5.15: Time-varying constant desired pitch and yaw for 2-DOF helicopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

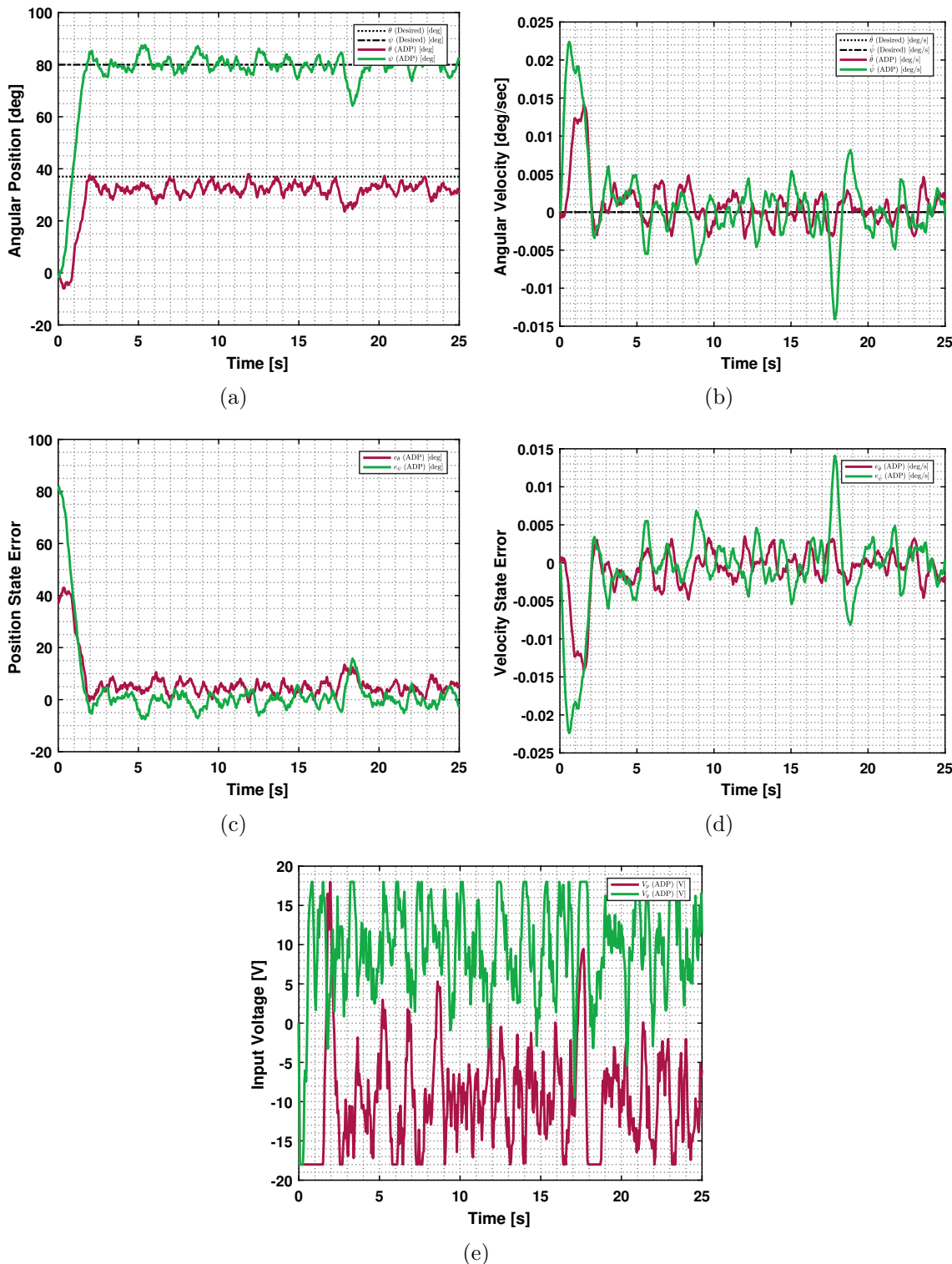


Figure 5.16: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.



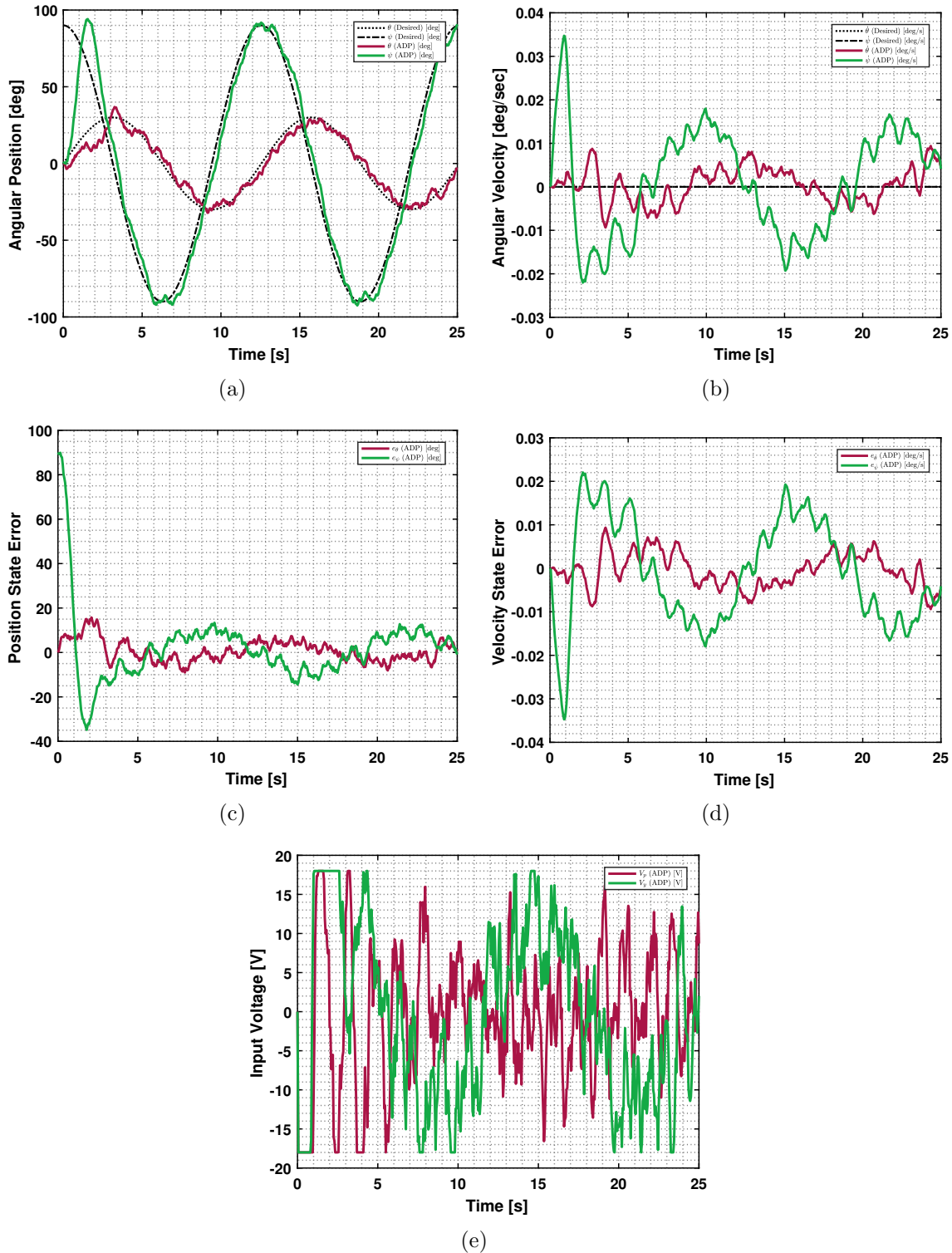


Figure 5.17: Time-varying constant desired pitch and yaw for half-quadcopter. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

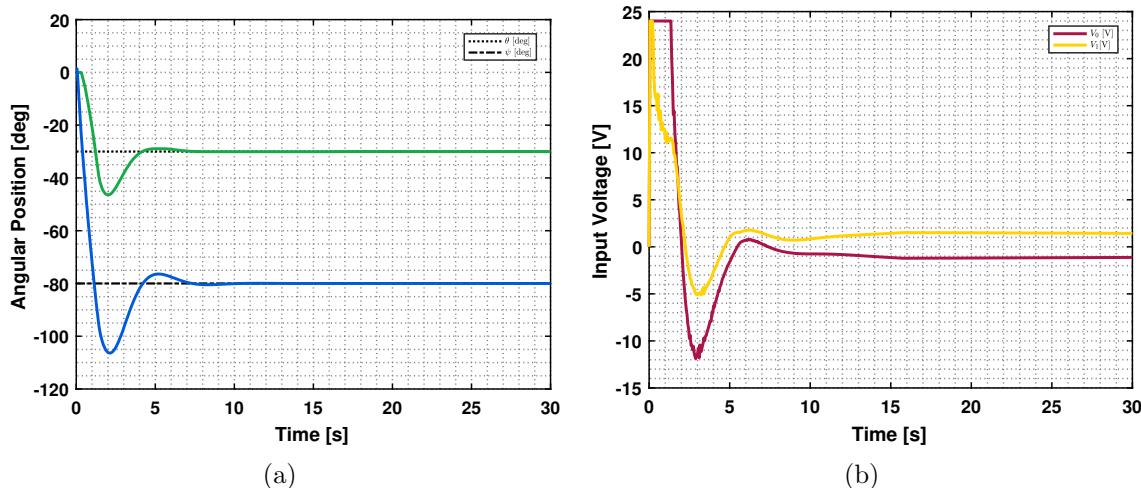


Figure 5.18: Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs.

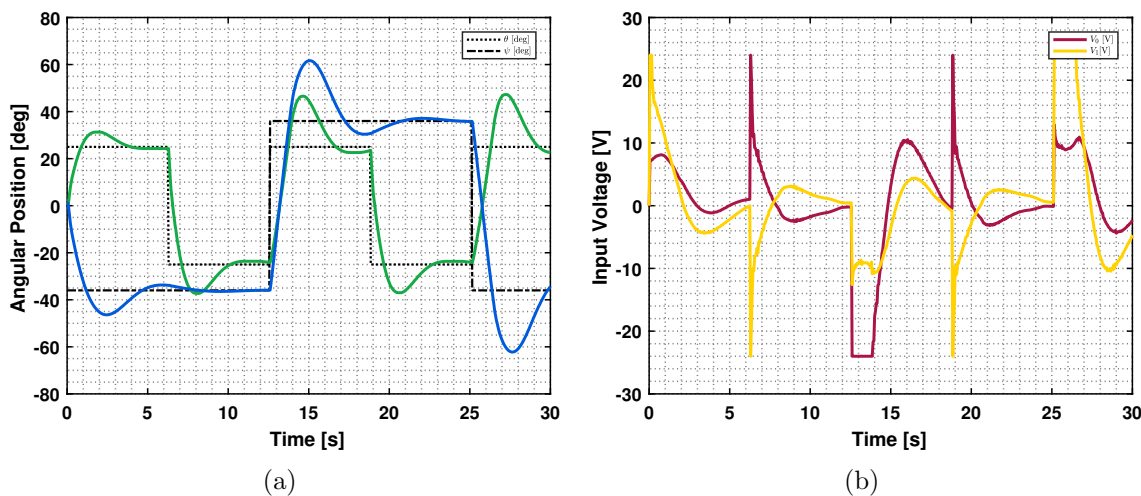


Figure 5.19: Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs.

complexity of ADP, latency issues arise between MATLAB and V-REP. MATLAB is not able to perform fast enough for ADP to simulate in real-time in V-REP. Options to fix this could either be optimizing ADP or using a faster language besides MATLAB. For now, it suffices to run V-REP not in real-time mode, made possible by the MATLAB calling function. The results predict that ADP will work in the V-REP simulations, but that optimization could greatly be advanced. Further research needs to be conducted to address this issue.

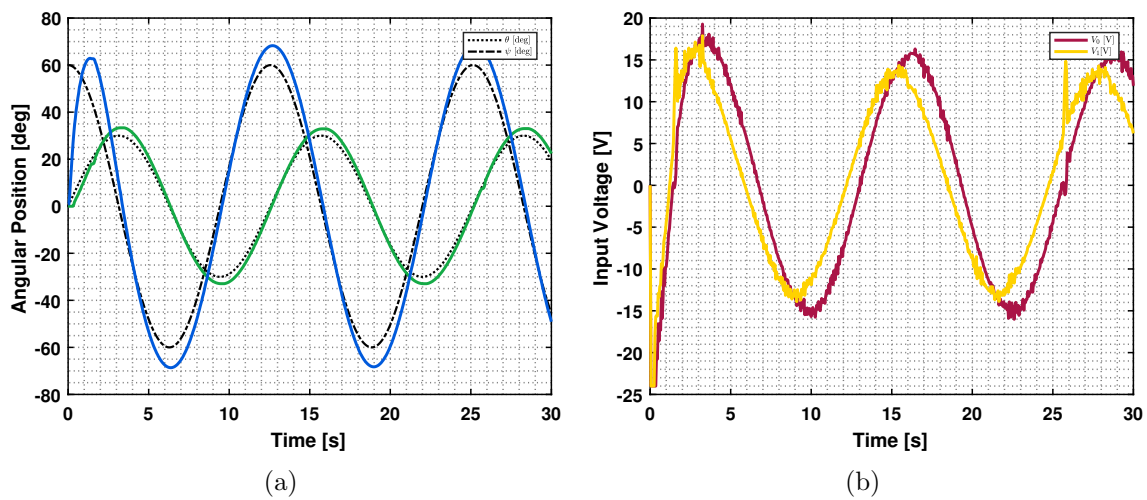


Figure 5.20: Time-varying constant desired pitch and yaw for half-quadcopter with PID controller. (a) Angular position. (b) Voltage inputs.



# Chapter 6

## Hardware Implementation

Once MATLAB simulations showed that ADP can be used as a controller for the Quanser AERO, ADP could be implemented to control the physical Quanser AERO. To accomplish this hardware implementation, we used the graphical programming language Simulink which is recommended by Quanser. Simulink is an industry-standard graphical programming language used for the purpose of C-code generation from the graphical interface. This is exactly why it has been implemented by Quanser. Our initial task was to implement ADP into a Simulink model. This is actually more difficult than it sounds because of the complexity of ADP.

The Quanser AERO can be configured with two different interface panels, the QFLEX 2 USB and the QFLEX 2 Embedded. The QFLEX 2 USB is the panel that allows the Quanser AERO to be controlled via USB from a personal computer. This panel requires specialized Quanser software which will be discussed. Quanser also provides plenty of documentation and Simulink examples for the QFLEX 2 USB panel.

The other panel provided by Quanser is the QFLEX 2 Embedded. This panel allows communication to the Quanser AERO via SPI communication. The QFLEX 2 Embedded is designed to be used with embedded devices such as a Raspberry Pi. We implemented this panel using a Raspberry Pi 3 (<https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>) which can be seen in Figure 6.1. The problem with the QFLEX 2 Embedded panel is that Quanser does not have much documentation or any examples for this panel. All of the results that will be discussed about the Raspberry Pi 3 were made from our own research and technical experiences.

One advantage of starting from the beginning with the Raspberry Pi 3 is that we had the freedom to expand its applications once we were able to get the basics under control. This can be seen with the application of the Android smart phone application which will be discussed as well.

Our initial goal for the project was to be able to implement ADP on the Quanser AERO through various media. The options can be seen in Figure 6.2. Not all media have been achieved, but the ones completed will be described in this chapter.

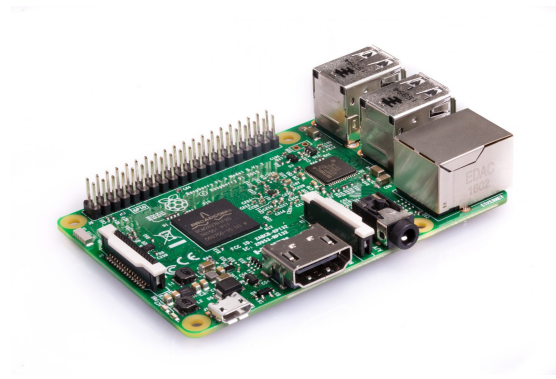


Figure 6.1: Raspberry Pi 3.

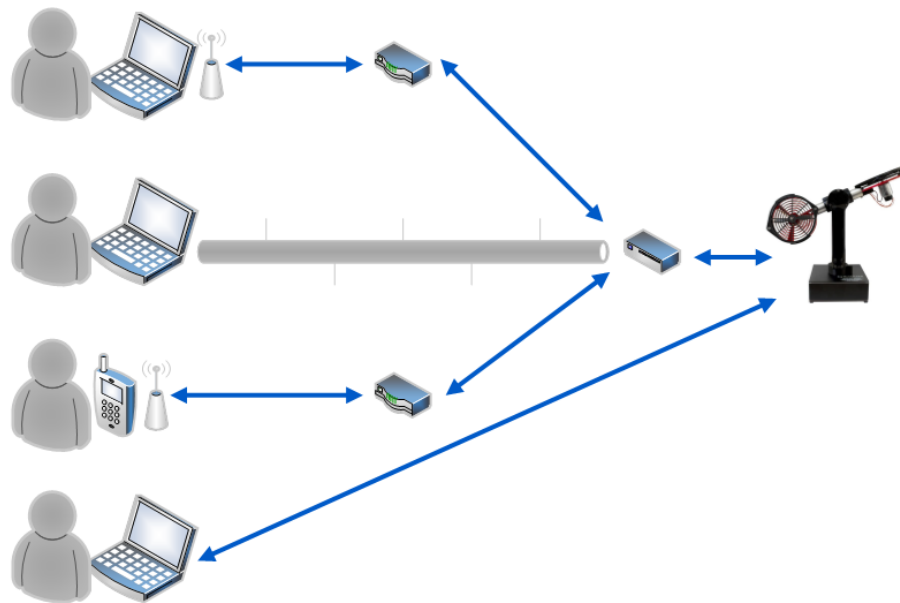


Figure 6.2: From top to bottom. 1) User controls the Quanser AERO from their laptop via Wi-Fi through the Raspberry Pi 3. 2) User controls the Quanser AERO from their laptop via Ethernet through the Raspberry Pi 3. 3) User controls the Quanser AERO from their cell phone via Wi-Fi through the Raspberry Pi 3. 4) User controls the Quanser AERO via a USB connection using the QFLEX 2 USB panel.

This chapter is organized as follows. Section 6.1 discusses the Simulink model of ADP. Section 6.2 discusses the QFLEX 2 USB panel and its use in running a Simulink model on the Quanser AERO. The use of the QFLEX 2 Embedded panel is discussed in Section 6.3 along with the use of the Raspberry Pi 3 and its complexities. Section 6.4 showcases the development of a Android smart phone application created by Simulink to control the Quanser AERO via the Raspberry Pi 3.

**NOTE:** MATLAB 2016b and its associated Simulink version were used for the entirety of this project. Different versions may affect some of the Simulink models.

## 6.1 ADP Algorithm in Simulink

In order to utilize ADP on the Quanser AERO, we have to convert the ADP MATLAB code to a Simulink model. The Simulink model is essentially a block diagram of ADP. We need to use Simulink to model ADP because the QUARC software, which will be discussed later, generates C-code from the Simulink model. This C-code is what drives the QFLEX 2 USB panel.

The Simulink model of ADP can be seen in Figure 6.3. The Simulink model seen in

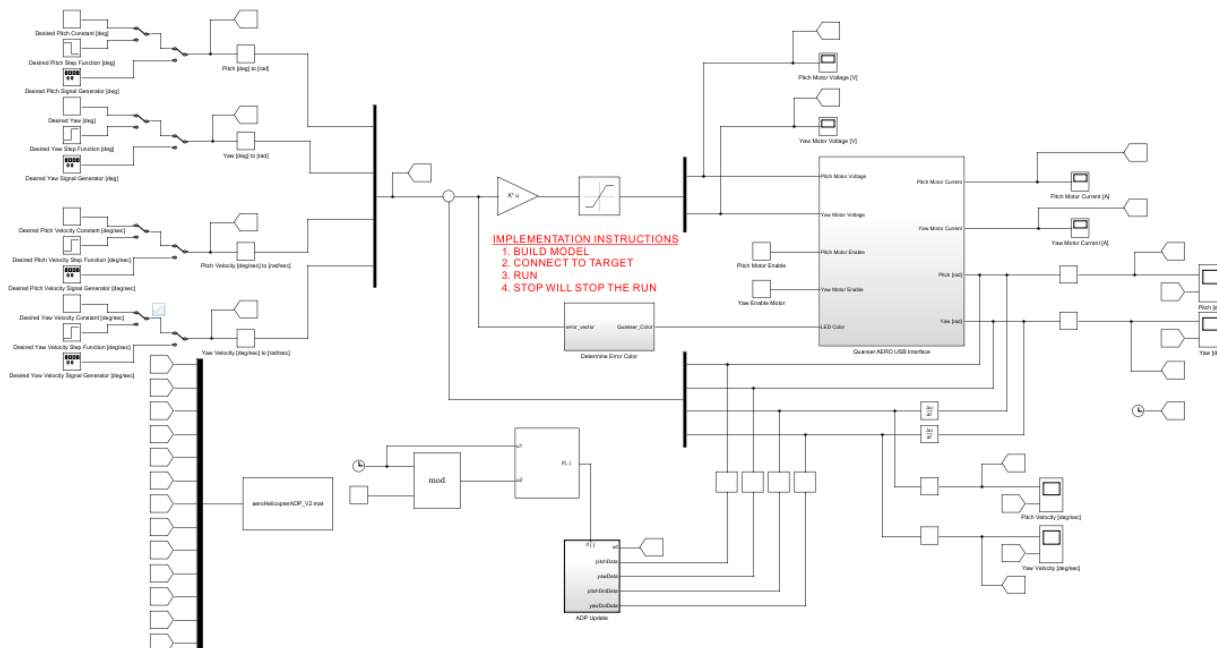


Figure 6.3: ADP algorithm in the form of a Simulink model.

Figure 6.3 is actually combined with the communication necessary for the QFLEX 2 USB, but we will only focus on the ADP aspects in this section.

To analyze the Simulink model shown in Figure 6.3, we can briefly describe the model clockwise starting in the top-left. Before we describe the model, it should be noted that the state-feedback gain needs to be initialized before ADP can be ran. This is accomplished in Simulink by calling an "init" function which is declared in the Model Properties. This initialization function essentially calculates the state-feedback gain using random error data. It then saves the state-feedback gain and other important values such as sampling times in the workspace. The specific initialization function can be seen in Appendix C.2. The MATLAB code is very similar to that of the MATLAB simulations because MATLAB executes this script before executing the Simulink model.

Now, observing the Simulink model in Figure 6.3, we can begin analyzing it starting with the inputs. The inputs to the Simulink model are the desired pitch and yaw and their

corresponding desired angular velocities. The inputs are setup such that they can be easily switched. Moving to right, the desired configurations are combined into a vector with the mux block. The vector then enters the state-feedback system. We can see the state-feedback gain and the system which is the the largest block in the Simulink model. The inputs are the two motor voltages which are saturated between -24 and 24 Volts by the saturation block. We saturated the voltage inputs to 18 Volts in the MATLAB simulations, but Quanser Simulink examples use a saturation level of 24 Volts, so we will implement that method.

Once the system uses the input voltages to update the position, we use feedback to find the pitch, yaw, and angular velocity error used to find new inputs. Because we are not measuring the angular velocity, we use the derivative block to find the angular velocity. The model thus far essentially represents a state-feedback system, but we have not discussed ADP yet.

ADP is the subsystem in the lower center of the Simulink model. This subsystem is only triggered when the sampling is a multiple of  $T$ . The subsystem contains the user-defined function in Figure 6.4. This user-defined function essentially contains another version of

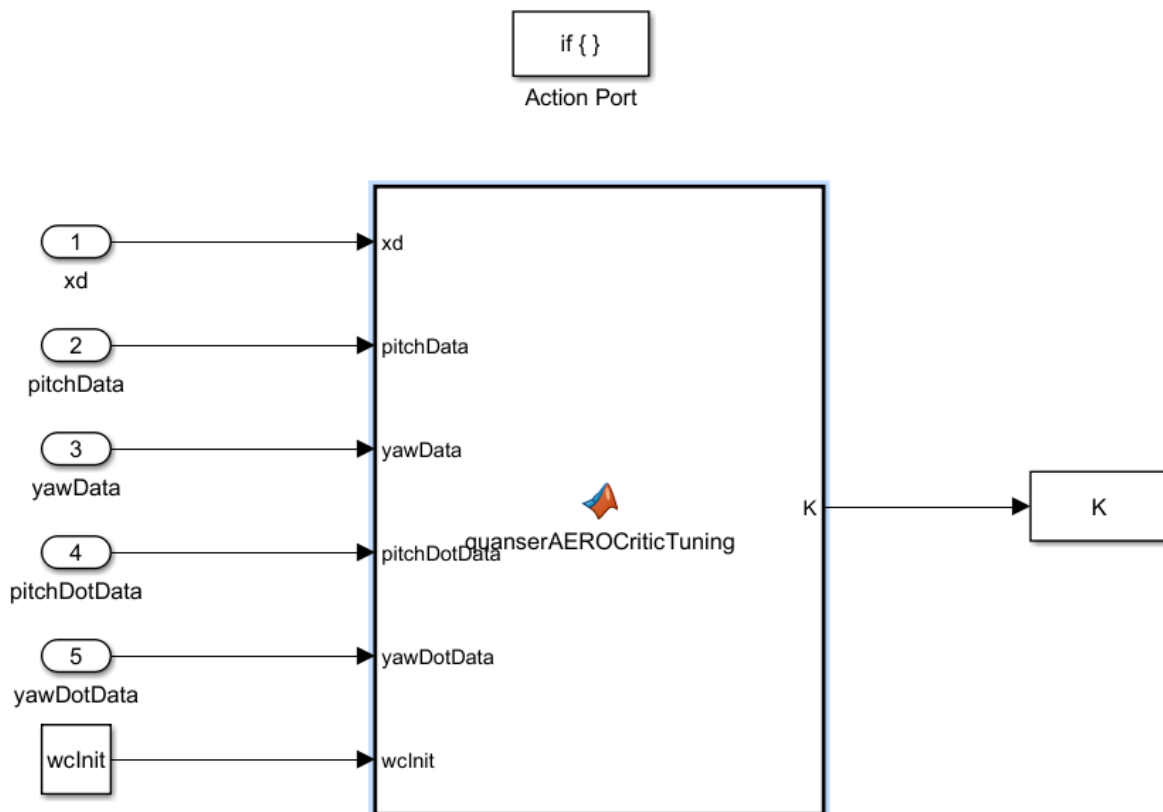


Figure 6.4: User-defined function to update the state-feedback gain in the ADP Simulink model.

ADP in MATLAB code. When designing the ADP Simulink model, it was determined that

it would be easier to utilize ADP MATLAB code rather than converting it to a model itself. The disadvantage is the MATLAB code must be compatible for C-code generation. The reason for this will be discussed in later sections, but for now, the ADP update MATLAB code can be seen in Appendix C.3. One can notice that certain functions used in the original MATLAB simulations had to be changed to overcome this C-code generation challenge. One example is the use of anonymous functions. It should also be noted that the error data needed to be saved for use in the ADP subsystem. To save the error data, a tapped delay block was used for each of the four states. The tapped delay allows you to specify the number of delays you would like stored in the block saved as an array.

The last subsystem yet to be discussed in the Simulink model in Figure 6.3 is the subsystem in the center. This subsystem will be used later for the QFLEX 2 USB, but it can be discussed now. The subsystem changes the color of the base LED. In the subsystem is another user-defined function based on the magnitude of the error seen in Figure 6.5. The

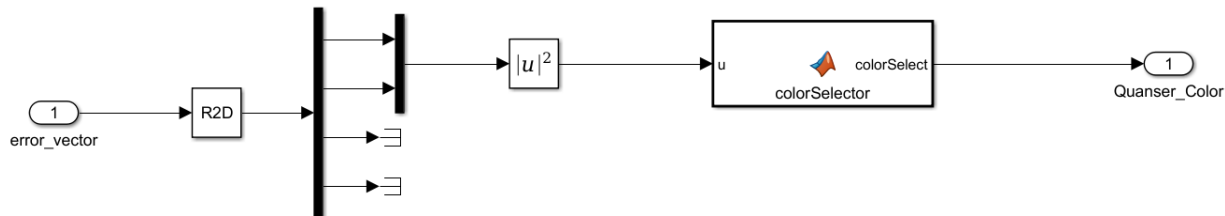


Figure 6.5: User-defined function to change the base LED color.

user-defined MATLAB function code can be seen in Appendix C.4. The output of the MATLAB code is the intensities of red, blue, and green values to be sent to the Quanser AERO.

The items mentioned here were the main components of ADP in the form of a Simulink model. It is actually a rather simple model since we already had the MATLAB code ready from the simulations. The model could have been made a lot more difficult if the user-defined MATLAB functions were not used. As for the other components of the model, there are multiple scopes and gotos. The gotos make signal routing much cleaner. They are also used to save data to a .mat file. This data can be used for plotting and comparing experiments.

## 6.2 QFLEX 2 USB

As mentioned before, the QFLEX 2 USB panel provides an interface between the Quanser AERO and a computer running QUARC which will be discussed next. The QUARC software is the software that uses the C-code generated by Simulink and makes it compatible to run on the QFLEX 2 USB panel. We have a control method set up, so the only item left is to send the actuator commands to the Quanser AERO with the use of the QFLEX 2

USB panel. Quanser provides the necessary Simulink blocks for this communication which will be discussed in the following sections.

**NOTE:** A tutorial is provided in Appendix C.1 on how to run a complete Simulink model for the Quanser AERO. The Simulink model must contain a control and communication method to send the actuator commands to the Quanser AERO.

### 6.2.1 Required Software

In order to utilize the QFLEX 2 USB panel, some extra software is needed. To obtain and use the Simulink blocks provided by Quanser, a licensed version of QUARC must be obtained. QUARC is the proprietary software of Quanser that installs the Simulink QUARC package that provides communication blocks for the QFLEX 2 USB panel. QUARC also provides additional Simulink blocks such as filters and interfaces in Simulink.

Additional MATLAB and Simulink packages must also be obtained in order for QUARC to operate correctly. The QFLEX 2 USB panel cannot understand a Simulink model; some form of C-code is needed for the Quanser AERO to operate correctly. Because we are using Simulink, we need to convert the graphical model to C-code. To do this, we must have the MATLAB Coder (<https://www.mathworks.com/products/matlab-coder.html>) and Simulink Coder (<https://www.mathworks.com/products/simulink-coder.html>) packages installed. These packages can be obtained by going to one's MATLAB. In the Home tab, click on Add-Ons. You can either use Manage Add-Ons or Get Add-Ons to find the MATLAB Coder and Simulink Coder. It should be noted that these add-ons are not free, so a MathWorks account is needed.

### 6.2.2 Quanser AERO Simulink Model

With the QUARC software, communication between Simulink and the Quanser AERO is made much easier. As discussed earlier, the QUARC software installs Simulink libraries and blocks supported by the Quanser AERO. For our case, only three additional blocks were needed to communicate ADP to the Quanser AERO. The three blocks can be seen in Figure 6.6. Figure 6.6 is the Simulink subsystem that depicts the physical system in Figure 6.3.

All three new blocks correlate to each other. The HIL Initialize block defines the platform we are implementing once the proper settings are entered. With the proper settings entered in the HIL Initialize block, the QUARC software knows what platform to send the Simulink signals to and from.

The HIL Write block does exactly what its name implies. The HIL Write block writes our provided signals to the Quanser AERO. For our purposes, we are only concerned with motor voltages, enabling motors, and the base color. These inputs to the block can be

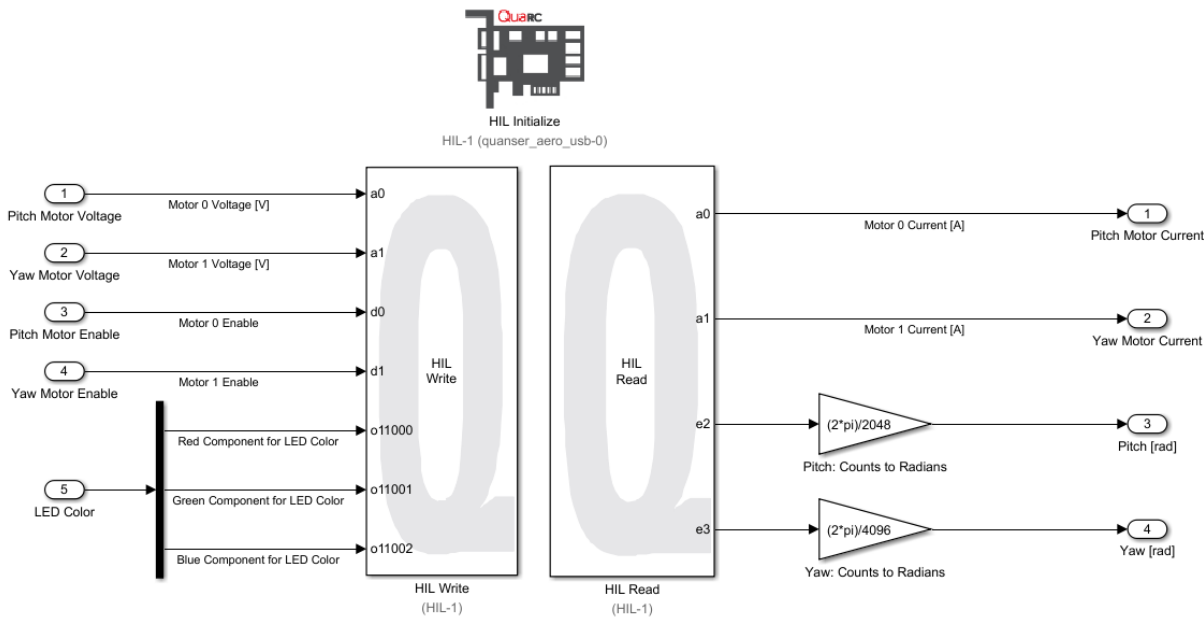


Figure 6.6: Simulink subsystem used to communicate with the QFLEX 2 USB panel.

changed by updating the settings inside the block. It is recommended to also refer to the help documentation associated with this block when selecting inputs as the names are abbreviated.

The HIL Read block also does exactly what its name implies. The HIL Read block reads our specified signals from the Quanser AERO. For our purposes, we are only concerned with the pitch, yaw, and motor currents. These outputs to the block can be changed by updating the settings inside the block. It is again recommended to also refer to the help documentation associated with this block when selecting inputs as the names are abbreviated.

With these three Simulink blocks implemented in your model and with the QUARC software installed, communication between Simulink and the Quanser AERO is much simpler. A tutorial on how to run your Simulink model on the Quanser AERO is provided in Appendix C.1.

### 6.2.3 Results

In order to see the effectiveness of ADP, we needed to see physical measurements from the implementation results. Because we ran ADP through Simulink, we could record various measurements from the Quanser AERO in a .mat file. in Section 4. The plots we generated can be seen in Figures 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17, 6.18, 6.19, and 6.20. For each of the experiments, we ran the experiment with a saturation level of either 18 or 24 Volts. The motors are rated at 18 Volts, but Quanser Simulink models run



them at 24 Volts, so we tried both voltages. In order to have a baseline comparison, we compared ADP to that of LQR. Quanser provides a Simulink model for LQR which we modified slightly to save the measurements. We then ran each test using both ADP and LQR.

It should be noted that ADP needs the  $\mathbf{B}$  matrix of the state-space model to update the state-feedback gains accurately. To make sure we had an accurate  $\mathbf{B}$  matrix, we measured the thrust gains. We measured the thrust gains by running the experiments provided in Quanser documentation. The documentation went through measuring and calculating the thrust gains. We then used the constants provided by Quanser to find the  $\mathbf{B}$  matrix. We did not use any experiments to find the  $\mathbf{A}$  matrix for our system; we used the constants provided. ADP should be able to correct if the  $\mathbf{A}$  matrix is slightly off, but we could see drastic changes in the performance of ADP using the calculated  $\mathbf{B}$  matrix and the one provided in Quanser documentation.

From Figures 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17, 6.18, 6.19, and 6.20 we can see that both ADP and LQR perform well for this particular system. The results are similar to the MATLAB simulations, but that is not enough evidence. To see if there are any bigger differences, we calculated the root-mean-squared-error of the measurements and also the root-mean-squared values of the power of the motors. These values can be seen in Tables 6.1, 6.2, and 6.3. It should also be noted that these experiments were conducted using the inefficient propellers provided with the Quanser AERO.

Table 6.1: Experimental Simulink angular RMSE measurements.

Figure	$\theta$ ADP [deg]	$\theta$ LQR [deg]	$\psi$ ADP [deg]	$\psi$ LQR [deg]
6.7	8.794	23.135	8.565	23.308
6.8	8.463	21.205	7.777	21.219
6.9	23.088	21.612	21.144	21.289
6.10	21.478	19.727	18.944	17.942
6.11	9.414	14.833	9.595	19.207
6.12	4.661	8.424	5.925	13.371
6.13	3.787	9.728	3.375	14.155
6.14	3.143	8.885	2.964	13.622
6.15	6.162	14.185	7.116	14.171
6.16	5.973	12.910	6.533	12.812
6.17	14.496	14.939	14.220	15.049
6.18	13.816	14.074	13.368	13.594
6.19	32.126	72.588	30.188	73.744
6.20	30.408	65.609	27.501	65.348

From Table 6.1, we can see that ADP performs better at minimizing the error with respect to the desired pitch and yaw compared to LQR. From Table 6.2, we can see that ADP



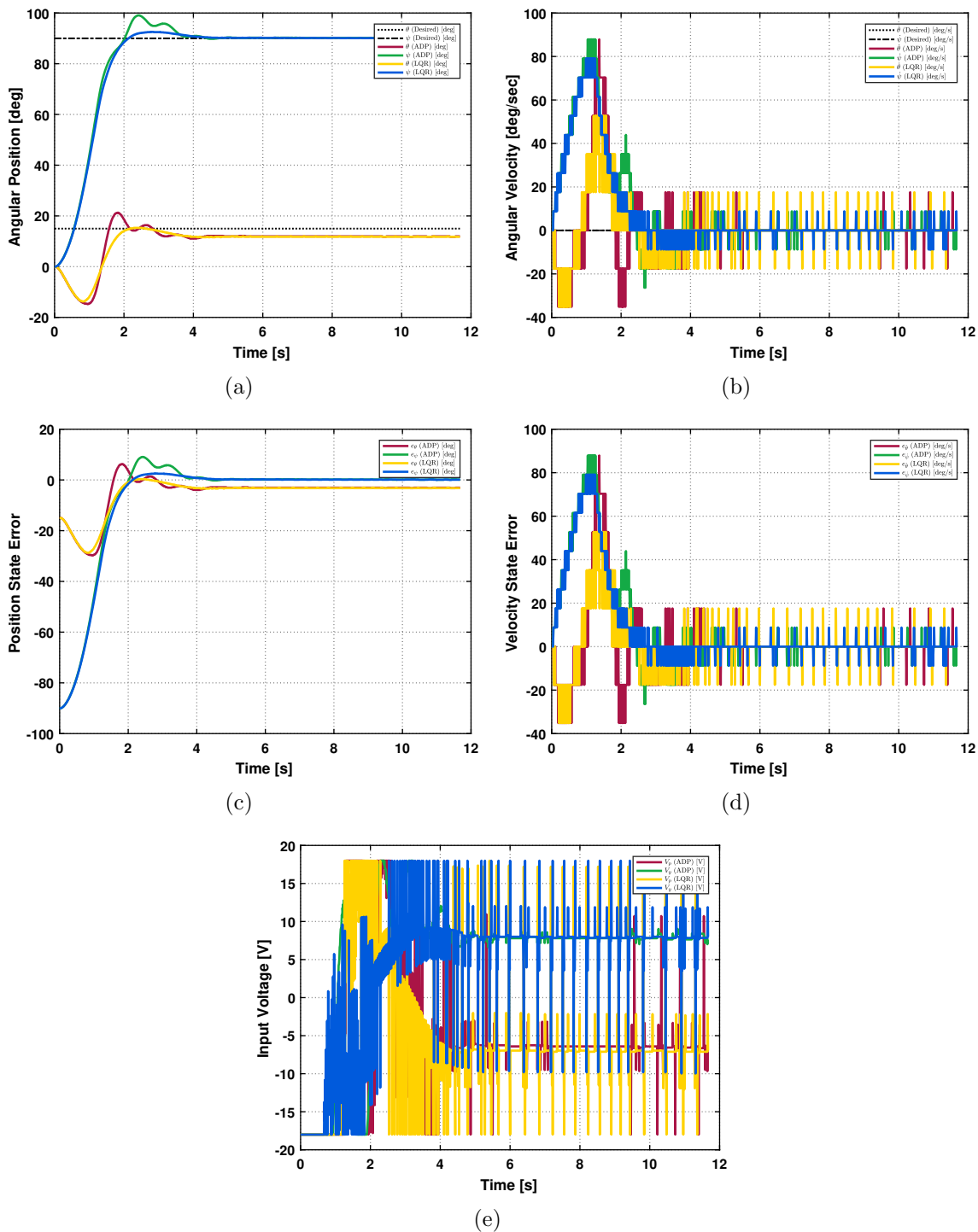


Figure 6.7: Constant desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

does a somewhat decent job in minimizing the error with respect to the desired angular velocities. ADP only performs slightly better, but this measurement can be difficult to

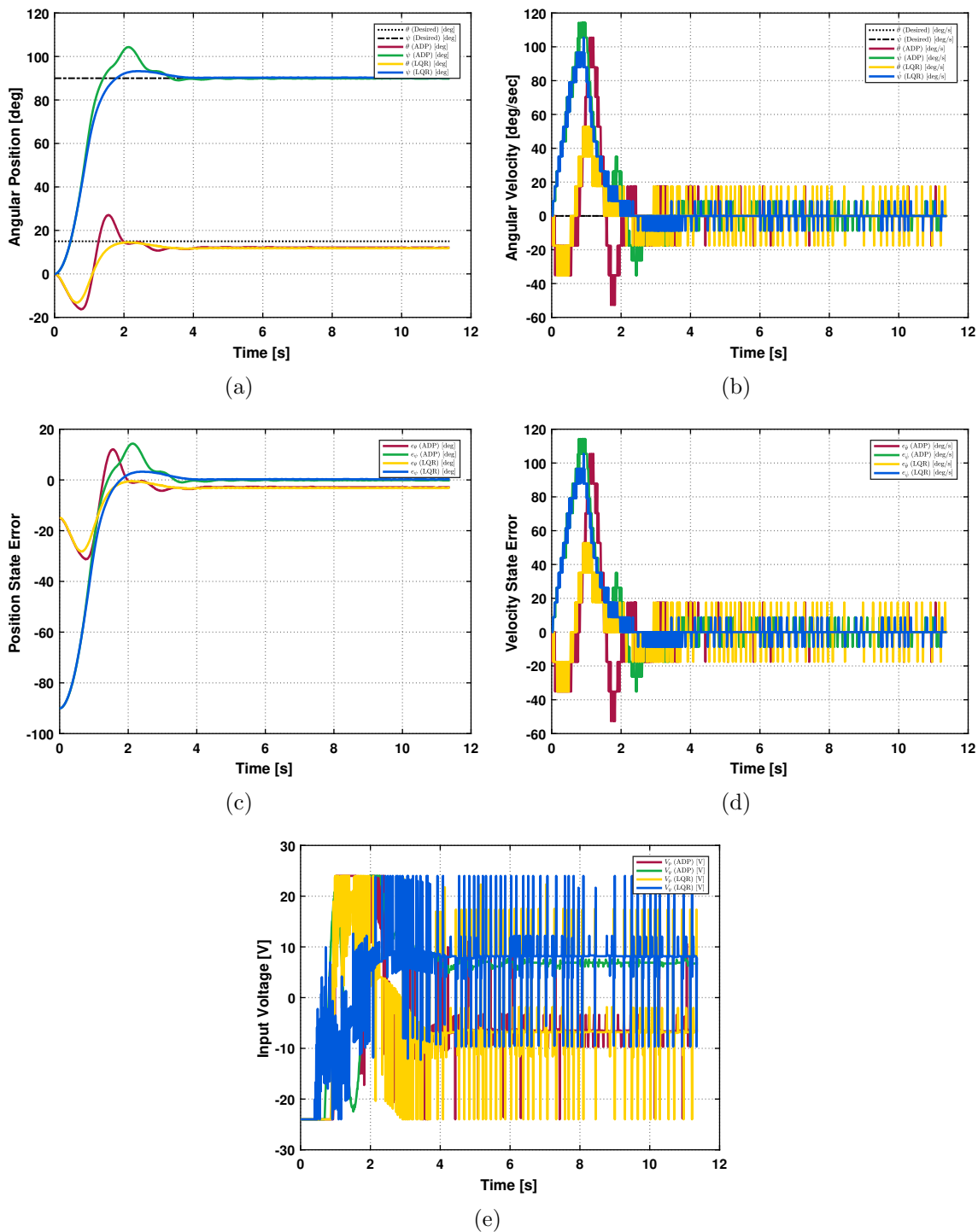


Figure 6.8: Constant desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

interpret. We are changing the desired trajectory, so the desired velocities cannot always be zero as we intended. As for the power of the motors, Table 6.3 shows that there is no

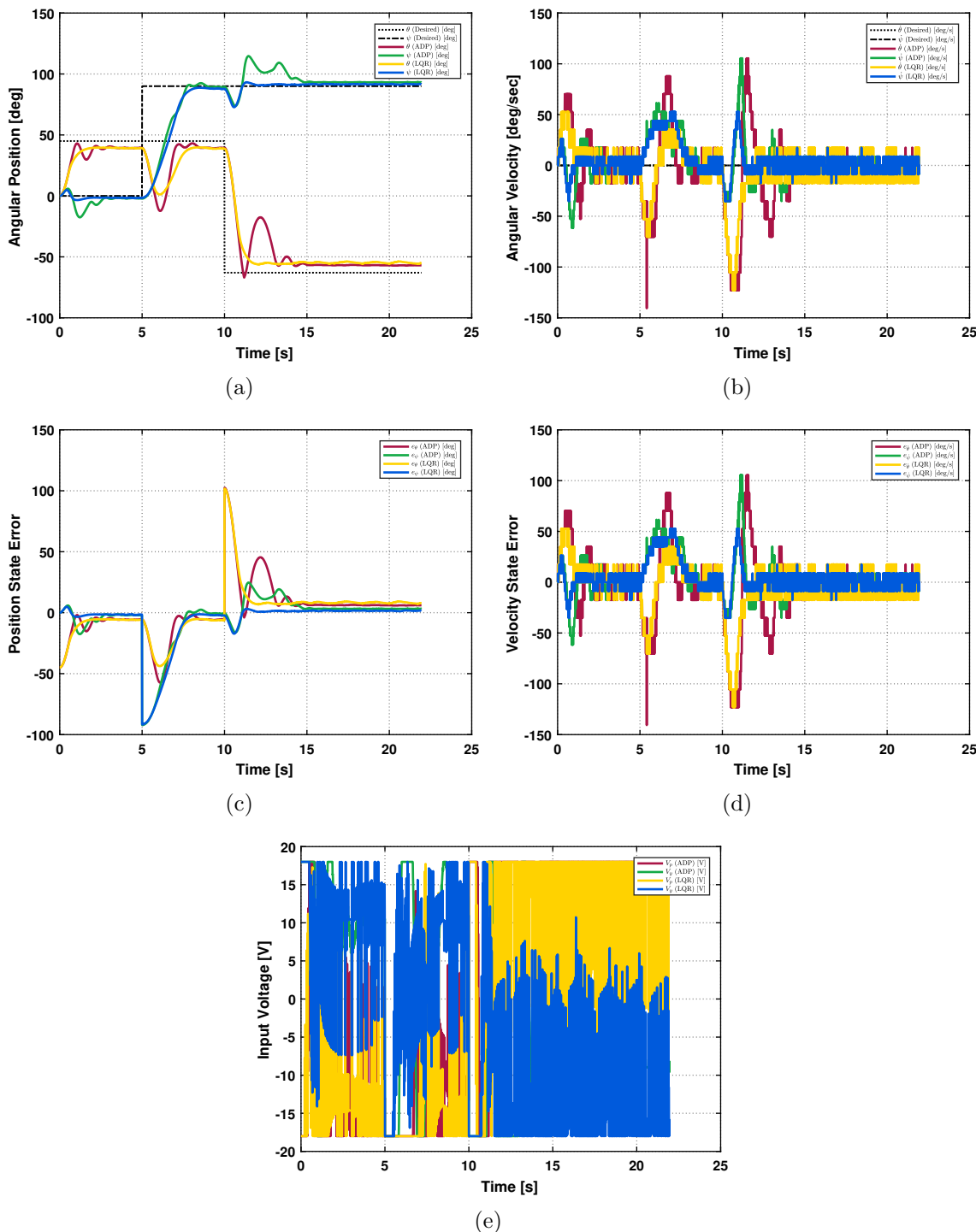


Figure 6.9: Step desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

clear advantage to either ADP or LQR. The results seem to be evenly split in favor of either. This is not great news because we were hoping that ADP would yield control that

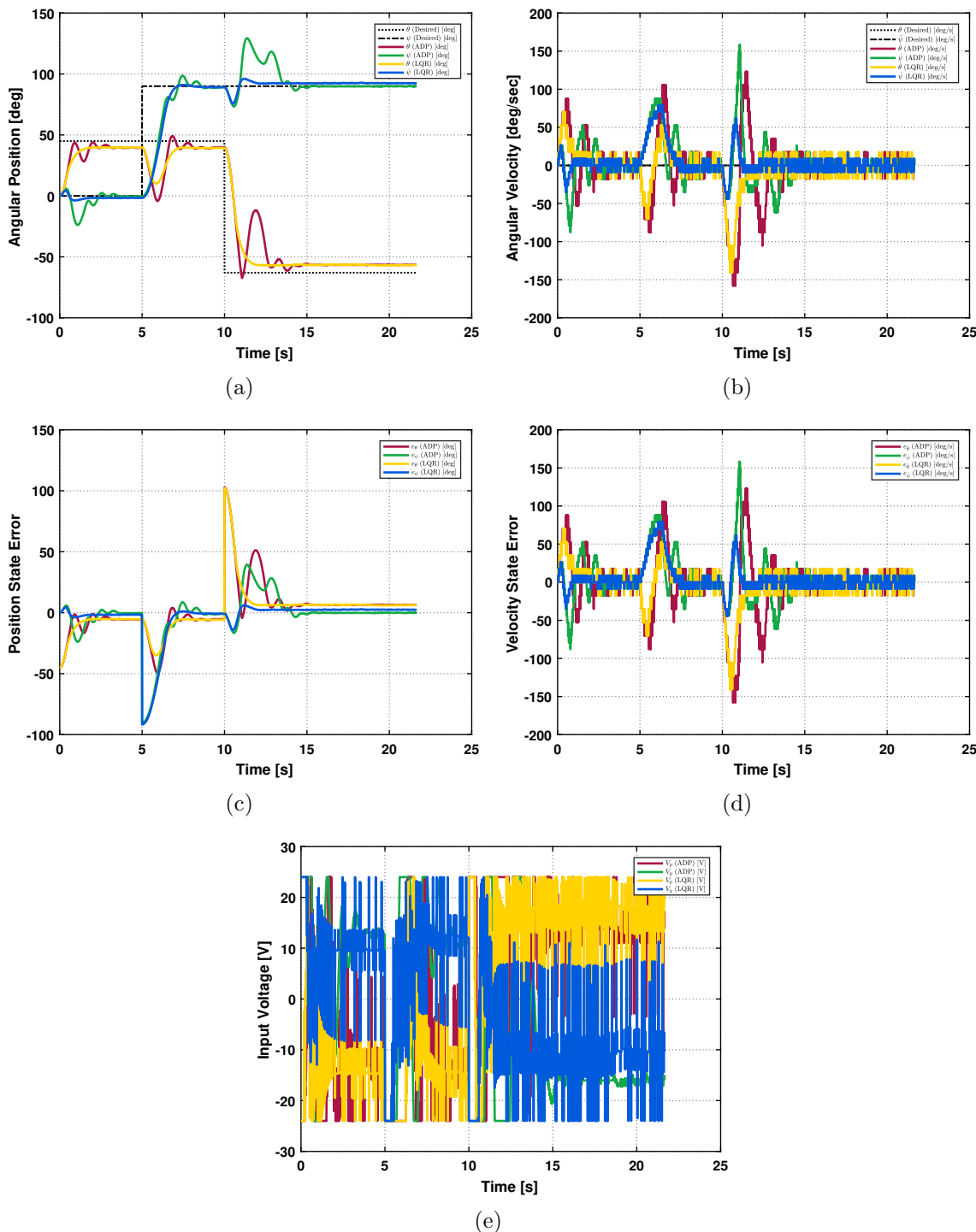


Figure 6.10: Step desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

would minimize the energy consumed. We just cannot say for sure that this occurs with the experiments we have ran here.

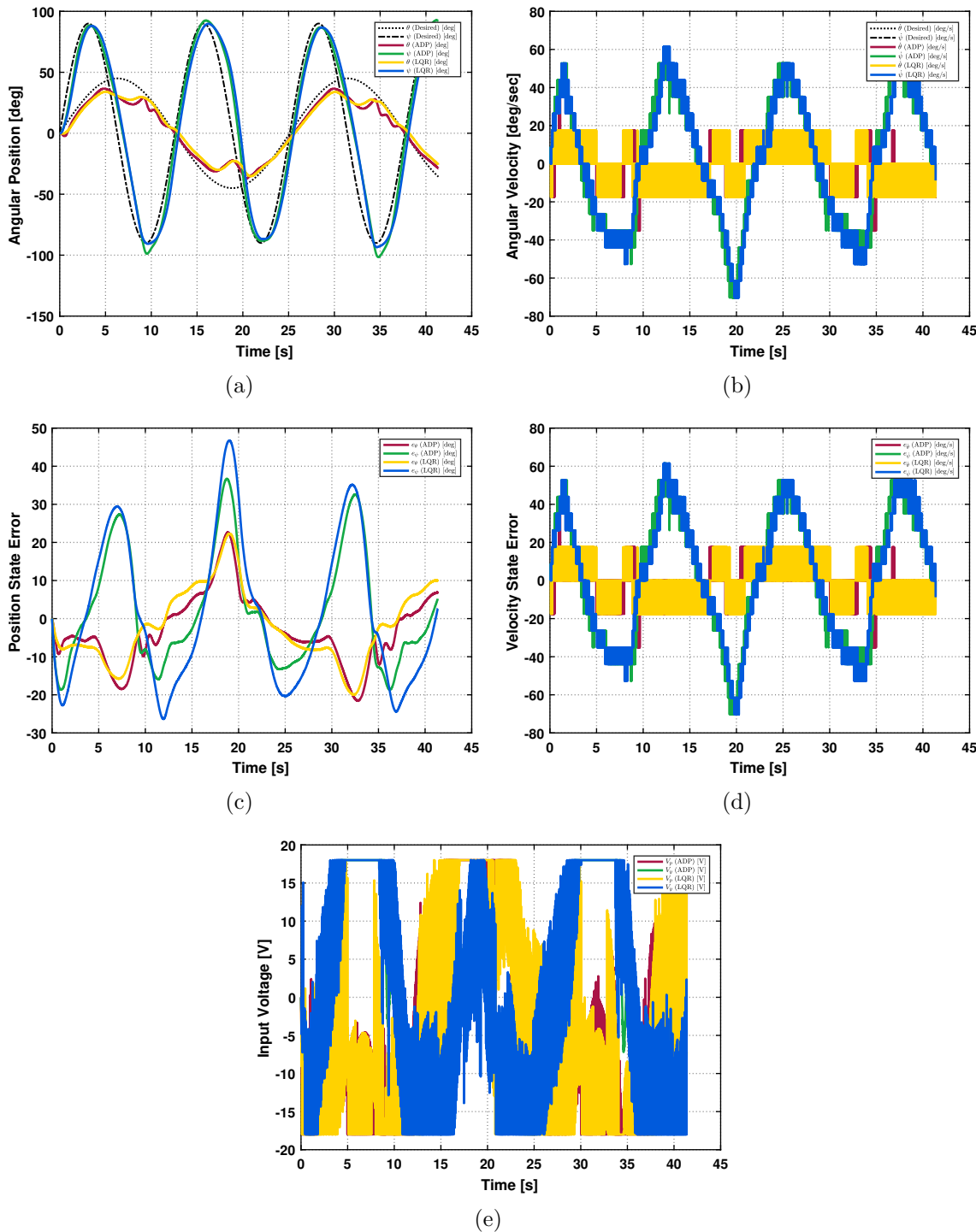


Figure 6.11: Sinusoidal desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

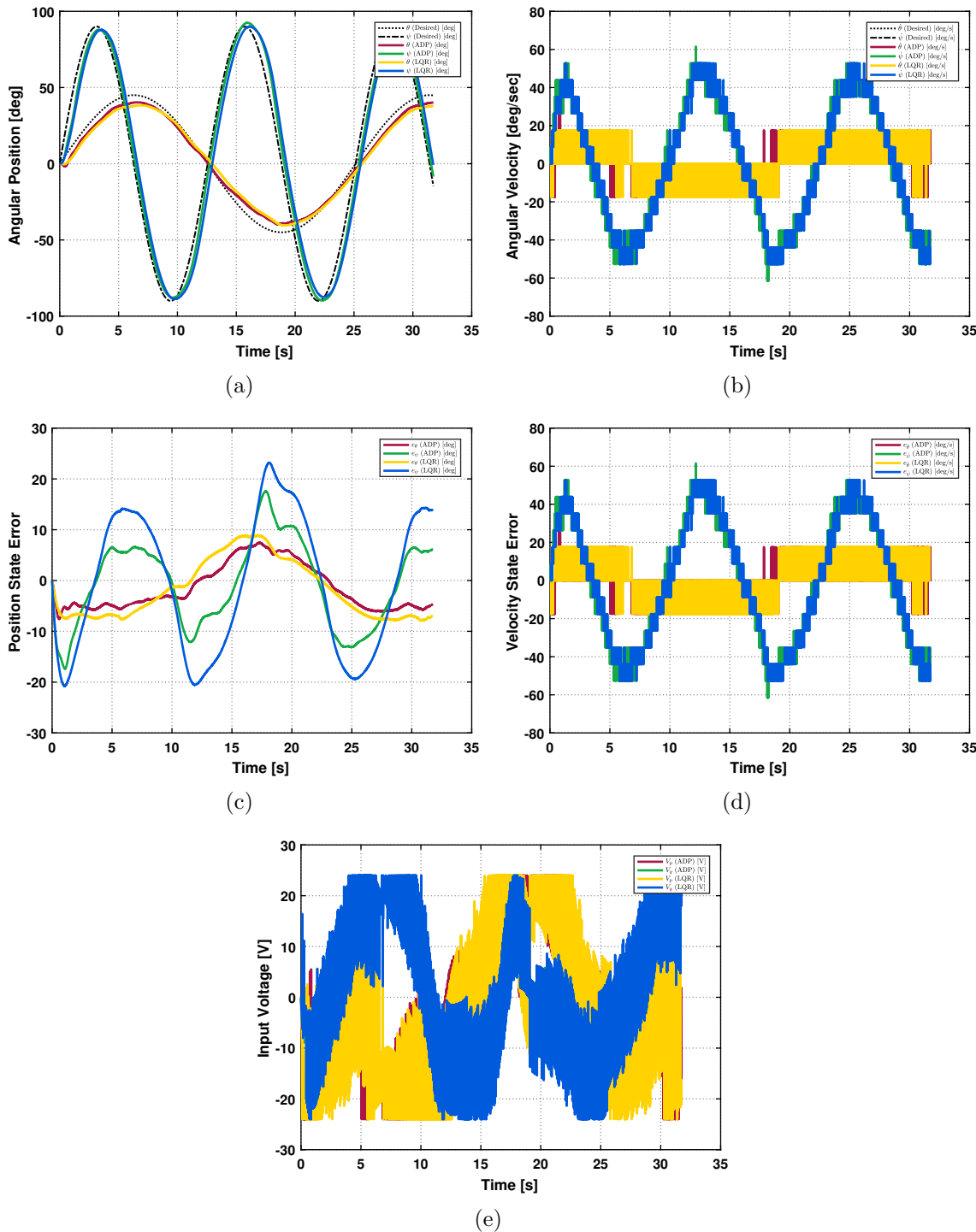


Figure 6.12: Sinusoidal desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

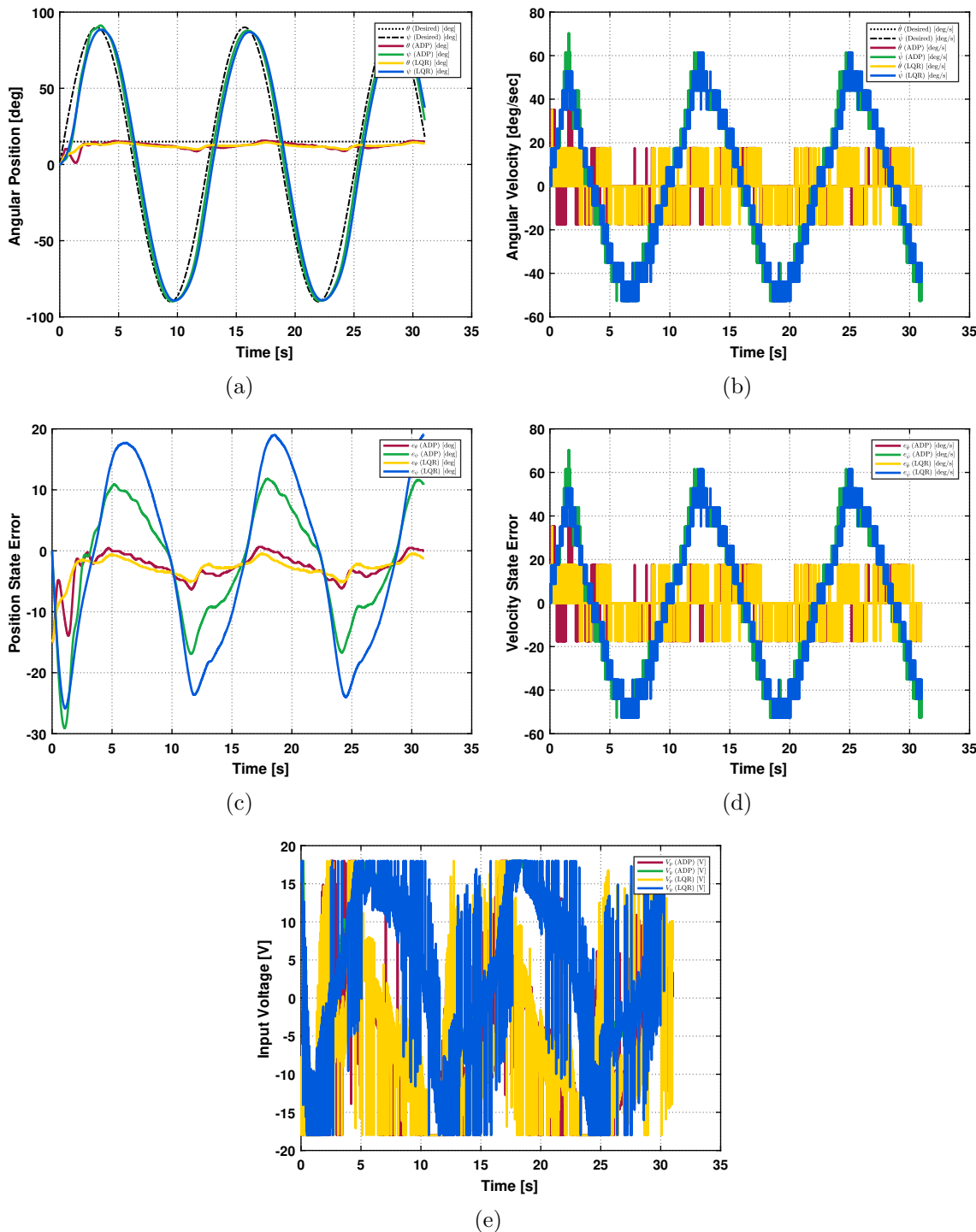


Figure 6.13: Constant desired pitch and sinusoidal desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

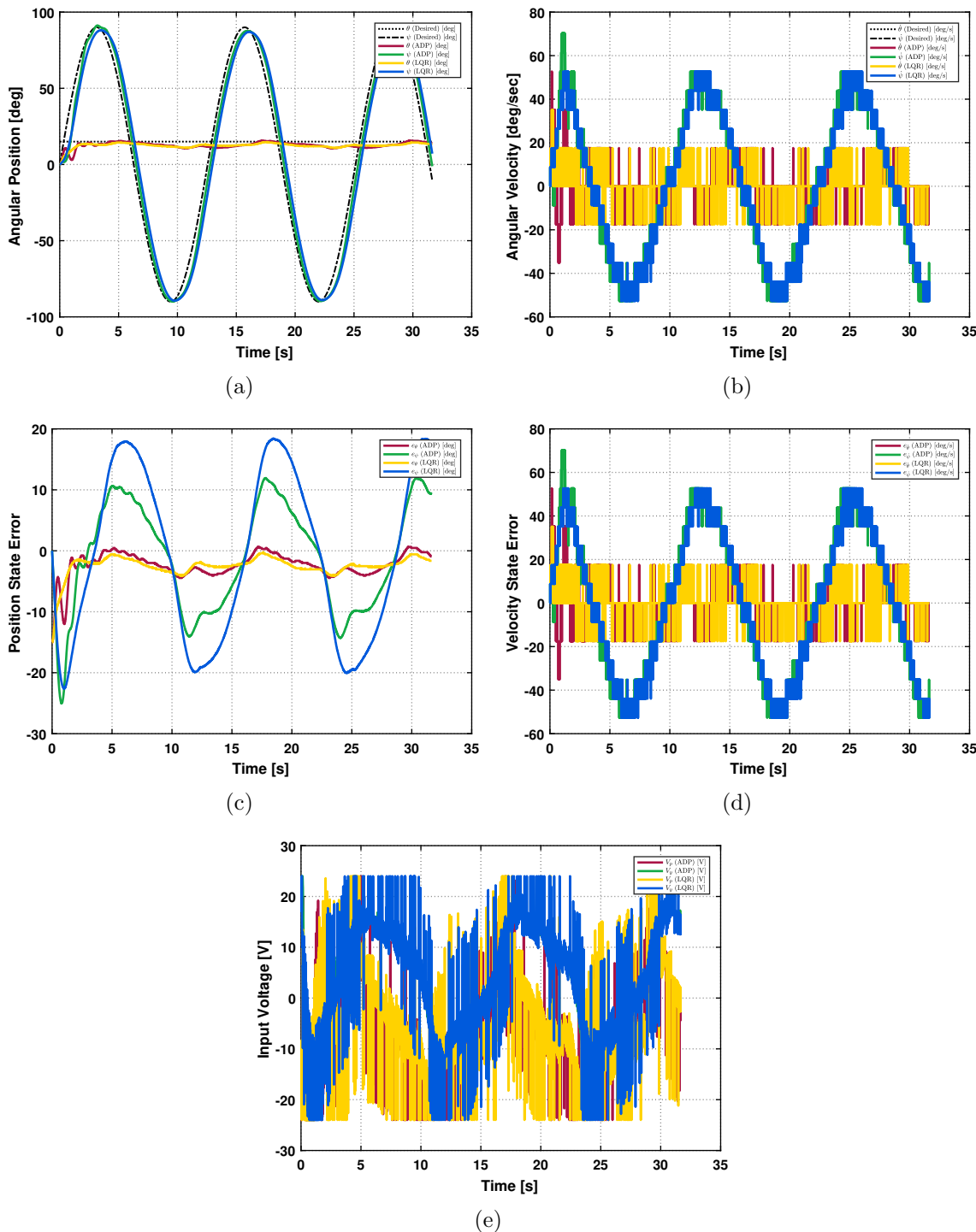


Figure 6.14: Constant desired pitch and sinusoidal desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.



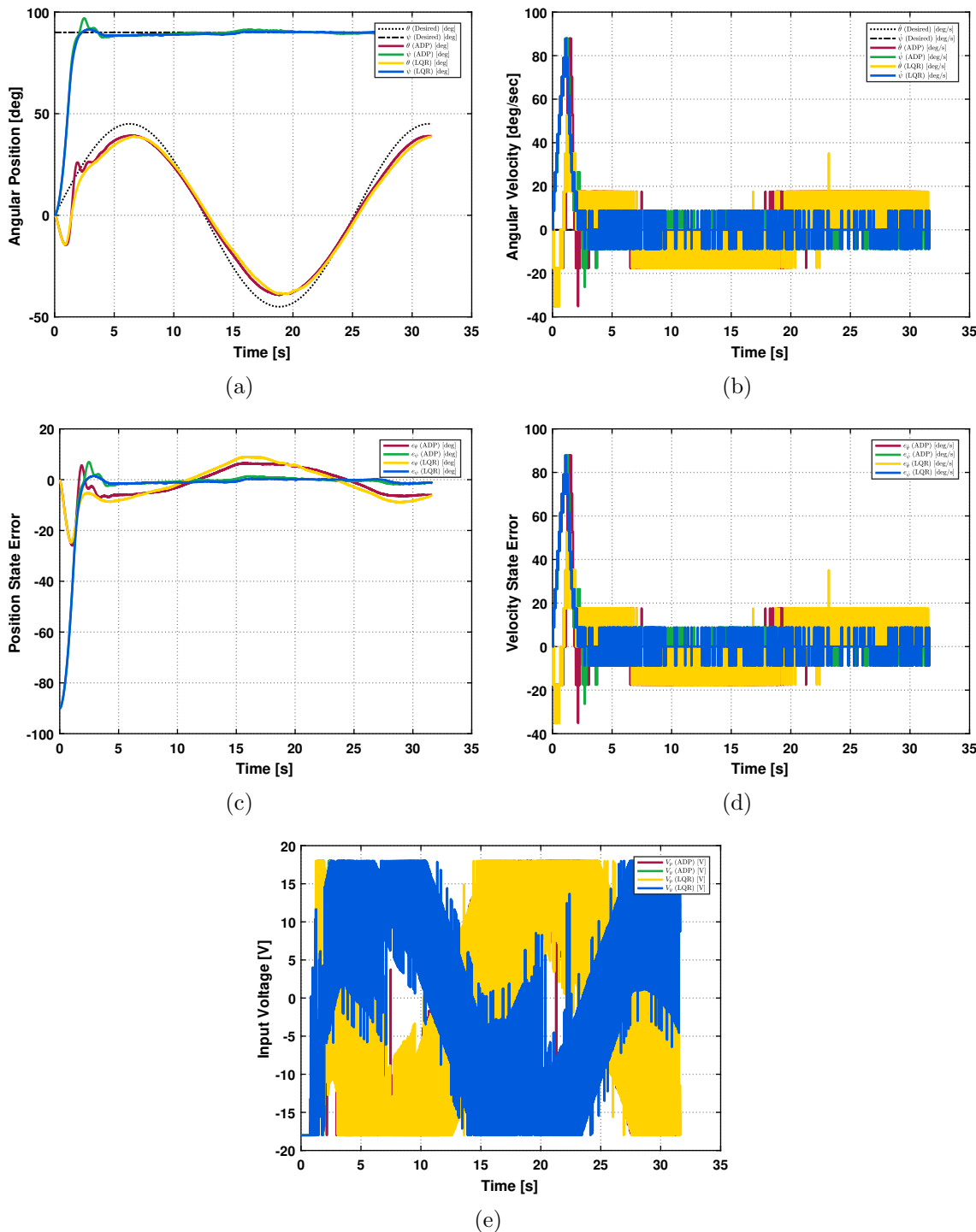


Figure 6.15: Sinusoidal desired pitch and constant desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

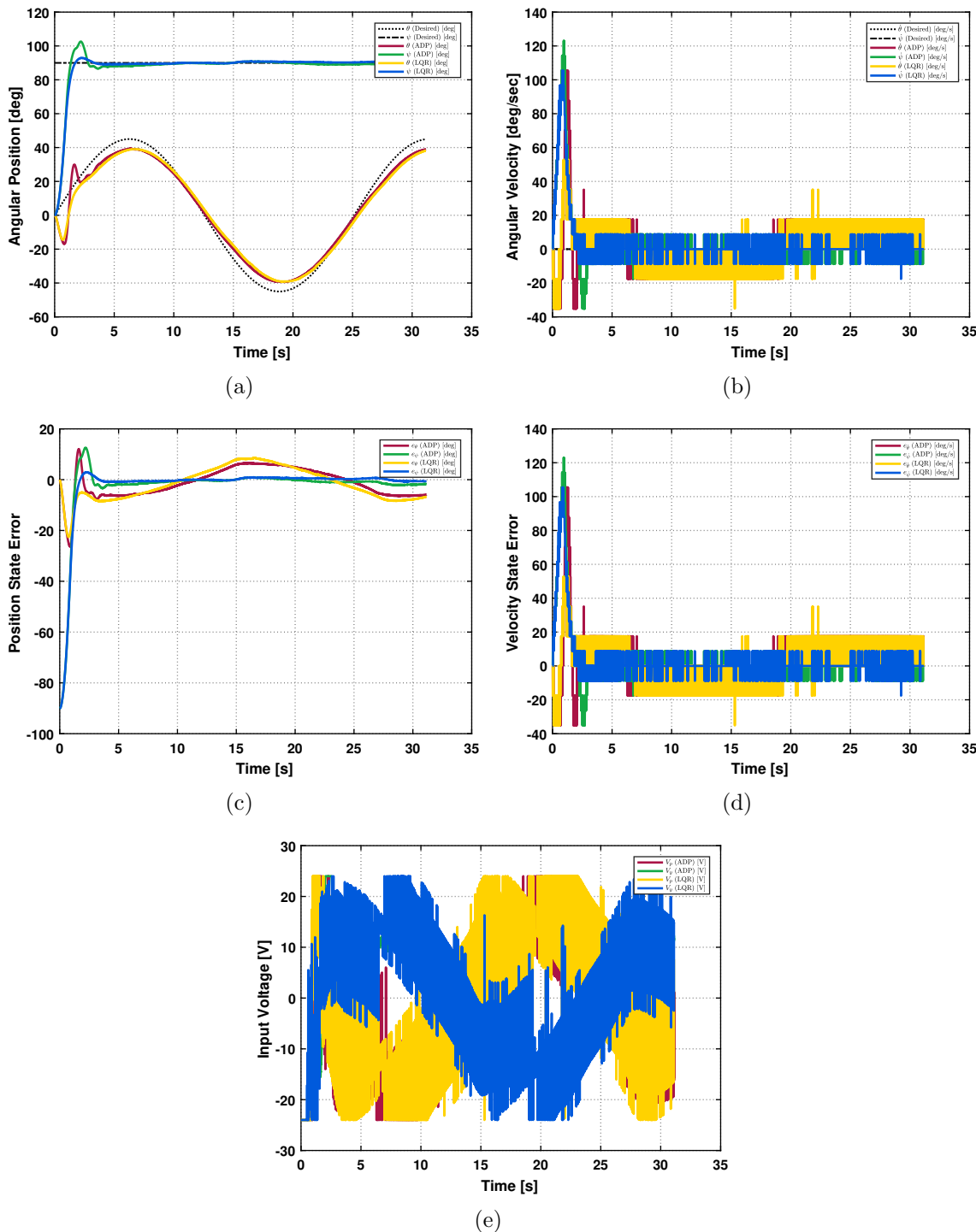


Figure 6.16: Sinusoidal desired pitch and constant desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

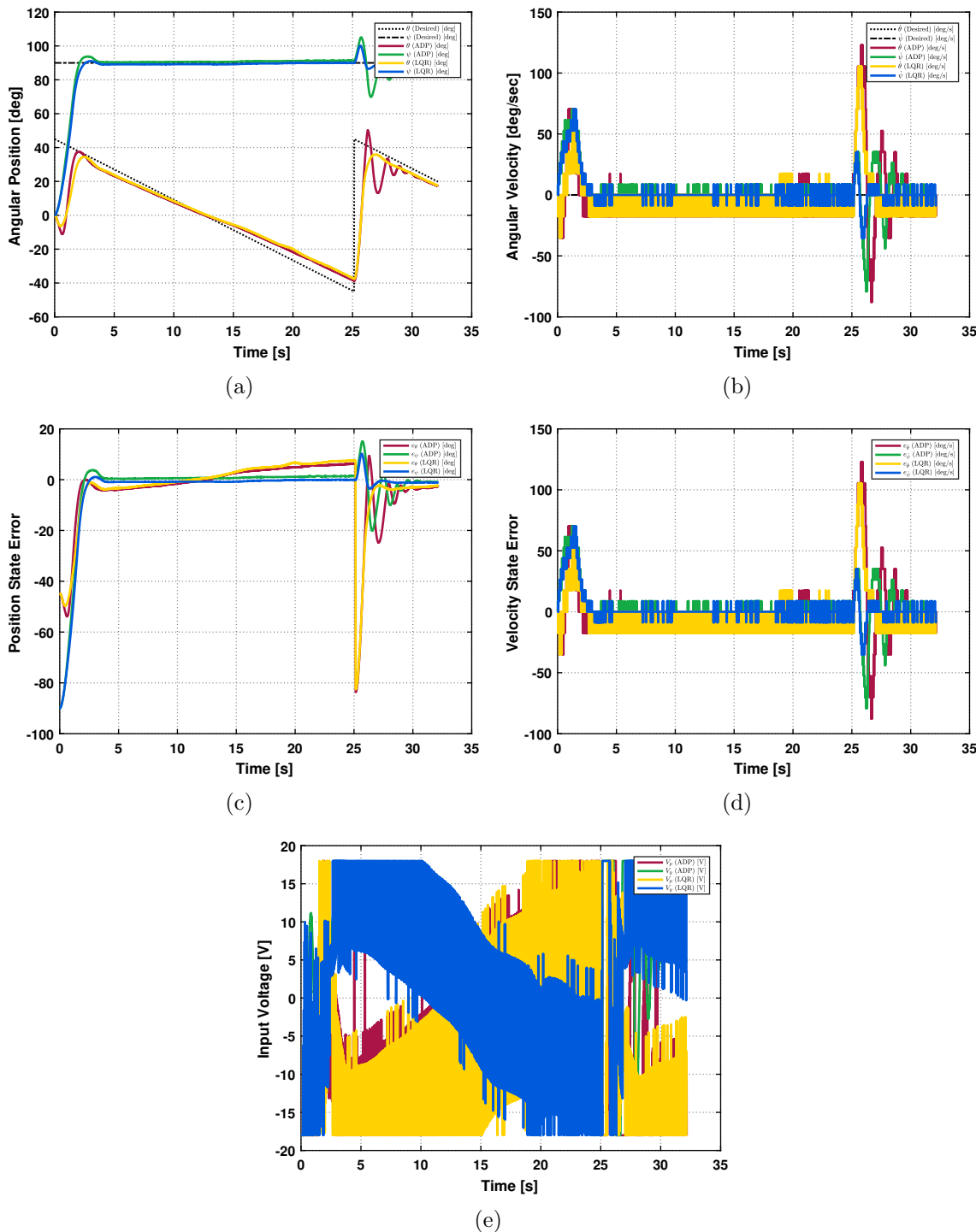


Figure 6.17: Sawtooth desired pitch and constant desired yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

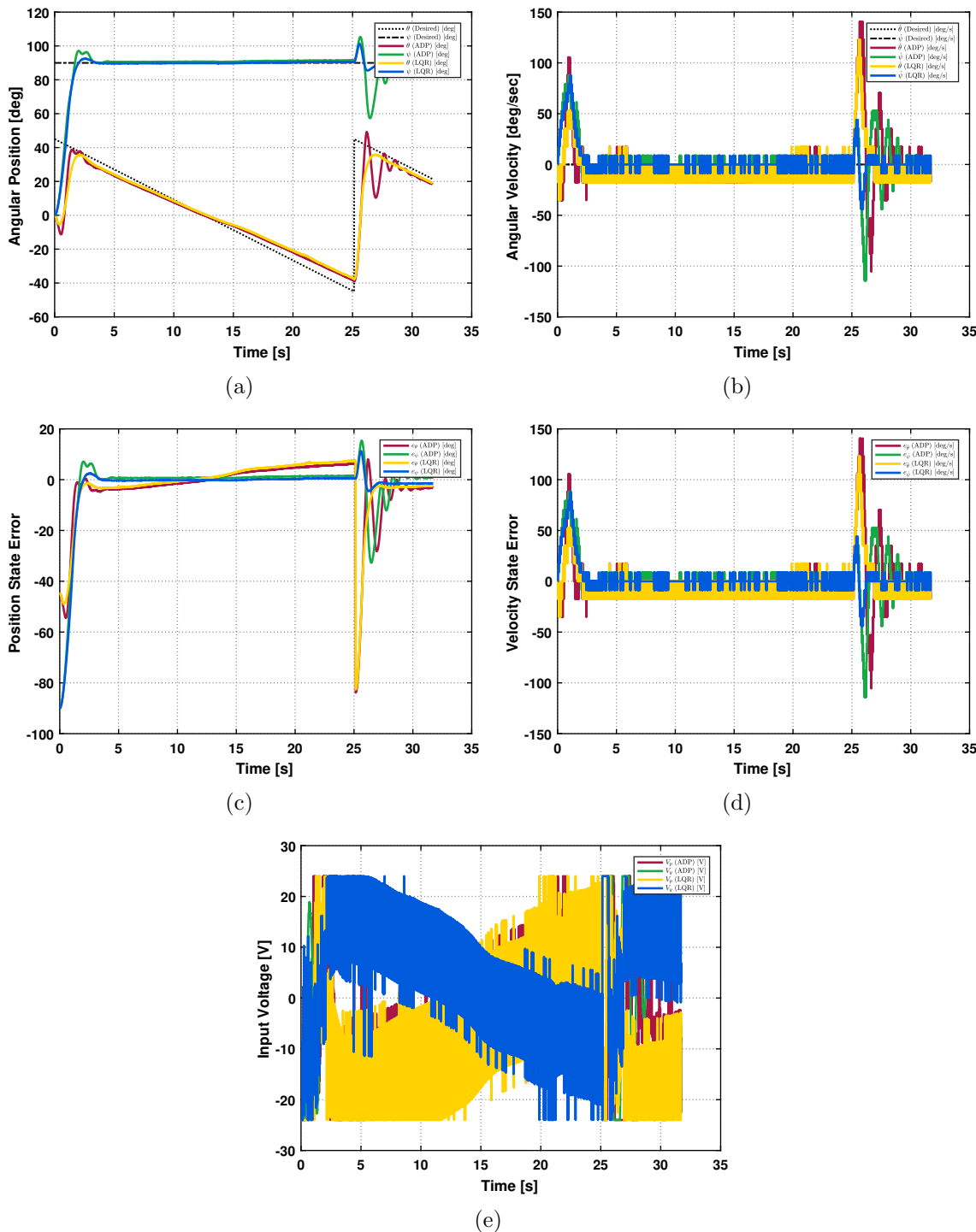


Figure 6.18: Sawtooth desired pitch and constant desired yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

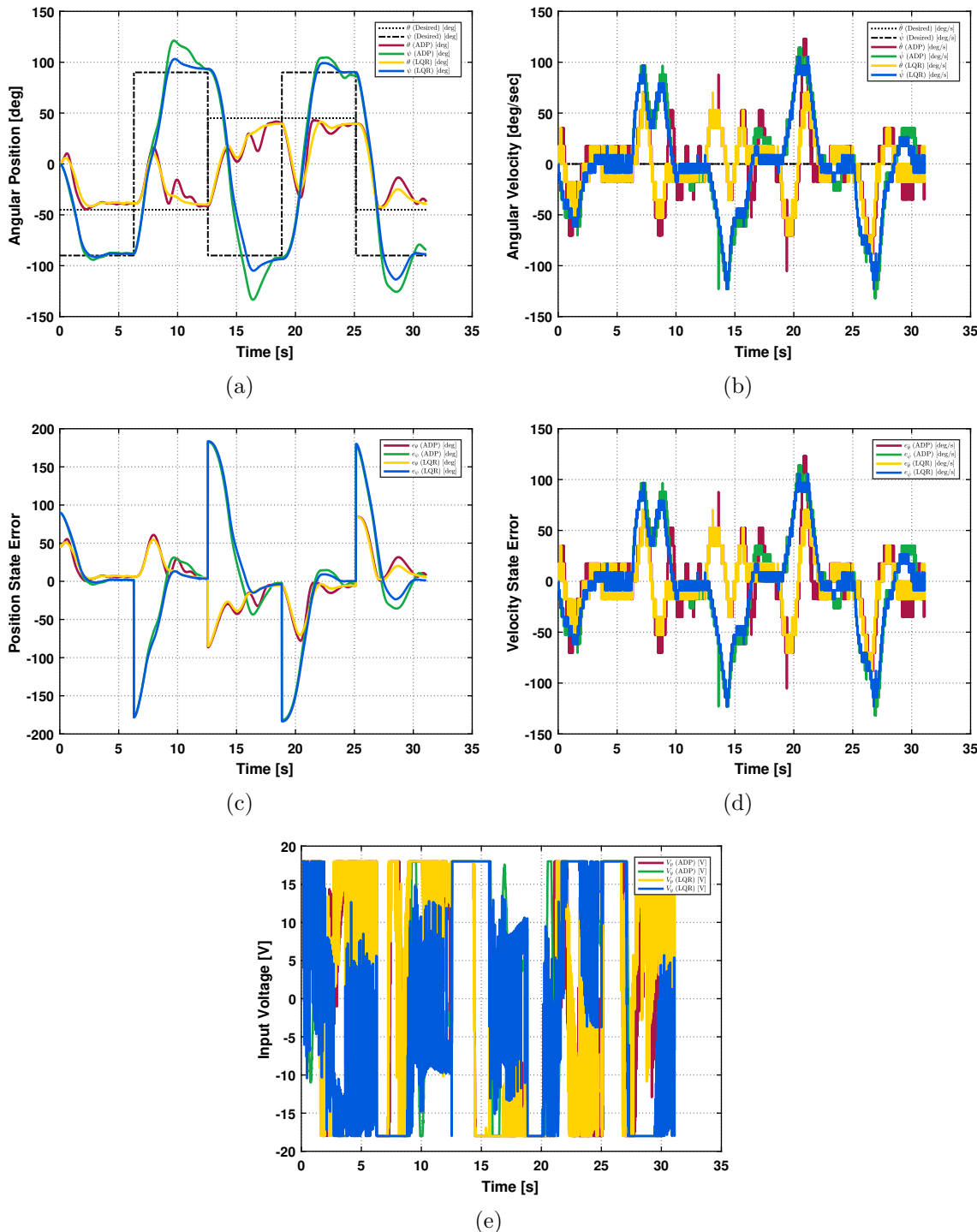


Figure 6.19: Square desired pitch and yaw with inputs saturated to 18 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

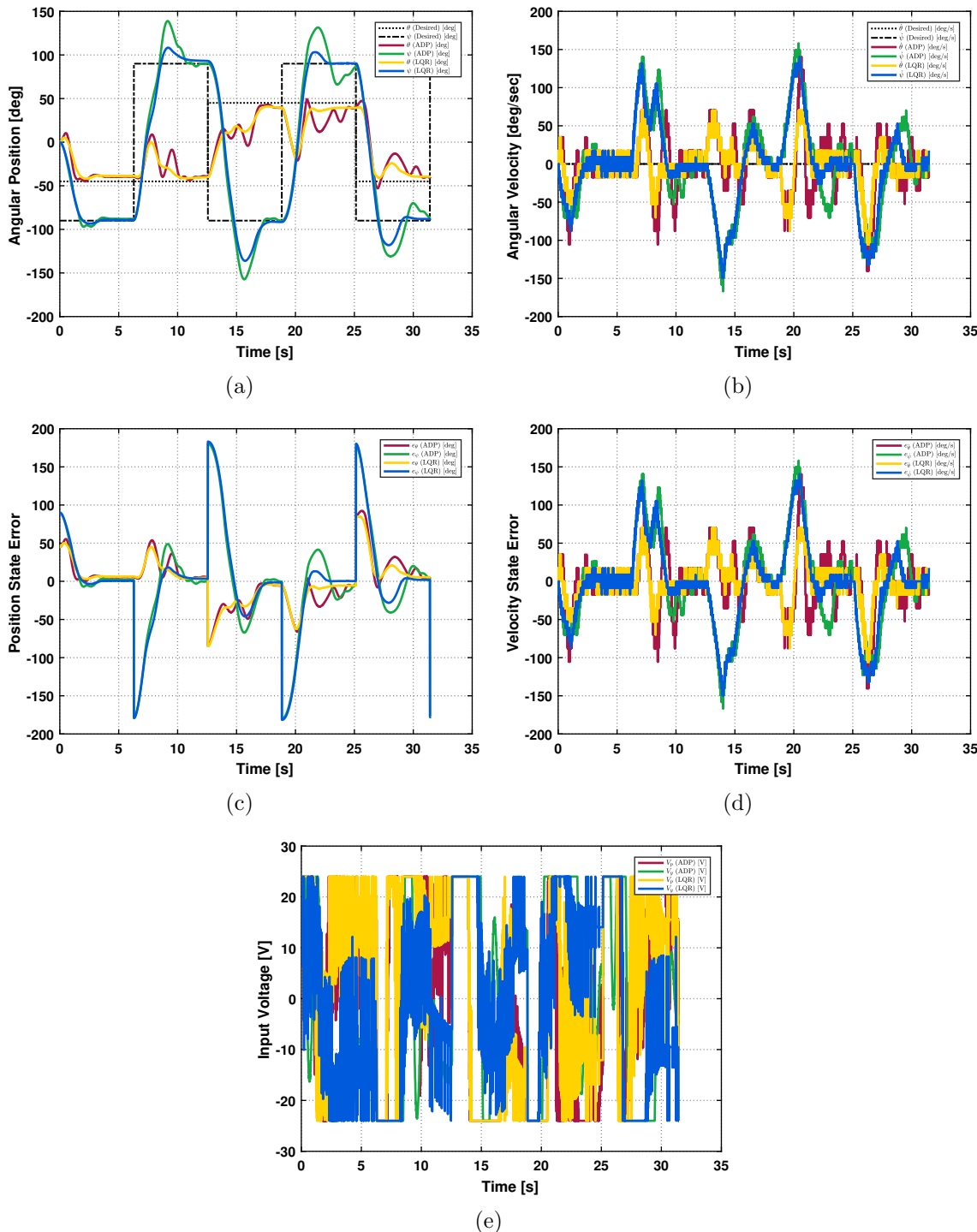


Figure 6.20: Square desired pitch and yaw with inputs saturated to 24 Volts. (a) Angular position. (b) Angular velocity. (c) Angular position error. (d) Angular velocity error. (e) Voltage inputs.

Table 6.2: Experimental Simulink angular velocity RMSE measurements.

Figure	$\dot{\theta}$ ADP [deg /s]	$\dot{\theta}$ LQR [deg /s]	$\dot{\psi}$ ADP [deg /s]	$\dot{\psi}$ LQR [deg /s]
6.7	15.338	22.159	11.205	20.706
6.8	19.229	26.309	11.923	23.233
6.9	34.616	20.821	24.156	15.431
6.10	38.509	28.835	23.992	17.961
6.11	11.556	33.329	10.588	32.736
6.12	10.892	32.345	10.616	31.863
6.13	7.162	32.494	5.905	31.616
6.14	6.929	32.827	5.726	31.761
6.15	13.835	13.742	11.990	13.107
6.16	15.412	16.558	12.239	14.713
6.17	22.027	16.116	15.976	12.396
6.18	24.220	19.828	16.577	14.104
6.19	32.972	48.646	26.456	44.624
6.20	39.173	58.155	26.308	52.175

Table 6.3: Experimental Simulink power RMS measurements.

Figure	Pitch ADP [W]	Pitch LQR [W]	Yaw ADP [W]	Yaw LQR [W]
6.7	8.076	7.102	7.418	6.332
6.8	13.532	11.845	12.628	9.272
6.9	12.140	13.725	11.994	11.271
6.10	19.356	16.368	21.413	12.878
6.11	8.250	10.956	6.247	8.544
6.12	10.275	14.123	7.019	11.655
6.13	5.874	7.358	5.819	6.309
6.14	7.272	9.266	7.241	6.737
6.15	9.305	12.979	5.532	9.620
6.16	11.851	15.868	8.287	10.886
6.17	8.811	9.215	7.004	7.809
6.18	13.406	12.252	12.312	9.589
6.19	11.874	12.993	11.634	10.315
6.20	19.969	19.9	20.693	16.344

## 6.3 QFLEX 2 Embedded/Raspberry Pi

Quanser provides the QFLEX 2 USB panel with the Quanser AERO making the Quanser AERO simple and easy to use. The problem with the QFLEX 2 USB panel is that a Simulink model must be ran with the QUARC software in order to actually control the Quanser AERO. This can become an issue when multiple licenses for QUARC are not an option or an embedded device is desired as the controller. To overcome these setbacks, Quanser has developed the QFLEX 2 Embedded panel. This panel's datasheet can be found in Appendix F.3.

The QFLEX 2 Embedded panel is different from the QFLEX 2 USB panel in the fact that it uses SPI communication from an embedded device to the Quanser AERO. The SPI protocol will be discussed in more detail later in this section. The universality of the SPI protocol allows most embedded systems to be used for control of the Quanser AERO. In our case, the embedded system is the Raspberry Pi 3. The challenge with the Raspberry Pi is that we have to convert the Simulink model used with the QFLEX 2 USB panel to one compatible with a Raspberry Pi and the QFLEX 2 Embedded panel. These challenges were overcome, but much research and innovation was needed on our part which we will discuss in the following sections.

### 6.3.1 Raspberry Pi 3

The Raspberry Pi 3 was given to us as a design constraint for the hardware implementation. Dr. Miah chose the Raspberry Pi 3 because it is readily available, and it has the functionality for Wi-Fi. Better embedded systems could have been considered, but we implemented ADP within the design constraints of the Raspberry Pi 3. More information on the Raspberry Pi 3 can be found at <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>.

### 6.3.2 Required Software

Associated with the Raspberry Pi is the difficulty in generating code for ADP. We basically had two options on how to implement ADP on the Raspberry Pi using the QFLEX 2 Embedded panel. The first option was to write code from scratch that was compatible with the Raspberry Pi and the SPI protocol. This could either be C, C++, Python, etc. The difficulty with this approach would be converting all of the MATLAB code into another language less "math-friendly." This could be considered a project all in its own.

Luckily for us, we found a solution that would make the task of generating the code much easier. We considered using Simulink to generate the C-code which we could then use on the Raspberry Pi. We would then have to manipulate the C-code such that the SPI protocol could be observed. This would, again, be an extreme task.



After some research, we found the solution. We found two support packages that help with code generation for the Raspberry Pi family. The Raspberry Pi Support Package for MATLAB (<https://www.mathworks.com/hardware-support/raspberry-pi-matlab.html>) and the Raspberry Pi Support Package for Simulink (<https://www.mathworks.com/hardware-support/raspberry-pi-simulink.html>) make communication between MATLAB/Simulink and a Raspberry Pi possible. The Raspberry Support Package for MATLAB is more intended for running code using MATLAB on the Raspberry Pi. On the other hand, the Raspberry Support Package for Simulink is intended to generate code to be ran solely on the Raspberry Pi.

These packages can be obtained by going to one's MATLAB. In the Home tab, click on Add-Ons. You can either use Manage Add-Ons or Get Add-Ons and search for Raspberry Pi. These support packages are free, but they do require some setup. The setup includes configuring an SD card for the Raspberry Pi to be used. The SD card is configured with a Linux version specified by MATLAB. To complete the setup just follow the steps provided in the prompts; It is rather simple. Once the setup is complete, you can create a Simulink model and run the generated code on a Raspberry Pi. A tutorial of this can be found in Appendix D.1.

It should also be noted that we were using MATLAB 2016b for the extent of this project. Newer versions of MATLAB correspond to newer versions of the support packages which may have additional features.

### 6.3.3 SPI Communication

In order to utilize the QFLEX 2 Embedded panel, transmission between an embedded system and the Quanser AERO must be achieved via SPI communication. SPI stands for serial peripheral interface. In a nutshell, SPI is the process of sending data between two devices one bit at a time. These bits can be combined to form data needed by each device. An overview of the SPI communication process can be seen in Figure 6.21.

There are four main signals to consider in SPI communication: SS (slave-select), clock, MOSI (master-in slave-out), and MISO (master-in slave-out). One of the devices is the considered the master, and the other device is considered the slave. The master generates a clock signal. This clock signal is used to synchronize communication between the master and slave devices. Essentially, one bit is passed for each period of the clock cycle. Different variations of SPI specify when a bit is starting to send and when a bit is supposed to be stable, but these specifications will not be discussed in detail in this report. The master also controls the value of the slave-select signal. This signal is active-low to specify to the slave that transmission is required. The last two signals are MOSI and MISO. MOSI is the signal generated by the master to be sent to the slave. For our case, we are only concerned with the sending the data for the motor voltages and the base colors. MISO, on the other hand, is the signal generated by the slave to be sent to the master. For our case, we are only concerned with receiving the data from the Quanser AERO corresponding to the pitch and yaw measurements.

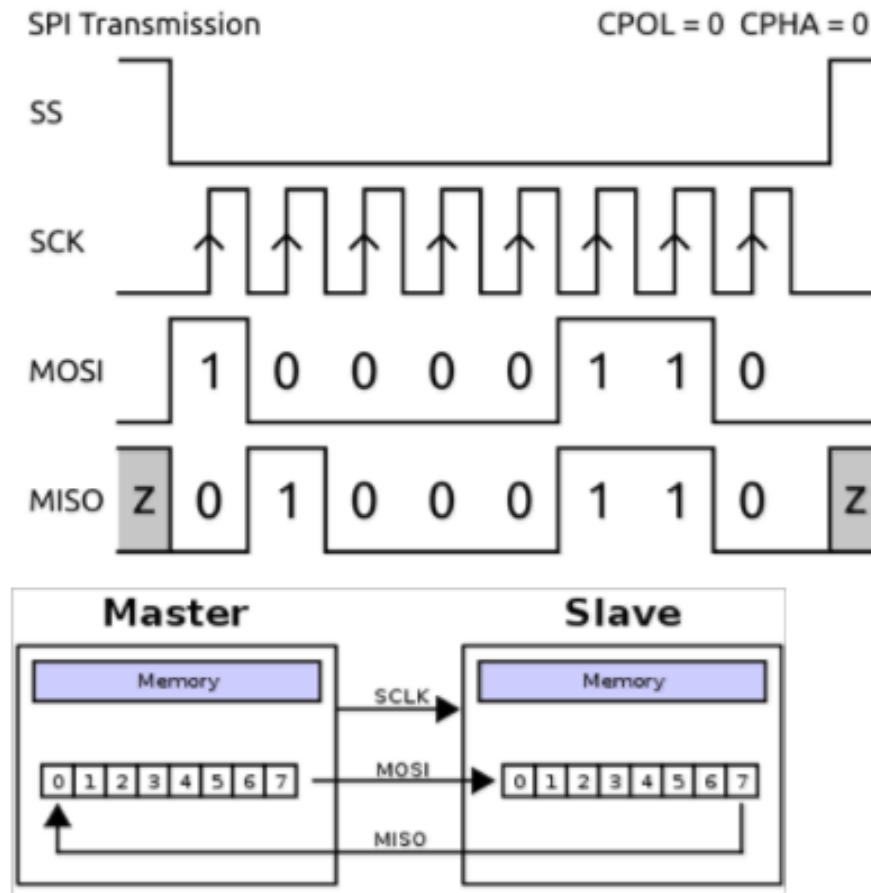


Figure 6.21: Overview of the SPI protocol. Image provided by <https://www.mathworks.com/help/supportpkg/raspberrypi/ug/support-spi-communication.html>.

There are actually multiple difficulties associated with SPI communication that Quanser failed to realize. To begin, SPI communication is intended for short information; that is low numbers of bits to be sent and received. We want to keep the information short, so we can update the information much faster. This is not the case with the Quanser AERO. Appendix F.3 specifies the data sent and received between the Quanser AERO and an embedded device. We can see that for each communication we are sending 51 bytes, or 408 bits. That is a lot of information to send via SPI communication. In our opinion, this is a major design flaw of the Quanser AERO. Specifically, we are only concerned with a few bytes, but we need to send everything for the system function properly.

The only way to make our system fast enough to send all of the bytes in a reasonable time is to increase the SPI clock frequency generated by the master. This is where you can run into hardware limitations. In the next section, we will discuss that Simulink can be used to generate signals on pins of the Raspberry Pi, but this can cause some difficulties. When we generated a clock signal on the Raspberry Pi 3, we were not getting any communication between the Quanser AERO and the Raspberry Pi. We set the clock period at 2 microseconds. With this period, we can send the 51 bytes in between each data

collection in ADP. This seemed reasonable, but there was no communication.

After much thought, we decided to connect an oscilloscope to the Raspberry Pi pin that was supposed to generate the clock signal. There was no clock signal. Why is that? We decided to try larger clock periods. When we reach a clock period of 100 microseconds, we actually begin to see a clock signal that is accurate. It appears that the Raspberry Pi is incapable of switching its GPIO pins at very fast rates. This seemed very odd given that the built in SPI clock of the Raspberry Pi can run in the MHz range. We, however, could not access this built in SPI clock, so we had to generate our own clock signal which was not very fast. So, we have a 10 kHz clock signal. That means we will update the motor voltages every 0.051 seconds. This is not the value of  $\tau$  we anticipated in the ADP. With this type of  $\tau$ , the gain of the state-feedback system is updated roughly every 1 second which is at all not ideal. This is, however, what we are dealt because of hardware limitations. This could either make or break the implementation of ADP.

There is also one more issue we found with Quanser's implementation of SPI communication. The standard is to send the least-significant bit first in the communication. By trial-and-error and some research, we determined that Quanser sends the most-significant bit first. This design is not at all conventional. This can make troubleshooting very difficult if this is not known.

### 6.3.4 SPI Communication in Simulink

We already knew the capabilities of Simulink in generating C-code for the QFLEX 2 USB panel. We also found a Simulink support package for the Raspberry Pi as discussed earlier. Our thought was can we combine these two aspects in Simulink to generate C-code to be used on the Raspberry Pi? The answer is yes we can. To do this, we need to take our ADP Simulink model and convert it to SPI communication that can be used on the Quanser AERO. The model used for this whole process can be seen in Figure 6.22.

There are two basic subsystem blocks in Figure 6.22. The subsystem block on the left is ADP. This subsystem block has the exact same structure used in the previous models. The structure can be seen in Figure 6.23. We kept the same structure, but we made ADP its own entity. Remember, we needed an initialization function to determine the gain for the first  $T$  seconds. We still use this approach, but we also add commands to set the timing of the SPI communication which will be discussed next. We also secure a connection to the Raspberry Pi through MATLAB and enable the GPIO pins of the Raspberry Pi. This initialization code can be seen in Appendix D.2.

The other subsystem block seen in Figure 6.22 is the SPI communication block which is zoomed in on in Figure 6.24. In Figure 6.24, we can see that the SPI communication block is only triggered on the rising-edge of a clock signal generated by a GPIO pin. We already discussed the hardware limitations of this pin earlier. The inputs to the subsystem are the calculated motor voltages and the color values for the base which were found in ADP. The outputs of the subsystem are the current pitch and yaw values.

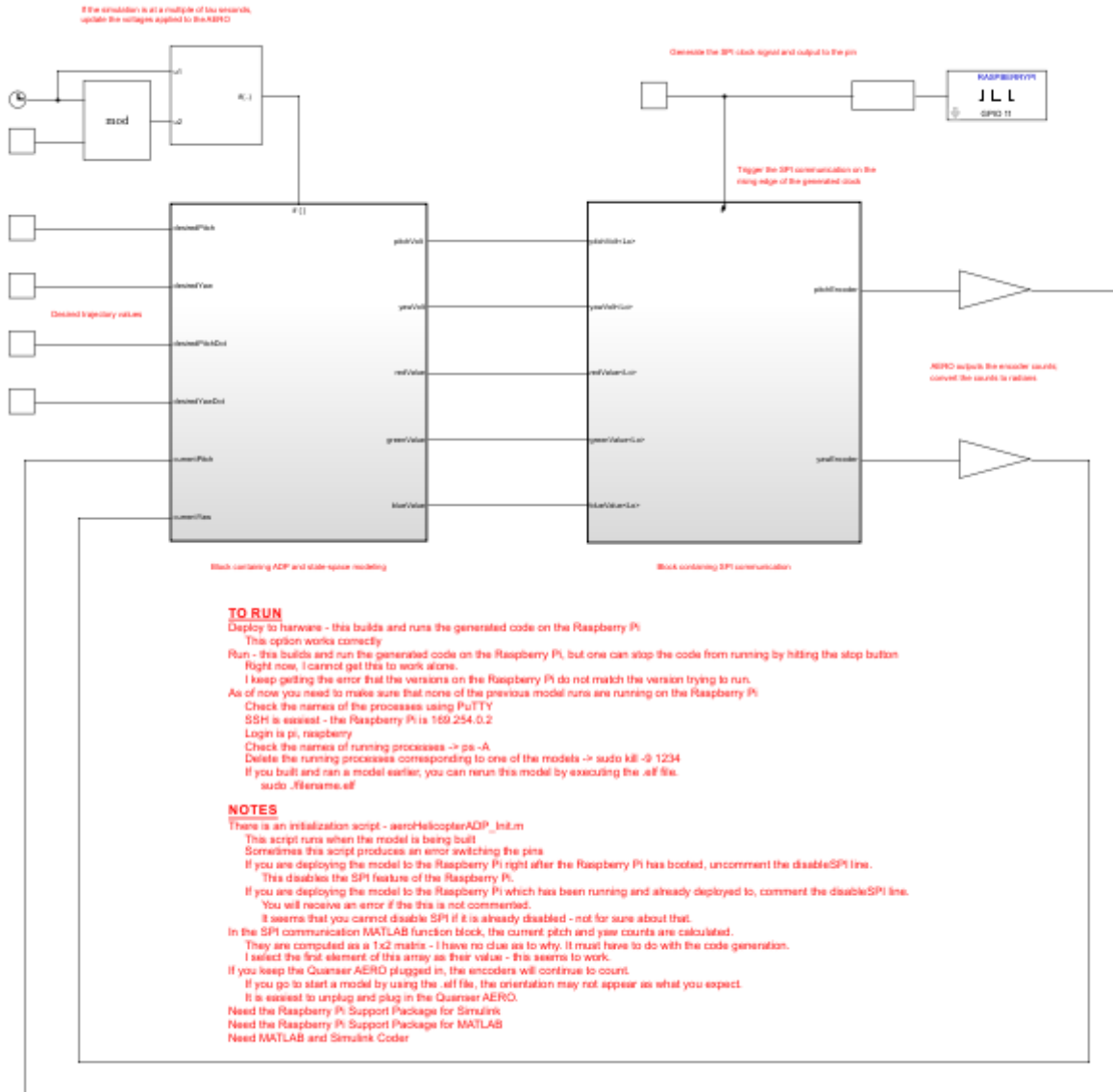


Figure 6.22: Simulink model used to generate the code for the Raspberry Pi 3.

Inside the SPI communication subsystem block, we can see what is really happening in Figure 6.25. Figure 6.25 shows how the information to be sent and received in the SPI communication is handled. Determining what bit to send is handled by a MATLAB function which can be found in Appendix D.3. The MATLAB function follows the transmission information provided in the QFLEX 2 Embedded datasheet found in Appendix F.3. The difficulty with this is we need to keep track of what bit and byte we are currently sending and receiving. This is why we have Data Store blocks in Figure 6.25. These Data Store blocks act as global variables for the MATLAB function. The MATLAB function in Appendix D.3 takes the information about the bit and byte we are currently sending along with the voltage and color information and sets the GPIO pins of the Raspberry Pi. Please refer to Appendix D.3 for the MATLAB code and to Appendix F.3 for how we thought

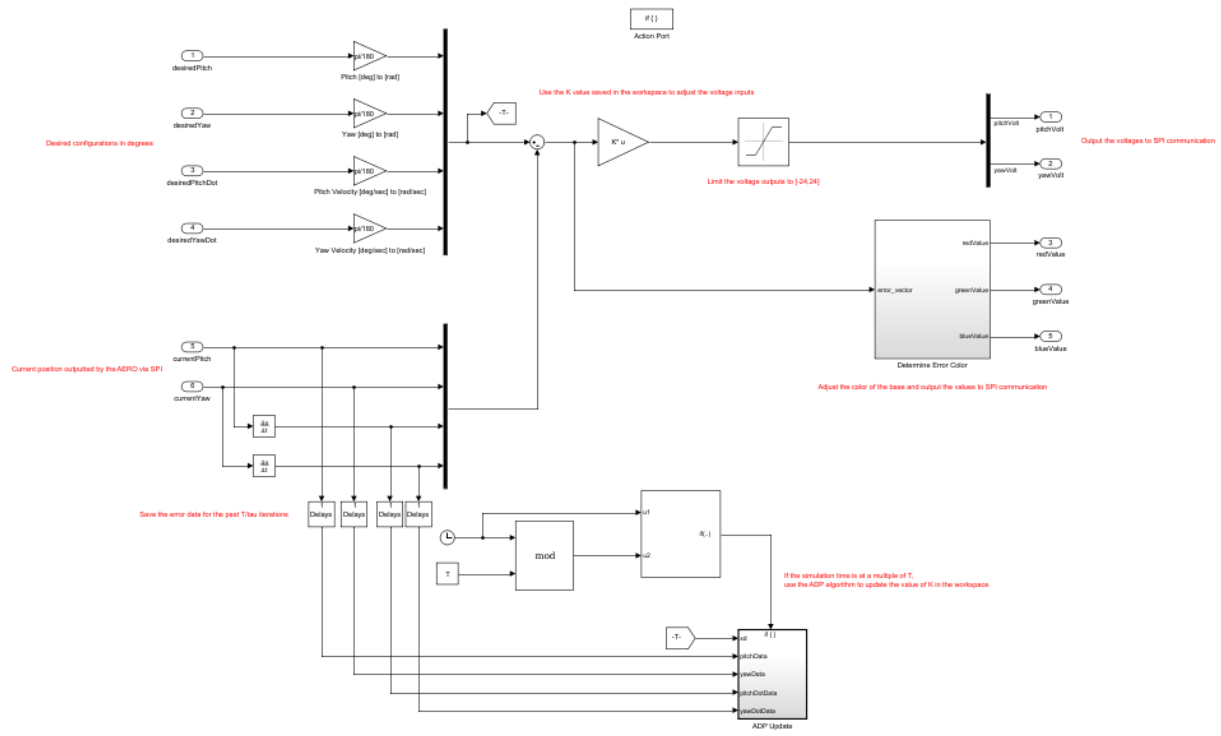


Figure 6.23: Simulink subsystem used for the ADP algorithm.

out the MATLAB code.

### 6.3.5 Results

As for the results with the Raspberry Pi, it is quite difficult to get a numerical analysis. We use Simulink to generate the C-code onto the Raspberry Pi. With that method, there is no way to record data that we know of. This will require further research. The only result we can get is physically looking at the motions to determine if the Quanser AERO is in the correct position.

When we generated the C-code using Simulink, we kept getting one persistent error. We had a size mismatch because the code generation made the current pitch and yaw values  $1 \times 2$  matrices. We have no idea as to why these values are matrices; they should only be single values. There are two options as to why this happened. The first option is the C-code generation is just wrong. The second option is the way we wrote the MATLAB function. We could have written the MATLAB function in such a way that the C-code generator interpreted it as a matrix instead of a single value. This might have worked and been undetectable in MATLAB, but the C-code generator experienced problems. Either way, we do not know the cause of the error which will require further research.

To overcome the error and proceed as scheduled, we decided to take the first element of the matrix we were supposedly generating. We know this is not the best approach, but at

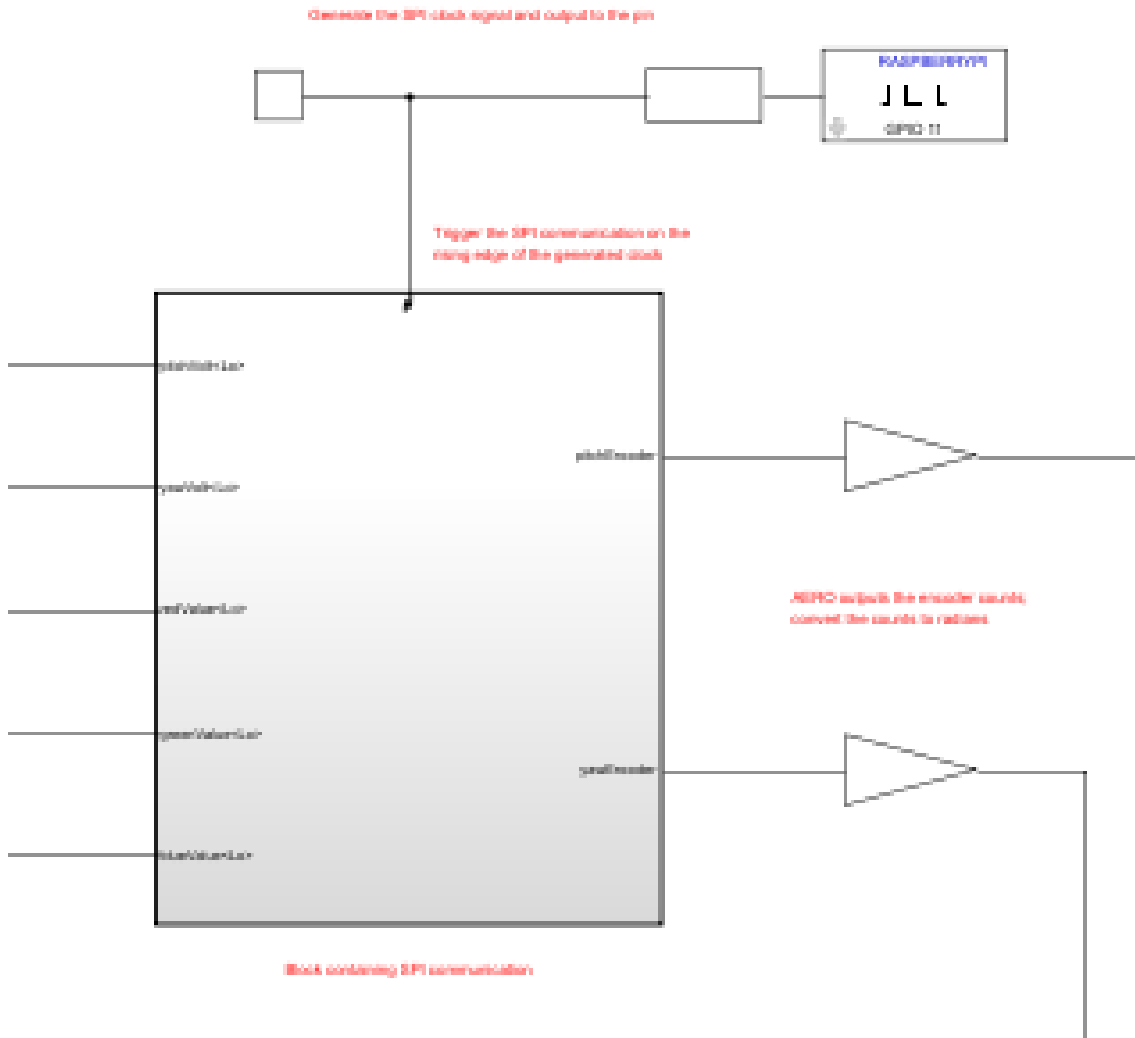


Figure 6.24: Simulink subsystem used specifically by the Raspberry Pi 3.

this point, we were behind schedule, and we needed some results. With that fix, Simulink was able to successfully generate the code on the Raspberry Pi. A full tutorial of how to execute the code is given in Appendix D.1. When we executed the code, we could see the results. It appears that the Quanser AERO goes to the position specified in the code. Because of our error and the hardware limitations discussed, the position is not perfect. The Quanser AERO appears to oscillate around the desired position. We believe, because of the timing, that the state-feedback gain is not being updated fast enough. The Quanser AERO overshoots the desired position, so it is forced to go back on the next update. There is really nothing else we can do given the hardware and algorithm constraints.

Even though ADP works but not very well in this case, we did show a proof of concept. The proof of concept is that we are able to control the Quanser AERO using a Raspberry Pi

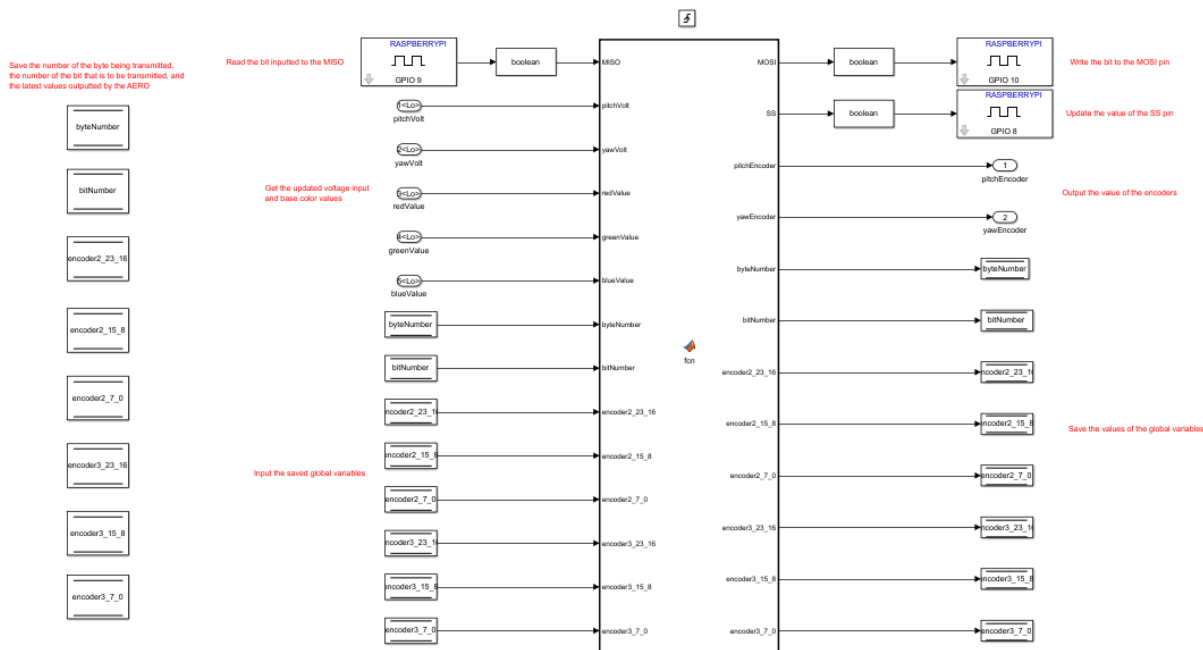


Figure 6.25: Simulink subsystem used to establish communication between the Raspberry Pi 3 and the Quanser AERO.

running ADP. This means ADP should be portable to other physical systems and embedded systems. The one drawback is that the desired position of the Quanser AERO has to be preprogrammed on the Raspberry Pi.

## 6.4 Raspberry Pi/Android

Our original objective of the project was to control the Quanser AERO via a Raspberry Pi and smart phone. Due to the difficulties with the C-code generation for the Raspberry Pi, we were unsure we would be able to complete the smart phone objective.

It turns out that by using Simulink, we actually made the process a lot easier. Simulink is actually able to create smart phone applications that can communicate with a Raspberry Pi. We literally had everything ready except for the application on the smart phone.

But, what is the point of the smart phone application? The development of a smart phone application can show another proof of concept. What if your system to be controlled is remote or your are actually flying an unmanned aerial vehicle? You cannot preprogram the embedded system like we did in the previous section; you need some sort of communication such as a smart phone.

### 6.4.1 Required Software

Because we are using the same concepts that we used for generating the C-code for the Raspberry Pi, we do not need much additional software. We do, however, need something to develop the smart phone application. Just as there was a Raspberry Pi Support Package for Simulink, there is an Android Support Package for Simulink (<https://www.mathworks.com/hardware-support/android-programming-simulink.html>). A helpful reference is located at <https://www.mathworks.com/videos/control-raspberry-pi-from-your-android-device.html>. This package can be obtained by going to one's MATLAB. In the Home tab, click on Add-Ons. You can either use Manage Add-Ons or Get Add-Ons and search for Android. This support packages is free, but it does require some setup. The setup includes setting your smart phone to allow you to develop applications. Follow the prompts in the setup, and the configuration is simple.

Because we will be using Wi-Fi as our communication medium, we also need a network. We need a local network connection for both the smart phone and the Raspberry Pi. We need a local network because it is much easier than the encrypted school network.

We also need to know the IP addresses of the smart phone and the Raspberry Pi once they are connected to the network. This task can be made simpler by downloading a device manager for your smart phone. We downloaded the IP Tools application from the Google Play Store. To manage the IP addresses of the devices on the network, we set the IP addresses as static using the router settings.

### 6.4.2 Simulink Model for Android Applications

Once we have all of the software set up for the Android smart phone application, we can begin updating the Simulink models. The Simulink model for the Raspberry Pi does not need to be edited significantly. We only need to change how we receive the desired pitch and yaw and also what we do with the current pitch and yaw. The updated Simulink model can be seen in Figure 6.26. This is the exact same Simulink model that was previously implemented. The only difference is how we are receiving the desired pitch and yaw.

The Simulink model in Figure 6.26 makes use of the UDP Send and Receive blocks. The UDP Receive block is specific for the Raspberry Pi. The only information needed in this block is the local port of the wireless network. This port number is specified in the UDP Send block. On the outputs of the current pitch and yaw, we see the UDP Send block. We are sending the UDP packet to an Android device, but we are doing it from a Raspberry Pi, so we need to use the Raspberry Pi UDP Send block. In the UDP Send block, we specify the local port number of the wireless network we want to connect through. We also need to specify the IP address of the receiver of the UDP packet. In our case, this will be the Android smart phone's IP address.



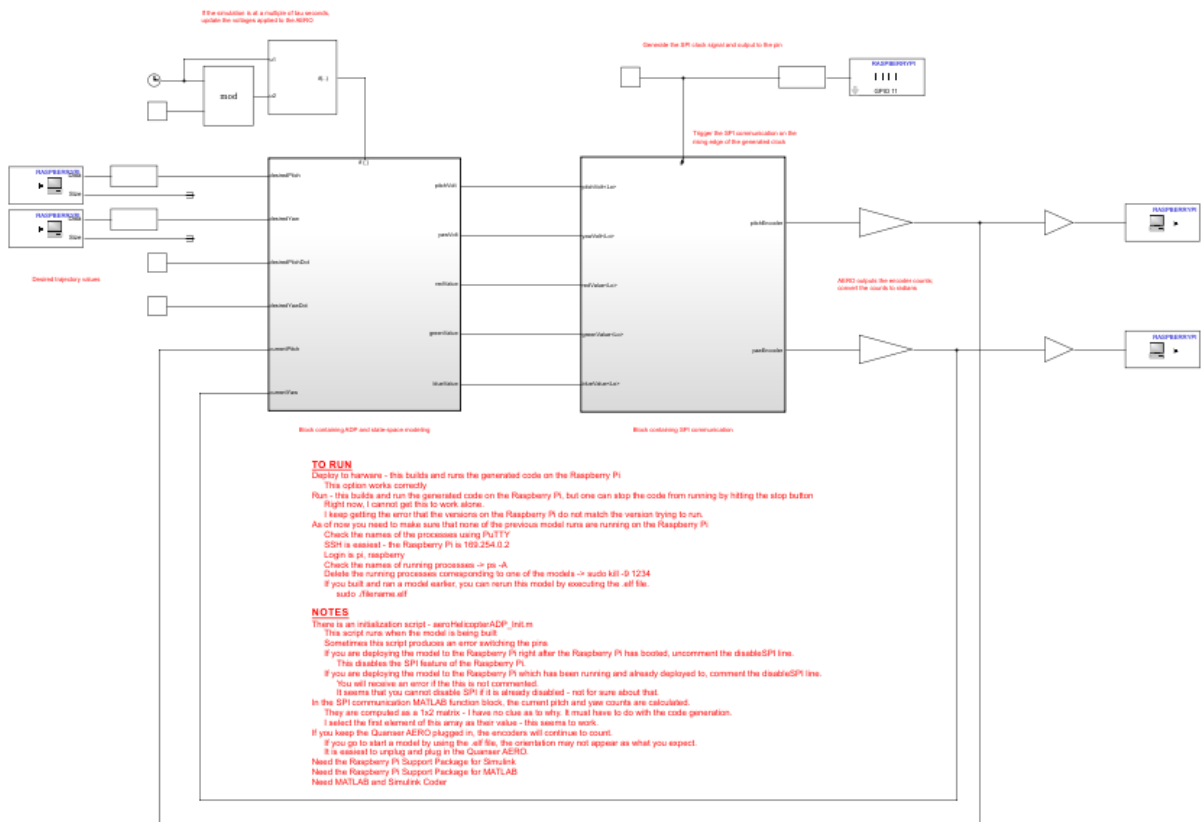


Figure 6.26: Simulink model used to communicate with the Android smart phone.

On the other end of the Wi-Fi communication is the Android smart phone. As discussed earlier, there is an Android Support Package for Simulink. With this package, we can create Simulink models that run on Android devices. The Simulink model can essentially be an application on the smart phone. Just as we did with the Raspberry Pi, we create the model for the Android smart phone, and then we generate the code on the physical device. A tutorial of how to accomplish this is given in Appendix E.1.

In order to generate the application, we need to create the Simulink model. This Simulink model can be seen in Figure 6.27. The Android Support Package for Simulink library has the necessary blocks to create the application. In our case, we used the Android Slider block and the Android Display block. These blocks basically imply what they mean, but they do it in the Android application. The Android Slider block is used to set the desired pitch and yaw. This value is then sent from the Android UDP Send block. The Android UDP Send block specifies the port number to be used on the wireless network. The block also specifies the IP address to where to send the UDP packet. In our case, this will be the Raspberry Pi’s IP address. In this Simulink model, we also utilize the Android UDP Receive block. This block receives the UDP packet of the actual pitch and yaw measurements from the Raspberry Pi model discussed earlier. This value can then be displayed on the Android application with the use of the Android Display block. This model can be generated on the Android smart phone and then ready to use with the Raspberry Pi.

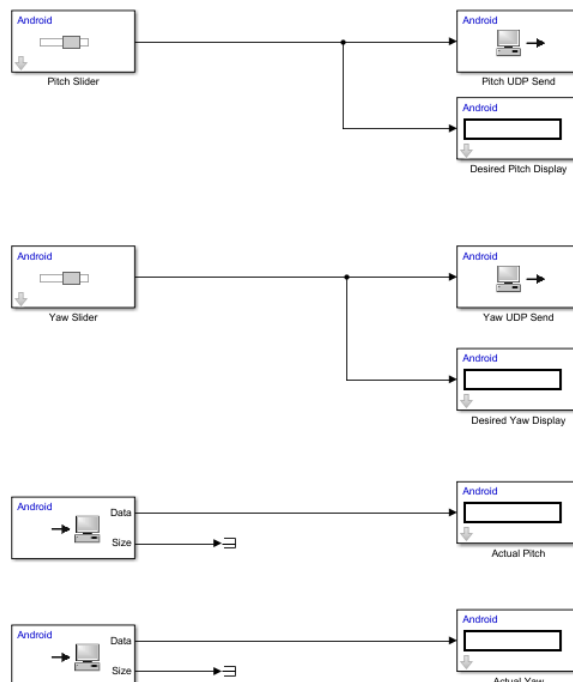


Figure 6.27: Simulink model used to create the Android smart phone application.

### 6.4.3 Results

The results for the smart phone application are more a matter of seeing the application work correctly. The model builds correctly for the Raspberry Pi 3. The model also builds correctly for the Android smart phone, but the build process takes a while longer. It turns out that you need to be connected to a network with Internet for the build process to complete. We are unsure why this needs to be true, but it does. A tutorial of how to run the application and get everything set up can be found in [Appendix E.1](#).

When we run the Android smart phone application, we actually see communication to the Quanser AERO. We see the desired pitch and yaw changing as the sliders are adjusted. We also see the actual pitch and yaw readings on the application as well. Overall, this accomplishes our objective.

# Chapter 7

## Conclusion and Future Work

The work presented in this project successfully followed electrical engineering methodologies. We were given a system and method of control for that system; our job was to make it work. Following electrical engineering methodologies, we started with the system. We derived a highly assumed and linearized model. With the system model in hand, we analyzed ADP modifying it so it would work correctly with the new model. Once we completed the mathematical analysis and modeling, we could begin simulations to see if ADP actually worked for our system.

Simulations were conducted in both MATLAB and V-REP, but linear system models limited the simulation results. For our linear system, simulations showed proper trajectory tracking. V-REP simulations took much longer to complete due to no available Quanser AERO platforms. With successful simulations, we could begin implementation.

For the implementation process, we started simple. We created ADP in the form a Simulink model which we could then apply directly to the Quanser AERO via the QFLEX 2 USB panel. With successful control of the Quanser AERO, we modified the Simulink model slightly for interfacing with the QFLEX 2 Embedded panel. We utilized Simulink support packages to expedite the process. Using Simulink, we also were able to generate C-code that functioned correctly on the Raspberry Pi and Android smart phone.

### 7.1 Conclusion

This project encompassed a large amount of work for only being a year long, but several conclusions can be drawn from that work. It is possible to use ADP on the Quanser AERO, but there are some drawbacks. The Quanser AERO behaves, for the most part, as a linear system; the coupling is not too significant. ADP, thus, could not be tested accurately for this system. It would be just as beneficial to use a conventional linear control technique such as LQR. The results shown in this report show similar results for both methods. In our opinion, it would be more beneficial to use LQR for this system to limit control

complexity and computational requirements. There really is no advantage to using ADP for this particular system.

Even though ADP shows no advantage for this particular system, simulations and implementation were successfully completed for the Quanser AERO. We were able to simulate the Quanser AERO in MATLAB using the system model. The MATLAB simulations were limited in the fact that it is difficult to simulate the nonlinear coupling. V-REP also provided accurate simulation results for LQR, but the complexity of ADP made real-time simulation almost impossible.

For implementation, we conclude that it is possible to apply ADP directly to a Quanser AERO using Simulink or a Raspberry Pi. Implementing the Raspberry Pi is a feat due to the fact that Quanser provides very limited documentation on the Raspberry Pi interfacing. We were able to successfully implement ADP on the Raspberry Pi with SPI communication to the Quanser QFLEX 2 Embedded panel. ADP tracks the desired trajectory reasonably well but limitations noted suggest why perfect tracking is not achieved. Extending our Simulink models and capabilities, we were also able to successfully implement an Android smart phone application that communicates with the Raspberry Pi.

Overall, we conclude that it is possible to modify, simulate, and implement ADP on the Quanser AERO.

## 7.2 Future Directions

Even though we accomplished many objectives in this project, there is still room for improvement and further research. Further research can be broken down into the following categories: system, ADP, and implementation. A better system can be used to test ADP. The Quanser AERO, for the most part, was a linear system. We were not able to see the true capabilities of ADP with such a linear system.

ADP itself can also be researched further. Because our project encompassed so many objective, we were unable to fully validate ADP. The use of the actor neural network can be researched such that the system can be truly learned by the algorithm. We needed an approximate system model in order to utilize ADP. We were also unable to research the optimal sampling times for ADP. We were given these sampling times at the beginning of the project, but are they truly optimal? Lastly, ADP is very slow because of the recursive nature of the algorithm. This was especially detrimental in the V-REP simulations. If ADP could be made faster or streamlined, simulations and implementation code could run much faster and more efficient.

Implementation can also be researched further. We were constrained to the Raspberry Pi, but can other, simpler embedded systems use ADP. With the Raspberry Pi, we were

limited in the switching frequency of the GPIO pins. Further research could determine if there are workarounds to this or if we are truly limited. Above all, further research is needed in determining the error we received from the C-code generator in Simulink. We outputted a matrix instead of a single value, but we do not know the cause of this error. Was it code or compiler error? Or both? We were also limited in our ability to switch the desired trajectory once the C-code was generated for the Raspberry Pi. Further Linux research can be performed to determine alternate methods to change this once the code has been generated.

Our research may have come to an end for this project, but there is plenty more to discover.

# APPENDICES

# Appendix A

## MATLAB

### A.1 ADP Simulation (MATLAB)

```
1 % Andrew Fandel
2 % Use exact the linearized model to find error data points
3 % Use the error data points directly in the neural network instead of
4 % randomizing the error
5 close all; clear; clc;
6
7 testName = 'heli_sineWave';
8 heli = 0;
9 halfq = 1;
10
11 [ A, B ] = getAeroAB(heli);
12 % 2-DOF QUANSER HELICOPTER PARAMETERS
13 % Maximum applied voltage for the rotor motors
14 maxVolt = 18;
15 % Number of state variables – theta, psi, dtheta, dpsi
16 n = 4;
17 theta = 1;    % Pitch
18 psi = 2;     % Yaw
19 thetaDot = 3; % Pitch angular velocity
20 psiDot = 4;  % Yaw angular velocity
21 % Number of inputs – Vp, Vy
22 m = 2;
23
24 % SIMULATION TIMING
25 % Initial and final simulation times [s]
26 t0 = 0; tf = 25;
27 % Sampling time [s]
28 tau = 0.01;
29 % Larger sampling time for updating the inputs [s]
30 T = 0.2;
31 % Number of time steps
32 tsteps = floor((tf-t0)/tau);
33 % Discrete time vecotor of sampling time (tau)
34 dt = tau*(0:tsteps);
```

```

35 % Number of equations for actor-critic neural network
36 % Number of training samples per T
37 nbar = floor(T/tau);
38
39 % COST FUNCTION
40 % Q and R matrices used in the cost function
41 %Q_Mat_ADP = diag([270 100 1 1]);
42 %R_Mat_ADP = 0.0001*diag([1 1]);
43 % Use the matrices used in Quanser documentation except add ones for the
44 % velocities -> need Q to be positive definite
45 Q_Mat_ADP = diag([200 75 1 1]);
46 R_Mat_ADP = 0.005*diag([1 1]);
47 % Modified LQR cost matrices
48 Q_Mat_LQR = diag([200 75 1 1]);
49 R_Mat_LQR = 0.005*diag([1 1]);
50 nonLinearAdjustment = 0;
51 % Name of .mat file to save the data
52
53
54 % INITIAL STATES
55 xInit = [deg2rad(-87) deg2rad(-45) (pi/180)*10 (pi/180)*7]';
56 %xInit = [0 0 0 0]';
57
58 % MATRIX INITIALIZATION
59 % Actual states - state variable x
60 xADP = zeros(n,tsteps+1);
61 xLQR = zeros(n,tsteps+1);
62 % Initialize x with initial conditions
63 xADP(:,1) = xInit;
64 xLQR(:,1) = xInit;
65 % Desired states
66 % Uncomment for desired states of all zero
67 %xd = zeros(n,tsteps+1);
68 % Uncomment for desired states that switch constant values
69 %xd = [deg2rad(37)*ones(1,tsteps+1); deg2rad(80)*ones(1,tsteps+1); zeros(1,
    tsteps+1); zeros(1,tsteps+1)];
70 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
    (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
    ; zeros(1,tsteps+1)];
71 % Uncomment for desired states that are sine and cosine waves
72 pitchSine = deg2rad(60)*sin(0.5*dt);
73 yawCosine = deg2rad(90)*cos(0.5*dt);
74 % pitchSquare = deg2rad(57)*square(0.5*dt);
75 % yawSquare = deg2rad(-36)*square(0.25*dt);
76 xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
77 % xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
78 % State errors for entire simulation
79 errorADP = zeros(n,tsteps+1);
80 errorLQR = zeros(n,tsteps+1);
81 % Initialize the error
82 errorADP(:,1) = xd(:,1) - xADP(:,1);
83 errorLQR(:,1) = xd(:,1) - xLQR(:,1);
84 % Actual inputs
85 uADP = zeros(m,tsteps+1);

```



```

86 uLQR = zeros(m, tsteps+1);
87
88 % CALCULATE THE CONTROLLER GAIN MATRIX USING LQR
89 fprintf('WORKING ON LQR GAIN MATRIX...\n');
90 kLQR = lqr(A,B,Q_Mat_LQR,R_Mat_LQR);
91
92 % FIND THE CRITIC NEURAL NETWORK WEIGHTS BEFORE APPLYING ANY INPUTS
93 % Calculate the critic weights
94 wcInit = quanserAEROCriticTuningInitial(A,B,((2*pi).*rand(4,nbar)-pi),tau,
    R_Mat_ADP,Q_Mat_ADP,xd(:,1));
95 % Use the weights to determine the P matrix
96 P_Mat = [wcInit(5) wcInit(6) wcInit(7) wcInit(8);
    wcInit(6) wcInit(9) wcInit(10) wcInit(11);
    wcInit(7) wcInit(10) wcInit(12) wcInit(13);
    wcInit(8) wcInit(11) wcInit(13) wcInit(14)];
100 % Use P to determine the state-feedback gain
101 kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
102
103 % RUN THE SIMULATION
104 % Start at the next time tau after time zero
105 for k = 2:tsteps+1
106     % Current time
107     t = (k-1)*tau; % generate value discrete time index
108     if (mod(t,5) == 0)
109         % Print the current time
110         fprintf('Current time, t = %g [s]\n',t);
111     end
112
113     % EXACT 2-DOF HELICOPTER MODEL
114     % Take previous values to find the derivative of the exact model
115     xdotNonLinearADP = A*xADP(:,k-1) + B*uADP(:,k-1);
116     xdotNonLinearLQR = A*xLQR(:,k-1) + B*uLQR(:,k-1);
117     % Update the states of the exact model
118     % Use Euler integration to estimate the current states of the nonlinear
119     % model
120     xADP(:,k) = xADP(:,k-1) + xdotNonLinearADP*tau + nonLinearAdjustment
    .* (2.*rand(4,1)-1);
121     xLQR(:,k) = xLQR(:,k-1) + xdotNonLinearLQR*tau + nonLinearAdjustment
    .* (2.*rand(4,1)-1);
122     % ADD SOME DISTURBANCE TO THE STATES
123     % This forces the states to some other position
124     %if ((t > 15) && (t < 17))
125         %xADP(:,k) = [rad2deg(-45) 0 0 0]';
126         %xLQR(:,k) = [rad2deg(-45) 0 0 0]';
127     %end
128     % Force the pitch angle to be in the range [-pi/2,pi/2] because of
129     % physical constraints
130     xADP(theta,k) = angleLimiterPitch(xADP(theta,k));
131     xLQR(theta,k) = angleLimiterPitch(xLQR(theta,k));
132     % Force the yaw angle to be in the range [-pi,pi] because the yaw is
133     % free to do a complete circle
134     xADP(psi,k) = angleLimiterYaw(xADP(psi,k));
135     xLQR(psi,k) = angleLimiterYaw(xLQR(psi,k));
136

```

```

137 % FIND THE ERROR BETWEEN THE EXACT STATES AND THE DESIRED STATES
138 errorADP(:,k) = xd(:,k) - xADP(:,k);
139 errorLQR(:,k) = xd(:,k) - xLQR(:,k);
140
141 % UPDATE THE CRITIC WEIGHTS EVERY T TIME
142 % Determine if the time is a multiple of T
143 if (mod(t,T) == 0)
144     % Update the critic weights
145     wc = quanserAEROCriticTuning(A, B, errorADP(:,(k-nbar):k),tau,
R_Mat_ADP,Q_Mat_ADP,xd(:,k),wcInit);
146     % Use the weights to determine the P matrix
147     P_Mat = [wc(5) wc(6) wc(7) wc(8);
148             wc(6) wc(9) wc(10) wc(11);
149             wc(7) wc(10) wc(12) wc(13);
150             wc(8) wc(11) wc(13) wc(14)];
151     % Use P to determine the state-feedback gain
152     kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
153 end
154 % Update the inputs using ADP
155 uNewADP = kADP*errorADP(:,k);
156 % Limit the voltages
157 uNewADP(1) = sign(uNewADP(1))*min(abs(uNewADP(1)),maxVolt);
158 uNewADP(2) = sign(uNewADP(2))*min(abs(uNewADP(2)),maxVolt);
159 % Update the input matrices
160 uADP(1,k) = uNewADP(1);
161 uADP(2,k) = uNewADP(2);
162 % Update the inputs using LQR
163 uNewLQR = kLQR*errorLQR(:,k);
164 % Limit the voltages
165 uNewLQR(1) = sign(uNewLQR(1))*min(abs(uNewLQR(1)),maxVolt);
166 uNewLQR(2) = sign(uNewLQR(2))*min(abs(uNewLQR(2)),maxVolt);
167 % Update the input matrices
168 uLQR(1,k) = uNewLQR(1);
169 uLQR(2,k) = uNewLQR(2);
170 end
171
172 % SAVE OUTPUTS FOR PLOTTING
173 % Save the time, error, input, states and desired states vectors
174 save([testName, '.mat'], 'dt', 'errorADP', 'uADP', 'xADP', 'xd', 'tau', 'R_Mat_ADP',
'Q_Mat_ADP', 'errorLQR', 'uLQR', 'xLQR', 'R_Mat_LQR', 'Q_Mat_LQR');
175 % Notify that the simulation is complete
176 disp('SIMULATION COMPLETE');
177
178 % ANGLE LIMITER FOR YAW FUNCTION
179 function angle = angleLimiterYaw(angle)
180 % This function limits the angle between -pi and pi
181     angle = mod(angle, 2*pi);
182
183     i=find(angle>pi);
184     angle(i)=angle(i)-2*pi;
185
186     i=find(angle<-pi);
187     angle(i)=angle(i)+2*pi;
188 end

```

```

189
190 % ANGLE LIMITER FOR PITCH FUNCTION
191 function angle = angleLimiterPitch(angle)
192 % This function limits the physical constraint of the pitch measurement to
193 % 90 degrees
194     if (angle < 0)
195         angle = max(angle,-pi/2);
196     end
197
198     if (angle > 0)
199         angle = min(angle,pi/2);
200     end
201 end
202
203 % CRITIC WEIGHT TUNING NEURAL NETWORK
204 % This function is specific to the helicopter because of the number of
205 % weights and error model
206 function weights = quanserAEROCriticTuningInitial(A,B,e_vec,tau,R_Mat,Q_Mat,
    xd)
207     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
208
209     % System dimensions specific for our model
210     [n,~] = size(B);
211
212     % ERROR MODEL OF THE HELICOPTER
213     % fbar and gbar — EQ 8
214     fbar = @(e) A*e;
215     gbar = -B;
216     hbar = -A*xd;
217
218     % DISCRETE-TIME ERROR MODEL FOR TIME TAU
219     % f and g — EQ 10
220     f = @(e) fbar(e)*tau + e;
221     g = gbar*tau;
222     h = hbar*tau;
223
224     % COST FUNCTION PARAMETERS
225     % State penalizing function in the continuous cost function
226     Qbar = @(e) e'*Q_Mat*e;
227     % Control penalizing matrix in the continuous cost function
228     Rbar = R_Mat;
229     % The discrete-time cost function will have terms:
230     % Right after EQ 11 in paper
231     % State penalizing function in the discretized cost function
232     Q = @(e) Qbar(e)*tau;
233     % Control penalizing matrix in the discretized cost function
234     R = Rbar*tau;
235
236     % NEURAL NETWORK FUNCTIONS
237     % Critic neural network activation functions
238     rho = @(e) [e(1); e(2); e(3); e(4);...
239         e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
240         e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
241     % Partial derivative of rho with respect to e

```

```

242     drhode =@(e) [1, 0, 0, 0;
243                 0, 1, 0, 0;
244                 0, 0, 1, 0;
245                 0, 0, 0, 1;
246                 2*e(1), 0, 0, 0;
247                 e(2), e(1), 0, 0;
248                 e(3), 0, e(1), 0;
249                 e(4), 0, 0, e(1);
250                 0, 2*e(2), 0, 0;
251                 0, e(3), e(2), 0;
252                 0, e(4), 0, e(2);
253                 0, 0, 2*e(3), 0;
254                 0, 0, e(4), e(3);
255                 0, 0, 0, 2*e(4)];
256
257     % TOLERANCES
258     % Convergence tolerance for control policy
259     EpsilonPolicy = 0.1;
260     % Convergence tolerance for critic neural network
261     EpsilonWcritic = 0.1;
262
263     % TRAINING PARAMETERS
264     % Number of outer loop iterations
265     outerLoopMax = 700;
266     % Number of inner loop iterations
267     innerLoopMax = 100;
268     % Number of equations needed for training, number of sub-intervals
269     % Number of training samples
270     [n,nbar] = size(e-vec);
271
272     % WEIGHT INITIALIZATION
273     % Initialize the weights of the critic neural network to zero
274     WcLast = zeros(length(rho(e-vec(:,1))),1);
275
276     % LEAST-SQUARES COMPUTATION INITIALIZATION
277     % Matrices required for computing least squares weights of the critic
278     % neural networks — EQ 20
279     V = zeros(nbar,1);
280     Lambda = zeros(nbar,length(rho(e-vec(:,1))));
281
282     % Matrix to hold the derivative of the error model during policy
283     % updating
284     e_k_plus_1 = zeros(n,nbar);
285
286     % Product of the least squares matrices must be invertible
287     % Logic flag indicating if the critic weights are unsolvable
288     % The weights are unsolvable because the least squares matrices have no
289     % solution — not invertible
290     diverged = 0;
291
292     % OUTER LOOP
293     for i = 1:(outerLoopMax-1)
294         % Determine if the least squares matrices are invertible
295         if diverged == 0

```

```

296     % For each of the data collection (discrete time index)
297     for k = 1:nbar
298         % Initialize the optimal inputs to zero
299         uNew = [0; 0];
300         % INNER LOOP
301         for j = 1:(innerLoopMax-1)
302             % Get the updated input value
303             uLast = uNew;
304             % Update the error model
305             e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
306             % Compute the new optimal inputs
307             uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))'*WcLast;
308
309             % Check convergence of the optimal inputs
310             if norm(uNew - uLast) < EpsilonPolicy
311                 break;
312             end
313         end
314
315         % Update the values for the least-squares computation
316         V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
317         Lambda(k,:) = rho(e_vec(:,k))';
318     end
319 end
320
321 % Verify the least square solution exists for the critic's weights
322 % If the error data is consistent or there is no error, this will not
323 % hold, so set the weights to zero
324 if det(Lambda'*Lambda) == 0
325     fprintf('AWESOME...YOU HAVE NO ERROR... I 'M GOING TO SET THE WEIGHTS
TO ZERO\n');
326     weights = zeros(length(rho(e_vec(:,1))),1);
327     break;
328 end;
329
330 % Calculate least squares solution of critic's weights — EQ 20
331 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
332 % Make sure the weights did not diverge
333 % If the weights are diverging, just set them to a large number
334 if isnan(WcNew)
335     fprintf('OOPS...DIVERGING WEIGHTS... I 'M GOING TO USE LARGE WEIGHTS\
n');
336     weights = 1000*ones(length(rho(e_vec(:,1))),1);
337     break;
338 end;
339
340 % Check for convergence of the critic weights
341 if norm(WcNew - WcLast) < EpsilonWcritic
342     weights = WcNew;
343     fprintf('GREAT...THE WEIGHTS CONVERGED\n');
344     break;
345 end
346 % If the weights did not converge, repeat the loop

```

```

347     WcLast = WcNew;
348
349     % If we reached the last iteration of the loop, just use the last
350     % weights found
351     if (i == (outerLoopMax-1))
352         fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP...I 'M GOING
353         TO USE THE LAST VALUE\n');
354         weights = WcNew;
355     end
356 end
357
358 % CRITIC WEIGHT TUNING NEURAL NETWORK
359 % This function is specific to the helicopter because of the number of
360 % weights and error model
361 function weights = quanserAEROCriticTuning(A, B, e_vec, tau, R_Mat, Q_Mat, xd,
362     wcInit)
363     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
364
365     % System dimensions specific for our model
366     [n, ~] = size(B);
367
368     % ERROR MODEL OF THE HELICOPTER
369     % fbar and gbar — EQ 8
370     fbar = @(e) A*e;
371     gbar = -B;
372     hbar = -A*xd;
373
374     % DISCRETE-TIME ERROR MODEL FOR TIME TAU
375     % f and g — EQ 10
376     f = @(e) fbar(e)*tau + e;
377     g = gbar*tau;
378     h = hbar*tau;
379
380     % COST FUNCTION PARAMETERS
381     % State penalizing function in the continuous cost function
382     Qbar = @(e) e'*Q_Mat*e;
383     % Control penalizing matrix in the continuous cost function
384     Rbar = R_Mat;
385     % The discrete-time cost function will have terms:
386     % Right after EQ 11 in paper
387     % State penalizing function in the discretized cost function
388     Q = @(e) Qbar(e)*tau;
389     % Control penalizing matrix in the discretized cost function
390     R = Rbar*tau;
391
392     % NEURAL NETWORK FUNCTIONS
393     % Critic neural network activation functions
394     rho = @(e) [e(1); e(2); e(3); e(4);...
395         e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
396         e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
397     % Partial derivative of rho with respect to e
398     drhode = @(e) [1, 0, 0, 0;
399         0, 1, 0, 0;

```

```

399         0, 0, 1, 0;
400         0, 0, 0, 1;
401         2*e(1), 0, 0, 0;
402         e(2), e(1), 0, 0;
403         e(3), 0, e(1), 0;
404         e(4), 0, 0, e(1);
405         0, 2*e(2), 0, 0;
406         0, e(3), e(2), 0;
407         0, e(4), 0, e(2);
408         0, 0, 2*e(3), 0;
409         0, 0, e(4), e(3);
410         0, 0, 0, 2*e(4)];
411
412 % TOLERANCES
413 % Convergence tolerance for control policy
414 EpsilonPolicy = 0.1;
415 % Convergence tolerance for critic neural network
416 EpsilonWcritic = 0.1;
417
418 % TRAINING PARAMETERS
419 % Number of outer loop iterations
420 outerLoopMax = 700;
421 % Number of inner loop iterations
422 innerLoopMax = 100;
423 % Number of equations needed for training, number of sub-intervals
424 % Number of training samples
425 [n,nbar] = size(e_vec);
426
427 % WEIGHT INITIALIZATION
428 % Initialize the weights of the critic neural network to zero
429 WcLast = zeros(length(rho(e_vec(:,1))),1);
430
431 % LEAST-SQUARES COMPUTATION INITIALIZATION
432 % Matrices required for computing least squares weights of the critic
433 % neural networks — EQ 20
434 V = zeros(nbar,1);
435 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
436
437 % Matrix to hold the derivative of the error model during policy
438 % updating
439 e_k_plus_1 = zeros(n,nbar);
440
441 % Product of the least squares matrices must be invertible
442 % Logic flag indicating if the critic weights are unsolvable
443 % The weights are unsolvable because the least squares matrices have no
444 % solution — not invertible
445 diverged = 0;
446
447 % OUTER LOOP
448 for i = 1:(outerLoopMax-1)
449 % Determine if the least squares matrices are invertible
450 if diverged == 0
451     % For each of the data collection (discrete time index)
452     for k = 1:nbar

```

```

453     % Initialize the optimal inputs to zero
454     uNew = [0; 0];
455     % INNER LOOP
456     for j = 1:(innerLoopMax-1)
457         % Get the updated input value
458         uLast = uNew;
459         % Update the error model
460         e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
461         % Compute the new optimal inputs
462         uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))*WcLast;
463
464         % Check convergence of the optimal inputs
465         if norm(uNew - uLast) < EpsilonPolicy
466             break;
467         end
468     end
469
470     % Update the values for the least-squares computation
471     V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
472     Lambda(k,:) = rho(e_vec(:,k))';
473 end
474 end
475
476 % Verify the least square solution exists for the critic's weights
477 % If the error data is consistent or there is no error, this will not
478 % hold, so set the weights to what they were initially before the
479 % simulation
480 if det(Lambda'*Lambda) == 0
481     fprintf('AWESOME...YOU HAVE NO ERROR... I 'M GOING TO USE THE
ORIGINAL WEIGHTS\n');
482     weights = wcInit;
483     break;
484 end;
485
486 % Calculcate least squares solution of critic's weights — EQ 20
487 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
488 % Make sure the weights did not diverge
489 % If the weights diverged, set the weights to what they were initially
490 % before the simulation
491 if isnan(WcNew)
492     fprintf('OOPS...DIVERGING WEIGHTS... I 'M GOING TO USE THE ORIGINAL
WEIGHTS\n');
493     weights = wcInit;
494     break;
495 end;
496
497 % Check for convergence of the critic weights
498 if norm(WcNew - WcLast) < EpsilonWcritic
499     fprintf('GREAT...THE WEIGHTS CONVERGED\n');
500     weights = WcNew;
501     break;
502 end
503 % If the weights did not converge, do another iteration of the loop

```



```

504     WcLast = WcNew;
505
506     % If we reach the last iteration of the loop, just use the last weights
507     if (i == (outerLoopMax-1))
508         fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP...I'M GOING
509         TO USE THE LAST VALUE\n');
510         weights = WcNew;
511     end
512 end
513
514 function [A, B] = getAeroAB(choice)
515 % This function builds the A and B matrix for either configuration of the
516 % Quanser Aero
517 % choice 0 selects the helicopter mode
518 % choice 1 selects the halfquad mode
519
519 % Quanser Aero Parameters
520 % Moment of Inertia of helicopter body (kg-m^2)
521 L_body = 6.5*0.0254; % length of horizontal body (metal tube)
522 m_body = 0.094; % mass of horizontal body (metal tube)
523 J_body = m_body * L_body^2 / 12; % horizontal cylinder rotating about CM
524
525 % Moment of Inertia of yoke fork that rotates about yaw axis (kg-m^2)
526 m_yoke = 0.526; % mass of entire yoke assembly (kg)
527 h_yoke = 9*0.0254; % height of yoke assembly (m)
528 r_fork = 0.04/2; % radius of each fork (approximated as cylinder)
529 J_yoke = 0.5*m_yoke*r_fork^2;
530
531 % Moment of Inertia from motor + guard assembly about pivot (kg-m^2)
532 m_prop = 0.43; % mass of dc motor + shield + propeller shield
533 m_motor = 0.203; % mass of dc motor
534 r_prop = 6.25*0.0254; % distance from CM to center of pitch axis
535 J_prop = m_prop * r_prop^2; % using parallel axis theorem
536
537 % Equivalent Moment of Inertia about Pitch and Yaw Axis (kg-m^2)
538 Jp = J_body + 2*J_prop; % pitch: body and 2 props
539 Jy = J_body + 2*J_prop + J_yoke; % yaw: body, 2 props, and yoke
540
541 % Thrust-torque constant (N-m/V) [found experimentally]
542 Kpp = 0.0011; % (pre-production unit: 0.0015)
543 Kyy = 0.0022; % (pre-production unit: 0.0040)
544 Kpy = 0.0021; % thrust acting on pitch from yaw (pre-production unit:
545     0.0020)
546 Kyp = -0.0027; % thrust acting on yaw from pitch (pre-production unit:
547     -0.0017)
548
549 % Stiffness (N-m/rad)[found experimentally]
550 Ksp = 0.037463;
551
552 % Viscous damping (N-m-s/rad) [found experimentally]
553 Dp = 0.0071116; % pitch axis (pre-production unit: Dp = 0.0226)
554 Dy = 0.0220; % yaw axis (pre-production unit: Dy = 0.0211)

```

```
554 if choice == 0
555     disp ( 'helicopter mode' );
556     A = [0 0 1 0;
557         0 0 0 1;
558         -Ksp/Jp 0 -Dp/Jp 0;
559         0 0 0 -Dy/Jy];
560     B = [0 0;
561         0 0;
562         Kpp/Jp Kpy/Jp;
563         Kyp/Jy Kyy/Jy];
564
565     disp( 'A - Matrix ' );
566     disp(A);
567     disp( 'B - Matrix ' );
568     disp(B);
569 else
570     disp ( 'halfquad mode' );
571     A = [0 0 1 0;
572         0 0 0 1;
573         -Ksp/Jp 0 -Dp/Jp 0;
574         0 0 0 -Dy/Jy];
575     B = [0 0;
576         0 0;
577         -Kpp/Jy Kpp/Jy;
578         -Kyy/Jy -Kyy/Jy];
579     disp( 'A - Matrix ' );
580     disp(A);
581     disp( 'B - Matrix ' );
582     disp(B);
583 end
584
585 end
```

# Appendix B

## Running V-REP

### B.1 Running a Simulation

#### B.1.1 File Setup

##### Setup the API Files

First, unpack the vrepAeroADP file to a spot on the computer. To use the following three MATLAB files, the version of them must be selected to reflect the 32/64 bit-architecture of the host system.

- remApi.m
- remoteApiProto.m
- remoteApi.dll (optional - MATLAB will compile)

Make sure your MATLAB uses the same bit-architecture as the remoteApi library: 64bit MATLAB with 32bit remoteApi library will not work, and vice-versa! The above files are located in V-REP's installation directory at:

```
../programming/remoteApiBindings/matlab
```

#### The Vortex Physics Engine

The Vortex Physics Engine is a unified real-time simulation platform that V-REP uses to calculate the dynamics of the scene. To obtain Vortex, an account at CM-Labs.com must be made, and a free license acquired.

1. [CM Labs Account and Vortex License](#)
2. [Vortex Download](#)

## B.1.2 Running the Simulation

To run a V-REP scene, controlled by MATLAB:

1. VREP: Select the Vortex physics engine.
2. VREP: Select  $dt = 50$  ms.
3. VREP: Click the clock icon, to the right of the play button. Make sure the output console reads: Toggled to non real-time simulation mode.
4. V-REP: Press the Play button in the V-REP scene.
5. MATLAB: Press the Play Button/Run icon under the **Editor** tab.

## B.2 Skeletal Framework

### B.2.1 MATLAB Framework

```

1
2 % 2-DOF QUANSER HELICOPTER PARAMETERS
3 % Maximum applied voltage for the rotor motors
4 maxVolt = 18;
5 % Number of state variables - theta, psi, dtheta, dpsi
6 n = 4;
7 theta = 1;    % Pitch
8 psi = 2;     % Yaw
9 thetaDot = 3; % Pitch angular velocity
10 psiDot = 4;  % Yaw angular velocity
11 % Number of inputs - Vp, Vy
12 m = 2;
13
14 % SIMULATION TIMING
15 % Initial and final simulation times [s]
16 t0 = 0; tf = 30;
17 % Sampling time [s]
18 tau = 0.05;
19 % Number of time steps
20 tsteps = floor((tf-t0)/tau);
21 % Discrete time vector of sampling time (tau)
22 dt = tau*(0:tsteps);
23
24 state = zeros(n, tsteps+1);
25 u = zeros(m, tsteps+1);
26
27 %% connection routine
28
29 ip = '127.0.0.1';
30 port = 19999;
31

```

```

32 % create a handle to remote API
33 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
34 vrep.simxFinish(-1); % just in case, close all opened connections
35 clientID = vrep.simxStart(ip, port, true, true, 5000, 5);
36
37 %% simulation
38 if (clientID > -1) % if clientID exists
39     disp('Connected to remote API server');
40
41 %Set up the handles
42 [rtn, aero] = vrep.simxGetObjectHandle(clientID, ...
43                                     'Aero', ...
44                                     vrep.simx_opmode_blocking)
45 [rtn, yJ] = vrep.simxGetObjectHandle(clientID, ...
46                                     'yawJoint', ...
47                                     vrep.simx_opmode_blocking)
48 [rtn, pJ] = vrep.simxGetObjectHandle(clientID, ...
49                                     'pitchJoint', ...
50                                     vrep.simx_opmode_blocking)
51
52 % setup joint position streaming
53 vrep.simxGetJointPosition(clientID, ...
54                            yJ, ...
55                            vrep.simx_opmode_streaming);
56 vrep.simxGetJointPosition(clientID, ...
57                            pJ, ...
58                            vrep.simx_opmode_streaming);
59
60 % add a delay to let the streaming operations initialize
61 pause(1);
62
63 [ret, state(1,1)] = vrep.simxGetJointPosition(clientID, ...
64                                               pJ, ...
65                                               vrep.simx_opmode_buffer);
66
67 [ret, state(2,1)] = vrep.simxGetJointPosition(clientID, ...
68                                               yJ, ...
69                                               vrep.simx_opmode_buffer);
70
71
72 state(3,1) = 0;
73 state(4,1) = 0;
74
75 % start a timer
76 tic;
77 % RUN THE SIMULATION
78 % Start at the next time tau after time zero
79 for k = 2:tsteps+1
80     % Current time
81     t = (k-1)*tau; % generate value discrete time index
82
83     % Step 1 – get/store the state of the quadcopter from VREP
84     [ret, state(1,k)] = vrep.simxGetJointPosition(clientID, ...
85                                                    pJ, ...

```

```

86                                     vrep.simx_opmode_buffer
87     );
88     [ret, state(2,k)] = vrep.simxGetJointPosition(clientID, ...
89                                     yJ, ...
90                                     vrep.simx_opmode_buffer
91     );
92
93
94     % algorithm goes here
95
96
97     u(1,k) = 5; % motor 0
98     u(2,k) = 5.01; % motor 1
99
100    % Step 3 – Update Inputs
101    V0V1=[u(1,k) u(2,k)];
102    packedData=vrep.simxPackFloats(V0V1);
103    vrep.simxSetStringSignal(clientID, ...
104                            'Vin_matlab', ...
105                            packedData, ...
106                            vrep.simx_opmode_oneshot);
107
108    pause(tau);
109
110    t = t + tau;
111
112
113    end
114    toc; % end the timer
115
116    ret = 1;
117    while(ret ~= 0)
118        [ret]=vrep.simxClearStringSignal(clientID, ...
119                                        'Vin_matlab', ...
120                                        vrep.simx_opmode_blocking);
121    end
122
123    else
124        disp('Failed connecting to remote API server');
125    end % if (clientID > -1)
126
127    disp('Sim ended');

```

### B.2.2 LUA Framework

```

1 function sysCall_init()
2     -- port to talk to MATLAB
3     simRemoteApi.start(19999)
4
5     -- Assign handles to Propeller Scripts
6     propellerScripts={-1,-1}
7     propellerScripts[1] = sim.getScriptHandle('propeller')

```

```

8     propellerScripts [2] = sim.getScriptHandle('propeller#0')
9
10  end
11
12  function sysCall_actuation()
13      -- Receive velocities from server
14      local packedData = sim.getStringSignal('Vin_matlab')
15
16      if packedData then
17          -- Clear the signal
18          sim.clearStringSignal('Vin_matlab')
19          -- unpack 2 floats , save in array "a={V0, V1}"
20          local a = sim.unpackFloatTable(packedData,0,2,0)
21
22          for i=1,2,1 do
23              -- set script variable Vin associated
24              -- with the propellers
25              sim.setScriptSimulationParameter(propellerScripts[i], 'Vin', a[i])
26          end
27      end
28  end
29
30  function sysCall_sensing()
31
32  end
33
34  function sysCall_cleanup()
35
36  end

```

## B.3 Calculate State-Space

```

1  function [ A, B ] = getAeroAB( choice )
2  %% this function builds the A and B matrix for either configuration of the
3  %% Quanser Aero
4  % choice 0 selects the helicopter mode
5  % choice 1 selects the halfquad mode
6
7  %% Quanser Aero Parameters
8  % Moment of Inertia of helicopter body (kg-m^2)
9  L_body = 6.5*0.0254; % length of horizontal body (metal tube)
10 m_body = 0.094; % mass of horizontal body (metal tube)
11 J_body = m_body * L_body^2 / 12; % horizontal cylinder rotating about CM
12
13 % Moment of Inertia of yoke fork that rotates about yaw axis (kg-m^2)
14 m_yoke = 0.526; % mass of entire yoke assembly (kg)
15 % h_yoke = 9*0.0254; % height of yoke assembly (m)
16 r_fork = 0.04/2; % radius of each fork (approximated as cylinder)
17 J_yoke = 0.5*m_yoke*r_fork^2;
18
19 % Moment of Inertia from motor + guard assembly about pivot (kg-m^2)
20 m_prop = 0.43; % mass of dc motor + shield + propeller shield
21 % m_motor = 0.203; % mass of dc motor

```

```

21 r_prop = 6.25*0.0254; % distance from CM to center of pitch axis
22 J_prop = m_prop * r_prop^2; % using parallel axis theorem
23
24 % Equivalent Moment of Inertia about Pitch and Yaw Axis (kg-m^2)
25 Jp = J_body + 2*J_prop; % pitch: body and 2 props
26 Jy = J_body + 2*J_prop + J_yoke; % yaw: body, 2 props, and yoke
27
28 % Thrust-torque constant (N-m/V) [found experimentally]
29 Kpp = 0.0011; % (pre-production unit: 0.0015)
30 Kyy = 0.0022; % (pre-production unit: 0.0040)
31 Kpy = 0.0021; % thrust acting on pitch from yaw (pre-production unit:
32 0.0020)
33 Kyp = -0.0027; % thrust acting on yaw from pitch (pre-production unit:
34 -0.0017)
35
36 % Stiffness (N-m/rad) [found experimentally]
37 Ksp = 0.037463;
38
39 % Viscous damping (N-m-s/rad) [found experimentally]
40 Dp = 0.0226; % pitch axis (pre-production unit: Dp = 0.0226)
41 Dy = 0.0220; % yaw axis (pre-production unit: Dy = 0.0211)
42
43 if choice == 0
44     disp('helicopter mode');
45     A = [0 0 1 0;
46          0 0 0 1;
47          -Ksp/Jp 0 -Dp/Jp 0;
48          0 0 0 -Dy/Jy];
49     B = [0 0;
50          0 0;
51          Kpp/Jp Kpy/Jp;
52          Kyp/Jy Kyy/Jy];
53
54     disp('A - Matrix');
55     disp(A);
56     disp('B - Matrix');
57     disp(B);
58 else
59     disp('halfquad mode');
60     A = [0 0 1 0;
61          0 0 0 1;
62          -Ksp/Jp 0 -Dp/Jp 0;
63          0 0 0 -Dy/Jy];
64     B = [0 0;
65          0 0;
66          -Kpp/Jy Kpp/Jy;
67          -Kyy/Jy -Kyy/Jy];
68
69     disp('A - Matrix');
70     disp(A);
71     disp('B - Matrix');
72     disp(B);
73 end
74 end

```



```

1 function [ A, B ] = getVREPAeroAB( choice )
2 %% this function builds the A and B matrix for either configuration of the
   Quanser Aero
3 % choice 0 selects the helicopter mode
4 % choice 1 selects the halfquad mode
5
6 %% VREP parameters
7 % Moment of Inertia of pitcharm
8 % M_fuselage = 3.375e-1;
9 % M_sideblock = 4.275e-2;
10 M_fuselage = 0.85;
11 M_sideblock = 0.4;
12 M_motor = 0.225;
13
14 x_fuselage = 3.0000e-2;
15 y_fuselage = 3.75000e-1;
16 z_fuselage = 3.0000e-2;
17
18 x_sideblock = 4.7500e-2;
19 y_sideblock = 3.0000e-2;
20 z_sideblock = 3.0000e-2;
21 r_sideblock = 3.8750e-2;
22
23 r_motor = 1.5875e-1;
24
25 % MoI of yoke
26 side_m = 0.18;
27 cyl_mass = 0.05;
28 cross_m = .1;
29
30 x_side = 5.0000e-2;
31 y_side = 5.0000e-2;
32 z_side = 2.0000e-1;
33 r_side = 8.7500e-2;
34
35 x_cross = 1.2500e-1;
36 y_cross = 5.0000e-2;
37 z_cross = 5.0000e-2;
38
39 dia_cyl = 5.0000e-2;
40 h_cyl = 2.0000e-2;
41
42 J_body = getrectPrismI(y_fuselage , z_fuselage , M_fuselage)
43         + getrectPrismI(y_sideblock , z_sideblock , M_sideblock)
44         + getrectPrismI(y_sideblock , z_sideblock , M_sideblock);
45
46 % Moment of Inertia of yoke fork that rotates about yaw axis (kg-m^2)
47 J_support_on_axis = getrectPrismI(x_side , y_side , side_m);
48 J_support = parAxisTheorem(J_support_on_axis , side_m , r_side);
49 J_cross = getrectPrismI(x_cross , y_cross , cross_m);
50 J_cyl = 0.5*h_cyl*(dia_cyl/2).^2;
51
52 J_yoke = 2*J_support + J_cross + J_cyl;
53

```

```

54 % Moment of Inertia from motor + guard assembly about pivot (kg-m^2)
55 J_prop = M_motor * r_motor^2; % using parallel axis theorem
56
57 % Equivalent Moment of Inertia about Pitch and Yaw Axis (kg-m^2)
58 Jp = J_body + 2*J_prop % pitch: body and 2 props
59 Jy = J_body + 2*J_prop + J_yoke % yaw: body, 2 props, and yoke
60
61 % Thrust-torque constant (N-m/V) [found experimentally]
62 Kpp = 0.03; % (pre-production unit: 0.0015)
63 Kyy = 0.0022; % (pre-production unit: 0.0040)
64 Kpy = 0.0021; % thrust acting on pitch from yaw (pre-production unit:
65 0.0020)
66 Kyp = -0.0027; % thrust acting on yaw from pitch (pre-production unit:
67 -0.0017)
68
69 % Stiffness (N-m/rad) [found experimentally]
70 Ksp = 0.037463;
71
72 % Viscous damping (N-m-s/rad) [found experimentally]
73 Dp = 0.0071116; % pitch axis (pre-production unit: Dp = 0.0226)
74 Dy = 0.0220; % yaw axis (pre-production unit: Dy = 0.0211)
75 %
76 %
77 %
78 %
79 %
80 %
81 %
82 %
83 %
84 %
85 %
86 %
87 %
88 %
89 %
90 %
91 %
92 %
93 %
94 %
95 %
96 %
97 %
98 %
99 %
100 %
101 %
102 %
103 %
104 %
105 %

```

```

54 % Moment of Inertia from motor + guard assembly about pivot (kg-m^2)
55 J_prop = M_motor * r_motor^2; % using parallel axis theorem
56
57 % Equivalent Moment of Inertia about Pitch and Yaw Axis (kg-m^2)
58 Jp = J_body + 2*J_prop % pitch: body and 2 props
59 Jy = J_body + 2*J_prop + J_yoke % yaw: body, 2 props, and yoke
60
61 % Thrust-torque constant (N-m/V) [found experimentally]
62 Kpp = 0.03; % (pre-production unit: 0.0015)
63 Kyy = 0.0022; % (pre-production unit: 0.0040)
64 Kpy = 0.0021; % thrust acting on pitch from yaw (pre-production unit:
65 0.0020)
66 Kyp = -0.0027; % thrust acting on yaw from pitch (pre-production unit:
67 -0.0017)
68
69 % Stiffness (N-m/rad) [found experimentally]
70 Ksp = 0.037463;
71
72 % Viscous damping (N-m-s/rad) [found experimentally]
73 Dp = 0.0071116; % pitch axis (pre-production unit: Dp = 0.0226)
74 Dy = 0.0220; % yaw axis (pre-production unit: Dy = 0.0211)
75 %
76 %
77 %
78 %
79 %
80 %
81 %
82 %
83 %
84 %
85 %
86 %
87 %
88 %
89 %
90 %
91 %
92 %
93 %
94 %
95 %
96 %
97 %
98 %
99 %
100 %
101 %
102 %
103 %
104 %
105 %

```

```

106 if choice == 0
107     disp ('helicopter mode');
108     A = [0 0 1 0;
109         0 0 0 1;
110         -Ksp/Jp 0 -Dp/Jp 0;
111         0 0 0 -Dy/Jy];
112     B = [0 0;
113         0 0;
114         Kpp/Jp Kpy/Jp;
115         Kyp/Jy Kyy/Jy];
116
117     disp ('A - Matrix');
118     disp(A);
119     disp ('B - Matrix');
120     disp(B);
121 else
122     disp ('halfquad mode');
123     A = [0 0 1 0;
124         0 0 0 1;
125         -Ksp/Jp 0 -Dp/Jp 0;
126         0 0 0 -Dy/Jy];
127     B = [0 0;
128         0 0;
129         -Kpp/Jy Kpp/Jy;
130         -Kyy/Jy Kyy/Jy];
131
132     disp ('A - Matrix');
133     disp(A);
134     disp ('B - Matrix');
135     disp(B);

```

```

106     end
107
108 end
109
110 function [I] = getrectPrismI(a,b,m)
111     I = m*(a.^2 + b.^2)/12;
112 end
113
114 function [pax] = parAxisTheorem(I,m,r)
115     pax = I + m*r*r;
116 end

```

## B.4 Joint Control

### B.4.1 LUA 2-DOF Helicopter

```

1  -- Initialization
2  =====
3  if (sim_call_type==sim.syscb_init) then
4      simRemoteApi.start(19999)
5
6      -- Get Handles for Propeller Objects and simulation timestep
7      pJ = sim.getObjectHandle('pitchJoint');
8      yJ = sim.getObjectHandle('yawJoint');
9      prop0 = sim.getObjectHandle('motor0');
10     prop1 = sim.getObjectHandle('motor1');
11     timestep = sim.getSimulationTimeStep()
12     print('timestep is : ', timestep)
13     -- Get parameter values to work with
14     Jy = sim.getScriptSimulationParameter(sim.handle_self, 'Jy')
15     print('Jy is : ', Jy)
16     Jp = sim.getScriptSimulationParameter(sim.handle_self, 'Jp')
17     print('Jp is : ', Jp)
18     Dp = sim.getScriptSimulationParameter(sim.handle_self, 'Dp')
19     print('Dp is : ', Dp)
20     Dy = sim.getScriptSimulationParameter(sim.handle_self, 'Dy')
21     print('Dy is : ', Dy)
22     Kpp = sim.getScriptSimulationParameter(sim.handle_self, 'Kpp')
23     print('Kpp is : ', Kpp)
24     Kpy = sim.getScriptSimulationParameter(sim.handle_self, 'Kpy')
25     print('Kpy is : ', Kpy)
26     Kyp = sim.getScriptSimulationParameter(sim.handle_self, 'Kyp')
27     print('Kyp is : ', Kyp)
28     Kyy = sim.getScriptSimulationParameter(sim.handle_self, 'Kyy')
29     print('Kyy is : ', Kyy)
30     Ksp = sim.getScriptSimulationParameter(sim.handle_self, 'Ksp')
31     print('Ksp is : ', Ksp)
32
33     -- Store current measured angular velocity to calculate angular
34     acceleration
35     prev_pJ_pose = sim.getJointPosition(pJ)
36     prev_yJ_pose = sim.getJointPosition(yJ)

```

```

35
36     sim.setJointTargetVelocity(pJ, 0)
37     sim.setJointTargetVelocity(yJ, 0)
38     sim.setJointTargetVelocity(prop0,0)
39     sim.setJointTargetVelocity(prop1,0)
40     pitchTarVel = 0
41     yawTarVel = 0
42
43     graphHandle=sim.getObjectHandle("Graph")
44
45     PitchDesired = sim.getIntegerSignal("PitchDesired")
46     YawDesired = sim.getIntegerSignal("YawDesired")
47     lastPAcc = 0
48     lastYAcc = 0
49
50     sim.setJointTargetVelocity(prop0,1*0.8)
51     sim.setJointTargetVelocity(prop1,-1*0.8)
52
53 end
54
55 -- Looping code


---


56 if (sim_call_type==sim.syscb_actuation) then
57
58
59     local t = sim.getSimulationTime()
60
61     -- Receive velocities from server
62     local packedData=sim.getStringSignal('MATLAB_SIG')
63     if packedData then
64         sim.clearStringSignal('MATLAB_SIG') -- Clear the signal
65         local V=sim.unpackFloatTable(packedData,0,4,0)
66
67         pJ_pose = sim.getJointPosition(pJ)
68         yJ_pose = sim.getJointPosition(yJ)
69         pJ_vel = (pJ_pose - prev_pJ_pose)/timestep
70         yJ_vel = (yJ_pose - prev_yJ_pose)/timestep
71
72         pAcc = -(Ksp/Jp)*math.sin(pJ_pose) + -(Dy/Jp)*pJ_vel + (Kpp/Jp)*V[1]
73         + (Kpy/Jp)*V[2]
74         yAcc = -(Dp/Jy)*yJ_vel + (Kyp/Jy)*V[1] + (Kyy/Jy)*V[2]
75
76         pitchTarVel = pitchTarVel + ((pAcc+lastPAcc)/2)*timestep
77         yawTarVel = yawTarVel + ((yAcc+lastYAcc)/2)*timestep
78
79         sim.setJointTargetVelocity(pJ, pitchTarVel)
80         sim.setJointTargetVelocity(yJ, yawTarVel)
81
82         prev_pJ_pose = pJ_pose
83         prev_yJ_pose = yJ_pose
84         lastPAcc = pAcc
85         lastYAcc = yAcc

```

```

86     sim.setGraphUserData( graphHandle , 'PitchDesired' ,V[3])
87     sim.setGraphUserData( graphHandle , 'YawDesired' ,V[4])
88
89     sim.setJointTargetVelocity( prop0 ,V[1]*0.8)
90     sim.setJointTargetVelocity( prop1 ,−V[2]*0.8)
91
92     end
93
94 end
95
96 function sysCall_sensing()
97     — put your sensing code here
98 end
99
100 if ( sim_call_type==sim.syscb_cleanup) then
101
102 end

```

### B.4.2 LUA 2-DOF Half-Quadcopter

```

1 — Initialization


---


2 if ( sim_call_type==sim.syscb_init) then
3     simRemoteApi.start(19999)
4
5     pJ = sim.getObjectHandle( 'pitchJoint' );
6     yJ = sim.getObjectHandle( 'yawJoint' );
7     prop0 = sim.getObjectHandle( 'motor0' );
8     prop1 = sim.getObjectHandle( 'motor1' );
9     timestep = sim.getSimulationTimeStep()
10    print( 'timestep is : ', timestep)
11    — Get parameter values to work with
12    Jy = sim.getScriptSimulationParameter( sim.handle_self , 'Jy' )
13    print( 'Jy is : ', Jy)
14    Jp = sim.getScriptSimulationParameter( sim.handle_self , 'Jp' )
15    print( 'Jp is : ', Jp)
16    Dp = sim.getScriptSimulationParameter( sim.handle_self , 'Dp' )
17    print( 'Dp is : ', Dp)
18    Dy = sim.getScriptSimulationParameter( sim.handle_self , 'Dy' )
19    print( 'Dy is : ', Dy)
20    Kpp = sim.getScriptSimulationParameter( sim.handle_self , 'Kpp' )
21    print( 'Kpp is : ', Kpp)
22    Kyy = sim.getScriptSimulationParameter( sim.handle_self , 'Kyy' )
23    print( 'Kyy is : ', Kyy)
24    Ksp = sim.getScriptSimulationParameter( sim.handle_self , 'Ksp' )
25    print( 'Ksp is : ', Ksp)
26
27    — Store current measured angular velocity to calculate angular
28    acceleration
29    prev_pJ_pose = sim.getJointPosition( pJ)
30    prev_yJ_pose = sim.getJointPosition( yJ)
31
32    sim.setJointTargetVelocity( pJ, 0)
33    sim.setJointTargetVelocity( yJ, 0)

```

```

33     sim.setJointTargetVelocity(prop0,0)
34     sim.setJointTargetVelocity(prop1,0)
35     pitchTarVel = 0
36     yawTarVel = 0
37
38     graphHandle=sim.getObjectHandle("Graph")
39
40     PitchDesired = sim.getIntegerSignal("PitchDesired")
41     lastPAcc = 0
42     lastYAcc = 0
43
44     sim.setJointTargetVelocity(prop0,1*0.8)
45     sim.setJointTargetVelocity(prop1,-1*0.8)
46
47 end
48
49 -- Looping code


---


50 if (sim_call_type==sim.syscb_actuation) then
51
52     local t = sim.getSimulationTime()
53     -- Receive velocities from server
54     local packedData=sim.getStringSignal('MATLAB_SIG')
55     if packedData then
56         sim.clearStringSignal('MATLAB_SIG') -- Clear the signal
57         local V=sim.unpackFloatTable(packedData,0,4,0)
58
59         -- Measure pitch and yaw
60         pJ_pose = sim.getJointPosition(pJ)
61         yJ_pose = sim.getJointPosition(yJ)
62
63         -- Calculate Velocites
64         pJ_vel = (pJ_pose - prev_pJ_pose)/timestep
65         yJ_vel = (yJ_pose - prev_yJ_pose)/timestep
66
67         pAcc = -(Ksp/Jy)*math.sin(pJ_pose) + -(Dy/Jy)*pJ_vel + -(Kpp/Jy)*V
[1] + (Kpp/Jy)*V[2]
68         yAcc = -(Dp/Jp)*yJ_vel + -(Kyy/Jy)*V[1] + -(Kyy/Jy)*V[2]
69
70
71         pitchTarVel = pitchTarVel + ((pAcc+lastPAcc)/2)*timestep
72         yawTarVel = yawTarVel + ((yAcc+lastYAcc)/2)*timestep
73
74         sim.setJointTargetVelocity(pJ, pitchTarVel)
75         sim.setJointTargetVelocity(yJ, yawTarVel)
76
77         prev_pJ_pose = pJ_pose
78         prev_yJ_pose = yJ_pose
79         lastPAcc = pAcc
80         lastYAcc = yAcc
81         sim.setGraphUserData(graphHandle, 'PitchDesired',V[3])
82         sim.setGraphUserData(graphHandle, 'YawDesired',V[4])
83

```

```

84     sim.setJointTargetVelocity(prop0,V[1]*0.8)
85     sim.setJointTargetVelocity(prop1,-V[2]*0.8)
86
87     end
88
89 end
90
91 function sysCall_sensing()
92 end
93
94 if (sim_call_type==sim_syscb_cleanup) then
95
96 end

```

### B.4.3 LQR Half-Quadcopter

```

1 % Use exact model of the 2-DOF helicopter and the linearized model to find
2 % error data points
3 % Use the error data points directly in the neural network instead of
4 % randomizing the error
5 close all; clear; clc;
6 % Name of .mat file to save the data
7 testName = 'half_joint_sinesine';
8 [ A, B ] = getAeroAB( 1 );
9
10
11 % 2-DOF QUANSER HELICOPTER PARAMETERS
12 % Maximum applied voltage for the rotor motors
13 maxVolt = 18;
14 % Number of state variables - theta, psi, dtheta, dpsi
15 n = 4;
16 theta = 1;    % Pitch
17 psi = 2;      % Yaw
18 thetaDot = 3; % Pitch angular velocity
19 psiDot = 4;   % Yaw angular velocity
20 % Number of inputs - Vp, Vy
21 m = 2;
22
23 % SIMULATION TIMING
24 % Initial and final simulation times [s]
25 t0 = 0; tf = 30;
26 % Sampling time [s]
27 tau = 0.05;
28 % Larger sampling time for updating the inputs [s]
29 T = 0.2;
30 % Number of time steps
31 tsteps = floor((tf-t0)/tau);
32 % Discrete time vector of sampling time (tau)
33 dt = tau*(0:tsteps);
34
35 % COST FUNCTION
36 % Q and R matrices used in the cost function
37 % Use the matrices used in Quanser documentation except add ones for the
38 % velocities -> need Q to be positive definite

```

```

39 % Modified LQR cost matrices
40 Q_Mat_LQR = diag([250 30 1 1]);
41 R_Mat_LQR = 0.005*diag([1 1]);
42 nonLinearAdjustment = 0.01;
43
44
45 % INITIAL STATES
46 % xInit = [deg2rad(0) deg2rad(0) (0*pi/180)*10 (0*pi/180)*7]';
47 %xInit = [0 0 0 0]';
48
49 % MATRIX INITIALIZATION
50 % Actual states – state variable x
51 %xADP = zeros(n,tsteps+1);
52 xLQR = zeros(n,tsteps+1);
53 pose = zeros(n,tsteps+1);
54 % xLQR(:,1) = xInit;
55 % State errors for entire simulation
56 errorLQR = zeros(n,tsteps+1);
57 % Actual inputs
58 uLQR = zeros(m,tsteps+1);
59
60 % Desired states
61 % Uncomment for desired states of all zero
62 %xd = zeros(n,tsteps+1);
63 % Uncomment for desired states that switch constant values
64 %xd = [deg2rad(37)*ones(1,tsteps+1); deg2rad(80)*ones(1,tsteps+1); zeros(1,
    tsteps+1); zeros(1,tsteps+1)];
65 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
    (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
    ; zeros(1,tsteps+1)];
66 % Uncomment for desired states that are sine and cosine waves
67 pitchSine = deg2rad(30)*sin(0.5*dt);
68 yawCosine = deg2rad(60)*cos(0.5*dt);
69 % pitchSquare = deg2rad(25)*square(0.5*dt);
70 % yawSquare = deg2rad(-36)*square(0.25*dt);
71 % xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
72 % xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
73
74
75
76 %% CALCULATE THE CONTROLLER GAIN MATRIX USING LQR
77 fprintf('WORKING ON LQR GAIN MATRIX...\n');
78 kLQR = lqr(A,B,Q_Mat_LQR,R_Mat_LQR);
79
80 pause(1);
81 %% connection routine
82
83 ip = '127.0.0.1';
84 port = 19999;
85
86 % create a handle to remote API
87 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
88 vrep.simxFinish(-1); % just in case, close all opened connections
89 clientID = vrep.simxStart(ip,port,true,true,5000,5);

```



```

90
91 %% simulation
92 if (clientID > -1) % if clientID exists
93     disp('Connected to remote API server');
94
95 %Set up the handles
96 [rtn , aero] = vrep.simxGetObjectHandle(clientID ,...
97                                         'Aero' ,...
98                                         vrep.simx_opmode_blocking)
99 [rtn , yJ] = vrep.simxGetObjectHandle(clientID ,...
100                                        'yawJoint' ,...
101                                        vrep.simx_opmode_blocking)
102 [rtn , pJ] = vrep.simxGetObjectHandle(clientID ,...
103                                        'pitchJoint' ,...
104                                        vrep.simx_opmode_blocking)
105
106 % setup joint position streaming
107 vrep.simxGetJointPosition(clientID ,...
108                            yJ ,...
109                            vrep.simx_opmode_streaming);
110 vrep.simxGetJointPosition(clientID ,...
111                            pJ ,...
112                            vrep.simx_opmode_streaming);
113
114 % add a delay to let the streaming operations initialize
115 pause(1);
116
117 [ret , pose(1,1)] = vrep.simxGetJointPosition(clientID ,...
118                                                pJ ,...
119                                                vrep.simx_opmode_buffer);
120
121 [ret , pose(2,1)] = vrep.simxGetJointPosition(clientID ,...
122                                                yJ ,...
123                                                vrep.simx_opmode_buffer);
124
125
126 pose(3,1) = 0;
127 pose(4,1) = 0;
128 xLQR(:,1) = pose(:,1);
129 % Initialize the error
130 errorLQR(:,1) = xd(:,1) - xLQR(:,1);
131
132
133 tic;
134 % RUN THE SIMULATION
135 % Start at the next time tau after time zero
136 for k = 2:tsteps+1
137     % Current time
138     t = (k-1)*tau; % generate value discrete time index
139
140     % Step 1 - get/store the state of the quadcopter from VREP
141     [ret , pose(1,k)] = vrep.simxGetJointPosition(clientID ,...
142                                                    pJ ,...
143                                                    vrep.simx_opmode_buffer)

```

```

;
144 [ret, pose(2,k)] = vrep.simxGetJointPosition(clientID, ...
145                                           yJ, ...
146                                           vrep.simx_opmode_buffer)
147 ;
148
149 pose(3,k) = (pose(1,k) - pose(1,k-1))\tau;
150 pose(4,k) = (pose(2,k) - pose(2,k-1))\tau;
151
152
153 xLQR(:,k) = pose(:,k);
154
155
156 % EXACT 2-DOF MODEL
157 % Take previous values to find the derivative of the exact model
158 xdotNonLinearLQR = A*xLQR(:,k-1) + B*uLQR(:,k-1);
159 % Update the states of the exact model
160 % Use Euler integration to estimate the current states of the nonlinear
161 % model
162 xLQR(:,k) = xLQR(:,k-1) + xdotNonLinearLQR*tau; % + nonLinearAdjustment
    .*(2.*rand(4,1)-1);
163
164 % xLQR(theta,k) = angleLimiterPitch(xLQR(theta,k));
165 % Force the yaw angle to be in the range [-pi,pi]
166 xLQR(psi,k) = angleLimiterYaw(xLQR(psi,k));
167
168 % FIND THE ERROR BETWEEN THE EXACT STATES AND THE DESIRED STATES
169 errorLQR(:,k) = xd(:,k) - xLQR(:,k);
170
171 % Update the inputs using LQR
172 uNewLQR = kLQR*errorLQR(:,k);
173 % Limit the voltages
174 uNewLQR(1) = sign(uNewLQR(1))*min(abs(uNewLQR(1)),maxVolt);
175 uNewLQR(2) = sign(uNewLQR(2))*min(abs(uNewLQR(2)),maxVolt);
176 % Update the input matrices
177 uLQR(1,k) = uNewLQR(1);
178 uLQR(2,k) = uNewLQR(2);
179
180 % Step 3 - Update Inputs
181 VREP_PACKET=[uNewLQR(1) uNewLQR(2) rad2deg(xd(1,k)) rad2deg(xd(2,k))];
182 % motorSpeeds=[20 0];
183 packedData=vrep.simxPackFloats(VREP_PACKET);
184 vrep.simxSetStringSignal(clientID, ...
185                           'MATLAB_SIG', ...
186                           packedData, ...
187                           vrep.simx_opmode_oneshot);
188
189 pause(tau);
190
191 t = t + tau;
192
193
194 end

```

```

195 toc;
196
197 %ret = 1;
198 while (ret ~= 0)
199     [ret]=vrep.simxClearStringSignal(clientID ,...
200                                     'MATLAB_SIG' ,...
201                                     vrep.simx_opmode_blocking);
202 end
203
204 else
205     disp('Failed connecting to remote API server');
206 end % if (clientID > -1)
207
208 disp('Sim ended');
209
210 figure
211 plot(dt,rad2deg(pose(1,:))); hold on;
212 plot(dt,rad2deg(xd(1,:))); hold off;
213 title('pitch')
214
215 figure
216 plot(dt,rad2deg(pose(2,:))); hold on;
217 plot(dt,rad2deg(xd(2,:))); hold off;
218 title('yaw')
219
220 figure
221 plot(dt,uLQR(1,:)); hold on;
222 plot(dt,uLQR(2,:)); hold off;
223 title('Inputs')
224
225 % SAVE OUTPUTS FOR PLOTTING
226 % Save the time, error, input, states and desired states vectors
227 save([testName, '.mat'], 'dt', 'xd', 'tau', 'errorLQR', 'uLQR', 'xLQR', 'R_Mat_LQR',
228     , 'Q_Mat_LQR');
229 % Notify that the simulation is complete
230 disp('SIMULATION COMPLETE');
231
232 % ANGLE LIMITER FOR YAW FUNCTION
233 function angle = angleLimiterYaw(angle)
234 % This function limits the angle between -pi and pi
235     angle = mod(angle, 2*pi);
236
237     i=find(angle>pi);
238     angle(i)=angle(i)-2*pi;
239
240     i=find(angle<-pi);
241     angle(i)=angle(i)+2*pi;
242 end

```

### B.4.4 LQR 2-DOF Helicopter

```

1 % Use exact model of the 2-DOF helicopter and the linearized model to find
2 % error data points
3 % Use the error data points directly in the neural network instead of

```

```

4 % randomizing the error
5 close all; clear; clc;
6
7 % Name of .mat file to save the data
8 testName = 'heli_joint_sqsq';
9
10 % LINEAR MODEL MATRIX PARAMETERS (2DoF Helicopter)
11 [ A, B ] = getAeroAB( 0 );
12
13 % 2-DOF QUANSER HELICOPTER PARAMETERS
14 % Maximum applied voltage for the rotor motors
15 maxVolt = 18;
16 % Number of state variables – theta, psi, dtheta, dpsi
17 n = 4;
18 theta = 1;    % Pitch
19 psi = 2;      % Yaw
20 thetaDot = 3; % Pitch angular velocity
21 psiDot = 4;   % Yaw angular velocity
22 % Number of inputs – Vp, Vy
23 m = 2;
24
25 % SIMULATION TIMING
26 % Initial and final simulation times [s]
27 t0 = 0; tf = 30;
28 % Sampling time [s]
29 tau = 0.05;
30 % Larger sampling time for updating the inputs [s]
31 T = 0.2;
32 % Number of time steps
33 tsteps = floor((tf-t0)/tau);
34 % Discrete time vector of sampling time (tau)
35 dt = tau*(0:tsteps);
36
37 % COST FUNCTION
38 % Q and R matrices used in the cost function
39 % Use the matrices used in Quanser documentation except add ones for the
40 % velocities -> need Q to be positive definite
41 % Modified LQR cost matrices
42 Q_Mat_LQR = diag([250 30 1 1]);
43 R_Mat_LQR = 0.005*diag([1 1]);
44 nonLinearAdjustment = 0.01;
45
46 % INITIAL STATES
47 % xInit = [deg2rad(0) deg2rad(0) (0*pi/180)*10 (0*pi/180)*7]';
48 %xInit = [0 0 0 0]';
49
50 % MATRIX INITIALIZATION
51 % Actual states – state variable x
52 xLQR = zeros(n, tsteps+1);
53 pose = zeros(n, tsteps+1);
54
55 % State errors for entire simulation
56 errorLQR = zeros(n, tsteps+1);
57 % Actual inputs

```

```

58 uLQR = zeros(m,tsteps+1);
59
60 % Desired states
61 % Uncomment for desired states of all zero
62 %xd = zeros(n,tsteps+1);
63 % Uncomment for desired states that switch constant values
64 %xd = [deg2rad(37)*ones(1,tsteps+1); deg2rad(80)*ones(1,tsteps+1); zeros(1,
        tsteps+1); zeros(1,tsteps+1)];
65 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
        (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
        ; zeros(1,tsteps+1)];
66 % Uncomment for desired states that are sine and cosine waves
67 % pitchSine = deg2rad(30)*sin(0.5*dt);
68 % yawCosine = deg2rad(60)*cos(0.5*dt);
69 pitchSquare = deg2rad(25)*square(0.5*dt);
70 yawSquare = deg2rad(-36)*square(0.25*dt);
71 %xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
72 xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
73
74
75 %% CALCULATE THE CONTROLLER GAIN MATRIX USING LQR
76 fprintf('WORKING ON LQR GAIN MATRIX...\n');
77 kLQR = lqr(A,B,Q_Mat_LQR,R_Mat_LQR);
78
79 %% connection routine
80 ip = '127.0.0.1';
81 port = 19999;
82
83 % create a handle to remote API
84 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
85 vrep.simxFinish(-1); % just in case, close all opened connections
86 clientID = vrep.simxStart(ip,port,true,true,5000,5);
87
88 %% simulation
89 if (clientID>-1) % if clientID exists
90     disp('Connected to remote API server');
91
92 %Set up the handles
93 [rtn, aero] = vrep.simxGetObjectHandle(clientID,...
94                                     'Aero',...
95                                     vrep.simx_opmode_blocking)
96 [rtn, yJ] = vrep.simxGetObjectHandle(clientID,...
97                                     'yawJoint',...
98                                     vrep.simx_opmode_blocking)
99 [rtn, pJ] = vrep.simxGetObjectHandle(clientID,...
100                                    'pitchJoint',...
101                                    vrep.simx_opmode_blocking)
102
103 % setup joint position streaming
104 vrep.simxGetJointPosition(clientID,...
105                            yJ,...
106                            vrep.simx_opmode_streaming);
107 vrep.simxGetJointPosition(clientID,...
108                            pJ,...

```

```

109         vrep.simx_opmode_streaming);
110
111 % add a delay to let the streaming operations initialize
112 pause(1);
113
114 [ret , pose(1,1)] = vrep.simxGetJointPosition(clientID ,...
115                                             pJ ,...
116                                             vrep.simx_opmode_buffer);
117
118 [ret , pose(2,1)] = vrep.simxGetJointPosition(clientID ,...
119                                             yJ ,...
120                                             vrep.simx_opmode_buffer);
121
122
123 pose(3,1) = 0;
124 pose(4,1) = 0;
125 xLQR(:,1) = pose(:,1);
126 % Initialize the error
127 errorLQR(:,1) = xd(:,1) - xLQR(:,1);
128
129
130 tic;
131 % RUN THE SIMULATION
132 % Start at the next time tau after time zero
133 for k = 2:tsteps+1
134     % Current time
135     t = (k-1)*tau; % generate value discrete time index
136
137     % Step 1 – get/store the state of the quadcopter from VREP
138     [ret , pose(1,k)] = vrep.simxGetJointPosition(clientID ,...
139                                                  pJ ,...
140                                                  vrep.simx_opmode_buffer)
141     ;
142     [ret , pose(2,k)] = vrep.simxGetJointPosition(clientID ,...
143                                                  yJ ,...
144                                                  vrep.simx_opmode_buffer)
145     ;
146     pose(3,k) = (pose(1,k) - pose(1,k-1))\tau;
147     pose(4,k) = (pose(2,k) - pose(2,k-1))\tau;
148
149
150     xLQR(:,k) = pose(:,k);
151
152
153 % EXACT 2-DOF MODEL
154 % Take previous values to find the derivative of the exact model
155 xdotNonLinearLQR = A*xLQR(:,k-1) + B*uLQR(:,k-1);
156 % Update the states of the exact model
157 % Use Euler integration to estimate the current states of the nonlinear
158 % model
159 xLQR(:,k) = xLQR(:,k-1) + xdotNonLinearLQR*tau; % + nonLinearAdjustment
    .* (2.*rand(4,1)-1);

```

```

160
161 %     xLQR(theta,k) = angleLimiterPitch(xLQR(theta,k));
162 % Force the yaw angle to be in the range [-pi,pi]
163     xLQR(psi,k) = angleLimiterYaw(xLQR(psi,k));
164
165 % FIND THE ERROR BETWEEN THE EXACT STATES AND THE DESIRED STATES
166 errorLQR(:,k) = xd(:,k) - xLQR(:,k);
167
168 % Update the inputs using LQR
169 uNewLQR = kLQR*errorLQR(:,k);
170 % Limit the voltages
171 uNewLQR(1) = sign(uNewLQR(1))*min(abs(uNewLQR(1)),maxVolt);
172 uNewLQR(2) = sign(uNewLQR(2))*min(abs(uNewLQR(2)),maxVolt);
173 % Update the input matrices
174 uLQR(1,k) = uNewLQR(1);
175 uLQR(2,k) = uNewLQR(2);
176
177 % Step 3 - Update Inputs
178 Vin=[uNewLQR(1) uNewLQR(2) rad2deg(xd(1,k)) rad2deg(xd(2,k))];
179 %   motorSpeeds=[20 0];
180 packedData=vrep.simxPackFloats(Vin);
181 vrep.simxSetStringSignal(clientID,...
182                         'MATLAB_SIG',...
183                         packedData,...
184                         vrep.simx_opmode_oneshot);
185
186 pause(tau);
187
188 t = t + tau;
189
190
191 end
192 toc;
193
194 %ret = 1;
195 while(ret ~= 0)
196     [ret]=vrep.simxClearStringSignal(clientID,...
197                                     'MATLAB_SIG',...
198                                     vrep.simx_opmode_blocking);
199 end
200
201 else
202     disp('Failed connecting to remote API server');
203 end % if (clientID > -1)
204
205 disp('Sim ended');
206
207 figure
208 plot(dt,rad2deg(pose(1,:))); hold on;
209 plot(dt,rad2deg(xd(1,:))); hold off;
210 title('pitch')
211
212 figure
213 plot(dt,rad2deg(pose(2,:))); hold on;

```

```

214 plot(dt, rad2deg(xd(2,:))); hold off;
215 title('yaw')
216
217 figure
218 plot(dt, uLQR(1,:)); hold on;
219 plot(dt, uLQR(2,:)); hold off;
220 title('Inputs')
221
222 % SAVE OUTPUTS FOR PLOTTING
223 % Save the time, error, input, states and desired states vectors
224 save(['testName', '.mat'], 'dt', 'xd', 'tau', 'errorLQR', 'uLQR', 'xLQR', 'R_Mat_LQR',
      'Q_Mat_LQR');
225 % Notify that the simulation is complete
226 disp('SIMULATION COMPLETE');
227
228 % ANGLE LIMITER FOR YAW FUNCTION
229 function angle = angleLimiterYaw(angle)
230 % This function limits the angle between -pi and pi
231     angle = mod(angle, 2*pi);
232
233     i=find(angle>pi);
234     angle(i)=angle(i)-2*pi;
235
236     i=find(angle<-pi);
237     angle(i)=angle(i)+2*pi;
238 end

```

### B.4.5 ADP Half-Quadcopter

```

1 % Use exact model of the 2-DOF helicopter and the linearized model to find
2 % error data points
3 % Use the error data points directly in the neural network instead of
4 % randomizing the error
5 close all; clear; clc;
6 testName = 'half_joint_adp_sinesine';
7
8 % LINEAR MODEL MATRIX PARAMETERS
9 [ A, B ] = getAeroAB( 1 )
10 % 2-DOF QUANSER HELICOPTER PARAMETERS
11 % Maximum applied voltage for the rotor motors
12 maxVolt = 18;
13 % Number of state variables - theta, psi, dtheta, dpsi
14 n = 4;
15 theta = 1; % Pitch
16 psi = 2; % Yaw
17 thetaDot = 3; % Pitch angular velocity
18 psiDot = 4; % Yaw angular velocity
19 % Number of inputs - Vp, Vy
20 m = 2;
21
22 % SIMULATION TIMING
23 % Initial and final simulation times [s]
24 t0 = 0; tf = 25;
25 % Sampling time [s]

```



```

26 tau = 0.05;
27 % Larger sampling time for updating the inputs [s]
28 T = 0.75;
29 % Number of time steps
30 tsteps = floor((tf-t0)/tau);
31 % Discrete time vecotor of sampling time (tau)
32 dt = tau*(0:tsteps);
33 % Number of equations for actor-critic neural network
34 % Number of training samples per T
35 nbar = floor(T/tau);
36
37 % COST FUNCTION
38 % Q and R matrices used in the cost function
39 %Q_Mat_ADP = diag([270 100 1 1]);
40 %R_Mat_ADP = 0.0001*diag([1 1]);
41 % Use the matrices used in Quanser documentation except add ones for the
42 % velocities -> need Q to be positive definite
43 Q_Mat_ADP = diag([200 75 1 1]);
44 R_Mat_ADP = 0.005*diag([1 1]);
45 % Modified LQR cost matrices
46 Q_Mat_LQR = diag([200 75 1 1]);
47 R_Mat_LQR = 0.005*diag([1 1]);
48 nonLinearAdjustment = 0.03;
49 % Name of .mat file to save the data
50
51
52
53
54 % MATRIX INITIALIZATION
55 % Actual states - state variable x
56 xADP = zeros(n,tsteps+1);
57 % State errors for entire simulation
58 errorADP = zeros(n,tsteps+1);
59 % Actual inputs
60 uADP = zeros(m,tsteps+1);
61
62 % Desired states
63 % Uncomment for desired states of all zero
64 %xd = zeros(n,tsteps+1);
65 % Uncomment for desired states that switch constant values
66 %xd = [deg2rad(37)*ones(1,tsteps+1); deg2rad(80)*ones(1,tsteps+1); zeros(1,
        tsteps+1); zeros(1,tsteps+1)];
67 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
        (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
        ; zeros(1,tsteps+1)];
68 % Uncomment for desired states that are sine and cosine waves
69 pitchSine = deg2rad(30)*sin(0.5*dt);
70 yawCosine = deg2rad(90)*cos(0.5*dt);
71 % pitchSquare = deg2rad(57)*square(0.5*dt);
72 % yawSquare = deg2rad(-36)*square(0.25*dt);
73 xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
74 %xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
75
76

```

```

77
78 % FIND THE CRITIC NEURAL NETWORK WEIGHTS BEFORE APPLYING ANY INPUTS
79 % Calculate the critic weights
80 wcInit = quanserAEROCriticTuningInitial(A, B, ((2*pi).*rand(4,nbar)-pi),tau,
      R_Mat_ADP,Q_Mat_ADP,xd(:,1));
81 % Use the weights to determine the P matrix
82 P_Mat = [wcInit(5) wcInit(6) wcInit(7) wcInit(8);
83          wcInit(6) wcInit(9) wcInit(10) wcInit(11);
84          wcInit(7) wcInit(10) wcInit(12) wcInit(13);
85          wcInit(8) wcInit(11) wcInit(13) wcInit(14)];
86 % Use P to determine the state-feedback gain
87 kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
88
89
90
91
92
93 %% connection routine
94
95 ip = '127.0.0.1';
96 port = 19999;
97
98 % create a handle to remote API
99 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
100 vrep.simxFinish(-1); % just in case, close all opened connections
101 clientID = vrep.simxStart(ip,port,true,true,5000,5);
102
103 %% simulation
104 if (clientID>-1) % if clientID exists
105     disp('Connected to remote API server');
106
107 %Set up the handles
108 [rtn, aero] = vrep.simxGetObjectHandle(clientID,...
109                                     'Aero',...
110                                     vrep.simx_opmode_blocking)
111 [rtn, yJ] = vrep.simxGetObjectHandle(clientID,...
112                                     'yawJoint',...
113                                     vrep.simx_opmode_blocking)
114 [rtn, pJ] = vrep.simxGetObjectHandle(clientID,...
115                                     'pitchJoint',...
116                                     vrep.simx_opmode_blocking)
117
118 % setup joint position streaming
119 vrep.simxGetJointPosition(clientID,...
120                           yJ,...
121                           vrep.simx_opmode_streaming);
122 vrep.simxGetJointPosition(clientID,...
123                           pJ,...
124                           vrep.simx_opmode_streaming);
125
126 % add a delay to let the streaming operations initialize
127 pause(1);
128
129 [ret, pose(1,1)] = vrep.simxGetJointPosition(clientID,...

```

```

130                                     pJ, ...
131                                     vrep.simx_opmode_buffer);
132
133 [ret, pose(2,1)] = vrep.simxGetJointPosition(clientID, ...
134                                     yJ, ...
135                                     vrep.simx_opmode_buffer);
136
137
138 pose(3,1) = 0;
139 pose(4,1) = 0;
140 xADP(:,1) = pose(:,1);
141 % Initialize the error
142 errorADP(:,1) = xd(:,1) - xADP(:,1);
143
144
145
146 % RUN THE SIMULATION
147 % Start at the next time tau after time zero
148
149
150 % RUN THE SIMULATION
151 % Start at the next time tau after time zero
152 tic;
153 for k = 2:tsteps+1
154     % Current time
155     t = (k-1)*tau; % generate value discrete time index
156
157     % Step 1 - get/store the state of the quadcopter from VREP
158     [ret, pose(1,k)] = vrep.simxGetJointPosition(clientID, ...
159                                         pJ, ...
160                                         vrep.simx_opmode_buffer)
161
162     ;
163
164     [ret, pose(2,k)] = vrep.simxGetJointPosition(clientID, ...
165                                         yJ, ...
166                                         vrep.simx_opmode_buffer)
167
168     ;
169
170     pose(3,k) = (pose(1,k) - pose(1,k-1))/tau;
171     pose(4,k) = (pose(2,k) - pose(2,k-1))/tau;
172
173
174     xADP(:,k) = pose(:,k);
175     % EXACT 2-DOF HELICOPTER MODEL
176     % Take previous values to find the derivative of the exact model
177     xdotNonLinearADP = A*xADP(:,k-1) + B*uADP(:,k-1);
178     % Update the states of the exact model
179     % Use Euler integration to estimate the current states of the nonlinear
180     % model
181     xADP(:,k) = xADP(:,k-1) + xdotNonLinearADP*tau + nonLinearAdjustment
182     .* (2.*rand(4,1)-1);
183     % ADD SOME DISTURBANCE TO THE STATES
184     % This forces the states to some other position
185     %if ((t > 15) && (t < 17))

```

```

181     %xADP(:,k) = [rad2deg(-45) 0 0 0]';
182     %xLQR(:,k) = [rad2deg(-45) 0 0 0]';
183 %end
184 % Force the pitch angle to be in the range [-pi/2,pi/2] because of
185 % physical constraints
186 xADP(theta,k) = angleLimiterPitch(xADP(theta,k));
187 % Force the yaw angle to be in the range [-pi,pi] because the yaw is
188 % free to do a complete circle
189 xADP(psi,k) = angleLimiterYaw(xADP(psi,k));
190
191 % FIND THE ERROR BETWEEN THE EXACT STATES AND THE DESIRED STATES
192 errorADP(:,k) = xd(:,k) - xADP(:,k);
193
194 % UPDATE THE CRITIC WEIGHTS EVERY T TIME
195 % Determine if the time is a multiple of T
196 if (mod(t,T) == 0)
197     % Update the critic weights
198     wc = quanserAEROCriticTuning(A, B, errorADP(:,(k-nbar):k),tau,
199 R_Mat_ADP,Q_Mat_ADP,xd(:,k),wcInit);
200     % Use the weights to determine the P matrix
201     P_Mat = [wc(5) wc(6) wc(7) wc(8);
202             wc(6) wc(9) wc(10) wc(11);
203             wc(7) wc(10) wc(12) wc(13);
204             wc(8) wc(11) wc(13) wc(14)];
205     % Use P to determine the state-feedback gain
206     kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
207 end
208 % Update the inputs using ADP
209 uNewADP = kADP*errorADP(:,k);
210 % Limit the voltages
211 uNewADP(1) = sign(uNewADP(1))*min(abs(uNewADP(1)),maxVolt);
212 uNewADP(2) = sign(uNewADP(2))*min(abs(uNewADP(2)),maxVolt);
213 % Update the input matrices
214 uADP(1,k) = uNewADP(1);
215 uADP(2,k) = uNewADP(2);
216
217 Vin=[uNewADP(1) uNewADP(2) rad2deg(xd(1,k)) rad2deg(xd(2,k))];
218 packedData=vrep.simxPackFloats(Vin);
219 vrep.simxSetStringSignal(clientID,...
220                          'MATLAB_SIG',...
221                          packedData,...
222                          vrep.simx_opmode_oneshot);
223
224 pause(tau);
225
226 t = t + tau;
227
228 end
229 toc;
230
231 %ret = 1;
232 while (ret ~= 0)
233     [ret]=vrep.simxClearStringSignal(clientID,...

```

```

234         'MATLAB_SIG' , ...
235         vrep.simx_opmode_blocking);
236 end
237
238 else
239     disp('Failed connecting to remote API server');
240 end % if (clientID > -1)
241
242 disp('Sim ended');
243
244
245 figure
246 plot(dt, rad2deg(pose(1, :))); hold on;
247 plot(dt, rad2deg(xd(1, :))); hold off;
248 title('pitch')
249
250 figure
251 plot(dt, rad2deg(pose(2, :))); hold on;
252 plot(dt, rad2deg(xd(2, :))); hold off;
253 title('yaw')
254
255 figure
256 plot(dt, uADP(1, :)); hold on;
257 plot(dt, uADP(2, :)); hold off;
258 title('Inputs')
259
260 % SAVE OUTPUTS FOR PLOTTING
261 % Save the time, error, input, states and desired states vectors
262 save([testName, '.mat'], 'dt', 'errorADP', 'uADP', 'xADP', 'xd', 'tau', 'R_Mat_ADP',
    'Q_Mat_ADP');
263 % Notify that the simulation is complete
264 disp('SIMULATION COMPLETE');
265
266 % ANGLE LIMITER FOR YAW FUNCTION
267 function angle = angleLimiterYaw(angle)
268 % This function limits the angle between -pi and pi
269     angle = mod(angle, 2*pi);
270
271     i=find(angle>pi);
272     angle(i)=angle(i)-2*pi;
273
274     i=find(angle<-pi);
275     angle(i)=angle(i)+2*pi;
276 end
277
278 % ANGLE LIMITER FOR PITCH FUNCTION
279 function angle = angleLimiterPitch(angle)
280 % This function limits the physical constraint of the pitch measurement to
281 % 90 degrees
282     if (angle < 0)
283         angle = max(angle, -pi/2);
284     end
285
286     if (angle > 0)

```

```

287     angle = min(angle, pi/2);
288     end
289 end
290
291 % CRITIC WEIGHT TUNING NEURAL NETWORK
292 % This function is specific to the helicopter because of the number of
293 % weights and error model
294 function weights = quanserAEROCriticTuningInitial(A, B, e_vec, tau, R_Mat,
    Q_Mat, xd)
295     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
296
297     % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
298     % A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
299     % B = [0 0; 0 0; 0.0512 0.0977; -0.1139 0.0928];
300     % System dimensions specific for our model
301     [n, ~] = size(B);
302
303     % ERROR MODEL OF THE HELICOPTER
304     % fbar and gbar — EQ 8
305     fbar = @(e) A*e;
306     gbar = -B;
307     hbar = -A*xd;
308
309     % DISCRETE-TIME ERROR MODEL FOR TIME TAU
310     % f and g — EQ 10
311     f = @(e) fbar(e)*tau + e;
312     g = gbar*tau;
313     h = hbar*tau;
314
315     % COST FUNCTION PARAMETERS
316     % State penalizing function in the continuous cost function
317     Qbar = @(e) e'*Q_Mat*e;
318     % Control penalizing matrix in the continuous cost function
319     Rbar = R_Mat;
320     % The discrete-time cost function will have terms:
321     % Right after EQ 11 in paper
322     % State penalizing function in the discretized cost function
323     Q = @(e) Qbar(e)*tau;
324     % Control penalizing matrix in the discretized cost function
325     R = Rbar*tau;
326
327     % NEURAL NETWORK FUNCTIONS
328     % Critic neural network activation functions
329     rho = @(e) [e(1); e(2); e(3); e(4); ...
330                e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4); ...
331                e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
332     % Partial derivative of rho with respect to e
333     drhode = @(e) [1, 0, 0, 0;
334                   0, 1, 0, 0;
335                   0, 0, 1, 0;
336                   0, 0, 0, 1;
337                   2*e(1), 0, 0, 0;
338                   e(2), e(1), 0, 0;
339                   e(3), 0, e(1), 0;

```

```

340         e(4), 0, 0, e(1);
341         0, 2*e(2), 0, 0;
342         0, e(3), e(2), 0;
343         0, e(4), 0, e(2);
344         0, 0, 2*e(3), 0;
345         0, 0, e(4), e(3);
346         0, 0, 0, 2*e(4)];
347
348 % TOLERANCES
349 % Convergence tolerance for control policy
350 EpsilonPolicy = 0.1;
351 % Convergence tolerance for critic neural network
352 EpsilonWcritic = 0.1;
353
354 % TRAINING PARAMETERS
355 % Number of outer loop iterations
356 outerLoopMax = 700;
357 % Number of inner loop iterations
358 innerLoopMax = 100;
359 % Number of equations needed for training, number of sub-intervals
360 % Number of training samples
361 [n,nbar] = size(e_vec);
362
363 % WEIGHT INITIALIZATION
364 % Initialize the weights of the critic neural network to zero
365 WcLast = zeros(length(rho(e_vec(:,1))),1);
366
367 % LEAST-SQUARES COMPUTATION INITIALIZATION
368 % Matrices required for computing least squares weights of the critic
369 % neural networks — EQ 20
370 V = zeros(nbar,1);
371 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
372
373 % Matrix to hold the derivative of the error model during policy
374 % updating
375 e_k_plus_1 = zeros(n,nbar);
376
377 % Product of the least squares matrices must be invertible
378 % Logic flag indicating if the critic weights are unsolvable
379 % The weights are unsolvable because the least squares matrices have no
380 % solution — not invertible
381 diverged = 0;
382
383 % OUTER LOOP
384 for i = 1:(outerLoopMax-1)
385 % Determine if the least squares matrices are invertible
386 if diverged == 0
387 % For each of the data collection (discrete time index)
388 for k = 1:nbar
389 % Initialize the optimal inputs to zero
390 uNew = [0; 0];
391 % INNER LOOP
392 for j = 1:(innerLoopMax-1)
393 % Get the updated input value

```

```

394         uLast = uNew;
395         % Update the error model
396         e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
397         % Compute the new optimal inputs
398         uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))*WcLast;
399
400         % Check convergence of the optimal inputs
401         if norm(uNew - uLast) < EpsilonPolicy
402             break;
403         end
404     end
405
406     % Update the values for the least-squares computation
407     V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
408     Lambda(k,:) = rho(e_vec(:,k))';
409 end
410 end
411
412 % Verify the least square solution exists for the critic's weights
413 % If the error data is consistent or there is no error, this will not
414 % hold, so set the weights to zero
415 if det(Lambda'*Lambda) == 0
416     fprintf('AWESOME...YOU HAVE NO ERROR... I 'M GOING TO SET THE WEIGHTS
TO ZERO\n');
417     weights = zeros(length(rho(e_vec(:,1))),1);
418     break;
419 end;
420
421 % Calculate least squares solution of critic's weights — EQ 20
422 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
423 % Make sure the weights did not diverge
424 % If the weights are diverging, just set them to a large number
425 if isnan(WcNew)
426     fprintf('OOPS...DIVERGING WEIGHTS... I 'M GOING TO USE LARGE WEIGHTS\
n');
427     weights = 1000*ones(length(rho(e_vec(:,1))),1);
428     break;
429 end;
430
431 % Check for convergence of the critic weights
432 if norm(WcNew - WcLast) < EpsilonWcritic
433     weights = WcNew;
434     fprintf('GREAT...THE WEIGHTS CONVERGED\n');
435     break;
436 end
437 % If the weights did not converge, repeat the loop
438 WcLast = WcNew;
439
440 % If we reached the last iteration of the loop, just use the last
441 % weights found
442 if (i == (outerLoopMax-1))
443     fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP... I 'M GOING
TO USE THE LAST VALUE\n');

```



```

444     weights = WcNew;
445     end
446     end
447 end
448
449 % CRITIC WEIGHT TUNING NEURAL NETWORK
450 % This function is specific to the helicopter because of the number of
451 % weights and error model
452 function weights = quanserAEROCriticTuning(A, B, e_vec, tau, R_Mat, Q_Mat, xd,
    wcInit)
453     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
454
455     % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
456     A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
457     B = [0 0; 0 0; 0.0512 0.0977; -0.1139 0.0928];
458     % System dimensions specific for our model
459     [n, ~] = size(B);
460
461     % ERROR MODEL OF THE HELICOPTER
462     % fbar and gbar — EQ 8
463     fbar = @(e) A*e;
464     gbar = -B;
465     hbar = -A*xd;
466
467     % DISCRETE-TIME ERROR MODEL FOR TIME TAU
468     % f and g — EQ 10
469     f = @(e) fbar(e)*tau + e;
470     g = gbar*tau;
471     h = hbar*tau;
472
473     % COST FUNCTION PARAMETERS
474     % State penalizing function in the continuous cost function
475     Qbar = @(e) e'*Q_Mat*e;
476     % Control penalizing matrix in the continuous cost function
477     Rbar = R_Mat;
478     % The discrete-time cost function will have terms:
479     % Right after EQ 11 in paper
480     % State penalizing function in the discretized cost function
481     Q = @(e) Qbar(e)*tau;
482     % Control penalizing matrix in the discretized cost function
483     R = Rbar*tau;
484
485     % NEURAL NETWORK FUNCTIONS
486     % Critic neural network activation functions
487     rho = @(e) [e(1); e(2); e(3); e(4); ...
488         e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4); ...
489         e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
490     % Partial derivative of rho with respect to e
491     drhode = @(e) [1, 0, 0, 0;
492         0, 1, 0, 0;
493         0, 0, 1, 0;
494         0, 0, 0, 1;
495         2*e(1), 0, 0, 0;
496         e(2), e(1), 0, 0;

```

```

497         e(3), 0, e(1), 0;
498         e(4), 0, 0, e(1);
499         0, 2*e(2), 0, 0;
500         0, e(3), e(2), 0;
501         0, e(4), 0, e(2);
502         0, 0, 2*e(3), 0;
503         0, 0, e(4), e(3);
504         0, 0, 0, 2*e(4)];
505
506 % TOLERANCES
507 % Convergence tolerance for control policy
508 EpsilonPolicy = 0.1;
509 % Convergence tolerance for critic neural network
510 EpsilonWcritic = 0.1;
511
512 % TRAINING PARAMETERS
513 % Number of outer loop iterations
514 outerLoopMax = 100;
515 % Number of inner loop iterations
516 innerLoopMax = 100;
517 % Number of equations needed for training, number of sub-intervals
518 % Number of training samples
519 [n,nbar] = size(e_vec);
520
521 % WEIGHT INITIALIZATION
522 % Initialize the weights of the critic neural network to zero
523 WcLast = zeros(length(rho(e_vec(:,1))),1);
524
525 % LEAST-SQUARES COMPUTATION INITIALIZATION
526 % Matrices required for computing least squares weights of the critic
527 % neural networks — EQ 20
528 V = zeros(nbar,1);
529 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
530
531 % Matrix to hold the derivative of the error model during policy
532 % updating
533 e_k_plus_1 = zeros(n,nbar);
534
535 % Product of the least squares matrices must be invertible
536 % Logic flag indicating if the critic weights are unsolvable
537 % The weights are unsolvable because the least squares matrices have no
538 % solution — not invertible
539 diverged = 0;
540
541 % OUTER LOOP
542 for i = 1:(outerLoopMax-1)
543 % Determine if the least squares matrices are invertible
544 if diverged == 0
545 % For each of the data collection (discrete time index)
546 for k = 1:nbar
547 % Initialize the optimal inputs to zero
548 uNew = [0; 0];
549 % INNER LOOP
550 for j = 1:(innerLoopMax-1)

```

```

551         % Get the updated input value
552         uLast = uNew;
553         % Update the error model
554         e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
555         % Compute the new optimal inputs
556         uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))'*WcLast;
557
558         % Check convergence of the optimal inputs
559         if norm(uNew - uLast) < EpsilonPolicy
560             break;
561         end
562     end
563
564     % Update the values for the least-squares computation
565     V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
566     Lambda(k,:) = rho(e_vec(:,k))';
567 end
568 end
569
570 % Verify the least square solution exists for the critic's weights
571 % If the error data is consistent or there is no error, this will not
572 % hold, so set the weights to what they were initially before the
573 % simulation
574 if det(Lambda'*Lambda) == 0
575     %fprintf('AWESOME...YOU HAVE NO ERROR...I 'M GOING TO USE THE
ORIGINAL WEIGHTS\n');
576     weights = wcInit;
577     break;
578 end;
579
580 % Calculcate least squares solution of critic's weights — EQ 20
581 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
582 % Make sure the weights did not diverge
583 % If the weights diverged, set the weights to what they were initially
584 % before the simulation
585 if isnan(WcNew)
586     %fprintf('OOPS...DIVERGING WEIGHTS...I 'M GOING TO USE THE ORIGINAL
WEIGHTS\n');
587     weights = wcInit;
588     break;
589 end;
590
591 % Check for convergence of the critic weights
592 if norm(WcNew - WcLast) < EpsilonWcritic
593     %fprintf('GREAT...THE WEIGHTS CONVERGED\n');
594     weights = WcNew;
595     break;
596 end
597 % If the weights did not converge, do another iteration of the loop
598 WcLast = WcNew;
599
600 % If we reach the last iteration of the loop, just use the last weights
601 if (i == (outerLoopMax-1))

```

```

602         %fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP...I'M GOING
        TO USE THE LAST VALUE\n');
603         weights = WcNew;
604     end
605 end
606 end

```

### B.4.6 ADP 2-DOF Helicopter

```

1 % Use exact model of the 2-DOF helicopter and the linearized model to find
2 % error data points
3 % Use the error data points directly in the neural network instead of
4 % randomizing the error
5 close all; clear; clc;
6 testName = 'heli_joint_adp_sinesine';
7
8 % LINEAR MODEL MATRIX PARAMETERS
9 [ A, B ] = getAeroAB( 0 )
10 % 2-DOF QUANSER HELICOPTER PARAMETERS
11 % Maximum applied voltage for the rotor motors
12 maxVolt = 18;
13 % Number of state variables – theta, psi, dtheta, dpsi
14 n = 4;
15 theta = 1;    % Pitch
16 psi = 2;     % Yaw
17 thetaDot = 3; % Pitch angular velocity
18 psiDot = 4;  % Yaw angular velocity
19 % Number of inputs – Vp, Vy
20 m = 2;
21
22 % SIMULATION TIMING
23 % Initial and final simulation times [s]
24 t0 = 0; tf = 25;
25 % Sampling time [s]
26 tau = 0.05;
27 % Larger sampling time for updating the inputs [s]
28 T = 0.75;
29 % Number of time steps
30 tsteps = floor((tf-t0)/tau);
31 % Discrete time vector of sampling time (tau)
32 dt = tau*(0:tsteps);
33 % Number of equations for actor-critic neural network
34 % Number of training samples per T
35 nbar = floor(T/tau);
36
37 % COST FUNCTION
38 % Q and R matrices used in the cost function
39 %Q_Mat_ADP = diag([270 100 1 1]);
40 %R_Mat_ADP = 0.0001*diag([1 1]);
41 % Use the matrices used in Quanser documentation except add ones for the
42 % velocities -> need Q to be positive definite
43 Q_Mat_ADP = diag([200 75 1 1]);
44 R_Mat_ADP = 0.005*diag([1 1]);
45 % Modified LQR cost matrices

```

```

46 Q_Mat_LQR = diag([200 75 1 1]);
47 R_Mat_LQR = 0.005*diag([1 1]);
48 nonLinearAdjustment = 0.03;
49 % Name of .mat file to save the data
50
51
52
53
54 % MATRIX INITIALIZATION
55 % Actual states – state variable x
56 xADP = zeros(n,tsteps+1);
57 % State errors for entire simulation
58 errorADP = zeros(n,tsteps+1);
59 % Actual inputs
60 uADP = zeros(m,tsteps+1);
61
62 % Desired states
63 % Uncomment for desired states of all zero
64 %xd = zeros(n,tsteps+1);
65 % Uncomment for desired states that switch constant values
66 %xd = [deg2rad(37)*ones(1,tsteps+1); deg2rad(80)*ones(1,tsteps+1); zeros(1,
        tsteps+1); zeros(1,tsteps+1)];
67 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
        (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
        ; zeros(1,tsteps+1)];
68 % Uncomment for desired states that are sine and cosine waves
69 pitchSine = deg2rad(60)*sin(0.5*dt);
70 yawCosine = deg2rad(90)*cos(0.5*dt);
71 % pitchSquare = deg2rad(57)*square(0.5*dt);
72 % yawSquare = deg2rad(-36)*square(0.25*dt);
73 xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
74 %xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
75
76
77
78 % FIND THE CRITIC NEURAL NETWORK WEIGHTS BEFORE APPLYING ANY INPUTS
79 % Calculate the critic weights
80 wcInit = quanserAEROCriticTuningInitial(A, B, ((2*pi).*rand(4,nbar)-pi),tau,
        R_Mat_ADP,Q_Mat_ADP,xd(:,1));
81 % Use the weights to determine the P matrix
82 P_Mat = [wcInit(5) wcInit(6) wcInit(7) wcInit(8);
83          wcInit(6) wcInit(9) wcInit(10) wcInit(11);
84          wcInit(7) wcInit(10) wcInit(12) wcInit(13);
85          wcInit(8) wcInit(11) wcInit(13) wcInit(14)];
86 % Use P to determine the state-feedback gain
87 kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
88
89
90
91
92
93 %% connection routine
94
95 ip = '127.0.0.1';

```

```

96 port = 19999;
97
98 % create a handle to remote API
99 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
100 vrep.simxFinish(-1); % just in case, close all opened connections
101 clientID = vrep.simxStart(ip, port, true, true, 5000, 5);
102
103 %% simulation
104 if (clientID > -1) % if clientID exists
105     disp('Connected to remote API server');
106
107 %Set up the handles
108 [rtn, aero] = vrep.simxGetObjectHandle(clientID, ...
109                                     'Aero', ...
110                                     vrep.simx_opmode_blocking)
111 [rtn, yJ] = vrep.simxGetObjectHandle(clientID, ...
112                                     'yawJoint', ...
113                                     vrep.simx_opmode_blocking)
114 [rtn, pJ] = vrep.simxGetObjectHandle(clientID, ...
115                                     'pitchJoint', ...
116                                     vrep.simx_opmode_blocking)
117
118 % setup joint position streaming
119 vrep.simxGetJointPosition(clientID, ...
120                           yJ, ...
121                           vrep.simx_opmode_streaming);
122 vrep.simxGetJointPosition(clientID, ...
123                           pJ, ...
124                           vrep.simx_opmode_streaming);
125
126 % add a delay to let the streaming operations initialize
127 pause(1);
128
129 [ret, pose(1,1)] = vrep.simxGetJointPosition(clientID, ...
130                                             pJ, ...
131                                             vrep.simx_opmode_buffer);
132
133 [ret, pose(2,1)] = vrep.simxGetJointPosition(clientID, ...
134                                             yJ, ...
135                                             vrep.simx_opmode_buffer);
136
137
138 pose(3,1) = 0;
139 pose(4,1) = 0;
140 xADP(:,1) = pose(:,1);
141 % Initialize the error
142 errorADP(:,1) = xd(:,1) - xADP(:,1);
143
144
145
146 % RUN THE SIMULATION
147 % Start at the next time tau after time zero
148
149

```

```

150 % RUN THE SIMULATION
151 % Start at the next time tau after time zero
152 tic;
153 for k = 2:tsteps+1
154     % Current time
155     t = (k-1)*tau; % generate value discrete time index
156
157     % Step 1 – get/store the state of the quadcopter from VREP
158     [ret , pose(1,k)] = vrep.simxGetJointPosition(clientID ,...
159                                                    pJ ,...
160                                                    vrep.simx_opmode_buffer)
161
162     ;
163
164     [ret , pose(2,k)] = vrep.simxGetJointPosition(clientID ,...
165                                                    yJ ,...
166                                                    vrep.simx_opmode_buffer)
167
168     ;
169
170     pose(3,k) = (pose(1,k) - pose(1,k-1))\tau;
171     pose(4,k) = (pose(2,k) - pose(2,k-1))\tau;
172
173     xADP(:,k) = pose(:,k);
174     % EXACT 2-DOF HELICOPTER MODEL
175     % Take previous values to find the derivative of the exact model
176     xdotNonLinearADP = A*xADP(:,k-1) + B*uADP(:,k-1);
177     % Update the states of the exact model
178     % Use Euler integration to estimate the current states of the nonlinear
179     % model
180     xADP(:,k) = xADP(:,k-1) + xdotNonLinearADP*tau + nonLinearAdjustment
181     .* (2.*rand(4,1)-1);
182     % ADD SOME DISTURBANCE TO THE STATES
183     % This forces the states to some other position
184     %if ((t > 15) && (t < 17))
185     %xADP(:,k) = [rad2deg(-45) 0 0 0]';
186     %xLQR(:,k) = [rad2deg(-45) 0 0 0]';
187     %end
188     % Force the pitch angle to be in the range [-pi/2,pi/2] because of
189     % physical constraints
190     xADP(theta,k) = angleLimiterPitch(xADP(theta,k));
191     % Force the yaw angle to be in the range [-pi,pi] because the yaw is
192     % free to do a complete circle
193     xADP(psi,k) = angleLimiterYaw(xADP(psi,k));
194
195     % FIND THE ERROR BETWEEN THE EXACT STATES AND THE DESIRED STATES
196     errorADP(:,k) = xd(:,k) - xADP(:,k);
197
198     % UPDATE THE CRITIC WEIGHTS EVERY T TIME
199     % Determine if the time is a multiple of T
200     if (mod(t,T) == 0)
201         % Update the critic weights
202         wc = quanserAEROCriticTuning(A, B, errorADP(:,(k-nbar):k), tau,
203         R_Mat_ADP, Q_Mat_ADP, xd(:,k), wcInit);
204         % Use the weights to determine the P matrix

```

```

200     P_Mat = [wc(5) wc(6) wc(7) wc(8);
201             wc(6) wc(9) wc(10) wc(11);
202             wc(7) wc(10) wc(12) wc(13);
203             wc(8) wc(11) wc(13) wc(14)];
204     % Use P to determine the state-feedback gain
205     kADP = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
206     end
207     % Update the inputs using ADP
208     uNewADP = kADP*errorADP(:,k);
209     % Limit the voltages
210     uNewADP(1) = sign(uNewADP(1))*min(abs(uNewADP(1)),maxVolt);
211     uNewADP(2) = sign(uNewADP(2))*min(abs(uNewADP(2)),maxVolt);
212     % Update the input matrices
213     uADP(1,k) = uNewADP(1);
214     uADP(2,k) = uNewADP(2);
215
216     Vin=[uNewADP(1) uNewADP(2) rad2deg(xd(1,k)) rad2deg(xd(2,k))];
217     packedData=vrep.simxPackFloats(Vin);
218     vrep.simxSetStringSignal(clientID,...
219                             'MATLAB_SIG',...
220                             packedData,...
221                             vrep.simx_opmode_oneshot);
222
223     pause(tau);
224
225     t = t + tau;
226
227
228 end
229 toc;
230
231 %ret = 1;
232 while(ret ~= 0)
233     [ret]=vrep.simxClearStringSignal(clientID,...
234                                     'MATLAB_SIG',...
235                                     vrep.simx_opmode_blocking);
236 end
237
238 else
239     disp('Failed connecting to remote API server');
240 end % if (clientID>-1)
241
242 disp('Sim ended');
243
244
245 figure
246 plot(dt,rad2deg(pose(1,:))); hold on;
247 plot(dt,rad2deg(xd(1,:))); hold off;
248 title('pitch')
249
250 figure
251 plot(dt,rad2deg(pose(2,:))); hold on;
252 plot(dt,rad2deg(xd(2,:))); hold off;
253 title('yaw')

```



```

254
255 figure
256 plot(dt,uADP(1,:)); hold on;
257 plot(dt,uADP(2,:)); hold off;
258 title('Inputs')
259
260 % SAVE OUTPUTS FOR PLOTTING
261 % Save the time, error, input, states and desired states vectors
262 save([testName, '.mat'], 'dt', 'errorADP', 'uADP', 'xADP', 'xd', 'tau', 'R_Mat_ADP',
      'Q_Mat_ADP');
263 % Notify that the simulation is complete
264 disp('SIMULATION COMPLETE');
265
266 % ANGLE LIMITER FOR YAW FUNCTION
267 function angle = angleLimiterYaw(angle)
268 % This function limits the angle between -pi and pi
269     angle = mod(angle, 2*pi);
270
271     i=find(angle>pi);
272     angle(i)=angle(i)-2*pi;
273
274     i=find(angle<-pi);
275     angle(i)=angle(i)+2*pi;
276 end
277
278 % ANGLE LIMITER FOR PITCH FUNCTION
279 function angle = angleLimiterPitch(angle)
280 % This function limits the physical constraint of the pitch measurement to
281 % 90 degrees
282     if (angle < 0)
283         angle = max(angle, -pi/2);
284     end
285
286     if (angle > 0)
287         angle = min(angle, pi/2);
288     end
289 end
290
291 % CRITIC WEIGHT TUNING NEURAL NETWORK
292 % This function is specific to the helicopter because of the number of
293 % weights and error model
294 function weights = quanserAEROCriticTuningInitial(A, B, e_vec, tau, R_Mat,
      Q_Mat, xd)
295     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
296
297     % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
298     % A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
299     % B = [0 0; 0 0; 0.0512 0.0977; -0.1139 0.0928];
300     % System dimensions specific for our model
301     [n, ~] = size(B);
302
303     % ERROR MODEL OF THE HELICOPTER
304     % fbar and gbar — EQ 8
305     fbar = @(e) A*e;

```

```

306   gbar = -B;
307   hbar = -A*xd;
308
309   % DISCRETE-TIME ERROR MODEL FOR TIME TAU
310   % f and g — EQ 10
311   f = @(e) fbar(e)*tau + e;
312   g = gbar*tau;
313   h = hbar*tau;
314
315   % COST FUNCTION PARAMETERS
316   % State penalizing function in the continuous cost function
317   Qbar = @(e) e'*Q_Mat*e;
318   % Control penalizing matrix in the continuous cost function
319   Rbar = R_Mat;
320   % The discrete-time cost function will have terms:
321   % Right after EQ 11 in paper
322   % State penalizing function in the discretized cost function
323   Q = @(e) Qbar(e)*tau;
324   % Control penalizing matrix in the discretized cost function
325   R = Rbar*tau;
326
327   % NEURAL NETWORK FUNCTIONS
328   % Critic neural network activation functions
329   rho = @(e) [e(1); e(2); e(3); e(4);...
330              e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
331              e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
332   % Partial derivative of rho with respect to e
333   drhode = @(e) [1, 0, 0, 0;
334                  0, 1, 0, 0;
335                  0, 0, 1, 0;
336                  0, 0, 0, 1;
337                  2*e(1), 0, 0, 0;
338                  e(2), e(1), 0, 0;
339                  e(3), 0, e(1), 0;
340                  e(4), 0, 0, e(1);
341                  0, 2*e(2), 0, 0;
342                  0, e(3), e(2), 0;
343                  0, e(4), 0, e(2);
344                  0, 0, 2*e(3), 0;
345                  0, 0, e(4), e(3);
346                  0, 0, 0, 2*e(4)];
347
348   % TOLERANCES
349   % Convergence tolerance for control policy
350   EpsilonPolicy = 0.1;
351   % Convergence tolerance for critic neural network
352   EpsilonWcritic = 0.1;
353
354   % TRAINING PARAMETERS
355   % Number of outer loop iterations
356   outerLoopMax = 700;
357   % Number of inner loop iterations
358   innerLoopMax = 100;
359   % Number of equations needed for training, number of sub-intervals

```

```

360 % Number of training samples
361 [~,nbar] = size(e_vec);
362
363 % WEIGHT INITIALIZATION
364 % Initialize the weights of the critic neural network to zero
365 WcLast = zeros(length(rho(e_vec(:,1))),1);
366
367 % LEAST-SQUARES COMPUTATION INITIALIZATION
368 % Matrices required for computing least squares weights of the critic
369 % neural networks — EQ 20
370 V = zeros(nbar,1);
371 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
372
373 % Matrix to hold the derivative of the error model during policy
374 % updating
375 e_k_plus_1 = zeros(n,nbar);
376
377 % Product of the least squares matrices must be invertible
378 % Logic flag indicating if the critic weights are unsolvable
379 % The weights are unsolvable because the least squares matrices have no
380 % solution — not invertible
381 diverged = 0;
382
383 % OUTER LOOP
384 for i = 1:(outerLoopMax-1)
385 % Determine if the least squares matrices are invertible
386 if diverged == 0
387 % For each of the data collection (discrete time index)
388 for k = 1:nbar
389 % Initialize the optimal inputs to zero
390 uNew = [0; 0];
391 % INNER LOOP
392 for j = 1:(innerLoopMax-1)
393 % Get the updated input value
394 uLast = uNew;
395 % Update the error model
396 e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
397 % Compute the new optimal inputs
398 uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))'*WcLast;
399
400 % Check convergence of the optimal inputs
401 if norm(uNew - uLast) < EpsilonPolicy
402 break;
403 end
404 end
405
406 % Update the values for the least-squares computation
407 V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
408 Lambda(k,:) = rho(e_vec(:,k))';
409 end
410 end
411
412 % Verify the least square solution exists for the critic's weights

```

```

413 % If the error data is consistent or there is no error, this will not
414 % hold, so set the weights to zero
415 if det(Lambda'*Lambda) == 0
416     fprintf('AWESOME...YOU HAVE NO ERROR... I 'M GOING TO SET THE WEIGHTS
417     TO ZERO\n');
418     weights = zeros(length(rho(e_vec(:,1))),1);
419     break;
420 end;
421 % Calculate least squares solution of critic's weights — EQ 20
422 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
423 % Make sure the weights did not diverge
424 % If the weights are diverging, just set them to a large number
425 if isnan(WcNew)
426     fprintf('OOPS...DIVERGING WEIGHTS... I 'M GOING TO USE LARGE WEIGHTS\
427     n');
428     weights = 1000*ones(length(rho(e_vec(:,1))),1);
429     break;
430 end;
431 % Check for convergence of the critic weights
432 if norm(WcNew - WcLast) < EpsilonWcritic
433     weights = WcNew;
434     fprintf('GREAT...THE WEIGHTS CONVERGED\n');
435     break;
436 end
437 % If the weights did not converge, repeat the loop
438 WcLast = WcNew;
439
440 % If we reached the last iteration of the loop, just use the last
441 % weights found
442 if (i == (outerLoopMax-1))
443     fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP... I 'M GOING
444     TO USE THE LAST VALUE\n');
445     weights = WcNew;
446 end
447 end
448
449 % CRITIC WEIGHT TUNING NEURAL NETWORK
450 % This function is specific to the helicopter because of the number of
451 % weights and error model
452 function weights = quanserAEROCriticTuning(A, B, e_vec, tau, R_Mat, Q_Mat, xd,
453     wcInit)
454     % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
455
456     % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
457     A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
458     B = [0 0; 0 0; 0.0512 0.0977; -0.1139 0.0928];
459     % System dimensions specific for our model
460     [n, ~] = size(B);
461
462     % ERROR MODEL OF THE HELICOPTER
463     % fbar and gbar — EQ 8

```

```

463 fbar = @(e) A*e;
464 gbar = -B;
465 hbar = -A*xd;
466
467 % DISCRETE-TIME ERROR MODEL FOR TIME TAU
468 % f and g — EQ 10
469 f = @(e) fbar(e)*tau + e;
470 g = gbar*tau;
471 h = hbar*tau;
472
473 % COST FUNCTION PARAMETERS
474 % State penalizing function in the continuous cost function
475 Qbar = @(e) e'*Q_Mat*e;
476 % Control penalizing matrix in the continuous cost function
477 Rbar = R_Mat;
478 % The discrete-time cost function will have terms:
479 % Right after EQ 11 in paper
480 % State penalizing function in the discretized cost function
481 Q = @(e) Qbar(e)*tau;
482 % Control penalizing matrix in the discretized cost function
483 R = Rbar*tau;
484
485 % NEURAL NETWORK FUNCTIONS
486 % Critic neural network activation functions
487 rho = @(e) [e(1); e(2); e(3); e(4);...
488             e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
489             e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
490 % Partial derivative of rho with respect to e
491 drhode = @(e) [1, 0, 0, 0;
492                0, 1, 0, 0;
493                0, 0, 1, 0;
494                0, 0, 0, 1;
495                2*e(1), 0, 0, 0;
496                e(2), e(1), 0, 0;
497                e(3), 0, e(1), 0;
498                e(4), 0, 0, e(1);
499                0, 2*e(2), 0, 0;
500                0, e(3), e(2), 0;
501                0, e(4), 0, e(2);
502                0, 0, 2*e(3), 0;
503                0, 0, e(4), e(3);
504                0, 0, 0, 2*e(4)];
505
506 % TOLERANCES
507 % Convergence tolerance for control policy
508 EpsilonPolicy = 0.1;
509 % Convergence tolerance for critic neural network
510 EpsilonWcritic = 0.1;
511
512 % TRAINING PARAMETERS
513 % Number of outer loop iterations
514 outerLoopMax = 100;
515 % Number of inner loop iterations
516 innerLoopMax = 100;

```

```

517 % Number of equations needed for training , number of sub-intervals
518 % Number of training samples
519 [~,nbar] = size(e_vec);
520
521 % WEIGHT INITIALIZATION
522 % Initialize the weights of the critic neural network to zero
523 WcLast = zeros(length(rho(e_vec(:,1))),1);
524
525 % LEAST-SQUARES COMPUTATION INITIALIZATION
526 % Matrices required for computing least squares weights of the critic
527 % neural networks — EQ 20
528 V = zeros(nbar,1);
529 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
530
531 % Matrix to hold the derivative of the error model during policy
532 % updating
533 e_k_plus_1 = zeros(n,nbar);
534
535 % Product of the least squares matrices must be invertible
536 % Logic flag indicating if the critic weights are unsolvable
537 % The weights are unsolvable because the least squares matrices have no
538 % solution — not invertible
539 diverged = 0;
540
541 % OUTER LOOP
542 for i = 1:(outerLoopMax-1)
543 % Determine if the least squares matrices are invertible
544 if diverged == 0
545 % For each of the data collection (discrete time index)
546 for k = 1:nbar
547 % Initialize the optimal inputs to zero
548 uNew = [0; 0];
549 % INNER LOOP
550 for j = 1:(innerLoopMax-1)
551 % Get the updated input value
552 uLast = uNew;
553 % Update the error model
554 e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast + h;
555 % Compute the new optimal inputs
556 uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))'*WcLast;
557
558 % Check convergence of the optimal inputs
559 if norm(uNew - uLast) < EpsilonPolicy
560 break;
561 end
562 end
563
564 % Update the values for the least-squares computation
565 V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1
(:,k));
566 Lambda(k,:) = rho(e_vec(:,k))';
567 end
568 end
569

```

```

570 % Verify the least square solution exists for the critic 's weights
571 % If the error data is consistent or there is no error, this will not
572 % hold, so set the weights to what they were initially before the
573 % simulation
574 if det(Lambda'*Lambda) == 0
575     %fprintf('AWESOME...YOU HAVE NO ERROR...I 'M GOING TO USE THE
ORIGINAL WEIGHTS\n');
576     weights = wcInit;
577     break;
578 end;
579
580 % Calculcate least squares solution of critic 's weights — EQ 20
581 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
582 % Make sure the weights did not diverge
583 % If the weights diverged, set the weights to what they were initially
584 % before the simulation
585 if isnan(WcNew)
586     %fprintf('OOPS...DIVERGING WEIGHTS...I 'M GOING TO USE THE ORIGINAL
WEIGHTS\n');
587     weights = wcInit;
588     break;
589 end;
590
591 % Check for convergence of the critic weights
592 if norm(WcNew - WcLast) < EpsilonWcritic
593     %fprintf('GREAT...THE WEIGHTS CONVERGED\n');
594     weights = WcNew;
595     break;
596 end
597 % If the weights did not converge, do another iteration of the loop
598 WcLast = WcNew;
599
600 % If we reach the last iteration of the loop, just use the last weights
601 if (i == (outerLoopMax-1))
602     %fprintf('OOPS...YOU REACHED THE END OF THE OUTER LOOP...I 'M GOING
TO USE THE LAST VALUE\n');
603     weights = WcNew;
604 end
605 end
606 end

```

## B.5 Motor Approach

### B.5.1 Quanser AERO Base

```

1 --- Initialization


---


2 if (sim_call_type==sim.syscb_init) then
3     simRemoteApi.start(19999)
4
5     pJ = sim.getObjectHandle('pitchJoint');
6     yJ = sim.getObjectHandle('yawJoint');

```

```

7   prop0 = sim.getObjectHandle('motor0');
8   prop1 = sim.getObjectHandle('motor1');
9   timestep = sim.getSimulationTimeStep()
10  print('timestep is : ', timestep)
11  — Get parameter values to work with
12  Jy = sim.getScriptSimulationParameter(sim.handle_self, 'Jy')
13  print('Jy is : ', Jy)
14  Jp = sim.getScriptSimulationParameter(sim.handle_self, 'Jp')
15  print('Jp is : ', Jp)
16  Dp = sim.getScriptSimulationParameter(sim.handle_self, 'Dp')
17  print('Dp is : ', Dp)
18  Dy = sim.getScriptSimulationParameter(sim.handle_self, 'Dy')
19  print('Dy is : ', Dy)
20  Kpp = sim.getScriptSimulationParameter(sim.handle_self, 'Kpp')
21  print('Kpp is : ', Kpp)
22  Kyy = sim.getScriptSimulationParameter(sim.handle_self, 'Kyy')
23  print('Kyy is : ', Kyy)
24  Ksp = sim.getScriptSimulationParameter(sim.handle_self, 'Ksp')
25  print('Ksp is : ', Ksp)
26
27  — Store current measured angular velocity to calculate angular
  acceleration
28  prev_pJ_pose = sim.getJointPosition(pJ)
29  prev_yJ_pose = sim.getJointPosition(yJ)
30
31  sim.setJointTargetVelocity(pJ, 0)
32  sim.setJointTargetVelocity(yJ, 0)
33  sim.setJointTargetVelocity(prop0,0)
34  sim.setJointTargetVelocity(prop1,0)
35  pitchTarVel = 0
36  yawTarVel = 0
37
38  graphHandle=sim.getObjectHandle("Graph")
39
40  PitchDesired = sim.getIntegerSignal("PitchDesired")
41  lastPAcc = 0
42  lastYAcc = 0
43
44  sim.setJointTargetVelocity(prop0,1*0.8)
45  sim.setJointTargetVelocity(prop1,-1*0.8)
46
47 end
48
49 — Looping code


---


50 if (sim_call_type==sim.syscb_actuation) then
51
52     local t = sim.getSimulationTime()
53     — Receive velocities from server
54     local packedData=sim.getStringSignal('MATLAB_SIG')
55     if packedData then
56         sim.clearStringSignal('MATLAB_SIG') — Clear the signal
57         local V=sim.unpackFloatTable(packedData,0,4,0)

```



```

58
59     — Measure pitch and yaw
60     pJ_pose = sim.getJointPosition(pJ)
61     yJ_pose = sim.getJointPosition(yJ)
62
63     — Calculate Velocites
64     pJ_vel = (pJ_pose - prev_pJ_pose)/timestep
65     yJ_vel = (yJ_pose - prev_yJ_pose)/timestep
66
67     pAcc = -(Ksp/Jy)*math.sin(pJ_pose) + -(Dy/Jy)*pJ_vel + -(Kpp/Jy)*V
[1] + (Kpp/Jy)*V[2]
68     yAcc = -(Dp/Jp)*yJ_vel + -(Kyy/Jy)*V[1] + -(Kyy/Jy)*V[2]
69
70
71     pitchTarVel = pitchTarVel + ((pAcc+lastPAcc)/2)*timestep
72     yawTarVel = yawTarVel + ((yAcc+lastYAcc)/2)*timestep
73
74     sim.setJointTargetVelocity(pJ, pitchTarVel)
75     sim.setJointTargetVelocity(yJ, yawTarVel)
76
77     prev_pJ_pose = pJ_pose
78     prev_yJ_pose = yJ_pose
79     lastPAcc = pAcc
80     lastYAcc = yAcc
81     sim.setGraphUserData(graphHandle, 'PitchDesired',V[3])
82     sim.setGraphUserData(graphHandle, 'YawDesired',V[4])
83
84     sim.setJointTargetVelocity(prop0,V[1]*0.8)
85     sim.setJointTargetVelocity(prop1,-V[2]*0.8)
86
87     end
88
89 end
90
91 function sysCall_sensing()
92 end
93
94 if (sim_call_type==sim.syscb_cleanup) then
95
96 end

```

### B.5.2 Quanser AERO Motor#0

```

1 function getI(V)
2
3     a = 0.0000277
4     b = -0.00000966
5     c = 0.0213
6     d = -0.00319
7     I = a*V*V*V + b*V*V + c*V + d;
8     return I
9 end
10
11 if (sim_call_type==sim.syscb_init) then

```

```

12
13  -- Get Handles for Propeller Objects and simulation timestep
14  propeller_obj=sim.getObjectAssociatedWithScript(sim.handle_self)
15  propeller_respondable=sim.getObjectHandle('propeller_respondable');
16  timestep = sim.getSimulationTimeStep()
17
18  -- Input voltage [Control]
19  Vin = sim.getScriptSimulationParameter(sim.handle_self, 'Vin')
20  -- Constants from Quanser Model
21  -- Thrust Coefficient in pitch
22  Kpp = sim.getScriptSimulationParameter(sim.handle_self, 'Kpp')
23  -- Drag coefficient [m/(rad/s)]
24  Kd = sim.getScriptSimulationParameter(sim.handle_self, 'Kd')
25  -- Motor internal resistance [ohms]
26  Rm = sim.getScriptSimulationParameter(sim.handle_self, 'Rm')
27  -- Motor torque constant [same as Km] [N*m/A]
28  Kt = sim.getScriptSimulationParameter(sim.handle_self, 'Kt')
29  -- Motor back EMF constant [same as Kt] [V/(rad/s)]
30  Km = sim.getScriptSimulationParameter(sim.handle_self, 'Km')
31  -- Rotor Inertia [kg*m^2]
32  Jm = sim.getScriptSimulationParameter(sim.handle_self, 'Jm')
33  -- Hub Inertia [kg*m^2]
34  Jh = sim.getScriptSimulationParameter(sim.handle_self, 'Jh')
35  -- Propeller Inertia [kg*m^2]
36  Jp = sim.getScriptSimulationParameter(sim.handle_self, 'Jp')
37  -- Inductance [H]
38  L = sim.getScriptSimulationParameter(sim.handle_self, 'L')
39  -- Propeller Center Radius
40  R = sim.getScriptSimulationParameter(sim.handle_self, 'R')
41
42  momentOfInertia = sim.getScriptSimulationParameter(sim.handle_self, '
motorMomentOfInertia')
43
44  -- Store current measured angular velocity to calculate angular
acceleration
45  prev_i_m = 0
46
47 end
48
49 if (sim_call_type==sim.syscb_actuation) then
50
51  -- Get the transformation matrix relative to global coordinates
52  m=sim.getObjectMatrix(propeller_respondable,-1)
53
54  -- assign Vin (this is set by the Aero script) [Volts]
55  Vin=sim.getScriptSimulationParameter(sim.handle_self, 'Vin')
56
57  -- calc motor current [Amps]
58  i_m = getI(Vin)
59
60  -- estimate di_dt for inductance calc in KVL loop
61  di_dt = (i_m - prev_i_m)/timestep
62
63  -- estimate motor rpm [rad/s]

```

```

64  omega_motor = (Vin - Rm*i_m - L*di_dt)/(Km)
65
66  -- set the thrust equal to Kpp*Vin
67  thrust = -Kpp*Vin/R
68
69  --assign 3 dimensional force
70  force={0.0,0.0, thrust}
71
72  -- Get rotation matrix and its inverse
73  m_rotation = m
74
75  -- remove translational component from
76  -- function return
77  m_rotation[4]=0
78  m_rotation[8]=0
79  m_rotation[12]=0
80
81  -- Calculate the coupling torque
82  torque_m = Km*i_m
83
84  -- Calculate Rotational Drag Torque (negative if clockwise and positive
85  if cclockwise)
86  torque_d = Kd*omega_motor
87
88  -- Calculate total Torque
89  torque = {0.0, 0.0, 0.0}
90  torque[3] = torque_m + torque_d
91
92  if(mccw) then
93      torque[3] = -torque[3]
94  end
95
96  -- Apply force and torque onto the body
97  force=sim.multiplyVector(m_rotation, force)
98  torque=sim.multiplyVector(m_rotation, torque)
99
100  sim.addForceAndTorque(propeller_respondable, force, torque)
101
102  -- Store current
103  prev_i_m = i_m
104 end
105
106 if (sim_call_type==sim.syscb_cleanup) then
107     -- Do nothing
108 end

```

### B.5.3 Quanser AERO Motor#1

```

1  function getI(V)
2
3  a = 0.0000277
4  b = -0.00000966
5  c = 0.0213

```

```

6   d = -0.00319
7   I = a*V*V*V + b*V*V + c*V + d;
8   return I
9 end
10
11 if (sim_call_type==sim.syscb_init) then
12
13     — Get Handles for Propeller Objects and simulation timestep
14     propeller_obj=sim.getObjectAssociatedWithScript(sim.handle_self)
15     propeller_respondable=sim.getObjectHandle('propeller_respondable');
16     timestep = sim.getSimulationTimeStep()
17
18     — Input voltage [Control]
19     Vin = sim.getScriptSimulationParameter(sim.handle_self, 'Vin')
20
21     — Constants from Quanser Model
22     — Thrust Coefficient in pitch
23     Kpp = sim.getScriptSimulationParameter(sim.handle_self, 'Kpp')
24     — Drag coefficient [m/(rad/s)]
25     Kd = sim.getScriptSimulationParameter(sim.handle_self, 'Kd')
26     — Motor internal resistance [ohms]
27     Rm = sim.getScriptSimulationParameter(sim.handle_self, 'Rm')
28     — Motor torque constant [same as Km] [N*m/A]
29     Kt = sim.getScriptSimulationParameter(sim.handle_self, 'Kt')
30     — Motor back EMF constant [same as Kt] [V/(rad/s)]
31     Km = sim.getScriptSimulationParameter(sim.handle_self, 'Km')
32     — Rotor Inertia [kg*m^2]
33     Jm = sim.getScriptSimulationParameter(sim.handle_self, 'Jm')
34     — Hub Inertia [kg*m^2]
35     Jh = sim.getScriptSimulationParameter(sim.handle_self, 'Jh')
36     — Propeller Inertia [kg*m^2]
37     Jp = sim.getScriptSimulationParameter(sim.handle_self, 'Jp')
38     — Inductance [H]
39     L = sim.getScriptSimulationParameter(sim.handle_self, 'L')
40     — Propeller Center Radius
41     R = sim.getScriptSimulationParameter(sim.handle_self, 'R')
42
43     momentOfInertia = sim.getScriptSimulationParameter(sim.handle_self, '
44     motorMomentOfInertia')
45
46     — Store current measured angular velocity to calculate angular
47     acceleration
48     prev_i_m = 0
49
50 end
51
52 if (sim_call_type==sim.syscb_actuation) then
53     spinDir = mccw
54
55     — Get the transformation matrix relative to global coordinates
56     m=sim.getObjectMatrix(propeller_respondable, -1)
57
58     — assign Vin (this is set by the Aero script) [Volts]
59     Vin=sim.getScriptSimulationParameter(sim.handle_self, 'Vin')

```

```

58
59  -- calc motor current [Amps]
60  i_m = getI(Vin)
61
62  -- estimate di_dt for inductance calc in KVL loop
63  di_dt = (i_m - prev_i_m)/timestep
64
65  -- estimate motor rpm [rad/s]
66  omega_motor = (Vin - Rm*i_m - L*di_dt)/(Km)
67
68  -- set the thrust equal to Kpp*Vin
69  thrust = -Kpp*Vin/R
70
71  --assign 3 dimensional force
72  force={0.0,0.0, thrust}
73
74  -- Get rotation matrix and its inverse
75  m_rotation = m
76
77  -- remove translational component from
78  -- function return
79  m_rotation[4]=0
80  m_rotation[8]=0
81  m_rotation[12]=0
82
83  -- Calculate the coupling torque
84  torque_m = -Km*i_m
85
86  -- Calculate Rotational Drag Torque (negative if clockwise and positive
87  if cclockwise)
88  torque_d = -Kd*omega_motor
89
90  -- Calculate total Torque
91  torque = {0.0, 0.0, 0.0}
92  torque[3] = torque_m + torque_d
93
94
95  -- Apply force and torque onto the body
96  force=sim.multiplyVector(m_rotation, force)
97  torque=sim.multiplyVector(m_rotation, torque)
98
99  sim.addForceAndTorque(propeller_respondable, force, torque)
100
101  -- Store current
102  prev_i_m = i_m
103 end
104
105 if (sim_call_type==sim.syscb_cleanup) then
106   -- Do nothing
107 end

```

### B.5.4 Quanser AERO Motor PID Control

```

1 % Use exact model of the 2-DOF helicopter and the linearized model to find
2 % error data points
3 % Use the error data points directly in the neural network instead of
4 % randomizing the error
5 close all; clear; clc;
6
7 % Name of .mat file to save the data
8 testName = 'heli_joint_sqsq';
9
10 % LINEAR MODEL MATRIX PARAMETERS (2DoF Helicopter)
11 [ A, B ] = getAeroAB( 0 );
12
13 % 2-DOF QUANSER HELICOPTER PARAMETERS
14 % Maximum applied voltage for the rotor motors
15 maxVolt = 24;
16 % Number of state variables – theta, psi, dtheta, dpsi
17 n = 4;
18 theta = 1;    % Pitch
19 psi = 2;      % Yaw
20 thetaDot = 3; % Pitch angular velocity
21 psiDot = 4;   % Yaw angular velocity
22 % Number of inputs – Vp, Vy
23 m = 2;
24
25 % SIMULATION TIMING
26 % Initial and final simulation times [s]
27 t0 = 0; tf = 30;
28 % Sampling time [s]
29 tau = 0.05;
30 % Larger sampling time for updating the inputs [s]
31 T = 0.2;
32 % Number of time steps
33 tsteps = floor((tf-t0)/tau);
34 % Discrete time vector of sampling time (tau)
35 dt = tau*(0:tsteps);
36
37
38 % INITIAL STATES
39 % xInit = [deg2rad(0) deg2rad(0) (0*pi/180)*10 (0*pi/180)*7]';
40 %xInit = [0 0 0 0]';
41
42 % MATRIX INITIALIZATION
43 % Actual states – state variable x
44 pose = zeros(n,tsteps+1);
45
46 % State errors for entire simulation
47 error = zeros(n,tsteps+1);
48 % Actual inputs
49 u = zeros(m,tsteps+1);
50
51 % Desired states
52 % Uncomment for desired states of all zero
53 %xd = zeros(n,tsteps+1);
54 % Uncomment for desired states that switch constant values

```

```

55 %xd = [deg2rad(-30)*ones(1,tsteps+1); deg2rad(-80)*ones(1,tsteps+1); zeros
        (1,tsteps+1); zeros(1,tsteps+1)];
56 %xd = [deg2rad(10)*ones(1,1000), deg2rad(60)*ones(1,tsteps+1-1000); deg2rad
        (80)*ones(1,1500), deg2rad(-120)*ones(1,tsteps+1-1500); zeros(1,tsteps+1)
        ; zeros(1,tsteps+1)];
57 % Uncomment for desired states that are sine and cosine waves
58 pitchSine = deg2rad(30)*sin(0.5*dt);
59 yawCosine = deg2rad(60)*cos(0.5*dt);
60 % pitchSquare = deg2rad(25)*square(0.5*dt);
61 % yawSquare = deg2rad(-36)*square(0.25*dt);
62 xd = [pitchSine; yawCosine; zeros(1,tsteps+1); zeros(1,tsteps+1)];
63 % xd = [pitchSquare; yawSquare; zeros(1,tsteps+1); zeros(1,tsteps+1)];
64
65
66 %% connection routine
67 ip = '127.0.0.1';
68 port = 19999;
69
70 % create a handle to remote API
71 vrep = remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
72 vrep.simxFinish(-1); % just in case, close all opened connections
73 clientID = vrep.simxStart(ip,port,true,true,5000,5);
74
75 %% simulation
76 if (clientID > -1) % if clientID exists
77     disp('Connected to remote API server');
78
79 %Set up the handles
80 [rtn, aero] = vrep.simxGetObjectHandle(clientID,...
81                                     'Aero',...
82                                     vrep.simx_opmode_blocking)
83 [rtn, yJ] = vrep.simxGetObjectHandle(clientID,...
84                                     'yawJoint',...
85                                     vrep.simx_opmode_blocking)
86 [rtn, pJ] = vrep.simxGetObjectHandle(clientID,...
87                                     'pitchJoint',...
88                                     vrep.simx_opmode_blocking)
89
90 % setup joint position streaming
91 vrep.simxGetJointPosition(clientID,...
92                           yJ,...
93                           vrep.simx_opmode_streaming);
94 vrep.simxGetJointPosition(clientID,...
95                           pJ,...
96                           vrep.simx_opmode_streaming);
97
98 % add a delay to let the streaming operations initialize
99 pause(1);
100
101 [ret, pose(1,1)] = vrep.simxGetJointPosition(clientID,...
102                                             pJ,...
103                                             vrep.simx_opmode_buffer);
104
105 [ret, pose(2,1)] = vrep.simxGetJointPosition(clientID,...

```

```

106                                     yJ, ...
107                                     vrep.simx_opmode_buffer);
108
109
110 pose(3,1) = 0;
111 pose(4,1) = 0;
112 % Initialize the error
113 error(:,1) = xd(:,1) - pose(:,1);
114
115 k_d = 3;
116 k_i = 10;
117 k_p = 8;
118
119 k_1 = 4;
120 k_2 = 4;
121
122 e_theta_sum = 0;
123 e_psi_sum = 0;
124
125 tic;
126 % RUN THE SIMULATION
127 % Start at the next time tau after time zero
128 for k = 2:tsteps+1
129     % Current time
130     t = (k-1)*tau; % generate value discrete time index
131
132     % Step 1 - get/store the state of the quadcopter from VREP
133     [ret, pose(1,k)] = vrep.simxGetJointPosition(clientID, ...
134                                                     pJ, ...
135                                                     vrep.simx_opmode_buffer)
136     ;
137     [ret, pose(2,k)] = vrep.simxGetJointPosition(clientID, ...
138                                                     yJ, ...
139                                                     vrep.simx_opmode_buffer)
140     ;
141     pose(3,k) = (pose(1,k) - pose(1,k-1))/tau;
142     pose(4,k) = (pose(2,k) - pose(2,k-1))/tau;
143
144     if(k>=3)
145         pose(3,k) = (pose(3,k) + pose(3,k-1) + pose(3,k-2))/3;
146         pose(4,k) = (pose(4,k) + pose(4,k-1) + pose(4,k-2))/3;
147     end
148
149     error(:,k) = xd(:,k) - pose(:,k);
150
151     e_theta_sum = e_theta_sum + error(1,k)*tau;
152     e_theta = k_p*error(1,k) + k_i*e_theta_sum + k_d*error(3,k);
153
154     e_psi_sum = e_psi_sum + error(2,k)*tau;
155     e_psi = k_p*error(2,k) + k_i*e_psi_sum + k_d*error(4,k);
156
157     u(1,k) = -k_1*e_theta - k_2*e_psi;

```



```

158     u(2,k) = -k_2*e_psi+k_1*e_theta;
159
160     u(1,k) = sign(u(1,k))*min(abs(u(1,k)),maxVolt);
161     u(2,k) = sign(u(2,k))*min(abs(u(2,k)),maxVolt);
162
163     % Step 3 – Update Inputs
164     Vin=[u(1,k) u(2,k)];
165     %   motorSpeeds=[20 0];
166     packedData=vrep.simxPackFloats(Vin);
167     vrep.simxSetStringSignal(clientID,...
168                             'MATLAB_SIG',...
169                             packedData,...
170                             vrep.simx_opmode_oneshot);
171
172     pause(tau);
173
174     t = t + tau;
175
176
177 end
178 toc;
179
180 %ret = 1;
181 while (ret ~= 0)
182     [ret]=vrep.simxClearStringSignal(clientID,...
183                                     'MATLAB_SIG',...
184                                     vrep.simx_opmode_blocking);
185 end
186
187 else
188     disp('Failed connecting to remote API server');
189 end % if (clientID > -1)
190
191 disp('Sim ended');
192
193 figure
194 plot(dt,rad2deg(pose(1,:))); hold on;
195 plot(dt,rad2deg(xd(1,:))); hold off;
196 title('pitch')
197
198 figure
199 plot(dt,rad2deg(pose(2,:))); hold on;
200 plot(dt,rad2deg(xd(2,:))); hold off;
201 title('yaw')
202
203 figure
204 plot(dt,u(1,:)); hold on;
205 plot(dt,u(2,:)); hold off;
206 title('Inputs')
207
208
209
210
211

```

```
212 % SAVE OUTPUTS FOR PLOTTING
213 % Save the time, error, input, states and desired states vectors
214 save([testName, '.mat'], 'dt', 'xd', 'tau', 'error', 'pose', 'u', 'k_p', 'k_i', 'k_d'
      );
215 % Notify that the simulation is complete
216 disp('SIMULATION COMPLETE');
```

# Appendix C

## QFLEX 2 USB

### C.1 QFLEX 2 USB Panel (Tutorial)

1. Verify that all of the proper software is installed as mentioned in Section 6.2.
2. Install the QFLEX 2 USB panel to the Quanser AERO as seen in Figure C.1.



Figure C.1: QFLEX 2 USB panel installed on the Quanser AERO.

3. Open the Simulink model that you would like to use.
4. Power on the Quanser AERO, and connect your laptop to the QFLEX 2 USB panel with the provided cable. See Figure C.2.

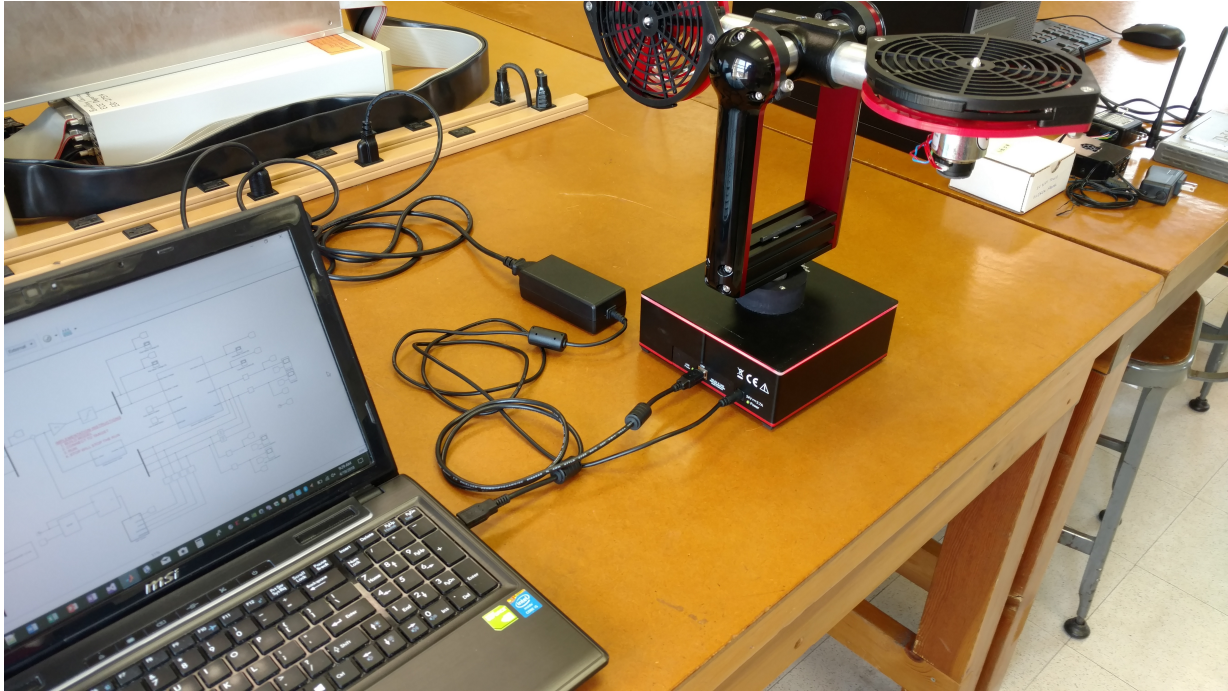


Figure C.2: Connections from your laptop to the Quanser AERO.

5. Build the Simulink model. See Figure C.3.

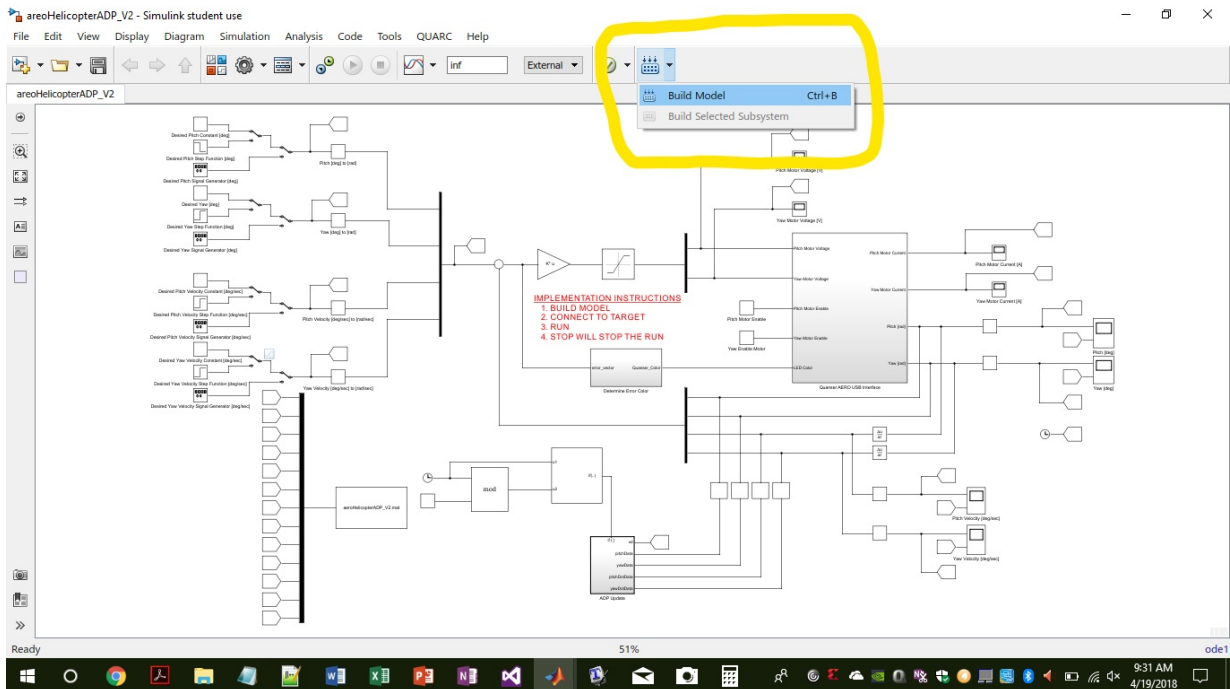


Figure C.3: Build the Simulink model.

6. Connect to the target. This starts the communication process between Simulink and

the Quanser AERO. See Figure C.4.

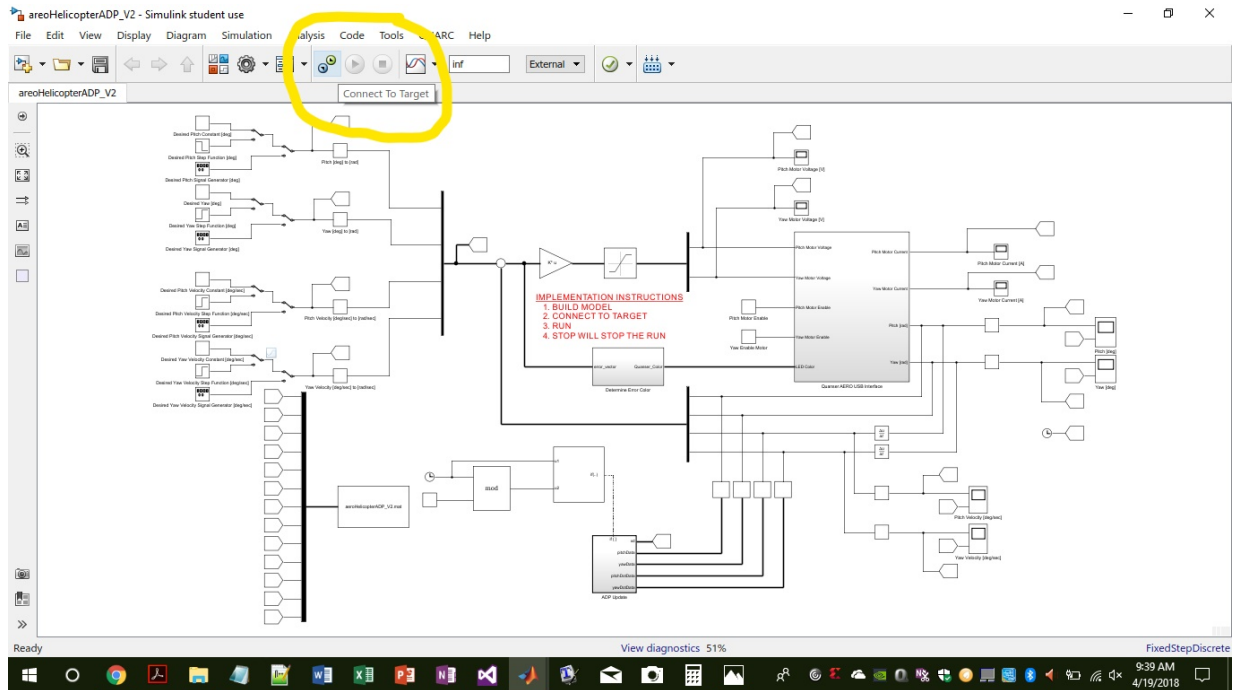


Figure C.4: Connect to the target.

7. Run the Simulink model. See Figure C.5.
8. Observe the Quanser AERO's motion control. See Figure C.6.
9. When you are done with the Quanser AERO, stop the model. See Figure C.7.
10. You may disconnect the Quanser AERO and close Simulink when finished.

#### 11. NOTES

- You should be able to partially adjust the desired positions while the Simulink model is running.
- You really only need to rebuild the model if you make major changes to it which will affect the code generation.
- After you stop the model, you can restart the model, but you have to connect to the target again.
- You will be prompted with errors if you are not physically connected when you try to connect to the target.



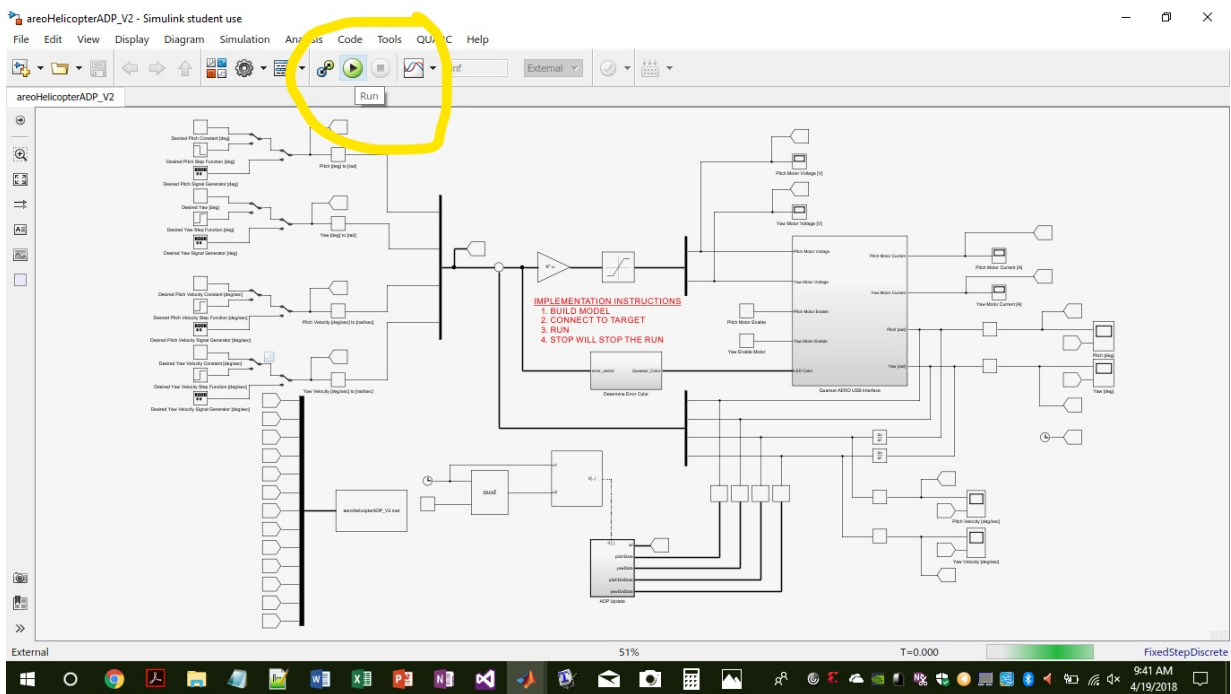


Figure C.5: Run the Simulink model.

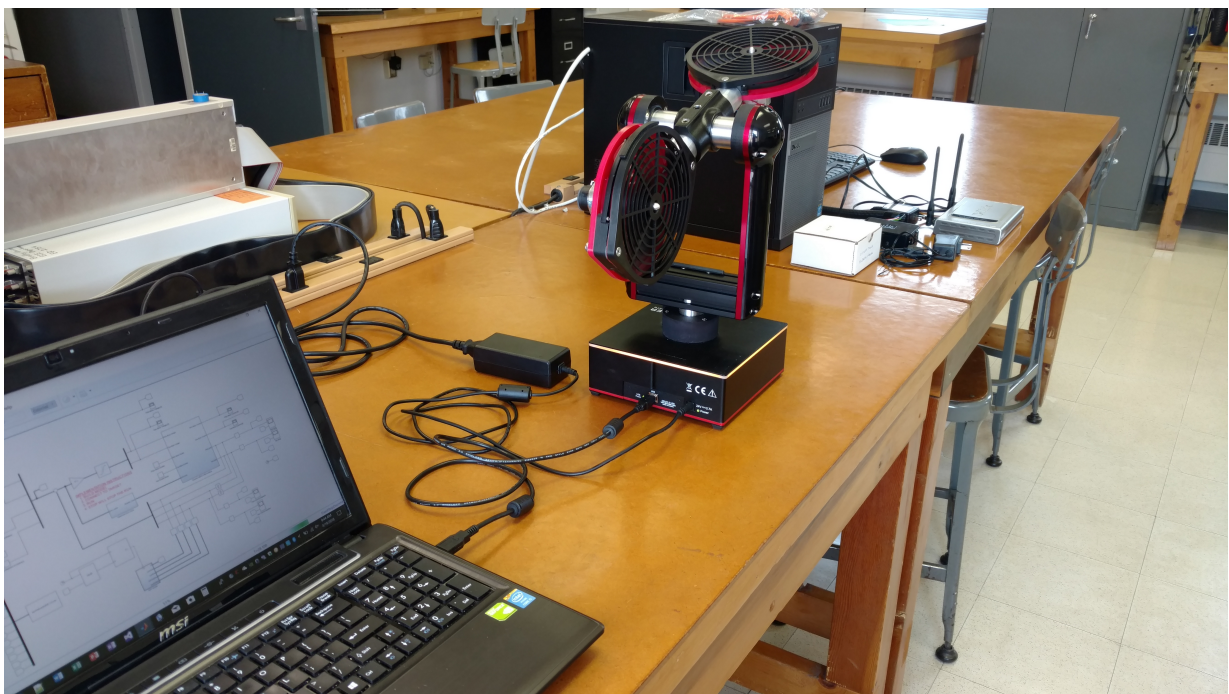


Figure C.6: Observe the Quanser AERO.

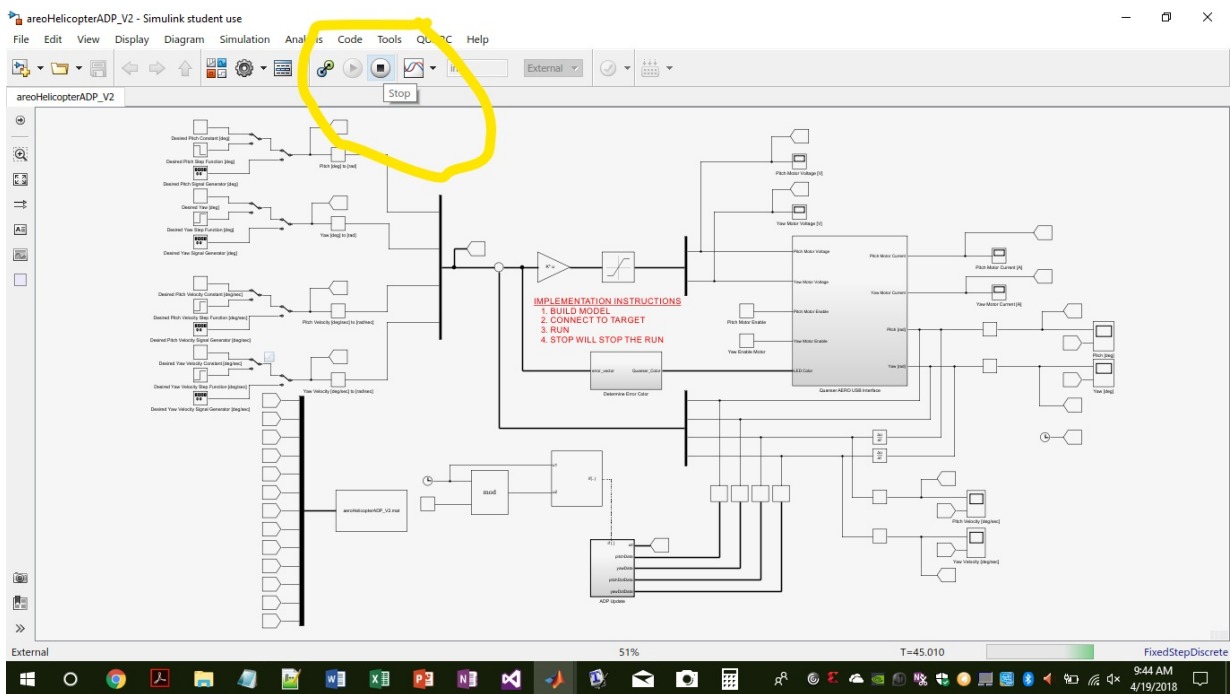


Figure C.7: Stop the Simulink model.

## C.2 Initialization Code (MATLAB)

```

1 % Andrew Fandel
2 % – FIND THE CRITIC NEURAL NETWORK WEIGHTS BEFORE RUNNING THE SIMULATION
3 % USING RANDOM ERROR DATA
4 close all; clear; clc;
5
6 % Sampling time [s]
7 tau = 0.01;
8 % Update time [s]
9 T = 0.2;
10
11 % COST FUNCTION
12 % Q and R matrices used in the cost function
13 Q_Mat_ADP = diag([270 100 1 1]);
14 R_Mat_ADP = 0.005*diag([1 1]);
15
16 % CRITIC WEIGHT TUNING NEURAL NETWORK
17 % This function is specific to the helicopter because of the number of
18 % weights and error model
19 % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
20
21 % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
22 A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
23 B = [0 0; 0 0; -0.0149 0.0414; -0.0751 -0.1295];
24 % UNCOMMENT TO USE A STATE-SPACE MODEL THAT IS NOT THE CORRECT ONE DERIVED
25 % A = [0 0 1 0; 0 0 0 1; 0 0 -0.3307 0; 0 0 0 -0.9283];
26 % B = [0 0; 0 0; 0.1 0.1; -0.3 -0.3];
27 % System dimensions specific for our model
28 [n,~] = size(B);
29
30 % CREATE RANDOM ERROR DATA TO TRAIN THE CRITIC NEURAL NETWORK
31 % Number of equations needed for training, number of sub-intervals
32 % Number of training samples
33 nbar = 100;
34 % Create random error in the range of -pi to pi
35 e_vec = (2*(pi)).*rand(n,nbar) - (pi);
36
37 % ERROR MODEL OF THE HELICOPTER
38 % fbar and gbar — EQ 8
39 % We do not have an xd term because we won't know the desired state vector
40 % when we execute this code
41 % This code can also use anonymous functions because this function is
42 % executed by MATLAB and not Simulink
43 fbar = @(e) A*e;
44 gbar = -B;
45
46 % DISCRETE-TIME ERROR MODEL FOR TIME TAU
47 % f and g — EQ 10
48 f = @(e) fbar(e)*tau + e;
49 g = gbar*tau;
50
51 % COST FUNCTION PARAMETERS

```



```

52 % State penalizing function in the continuous cost function
53 Qbar = @(e) e'*Q_Mat_ADP*e;
54 % Control penalizing matrix in the continuous cost function
55 Rbar = R_Mat_ADP;
56 % The discrete-time cost function will have terms:
57 % Right after EQ 11 in paper
58 % State penalizing function in the discretized cost function
59 Q = @(e) Qbar(e)*tau;
60 % Control penalizing matrix in the discretized cost function
61 R = Rbar*tau;
62
63 % NEURAL NETWORK FUNCTIONS
64 % Critic neural network activation functions
65 rho = @(e) [e(1); e(2); e(3); e(4);...
66             e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
67             e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
68 % Partial derivative of rho with respect to e
69 drhode =@(e) [1, 0, 0, 0;
70               0, 1, 0, 0;
71               0, 0, 1, 0;
72               0, 0, 0, 1;
73               2*e(1), 0, 0, 0;
74               e(2), e(1), 0, 0;
75               e(3), 0, e(1), 0;
76               e(4), 0, 0, e(1);
77               0, 2*e(2), 0, 0;
78               0, e(3), e(2), 0;
79               0, e(4), 0, e(2);
80               0, 0, 2*e(3), 0;
81               0, 0, e(4), e(3);
82               0, 0, 0, 2*e(4)];
83
84 % TOLERANCES
85 % Convergence tolerance for control policy
86 EpsilonPolicy = 0.1;
87 % Convergence tolerance for critic neural network
88 EpsilonWcritic = 0.1;
89
90 % TRAINING PARAMETERS
91 % Number of outer loop iterations
92 outerLoopMax = 700;
93 % Number of inner loop iterations
94 innerLoopMax = 100;
95 % Number of equations needed for training, number of sub-intervals
96 % Number of training samples
97 [~,nbar] = size(e_vec);
98
99 % WEIGHT INITIALIZATION
100 % Initialize the weights of the critic neural network to zero
101 WcLast = zeros(length(rho(e_vec(:,1))),1);
102
103 % LEAST-SQUARES COMPUTATION INITIALIZATION
104 % Matrices required for computing least squares weights of the critic
105 % neural networks — EQ 20

```

```

106 V = zeros(nbar,1);
107 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
108
109 % Matrix to hold the derivative of the error model during policy
110 % updating
111 e_k_plus_1 = zeros(n,nbar);
112
113 % Product of the least squares matrices must be invertible
114 % Logic flag indicating if the critic weights are unsolvable
115 % The weights are unsolvable because the least squares matrices have no
116 % solution — not invertible
117 diverged = 0;
118
119 % OUTER LOOP
120 for i = 1:(outerLoopMax-1)
121 % Determine if the least squares matrices are invertible
122 if diverged == 0
123 % For each of the data collection (discrete time index)
124 for k = 1:nbar
125 % Initialize the optimal inputs to zero
126 uNew = [0; 0];
127 % INNER LOOP
128 for j = 1:(innerLoopMax-1)
129 % Get the updated input value
130 uLast = uNew;
131 % Update the error model
132 e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast;
133 % Compute the new optimal inputs
134 uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))'*WcLast;
135
136 % Check convergence of the optimal inputs
137 if norm(uNew - uLast) < EpsilonPolicy
138 break;
139 end
140 end
141
142 % Update the values for the least-squares computation
143 V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1(:,k))
144 ;
145 Lambda(k,:) = rho(e_vec(:,k))';
146 end
147
148 % Verify the least square solution exists for the critic's weights
149 % If the error data is consistent or there is no error, this will not
150 % hold, so set the weights to zero
151 if det(Lambda'*Lambda) == 0
152 weights = zeros(length(rho(e_vec(:,1))),1);
153 break;
154 end;
155
156 % Calculate least squares solution of critic's weights — EQ 20
157 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
158 % Make sure the weights did not diverge

```

```

159 % If the weights are diverging, just set them to a large number
160 if isnan(WcNew)
161     weights = 1000*ones(length(rho(e_vec(:,1))),1);
162     break;
163 end;
164
165 % Check for convergence of the critic weights
166 if norm(WcNew - WcLast) < EpsilonWcritic
167     weights = WcNew;
168     break;
169 end
170 % If the weights did not converge, repeat the loop
171 WcLast = WcNew;
172
173 % If we reached the last iteration of the loop, just use the last
174 % weights found
175 if (i == (outerLoopMax-1))
176     weights = WcNew;
177 end
178 end
179
180 % Use the weights to determine the P matrix
181 P_Mat = [weights(5) weights(6) weights(7) weights(8);
182         weights(6) weights(9) weights(10) weights(11);
183         weights(7) weights(10) weights(12) weights(13);
184         weights(8) weights(11) weights(13) weights(14)];
185
186 % Find the state-feedback gain EQ
187 K = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
188
189 % Save the initial weights
190 wcInit = weights;
191
192 % Keep only the ADP gain
193 clearvars -except K tau T wcInit;

```

### C.3 Update ADP Gain (MATLAB)

```

1 % CRITIC WEIGHT TUNING NEURAL NETWORK
2 % This function is specific to the helicopter because of the number of
3 % weights and error model
4 % THIS FUNCTION CANNOT HAVE ANY ANONYMOUS FUNCTIONS DUE TO THE C CODE
5 % GENERATION UNSUPPORTING IT
6 function K = quanserAEROCriticTuning(xd, pitchData, yawData, pitchDotData,
7     yawDotData, wcInit)
8     % CREATE THE ERROR VECTOR MATRIX
9     % Use the data from the tapped delay blocks to create the error state
10    % vector matrix
11    e_vec = [pitchData'; yawData'; pitchDotData'; yawDotData'];
12
13    % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
14    % Sampling time [s]
15    tau = 0.01;

```

```

15
16 % COST FUNCTION
17 % Q and R matrices used in the cost function
18 Q_Mat = diag([270 100 1 1]);
19 R_Mat = 0.005*diag([1 1]);
20
21 % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
22 A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
23 B = [0 0; 0 0; -0.0149 0.0414; -0.0751 -0.1295];
24 % UNCOMMENT TO USE A STATE-SPACE MODEL THAT IS NOT THE CORRECT ONE
DERIVED
25 % A = [1 1 1 1; 1 1 1 1; 1 1 1 1; 1 1 1 1];
26 % B = [0 0; 0 0; 0.1 0.1; -0.3 -0.3];
27 % System dimensions specific for our model
28 [n,~] = size(B);
29
30 % ERROR MODEL OF THE HELICOPTER
31 % gbar and hbar — EQ 8
32 % An anonymous function cannot be used for fbar
33 gbar = -B;
34 hbar = -A*xd;
35
36 % DISCRETE-TIME ERROR MODEL FOR TIME TAU
37 % g and h — EQ 10
38 % An anonymous function cannot be used for f
39 g = gbar*tau;
40 h = hbar*tau;
41
42 % COST FUNCTION PARAMETERS
43 % The Q matrix cannot be treated as an anonymous function as in the
44 % MATLAB simulations
45 % Control penalizing matrix in the continuous cost function
46 Rbar = R_Mat;
47 % The discrete-time cost function will have terms:
48 % Right after EQ 11 in paper
49 % Control penalizing matrix in the discretized cost function
50 R = Rbar*tau;
51
52 % NEURAL NETWORK FUNCTIONS
53 % These functions cannot be written as anonymous functions as in the
54 % MATLAB simulations
55
56 % TOLERANCES
57 % Convergence tolerance for control policy
58 EpsilonPolicy = 0.1;
59 % Convergence tolerance for critic neural network
60 EpsilonWcritic = 0.1;
61
62 % TRAINING PARAMETERS
63 % Number of outer loop iterations
64 outerLoopMax = 700;
65 % Number of inner loop iterations
66 innerLoopMax = 100;
67 % Number of equations needed for training, number of sub-intervals

```

```

68 % Number of training samples
69 [~,nbar] = size(e_vec);
70
71 % WEIGHT INITIALIZATION
72 % Initialize the weights of the critic neural network to zero
73 % Number of rows hard-coded for the helicopter
74 WcLast = zeros(14,1);
75
76 % LEAST-SQUARES COMPUTATION INITIALIZATION
77 % Matrices required for computing least squares weights of the critic
78 % neural networks — EQ 20
79 V = zeros(nbar,1);
80 Lambda = zeros(nbar,14);
81
82 % Matrix to hold the derivative of the error model during policy
83 % updating
84 e_k_plus_1 = zeros(n,nbar);
85
86 % Product of the least squares matrices must be invertible
87 % Logic flag indicating if the critic weights are unsolvable
88 % The weights are unsolvable because the least squares matrices have no
89 % solution — not invertible
90 diverged = 0;
91
92 % OUTER LOOP
93 for i = 1:(outerLoopMax-1)
94 % Determine if the least squares matrices are invertible
95 if diverged == 0
96 % For each of the data collection (discrete time index)
97 for k = 1:nbar
98 % Initialize the optimal inputs to zero
99 uNew = [0; 0];
100 % INNER LOOP
101 for j = 1:(innerLoopMax-1)
102 % Get the updated input value
103 uLast = uNew;
104 % Update the error model
105 % Because of no anonymous functions we have modified f
106 % fbar = @(e) A*e;
107 % f = @(e) fbar(e)*tau + e;
108 e_k_plus_1(:,k) = (A*e_vec(:,k)*tau) + e_vec(:,k) + g*uLast
+ h;
109
110 % Compute the new optimal inputs
111 % Partial derivative of rho with respect to e
112 drhode = [1, 0, 0, 0;
113           0, 1, 0, 0;
114           0, 0, 1, 0;
115           0, 0, 0, 1;
116           2*e_vec(1,k), 0, 0, 0;
117           e_vec(2,k), e_vec(1,k), 0, 0;
118           e_vec(3,k), 0, e_vec(1,k), 0;
119           e_vec(4,k), 0, 0, e_vec(1,k);
120           0, 2*e_vec(2,k), 0, 0;
121           0, e_vec(3,k), e_vec(2,k), 0;

```

```

121         0, e_vec(4,k), 0, e_vec(2,k);
122         0, 0, 2*e_vec(3,k), 0;
123         0, 0, e_vec(4,k), e_vec(3,k);
124         0, 0, 0, 2*e_vec(4,k)];
125     uNew = -0.5*(R^(-1))*g'*drhode'*WcLast;
126
127     % Check convergence of the optimal inputs
128     if norm(uNew - uLast) < EpsilonPolicy
129         break;
130     end
131 end
132
133 % Update the values for the least-squares computation
134 % Critic neural network activation functions
135 rhoE = [e_vec(1,k); e_vec(2,k); e_vec(3,k); e_vec(4,k);...
136         e_vec(1,k)^2; e_vec(1,k)*e_vec(2,k);...
137         e_vec(1,k)*e_vec(3,k); e_vec(1,k)*e_vec(4,k);...
138         e_vec(2,k)^2; e_vec(2,k)*e_vec(3,k);...
139         e_vec(2,k)*e_vec(4,k); e_vec(3,k)^2;...
140         e_vec(3,k)*e_vec(4,k); e_vec(4,k)^2];
141 rhoEK = [e_k_plus_1(1,k); e_k_plus_1(2,k); e_k_plus_1(3,k);...
142         e_k_plus_1(4,k); e_k_plus_1(1,k)^2;...
143         e_k_plus_1(1,k)*e_k_plus_1(2,k);...
144         e_k_plus_1(1,k)*e_k_plus_1(3,k);...
145         e_k_plus_1(1,k)*e_k_plus_1(4,k);...
146         e_k_plus_1(2,k)^2; e_k_plus_1(2,k)*e_k_plus_1(3,k);...
147         e_k_plus_1(2,k)*e_k_plus_1(4,k); e_k_plus_1(3,k)^2;...
148         e_k_plus_1(3,k)*e_k_plus_1(4,k); e_k_plus_1(4,k)^2];
149 % State penalizing function in the continuous cost function
150 % Qbar = @(e) e'*Q_Mat*e;
151 % State penalizing function in the discretized cost function
152 % Q = @(e) Qbar(e)*tau;
153 V(k,:) = (e_vec(:,k))*Q_Mat*e_vec(:,k)*tau + uNew'*R*uNew +
WcLast'*rhoEK;
154     Lambda(k,:) = rhoE';
155 end
156 end
157
158 % Verify the least square solution exists for the critic's weights
159 % If the error data is consistent or there is no error, this will not
160 % hold, so set the weights to what they were initially before the
161 % simulation
162 if det(Lambda'*Lambda) == 0
163     weights = wcInit;
164     break;
165 end;
166
167 % Calculcate least squares solution of critic's weights — EQ 20
168 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
169 % Make sure the weights did not diverge
170 % If the weights diverged, set the weights to what they were initially
171 % before the simulation
172 if isnan(WcNew)
173     weights = wcInit;

```

```

174     break;
175     % Check for convergence of the critic weights
176     elseif norm(WcNew - WcLast) < EpsilonWcritic
177         weights = WcNew;
178         break;
179     % If we reach the last iteration of the loop, just use the last weights
180     elseif (i == (outerLoopMax-1))
181         weights = WcNew;
182     % If all else fails, just set the weights to the initial weights
183     else
184         weights = wcInit;
185     end
186     % If the weights did not converge, do another iteration of the loop
187     WcLast = WcNew;
188     end
189
190     % Use the weights to determine the P matrix
191     P_Mat = [weights(5) weights(6) weights(7) weights(8);
192             weights(6) weights(9) weights(10) weights(11);
193             weights(7) weights(10) weights(12) weights(13);
194             weights(8) weights(11) weights(13) weights(14)];
195
196     % Find the state-feedback gain EQ
197     K = 0.5*(R_Mat^-1)*B'*P_Mat;
198 end

```

## C.4 Base Color (MATLAB)

```

1 % THIS FUNCTION DETERMINES THE PROPER COLOR MATRIX FOR THE QUANSER AERO
2 function colorSelect = colorSelector(u)
3
4 % If the magnitude is less than 1, output the color green
5 if (u < 1)
6     colorSelect = [0 1 0];
7 % If the magnitude is between 3 and 10, output the color yellow
8 elseif (u < 10)
9     colorSelect = [1 1 0];
10 % If the magnitude is greater than 10, output the color blue
11 else
12     colorSelect = [0 0 1];
13 end

```



# Appendix D

## Raspberry Pi 3

### D.1 Raspberry Pi 3 (Tutorial)

1. Verify that all of the proper software is installed as mentioned in Section 6.3.
2. Install the QFLEX 2 Embedded panel to the Quanser AERO as seen in Figure D.1.



Figure D.1: QFLEX 2 Embedded panel installed on the Quanser AERO.

3. Connect your laptop to the Raspberry Pi and power it on. Use an Ethernet cable to connect to the Raspberry Pi. See Figure D.2. You can connect the Raspberry Pi to the Quanser AERO, but I prefer to do that in a later step.



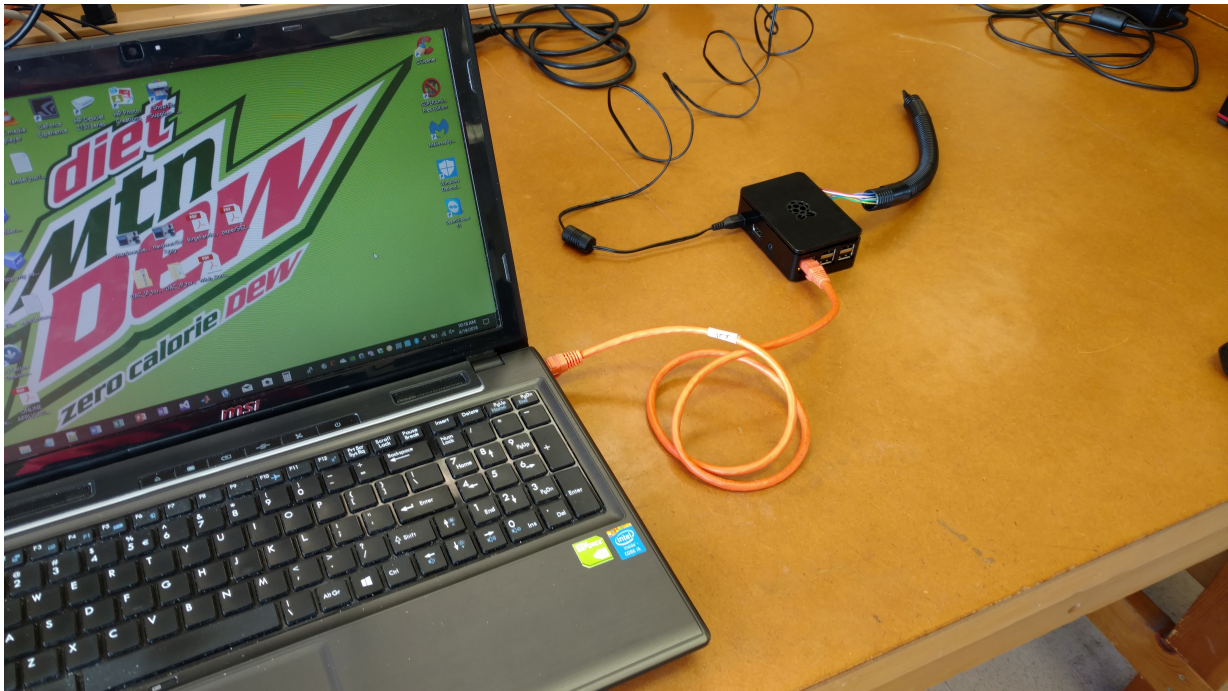


Figure D.2: Connect the Raspberry Pi to your laptop and power it on.

4. Wait a few moments and then connect to the Raspberry Pi using PuTTY. The IP address of our particular Raspberry Pi is 169.254.0.2. See Figure D.3.

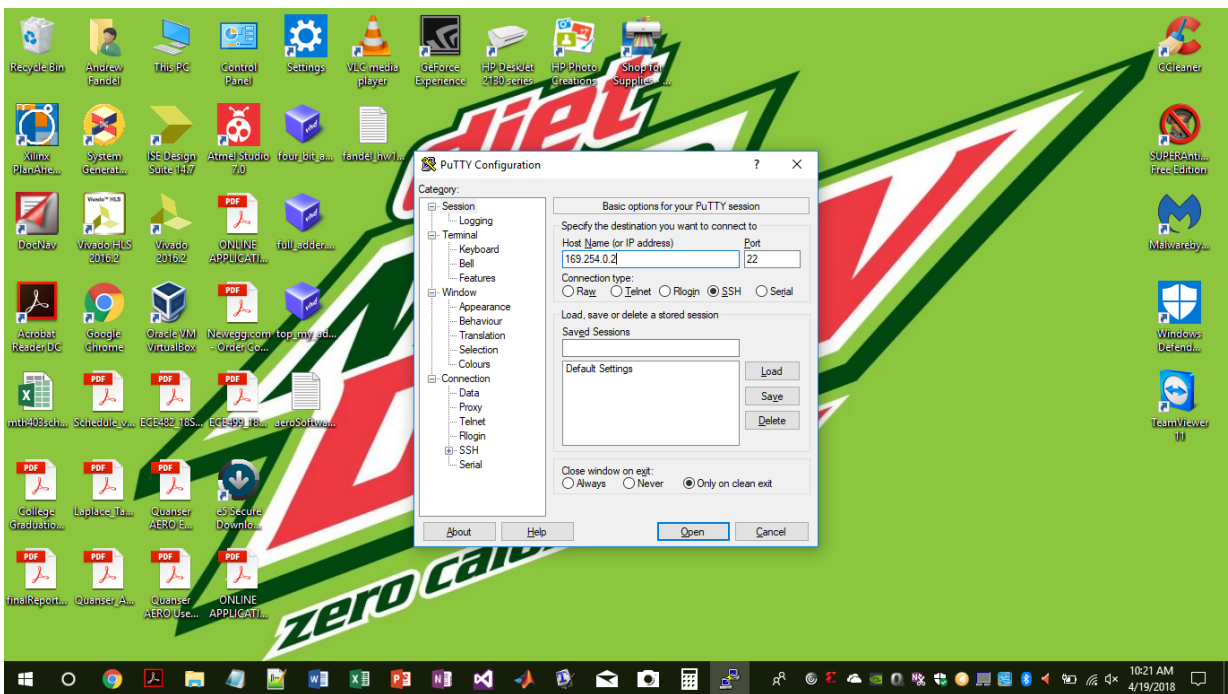
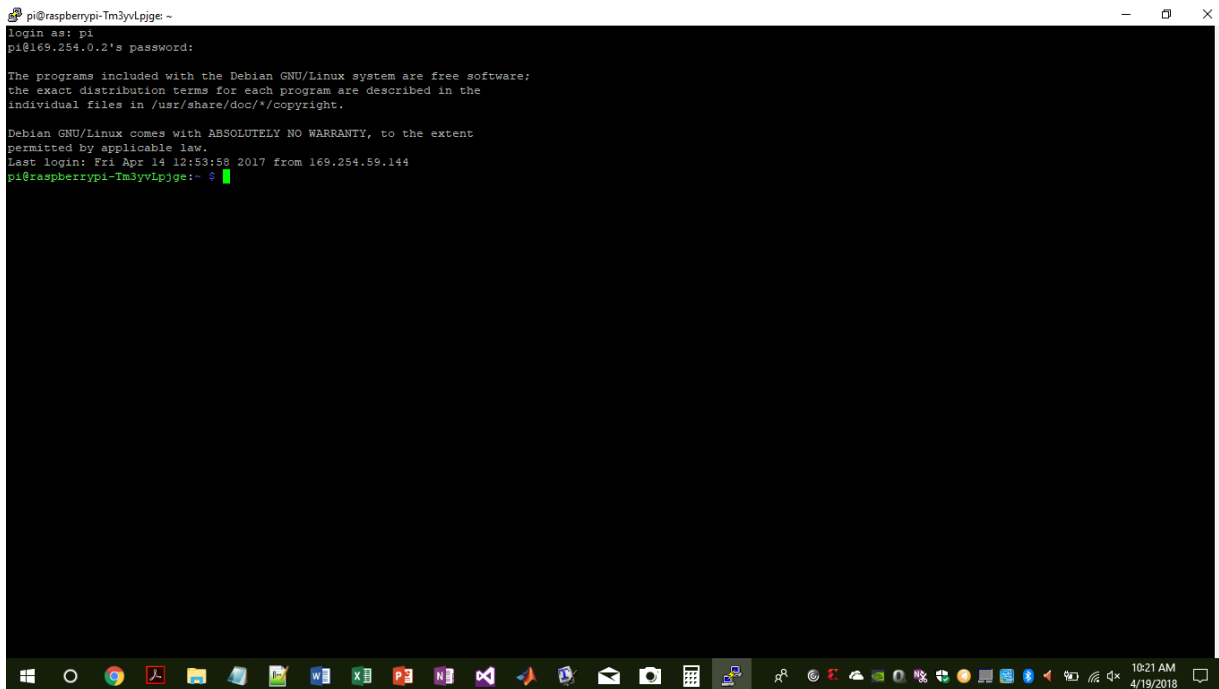


Figure D.3: Connect to the Raspberry Pi using PuTTY.

5. Once PuTTY is connected to the Raspberry Pi, login to it. Login as "pi" with the password "raspberrypi." See Figure D.4.



```
pi@raspberrypi-Tm3yvLpJge: ~
login as: pi
pi@169.254.0.2's password:
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Apr 14 12:53:58 2017 from 169.254.59.144
pi@raspberrypi-Tm3yvLpJge: ~
```

Figure D.4: Login to the Raspberry Pi.

6. We can see that this is the Linux version we installed on the SD card when configuring the support package. See Figure D.5. We can see where the code that will be generated by Simulink will be located.
7. Open the Simulink model that you would like to push to the Raspberry Pi.
8. Deploy the model to hardware. This will generate the C-code on the Raspberry Pi. See Figure D.6. This build may take a while.
9. **NOTE** - The Simulink model provided uses an initialization function to enable the SPI communication pins. If you just powered on the Raspberry Pi, you need to uncomment the line that disables SPI. If have already pushed to the Raspberry Pi while it has been on and want to do it again, you need to comment out the line that disables SPI. If you don't do this, there will be errors aborting the build.
10. Once the model is pushed to the Raspberry Pi, the model will actually begin running as a process.
11. Use the `ps -A` Linux command to see all the processes running. One should have a similar name of the model; find its process number.
12. Once you have the process number, use the `sudo kill -9 1234` Linux command to kill the process. 1234 is the process number you currently found.

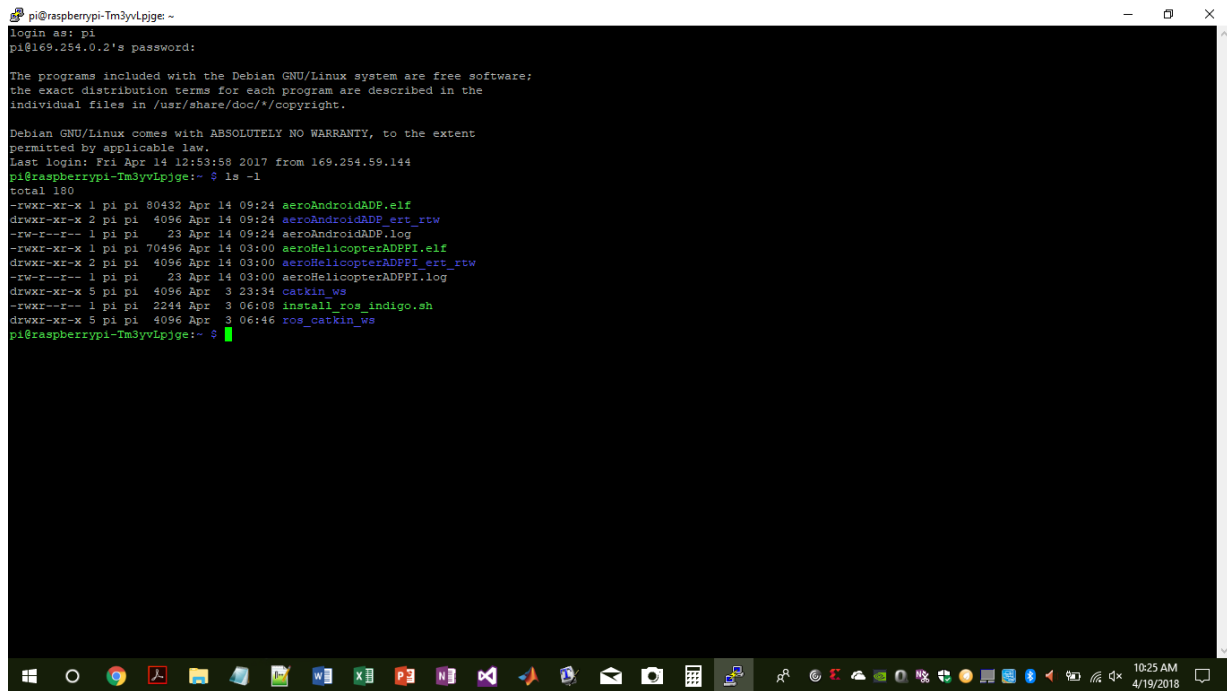


Figure D.5: Version of Linux installed on SD card by MATLAB.

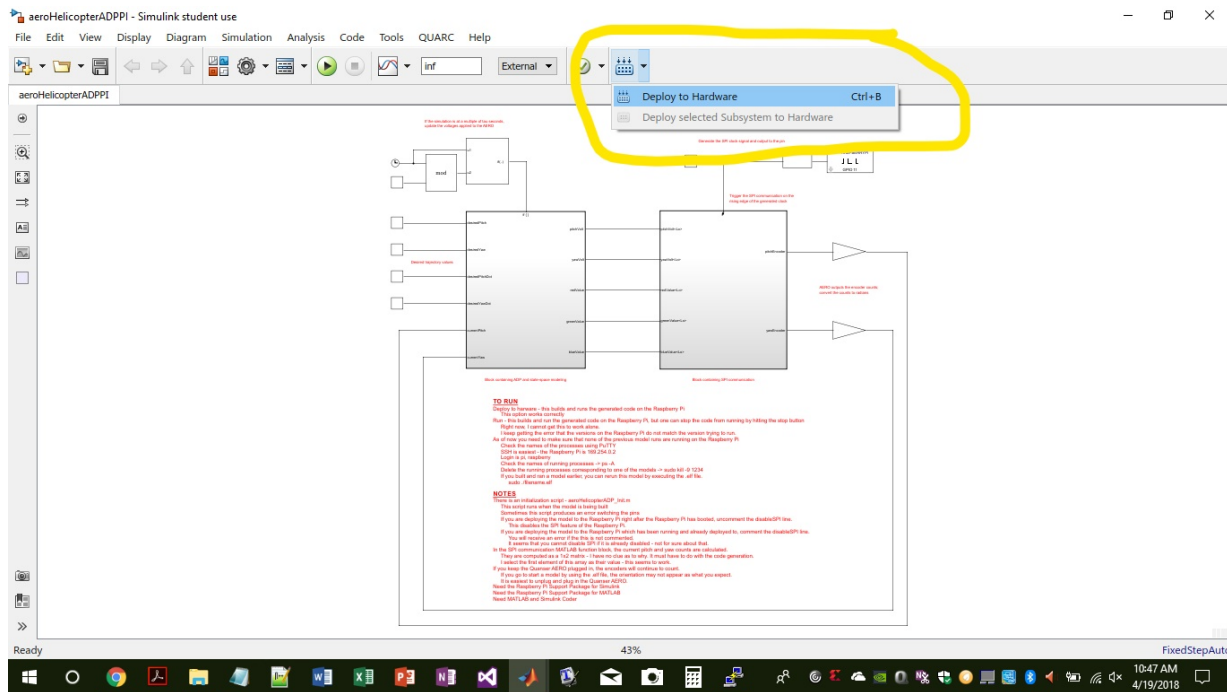


Figure D.6: Deploy model to hardware.

13. You now have the generated code on the Raspberry Pi, but it is not currently running.
14. Connect the Raspberry Pi to the Quanser AERO. See Figure D.7. The SPI wires

should already be connected to the Raspberry Pi. If not, you will need to consult the MATLAB initialization code and the Raspberry Pi data sheet.

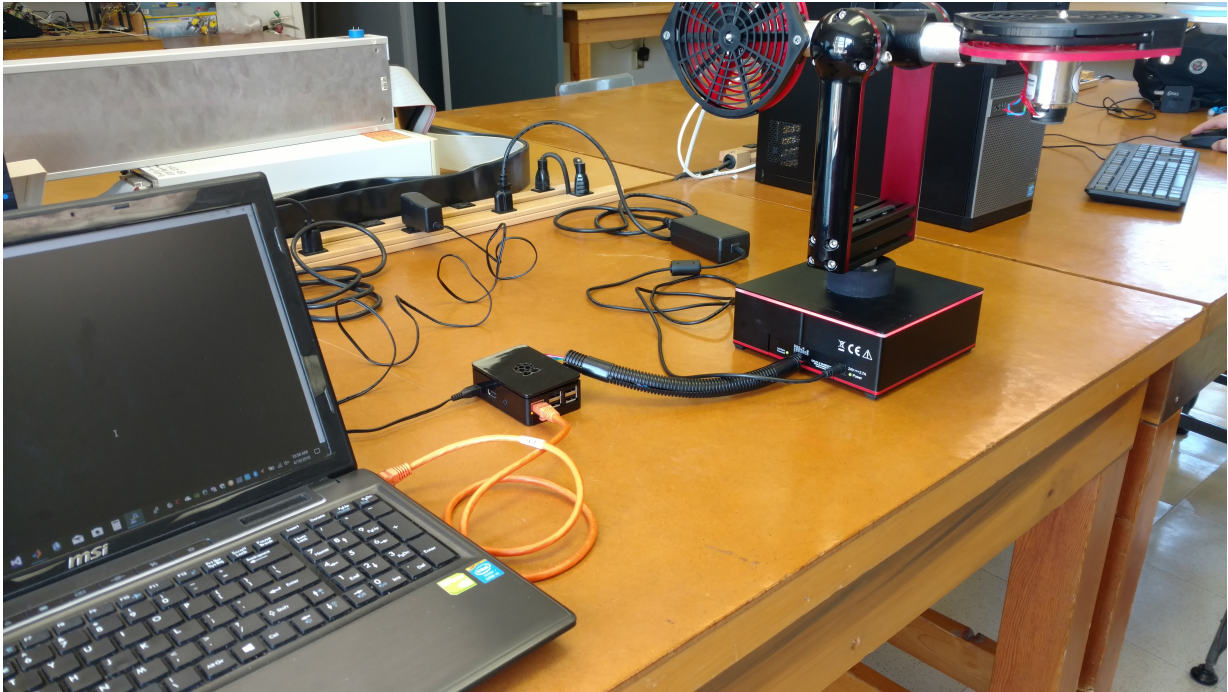


Figure D.7: Connect the Raspberry Pi to the Quanser AERO.

15. Execute the .elf file to run the generated code. Use the Linux command `sudo ./file.elf`. See Figure [D.8](#).
16. The generated C-code should now be running on the Raspberry Pi.
17. When you are done running the code, repeat the steps used to kill the process. Sometimes you can also use Control-C to stop the running model.



```

pi@raspberrypi-Tm3yvlpjge: ~
login as: pi
pi@169.254.0.2's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Apr 14 12:53:58 2017 from 169.254.59.144
pi@raspberrypi-Tm3yvlpjge:~$ ls -l
total 180
-rwxr-xr-x 1 pi pi 80432 Apr 14 09:24 aeroAndroidADP.elf
drwxr-xr-x 2 pi pi 4096 Apr 14 09:24 aeroAndroidADP_ext_rtw
-rw-r--r-- 1 pi pi 23 Apr 14 09:24 aeroAndroidADP.log
-rwxr-xr-x 1 pi pi 70496 Apr 14 03:00 aeroHelicopterADPPI.elf
drwxr-xr-x 2 pi pi 4096 Apr 14 03:00 aeroHelicopterADPPI_ert_rtw
-rw-r--r-- 1 pi pi 23 Apr 14 03:00 aeroHelicopterADPPI.log
drwxr-xr-x 5 pi pi 4096 Apr 3 23:34 catkin_ws
-rwxr-xr-x 1 pi pi 2244 Apr 3 06:09 install_ros_indigo.sh
drwxr-xr-x 5 pi pi 4096 Apr 3 06:46 ros_catkin_ws
pi@raspberrypi-Tm3yvlpjge:~$ sudo ./aeroHelicopterADPPI.elf

```

Figure D.8: Execute the code on the Raspberry Pi.

## D.2 Initialization Code (MATLAB)

```

1 % Andrew Fandel
2 % - FIND THE CRITIC NEURAL NETWORK WEIGHTS BEFORE RUNNING THE SIMULATION
3 % USING RANDOM ERROR DATA
4 close all; clear all; clc;
5
6 % Sampling time [s]
7 tau = 0.05;
8 % Update time [s]
9 T = 1;
10
11 % COST FUNCTION
12 % Q and R matrices used in the cost function
13 Q.Mat_ADP = diag([270 100 1 1]);
14 R.Mat_ADP = 0.005*diag([1 1]);
15
16 % CRITIC WEIGHT TUNING NEURAL NETWORK
17 % This function is specific to the helicopter because of the number of
18 % weights and error model
19 % REFERENCE DR. MIAH'S PAPER FOR EQUATION NUMBERS
20
21 % SYSTEM PARAMETERS SPECIFIC TO THE 2-DOF QUANSER AERO
22 A = [0 0 1 0; 0 0 0 1; -1.7442 0 -0.3307 0; 0 0 0 -0.9283];
23 B = [0 0; 0 0; -0.0149 0.0414; -0.0751 -0.1295];
24 % System dimensions specific for our model
25 [n, ~] = size(B);
26

```

```

27 % CREATE RANDOM ERROR DATA TO TRAIN THE CRITIC NEURAL NETWORK
28 % Number of equations needed for training , number of sub-intervals
29 % Number of training samples
30 nbar = 100;
31 % Create random error in the range of -pi to pi
32 e_vec = (2*(pi)).*rand(n,nbar) - (pi);
33
34 % ERROR MODEL OF THE HELICOPTER
35 % fbar and gbar — EQ 8
36 % We do not have an xd term because we won't know the desired state vector
37 % when we execute this code
38 % This code can also use anonymous functions because this function is
39 % executed by MATLAB and not Simulink
40 fbar = @(e) A*e;
41 gbar = -B;
42
43 % DISCRETE-TIME ERROR MODEL FOR TIME TAU
44 % f and g — EQ 10
45 f = @(e) fbar(e)*tau + e;
46 g = gbar*tau;
47
48 % COST FUNCTION PARAMETERS
49 % State penalizing function in the continuous cost function
50 Qbar = @(e) e'*Q_Mat_ADP*e;
51 % Control penalizing matrix in the continuous cost function
52 Rbar = R_Mat_ADP;
53 % The discrete-time cost function will have terms:
54 % Right after EQ 11 in paper
55 % State penalizing function in the discretized cost function
56 Q = @(e) Qbar(e)*tau;
57 % Control penalizing matrix in the discretized cost function
58 R = Rbar*tau;
59
60 % NEURAL NETWORK FUNCTIONS
61 % Critic neural network activation functions
62 rho = @(e) [e(1); e(2); e(3); e(4);...
63             e(1)^2; e(1)*e(2); e(1)*e(3); e(1)*e(4);...
64             e(2)^2; e(2)*e(3); e(2)*e(4); e(3)^2; e(3)*e(4); e(4)^2];
65 % Partial derivative of rho with respect to e
66 drhode = @(e) [1, 0, 0, 0;
67                0, 1, 0, 0;
68                0, 0, 1, 0;
69                0, 0, 0, 1;
70                2*e(1), 0, 0, 0;
71                e(2), e(1), 0, 0;
72                e(3), 0, e(1), 0;
73                e(4), 0, 0, e(1);
74                0, 2*e(2), 0, 0;
75                0, e(3), e(2), 0;
76                0, e(4), 0, e(2);
77                0, 0, 2*e(3), 0;
78                0, 0, e(4), e(3);
79                0, 0, 0, 2*e(4)];
80

```

```

81 % TOLERANCES
82 % Convergence tolerance for control policy
83 EpsilonPolicy = 0.1;
84 % Convergence tolerance for critic neural network
85 EpsilonWcritic = 0.1;
86
87 % TRAINING PARAMETERS
88 % Number of outer loop iterations
89 outerLoopMax = 700;
90 % Number of inner loop iterations
91 innerLoopMax = 100;
92 % Number of equations needed for training, number of sub-intervals
93 % Number of training samples
94 [n,nbar] = size(e_vec);
95
96 % WEIGHT INITIALIZATION
97 % Initialize the weights of the critic neural network to zero
98 WcLast = zeros(length(rho(e_vec(:,1))),1);
99
100 % LEAST-SQUARES COMPUTATION INITIALIZATION
101 % Matrices required for computing least squares weights of the critic
102 % neural networks — EQ 20
103 V = zeros(nbar,1);
104 Lambda = zeros(nbar,length(rho(e_vec(:,1))));
105
106 % Matrix to hold the derivative of the error model during policy
107 % updating
108 e_k_plus_1 = zeros(n,nbar);
109
110 % Product of the least squares matrices must be invertible
111 % Logic flag indicating if the critic weights are unsolvable
112 % The weights are unsolvable because the least squares matrices have no
113 % solution — not invertible
114 diverged = 0;
115
116 % OUTER LOOP
117 for i = 1:(outerLoopMax-1)
118 % Determine if the least squares matrices are invertible
119 if diverged == 0
120 % For each of the data collection (discrete time index)
121 for k = 1:nbar
122 % Initialize the optimal inputs to zero
123 uNew = [0; 0];
124 % INNER LOOP
125 for j = 1:(innerLoopMax-1)
126 % Get the updated input value
127 uLast = uNew;
128 % Update the error model
129 e_k_plus_1(:,k) = f(e_vec(:,k)) + g*uLast;
130 % Compute the new optimal inputs
131 uNew = -0.5*(R^(-1))*g'*drhode(e_k_plus_1(:,k))*WcLast;
132
133 % Check convergence of the optimal inputs
134 if norm(uNew - uLast) < EpsilonPolicy

```

```

135         break;
136     end
137 end
138
139 % Update the values for the least-squares computation
140 V(k,:) = Q(e_vec(:,k)) + uNew'*R*uNew + WcLast'*rho(e_k_plus_1(:,k))
141 ;
142     Lambda(k,:) = rho(e_vec(:,k))';
143 end
144
145 % Verify the least square solution exists for the critic's weights
146 % If the error data is consistent or there is no error, this will not
147 % hold, so set the weights to zero
148 if det(Lambda'*Lambda) == 0
149     weights = zeros(length(rho(e_vec(:,1))),1);
150     break;
151 end
152
153 % Calculate least squares solution of critic's weights — EQ 20
154 WcNew = (Lambda'*Lambda)^(-1)*Lambda'*V;
155 % Make sure the weights did not diverge
156 % If the weights are diverging, just set them to a large number
157 if isnan(WcNew)
158     weights = 1000*ones(length(rho(e_vec(:,1))),1);
159     break;
160 end
161
162 % Check for convergence of the critic weights
163 if norm(WcNew - WcLast) < EpsilonWcritic
164     weights = WcNew;
165     break;
166 end
167 % If the weights did not converge, repeat the loop
168 WcLast = WcNew;
169
170 % If we reached the last iteration of the loop, just use the last
171 % weights found
172 if (i == (outerLoopMax-1))
173     weights = WcNew;
174 end
175 end
176
177 % Use the weights to determine the P matrix
178 P_Mat = [weights(5) weights(6) weights(7) weights(8);
179          weights(6) weights(9) weights(10) weights(11);
180          weights(7) weights(10) weights(12) weights(13);
181          weights(8) weights(11) weights(13) weights(14)];
182
183 % Find the state-feedback gain EQ
184 K = 0.5*(R_Mat_ADP^-1)*B'*P_Mat;
185
186 % Save the initial weights
187 wcInit = weights;

```



```

188
189 % After much trial and error, it seems that when SPI is initiated, the
190 % channel remains high and there is no clock signal
191 % I will create a clock signal and issue the slave select
192 % Create a connection between MATLAB and the Raspberry Pi
193 aeroPi = raspi;
194 % Make sure that SPI is disable on the Raspberry Pi so we can use those
195 % same GPIO pins
196 % You may have to comment out this line if you are rebuilding a model and
197 % the SPI is already disabled
198 %disableSPI(aeroPi);
199 % Configure the pins for either inputs or outputs
200 % MOSI – output
201 configurePin(aeroPi,10,'DigitalOutput');
202 % MISO – input
203 configurePin(aeroPi,9,'DigitalInput');
204 % SPI Clock – output
205 configurePin(aeroPi,11,'DigitalOutput');
206 % SS – output
207 configurePin(aeroPi,8,'DigitalOutput');
208 % Specify the period of the SPI clockoutput
209 % 500 kHz = 2 us
210 spiPeriod = 0.0001;
211
212 % Keep only the variables pertaining to what we will need in the workspace
213 clearvars –except K tau T wcInit aeroPi spiPeriod;

```

## D.3 SPI Communication (MATLAB)

```

1 % THIS FUNCTION CONTROLS THE SPI INTERFACING
2 % THIS FUNCTION DETERMINES WHAT DATA IS SENT TO THE QUANSER AERO AND WHAT
3 % DATA IS RETRIEVED FROM THE QUANSER AERO
4 function [MOSI,SS,pitchEncoder,yawEncoder,byteNumber,bitNumber,
    encoder2_23_16,encoder2_15_8,encoder2_7_0,encoder3_23_16,encoder3_15_8,
    encoder3_7_0] = fcn(MISO, pitchVolt,yawVolt,redValue,greenValue,blueValue
    ,byteNumber,bitNumber,encoder2_23_16,encoder2_15_8,encoder2_7_0,
    encoder3_23_16,encoder3_15_8,encoder3_7_0)
5 % Variables used in the function
6 complement = '0000000000000000';
7 pitchEncoderBin = '0000000000000000000000000000';
8 yawEncoderBin = '0000000000000000000000000000';
9 tempBin = '00000000000000000000000000000000';
10 tempVoltBin = '00000000';
11
12 % Output the value of the pitch encoder using the latest byte values
13 % Combine all of the encoder bytes
14 pitchEncoderIn = [encoder2_23_16 encoder2_15_8 encoder2_7_0];
15 % Convert the encoder binary vector into a character array
16 for i = 1:24
17     if (pitchEncoderIn(i) == 1)
18         pitchEncoderBin(i) = '1';
19     else
20         pitchEncoderBin(i) = '0';

```

```
21     end
22 end
23 % Calculate the 2's complement value of the encoder
24 if(pitchEncoderBin(1) == '1')
25     for i = 1:32
26         if (i < 9)
27             tempBin(i) = '1';
28         else
29             tempBin(i) = pitchEncoderBin(i - 8);
30         end
31     end
32 else
33     for i = 1:32
34         if (i < 9)
35             tempBin(i) = '0';
36         else
37             tempBin(i) = pitchEncoderBin(i - 8);
38         end
39     end
40 end
41 pitchEncoderTemp = typecast(uint32(bin2dec(tempBin)), 'int32');
42 pitchEncoderOut = cast(pitchEncoderTemp, 'double');
43 pitchEncoder = pitchEncoderOut(1);
44 % pitchEncoder = cast(pitchEncoderTemp, 'double');
45
46 % Output the value of the yaw encoder using the latest byte values
47 % Combine all of the encoder bytes
48 yawEncoderIn = [encoder3_23_16 encoder3_15_8 encoder3_7_0];
49 % Convert the encoder binary vector into a character array
50 for i = 1:24
51     if (yawEncoderIn(i) == 1)
52         yawEncoderBin(i) = '1';
53     else
54         yawEncoderBin(i) = '0';
55     end
56 end
57 % Calculate the 2's complement value of the encoder
58 if(yawEncoderBin(1) == '1')
59     for i = 1:32
60         if (i < 9)
61             tempBin(i) = '1';
62         else
63             tempBin(i) = yawEncoderBin(i - 8);
64         end
65     end
66 else
67     for i = 1:32
68         if (i < 9)
69             tempBin(i) = '0';
70         else
71             tempBin(i) = yawEncoderBin(i - 8);
72         end
73     end
74 end
```

```

75 yawEncoderTemp = typecast(uint32(bin2dec(tempBin)), 'int32');
76 yawEncoderOut = cast(yawEncoderTemp, 'double');
77 yawEncoder = yawEncoderOut(1);
78 % yawEncoder = cast(yawEncoderTemp, 'double');
79
80 % Determine which byte we are currently sending
81 % Specify the slave select value
82 % - 0 to activate the SPI on the Quanser AERO
83 % - 1 to de-activate the SPI on the Quanser AERO
84 % Determine the what the byte to be transmitted should be
85 switch byteNumber
86     case 0
87         % ** START OF BASE PACKET **
88         % BYTE 0
89         % MOSI DATA - BASE MODE (0X01)
90         % MISO DATA - BASE ID MSB
91         % Beginning of the transmission, so SS goes low
92         SS = 0;
93         readyMOSI = dec2bin(1,8);
94
95     case 1
96         % BYTE 1
97         % MOSI DATA - PADDING BYTE (0X00)
98         % MISO DATA - BASE ID LSB
99         SS = 0;
100        readyMOSI = dec2bin(0,8);
101
102    case 2
103        % BYTE 2
104        % MOSI DATA - BASE WRITE MASK
105        % MISO DATA - ENCODER 2 (23-16)
106        SS = 0;
107        % Enable the overwriting of encoders 2 and 3 and the LED colors
108        % Bit 4 - Set encoder 3 enable
109        % Bit 3 - Set encoder 2 enable
110        % Bit 2 - Write blue LED
111        % Bit 1 - Write green LED
112        % Bit 0 - Write red LED
113        % Don't want to overwrite the encoder values
114        readyMOSI = dec2bin(7,8);
115        % Receive the MISO byte bit by bit and update the vector holding the
116        % byte
117        switch bitNumber
118            case 1
119                if (MISO == true)
120                    encoder2_23_16(1) = 1;
121                else
122                    encoder2_23_16(1) = 0;
123                end
124            case 2
125                if (MISO == true)
126                    encoder2_23_16(2) = 1;
127                else
128                    encoder2_23_16(2) = 0;

```

```
129         end
130     case 3
131         if (MISO == true)
132             encoder2_23_16(3) = 1;
133         else
134             encoder2_23_16(3) = 0;
135         end
136     case 4
137         if (MISO == true)
138             encoder2_23_16(4) = 1;
139         else
140             encoder2_23_16(4) = 0;
141         end
142     case 5
143         if (MISO == true)
144             encoder2_23_16(5) = 1;
145         else
146             encoder2_23_16(5) = 0;
147         end
148     case 6
149         if (MISO == true)
150             encoder2_23_16(6) = 1;
151         else
152             encoder2_23_16(6) = 0;
153         end
154     case 7
155         if (MISO == true)
156             encoder2_23_16(7) = 1;
157         else
158             encoder2_23_16(7) = 0;
159         end
160     case 8
161         if (MISO == true)
162             encoder2_23_16(8) = 1;
163         else
164             encoder2_23_16(8) = 0;
165         end
166     end
167
168     case 3
169     % BYTE 3
170     % MOSI DATA – RED LED MSB
171     % MISO DATA – ENCODER 2 (15–8)
172     SS = 0;
173     readyMOSI = dec2bin(redValue(1),8);
174     % Receive the MISO byte bit by bit and update the vector holding the
175     % byte
176     switch bitNumber
177     case 1
178         if (MISO == true)
179             encoder2_15_8(1) = 1;
180         else
181             encoder2_15_8(1) = 0;
182         end
```

```
183     case 2
184         if (MISO == true)
185             encoder2_15_8(2) = 1;
186         else
187             encoder2_15_8(2) = 0;
188         end
189     case 3
190         if (MISO == true)
191             encoder2_15_8(3) = 1;
192         else
193             encoder2_15_8(3) = 0;
194         end
195     case 4
196         if (MISO == true)
197             encoder2_15_8(4) = 1;
198         else
199             encoder2_15_8(4) = 0;
200         end
201     case 5
202         if (MISO == true)
203             encoder2_15_8(5) = 1;
204         else
205             encoder2_15_8(5) = 0;
206         end
207     case 6
208         if (MISO == true)
209             encoder2_15_8(6) = 1;
210         else
211             encoder2_15_8(6) = 0;
212         end
213     case 7
214         if (MISO == true)
215             encoder2_15_8(7) = 1;
216         else
217             encoder2_15_8(7) = 0;
218         end
219     case 8
220         if (MISO == true)
221             encoder2_15_8(8) = 1;
222         else
223             encoder2_15_8(8) = 0;
224         end
225     end
226
227     case 4
228     % BYTE 4
229     % MOSI DATA – RED LED LSB
230     % MISO DATA – ENCODER 2 (7–0)
231     SS = 0;
232     readyMOSI = dec2bin(redValue(2),8);
233     % Receive the MISO byte bit by bit and update the vector holding the
234     % byte
235     switch bitNumber
236         case 1
```

```
237         if (MISO == true)
238             encoder2_7_0(1) = 1;
239         else
240             encoder2_7_0(1) = 0;
241         end
242     case 2
243         if (MISO == true)
244             encoder2_7_0(2) = 1;
245         else
246             encoder2_7_0(2) = 0;
247         end
248     case 3
249         if (MISO == true)
250             encoder2_7_0(3) = 1;
251         else
252             encoder2_7_0(3) = 0;
253         end
254     case 4
255         if (MISO == true)
256             encoder2_7_0(4) = 1;
257         else
258             encoder2_7_0(4) = 0;
259         end
260     case 5
261         if (MISO == true)
262             encoder2_7_0(5) = 1;
263         else
264             encoder2_7_0(5) = 0;
265         end
266     case 6
267         if (MISO == true)
268             encoder2_7_0(6) = 1;
269         else
270             encoder2_7_0(6) = 0;
271         end
272     case 7
273         if (MISO == true)
274             encoder2_7_0(7) = 1;
275         else
276             encoder2_7_0(7) = 0;
277         end
278     case 8
279         if (MISO == true)
280             encoder2_7_0(8) = 1;
281         else
282             encoder2_7_0(8) = 0;
283         end
284     end
285
286     case 5
287     % BYTE 5
288     % MOSI DATA – GREEN LED MSB
289     % MISO DATA – ENCODER 3 (23–16)
290     SS = 0;
```

```
291 readyMOSI = dec2bin(greenValue(1),8);
292 % Receive the MISO byte bit by bit and update the vector holding the
293 % byte
294 switch bitNumber
295     case 1
296         if (MISO == true)
297             encoder3_23_16(1) = 1;
298         else
299             encoder3_23_16(1) = 0;
300         end
301     case 2
302         if (MISO == true)
303             encoder3_23_16(2) = 1;
304         else
305             encoder3_23_16(2) = 0;
306         end
307     case 3
308         if (MISO == true)
309             encoder3_23_16(3) = 1;
310         else
311             encoder3_23_16(3) = 0;
312         end
313     case 4
314         if (MISO == true)
315             encoder3_23_16(4) = 1;
316         else
317             encoder3_23_16(4) = 0;
318         end
319     case 5
320         if (MISO == true)
321             encoder3_23_16(5) = 1;
322         else
323             encoder3_23_16(5) = 0;
324         end
325     case 6
326         if (MISO == true)
327             encoder3_23_16(6) = 1;
328         else
329             encoder3_23_16(6) = 0;
330         end
331     case 7
332         if (MISO == true)
333             encoder3_23_16(7) = 1;
334         else
335             encoder3_23_16(7) = 0;
336         end
337     case 8
338         if (MISO == true)
339             encoder3_23_16(8) = 1;
340         else
341             encoder3_23_16(8) = 0;
342         end
343 end
344
```

```
345     case 6
346     % BYTE 6
347     % MOSI DATA – GREEN LED LSB
348     % MISO DATA – ENCODER 3 (15–8)
349     SS = 0;
350     readyMOSI = dec2bin(greenValue(2),8);
351     % Receive the MISO byte bit by bit and update the vector holding the
352     % byte
353     switch bitNumber
354     case 1
355         if (MISO == true)
356             encoder3_15_8(1) = 1;
357         else
358             encoder3_15_8(1) = 0;
359         end
360     case 2
361         if (MISO == true)
362             encoder3_15_8(2) = 1;
363         else
364             encoder3_15_8(2) = 0;
365         end
366     case 3
367         if (MISO == true)
368             encoder3_15_8(3) = 1;
369         else
370             encoder3_15_8(3) = 0;
371         end
372     case 4
373         if (MISO == true)
374             encoder3_15_8(4) = 1;
375         else
376             encoder3_15_8(4) = 0;
377         end
378     case 5
379         if (MISO == true)
380             encoder3_15_8(5) = 1;
381         else
382             encoder3_15_8(5) = 0;
383         end
384     case 6
385         if (MISO == true)
386             encoder3_15_8(6) = 1;
387         else
388             encoder3_15_8(6) = 0;
389         end
390     case 7
391         if (MISO == true)
392             encoder3_15_8(7) = 1;
393         else
394             encoder3_15_8(7) = 0;
395         end
396     case 8
397         if (MISO == true)
398             encoder3_15_8(8) = 1;
```



```
399         else
400             encoder3_15_8(8) = 0;
401         end
402     end
403
404     case 7
405         % BYTE 7
406         % MOSI DATA – BLUE LED MSB
407         % MISO DATA – ENCODER 3 (7–0)
408         SS = 0;
409         readyMOSI = dec2bin(blueValue(1),8);
410         % Receive the MISO byte bit by bit and update the vector holding the
411         % byte
412         switch bitNumber
413             case 1
414                 if (MISO == true)
415                     encoder3_7_0(1) = 1;
416                 else
417                     encoder3_7_0(1) = 0;
418                 end
419             case 2
420                 if (MISO == true)
421                     encoder3_7_0(2) = 1;
422                 else
423                     encoder3_7_0(2) = 0;
424                 end
425             case 3
426                 if (MISO == true)
427                     encoder3_7_0(3) = 1;
428                 else
429                     encoder3_7_0(3) = 0;
430                 end
431             case 4
432                 if (MISO == true)
433                     encoder3_7_0(4) = 1;
434                 else
435                     encoder3_7_0(4) = 0;
436                 end
437             case 5
438                 if (MISO == true)
439                     encoder3_7_0(5) = 1;
440                 else
441                     encoder3_7_0(5) = 0;
442                 end
443             case 6
444                 if (MISO == true)
445                     encoder3_7_0(6) = 1;
446                 else
447                     encoder3_7_0(6) = 0;
448                 end
449             case 7
450                 if (MISO == true)
451                     encoder3_7_0(7) = 1;
452                 else
```

```
453         encoder3_7_0(7) = 0;
454     end
455     case 8
456         if (MISO == true)
457             encoder3_7_0(8) = 1;
458         else
459             encoder3_7_0(8) = 0;
460         end
461     end
462
463     case 8
464     % BYTE 8
465     % MOSI DATA – BLUE LED LSB
466     % MISO DATA – TACHOMETER 2 (23–16)
467     SS = 0;
468     readyMOSI = dec2bin(blueValue(2),8);
469
470     case 9
471     % BYTE 9
472     % MOSI DATA – SET ENCODER 2 (23–16)
473     % MISO DATA – TACHOMETER 2 (15–8)
474     SS = 0;
475     readyMOSI = dec2bin(0,8);
476
477     case 10
478     % BYTE 10
479     % MOSI DATA – SET ENCODER 2 (15–8)
480     % MISO DATA – TACHOMETER 2 (7–0)
481     SS = 0;
482     readyMOSI = dec2bin(0,8);
483
484     case 11
485     % BYTE 11
486     % MOSI DATA – SET ENCODER 2 (7–0)
487     % MISO DATA – TACHOMETER 3 (23–16)
488     SS = 0;
489     readyMOSI = dec2bin(0,8);
490
491     case 12
492     % BYTE 12
493     % MOSI DATA – SET ENCODER 3 (23–16)
494     % MISO DATA – TACHOMETER 3 (15–8)
495     SS = 0;
496     readyMOSI = dec2bin(0,8);
497
498     case 13
499     % BYTE 13
500     % MOSI DATA – SET ENCODER 3 (15–8)
501     % MISO DATA – TACHOMETER 3 (7–0)
502     SS = 0;
503     readyMOSI = dec2bin(0,8);
504
505     case 14
506     % BYTE 14
```

```

507 % MOSI DATA – SET ENCODER 3 (7–0)
508 % MISO DATA – RESERVED (0X00)
509 SS = 0;
510 readyMOSI = dec2bin(0,8);
511
512 case 15
513 % ** START OF CORE PACKET **
514 % BYTE 15
515 % MOSI DATA – CORE MODE (0X01)
516 % MISO DATA – CORE ID MSB
517 SS = 0;
518 readyMOSI = dec2bin(1,8);
519
520 case 16
521 % BYTE 16
522 % MOSI DATA – PADDING BYTE (0X00)
523 % MISO DATA – CORE ID LSB
524 SS = 0;
525 readyMOSI = dec2bin(0,8);
526
527 case 17
528 % BYTE 17
529 % MOSI DATA – CORE WRITE MASK
530 % MISO DATA – CURRENT SENSE 0 (15–8)
531 SS = 0;
532 % Do not enable the overwriting of encoders 0 and 1, but do enable the
533 % overwriting of the motor voltages
534 % Bit 5 – Set encoder 1 enable
535 % Bit 4 – Set encoder 0 enable
536 % Bit 3 – Write motor 1 voltage
537 % Bit 2 – Write motor 1 enable
538 % Bit 1 – Write motor 0 voltage
539 % Bit 0 – Write motor 0 enable
540 readyMOSI = dec2bin(15,8);
541
542 case 18
543 % BYTE 18
544 % MOSI DATA – MOTOR 0 COMMAND (15–8)
545 % MISO DATA – CURRENT SENSE 0 (7–0)
546 SS = 0;
547 % Convert the desired voltage to a value between –999 and 999
548 % 24 is the saturation level in the model
549 pitchVoltTemp = ceil((999*pitchVolt)/24);
550 % If the desired voltage is positive, concatenate a '1' to the front of
551 % the voltage value
552 % Pass the MSB of the new value
553 if (sign(pitchVoltTemp) == 1)
554     temp = dec2bin(pitchVoltTemp,15);
555     for i = 1:8
556         if (i == 1)
557             tempVoltBin(i) = '1';
558         else
559             tempVoltBin(i) = temp(i - 1);
560         end

```

```

561     end
562     readyMOSI = tempVoltBin;
563     % If the desired voltage is negative, find the 2's complement value of
564     % the voltage
565     elseif (sign(pitchVoltTemp) == -1)
566         temp = dec2bin(-1*pitchVoltTemp,15);
567         % Find the complement
568         for i = 1:15
569             if (temp(i) == '1')
570                 complement(i) = '0';
571             else
572                 complement(i) = '1';
573             end
574         end
575         temp1 = bin2dec(complement) + 1;
576         temp2 = dec2bin(temp1,15);
577         for i = 1:8
578             if (i == 1)
579                 tempVoltBin(i) = '1';
580             else
581                 tempVoltBin(i) = temp2(i - 1);
582             end
583         end
584         readyMOSI = tempVoltBin;
585     % If the desired voltage is zero, don't activate the motor
586     else
587         readyMOSI = dec2bin(0,8);
588     end
589
590     case 19
591     % BYTE 19
592     % MOSI DATA – MOTOR 0 COMMAND (7–0)
593     % MISO DATA – CURRENT SENSE 1 (15–8)
594     SS = 0;
595     % Convert the desired voltage to a value between -999 and 999
596     % 24 is the saturation level in the model
597     pitchVoltTemp = ceil((999*pitchVolt)/24);
598     % If the voltage is positive, pass the LSB
599     if (sign(pitchVoltTemp) == 1)
600         temp = dec2bin(pitchVoltTemp,15);
601         readyMOSI = temp(8:15);
602     % If the voltage is negative, find the 2's complement value and then
603     % pass the LSB
604     elseif (sign(pitchVoltTemp) == -1)
605         temp = dec2bin(-1*pitchVoltTemp,15);
606         % Find the complement
607         for i = 1:15
608             if (temp(i) == '1')
609                 complement(i) = '0';
610             else
611                 complement(i) = '1';
612             end
613         end
614         temp1 = bin2dec(complement) + 1;

```

```

615     temp2 = dec2bin(temp1,15);
616     readyMOSI = temp2(8:15);
617     % If the desired voltage is zero, do not activate the motor
618     else
619         readyMOSI = dec2bin(0,8);
620     end
621
622     case 20
623     % BYTE 20
624     % MOSI DATA – MOTOR 1 COMMAND (15–8)
625     % MISO DATA – CURRENT SENSE 1 (7–0)
626     SS = 0;
627     % Convert the desired voltage to a value between –999 and 999
628     % 24 is the saturation level in the model
629     yawVoltTemp = ceil((999*yawVolt)/24);
630     % If the desired voltage is positive, concatenate a '1' to the front of
631     % the voltage value
632     % Pass the MSB of the new value
633     if (sign(yawVoltTemp) == 1)
634         temp = dec2bin(yawVoltTemp,15);
635         for i = 1:8
636             if (i == 1)
637                 tempVoltBin(i) = '1';
638             else
639                 tempVoltBin(i) = temp(i - 1);
640             end
641         end
642         readyMOSI = tempVoltBin;
643     % If the desired voltage is negative, find the 2's complement value of
644     % the voltage
645     elseif (sign(yawVoltTemp) == -1)
646         temp = dec2bin(-1*yawVoltTemp,15);
647         % Find the complement
648         for i = 1:15
649             if (temp(i) == '1')
650                 complement(i) = '0';
651             else
652                 complement(i) = '1';
653             end
654         end
655         temp1 = bin2dec(complement) + 1;
656         temp2 = dec2bin(temp1,15);
657         for i = 1:8
658             if (i == 1)
659                 tempVoltBin(i) = '1';
660             else
661                 tempVoltBin(i) = temp2(i - 1);
662             end
663         end
664         readyMOSI = tempVoltBin;
665     % If the desired voltage is zero, don't activate the motor
666     else
667         readyMOSI = dec2bin(0,8);
668     end

```

```

669
670     case 21
671     % BYTE 21
672     % MOSI DATA – MOTOR 1 COMMAND (7–0)
673     % MISO DATA – TACHOMETER 0 (23–16)
674     SS = 0;
675     % Convert the desired voltage to a value between –999 and 999
676     % 24 is the saturation level in the model
677     yawVoltTemp = ceil((999*yawVolt)/24);
678     % If the voltage is positive, pass the LSB
679     if (sign(yawVoltTemp) == 1)
680         temp = dec2bin(yawVoltTemp,15);
681         readyMOSI = temp(8:15);
682     % If the voltage is negative, find the 2's complement value and then
683     % pass the LSB
684     elseif (sign(yawVoltTemp) == -1)
685         temp = dec2bin(-1*yawVoltTemp,15);
686         for i = 1:15
687             if (temp(i) == '1')
688                 complement(i) = '0';
689             else
690                 complement(i) = '1';
691             end
692         end
693         temp1 = bin2dec(complement) + 1;
694         temp2 = dec2bin(temp1,15);
695         readyMOSI = temp2(8:15);
696     % If the desired voltage is zero, do not activate the motor
697     else
698         readyMOSI = dec2bin(0,8);
699     end
700
701     case 22
702     % BYTE 22
703     % MOSI DATA – SET ENCODER 0 (23–16)
704     % MISO DATA – TACHOMETER 0 (15–8)
705     SS = 0;
706     readyMOSI = dec2bin(0,8);
707
708     case 23
709     % BYTE 23
710     % MOSI DATA – SET ENCODER (7–0)
711     % MISO DATA – TACHOMETER 0 (7–0)
712     SS = 0;
713     readyMOSI = dec2bin(0,8);
714
715     case 24
716     % BYTE 24
717     % MOSI DATA – SET ENCODER 0 (7–0)
718     % MISO DATA – TACHOMETER 1 (23–16)
719     SS = 0;
720     readyMOSI = dec2bin(0,8);
721
722     case 25

```

```
723 % BYTE 25
724 % MOSI DATA – SET ENCODER 1 (23–16)
725 % MISO DATA – TACHOMETER 1 (15–8)
726 SS = 0;
727 readyMOSI = dec2bin(0,8);
728
729 case 26
730 % BYTE 26
731 % MOSI DATA – SET ENCODER 1 (15–8)
732 % MISO DATA – TACHOMETER 1 (7–0)
733 SS = 0;
734 readyMOSI = dec2bin(0,8);
735
736 case 27
737 % BYTE 27
738 % MOSI DATA – SET ENCODER 1 (7–0)
739 % MISO DATA – STATUS
740 SS = 0;
741 readyMOSI = dec2bin(0,8);
742
743 case 28
744 % BYTE 28
745 % MOSI DATA – PADDING BYTE (0X00)
746 % MISO DATA – ENCODER 0 (23–16)
747 SS = 0;
748 readyMOSI = dec2bin(0,8);
749
750 case 29
751 % BYTE 29
752 % MOSI DATA – PADDING BYTE (0X00)
753 % MISO DATA – ENCODER 0 (15–8)
754 SS = 0;
755 readyMOSI = dec2bin(0,8);
756
757 case 30
758 % BYTE 30
759 % MOSI DATA – PADDING BYTE (0X00)
760 % MISO DATA – ENCODER 0 (7–0)
761 SS = 0;
762 readyMOSI = dec2bin(0,8);
763
764 case 31
765 % BYTE 31
766 % MOSI DATA – PADDING BYTE (0X00)
767 % MISO DATA – ENCODER 1 (23–16)
768 SS = 0;
769 readyMOSI = dec2bin(0,8);
770
771 case 32
772 % BYTE 32
773 % MOSI DATA – PADDING BYTE (0X00)
774 % MISO DATA – ENCODER 1 (15–8)
775 SS = 0;
776 readyMOSI = dec2bin(0,8);
```

```
777
778     case 33
779     % BYTE 33
780     % MOSI DATA – PADDING BYTE (0X00)
781     % MISO DATA – ENCODER 1 (7–0)
782     SS = 0;
783     readyMOSI = dec2bin(0,8);
784
785     case 34
786     % BYTE 34
787     % MOSI DATA – PADDING BYTE (0X00)
788     % MISO DATA – ACCELEROMETER X (15–8)
789     SS = 0;
790     readyMOSI = dec2bin(0,8);
791
792     case 35
793     % BYTE 35
794     % MOSI DATA – PADDING BYTE (0X00)
795     % MISO DATA – ACCELEROMETER X (7–0)
796     SS = 0;
797     readyMOSI = dec2bin(0,8);
798
799     case 36
800     % BYTE 36
801     % MOSI DATA – PADDING BYTE (0X00)
802     % MISO DATA – ACCELEROMETER Y (15–8)
803     SS = 0;
804     readyMOSI = dec2bin(0,8);
805
806     case 37
807     % BYTE 37
808     % MOSI DATA – PADDING BYTE (0X00)
809     % MISO DATA – ACCELEROMETER Y (7–0)
810     SS = 0;
811     readyMOSI = dec2bin(0,8);
812
813     case 38
814     % BYTE 38
815     % MOSI DATA – PADDING BYTE (0X00)
816     % MISO DATA – ACCELEROMETER Z (15–8)
817     SS = 0;
818     readyMOSI = dec2bin(0,8);
819
820     case 39
821     % BYTE 39
822     % MOSI DATA – PADDING BYTE (0X00)
823     % MISO DATA – ACCELEROMETER Z (7–0)
824     SS = 0;
825     readyMOSI = dec2bin(0,8);
826
827     case 40
828     % BYTE 40
829     % MOSI DATA – PADDING BYTE (0X00)
830     % MISO DATA – GYROSCOPE X (15–8)
```



```
831 SS = 0;
832 readyMOSI = dec2bin(0,8);
833
834 case 41
835 % BYTE 41
836 % MOSI DATA – PADDING BYTE (0X00)
837 % MISO DATA – GYROSCOPE X (7–0)
838 SS = 0;
839 readyMOSI = dec2bin(0,8);
840
841 case 42
842 % BYTE 42
843 % MOSI DATA – PADDING BYTE (0X00)
844 % MISO DATA – GYROSCOPE Y (15–8)
845 SS = 0;
846 readyMOSI = dec2bin(0,8);
847
848 case 43
849 % BYTE 43
850 % MOSI DATA – PADDING BYTE (0X00)
851 % MISO DATA – GYROSCOPE Y (7–0)
852 SS = 0;
853 readyMOSI = dec2bin(0,8);
854
855 case 44
856 % BYTE 44
857 % MOSI DATA – PADDING BYTE (0X00)
858 % MISO DATA – GYROSCOPE Z (15–8)
859 SS = 0;
860 readyMOSI = dec2bin(0,8);
861
862 case 45
863 % BYTE 45
864 % MOSI DATA – PADDING BYTE (0X00)
865 % MISO DATA – GYROSCOPE Z (7–0)
866 SS = 0;
867 readyMOSI = dec2bin(0,8);
868
869 case 46
870 % BYTE 46
871 % MOSI DATA – PADDING BYTE (0X00)
872 % MISO DATA – RESERVED (0X00)
873 SS = 0;
874 readyMOSI = dec2bin(0,8);
875
876 case 47
877 % BYTE 47
878 % MOSI DATA – PADDING BYTE (0X00)
879 % MISO DATA – RESERVED (0X00)
880 SS = 0;
881 readyMOSI = dec2bin(0,8);
882
883 case 48
884 % BYTE 48
```

```
885 % MOSI DATA – PADDING BYTE (0X00)
886 % MISO DATA – RESERVED (0X00)
887 SS = 0;
888 readyMOSI = dec2bin(0,8);
889
890 case 49
891 % BYTE 49
892 % MOSI DATA – PADDING BYTE (0X00)
893 % MISO DATA – RESERVED (0X00)
894 SS = 0;
895 readyMOSI = dec2bin(0,8);
896
897 case 50
898 % BYTE 50
899 % MOSI DATA – PADDING BYTE (0X00)
900 % MISO DATA – RESERVED (0X00)
901 readyMOSI = dec2bin(0,8);
902 % Reset the the slave select to begin the process again
903 % Maybe not needed
904 SS = 1;
905
906 otherwise
907 % Just in case
908 readyMOSI = dec2bin(0,8);
909 SS = 0;
910 end
911 % Pass the byte we have derived bit by bit
912 % Use the character array to determine the bit value
913 switch bitNumber
914 case 1
915     if (readyMOSI(1) == '1')
916         MOSI = 1;
917     else
918         MOSI = 0;
919     end
920 case 2
921     if (readyMOSI(2) == '1')
922         MOSI = 1;
923     else
924         MOSI = 0;
925     end
926 case 3
927     if (readyMOSI(3) == '1')
928         MOSI = 1;
929     else
930         MOSI = 0;
931     end
932 case 4
933     if (readyMOSI(4) == '1')
934         MOSI = 1;
935     else
936         MOSI = 0;
937     end
938 case 5
```

```
939     if (readyMOSI(5) == '1')
940         MOSI = 1;
941     else
942         MOSI = 0;
943     end
944     case 6
945         if (readyMOSI(6) == '1')
946             MOSI = 1;
947         else
948             MOSI = 0;
949         end
950     case 7
951         if (readyMOSI(7) == '1')
952             MOSI = 1;
953         else
954             MOSI = 0;
955         end
956     case 8
957         if (readyMOSI(8) == '1')
958             MOSI = 1;
959         else
960             MOSI = 0;
961         end
962     % If we have just sent the last bit, update the byte number
963     byteNumber = byteNumber + 1;
964     % Reset back to zero if we have finished one transfer
965     if (byteNumber == 51)
966         byteNumber = 0;
967     end
968     otherwise
969         MOSI = 0;
970 end
971 % Update the bit number that we are sending
972 bitNumber = bitNumber + 1;
973 % Reset back to 1 if we have finished one byte
974 if (bitNumber == 9)
975     bitNumber = 1;
976 end
977 end
```

# Appendix E

## Android Application

### E.1 Android Application (Tutorial)

1. Verify that all of the proper software is installed as mentioned in Section [6.4](#).
2. Repeat the steps used in Appendix [D.1](#) to push the Simulink model onto the Raspberry Pi. The Simulink model is specific for Android communication.
3. Open the Simulink model for the Android smart phone application.
4. Connect your Android smart phone to your laptop using a USB cable. Mine was provided with the phone. See Figure [E.1](#).
5. Deploy the model to the hardware. This is the same as that in Appendix [D.1](#), but this time the hardware is your smart phone.
6. You need to have your phone connected to internet in order for the build to complete.
7. Once the build is complete, you should see an application with the name of your model. See Figure [E.2](#).
8. Connect your smart phone to the local network you set up with static IPs. This was discussed in Section [6.4](#).
9. Open the application and the model should begin to run. See Figure [E.3](#).
10. Execute the code on the Raspberry Pi for its corresponding model. This is the same as in Appendix [D.1](#).
11. If both models are setup correctly and the IP addresses are consistent with the models, there should be communication working between the two devices.

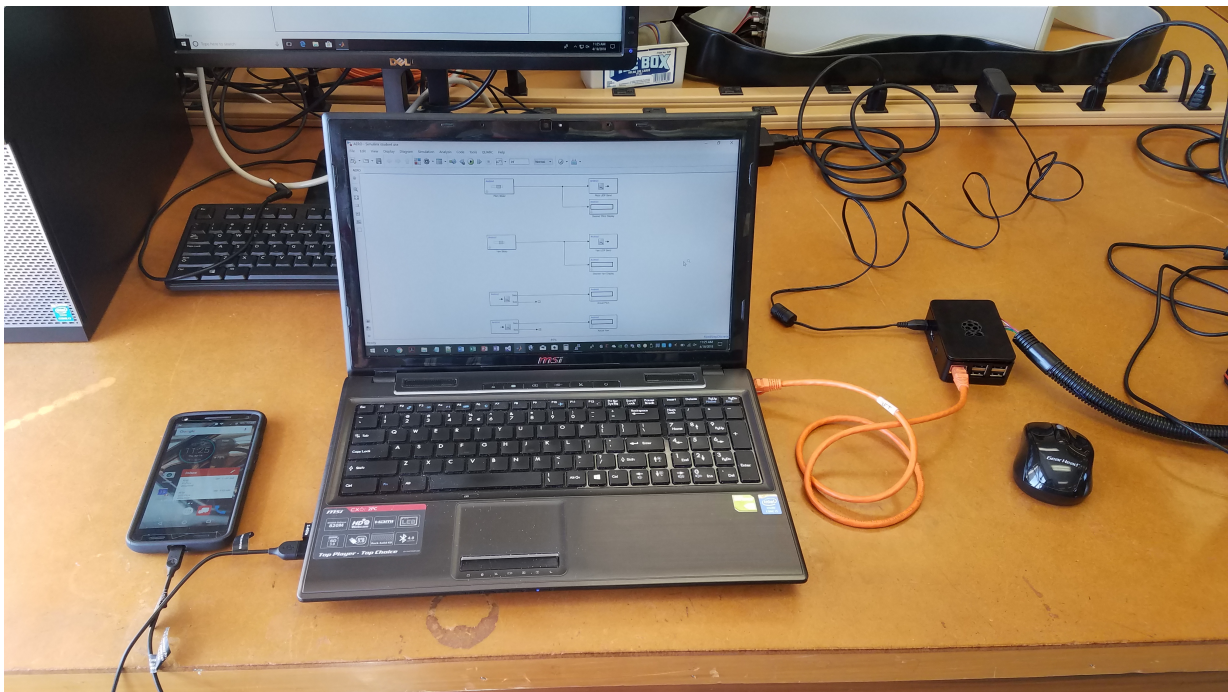


Figure E.1: Connect your Android smart phone to your laptop.

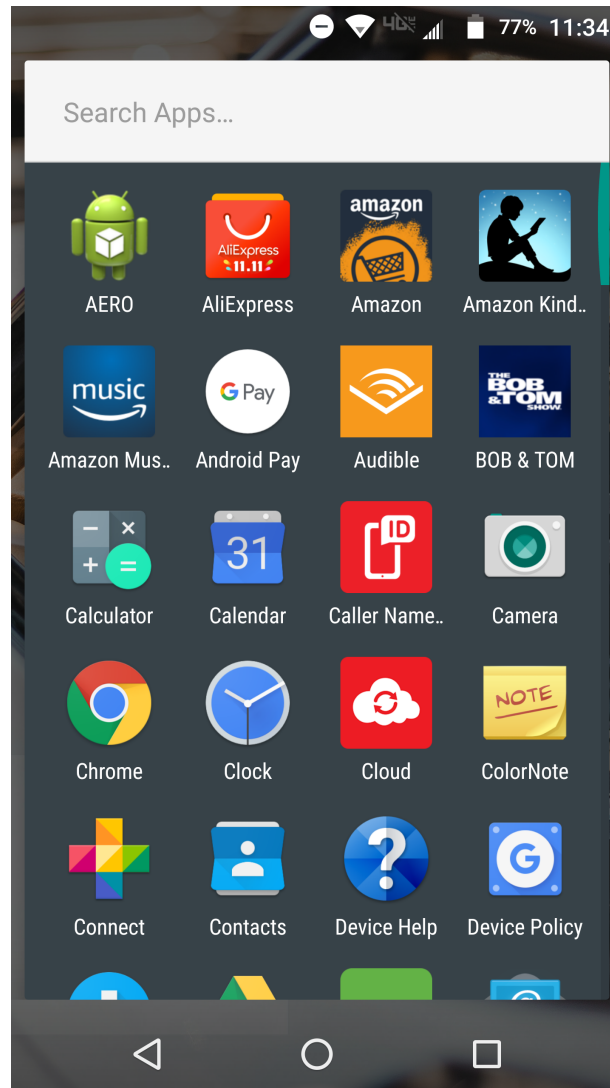


Figure E.2: Android smart phone application.

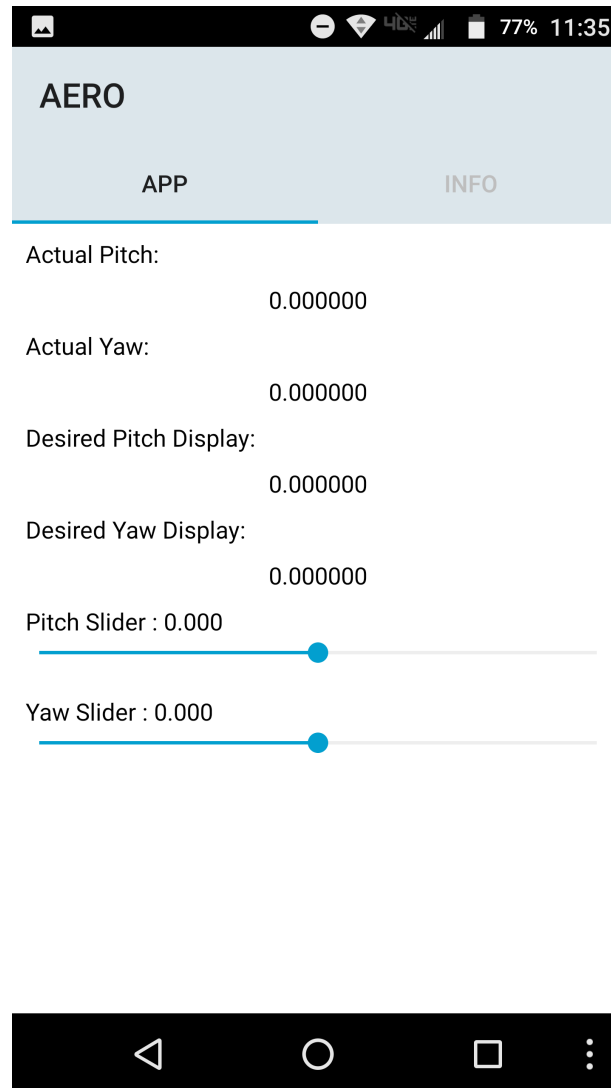


Figure E.3: Android smart phone application.

# Appendix F

## Quanser AERO Data Sheets

### F.1 Quanser AERO Product Specifications



## Dual-rotor aerospace experiment with reconfigurable dynamic components for mechatronics exploration and controls.

### FLEXIBLE PLATFORM FOR MECHATRONICS AND CONTROLS

The Quanser AERO is a fully integrated lab experiment, designed for teaching mechatronics and control concepts at the undergraduate level, as well as for advanced aerospace research applications.

The experiment is reconfigurable for various aerospace systems, from 1 DOF attitude control and 2 DOF helicopter to half-quadrotor. Integrating Quanser-developed QFLEX 2 computing interface technology, the Quanser AERO also offers flexibility in lab configurations, using a PC, or microcontrollers, such as NI myRIO, Arduino and Raspberry Pi. With the comprehensive course materials included, you can build a state-of-the-art teaching lab for your mechatronics or control courses, engage students in various design and capstone projects, and validate your research concepts on a high-quality, robust, and precise platform.

### HOW IT WORKS

The Quanser AERO consists of two propellers, powered by DC motors. Combined with the light-weight design of the experiment, this makes the system capable of highly responsive movements. The Quanser AERO's compact base includes a built-in amplifier with an integrated current sensor, built-in data acquisition device, and an interchangeable QFLEX 2 interface panel. The experiment comes with additional propellers to illustrate the efficiency of different propeller designs and effects of cross-coupling.

The propeller motors are equipped with optical encoders. The motor current and voltage sensors can be used to monitor the power consumption of the experiment.

The slip ring mechanism allows for continuous 360° yaw rotation. The angles of the pitch and yaw axes are measured using high-resolution optical encoders. The pitch and yaw axis can be independently locked, and the angle of the propeller assemblies can be adjusted between horizontal and vertical positions. This allows users to reconfigure the Quanser AERO for various aerospace systems (1 DOF attitude control, 2 DOF helicopter, half-quadrotor) and experiments (e.g. pitch-only system modeling).

The Inertial Measurement Unit (IMU) board includes accelerometer and gyroscope sensors, which can be used for attitude and yaw estimation and verification against the direct position measurements from the encoders.

The Quanser AERO also has a user-controllable tri-color LED strip. It can be programmed to indicate state, power, or other control performance characteristics of the Quanser AERO.



See system specifications on reverse.

### QUANSER AERO SOLUTION COMPONENTS

- ✓ Quanser AERO with QFLEX 2 interface panel of your choice (USB or Embedded)  
*Optional: Additional QFLEX 2 interface panel*
- ✓ Quanser Control Software (required for Quanser AERO USB experiment): QUARC for MATLAB/Simulink or QRCPC for LabVIEW\*
- ✓ Complete dynamic model and pre-build controllers
- ✓ ABET<sup>1</sup>-aligned, flexible digital media courseware (for Quanser AERO USB experiment)
- ✓ Arduino examples and interfacing datasheet (for Quanser AERO Embedded experiment)

*\*MATLAB/Simulink and LabVIEW licenses not included*

### QUANSER AERO INTERFACE OPTIONS

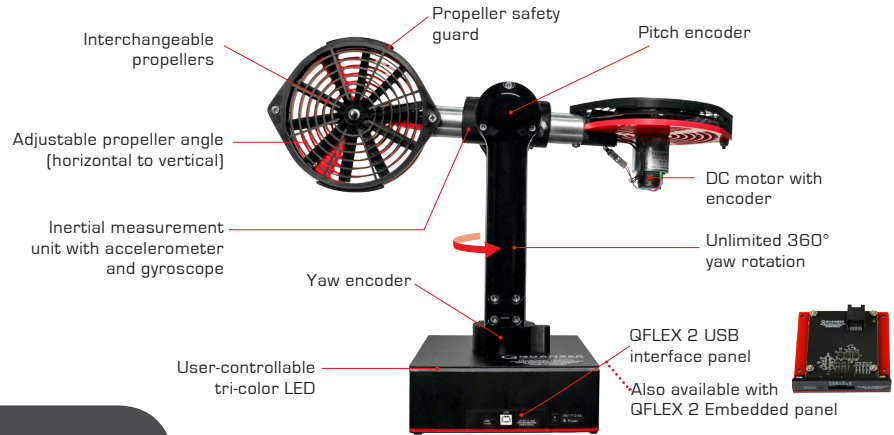
The Quanser AERO is available with two different, easily interchangeable interface panels:

- Quanser AERO USB experiment (with QFLEX 2 USB panel) interfaces to Quanser's control software running on your lab's PC via a standard USB 2.0 connection. The Quanser AERO USB can be used with MATLAB<sup>®</sup>/Simulink<sup>®</sup> and Quanser QUARC software, or with LabVIEW<sup>™</sup> using the Quanser RCP software. With the USB version of the experiment, you can take full advantage of the comprehensive course materials and lab experiments for your controls-based courses and projects.
- Quanser AERO Embedded experiment (with QFLEX 2 Embedded panel) interfaces to your microcontroller (not included with the experiment) via SPI connection. The Quanser AERO Embedded does not require any additional software. This option is ideal to expose students to various microcontroller techniques, as well as for final (capstone) projects in mechatronics, control, or other similar programs.

*Note: The Quanser AERO experiment includes one interface panel of your choice. Additional interface panel(s) can be purchased separately.*

# SYSTEM SPECIFICATIONS

## Quanser AERO



### FEATURES

- Compact and integrated system
- High-efficiency coreless DC motors
- High resolution optical encoders
- Pitch & yaw axes and DC motors/rotors speed measurements through digital tachometer
- Built-in voltage amplifier with integrated current sensor
- Integrated data acquisition (DAQ) device
- User-controllable tri-color LED
- Easy-connect cables and connectors
- Flexible QFLEX 2 computing interface for USB and SPI connections
- Open architecture design, allowing users to design their own controller
- Fully compatible with MATLAB®/Simulink® and LabVIEW™
- Fully documented system models and parameters provided for MATLAB®/Simulink® and LabVIEW™
- ABET-aligned, modular, digital media courseware provided for the Quanser AERO USB
- Microcontroller examples and interfacing datasheet provided for the Quanser AERO Embedded
- Additional community-created resources available on [www.QuanserShare.com](http://www.QuanserShare.com)

### COURSEWARE TOPICS COVERED

The Quanser AERO USB courseware includes ABET<sup>1</sup>-aligned **Instructor and Student Workbooks** with complete lab exercises, covering topics:

- Hardware integration
- Single propeller speed control
- **1 DOF attitude control configuration:**
  - PID control
  - Introduction to IMU
  - Modeling using transfer function
  - System identification
  - Gain scheduling

**Laboratory Guides** with modeling and control design examples:

- **2 DOF helicopter configuration**
  - Modeling
  - Linear state-space representation
  - State-feedback control
  - Coupled dynamics
- **Half-quadrotor configuration**
  - Modeling
  - Simple yaw control
  - Kalman filter

### DEVICE SPECIFICATION

Base dimensions (W x H x D)	17.8 cm x 17.8 cm x 7 cm
Device height (with propeller in horizontal position)	35.6 cm
Device length	51 cm
Device mass	3.6 kg
Propeller diameter	12.7 cm
Yaw angle range	360°
Elevation angle range	124° (± 62° from horizontal) half-quadrotor configuration
Pitch encoder resolution (in quadrature)	512 counts/revolution
Yaw/travel encoder resolution (in quadrature)	1024 counts/revolution
Motor current torque constant	57.7 mN.m/A
Tri-axis gyroscope range	± 245 dps
Tri-axis accelerometer range	± 2g
Interfaces available: QFLEX 2 USB	USB 2.0
QFLEX 2 Embedded	SPI

#### About Quanser:

Quanser is the world leader in education and research for real-time control design and implementation. We specialize in outfitting engineering control laboratories to help universities captivate the brightest minds, motivate them to success and produce graduates with industry-relevant skills. Universities worldwide implement Quanser's open architecture control solutions, industry-relevant curriculum and cutting-edge work stations to teach Introductory, Intermediate or Advanced controls to students in Electrical, Mechanical, Mechatronics, Robotics, Aerospace, Civil, and various other engineering disciplines.

## F.2 Quanser AERO User Manual



# USER MANUAL

## Quanser AERO Experiment

Set Up and Configuration



CAPTIVATE. MOTIVATE. GRADUATE.

© 2016 Quanser Inc., All rights reserved.

Quanser Inc.  
119 Spy Court  
Markham, Ontario  
L3R 5H6  
Canada  
info@quanser.com  
Phone: 1-905-940-3575  
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:  
<http://www.quanser.com>

This document and the software described in it are provided subject to a license agreement. Neither the software nor this document may be used or copied except as specified under the terms of that license agreement. All rights are reserved and no part may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Quanser Inc.

**FCC Notice** This device complies with Part 15 of the FCC rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

**Industry Canada Notice** This Class A digital apparatus complies with Canadian ICES-003. Cet appareil numérique de la classe A est conforme à la norme NMB-003 du Canada.

**Japan VCCI Notice** This is a Class A product based on the standard of the Voluntary Control Council for Interference (VCCI). If this equipment is used in a domestic environment, radio interference may occur, in which case the user may be required to take corrective actions.

この装置は、クラス A 情報技術装置です。この装置を家庭環境で使用する  
と電波妨害を引き起こすことがあります。この場合には使用者が適切な対策  
を講ずるよう要求されることがあります。 VCCI-A

**Korea Communications Commission (KCC) Notice** This equipment is Industrial (Class A) electromagnetic wave suitability equipment and seller or user should take notice of it, and this equipment is to be used in the places except for home.

이 기기는 업무용(A급) 전자파적합기기로서 판  
매자 또는 사용자는 이 점을 주의하시기 바라  
며, 가정외의 지역에서 사용하는 것을 목적으로  
합니다.

**Waste Electrical and Electronic Equipment (WEEE)**



This symbol indicates that waste products must be disposed of separately from municipal household waste, according to Directive 2002/96/EC of the European Parliament and the Council on waste electrical and electronic equipment (WEEE). All products at the end of their life cycle must be sent to a WEEE collection and recycling center. Proper WEEE disposal reduces the environmental impact and the risk to human health due to potentially hazardous substances used in such equipment. Your cooperation in proper WEEE disposal will contribute to the effective usage of natural resources. For information about the available collection and recycling scheme in a particular country, go to [ni.com/citizenship/weee](http://ni.com/citizenship/weee).

电子信息产品污染控制管理办法 (中国 RoHS)



中国客户 National Instruments 符合中国电子信息产品中限制使用某些有害物质命令 (RoHS)。  
关于National Instruments 中国 RoHS合规性信息, 请登录 [ni.com/environment/rohs\\_china](http://ni.com/environment/rohs_china)  
(For information about China RoHS compliance, go to [ni.com/environment/rohs\\_china](http://ni.com/environment/rohs_china))

**CE Compliance**

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

**Warning:** This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures

# CONTENTS

<b>1</b>	<b>Presentation</b>	<b>4</b>
<b>2</b>	<b>System Hardware</b>	<b>5</b>
2.1	System Schematic	5
2.2	Hardware Components	5
2.3	Environmental	8
2.4	System Parameters	9
<b>3</b>	<b>System Setup</b>	<b>10</b>
3.1	Components	10
3.2	QFLEX 2 USB Hardware Setup	10
3.3	QFLEX 2 Embedded Hardware Setup	11
3.4	Exchanging QFLEX 2 Panels	11
3.5	Changing Thruster Orientation	12
3.6	Exchanging Propellers	12
3.7	Pitch and Yaw Locks	14

# 1 PRESENTATION

The Quanser Quanser Aero, pictured in Figure 1.1, is a compact dual-rotor two degree-of-freedom aerospace system that can be used to perform a variety of actuator and flight control based experiments. The Quanser Aero can be configured with either the QFLEX 2 USB or QFLEX 2 Embedded interface modules. The QFLEX 2 USB allows control by a computer via USB connection. The QFLEX 2 Embedded allows for control by a microcontroller device such as an Arduino via a 4-wire SPI interface.

For all versions, the system is driven using two direct-drive 18V brushed DC motors. The motors are powered by a built-in PWM amplifier with built-in current sense. Single-ended rotary encoders are used to measure the angular position of the DC motors, and the speed of the motors can be measured with a software tachometer.

Main Quanser Aero features:

- Compact and complete aerospace control system
- 18V direct-drive brushed DC motors
- Encoders mounted on DC motors and support yolk
- DC motor tachometer output
- Built-in PWM amplifier with integrated current sense
- Built-in data acquisition (DAQ) device
- Low and High-efficiency propellers
- Lockable pitch and yaw axes
- Tri-color LED indicator lights



Figure 1.1: Quanser Aero



**Caution:** This equipment is designed to be used for educational and research purposes and is not intended for use by the general public. The user is responsible to ensure that the equipment will be used by technically qualified personnel only.

# 2 SYSTEM HARDWARE

## 2.1 System Schematic

The Quanser Aero can be configured with one of two different I/O interfaces: the QFLEX 2 USB, and the QFLEX 2 Embedded. The QFLEX 2 USB provides a USB interface for use with a computer. The QFLEX 2 Embedded provides a 4-wire SPI interface for use with an external microcontroller board.

The interaction between the different system components on the Quanser Aero is illustrated in Figure 2.1. On the data acquisition (DAQ) device block, the motor position encoders are connected to Encoder Input (EI) channels #0 and #1. EI2 reads the pitch angle of the Aero body, and EI3 reads the yaw angle of the yoke. The Analog Output (AO) channels are connected to the power amplifier command, which then drives the DC fan motors. The DAQ Analog Input (AI) channels are connected to the PWM amplifier current sense circuitry. The DAQ also controls the integrated tri-colour LEDs via an internal serial data bus. The DAQ can be interfaced to the PC or laptop via USB link in the QFLEX 2 USB, or to an external microcontroller via SPI in the QFLEX 2 Embedded.

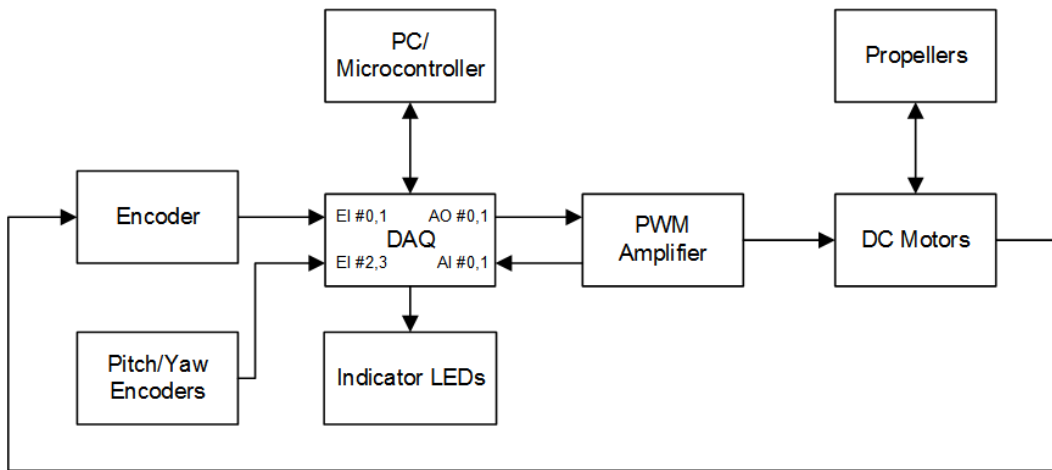


Figure 2.1: Interaction between Quanser Aero components.

The schematic given in Figure ?? illustrates the main Quanser Aero components and how they interact with a data acquisition (DAQ) device.

## 2.2 Hardware Components

The main Quanser Aero components - for the USB and SPI embedded interfaces - are listed in Table 2.1. The components on the QFLEX 2 USB are labeled in Figure 2.2c, the components on the QFLEX 2 Embedded are shown in Figure 2.2d.



**ESD Warning:** Quanser Aero internal components are sensitive to electrostatic discharge. Before handling the Quanser Aero ensure that you have been properly grounded.



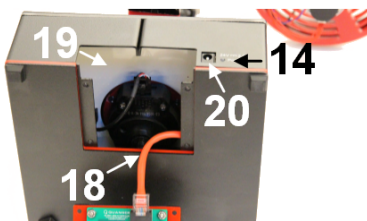
ID	Component	ID	Component
1	Aero base	11	Propeller guard screw
2	Yaw pivot	12	Propeller attachment hub
3	Support yolk	13	IMU (Internal to aero body)
4	Aero body	14	System Power LED
5	Pitch pivot	15	Interface Power LED
6	Pitch lock screws	16	SPI Data Connector*
7	Status LED strip	17	USB connector†
8	Thruster rotation locks	18	Quanser Aero internal data bus
9	Thruster 0	19	Quanser Aero DAQ/amplifier board
10	Thruster 1	20	Power Connector

Table 2.1: Quanser Aero Components

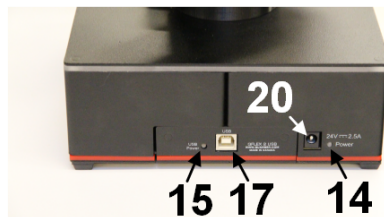
† only on QFLEX 2 USB  
 \*only on QFLEX 2 Embedded



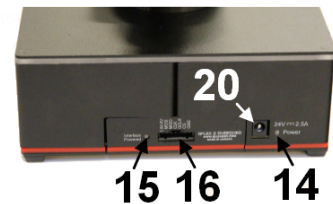
(a) Quanser Aero Top View



(b) Quanser Aero Interior



(c) Quanser Aero with QFLEX 2 USB



(d) Quanser Aero with QFLEX 2 Embedded

Figure 2.2: Quanser Aero components

### 2.2.1 DC Motor

The Quanser Aero includes two direct-drive 18V brushed DC motors. The motor specifications are given in Table 2.2.

The Quanser Aero incorporates the Allied Motion CL40 Series Coreless DC Motor model 16705. The complete specification sheet of the motor is included at: <http://alliedmotion.com/Products/Series.aspx?s=29>.



**Caution:** Max motor input  $\pm 18$  V, 2 A peak, 0.5 A continuous.



**Caution:** Exposed moving parts.



**Caution:** Holding the motor in a stalled position for a prolonged period of time at applied voltages of over 5V can result in permanent damage.

## 2.2.2 Thruster Assemblies

The Quanser Aero has two identical thruster assemblies which are attached to the Aero body. Thruster 0 can be identified by locating the pitch lock screws or the yaw lock magnets. Both of these items are on the side of the yolk facing thruster 0.

## 2.2.3 High-efficiency Propeller

The Quanser Aero ships with two counter-rotating APC 5.0x4.6 propellers, models LP05046E/EP. More information on the propellers can be found on the Advanced Precision Composites website ([www.apcprop.com](http://www.apcprop.com)).

## 2.2.4 Low-efficiency Propeller

The Quanser Aero ships with two eight-vane counter-rotating 3D-printed propellers.

## 2.2.5 Propeller Hub

The Quanser Aero propellers are connected to the DC motors with aluminum prop adapters with collets. The propeller adapters are E-flite part number EFLM1922.

## 2.2.6 Pitch and Motor Position Encoders

The encoders used to measure the pitch of the Aero body and the angular position of the DC motor on the Quanser Aero is a single-ended optical shaft encoder. It outputs 2048 counts per revolution in quadrature mode (512 lines per revolution).

The encoders used to measure the pitch of the Aero body, and angular position of the DC motors on the Quanser Aero is the US Digital E8P-512-118 single-ended optical shaft encoder. The complete specification sheet of the E8P optical shaft encoder is given in E8P Data Sheet.

## 2.2.7 Yaw Encoder

The encoders used to measure the yaw of the support yolk on the Quanser Aero is an optical encoder. It outputs 4096 counts per revolution in quadrature mode (1024 lines per revolution).

The encoders used to measure the yaw of the support yolk on the Quanser Aero is the US Digital E3-1024-984 optical encoder. The complete specification sheet of the E3 optical shaft encoder is given in E3 Data Sheet.

## 2.2.8 Inertial Measurement Unit

The Quanser Aero includes an integrated IMU mounted within the Aero body. This module allows for real-time measurement of the angular position and velocity along all three of the primary axes of the Aero body.

The IMU incorporated into the Quanser Aero is the STMicroelectronics LSM6DS0 iNEMO inertial module. Further information on the module can be found in the LSM6DS0 data sheet.

## 2.2.9 Data Acquisition (DAQ) Device

The Quanser Aero includes an integrated data acquisition device with four 16-bit encoder channels with quadrature decoding and two PWM analog output channels. The DAQ also incorporates a 12-bit ADC which provides current sense feedback for the motors. The current feedback is used to detect motor stalls and will disable the amplifier if a prolonged stall is detected.

## 2.2.10 Power Amplifier

The Quanser Aero circuit board includes a PWM voltage-controlled power amplifier capable of providing 2 A peak current and 0.5 A continuous current (based on the thermal current rating of the motor). The output voltage range to the load is between  $\pm 24$  V.

## 2.2.11 Power Supply

The Quanser Aero is equipped with an external DC power supply which provides power for the sensors and motors. This supply is intended for use with 100-240 VAC at 50-60 Hz.

Only the provided power supply and AC cord should be used with the Quanser Aero. The included supply is an Adapter Technology Co Ltd model ATS065-P241.

## 2.2.12 Embedded System Connector

The SPI data connector pictured on the QFLEX 2 Embedded in Figure 2.2d allows an external microcontroller to set motor voltage and LED brightnesses, read and set encoder counters, and read motor speed and current flow. See the QFLEX 2 Embedded data sheet for information on connecting the SPI interface.

## 2.3 Environmental

The Quanser Aero is designed to function under the following environmental conditions:

- Standard rating
- Indoor use only
- Temperature 5°C to 40°C
- Altitude up to 2000 m
- Maximum relative humidity of 80% up to 31°C decreasing linearly to 50% relative humidity at 40°C
- Pollution Degree 2
- Mains supply voltage fluctuations up to  $\pm 10\%$  of nominal voltage
- Maximum transient overvoltage 2500 V

- Marked degree of protection to IEC 60529: Ordinary Equipment (IPX0)

## 2.4 System Parameters

Table 2.2 lists and characterizes the main parameters associated with the Quanser Aero.

Symbol	Description	Value
<b>DC Motor</b>		
$V_{\text{nom}}$	Nominal input voltage	18.0 V
$\tau_{\text{nom}}$	Nominal torque	22.0 mN-m
$\omega_{\text{nom}}$	Nominal speed	3050 RPM
$I_{\text{nom}}$	Nominal current	0.540 A
$R_m$	Terminal resistance	8.4 $\Omega$
$k_t$	Torque constant	0.042 N-m/A
$k_m$	Motor back-emf constant	0.042 V/(rad/s)
$J_m$	Rotor inertia	$4.0 \times 10^{-6}$ kg-m <sup>2</sup>
$L_m$	Rotor inductance	1.16 mH
<b>Aero Body</b>		
$M_b$	Mass of body	1.075 kg
$D_m$	Center of mass	-7.59 mm
$J_p$	Pitch inertia	$2.15 \times 10^{-2}$ kg-m <sup>2</sup>
$J_y$	Yaw inertia	$2.37 \times 10^{-2}$ kg-m <sup>2</sup>
$D_t$	Thrust displacement	15.8 cm
<b>Motor and Pitch Encoders</b>		
	Encoder line count	512 lines/rev
	Encoder line count in quadrature	2048 lines/rev
	Encoder resolution (in quadrature)	0.176 deg/count
<b>Yaw Encoder</b>		
	Encoder line count	1024 lines/rev
	Encoder line count in quadrature	4096 lines/rev
	Encoder resolution (in quadrature)	0.088 deg/count
<b>Amplifier</b>		
	Amplifier type	PWM
	Peak Current	2 A
	Continuous Current	0.5 A
	Output voltage range (recommended)	$\pm 18$ V
	Output voltage range (maximum)	$\pm 24$ V

Table 2.2: Quanser Aero System Parameters

# 3 SYSTEM SETUP



**Caution:** If the equipment is used in a manner not specified by the manufacturer, the protection provided by the equipment may be impaired.

## 3.1 Components

To setup the Quanser Aero system, you need the following components:

1. Quanser Aero (USB or Embedded version)
2. High and low-efficiency propellers
3. Propeller adapters
4. Power supply with the following ratings:
  - Input Rating: 100-240 V AC, 50-60 Hz, 1.4 A
  - Output Rating: 24 V DC, 2.71 A

**Note:** Only the power supply provided (AC-DC adapter by Adapter Technology Co Ltd, model ATS065-P241) should be used with the Quanser Aero

5. Power cable

**Note:** Only the power cable provided should be used with the Quanser Aero

**Note:** Make sure that the power cable is accessible for disconnection in case of emergency.



**Caution:** Precaution must be taken during the connection of this equipment to the AC outlet to make sure the grounding (earthing) is in place, and that the ground wire is not disconnected

6. USB 2.0 A/B cable (for QFLEX 2 USB) or jumper wires (for QFLEX 2 Embedded)

## 3.2 QFLEX 2 USB Hardware Setup

To setup the QFLEX 2 USB follow these steps:

1. The Quanser Aero should have one of the included sets of propellers installed. If the other propellers are required, follow the procedure for exchanging propellers in Section ?? **before** connecting power.
2. Connect USB 2.0 cable from back cover of Quanser Aero to an enabled USB 2.0 port on your desktop PC or laptop.
3. Connect the **Power** connector on the Quanser Aero to the power supply. Ensure the power supply is connected to a wall outlet using the appropriate power cable.
4. The QFLEX 2 USB driver should install automatically. If not, then you may not have installed all the required software to support the device including either **QUARC®** or **Quanser Rapid Control Prototyping Toolkit®**.

### 3.3 QFLEX 2 Embedded Hardware Setup

This section describes how to connect the QFLEX 2 Embedded to an external microcontroller board. The connection procedure is given below, and summarized in Table 3.1. The wires required to connect the QFLEX 2 Embedded are not included with the unit, connections may be made with jumper wires or a custom wiring solution dependent on the external controller being used.

Follow these steps to connect the QFLEX 2 Embedded to your microcontroller device:

1. Before proceeding make sure your microcontroller device has been setup and successfully tested. Refer to the documentation supplied with your control system for set up and testing instructions.
2. Make sure everything is powered off before making any of these connections. This includes turning off the external microcontroller board.
3. Connect the GND pin on the QFLEX 2 Embedded to a digital ground connection on the microcontroller board.
4. Connect the MOSI, MISO, and CLK pins on the QFLEX 2 Embedded to the microcontroller board as outlined in the SPI interface documentation for your controller.
5. Connect the CS pin on the QFLEX 2 Embedded to a digital output on the microcontroller board.
6. Connect the 1.8V-5V pin on the QFLEX 2 Embedded to a signal level power pin on the microcontroller board in the 1.8V to 5V range.



**Caution:** Applying voltages in excess of 5V to the 1.8V-5V input on the QFLEX 2 Embedded may result in damage to the QFLEX 2 Embedded.

Cable #	From Microcontroller	To QFLEX 2 Embedded	Signal
1	VCC/VDD(1.8V-5V)	1.8V-5V	QFLEX 2 Embedded interface power.
2	MOSI/SDO/SO	MOSI	SPI master out, slave in data line.
3	MISO/SDI/SI	MISO	SPI master in, slave out data line.
4	SCLK/SCK	CLK	SPI clock line.
5	Digital output line	CS	SPI slave select line.
6	GND/DGND	GND	SPI digital signal ground.

Table 3.1: QFLEX 2 Embedded wiring summary

### 3.4 Exchanging QFLEX 2 Panels

Follow these steps to install the QFLEX 2 USB or QFLEX 2 Embedded panel in your Quanser Aero.

1. Disconnect the 24VDC power input from the Quanser Aero.
2. Disconnect any connections between the currently installed QFLEX panel and the computer or microcontroller board.



**ESD Warning:** The interior of the Quanser Aero contains components which are sensitive to electrostatic discharge. Before opening the Quanser Aero case, ensure that both you and the workspace are properly grounded.

3. Remove the four screws at the corners of the QFLEX panel to release the panel from the Aero chassis. The Quanser Aero is shown in Figure 3.1 below with the screws removed.

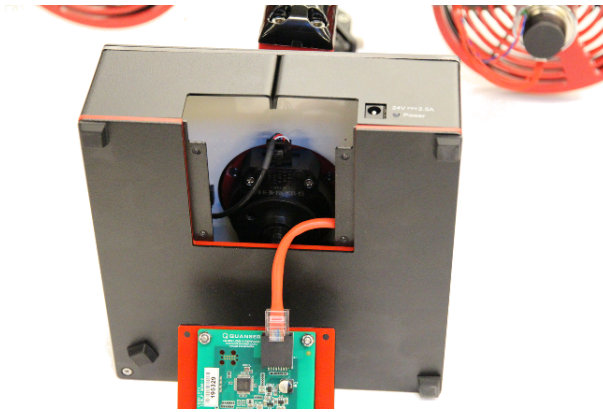


Figure 3.1: Quanser Aero with QFLEX panel detached.

4. Disconnect the Aero internal data cable from the QFLEX panel by depressing the latching tab.
5. Connect the Aero internal data cable to the QFLEX panel to be installed, pressing the connector into the socket until a click is heard and the connector latches in place.
6. Anchor the QFLEX panel in place using the four screws removed earlier.



**Caution:** Ensure that the Quanser Aero is completely reassembled, with all screws in place before connecting power or attempting operation.

## 3.5 Changing Thruster Orientation

Follow these steps to change the orientation of the thrusters on the Quanser Aero.

1. Use the included hex key to loosen the thruster rotation lock set-screw on the thruster you wish to rotate by one quarter turn as shown in Figure 3.2a



**Caution:** Do not loosen the set screws more than one half turn to prevent accidentally detaching the thruster assembly.

2. Rotate the propeller to the desired angle as shown in Figure 3.2b  
**Note:** Each thruster has a 90 degree range motion and will only rotate in one direction from either the vertical or horizontal positions.
3. Tighten the thruster rotation lock set-screw.

## 3.6 Exchanging Propellers

Follow these steps to change the propellers in the Quanser Aero.



**Caution:** The Quanser Aero is intended only for use with the included propellers. Operating the Quanser Aero with any other propellers may result in damage and/or injury.

1. Disconnect the 24VDC power input from the Quanser Aero.
2. Disconnect any connections between the currently installed QFLEX panel and the computer or microcontroller board.
3. Unfasten the propeller guard screws and remove the propeller guard.



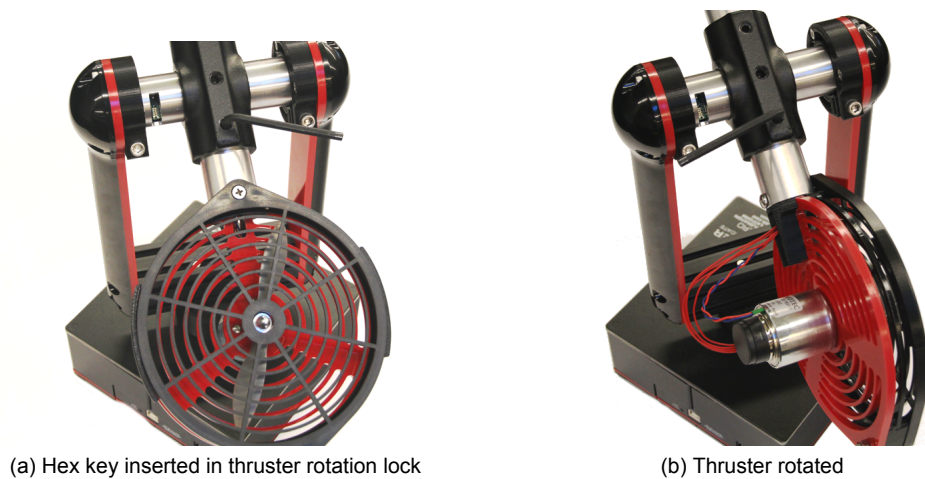


Figure 3.2: Quanser Aero propeller change steps

4. Insert a small hex key or similar object (not provided) through the cap of the propeller hub as shown in Figure 3.3a.
5. While holding the propeller still, loosen the collet in the propeller hub slightly by turning the cap counter-clockwise.
6. Pull gently on the propeller hub to slide the assembly off the motor shaft.
7. Disassemble the propeller hub as shown in Figure 3.3b



Figure 3.3: Quanser Aero propeller change steps

8. Identify the correct propeller from the counter-rotating pairs for the thruster being swapped. Under positive voltage, viewed from above, thruster 0 rotates counter-clockwise and thruster 1 rotates clockwise. Select the propeller such that positive voltage results in downward thrust. In the case of the high-efficiency propellers, the prop labeled 5x4.6E is intended for thruster 0, and that labeled 5x4.6EP is intended for thruster 1.
9. Slide the propeller assembly on to the motor shaft and tighten the cap.
10. Place the propeller cover back in position and replace the screws holding it in place.





**Caution:** Ensure that the Quanser Aero is completely reassembled, with all screws in place before connecting power or attempting operation. The outer propeller guard screw must be fastened with the included lock nut. Improper assembly may result in damage and/or injury.

## 3.7 Pitch and Yaw Locks

To lock the pitch of the Aero body, use the included hex key to tighten the pitch lock screws as shown in Figure 3.4a

To lock the yaw of the yoke, remove the hex key from its storage location in the bottom of the yoke and reinsert it with the long arm of the key down as shown in Figure 3.4b. Once the key is inserted all the way and protruding from the bottom of the yoke, rotate the yoke clockwise until the key comes in contact with the magnetized stop on the yaw pivot.

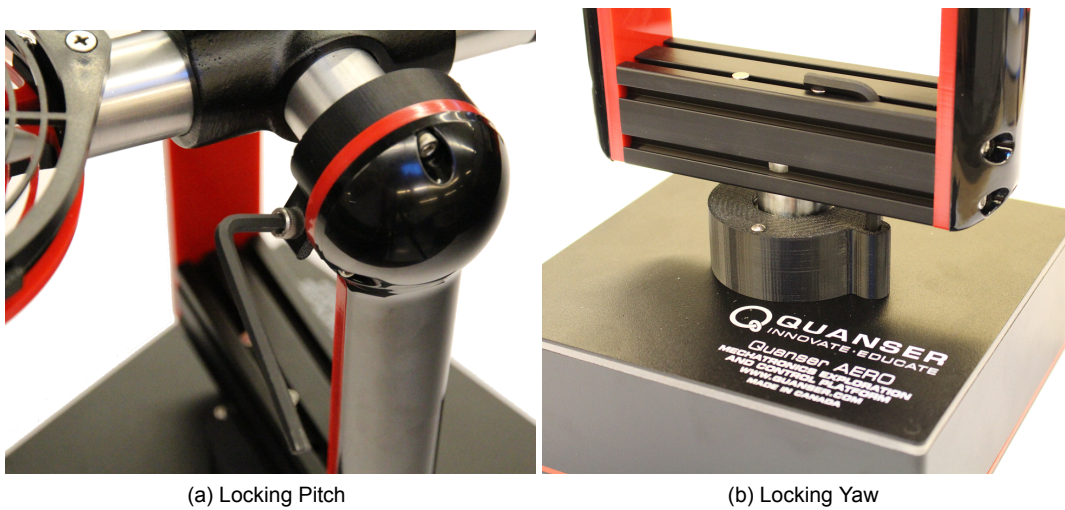


Figure 3.4: Quanser Aero attitude locks

# Quanser aerospace and unmanned systems for teaching and research



These systems allow you to study or research traditional and modern controls applications relating to spacecraft, unmanned vehicles, rescue missions and autonomous control. For more information please contact [info@quanser.com](mailto:info@quanser.com)

©2016 Quanser Inc. All rights reserved.



## F.3 Quanser AERO Embedded Data Sheet

# QFLEX 2 Embedded for Quanser AERO

For additional resources visit [www.quanser.com/courseware](http://www.quanser.com/courseware)

## Features

- Control DC motor voltages
- Read encoder feedback from DC motors
- Read encoder value from pitch and yaw pivots
- Read current sense feedback from PWM amplifier
- Read acceleration and velocity information from IMU
- Set encoder reference points
- Control tri-colour LED output

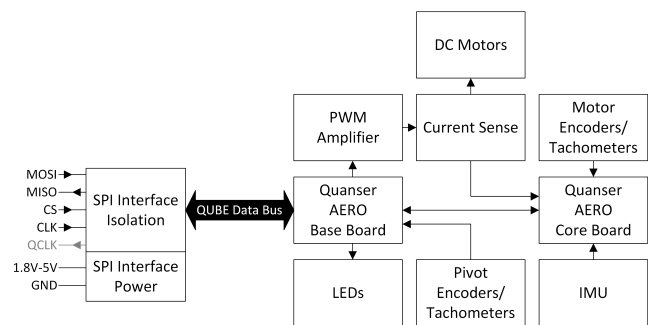


Figure 1: System Diagram

## System Overview

The QFLEX 2 Embedded provides a standard SPI interface for interacting with the Quanser AERO. The QFLEX 2 Embedded panel is powered via the 1.8V-5V and GND input pins and provides signal isolation between the external connector and the internal QUBE data bus. Data signals in the SPI interface will operate at the voltage provided on the 1.8V-5V pin. The Quanser AERO controller receives the data from the SPI interface and sets the motor PWM duty cycle, LED brightness, and encoder reference. The controller collects current sense and encoder read values and feeds this information back over the data bus, back to the SPI interface. The full system diagram is shown in Figure 1.

## Connections

The QFLEX 2 Embedded has a 7-pin connector (shown in Figure 2) which mates with a TE Connectivity 104257-6 rectangular connector. Table 1 outlines the IO connection pins

## Communication

The QFLEX 2 Embedded operates as a standard SPI slave with SPI mode 2. Note that the complete data

Table 1: QFLEX 2 Embedded IO connections

#	Pin Name	Description
1	1.8V-5V	Controller Vcc
2	MOSI	AERO data IN
3	MISO	AERO data OUT
4	CLK	SPI data clock IN
5	QCLK	QBUS clock OUT
6	CS	AERO slave chip select
7	GND	Controller ground

packet is comprised of two sub-packets, the first intended for the AERO base board, the second intended for the core board. When SPI communication is initiated by pulling the CS line low, the first byte of data received by the QFLEX 2 Embedded, referred to as the mode byte, will determine the communication mode to follow.

**Mode = 0** Only read ID (no settings changed)

**Mode = 1** A full command packet is transmitted

Concurrently, as the mode byte is being received, the Quanser AERO will respond with the upper byte of the base board ID. The next byte sent to the QFLEX 2 Em-



Figure 2: QFLEX 2 Embedded Panel

bedded is a padding byte with a value of 0, the data returned during this transmission will be the lower byte of the base board ID. At this point if Mode = 0, the next expected byte will be the mode byte for the core board, or byte 15 in a normal packet. If Mode = 1, the command communication packet continues as outlined in Table 2. Similarly, if mode 0 is selected for the core board, a padding byte will be expected and the lower byte of the core board ID will be returned, after which communication is complete and the CS line can be returned high. Otherwise if Mode = 1 for the core board, communication continues as outlined in Table 2.

### Transmit/Receive Bytes

The expected values and description for each of the data bytes are as follows.

#### Base Mode (Transmit byte 0)

Expected value 0x00-0x01. A value of 0 will result in only the Base ID being returned. A value of 1 will initiate the transmission of a full command packet.

#### Base Write Mask (Transmit byte 2)

Expected value 0x00-0x7F. This byte controls what values will be overwritten on the AERO base board. The mapping of the bits in the mask to written values is shown in Table 3 To zero the pivot encoders, the write mask must be 0b00011xxx. To set the LED values, the mask must be 0b000xx111.

#### LED Values (Transmit bytes 3-8)

Expected value 0x0000-0x03E7. The brightness of the LEDs is controlled on a scale from 0 to 999 (decimal) this value is transmitted over two bytes for each LED with the MSB preceding the LSB.

#### Set Encoder Values (Transmit bytes 9-14,22-27)

These bytes allow for the encoder counts value to be set as desired. The most likely application for these bytes is to send 0x00 for all bytes to zero the encoder counts.

Table 2: QFLEX 2 Embedded Command Packet Structure

B	MOSI Data	MISO Data
Start of Base Packet		
0	Base Mode (0x01)	Base ID MSB
1	Padding byte (0x00)	Base ID LSB
2	Base Write Mask	Encoder 2 (23-16)
3	Red LED MSB	Encoder 2 (15-8)
4	Red LED LSB	Encoder 2 (7-0)
5	Green LED MSB	Encoder 3 (23-16)
6	Green LED LSB	Encoder 3 (15-8)
7	Blue LED MSB	Encoder 3 (7-0)
8	Blue LED LSB	Tachometer 2 (23-16)
9	Set Encoder 2 (23-16)	Tachometer 2 (15-8)
10	Set Encoder 2 (15-8)	Tachometer 2 (7-0)
11	Set Encoder 2 (7-0)	Tachometer 3 (23-16)
12	Set Encoder 3 (23-16)	Tachometer 3 (15-8)
13	Set Encoder 3 (15-8)	Tachometer 3 (7-0)
14	Set Encoder 3 (7-0)	Reserved (0x00)
Start of Core Packet		
15	Core Mode (0x01)	Core ID MSB
16	Padding byte (0x00)	Core ID LSB
17	Core Write Mask	Current Sense 0 (15-8)
18	Motor 0 Command (15-8)	Current Sense 0 (7-0)
19	Motor 0 Command (7-0)	Current Sense 1 (15-8)
20	Motor 1 Command (15-8)	Current Sense 1 (7-0)
21	Motor 1 Command (7-0)	Tachometer 0 (23-16)
22	Set Encoder 0 (23-16)	Tachometer 0 (15-8)
23	Set Encoder 0 (15-8)	Tachometer 0 (7-0)
24	Set Encoder 0 (7-0)	Tachometer 1 (23-16)
25	Set Encoder 1 (23-16)	Tachometer 1 (15-8)
26	Set Encoder 1 (15-8)	Tachometer 1 (7-0)
27	Set Encoder 1 (7-0)	Status
28	Padding byte (0x00)	Encoder 0 (23-16)
29	Padding byte (0x00)	Encoder 0 (15-8)
30	Padding byte (0x00)	Encoder 0 (7-0)
31	Padding byte (0x00)	Encoder 1 (23-16)
32	Padding byte (0x00)	Encoder 1 (15-8)
33	Padding byte (0x00)	Encoder 1 (7-0)
34	Padding byte (0x00)	Accelerometer X (15-8)
35	Padding byte (0x00)	Accelerometer X (7-0)
36	Padding byte (0x00)	Accelerometer Y (15-8)
37	Padding byte (0x00)	Accelerometer Y (7-0)
38	Padding byte (0x00)	Accelerometer Z (15-8)
39	Padding byte (0x00)	Accelerometer Z (7-0)
40	Padding byte (0x00)	Gyroscope X (15-8)
41	Padding byte (0x00)	Gyroscope X (7-0)
42	Padding byte (0x00)	Gyroscope Y (15-8)
43	Padding byte (0x00)	Gyroscope Y (7-0)
44	Padding byte (0x00)	Gyroscope Z (15-8)
45	Padding byte (0x00)	Gyroscope Z (7-0)
46	Padding byte (0x00)	Reserved (0x00)
47	Padding byte (0x00)	Reserved (0x00)
48	Padding byte (0x00)	Reserved (0x00)
49	Padding byte (0x00)	Reserved (0x00)
50	Padding byte (0x00)	Reserved (0x00)

**Table 3: Base Write Mask Bit Mapping**

b	Action Enabled
7	-
6	-
5	-
4	Set Encoder 3
3	Set Encoder 2
2	Write Blue LED
1	Write Green LED
0	Write Red LED

**Core Mode (Transmit byte 15)**

Expected value 0x00-0x01. A value of 0 will result in only the Core ID being returned, ending communication after byte 16. A value of 1 will continue the transmission of a full command packet.

**Core Write Mask (Transmit byte 2)**

Expected value 0x00-0x7F. This byte controls what values will be overwritten on the AERO base board. The mapping of the bits in the mask to written values is shown in Table 3 To set the motor encoders, the write

**Table 4: Core Write Mask Bit Mapping**

b	Action Enabled
7	-
6	-
5	Set Encoder 1
4	Set Encoder 0
3	Write Motor 1 Voltage
2	Write Motor 1 Enable
1	Write Motor 0 Voltage
0	Write Motor 0 Enable

mask must be 0b0011xxxx. To enable the motors and allow a command value to be written the mask must be 0b00xx1111.

**Motor Command (Transmit bytes 18-21)**

Bit 15 of the motor commands control whether the PWM amplifier in the Quanser AERO for that motor is activated. Thus the upper byte of the motor command must be 0b1xxxxxxx in order for the motor to be enabled. The expected value of the bits 14-0 of the motor command is a value between -999 and 999 (decimal) formatted in 2's complement and represents a value equal to 10 times the desired percentage duty cycle of the PWM amplifier. For example, writing the motor command value 0x81F4 would activate the PWM amplifier and apply a 50% duty cycle, equivalent to approximately 12VDC.

**Padding Bytes (Transmit bytes 28-50)**

The value of these bytes are ignored by the QFLEX 2 Embedded, serving only to provide clock cycles for additional MISO data. By convention these bytes are usually zero.

**Base Module ID (Receive bytes 0-1)**

Expected Value of these bytes for the Quanser AERO Base Module is always 0x304 or 772 decimal. Any other value indicates a fault in communication between the QFLEX 2 Embedded and the AERO base board.

**Read Encoder Values (Receive bytes 2-7,28-33)**

These bytes represent the current value of the encoder counts, represented in 2's complement. Each value consists of three bytes and begins with the most significant byte. Encoders 0 and 1 are the encoders indicating the position of the DC motors. Encoder 2 is the encoder connected to the shoulder "pitch" pivot of the AERO body. Encoder 3 is the encoder connected to the base "yaw" pivot of the support yolk. Note that encoders 0-2 are 2048 quadrature counts per rotation, while encoder 3 is 4096 quadrature counts per rotation.

**Tachometer Value (Receive bytes 8-13,21-26)**

These bytes represent the current value read from the Quanser AERO internal tachometer, consisting of three bytes and beginning with the most significant byte. Tachometer values are calculated for each encoder. The most significant bit represents the direction of rotation with a value of zero indicating counter-clockwise rotation. Bits 23-0 represent the number of clock cycles (at 40MHz) between rising edges of the encoder A signal line. The encoder value can be converted to encoder counts per second with the following equation:

$$\text{Counts/Sec} = \frac{4}{T_{ach} \cdot (25 \cdot 10^{-9})} \quad (1)$$

Note that when the motor/pivot velocity is 0 the tachometer value will indicate the maximum value of 0x7FFFFFFF indicating a reading of approximately 20 counts/s.

**Core Module ID (Receive bytes 15-16)**

Expected Value of these bytes for the Quanser AERO Core Module is always 0x307 or 775 decimal. Any other value indicates a fault in communication between the QFLEX 2 Embedded and the AERO core board.

### Current Sense (Receive bytes 17-20)

These bytes represent the measured current draw of the DC motors, represented in two's complement. The current can be calculated using the equation:

$$\text{Current (mA)} = \frac{I_{sense} - 8190}{9828} \quad (2)$$

### Status (Receive byte 27)

The three least significant bits of this byte represent various warning or error states as outlined in Table 5. A status byte value of 0x00 indicates normal operation.

**Table 5: Status Bits**

b	Indicated Status
7	-
6	-
5	Motor 1 Stall Error
4	Motor 1 Stall Detected
3	Motor 0 Stall Error
2	Motor 0 Stall Detected
1	Amplifier 1 Over-Current Fault
0	Amplifier 0 Over-Current Fault

### Accelerometer Readings (Receive bytes 34-39)

There are three accelerometer readings each represented by a 16-bit signed integer in two's complement. The full scale reading is  $\pm 2g$  so the acceleration can be calculated from the measured value using the following formula:

$$\text{Acceleration} = \frac{2 \cdot Acc_n \cdot 9.81}{32768} \quad (3)$$

Note that only the X axis (extending down the center of the AERO body towards thruster 0) and Z axis (extending vertically when the AERO body is horizontal) will be of interest. Under normal operation the Y-axis acceleration should always be approximately zero.

### Gyroscope Readings (Receive bytes 40-45)

There are three accelerometer readings each represented by a 16-bit signed integer in two's complement. The full scale reading is  $\pm 245$  degrees/s so the angular velocity can be calculated from the measured value using the following formula:

$$\omega_n = \frac{Gyro_n \cdot 245}{32768} \quad (4)$$

© 2016 Quanser Inc., All rights reserved.

Quanser Inc.  
119 Spy Court  
Markham, ON  
L3R 5H6  
Canada  
info@quanser.com  
Phone: 1-905-940-3575  
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at: <http://www.quanser.com>

This document and the software described in it are provided subject to a license agreement. Neither the software nor this document may be used or copied except as specified under the terms of that license agreement. Quanser Inc. grants the following rights: a) The right to reproduce the work, to incorporate the work into one or more collections, and to reproduce the work as incorporated in the collections, b) to create and reproduce adaptations provided reasonable steps are taken to clearly identify the changes that were made to the original work, c) to distribute and publicly perform the work including as incorporated in collections, and d) to distribute and publicly perform adaptations. The above rights may be exercised in all media and formats whether now known or hereafter devised. These rights are granted subject to and limited by the following restrictions: a) You may not exercise any of the rights granted to You in above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation, and b) You must keep intact all copyright notices for the Work and provide the name Quanser Inc. for attribution. These restrictions may not be waived without express prior written permission of Quanser Inc.

# Bibliography

- [1] Q. Ahmed, A. I. Bhatti, S. Iqbal, and I. H. Kazmi. 2-sliding mode based robust control for 2-dof helicopter. In *2010 11th International Workshop on Variable Structure Systems (VSS)*, pages 481–486, June 2010.
- [2] Asma Al-Tamimi, Frank L Lewis, and Murad Abu-Khalaf. Discrete-time nonlinear hjb solution using approximate dynamic programming: convergence proof. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 38(4):943–949, 2008.
- [3] L. M. Argentim, W. C. Rezende, P. E. Santos, and R. A. Aguiar. Pid, lqr and lqr-pid on a quadcopter platform. In *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 1–6, May 2013.
- [4] P. Bhatkhande and T. C. Havens. Real time fuzzy controller for quadrotor stability control. In *2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 913–919, July 2014.
- [5] J.H.; Lee H.J. Chang, W.; Moon. Fuzzy model-based output-tracking control for 2 degree-of-freedom helicopter. *Journal of Electrical Engineering Technology*, 12.00(1):1921–1928, 2017. Quanser product(s): 2 DOF Helicopter.
- [6] W. Gao and Z. P. Jiang. Data-driven adaptive optimal output-feedback control of a 2-dof helicopter. In *2016 American Control Conference (ACC)*, pages 2512–2517, July 2016.
- [7] Eswarmurthi Gopalakrishnan. Quadcopter flight mechanics model and control algorithms. In *Czech Technical University - Department of Control Engineering*, May 2016.
- [8] Simon Haykin. *Neural networks and learning machines*. Prentice Hall/Pearson, New York, third edition, 2009.
- [9] M. Hernandez-Gonzalez, A.Y. Alanis, and E.A. Hernandez-Vargas. Decentralized discrete-time neural control for a quanser 2-dof helicopter. In *Applied Soft Computing*, pages 2462–2469, February 2012.
- [10] Ali Heydari. Revisiting approximate dynamic programming and its convergence. *IEEE Transactions on Cybernetics*, 48:2733–2743, 12 2014.



- 
- [11] Tomas Jinec. Stabilization and control of unmanned quadcopter. In *Lule University of Technology - Department of Computer Science, Electrical and Space Engineering*, pages 913–919, May 2011.
- [12] E. Kayacan and M.A. Khanesar. Recurrent interval type-2 fuzzy control of 2-dof helicopter with finite time training algorithm. In *IFAC-PapersOnLine*, pages 293–299, July 2016.
- [13] Frank L. Lewis, Draguna L. Vrabie, and Vassilis L. Syrmos. *Optimal Control*. John Wiley and Sons, Hoboken, New Jersey, third edition, 2012.
- [14] Teppo Luukkonen. Modelling and control of quadcopter. In *2014 American Control Conference*, volume Aalto School of Science, June 2011.
- [15] K. D. Sebesta and N. Boizot. A real-time adaptive high-gain ekf, applied to a quadcopter inertial navigation system. *IEEE Transactions on Industrial Electronics*, 61(1):495–503, Jan 2014.
- [16] K. Subbarao, P. Nuthi., and G. Atmeh. Reinforcement learning based computational adaptive optimal control and system identification for linear systems. In *Annual Reviews in Control*, pages 319–331, September 2016.
- [17] R.G. Subramanian and V.K. Elumalai. Robust mrac augmented baseline lqr for tracking control of 2-dof helicopter. In *Robotics and Autonomous Systems*, pages 70–77, August 2016.