4-20-2023

Implementation of Model-Free Sliding Mode Control on a Multi-Input Multi-Output System

Walker Hare
wth8798@rit.edu

ROCHESTER INSTITUTE OF TECHNOLOGY

Implementation of Model-Free Sliding Mode Control on a Multi-Input Multi-Output System

By: Walker Hare

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Mechanical Engineering

Advisor: Dr. Agamemnon Crassidis

DEPARTMENT OF MECHANICAL ENGINEERING KATE

GLEASON COLLEGE OF ENGINEERING

Rochester, NY

April 20th, 2023

Implementation of Model-Free Sliding Mode Control on a Multi-Input Multi-Output System

By: Walker Hare

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Mechanical Engineering

Department of Mechanical Engineering

Kate Gleason College of Engineering

Rochester Institute of Technology

Approved By:

Dr. Agamemnon Crassidis Date
Thesis Advisor
Department of Mechanical Engineering

Dr. Kathleen Lamkin-Kennard Date
Thesis Committee Member
Department of Mechanical Engineering

Dr. Sergey Lyshevski Date
Thesis Committee Member
Department of Electrical Engineering

Dr. Sarilyn Ivancic Date
Department Representative
Department of Mechanical Engineering

Abstract

Bioinspired flight serves to balance the advantages of fixed-wing flight as well as traditional rotary systems. A flapping wing aerial system exhibits high maneuverability at low speeds like a rotary system and the higher efficiency that fixed-wing systems exhibit in forward flight. A method to actuate a flapping wing structure is using piezoelectric elements. Using flapping wings to perform aerial maneuvers and facilitate stable flight will require precise control of the piezoelectric elements and control surfaces. The focus of the research is the construction of a feedback control platform for the flapping wing aerial system. For any type of feedback control architecture, an accurate way of estimating the current state of the system is required for precise tracking control. Since the control system will be implemented in an aerial vehicle, the system must be lightweight to allow for a larger payload capacity for the aircraft. Nonlinearities arising from uncertainties and unmodeled environmental effects lead to changes in the system model and a linear control scheme will exhibit suboptimal performance and instability. The desire of this aircraft to be able to perform under different flight conditions make determination of an accurate system model a time-consuming and vast undertaking. These flight conditions include take off, gliding, and landing under different wind, pressure, temperature, and payload specifications. Sliding Mode Control and Lyapunov-based methods have been explored extensively due to their capability of handling nonlinear systems directly while guaranteeing stable tracking control. However, determining an appropriate model to approximate the system dynamics is difficult and can lead to undesirable results. These variations can be accounted for by using a model-free control approach. In this work, a model-free controller based on the Sliding Mode Control method implemented for a flapping wing aerial system is considered. The model-free control algorithm only requires knowledge of the system parameters such as the order of the system and state measurements. The algorithm has been implemented successfully in traditional unmanned aerial systems achieving adequate control of system states. While the goal is for the Model-free algorithm to be implemented on a flapping-wing unmanned aerial system, at the time of this thesis, the construction is still in progress. Therefore, an analogous system was built to test the algorithms. A balancing system consisting of a DC motor and propeller mounted to a servo motor for thrust vectoring was constructed. A linear controller derived from the system model was compared in simulation and testing to the Model-Free Sliding Mode Control algorithm.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

NONMENCLATURE

α_u = *Control Input Low-Pass Filter Constant*

b = *Control Input Gain*

$\hat{b}$ = *Estimate of control input gain*

β = *Auxiliary variable for on-line parameter estimation*

b_u, b_l = *Upper and lower bounds of the control input gain*

η = *Increment to the sliding gain (always positive)*

$\hat{\eta}$ = *Estimate of the increment to the switching gain*

$\varepsilon(u)$ = *Input error*

$\hat{\varepsilon}(u)$ = *Estimate of the control input error*

$\frac{d}{dt}x$ = *First derivative of x*

$f(\boldsymbol{x})$ = *Function of system states*

$\hat{f}$ = *Approximation of the function of system states*

K = *Switching gain*

k_0 = *Pre-specified bound for estimator gain matrix magnitude*

λ = *Slope of the sliding surface (always positive)*

λ_p = *Time-varying forgetting factor for the determination of the estimator gain matrix,* $\boldsymbol{P}$

λ_0 = *Maximum forgetting factor rate*

n = *System Order*

P = *Estimator gain matrix*

$\boldsymbol{\varphi}$ = *Boundary Layer*

RMS = Root-Mean-Square

s = *Time-varying sliding surface*

Sat() = Saturation function

Sgn() = Signum (relay) function (for s>0, sgn(s)=1, for s<0, sgn(s)=-1)

σ_l, σ_u *= Upper and lower bounds for control input estimation*

u *= Control Input*

$\hat{u}$ *= Redefined control law*

u_{k-1}, u_{k-2} *= Previous and second previous control inputs*

x *= System state vector*

$\dot{x}$ *= First derivative of x*

$\tilde{x}$ *= Tracking error*

x_d *= Desired tracking*

$x(0)$ *= Initial value of "x"*

1.0 MOTIVATION

Advancements in Unmanned Aircraft Systems (UASs) technology have led to increased prevalence in the warzone of today in applications such as surveillance and combat capacity. Unfortunately, the distinguishing profile of traditional UASs and sound signatures limits the effectiveness in operations where remaining undetected is of utmost importance. The goal of designing a control system is to achieve desired system performance in response to user inputs as well as achieving fully autonomous navigation control.

Flapping-wing flight provides advantages over fixed-wing flight under certain operating conditions. Sachs [1] examined flight in biological systems and found flapping wings potentially offer increased maneuverability and energy savings compared to fixed-wing flight. However, these advantages are greatest when the size of the aircraft is small, and the velocity is low [2]. The slower tip speed of the wings decreases the noise generated in flight [3]. To minimize the complexity and cost, many flapping wing aerial systems are actuated via electric motors [4]. However, as the size of the motor is decreased, the overall efficiency decreases as well [4]. Transmitting the power from the motor to the wings requires the use of gears and linkages. Degradation of the mechanical components within the transmission system may lead to increased maintenance, shorter lifespan, and increased operational costs.

O'Rourke [6] developed a second-order linear model using system identification techniques to model the displacement of the tip of the piezoelectric actuator (PZT). In the work, a nonlinear plant was constructed in Simscape. The model used the Euler-Bernoulli finite-element method and coupling terms to consider the electromechanical interaction. By inputting the PZT technical parameters from the specification documentation, the system outputs for desired inputs and external disturbances were determined. The second-order approximation outlined in O'Rourke [6] was lacking for proper control of the PZT most likely due to a linear approximation not accounting for nonlinearities such as hysteresis and creep within the actuator. In O'Rourke [6], unmodeled dynamics could have led to discrepancies between the model and test data.

The control law is designed to satisfy parameters such as the settling time given an input, steady-state error, and overshoot. Linear controllers such as the classical Proportional-Integral-Derivative Controller (PID) and Linear Quadratic Regulator (LQR) offer adequate results for many classes of linear systems. For PID Control, time and frequency domain analysis are typically used to derive the controller gains to achieve the desired system performance. In the design of an LQR Controller, a cost function is minimized to determine a set of gains ensuring optimal control. However, the LQR performance, when implemented, is limited to the accuracy of the linear approximation of the dynamics. Therefore,

uncertainties in the model parameters lead to decreasing robustness. While many different linear control laws can be implemented on real systems with reliable results, a majority fall short when attempting to control higher-order nonlinear systems.

Nonlinear control methods, such as Sliding-Mode Control (SMC), have been implemented to compensate where linear system model approximations do not yield acceptable closed-loop performance. SMC is more flexible compared to other types of control laws since direct control of higher-order nonlinear systems is possible while also providing robustness to parametric uncertainties. The SMC algorithm is a specific form of a variable structure control method where the control effort switches from one control structure to another depending on the system state trajectory location in the state or phase plane [7]. The control law is designed to drive the system trajectory states onto the desired trajectory where the difference between the current state values and the desired values defines a Sliding Surface. If the state trajectories are not on the Sliding Surface, a switching control law is used in the SMC scheme to force the trajectories toward the Sliding Surface (referred to as the Reaching Phase). Once on the Sliding Surface, the state trajectories "slide" towards the origin, and perfect tracking performance is theoretically achieved. However, the traditional SMC method has drawbacks in control chattering or high-frequency switching of the control system since the state trajectories are not perfectly on the sliding surface due to parametric uncertainties caused by the discontinuous term within the algorithm. The discontinuous term forces the state trajectories in the proper direction depending on if the trajectories are below or above the Sliding Surface. Applying boundary layers around the Sliding Surface is a method to reduce the chattering caused by control law.

The reliance on an accurate system model as well as tuning of control parameters can lead to less-than-optimal results on hardware implementations leading to the development of a Model-Free Sliding-Mode Control (MFSMC) algorithm. Crassidis and Mizov [10] developed a MFSMC algorithm applied to a first-order and second-order linear and nonlinear system and achieved satisfactory tracking performance with global asymptotic tracking stability guaranteed using the SMC method as a basis for the MFSMC development. Due to the complex nature of the flapping wing aerial system, control of the flapping wings under aerodynamic loading and pilot input requires precise control of the actuators. Using a MFSMC approach may solve the control requirement and is to be considered here.

2.0 LITERATURE REVIEW

2.1 Previous Flapping Wing Unmanned Aerial Systems

Sachs [1] compared the power requirement of flapping and fixed-wing aerial vehicles. The flapping wing induces significantly more drag compared to fixed-wing flight. The higher induced drag results from the tilting lift vector and the change in magnitude of the vector at different points in the flapping cycle. In fixed-wing vehicles, the propeller efficiency greatly affects the overall system efficiency. In this paper, it was determined that the propeller efficiency decreases proportionally as the vehicle size is also decreased. Concluding that for a smaller aerial vehicle, flapping wings provide higher efficiency flight.

Flapping Wing Micro Air Vehicles (FWMAV) exhibit advantages over traditional aerial vehicles in their high maneuverability, quick transition between flight modes, robustness to crashes, and smaller profile [2]. The typical mass of a FWMAV is a few grams. De Wagter et Al. created a 20 gram FWMAV that was able to carry a 4 gram sensor payload. The vehicle features a max velocity of 25 kilometers per hour, a turn rate of 500 degrees per second, and the ability to perform maneuvers such as flips and high-speed turns while remaining stable. However, the control design utilized external vision sensing to control the position and attitude of the vehicle. For an application outside of a laboratory setting, the control system as well as the state measurement algorithm must be included in the onboard hardware.

Croon et Al. [3] laid a road map for the development of a flapping wing micro aerial vehicle. In 2005, the DelFly I was designed featuring a wingspan of 35 centimeters and a mass of 21 grams. However, the DelFly I was not capable of forward flight. The DelFly II, created in 2008, had a reduced wingspan of 28 centimeters but could fly at a forward velocity of 7 meters per second, hover, and fly backward. The work lays a foundation for the construction of a FWMAV and explores mechanical design and material selection, electronics and power distribution, actuator selection, wing design and aerodynamics, and experimentation. While not everything covered in this book will be explored here, it will be used as a reference for the simulation and experimentation of the FWUAS developed in this paper.

Ning [4] sought to control the position and attitude of a FWMAV. The traditional method of deriving a system model, linearization, then designing a linear controller would not be suitable due to complexities due to uncertainty, nonlinearities, and multi-coupled parameters. Quaternion representation was used to account for gimbal lock within the state feedback structure. An adaptive fuzzy control law and an H_∞ control law were derived to estimate the unknown parameters and attenuate external disturbances. An "idealized" system model was created to test the control laws. However, it was noted this model may not accurately represent the dynamics due to neglecting the effect of flexible wings on the aerodynamic forces

and torques. While the controllers successfully stabilized the system and minimized tracking errors, the control law was not tested experimentally.

Jafferis et Al. [5] developed the first self-sustained piezoelectrically actuated flight. RoboBee utilized an X-wing design and weighed less the 500 milligrams with a wingspan of 5 centimeters. Two bimorph actuators were used but they were controlled by a single driving signal to reduce the weight required by the extra electrical components. The use of a photovoltaic array eliminated the added weight of batteries but only provide sufficient power under UV light that was 3x the density of sunlight. The ultra-light weight of this system prevents onboard sensors and cameras from being implemented.

O'Rourke [6] used experimental and theoretical analysis methods to study the lift and drag effects of a piezoelectric-driven flapping wing motion. The bending motion of the piezoelectric element negates the need for a complicated gearing mechanism that is common with motor-driven flapping wing vehicles and is susceptible to clogging and wear in adverse conditions. The work explored the feasibility of larger, forward flight systems and develops the framework for the development of a piezoelectrically actuated flapping wing aerial system. The Python PBA6014-5H200 PZT was used which had a maximum stroke of ~3.04 millimeters and a blocking force of 0.14 Newtons. A diagram of the multi-layered bimorph is shown in Figure 1. A simple wing using carbon fiber tubes and Mylar was constructed and tested under different flapping frequencies and phase shifts between the front and back wing spars. Resonance-like behavior was observed between 14 and 16 Hz producing the maximum lift and thrust. While flapping between 11 and 16 Hz, phase shifts of 30^o, 60^o, and 90^o were found to maximize lift and thrust generation. While the results indicated the current design would only be able to support a payload of 6 grams, further research into wing design, actuation mechanism, and control could lead to more desirable flight characteristics.

2.2 Sliding Mode Control

Jing et Al. [7] sought to find a robust control algorithm for the variable speed control of a wind turbine. To achieve the highest power output, the turbine must perform at peak efficiency at all wind speeds. Nonlinearities arising from parameter uncertainties, electromechanical coupling, and external disturbances led to a complex control problem. For this reason, a Quasi-Continuous High-Order Sliding Mode Controller was developed. While the design of the controller was based on a linearized system model, robust control of the turbine was achieved while also eliminating the chattering effect on the control input which is common with SMC techniques. While the approach achieved high robustness compared to a traditional PID control strategy, the reliance on a linear approximation of the model for control derivation limits its usefulness.

Slotine and Li [8] eliminated the chattering effect of SMC by including a smoothing boundary layer. The boundary layer adds a low-pass filter-like structure to the local dynamics around the sliding surface. They redefined the control law by including a Saturation function instead of the traditional Signum function. While this approach does not guarantee perfect tracking, the control law significantly reduces chattering.

2.3 Model-Free Control Methods

Madadi, Dong, and Soffker [9] evaluated the control of nonlinear systems and the unknown effects that are to be approximated in parallel with the control law. The criteria for assessing the performance of a Model-Free Control (MFC) law is in its' robustness against unknown system parameters and disturbances and the achieved tracking performance. In [10], two model-free control approaches are summarized and evaluated. An Intelligent Proportional-Integral Controller (iPI) and a Model-Free Adaptive Control (MFAC) algorithm are implemented on a three-tank system. The iPI algorithm describes a local model used to approximate unknown system dynamics in a short time window. The system parameters are updated at every time interval to provide an accurate estimate of the unknown dynamics. When compared with a fuzzy controller and a classical PI and PID, the iPI algorithm yielded more accurate and smoother tracking given different types of input signals. The MFAC algorithm approximates the Pseudo Partial Derivative (PPD) using online input-output data. The PPD algorithm is used at each time interval to build a working model instead of identifying the model parameters of a nonlinear plant. One major difference between iPI and MFAC is iPI uses offline data to build a dynamic model while MFAC uses online data to adapt to variations within the plant and control system. The iPI controller was also shown to be unaffected by time delays within the system while MFAC requires additional changes to be made to accommodate the effects. Like SMC, the iPI algorithm induces a chattering control effect caused by the use of the differentiator within the control law. The chattering is exacerbated by the fast actuator and sensor dynamics which will be present in the control of the PZT.

2.4 Model-Free Sliding Mode Control

Crassidis and Mizov [10] applied a control law based on SMC for linear and nonlinear systems that are grounded in observable measurements of the states of the system and the assumed order of the system. A switching gain is implemented to achieve stability according to Lyapunov's Direct Method. The chattering control effort, which is a common aspect of SMC, was eliminated by implementing a smoothing moving boundary layer. In real systems, state measurement noise must be accounted for and was determined outside the scope of this work. While different filtering and differentiation methods can be used, for application on a real-world system, the control law must be made robust to these elements.

Reis [11] developed a model-free control scheme based on the SMC method. The developed control effort is a function of the previous control inputs and state measurements. However, knowledge of the system order and the bounds of the control input gain must be known. To ensure asymptotic tracking stability, Lyapunov's stability theorem was used to derive the switching gain. The first case tested utilized a relay (signum) function to apply the SMC law discontinuous term. The relay function caused high-frequency "chatter" of the control effort. The chattering caused poor tracking of the higher-order states of the system. To reduce the chattering, a smoothing boundary layer was introduced in place of the relay function. The high-frequency chattering was eliminated and closed-loop asymptotic stability was guaranteed. The MFSMC technique proposed exhibited excellent tracking for both unitary and non-unitary control input gains. However, in the presence of state measurement noise, the MFSMC parameters chosen led to noisy state tracking performance. Furthermore, tuning the gain parameters was required to achieve acceptable tracking results. Additional filters and noise-reduction techniques can be implemented within the control

law to reduce the noise. For reliable performance, the bounds of the system uncertainties must be large, yielding excessive control effort within the system. To close the gap, the control input gain can instead be approximated by a parameter estimation technique considered here.

Schulken [12], expanded on the work done for MFSMC by simulating different aerial system models and disturbances such as the effects of discretization and state estimation. The MFSMC law and a linear PID Control law were examined. The performances were evaluated in terms of the quadratic mean of the tracking error, implementation complexity, and control law tuning. The MFSMC law exhibited similar tracking results as the PID controller but without the extensive tuning and testing process. However, both control laws were deemed unacceptable for real-world applications due to unsatisfactory tracking. The large tracking errors were attributed to the uncertainty in the state estimation algorithm. For the application of this algorithm on a real system, improvement in the state estimation process must be considered.

Sreeraj and Raj [13] worked to improve the previous work [12, 13, 14] performed by generalizing the control law for different classes of unmanned aerial systems. By simulating hardware such as the outputs of an Inertial Measurement Unit (IMU), operator input, noise, and sensor delays the work strived to develop a simulation that closely approached that of a real-world application. Improvements for the state estimation algorithm utilized an IMU placed at the center-of-mass of the aircraft. The full-state feedback required by the MFSMC lgorithmm was achieved. Calculation of the state derivatives in practical applications requires discrete smoothing filters. The filter ensures that all signals remain smooth and are twice differentiable. After the control laws were derived and successfully verified using simulation, they were implemented on a quadrotor system testbed. The experiment study showed better tracking performance of the MFSMC law and lower power consumption compared to the linear control law.

Monti [14] worked to expand on the application of a MFSMC law to a quadrotor system. Parameter uncertainties were examined by performing simulations while varying the mass and inertial properties. Three simulations were performed: original aircraft parameters, doubled mass, doubled moments of inertia, and doubled mass and moments of inertia. All of the test cases were simulated using the MFSMC and PID algorithms. Results of the simulation effort showed the MFSMC law achieved better tracking and lower power consumption than the linear control law. In this work, the control input gain matrix, B, was derived using knowledge of the system parameters. However, a scheme to account for not knowing the input gain must be formulated in achieving a truly model-free approach.

2.5 Model-Free Sliding Mode Control with On-Line Parameter Estimation

Islam [15] expanded the previous MFSMC law by using on-line parameter estimation to approximate the control input gain and the increment to the switching gain. To avoid gain unboundedness and maintain the benefits of weighting recent parameter calculations higher than previous values, a bounded forgetting tuning technique was implemented. The estimator assured the resulting estimated parameter is upper bounded by terminating the data forgetting if the upper bound is reached. The data-forgetting technique allowed for time-varying parameters including actuator degradation to be accounted for. Near perfect tracking was achieved for linear and nonlinear Multi-Input Multi-Output (MIMO) and Single-Input Single-

Output (SISO) systems. However, if the initial estimate of the input influence was far from the true value, the estimation technique was inadequate to achieve stable tracking convergence. The proposed work closes the previous research gap by estimating the gain parameter within the control law itself resulting in a parameter convergence including poor initial guess estimates of the input influence gain.

3.0 OBJECTIVES OF THE PROPOSED WORK

1. Compare control performance in terms of the root-mean-square error of the states of the system between linear control and Model Free Sliding-Mode Control.
2. Develop a Model Free Sliding-Mode Control algorithm for tracking control of a FWUAS.

4.0 Model-Free Sliding Mode Control Algorithm

The goal of designing a control system is to optimize the system's response to a given input in terms of the speed of response, stability, and accuracy. Classical methods rely on a derived system model to determine a control law. Most methods, such as a Proportional-Integral-Derivative (PID) controller, compensate the error between a desired and a measured state of the system. The gains are determined using a linear approximation of the system. Uncertainties and time-varying parameters within the system lead to suboptimal performance. More advanced techniques such a Sliding Mode Control (SMC) feature increased robustness and the ability to control higher-order nonlinear dynamic systems with uncertain conditions. However, like the classical control method, a reliance on an accurate system model is an inherent drawback. A model-free control algorithm utilizes a control law derived without the need for a system model. The only requirements being knowledge of the system order and accurate state estimates of the system. For a system that is highly nonlinear with many modeling uncertainties, such as a flapping-wing system, a model-free algorithm will provide more stability and robustness.

4.1 Sliding Mode Control

A single input, single output system is defined [10]:

$$x^n = f(x) + b(x)u \qquad (4.1)$$

for the system order n, the state variable x, the control input gain $b(x)$, and $f(x)$, defined as a function of the system states. The tracking trajectory $x_{d,}$, where x_d (0) = x (0), can be used to determine the tracking error of the system:

$$\tilde{x} = x - x_d \qquad (4.2)$$

The time-varying sliding surface is dependent on the order of the system and characterized by:

$$s = (\frac{d}{dt} + \lambda)^{n-1}\tilde{x} \qquad (4.3)$$

For the system to remain stable, the system must remain on the sliding surface. Thus when the tracking error is zero, the sliding surface, s, will be driven to zero. To satisfy this condition, a control law, u, is derived to satisfy the sliding condition [8]:

$$\frac{1}{2}\frac{d}{dt}s^2 \leq -\eta|s| \tag{4.4}$$

where η, is a strictly a positive constant. The control law drives the trajectory onto the sliding surface and maintains the system within the bounds of the sliding surface.

As a example, consider a second-order system to be controlled as:

$$\ddot{x} = f + u \tag{4.5}$$

where *f* is a function dependent on the system states and time. If the bounds are known, a best estimate, $\hat{f}$, can be approximated.

For a second-order system, the sliding surface is:

$$s = \dot{x} - \dot{x}_d + \lambda(x - x_d) \tag{4.6}$$

To ensure no movement of the system states on the sliding surface once the state trajectories reach the sliding surface, the first derivative, $\dot{s}$, is set equal to zero.

$$\dot{s} = \ddot{x} - \ddot{x}_d + \lambda\dot{\tilde{x}} = 0 \tag{4.7}$$

From Eq. (5.5) and (5.6), the estimated control, $\hat{u}$, is defined:

$$\hat{u} = -\hat{f} + \ddot{x}_d - \lambda\dot{\tilde{x}} \tag{4.8}$$

In order to satisfy the sliding condition:

$$s\dot{s} = \frac{1}{2}\frac{d}{dt}s^2 \leq -\eta|s| \tag{4.9}$$

a discontinuous Signum function, *sgn*( ), is added to the control law. The Signum function outputs a value of positive or negative one depending on the sign of the input. The updated control law becomes:

$$u = -\hat{f} + \ddot{x}_d - \lambda\dot{\tilde{x}} - K * sgn(s) \tag{4.10}$$

To satisfy the sliding condition in the most conservative manner, the minimum value of *k* is determined for the largest bound of *f* by:

$$K \geq \left|f - \hat{f}\right| + \eta \tag{4.11}$$

The control law defined above exhibits chattering or high-frequency switching. Chattering is unsuitable for applications on real systems which can lead to unsatisfactory performance and actuator degradation. A time-varying smoothing boundary layer is included in the control law to satisfy the new sliding condition [8]:

$$|s| \geq \varphi \ \rightarrow \frac{1}{2}\frac{d}{dt}s^2 \leq (\varphi - \eta)|s| \tag{4.12}$$

The K*$sgn(s)$ term is now replaced by $(K - \dot{\varphi}) * sat(\frac{s}{\varphi})$. The $sat(\ )$ function is defined by:

$$sat(y) = y \ if \ |y| \leq 1 \tag{4.13}$$

$$sat(y) = sgn(y) \ \ if \ else$$

The boundary layer is a first order differential equation:

$$\dot{\varphi} = -\lambda\varphi + K = -\lambda\varphi + \left|f - \hat{f}\right| + \eta; \ \varphi(0) = \frac{\eta}{\lambda} \tag{4.14}$$

Mizov [10] sought to control the following second order nonlinear system:

$$\ddot{x} = f + u \tag{4.15}$$

where:

$$f = -a(t)\dot{x}^2 \cos(3x) \ and \quad 1 \leq a(t) \leq 2 \tag{4.16}$$

The results below show the effect that boundary layer inclusion has on the effectiveness of the control law. Figure 2 clearly shows the high-frequency chatter.

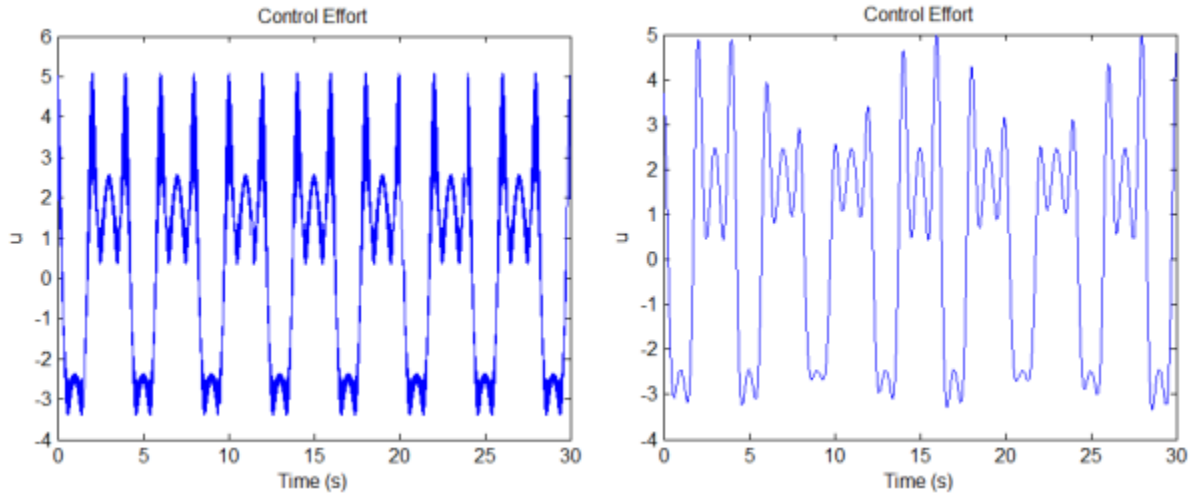


Figure 1: *Left – SMC without the inclusion of the boundary layer. Right – SMC with the inclusion of the boundary layer [10]*

4.2 Model-Free Sliding Mode Control Derivation for a Second Order System with Unitary Input Gain

For an n^{th} order SISO system, the system can be approximated in the discrete form as:

$$x^n \approx x^n + bu - bu_{k-1} \tag{4.17}$$

where u_{k-1} is the previous control input. When a unitary input gain is assumed, b is equal to one, following the same procedure outlined above, the estimated control law is defined as:

$$\hat{u} = -\left(\frac{d}{dt} + \lambda\right)^n \tilde{x} + u_{k-1} \tag{4.18}$$

and the updated control law with the inclusion of the smoothing boundary layer becomes:

$$u = -\left(\frac{d}{dt} + \lambda\right)^n \tilde{x} + u_{k-1} - (K - \dot{\varphi}) sat(\frac{s}{\varphi}) \tag{4.19}$$

where:

$$\dot{\varphi} = -\lambda\varphi + \eta;\ \varphi(0) = \frac{\eta}{\lambda} \tag{4.20}$$

4.3 Model-Free Sliding Mode Control Derivation for a Second Order System with Non-Unitary Input Gain

For a non-unitary input gain ($b \neq 1$), an n^{th} order SISO autonomous system can be defined as:

$$x^n = x^n + bu - bu_{k-1} - b\varepsilon(u) \tag{4.21}$$

where $\varepsilon(u)$ is the difference between the previous control input and the current control input.

The upper and lower limits of the control input gain are assumed to be known:

$$b_l \leq b \leq b_u \tag{4.22}$$

An estimation of the control input gain can be specified by the following:

$$\hat{b} = \sqrt{b_u b_l} \tag{4.23}$$

and an auxiliary variable, β, is defined to be used later to simplify equations later.

$$\beta = \sqrt{\frac{b_u}{b_l}} \tag{4.24}$$

In order to avoid an algebraic loop within the control law, an estimate of the control error, $\hat{\varepsilon}(u)$ is defined:

$$\hat{\varepsilon}(u) = u_{k-2} - u_{k-1} \tag{4.25}$$

While the control input error is unknown, the estimation is assumed bounded between:

$$(1-\sigma_l)\hat{\varepsilon}(u) \leq \varepsilon(u) \leq (1-\sigma_u)\hat{\varepsilon}(u) \tag{4.26}$$

To insure closed-loop asymptotic stability, the following inequality must be satisfied:

$$s\dot{s} \leq -\eta|s| \tag{4.27}$$

Through Eq. (5.27), the following control law is derived:

$$u = \hat{b}^{-1}\left[-\left(\frac{d}{dt}+\lambda\right)^n \tilde{x} + u_{k-1} - (K-\dot{\varphi})sat(\frac{s}{\varphi})\right] + 2u_{k-1} - u_{k-2} \tag{4.28}$$

where the switch gain, *K*, is characterized by:

$$K = |\dot{x}-\dot{x}_d||\beta-1| + \lambda|x-x_d||\beta-1| + \left|\hat{b}\sigma_u(u_{k-2}-u_{k-1})\right| + \beta\eta \tag{4.29}$$

The boundary layer is also redefined as:

$$\dot{\varphi} = -\lambda\varphi + |\dot{x}-\dot{x}_d||\beta-1| + \lambda|x-x_d||\beta-1| + \left|\hat{b}\sigma_u(u_{k-2}-u_{k-1})\right| + \beta\eta; \quad \varphi(0) = \frac{\eta}{\lambda} \tag{4.30}$$

4.4 Asymptotic Stability of the Controller

Lyapunov's Direct method will be applied to controller using a candidate Lyapunov function with the goal of achieving asymptotic stability during the reaching phase. The basis of Lyapunov's Direct Method is to ensure that the total energy of the system dissipates with time. The stability of the system can be evaluated by studying the energy variation as the system converges to an equilibrium point. An equilibrium point is determined asymptotically stable if it is both stable and convergent. The Lyapunov function should be chosen such that it is positive definite in *D* if the following conditions are satisfied [16]:

$$0 \in D \; and \; V(0) = 0 \tag{4.31}$$

$$V(\boldsymbol{x}) > 0 \; in \; D - \{0\} \tag{4.32}$$

By differentiating *V(x)* the systems change in energy with respect to time may be evaluated.

Let $\boldsymbol{x} = 0$ be an equilibrium point of the system: $\dot{x} = \boldsymbol{f}(\boldsymbol{x})$ where $f : D \rightarrow R^n$, and let $V:D \rightarrow R$ be a continuous and differential function such that:

$$V(0) = 0 \tag{4.33}$$

$$V(\boldsymbol{x}) > 0 \; in \; D - \{0\} \tag{4.34}$$

$$\dot{V}(\boldsymbol{x}) < 0 \; in \; D - \{0\} \tag{4.35}$$

If the above conditions are satisfied, the equilibrium point is considered asymptotically stable. A candidate Lyapunov Function is chosen as"

$$\boldsymbol{V(x) = \frac{1}{2} s^2} \tag{4.36}$$

which is considered positive definite and radially bounded. Differentiating Eq. (##) results in:

$$\dot{V}(\boldsymbol{x}) = \dot{s}s \tag{4.37}$$

Substituting the control law from Eq. (4.28) into Eq. (4.37) and defining it as strictly negative yields:

$$\dot{V}(\boldsymbol{x}) = \boldsymbol{s}(\hat{b}^{-1}\left[-\left(\frac{d}{dt} + \lambda\right)^n \tilde{x} + u_{k-1} - (K - \dot{\varphi})sat(\frac{s}{\varphi})\right] + 2u_{k-1} - u_{k-2}) \; \leq 0 \tag{4.38}$$

After simplification:

$$\dot{V}(\boldsymbol{x}) = \boldsymbol{s}[-(K - \dot{\varphi})sat\left(\frac{s}{\varphi}\right)] \leq 0 \tag{4.39}$$

The *sat* function is negative unitary when the sliding surface is negative and positive unitary when the sliding surface is positive so $\boldsymbol{s} * sat\left(\frac{s}{\varphi}\right)$ can be replaced by |**s**|.

$$\dot{V}(\boldsymbol{x}) = \; -(K - \dot{\varphi})|\boldsymbol{s}| \leq 0 \tag{4.40}$$

Concluding the derivative of the Lyapunov function is strictly negative since $(K - \dot{\varphi})$ is strictly positive. Therefore, the system is considered closed-loop asymptotically stable.

4.5 Model-Free Sliding Mode Control Derivation for a Second Order System with Online Parameter Estimation

The previously described approach assumed the upper and lower bounds of the input influence gain were known leading to an approach that is not identically model-free. In Islam [15], a Sliding Mode Model Free Control algorithm with on-line parameter estimation was derived to estimate the input influence gain based on satisfying the sliding condition. The approach assumes a non-unitary gain for the input influence parameter with unknown bounds for the updated control law to be implemented is shown as:

$$\hat{u} = \hat{b}^{-1}[-\ddot{\tilde{x}} - \lambda\dot{\tilde{x}} - (K - \dot{\varphi})sat\left(\frac{\boldsymbol{s}}{\varphi}\right)] + 2u_{k-1} - u_{k-2} \tag{4.41}$$

The switching gain, *K* is defined by:

$$K = \left|(\beta - 1)\ddot{\tilde{x}}\right| + \left|(\beta - 1)\lambda\dot{\tilde{x}}\right| + \left|\hat{b}\sigma_u(u_{k-2} - u_{k-1})\right| + \beta\eta \tag{4.42}$$

where the upper bound of the control input error estimate is represented by σ_u.

The ratio of the estimate of the control input gain (which can be constant or as a user-defined function of time) to the initial guess of the input gain is defined as β.

$$\beta = \frac{\hat{b}}{b} \tag{4.43}$$

4.6 Online Parameter Estimation Law

A least-squares optimization technique is used to estimate the control input gain of the system. The key is designing an appropriate model that relates the control input to the output of the system. For example, a linear model is shown below.

$$y(k) = bu(k) + e(k) \tag{4.44}$$

where $y(k)$ and $u(k)$ are the outputs and inputs of the system at time k, $e(k)$ is the measurement error and b is the parameter to be estimated. The Least-Squares technique adjusts the unknown parameter vector to minimize the sum squared error between the actual output and the predicted output. The initial value of the inverse covariance matrix is chosen as the identity matrix and the initial estimate (i.e., the input influence gain) is chosen to be between the upper and lower bounds specified by the user.

The Incremental change in the covariance matrix is defined as:

$$\tilde{P}(k) = \frac{P(k-1)*u(k)*u(k)*P(k-1)}{1+\lambda*u(k)*P(k-1)} \tag{4.45}$$

Where λ is the forgetting factor bounded between 0 and 1. The updated covariance matrix is calculated:

$$P(k) = \lambda^{-1} * (P(k-1) - \tilde{P}(k)) \tag{4.46}$$

The estimation error is now determined by:

$$e(k) = y(k) - \hat{b}(k-1) * u(k) \tag{4.47}$$

The updated control input gain estimate is calculated by:

$$\hat{b}(k) = \hat{b}(k-1) + \tilde{P}(k) * e(k) \tag{4.48}$$

The process is completed during every time step to provide an updated estimate of the control input gain.

4.7 Model-Free Sliding Mode Control for a MIMO Second Order System

The MFSMC algorithm was implemented on a second-order MIMO system to evaluate the performance of the method. The increment to the switching gain was estimated using the On-Line Parameter Estimation in the previous section assuming unitary input gains for the control law. This initial approach was used to test the feasibility of implementing On-Line Parameter Estimation algorithms for estimating the non-unitary input influence parameters in the proposed control law.

$$m_1\ddot{x}_1 - c_2(\dot{x}_2 - \dot{x}_1)|\dot{x}_2 - \dot{x}_1| - k_2(x_2 - x_1) + d_2(x_2 - x_1)^3 + c_1\dot{x}_1|\dot{x}_1| + k_1x_1 - d_1x_1^3 = b_1u_1 \quad (4.49)$$

$$m_2\ddot{x}_2 + c_2(\dot{x}_2 - \dot{x}_1)|\dot{x}_2 - \dot{x}_1| + k_2(x_2 - x_1) - d_2(x_2 - x_1)^3 = b_2u_2 \quad (4.50)$$

The reference signals are defined as:

$$x_{1,d}(t) = x_{2,d}(t) = \sin\left(\frac{\pi}{2}t\right) \quad (4.51)$$

The simulation was performed in Simulink using the fixed-step solver, *ode5* (Dormand-Prince), with a sample time of 0.001 seconds for 20 seconds. The control law and parameter estimation algorithm derived in the previous section were programmed in Simulink. The control parameters are defined below:

Controller parameters		η estimator parameters	
Parameter	Value	Parameter	Value
λ_1	4	λ_{0_1}	80
λ_2	3	λ_{0_2}	80
η_{0_1}	0.00060	P_{0_1}	10
η_{0_2}	0.00022	P_{0_2}	10
σ_{u_1}	0.2	K_{0_1}	500
σ_{u_2}	0.2	K_{0_1}	500

Table 1: *Control and estimation parameters of nonlinear MIMO system from [15]*

Figures 2 and 3 show the tracking errors and trajectories of all states. To evaluate the tracking ability of the control law the root-mean-square (RMS) of the tracking error was determined. The tracking errors are deemed negligible for all states and outstanding tracking performance is observed.

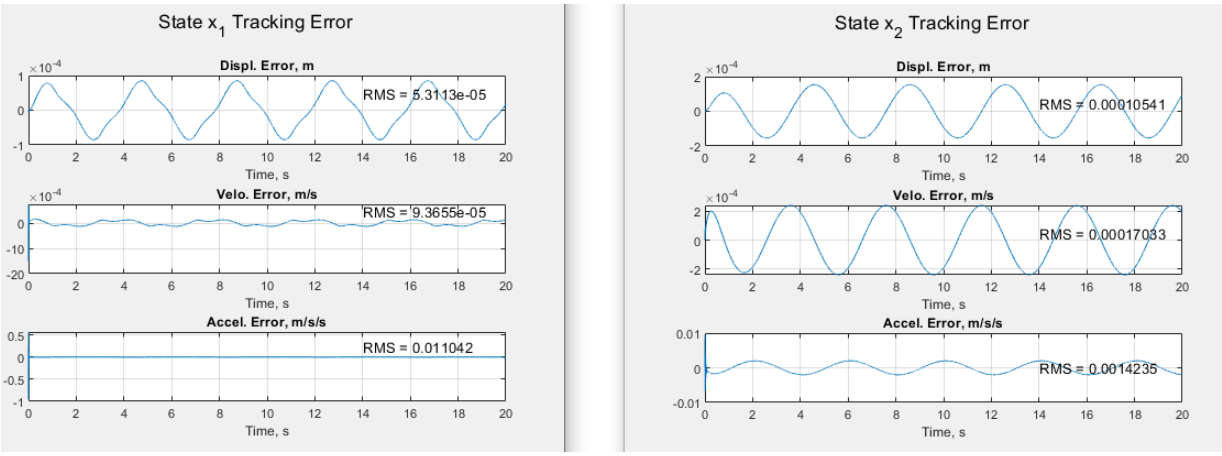


Figure 2: *Tracking error of states x_1 and x_2 for a nonlinear MIMO system*

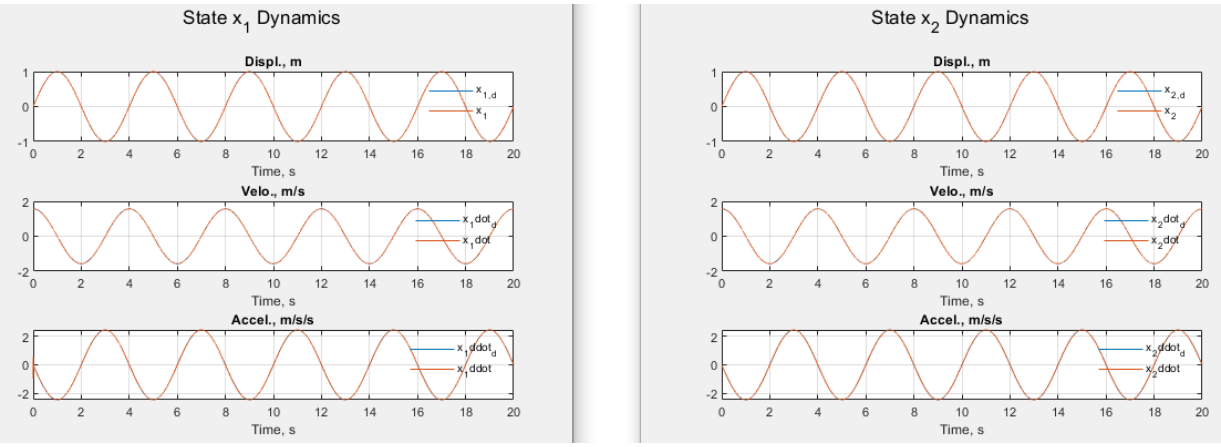


Figure 3: *Dynamics of states x_1 and x_2 for a nonlinear MIMO system*

The control effort and boundary layer dynamics are shown in Figure 4. The sliding surface remains within the boundary layer for all time. The control effort is shown to be smooth, indicating the effectiveness of the smoothing boundary layer in eliminating control chattering.

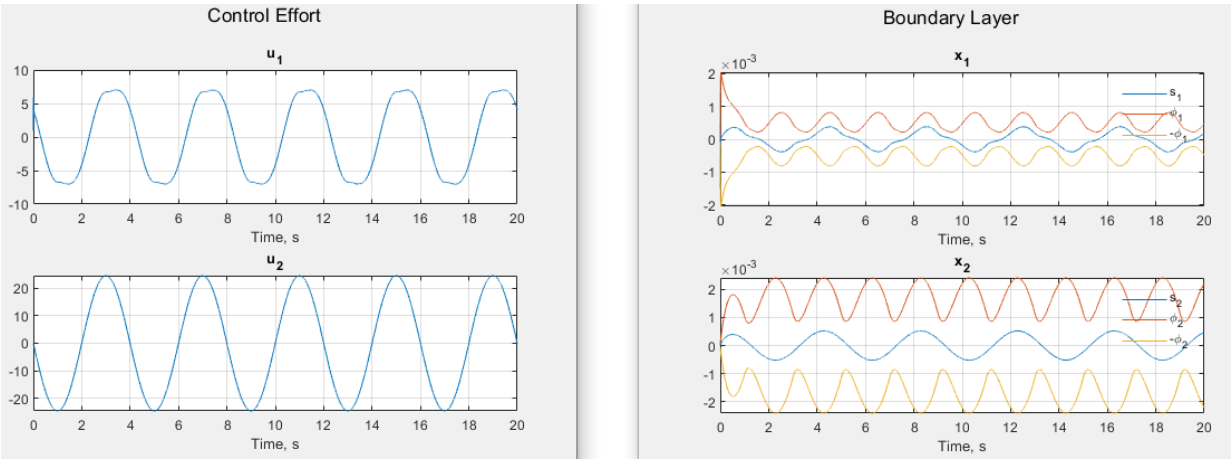


Figure 4: *Control effort (left) and Boundary Layer dynamics (right) of a nonlinear MIMO system*

Figure 5 shows the switching gain and estimated increment to the switching gain. The gains are estimated to drive the sliding surface back within the boundary layer.

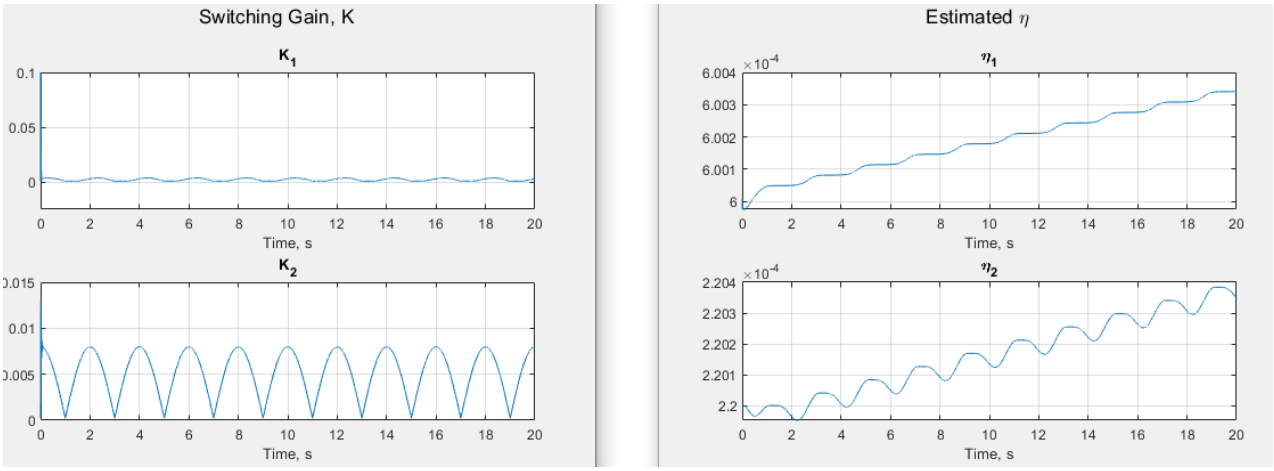


Figure 5: *Switching Gain (left) and Estimated increment to the switching gain (right) for a nonlinear MIMO system*

Error State	RMS	Error State	RMS
$\tilde{x}_1$	5.3113E-5	$\tilde{x}_2$	1.0541E-4
$\dot{\tilde{x}}_1$	9.3655E-5	$\dot{\tilde{x}}_2$	1.7033E-4
$\ddot{\tilde{x}}_1$	0.0110	$\ddot{\tilde{x}}_2$	0.0014

Table 2: *RMS of the tracking error of all states for a nonlinear MIMO system*

The results matched those found by Islam [15] and satisfied the tracking and stability requirements. However, this algorithm was not tested under the influence of state measurement noise. Further tuning of the control parameters for implementation on a real system. The parameters will be tuned according to the work done by Reis [11].

4.8 Model-Free Sliding Mode Control for a 2DOF Model of an Aerial Vehicle

A simplified model of the dynamics of a 2DOF aircraft in a hovering condition was created. Two linear second-order equations representing the altitude and roll were derived. The aircraft in the simulation features two actuators independently controlling a wing on either side of the fuselage. The goal of the simulation effort is to test the altitude and roll control of the MFSMC algorithm. A simplified free-body-diagram of the system model is shown in Figure 6.

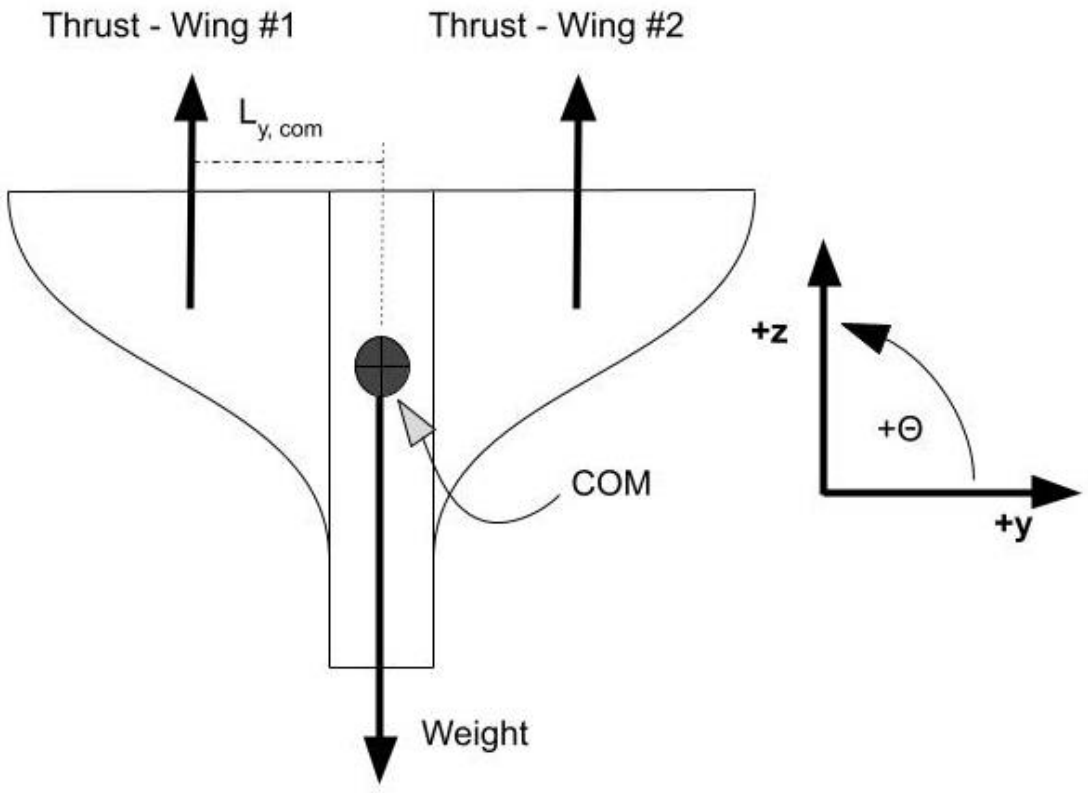


Figure 6: *Free Body Diagram of simplified 2DOF model of a flapping wing aerial vehicle*

The equations-of-motion for the 2DOF model are summarized next.

$$m\ddot{z} = T_{w,1} + T_{w,2} - W \tag{4.52}$$

$$J\ddot{\theta} = -L_{y,com}T_{w,1} + L_{y,com}T_{w,2} \tag{4.53}$$

The mass and inertial properties were chosen arbitrarily to demonstrate the ability of the control law to handle parameter uncertainties and are listed in Table 3.

Mass of aircraft, m [kg]	*1*
*Inertia about COM, J [kg*m^2]*	*1*

Table 3: *Mass and Inertial properties of the airframe used in simulation*

The parameters remain unchanged from the parameters used for the nonlinear MIMO system listed in Table 1. The integration parameters also remain unchanged. The desired reference inputs are:

$$z(t) = \sin\left(\frac{\pi}{2}t\right) \tag{4.54}$$

$$\theta(t) = 0.05 * \sin\left(\frac{\pi}{2}t\right) \tag{4.55}$$

The tracking error for the states is shown in Figure 7. The tracking is deemed acceptable for all states save the angular acceleration, $\ddot{\theta}$, has a significantly higher RMS than the other states. Further tuning of the control parameters can improve performance.

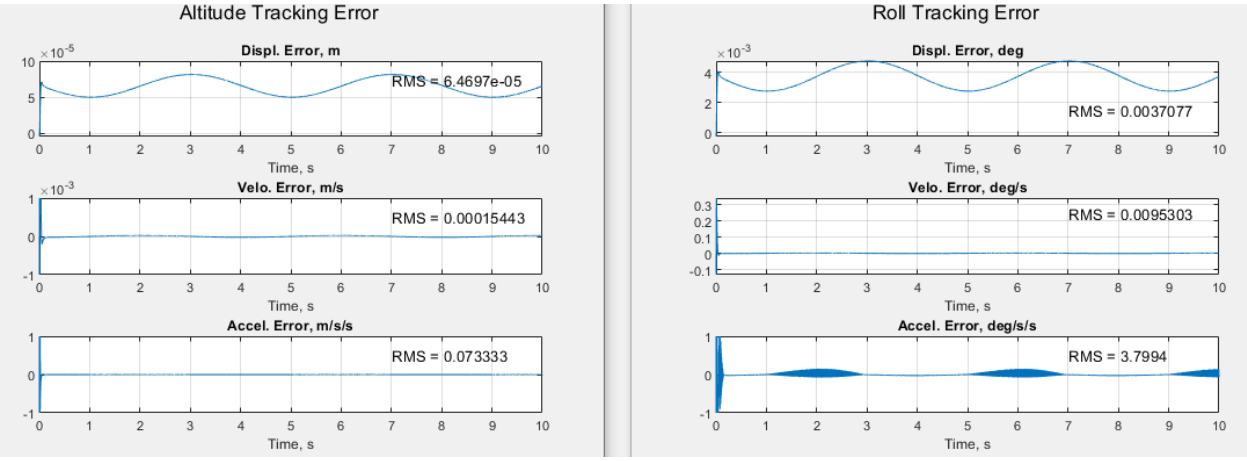


Figure 7: *Altitude (left) and Roll (right) tracking error for the simplified 2D model*

The dynamics of the system are shown in Figure 8. All states follow the desired tracking trajectories for all time. However, there is a large discrepancy at the start of the simulation for the acceleration states (to be expected since the states cannot be initialized properly). As time passes, the controller successfully drives the states to their desired trajectories.

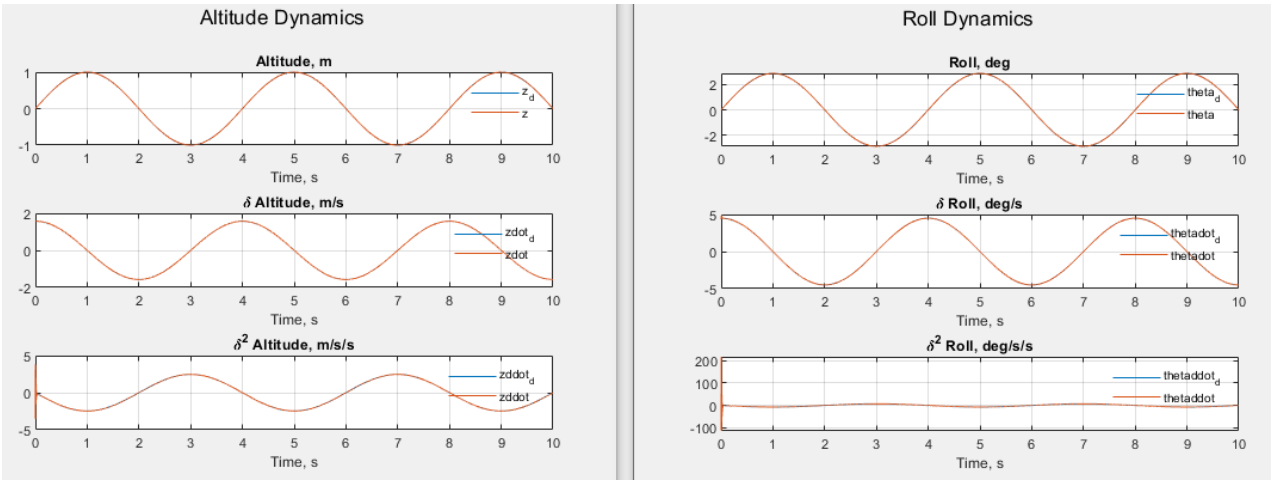


Figure 8: *Dynamics of states for simplified 2DOF model*

The control effort and boundary layer dynamics are shown in Figure 10. The chattering is successfully eliminated and the sliding surface stays within the boundary layer for all time.

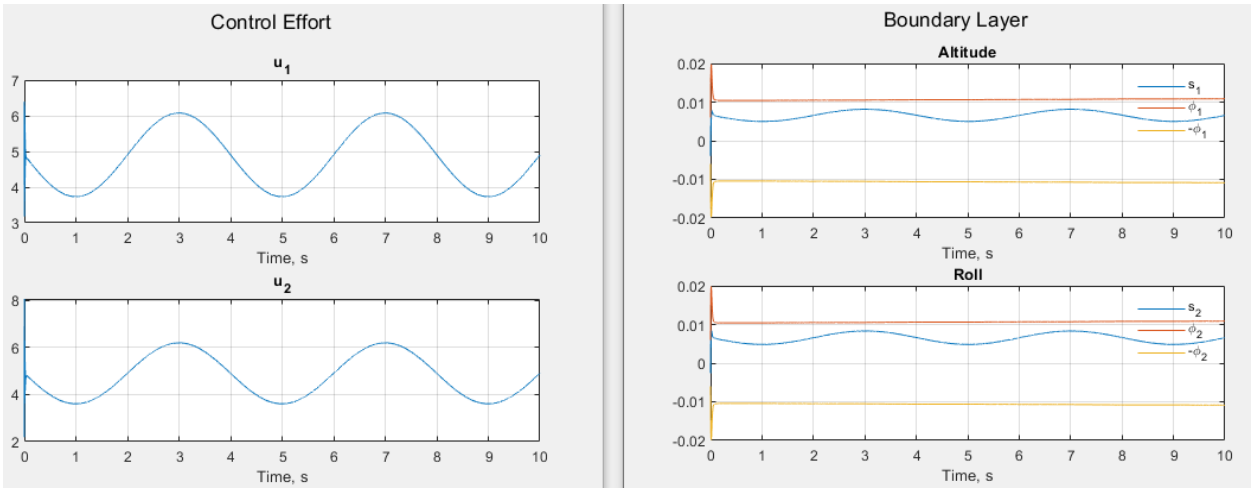


Figure 9: *Control effort (left) and boundary layer dynamics (right) of the simplified 2D model*

Figure 11 displays the switching gain and estimated increment to the switching gain. The estimate increment does not change significantly since the sliding condition is satisfied during the time period (see Figure 10). The gains are estimated to drive the sliding surface back within the boundary layer.

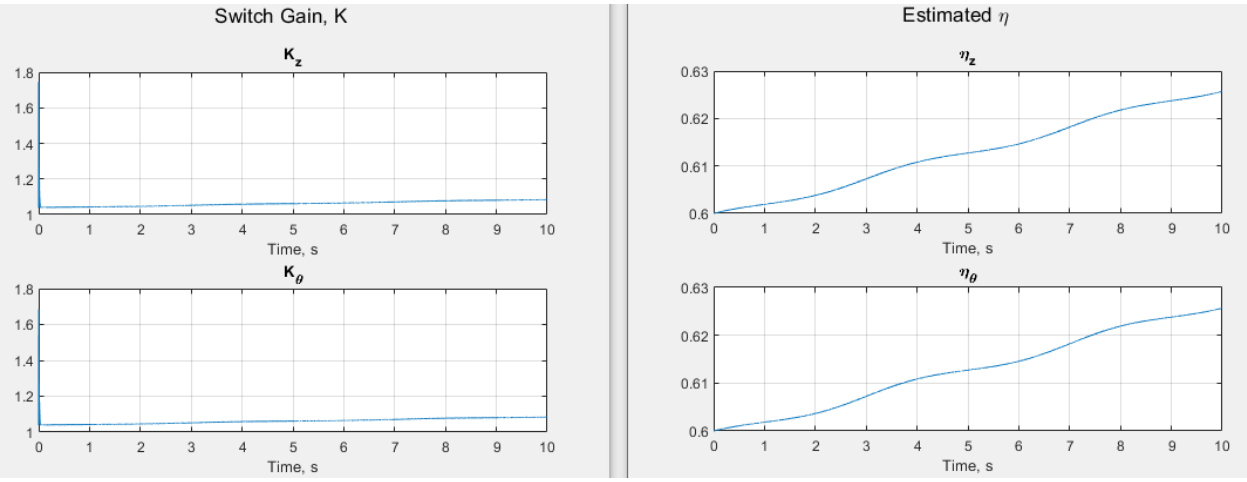


Figure 10: *Switching Gain (left) and Estimated increment to the switching gain (right) for a nonlinear MIMO system*

Table 5 displays the RMS tracking error for the states and are small save for the acceleration state. The large error is due initial condition mismatching and should not be observed in application.

Error State	RMS	Error State	RMS
$\tilde{z}$	6.4697E-5	$\tilde{\theta}$	0.0037
$\dot{\tilde{z}}$	1.5443-5	$\dot{\tilde{\theta}}$	0.0095
$\ddot{\tilde{z}}$	0.0733	$\ddot{\tilde{\theta}}$	3.7994

Table 4: *RMS of the tracking error of all states for a 2DOF simplified model*

5.0 Experimental Implementation of MFSMC

Testing of the MFSMC algorithm was done on a simplified system to evaluate the effectiveness of the control law. When deciding on a system, one was chosen that would be analogous to the final system. Thus, a balancing system consisting of a DC motor and propeller mounted to a servo for thrust vectoring was chosen.

5.1 Constrained Tiltrotor-Balancing System

The system, that the MFSMC law will be tested on, is a constrained tilt-rotor system. The inputs of the system are the voltage of propeller DC motor (i.e., velocity) and voltage to the servomotor allowing for thrust vectoring of the propeller. The outputs to be controlled are the pitch and yaw angle of the tilt-rotor system. To assess the robust nature of the model-free algorithm, a counterweight is placed on the opposite side and will be varied between tests effectively varying the parameters within the system. A BNO055 IMU was used for the state feedback with an SG90 servo motor and a hobby 5V DC motor and propeller.

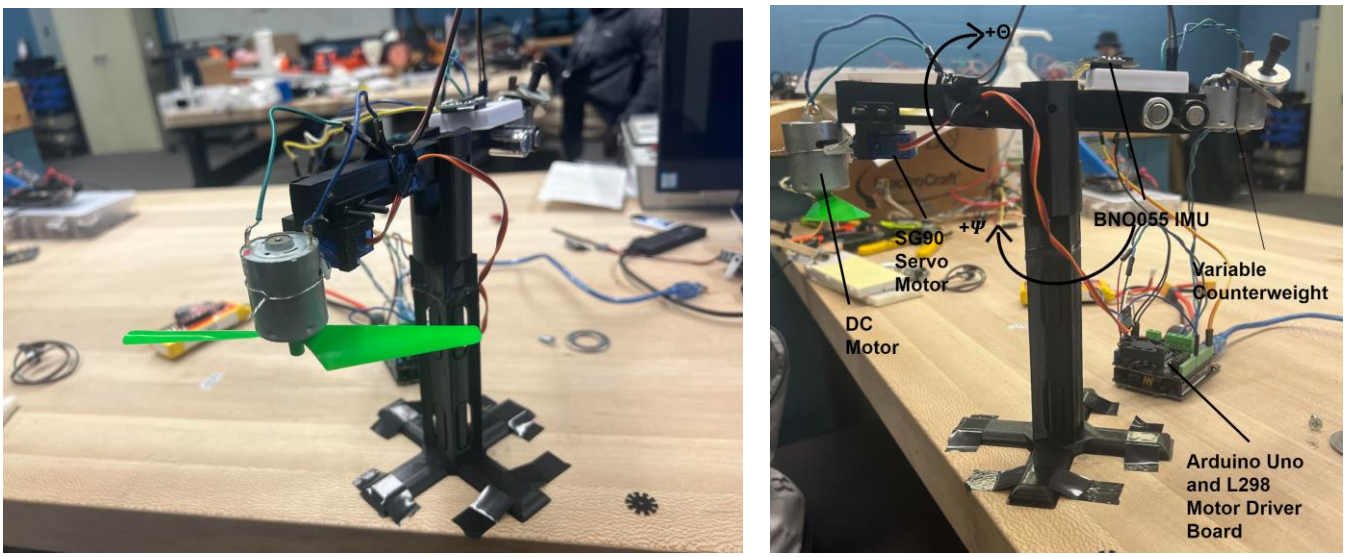


Figure 11: Image of the Tilt-Rotor System

5.1.1 System Description and Equations-of-Motion

Pictured below are the free-body diagrams of the system.

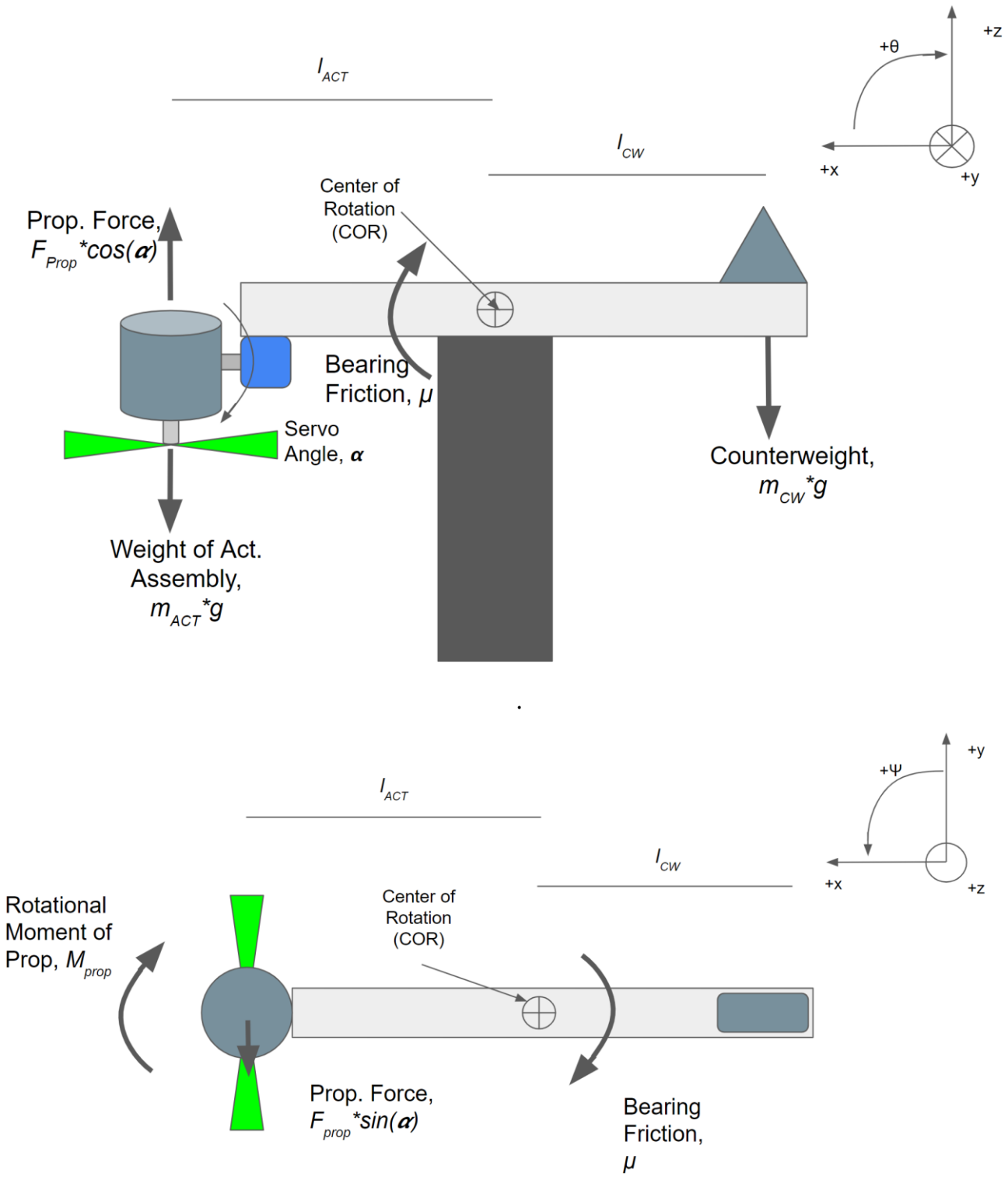


Figure 12: FBD of Tilt-Rotor System

The equations-of-motion are derived using Newton's Second Law for constant mass. The force and the generated moment exerted by the propeller is represented by F_{prop} and M_{prop}, respectively. The servo angle expressed as α, and the moment arms of the counterweight and actuator assembly are l_{CW} and l_{ACT}. The counterweight and actuator assembly masses are represented by m_{CW} and m_{ACT}. Bearing friction is also accounted for by μ. The mass of the beam is neglected so that:

$$\Sigma M_y = I_y\ddot{\theta} = m_{CW}gl_{CW} - m_{ACT}gl_{ACT} + F_{prop}\cos(\alpha)\, l_{ACT} - \mu\dot{\theta} \tag{5.1}$$

$$\Sigma M_z = I_z\ddot{\psi} = F_{prop}\sin(\alpha)l_{ACT} - M_{prop} - \mu\dot{\psi} \tag{5.2}$$

$$\ddot{\theta} = \left(\frac{1}{I_y}\right)[m_{CW}gl_{CW} - m_{ACT}gl_{ACT} + F_{prop}\cos(\alpha)\, l_{ACT} - \mu\dot{\theta}] \tag{5.3}$$

$$\ddot{\psi} = \left(\frac{1}{I_z}\right)[F_{prop}\sin(\alpha)l_{ACT} - M_{prop} - \mu\dot{\psi}] \tag{5.4}$$

5.1.2 Linear Control Law

Based on the model derived above, a linear control law was determined to compare against the MFSMC law. While many different model-based control algorithms would be sufficient for this system, a full-state feedback control law was chosen due to its high robustness. The first step in deriving the feedback gains is to linearize the system about a predetermined operating point. After linearization, the system can be converted into discrete state-space form. To derive the feedback gains, a Linear Quadratic Regulator scheme was used using Matlab's "dlqr" function. The function's input are the matrices from the state-space model and using the weighting matrices, the solution of the discrete algebraic Riccatti equation is determined. The states of the system to be controlled:

$$x = [x_1 \quad x_2 \quad \dots \quad x_8]' = [\theta \quad \dot{\theta} \quad \psi \quad \dot{\psi} \quad \int e_\theta dt \quad \int e_\psi dt \quad e_\theta \quad e_\psi]' \tag{5.5}$$

The discrete state, control, and feedback gain matrix are as follows:

A =

	x1	x2	x3	x4	x5	x6	x7	x8
x1	1	0.004999	0	0	0	0	0	0
x2	0	0.9998	0	0	0	0	0	0
x3	0	0	1	0.004999	0	0	0	0
x4	0	0	0	0.9998	0	0	0	0
x5	-0.005	-1.25e-05	0	0	1	0	0	0
x6	0	0	-0.005	-1.25e-05	0	1	0	0
x7	-0.004988	-1.248e-05	0	0	0	0	0.995	0
x8	0	0	-0.004988	-1.248e-05	0	0	0	0.995

B =

	u1	u2
x1	0.0001875	0
x2	0.07499	0
x3	0	0.0005625
x4	0	0.225
x5	-3.125e-07	0
x6	0	-9.374e-07
x7	-3.121e-07	0
x8	0	-9.363e-07

```
K =

    1.8031    1.0737   -0.0000   -0.0000   -0.9591    0.0000   -0.1160   -0.0000
   -0.0000   -0.0000    1.6310    0.9286   -0.0000   -0.8899    0.0000   -0.1122
```

Using the fixed step solver, *ode5*, with a time step of 0.005 seconds, the following results were determined. A white noise block was also used to simulate the measurement noise from the IMU. Through testing of the IMU, it was determined that a noise power of 0.0005 sufficiently approximated the sensor dynamics.

Figure 13 illustrates the observed state dynamics as well as the control effort. LQR is a state feedback control law, therefore the presence of high-frequency noise in the system leads to high-frequency switching of the control signal. The signal is not acceptable for implementation on a real system due to potential instability and actuator damage.

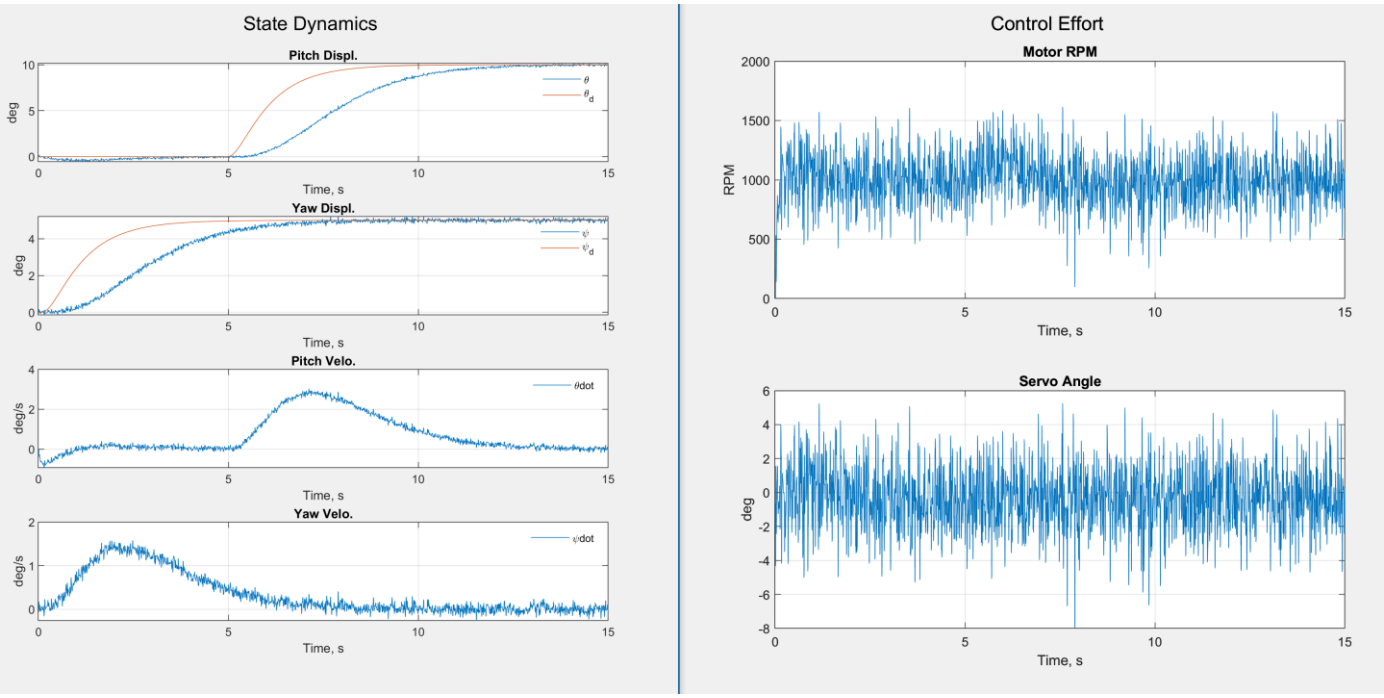


Figure 13 Thrust-Vector Balance State Dynamics and Control Effort

While the LQR controller successfully tracked the desired signal, the measurement noise generated a noisy control signal due to the state feedback nature of the control algorithm. In simulation, high frequency switching of the control input is not a problem but may lead to system instability during implementation on the real system.

Figure 14 displays the tracking error of the pitch and yaw states of the system. Velocity and Displacement errors are successfully minimized.

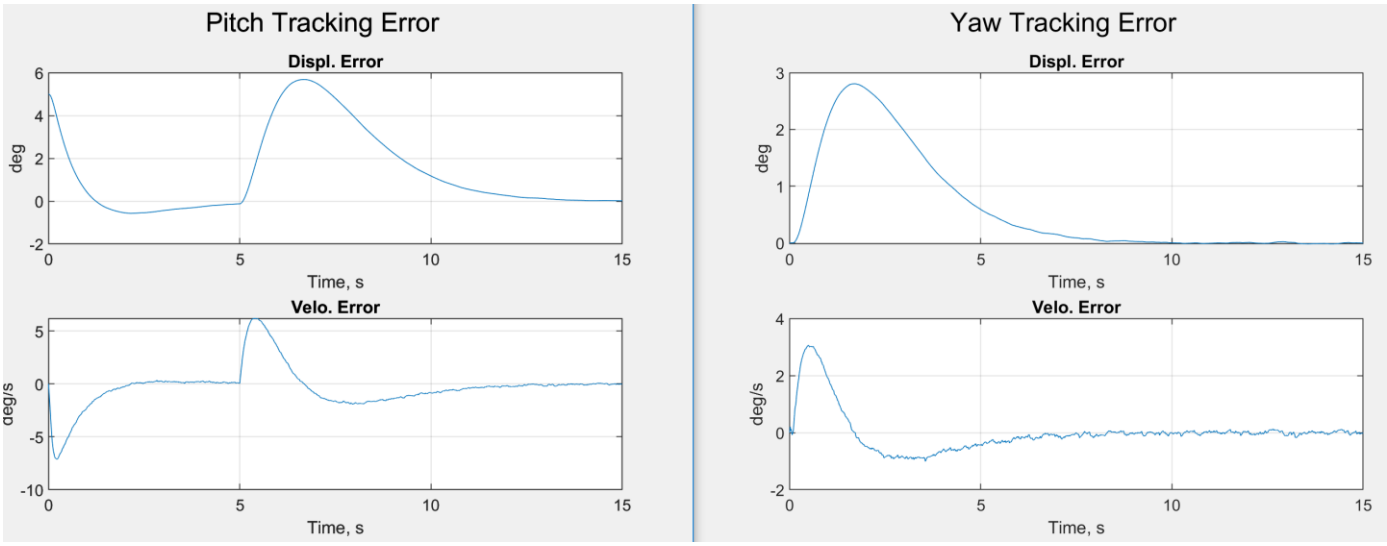


Figure 14: Thrust-Vector Balance Tracking Error

5.1.3 Control Architecture & Changes Made to MFSMC Algorithm

The inputs to the MFSMC algorithm are the pitch and yaw displacement, velocity, and acceleration. When implementing the MFSMC algorithm onto the real system, it was determined that the calculated control input was often too aggressive which led to system instability. To remedy this issue, a first-order low-pass filter was applied to the calculated control input before being fed to the actuator. In testing, an ideal range of actuator input values was determined. Finally, the filtered control input was mapped to the range of actuator values and written to the actuators. These two steps significantly increased the stability and performance of the system.

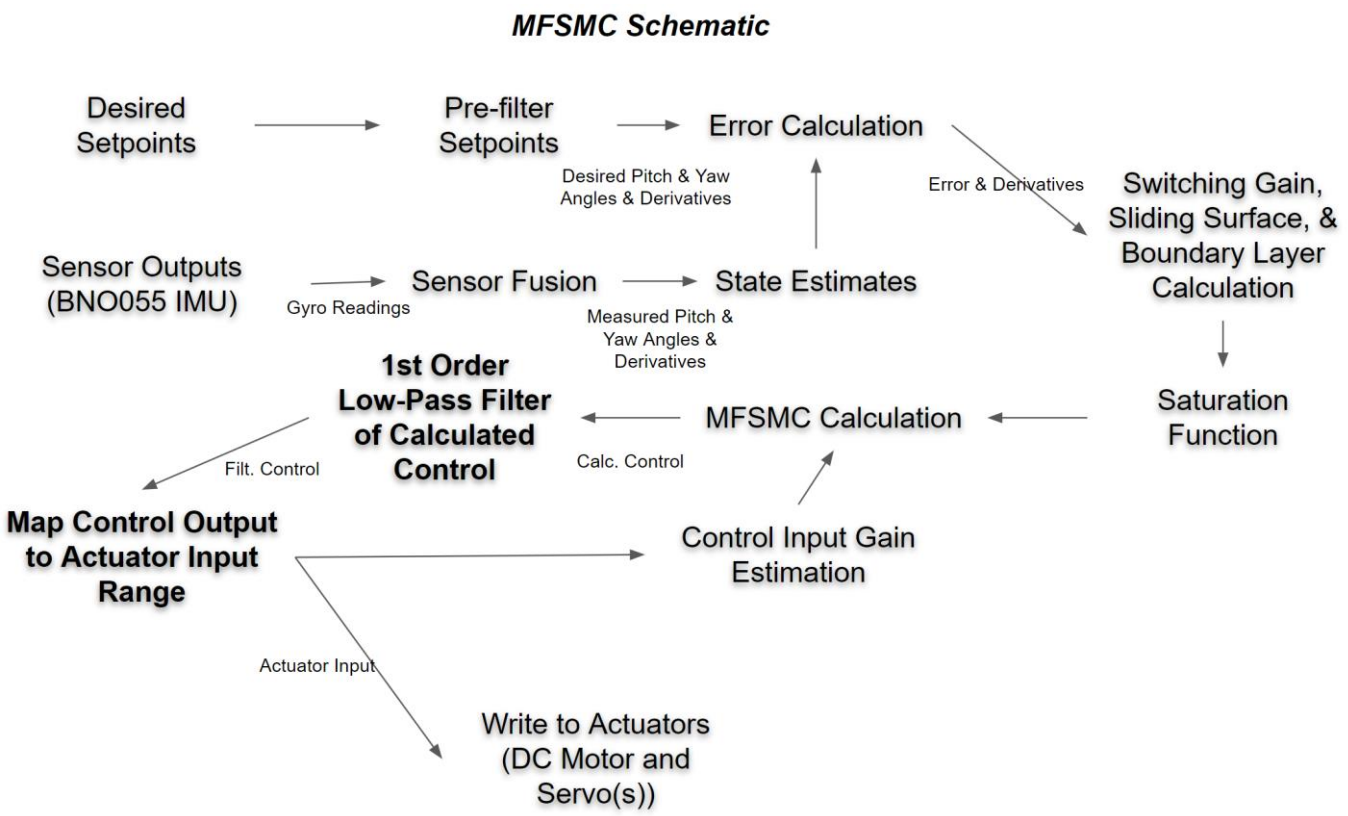


Figure 15: Schematic of the Control Scheme

To implement the 1st-order low-pass filter in discrete form, the following equation was used:

$$u_{Control} = (\alpha_u)u_{MFSMC,k} + (1 - \alpha_u)u_{MFSMC,k-1} \tag{5.6}$$

where *k* and *k-1* are the current and previously calculated control inputs and α_u is a constant between zero and one that can be tuned to vary the aggressiveness of the controller. The filter constant was chosen during testing and varies between different actuators. If the filter constant is closer to one, the controller will be more aggressive.

5.1.4 Simulation Results

The model-free control parameters are listed in the table below. The discrete solver, *ode5*, was chosen with a time step of 0.005. The step time of 5 milliseconds was chosen because that was the highest sampling rate that was able to be achieved on hardware. Increasing the time-step to 25 milliseconds had the tendency to cause the system to go unstable when the system was simulated.

Controller		Estimator	
Parameter	**Value**	**Parameter**	**Value**
λ_1, λ_2	1, 1.25	$\widehat{b_{1,0}}, \widehat{b_{2,0}}$	0.8, 0.8
η_1, η_2	0.225, 0.225	$\lambda_{0,1}, \lambda_{0,2}$	80, 80
σ_1, σ_2	0.025, 0.025	$k_{1,0}, k_{2,0}$	500, 500
$\alpha_{u,1}, \alpha_{u,2}$	0.21, 0.21	$P_{b1,0}, P_{b2,0}$	10, 10

Table 5: Controller and Estimator Parameters for Thrust-Vector Balance Simulation

Below are the simulation results of the MFSMC algorithm on the thrust-vector balancing system.

Figure 16 displays the state dynamics for pitch and yaw. While initially, the states do not track perfectly, over time they converge to the reference inputs.

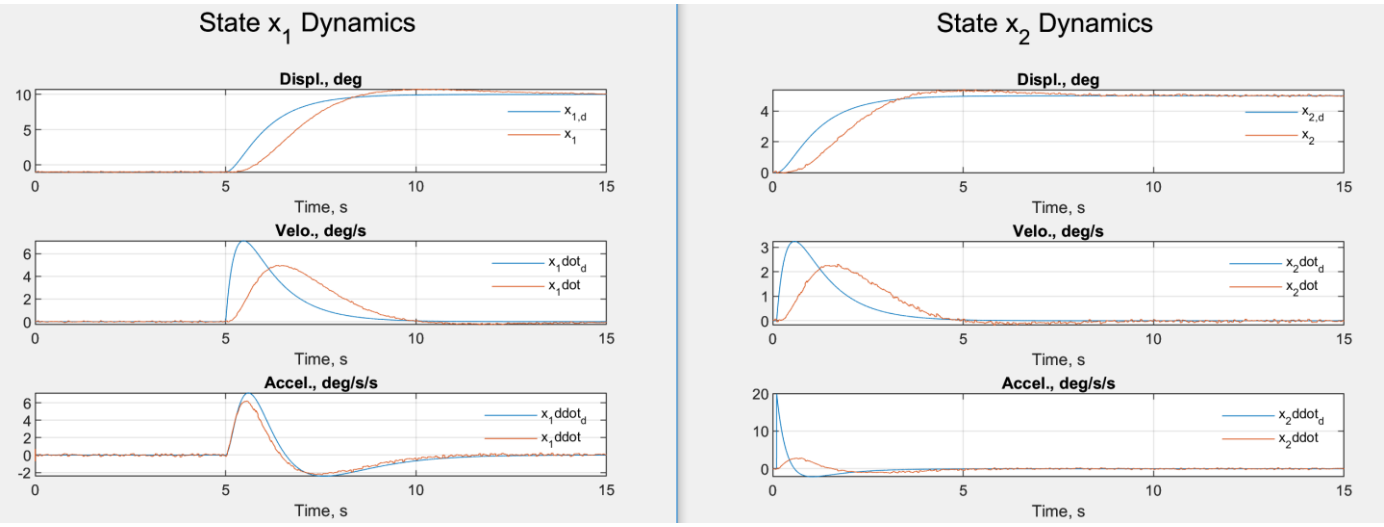


Figure 16: Pitch (x_1) and Yaw (x_2) state dynamics

Figure 17 shows the successful minimization of tracking error for all states. The effect of the state measurement noise can be seen here.

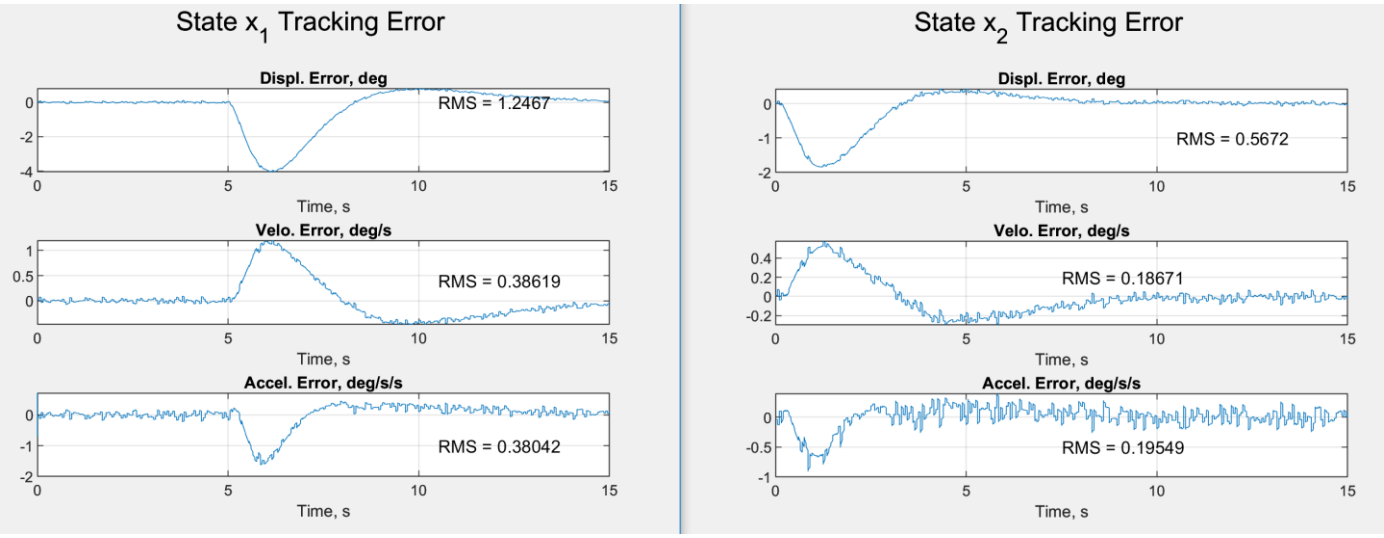


Figure 17: Pitch (x_1) and Yaw (x_2) state tracking errors

The boundary layer dynamics and estimated control gain are shown below. While the sliding surface does not remain within the boundary layer for all time, it converges quickly. Further tuning of λ, the slope of the sliding surface, could improve the convergence dynamic. However, the current value was deemed satisfactory for the simulation. The control input gain converges quickly to the true value, showing the effectiveness of the estimating algorithm.

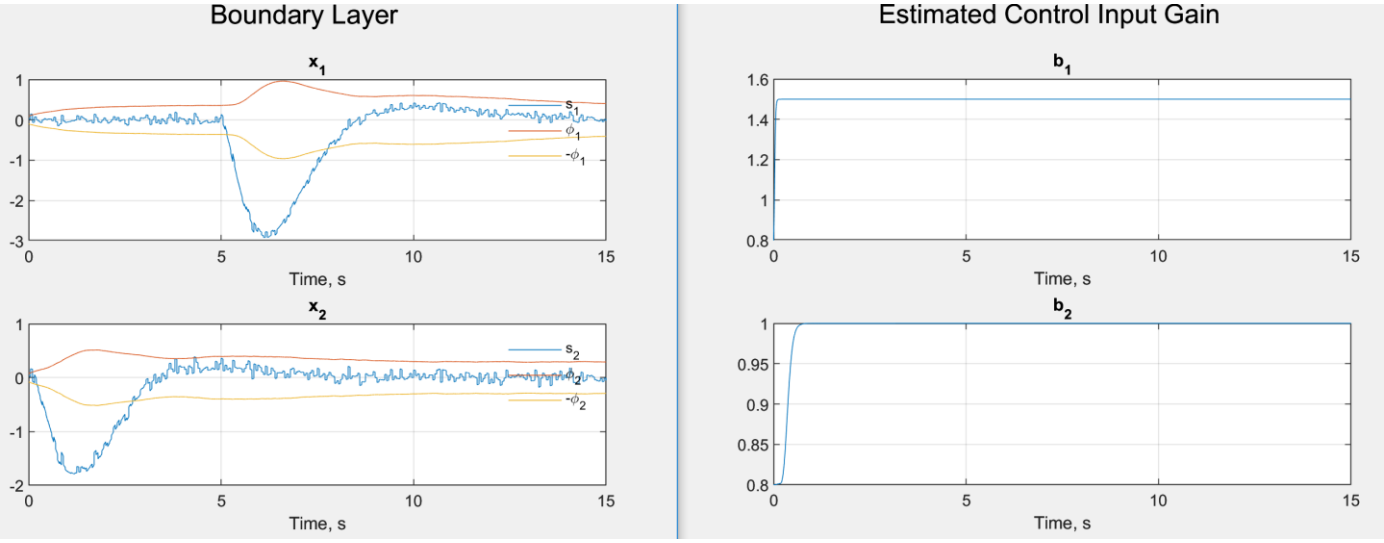


Figure 18: Boundary Layer Dynamics & Estimated Control Input Gain

Figure 19 shows the control effort for the motor and servo. The effect of the additional low-pass filter and input mapping can be seen here.

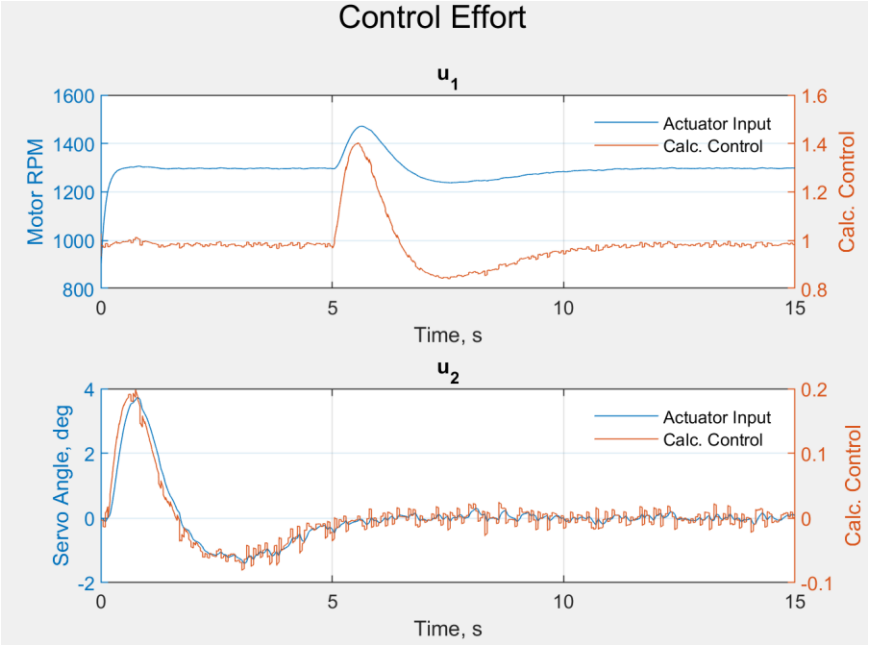


Figure 19: Control Effort for MFSMC on the Tilt-Rotor System

The root-mean-square of the error was used to evaluate the ability of each controller to track a desired trajectory. Other methods such as the "integral-square-error" (ISE) and the determination of the settling time, rise time, and steady state error could have also been used. However, RMS was used for the testing done here. It can be noted that the MFSMC significantly outperformed the LQR controller for all states. The insensitivity to measurement noise of the MFSMC law likely contributed significantly to the results below.

Error State – Pitch	**RMS – LQR**	**RMS – MFSMC**		**Error State – Yaw**	**RMS – LQR**	**RMS – MFSMC**
$\tilde{\theta}$	2.3424	1.2467		$\tilde{\psi}$	1.0823	0.5672
$\dot{\tilde{\theta}}$	1.9900	0.3862		$\dot{\tilde{\psi}}$	0.7391	0.1867
$\ddot{\tilde{\theta}}$	6.6150	*0.*3804		$\ddot{\tilde{\psi}}$	2.8460	0.1955

Table 6: Comparison of Root Mean Square Error values for the linear controller and Model-Free controller

5.1.5 IMU Validation

Prior to testing, the measurements from the inertial measurement unit were validated to determine their accuracy. The Adafruit BNO055 uses an imbedded sensor fusion algorithm to determine its orientation about each axis. The raw data from the gyroscope can also be read from the sensor and the angular velocity can be compared to the calculated angular velocity to determine the accuracy of the measurement.

While the IMU was rotated about each axis, the raw data from the gyroscope and estimated orientation values were taken from the sensor. The orientation values were used to calculate their approximate

derivatives using the MATLAB function, *diff()*. A scaling factor of 0.45 was determined to be used to correct the difference between the measured and calculated derivative.

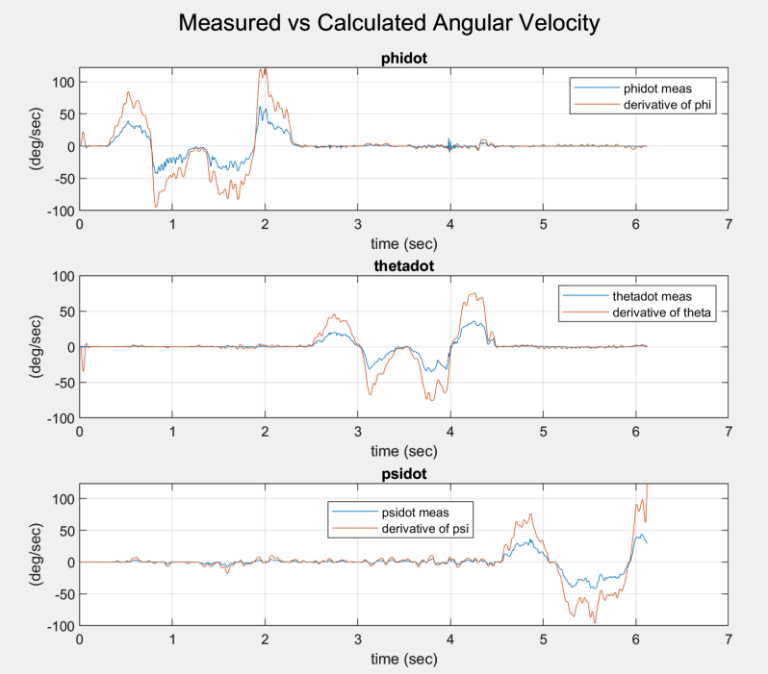


Figure 20: Measured angular velocity from gyroscope and calculated angular velocity

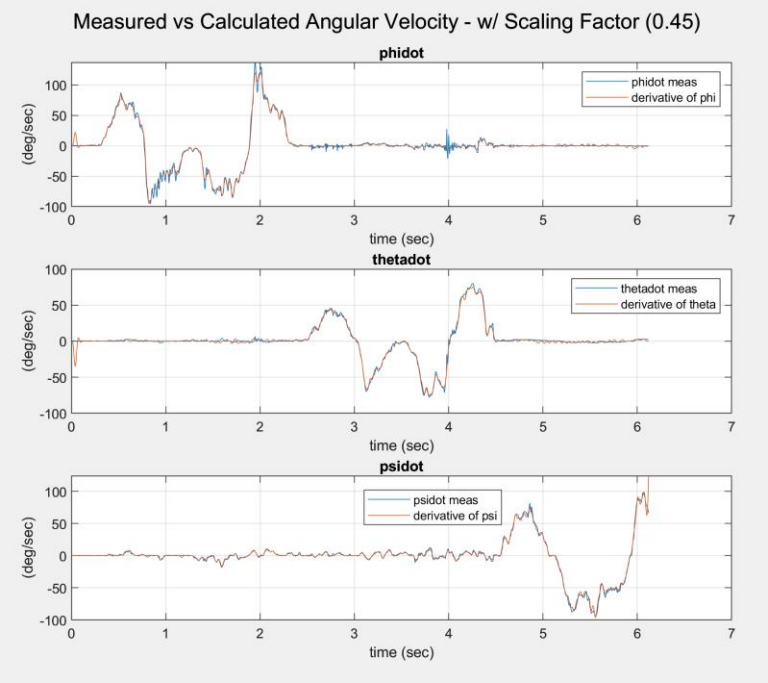


Figure 21: Measured angular velocity with scaling factor from gyroscope and calculated angular velocity

As stated above, the output of the IMU is a reading of the displacement and velocity states of the system. However, for MFSMC, the second derivative, acceleration, is also required. There are many different ways to take derivatives of signals including the use of a dynamic observer and sliding-mode differentiation. Sliding mode differentiation has excellent noise attenuation properties, however is computationally expensive and complicated to implement. A dynamic observer also relies on an accurate model of the system dynamics to approximate the states of the system. For simplicity, a backward difference approximation was used for testing. This method relies on taking the difference between the current measurement and the measurement at the previous time step and dividing by the change in time. Using backwards difference creates a noisy signal so a first order low-pass filter was also used to attenuate the noise.

5.1.6 Baseline Testing Results – LQR

For testing on the real system, the control gains determined from the system model were not modified in the implementation. All state measurement filtering was also kept the same for both the implementation of LQR and MFSMC algorithms. For the "baseline" test, the counterweight's mass remained unchanged from the system model. By reducing the mass of the counterweight, a parametric change in the system can be evaluated for the control laws. For the experimental tests, the mass of the counterweight was reduced by ~20% for each test to evaluate the ability of the respective control laws to handle uncertainty in the system model. The mass was changed by removing bolts from the counterweight.

The goal of the following tests is to evaluate the ability of the LQR and MFSMC laws to maintain the same performance in the baseline and modified systems. As the system is changed from the original system of which the control law was derived, it is expected that the LQR controller's performance will degrade, while the MFSMC will maintain a similar level of performance.

A pre-filtered step input of 4 degrees at 1 second and 8 degrees at 7.5 seconds for pitch. For yaw, the goal was to counteract the propeller's generated moment and stabilize the system at 0 degrees for all time.

Figure 22 shows the state dynamics for the system. While the states do converge to their desired reference signals, the rise time of the system is much lower compared to the simulation results. Additional tuning of the feedback gains would decrease the rise time.

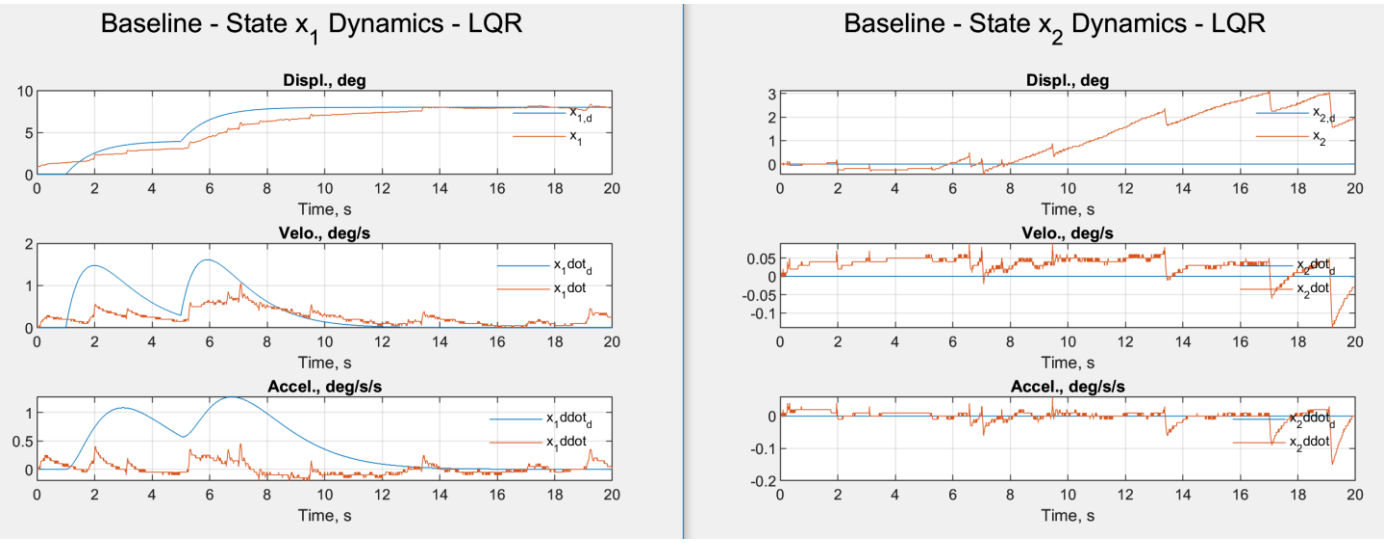


Figure 22: Baseline LQR Test - Pitch (x_1) and Yaw (x_2) state dynamics

The tracking errors for the pitch and yaw states are displayed in Figure 23 for the baseline system. For the pitch states, the LQR controller was inadequate to track large changes in the reference input but over the course of the test, was able to successfully minimize the error. Inaccuracies in the system model that relates propeller velocity to the generated moment may have led to the system's tendency to yaw away from the setpoint of 0 degrees.

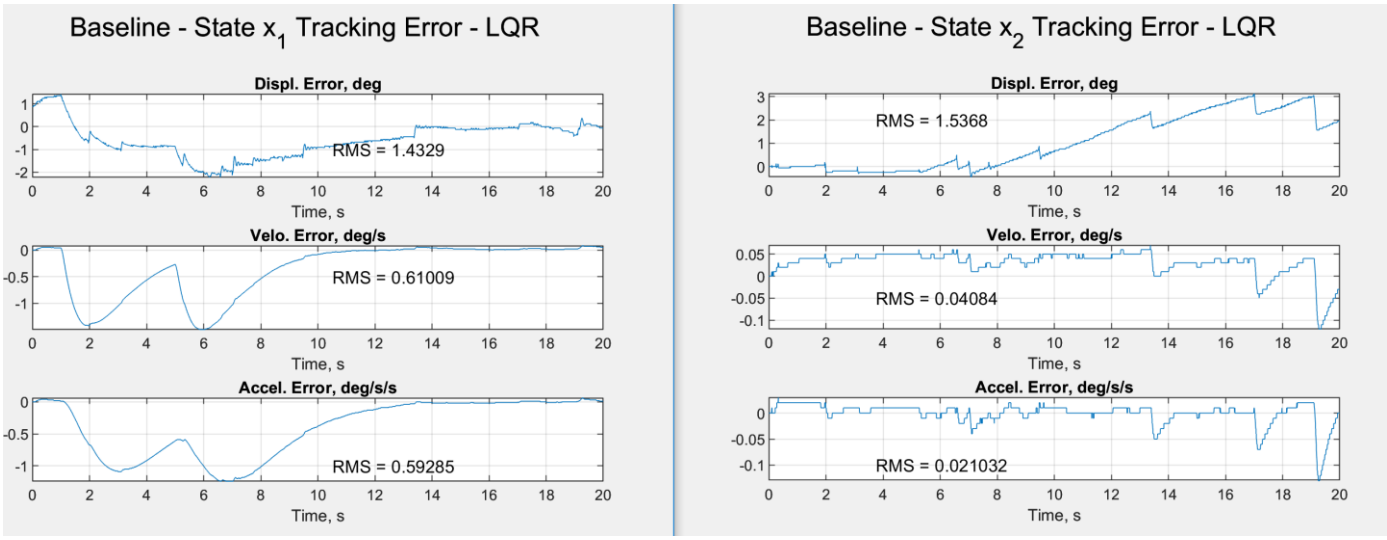


Figure 23: Baseline LQR Test - Pitch (x_1) and Yaw (x_2) state tracking errors

The control effort of the LQR controller is shown below. After simulating the control law, a concern was that state measurement noise would greatly reduce the effectiveness of the controller due to the state feedback nature of the control law. However, this was not observed in testing.

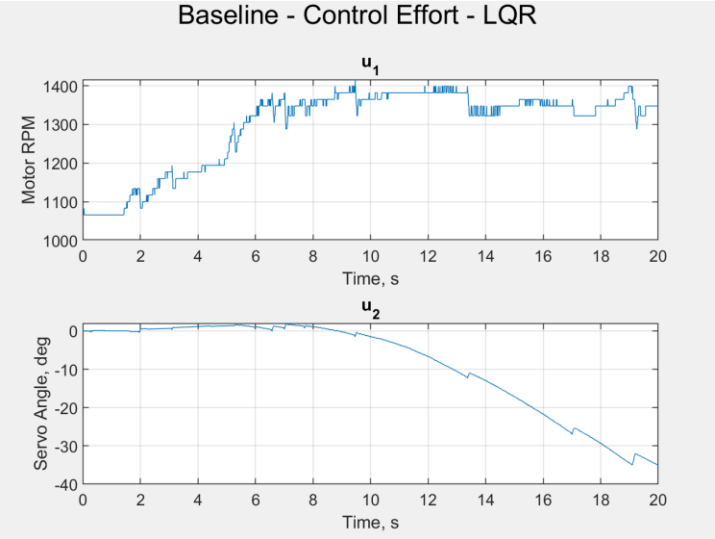


Figure 24: Baseline LQR Test – Control Effort

5.1.7 Baseline Testing Results - MFSMC

For testing, the controller and estimator parameters remain the same as in the simulation. The inputs to the control algorithm are the measured displacement, velocity, and acceleration states for pitch and yaw. The sliding surfaces of the system are defined below.

$$s_{Pitch} = \tilde{x}_1\lambda_1 + \dot{\tilde{x}}_1 \tag{5.7}$$

$$s_{Yaw} = \tilde{x}_2\lambda_2 + \dot{\tilde{x}}_2 \tag{5.8}$$

where $\tilde{x}_n$ is the tracking error for each respective state. The tracking error is defined as the difference between the measured state and the desired state.

$$\tilde{x}_n = (x_n - x_{n,d}) \tag{5.9}$$

The state dynamics of the system are displayed in Figure 25. A slight overshoot occurs in both displacement states. During multiple tests, varying the mass of the counterweight, this tendency was observed. It should be noted, depending on the type of system this algorithm is implemented on, the overshoot may not be acceptable. Notwithstanding, the states converge to their respective reference inputs.

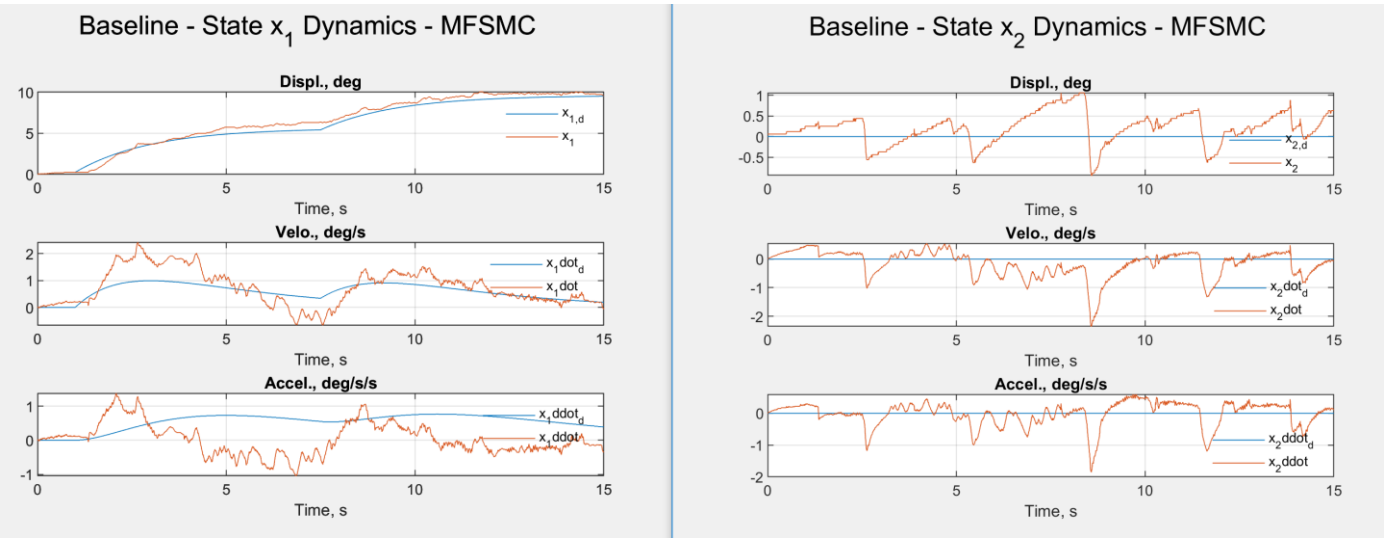


Figure 25: MFSMC Test - Pitch (x_1) and Yaw (x_2) state dynamics

In the figure below, the state tracking errors for pitch and yaw are shown with the MFSMC active for the baseline system. While the magnitude of the displacement tracking error was larger than in simulation, excellent convergence to zero is observed.

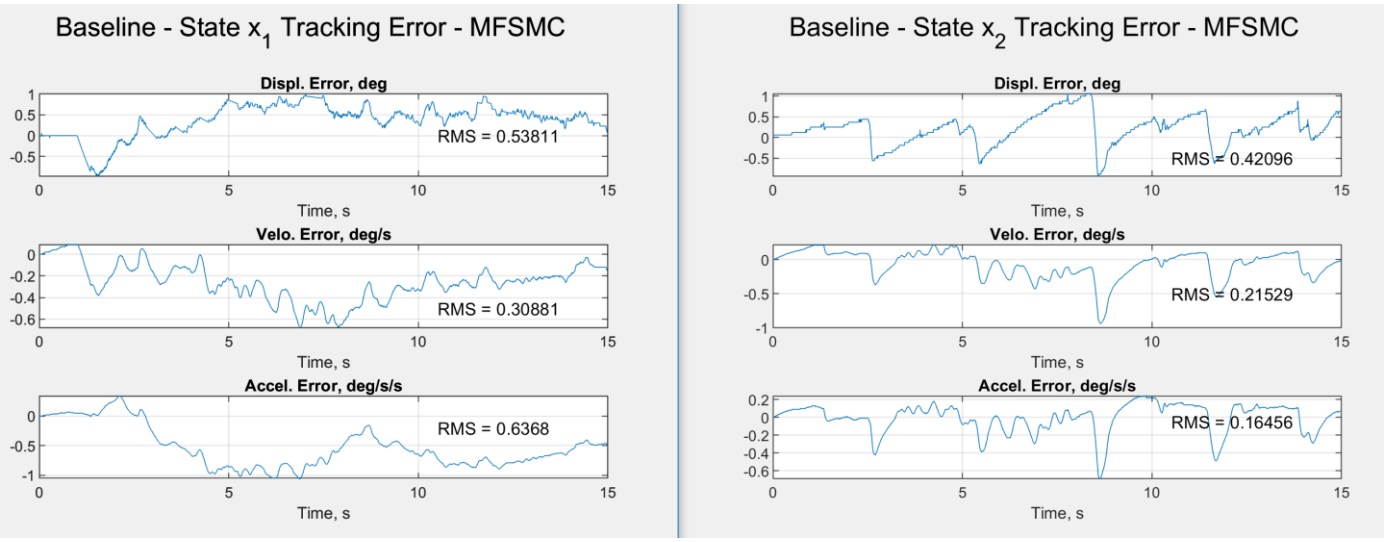


Figure 26: Baseline MFSMC Test - Pitch (x_1) and Yaw (x_2) state tracking errors

The boundary layer dynamics and control input gain estimation are shown below. The sliding surfaces remain within the boundary layer for nearly all time. The estimator was active when the sliding surfaces exceeded the boundary limits as shown in Figure 27. It should be noted, the test outperformed the expected simulation results. The control input gain also converges to nearly the same value as expected in simulation.

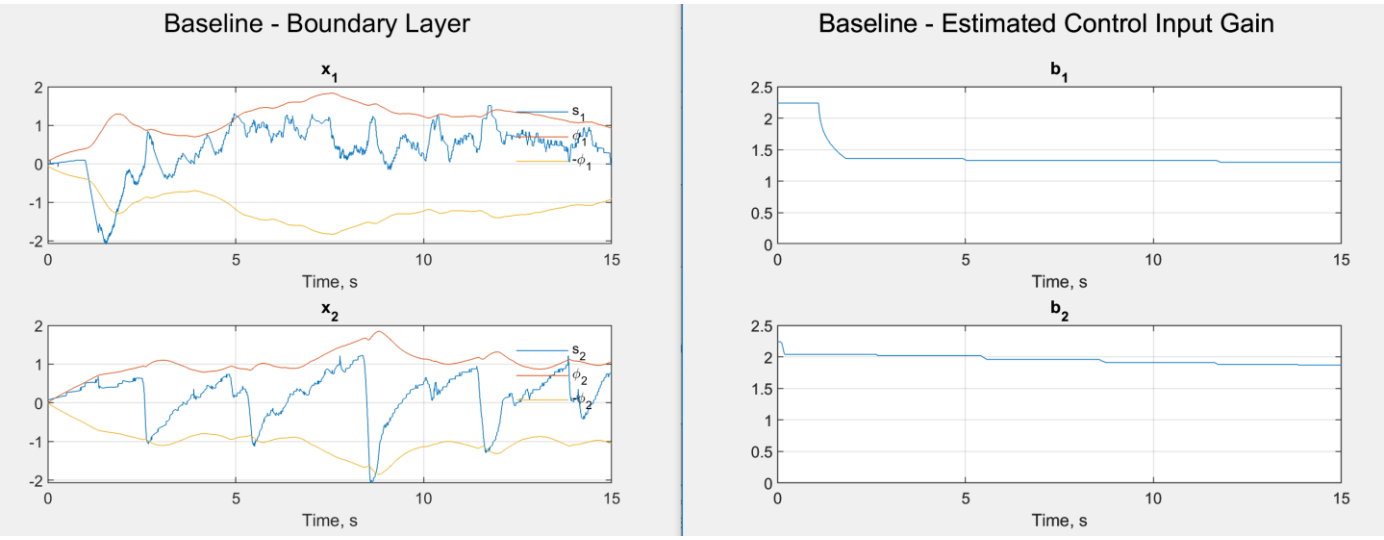


Figure 27: Baseline MFSMC Test – Boundary Layer Dynamics & Estimated Control Input Gain

Figure 28 shows the calculated control input as well as the actuator input. A low-pass filter is applied to the control determined from the MFSMC algorithm and is then mapped to a predetermined range of values for each respective actuator. No control chattering is observed.

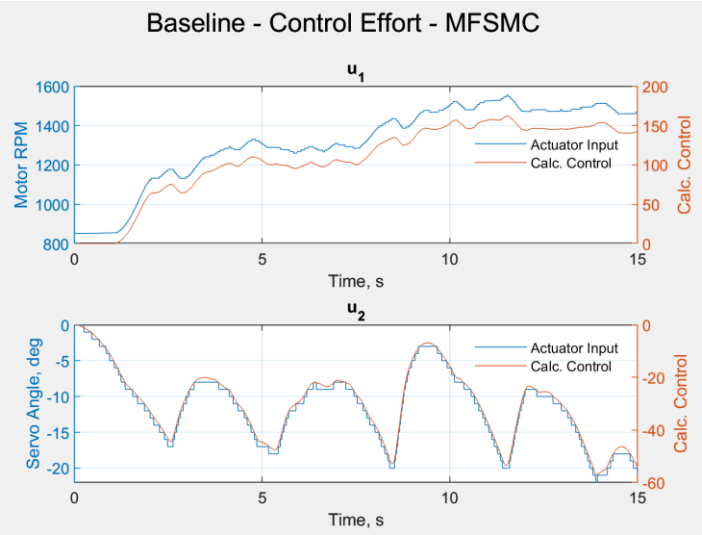


Figure 28: Baseline MFSMC Test – Control Effort

The root-mean-square values for each simulated system and test are shown in Table 7. For all states, the MFSMC algorithm performed better than the linear controller. The pitch RMS value for the linear controller was nearly three times that of the MFSMC algorithm. It is observed that the LQR controller performed significantly better on the real system than it did in simulation. Further adjustments of the LQR gains could improve the performance for the yaw states. The MFSMC algorithm had better tracking compared to simulation and previous tests for the yaw states, but both systems performed adequately.

Error State	Simulation		Test		Error State	Simulation		Test	
Pitch	LQR	MFSMC	LQR	MFSMC	Yaw	LQR	MFSMC	LQR	MFSMC
$\tilde{\theta}$	2.8529	1.2467	1.4329	0.5381	$\tilde{\psi}$	1.3254	0.5672	1.5368	0.4209
$\dot{\tilde{\theta}}$	2.4259	0.3862	0.6100	0.3088	$\dot{\tilde{\psi}}$	0.9043	0.1867	0.0408	0.2152
$\ddot{\tilde{\theta}}$	7.9169	0.3804	0.5928	0.6368	$\ddot{\tilde{\psi}}$	3.1359	0.1955	0.0210	0.1645

Table 7: Comparison of Baseline RMS values for simulation and test data for TV-Balance

5.1.8 Modified System I Test Results – LQR

To evaluate the robustness of the respective control laws, the system parameters were changed. In the following tests, the mass of the counterweight was reduced by ~20%. Effectively increasing the total moment the system must raise. The desired setpoints remain the same as the previous tests.

The state dynamics for the system are shown in Figure 29. The response is even slower than observed in the previous simulation due to the increased load on the motor. The yaw tracking has not changed significantly between the two tests.

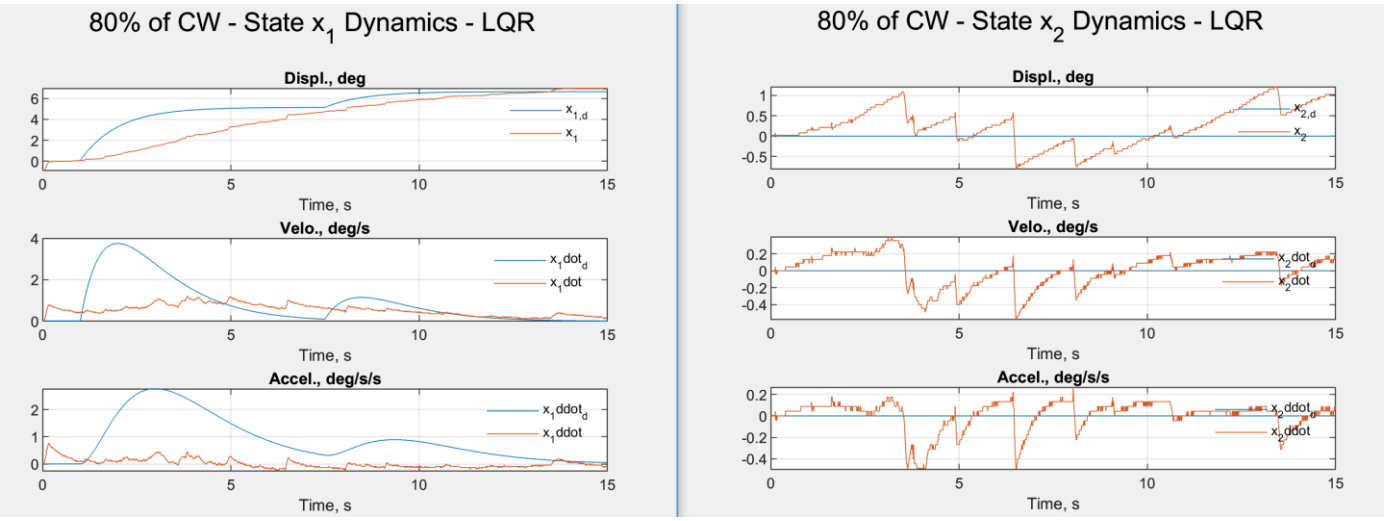


Figure 29: 80% of Counterweight, LQR – Pitch (x_1) and Yaw (x_2) state dynamics

Figure 30 displays the tracking error for the LQR algorithm. As expected, the controller is slow to respond for the pitch states due to the inadequate control gains. However, the system remains stable and the errors do converge to zero.

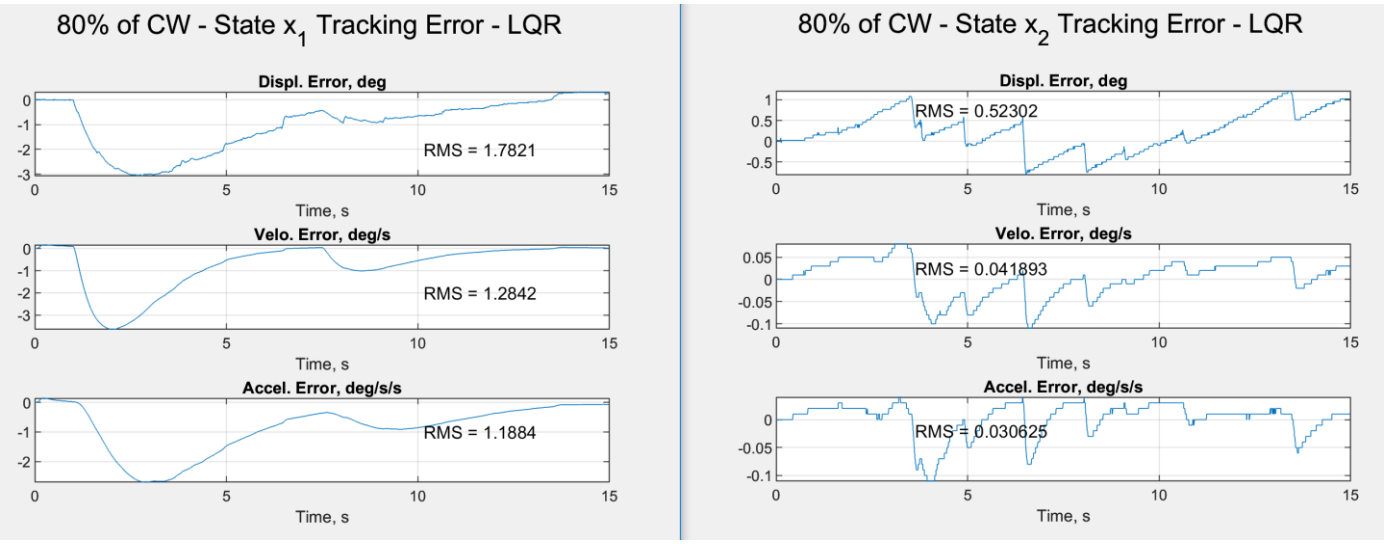


Figure 30: 80% of Counterweight, LQR – Pitch (x_1) and Yaw (x_2) tracking error

Figure 31 depicts the control effort of the DC motor and the servo. Again, the chattering that was observed in simulation was not present in testing.

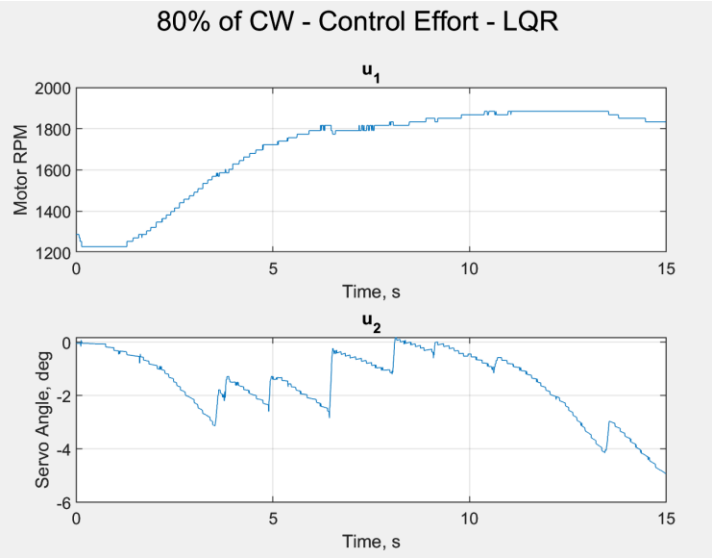


Figure 31: 80% of Counterweight, LQR – Control Effort

5.1.9 Modified System I Test Results - MFSMC

The state dynamics of the modified system are shown in Figure 32. As with previous tests, an overshoot is observed in the pitch state. The measured states still converge to the desired setpoints.

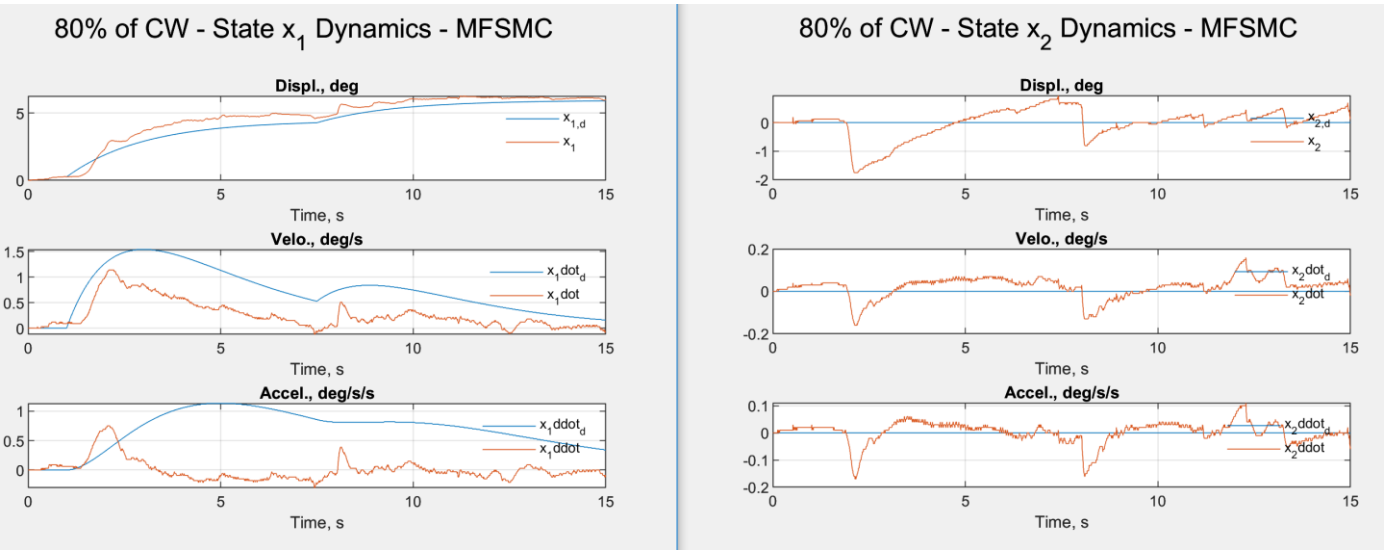


Figure 32: 80% of Counterweight, MFSMC – Pitch (x_1) and Yaw (x_2) state dynamics

The state tracking errors for the MFSMC algorithm are shown in Figure 33. The error was successfully driven to zero. The baseline and modified systems performed similarly in both cases.

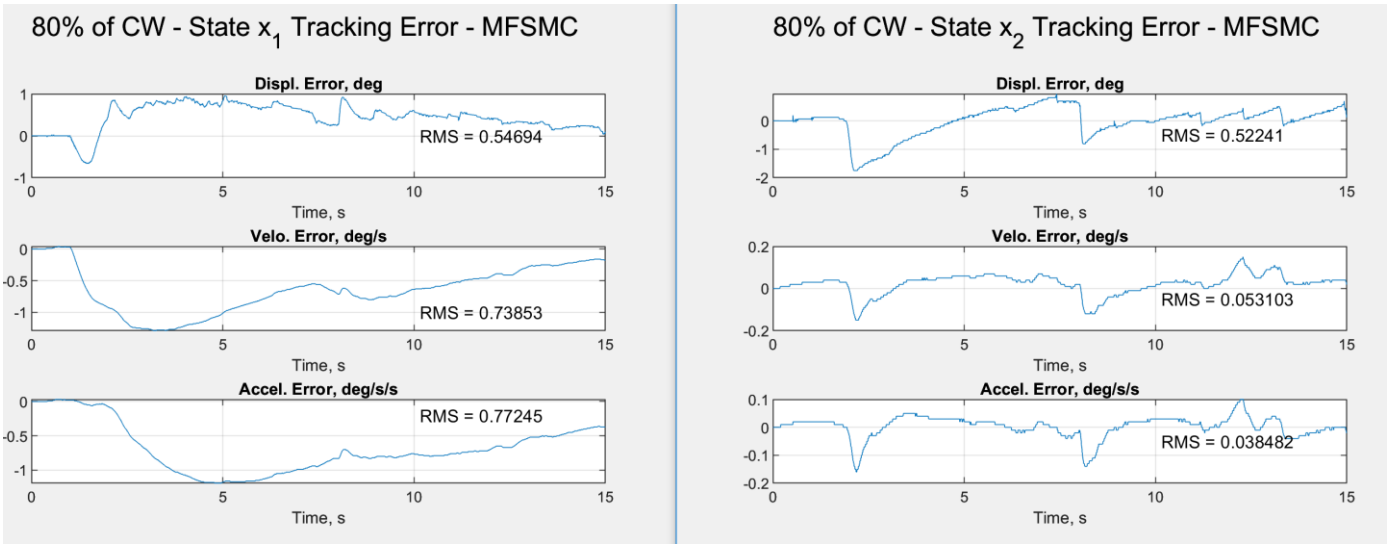


Figure 33: 80% of Counterweight, MFSMC – Pitch (x_1) and Yaw (x_2) tracking error

The boundary layer dynamics and estimated control input gain are depicted below. The sliding surface remains within the boundary layer for nearly all time for both states. Like the previous test, the large divergence of the sliding surface that was seen in simulation was not present in the tests on real hardware. The control input gain for the DC Motor (b_1), was estimated to be slightly smaller than in the previous simulation due to the change in system parameters. Conversely, the control input gain for the servo (b_2) was estimated to be higher than the previous test and simulation.

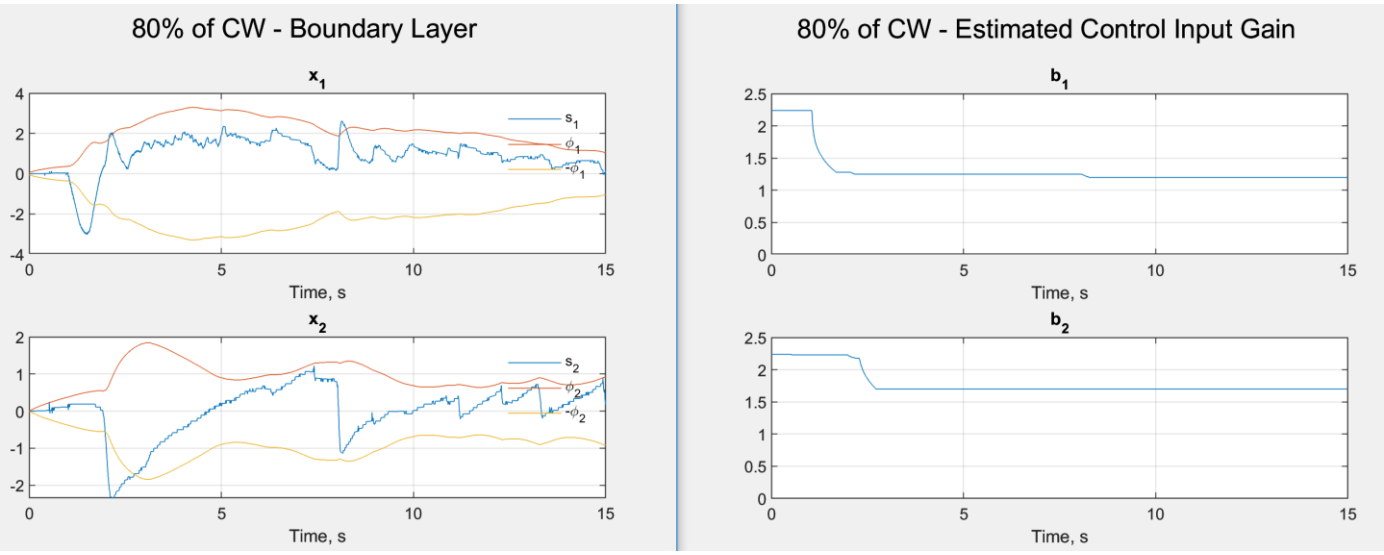


Figure 34: 80% of Counterweight, MFSMC – Boundary layer dynamics and Control input gain estimation

Figure 35 shows the control effort of the servo and DC motor for the modified system. The calculated and actuator control inputs are shown for each respective actuator. The effect of the control mapping can be seen.

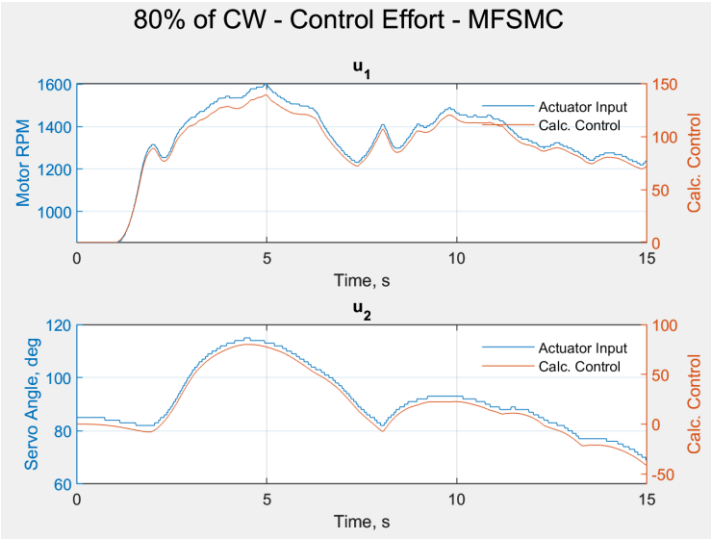


Figure 35: 80% of Counterweight, MFSMC – Control Effort

5.1.10 Modified System II Testing Results – LQR

For the next set of tests, the mass of the counterbalance was further reduced by a total of 40%. The control parameters and reference inputs remain the same as previous tests. Figure 34 shows the state dynamics for the system. Like previous tests, the pitch response has a large rise time compared to previous tests. Small oscillations are also observed. For yaw, the system tracks the reference inputs better than previous tests.

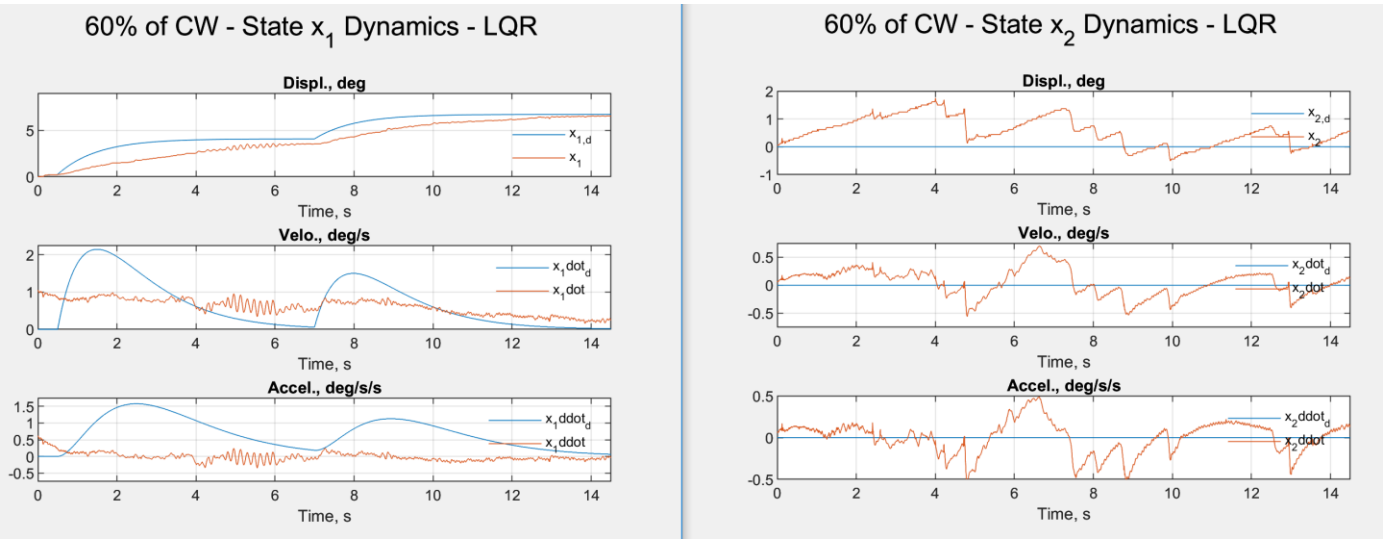


Figure 36: 60% of Counterweight, LQR – Pitch (x_1) and Yaw (x_2) state dynamics

Figure 37 displays the tracking error for the pitch and yaw states. The error is driven to zero for all states, but inadequate control gains hinder the system performance.

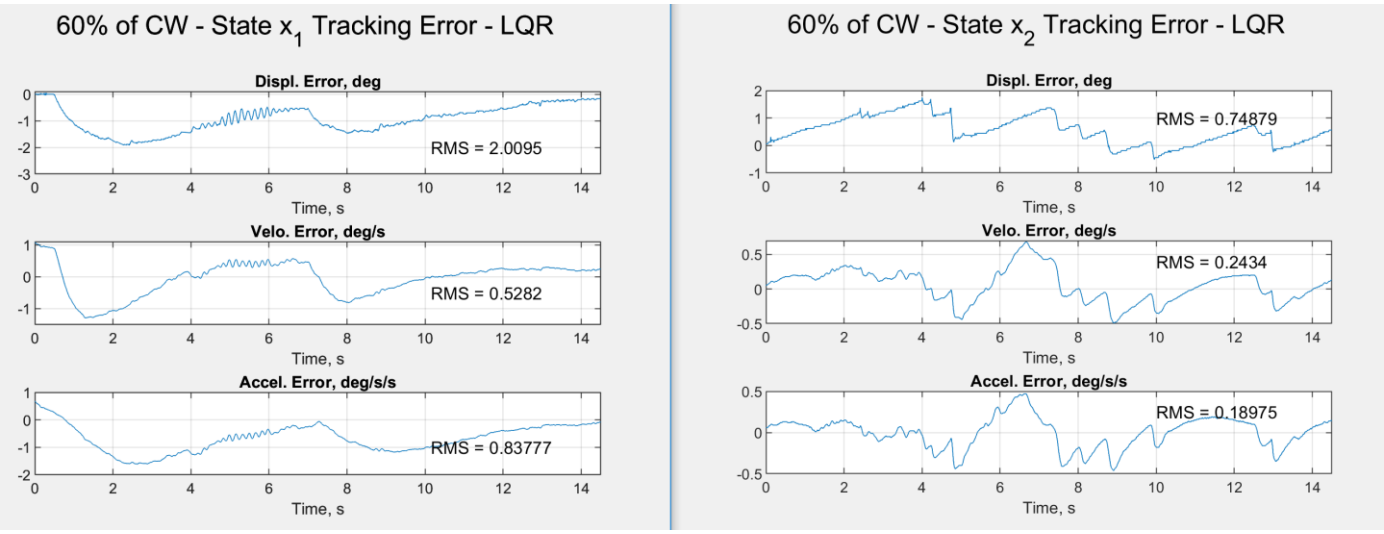


Figure 37: 60% of Counterweight, LQR – Pitch (x_1) and Yaw (x_2) tracking error

The control effort of the two actuators is shown in Figure 38. Chatter can be seen in the control signal to the DC Motor (u_1). The chatter caused the oscillations seen in the pitch states.

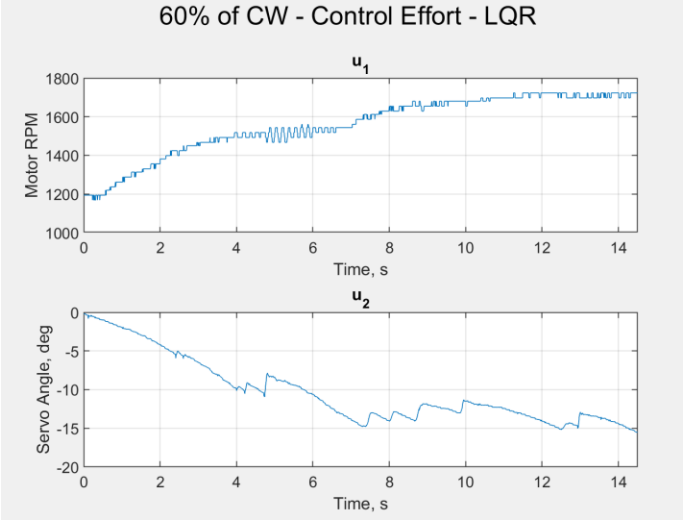


Figure 38: 60% of Counterweight, LQR – Control Effort

5.1.11 Modified System II Testing Results – MFSMC

In the following tests, the mass of the counterweight was reduced by an additional 20%. The parameters for the MFSMC algorithm remain unchanged from previous tests. The state dynamics are shown in the figure below. The system has excellent tracking for all states.

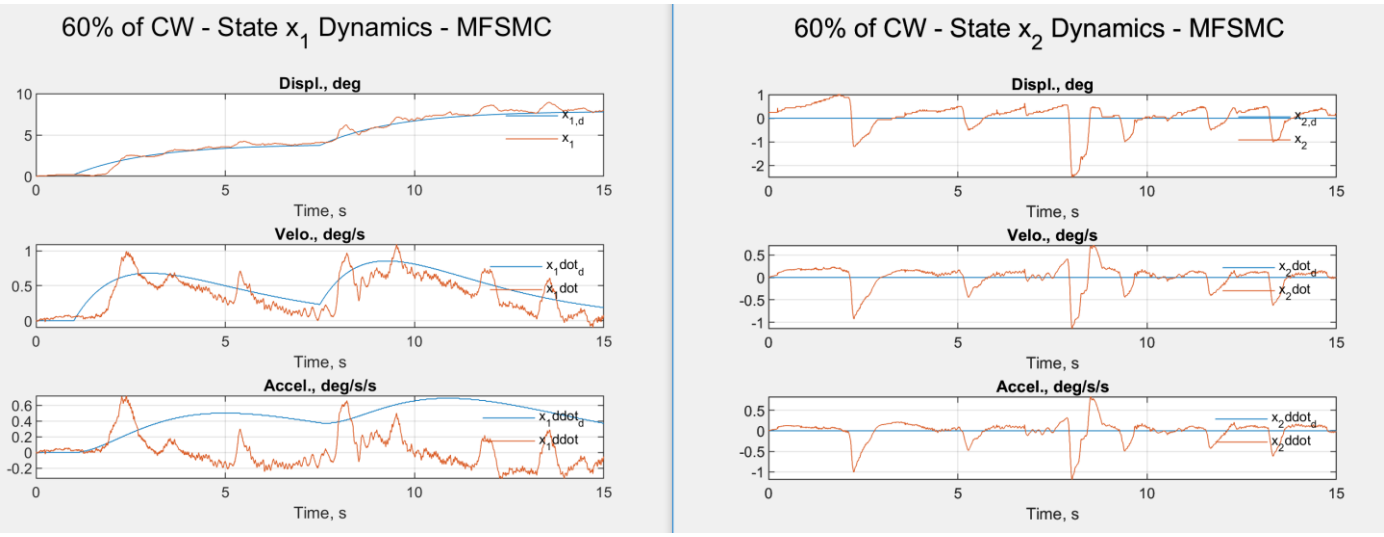


Figure 39: 60% of Counterweight, MFSMC – Pitch (x_1) and Yaw (x_2) state dynamics

The state tracking errors are displayed in Figure 40. As observed in previous tests, the state errors all converge to near-zero with time. However, perfect tracking is never achieved as both pitch and yaw have a tendency to oscillate with a magnitude of around 2 degrees.

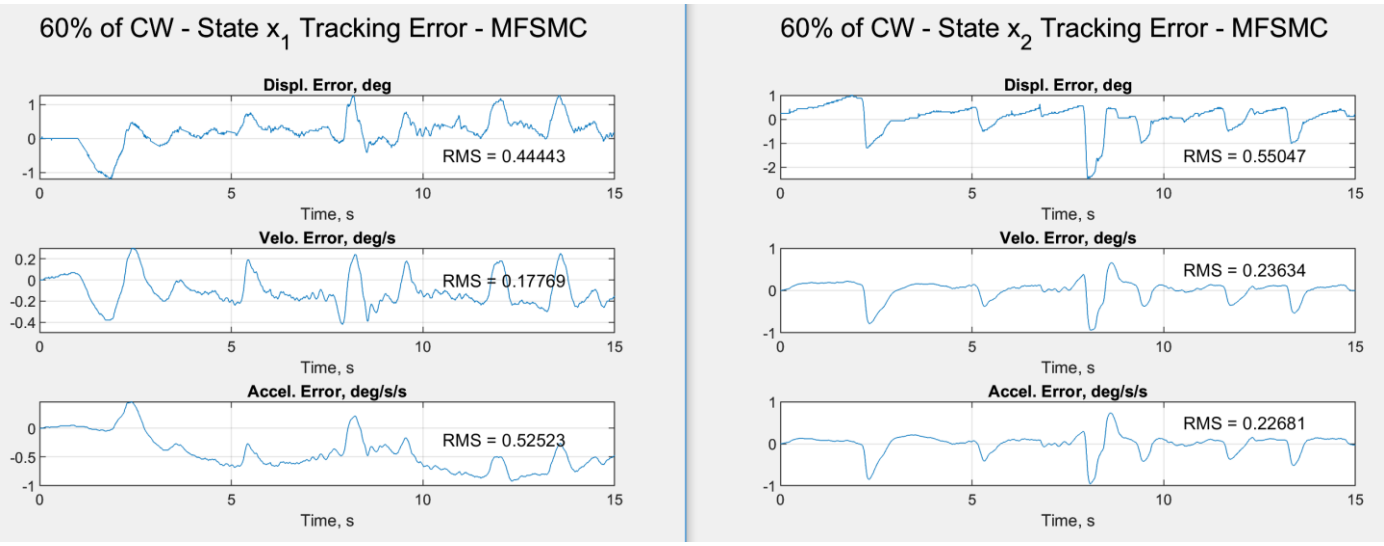


Figure 40: 60% of Counterweight, MFSMC – Pitch (x_1) and Yaw (x_2) tracking error

The boundary layer dynamics and control input gain estimation are shown below. It should be noted that the estimation is updated when the sliding surface is outside the boundary layer in order to drive the system to its' desired state.

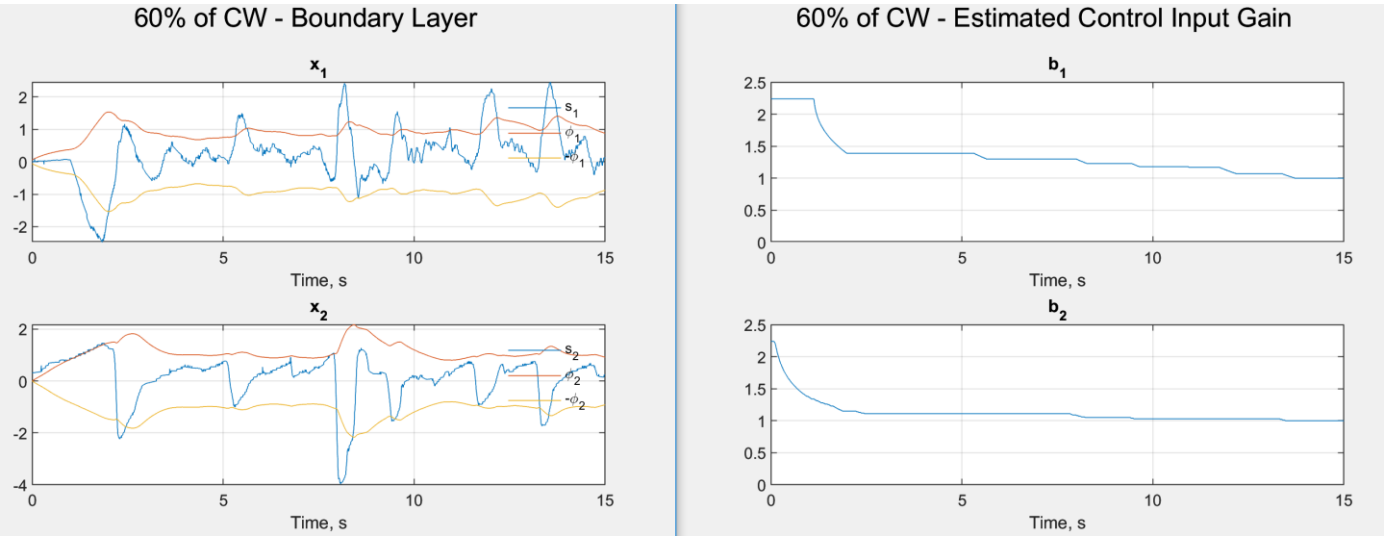


Figure 41: 60% of Counterweight, MFSMC – Boundary layer dynamics and Control input gain estimation

The control effort for the system is shown in Figure 42. The effect of the control mapping can be seen for both the DC motor and servo motor inputs.

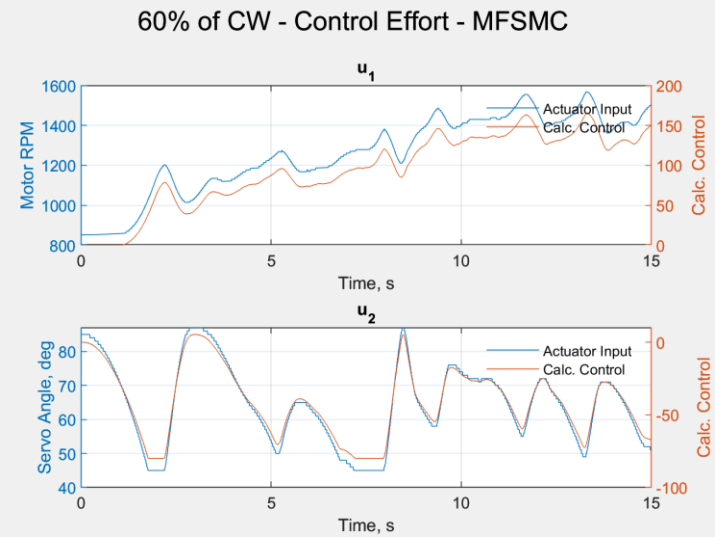


Figure 42: 60% of Counterweight, MFSMC – Control Effort

5.1.12 Discussion of Results

Table 8 shows the summation of the results from the simulation and testing of the MFSMC and LQR control laws. The simulation, performed in Simulink, approximated the system dynamics using a nonlinear model. State measurement noise was approximated by using a random noise generator within the model. The noise parameters were chosen to closely resemble the values seen during initial testing of the IMU. In simulation the MFSMC algorithm outperformed the LQR control law for all states.

For the baseline tests, the real system parameters were not modified in from the derived system model. Both control laws significantly outperformed the simulated tests.

In the next tests, the system parameters were modified. The mass of the counterweight was reduced by roughly 20% and 40% to evaluate the control laws' robustness to system uncertainty. By reducing the mass of the counterweight, the load that the system was required to lift was increased. For the linear control law, the performance degraded significantly between the baseline and modified system for the pitch states. Conversely, the MFSMC algorithm performed better than in the baseline and simulation tests. The RMS value for the LQR control law increased significantly between tests while the MFSMC algorithm's RMS value remained nearly the same between all tests. While the LQR control law performed well under the effects of system uncertainty, the MFSMC algorithm was shown to have increased robustness in simulation and in testing on real hardware.

Error State	Simulation		Baseline (Same system as in simulation)		Modified System (CW reduced by 20%)		Modified System (CW reduced by 40%)	
Pitch	**LQR**	**MFSMC**	**LQR**	**MFSMC**	**LQR**	**MFSMC**	**LQR**	**MFSMC**
$\tilde{\theta}$	2.852	1.246	1.432	0.531	1.782	0.546	2.009	0.444
$\dot{\tilde{\theta}}$	2.425	0.386	0.610	0.308	1.284	0.738	0.528	0.177
$\ddot{\tilde{\theta}}$	7.916	*0.*380	0.592	0.636	1.188	0.772	0.837	0.525
Yaw	**LQR**	**MFSMC**	**LQR**	**MFSMC**	**LQR**	**MFSMC**	**LQR**	**MFSMC**
$\tilde{\psi}$	1.325	0.567	1.536	0.420	0.523	0.522	0.748	0.550
$\dot{\tilde{\psi}}$	0.904	0.186	0.040	0.215	0.041	0.053	0.243	0.236
$\ddot{\tilde{\psi}}$	3.135	0.195	0.021	0.164	0.030	0.038	0.189	0.226

Table 8: Simulation and testing results for the MFSMC algorithm and LQR control law

6.0 Conclusion

In this work, a model-free control law was implemented on various systems and tested in simulation and application. Instead of being based on a derived system model, the MFSMC algorithm was based on the system order, previous control inputs, and state measurements. The control law exhibited near perfect tracking of all states in simulation which reinforced the results found in previous works. This thesis, however, emphasized the application of MFSMC to a hardware system.

A tilt-rotor balancing system consisting of a DC motor with a propeller and a servo motor for thrust vectoring was chosen because it is analogous to a FWUAS. A system model was derived to sufficiently approximate the system dynamics. Sensor noise was also included in this simulation to replicate the output from the IMU. A full state feedback controller was derived using this system model which would be tested in simulation and on hardware. In simulation, the MFSMC algorithm outperformed the linear control law by a significant margin. Due to the state feedback nature of the linear controller, sensor noise generated a noisy control signal leading to a degradation in performance.

When implementing the MFSMC law on the system, it was determined that additional changes needed to be made. The controller was highly aggressive, which led to stability issues when large changes of the desired reference signals were fed to the system. A first-order low-pass filter was added to the calculated control input then mapped to a range of actuator values that were predetermined to be sufficient for the application. The changes significantly improved the performance of the system. The root-mean-square of the error was used to evaluate the performance of each control law under different conditions.

For the first set of tests, the system remained unchanged from that used in simulation. The linear controller outperformed the simulated controller significantly while the MFSMC algorithm performed almost identically. Control chatter, a common problem with the implementation of SMC-based techniques was not observed in simulation or testing.

The next set of tests involved changing the system parameters to evaluate the robustness of each control law. The mass of the counterweight was reduced by roughly 20% and 40%. Effectively increasing the load that the system is required to lift. The RMS of the error increased significantly for the linear controller while the performance of the MFSMC algorithm remained nearly the same. While both algorithms performed well under the system uncertainty, in simulation and in testing, the MFSMC law outperformed the linear control in terms of robustness to uncertainty for all states. The experimentation done in this paper demonstrated the ability of MFSMC to control systems where the exact parameters of the system model are unknown.

6.1 Future Work

The goal of this work was to evaluate the performance of a MFSMC algorithm on a FWUAS. However, this was not accomplished. The algorithm outlined in this paper should be applied to the FWUAS when the construction of the airframe is complete.

MFSMC has demonstrated its' inherent robustness to system uncertainties on SISO and MIMO systems. However, both systems tested are considered square systems. A square system is one that has the same number of inputs as outputs. Further testing would need to be done to determine what changes must be made to implement MFSMC on a non-square system. Future work may also be performed on scaling the algorithm to larger systems such as a quadcopter. Previous work on implementation for a quadcopter was done unsuccessfully. If the low-pass filter and control input mapping improves the performance of the quadcopter should be a point of emphasis if that route is explored.

7.0 Bibliography

[1] G. Sachs, "Comparison of Power Requirements: Flapping vs. Fixed Wing Vehicles," Aerospace, vol. 3, no. 4, p. 31, Sep. 2016, doi: 10.3390/aerospace3040031.

[2] C. De Wagter, S. Tijmons, B. D. Remes, and G. C. de Croon, ""Autonomous flight of a 20- gram flapping wing mav with a 4-gram onboard stereo vision system," 2014, pp. 4982–4987.

[3] G. C. H. E. De Croon, M. Percin, B. D. W. Remes, R. Ruijsink, and C. De Wagter, The Delfly: Design, Aerodynamics, and Artificial Intelligence of a Flapping Wing Robot. Springer, 2016.

[4] Che, Ning. "Attitude and Position Control of Flapping–Wing Micro Aerial Vehicles." Lakehead University Library, 2018, https://knowledgecommons.lakeheadu.ca/handle/2453/4485.

[5] N. T. Jafferis, E. F. Helbling, M. Karpelson, and R. J. Wood, "Untethered flight of an insect-sized flapping-wing microscale aerial vehicle," Nature, vol. 570, no. 7762, pp. 491–495, 2019.

[6] Q. O'Rourke, "Efficacy of Flapping-wing Flight Via Dual Piezoelectric Actuation," Rochester Institute of Technology, Rochester, NY, 2022. [Online]. Available: https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=12229&context=theses

[7] Jing, Y., Sun, H., Zhang, L. & Zhang, T. (2017). Variable speed control of wind turbines based on the quasi-continuous high-order sliding mode method. Energies, 10 (10), 1626-1-1626-21

[8] J.-J. E. Slotine and W. Li, Applied Nonlinear Control. Upper Saddle River, New Jersey: Prentice Hall, Inc., 1991.

[9] E. Madadi, Y. Dong, and D. Soffker, "Comparison of Different Model-Free Control Methods Concerning Real-Time Benchmark," J. Dyn. Syst. Meas. CONTROL, Aug. 2018, doi: 10.1115/1.4040967.

[10] A. Crassidis and A. Mizov, "A Model-Free Control Algorithm Derived Using the Sliding Mode Control Method," Proc. 2 Int. Conf. Control Dyn. Syst. Robot., vol. 166, May 2015.

[11] Mittmann Reis, Raul, "A New Model-Free Sliding Mode Control Method with Estimation of Control Input Error" (2016). Thesis. Rochester Institute of Technology. Accessed from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=10164&context=theses

[12] Schulken, Eric, "Investigations of Model-Free Sliding Mode Control Algorithms including Application to Autonomous Quadrotor Flight" (2017). Thesis. Rochester Institute of Technology. Accessed from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=10786&context=theses

[13] Kadungoth Sreeraj, Adarsh Raj, "A Model-Free Control Algorithm Based on the Sliding Mode Control Method with Applications to Unmanned Aircraft Systems" (2019). Thesis. Rochester Institute of Technology. Accessed from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11198&context=theses

[14] Monti, Christian, "Model-Free Control of an Unmanned Aircraft Quadcopter Type System" (2020). Thesis. Rochester Institute of Technology. Accessed from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11636&context=theses

[15] M. S. Islam, "A Model-Free Control System Based on the Sliding Mode Control with Automatic Tuning Using as On-Line Parameter Estimation Approach," Rochester Institute of Technology, Rochester, NY, 2020.

[16] H. J. Marquez, Nonlinear Control Systems Analysis and Design, Hoboken, New Jersey: Jhon Wiley & Sons, Inc, 2003

8.0 Appendix

MATLAB CODE

```
%% Model Free Sliding Mode Control Master File
% Walker Hare - 3/30/2023
% Master's Thesis

clear all, clc

%% Second Order MIMO System - Unitary Input Gain
 % "MFSMC_MIMO_Sariful.slx"
clear all, clc
tf = 20;
m1 = 10; %mass
m2 = 20;
c1= 5; %damping coefficient
c2= 8;
k1= 3; %spring constant
k2= 7;
d1= -1.5; %spring stiffness
d2= -3;
x1_0 = [pi/2 0];
x2_0 = [pi/2 0];

b1_real = 3;
b2_real = 2;
b1_up = 8.5;
b1_low = 2.5;
b2_up = 2.5;
b2_low = 1.5;
bhat1 = 1; %sqrt(b1_up*b1_low);
bhat2 = 1; %sqrt(b2_up*b2_low);
bhat = [bhat1 bhat2];
beta1 = 1; %sqrt(b1_up/b1_low);
beta2 = 1; %sqrt(b2_up/b2_low);
beta = [beta1 beta2];
su1 = 0.2;
su2 = 0.2;
su = [su1 su2];
lambda1 = 4;
lambda2 = 3;
lambda = [lambda1 lambda2];
eta1 = 0.0006;
eta2 = 0.00022;
eta = [eta1 eta2];
phi1_0 = eta1/lambda1;
phi2_0 = eta2/lambda2;
phi0 = [phi1_0 phi2_0];
lambda1_0 = 50;
lambda2_0 = 50;
lambda0 = [lambda1_0 lambda2_0];
k1_0 = 500;
k2_0 = 500;
```

```
k0 = [k1_0 k2_0];
eta1hat0 = eta1;
eta2hat0 = eta2;
etahat0 = [eta1hat0 eta2hat0];
P_eta1_0 = 10;
P_eta2_0 = 10;
P_eta0 = [P_eta1_0 P_eta2_0];

out = sim('MFSMC_MIMO_Sariful.slx');
%Simulink.sdi.view

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
eta1_hat = out.eta1_hat;
eta2_hat = out.eta2_hat;
s1 = out.s1;
s2 = out.s2;
x1_d = out.x1_d;
x1dot_d = out.x1dot_d;
x1ddot_d = out.x1ddot_d;
x2_d = out.x2_d;
x2dot_d = out.x2dot_d;
x2ddot_d = out.x2ddot_d;
x2 = out.x2;
x2dot = out.x2dot;
x2ddot = out.x2ddot;
x1 = out.x1;
x1dot = out.x1dot;
x1ddot = out.x1ddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda;

%RMS
x1RMS = rms(x1_e);
x1dotRMS = rms(x1dot_e)
x1ddotRMS = rms(x1ddot_e)
x2RMS = rms(x2_e)
x2dotRMS = rms(x2dot_e)
x2ddotRMS = rms(x2ddot_e)

%% Data Logging and Plotting


figure(1), clf
```

```
sgtitle('State x_1 Tracking Error')
subplot(311)
plot(t,x1_e),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x1RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x1dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-3,['RMS = ' num2str(x1dotRMS)])
title('Velo. Error, m/s')
subplot(313)
plot(t,x1ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.25,['RMS = ' num2str(x1ddotRMS)])
title('Accel. Error, m/s/s')

figure(2), clf
sgtitle('State x_2 Tracking Error')
subplot(311)
plot(t,x2_e),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x2dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2dotRMS)])
title('Velo. Error, m/s')
subplot(313)
plot(t,x2ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2ddotRMS)])
title('Accel. Error, m/s/s')

figure(3), clf
sgtitle('State x_1 Dynamics')
subplot(311)
plot(t,x1_d,t,x1), legend('x_1_,_d','x_1'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., m')
subplot(312)
plot(t,x1dot_d,t,x1dot), legend('x_1dot_d','x_1dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., m/s')
subplot(313)
plot(t,x1ddot_d,t,x1ddot), legend('x_1ddot_d','x_1ddot'), legend boxoff, grid on,
xlabel('Time, s')
title('Accel., m/s/s')

figure(4), clf
sgtitle('State x_2 Dynamics')
subplot(311)
plot(t,x2_d,t,x2), legend('x_2_,_d','x_2'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., m')
subplot(312)
plot(t,x2dot_d,t,x2dot), legend('x_2dot_d','x_2dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., m/s')
subplot(313)
plot(t,x2ddot_d,t,x2ddot), legend('x_2ddot_d','x_2ddot'), legend boxoff, grid on,
xlabel('Time, s')
```

```
title('Accel., m/s/s')

figure(5), clf
sgtitle('Control Effort')
subplot(211)
plot(t,u1), grid on, xlabel('Time, s')
title('u_1')
subplot(212)
plot(t,u2), grid on, xlabel('Time, s')
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
plot(t,s1,t,phi1,t,-phi1), legend('s_1','\phi_1','-\phi_1'), legend boxoff, grid on,
xlabel('Time, s')
title('x_1')
subplot(212)
plot(t,s2,t,phi2,t,-phi2), legend('s_2','\phi_2','-\phi_2'), legend boxoff, grid on,
xlabel('Time, s')
title('x_2')

figure(7), clf
sgtitle('Switching Gain, K')
subplot(211)
plot(t,K1), grid on, xlabel('Time, s')
axis([0 tf -.025 .1])
title('K_1')
subplot(212)
plot(t,K2), grid on, xlabel('Time, s')
title('K_2')

figure(8), clf
sgtitle('Estimated \eta')
subplot(211)
plot(t,eta1_hat), grid on, xlabel('Time, s')
title('\eta_1')
subplot(212)
plot(t,eta2_hat), grid on, xlabel('Time, s')
title('\eta_2')

%% Second Order MIMO System - Control Input Gain Estimation
 % "MFSMC_MIMO_Sariful.slx"
clear all, clc

tf = 10;

%%%Incorporate integral term into sliding surface%%%
% ' = 0' for s = lambda*x~ + xdot~ + int(x~)
% ' = 1' for s = lambda*x~ + xdot~
alt_SlidingSurface = 0;

%%%System Params%%%
m1 = 10; %mass
m2 = 20;
```

```
c1= 5; %damping coefficient
c2= 8;
k1= 3; %spring constant
k2= 7;
d1= -1.5; %spring stiffness
d2= -3;
x1_0 = [pi/2 0];
x2_0 = [pi/2 0];

su1 = 0.2;
su2 = 0.2;
su = [su1 su2];
lambda1 = 4;
lambda2 = 3;
lambda = [lambda1 lambda2];
eta1 = 0.0006;
eta2 = 0.00022;
eta = [eta1 eta2];
phi1_0 = eta1/lambda1;
phi2_0 = eta2/lambda2;
phi0 = [phi1_0 phi2_0];

b1_real = 3;
b2_real = 2;
b_real = [b1_real b2_real];
b1hat0 = 0.8;
b2hat0 = 0.8;
bhat0 = [b1hat0 b2hat0];
lambda1_0 = 80;
lambda2_0 = 80;
lambda0 = [lambda1_0 lambda2_0];
k1_0 = 500;
k2_0 = 500;
k0 = [k1_0 k2_0];
P_b1_0 = 10;
P_b2_0 = 10;
P_b0 = [P_b1_0 P_b2_0];

out = sim('MFSMC_MIMO_bhat_Sariful.slx');

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
b1hat = out.b1hat;
b2hat = out.b2hat;
b1real = out.b1real;
b2real = out.b2real;
s1 = out.s1;
s2 = out.s2;
```

```
x1_d = out.x1_d;
x1dot_d = out.x1dot_d;
x1ddot_d = out.x1ddot_d;
x2_d = out.x2_d;
x2dot_d = out.x2dot_d;
x2ddot_d = out.x2ddot_d;
x2 = out.x2;
x2dot = out.x2dot;
x2ddot = out.x2ddot;
x1 = out.x1;
x1dot = out.x1dot;
x1ddot = out.x1ddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda;

%RMS
x1RMS = rms(x1_e);
x1dotRMS = rms(x1dot_e);
x1ddotRMS = rms(x1ddot_e);
x2RMS = rms(x2_e);
x2dotRMS = rms(x2dot_e);
x2ddotRMS = rms(x2ddot_e);
b1RMS = rms(b1hat-b1real);
b2RMS = rms(b2hat-b2real);
ErrorState_x1 = ["x1~"; "x1dot~"; "x1ddot~"];
ErrorState_x2 = ["x2~"; "x2dot~"; "x2ddot~"];
RMSx1 = [x1RMS; x1dotRMS; x1ddotRMS];
RMSx2 = [x2RMS; x2dotRMS; x2ddotRMS];
StateRMStable = table(ErrorState_x1, RMSx1, ErrorState_x2, RMSx2)
EstParam = ['b_1'; 'b_2'];
RMSb = [b1RMS; b2RMS];
EstRMStable = table(EstParam, RMSb)

%% Data Logging and Plotting

figure(1), clf
sgtitle('State x_1 Tracking Error')
subplot(311)
plot(t,x1_e),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x1RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x1dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-3,['RMS = ' num2str(x1dotRMS)])
title('Velo. Error, m/s')
subplot(313)
plot(t,x1ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.25,['RMS = ' num2str(x1ddotRMS)])
title('Accel. Error, m/s/s')

figure(2), clf
```

```
sgtitle('State x_2 Tracking Error')
subplot(311)
plot(t,x2_e),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x2dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2dotRMS)])
title('Velo. Error, m/s')
subplot(313)
plot(t,x2ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x2ddotRMS)])
title('Accel. Error, m/s/s')

figure(3), clf
sgtitle('State x_1 Dynamics')
subplot(311)
plot(t,x1_d,t,x1), legend('x_1_,_d','x_1'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., m')
subplot(312)
plot(t,x1dot_d,t,x1dot), legend('x_1dot_d','x_1dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., m/s')
subplot(313)
plot(t,x1ddot_d,t,x1ddot), legend('x_1ddot_d','x_1ddot'), legend boxoff, grid on,
xlabel('Time, s')
title('Accel., m/s/s')

figure(4), clf
sgtitle('State x_2 Dynamics')
subplot(311)
plot(t,x2_d,t,x2), legend('x_2_,_d','x_2'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., m')
subplot(312)
plot(t,x2dot_d,t,x2dot), legend('x_2dot_d','x_2dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., m/s')
subplot(313)
plot(t,x2ddot_d,t,x2ddot), legend('x_2ddot_d','x_2ddot'), legend boxoff, grid on,
xlabel('Time, s')
title('Accel., m/s/s')

figure(5), clf
sgtitle('Control Effort')
subplot(211)
plot(t,u1), grid on, xlabel('Time, s')
title('u_1')
subplot(212)
plot(t,u2), grid on, xlabel('Time, s')
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
```

```
plot(t,s1,t,phi1,t,-phi1), legend('s_1','\phi_1','-\phi_1'), legend boxoff, grid on,
xlabel('Time, s')
title('x_1')
subplot(212)
plot(t,s2,t,phi2,t,-phi2), legend('s_2','\phi_2','-\phi_2'), legend boxoff, grid on,
xlabel('Time, s')
title('x_2')

figure(7), clf
sgtitle('Switching Gain, K')
subplot(211)
plot(t,K1), grid on, xlabel('Time, s')
axis([0 tf -.025 .1])
title('K_1')
subplot(212)
plot(t,K2), grid on, xlabel('Time, s')
title('K_2')

figure(8), clf
sgtitle('Estimated Control Input Gain, b')
subplot(211)
plot(t,b1hat), grid on, xlabel('Time, s')
title('b_1'), yline(b1real), axis([0 tf 0 1.1*b1real])
text(.7*tf,.75*b1real,['b_1_,_t_r_u_e = ' num2str(b1real)])
text(.7*tf,.5*b1real,['RMS = ' num2str(b1RMS)])
subplot(212)
plot(t,b2hat), grid on, xlabel('Time, s')
title('b_2'), yline(b2real), axis([0 tf 0 1.1*b2real])
text(.7*tf,.75*b2real,['b_2_,_t_r_u_e = ' num2str(b2real)])
text(.7*tf,.5*b2real,['RMS = ' num2str(b2RMS)])

%% Altitude and Roll Control w/o Actuator Dynamics - Control Input Gain Estimation
 % Actuator Dynamics neglected
 % One thrust source on either side of COM
 % "MFSMC_AltitudeRollControl_b_est.slx"
clear all, clc

tf = 10;

%%%Incorporate integral term into sliding surface%%%
% ' = 0' for s = lambda*x~ + xdot~ + int(x~)
% ' = 1' for s = lambda*x~ + xdot~
alt_SlidingSurface = 0;

%%%Aircraft Params%%%
m = 1; %0.02; %Craft Mass, kg
I = 1; %0.035; %Craft Inertia
L = 1; %0.069; %Craft Moment Arm from COM to Center of Thrust, m

%%%IC%%%
g1 = 1;
g2 = 0.05;
z0 = g1*[pi/2 0]; %Height, m
theta0 = g2*[pi/2 0]; %Roll, rad
```

```
%%%MFSMC Params%%%
su1 = 0.2;
su2 = 0.2;
su = [su1 su2];
lambda1 = 4;
lambda2 = 3;
lambda = [lambda1 lambda2];
eta1 = 0.0006;
eta2 = 0.00022;
eta = [eta1 eta2];
phi1_0 = eta1/lambda1;
phi2_0 = eta2/lambda2;
phi0 = [phi1_0 phi2_0];

b1_real = 3;
b2_real = 2;
b_real = [b1_real b2_real];
b1hat0 = 0.8;
b2hat0 = 0.8;
bhat0 = [b1hat0 b2hat0];
lambda1_0 = 80;
lambda2_0 = 80;
lambda0 = [lambda1_0 lambda2_0];
k1_0 = 500;
k2_0 = 500;
k0 = [k1_0 k2_0];
P_b1_0 = 10;
P_b2_0 = 10;
P_b0 = [P_b1_0 P_b2_0];

out = sim('MFSMC_AltitudeRollControl_b_est.slx');
%Simulink.sdi.view

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
b1hat = out.b1hat;
b2hat = out.b2hat;
b1real = out.b1real;
b2real = out.b2real;
s1 = out.s1;
s2 = out.s2;
x1_d = out.z_d;
x1dot_d = out.zdot_d;
x1ddot_d = out.zddot_d;
x2_d = out.theta_d*180/pi;
x2dot_d = out.thetadot_d*180/pi;
x2ddot_d = out.thetaddot_d*180/pi;
x2 = out.theta*180/pi;
```

```
x2dot = out.thetadot*180/pi;
x2ddot = out.thetaddot*180/pi;
x1 = out.z;
x1dot = out.zdot;
x1ddot = out.zddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda*180/pi;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda*180/pi;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda*180/pi;

%RMS
x1RMS = rms(x1_e);
x1dotRMS = rms(x1dot_e);
x1ddotRMS = rms(x1ddot_e);
x2RMS = rms(x2_e);
x2dotRMS = rms(x2dot_e);
x2ddotRMS = rms(x2ddot_e);
b1RMS = rms(b1hat-b1real);
b2RMS = rms(b2hat-b2real);
ErrorState_Altitude = ["x1~"; "x1dot~"; "x1ddot~"];
ErrorState_Roll = ["x2~"; "x2dot~"; "x2ddot~"];
RMS_Alt = [x1RMS; x1dotRMS; x1ddotRMS];
RMS_Roll = [x2RMS; x2dotRMS; x2ddotRMS];
StateRMStable = table(ErrorState_Altitude, RMS_Alt, ErrorState_Roll, RMS_Roll)
EstParam = ['b_1'; 'b_2'];
RMSb = [b1RMS; b2RMS];
EstRMStable = table(EstParam, RMSb)

%% Data Logging and Plotting

figure(1), clf
sgtitle('Altitude Tracking Error')
subplot(311)
plot(t,x1_e),grid on, xlabel('Time, s')
text(.7*tf,7.5E-5,['RMS = ' num2str(x1RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x1dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-3,['RMS = ' num2str(x1dotRMS)])
axis([0 tf -.001 .001])
title('Velo. Error, m/s')
subplot(313)
plot(t,x1ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5,['RMS = ' num2str(x1ddotRMS)])
axis([0 tf -1 1])
title('Accel. Error, m/s/s')

figure(2), clf
sgtitle('Roll Tracking Error')
subplot(311)
plot(t,x2_e),grid on, xlabel('Time, s')
text(.7*tf,1.5E-3,['RMS = ' num2str(x2RMS)])
title('Displ. Error, deg')
```

```
subplot(312)
plot(t,x2dot_e), grid on, xlabel('Time, s')
text(.7*tf,-0.025,['RMS = ' num2str(x2dotRMS)])
title('Velo. Error, deg/s')
subplot(313)
plot(t,x2ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5,['RMS = ' num2str(x2ddotRMS)])
axis([0 tf -1 1])
title('Accel. Error, deg/s/s')

figure(3), clf
sgtitle('Altitude Dynamics')
subplot(311)
plot(t,x1_d,t,x1), legend('z_d','z'), legend boxoff, grid on, xlabel('Time, s')
title('Altitude, m')
subplot(312)
plot(t,x1dot_d,t,x1dot), legend('zdot_d','zdot'), legend boxoff, grid on, xlabel('Time, s')
title('\delta Altitude, m/s')
subplot(313)
plot(t,x1ddot_d,t,x1ddot), legend('zddot_d','zddot'), legend boxoff, grid on, xlabel('Time,
s')
title('\delta^2 Altitude, m/s/s')

figure(4), clf
sgtitle('Roll Dynamics')
subplot(311)
plot(t,x2_d,t,x2), legend('theta_d','theta'), legend boxoff, grid on, xlabel('Time, s')
title('Roll, deg')
subplot(312)
plot(t,x2dot_d,t,x2dot), legend('thetadot_d','thetadot'), legend boxoff, grid on,
xlabel('Time, s')
title('\delta Roll, deg/s')
subplot(313)
plot(t,x2ddot_d,t,x2ddot), legend('thetaddot_d','thetaddot'), legend boxoff, grid on,
xlabel('Time, s')
axis([0 tf -10 10]);
title('\delta^2 Roll, deg/s/s')

figure(5), clf
sgtitle('Control Effort')
subplot(211)
plot(t,u1), grid on, xlabel('Time, s')
title('u_1')
subplot(212)
plot(t,u2), grid on, xlabel('Time, s')
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
plot(t,s1,t,phi1,t,-phi1), legend('s_1','\phi_1','-\phi_1'), legend boxoff, grid on,
xlabel('Time, s')
title('Altitude')
subplot(212)
```

```
plot(t,s2,t,phi2,t,-phi2), legend('s_2','\phi_2','-\phi_2'), legend boxoff, grid on,
xlabel('Time, s')
title('Roll')

figure(7), clf
sgtitle('Switching Gain, K')
subplot(211)
plot(t,K1), grid on, xlabel('Time, s')
title('K_z')
subplot(212)
plot(t,K2), grid on, xlabel('Time, s')
title('K_\theta')

figure(8), clf
sgtitle('Estimated Control Input Gain, b')
subplot(211)
plot(t,b1hat), grid on, xlabel('Time, s')
title('b_1'), yline(b1real), axis([0 tf 0 1.1*b1real])
text(.7*tf,.75*b1real,['b_1_,_t_r_u_e = ' num2str(b1real)])
text(.7*tf,.5*b1real,['RMS = ' num2str(b1RMS)])
subplot(212)
plot(t,b2hat), grid on, xlabel('Time, s')
title('b_2'), yline(b2real), axis([0 tf 0 1.1*b2real])
text(.7*tf,.75*b2real,['b_2_,_t_r_u_e = ' num2str(b2real)])
text(.7*tf,.5*b2real,['RMS = ' num2str(b2RMS)])

%% Altitude and Roll Control w/ State Measurement Noise - Control Input Gain Estimation
 % Actuator Dynamics neglected
 % One thrust source on either side of COM
 % "MFSMC_AltitudeRollControl_b_est_SMN.slx"
clear all, clc


tf = 10;

%%%Incorporate integral term into sliding surface%%%
% ' = 0' for s = lambda*x~ + xdot~ + int(x~)
% ' = 1' for s = lambda*x~ + xdot~
alt_SlidingSurface = 0;

%%%Aircraft Params%%%
m = 1; %0.02; %Craft Mass, kg
I = 1; %0.035; %Craft Inertia
L = 1; %0.069; %Craft Moment Arm from COM to Center of Thrust, m
Vmax = 160; %Max Voltage to Actuator
Vmin = -90; %Min Voltage to Actuator
Voff = 35; %Offset voltage to account for bimorph polarization
Vupper = Vmax-Voff;

%%%Sensor Noise Characteristics%%%
    % Taken from MFSMC_Sreeraj
    % Param - [x_accel y_accel z_accel x_gyro y_gyro z_gyro]
noise_mean = [0.02 0 -9.81 -0.004 0 0.0017];
noise_Vpp = [0.35 0.3 0.4 0.0157 0.0139 0.017];
noise_stddev = [0.0583 0.05 0.06 0.0026 0.0023 0.0029];
```

```
%%%IC%%%
g1 = 4;
g2 = 0.05;
z0 = g1*[pi/2 0]; %Height, m
theta0 = g2*[pi/2 0]; %Roll, rad

%%%MFSMC Params%%%
n = 2; %system order
su1 = 0.2;
su2 = 0.2;
su = [su1 su2];
lambda1 = 4;
lambda2 = 3;
eta1 = 0.0006;
eta2 = 0.00022;
lambda1_new = (eta1/max(noise_Vpp))^(1/n); %max value
lambda2_new = (eta2/max(noise_Vpp))^(1/n); %max value
lambda = [lambda1_new lambda2_new];
eta1_new = eta1 + (max(noise_stddev)/eta1)*(lambda(1)/2);
eta2_new = eta2 + (max(noise_stddev)/eta2)*(lambda(2)/2);
eta = [eta1_new eta2_new];
phi1_0 = eta1_new/lambda1_new;
phi2_0 = eta2_new/lambda2_new;
phi0 = [phi1_0 phi2_0];

b1_real = 3;
b2_real = 2;
b_real = [b1_real b2_real];
b1hat0 = 0.8;
b2hat0 = 0.8;
bhat0 = [b1hat0 b2hat0];
lambda1_0 = 80;
lambda2_0 = 80;
lambda0 = [lambda1_0 lambda2_0];
k1_0 = 500;
k2_0 = 500;
k0 = [k1_0 k2_0];
P_b1_0 = 10;
P_b2_0 = 10;
P_b0 = [P_b1_0 P_b2_0];

out = sim('MFSMC_AltitudeRollControl_b_est_SMN.slx');
%Simulink.sdi.view

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
b1hat = out.b1hat;
```

```
b2hat = out.b2hat;
b1real = out.b1real;
b2real = out.b2real;
s1 = out.s1;
s2 = out.s2;
x1_d = out.z_d;
x1dot_d = out.zdot_d;
x1ddot_d = out.zddot_d;
x2_d = out.theta_d*180/pi;
x2dot_d = out.thetadot_d*180/pi;
x2ddot_d = out.thetaddot_d*180/pi;
x2 = out.theta*180/pi;
x2dot = out.thetadot*180/pi;
x2ddot = out.thetaddot*180/pi;
x1 = out.z;
x1dot = out.zdot;
x1ddot = out.zddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda*180/pi;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda*180/pi;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda*180/pi;

%RMS
x1RMS = rms(x1_e);
x1dotRMS = rms(x1dot_e);
x1ddotRMS = rms(x1ddot_e);
x2RMS = rms(x2_e);
x2dotRMS = rms(x2dot_e);
x2ddotRMS = rms(x2ddot_e);
b1RMS = rms(b1hat-b1real);
b2RMS = rms(b2hat-b2real);
ErrorState_Altitude = ["x1~"; "x1dot~"; "x1ddot~"];
ErrorState_Roll = ["x2~"; "x2dot~"; "x2ddot~"];
RMS_Alt = [x1RMS; x1dotRMS; x1ddotRMS];
RMS_Roll = [x2RMS; x2dotRMS; x2ddotRMS];
StateRMStable = table(ErrorState_Altitude, RMS_Alt, ErrorState_Roll, RMS_Roll)
EstParam = ['b_1'; 'b_2'];
RMSb = [b1RMS; b2RMS];
EstRMStable = table(EstParam, RMSb)

%% Data Logging and Plotting

figure(1), clf
sgtitle('Altitude Tracking Error')
subplot(311)
plot(t,x1_e),grid on, xlabel('Time, s')
text(.7*tf,7.5E-5,['RMS = ' num2str(x1RMS)])
title('Displ. Error, m')
subplot(312)
plot(t,x1dot_e), grid on, xlabel('Time, s')
text(.7*tf,.5E-3,['RMS = ' num2str(x1dotRMS)])
axis([0 tf -.001 .001])
title('Velo. Error, m/s')
```

```
subplot(313)
plot(t,x1ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5,['RMS = ' num2str(x1ddotRMS)])
axis([0 tf -1 1])
title('Accel. Error, m/s/s')

figure(2), clf
sgtitle('Roll Tracking Error')
subplot(311)
plot(t,x2_e),grid on, xlabel('Time, s')
text(.7*tf,1.5E-3,['RMS = ' num2str(x2RMS)])
title('Displ. Error, deg')
subplot(312)
plot(t,x2dot_e), grid on, xlabel('Time, s')
text(.7*tf,-0.025,['RMS = ' num2str(x2dotRMS)])
title('Velo. Error, deg/s')
subplot(313)
plot(t,x2ddot_e), grid on, xlabel('Time, s')
text(.7*tf,.5,['RMS = ' num2str(x2ddotRMS)])
axis([0 tf -1 1])
title('Accel. Error, deg/s/s')

figure(3), clf
sgtitle('Altitude Dynamics')
subplot(311)
plot(t,x1_d,t,x1), legend('z_d','z'), legend boxoff, grid on, xlabel('Time, s')
title('Altitude, m')
subplot(312)
plot(t,x1dot_d,t,x1dot), legend('zdot_d','zdot'), legend boxoff, grid on, xlabel('Time, s')
title('\delta Altitude, m/s')
subplot(313)
plot(t,x1ddot_d,t,x1ddot), legend('zddot_d','zddot'), legend boxoff, grid on, xlabel('Time,
s')
title('\delta^2 Altitude, m/s/s')

figure(4), clf
sgtitle('Roll Dynamics')
subplot(311)
plot(t,x2_d,t,x2), legend('theta_d','theta'), legend boxoff, grid on, xlabel('Time, s')
title('Roll, deg')
subplot(312)
plot(t,x2dot_d,t,x2dot), legend('thetadot_d','thetadot'), legend boxoff, grid on,
xlabel('Time, s')
title('\delta Roll, deg/s')
subplot(313)
plot(t,x2ddot_d,t,x2ddot), legend('thetaddot_d','thetaddot'), legend boxoff, grid on,
xlabel('Time, s')
axis([0 tf -10 10]);
title('\delta^2 Roll, deg/s/s')

figure(5), clf
sgtitle('Control Effort')
subplot(211)
plot(t,u1), grid on, xlabel('Time, s')
title('u_1')
```

```
subplot(212)
plot(t,u2), grid on, xlabel('Time, s')
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
plot(t,s1,t,phi1,t,-phi1), legend('s_1','\phi_1','-\phi_1'), legend boxoff, grid on,
xlabel('Time, s')
title('Altitude')
subplot(212)
plot(t,s2,t,phi2,t,-phi2), legend('s_2','\phi_2','-\phi_2'), legend boxoff, grid on,
xlabel('Time, s')
title('Roll')

figure(7), clf
sgtitle('Switching Gain, K')
subplot(211)
plot(t,K1), grid on, xlabel('Time, s')
title('K_z')
subplot(212)
plot(t,K2), grid on, xlabel('Time, s')
title('K_\theta')

figure(8), clf
sgtitle('Estimated Control Input Gain, b')
subplot(211)
plot(t,b1hat), grid on, xlabel('Time, s')
title('b_1'), yline(b1real), axis([0 tf 0 1.1*b1real])
text(.7*tf,.75*b1real,['b_1_,_t_r_u_e = ' num2str(b1real)])
text(.7*tf,.5*b1real,['RMS = ' num2str(b1RMS)])
subplot(212)
plot(t,b2hat), grid on, xlabel('Time, s')
title('b_2'), yline(b2real), axis([0 tf 0 1.1*b2real])
text(.7*tf,.75*b2real,['b_2_,_t_r_u_e = ' num2str(b2real)])
text(.7*tf,.5*b2real,['RMS = ' num2str(b2RMS)])

%% Altitude, Lateral, and Roll Control w/ State Measurement Noise - Control Input Gain
Estimation
 % Actuator Dynamics neglected
 % One thrust source on either side of COM
 % "MFSMC_AltitudeLateralRollControl_b_est_SMN.slx"
clear all, clc

tf = 10;

%%%Incorporate integral term into sliding surface%%%
% ' = 0' for s = lambda*x~ + xdot~ + int(x~)
% ' = 1' for s = lambda*x~ + xdot~
alt_SlidingSurface = 0;

%%%Aircraft Params%%%
m = 1; %0.02; %Craft Mass, kg
I = 1; %0.035; %Craft Inertia
Lyf = 1; %0.069; %Craft Moment Arm from COM to Center of Thrust, m
```

```
Lyr = 1;
Vmax = 160; %Max Voltage to Actuator
Vmin = -90; %Min Voltage to Actuator
Voff = 35; %Offset voltage to account for bimorph polarization
Vupper = Vmax-Voff;

%%%Sensor Noise Characteristics%%%
    % Taken from MFSMC_Sreeraj
    % Param - [x_accel y_accel z_accel x_gyro y_gyro z_gyro]
noise_mean = [0.02 0 -9.81 -0.004 0 0.0017];
noise_Vpp = [0.35 0.3 0.4 0.0157 0.0139 0.017];
noise_stddev = [0.0583 0.05 0.06 0.0026 0.0023 0.0029];

%%%IC%%%
g1 = 0;
g2 = 0;
g3 = 0;
z0 = g1*[pi/2 0]; %Height, m
theta0 = g2*[pi/2 0]; %Roll, rad
y0 = g3*[pi/2 0];

%%%MFSMC Params%%%
n = 2; %system order
su1 = 0.2;
su2 = 0.2;
su3 = 0.2;
su = diag([su1 su2 su3]);
lambda1 = 4;
lambda2 = 3;
lambda3 = 4;
eta1 = 0.0006;
eta2 = 0.00022;
eta3 = 0.0006;
lambda1_new = lambda1;%(eta1/max(noise_Vpp))^(1/n); %max value
lambda2_new = lambda2;%(eta2/max(noise_Vpp))^(1/n); %max value
lambda3_new = lambda3;%(eta3/max(noise_Vpp))^(1/n); %max value
lambda = diag([lambda1_new lambda2_new lambda3_new]);
eta1_new = eta1;% + (max(noise_stddev)/eta1)*(lambda(1)/2);
eta2_new = eta2;% + (max(noise_stddev)/eta2)*(lambda(2)/2);
eta3_new = eta3;% + (max(noise_stddev)/eta3)*(lambda(3)/2);
eta = [eta1_new eta2_new eta3_new]';
phi1_0 = eta1_new/lambda1_new;
phi2_0 = eta2_new/lambda2_new;
phi3_0 = eta3_new/lambda3_new;
phi0 = [phi1_0 phi2_0 phi3_0]';

%Control Input Gain: Outputs, p=3 & pseudo Inputs, m=3 => [b]:[3x3]
b11_real = 3; b12_real = 1; b13_real = 1;
b21_real = 1; b22_real = 2; b23_real = 1;
b31_real = 1; b32_real = 1; b33_real = 3;
b_real = [b11_real b12_real b13_real;
          b21_real b22_real b23_real;
          b31_real b32_real b33_real];
bhat0 = 0.8;
bhat0 = bhat0*eye(3,3);
```

```
lambda1_0 = 80;
lambda2_0 = 80;
lambda3_0 = 80;
lambda0 = [lambda1_0 lambda2_0 lambda3_0];
k1_0 = 500;
k2_0 = 500;
k3_0 = 500;
k0 = [k1_0 k2_0 k3_0];
P_b1_0 = 10;
P_b2_0 = 10;
P_b3_0 = 10;
P_b0 = [P_b1_0 P_b2_0 P_b3_0];

out = sim('MFSMC_AltitudeRollControl_b_est_SMN.slx');
%Simulink.sdi.view

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
b1hat = out.b1hat;
b2hat = out.b2hat;
b1real = out.b1real;
b2real = out.b2real;
s1 = out.s1;
s2 = out.s2;
x1_d = out.z_d;
x1dot_d = out.zdot_d;
x1ddot_d = out.zddot_d;
x2_d = out.theta_d*180/pi;
x2dot_d = out.thetadot_d*180/pi;
x2ddot_d = out.thetaddot_d*180/pi;
x2 = out.theta*180/pi;
x2dot = out.thetadot*180/pi;
x2ddot = out.thetaddot*180/pi;
x1 = out.z;
x1dot = out.zdot;
x1ddot = out.zddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda*180/pi;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda*180/pi;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda*180/pi;

%RMS
x1RMS = rms(x1_e);
x1dotRMS = rms(x1dot_e);
x1ddotRMS = rms(x1ddot_e);
```

```
x2RMS = rms(x2_e);
x2dotRMS = rms(x2dot_e);
x2ddotRMS = rms(x2ddot_e);
b1RMS = rms(b1hat-b1real);
b2RMS = rms(b2hat-b2real);
ErrorState_Altitude = ["x1~"; "x1dot~"; "x1ddot~"];
ErrorState_Roll = ["x2~"; "x2dot~"; "x2ddot~"];
RMS_Alt = [x1RMS; x1dotRMS; x1ddotRMS];
RMS_Roll = [x2RMS; x2dotRMS; x2ddotRMS];
StateRMStable = table(ErrorState_Altitude, RMS_Alt, ErrorState_Roll, RMS_Roll)
EstParam = ['b_1'; 'b_2'];
RMSb = [b1RMS; b2RMS];
EstRMStable = table(EstParam, RMSb)
%% Thrust Vector Balance - LQR
tf = 15;
Iy = 1;
Iz = 1;
mm = 1.5; % motor mass
lm = 3; % motor moment arm
mc = 1; % counterweight mass
lc = 3; % counterweight moment arm
mu_z = 0.05; % friction constant
mu_y = 0.05; % frinction constant
g = 9.81;
Kt = 5;

syms x1 x2 x3 x4 x5 x6 x7 x8 u1 u2
eqn1 = x2;
eqn2 = (1/Iy)*(mc*g*lc + mm*g*lm + Kt*u1*cos(u2)*lm - mu_y*x2);
eqn3 = x4;
eqn4 = (1/Iz)*(Kt*u1*sin(u2)*lm - mu_z*x4);


A = [diff(eqn1,x1) diff(eqn1,x2) diff(eqn1,x3) diff(eqn1,x4) 0 0 0 0;
     diff(eqn2,x1) diff(eqn2,x2) diff(eqn2,x3) diff(eqn2,x4) 0 0 0 0;
     diff(eqn3,x1) diff(eqn3,x2) diff(eqn3,x3) diff(eqn3,x4) 0 0 0 0;
     diff(eqn4,x1) diff(eqn4,x2) diff(eqn4,x3) diff(eqn4,x4) 0 0 0 0;
     -1 0 0 0 0 0 0 0;
     0 0 -1 0 0 0 0 0;
     -1 0 0 0 0 0 -1 0;
     0 0 -1 0 0 0 0 -1];
B = [diff(eqn1,u1) diff(eqn1,u2);
    diff(eqn2,u1) diff(eqn2,u2);
    diff(eqn3,u1) diff(eqn3,u2);
    diff(eqn4,u1) diff(eqn4,u2);
    0 0;
    0 0;
    0 0;
    0 0];
C = [1 0 0 0 0 0 0 0; 0 1 0 0 0 0 0 0; 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0;
     0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0];
D = [0];

A = double(A);
B = double(subs(B,[u1 u2],[3 0]));
```

```
sysC = ss(A,B,C,D);

Ts = 0.005;
sysD = c2d(sysC,Ts);

%%LQR

Q = eye(8,8); R = 1;
[K,S,P] = dlqr(sysD.A,sysD.B,Q,R)

wn_d = 2; zeta_d = 1.2;
data = sim("TVBalance_LQR.slx")

theta = data.simout(:,1);
thetadot = data.simout(:,2);
psi = data.simout(:,3);
psidot = data.simout(:,4);
xe_theta = data.simout(:,5);
xe_psi = data.simout(:,6);
e_theta = data.simout(:,7);
e_psi = data.simout(:,8);
Vdc = data.simout(:,9);
MotorRPM_LQR = (Vdc/5)*(4350/2) + (1/5)*(4350/2);
alpha = data.simout(:,10);
edot_theta = data.simout(:,11);
edot_psi = data.simout(:,12);
r_theta = data.simout(:,13);
r_psi = data.simout(:,14);
t = data.simout(:,15);
eddot_theta = data.simout(:,16);
eddot_psi = data.simout(:,17);

%% Data Logging and Plotting

%RMS
x1RMS = rms(e_theta)
x1dotRMS = rms(edot_theta)
x1ddotRMS = rms(eddot_theta)

x2RMS = rms(e_psi)
x2dotRMS = rms(edot_psi)
x2ddotRMS = rms(eddot_psi)

figure(1), clf
sgtitle('Pitch Tracking Error')
subplot(211)
plot(t,e_theta),grid on, ylabel('deg'), xlabel('Time, s')
title('Displ. Error')
subplot(212)
plot(t,edot_theta), grid on,ylabel('deg/s'), xlabel('Time, s')
title('Velo. Error')

figure(2), clf
sgtitle('Yaw Tracking Error')
```

```
subplot(211)
plot(t,e_psi),grid on,ylabel('deg'), xlabel('Time, s')
title('Displ. Error')
subplot(212)
plot(t,edot_psi), grid on,ylabel('deg/s'), xlabel('Time, s')
title('Velo. Error')

figure(3), clf
sgtitle('State Dynamics')
subplot(411)
plot(t,theta,t,r_theta), legend('\theta','\theta_d'), legend boxoff, grid on,
ylabel('deg'),xlabel('Time, s')
title('Pitch Displ.')
subplot(412)
plot(t,psi,t,r_psi), legend('\psi','\psi_d'), legend boxoff, grid on, ylabel('deg'),
xlabel('Time, s')
title('Yaw Displ.')
subplot(413)
plot(t,thetadot), legend('\thetadot'), legend boxoff, grid on, ylabel('deg/s'),xlabel('Time,
s')
title('Pitch Velo.')
subplot(414)
plot(t,psidot), legend('\psidot'), legend boxoff, grid on, ylabel('deg/s'), xlabel('Time, s')
title('Yaw Velo.')

figure(4), clf
sgtitle('Control Effort')
subplot(211)
plot(t,MotorRPM_LQR), grid on, ylabel('Volt'), xlabel('Time, s')
title('Motor RPM')
subplot(212)
plot(t,alpha*20), grid on, ylabel('deg'), xlabel('Time, s')
title('Servo Angle')

%% Thrust Vector Balance - MFSMC

Iy = 1;
Iz = 1;
mm = 1.5; % motor mass
lm = 3; % motor moment arm
mc = 1; % counterweight mass
lc = 3; % counterweight moment arm
mu_z = 0.05; % friction constant
mu_y = 0.05; % frinction constant
g = 9.81;
Kt = 5;
alpha = 0.21;

tf = 15;
%IC
theta0 = [0 -1]; % Pitch
psi0 = [0 0]; % Yaw

%Pre-filter Input
wn_d=2; zeta_d=1.2;
```

```
%Control Input Filter
wn_u=10; zeta_u=.5;

%Incorporate integral term into sliding surface%%%
% ' = 0' for s = lambda*x~ + xdot~ + int(x~)
% ' = 1' for s = lambda*x~ + xdot~
alt_SlidingSurface = 1;

%%%MFSMC Params%%%
su1 = 0.2;
su2 = 0.2;
su = [su1 su2];
lambda1 = 1;
lambda2 = 1.25;
lambda = [lambda1 lambda2];
eta1 = 0.1;
eta2 = 0.1;
sigma = 0.025;
eta = [eta1+(sigma/0.1)*lambda1*0.5 eta2+(sigma/0.1)*lambda1*0.5];
phi1_0 = eta1/lambda1;
phi2_0 = eta2/lambda2;
phi0 = [phi1_0 phi2_0];

b1_real = 1.5;
b2_real = 1;
b_real = [b1_real b2_real];
bup = 2;
blow = 1;
beta0 = [sqrt(bup/blow) sqrt(bup/blow)];
b1hat0 = 0.8;%sqrt(bup*blow);
b2hat0 = 0.8;%sqrt(bup*blow);
bhat0 = [b1hat0 b2hat0];
lambda1_0 = 80;
lambda2_0 = 80;
lambda0 = [lambda1_0 lambda2_0];
k1_0 = 500;
k2_0 = 500;
k0 = [k1_0 k2_0];
P_b1_0 = 10;
P_b2_0 = 10;
P_b0 = [P_b1_0 P_b2_0];

out = sim('ThrustVectorBalance_SM_R21.slx');

t = out.tout;
u1 = out.u1;
u2 = out.u2;
phi1 = out.phi1;
phi2 = out.phi2;
K1 = out.K1;
K2 = out.K2;
s1dot = out.s1dot;
s2dot = out.s2dot;
s1 = out.s1;
s2 = out.s2;
```

```
x1_d = out.x1_d;
x1dot_d = out.x1dot_d;
x1ddot_d = out.x1ddot_d;
x2_d = out.x2_d;
x2dot_d = out.x2dot_d;
x2ddot_d = out.x2ddot_d;
x2 = out.x2;
x2dot = out.x2dot;
x2ddot = out.x2ddot;
x1 = out.x1;
x1dot = out.x1dot;
x1ddot = out.x1ddot;
x1_e = out.x1_tilda;
x2_e = out.x2_tilda;
x1dot_e = out.x1dot_tilda;
x2dot_e = out.x2dot_tilda;
x1ddot_e = out.x1ddot_tilda;
x2ddot_e = out.x2ddot_tilda;
u1Control = out.u1Control;
u2Control = out.u2Control;
b1hat = out.b1hat;
b2hat = out.b2hat;
MotorRPM = out.MotorRPM;
ServoAngle = out.ServoAngle;

%% Data Logging and Plotting

%RMS
x1RMS = rms(x1_e)
x1dotRMS = rms(x1dot_e)
x1ddotRMS = rms(x1ddot_e)
x2RMS = rms(x2_e)
x2dotRMS = rms(x2dot_e)
x2ddotRMS = rms(x2ddot_e)

figure(1), clf
sgtitle('State x_1 Tracking Error')
subplot(311)
plot(t,x1_e),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x1RMS)])
title('Displ. Error, deg')
subplot(312)
plot(t,x1dot_e), grid on, xlabel('Time, s')
text(.7*tf,.4,['RMS = ' num2str(x1dotRMS)])
title('Velo. Error, deg/s')
subplot(313)
plot(t,x1ddot_e), grid on, xlabel('Time, s')
text(.7*tf,-1,['RMS = ' num2str(x1ddotRMS)])
title('Accel. Error, deg/s/s')

figure(2), clf
sgtitle('State x_2 Tracking Error')
subplot(311)
plot(t,x2_e),grid on, xlabel('Time, s')
text(.7*tf,-1,['RMS = ' num2str(x2RMS)])
```

```
title('Displ. Error, deg')
subplot(312)
plot(t,x2dot_e), grid on, xlabel('Time, s')
text(.5*tf,.2,['RMS = ' num2str(x2dotRMS)])
title('Velo. Error, deg/s')
subplot(313)
plot(t,x2ddot_e), grid on, xlabel('Time, s')
text(.5*tf,-.5,['RMS = ' num2str(x2ddotRMS)])
title('Accel. Error, deg/s/s')

figure(3), clf
sgtitle('State x_1 Dynamics')
subplot(311)
plot(t,x1_d,t,x1), legend('x_1_,_d','x_1'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., deg')
subplot(312)
plot(t,x1dot_d,t,x1dot), legend('x_1dot_d','x_1dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., deg/s')
subplot(313)
plot(t,x1ddot_d,t,x1ddot), legend('x_1ddot_d','x_1ddot'), legend boxoff, grid on,
xlabel('Time, s')
title('Accel., deg/s/s')

figure(4), clf
sgtitle('State x_2 Dynamics')
subplot(311)
plot(t,x2_d,t,x2), legend('x_2_,_d','x_2'), legend boxoff, grid on, xlabel('Time, s')
title('Displ., deg')
subplot(312)
plot(t,x2dot_d,t,x2dot), legend('x_2dot_d','x_2dot'), legend boxoff, grid on, xlabel('Time,
s')
title('Velo., deg/s')
subplot(313)
plot(t,x2ddot_d,t,x2ddot), legend('x_2ddot_d','x_2ddot'), legend boxoff, grid on,
xlabel('Time, s')
title('Accel., deg/s/s')

figure(5), clf
sgtitle('Control Effort')
subplot(211)
yyaxis left
plot(t,MotorRPM), grid on, xlabel('Time, s')
ylabel('Motor RPM')
yyaxis right
plot(t,u1)
ylabel('Calc. Control')
legend('Actuator Input','Calc. Control'), legend boxoff
title('u_1')
subplot(212)
yyaxis left
plot(t,ServoAngle), grid on, xlabel('Time, s')
ylabel('Servo Angle, deg')
yyaxis right
plot(t,u2)
```

```
ylabel('Calc. Control')
legend('Actuator Input','Calc. Control'), legend boxoff
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
plot(t,s1,t,phi1,t,-phi1), legend('s_1','\phi_1','-\phi_1'), legend boxoff, grid on,
xlabel('Time, s')
title('x_1')
subplot(212)
plot(t,s2,t,phi2,t,-phi2), legend('s_2','\phi_2','-\phi_2'), legend boxoff, grid on,
xlabel('Time, s')
title('x_2')

figure(7), clf
sgtitle('Switching Gain, K')
subplot(211)
plot(t,K1), grid on, xlabel('Time, s')
title('K_1')
subplot(212)
plot(t,K2), grid on, xlabel('Time, s')
title('K_2')

figure(8), clf
sgtitle('Estimated Control Input Gain')
subplot(211)
plot(t,b1hat), grid on, xlabel('Time, s')
title('b_1')
subplot(212)
plot(t,b2hat), grid on, xlabel('Time, s')
title('b_2')

%% Test Data Analysys

t_test = table2array(TestData1(:,1));
x1_ref_test = table2array(TestData1(:,2));
x1_test = table2array(TestData1(:,3));
x2_ref_test = table2array(TestData1(:,4));
x2_test = table2array(TestData1(:,5));
x1dot_ref_test = table2array(TestData1(:,6));
x1dot_test = table2array(TestData1(:,7));
x2dot_ref_test = table2array(TestData1(:,8));
x2dot_test = table2array(TestData1(:,9));
x1ddot_ref_test = table2array(TestData1(:,10));
x1ddot_test = table2array(TestData1(:,11));
x2ddot_ref_test = table2array(TestData1(:,12));
x2ddot_test = table2array(TestData1(:,13));
u1MFSMC_test = table2array(TestData1(:,14));
u1Control_test = table2array(TestData1(:,15));
u2MFSMC_test = table2array(TestData1(:,16));
u2Control_test = table2array(TestData1(:,17));
xe1_test = table2array(TestData1(:,18));
xe2_test = table2array(TestData1(:,19));
e1_test = table2array(TestData1(:,20));
```

```
e2_test = table2array(TestData1(:,21));
e1dot_test = table2array(TestData1(:,22));
e2dot_test = table2array(TestData1(:,23));
e1ddot_test = table2array(TestData1(:,24));
e2ddot_test = table2array(TestData1(:,25));
b1hat_test = table2array(TestData1(:,26));
b2hat_test = table2array(TestData1(:,27));
s1dot_test = table2array(TestData1(:,28));
s2dot_test = table2array(TestData1(:,29));
s1_test = table2array(TestData1(:,30));
phi1_test = table2array(TestData1(:,31));
s2_test = table2array(TestData1(:,33));
phi2_test = table2array(TestData1(:,34));
MotorRPM_test = (u1Control_test/255)*(4350/2);
ServoAngle_test = u2Control_test;


t_LQR = table2array(TestDataLQR(1:2999,1));
x1_ref_LQR = table2array(TestDataLQR(1:2999,2)) + 22*ones(2999,1);
x1_LQR = table2array(TestDataLQR(1:2999,3)) + 22*ones(2999,1);
x2_ref_LQR = table2array(TestDataLQR(1:2999,4));
x2_LQR = table2array(TestDataLQR(1:2999,5));
x1dot_ref_LQR = table2array(TestDataLQR(1:2999,6));
x1dot_LQR = table2array(TestDataLQR(1:2999,7));
x2dot_ref_LQR = table2array(TestDataLQR(1:2999,8));
x2dot_LQR = table2array(TestDataLQR(1:2999,9));
x1ddot_ref_LQR = table2array(TestDataLQR(1:2999,10));
x1ddot_LQR = table2array(TestDataLQR(1:2999,11));
x2ddot_ref_LQR = table2array(TestDataLQR(1:2999,12));
x2ddot_LQR = table2array(TestDataLQR(1:2999,13));
u1LQR = table2array(TestDataLQR(1:2999,14));
u1Control_LQR = table2array(TestDataLQR(1:2999,15));
u2LQR = table2array(TestDataLQR(1:2999,16));
u2Control_LQR = table2array(TestDataLQR(1:2999,17));
xe1_LQR = table2array(TestDataLQR(1:2999,18));
xe2_LQR = table2array(TestDataLQR(1:2999,19));
e1_LQR = table2array(TestDataLQR(1:2999,20));
e2_LQR = table2array(TestDataLQR(1:2999,21));
e1dot_LQR = table2array(TestDataLQR(1:2999,22));
e2dot_LQR = table2array(TestDataLQR(1:2999,23));
e1ddot_LQR = table2array(TestDataLQR(1:2999,24));
e2ddot_LQR = table2array(TestDataLQR(1:2999,25));
MotorRPM_LQR = (u1Control_LQR/255)*(4350/2);
ServoAngle_LQR = u2Control_LQR;

%RMS
x1RMS_test = rms(e1_test)
x1dotRMS_test = rms(e1dot_test)
x1ddotRMS_test = rms(e1ddot_test)
x2RMS_test = rms(e2_test)
x2dotRMS_test = rms(e2dot_test)
x2ddotRMS_test = rms(e2ddot_test)
x1RMS_LQR = rms(e1_LQR)
x1dotRMS_LQR = rms(e1dot_LQR)
x1ddotRMS_LQR = rms(e1ddot_LQR)
```

```
x2RMS_LQR = rms(e2_LQR)
x2dotRMS_LQR = rms(e2dot_LQR)
x2ddotRMS_LQR = rms(e2ddot_LQR)

figure(1), clf
sgtitle('State x_1 Tracking Error - MFSMC')
subplot(311)
plot(t_test,e1_test),grid on, xlabel('Time, s')
text(.7*tf,.5E-4,['RMS = ' num2str(x1RMS_test)])
title('Displ. Error, deg')
subplot(312)
plot(t_test,e1dot_test), grid on, xlabel('Time, s')
text(.7*tf,-1.5,['RMS = ' num2str(x1dotRMS_test)])
title('Velo. Error, deg/s')
subplot(313)
plot(t_test,e1ddot_test), grid on, xlabel('Time, s')
text(.7*tf,-1,['RMS = ' num2str(x1ddotRMS_test)])
title('Accel. Error, deg/s/s')

figure(2), clf
sgtitle('State x_2 Tracking Error - MFSMC')
subplot(311)
plot(t_test,e2_test),grid on, xlabel('Time, s')
text(.7*tf,1,['RMS = ' num2str(x2RMS_test)])
title('Displ. Error, deg')
subplot(312)
plot(t_test,e2dot_test), grid on, xlabel('Time, s')
text(.25*tf,-.3,['RMS = ' num2str(x2dotRMS_test)])
title('Velo. Error, deg/s')
subplot(313)
plot(t_test,e2ddot_test), grid on, xlabel('Time, s')
text(.25*tf,-.25,['RMS = ' num2str(x2ddotRMS_test)])
title('Accel. Error, deg/s/s')

figure(3), clf
sgtitle('State x_1 Dynamics - MFSMC')
subplot(311)
plot(t_test,x1_ref_test,t_test,x1_test), legend('x_1_,_d','x_1'), legend boxoff, grid on,
xlabel('Time, s')
title('Displ., deg')
subplot(312)
plot(t_test,x1dot_ref_test,t_test,x1dot_test), legend('x_1dot_d','x_1dot'), legend boxoff,
grid on, xlabel('Time, s')
title('Velo., deg/s')
subplot(313)
plot(t_test,x1ddot_ref_test,t_test,x1ddot_test), legend('x_1ddot_d','x_1ddot'), legend boxoff,
grid on, xlabel('Time, s')
title('Accel., deg/s/s')

figure(4), clf
sgtitle('State x_2 Dynamics - MFSMC')
subplot(311)
plot(t_test,x2_ref_test,t_test,x2_test), legend('x_2_,_d','x_2'), legend boxoff, grid on,
xlabel('Time, s')
title('Displ., deg')
```

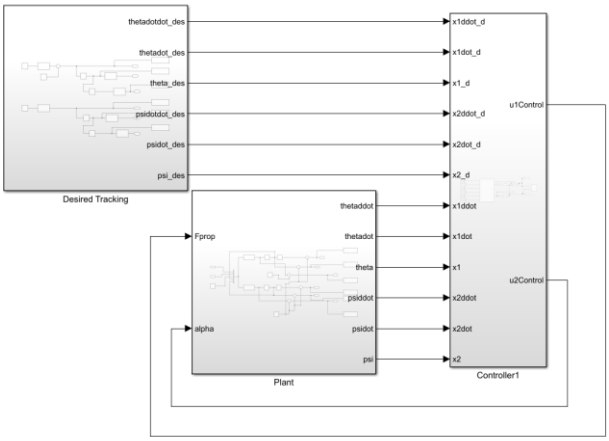
```
subplot(312)
plot(t_test,x2dot_ref_test,t_test,x2dot_test), legend('x_2dot_d','x_2dot'), legend boxoff,
grid on, xlabel('Time, s')
title('Velo., deg/s')
subplot(313)
plot(t_test,x2ddot_ref_test,t_test,x2ddot_test), legend('x_2ddot_d','x_2ddot'), legend boxoff,
grid on, xlabel('Time, s')
title('Accel., deg/s/s')

figure(5), clf
sgtitle('Control Effort - MFSMC')
subplot(211)
yyaxis left
plot(t_test,MotorRPM_test), grid on, xlabel('Time, s')
ylabel('Motor RPM')
yyaxis right
plot(t_test,u1MFSMC_test)
ylabel('Calc. Control')
legend('Actuator Input','Calc. Control'), legend boxoff
title('u_1')
subplot(212)
yyaxis left
plot(t_test,ServoAngle_test), grid on, xlabel('Time, s')
ylabel('Servo Angle, deg')
yyaxis right
plot(t_test,u2MFSMC_test)
ylabel('Calc. Control')
legend('Actuator Input','Calc. Control'), legend boxoff
title('u_2')

figure(6), clf
sgtitle('Boundary Layer')
subplot(211)
plot(t_test,s1_test,t_test,phi1_test,t_test,-phi1_test), legend('s_1','\phi_1','-\phi_1'),
legend boxoff, grid on, xlabel('Time, s')
title('x_1')
subplot(212)
plot(t_test,s2_test,t_test,phi2_test,t_test,-phi2_test), legend('s_2','\phi_2','-\phi_2'),
legend boxoff, grid on, xlabel('Time, s')
title('x_2')

figure(7), clf
sgtitle('Estimated Control Input Gain')
subplot(211)
plot(t_test,b1hat_test), grid on, xlabel('Time, s')
axis([0 tf 0 2.5])
title('b_1')
subplot(212)
plot(t_test,b2hat_test), grid on, xlabel('Time, s')
axis([0 tf 0 2.5])
title('b_2')

figure(8), clf
sgtitle('State x_1 Tracking Error - LQR')
subplot(311)
```

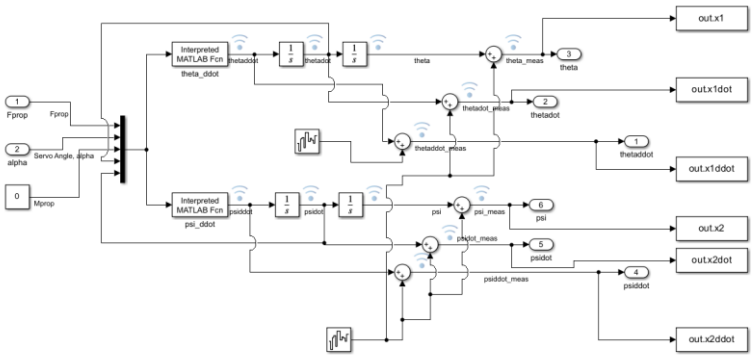
```
plot(t_LQR,e1_LQR),grid on, xlabel('Time, s')
text(.7*tf,-10,['RMS = ' num2str(x1RMS_LQR)])
title('Displ. Error, deg')
subplot(312)
plot(t_LQR,e1dot_LQR), grid on, xlabel('Time, s')
text(.7*tf,-5,['RMS = ' num2str(x1dotRMS_LQR)])
title('Velo. Error, deg/s')
subplot(313)
plot(t_LQR,e1ddot_LQR), grid on, xlabel('Time, s')
text(.7*tf,-2.5,['RMS = ' num2str(x1ddotRMS_LQR)])
title('Accel. Error, deg/s/s')

figure(9), clf
sgtitle('State x_2 Tracking Error - LQR')
subplot(311)
plot(t_LQR,e2_LQR),grid on, xlabel('Time, s')
text(.25*tf,-.4,['RMS = ' num2str(x2RMS_LQR)])
title('Displ. Error, deg')
subplot(312)
plot(t_LQR,e2dot_LQR), grid on, xlabel('Time, s')
text(.25*tf,-.15,['RMS = ' num2str(x2dotRMS_LQR)])
title('Velo. Error, deg/s')
subplot(313)
plot(t_LQR,e2ddot_LQR), grid on, xlabel('Time, s')
text(.25*tf,-.1,['RMS = ' num2str(x2ddotRMS_LQR)])
title('Accel. Error, deg/s/s')

figure(10), clf
sgtitle('State x_1 Dynamics - LQR')
subplot(311)
plot(t_LQR,x1_ref_LQR,t_LQR,x1_LQR), legend('x_1_,_d','x_1'), legend boxoff, grid on,
xlabel('Time, s')
title('Displ., deg')
subplot(312)
plot(t_LQR,x1dot_ref_LQR,t_LQR,x1dot_LQR), legend('x_1dot_d','x_1dot'), legend boxoff, grid
on, xlabel('Time, s')
title('Velo., deg/s')
subplot(313)
plot(t_LQR,x1ddot_ref_LQR,t_LQR,x1ddot_LQR), legend('x_1ddot_d','x_1ddot'), legend boxoff,
grid on, xlabel('Time, s')
title('Accel., deg/s/s')

figure(11), clf
sgtitle('State x_2 Dynamics - LQR')
subplot(311)
plot(t_LQR,x2_ref_LQR,t_LQR,x2_LQR), legend('x_2_,_d','x_2'), legend boxoff, grid on,
xlabel('Time, s')
title('Displ., deg')
subplot(312)
plot(t_LQR,x2dot_ref_LQR,t_LQR,x2dot_LQR), legend('x_2dot_d','x_2dot'), legend boxoff, grid
on, xlabel('Time, s')
title('Velo., deg/s')
subplot(313)
plot(t_LQR,x2ddot_ref_LQR,t_LQR,x2ddot_LQR), legend('x_2ddot_d','x_2ddot'), legend boxoff,
grid on, xlabel('Time, s')
```

```
title('Accel., deg/s/s')

figure(12), clf
sgtitle('Control Effort - LQR')
subplot(211)
plot(t_LQR,MotorRPM_LQR), grid on, xlabel('Time, s')
ylabel('Motor RPM')
title('u_1')
subplot(212)
plot(t_LQR,ServoAngle_LQR), grid on, xlabel('Time, s')
ylabel('Servo Angle, deg')
title('u_2')
```

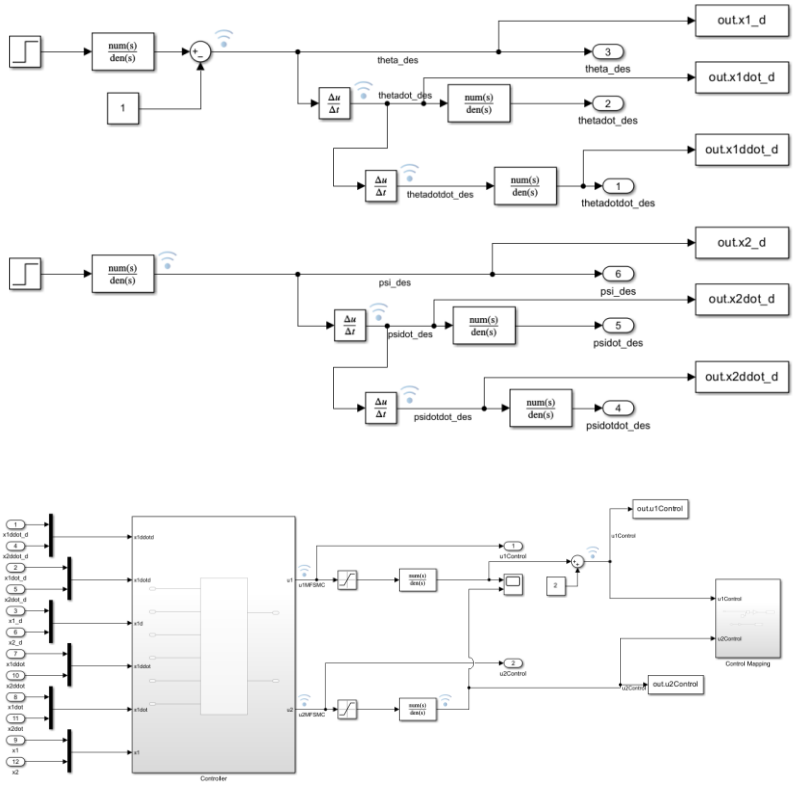
Simulink Diagrams

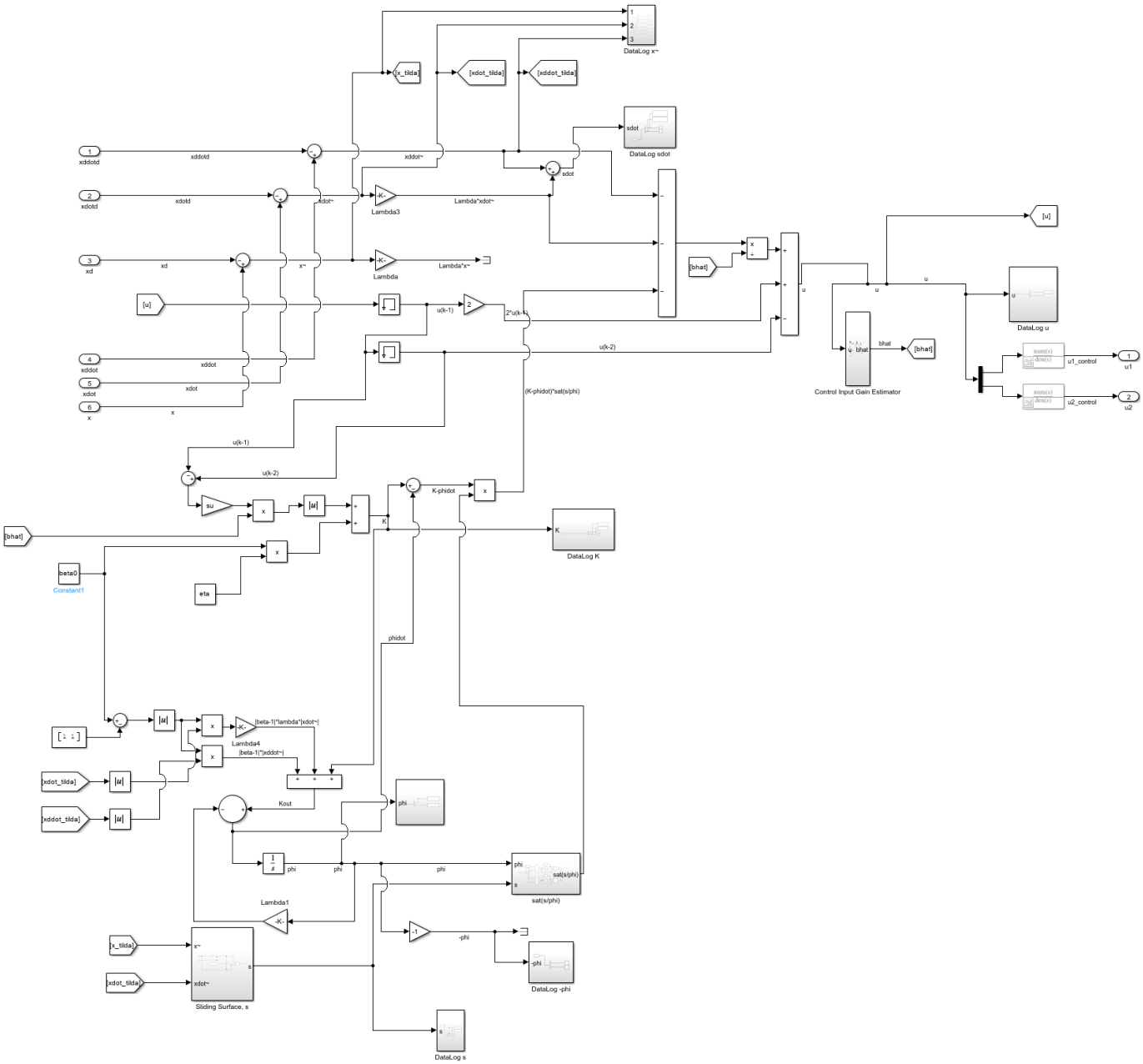
- **TV -Balance MFSMC**



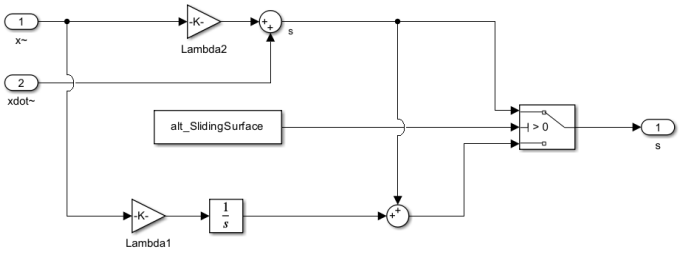


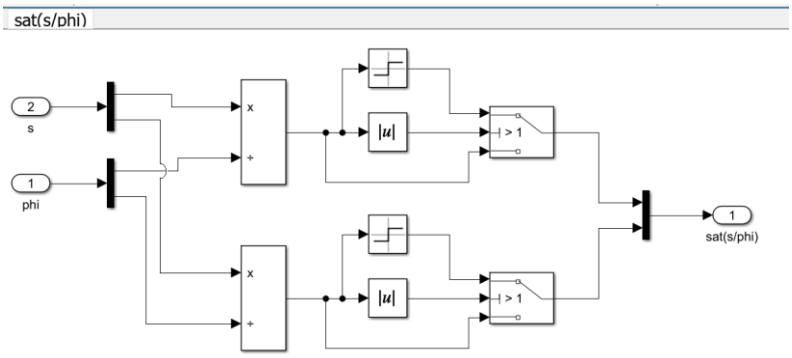
Desired Tracking



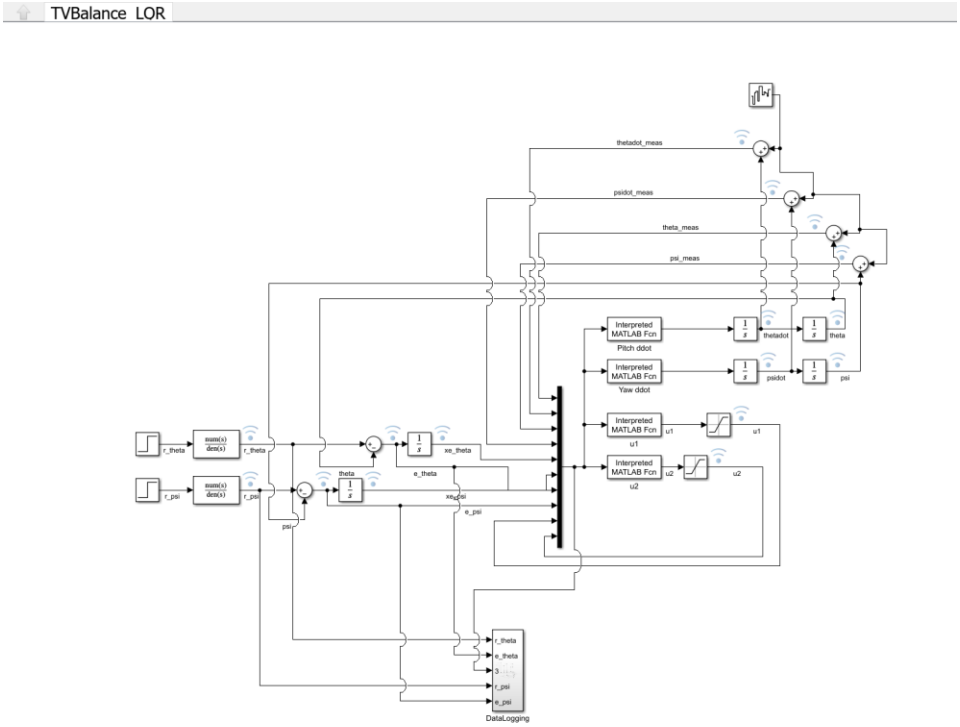


Sliding Surface, s





- **TV Balance LQR**



<u>Ardiuno Code</u>

```
/*
 * Model-Free Sliding Mode Control of a DC Motor Balancing System
 * 5V dc motor with a Servo for thrust vectoring
 * MISO System
 * Best Final RSSQ(e) Score =====> Pitch: 65.01 | Yaw: 22.88
 */

#include <BasicLinearAlgebra.h>
#include <ElementStorage.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
```

```
#include <utility/imumaths.h>
#include <SPI.h>
#include <SD.h>
#include <Servo.h>

File TestFile;
Servo servo1;
Servo servo2;

#define in1_pin 5
#define in2_pin 6
#define EEP 2 //Initialization pin

uint16_t BNO055_SAMPLERATE_DELAY_MS = 5; //how often to read data from the board, milli
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);

double servoOffset[2] = {135, 45};

double dt = BNO055_SAMPLERATE_DELAY_MS, t_prev = 0, t = 0;
int tfinal = 10;

int n = 0, ntotal = 200;
int j = 0;

void setup() {
  Serial.begin(115200);
  /*
  // Open serial communications and wait for port to open:
  while (!Serial) {
  ; // wait for serial port to connecNeeded for native USB port only
  }
  Serial.print("Initializing SD card...");
  if (!SD.begin(4)) {
  Serial.println("initialization failed!");
  while (1);
  }
  Serial.println("initialization done.");
  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  TestFile = SD.open("TestFile.txt", FILE_WRITE);
  */
  pinMode(EEP,1);
  pinMode(in1_pin,1);
  pinMode(in2_pin,1);
  servo1.attach(7);
  servo2.attach(8);
```

```
  servo1.write(servoOffset[0]);
  servo2.write(servoOffset[1]);

  Wire.begin();
   if (!bno.begin())
  {
   Serial.print("No BNO055 detected");
   while (1);
  }

}

void loop(){
   Serial.println("START");
  ////SUGGESTED TUNING PARAMETERS////
   double a_T = 0.1;
   double a_u[2] = {0.21,0.05}; //Lowpass Filter Constant for prop/servo control input (0,1): Higher
=> more aggressive control but prone to oscillations
   double uCP_max = 255, uCP_min = 100; // Max and min PWM Signal
   double uMP_max = 155, uMP_min = 0; // Max and min control law output
   double uCS1_max = 95, uCS1_min = 135; // Max and min Servo1 Angle
   double uMS1_max = 10, uMS1_min = -10; // Max and min control law output
   double uCS2_max = 85, uCS2_min = 45; // Max and min Servo2 Angle
   double uMS2_max = 10, uMS2_min = -10; // Max and min control law output
   double a_r[2] = {0.005,0.005};
   double lambda[2] = {1,.5}; //Slope of Sliding Surface - prop, servo: s = lambda*e + edot

   double uTail = 0, uTail_prev = 0;
   double uControl[3] = {uCP_min,uCS1_min,uCS2_min}, uMFSMC[3], uMFSMC_prev[3] = {0,0,0},
uMFSMC_pprev[3] = {0,0,0};
   double uC_max[3] = {uCP_max,uCS1_max,uCS2_max}, uC_min[3] =
{uCP_min,uCS1_min,uCS2_min}; //Max and Min Actuator Limits {PWM,DEG}
   double uM_max[3] = {uMP_max,uMS1_max,uMS2_max}, uM_min[3] =
{uMP_min,uMS1_min,uMS2_min}; //Max and Min Control Law Outputs {Prop,Servo}
   double a_g = .005, a_e = 0.1;
   double xDPS_prev[2] = {0,0}, xD_prev[2] = {0,0};
   double xD[2], xDPS[2], xDPSS[2], xDPSS_prev[2] = {0,0};
   double xD_ref[2] = {0,0}, xD_refp[2] = {0,0};
   double xDPS_ref[2] = {0,0}, xDPSS_ref[2] = {0,0};
   double xDPS_refp[2] = {0,0}, xDPSS_refp[2] = {0,0};
   double xD_offset[2], xD_offset_total[2], xD_offset_total_prev[2] = {0,0};
   double xD_offset_avg[2];

   double e[2], xe[2], edot[2], eddot[2];
   double e_prev[2] = {0,0}, xe_prev[2] = {0,0}, edot_prev[2] = {0,0}, eddot_prev[2] = {0,0};
```

```
  double e2total_prev[2] = {0,0}, e2total[2];
  double a_s = 0.05, s_phi[2], sat[2];
  double sigma[2] = {0.025,0.025}; //Noise Mean - 0.0044
  double eta[2], eta0[2] = {0.1,0.1};
  double su[2] = {0.2,0.2}, phi0[2] = {eta0[0]/lambda[0],eta0[1]/lambda[1]}, phi_prev[2] =
{phi0[0],phi0[1]};
  double s[2], s_prev[2] = {0,0}, sdot[2], phi[23], phidot[2], K[2];
  double bup[2] = {5,5}, blow[2] = {1,1}, beta0[2] = {sqrt(bup[0]/blow[0]),sqrt(bup[1]/blow[1])};
  double bhat0[2] = {sqrt(bup[0]*blow[0]),sqrt(bup[1]*blow[1])};

  for (j = 0; j <= 1; j += 1) {
    eta[j] = eta0[j] + (sigma[j]/0.1)*0.5*lambda[j];
  }
  //Estimator
  double lambdaFF = 0.9;
  double P_tilda[2], P[2], P_prev[2] = {2};
  double bhat_prev[2] = {bhat0[0]}, bhat[2] = {bhat0[0], bhat0[1]}, e1[2];
  double a_b = 0.1;

 digitalWrite(EEP,1);
 analogWrite(in1_pin, 0);
 analogWrite(in2_pin,0);

 delay(2000);
 Serial.println("Begin Calibration");

  for (n = 0; n <= ntotal; n +=1){
   //BNO055
   imu::Vector<3> angRate = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
   imu::Vector<3> angPos = bno.getVector(Adafruit_BNO055::VECTOR_EULER);

   xD_offset[0] = angPos.x();
     if (xD_offset[0] > 180){
       xD_offset[0] = xD_offset[0] - 360;
     }
   xD_offset[1] = -angPos.z();

   for (j = 0; j <= 1; j += 1) {
     xD_offset_total[j] = xD_offset[j] + xD_offset_total_prev[j];
     xD_offset_total_prev[j] = xD_offset_total[j];
   }
   delay(dt);
   }
   for (j = 0; j <= 1; j += 1) {
     xD_offset_avg[j] = xD_offset_total[j] / ntotal;
   }
```

```
/*  Serial.print(xD_offset_avg[0]);
  Serial.print(" ");
  Serial.println(xD_offset_avg[1]);*/
  Serial.println("Calibration Complete... Begin Stabilization");

for (t = 1; t <= tfinal*1000; t += dt){

  if (t/1000 < 1) {
    for (j = 0; j <= 1; j += 1) {
      xD_ref[j] = 0;
      xDPS_ref[j] = 0;
      xDPSS_ref[j] = 0;
    }
  }
  if ((t/1000) > 1 && (t/1000) < 7.5) {
    for (j = 0; j <= 1; j += 1) {
      // STEP INPUT
      xD_ref[0] = 0;
      // PRE-FILTER
      xD_ref[0] = a_r[0]*xD_ref[0] + (1-a_r[0])*xD_refp[0];
      // Velo Derivative and Low-Pass Filter
      xDPS_ref[0] = (xD_ref[0] - xD_refp[0]) / (dt/1000);
      xDPS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPS_refp[0];
      // Acc Derivative and Low-Pass Filter
      xDPSS_ref[0] = (xDPS_ref[0] - xDPS_refp[0]) / (dt/1000);
      xDPSS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPSS_refp[0];
    }
  }
  if ((t/1000) > 7.5 && (t/1000) < tfinal) {
    // STEP INPUT
    xD_ref[0] = 0;
    xD_ref[1] = 0;
    // PRE-FILTER
    for (j = 0; j <= 1; j += 1) {
      xD_ref[j] = a_r[j]*xD_ref[j] + (1-a_r[j])*xD_refp[j];
      // Velo Derivative and Low-Pass Filter
      xDPS_ref[j] = (xD_ref[j] - xD_refp[j]) / (dt/1000);
      xDPS_ref[j] = a_r[j]*xDPS_ref[j] + (1-a_r[j])*xDPS_refp[j];
      // Acc Derivative and Low-Pass Filter
      xDPSS_ref[j] = (xDPS_ref[j] - xDPS_refp[j]) / (dt/1000);
      xDPSS_ref[j] = a_r[j]*xDPS_ref[j] + (1-a_r[j])*xDPSS_refp[j];
    }
  }

  //BNO055
```

```
   imu::Vector<3> angRate = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
   imu::Vector<3> angPos = bno.getVector(Adafruit_BNO055::VECTOR_EULER);

   xD[0] = angPos.x() - xD_offset_avg[0];
    if (xD[0] > 180){
      xD[0] = xD[0] - 360;
    }
   xDPS[0] = -angRate.z();
   xD[1] = -angPos.z() - xD_offset_avg[1];
   xDPS[1] = angRate.x();
   for (j = 0; j <= 1; j += 1) {
    xDPS[j] = a_g*xDPS[j] + (1-a_g)*xDPS_prev[j];
    xDPSS[j] = (xDPS[j] - xDPS_prev[j]) / (dt/1000);
    xDPSS[j] = a_g*xDPSS[j] + (1-a_g)*xDPSS_prev[j];
   }

// ERROR FCN //
   for (j = 0; j <= 1; j += 1) {
      e[j] = xD[j] - xD_ref[j];
      xe[j] = e[j]*(dt/1000) + xe_prev[j];
      edot[j] = xDPS[j] - xDPS_ref[j];
      // Low Pass Filter
      edot[j] = edot[j]*a_e + (1-a_e)*edot_prev[j];
      eddot[j] = xDPSS[j] - xDPSS_ref[j];
      eddot[j] = eddot[j]*a_e + (1-a_e)*eddot_prev[j];
   }
/////////////////
///// MFSMC /////
   for (j = 0; j <= 1; j += 1) {
    //Switching Gain
    K[j] = abs(edot[j])*abs(beta0[j]-1) + lambda[j]*abs(e[j])*abs(beta0[j]-1) +
abs(bhat[j]*su[j]*(uMFSMC_pprev[j]-uMFSMC_prev[j])) + eta[j]*beta0[j];

    //Boundary Layer
    phidot[j] =  - lambda[j]*phi_prev[j] + K[j];
    phi[j] = phidot[j]*(dt/1000) + phi_prev[j];
    //Sliding Surface
    s[j] = (lambda[j]*e[j] + edot[j]);
    sdot[j] = (lambda[j]*edot[j] + eddot[j]);

    //SAT FUNCTION
    s_phi[j] = abs(s[j]/phi[j]);
    if (s_phi[j] > 1) {
     if (s[j]/phi[j] > 0) sat[j] = 1;
     if (s[j]/phi[j] < 0) sat[j] = -1;
    }
```

```
    if (s_phi[j] <= 1) {
      sat[j] = s[j]/phi[j];
    }

     uMFSMC[j] = (1/bhat[j])*(-sdot[j] - (K[j]-phidot[j])*sat[j]) + (2*uMFSMC_prev[j] -
uMFSMC_pprev[j]);
  }

  uMFSMC[0] = abs(a_u[0]*uMFSMC[0] + (1-a_u[0])*uMFSMC_prev[0]);
  uMFSMC[1] = a_u[1]*uMFSMC[1] + (1-a_u[1])*uMFSMC_prev[1];

  for (j = 0; j <= 1; j += 1) {
    if (uMFSMC[j] > uM_max[j]) {
      uMFSMC[j] = uM_max[j];
    }
    if (uMFSMC[j] < uM_min[j]) {
      uMFSMC[j] = uM_min[j];
    }
  }

  uControl[0] = map(uMFSMC[0],uM_min[0],uM_max[0],uC_min[0],uC_max[0]);

  if (xD[0] > 0) {
    uTail = (1/bhat[0])*(-sdot[0] - (K[0]-phidot[0])*sat[0]) + (2*uMFSMC_prev[0] -
uMFSMC_pprev[0]);
    uTail = a_T*uTail + (1-a_T)*uTail_prev;
    uControl[1] = map(uTail,0,10,uC_min[1],uC_max[1]);// -
map(uMFSMC[1],uM_min[1],uM_max[1],uC_min[1],uC_max[1]);
    uControl[2] = map(uTail,0,10,uC_min[2],uC_max[2]);// +
map(uMFSMC[1],uM_min[1],uM_max[1],uC_min[1],uC_max[1]);
  }
  else {
    uControl[1] = uC_min[1];// - map(uMFSMC[1],uM_min[1],uM_max[1],uC_min[1],uC_max[1]);
    uControl[2] = uC_min[2];// + map(uMFSMC[1],uM_min[1],uM_max[1],uC_min[1],uC_max[1]);
  }

  if (uControl[1] > uC_min[1]) {
    uControl[1] = uC_min[1];
  }
  if (uControl[1] < uC_max[1]) {
    uControl[1] = uC_max[1];
  }
  if (uControl[2] > uC_max[2]) {
    uControl[2] = uC_max[2];
  }
  if (uControl[2] < uC_min[2]) {
```

```
      uControl[2] = uC_min[2];
    }
////Estimator////
/*
    for (j = 0; j <= 1; j += 1) {
      P_tilda[j] = (P_prev[j]*uMFSMC[j]*uMFSMC[j]*P_prev[j])/(1 +
lambdaFF*uMFSMC[j]*P_prev[j]*uMFSMC[j]);
      P[j] = (1/lambdaFF)*(P_prev[j] - P_tilda[j]);
      if (abs(s[j]) >= phi[j]){
        e1[j] = xDPS[j] - bhat_prev[j]*uMFSMC[j];
      }
      else {
        e1[j] = 0;
      }
      bhat[j] = bhat_prev[j] + P_tilda[j]*uMFSMC[j]*e1[j];
      bhat[j] = a_b*bhat[j] + (1-a_b)*bhat_prev[j];
      if (bhat[j] < 0.1) {
        bhat[j] = 0.1;
      }
      if (bhat[j] > 5) {
        bhat[j] = 5;
      }
    }*/
/////////////////
/////////////////

    //analogWrite(in1_pin, uControl[0]);
    analogWrite(in1_pin, 0);
    analogWrite(in2_pin, 0);
    //servo1.write(uControl[1]);
    //servo2.write(uControl[2]);
    servo1.write(135);
    servo2.write(45);

    for (j = 0; j <= 1; j += 1) {
      if (t/1000 > 2) {
        e2total[j] = e[j]*e[j] + e2total_prev[j];
      }
      else {
        e2total[j] = 0;
      }
    }
/*
    Serial.print(t/1000);
    Serial.print(" | ");
    Serial.print(xD_ref[0] + xD_offset_avg[0]);
```

```
Serial.print(' ');
Serial.print(xD[0]);
Serial.print(",");
Serial.print(xD_ref[1] + xD_offset_avg[1]);
Serial.print(' ');*/
Serial.print(xD[1]);
Serial.print(" | ");/*
Serial.print(xDPS_ref[0]);
Serial.print(' ');
Serial.print(xDPS[0]);
Serial.print(' ');
Serial.print(xDPS_ref[1]);
Serial.print(' ');
Serial.println(xDPS[1]);
Serial.print(" | ");
Serial.print(xDPSS_ref[0]);
Serial.print(' ');
Serial.print(xDPSS[0]);
Serial.print(' ');
Serial.print(xDPSS_ref[1]);
Serial.print(' ');
Serial.print(xDPSS[1]);
Serial.print(' ');
Serial.print(uMFSMC[0]);
Serial.print(' ');
Serial.print(uControl[0]);
Serial.print(" | ");*/
Serial.print(uMFSMC[1]);
Serial.print(" | ");/*
Serial.print(uTail);
Serial.print(' ');
Serial.print(uControl[1]);
Serial.print(' ');
Serial.println(uControl[2]);
Serial.print(" | ");
Serial.print(xe[0]);
Serial.print(' ');
Serial.print(e[0]);
Serial.print(' ');
Serial.print(edot[0]);
Serial.print(' ');
Serial.print(eddot[0]);
Serial.print(' ');
Serial.print(xe[1]);
Serial.print(' ');
Serial.print(e[1]);
```

```
Serial.print(' ');
Serial.print(edot[1]);
Serial.print(' ');
Serial.print(eddot[1]);
Serial.print(" | ");
Serial.print(bhat[0]);
Serial.print(' ');
Serial.print(bhat[1]);
Serial.print(' ');
Serial.print(sdot[0]);
Serial.print(' ');*/
Serial.print(sdot[1]);
Serial.print(' ');/*
Serial.print(s[0]);
Serial.print(' ');
Serial.print(phi[0]);
Serial.print(' ');
Serial.print(-phi[0]);
Serial.print(' ');*/
Serial.print(s[1]);
Serial.print(' ');
Serial.print(phi[1]);
Serial.print(' ');/*
Serial.print(-phi[1]);
Serial.print(' ');
Serial.print(K[0]);
Serial.print(' ');*/
Serial.println(K[1]);/*
Serial.print(' ');
Serial.print(sqrt(e2total[0]));
Serial.print(' ');
Serial.print(sqrt(e2total[1]));
Serial.print(' ');
Serial.print(e1[0]);
Serial.print(' ');
Serial.print(P[0]);
Serial.print(' ');
Serial.print(e1[1]);
Serial.print(' ');
Serial.println(P[1]);*/

for (j = 0; j <= 1; j += 1) {
  e2total_prev[j] = e2total[j];
  xDPSS_prev[j] = xDPSS[j];
  xDPSS_refp[j] = xDPSS_ref[j];
  xDPS_refp[j] = xDPS_ref[j];
```

```
    xD_refp[j] = xD_ref[j];
    xDPS_prev[j] = xDPS[j];
    xD_prev[j] = xD[j];
    e_prev[j] = e[j];
    xe_prev[j] = xe[j];
    edot_prev[j] = edot[j];
    eddot_prev[j] = eddot[j];
    s_prev[j] = s[j];
    phi_prev[j] = phi[j];
    bhat_prev[j] = bhat[j];
    P_prev[j] = P[j];
    uMFSMC_pprev[j] = uMFSMC_prev[j];
    uMFSMC_prev[j] = uMFSMC[j];
  }
  uTail_prev = uTail;
  delay(dt);
}/*
  if (xD[0] > 30) {
    analogWrite(in1_pin,0);
    analogWrite(in2_pin,0);
    servo1.write(servoOffset[0]);
    servo2.write(servoOffset[1]);
    Serial.println("Kill-Switch Engaged");
    //TestFile.println("Kill-Switch Engaged");
    //TestFile.close();
    delay(dt);
    exit(0);
  }
  if (xD[0] < -30) {
    analogWrite(in1_pin,0);
    analogWrite(in2_pin,0);
    servo1.write(servoOffset[0]);
    servo2.write(servoOffset[1]);
    Serial.println("Kill-Switch Engaged");
    //TestFile.println("Kill-Switch Engaged");
    delay(dt);
    exit(0);
    //TestFile.close();
  }*/
  if (t > tfinal*1000) {
    Serial.print("Test Complete... Final Score: ");
    Serial.print(sqrt(e2total[0]));
    Serial.print(' ');
    Serial.print(sqrt(e2total[1]));
    //TestFile.print("Test Complete... Final Score: ");
    //TestFile.print(sqrt(e2total[0]));
```

```
    //TestFile.print(' ');
    //TestFile.print(sqrt(e2total[1]));
    //TestFile.print(' ');
    //TestFile.print(sqrt(e2total[2]));
    //TestFile.close();
    delay(dt);
    analogWrite(in1_pin, 0);
    analogWrite(in2_pin, 0);
    servo1.write(servoOffset[0]);
    servo2.write(servoOffset[1]);
    exit(0);
 }
}


//////////////////////

/*
 * LQR Control of a DC Motor Balancing System
 * 5V dc motor with a Servo for thrust vectoring
 * MISO System
 * Best Final RSSQ(e) Score =====> Pitch: 65.01 | Yaw: 22.88
 */

#include <BasicLinearAlgebra.h>
#include <ElementStorage.h>
#include <Servo.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

Servo servo;
const int MotorPinA = 4;
const int MotorSpeedPinA = 5;

const int CCW = HIGH;
const int CW = LOW;
uint16_t BNO055_SAMPLERATE_DELAY_MS = 5; //how often to read data from the board, milli
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);

////SUGGESTED TUNING PARAMETERS////
double uCP_max = 235, uCP_min = 125; // Max and min PWM Signal
double uMP_max = 50, uMP_min = 0; // Max and min control law output
double uCS_max = 45, uCS_min = -45; // Max and Min Servo angle
double uMS_max = 80, uMS_min = -80; // Max and min control law output
double a_r[2] = {0.005, 0.001};
```

```
double ServoOffset = 90;
double uControl[2], uLQR[2];
double uC_max[2] = {uCP_max,uCS_max}, uC_min[2] = {uCP_min,uCS_min}; //Max and Min Actuator Limits {PWM,DEG}
double uM_max[2] = {uMP_max,uMS_max}, uM_min[2] = {uMP_min,uMS_min}; //Max and Min Control Law Outputs {Prop,Servo}

double a_g = .005, a_e = 0.1;
double xDPS_prev[2] = {0,0}, xD_prev[2] = {0,0};
double xD[2], xDPS[2], xDPSS[2], xDPSS_prev[2] = {0,0};
double xD_ref[2] = {0,0}, xD_refp[2] = {0,0};
double xDPS_ref[2] = {0,0}, xDPSS_ref[2] = {0,0};
double xDPS_refp[2] = {0,0}, xDPSS_refp[2] = {0,0};
double xD_offset[2], xD_offset_total[2], xD_offset_total_prev[2] = {0,0};
double xDPS_offset[2], xDPS_offset_total[2], xDPS_offset_total_prev[2] = {0,0};
double xD_offset_avg[2], xDPS_offset_avg[2];

double e[2], xe[2], edot[2], eddot[2];
double e_prev[2] = {0,0}, xe_prev[2] = {0,0}, edot_prev[2] = {0,0}, eddot_prev[2] = {0,0};
double dt = BNO055_SAMPLERATE_DELAY_MS, t_prev = 0, t = 0;
const int tfinal = 25;

double e2total_prev[2] = {0,0}, e2total[2];

int n = 0, ntotal = 200;
int j = 0;

void setup() {
  Serial.begin(115200);

  pinMode(MotorPinA, OUTPUT);
  pinMode(MotorSpeedPinA, OUTPUT);

  servo.attach(8);
  servo.write(ServoOffset);
  Wire.begin();
    if (!bno.begin())
  {
    Serial.print("No BNO055 detected");
    while (1);
  }
}

void loop(){
  digitalWrite(MotorPinA, CW);
```

```
analogWrite(MotorSpeedPinA, uCP_min);
delay(2000);
Serial.println("Begin Calibration");

 for (n = 0; n <= ntotal; n +=1){
  //BNO055
  imu::Vector<3> angRate = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
  imu::Vector<3> angPos = bno.getVector(Adafruit_BNO055::VECTOR_EULER);

  xD_offset[0] = angPos.z();
  xDPS_offset[0] = -angRate.x();
  xD_offset[1] = angPos.x();
    if (xD_offset[1] > 180){
      xD_offset[1] = xD_offset[1] - 360;
    }
  xDPS_offset[1] = -angRate.z();
  xD_offset_total[0] = xD_offset[0] + xD_offset_total_prev[0];
  xDPS_offset_total[0] = xDPS_offset[0] + xDPS_offset_total_prev[0];
  xD_offset_total[1] = xD_offset[1] + xD_offset_total_prev[1];
  xDPS_offset_total[1] = xDPS_offset[1] + xDPS_offset_total_prev[1];
  xD_offset_total_prev[0] = xD_offset_total[0];
  xDPS_offset_total_prev[0] = xDPS_offset_total[0];
  xD_offset_total_prev[1] = xD_offset_total[1];
  xDPS_offset_total_prev[1] = xDPS_offset_total[1];
  delay(10);
  }
  xD_offset_avg[0] = xD_offset_total[0] / ntotal;
  xDPS_offset_avg[0] = xDPS_offset_total[0] / ntotal;
  xD_offset_avg[1] = xD_offset_total[1] / ntotal;
  xDPS_offset_avg[1] = xDPS_offset_total[1] / ntotal;

  Serial.print(xD_offset_avg[0]);
  Serial.print(" ");
  Serial.print(xDPS_offset_avg[0]);
  Serial.print(" ");
  Serial.print(xD_offset_avg[1]);
  Serial.print(" ");
  Serial.println(xDPS_offset_avg[1]);
  Serial.println("Calibration Complete... Begin Stabilization");

for (t = 1; t <= tfinal*1000; t += dt){
  digitalWrite(MotorPinA, CW);

  if (t/1000 < 1) {
    //Pitch
    xD_ref[0] = 0;
```

```
  xDPS_ref[0] = 0;
  xDPSS_ref[0] = 0;
  //Yaw
  xD_ref[1] = 0;
  xDPS_ref[1] = 0;
  xDPSS_ref[1] = 0;
 }
 if ((t/1000) > 1 && (t/1000) < 7.5) {
  // STEP INPUT
  xD_ref[0] = -xD_offset_avg[0];
  xD_ref[1] = 0;
  // PRE-FILTER
  xD_ref[0] = a_r[0]*xD_ref[0] + (1-a_r[0])*xD_refp[0];
  xD_ref[1] = a_r[1]*xD_ref[1] + (1-a_r[1])*xD_refp[1];
  // Velo Derivative and Low-Pass Filter
  xDPS_ref[0] = (xD_ref[0] - xD_refp[0]) / (dt/1000);
  xDPS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPS_refp[0];
  xDPS_ref[1] = (xD_ref[1] - xD_refp[1]) / (dt/1000);
  xDPS_ref[1] = a_r[1]*xDPS_ref[1] + (1-a_r[1])*xDPS_refp[1];
  // Acc Derivative and Low-Pass Filter
  xDPSS_ref[0] = (xDPS_ref[0] - xDPS_refp[0]) / (dt/1000);
  xDPSS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPSS_refp[0];
  xDPSS_ref[1] = (xDPS_ref[1] - xDPS_refp[1]) / (dt/1000);
  xDPSS_ref[1] = a_r[1]*xDPS_ref[1] + (1-a_r[1])*xDPSS_refp[1];
 }
 if ((t/1000) > 7.5 && (t/1000) < tfinal) {
  // STEP INPUT
  xD_ref[0] = -xD_offset_avg[0] + 7.5;
  xD_ref[1] = 0;
  // PRE-FILTER
  xD_ref[0] = a_r[0]*xD_ref[0] + (1-a_r[0])*xD_refp[0];
  xD_ref[1] = a_r[1]*xD_ref[1] + (1-a_r[1])*xD_refp[1];
  // Velo Derivative and Low-Pass Filter
  xDPS_ref[0] = (xD_ref[0] - xD_refp[0]) / (dt/1000);
  xDPS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPS_refp[0];
  xDPS_ref[1] = (xD_ref[1] - xD_refp[1]) / (dt/1000);
  xDPS_ref[1] = a_r[1]*xDPS_ref[1] + (1-a_r[1])*xDPS_refp[1];
  // Acc Derivative and Low-Pass Filter
  xDPSS_ref[0] = (xDPS_ref[0] - xDPS_refp[0]) / (dt/1000);
  xDPSS_ref[0] = a_r[0]*xDPS_ref[0] + (1-a_r[0])*xDPSS_refp[0];
  xDPSS_ref[1] = (xDPS_ref[1] - xDPS_refp[1]) / (dt/1000);
  xDPSS_ref[1] = a_r[1]*xDPS_ref[1] + (1-a_r[1])*xDPSS_refp[1];
 }

 //BNO055
 imu::Vector<3> angRate = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
```

```
    imu::Vector<3> angPos = bno.getVector(Adafruit_BNO055::VECTOR_EULER);

    xD[0] = angPos.z() - xD_offset_avg[0];
    xDPS[0] = -angRate.x();
    xDPS[0] = a_g*xDPS[0] + (1-a_g)*xDPS_prev[0];
    xDPSS[0] = (xDPS[0] - xDPS_prev[0]) / (dt/1000);
    xDPSS[0] = a_g*xDPSS[0] + (1-a_g)*xDPSS_prev[0];
    xD[1] = angPos.x();
      if (xD[1] > 180){
        xD[1] = xD[1] - 360 - xD_offset_avg[1];
      }
      else {
        xD[1] = angPos.x() - xD_offset_avg[1];
      }
    xDPS[1] = -angRate.z();
    xDPS[1] = a_g*xDPS[1] + (1-a_g)*xDPS_prev[1];
    xDPSS[1] = (xDPS[1] - xDPS_prev[1]) / (dt/1000);
    xDPSS[1] = a_g*xDPSS[1] + (1-a_g)*xDPSS_prev[1];

// ERROR FCN //
   for (j = 0; j <= 1; j += 1){
    e[j] = xD[j] - xD_ref[j];
    xe[j] = e[j]*(dt/1000) + xe_prev[j];
    edot[j] = xDPS[j] - xDPS_ref[j];
    // Low Pass Filter
    edot[j] = edot[j]*a_e + (1-a_e)*edot_prev[j];
    eddot[j] = xDPSS[j] - xDPSS_ref[j];
    eddot[j] = eddot[j]*a_e + (1-a_e)*eddot_prev[j];
   }
////////////////
///// LQR /////
                //{theta,      phi, thetadot, phidot,   xe_theta,  xe_phi,   e_theta,   e_phi}
    double K1[8] = {1.8031,   1.0737,       0,      0,    0.9591,       0,    0.116,       0};
    double K2[8] = {0     ,       0,    1.631, 0.9286,          0, -0.8899,         0, -0.1122};
    uLQR[0] = -(K1[0]*xD[0] + K1[1]*xDPS[0] + K1[2]*xD[1] + K1[3]*xDPS[1] + K1[4]*xe[0] +
K1[5]*xe[1] + K1[6]*e[0] + K1[7]*e[1]);
    uLQR[1] = -(K2[0]*xD[0] + K2[1]*xDPS[0] + K2[2]*xD[1] + K2[3]*xDPS[1] + K2[4]*xe[0] +
K2[5]*xe[1] + K2[6]*e[0] + K2[7]*e[1]);



    uControl[0] = map(uLQR[0],uMP_min,uMP_max,uCP_min,uCP_max);
    uControl[1] = uLQR[1];//map(uLQR[1],uMS_min,uMS_max,uCS_min,uCS_max);

    if (uControl[0] > uC_max[0]) {
      uControl[0] = uC_max[0];
```

```
    }
    if (uControl[0] < uC_min[0]) {
      uControl[0] = uC_min[0];
    }
    if (uControl[1] > uC_max[1]) {
      uControl[1] = uC_max[1];
    }
    if (uControl[1] < uC_min[1]) {
      uControl[1] = uC_min[1];
    }
////////////////
////////////////

    analogWrite(MotorSpeedPinA, uControl[0]);
    servo.write(ServoOffset + uControl[1]);

   for (j = 0; j <= 1; j += 1){
    if (t/1000 > 2) {
      e2total[j] = e[j]*e[j] + e2total_prev[j];
    }
    else {
      e2total[j] = 0;
    }
   }

    Serial.print(t/1000);
    Serial.print(' ');
    Serial.print(xD_ref[0] + xD_offset_avg[0]);
    Serial.print(' ');
    Serial.print(xD[0] + xD_offset_avg[0]);
    Serial.print(' ');
    Serial.print(xD_ref[1]);
    Serial.print(' ');
    Serial.print(xD[1]);
    Serial.print(' ');
    Serial.print(xDPS_ref[0]);
    Serial.print(' ');
    Serial.print(xDPS[0]);
    Serial.print(' ');
    Serial.print(xDPS_ref[1]);
    Serial.print(' ');
    Serial.print(xDPS[1]);
    Serial.print(' ');
    Serial.print(xDPSS_ref[0]);
    Serial.print(' ');
    Serial.print(xDPSS[0]);
```

```
Serial.print(' ');
Serial.print(xDPSS_ref[1]);
Serial.print(' ');
Serial.print(xDPSS[1]);
Serial.print(' ');
Serial.print(uLQR[0]);
Serial.print(' ');
Serial.print(uControl[0]);
Serial.print(' ');
Serial.print(uLQR[1]);
Serial.print(' ');
Serial.print(uControl[1]);
Serial.print(' ');
Serial.print(xe[0]);
Serial.print(' ');
Serial.print(xe[1]);
Serial.print(' ');
Serial.print(e[0]);
Serial.print(' ');
Serial.print(e[1]);
Serial.print(' ');
Serial.print(edot[0]);
Serial.print(' ');
Serial.print(edot[1]);
Serial.print(' ');
Serial.print(eddot[0]);
Serial.print(' ');
Serial.print(eddot[1]);
Serial.print(' ');
Serial.print(sqrt(e2total[0]));
Serial.print(' ');
Serial.println(sqrt(e2total[1]));


for (j = 0; j <= 1; j += 1) {
e2total_prev[j] = e2total[j];
xDPSS_prev[j] = xDPSS[j];
xDPSS_refp[j] = xDPSS_ref[j];
xDPS_refp[j] = xDPS_ref[j];
xD_refp[j] = xD_ref[j];
xDPS_prev[j] = xDPS[j];
xD_prev[j] = xD[j];
e_prev[j] = e[j];
xe_prev[j] = xe[j];
edot_prev[j] = edot[j];
eddot_prev[j] = eddot[j];
```

```
    }
    delay(dt);

    if (xD[0] > 45 || xD[1] > 25) {
      analogWrite(MotorSpeedPinA,0);
      Serial.println("Kill-Switch Engaged");
      delay(dt);
      exit(0);
    }
    if (xD[0] < -45 || xD[1] < -25) {
      analogWrite(MotorSpeedPinA,0);
      Serial.println("Kill-Switch Engaged");
      delay(dt);
      exit(0);
      }
  }
  Serial.print("Test Complete... Final Score: ");
  Serial.print(sqrt(e2total[0]));
  Serial.print(" ");
  Serial.println(sqrt(e2total[1]));

  delay(dt);
  analogWrite(MotorSpeedPinA, 0);
  servo.write(ServoOffset);
  exit(0);
}
```