MAE 6780

Cornell University

Stabilization of a Quadrotor Drone from Random Initial Conditions Using Integral Sliding Mode Control

Author:	*NetID:*
Mitchell Dominguez	md697
Group Members:	
Amlan Sinha	as2558
Patrick Voorhees	pwv9

Instructor:

Silvia Ferrari

11 May, 2018

Contents

Abstract

Quadrotor drones are incredibly versatile machines that have uses in everything from recreation to military applications. As such, the control of such vehicles is an incredibly relevant and important problem to solve. In this paper, integral sliding mode control is implemented to stabilize a quadrotor drone. Since the quadrotor is underactuated, with six degrees of freedom but only four actuators, the attitude and height of the drone can be directly controlled while the horizontal position is brought to a constant. The stability of the controller is proved through Lyapunov stability theory, and the controller is shown through simulation to achieve stability of the controlled states of the quadrotor, even under extreme initial conditions such as the drone being thrown into the air upside down. However, the presence of steady state errors in the controlled states results in unwanted motion in the horizontal plane, which will require further tuning of the controller and development of position control to eliminate.

1 Introduction and Theory

1.1 Overview

Since the normal operation of a quadcopter does not involve much deviation in the roll, pitch, and yaw angles from those in the hovering condition, it is often convenient to design controllers for quadcopters based on linearizations of the equations of motion about the hovering condition. However, the goal of stabilizing the quadcopter under any set of initial conditions necessarily means that the quadcopter will experience states far from the hovering condition. Thus designing a linear controller to achieve the robust stabilization of the quadcopter would involve linearizing about several states and implementing a form of gain scheduling, which would be quite complex and impractical.

In this paper, an integral sliding mode controller is developed to control the quadcopter using the full nonlinear equations of motion of the vehicle. This approach avoids the problems that arise from dealing with linearized systems, namely that any linearization of a nonlinear system is only useful within certain bounds. Outside these bounds, the linearized model would cease to be accurate and any control law applied in the situation would be useless. A nonlinear control law such as sliding mode control does not suffer from these same issues, and thus is ideal for applications such as this.

1.2 Nonlinear Dynamics of the Quadrotor

The position and attitude of a quadcopter in inertial space is described by $[x \quad y \quad z \quad \phi \quad \theta \quad \psi]^T$, where x,y, and z correspond to the Cartesian position of the vehicle, and ϕ, θ, and ψ are the roll, pitch, and yaw angles that correspond to a 1-2-3 Euler angle set.

The dynamics of the quadrotor were initially derived relative to the body-frame rates $[u \quad v \quad w \quad p \quad q \quad r]^T$ corresponding to velocities in the body x,y, and z directions and angular velocities about the body x,y, and z directions respectively. These equations can be seen in Equation 1, where I_x, I_y, and I_z correspond to the moments of inertia of the quadcopter about its body axes, J_R is the rotor moment of inertia, g is the acceleration due to gravity, L is the length of each arm of the quadcopter, $\Omega = -\Omega_1 + \Omega_2 - \Omega_3 + \Omega_4$ is a combination of the rotor speeds Ω_i, and U_i correspond to the control inputs which are defined in the next section.

$$
\begin{aligned}
\dot{u} &= (vr - wq) + g\sin\theta \\
\dot{v} &= (wp - ur) - g\sin\phi\cos\theta \\
\dot{w} &= (uq - vp) - g\cos\phi\cos\theta + \frac{U1}{m} \\
\dot{p} &= \left(\frac{I_y - I_z}{I_x}\right) qr - \frac{J_R}{I_x} q\Omega + \frac{L}{I_x} U_2 \\
\dot{q} &= \left(\frac{I_z - I_x}{I_y}\right) pr + \frac{J_R}{I_y} p\Omega + \frac{L}{I_y} U_3 \\
\dot{r} &= \left(\frac{I_x - I_y}{I_z}\right) pq + \frac{1}{I_z} U_4
\end{aligned}
\tag{1}
$$

While the body-frame equations are sufficient to bring all of the quadrotor's positional and angular rates to zero, they are however insufficient to stabilize the inertial positions and angles. Thus, a conversion to inertial equations of motion is necessary. This was done by making the assumption that the angular rates are small, resulting in $p \approx \dot{\phi}$, $q \approx \dot{\theta}$, and $r \approx \dot{\psi}$. Applying this assumption, the inertial equations of motion in (2) were thus derived [1].

$$
\begin{aligned}
\ddot{x} &= \frac{U_1}{m}\left(\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi\right) \\
\ddot{y} &= \frac{U_1}{m}\left(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi\right) \\
\ddot{z} &= \frac{U_1}{m}\left(\cos\phi\cos\psi\right) - g \\
\ddot{\phi} &= \left(\frac{I_y - I_z}{I_x}\right)\dot{\theta}\dot{\psi} - \frac{J_R}{I_x}\dot{\theta}\Omega + \frac{L}{I_x} U_2 \\
\ddot{\theta} &= \left(\frac{I_z - I_x}{I_y}\right)\dot{\phi}\dot{\psi} + \frac{J_R}{I_y}\dot{\phi}\Omega + \frac{L}{I_y} U_3 \\
\ddot{\psi} &= \left(\frac{I_x - I_y}{I_z}\right)\dot{\theta}\dot{\phi} + \frac{1}{I_z} U_4
\end{aligned}
\tag{2}
$$

The system was then integrated by using the following state:

$$
\boldsymbol{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & z & \dot{z} & \phi & \dot{\phi} & \theta & \dot{\theta} & \psi & \dot{\psi} \end{bmatrix}^T \tag{3}
$$

1.3 Definition of Control Inputs

In the physical implementation of a quadcopter, voltage is provided to the motors to change the rotor speeds. The relation between voltage and rotor speed is shown in Equation 4.

$$\boldsymbol{\Omega} = \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \\ \Omega_4 \end{bmatrix} = \begin{bmatrix} a_1 & 0 & 0 & 0 \\ 0 & a_2 & 0 & 0 \\ 0 & 0 & a_3 & 0 \\ 0 & 0 & 0 & a_4 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = A_V \boldsymbol{V} + \boldsymbol{b} \tag{4}$$

To relate the control inputs to rotor speed, however, a different relation is necessary. Equation 5 relates the control inputs U_i to the rotor speeds, where b is the coefficient of thrust and d is the coefficient of drag [2]. U_1 corresponds to thrust, and U_2, U_3, and U_4 correspond to the roll, pitch, and yaw moments respectively.

$$\boldsymbol{U} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \\ b(\Omega_4^2 - \Omega_2^2) \\ b(\Omega_3^2 - \Omega_1^2) \\ d(\Omega_4^2 + \Omega_2^2 - \Omega_3^2 - \Omega_1^2) \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & -b & 0 & b \\ -b & 0 & b & 0 \\ -d & d & -d & d \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} = A_U \boldsymbol{\Omega^2} \tag{5}$$

One of the major constraints on the system is that the actuators are not always able to provide the rotation rates that are called for by the controllers. This is because the quadrotor is a physical system where there are physical limits on the voltage that can be supplied to the motors and the speed with which the rotors can spin. Thus, as will be described later in the procedure section, the Simulink model for the controlled quadrotor has saturation blocks to simulate the physical saturation of the actuators on an actual drone.

Additionally, it is evident that there are only four control inputs, whereas the equations of motion dictate that there are six degrees of freedom in this system. Thus, this is an underactuated system. Four out of the six degrees of freedom can be chosen to be controlled, and the other two can only be stabilized. Since the goal of this project was to stabilize the attitude and height of the quadrotor, the roll, pitch, yaw, and height of the quadrotor were chosen as the controlled states. The result of this choice of controlled variables was of course that there was no explicit control of the position of the quadrotor in the horizontal plane.

2 Procedure

2.1 Derivation of the Control Law

A system under sliding mode control evolves in two phases: the reaching mode and the sliding mode. This is illustrated in Figure 1. The reaching mode involves forcing the system onto a sliding surface which is defined such that the system is asymptotically stable when on the surface. Once the system arrives on the sliding surface, it then needs to stay there for all time. This is the sliding mode [3].

Following from these characteristics of sliding mode control, it follows that there are two main phases to designing a sliding mode controller. The first involves finding a manifold of asymptotic stability in the error space of the system. Such a manifold ensures that the error goes to zero. Once the sliding surface is defined, a control law that both drives the system to the sliding surface and then keeps it there must be designed[3].

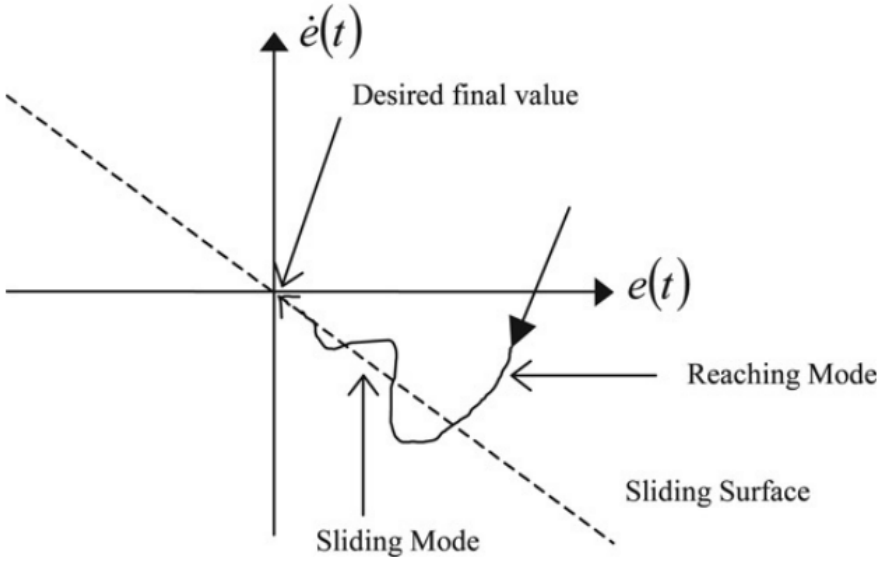

Figure 1: Visualization of sliding mode control [4].

For integral sliding mode control, the sliding surface and its derivative can be defined as [3]:

$$\begin{aligned} \sigma(\boldsymbol{x}(t)) &= k_p \boldsymbol{e}(t) + k_d \dot{\boldsymbol{e}}(t) + k_i \int_{t_0}^{t} \boldsymbol{e}(\tau) d\tau \\ \dot{\sigma}(\boldsymbol{x}(t)) &= k_p \dot{\boldsymbol{e}}(t) + k_d \ddot{\boldsymbol{e}}(t) + k_i \boldsymbol{e}(t) \end{aligned} \tag{6}$$

Lyapunov stability theory is then used to prove the asymptotic stability of the sliding surface and to derive the corresponding control law. A function $V(\boldsymbol{x}(t))$ is a Lyapunov function if it satisfies the following criteria:

$$\begin{aligned} V(\boldsymbol{x}(t)) &= 0 \ \ \forall \ \ \boldsymbol{x}(t) = 0 \\ V(\boldsymbol{x}(t)) &> 0 \ \ \forall \ \ \boldsymbol{x}(t) \neq 0 \\ \dot{V}(\boldsymbol{x}(t)) &\leq 0 \ \ \forall \ \ \boldsymbol{x}(t) \neq 0 \end{aligned} \tag{7}$$

A candidate Lyapunov function must be equal to zero when $\boldsymbol{x}(t) = 0$ and must be positive definite for all nonzero values of $\boldsymbol{x}(t)$. Additionally, the time derivative of the Lyapunov function must be negative semidefinite for all nonzero states. If there exists a Lyapunov function for the system, then it can be said that the system is asymptotically stable [5]. As the sliding surface must be asymptotically stable, a Lyapunov function for the system must be found.

A quadratic candidate Lyapunov function is often a good choice to ensure asymptotic stability [1]. As such, the candidate Lyapunov function is defined below:

$$\begin{aligned} V(\boldsymbol{x}(t)) &= \frac{1}{2}\sigma^T\sigma \\ \dot{V}(\boldsymbol{x}(t)) &= \sigma\dot{\sigma} \end{aligned} \tag{8}$$

The Lyapunov function candidate is positive semidefinite, so the first two Lyapunov stability criteria are achieved. Now the time derivative of the candidate Lyapunov function must be constrained to be negative semidefinite. It can be shown [3] [1] that if we set

$$\dot{\sigma}(\boldsymbol{x}(t)) = -k\,\mathrm{sign}(\sigma) - \eta\sigma \tag{9}$$

then $\dot{V}(\boldsymbol{x}(t)) = -\kappa\sigma\,\mathrm{sign}(\sigma) - \eta\sigma^2 \leq 0$. Thus, setting the time derivative of the sliding surface equal to the expression in Equation 9 yields a Lyapunov function for the sliding surface. Thus, applying Equation 9 to the definition of the sliding surface yields an asymptotically stable system in the error space of the system.

The presence of the sign function in Equation 9 makes this the "switching function" of the sliding mode controller. The function changes sign depending on the sign of the sliding variable σ to always drive the system in the direction of the sliding manifold when it is not already on it. However, the use of the sign function tends to introduce a phenomenon called chattering in the sliding mode controller [3]. This is demonstrated in Figure 1 by the oscillation of the system state around the sliding surface.

Chattering is an undesirable phenomenon because of the large control effort at all times necessary to combat it. Thus, in the implementation of sliding mode control, smoother alternatives to the sign function are often used [3]. In this project, the hyperbolic tangent function was selected as the alternative to the sign function to reduce the chattering phenomenon.

Applying the substitution in Equation 9 and replacing sign with tanh yields

$$k_p\dot{e}(t) + k_d\ddot{e}(t) + k_i e(t) = -\kappa \tanh(\sigma) - \eta\sigma \tag{10}$$

Substituting in the error in the state $\boldsymbol{e}(t) = \boldsymbol{x}_d(t) - \boldsymbol{x}(t)$ yields

$$k_p\left(\dot{\boldsymbol{x}}_d(t) - \dot{\boldsymbol{x}}(t)\right) + k_d\left(\ddot{x}_d(t) - \ddot{x}(t) + k_i\left(\boldsymbol{x}_d(t) - \boldsymbol{x}(t)\right)\right) = -\kappa \tanh(\sigma) - \eta\sigma \tag{11}$$

Substituting the equations of motion $\ddot{\boldsymbol{x}}$ from Equation 2, the control inputs can be derived to be the following:

$$\begin{aligned}
U_1 &= \frac{m}{\cos\phi\cos\theta}\left(g + \ddot{z}_d + \frac{k_{p_z}}{k_{d_z}}(\dot{z}_d - \dot{z}) + \frac{k_{i_z}}{k_{d_z}}(z_d - z) + \frac{\kappa_z}{k_{d_z}}\tanh(\sigma_z) + \frac{\eta_z}{k_{d_z}}\sigma_z\right) \\
U_2 &= \frac{I_{xx}}{L}\left(-\frac{I_{yy} - I_{zz}}{I_{xx}}\dot{\theta}\dot{\psi} + \frac{J_R}{I_{xx}}\dot{\theta}\Omega + \ddot{\phi}_D + \frac{k_{p_\phi}}{k_{d_\phi}}(\dot{\phi}_d - \dot{\phi}) + \frac{k_{i_\phi}}{k_{d_\phi}}(\phi_d - \phi) + \frac{\kappa_\phi}{k_{d_\phi}}\tanh(\sigma_\phi) + \frac{\eta_\phi}{k_{d_\phi}}\sigma_\phi\right) \\
U_3 &= \frac{I_{xx}}{L}\left(-\frac{I_{zz} - I_{xx}}{I_{yy}}\dot{\phi}\dot{\psi} + \frac{J_R}{I_{xx}}\dot{\phi}\Omega + \ddot{\theta}_D + \frac{k_{p_\theta}}{k_{d_\theta}}(\dot{\theta}_d - \dot{\theta}) + \frac{k_{i_\theta}}{k_{d_\theta}}(\theta_d - \phi) + \frac{\kappa_\theta}{k_{d_\theta}}\tanh(\sigma_\theta) + \frac{\eta_\theta}{k_{d_\theta}}\sigma_\theta\right) \\
U_4 &= I_{zz}\left(\frac{I_{xx} - I_{yy}}{I_{zz}}\dot{\phi}\dot{\theta} + \ddot{\psi}_D + \frac{k_{p_\psi}}{k_{d_\psi}}(\dot{\psi}_d - \dot{\psi}) + \frac{k_{i_\psi}}{k_{d_\psi}}(\psi_d - \psi) + \frac{\kappa_\psi}{k_{d_\psi}}\tanh(\sigma_\psi) + \frac{\eta_\psi}{k_{d_\psi}}\sigma_\psi\right)
\end{aligned} \tag{12}$$

All k, κ, and η values are tunable parameters that are used to optimize the performance of the controller through repeated simulations. Thus, there are 20 tunable parameters that can be adjusted to optimize the controller.

2.2 Simulation of the Quadrotor System

2.2.1 Top-Level Block Diagram

The controlled quadrotor system was simulated using Mathworks Simulink software. The top-level block diagram of the controller can be shown in the below figure:

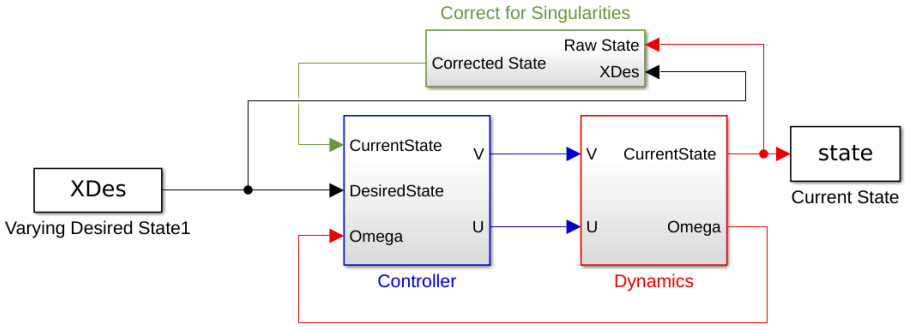

Figure 2: Top-level block diagram

The desired state, corrected current state, and the current Ω are all fed into the controller block, which then calculates corresponding voltages and control inputs. Those are then fed into the dynamics block, which outputs the updated state and Ω value. The current state goes through the correction block to account for singularities at $\phi = \frac{\pi}{2}$ and $\theta = \frac{\pi}{2}$. This is discussed in greater detail in Section 2.2.4.

2.2.2 Controller Block Diagram

The block diagram of the controller is shown below in Figure 3. As can be seen from the figure, the corrected state, desired state, and Ω value is fed into the `sliding surface generator` block, which generates the sliding surfaces as defined in Equation 6. Then, the desired state, corrected current state, their derivatives, the sliding surface values, and Ω are fed into the `calculateU` block.

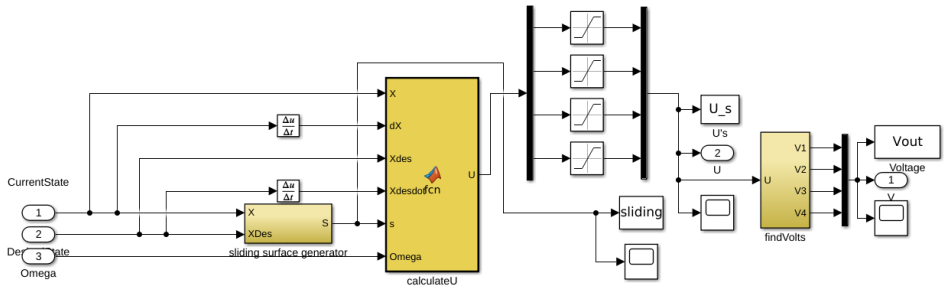

Figure 3: Block diagram of the controller

The `calculateU` block outputs the control inputs U_i by implementing Equations 12. Each U is then fed into a saturation function which imposes constraints on the system to behave physically. More specifically, U_1 is bounded to greater than or equal to zero and less than an upper limit,

as it is assumed that the rotors cannot output negative thrust and also have a maximum speed. If U_1 were negative, it would physically mean that the rotors start spinning in reverse, which is unrealistic. The other U values have their magnitudes bounded at reasonable lmimits to ensure that the simulated system cannot impart unrealistic moments.

The resulting control inputs are then outputted from the controller subsystem, and are also fed into the `findVolts` block. This block outputs the rotor voltages as calculated with Equation 4. These voltage values are also outputted from the controller subsystem.

2.2.3 Dynamics Block Diagram

The voltages and control inputs calculated in the controller subsystem are fed into the dynamics subsystem, which is shown in Figure 4. The `Motor Dynamics` subsystem implements first order dynamics for the motors to output Ω. The motor 1 dynamics are shown in Figure 5. While the majority of the dynamics subsystem was provided on Blackboard, the original blocks that calculated U were removed as the control inputs were fed into the dynamics by the controller.

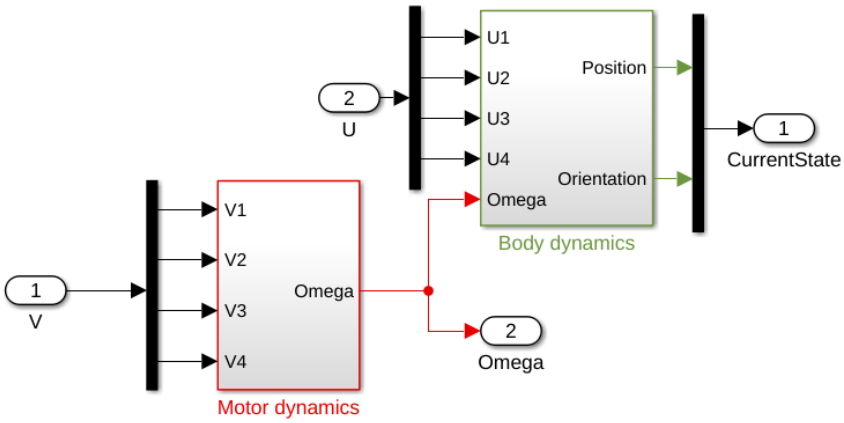

Figure 4: Block diagram of the dynamics

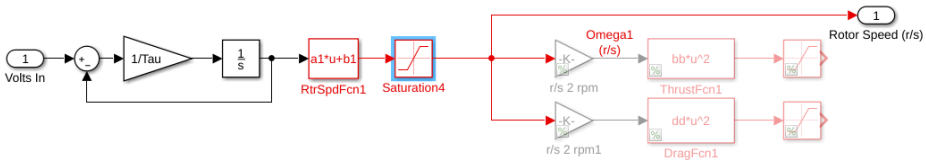

Figure 5: Block diagram of the motor dynamics

Once Ω is calculated in `Motor Dynamics`, the equations of motion as shown in Equations 2 are simulated, with Ω and the U_i values as inputs to the `Body Dynamics` block.

2.2.4 Singularity Rejection

As a result of the attitude of the quadrotor being represented with Euler angles, the controller suffers from issues with singularities. As can be seen in the U_1 equation in 12, the entire equation for the thrust control input is divided by $\cos\phi\cos\theta$. Thus, when either ϕ or θ approach $\frac{\pi}{2}$, the U_1 control input approaches infinity.

The occurrence of singularities is avoided in the `Correct for Singularities` subsystem, the contents of which are shown in Figure 6. The `Singularity in phi` and `Singularity in theta` blocks each take in the current (uncorrected) state and the desired state and they each output the state with a corrected angular term. If the ϕ or θ states are within an angular tolerance t of $\frac{\pi}{2}$, then the subsystem outputs the state with the angle in question $2t$ closer to the desired state. The corrected ϕ and θ angles are overlaid on the raw angles in Figure 7. It should be noted that this process only affects the state that is input into the controller subsystem; the data from the output of the dynamics is not altered.

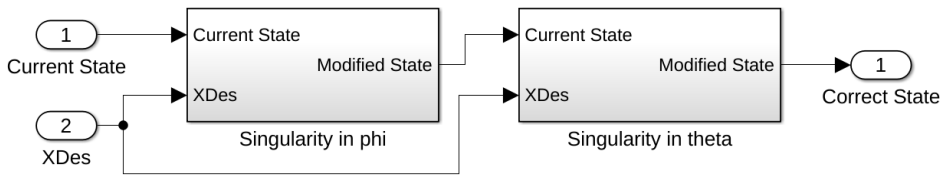

Figure 6: Block diagram of the singularity rejection subsystem.

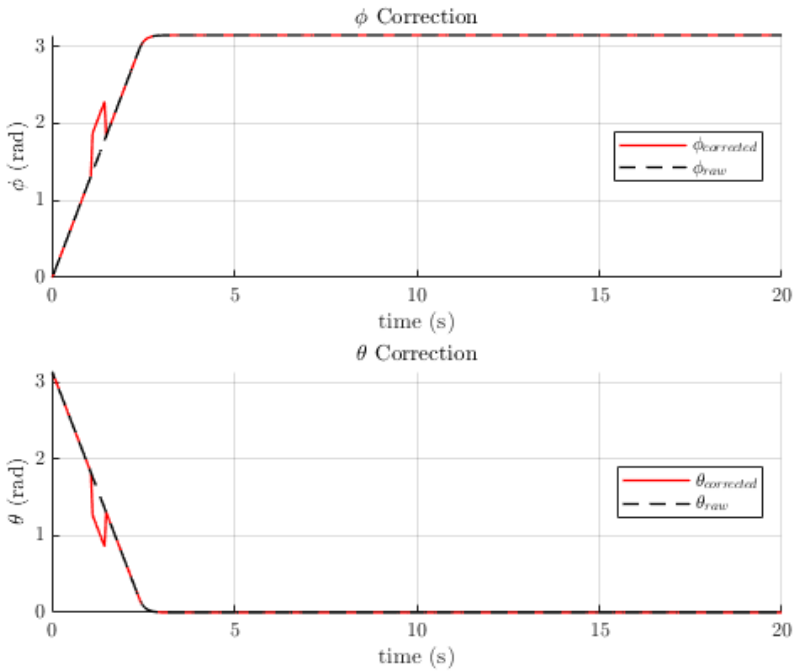

Figure 7: Plot of the corrected vs. raw ϕ and θ values

2.3 Tuning Parameters

In order to achieve the goal of stabilizing the quadcopter from random initial conditions, the tuning parameters of the controller had to be adjusted. The first test case that was used to tune the parameters was the "upside down throw into the air." The desired final state was $z_d = 12$, with all the other states equal to zero, and the initial conditions were as follows:

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 5 & \pi & 0 & 0 & 0 & \frac{\pi}{2} & \pi \end{bmatrix}^T \tag{13}$$

The main two parameters that were varied were the proportional, integral, and derivative terms. The proportional term was increased to reduce the response time of the system, the integral term was increased to reduce steady state error, and the derivative term was increased to reduce overshoot. These parameters were tuned until a response with minimal ringing and steady state error was achieved. Table 1 demonstrates the results of this tuning.

Table 1: Sliding Mode Control Tuning Parameters

	z	ϕ	θ	ψ
k_p	50	50	50	50
k_i	200	100	100	50
k_d	10	10	10	10
η	20	20	20	20
κ	5	5	5	5

3 Results

3.1 Upside-Down Toss

For the initial and desired conditions that were described in Section 2.3, the results are shown in Figure 8. As can be seen from the figure, the parameters chosen did not perfectly eliminate steady state error or overshoot. However, the rise time for all of the angles is well below one second, and steady state in the angles is reached by 2.5 seconds. The upwards step of 10 meters had a rise time of less than 2 seconds, and the system settles to the steady state height in under 5 seconds.

The results of this simulation show that the goal of this project was indeed met. The integral sliding mode controller was able to stabilize the quadrotor under the extreme initial condition of starting upside down.

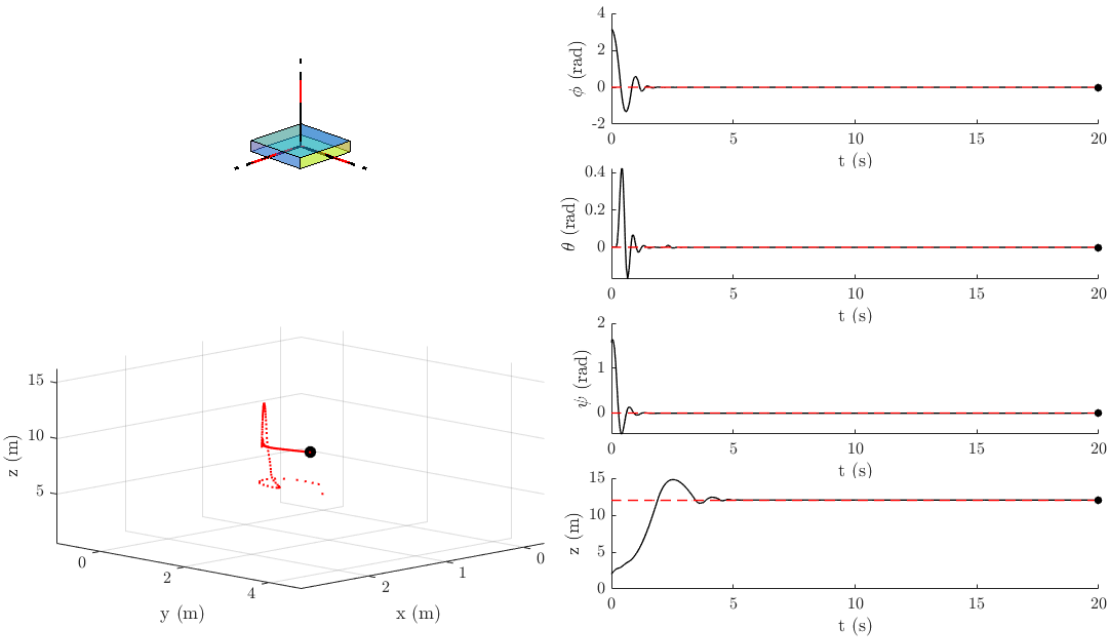

Figure 8: Results of the upside down throw

3.2 Extreme Angular Rates

To test the robustness of the controller, an incredibly large initial condition was used. The initial conditions and desired conditions for this result are the same as for the upside-down toss, except an initial pitch rate of 100 rad/s was used.

As can be seen, the controller does manage to bring the roll, pitch, yaw, and height states to constant values. However, there was significant steady state error. To combat this, all of the integral gains were doubled from the values stated in Section 2.3, and the results of this test can be seen in Figure 9. Despite the incredibly high initial pitch rate, the controller does manage to bring the quadrotor to a steady angular state within 6 seconds, and a steady height in under 12 seconds.

It is interesting to note that despite the quadrotor's nonzero steady state angles, it was able to bring itself to a steady-state height. This makes sense, since height is one of the controlled variables. However, one of the major shortcomings of this controller is immediately obvious. Since x and y position are not controlled, any steady state error in the angular states causes the quadcopter to fly at a steady rate in the horizontal plane. Thus, in applications where the horizontal position of the quadrotor is important, attitude control of the quadrotor is not necessarily the ideal method of stabilization.

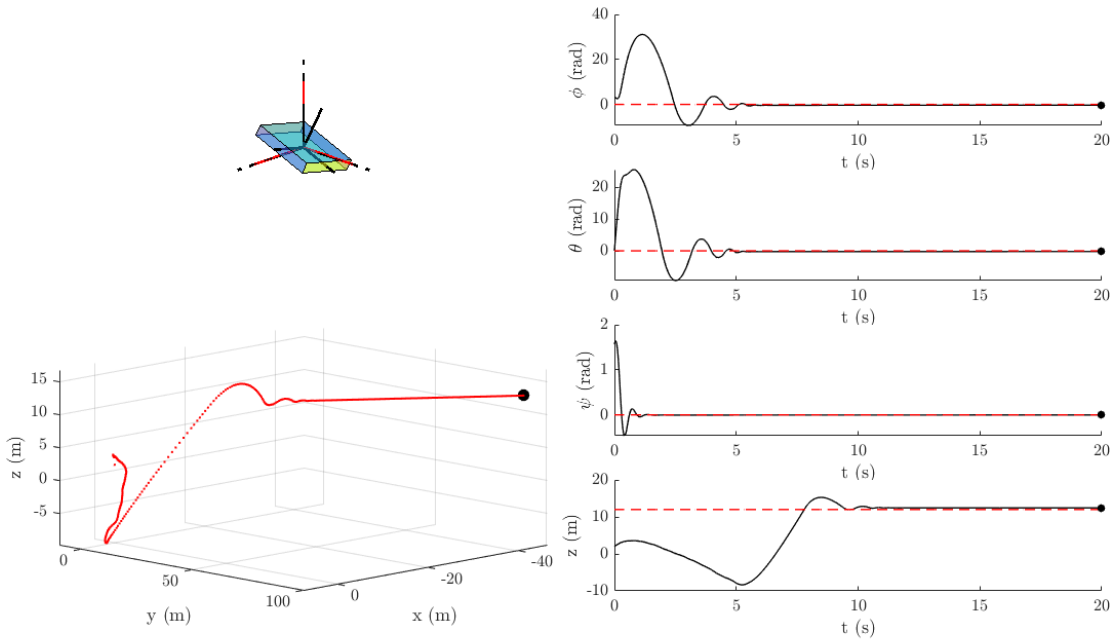

Figure 9: Results of the extreme angular rates test

3.3 2D Weaving

While explicit position control in the horizontal plane is not possible with the implemented integral sliding mode control, if a trajectory in the horizontal plane could be calculated based on the variation of the angular parameters, then position control in the horizontal plane could be attempted implicitly. This test case was to test the ability of the controller to follow a predetermined attitude time history, while also attempting to have the quadcopter follow a pseudo-sinusoidal path in the horizontal plane. The initial conditions of this test are all zeros, the integral gains remain at the same values as in the extreme pitch rate test, and the desired state was as follows, where $a = 1$ and $b = 1$ are constants with units of radians.

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & a\cos t & 0 & -\frac{\pi}{6} & 0 & b\sin t & 0 \end{bmatrix}^T \tag{14}$$

As can be seen by Figure 10, the integral sliding mode controller is able to accurately follow the desired roll and yaw angles. However, there is interesting behavior in the pitch angle. The pitch angle does not remain steadily at $-\pi/6$ as would be expected. Instead, the angle seems to be perturbed around 8 and 9 seconds into the simulation, as well as around 14 and 15 seconds.

This could be due to chattering of the sliding mode controller. While the tanh terms in the control inputs were used to mitigate chattering while the system was moving along the sliding surface, the possibility of chattering still exists. One way to potentially eliminate the chattering during this test case could be to introduce an additional term inside the tanh function within the control inputs, which could scale the argument of the function such that chattering is eliminated.

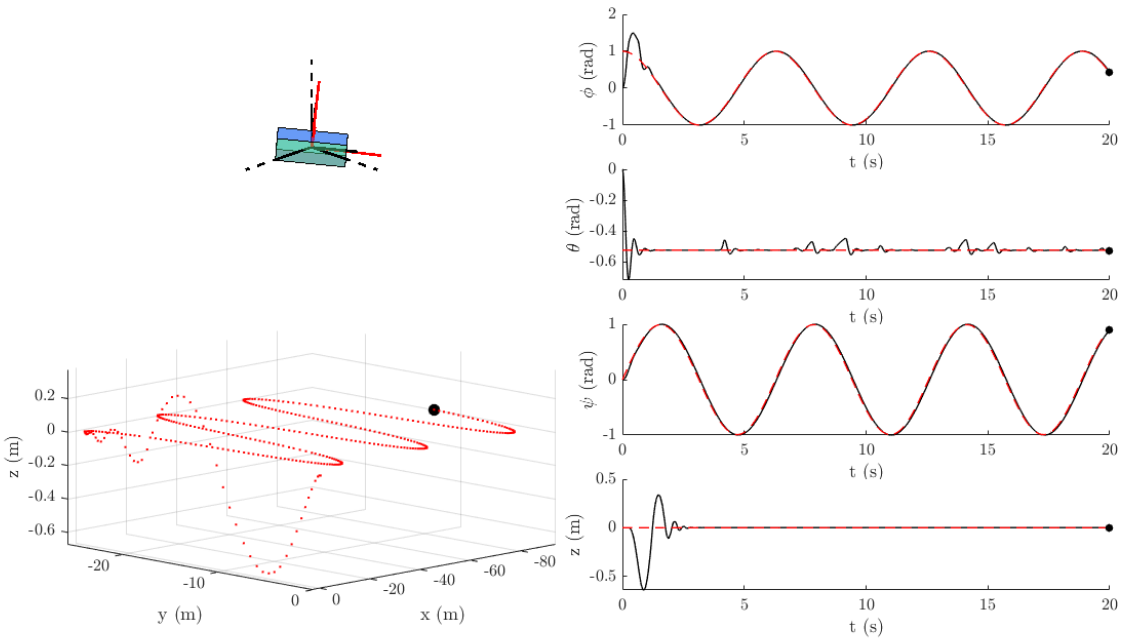

Figure 10: Results of the weaving test

3.4 Step Disturbance Rejection

It was desired to test the disturbance rejection abilities of the implemented controller. Thus, the weaving case was attempted with a wind disturbance in the z direction. Wind was modeled as follows:

$$F_{z-wind} = -k(v_{z-wind} - v_{z-current}) \tag{15}$$

and the disturbance in the z velocity was added into the body dynamics block as can be seen in Figure 11. In this test case, a wind velocity of 40 m/s in the z direction is applied from 4 to 5 seconds.

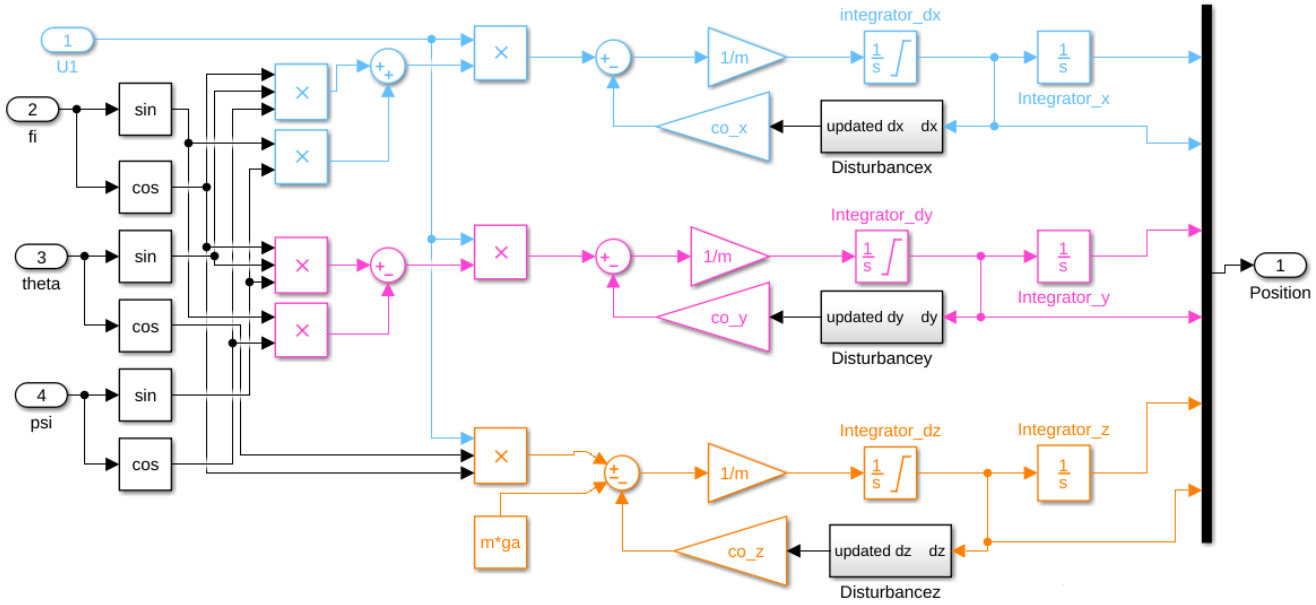

Figure 11: Modeling of wind disturbance in Simulink

The desired states were the same as in the 2D weaving case, and the initial conditions that were applied for this test case were:

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 10 & 0 & 1 & 0 & -\frac{\pi}{6} & 0 & 0 & 1 \end{bmatrix}^T \tag{16}$$

As can be seen in Figure 12, the disturbance causes an increase in the quadcopter's height of almost 26 meters, and the controller brings the drone back to steady state in under 10 seconds. However, the result of the disturbance is that the steady state error at the end of the simulation is -0.4 meters. It is clear from this result, as well as the previous ones, that the gains of the controller need to be better optimized to reduce steady state errors.

The chattering phenomenon is also more evident in the step disturbance test. The values of θ vary up to 0.1 rad, which is a significant amount of chatter. As mentioned in the 2D weaving section, this could be corrected by adding an additional tuning parameter in the tanh function that could be adjusted to smooth out the jagged behavior.

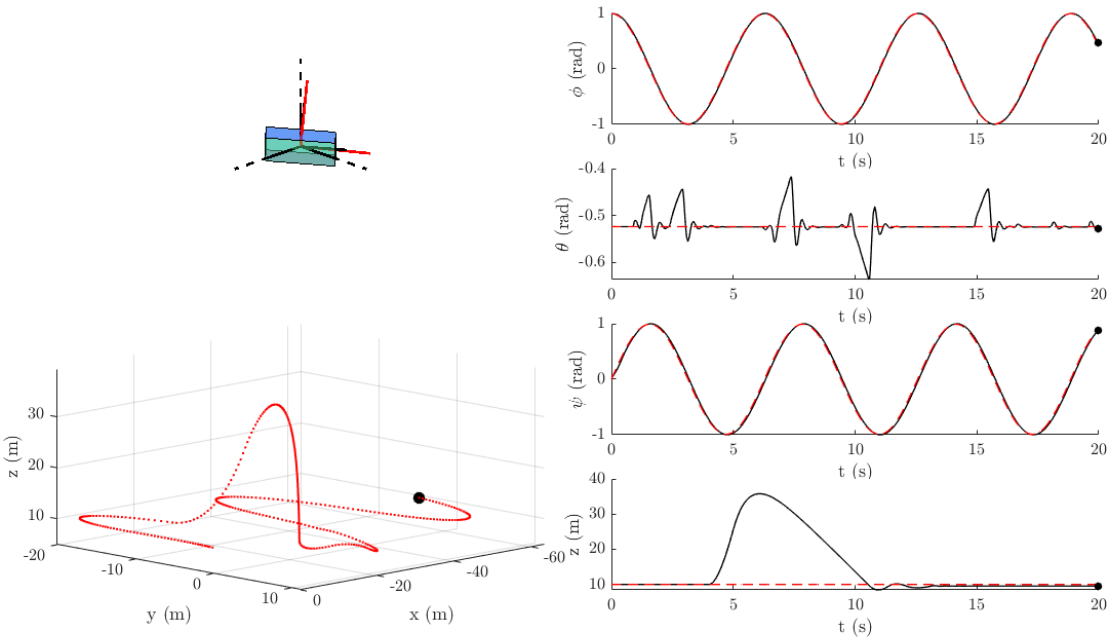

Figure 12: Results of modeling a disturbance in the z direction.

4 Conclusion

Through the work documented in this paper, an integral sliding mode controller has been designed to stabilize the attitude and height of a quadrotor helicopter under arbitrary initial conditions. The controller converged to steady state values quickly, rejected disturbances (albeit with some steady state error), and was effective even under very difficult initial conditions, such as being thrown while the drone was upside down. However, it was determined that pure attitude control coupled with height control of a quadrotor is not the ideal method of stabilization, as significant movement can occur in the uncontrolled horizontal plane due to even slight nonzero steady state errors in the roll, pitch, and yaw angles. Improvements can be made to the controller by seeking analytical ways of optimizing the values of the tuning parameters to improve the steady state error. Additionally, another major improvement would come from expressing positional rates in terms of the other states in order to also control and stabilize the horizontal position of the quadrotor.

References

[1] Sudhir and A. Swamp, "Second order sliding mode control for quadrotor," *2016 IEEE First International Conference on Control, Measurement and Instrumentation (CMI)*, 2016.

[2] H. Jayakrishnan, "Position and Attitude control of a Quadrotor UAV using Super Twisting Sliding Mode," *IFAC-PapersOnLine*, vol. 49, no. 1, pp. 284–289, 2016.

[3] Y. Shtessel, C. Edwards, L. Fridman, and A. Levant, *Sliding Mode Control and Observation*. New York, NY: Springer, Imprint: Birkhäuser, 2014.

[4] O. Camacho, R. Rojas, and W. García-Gabín, "Some long time delay sliding mode control approaches," *ISA Transactions*, vol. 46, no. 1, pp. 95–101, 2007.

[5] Khalil, H. (2002). *Nonlinear systems*. Upper Saddle River, NJ: Prentice Hall.

A runMain.m

```
clear all
close all
clc

% run scripts for parameters
run('QuadrotorConstants.m')
run('InitialConditions.m')
run('TuningParameters.m')

% set default
set(0,'defaultTextInterpreter','latex');

% set time span
tSpan = 20;

t = linspace(0,tSpan,200).';
%
% % % set desired state
% xdes      = 0;
% ydes      = 0;
% zdes      = 10;
% phides    = 0;
% thetades  = 0;
% psides    = 0;
% dxdes     = 0;
% dydes     = 0;
% dzdes     = 0;
% dphides   = 0;
% dthetades = 0;
% dpsides   = 0;
% XDes = [xdes,dxdes,ydes,dydes,zdes,dzdes,phides,dphides,thetades,
   dthetades,psides,dpsides].';

xdes      = ones(length(t),1).*0;
dxdes     = ones(length(t),1).*0;
ydes      = ones(length(t),1).*0;
```

```
dydes     = ones(length(t),1).*0;
zdes      = ones(length(t),1).*0;
dzdes     = ones(length(t),1).*0;
phides    = ones(length(t),1).*cos(t);
dphides   = ones(length(t),1).*0;
thetades  = ones(length(t),1).*-pi/6;
dthetades = ones(length(t),1).*0;
psides    = ones(length(t),1).*sin(t);
dpsides   = ones(length(t),1).*0;
XDes = [t,xdes,dxdes,ydes,dydes,zdes,dzdes,phides,dphides,thetades,
   dthetades,psides,dpsides];

tdes = linspace(0,tSpan,200);

% xmin = -10;
% xmax = 300;
% ymin = -10;
% ymax = 300;
% zmin = -1;
% zmax = 15;

disturbancex = 0;
disturbancey = 0;
disturbancez = 0;
disturbancePhi = 0;
disturbanceTheta = 0;
disturbancePsi = 0;

% run simulink simulation
sim('CL_Xpro_model.slx');

% store simulation info into variables
Time   = state.Time;
x      = state.Data(:,1);
dx     = state.Data(:,2);
y      = state.Data(:,3);
dy     = state.Data(:,4);
z      = state.Data(:,5);
```

```
dz     = state.Data(:,6);
phi    = state.Data(:,7);
dphi   = state.Data(:,8);
theta  = state.Data(:,9);
dtheta = state.Data(:,10);
psi    = state.Data(:,11);
dpsi   = state.Data(:,12);

% animateQuadrotor(state,XDes)


%% plot trajectory
figure(1)
hold on
title('Trajectory')
plot3(x,y,z,'r','LineWidth',1);
xlabel('x (m)');
ylabel('y (m)');
zlabel('z (m)');
grid on
hold off

%% plots linear positions
figure(2)
subplot(3,1,1)
plot(Time,x,'k','LineWidth',1)
title('$x$');
xlabel('t (s)');
ylabel('x (m)');
subplot(3,1,2)
plot(Time,y,'k','LineWidth',1)
title('$y$');
xlabel('t (s)');
ylabel('y (m)');
subplot(3,1,3)
hold on
plot(Time,z,'k','LineWidth',1)
plot(tdes,zdes,'r--','LineWidth',1)
```

```
title('$z$');
xlabel('t (s)');
ylabel('z (m)');

%% plots angular positions
figure(3)
subplot(3,1,1)
hold on
plot(Time,phi,'k','LineWidth',1)
plot(tdes,phides,'r--','LineWidth',1)
hold off
title('$\phi$');
xlabel('t (s)');
ylabel('$\phi$ (rad)');
subplot(3,1,2)
hold on
plot(Time,theta,'k','LineWidth',1)
plot(tdes,thetades,'r--','LineWidth',1)
hold off
title('$\theta$');
xlabel('t (s)');
ylabel('$\theta$ (rad)');
subplot(3,1,3)
hold on
plot(Time,psi,'k','LineWidth',1)
plot(tdes,psides,'r--','LineWidth',1)
hold off
title('$\psi$');
xlabel('t (s)');
ylabel('$\psi$ (rad)');

%% plots linear rates
figure(4)
subplot(3,1,1)
plot(Time,dx,'k','LineWidth',1)
title('$\dot{x}$');
xlabel('t (s)');
ylabel('$\dot{x}$ (m/s)');
```

```
subplot(3,1,2)
plot(Time,dy,'k','LineWidth',1)
title('$\dot{y}$');
xlabel('t (s)');
ylabel('$\dot{y}$ (m/s)');
subplot(3,1,3)
plot(Time,dz,'k','LineWidth',1)
title('$\dot{z}$');
xlabel('t (s)');
ylabel('$\dot{z}$ (m/s)');

%% plots angular rates
figure(5)
subplot(3,1,1)
plot(Time,dphi,'k','LineWidth',1)
title('$\dot{\phi}$')
xlabel('t (s)');
ylabel('$\dot{\phi}$ (rad/s)');
subplot(3,1,2)
plot(Time,dtheta,'k','LineWidth',1)
title('$\dot{\theta}$')
xlabel('t (s)');
ylabel('$\dot{\theta}$ (rad/s)');
subplot(3,1,3)
plot(Time,dpsi,'k','LineWidth',1)
title('$\dot{\psi}$')
xlabel('t (s)');
ylabel('$\dot{\psi}$ (rad/s)');

%% Animate
figure(99)
animateQuadrotor(state,XDes,'HelicalTrajectoryFollowing.gif',0)
% animateQuadrotorConstant(state,XDes,'UpsideDown.gif',1)

%% plot voltages
t  = Vout.Time;
v1 = Vout.Data(:,1);
v2 = Vout.Data(:,2);
```

```
v3 = Vout.Data(:,3);
v4 = Vout.Data(:,4);
figure(6)
plot(t,v4,t,v3,t,v2,t,v1)
title('Temporal variation of voltage')
xlabel('t (s)');
ylabel('$V_{out}$ (V)');
legend({'V_1','V_2','V_3','V_4'})

%% plot U's
t = U_s.Time;
us1 = U_s.Data(:,1);
us2 = U_s.Data(:,2);
us3 = U_s.Data(:,3);
us4 = U_s.Data(:,4);
figure(7)
plot(t,us1,t,us2,t,us3,t,us4)
title('Temporal variation of control inputs')
xlabel('t (s)');
yyaxis left
ylabel('U (N)');
yyaxis right
ylabel('U (N-m)');
legend({'U_1','U_2','U_3','U_4'})
%}

% % plot sliding surfaces
% t = sliding.Time;
% sliding1 = sliding.Data(:,1);
% sliding2 = sliding.Data(:,2);
% sliding3 = sliding.Data(:,3);
% sliding4 = sliding.Data(:,4);
% figure
% clf
% plot(t,sliding4,t,sliding3,t,sliding2,t,sliding1)
% legend({'sliding4','sliding3','sliding2','sliding1'})
```

B QuadrotorConstants.m

```
ga = 9.807;                % Gravitational Acceleration (m/s^2)
m  = 2.1;                  % Total mass (kg)

Jr = 4.86851*0.1;          % Rotor inertia (kgm^2)
L  = 0.422;                % Moment arm CG to rotor (m)

Ixx = 2.1385*1;          % Phi (x-axis) Inertia (kgm^2)
Iyy = Ixx;                 % Theta (y-axis) Inertia (kgm^2)
Izz = 3.7479*1;          % Psi (z-axis) Inertia (kgm^2)

% Ixx = 2.1385*1e-2;         % Phi (x-axis) Inertia (kgm^2)
% Iyy = Ixx;                % Theta (y-axis) Inertia (kgm^2)
% Izz = 3.7479*1e-2;         % Psi (z-axis) Inertia (kgm^2)

Tau = 0.1;                 % Motor/Rotor time const (s)

bb = 2.6846e-07;
dd = 2.4481e-03;

% Rotor Speed-to-Lift & Drag Functions:
% Lift (N) = c*(r/s)^2+d*(r/s)+e
c = 0.0002;                % Rotor Speed (r/s)-to-Lift (N), 1st const
d = 0.0071;                % Rotor Speed (r/s)-to-Lift (N), 2nd const
e = -0.2625;               % Rotor Speed (r/s)-to-Lift (N), 3rd const

% Drag (Nm) = f*(r/s)^2+g*(r/s)+h
f = 5e-6;                  % Rotor Speed (r/s)-to-Drag (Nm), 1st const
g = 0.0008;                % Rotor Speed (r/s)-to-Drag (Nm), 2nd const
h = -0.0282;               % Rotor Speed (r/s)-to-Drag (Nm), 3rd const

% Specific Motor/Rotor Speed Constants: (r/s) = a*(Volts)+b
a1 = 16.235;               % Motor 1 Volts-to-Rotor Speed (r/s), 1st
   const
b1 = -0.5036;              % Motor 1 Volts-to-Rotor Speed (r/s), 2nd
   const
a2 = 17.912;               % Motor 2 Volts-to-Rotor Speed (r/s), 1st
```

```
   const
b2 = -12.728;            % Motor 2 Volts-to-Rotor Speed (r/s), 2nd
   const
a3 = 17.914;             % Motor 3 Volts-to-Rotor Speed (r/s), 1st
   const
b3 = -6.5646;            % Motor 3 Volts-to-Rotor Speed (r/s), 2nd
   const
a4 = 18.161;             % Motor 4 Volts-to-Rotor Speed (r/s), 1st
   const
b4 = -7.4637;            % Motor 4 Volts-to-Rotor Speed (r/s), 2nd
   const

co_x=1.7;
co_y=1.7;
% co_z=90.7;
co_z = 3;

c_mi_x=1.70;
c_mi_y=1.70;
c_mi_z=.40;

angSatLimit = 250; % Angular Saturation Limit

save('QuadrotorConstants.mat')
```

C TuningParameters.m

```
% Tuning Parameters.m
kd_z = 10;
kd_phi = 10;
kd_theta = 10;
kd_psi = 10;

% s = (dXdes-dX)+lambda_x*(Xdes-X)+Ki*int(Xdes-X)dtau
% Proportional
lambda_z     = 50/kd_z;
lambda_phi   = 50/kd_phi;
lambda_theta = 50/kd_theta;
```

```
lambda_psi   = 50/kd_psi;

lambda_x   = 150;
lambda_y   = 150;

lambda = [lambda_z lambda_phi lambda_theta lambda_psi lambda_x
   lambda_y].';

% Kappa
k_z     = 5/kd_z;
k_phi   = 5/kd_phi;
k_theta = 5/kd_theta;
k_psi   = 5/kd_psi;

K = [k_z k_phi k_theta k_psi].';

% delta_z     = 0.3;
% delta_phi   = 0.3;
% delta_theta = 0.3;
% delta_psi   = 0.3;
%
% delta = [delta_z delta_phi delta_theta delta_psi].';

% Eta
eta_z     = 20/kd_z;
eta_phi   = 20/kd_phi;
eta_theta = 20/kd_theta;
eta_psi   = 20/kd_psi;

eta = [eta_z eta_phi eta_theta eta_psi].';

tau_z     = .01;
tau_phi   = .1e+3;
tau_theta = .1e+3;
tau_psi   = .1e+3;

tau = 1e1*[tau_z tau_phi tau_theta tau_psi].';
```

```
% Integral
lambdai_z     = 400/kd_z;
lambdai_phi   = 200/kd_phi;
lambdai_theta = 200/kd_theta;
lambdai_psi   = 100/kd_psi;

lambdai_x   = 1500;
lambdai_y   = 1500;

lambdai = 10*[lambdai_z lambdai_phi lambdai_theta lambdai_psi
   lambdai_x lambdai_y].';

clear kd_z kd_phi kd_theta kd_psi
save('TuningParameters.mat')
```

D animateQuadrotor.m

```
%% Animate the Quadrotor
function animateQuadrotor(state,XDes,filename,animate)
% Unpack state object
t   = state.Time;
x       = state.Data(:,1);
% dx      = state.Data(:,2);
y       = state.Data(:,3);
% dy      = state.Data(:,4);
z       = state.Data(:,5);
% dz      = state.Data(:,6);
phi     = state.Data(:,7);
% dphi    = state.Data(:,8);
theta   = state.Data(:,9);
% dtheta = state.Data(:,10);
psi     = state.Data(:,11);
% dpsi    = state.Data(:,12);

tdes = XDes(:,1);
xdes = XDes(:,2);
ydes = XDes(:,4);
zdes = XDes(:,6);
```

```
phides = XDes(:,8);
thetades = XDes(:,10);
psides = XDes(:,12);

%% Set up figure
fig = figure(99)
set(fig,'units','normalized','outerposition',[0 0 1 1]);
subplot(4,4,[1 2 5 6])
set(gca,'FontSize', 18);
axsize = 2;
axis([-axsize axsize -axsize axsize -axsize axsize])
axis square
axis off
hold on
view(135,20)
% view(3)

subplot(4,4,[9 10 13 14])


subplot(4,4,[1 2 5 6])


%% Defining axes
xaxisref = [0 1;0 0;0 0];
yaxisref = [0 0;0 1;0 0];
zaxisref = [0 0;0 0;0 1];

%% Setting up line objects for each axis in top left figure
subplot(4,4,[1 2 5 6])
set(gca,'FontSize', 30);
xaxis = line('xdata',xaxisref(1,:),'ydata',xaxisref(2,:),...
    'zdata',xaxisref(3,:),'color','black','LineWidth',2);
yaxis = line('xdata',yaxisref(1,:),'ydata',yaxisref(2,:),...
    'zdata',yaxisref(3,:),'color','black','LineWidth',2);
zaxis = line('xdata',zaxisref(1,:),'ydata',zaxisref(2,:),...
    'zdata',zaxisref(3,:),'color','black','LineWidth',2);
```

```
%% Plot Inertial Frame in top left figure
subplot(4,4,[1 2 5 6])
iSize = 3;
plot3(iSize*xaxisref(1,:),iSize*xaxisref(2,:),iSize*xaxisref(3,:),'k
   --','LineWidth',2);
plot3(iSize*yaxisref(1,:),iSize*yaxisref(2,:),iSize*yaxisref(3,:),'k
   --','LineWidth',2);
plot3(iSize*zaxisref(1,:),iSize*zaxisref(2,:),iSize*zaxisref(3,:),'k
   --','LineWidth',2);

%% Setting up line objects for each axis in top left figure
subplot(4,4,[1 2 5 6])
xaxis = line('xdata',xaxisref(1,:),'ydata',xaxisref(2,:),...
    'zdata',xaxisref(3,:),'color','black','LineWidth',2);
yaxis = line('xdata',yaxisref(1,:),'ydata',yaxisref(2,:),...
    'zdata',yaxisref(3,:),'color','black','LineWidth',2);
zaxis = line('xdata',zaxisref(1,:),'ydata',zaxisref(2,:),...
    'zdata',zaxisref(3,:),'color','black','LineWidth',2);

subplot(4,4,[1 2 5 6])
xaxisdes = line('xdata',xaxisref(1,:),'ydata',xaxisref(2,:),...
    'zdata',xaxisref(3,:),'color','red','LineWidth',2);
yaxisdes = line('xdata',yaxisref(1,:),'ydata',yaxisref(2,:),...
    'zdata',yaxisref(3,:),'color','red','LineWidth',2);
zaxisdes = line('xdata',zaxisref(1,:),'ydata',zaxisref(2,:),...
    'zdata',zaxisref(3,:),'color','red','LineWidth',2);



%% Plot Inertial Frame in top left figure
subplot(4,4,[1 2 5 6])
iSize = 3;
plot3(iSize*xaxisref(1,:),iSize*xaxisref(2,:),iSize*xaxisref(3,:),'k
   --','LineWidth',2);
plot3(iSize*yaxisref(1,:),iSize*yaxisref(2,:),iSize*yaxisref(3,:),'k
   --','LineWidth',2);
```

```
plot3(iSize*zaxisref(1,:),iSize*zaxisref(2,:),iSize*zaxisref(3,:),'k
   --','LineWidth',2);

%% Defining pretty box in top left figure
subplot(4,4,[1 2 5 6])
vertices = [-0.75 -0.75 -0.125] + ...
            [0 0 0;
             0 0 0.25;
             0 1.5 0.25;
             0 1.5 0;
             1.5 1.5 0;
             1.5 1.5 0.25;
             1.5 0 0.25;
             1.5 0 0];
faces = [1 2 3 4;5 6 7 8;2 3 6 7;1 4 5 8;1 2 7 8;3 4 5 6];
facecol = [0;1;2;3;4;5];

%% Plot desired axes in top left figure
% subplot(4,4,[1 2 5 6])
% phival = XDes(:,7);
% thetaval = XDes(:,9);
% psival = XDes(:,11);
% A = [1 0 0;0 cos(phival) -sin(phival);0 sin(phival) cos(phival)];
% B = [cos(thetaval) 0 sin(thetaval);0 1 0;-sin(thetaval) 0 cos(
   thetaval)];
% C = [cos(psival),-sin(psival) 0; sin(psival) cos(psival) 0; 0 0
   1];
%
% iCb = A*B*C;
%
% b1des_I = iCb*[1.5;0;0];
% b2des_I = iCb*[0;1.5;0];
% b3des_I = iCb*[0;0;1.5];
%
% plot3([0 b1des_I(1,:)],[0 b1des_I(2,:)],[0 b1des_I(3,:)],'r-','
   LineWidth',2);
% plot3([0 b2des_I(1,:)],[0 b2des_I(2,:)],[0 b2des_I(3,:)],'r-','
   LineWidth',2);
```

```
% plot3([0 b3des_I(1,:)],[0 b3des_I(2,:)],[0 b3des_I(3,:)],'r-','
   LineWidth',2);


%% Set up patch object in top left figure
p = patch('vertices',vertices-0.5,'faces',faces,...
    'facevertexcdata',facecol,'FaceColor','flat', 'FaceAlpha',.5);

drawnow
tic
tnow = toc;

%% Set up bottom left figure
subplot(4,4,[9 10 13 14])
set(gca,'FontSize', 12);
hold on
axis( [min(x)-0.1*max(x), 1.1*max(x), ...
    min(y)-0.1*max(y), 1.1*max(y), ...
    min(z)-0.1*max(z), 1.1*max(z)]);
view(135,20)
xlabel('x (m)')
ylabel('y (m)')
zlabel('z (m)')
grid on


%% Set up COM marker in bottom left figure
subplot(4,4,[9 10 13 14])
CM = line('xdata',0,'ydata',0,'zdata',0,'color','black','marker','.'
   ,'MarkerSize',30);
% CMtrack = line('xdata',0,'ydata',0,'zdata',0,'color','red','marker
   ','.',...
%     'LineWidth',2,'MarkerSize',10);

%% Plot figures on right side
subplot(4,4,[3 4])
set(gca,'FontSize', 12);
```

```
hold on
plot(t,phi,'k','LineWidth',1)
plot(tdes,phides,'r--','LineWidth',1)
hold off
% title('$\phi$');
xlabel('t (s)');
ylabel('$\phi$ (rad)');

subplot(4,4,[7 8])
set(gca,'FontSize', 12);

hold on
plot(t,theta,'k','LineWidth',1)
plot(tdes,thetades,'r--','LineWidth',1)
hold off
% title('$\theta$');
xlabel('t (s)');
ylabel('$\theta$ (rad)');

subplot(4,4,[11 12])
set(gca,'FontSize', 12);

hold on
plot(t,psi,'k','LineWidth',1)
plot(tdes,psides,'r--','LineWidth',1)
hold off
% title('$\psi$');
xlabel('t (s)');
ylabel('$\psi$ (rad)');

subplot(4,4,[15 16])
set(gca,'FontSize', 12);

hold on
plot(t,z,'k','LineWidth',1)
plot(tdes,zdes,'r--','LineWidth',1)
hold off
xlabel('t (s)')
```

```
ylabel('z (m)')

%% Set up Moving Markers on Right Side
subplot(4,4,[3 4])
phi_marker = line('xdata',t(1),'ydata',phi(1),'color','black','
   marker','.','MarkerSize',20);

subplot(4,4,[7 8])
theta_marker = line('xdata',t(1),'ydata',theta(1),'color','black','
   marker','.','MarkerSize',20);

subplot(4,4,[11 12])
psi_marker = line('xdata',t(1),'ydata',psi(1),'color','black','
   marker','.','MarkerSize',20);

subplot(4,4,[15 16])
z_marker = line('xdata',t(1),'ydata',z(1),'color','black','marker','
   .','MarkerSize',20);

scale = 1; % Longer scale means slower animation
i = 1;

if animate == 1
    f = getframe;
    [im,map] = rgb2ind(f.cdata,256,'nodither');
    im(1,1,1,20) = 0;
end
k = 1;

%% Loop through animation
while tnow/scale<=t(end)
    %% Interpolate configuration of quadrotor
    tval = interp1(t*scale,t,tnow);
    xval = interp1(t*scale,x,tnow);
    yval = interp1(t*scale,y,tnow);
    zval = interp1(t*scale,z,tnow);
    phival = interp1(t*scale,phi,tnow);
    thetaval = interp1(t*scale,theta,tnow);
```

```
psival = interp1(t*scale,psi,tnow);

xval_vec(k) = xval;
yval_vec(k) = yval;
zval_vec(k) = zval;

xdesval = interp1(tdes*scale,xdes,tnow);
ydesval = interp1(tdes*scale,ydes,tnow);
zdesval = interp1(tdes*scale,zdes,tnow);
phidesval = interp1(tdes*scale,phides,tnow);
thetadesval = interp1(tdes*scale,thetades,tnow);
psidesval = interp1(tdes*scale,psides,tnow);

%% Desired triad

Ades = [1 0 0;0 cos(phidesval) -sin(phidesval);0 sin(phidesval)
   cos(phidesval)];
Bdes = [cos(thetadesval) 0 sin(thetadesval);0 1 0;-sin(
   thetadesval) 0 cos(thetadesval)];
Cdes = [cos(psidesval),-sin(psidesval) 0; sin(psidesval) cos(
   psidesval) 0; 0 0 1];

iCbdes = Ades*Bdes*Cdes;

b1des_I = iCbdes*[1.5;0;0];
b2des_I = iCbdes*[0;1.5;0];
b3des_I = iCbdes*[0;0;1.5];

%% Set rotation matrix
A = [1 0 0;0 cos(phival) -sin(phival);0 sin(phival) cos(phival)
   ];
B = [cos(thetaval) 0 sin(thetaval);0 1 0;-sin(thetaval) 0 cos(
   thetaval)];
C = [cos(psival),-sin(psival) 0; sin(psival) cos(psival) 0; 0 0
   1];

iCb = A*B*C;
```

```
    b1_I = iCb*[1;0;0];
    b2_I = iCb*[0;1;0];
    b3_I = iCb*[0;0;1];

    %% Plot Height in Bottom Left Figure
    subplot(4,4,[9 10 13 14])
%     set(CMtrack,'xdata',xval_vec(1:i),'ydata',yval_vec(1:i),'zdata
  ',zval_vec(1:i),'color','red',...
%     'LineWidth',2,'MarkerSize',30,'LineStyle','-');
    set(CM,'xdata',xval,'ydata',yval,'zdata',zval,'color','black','
       markers',30)
    plot3(xval,yval,zval,'r.','markers',5)


    %% Plot Moving Points on Right Side
    subplot(4,4,[3 4])
    set(phi_marker,'xdata',tval,'ydata',phival,'color','black','
       marker','.','MarkerSize',20)

    subplot(4,4,[7 8])
    set(theta_marker,'xdata',tval,'ydata',thetaval,'color','black','
       marker','.','MarkerSize',20)

    subplot(4,4,[11 12])
    set(psi_marker,'xdata',tval,'ydata',psival,'color','black','
       marker','.','MarkerSize',20)

    subplot(4,4,[15 16])
    set(z_marker,'xdata',tval,'ydata',zval,'color','black','marker',
       '.','MarkerSize',20)

    %% Calculate New Box Coordinates in top left figure
    subplot(4,4,[1 2 5 6])
    vert = vertices;
    for i = 1:8
        temp = (iCb*vert(i,:).').';
        vert(i,:) = temp;
    end
```

```
    p.Vertices = vert;

    xval = 0;
    yval = 0;
    zval = 0;
    set(xaxis,'Xdata',[xval, xval+b1_I(1)],...
        'Ydata',[yval, yval+b1_I(2)],...
        'Zdata',[zval, zval+b1_I(3)]);
    set(yaxis,'Xdata',[xval, xval+b2_I(1)],...
        'Ydata',[yval, yval+b2_I(2)],...
        'Zdata',[zval, zval+b2_I(3)]);
    set(zaxis,'Xdata',[xval, xval+b3_I(1)],...
        'Ydata',[yval, yval+b3_I(2)],...
        'Zdata',[zval, zval+b3_I(3)]);

    subplot(4,4,[1 2 5 6])
        xvaldes = 0;
    yvaldes = 0;
    zvaldes = 0;

    set(xaxisdes,'Xdata',[xvaldes, xvaldes+b1des_I(1)],...
        'Ydata',[yvaldes, yvaldes+b1des_I(2)],...
        'Zdata',[zvaldes, zvaldes+b1des_I(3)]);
    set(yaxisdes,'Xdata',[xvaldes, xvaldes+b2des_I(1)],...
        'Ydata',[yvaldes, yvaldes+b2des_I(2)],...
        'Zdata',[zvaldes, zvaldes+b2des_I(3)]);
    set(zaxisdes,'Xdata',[xvaldes, xvaldes+b3des_I(1)],...
        'Ydata',[yvaldes, yvaldes+b3des_I(2)],...
        'Zdata',[zvaldes, zvaldes+b3des_I(3)]);

%     plot3([0 b1des_I(1,:)],[0 b1des_I(2,:)],[0 b1des_I(3,:)],'r
   -','LineWidth',2);
%     plot3([0 b2des_I(1,:)],[0 b2des_I(2,:)],[0 b2des_I(3,:)],'r
   -','LineWidth',2);
%     plot3([0 b3des_I(1,:)],[0 b3des_I(2,:)],[0 b3des_I(3,:)],'r
   -','LineWidth',2);
```

```
    drawnow
    tnow = toc;

%     frame = getframe(1);
%     im = frame2im(frame);
%     [imind,cm] = rgb2ind(im,256);
%     if i == 1
%         imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
%     else
%         imwrite(imind,cm,filename,'gif');
%     end
%
    % Write to the GIF File
    if animate == 1
        frame = getframe(gcf);
        im = frame2im(frame);
        [imind,cm] = rgb2ind(im,256);

        if k == 1
            imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
        else
            imwrite(imind,cm,filename,'gif','WriteMode','append','
               DelayTime',0);
        end
    end

   k = k+1;
end

end
```