Drone Controller Optimization Performance Comparison

By

Aditya Bondada

The problem we aim to address is the effective control of drones with a focus on optimizing their performance. The project explicitly focuses on the development and simulation of various control techniques to enhance drone performance in terms of stability and response to disturbances using MATLAB.

The scope of our project includes:

- Mathematical modeling of drone dynamics.
- Designing and simulating different control strategies like PID control, Sliding mode control, and Backstepping control.
- Extensive use of MATLAB for simulation and analysis
- Excluding hardware implementation from the scope, focusing solely on theoretical and simulation aspects.

Importance of Drone Control Optimization

- Drones, or Unmanned Aerial Vehicles (UAVs), have become integral in various sectors, including security, network planning, and logistics.
- Effective drone control is crucial for reliable and efficient operations in these domains. Optimized control ensures precise path planning, stability in varied conditions, and better resource allocation.
- The evolving complexity of tasks drones perform, such as aerial base stations for communication networks or surveillance in security applications, necessitates advanced control strategies for improved performance and coverage.

The optimization of drone control is a fundamental challenge addressed by researchers and engineers globally. Various techniques, including classical methods like PID control, Fuzzy logic Control, and Reactive control to more advanced strategies like Linear Quadratic Regulator (LQR), Model predictive control (MPC), Sliding Mode control, Backstepping control, and various AI, ML, DL techniques have been employed. Researchers strive to strike a balance between theoretical elegance and practical effectiveness, considering factors like robustness, adaptability, and ease of implementation. This project builds on this background, aiming to contribute to the existing knowledge base by systematically evaluating and comparing multiple control strategies in a simulated environment.

In pursuit of optimizing drone performance, we developed a specialized model to effectively implement a Proportional-Integral-Derivative (PID) controller. This model was segmented into five functional blocks: motors and propellers, rotational dynamics, linear dynamics, disturbances, and human control, each represented in Simulink.

Initial Steps in Drone Optimization

Approach 1:

Our first step in optimizing drone performance involved establishing a relationship between voltage and revolutions per minute (RPM), crucial for understanding motor efficiency under various loads. Utilizing a Parrot drone as our model, we developed the following equation to represent this relationship:

$$RPM = -2.6931 \cdot V^3 + 1400 \cdot V$$

This equation was based on an approximation, particularly under an applied voltage of 20 volts. The accuracy of RPM measurements is vital since a drone's lift-off and maneuverability largely depend on its propellers' rotational speed.

To compute the thrust necessary for lift-off, we employed the thrust equation:

$$Thrust = C_T \cdot rho \cdot n^2 \cdot D^4$$

In this formula, rho (air density) is assumed constant at sea level for simplicity. The coefficient of thrust (CT), a critical factor in determining thrust, is calculated using the relationship:

$$C_T = 2 \cdot 10^{-15} \cdot RPM^3 - 4 \cdot 10^{-11} \cdot RPM^2 + 3 \cdot 10^{-7} \cdot RPM + 0.1013$$

Understanding torque's role in drone dynamics led us to formulate an equation connecting torque and RPM:

$$Torque = 4 \cdot 10^{-14} \cdot RPM^3 + 8 \cdot 10^{-12} \cdot RPM^2 + 3 \cdot 10^{-6} \cdot RPM$$

Moreover, we simplified the drone's model by representing its propellers as two perpendicular bars (as shown in Figure 1). This approach aids in calculating the drone's yaw inertia, which we determined by measuring the distance from the propeller's end to the drone's center of gravity, considering the mass at these endpoints. The propellers were treated as rods in our inertia equation:

$$Inertia = 4 \cdot \frac{1}{3} \cdot m \cdot L^2$$

Fig. 1

Further, to calculate the moment of inertia for both roll and pitch, we continued treating each propeller as a single rod, given their symmetry and identical appearance from all sides. As depicted in Figure 1, this assumption implies equal mass and length for all propellers, thus simplifying our calculations.

Additionally, to realistically optimize the drone's performance, we accounted for air resistance or drag using the equation:

$$Drag = \frac{1}{2} \cdot rho \cdot V^2 \cdot A \cdot C_d$$

This equation considers the drone's exposure to wind forces from all directions, significantly affecting its stability and control.

Implementation for Project

Approach 1:

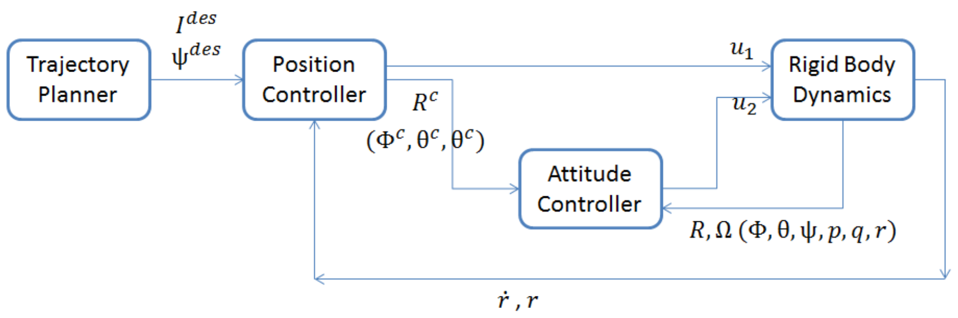
In the development of our Simulink drone model, several key design decisions were pivotal in shaping its functionality and realism. The project began with the motors and propeller's function, which was crucial for simulating the interplay between voltage input and propeller RPM. This aspect is fundamental, as it dictates the drone's thrust and torque, vital for takeoff and flight. Integrating the RPM equation into Simulink allowed us to dynamically analyze how voltage changes impact RPM, providing valuable insights into the propulsion system's efficiency. To further enhance the model's realism, we incorporated a disturbance factor in the linear dynamics function. This allowed us to simulate external influences like wind, which are essential for accurately depicting a drone's flight in different environmental conditions. Additionally, we integrated a function block representing the drone controller from a user’s viewpoint. This block is critical for translating user inputs into voltage changes for the motors, crucial for the drone's motion.

Building on these foundations, we then focused on the drone's rotational and linear dynamics, which are essential for an accurate representation of its movement. By modeling angular and linear velocity and position, we could understand and simulate the drone's behavior in three dimensions. A significant enhancement was adding a disturbance function to mimic real-world conditions, particularly wind forces, thereby increasing the model's authenticity. The final and perhaps most critical design decision was implementing and fine-tuning Proportional-Integral-Derivative (PID) controls for the drone's yaw, pitch, and roll. This step was vital for achieving consistent rotational velocities, key to the drone's stable and responsive flight. The PID controls, known for their adaptive capability, significantly improved the drone's orientation and stability, ensuring accurate responses under varying flight conditions. These design choices collectively enhanced the drone’s operational dynamics, making the Simulink model a more accurate and effective tool for drone simulation and analysis.

Approach 2:

The initial exploration delved into the fundamental mechanics of the quadrotor, identifying the forces, including rotor thrusts (Fi) and gravity (-mga3), as well as the moments arising from thrust and drag moment (Mi) due to propeller rotation.

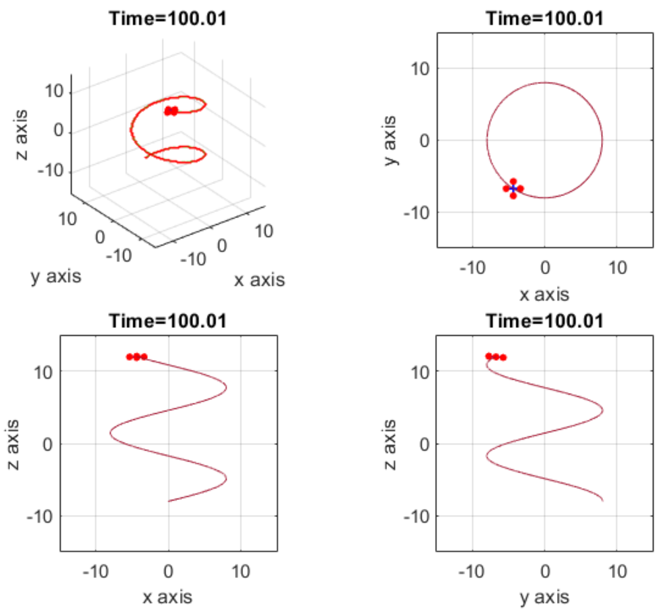
Within the realm of control, the widely studied Linear Control incorporates PD (Proportional Derivative) Control, while Non-Linear Control explores techniques such as First Order Sliding Mode Control and Back Stepping Control Law. Notably, a modular approach is highlighted, suggesting that attitude and position control can be treated independently. This allows for diverse combinations of control laws, such as employing PD control for position control and Lyapunov theory for attitude stabilization.



The control block diagram illustrates an inner loop for attitude control and an outer loop for position control. An essential but not extensively discussed aspect is trajectory generation. Emphasizing the dynamic nature of the quadrotor, it stresses the need for smooth trajectories using waypoints and higher-order polynomial splines to prevent issues like discontinuities and infinite accelerations that actuators cannot handle.

Specifically, a 4th order Minimum Snap Trajectory is proposed for quadrotors based on the relative degree of position variables concerning input u1 and u2. The model employed simplifies by excluding Coriolis forces, and aerodynamic nonlinearities (drag, blade flapping, ground effects), assuming their absence in controlled hover conditions.

In simulation scenarios, the report notes that nonlinearities are assumed absent, and the focus is on presenting control techniques used in UAV research. The controller output is directly fed into the dynamic model without actuator space mapping. However, it acknowledges the necessity of considering actuator constraints in practical implementations. In the simulations, thresholds are imposed on thrust and torque inputs to ensure practicality, preventing situations where high inputs may exceed the actuators' capabilities.



Results:

In the absence of disturbances, PD control demonstrates excellent tracking with negligible error. SMC control improves tracking accuracy over PD but exhibits discontinuous and frequent switching in control. Backstepping control outperforms both PD and SMC, achieving the least error magnitudes with smooth and continuous control. Figures illustrate the precision of positions, orientations, control inputs, and trajectory errors under disturbance-free conditions.

PID results:

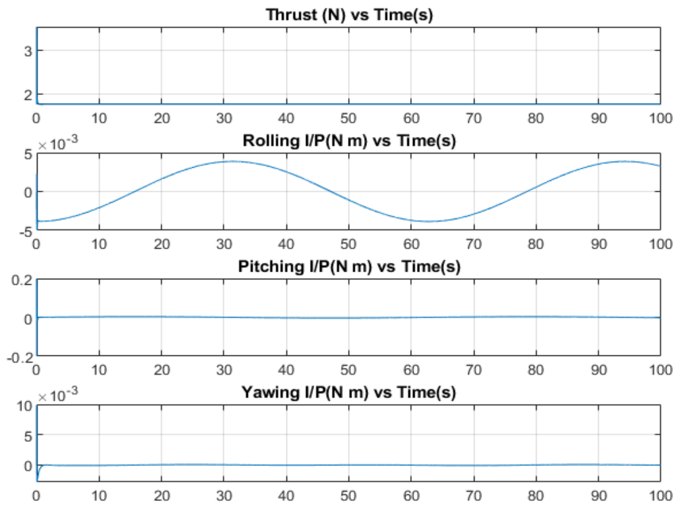


Fig. 2 Thrust, Rolling, Pitching and Yawing Inputs vs Time

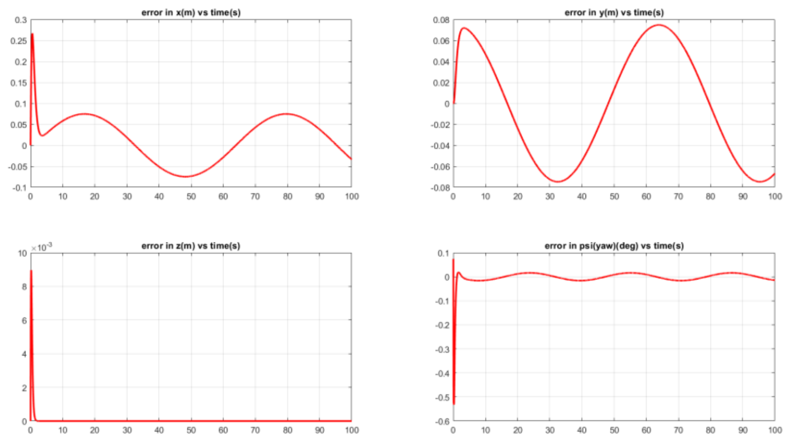


Fig. 3 Errors in X,Y,Z and yaw

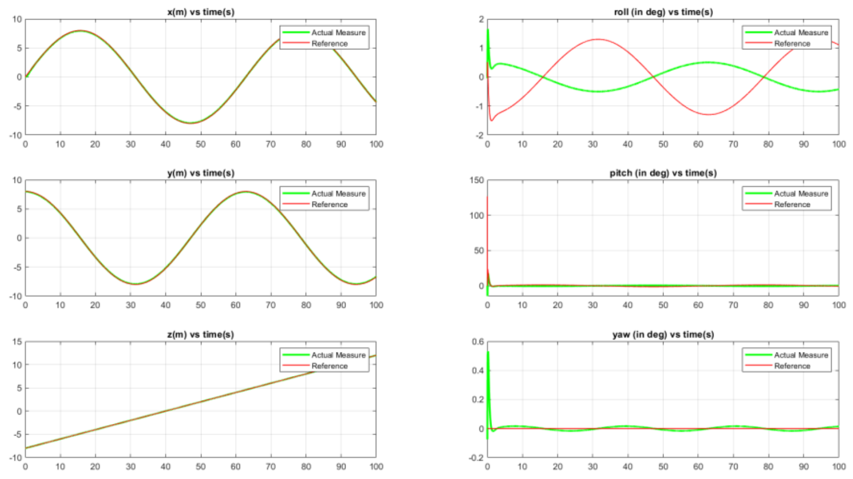


Fig. 4 Position and Orientation vs Time

SMC RESULTS

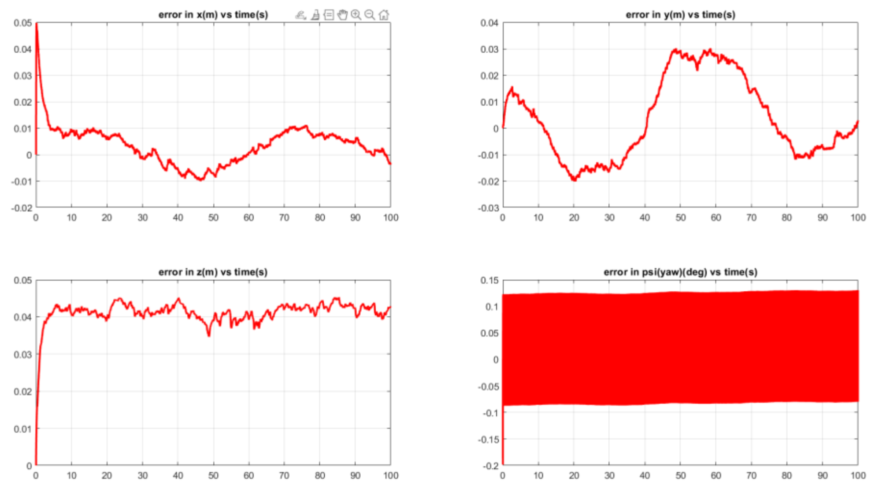


Fig. 5 Errors in x,y,z and yaw

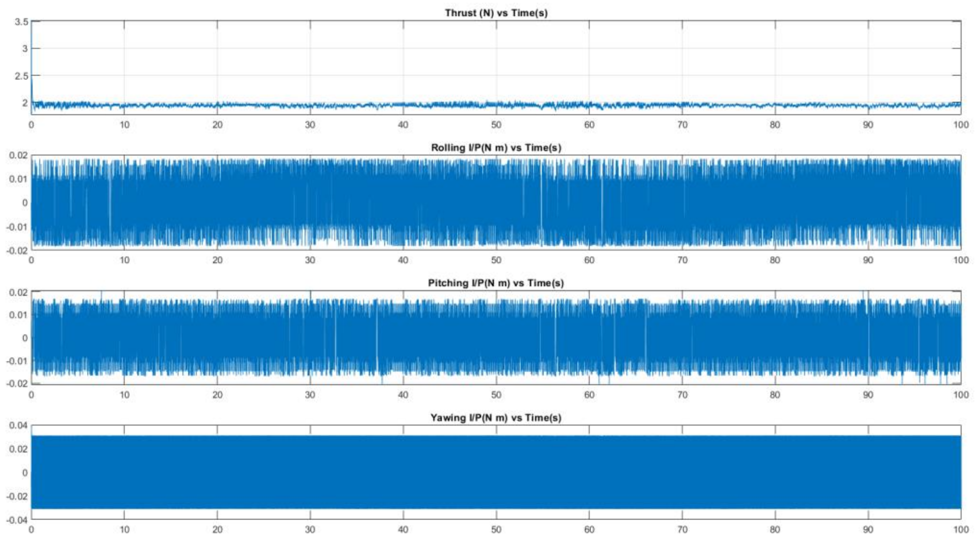


Fig.6 Thrust, Rolling, Pitching and Yawing Inputs vs Time

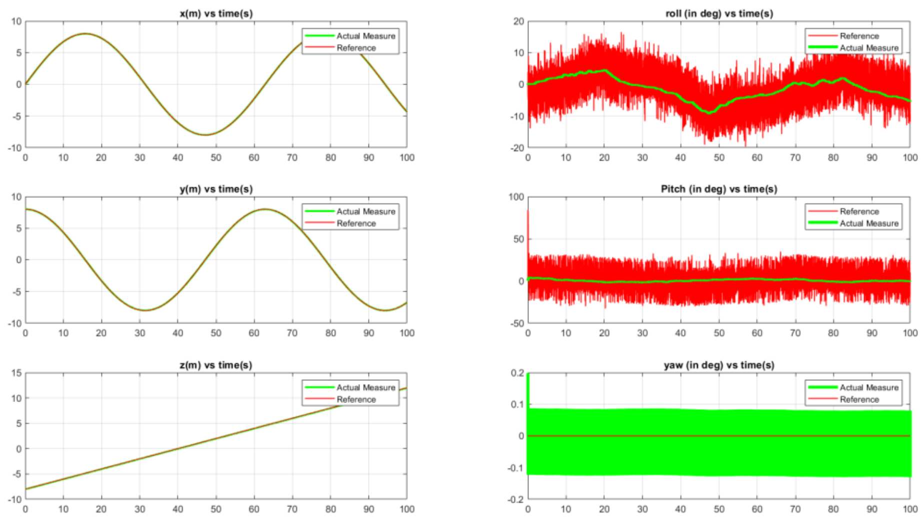


Fig.7 Position and Orientation vs Time

Back Stepping Results:

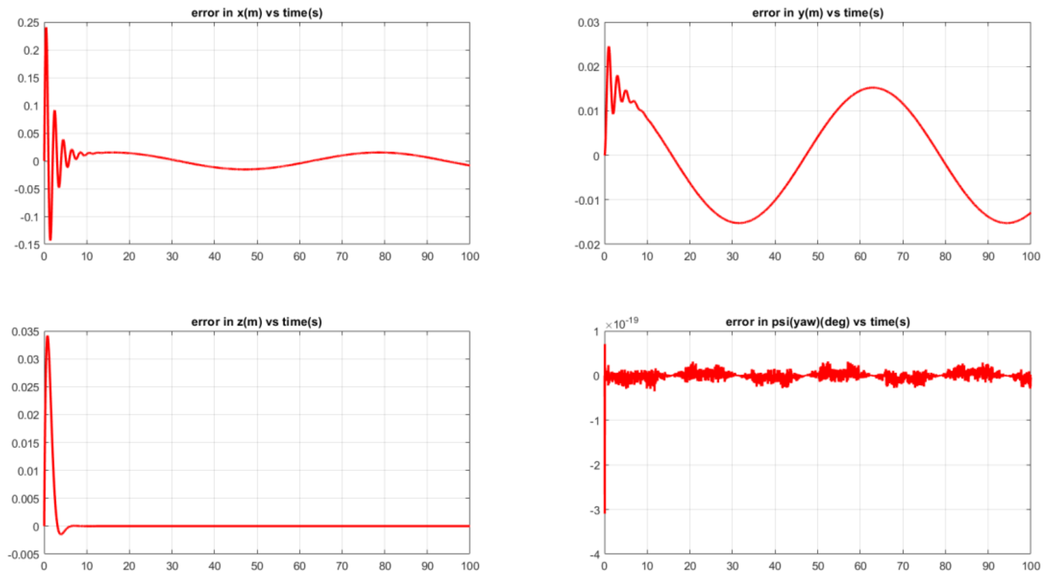


Fig. 8 Errors in x,y,z and yaw

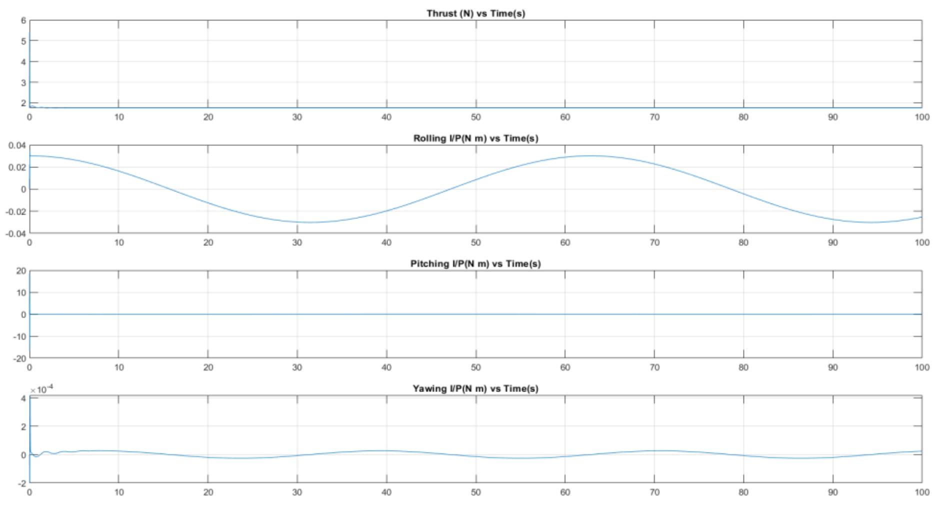


Fig.9 Thrust, Rolling, Pitching and Yawing Inputs vs Time

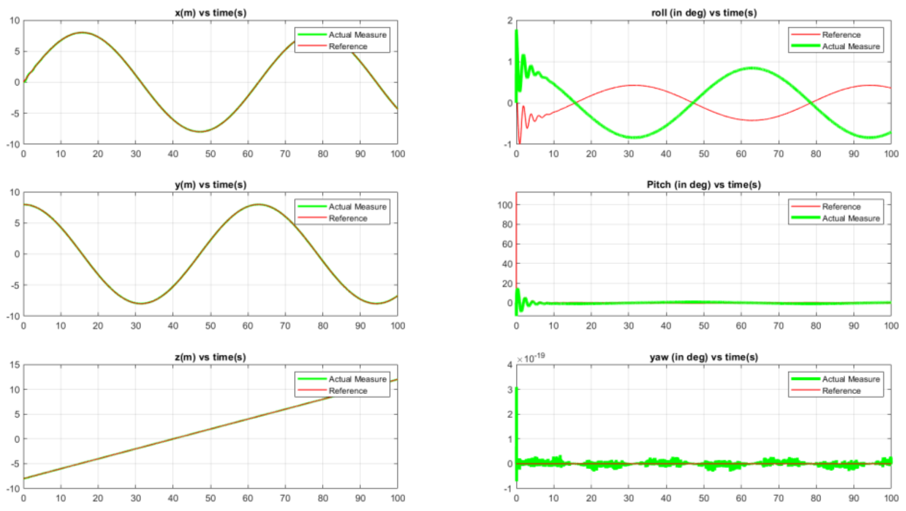


Fig. 10 Position and Orientation vs Time

Summary:

Summary of Proportional Derivative (PD) Control:

PD Control is simple to implement and understand, requiring minimal tuning parameters. However, its use of the derivative term can amplify noise in real systems, and it lacks robustness against disturbances and parameter variations.

Summary of Sliding Mode Control (SMC):
SMC is easy to implement and robust against parameter variations and disturbances. Nevertheless, its discontinuous control law may adversely affect actuators, and the reliance on high gains poses a risk of actuator saturation.

Summary of Backstepping Control (BSC):

BSC ensures Lyapunov Stability and is not dependent on feedback linearization, offering system independence. Despite its mathematical complexity and the need to tune multiple gains, it remains stable in the presence of disturbances, with the option of employing integral backstepping to address finite steady-state errors.

Challenges:

Throughout the project, both individual and team challenges arose. Initially, implementing human control in the simple model proved problematic, as providing a constant voltage was deemed inadequate. Determining the appropriate placement for the PID controller was another challenge, with a systematic approach requiring the addition of PID controllers for yaw, pitch, and roll adjustments. The limitations of the simple model prompted the development of a more generalized form to compare various drone types effectively.

A significant hurdle involved implementing different controllers within the same framework. Existing frameworks, found online, were often focused on a single controller and lacked modularity. To address this, we opted for a simple, modular model to facilitate the seamless integration of multiple controllers, such as LQR and MPC, for comprehensive and justifiable comparisons.

Takeaways from Project

Throughout the project, we discovered the nuances of various controllers, recognizing that each comes with its own merits and demerits. This emphasized the importance of careful consideration when selecting a control algorithm for a specific system, as there is no universally superior algorithm.

Simultaneously, our engagement with drone mechanics in Simulink deepened our understanding of how a drone's mechanical components and flight dynamics are interrelated. We also gained insights into the complexities of user-drone interaction, realizing the critical role of accurate signal translation from controller input to voltage changes for precise control.

Additionally, our exploration of different types of controllers provided a practical understanding of their effective use based on data changes, particularly in stabilizing the drone. This comprehensive learning approach allowed us to bridge theoretical knowledge with practical implementation, enhancing our overall comprehension of drone control systems.

Future Scope:

Future project plans include experimentation with control algorithms like LQR and NMPC aim to optimize drone performance, and we'll improve the Simulink model to mimic real drone controllers more closely. Additionally, obstacle interaction tests in the simulation will enhance understanding of drone maneuverability in challenging scenarios. We also plan to implement a PID controller for throttle refinement during critical drone stages. We'll fine-tune PID settings for varying air densities, enhancing adaptability.

Bibliography

Control of quadrotor using nonlinear model predictive control - MATLAB & Simulink. (n.d.). MathWorks. Retrieved December 4, 2023, from https://www.mathworks.com/help/mpc/ug/control-of-quadrotor-using-nonlinear-model-predictive-control.html

Douglas, B. (n.d.). *Drone simulation and control, part 1: Setting up the control problem video*. Drone Simulation and Control, Part 1 Video - MATLAB & Simulink. Retrieved December 4, 2023, from https://www.mathworks.com/videos/drone-simulation-and-control-part-1-setting-up-the-control-problem-1539323440930.html?s_tid=srchtitle_videos_main_4_Drone%20Simulation%20and%20Control

Ferry, N. (n.d.). *MATLAB/Simulink implementaioof simple quadcopter model and controller*.

Li, D. (2022, June 24). *Modeling, Controlling, and Flight Testing of a Small Quadcopter*. https://deepblue.lib.umich.edu/bitstream/handle/2027.42/176758/Honors_Capstone_Modeling_Controlling_and_Flight_Testing_of_a_Small_Quadcopter_-_David_Li.pdf?sequence=1

Quadcopter Project - MATLAB & Simulink. (n.d.-a). MathWorks . Retrieved December 4, 2023, from https://www.mathworks.com/help/aeroblks/quadcopter-project.html

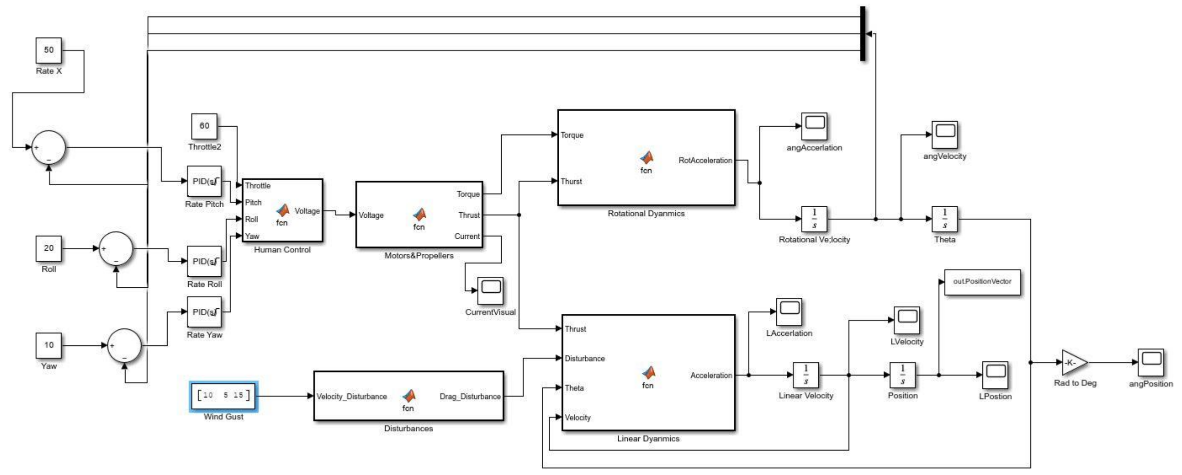
Quadcopter Project - MATLAB & Simulink. (n.d.-b). MathWorks. Retrieved December 4, 2023, from https://www.mathworks.com/help/aeroblks/quadcopter-project.html

Robert Mahony, Vijay Kumar and Peter Corke, *Modeling, Estimation, and Control of Quadrotor*. *Digital Object Identifer.* 10.1109/MRA.2012.2206474, Date of publication: 27 August 2012

Robotics: Aerial robotics. (n.d.). Coursera. Retrieved December 4, 2023, from https://www.coursera.org/learn/robotics-flight

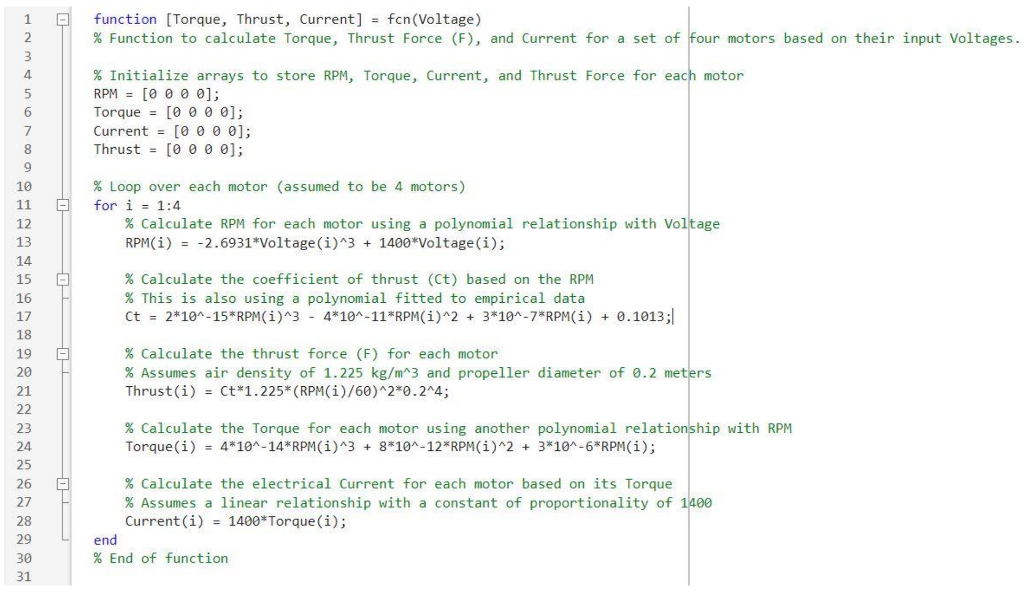
Appendix

Simulink Model for Mavic



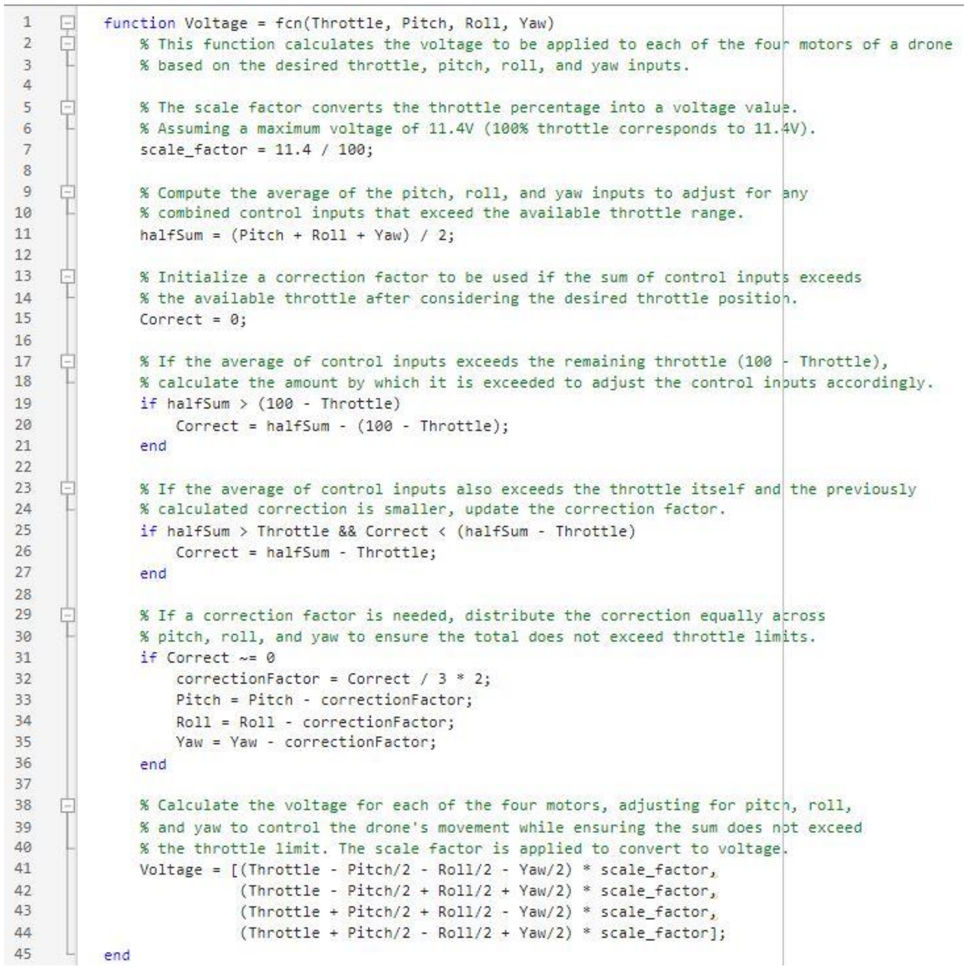
Code for Functions:

Motors & Propellers



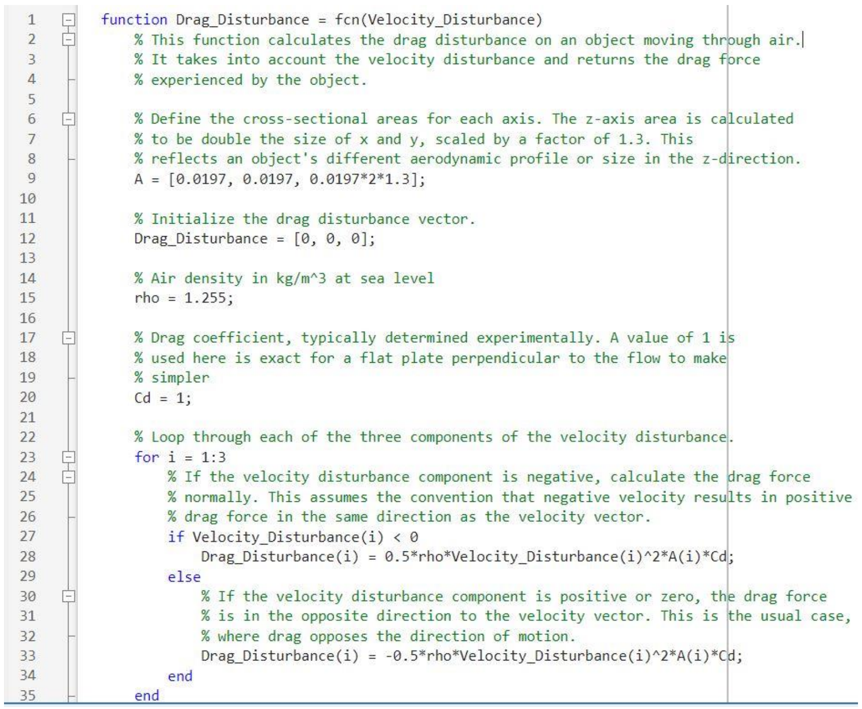
```
function [Torque, Thrust, Current] = fcn(Voltage)
% Function to calculate Torque, Thrust Force (F), and Current for a set of four motors based on their input Voltages.

% Initialize arrays to store RPM, Torque, Current, and Thrust Force for each motor
RPM = [0 0 0 0];
Torque = [0 0 0 0];
Current = [0 0 0 0];
Thrust = [0 0 0 0];

% Loop over each motor (assumed to be 4 motors)
for i = 1:4
    % Calculate RPM for each motor using a polynomial relationship with Voltage
    RPM(i) = -2.6931*Voltage(i)^3 + 1400*Voltage(i);

    % Calculate the coefficient of thrust (Ct) based on the RPM
    % This is also using a polynomial fitted to empirical data
    Ct = 2*10^-15*RPM(i)^3 - 4*10^-11*RPM(i)^2 + 3*10^-7*RPM(i) + 0.1013;

    % Calculate the thrust force (F) for each motor
    % Assumes air density of 1.225 kg/m^3 and propeller diameter of 0.2 meters
    Thrust(i) = Ct*1.225*(RPM(i)/60)^2*0.2^4;

    % Calculate the Torque for each motor using another polynomial relationship with RPM
    Torque(i) = 4*10^-14*RPM(i)^3 + 8*10^-12*RPM(i)^2 + 3*10^-6*RPM(i);

    % Calculate the electrical Current for each motor based on its Torque
    % Assumes a linear relationship with a constant of proportionality of 1400
    Current(i) = 1400*Torque(i);
end
% End of function
```

Human Controller



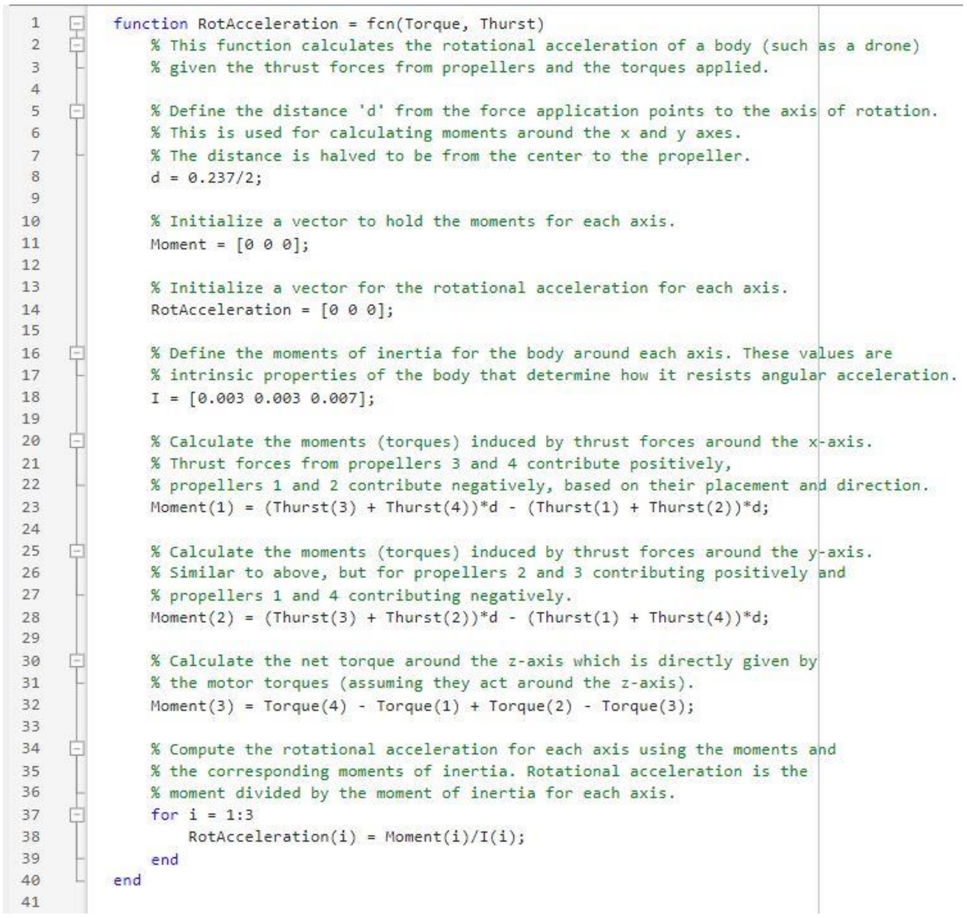
```matlab
function Voltage = fcn(Throttle, Pitch, Roll, Yaw)
    % This function calculates the voltage to be applied to each of the four motors of a drone
    % based on the desired throttle, pitch, roll, and yaw inputs.

    % The scale factor converts the throttle percentage into a voltage value.
    % Assuming a maximum voltage of 11.4V (100% throttle corresponds to 11.4V).
    scale_factor = 11.4 / 100;

    % Compute the average of the pitch, roll, and yaw inputs to adjust for any
    % combined control inputs that exceed the available throttle range.
    halfSum = (Pitch + Roll + Yaw) / 2;

    % Initialize a correction factor to be used if the sum of control inputs exceeds
    % the available throttle after considering the desired throttle position.
    Correct = 0;

    % If the average of control inputs exceeds the remaining throttle (100 - Throttle),
    % calculate the amount by which it is exceeded to adjust the control inputs accordingly.
    if halfSum > (100 - Throttle)
        Correct = halfSum - (100 - Throttle);
    end

    % If the average of control inputs also exceeds the throttle itself and the previously
    % calculated correction is smaller, update the correction factor.
    if halfSum > Throttle && Correct < (halfSum - Throttle)
        Correct = halfSum - Throttle;
    end

    % If a correction factor is needed, distribute the correction equally across
    % pitch, roll, and yaw to ensure the total does not exceed throttle limits.
    if Correct ~= 0
        correctionFactor = Correct / 3 * 2;
        Pitch = Pitch - correctionFactor;
        Roll = Roll - correctionFactor;
        Yaw = Yaw - correctionFactor;
    end

    % Calculate the voltage for each of the four motors, adjusting for pitch, roll,
    % and yaw to control the drone's movement while ensuring the sum does not exceed
    % the throttle limit. The scale factor is applied to convert to voltage.
    Voltage = [(Throttle - Pitch/2 - Roll/2 - Yaw/2) * scale_factor,
               (Throttle - Pitch/2 + Roll/2 + Yaw/2) * scale_factor,
               (Throttle + Pitch/2 + Roll/2 - Yaw/2) * scale_factor,
               (Throttle + Pitch/2 - Roll/2 + Yaw/2) * scale_factor];
end
```

Disturbances



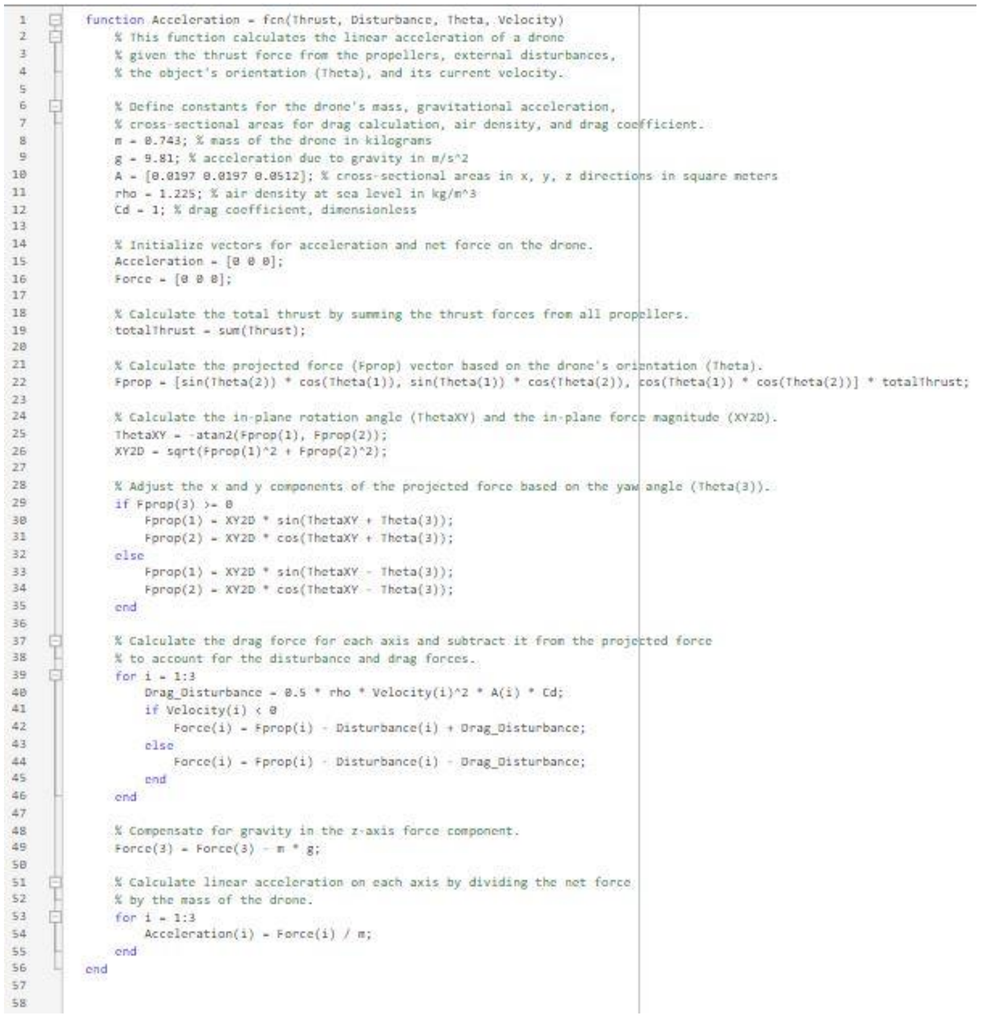
```
function Drag_Disturbance = fcn(Velocity_Disturbance)
    % This function calculates the drag disturbance on an object moving through air.
    % It takes into account the velocity disturbance and returns the drag force
    % experienced by the object.

    % Define the cross-sectional areas for each axis. The z-axis area is calculated
    % to be double the size of x and y, scaled by a factor of 1.3. This
    % reflects an object's different aerodynamic profile or size in the z-direction.
    A = [0.0197, 0.0197, 0.0197*2*1.3];

    % Initialize the drag disturbance vector.
    Drag_Disturbance = [0, 0, 0];

    % Air density in kg/m^3 at sea level
    rho = 1.255;

    % Drag coefficient, typically determined experimentally. A value of 1 is
    % used here is exact for a flat plate perpendicular to the flow to make
    % simpler
    Cd = 1;

    % Loop through each of the three components of the velocity disturbance.
    for i = 1:3
        % If the velocity disturbance component is negative, calculate the drag force
        % normally. This assumes the convention that negative velocity results in positive
        % drag force in the same direction as the velocity vector.
        if Velocity_Disturbance(i) < 0
            Drag_Disturbance(i) = 0.5*rho*Velocity_Disturbance(i)^2*A(i)*Cd;
        else
            % If the velocity disturbance component is positive or zero, the drag force
            % is in the opposite direction to the velocity vector. This is the usual case,
            % where drag opposes the direction of motion.
            Drag_Disturbance(i) = -0.5*rho*Velocity_Disturbance(i)^2*A(i)*Cd;
        end
    end
```

Rotational Dynamics



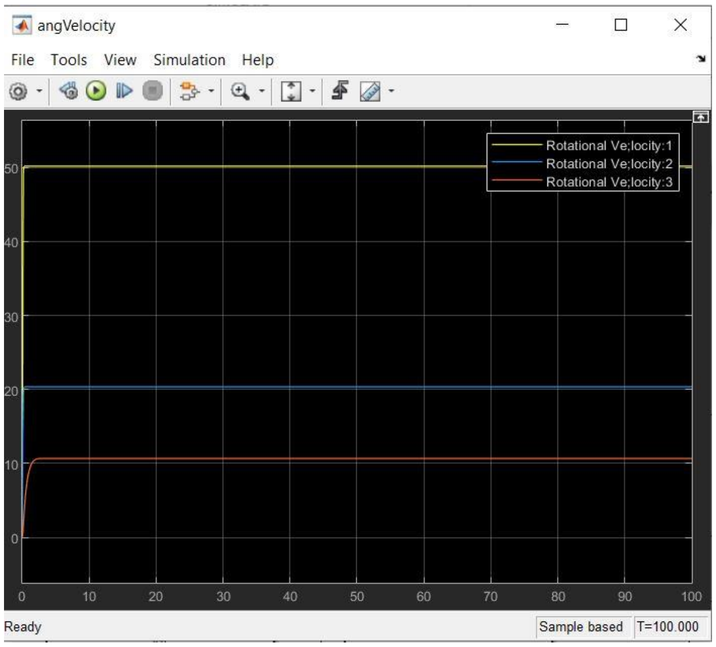
```
function RotAcceleration = fcn(Torque, Thurst)
    % This function calculates the rotational acceleration of a body (such as a drone)
    % given the thrust forces from propellers and the torques applied.

    % Define the distance 'd' from the force application points to the axis of rotation.
    % This is used for calculating moments around the x and y axes.
    % The distance is halved to be from the center to the propeller.
    d = 0.237/2;

    % Initialize a vector to hold the moments for each axis.
    Moment = [0 0 0];

    % Initialize a vector for the rotational acceleration for each axis.
    RotAcceleration = [0 0 0];

    % Define the moments of inertia for the body around each axis. These values are
    % intrinsic properties of the body that determine how it resists angular acceleration.
    I = [0.003 0.003 0.007];

    % Calculate the moments (torques) induced by thrust forces around the x-axis.
    % Thrust forces from propellers 3 and 4 contribute positively,
    % propellers 1 and 2 contribute negatively, based on their placement and direction.
    Moment(1) = (Thurst(3) + Thurst(4))*d - (Thurst(1) + Thurst(2))*d;

    % Calculate the moments (torques) induced by thrust forces around the y-axis.
    % Similar to above, but for propellers 2 and 3 contributing positively and
    % propellers 1 and 4 contributing negatively.
    Moment(2) = (Thurst(3) + Thurst(2))*d - (Thurst(1) + Thurst(4))*d;

    % Calculate the net torque around the z-axis which is directly given by
    % the motor torques (assuming they act around the z-axis).
    Moment(3) = Torque(4) - Torque(1) + Torque(2) - Torque(3);

    % Compute the rotational acceleration for each axis using the moments and
    % the corresponding moments of inertia. Rotational acceleration is the
    % moment divided by the moment of inertia for each axis.
    for i = 1:3
        RotAcceleration(i) = Moment(i)/I(i);
    end
end
```

Linear Dynamics



```matlab
function Acceleration = fcn(Thrust, Disturbance, Theta, Velocity)
    % This function calculates the linear acceleration of a drone
    % given the thrust force from the propellers, external disturbances,
    % the object's orientation (Theta), and its current velocity.

    % Define constants for the drone's mass, gravitational acceleration,
    % cross-sectional areas for drag calculation, air density, and drag coefficient.
    m = 0.743; % mass of the drone in kilograms
    g = 9.81; % acceleration due to gravity in m/s^2
    A = [0.0197 0.0197 0.0512]; % cross-sectional areas in x, y, z directions in square meters
    rho = 1.225; % air density at sea level in kg/m^3
    Cd = 1; % drag coefficient, dimensionless

    % Initialize vectors for acceleration and net force on the drone.
    Acceleration = [0 0 0];
    Force = [0 0 0];

    % Calculate the total thrust by summing the thrust forces from all propellers.
    totalThrust = sum(Thrust);

    % Calculate the projected force (Fprop) vector based on the drone's orientation (Theta).
    Fprop = [sin(Theta(2)) * cos(Theta(1)), sin(Theta(1)) * cos(Theta(2)), cos(Theta(1)) * cos(Theta(2))] * totalThrust;

    % Calculate the in-plane rotation angle (ThetaXY) and the in-plane force magnitude (XY2D).
    ThetaXY = -atan2(Fprop(1), Fprop(2));
    XY2D = sqrt(Fprop(1)^2 + Fprop(2)^2);

    % Adjust the x and y components of the projected force based on the yaw angle (Theta(3)).
    if Fprop(3) >= 0
        Fprop(1) = XY2D * sin(ThetaXY + Theta(3));
        Fprop(2) = XY2D * cos(ThetaXY + Theta(3));
    else
        Fprop(1) = XY2D * sin(ThetaXY - Theta(3));
        Fprop(2) = XY2D * cos(ThetaXY - Theta(3));
    end

    % Calculate the drag force for each axis and subtract it from the projected force
    % to account for the disturbance and drag forces.
    for i = 1:3
        Drag_Disturbance = 0.5 * rho * Velocity(i)^2 * A(i) * Cd;
        if Velocity(i) < 0
            Force(i) = Fprop(i) - Disturbance(i) + Drag_Disturbance;
        else
            Force(i) = Fprop(i) - Disturbance(i) - Drag_Disturbance;
        end
    end

    % Compensate for gravity in the z-axis force component.
    Force(3) = Force(3) - m * g;

    % Calculate linear acceleration on each axis by dividing the net force
    % by the mass of the drone.
    for i = 1:3
        Acceleration(i) = Force(i) / m;
    end
end
```

Angular Velocity Scope



Approach 2

Code to run the simulation:

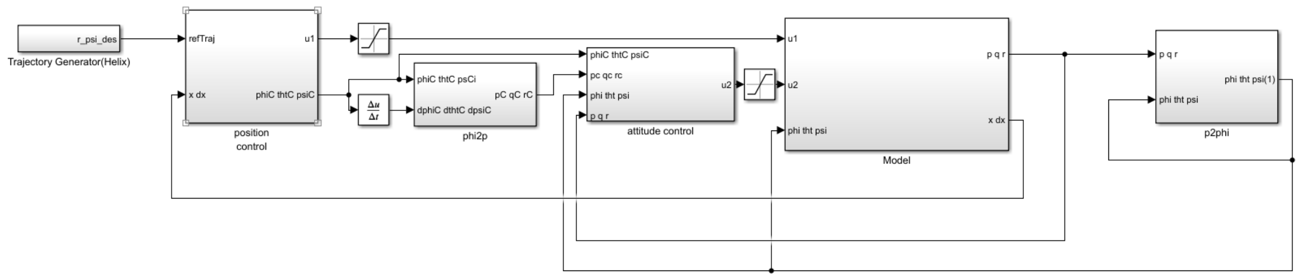
```
clear all
close all
 global m I g tuning_parameter


m = 0.18;
 I = [0.00025,   0,          2.55e-6;
      0,         0.000232,   0;
      2.55e-6,   0,          0.0003738];
 g=9.8;
 tuning_parameter=100;
 sim('quad_control');
 d=1;
 x=xyz(:,1);y=xyz(:,2);z=xyz(:,3);
 phi=phi_tht_psi(:,1);tht=phi_tht_psi(:,2);psi=phi_tht_psi(:,3);
 phiC=angc(:,1);thtC=angc(:,2);psiC=angc(:,3);
 zup=[0;0;0.2];
 for i=1:50:length(x)
       Rotn=[cos(psi(i))*cos(tht(i))-sin(phi(i))*sin(psi(i))*sin(tht(i)) -sin(psi(i))*cos(phi(i))
cos(psi(i))*sin(tht(i))+sin(psi(i))*sin(phi(i))*cos(tht(i));
             cos(tht(i))*sin(psi(i))+cos(psi(i))*sin(phi(i))*sin(tht(i))  cos(phi(i))*cos(psi(i))
sin(psi(i))*sin(tht(i))-cos(psi(i))*cos(tht(i))*sin(phi(i));
                                       -cos(phi(i))*sin(tht(i))           sin(phi(i))
cos(phi(i))*cos(tht(i))];


      A=[x(i);y(i);z(i)]+Rotn*[0;-d;0];
       B=[x(i);y(i);z(i)]+Rotn*[d;0;0];
       C=[x(i);y(i);z(i)]+Rotn*[0;d;0];
       D=[x(i);y(i);z(i)]+Rotn*[-d;0;0];
       Zup=[x(i);y(i);z(i)]+Rotn*zup;
       ACx=linspace(A(1),C(1),10);
       ACy=linspace(A(2),C(2),10);
       ACz=linspace(A(3),C(3),10);
       BDx=linspace(B(1),D(1),10);
       BDy=linspace(B(2),D(2),10);
       BDz=linspace(B(3),D(3),10);
       Zupx=linspace(x(i),Zup(1),10);
       Zupy=linspace(y(i),Zup(2),10);
       Zupz=linspace(z(i),Zup(3),10);




    subplot(221)


    plot3(A(1),A(2),A(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
     hold on
     plot3(B(1),B(2),B(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
     hold on
     plot3(C(1),C(2),C(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
     hold on
     plot3(D(1),D(2),D(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 %   plot3(ref(:,1),ref(:,2),ref(:,3),'--gs','LineWidth',1,'MarkerSize',1,'MarkerEdgeColor','r');
     plot3(ref(:,1),ref(:,2),ref(:,3),'--
gs','LineWidth',0.5,'MarkerSize',0.5,'MarkerEdgeColor','r');
     hold on
     plot3(ACx,ACy,ACz,'-b','LineWidth',1);
     hold on
     plot3(BDx,BDy,BDz,'-b','LineWidth',1);
     hold on
     plot3(Zupx,Zupy,Zupz,'linewidth',1);
     hold off
```

```matlab
 axis([-15 +15 -15 +15 -15 +15]);
 axis square
 grid
 xlabel('x axis');
 ylabel('y axis');
 zlabel('z axis');
 title(['Time=',num2str(i*0.01)])
 pause(0.01);


subplot(222)
 title(['Time=',num2str(i*0.01)])
 plot(ACx,ACy,'-b','LineWidth',1);
 hold on;
 plot(BDx,BDy,'-b','LineWidth',1);
 hold on;
 plot(A(1),A(2),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(B(1),B(2),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(C(1),C(2),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(D(1),D(2),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 plot(ref(:,1),ref(:,2));
 hold off;
 grid;
 axis([-15 +15 -15 +15]);
 axis square
 xlabel('x axis');
 ylabel('y axis');
 title(['Time=',num2str(i*0.01)])


subplot(223)
 plot(ACx,ACz,'-b','LineWidth',1);
 hold on;
 plot(BDx,BDz,'-b','LineWidth',1);
 hold on;
 plot(A(1),A(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(B(1),B(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(C(1),C(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 hold on;
 plot(D(1),D(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
 plot(ref(:,1),ref(:,3));
 hold off;
 grid;
 axis([-15 +15 -15 +15]);
 axis square
 xlabel('x axis');
 ylabel('z axis');
 title(['Time=',num2str(i*0.01)])
```

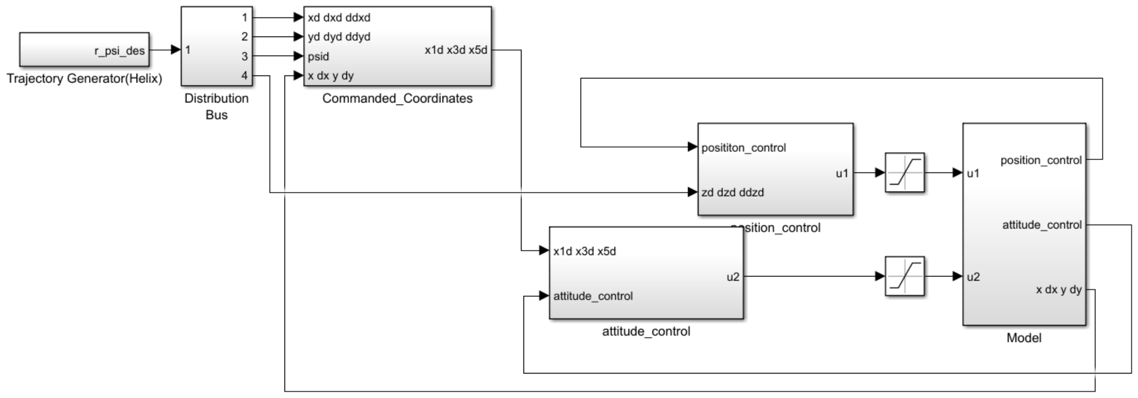
```
   subplot(224)
    plot(ACy,ACz,'-b','LineWidth',1);
    hold on;
    plot(BDy,BDz,'-b','LineWidth',1);
    hold on;
    plot(A(2),A(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
    hold on;
    plot(B(2),B(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
    hold on;
    plot(C(2),C(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
    hold on;
    plot(D(2),D(3),'-mo','MarkerFaceColor',[1 0 0],'MarkerSize',3,'MarkerEdgeColor','r');
    plot(ref(:,2),ref(:,3));
    hold off;
    grid;
    axis([-15 +15 -15 +15]);
    axis square
    xlabel('y axis');
    ylabel('z axis');
    title(['Time=',num2str(i*0.01)])
end
figure(2)
subplot(321);
plot(t,x,'-g','Linewidth',2);
hold on
plot(t,ref(:,1),'-r','Linewidth',1);
hold off;
grid;
title('x(m) vs time(s)')
legend('Actual Measure','Reference')
subplot(322);
plot(t,(180/pi)*phi,'-g','Linewidth',2);
hold on;
plot(t,(180/pi)*phiC,'-r','Linewidth',1);
hold off;
grid;
title('roll (in deg) vs time(s)');
legend('Actual Measure','Reference')
subplot(323);
plot(t,y,'-g','Linewidth',2);
hold on
plot(t,ref(:,2),'-r','Linewidth',1);
hold off;
grid;
title('y(m) vs time(s)')
legend('Actual Measure','Reference')
subplot(324);
plot(t,(180/pi)*tht,'-g','Linewidth',2);
hold on;
plot(t,(180/pi)*thtC,'-r','Linewidth',1);
hold off;
grid;
title('pitch (in deg) vs time(s)');
legend('Actual Measure','Reference')
```

```
 subplot(325);
 plot(t,z,'-g','Linewidth',2);
 hold on
 plot(t,ref(:,3),'-r','Linewidth',1);
 hold off;
 grid;
 title('z(m) vs time(s)');
 legend('Actual Measure','Reference')
 subplot(326);
 plot(t,(180/pi)*psi,'-g','Linewidth',2);
 hold on;
 plot(t,ref(:,4),'-r','Linewidth',1);
 hold off;
 grid;
 title('yaw (in deg) vs time(s)');
 legend('Actual Measure','Reference')
 figure(3)
 subplot(411)
 plot(t,u1);
 grid
 title('Thrust (N) vs Time(s)');
 subplot(412)
 plot(t,u2(:,1));
 grid
 title('Rolling I/P(N m) vs Time(s)');
 subplot(413)
 plot(t,u2(:,2));
 grid
 title('Pitching I/P(N m) vs Time(s)');
 subplot(414)
 plot(t,u2(:,3));
 grid
 title('Yawing I/P(N m) vs Time(s)');


figure(4)


subplot(221);
 plot(t,ref(:,1)-x,'-r','Linewidth',2);
 grid;
 title('error in x(m) vs time(s)')
 subplot(222);
 plot(t,ref(:,2)-y,'-r','Linewidth',2);
 grid;
 title('error in y(m) vs time(s)')
 subplot(223);
 plot(t,ref(:,3)-z,'-r','Linewidth',2);
 grid;
 title('error in z(m) vs time(s)')
 subplot(224);
 plot(t,(ref(:,4)-psi)*180/pi,'-r','Linewidth',2);
 grid;
 title('error in psi(yaw)(deg) vs time(s)')
```

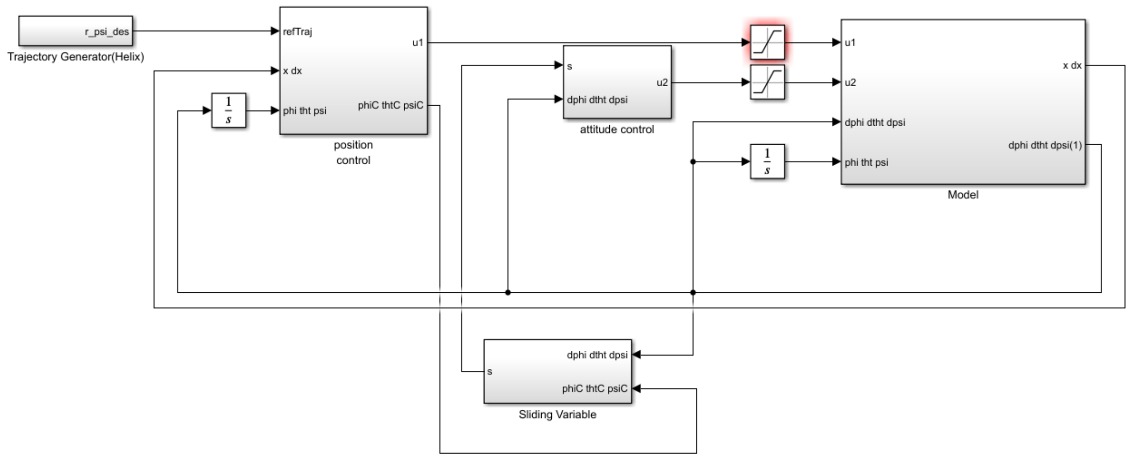
Simulink Implementation for PD Controller :



Simulink implementation for Backstepping control:



Simulink implementation for Sliding Mode Control:


r_psi_des
Trajectory Generator(Helix)
refTraj
x dx
phi tht psi
u1
phiC thtC psiC
position control
1/s
s
dphi dtht dpsi
u2
attitude control
u1
u2
dphi dtht dpsi
phi tht psi
x dx
dphi dtht dpsi(1)
Model
1/s
dphi dtht dpsi
s
phiC thtC psiC
Sliding Variable