Quadcopter Recovery Mission: Comparing MRAC Model-Based and NN Model-Free Adaptive Control

Qinwen Qian
Electrical Engineering

Kaifan Yue
Robotics

Angelo Hawa
Mechanical Engineering

Aleksandra Dudek
Mechanical Engineering

***Abstract*—Quadcopters have emerged as a useful technology for assisting in various applications from supply delivery to agriculture, which require precise control of the movement of the quadcopter. This precise movement must be preserved as the mass carried by the quadcopter varies due to the recovery and delivery of objects of varying mass. However, limitations in on-board computational power and the nonlinear dynamics of quadcopter motion present challenges in maintaining safe control of the quadcopter. Adaptive control provides a methodology wherein the control can be altered to account for variations and uncertainties in parameters, such as the mass carried. As such, this project turns to comparing two quadcopter adaptive control strategies - a model-based method (direct model reference adaptive control) and a model-free method (direct adaptive control using a Neural Network). Both methods are able to track the reference trajectory as desired, and the behavior of each is compared against a trajectory that an MPC contoller is able to track almost exactly, assuming know plant parameters. Advantages and challenges of both methods are further explored throughout the discussion of this work.**

I. Introduction

In the rapidly advancing field of unmanned aerial vehicle (UAV) technology, quadrotors have emerged as a fundamental innovation with many applications in both the professional and recreational domains. A quadrotor is a type of helicopter propelled by four electronic rotors that can be controlled independently. This design results in a highly agile device that can be used in a variety of settings.

Precise control of the orientation of the quadrotor can be achieved by varying the rotational velocity of each rotor. Typically, the rotors are arranged in a stable X-shaped configuration, and two pairs of rotors spin in opposite directions to counteract torque forces for better stability.

One primary application for quadrotors is search and rescue missions, which take advantage of their agility and widespread surveillance capabilities. This application is particularly important in regions that have been hit by natural disasters, where aerial transport may be the only practical mode of rescue. This process consists of assigning a transport task to the quadrotor, by which the quadrotor can both seek and deliver supplies to stranded individuals. However, supply transport can drastically alter the flight dynamics of the quadrotor, resulting in sharp alterations in the mass of the device and the perceived drag forces. Given the critical nature of these tasks, providing robust control of the flight of the quadcopter becomes indispensable.

Control strategies including Model Predictive Control (MPC) are able to compute the optimal input sequence for a dynamic system for a given horizon. A limitation in applying MPC on board the quadcopter is the limited computational power. Further, although MPC is robust to disturbances as it reoptimizes the input sequence following each control horizon, the significant change in mass could potentially destabilize the quadcopter yielding a state from which it could not recover.

Our team proposes the use of traditional adaptive control frameworks to ensure mission success under variable flight conditions. Specifically, we seek to incorporate both model-based and model-free control in a comparative analysis. Given the complex set of dynamics, many researchers have incorporated model simplification techniques such as linearization and reduced order modeling [2] to lower the computational burden of high fidelity simulations. Our team compared the performance of adaptive control using (1) Model Reference Adaptive Control with a simplified, linearized model and (2) Model-Free Adaptive Control with high-fidelity input/output data. We further consider the advantages and challenges associated with each method.

II. Methods and Procedure

A. Problem Statement

In this project we propose to analyze and compare the use of model-free and model-based adaptive control. This work considers a quadcopter that can be used to retrieve objects of a given mass. Adaptive control is an advantageous tool to enable the quadcopter to maintain stable control and flight that appropriately accounts for these variations in the system. The nonlinear model is further detailed in Section II-C.

However, it is difficult to construct or obtain an accurate mathematical model of the UAV especially in extreme environments. Rather than relying on the high-fidelity model under certain environment assumptions for model-based adaptive control, we linearize the model about an equilibrium point, specifically around hovering. We then further use the data from the simulated high-fidelity model to design a model-free adaptive control law. Here, we look to compare the performance of these two approaches and their ability to account for the varied mass throughout the trajectory of the quadcopter. These results provide insight to simplification methods that can be used in conditions of low online computational resources.

B. Designated Task

The application targeted in this study is the delivery of a life-saving object to a stranded individual. To this end, the

quadcopter's task is to move to a designated location from the origin, recover an object, transport the object to another location, deliver the object, and return to the origin. Object retrieval and delivery were modeled as a mass gain and reduction, respectively. The nominal path was defined as a interpolated trajectory of way points in 3D Cartesian coordinates, shown in Table I, where the points were to be reached at a specific point in time to allow for comparison between the different control strategies. The object is positioned at the the coordinate $(20, 0, 0)$, while the stranded individual is at $(20, 30, 0)$.

TABLE I
LIST OF WAY POINTS IN NOMINAL TRAJECTORY

Way Point	Coordinates [m] (x, y, z)	Time (s)
1	$(0, 0, 0)$	0
2	$(0, 0, 10)$	5.2
3	$(20, 0, 10)$	17
4	$(20, 0, 2)$	22
5	$(20, 0, 10)$	26
6	$(20, 30, 10)$	42
7	$(0, 0, 10)$	61
8	$(0, 0, 0)$	66.5

To incorporate a realistic life-threatening scenario, the object was chosen to be a pizza (Honolulu Hawaiian, large hand-tossed crust) from Domino's Pizza, Inc. The mass of the pizza is 0.65 kg, while the quadcopter's mass is 0.85 kg. A visual representation of the nominal trajectory is provided in Fig.1.

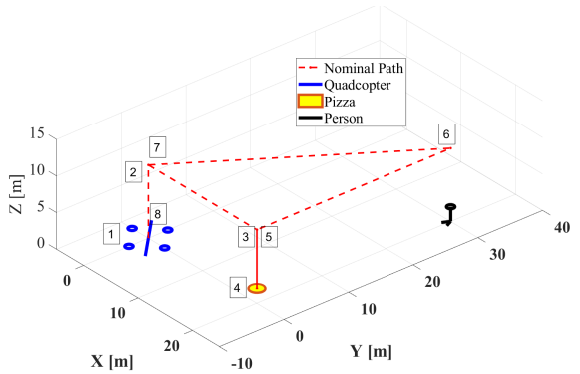


Fig. 1. Visual representation of the nominal trajectory with labeled way points and objects.

C. Reference Standard: Model Predictive Control

To provide a basis for performance comparison, model predictive control (MPC) was chosen as the reference standard. MPC is a highly utilized control framework in the aerospace field and offers high applicability in this specific task. The general dynamics of the quadcopter incorporate 12 states:

$$\boldsymbol{x} = \begin{bmatrix} x & y & z & \phi & \theta & \psi & \dot{x} & \dot{y} & \dot{z} & p & q & r \end{bmatrix}^T \tag{1}$$

where x, y, and z represent linear positions in their respective axes [m], ϕ, θ, and ψ represent the roll, pitch, and yaw angles [rad], respectively; $\dot{x}$, $\dot{y}$, $\dot{z}$, p, q, r represent the linear and angular velocities $[\frac{m}{s}, \frac{rad}{s}]$. The initial condition of the system ($\boldsymbol{x_0}$) was set to $0_{12\times 1}$.

The inputs of the system are expressed as follows:

$$\boldsymbol{u} = \begin{bmatrix} \omega_1^2 & \omega_2^2 & \omega_3^2 & \omega_4^2 \end{bmatrix} \tag{2}$$

where ω_i^2 represents the squared angular velocity $[(rad/s)^2]$ of the i^{th} rotor.

The complete dynamical model used in the MPC formulation can be found in [3], which uses the Euler-Lagrange equations to define the equations of motion considering the Jacobian, $\boldsymbol{J}$ which expresses the rotational energy in the inertial frame and the Coriolis Matrix, $\mathbf{C}(\eta, \dot{\eta})$, which accounts for gyroscopic and centripetal terms. In short, the equations of motion can be summarized in equations 3 and 4, where g is gravity, R is the rotation matrix between the body and intertial frames, and τ_B are the torques.

$$\ddot{\xi} = -\begin{bmatrix} 0 & 0 & g \end{bmatrix}^T + \frac{R}{m}\begin{bmatrix} 0 & 0 & T \end{bmatrix}^T \tag{3}$$

$$\ddot{\eta} = J - \tau_B - C(\eta, \dot{\eta})\dot{\eta} \tag{4}$$

The MPC cost function and constraints are provided in Eqs. 5 and 6.

$$\begin{aligned} \mathcal{J} = \sum_{j=1}^{N} \Big(& ||\boldsymbol{x}_k - \boldsymbol{x}_{ref,k}||_Q^2 + ||\boldsymbol{u}_k - \boldsymbol{u}_{target,k}||_R^2 \\ & + ||\Delta \boldsymbol{u}_{k+1} - \Delta \boldsymbol{u}_k||_H^2 \Big) \end{aligned} \tag{5}$$

$$\begin{aligned} \hat{\mathbf{u}}^* = \underset{\hat{\mathbf{u}}}{\text{argmin}} \quad & \mathcal{J}(\hat{\mathbf{x}}(\mathbf{k}), \hat{\mathbf{u}}(\mathbf{k})) \\ \text{subject to} \quad & \hat{\mathbf{x}}(k+1) = \hat{h}(\hat{\mathbf{x}}(k), \hat{\mathbf{u}}(k)) \quad \text{(model dynamics)} \\ & \hat{\mathbf{x}}(0) = \mathbf{x}(t) \quad \text{(initial condition)} \\ & \hat{\mathbf{x}}(k) \in \mathcal{X}(t) \quad \text{(state constraint)} \\ & \hat{\mathbf{u}}(k) \in \mathcal{U}(t) \quad \text{(input constraint)} \\ & \boldsymbol{\Delta}\hat{\mathbf{u}}(k) \in \Delta\mathcal{U}(t) \quad \text{(input rate constraint)} \end{aligned} \tag{6}$$

TABLE II
MPC PARAMETERS

Parameter	Value
Sample Time	$0.1s$
Prediction Horizon	18
Control Horizon	2
u_{min}	0
u_{max}	10
Δu_{min}	-2
Δu_{max}	2

The weights in the MPC cost function (Q, R, H) are defined as follows:

$$\underset{(12\times12)}{Q} = \begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ & & & & & 1 & & & & & & \\ & & & & & & 0 & & & & & \\ & & & & & & & \ddots & & & & \\ & & & & & & & & 0 & & & \\ & & & & & & & & & 0 & & \\ & & & & & & & & & & 0 & \\ & & & & & & & & & & & 0 \end{bmatrix}$$

$$\underset{(4\times4)}{R} = \begin{bmatrix} 0.1 & & & \\ & 0.1 & & \\ & & 0.1 & \\ & & & 0.1 \end{bmatrix}$$

$$\underset{(4\times4)}{H} = \begin{bmatrix} 0.1 & & & \\ & 0.1 & & \\ & & 0.1 & \\ & & & 0.1 \end{bmatrix}$$

The values are defined to prioritize state tracking of reference values for the linear and angular positions of the quadcopter, and to reduce the cost of control and change in control.

MATLAB's nonlinear MPC toolbox was used for all calculations, wherein the code used is provided separately in the Appendix.

D. Model-Based Control - MRAC

1) Equations of Motion: The general dynamics of the quadrotor are given as follows:

$$\ddot{x} = -\frac{T}{m}\left[\cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi)\right]$$

$$\ddot{y} = -\frac{T}{m}\left[\cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi)\right]$$

$$\ddot{z} = -\frac{T}{m}\cos(\phi)\cos(\theta) + g$$

$$\dot{p} = \frac{\tau_x + I_y qr - I_z qr}{I_x}$$

$$\dot{q} = \frac{\tau_y - I_x pr + I_z pr}{I_y}$$

$$\dot{r} = \frac{\tau_z + I_x pq - I_y pq}{I_z}$$

We introduced two new variables, Euler rates, Θ, and angular velocity, ω. Let $\Theta = [\phi, \theta, \psi]^T$ and let $w = [p, q, r]^T$, such that

$$\dot{\Theta} = W^{-1} w$$

where

$$W^{-1} = \frac{1}{\cos(\theta)}\begin{bmatrix} \cos(\theta) & \sin(\phi)\sin(\theta) & \cos(\phi)\sin(\theta) \\ 0 & \cos(\phi)\cos(\theta) & -\sin(\phi)\cos(\theta) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

With the above transformation function, we can express the Euler rates in terms of the angular velocities in the body frame.

$$\dot{\phi} = p + \frac{rcos(\phi)sin(\theta)}{cos(\theta)} + \frac{qsin(\theta)sin(\phi)}{cos(\theta)}$$

$$\dot{\theta} = qcos(\phi) - rsin(\phi)$$

$$\dot{\psi} = \frac{rcos(\phi)}{cos(\theta)} + \frac{qsin(\phi)}{cos(\theta)}$$

2) State-Space Representation: With the derivatives of the angular angles defined in Section II-D1, we can express the system dynamics in a state-space representation. Recalling the state vector x as defined in Section 1, the state-space representation can be expressed as follows:

$$\begin{bmatrix} \dot{x_1} & \dot{x_2} & \dot{x_3} \end{bmatrix}^T = \begin{bmatrix} x_7 & x_8 & x_9 \end{bmatrix}^T \tag{7}$$

$$\dot{x_4} = x_{10} + sin(x_4)tan(x_5)x_{11} + cos(x_4)tan(x_5)x_{12} \tag{8}$$

$$\dot{x_5} = cos(x_4)x_{11} - sin(x_4)x_{12} \tag{9}$$

$$\dot{x_6} = \frac{sin(x_4)}{cos(x_5)}x_{11} + \frac{cos(x_4)}{cos(x_5)}x_{12} \tag{10}$$

$$\dot{x_7} = -\frac{T}{m}[cos(x_4)sin(x_5)cos(x_6) + sin(x_4)sin(x_6)] \tag{11}$$

$$\dot{x_8} = -\frac{T}{m}[cos(x_4)sin(x_5)sin(x_6) - sin(x_4)cos(x_6)] \tag{12}$$

$$\dot{x_9} = -\frac{T}{m}cos(x_4)cos(x_5) + g \tag{13}$$

$$\dot{x_{10}} = \frac{\tau_x}{I_x} + \frac{I_y - I_z}{I_x}x_{11}x_{12} \tag{14}$$

$$\dot{x_{11}} = \frac{\tau_y}{I_y} + \frac{-I_x + I_z}{I_y}x_{10}x_{12} \tag{15}$$

$$\dot{x_{12}} = \frac{\tau_z}{I_z} + \frac{I_x - I_y}{I_z}x_{10}x_{12} \tag{16}$$

where $T, \tau_x, \tau_y, \tau_z$ are defines as the total thrust of the drone, torque about the x-axis, torque about the y-axis, and torque about the z-axis respectively, as defined below:

$$T = K_a(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \tag{17}$$

$$\tau_x = K_a l(\omega_4^2 - \omega_2^2)$$

$$\tau_y = K_a l(\omega_1^2 - \omega_3^2)$$

$$\tau_z = K_m(\omega_1^2 - \omega_2^2 + \omega_3^3 - \omega_4^2)$$

The constants K_a and K_m are aerodynamic values used to calculate thrust and torque exerted by each rotor, and are assumed to be known.

3) Linearization about hovering: The dynamic model of the quadrotor system is highly nonlinear and complex. To simplify the implementation and reduce computation power, we linearize the equation of motion at the hovering condition, such that derivative of state vector, $\dot{x} = 0$ and all four rotational speeds of the rotors are equal to each other.

$$X_e = \begin{bmatrix} x_e & y_e & z_e & 0 & 0 & \psi_e & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (18)$$

$$U_e = \begin{bmatrix} \omega_e^2 & \omega_e^2 & \omega_e^2 & \omega_e^2 \end{bmatrix}^T \quad (19)$$

However, it is important to recognize that there is a non-zero yaw value at the hovering equilibrium which is meant to cancel the gravity of the quadrotor $F = mg$. Considering the thrust defined in 17, the rotational speed of the rotor is

$$\omega_{eqm} = \sqrt{\frac{mg}{4k_a}}$$

.

We take the Jacobian matrices at the equilibrium to get the linear representation of the system, $\Delta\dot{x} = A\Delta x + B\Delta u$ with

$$A = \frac{\partial}{\partial x} f(x,u)|_{(x_{eqm}, u_{eqm})}$$

$$B = \frac{\partial}{\partial u} f(x,u)|_{(x_{eqm}, u_{eqm})}$$

With the equations of motions and the Jacobian representation, we find A and B to be:

$$A = \begin{bmatrix} 0_{6\times6} & I_{6\times6} \\ \Lambda & 0_{6\times6} \end{bmatrix} \quad (20)$$

$$B = \begin{bmatrix} 0_{8\times4} \\ \Delta \end{bmatrix} \quad (21)$$

where

$$\Lambda = \begin{bmatrix} 0 & 0 & 0 & gsin(\psi_{eqm}) & gcos(\psi_{eqm}) & 0 \\ 0 & 0 & 0 & -gcos(\psi_{eqm}) & gsin(\psi_{eqm}) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Delta = \begin{bmatrix} \frac{K_a}{m} & \frac{K_a}{m} & \frac{K_a}{m} & \frac{K_a}{m} \\ 0 & -\frac{K_a l}{I_x} & 0 & \frac{K_a l}{I_x} \\ \frac{K_a l}{I_y} & 0 & -\frac{K_a l}{I_y} & 0 \\ \frac{K_m}{I_z} & -\frac{K_m}{I_z} & \frac{K_m}{I_z} & -\frac{K_m}{I_z} \end{bmatrix}$$

4) MRC using Pole-placement: By setting $\Delta u = -K\Delta x$, the closed-loop dynamics will become to $\Delta\dot{x} = (A - BK)\Delta x$. To have a stable closed-loop system, the closed-loop poles must be in the left-half plane. One of the ways to achieve this is using MATLAB code, `place`, which effectively places these poles in the stable region.

5) MRAC Design: For simpler notation, we define Δx and Δu as x and u. The reference model is built using the drone parameter defined in Section II-B and the linearized system equations defined in Eqs. 20 and 21. The estimated plant is designed using mass with 10% error ($A_{estimated}$ is independent of mass, so only $B_{estimated}$ is influenced by the change in the mass).

As matrix A and B are known, we apply the control law, $u = -K_x x + K_r r$ and obtain the closed-loop plant dynamics $\dot{x} = (A - BK_x^*)x + BK_r r$. Hence $A - BK_r^* = A_m$ and $BK_r^* = B_m$. [2]

The adaptation laws are:

$$\dot{K}_x(t) = \Gamma_x x(t) e^T(t) PB \quad (22)$$

$$\dot{K}_r(t) = -\Gamma_r r(t) e^T(t) PB \quad (23)$$

where $e(t)$ is the error defined as $e(t) = x(t) - x_m(t)$, P is the unique solution of the Riccati Equation which is positive definite, and Γ is the adaptation gain matrix. We choose initial conditions of $K_x(0)$ and $K_r(0)$ close to the equilibrium for stability.

6) Robustness: Further robustness could be implemented in the MRAC model-based control architecture to account for unmodeled dynamics and uncertainties. This would best be implemented by impacting the adaptive laws, either by including a leakage term to avoid pure integration or through normalization. This would further improve the behavior of the adaptively controlled quadcopter.

E. Model-Free Control - Neural Network

The model-free control used in this project is based on that described in [4]. In this work, the authors utilized a direct adaptive control scheme in which they employ a trained neural network (NN) to generate control outputs that account for variation in the plant and unexpected disturbances.

The controller structure used is composed of a hierarchical structure, wherein the high-level controller accounts for trajectory planning by determining the desired thrust and angular velocities, whereas the low-level controller translates these into specific motor commands for input into the quadcopter motors. This hierarchical structure can be seen in Fig. 2.

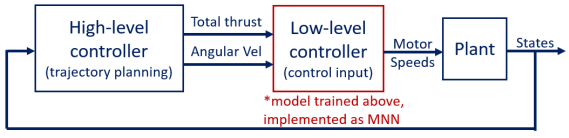


Fig. 2. Hierarchical control structure composed of high-level trajectory planning and low-level motor commands.

The training of this low-level controller employs a dual strategy: imitation learning from model-based controller and reinforcement learning, as visualized in Fig. 3 and further detailed in [4].

This trained controller results in a NN implemented as a 3 layer multi-layer perceptron (MLP) with 356 nodes per

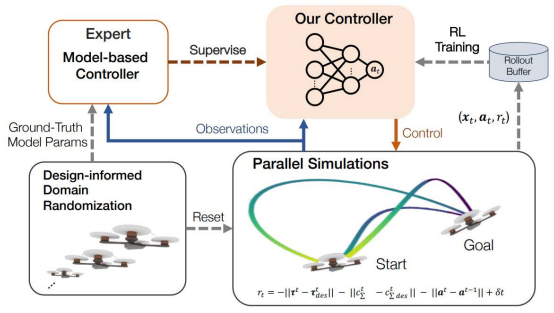


Fig. 3. Training process of adaptive controller which employs a dual strategy of reinforcement learning and supervision through model-based imitation learning (image taken from source)[4].

layer. The input into this NN is a 34 dimensional vector that includes the states that are fed into the model-based controller (i.e. attitude, angular velocities), as well as the thrusts, drone specifications including mass, mass moment of inertia (MMOI), drag coefficient, etc. These non-state inputs are referred to as environmental factors by the authors and are further accounted for by another 2 layer MLP with 128 nodes per layer. Lastly, to account for the temporaral correlation in these inputs, a 3 layer, 1 dimensional convolutional NN is used. The training for this NN is done using simulation paths such as those below in Fig.4, accompanied by a wide range of quadcopter specifications and environmental factors.

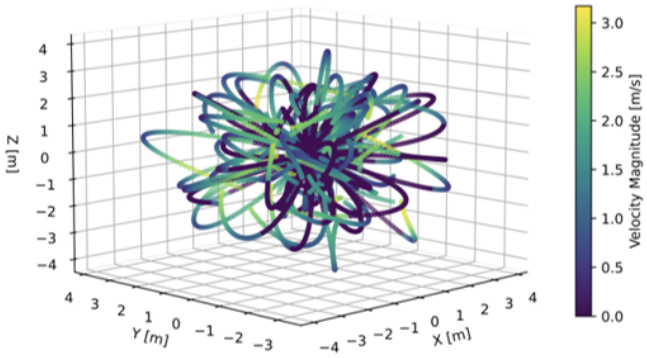


Fig. 4. Simulation training paths for NN (image taken from source) [4].

To adapt this source code for our own application, we first encode the nominal trajectory described in I into a series of time-indexed waypoints based on a specified flight speed. Then, the waypoints are interpolated based on the simulation time step to ensure smooth transitions in position, velocity, and acceleration. A position controller computes the required acceleration commands to minimize the error between the desired and current positions. These commands are translated into thrust and angular velocity demands by an attitude controller. During the simulation, we iteratively generate the control inputs between consecutive waypoints and update the quadcopter's states through its physical model.

To simulate the pickup and dropoff of the pizza, the mass of the quadcopter is varied at the desired simulation time steps. This change affects the dynamics of the quadcopter, challenging the controllers to adapt to the altered system inertia and thrust requirements to maintain trajectory adherence.

III. Results

The simulation records the quadcopter's state, including position, velocity, attitude, and motor forces, throughout the flight. The position is compared against the reference trajectory and trajectory generated by MPC, which are visualized in Fig.5. The NN model-free controller is able to track the reference path with relatively good accuracy. However, there are more significant deviations and overshoots when approaching and leaving the goal positions, such as the x position goal of 20 m around 17 seconds and the drop-off z position goal around 22 seconds. The reason is that the controller generates aggressive input commands during acceleration and deceleration, causing extreme roll and pitch angles. The effect of carrying the payload is most pronounced on the Z axis. It can be observed that, at 22 seconds, the quadcopter descends abruptly, reacting to the increased weight, and then struggles to elevate at the reference rate during due to the higher load on the motors.

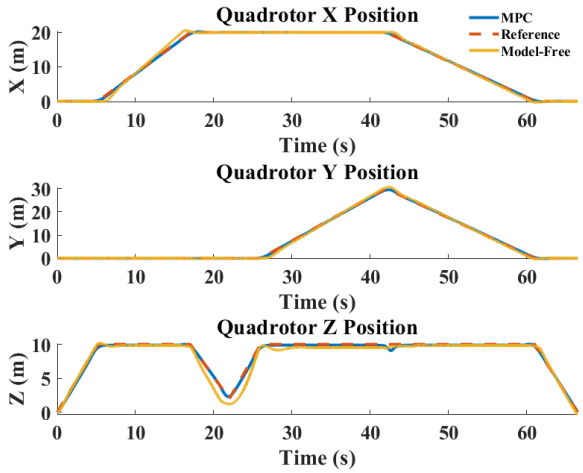


Fig. 5. Flight trajectories using the MPC controller and the NN model-free controller.

Note that with the NN model-free approach, the payload mass is outside of the range of the training parameters, thus challenging the NN model to extrapolate to find the optimal input commands. The steady state Z position after picking up the payload (30 seconds) is slightly below the reference position which is either due to a low integral gain in the high-level position controller or reaching the motor limit. However, the undershoot is mitigated once the payload is dropped off at 43 seconds. Overall, the NN-based model-free controller has proved to be robust to unforeseen system changes with slightly compromised performance compared to the standard MPC.

For the MRAC approach, we planned the trajectory for each pair of way points. The performance of the MRAC controller

with linearized system dynamics is shown in , where the dotted red line is the reference system's planned trajectory and the blue line is the adaptive model reference controller's trajectory. Due to the overall scale, the performance of the MRAC controller did fairly good despite of one small deviation of the output trajectory around 5 seconds along the direction of y. The estimated system can track the reference system quite well. However, there are some significant undershoot at the pizza drop-off position and the altitude of the drone around 20 seconds turns out to be at below the designed horizon as shown in Fig. 7.

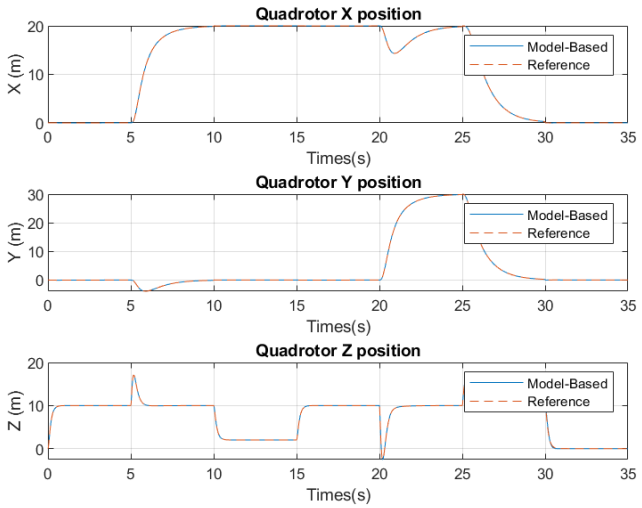


Fig. 6. Flight trajectories using the MRAC controller with linearized plant dynamics.

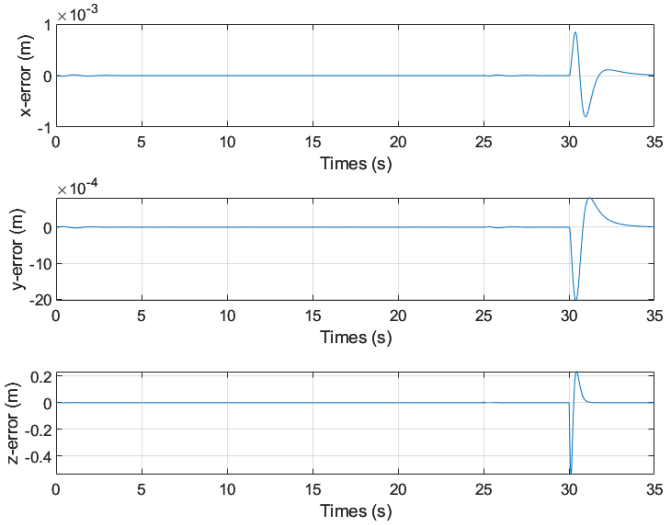


Fig. 7. Error of position of the drone between estimated plant and reference plant.

A limitation in comparing the results between the control strategies used is the inherent differences in the control structures. For the NN-based approach, it is important to recognize

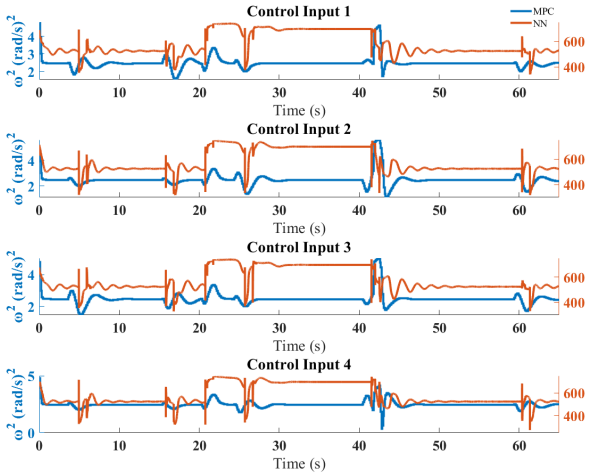


Fig. 8. Control inputs compared between MPC and NN model-free adaptive control.

that the hyperparameters that define the NN, including the number of layers and nodes greatly impact the performance of the NN in determining the inputs to send to the system.

Further, the MRAC approach depends heavily on how the linearized system is defined. For simpler implementation, we defined the state to control altitude and position at the same time. However, in commonly used linearized drone dynamics, the altitude control and rotational velocity control are often separated for better altitude control. Since the way we defined the MRAC control is to track the trajectory between two way points in a time span of 5 seconds, it is also difficult to compare the total time taken for the move. Although with linearized model dynamics it still takes long time to compute the system dynamics.

Further, as seen below in Fig.8, the neural network did not assume the same scale of inputs as the MPC model. The referenced MPC model set an upper bound on the inputs at $10(rad/s)^2$, whereas the magnitude of the NN-based control inputs was on the scale of 10^2. This model mismatch further challenges the direct comparison between these control strategies.

Tables III and IV below quantify the maximum and mean errors between the reference trajectory, MPC results and the NN model-free results.

TABLE III
MPC AND NN MODEL-FREE MAX ERROR IN POSITION

Dimension	MPC [m]	NN model-free [m]
x	0.25	1.80
y	0.48	1.56
z	0.33	0.63

TABLE IV
MPC AND NN MODEL-FREE MEAN ERROR IN POSITION

Dimension	MPC [m]	NN model-free [m]
x	0.0008	0.033
y	0.0022	0.012
z	0.13	0.40

IV. DISCUSSION

More improvements of MRAC can be made to let the drone start the next tracking way points' pair if the drone is already in the desired position. We assume zero external disturbances, *i.e.* wind, when building our model. For MRAC approach, the linearized drone dynamics of the quadrotor will be impacted and need to be recomputed if there exists wind. More system dynamics computation can be found in [5]. If the disturbance is unknown, we can incorporate an extra parameter estimator in the MRAC system.

To further explore the robustness of the NN model-free controller, we modeled a wind disturbance to observe the ability of the system to maintain tracking. In doing so, we saw no impact to the reference tracking ability of the quadcopter. However, to ensure that the disturbance was impacting the model, we compared the control inputs, as seen below in Fig.9.

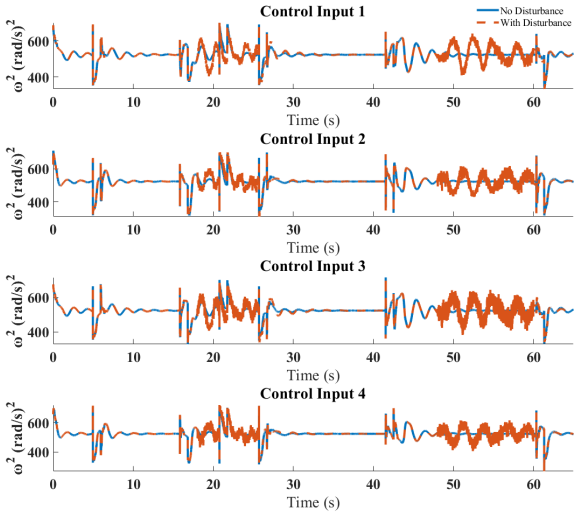


Fig. 9. Model-free NN control input with and without disturbance.

These results suggest that not only is the model-free approach able to adapt to unknown plant parameters, such as the mass considered throughout this project, but also to disturbances and unknown environmental parameters.

V. PEER FEEDBACK

- Feedback 1: Results are not well organized.
- *To address this, we reorganized our report to instead introduce the control approaches first before discussing the results. For results where direct comparisons were possible, we included the trajectories and control inputs on the same plots.*
- Feedback 2: How much the added mass can drive the system to unstable?
- *The added mass of 0.65 kg relative to the 0.85 kg drone already surpassed the reported limitations in the referenced NN model-free control (35.7% of the drone mass) [4] without destabilizing the system. Although we were unable to find a more concrete amount, the ability of the system to handle disturbances this large reflects the value in using adaptive control strategies.*
- Feedback 3: Better metrics comparing their performances would be extremely helpful in your final report.
- *Feedback 1 somewhat addresses this through more clear comparison of the performance. Further, reporting the position error values relative to the reference trajectory provide perspective on the tolerance in path deviation that would have to be allowable to employ these adaptive controllers.*
- Feedback 4: Can you compare how much computation is needed for each method, for example, how much time is used for simulation for each method?
- *A challenge in comparing the computation time between the methods is that the methods are implemented in MATLAB and Python, suggesting that timings might be a poor metric to compare the computational demand [1].*

VI. CONCLUSION

This work assesses the viability of adaptive control frameworks in a quadcopter object delivery mission. To account for scenarios with limited online processing capabilities, two adaptive control approaches were implemented. The first leverages linearized model dynamics within an MRAC framework, while the second incorporates a neural network to provide model-free computations. A third approach using MPC was derived in order to provide reference standard performance. The results of the study indicate that both adaptive approaches provide adequate state tracking across the trajectory of the quadcopter. All models exhibit slight oscillation at the onset of the mass change; however, stability is preserved through the entire trajectory.

Future work will focus on creating randomized disturbances to the system in the form of wind and sensor noise to observe model performance. Additional efforts will incorporate experimental validation of the proposed claims.

ACKNOWLEDGMENTS

We would like to acknowledge Professor Sun for her clear explanation of course material and prioritization of learning.

REFERENCES

[1] Steve Hanley. Matlab vs python: Speed test for vibration analysis. URL https://blog.endaq.com/matlab-vs-python-speed-for-vibration-analysis-free-download#:~:text=Check%20out%20a%20log%2Dlog,an%20order%20of%20magnitude%20faster!

[2] P. A. Ioannou and J. Sun. *Robust Adaptive Control*. Dover Books on Electrical Engineering Series. Dover Publications, Incorporated, 2012. ISBN 9780486498171. URL https://books.google.com/books?id=pXWFY_vbg1MC.

[3] Teppo Luukkonen. Modelling and control of quadcopter, 2011. Independent research project in applied mathematics, Aalto University.

[4] Dingqi Zhang, Antonio Loquercio, Jerry Tang, Ting-Hao Wang, Jitendra Malik, and Mark W. Mueller. A learning-based quadcopter controller with extreme adaptation, 2024. URL https://arxiv.org/abs/2409.12949.

[5] Y. Zhang, L. Wen, and B. Jiang. Analysis of dynamic mutation of a quadrotor system under wind disturbance. In *2023 6th International Symposium on Autonomous Systems (ISAS)*, pages 1–6, June 2023. doi: 10.1109/ISAS59543.2023.10164497.

```
function [A,B] = QuadrotorStateJacobianFcnm2(in1,in2)
%QuadrotorStateJacobianFcn
%    [A,B] = QuadrotorStateJacobianFcn(IN1,IN2)

%    This function was generated by the Symbolic Math Toolbox version 24.2.
%    08-Dec-2024 16:53:04

phi_t = in1(4,:);
phi_dot_t = in1(10,:);
psi_t = in1(6,:);
psi_dot_t = in1(12,:);
theta_t = in1(5,:);
theta_dot_t = in1(11,:);
u1 = in2(1,:);
u2 = in2(2,:);
u3 = in2(3,:);
u4 = in2(4,:);
t2 = cos(phi_t);
t3 = cos(psi_t);
t4 = cos(theta_t);
t5 = sin(phi_t);
t6 = sin(psi_t);
t7 = sin(theta_t);
t8 = phi_t.*2.0;
t9 = psi_dot_t.^2;
t10 = theta_t.*2.0;
t11 = theta_dot_t.^2;
t21 = u1+u2+u3+u4;
t12 = cos(t8);
t13 = t2.^2;
t14 = cos(t10);
t15 = t4.^2;
t16 = t4.^3;
t17 = sin(t8);
t18 = t5.^2;
t19 = sin(t10);
t20 = t7.^2;
t22 = t4.*6.0e+1;
t23 = 1.0./t4;
t26 = t7.*9.2e+1;
t27 = t7.*1.15e+2;
t32 = (t2.*t4)./2.0;
t33 = (t3.*t5)./2.0;
t34 = (t5.*t6)./2.0;
t41 = t2.*t4.*t5.*5.5e+1;
t42 = t2.*t5.*t7.*5.5e+1;
t47 = (t2.*t3.*t7)./2.0;
t50 = (t2.*t6.*t7)./2.0;
t24 = 1.0./t15;
t25 = 1.0./t16;
t28 = -t26;
t29 = t13.*4.4e+1;
```

```
t30 = t13.*5.5e+1;
t31 = t17.*2.2e+1;
t37 = -t33;
t43 = t7.*t13.*-4.4e+1;
t44 = t7.*t13.*-5.5e+1;
t52 = t7.*t41;
t53 = t4.*t13.*theta_dot_t.*5.06e+2;
t54 = t4.*t13.*u3.*-5.5e+1;
t59 = phi_dot_t.*psi_dot_t.*t13.*t15.*5.06e+2;
t60 = psi_dot_t.*t4.*t7.*t13.*theta_dot_t.*-5.06e+2;
t62 = t7.*t9.*t13.*t15.*5.06e+2;
t63 = t34+t47;
t35 = -t29;
t36 = -t30;
t38 = t4.*t30;
t39 = t7.*t29;
t40 = t7.*t30;
t51 = t15.*t30;
t55 = t44.*u4;
t56 = phi_dot_t.*t53;
t57 = t7.*t53;
t61 = -t59;
t64 = t37+t50;
t45 = t38.*u1;
t46 = t38.*u3;
t48 = t40.*u2;
et1 = (t24.*(t4.*u1.*-9.2e+1+t4.*u2.*9.2e+1-
t4.*u3.*9.2e+1+t4.*u4.*9.2e+1+phi_dot_t.*t14.*theta_dot_t.*4.6e+1-
psi_dot_t.*t7.*theta_dot_t.*1.058e+3-t4.*t13.*u2.*4.4e+1-
t4.*t13.*u4.*4.4e+1+t4.*t29.*u1+t4.*t29.*u3+phi_dot_t.*t13.*t15.*theta_dot_t.
*5.06e+2-
phi_dot_t.*t13.*t20.*theta_dot_t.*5.06e+2+psi_dot_t.*t7.*t13.*theta_dot_t.*5.
06e+2-
t2.*t5.*t15.*u1.*5.5e+1+t2.*t5.*t15.*u3.*5.5e+1+t4.*t7.*t13.*u2.*1.1e+2+t2.*t
5.*t20.*u1.*5.5e+1-t4.*t7.*t13.*u4.*1.1e+2-
t2.*t5.*t20.*u3.*5.5e+1+phi_dot_t.*psi_dot_t.*t2.*t5.*t16.*5.06e+2+psi_dot_t.
*t7.*t13.*t15.*theta_dot_t.*1.518e+3+t2.*t4.*t5.*t7.*t9.*1.012e+3-
t2.*t4.*t5.*t7.*t11.*1.012e+3-
phi_dot_t.*psi_dot_t.*t2.*t4.*t5.*t20.*1.012e+3))./5.52e+2;
et2 = (t7.*t25.*(u2.*-1.15e+2+u4.*1.15e+2+t7.*t56-t7.*u1.*9.2e+1-
t7.*u3.*9.2e+1-
t13.*u4.*5.5e+1+t26.*u2+t26.*u4+t30.*u2+t39.*u1+t39.*u3+t43.*u2+t43.*u4+t51.*
u4+t52.*u3+phi_dot_t.*t19.*theta_dot_t.*2.3e+1+psi_dot_t.*t4.*theta_dot_t.*1.
058e+3-t13.*t15.*u2.*5.5e+1-psi_dot_t.*t4.*t13.*theta_dot_t.*5.06e+2-
psi_dot_t.*t13.*t16.*theta_dot_t.*5.06e+2-
t2.*t5.*t9.*t15.*5.06e+2+t2.*t5.*t11.*t15.*5.06e+2-
t2.*t4.*t5.*t7.*u1.*5.5e+1+phi_dot_t.*psi_dot_t.*t2.*t5.*t7.*t15.*5.06e+2))./
2.76e+2;
et3 = t24.*(t4.*u2.*1.15e+2-
t4.*u4.*1.15e+2+t38.*u4+t42.*u3+phi_dot_t.*t7.*theta_dot_t.*4.6e+1-
psi_dot_t.*t14.*theta_dot_t.*1.058e+3-
t4.*t13.*u2.*5.5e+1+phi_dot_t.*t7.*t13.*theta_dot_t.*5.06e+2+psi_dot_t.*t13.*
t15.*theta_dot_t.*5.06e+2-
psi_dot_t.*t13.*t20.*theta_dot_t.*5.06e+2+t2.*t5.*t9.*t16.*5.06e+2-
```

```
t2.*t5.*t7.*u1.*5.5e+1-
t2.*t4.*t5.*t9.*t20.*1.012e+3+phi_dot_t.*psi_dot_t.*t2.*t4.*t5.*t7.*1.012e+3)
.*(-1.0./5.52e+2);
et4 = (t7.*t25.*(t48+t55+t56+t60-u1.*9.2e+1+u2.*9.2e+1-u3.*9.2e+1+u4.*9.2e+1-
t7.*u2.*1.15e+2-t13.*u2.*4.4e+1-
t13.*u4.*4.4e+1+t29.*u1+t27.*u4+t29.*u3+t41.*u3+phi_dot_t.*t4.*theta_dot_t.*4
.6e+1+psi_dot_t.*t19.*theta_dot_t.*5.29e+2-
t2.*t4.*t5.*u1.*5.5e+1+phi_dot_t.*psi_dot_t.*t2.*t5.*t15.*5.06e+2-
t2.*t5.*t7.*t9.*t15.*5.06e+2))./2.76e+2;
mt1 =
[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0,0.0,(t21.*(t2.*t6-t3.*t5.*t7))./
2.0,t21.*(t2.*t3+t5.*t6.*t7).*(-1.0./2.0),t4.*t5.*t21.*(-1.0./2.0)];
mt2 = [(t24.*(t7.*t46+t7.*t59-
t9.*t13.*t15.*5.06e+2+t11.*t13.*t15.*5.06e+2+t9.*t15.*t18.*5.06e+2-
t11.*t15.*t18.*5.06e+2-t2.*t5.*u2.*1.1e+2+t2.*t5.*u4.*1.1e+2+t4.*t44.*u1-
t2.*t5.*t7.*u1.*8.8e+1+t2.*t5.*t7.*u2.*8.8e+1-
t2.*t5.*t7.*u3.*8.8e+1+t2.*t5.*t7.*u4.*8.8e+1+t2.*t5.*t15.*u2.*1.1e+2-
t2.*t5.*t15.*u4.*1.1e+2+t4.*t7.*t18.*u1.*5.5e+1-t4.*t7.*t18.*u3.*5.5e+1-
phi_dot_t.*psi_dot_t.*t7.*t15.*t18.*5.06e+2+psi_dot_t.*t2.*t4.*t5.*theta_dot_
t.*1.012e+3+psi_dot_t.*t2.*t5.*t16.*theta_dot_t.*1.012e+3-
phi_dot_t.*t2.*t4.*t5.*t7.*theta_dot_t.*1.012e+3))./5.52e+2];
mt3 = [t23.*(t48+t55+t56+t60+t12.*u1.*4.4e+1-t12.*u2.*4.4e+1+t12.*u3.*4.4e+1-
t12.*u4.*4.4e+1-t7.*t18.*u2.*5.5e+1+t7.*t18.*u4.*5.5e+1-
phi_dot_t.*t4.*t18.*theta_dot_t.*5.06e+2-
t2.*t4.*t5.*u1.*1.1e+2+t2.*t4.*t5.*u3.*1.1e+2+phi_dot_t.*psi_dot_t.*t2.*t5.*t
15.*1.012e+3+psi_dot_t.*t4.*t7.*t18.*theta_dot_t.*5.06e+2-
t2.*t5.*t7.*t9.*t15.*1.012e+3).*(-1.0./5.52e+2)];
mt4 = [t24.*(t45+t54+t61+t62+t2.*t5.*u1.*8.8e+1-
t2.*t5.*u2.*8.8e+1+t2.*t5.*u3.*8.8e+1-t2.*t5.*u4.*8.8e+1-
t4.*t18.*u1.*5.5e+1+t4.*t18.*u3.*5.5e+1+phi_dot_t.*psi_dot_t.*t15.*t18.*5.06e
+2-t7.*t9.*t15.*t18.*5.06e+2+t2.*t5.*t7.*u2.*1.1e+2-
t2.*t5.*t7.*u4.*1.1e+2+phi_dot_t.*t2.*t4.*t5.*theta_dot_t.*1.012e+3-
psi_dot_t.*t2.*t4.*t5.*t7.*theta_dot_t.*1.012e+3).*(-1.0./5.52e+2),0.0,0.0,0.
0,0.0,0.0,0.0,t3.*t21.*t32,t6.*t21.*t32,t2.*t7.*t21.*(-1.0./2.0),et1+et2];
mt5 = [(t23.*(t7.*u1.*6.0e+1-t7.*u3.*6.0e+1+t40.*u1+t41.*u4+t44.*u3-
t9.*t13.*t16.*5.06e+2+phi_dot_t.*psi_dot_t.*t4.*t7.*1.104e+3+t4.*t9.*t13.*t20
.*1.012e+3-t2.*t4.*t5.*u2.*5.5e+1-
phi_dot_t.*psi_dot_t.*t4.*t7.*t13.*1.012e+3+phi_dot_t.*t2.*t5.*t7.*theta_dot_
t.*5.06e+2+psi_dot_t.*t2.*t5.*t15.*theta_dot_t.*5.06e+2-
psi_dot_t.*t2.*t5.*t20.*theta_dot_t.*5.06e+2))./5.52e+2-
(t7.*t24.*(t45+t54+t61+t62-t4.*u3.*6.0e+1-t17.*u2.*2.2e+1-
t17.*u4.*2.2e+1+t22.*u1+t31.*u1+t31.*u3+t42.*u2+phi_dot_t.*psi_dot_t.*t15.*5.
52e+2-t2.*t5.*t7.*u4.*5.5e+1+phi_dot_t.*t2.*t4.*t5.*theta_dot_t.*5.06e+2-
psi_dot_t.*t2.*t4.*t5.*t7.*theta_dot_t.*5.06e+2))./
5.52e+2,et3+et4,0.0,0.0,0.0,0.0,0.0,0.0];
mt6 = [(t21.*(t3.*t5-t2.*t6.*t7))./2.0,(t21.*(t5.*t6+t2.*t3.*t7))./
2.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.
0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,
(t24.*(t57+t19.*theta_dot_t.*2.3e+1+psi_dot_t.*t2.*t5.*t7.*t15.*5.06e+2))./
5.52e+2,t23.*(psi_dot_t.*t15.*5.52e+2-
psi_dot_t.*t13.*t15.*5.06e+2+t2.*t4.*t5.*theta_dot_t.*5.06e+2).*(-1.0./5.52e+
```

```
2),(t24.*(t53+t4.*theta_dot_t.*4.6e+1+psi_dot_t.*t2.*t5.*t15.*5.06e+2))./
5.52e+2,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0];
mt7 = [(t24.*(phi_dot_t.*t19.*2.3e+1+psi_dot_t.*t4.*1.058e+3-
psi_dot_t.*t4.*t13.*5.06e+2-
psi_dot_t.*t13.*t16.*5.06e+2+phi_dot_t.*t4.*t7.*t13.*5.06e+2+t2.*t5.*t15.*the
ta_dot_t.*1.012e+3))./5.52e+2,t23.*(phi_dot_t.*t2.*t4.*t5.*5.06e+2-
psi_dot_t.*t2.*t4.*t5.*t7.*5.06e+2).*(-1.0./5.52e+2),
(t24.*(phi_dot_t.*t4.*4.6e+1+psi_dot_t.*t19.*5.29e+2+phi_dot_t.*t4.*t13.*5.06
e+2-psi_dot_t.*t4.*t7.*t13.*5.06e+2))./
5.52e+2,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,t24.*(t53-
t4.*theta_dot_t.*1.058e+3+t13.*t16.*theta_dot_t.*5.06e+2+psi_dot_t.*t2.*t5.*t
15.*1.012e+3-phi_dot_t.*t2.*t5.*t7.*t15.*5.06e+2).*(-1.0./5.52e+2)];
mt8 = [t23.*(phi_dot_t.*t15.*5.52e+2-
phi_dot_t.*t13.*t15.*5.06e+2+psi_dot_t.*t7.*t13.*t15.*1.012e+3-
t2.*t4.*t5.*t7.*theta_dot_t.*5.06e+2).*(-1.0./5.52e+2),
(t24.*(t19.*theta_dot_t.*5.29e+2+phi_dot_t.*t2.*t5.*t15.*5.06e+2-
t4.*t7.*t13.*theta_dot_t.*5.06e+2-psi_dot_t.*t2.*t5.*t7.*t15.*1.012e+3))./
5.52e+2];
A = reshape([mt1,mt2,mt3,mt4,mt5,mt6,mt7,mt8],12,12);
if nargout > 1
    mt9 =
[0.0,0.0,0.0,0.0,0.0,0.0,t63,t64,t32,t24.*(t26+t43+t52).*(-1.0./5.52e+2),t23.
*(t22+t31+t38).*(-1.0./5.52e+2),t24.*(t35+t41+9.2e+1).*(-1.0./5.52e+2),0.0,0.
0,0.0,0.0,0.0,0.0,t63,t64,t32,t24.*(t28+t36+t39+t51+1.15e+2).*(-1.0./5.52e+2)
,(t23.*(t31-t42))./
5.52e+2,t24.*(t27+t29+t44-9.2e+1).*(-1.0./5.52e+2),0.0,0.0,0.0,0.0,0.0,0.0,t6
3,t64,t32,(t24.*(t28+t39+t52))./5.52e+2,(t23.*(t22-t31+t38))./5.52e+2,
(t24.*(t29+t41-9.2e+1))./5.52e+2,0.0,0.0,0.0,0.0,0.0,0.0,t63,t64,t32,
(t24.*(t26+t36+t43+t51+1.15e+2))./5.52e+2,(t23.*(t31+t42))./5.52e+2];
    mt10 = [(t24.*(t27+t35+t44+9.2e+1))./5.52e+2];
    B = reshape([mt9,mt10],12,4);
end
end

Not enough input arguments.

Error in QuadrotorStateJacobianFcnm2 (line 8)
phi_t = in1(4,:);
        ^^^^^^^^
```

Published with MATLAB® R2024b

```
function f = QuadrotorStateFcnm1(in1,in2)
%QuadrotorStateFcn
%    F = QuadrotorStateFcn(IN1,IN2)

%    This function was generated by the Symbolic Math Toolbox version 24.2.
%    08-Dec-2024 16:52:49

phi_t = in1(4,:);
phi_dot_t = in1(10,:);
psi_t = in1(6,:);
psi_dot_t = in1(12,:);
theta_t = in1(5,:);
theta_dot_t = in1(11,:);
u1 = in2(1,:);
u2 = in2(2,:);
u3 = in2(3,:);
u4 = in2(4,:);
x_dot_t = in1(7,:);
y_dot_t = in1(8,:);
z_dot_t = in1(9,:);
t2 = cos(phi_t);
t3 = cos(psi_t);
t4 = cos(theta_t);
t5 = sin(phi_t);
t6 = sin(psi_t);
t7 = sin(theta_t);
t8 = phi_t.*2.0;
t9 = psi_dot_t.^2;
t10 = theta_t.*2.0;
t15 = u1+u2+u3+u4;
t11 = t2.^2;
t12 = t4.^2;
t13 = sin(t8);
t14 = sin(t10);
t16 = 1.0./t12;
mt1 =
[x_dot_t;y_dot_t;z_dot_t;phi_dot_t;theta_dot_t;psi_dot_t;t15.*(t5.*t6+t2.*t3.
*t7);-t15.*(t3.*t5-t2.*t6.*t7);t2.*t4.*t15-9.81e+2./1.0e+2];
mt2 = [(t16.*(u2.*-1.15e+2+u4.*1.15e+2-t7.*u1.*9.2e+1+t7.*u2.*9.2e+1-
t7.*u3.*9.2e+1+t7.*u4.*9.2e+1+t11.*u2.*5.5e+1-
t11.*u4.*5.5e+1+phi_dot_t.*t14.*theta_dot_t.*2.3e+1+psi_dot_t.*t4.*theta_dot_
t.*1.058e+3+t7.*t11.*u1.*4.4e+1-t7.*t11.*u2.*4.4e+1+t7.*t11.*u3.*4.4e+1-
t7.*t11.*u4.*4.4e+1-t11.*t12.*u2.*5.5e+1+t11.*t12.*u4.*5.5e+1-
psi_dot_t.*t4.*t11.*theta_dot_t.*5.06e+2-t2.*t5.*t9.*t12.*5.06e+2-
psi_dot_t.*t4.^3.*t11.*theta_dot_t.*5.06e+2+t2.*t5.*t12.*theta_dot_t.^2.*5.06
e+2+phi_dot_t.*t4.*t7.*t11.*theta_dot_t.*5.06e+2-
t2.*t4.*t5.*t7.*u1.*5.5e+1+t2.*t4.*t5.*t7.*u3.*5.5e+1+phi_dot_t.*psi_dot_t.*t
2.*t5.*t7.*t12.*5.06e+2))./5.52e+2];
mt3 = [((t4.*u1.*6.0e+1-t4.*u3.*6.0e+1+t13.*u1.*2.2e+1-
t13.*u2.*2.2e+1+t13.*u3.*2.2e+1-
t13.*u4.*2.2e+1+phi_dot_t.*psi_dot_t.*t12.*5.52e+2+t4.*t11.*u1.*5.5e+1-
t4.*t11.*u3.*5.5e+1-
```

```
phi_dot_t.*psi_dot_t.*t11.*t12.*5.06e+2+t7.*t9.*t11.*t12.*5.06e+2+t2.*t5.*t7.
*u2.*5.5e+1-
t2.*t5.*t7.*u4.*5.5e+1+phi_dot_t.*t2.*t4.*t5.*theta_dot_t.*5.06e+2-
psi_dot_t.*t2.*t4.*t5.*t7.*theta_dot_t.*5.06e+2).*(-1.0./5.52e+2))./t4];
mt4 = [(t16.*(u1.*-9.2e+1+u2.*9.2e+1-u3.*9.2e+1+u4.*9.2e+1-
t7.*u2.*1.15e+2+t7.*u4.*1.15e+2+t11.*u1.*4.4e+1-
t11.*u2.*4.4e+1+t11.*u3.*4.4e+1-
t11.*u4.*4.4e+1+phi_dot_t.*t4.*theta_dot_t.*4.6e+1+psi_dot_t.*t14.*theta_dot_
t.*5.29e+2+t7.*t11.*u2.*5.5e+1-
t7.*t11.*u4.*5.5e+1+phi_dot_t.*t4.*t11.*theta_dot_t.*5.06e+2-
t2.*t4.*t5.*u1.*5.5e+1+t2.*t4.*t5.*u3.*5.5e+1+phi_dot_t.*psi_dot_t.*t2.*t5.*t
12.*5.06e+2-psi_dot_t.*t4.*t7.*t11.*theta_dot_t.*5.06e+2-
t2.*t5.*t7.*t9.*t12.*5.06e+2))./5.52e+2];
f = [mt1;mt2;mt3;mt4];
end
```

```
Not enough input arguments.

Error in QuadrotorStateFcnm1 (line 8)
phi_t = in1(4,:);
        ^^^^^^^^
```

Published with MATLAB® R2024b

```
clc;
clear;
close all;

% Define the quadrotor dynamics and Jacobian functions (assuming these
functions are defined elsewhere)
% (place the QuadrotorStateFcnm1, QuadrotorStateJacobianFcnm1,
QuadrotorStateFcnm2, and QuadrotorStateJacobianFcnm2 code here)

% Define the nonlinear MPC controller
nx = 12;
ny = 12;
nu = 4;
nlmpcobj = nlmpc(nx, ny, nu);

% Specify the prediction model state function and Jacobians (for MPC
configuration)
nlmpcobj.Model.StateFcn = "QuadrotorStateFcnm1";
nlmpcobj.Jacobian.StateFcn = @QuadrotorStateJacobianFcnm1;

validateFcns(nlmpcobj, rand(nx,1), rand(nu,1));

% Specify sample time, prediction horizon, and control horizon
Ts = 0.1;
p = 18;
m = 2;
nlmpcobj.Ts = Ts;
nlmpcobj.PredictionHorizon = p;
nlmpcobj.ControlHorizon = m;

% Define constraints on the control inputs
limitRatio = 1; %0.32;
nlmpcobj.MV = struct(...
    'Min', {0;0;0;0}, ...
    'Max', {10*limitRatio;10*limitRatio;10*limitRatio;10*limitRatio}, ...
    'RateMin', {-2;-2;-2;-2}, ...
    'RateMax', {2;2;2;2});

% Define weights for output variables, manipulated variables, and rates
nlmpcobj.Weights.OutputVariables = [1 1 1 1 1 1 0 0 0 0 0 0];
nlmpcobj.Weights.ManipulatedVariables = [0.1 0.1 0.1 0.1];
nlmpcobj.Weights.ManipulatedVariablesRate = [0.1 0.1 0.1 0.1];

% Define initial conditions for the quadrotor
x0 = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0];

% Define waypoints for the desired trajectory
waypoints = [
    0 0 0;
    0 0 10;
    20 0 10;
    20 0 2;
```

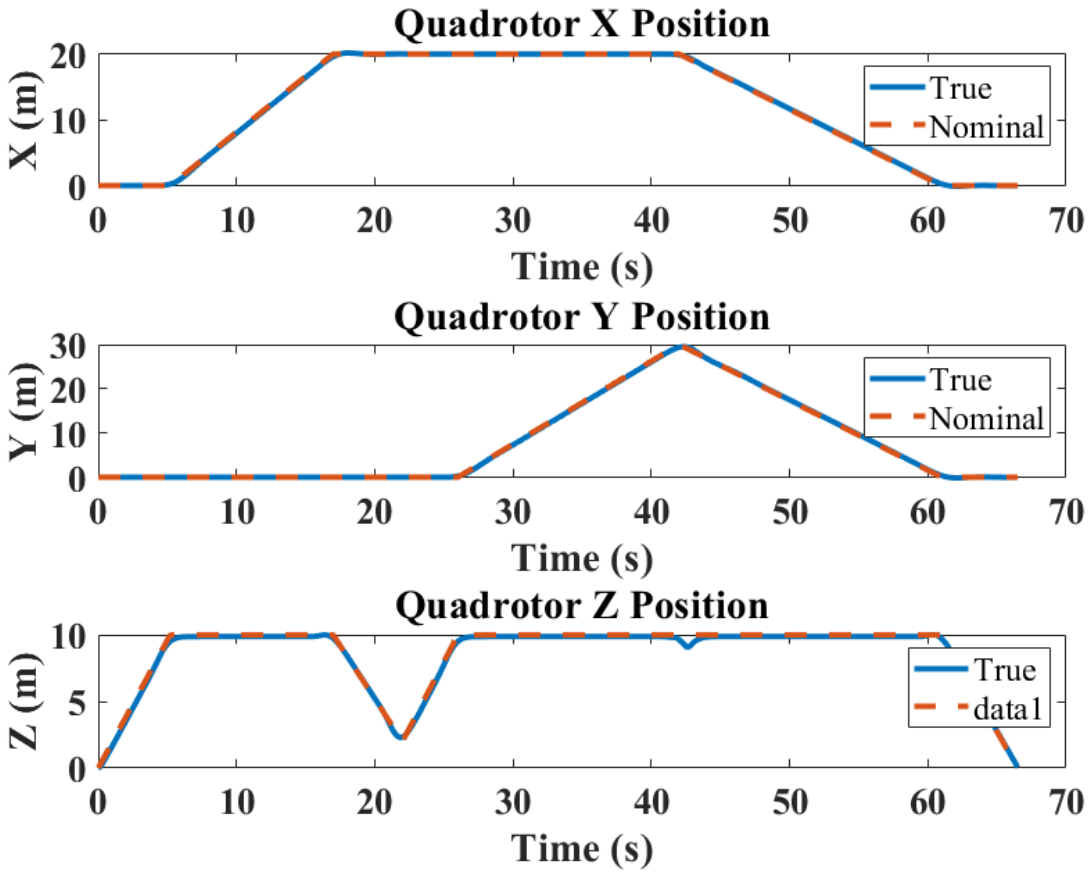
```
    20 0 10;
    20 30 10;
    0 0 10;
    0 0 0
];
nPoints = size(waypoints, 1);

% Define the times at which the waypoints should be reached
tWaypoints = [0, 5.2, 17, 22, 26, 42, 61, 66.5]; % New specified times for
waypoints

% Define the total duration of the simulation
Duration = tWaypoints(end);  % 66.5 seconds
tSim = 0:Ts:Duration;  % Time vector for the entire simulation duration

% Interpolate reference trajectory for all time steps using time vectors
refTrajectory = interp1(tWaypoints, waypoints(1:length(tWaypoints), :),
tSim, 'linear', 'extrap');

% % Define the total duration of the simulation
% Duration = 66.5;
% tWaypoints = linspace(0, Duration, nPoints); % Time points for waypoints
%
% % Define the time vector for the entire simulation duration
% tSim = 0:Ts:Duration;
%
% % Interpolate reference trajectory for all time steps using time vectors
% refTrajectory = interp1(tWaypoints, waypoints, tSim, 'linear', 'extrap');

% Initialize simulation variables
steps = length(tSim) - 1;
x = x0;
nlopts = nlmpcmoveopt;
nlopts.MVTarget = [4.9 4.9 4.9 4.9];
lastMV = nlopts.MVTarget;

xHistory = zeros(nx, steps+1);
xHistory(:, 1) = x0;
uHistory = zeros(nu, steps);

hbar = waitbar(0, 'Simulation Progress');

for k = 1:steps
    % Calculate the reference at the next prediction horizon
    t = tSim(k) + (0:Ts:(p-1)*Ts);
    yref = interp1(tSim, refTrajectory, t, 'linear', 'extrap');

    % Ensure yref has 12 columns
    yref = [yref, zeros(p, ny - size(yref, 2))];

    % Compute the control input using MPC
    [uk, nlopts, info] = nlmpcmove(nlmpcobj, x, lastMV, yref, [], nlopts);

    % Determine which dynamics system to use
```

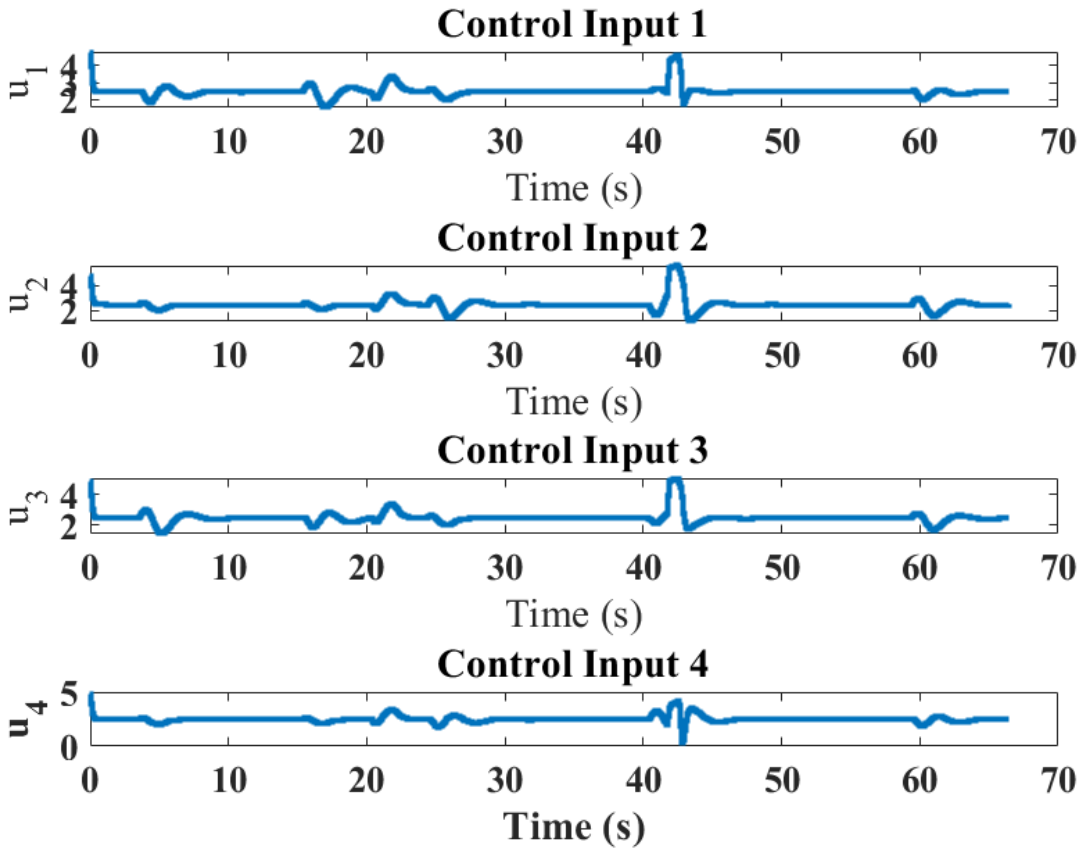
```
    if norm(x(1:3) - [20; 0; 2]) < 1
        % Switch to dynamics function 2
        odefun = @(t,xk) QuadrotorStateFcnm1(xk, uk);
    elseif norm(x(1:3) - [20; 30; 10]) < 1
        % Switch back to dynamics function 1
        odefun = @(t,xk) QuadrotorStateFcnm2(xk, uk);
    else
        % Continue with the current state function
        odefun = @(t,xk) QuadrotorStateFcnm1(xk, uk);
    end

    % Simulate the quadrotor dynamics
    [~, xout] = ode45(odefun, [0 Ts], x);
    x = xout(end, :)';

    % Store the state and control input history
    xHistory(:, k+1) = x;
    uHistory(:, k) = uk;
    lastMV = uk;

    waitbar(k / steps, hbar);
end

close(hbar);


err = xHistory(1:3,:)'-refTrajectory;



% Plot the results
figure;
subplot(3, 1, 1); plot(tSim, xHistory(1, :),'LineWidth',3); title('Quadrotor
X Position'); xlabel('Time (s)'); ylabel('X (m)');
set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
hold on; plot(tSim, refTrajectory(:, 1), '--','LineWidth',3); hold off;
set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
    legend('True','Nominal')
subplot(3, 1, 2); plot(tSim, xHistory(2, :),'LineWidth',3); title('Quadrotor
Y Position'); xlabel('Time (s)'); ylabel('Y (m)');
hold on; plot(tSim, refTrajectory(:, 2), '--','LineWidth',3); hold off;
set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
```

```
    legend('True','Nominal')
subplot(3, 1, 3);
set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
    plot(tSim, xHistory(3, :),'LineWidth',3); title('Quadrotor Z Position');
xlabel('Time (s)'); ylabel('Z (m)');
hold on;
legend('True','Nominal')
set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
    plot(tSim, refTrajectory(:, 3), '--','LineWidth',3); hold off;



set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])


figure;
for i = 1:nu
    subplot(nu, 1, i);
    stairs(tSim(1:end-1), uHistory(i, :),'LineWidth',3);
    set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])
    title(['Control Input ', num2str(i)]);
    xlabel('Time (s)');
    ylabel(['u_', num2str(i)]);
    % set(gca,'FontSize',20)
    % set(gca,'FontName','Times New Roman')
    % set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    % set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    % set(gcf, 'Position',  [800, 300, 900, 700])
end

set(gca,'FontSize',20)
    set(gca,'FontName','Times New Roman')
    set(get(gca, 'XAxis'), 'FontWeight', 'bold');
    set(get(gca, 'YAxis'), 'FontWeight', 'bold');
    set(gcf, 'Position',  [800, 300, 900, 700])

% Optional: Animate the trajectory of the quadrotor if you have an animation
```

```
function
% animateQuadrotorTrajectory(xHistory(1:3, :)', waypoints, Ts, Duration);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Analysis of user-provided model, cost, and constraint functions complete.
Warning: Ignoring extra legend entries.





Published with MATLAB® R2024b

```
% This script defines a continuous-time nonlinear quadrotor model and
% generates a state function and its Jacobian function used by the
% nonlinear MPC controller in the quadrotor trajectory tracking example.



% Create symbolic functions for time-dependent angles
% phi: roll angle
% theta: pitch angle
% psi: yaw angle
syms phi(t) theta(t) psi(t)

% Transformation matrix for angular velocities from inertial frame
% to body frame
W = [ 1,  0,        -sin(theta);
      0,  cos(phi),  cos(theta)*sin(phi);
      0, -sin(phi),  cos(theta)*cos(phi) ];

% Rotation matrix R_ZYX from body frame to inertial frame
R = rotationMatrixEulerZYX(phi,theta,psi);

% Create symbolic variables for diagonal elements of inertia matrix
syms Ixx Iyy Izz

% Jacobian that relates body frame to inertial frame velocities
I = [Ixx, 0, 0; 0, Iyy, 0; 0, 0, Izz];
J = W.'*I*W;

% Coriolis matrix
dJ_dt = diff(J);
h_dot_J = [diff(phi,t), diff(theta,t), diff(psi,t)]*J;
grad_temp_h = transpose(jacobian(h_dot_J,[phi theta psi]));
C = dJ_dt - 1/2*grad_temp_h;
C = subsStateVars(C,t);

% Define fixed parameters and control inputs
% k: lift constant
% l: distance between rotor and center of mass
% m: quadrotor mass
% b: drag constant
% g: gravity
% ui: squared angular velocity of rotor i as control input
syms k l m b g u1 u2 u3 u4

% Torques in the direction of phi, theta, psi
tau_beta = [l*k*(-u2+u4); l*k*(-u1+u3); b*(-u1+u2-u3+u4)];

% Total thrust
T = k*(u1+u2+u3+u4);

% Create symbolic functions for time-dependent positions
syms x(t) y(t) z(t)
```

```
% Create state variables consisting of positions, angles,
% and their derivatives
state = [x; y; z; phi; theta; psi; diff(x,t); diff(y,t); ...
    diff(z,t); diff(phi,t); diff(theta,t); diff(psi,t)];
state = subsStateVars(state,t);

f = [ % Set time-derivative of the positions and angles
      state(7:12);

      % Equations for linear accelerations of the center of mass
      -g*[0;0;1] + R*[0;0;T]/m;

      % Euler-Lagrange equations for angular dynamics
      inv(J)*(tau_beta - C*state(10:12))
];

f = subsStateVars(f,t);

% Replace fixed parameters with given values here
IxxVal = 1.2;
IyyVal = 1.2;
IzzVal = 2.3;
kVal = 1;
lVal = 0.25;
mVal = 2;
bVal = 0.2;
gVal = 9.81;

f = subs(f, [Ixx Iyy Izz k l m b g], ...
    [IxxVal IyyVal IzzVal kVal lVal mVal bVal gVal]);
f = simplify(f);

% Calculate Jacobians for nonlinear prediction model
A = jacobian(f,state);
control = [u1; u2; u3; u4];
B = jacobian(f,control);

% Create QuadrotorStateFcn.m with current state and control
% vectors as inputs and the state time-derivative as outputs
matlabFunction(f,"File","QuadrotorStateFcn", ...
    "Vars",{state,control});

% Create QuadrotorStateJacobianFcn.m with current state and control
% vectors as inputs and the Jacobians of the state time-derivative
% as outputs
matlabFunction(A,B,"File","QuadrotorStateJacobianFcn", ...
    "Vars",{state,control});

% Confirm the functions are generated successfully
while isempty(which('QuadrotorStateJacobianFcn'))
    pause(0.1);
end
```

```
function [Rz,Ry,Rx] = rotationMatrixEulerZYX(phi,theta,psi)
% Euler ZYX angles convention
    Rx = [ 1,           0,           0;
           0,           cos(phi),  -sin(phi);
           0,           sin(phi),   cos(phi) ];
    Ry = [ cos(theta),  0,           sin(theta);
           0,           1,           0;
          -sin(theta),  0,           cos(theta) ];
    Rz = [cos(psi),    -sin(psi),   0;
          sin(psi),     cos(psi),   0;
          0,           0,           1 ];
    if nargout == 3
        % Return rotation matrix per axes
        return;
    end
    % Return rotation matrix from body frame to inertial frame
    Rz = Rz*Ry*Rx;
end

function stateExpr = subsStateVars(timeExpr,var)
    if nargin == 1
        var = sym("t");
    end
    repDiff = @(ex) subsStateVarsDiff(ex,var);
    stateExpr = mapSymType(timeExpr,"diff",repDiff);
    repFun = @(ex) subsStateVarsFun(ex,var);
    stateExpr = mapSymType(stateExpr,"symfunOf",var,repFun);
    stateExpr = formula(stateExpr);
end

function newVar = subsStateVarsFun(funExpr,var)
    name = symFunType(funExpr);
    name = replace(name,"_Var","");
    stateVar = "_" + char(var);
    newVar = sym(name + stateVar);
end

function newVar = subsStateVarsDiff(diffExpr,var)
    if nargin == 1
      var = sym("t");
    end
    c = children(diffExpr);
    if ~isSymType(c{1},"symfunOf",var)
      % not f(t)
      newVar = diffExpr;
      return;
    end
    if ~any([c{2:end}] == var)
      % not derivative wrt t only
      newVar = diffExpr;
      return;
    end
    name = symFunType(c{1});
    name = replace(name,"_Var","");
```

```
    extension = "_" + join(repelem("d",numel(c)-1),"") + "ot";
    stateVar = "_" + char(var);
    newVar = sym(name + extension + stateVar);
end
```


Published with MATLAB® R2024b

Contents


```
% NA583 Final Project-Quadrotor MRAC

clear;clc;close all;

% Drone parameter values
global m Ka Km Ix Iy Iz g l;

g = 9.81;
l = 0.2;
Ix = 7.5e-3;
Iy = 7.5e-3;
Iz = 13e-3;
Km = 3.1e-7*0.2;

m = 0.85; %kg
Ka = m*g/(400)^2;

% linearized system around hovering
global K A B;
psi = 0; % @hovering, psi=0
Phi = [0,0,0,-g*sin(psi),-g*cos(psi),0;
    0,0,0,g*cos(psi),-g*sin(psi),0;
    zeros(4,6)];

A = [zeros(6,6), eye(6);
    Phi, zeros(6,6)];
Delta = [Ka/m, Ka/m, Ka/m, Ka/m;
    0, -Ka*l/Ix, 0, Ka*l/Ix;
    Ka*l/Iy, 0, -Ka*l/Iy, 0;
    Km/Iz, -Km/Iz, Km/Iz, -Km/Iz];
B = [zeros(8,4);
    Delta];

% assume 10% error of drone param
m_est = 1.1*m;
l_est = 1.1*l;

global A_act B_act
A_act = A;
m_act = 1.5 * m;
l_act = 1.5 * l;
Delta_act = [   Ka/m_act,     Ka/m_act,     Ka/m_act,    Ka/m_act;
              0, -Ka*l_act/Ix,        0, Ka*l_act/Ix;
         Ka*l_act/Iy,        0, -Ka*l_act/Iy,       0;
           Km/Iz,   -Km/Iz,    Km/Iz,  -Km/Iz];

B_act = [zeros(8,4);
         Delta_act];
```

MRC using pole placement

```
pole_des = -linspace(1,20,12);
K = place(A,B,pole_des);
global Kr_lin;
Kr_lin = B\(A-B*K);

% % Check closed loop pole of the system
% array2table(eig(A-B*K)
```

MRAC

```
tspan = [0,5];

% x = [x,y,z,phi,theta,psi,xdot,ydot,zdot,p,q,r]
%x0 = [0;0;0;0;0;0;0;0;0;0;0;0];
global wayppt;
wayppt = [0, 0, 0; % x0
    0, 0, 10;
    20, 0, 10;
    20, 0, 2; % m +0.65
    20, 0, 10;
    20, 30, 10; % m -0.65, back to 0.85
    0, 0, 10;
```

```
    0, 0, 0];
traj = [wayppt, zeros(8,12-3)]; % dim = 8x12
% number of waypoint
n = 8;

global r;

all_t = [];
all_x = [];
all_xm = [];


for i = 1:n-1
% i = 2;

if i==4
    m = 0.85+0.65; %kg
    Ka = m*g/(400)^2;
    Delta = [Ka/m, Ka/m, Ka/m, Ka/m;
        0, -Ka*l/Ix, 0, Ka*l/Ix;
        Ka*l/Iy, 0, -Ka*l/Iy, 0;
        Km/Iz, -Km/Iz, Km/Iz, -Km/Iz];
    B = [zeros(8,4);
         Delta];
    m_act = 1.5 * m;
    l_act = 1.5 * l;
    Delta_act = [   Ka/m_act,     Ka/m_act,     Ka/m_act,    Ka/m_act;
               0, -Ka*l_act/Ix,        0, Ka*l_act/Ix;
         Ka*l_act/Iy,        0, -Ka*l_act/Iy,       0;
           Km/Iz,   -Km/Iz,    Km/Iz,  -Km/Iz];

    B_act = [zeros(8,4);
         Delta_act];
elseif i==6
    m = 0.85; %kg
    Ka = m*g/(400)^2;
    Delta = [Ka/m, Ka/m, Ka/m, Ka/m;
        0, -Ka*l/Ix, 0, Ka*l/Ix;
        Ka*l/Iy, 0, -Ka*l/Iy, 0;
        Km/Iz, -Km/Iz, Km/Iz, -Km/Iz];
    B = [zeros(8,4);
         Delta];
    m_act = 1.5 * m;
    l_act = 1.5 * l;
    Delta_act = [   Ka/m_act,     Ka/m_act,     Ka/m_act,    Ka/m_act;
               0, -Ka*l_act/Ix,        0, Ka*l_act/Ix;
         Ka*l_act/Iy,        0, -Ka*l_act/Iy,       0;
           Km/Iz,   -Km/Iz,    Km/Iz,  -Km/Iz];

    B_act = [zeros(8,4);
         Delta_act];
end

    x0 = zeros(88,1)+[traj(i,:)';traj(i,:)';zeros(88-24,1)];
    r = traj(i+1,:)';

    temp_K = -K';
    x0(25:72) = [temp_K(1,:) temp_K(2,:) temp_K(3,:) temp_K(4,:) temp_K(5,:) temp_K(6,:) temp_K(7,:) temp_K(8,:) temp_K(9,:) temp_K(10,:) temp_K(11,:) temp_K(12,:)]
    x0(73) = 1;
    x0(78) = 1;
    x0(83) = 1;
    x0(88) = 1;
    [t, x] = ode45(@MRAC, tspan, x0);
    x_adap_act = x(:,1:12);
    xm_adap_act = x(:,13:24);

    all_x = [all_x;x_adap_act];
    all_xm = [all_xm;xm_adap_act];
    all_t = [all_t;t+(i-1)*tspan(2)];
end

%     t_plot = [t_plot,t];
%     x_adap_plot = [x_adap_plot, x_adap_act];
%     xm_plot = [xm_plot, xm_adap_act];
%
% end
```

plot

```
figure; hold on;
x_plot = all_x(:,1);
y_plot = all_x(:,2);
z_plot = all_x(:,3);
xm_plot = all_xm(:,1);
ym_plot = all_xm(:,2);
zm_plot = all_xm(:,3);
```

```
subplot(3,1,1)
plot(all_t,x_plot); hold on;
plot(all_t, xm_plot,'--');
legend('Model-Based','Reference');
xlabel('Times(s)');
ylabel('X (m)');
title('Quadrotor X position');
grid on;

subplot(3,1,2)
plot(all_t,y_plot); hold on;
plot(all_t, ym_plot,'--');
legend('Model-Based','Reference');
xlabel('Times(s)');
ylabel('Y (m)');
title('Quadrotor Y position');
grid on;

subplot(3,1,3)
plot(all_t,z_plot); hold on;
plot(all_t, zm_plot,'--');
legend('Model-Based','Reference');
xlabel('Times(s)');
ylabel('Z (m)');
title('Quadrotor Z position');
grid on;

hold off;
```

comparison

```
figure
subplot(3,1,1)
plot(all_t,(xm_plot - x_plot));
ylabel('x-error (m)')
xlabel('Times (s)')
grid on;

subplot(3,1,2)
plot(all_t,ym_plot-y_plot);
ylabel('y-error (m)')
xlabel('Times (s)')
grid on;

subplot(3,1,3)
plot(all_t,zm_plot-z_plot)
xlabel('Times (s)')
ylabel('z-error (m)')
grid on;
```

function

```
function xdot = MRAC(t,x)
global K Kr_lin A B A_act B_act r;       % x: [x xm kx(1-4,:) km(1-4,:)]

disp(t);
Am = A - B*K;
Bm = B;
Q = 600*eye(12);          % 10
P = lyap(Am',Q);
gamma_x = 0.005*eye(12);    % 0.05
gamma_r = 0.005*eye(4);     % 0.05
rt = -1* Kr_lin * r;

x = reshape(x,[1,88]);
xt = x(1:12)';
xm = x(13:24)';
Kx = [x(25:28);x(29:32);x(33:36);x(37:40);x(41:44);x(45:48);x(49:52);x(53:56);x(57:60);x(61:64);x(65:68);x(69:72)];
Kr = [x(73:76) ; x(77:80) ; x(81:84) ; x(85:88)];
ut = Kx' * xt + Kr' * rt;

e = xt - xm;

Kx_dot = -gamma_x * xt * e' * P * B_act;
Kr_dot = -gamma_r * rt * e' * P * B_act;

xdot = zeros(88,1);
xdot(1:12) = A_act*xt + B_act*ut;
xdot(13:24) = Am*xm + Bm*rt;
xdot(25:72) = [Kx_dot(1,:) Kx_dot(2,:) Kx_dot(3,:) Kx_dot(4,:) Kx_dot(5,:) Kx_dot(6,:) Kx_dot(7,:) Kx_dot(8,:) Kx_dot(9,:) Kx_dot(10,:) Kx_dot(11,:) Kx_dot(12,:)]';
xdot(73:88) = [Kr_dot(1,:) Kr_dot(2,:) Kr_dot(3,:) Kr_dot(4,:)]';
end
```

Published with MATLAB® R2022b

```
from __future__ import print_function, division

import numpy as np
import matplotlib.pyplot as plt

from py3dmath import Vec3, Rotation
from uav_sim.vehicle import Vehicle

from uav_sim.positioncontroller import PositionController
from uav_sim.attitudecontroller import QuadcopterAttitudeControllerNested
from uav_sim.mixer import QuadcopterMixer

from xadapt_controller.utils import QuadState,Model
from xadapt_controller.controller import AdapLowLevelControl
from pyplot3d.utils import ypr_to_R
from animate import animate_quadcopter_history
import pandas as pd

np.random.seed(0)

#=============================================================================
# Define the simulation
#=============================================================================
dt = 0.005  # sdifferent
endTime = 75

#=============================================================================
# Define the vehicle
#=============================================================================
mass = 0.850  # kg
Ixx = 5e-3
Iyy = 5e-3
Izz = 10e-3
Ixy = 0
Ixz = 0
Iyz = 0
omegaSqrToDragTorque = np.matrix(np.diag([0, 0, 0.00014]))  # N.m/(rad/s)**2
armLength = 0.166  # m

##MOTORS##
motSpeedSqrToThrust = 7.6e-6  # propeller coefficient
motSpeedSqrToTorque = 1.07e-7  # propeller coefficient
motInertia    = 1e-6  #inertia of all rotating parts (motor + prop) [kg.m**2]

motTimeConst = 0.001  # time constant with which motor's speed responds [s]
motMinSpeed  = 0  #[rad/s]
motMaxSpeed  = 1000  #[rad/s]

#=============================================================================
# Define the disturbance
#=============================================================================
stdDevTorqueDisturbance = 0e-3  # [N.m]

#=============================================================================
# Define the attitude controller
#=============================================================================
#time constants for the angle components:
timeConstAngleRP = 0.08  # [s]
timeConstAngleY  = 0.40  # [s]

#gain from angular velocities
timeConstRatesRP = 0.04  # [s]
timeConstRatesY  = 0.20   # [s]

#=============================================================================
# Define the position controller
#=============================================================================
disablePositionControl = False
posCtrlNatFreq = 2  # rad/s
posCtrlDampingRatio = 0.7 #0.7  # -

#=============================================================================
# Define RL lowlevel controller
#=============================================================================
low_level_controller = AdapLowLevelControl()
# Initialize our quadrotor's state
cur_state = QuadState()
# Set the maximum motor speed for this quadcopter model in RPM
# Note: because the controller can adapt down to motors-level, this highest-rpm
# can be gained directly from motor datasheet without measurements from the experiments.
low_level_controller.set_max_motor_spd(motMaxSpeed)

#=============================================================================
# Compute all things:
#=============================================================================

inertiaMatrix = np.matrix([[Ixx, Ixy, Ixz], [Ixy, Iyy, Iyz], [Ixz, Iyz, Izz]])
quadrocopter = Vehicle(mass, inertiaMatrix, omegaSqrToDragTorque, stdDevTorqueDisturbance)


# Our quadcopter model as
#
#           x
#           ^
#    (-)mot3 | mot0(+)
#            |
#      y<----+-----
#            |
#    (+)mot2 | mot1(-)


motor_pos = armLength*(2**0.5)

quadrocopter.add_motor(Vec3( motor_pos, -motor_pos, 0), Vec3(0,0,1), motMinSpeed, motMaxSpeed, motSpeedSqrToThrust, motSpeedSqrToTorque, motTimeConst, motInert
quadrocopter.add_motor(Vec3( -motor_pos, -motor_pos, 0), Vec3(0,0,-1), motMinSpeed, motMaxSpeed, motSpeedSqrToThrust, motSpeedSqrToTorque, motTimeConst, motIne
quadrocopter.add_motor(Vec3(-motor_pos, +motor_pos, 0), Vec3(0,0,1), motMinSpeed, motMaxSpeed, motSpeedSqrToThrust, motSpeedSqrToTorque, motTimeConst, motInert
```

```
quadrocopter.add_motor(Vec3( motor_pos,motor_pos, 0), Vec3(0,0, -1), motMinSpeed, motMaxSpeed, motSpeedSqrToThrust, motSpeedSqrToTorque, motTimeConst, motInert

posControl = PositionController(posCtrlNatFreq, posCtrlDampingRatio)
attController = QuadcopterAttitudeControllerNested(timeConstAngleRP, timeConstAngleY, timeConstRatesRP, timeConstRatesY)
mixer = QuadcopterMixer(mass, inertiaMatrix, motor_pos, motSpeedSqrToTorque/motSpeedSqrToThrust)

#=============================================================================
# Define the trajectory
#=============================================================================

# Define the trajectory starting state:
# pos0 = np.random.uniform(-2, 2, 3)  # position
# pos0[2] += 2 # make sure it starts above the ground
# vel0 = np.random.uniform(-2,2,3) #velocity
# att0 = np.random.uniform(-1,1,4) #attitude
# att0 /= np.linalg.norm(att0) #normalize

pos0 = [0,0,0]  # position
vel0 = [0,0,0] #velocity
att0 = [1,0,0,0] #attitude
att0 /= np.linalg.norm(att0) #normalize

# Define the goal state:
# posf = [2, 2, 2]  # position
# velf = [0, 0, 0]  # velocity
# accf = [0, 0, 0]  # acceleration

quadrocopter.set_position(Vec3(pos0))
quadrocopter.set_velocity(Vec3(vel0))
quadrocopter.set_attitude(Rotation(att0[0], att0[1], att0[2], att0[3]))

#start at equilibrium rates:
quadrocopter._omega = Vec3(0,0,0)

#=========================================
from scipy.interpolate import interp1d
import numpy as np

# Define the waypoints
waypoints = np.array([
    [0, 0, 0],
    [0, 0, 10],
    [20, 0, 10],
    [20, 0, 2],
    [20, 0, 10],
    [20, 30, 10],
    [0, 0, 10],
    [0, 0, 0]
])

# Define constant speed (m/s) and pause duration (s)
speed = 2
pause_duration = 1

# Initialize lists for times and positions
times = [0]  # Start time
positions = [waypoints[0]]  # Starting position

# Generate trajectory
for i in range(1, len(waypoints)):
    # Previous position
    start = waypoints[i - 1]
    end = waypoints[i]

    # Calculate travel time for the segment
    distance = np.linalg.norm(end - start)
    travel_time = distance / speed

    # Transition segment: Add intermediate points between start and end
    segment_times = np.linspace(times[-1], times[-1] + travel_time, num=100, endpoint=False)
    segment_positions = np.linspace(start, end, num=100, endpoint=False)
    positions.extend(segment_positions)
    times.extend(segment_times)

    # Hover segment: Hold position for the pause duration
    hover_times = np.linspace(times[-1], times[-1] + pause_duration, num=50, endpoint=False)
    hover_positions = [end] * 50
    positions.extend(hover_positions)
    times.extend(hover_times)

# Add the final waypoint
times.append(times[-1] + pause_duration)
positions.append(waypoints[-1])

# Convert lists to numpy arrays
trajectory_positions = np.array(positions[1:])
# print(trajectory_positions)
trajectory_times = np.array(times[1:])

# Verify lengths
if len(times) != len(positions):
    print(f"Length mismatch: times={len(times)}, positions={len(positions)}")

# Interpolation functions for position, velocity, and acceleration
x_interp = interp1d(trajectory_times, trajectory_positions[:, 0], kind='linear', fill_value="extrapolate")
y_interp = interp1d(trajectory_times, trajectory_positions[:, 1], kind='linear', fill_value="extrapolate")
z_interp = interp1d(trajectory_times, trajectory_positions[:, 2], kind='linear', fill_value="extrapolate")
#=============================================================================
# Run the simulation
#=============================================================================

numSteps = int((endTime)/dt)
index = 0

t = 0
```

```
posHistory       = np.zeros([numSteps,3])
velHistory       = np.zeros([numSteps,3])
posCmdHistory    = np.zeros([numSteps,3])
velCmdHistory    = np.zeros([numSteps,3])
angVelHistory    = np.zeros([numSteps,3])
attHistory       = np.zeros([numSteps,3])
motForcesHistory = np.zeros([numSteps,quadrocopter.get_num_motors()])
inputHistory     = np.zeros([numSteps,quadrocopter.get_num_motors()])
times            = np.zeros([numSteps,1])


#=======================================================================
current_waypoint_index = 0
hover_start_time = None
is_pausing = False


while index < numSteps:

    posf = [x_interp(t), y_interp(t), z_interp(t)]
    velf = [
        (x_interp(t + dt) - x_interp(t)) / dt,
        (y_interp(t + dt) - y_interp(t)) / dt,
        (z_interp(t + dt) - z_interp(t)) / dt
    ]
    accf = [
            (x_interp(t + dt) - 2 * x_interp(t) + x_interp(t - dt)) / (dt ** 2),
            (y_interp(t + dt) - 2 * y_interp(t) + y_interp(t - dt)) / (dt ** 2),
            (z_interp(t + dt) - 2 * z_interp(t) + z_interp(t - dt)) / (dt ** 2)
            ]
    # Define commands
    accDes = posControl.get_acceleration_command(Vec3(*posf), Vec3(*velf), Vec3(*accf),
                                                 quadrocopter._pos, quadrocopter._vel)
    if disablePositionControl:
        accDes *= 0  # Disable position control

    # RL Control
    thrustNormDes = accDes + Vec3(0, 0, 9.81)
    angVelDes = attController.get_angular_velocity(thrustNormDes, quadrocopter._att, quadrocopter._omega)
    cur_state.att = quadrocopter._att.to_array().flatten()
    cur_state.omega = quadrocopter._omega.to_array().flatten()
    cur_state.proper_acc = quadrocopter._accel.to_array().flatten()
    cur_state.cmd_collective_thrust = thrustNormDes.z
    cur_state.cmd_bodyrates = angVelDes.to_array().flatten()

    motCmds = low_level_controller.run(cur_state)
    quadrocopter.run(dt, motCmds, spdCmd=True)

    # Logging and visualization updates
    times[index] = t
    inputHistory[index, :] = motCmds
    posHistory[index, :] = quadrocopter._pos.to_list()
    velHistory[index, :] = quadrocopter._vel.to_list()
    posCmdHistory[index, :] = posf
    velCmdHistory[index, :] = velf
    attHistory[index, :] = quadrocopter._att.to_euler_YPR()
    angVelHistory[index, :] = quadrocopter._omega.to_list()
    motForcesHistory[index, :] = quadrocopter.get_motor_forces()

    t += dt
    index += 1

     # add payload
    if t >= 21 and t < 42:
        print("payload added")
        quadrocopter._mass = 1.5
    elif t >= 42:
        print("payload dropped")
        quadrocopter._mass = 0.85

#=======================================================================
# Make the plots
#=======================================================================

fig, ax = plt.subplots(3,1, sharex=True)

ax[0].plot(times, posHistory[:,0], label='x')
ax[0].plot(times, posHistory[:,1], label='y')
ax[0].plot(times, posHistory[:,2], label='z')
ax[0].plot(times, posCmdHistory[:,0], linestyle='--', label='x_ref')
ax[0].plot(times, posCmdHistory[:,1], linestyle='--', label='y_ref')
ax[0].plot(times, posCmdHistory[:,2], linestyle='--', label='z_ref')
# ax[0].plot(times, np.ones_like(times)*posf[0], '--', color='C0')
# ax[0].plot(times, np.ones_like(times)*posf[1], '--', color='C1')
# ax[0].plot(times, np.ones_like(times)*posf[2], '--', color='C2')
# ax[1].plot(times, velHistory)
# ax[1].plot(times, velCmdHistory)
ax[1].plot(times, attHistory[:,0]*180/np.pi, label='Yaw')
ax[1].plot(times, attHistory[:,1]*180/np.pi, label='Pitch')
ax[1].plot(times, attHistory[:,2]*180/np.pi, label='Roll')
# ax[3].plot(times, angVelHistory[:,0], label='p')
# ax[3].plot(times, angVelHistory[:,1], label='q')
# ax[3].plot(times, angVelHistory[:,2], label='r')
ax[2].plot(times, inputHistory)
ax[2].plot(times, inputHistory)

ax[-1].set_xlabel('Time [s]')

ax[0].set_ylabel('Pos')
# ax[1].set_ylabel('Vel')
ax[1].set_ylabel('Att [deg]')
# ax[3].set_ylabel('AngVel (in B)')
ax[2].set_ylabel('Motor Forces')

ax[0].set_xlim([0, endTime])
ax[0].legend()
```

```
ax[1].legend()
# ax[3].legend()

plt.show()

data_dict = {
    'Time': times.flatten(),
    'PosX': posHistory[:, 0],
    'PosY': posHistory[:, 1],
    'PosZ': posHistory[:, 2],
    'VelX': velHistory[:, 0],
    'VelY': velHistory[:, 1],
    'VelZ': velHistory[:, 2],
    'Yaw': attHistory[:, 0],
    'Pitch': attHistory[:, 1],
    'Roll': attHistory[:, 2],
    'Motor1': inputHistory[:, 0],
    'Motor2': inputHistory[:, 1],
    'Motor3': inputHistory[:, 2],
    'Motor4': inputHistory[:, 3],
}
# Create a DataFrame from the dictionary
data = pd.DataFrame(data_dict)

# Save the DataFrame to a CSV file
data.to_csv('data/quadcopter_data.csv', index=False)


# Load your data
data = pd.read_csv('data/quadcopter_data.csv')
times = data['Time'].values

posHistory = data[['PosX', 'PosY', 'PosZ']].values
attHistory = data[['Yaw','Pitch', 'Roll']].values
x = posHistory.T
steps = len(times)

R = np.zeros((3, 3, steps))
for i in range(steps):
    ypr = attHistory[i,:]
    R[:, :, i] = ypr_to_R(ypr, degrees=False)


# animate_quadcopter_history(times, x, R, arm_length=0.8)
```