Submitted by: Shahram Najam Syed **Dated: 2nd November 2023**
AndrewID: snsyed **Homework # 3 (16-665 Robot Mobility)**

Part A:

1. ***System Modeling [10]:*** **Based on lecture 2, complete the "dynamics" function in the "Quadrotor_2D" class. Run "python 2d.py 1" to observe the output. We have disabled randomness for this question so you should see results same or similar (up to numerical errors) to this:**

```
pos: [0. 0.]
vel: [0.     0.0545]
theta: 0.0
omega: 0.1487801150528891
************************
pos: [-0.00074911  0.02996508]
vel: [-0.03744681  0.59866976]
theta: 0.08182906327908901
omega: 1.6365812655817802
************************
pos: [-0.01354985  0.11368547]
vel: [-0.3000653   1.11769881]
theta: 0.31243824161106704
omega: 3.1243824161106697
```

Take a screenshot of your output.

Ans: In this question, we are asked to implement the dynamics model for the 2D drone. For that reason, the "dynamics" function in the "Quadrotor_2D" class was completed, and the code was executed to receive the drone's state at index $\mathrm{i} = 0, 10, 20$ for a constant input:

$$u = np.array([0.019, 0.023]) * 9.81.$$

We were further asked to juxtapose the results obtained with those shown in the writeup (which should matchup up to numerical errors). Following is the result I got from my implementation of the "dynamics" function:

```
(am_hw3) C:\Users\DELL\Desktop\aerial_mobility_release\aerial_mobility_release>python 2d.py 1
************************
pos: [0. 0.]
vel: [0.     0.0545]
theta: 0.0
omega: 0.1487801150528891
************************
pos: [-0.00074911  0.02996508]
vel: [-0.03744681  0.59866976]
theta: 0.08182906327908901
omega: 1.6365812655817802
************************
pos: [-0.01354985  0.11368547]
vel: [-0.3000653   1.11769881]
theta: 0.31243824161106704
omega: 3.1243824161106697
You can open the visualizer by visiting the following URL:
http://127.0.0.1:7000/static/
```

And following is the touchstone result to be compared against in the writeup:

```
pos: [0. 0.]
vel: [0.     0.0545]
theta: 0.0
omega: 0.1487801150528891
**********
pos: [-0.00074911  0.02996508]
vel: [-0.03744681  0.59866976]
theta: 0.08182906327908901
omega: 1.6365812655817802
**********
pos: [-0.01354985  0.11368547]
vel: [-0.3000653   1.11769881]
theta: 0.31243824161106704
omega: 3.1243824161106697
```

The mathematical implementation that defines the dynamics of the 2D drone in the "dynamics" function is as follows:

- **Control Inputs:**
  - Total thrust: $\mathrm{T} = u_1 + u_2$
  - Torque: $\tau = (u_1 - u_2) * arm$
- **Dynamics Equations:**
  - Position update: $p_{dot} = v$
  - Velocity update: $v_{dot} = \begin{bmatrix} 0 \\ -g \end{bmatrix} + \frac{F}{m}\begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$
  - Angular position (θ) update: $\theta_{dot} = \omega$
  - Angular velocity (ω) update: $\omega_{dot} = \tau/\mathrm{J}$
- **State updates (considering time discretization and some noise):**
  - Updated position: $p_{new} = p_{old} + dt * p_{dot}$
  - Updated velocity: $v_{new} = v_{old} + dt * v_{dot} + dt * \sigma_t * random\ noise$
  - Updated angular position: $\theta_{new} = \theta_{old} + dt * \theta_{dot}$
  - Updated angular velocity: $\omega_{new} = \omega_{old} + dt * \omega_{dot} + dt * \sigma_r * random\ noise$

Where,

- u_1 and u_2 are the control inputs (thrusts from the two rotors).
- g is gravitational acceleration.
- T is the total thrust.
- τ is the torque.
- m is the drone's mass.
- θ is the drone's angular position.
- J is the moment of inertia of the drone.
- arm is the distance between drone's center to each rotor (assumed to be the same for both rotors).
- σ_t and σ_r represent noise terms in the velocity and angular velocity respectively to simulate uncertainties or disturbances in the real-world dynamics.
- dt is the time step for discretization.

2. ***Cascaded Setpoint Control [10]:*** **Based on lecture 3, complete the "cascaded_control" function in the "Quadrotor_2D" class. Design PD gains for position and attitude control. Run "python 2d.py 2" to track two setpoints: from initial condition [0, 0] to [1, 1], and from [0, 0] to [1, 0].**

Ans: The "cascaded_control" function implemented in the code details a cascaded approach for a 2D drone. The goal is to track a desired trajectory or setpoint defined by the position p_d, velocity v_d, acceleration a_d, desired angular velocity ω_d and desired torque τ_d. The controller is split into two main steps: the outer loop of position control and the inner loop of attitude control.

- **Position Control:**

  The desired force $f_d = g\begin{bmatrix}0\\1\end{bmatrix} + a_d - K_P(p - p_d) - K_D(v - v_d)$

  Where:
  - g is the gravitational acceleration
  - K_P and K_D are the proportional and derivative gains for positional control
  - p and v are the current position and velocity
  - p_d , v_d , and a_d are the desired position, velocity, and acceleration
- **Attitude Control:**

  After computing the desired force, we need to determine the desired thrust T and the desired attitude θ_d:

$$T = m\begin{bmatrix}-\sin\theta\\ \cos\theta\end{bmatrix} f_d$$

$$\theta_d = -arctan2(f_{d,x}/f_{d,y})$$

  With T and θ_d known, the torque τ is then computed as:

$$\tau = J\,(-K_{P,\tau}(\theta - \theta_d) - -K_{D,\tau}\,(\omega - \omega_d) + \tau_d)$$

  Where,

  - J is the moment of inertia.
  - θ is the current angular position.
  - ω is the current angular velocity.
  - $K_{P,\tau}$ and $K_{D,\tau}$ are the proportional and derivative gains for attitude control.
- **Conversion to Rotor Forces:**

  The desired thrust T and torque τ are converted to individual rotor forces u_1 and u_2 using:

$$\begin{bmatrix}u_1\\u_2\end{bmatrix} = \begin{bmatrix}1 & 1\\ -arm & arm\end{bmatrix}^{-1}\begin{bmatrix}T\\ \tau\end{bmatrix}$$

The implementation provided separates the horizontal and vertical dynamics (position control) from the rotational dynamics (attitude control). The idea is that by controlling the position and velocity first (outer loop), you can then derive desired angles and rotational rates which are controlled in the inner loop.

- **[5] Report the figure for both setpoints, and comment on their differences. For both setpoints, take three screenshots of the meshcat animation at different time steps.**

The PD gains I chose were: $K_P = 1.75$, $K_D = 2.5$, $K_{P,\tau} = 150$, $K_{D,\tau} = 25$

Following is the screenshot of the states as the drone moves from [0,0] to setpoint [1,1]:

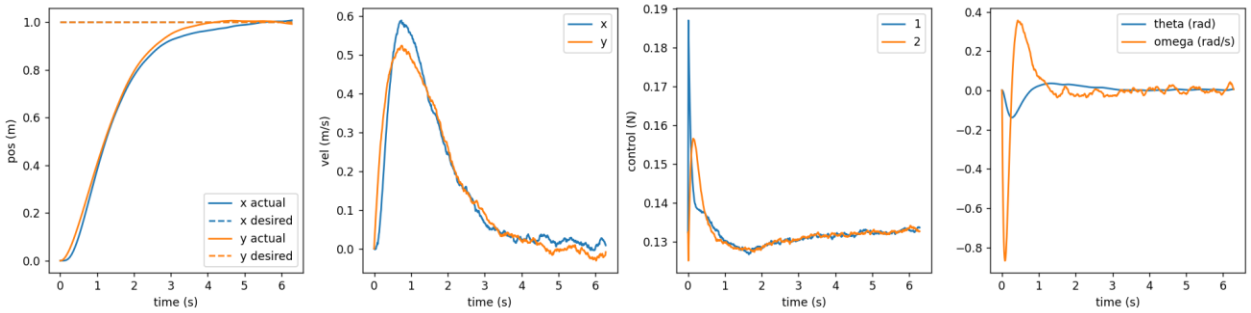
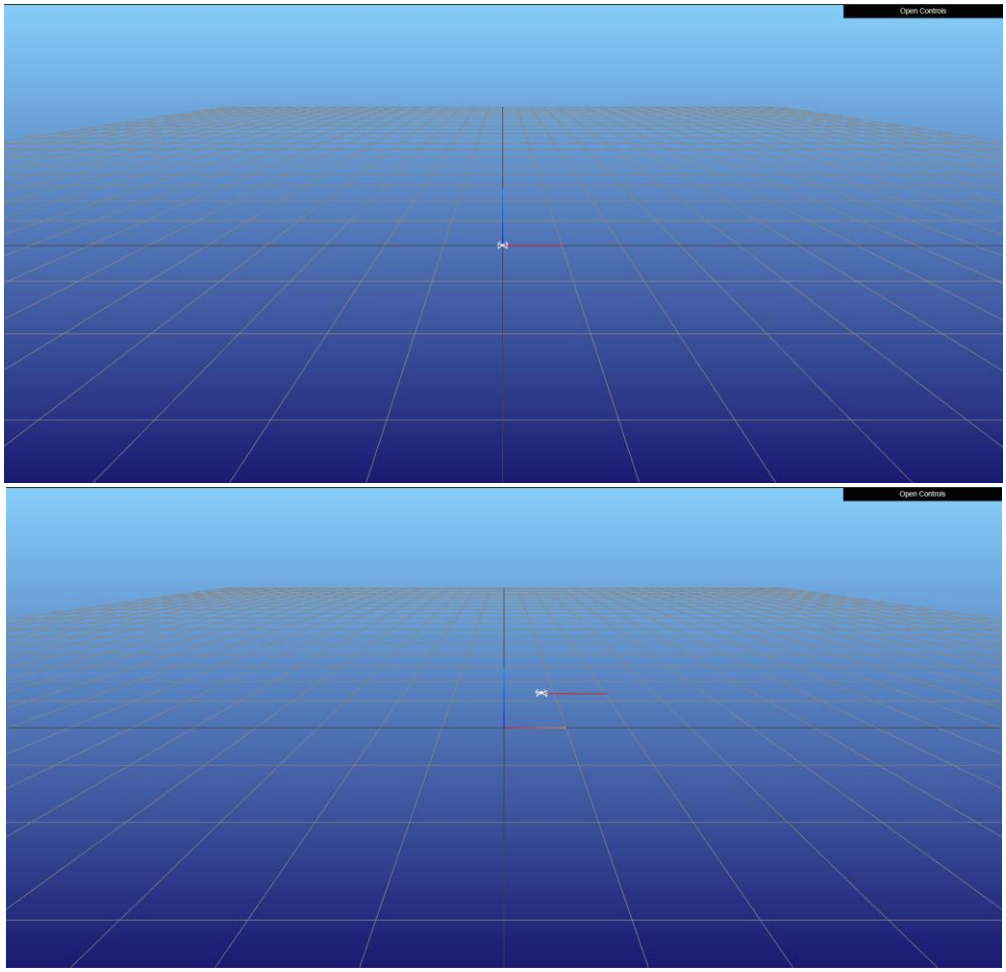
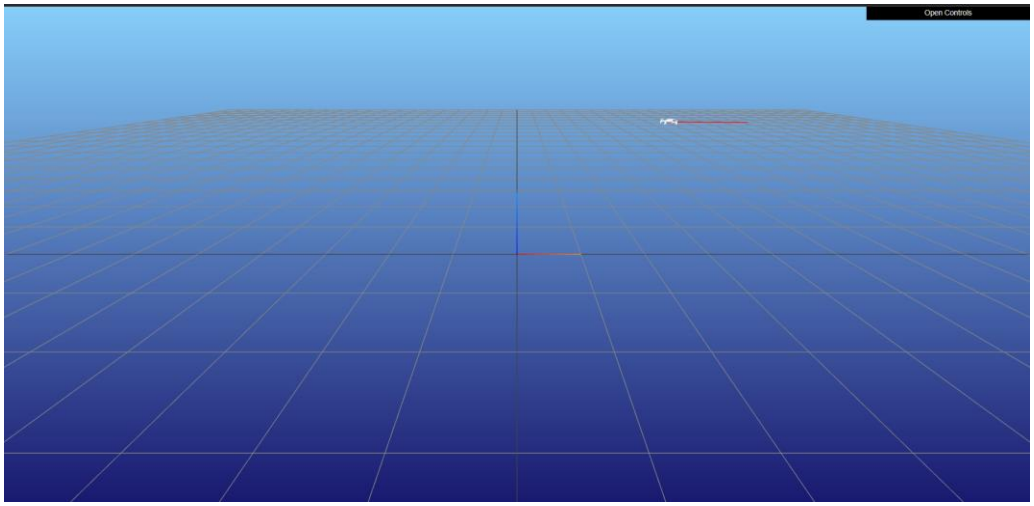


Figure: State and control profiles for setpoint tracking using Cascaded Control from [0,0] to [1,1]

Since we want the robot to stop motion as it approaches the desired setpoint position, hence for this case $p_d = [1,1]$, $v_d = [0,0]$, $a_d = [0,0]$. Following are the three screenshots from the meshcat for this setpoint tracking case:





Similarly, following is the screenshot of the states as the drone moves from [0,0] to setpoint [1,0]:

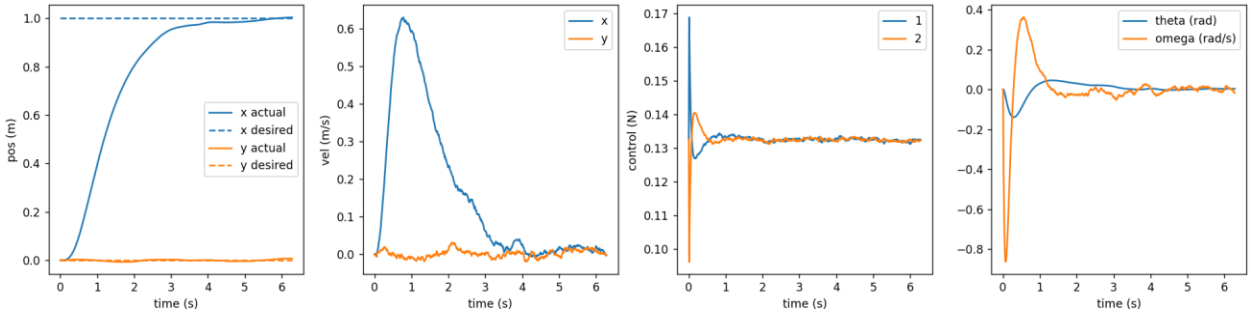
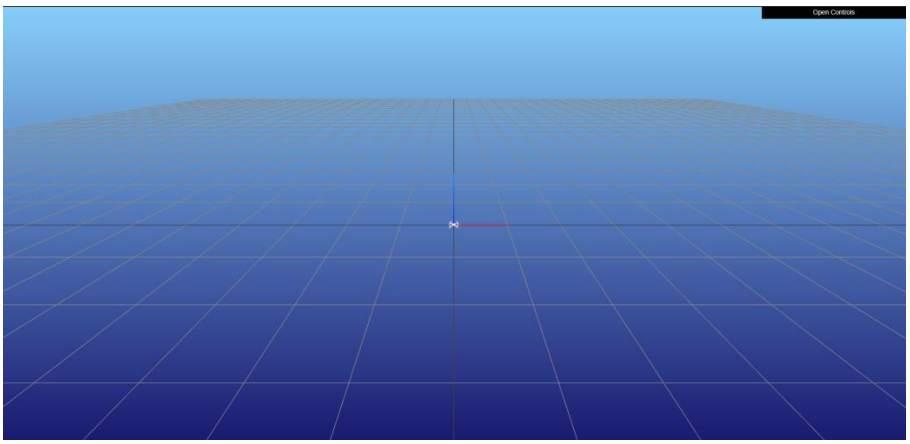
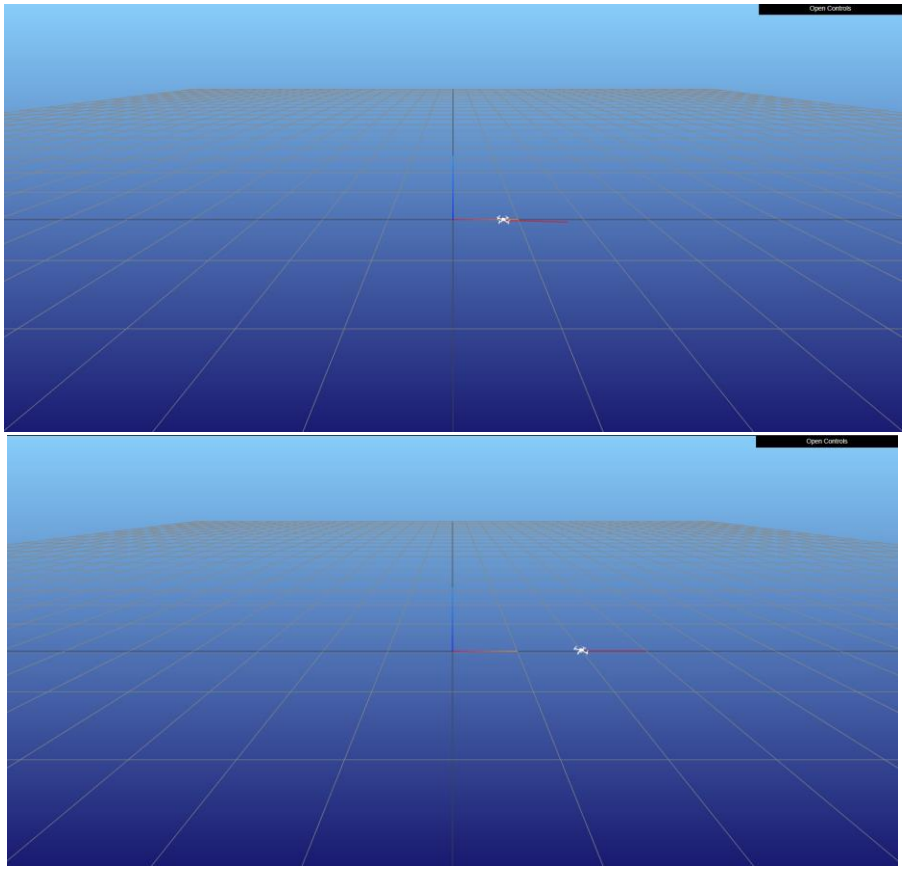


Figure: State and control profiles for setpoint tracking using Cascaded Control from [0,0] to [1,0]

Since we want the robot to stop motion as it approaches the desired setpoint position, hence for this case $p_d = [1,0]$, $v_d = [0,0]$, $a_d = [0,0]$. Following are the three screenshots from the meshcat for this setpoint tracking case:





The differences observed in the state and control profiles for the setpoint tracking from [0,0] to [1,1] and that for the setpoint tracking from [0,0] to [1,0] is as follows:

i. Position trajectories:
   a. In the first scenario ([0,0] to [1,1]), both x and y position of the drone increase simultaneously until they reach the desired setpoint.
   b. In the second scenario ([0,0] to [1,0]), only x position of the drone changes, while the y position remains constant at 0. The drone is essentially moving along a straight line in the x-direction.
   c. The drone in the first scenario moves diagonally from the origin to a point northeast, while in the second plot it moves straight east.

ii. Velocity profiles:
   a. In the first scenario ([0,0] to [1,1]), the velocities in both x and y directions exhibit a peak before gradually settling as the positions reach their setpoints. The velocities are not identical, which supports the observations from the position plot.
   b. In the second scenario ([0,0] to [1,0]), only x velocity shows a significant change. It peaks and then falls back, trying to reach zero as the x position reaches its setpoint.

iii. Control Input:
   a. In the first scenario ([0,0] to [1,1]), there are two distinct control signals. Both have a sharp spike, indicating a strong initial control input to get the system moving. The control inputs then decrease which could be due to reduced necessity for strong controls as the system nears its setpoint.

b. In the second scenario ([0,0] to [1,0]), the control thrust have spikes in the opposite direction, with the peak of the first thruster control having significantly larger magnitude as compared to the second thruster control. This signifies that the 2D drone tilts and then starts moving along a straight line towards the setpoint, and with the control inputs reaching equal magnitudes to stabilize and eventually stop.

iv. Theta and Angular Velocity:

a. In the first scenario ([0,0] to [1,1]), theta shows an initial sharp increase before stabilizing, while omega exhibits a corresponding shape peak followed by oscillations that dampen over time.

b. In the second scenario ([0,0] to [1,0]), theta and omega demonstrate similar behaviors, but the magnitude of the initial peak in omega is larger and its oscillations appear more pronounced.

The diagonal move is more complex (in the first scenario) as it involves simultaneous motion in both x and y directions. This is reflected in both the position and velocity profiles. This also translates to the control effects exhibited for the diagonal motion, suggesting that more thrust or force is required for simultaneous motion in two axes.

- **[5] What PD gains do you choose? Compare the performance of the gain values you choose and one set of other gain values, in terms of the rise time (90%), maximum overshoot, and average control energy (the average of $||u||^2$).**

The PD gains I chose were: $K_P = 1.75$, $K_D = 2.5$, $K_{P,\tau} = 150$, $K_{D,\tau} = 25$

Following is an ablation of the performance of the gain values:

Setpoint	K_P	K_D	$K_{P,\tau}$	$K_{D,\tau}$	Rise time (x)	Rise time (y)	Maximum overshoot (x)	Maximum overshoot (y)	Average Control Energy
[1,1]	1.75	2.5	150	25	~2.22 s	~2.18s	~-0.725%	~0.610%	~0.0352
[1,0]	1.75	2.5	150	25	~2.24s	~0.0s	~-0.51%	~0.0%	~0.0352
[1,1]	1	1	10	10	~1.35s	~1.6s	~74.94%	18.9%	~0.0350
[1,0]	1	1	10	10	~1.31s	~0.0s	~75.03%	~0.0%	~0.0351

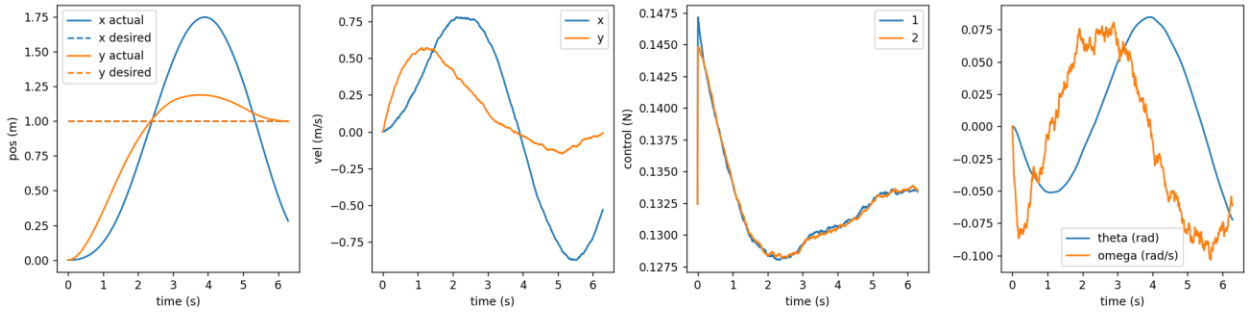


Figure: State and control profiles for For $K_P = 1$ $K_D = 1$ $K_{P,\tau} = 10$ $K_{D,\tau} = 10$, and p_d = [1,1]

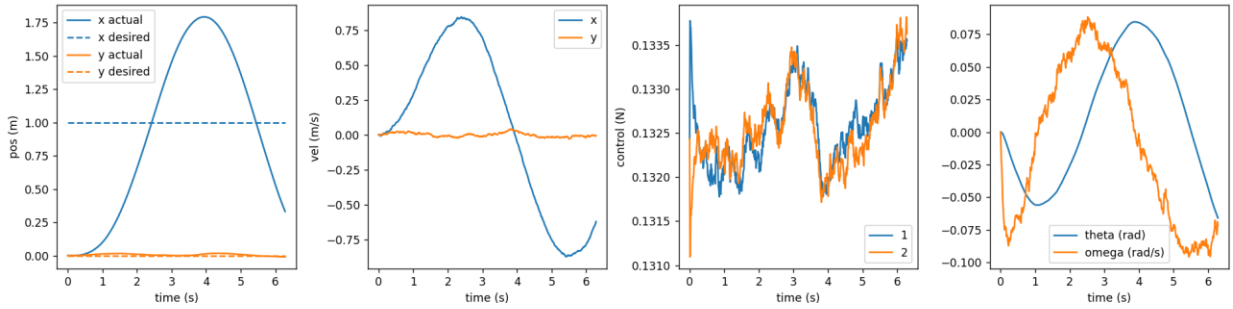


Figure: State and control profiles for $K_P = 1$ $K_D = 1$ $K_{P,\tau} = 10$ $K_{D,\tau} = 10$, and p_d = [1,0]

3. **Linear Control and LQR [10] (and [5] extra points): Based on lecture 3, complete the "linear_control" function in the "Quadrotor_2D" class. Choose diagonal and positive definite Q, R matrix and use the "lqr" function in "control" library to compute the K matrix. Like problem 2, run "python 2d.py 3" to track two setpoints: [1, 1] and [1, 0].**

Ans: The implementation of the 'linear_control' function can be summarized into a few key steps, which are as follows:

- The system dynamics matrices are defined by:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -g & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

  The mass and inertia are not considered in the system dynamics, and are multiplied later due to the values being really small and thus avoiding any instabilities.
- The cost function matrices were slightly tuned to be:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- The optimal control gain K is computed using the LQR function which minimizes the cost function.
- The state error is defined as:

$$state\_error = \begin{bmatrix} position\ error\ x \\ position\ error\ y \\ velocity\ error\ x \\ velocity\ error\ y \\ angular\ position \\ angular\ velocity \end{bmatrix}$$

- Control Action:

$$change\_vector = -K * state\_error$$
$$T = m * (change\_vector[0] + g)$$
$$\tau = change\_vector[1] * J$$

- Control rotor thrusts:

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -arm & arm \end{bmatrix}^{-1} \begin{bmatrix} T \\ \tau \end{bmatrix}$$

- **[10] Report the figure for both setpoints. Quantify and comment on the difference between LQR control and the cascaded setpoint control in problem 2, in terms of the rise time (90%), maximum overshoot, and average control energy.**

Following are the state and control profiles for LQR:

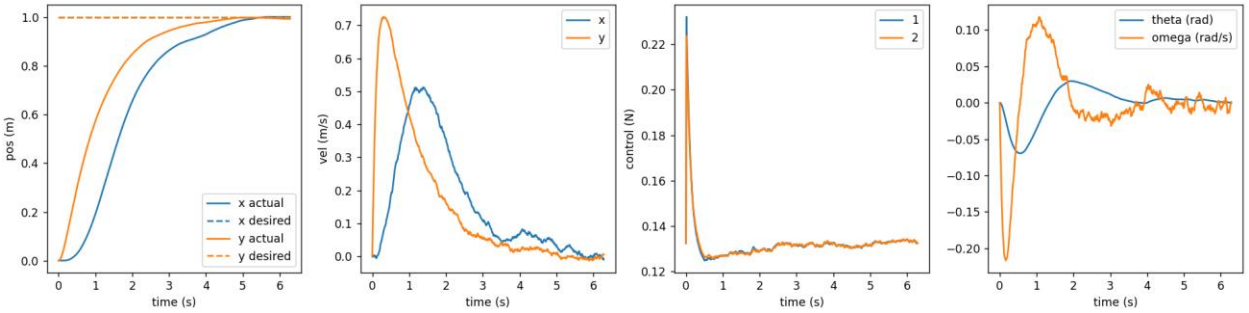


Figure: State and control profiles for p_d = [1,1] using linear control (LQR)

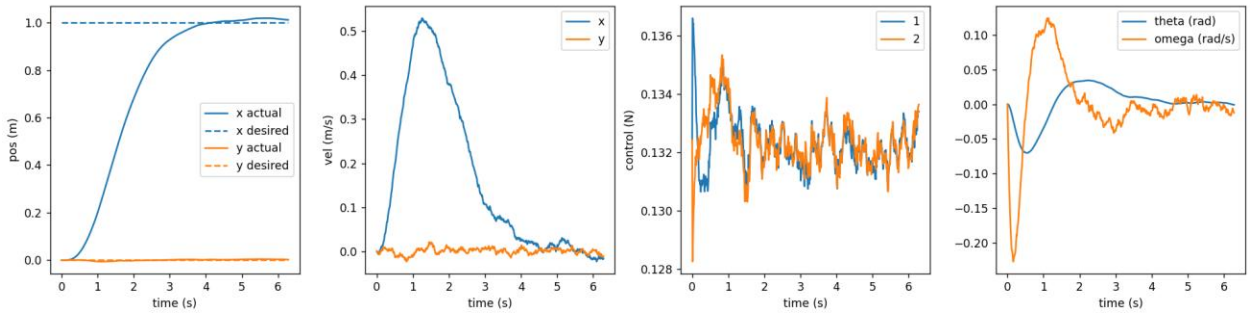


Figure: State and control profiles for p_d = [1,0] using linear control (LQR)

The following table summarizes the rise time and maximum overshoot in X and Y direction, along with average control energy when using LQR controller:

Setpoint (p_d)	Q	R	Rise time (x)	Rise time (y)	Maximum overshoot (x)	Maximum overshoot (y)	Average Control Energy
[1,1]	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	~2.72s	~2.22	~0.086%	~0.311	~0.0352
[1,0]			~2.12s	~0.0s	~1.87%	~0.0%	~0.0352

The following table summarizes the rise time and maximum overshoot in X and Y direction, along with average control energy when using cascaded controller:

Setpoint	K_P	K_D	$K_{P,\tau}$	$K_{D,\tau}$	Rise time (x)	Rise time (y)	Maximum overshoot (x)	Maximum overshoot (y)	Average Control Energy
[1,1]	1.75	2.5	150	25	~2.22 s	~2.18s	~-0.725%	~0.610%	~0.0352
[1,0]	1.75	2.5	150	25	~2.24s	~0.0s	~-0.51%	~0.0%	~0.0352

It can be observed from the above tables that:

i. For the setpoint [1,1], the rise times are almost identical between the two methods, while for the setpoint [1,0], the LQR control has a faster rise time for x, while both methods achieve an instantaneous rise time for y with minimal oscillations.

ii. For the setpoint [1,1], the cascaded control undershoots for x, while LQR control has a very minimal positive overshoot. For y, both methods exhibit overshoot with cascaded control having a higher value. Similarly, for the setpoint [1,0], LQR control shows a higher positive overshoot for x compared to the undershoot from cascaded control. For y, both methods have nearly negligible overshoot.

iii. The average control energy is identical for both methods across the two setpoints.

- **([5] extra points) What Q and R do you choose? Scale the R matrix by 0.1 and 10 and elaborate what happens. Change the ratio between pos/vel error and theta/omega error in the Q matrix and elaborate what happens. Moreover, initialize the drone with a large angle ($\theta = \pi/3$), is LQR still good (compare to the cascaded control method in problem A-2)?**

The choice I made for Q and R for this question (i.e., question 3 of part A) are:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The R matrix in the LQR represents the weightings on the control effort in the cost function, where it is used to penalize control effort, while the Q matrix is used to penalize state deviations. The optimization problem that LQR tries to solve is essentially a trade-off between minimizing state deviations (as weighted by Q) and minimizing control effort as weighted by R.

Following are the state and control profiles when matrix R is scaled by 0.1:

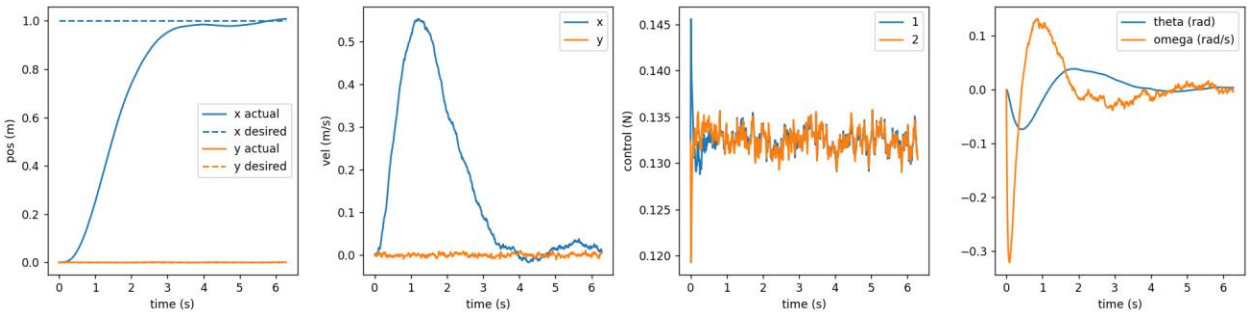


Figure: State and control profiles for p_d = [1,0] using linear control (LQR) R scaled by 0.1

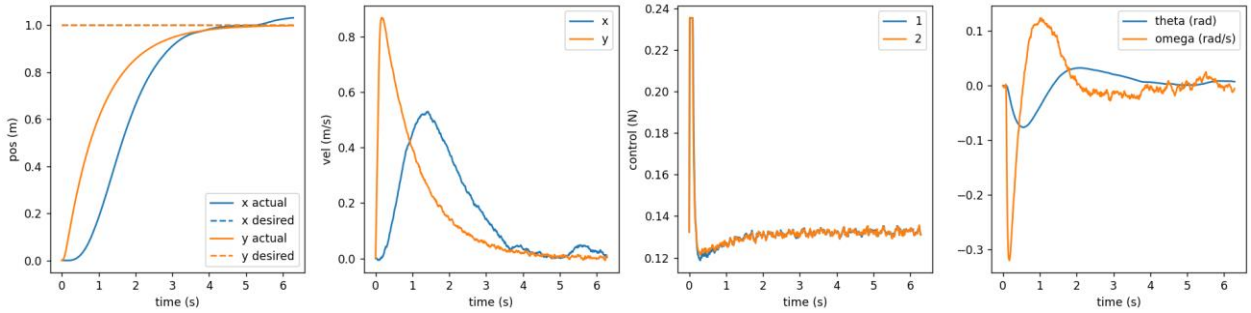


Figure: State and control profiles for p_d = [1,1] using linear control (LQR) R scaled by 0.1

Comparing the above state and control profiles with those produced without the scaling, it is observed that:

i. This reduces the penalty on control effort.
ii. The controller is more aggressive as it prioritizes state error reduction over control effort.
iii. The result leads to faster risetime , but the control inputs are larger leading to more energy consumption.
iv. It introduces overshoot due to the aggressive nature of the control.

Now, following are the state and control profiles when matrix R is scaled by 10:

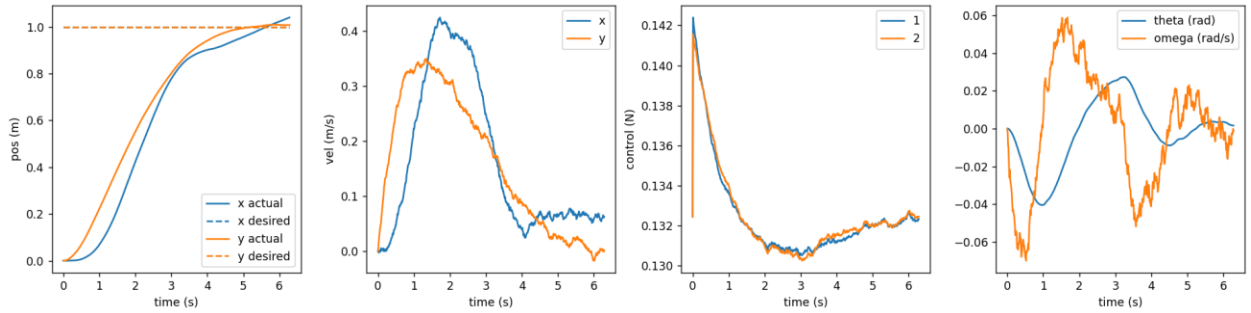


Figure: State and control profiles for p_d = [1,0] using linear control (LQR) R scaled by 10

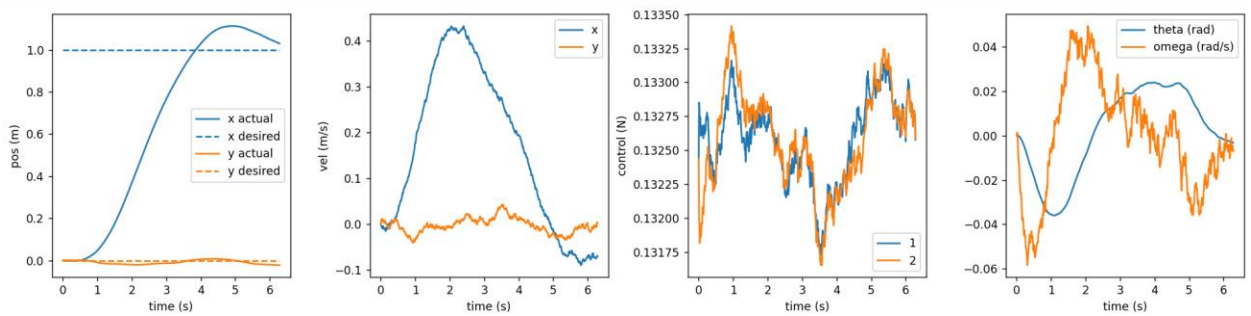


Figure: State and control profiles for p_d = [1,1] using linear control (LQR) R scaled by 10

Comparing the above state and control profiles with those produced without the scaling, it is observed that:

- This increases the penalty on control effort.
- The controller becomes more conservative as it prioritizes minimizing control effort over state error reduction.
- The response is slower and smoother with reduced control.
- Larger steady-state error and longer settling time is observed since the controller is reluctant to apply higher control inputs.

The observations made above for the R matrix scaling is further bolstered by the following table:

Setpoint (p_d)	Q	R	Rise time (x)	Rise time (y)	Maximum overshoot (x)	Maximum overshoot (y)	Average Control Energy
[1,1]	$\begin{bmatrix}1&0&0&0&0&0\\0&5&0&0&0&0\\0&0&1&0&0&0\\0&0&0&5&0&0\\0&0&0&0&1&0\\0&0&0&0&0&5\end{bmatrix}$	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	~2.72s	~2.22	~0.086%	~0.311	~0.0352
[1,0]			~2.12s	~0.0s	~1.87%	~0.0%	~0.0352
[1,1]	$\begin{bmatrix}1&0&0&0&0&0\\0&5&0&0&0&0\\0&0&1&0&0&0\\0&0&0&5&0&0\\0&0&0&0&1&0\\0&0&0&0&0&5\end{bmatrix}$	$\begin{bmatrix}0.1&0\\0&0.1\end{bmatrix}$	~2.13s	~2.19s	~3.066%	~-0.238%	~0.0355
[1,0]			~1.94s	~0.0s	~0.85%	~0.0	0.0355
[1,1]	$\begin{bmatrix}1&0&0&0&0&0\\0&5&0&0&0&0\\0&0&1&0&0&0\\0&0&0&5&0&0\\0&0&0&0&1&0\\0&0&0&0&0&5\end{bmatrix}$	$\begin{bmatrix}10&0\\0&10\end{bmatrix}$	~2.84s	~2.98s	~3.99%	0.91%	0.0351
[1,0]			~2.2s	~0.0s	11.24%	0%	0.0351

Now, coming to the Q matrix which represents the weightings on the state errors in the cost function for the LQR controller. In this matrix:

i. The terms on the diagonal associated with the position and velocity errors are the first four elements.
ii. The terms on the diagonal associated with the angular position and angular velocity errors are the fifth and sixth elements.

My intuition dictates that:

i. To increase the emphasis on position and velocity errors relative to theta and omega error weightings, I will adjust the diagonal elements as follows:
   a. Increase the values of the first four diagonal elements.
   b. Decrease the values of the last two diagonal elements.
ii. To increase the emphasis on theta/omega errors relative to position and velocity errors:
   a. Decrease the values of the first four diagonal elements.
   b. Increase the values of the last two diagonal elements.

Case 1- Prioritizing position and velocity error: Following are the state and control profiles for increased emphasis on position and velocity errors relative to theta and omega errors, the first four elements of the diagonal are increased to 50, and the last two elements of the diagonal are decreased to 0.05, i.e.

$$Q = \begin{bmatrix} 50 & 0 & 0 & 0 & 0 & 0 \\ 0 & 50 & 0 & 0 & 0 & 0 \\ 0 & 0 & 50 & 0 & 0 & 0 \\ 0 & 0 & 0 & 50 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.05 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.05 \end{bmatrix}$$

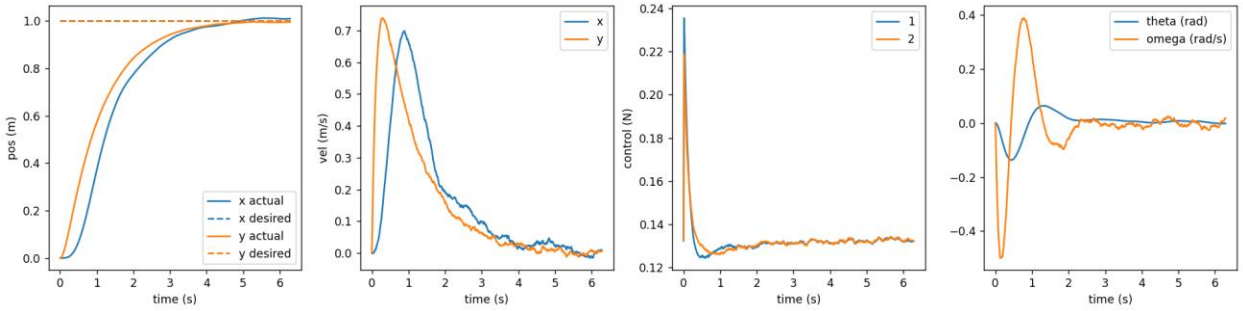


Figure: State and control profiles for p_d = [1,1] using linear control (LQR) Prioritizing position and velocity error

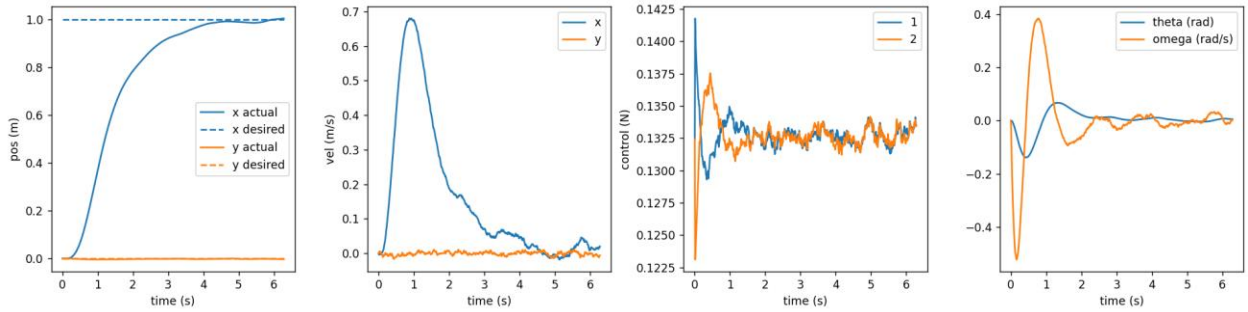


Figure: State and control profiles for p_d = [1,0] using linear control (LQR) Prioritizing position and velocity error

Here is can clearly be observed (if juxtaposed with the results for the original Q matrix) that there control for position and velocity is much smoother and within bounds, while the theta and omega increases ~4 folds.

Case 2- Prioritizing theta and omega error: Following are the state and control profiles for increased emphasis on theta and omega errors relative to position and velocity errors, the first four elements of the diagonal are decreased to 0.05, and the last two elements of the diagonal are increased to 50, i.e.

$$Q = \begin{bmatrix} 0.05 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.05 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.05 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.05 & 0 & 0 \\ 0 & 0 & 0 & 0 & 50 & 0 \\ 0 & 0 & 0 & 0 & 0 & 50 \end{bmatrix}$$

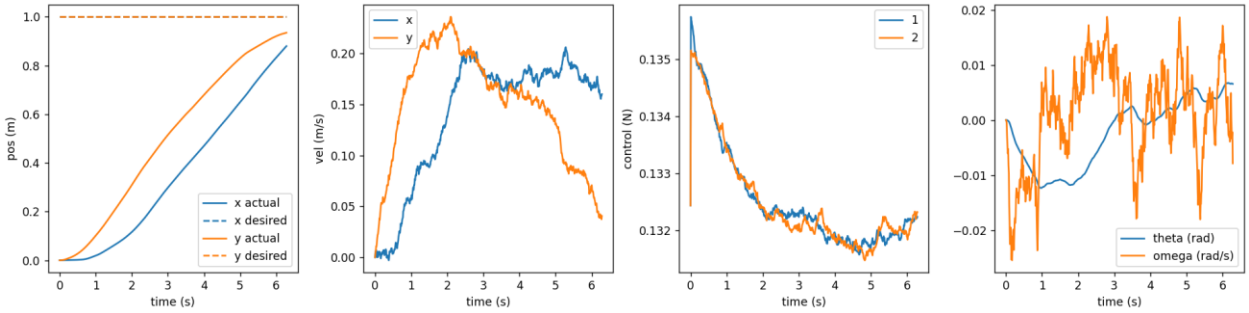


Figure: State and control profiles for p_d = [1,1] using linear control (LQR) Prioritizing theta and omega error

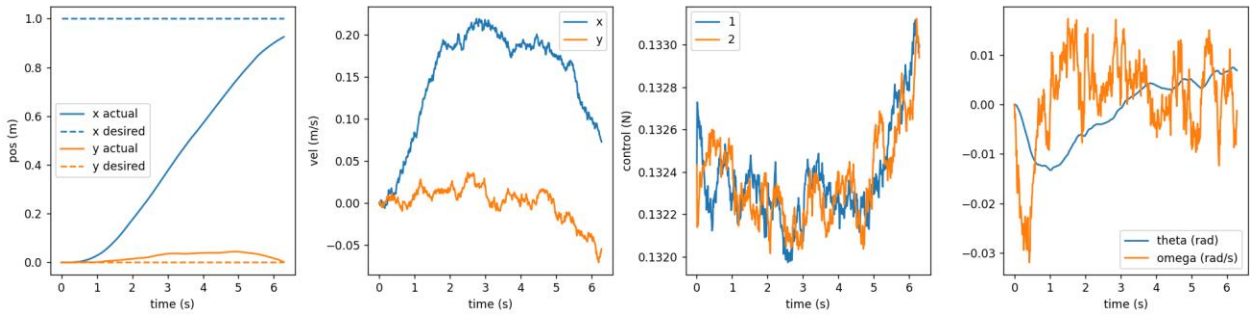


Figure: State and control profiles for p_d = [1,0] using linear control (LQR) Prioritizing theta and omega error

In contrast to the prior case, the controller now grapples with maintaining accurate tracking for position and velocity states. The relaxed weighting in the Q matrix translates to a noticeable struggle in reaching setpoints for these states. The enhanced priority on theta and omega is evident. These states exhibit significantly reduced variations when juxtaposed against results from the original matrix, indicating the controller's focused efforts to tightly control these parameters.

Now, setting the theta to become $\pi/3$, following are the results I get for Cascaded control and LQR respectively for setpoint tracking of [1,1]:

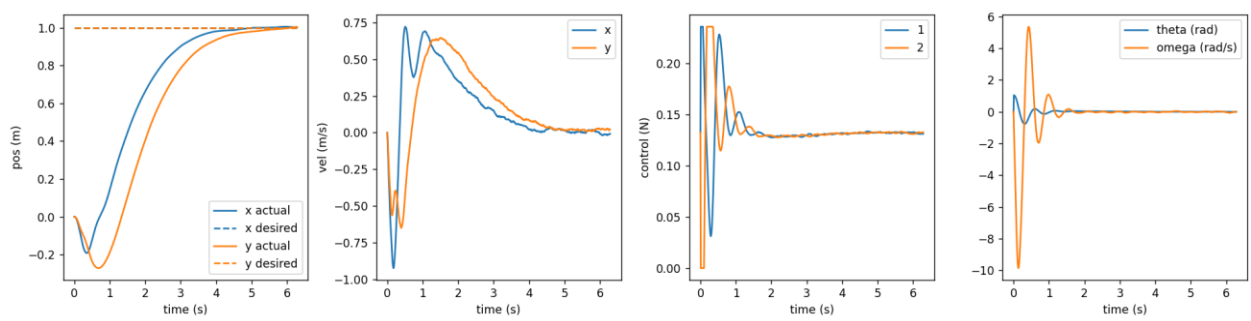


Figure: State and control profiles for p_d = [1,1] using cascaded control with $\theta = \pi/3$

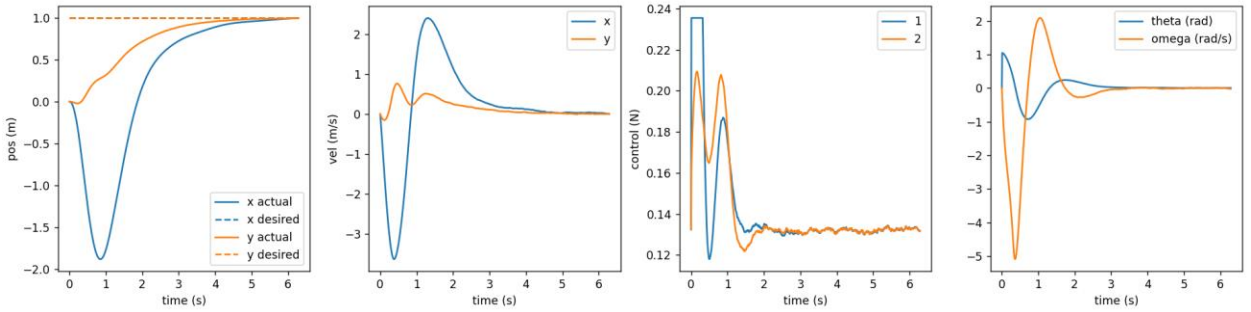


Figure: State and control profiles for p_d = [1,1] using LQR with $\theta = \pi/3$

Similarly, following are the results I get for Cascaded control and LQR respectively for setpoint tracking of [1,0]:

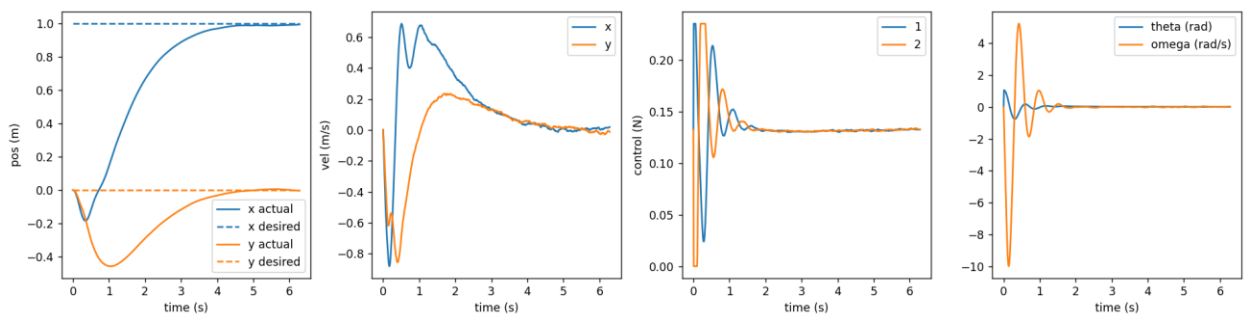


Figure: State and control profiles for p_d = [1,0] using cascaded control with $\theta = \pi/3$

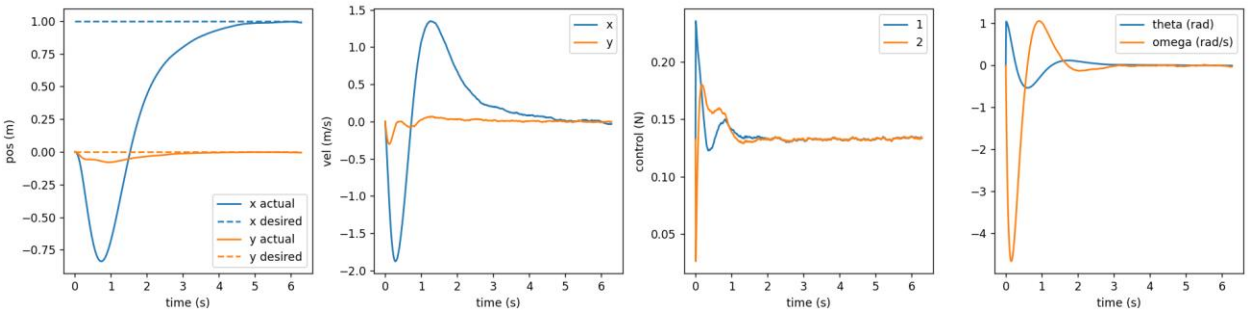


Figure: State and control profiles for p_d = [1,0] using LQR with $\theta = \pi/3$

From the above graphs, following comparison conclusions can be deduced:

i. LQR, by design, being optimal for linear systems when initialized with a large θ, the dynamics become notably non-linear hence degrading the performance for LQR. On the other hand, cascaded control method in our case handle large initial conditions much better.

ii. LQR controller minimizes not just the state error but also the control effort. When subjected to a large initial θ, the LQR produces control signal that are conservative in nature aiming to bring the system to the desired state with minimal energy. The cascaded control method in our case is more aggressive and brings the drone back to the desired state more swiftly but at a trade off of larger control inputs.

iii. Our cascaded control method has an inner and outer loop specifically designed for attitude control and position control. This segregation makes it much more robust to large disturbances in specific states during initialization. In contrast, LQR treats all states with the weights given in the Q matrix, which is not generalized well for robustness across large initial conditions.

iv. Our LQR design was linearized around the hovering position, and when we deviate the system far from this point the LQR doesn't perform optimally. The cascaded control on the other hand have a broader region of attraction.

iv. *Cascaded Tracking Control [10]:* Based on lecture 3, use the "cascaded_control" function in the "Quadrotor_2D" class to track a figure-8 trajectory: $px = \sin(t)$, $py = 1\,2\cos(2t + \pi/2)$. Run "python 2d.py 4"

Ans: In this question, rather than setpoint tracking we are asked to track a trajectory instead. For that reason, following time variant p_d, v_d, and a_d are used:

$$p_x = \sin(t)\,;\; p_y = \frac{1}{2}\cos\left(2t + \frac{\pi}{2}\right)$$
$$v_x = \cos(t)\,;\; v_y = -\sin\left(2t + \frac{\pi}{2}\right)$$
$$a_x = -\sin(t)\,;\; a_y = -2\cos\left(2t + \frac{\pi}{2}\right)$$

- **Report the figure, and compute the controller performance, using position RMSE (rooted mean squared error) as the metric.**

The PD gains I chose were: K_P = 1.75, K_D = 2.5, $K_{P,\tau}$ = 150, $K_{D,\tau}$ = 25

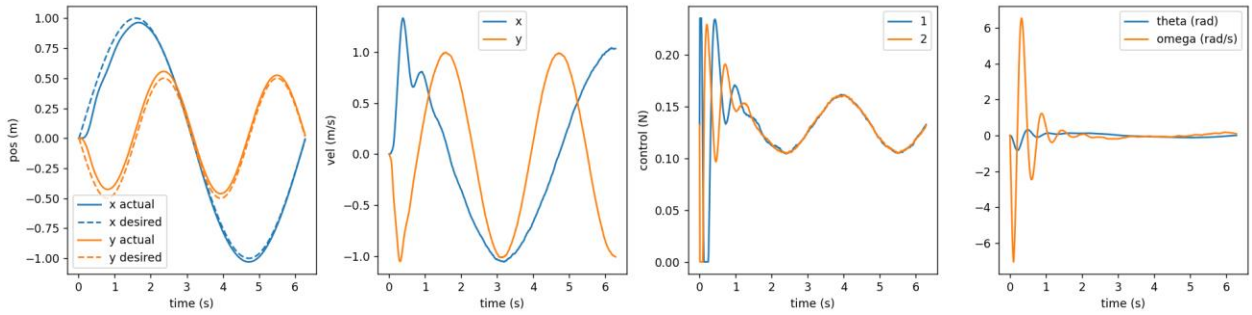


Figure: State and control profiles Vanilla cascaded trajectory tracking controller

Experiment	RMSE for x	RMSE for y	Combined RMSE
Vanilla cascaded trajectory tracking controller	0.062	0.056	0.084

- **Remove the acceleration feedforward term ad in the controller. Elaborate what happens and explain why. Further set $vd = 0$ in the controller and elaborate and explain what the difference is. For both cases, report figures and RMSE numbers.**

The PD gains I chose were: K_P = 1.75, K_D = 2.5, $K_{P,\tau}$ = 150, $K_{D,\tau}$ = 25

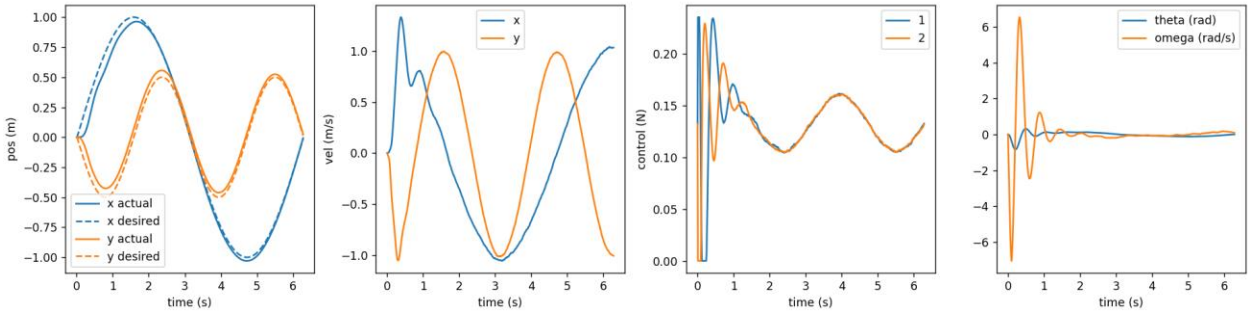


Figure: State and control profiles Vanilla cascaded trajectory tracking controller

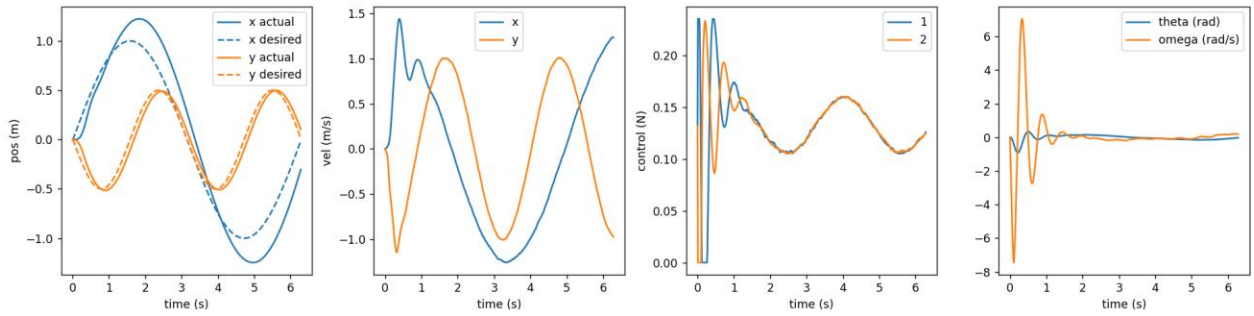


Figure: State and control profiles cascaded trajectory tracking controller with a_d removed

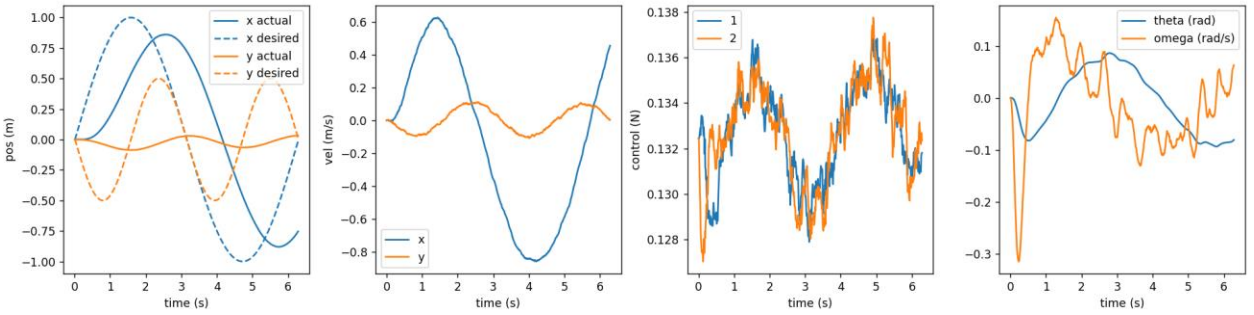


Figure: State and control profiles cascaded trajectory tracking controller with a_d and v_d removed

And following is the comparison of RMSE for tracking:

Experiment	RMSE for x	RMSE for y	Combined RMSE
Vanilla cascaded trajectory tracking controller	0.062	0.056	0.084
cascaded trajectory tracking controller with a_d removed	0.241	0.071	0.251
cascaded trajectory tracking controller with a_d and v_d removed	0.554	0.356	0.659

Cascaded control without acceleration feedforward (a_d):

i. The feedforward control provides the system with knowledge about future changes in the trajectory. By removing a_d, the controller only reacts to the errors in position and velocity without anticipating future changes due to acceleration.
ii. Without the feedforward acceleration, there is an increased tracking error, especially when there are sudden directionally changes along the trajectory.
iii. The controller is only reacting to the errors, particularly when desired trajectory changes rapidly. The controller is just chasing the trajectory rather than anticipating it.
iv. The agility and responsiveness degrades and the drone is sluggish in response to sharp trajectory changes.
v. Compared with the original results for trajectory tracking with the feedforward term, the velocity profiles exhibited showcases the oscillations.

Cascaded control without acceleration feedforward (a_d) and $v_d = 0$:

- Setting $v_d = 0$ esentially aims for a hover or stationary position. The controller now no longer tries to track a moving trajectory, but rather aims to maintain a specific point.
- The drone now aims to maintain a stable hover, reducing the need for significant tilts and changes in orientation.
- Control inputs are primarily concerned with counteracting disturbances and maintaining a hover, as opposed to following a trajectory.
- The desired trajectory is ignored, and the drone barely attempted to move along it due to the zero desired velocity, potentially ignoring any provided path.

v. ***Trajectory Generation and Differential Flatness [10] (and [5] extra points):*** **Based on lecture 5, design a polynomial reference trajectory from [0, 0] to [1, 0]. Then using the "cascaded_control" function in the "Quadrotor_2D" class to track this trajectory. Use differential flatness to compute ωd, τd for "cascaded_control". Run "python 2d.py 5" for this question.**

Ans: The first step towards solving this problem is to use polynomial trajectory generation. This technique is useful for our case especially since our drone will move smoothly between two waypoints i.e. [0,0] and [1,0]. Following is the step-by-step procedure used for generating the determining this polynomial:

i. Setup:
   a. We have a 2D drone that can track any smooth trajectory with a bounded acceleration (a), jerk (j), and snap (s).
   b. The idea is to use polynomials to connect waypoints.
   c. The goal is to find a polynomial p(t) that connects point pa (hovering at origin [0,0]) to pb (hovering at destination [1,0]).

ii. Constraints:
   a. Initial conditions: $\mathrm{p}(0) = \mathrm{p}_a,\ \mathrm{p'}(0) = 0, \mathrm{p''}(0) = 0, \mathrm{p'''}(0) = 0$
   b. Final conditions: $\mathrm{p}(\mathrm{T}) = \mathrm{p}_b, \mathrm{p'}(\mathrm{T}) = 0, \mathrm{p''}(\mathrm{T}) = 0, \mathrm{p'''}(\mathrm{T}) = 0, \mathrm{p''''}(\mathrm{T}) = 0$

iii. Choice of degree of polynomial:
   a. Given the 8 constraints (4 initial and 5 final), an 8th degree polynomial is selected. The polynomial is represented as:

$$p(t) = \sum_{i=0}^{8} a_i t^i ------(a)$$

With that setup, we can start designing the polynomial reference trajectory from [0,0] to [1,0]:

Given:

$$\mathrm{p}_a = [0,0]$$

$$\mathrm{p}_b = [1,0]$$

The constraints are:

$$p(0) = 0$$

$$p'(0) = 0$$

$$p''(0) = 0$$

$$p'''(0) = 0$$

$$p(T) = 1$$

$$p'(T) = 0$$

$$p''(T) = 0$$

$$p'''(T) = 0$$

$$p''''(T) = 0$$

Using eq(a) and all the mentioned constraints, we obtain:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 1 & t & t^2 & t^3 & t^4 & t^5 & t^6 & t^7 & t^8 \\ 0 & 1 & 2t & 3t^2 & 4t^3 & 5t^4 & 6t^5 & 7t^6 & 8t^7 \\ 0 & 0 & 2 & 6t & 12t^2 & 20t^3 & 30t^4 & 42t^5 & 56t^6 \\ 0 & 0 & 0 & 6 & 24t & 60t^2 & 120t^3 & 210t^4 & 336t^5 \\ 0 & 0 & 0 & 0 & 24 & 120t & 360t^2 & 840t^3 & 1680t^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this by hand would be been tedious, hence, it was solved using a python script which makes use of np.linalg.solve to get $a_0 \dots a_8$.

The above one is just for px, and can be repeated to get py using:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 1 & t & t^2 & t^3 & t^4 & t^5 & t^6 & t^7 & t^8 \\ 0 & 1 & 2t & 3t^2 & 4t^3 & 5t^4 & 6t^5 & 7t^6 & 8t^7 \\ 0 & 0 & 2 & 6t & 12t^2 & 20t^3 & 30t^4 & 42t^5 & 56t^6 \\ 0 & 0 & 0 & 6 & 24t & 60t^2 & 120t^3 & 210t^4 & 336t^5 \\ 0 & 0 & 0 & 0 & 24 & 120t & 360t^2 & 840t^3 & 1680t^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Once we have our trajectory polynomial, we can determine desired velocity and desired acceleration simply by differentiating them w.r.t. time "t".

Now unlike in Question A2 – A4, where ω_d and τ_d were set to zero, here we use differential flatness to determine them on the run:

$$\omega_d = -\frac{J^T x}{T}$$

$$\tau_d = -\frac{s^T x + 2J^T y\omega}{T}$$

- **[5] Report the figure and RMSE results with two terminal times: 2*pi seconds (the default value) and 2 seconds, by changing the "total_time" variable. Compare the tracking results with the result from tracking a "naïve" reference trajectory with constant velocity ($px = t/total\_time$), for both terminal times. What do you observe and why?**

Following is the figure for the terminal time 2*pi seconds using polynomial trajectory generation:

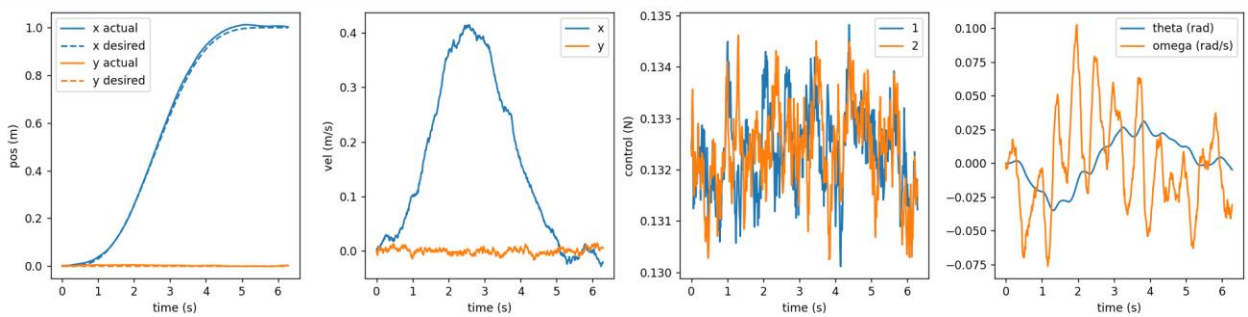


Figure: State and control profiles cascaded controller using polynomial trajectory generation for 2π sec

Following is the figure for the terminal time 2 seconds using polynomial trajectory generation:

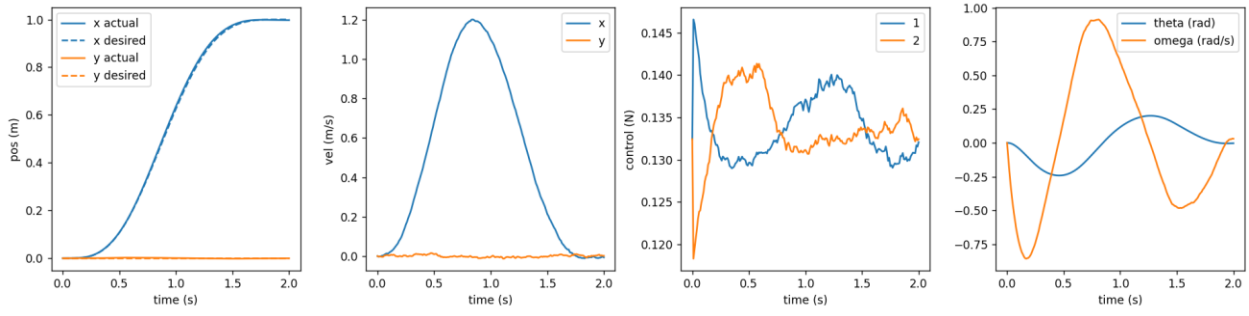


Figure: State and control profiles cascaded controller using polynomial trajectory generation for 2sec

Terminal time (sec)	RMSE for x	RMSE for y	Combined RMSE
2π	0.0106	0.0023	0.0108
2	0.00571	0.0013	0.0058

Following is the figure for the terminal time 2*pi seconds using naïve trajectory (t/total_time):

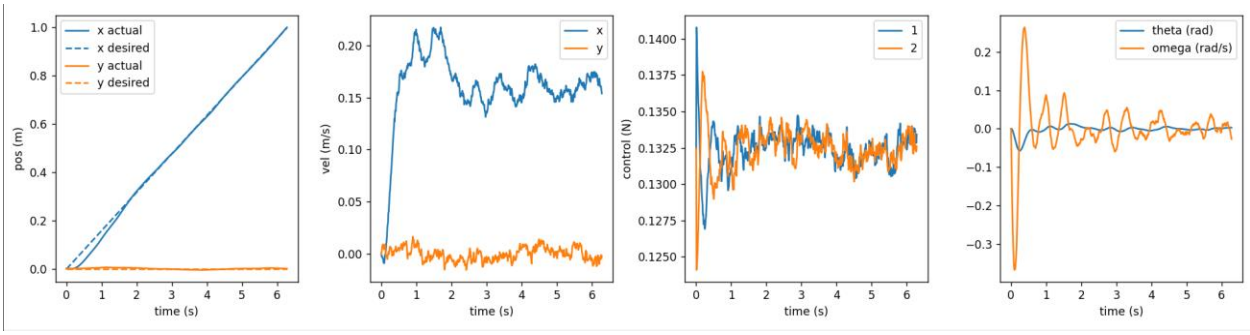


Figure: State and control profiles cascaded controller using naïve trajectory generation for 2π sec

Following is the figure for the terminal time 2 seconds using t/total_time:

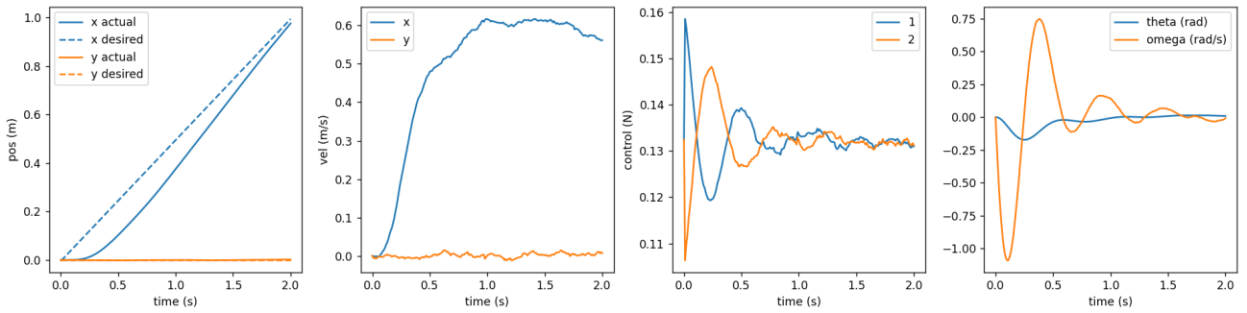


Figure: State and control profiles cascaded controller using naïve trajectory generation for 2 sec

The following table summarizes performance across the two terminal times:

Terminal time (sec)	RMSE for x	RMSE for y	Combined RMSE
2π	0.0168	0.0029	0.0171
2	0.0997	0.0009	0.0997

Given the results above, following are my observations from state and control profile comparisons, and reasoning behind the reason for the results:

i. The polynomial trajectory tracking is tighter and more accurate as compared to the naïve trajectory, but in essence requires more control effort, especially with a shorter terminal time.

ii. The naïve trajectory, being simpler, requires lesser control effort, reflect in smoother control inputs. This is expected since a constant velocity trajectory is inherently less demanding.

iii. The polynomial trajectory requires more changes in orientation and has higher angular velocities, especially with shorter terminal times. This is due to the non-linear nature of the polynomial trajectory.

iv. A shorter terminal time compresses the trajectory, demanding higher velocities and control efforts in a shorter time frame, which is evident in sharper peaks in the velocity and control plots.

- **[5] For both terminal times (2*pi seconds and 2 seconds), use the "cascaded_control" function with ω_d = 0, τ_d = 0 to track the designed polynomial reference trajectory. Report the figure and compare the result with the differential-flatness-based method used in the last question. What is the difference and why?**

In this question, ω_d = 0, τ_d = 0 are set to zero to track the designed polynomial reference trajectory. Following is the result for terminal time 2π seconds:

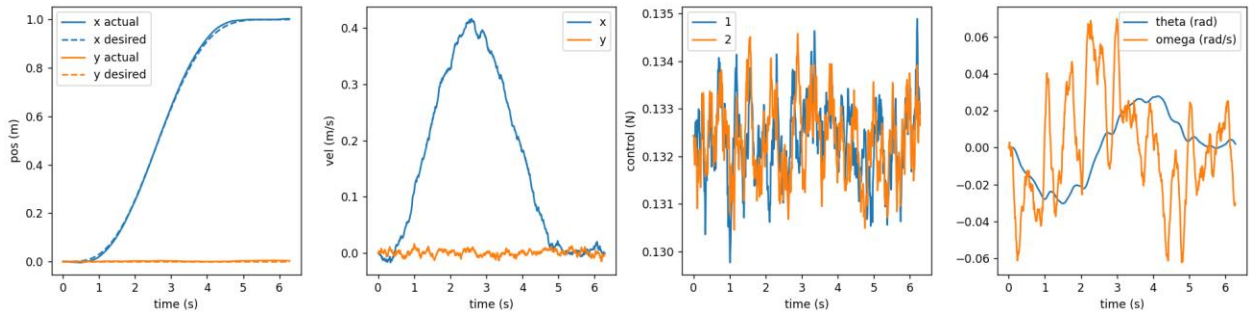


Figure: State and control profiles cascaded controller using polynomial trajectory generation for 2π sec and ω_d = 0, τ_d = 0

Following is the result for terminal time 2 seconds:

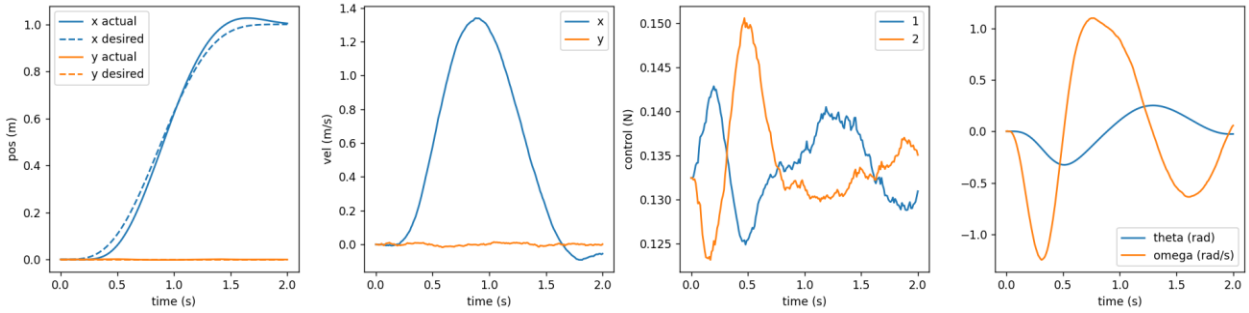


Figure: State and control profiles cascaded controller using polynomial trajectory generation for 2 sec and ω_d = 0, τ_d = 0

The following table summarizes performance across the two terminal times when ω_d = 0, τ_d = 0:

Terminal time (sec)	RMSE for x	RMSE for y	Combined RMSE
2π	0.0072	0.0022	0.0075
2	0.0307	0.0009	0.0307

The difference between the results using non-zero ω_d and τ_d (calculated using differential flatness) verses zero ω_d and τ_d is summarized as:

i. RMSE differences:
   a. For a terminal time for 2π seconds, the positional RMSE when using differential flatness is slightly higher as compared to using zero ω_d and τ_d.

b. For a terminal time of 2 seconds, the positional RMSE when using differential flatness is also higher as compared to using zero ω_d and τ_d;

ii. Graphical differences:

a. From the generated state and control profiles, it can be observed that with a terminal time of 2π seconds, the actual trajectory closely follows the desired trajectory for both methods, but there are visible oscillations in the control input when using the differential flatness method.

b. For a terminal time of 2 seconds, the trajectory tracking with zero ω_d and τ_d appears to have more noticeable deviations from the trajectory, especially in the x-direction.

The difference in tracking performance and RMSE values between the two methods can be attributed to the following reasons:

i. Influence of initial derivations: Setting ω_d and τ_d to zero means that the initial angular velocity and torque are disregarded. This can result in less aggressive control actions, leading to smoother control inputs but potentially poorer tracking performance, especially in scenarios with shorter terminal times with sharper directional changes.

ii. Control strategy: Differential flatness-based methods use more information about the system dynamics and the desired trajectory to generate control inputs. This leads to better tracking performance but results in more aggressive control inputs, causing oscillations.

iii. Trajectory complexity: For shorter terminal times (in our case 2 seconds), the trajectory needs to be tracked more aggressively. This could explain the larger deviations seen in the zero ω_d and τ_d case for this duration.

In conclusion, while the differential flatness-based method provides more aggressive control leading to potentially better tracking for complex trajectories, setting ω_d and τ_d to zero offers smoother control but might not perform as well in all scenarios.

Part B

1. System Modeling [15]: Based on lecture 2, complete the "dynamics" function in the "Quadrotor" class. Run "python quadrotor.py 1" to observe the output. We have disabled randomness for this question so you should see results same or similar (up to numerical errors) to this:

```
pos: [0. 0. 0.]
vel: [0.     0.     0.0327]
quaternion: [1. 0. 0. 0.]
omega: [2.97560230e-01 5.01040677e-17 1.54269014e-02]
**********
pos: [ 2.40553113e-06 -1.28359065e-03  1.79510018e-02]
vel: [ 2.05606214e-04 -6.41192722e-02  3.56855592e-01]
quaternion: [9.96644885e-01 8.17371282e-02 1.46189822e-04 4.24186640e-03]
omega: [3.27313302 0.0136595  0.16969186]
**********
pos: [ 0.00019429 -0.02303064  0.06606604]
vel: [ 0.00683764 -0.50468237  0.595935  ]
quaternion: [0.95145924 0.30734012 0.00232097 0.01618546]
omega: [6.24783997 0.101806   0.32383767]
```

. Take a screenshot of your output.

Ans: In this particular question, we are asked to implement the dynamics model for the quadrotor. For that reason, the "dynamics" function in the "Quadrotor_" class was completed and the code was executed to receive the drone's state at index $\mathrm{i} = 0, 10, 20$for a constant input

$$u = np.array([0.006, 0.008, 0.010, 0.012]) * 9.81.$$

We were further asked to juxtapose the results obtained with that shown in the writeup (which should matchup up to numerical errors). Following is the results I got from my implementation of the "dynamics" function:

```
(am_hw3) C:\Users\DELL\Desktop\aerial_mobility_release\aerial_mobility_release>python quadrotor.py 1
**********
pos: [0. 0. 0.]
vel: [0.     0.     0.0327]
quaternion: [1. 0. 0. 0.]
omega: [2.97560230e-01 5.01040677e-17 1.54269014e-02]
**********
pos: [ 2.40553113e-06 -1.28359065e-03  1.79510018e-02]
vel: [ 2.05606214e-04 -6.41192722e-02  3.56855592e-01]
quaternion: [9.96644885e-01 8.17371282e-02 1.46189822e-04 4.24186640e-03]
omega: [3.27313302 0.0136595  0.16969186]
**********
pos: [ 0.00019429 -0.02303064  0.06606604]
vel: [ 0.00683764 -0.50468237  0.595935  ]
quaternion: [0.95145924 0.30734012 0.00232097 0.01618546]
omega: [6.24783997 0.101806   0.32383767]
You can open the visualizer by visiting the following URL:
http://127.0.0.1:7000/static/
```

And following is the touchstone result to be compared against in the writeup:

```
pos: [0. 0. 0.]
vel: [0.     0.     0.0327]
quaternion: [1. 0. 0. 0.]
omega: [2.97560230e-01 5.01040677e-17 1.54269014e-02]
************************
pos: [ 2.40553113e-06 -1.28359065e-03  1.79510018e-02]
vel: [ 2.05606214e-04 -6.41192722e-02  3.56855592e-01]
quaternion: [9.96644885e-01 8.17371282e-02 1.46189822e-04 4.24186640e-03]
omega: [3.27313302 0.0136595  0.16969186]
************************
pos: [ 0.00019429 -0.02303064  0.06606604]
vel: [ 0.00683764 -0.50468237  0.595935  ]
quaternion: [0.95145924 0.30734012 0.00232097 0.01618546]
omega: [6.24783997 0.101806   0.32383767]
```

The mathematical implementation that defines the dynamics of the 2D drone in the "dynamics" function is as follows:

i. Control Input clipping:
To ensure that the control inputs are within the permissible bounds, the control input is clipped using u_{min} and u_{max}.

ii. Convert rotor forces to total thrust and torque:

$$\begin{bmatrix} Total\ thrust \\ Torque \end{bmatrix} = B\ u$$

iii. Translational Dynamics:
   a. Compute the derivative of the position using velocity:

$$p_{dot} = v$$

   b. Gravity force acting on the drone:

$$F_{gravity} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix}$$

   Where m is the mass of the drone and g is the acceleration due to gravity.
   c. Net force on the drone:

$$F_{net} = F_{gravity} + R \times \begin{bmatrix} 0 \\ 0 \\ Total\ thrust \end{bmatrix}$$

   d. Compute the derivative of velocity using the net force:

$$v_{dot} = \frac{F_{net}}{m}$$

iv. Rotational Dynamics (using Euler's equations):
   a. Extract torques from the thrust-torque vector.
   b. Compute the derivative of angular velocity ω using the torques and the moment of inertia matrix J.

$$\omega_{dot} = J^{-1}\left(\tau - \omega \times (J \times \omega)\right)$$

v. Update States:

a. Update position:

$$p_{new} = p_{old} + \Delta t \times p_{dot}$$

b. Update velocity:

$$v_{new} = v_{old} + \Delta t \times v_{dot} + \Delta t \times (\sigma_t \times random_{noise} + d)$$

c. Update quaternion(attitude):

$$q_{new} = qintegrate\,(q_{old}, \omega, \Delta t)$$

d. Update rotation matrix from quaternion:

$$R_{new} = qtoR(q_{new})$$

e. Update angular velocity:

$$\omega_{new} = \omega_{old} + \Delta t \times \omega_{dot} + \Delta t \times \sigma_r \times random\_noise$$

f. Extract Euler angles from rotation matrix

2. Cascaded Control [20]: Based on lecture 3, complete the "cascaded_control" function in the "Quadrotor" class. Design PD gains for position and attitude control. Run "python quadrotor.py 2" to track a setpoint [1, 1, 1], and a spiral trajectory px = sin(2t) , py = cos(2t) − 1, pz = 0.5t.

Ans: We are asked in this question to implement/complete the "cascaded_control" function in the "Quadrotor" class. This is a two layer controller where the outer loop deals with position control, and the inner loop deals with attitude control.

i. Position Control:
   a. Goal: To ensure the drone tracks a desired trajectory in terms of position p_d, velocity v_d, and acceleration a_d.
   b. Error calculation:
      - Position error: $e_p = p - p_d$
      - Velocity error: $e_v = v - v_d$
   c. Control Law for desired force f_d:

$$f_d = g \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - K_P \times e_p - K_D \times e_v + a_d.$$

   Where,
      - g is the gravitational force
      - K_P and K_D are the proportional and derivative gains respectively

ii. Attitude Control:
   a. Goal: Given a desired force direction from the position control, it ensures the drone's orientation is such that the thrust is aligned with this desired force direction.
   b. Orientation representation
      - The desired force direction is represented as z_d and is computed by normalizing the desired force f_d:

$$z_d = \frac{f_d}{\|f_d\|}$$

      - A rotation from drone's current orientation to the desired orientation (aligned with z_d) is represented using Rodrigues rotation formula. Here, n is the axis of rotation and ρ is the angle of rotation.

$$n = e_3 \times z_d$$
$$\rho = \arcsin(\|n\|)$$
$$n_{normalized} = \frac{n}{\|n\|}$$
$$rotation_{vector} = \rho \times \mathrm{n}_{\mathrm{normalized}}$$
$$R_{EB} = rotation.from_{rotvec(rotation_{vector})}.as\_matrix()$$

      - R_{AE} is the rotation about the z-axis by the desired yaw angle yaw_d

$$R_{AE} = Rotation.from_{euler('z', yaw_d)}.as\_matrix()$$

      - And then finally, the overall desired orientation is the combination of yaw rotation R_{AE} and the alignment rotation R_{EB}.

$$R_d = R_{AE} @ R_{EB}$$

This desired orientation R_d is used by the attitude control to compute the orientation error and consequently the control torques to drive the drone to this desired orientation.

c. Error Calculation:
   - The error in the orientation is represented by $R_e = R_d.T \times R$, where R_d is the desired rotation matrix and R is the current rotation matrix of the drone.
   - From R_e, we compute the skew-symmetric error matrix R_{diff}. The skew-symmetric representation is S_{inv}.

d. Control Law for Desired Angular Acceleration α:
   $\alpha = -K_{p,\tau}\, S_{inv} - K_{D,\tau}\,(\omega - \omega_d) + \alpha_d$.

e. Control input calculation:
   - The total control input u consists of the desired thrust T and the torques τ.
   - T is calculated as the projection of f_d onto the drone's current z-axis.
   - The torques τ are derived from the angular acceleration α. The relationship between them involves the inertia J and accounts for the gyroscopic effects.
   - Finally, u is a concatenation of T and τ.

In summary, the cascaded controller aims to make the quadrotor follow a given trajectory in terms of position, velocity, and acceleration while also ensuring the drone's orientation is consistent with the required movement which is achieved by an outer loop of position control (generated desired force direction), and the inner loop of attitude controller (makes sure the drone aligns with the direction).

- **For the setpoint tracking task, first set the desired yaw angle (ψd) to be 0 and try a time-variant desired yaw angle ($\psi d = t\ totaltime * \pi$ 3). Report both figures and compare the position RMSE and average control energy. For both cases, take three screenshots of the meshcat animation at different time steps.**

The PD gains that I chose are:

i. K_p = 2.5
ii. K_D = 2.5
iii. K_{Ptau} = 200
iv. K_{Dtau} = 20

Following is the graph for state and control profiles for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle Ψ_d=0:

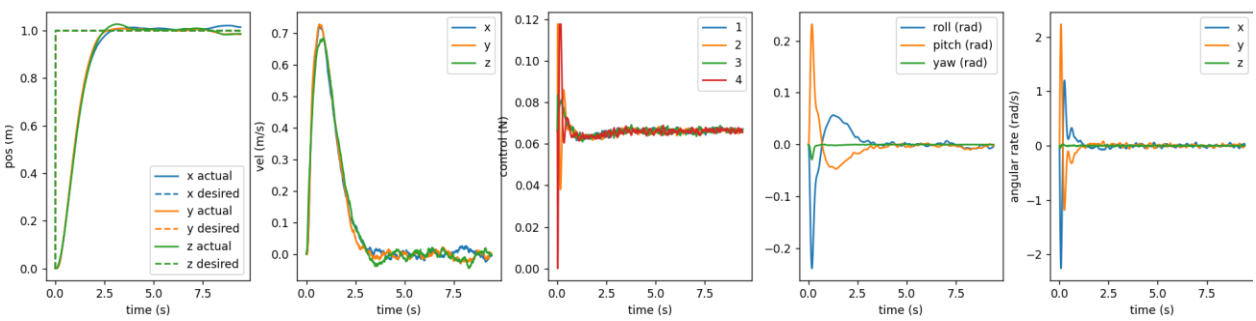


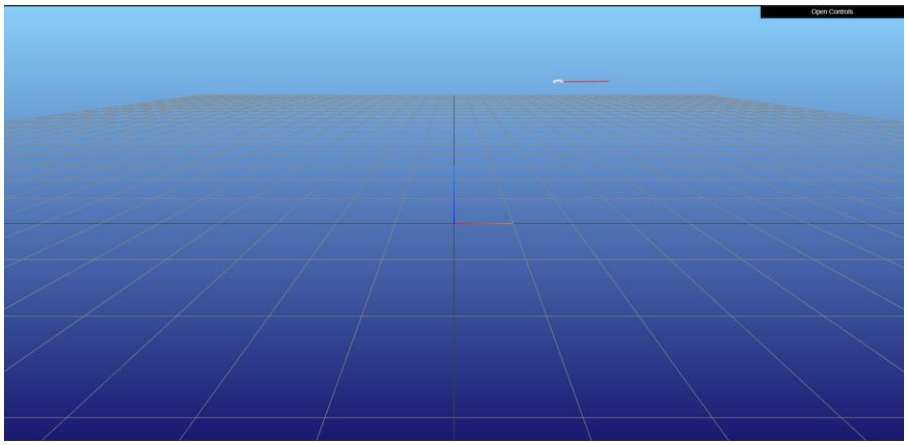
Figure: State and control profiles for cascaded control for Ψ_d=0 for setpoint tracking from [0,0,0] to [1,1,1]

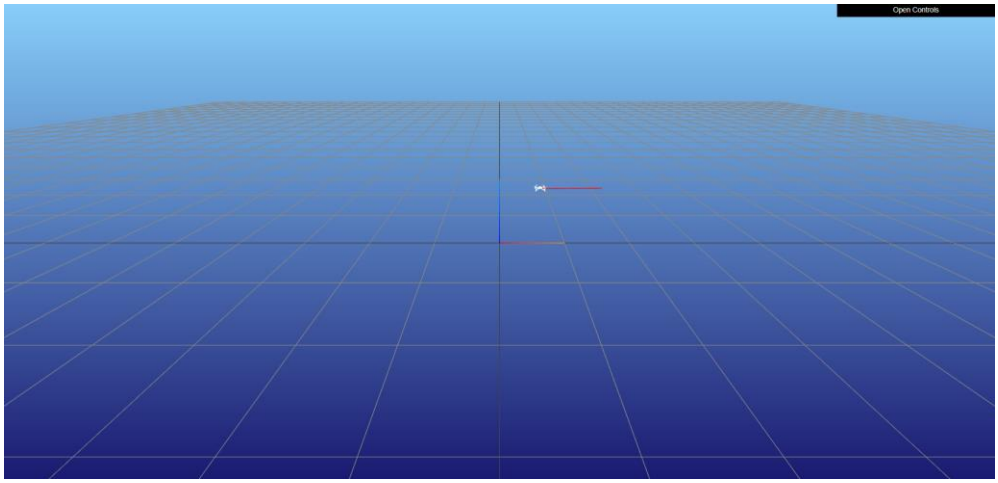
The following is the positional RMSE, and average control energy for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle Ψ_d=0:

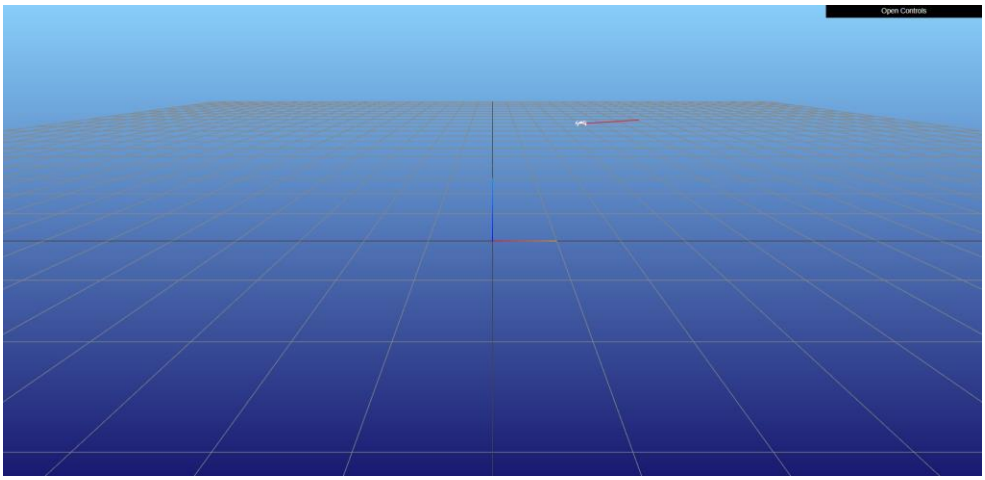
```
RMSE for x: 0.2796059422924717
RMSE for y: 0.27664832032992365
RMSE for z: 0.2801376268739797
Combined RMSE for position: 0.4828984014233368
Average Control Energy: 0.01769028620195436
```

Figure: Positional RMSE and average control energy for cascaded control for Ψ_d=0 for setpoint tracking from [0,0,0] to [1,1,1]

Following are the three screenshots from meshcat for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle Ψ_d=0:







Following is the graph for state and control profiles for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle $\Psi_d = \frac{t}{total\_time} * \frac{\pi}{3}$:

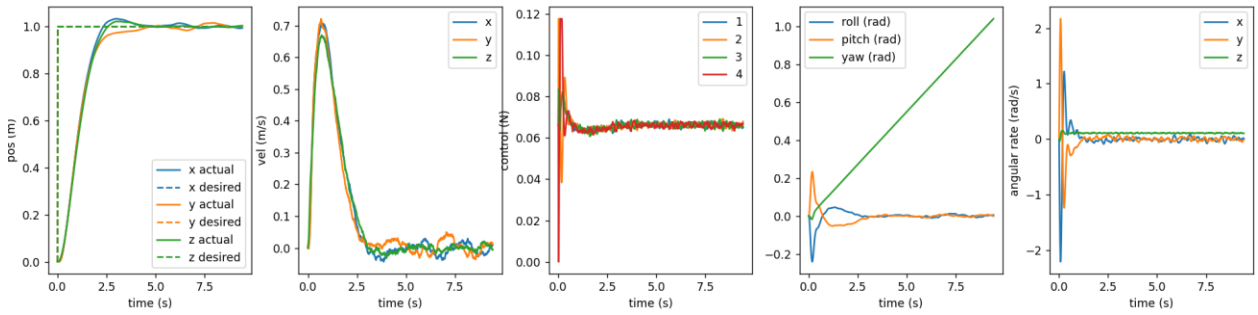


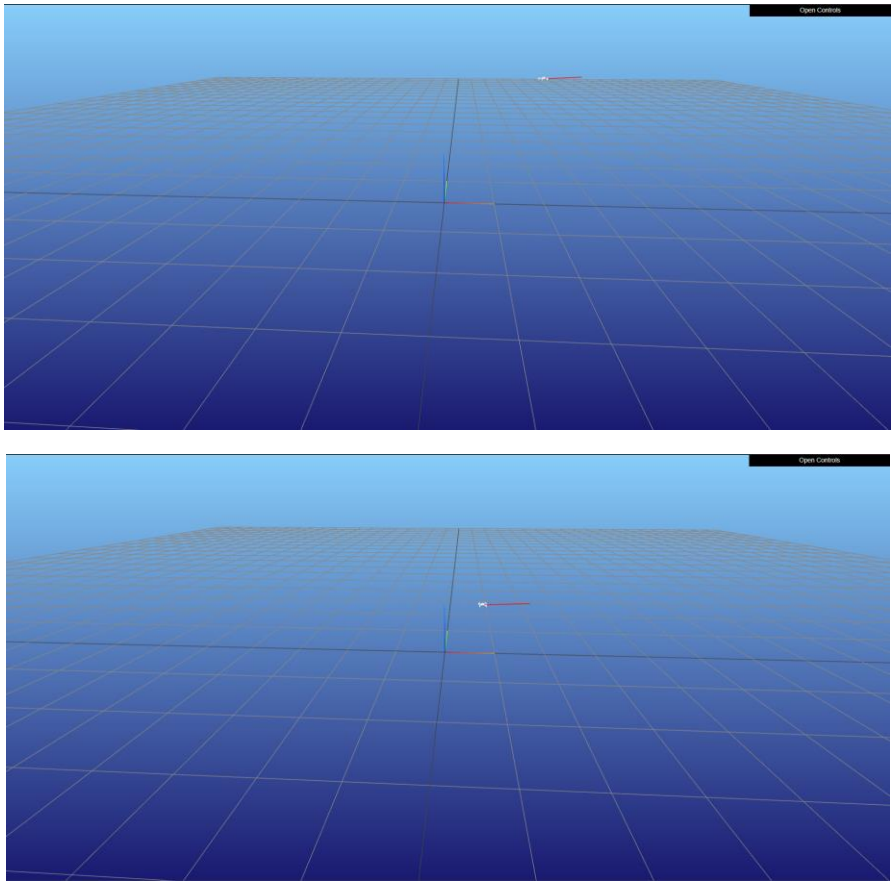
Figure: State and control profiles for cascaded control for $\Psi_d = \frac{t}{total\_time} * \frac{\pi}{3}$ for setpoint tracking from [0,0,0] to [1,1,1]

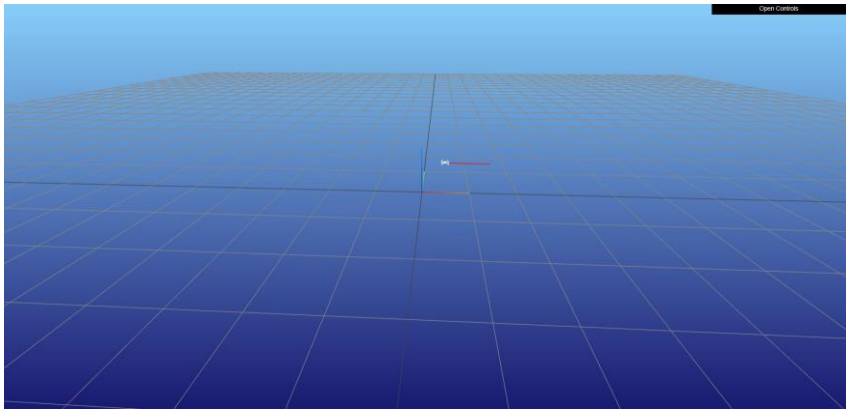
The following is the positional RMSE, and average control energy for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle $\Psi_d = \frac{t}{total\_time} * \frac{\pi}{3}$:

```
RMSE for x: 0.2792962969526272
RMSE for y: 0.2807235533018968
RMSE for z: 0.2828612736987311
Combined RMSE for position: 0.486644259216484 96
Average Control Energy: 0.017695875058226857
```

Figure: Positional RMSE and average control energy for cascaded control for $\Psi_d = \frac{t}{total\_time} * \frac{\pi}{3}$ for setpoint tracking from [0,0,0] to [1,1,1]

Following are the three screenshots from meshcat for setpoint tracking from [0,0,0] to [1,1,1] with yaw angle $\Psi_d = \frac{t}{total\_time} * \frac{\pi}{3}$:





Following is a tabular comparison for the two cases:

ψd	RMSE of x	RMSE of y	RMSE of z	Combined RMSE for position	Average Control Energy
0	0.278	0.277	0.280	0.483	0.0177
$\frac{t}{total\ time} * \frac{\pi}{3}$	0.279	0.281	0.283	0.487	0.0177

- **[10] For the trajectory tracking task ($\psi d = 0$), what PD gain values do you choose? Compare the performance of the gain values you choose and two sets of other gain values, in terms of the position RMSE. Report all three figures.**

The PD gains that I chose are:

v. $K_p = 2.5$
vi. $K_D = 2.5$
vii. $K_{Ptau} = 200$
viii. $K_{Dtau} = 20$

With the following state and control profile:

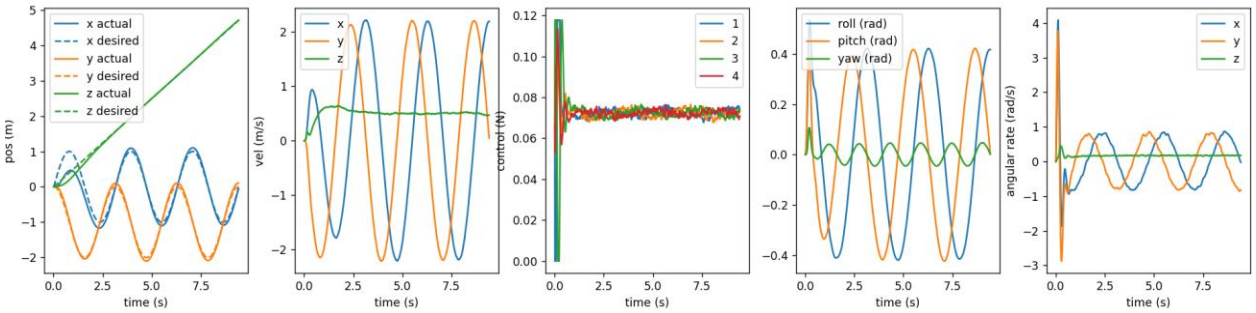


Figure: State and control profile for trajectory tracking using Kp = 2.5, KD = 2.5, KPtau = 200 KDtau = 20

And the following RMSE:

```
RMSE for x: 0.20398067193177 62
RMSE for y: 0.06585282783635679
RMSE for z: 0.06811519788486825
Combined RMSE for position: 0.224909736647123 66
```

Following is an ablation table for other set of PD gains:

K_P	K_D	K_{Ptau}	K_{Dtau}	RMSE X	RMSE Y	RMSE Z	Combined RMSE
2.5	2.5	200	20	0.2040	0.0659	0.0681	0.2250
1	1	10	10	1.1336	0.9673	0.3111	1.5223
2	2	150	10	0.2448	0.0565	0.1636	0.2998

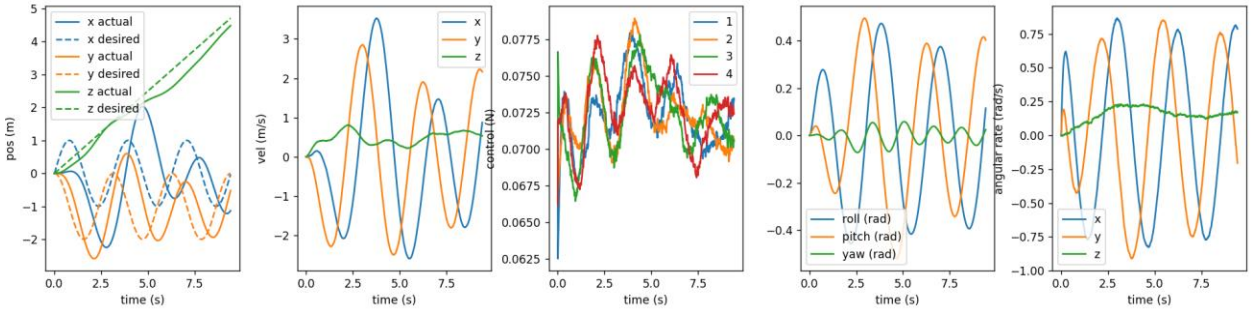


Figure: State and control profile for trajectory tracking using Kp = 1, KD = 1, KPtau = 10 KDtau = 10

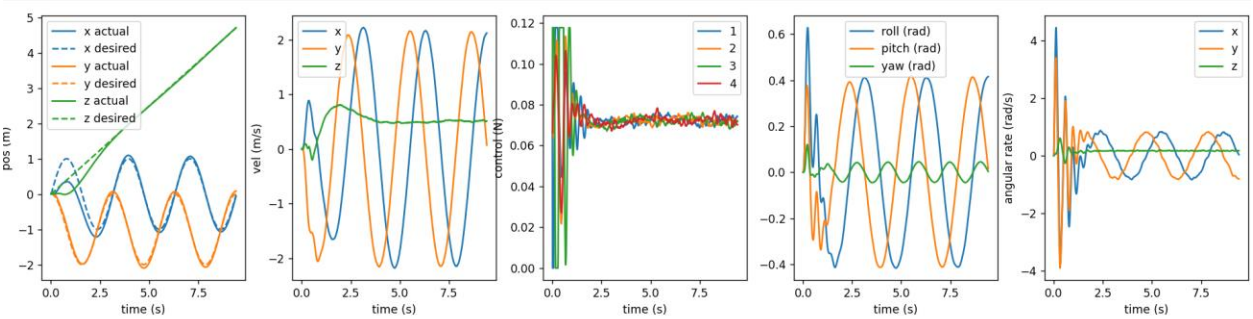


Figure: State and control profile for trajectory tracking using Kp = 2, KD = 2, KPtau = 150 KDtau = 10

3. **Integral Control and Adaptive Control [15] (and [5] extra points): In this question we have added some disturbance to the robot ("robot.d"). Based on lecture 4, complete the "adaptive_control" function in the "Quadrotor" class. Run "python quadrotor.py 3" to track a setpoint [1, 1, 1], and a spiral trajectory $px = \sin(2t)$, $py = \cos(2t) - 1$, $pz = 0.5t$. $\psi d = 0$ for both cases.**

- **[5] Use the default constant disturbance value ([0.5, 0.5, 1.0]). Implement integral control as the most simple adaptive control. Choose three different I gains and compare their performance, for both setpoint tracking (in terms of rise time, position RMSE, and maximum overshoot) and trajectory tracking (in terms of position RMSE).**

Ans: In this section of the question, we are asked to integrate integral control as an adaptive controller within the cascaded control (PD controller) we had used in the previous question. For this purpose, the positional error e_p is aggregated with time, and integrated with the f_d term to account for the constant disturbances. Carrying forward the PD gains from before, I used

Following are the results for setpoint tracking from [0,0,0] to [1,1,1]:

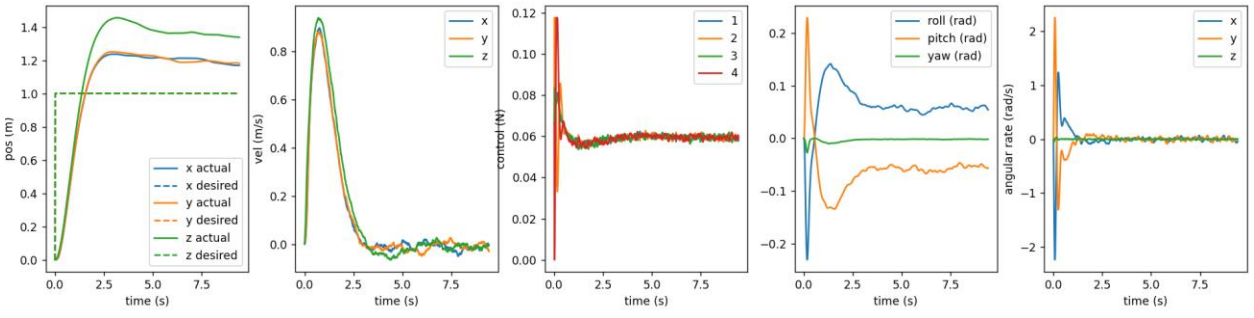


Figure: State and Control profiles for $K_I = 0.001$

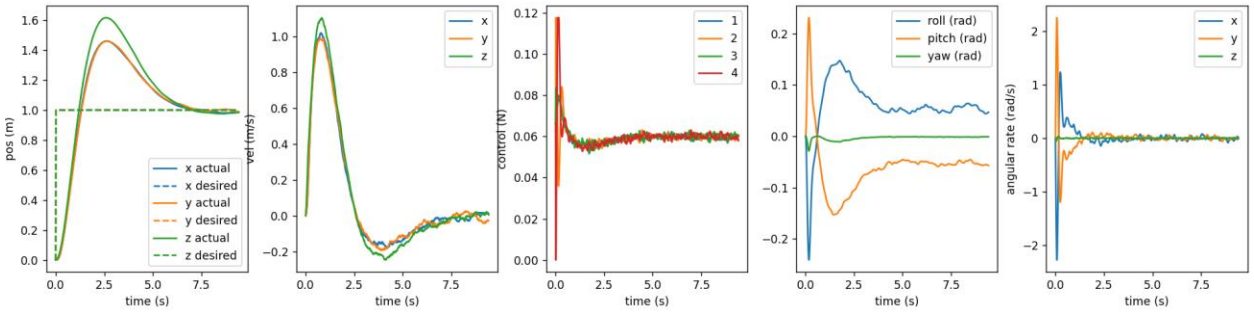


Figure: State and Control profiles for $K_I = 0.01$

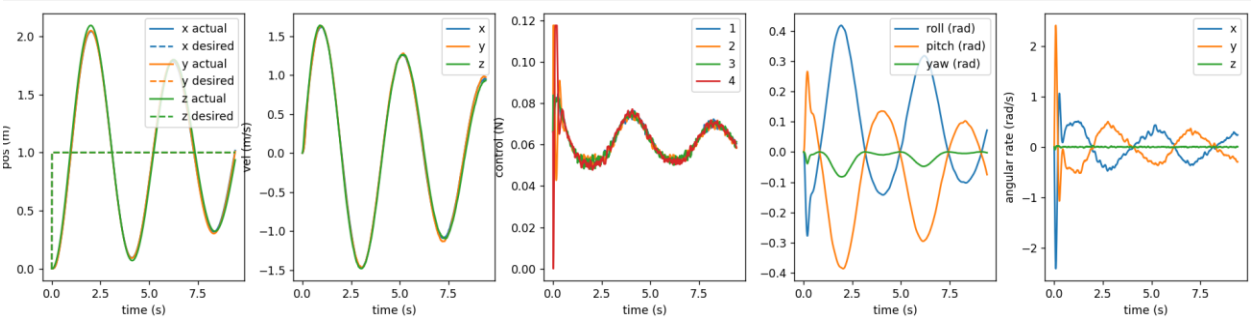


Figure: State and Control profiles for K_I = 0.05

Following is a ablation table for comparing values for different K_I for the case for setpoint tracking from [0,0,0] to [1,1,1]

K_I	Risetime X	Risetime Y	Risetime Z	RMSE X	RMSE Y	RMSE Z	Maximum Overshoot X	Maximum Overshoot Y	Maximum Overshoot Z
0.001	1.0105	1.0105	0.9204	0.3171	0.3177	0.4224	23.65%	25.03%	45.56%
0.01	0.8404	0.8604	0.7904	0.3273	0.3276	0.3737	46.13%	45.9%	61.6%
0.05	0.4926	0.4879	0.4930	0.6222	0.6278	0.6374	104.18%	104.81%	109.32%

Following are the results for spiral trajectory tracking:

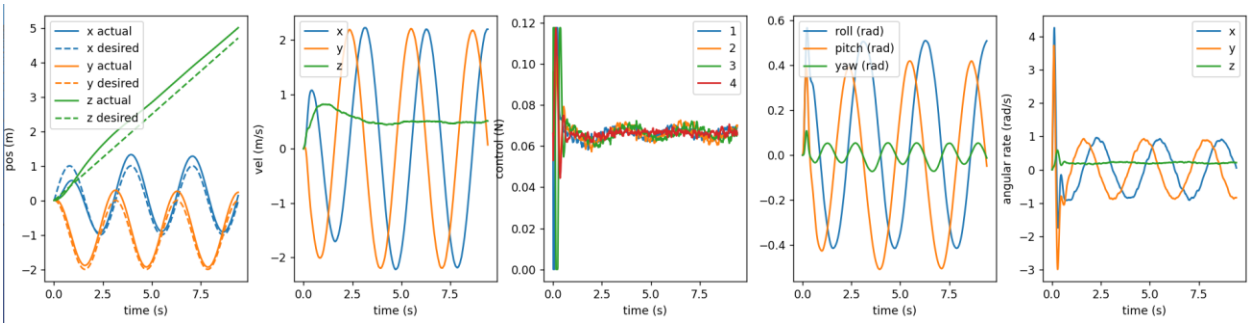


Figure: State and Control profiles for K_I = 0.001

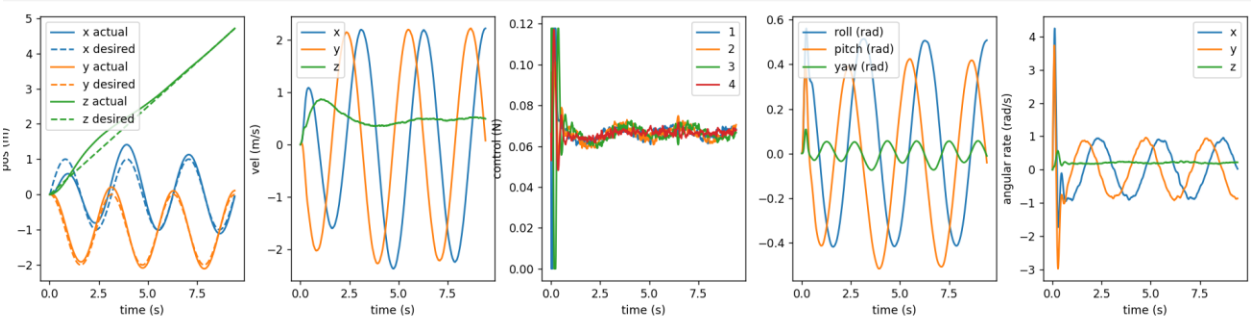


Figure: State and Control profiles for K_I = 0.01

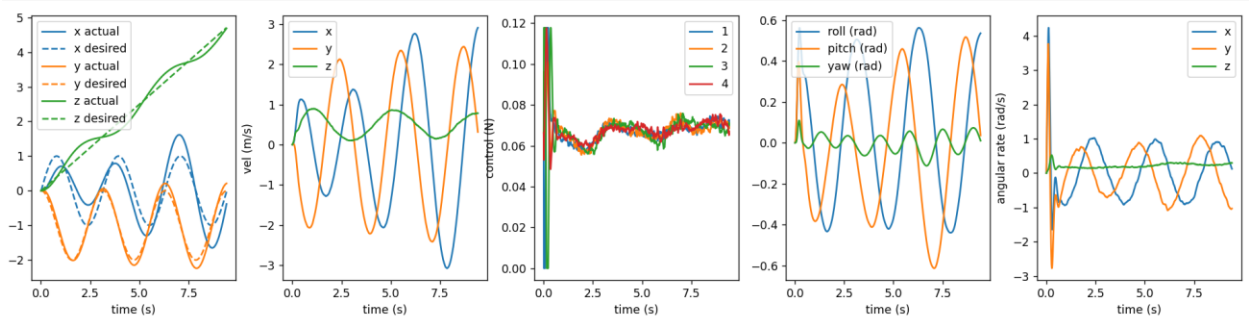


Figure: State and Control profiles for K_I = 0.05

Following is a ablation table for comparing values for different K_I for the case for spiral trajectory tracking:

K_I	RMSE X	RMSE Y	RMSE Z
0.001	0.2329	0.1777	0.3181
0.01	0.2356	0.1046	0.1723
0.05	0.4255	0.1225	0.1741

- **[5] Change the disturbance to be time-variant ([0.5, sin(t), cos(\sqrt{2}t)]). Apply the best controller in the last question for both setpoint and trajectory tracking. Comment on the difference from the constant disturbance case**

Using the best performing K_I term of "0.01" and changing the disturbance from constant value to a time-variant one $[0.5, \sin(t), \cos(\sqrt{2}t)]$, following are my observations:

i. The time-variant disturbance introduces variations are not completely countered by the integral term in the PID controller. The reason behind is that the controller has been tuned to manage constant disturbance. Time-variant disturbances can introduce additional dynamics to the system that might not have been considered during the tuning process.

ii. Since the disturbance is no longer constant, the integral term accumulates error more aggressively during periods when the disturbance is increasing making the controller more unstable.

iii. With a time-variant disturbance, the system exhibits different overshoot behaviors as compared to the constant disturbance case which depend on the phase and magnitude of the disturbance relative to the quadrotor trajectory.

iv. In the constant disturbance scenario, after an initial transient, the system reached a steady state, but with a time-variant disturbance, the system oscillates and continuously displays transient behavior.

Following are the results for spiral trajectory tracking with the time-variant disturbances along y and z:

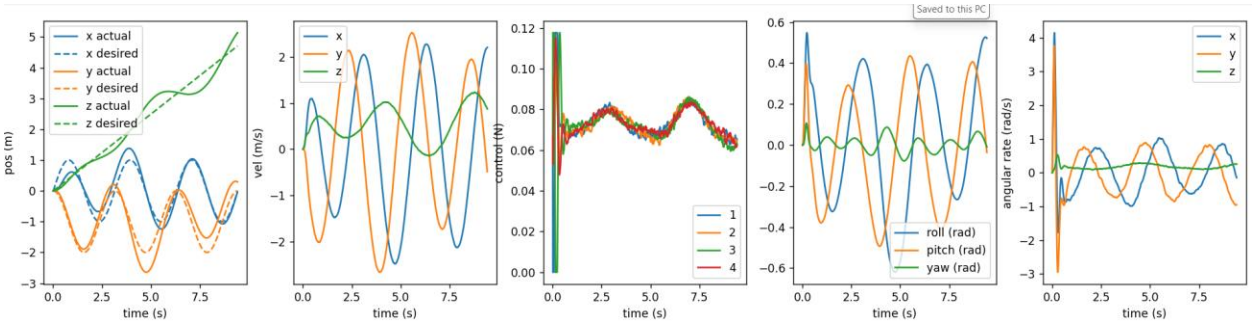


Figure: State and Control profiles for K_I = 0.01 with time-variant disturbances for spital trajectory tracking

Following table summarizes the RMSE for position in X, Y, and Z:

RMSE X	RMSE Y	RMSE Z
0.2551	0.3508	0.2653

Following are the results for setpoint tracking from [0,0,0] to [1,1,1] with the time-variant disturbances along y and z:

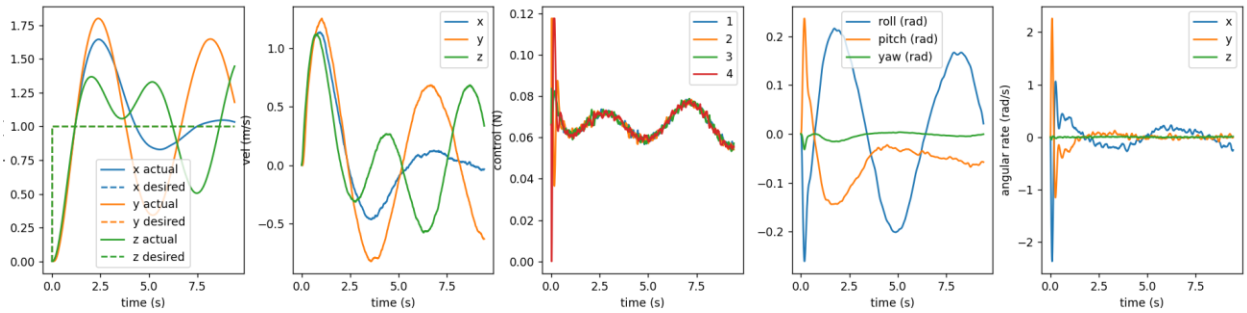


Figure: State and Control profiles for K_I = 0.01 with time-variant disturbances for setpoint tracking from [0,0,0] to [1,1,1]

Following table summarizes the risetime, RMSE, and maximum overshoot for position in X, Y, and Z:

Risetime X	Risetime Y	Risetime Z	RMSE X	RMSE Y	RMSE Z	Maximum Overshoot X	Maximum Overshoot Y	Maximum Overshoot Z
1.2304	1.2496	1.2362	0.3604	0.5340	0.3504	64.60%	80.13%	44.67%

- **[5] Apply the time-variant disturbance [0.5, sin(t), cos(\sqrt{2}t)]. Implement the L-1 adaptive control method taught in lecture 4 for both setpoint and trajectory tracking. Is it better than integral control and why?**

Ans: For this question, we are asked to implement the L1 adaptive control method and compare its performance for time-variant disturbances. The L1 adaptive control has the advantage that it can adjust its parameters in real-time to estimate the disturbance to reduce its adverse effects. Following is the hierarchical implementation for L1 adaptive control:

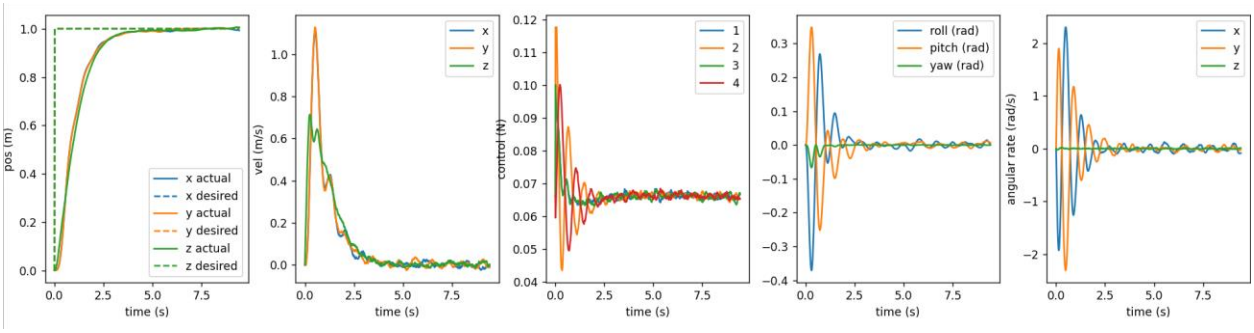


Figure: State and control profiles for L1 adaptive control for setpoint tracking under time-variant disturbances

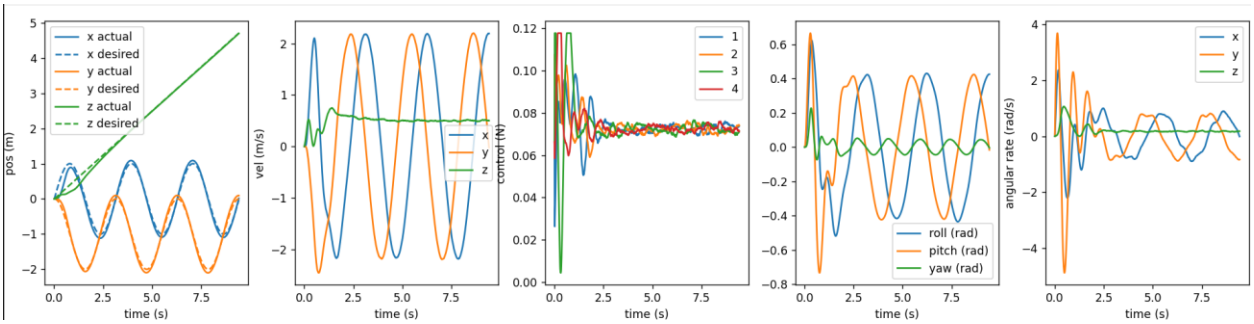


Figure: State and control profiles for L1 adaptive control for trajectory tracking under time-variant disturbances

Controller	Risetime X	Risetime Y	Risetime Z	RMSE X	RMSE Y	RMSE Z	Maximum Overshoot X	Maximum Overshoot Y	Maximum Overshoot Z
Integral (KI = 0.01)	1.2304	1.2496	1.2362	0.3604	0.5340	0.3504	64.60%	80.13%	44.67%
L1 adaptive control (A_s = -0.5, α = 0.3)	1.6909	1.701	1.9310	0.2622	0.2623	0.2628	0.31%	0.48%	0.64%

Controller	RMSE X	RMSE Y	RMSE Z
Integral (KI = 0.01)	0.2551	0.3508	0.2653
L1 adaptive control (A_s = -0.5, α = 0.3)	0.1199	0.0749	0.0806

The L1 adaptive controller outperforms the integral controller for time-variant disturbances as can be seen in the ablation table above. Following is a comparison of the integral controller and L1 adaptive controller as observed from the results.

a. L1 adaptive controller has this ability to provide fast adaptation without causing a transient overshoot. While in contrast, the integral control can eliminate steady-state errors, but introduces larger transient overshoots, especially for aggressive integral gains.
b. L1 adaptive control offers robustness to uncertainties since it operates using estimates of system state parameters. This makes it more robust to time-variant disturbances. On the other hand, integral control can address steady-state errors but large or time varying distances are a bane for integral controller (as observed in the previous question).
c. L1 control architecture offers a clear decoupling of the adaptation mechanism from the control performance, enabling independent tuning, which is not the case of PID tuning
d. Since L1 adaptive control doesn't accumulate error over time hence there is no integral windup as observed for the integral controller.