Assignment 2

Control & Navigation of a Quadrotor

EC4.402 - Introduction to UAV Design

Avneesh Mishra
avneesh.mishra@research.iiit.ac.in *

April 16, 2022

Contents

*M.S. by research - CSE, IIIT Hyderabad, Roll No: 2021701032

1 System Model

This section describes the system model of the UAV. Some parts of this are inspired by [MKC12]. The final results (through this model) is presented in section 2.

The basic system overview used is shown in figure 1.

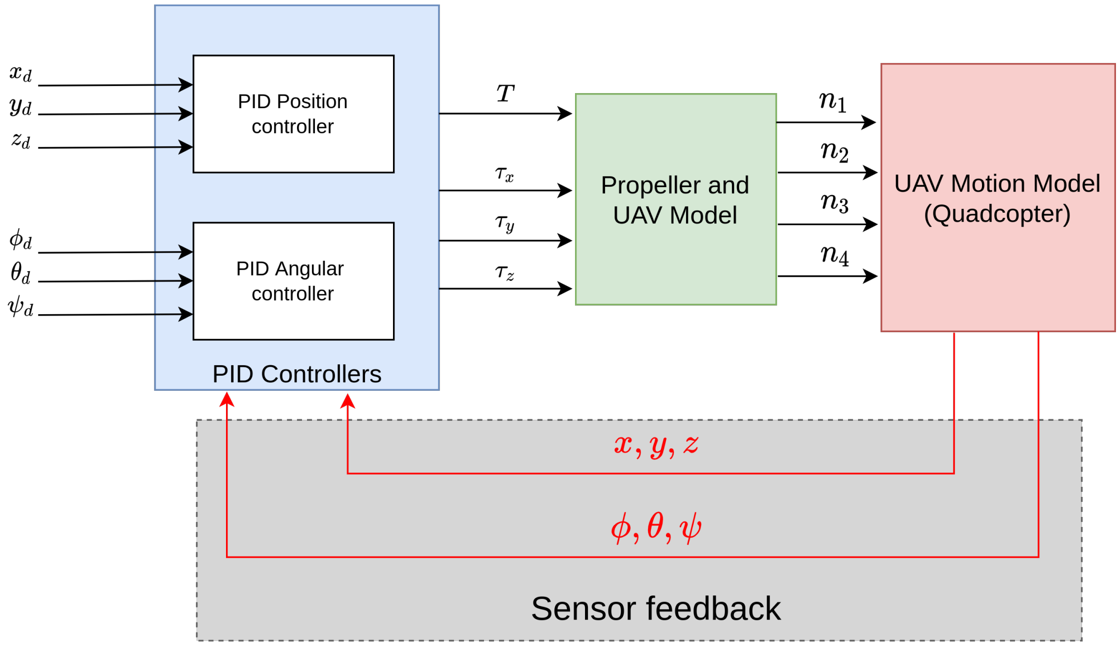


Figure 1: UAV System overview
The PID controllers take the desired position and angles, and the current position and angles (taken through sensor data); and yield the thrust and torque action required (through predicting acceleration and using inertial model).
The Propeller and UAV Model takes these (desired) thrust and torque actions, and converts them into (desired) propeller speed commands.
These are given to the UAV (quadcopter). Here, we'll simulate one. Through physics (motion modeling), the new state is obtained.

1.1 PID Controllers

There are two PID controllers. They predict the desired accelerations (control action) using PID equations. The PD controller model is described below

$$\ddot{\mathbf{x}} = \mathbf{k_p}(\mathbf{x_d} - \mathbf{x}) + \mathbf{k_d}(\dot{\mathbf{x}}_\mathbf{d} - \dot{\mathbf{x}}) \qquad \ddot{\alpha} = \mathbf{k}_{\alpha_\mathbf{p}}(\alpha_d - \alpha) + \mathbf{k}_{\alpha_\mathbf{d}}(\dot{\alpha}_d - \dot{\alpha}) \tag{1.1}$$

Where $\mathbf{x} = [x, y, z]$ is the position and $\alpha = [\phi, \theta, \psi]$ is the orientation in world frame (inertial, not body).

First, we get the desired linear acceleration (and compensate for gravity). This is converted to desired thrust by multiplying with mass.

The thrust vector is converted to desired angles using knowledge of spherical angles. Basically, the thrust vector has to be aligned with the -Z axis of the UAV (that's where the propulsion is).

The desired angles have to be *clipped* (we cannot expect the UAV to fly at 90° roll or pitch, we clip all angles received to a small angle like 20°).

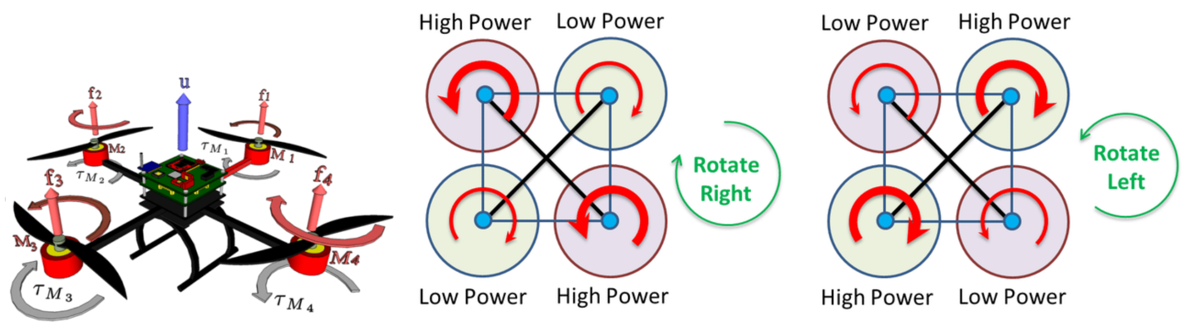
The angular acceleration is calculated through the $\ddot{\alpha}$ equation above.

Angular acceleration is converted to body torques using the inertia tensor $\mathbf{J}$, that is $\tau = \mathbf{J}\ddot{\alpha} = [\tau_x, \tau_y, \tau_z]$. The thrust is already derived from $\ddot{\mathbf{x}}$ (as described above).

1.2 Propeller equations

Refer to the image below for getting the directional sense of propeller rotations.

From figure 2, the following system of equations are clear



(a) Thrust

(b) Yaw control

Figure 2: UAV Propeller Thrust

$$\begin{bmatrix} -T \\ \tau_{x_b} \\ \tau_{y_b} \\ \tau_{z_b} \end{bmatrix} = \underbrace{\begin{bmatrix} k_t & k_t & k_t & k_t \\ 0.5Lk_\tau & -0.5Lk_\tau & -0.5Lk_\tau & 0.5Lk_\tau \\ 0.5Lk_\tau & 0.5Lk_\tau & -0.5Lk_\tau & -0.5Lk_\tau \\ k_\tau & -k_\tau & k_\tau & -k_\tau \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} n_1^2 \\ n_2^2 \\ n_3^2 \\ n_4^2 \end{bmatrix} \tag{1.2}$$

Using equation 1.2, we can find the motor speeds n (by inverting $\mathbf{M}$).

Note that we used $-T$ here because thrust, which is to counter weight, is in the -Z direction (it'll finally be positive).

A clipping for propeller speeds is also applied, the motors can only give a particular maximum rotation speed.

1.3 UAV Motion Model

Using the (clipped) propeller speeds n_i, we first get the thrust T (should be -ve because it is along -Z). We represent this T in the inertial frame using the relation below

$$\mathbf{T}_{\text{inertial}} = \underbrace{\begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix}}_{{}^i_b\mathbf{R}=\text{R}(\text{Z},\psi)\text{R}(\text{Y},\theta)\text{R}(\text{X},\phi)} \mathbf{T}_{\text{body}} \tag{1.3}$$

In equation 1.3, $\mathbf{T}_{\text{body}} = [0 \;\; 0 \;\; -T]^\top$ where T is the (positive) thrust from propellers.

We add $[0 \;\; 0 \;\; mg]^\top$ (weight) to $\mathbf{T}_{\text{inertial}}$ and divide by m to get the *linear acceleration* (in world fixed/inertial frame).

We use equation 1.2 to get the body torque vector $\tau = [\tau_x \;\; \tau_y \;\; \tau_z]^\top$. We use the coriolis equation below to get the angular accelerations in the body frame

$$\mathbf{J}\left(\frac{d\mathbf{\Omega}}{dt}\right)_{\text{B}} + \mathbf{\Omega} \times (\mathbf{J}\mathbf{\Omega}) = \tau \tag{1.4}$$

Where Ω is the angular velocity in the (get it from equation 1.5). Using equation 1.4, we get the angular acceleration $({}^{d\mathbf{\Omega}}/_{dt})_{\text{B}}$.

The relation between the angular velocity in the inertial frame and the angular velocity in the body frame is given below

$$\begin{aligned} \mathbf{\Omega} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} &= \begin{bmatrix} \dot\phi \\ 0 \\ 0 \end{bmatrix} + \text{Rot}(X, -\phi) \begin{bmatrix} 0 \\ \dot\theta \\ 0 \end{bmatrix} + \text{Rot}(X, -\phi)\text{Rot}(Y, -\theta) \begin{bmatrix} 0 \\ 0 \\ \dot\psi \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & s_\phi c_\theta \\ 0 & -s_\phi & c_\phi c_\theta \end{bmatrix}}_{\mathbf{M}_\Omega} \begin{bmatrix} \dot\phi \\ \dot\theta \\ \dot\psi \end{bmatrix} \end{aligned} \tag{1.5}$$

We apply 1.5 to get Ω, and then apply 1.4 to get $\dot\Omega$. Angular acceleration of body frame is derived by inverting $\mathbf{M}_\Omega$ in equation 1.5. We now have *body's angular accelerations* in the inertial frame.

We use the obtained angular acceleration to update angular velocity, and subsequently the angles $[\phi \;\; \theta \;\; \psi]^\top$. We also use the obtained linear accelerations to update the body velocity and acceleration.

1.4 Conclusion

The above three sections are applied iteratively in a loop till the final time. Maybe drag can be added in the force calculations (for drag force of body frame in inertial frame). This submission doesn't include that.

2 Results

2.1 Simulation variables

The following simulation variables were set. Except mass, all others were assumed.

```
# %%
# --- Targets (final frame) ---
# pos_d = np.array([0., 0., 0.])  # Desired X, Y, Z position
# pos_d = np.array([20., 40., -5.])      # Desired X, Y, Z position
# pos_d = np.array([30., -50., -5.])      # Desired X, Y, Z position
# pos_d = np.array([-10., -60., -5.])      # Desired X, Y, Z position
pos_d = np.array([-70., 30., -5.])      # Desired X, Y, Z position
ang_d = np.array([0., 0., 0.])      # Desired inertial X, Y, Z angles
vel_d = np.array([0., 0., 0.])      # Desired X, Y, Z velocity
# --- Initial state of the quadrotor ---
pos_init = np.array([0., 0., 0.])  # X, Y, Z   - in m
vel_init = np.array([0., 0., 0.])  # vx, vy, vz - in m/s
ang_init = np.array([0., 0., 0.])  # Phi, Theta, Psi - in rad
# ang_init = np.random.rand(3) / 5.0
ang_vel_init = np.array([0., 0., 0.])  # Phi dot, theta dot, psi dot
# --- Properties of the UAV ---
m = 2.0   # Mass of UAV - in kg
g = 9.8 # Gravity magnitude - in m/(s^2)
J = np.array([[1.2472e-4, 0., 0.],  # Ixx, Ixy, Ixz -|
    [0., 1.2472e-4, 0.],            # Iyx, Iyy, Iyz -- Inertia tensor
    [0., 0., 8.4488e-5]])           # Izx, Izy, Izz -|
pkT = 2e-7  # Thrust constant of propeller (pkT * (n**2) = thrust)
pkt = 1e-9  # Torque constant of propeller (pKt * (n**2) = torque)
"""pkT and pkt use 'n' in RPM (revs per min). SI is rad per sec"""
qL = 0.4    # Distance between propeller centers on the same side (m)
p_max_RPM = 7000.    # Max. revs per minute of the propellers
# --- Controller properties ---
Kp_pos = np.array([0.5, 0.5, 10.0])     # K_p for pos: X, Y, Z
Kd_pos = np.array([1.0, 1.0, 10.0])       # K_d for pos: X, Y, Z
# K_p for inertial ang: X, Y, Z
Kp_ang = np.array([0.3, 0.3, 5.0])
# K_d for inertial ang: X, Y, Z
Kd_ang = np.array([0.1, 0.1, 1.0])
max_phi = np.deg2rad(25.)    # Maximum Phi angle (rot. along X)
max_theta = np.deg2rad(25.)  # Maximum Theta angle (rot. along Y)
max_psi = np.deg2rad(5.)     # Maximum Psi angle (rot. along Z)
# max_ang_acc = 0.5           # Maximum angular
# --- Simulation properties ---
dt = 5e-4   # Time steps for simulation - in sec
start_time = 0.0    # Start time - in sec
end_time = 15.0     # End time - in sec
```

The total code is present in Appendix A.1. The remainder of this section presents graphs generated by the code. Note that each target has the initial conditions as above (zero).

It is also interesting to observe that all items settle to zero in the end, except the thrust to counter weight (which is reflected in the desired acceleration, thrust, and even the actual motor speeds).

2.2 Target 1: (20, 40, -5)

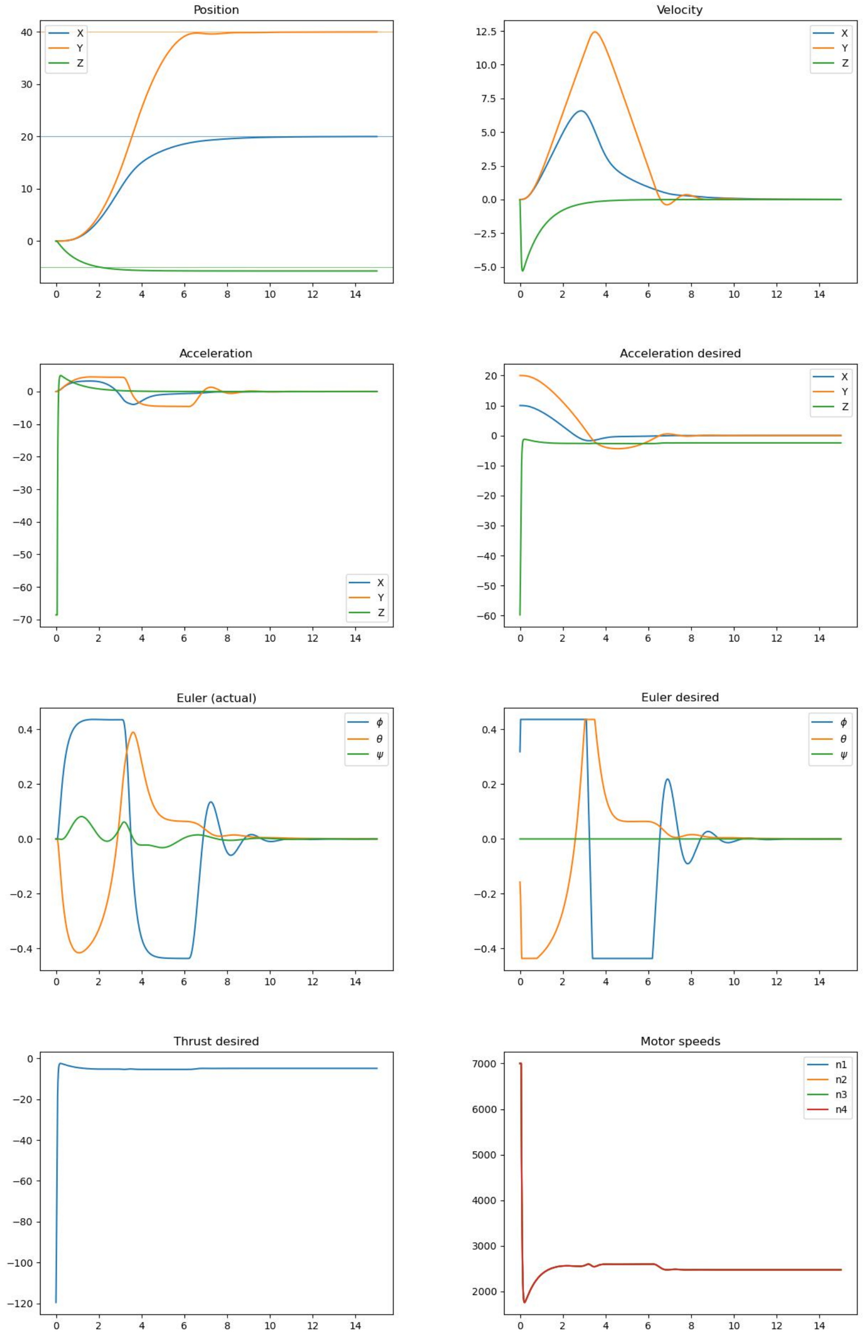


Figure 3: Plots for $\mathbf{x}_d = [20 \;\; 40 \;\; -5]^\top$

The *first* row has actual position and velocity. The *second* row has actual and desired acceleration. The *third* row has actual and desired euler angles (euler, w.r.t. ground/world frame). The *fourth* row has the thrust desired (to counter weight) and the motor speeds.

2.3 Target 2: (30, -50, -5)

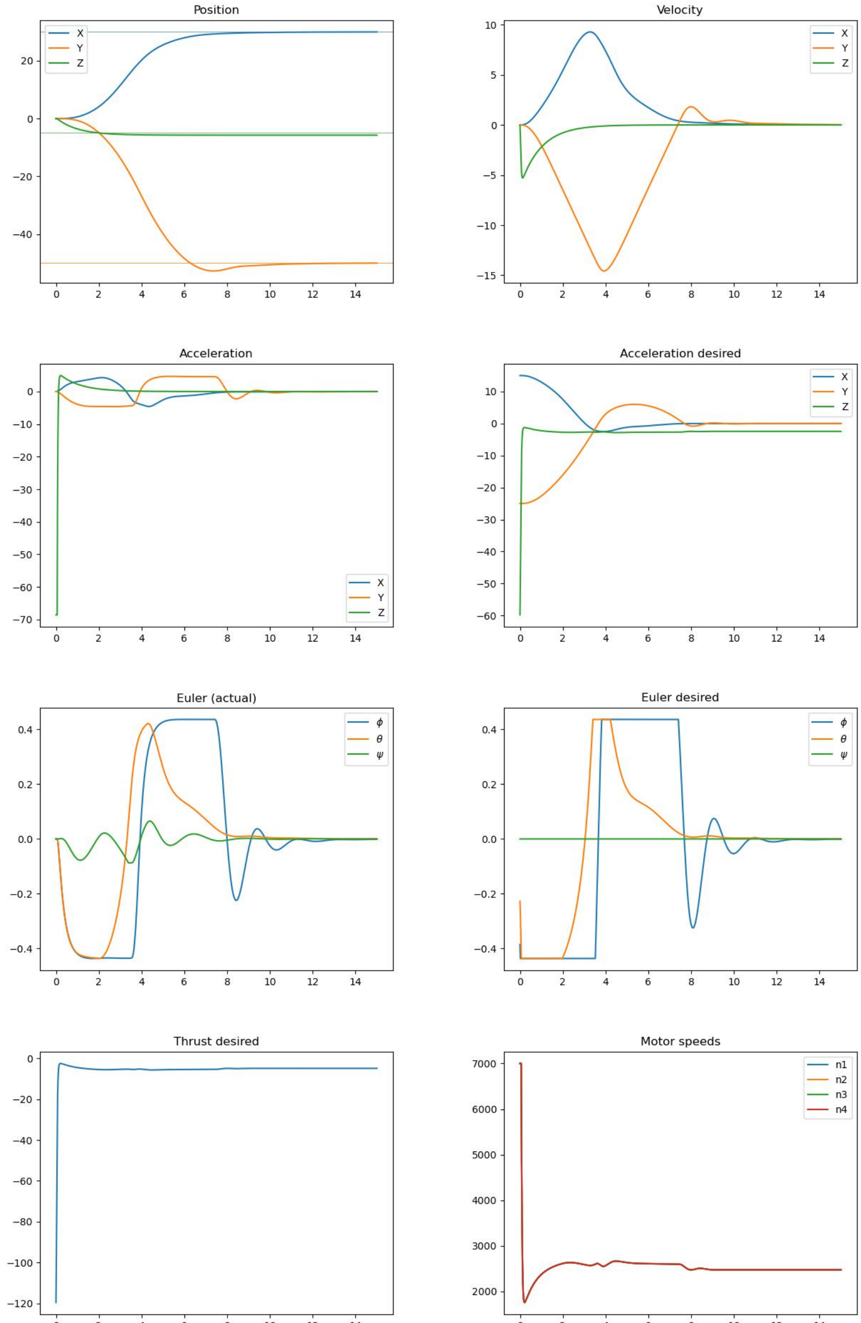


Figure 4: Plots for $\mathbf{x}_d = [20 \;\; 40 \;\; -5]^\top$

The *first* row has actual position and velocity. The *second* row has actual and desired acceleration. The *third* row has actual and desired euler angles (euler, w.r.t. ground/world frame). The *fourth* row has the thrust desired (to counter weight) and the motor speeds.

2.4 Target 3: (-10, -60, -5)

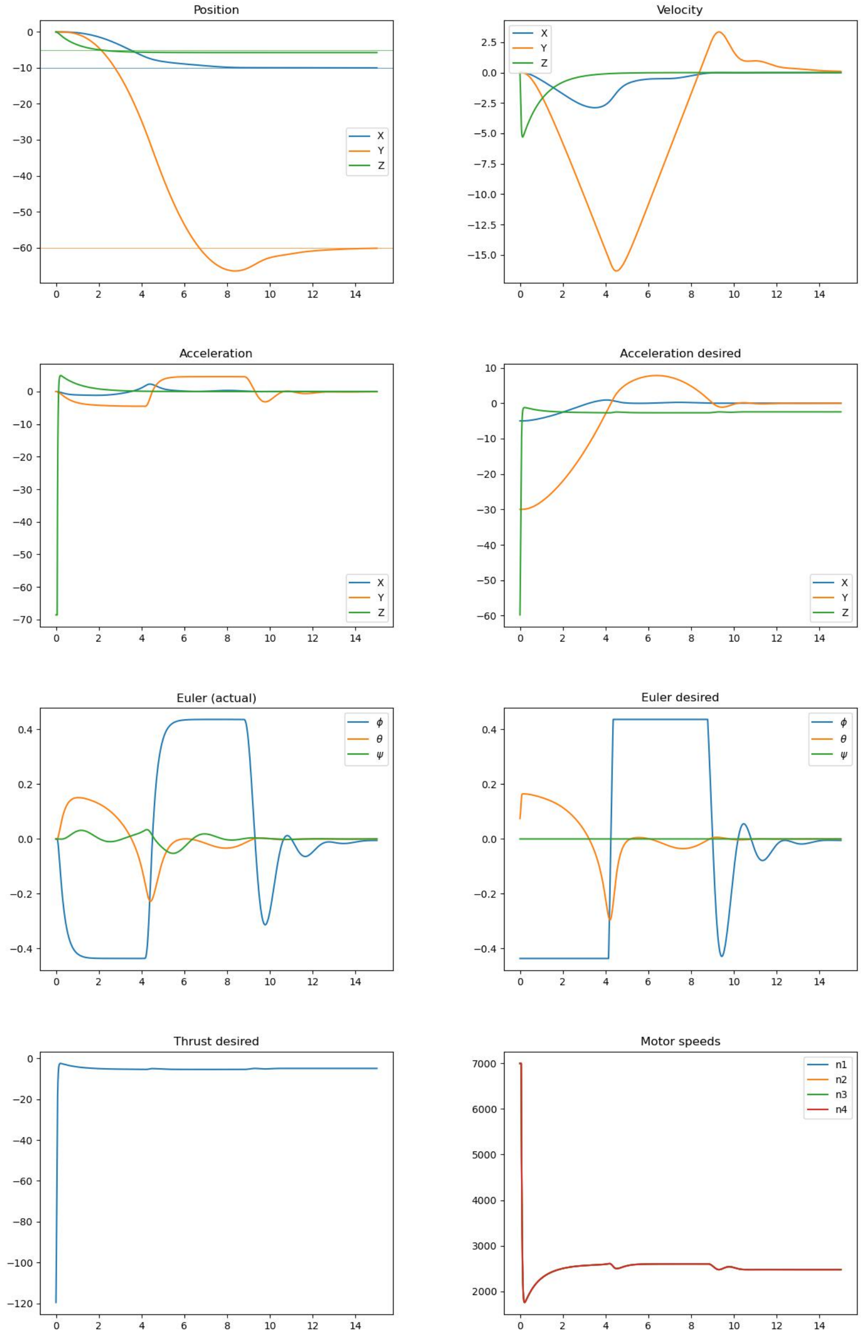


Figure 5: Plots for $\mathbf{x}_d = [20 \;\; 40 \;\; -5]^\top$

The *first* row has actual position and velocity. The *second* row has actual and desired acceleration. The *third* row has actual and desired euler angles (euler, w.r.t. ground/world frame). The *fourth* row has the thrust desired (to counter weight) and the motor speeds.

2.5 Target 4: (-70, 30, -5)

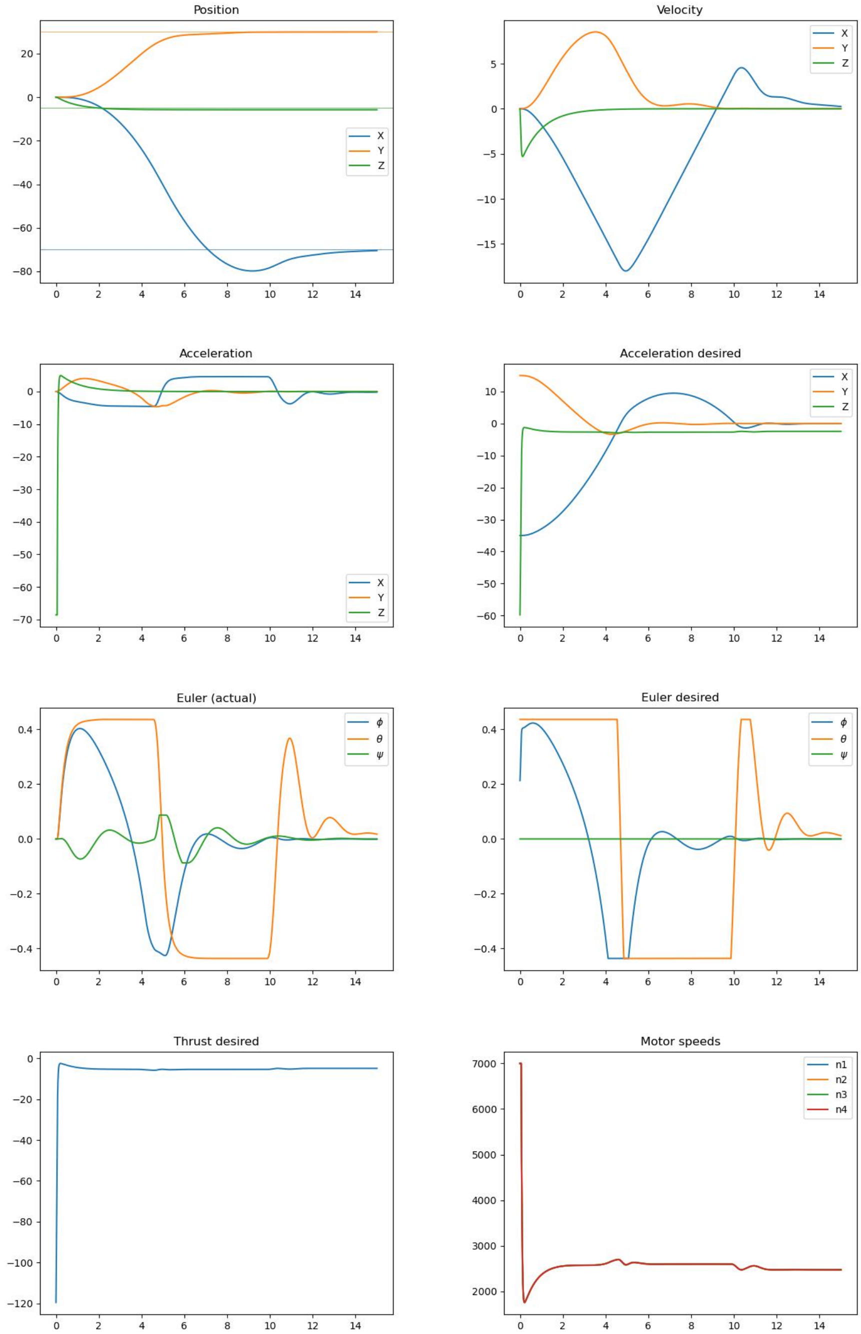


Figure 6: Plots for $\mathbf{x}_d = [20 \;\; 40 \;\; -5]^\top$

The *first* row has actual position and velocity. The *second* row has actual and desired acceleration. The *third* row has actual and desired euler angles (euler, w.r.t. ground/world frame). The *fourth* row has the thrust desired (to counter weight) and the motor speeds.

References

[MKC12] Robert Mahony, Vijay Kumar, and Peter Corke. "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor". In: *IEEE Robotics and Automation magazine* 19.3 (2012), pp. 20–32.

A Code

A.1 Generating plots

The code that uses the system model and generates the plots is presented below. The latest code and all material pertaining to the course can be found at github: TheProjectsGuy/UAV22-EC4.402

```python
# Main motion model
"""
    Using the initial and desired states of the quadrotor, generate a
    CSV file which contains the positions, velocities, desired
    accelerations, desired roll, desired pitch, desired yaw, total
    desired thrust, motor speeds, along with timestamps in the
    beginning.

    Frame X, Y, Z is following the NED convention. Using PD controller
    for generating thrust and angle actions

    Set the variables in `Targets (final frame)` and run the code.
    Plots are in the end.
"""

# %% Import everything
import numpy as np
from matplotlib import pyplot as plt
from tqdm import tqdm

# %%
# --- Targets (final frame) ---
# pos_d = np.array([0., 0., 0.])  # Desired X, Y, Z position
# pos_d = np.array([20., 40., -5.])     # Desired X, Y, Z position
# pos_d = np.array([30., -50., -5.])     # Desired X, Y, Z position
# pos_d = np.array([-10., -60., -5.])     # Desired X, Y, Z position
pos_d = np.array([-70., 30., -5.])     # Desired X, Y, Z position
ang_d = np.array([0., 0., 0.])     # Desired inertial X, Y, Z angles
vel_d = np.array([0., 0., 0.])     # Desired X, Y, Z velocity
# --- Initial state of the quadrotor ---
pos_init = np.array([0., 0., 0.])  # X, Y, Z   - in m
vel_init = np.array([0., 0., 0.])  # vx, vy, vz - in m/s
ang_init = np.array([0., 0., 0.])  # Phi, Theta, Psi - in rad
# ang_init = np.random.rand(3) / 5.0
ang_vel_init = np.array([0., 0., 0.])  # Phi dot, theta dot, psi dot
# --- Properties of the UAV ---
m = 2.0   # Mass of UAV - in kg
g = 9.8 # Gravity magnitude - in m/(s^2)
J = np.array([[1.2472e-4, 0., 0.],  # Ixx, Ixy, Ixz -|
    [0., 1.2472e-4, 0.],            # Iyx, Iyy, Iyz -- Inertia tensor
    [0., 0., 8.4488e-5]])           # Izx, Izy, Izz -|
pkT = 2e-7  # Thrust constant of propeller (pkT * (n**2) = thrust)
pkt = 1e-9  # Torque constant of propeller (pKt * (n**2) = torque)
"""pkT and pkt use 'n' in RPM (revs per min). SI is rad per sec"""
qL = 0.4    # Distance between propeller centers on the same side (m)
p_max_RPM = 7000.   # Max. revs per minute of the propellers
# --- Controller properties ---
Kp_pos = np.array([0.5, 0.5, 10.0])     # K_p for pos: X, Y, Z
Kd_pos = np.array([1.0, 1.0, 10.0])       # K_d for pos: X, Y, Z
# K_p for inertial ang: X, Y, Z
Kp_ang = np.array([0.3, 0.3, 5.0])
# K_d for inertial ang: X, Y, Z
Kd_ang = np.array([0.1, 0.1, 1.0])
max_phi = np.deg2rad(25.)     # Maximum Phi angle (rot. along X)
max_theta = np.deg2rad(25.)   # Maximum Theta angle (rot. along Y)
max_psi = np.deg2rad(5.)      # Maximum Psi angle (rot. along Z)
# max_ang_acc = 0.5             # Maximum angular
# --- Simulation properties ---
dt = 5e-4   # Time steps for simulation - in sec
start_time = 0.0    # Start time - in sec
end_time = 15.0     # End time - in sec

# %% Functions
```

```python
# Convert desired ang. acceleration and thrust to motor speeds
def angaccs_thr_to_motor_speeds(ang_accs: np.ndarray,
        des_thrust: float):
    # Inverse of relation matrix
    M_inv = np.array([
        [0.25/pkT, 0.5/(qL*pkt), 0.5/(qL*pkt), 0.25/pkt],
        [0.25/pkT, -0.5/(qL*pkt), 0.5/(qL*pkt), -0.25/pkt],
        [0.25/pkT, -0.5/(qL*pkt), -0.5/(qL*pkt), 0.25/pkt],
        [0.25/pkT, 0.5/(qL*pkt), -0.5/(qL*pkt), -0.25/pkt]
    ])
    # Tau = I * alpha
    body_tau: np.ndarray = J @ ang_accs.reshape(3, 1)   # tx, ty, tz
    # Force (effort) vector: [-T, tx, ty, tz]
    f_vect = np.array([-des_thrust, *body_tau.flatten().tolist()]).\
        reshape(4, 1)
    # Get n**2
    ms_sq = M_inv @ f_vect
    ms_sq[ms_sq < 0] = 0    # This probably should not happen!
    ms_unclipped = ms_sq**(0.5)
    ms_clipped = np.clip(ms_unclipped, np.zeros_like(ms_unclipped),
        np.array([4*[p_max_RPM]]).reshape(4, 1))
    # Return clipped motor speeds
    return ms_clipped.flatten() # As (4,)

# Convert motor speeds to body torque
def mspeeds_btorque(m_speeds: np.ndarray):
    # Extract speeds
    n1s, n2s, n3s, n4s = m_speeds**2    # Square of speeds
    # Torques due to thrust of propellers
    tx = 0.5*qL*pkT*(n1s - n2s- n3s + n4s)
    ty = 0.5*qL*pkT*(n1s + n2s - n3s - n4s)
    # Torque due to body reaction of propellers
    tz = pkt*(n1s - n2s + n3s - n4s)
    return np.array([tx, ty, tz])

# Rotation matrix: R_inertial_body: Body to inertial transformation
def rotmat_inertial_body(curr_angs):
    """
        Given `curr_angs`, return R_inertial_body: Rotation matrix
        transforming a vector in body frame, into the inertial frame.
        It is basically the body frame expressed in inertial frame.

        Note that: curr_angs = [phi, theta, psi]
            phi: Rot(X): Roll, theta: Rot(Y): Pitch, psi: Rot(Z): Yaw

        We consider rotation sequence:
            Rot(Z, psi) * Rot(Y, theta) * Rot(X, roll)
        For above: ZYX Euler or XYZ fixed - both are same
    """
    roll, pitch, yaw = curr_angs    # Extract angles
    # Short for trig. exprs [[s]in | [c]os] [[ph]i | [th]eta | [ps]i]
    cph, sph = np.cos(roll), np.sin(roll)       # Phi - 1 - Roll
    cth, sth = np.cos(pitch), np.sin(pitch)     # Theta - 2 - Pitch
    cps, sps = np.cos(yaw), np.sin(yaw)         # Psi - 3 - Yaw
    # Rotation matrix
    rot_mat = np.array([
        [cth*cps, sph*sth*cps - cph*sps, sph*sps + cph*sth*cps],
        [cth*sps, sph*sth*sps + cph*cps, -sph*cps + cph*sth*sps],
        [-sth, sph*cth, cph*cth]
    ])
    return rot_mat

# Rotation matrix: R_body_inertial: Inertial to body transformation
def rotmat_body_inertial(curr_angs):
    """
        Given `curr_angs`, return R_body_inertial: Rotation matrix
        transforming a vector in inertial frame, into the body frame.
        It is basically the inertial frame expressed in body frame.

        Note that: curr_angs = [phi, theta, psi]
            phi: Rot(X): Roll, theta: Rot(Y): Pitch, psi: Rot(Z): Yaw

        Just take transpose of rotmat_inertial_body
    """
    R_inertial_body = rotmat_inertial_body(curr_angs)
    R_body_inertial = R_inertial_body.T
    return R_body_inertial
```

```
# Body angular velocity to inertial angular velocity
def angvel_body_inertial(b_angvel, curr_angs):
    """
    Converts `b_angvel` (the angular velocity in the body frame) to
    angular velocities in the inertial frame (`omega` below).


    omega = M * (eul_dot)
    omega = [p, q, r]
          = [phi_dot, 0, 0] + Rot(X, -phi) * [0, theta_dot, 0]
              + Rot(X, -phi) * Rot(Y, -theta) * [0, 0, psi_dot]
          = M * [phi_dot, theta_dot, psi_dot]
    """
    ph, th, ps = curr_angs  # Body angles
    M = np.array([
        [1, 0, -np.sin(th)],
        [0, np.cos(ph), np.sin(ph)*np.cos(th)],
        [0, -np.sin(ph), np.cos(ph)*np.cos(th)]
    ])
    omega = M @ b_angvel.reshape(3, 1)
    return omega.flatten()  # As (3,)

# Using equations of coriolis, get the angular acceleration in {I}
def angacc_inertial_coriolis(omega, tau_b):
    """
        Given the torques acting on the body and the angular velocity
        in the inertial frame, calculate the angular acceleration in
        the inertial frame using the equations of coriolis.

        Parameters:
        - omega = [p, q, r]   Inertial angular velocity
        - tau_b = [t_x, t_y, t_z]   Torques acting on the body (in
                                    inertial)

        Returns:
        - omega_dot: Inertial angular acceleration

        J * (omega_dot) + omega x (J * omega) = tau_b

        Invert the above equation to get omega_dot
    """
    omega_dot = np.linalg.inv(J) @ (tau_b - np.cross(omega, J@omega))
    return omega_dot

# Convert inertial angular acceleration to body
def angacc_inertial_body(curr_ang, omega_dot):
    """
        Given the current euler angles (body orientation) - phi,
        theta, psi - convert the given inertial angular accelerations
        `omega_dot` into the body frame.

        Assuming that the change in the angle matrix `M` is small,
        the matrix `M` in `angvel_body_inertial` can be inverted and
        applied.

        Returns the angular acceleration in the body frame.
    """
    ph, th, ps = curr_ang   # Extract local {body} euler angles
    M_inv = np.array([
        [1, np.sin(ph)*np.tan(th), np.cos(ph)*np.tan(th)],
        [0, np.cos(ph), -np.sin(ph)],
        [0, np.sin(ph)/np.cos(th), np.cos(ph)/np.cos(th)]
    ])
    ang_acc_b = M_inv @ omega_dot
    return ang_acc_b

# %% Main simulation
# --- Simulation variables ---
time_vals = np.arange(start_time, end_time+dt, dt)
cpos = pos_init # Current position - [x, y, z] in m
cvel = vel_init # Current velocity - [x, y, z] in m/s
gvect = np.array([0., 0., g]) # Gravity in world (inertial) frame
cang = ang_init # Current angles (inertial) - phi, theta, psi
cangvel = ang_vel_init  # Current angular velocity (inertial)
max_angs = np.array([max_phi, max_theta, max_psi])
min_angs = -max_angs    # Minimum bound = -(Maximum bound)
```

```
angvel_d = np.array([0., 0., 0.])   # Desired ang. vel.
# --- Logging variables ---
pos_vals = []   # List of x, y, z positions
vel_vals = []   # List of x, y, z velocities
acc_vals = []   # List of x, y, z accelerations
des_acc_vals = []   # List of x, y, z accelerations (desired)
thrustd_vals = []   # List of desired thrust (for UAV propellers)
des_ang_vals = []   # List of phi, theta, psi desired - unclipped
des_ang_clipped_vals = []   # List of Ph, The, Ps desired - clipped
cur_ang_vals = []   # List of current angles - phi, theta, psi
motor_sp_vals = []  # List of motor speeds - n1, n2, n3, n4

SIM_STOP = float('inf') # For a breakpoint

# --- Main simulation ---
for num_i in tqdm(range(len(time_vals)), ncols=80):
    # Get thrust action (position error -> controller)
    pos_err = pos_d - cpos  # Position error
    vel_err = vel_d - cvel  # Velocity error
    des_acc = Kp_pos * pos_err + Kd_pos * vel_err   # PD action
    # Componsate for gravity pull (controller needs angle)
    des_acc[2] = (des_acc[2] - g)/(np.cos(cang[0])*np.cos(cang[1]))
    # Thrust needed (ideal) (in inertial frame)
    des_thrust = m*des_acc[2]
    # Calculate angle desired (from spherical to cartesian formulas)
    des_acc_mag = np.linalg.norm(des_acc)
    if des_acc_mag == 0:
        des_acc_mag = 1.0   # If no acceleration vector needed
    # print(f"{cang}")
    # print(f"{des_acc} \t {des_acc_mag}")
    des_ang = np.array([ # Desired angles in the inertial frame
        # Invert 1*sin(phi)*cos(theta) = acc_y_hat (unit vect.)
        np.arcsin(np.clip(des_acc[1] / des_acc_mag / np.cos(cang[1]),
            -0.95, 0.95)),    # asin needs only [-1, 1]
        # Invert sin(theta) = -acc_x_hat (unit vect.)
        np.arcsin(-des_acc[0] / des_acc_mag),
        # Psi is always desired to be zero
        0])  # Desired phi, theta, psi calculated
    # print(f"{des_ang[0]:.4f}, {des_acc[1]:.4f}, {des_acc_mag:.4f},"
    #     f" {cang[1]:.4f}")
    # Threshold angles (cap them)
    des_ang_clipped = np.clip(des_ang, min_angs, max_angs)
    # Get angle action (angle error -> controller)
    ang_err = des_ang_clipped - cang    # Angle error
    ang_vel_err = angvel_d - cangvel    # Angular velocity error
    des_angacc = Kp_ang * ang_err + Kd_ang * ang_vel_err    # PD act.
    # Get the motor speeds using torque and thrust equations
    m_speeds = angaccs_thr_to_motor_speeds(des_angacc, des_thrust)
    """
        Ideally, we would give `m_speeds` to an actual UAV and get
        the new values (for positions, velocities, etc.) from sensors.

        Here, we try 'simulating' a virtual UAV (using a mock physics
        model of a UAV) so that we can get states (according to how
        the system would behave).

        All variables of this virtual 'simulator' start with "uav_"
        so that it is easier to track them.
    """
    # print(f"{cang}")
    # -- Simulating a virtual UAV model --
    uav_thrust = -pkT * (m_speeds.sum())**2 # -ve because Z is down
    uav_R_ib = rotmat_inertial_body(cang)   # R_inertial_body -> B2I
    uav_R_bi = rotmat_body_inertial(cang)   # R_body_inertial -> I2B
    # Obtain linear acceleration in {inertial}
    uav_bodyf_B = np.array([0., 0., uav_thrust])  # Forces in {body}
    uav_bodyf_I = uav_R_ib @ uav_bodyf_B    # Force in {inertial}
    uav_weight_I = np.array([0, 0, m*g])    # Weight in {inertial}
    uav_linacc_I = (uav_bodyf_I + uav_weight_I)/m   # Lin. acc in I
    # Obtain body torques on the UAV (inverse to motor speeds)
    uav_btorque = mspeeds_btorque(m_speeds)
    # Obtain angular velocity in the inertial frame
    uav_angvel_I = angvel_body_inertial(cangvel, cang)  # Omega - {I}
    # print(f"{uav_angvel_I}")
    # Obtain angular acceleration in inertial frame using coriolis
    uav_angacc_I = angacc_inertial_coriolis(uav_angvel_I,
        uav_btorque)    # Omega_dot = p_dot, q_dot, r_dot in {I}
```

```
    # omega & omega_dot -> Angular acceleration (Euler) in {body}
    uav_angacc_B = angacc_inertial_body(cang, uav_angacc_I)
    """
        We are using pure odometry (multiply dt and acceleration to
        get velocity, multiply dt and velocity to get position). This
        would be handled by the 'environment' in real.
    """
    # -- Variable updates --
    cangvel += uav_angacc_B * dt    # Angular velocity update
    cang += cangvel * dt     # Angle update (from new ang. vel.)
    cang = np.clip(cang, min_angs, max_angs)
    cvel += uav_linacc_I * dt    # Linear velocity (from acc. in {I})
    cpos += cvel * dt       # Position in {world}
    # -- Log all values --
    pos_vals.append(cpos.copy())
    vel_vals.append(cvel.copy())
    acc_vals.append(uav_linacc_I.copy())
    des_acc_vals.append(des_acc.copy())
    thrustd_vals.append(des_thrust.copy())
    des_ang_vals.append(des_ang.copy())
    des_ang_clipped_vals.append(des_ang_clipped.copy())
    cur_ang_vals.append(cang.copy())
    motor_sp_vals.append(m_speeds.copy())
    # Testing environment
    # print(f"Testing environment")
    if num_i > SIM_STOP:   # FIXME: This is used only when not 'inf'
        break
# Convert all logs to numpy
pos_vals = np.array(pos_vals)
vel_vals = np.array(vel_vals)
acc_vals = np.array(acc_vals)
des_acc_vals = np.array(des_acc_vals)
thrustd_vals = np.array(thrustd_vals)
des_ang_vals = np.array(des_ang_vals)
des_ang_clipped_vals = np.array(des_ang_clipped_vals)
cur_ang_vals = np.array(cur_ang_vals)
motor_sp_vals = np.array(motor_sp_vals)

# %% Experimental
# print(f"Desired angles (clipped): {np.rad2deg(des_ang_clipped)}")
# print(f"Desired thrust (up): {des_thrust}")

# %%

# %% View all plots
SIM_STOP = len(time_vals)   # If everything above went successful

# %% Position plots
plt.figure()
plt.title("Position")
l = plt.plot(time_vals[:SIM_STOP], pos_vals[:SIM_STOP, 0], label="X")
plt.axhline(pos_d[0], lw=0.5, c=l[0].get_color())
l = plt.plot(time_vals[:SIM_STOP], pos_vals[:SIM_STOP, 1], label="Y")
plt.axhline(pos_d[1], lw=0.5, c=l[0].get_color())
l = plt.plot(time_vals[:SIM_STOP], pos_vals[:SIM_STOP, 2], label="Z")
plt.axhline(pos_d[2], lw=0.5, c=l[0].get_color())
plt.legend()
plt.savefig("./position.jpg")
plt.show(block=False)


# %% Velocity plots
plt.figure()
plt.title("Velocity")
plt.plot(time_vals[:SIM_STOP], vel_vals[:SIM_STOP, 0], label="X")
plt.plot(time_vals[:SIM_STOP], vel_vals[:SIM_STOP, 1], label="Y")
plt.plot(time_vals[:SIM_STOP], vel_vals[:SIM_STOP, 2], label="Z")
plt.legend()
plt.savefig("./velocity.jpg")
plt.show(block=False)

# %% Acceleration plots
plt.figure()
plt.title("Acceleration")
plt.plot(time_vals[:SIM_STOP], acc_vals[:SIM_STOP, 0], label="X")
plt.plot(time_vals[:SIM_STOP], acc_vals[:SIM_STOP, 1], label="Y")
plt.plot(time_vals[:SIM_STOP], acc_vals[:SIM_STOP, 2], label="Z")
```

```python
plt.legend()
plt.savefig("./acceleration.jpg")
plt.show(block=False)

# %% Acceleration (desired) plots
plt.figure()
plt.title("Acceleration desired")
plt.plot(time_vals[:SIM_STOP], des_acc_vals[:SIM_STOP, 0], label="X")
plt.plot(time_vals[:SIM_STOP], des_acc_vals[:SIM_STOP, 1], label="Y")
plt.plot(time_vals[:SIM_STOP], des_acc_vals[:SIM_STOP, 2], label="Z")
plt.legend()
plt.savefig("./acceleration_d.jpg")
plt.show(block=False)

# %% Thrust desired value
plt.figure()
plt.title("Thrust desired")
plt.plot(time_vals[:SIM_STOP], thrustd_vals[:SIM_STOP])
plt.savefig("./thrust_d.jpg")
plt.show(block=False)

# %% Euler angles (desired)
plt.figure()
plt.title("Euler desired")
plt.plot(time_vals[:SIM_STOP], des_ang_clipped_vals[:SIM_STOP, 0],
    label=r"$\phi$")
plt.plot(time_vals[:SIM_STOP], des_ang_clipped_vals[:SIM_STOP, 1],
    label=r"$\theta$")
plt.plot(time_vals[:SIM_STOP], des_ang_clipped_vals[:SIM_STOP, 2],
    label=r"$\psi$")
plt.legend()
plt.savefig("./euler_d.jpg")
plt.show(block=False)

# %% Euler angles (actual)
plt.figure()
plt.title("Euler (actual)")
plt.plot(time_vals[:SIM_STOP], cur_ang_vals[:SIM_STOP, 0],
    label=r"$\phi$")
plt.plot(time_vals[:SIM_STOP], cur_ang_vals[:SIM_STOP, 1],
    label=r"$\theta$")
plt.plot(time_vals[:SIM_STOP], cur_ang_vals[:SIM_STOP, 2],
    label=r"$\psi$")
plt.legend()
plt.savefig("./euler.jpg")
plt.show(block=False)

# %% Motor speeds
plt.figure()
plt.title("Motor speeds")
plt.plot(time_vals[:SIM_STOP], motor_sp_vals[:SIM_STOP, 0],
    label="n1")
plt.plot(time_vals[:SIM_STOP], motor_sp_vals[:SIM_STOP, 1],
    label="n2")
plt.plot(time_vals[:SIM_STOP], motor_sp_vals[:SIM_STOP, 2],
    label="n3")
plt.plot(time_vals[:SIM_STOP], motor_sp_vals[:SIM_STOP, 3],
    label="n4")
plt.legend()
plt.savefig("./motor_speeds.jpg")
try:
    plt.show()
except KeyboardInterrupt as exc:
    print(f"Keyboard interrupt received")
```