Control of a 2D Quadrotor: Approach, Implementation, Results, and Discussion

Amir Iqbal

Problem Statement

Desired drone behavior:

- Move the drone counterclockwise (yaw rate is pointing outward) in a circle on the XY plane with center at origin (0, 0), radius 1.0 m, and a user specified desired speed v_d along the heading direction (i.e., along body frame x-direction) or user specified desired yaw rate ω_d .

Kinematic equation in the world frame:

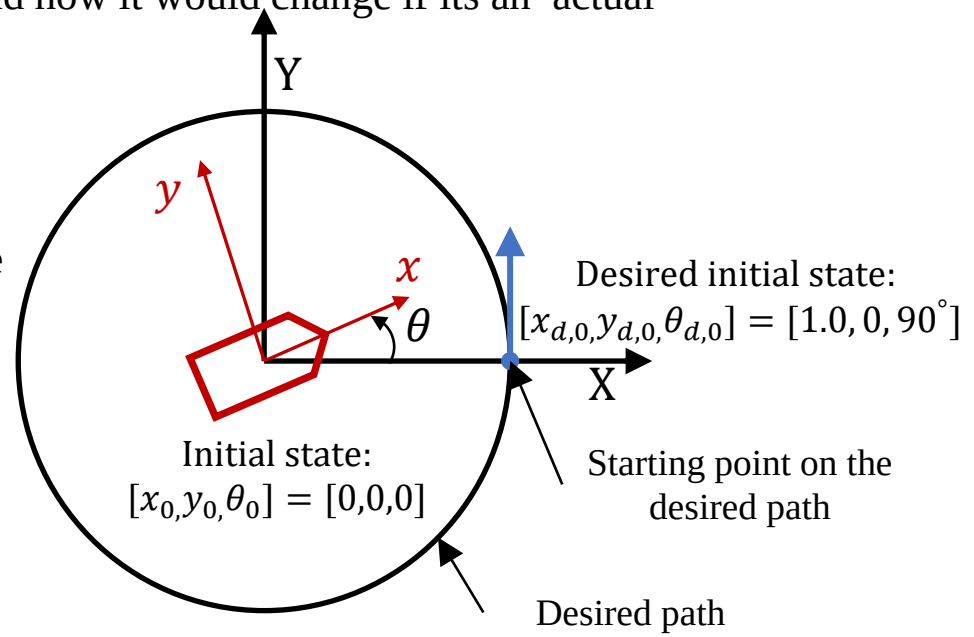
- $\dot{x}_d = v_d \cos\theta = v_{d,x}$, $\dot{y}_d = v_d \sin\theta = v_{d,y}$, and $\dot{\theta}_d = \omega_d = \omega_{d,z}$.

Tasks:

1. Write a ROS publisher that publish this trajectory at 10Hz.
2. Write a ROS subscriber that subscribes to this trajectory and publishes the control commands to the quadrotor to track the circular trajectory.
3. Write a short note (or discuss) on the pros/cons of your approach and how it would change if its an actual quadrotor tracking a 3D trajectory.

Observations

- For circular motion, v_d and ω_d are related, i.e., $v_d = \omega_d$R, where R is the radius of the circle.
- Therefore, we only need any two of the parameters v_d, ω_d, and R
- In the given problem R=1.0 m, we can ask user to input either v_d or ω_d
  - I will take v_d and R as input and compute ω_d as: $\omega_d = \frac{v_d}{R}$



Problem Solution Outline

Desired and current 2D pose trajectory in the world frame:

- Let $\dot{x}_d = v_{d,x}$, $\dot{y}_d = v_{d,y}$, and $\dot{\theta}_d = \omega_{d,z}$, then the desired 2D pose trajectories are given as:
  $x_d(t) = x_{d,0} + \int_0^t v_{d,x}(t)dt$, $y_d(t) = y_{d,0} + \int_0^t v_{d,y}(t)dt$, and $\theta_d(t) = \theta_{d,0} + \int_0^t \omega_{d,z}(t)dt$.
- The current pose of the drone can be obtained by integrating the commanded velocity along with given initial pose as:
  $x_c(t) = x_0 + \int_0^t v_x(t)dt$, $y_c(t) = y_0 + \int_0^t v_y(t)dt$, and $\theta_c(t) = \theta_0 + \int_0^t \omega_z(t)dt$.

Control commands to track the desired 2D pose trajectories:

- I chose to design the controller to be a simple proportional derivative (PD) controller to augment x-velocity v_x, y-velocity v_y, and yaw rate ω_z based on tracking error.
- **Feedback inputs:** $u_x(t) = k_{p,x}\big(x_d(t) - x_c(t)\big) + k_{d,x}(vx_d(t) - vx_c(t))$, $u_y(t) = k_{p,y}\big(y_d(t) - y_c(t)\big) + k_{d,y}(vy_d(t) - vy_c(t))$, and $u_\omega(t) = k_{p,\omega}\big(\theta_d(t) - \theta_c(t)\big) + k_{d,\omega}(\omega_d(t) - \omega_c(t))$.
- **Closed-loop velocity commands:** $v_x = v_x + u_x$, $v_y = v_y + u_y$, and $\omega_z = \omega_z + u_\omega$.

Implementation:

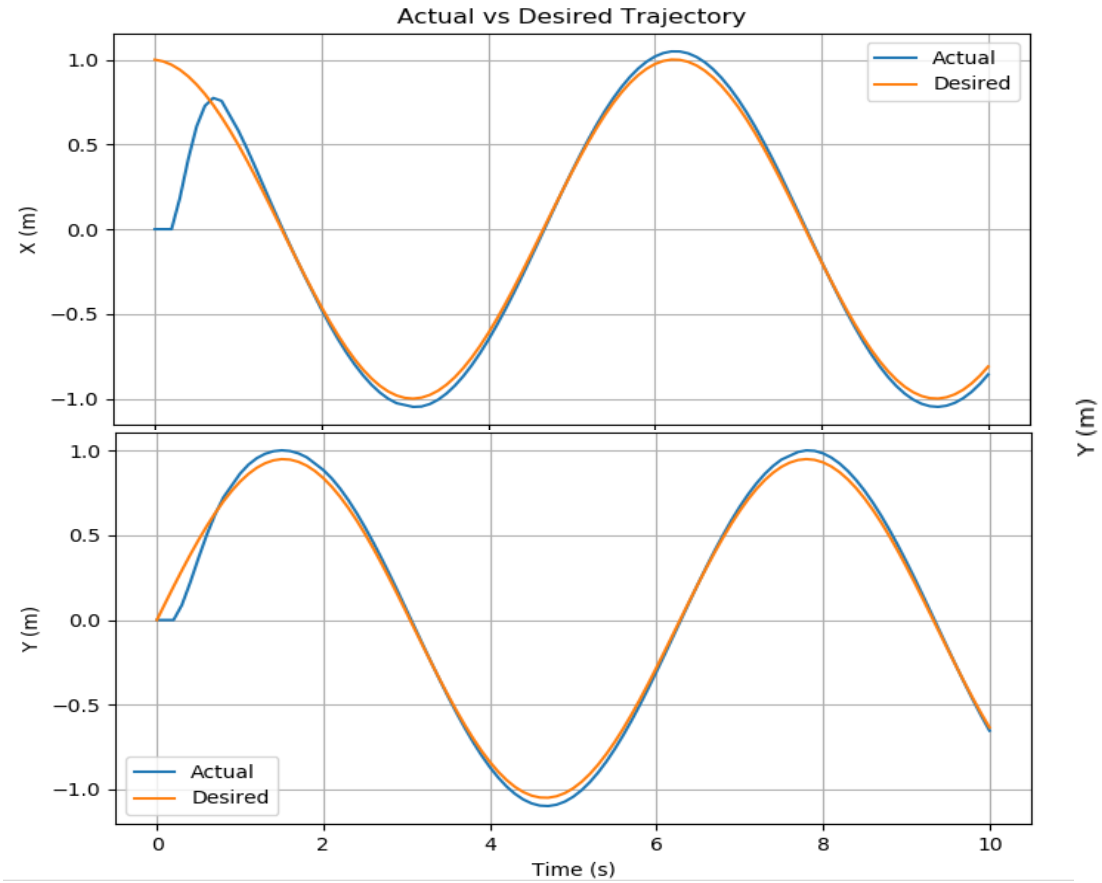
- Discretize the time into steps equal to control loop frequency and numerically integrate to get the desired and current (actual) 2D pose of the drone
- Command velocities computed by the closed-loop feedback control

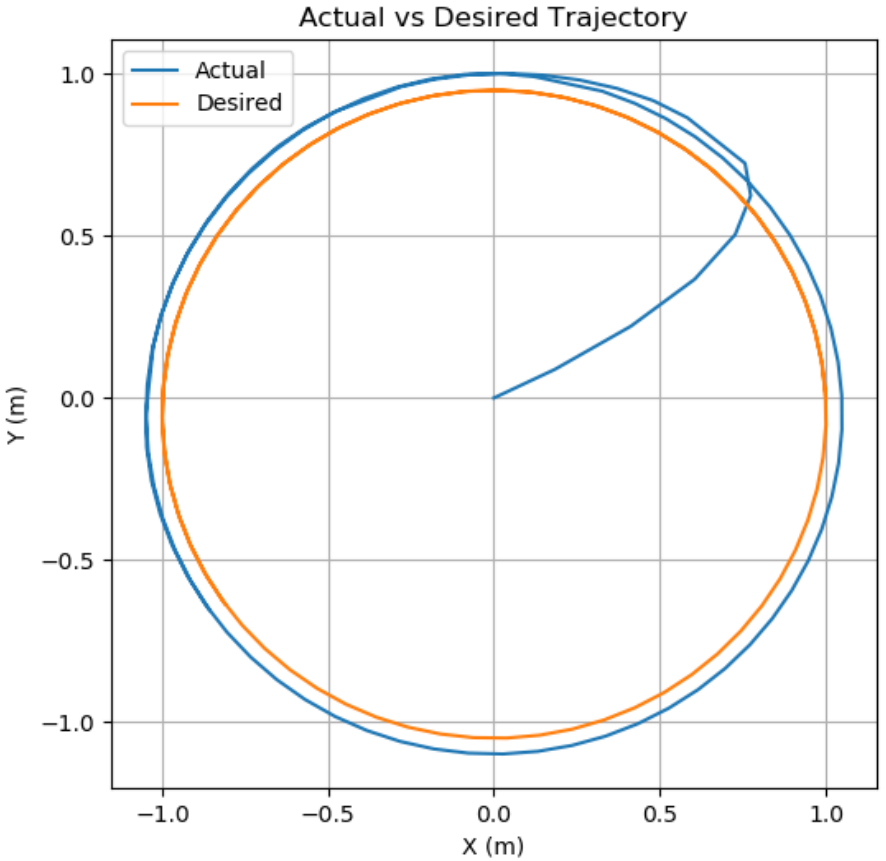
My initial thoughts:

- Numerical implementation will result in approximate tracking on the desired circle, i.e., a circle approximated by polygons with the number of sides = number of control points for completing a circle
- Therefore, a faster control frequency or a smaller speed will result in better trajectory tracking and smoother circle

Results

Control loop rate =10 Hz;
Control gains: kp_x = 2.0, kp_y = 2.0 kp_theta = 1.0, kd_x = 0.50, kd_y = 0.50 kd_theta = 0.1





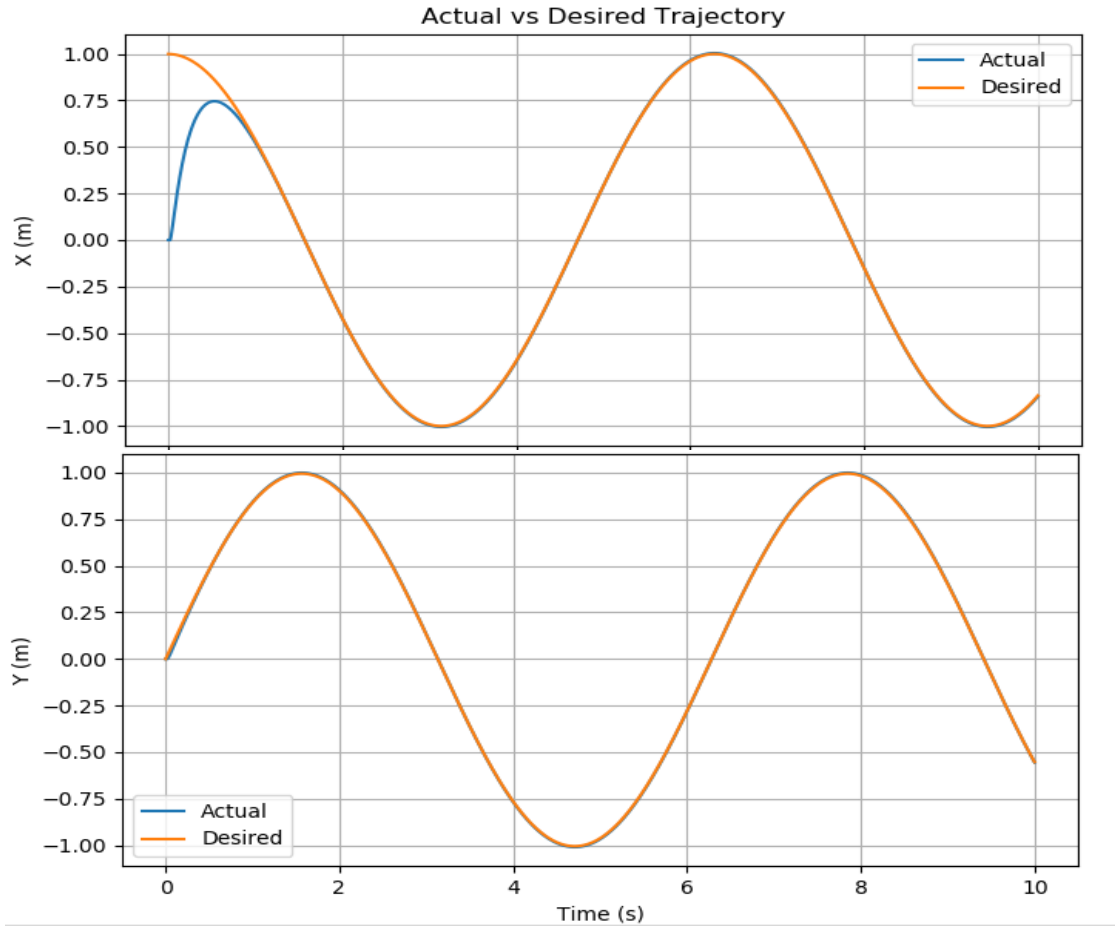
Observation: There seems to be steady state error as evident by offset between actual and desired curve in XY plane

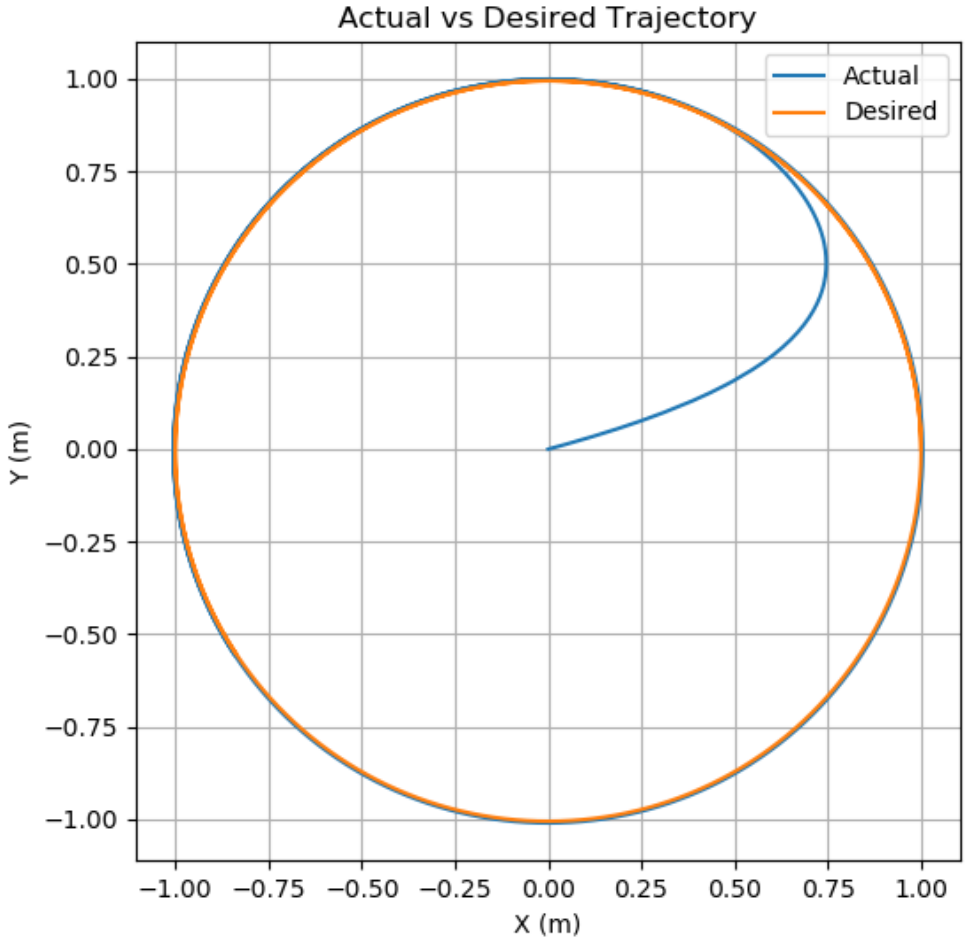
- By using integral control law, we can overcome this offset

Results

Control loop rate =100 Hz;

Control gains: kp_x = 2.0, kp_y = 2.0 kp_theta = 1.0, kd_x = 0.50, kd_y = 0.50 kd_theta = 0.1



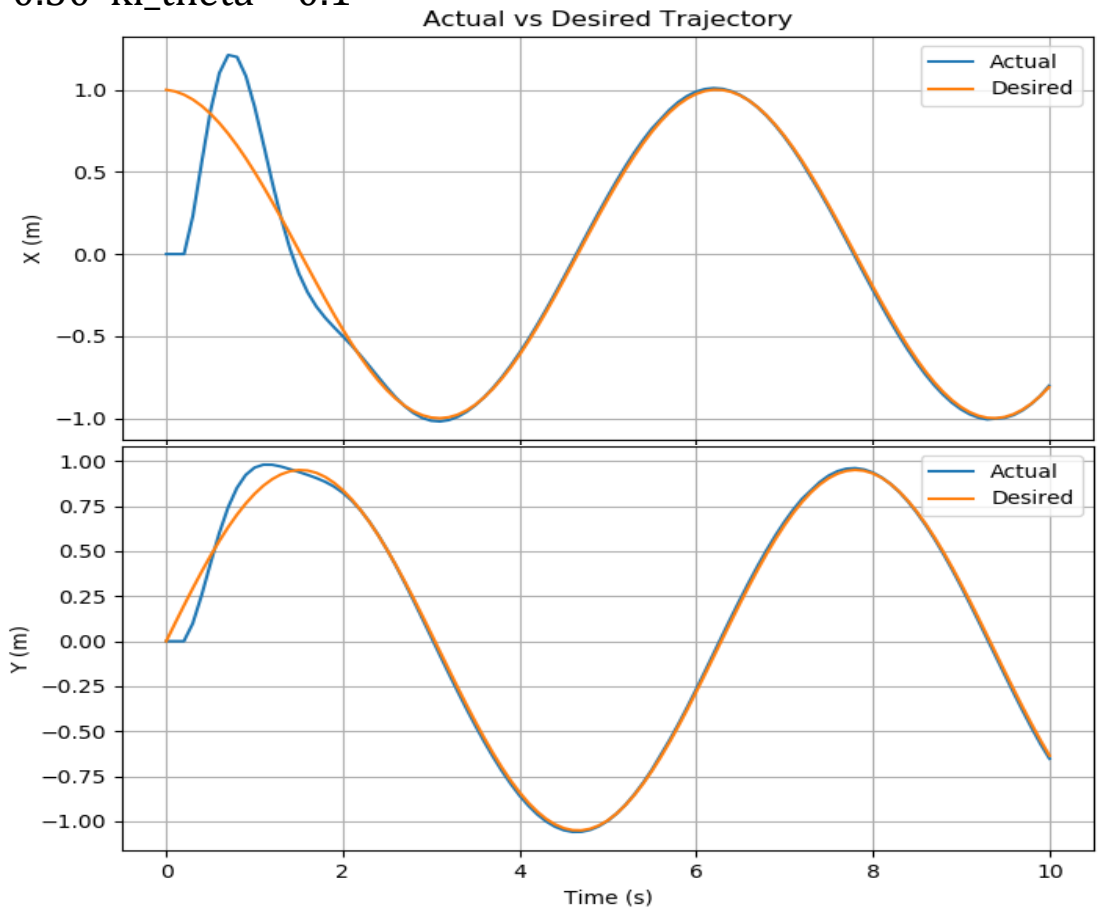


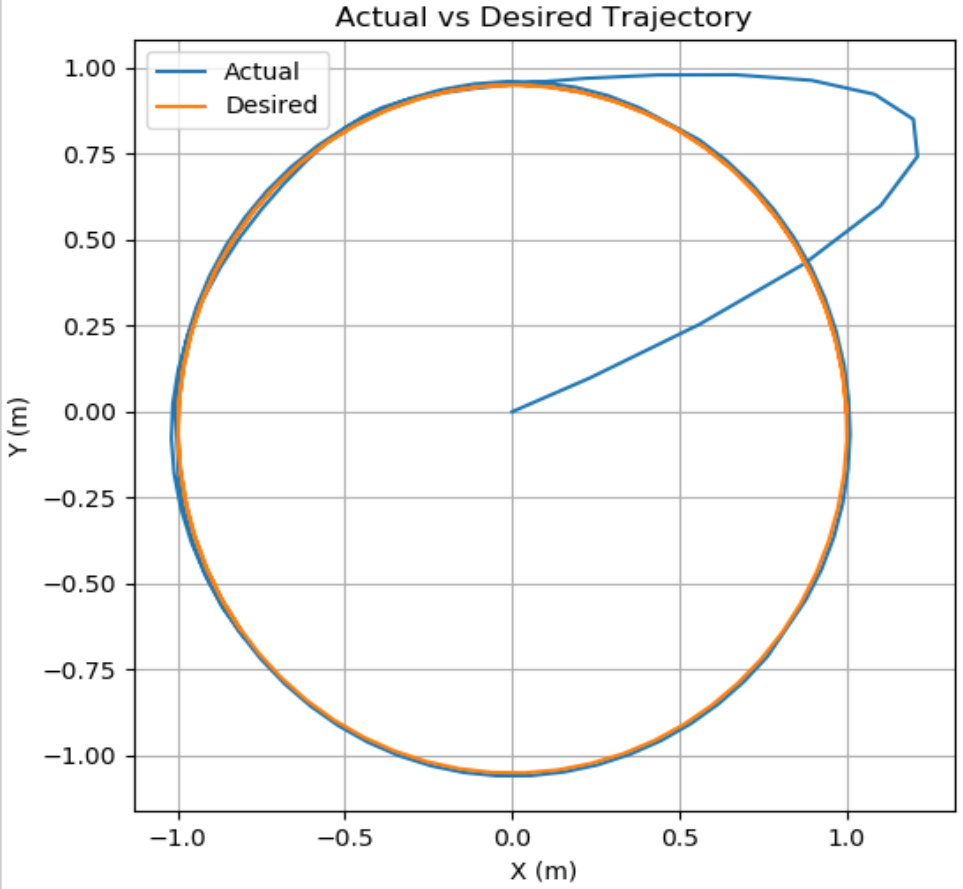
Observation: Under same PD gains, if we increase the control update frequency from 10 Hz to 100 Hz the steady state error seems to vanish.

Results

Control loop rate =10 Hz; Added integral control law to eliminate steady state error

Control gains: kp_x = 2.0, kp_y = 2.0 kp_theta = 1.0, kd_x = 0.50, kd_y = 0.50 kd_theta = 0.1, ki_x = 0.50, ki_y = 0.50 ki_theta = 0.1





Observation: The steady state error are eliminated by adding an integral control to the originally designed PD controller

- Use of integral law resulted in significant overshoot that can be eliminated by proper tuning of the control gains

Pros and Cons of the Implementation

Pros:

- **Modular design:** The code is organized into a class-based structure, promoting modularity and reusability.
- **ROS integration:** Utilizing ROS for communication allows seamless interaction with other ROS nodes, sensors, and actuators.
- **Feedback control:** The control mechanism uses proportional, derivative, and integral (PID) feedback to adjust the quadrotor's velocity to track the desired trajectory
- **Visualization:** The graphical presentation of results enables easy visualization of actual and desired trajectories under the closed-loop control

Cons:

- **Assumptions:** The implementation assumes perfect execution of control commands without considering real-world limitations
- **Lack of dynamics:** The implementation considers a given kinematic model, which does not account for the quadrotor's dynamics, potentially leading to inaccuracies in the simulation.
- **Limited realism:** The implementation does not account for real-world factors like sensor noise, motor dynamics, delays, atmospheric turbulence
- **Simplified control:** The control strategy is a classical PID control, which might not be suitable for complex objectives and uncertain scenarios.

Implementation Tweaks for Actual Quadrotor and 3D Trajectory

Implementation tweak from 2D to a 3D quadrotor

- **Augment state vector:** A 3D quadrotor will have six degrees of freedom for the body frame,
- **Update kinematic model:** A 3D quadrotor will have a significantly complex kinematic model
- **Update control law:** The control law must be modified to give control inputs to all six trajectories

Implementation on an actual quadrotor

- **Hardware preparation:** Model the hardware dynamics, actuators, sensors, and ensure that the quadrotors API is functioning as expected
- **Real-world uncertain factors:** Account for sensor noise, motor dynamics, delays, atmospheric drag, etc.
- **State estimation:** In real-world operation, the current state of the quadrotor needs to be estimated based on some filters such as KF, EKF, or InEKF to ensure robust state estimates under noisy sensor readings
- **Transform control input:** For real quadrotors, the inputs are usually rotor speed, so the controller should transform the velocity commands into rotor speed command
- **Incorporate dynamics:** Including quadrotor's dynamics in the model will potentially improve the model's accuracy
- **Advanced control scheme:** While a classical PID control may be adequate under nominal conditions, using an advanced modern control technique such as QP or MPC can ensure performance under uncertainty and constraints, which will be more suitable for complex objectives and uncertain scenarios in real-world operation

Conclusions

- Implemented ROS Publisher to publish the desired trajectory
- Implemented ROS Subscriber that subscribes to the desired trajectory and publishes the control commands to the quadrotor
- The call back function first implemented a PD control and later modified to a PID control for eliminating steady state error
- Presented tracking results under different control frequency of a PD control and a PID control
- Briefly described the pros and cons of my implementation and discussed the modifications required for a 3D trajectory and actual quadrotors.
- The source code for the implementation is supplied separately as a Python script "Control_of_2D_Drone_v2.py". Dependencies: rospy, geometry_msg, matplotlib, numpy
- To run the code– (a) open two terminals, (b) run "roscore" in a terminal, and (b) run the script in the other terminal

I will be happy to answer any questions on my implementation, approach, or discussion on results. Thank you.

Screenshots of a Trial Run: Code Execution and Output