Tracking a Car using a Drone with a Camera

Angela Voo

December 2024

1 Abstract

This project simulates a drone equipped with a pan-tilt camera tracks a moving car. The quadcopter, modeled as a rigid-body system, uses a Linear Quadratic Regulator (LQR) for optimal control to maintain the car centered within the camera's field of view. The car's dynamics are based on the Ackerman steering model, used in the motion planner which predicts the car's future positions, generating a trajectory for the drone to intercept it while adjusting camera angles to maintain visual alignment. The simulation evaluates the system's performance under three car paths: linear, circular, and spiral. Results demonstrate the drone's capability to dynamically track the car using the LQR-based control strategy.

2 Introduction

In our final project, we simulate a drone tracking a car from above using a pan-tilt camera modeled as a pinhole camera. This has many applications, such as reconnaissance for military operations or shooting a movie. The drone uses LQR to control its movement to keep the car centered in the camera field of view as best as possible.

3 Methods

3.1 Drone dynamics

The quadcopter's dynamics are modeled as a rigid-body system operating in three-dimensional space. The drone's state is the 12-dimensional vector:

$$\mathbf{x} = [x, \dot{x}, y, \dot{y}, z, \dot{z}, \phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}]^T \tag{1}$$

where (x, y, z) are the drone's global coordinates, (ϕ, θ, ψ) are the roll, pitch, and yaw angles, and their derivatives represent their respective angular velocities. The control inputs consist of four rotor thrusts (u_1, u_2, u_3, u_4) which generate forces and torques acting on the drone. The dynamics incorporate translational and rotational motions:

$$\ddot{x} = \frac{1}{m}(F_1 + F_2 + F_3 + F_4)(\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi) \tag{2}$$

$$\ddot{y} = \frac{1}{m}(F_1 + F_2 + F_3 + F_4)(\cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi) \tag{3}$$

$$\ddot{z} = \frac{1}{m}((F_1 + F_2 + F_3 + F_4)\cos\phi \cos\theta - mg) \tag{4}$$

$$\ddot{\phi} = \frac{1}{I_x}((F_1 - F_3)l - \dot{\theta}\dot{\psi}(I_y - I_z)) \tag{5}$$

$$\ddot{\theta} = \frac{1}{I_y}((F_2 - F_4)l - \dot{\psi}\dot{\phi}(I_z - I_x)) \tag{6}$$

$$\ddot{\psi} = \frac{1}{I_z}(M_2 + M_4 - M_1 - M_3 + \dot{\phi}\dot{\theta}(I_x - I_y)) \tag{7}$$

where m is the drone mass, g is the gravitational acceleration, I_x, I_y, I_z are moments of inertia, and l is the distance from the center to each rotor. For additional details on the drone dynamics, please refer to reference [1].

3.2 Car Dynamics

The car motion follows a path controlled by a constant speed v and a changing steering angle α. The vehicle's state at any time step k is represented by the state vector:

$$\boldsymbol{x}_k = \begin{bmatrix} x \\ y \\ \theta \\ v \\ \alpha \\ z \end{bmatrix} \tag{8}$$

where x and y are the position coordinates, θ is the orientation angle, v is the linear velocity input, α is the steering angle input, and z represents a constant height. Below are the equations of motion that describe the car using the Ackerman steering kinematic model:

$$\dot{x} = v\cos\theta \tag{9}$$

$$\dot{y} = v\sin\theta \tag{10}$$

$$\dot{\theta} = \frac{v}{L}\tan\alpha \tag{11}$$

This used to derive the discrete dynamic model below:

$$x_{k+1} = x_k + v\cos(\theta_k)dt \tag{12}$$

$$y_{k+1} = y_k + v\sin(\theta_k)dt \tag{13}$$

$$\theta_{k+1} = \theta_k + \frac{v}{L}\tan(\alpha_k)dt \tag{14}$$

where L is the vehicle's wheelbase.

3.3 Motion Planning

The motion planner gives each next set point for the drone to create a trajectory for the drone to intercept the car position. Using the car's dynamics in equations 12-14, the positions of the car at future time steps can be predicted (over a 5-second horizon). The estimated time for the drone to reach each future car position is computed using the drone's maximum speed. The motion planner then identifies the time step where the drone and car's predicted trajectories align most closely. The predicted (x, y) car position is used as the drone's target position for the next set point, which is fed to the LQR controller.

The motion planner also computes the pan and tilt of the camera based on the car's relative position to keep car centered on the camera field-of-view. The camera's pan and tilt angles are updated by transforming the car's target position into the camera's frame using the transformation matrix. The use of rotation matrices and homogeneous transformations ensures consistency in calculations across different frames of reference (global, drone, and camera).

3.4 LQR Optimal Set Point Control

The objective of the Linear Quadratic Regulator (LQR) controller is to drive the controlled output state z of the drone to a given set point r computed by the motion planner. The system dynamics are defined as:

$$\dot{\tilde{x}} = A\tilde{x} + B\tilde{u}. \tag{15}$$

$$\dot{\tilde{z}} = G\tilde{x} + H\tilde{u}. \tag{16}$$

where $\tilde{x} = x - x_{eq}$, $\tilde{z} = z - r$ and $\tilde{u} = u - u_{eq}$, for an equilibrium such that:

$$Ax_{eq} + Bu_{eq} = 0 \tag{17}$$

$$r = Gx_{eq} + Hu_{eq} \tag{18}$$

The LQR controller minimizes a quadratic cost function:

$$J = \int_0^\infty \tilde{z}(t)^T Q\tilde{z}(t) + \tilde{u}(t)^T R\tilde{u}(t)\, dt \tag{19}$$

where Q penalizes state deviations and R penalizes control effort. The optimal control law is:

$$u(t) = -K(x(t) - x_{eq}) + u_{eq} \tag{20}$$

where u_{eq} in this problem is the control input needed to keep the drone at a specific position (height $z > 0$). K is derived from solving for P from the algebraic Riccati equation:

$$K = R^{-1}B^T P. \tag{21}$$

$$A^T P + PA - PBR^{-1}B^T P + Q = 0, \tag{22}$$

The resulting control ensures smooth and optimal trajectory tracking.

3.5 Camera Sensor Model

The pan-tilt camera is modeled as a pinhole camera, with pan (ψ) and tilt (ϕ) angles. The transformation matrices for tilt ($\boldsymbol{R}_\phi$) and pan ($\boldsymbol{R}_\psi$) rotations are given by:

$$\boldsymbol{R}_\phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \tag{23}$$

$$\boldsymbol{R}_\psi = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{24}$$

The projected image coordinates p_x and p_y of the car state $\boldsymbol{x}_t$ onto the virtual image plane are:

$$p_x = \lambda\frac{q_x}{q_z}, \quad p_y = \lambda\frac{q_x}{q_z} \tag{25}$$

where λ is the camera focal length, $\boldsymbol{x}_c$ is the position of the camera (pinhole) in the inertial frame and the components of $\boldsymbol{q}_t$ are defined as:

$$\boldsymbol{q}_t \triangleq \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} = \boldsymbol{R}_\phi^T \boldsymbol{R}_\psi^T([\boldsymbol{x}_t^T 0]^T - \boldsymbol{x}_c] \tag{26}$$

These coordinates are valid only if the projected point lies within the image plane boundaries. We assume that the camera does not have any noise.

4 Procedure

The simulation demonstrates the drone's ability to dynamically track the car by combining realistic drone and car dynamics with an efficient control strategy. The car is simulated with three different paths: line, circle, and spiral. Over the given time of the simulation, the simulation is run for each timestep dt.

1. **Initialization:**
   - Define initial states and physical parameters for the drone and car.
   - Set camera parameters such as field of view and focal length.
   - Compute LQR gains using a linearized state-space model.
2. **Iterate:**
   - Record sensor measurements and tracking errors at each time step
   - Update car dynamics using the car dynamic model
   - Calculate the drone's desired state using a lookahead strategy to predict the car's position
   - Use LQR to compute the control inputs for the drone
   - Update the drone's position and adjust the camera's pan and tilt angles to center the car in the field of view
3. **Visualization:**
   - Track the drone and car paths in 2D and 3D space.
   - Plot the drone's x, y, and z positions over time.
   - Visualize camera measurements to assess tracking accuracy.

5 Results

5.1 Point-to-Point Setpoint Control

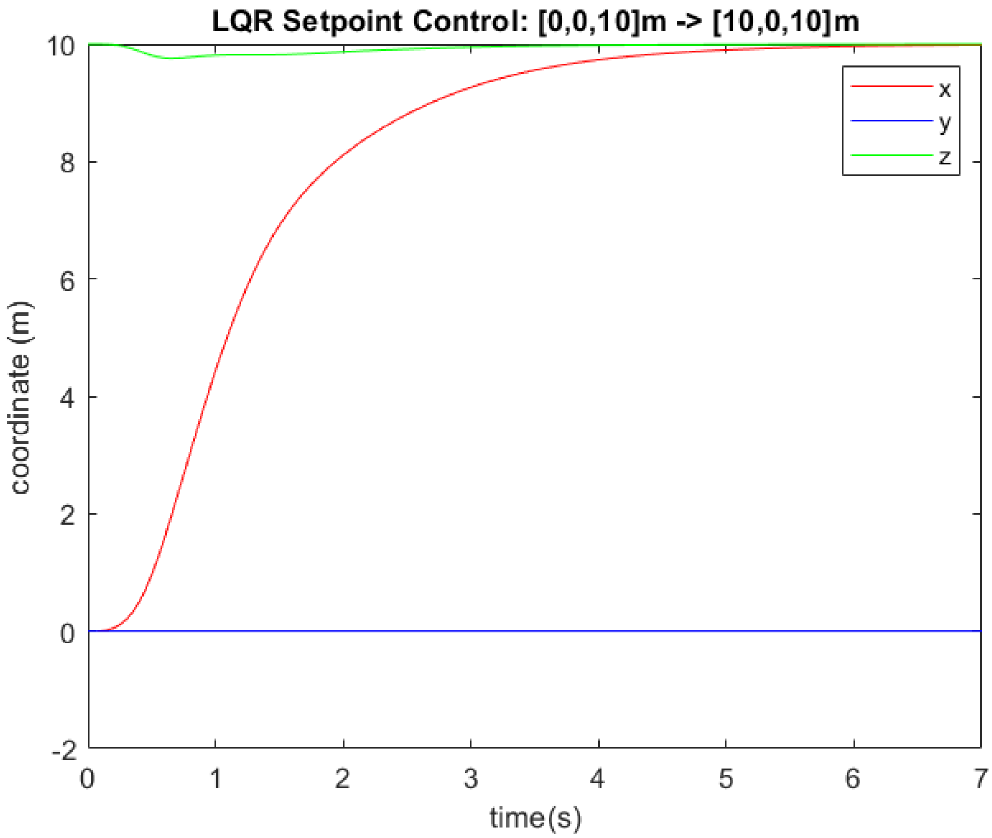

Figure 1: Position of drone from initial position to a setpoint

The rise time is about 1.8 seconds and the settling time is about 5.5 seconds.

5.2 Car Path: Line

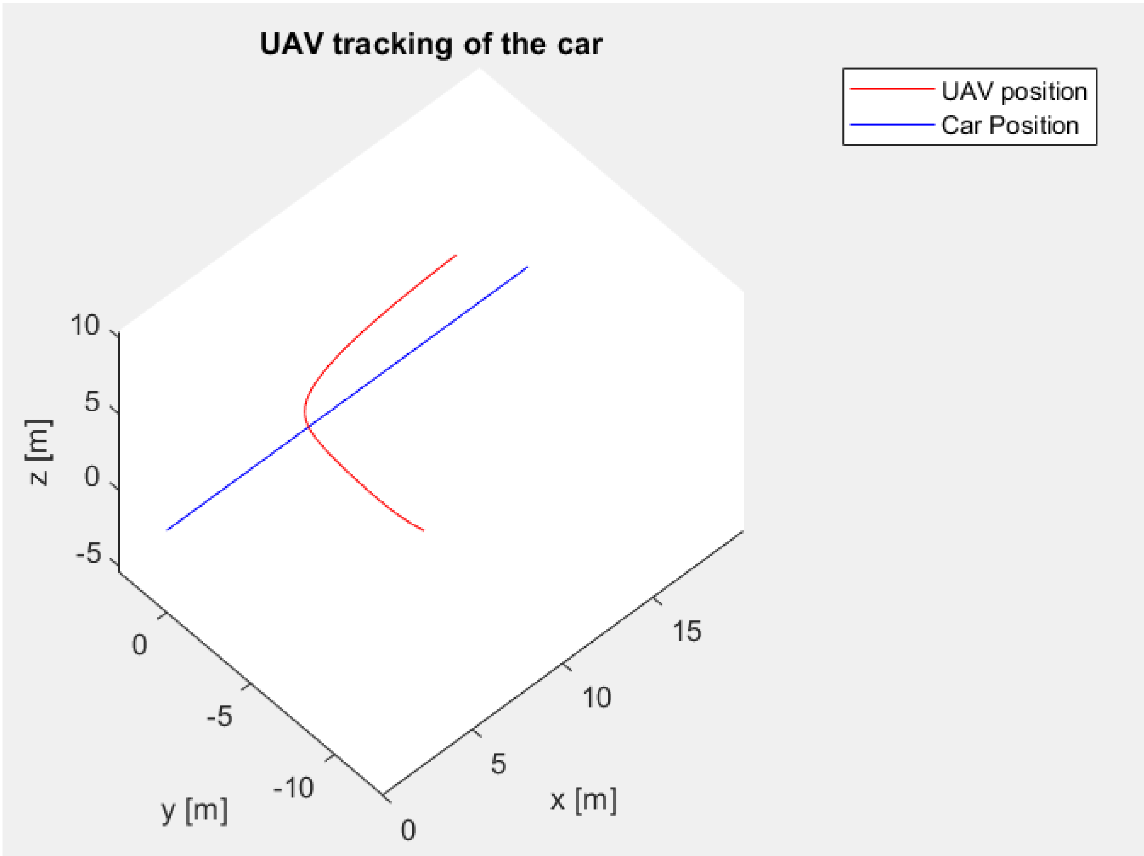

Figure 2: Car driving in a line and the corresponding drone trajectory

The plot shows the drone intercepting the straight-line path of the car on an optimal trajectory.

5.3 Car Path: Circle

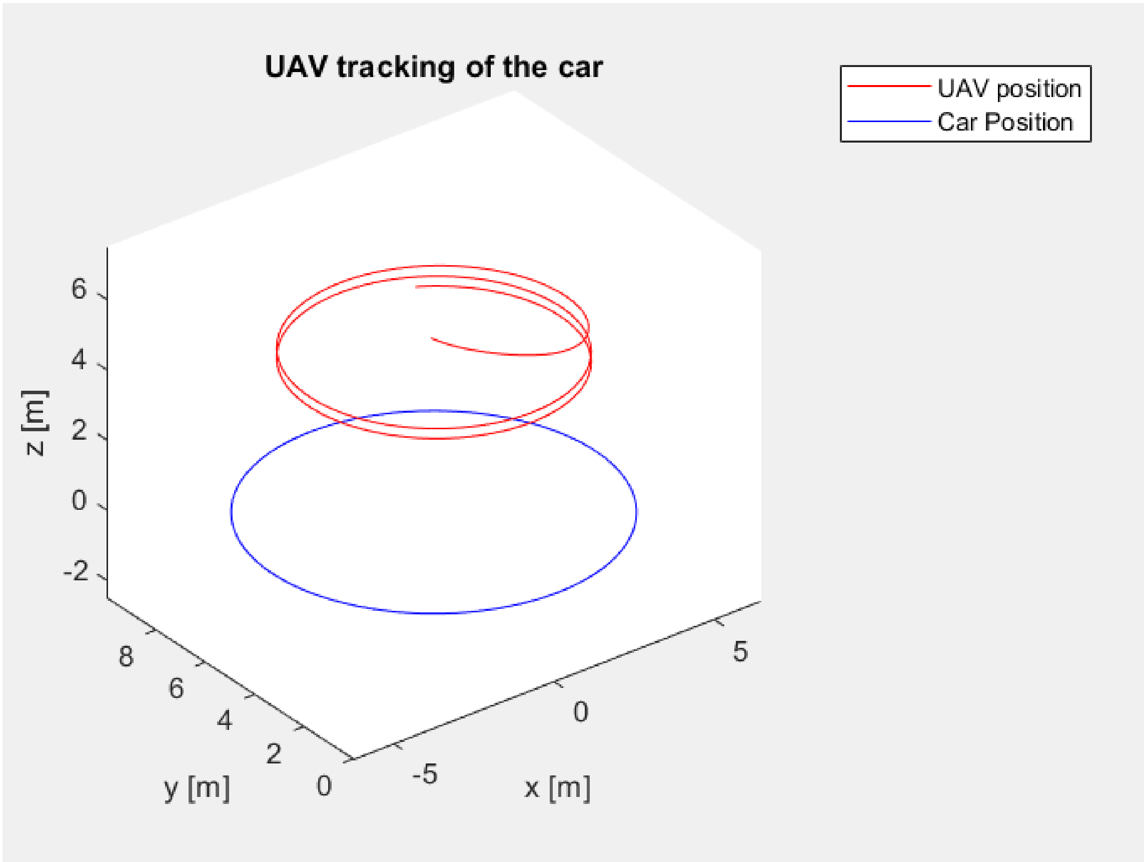

Figure 3: Car driving in a circle and the corresponding drone trajectory

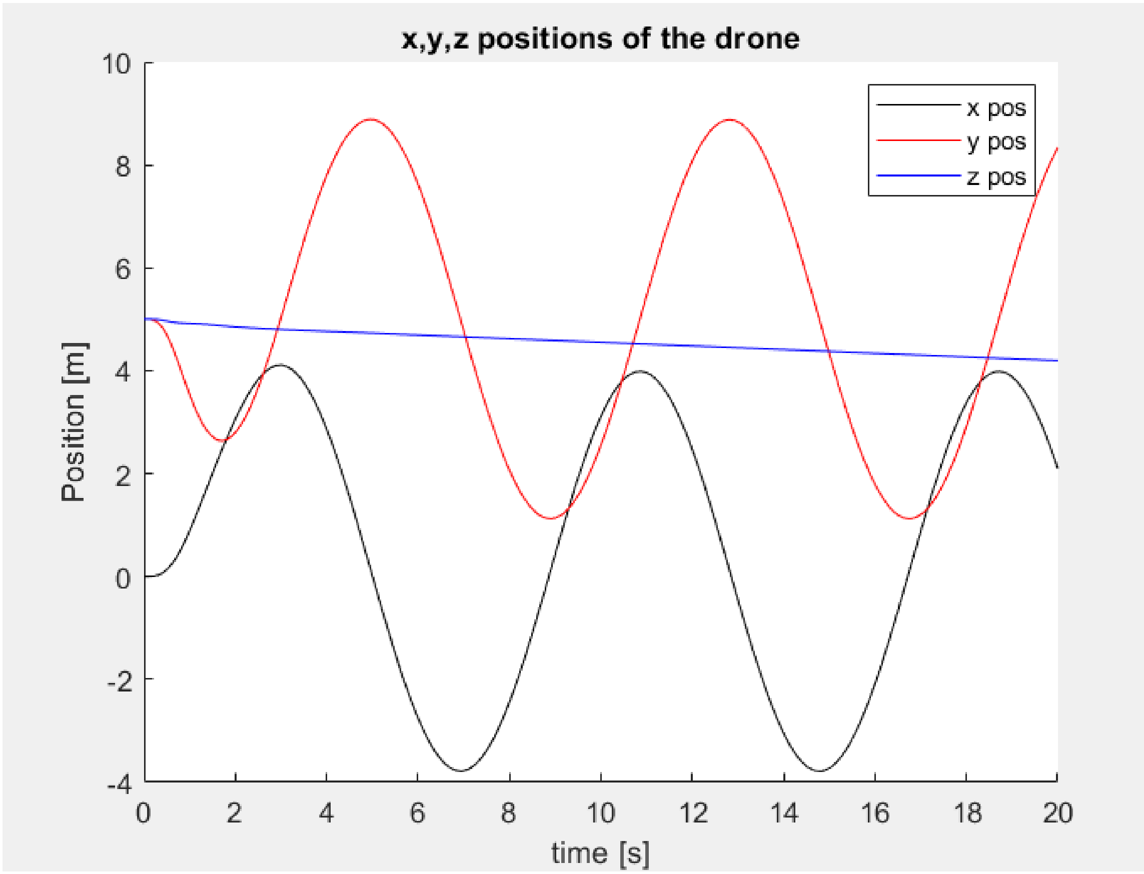

Figure 4: 3D position of drone while tracking a car driving in a circle

You can see in the graph that the drone's height dips a little as it tries to stay at a constant height while tracking the car.

5.4 Car Path: Spiral

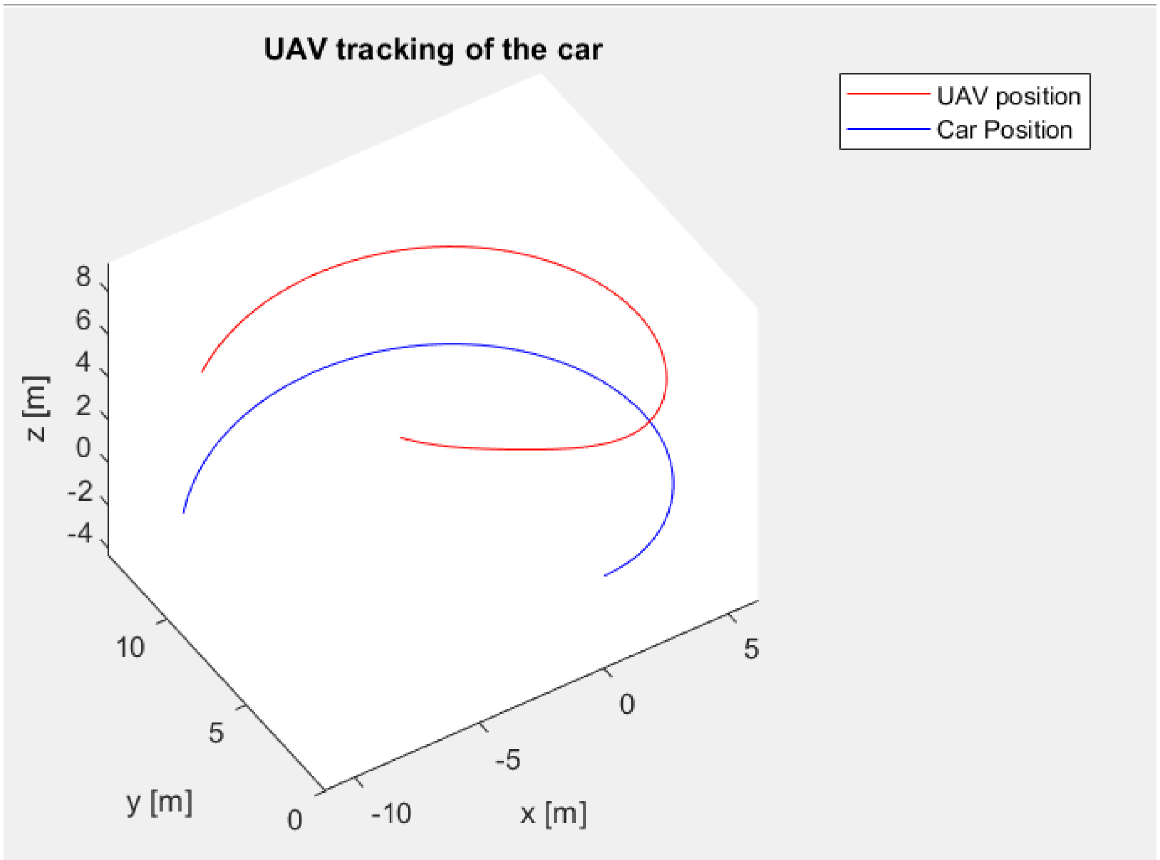

Figure 5: Car driving in a spiral and the corresponding drone trajectory

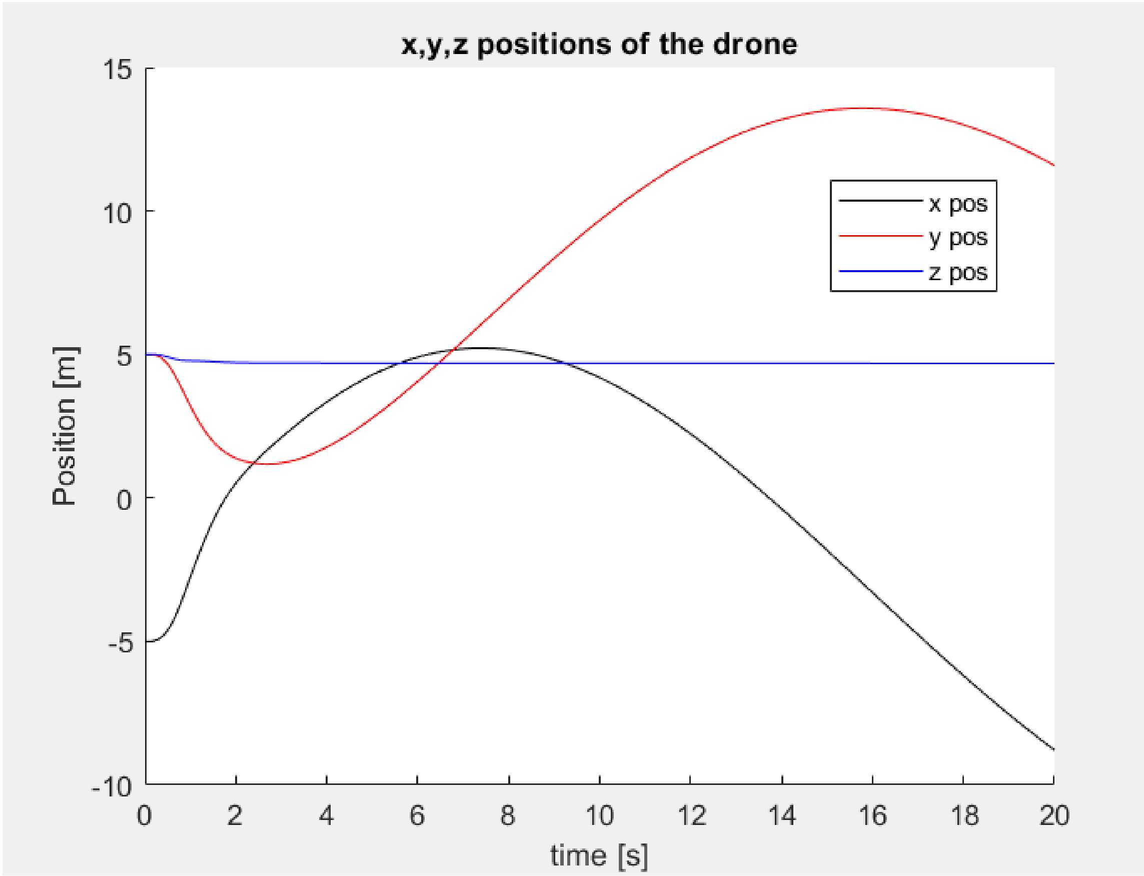

Figure 6: 3D position of drone while tracking a car driving in a spiral

The drone's height stays about constant throughout the trajectory.

6 Conclusion

The simulation demonstrates that an LQR-based control system enables the drone to successfully track a moving car while keeping it within the camera's field of view. A potential improvement that could be made is tuning the Q and R matrices in the LQR control to reduce settling time or reduce control effort. We could also try using a model predictive controller (MPC) instead of the LQR controller. Continuing this project, we could drop the assumption that the camera is noiseless and use an Extended Kalman Filter to estimate the state of the car, and possibly also localize the drone.

7 Appendix A: References

[1] S. Li, Y. Xu, C. Wang, and Z. Zhang, "Deep reinforcement learning for robotic manipulation with hybrid action space," *Procedia CIRP*, vol. 97, pp. 752-757, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214785320329047?fr=RR-2ref=pdf$_downloadrr = 8edf7804fe560f9wd$

8 Appendix B: MATLAB Simulation Code

```
clear;
clc
close all;
% files needed:
% intialize_drone_sim
% LQRcontrol
% camera_model
% car_simulate

[f_m,K] = intialize_drone_sim(0.1,100000,0.00000001); %initializes drone
    dyanmics, state, and LQR control
[drone_x, drone_y] = getDroneTarget();
x0 = [0; 0; 0; 0; 5; 0; 0; 0; 0; 0; 0; 0;]; %initial drone state
drone_pos = x0; %tracks drone position
drone_max_speed = 10; %max speed of the drone
car_EKF = EKF;
% drone_state = [x;xdot;y;ydot;z;zdot;phi;phidot;theta;thetadot;psi;
    psidot]
%car_est = [x;y;theta;v;alpha;z]
x_c0 = [0;0;0]; %initial car state
x_est = [0;0;0;0;0;0]; %initial EKF estimate
p_est = eye(6);
%car simulation parameters
x_car = x_c0;
v = 4;
L = 5;
R = 3;
kappa = 1/R;
alpha = atan(L*kappa/v);
%simulation parameters
dt = 0.05;
ti = 0;
tf = 5;
t = ti:dt:tf;
nt = length(t);
psi = 0; phi = 0; %initial pan,tilt angles
u_max = 480; v_max = 350; %pixel num
sx = 0.085; sy = 0.085*v_max/u_max; %FOV measurements
lambda = 0.035; %focal length
cam_measurements = zeros(2,nt);
for k=1:(nt-1)
    x_car_pos = x_car(:,k); %true state of the car
    xinit=drone_pos(:,k);
    % get cam and ground transformation matrices
    [H_g_to_cam,H_cam_to_g] = getCamHs(xinit,[phi,psi]);
    % get target pos in cam frame
    cam_measurement = camera_model(psi,phi,lambda,[x_car_pos(1);
        x_car_pos(2)],[drone_pos(1,k);drone_pos(3,k);drone_pos(5,k)
        -0.1],sx,sy);
    cam_measurements(:,k) = cam_measurement;
    current_car_state = [x_car_pos(1),x_car_pos(2),x_car_pos(3),v,alpha
        ,0];
    drone_target_state = findDroneTargetXY(current_car_state,
        drone_max_speed,xinit, L,dt);
```

```
    %move drone
    drone_pos(:,k+1)= LQRcontrol(f_m,K,dt,xinit,drone_target_state);
    %move drone camera
    new_cam_state = getNextCameraState(current_car_state,H_cam_to_g,[phi
        , psi],L,dt);
    phi = new_cam_state(1);
    psi = new_cam_state(2);
    %move the car to the new state
    kappa = 1/R;
    % alpha = 0; % straight line
    % alpha = pi/4 % circle
    %alpha = atan(L*kappa/v); % spiral
    x_car(:,k+1) = car_simulate(dt,x_car_pos,v,alpha,L);
    R = R + kappa*dt;
end

figure
x = drone_pos(1,:);
y = drone_pos(3,:);
z = drone_pos(5,:);
hold on
plot(t,x,'k')
plot(t,y,'r')
plot(t,z,'b')
title("x,y,z positions of the drone")
xlabel("time [s]")
ylabel("Position [m]")
legend("x pos","y pos","z pos")
hold off

figure
hold on
plot(x_car(1,:),x_car(2,:),'k')
title("car path in x -y plane")
hold off

figure
hold on
axis equal
plot3(x,y,z,'r')
plot3(x_car(1,:),x_car(2,:),zeros(1,nt),'b')
view(3)
title("UAV tracking of the car")
xlabel("x [m]")
ylabel("y [m]")
zlabel("z [m]")
legend("UAV position","Car Position")
hold off
writematrix([t;drone_pos;x_car;],'UAV_track_3.csv');

figure
hold on
axis equal
plot(x,y,'r')
```

```
plot(x_car(1,:),x_car(2,:),'b')

figure
hold on
xlim([-sx sx]);
ylim([-sy sy]);
plot(cam_measurements(1,:),cam_measurements(2,:),'r*')
hold off

function target_drone_state = findDroneTargetXY(car_state,drone_max_v, drone_state, L,dt)
    max_lookahead_time = 10; %total time to look foward in seconds
    this_time = 0;
    best_time_dif = 10^10;
    this_car_state = car_state;
    best_car_state = car_state;
    while this_time <= max_lookahead_time && best_time_dif > dt*2
        this_time_dif = abs(this_time-sqrt((this_car_state(1)-drone_state(1))^2 + (this_car_state(2)-drone_state(3))^2)/drone_max_v);
        if this_time_dif < best_time_dif
            best_time_dif = this_time_dif;
            best_car_state = this_car_state;
        end
        this_car_state = predictCarState(this_car_state,L, dt);
        this_time = this_time + dt;
    end
    target_drone_state = [best_car_state(1);0;best_car_state(2);0;drone_state(5);0;0;0;0;0;0;0];
end

function new_cam_state = getNextCameraState(current_car_state,H_cam_to_g,cam_state,L,dt)
    next_car_state = predictCarState(current_car_state, L, dt);
    pos_target = [next_car_state(1);next_car_state(2);next_car_state(6)];
    P = [pos_target; 1];
    target_pos_cam_frame = H_cam_to_g * P;
    % Compute tilt angle about the X-axis
    alpha = atan2(-target_pos_cam_frame(2), target_pos_cam_frame(3)); % Angle in radians
    % Compute pan angle about the Y-axis
    theta = atan2(target_pos_cam_frame(1), target_pos_cam_frame(3));
    new_cam_state = [alpha + cam_state(1), theta + cam_state(2)];
end

function next_car_state = predictCarState(car_state,L, dt)
    % Function to compute the next state of the system
    x = car_state(1);
    y = car_state(2);
    theta = car_state(3);
    v = car_state(4);
    alpha = car_state(5);
    z = car_state(6);
```

```

    % Update equations
    x_next = x + v * cos(theta) * dt;
    y_next = y + v * sin(theta) * dt;
    theta_next = theta + (v / L) * tan(alpha) * dt;

    %assuming v, z and alpha dont change
    alpha_next = alpha;
    v_next = v;
    z_next = z;
    next_car_state = [x_next,y_next,theta_next,v_next,alpha_next,z_next
        ];
end

function [H_g_to_cam,H_cam_to_g] = getCamHs(drone_state, camera_state)
% drone_state = [x;xdot;y;ydot;z;zdot;phi;phidot;theta;thetadot;psi;
    psidot]
    R_cam_x=basic_rotation_matrix(camera_state(1),'x');
    R_cam_yPrime=basic_rotation_matrix(camera_state(2),'y');
    R_drone_x = basic_rotation_matrix(drone_state(7),'x');
    R_drone_yPrime = basic_rotation_matrix(drone_state(9),'y');
    R_drone_zPrime2 = basic_rotation_matrix(drone_state(11),'z');
    R_g_to_drone = R_drone_x*R_drone_yPrime*R_drone_zPrime2;
    R_g_to_cam = R_g_to_drone*R_cam_x*R_cam_yPrime;

    P_g_to_drone = [drone_state(1);drone_state(3);drone_state(5)];
    P_drone_to_cam = [0;0;-0.1];
    P_g_to_cam = P_g_to_drone + R_g_to_drone*P_drone_to_cam;
    H_g_to_cam = [R_g_to_cam,P_g_to_cam;0,0,-0,1];
    H_cam_to_g = [R_g_to_cam', -R_g_to_cam' * P_g_to_cam; 0, 0, 0, 1];

end

function R = basic_rotation_matrix(angle, axis)
    % Simple rotation matrix about axis (string 'x' 'y' or 'z')
    c = cos(angle);
    s = sin(angle);
    if strcmp(axis, 'x')
        R = [1 0 0; 0 c -s; 0 s c];
    elseif strcmp(axis, 'z')
        R = [c -s 0; s c 0; 0 0 1];
    elseif strcmp(axis, 'y')
        R = [c 0 s; 0 1 0; -s 0 c];
    else
        error('Invalid rotation axis.');
    end
end

function [f_m,K] = intialize_drone_sim(dt,a,rho)
%drone parameters
m= 1;
g = 9.81;
Ix = 0.11;
Iy = 0.11;
```

```
Iz = 0.04;
Ki = 3e-6;
Kd = 4e-9;
l = 0.2;

%define symbolic system perameters and control inputs
syms x xdot y ydot z zdot phi phidot th thdot psii psiidot u1 u2 u3 u4

%drone model
F1 = Ki*u1;
F2 = Ki*u2;
F3 = Ki*u3;
F4 = Ki*u4;

M1 = Kd*u1;
M2 = Kd*u2;
M3 = Kd*u3;
M4 = Kd*u4;

xddot = (1/m)*(F1 + F2 + F3 + F4)*(cos(phi)*sin(th)*cos(psii) + sin(phi)*sin(psii));
yddot = (1/m)*(F1 + F2 + F3 + F4)*(cos(phi)*sin(th)*sin(psii) + sin(phi)*cos(psii));
zddot = (1/m)*((F1 + F2 + F3 + F4)*(cos(phi)*cos(th))-m*g);
phiddot = (1/Ix)*((F1-F3)*l + thdot*psiidot*(Iy-Iz));
thddot = (1/Iy)*((F2-F4)*l + psiidot*phidot*(Iz-Ix));
psiddot = (1/Iz)*((M2 + M4 - M1 - M3) + phidot*thdot*(Ix - Iy));


%drone dynamic model
f = [xdot; xddot; ydot; yddot; zdot; zddot; phidot; phiddot; thdot; thddot; psiidot; psiddot;];
f_m = matlabFunction(f,"Vars",[x xdot y ydot z zdot phi phidot th thdot psii psiidot u1 u2 u3 u4]);

A_c = jacobian(f,[x;xdot;y;ydot;z;zdot;phi;phidot;th;thdot;psii;psiidot]);
B_c = jacobian(f,[u1;u2;u3;u4]);
A_c = matlabFunction(A_c);
B_c = matlabFunction(B_c);

control = m*g/(4*Ki);
A = A_c(0,0,0,0,0,0,control,control,control,control);
B = B_c(0,0,0);
C = eye(12);
sys = ss(A,B,C,0);
sys_d = c2d(sys,dt);
Q = a*(C'*C);
R = rho*eye(4);
[K,~,~] = dlqr(sys_d.A,sys_d.B,Q,R);
end

function drone_state = LQRcontrol(f_m,K,dt,xinit,xtarget)
    Ki = 3e-6;
```

```
    m= 1;
    g = 9.81;
    control = m*g/(4*Ki);
    u = -K*(xinit - xtarget) + control*[1;1;1;1];
    tspan=[0 dt];
    [~,x]=ode45(@(t,x) odefunc(t,x,u,f_m),tspan,xinit);
    drone_state=x(end,:)';
end

function dxdt = odefunc(~,x,u,f_m)
dxdt = f_m(x(1),x(2),x(3),x(4),x(5),x(6),x(7),x(8),x(9),x(10),x(11),x
    (12),u(1),u(2),u(3),u(4));
end
function sensor_meas=camera_model(psi,phi,lambda,x_t,x_c,sx,sy)
%sensor model for the camera
%psi and phi are pan and tilt angles
%lambda is focal length of the parameter
%x_t is location of robot in inertial frame (must be a 2d vector)
%x_c is coordinates of the camera in the inertial frame (3d vector)
%returns sensor_meas which is a 2d vector (xp, yp)

R_phi = [1      0            0;
         0  cos(phi)  sin(phi);
         0 -sin(phi)  cos(phi)];
R_psi = [cos(psi) sin(psi) 0;
         -sin(psi) cos(psi) 0;
         0          0        1];
q_t = R_phi*R_psi*([x_t' 0]' - x_c);
p_t = lambda*[q_t(1)/q_t(3) q_t(2)/q_t(3)]';
if abs(p_t(1)) <= sx && abs(p_t(2)) <= sy
    sensor_meas = -p_t;
else
    sensor_meas = NaN;
end
end

function car_pos = car_simulate(dt,car_state,v,alpha,L)
    x = car_state(1) + v*cos(car_state(3))*dt;
    y = car_state(2) + v*sin(car_state(3))*dt;
    theta = car_state(3) + v/L * tan(alpha)*dt;
    car_pos = [x;y;theta];
end
```