COMP0128 Robotic Control Theory and Systems Quadcopter Simulation

Ela Kanani, Joseph Rowell, Yuansheng Zhang

08/11/2021

Question 1

Implementing a simulation of a quadcopter, following a non-linear model, The inputs are the squared angular velocities $(\gamma_1, \gamma_2, \gamma_3, \gamma_4)$, for each propellor. The other fixed parameters input were: Mass of quadcopter $m = 0.2[kg]$,

Quadcopter Rotational Inertia Matrix $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$,

Gravitational Acceleration Constant $g = 9.2[m/s^2]$,

Friction Constant $kd = 0.1$ (assumed equal for x,y,z),

Length of propeller $L = 0.2[m]$,

Propeller Constants $k = 1$, $b = 0.1$

In the Matlab file 'Drone.m', the quadcopter starts at 5m altitude, between the times of 0-2s, the inputs to the drone squared angular velocities were calculated by inputting the desired z axis acceleration (zero when hovering at 5m) in the function 'solveInputs.m', which then outputs the ideal thrust, given forces due to gravity and drag. The rotation matrix to convert between the body and position inertial frames is also calculated via the function 'Rotation.m' with inputs theta. As shown below:

$$R = \begin{bmatrix} cos(\psi)cos(\theta), & cos(\psi)sin(\phi)sin(\theta) - cos(\phi)sin(\psi), & sin(\phi)sin(\psi) + cos(\phi)cos(\psi)sin(\theta) \\ cos(\theta)sin(\psi), & cos(\phi)cos(\psi) + sin(\phi)sin(\psi)sin(\theta), & cos(\phi)sin(\psi)sin(\theta) - cos(\theta)sin(\phi) \\ -sin(\theta), & cos()sin(\phi), & cos(\phi)cos(\theta) \end{bmatrix}$$

```
%% Rotation
        function R = rotation(obj)
            % Inputs: phi(roll), theta (pitch), psi (yaw)
            % Outputs: Rotation Matrix R
            phi = obj.th(1);
            theta = obj.th(2);
            psi = obj.th(3);

            Rx = [1 0 0;
                0 cos(phi) -sin(phi); ...
                0 sin(phi) cos(phi)];
            Ry = [ cos(theta) 0 sin(theta);
                0 1 0; ...
                -sin(theta) 0 cos(theta)];
            Rz = [cos(psi) -sin(psi) 0; ...
                sin(psi) cos(psi) 0; ...
                0 0 1];
            R = Rz*Ry*Rx;

        end
```

Listing 1: Question 1 Rotation matrix calculation

Between the times of 2-4s, the values of the inputs are increased 1.2x, so the quadcopter rises, and from 4-8 s, the $\gamma_4 = 0$. The balance in the torque created maintained by the coupled propellers breaks and thus the drone rotates and falls to the ground. This is shown in Listing 2 function update(obj).

```
function update(obj)
            %update simulation time
            obj.time = obj.time + obj.time_interval;

            % Update inputs as required in question 1
            if obj.time<2
                obj.inputs = obj.equ_inputs;
            elseif obj.time>=2 && obj.time<4
                obj.inputs = 1.2 .* obj.equ_inputs;
            elseif obj.time>=4 && obj.time<=8
                obj.inputs(4) = 0;
            end

            %change position and orientation of drone
            simulation_dynamics(obj);

            %draw drone on figure
            draw(obj);
        end
```

Listing 2: Question 1 function update(obj)

```
  function inputs = solveInputs(a_target, obj)
            %Inputs; a_target: The target z acceleration of the drone
            %obj: The global variables specified at beginning of file

            %Outputs; Inputs: the gamma_1 to gamma_4 squared angular
            %velocity of each propeller
            gravity=[0; 0; -obj.g];
            Fd = -obj.kd * obj.pos_dot;
            obj.R = rotation(obj);
            T = Fd+ obj.m * (a_target-gravity) / (cos(obj.th(2))*cos(obj.th(1)));
            inputs = zeros(4, 1);
            inputs(:) = T(3)/4;
        end
```

Listing 3: Question 1 SolveInputs in file Drone.m

The acceleration function shown below in Listing 4 calculates the acceleration of the drone given the forces acting upon it, for example force due to drag, and force due to gravity.

```
function a = acceleration(obj)
            % Inputs; obj: Global varibales at beginning of file
                    % g: acceleration due to gravity (-9.2m/s/s)
                    % kd: Drag coefficient
                    % pos_dot: velocity of the drone
            %Outputs; a: Acceleration of the drone
            gravity=[0; 0; -obj.g];
            obj.R = rotation(obj);
            T = obj.R * thrust(obj.inputs, obj);
            Fd = -obj.kd * obj.pos_dot;
            a = gravity + 1 / obj.m * T + Fd;
        end
```

Listing 4: Question 1 Acceleration function in file Drone.m

The thrust function computes the thrust vector for the quadcopter, given the squared angular velocities of each propeller, as shown in Listing below.

```
function T = thrust(inputs, obj)
            %Inputs are for Wi^2= gamma_1 to 4
            %Outputs T Thrust Vector
            T= [0; 0; obj.k * sum(inputs)];
        end
```

Listing 5: Question 1 Thrust function in file Drone.m

The torques function computes the τ torque vector given propeller inputs and propeller properties, as shown in the Listing below.

```
function tau = torques(inputs, obj)
            %Inputs are values for wi^2
            % obj global variables
            % Outputs tau Torque Vector
            tau = [
                obj.L * obj.k * (inputs(1) - inputs(3))
                obj.L * obj.k * (inputs(2) - inputs(4))
                obj.b * (inputs(1) - inputs(2) + inputs(3) - inputs(4))];
        end
```

Listing 6: Question 1 Torques function in file Drone.m

The function thetadot2omega converts the first derivative of roll, pitch, yaw to angular velocity vector ω. While the omega2thetadot function converts these values vice versa. The angular_acceleration function computes the derivative of angular velocity ω to output angular acceleration vector $\dot{\omega}$.

```
 function omega = thetadot2omega(obj)
            % Inputs: thetadot, first derivative of roll, pitch, yaw
            % Outputs omega
            omega = [1, 0, -sin(obj.th(2));
                0, cos(obj.th(1)), cos(obj.th(2)) * sin(obj.th(1));
                0, -sin(obj.th(1)), cos(obj.th(2)) * cos(obj.th(1))] * obj.thd;
    end

function omegadot = angular_acceleration(obj)
            % Inputs: Drone squared angular velocity inputs
            % I quadcopter rotational inertia matrix
            % Outputs: omegadot Angular acceleration vector
            tau = torques(obj.inputs, obj);
            omegadot = (obj.I) \ (tau - cross(obj.omega, obj.I * obj.omega));
    end


function thd = omega2thetadot(obj)
            %Inputs: phi, theta, psi
            %Outputs thd
            thd = [1, 0, -sin(obj.th(2));
                0, cos(obj.th(1)), cos(obj.th(2)) * sin(obj.th(1));
                0, -sin(obj.th(1)), cos(obj.th(2)) * cos(obj.th(1))] \ obj.omega;
    end
```

Listing 7: Question 1 angular dynamics functions in file Drone.m

Figure 1 illustrates the drone's trajectory and orientation over time in 2D, whilst Figure 2 illustrates the trajectory over time in 3D.

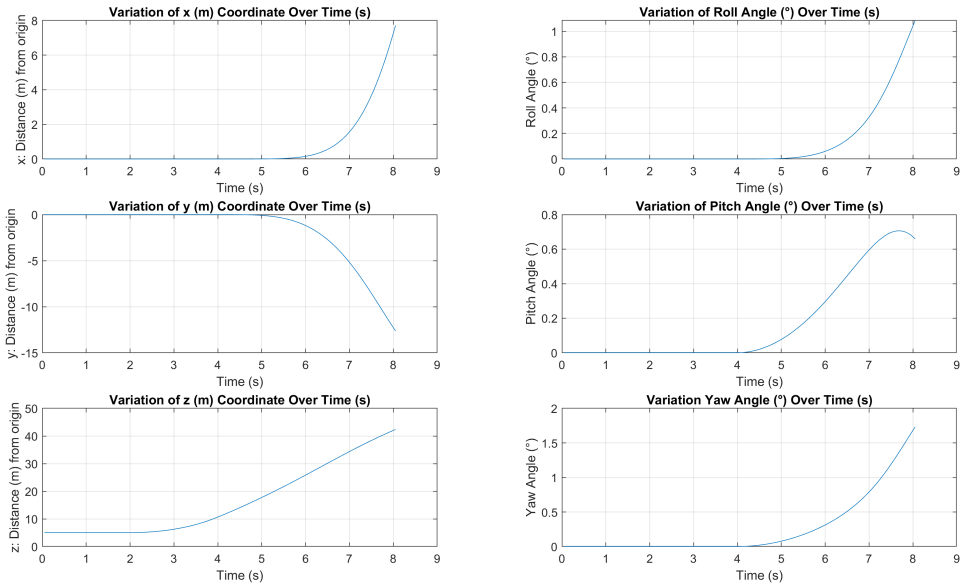

Figure 1: Question 1, 2D Plots of the simulation, displaying position and orientation of the quadcopter over time.

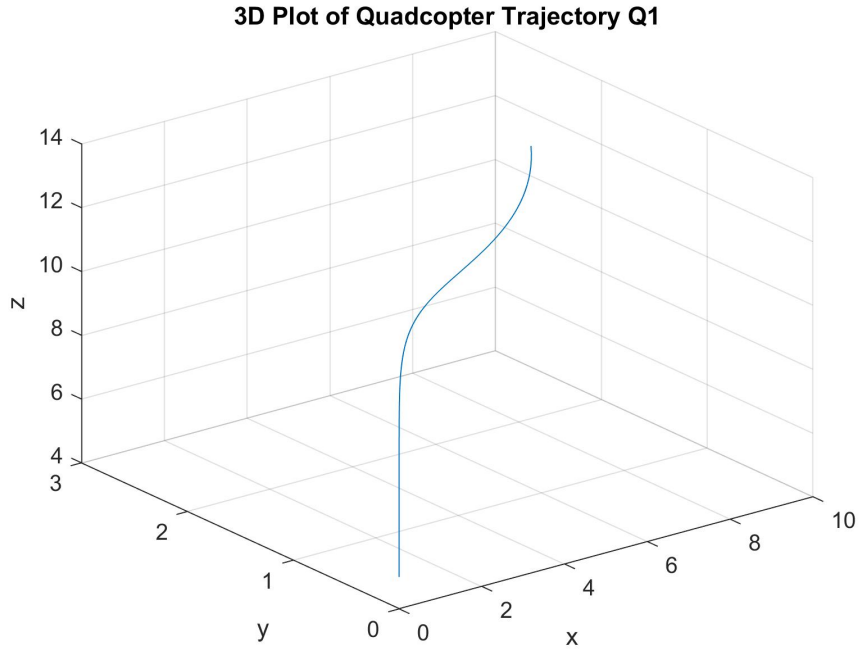

Figure 2: Question 1, 3D Plot of the simulation, displaying position and orientation of the quadcopter over time.

Question 2

2a

The non-linear dynamics of the quadcopter in state-space representation are described in [1], in pages 7 and 8. Derive a linearised model approximation with the shape $\dot{x} = \boldsymbol{Ax + Bu}$ *any chosen equilibrium point (justify your choice). Then discretise the linear system. You can use Matlab to help with all your calculations in this question (see: syms, jacobian, ss, c2d).[10 marks]*

Linearised model approximation in the form $\dot{x} = \mathbf{Ax} + \mathbf{Bu}$ was achieved in the file 'q2a.m'. In this script, the symbolic variables of position (x, y, z), velocity $(\dot{x}, \dot{y}, \dot{z})$, angular velocity roll, pitch and yaw (ϕ, θ, ψ) respectively, and angular acceleration $(\dot{\phi}, \dot{\theta}, \dot{\psi})$; were concatenated into a $[12, 1]$ array. The dynamics of the system were calculated by calculating the force due to drag, thrust and torques; in both the body frame and the absolute frame or reference, as shown below.

Rotation Dynamics

$$R = \begin{bmatrix} cos(\phi) * cos(\psi) - cos(\theta) * sin(\phi) * sin(\psi), & -cos(\psi) * sin(\phi) - cos(\phi) * cos(\theta) * sin(\psi), & sin(\theta) * sin(\psi) \\ cos(\theta) * cos(\phi) * sin(\phi) + cos(\phi) * sin(\psi), & cos(\phi) * cos(\theta) * cos(\psi) - sin(\phi) * sin(\psi), & -cos(\psi) * sin(\theta) \\ sin(\phi) * sin(\theta), & cos(\phi) * sin(\theta), & cos(\theta) \end{bmatrix}$$

Force due to drag $F_d = -k_d(\ddot{x}, \ddot{y}, \ddot{z})$
Thrust $T_b = (a_{target} - gravity)m - F_d$
Torques

$$\tau = \begin{bmatrix} L * k * (\gamma_1 - \gamma_3) \\ L * k * (\gamma_2 - \gamma_4) \\ b * ((\gamma_1 - \gamma_2 + \gamma_3 - \gamma_4)) \end{bmatrix}$$

The jacobian of the matrix was then calculated:
$Aj = jacobian(xd, x)$
$Bj = jacobian(xd, u)$
The system can be linearised around the equilibrium point, the equilibrium point was determined by the state being unchanging, in that the acceleration, and angular acceleration are equal to zero. Thus, the net force on the system is zero For simplicity, the position and velocity are equal to zero also.

$$\dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z}, \phi, \theta, \psi = 0$$

```
function [Ad, Bd] = disc_linearisation(obj)
    % symbolic variables used to compute analytical functions
    %{
    x 1-3: pos_x pos_y pos_z
    x 4-6: posd_x posd_y posd_z
    x 7-9: phi theta psi
    x 10-12: omg1 omg2 omg3

    xd 1-3: posd_x posd_y posd_z
    xd 4-6: podd_x podd_y podd_z
    xd 7-9: thd1 thd2 thd3
    xd 10-12: omgd1 omgd2 omgd3
    %}

    syms x [12,1]
    syms xd [12,1]
    syms u [4,1]

    %% define parameters
    gravity=[0; 0; -obj.g];

    %% compute dynamics
    phi = x(7);
    theta = x(8);
    psi = x(9);

```

```
    Rx = [1 0 0;
        0 cos(phi) -sin(phi); ...
        0 sin(phi) cos(phi)];
    Ry = [ cos(theta) 0 sin(theta);
        0 1 0; ...
        -sin(theta) 0 cos(theta)];
    Rz = [cos(psi) -sin(psi) 0; ...
        sin(psi) cos(psi) 0; ...
        0 0 1];
    R = Rz*Ry*Rx;

    % Drag force
    Fd = -obj.kd * [x(4); x(5); x(6)] ;

    tau = [
        obj.L * obj.k * (u(1) - u(3))
        obj.L * obj.k * (u(2) - u(4))
        obj.b * (u(1) - u(2) + u(3) - u(4))];

    Tb = [0; 0; obj.k * sum(u)];

    %% state space
    xd(1:3) = x(4:6);
    xd(4:6) = gravity +(1/obj.m) * R *(Tb) + (1/obj.m) *Fd;
    xd(7:9) = [1, 0, -sin(x(8));
        0, cos(x(7)), cos(x(8)) * sin(x(7));
        0, -sin(x(7)), cos(x(8)) * cos(x(7))] \ x(10:12);
    xd(10:12) = (obj.I) \ (tau - cross(x(10:12), obj.I * x(10:12)));

    %Compute Jacobians
    Aj = jacobian(xd,x);
    Bj = jacobian(xd,u);

    A =  subs(Aj,...
        [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12],...
        obj.state.');
    A = double (subs(A, u, obj.inputs));
    B = double (subs(Bj,...
        [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12],...
        obj.state.'));

    % Define C and D matrices
    C = eye(size(A));
    D = zeros(size(B));

    cont_sys = ss(A,B,C,D);
    disc_sys = c2d(cont_sys,obj.time_interval,'zoh');

    Ad = disc_sys.A;
    Bd = disc_sys.B;

end
```

Listing 8: Question 2a Linearised Model approximation function

2b

Implement a second simulation of the quadcopter, but now using the linear approximation derived in 2a. To demonstrate its behaviour, use the same experiment from question 1. Submit the working code [8 marks]. For the written answer, explain how the code works [4 marks] and compare the quadcopter position and orientation computed with the linear approximation versus the original non-linear model, by plotting both trajectories side-by-side [4 marks]. Comment on the results [4 marks].

In a linear time invariant (LTI) system, with a jacobian, it is always assumed there is an equilibrium point. It was chosen so that the net force on the system = 0, in reality, there will be an acceleration in order to change position from a stop, thus there is an error in the discretised LTI approximation. The error in the discretised (LTI) system is shown as the difference between the linear and the discretised (LTI) system in Figure 3 and Figure 4.

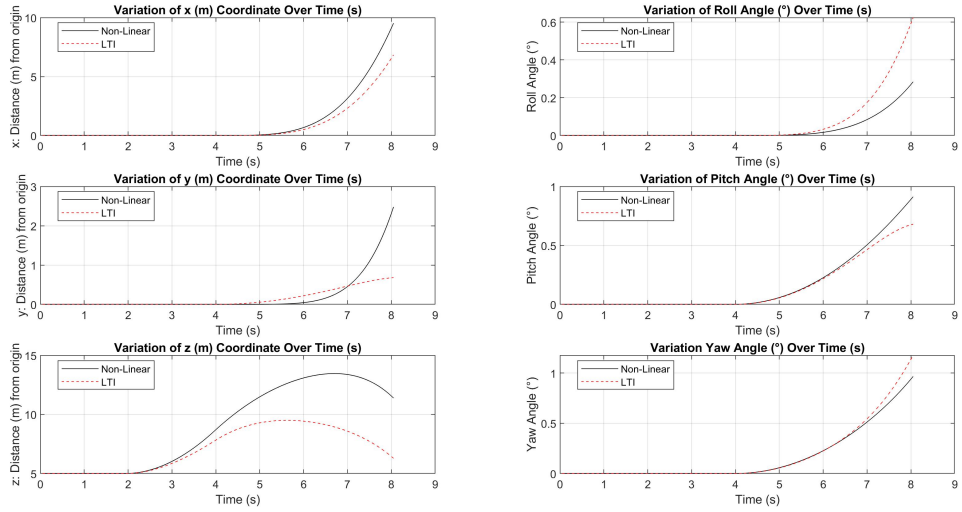

Figure 3: Question 2b, 2D Plots of the simulation, displaying position and orientation of the quadcopter over time.

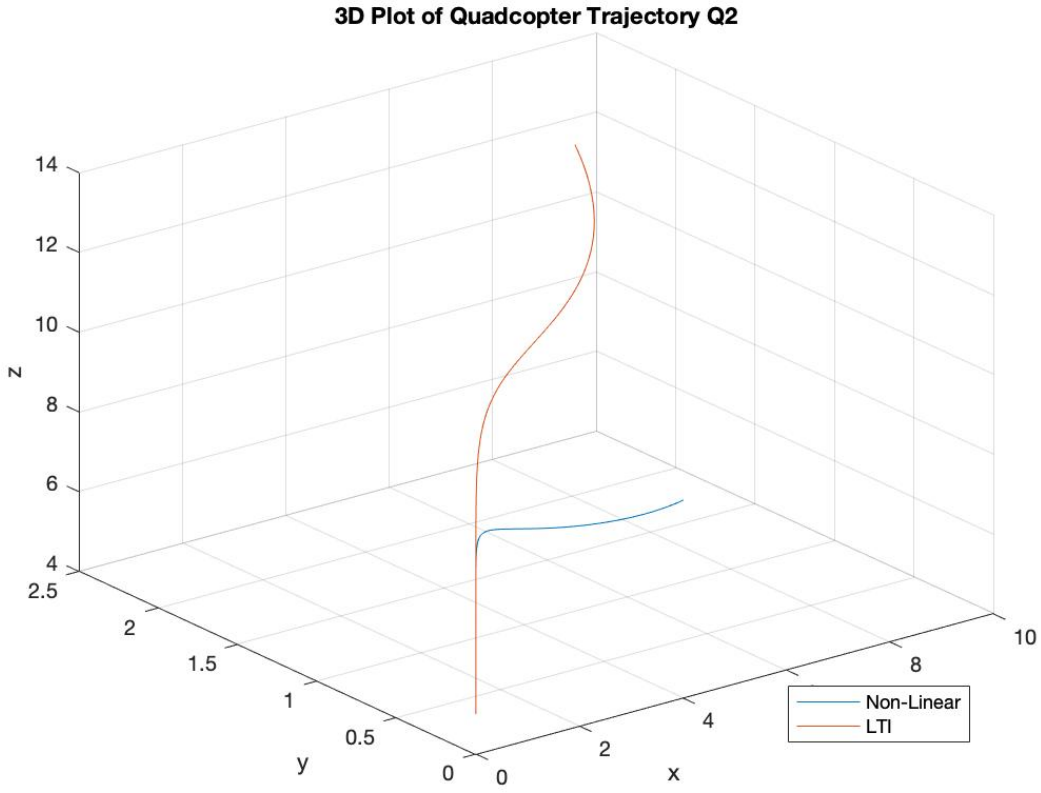

Figure 4: Question 2b, 3D Plots of the simulation, displaying position and orientation of the quadcopter over time.

Below is shown the update(obj) file for implementing a simulation of the quadcopter, using an the LTI approximation derived in (2a). In this update function, the inputs are the same as in Question 1, at first between 0-2s, the drone will hover at constant position, as the inputs to each propeller are

equal. Then, between 2-4s, the drone will rise as inputs are increased x1.2. Finally, the drone will psin and fall as one propeller input = 0. The disc_linearisation function is called, this computes the jacobian of the system and the state is updated with delta_x.

```
function update(obj)
            %update simulation time
            obj.time = obj.time + obj.time_interval;

            %update inputs as required
            % Drone hovers with equal inputs matching gravity and drag
            if obj.time<2
                obj.inputs = obj.equ_inputs;
                % Drone rises as inputs are increased
            elseif obj.time>=2 && obj.time<4
                obj.inputs = 1.2 .* obj.equ_inputs;
                % Drone spins and falls as gamma_4 =0
            elseif obj.time>=4 && obj.time<=8
                obj.inputs(4) = 0;

            end
            %  x [k + 1] = Ad  x [k] + Bd  u [k],
            % [obj.pos; obj.pos_dot; obj.th; obj.omega]

            [obj.Ad, obj.Bd] = disc_linearisation(obj);
            obj.delta_x = obj.Ad * (obj.delta_x)...
                +obj.Bd * (obj.inputs- obj.pre_inputs);
            obj.state = obj.state + obj.delta_x;

            % update variables
            obj.pre_inputs = obj.inputs;
            obj.pos = obj.state (1:3);
            obj.pos_dot = obj.state (4:6);
            obj.th = obj.state (7:9);
            obj.omega = obj.state (10:12);
            obj.R = eul2rotm(obj.th');

            % keep track of current and previous position, orientation and
            % time
            obj.xyzpos = [obj.xyzpos,obj.pos];
            obj.orientation = [obj.orientation, obj.th];

            %draw drone on figure
            draw(obj);
        end
```

Listing 9: Question 2b Implementing linear approximation in simulation

Question 3

3a

Assume the entire state of the quadcopter can be measured by "perfect" sensors with 100% accuracy. Using the quadcopter simulator from question 1, implement a feedback controller that makes the drone perform the following trajectory:

- *Starts at (0,0,0)*
- *Moves up to (5,5,5)*
- *Stays at (5,5,5) for 10 seconds.*
- *Passes through (5,-5,10), (-5,-5,6), (-5,5,2), (0,0,2).*
- *Lands at (0,0,0) safely (less than 0.1 m/s linear velocity when hitting the ground).*

Submit the working code [13 marks]. For the written answer, explain how the code works [6 marks] and provide adequate plots of the simulated quadcopter trajectory in position and orientation [6 marks].

To determine whether a full state feedback controller can be utilised in the discrete LTI problem, the reachability matrix needs to be a full rank matrix.
In the form: $x[T] - A^T x[0] = W_r[u[T-1], ..., u[1], u[0]]^T$
Given any combination of values for x[0] and x[T], we can choose u[0], u[1],... u[k-1] to satisfy the above equation for a finite T, as W_r was determined to be a full rank matrix in the code shown below.

```
%Run after quadcopter_script to have the drone variable
% Find A and B matrices
[Ad, Bd] = disc_linearisation(drones);

%Step 1: Calculate reachability matrix
Wr = reachability_matrix(Ad,Bd);
Co = ctrb(Ad,Bd);
%Step 2: Calculate rank of reachability matrix
Wr_rank = rank(Wr,1e-16);
%Step 3: compare rank to matrix dimensions to see if full rank
if Wr_rank == min(size(Wr))
    reachability = true; %if full rank, the solution is reachable
    disp('Full rank reachability matrix -- the solution is reachable.')
else
    disp('Reachability matrix is not full rank -- the solution is unreachable.')
end


%% Functions

%Calculate the reachability matrix (Wr) using A and B from the discrete
%linearisation (LTI) form, where Wr = [B AB ... A^(T-1)B]

function Wr = reachability_matrix(A,B)
    %Inputs A and B jacobian matrices
    % Outputs Wr Reachability matrix
    Wr =[]; %initialise matrix
    Wr(:,1:4) = B; %add B matrix first
    for T = 1:size(A,1)-1
        AB = A^(T)*B;
        Wr = [Wr, AB];
    end
end
```

Listing 10: Question 3a Full Rank Reachability Matrix Matlab Code

Therefore a full state feedback (FSF) controller was used to carry out the instructions for 3a. A FSF controller is a linear combination of the errors with all of the state variables, compared with a specified reference, r_x, i.e. $u[k] = -K(x[k] - r_x)$. This is where the complete system is as follows, including the Jacobi (A and B):

$$x[k+1] = (A - BK)x[k] + BKr_x$$

K considers the inputs of the poles (eigenvalues) selected such that $A - BK$ is equal to system eigenvalues. We can arbitrarily select these eigenvalues, as long as they are less than 1, do not repeat, and stabilise the system. The following eigenvalues were selected, with a trade off between system response and stability, and ensuring that K was not made up of high gains:

$$obj.eigenvalues = [0.9, 0.9, 0.982, 0.995, 0.987, 0.807, 0.987, 0.992, 0.971, 0.877, 0.791, 0.825]$$

First, the reference points for the simulation, and the eigenvalues were assigned to the objects. The time inteval in the quadcopter script, dt, was assigned to 0.2; a lower time interval allows for a better controller as a trade off with computational power. The following function was used to implement the full-state feedback controller:

```
        function fsfInputs(ref_pos, obj)
            %Calculates inputs, u, using full-state feedback when given the
            %reference position and the object
            %calculate K matrix
            obj.K = place(obj.Ad,obj.Bd,obj.eigenvalues);
            %find error (i.e. difference between x and reference position)
```

```
            e = obj.state-ref_pos;
            %adjust input relative to the equilibrium inputs
            obj.inputs = obj.equ_inputs- obj.K*e;
            obj.inputs = obj.inputs(:);
        end
```

Listing 11: Question 3a Full State Feedback function

Here, the controller is fed the reference position, and the object. First, it calculates the K matrix (via the *place*() function) using the discretised A and B matrices found using the *disc_linearisation*(*obj*) function, considering the equilibrium inputs, and the eigenvalues. Then the error is found, and the inputs are updated by comparing the equilibrium inputs and the K matrix amalgamated with the error.

This function was implemented within the *update*() function, using the same dynamics simulation discussed in Question One, as follows:

```
        function update(obj)
            tol = 0.1; %tolerance for controller
            %update simulation time
            obj.time = obj.time + obj.time_interval;
            %update inputs as required
            %if checkpoint one isn't reached
            if obj.checkpoints(1) == 0 %while the target isn't reached
                ref_pos = obj.q3_ref(:,1); %target = point 5,5,5
                disp ('aiming for [5,5,5]')
                fsfInputs(ref_pos, obj);
                simulation_dynamics(obj);
                if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                    obj.checkpoints(1) = 1;
                    obj.time_stamp = obj.time;
                end

            elseif obj.checkpoints(1) == 1 ...
                    && obj.checkpoints(2) == 0 %when previous target is reached
                if obj.time < obj.time_stamp+10
                    disp ('Maintaining position of the first target');
                    ref_pos = obj.q3_ref(:,1); %target = point 5,5,5
                    fsfInputs(ref_pos, obj);
                    simulation_dynamics(obj);
                else
                    ref_pos = obj.q3_ref(:,2); %target = point 5,-5,10
                    disp ('aiming for [5,-5,10]')
                    fsfInputs(ref_pos, obj);
                    simulation_dynamics(obj);
                    if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                        obj.checkpoints(2) = 1;
                    end
                end

            elseif obj.checkpoints(2) == 1 ...
                && obj.checkpoints(3) == 0 %when previous target is reached
                ref_pos = obj.q3_ref(:,3); %target = point -5,-5,6
                disp ('aiming for [-5,-5,6]')
                fsfInputs(ref_pos, obj);
                simulation_dynamics(obj);
                if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                    obj.checkpoints(3) = 1;
                end

            elseif obj.checkpoints(3) == 1 ...
                    && obj.checkpoints(4) == 0 %when previous target is reached
                ref_pos = obj.q3_ref(:,4); %target = point -5,5,2
                disp ('aiming for [-5,5,2]')
                fsfInputs(ref_pos, obj);
                simulation_dynamics(obj);
                if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                    obj.checkpoints(4) = 1;
                end

            elseif obj.checkpoints(4) == 1 ...
                    && obj.checkpoints(5) == 0 %when previous target is reached
```

```
                ref_pos = obj.q3_ref(:,5); %target = point 0,0,2
                disp ('aiming for [0,0,2]')
                fsfInputs(ref_pos, obj);
                simulation_dynamics(obj);
                if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                    obj.checkpoints(5) = 1;
                end

            elseif obj.checkpoints(5) == 1 ...
                    && obj.checkpoints(6) == 0 %when previous target is reached
                ref_pos = obj.q3_ref(:,6); %target = point 0,0,0
                disp ('landing...')
                fsfInputs(ref_pos, obj);
                simulation_dynamics(obj);

                %{
                % velocity limitor for landing
                v_linear = sqrt( sum(obj.pos_dot.^2));
                if v_linear > 0.1
                    obj.pos_dot = 0.1 * obj.pos_dot / v_linear
                end
                %}

                if sqrt(sum((ref_pos(1:3)-obj.pos).^2)) < tol
                    obj.checkpoints(6) = 1;
                    obj.inputs = zeros(4,1);
                    disp ('Mission Complete');
                    obj.time_stamp = obj.time;
                end

            elseif obj.checkpoints(6) == 1 && obj.time > obj.time_stamp + 3
                simulation_dynamics(obj);
                obj.mission_complete = true;
            end

            % altitude limitor
            if obj.pos(3)<0
                obj.pos(3) = 0;
            end

            disp(obj.pos)

            % update state
            obj.state = [obj.pos;obj.pos_dot;obj.th;obj.omega];

            % draw drone to figure
            draw(obj);

            % keep track of current and previous position, orientation and
            % time
            obj.xyzpos = [obj.xyzpos,obj.pos];
            obj.orientation = [obj.orientation, obj.th];
            obj.times = [obj.times, obj.time];
        end
```

Listing 12: Question 3a Updating the simulation with the Full State Feedback function

Within this, the code uses 'checkpoints' (a boolean array) to assess whether each desired point has been reached and whether the target coordinate should change. A tolerance of ± 0.1 has been added to each coordinate, such that the controller accepts this as the target and does not spend too much time trying to exactly match the coordinates.

The trajectory plots of the quadcopter in Question 3a using a state space controller are depicted below, showing position and orientation over time. It may be noted that the quadcopter does not perfectly match the targets, which is due to the tolerance value added.

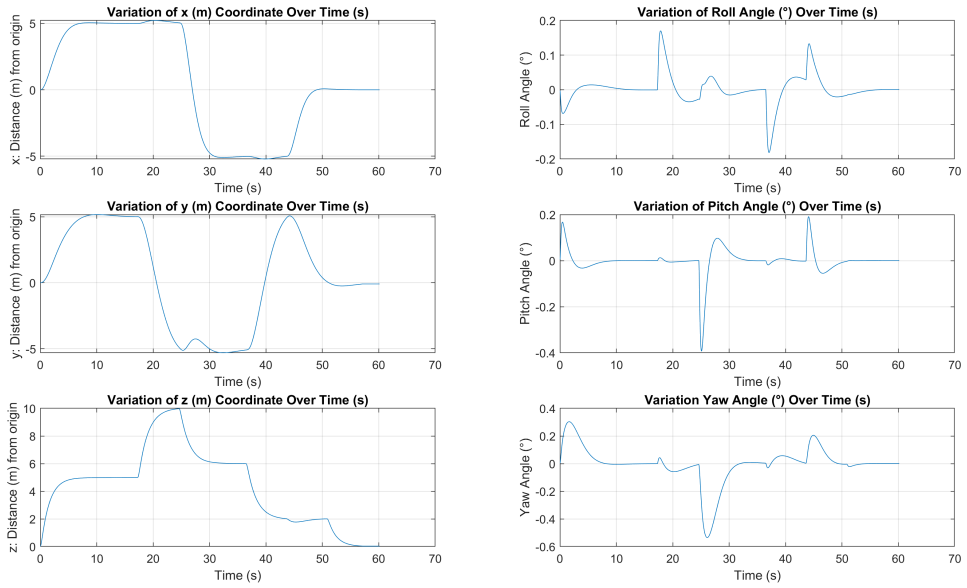

Figure 5: Question 3a, 2D Plots of the simulation, displaying position and orientation of the quadcopter over time.

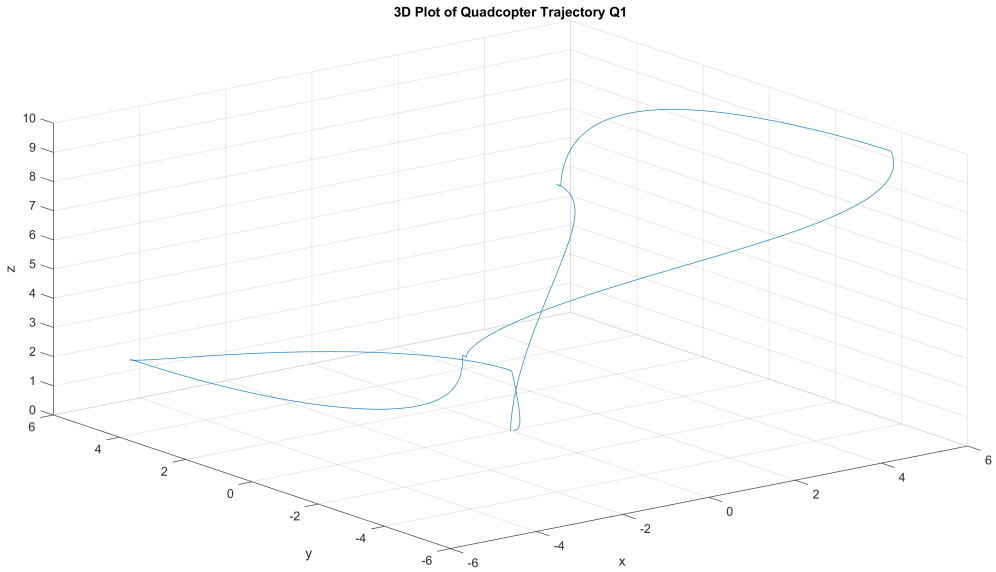

Figure 6: Question 3a, 3D Plots of the simulation, displaying position of the quadcopter over time.

3b

Add Gaussian noise to the robot state measurements. Choose reasonable values for the noise magnitude of each state measurement, by identifying examples of sensors that could be utilised to measure them (you can be creative here). Re-run experiment from 3a) and provide plots of trajectories under noisy measurements.

Examples of sensors that could be used for state measurements include GPS for (x, y) tracking,

barometric pressure sensor for altitude (z) tracking, gyroscope for roll, pitch and yaw (ϕ, θ, ψ) measurements, and an accelerometer for acceleration ($\ddot{x}, \ddot{y}, \ddot{z}$). An indoor navigation system (a combination of camera and reflective patterns on the drone) may also be applied to measure various variables. Suitable additive Gaussian white noise magnitudes for a typical drone modules such as these is a Gaussian distribution, Gaussian magnitude , mean 0, variance of 1. As shown in the code below. The drone was found to fail when the Gaussian magnitude was 0.2 and struggled to stay in the desired positions. However, the maximum noise it could handle with reasonable results was Gaussian_magnitude = 0.15.

```
function [noisy_pos, noisy_pos_dot, noisy_theta, noisy_omega] = noise_func(obj)
                    %Magnitude of 0.01, edit this to see when it fails
                    gaussian_magnitude = 7;
                    % Variance= 1, mean = 0
                    noise = gaussian_magnitude * (sqrt(1) * randn + 0);
                    noise = noise * ones(3,1);
                    noisy_pos = obj.pos + noise;
                    noisy_pos_dot = obj.pos_dot + noise;
                    noisy_theta = obj.th + noise;
                    noisy_omega = obj.omega + noise;
                end
```

Listing 13: Additive White Gaussian Noise Generator function

Figures below show the trajectory and orientation of the drone when Gaussian noise is added to sensed elements such as position, acceleration, angular position and acceleration.

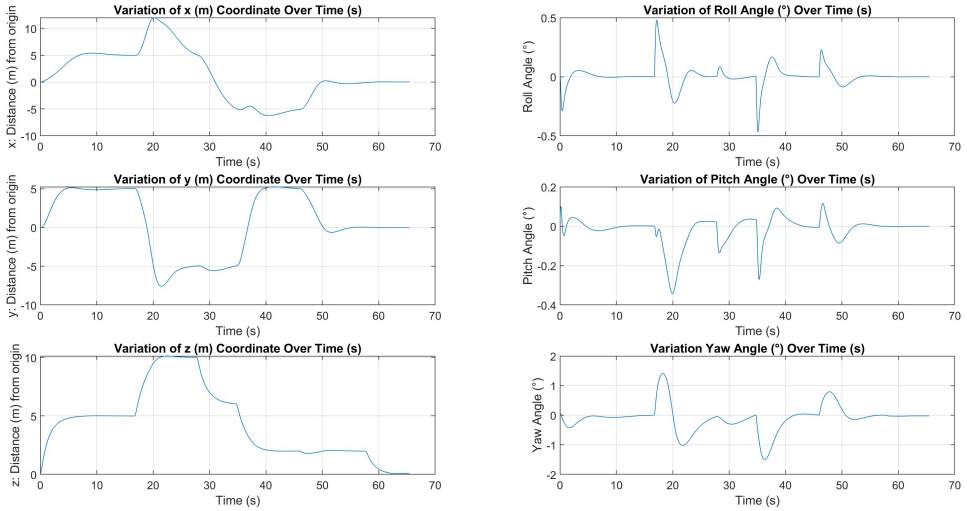

Figure 7: Question 3b, Noisy 2D Plots of the simulation, displaying position and orientation of the quadcopter over time.

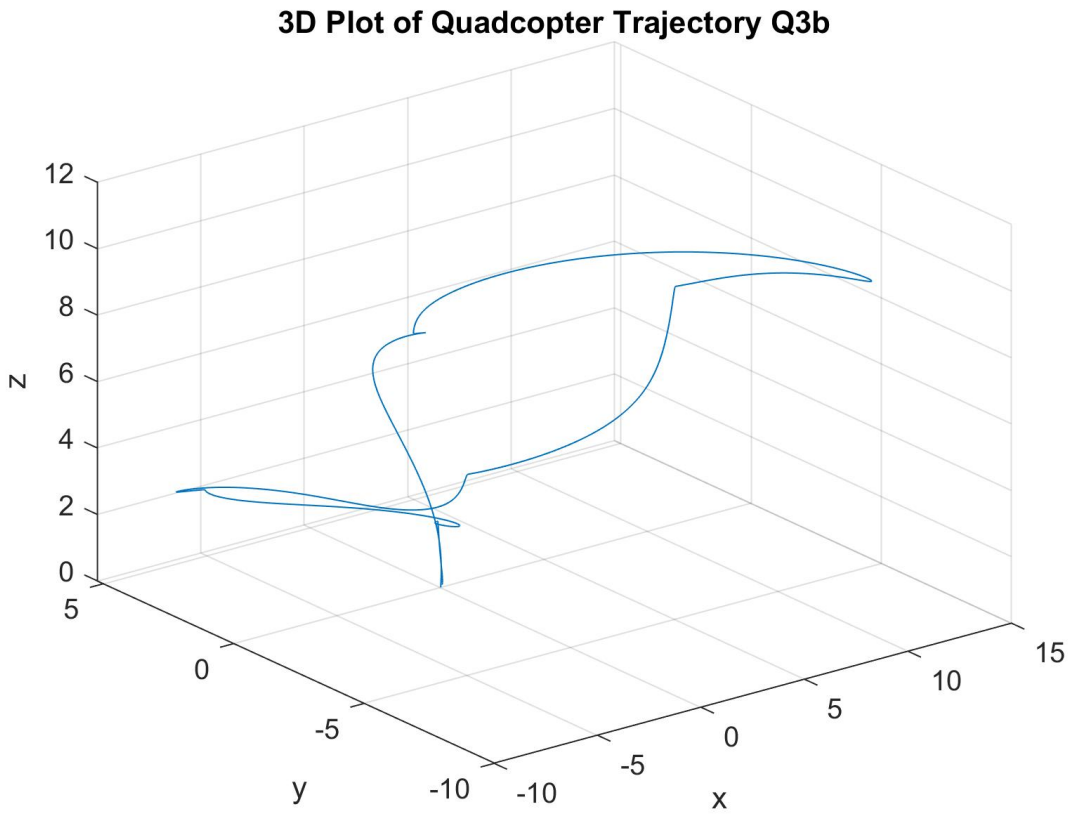

Figure 8: Question 3b,Noisy 3D Plots of the simulation, displaying position of the quadcopter over time.

3c

Add a wind model to your simulation including two components: a mean wind field from a random direction, and a turbulent wind. You can use the wind turbulence model described in [2]. You will need to define some parameters of the model yourself. Investigate how much wind the controller is able to handle before starting to fail. Re-run experiment from 3a), and provide plots of windy trajectories.
The mean wind field is described as a wind magnitude of u_{20} at 20 feet from the ground ($6m$). The resultant wind field then depends logarithmicly on the altitude of the drone [2]. Thus, the navigation frame wind vector w_c at altitude p_z is modelled by:

$$w_c = \begin{bmatrix} u_{20} log(\frac{-p_z}{0.15})/(log(frac200.15)) & 0 & 0 \end{bmatrix}^T$$

Whilst at altitudes of less than 1000 feet (300m), turbulent wind field can be modelled as a random process relative to the drone's velocity. The turbulent scales lengths L_u, L_v, L_w and intensities $\sigma_u, \sigma_v, \sigma_w$ can be calculated form the drone altitude p_z, net velocity V, and mean wind magnitude u_{20}.
$L_w = -p_z$
$L_v = L_u = \frac{-p_z}{(0.177+0.000823h)^{0.4}}$
$\sigma_w = 0.1 u_{20}$

$$\frac{\sigma_u}{\sigma_w} = \frac{\sigma_v}{\sigma_w} = \frac{1}{(0.177 + 0.000823h)^{0.4}}$$

The Dryden spectrum can then be implemented to give the full wind model as shown below.

$$\begin{bmatrix} w_u^{t+T} \\ w_v^{t+T} \\ w_w^{t+T} \end{bmatrix} = \begin{bmatrix} (1-\frac{VT}{L_u})w_u^t + \sqrt{\frac{2VT}{L_u}}\epsilon_u \\ (1-\frac{VT}{L_v})w_v^t + \sqrt{\frac{2VT}{L_v}}\epsilon_v \\ (1-\frac{VT}{L_w})w_w^t + \sqrt{\frac{2VT}{L_w}}\epsilon_w \end{bmatrix}$$

This is implemented in the code listing shown below.

```
function T_w_c = TurbulentWindField(p_z, h, u_20, V, T, w_u, w_v, w_w)
%Inputs; p_z: Altitude of wind platform
    % h: Height of drone
    % u_20: Magnitude of wind at 20 feet
    % V: norm of velocity matrix of drone
    % T: Time interval
    % w_u, w_v, w_w wind at current time

%Outputs: T_w_c: Wind vector at time t+T
    %Turbulent Scale lengths L
    L_u = p_z/(0.177 + 0.000823*h)^1.2;
    L_v = L_u;
    L_w = p_z;

    %Turbulence intensities
    sigma_w = 0.1*u_20;
    sigma_u = sigma_w/(0.177 + 0.000823*h)^0.4;
    sigma_v = sigma_u;

    %Epsilon White noise pertubations with variance sigma^2, noise = sqrt(var^2) *
    random number mean 0.
    % noise = sqrt(variance) *randn(1);
    eps_u = sigma_w * randn(1);
    eps_v = sigma_u * randn(1);
    eps_w = sigma_v * randn(1);

    % wind strength at time t+T:
    T_w_c = [ (1 - ( V * T / L_u )) * w_u + sqrt( 2 * V * T / L_u ) * eps_u;
              (1 - ( V * T / L_v )) * w_v + sqrt( 2 * V * T / L_v ) * eps_v;
              (1 - ( V * T / L_w )) * w_w + sqrt( 2 * V * T / L_w ) * eps_w];
end

function M_w_c = WindMeanField(u_20, p_z)
% Inputs; u_20: Wind magnitude at 20 feet
        % p_z: Altitude of wind platform
% Outputs; M_w_c Mean wind field
    M_w_c = [u_20 * (log10(p_z/0.15)/log10(20/0.15)); 0; 0];
    %Vary u_20 to see what drone can handle
end

function w_total = Wind(pos_dot, p_z, prev_w, T, u_20, obj)
    % Inputs; pos_dot: Drone Velocities
            % p_z: Altitude of wind platform
            % prev_w: Previous wind vector
            % T: Time interval
            % u_20: Magnitude of wind at 20 feet
    %Outputs; w_total: total wind vector including mean wind field and
    %turbulent wind field.
    % Alter u_20 input to this to determine what drone can take
    %Avoid log singularity by having no wind at v low altitude
    if obj.pos(3) <1
        w_total = zeros(3,1);
    else
        V = norm(pos_dot);
        M_w_c = WindMeanField(u_20, p_z);
        T_w_c = TurbulentWindField(p_z, p_z, u_20, V, T, prev_w(1),  prev_w(2), prev_w
    (3));
        w_total = M_w_c + T_w_c;
end
```

Listing 14: Turbulent Wind Field and Mean Wind Field Model functions

Plots of the windy trajectories are shown in Figures 9 and 10 below, the drone was unable to complete tasks within given 100s timeframe, due to the wind interference.

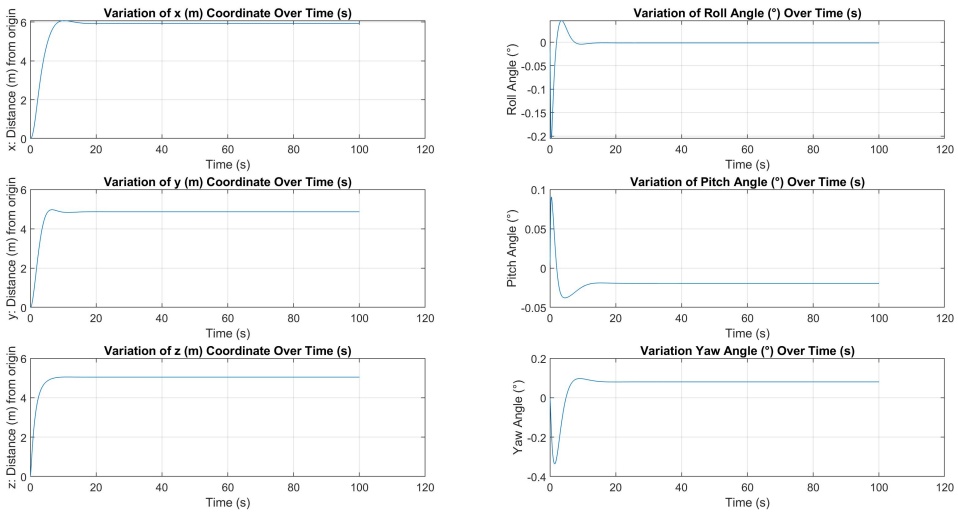

Figure 9: Question 3c, Windy 2D Plots of the simulation, displaying position and orientation of the quadcopter over time.

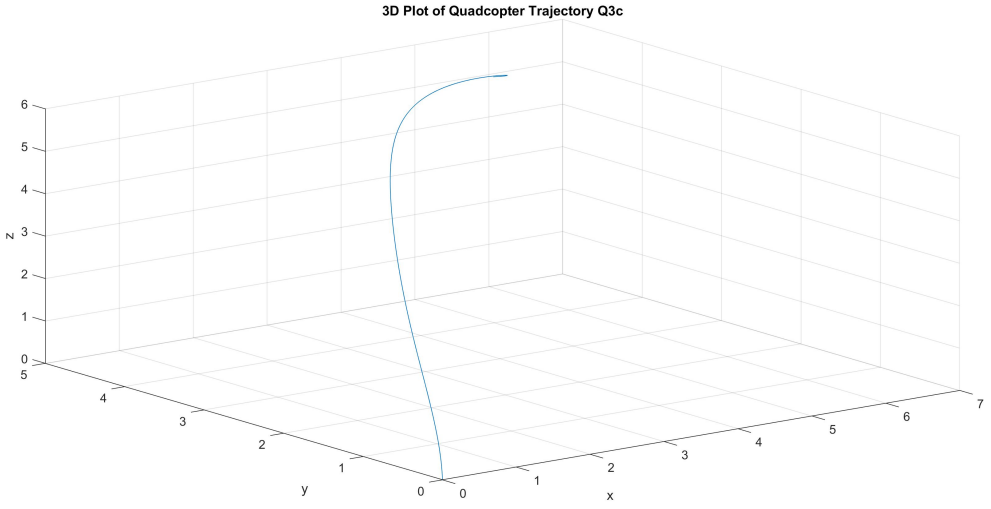

Figure 10: Question 3c,Windy 3D Plots of the simulation, displaying position of the quadcopter over time.

References

[1] A. Gibiansky, Quadcopter Dynamics, Simulation, and Control.
[2] A. Symington, R. De Nardi, S. Julier and S. Hailes, "Simulating quadrotor UAVs in outdoor scenarios," 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014, pp. 3382-3388, doi: 10.1109/IROS.2014.6943033.

Appendix

All code written can be found in the Matlab drive:
https://drive.matlab.com/sharing/acc1f6e7-fd68-4c6b-8a28-bd3757e5c685