Contents

1.3 D Controller and Trajectory Tracking for a Quadcopter

I. 3D Controller and Trajectory Tracking of a Quadcopter 3

I. a System Model 3
I. b Robot Controllers 3
I. c Trajectory Generation 9
I. d Matlab Simulation 10
I. e Programming 11

II. Optimal Control and Kalman Filtering on 2D Quadcopter 16

II. a Math Model 16
II. b Simulation of Nonlinear Dynamics 17
II. c Stabilising Feedback law using Lyapunov 19
II. d Simulation of Closed loop Dynamics 21
II. e Continuous time LQR design 23
II. f Kalman Filter 25

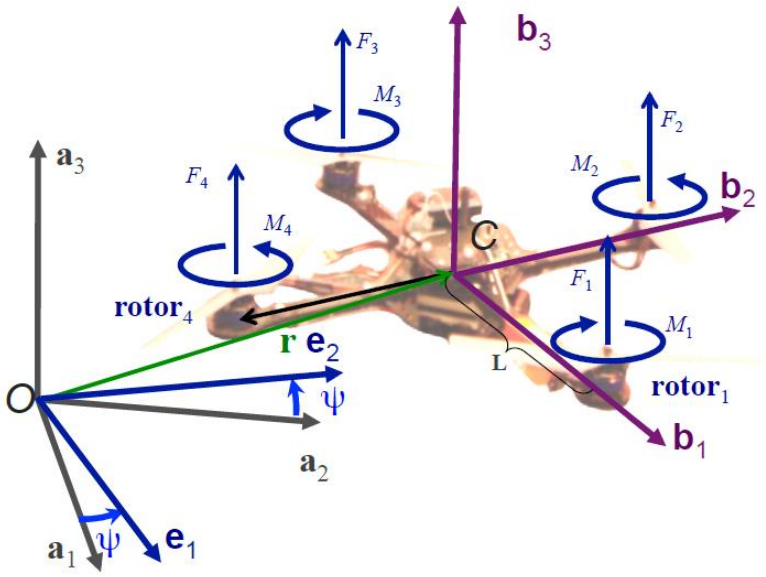
III. Bibliography 30

3 D Controller and Trajectory Tracking for a Quadcopter

1. Introduction :

A Proportional Derivative Controller is being designed to control the motion of the Quadcopter in the 3D X-Y-Z plane. Time parameterized trajectories will be generated such that the quadrotor will navigate through the set way points using by minimizing the snap function.

2. System Model :



1. Quadcopter model with fixed frame-(A) and inertial frame -(B)

2.1 Coordinate Systems

The coordinate systems and free body diagram for the quadrotor are shown in Fig. 1. The inertial frame, $\boldsymbol{A}$, is defined by the axes $\boldsymbol{a1}$; $\boldsymbol{a2}$, and $\boldsymbol{a3}$ around the origin (O). The body frame, $\boldsymbol{B}$, is attached to the center of mass of the quadrotor with $b1$ coinciding with the preferred forward direction and $\boldsymbol{b3}$ perpendicular to the plane of the rotors pointing vertically up during perfect hover. These vectors are parallel to the principal axes. The center of mass is C. Rotor 1 is a distance L away along $\boldsymbol{b1}$, 2 is L away along $\boldsymbol{b2}$, while 3 and 4 are similarly L away along the negative $\boldsymbol{b1}$ and $\boldsymbol{b2}$ respectively.

Since bi are principal axes, the inertia matrix referenced to the center of mass along the $\boldsymbol{bi}$ reference triad, I, is a diagonal matrix.

2.2 Motor Model :

Each rotor has an angular speed ωi and produces a vertical force Fi according to

$$Fi = k_F {\omega_i}^2$$

The k_F is assumed to be ≈ 6.11 X 10^{-8} N/rpm^2

$$Mi = k_M \omega_i{}^2$$

The k_M is assumed to be ≈ 1.5 X 10^{-9} N/rpm^2

2.3 Rigid Body Dynamics

$Z - X - Y$ Euler angles were used to model the rotation of the quadrotor in the world frame. To get from $\boldsymbol{A}$ to $\boldsymbol{B}$, first rotation is done about $\boldsymbol{a3}$ through the the yaw angle, ψ , to get the triad ei. A rotation about the e1 through the roll angle, $\boldsymbol{\Phi}$ gets us to the triad $\boldsymbol{fi}$ (not shown in the figure). A third pitch rotation about $\boldsymbol{f2}$ through θ results in the body-fixed triad $\boldsymbol{bi}$. The rotation matrix for transforming components of vectors in B to components of vectors in A is:

$$^{\mathcal{A}}[R]_{\mathcal{B}} = \begin{bmatrix} c\psi c\theta - s\phi s\psi s\theta & -c\phi s\psi & c\psi s\theta + c\theta s\phi s\psi \\ c\theta s\psi + c\psi s\phi s\theta & c\phi c\psi & s\psi s\theta - c\psi c\theta s\phi \\ -c\phi s\theta & s\phi & c\phi c\theta \end{bmatrix}$$

The components of angular velocity of the robot in the body frame is denoted by p, q and r.

$$^{\mathcal{A}}\omega_{\mathcal{B}} = p\boldsymbol{b}_1 + q\boldsymbol{b}_2 + r\boldsymbol{b}_3.$$

These values are related to the derivatives of the roll, pitch, and yaw angles according to

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} c\theta & 0 & -c\phi s\theta \\ 0 & 1 & s\phi \\ s\theta & 0 & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

Newton's Equations of Motion :

Let $\boldsymbol{r}$ denote the position vector of C in A. The forces on the system are gravity, in the $-\boldsymbol{a_3}$ direction, and the forces from each of the rotors, Fi, in the $\boldsymbol{b_3}$ direction. The equations governing the acceleration of the center of mass are

$$m\ddot{\boldsymbol{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix}$$

The first input u_1 to be

$$u_1 = \Sigma_{i=1}^{4} F_i.$$

Euler's Equation of Motion :

In addition to forces, each rotor produces a moment perpendicular to the plane of rotation of the blade, Mi. Rotors 1 and 3 rotate in the $-\boldsymbol{b_3}$ direction while 2 and 4 rotate in the $+\boldsymbol{b_3}$ direction. Since the moment produced on the quadrotor is opposite to the direction of rotation of the blades, $M1$ and $M3$ act in the $-\boldsymbol{b_3}$ direction while $M2$ and $M4$ act in the $-\boldsymbol{b_3}$

direction. L is the distance from the axis of rotation of the rotors to the center of mass of the quadrotor.

The angular acceleration determined by the Euler equations is

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} 0 & L & 0 & -L \\ -L & 0 & L & 0 \\ \gamma & -\gamma & \gamma & -\gamma \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

where $\gamma = \frac{k_M}{k_F}$ is the relationship between lift and drag. The second set of inputs to the vector of moments $\boldsymbol{u_2}$ is given by :

$$\boldsymbol{u}_2 = \begin{bmatrix} 0 & L & 0 & -L \\ -L & 0 & L & 0 \\ \gamma & -\gamma & \gamma & -\gamma \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix}$$

The angular acceleration is determined by Euler's equation of motion as,

$$I_{xx}\ddot{\phi} = L(F_1 - F_2) = u_2$$

3. Robot Controllers

3.1 Nominal State :

The controller is derived by linearizing the equations of motion and motor model at an operating point that corresponds to nominal hover state is given by : $\boldsymbol{r} = \boldsymbol{r_0}$, $\theta = \Phi = 0$, $\psi = \psi_0$, $\dot{r} = 0$, $\dot{\Phi} = \dot{\psi} = \dot{\theta} = 0$ where roll and pitch angles are small.

At this state the lift from the propellers is given by :

$$F_{i,0} = \frac{mg}{4}, \qquad \text{---------(7)}$$

The nominal values for the inputs at hover are $u_{1,0} = mg$, $u_{2,0} = 0$

Linearizing the newtons equations of motion, we get:

$$\begin{aligned}\ddot{r}_1 &= g(\Delta\theta\cos\psi_0 + \Delta\phi\sin\psi_0)\\ \ddot{r}_2 &= g(\Delta\theta\sin\psi_0 - \Delta\phi\cos\psi_0)\\ \ddot{r}_3 &= \frac{1}{m}u_1 - g\end{aligned}$$ ---------(8)

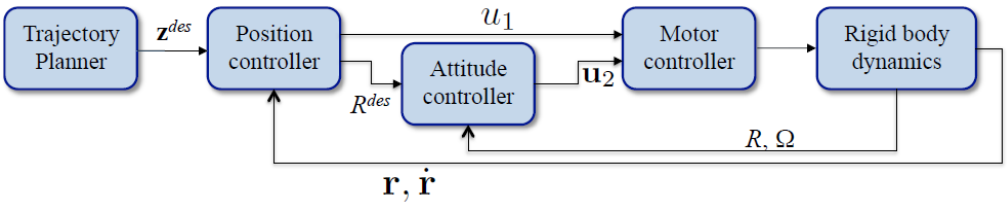
Linearising Euler's equation we get :

$$\begin{bmatrix}\dot{p}\\ \dot{q}\\ \dot{r}\end{bmatrix} = I^{-1}\begin{bmatrix}0 & L & 0 & -L\\ -L & 0 & L & 0\\ \gamma & -\gamma & \gamma & -\gamma\end{bmatrix}\begin{bmatrix}F_1\\ F_2\\ F_3\\ F_4\end{bmatrix}$$ ----------(9)

Assuming the quadcopter to be symmetric, Ixx = Iyy , we get :

$$\begin{aligned}\dot{p} &= \frac{u_{2,x}}{I_{xx}} = \frac{L}{I_{xx}}(F_2 - F_4)\\ \dot{q} &= \frac{u_{2,y}}{I_{yy}} = \frac{L}{I_{yy}}(F_3 - F_1),\\ \dot{r} &= \frac{u_{2,z}}{I_{zz}} = \frac{\gamma}{I_{zz}}(F_1 - F_2 + F_3 - F_4)\end{aligned}$$.

3.2 Position and attitude control



2 Position and attitude control loops

The control problem is to determine the four inputs, $\{u_1, u_2\}$ required to hover or to follow a desired trajectory, z_{des}. As shown in Figure 2, we will use errors in the robot's position to drive a position controller from (8) which directly determines u1. The model in (8) is used to derive a desired orientation. The attitude controller for this orientation is derived from the model in (9).

The inner attitude control loop uses onboard accelerometers and gyros to control the roll, pitch, and yaw, while the outer position control loop uses estimates of position and velocity of the center of mass to control the trajectory in three dimensions.

Attitude Control

A proportional plus derivative (PD) attitude controller is designed to track a trajectory in SO(3) specified in terms of a desired roll, pitch and yaw angle. The development of the controller is based on linearized equations of motion, the attitude must be close to the nominal hover state

where the roll and pitch angles are small. Near the nominal hover state the proportional plus derivative control laws take the form:

$$\boldsymbol{u}_2 = \begin{bmatrix} k_{p,\phi}(\phi_{\text{des}} - \phi) + k_{d,\phi}(p_{\text{des}} - p) \\ k_{p,\theta}(\theta_{\text{des}} - \theta) + k_{d,\theta}(q_{\text{des}} - q) \\ k_{p,\psi}(\psi_{\text{des}} - \psi) + k_{d,\psi}(r_{\text{des}} - r) \end{bmatrix} \text{----------(10)}$$

Position Control

Two position control methods are presented here:

The roll and pitch angles are used as inputs to drive the position of the quadrotor. In both methods, the position control algorithm will determine the desired roll and pitch angles, θdes and Φdes, which can be used to compute the desired speeds from (10). The first, a hover controller, is used for station-keeping or maintaining the position at a desired position vector, $r0$. The second tracks a specified trajectory, $r_T\ (t)$, in three dimensions. In both cases, the desired yaw angle, is specified independently. It can either be a constant, 0 or a time varying quantity, $\psi_T\ (t)$. Let the desired trajectory be:

$$\boldsymbol{z}_{des} = \begin{bmatrix} \boldsymbol{r}_T(t) \\ \psi_T(t) \end{bmatrix}$$

which is provided as an input to specify the trajectory of the position vector and the yaw angle that is to be tracked.

3.3 Hover Controller

For hovering, $r_T\ (t)\ =\ r_0$ and $\psi_T\ (t) =\ 0$. The command accelerations, $\ddot{r}_{i\ des}$, are calculated using a proportional plus derivative (PD) controller. Defining the position error in terms of components using the standard reference triad ai by:

$$e_i = (r_{i,T} - r_i).$$

In order to guarantee that the error goes exponentially to zero, we require :

$$(\ddot{r}_{i,T} - \ddot{r}_{i,\text{des}}) + k_{d,i}(\dot{r}_{i,T} - \dot{r}_i) + k_{p,i}(r_{i,T} - r_i) = 0, \text{ ----------(11)}$$

Where $\dot{r}_{i,T} = \ddot{r}_{i,T} = 0$ for hover.

From (8) we obtain the relationship between the desired accelerations and roll and pitch angles for the attitude controller as:

$$\begin{aligned} \ddot{r}_{1,\text{des}} &= g(\theta_{\text{des}} \cos\psi_T + \phi_{\text{des}} \sin\psi_T) \\ \ddot{r}_{2,\text{des}} &= g(\theta_{\text{des}} \sin\psi_T - \phi_{\text{des}} \cos\psi_T) \\ \ddot{r}_{3,\text{des}} &= \frac{1}{m}u_1 - g. \end{aligned} \text{----------(12)}$$

For hover the third equation yields as :

$$u_1 = mg + m\ddot{r}_{3,\text{des}} = mg - m\left(k_{d,3}\dot{r}_3 + k_{p,3}(r_3 - r_{3,0})\right)$$ ----------(13)

The first and second equations can be used to compute the desired roll and pitch angles for the attitude controller:

$$\phi_{\text{des}} = \frac{1}{g}(\ddot{r}_{1,\text{des}} \sin\psi_T - \ddot{r}_{2,\text{des}} \cos\psi_T)$$
$$\theta_{\text{des}} = \frac{1}{g}(\ddot{r}_{1,\text{des}} \cos\psi_T + \ddot{r}_{2,\text{des}} \sin\psi_T)$$ ----------(14)

The desired roll and pitch are taken to be zero. Since the yaw, $\psi_T(t)$ is prescribed independently by the trajectory generator, we get:

$$p_{\text{des}} = 0$$
$$q_{\text{des}} = 0$$ ----------(15)

$$\psi_{\text{des}} = \psi_T(t)$$
$$r_{\text{des}} = \dot{\psi}_T(t)$$ ----------(16)

3.4 3D Trajectory Control

```
trajhandle = @traj_generator;
waypoints = [0    0   0;
             1    1   1;
             2    0   2;
             3    -1  1;
             4    0   0]';
trajhandle([],[],waypoints);
```

The 3-D Trajectory Controller is used to follow three-dimensional trajectories with modest accelerations so that the near-hover assumptions hold. To derive this controller we follow the steps as in (11) except that, $\ddot{r}_{i\ T}$, and $\dot{r}_{i\ T}$ are no longer zero but are obtained from the specification of the trajectory. If the near-hover assumption holds and the dynamics are linear with no saturation on the inputs, a controller that generates the desired acceleration $\ddot{r}_{i\ des}$ according to (11) is guaranteed to drive the error exponentially to zero. However, it is possible that the commanded trajectory has twists and turns that are too hard to follow perfectly because of errors in the model or limitations on the input thrusts.

Let r_T denote the closest point on the desired trajectory to the the current position r, and let the desired velocity and acceleration obtained by differentiating the specified trajectory be given by $\dot{r}_T$ and $\ddot{r}_T$ respectively. Let the unit tangent vector of the trajectory (unit vector along $\dot{r}_T$) be $\hat{t}$. The unit normal to the trajectory, $\hat{t}$, is derived by differentiating the tangent vector

with respect to time or arc length, and finally the unit binormal vector is the cross product $\hat{b} = \hat{t}\, X\, \hat{n}$. We define the position and velocity errors according to the following equations:

$$\boldsymbol{e}_p = ((\boldsymbol{r}_T - \boldsymbol{r}) \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} + ((\boldsymbol{r}_T - \boldsymbol{r}) \cdot \hat{\boldsymbol{b}})\hat{\boldsymbol{b}}$$
$$\boldsymbol{e}_v = \dot{\boldsymbol{r}}_T - \dot{\boldsymbol{r}}.$$

Here the position error in the tangential direction is ignored and only the position error in the plane that is normal to the curve at the closest point is considered. The commanded acceleration is calculated, $\ddot{r}_{i\ des}$, from the PD feedback as shown in (11). In vector notation, this is:

$$(\ddot{\boldsymbol{r}}_T - \ddot{\boldsymbol{r}}_{\text{des}}) + \boldsymbol{k}_d\boldsymbol{e}_v + \boldsymbol{k}_p\boldsymbol{e}_p = 0,$$

Finally equations (14 - 16) is used to compute the desired roll and pitch angles.

4. Trajectory Generation

A necessary condition for a minimum snap trajectory is that it is a 7th order polynomial. If we are given a set of n + 1 intermediate waypoints, the minimum snap trajectory is a piecewise polynomial composed of n 7th order polynomials. Each polynomial piece travels between a pair of waypoints and takes a known amount of time Ti.

Let p_i be the i^{th} such polynomial between positions w_i and w_{i+1}, which takes T_i time to complete, let

$$S_i = \sum_{k=0}^{i-1} T_k.$$

Then, the polynomial p_i has the form:

$$p_i(t) = \alpha_{i0} + \alpha_{i1}\frac{t - S_i}{T_i} + \alpha_{i2}\left(\frac{t - S_i}{T_i}\right)^2 + \ldots + \alpha_{i7}\left(\frac{t - S_i}{T_i}\right)^7$$

To obtain the complete equation for the piecewise trajectory, we need to solve for all the coefficients α_{ij} . These coefficients must satisfy a series of constraints.

First, the polynomials has to go through the given waypoints:

$$p_0(0) = w_0 = \alpha_{00}$$

$$p_i(S_i) = w_{i+1} \text{ for all } i = 0..n$$

Secondly, the quadrotor must start and stop at rest:

$$p_n^{(k)}(0) = 0 \text{ for all } 0 \leq k \leq 3$$

$$p_n^{(k)}(S_n) = 0 \text{ for all } 0 \leq k \leq 3$$

Additionally, The first four derivatives are continuous at the waypoints:

$$p_{i-1}^{(k)}(S_i) = p_i^{(k)}(S_i) \text{ for all } i = 1..n \text{ and for all } k = 1..4$$

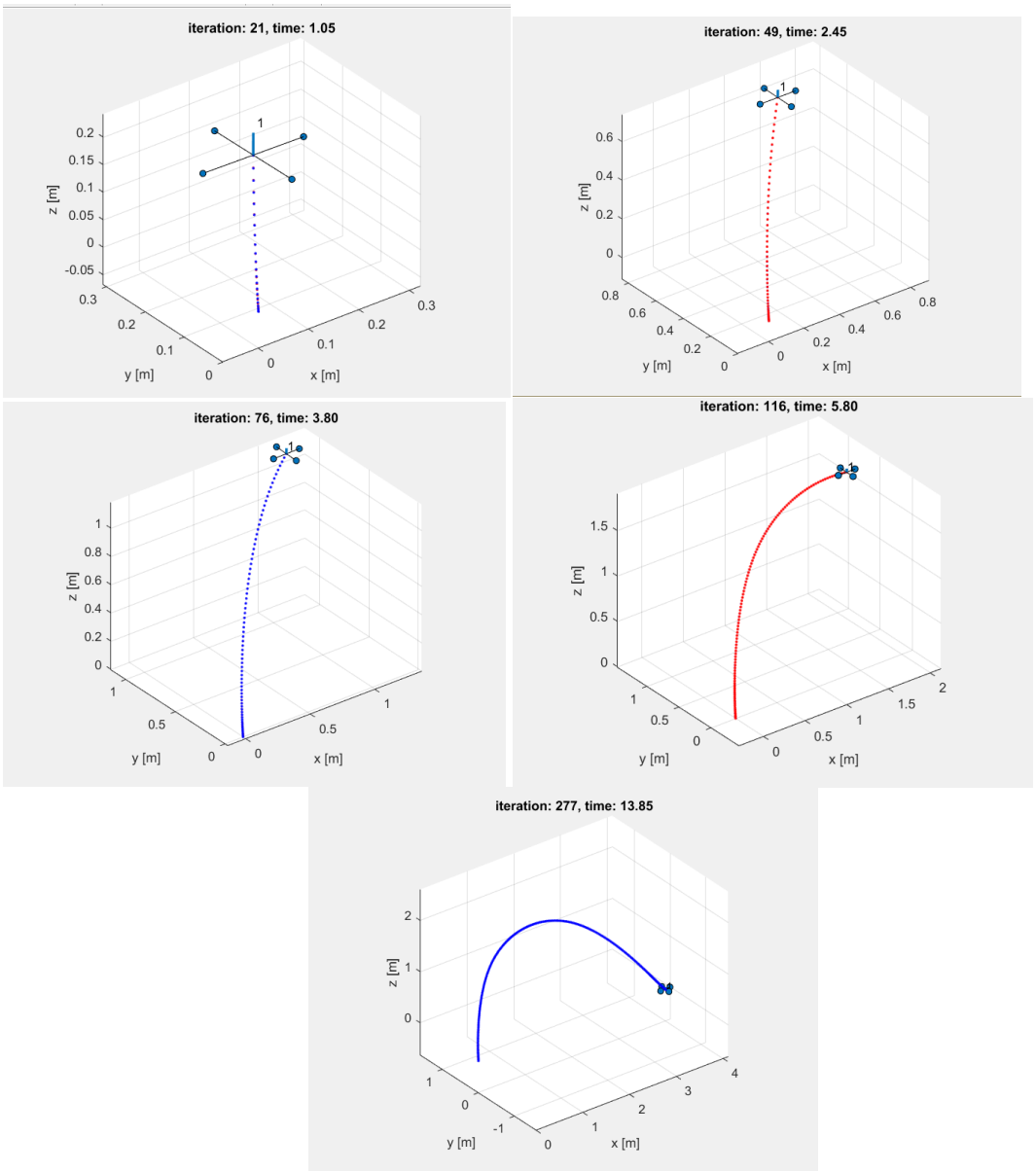
To form a complete set of constraints, we can add additional constraints of the form:

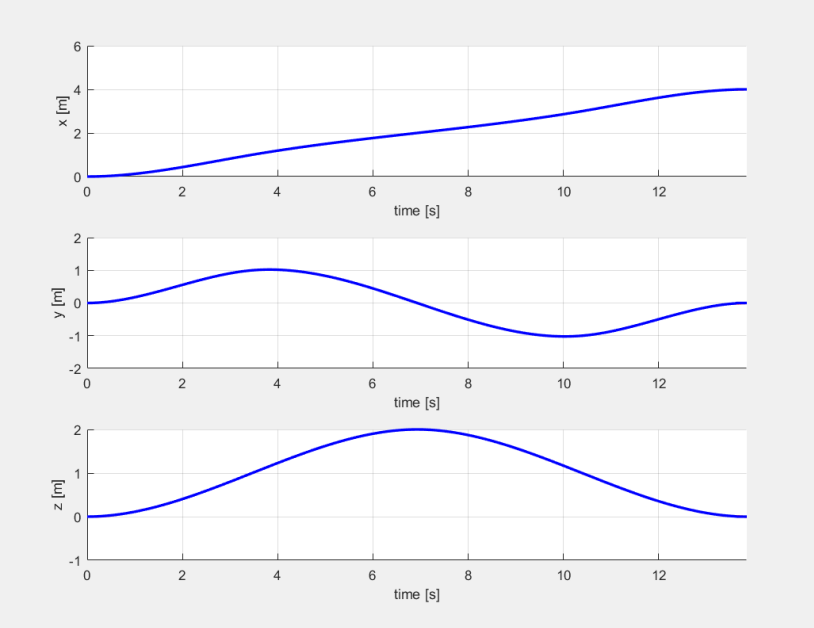
$$p_{i-1}^{(k)}(S_i) = p_i^{(k)}(S_i) \text{ for all } i = 1...n \text{ and for all } k = 1...7$$

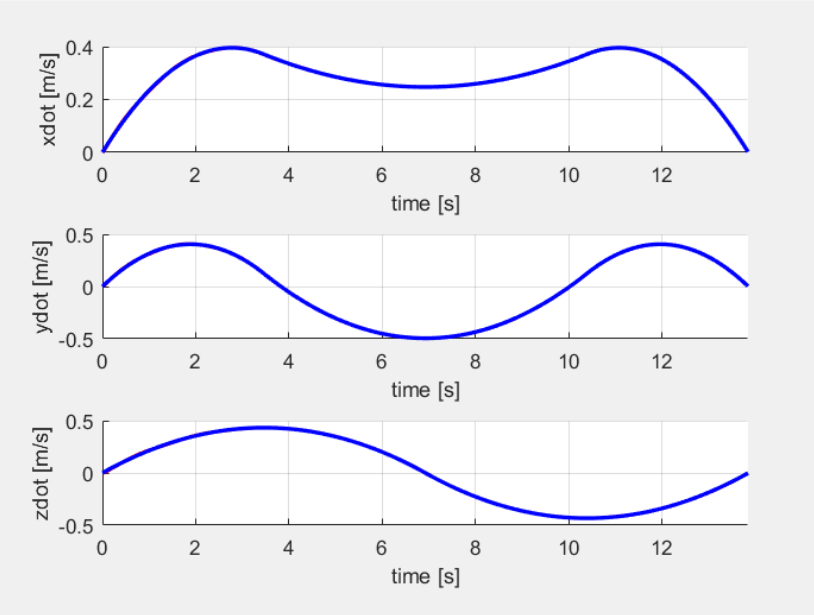
Now the number of constraint equations matches the number of unknown coefficients α_{ij} and thus can be used to solve for the unknown coefficients.

5. Matlab Simulation :

The following is the simulation of the quadcopter tracing the trajectory in 3D space. The quadcopter passes through the intermediate waypoints specified as the input by using the trajectory generator function using the optimized path generated by it.







6. Programming

6.1 System Parameters :

```
function params = sys_params()
% SYS_PARAMS basic parameters for the quadrotor

m = 0.18; % kg
g = 9.81; % m/s/s
I = [0.00025,   0,          2.55e-6;
     0,         0.000232,   0;
     2.55e-6,   0,          0.0003738];

params.mass = m;
```

```
params.I    = I;
params.invI = inv(I);
params.gravity = g;
params.arm_length = 0.086; % m

params.minF = 0.0;
params.maxF = 2.0*m*g;
```

6.2 System Controller :

```
function [F, M] = controller(t, state, des_state, params)
%CONTROLLER  Controller for the quadrotor
%   state: The current state of the robot with the following fields:
%   state.pos = [x; y; z], state.vel = [x_dot; y_dot; z_dot],
%   state.rot = [phi; theta; psi], state.omega = [p; q; r]
%   des_state: The desired states are:
%   des_state.pos = [x; y; z], des_state.vel = [x_dot; y_dot; z_dot],
%   des_state.acc = [x_ddot; y_ddot; z_ddot], des_state.yaw,
%   des_state.yawdot
%   params: robot parameters

M = zeros(3,1);
Kpx = 25;
Kdx = 1.5;
Kpz = 22;
Kdz = 1.5;
Kpphi = 20;
Kdphi = .1;

Kpy = Kpx;
Kdy = Kdx;
Kppsi = Kpphi;
Kdpsi = Kdphi;
Kptheta = Kpphi;
Kdtheta = Kdphi;


r1des_ddot = des_state.acc(1)+Kdx*(des_state.vel(1)-
state.vel(1))+Kpx*(des_state.pos(1)-state.pos(1));
r2des_ddot = des_state.acc(2)+Kdy*(des_state.vel(2)-
state.vel(2))+Kpy*(des_state.pos(2)-state.pos(2));
r3des_ddot = des_state.acc(3)+Kdz*(des_state.vel(3)-
state.vel(3))+Kpz*(des_state.pos(3)-state.pos(3));

phi_des     = (r1des_ddot*des_state.yaw - r2des_ddot)/params.gravity ;
theta_des   = (r1des_ddot + r2des_ddot*des_state.yaw)/params.gravity ;

M =[Kpphi*(phi_des-state.rot(1)) + Kdphi*(0-state.omega(1));
    Kptheta*(theta_des-state.rot(2)) + Kdtheta*(0-state.omega(2));
    Kppsi*(des_state.yaw-state.rot(3)) + Kdpsi*(des_state.yawdot-
state.omega(3))];

F = params.mass*(params.gravity+r3des_ddot);
```

6.3 Trajectory Generator :

```
function [ desired_state ] = traj_generator(t, state, waypoints)

k= [0 0; 0 0; 0 0];
persistent waypoints0 traj_time d0

if nargin > 2 %so that after initialisation the foll is not calculated for
every call, only t and state are passed for calls after intitialisation.
    d = waypoints(:,2:end) - waypoints(:,1:end-1); % distance between points
[3x4]matrix
    d0 = 2 * sqrt(d(1,:).^2 + d(2,:).^2 + d(3,:).^2); % time interval between
points
    traj_time = [0, cumsum(d0)]; %cumilative time in between waypoints
    waypoints0 = waypoints; % waypoints

else
    if(t > traj_time(end)-0.25)
        t = traj_time(end);
        desired_state.pos = waypoints0(:,end);
    desired_state.vel=0;
        desired_state.acc =0; %(v_f - v_i)/t
    desired_state.yaw = 0;
    desired_state.yawdot = 0;
    end
%     t_index = find(traj_time >= t,1); %finds one position of traj_time that's
greater than t
%
%     if(t_index > 1)
%         t = t - traj_time(t_index-1);
%     end
    if(t == 0)
        desired_state.pos = waypoints0(:,1);
    else
%          desired_state.pos = ppval(pp,t);


%         scale = t/d0(t_index-1);
%         desired_state.pos = (1 - scale) * waypoints0(:,t_index-1) + scale *
waypoints0(:,t_index);

    end

  pp=spline(traj_time,[k(:,1) waypoints0 k(:,2)]);
    desired_state.pos = ppval(pp,t);
    desired_state.vel=((ppval(pp,t+.001)-ppval(pp,t))/(.001));
    v1 = (ppval(pp,t+.0005)-ppval(pp,t))/(.0005);
    v2 = (ppval(pp,t+.001)-ppval(pp,t+.0005))/(.0005);
    desired_state.acc = ((v2-v1)/(.0005)); %(v_f - v_i)/t
    desired_state.yaw = 0;
    desired_state.yawdot = 0;
end
%
```

6.4 Runsim Simulation :

```
close all;
addpath('utils');
trajhandle = @traj_generator;
waypoints = [0    0    0;
             1    1    1;
```

```
             2    0   2;
             3    -1  1;
             4    0   0]';
trajhandle([],[],waypoints);
%% controller
controlhandle = @controller;
t, state] = simulation_3d(trajhandle, controlhandle);
```

6.5 Equations of Motion :

```
function sdot = quadEOM(t, s, controlhandle, trajhandle, params)
% QUADEOM Wrapper function for solving quadrotor equation of motion
%   quadEOM takes in time, state vector, controller, trajectory generator
%   and parameters and output the derivative of the state vector, the
%   actual calcution is done in quadEOM_readonly.
%
% INPUTS:
% t             - 1 x 1, time
% s             - 13 x 1, state vector = [x, y, z, xd, yd, zd, qw, qx, qy, qz,
p, q, r]
% controlhandle - function handle of your controller
% trajhandle    - function handle of your trajectory generator
% params        - struct, output from sys_params() and whatever parameters you
want to pass in
%
% OUTPUTS:
% sdot          - 13 x 1, derivative of state vector s
%
% convert state to quad stuct for control
current_state = stateToQd(s);
% Get desired_state
desired_state = trajhandle(t, current_state);
% get control outputs
[F, M] = controlhandle(t, current_state, desired_state, params);
% compute derivative
sdot = quadEOM_readonly(t, s, F, M, params);

end

function sdot = quadEOM_readonly(t, s, F, M, params)
% QUADEOM_READONLY Solve quadrotor equation of motion
%   quadEOM_readonly calculate the derivative of the state vector
%
% INPUTS:
% t      - 1 x 1, time
% s      - 13 x 1, state vector = [x, y, z, xd, yd, zd, qw, qx, qy, qz, p, q,
r]
% F      - 1 x 1, thrust output from controller (only used in simulation)
% M      - 3 x 1, moments output from controller (only used in simulation)
% params - struct, output from nanoplus() and whatever parameters you want to
pass in
%
% OUTPUTS:
% sdot   - 13 x 1, derivative of state vector s
%
%************ EQUATIONS OF MOTION ************************
% Limit the force and moments due to actuator limits
A = [0.25,                      0, -0.5/params.arm_length;
     0.25,  0.5/params.arm_length,                      0;
     0.25,                      0,  0.5/params.arm_length;
     0.25, -0.5/params.arm_length,                      0];
```

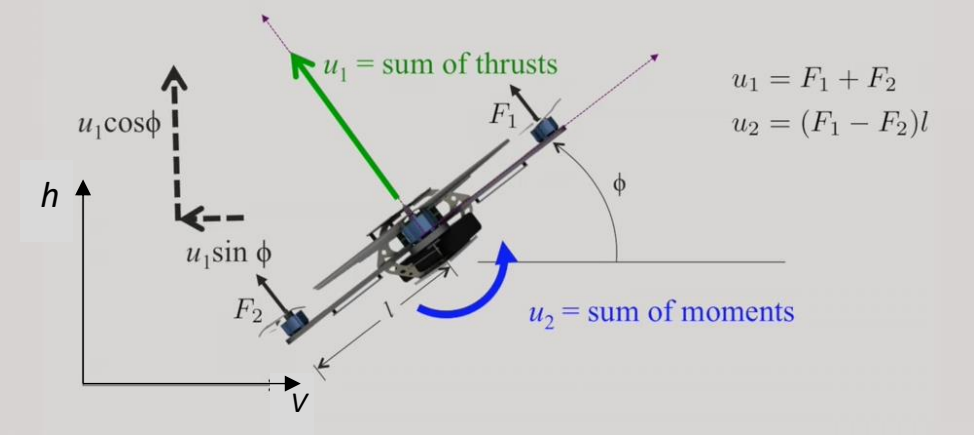
```
prop_thrusts = A*[F;M(1:2)]; % Not using moment about Z-axis for limits
prop_thrusts_clamped = max(min(prop_thrusts, params.maxF/4), params.minF/4);

B = [                  1,                  1,                  1,
1;
                       0, params.arm_length,                  0, -
params.arm_length;
     -params.arm_length,                  0, params.arm_length,
0];
F = B(1,:)*prop_thrusts_clamped;
M = [B(2:3,:)*prop_thrusts_clamped; M(3)];

% Assign states
x = s(1);y = s(2);
z = s(3);xdot = s(4);
ydot = s(5);zdot = s(6);
qW = s(7);qX = s(8);
qY = s(9);qZ = s(10);
p = s(11);q = s(12);
r = s(13);
quat = [qW; qX; qY; qZ];
bRw = QuatToRot(quat);
wRb = bRw';

% Acceleration
accel = 1 / params.mass * (wRb * [0; 0; F] - [0; 0; params.mass *
params.gravity]);

% Angular velocity
K_quat = 2; %this enforces the magnitude 1 constraint for the quaternion
quaterror = 1 - (qW^2 + qX^2 + qY^2 + qZ^2);
qdot = -1/2*[0, -p, -q, -r;...
             p,  0, -r,  q;...
             q,  r,  0, -p;...
             r, -q,  p,  0] * quat + K_quat*quaterror * quat;

% Angular acceleration
omega = [p;q;r];
pqrdot   = params.invI * (M - cross(omega, params.I*omega));

% Assemble sdot
sdot = zeros(13,1); sdot(1)  = xdot;
sdot(2)  = ydot; sdot(3)  = zdot;
sdot(4)  = accel(1);sdot(5)  = accel(2);
sdot(6)  = accel(3);sdot(7)  = qdot(1);
sdot(8)  = qdot(2);sdot(9)  = qdot(3);
sdot(10) = qdot(4);sdot(11) = pqrdot(1);
sdot(12) = pqrdot(2);sdot(13) = pqrdot(3);
end
```

OPTIMAL CONTROL AND KALMAN FILTERING ON A 2D QUADCOPTER

Quadcopters are a useful tool for researchers to test and evaluate new ideas in a number of different fields, including flight control theory, navigation, real time systems, and robotics. Academics are working together to make significant improvements to the way quadcopters perform increasingly complex aerial maneuvers.

In this project we consider the dynamics of a quadcopter in the 2D vertical plane and provide an optimal control using concepts from Linear Quadratic Regulator and Algebraic Riccatti Equation. A Kalman Filter is implemented later to generate estimates of the states of the system from noisy measurements.

Part 1 . Mathematical modelling of a 2D Quadcopter:



[3]. Free Body Diagram of the Quadcopter in 2D vertical plane

The equations of motion of the quadcopter in the vertical plane is given by :

$$m\ddot{h} = u_1 \sin\phi,$$
$$m\ddot{v} = -mg + u_1 \cos\phi,$$
$$I\ddot{\phi} = u_2$$

where (h, v) denote the quadrotor (horizontal, vertical) position and ϕ denotes the quadrotor's rotation, (m, I) denote quadrotor (mass, inertia), g is acceleration due to gravity, and ($u1$, $u2$) denote the net (thrust, torque) applied by the spinning rotors.

If we measure or observe positions (h, v), e.g. with GPS, then the control system model is :

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ F((q,\dot{q}),u) \end{bmatrix} = f((q,\dot{q}),u),\ y = h(q,\dot{q})$$

where $q = (h, v, \theta) \in \mathbb{R}^3$, $u = (u_1, u_2) \in \mathbb{R}^2$, $F : \mathbb{R}^3 \times \mathbb{R}^2 \to \mathbb{R}^3$ is defined by

$$F((q,\dot{q}),u) = \ddot{q} = \begin{bmatrix} \frac{u_1}{m}\sin\phi \\ -g + \frac{u_1}{m}\cos\phi \\ \frac{u_2}{I} \end{bmatrix},$$

and $h : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$h(q,\dot{q}) = (h, v).$$

Simulation of the Nonlinear dynamics:

1(a). The Quadcopter dynamics were simulated using Forward Euler numerical approximation. Functions *fun* and h, that take (t, u, x) as arguments and return the dynamics $fun(t,x,u)$ and the observation $h(t,x,u)$ respectively.

Taylor expansion of the function y around t_0 is given by:

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}h^2 y''(t_0) + O(h^3).$$

So the finite difference formula for the derivative becomes:

$$y'(t_0) \approx \frac{y(t_0 + h) - y(t_0)}{h}$$

Function fun:

$$f(t,x,u) = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \\ \dot{x_5} \\ \dot{x_6} \end{bmatrix} = \begin{bmatrix} x_4 \\ x_5 \\ x_6 \\ \frac{sin(x_3)u_1}{m} \\ \frac{cos(x_3)u_1}{m} - g \\ \frac{u_2}{I} \end{bmatrix}$$

Matlab:
```
function xdot=fun(t,x,U)

    g=9.81;m=1;I=1;w=1;
    u = U(:,t);

    h=x(1); v=x(2); th=x(3); dh=x(4); dv=x(5); dth=x(6); u1=u(1); u2=u(2);

    dot2h = (u1*sin(th))/m;
    doth  = dh;
    dot2v = -m*g +(cos(th)*u1);
    dotv = dv;
    dot2th = (u2*1/I);
    dotth = dth;

    xdot = [doth;dotv;dotth;dot2h;dot2v;dot2th];
```

1(b) . Equilibrium states and inputs

For equilibrium df = 0. This will help us find the equilibrium states. Substituting u1 and u2 in f, we get equilibrium to be (0,0,0,0,0,0) for $(h,v,\theta,\dot{h},\dot{v},\dot{\theta})$

1(c) . Forward Euler Simulation

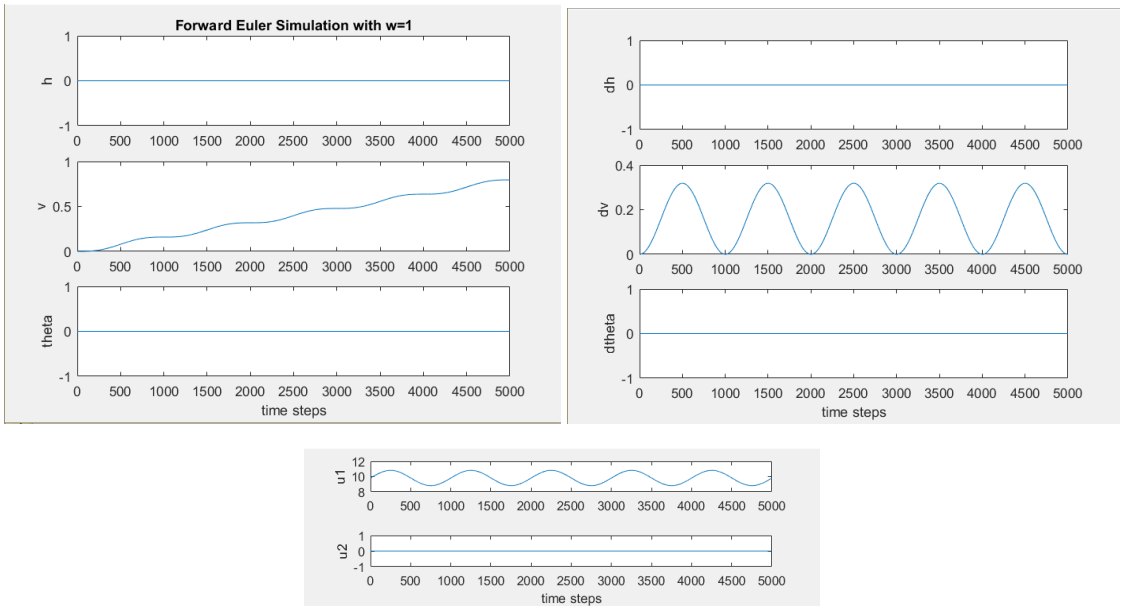
$$u(t) = \begin{bmatrix} mg + \sin(2t\pi\omega) \\ 0 \end{bmatrix}$$

Matlab:
```
N=10000; %number of time steps
    t=linspace(0,10, N+1); %taking the time step
    step = t(2)-t(1);
    w=1;
```

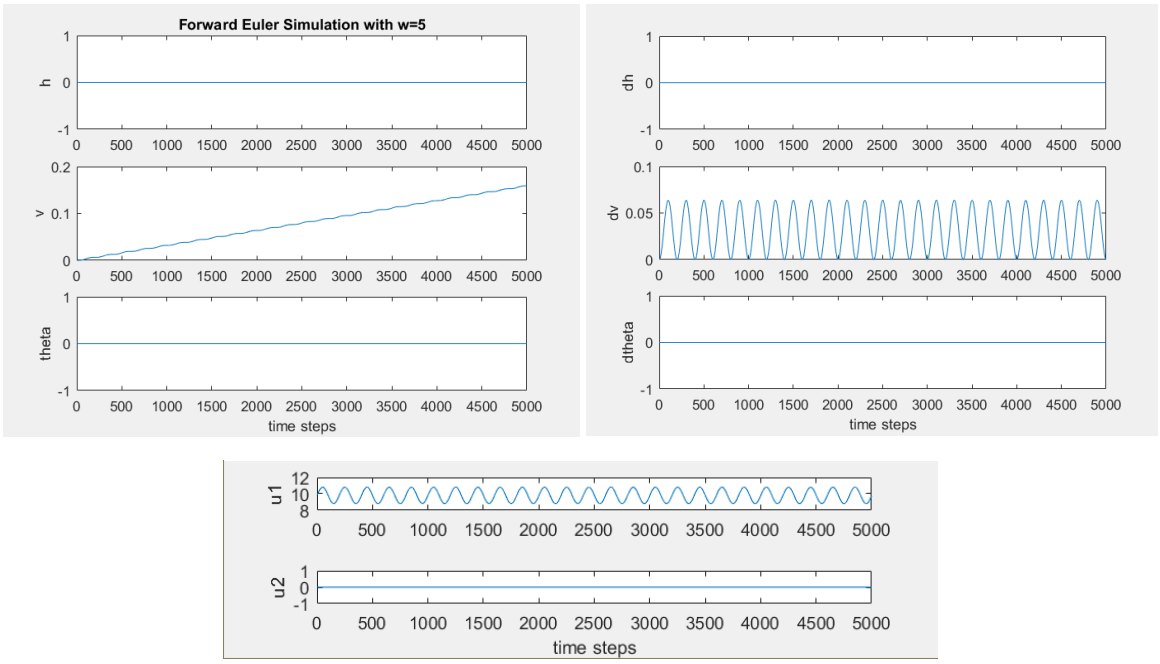
```
Qx = zeros(6, N+1); %initial condition is set at x[:,0] to [0;0;0;0;0;0]
        for i =1:N+1
                u(:,i)= [(m*g + sin(2*t(i)*pi*w));0];
        end
% Simulation forward euler:
        for i=2:N+1
                % defining the U vector
                Qx(:,i) = step*(fun(i,Qx(:,i-1),u))+ Qx(:,i-1);
        end
```

Plots :

Forward Euler Simulation with w=1 :



Forward Euler Simulation with w=5 :



Part 2 . <u>Stabilisation of the 2D Quadcopter:</u>

2(a) . Linearization

The system is linearised around the equilibrium $[0\ 0.1\ 0\ 0\ 0\ 0]^T$ using first order approximation using Jacobian matrix.

$$\frac{\partial \dot{f}}{\partial x_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial x_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial x_3} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{\cos(x_3)u_1}{u_1} \\ -\frac{\sin(x_3)u_1}{m} \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial x_4} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \frac{\partial \dot{f}}{\partial x_5} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial x_6} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial u_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{\sin(x_3)}{m} \\ \frac{\cos(x_3)}{m} \\ 0 \end{bmatrix}; \frac{\partial \dot{f}}{\partial u_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Substituting values of equlibrium states $[0\ 0.1\ 0\ 0\ 0\ 0]^T$ and inputs $[mg\ 0]^T$, we get :

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{sin(x_3)}{m} & 0 \\ \frac{cos(x_3)}{m} & 0 \\ 0 & 1 \end{bmatrix}$$

2(b) . Controllability

The controllability matrix of the system is given by : $\mathcal{C} = [\ A\ ,\ AB\ ,\ A^2B\ ,\ A^3B\ ,\ A^4B\ ,\ A^5B\ ]$

The rank of the controllability matrix needs to be equal to the rank of the A matrix for the system to be controllable. $\mathcal{C} =$

$$\mathcal{C} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9.8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & g & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}; \qquad \text{Rank}(\mathcal{C}) = 6 = \text{Rank}(A)$$

Thus the system is controllable.

2(c) . Stabilizing state feedback control law using Lyapunov :

2(c) 1 . If (A, B) is controllable so is $(-\lambda I - A, B)$ for every $\lambda \in \mathbb{R}$.

Let us consider A which is a controllable matrix. First we write:

$$Ax = \lambda * x$$

Where x and λ are the eigenvector and eigenvalue pair of A. If we manipulate this expression by multiplying by -1 and subtracting μx on both sides we obtain:

$$-Ax - \mu * x = -\lambda * x - \mu * x$$
$$(-A - \mu)x = (-\lambda - \mu)x$$

Clearly, given (λ, x) is the eigenvalue and eigenvector pair for the controllable A then it follows from above that $(-A - \mu)$ is also controllable. Since they both have the same eigenvectors and therefore (λ, x) is an eigenvalue-eigenvector pair for A if and only if $(-\mu - \lambda, x)$ is an eigenvalue-eigenvector pair for $(-\mu I - A)$, implying that all these vector directions are controllable.

2(c) 2 . Finding K that stabilizes the system :

If $(-\mu I - A)$ is stable, the Lyapunov equation can be expressed as,

$$(-\mu I - A)W + W(-\mu I - A)^T = -BB^T$$

$$(A)W + W(A)^T - BB^T = -2\mu W$$

We can rewrite this in the Lyapunov form by multiplying by W^{-1} [ ] W^{-1} and letting W^{-1} = P

Let $B^T P = 2K$, then

$$P(A) + (A)^T P - PBB^T P = -2\mu P$$

$$P(A) + (A)^T P - 2PBK = -2\mu P$$

$$P(A) - PBK + (A)^T P - PBK = -2\mu P$$

$$P(A - BK) + (A - BK)^T P = -2\mu P$$

$$\text{Thus } K = \frac{1}{2} B^T P$$

Let the desired eigen values be, $p = [-3.5;\ -3.6;\ -3.7;\ -3.8;\ -4.5;\ -4.9]$

Matlab:
```
%(-muI-A)W + W(-muI-A)' + BB' =  lyapunov
lA = (diag(p) - A);
lQ = B*B';
W = lyap(lA,lQ);

% from the calculation from 2(C)(3) we know that inv(P) = W; sp P = inv(W)
lP = inv(W);
K0= 0.5*B'*lP ;
```

The $K0$ for the desired poles is found to be,

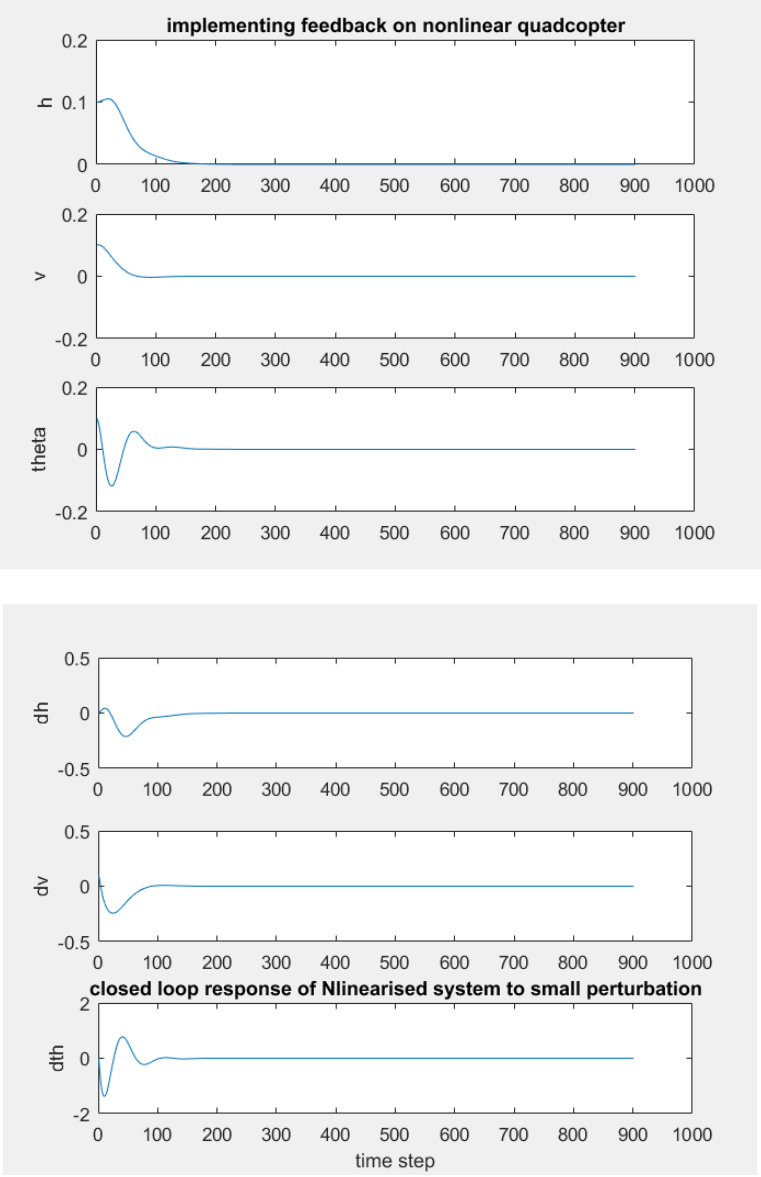
$$K0 = \begin{bmatrix} 0 & 29.1600 & 0 & 0 & 8.1000 & 0 \\ 157.6800 & 0 & 174.9000 & 90.0582 & 0 & 15.9000 \end{bmatrix}$$

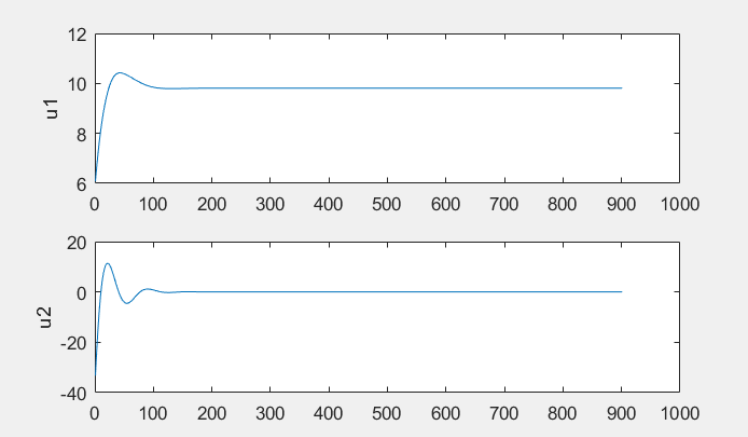
The eigen values of the closed loop system , $(A - B\ K0)$ are stable :

$$\begin{bmatrix} -4.0094 + 8.9135i \\ -4.0094 - 8.9135i \\ -3.9406 + 0.8051i \\ -3.9406 - 0.8051i \\ -4.0500 + 3.5718i \\ -4.0500 - 3.5718i \end{bmatrix}$$

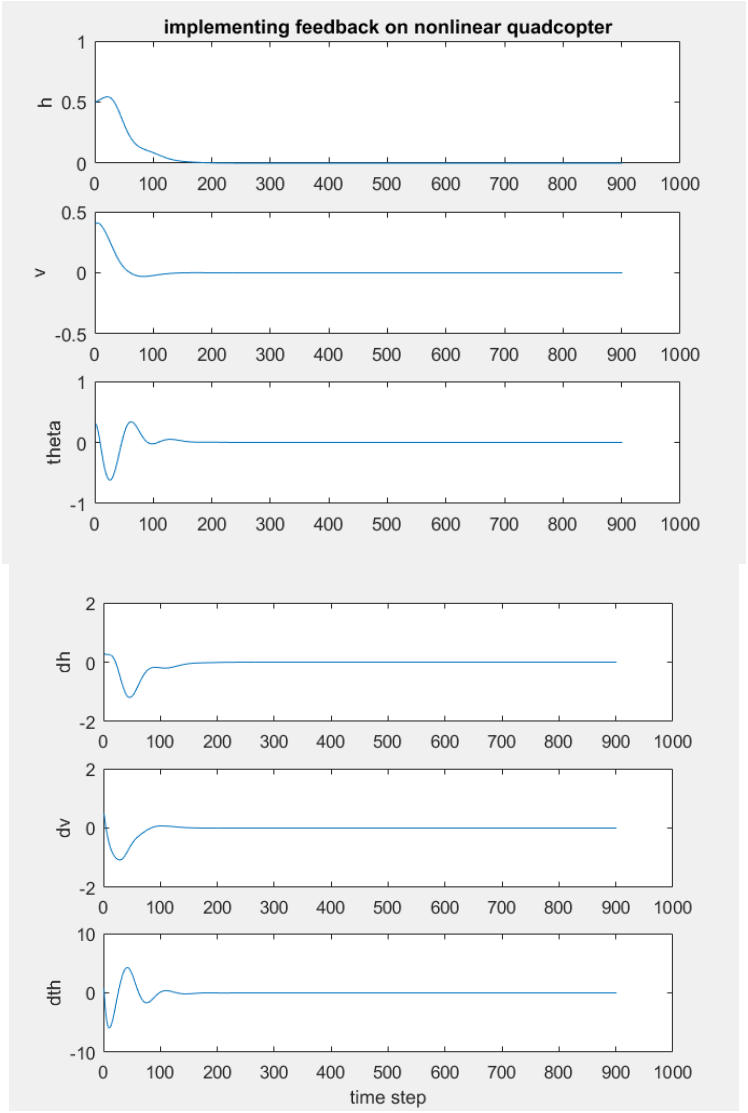
2(d) . Simulation of the closed loop system in the equilibrium neighborhood:

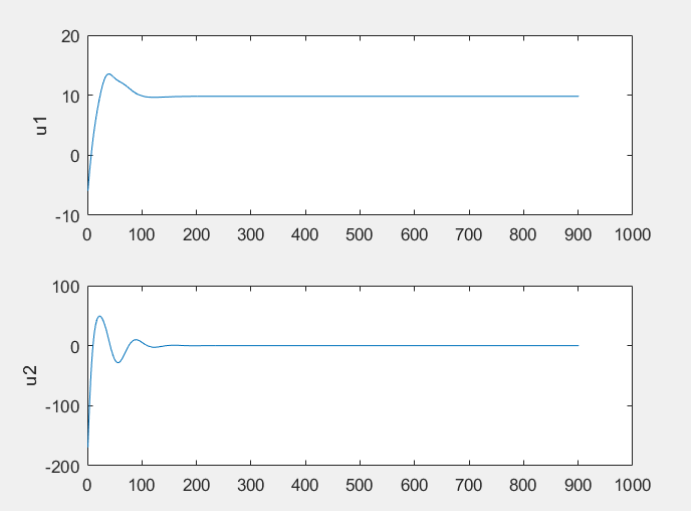
Closed Loop Response of Nonlinear System to $x0$ = $[0.1, 0.1, 0.1, 0, 0.1, 0]^T$**:**





Closed Loop Response of Nonlinear System to $x\mathbf{0}$ = $[0.1, 0.1, 0.1, 0, 0.1, 0]^T$:





Part 3 . Continuous time LQR design:

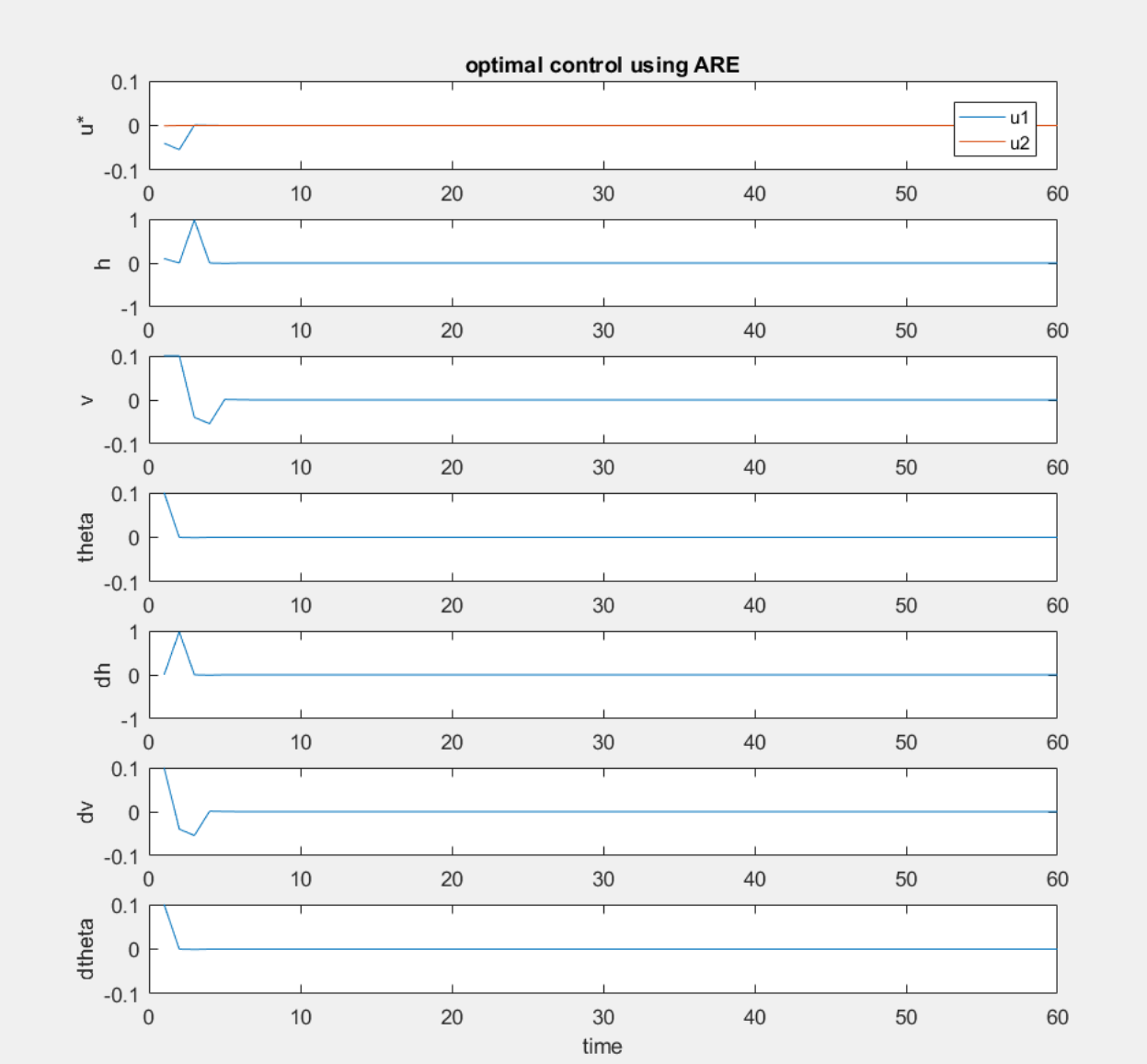
3(a).1 Generating positive definite Matrices :

Matlab:
```
function pd=pdef(size)
% Generate a dense n x n symmetric, positive definite matrix
pd = rand(size,size); % generate a random n x n matrix
% construct a symmetric matrix using either
pd = 0.5*(pd+pd');
% or A = A*A';
pd = pd + size*eye(size);
pd;
```

3(a).2 Numerical Scheme to solve Algebraic Riccatti Equation :

Matlab:
```
m=2; n=6; rN=N; %rN =75
rQ = pdef(n);
rR = pdef(m)
rP = zeros(n,n,rN+1); % time goes from zero to N, so indices from 1 to N+1
rP(:,:,rN+1) = rQ; % we start by setting the final P value as Q
rK = zeros(m,n,rN+1);
Nrm = zeros(rN,1);
for i = rN:-1:1
% algebraic riccatti recursion eqn for P
rP(:,:,i) = rQ + A'*rP(:,:,i+1)*A -
A'*rP(:,:,i+1)*B*pinv(rR+B'*rP(:,:,i+1)*B)*B'*rP(:,:,i+1)*A;
% optimal K_t( time varying feedback) associated with each iteration
rK(:,:,i) = -pinv(rR+B'*rP(:,:,i+1)*B)*B'*rP(:,:,i+1)*A;
```

3(b). Numerical Scheme to solve Algebraic Riccatti Equation :



We see a clear of difference in the controlling patterns and the evolution of states over the time domain when comparing ARE response and Closed loop response.

Part 4 . Kalman Filter on closed loop linearised system:

4(a). Discretizing the Closed loop linearized system :

Considering the continuous time model

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{w}(t)$$
$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) + \mathbf{v}(t)$$

$$\mathbf{w}(t) \sim N(0, \mathbf{Q})$$
$$\mathbf{v}(t) \sim N(0, \mathbf{R})$$

where *v* and *w* are continuous zero-mean white noise sources with covariances

can be discretized, assuming zero-order hold for the input *u* and continuous integration for the noise *v*, to

$$\mathbf{x}[k+1] = \mathbf{A}_d\mathbf{x}[k] + \mathbf{B}_d\mathbf{u}[k] + \mathbf{w}[k]$$
$$\mathbf{y}[k] = \mathbf{C}_d\mathbf{x}[k] + \mathbf{D}_d\mathbf{u}[k] + \mathbf{v}[k]$$

with covariances

$$\mathbf{w}[k] \sim N(0, \mathbf{Q}_d)$$
$$\mathbf{v}[k] \sim N(0, \mathbf{R}_d)$$

where

$$\mathbf{A}_d = e^{\mathbf{A}T} = \mathcal{L}^{-1}\{(s\mathbf{I} - \mathbf{A})^{-1}\}_{t=T}$$
$$\mathbf{B}_d = \left(\int_{\tau=0}^{T} e^{\mathbf{A}\tau} d\tau\right)\mathbf{B} = \mathbf{A}^{-1}(\mathbf{A}_d - I)\mathbf{B},$$
$$\mathbf{C}_d = \mathbf{C}$$
$$\mathbf{D}_d = \mathbf{D}$$
$$\mathbf{Q}_d = \int_{\tau=0}^{T} e^{\mathbf{A}\tau}\mathbf{Q}e^{\mathbf{A}^\top\tau} d\tau$$
$$\mathbf{R}_d = \frac{1}{T}\mathbf{R}$$

$$x_{t+1} = A_{\mathrm{cl},d}\, x_t$$

Matlab:
```
D = zeros(6,2);
T = step; %(discretization time step)
sysc = ss(Acl,B,C,D);
sysd = c2d(sysc,T);
Ad = sysd.A;
Bd = sysd.B;
Cd = sysd.C;

dA = expm(Acl*T)
dB = inv(Acl)*(dA -eye(6))*B;
dC = C;
dD = zeros(6,2)
```

4(b). Generating noisy samples of the discrete time system :

$$\begin{aligned} x_{t+1} &= A_{\mathrm{cl},d}x_t + F_t w_t \\ y_t &= C_d x_t + H_t v_t \end{aligned}$$

We have :

$\mathbb{E}[w_t w_t^T] = W,$

$\mathbb{E}[v_t v_t^T] = V,$

$\mathbb{E}[x_0] = \bar{x}_0$ and

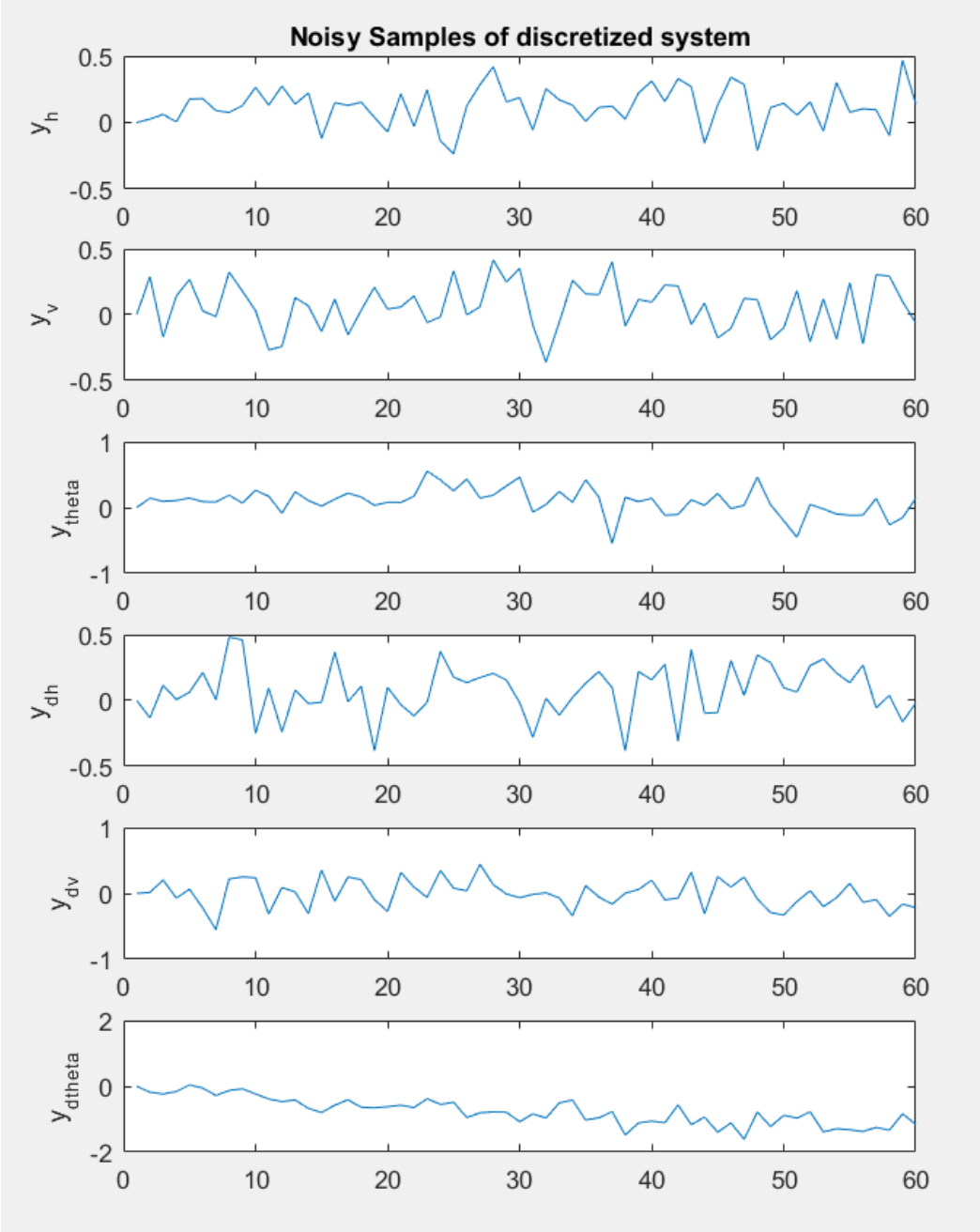
$\mathbb{E}[(x_0 - \bar{x}_0)(x_0 - \bar{x}_0)^T] = \Sigma_0$. Start with

$v_t \sim \mathcal{N}(0, 0.2I_{2\times 2})$ and $w_t \sim \mathcal{N}(0, 0.1I_{2\times 2})$, $\Sigma_0 = \mathrm{diag}(0.1, 0.1, 0.1, 0.1, 0.1, 0.1).$

Let F_t be the discretized control dynamics B_d and

$H_t = I.$

Matlab:
```
kx = zeros(6,N+1);
kx(:,1) = [0.1 0.1 0.1 0 0.1 0]';
for i = 2:N+1
    kw = normrnd(0,0.1,2,1);
    kv = normrnd(0,0.2,6,1);
kx(:,i) = Ad*kx(:,i-1) + F*kw;
kynoN(:,i) = dC*kx(:,i);
ky(:,i) = dC*kx(:,i)+ H*kv;
end

plot(ky(1,1:60))
hold on;
figure(12)
subplot(6,1,1)
plot(ky(1,1:60))
ylabel('y_h')
hold on;
title('Noisy Samples of discretized system')
subplot(6,1,2)
plot(ky(2,1:60))
ylabel('y_v')
hold on;
subplot(6,1,3)
plot(ky(3,1:60))
ylabel('y_t_h_e_t_a')
```


Noisy Samples of discretized system
y_h
y_v
y_theta
y_dh
y_dv
y_dtheta
0
10
20
30
40
50
60

4(c). Implementing the discrete time Kalman Filter:

Model and Observation:

$$\begin{aligned} \mathbf{x}_k &= \mathbf{A}_{k-1}\mathbf{x}_{k-1} + \mathbf{B}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \\ \mathbf{z}_k &= \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \end{aligned}$$

Initialization:

$$\mathbf{x}_0^a = \mu_0 \text{ with error covariance } \mathbf{P}_0$$

Model Forecast Step/Predictor:

$$\begin{aligned} \mathbf{x}_k^f &= \mathbf{A}_{k-1}\mathbf{x}_{k-1}^a + \mathbf{B}_{k-1}\mathbf{u}_{k-1} \\ \mathbf{P}_k^f &= \mathbf{A}_{k-1}\mathbf{P}_{k-1}\mathbf{A}_{k-1}^T + \mathbf{Q}_{k-1} \end{aligned}$$

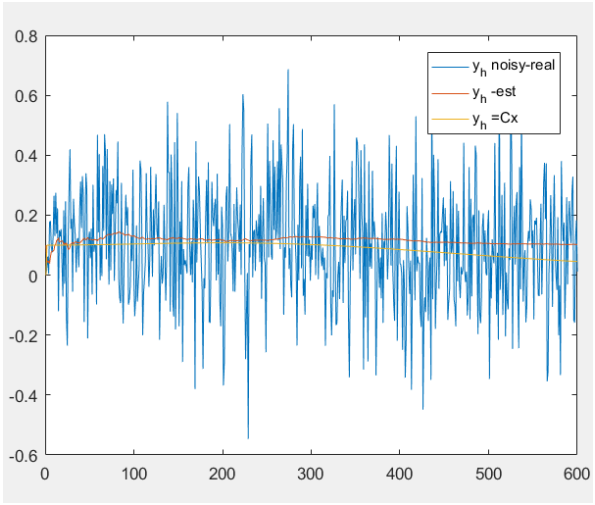
Data Assimilation Step/Corrector:

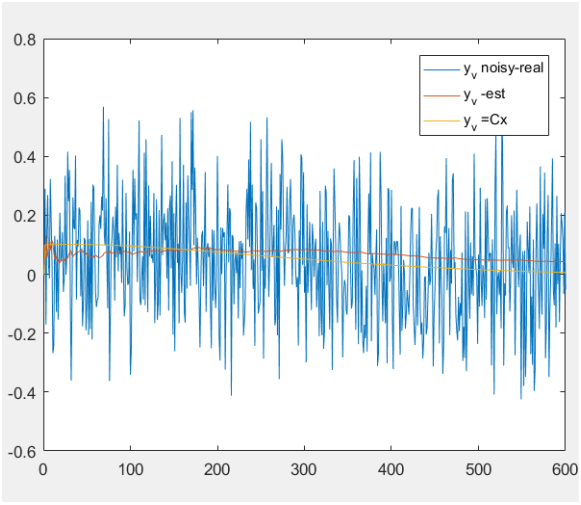
$$\begin{aligned} \mathbf{x}_k^a &= \mathbf{x}_k^f + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}_k\mathbf{x}_k^f) \\ \mathbf{K}_k &= \mathbf{P}_k^f\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k^f\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \\ \mathbf{P}_k &= (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_k^f \end{aligned}$$

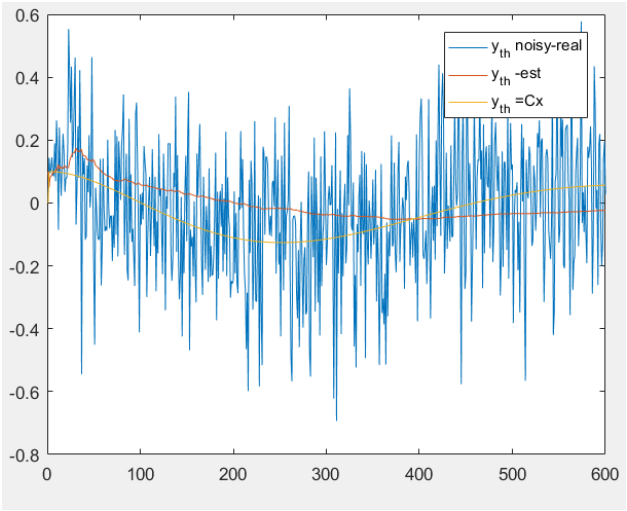
Matlab:

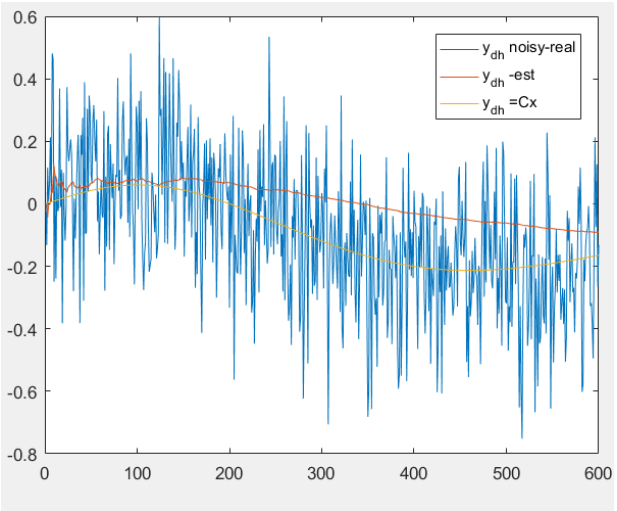
```
Q = [1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
     0     0     0     0     1     0
     0     0     0     0     0     1];  % process noise w

R = 0.1*eye(6); % measurement noise v
K = [];


xest_b(:,1) =kx0; %before
xest_c(:,1)= kx0; %current

Pest_b(:,:,1) = ksigma;
Pest_c(:,:,1) = ksigma;

for n=1:rN

xest_b(:,n+1) = Ad*xest_c(:,n);%+dB*u; %estimate with data before current
Pest_b(:,:,n+1) = Ad*Pest_c(:,:,n)*Ad' + Q; % estimate with data before current

K(:,:,n) = Pest_c(:,:,n)*(dC'*inv(dC*Pest_c(:,:,n)*dC' + R));

xest_c(:,n+1) = xest_c(:,n)+(K(:,:,n)*(ky(:,n)-dC*xest_c(:,n)));
Pest_c(:,:,n+1) = (eye(6)- K(:,:,n)*C)*Pest_c(:,:,n);
yest(:,n) = dC*xest_c(:,n);

end
```

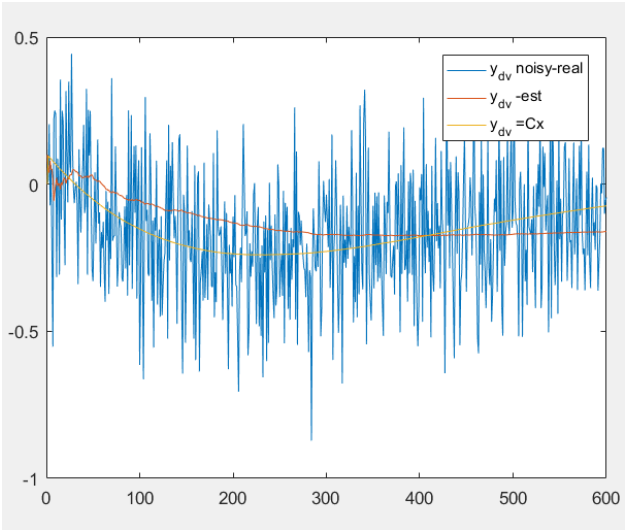
4(d). Applying the discrete time Kalman Filter to the system:

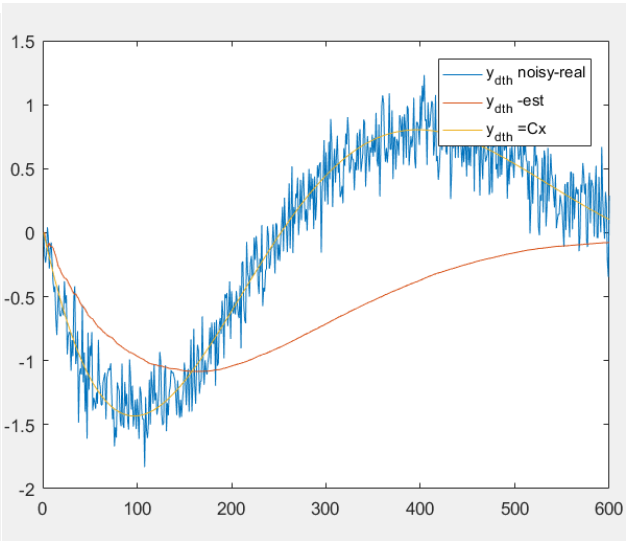












Bibliography :

- Autonomous navigation in complex indoor and outdoor environments with micro aerial vehicles Shen, Shaojie. University of Pennsylvania, ProQuest Dissertations Publishing, 2014. 3670964.
- Mellinger, D., Michael, N. and Kumar, V. (2012). Trajectory generation and control for precise aggressive maneuvers with quadrotors. The International Journal of Robotics Research, 31(5), pp.664-674.
- Minimum snap trajectory generation and control for quadrotors - IEEE Conference Publication.

- The GRASP Multiple Micro-UAV Testbed - IEEE Journals & Magazine.

- Design of small, safe and robust quadrotor swarms - IEEE Conference Publication.

-Dynamical systems theory: Nonlinear dynamics and chaosby Strogatz

-Sniedovich, M. (2010), Dynamic Programming: Foundations and Principles, Taylor & Francis, ISBN 978-0-8247-4099-3

-Stuart Dreyfus. "Richard Bellman on the birth of Dynamical Programming"

-O Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, ISBN 0-262-03293-7 . pp. 344.

Full Matlab Code :

```
clc, clear all, close all

g=9.81;m=1;I=1;

 x0 = [0;0;0;0;0;0]';
% x0 = [0.1 0.1 0.1 0 0.1 0]';
Qxcl(:,1) = x0;
N=10000; %number of time steps
t=linspace(0,10, N+1); %taking the time step
step = t(2)-t(1);
w=5;

Qx = zeros(6, N+1); %initial condition is set at x[:,0] to [0;0;0;0;0;0]
for i =1:N+1
   u(:,i)= [(m*g + sin(2*t(i)*pi*w));0];
end
% Simulation forward euler:
for i=2:N+1
    % defining the U vector
    Qx(:,i) = step*(fun(i,Qx(:,i-1),[(m*g + sin(2*t(i)*pi*w));0]))+ Qx(:,i-1);
end

% % ode 45 verification
% [tq,q]=ode45(@(tq,q)fun(tq,q),t,Qx(:,1));
% q=q';

% plot(q(2,:))
```

```
% hold on;
% plot(q(5,:))

% figure(1)
% hold on
% plot(q(1,:))
% hold on;
% plot(q(2,:))
% hold on;
% plot(q(5,:))

figure(2)

subplot(3,1,1)
plot(Qx(1,1:5000))
ylabel('h')
hold on;
title('Forward Euler Simulation with w=5')
subplot(3,1,2)
plot(Qx(2,1:5000))
hold on;
ylabel('v')
subplot(3,1,3)
plot(Qx(3,1:5000))
ylabel('theta')
xlabel('time steps');

figure(3)
subplot(3,1,1)
plot(Qx(4,1:5000))
ylabel('dh')
hold on;
subplot(3,1,2)
plot(Qx(5,1:5000))
hold on;
ylabel('dv')
subplot(3,1,3)
plot(Qx(6,1:5000))
ylabel('dtheta')
xlabel('time steps');

figure(4)
subplot(2,1,1)
plot(u(1,1:5000))
hold on;
ylabel('u1')
subplot(2,1,2)
plot(u(2,1:5000))
ylabel('u2')
xlabel('time steps');


%% Part 2 : Stabilisation:

%Linearised Matrices

A = [0 0 0 1 0 0;
    0 0 0 0 1 0 ;
    0 0 0 0 0 1;
    0 0 9.8 0 0 0;
    0 0 0 0 0 0;
    0 0 0 0 0 0]
```

```
B = [0 0;
    0 0;
    0 0;
    0 0;
    1 0;
    0 1;]

CTRB = [B A*B A^2*B A^3*B A^4*B A^5*B]

C = [1,0,0,0,0,0;
    0,1,0,0,0,0;
    0,0,1,0,0,0;
    0,0,0,1,0,0;
    0,0,0,0,1,0;
    0,0,0,0,0,1]



            % Ans : CTRB =
            %
            %       [  0   0   0   0   0   0   0  9.8  0   0   0   0
            %          0   0   1   0   0   0   0   0   0   0   0   0
            %          0   0   0   1   0   0   0   0   0   0   0   0
            %          0   0   0   0   0  9.8  0   0   0   0   0   0
            %          1   0   0   0   0   0   0   0   0   0   0   0
            %          0   1   0   0   0   0   0   0   0   0   0   0  ]


rank(CTRB)
% Ans : 6; verified with rank(ctrb(A,B))

% thus Rank = 6;

% ---> 2(c)(1)
D=eig(A);
lambda = rand(10);
lambda= lambda(1,:);
for k=1:length(lambda)
rtest(k)=rank(ctrb(-lambda(k)*eye(6) - A,B));
end
% satisfies the controllability condition

% ---> 2(c)(2)

p = [-3.5; -3.6; -3.7; -3.8; -4.5; -4.9]; % be the mu values (the shift in poles)
K0 = place(A,B,p); % K0 from matlab solver


%(-muI-A)W + W(-muI-A)' + BB' =  lyapunov
        lA = (diag(p) - A);
        lQ = B*B';
        W = lyap(lA,lQ);

% from the calculation from 2(C)(3) we know that inv(P) = W; sp P = inv(W)

        lP = inv(W);

        K0= 0.5*B'*lP ;

%checking controllability
        eig(A-B*K0)
                % ans = yes controllable!
                %
```

```
%   -4.0094 + 8.9135i
%   -4.0094 - 8.9135i
%   -3.9406 + 0.8051i
%   -3.9406 - 0.8051i
%   -4.0500 + 3.5718i
%   -4.0500 - 3.5718i

% closed Loop A
Acl = A-B*K0;

%% -----> 2(d) Simulation of closed loop dynamics from a small perturbation
Qxlcl = zeros(6, N+1);
Qxcl = zeros(6, N+1); %initial condition is set at x[:,0] to [0;0;0;0;0;0]
Qxcl(:,1) = [0.1 0.1 0.1 0 0.1 0]';
% Qxcl(:,1) = [0.2 0.4 0.3 0.1 0.2 0.1]';
Qxlcl(:,1) = Qxcl(:,1);
unl = zeros(2, N+1);
% Simulation forward euler:
for i=2:N+1
    % defining the U vector
    unl(:,i) = -K0*Qxcl(:,i-1)+ [9.8;0];
     Qxcl(:,i) = step*(fun(i,Qx(:,i-1),unl(:,i)))+ Qx(:,i-1);
 Qxlcl(:,i) = step*(clloop(t(i),Qxlcl(:,i-1)))+ Qxlcl(:,i-1);
end

% ode 45 verification
% [tq,qcl]=ode45(@(tq,qcl)clloop(tq,qcl),t,Qxcl(:,1));
% qcl=qcl';
%
% plot(qcl(2,:))
% hold on;
% plot(qcl(5,:))

figure(3)
hold on
plot(Qxlcl(1,1:2000))
hold on;
plot(Qxlcl(2,1:2000))
hold on;
plot(Qxlcl(3,1:2000))
hold on;
plot(Qxlcl(4,1:2000))
hold on;
plot(Qxlcl(5,1:2000))
hold on;
plot(Qxlcl(6,1:2000))
title('closed loop response of linearised system to small perturbation');
xlabel('time step')
legend ('h','v','th','dh','dv','dth')

% figure(4)
%
subplot(3,1,1)
plot(Qx(1,1:5000))
ylabel('h')
hold on;
title('closed loop response of non linear system to small perturbation')
subplot(3,1,2)
plot(Qx(2,1:5000))
hold on;
ylabel('v')
subplot(3,1,3)
plot(Qx(3,1:5000))
```

```
ylabel('theta')
xlabel('time steps');

figure(5)
subplot(3,1,1)
plot(Qx(4,1:5000))
ylabel('dh')
hold on;
subplot(3,1,2)
plot(Qx(5,1:5000))
hold on;
ylabel('dv')
subplot(3,1,3)
plot(Qx(6,1:5000))
ylabel('dtheta')
xlabel('time steps');

figure(6)
subplot(2,1,1)
plot(u(1,1:5000))
hold on;
ylabel('u1')
subplot(2,1,2)
plot(u(2,1:5000))
ylabel('u2')
xlabel('time steps');
%% 3 (b) -> optimal control usin ARE

m=2; n=6; rN=N; %rN =75

rQ = pdef(n);
rR = pdef(m)


rP = zeros(n,n,rN+1); % time goes from zero to N, so indices from 1 to N+1
rP(:,:,rN+1) = rQ; % we start by setting the final P value as Q
rK = zeros(m,n,rN+1);
Nrm = zeros(rN,1);
for i = rN:-1:1

% algebraic riccatti recursion eqn for P
rP(:,:,i) = rQ + A'*rP(:,:,i+1)*A -
A'*rP(:,:,i+1)*B*pinv(rR+B'*rP(:,:,i+1)*B)*B'*rP(:,:,i+1)*A;

% optimal K_t( time varying feedback) associated with each iteration
rK(:,:,i) = -pinv(rR+B'*rP(:,:,i+1)*B)*B'*rP(:,:,i+1)*A;

%norm gives a scalar value for the P matrix. (2norm is used here to show
%convergence
Nrm(rN+1-i) = norm(rP(:,:,i+1)-rP(:,:,i))/norm(rP(:,:,i+1));
end

%showing convergence of P
figure(9)
plot (Nrm(1:15))
ylabel('norm(p_i_+_1 - p_i) / norm(p_i_+_1)')
xlabel('iterations')


for i=1:length(rP)
    Z(i) = norm(rP(:,:,i));
end
```

```
%%  optimal time varying
rKb = reshape(permute(rK, [2 1 3]), size(rK, 2), [])'
rk1 = zeros(76,6);
rk2 = zeros(76,6);
s=0;t=0;
for i=1:length(rKb)
    if(mod(i,2)==0)
        s=s+1;
        rk2(s,:)=rKb(i,:);
    else
        t=t+1;
        rk1(t,:)=rKb(i,:);
    end
end



figure(7)
subplot(2,1,1)
plot(rk1(:,1))
ylabel('K1')
hold on;

subplot(2,1,2)
plot(rk2(:,1))
ylabel('K2')
hold on;


%% finding initial condition x0, with norm not exceeding one, that maximises value of
J


% [v,d] = eig(rP(:,:,1))

%maximizing a quadratic form subject to ?x0?=1?x0?=1 constraints for a symmetric
matrix corresponds to a special matrix norm operator whose value is the maximum
eigenvalue and the maximizer is the associated eigenvector.
%
% rx0 = v(:,1);

%norm_x0 = norm(x0)
%
% creating x vector

rx0 = [0.1 0.1 0.1 0 0.1 0]'  ;
rx = zeros(n,rN+1);
rx(:,1) = rx0;

ru = zeros(m,rN+1);
ru(:,1) = [rk1(1,:);rk2(1,:)] * rx(:,1);

% %% 2( Simulating system from x0 with u0 (normal state space simulation)
%
for (i=1:rN)
    rx(:,i+1) = A*rx(:,i) + B*ru(:,i);
%     u(:,i+1) = Kb(:,i+1)'*x(:,i+1); % u* calculated from Kb that we calculated
earlier
    ru(:,i+1) = [rk1(i+1,:);rk2(i+1,:)] * rx(:,i+1);
end

figure(3)
subplot(7,1,1)
```

```
plot(ru(1,1:60))
hold on;
title('optimal control using ARE')
plot(ru(2,1:60))
legend('u1','u2');
ylabel('u*')
hold on;

subplot(7,1,2)
plot(rx(1,1:60))
ylabel('h')
hold on;

subplot(7,1,3)
plot(rx(2,1:60))
ylabel('v')
hold on;

subplot(7,1,4)
plot(rx(3,1:60))
ylabel('theta')
hold on;

subplot(7,1,5)
plot(rx(4,1:60))
ylabel('dh')
hold on;

subplot(7,1,6)
plot(rx(5,1:60))
ylabel('dv')
hold on;

subplot(7,1,7)
plot(rx(3,1:60))
ylabel('dtheta')
xlabel('time')
hold on;

%
%% on nonlinear quadcopter - check if required

 % rn number of time steps
% t=linspace(0,100, rN+1); %taking the time step
% step = t(2)-t(1);
w=1;


ru1 = ru+[(1*g ) 0;0 0]*ones(2,rN+1);


Qx = zeros(6, rN+1); %initial condition is set at x[:,0] to [0;0;0;0;0;0]
Qx(:,1)=[0.1 0.1 0.1 0 0.1 0]';
% Simulation forward euler:
for i=2:rN+1

    % defining the U vector

     Qx(:,i) = step*(fun(i,Qx(:,i-1),ru1))+ Qx(:,i-1);
end

figure(9)
hold on
```

```
plot(Qx(1,1:80))
hold on;
plot(Qx(2,1:80))
hold on;
plot(Qx(5,1:80))


%% Part 4 Kalman Filter on closed loop system
%N=100;

%----> 4(a) Discretisation

D = zeros(6,2);
T = step; %(discretization time step)
sysc = ss(Acl,B,C,D);
sysd = c2d(sysc,T);
Ad = sysd.A;
Bd = sysd.B;
Cd = sysd.C;

dA = expm(Acl*T)
dB = inv(Acl)*(dA -eye(6))*B;
dC = C;
dD = zeros(6,2)


%----> 4(b) generating noisy samples of discrete system

ksigma = .1*eye(6);
% kw = normrnd(0,0.1,N+1,1);
% kv = normrnd(0,0.2,N+1,1);

% kw1 = zeros(2,N+1);
% kv1 = zeros(6,N+1);

H = eye(6);
F = dB;
% for i=1:N+1
%     kw1(:,i) = kw(i)*[1;1];
%     kv1(:,i) = kv(i)*[1;1;1;1;1;1];
% end

%Nonlinear Simulation
kx0 = [0.1 0.1 0.1 0 0.1 0]';


kx_real = zeros(6,N+1);
kx_real(:,1) = kx0;
for i = 2:N

kx_real(:,i) = Ad*kx_real(:,i-1) ;
end


figure(11)

plot(kx_real(1,1:60))
hold on;
% plot(kx_real(2,1:60))
% hold on;
% plot(kx_real(3,1:60))
% hold on;
% plot(kx_real(4,1:60))
```

```
% hold on;
% plot(kx_real(5,1:60))
% hold on;
% plot(kx_real(6,1:60))
% ylabel('x6')
% xlabel('time')

%% simulating noisy observation

kx = zeros(6,N+1);
kx(:,1) = [0.1 0.1 0.1 0 0.1 0]';
for i = 2:N+1
    kw = normrnd(0,0.1,2,1);
    kv = normrnd(0,0.2,6,1);
kx(:,i) = Ad*kx(:,i-1) + F*kw;
kynoN(:,i) = dC*kx(:,i);
ky(:,i) = dC*kx(:,i)+ H*kv;
end
%


plot(ky(1,1:60))
hold on;
figure(12)
subplot(6,1,1)
plot(ky(1,1:60))
ylabel('y_h')
hold on;
title('Noisy Samples of discretized system')
subplot(6,1,2)
plot(ky(2,1:60))
ylabel('y_v')
hold on;
subplot(6,1,3)
plot(ky(3,1:60))
ylabel('y_t_h_e_t_a')
hold on;
subplot(6,1,4)
plot(ky(4,1:60))
ylabel('y_d_h')
hold on;
subplot(6,1,5)
plot(ky(5,1:60))
ylabel('y_d_v')
hold on;
subplot(6,1,6)
plot(ky(6,1:60))
ylabel('y_d_t_h_e_t_a')

%% % kalman filter

Q = [1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
     0     0     0     0     1     0
     0     0     0     0     0     1];  % process noise w

R = 0.1*eye(6); % measurement noise v
K = [];


xest_b(:,1) =kx0; %before
```

```
xest_c(:,1)= kx0; %current

Pest_b(:,:,1) = ksigma;
Pest_c(:,:,1) = ksigma;

for n=1:rN

xest_b(:,n+1) = Ad*xest_c(:,n);%+dB*u; %estimate with data before current
Pest_b(:,:,n+1) = Ad*Pest_c(:,:,n)*Ad' + Q; % estimate with data before current

K(:,:,n) = Pest_c(:,:,n)*(dC'*inv(dC*Pest_c(:,:,n)*dC' + R));

xest_c(:,n+1) = xest_c(:,n)+(K(:,:,n)*(ky(:,n)-dC*xest_c(:,n)));
Pest_c(:,:,n+1) = (eye(6)- K(:,:,n)*C)*Pest_c(:,:,n);
yest(:,n) = dC*xest_c(:,n);

end
%
%
%  plot(1:100,ky(1,:),1:100,yest(1,:))
%    legend('true measurement','KF estimated measurement'), axis([0 100 -0.5
0.5]),xlabel('time'),ylabel('Position observations')
%

figure(20)

plot(ky(1,1:600));
hold on
plot (yest(1,1:600))
hold on
plot(kynoN(1,1:600))
legend('y_h noisy-real','y_h -est','y_h =Cx')

figure(21)
plot(ky(2,1:600));
hold on
plot (yest(2,1:600))
hold on
plot(kynoN(2,1:600))
legend('y_v noisy-real','y_v -est','y_v =Cx')

figure(22)
plot(ky(3,1:600));
hold on
plot (yest(3,1:600))
hold on
plot(kynoN(3,1:600))
legend('y_t_h noisy-real','y_t_h -est','y_t_h =Cx')

figure(23)
plot(ky(4,1:600));
hold on
plot (yest(4,1:600))
hold on
plot(kynoN(4,1:600))
legend('y_d_h noisy-real','y_d_h -est','y_d_h =Cx')


figure(24)
plot(ky(5,1:600));
hold on
plot (yest(5,1:600))
hold on
```

```
plot(kynoN(5,1:600))
legend('y_d_v noisy-real','y_d_v -est','y_d_v =Cx')

figure(25)
plot(ky(6,1:600));
hold on
plot (yest(6,1:600))

plot(kynoN(6,1:600))
legend('y_d_t_h noisy-real','y_d_t_h -est','y_d_t_h =Cx')

%% using psd to generate new kalman estimates

Q = pdef(6);
R = pdef(6);
K = [];


xest_b(:,1) =kx0; %before
xest_c(:,1)= kx0; %current

Pest_b(:,:,1) = ksigma;
Pest_c(:,:,1) = ksigma;

for n=1:rN

xest_b(:,n+1) = Ad*xest_c(:,n);%+dB*u; %estimate with data before current
Pest_b(:,:,n+1) = Ad*Pest_c(:,:,n)*Ad' + Q; % estimate with data before current

K(:,:,n) = Pest_c(:,:,n)*(dC'*inv(dC*Pest_c(:,:,n)*dC' + R));

xest_c(:,n+1) = xest_c(:,n)+(K(:,:,n)*(ky(:,n)-dC*xest_c(:,n)));
Pest_c(:,:,n+1) = (eye(6)- K(:,:,n)*C)*Pest_c(:,:,n);
yest(:,n) = dC*xest_c(:,n);

end

figure(26)

plot(ky(1,1:600));
hold on
plot (yest(1,1:600))
hold on
plot(kynoN(1,1:600))
legend('y_h noisy-real','y_h -est','y_h =Cx')

figure(27)
plot(ky(2,1:600));
hold on
plot (yest(2,1:600))
hold on
plot(kynoN(2,1:600))
legend('y_v noisy-real','y_v -est','y_v =Cx')

figure(28)
plot(ky(3,1:600));
hold on
plot (yest(3,1:600))
hold on
plot(kynoN(3,1:600))
legend('y_t_h noisy-real','y_t_h -est','y_t_h =Cx')

figure(29)
```

```
plot(ky(4,1:600));
hold on
plot (yest(4,1:600))
hold on
plot(kynoN(4,1:600))
legend('y_d_h noisy-real','y_d_h -est','y_d_h =Cx')


figure(30)
plot(ky(5,1:600));
hold on
plot (yest(5,1:600))
hold on
plot(kynoN(5,1:600))
legend('y_d_v noisy-real','y_d_v -est','y_d_v =Cx')

figure(31)
plot(ky(6,1:600));
hold on
plot (yest(6,1:600))
hold on
plot(kynoN(6,1:600))
legend('y_d_t_h noisy-real','y_d_t_h -est','y_d_t_h =Cx')
```
