```
% AE740 HW1 akshatdy
clc;
close all;
```

```
% 1.a equations in state variable form
```

$$\dot{x_1} = x_2$$

$$\dot{x_2} = -\frac{g}{l}sin(x_1 + \frac{u}{ml^2})$$

$$y = x_1$$

```
% 1.b (see the function below after all the script code)
```

```
% 1.c equilibrium at u = 20
ueq = 20;
xeq = findPendEq(ueq);
fprintf('1.c equilibrium at u = 20\n')
fprintf('xeq = [%f; %f]\n', xeq(1), xeq(2));

% 1.d symbolically linearized model at u = 20 and x = xeq
[A, B] = symLin(xeq, ueq);
fprintf('\n1.d symbolically linearized model at u = 20 and x = [%f; %f]\n',
xeq(1), xeq(2));
fprintf('A = \n[%f, %f;\n%f, %f]\n', A(1, 1), A(1, 2), A(2, 1), A(2, 2));
fprintf('B = \n[%f;\n%f]\n', B(1), B(2));

% 1.e numerically linearized model at u = 20 and x = xeq
[A, B] = cdLin(xeq, ueq);
fprintf('\n1.e numerically linearized model at u = 20 and x = [%f; %f]\n',
xeq(1), xeq(2));
fprintf('A = \n[%f, %f;\n%f, %f]\n', A(1, 1), A(1, 2), A(2, 1), A(2, 2));
fprintf('B = \n[%f;\n%f]\n', B(1), B(2));

% 1.f simulate over 10 seconds
odefun = @(t, x) pendModel(x, ueq + uIn(t));
tspan = [0:0.01:10];
x0 = [0; 0];
[Tode, Xode] = ode45(odefun, tspan, x0);
figure(1);
plot(Tode, Xode(:, 1));
title('1.f pendulum angle over time using ode45');
xlabel('time (s)');
ylabel('angle (rad)');

% 1.g simulate over 10 seconds using forward euler
figure(2);
title('1.g pendulum angle over time using forward euler');
xlabel('time (s)');
ylabel('angle (rad)');
hold on;
% 0.01 time step
```

```
[T, X] = eulerF(odefun, tspan, x0);
plot(T, X(:, 1));
% 0.1 time step
tspan = [0:0.1:10];
[T, X] = eulerF(odefun, tspan, x0);
plot(T, X(:, 1));
plot(Tode, Xode(:, 1));
legend('0.01 time step', '0.1 time step', 'ode45', 'Location','southwest');
hold off;
fprintf('\n1.g if we use a smaller time step, the accuracy of the model is
closer to what we get from ode45\n');

% 1.h simulate using model linearlized via central difference
[Acd, Bcd] = cdLin(xeq, ueq);
odeLinSim = @(t, x) Acd*x + Bcd*uIn(t);
tspan = [0:0.01:10];
x0 = [0; 0];
[T1, X1] = ode45(odeLinSim, tspan, x0(:)-xeq(:));
figure(3);
hold on;
title('1.h pendulum angle over time using central difference linearlized
model');
xlabel('time (s)');
ylabel('angle (rad)');
plot(T1, X1(:, 1)+xeq(1));
plot(Tode, Xode(:, 1));
legend('linearized', 'ode45', 'Location','southwest');
hold off;
fprintf('\n1.h the accuracy of the linearized model is good, it is very close
to the simulation done using ode45\n');

% functions need to be at the bottom??
% 1.b pendulum model
function xdot = pendModel(x, u)
    g = 9.81;
    m = 1;
    l = 10;
    xdot = [x(2); -g/l*sin(x(1)) + u/(m*l^2)];
end

% 1.c find equilibrium point
function xeq = findPendEq(ueq)
    xeq = fsolve(@(x) pendModel(x, ueq), [0; 0]);
end

% 1.d linearize pendulum model symbolically
function [A, B] = symLin(x0, u0)
    syms x1 x2 u
    x = [x1; x2];
    f = pendModel(x, u);
    A = jacobian(f, x);
    A = subs(A, [x; u], [x0; u0]);
    B = jacobian(f, u);
    B = subs(B, [x; u], [x0; u0]);
```

```
end

% 1.e linearize pendulum model numerically
function [A, B] = cdLin(x0, u0)
    epsilon = 1e-6;

    A = zeros(2, 2);
    for i = 1:2
        x = x0;
        x(i) = x(i) + epsilon;
        f1 = pendModel(x, u0);
        x = x0;
        x(i) = x(i) - epsilon;
        f2 = pendModel(x, u0);
        A(:, i) = (f1 - f2)/(2*epsilon);
    end

    B = zeros(2, 1);
    u = u0;
    u = u + epsilon;
    f1 = pendModel(x0, u);
    u = u0;
    u = u - epsilon;
    f2 = pendModel(x0, u);
    B = (f1 - f2)/(2*epsilon);
end

% 1.f simulate input over time
function u = uIn(t)
    u = 2*sin(t);
end

% 1.g simulate using forward euler
function [T, X] = eulerF(odefun, tspan, x0)
    T = tspan;
    time_step = tspan(2) - tspan(1);
    X = zeros(length(tspan), length(x0));
    X(1, :) = x0;
    for i = 2:length(tspan)
        % use the (') to transpose output from odefun so it is 1x2
        X(i, :) = X(i-1, :) + time_step*odefun(T(i-1), X(i-1, :))';
    end
end
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

1.c equilibrium at u = 20
xeq = [0.205313; 0.000000]
```

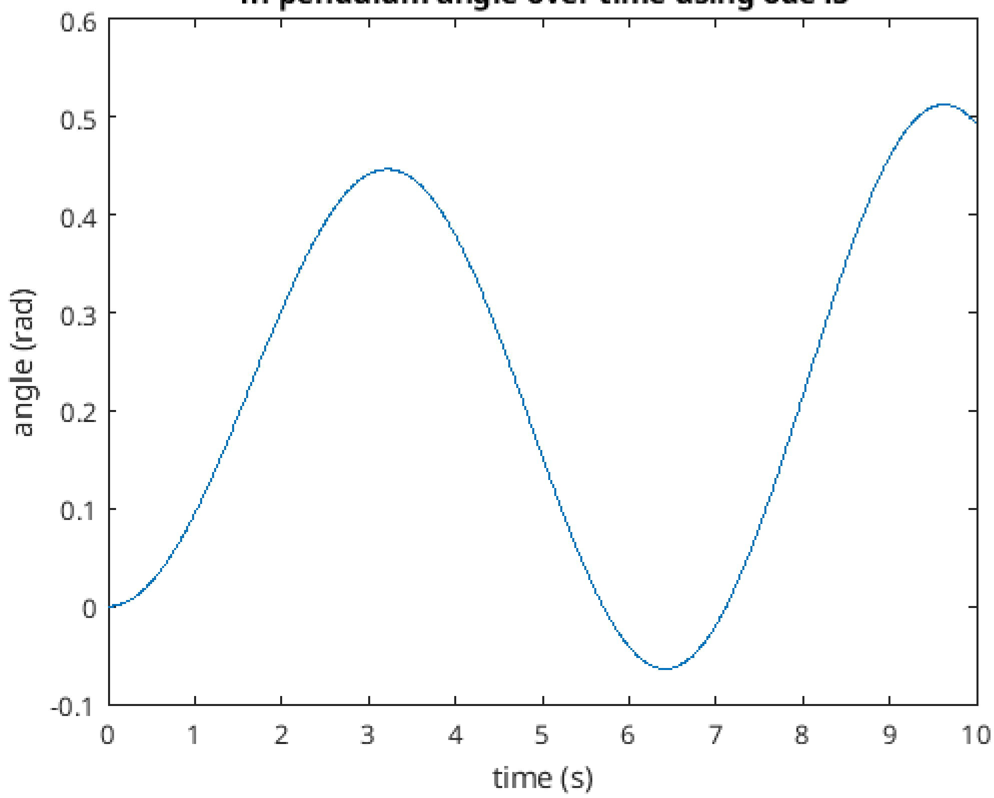
1.d symbolically linearized model at u = 20 and x = [0.205313; 0.000000]
A =
[0.000000, 1.000000;
-0.960396, 0.000000]
B =
[0.000000;
0.010000]

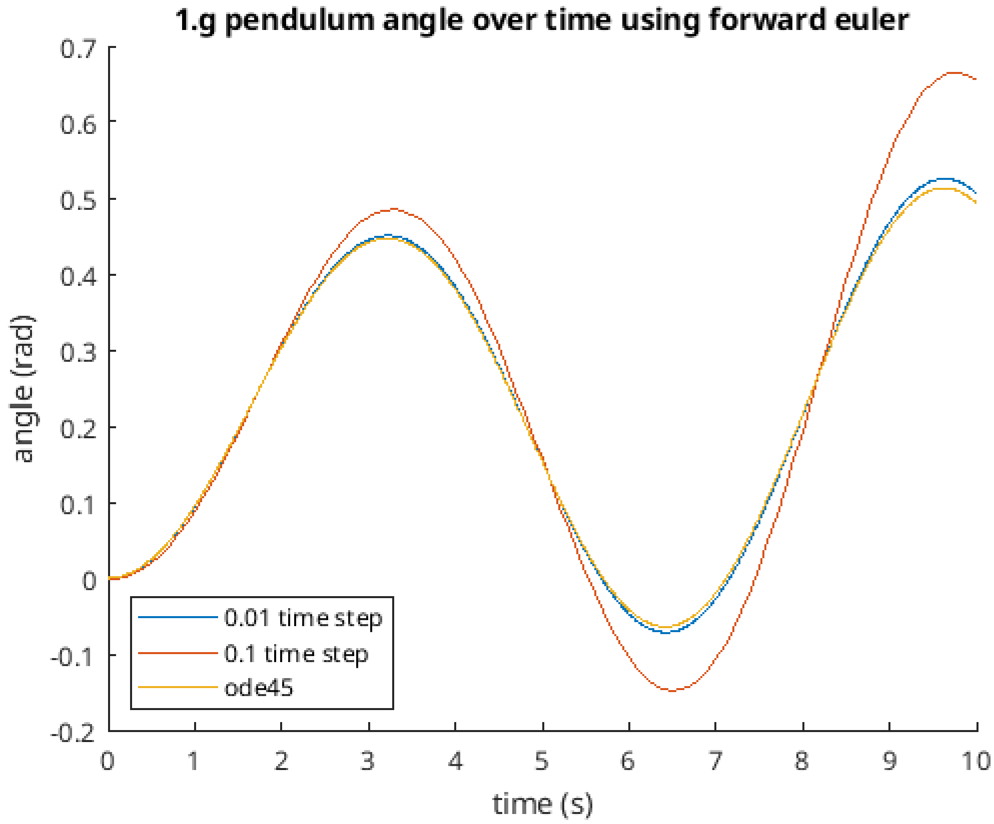
1.e numerically linearized model at u = 20 and x = [0.205313; 0.000000]
A =
[0.000000, 1.000000;
-0.960396, 0.000000]
B =
[0.000000;
0.010000]

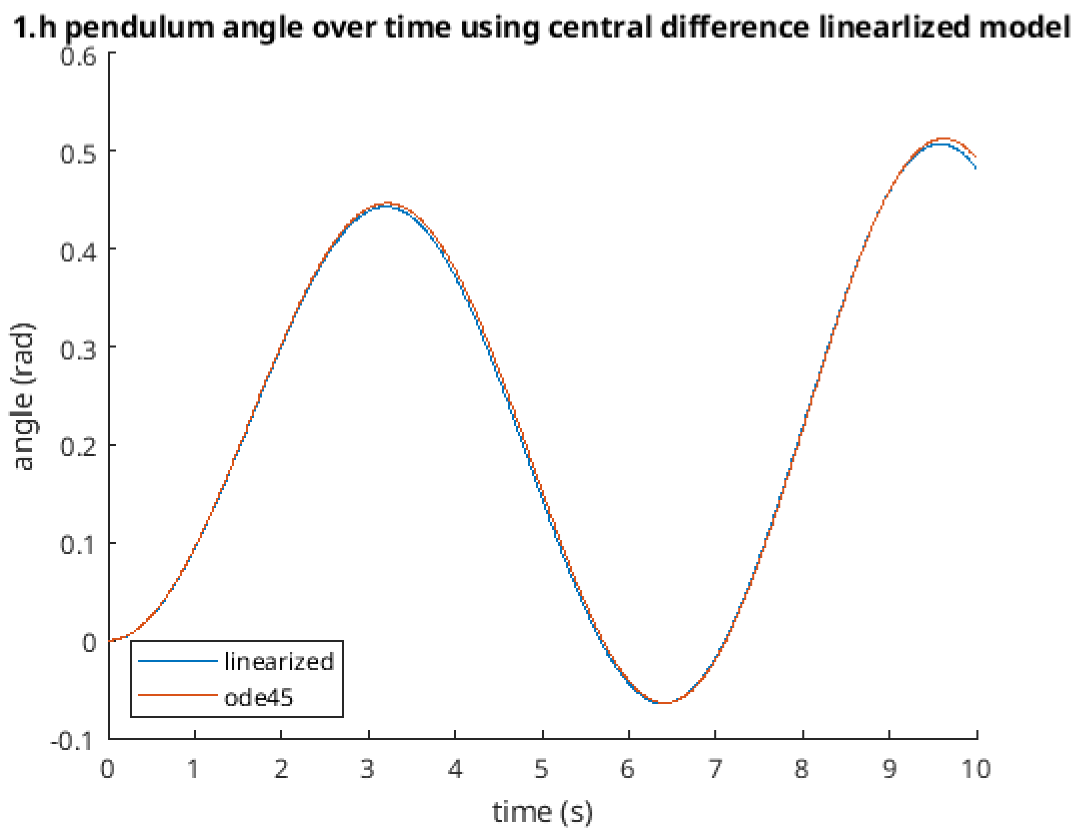
1.g if we use a smaller time step, the accuracy of the model is closer to what we get from ode45

1.h the accuracy of the linearized model is good, it is very close to the simulation done using ode45

1.f pendulum angle over time using ode45




1.g pendulum angle over time using forward euler
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
angle (rad)
0 1 2 3 4 5 6 7 8 9 10
time (s)
0.01 time step
0.1 time step
ode45


1.h pendulum angle over time using central difference linearlized model
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
angle (rad)
0 1 2 3 4 5 6 7 8 9 10
time (s)
linearized
ode45

Published with MATLAB® R2023b

```
% AE740 HW1 akshatdy
clc;
close all;
```

2

```
Ac = [
    0.0000, 1.0000, 0.0000, 0.0000, 0.0000;
    0.0000, -0.86939, 43.2230, -17.2510, -1.5766;
    0.0000, 0.99335, -1.3411, -0.16897, -0.25183;
    0.0000, 0.0000, 0.0000, -20.0000, 0.0000;
    0.0000, 0.0000, 0.0000, 0.0000, -20.0000;
];
Bc = [
    0.0000, 0.0000;
    0.0000, 0.0000;
    0.0000, 0.0000;
    20.0000, 0.0000;
    0.0000, 20.0000;
];
Cc = [
    1.0000, 0.0000, 0.0000, 0.0000, 0.0000;
    1.0000, 0.0000, -1.0000, 0.0000, 0.0000;
];
```

2.a check if model is open loop stable

```
eigAc = eig(Ac);
fprintf('2.a\nmodel is ');
% check if all eigenvalues are in left half plane
notLeftHalfPlane = eigAc(real(eigAc) >= 0);
if size(notLeftHalfPlane) ~= 0
    fprintf('open loop unstable, not all eigenvalues in left half plane:');
    display(notLeftHalfPlane);
else
    fprintf('open loop stable\n');
end
```

```
2.a
model is open loop unstable, not all eigenvalues in left half plane:
notLeftHalfPlane =

         0
    5.4515
```

2.b check if model is controllable

check if rank of controllability matrix is equal to dimension of x

```
fprintf('2.b\ncontrollability matrix rank: %d\n', rank(ctrb(Ac, Bc)));
fprintf('dimension of x: %d\n', size(Ac, 1));
fprintf('model is ');
if rank(ctrb(Ac, Bc)) == size(Ac, 1)
    fprintf('controllable\n');
else
    fprintf('not controllable\n');
end
fprintf('time horizon is infinitely small since this is a continuous time
system and input is unconstrained\n')
```

```
2.b
controllability matrix rank: 5
dimension of x: 5
model is controllable
time horizon is infinitely small since this is a continuous time system and
input is unconstrained
```

2.c convert model to discrete time

```
Ts = 0.01;
Dc = zeros(size(Cc, 1), size(Bc, 2));
[Ad, Bd, Cd, Dd] = c2dm(Ac, Bc, Cc, Dc, Ts, 'zoh');
fprintf('\n2.c');
display(Ad);
display(Bd);
```

```
2.c
Ad =

    1.0000    0.0100    0.0021   -0.0008   -0.0001
         0    0.9935    0.4278   -0.1561   -0.0147
         0    0.0098    0.9888   -0.0023   -0.0023
         0         0         0    0.8187         0
         0         0         0         0    0.8187


Bd =

   -0.0001   -0.0000
   -0.0161   -0.0015
   -0.0002   -0.0002
    0.1813         0
         0    0.1813
```

2.d convert to discrete time using expressions given in class

```
Ad2 = expm(Ac * Ts);
Bd2 = inv(Ac) * (Ad2 - eye(size(Ad2))) * Bc;
```

```
fprintf('\n2.d');
display(Ad2);
display(Bd2);
fprintf('Ac is a singular matrix, so it is not invertible. Hence we cannot
use it to find Bd using the expressions in class\n');
```

```
Warning: Matrix is singular to working precision.

2.d
Ad2 =

    1.0000    0.0100    0.0021   -0.0008   -0.0001
         0    0.9935    0.4278   -0.1561   -0.0147
         0    0.0098    0.9888   -0.0023   -0.0023
         0         0         0    0.8187         0
         0         0         0         0    0.8187


Bd2 =

   NaN   NaN
   NaN   NaN
   NaN   NaN
   NaN   NaN
   NaN   NaN

Ac is a singular matrix, so it is not invertible. Hence we cannot use it to
find Bd using the expressions in class
```

2.e check if discrete time model is open loop stable

```
eigAd = eig(Ad);
fprintf('\n2.e\ndiscrete time model is ');
% check if all eigenvalues are in unit circle
outsideUnitCircle = eigAd(abs(eigAd) >= 1);
if size(outsideUnitCircle) ~= 0
    fprintf('open loop unstable, eigenvalues outside unit circle:');
    display(outsideUnitCircle);
else
    fprintf('open loop stable\n');
end
```

```
2.e
discrete time model is open loop unstable, eigenvalues outside unit circle:
outsideUnitCircle =

    1.0000
    1.0560
```

2.f check if discrete time model is controllable

```
fprintf('\n2.f\ncontrollability matrix rank: %d\n', rank(ctrb(Ad, Bd)));
fprintf('dimension of x: %d\n', size(Ad, 1));
fprintf('discrete time model is ');
if rank(ctrb(Ad, Bd)) == size(Ad, 1)
    fprintf('controllable\n');
else
    fprintf('not controllable\n');
end
fprintf('since n=%d, the time horizon is n*Ts=%.2fs\n', size(Ad, 1), size(Ad,
1) * Ts);
```

```
2.f
controllability matrix rank: 5
dimension of x: 5
discrete time model is controllable
since n=5, the time horizon is n*Ts=0.05s
```

2.g check if continuous time model is closed-loop stable

```
Fc = [
    -2.8900, 0.7780;
    1.9800, 3.3400;
];
Kc = [
    2.1100, 0.8906, 4.9107, -0.5343, -0.1009;
    -5.3200, -0.8980, -4.6618, 0.4280, 0.1099;
];

fprintf('\n2.g\n');
```

```
2.g
```

$\dot{x_c} = A_c x_c + B_c u_c$ where $u_c = F_c r + K_c x_c$

$\dot{x_c} = A_c x_c + B_c(F_c r + K_c x_c)$

$\dot{x_c} = A_c x_c + B_c F_c r + B_c K_c x_c$

$\dot{x_c} = (A_c + B_c K_c) x_c + B_c F_c r$

check matrix $(A_c + B_c K_c)$

```
eigAcBcKc = eig(Ac + Bc * Kc);
notLeftHalfPlane = eigAcBcKc(real(eigAcBcKc) >= 0);
fprintf('system with feedforward and feedback is open loop ')
if size(notLeftHalfPlane) ~= 0
    fprintf('unstable, eigenvalues not in left half plane:');
```

```
    display(notLeftHalfPlane);
else
    fprintf('stable, all eigenvalues lie in the left half plane\n');
end
```

system with feedforward and feedback is open loop stable, all eigenvalues lie in the left half plane

to get steady state gain, we set $\dot{x}_c = 0$

$$0 = (A_c + B_c K_c) x_c + B_c F_c r$$

$$x_c = -(A_c + B_c K_c)^{-1} B_c F_c r$$

$$y_c = C_c x_c$$

$$y_c = C_c(-(A_c + B_c K_c)^{-1} B_c F_c r)$$

if $y_c = Hr$, then

$$H = -C_c(A_c + B_c K_c)^{-1} B_c F_c$$

```
H = - Cc * inv(Ac + Bc * Kc) * Bc * Fc;
fprintf('steady state gain H:\n');
display(H);
```

```
steady state gain H:

H =

    1.0026   -0.0020
    0.0004    0.9992
```

2.h continuous time closed-loop simulation for 5 seconds

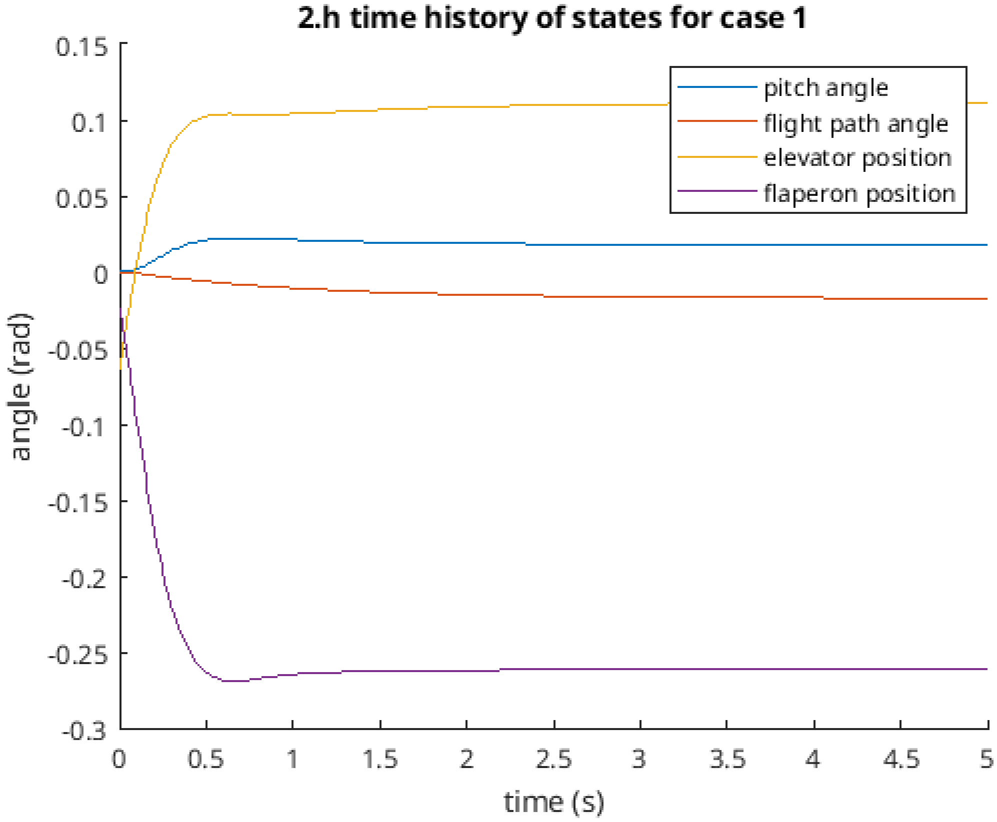
```
fprintf('\n2.h\n');
tspan = [0 5];
x0 = zeros(size(Ac, 1), 1);
```

2.h

case 1

```
r = [0.01745; -0.01745];
odefun = @(t, x) f16modelCont(x, uSaturated(r, x, Fc, Kc), Ac, Bc);
[T, X] = ode45(odefun, tspan, x0);
Y = (Cc * X')';
U = uSaturated(r, X', Fc, Kc)';
figure(1);
```

```
hold on;
plot(T, Y);
plot(T, U);
title('2.h time history of states for case 1');
xlabel('time (s)');
ylabel('angle (rad)');
legend('pitch angle', 'flight path angle', 'elevator position', 'flaperon
position');
hold off;

fprintf('Case 1: the closed loop system is stable, ');
fprintf('because the feedforward input is small enough ');
fprintf('to not saturate the total input to the system\n');
```



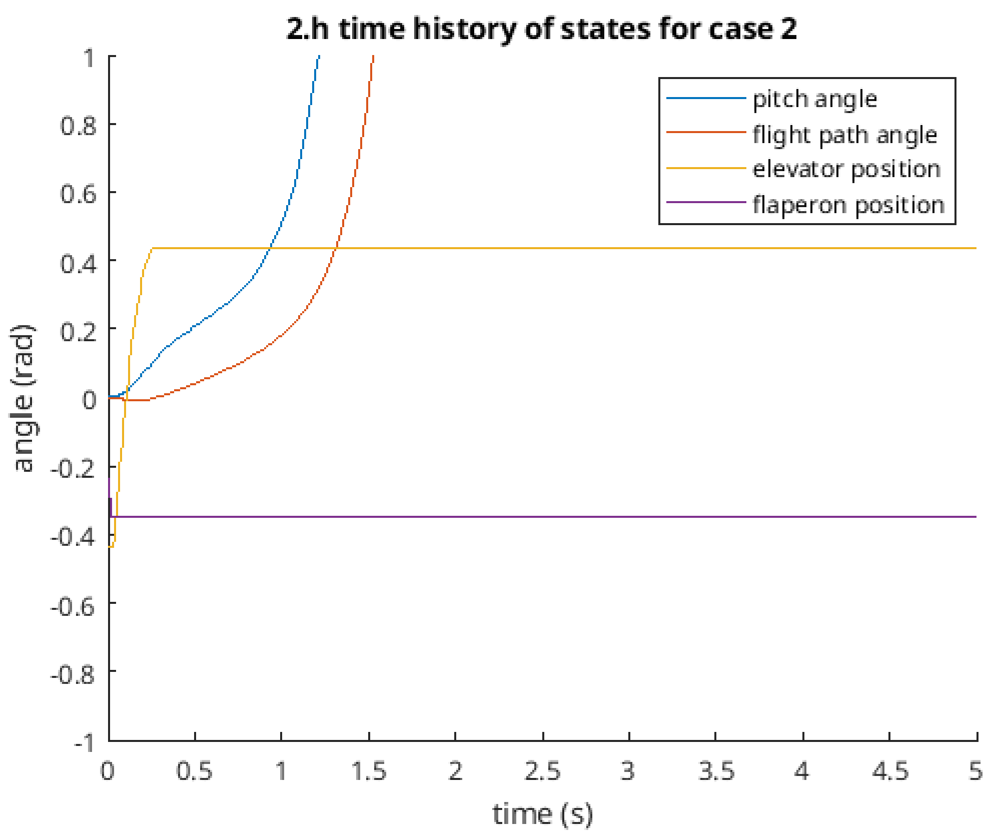
Case 1: the closed loop system is stable, because the feedforward input is small enough to not saturate the total input to the system

case 2

```
r = [0.1745; -0.1745];
odefun = @(t, x) f16modelCont(x, uSaturated(r, x, Fc, Kc), Ac, Bc);
[T, X] = ode45(odefun, tspan, x0);
Y = (Cc * X')';
U = uSaturated(r, X', Fc, Kc)';
figure(2);
hold on;
```

```
plot(T, Y);
plot(T, U);
title('2.h time history of states for case 2');
xlabel('time (s)');
ylabel('angle (rad)');
ylim([-1 1]);
legend('pitch angle', 'flight path angle', 'elevator position', 'flaperon
position');
hold off;

fprintf('Case 2: the closed loop system is not stable, ');
fprintf('because the feedforward inputs saturate the overall input, ');
fprintf('and the system essentially has no way to stabilize itself\n');
```

Case 2: the closed loop system is not stable, because the feedforward inputs saturate the overall input, and the system essentially has no way to stabilize itself



2.i check if discrete time closed-loop model is stable

```
eigAdBdKc = eig(Ad + Bd * Kc);
outsideUnitCircle = eigAdBdKc(abs(eigAdBdKc) >= 1);
fprintf('\n2.i\ndiscrete time closed loop system is ');
if size(outsideUnitCircle) ~= 0
```

```
    fprintf('unstable, eigenvalues outside unit circle:');
    display(outsideUnitCircle);
else
    fprintf('stable, all eigenvalues lie in the unit circle\n');
end
```

```
2.i
discrete time closed loop system is stable, all eigenvalues lie in the unit
circle
```

2.j check if discrete time closed-loop model is stable with Ts=0.5s

```
Ts5 = 0.5;
[Ad5, Bd5, Cd5, Dd5] = c2dm(Ac, Bc, Cc, Dc, Ts5, 'zoh');
eigAdBdKc = eig(Ad5 + Bd5 * Kc);
outsideUnitCircle = eigAdBdKc(abs(eigAdBdKc) >= 1);
fprintf('\n2.j with Ts=0.5, discrete time closed-loop model is ');
if size(outsideUnitCircle) ~= 0
    fprintf('unstable, eigenvalues outside unit circle:');
    display(outsideUnitCircle);
else
    fprintf('stable, all eigenvalues lie in the unit circle\n');
end
```

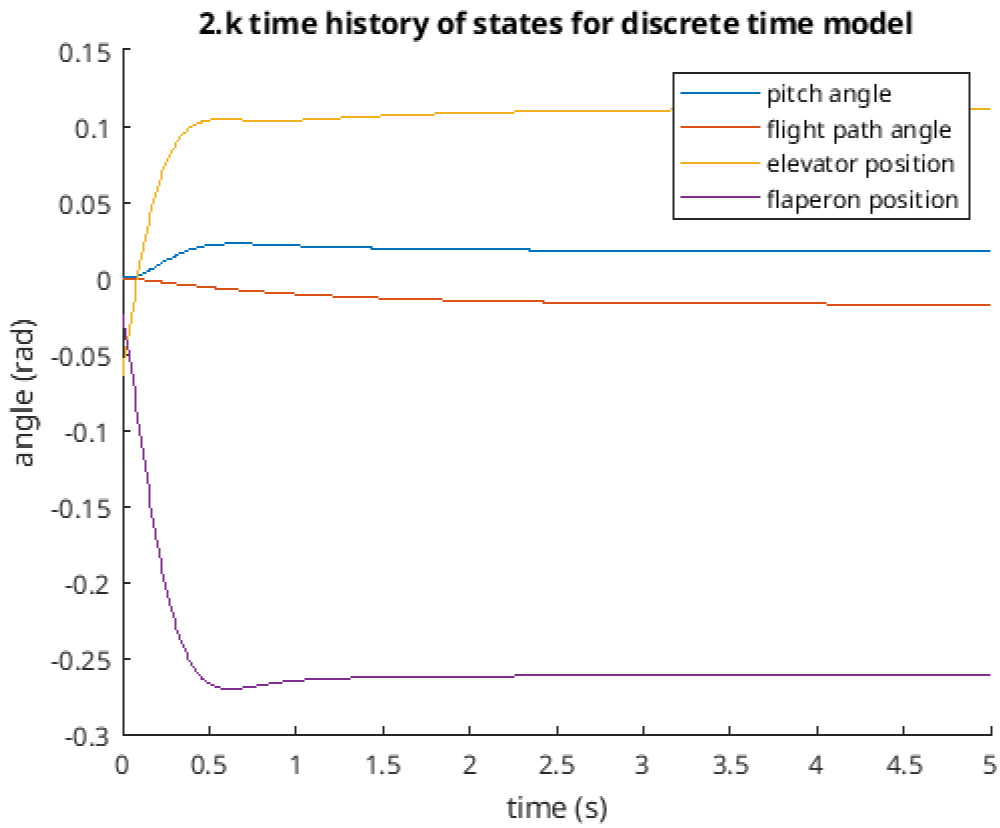
```
2.j with Ts=0.5, discrete time closed-loop model is unstable, eigenvalues
outside unit circle:
outsideUnitCircle =

  -14.1622
```

2.k discrete time closed-loop simulation for 5 seconds

```
fprintf('\n2.k\n');
tspan = [0:Ts:5];

x0 = zeros(size(Ad, 1), 1);
% case 1
r = [0.01745; -0.01745];
X = zeros(length(tspan), length(x0));
X(1, :) = x0;
for t=2:length(tspan)
    X(t, :) = f16modelDisc(X(t-1, :)', r, Ad, Bd, Kc, Fc)';
end
Y = (Cd * X')';
U = uSaturated(r, X', Fc, Kc)';
figure(3);
```

```
hold on;
plot(tspan, Y);
plot(tspan, U);
title('2.k time history of states for discrete time model');
xlabel('time (s)');
ylabel('angle (rad)');
legend('pitch angle', 'flight path angle', 'elevator position', 'flaperon
position');
hold off;

fprintf('The discrete time closed loop system is stable');
```

```
2.k
The discrete time closed loop system is stable
```



functions for all the parts, needs to be at the end

```
% 2.h
function xdot = f16modelCont(x, u, A, B)
    xdot = A*x + B*u;
end

function u = uSaturated(r, x, Fc, Kc)
```

```
    uMax = [0.4363; 0.3491];
    uMin = -uMax;
    u = Fc * r + Kc * x;
    u = max(uMin, min(uMax, u));
end

% 2.k
function xnext = f16modelDisc(x, r, Ad, Bd, Kc, Fc)
    xnext = Ad*x + Bd*uSaturated(r, x, Fc, Kc);
end
```

Case 1: the closed loop system is stable, because the feedforward input is small enough to not saturate the total input to the system

?

Published with MATLAB® R2023b

```
% AE740 HW1 akshatdy
clc;
close all;
```

3

```
n = 0.00114;
Ac = [
    0 0    0    1    0 0;
    0 0    0    0    1 0;
    0 0 0 0 0 1;
    3*n^2 0    0    0    2*n 0;
    0 0    0    -2*n 0 0;
    0 0    -n^2 0    0 0;
];
Cc = [
    0.5 0.5 0 0 0 0;
    0 0 1 0 0 0;
];
```

3.a get Ad, Cd

```
Ts = 30;
[Ad, Bd, Cd, Dd] = c2dm(Ac, [], Cc, [], Ts, 'zoh');
fprintf('3.a\n');
display(Ad);
display(Cd);
```

```
3.a

Ad =

    1.0018         0         0   29.9942    1.0259         0
   -0.0000    1.0000         0   -1.0259   29.9766         0
         0         0    0.9994         0         0   29.9942
    0.0001         0         0    0.9994    0.0684         0
   -0.0000         0         0   -0.0684    0.9977         0
         0         0   -0.0000         0         0    0.9994


Cd =

    0.5000    0.5000         0         0         0         0
         0         0    1.0000         0         0         0
```

3.b check if model is observable

```
fprintf('3.b\nobservability matrix rank: %d\n', rank(obsv(Ad, Cd)));
fprintf('number of states: %d\n', size(Ad, 1));
```

```
if rank(obsv(Ad, Cd)) == size(Ad, 1)
    fprintf('system is observable\n');
else
    fprintf('system is not observable\n');
end
```

```
3.b
observability matrix rank: 6
number of states: 6
system is observable
```

3.c design Luenberger observer

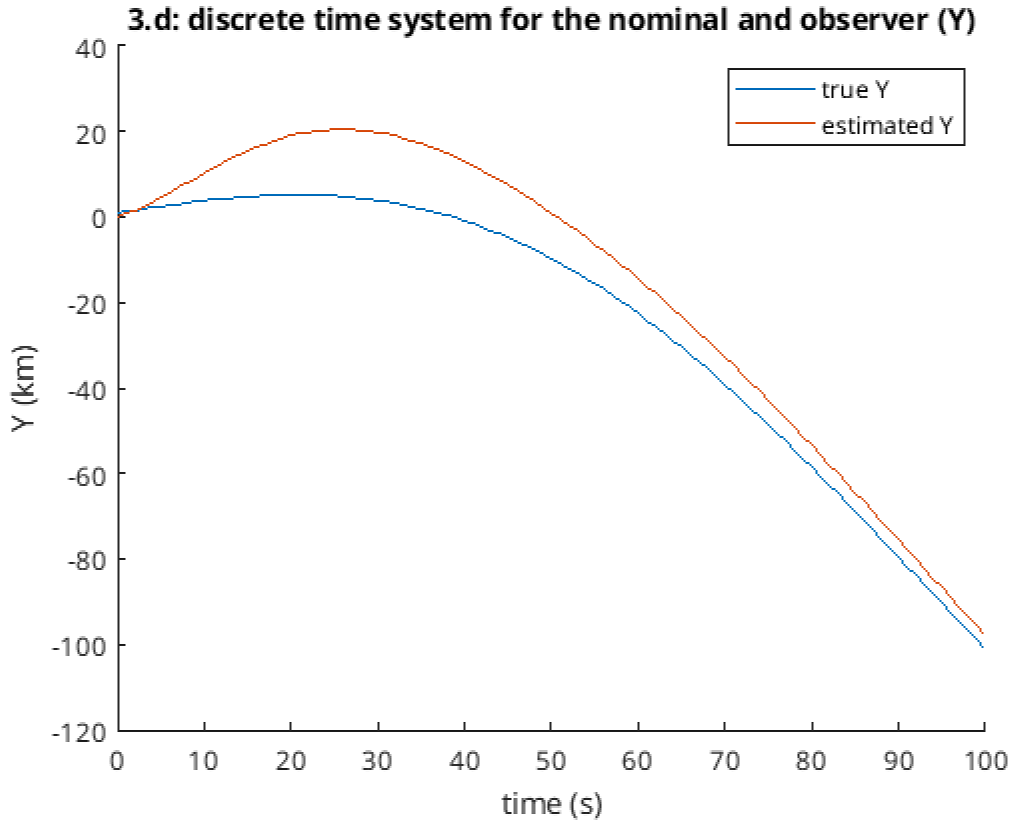
```
E = [
    0.9712;
    0.9377;
    0.9348 + 0.0686i;
    0.9348 - 0.0686i;
    0.9471 + 0.0753i;
    0.9471 - 0.0753i;
];
Ld = -place(Ad', Cd', E)';
fprintf('3.c\n');
display(Ld);
```

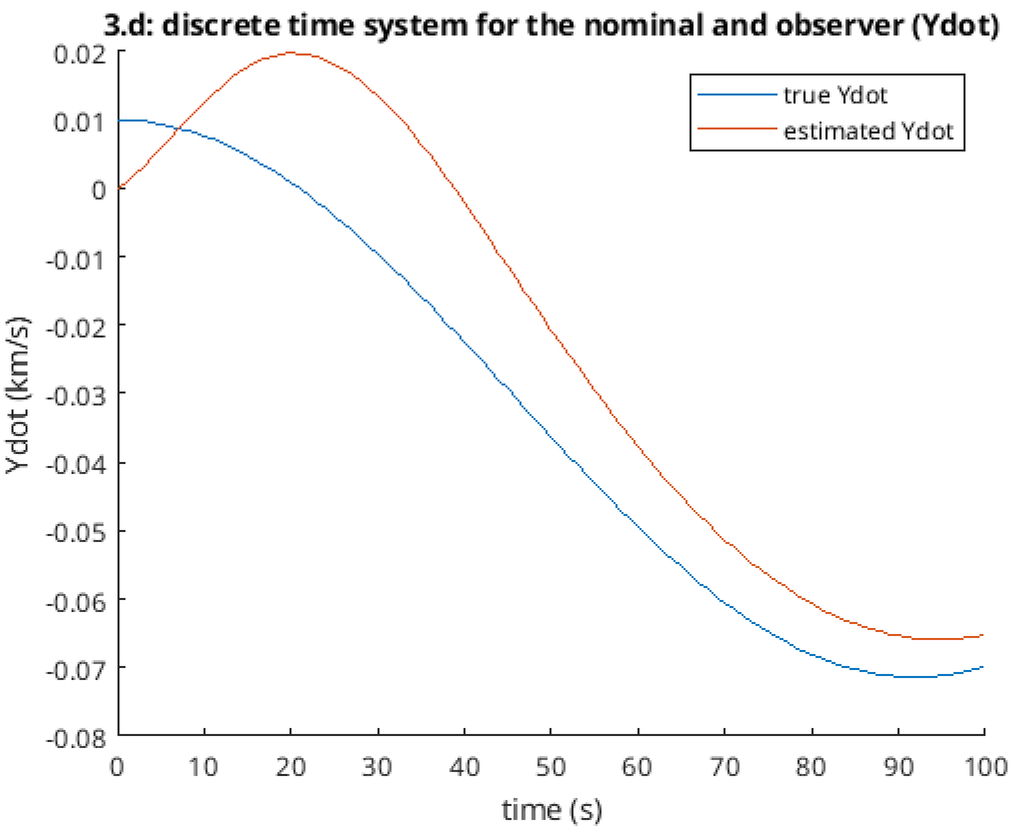
```
3.c

Ld =

    0.9425    0.0035
   -1.3674    0.0103
    0.0028   -0.1125
    0.0005    0.0000
   -0.0018    0.0000
   -0.0000   -0.0002
```

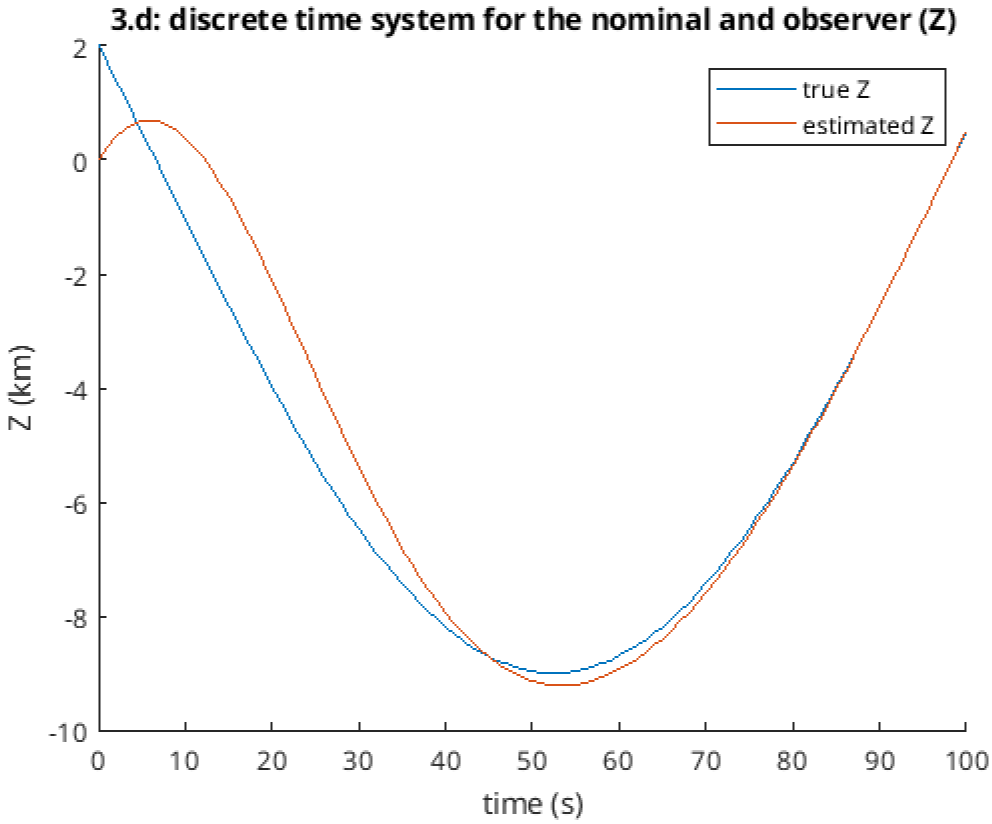
3.d simulate discrete time

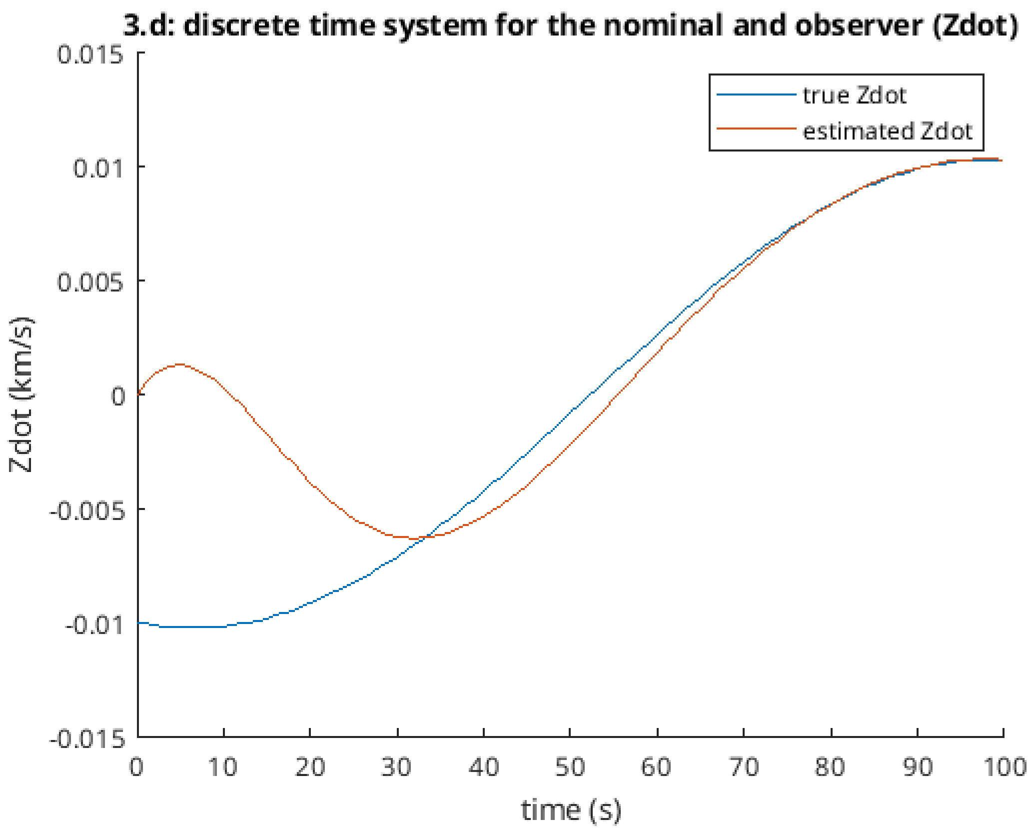
```
Xd = [0.1; 1; 2; 0;    0.01;    -0.01];
Xdhat = [0; 0; 0; 0; 0; 0];

Traj = [];
for k = 0:1:100
    Traj.time(k+1) = k;
    Traj.Xd(k+1,:) = Xd;
    Traj.Xdhat(k+1,:) = Xdhat;
    Yd = Cd*Xd;    % true output measurement
    Ydhat = Cd*Xdhat; % estimated output measurement
    Xd = Ad*Xd; % update model
    Xdhat = Ad*Xdhat + Ld*(Ydhat - Yd); % update the observer
end
```

```
figure(1);
hold on;
plot(Traj.time, Traj.Xd(:,2));
plot(Traj.time, Traj.Xdhat(:,2));
title('3.d: discrete time system for the nominal and observer (Y)');
xlabel('time (s)');
ylabel('Y (km)');
legend('true Y', 'estimated Y');
hold off;

figure(2);
hold on;
plot(Traj.time, Traj.Xd(:,5));
plot(Traj.time, Traj.Xdhat(:,5));
title('3.d: discrete time system for the nominal and observer (Ydot)');
xlabel('time (s)');
ylabel('Ydot (km/s)');
legend('true Ydot', 'estimated Ydot');
hold off;

figure(3);
hold on;
plot(Traj.time, Traj.Xd(:,3));
plot(Traj.time, Traj.Xdhat(:,3));
title('3.d: discrete time system for the nominal and observer (Z)');
xlabel('time (s)');
ylabel('Z (km)');
legend('true Z', 'estimated Z');
hold off;

figure(4);
hold on;
plot(Traj.time, Traj.Xd(:,6));
plot(Traj.time, Traj.Xdhat(:,6));
title('3.d: discrete time system for the nominal and observer (Zdot)');
xlabel('time (s)');
ylabel('Zdot (km/s)');
legend('true Zdot', 'estimated Zdot');
hold off;
```


3.d: discrete time system for the nominal and observer (Y)
true Y
estimated Y
Y (km)
time (s)
40
20
0
-20
-40
-60
-80
-100
-120
0
10
20
30
40
50
60
70
80
90
100


3.d: discrete time system for the nominal and observer (Ydot)
true Ydot
estimated Ydot
Ydot (km/s)
time (s)
0.02
0.01
0
-0.01
-0.02
-0.03
-0.04
-0.05
-0.06
-0.07
-0.08
0
10
20
30
40
50
60
70
80
90
100

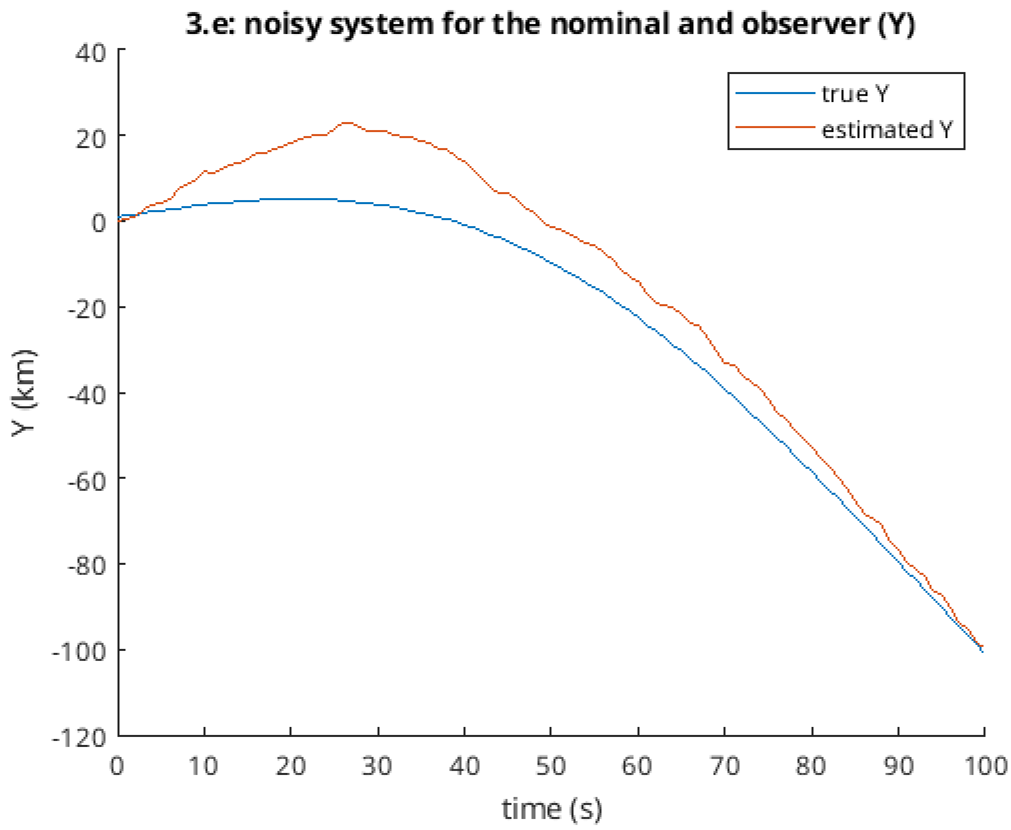

3.d: discrete time system for the nominal and observer (Z)
true Z
estimated Z
Z (km)
time (s)
2
0
-2
-4
-6
-8
-10
0
10
20
30
40
50
60
70
80
90
100

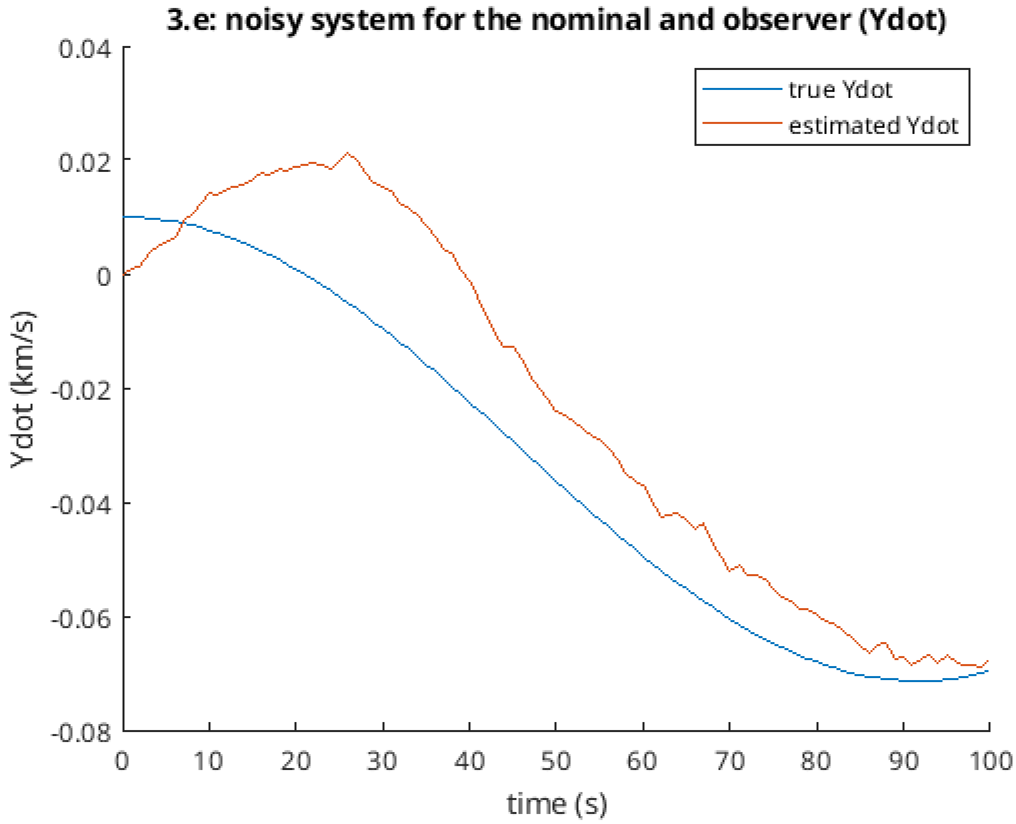

3.d: discrete time system for the nominal and observer (Zdot)
true Zdot
estimated Zdot
Zdot (km/s)
time (s)
0.015
0.01
0.005
0
-0.005
-0.01
-0.015
0
10
20
30
40
50
60
70
80
90
100

3.e simulate discrete time with noise

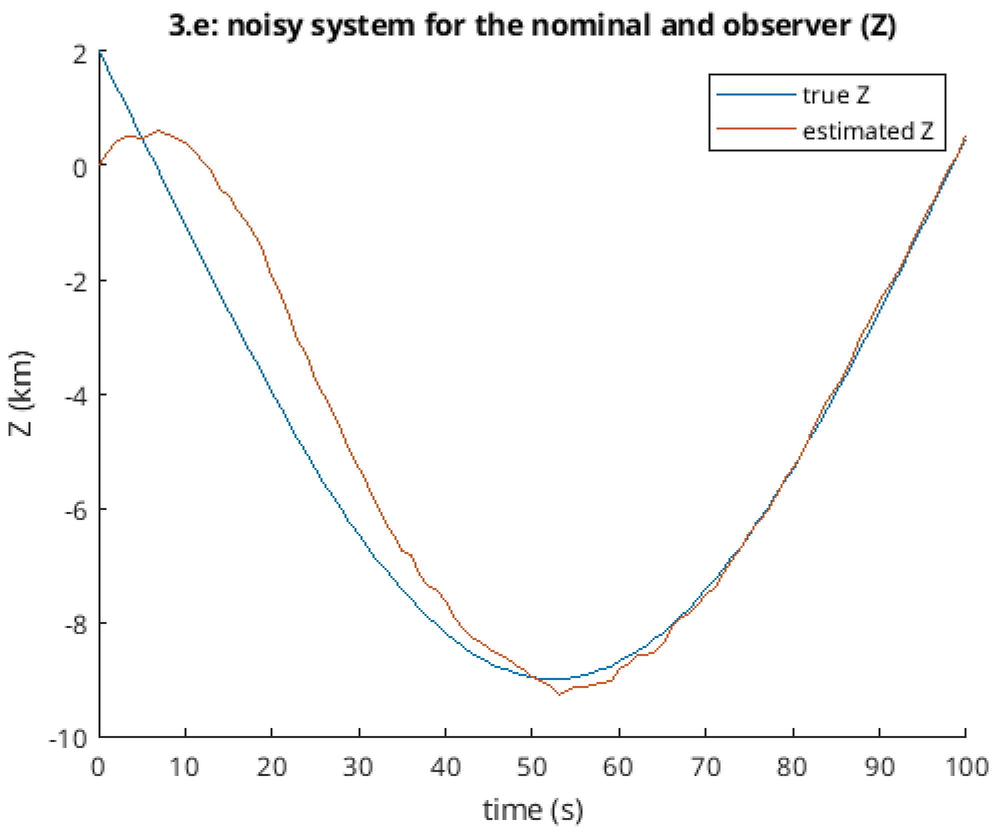
```
Xd = [0.1; 1; 2; 0;    0.01;    -0.01];
Xdhat = [0; 0; 0; 0; 0; 0];

Traj = [];
for k = 0:1:100
    Traj.time(k+1) = k;
    Traj.Xd(k+1,:) = Xd;
    Traj.Xdhat(k+1,:) = Xdhat;
    Yd = Cd*Xd+ (randn(2,1)*0.5);    % true output measurement with noise
    Ydhat = Cd*Xdhat; % estimated output measurement
    Xd = Ad*Xd; % update model
    Xdhat = Ad*Xdhat + Ld*(Ydhat - Yd); % update the observer
end

figure(5);
hold on;
plot(Traj.time, Traj.Xd(:,2));
plot(Traj.time, Traj.Xdhat(:,2));
title('3.e: noisy system for the nominal and observer (Y)');
xlabel('time (s)');
ylabel('Y (km)');
legend('true Y', 'estimated Y');
hold off;

figure(6);
hold on;
plot(Traj.time, Traj.Xd(:,5));
plot(Traj.time, Traj.Xdhat(:,5));
title('3.e: noisy system for the nominal and observer (Ydot)');
xlabel('time (s)');
ylabel('Ydot (km/s)');
legend('true Ydot', 'estimated Ydot');
hold off;

figure(7);
hold on;
plot(Traj.time, Traj.Xd(:,3));
plot(Traj.time, Traj.Xdhat(:,3));
title('3.e: noisy system for the nominal and observer (Z)');
xlabel('time (s)');
ylabel('Z (km)');
legend('true Z', 'estimated Z');
hold off;

figure(8);
hold on;
plot(Traj.time, Traj.Xd(:,6));
plot(Traj.time, Traj.Xdhat(:,6));
title('3.e: noisy system for the nominal and observer (Zdot)');
xlabel('time (s)');
ylabel('Zdot (km/s)');
```

```
legend('true Zdot', 'estimated Zdot');
hold off;
```

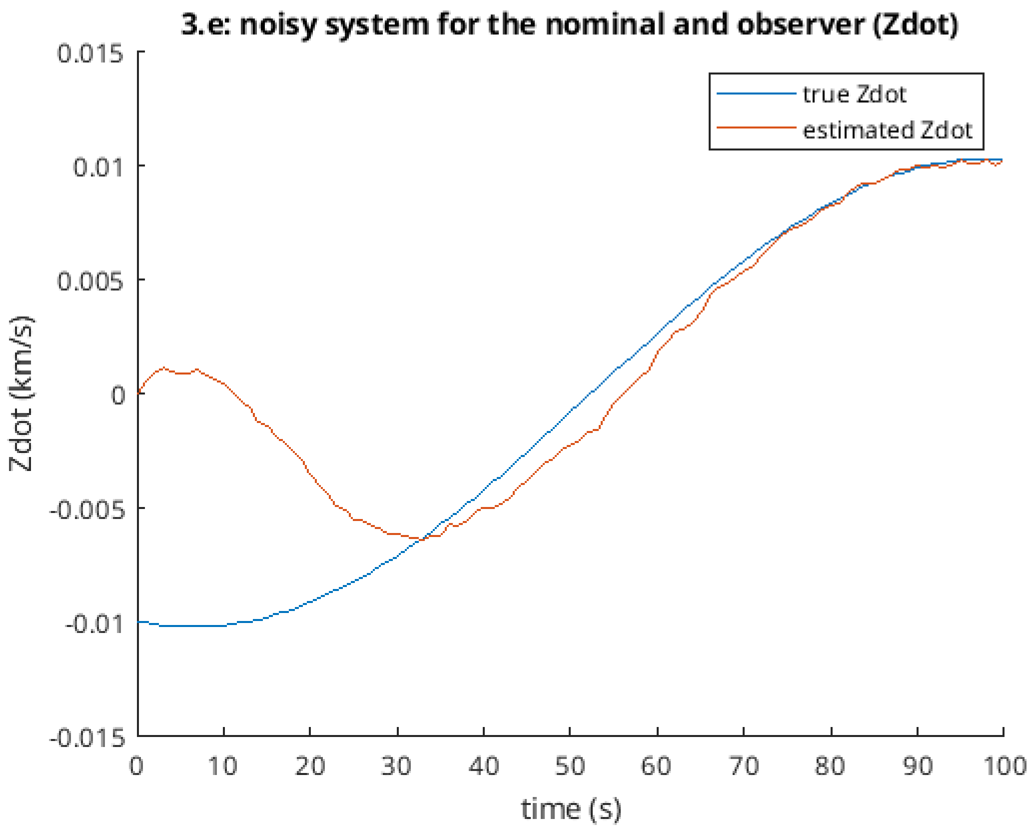




3.e: noisy system for the nominal and observer (Ydot)
true Ydot
estimated Ydot
Ydot (km/s)
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0
10
20
30
40
50
60
70
80
90
100
time (s)



3.e: noisy system for the nominal and observer (Z)
true Z
estimated Z
Z (km)
2
0
-2
-4
-6
-8
-10
0
10
20
30
40
50
60
70
80
90
100
time (s)



Published with MATLAB® R2023b