Funwork 2

Table of Contents

Evan Greene

2020-02-13

```
clear;
close all;
```

The Nonlinear Model

In Funwork Assignment 1, we created a nonlinear state-space model of a double inverted pendulum on a cart. That model is given by the equation

$$\dot{\mathbf{x}} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_3 \\ \mathbf{0} & -\mathbf{D}^{-1}\mathbf{C} \end{bmatrix} x + \begin{bmatrix} \mathbf{0} \\ -\mathbf{D}^{-1}\mathbf{g} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{D}^{-1}\mathbf{H} \end{bmatrix} u$$

where $x = \begin{bmatrix} x & \theta_1 & \theta_2 & \dot{x} & \dot{\theta}_1 & \dot{\theta}_2 \end{bmatrix}$

$$\mathbf{D} = \begin{bmatrix} M + m_1 + m_2 & (m_1 + m_2)l_1 \cos\theta_1 & m_2 l_2 \cos\theta_2 \\ (m_1 + m_2)l_1 \cos\theta_1 & (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(\theta_2 - \theta_1) \\ m_2 l_2 \cos\theta_2 & m_2 l_1 l_2 \cos(\theta_2 - \theta_1) & m_2 l_2^2 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 0 & -(m_1 + m_2)l_1 \dot{\theta}_1 \sin\theta_1 & m_2 l_2 \dot{\theta}_2 \sin\theta_2 \\ 0 & 0 & -m_2 l_1 l_2 \dot{\theta}_2 \sin(\theta_2 - \theta_1) \\ 0 & -m_2 l_1 l_2 \dot{\theta}_1 \sin(\theta_2 - \theta_1) & 0 \end{bmatrix}$$

$$\mathbf{g} = \begin{bmatrix} 0 \\ (m_1 + m_2) g l_1 \sin\theta_1 \\ m_2 g l_2 \sin\theta_2 \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Or, in MATLAB code

```
% Establish constants
M = 1.5; % kg
m1 = 0.5; % kg
m2 = 0.75; % kg
l1 = 0.5; % m
l2 = 0.75; % m
g = 9.81; % m/s^2
m12 = m1 + m2;
degrees = 180 / pi; % degrees / radian

D = @(x) [M + m12, ...
        m12 * l1 * cos(x(2)), ...
        m2 * l2 * cos(x(3));
        m12* l1 * cos(x(2)), ...
        m12* l1^2, ...
        m2 * l1 * l2 * cos(x(3) - x(2));
        m2 * l2 * cos(x(3)), ...
        m2 * l1 * l2 * cos(x(3) - x(2)), ...
        m2 * l2^2 ];

C = @(x) [ 0, -m12 * l1 * x(5) * sin(x(2)), -m2 * l2 * x(6) *
 sin(x(3));
        0, 0,             -m2 * l1 * l2 * x(6) * sin(x(3) - x(2));
        0, m2 * l1 * l2 * x(5) * sin(x(3) - x(2)), 0];

G = @(x) [0,-m12 * g * l1 * sin(x(2)), -m2 * g * l2 * sin(x(3))]';

H = [1 0 0]';

f = @(x, u) [zeros(3),  eye(3); ...
            zeros(3),   -D(x) \ C(x)] * x ...
          + [zeros(3, 1); ...
            -D(x)\G(x)] ...
            + [zeros(3, 1); ...
              D(x)\H]*u;
```

The Linear Model

We can linearize the non-linear state-space model about x = 0 as

$$\dot{\mathbf{x}} \approx \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x} - \mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial u}(u - u_0)$$

Or, in MATLAB code

```
% create symbolic variables so the Jacobian function will work.
```

```
syms x [6 1]
syms u

% Calculate the jacobians of f with respect to
J_f_x = jacobian(f(x, u), x);
J_f_u = jacobian(f(x, u), u);

% evaluate those jacobians at x = 0 and u = 0 to get our
 linearization.
A_linear = double(subs(J_f_x, x, zeros(6, 1)))
b_linear = double(subs(J_f_u, x, zeros(6, 1)))

clear x u
% we also need to find our output matrix C. In this case, the output
 is
% just the first three elements of the state vector, so
C_linear = [eye(3), zeros(3)];
```

```
A_linear =

         0         0         0    1.0000         0         0
         0         0         0         0    1.0000         0
         0         0         0         0         0    1.0000
         0   -8.1750         0         0         0         0
         0   65.4000  -29.4300         0         0         0
         0  -32.7000   32.7000         0         0         0


b_linear =

         0
         0
         0
    0.6667
   -1.3333
         0
```

Controller Design

Matlab's `place` function makes designing a controller for the system simple. It's just a matter of placing the poles.

We will place the poles at

$s = -2 \pm 2j, \; -3 \pm 3j, \; -4, \; -5$

```
p = [-2 + 2j, -2 - 2j, -3 + 3j, -3 - 3j, -4, -5]';

K = place(A_linear, b_linear, p)

controller = @(x) -K*x;
```

```
K =

    6.7334 -190.2083  222.6568    8.6412   -9.9294   38.3915
```

Observer design

The `place` function is can also be used for designing observers. Again, the only necessary part is pole placement.

We will place the poles at

$s = -5 \pm 5j, -10 \pm 10j, -15, -20$

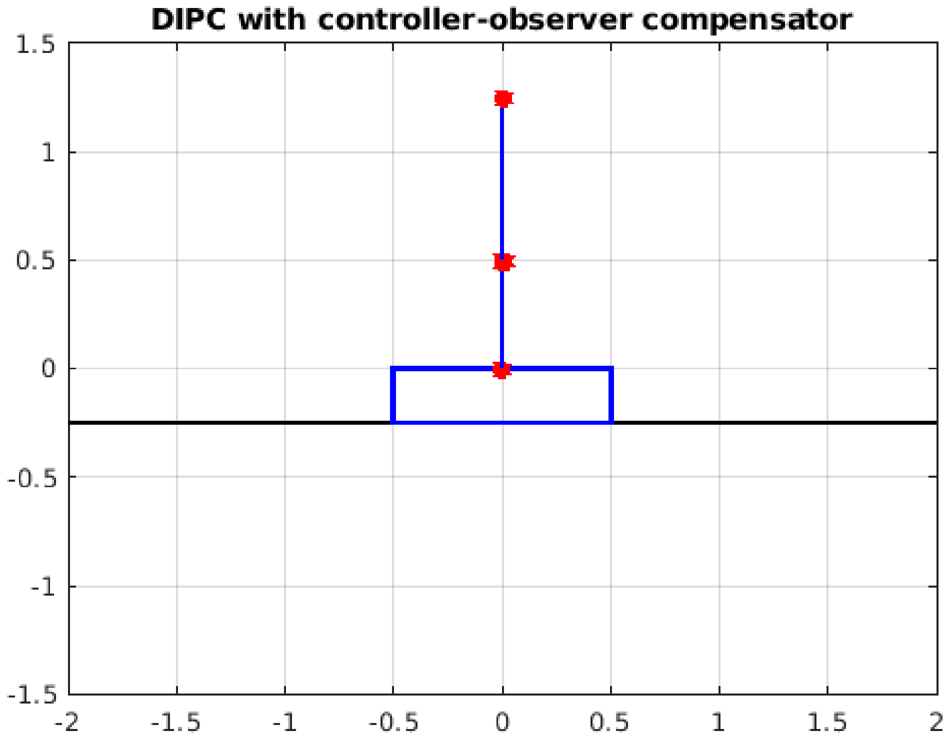
```
p = [-5 + 5i, -5 - 5i, -10 + 10i, -10 - 10i, -15, -20];

L = place(A_linear', C_linear', p)'
observer = @(xhat, y, u) (A_linear-L*C_linear)*xhat + ...
    b_linear*u + L*y;
```

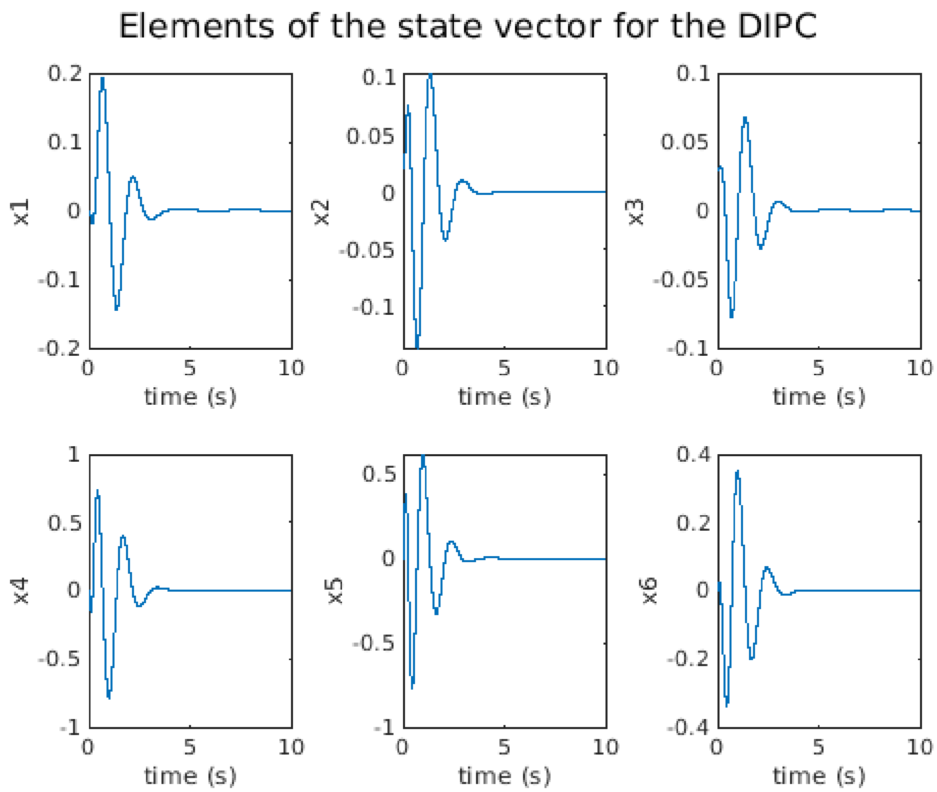
```
L =

   35.0000         0         0
         0   14.9994    5.0085
         0   -4.9915   15.0006
  300.0000   -8.1750         0
         0  165.3773  -29.3752
         0  -32.6453  132.7227
```

Animation of the combined controller-observer compensator

Now we can animate the combined controller-observer compensator.

```
% Set an inital state. If it's set too far from zero the controller
 will
% fail.
x_init = [0 0.02 0.03 0 0 0]';
animate(f, controller, observer, x_init)
figure(1)
title("DIPC with controller-observer compensator");
figure(2)
sgtitle("Elements of the state vector for the DIPC");
pause
```


DIPC with controller-observer compensator
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2


Elements of the state vector for the DIPC
x1
x2
x3
x4
x5
x6
time (s)

Adding an extra actuator to create a MIMO model.

If we wish to adjust our model to account for a second input in the form of a torque on the first joint, our equations of motion change very little. From the equation

$$\dot{\mathbf{x}} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_3 \\ \mathbf{0} & -\mathbf{D}^{-1}\mathbf{C} \end{bmatrix} x + \begin{bmatrix} \mathbf{0} \\ -\mathbf{D}^{-1}\mathbf{g} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ -\mathbf{D}^{-1}\mathbf{H} \end{bmatrix} u$$

only the value of $\mathbf{H}$ changes. The values of $\mathbf{D}$, $\mathbf{C}$ and $\mathbf{g}$ remain the same.

```
% Replace H and update f to reflect the new H;
H = [1  0;
    0   1;
    0   0;];

f = @(x, u) [zeros(3),  eye(3); ...
            zeros(3),   -D(x) \ C(x)] * x ...
          + [zeros(3, 1); ...
            -D(x)\G(x)] ...
            + [zeros(3, 2); ...
              D(x)\H]*u;
```

Linearizing the MIMO model

Linearizing the MIMO model is conceptually no different from linearizing the SIMO model. The formula remains

$$\dot{\mathbf{x}} \approx \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x} - \mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial u}(u - u_0)$$

where $x_0 = \vec{0}_6$ and $u_0 = \vec{0}_2$.

The MATLAB code also changes very little.

```
% create symbolic variables so the Jacobian function will work.
syms x [6 1]
syms u [2 1]

% Calculate the jacobians of f with respect to
J_f_x = jacobian(f(x, u), x);
J_f_u = jacobian(f(x, u), u);

% evaluate those jacobians at x = 0 and u = 0 to get our
 linearization.
A_linear = double(subs(J_f_x, x, zeros(6, 1)))
b_linear = double(subs(J_f_u, x, zeros(6, 1)))

clear x u
% we also need to find our output matrix C. In this case, the output
 is
```

```
% just the first three elements of the state vector, so
C_linear = [eye(3), zeros(3)];


A_linear =

         0         0         0    1.0000         0         0
         0         0         0         0    1.0000         0
         0         0         0         0         0    1.0000
         0   -8.1750         0         0         0         0
         0   65.4000  -29.4300         0         0         0
         0  -32.7000   32.7000         0         0         0


b_linear =

         0         0
         0         0
         0         0
    0.6667   -1.3333
   -1.3333   10.6667
         0   -5.3333
```

Controller design for the MIMO model

Designing a controller for the MIMO model is very similar to the SIMO model. We can again use the `place` function with the same poles as before

```
p = [-2 + 2i, -2 - 2i, -3 + 3i, -3 - 3i, -4, -5];

K = place(A_linear, b_linear, p)
controller = @(x) -K*x;


K =

  -38.0233  -23.5515 -173.4345  -31.1361  -19.6071  -45.4905
   -3.0024    5.3573  -19.3435   -2.8038   -1.2577   -4.3672
```

Observer design for the MIMO model

Designing an observer for the MIMO model is again very similar to the SIMO model. Using MATLAB's `place` and the same poles as before, we find

```
p = [-5 + 5j, -5 - 5j, -10 + 10j, -10 - 10j, -15, -20];

L = place(A_linear', C_linear', p)'
observer = @(xhat, y, u) (A_linear-L*C_linear)*xhat + ...
    b_linear*u + L*y;
```

```
L =

   35.0000         0         0
         0   14.9994    5.0085
         0   -4.9915   15.0006
  300.0000   -8.1750         0
         0  165.3773  -29.3752
         0  -32.6453  132.7227
```

Animation for the controller-observer compensator for the MIMO model

Now we can animate the combined controller-observer compensator.

```
% The initial state is the same as in the Funwork #1 assignment.
animate(f, controller, observer, x_init)
figure(1)
title("Two-input DIPC with controller-observer compensator")
figure(2)
sgtitle("Elements of the state vector for the two-input DIPC")

% This is where the figure should be that animates the two-input DIPC.
% If it's not here, I don't know why.
```

Animation

To easily animate the DIPC, we can create a function to save having to repeat work.

```
% we can check whether this function works by calling animate with
% the controller set to zero and the observer set to the actual state.
% controller = @(x) 0;
% observer = @(xhat, y, u)  f(x, u);
% x_init = [0 0.01 0.02 0 0 0]';
%
% animate(f, controller, observer, x_init)

function animate(f, controller, observer, x_init)
% inputs --
% f              -   the state-space function dx/dt  = f(x, u)
% controller     -   the function that relates the control input to the
%                    estimated state, u = controller(~x)
% observer       -   the function that relates the input and output to
%                    the estimated state d~x/dt = observer(~x, x, u)
% x_init         -   the initial state of the system.
% outputs --
% None. Plays animation

% close all open figures
close all
figure(1)
```

```
% Set the paramters for Euler integration
tfinal = 10; % seconds of animation
dt = 0.001; % step size
time = linspace(0, tfinal, tfinal / dt);

% Set up arrays for logging data here
logging = 1;
if (logging)
    x_log = zeros(6, length(time));
    x_est_log = zeros(6, length(time));
end

% Set up the video recording
% figure out the frame rate.
% the step size is very small, so only update the graphics like once
 every
% few frames.
frameRate = 60;
stepsPerFrame = 1 / (dt * frameRate);

% flag for whether to record video.
record_video = 0;

% if the flag is true, create the movie
if (record_video)
    % create movie
    v = VideoWriter('DIPC.avi');
    v.FrameRate = frameRate;
    v.open()
end

% set up the current and estimated states
x_current = x_init;
x_estimated = zeros(size(x_current));
% initialize input
u = 0;
% initialize output
y = x_current(1:3);

% Create graphical elements
% Rotation matrices
R1 = @(x) [cos(x(2)), -sin(x(2)); sin(x(2)), cos(x(2))];
R2 = @(x) [cos(x(3)), -sin(x(3)); sin(x(3)), cos(x(3))];


% the location of the first mass from the base.
l1 = 0.5; l2 = 0.75; % m
point1 = @(x) [x(1); 0] + R1(x) * [0; l1];
point1_current = point1(x_current);

% the location of the second mass
point2 = @(x) point1(x) + R2(x) * [0;l2];
point2_current = point2(x_current);
```

```
% The size of the cart
cart_width = 1; cart_height = 0.25;
cart_position = [x_current(1) - 0.5*cart_width, -cart_height, ...
                cart_width, cart_height];

% a line for the floor
floor = line('xdata', [-2, 2], ...
             'ydata', [-cart_height, -cart_height], ...
             'linewidth', 2, 'color', 'k');

% a rectangle for the cart
cart = rectangle('Position', cart_position, ...
                    'EdgeColor', 'b', 'linewidth', 2);

% the hinge of the pendulum base
mass0 = line('xdata', double(x_current(1)), ...
             'ydata', 0, ...
             'linewidth', 3, 'color', 'r', 'marker', '*');

% line connecting the hinge and the first mass
bar1 = line('xdata', [x_current(1), point1_current(1)], ...
            'ydata', [0,            point1_current(2)], ...
            'linewidth', 2, 'color', 'b');

% the first mass object.
mass1 = line('xdata', point1_current(1), ...
                'ydata', point1_current(2), ...
                'linewidth', 5, 'color', 'r', 'marker', '*');

% line connecting first and second masses
bar2 = line('xdata', [point1_current(1), point2_current(1)],...
            'ydata', [point1_current(2), point2_current(2)], ...
            'linewidth', 2, 'color', 'b');

% second mass
mass2 = line('xdata', point2_current(1), ...
             'ydata', point2_current(2), ...
             'linewidth', 3, 'color', 'r', 'marker', '*');

% graph settings
axis([-2 2, -1.5, 1.5])
set(gca, 'dataaspectratio', [1 1 1])
axis on
grid on
box on

for index = 1:length(time) - 1

    % find the controller input
    u = controller(x_estimated);

    % estimate the state using the observer
    % find input as a function of the state in the last time step
    dx_estimated = observer(x_estimated, y, u);
```

```
    % Euler integration -- x[k] = x[k-1] + dx[k-1]/dt * dt
    x_estimated = x_estimated + dx_estimated * dt;

    % update plant model.
    dx_current = f(x_current, u); % find dx/dt
    x_current = x_current + dx_current * dt;

    % find the output
    y = x_current(1:3);

    % Perform logging here
    if (logging)
        x_log(:, index) = x_current;
        x_est_log(:, index) = x_estimated;
    end

    % allows the fps of the animation to be different from the euler
    % integration step size.
    if mod(index, stepsPerFrame) < .999
        % update point1 and point2
        point1_current = point1(x_current);
        point2_current = point2(x_current);
        % set all the graphical elements.
        cart_position = [x_current(1) - 0.5*cart_width, -
cart_height, ...
                        cart_width, cart_height];
        set(cart,   'Position', cart_position);
        set(mass0,  'xdata', x_current(1),            ...
                    'ydata', 0);
        set(bar1,   'xdata', [x_current(1), point1_current(1)],   ...
                    'ydata', [0,            point1_current(2)]);
        set(mass1,  'xdata', point1_current(1), ...
                    'ydata', point1_current(2));
        set(bar2,   'xdata', [point1_current(1),
 point2_current(1)],...
                    'ydata', [point1_current(2), point2_current(2)]);
        set(mass2,  'xdata', point2_current(1), ...
                    'ydata', point2_current(2))
        drawnow;
        if (record_video)
            frame = getframe;
            writeVideo(v, frame);
        end
    end
end

if (record_video)
    close(v);
end

if (logging)
    figure(2)
    for index = 1:6
        subplot(2, 3, index)
```

```
        plot(time, x_log(index, :))
        hold on
%         plot(time, x_est_log(index, :))
%         hold off
%         legend("x", "\hat{x}")
        xlabel("time (s)");
        ylabel(sprintf("x%d", index));
            end
end
end
```

Published with MATLAB® R2019b