## Contents

## Rotary pendulum analysis and control

```
XX = 24;
YY = 08;
ZZ = 18;
g = 9.81;
L_a = (10+XX*20/100)*0.01;
L_p = (5+YY*25/100)*0.01;
M_a = L_a/10+0.04;
M_p = ZZ/100;
J_a = (M_a *L_a^2)/12 ; J_p = (M_p*L_p^2)/12 ; J_t = J_a*J_p + J_a*M_p*(L_p/2)^2 + J_p*M_p*
A = [0 0 1 0 ; 0 0 0 1; 0 M_p^2*L_p^2*L_a*g/(4*J_t) 0 0;0 M_p*L_p*g*(J_a+M_p*L_a^2)/(2*J_t)
B = [0;0;(J_p + 1/4*M_p*L_p^2)/J_t ; M_p*L_p*L_a/(2*J_t)];
C = [1 0 0 0; 0 1 0 0 ];
D = [0;0];
sys_ss = ss(A,B,C,D); %state space representation
```

## From state space to transfer function

To convert our state space equations to transfer function form, we need to find $H(s)$ such that:

$$Y(s) = H(s)T(s)$$

First, we take the laplace transform of both our state space equations:

$$\mathcal{L}\left[\dot{x}(t)\right] = \mathcal{L}\left[Ax(t) + BT(t)\right]$$

$$sX(s) = AX(s) + BT(s)$$

And for our output equation:

$$\mathcal{L}[y(t)] = \mathcal{L}\left[Cx(t)\right]$$
$$Y(s) = CX(s)$$

We can factor out and isolate $X(s)$ in our first equation:

$$(Is - A)X(s) = BT(s)$$
$$(Is - A)^{-1}(Is - A)X(s) = (Is - A)^{-1}BT(s)$$
$$X(s) = (Is - A)^{-1}BT(s)$$

We can insert this expression for $X(s)$ into our second equation:

$$Y(s) = C(Is - A)^{-1}BT(s)U(s)$$

Finally, we can divide by $U(s)$ on both sides to find our transfer function:

$$H(s) = \frac{Y(s)}{U(s)} = C(Is - A)^{-1}$$

## Computing $H(s)$

Let's find the transfer function for our system, manually and with matlab's functionality.

I round off the entries in A to make my life a bit easier.

```
A = [0 0 1 0;
     0 0 0 1;
     0 181 0 0;
     0 782 0 0;];
```

```
B = [0;
     0;
     921;
     2921;];

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

D = [0;
     0];

sys_ss = ss(A, B, C, D);
s = tf('s');
```

The easy way:

```
H_matlab = tf(sys_ss);
```

Manually:

```
I = eye(4); % 4x4 identity
H_manual = C*((I*s-A)\B);
```

We now have the transfer function for our plant, one computed with tf() and one manually.

```
H_matlab
```

```
H_matlab =

  From input to output...
       921 s^2 + 1.636e-12 s - 1.915e05
   1:  --------------------------------
        s^4 - 1.066e-14 s^3 - 782 s^2

                2921
   2:  -----------------------
       s^2 - 1.066e-14 s - 782

Continuous-time transfer function.
```

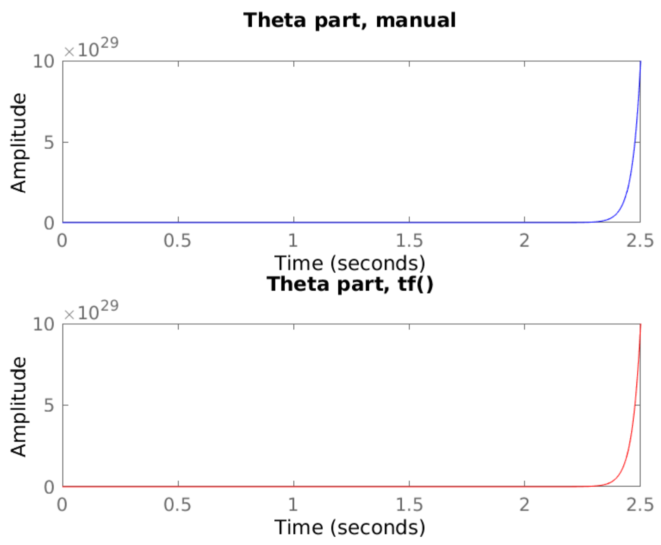
```
H_manual
```

```
H_manual =

  From input to output...
                   921 s^2 - 1.178e-10 s - 1.915e05
   1:  ------------------------------------------------------
       s^4 - 1.495e-14 s^3 - 782 s^2 + 6.066e-13 s - 1.97e-29

         2921
   2:  ---------
       s^2 - 782

Continuous-time transfer function.
```

## Verifying the two transfer functions

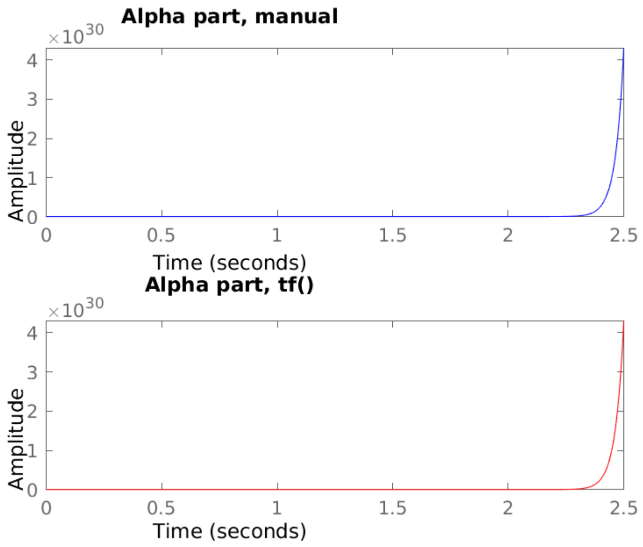
The two transfer functions seemed a little different so i plotted their step responses to make sure they behave the same:

```
subplot(2,1,1);
step(H_manual(1), 'b-')
title('Theta part, manual')
subplot(2,1,2);
step(H_matlab(1), 'r-')
title('Theta part, tf()')
```



```
close;
```

```
subplot(2,1,1);
step(H_manual(2), 'b-')
title('Alpha part, manual')
subplot(2,1,2);
step(H_matlab(2), 'r-')
title('Alpha part, tf()')
```



The manually computed and tf(sys_ss) transfer functions seem to behave the same, so I conclude that they only 'look' different due to numerical differences and matlab's tf() implmentation.

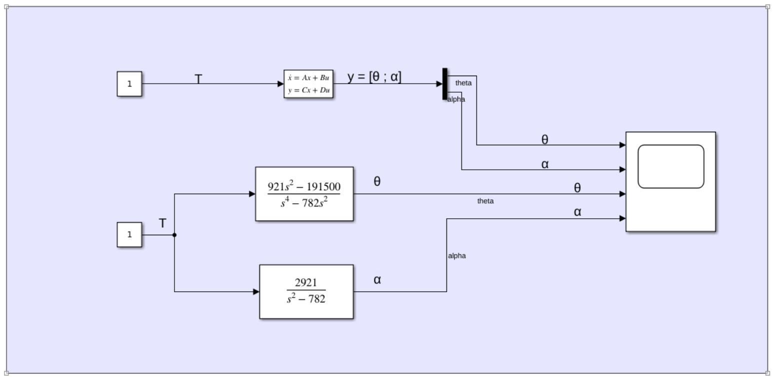
## Plant transfer functions

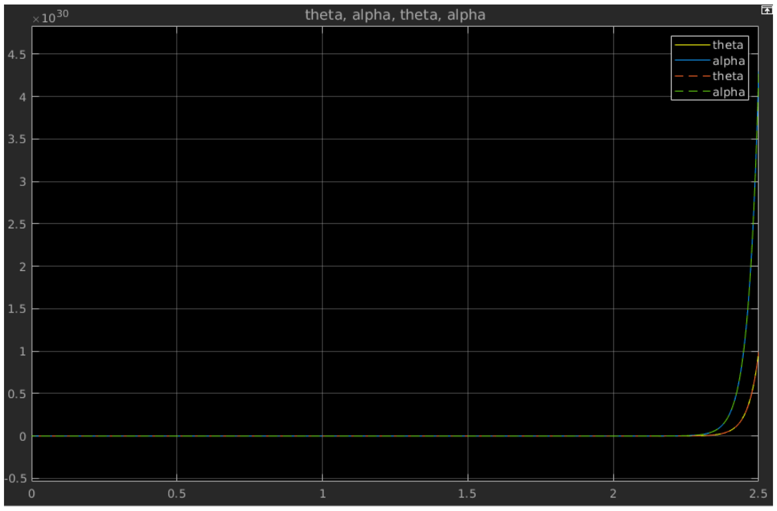
Before I move on, i remove the extremely small coefficients in the transfer functions, as they have virtually no impact) I verified this by checking that the poles didnt change.

```
close;

H_theta = (921*s^2 - 191500)/(s^4 - 782*s^2); %theta transfer function
H_alpha  = (2921)/(s^2-782); %alpha transfer function
H = [H_theta;H_alpha];
```

## Verifying the transfer function representation in simulink

To make sure the transfer function accurately describes our system, i plotted the responses of the original state space system together with the responses of the equivalent transfer functions.





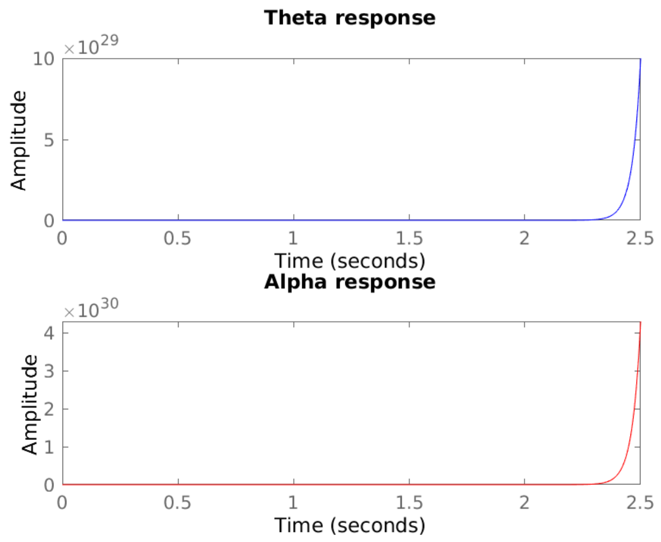
The responses from the two representations are identical, so we are ready to move on with the transfer function analysis. We end up with the following plant transfer functions for $\theta$ and $\alpha$:

$$H_\theta = \frac{921s^2 - 191500}{s^4 - 782s^2}$$

$$H_\alpha = \frac{2921}{s^2 - 782}$$

## Plant step response

```
close;
subplot(2,1,1);
step(H_theta, 'b')
title('Theta response')
subplot(2,1,2);
step(H_alpha, 'r')
title('Alpha response')
```



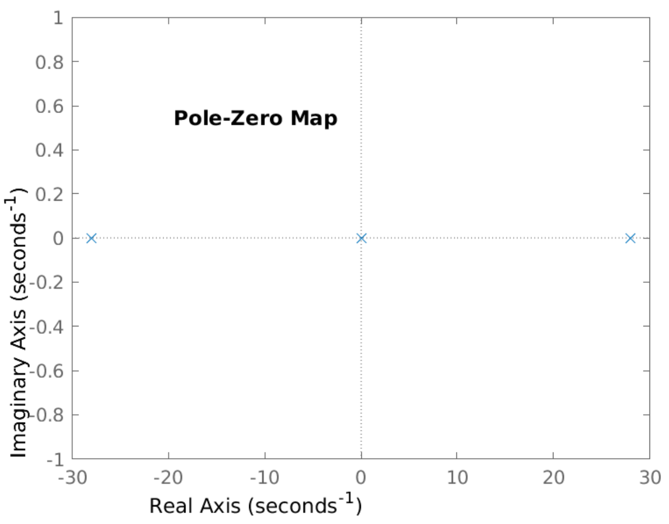
## System stability and poles

The system is clearly unstable in both $\theta$ and $\alpha$.

Checking the poles with pzplot() and pole():

```
close;
pzplot(H);
pole(sys_ss)
```

```
ans =

         0
         0
   27.9643
  -27.9643
```



Our system has four poles:

$$[0, 0, 27.9643, -27.9643]$$

The system has poles in the right half-plane, and is therefore unstable.

```
close;
```

## Feedback gain vector k

We need to find a gain vector $\vec{k} = [k_1, k_2, k_3, k_4]$ Which brings the poles/eigenvalues to -10, for our system:

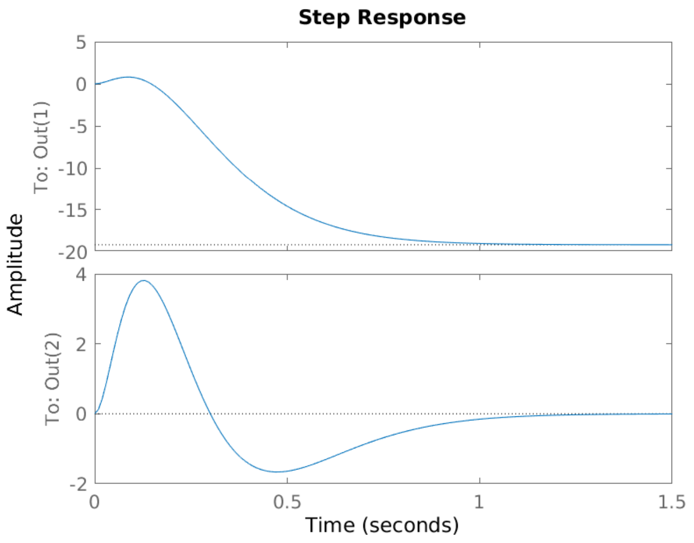
$$\dot{\vec{x}} = A\vec{x} + \vec{B}T$$

We can do this by using the acker() commands, specifying our desired pole locations

```
P = [-10, -10, -10, -10];
K = acker(A, B, P);
A_CL = A-B*K;
```

Plotting the new step response of our controlled system with desired pole placement (top: $\theta$, bottom: $\alpha$):

```
sim_time = 3;
new_sys = ss(A_CL, B, C, D);

close;
step(new_sys);
```



As we can see, by using full state feedback to place our poles, the system is now stable.

## State space representation with full state feedback

We have found a K matrix/vector such that the poles of our new system is stable (this requires full state estimation). We did this by implicitly defining our control input $T$ as:

$$T = r - \vec{k}\vec{x}$$

Where $r$ is the reference/input. In this way, our new system becomes:

$$\dot{\vec{x}} = A\vec{x} + B(r - \vec{k}\vec{x})$$

$$\dot{\vec{x}} = A\vec{x} + Br - B\vec{k}\vec{x}$$

$$\dot{\vec{x}} = (A - B\vec{k})\vec{x} + Br$$

We now arrive at a new state space representation of our system, by defining our modified A-matrix $A_{CL}$:

$$A_{CL} = (A - B\vec{k})$$

$$\dot{\vec{x}} = A_{CL}\vec{x} + Br$$

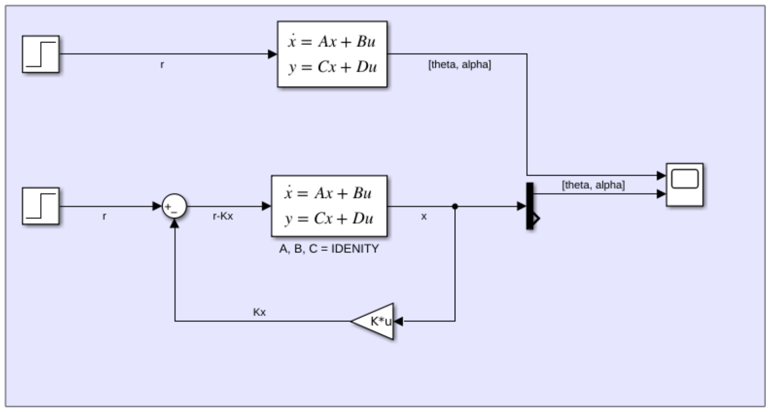
$$\vec{y} = C\vec{x}$$

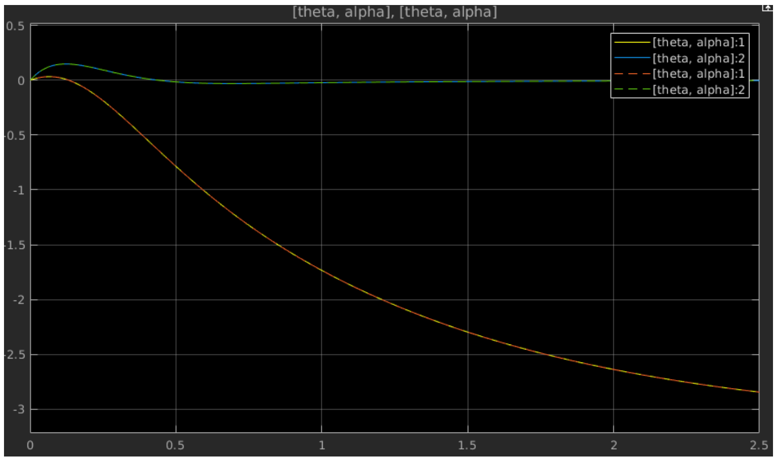
```
A_CL
```

```
A_CL =

         0         0    1.0000         0
         0         0         0    1.0000
   48.0887 -269.9112   19.2355  -18.6771
  152.5159 -648.0887   61.0064  -59.2355
```

## Verifying new state space representation

To verify that our reduced state space formulation is correct, i compare the responses in simulink:





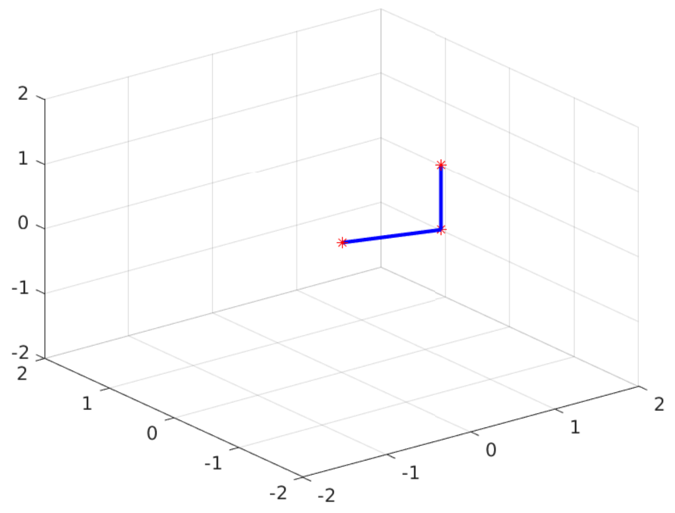
As we can see, the reduced system is identical to the original full-state feedback system.

## Pendulum animation

Here I make an attempt at animating the step response of the controlled pendulum.

```
close;
[y, t] = step(new_sys, sim_time);
timesteps = size(t);
delta_t = sim_time/timesteps(1);

for k=1:length(t)
    draw_pendulum(y(k,:));
    %pause(delta_t);
```

```
end
```



## LQR method

We can compute other K-vectors by defining Q and R matrices, which weight the cost of state errors and control inputs. LQR is a method of finding optimal gain matrices by minimizing the quadratic cost function:

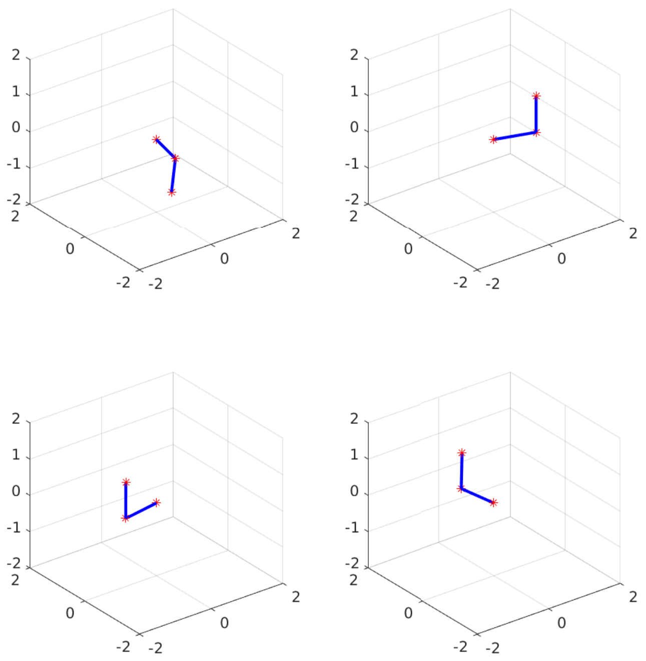
$$\int_0^\infty (x^T Qx + u^T Ru)dt$$

This is a quadratic cost function, and the minima of this function has an explicit solution $\vec{u} = -K\vec{x}$. We can use the matlab lqr() function to get this gain matrix $K$, for our given system of linear ODE's and our two weight matrices $Q$ and $R$.

In this section, i play around with different weight matrices, and see how their corresponding K-matrices affect the system. LQR assumes full-state feedback, and is similar to pole placement, except instead of specifying specific poles, we specify whether we care more about the fast convergence of certain system states, or if we care more about having a power-efficient/gentle controller.

```
close;
sim_time = 3;
% Doesn't care about state, terrible control:
```

```
Q = 0.0001*eye(4);
R = 10;
K = lqr(sys_ss, Q, R);
A_CL = A-B*K;
sys_lqr1 = ss(A_CL, B, C, D);
[y_lqr1, t] = step(10*sys_lqr1, sim_time);


% Aggressive controller
Q = 10000*eye(4);
R = 10;
K = lqr(sys_ss, Q, R);
A_CL = A-B*K;
sys_lqr2 = ss(A_CL, B, C, D);
[y_lqr2, t] = step(10*sys_lqr2, sim_time);

% Somewhere in between:
Q = 100*eye(4);
R = 10;
K = lqr(sys_ss, Q, R);
A_CL = A-B*K;
sys_lqr3 = ss(A_CL, B, C, D);
[y_lqr3, t] = step(10*sys_lqr3, sim_time);

% Lazy/economical controller:
Q = 10*eye(4);
R = 100;
K = lqr(sys_ss, Q, R);
A_CL = A-B*K;
sys_lqr4 = ss(A_CL, B, C, D);
[y_lqr4, t] = step(10*sys_lqr4, sim_time);


timesteps = size(t);
delta_t = sim_time/timesteps(1);
for k=1:length(t)
    draw_pendulum_2x2(y_lqr1(k,:), 1);
    draw_pendulum_2x2(y_lqr2(k,:), 2);
    draw_pendulum_2x2(y_lqr3(k,:), 3);
    draw_pendulum_2x2(y_lqr4(k,:), 4);
    %pause(delta_t);
end
```


2
1
0
-1
-2
2
0
-2
-2
0
2