Task 1

Analysis of non-linear and linearized water tank model with and without PID control.

Contents

- Model and linearization
- State Space Representation
- Simulink implementation
- Simulation of open loop system
- Transfer function response
- Step response with control
- Applying the control to the non-linear and linearized plants in simulink

Model and linearization

Differential equation describing the tank water level:

$$\frac{d}{dt}H = \frac{bV - a\sqrt{H}}{A}$$

This ode is non-linear in $\sqrt{H}$. However, we can approximate this by a first order taylor expansion/linearization in a neighbourhood $H_0 + \hat{H}$ around the stationary point H_0:

$$\sqrt{H_0 + \hat{H}} \approx \sqrt{H_0} + \frac{1}{2\sqrt{H_0}} \cdot (H - H_o)$$

With this linearization we arrive at the linear ODE:

$$\frac{dH}{dt} = \frac{b}{A}V - \frac{a}{2A}\sqrt{H_0}(H - H_0) - \frac{a\sqrt{H_0}}{A}$$

State Space Representation

We can find the state space representation of this system (ignoring the non-homogeneous part):

$$\frac{d}{dt}H = \left[-\frac{a}{2A\sqrt{H_0}} \right] H + \left[\frac{b}{A} \right] V$$

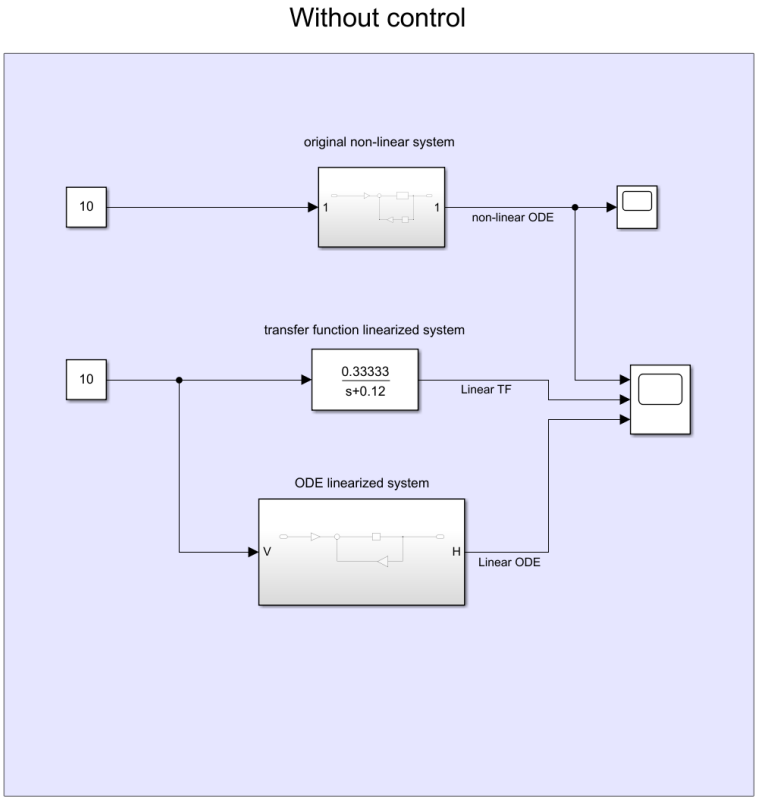
Alternatively, with our constants and linearization point:

$$\dot{H} = \left[-\frac{3\sqrt{10}}{80} \right] H + \left[\frac{1}{3} \right] V$$

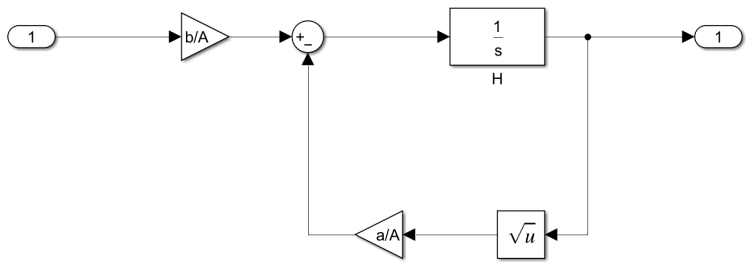
Our state space consists of a one-dimensional state vector H and a one-dimensional control vector V

Simulink implementation

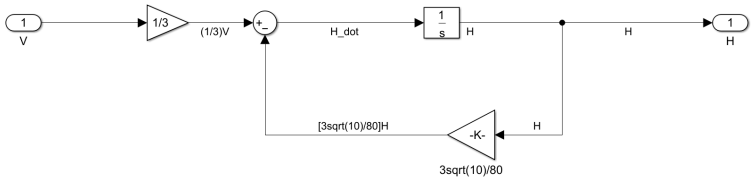
High-level overview of non-linear system, linear TF system and linear ODE system.



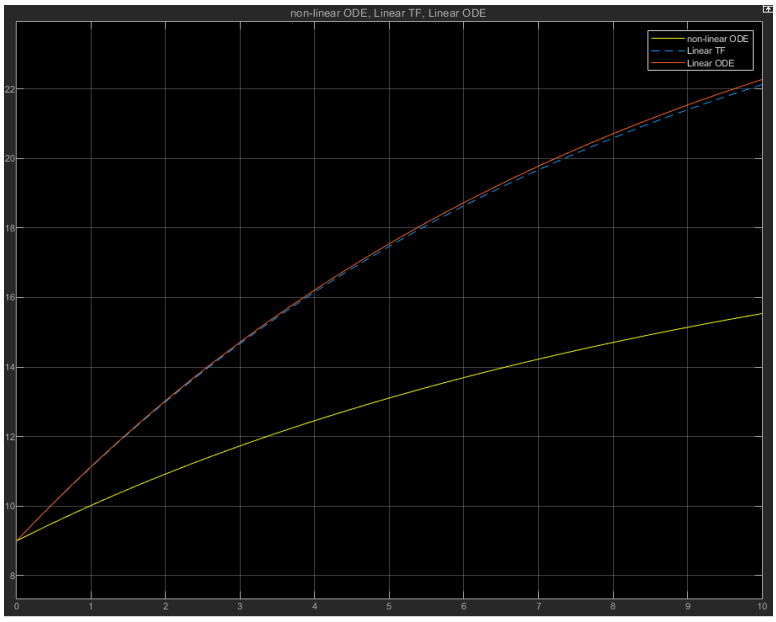
Open-loop non-linear system:



Open-loop linear system:



Plotting the responses of each system (input 10, initial output 9):



As can be seen, the TF and ODE implementation are identical (up to number

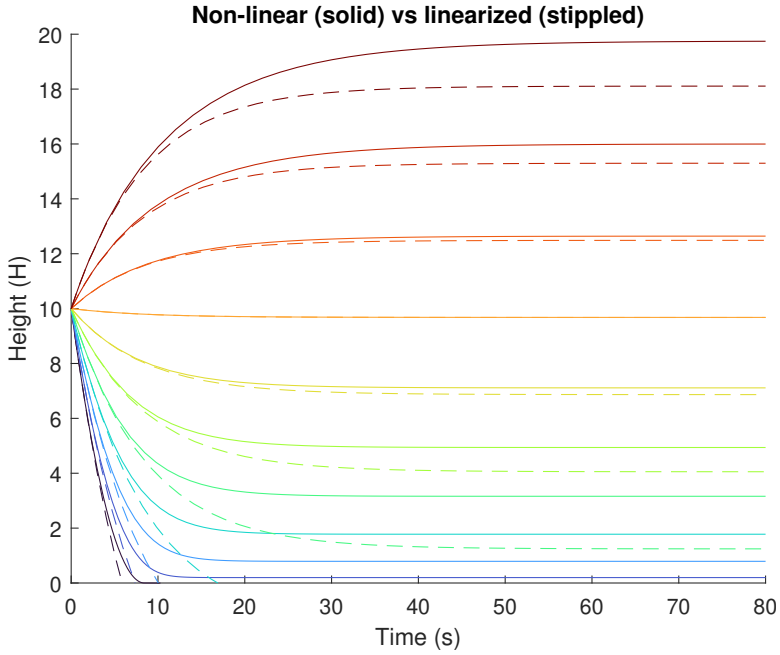
rounding), while the linear approximations become less and less accurate relative to the true non-linear system.

Simulation of open loop system

For fun, we can simulate the non-linear and linearized differential equations to inspect the accuracy of the linear approximation. To do this I use matlab's ode45 which can simulate most ordinary differential equations The non-linear and linear ODE's are defined in tank_system_linear.m and tank_system_nonlinear.m, respectively.

```
A = 24; b = 8; a = 18; %water tank parameters

timespan = [0, 80]; %simulaion interval
H0 = 10; %initial level
colormap = turbo(11);

% Solve the and plot the ODE's
figure;
hold on;
for V = 0:10
    [t, H] = ode45(@(t, H) tank_system_nonlinear(t, H, a, b, A, V), timespan, H0);
    plot(t, H, 'Color', colormap(V+1, :));

    [t, H] = ode45(@(t, H) tank_system_linear(t, H, a, b, A, V, H0), timespan, H0);
    plot(t, H,'--','Color', colormap(V+1, :));
end

xlim(timespan);       ylim([0,20]);
xlabel('Time (s)');   ylabel('Height (H)');
title('Non-linear (solid) vs linearized (stippled)');
```

```
Warning: Imaginary parts of complex X and/or Y arguments ignored.
```



Transfer function response

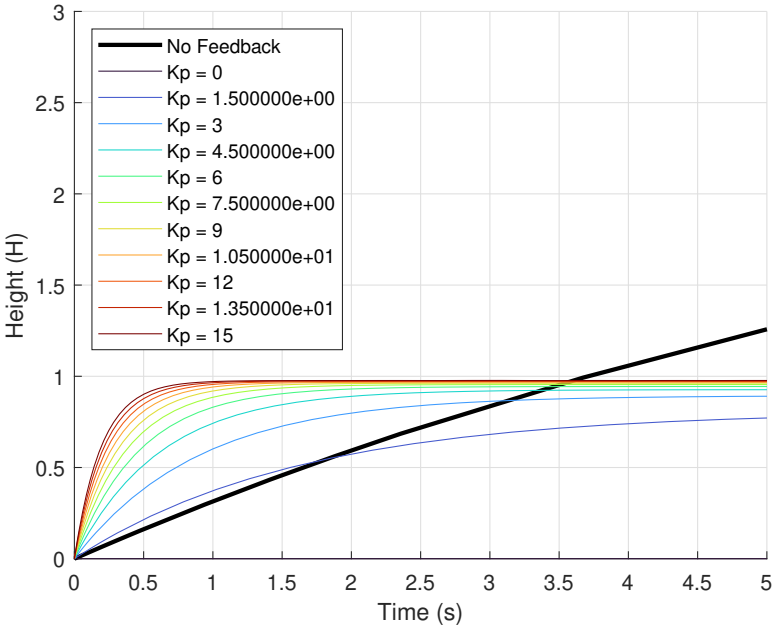
The transfer function is determined by taking the laplace transform of the homogeneous linear ODE, which after rounding gives:

$$\frac{H(s)}{V(s)} = \frac{\frac{b}{A}}{s + 0.16\frac{a}{A}}$$

Here, I simulate and plot the closd loop step responses for varying proportional gains:

```
close;

H0 = 0; %initial tank level
V = 1; %step response no feedback
H_desired = 1; %setpoint
colormap = turbo(11);

step_timespan = [0, 5];
figure;
grid on;
hold on;
```

```
%without feedback:
Kp = 0;
[t, H] = ode45(@(t, H) tank_system_step(t, H, a, b, A, H_desired, H0, V), timespan, H0);
plot(t, H, 'Color', 'k', 'LineWidth', 2, 'DisplayName', 'No Feedback');

for i = 0:10
    Kp = i*1.5;
    [t, H] = ode45(@(t, H) tank_system_step_feedback(t, H, a, b, A, H_desired, H0, Kp), time
    plot(t, H, 'Color', colormap(i+1, :), 'DisplayName', sprintf('Kp = %d', Kp));
end

xlim(step_timespan);       ylim([0,3]);
xlabel('Time (s)');   ylabel('Height (H)');
legend('Location', 'northwest');
```



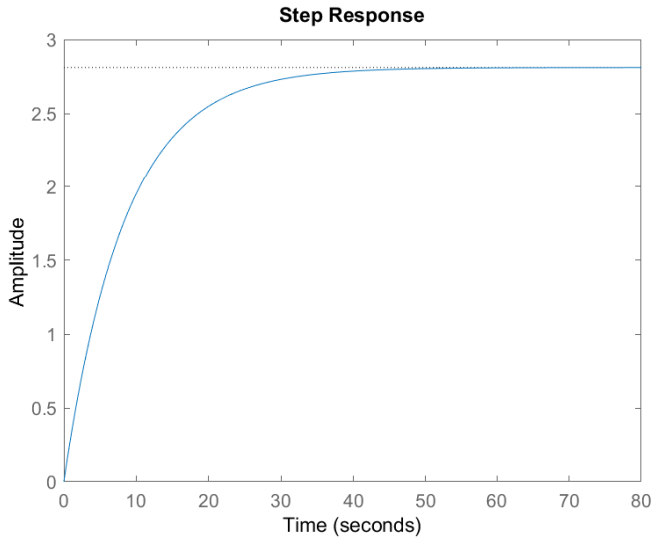
Since the system is a first-order linear system without disturbances, it behaves nicely and doesn't reach any oscillations.

Step response with control

Now we develop a complete PID controller for our plant using the transfer function of the linearized model.

```
close;
```

```
s = tf('s');

A = 24; b = 8; a = 18; %water tank parameters
tf_linear = (b/A)/(s+1/(2*sqrt(10))*(a/A)); %defining transfer function as given in the task
step(tf_linear);
```

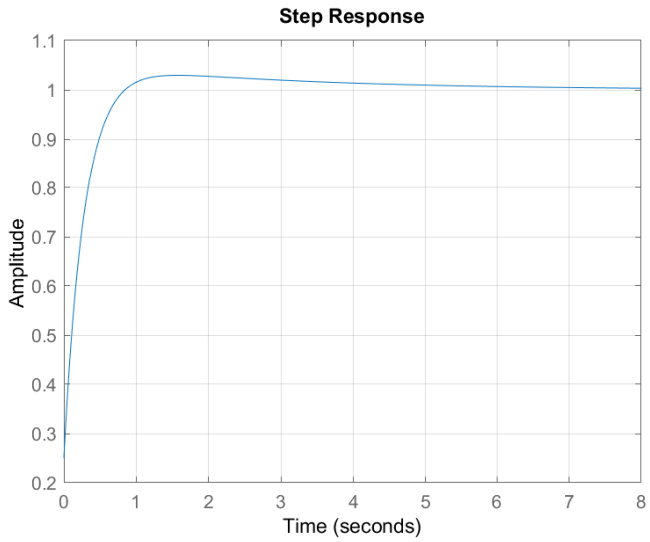


Now we tune all the gains to achieve the desired response

```
Kp = 15;
Ki = 5;
Kd = 1;

H = tf_linear; %plant transfer function
C = Kp + Ki/s + Kd*s; %PID transfer function

CH = C*H; %closed loop transfer function
closed_tf = CH/(CH+1);
step(closed_tf);

info = stepinfo(closed_tf);
overshoot     = info.Overshoot;
rise_time     = info.RiseTime;
settling_time = info.SettlingTime;

% Display results
fprintf('Overshoot: %.2f%%\n', overshoot);
fprintf('Rise Time: %.2f s\n', rise_time);
```

```
fprintf('Settling Time: %.2f s\n', settling_time);
grid on;
```

```
Overshoot: 2.95%
Rise Time: 0.51 s
Settling Time: 3.76 s
```

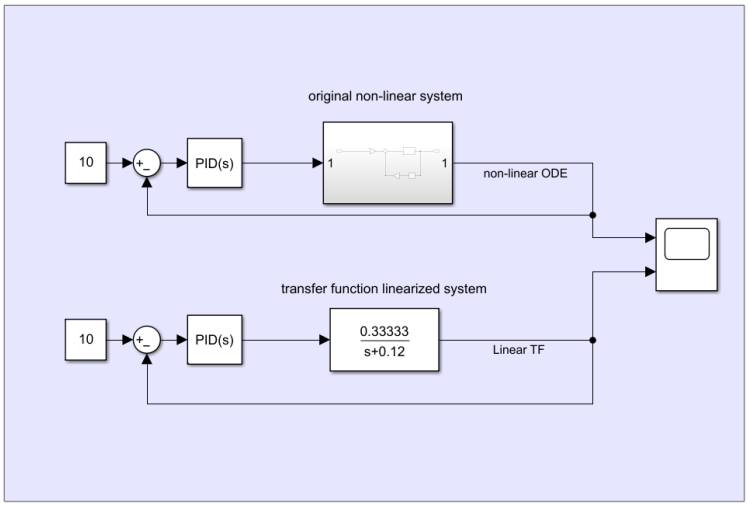


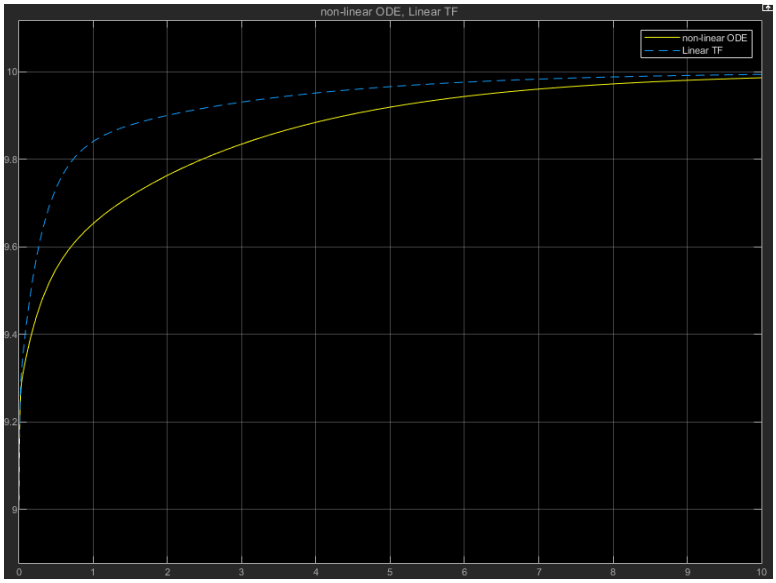
Applying the control to the non-linear and linearized plants in simulink

With our conservative gains [Kp, Ki, Kd] = [15, 5, 1] We achieved a nice response.

Here I plot the response for a setpoint of 10 an an initial tank level of 9, for both the non-linear and linear systems.

With control





The response of the linear system here is not just a scaled version of the closed_loop unit step response, since the setpoint and initial conditions arent just scaled versions of the unit step (theres a translation in the setpoint). If we instead plotted the response of a setpoint of 10 and an initial condition of 0, the response of the linearized system would be identical to a scaled version of the unit step response. We also see again that the linear approximation starts out good, but diverges from the true state of the system as we get further away from the linearization point.

Task 2

Analysis of piloted airplaine stability.

Contents

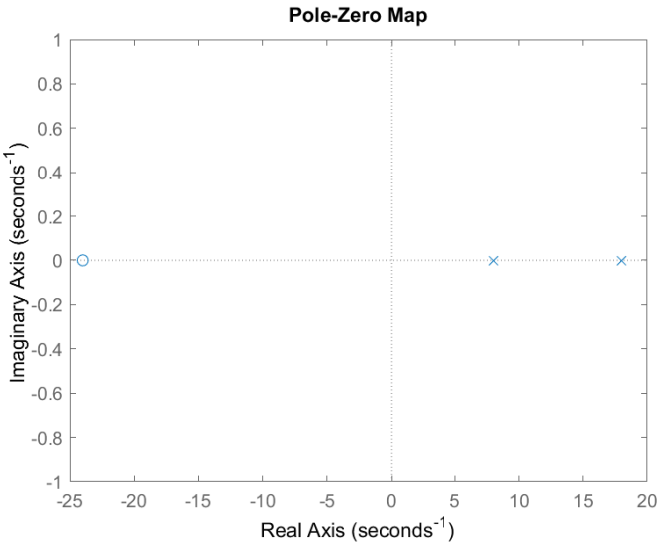
Transfer function

the transfer function, with canard deflection as input and pitch altitude as output, is given as:

$$\frac{\theta}{\delta_c} = \frac{s+24}{(s-8)(s-18)}$$

This system is clearly unstable, as both poles are positive real numbers (8 and 18). We can verify this by using pzplot and looking at the poles:

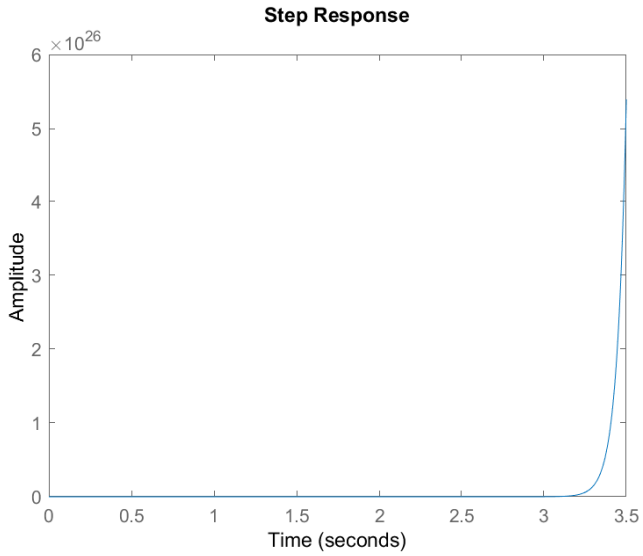
```
s = tf('s');
sys = (s+24)/((s-8)*(s-18));
pzplot(sys);
```



As we can see, both poles have positive real part, so the system must be unstable

Step response and stability

```
close;
step(sys);
```

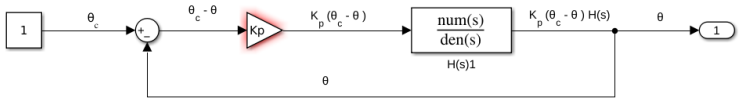


The output of our system blows up to infinity, so it is clearly unstable.

The open-loop system does not satisfy BIBO, and requires closed loop control to become stable.

Propotional control transfer function

We add a proportional control and find an equivalent transfer function for the whole system. I use the simulink block diagram to help find the expression for the new system:



$$\theta = K_p\theta_c H(s) - K_p\theta H(s)$$

$$\theta(K_p H(s) + 1) = \theta_c K_p H(s)$$

$$\frac{\theta}{\theta_c} = \frac{K_p H(s)}{K_p H(s) + 1}$$

I insert the plant model $H(s)$ to obtain the full closed loop transfer function:

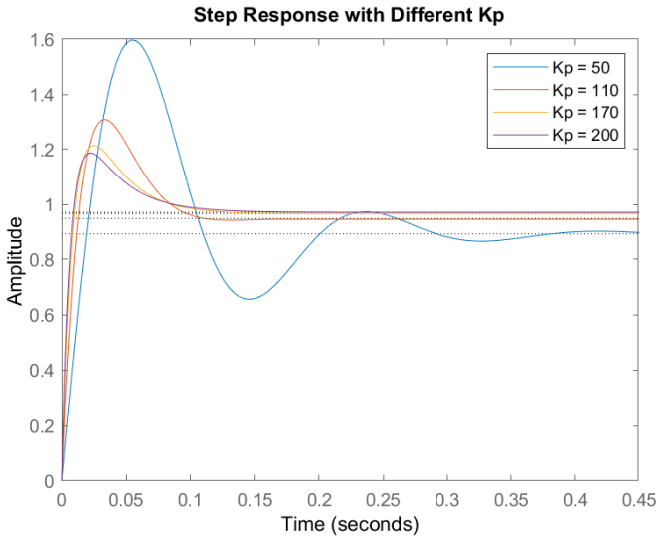
$$\frac{\theta}{\theta_c} = \frac{K_p \frac{s+24}{s^2-26+144}}{K_p \frac{s+24}{s^2-26+144} + 1}$$

Finally, simplifying the fraction gives:

$$\frac{\theta}{\theta_c} = \frac{K_p s + K_p 24}{s^2 + (K_p - 26)s + (144 + 24K_p)}$$

Now that we have obtained the transfer function for our controlled system, We can plot the step responses for varying Kp's (proportional gains):

```
figure;
hold on;
for Kp = [50 110 170 200]
    closed_loop = (Kp*s+Kp*24)/(s^2+(Kp-26)*s+(144+24*Kp));
    step(closed_loop);
end

title('Step Response with Different Kp');
legend('Kp = 50', 'Kp = 110', 'Kp = 170', 'Kp = 200');
```



Increasing the proportional gain results in a shorter rise time, and a smaller steady state error. However, we cannot completely eliminate the steady state error completely, without introducing an integral term.

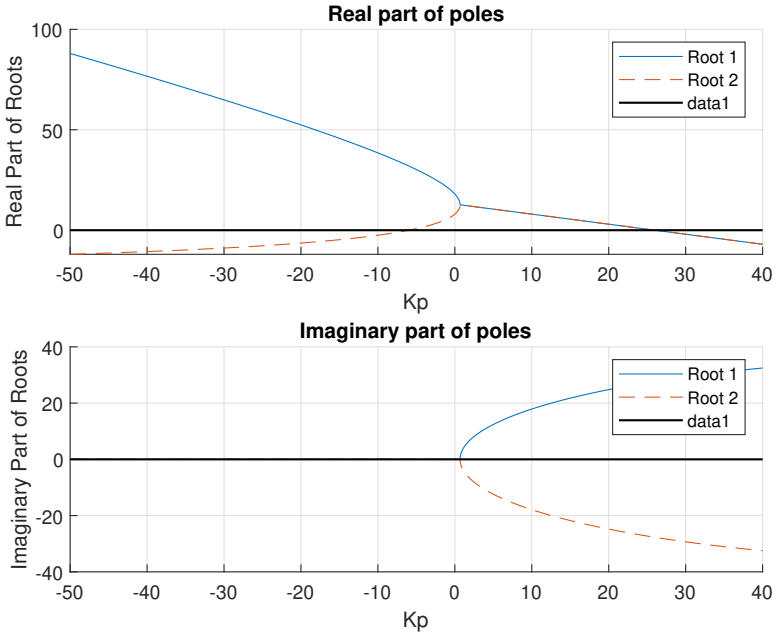
P-control stability

We can also plot the poles of this system as a function of the proportional gain K_p The poles are given by the roots of the polynomial in the denominator of our transfer function: We can use the quadratic formula to find the roots and plot $Re\{s\}$ and $Im\{s\}$ separately. We solve the equation:

$$s^2 + (Kp - 26)s + (144 + 24)Kp = 0$$

```
close;
```

```
Kp = -50:0.05:40; %range of Kp's to analyse
%find poles using the quadratic formula
s_1 = (-(Kp-26)+sqrt((Kp-26).^2-4*(144+24.*Kp)))/2;
s_2 = (-(Kp-26)-sqrt((Kp-26).^2-4*(144+24.*Kp)))/2;

figure;

subplot(2,1,1);
grid on;
hold on;
plot(Kp, real(s_1), 'DisplayName', 'Root 1', 'LineStyle', '-');
plot(Kp, real(s_2), 'DisplayName', 'Root 2', 'LineStyle', '--');
plot(Kp, zeros(size(Kp)), 'k', 'LineWidth', 1);  % x-axis
xlabel('Kp'); ylabel('Real Part of Roots');
title('Real part of poles');
legend;

subplot(2,1,2);
grid on;
hold on;
plot(Kp, imag(s_1), 'DisplayName', 'Root 1', 'LineStyle', '-');
plot(Kp, imag(s_2), 'DisplayName', 'Root 2', 'LineStyle', '--');
plot(Kp, zeros(size(Kp)), 'k', 'LineWidth', 1);  % x-axis
xlabel('Kp'); ylabel('Imaginary Part of Roots');
title('Imaginary part of poles');
legend;
```



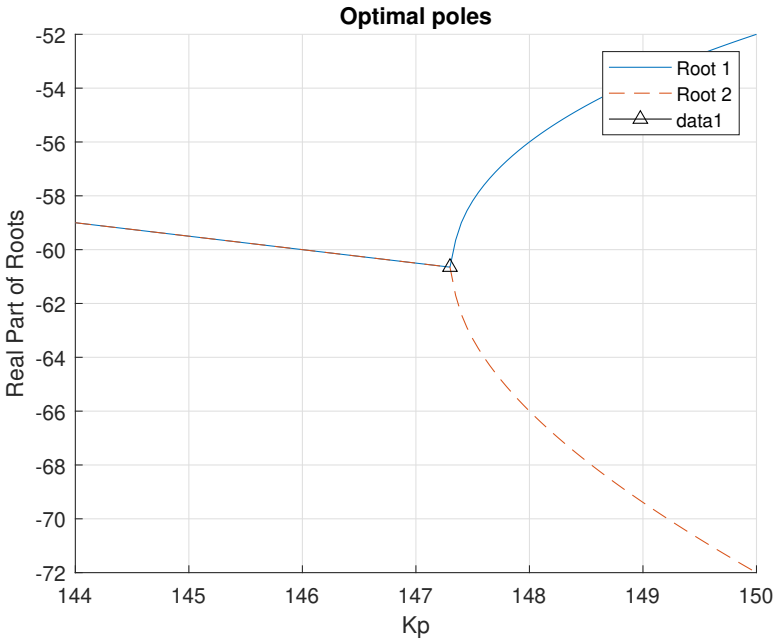
Kp-stable values

From inspecting the plot we see that both poles/eigenvalues have negative real part when $K_p > 26$.

For the gain, i somewhat arbitrarily chose $Kp = 147.3$, which is where the poles start to separate again:

```
close;
figure;
grid on;
hold on;

Kp = 144:0.05:150;
s_1 = (-(Kp-26)+sqrt((Kp-26).^2-4*(144+24.*Kp)))/2;
s_2 = (-(Kp-26)-sqrt((Kp-26).^2-4*(144+24.*Kp)))/2;
plot(Kp, real(s_1), 'DisplayName', 'Root 1', 'LineStyle', '-');
plot(Kp, real(s_2), 'DisplayName', 'Root 2', 'LineStyle', '--');

Kp = 147.3;
plot(Kp, (-(Kp-26)+sqrt((Kp-26).^2-4*(144+24.*Kp)))/2, 'marker', '^', 'Color', 'k')
xlabel('Kp'); ylabel('Real Part of Roots');
title('Optimal poles');
legend;
```

```
Warning: Imaginary parts of complex X and/or Y arguments ignored.
```



We can also show this mathematically by finding the Kp values where all our poles are in the negative half-plane (negative real part). To do this we look we analyze the denominator of our transfer function with the quadratic formula.

equation:

$$s^2 + (K_p - 26)s + (144 + 24Kp) = 0$$

We need the real parts of the roots to both be negative. The roots are found by the quadratic equation:

$$\frac{-(K_p - 26) \pm \sqrt{(K_p - 26)^2 - 4(144 + 24K_p)}}{2}$$

The strictly real part is $\frac{26-K_p}{2}$. If the discriminant $(K_p - 26)^2 - 4(144 + 24K_p)$ is negative or zero, then the real part is just $\frac{26-K_p}{2}$, and our system is obviously stable for all $K_p > 26$. If the discriminant is positive however, the root will be strictly real, and we need to verify that they are still negative, by checking that

$$-(K_p - 26) > \sqrt{(K_p - 26)^2 - 4(144 + 24K_p)}, \quad \forall K_p > 26$$

$$(-K_p + 26)^2 > (K_p - 26)^2 - 4(14 + 24K_p)$$

$$K_p{}^2 - 52K_p + 26^2 > K_p^2 - 52K_p + 26^2 - 576 - 96K_p$$

$$96K_p > 576$$

$$K_p > 6$$

This inequality will obviously also hold for $K_p > 26$. We have now shown that all the roots have negative real parts for all $K_p > 26$, so that our system is stable.

Verifying that $Kp = 26$ gives us zero-poles (marginally stable):

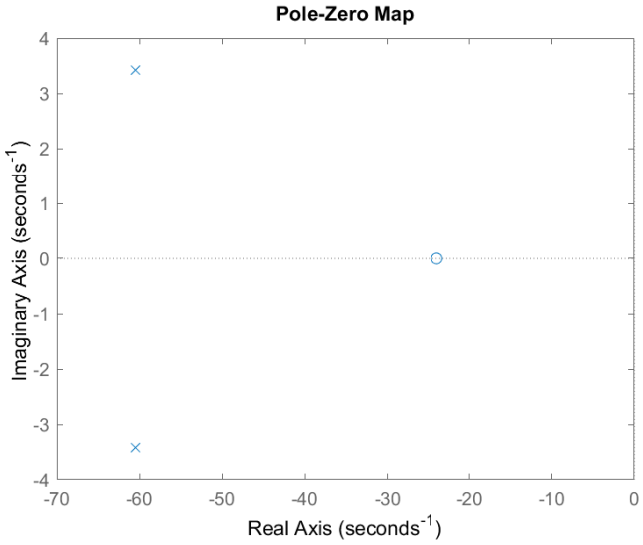
```
close;
Kp = 26;
closed_loop = (Kp*s+Kp*24)/(s^2+(Kp-26)*s+(144+24*Kp));
real(pole(closed_loop))
```

```
ans =

     0
     0
```

We can check the poles for our new controller with $Kp = 147.3$

```
Kp = 147;
closed_loop = (Kp*s+Kp*24)/(s^2+(Kp-26)*s+(144+24*Kp));
real(pole(closed_loop))
```

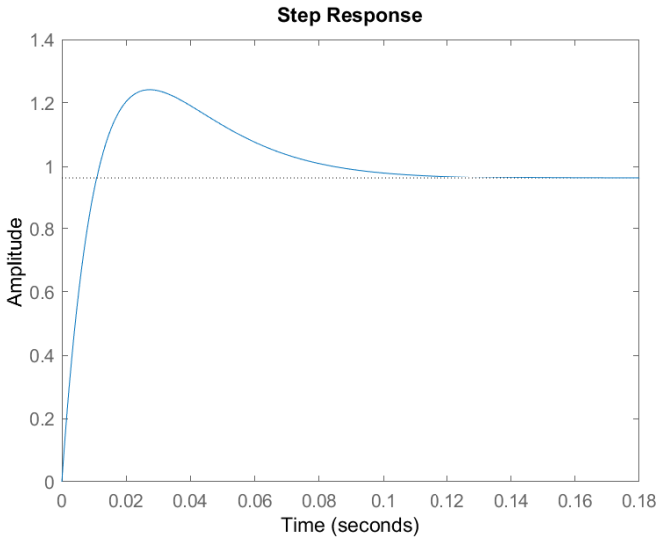
```
ans =

  -60.5000
  -60.5000


close;
pzplot(closed_loop);
```



Control system step response and steady-state error

We have verified that the poles of our new system are all negative real part. This means the system stable, and we can verify this by viewing the step response of the closed loop system:

```
step(closed_loop)
[y,t]=step(closed_loop); %save the output values to check steady state
SS_error = abs(1-y(end));
%verifying that the new system is stable
isstable(closed_loop)

ans =

  logical

   1
```



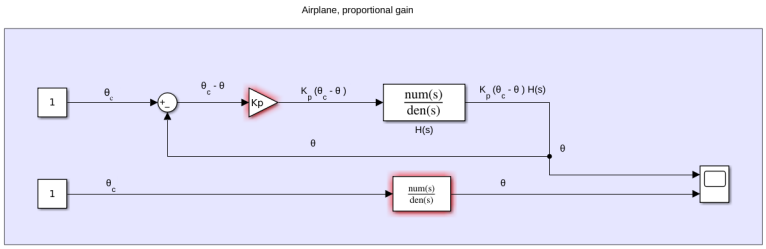
Thus we have verified that our control system is stable and reaches a steady state error of 3.89%, with a steady state value of

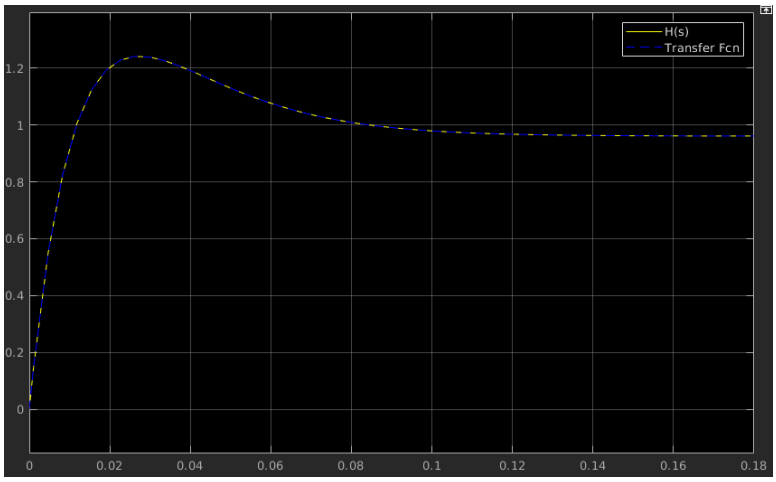
$$(1 - \frac{3.89}{100}) = 0.9611$$

We do have some overshoot here, and there is also a steady-state error in our system. The overshoot could be compensated for by introducing a derivative term, and the steady-state error could be eliminated by an integral term, giving us a full PID controller.

Simulink step response

Both the original simulink block diagram and the reduced transfer function show the same step response as the matlab code:




H(s)
Transfer Fcn
1.2
1
0.8
0.6
0.4
0.2
0
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0.18

Task 3

Rotary pendulum analysis and control

Contents


```
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.
```

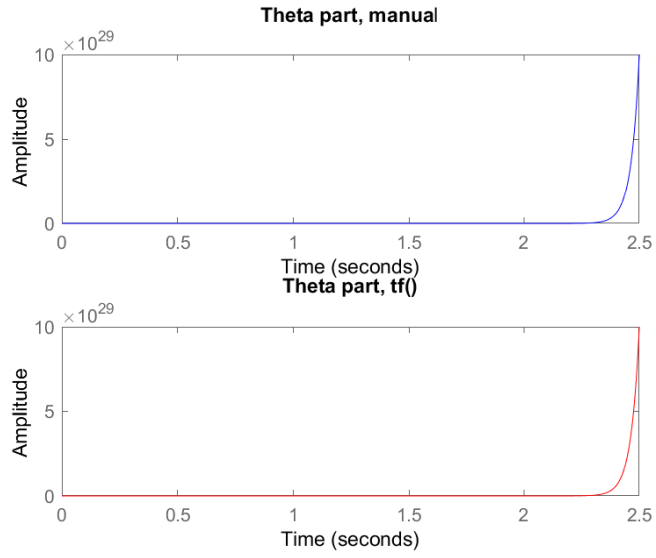
```
H_manual
```

```
H_manual =

  From input to output...
                 921 s^2 - 1.243e-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.
```

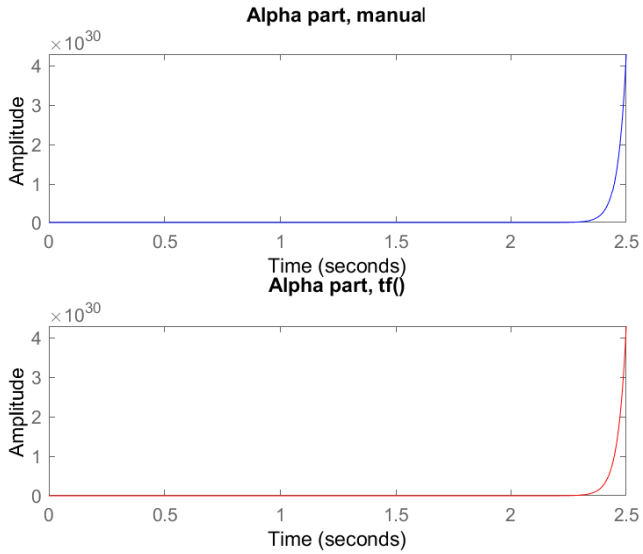
Verifying the two transfer functions

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

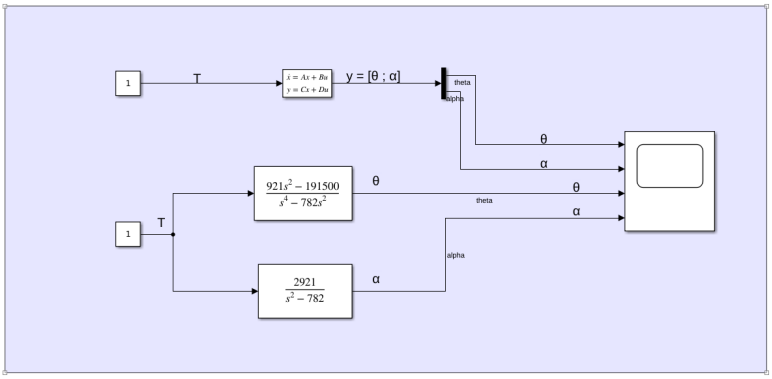
Before I move on, i remove the extremely small coefficients in the transfer functions, as they have virtually no impact. I verified this by checking that the poles didn't change.

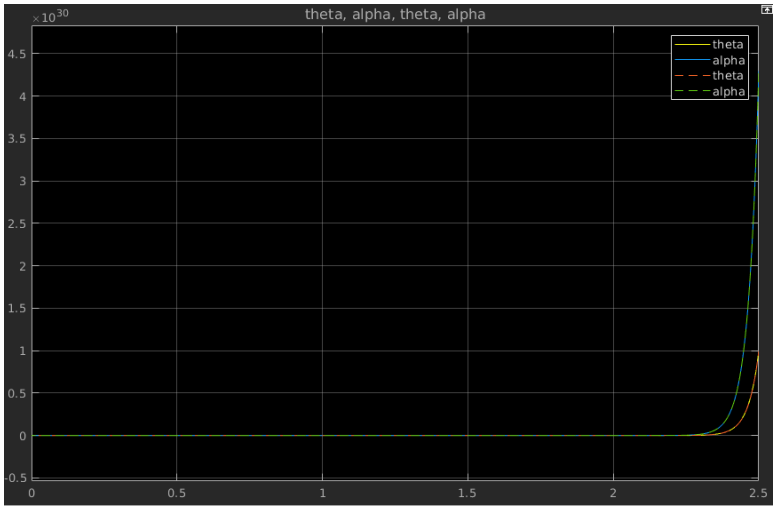
```
close;

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.





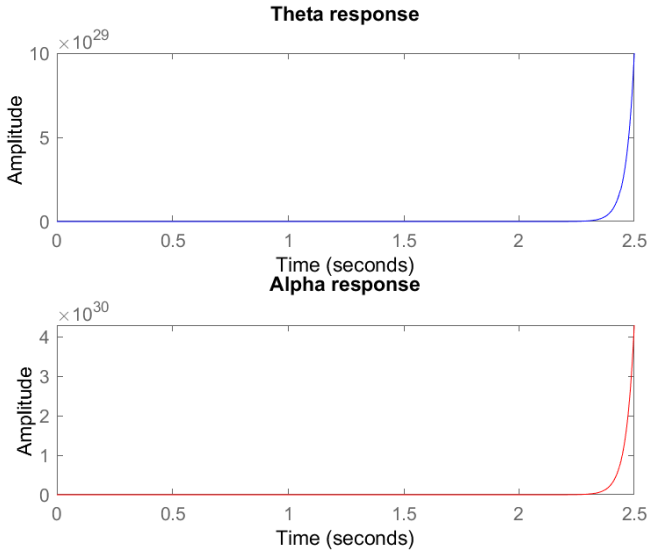
The responses from the two representations are identical, so we are ready to move on with the transfer function analysis. We end up with the following plant transfer functions for θ and α:

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

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

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')
```



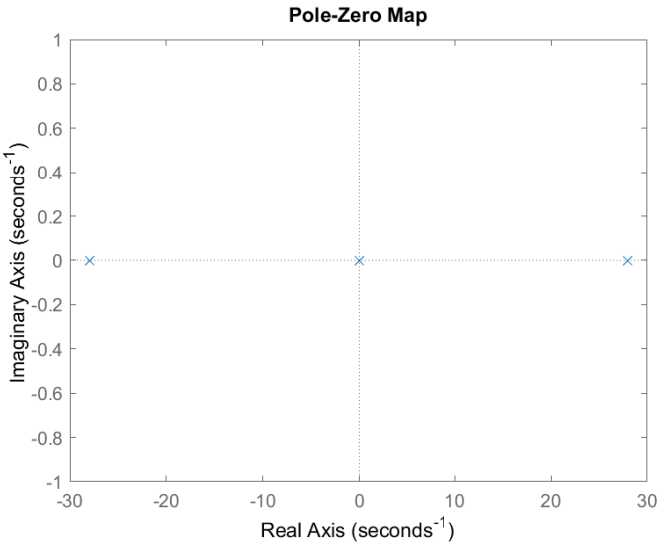
System stability and poles

The system is clearly unstable in both θ and α.

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:

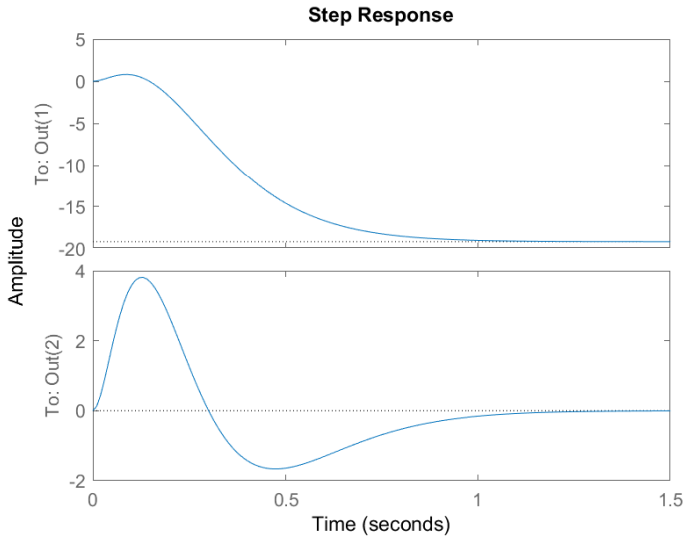
$$\dot{\vec{x}} = A\vec{x} + \vec{B}T$$

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: θ, bottom: α):

```
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}:

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

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

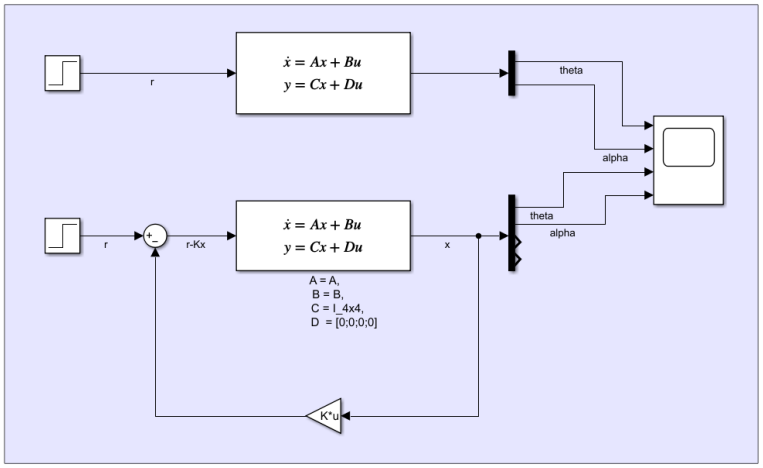
$$\vec{y} = C\vec{x}$$

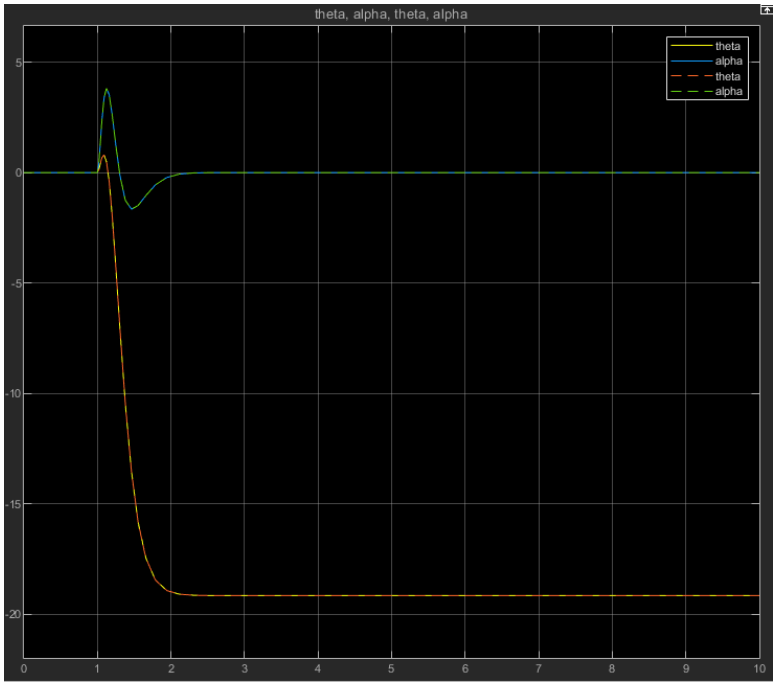
```
A_CL
```

```
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:



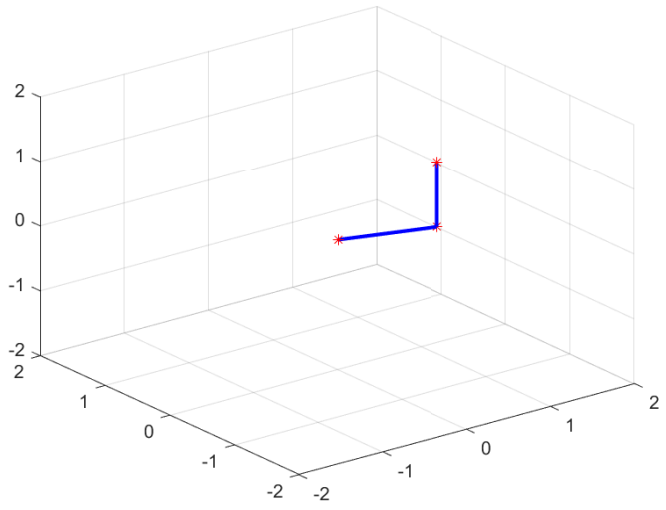


As we can see, the reduced system is identical to the original full-state feedback system.

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:

$$\int_0^\infty (x^T Q x + u^T R u) dt$$

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 = lqr(sys_ss, Q, R);
A_CL_lqr = A-B*K_lqr;
sys_lqr1 = ss(A_CL_lqr, B, C, D);
[y_lqr1, t] = step(10*sys_lqr1, sim_time);


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

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

% Lazy/economical controller:
Q = 10*eye(4);
R = 100;
K_lqr = lqr(sys_ss, Q, R);
A_CL_lqr = A-B*K_lqr;
sys_lqr4 = ss(A_CL_lqr, 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
```