Homework 1_Hannes Du

Thursday, October 5, 2023

1. Github Link: https://github.com/HannesDu/

2. (a)

$$G(s) = \frac{(s+2)(s-2)(s+5)(s-5)}{(s+1)(s-1)(s+3)(s-3)(s+6)(s-6)} = \frac{b(s)}{a(s)}$$

$$T = \frac{G(s)D(s)}{1+G(s)D(s)}$$

To place the poles of T(s) at s = {-1, -1, -3, -3, -6, -6}, we need:

$$1 + G(s)D(s) = (s+1)^2(s+3)^2(s+6)^2$$

Solving for D(s) gives:

$$1 + \frac{b(s)}{a(s)}D(s) = (s+1)^2(s+3)^2(s+6)^2$$

$$\boldsymbol{D(s) = \frac{a(s)(s+1)^2(s+3)^2(s+6)^2 - a(s)}{b(s)}}$$

Using MATLAB to substitute simplify values gives the controller polynomial:

$$\boldsymbol{D(s) = \frac{s^{12} + 20s^{11} + 108s^{10} - 344s^9 - 5626s^8 - 18144s^7 + 6731s^6 + 161352s^5 + 337087s^4 + 172044s^3 - 233649s^2 - 314928s - 104652}{(s+2)(s-2)(s+5)(s-5)}}$$

Finally, implementing the control with the plant G(s) gives $T(s) = \frac{G(s)D(s)}{1+G(s)D(s)}$:

```
T =
  RR_tf with properties:
num:   1.0e+03 *
    0.0010    0.0200    0.1540    0.5760    1.0890    0.9720    0.3230
den:   1.0e+03 *
    0.0010    0.0200    0.1540    0.5760    1.0890    0.9720    0.3240
Continuous-time transfer function
  m=6, n=6, n_r=n-m=0, semiproper, K=     1
  z:   -6.0644   -5.9308   -3.1630   -2.8275   -1.1080   -0.9062
  p:  -1.0000 - 0.0000i  -1.0000 + 0.0000i  -3.0000 - 0.0000i  -3.0000 + 0.0000i  -6.0000 + 0.0000i  -6.0000 + 0.0000i
```

As seen, the poles are located at s = {-1, -1, -3, -3, -6, -6}.

To verify with RR_diophantine:

```
x =
  RR_poly with properties:
poly:    -0.6710   -2.2869   10.1755   24.1641
roots:   -5.0009   -2.0030    3.5955
    n: 3
y =
  RR_poly with properties:
poly:     0.6710    2.2869  -21.5818  -62.0409   42.6886   81.5318
roots:   -6.0000   -3.0000   -1.0000    1.2681    5.3235
    n: 5
test =
  RR_poly with properties:
poly:    1.0e+03 *
    0.0010    0.0200    0.1540    0.5760    1.0890    0.9720    0.3240
roots:  -6.0000 - 0.0000i  -6.0000 + 0.0000i  -3.0000 - 0.0000i  -3.0000 + 0.0000i  -1.0000 - 0.0000i  -1.0000 + 0.0000i
    n: 6
residual =
   1.1369e-13
```

Indeed, the residual is approximately 0. Therefore, the controller works.

(b)
The degree of the numerator of D(s) is 12. The degree of the denominator of D(s) is 4.
Brute-force solving for D(s) gives:

$$T(s) = \frac{G(s)D(s)}{1+G(s)D(s)}$$

$$T(s) + T(s)G(s)D(s) = G(s)D(s)$$

$$D(s)[T(s)G(s) - G(s)] = -T(s)$$

$$D(s) = \frac{T(s)}{G(s)[1-T(s)]} = \frac{\frac{z^6}{p^6(s+20)^k}}{\frac{z^4}{p^6}\left[1-\frac{z^6}{p^6(s+20)^k}\right]} = \frac{\frac{z^2}{(s+20)^k}}{1-\frac{z^6}{p^6(s+20)^k}} = \frac{z^2}{(s+20)^{k-1}}$$

Therefore, in order for this particular controller D(s) to be proper, k must be greater than or equal to 3.

Using MATLAB, the new controller D(s) has a numerator polynomial to the 18th degree and a denominator polynomial to the 19th degree (strictly proper):

```
D2 =
  RR_tf with properties:
num:   1.0e+11 *
  Columns 1 through 5
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
  Columns 6 through 10
   0.0001 + 0.0000i   0.0003 + 0.0000i  -0.0017 - 0.0000i  -0.0227 - 0.0000i  -0.1090 + 0.0000i
  Columns 11 through 15
  -0.2232 + 0.0000i   0.2465 - 0.0000i   2.7367 - 0.0000i   7.2084 - 0.0000i   8.4976 - 0.0000i
  Columns 16 through 19
   1.2106 + 0.0000i  -8.2780 + 0.0000i  -8.5530 + 0.0000i  -2.7126 + 0.0000i
den:   1.0e+15 *
  Columns 1 through 5
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 - 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
  Columns 6 through 10
   0.0000 + 0.0000i   0.0000 - 0.0000i   0.0000 - 0.0000i   0.0001 + 0.0000i   0.0003 + 0.0000i
  Columns 11 through 15
  -0.0017 + 0.0000i  -0.0198 - 0.0000i  -0.0920 - 0.0000i  -0.2317 + 0.0000i  -0.2296 - 0.0000i
  Columns 16 through 20
   0.4390 + 0.0000i   1.9019 + 0.0000i   2.9370 + 0.0000i   2.2186 + 0.0000i   0.6725 + 0.0000i
Continuous-time transfer function
  m=18, n=19, n_r=n-m=1, strictly proper, K=     1
```

Then, testing with RR_diophantine with the new f(s) polynomial from of T(s) gives:

```
test =
  RR_poly with properties:
poly:    1.0e+06 *
    0.0000    0.0001    0.0026    0.0418    0.3804    1.9895    5.9734    9.8978    8.1648    2.5920
roots:  Columns 1 through 5
 -20.0047 + 0.0000i -19.9976 - 0.0041i -19.9976 + 0.0041i  -6.0000 + 0.0000i  -6.0000 + 0.0000i
  Columns 6 through 9
  -3.0000 - 0.0000i  -3.0000 + 0.0000i  -1.0000 - 0.0000i  -1.0000 + 0.0000i
    n: 9
residual =
   1.4529e+07
```

With the controller I designed in 2(a), k only needed to be greater than or equal to 3 in order for D(s) to be proper. In the context of the Diophantine equation, the choice of k should indeed be sufficiently large.

However, in this case, we want to match the poles of the closed-loop system. By adding more poles at s=−20 in the target transfer function f(s), we are effectively introducing a more aggressive control action. While adding more poles in the controller might ensure that the system matches the desired poles, it can also lead to a highly aggressive controller. An overly aggressive controller can result in instability or reduced robustness to uncertainties or variations in the system's parameters.

3.

```matlab
%% Homework 1 Problem 3
% Test HBD_C2D_matched
syms s z1 p1;
bs = s + z1;
as = s * (s + p1);
h = 0.1;
omega_bar = 0.5;        % Set your desired critical frequency
strictly_causal = true; % Set to true for a strictly causal D(z)

[bz_custom, az_custom] = HBD_C2D_matched(bs, as, h, omega_bar, strictly_causal);
disp("Strictly Causal with omega_bar = 0.5:");
pretty(bz_custom);
pretty(az_custom);

% Now, compare with MATLAB's 'matched' option
sys = tf(bs, as);
sys_d = c2d(sys, h, 'matched');
disp("MATLAB's 'matched' option:");
pretty(tf(sys_d));

% Function HBD_C2D_matched
function [bz, az] = HBD_C2D_matched(bs, as, h, omega_bar, strictly_causal)
% Convert D(s) to D(z) using matched z-transform.
% Inputs:
%   bs: Coefficients of the numerator polynomial of D(s).
%   as: Coefficients of the denominator polynomial of D(s).
%   h: Time step for discretization.
%   omega_bar (optional): Critical frequency of interest (default: 0).
%   strictly_causal (optional): If true, forces a strictly causal D(z) (default: false).
% Outputs:
%   bz: Coefficients of the numerator polynomial of D(z).
%   az: Coefficients of the denominator polynomial of D(z).

if nargin < 4
    omega_bar = sym(0); % Default value for omega_bar as a symbolic variable
end

if nargin < 5
    strictly_causal = false; % Default value for strictly_causal
end

n = length(bs) - 1;
m = length(as) - 1;

bz = sym(zeros(1, n + 1));  % Initialize symbolic arrays
az = sym(zeros(1, m + 1));  % Initialize symbolic arrays
z = sym('z');  % Define z as a symbolic variable

for j = 0:n
    bz(j + 1) = bs(n - j + 1) * z^j;  % Create symbolic expressions
end

for j = 0:m
    if j == m && strictly_causal
        az(j + 1) = as(m - j + 1) * (z - 1)^j;  % Map an infinite zero to z = 1 for strictly causal
    else
        az(j + 1) = as(m - j + 1) * z^j;  % Map other poles and zeros
    end
end

% Normalize the resulting polynomials
bz = bz / az(1);
az = az / az(1);
end
```

For z1 = 1, p1 = 10, we have for D(z):

```
ans =
(z + 1)/(z*(z + 10))
```

So, for the given transfer function with z1 = 1 and p1 = 10, and with strictly_causal = true, the manual calculation for D(z) is 0.

Using MATLAB'S C2D function gives:

```
MATLAB's 'matched' option:


ans =


  0.6643 z - 0.601
  ----------------
     z - 0.3679
```

My custom MATLAB code offers control and transparency but may be more complex and lacks some built-in optimizations. MATLAB's c2d function is efficient and widely applicable but provides less control. The choice depends on my specific needs:

If I need fine-tuned control and have time to develop, the custom code is suitable.

If efficiency, standardization, and ease of use matter more, MATLAB's c2d function is better.