HW10

November 6, 2022

Group number
3

Group members
Alexander Stoustrup
Mathias Tyranski
Benjamin Simonsen

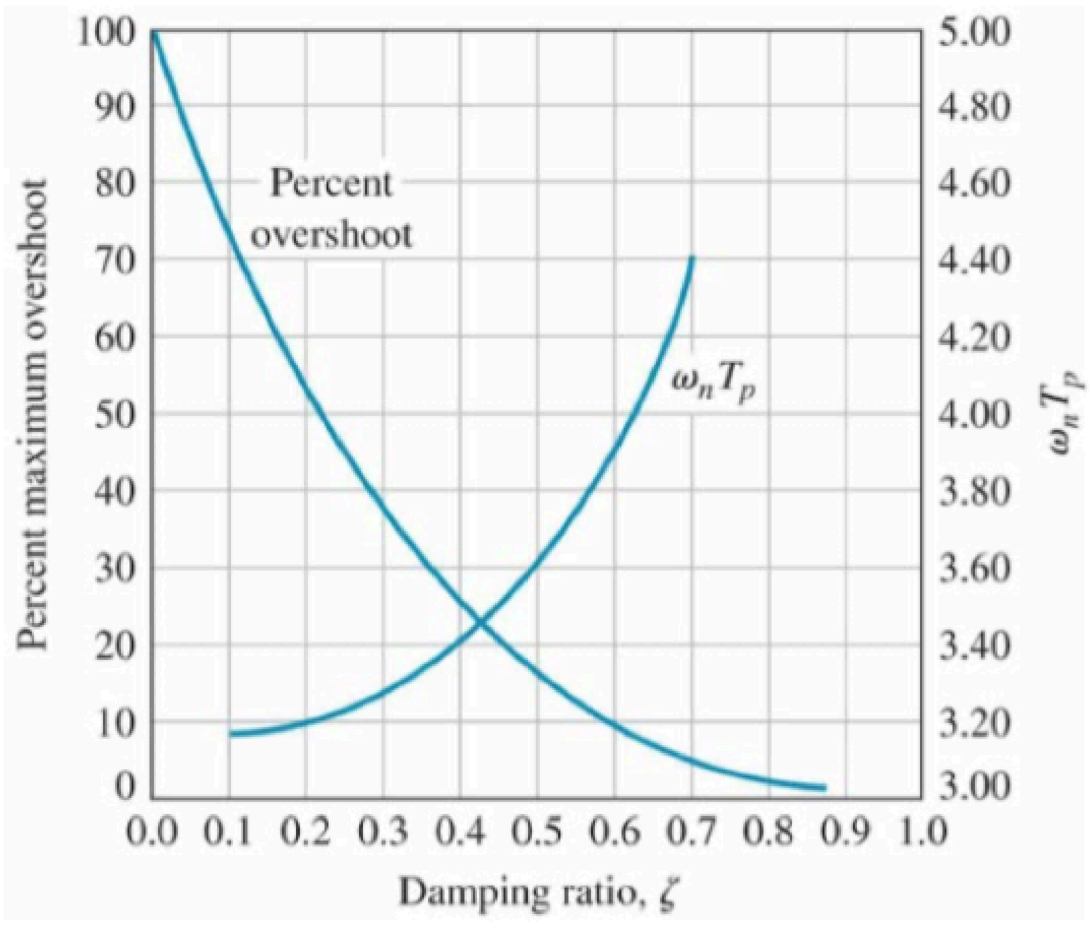
[1]:
```python
import numpy as np
from scipy.optimize import minimize
import scipy.signal as si
import sympy as sp
import control as ct
from typing import List
from sympy.plotting import plot
import matplotlib.pyplot as plt
from IPython.display import display, Latex, Math, Image
%matplotlib inline

def eq_disp(varstring, expr, unit=""):
    display(Latex(f"${varstring}={sp.latex(expr)} \: {unit}$"))

def reduce_feedback(G_fwd, G_bwd):
    """Assumes feedback is deducted from signal, if not
    change sign of feedback"""
    return sp.simplify(G_fwd/(1+G_fwd*G_bwd))

def RHarray(coeffs: List):
    # first 2 rows from coefficients
    n = len(coeffs)
    arr = sp.zeros(n, n//2+2)
    i = 0
    for i in range(0,n,2):
        arr[0, i//2] = coeffs[i]
    for i in range(1,n,2):
        arr[1, i//2] = coeffs[i]

    for j in range(2, arr.shape[0]):
        for i in range(arr.shape[1]-1):
            a0 = arr[j-2,0]
            a3 = a1 = arr[j-1,i+1]
            a1 = arr[j-1,0]
            a2 = arr[j-2,i+1]
            arr[j, i] = (a1*a2-a0*a3)/a1
    return arr
```

1 AP10.4

Find the damping ratio from the figure

[2]:
```
Image('PO.png')
```

[2]:



ω_n can be calculcated from the settling time

```
[3]: zeta = 0.7
     omega = 4/zeta
```

Calculate the loop function $L(s)$

```
[4]: K1, K2, s = sp.symbols("K_1, K_2, s")
     G = 8/(s*(s+8))
     L = reduce_feedback(K1*G, K2*s)
     eq_disp('L(s)',L)
```

$L(s) = \frac{8K_1}{s(8K_1K_2+s+8)}$

Since the loop function is a second order system we can directly solve for K_1 and K_2 by comparing it to a second order system with the specified damping ratio and natural frequency.
The target second order system is:

```
[5]: msd_sys = omega**2/(s*(s+2*zeta*omega))
     msd_sys
```

[5]: $$\frac{32.6530612244898}{s\,(s+8.0)}$$

K_1 and K_2 is then

```
[6]: K1, K2 = sp.solve(L-msd_sys, [K1, K2])[0]
     eq_disp('K_1', K1)
     eq_disp('K_2', K2)
```

$K_1 = 4.08163265306122$

$K_2 = 0.0$

2 AP10.8

2.1 Step 1: Find pole locations

We first translate the requirements into desired pole locations

```
[7]: PO = 20
     Tr = 0.5
     Ts = 1.2
     Kv = 10
```

```
[8]: zeta, omega = sp.symbols('zeta, omega_n')
     zeta = float(sp.solve(100*sp.exp(-zeta*sp.pi/sp.sqrt(1-zeta**2))-PO, zeta)[0])
     omega = float(sp.solve((4/(omega*zeta)-Ts))[0])
```

```
[9]: zeta
```

```
[9]: 0.4559498107691261
```

```
[10]: omega
```

```
[10]: 7.310746171185917
```

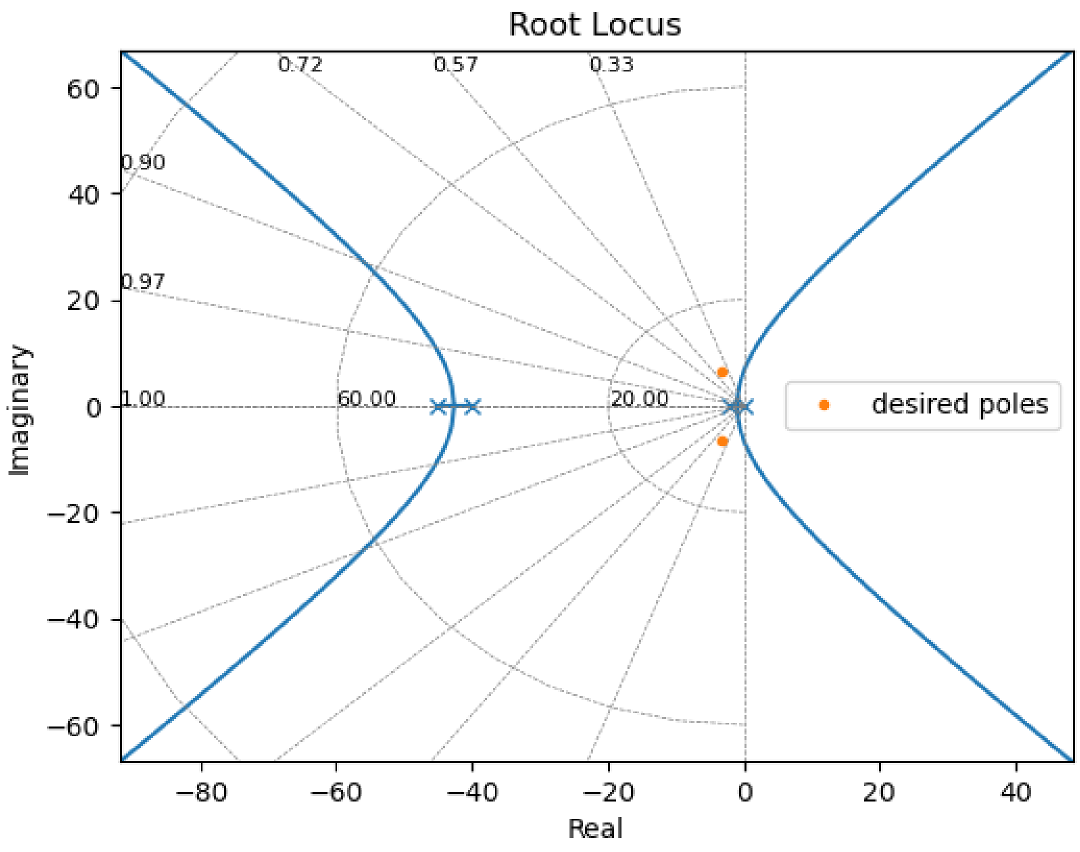
```
[11]: p_goal = -complex(zeta*omega, -omega*np.sqrt(1-zeta**2))
      p_goal
```

```
[11]: (-3.333333333333332+6.506604219437235j)
```

2.2 Step 2: Uncompensated root locus

```
[12]: s = ct.tf('s')
      G = 250/(s*(s+2)*(s+40)*(s+45))
      rl = ct.rlocus(G)
      handle = plt.plot([p_goal.real, p_goal.real], [p_goal.imag,-p_goal.imag],␣
       ↪marker='.', label='desired poles', linestyle="None")
      plt.legend(handles=handle)
```

```
[12]: <matplotlib.legend.Legend at 0x7fe12d4c7be0>
```



We can see the desired poles can not be obtained without compensation

2.3 Step 3: Placing zero directly under the desired root location

```
[13]: z = p_goal.real
      z
```

```
[13]: -3.333333333333332
```

2.4 Step 4: Use angle criteria to determine the pole location

```
[14]: p_angle = np.angle(p_goal-G.poles())
      p_angle
```

```
[14]: array([0.15490742, 0.17562463, 1.77291815, 2.04423545])
```

```
[15]: angle_criterion = lambda x: np.pi - (sum(p_angle)+np.angle(p_goal - x) - np.
       ↪angle(p_goal - z))
```

```
[16]: from scipy.optimize import fsolve
      p = fsolve(angle_criterion, 0.5)[0]
      p
```

```
[16]: -13.603875337607759
```

2.5 Step 5: Determine the gain using magnitude criterion

```
[17]: P = G*(s-p_goal.real)/(s-p)
```

```
[18]: K = 1/abs(P(p_goal))
      K
```

```
[18]: 569.9604671830552
```

2.6 Step 6: Check if requirements are met

Requirements were not met the first time due to the overshoot being too large, therefore the pole is moved further to the left and the gain is adjusted, after which the requirements were met.

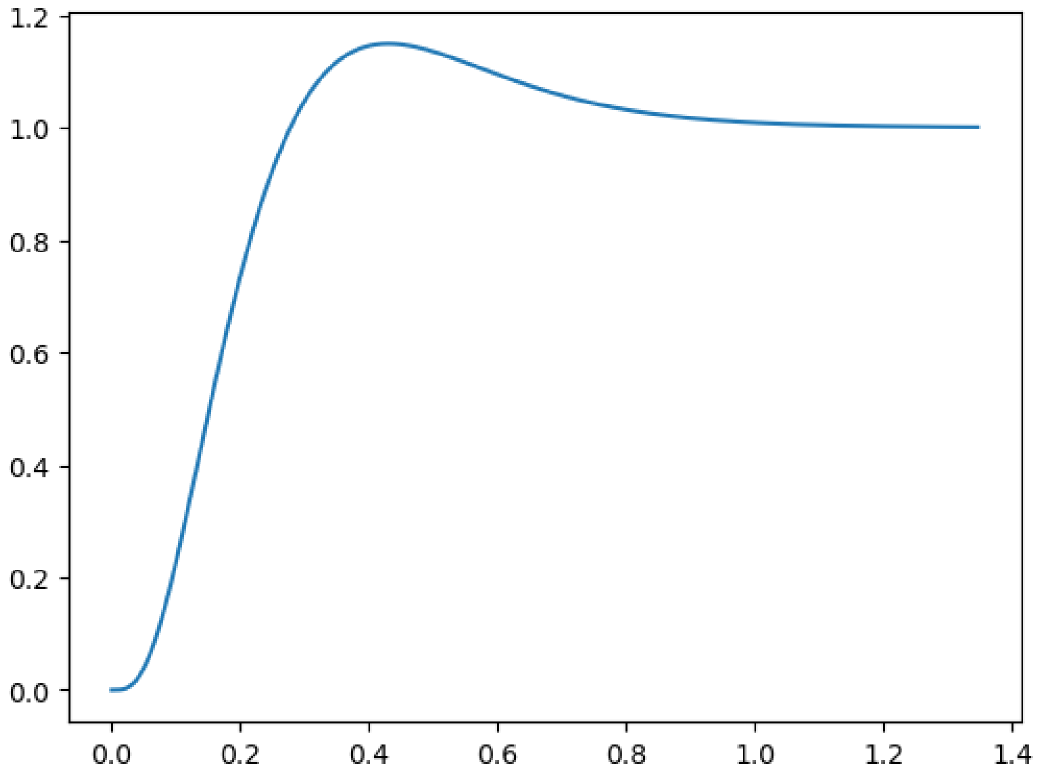
```
[19]: z = p_goal.real
      p = p - 30
      P = G*(s-z)/(s-p)
```

```
[20]: K = 1/abs(P(p_goal))
      K
```

```
[20]: 1912.3251691919936
```

```
[21]: %matplotlib inline
      T = ct.feedback(K*P)
      plt.plot(*ct.step_response(T))
      ct.step_info(T)
```

```
[21]: {'RiseTime': 0.16929206244434314,
       'SettlingTime': 0.8818347730309815,
       'SettlingMin': 0.9003798074766735,
       'SettlingMax': 1.150162564932385,
       'Overshoot': 15.016256493238501,
       'Undershoot': 0,
       'Peak': 1.150162564932385,
       'PeakTime': 0.42954702411251244,
       'SteadyStateValue': 1.0}
```



Velocity constant K_v is given by

```
[22]: s_sym = sp.symbols('s')
      Gs = 250/(s_sym*(s_sym+2)*(s_sym+40)*(s_sym+45))
      Gcs = K*(s_sym-z)/(s_sym-p)
      eq_disp('K_v', sp.N(sp.limit(s_sym*Gs*Gcs, s_sym, 0),3))
```

$K_v = 10.2$

3 P10.32

3.1 a)

```
[23]: Ksym, s = sp.symbols('K, s')
      Kv = 100
      Gc = Ksym
      G = (s + 500)/(s*(s + 0.0325)*(s**2 + 2.57*s + 6667))
```

So the loop function is

```
[24]: L = Gc*G
      eq_disp('L', L)
```

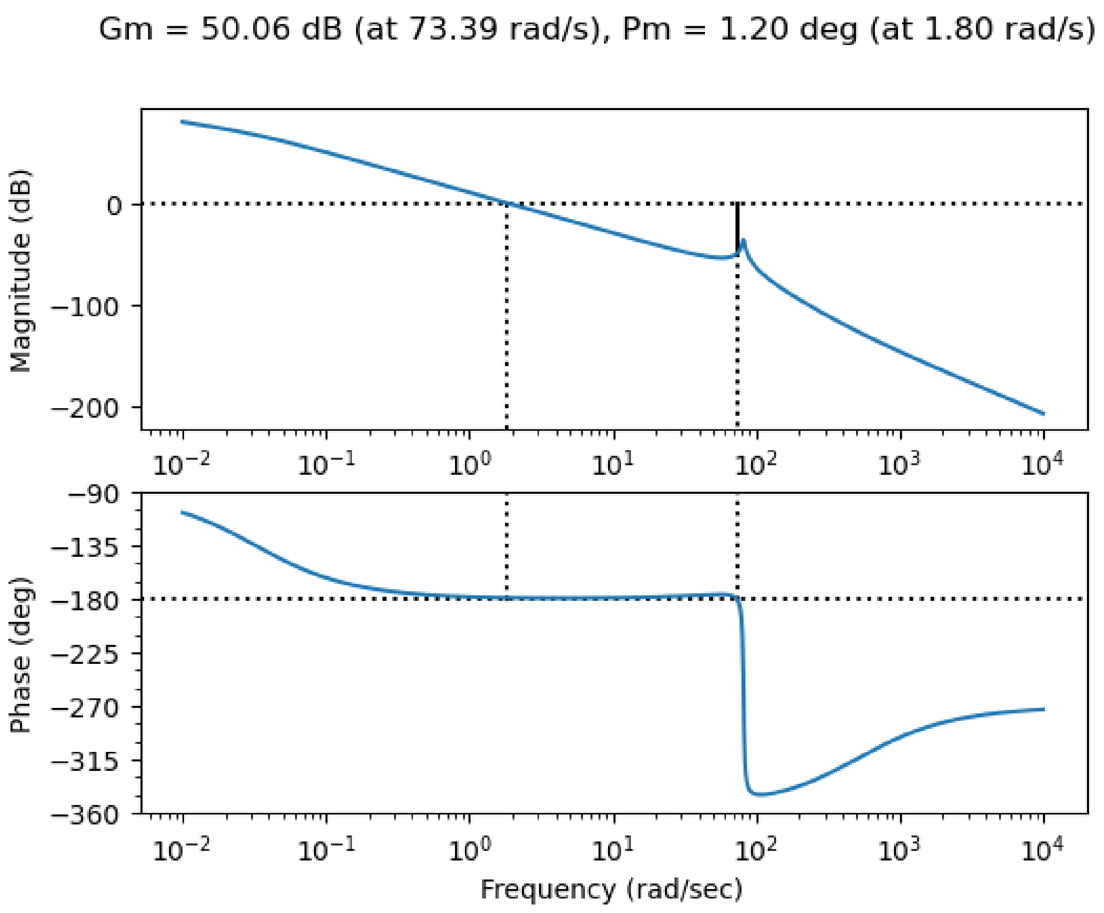
$$L = \frac{K(s+500)}{s(s+0.0325)(s^2+2.57s+6667)}$$

```
[25]: K = float(sp.solve(sp.limit(sp.simplify(s*L), s, 0) - Kv, Ksym)[0])
      eq_disp('K', K)
```

$K = 43.3355$

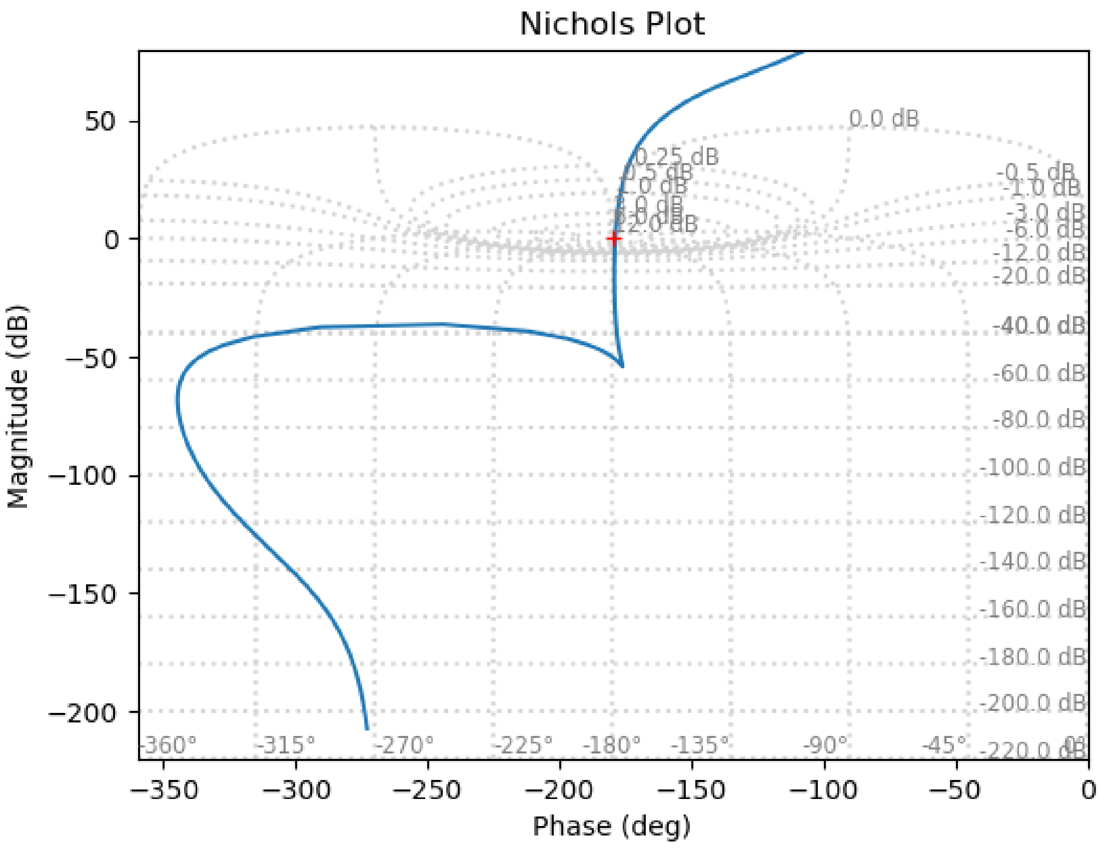
Getting the phase and gain margin and plotting the bode plot

```
[26]: s = ct.tf('s')
      Gc = K
      G = (s + 500)/(s*(s + 0.0325)*(s**2 + 2.57*s + 6667))
      L = Gc*G
      gm, pm, wcg, wcp = ct.margin(L)
      mag, phase, omega = ct.bode(L, dB=True, margins=True)
```



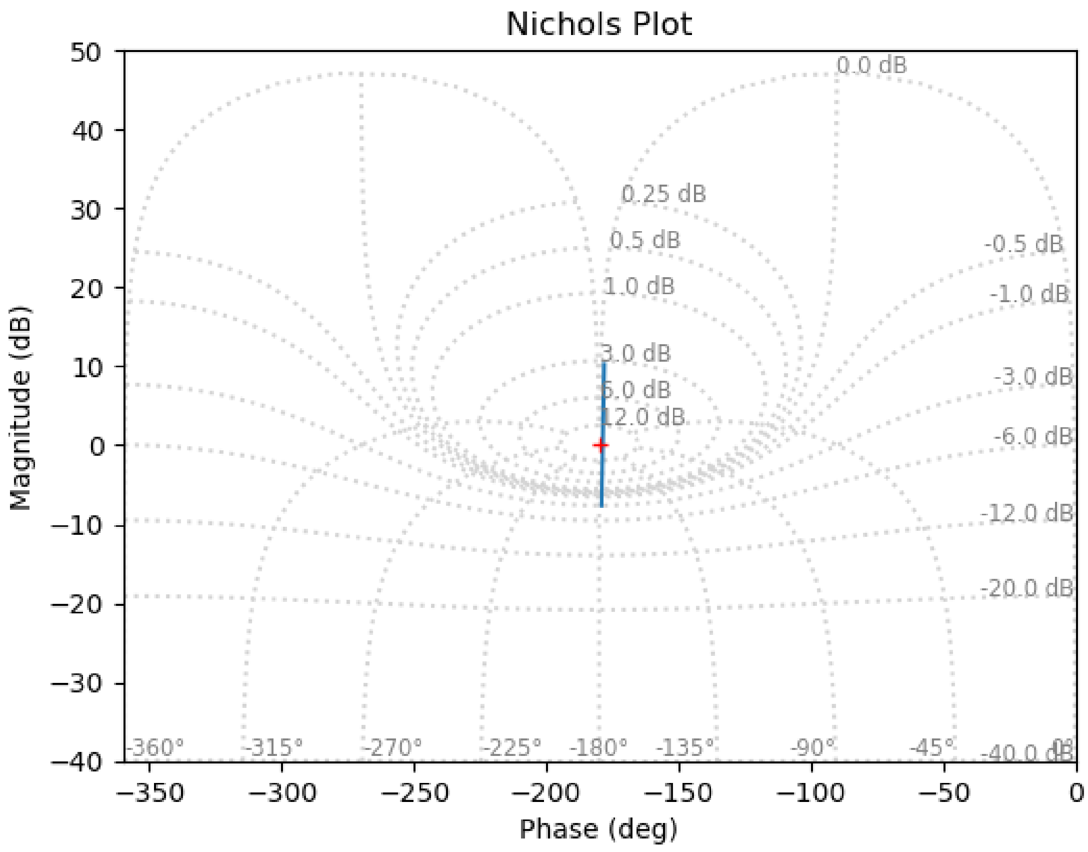
3.2 b)

[27]:
```
ct.nichols(L)
```



The Nichols plot does not display higher than 12 dB so $M_{p\omega}$ and ω_r cannot be read

[28]:
```
ct.nichols(L, omega=[1, 2.8])
```



So the bandwidth is (where it crosses -3 dB)

[29]:
```
omega_B = 2.8
eq_disp('\omega_B', omega_B, 'rad/s')
```

$\omega_B = 2.8\, rad/s$

3.3 c)

3.3.1 1. Evaluate the uncompensated system phase margin

[30]:
```
eq_disp('\phi_{pm}', pm, 'deg')
```

$\phi_{pm} = 1.19940463865854\, deg$

3.3.2 2. Determine necessary additional phase lead

[31]:
```
phi_pm = 35
phi_m = phi_pm - pm
eq_disp('\phi_m', phi_m, 'deg')
```

$\phi_m = 33.8005953613415\, deg$

3.3.3 3. Evaluate α

[32]:
```
alphaSym = sp.symbols('alpha')
alpha = float(sp.solve(sp.sin(phi_m) - (alphaSym - 1)/(alphaSym + 1),␣
 ↪alphaSym)[0])
eq_disp('\\alpha', alpha)
```

$\alpha = 5.38342289257295$

[33]:
```
float(-10*sp.log(alpha, 10))
```

[33]:
```
-7.310584969802832
```

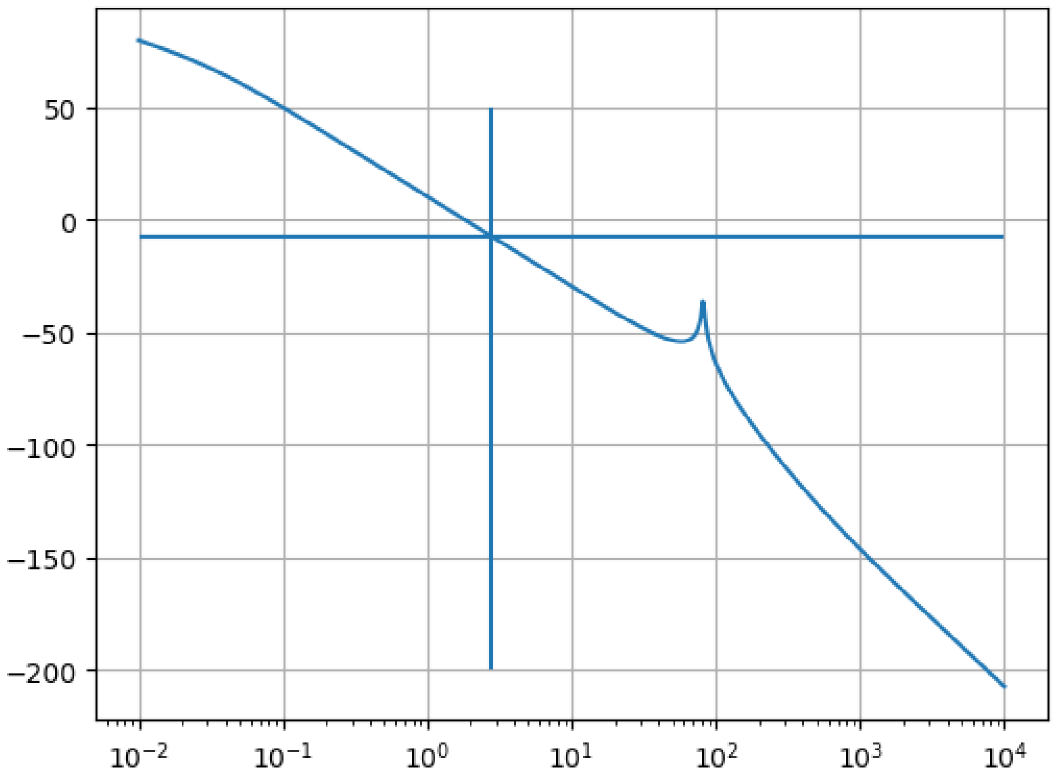
3.3.4 4. Determine the frequency ω_m where the uncompensated magnitude curve is equal to $-10log(\alpha)$ dB

[34]:
```
exp = float(-10*sp.log(alpha, 10))

for i, m in enumerate(20*np.log10(mag)):
    if m >= exp:
        index = i

omega_m = omega[index]
omega_m
```

[34]:
```
2.7443433032283653
```

Plot the magnitude to check

[35]:
```
plt.plot(omega,20*np.log10(mag))
plt.hlines(float(-10*sp.log(alpha, 10)), 10**(-2), 10**(4))
plt.vlines(omega_m, -200, 50)
plt.xscale('log')
plt.grid()
```



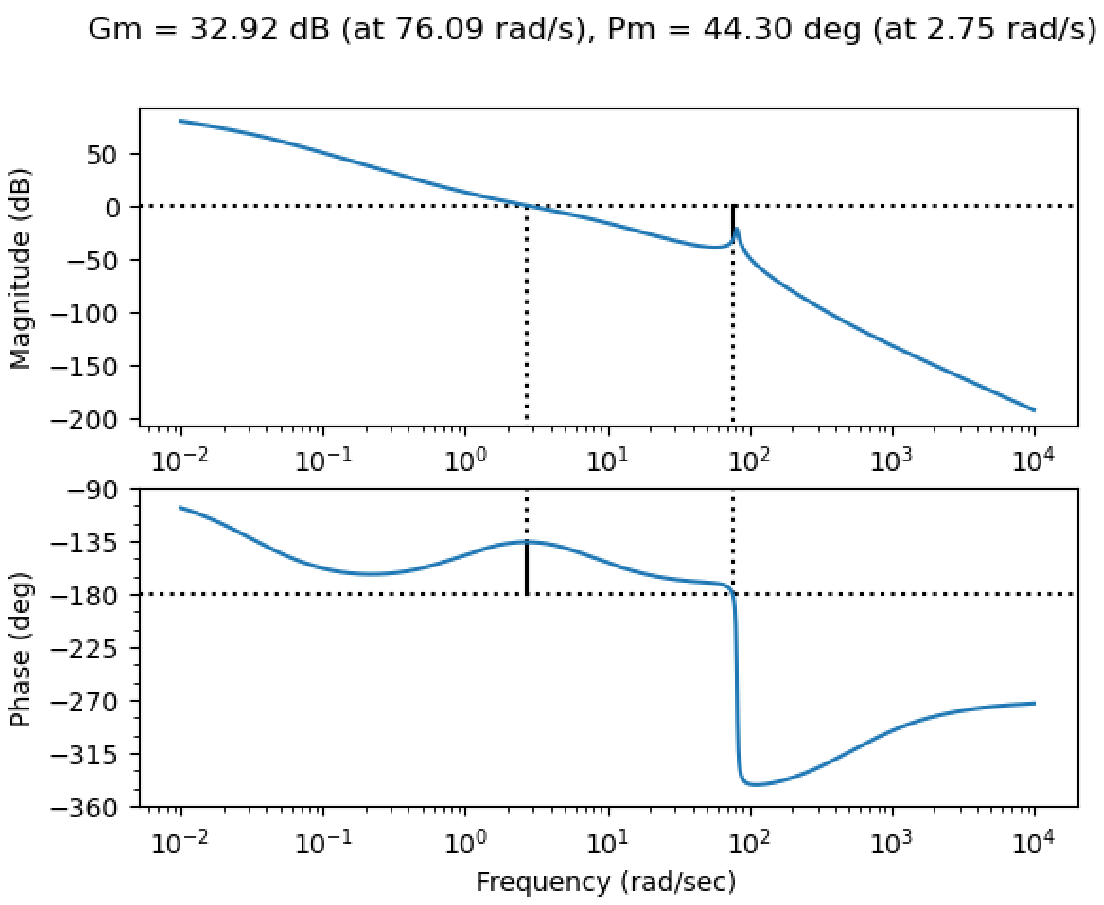
3.3.5 5. Calculate the pole and zero

[36]:
```
p = omega_m*np.sqrt(alpha)
z = p/alpha
```

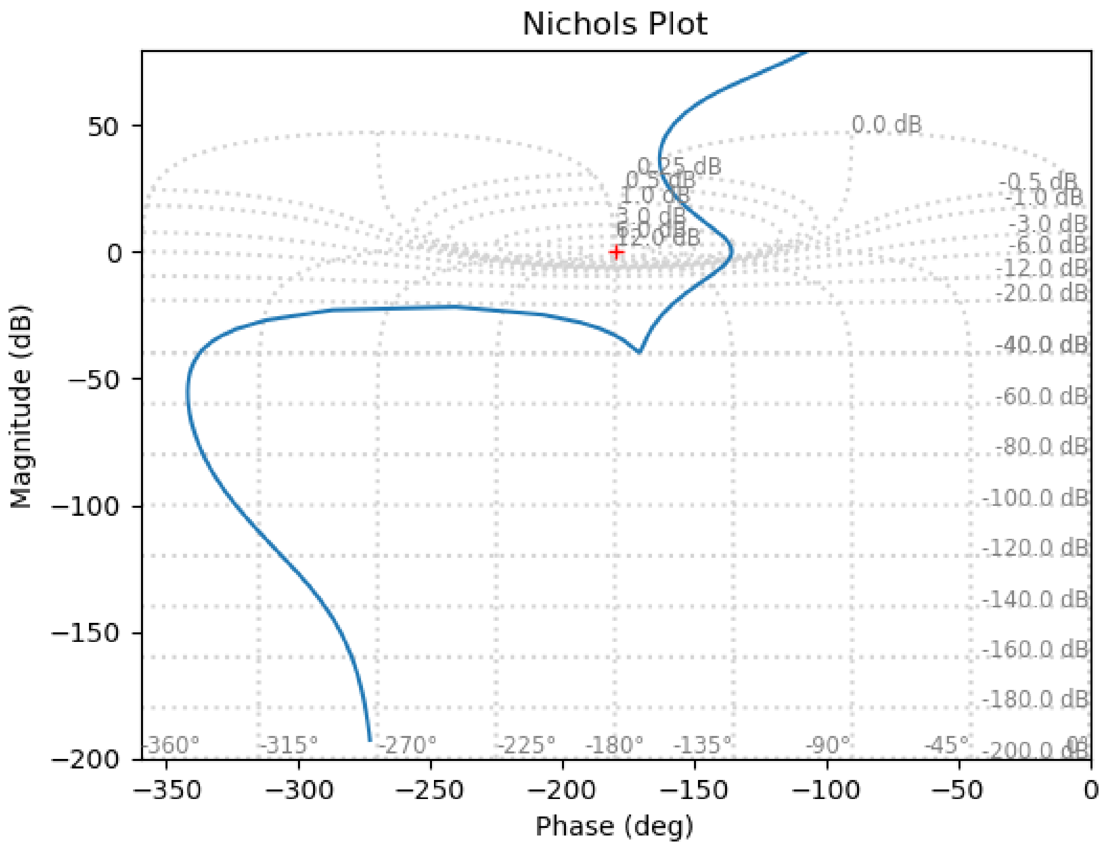
3.3.6 6. Draw the compensated system frequency response and check phase margin

Drawing the Bode plot and displaying the phase and gain margin

[37]:
```
s = ct.tf('s')
G_comp = alpha*(s + z)/(s + p)
Lc = G_comp*L
mag_c, phase_c, omega_c = ct.bode(Lc, dB=True, margins=True)
```



```
[38]: %matplotlib inline
      ct.nichols(Lc)
```



[39]:
```
%matplotlib qt
ct.nichols(Lc, omega=[4.5, 4.7])
```

From the nichols plot the resonant peak, resonant frequency and bandwidth is found

[40]:
```
M_pw = 4
omega_r = 1.9
omega_B = 4.6
eq_disp('M_{p\\omega}', M_pw, 'dB')
eq_disp('\\omega_r', omega_r, 'rad/s')
eq_disp('\\omega_B', omega_B, 'rad/s')
```

$M_{p\omega} = 4\,dB$

$\omega_r = 1.9\,rad/s$

$\omega_B = 4.6\,rad/s$

4 DP10.1

We will start with the lead compensator design to meet settling time and overshoot requirements, and then subsequently add the lag compensator to meet the error constant requirement

```
[41]: %matplotlib inline
      s = ct.tf('s')
      G = 20/(s*(s + 2))
```

4.1 Step 1: Find pole locations

We first translate the requirements into desired pole locations. A safetyfactor is applied to the requirements to make sure they the design is within the boundaries

```
[42]: fs = 1.2  #safety factor
      PO = 15/fs
      Ts = 1/fs
      Kv = 1/0.02
```

```
[43]: zeta, omega = sp.symbols('zeta, omega_n')
      zeta = float(sp.solve(100*sp.exp(-zeta*sp.pi/sp.sqrt(1-zeta**2))-PO, zeta)[0])
      omega = float(sp.solve((4/(omega*zeta)-Ts))[0])
      eq_disp('\zeta', zeta)
      eq_disp('\omega', omega)
```

$\zeta = 0.551949300834197$

$\omega = 8.69645091088158$

Desired pole location

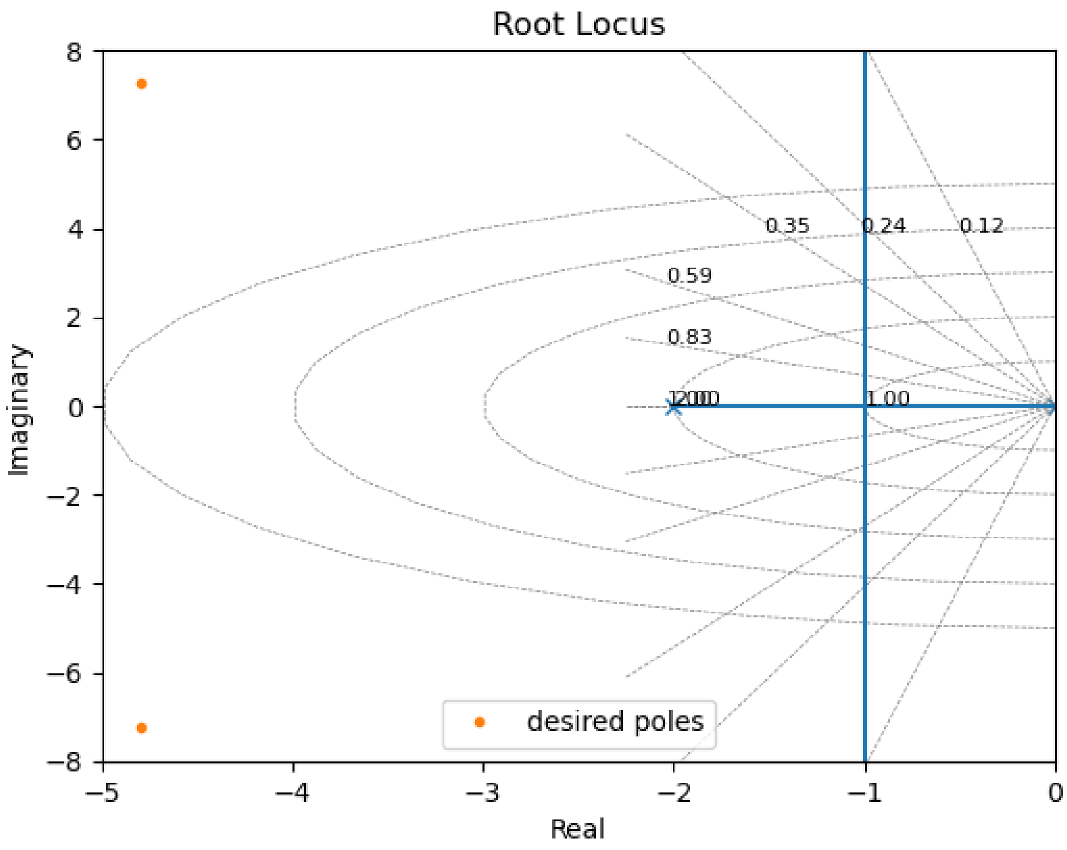
```
[44]: p_goal = -complex(zeta*omega, -omega*np.sqrt(1-zeta**2))
      p_goal
```

```
[44]: (-4.800000000000003+7.251776226923514j)
```

4.2 Step 2: Uncompensated root locus

```
[45]: rl = ct.rlocus(G)
      handle = plt.plot([p_goal.real, p_goal.real], [p_goal.imag,-p_goal.imag],␣
       ↪marker='.', label='desired poles', linestyle="None")
      plt.legend(handles=handle)
      plt.xlim(-5, 0)
      plt.ylim(-8, 8)
```

```
[45]: (-8.0, 8.0)
```



We can see the desired poles can not be obtained without compensation

4.3 Step 3: We cancel the minimum valued system pole with our controller zero

```
[46]: z = float(np.min(G.poles()))
```

```
/var/folders/mr/8dv953yx4cg4k1jrzprcqlzw0000gn/T/ipykernel_10597/51982335.py:1:
ComplexWarning: Casting complex values to real discards the imaginary part
  z = float(np.min(G.poles()))
```

4.4 Step 4: Use angle criteria to determine the pole location

Angles contributed by system poles

```
[47]: p_angle = np.angle(p_goal-G.poles())
```

Angle criterion equation

```
[48]: angle_criterion = lambda x: np.pi - (sum(p_angle)+np.angle(p_goal - x) - np.
        ↪angle(p_goal - z))
```

Solve for pole location

```
[49]: from scipy.optimize import fsolve
      p = fsolve(angle_criterion, 0.5)[0]
      p
```

```
[49]: -9.600000000000005
```

4.5 Step 5: Determine the gain using magnitude criterion

```
[50]: P = G*(s-z)/(s-p)
```

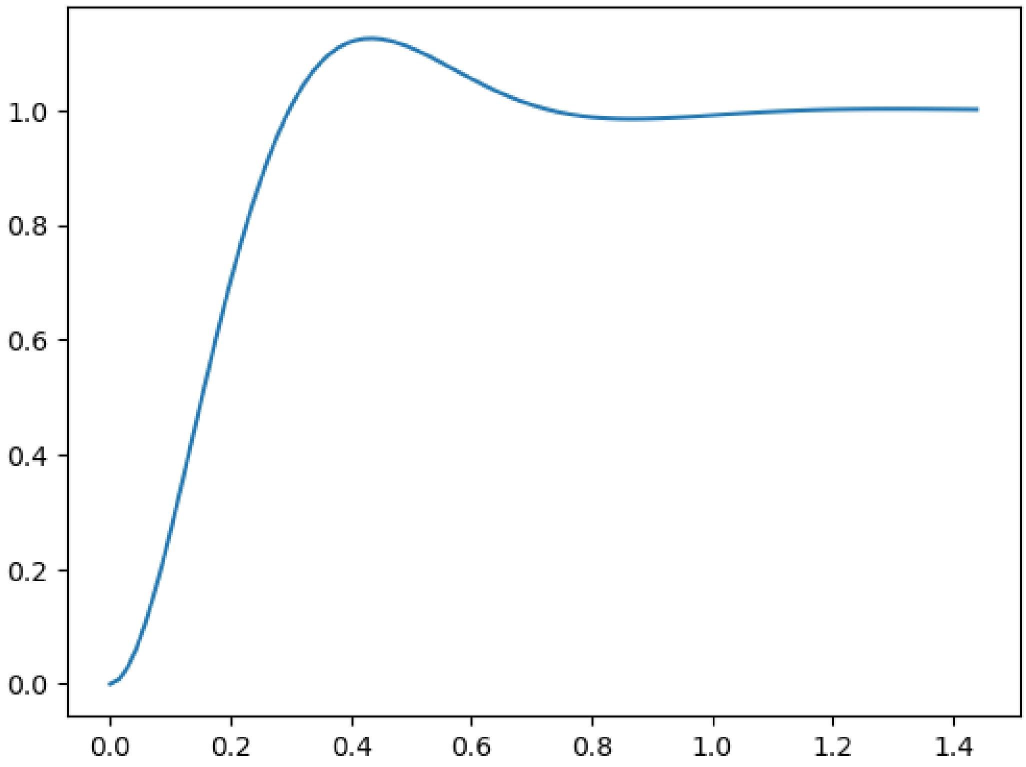
```
[51]: K = 1/abs(P(p_goal))
      K
```

```
[51]: 3.7814129222686534
```

4.6 Step 6: Plot the result and verify that specifications are met

```
[52]: Glead = K*(s-z)/(s-p)
      T = ct.feedback(Glead*G)
      plt.plot(*ct.step_response(T))
      ct.step_info(T)
```

```
[52]: {'RiseTime': 0.20351130872422116,
       'SettlingTime': 0.6832165364313139,
       'SettlingMin': 0.912851576803955,
       'SettlingMax': 1.1249611907772497,
       'Overshoot': 12.496119077724966,
       'Undershoot': 0,
       'Peak': 1.1249611907772497,
       'PeakTime': 0.43609566155190244,
       'SteadyStateValue': 1.0}
```



4.7 Lag compensator design

First we calculcate the value of α needed to satisfy the steady state requirement

[53]:
```
s_sym = sp.symbols('s')
Gs = 20/(s_sym*(s_sym + 2))
Gcs = K*(s_sym-z)/(s_sym-p)
Kv_current = float(sp.limit(s_sym*Gs*Gcs, s_sym, 0))
eq_disp('K_v', sp.N(Kv_current,3))
```

$K_v = 7.88$

[54]:
```
alpha = Kv/Kv_current
eq_disp('\\alpha', alpha)
```

$\alpha = 6.34683397273663$

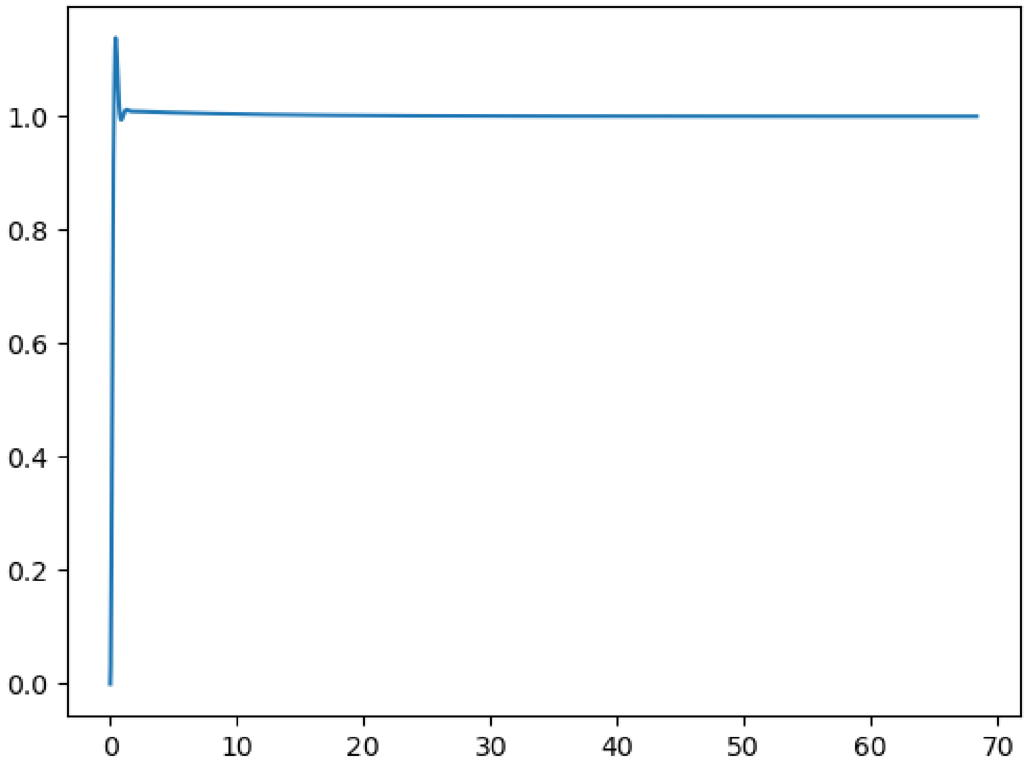
We set the zero position to 0.1 and calculate the pole position from α

[55]:
```
z_lag = 0.1
p_lag = z_lag/alpha
Glag = (s+z_lag)/(s+p_lag)
```

We can now test to see if we meet the requirements

```
[56]: T = ct.feedback(Glead*Glag*G)
      plt.plot(*ct.step_response(T))
      ct.step_info(T)
```

```
[56]: {'RiseTime': 0.20321184180929006,
       'SettlingTime': 0.7257565778903217,
       'SettlingMin': 0.9190036694691347,
       'SettlingMax': 1.1382098574468962,
       'Overshoot': 13.820985744689619,
       'Undershoot': 0,
       'Peak': 1.1382098574468962,
       'PeakTime': 0.435453946734193,
       'SteadyStateValue': 1.0}
```



Velocity constant K_v is given by

```
[57]: s_sym = sp.symbols('s')
      Gs = 20/(s_sym*(s_sym + 2))
      Gcs = K*(s_sym-z)/(s_sym-p)*(s_sym-z_lag)/(s_sym-p_lag)
      Kv_current = float(sp.limit(s_sym*Gs*Gcs, s_sym, 0))
      eq_disp('K_v', sp.N(Kv_current,3))
```

$K_v = 50.0$

Giving a steady state error of

[58]:
```
eq_disp('e_{ss}', sp.N(1/Kv_current,3))
```

$e_{ss} = 0.02$

5 DP10.10

[59]:
```
G = (s + 1.59)/(s*(s + 3.7)*(s**2 + 2.4*s + 0.43))
G
```

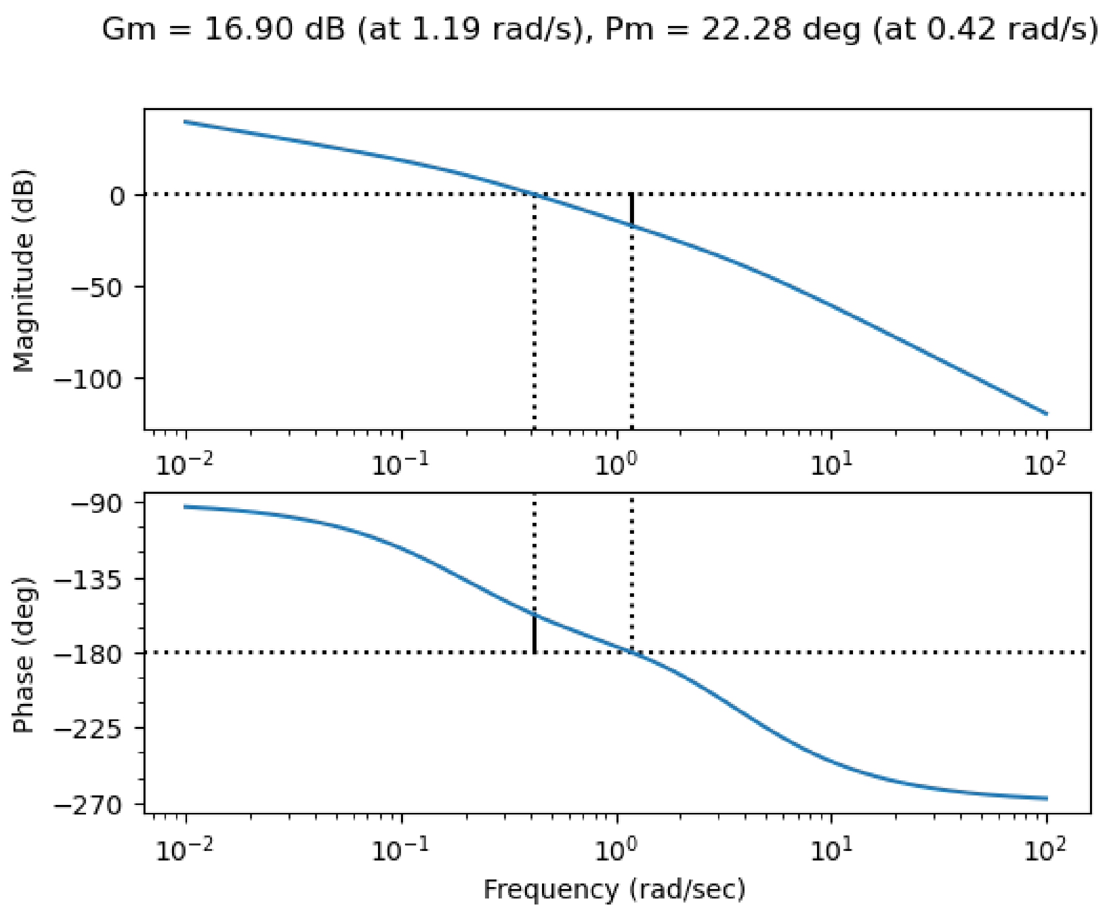
[59]:

$$\frac{s + 1.59}{s^4 + 6.1s^3 + 9.31s^2 + 1.591s}$$

The requirements are: 1. $20log|L| > 20$ dB at $omega \leq 0.01rad/s$ 2. $20log|L| < 20$ dB at $omega \leq 10rad/s$

5.1 The uncompensated frequency response

[60]:
```
%matplotlib inline
mag, phase, omega = ct.bode(G, dB=True, margins=True)
omega20 = 0.01
omegam20 = 10
```

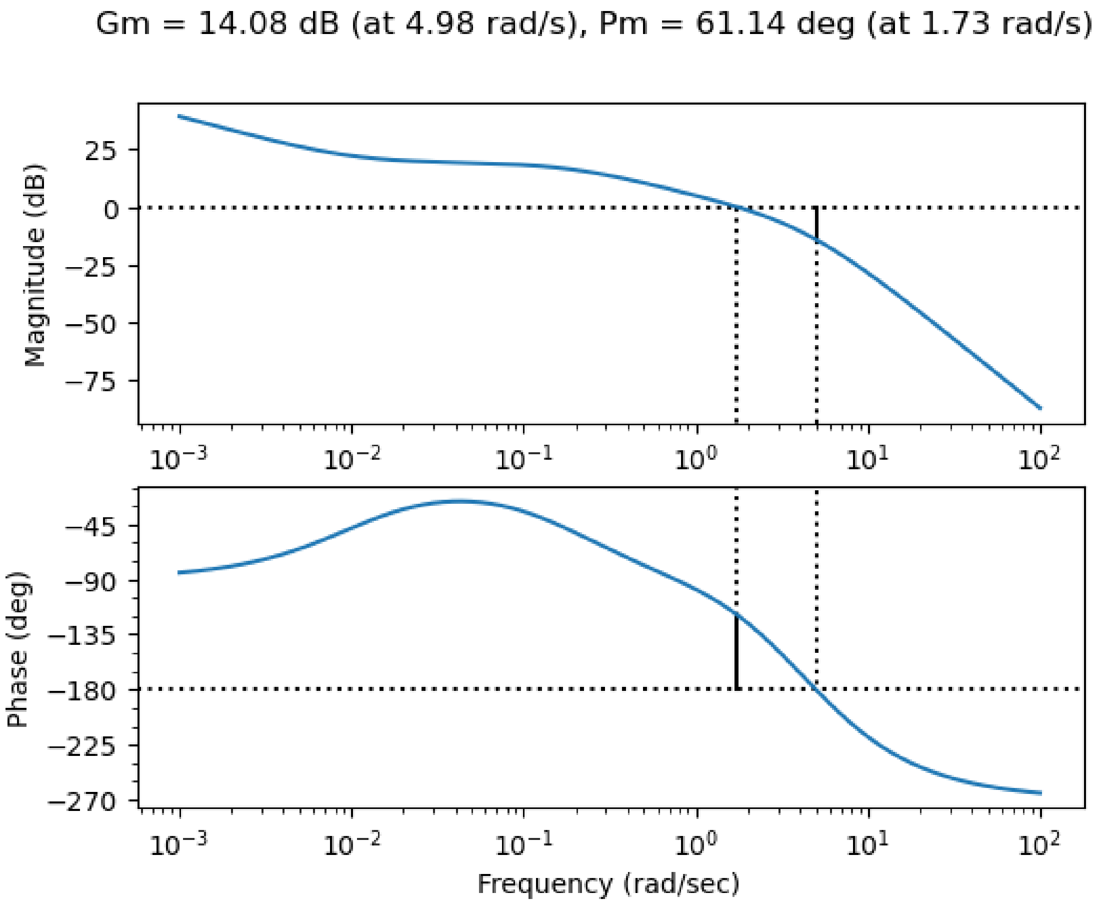


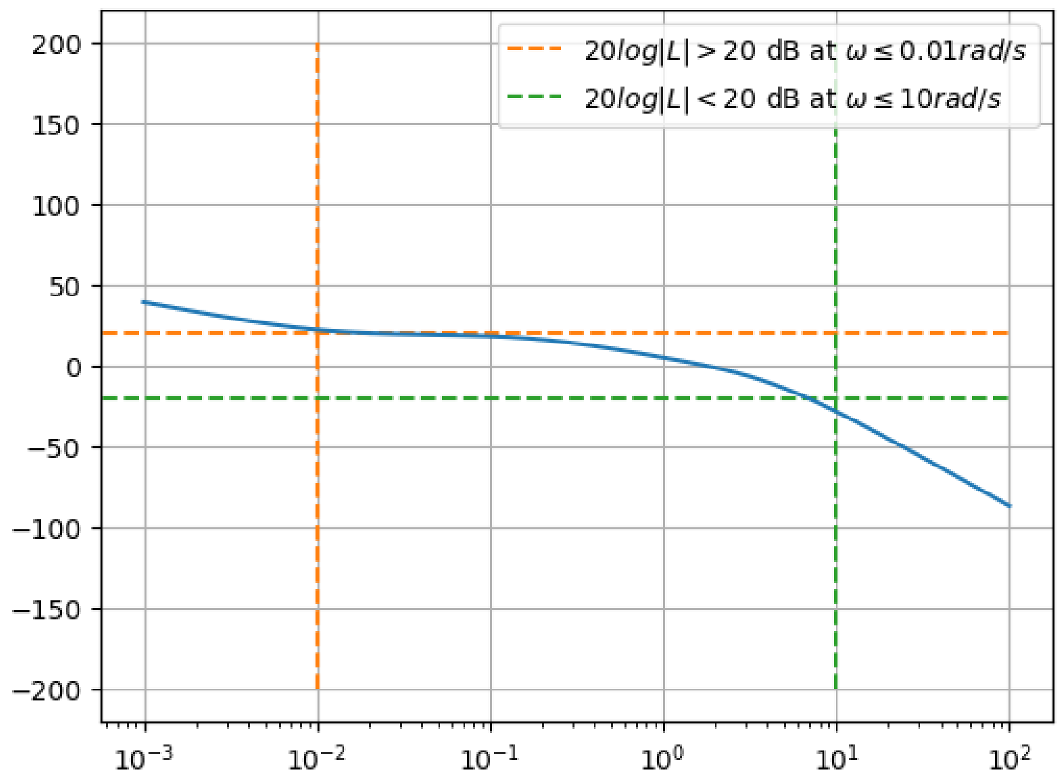
5.2 The compensated frequency response

A zero is placed at low frequency to increase the magnitude slope, a gain increases the magnitude both to satisfy the first requirement. A pole is placed at higher frequency to decrease the slope to meet the second requirement. The values are found by trial and error

[61]:
```
Gc = 45*(s + 0.01)/(s + 5)
L = Gc*G
mag, phase, omega = ct.bode(L, dB=True, margins=True)



fig, (ax) = plt.subplots()
ax.plot(omega,20*np.log10(mag))

# mag=20
plt.hlines(20, 0, 10**(2), linestyle='--', color='tab:orange',␣
 ↪label='$20log|L|>20$ dB at $\\omega \leq 0.01 rad/s$')
plt.vlines(omega20, -200, 200, linestyle='--', color='tab:orange')
```

```python
# mag=-20
plt.hlines(-20, 0, 10**(2), linestyle='--', color='tab:green',␣
 ↪label='$20log|L|<20$ dB at $\\omega \leq 10 rad/s$')
plt.vlines(omegam20, -200, 200, linestyle='--', color='tab:green')

plt.legend()
ax.set_xscale('log')
ax.grid()
```





The plot shows that the design meets the two requirements.

High gain magnitude is wanted at low frequencies to decrease the sensitivity to disturbances and to decrease the sensitivity to plant changes over time. Low gain magnitude is wanted at high frequencies to decrease the sensitivity to noise signals.