Root-Locus Analysis

Lecture 6

Overview

- Introduction to Root-Locus
- Drawing Root Locus using Python Control Module
- We see several examples
- Controller Design using Root Locus Method
    - Lead Compensator
    - Lag Compensator

Root-Locus Plots

- Closed-loop transfer function:

$$\frac{C(s)}{R(s)} = \frac{G(s)}{1 + G(s)H(s)}$$

- The **characteristic equation** is obtained by equating the denominator to zero.

$$1 + G(s)H(s) = 0 \qquad \text{OR} \qquad G(s)H(s) = -1$$

- Since G(s)H(s) is a complex quantity, the following conditions are satisfied:

Angle condition:

$$\angle G(s)H(s) = \pm 180^\circ(2k + 1) \qquad (k = 0, 1, 2, \ldots)$$

Magnitude condition:

$$|G(s)H(s)| = 1$$

- The characteristic equation can be written as follows:

$$1 + \frac{K(s + z_1)(s + z_2) \cdots (s + z_m)}{(s + p_1)(s + p_2) \cdots (s + p_n)} = 0$$

- The root-loci for the system are the loci of closed-loop poles as the gain K is varied from 0 to infinity. At each point on root locus, the root satisfies both the angle and gain criteria.

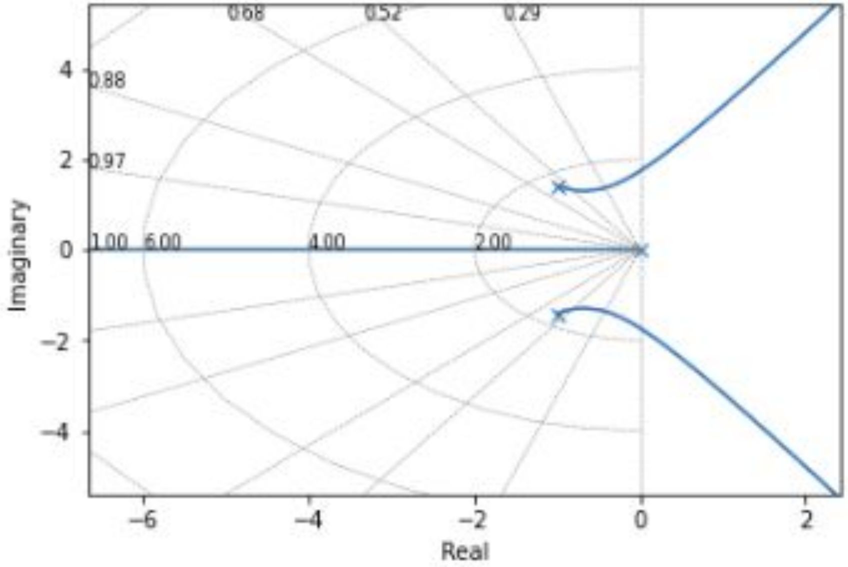
- An approximate root-locus can be drawn by hand without needing any computer.

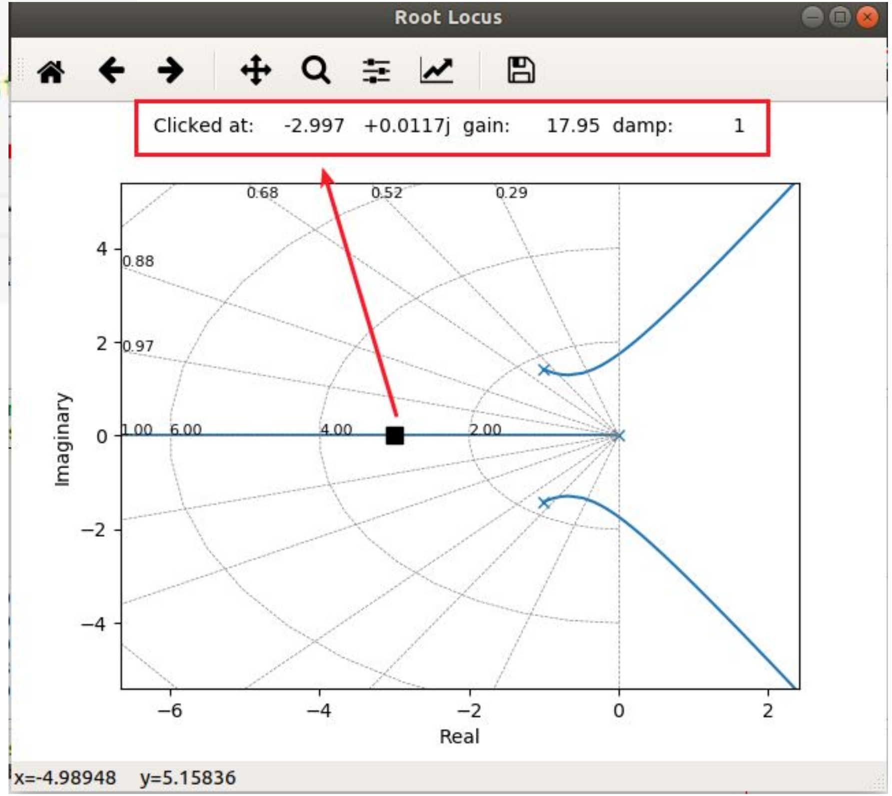
- Closed-loop system behaviour (e.g. stability) can be analyzed without computing the closed-loop poles.

- It can be used as a design tool to select suitable control parameters (such as gain K) so that desirable closed-loop performance specifications are met.

```python
from control import *
import numpy as np
import matplotlib.pyplot as plt
#%matplotlib qt   # uncomment it on your syst
g = tf(1, [1,2,3,0])
k = rlocus(g)
print('Poles:{}'.format(g.pole()))
```

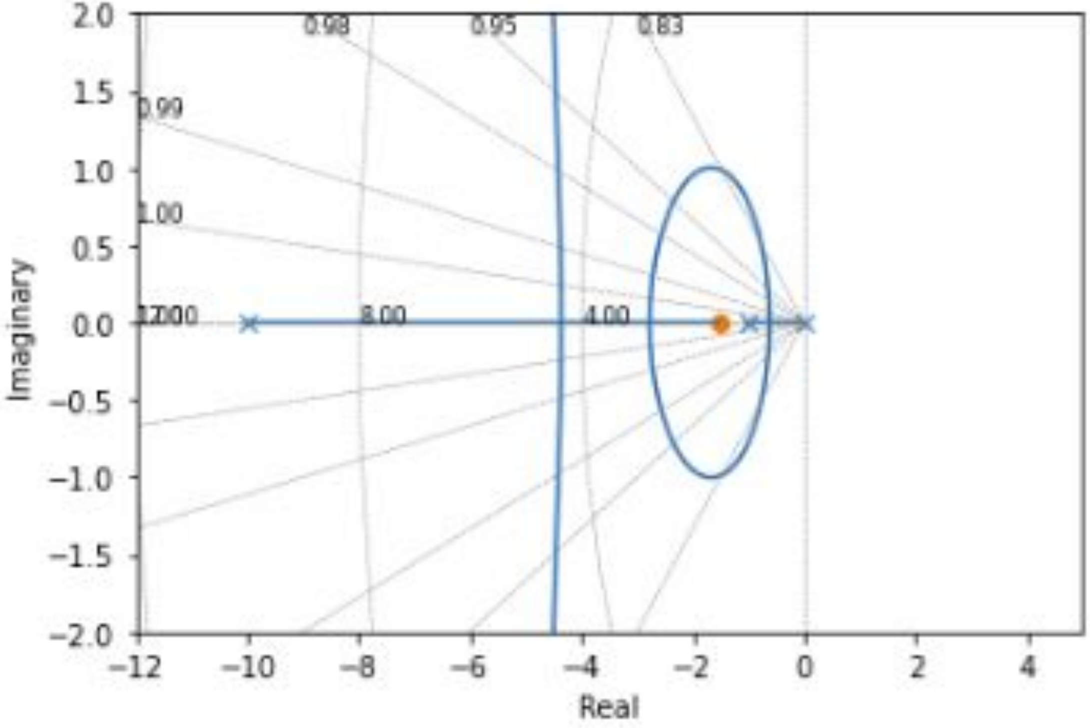
```
Poles:[-1.+1.41421356j -1.-1.41421356j  0.+0.j
```





```python
import numpy as np
from matplotlib import pyplot as plt
import control

#%matplotlib
fig = plt.figure()
G = control.TransferFunction((1, 1.5), (1, 11, 10, 0))

rlist, klist = control.root_locus(G, kvect=np.linspace(-100,100, num=100),
                                  xlim=(-12,5), ylim=(-2,2))
#rlist, klist = control.rlocus(G, kvect=np.linspace(-100,100, num=100),
#                                  xlim=(-12,5), ylim=(-2,2))

print('shape of rlist: ', np.shape(rlist))
print('shape of klist: ', np.shape(klist))

plt.show()

```

```
shape of rlist:  (100, 3)
shape of klist:  (100,)
<Figure size 432x288 with 0 Axes>
```



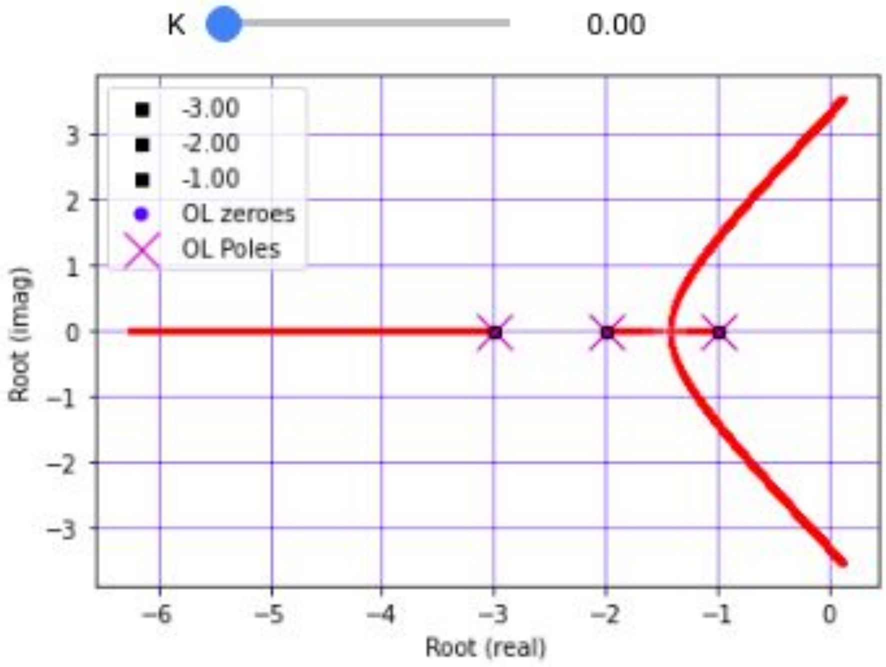
It is possible to zoom in by specifying xlim and ylim variables.

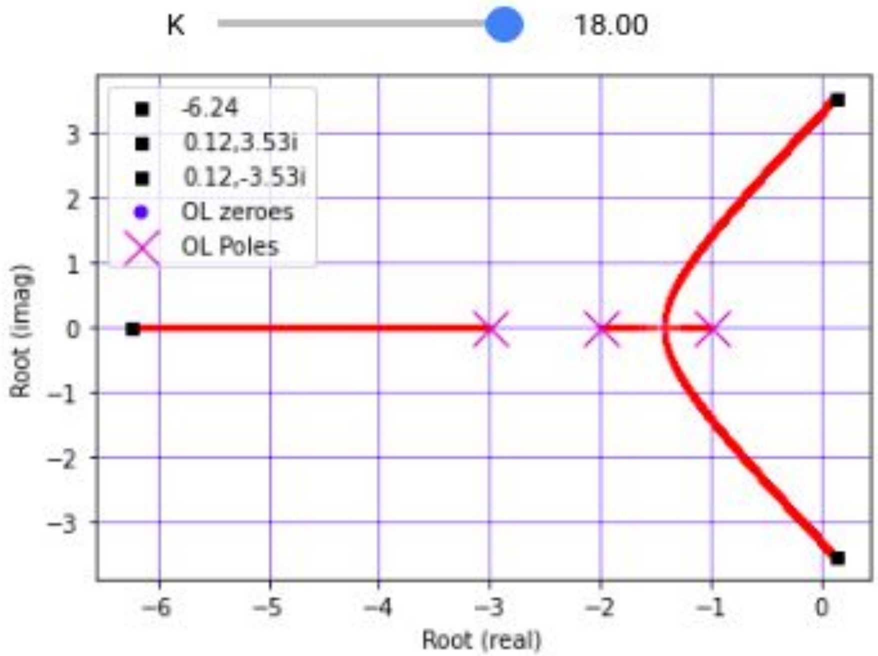
```python
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact
from ipywidgets import *
%matplotlib inline

# open loop
num = [4.0]
den = [1.0,6.0,11.0,6.0]

# Characteristic Polynomial
# 1 + G(s)H(s) = s^3 + 6s^2 + 11s+ 6+4K = 0
def dcl(K):
    return [1.0,6.0,11.0,4.0*K+6.0]

# root locus plot
n = 10000 # number of points to plot
nr = len(den)-1 # number of roots
rs = np.zeros((n,2*nr))   # store results

# Range of Gain
Kc1 = -2.0
Kc2 = 18.0
Kc = np.linspace(Kc1,Kc2,n)  # Kc values
for i in range(n):         # cycle through n time
    roots = np.roots(dcl(Kc[i]))
    for j in range(nr):   # store roots
        rs[i,j] = roots[j].real # store real
        rs[i,j+nr] = roots[j].imag # store imagi
```

```python
def update(K=0):
    indx = (np.abs(Kc-K)).argmin()
    for i in range(nr):
        plt.plot(rs[:,i],rs[:,i+nr],'r.',markersize=2)
        if math.isclose(rs[indx,i+nr], 0.0):
            lbl = '{:.2f}'.format(rs[indx,i])
        else:
            lbl = '{:.2f},{:.2f}i'.format(rs[indx,i], rs[indx,i+nr])
        plt.plot(rs[indx, i], rs[indx,i+nr], 'ks', markersize=5,label=lbl)

    plt.legend(loc='best')
    plt.xlabel('Root (real)')
    plt.ylabel('Root (imag)')
    plt.grid(b=True, which='major', color='b', linestyle='-',alpha=0.5)
    plt.grid(b=True, which='minor', color='r', linestyle='--',alpha=0.5)

interact(update, K=widgets.FloatSlider(value=Kc1,
                                       min = Kc1,
                                       max = Kc2,
                                       step = 0.1))
```

$$G(s) = \frac{4K}{(s+3)(s+2)(s+1)}$$

Root Loci start at open-loop poles when K =0 and then go to infinity or open-loop zeros as K tends to infinity.





Interactive Sliding bar to vary K

Drawing the root locus for the following open loop system:

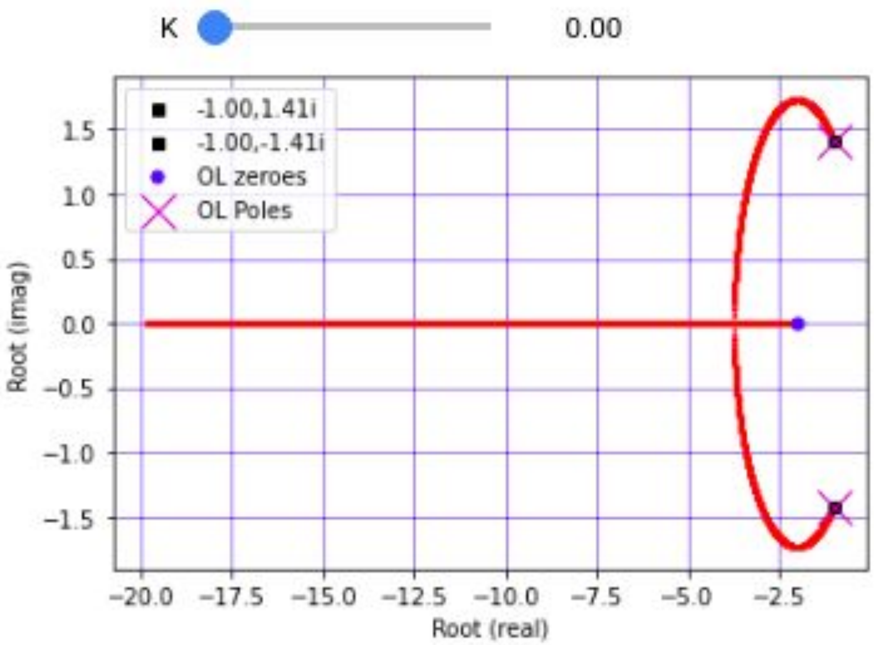
$$G(s) = \frac{K(s+2)}{s^2+2s+3}$$

```python
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact
from ipywidgets import *
%matplotlib inline

def update(kvect, rs, nr, p_real, p_imag, z_real, z_imag, K=0):
    indx = (np.abs(kvect-K)).argmin()
    for i in range(nr):
        plt.plot(rs[:,i], rs[:,i+nr],'r.',markersize=2)
        if math.isclose(rs[indx,i+nr], 0.0):
            lbl = '{:.2f}'.format(rs[indx,i])
        else:
            lbl = '{:.2f},{:.2f}i'.format(rs[indx,i], rs[indx,i+nr])
        plt.plot(rs[indx, i], rs[indx,i+nr], 'ks', markersize=5,label=lbl)

    # Plot open-loop poles and zeros
    plt.plot(z_real, z_imag, 'bo', markersize=5, label='OL zeroes')
    plt.plot(p_real, p_imag, 'mx', markersize=15, label='OL Poles')
    plt.legend(loc='best')
    plt.xlabel('Root (real)')
    plt.ylabel('Root (imag)')
    plt.grid(b=True, which='major', color='b', linestyle='-',alpha=0.5)
    plt.grid(b=True, which='minor', color='r', linestyle='--',alpha=0.5)
```

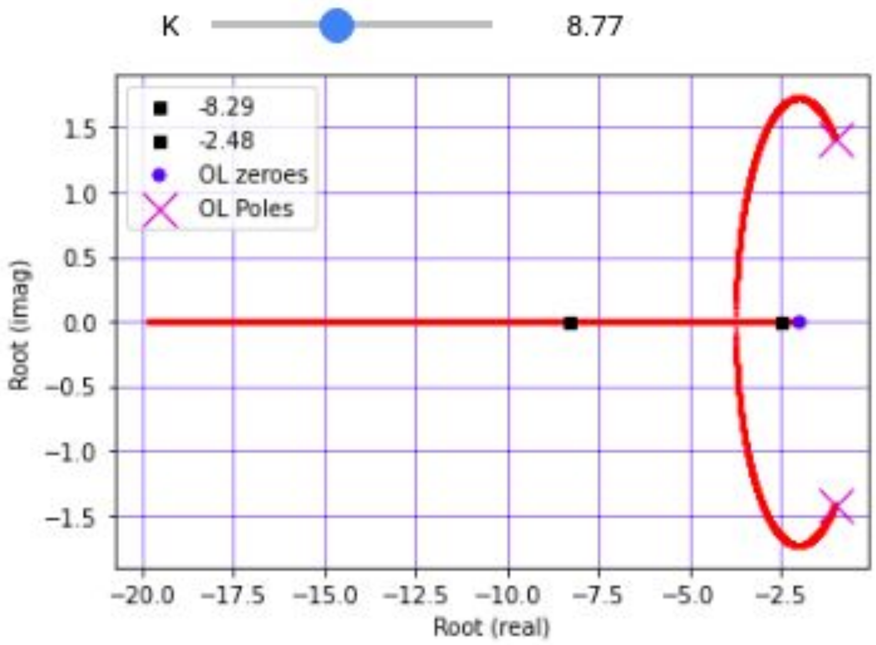
```python
def RootLocus(num, den, kvect):
    '''
    Plots Root Locus of an open-loop transfer function g = tf(num,den)
    for a given gain vector
    '''

    # open-loop poles and zeroes
    sys_zeroes = np.roots(num)
    sys_poles = np.roots(den)

    # Real & Imag Parts of Poles and zeroes
    z_real = [sys_zeroes[i].real for i in range(len(sys_zeroes))]
    z_imag = [sys_zeroes[i].imag for i in range(len(sys_zeroes))]
    p_real = [sys_poles[i].real for i in range(len(sys_poles))]
    p_imag = [sys_poles[i].imag for i in range(len(sys_poles))]

    Kmin = np.min(kvect)
    Kmax = np.max(kvect)

    n = len(kvect) # no. of data points
    nr = len(den) - 1 # no. of roots
    rs = np.zeros((n, 2*nr))

    for i in range(n):
        # Characteristic Polynomial
        char_poly = np.polyadd(den, kvect[i]*np.asarray(num))
        # Closed loop poles
        roots = np.roots(char_poly)
        for j in range(nr):
            rs[i,j] = roots[j].real # real part
            rs[i,j+nr] = roots[j].imag  # imaginary part

    # interactive plot
    interact(update, kvect=fixed(kvect),rs=fixed(rs), nr=fixed(nr),\
             p_real = fixed(p_real), p_imag=fixed(p_imag),
             z_real = fixed(z_real), z_imag = fixed(z_imag),
             K=widgets.FloatSlider(value=Kmin, min = Kmin,
                                   max = Kmax, step = 0.01))
```

```python
num = [1, 2]
den = [1, 2, 3]
K = np.linspace(0,20, 10000)

RootLocus(num, den, K)
```

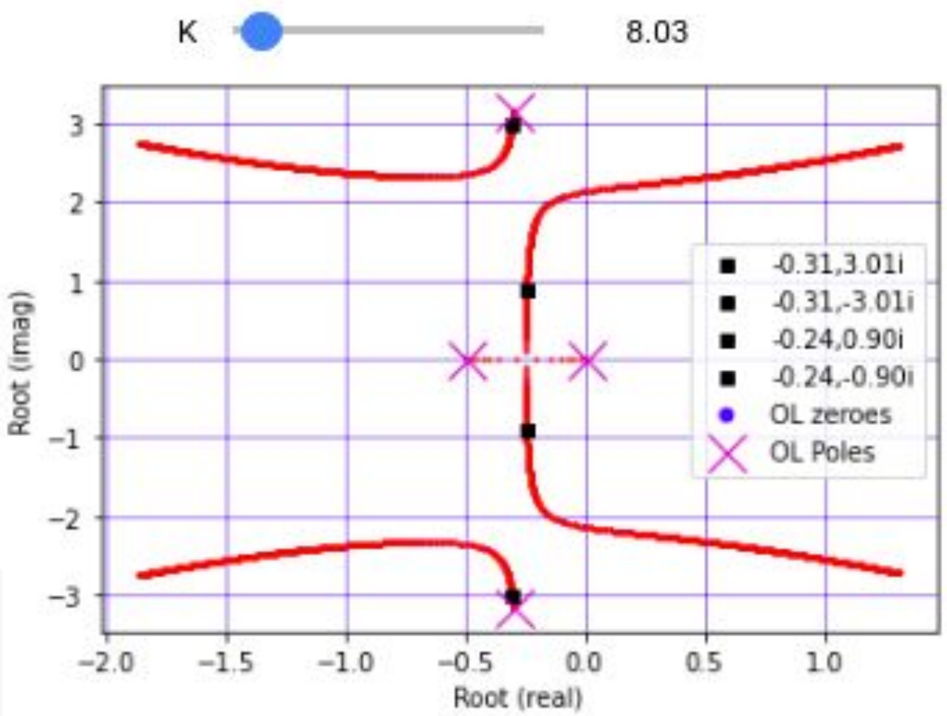
We create our own RootLocus() function to draw root locus with an interactive sliding bar for the gain.



Closed-loop poles starts at open-loop poles with K=0 and terminate at open-loop zeros or infinity when k ---> \infty.

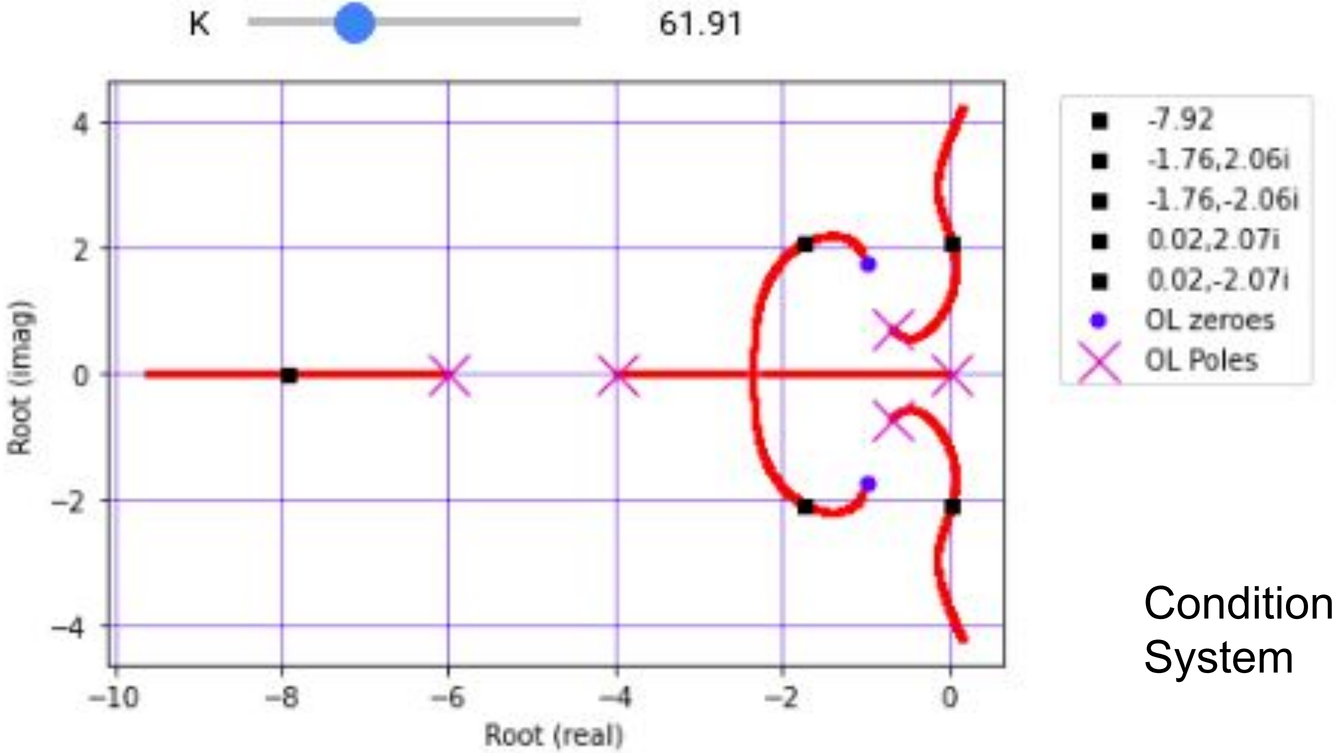


$$G(s) = \frac{K}{s(s+0.5)(s^2+0.6s+10)}$$



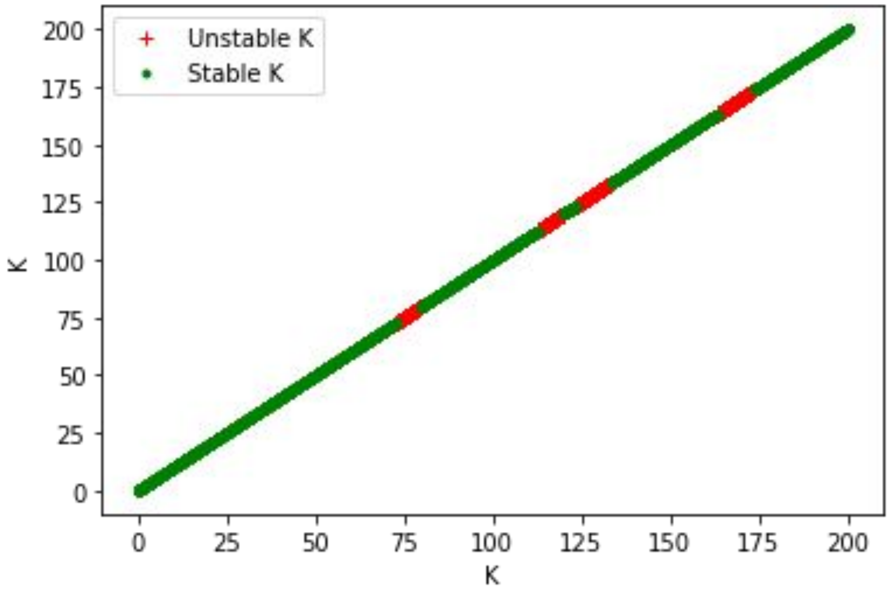

```
num = [1.]
den = [1., 1.1, 10.3, 5., 0]
K = np.linspace(0, 100, 1000)
RootLocus(num, den, K)
```

$$G(s) = \frac{K(s^2+2s+4)}{s(s+4)(s+6)(s^2+1.4s+1)}$$



Conditionally Stable System

```python
num = [1., 2., 4.]
den = [1, 11.4, 39., 43.6, 24., 0]
K = np.linspace(0,200,10000)

# Draw Root Locus
k, r = RootLocus(num, den, K)

# Plot stable and Unstable Gains
print('shape of r:', np.shape(r))
nr = len(den) - 1 # no. of roots
r_roots = [r[:,i] for i in range(nr)]
r_roots = np.reshape(r_roots, (len(k), nr))
print('size of r_roots: ', np.shape(r_roots))

posk_idx = np.where(r_roots >= 0)
posk_idx = np.unique(posk_idx[0])
print('size of posk_idx:', np.shape(posk_idx))

posk = k[posk_idx]
negk = np.delete(k, posk_idx, 0)
print('size of posk: ', np.shape(posk))
print('size of negk: ', np.shape(negk))

plt.plot(posk, posk, 'r+', label='Unstable K')
plt.plot(negk, negk, 'g.', label='Stable K' )
plt.xlabel('K')
plt.ylabel('K')
plt.legend(loc='best')
```



System is stable for a limited range of K:

Controller Design using Root Locus

- Root-locus method is a graphical method for determining the location of closed-loop poles from the knowledge of open-loop poles and zeros as some parameters (usually the gain) is varied from 0 to infinity.

- Desirable closed-loop behaviour can be obtained by adding additional poles and zeros to the open-loop system transfer function and selecting suitable gain from the RL plot.

- Root-locus method is a powerful method for designing controller when the performance specifications are provided in terms of time-domain quantities such as damping ratio, natural frequency, peak overshoot, settling time etc.

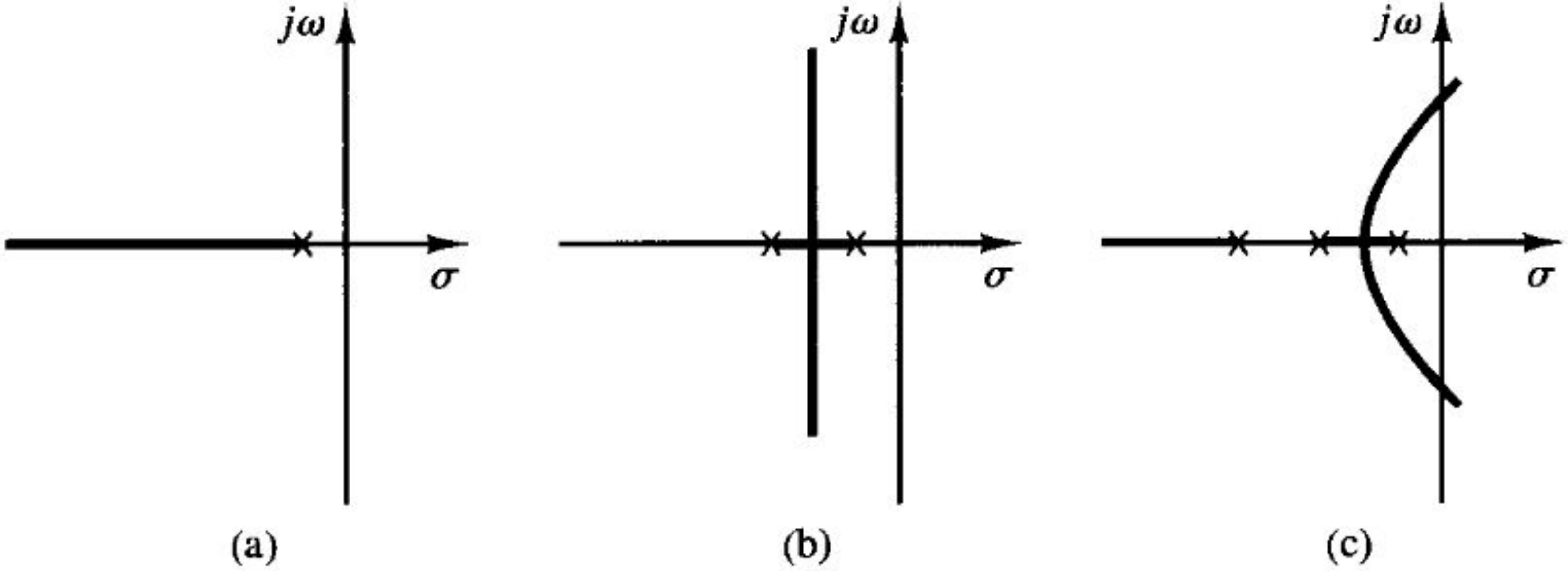
Effect of addition of poles & Zeros

- The addition of a pole to the open-loop transfer function has the effect of pulling the root-locus to the right, tending to lower system's relative stability and to slow down the settling of the response.

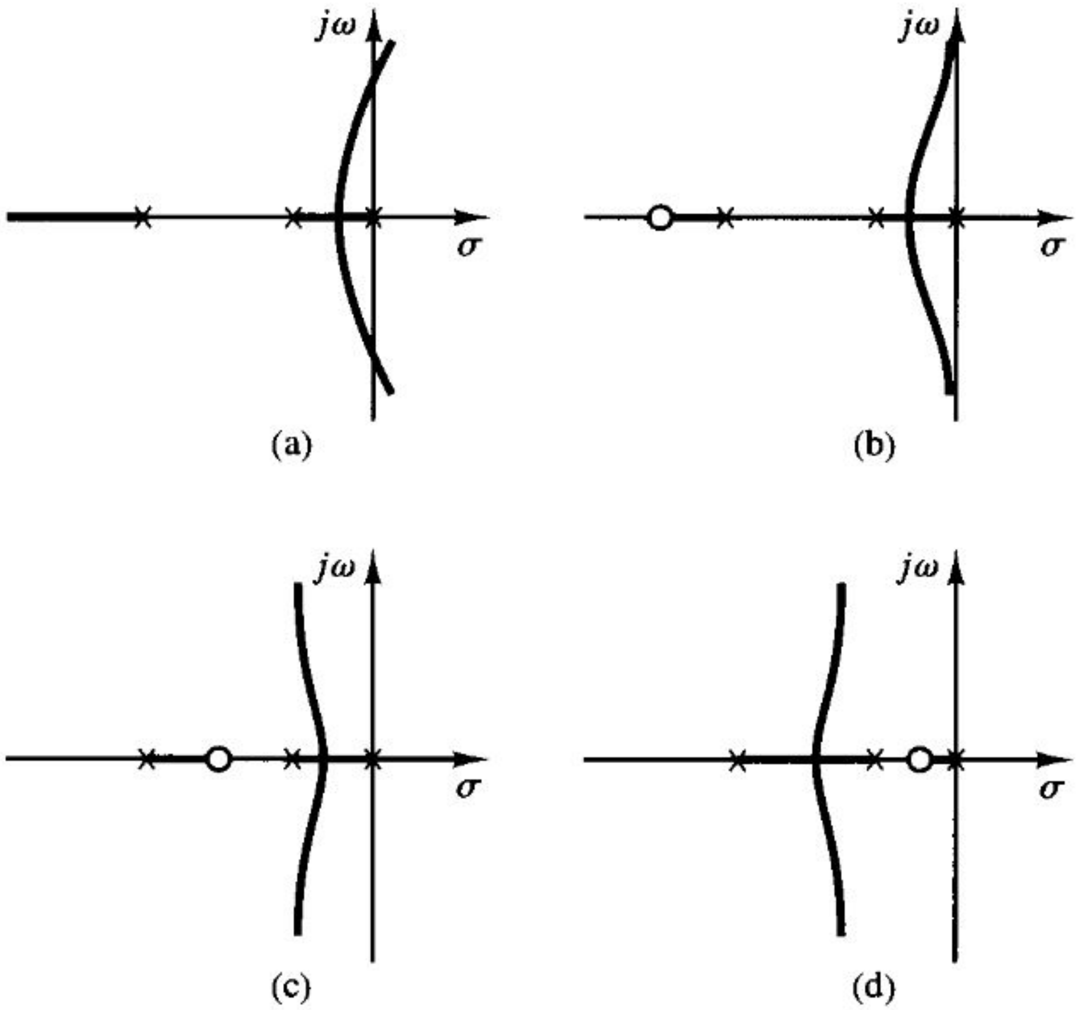
- The addition of a zero to the open-loop transfer function has the effect of pulling the root-locus to the left, tending to make the system more stable and to speed up the settling of the response.

- Adding zero is similar to adding PD control action to the system.

- Adding a pole is similar to adding an integral control action to the system.



Effect of adding poles on the R-L plot



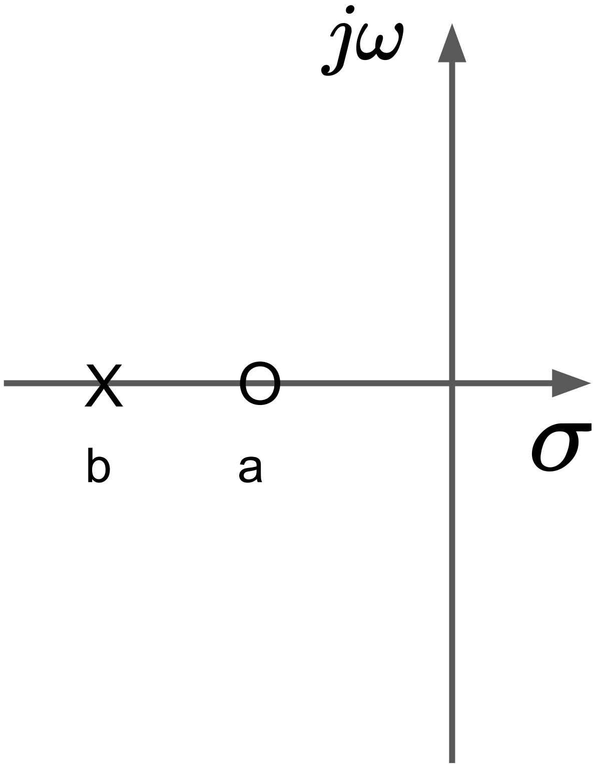
Effect of adding zeros to the open-loop system on the Root Locus Plot.

Counting from right to left, the part of real-axis that lies to the left of an odd number of poles and zeros is a part of the root locus.

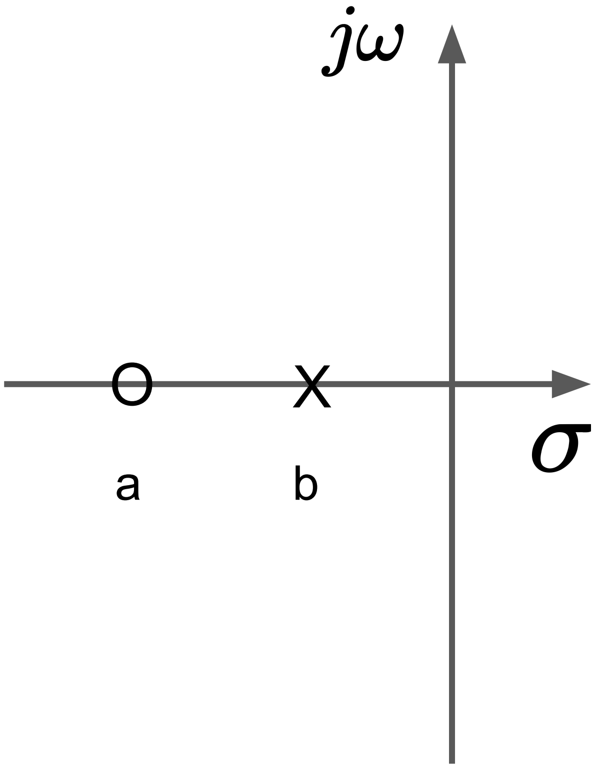
Lead and Lag Compensator

- A **lead compensator** adds positive phase angle to the Sinusoidal response of the closed system.
  - This has the effect of improving the transient behaviour of the system - faster damping of oscillations, smaller rise time and settling time, lower peak overshoot etc.

- A **lag compensator** adds negative phase angle to the sinusoidal response of the system.
  - This has the effect of slowing down the system response - increased settling time, larger oscillations (higher peak overshoot).



Lead Compensator
(a > b)

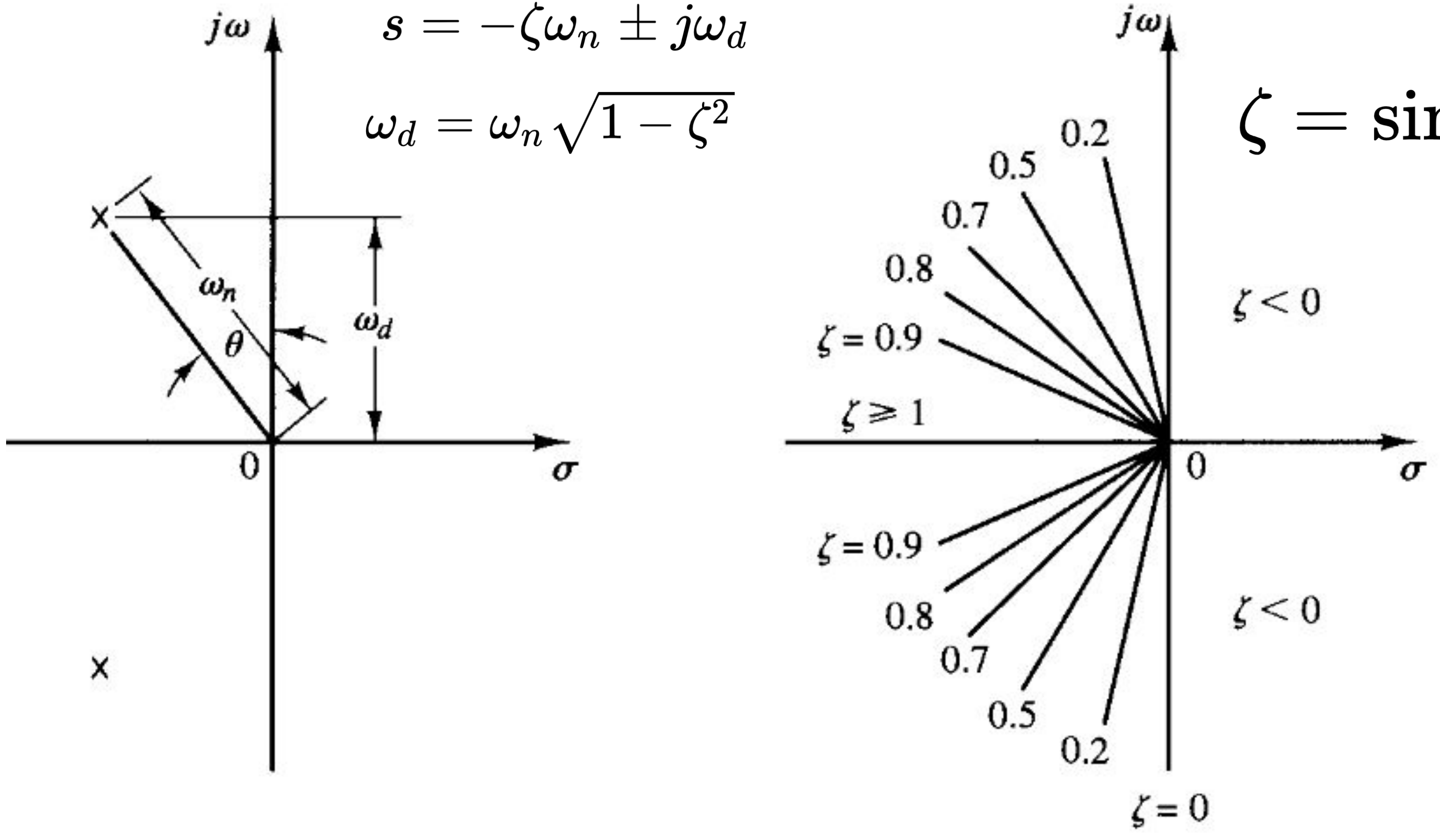


Lag Compensator
(a < b)

$$s = -\zeta\omega_n \pm j\omega_d$$

$$\omega_d = \omega_n\sqrt{1-\zeta^2}$$

$$\zeta = \sin(\theta)$$



(a) (b)

Example: Designing a Lead Compensator for a system

$$G(s) = \frac{4}{s(s+2)}$$

$$\frac{C(s)}{R(s)} = \frac{4}{s^2 + 2s + 4}$$

$$s = -1 \pm j\sqrt{3}$$

Goal is to design a compensator that will reduce the settling time and rise time of the system performance.

$$\theta = \tan^{-1}(\frac{1}{\sqrt{3}})$$

$$\zeta = \sin(\theta) = 0.5$$

$$\sigma = \zeta\omega_n = 1$$

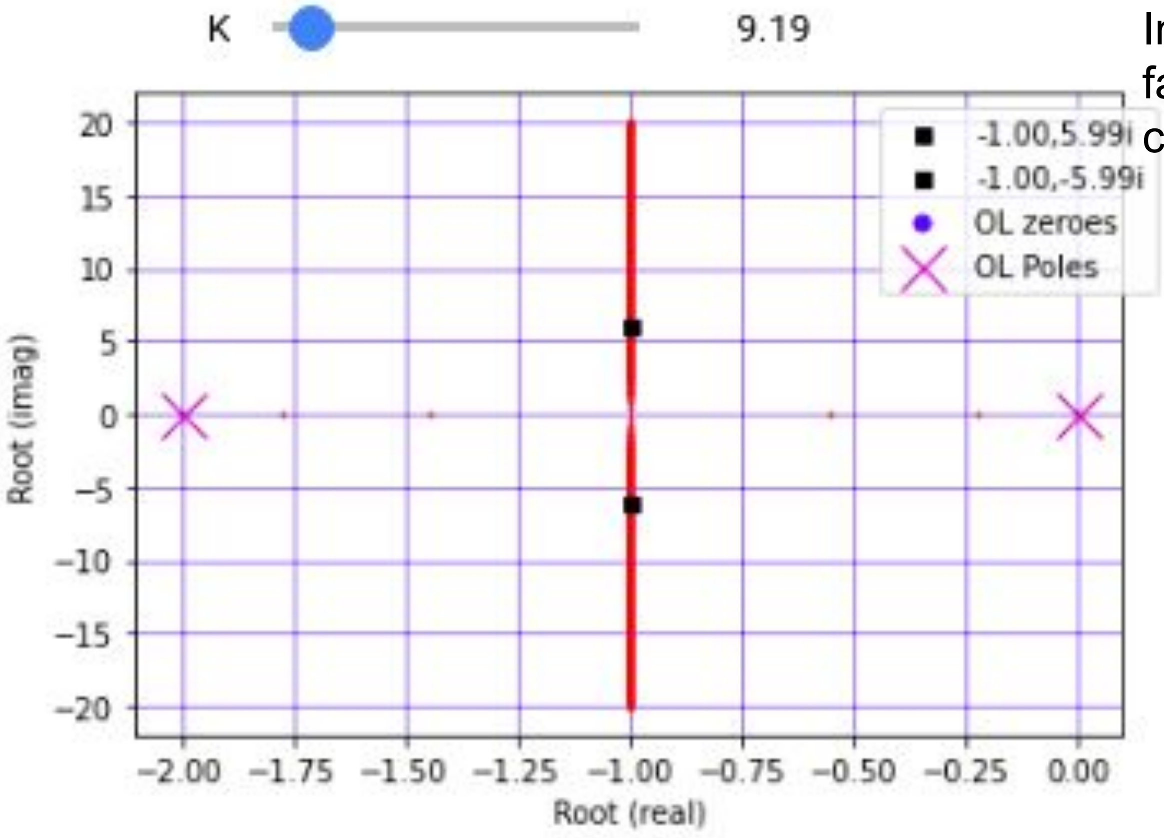
$$\omega_d = \omega_n\sqrt{1-\zeta^2} = \sqrt{3}$$

$$\beta = \tan^{-1}(\frac{\omega_d}{\sigma})$$

$$t_s = \frac{\sigma}{\omega_d} = 4 \text{ sec}$$

$$M_p = e^{-(\frac{\sigma}{\omega_d})\pi} = 16\%$$

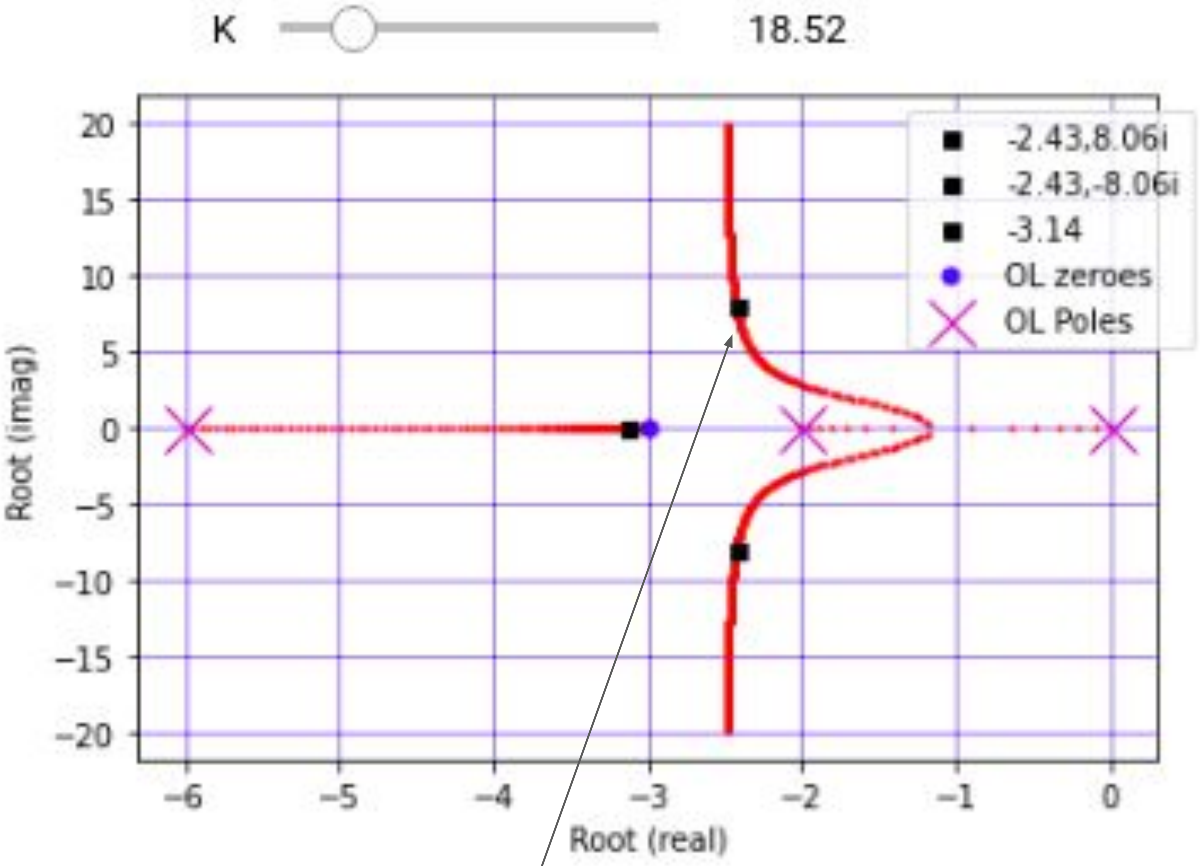
$$t_r = \frac{(\pi-\beta)}{\omega_d} = 1.21 \text{ sec}$$



In order to make the system faster, we will add a lead compensator to the system.

- That means we should pull the RL to the left.
- So add a zero and a pole to the left of open-loop pole s = -2.0
- It is possible to add a zero to cancel one of the existing poles.

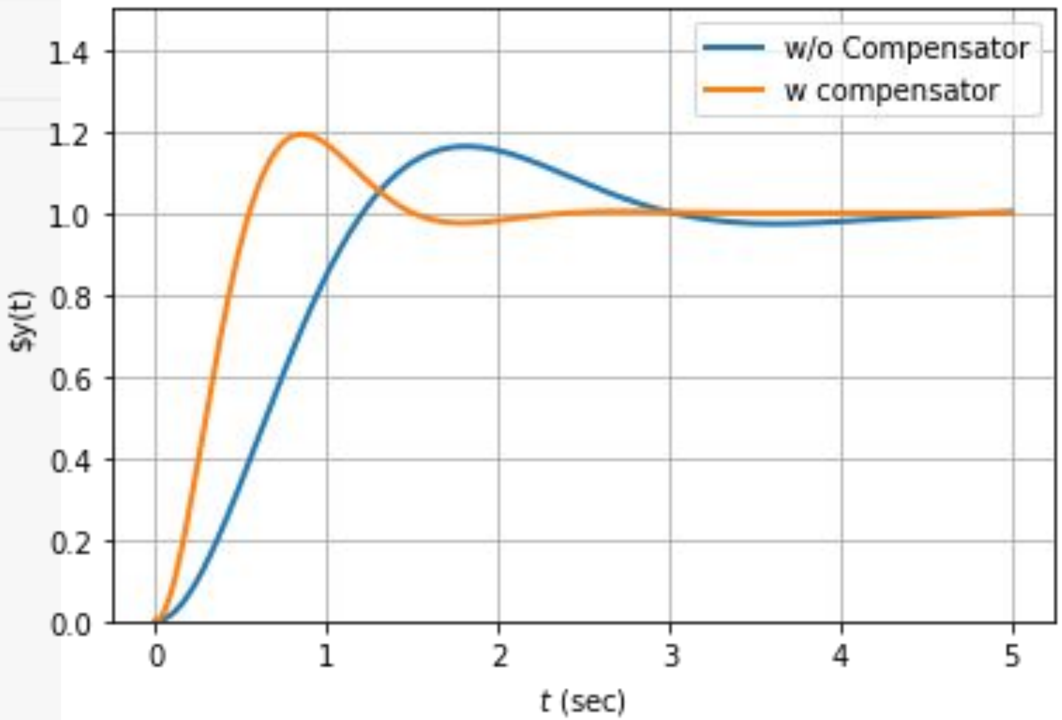
$$G_c(s) = \frac{K(s+3)}{(s+6)}$$



These are to the left of dominant poles for the uncompensated system.

- Now redraw the root locus with G*Gc as the open-loop transfer function.
- Now select a gain value (4K here) for a desirable transient performance.

```
from control import *
num1 = [4]
den1 = [1, 2, 0]

# open-loop plant
g = tf(num1,den1)

K = 5.0
num2 = [1,3]
den2 = [1,6]

# Controller
c = tf(K*np.asarray(num2), den2)

# Unit feedback system
gc1 = feedback(g,1,-1)
# Closed-loop system with Compensator
gc2 = feedback(series(g,c),1,-1)

t = np.linspace(0,5,1000)
t, y1 = step_response(gc1, t)
t, y2 = step_response(gc2, t)

plt.plot(t, y1, lw = 2, label='w/o Compensator')
plt.plot(t, y2, lw = 2, label='w compensator')
plt.xlabel('$t$ (sec)')
plt.ylabel('$y(t)')
plt.ylim((0,1.5))
plt.grid()
plt.legend(loc='best')
```



```python
import math

def time_spec(sigma, wd):
    theta = math.atan(sigma/wd)
    zeta = math.sin(theta)
    ts = 4/sigma
    mp = math.e**(-math.pi*sigma/wd)
    beta = math.atan(wd/sigma)
    tr = (math.pi - beta)/wd

    print('zeta = {:.2f}'.format(zeta))
    print('Mp = {:.2f}'.format(mp))
    print('ts = {:.2f}'.format(ts))
    print('tr = {:.2f}'.format(tr))

# open-loop dominant poles
print('open-loop transient response paramters:')
sigma1 = 1
wd1 = math.sqrt(3)
time_spec(sigma1, wd1)

print('\n-----------\n')

# closed-loop dominant poles
print('closed-loop transient response specs:')
sigma2 = 2.4
wd2 = 8.06
time_spec(sigma2, wd2)
```

```
open-loop transient response paramters:
zeta = 0.50
Mp = 0.16
ts = 4.00
tr = 1.21

-----------

closed-loop transient response specs:
zeta = 0.29
Mp = 0.39
ts = 1.67
tr = 0.23
```

Design a Lag Compensator

$$G(s) = \frac{1}{s(s+1)(s+2)}$$

$$\frac{C(s)}{R(s)} = \frac{1}{s(s+1)(s+2)+1}$$

For the uncompensated system:

Dominant Poles:

$$s = -0.33 \pm 0.56j$$

Velocity error constant:

$$K_v = \lim_{s \to 0} sG(s) = 0.5$$

Design goal is to increase the velocity error constant by 10 times (reduce steady state error by 10 times) keeping other transient specs unchanged.

Consider a lag compensator given as follows:

$$G_c(s) = \frac{K(s+a)}{(s+b)} = \frac{K(s+0.05)}{(s+0.005)}$$

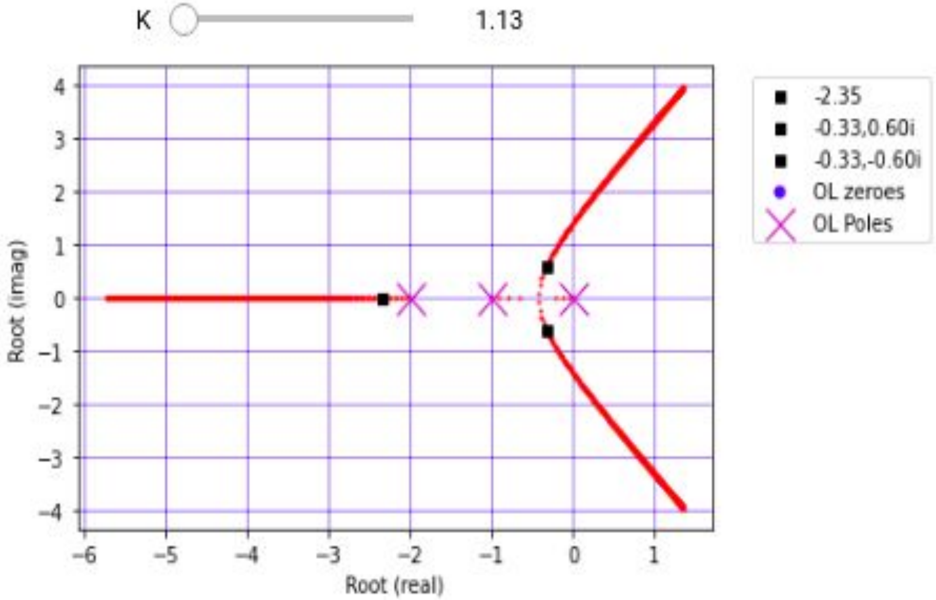
$$K_v' = \lim_{s \to 0} sG_c(s)G(s) = 5K = 5 \Rightarrow K = 1$$

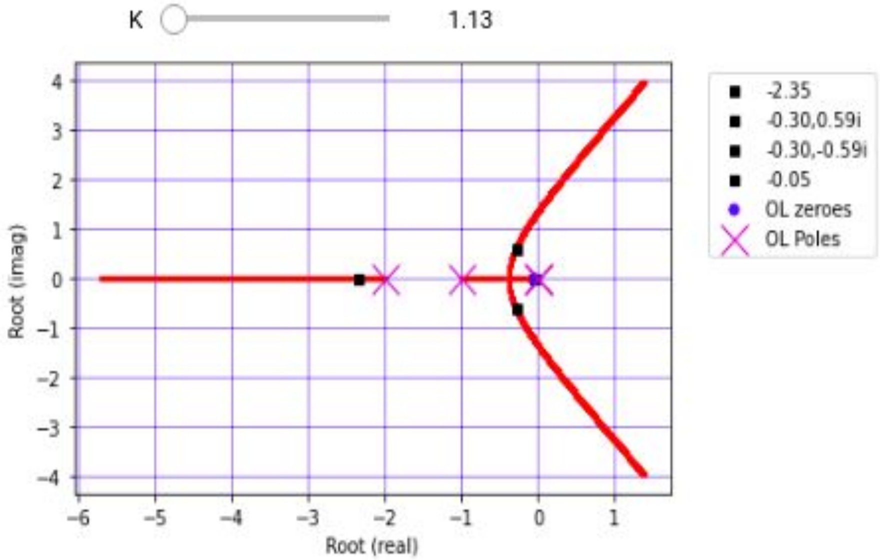
- Controller poles are zeros are very far away from dominant poles and hence does not affect the transient response.
- Gain of the compensator is obtained from the steady-state performance requirement
- This lag compensators contributes a small lag (phase angle)

```python
num1 = [1]
den1 = [1, 3, 2, 0]
K = np.linspace(0,100,1000)
k,r = RootLocus(num1, den1, K)

# closed-loop poles of uncompensated system
den_c1 = [1, 3, 2,1]
print('Uncompensated CL poles: ',np.roots(den_c1))
```

```python
num_c = [1, 0.05]
den_c = [1, 0.005]

# Gc = G*C
num2 = np.polymul(num_c, num1)
den2 = np.polymul(den_c, den1)

K = np.linspace(0,100, 50000)
k,r = RootLocus(num2, den2, K)
```

With Compensator

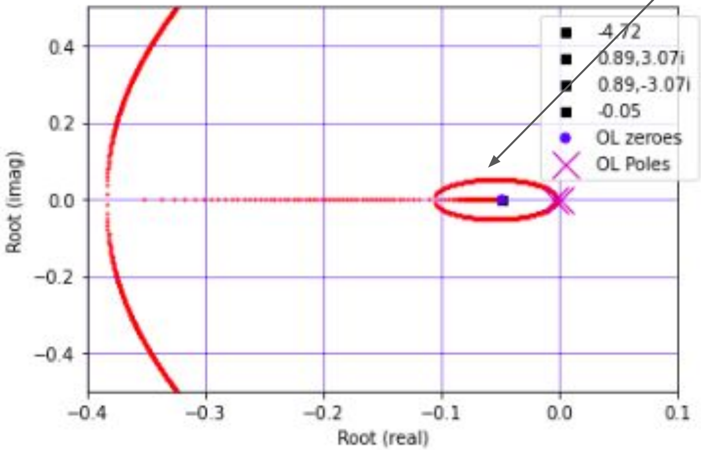




```
num_c = [1, 0.05]
den_c = [1, 0.005]

# Gc = G*C
num2 = np.polymul(num_c, num1)
den2 = np.polymul(den_c, den1)
print(num2)
print(den2)
xr = ((-0.4, 0.1))
yr = ((-0.5,0.5))
K = np.linspace(0,100, 50000)
k,r = RootLocus(num2, den2, K, xr, yr)
```

```
[1.   0.05]
[1.    3.005 2.015 0.01  0.    ]
```

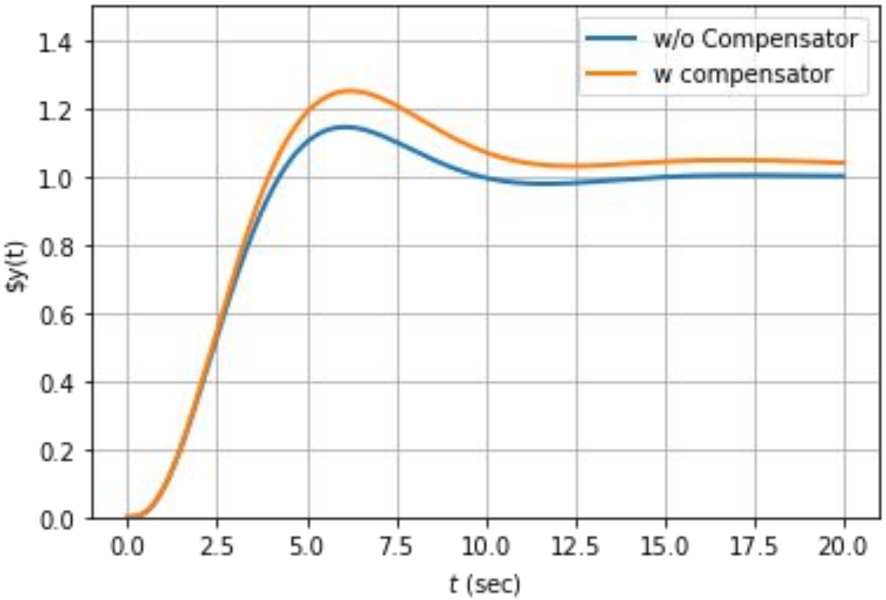
K 48.42



Zoomed in Part near the origin

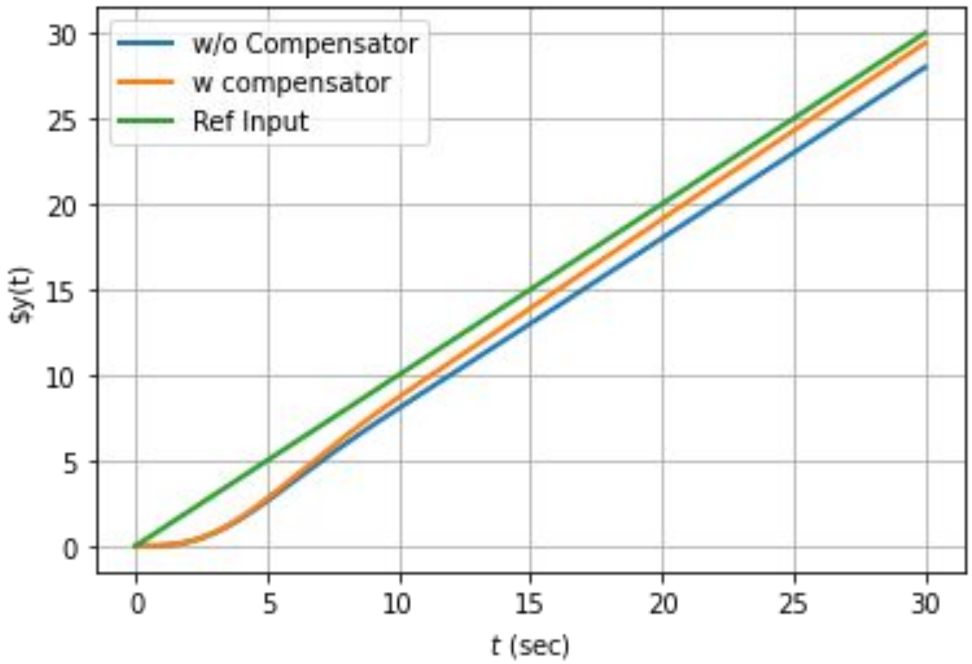
Pole-zero pair added near the origin does not affect the dominant poles for the compensated system and hence, the transient behaviour remains same as the uncompensated system.

```python
# Step Response
from control import *
# open-loop plant
num1 = [1]
den1 = [1, 3, 2, 0]
g = tf(num1,den1)

# Controller
K = 1.0
num2 = [1,0.05]
den2 = [1,0.005]
c = tf(K*np.asarray(num2), den2)

# Unit feedback system
gc1 = feedback(g,1,-1)
# Closed-loop system with Compensator
gc2 = feedback(series(g,c),1,-1)

t = np.linspace(0,20,1000)
t, y1 = step_response(gc1, t)
t, y2 = step_response(gc2, t)

plt.plot(t, y1, lw = 2, label='w/o Compensator')
plt.plot(t, y2, lw = 2, label='w compensator')
plt.xlabel('$t$ (sec)')
plt.ylabel('$y(t)')
plt.ylim((0,1.5))
plt.grid()
plt.legend(loc='best')
```



Transient response of closed-loop system remains unchanged.

```python
# Ramp response
from control import *

# open-loop plant
num1 = [1]
den1 = [1, 3, 2, 0]
g = tf(num1,den1)

# Controller
K = 1.0
num2 = [1,0.05]
den2 = [1,0.005]
c = tf(K*np.asarray(num2), den2)

# Unit feedback system
gc1 = feedback(g,1,-1)
# Closed-loop system with Compensator
gc2 = feedback(series(g,c),1,-1)

t = np.linspace(0,30,1000)
u = t
t, y1, x1 = forced_response(gc1, t, u)
t, y2, x2 = forced_response(gc2, t, u)

plt.plot(t, y1, lw = 2, label='w/o Compensator')
plt.plot(t, y2, lw = 2, label='w compensator')
plt.plot(t,u, lw = 2, label='Ref Input')
plt.xlabel('$t$ (sec)')
plt.ylabel('$y(t)')
plt.grid()
plt.legend(loc='best')
```



Response to Ramp Input

Steady-State performances improves significantly with the lag compensator as desired.

Summary

- Root-Locus is a powerful control design tool for LTI systems.
- Root-Locus can be easily drawn without using computers and hence was a dominant tool in the early 50’s.
- Controller design primarily involves adding zeros and poles to the open-loop transfer function and see its effect on the RL plot.
- We explored Python Control Module to draw and analyze root locus plots.
- We demonstrated two examples of controller design using RL method.