L7-3: Frequency Domain Analysis

CDS 110/ChE 105, Winter 2024
Richard M. Murray

The purpose of this lecture is to introduce tools that can be used for frequency domain modeling and analysis of linear systems.

In [1]:
```
# Import standard packages needed for this exercise
import numpy as np
import matplotlib.pyplot as plt
import math

from math import pi, sin, cos

try:
  import control as ct
  print("python-control", ct.__version__)
except ImportError:
  !pip install control
  import control as ct
```

```
python-control 0.10.0
```

Stable system: servomechanism

We start with a simple example a stable system for which we wish to design a simple controller and analyze its performance, demonstrating along the way that basic frequency domain analysis functions in the Python control toolbox (python-control).

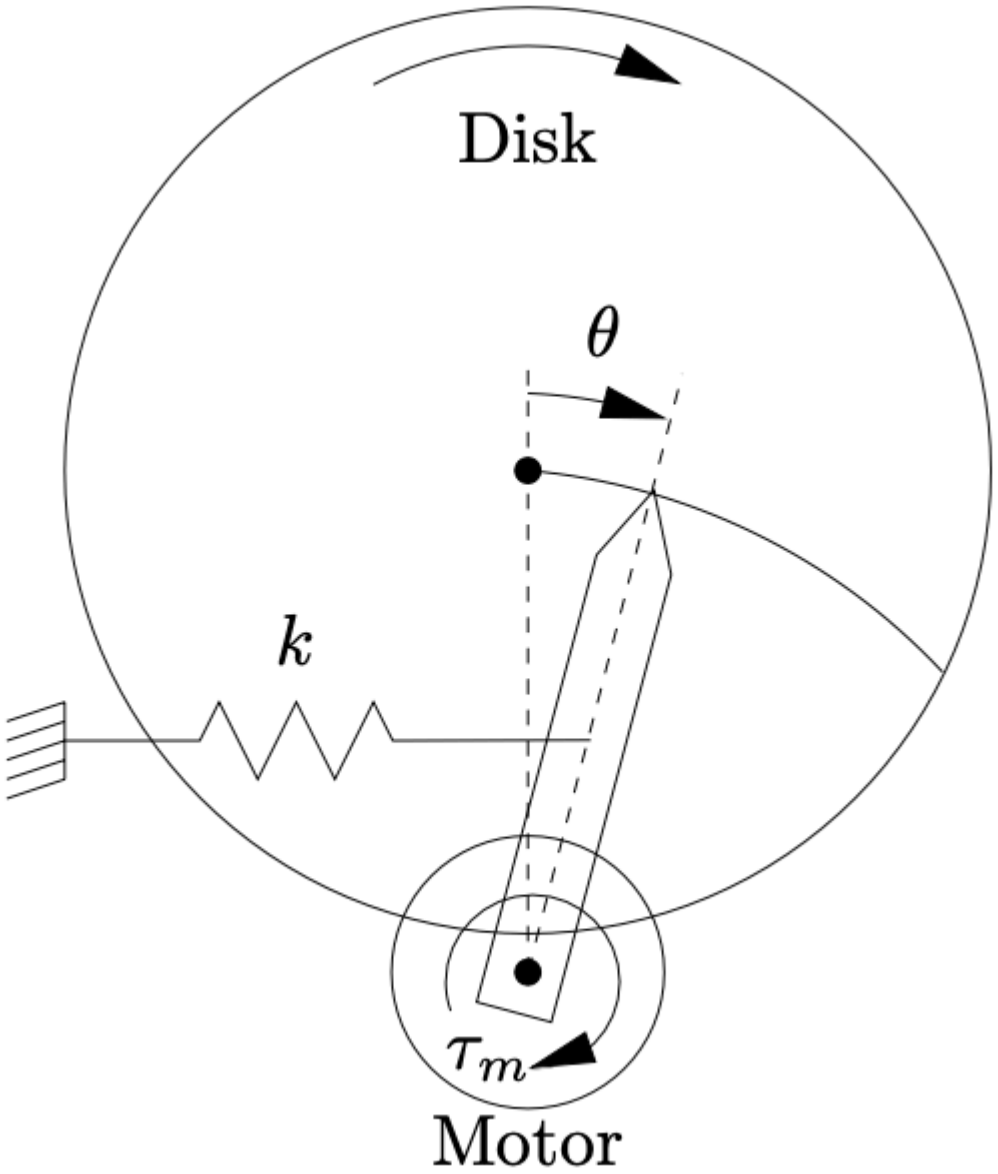
Consider a simple mechanism for positioning a mechanical arm whose equations of motion are given by

$$J\ddot{\theta} = -b\dot{\theta} - kr\sin\theta + \tau_{\mathrm{m}},$$

which can be written in state space form as

$$\frac{d}{dt}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\dot{\theta} \\ -kr\sin\theta/J - b\dot{\theta}/J\end{bmatrix} + \begin{bmatrix}0 \\ 1/J\end{bmatrix}\tau_{\mathrm{m}}.$$

The system consists of a spring loaded arm that is driven by a motor, as shown below.



The motor applies a torque that twists the arm against a linear spring and moves the end of the arm across a rotating platter. The input to the system is the motor torque τ_{m}. The force exerted by the spring is a nonlinear function of the head position due to the way it is attached.

The system parameters are given by

$$k = 1, \quad J = 100, \quad b = 10, \quad r = 1, \quad l = 2, \quad \epsilon = 0.01.$$

and we assume that time is measured in msec and distance in cm. (The constants here are made up and don't necessarily reflect a real disk drive, though the units and time constants are motivated by computer disk drives.)

The system dynamics can be modeled in python-control using a `NonlinearIOSystem` object, which we create with the `nlsys` function:

In [2]:

```python
# Parameter values
servomech_params = {
    'J': 100,              # Moment of inertial of the motor
    'b': 10,               # Angular damping of the arm
    'k': 1,                # Spring constant
    'r': 1,                # Location of spring contact on arm
    'l': 2,                # Distance to the read head
    'eps': 0.01,           # Magnitude of velocity-dependent perturbation
}

# State derivative
def servomech_update(t, x, u, params):
    # Extract the configuration and velocity variables from the state vector
    theta = x[0]                # Angular position of the disk drive arm
    thetadot = x[1]             # Angular velocity of the disk drive arm
    tau = u[0]                  # Torque applied at the base of the arm

    # Get the parameter values
    J, b, k, r = map(params.get, ['J', 'b', 'k', 'r'])

    # Compute the angular acceleration
    dthetadot = 1/J * (
        -b * thetadot - k * r * np.sin(theta) + tau)

    # Return the state update law
    return np.array([thetadot, dthetadot])

# System output (end of arm)
def servomech_output(t, x, u, params):
    l = params['l']
    return np.array([l * x[0]])

# System dynamics
servomech = ct.nlsys(
    servomech_update, servomech_output, name='servomech',
    params=servomech_params,
    states=['theta_', 'thdot_'],
    outputs=['y'], inputs=['tau'])

print(servomech)
print("\nParams:", servomech.params)
```

```
<NonlinearIOSystem>: servomech
Inputs (1): ['tau']
Outputs (1): ['y']
States (2): ['theta_', 'thdot_']

Update: <function servomech_update at 0x13e018f40>
Output: <function servomech_output at 0x13e018ea0>

Params: {'J': 100, 'b': 10, 'k': 1, 'r': 1, 'l': 2, 'eps': 0.01}
```

Linearization

To study the open loop dynamicsof the system, we compute the linearization of the dynamics about the equilibrium point corresponding to $\theta_e = 15^\circ$.

In [3]:
```
# Convert the equilibrium angle to radians
theta_e = (15 / 180) * np.pi

# Compute the input required to hold this position
u_e = servomech.params['k'] * servomech.params['r'] * np.sin(theta_e)
print("Equilibrium torque = %g" % u_e)

# Linearize the system about the equilibrium point
P = servomech.linearize([theta_e, 0], u_e, name='P')
P.name = 'P'   # Bug
print("Linearized dynamics:", P)
print("Zeros: ", P.zeros())
print("Poles: ", P.poles())
print("")

# Transfer function representation
P_tf = ct.tf(P)
print(P_tf)
```

```
Equilibrium torque = 0.258819
Linearized dynamics: <StateSpace>: P
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']
States (2): ['x[0]', 'x[1]']

A = [[ 0.          1.        ]
     [-0.00965926 -0.1       ]]

B = [[0.  ]
     [0.01]]

C = [[2. 0.]]

D = [[0.]]

Zeros:  []
Poles:  [-0.05+0.08461239j -0.05-0.08461239j]

<TransferFunction>: P$converted
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


         0.02
----------------------
s^2 + 0.1 s + 0.009659
```

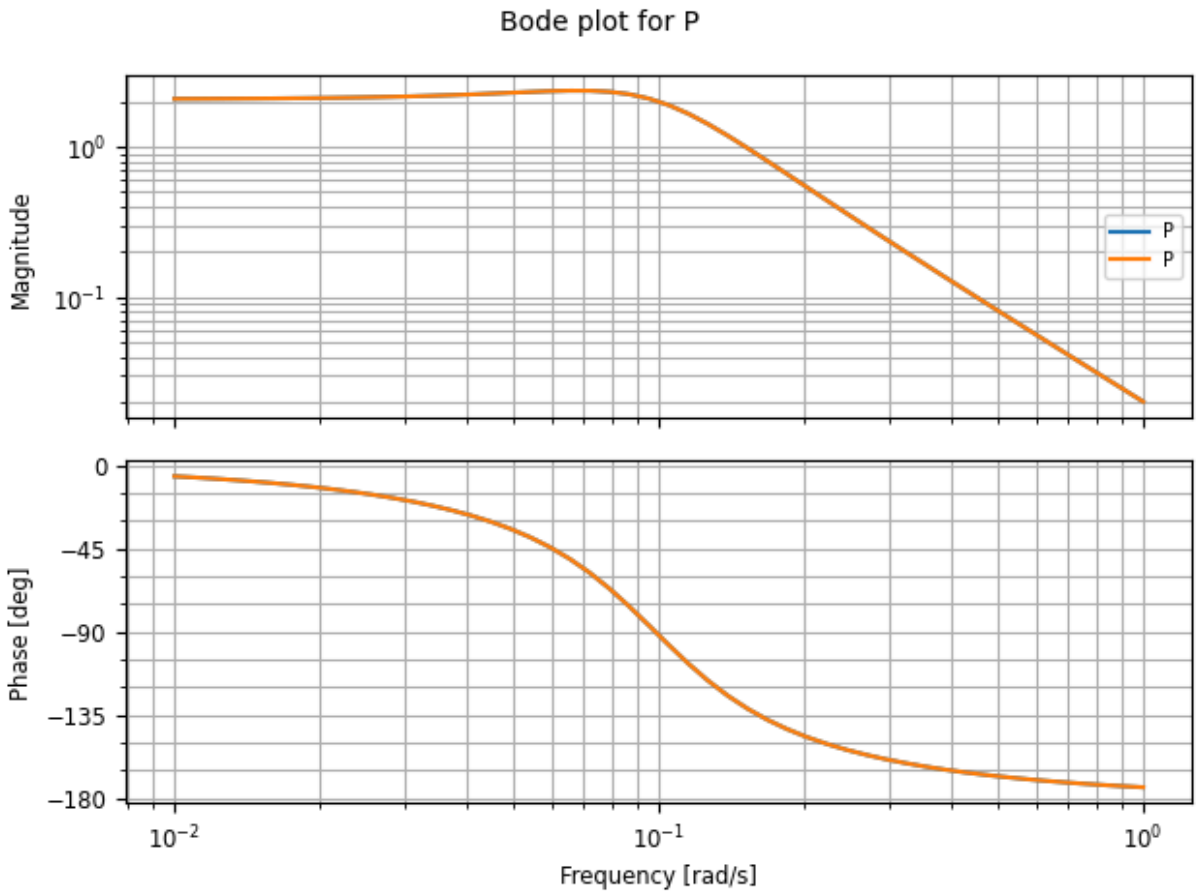
Open loop frequency response

A standard method for understanding the dynamics is to plot the output of the system in response to sinusoids with unit magnitude at differening frequencies.

We use the `frequency_response` function to plot the step response of the linearized, open-loop system.

In [4]:
```
# Frequency response
freqresp = ct.frequency_response(P, np.logspace(-2, 0))
freqresp.plot()

# Equivalent command
ct.bode_plot(P, np.logspace(-2, 0))
```

Out[4]:
```
array([[list([<matplotlib.lines.Line2D object at 0x13e0efd10>])],
       [list([<matplotlib.lines.Line2D object at 0x13e0efb90>])]],
      dtype=object)
```



Feedback control design

We next design a feedback controller for the system using a proportional integral controller, which has transfer function

$$C(s) = \frac{k_\mathrm{p} s + k_\mathrm{i}}{s}$$

We will learn how to choose k_p and k_i more formally in W9. For how we just pick different values to see how the dynamics are impacted.

In [5]:
```
kp = 1
ki = 1

# Create tf from numerator/denominator coefficients
C = ct.tf([kp, ki], [1])
print(C)

# Alternative method: define "s" and use algebra
s = ct.tf('s')
C = kp + ki/s
C.name = 'C'
print(C)
```

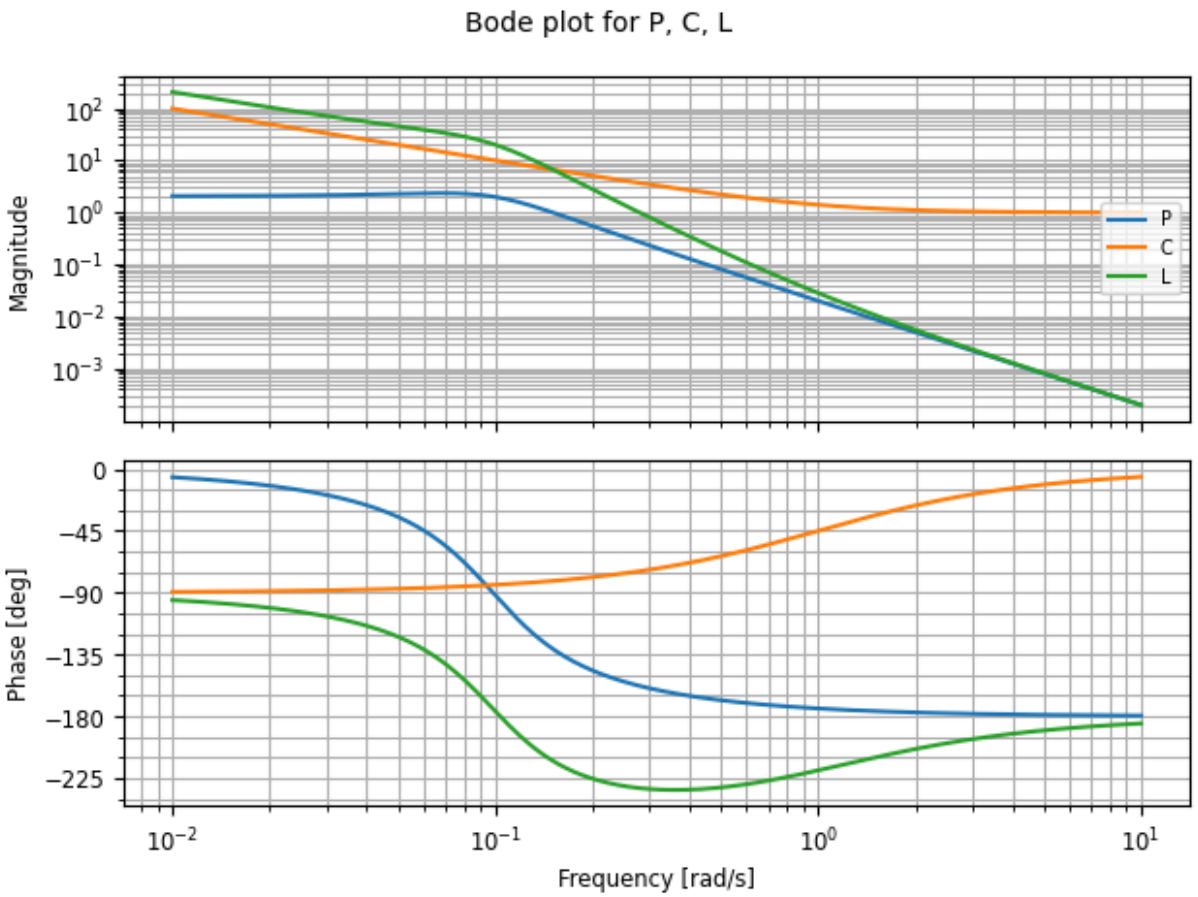
```
<TransferFunction>: sys[4]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


s + 1
-----
  1

<TransferFunction>: C
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


s + 1
-----
  s
```

In [6]:
```
# Implementation: use state space "realization"
def ctrl_update(t, x, u, params):
  return u
def ctrl_output(t, x, u, params):
  return kp * u + ki * x
ctrl = ct.nlsys(ctrl_update, ctrl_output, inputs=1, outputs=1, states=1)
print(ct.tf(ctrl.linearize(0, 0)))
```

```
<TransferFunction>: sys[11]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


s + 1
-----
  s
```

In [7]:
```
# Loop transfer function
L = P * C
```

```
L.name = 'L'
ct.bode_plot([P, C, L]);
```

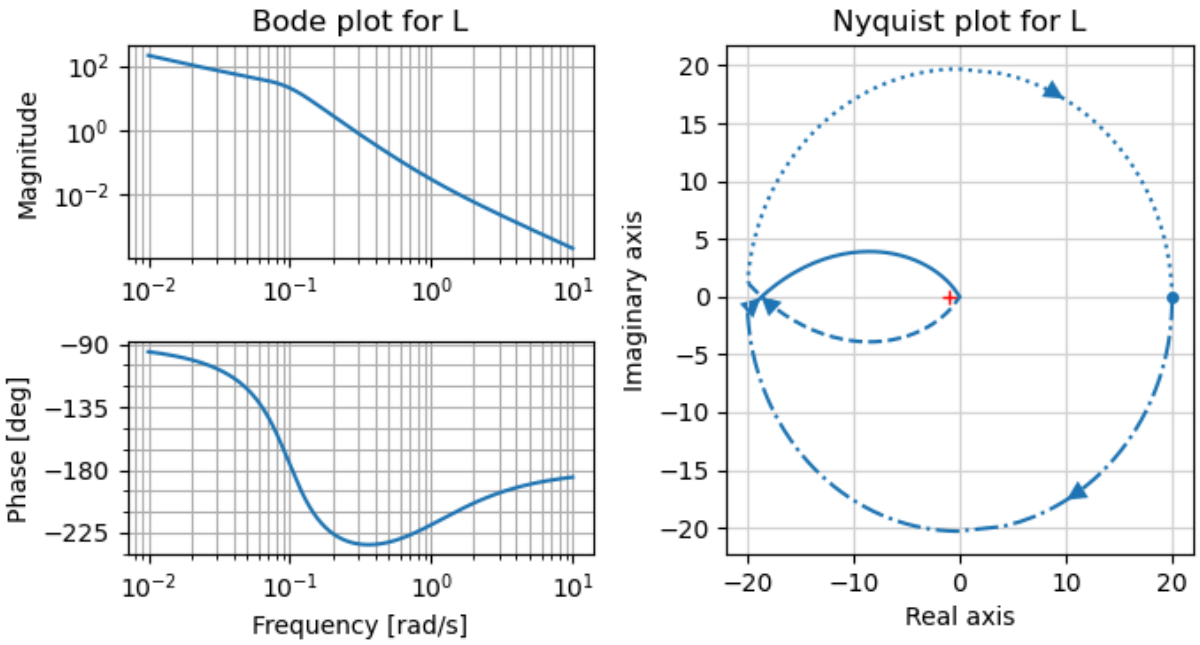


Note that L = P * C corresponds to addition in both the magnitude and the phase.

Nyquist analysis

To check stability (and eventually robustness), we use the Nyquist criterion.

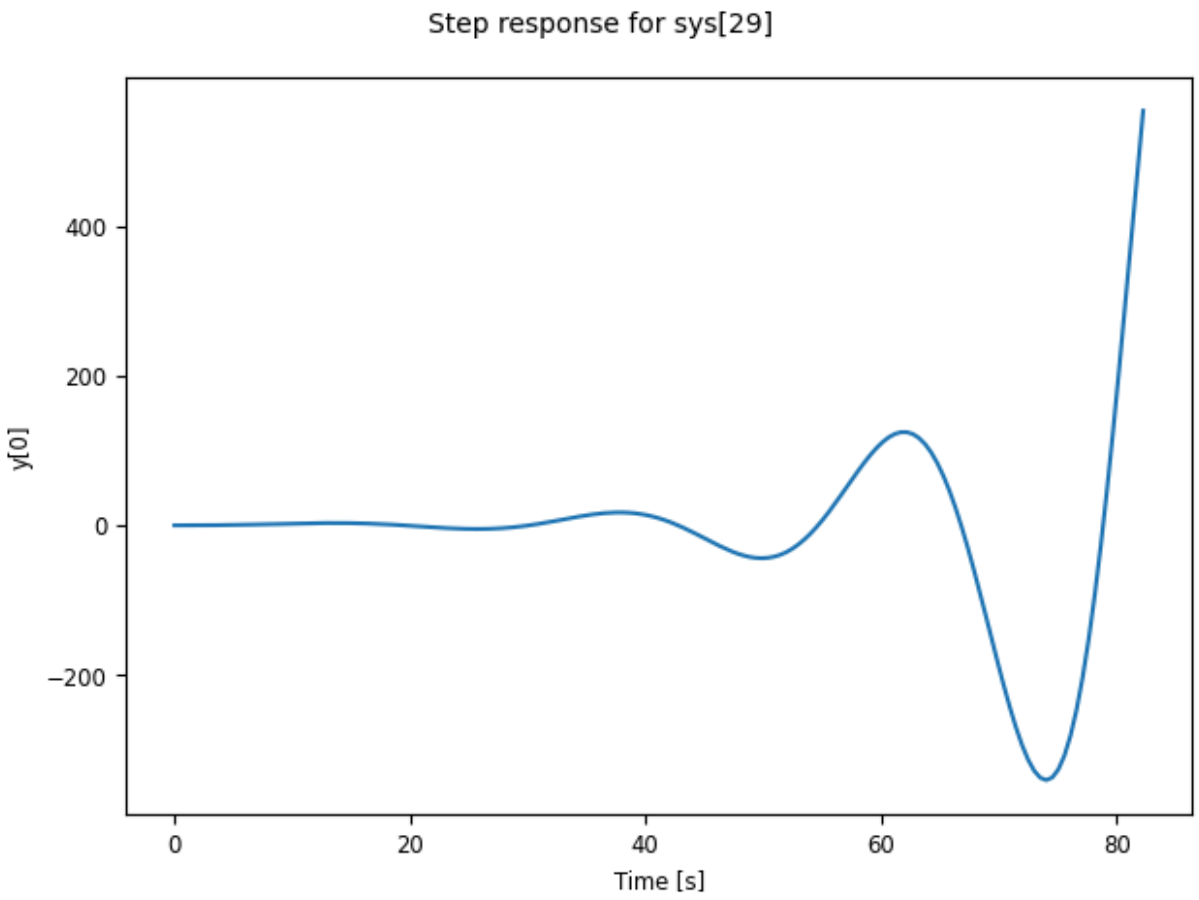
In [8]:
```
plt.figure(figsize=[7, 4])
ax1 = plt.subplot(2, 2, 1)
plt.title("Bode plot for L")
ax2 = plt.subplot(2, 2, 3)
ct.bode_plot(L, ax=np.array([[ax1], [ax2]]))

plt.subplot(1, 2, 2)
ct.nyquist_plot(L)
plt.title("Nyquist plot for L")

plt.suptitle("")
plt.tight_layout()
```



We see from this plot that the loop transfer function encircles the -1 point => closed loop system should be unstable. We can check this by making use of additional features of Nyquist analysis.

In [9]:
```
# Get the Nyquist *response*, so that we can get back encirclements
nyqresp = ct.nyquist_response(L)
print("N = encirclements: ", nyqresp.count)
print("P = RHP poles of L: ", np.sum(np.real(L.poles()) > 0))
print("Z = N + P = RHP zeros of 1 + L:", np.sum(np.real((1 + L).zeros()) >=
print("Poles of L = ", L.poles())
print("")

T = ct.feedback(L)
ct.step_response(T).plot();
```

```
N = encirclements:  2
P = RHP poles of L:  0
Z = N + P = RHP zeros of 1 + L: 2
Poles of L =  [-0.05+0.08461239j -0.05-0.08461239j -0.  +0.j        ]
```



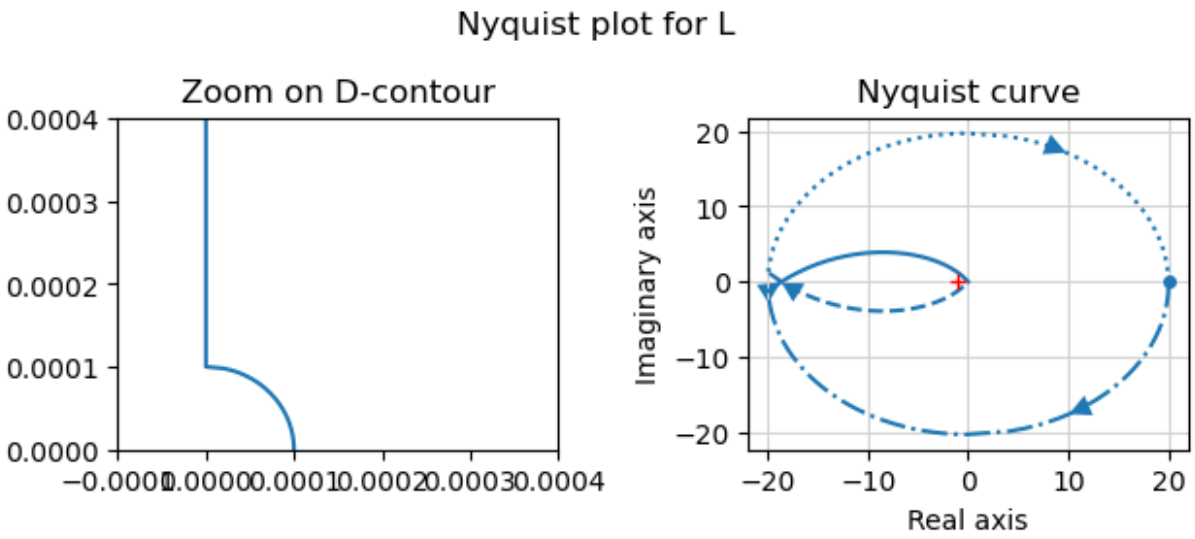
Poles on the $j\omega$ axis

Note that we have a pole at 0 (due to the integrator in the controller). How is this handled?

A: use a small loop to the right around poles on the $j\omega$ axis => not inside the contour.

To see this, we use the `nyquist_response` function, which returns the contour used to compute the Nyquist curve. If we zoom in on the contour near the origin, we see how the outer edge of the Nyquist curve is computed.

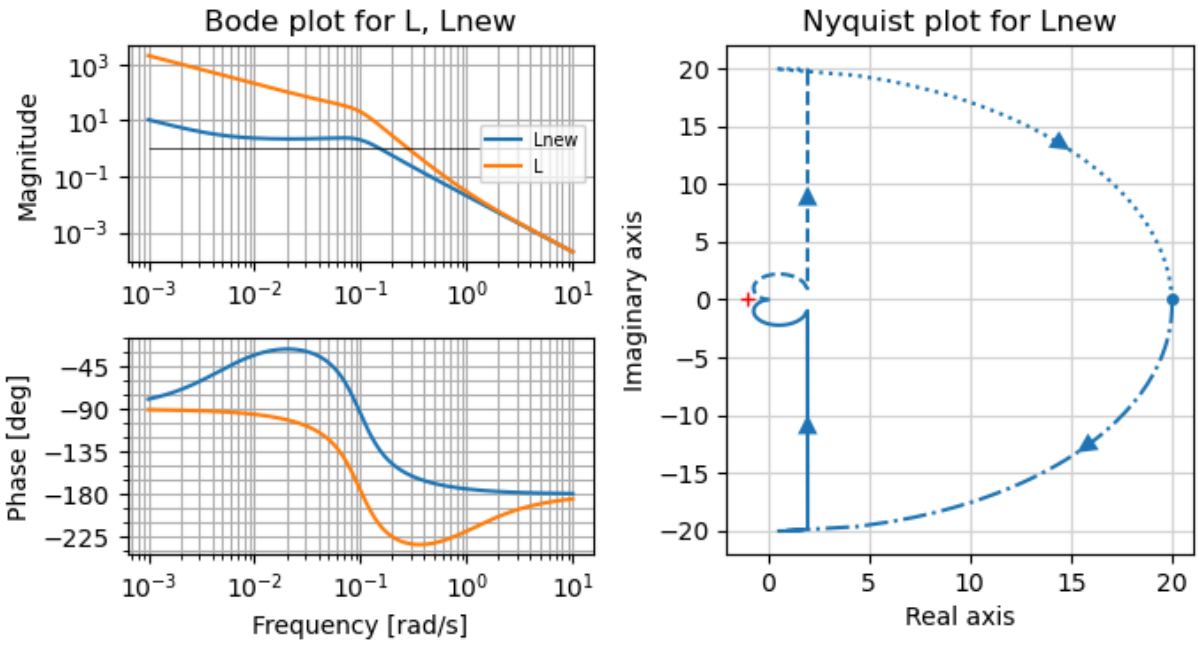
In [10]:
```
# Plot the D contour
ax = plt.subplot(2, 2, 1)
plt.plot(np.real(nyqresp.contour), np.imag(nyqresp.contour))
plt.axis([-1e-4, 4e-4, 0, 4e-4])
plt.title("Zoom on D-contour")

plt.subplot(2, 2, 2)
ct.nyquist_plot(L)
plt.title("Nyquist curve")

plt.tight_layout()
```



Updated feedback control design

We now redesign the control system to give something that is stable. We can do this by moving the zero for the controller to a lower frequency, so that the phase lag from the integrator does not overlap with the phase lag from the system dynamics.

In [11]:
```
# Change the frequency response to avoid crossing over -180 with large gain
Cnew = kp + (ki/200)/s
Cnew.name = 'Cnew'
Lnew = P * Cnew
Lnew.name = 'Lnew'

plt.figure(figsize=[7, 4])
ax1 = plt.subplot(2, 2, 1)
plt.title("Bode plot for L, Lnew")
ax2 = plt.subplot(2, 2, 3)
ct.bode_plot([Lnew, L], ax=np.array([[ax1], [ax2]]))
ax1.loglog([1e-3, 1e1], [1, 1], 'k', linewidth=0.5)

plt.subplot(1, 2, 2)
ct.nyquist_plot(Lnew)
plt.title("Nyquist plot for Lnew")

plt.suptitle("")
plt.tight_layout()
```

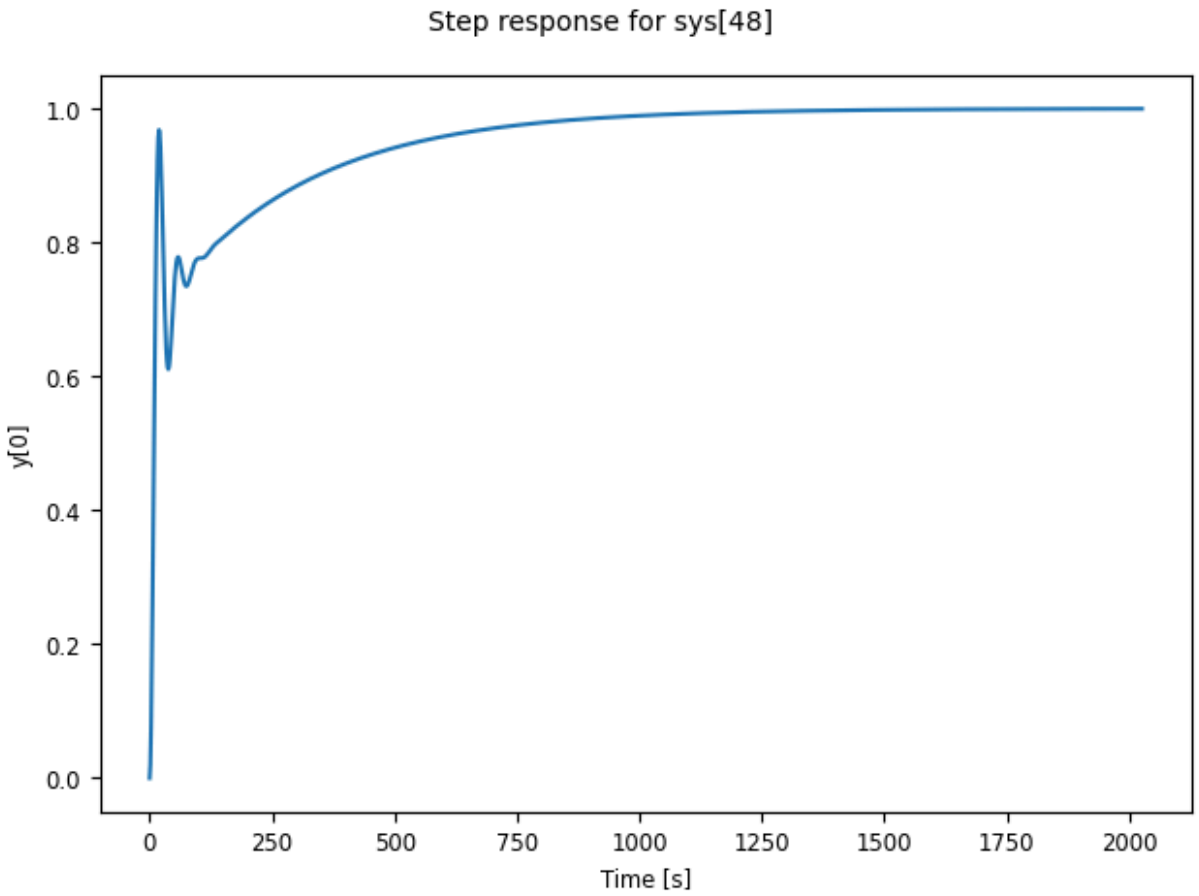


We see now that we have no encirclements, and so the system should be stable.

Note however that the Nyquist curve is close to the -1 point => not *that* stable.

In [12]:
```
# Compute the transfer function from r to y
Tnew = ct.feedback(Lnew)
ct.step_response(Tnew).plot()
```

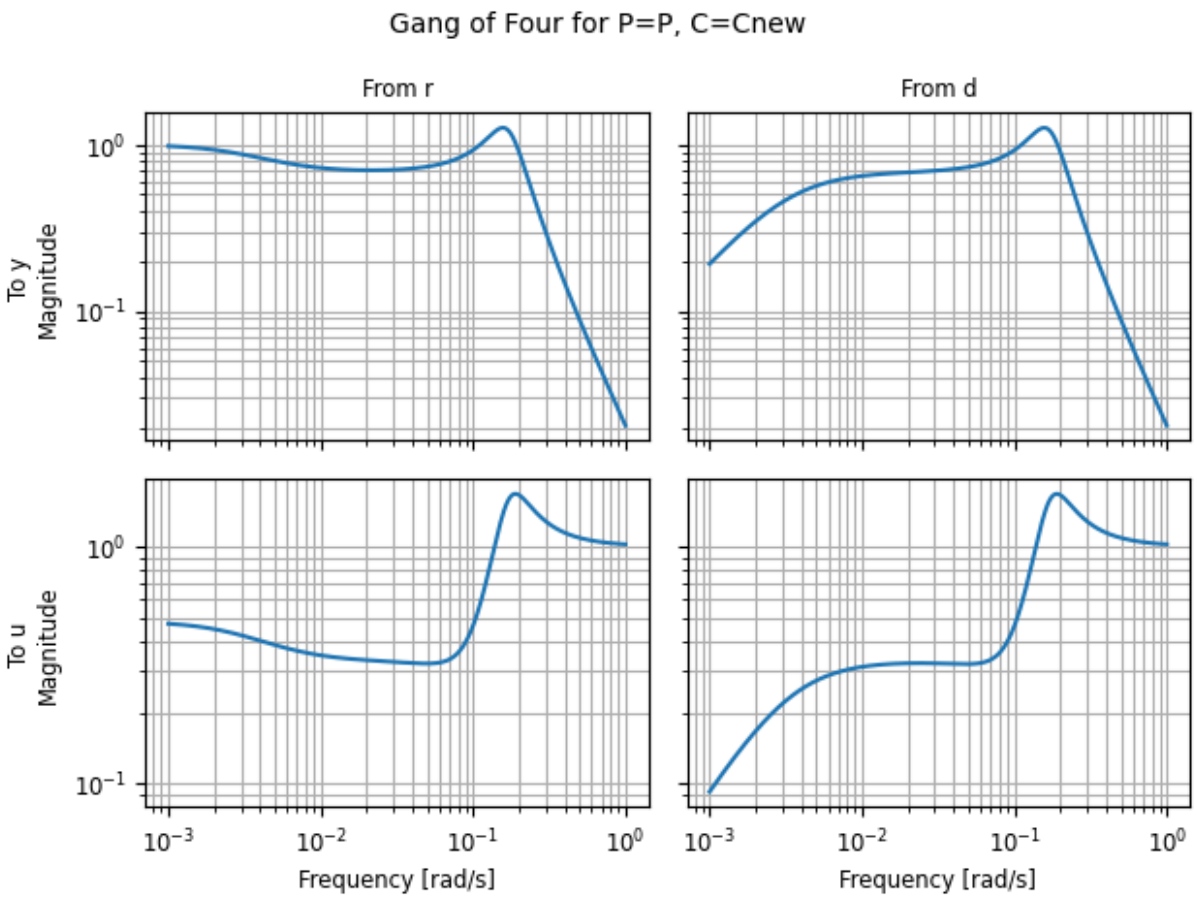
Out[12]:
```
array([[list([<matplotlib.lines.Line2D object at 0x13eeea7b0>])]],
      dtype=object)
```



Closed loop frequency response

We can also look at the closed loop frequency response to understand how different inputs affect different outputs. The `gangof4` function computes the standard transfer functions:

In [13]:
```
ct.gangof4(P, Cnew);
```



Stability margins

Another standard set of analysis tools is to identify the gain, phase, and stability margins for the sytem:

- **Gain margin:** the maximimum amount of additional gain that we can put into the loop and still maintain stability.
- **Phase margin:** the maximum amount of additional phase (lag) that we can put into the loop and still maintain stability.
- **Stability margin:** the maximum amount of combined gain and phase at the critical frequency that can be put into the loop and still maintain stability.

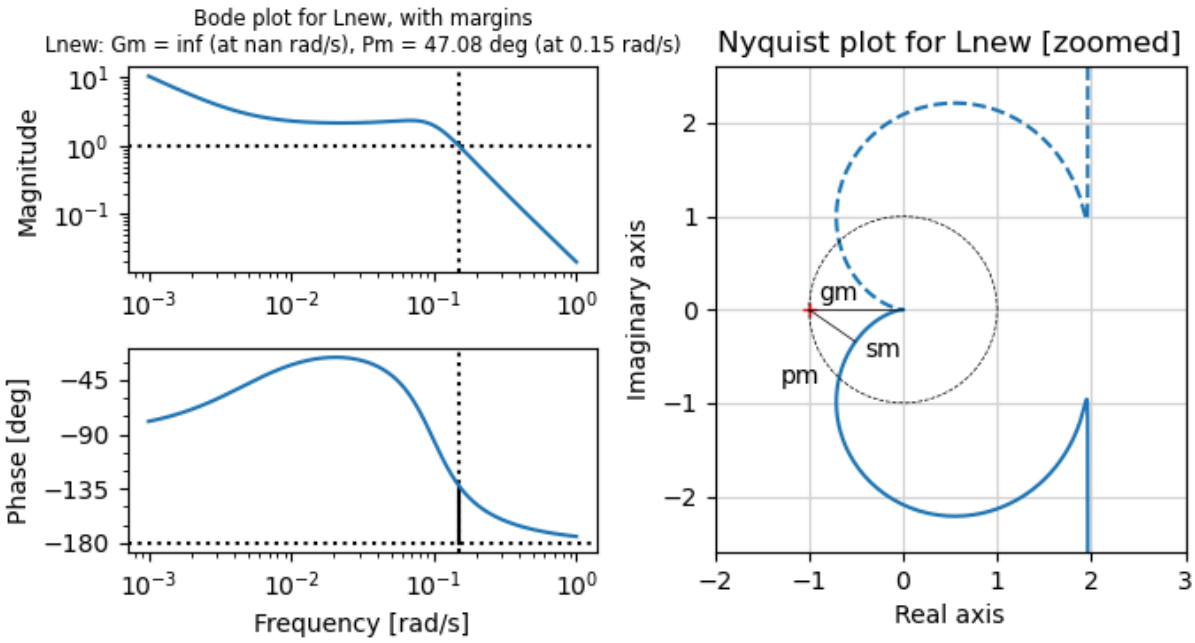
The first two of the items can be computed either by looking at the frequeny response or by using the `margin` command.

The stabilty margin is the minimum distance between -1 and $L(jw)$, which is just the minimum value of $|1 - L(j\omega)

In [14]:
```
plt.figure(figsize=[7, 4])

# Gain and phase margin on Bode plot
ax1 = plt.subplot(2, 2, 1)
plt.title("Bode plot for Lnew, with margins")
ax2 = plt.subplot(2, 2, 3)
```

```
ct.bode_plot(Lnew, ax=np.array([[ax1], [ax2]]), margins=True)

# Compute gain and phase margin
gm, pm, wpc, wgc = ct.margin(Lnew)
print(f"Gm = {gm:2.2g} (at {wpc:.2g} rad/s)")
print(f"Pm = {pm:3.2g} deg (at {wgc:.2g} rad/s)")

# Compute the stability margin
resp = ct.frequency_response(1 + Lnew)
sm = np.min(resp.magnitude)
wsm = resp.omega[np.argmin(resp.magnitude)]
print(f"Sm = {sm:2.2g} (at {wsm:.2g} rad/s)")

# Plot the Nyquist curve
plt.subplot(1, 2, 2)
ct.nyquist_plot(Lnew)
plt.title("Nyquist plot for Lnew [zoomed]")
plt.axis([-2, 3, -2.6, 2.6])

#
# Annotate it to see the margins
#

# Gain margin (special case here, since infinite)
Lgm = 0
plt.plot([-1, Lgm], [0, 0], 'k-', linewidth=0.5)
plt.text(-0.9, 0.1, "gm")

# Phase margin
theta = np.linspace(0, 2 * pi)
plt.plot(np.cos(theta), np.sin(theta), 'k--', linewidth=0.5)
plt.text(-1.3, -0.8, "pm")

# Stability margin
Lsm = Lnew(wsm * 1j)
plt.plot([-1, Lsm.real], [0, Lsm.imag], 'k-', linewidth=0.5)
plt.text(-0.4, -0.5, "sm")

plt.suptitle("")
plt.tight_layout()
```

```
Gm = inf (at nan rad/s)
Pm =  47 deg (at 0.15 rad/s)
Sm = 0.6 (at 0.19 rad/s)
```



Unstable system: inverted pendulum

When we have a system that is open loop unstable, the Nyquist curve will need to have encirclements to be stable. In this case, the interpreation of the various characteristics can be more complicated.

To explore this, we consider a simple model for an inverted pendulum, which has (normalized) dynamics:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & 0.1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, \qquad y = \begin{bmatrix} 1 & 0 \end{bmatrix} x$$

Transfer function for the system can be shown to be

$$P(s) = \frac{1}{s^2 + 0.1s - 1}.$$

This system is unstable, with poles $\sim \pm 1$.

In [15]:
```
P = ct.tf([1], [1, 0.1, -1])
P.poles()
```

Out[15]:
```
array([-1.05124922+0.j,  0.95124922+0.j])
```

PD controller

We construct a proportional-derivative (PD) controller for the system,

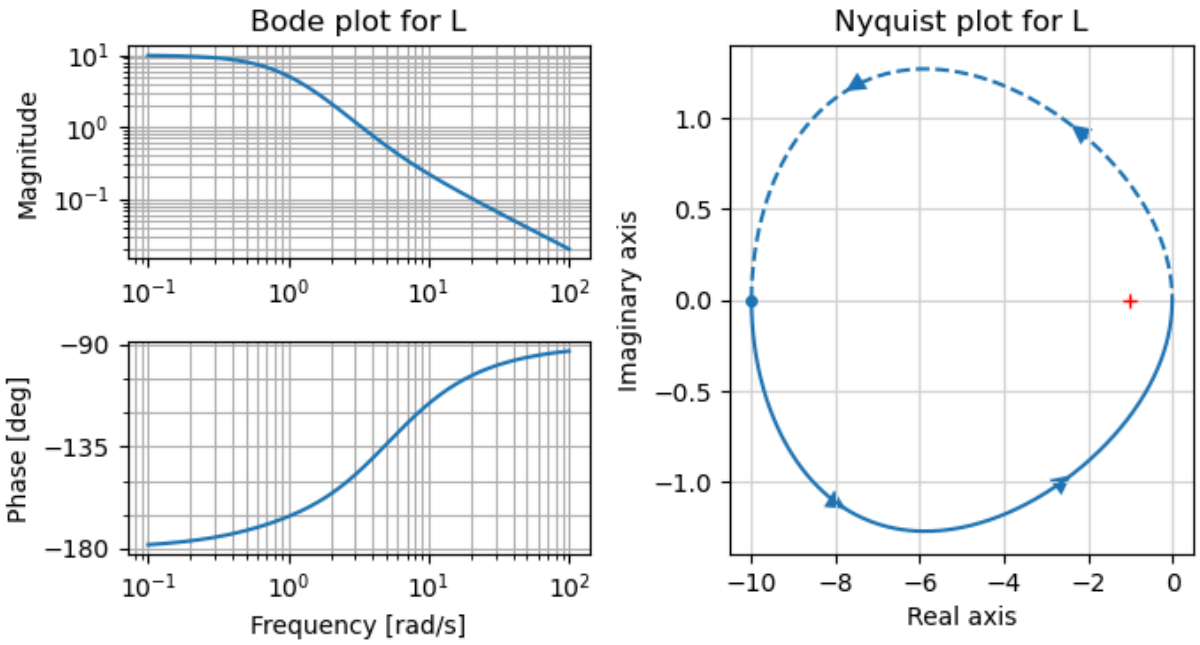
$$u = k_\mathrm{p} e + k_\mathrm{d} \dot{e}$$

which is roughly the equivalent of using state feedback (since the system states are θ and $\dot{\theta}$).

In [16]:
```
# Transfer function for a PD controller
kp = 10
kd = 2
C = ct.tf([kd, kp], [1])

# Loop transfer function
L = P * C
L.name = 'L'
print(L)
print("Zeros: ", L.zeros())
print("Poles: ", L.poles())
```

```
<TransferFunction>: L
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


   2 s + 10
---------------
s^2 + 0.1 s - 1

Zeros:  [-5.+0.j]
Poles:  [-1.05124922+0.j  0.95124922+0.j]
```

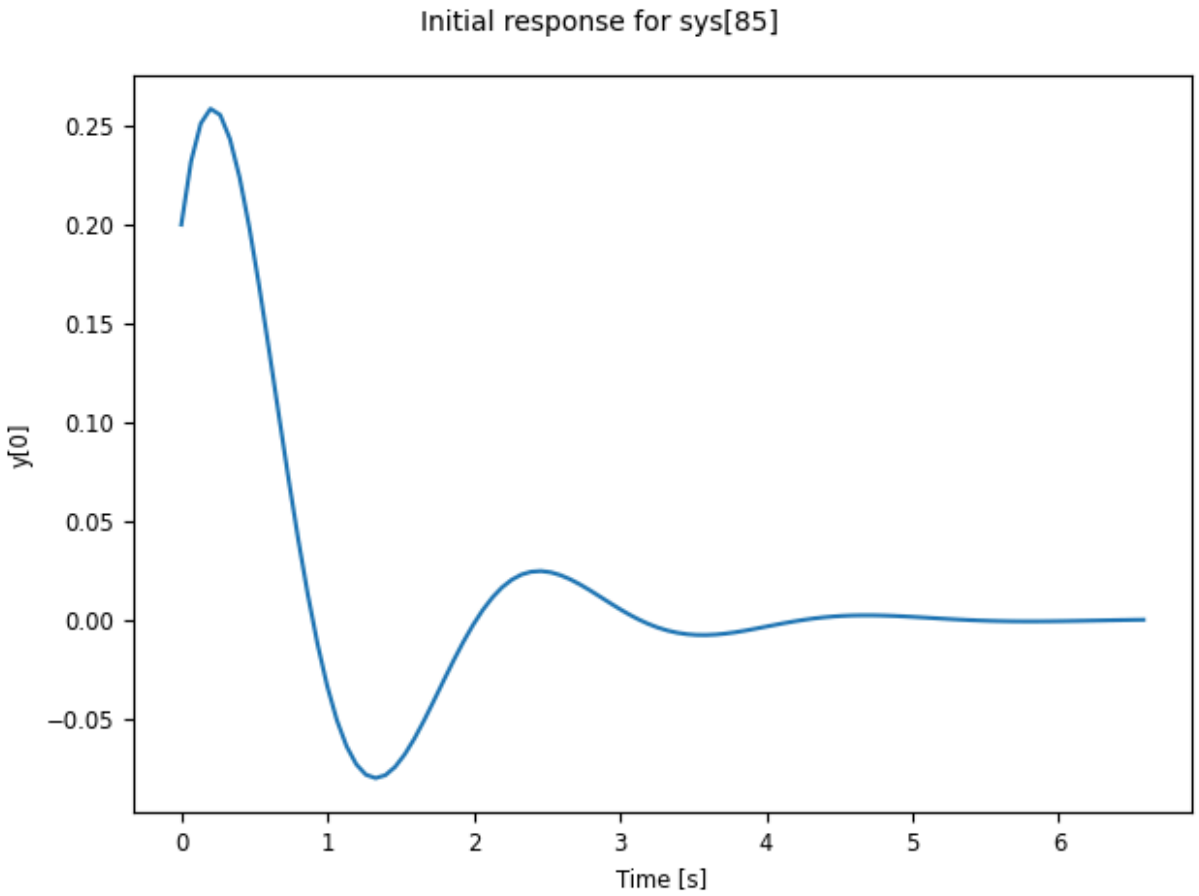
In [17]:
```
# Bode and Nyquist plots
plt.figure(figsize=[7, 4])
ax1 = plt.subplot(2, 2, 1)
plt.title("Bode plot for L")
ax2 = plt.subplot(2, 2, 3)
ct.bode_plot(L, ax=np.array([[ax1], [ax2]]))

plt.subplot(1, 2, 2)
ct.nyquist_plot(L)
plt.title("Nyquist plot for L")

plt.suptitle("")
plt.tight_layout()
```



In [18]:
```
# Check the Nyquist criterion
nyqresp = ct.nyquist_response(L)
print("N = encirclements: ", nyqresp.count)
print("P = RHP poles of L: ", np.sum(np.real(L.poles()) > 0))
print("Z = N + P = RHP zeros of 1 + L:", np.sum(np.real((1 + L).zeros()) >=
print("Poles of L = ", L.poles())
print("Zeros of 1 + L = ", (1 + L).zeros())
print("")

T = ct.feedback(L)
ct.initial_response(T, X0=[0.1, 0]).plot();
```

```
N = encirclements:  -1
P = RHP poles of L:  1
Z = N + P = RHP zeros of 1 + L: 0
Poles of L =  [-1.05124922+0.j  0.95124922+0.j]
Zeros of 1 + L =  [-1.05+2.8102491j -1.05-2.8102491j]
```

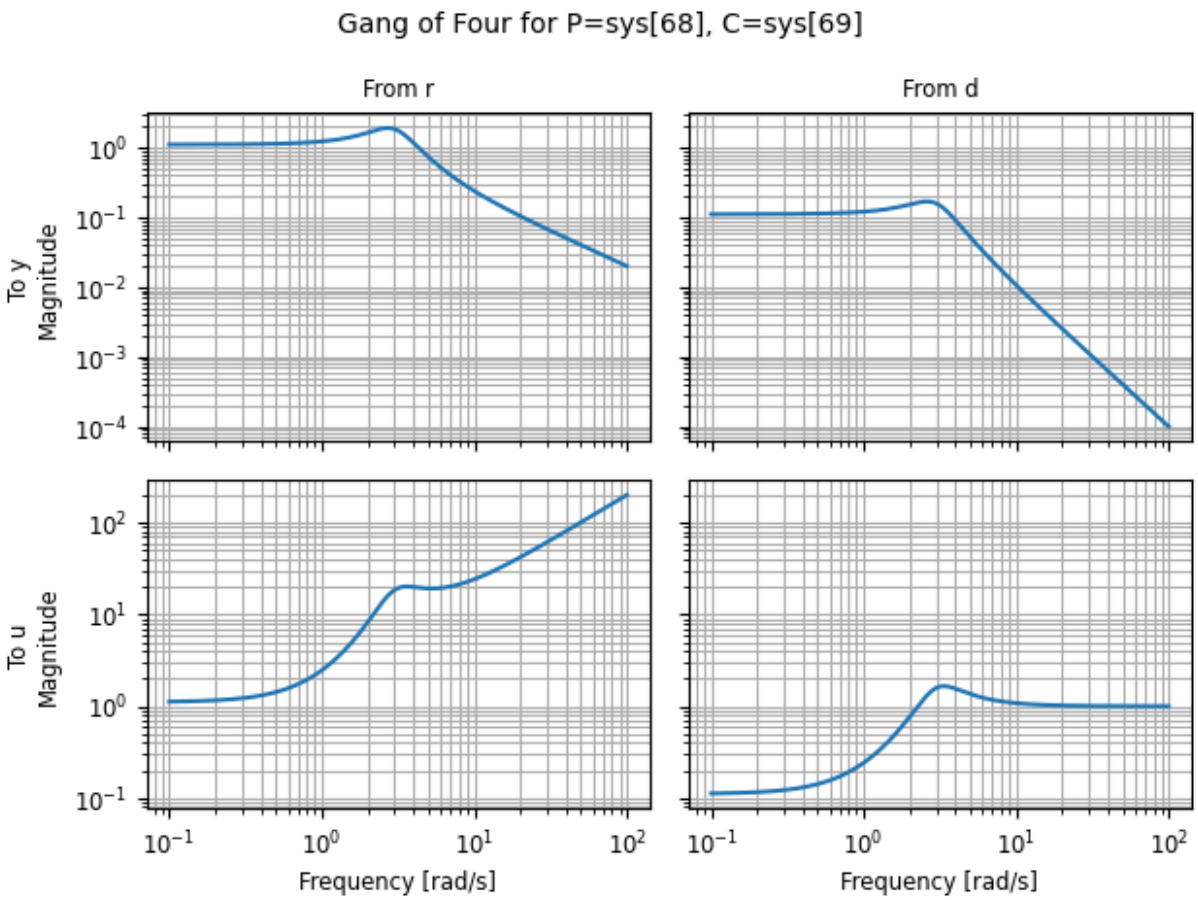
```
/Users/murray/miniconda3/envs/cds110/lib/python3.12/site-packages/control/ti
meresp.py:1009: UserWarning: Non-zero initial condition given for transfer f
unction system. Internal conversion to state space used; may not be consiste
nt with given X0.
  warnings.warn(
```



Gang of 4

Another useful thing to look at is the transfer functions from noise and disturbances to the system outputs and inputs:

In [19]:
```
ct.gangof4(P, C);
```



We see that the response from the input r (or equivalently noise n) to the process input is very large for large frequencies. This means that we are amplifying high frequency noise (and comes from the fact that we used derivative feedback).

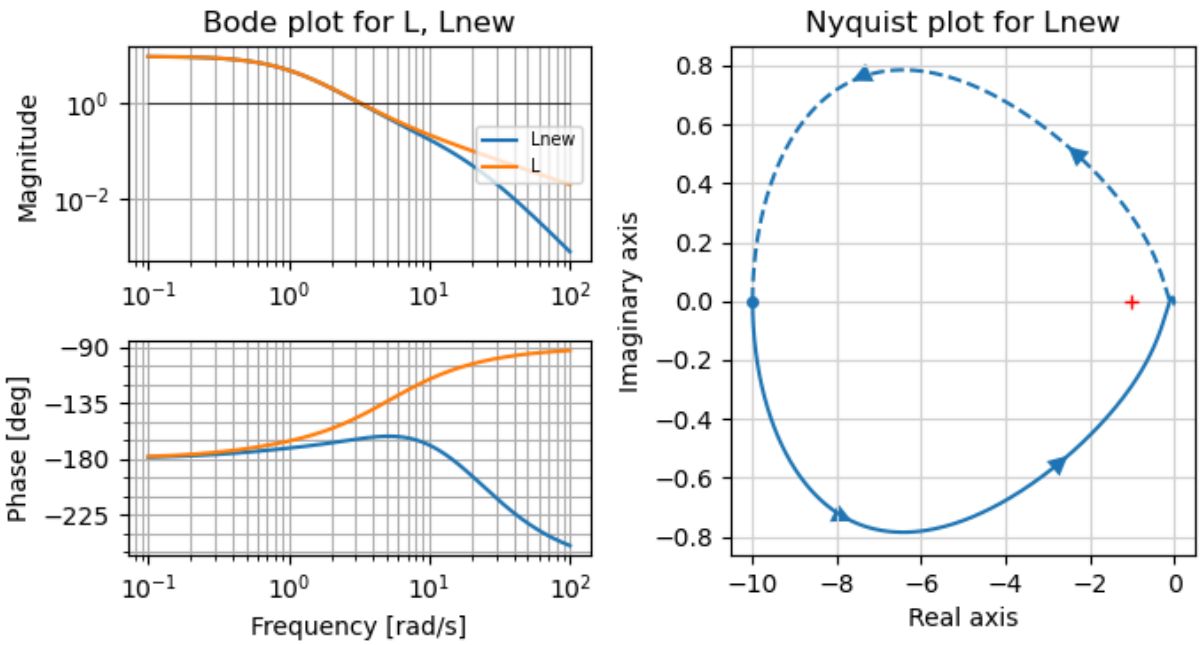
High frequency rolloff

We can attempt to resolve this by "rolling off" the derivative action at high frequencies:

In [20]:
```
Cnew = (kp + kd * s) / (s/20 + 1)**2
Cnew.name = 'Cnew'
print(Cnew)

Lnew = P * Cnew
Lnew.name = 'Lnew'

plt.figure(figsize=[7, 4])
ax1 = plt.subplot(2, 2, 1)
plt.title("Bode plot for L, Lnew")
ax2 = plt.subplot(2, 2, 3)
ct.bode_plot([Lnew, L], ax=np.array([[ax1], [ax2]]))
ax1.loglog([1e-1, 1e2], [1, 1], 'k', linewidth=0.5)

plt.subplot(1, 2, 2)
ct.nyquist_plot(Lnew)
plt.title("Nyquist plot for Lnew")
```

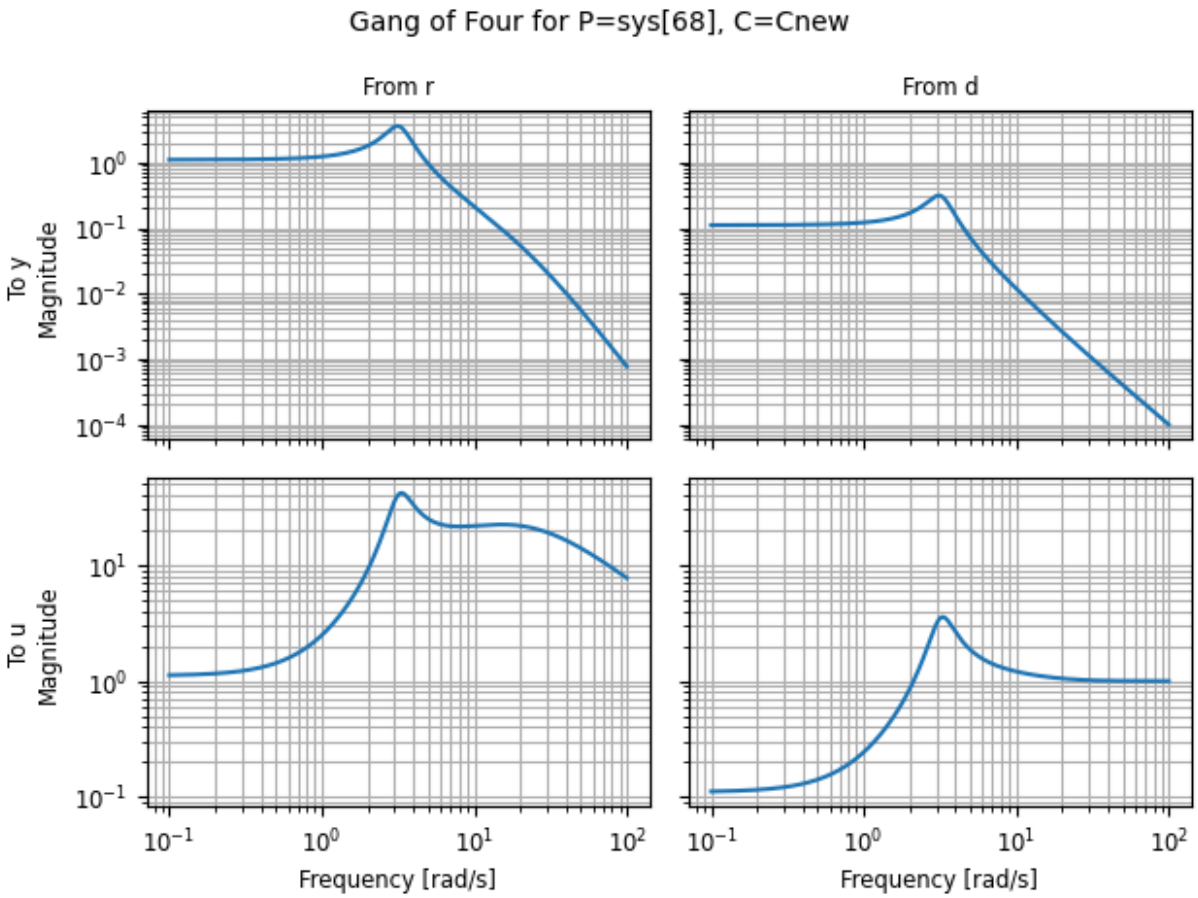
```
plt.suptitle("")
plt.tight_layout()
```

```
<TransferFunction>: Cnew
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']


  800 s + 4000
----------------
s^2 + 40 s + 400
```



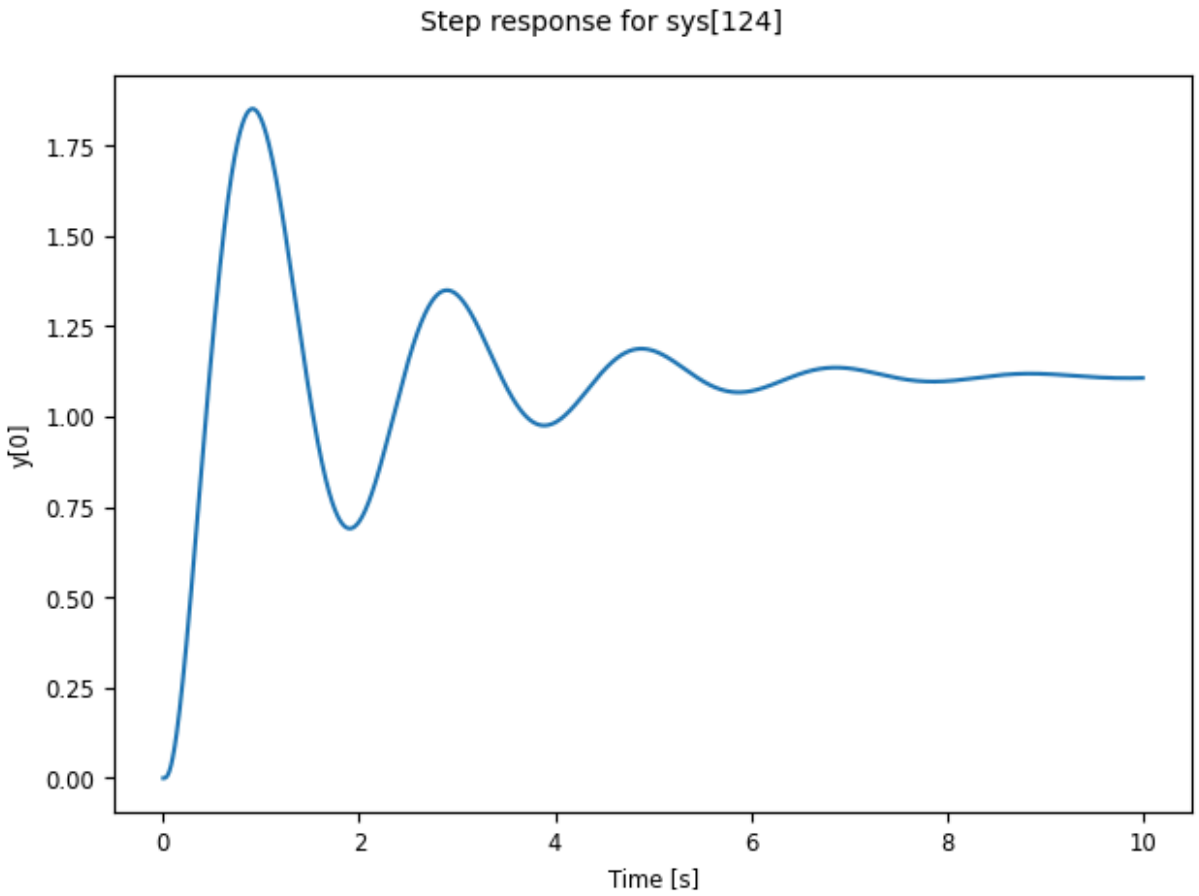
While not (yet) a very high performing controller, this change does get rid of the issues with the high frequency noise:

In [21]:
```
# Check the gang of 4
ct.gangof4(P, Cnew);
```



In [22]:
```
# See what the step response looks like
Tnew = ct.feedback(Lnew)
ct.step_response(Tnew, 10).plot()
```

Out[22]:
```
array([[list([<matplotlib.lines.Line2D object at 0x13f51f530>])]],
      dtype=object)
```


Step response for sys[124]
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
y[0]
0
2
4
6
8
10
Time [s]

In [ ]: