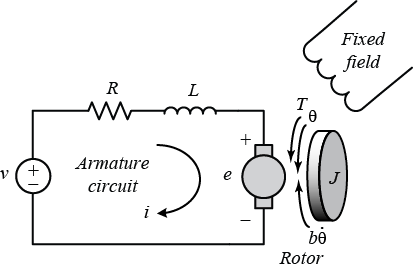
**DC Motor Speed: System Modeling**

## Physical setup

A common actuator in control systems is the DC motor. It directly provides rotary motion and, coupled with wheels or drums and cables, can provide translational motion. The electric equivalent circuit of the armature and the free-body diagram of the rotor are shown in the following figure.



For this example, we will assume that the input of the system is the voltage source ($V$) applied to the motor's armature, while the output is the rotational speed of the shaft $\dot{\theta}$. The rotor and shaft are assumed to be rigid. We further assume a viscous friction model, that is, the friction torque is proportional to shaft angular velocity.

The physical parameters for our example are:

(J) moment of inertia of the rotor 0.01 kg.m^2

(b) motor viscous friction constant 0.1 N.m.s

(Ke) electromotive force constant 0.01 V/rad/sec

(Kt) motor torque constant 0.01 N.m/Amp

(R) electric resistance 1 Ohm

(L) electric inductance 0.5 H

## System equations

In general, the torque generated by a DC motor is proportional to the armature current and the strength of the magnetic field. In this example we will assume that the magnetic field is constant and, therefore, that the motor torque is proportional to only the armature current $i$ by a constant factor $K_t$ as shown in the equation below. This is referred to as an armature-controlled motor.

(1)$$  T = K_{t} i$$

The back emf, $e$, is proportional to the angular velocity of the shaft by a constant factor $K_e$.

(2)$$  e = K_{e} \dot{\theta}$$

In SI units, the motor torque and back emf constants are equal, that is, $K_t = K_e$; therefore, we will use $K$ to represent both the motor torque constant and the back emf constant.

From the figure above, we can derive the following governing equations based on Newton's 2nd law and Kirchhoff's voltage law.

(3)$$ J\ddot{\theta} + b \dot{\theta} = K i $$

(4)$$ L \frac{di}{dt} + Ri = V - K\dot{\theta}$$

**1. Transfer Function**

Applying the Laplace transform, the above modeling equations can be expressed in terms of the Laplace variable *s*.

(5)$$ s(Js + b)\Theta(s) = KI(s) $$

(6)$$ (Ls + R)I(s) = V(s) - Ks\Theta(s) $$

We arrive at the following open-loop transfer function by eliminating $I(s)$ between the two above equations, where the rotational speed is considered the output and the armature voltage is considered the input.

(7)$$ P(s) = \frac {\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2} \qquad [ \frac{rad/sec}{V}] $$

**2. State-Space**

In state-space form, the governing equations above can be expressed by choosing the rotational speed and electric current as the state variables. Again the armature voltage is treated as the input and the rotational speed is chosen as the output.

(8)$$\frac{d}{dt}\left [\begin{array}{c} \dot{\theta} \\ \ \\ i \end{array} \right] =
\left [\begin{array}{cc} -\frac{b}{J} & \frac{K}{J} \\ \ \\ -\frac{K}{L} &
-\frac{R}{L} \end{array} \right] \left [\begin{array}{c} \dot{\theta} \\ \ \\ i \end{array} \right]  +
\left [\begin{array}{c} 0 \\ \ \\ \frac{1}{L} \end{array} \right] V$$

(9)$$ y = [ \begin{array}{cc}1 & 0\end{array}] \left [ \begin{array}{c} \dot{\theta} \\ \ \\ i
\end{array} \right] $$

## Design requirements

First consider that our uncompensated motor rotates at 0.1 rad/sec in steady state for an input voltage of 1 Volt (this is demonstrated in the [DC Motor Speed: System Analysis](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemAnalysis) page where the system's open-loop response is simulated). Since the most basic requirement of a motor is that it should rotate at the desired speed, we will require that the steady-state error of the motor speed be less than 1%. Another performance requirement for our motor is that it must accelerate to its steady-state speed as soon as it turns on. In this case, we want it to have a settling time less than 2 seconds. Also, since a speed faster than the reference may damage the equipment, we want to have a step response with overshoot of less than 5%.

In summary, for a unit step command in motor speed, the control system's output should meet the following requirements.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

## MATLAB representation

**1. Transfer Function**

We can represent the above open-loop transfer function of the motor in MATLAB by defining the parameters and transfer function as follows. Running this code in the command window produces the output shown below.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

s = tf('s');

P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2)

P\_motor =

0.01

---------------------------

0.005 s^2 + 0.06 s + 0.1001

Continuous-time transfer function.

**2. State Space**

We can also represent the system using the state-space equations. The following additional MATLAB commands create a state-space model of the motor and produce the output shown below when run in the MATLAB command window.

A = [-b/J K/J

-K/L -R/L];

B = [0

1/L];

C = [1 0];

D = 0;

motor\_ss = ss(A,B,C,D)

motor\_ss =

A =

x1 x2

x1 -10 1

x2 -0.02 -2

B =

u1

x1 0

x2 2

C =

x1 x2

y1 1 0

D =

u1

y1 0

Continuous-time state-space model.

The above state-space model can also be generated by converting your existing transfer function model into state-space form. This is again accomplished with the ss command as shown below.

motor\_ss = ss(P\_motor);

# DC Motor Speed: System Analysis

From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

(1)$$ s(Js + b)\Theta(s) = KI(s) $$

(2)$$ (Ls + R)I(s) = V(s) - Ks\Theta(s) $$

(3)$$  P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2}  \qquad  [ \frac{rad/sec}{V} ] $$

For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page.

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

## Open-loop response

First create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and type in the following commands (refer to the main problem for the details of getting these commands).

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

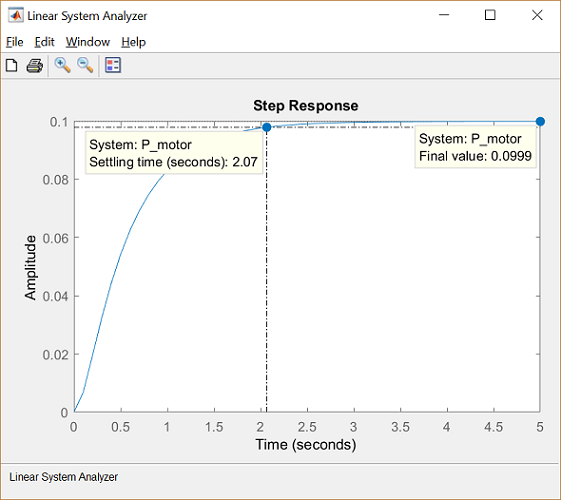
L = 0.5;

s = tf('s');

P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2);

Now let's see how the original open-loop system performs. Add the following linearSystemAnalyzer command onto the end of the m-file and run it in the MATLAB command window. You can access the **Linear System Analyzer** also by going to the **APPS** tab of the MATLAB toolstrip and clicking on the app icon under **Control System Design and Analysis**. In the command below, the string 'step' passed to the function specifies to generate a unit step response plot for the system P\_motor. The range of numbers 0:0.1:5 specify that the step response plot should include data points for times from 0 to 5 seconds in steps of 0.1 seconds. The resulting plot is shown in the figure below, where you can view some of the system's characteristics by right-clicking on the figure and choosing from the **Characteristics** menu such performance aspects as **Settling Time** and **Steady State**.

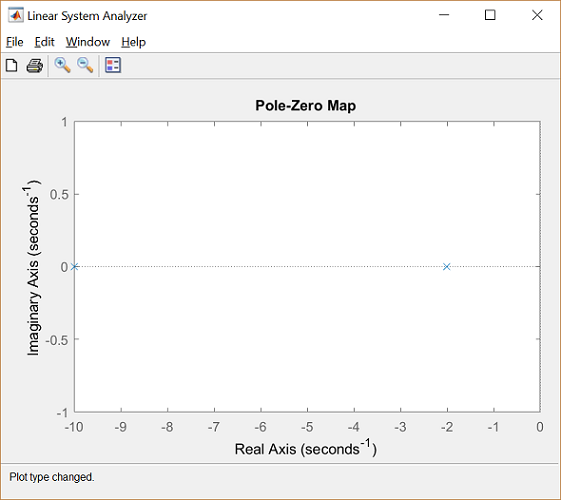
linearSystemAnalyzer('step', P\_motor, 0:0.1:5);



From the plot we see that when 1 Volt is applied to the system the motor can only achieve a maximum speed of 0.1 rad/sec, ten times smaller than our desired speed. Also, it takes the motor 2.07 seconds to reach its steady-state speed; this does not satisfy our 2 second settling time criterion.

## LTI model characteristics

Since our open-loop transfer function has the form of a canonical second-order system, we should be able to accurately predict the step response characteristics observed above based on the transfer function's pole locations. You can graphically see the location of the poles (and zeros) of the P\_motor system from within the **Linear System Analyzer** by right-clicking on the plot area and selecting **Plot Types > Pole/Zero** from the resulting menu. Performing this action will change the **Linear System Analyzer** to the following map where the blue x's identify the locations of poles.



From the above you can see that the open-loop transfer function has two real poles, one at *s* = -2 and one at *s* = -10. Since both poles are real, there is no oscillation in the step response (or overshoot) as we have already seen. Futhermore, since the one pole is 5 times more negative than the other, the slower of the two poles will dominate the dynamics. That is, the pole at *s* = -2 primarily determines the speed of response of the system and the system behaves similarly to a first-order system.

Let's see just how closely a first-order model approximates our original motor model. Enter the following command at the MATLAB command line to build a first-order transfer function with pole at *s* = -2 and steady-state value matching the original transfer function.

rP\_motor = 0.1/(0.5\*s+1)

rP\_motor =

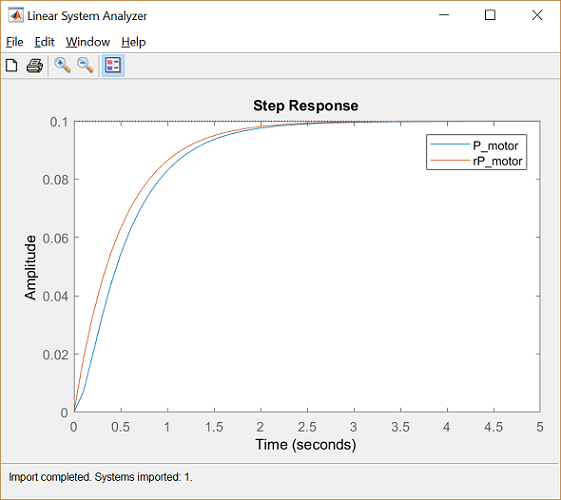
0.1

---------

0.5 s + 1

Continuous-time transfer function.

We can then import this new model into the **Linear System Analyzer**. This is accomplished by selecting **Import** from the **File** menu at the top of the **Linear System Analyzer** window. From the resulting window choose rP\_motor from the **Systems in Workspace** area and then click the **OK** button. The **Linear System Analyzer** will now show plots of both the original and the reduced transfer functions. You can then switch back to step response plots by again choosing **Plot Types** from the right-click menu. You can remove the plot annotations by right-clicking on the plot and using the **Characteristics** submenu. You can also add a legend by clicking the legend icon on the toolbar. Now the **Linear System Analyzer** should appear as shown below.



From the above, we can see that a first-order approximation of our motor system is relatively accurate. The primary difference can be seen at t = 0 where a second order system will have a derivative of zero, but our first-order model will not.

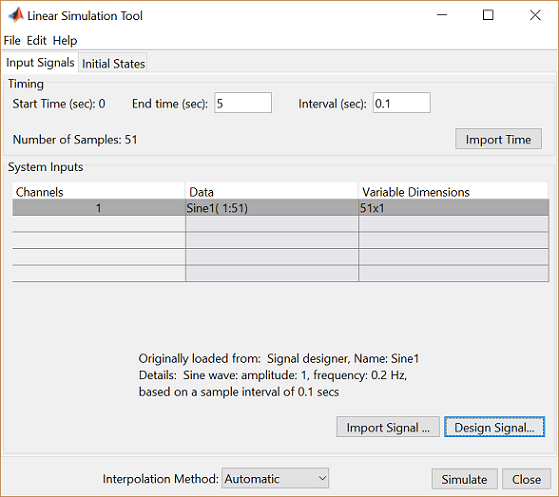
With a first-order system, the settling time is equal to

(4)$$ T_s = 4 \tau $$

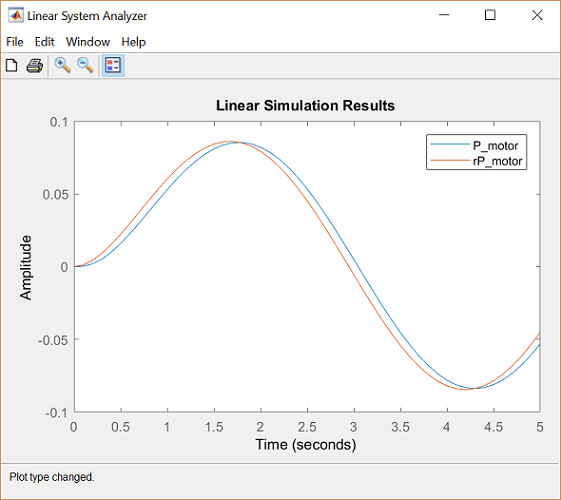
where $\tau$ is the time constant which in this case is 0.5. Therefore, our first-order model has a settling time of 2 seconds which is close to the 2.07 seconds of our actual system. Throughout the rest of the pages of this example, different controllers will be designed to reduce the steady-state error significantly and the settling time slightly while still meeting the given overshoot requirement.

## Response to other types of inputs

While the requirements for this example are given in terms of the system's step response, it is likely that the system will in practice be subject to other types of inputs. Even so, a system's step response can give insight into how the system will respond to other types of signals. In order to determine the system's specific response to other types of inputs, you can employ Simulink or the MATLAB command lsim. Furthermore, you can simulate the system's response to other types of inputs straight from the **Linear System Analyzer**. This is accomplished by right-clicking on the displayed plots and choosing **Plot Types > Linear Simulation**. The following window will then appear.



Within this window set the **End time (sec)** to "5" and the **Interval (sec)** to "0.1". Then under the **System inputs** section of the window, you can import an input signal, or design one from a select set of choices. In this instance, click the **Design signal** button and choose a **Signal type** of Sine wave from within the window that appears. Then change the **Frequency (Hz)** to "0.2" and leave the **Amplitude** and **Duration (secs)** as their default values. Then click the **Insert** button at the bottom of the **Signal Designer** window and the **Simulate** button at the bottom of the **Linear Simulation Tool** window. The responses of our two currently identified systems to the sine wave input are then produced in the **Linear System Analyzer** window. If you double-click on the y-axis of the plot, you can then change the limits to match the figure shown below.



# DC Motor Speed: PID Controller Design

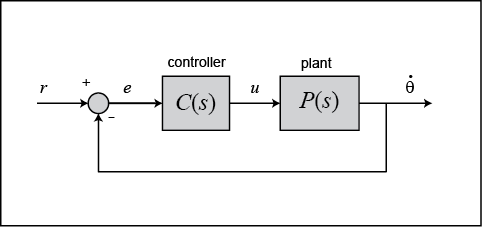
From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

(1)$$ s(Js + b)\Theta(s) = KI(s) $$

(2)$$ (Ls + R)I(s) = V(s) - Ks\Theta(s) $$

(3)$$ P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2}  \qquad [\frac{rad/sec}{V}] $$

The structure of the control system has the form shown in the figure below.



For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page.

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

Now let's design a controller using the methods introduced in the [Introduction: PID Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID) page. Create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and type in the following commands.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

s = tf('s');

P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2);

Recall that the transfer function for a PID controller is:

(4)$$ C(s) = K_{p} + \frac {K_{i}} {s} + K_{d}s = \frac{K_{d}s^2 + K_{p}s + K_{i}} {s} $$

## Proportional control

Let's first try employing a proportional controller with a gain of 100, that is, *C*(*s*) = 100. To determine the closed-loop transfer function, we use the feedback command. Add the following code to the end of your m-file.

Kp = 100;

C = pid(Kp);

sys\_cl = feedback(C\*P\_motor,1);

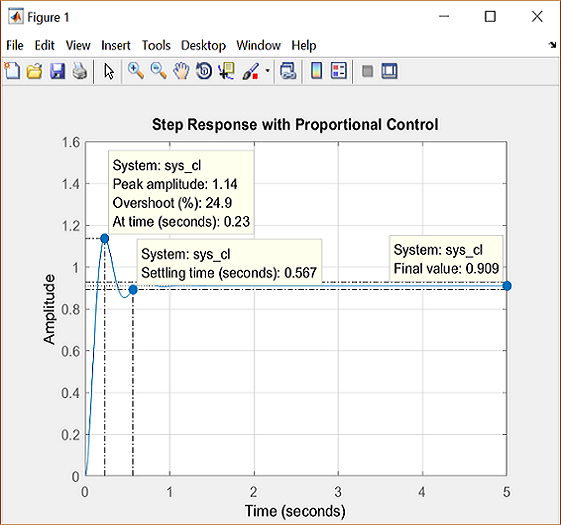
Now let's examine the closed-loop step response. Add the following commands to the end of your m-file and run it in the command window. You should generate the plot shown below. You can view some of the system's characteristics by right-clicking on the figure and choosing **Characteristics** from the resulting menu. In the figure below, annotations have specifically been added for **Settling Time**, **Peak Response**, and **Steady State**.

t = 0:0.01:5;

step(sys\_cl,t)

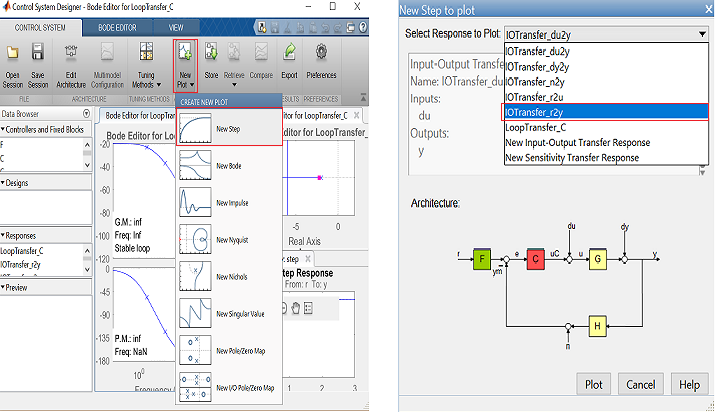
grid

title('Step Response with Proportional Control')

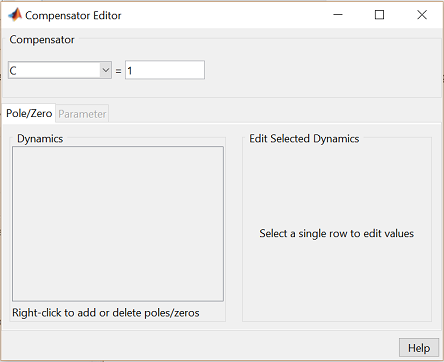


From the plot above we see that both the steady-state error and the overshoot are too large. Recall from the [Introduction: PID Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID) page that increasing the proportional gain $K_p$ will reduce the steady-state error. However, also recall that increasing $K_p$ often results in increased overshoot, therefore, it appears that not all of the design requirements can be met with a simple proportional controller.

This fact can be verified by experimenting with different values of $K_p$. Specifically, you can employ the **Control System Designer** by entering the command controlSystemDesigner(P\_motor) or by going to the **APPS** tab and clicking on the app icon under **Control System Design and Analysis** and then opening a closed-loop step response plot from the **New Plot** tab of the **Control System Designer** window as shown below.



After that you can right-click on the plot and select **Edit Compensator**. You can then vary the control gain in the **Compensator Editor** window and see the resulting effect on the closed-loop step response as shown below.



A little experimentation verifies what we anticipated, a proportional controller is insufficient for meeting the given design requirements; derivative and/or integral terms must be added to the controller.

## PID control

Recall from the [Introduction: PID Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID) page, adding an integral term will eliminate the steady-state error to a step reference and a derivative term will often reduce the overshoot. Let's try a PID controller with small $K_i$ and $K_d$. Modify your m-file so that the lines defining your control are as follows. Running this new m-file gives you the plot shown below.

Kp = 75;

Ki = 1;

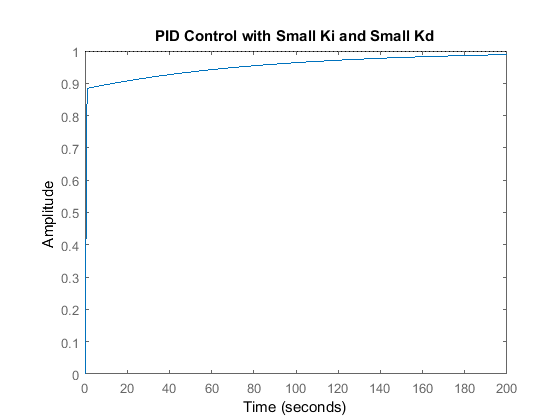
Kd = 1;

C = pid(Kp,Ki,Kd);

sys\_cl = feedback(C\*P\_motor,1);

step(sys\_cl,[0:1:200])

title('PID Control with Small Ki and Small Kd')



Inspection of the above indicates that the steady-state error does indeed go to zero for a step input. However, the time it takes to reach steady-state is far larger than the required settling time of 2 seconds.

## Tuning the gains

In this case, the long tail on the step response graph is due to the fact that the integral gain is small and, therefore, it takes a long time for the integral action to build up and eliminate the steady-state error. This process can be sped up by increasing the value of $K_i$. Go back to your m-file and change $K_i$ to 200 as in the following. Rerun the file and you should get the plot shown below. Again the annotations are added by right-clicking on the figure and choosing **Characteristics** from the resulting menu.

Kp = 100;

Ki = 200;

Kd = 1;

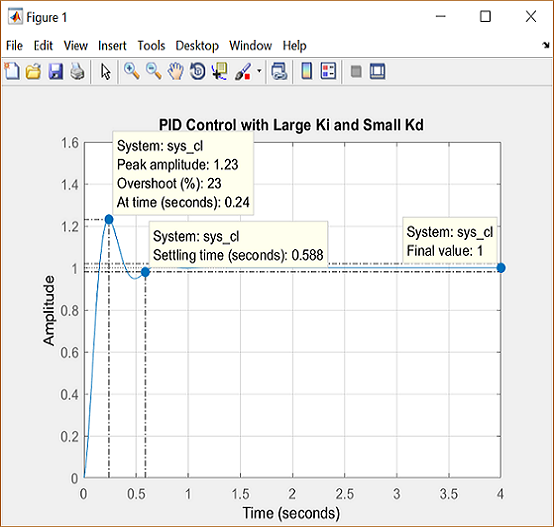
C = pid(Kp,Ki,Kd);

sys\_cl = feedback(C\*P\_motor,1);

step(sys\_cl, 0:0.01:4)

grid

title('PID Control with Large Ki and Small Kd')



As expected, the steady-state error is now eliminated much more quickly than before. However, the large $K_i$ has greatly increased the overshoot. Let's increase $K_d$ in an attempt to reduce the overshoot. Go back to the m-file and change $K_d$ to 10 as shown in the following. Rerun your m-file and the plot shown below should be generated.

Kp = 100;

Ki = 200;

Kd = 10;

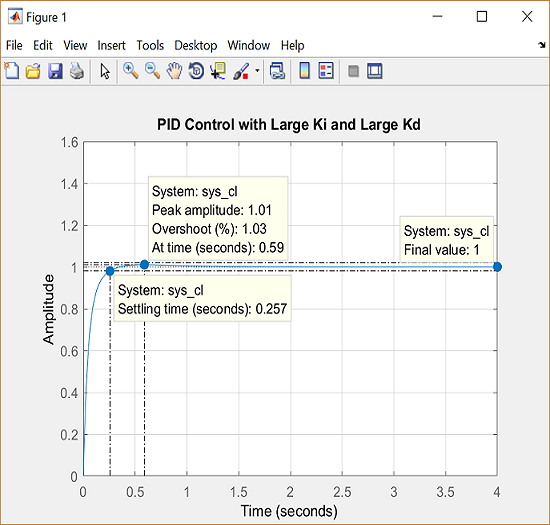
C = pid(Kp,Ki,Kd);

sys\_cl = feedback(C\*P\_motor,1);

step(sys\_cl, 0:0.01:4)

grid

title('PID Control with Large Ki and Large Kd')



As we had hoped, the increased $K_d$ reduced the resulting overshoot. Now we know that if we use a PID controller with $K_p$ = 100, $K_i$ = 200, and $K_d$ = 10, all of our design requirements will be satisfied.

# DC Motor Speed: Root Locus Controller Design

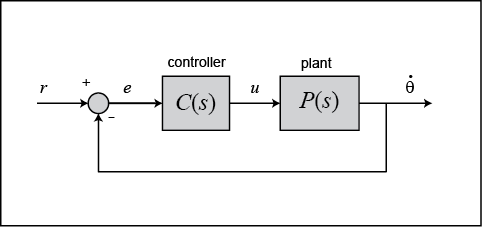
From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

(1)$$ s(Js + b)\Theta(s) = KI(s) $$

(2)$$ (Ls + R)I(s) = V(s) - Ks\Theta(s) $$

(3)$$ P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2}  \qquad [\frac{rad/sec}{V}] $$

The structure of the control system has the form shown in the figure below.



For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page.

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

Now let's design a controller using the methods introduced in the [Introduction: Root Locus Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlRootLocus) page. Create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and type in the following commands.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

s = tf('s');

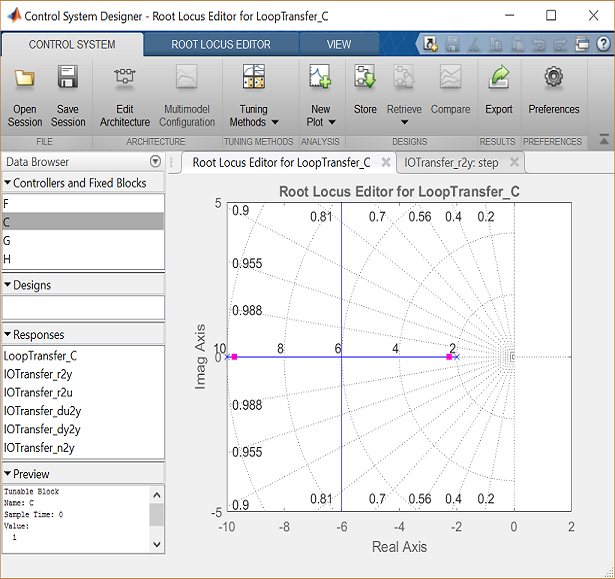
P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2);

## Drawing the open-loop root locus

The main idea of root locus design is to predict the closed-loop response from the root locus plot which depicts possible closed-loop pole locations and is drawn from the open-loop transfer function. Then by adding zeros and/or poles via the controller, the root locus can be modified in order to achieve a desired closed-loop response.

We will use for our design the **Control System Designer** graphical user interface. This tool allows you to graphically tune the controller via the root locus plot. Let's first view the root locus for the uncompenstated plant. This is accomplished by adding the command controlSystemDesigner('rlocus', P\_motor) to the end of your m-file and running the file at the command line or by going to the **APPS** tab of the MATLAB toolstrip and clicking on the app icon under **Control System Analysis and Design**.

One window titled **Control System Designer** will open initially having the form shown in the figure below. In the window, you will be able to see both the root locus plot and the closed-loop step response of the transfer function passed via the controlSystemDesigner function. If the string 'rlocus' is omitted from the function call, the default initial window includes the Bode plot, in addition to the root locus plot and closed-loop step response plot. You can arrange the position of plots from the **VIEW** tab of the **Control System Designer** window. Right-clicking on the root locus plot and selecting **Grid** will make your window appear as follows.

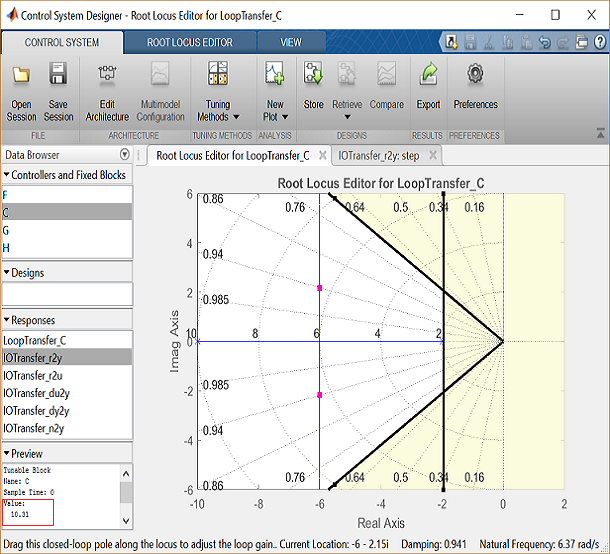


## Finding the loop gain

Recall that our design requirements specify that the settling time be less than 2 seconds and that the overshoot be less than 5%. The location of the system's closed-loop poles provide information regarding the system's transient response. The **Control System Designer** allows you to specify the region in the complex *s*-plane corresponding to specific design requirements. The provided regions correspond to a canonical second-order system, but in general are a good place to start from even for higher-order systems or systems with zeros.

These desired regions can be added to the root locus plot by right-clicking on the plot and choosing **Design Requirements > New** from the resulting menu. You can add many design requirements including Settling time, Percent overshoot, Damping ratio, Natural frequency, and generic Region constraint.

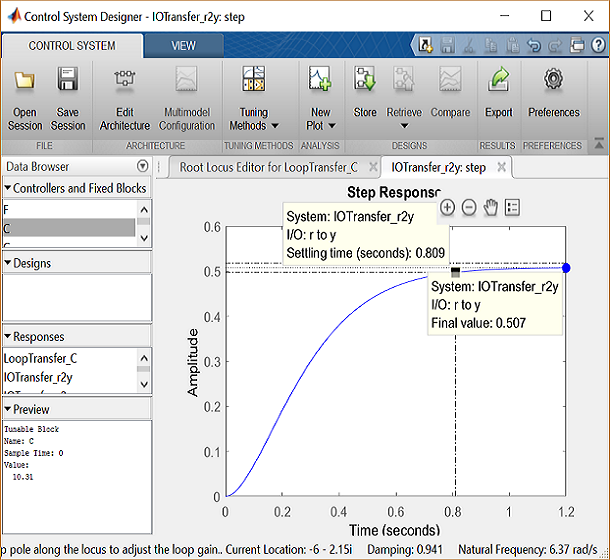
Adding our settling time and percent overshoot requirements to the root locus plot produces the following figure.



The resulting desired region for the closed-loop poles is shown by the unshaded region of the above figure. More specifically, the two rays centered at the origin represent the overshoot requirement; the smaller the angle these rays make with the negative real-axis, the less overshoot is allowed. The vertical line at *s* = -2 represents the settling time requirement, where the farther to left the closed-loop poles are located the smaller the settling time is. From examination of the above figure, there are values of the loop gain that will place both closed-loop poles in the desired region. This can be seen from the fact that the two branches of the root locus are symmetric and pass through the unshaded region. Furthermore, since the closed-loop system has two poles with no zeros, placing the closed-loop poles in the shown region will guarantee satisfaction of our transient response requirements.

You can select a specific pair of closed-loop poles from the resulting figure in order to determine the corresponding loop gain that places the poles at that location. For our system, let's choose to place the closed-loop poles so that they are located on the vertical branches of the root-locus between the real axis and the boundary of the overshoot requirement. The pink boxes on the root locus indicate the location of the closed-loop poles for the current loop gain. Clicking on the pink boxes and dragging them along the root locus to the desired location automatically modifies the controller to place the closed-loop poles at the indicated position. Let us drag a closed-loop pole to a location near -6 + 2i. The pole location will be indicated at the bottom of the window along with the corresponding damping ratio and natural frequency. We can also check the corresponding loop gain in the lower left corner by clicking on **C** in the **Controllers and Fixed Blocks** tab. The loop gain, as we can see in the figure, is approximately 10.

We can check the closed-loop step response for the system with this new gain by moving to the **IOTransfer\_r2y: step** tab. If you have accidentally closed this tab, you can re-open it from the **Control System Designer** window by clicking on the **New Plot** menu and selecting **New Step**. In response, a new window titled **New Step to plot** will appear. From the **Select Responses to Plot** menu, then choose **IOTransfer\_r2y** and click the button **Plot**. The response of the output *y* of the closed-loop system for a step reference *r* will then appear in the **Control System Designer** window. You can also identify some characteristics of the step response. Specifically, right-click on the figure and under **Characteristics** choose **Settling Time**. Then repeat for **Steady State**. Your figure will appear as shown below.



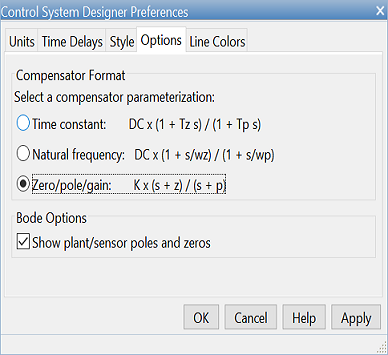
From inspection of the above, one can see that there is no overshoot and the settling time is less than one second, therefore, the overshoot and settling time requirements are satisfied. However, we can also observe that the steady-state error is approximately 50%. If we increase the loop gain to reduce the steady-state error, the overshoot will become too large. You can see this for yourself by graphically moving the closed-loop poles vertically upward along the root locus, this corresponds to increasing the loop gain. The step response plot will change automatically to reflect the modified loop gain. We will attempt to add a lag controller to reduce the steady-state error requirement while still satisfying the transient requirements.

## Adding a lag controller

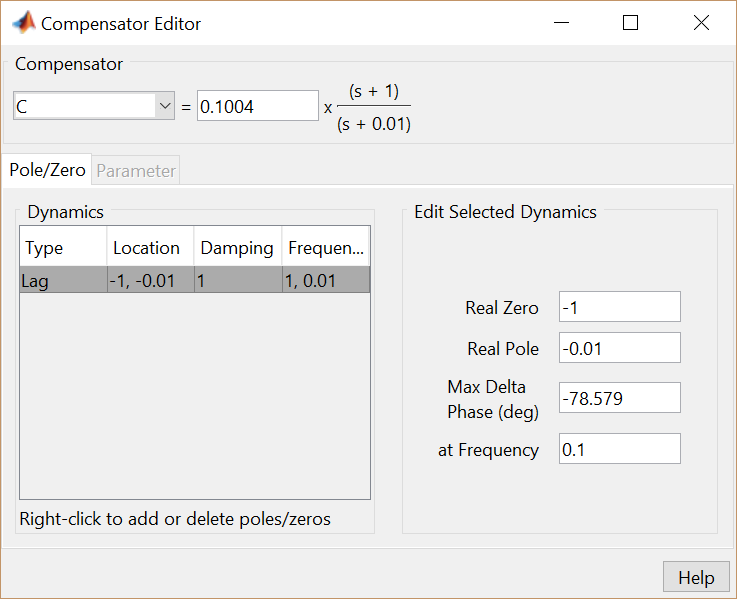
In the above we saw that the overshoot and settling time criteria were met with the proportional controller, but the steady-state error requirement was not. A **lag compensator** is one type of controller known to be able to reduce steady-state error. However, we must be careful in our design to not increase the settling time too much. Let's first try adding a lag compensator of the form given below.

(4)$$ C(s) = \frac {(s + 1)} { (s + 0.01) } $$

We can use the **Control System Designer** to design our lag compensator. To make the **Control System Designer** have a compensator parameterization corresponding to the one shown above, click on the **Preferences** menu at the top of the **Control System Designer** window. Then From the **Options** tab, select a **Zero/pole/gain** parameterization as shown below.



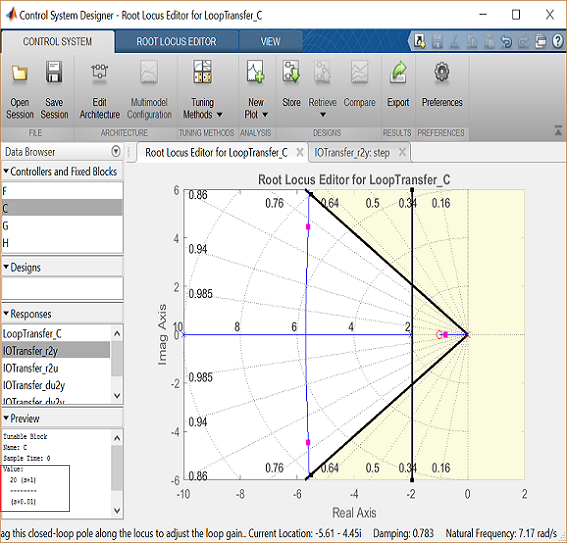
To add the lag compensator, right click on the root locus plot and select **Edit Compensator**. To add a pole zero pair to your compensator, in the **Compensator Editor** window, right-click the **Dynamics** table and select **Add Pole/Zero > lag**. After that, enter the **Real Zero** and **Real Pole** locations as shown in the following figure.



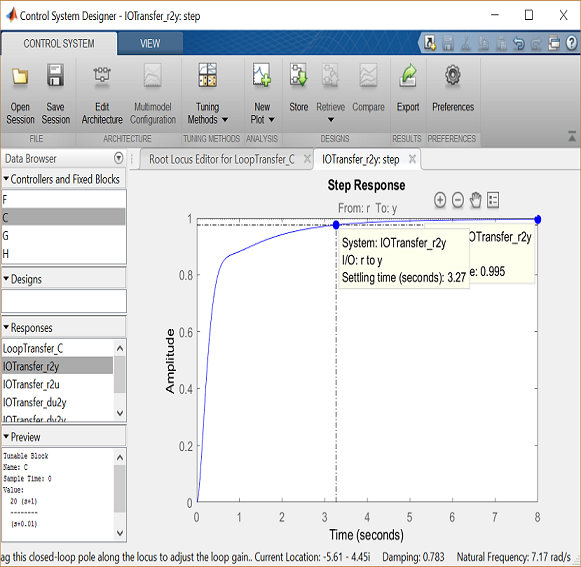
Note that the maximum phase lag contributed by the compensator and the frequency where it is located are updated to match the pole and zero locations chosen.

## Finding the loop gain with a lag controller

Notice how the root locus has changed to reflect the addition of the pole and zero from the lag compensator as shown in the figure below. We can again choose closed-loop pole locations to attempt to achieve our desired transient requirements. Let's attempt to place two of the closed-loop poles in our desired region near the boundary of the overshoot requirement. For example, a loop gain of approximately 20 (set in the **Compensator Editor**) will place the poles at the positions shown in the figure below.

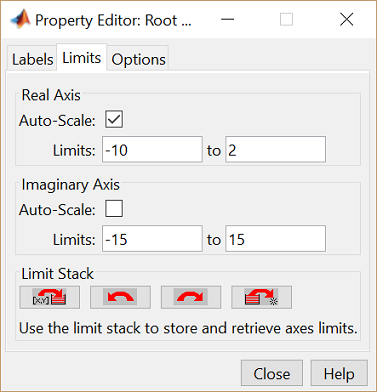


The corresponding closed-loop step response will then update automatically to match the figure shown below.

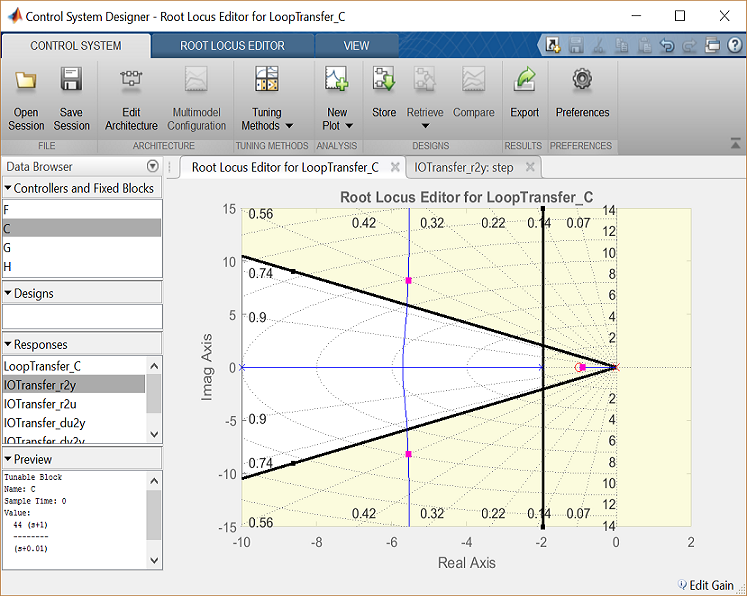


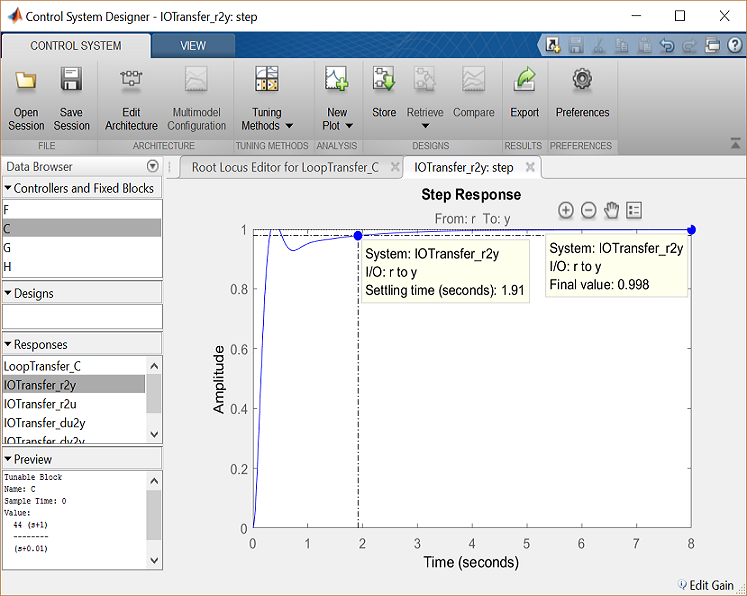
As you can see, the response is not quite satisfactory even though two of the closed-loop poles were placed in the desired region. The reason for this is because the closed-loop system no longer has the form of a canonical second-order system. Specifically, there is a third pole on the real axis indicated in the root locus plot above that is outside of the desired region. The fact that this third pole is to the right of the two conjugate poles placed above means that it will slow the system response down, that is why the settling time requirement is no longer met. Additionally, the overshoot requirement is met easily even though the two conjugate poles are near the edge of the allowed region. This is due again to the third pole which is well damped and tends to dominate the response because it is "slower" than the other poles. What this means is that we can further increase the loop gain such that the conjugate poles move beyond the diagonal lines while still meeting the overshoot requirement.

You can now return to the root locus plot and graphically move the conjugate poles farther away from the real axis; this corresponds to increasing the loop gain. As you move the closed-loop poles a sufficient distance, the limits of the plot should update automatically. Alternatively, you can change the limits manually by right-clicking on the root locus and selecting **Properties** to open the **Property Editor**. Then you can click on the **Limits** tab and change the imaginary axis limits to [-15,15], for example, as shown below.



Experiment with different gains (closed-loop pole locations) until you achieve the desired response. Below is the root locus with a loop gain of 44 and the corresponding closed-loop step response.





Now the settling time is less than 2 seconds and the steady-state error and overshoot requirements are still met. As you can see, the root locus design process requires some trial and error. The **Control System Designer** is very helpful in this process. Using the **Control System Designer**, it is very easy to tune your controller and immediately see the effect on the root locus and various analysis plots, like the closed-loop step response. If we had not been able to get a satisfactory response by tuning the loop gain, we could have tried moving the pole and zero of the lag compensator or we could have tried a different type of dynamic compensator (additional poles and/or zeros).

# DC Motor Speed: Frequency Domain Methods for Controller Design

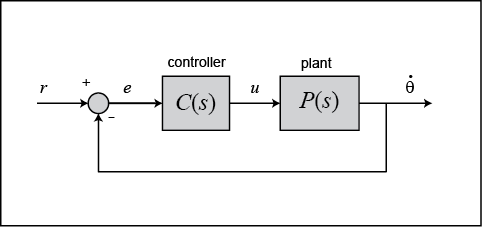
From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

(1)$$ s(Js + b)\Theta(s) = KI(s) $$

(2)$$ (Ls + R)I(s) = V(s) - Ks\Theta(s) $$

(3)$$ P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2}  \qquad [\frac{rad/sec}{V}] $$

The structure of the control system has the form shown in the figure below.



For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

Now let's design a controller using the methods introduced in the [Introduction: Frequency Domain Methods for Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlFrequency) page. Create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and type in the following commands.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

s = tf('s');

P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2);

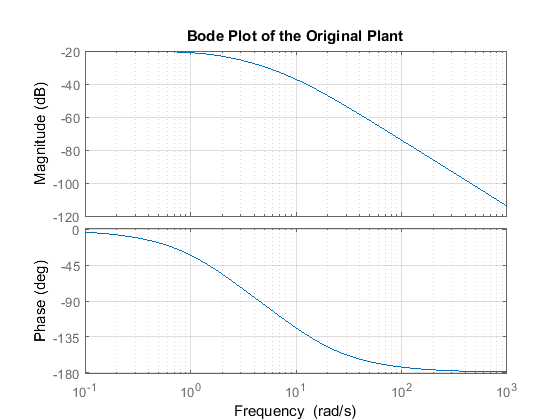
## Drawing the original Bode plot

The main idea of frequency-based design is to use the Bode plot of the open-loop transfer function to estimate the closed-loop response. Adding a controller to the system changes the open-loop Bode plot, thereby changing the closed-loop response. It is our goal to design the controller to shape the open-loop Bode plot in such a way that the closed-loop system behaves in a desired manner. Let's first draw the Bode plot for the original open-loop plant transfer function. Add the following code to the end of your m-file and run it in the MATLAB command window. You should generate the Bode plot shown below.

bode(P\_motor)

grid

title('Bode Plot of the Original Plant')



## Adding proportional gain

From the Bode plot above, it appears that the gain margin and phase margin of this system are currently infinite which indicates the system is robust and has minimal overshoot. The problem with this is that the phase margin is infinite because the magnitude plot is below 0 dB at all frequencies. This indicates that the system will have trouble tracking various reference signals without excessive error. Therefore, we would like to increase the gain of the system while still achieving enough phase margin.

A phase margin of 60 degrees is generally sufficient for stability margin. From the above Bode plot, this phase margin is achieved for a crossover frequency of approximately 10 rad/sec. The gain needed to raise the magnitude plot so that the gain crossover frequency occurs at 10 rad/sec appears to be approximately 40 dB. The exact phase and gain of the Bode plot at a given frequency can be determined by clicking on the graph at the corresponding frequency. The bode command, invoked with left-hand arguments, can also be used to provide the exact phase and magnitude at 10 rad/sec as shown below.

[mag,phase,w] = bode(P\_motor,10)

mag =

0.0139

phase =

-123.6835

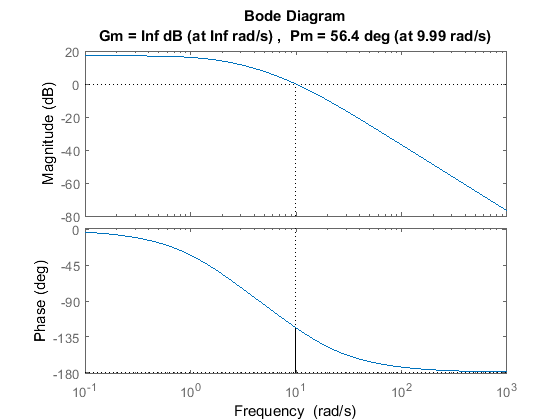
w =

10

Therefore, the exact phase margin for a gain crossover frequency of 10 rad/sec is 180 - 123.7 = 56.3 degrees. Since the exact magnitude at this frequency is 20 log 0.0139 = -37.1 dB, 37.1 dB of gain must be added to the system. Otherwise stated, a proportional gain of 1/0.0139 = 72 will achieve an open-loop gain of 1 at 10 rad/sec. Add the following commands to your m-file to observe the effect of this proportional controller on the system. In this case, we use the margin command instead of the bode command in order to explicitly see the new gain and phase margins and crossover frequencies.

C = 72;

margin(C\*P\_motor);



## Plotting the closed-loop response

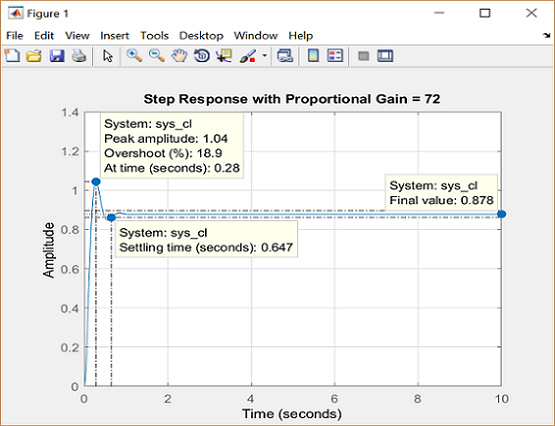
From the plot above we see that the resulting phase margin and gain crossover frequency are as we expected. Let's see what the closed-loop response look like. Add a % in front of the bode and margin commands to comment them out, then add the following code to the end of your m-file. Rerunning the m-file will produce the step response shown below where the annotations were added by right-clicking on the plot and choosing **Characteristics** from the resulting menu.

sys\_cl = feedback(C\*P\_motor,1);

t = 0:0.01:10;

step(sys\_cl,t), grid

title('Step Response with Proportional Gain = 72')



Note that the settling time is fast enough, but the overshoot and the steady-state error are too high. The overshoot can be reduced by decreasing the gain in order to achieve a larger phase margin, but this would cause the steady-state error to become even larger. A lag compensator could be helpful here in that it can decrease the gain crossover frequency in order to increase the phase margin without decreasing the system's DC gain.

## Adding a lag compensator

Consider the following lag compensator:

(4)$$ C(s) = \frac {(s + 1)} { (s + 0.01) } $$

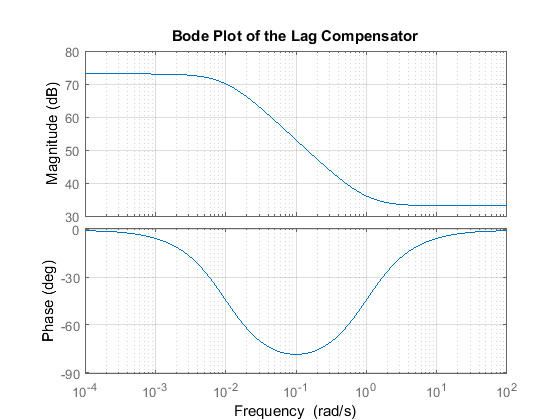
This lag compensator has a DC gain of 1/0.01 = 100 which means it will increase the system's static position error constant by a factor of 100 and will reduce the steady-state error associated with the system's closed-loop step response. In fact, it allows us to reduce the proportional gain of 72 used earlier, while still meeting the requirement on steady-state error. We will employ a gain of 45. Furthermore, since the corner frequencies of the pole and zero are a decade or more below the current gain crossover frequency of 10 rad/sec, the phase lag contributed by the compensator shouldn't adversely affect performance much. A Bode plot of the lag compensator can be generated employing the following commands.

C = 45\*(s + 1)/(s + 0.01);

bode(C)

grid

title('Bode Plot of the Lag Compensator')



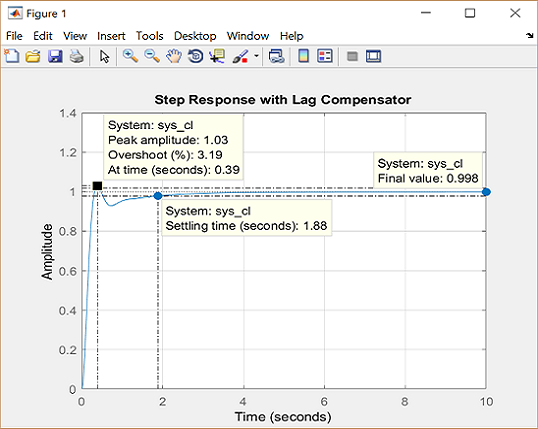
The resulting step response can then be observed by modifying the code in your m-file as follows.

sys\_cl = feedback(C\*P\_motor,1);

t = 0:0.01:10;

step(sys\_cl,t), grid

title('Step Response with Lag Compensator')



Inspection of the above demonstrates that all of the given requirements are now met when the lag compensator described above is employed.

# DC Motor Speed: State-Space Methods for Controller Design

From the main problem, the dynamic equations in state-space form are given below.

(1)$$\frac{d}{dt}\left [\begin{array}{c} \dot{\theta} \\ \ \\ i \end{array} \right] =
\left [\begin{array}{cc} -\frac{b}{J} & \frac{K}{J} \\ \ \\ -\frac{K}{L} &
-\frac{R}{L} \end{array} \right] \left [\begin{array}{c} \dot{\theta} \\ \ \\ i \end{array} \right]  +
\left [\begin{array}{c} 0 \\ \ \\ \frac{1}{L} \end{array} \right] V$$

(2)$$ y = [ \begin{array}{cc}1 & 0\end{array}] \left [ \begin{array}{c} \dot{\theta} \\ \ \\ i
\end{array} \right] $$

For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page. These state-space equations have the standard form shown below where the state vector ${\bf x} = [ \dot{\theta} \ i ]^{T}$ and the input $u = V$.

(3)$$ \dot{{\bf x}} = A{\bf x} + Bu $$

(4)$$ y = C\bf{x} $$

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-stage error less than 1%

Now let's design a controller using the methods introduced in the [Introduction: State-Space Methods for Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlStateSpace) page. Create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and type in the following commands.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

A = [-b/J K/J

-K/L -R/L];

B = [0

1/L];

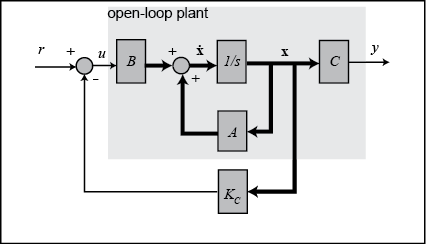
C = [1 0];

D = 0;

sys = ss(A,B,C,D);

## Designing the full-state feedback controller

Since both of the state variables in our problem are easy to measure (simply add an ammeter for current and a tachometer for the speed), we can design a full-state feedback controller for the system without worrying about having to add an observer. The control law for a full-state feedback system has the form $u = r - K_c{\bf x}$ and the associated schematic is shown below.



Recall that the characteristic polynomial for this closed-loop system is the determinant of $sI - (A - BK_c)$ where $s$ is the Laplace variable. Since the matrices $A$ and $BK_c$ are both 2x2 matrices, there should be 2 poles for the system. This fact can be verified with the MATLAB command order. If the given system is controllable, then by designing a full-state feedback controller we can move these two poles anywhere we'd like. Whether the given system is controllable or not can be determined by checking the rank of the controllability matrix $[ B\ AB\ A^2B\ \dots]$. The MATLAB command ctrb constructs the controllability matrix given matrices $A$ and $B$. Additionally, the command rank determines the rank of a given matrix. The following commands executed at the command line will verify the system's order and whether or not the system is controllable.

sys\_order = order(sys)

sys\_rank = rank(ctrb(A,B))

sys\_order =

2

sys\_rank =

2

From the above, we know that our system is controllable since the controllability matrix is full rank. We will first place the poles at -5+i and -5-i (note that this corresponds to a $\zeta$ = 0.98 which gives close to 0% overshoot and a $\sigma$ = 5 which provides a 0.8 second settling time). Once we have determined the pole locations we desire, we can use the MATLAB commands place or acker to determine the controller gain matrix, $K_c$, to achieve these poles. We will use the command place since it is numerically better conditioned than acker. However, if we wished to place a pole with multiplicity greater than the rank of the matrix $B$, then we would have to use the command acker. Add the following code to the end of your m-file. Running in the command window will generate the feedback gain matrix output below.

p1 = -5 + 1i;

p2 = -5 - 1i;

Kc = place(A,B,[p1 p2])

Kc =

12.9900 -1.0000

Referring back to the state-space equations at the top of the page, we see that substituting the state-feedback law $u = r - K_c{\bf x}$ for $u$ leads to the following expression.

(5)$$ \dot{{\bf x}} = (A - BK_c){\bf x} + Br $$

(6)$$ y = C\bf{x} $$

We can then see the closed-loop response by simply adding the following lines to the end of your m-file. Running your m-file in the command window will then give the plot shown below.

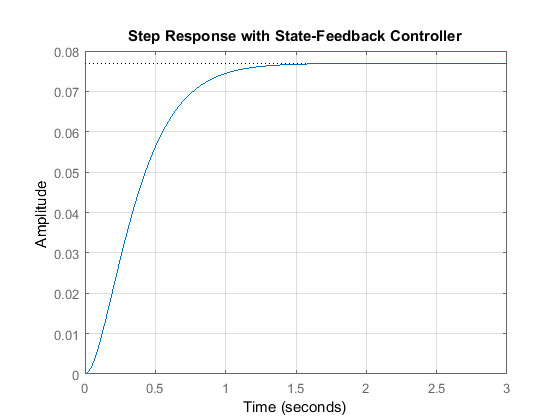
t = 0:0.01:3;

sys\_cl = ss(A-B\*Kc,B,C,D);

step(sys\_cl,t)

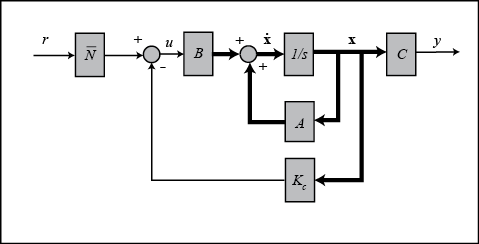
grid

title('Step Response with State-Feedback Controller')



## Adding a precompensator

From this plot we see that the steady-state error is too large. One approach for eliminating the steady-state error is to simply scale the input so that the output in turn is scaled to the desired level. This is a little challenging in our example because we have two states to consider. Therefore, we need to compute what the steady-state values of both states should be, multiply them by the chosen gain $K_c$, and use the result as our "reference" for computing the input $u$. This can be done in one step by adding a constant gain precompensator $\overline{N}$ after the reference as shown in the following schematic.



We can find this $\overline{N}$ factor by employing the user-defined function [rscale.m](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_rscale) as shown below.

Nbar = rscale(sys,Kc)

Nbar =

13.0000

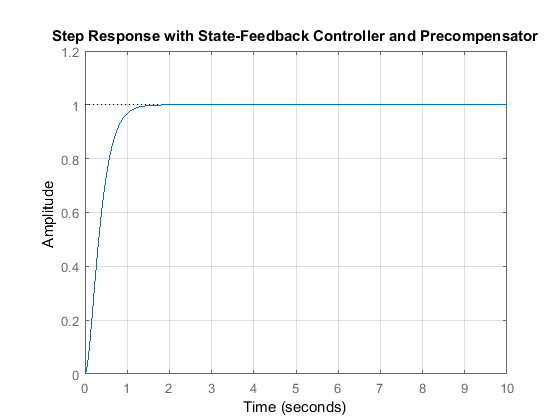
Note that the function rscale.m is not a standard function in MATLAB. You will have to download it and place it in your current directory. Click [here](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Function) for further information. Now you can plot the step response by adding the above and following lines of code to your m-file and re-running at the command line.

t = 0:0.01:10;

step(sys\_cl\*Nbar,t)

grid

title('Step Response with State-Feedback Controller and Precompensator')



This time, the steady-state error is much less than 1%, and all the other design criteria have been met as well.

Note that the precompensator $\overline{N}$ employed above is calculated based on the model of the plant and further that the precompensator is located outside of the feedback loop. Therefore, if there are errors in the model (or unknown disturbances) the precompensator will not correct for them and there will be steady-state error. You may recall that the addition of integral control may also be used to eliminate steady-state error, even in the presence of model uncertainty and step disturbances. For an example of how to implement integral control in the state space setting, see the [DC Motor Position: State-Space Methods for Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorPosition&section=ControlStateSpace) example. The tradeoff with using integral control is that the error must first develop before it can be corrected for, therefore, the system may be slow to respond. The precompensator on the other hand is able to anticipitate the steady-state offset using knowledge of the plant model. A useful technique is to combine the precompensator with integral control to leverage the advantages of each approach.

# DC Motor Speed: Digital Controller Design

In this page, we will consider the digital version of the DC motor speed control problem. A sampled-data DC motor model can be obtained from conversion of the analog model, as we will describe. In this example we will design a PID controller.

The continuous open-loop transfer function for an input of armature voltage and an output of angular speed was derived previously as the following.

(1)$$ P(s) = \frac{\dot{\Theta}(s)}{V(s)} = \frac{K}{(Js + b)(Ls + R) + K^2}
\qquad [\frac{rad/sec}{V}] $$

For the original problem setup and the derivation of the above equations, please refer to the [DC Motor Speed: System Modeling](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling) page.

For a 1-rad/sec step reference, the design criteria are the following.

* Settling time less than 2 seconds
* Overshoot less than 5%
* Steady-state error less than 1%

## Creating a sampled-data model of the plant

The first step in the design of a digital control system is to generate a sampled-data model of the plant. Therefore, it is necessary to choose a frequency with which the continuous-time plant is sampled. In choosing a sampling period, it is desired that the sampling frequency be fast compared to the dynamics of the system in order that the sampled output of the system captures the system's full behavior, that is, so that significant inter-sample behavior isn't missed.

Let's create a continuous-time model of the plant. Create a new [m-file](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Mfile) and add the following MATLAB code (refer to the main problem for the details of getting these commands). Running the m-file within the MATLAB command window will generate the output shown below.

J = 0.01;

b = 0.1;

K = 0.01;

R = 1;

L = 0.5;

s = tf('s');

P\_motor = K/((J\*s+b)\*(L\*s+R)+K^2);

zpk(P\_motor)

ans =

2

-------------------

(s+9.997) (s+2.003)

Continuous-time zero/pole/gain model.

The use of the zpk command above transforms the transfer function into a form where the zeros, poles, and gain can be seen explicitly. Examining the poles of the plant (or its frequency response), the dominant pole of the plant ($\sigma$ approximately equal to 2) corresponds to a settle time of approximately 2 seconds (4 / $\sigma$). Therefore, choosing a sampling period of 0.05 seconds is significantly faster than the dynamics of the plant. This sampling period is also fast compared to the speed that will be achieved by the resultant closed-loop system.

In this case, we will convert the given transfer function from the continuous Laplace-domain to the discrete z-domain. MATLAB can be used to achieve this conversion through the use of the c2d command. The c2d command requires three arguments: a system model, the sampling time ($T_s$), and the type of hold circuit. In this example, we will assume a zero-order hold (zoh) circuit. Refer to the [Introduction: Digital Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlDigital) page for further details. Adding the following commands to your m-file and running in the MATLAB command window generates the sampled-data model shown below.

Ts = 0.05;

dP\_motor = c2d(P\_motor,Ts,'zoh');

zpk(dP\_motor)

ans =

0.0020586 (z+0.8189)

---------------------

(z-0.9047) (z-0.6066)

Sample time: 0.05 seconds

Discrete-time zero/pole/gain model.

We would now like to analyze the closed-loop response of the system without any additional compensation. First, we have to close the loop of the transfer function by using the feedback command. After closing the loop, let's inspect the closed-loop step response with the zero-order hold. This can be accomplished by using the step and stairs commands. Since the step command is fed a discrete model, it will output a vector of discrete samples at the sample time $T_s$ associated with the model (click [here](https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_step) for further details). The syntax below specifies to simulate the step response for 0.5 seconds. The stairs command draws these discrete data points as a stairstep, just like what would be produced by a zero-order hold circuit. Add the following MATLAB code at the end of your previous m-file and rerun it. You should generate the plot shown below.

sys\_cl = feedback(dP\_motor,1);

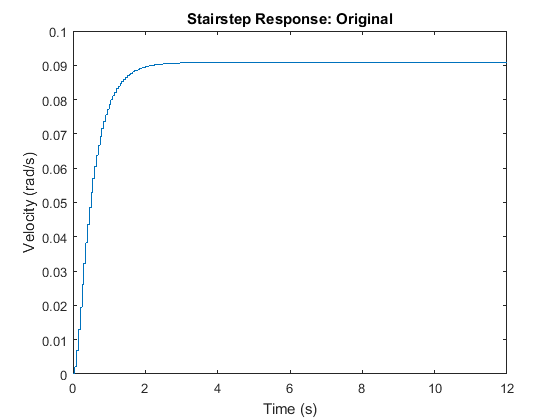
[y,t] = step(sys\_cl,12);

stairs(t,y);

xlabel('Time (s)')

ylabel('Velocity (rad/s)')

title('Stairstep Response: Original')



Examination of the above shows that the steady-state error is much too large and the settle time may be a little slow.

## PID Controller

Recall that the continuous-time transfer function for a PID controller is:

(2)$$ C(s) = K_{p} + \frac {K_i}{s} + K_{d}s $$

There are several ways for mapping from the *s*-plane to *z*-plane. Above we used a zero-order hold conversion for the plant model because that reflected the type of hold circuit that would be used in sampling the signals from the plant in a physical implementation of the control system. For the controller, we may prefer to use a conversion that more accurately approximates the behavior that would be achieved with a continuous, rather than digital, controller. The exact conversion between the Laplace- and *z*-domains is given below.

(3)$$ z = e^{sT_s} $$

This conversion, however, involves a trancendental function and the resulting transfer function cannot be represented as a ratio of polynomials. This makes it difficult to implement such a control algorithm on a digital computer. Therefore, we will use an approximate conversion. In particular, we are going to use the bilinear transformation shown below.

(4)$$ s = \frac{2}{T_s}.\frac{z-1}{z+1} $$

Equivalently, we will again use the c2d command in MATLAB to convert the continuous-time PID compensator to a discrete-time PID compensator by specifying 'tustin' as the conversion method. Tustin's method uses the bilinear transformation to convert a continuous model to discrete time. According to the [DC Motor Speed: PID Controller Design](https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=ControlPID) page , $K_p$ = 100, $K_i$ = 200 and $K_d$ = 10 were found to satisfy all of the given design requirements. We will use these gains again for this example. Now add the following MATLAB commands to your previous m-file and rerun it in the MATLAB command window.

Kp = 100;

Ki = 200;

Kd = 10;

C = Kp + Ki/s + Kd\*s;

dC = c2d(C,Ts,'tustin')

dC =

505 z^2 - 790 z + 305

---------------------

z^2 - 1

Sample time: 0.05 seconds

Discrete-time transfer function.

Let's see if the performance of the closed-loop response with PID compensator satisfies the given design requirements. Add the following code to the end of your m-file and rerun it. You should get the following closed-loop stairstep response.

sys\_cl = feedback(dC\*dP\_motor,1);

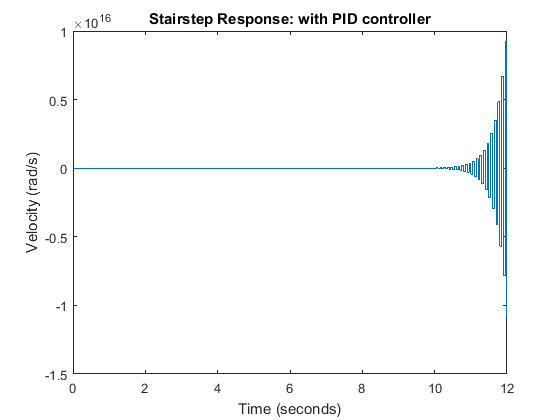
[x2,t] = step(sys\_cl,12);

stairs(t,x2)

xlabel('Time (seconds)')

ylabel('Velocity (rad/s)')

title('Stairstep Response: with PID controller')

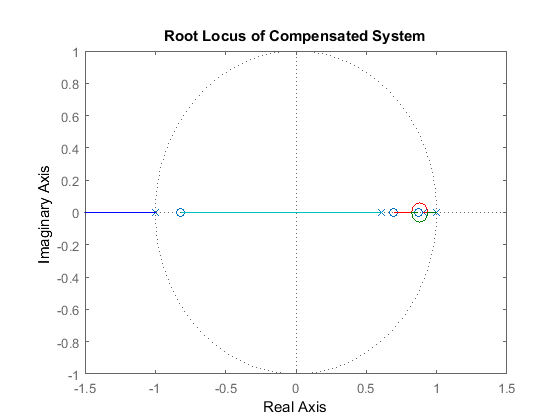


As you can see from the above plot, the closed-loop response of the system is unstable. Therefore, something must be wrong with the compensated system. We should take a look at the root locus of the compensated system. Add the following MATLAB commands onto the end of your m-file and rerun it.

rlocus(dC\*dP\_motor)

axis([-1.5 1.5 -1 1])

title('Root Locus of Compensated System')



From this root-locus plot, we see that the denominator of the PID controller has a pole at -1 in the *z*-plane. We know that if a pole of a system is outside the unit circle, the system will be unstable. This compensated system will always be unstable for any positive gain because there are an even number of poles and zeros to the right of the pole at -1. Therefore, a closed-loop pole will always move to the left and outside the unit circle as the loop gain is increased. The pole at -1 comes from the compensator, hence we can change its location by changing the compensator design. In this case we choose to cancel the zero at -0.82, this will make the system stable for at least some gains. Add the following code to your m-file and rerun it at the command line to generate the root locus plot shown below.

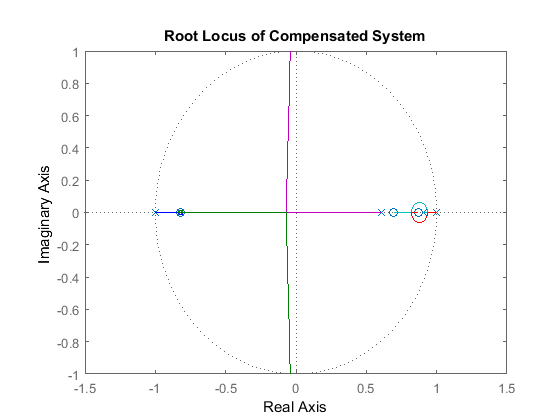
z = tf('z',Ts);

dC = dC/(z+0.82);

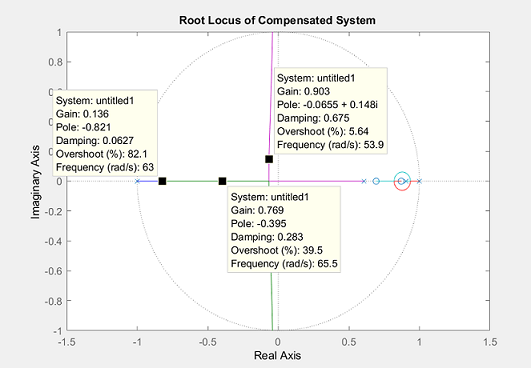
rlocus(dC\*dP\_motor);

axis([-1.5 1.5 -1 1])

title('Root Locus of Compensated System');



The new closed-loop system will have a pole near -0.82 instead of -1, which almost cancels the zero of the uncompensated plant. Click on the root locus curve in several places to see the gain that places a closed-loop pole at a particular location, along with the corresponding overshoot, damping, etc., as shown in the figure below. Of course, the damping and overshoot numbers correspond to the actual response only if the selected closed-loop pole is dominant.



You can also determine the gain corresponding to a specfic closed-loop pole location using the rlocfind command. Typing [K,poles] = rlocfind(dC\*dP\_motor) at the command line will generate a cursor that you can then use to click on the point of interest on the root locus. MATLAB will then return the appropriate gain K and all of the corresponding closed-loop poles. This is useful in that it lists all of the closed-loop poles, not just the point you clicked on.

We will choose a gain of 0.8 and examine the resulting closed-loop step response by typing the following commands at the MATLAB command window.

sys\_cl = feedback(0.8\*dC\*dP\_motor,1);

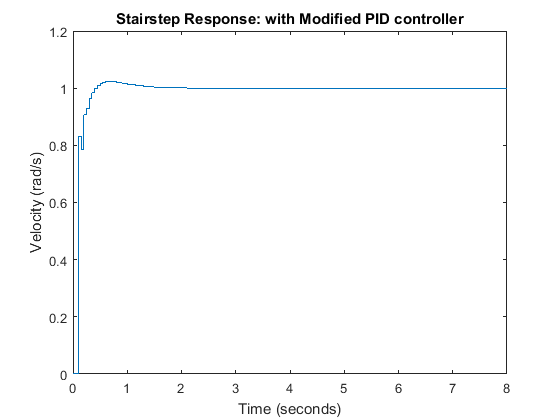
[x3,t] = step(sys\_cl,8);

stairs(t,x3)

xlabel('Time (seconds)')

ylabel('Velocity (rad/s)')

title('Stairstep Response: with Modified PID controller')



The plot above shows that the settling time is less than 2 seconds and the percent overshoot is around 2%. Additionally, the steady-state error is zero. Therefore, this response satisfies all of the given design requirements.