# PID Controller Design

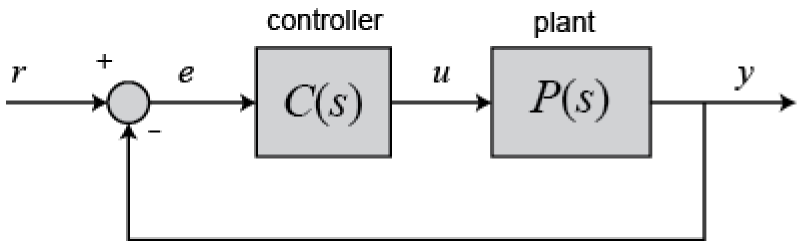
In this tutorial we will introduce a simple yet versatile feedback compensator structure, the Proportional-Integral-Derivative (PID) controller. We will discuss the effect of each of the pid parameters on the closed-loop dynamics and demonstrate how to use a PID controller to improve the system performance.

Key MATLAB commands used in this tutorial are: `tf` , `step` , `pid` , `feedback` , `pidtool` , `pidtune`

## Contents

## PID Overview

In this tutorial, we will consider the following unity feedback system:



The output of a PID controller, equal to the control input to the plant, in the time-domain is as follows:

(1) $$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt}$$

First, let's take a look at how the PID controller works in a closed-loop system using the schematic shown above. The variable ($e$) represents the tracking error, the difference between the desired input value ($r$) and the actual output ($y$). This error signal ($e$) will be sent to the PID controller, and the controller computes both the derivative and the integral of this error signal.

The control signal ($u$) to the plant is equal to the proportional gain ($K_p$) times the magnitude of the error plus the integral gain ($K_i$) times the integral of the error plus the derivative gain ($K_d$) times the derivative of the error.

This control signal ($u$) is sent to the plant, and the new output ($y$) is obtained. The new output ($y$) is then fed back and compared to the reference to find the new error signal ($e$). The controller takes this new error signal and computes its derivative and its integral again, ad infinitum.

The transfer function of a PID controller is found by taking the Laplace transform of Eq.(1).

(2) $$K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s}$$

$K_p$= Proportional gain $K_i$= Integral gain $K_d$= Derivative gain

We can define a PID controller in MATLAB using the transfer function directly, for example:

```
Kp = 1;
Ki = 1;
Kd = 1;

s = tf('s');
C = Kp + Ki/s + Kd*s
C =

  s^2 + s + 1
  -----------
       s

Continuous-time transfer function.
```

Alternatively, we may use MATLAB's **pid controller object** to generate an equivalent continuous-time controller as follows:

```
C = pid(Kp,Ki,Kd)
C =

              1
  Kp + Ki * --- + Kd * s
              s

  with Kp = 1, Ki = 1, Kd = 1

Continuous-time PID controller in parallel form.
```

Let's convert the pid object to a transfer function to see that it yields the same result as above:

```
tf(C)
ans =
  s^2 + s + 1
  -----------
       s

Continuous-time transfer function.
```

## The Characteristics of P, I, and D Controllers

A proportional controller ($K_p$) will have the effect of reducing the rise time and will reduce but never eliminate the **steady-state error**. An integral control ($K_i$) will have the effect of eliminating the steady-state error for a constant or step input, but it may make the transient response slower. A derivative control ($K_d$) will have the effect of increasing the stability of the system, reducing the overshoot, and improving the transient response.

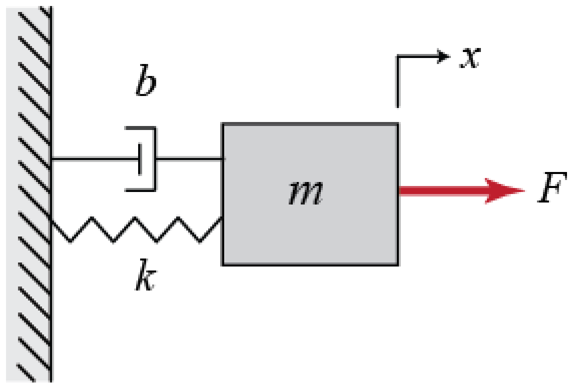
The effects of each of controller parameters, $K_p$, $K_d$, and $K_i$ on a closed-loop system are summarized in the table below.

| CL RESPONSE | RISE TIME | OVERSHOOT | SETTLING TIME | S-S ERROR |
|---|---|---|---|---|
| **Kp** | Decrease | Increase | Small Change | Decrease |
| **Ki** | Decrease | Increase | Increase | Eliminate |
| **Kd** | Small Change | Decrease | Decrease | No Change |

Note that these correlations may not be exactly accurate, because $K_p$, $K_i$, and $K_d$ are dependent on each other. In fact, changing one of these variables can change the effect of the other two. For this reason, the table should only be used as a reference when you are determining the values for $K_i$, $K_p$ and $K_d$.

## Example Problem

Suppose we have a simple mass, spring, and damper problem.



The modeling equation of this system is

(3) $M\ddot{x} + b\dot{x} + kx = F$

Taking the Laplace transform of the modeling equation, we get

(4) $Ms^2X(s) + bsX(s) + kX(s) = F(s)$

The transfer function between the displacement $X(s)$ and the input $F(s)$ then becomes

(5) $\frac{X(s)}{F(s)} = \frac{1}{Ms^2 + bs + k}$

Let

```
M = 1 kg
b = 10 N s/m
k = 20 N/m
F = 1 N
```

Plug these values into the above transfer function

(6) $\frac{X(s)}{F(s)} = \frac{1}{s^2 + 10s + 20}$

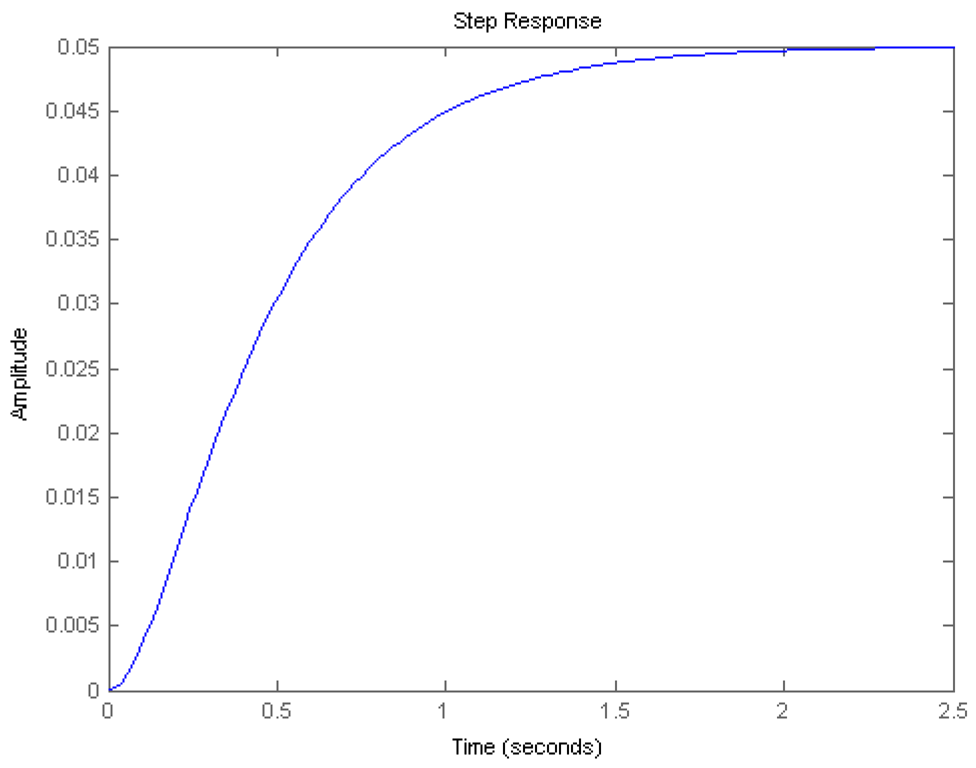
The goal of this problem is to show you how each of $K_p$, $K_i$ and $K_d$ contributes to obtain

```
Fast rise time
Minimum overshoot
No steady-state error
```

## Open-Loop Step Response

Let's first view the open-loop step response. Create a new m-file and run the following code:

```
s = tf('s');
P = 1/(s^2 + 10*s + 20);
step(P)
```



The DC gain of the plant transfer function is 1/20, so 0.05 is the final value of the output to an unit step input. This corresponds to the steady-state error of 0.95, quite large indeed. Furthermore, the rise time is about one second, and the settling time is about 1.5 seconds. Let's design a controller that will reduce the rise time, reduce the settling time, and eliminate the steady-state error.

## Proportional Control

From the table shown above, we see that the proportional controller (Kp) reduces the rise time, increases the overshoot, and reduces the steady-state error.

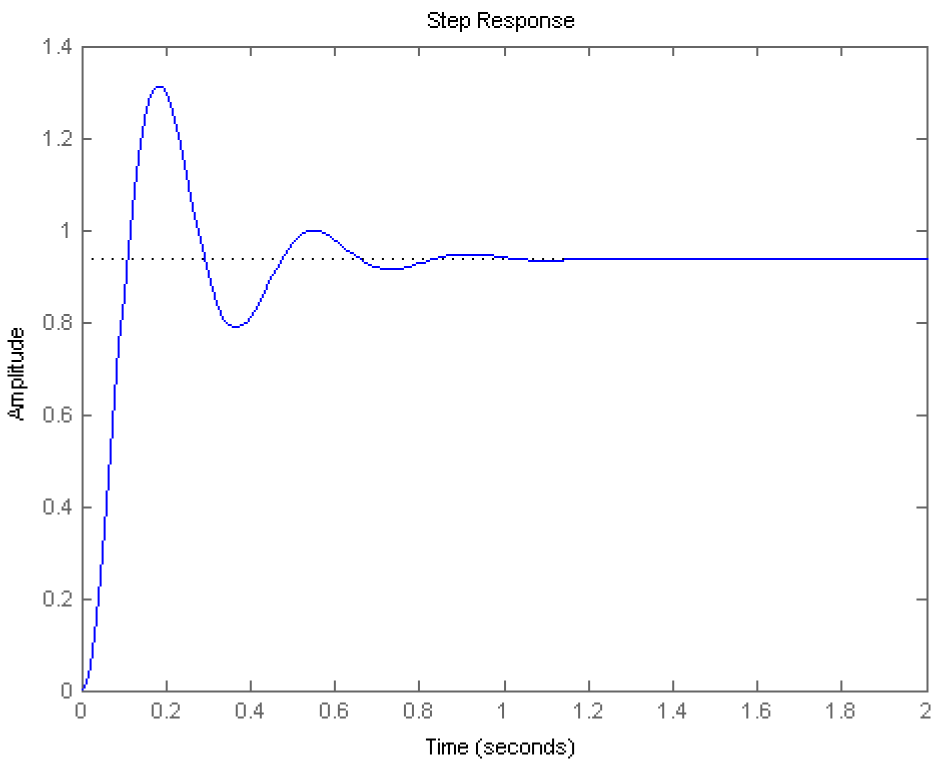
The closed-loop transfer function of the above system with a proportional controller is:

$$(7)\quad \frac{X(s)}{F(s)} = \frac{K_p}{s^2 + 10s + (20 + K_p)}$$

Let the proportional gain ($K_P$) equal 300 and change the m-file to the following:

```
Kp = 300;
C = pid(Kp)
T = feedback(C*P,1)

t = 0:0.01:2;
```

```
step(T,t)
C =

  Kp = 300

P-only controller.


T =

         300
  ----------------
  s^2 + 10 s + 320

Continuous-time transfer function.
```



The above plot shows that the proportional controller reduced both the rise time and the steady-state error, increased the overshoot, and decreased the settling time by small amount.

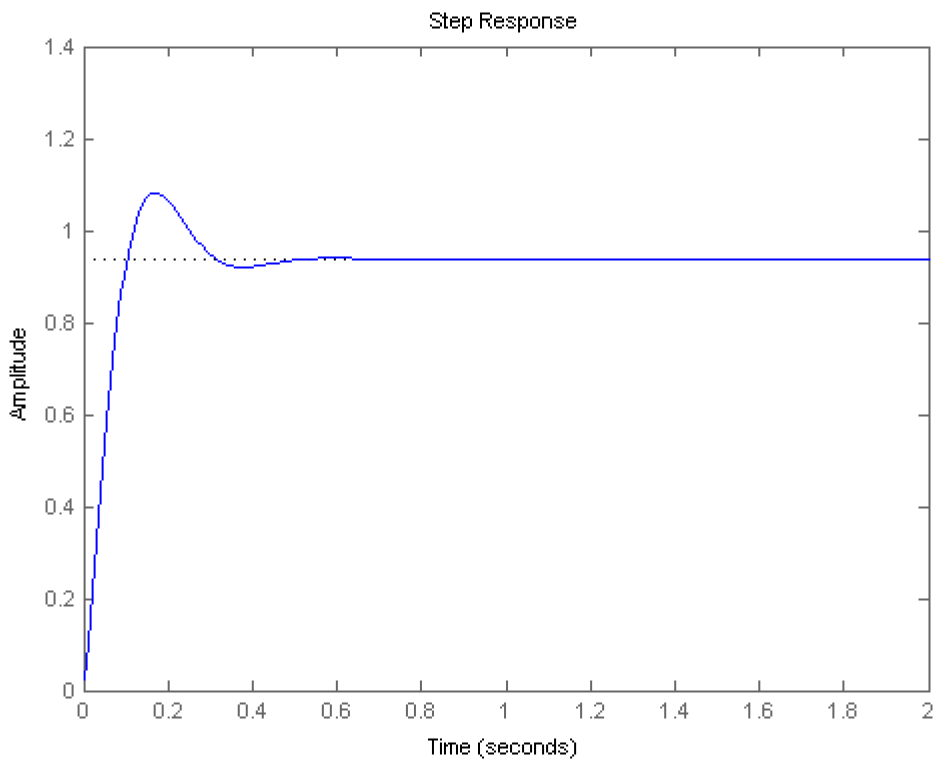
## Proportional-Derivative Control

Now, let's take a look at a PD control. From the table shown above, we see that the derivative controller (Kd) reduces both the overshoot and the settling time. The closed-loop transfer function of the given system with a PD controller is:

(8) $$\frac{X(s)}{F(s)} = \frac{K_d s + K_p}{s^2 + (10 + K_d)s + (20 + K_p)}$$

Let $K_p$ equal 300 as before and let $K_d$ equal 10. Enter the following commands into an m-file and run it in the MATLAB command window.

```
Kp = 300;
Kd = 10;
C = pid(Kp,0,Kd)
T = feedback(C*P,1)

t = 0:0.01:2;
step(T,t)
C =
 

  Kp + Kd * s
 

  with Kp = 300, Kd = 10
 
Continuous-time PD controller in parallel form.


T =
 
     10 s + 300
  ----------------
  s^2 + 20 s + 320
 
Continuous-time transfer function.
```



This plot shows that the derivative controller reduced both the overshoot and the settling time, and had a small effect on the rise time and the steady-state error.

## Proportional-Integral Control

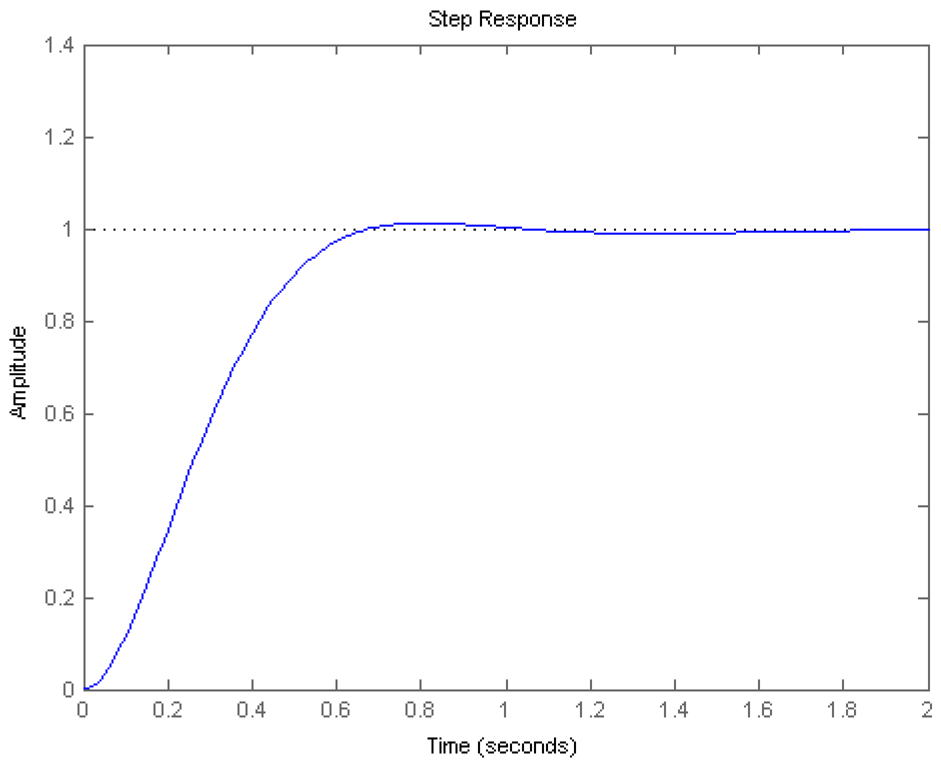
Before going into a PID control, let's take a look at a PI control. From the table, we see that an integral controller (Ki) decreases the rise time, increases both the overshoot and the settling time, and eliminates the steady-state error. For the given system, the closed-loop transfer function with a PI control is:

(9) $$\frac{X(s)}{F(s)} = \frac{K_p s + K_i}{s^3 + 10s^2 + (20 + K_p s + K_i)}$$

Let's reduce the $K_p$to 30, and let $K_i$equal 70. Create an new m-file and enter the following commands.

```
Kp = 30;
Ki = 70;
C = pid(Kp,Ki)
T = feedback(C*P,1)

t = 0:0.01:2;
step(T,t)
```

```
C =

              1
  Kp + Ki * ---
              s

  with Kp = 30, Ki = 70

Continuous-time PI controller in parallel form.


T =

         30 s + 70
  ------------------------
  s^3 + 10 s^2 + 50 s + 70

Continuous-time transfer function.
```



Run this m-file in the MATLAB command window, and you should get the following plot. We have reduced the proportional gain (Kp) because the integral controller also reduces the rise time and increases the overshoot as the proportional controller does (double effect). The above response shows that the integral controller eliminated the steady-state error.

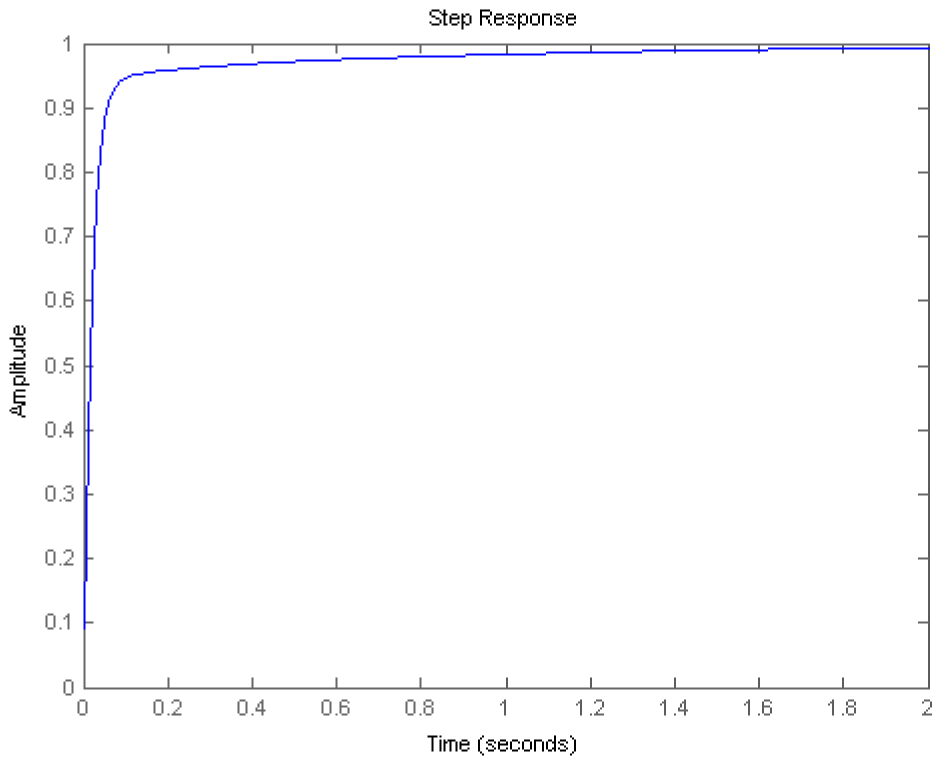
## Proportional-Integral-Derivative Control

Now, let's take a look at a PID controller. The closed-loop transfer function of the given system with a PID controller is:

$$(10)\quad \frac{X(s)}{F(s)} = \frac{K_d s^2 + K_p s + K_i}{s^3 + (10 + K_d)s^2 + (20 + K_p)s + K_i}$$

After several trial and error runs, the gains $K_p$= 350, $K_i$= 300, and $K_d$= 50 provided the desired response. To confirm, enter the following commands to an m-file and run it in the command window. You should get the following step response.

```
Kp = 350;
Ki = 300;
Kd = 50;
C = pid(Kp,Ki,Kd)
T = feedback(C*P,1);

t = 0:0.01:2;
step(T,t)
C =

             1
  Kp + Ki * --- + Kd * s
             s

  with Kp = 350, Ki = 300, Kd = 50

Continuous-time PID controller in parallel form.
```



Now, we have obtained a closed-loop system with no overshoot, fast rise time, and no steady-state error.

## General Tips for Designing a PID Controller

When you are designing a PID controller for a given system, follow the steps shown below to obtain a desired response.

1. Obtain an open-loop response and determine what needs to be improved
2. Add a proportional control to improve the rise time
3. Add a derivative control to improve the overshoot
4. Add an integral control to eliminate the steady-state error
5. Adjust each of Kp, Ki, and Kd until you obtain a desired overall response. You can always refer to the table shown in this "PID Tutorial" page to find out which controller controls what characteristics.

Lastly, please keep in mind that you do not need to implement all three controllers (proportional, derivative, and integral) into a single system, if not necessary. For example, if a PI controller gives a good enough response (like the above example), then you don't need to implement a derivative controller on the system. Keep the controller as simple as possible.

## Automatic PID Tuning

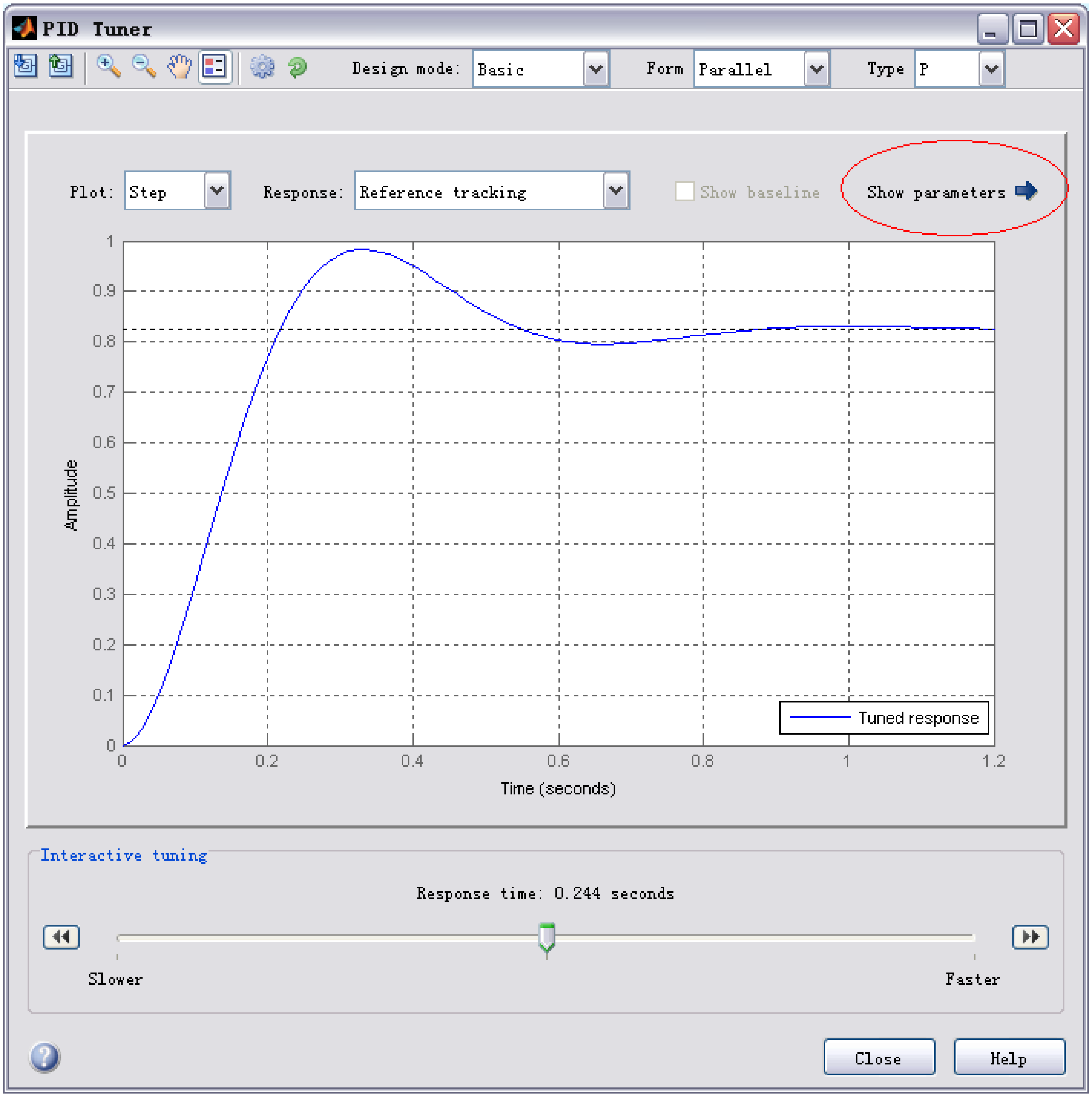
MATLAB provides tools for automatically choosing optimal PID gains which makes the trial and error process described above unnecessary. You can access the tuning algorithm directly using **pidtune** or through a nice graphical user interface (GUI) using **pidtool**.

The MATLAB automated tuning algorithm chooses PID gains to balance performance (response time, bandwidth) and robustness (stability margins). By default the algorthm designs for a 60 degree phase margin.

Let's explore these automated tools by first generating a proportional controller for the mass-spring-damper system by entering the following commands:

```
pidtool(P,'p')
```

The pidtool GUI window, like that shown below, should appear.


PID Tuner

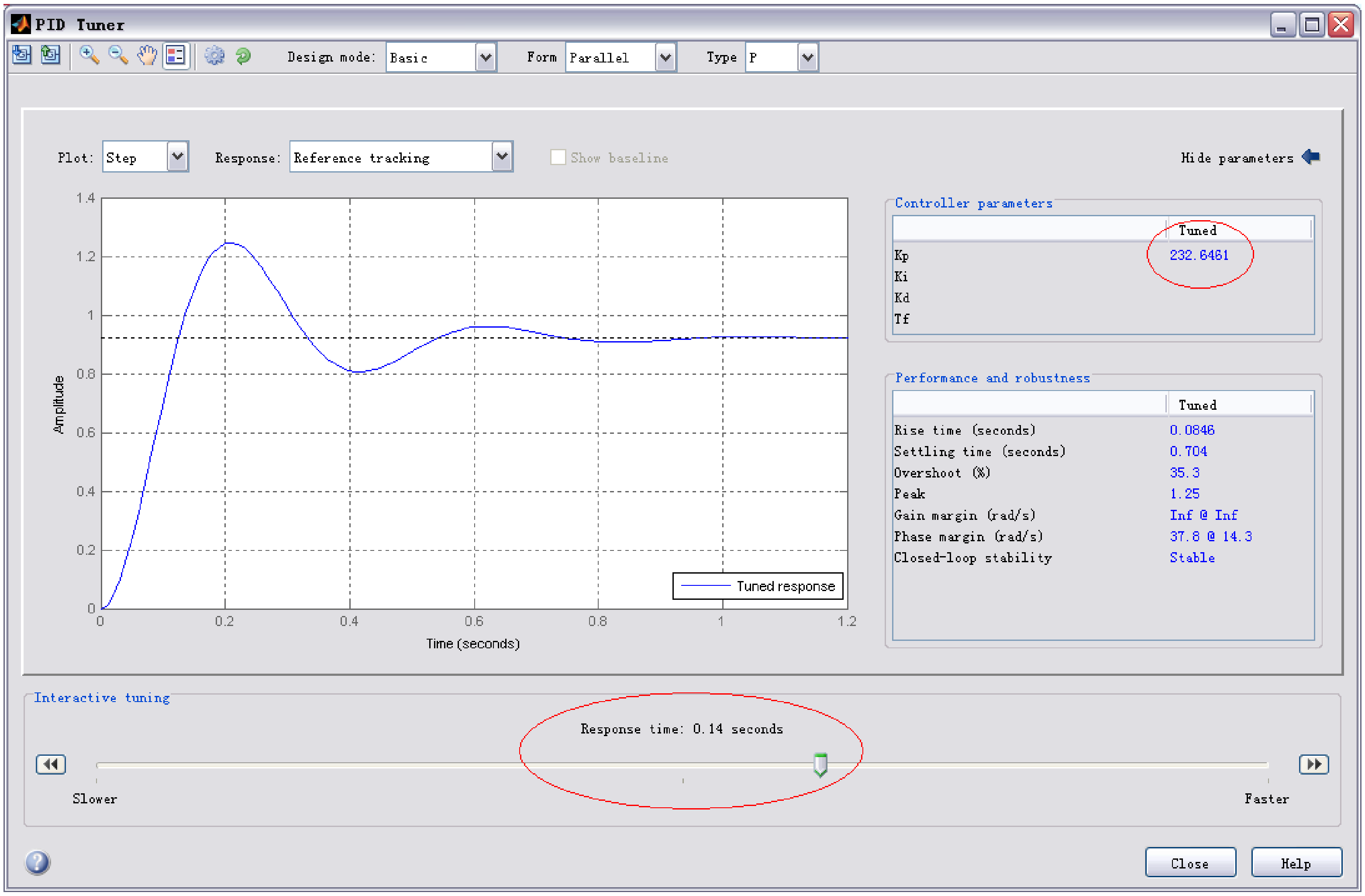
Design mode: Basic  Form Parallel  Type P

Plot: Step  Response: Reference tracking  Show baseline  Show parameters

Response time: 0.244 seconds

Notice that the step response shown is slower than the proportional controller we designed by hand. Now click on the **Show Parameters** button on the top right. As expected the proportional gain constant, Kp, is lower than the one we used, Kp = 94.85 < 300.

We can now interactively tune the controller parameters and immediately see the resulting response int he GUI window. Try dragging the resposne time slider to the right to 0.14s, as shown in the figure below. The response does indeeed speed up, and we can see Kp is now closer to the manual value. We can also see all the other performance and robustness parameters for the system. Note that the phase margin is 60 degrees, the default for pidtool and generally a good balance of robustness and performance.



Controller parameters

| | Tuned |
|---|---|
| Kp | 232.6461 |
| Ki | |
| Kd | |
| Tf | |

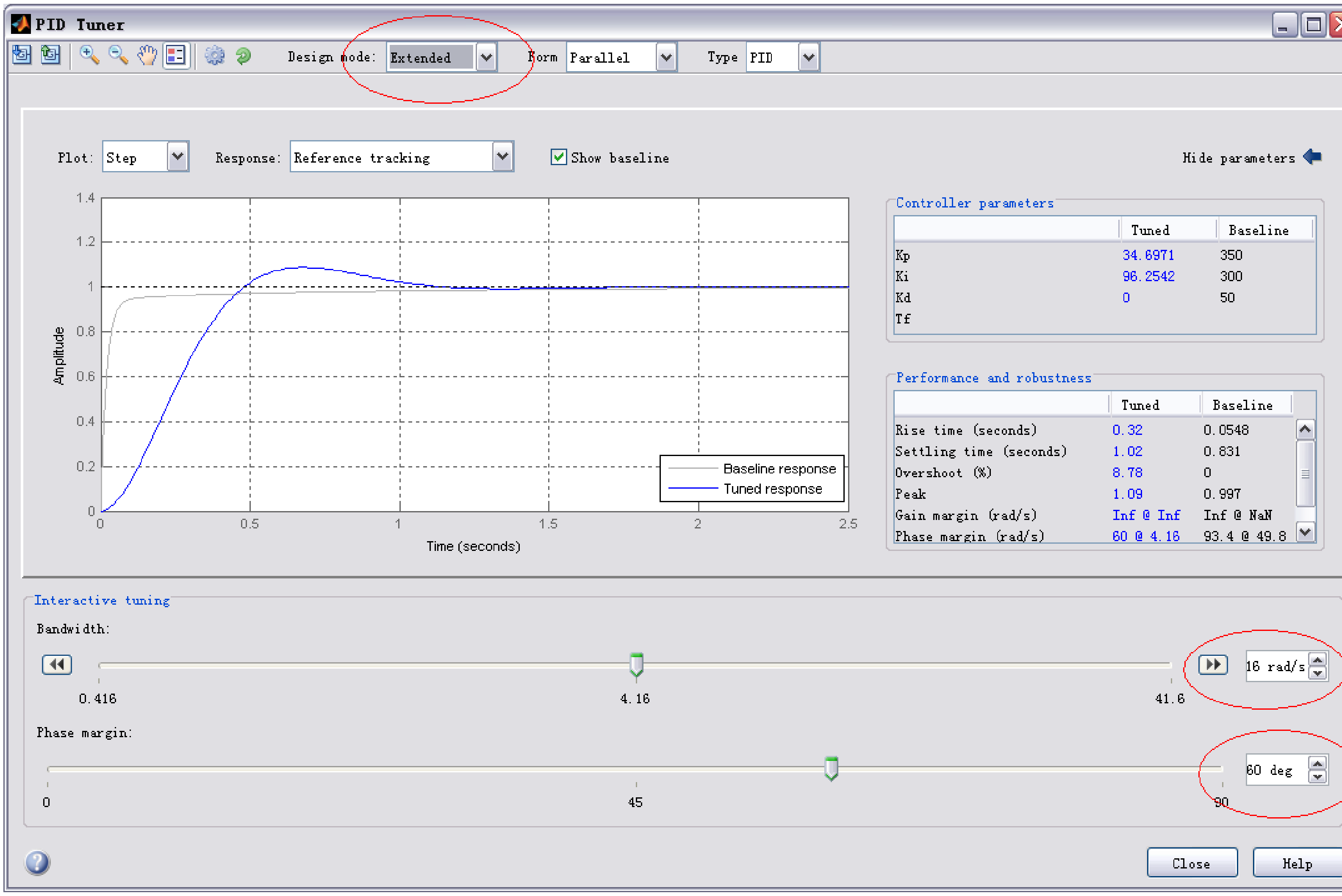
Performance and robustness

| | Tuned |
|---|---|
| Rise time (seconds) | 0.0846 |
| Settling time (seconds) | 0.704 |
| Overshoot (%) | 35.3 |
| Peak | 1.25 |
| Gain margin (rad/s) | Inf @ Inf |
| Phase margin (rad/s) | 37.8 @ 14.3 |
| Closed-loop stability | Stable |

Now let's try designing a PID controller for our system. By specifying the previously designed or (baseline) controller, C, as the second parameter, pidtool will design another PID controller (instead of P or PI) and will compare the response of the system with the automated controller with that of the baseline.

```
pidtool(P,C)
```

We see in the output window that the automated controller responds slower and exhibits more overshoot than the baseline. Now choose the **Design Mode: Extended** option at the top, which reveals more tuning parameters.



PID Tuner

Design mode: Extended  Form Parallel  Type PID

Plot: Step  Response: Reference tracking  [✓] Show baseline  Hide parameters

Controller parameters

| | Tuned | Baseline |
|---|---|---|
| Kp | 34.6971 | 350 |
| Ki | 96.2542 | 300 |
| Kd | 0 | 50 |
| Tf | | |

Performance and robustness

| | Tuned | Baseline |
|---|---|---|
| Rise time (seconds) | 0.32 | 0.0548 |
| Settling time (seconds) | 1.02 | 0.831 |
| Overshoot (%) | 8.78 | 0 |
| Peak | 1.09 | 0.997 |
| Gain margin (rad/s) | Inf @ Inf | Inf @ NaN |
| Phase margin (rad/s) | 60 @ 4.16 | 93.4 @ 49.8 |

Bandwidth: 16 rad/s

Phase margin: 60 deg

Now type in **Bandwidth: 32 rad/s** and **Phase Margin: 90 deg** to generate a controller similar in performance to the baseline. Keep in mind that a higher bandwidth (0 dB crossover of the open-loop) results in a faster rise time, and a higher phase margin reduces the overshoot and improves the system stability.

Finally we note that we can generate the same controller using the command line tool **pidtune** instead of the pidtool GUI

```
opts = pidtuneOptions('CrossoverFrequency',32,'PhaseMargin',90);
[C, info] = pidtune(P, 'pid', opts)
C =

             1
  Kp + Ki * --- + Kd * s
             s

  with Kp = 320, Ki = 169, Kd = 31.5

Continuous-time PID controller in parallel form.
```

```
info =
                  Stable: 1
    CrossoverFrequency: 32
           PhaseMargin: 90
```

# DC Motor Speed: PID Controller Design

Key MATLAB commands used in this tutorial are: `tf` , `step` , `feedback`

## Contents

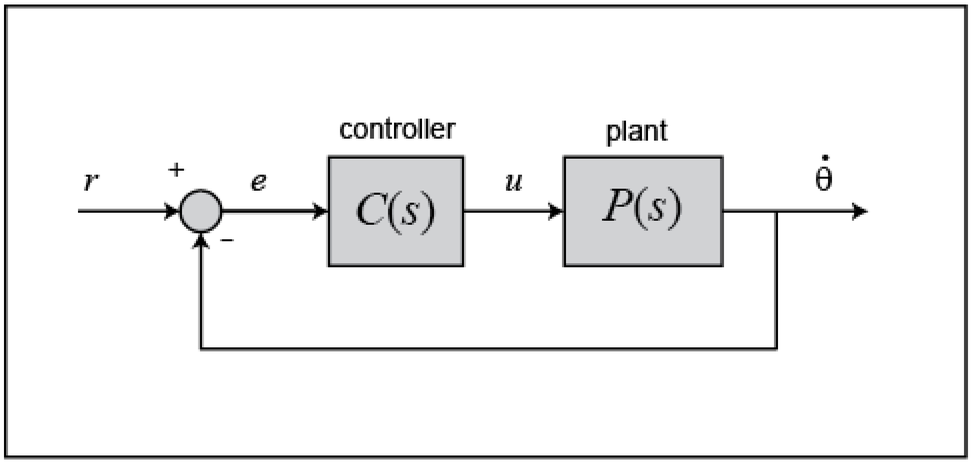
From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

(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 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 page. Create a new m-file 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:

$$C(s) = K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s} \quad (4)$$

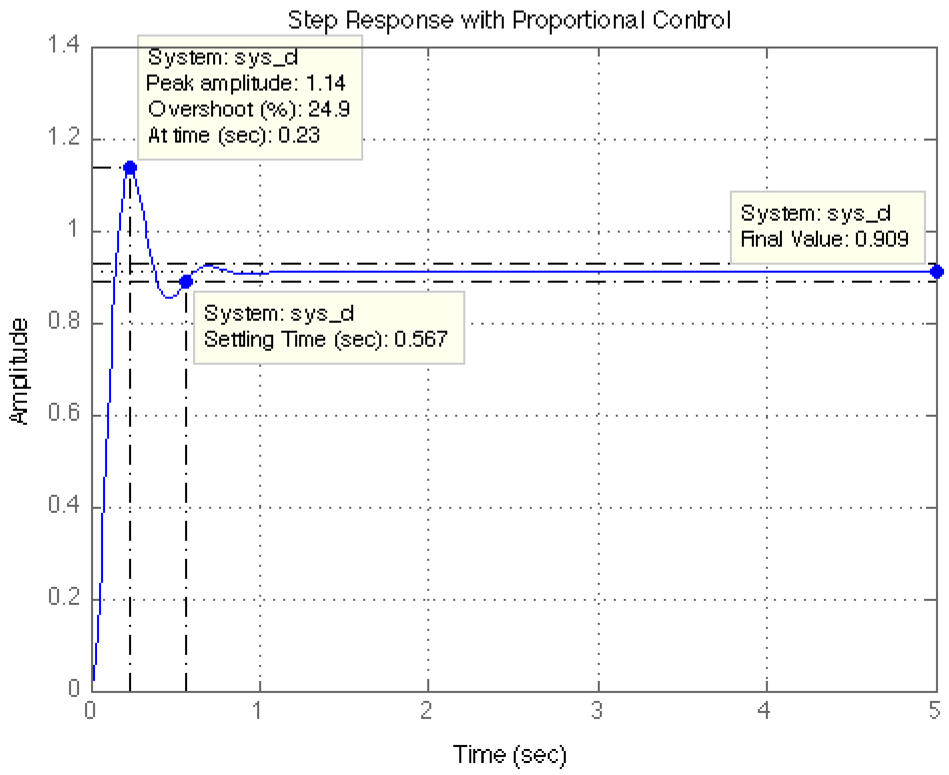
## Proportional control

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')
```



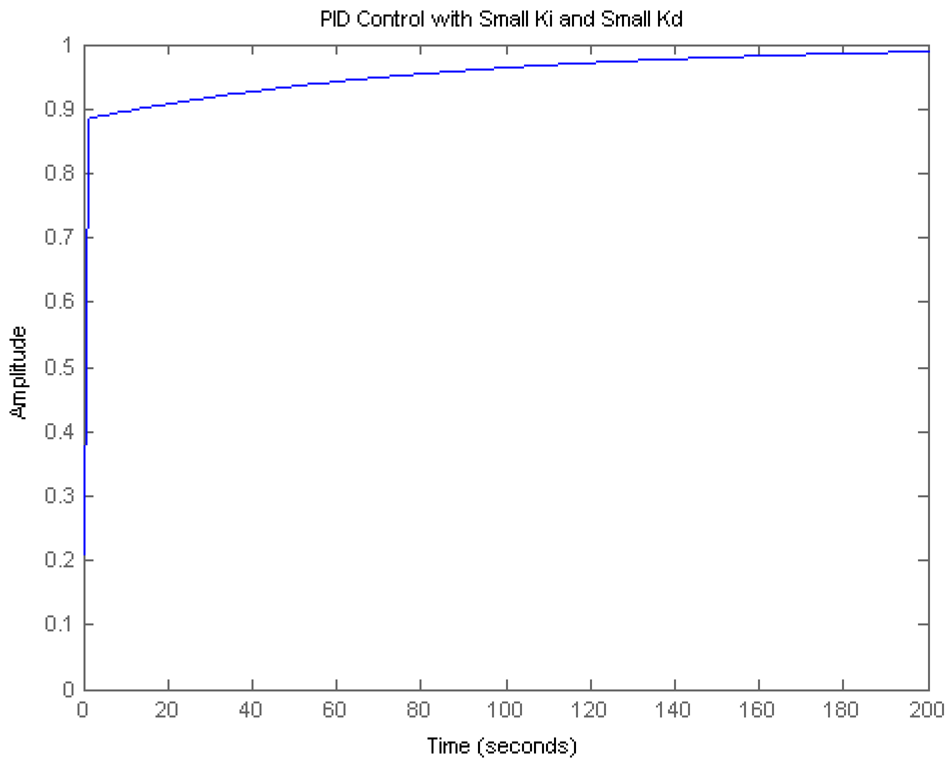
From the plot above we see that both the steady-state error and the overshoot are too large. Recall from the Introduction: PID Controller Design page that increasing the proportional gain *Kp* will reduce the steady-state error. However, also recall that increasing *Kp* often results in increased overshoot, 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 *Kp*. Specifically, you can employ the **SISO Design Tool** by entering the command `sisotool(P_motor)` then opening a closed-loop step response plot from the **Analysis Plots** tab of the **Control and Estimation Tools Manager** window. With the **Real-Time Update** box checked, you can then vary the control gain in the **Compensator Editor** tab and see the resulting effect on the closed-loop step response. 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 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 *Ki* and *Kd*. 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')
```



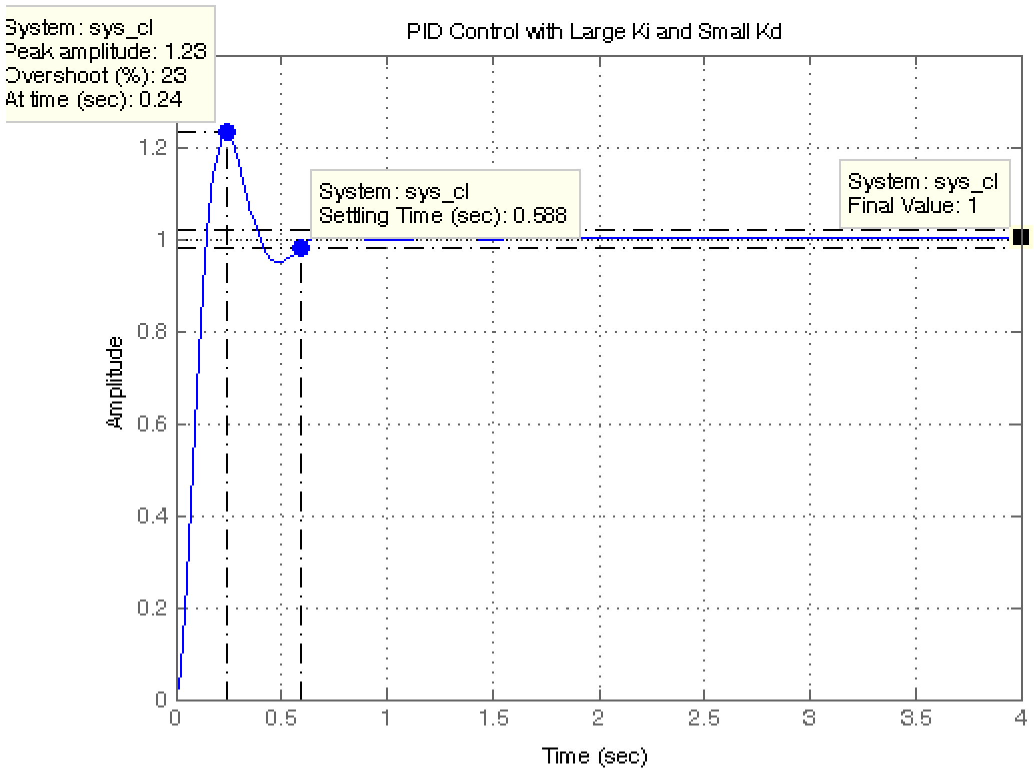
Inspection of the above indicates that the steady-state error does indeed go to zero for a step input. However, the time it takes to reach steady-state is far larger than the required settling time of 2 seconds.

## 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 *Ki*. Go back to your m-file and change *Ki* 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.

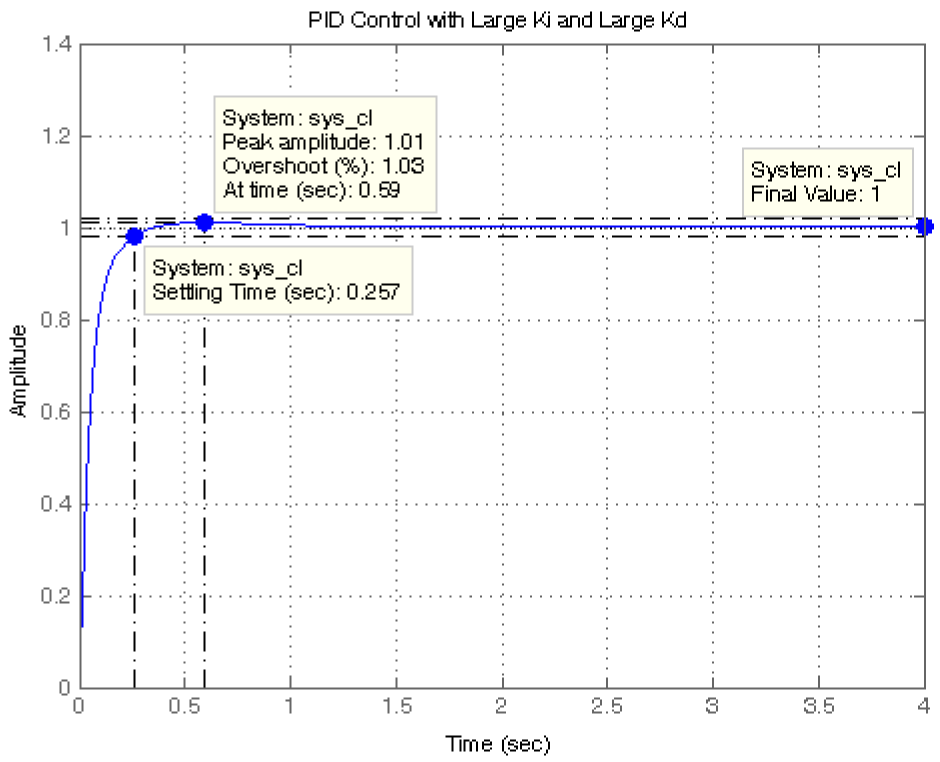
```
Kp = 100;
Ki = 200;
Kd = 1;
C = pid(Kp,Ki,Kd);
```

```
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 *Ki* has greatly increased the overshoot. Let's increase *Kd* in an attempt to reduce the overshoot. Go back to the m-file and change *Kd* 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 *Kd* reduced the resulting overshoot. Now we know that if we use a PID controller with

*Kp* = 100, *Ki* = 200, and *Kd* = 10,

all of our design requirements will be satisfied.