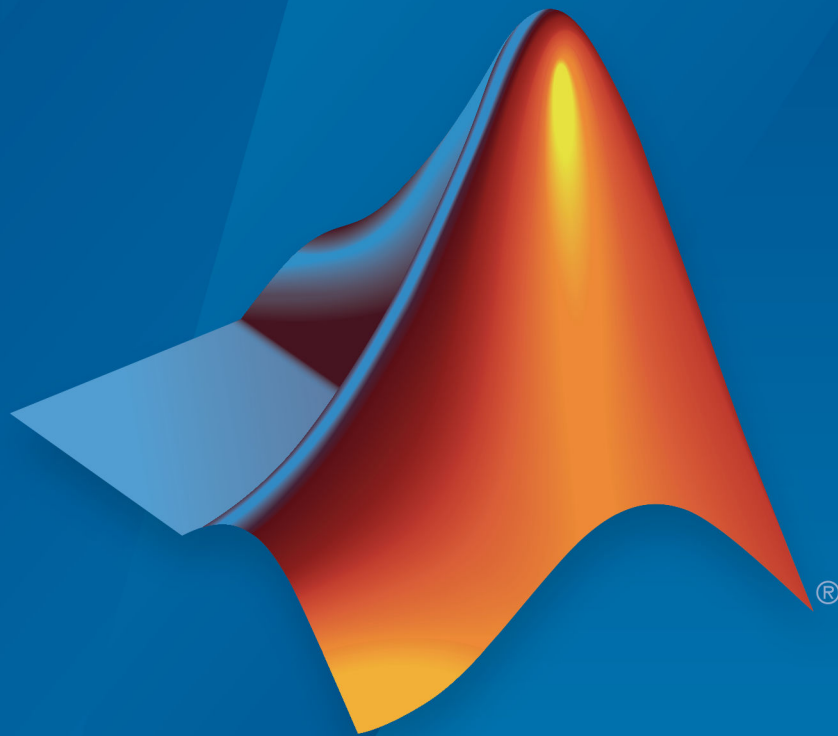# Control System Toolbox™

Getting Started Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Control System Toolbox™ Getting Started Guide

© COPYRIGHT 2000–2018 by The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

November 2002	First printing	New Version 5.0 (Release 12)
June 2001	Second printing	Revised for Version 5.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.2 (Release 13)
June 2004	Online only	Revised for Version 6.0 (Release 14)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 7.0 (Release 2006a)
September 2006	Online only	Revised for Version 7.1 (Release 2006b)
March 2007	Online only	Revised for Version 8.0 (Release 2007a)
September 2007	Online only	Revised for Version 8.0.1 (Release 2007b)
March 2008	Online only	Revised for Version 8.1 (Release 2008a)
October 2008	Third printing	Revised for Version 8.2 (Release 2008b)
March 2009	Online only	Revised for Version 8.3 (Release 2009a)
September 2009	Online only	Revised for Version 8.4 (Release 2009b)
March 2010	Online only	Revised for Version 8.5 (Release 2010a)
September 2010	Online only	Revised for Version 9.0 (Release 2010b)
April 2011	Online only	Revised for Version 9.1 (Release 2011a)
September 2011	Online only	Revised for Version 9.2 (Release 2011b)
March 2012	Online only	Revised for Version 9.3 (Release 2012a)
September 2012	Online only	Revised for Version 9.4 (Release 2012b)
March 2013	Online only	Revised for Version 9.5 (Release 2013a)
September 2013	Online only	Revised for Version 9.6 (Release 2013b)
March 2014	Online only	Revised for Version 9.7 (Release 2014a)
October 2014	Online only	Revised for Version 9.8 (Release 2014b)
March 2015	Online only	Revised for Version 9.9 (Release 2015a)
September 2015	Online only	Revised for Version 9.10 (Release 2015b)
March 2016	Online only	Revised for Version 10.0 (Release 2016a)
September 2016	Online only	Revised for Version 10.1 (Release 2016b)
March 2017	Online only	Revised for Version 10.2 (Release 2017a)
September 2017	Online only	Revised for Version 10.3 (Release 2017b)
March 2018	Online only	Revised for Version 10.4 (Release 2018a)

Contents

Analyzing Models

3

Designing Compensators

4

Product Overview

Control System Toolbox Product Description

Design and analyze control systems

Control System Toolbox provides algorithms and apps for systematically analyzing, designing, and tuning linear control systems. You can specify your system as a transfer function, state-space, zero-pole-gain, or frequency-response model. Apps and functions, such as step response plot and Bode plot, let you analyze and visualize system behavior in the time and frequency domains.

You can tune compensator parameters using interactive techniques such as Bode loop shaping and the root locus method. The toolbox automatically tunes both SISO and MIMO compensators, including PID controllers. Compensators can include multiple tunable blocks spanning several feedback loops. You can tune gain-scheduled controllers and specify multiple tuning objectives, such as reference tracking, disturbance rejection, and stability margins. You can validate your design by verifying rise time, overshoot, settling time, gain and phase margins, and other requirements.

Key Features

- Transfer-function, state-space, zero-pole-gain, and frequency-response models of linear systems
- Step response, Nyquist plot, and other time-domain and frequency-domain tools for analyzing stability and performance
- Automatic tuning of PID, gain-scheduled, and arbitrary SISO and MIMO control systems
- Root locus, Bode diagrams, LQR, LQG, and other classical and state-space design techniques
- Model representation conversion, continuous-time model discretization, and low-order approximation of high-order systems

2

Building Models

Linear (LTI) Models

What Is a Plant?

Typically, control engineers begin by developing a mathematical description of the dynamic system that they want to control. The system to be controlled is called a *plant*. As an example of a plant, this section uses the DC motor. This section develops the differential equations that describe the electromechanical properties of a DC motor with an inertial load. It then shows you how to use the Control System Toolbox functions to build linear models based on these equations.

Linear Model Representations

You can use Control System Toolbox functions to create the following model representations:

- State-space models (SS) of the form

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

  where A, B, C, and D are matrices of appropriate dimensions, x is the state vector, and u and y are the input and output vectors.

- Transfer functions (TF), for example,

$$H(s) = \frac{s+2}{s^2 + s + 10}$$

- Zero-pole-gain (ZPK) models, for example,

$$H(z) = 3\frac{(z+1+j)(z+1-j)}{(z+0.2)(z+0.1)}$$

- Frequency response data (FRD) models, which consist of sampled measurements of a system's frequency response. For example, you can store experimentally collected frequency response data in an FRD model.

> **Note** The design of FRD models is a specialized subject that this topic does not address. See "Frequency Response Data (FRD) Models" for a discussion of this topic.

SISO Example: The DC Motor

A simple model of a DC motor driving an inertial load shows the angular rate of the load,

$\omega(t)$, as the output and applied voltage, $v_{app}(t)$, as the input. The ultimate goal of this example is to control the angular rate by varying the applied voltage. This figure shows a simple model of the DC motor.
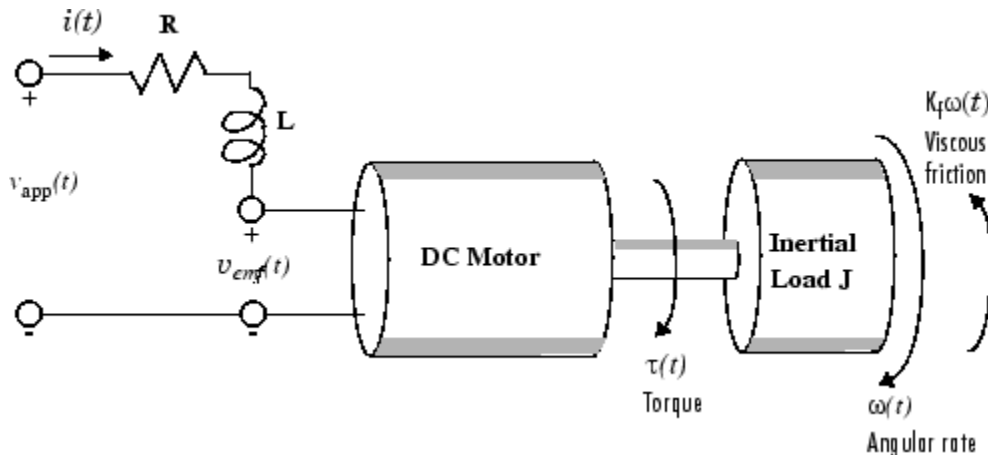
A Simple Model of a DC Motor Driving an Inertial Load

In this model, the dynamics of the motor itself are idealized; for instance, the magnetic field is assumed to be constant. The resistance of the circuit is denoted by R and the self-inductance of the armature by L. If you are unfamiliar with the basics of DC motor modeling, consult any basic text on physical modeling. With this simple model and basic laws of physics, it is possible to develop differential equations that describe the behavior of this electromechanical system. In this example, the relationships between electric potential and mechanical force are Faraday's law of induction and Ampère's law for the force on a conductor moving through a magnetic field.

Mathematical Derivation

The torque τ seen at the shaft of the motor is proportional to the current i induced by the applied voltage,

$$\tau(t) = K_m i(t)$$

where K_m, the armature constant, is related to physical properties of the motor, such as magnetic field strength, the number of turns of wire around the conductor coil, and so on.

The back (induced) electromotive force, υ_{emf}, is a voltage proportional to the angular rate ω seen at the shaft,

$$\upsilon_{emf}(t) = K_b \omega(t)$$

where K_b, the emf constant, also depends on certain physical properties of the motor.

The mechanical part of the motor equations is derived using Newton's law, which states that the inertial load J times the derivative of angular rate equals the sum of all the torques about the motor shaft. The result is this equation,

$$J \frac{dw}{dt} = \sum \tau_i = -K_f \omega(t) + K_m i(t)$$

where $K_f \omega$ is a linear approximation for viscous friction.

Finally, the electrical part of the motor equations can be described by

$$\upsilon_{app}(t) - \upsilon_{emf}(t) = L \frac{di}{dt} + Ri(t)$$

or, solving for the applied voltage and substituting for the back emf,

$$\upsilon_{app}(t) = L \frac{di}{dt} + Ri(t) + K_b \omega(t)$$

This sequence of equations leads to a set of two differential equations that describe the behavior of the motor, the first for the induced current,

$$\frac{di}{dt} = -\frac{R}{L} i(t) - \frac{K_b}{L} \omega(t) + \frac{1}{L} \upsilon_{app}(t)$$

and the second for the resulting angular rate,

$$\frac{d\omega}{dt} = -\frac{1}{J}K_f\omega(t) + \frac{1}{J}K_m i(t)$$

State-Space Equations for the DC Motor

Given the two differential equations derived in the last section, you can now develop a state-space representation of the DC motor as a dynamic system. The current *i* and the

angular rate ω are the two states of the system. The applied voltage, υ_{app}, is the input to the system, and the angular velocity ω is the output.

$$\frac{d}{dt}\begin{bmatrix} i \\ \omega \end{bmatrix} = \begin{bmatrix} -\dfrac{R}{L} & -\dfrac{K_b}{L} \\ \dfrac{K_m}{J} & -\dfrac{K_f}{J} \end{bmatrix} \cdot \begin{bmatrix} i \\ \omega \end{bmatrix} + \begin{bmatrix} \dfrac{1}{L} \\ 0 \end{bmatrix} \cdot \upsilon_{app}(t)$$

$$y(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ \omega \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \cdot \upsilon_{app}(t)$$

State-Space Representation of the DC Motor Example

Building SISO Models

After you develop a set of differential equations that describe your plant, you can construct SISO models using simple commands. The following sections discuss

- Constructing a state-space model of the DC motor
- Converting between model representations
- Creating transfer function and zero/pole/gain models

Constructing a State-Space Model of the DC Motor

Enter the following nominal values for the various parameters of a DC motor.

```
R= 2.0 % Ohms
L= 0.5 % Henrys
Km = .015 % torque constant
Kb = .015 % emf constant
```

```
Kf = 0.2 % Nms
J= 0.02 % kg.m^2
```

Given these values, you can construct the numerical state-space representation using the `ss` function.

```
A = [-R/L -Kb/L; Km/J -Kf/J]
B = [1/L; 0];
C = [0 1];
D = [0];
sys_dc = ss(A,B,C,D)
```

These commands return the following result:

```
a =
                      x1           x2
          x1          -4        -0.03
          x2        0.75          -10


b =
                      u1
          x1           2
          x2           0


c =
                      x1           x2
          y1           0            1


d =
                      u1
          y1           0
```

Converting Between Model Representations

Now that you have a state-space representation of the DC motor, you can convert to other model representations, including transfer function (TF) and zero/pole/gain (ZPK) models.

Transfer Function Representation

You can use `tf` to convert from the state-space representation to the transfer function. For example, use this code to convert to the transfer function representation of the DC motor.

```
sys_tf = tf(sys_dc)

Transfer function:
      1.5
------------------
s^2 + 14 s + 40.02
```

Zero/Pole/Gain Representation

Similarly, the `zpk` function converts from state-space or transfer function representations to the zero/pole/gain format. Use this code to convert from the state-space representation to the zero/pole/gain form for the DC motor.

```
sys_zpk = zpk(sys_dc)

Zero/pole/gain:
        1.5
------------------
(s+4.004) (s+9.996)
```

Note The state-space representation is best suited for numerical computations. For highest accuracy, convert to state space prior to combining models and avoid the transfer function and zero/pole/gain representations, except for model specification and inspection.

Constructing Transfer Function and Zero/Pole/Gain Models

In the DC motor example, the state-space approach produces a set of matrices that represents the model. If you choose a different approach, you can construct the corresponding models using `tf`, `zpk`, `ss`, or `frd`.

```
sys = tf(num,den)              % Transfer function
sys = zpk(z,p,k)               % Zero/pole/gain
sys = ss(a,b,c,d)              % State-space
sys = frd(response,frequencies) % Frequency response data
```

For example, you can create the transfer function by specifying the numerator and denominator with this code.

```
sys_tf = tf(1.5,[1 14 40.02])
```

```
Transfer function:
        1.5
------------------
s^2 + 14 s + 40.02
```

Alternatively, if you want to create the transfer function of the DC motor directly, use these commands.

```
s = tf('s');
sys_tf = 1.5/(s^2+14*s+40.02)
```

These commands result in this transfer function.

```
Transfer function:
         1.5
--------------------
s^2 + 14 s + 40.02
```

To build the zero/pole/gain model, use this command.

```
sys_zpk = zpk([],[-9.996 -4.004], 1.5)
```

This command returns the following zero/pole/gain representation.

```
Zero/pole/gain:
        1.5
------------------
(s+9.996) (s+4.004)
```

Constructing Discrete Time Systems

The Control System Toolbox software provides full support for discrete-time systems. You can create discrete systems in the same way that you create analog systems; the only difference is that you must specify a sample time period for any model you build. For example,

```
sys_disc = tf(1, [1 1], .01);
```

creates a SISO model in the transfer function format.

```
Transfer function:
  1
-----
z + 1

Sample time: 0.01
```

Adding Time Delays to Discrete-Time Models

You can add time delays to discrete-time models by specifying an input delay, output delay, or I/O delay when building the model. The time delay must be a nonnegative integer that represents a multiple of the sample time. For example,

```
sys_delay = tf(1, [1 1], 0.01,'ioDelay',5)
```

returns a system with an I/O delay of 5 s.

```
Transfer function:
            1
z^(-5) * -----
         z + 1

Sample time: 0.01
```

Adding Delays to Linear Models

You can add time delays to linear models by specifying an input delay, output delay, or I/O delay when building a model. For example, to add an I/O delay to the DC motor, use this code.

```
sys_tfdelay = tf(1.5,[1 14 40.02],'ioDelay',0.05)
```

This command constructs the DC motor transfer function, but adds a 0.05 second delay.

```
Transfer function:
                   1.5
exp(-0.05*s) * ------------------
               s^2 + 14 s + 40.02
```

For more information about adding time delays to models, see "Time Delays in Linear Systems".

LTI Objects

For convenience, the Control System Toolbox software uses custom data structures called *LTI objects* to store model-related data. For example, the variable `sys_dc` created for the DC motor example is called an *SS object*. There are also TF, ZPK, and FRD objects for transfer function, zero/pole/gain, and frequency data response models respectively. The four LTI objects encapsulate the model data and enable you to manipulate linear systems as single entities rather than as collections of vectors or matrices.

To see what LTI objects contain, use the `get` command. This code describes the contents of `sys_dc` from the DC motor example.

```
get(sys_dc)
                A: [2×2 double]
                B: [2×1 double]
                C: [0 1]
                D: 0
                E: []
           Scaled: 0
        StateName: {2×1 cell}
        StateUnit: {2×1 cell}
    InternalDelay: [0×1 double]
       InputDelay: 0
      OutputDelay: 0
               Ts: 0
         TimeUnit: 'seconds'
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1×1 struct]
       OutputName: {''}
       OutputUnit: {''}
      OutputGroup: [1×1 struct]
            Notes: [0×1 string]
         UserData: []
             Name: ''
     SamplingGrid: [1×1 struct]
```

You can manipulate the data contained in LTI objects using the `set` command; see the Control System Toolbox online reference pages for descriptions of `set` and `get`.

Another convenient way to set or retrieve LTI model properties is to access them directly using dot notation. For example, if you want to access the value of the A matrix, instead of using `get`, you can type

```
sys_dc.A
```

at the MATLAB® prompt. This notation returns the A matrix.

```
ans =

   -4.0000    -0.0300
    0.7500   -10.0000
```

Similarly, if you want to change the values of the A matrix, you can do so directly, as this code shows.

```
A_new = [-4.5 -0.05; 0.8 -12.0];
sys_dc.A = A_new;
```

See Also
ss | tf | zpk

Related Examples
- "Transfer Functions"
- "State-Space Models"
- "Discrete-Time Numeric Models"

More About
- "Numeric Models"

MIMO Models

State-Space Model of Jet Transport Aircraft

This example shows how to build a MIMO model of a jet transport. Because the development of a physical model for a jet aircraft is lengthy, only the state-space equations are presented here. See any standard text in aviation for a more complete discussion of the physics behind aircraft flight.

The jet model during cruise flight at MACH = 0.8 and H = 40,000 ft. is

```
A = [-0.0558   -0.9968    0.0802    0.0415
      0.5980   -0.1150   -0.0318         0
     -3.0500    0.3880   -0.4650         0
          0     0.0805    1.0000        0];

B = [ 0.0073         0
     -0.4750    0.0077
      0.1530    0.1430
          0        0];

C = [0     1    0    0
     0     0    0    1];

D = [0     0
     0     0];
```

Use the following commands to specify this state-space model as an LTI object and attach names to the states, inputs, and outputs.

```
states = {'beta' 'yaw' 'roll' 'phi'};
inputs = {'rudder' 'aileron'};
outputs = {'yaw rate' 'bank angle'};

sys_mimo = ss(A,B,C,D,'statename',states,...
'inputname',inputs,...
'outputname',outputs);
```

You can display the LTI model by typing `sys_mimo`.

```
sys_mimo

a =
```

```
                      beta          yaw          roll          phi
         beta      -0.0558      -0.9968        0.0802        0.0415
          yaw        0.598       -0.115       -0.0318             0
         roll        -3.05        0.388        -0.465             0
          phi            0       0.0805             1             0


b =
                    rudder      aileron
         beta       0.0073            0
          yaw       -0.475       0.0077
         roll        0.153        0.143
          phi            0            0


c =
                      beta          yaw          roll          phi
     yaw rate            0            1             0             0
   bank angle            0            0             0             1


d =
                    rudder      aileron
     yaw rate            0            0
   bank angle            0            0
```

Continuous-time model.

The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in degrees.

As in the SISO case, use `tf` to derive the transfer function representation.

`tf(sys_mimo)`

```
Transfer function from input "rudder" to output...
            -0.475 s^3 - 0.2479 s^2 - 0.1187 s - 0.05633
 yaw rate:  ----------------------------------------------------
            s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674


                    0.1148 s^2 - 0.2004 s - 1.373
 bank angle:  ----------------------------------------------------
            s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674
```

```
Transfer function from input "aileron" to output...
           0.0077 s^3 - 0.0005372 s^2 + 0.008688 s + 0.004523
 yaw rate:  -------------------------------------------------
           s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

                      0.1436 s^2 + 0.02737 s + 0.1104
 bank angle:  -------------------------------------------------
             s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674
```

Constructing MIMO Transfer Functions

MIMO transfer functions are two-dimensional arrays of elementary SISO transfer functions. There are two ways to specify MIMO transfer function models:

- Concatenation of SISO transfer function models
- Using `tf` with cell array arguments

Concatenation of SISO Models

Consider the following single-input, two-output transfer function.

$$H(s) = \begin{bmatrix} \dfrac{s-1}{s+1} \\ \dfrac{s+2}{s^2 + 4s + 5} \end{bmatrix}$$

You can specify $H(s)$ by concatenation of its SISO entries. For instance,

```
h11 = tf([1 -1],[1 1]);
h21 = tf([1 2],[1 4 5]);
```

or, equivalently,

```
s = tf('s')
h11 = (s-1)/(s+1);
h21 = (s+2)/(s^2+4*s+5);
```

can be concatenated to form $H(s)$.

```
H = [h11; h21]
```

This syntax mimics standard matrix concatenation and tends to be easier and more readable for MIMO systems with many inputs and/or outputs.

Using the tf Function with Cell Arrays

Alternatively, to define MIMO transfer functions using tf, you need two cell arrays (say, N and D) to represent the sets of numerator and denominator polynomials, respectively. See "What Is a Cell Array?" (MATLAB) for more details on cell arrays.

For example, for the rational transfer matrix $H(s)$, the two cell arrays N and D should contain the row-vector representations of the polynomial entries of

$$N(s) = \left[\frac{s-1}{s+2} \right] \qquad D(s) = \left[\frac{s+1}{s^2 + 4s + 5} \right]$$

You can specify this MIMO transfer matrix $H(s)$ by typing

```
N = {[1 -1];[1 2]};    % Cell array for N(s)
D = {[1 1];[1 4 5]}; % Cell array for D(s)
H = tf(N,D)
```

These commands return the following result:

```
Transfer function from input to output...
       s - 1
 #1:  -----
       s + 1

          s + 2
 #2:  -------------
       s^2 + 4 s + 5
```

Notice that both N and D have the same dimensions as H. For a general MIMO transfer matrix $H(s)$, the cell array entries N{i,j} and D{i,j} should be row-vector representations of the numerator and denominator of $H_{ij}(s)$, the ijth entry of the transfer matrix $H(s)$.

Accessing I/O Pairs in MIMO Systems

After you define a MIMO system, you can access and manipulate I/O pairs by specifying input and output pairs of the system. For instance, if sys_mimo is a MIMO system with two inputs and three outputs,

```
sys_mimo(3,1)
```

extracts the subsystem, mapping the first input to the third output. Row indices select the outputs and column indices select the inputs. Similarly,

```
sys_mimo(3,1) = tf(1,[1 0])
```

redefines the transfer function between the first input and third output as an integrator.

Arrays of Linear Models

You can specify and manipulate collections of linear models as single entities using LTI arrays. For example, if you want to vary the Kb and Km parameters for the DC motor and store the resulting state-space models, use this code.

```
K = [0.1 0.15 0.2]; % Several values for Km and Kb
A1 = [-R/L -K(1)/L; K(1)/J -Kf/J];
A2 = [-R/L -K(2)/L; K(2)/J -Kf/J];
A3 = [-R/L -K(3)/L; K(3)/J -Kf/J];
sys_lti(:,:,1)= ss(A1,B,C,D);
sys_lti(:,:,2)= ss(A2,B,C,D);
sys_lti(:,:,3)= ss(A3,B,C,D);
```

The number of inputs and outputs must be the same for all linear models encapsulated by the LTI array, but the model order (number of states) can vary from model to model within a single LTI array.

The LTI array `sys_lti` contains the state-space models for each value of K. Type `sys_lti` to see the contents of the LTI array.

```
Model sys_lti(:,:,1,1)
======================

  a =
                        x1            x2
            x1          -4          -0.2
            x2           5           -10
.
.
.
Model sys_lti(:,:,2,1)
======================

  a =
                        x1            x2
            x1          -4          -0.3
            x2         7.5           -10
.
.
.
Model sys_lti(:,:,3,1)
======================
```
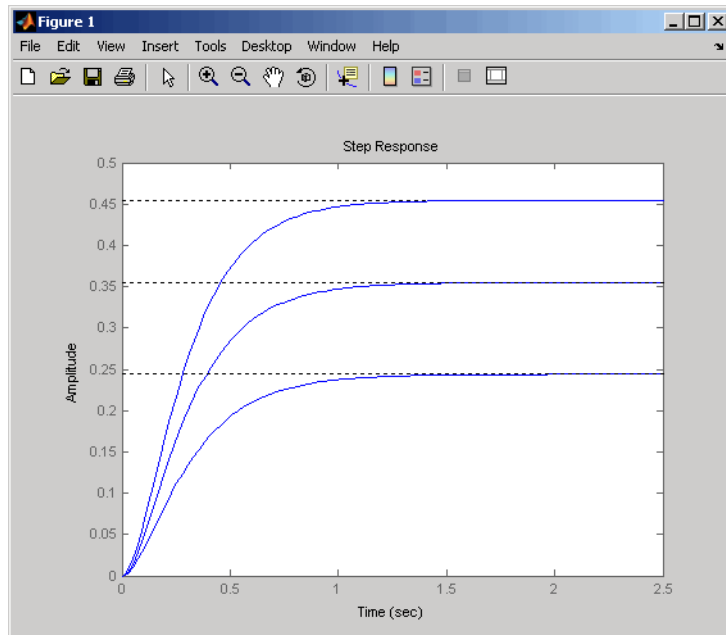
```
a =
                        x1            x2
         x1             -4          -0.4
         x2             10           -10
.
.
.
3x1 array of continuous-time state-space models.
```

You can manipulate the LTI array like any other object. For example,

```
step(sys_lti)
```

produces a plot containing step responses for all three state-space models.

Step Responses for an LTI Array Containing Three Models

LTI arrays are useful for performing batch analysis on an entire set of models. For more information, see "Model Arrays".

Model Characteristics

You can use the Control System Toolbox commands to query model characteristics such as the I/O dimensions, poles, zeros, and DC gain. These commands apply to both continuous- and discrete-time models. Their LTI-based syntax is summarized in the table below.

Commands to Query Model Characteristics

Command	Description
size(model_name)	Number of inputs and outputs
ndims(model_name)	Number of dimensions
isct(model_name)	Returns 1 for continuous systems
isdt(model_name)	Returns 1 for discrete systems
hasdelay(model_name)	True if system has delays
pole(model_name)	System poles
zero(model_name)	System (transmission) zeros
dcgain(model_name)	DC gain
norm(model_name)	System norms (H_2 and L_∞)
covar(model_name,W)	Covariance of response to white noise
bandwidth(model_name)	Frequency response bandwidth
order(model_name)	LTI model order
pzmap(model_name)	Compute pole-zero map of LTI models
damp(model_name)	Natural frequency and damping of system poles
class(model_name)	Create object or return class of object
isa(model_name)	Determine whether input is object of given class
isempty(model_name)	Determine whether LTI model is empty
isproper(model_name)	Determine whether LTI model is proper
issiso(model_name)	Determine whether LTI model is single-input/single-output (SISO)
isstable(model_name)	Determine whether system is stable
reshape(model_name)	Change shape of LTI array

Interconnecting Linear Models

Arithmetic Operations for Interconnecting Models

You can perform arithmetic on LTI models, such as addition, multiplication, or concatenation. Addition performs a parallel interconnection. For example, typing

```
tf(1,[1 0]) + tf([1 1],[1 2])   % 1/s + (s+1)/(s+2)
```

produces this transfer function.

```
Transfer function:
s^2 + 2 s + 2
-------------
  s^2 + 2 s
```

Multiplication performs a series interconnection. For example, typing

```
2 * tf(1,[1 0])*tf([1 1],[1 2])   % 2*1/s*(s+1)/(s+2)
```

produces this cascaded transfer function.

```
Transfer function:
2 s + 2
---------
s^2 + 2 s
```

If the operands are models of different types, the resulting model type is determined by precedence rules; see "Rules That Determine Model Type" for more information.

For more information about model arithmetic functions, see "Catalog of Model Interconnections".

You can also use the `series` and `parallel` functions as substitutes for multiplication and addition, respectively.

Equivalent Ways to Interconnect Systems

Operator	Function	Resulting Transfer Function
`sys1 + sys2`	`parallel(sys1,sys2)`	Systems in parallel
`sys1 - sys2`	`parallel(sys1,-sys2)`	Systems in parallel
`sys1 * sys2`	`series(sys2,sys1)`	Cascaded systems

Feedback Interconnections

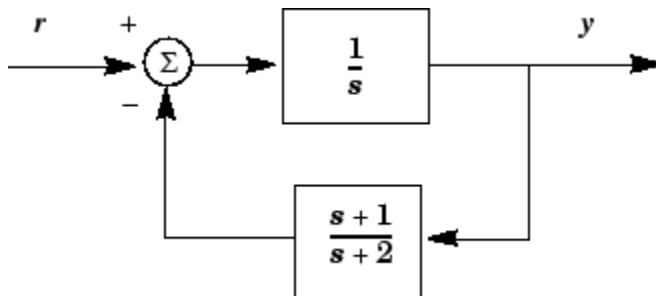You can use the `feedback` and `lft` functions to derive closed-loop models. For example,

```
sys_f = feedback(tf(1,[1 0]), tf([1 1],[1 2])
```

computes the closed-loop transfer function from *r* to *y* for the feedback loop shown below. The result is

```
Transfer function:
    s + 2
-------------
s^2 + 3 s + 1
```

This figure shows the interconnected system in block diagram format.

Feedback Interconnection

You can use the `lft` function to create more complicated feedback structures. This function constructs the linear fractional transformation of two systems. See the reference page for more information.

Converting Between Continuous- and Discrete- Time Systems

Available Commands for Continuous/Discrete Conversion

The commands `c2d`, `d2c`, and `d2d` perform continuous to discrete, discrete to continuous, and discrete to discrete (resampling) conversions, respectively.

```
sysd = c2d(sysc,Ts)   % Discretization w/ sample period Ts
sysc = d2c(sysd)      % Equivalent continuous-time model
sysd1= d2d(sysd,Ts)   % Resampling at the period Ts
```

Available Methods for Continuous/Discrete Conversion

Various discretization/interpolation methods are available, including zero-order hold (default), first-order hold, Tustin approximation with or without prewarping, and matched zero-pole. For example,

```
sysd = c2d(sysc,Ts,'foh')   % Uses first-order hold
sysc = d2c(sysd,'tustin')   % Uses Tustin approximation
```

Digitizing the Discrete DC Motor Model

You can digitize the DC motor plant using the `c2d` function and selecting an appropriate sample time. Choosing the right sample time involves many factors, including the performance you want to achieve, the fastest time constant in your system, and the speed at which you expect your controller to run. For this example, choose a time constant of 0.01 second. See "SISO Example: The DC Motor" on page 2-3 for the construction of the SS object `sys_dc`.

```
Ts=0.01;
sysd=c2d(sys_dc,Ts)

a =
                    x1          x2
          x1     0.96079   -0.00027976
          x2    0.006994       0.90484


b =
```

```
                     u1
        x1     0.019605
        x2  7.1595e-005



c =
                     x1            x2
        y1            0             1



d =
                     u1
        y1            0
```

Sample time: 0.01
Discrete-time model.

To see the discrete-time zero-pole gain for the digital DC motor, use `zpk` to convert the model.

```
fd=zpk(sysd)
```

```
Zero/pole/gain:
7.1595e-005 (z+0.9544)
-----------------------
 (z-0.9608) (z-0.9049)
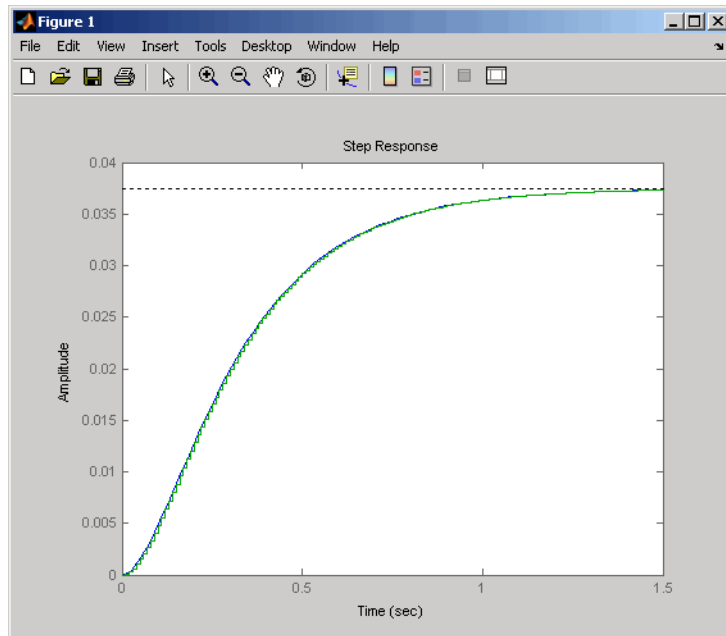```

Sample time: 0.01

You can compare the step responses of sys_dc and sysd by typing

```
step(sys_dc,sysd)
```

This figure shows the result.

Note the step response match. Continuous and FOH-discretized step responses match for models without internal delays.

Reducing Model Order

Model Order Reduction Commands

You can derive reduced-order SISO and MIMO models with the commands shown in the following table.

Model Order Reduction Commands	
hsvd	Compute Hankel singular values of LTI model
balred	Reduced-order model approximation
minreal	Minimal realization (pole/zero cancellation)
balreal	Compute input/output balanced realization
modred	State deletion in I/O balanced realization
sminreal	Structurally minimal realization

Techniques for Reducing Model Order

To reduce the order of a model, you can perform any of the following actions:

- Eliminate states that are structurally disconnected from the inputs or outputs using `sminreal`.

  Eliminating structurally disconnected states is a good first step in model reduction because the process is cheap and does not involve any numerical computation.

- Compute a low-order approximation of your model using `balred`.

- Eliminate cancelling pole/zero pairs using `minreal`.

Example: Gasifier Model

This example presents a model of a gasifier, a device that converts solid materials into gases. The original model is nonlinear.

Loading the Model

To load a linearized version of the model, type

```
load ltiexamples
```

at the MATLAB prompt; the gasifier example is stored in the variable named `gasf`. If you type

```
size(gasf)
```

you get in return

```
State-space model with 4 outputs, 6 inputs, and 25 states.
```

SISO Model Order Reduction

You can reduce the order of a single I/O pair to understand how the model reduction tools work before attempting to reduce the full MIMO model as described in "MIMO Model Order Reduction" on page 2-30.

This example focuses on a single input/output pair of the gasifier, input 5 to output 3.
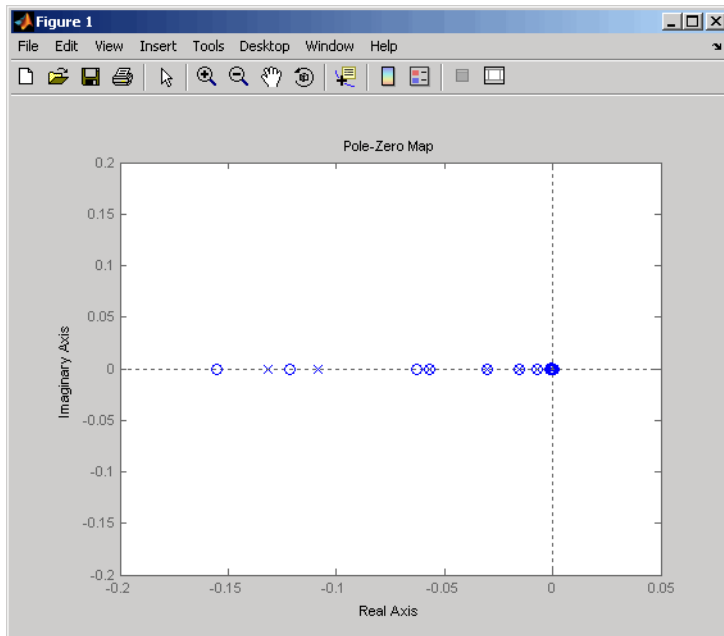
```
sys35 = gasf(3,5);
```

Before attempting model order reduction, inspect the pole and zero locations by typing

```
pzmap(sys35)
```

Zoom in near the origin on the resulting plot using the zoom feature or by typing

```
axis([-0.2 0.05 -0.2 0.2])
```

The following figure shows the results.

Pole-Zero Map of the Gasifier Model (Zoomed In)

Because the model displays near pole-zero cancellations, it is a good candidate for model reduction.
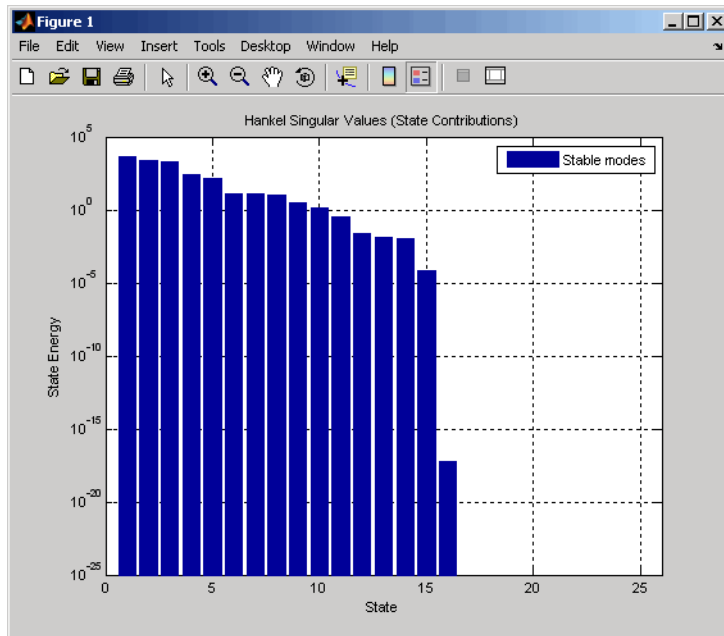
To find a low-order reduction of this SISO model, perform the following steps:

1   Select an appropriate order for your reduced system by examining the relative amount of energy per state using a Hankel singular value (HSV) plot. Type the command

```
hsvd(sys35)
```

to create the HSV plot.

Changing to log scale using the right-click menu results in the following plot.

2-27

Small Hankel singular values indicate that the associated states contribute little to the I/O behavior. This plot shows that discarding the last 10 states (associated with the 10 smallest Hankel singular values) has little impact on the approximation error.

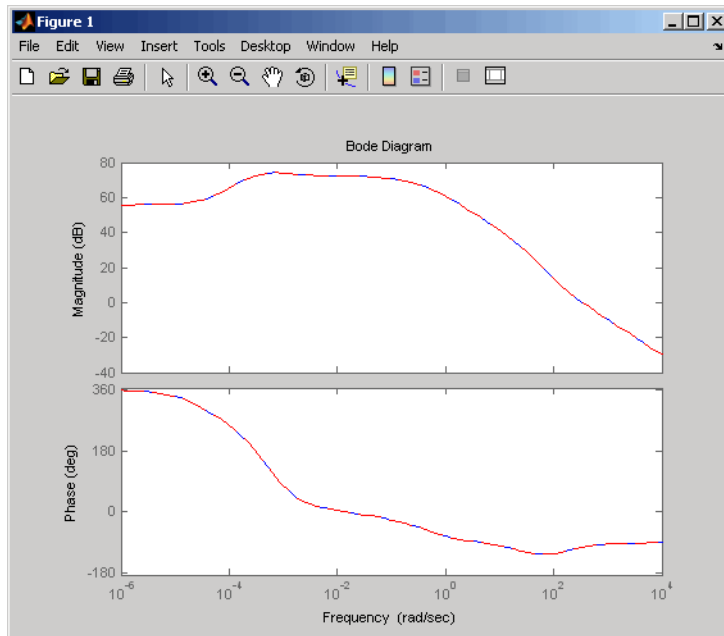2   To remove the last 10 states and create a 15th order approximation, type

```
rsys35 = balred(sys35,15);
```

You can type `size(rsys35)` to see that your reduced system contains only 15 states.

3   Compare the Bode response of the full-order and reduced-order models using the bode command:

```
bode(sys35,'b',rsys35,'r--')
```

This figure shows the result.

As the overlap of the curves in the figure shows, the reduced model provides a good approximation of the original system.

You can try reducing the model order further by repeating this process and removing more states. Reduce the `gasf` model to 5th, 10th, and 15th orders all at once by typing the following command

```
rsys35 = balred(sys35,[5 10 15]);
```

Plot a bode diagram of these three reduced systems along with the full order system by typing

```
bode(sys35,'b',rsys35,'r--')
```
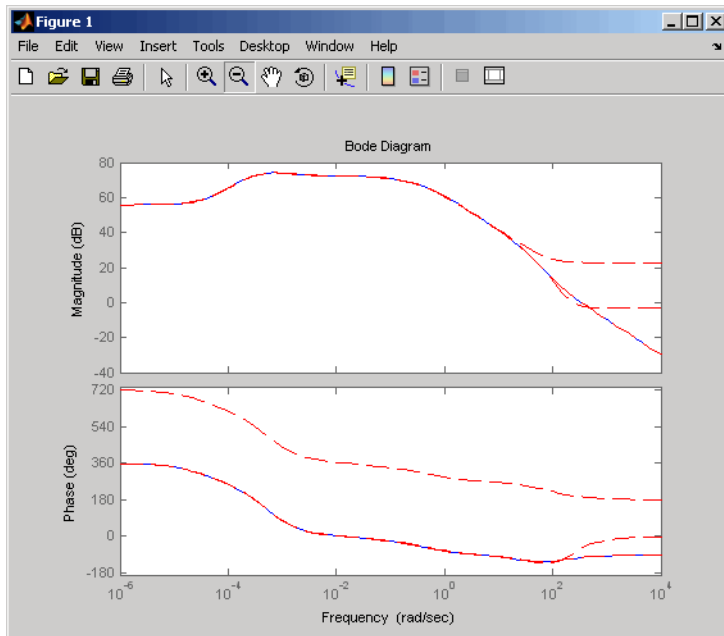
Observe how the error increases as the order decreases.

MIMO Model Order Reduction

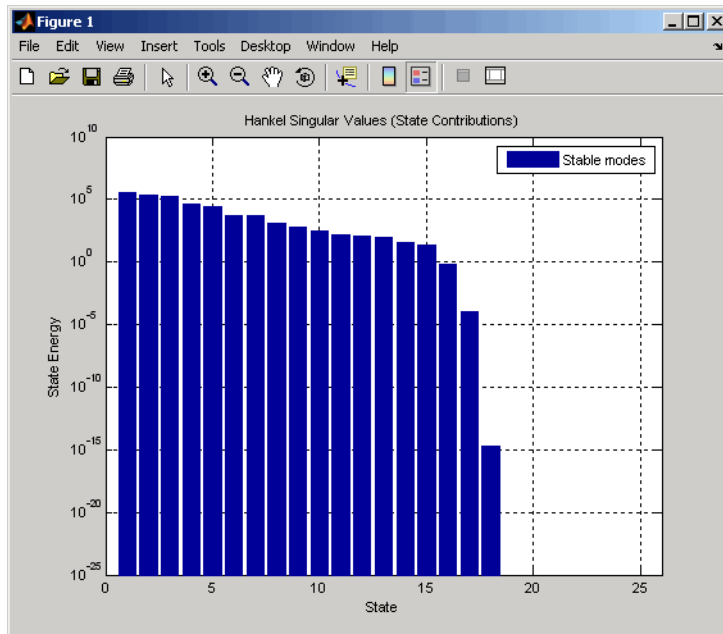You can approximate MIMO models using the same steps as SISO models as follows:

1 Select an appropriate order for your reduced system by examining the relative amount of energy per state using a Hankel singular value (HSV) plot.

Type

```
hsvd(gasf)
```

to create the HSV plot.

Discarding the last 8 states (associated with the 8 smallest Hankel singular values) should have little impact on the error in the resulting 17th order system.

2 To remove the last 8 states and create a 17th order MIMO system, type
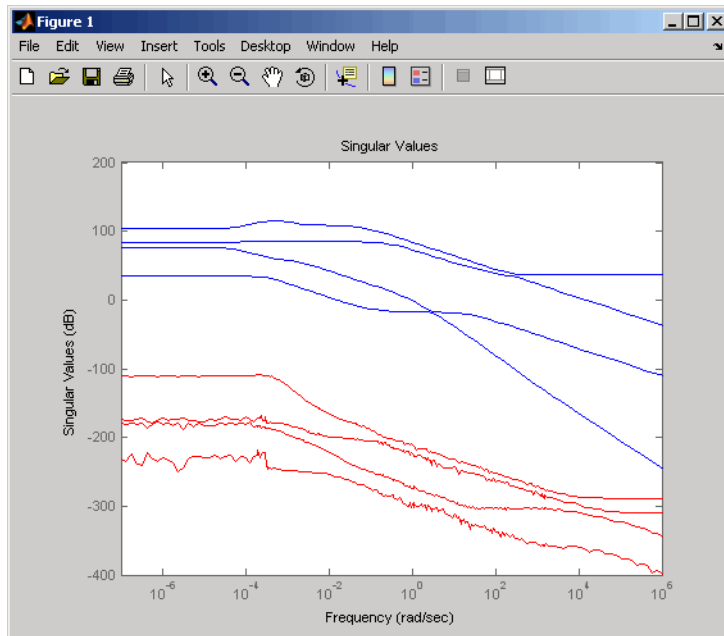
```
rsys=balred(gasf,17);
```

You can type `size(gasf)` to see that your reduced system contains only 17 states.

3 To facilitate visual inspection of the approximation error, use a singular value plot rather than a MIMO Bode plot. Type

```
sigma(gasf,'b',gasf-rsys,'r')
```

to create a singular value plot comparing the original system to the reduction error.

The reduction error is small compared to the original system so the reduced order model provides a good approximation of the original model.

Acknowledgment

MathWorks thanks ALSTOM® Power UK for permitting use of their gasifier model for this example. This model was issued as part of the ALSTOM Benchmark Challenge on Gasifier Control. For more details see Dixon, R., (1999), "Advanced Gasifier Control," *Computing & Control Engineering Journal*, IEE, Vol. 10, No. 3, pp. 92–96.

3

Analyzing Models

- "Linear Analysis Using the Linear System Analyzer" on page 3-2
- "Simulate Models with Arbitrary Inputs and Initial Conditions" on page 3-7

Linear Analysis Using the Linear System Analyzer

In this example, you learn how to analyze the time-domain and frequency-domain responses of one or more linear models using the Linear System Analyzer app.

Before you can perform the analysis, you must have already created linear models in the MATLAB workspace. For information on how to create a model, see "Basic Models".

To perform linear analysis:

1   Open the Linear System Analyzer showing one or more models using the following syntax:
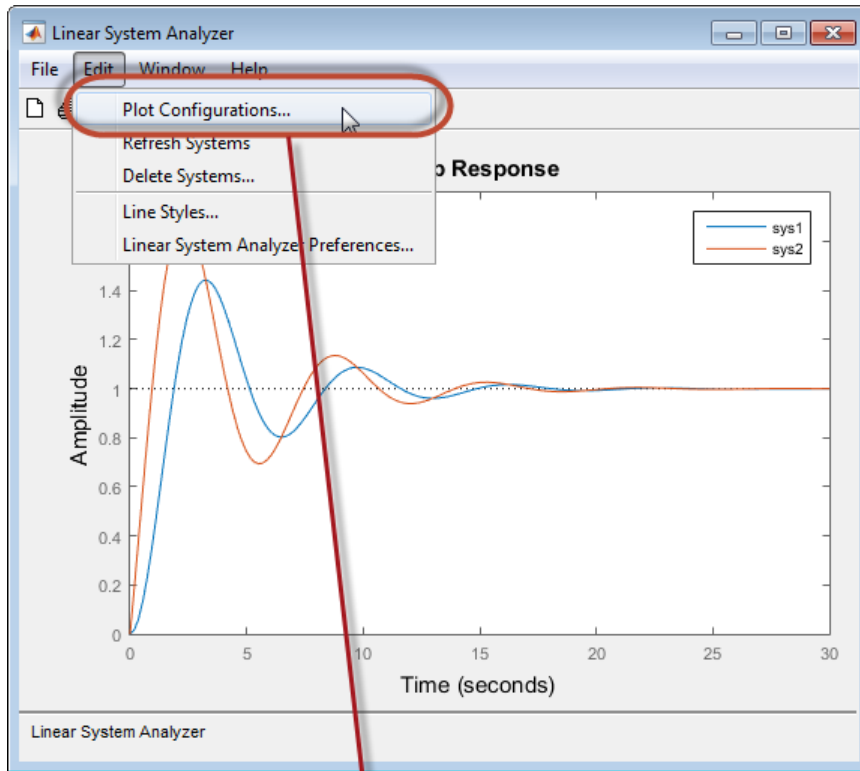
```
linearSystemAnalyzer(sys1,sys2,...,sysN)
```

By default, this syntax opens a step response plot of your models, as shown in the following figure.
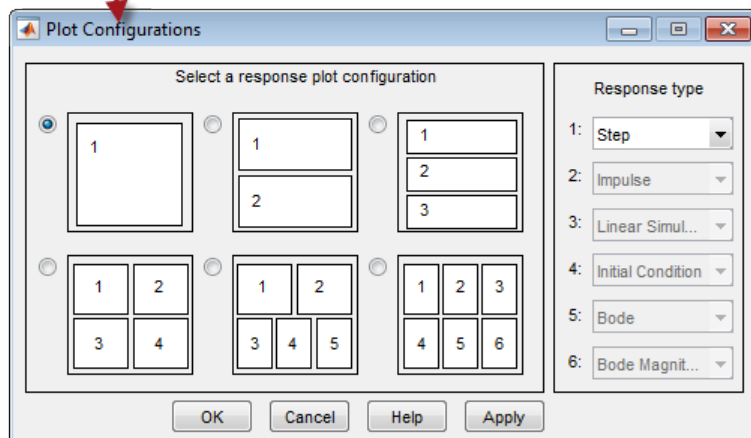
click to add a legend
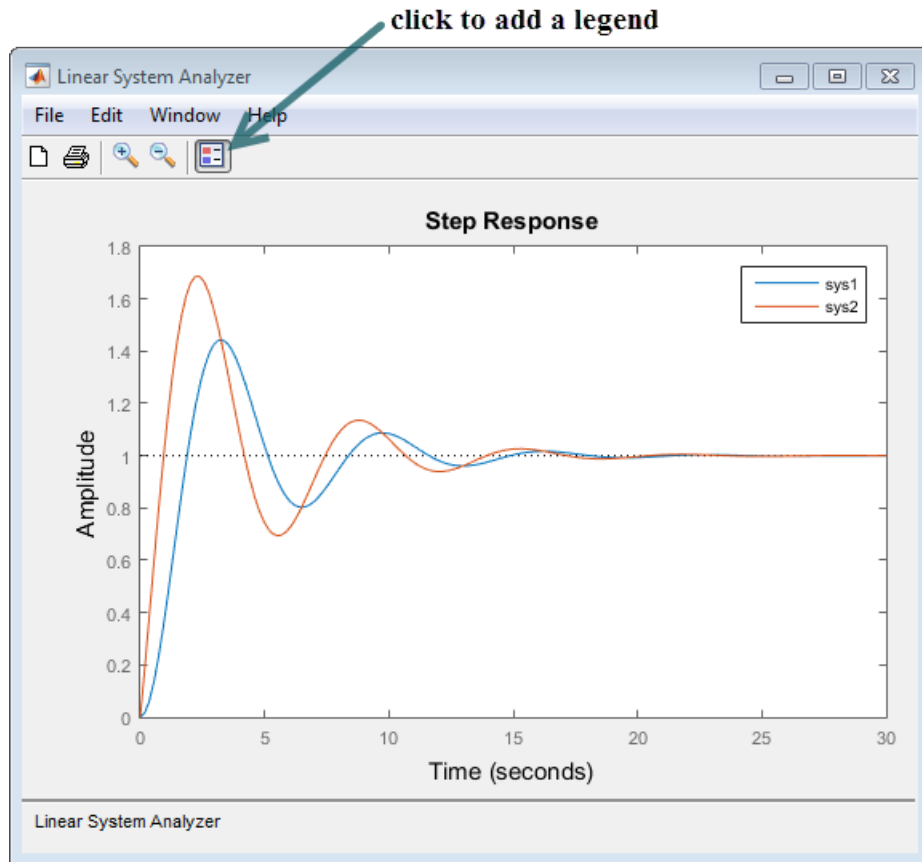
Note Alternatively, open Linear System Analyzer from the **Apps** tab in the MATLAB desktop. When you do so, select **File > Import** to load linear models from the MATLAB workspace or a MAT file.
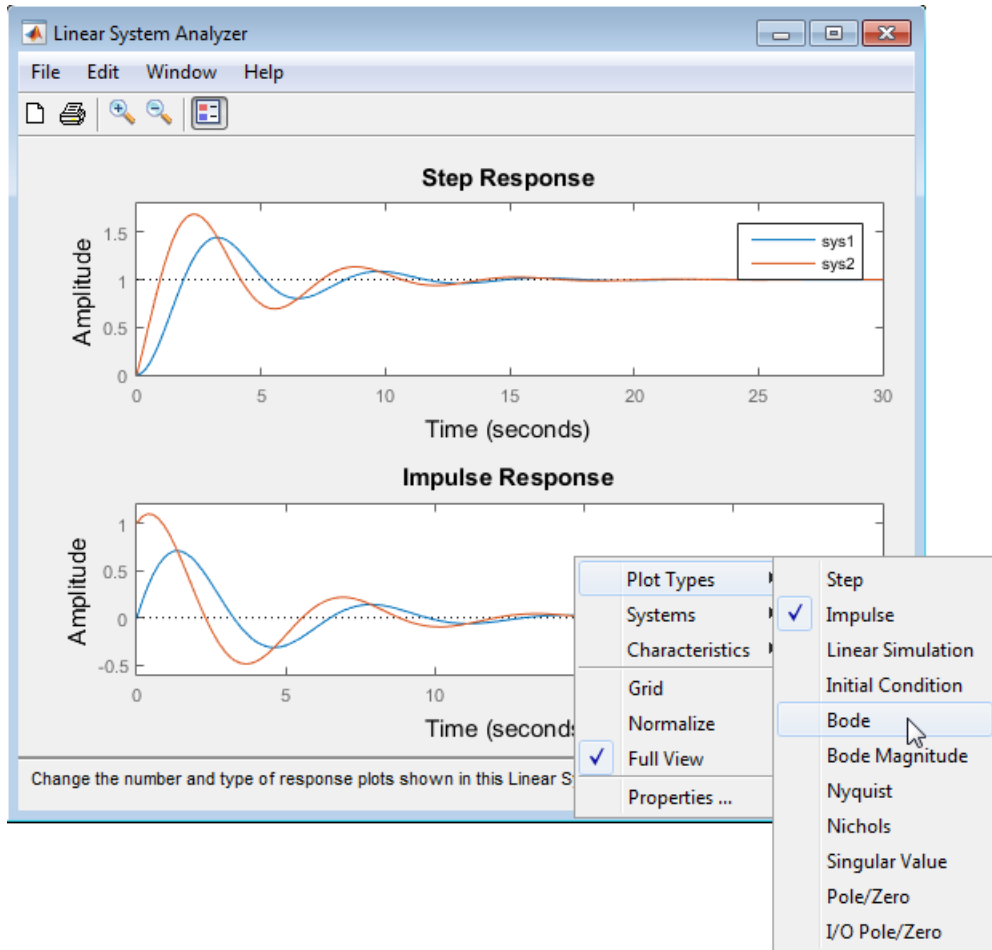
2 Add more plots to the Linear System Analyzer.

a Select **Edit > Plot Configurations**.

b In the Plot Configurations dialog box, select the number of plots to open.
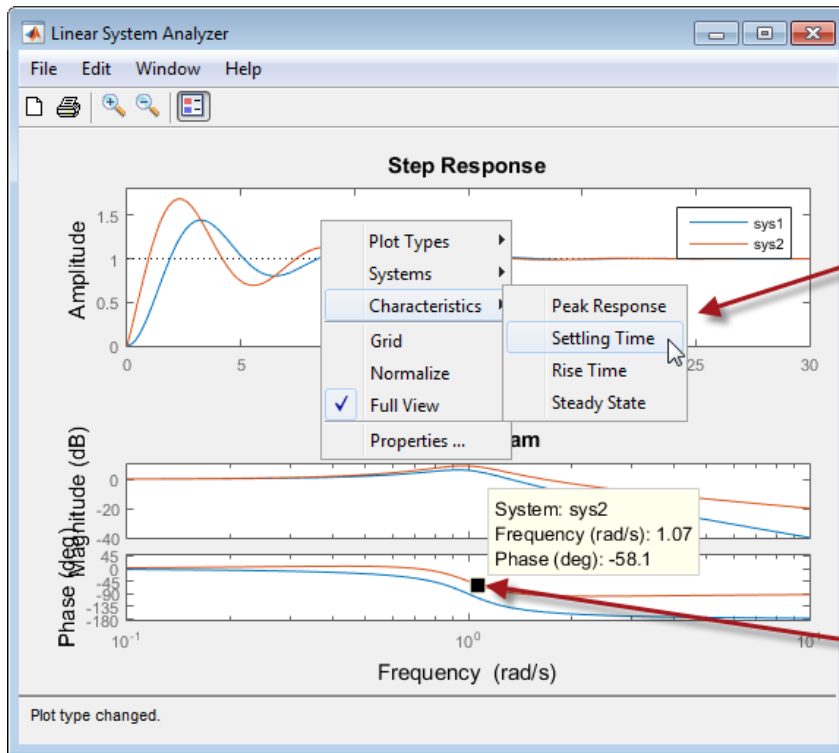
3 To view a different type of response on a plot, right-click and select a plot type.

4 Analyze system performance. For example, you can analyze the peak response in the Bode plot and settling time in the step response plot.

    a Right-click to select performance characteristics.

    b Click on the dot that appears to view the characteristic value.

See Also
Linear System Analyzer | `lsim`

Related Examples

- "Joint Time-Domain and Frequency-Domain Analysis"

More About

- "Linear System Analyzer Overview"

Simulate Models with Arbitrary Inputs and Initial Conditions

What is the Linear Simulation Tool?

You can use the Linear Simulation Tool to simulate linear models with arbitrary input signals and initial conditions.

The Linear Simulation Tool lets you do the following:

- Import input signals from the MATLAB workspace.
- Import input signals from a MAT-file, Microsoft® Excel® spreadsheet, ASCII flat-file, comma-separated variable file (CSV), or text file.
- Generate arbitrary input signals in the form of a sine wave, square wave, step function, or white noise.
- Specify initial states for state-space models.

  Default initial states are zero.

Opening the Linear Simulation Tool

To open the Linear Simulation Tool, do one of the following:

- In the Linear System Analyzer, right-click the plot area and select **Plot Types > Linear Simulation**.
- Use the `lsim` function at the MATLAB prompt:
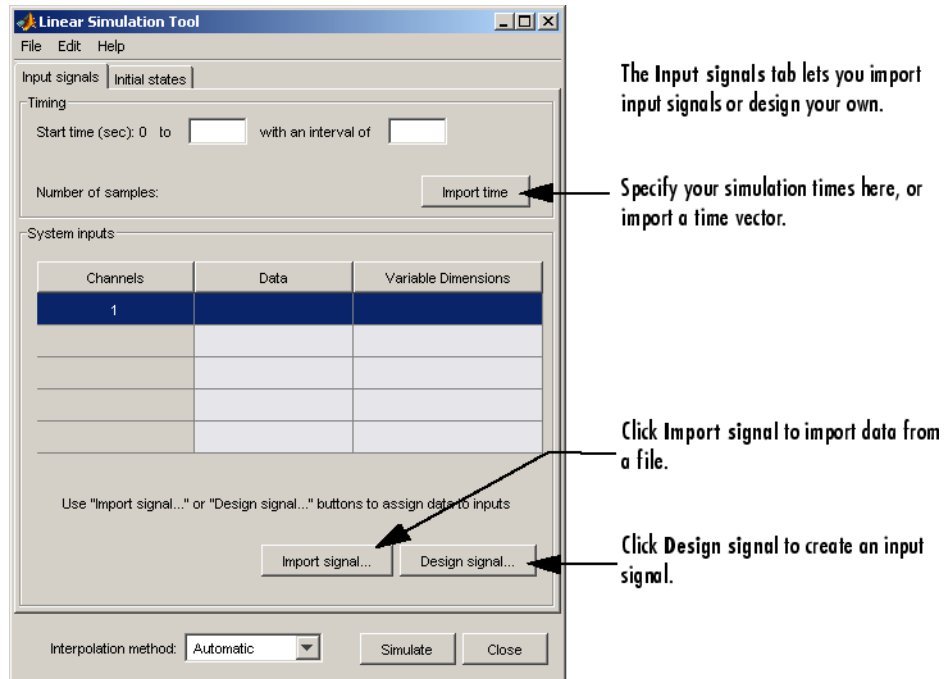
  `lsim(modelname)`
- In the MATLAB Figure window, right-click a response plot and select **Input data**.

Working with the Linear Simulation Tool

The Linear Simulation Tool contains two tabs, **Input signals** and **Initial states**.

After opening the Linear Simulation Tool (as described in "Opening the Linear Simulation Tool" on page 3-7), follow these steps to simulate your model:

1 Click the **Input signals** tab, if it is not displayed.

The Input signals tab lets you import input signals or design your own.

Specify your simulation times here, or import a time vector.

Click **Import signal** to import data from a file.

Click **Design signal** to create an input signal.

2 In the **Timing** area, specify the simulation time vector by doing one of the following:

- Import the time vector by clicking **Import time**.
- Enter the end time and the time interval in seconds. The start time is set to 0 seconds.

3 Specify the input signal by doing one of the following:

- Click **Import signal** to import it from the MATLAB workspace or a file. For more information, see "Importing Input Signals" on page 3-10.
- Click **Design signal** to create your own inputs. For more information, see "Designing Input Signals" on page 3-16.

4 If you have a state-space model and want to specify initial conditions, click the **Initial states** tab. By default, all initial states are set to zero.

You can either enter state values in the **Initial value** column, or import values by clicking **Import state vector**. For more information about entering initial states, see "Specifying Initial Conditions" on page 3-18.
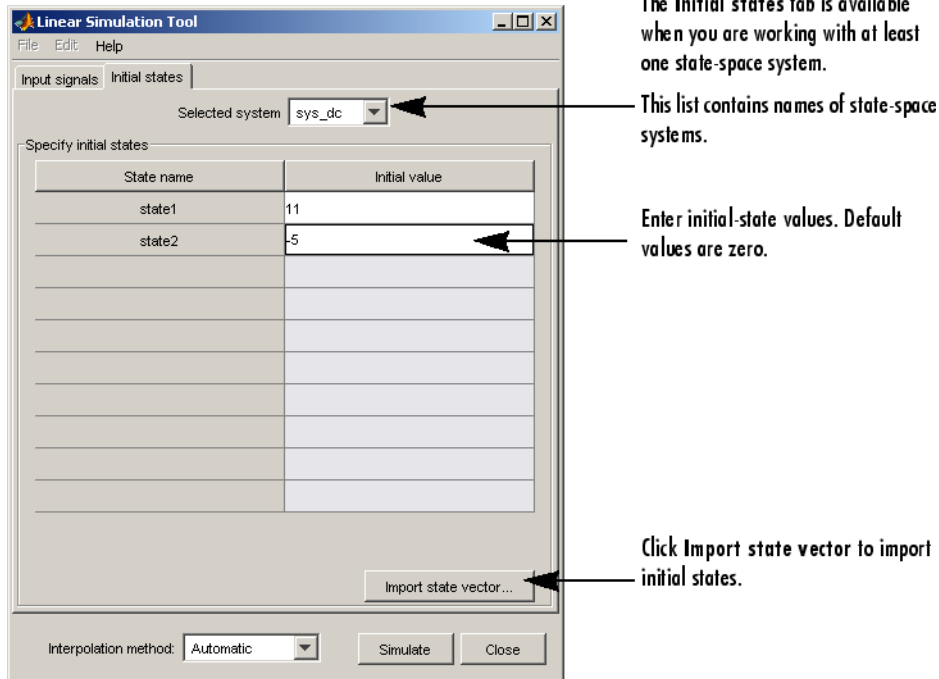
5 For a continuous model, select one of the following interpolation methods in the **Interpolation method** list to be used by the simulation solver:

- Zero order hold
- First order hold (linear interpolation)
- Automatic (Linear Simulation Tool selects first order hold or zero order hold automatically, based on the smoothness of the input)

Note The interpolation method is not used when simulating discrete models.

6 Click **Simulate**.

Importing Input Signals

You can import input signals from the MATLAB workspace after opening the Linear Simulation Tool (see "Opening the Linear Simulation Tool" on page 3-7). You can also import inputs from a MAT-file, Microsoft Excel spreadsheet, ASCII flat-file, comma-separated variable file (CSV), or text file.

For information about creating your own inputs, see "Designing Input Signals" on page 3-16. For an overview of working with the Linear Simulation Tool, see "Working with the Linear Simulation Tool" on page 3-7.

To import one or more input signals:

1 In the Linear Simulation Tool, click the **Input signals** tab, if it is not displayed.

2 Specify the simulation time in the **Timing** area.

3 Select one or more rows for the input channels you want to import. The following figure shows an example with two selected channels.
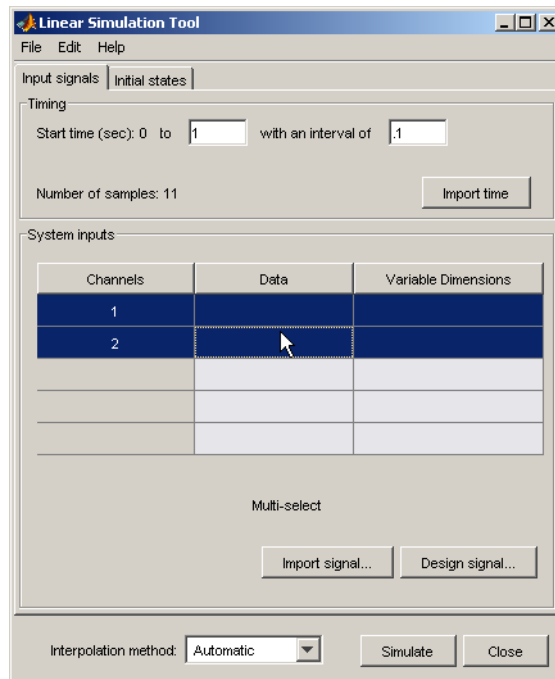
4 Click **Import signal** to open the Data Import dialog box. The following figure shows an example of the Data Import dialog box.
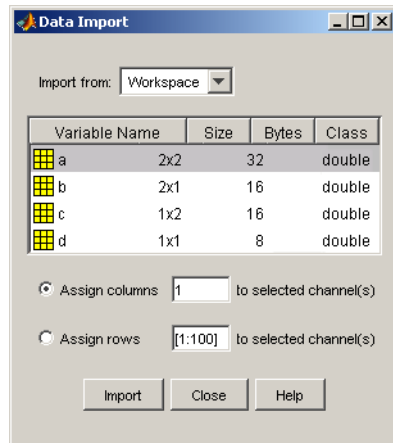
5 In the **Import from** list, select the source of the input signals. It can be one of the following:

- `Workspace`
- `MAT file`
- `XLS file`
- `CSV file`
- `ASCII file`

6 Select the data you want to import. The Data Import dialog box contains different options depending on which source format you selected.

7 Click **Import**.

For an example of importing input signals, see the following:

- "Example: Loading Inputs from a Microsoft Excel Spreadsheet" on page 3-11
- "Example: Importing Inputs from the Workspace" on page 3-12

Example: Loading Inputs from a Microsoft Excel Spreadsheet

To load inputs from a Microsoft Excel (XLS) spreadsheet:

1. In the Linear Simulation Tool, click **Import signal** in the **Input signals** tab to open the Data Import dialog box.

2. Select **XLS file** in the **Import from** list.

3. Click **Browse**.

4. Select the file you want to import and click **Open**. This populates the Data Import dialog box with the data from the Microsoft Excel spreadsheet.
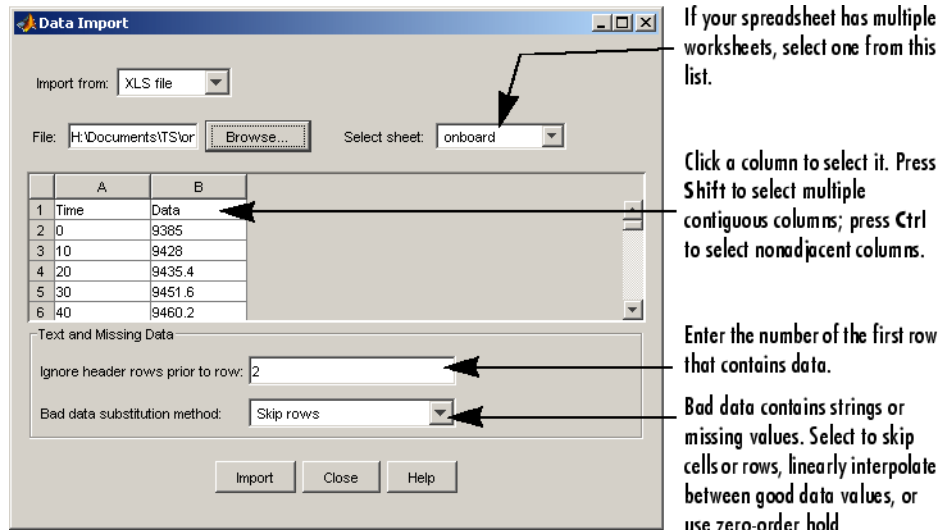
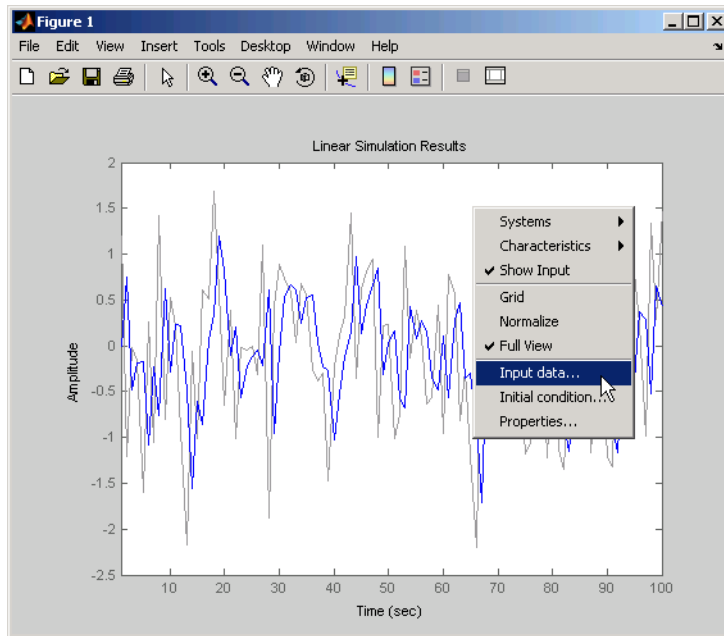

Example: Importing Inputs from the Workspace
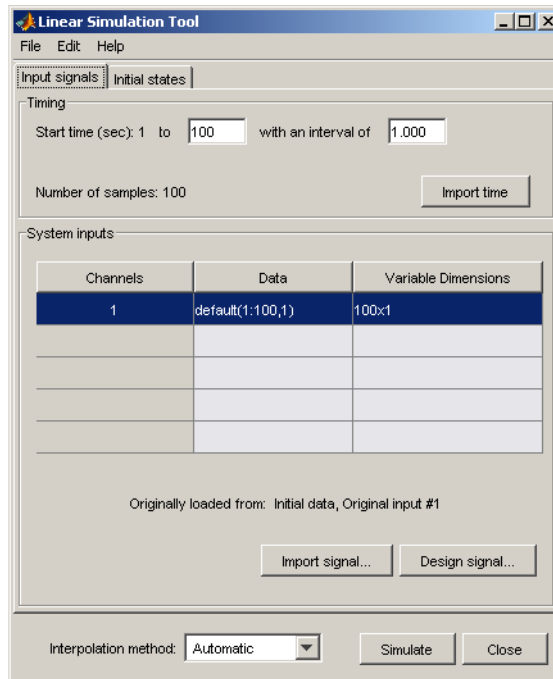
To load an input signal from the MATLAB workspace:

1. Enter this code to open a response plot with a second-order system:

```
s=tf('s');
ss=(s+2)/(s^2+3*s+2);
lsim(ss,randn(100,1),1:100);
```

2. Right-click the plot background and select **Input data**.

This opens the Linear Simulation Tool with default input data.

3   Create an input signal for your system in the MATLAB Command Window, such as the following:

```
new_signal=[-3*ones(1,20) 2*ones(1,30) 0.5*ones(1,50)]';
```

4   In the Linear Simulation Tool, click **Import signal**.

5   In the Data Import dialog box, click, **Assign columns** to assign the first column of the input signal to the selected channel.
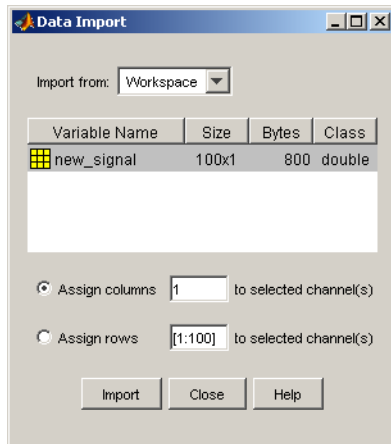
6 Click **Import**. This imports the new signal into the Linear Simulation Tool.
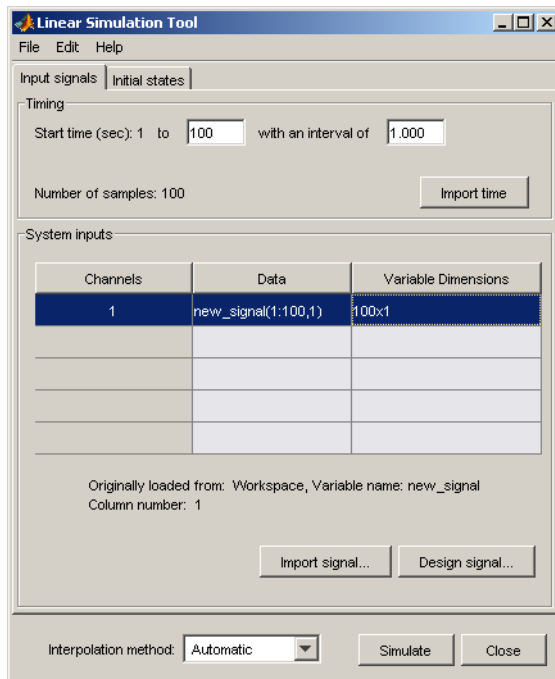
7 Click **Simulate** to see the response of your second-order system to the imported signal.

Designing Input Signals

You can generate arbitrary input signals in the form of a sine wave, square wave, step function, or white noise after opening the Linear Simulation Tool (see "Opening the Linear Simulation Tool" on page 3-7).
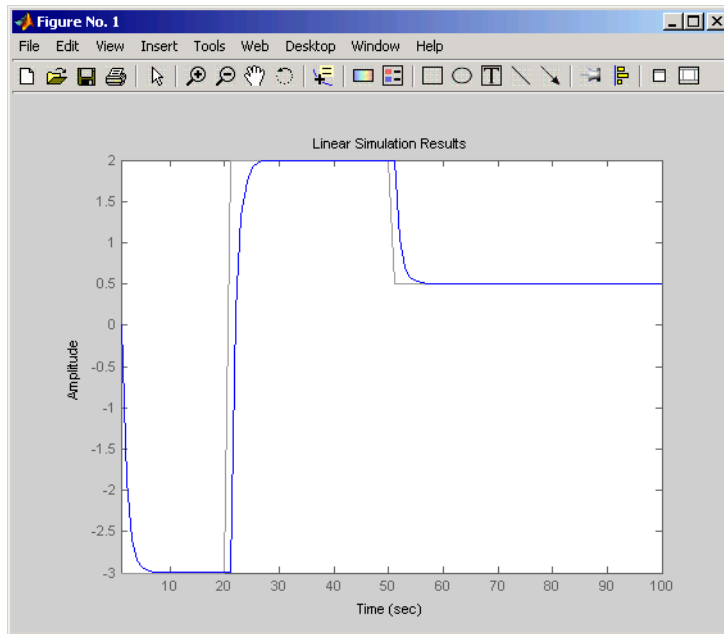
For information about importing inputs from the MATLAB workspace or from a file, see "Importing Input Signals" on page 3-10. For an overview of working with the Linear Simulation Tool, see "Working with the Linear Simulation Tool" on page 3-7.

To design one or more input signals:

1 In the Linear Simulation Tool, click the **Input signals** tab (if it is not displayed).

2 Specify the simulation time in the **Timing** area. The time interval (in seconds) is used to evaluate the input signal you design in later steps of this procedure.

3 Select one or more rows for the signal channels you want to design. The following figure shows an example with two selected channels.
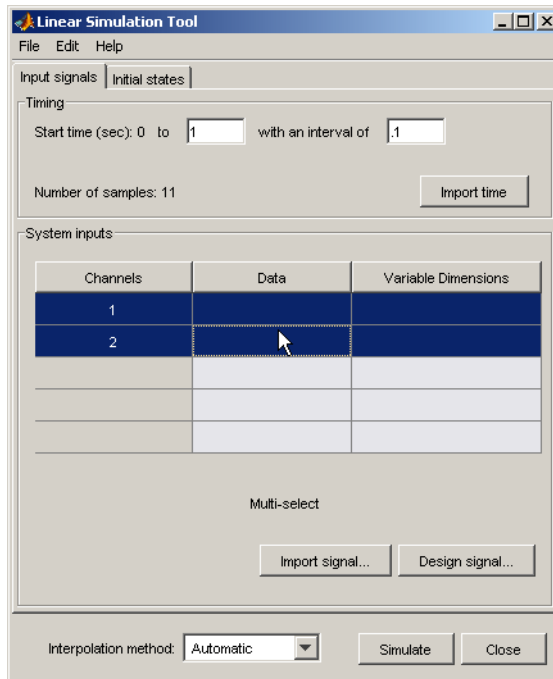
4 Click **Design signal** to open the Signal Designer dialog box. The following figure shows an example of the Signal Designer dialog box.
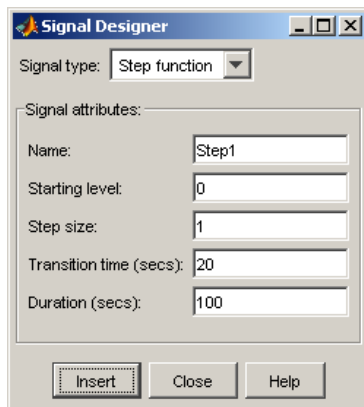
5 In the **Signal type** list, select the type of signal you want to create. It can be one of the following:

- Sine wave
- Square wave
- Step function
- White noise

6 Specify the signal characteristics. The Signal Designer dialog box contains different options depending on which signal type you selected.

7 Click **Insert**. This brings the new signal into the Linear Simulation Tool.

8 Click **Simulate** in the Linear Simulation Tool to view the system response.

Specifying Initial Conditions

If your system is in state-space form, you can enter or import initial states after opening the Linear Simulation Tool (see "Opening the Linear Simulation Tool" on page 3-7).

For an overview of working with the Linear Simulation Tool, see "Working with the Linear Simulation Tool" on page 3-7.

You can also import initial states from the MATLAB workspace.

To import one or more initial states:

1 In the Linear Simulation Tool, click the **Initial states** tab (if it is not already displayed).

2 In the **Selected system** list, select the system for which you want to specify initial conditions.

3 You can either enter state values in the **Initial value** column, or import values from the MATLAB workspace by clicking **Import state vector**. The following figure shows an example of the import window:
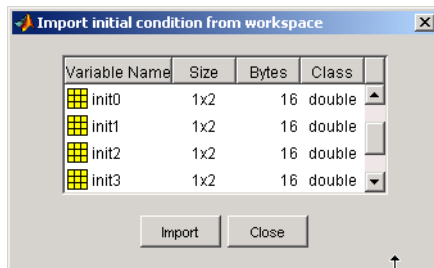
> **Note** For *n*-states, your initial-condition vector must have *n* entries.

4 After specifying the initial states, click **Simulate** in the Linear Simulation Tool to view the system response.

See Also

Linear System Analyzer | `lsim`

Related Examples

- "Joint Time-Domain and Frequency-Domain Analysis"
- "Response from Initial Conditions"

Designing Compensators

Choosing a PID Controller Design Tool

Control System Toolbox software provides several tools for designing PID controllers for plants represented by LTI models.

The following table summarizes these tools and when to use them. For information about tuning PID controllers in Simulink® models, see "Model-Based PID Controller Tuning" (Simulink Control Design).

Tool	When to use
PID Tuner on page 4-4	• Automatic, interactive tuning of SISO PID controller in the feed-forward path of single-loop, unity-feedback control configuration. • Automatic, interactive tuning of 2-DOF PID controller in the loop configuration of this illustration: • Interactive fitting of a plant model from measured SISO response data and automatic tuning of PID controller for the resulting model (requires System Identification Toolbox™ software).
Control System Designer	Tuning PID controllers in any other loop configuration.

Tool	When to use
Command-line PID tuning on page 4-18	Programmatic tuning of SISO PID controllers.

Designing PID Controllers with PID Tuner

In Control System Toolbox, **PID Tuner** lets you perform automatic, interactive tuning of PID controllers for plants represented by LTI models.

For information about using **PID Tuner** to tune a PID Controller block in a Simulink model, see "Introduction to Model-Based PID Tuning in Simulink" (Simulink Control Design).

PID Tuner Overview

Use **PID Tuner** to interactively design a SISO PID controller in the feed-forward path of single-loop, unity-feedback control configuration.

PID Tuner automatically designs a controller for your plant. You specify the controller type (P, I, PI, PD, PDF, PID, PIDF) and form (parallel or standard).

You can also use **PID Tuner** to design a 2-DOF PID controller for the feedback configuration of this illustration:
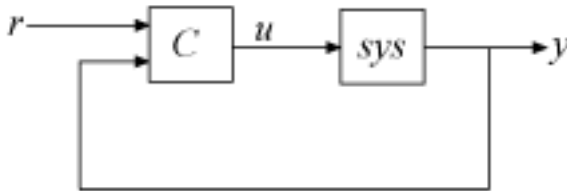
PID Tuner can design 2-DOF PID controllers in which the setpoint weights can be free and tunable parameters. **PID Tuner** can also design controllers in which the setpoint weights are fixed in common control configurations, such as I-PD and PI-D.

You can analyze the design using a variety of response plots, and interactively adjust the design to meet your performance requirements.

To launch **PID Tuner**, use the `pidTuner` command:

```
pidTuner(sys,type)
```

where `sys` is a linear model of the plant you want to control, and `type` is indicates the controller type on page 4-5 to design.

Alternatively, enter

```
pidTuner(sys,Cbase)
```

where `Cbase` is a baseline controller, allowing you to compare the performance of the designed controller to the performance of `Cbase`.

For more information about `sys` and `Cbase`, see the `pidTuner` reference page.

Note You can also open **PID Tuner** from the MATLAB desktop, in the **Apps** tab. When you do so, use the **Plant** menu in **PID Tuner** to specify your plant model.

PID Controller Type

To select the controller type, use one of these methods:

- Provide the `type` argument to the `pidTuner` command when you open **PID Tuner**. For example, `pidTuner(G,'PIDF2')` opens **PID Tuner** with an initial design that is a 2-DOF PID controller with a filter on the derivative term.
- Provide the baseline-controller `Cbase` argument to the `pidTuner` command when you open **PID Tuner**. **PID Tuner** designs a controller of the same type as `Cbase`. For example, suppose `C0` is a `pid` controller object that has proportional and derivative action only (PD controller). Then, `pidTuner(G,C0)` opens **PID Tuner** with an initial design that is a PD controller.
- In **PID Tuner**, use the **Type** menu to change controller types.
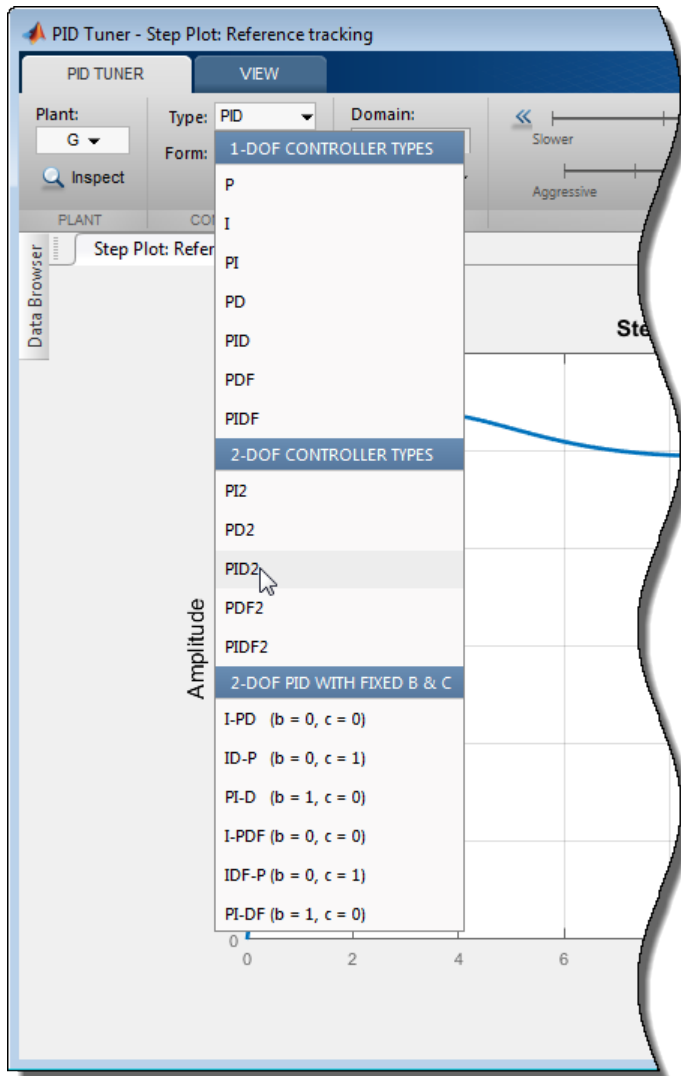
The following tables summarize the available PID controller types. For more information about these controller types, see "PID Controller Types for Tuning".

1-DOF Controllers

type input to pidTuner	Entry in Type menu	Controller Actions
'P'	P	Proportional only
'I'	I	Integral only
'PI'	PI	Proportional and integral
'PD'	PD	Proportional and derivative
'PDF'	PDF	Proportional and derivative with first-order filter on derivative term
'PID'	PID	Proportional, integral, and derivative
'PIDF'	PIDF	Proportional, integral, and derivative with first-order filter on derivative term

2-DOF Controllers

PID Tuner can automatically design 2-DOF PID controller types with free setpoint weights. The following table summarizes the 2-DOF controller types in **PID Tuner**. For more information about 2-DOF PID controllers generally, see "Two-Degree-of-Freedom PID Controllers".

type input to pidTuner	Entry in Type menu	Controller Actions
'PI2'	PI2	2-DOF proportional and integral
'PD2'	PD2	2-DOF proportional and derivative
'PDF2'	PDF2	2-DOF proportional and derivative with first-order filter on derivative term
'PID2'	PID2	2-DOF proportional, integral, and derivative
'PIDF2'	PIDF2	2-DOF proportional, integral, and derivative with first-order filter on derivative term

2-DOF Controllers with Fixed Setpoint Weights

Use **PID Tuner** to design the fixed-setpoint-weight controller types summarized in the following table. For more information about these controller types, see "PID Controller Types for Tuning".

type input to pidTuner	Entry in Type menu	Controller Actions
'I-PD'	I-PD	2-DOF PID with $b = 0$, $c = 0$
'I-PDF'	I-PDF	2-DOF PIDF with $b = 0$, $c = 0$
'ID-P'	ID-P	2-DOF PID with $b = 0$, $c = 1$
'IDF-P'	IDF-P	2-DOF PIDF with $b = 0$, $c = 1$
'PI-D'	PI-D	2-DOF PID with $b = 1$, $c = 0$
'PI-DF'	PI-DF	2-DOF PIDF with $b = 1$, $c = 0$

Discrete-Time Controller Types

If sys is a discrete-time model with sample time Ts, **PID Tuner** designs a discrete-time pid controller using the ForwardEuler discrete integrator formula. To design a controller that has different discrete integrator formulas, use one of the following methods:

- Provide a discrete-time baseline controller Cbase to the launch command pidTuner. **PID Tuner** designs a controller that has the same discrete integrator formulas as Cbase.

- After launching **PID Tuner**, click **Options** to open the **Controller Options** dialog box. Select discrete integrator formulas from the **Integral Formula** and **Derivative Formula** menus.

For more information about discrete integrator formulas, see the pid, pid2, pidstd, and pidstd2 reference pages.

PID Controller Form

When you use the type input to pidTuner, **PID Tuner** designs a controller in parallel form. To design a controller in standard form, use one of the following methods:

- Provide a standard-form baseline controller Cbase to the launch command pidTuner. **PID Tuner** designs a controller of the same form as Cbase.

- Use the **Form** menu to change controller form after launching **PID Tuner**.

For more information about parallel and standard controller forms, see the `pid`, `pid2`, `pidstd`, and `pidstd2` reference pages.

See Also

More About

- "Proportional-Integral-Derivative (PID) Controllers"
- "Two-Degree-of-Freedom PID Controllers"
- "Analyze Design in PID Tuner" on page 4-10
- "PID Controller Design for Fast Reference Tracking" on page 4-18

Analyze Design in PID Tuner

In Control System Toolbox, **PID Tuner** provides system response plots and other tools for tuning PID controllers for plants represented by LTI models.

For information about analysis in **PID Tuner** with Simulink models, see "Analyze Design in PID Tuner" (Simulink Control Design).

Plot System Responses

To determine whether the compensator design meets your requirements, you can analyze the system response using the response plots. On the **PID Tuner** tab, select a response plot from the **Add Plot** menu. The **Add Plot** menu also lets you choose from several step plots (time-domain response) or Bode plots (frequency-domain response).
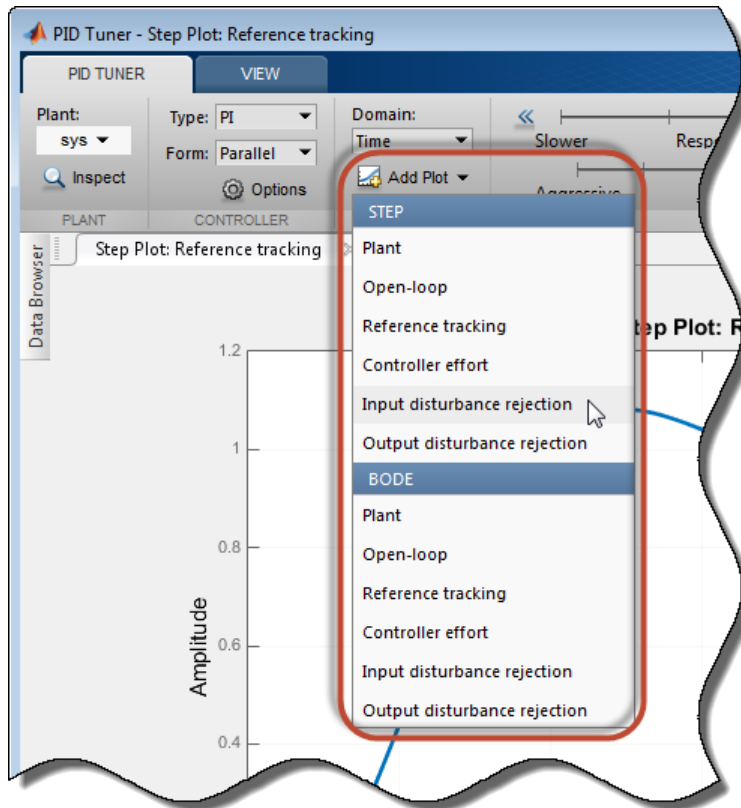
For 1-DOF PID controller types such as PI, PIDF, and PDF, **PID Tuner** computes system responses based upon the following single-loop control architecture:
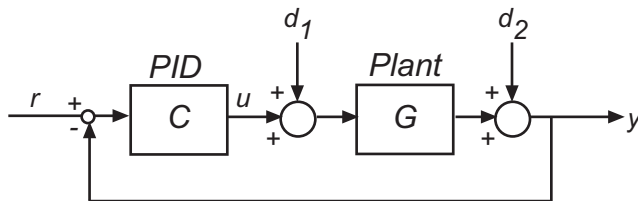
For 2-DOF PID controller types such as PI2, PIDF2, and I-PD, **PID Tuner** computes responses based upon the following architecture:
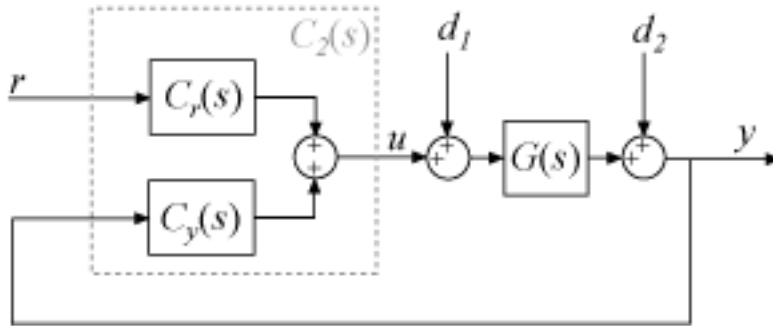
The system responses are based on the decomposition of the 2-DOF PID controller, C_2, into a setpoint component Cr and a feedback component Cy, as described in "Two-Degree-of-Freedom PID Controllers".

The following table summarizes the available responses for analysis plots in **PID Tuner**.

Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
Plant	G	G	Shows the plant response. Use to examine plant dynamics.
Open-loop	GC	$-GC_y$	Shows response of the open-loop controller-plant system. Use for frequency-domain design. Use when your design specifications include robustness criteria such as open-loop gain margin and phase margin.

Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
Reference tracking	$\dfrac{GC}{1+GC}$ (from r to y)	$\dfrac{GC_r}{1-GC_y}$ (from r to y)	Shows the closed-loop system response to a step change in setpoint. Use when your design specifications include setpoint tracking.
Controller effort	$\dfrac{C}{1+GC}$ (from r to u)	$\dfrac{C_r}{1-GC_y}$ (from r to u)	Shows the closed-loop controller output response to a step change in setpoint. Use when your design is limited by practical constraints, such as controller saturation.
Input disturbance rejection	$\dfrac{G}{1+GC}$ (from d_1 to y)	$\dfrac{G}{1-GC_y}$ (from d_1 to y)	Shows the closed-loop system response to load disturbance (a step disturbance at the plant input). Use when your design specifications include input disturbance rejection.
Output disturbance rejection	$\dfrac{1}{1+GC}$ (from d_2 to y)	$\dfrac{1}{1-GC_y}$ (from d_2 to y)	Shows the closed-loop system response to a step disturbance at plant output. Use when you want to analyze sensitivity to measurement noise.

Compare Tuned Response to Baseline Response

If you have defined a baseline controller, then by default **PID Tuner** displays both the responses using the current **PID Tuner** design and the responses using the baseline controller.
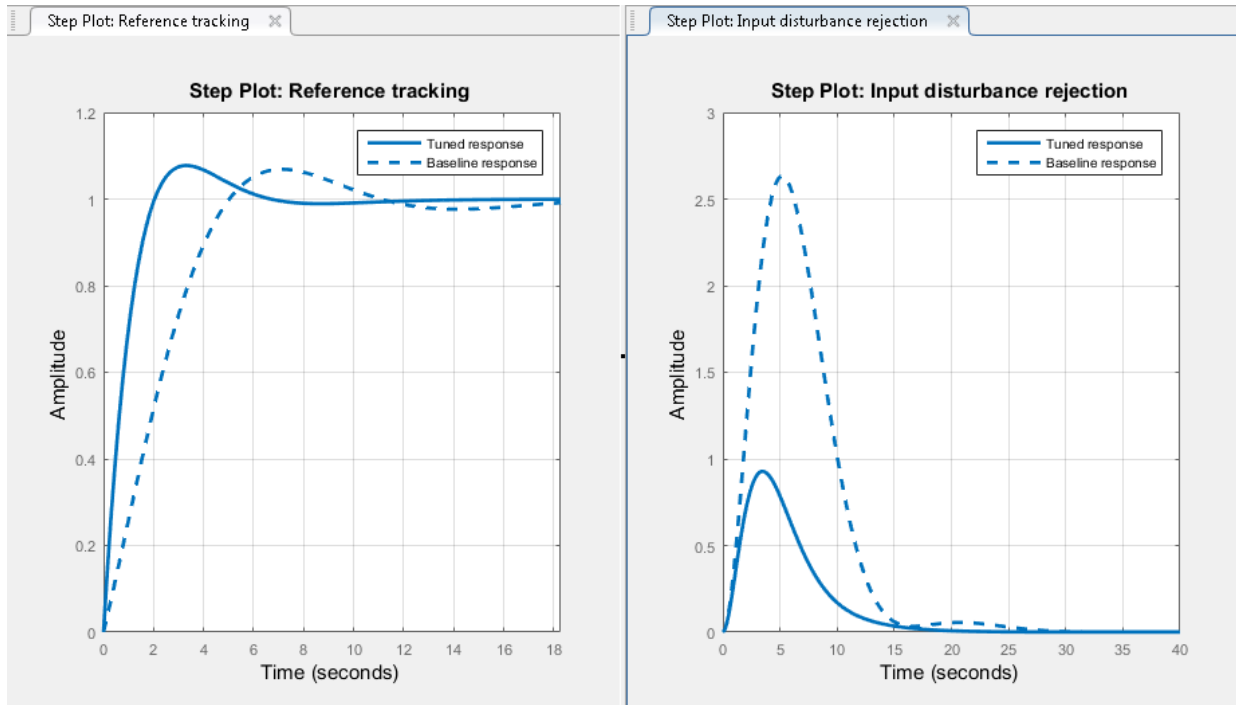
There are two ways to define a baseline controller:

- Load a baseline controller when you open **PID Tuner**, using the syntax `pidTuner(sys,C0)`.

- Make the current **PID Tuner** design the baseline controller at any time, by clicking the **Export** arrow ▾ and selecting `Save as Baseline`.
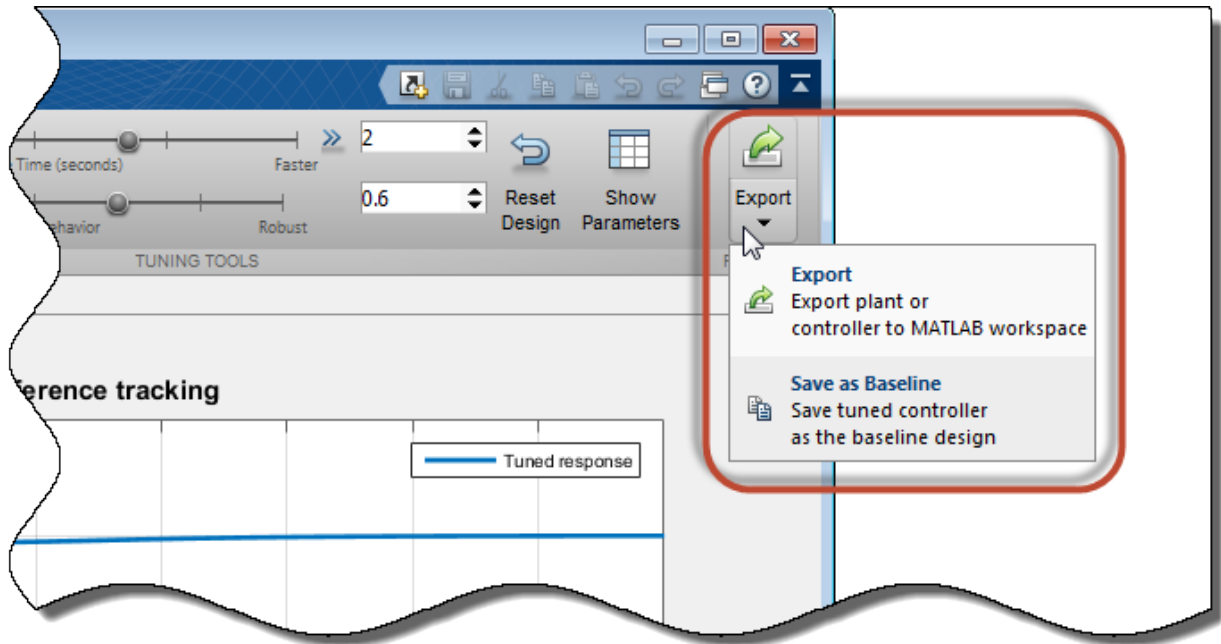
When you do so, the current Tuned response becomes the Baseline response. Further adjustment of the current design creates a new Tuned response line.

To hide the Baseline response, click ⚙ **Options**, and uncheck **Show Baseline Controller Data**.

View Numeric Values of System Characteristics

You can view the values for system characteristics, such as peak response and gain margin, either:

- Directly on the response plot — Use the right-click menu to add characteristics, which appear as blue markers. Then, left-click the marker to display the corresponding data panel.

- In the **Performance and robustness** table — To display this table, click ⊞ **Show Parameters**.
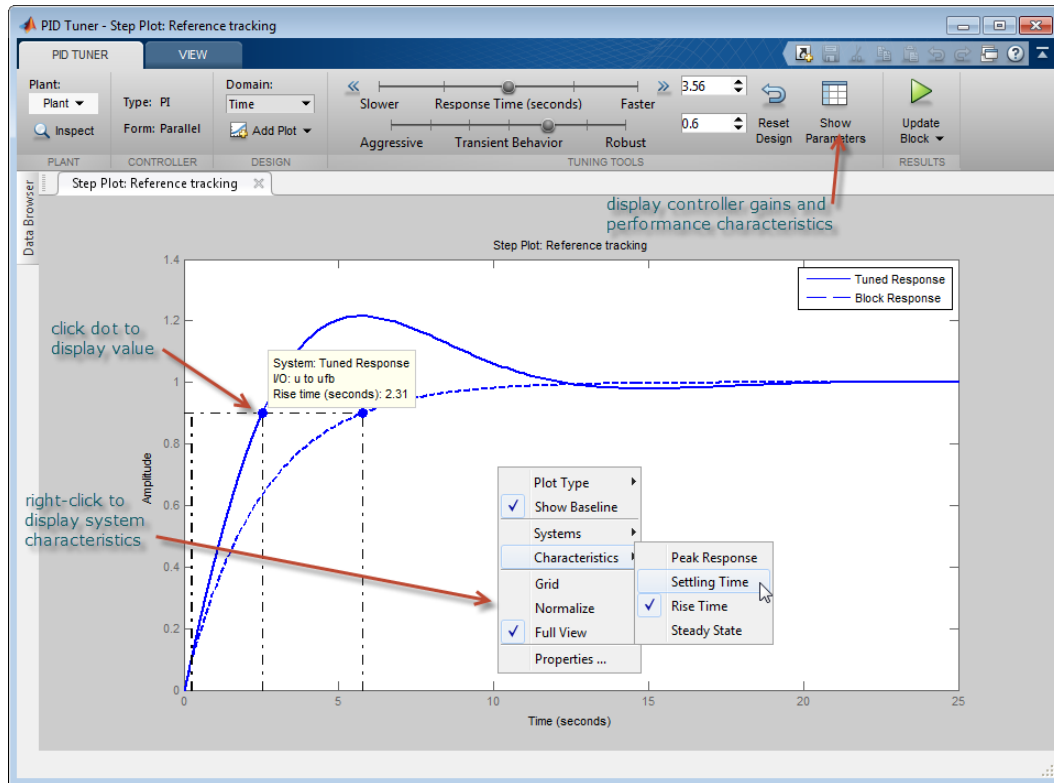
Refine the Design

If the response of the initial controller design does not meet your requirements, you can interactively adjust the design. **PID Tuner** gives you two **Domain** options for refining the controller design:

- `Time` domain (default) — Use the **Response Time** slider to make the closed-loop response of the control system faster or slower. Use the **Transient Behavior** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

- `Frequency` — Use the **Bandwidth** slider to make the closed-loop response of the control system faster or slower (the response time is $2/w_c$, where w_c is the bandwidth). Use the **Phase Margin** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

In both modes, there is a trade-off between reference tracking and disturbance rejection performance. For an example that shows how to use the sliders to adjust this trade-off, see "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)" on page 4-28.

Tip To revert to the initial controller design after moving the sliders, click **Reset Design**.

See Also

Related Examples

• "PID Controller Design for Fast Reference Tracking" on page 4-18

PID Controller Design for Fast Reference Tracking

This example shows how to use **PID Tuner** to design a controller for the plant:

$$sys = \frac{1}{(s+1)^3}.$$

The design requirements are for the closed loop system to track a reference input with a rise time less than 1.5 s, and settling time less than 6 s.

In this example, you represent the plant as an LTI model. For information about using **PID Tuner** to tune a PID Controller block in a Simulink model, see "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection" (Simulink Control Design).

1   Create the plant model and open **PID Tuner** to design a PI controller for a first pass design.

```
sys = zpk([],[-1 -1 -1],1);
pidTuner(sys,'pi')
```
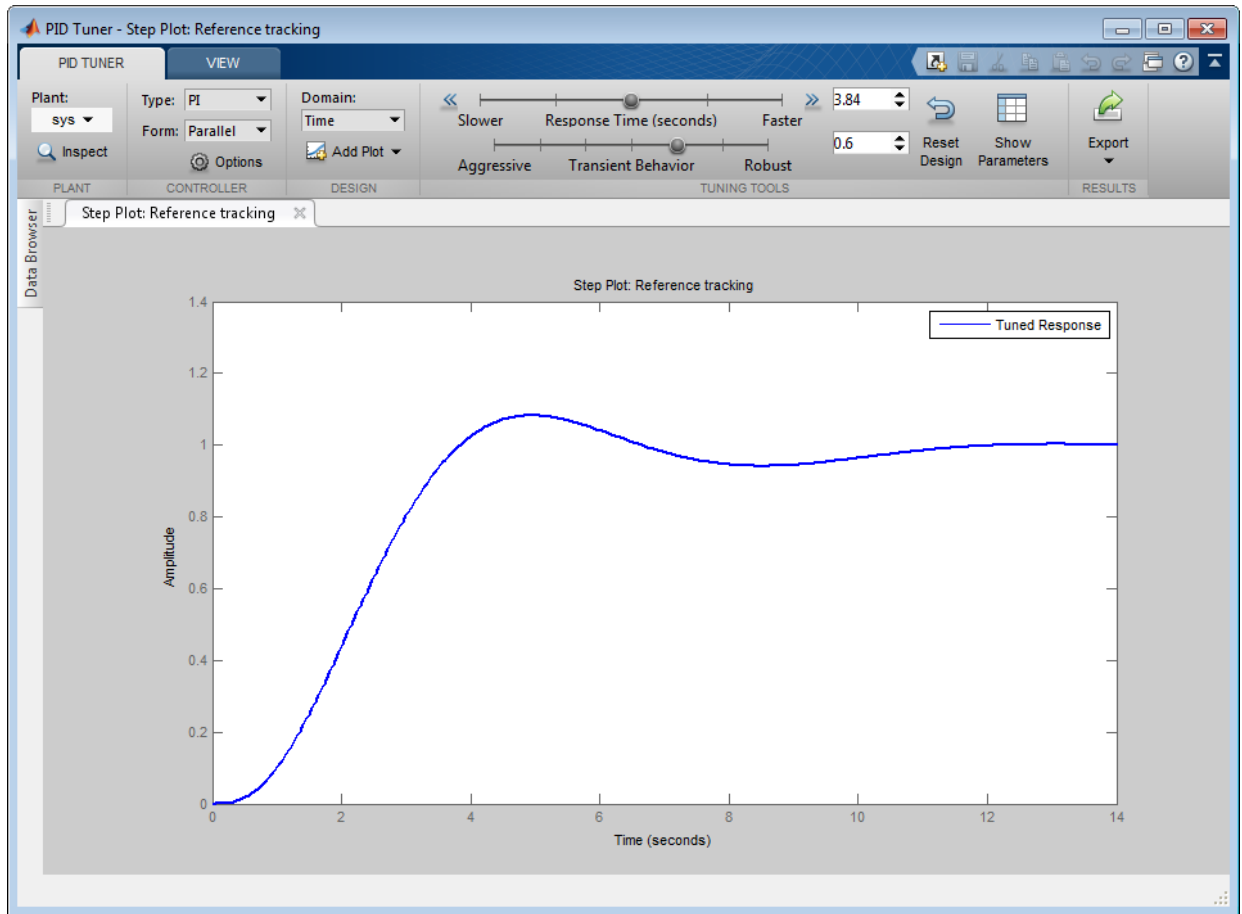
When you open **PID Tuner**, it automatically designs a controller of the type you specify (here, PI). The controller is designed for a balance between performance (response time) and robustness (stability margins). **PID Tuner** displays the closed-loop step response of the system with the designed controller.

Tip You can also open **PID Tuner** from the MATLAB desktop, in the **Apps** tab. When you do so, use the **Plant** menu in **PID Tuner** to specify your plant model.

---

2 Examine the reference tracking rise time and settling time.

Right-click on the plot and select `Characteristics > Rise Time` to mark the rise time as a blue dot on the plot. Select `Characteristics > Settling Time` to mark the settling time. To see tool-tips with numerical values, click each of the blue dots.
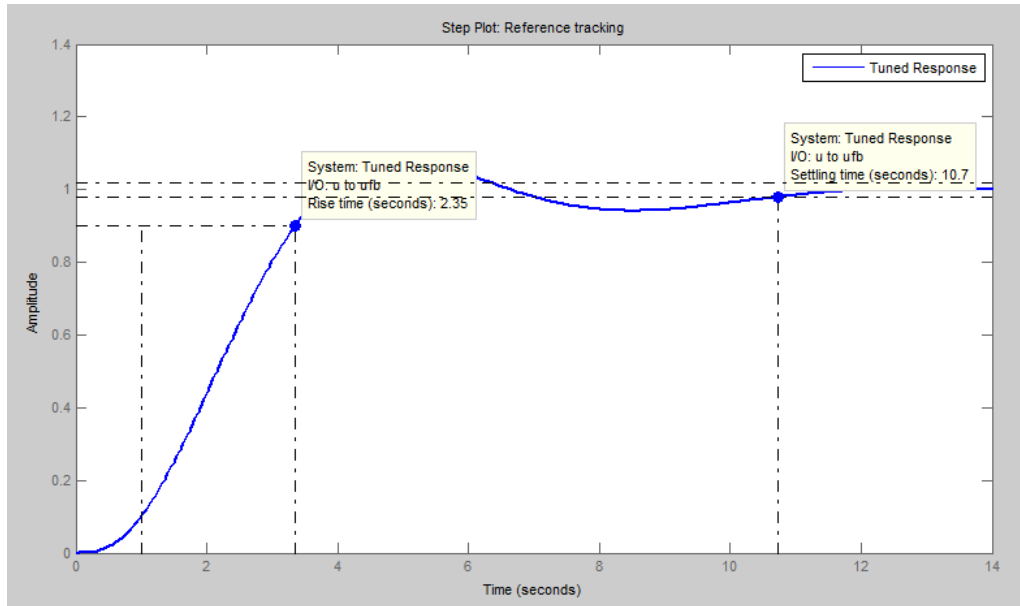
The initial PI controller design provides a rise time of 2.35 s and settling time of 10.7 s. Both results are slower than the design requirements.
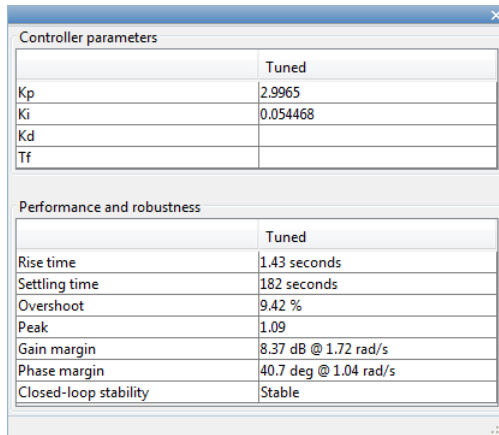
Note To display the performance metrics in a table instead of in tool-tips on the plot, click **Show parameters**. This action opens a display containing performance and robustness metrics and the tuned controller gains.

---

3 Slide the **Response time** slider to the right to try to improve the loop performance. The response plot automatically updates with the new design.
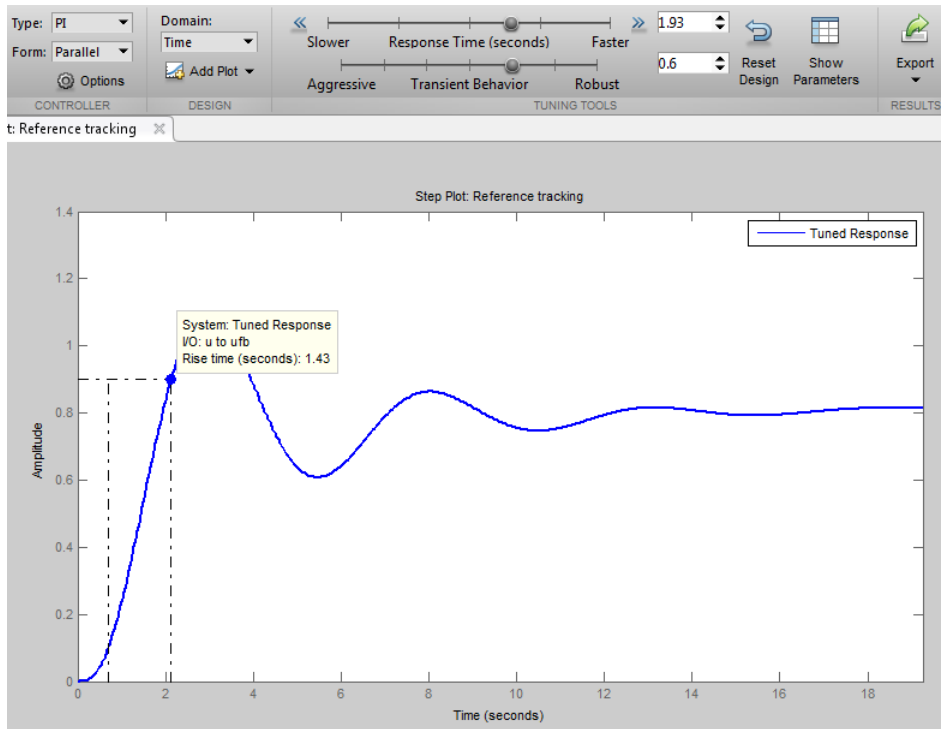
Moving the **Response time** slider far enough to meet the rise time requirement of less than 1.5 s results in more oscillation. Additionally, the parameters display shows that the new response has an unacceptably long settling time.

Controller parameters	
	Tuned
Kp	2.9965
Ki	0.054468
Kd	
Tf	

Performance and robustness	
	Tuned
Rise time	1.43 seconds
Settling time	182 seconds
Overshoot	9.42 %
Peak	1.09
Gain margin	8.37 dB @ 1.72 rad/s
Phase margin	40.7 deg @ 1.04 rad/s
Closed-loop stability	Stable

To achieve the faster response speed, the algorithm must sacrifice stability.

4 Change the controller type to improve the response.

Adding derivative action to the controller gives **PID Tuner** more freedom to achieve adequate phase margin with the desired response speed.

In the **Type** menu, select `PIDF`. **PID Tuner** designs a new PIDF controller. (See "PID Controller Type" on page 4-5 for more information about available controller types.)
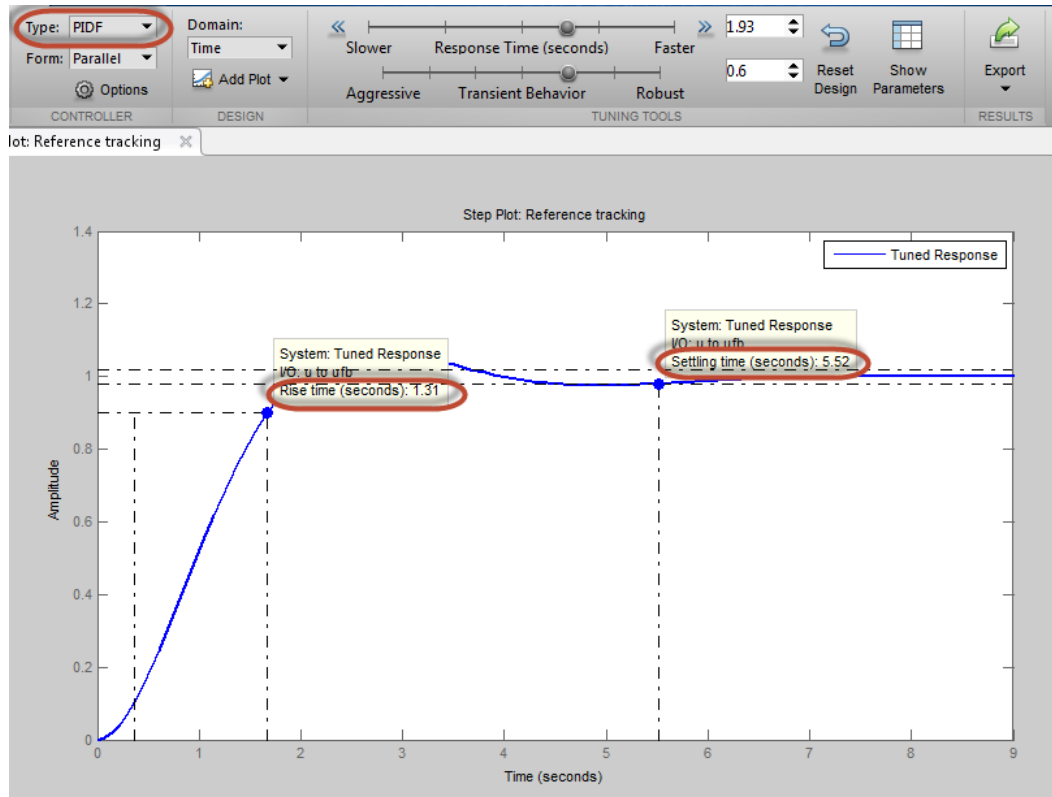
The rise time and settling time now meet the design requirements. You can use the **Response time** slider to make further adjustments to the response. To revert to the default automated tuning result, click **Reset Design**.

Note To adjust the closed-loop bandwidth instead of the response time, select Frequency domain from the **Design mode** menu . The bandwidth is inversely proportional to the response time.

5 Analyze other system responses, if appropriate.

To analyze other system responses, click **Add Plot**. Select the system response you want to analyze.
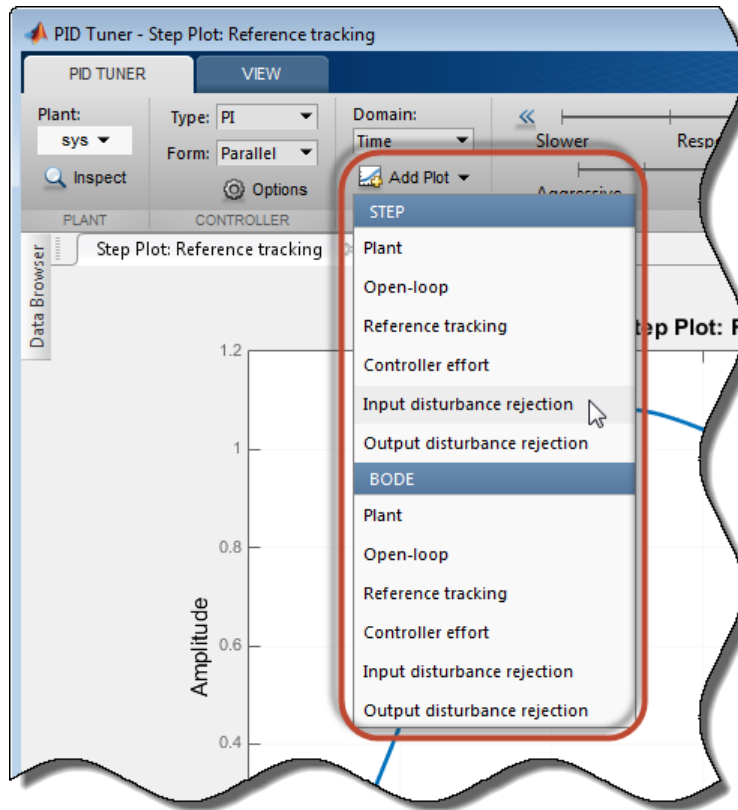
For example, to observe the closed-loop step response to disturbance at the plant input, in the **Step** section of the **Add Plot** menu, select `Input disturbance rejection`. The disturbance rejection response appears in a new figure.
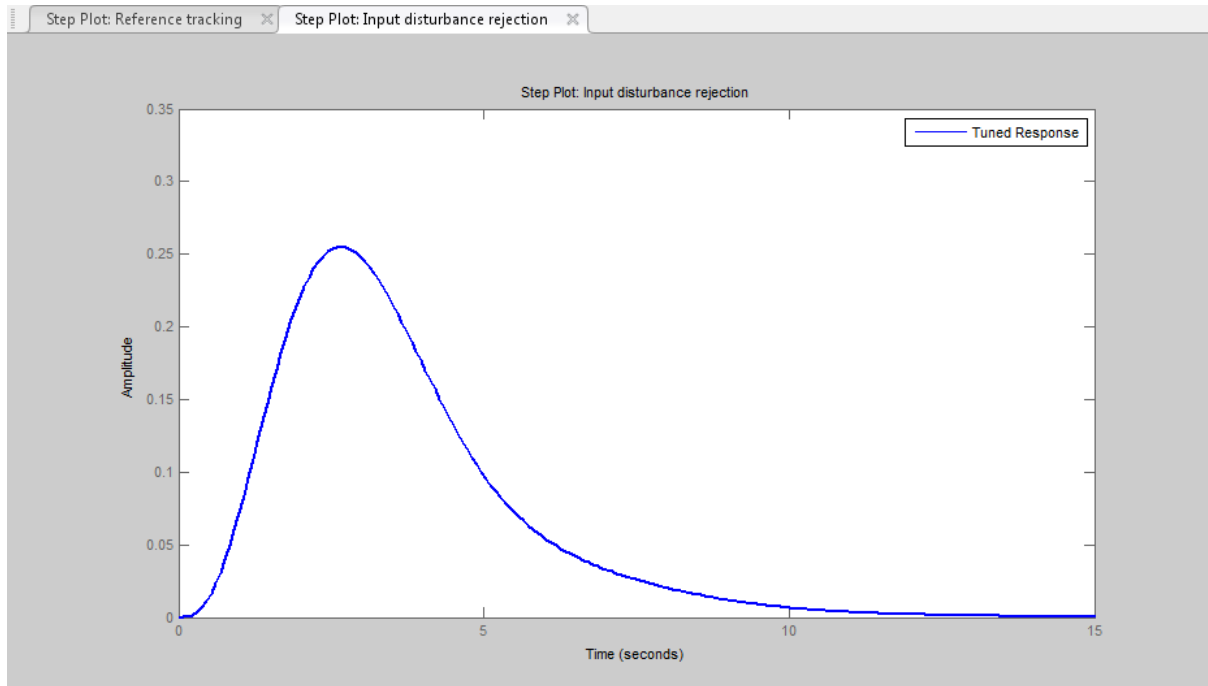
See "Analyze Design in PID Tuner" on page 4-10 for more information about available response plots.

Tip Use the options in the **View** tab to change how **PID Tuner** displays multiple plots.

---

6 Export your controller design to the MATLAB workspace.

To export your final controller design to the MATLAB workspace, click **Export**. **PID Tuner** exports the controller as a

- `pid` controller object, if the **Form** is `Parallel`
- `pidstd` controller object, if the **Form** is `Standard`

Alternatively, you can export a model using the right-click menu in the **Data Browser**. To do so, click the **Data Browser** tab.
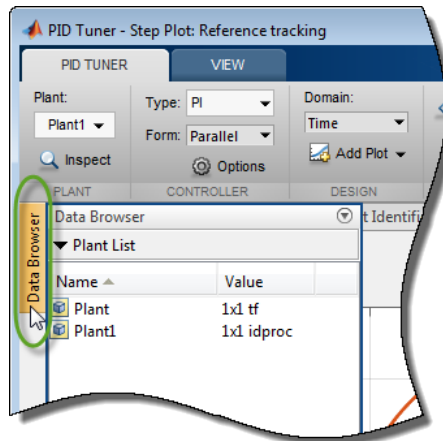
Then, right-click the model and select **Export**.
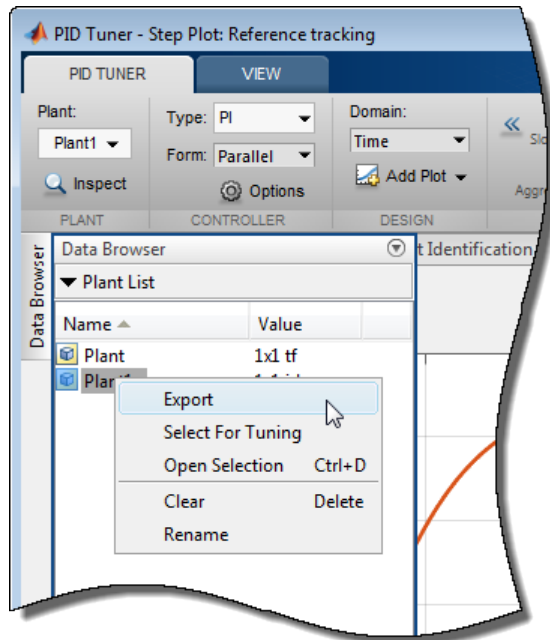
See Also

More About

*   "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)" on page 4-28
*   "Analyze Design in PID Tuner" on page 4-10

Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)

This example shows how to tune a PID controller to reduce overshoot in reference tracking or to improve rejection of a disturbance at the plant input. Using the **PID Tuner** app, the example illustrates the tradeoff between reference tracking and disturbance-rejection performance in PI and PID control systems.

In this example, you represent the plant as an LTI model. For information about using **PID Tuner** to tune a PID Controller block in a Simulink model, see "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection" (Simulink Control Design).

Consider the control system of the following illustration.
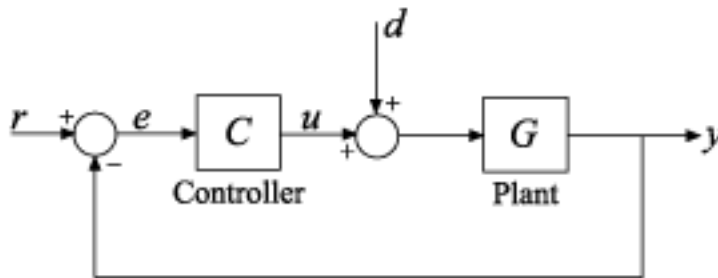
The plant in this example is:

$$Plant = \frac{0.3}{s^2 + 0.1s}.$$

Reference tracking is the response at y to signals at r. Disturbance rejection is a measure of the suppression at y of signals at d. When you use **PID Tuner** to tune the controller, you can adjust the design to favor reference tracking or disturbance rejection as your application requires.

Design Initial PI Controller

Having an initial controller design provides a baseline against which you can compare results as you tune a PI controller. Create an initial PI controller design for the plant using PID tuning command `pidtune`.

```
G = tf(0.3,[1,0.1,0]);     % plant model
C = pidtune(G,'PI');
```

Use the initial controller design to open **PID Tuner**.

```
pidTuner(G,C)
```
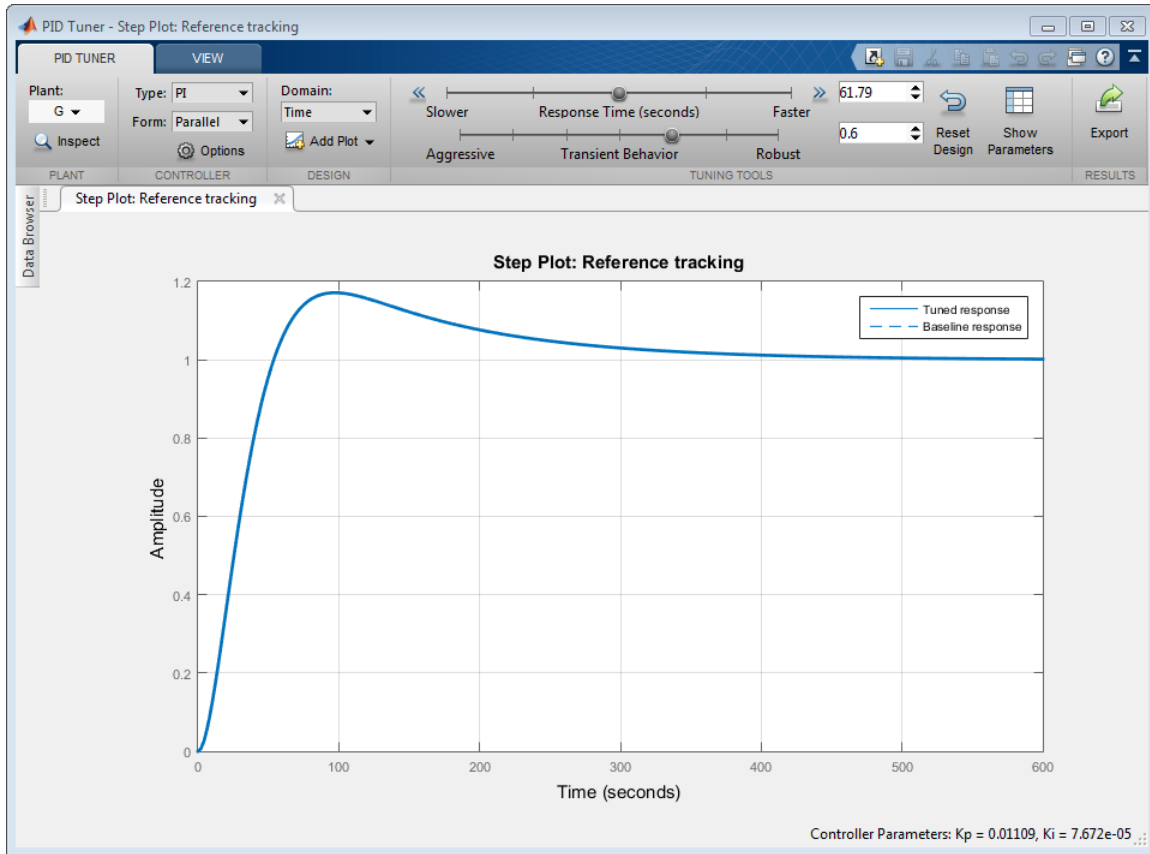

Add a step response plot of the input disturbance rejection. Select **Add Plot > Input Disturbance Rejection**.
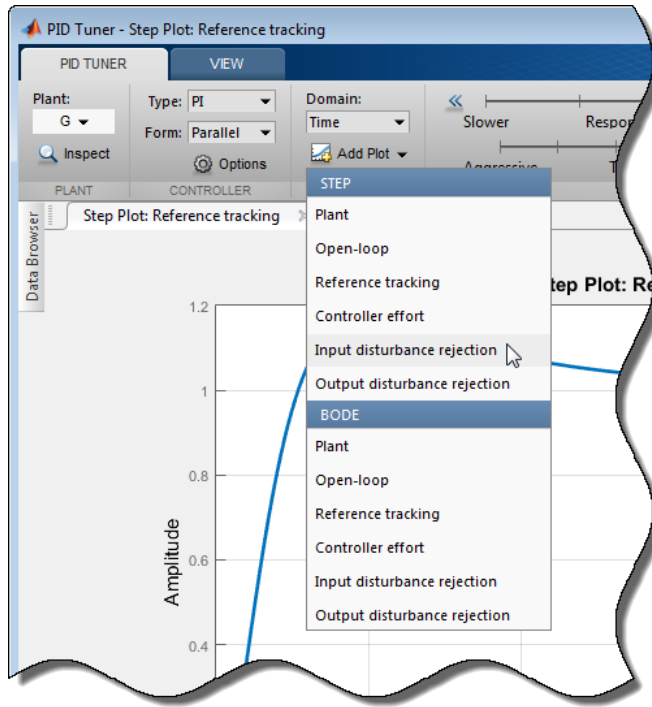
PID Tuner tiles the disturbance-rejection plot side by side with the reference-tracking plot.
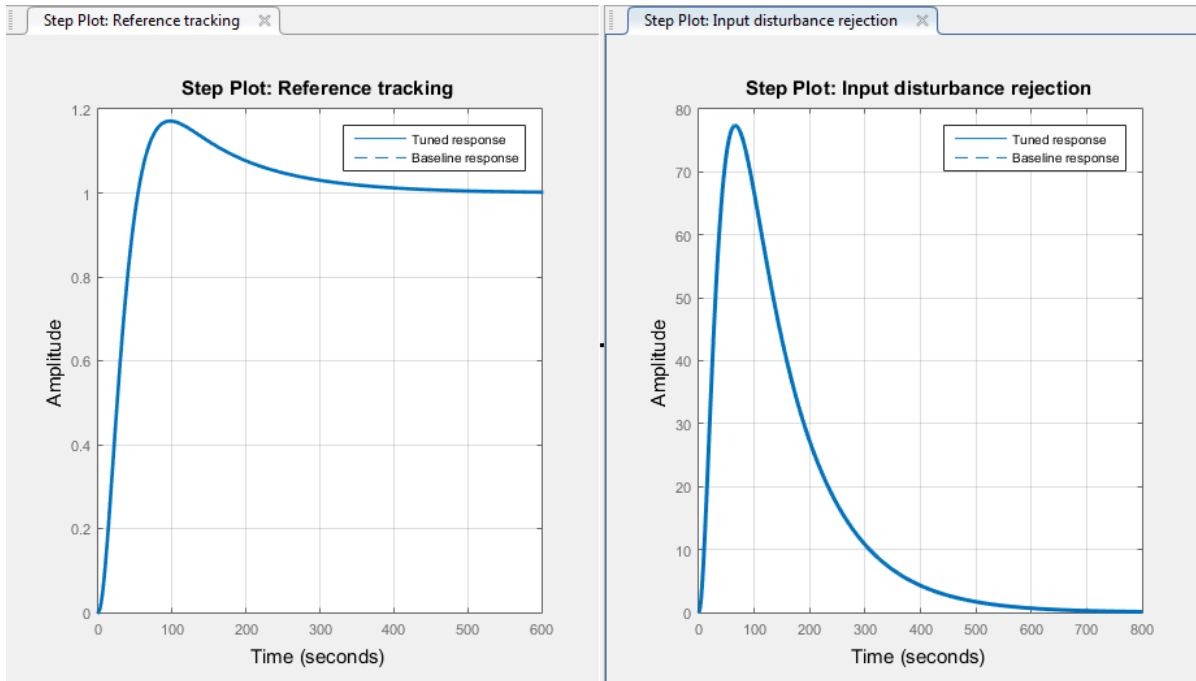
Tip Use the options in the **View** tab to change how **PID Tuner** displays multiple plots.

By default, for a given bandwidth and phase margin, **PID Tuner** tunes the controller to achieve a balance between reference tracking and disturbance rejection. In this case, the controller yields some overshoot in the reference-tracking response. The controller also suppresses the input disturbance with a longer settling time than the reference tracking, after an initial peak.

Adjust Transient Behavior

Depending on your application, you might want to alter the balance between reference tracking and disturbance rejection to favor one or the other. For a PI controller, you can alter this balance using the **Transient Behavior** slider. Move the slider to the left to improve the disturbance rejection. The responses with the initial controller design are now displayed as the `Baseline response` (dotted line).
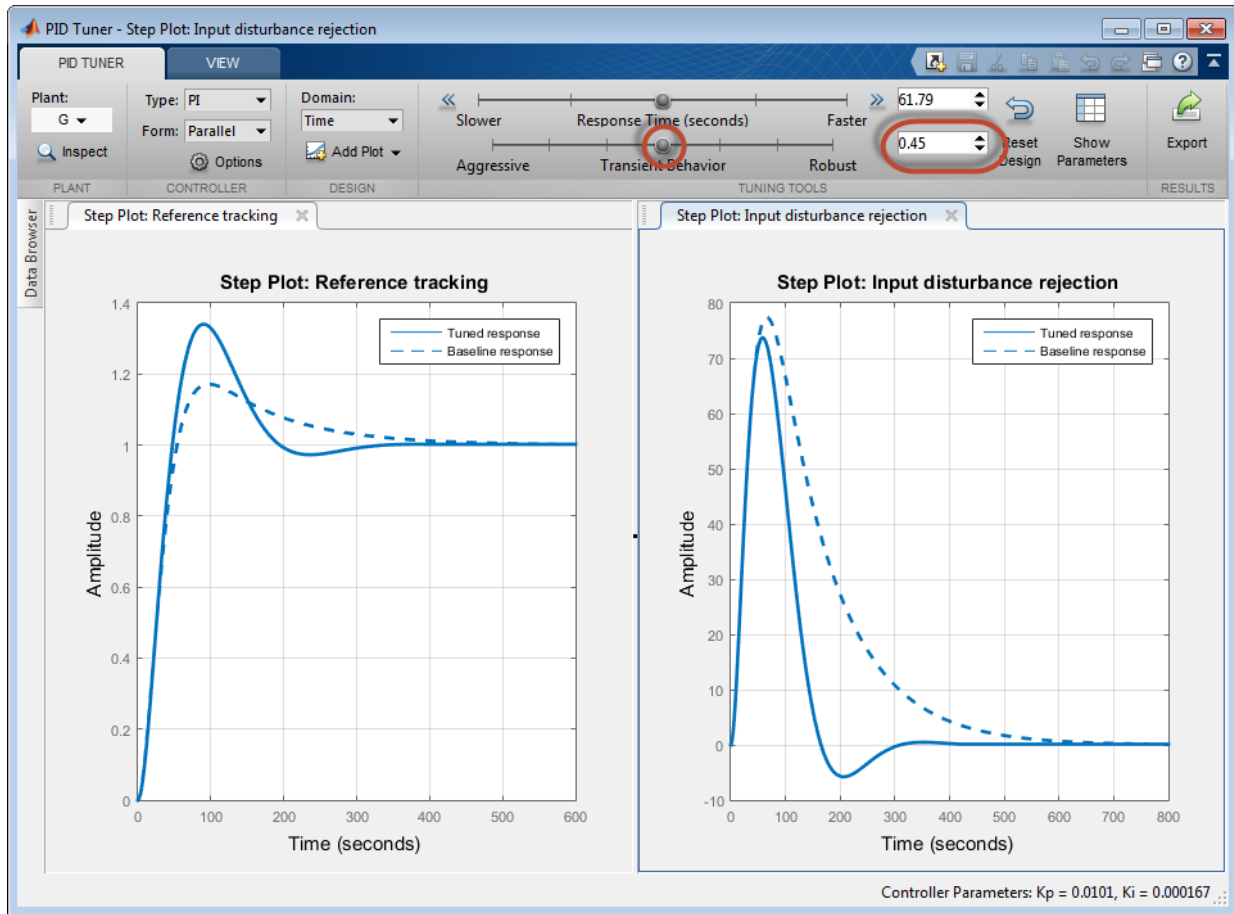
Lowering the transient-behavior coefficient to 0.45 speeds up disturbance rejection, but also increases overshoot in the reference-tracking response.

Tip Right-click on the reference-tracking plot and select **Characteristics > Peak Response** to obtain a numerical value for the overshoot.

Move the **Transient behavior** slider to the right until the overshoot in the reference-tracking response is minimized.
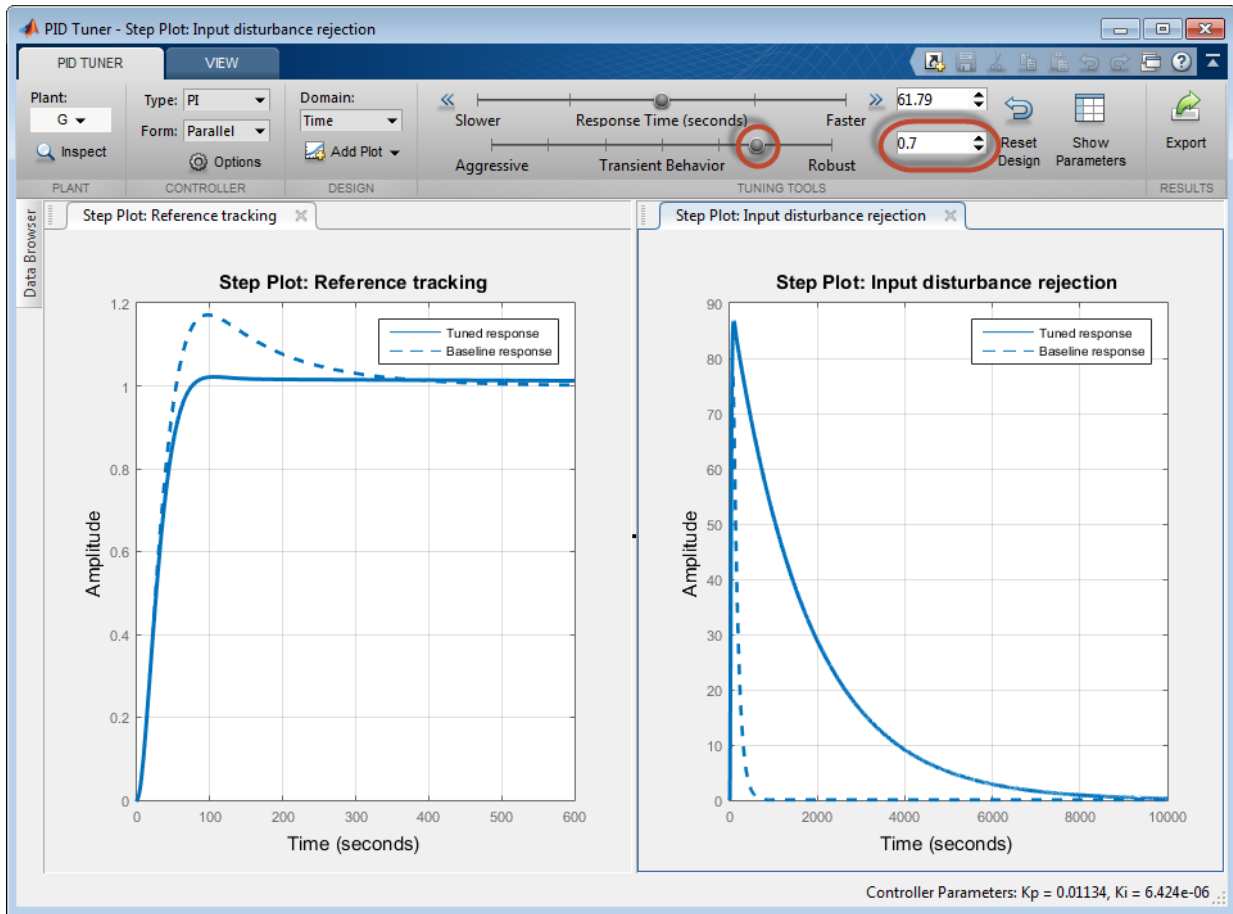
Increasing the transient-behavior coefficient to 0.70 nearly eliminates the overshoot, but results in extremely sluggish disturbance rejection. You can try moving the **Transient behavior** slider until you find a balance between reference tracking and disturbance rejection that is suitable for your application. The effect that changing the slider has on the balance depends on the plant model. For some plant models, the effect is not as large as shown in this example.

Change PID Tuning Design Focus

So far, the response time of the control system has remained fixed while you have changed the transient-behavior coefficient. These operations are equivalent to fixing the

bandwidth and varying the target minimum phase margin of the system. If you want to fix both the bandwidth and target phase margin, you can still change the balance between reference tracking and disturbance rejection. To tune a controller that favors either disturbance rejection or reference tracking, you change the design focus of the PID tuning algorithm.

Changing the **PID Tuner** design focus is more effective the more tunable parameters there are in the control system. Therefore, it does not have much effect when used with a PI controller. To see its effect, change the controller type to PIDF. In the **Type** menu, select PIDF.
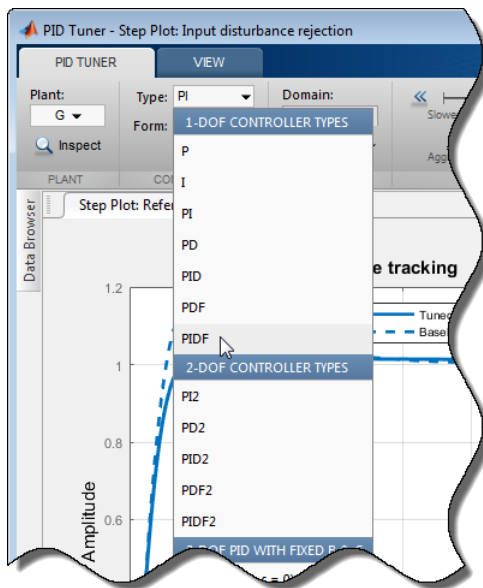
PID Tuner automatically designs a controller of the new type, PIDF. Move the **Transient Behavior** slider to set the coefficient back to 0.6.

Save this new design as the baseline design, by clicking the **Export** arrow ▼ and selecting Save as Baseline.
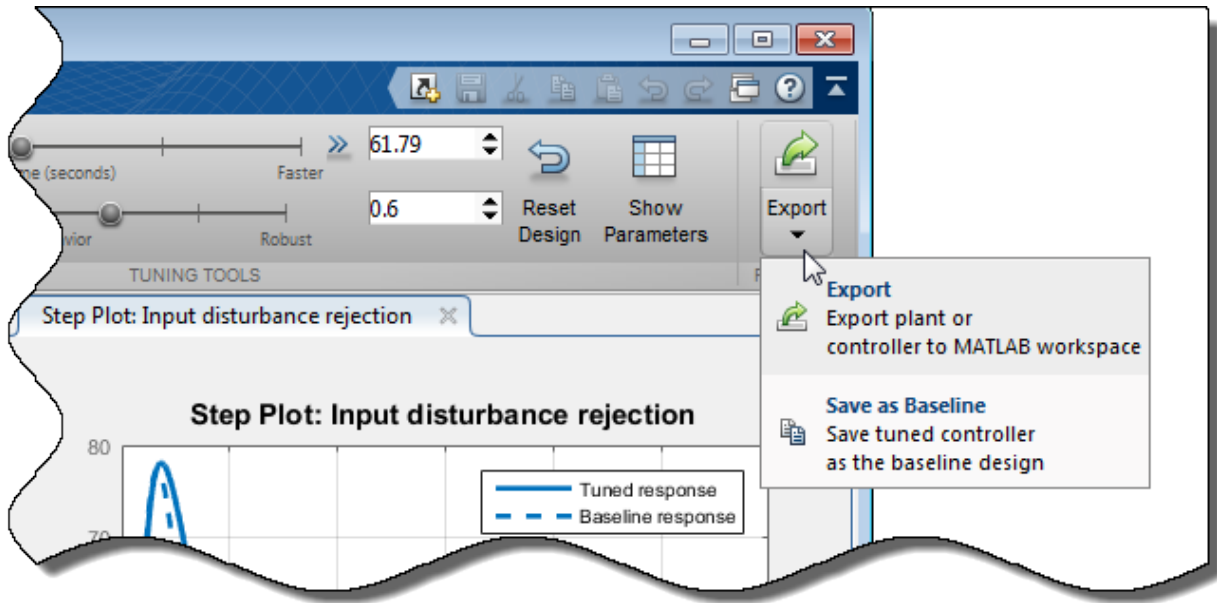
The PIDF design replaces the original PI design as the baseline plot.

As in the PI case, the initial PIDF design balances reference tracking and disturbance rejection. Also as in the PI case, the controller yields some overshoot in the reference-tracking response, and suppresses the input disturbance with a similar settling time.
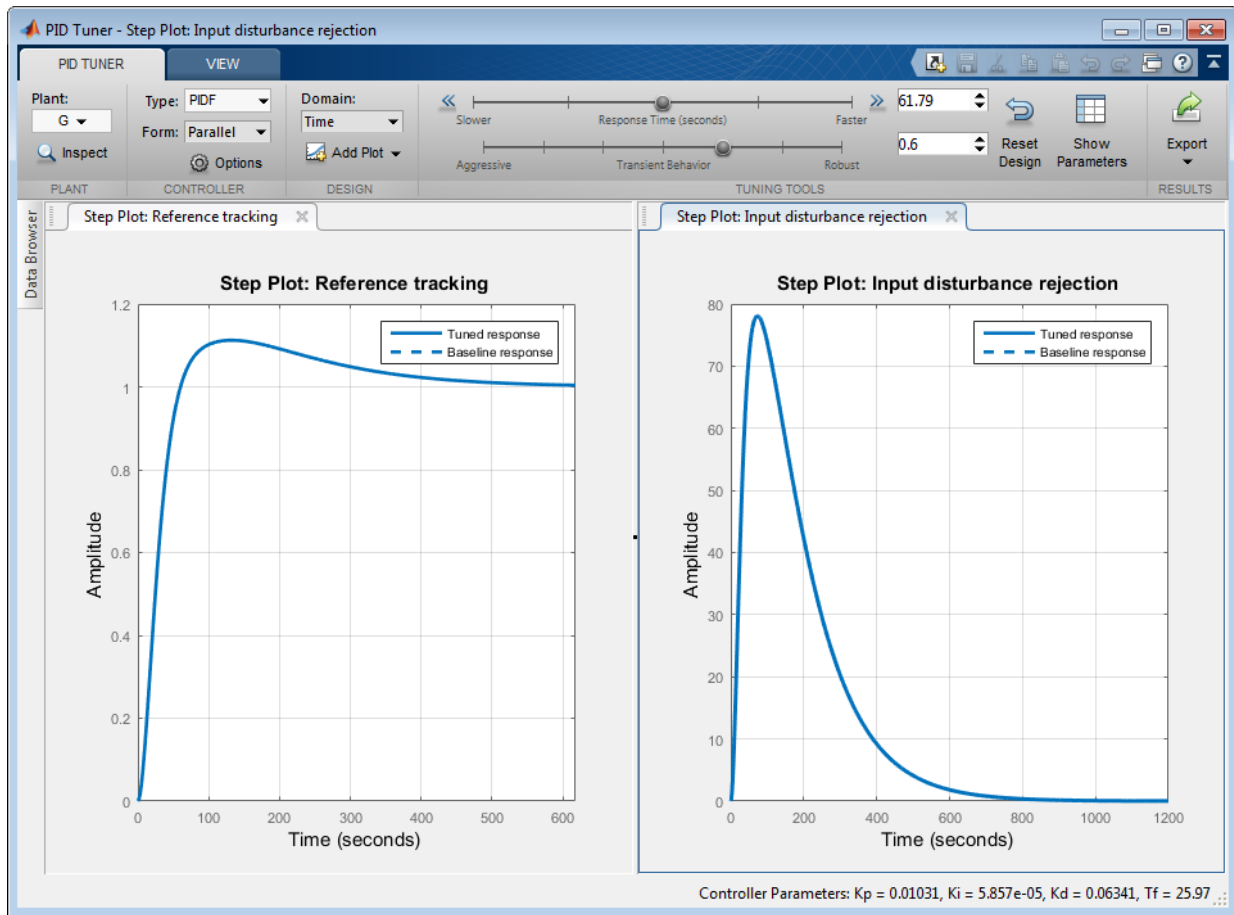
Change the **PID Tuner** design focus to favor reference tracking without changing the response time or the transient-behavior coefficient. To do so, click ⚙ **Options**, and in the **Focus** menu, select `Reference tracking`.
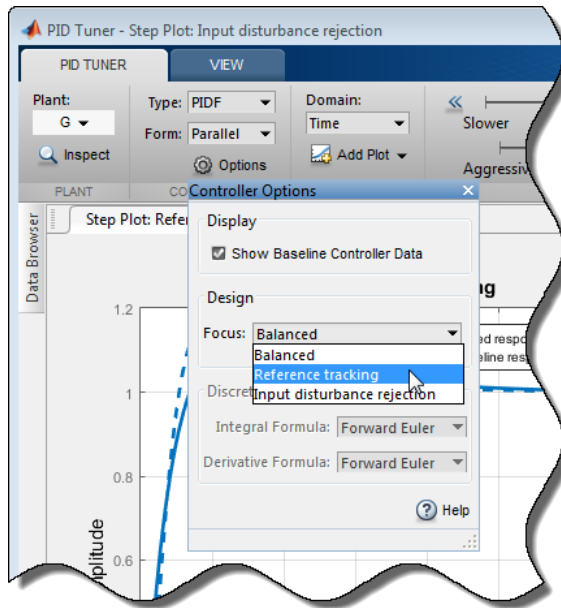
PID Tuner automatically retunes the controller coefficients with a focus on reference-tracking performance.
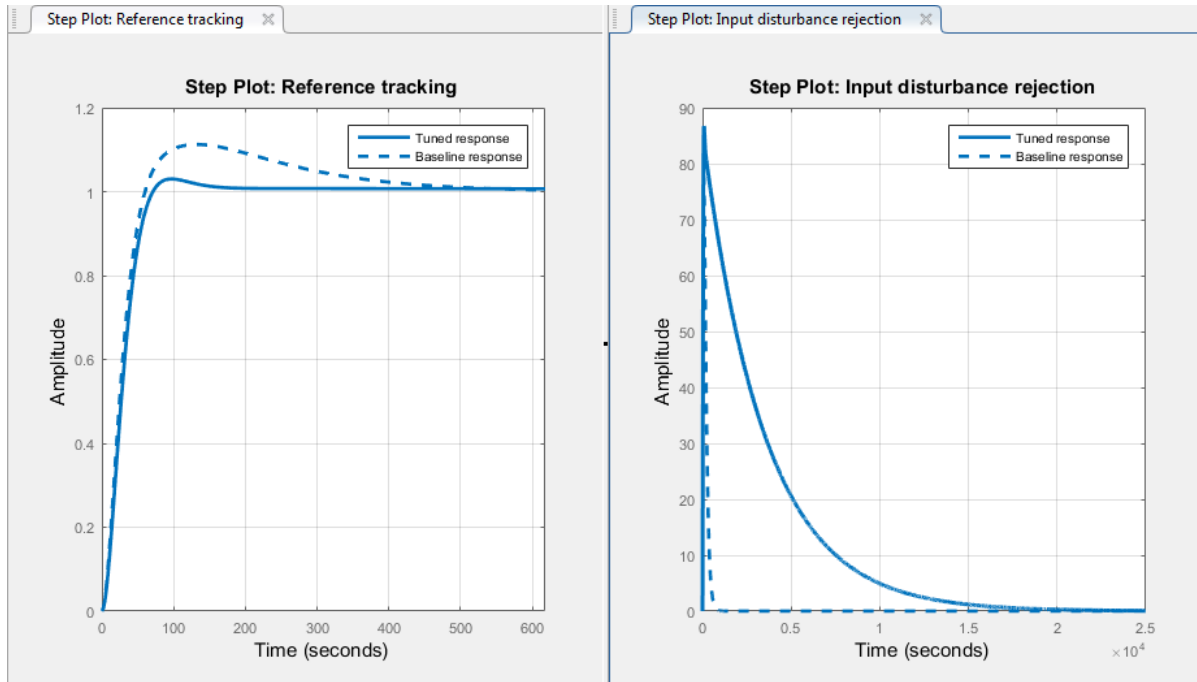
The PIDF controller tuned with reference-tracking focus is displayed as `Tuned response` (solid line). The plots show that the resulting controller tracks the reference input with considerably less overshoot and a faster settling time than the balanced controller design. However, the design yields much poorer disturbance rejection.

Change the design focus to favor disturbance rejection. In the ⚙ **Options** dialog box, in the **Focus** menu, select `Input disturbance rejection`.
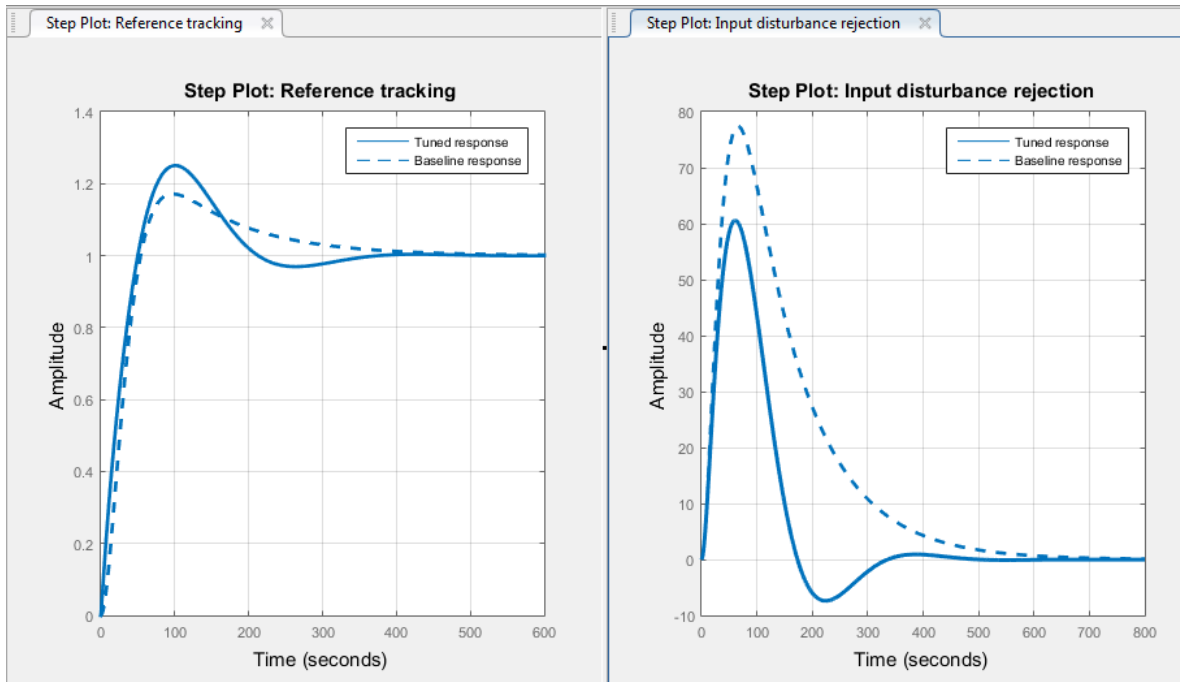
This controller design yields improved disturbance rejection, but results in some increased overshoot in the reference-tracking response.

When you use design focus option, you can still adjust the **Transient Behavior** slider for further fine-tuning of the balance between the two measures of performance. Use the design focus and the sliders together to achieve the performance balance that best meets your design requirements. The effect of this fine tuning on system performance depends strongly on the properties of your plant. For some plants, moving the **Transient Behavior** slider or changing the **Focus** option has little or no effect.

See Also

More About

*   "PID Tuning Algorithm" on page 4-65
*   "PID Controller Design for Fast Reference Tracking" on page 4-18

- "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)" on page 4-41

Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)

This example shows how to use command-line PID tuning options to reduce overshoot in reference tracking or to improve rejection of a disturbance at the plant input. Using the `pidtune` command, the example illustrates the tradeoff between reference tracking and disturbance-rejection performance in PI and PID control systems.

Consider the control system of the following illustration.
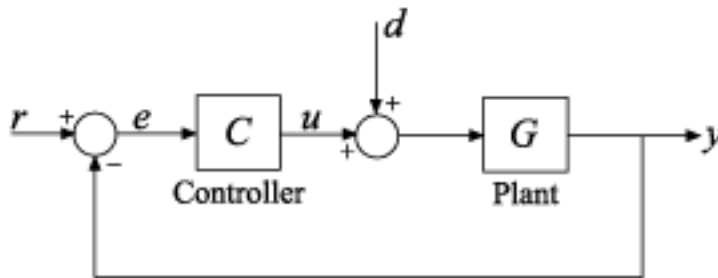
Setpoint tracking is the response at *y* to signals at *r*. Input disturbance rejection is the suppression at *y* of signals at *d*.

Create a model of the plant, which for this example is given by:

$$G = \frac{0.3}{s^2 + 0.1s}.$$

```
G = tf(0.3,[1 0.1 0]);
```

Design a PI controller for this plant, using a bandwidth of 0.03 rad/s.

```
wc = 0.03;
[C1,info] = pidtune(G,'PI',wc);
```

Examine the step-reference tracking and step-disturbance rejection of the control system using the default controller. The disturbance response from *d* to *y* is equivalent to the response of a closed loop given by `feedback(G,C1)`.

```
T1 = feedback(G*C1,1);
GS1 = feedback(G,C1);
```

```
subplot(2,1,1);
stepplot(T1)
title('Reference Tracking')
subplot(2,1,2);
stepplot(GS1)
title('Disturbance Rejection')
```
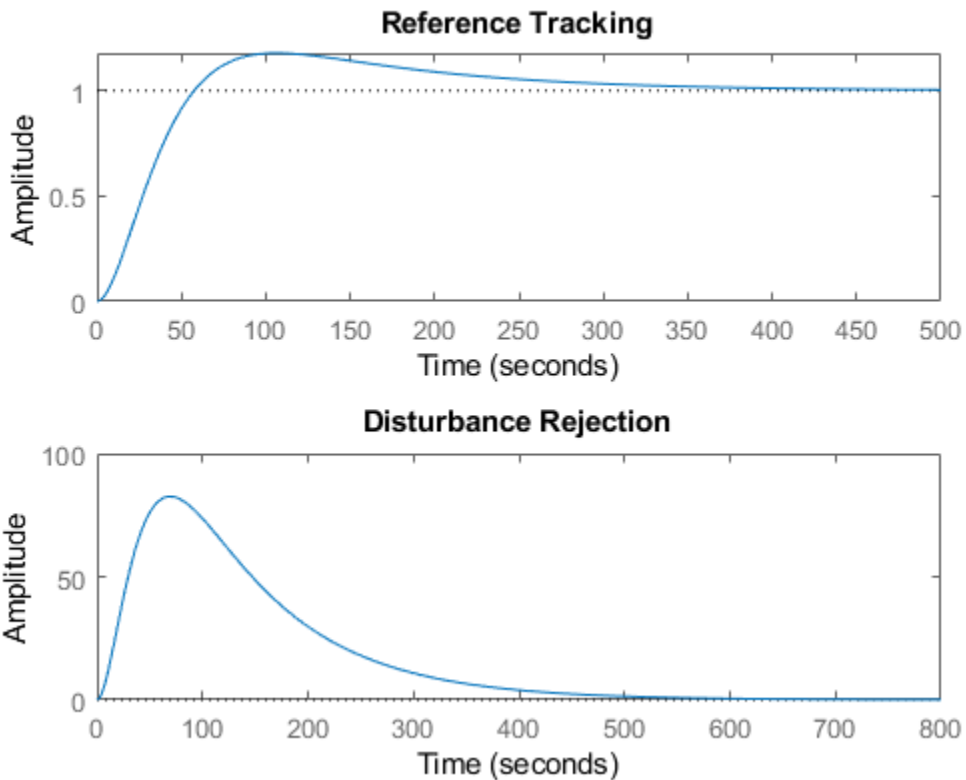

By default, for a given bandwidth, `pidtune` tunes the controller to achieve a balance between reference tracking and disturbance rejection. In this case, the controller yields some overshoot in the reference-tracking response. The controller also suppresses the input disturbance with a somewhat longer settling time than the reference tracking, after an initial peak.

Depending on your application, you might want to alter the balance between reference tracking and disturbance rejection to favor one or the other. For a PI controller, you can alter this balance by changing the phase margin of the tuned system. The default controller returned by `pidtune` yields a phase margin of 60°.

```
info.PhaseMargin
```

```
ans =

    60.0000
```

Design controllers for phase margins of 45° and 70° with the same bandwidth, and compare the resulting reference tracking and disturbance rejection.

```
opts2 = pidtuneOptions('PhaseMargin',45);
C2 = pidtune(G,'PI',wc,opts2);
T2 = feedback(G*C2,1);
GS2 = feedback(G,C2);

opts3 = pidtuneOptions('PhaseMargin',70);
C3 = pidtune(G,'PI',wc,opts3);
T3 = feedback(G*C3,1);
GS3 = feedback(G,C3);

subplot(2,1,1);
stepplot(T1,T2,T3)
legend('PM = 60','PM = 45','PM = 70')
title('Reference Tracking')
subplot(2,1,2);
stepplot(GS1,GS2,GS3)
title('Disturbance Rejection')
```
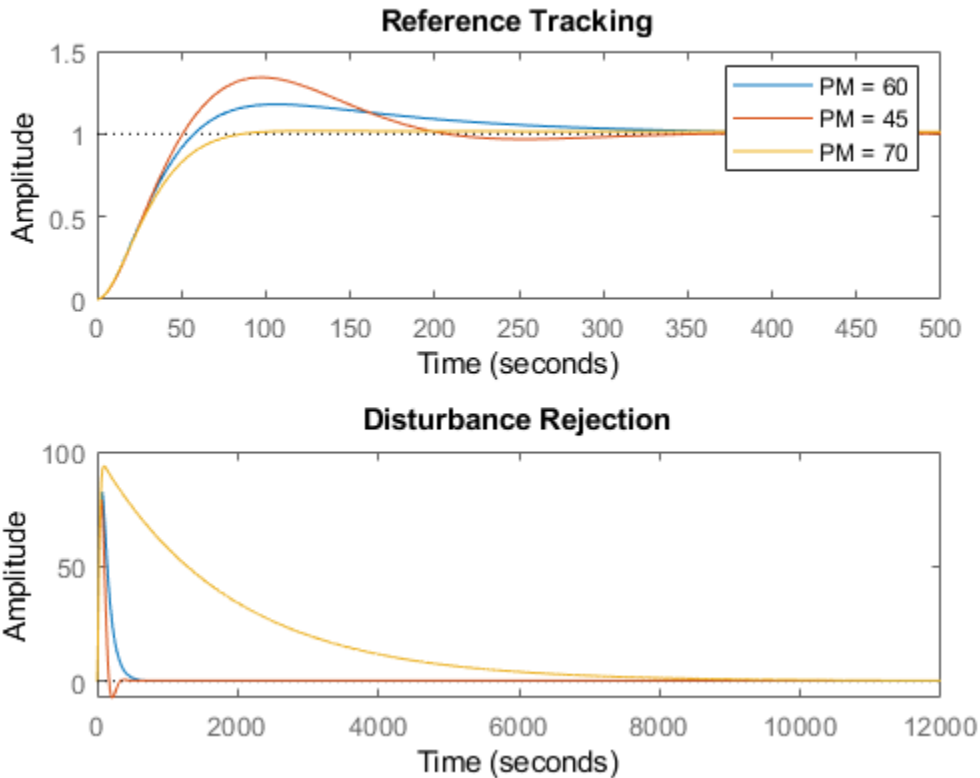
Lowering the phase margin to 45° speeds up disturbance rejection, but also increases overshoot in the reference tracking response. Increasing the phase margin to 70° eliminates the overshoot completely, but results in extremely sluggish disturbance rejection. You can try different phase margin values until you find one that balances reference tracking and disturbance rejection suitably for your application. The effect of the phase margin on this balance depends on the plant model. For some plant models, the effect is not as large as shown in this example.

If you want to fix both the bandwidth and phase margin of your control system, you can still change the balance between reference tracking and disturbance rejection using the `DesignFocus` option of `pidtune`. You can set `DesignFocus` to either `'disturbance-rejection'` or `'reference-tracking'` to tune a controller that favors one or the other.

The `DesignFocus` option is more effective for a control system with more tunable parameters. Therefore, it does not have much effect when used with a PI controller. To see its effect, design a PIDF controller for the same bandwidth and the default phase margin (60°) using each of the `DesignFocus` values. Compare the results.

```
opts4 = pidtuneOptions('DesignFocus','balanced');   % default focus
C4 = pidtune(G,'PIDF',wc,opts4);
T4 = feedback(G*C4,1);
GS4 = feedback(G,C4);

opts5 = pidtuneOptions('DesignFocus','disturbance-rejection');
C5 = pidtune(G,'PIDF',wc,opts5);
T5 = feedback(G*C5,1);
GS5 = feedback(G,C5);

opts6 = pidtuneOptions('DesignFocus','reference-tracking');
C6 = pidtune(G,'PIDF',wc,opts6);
T6 = feedback(G*C6,1);
GS6 = feedback(G,C6);

subplot(2,1,1);
stepplot(T4,T5,T6)
legend('Balanced','Rejection','Tracking')
title('Reference Tracking')
subplot(2,1,2);
stepplot(GS4,GS5,GS6)
title('Disturbance Rejection')
```
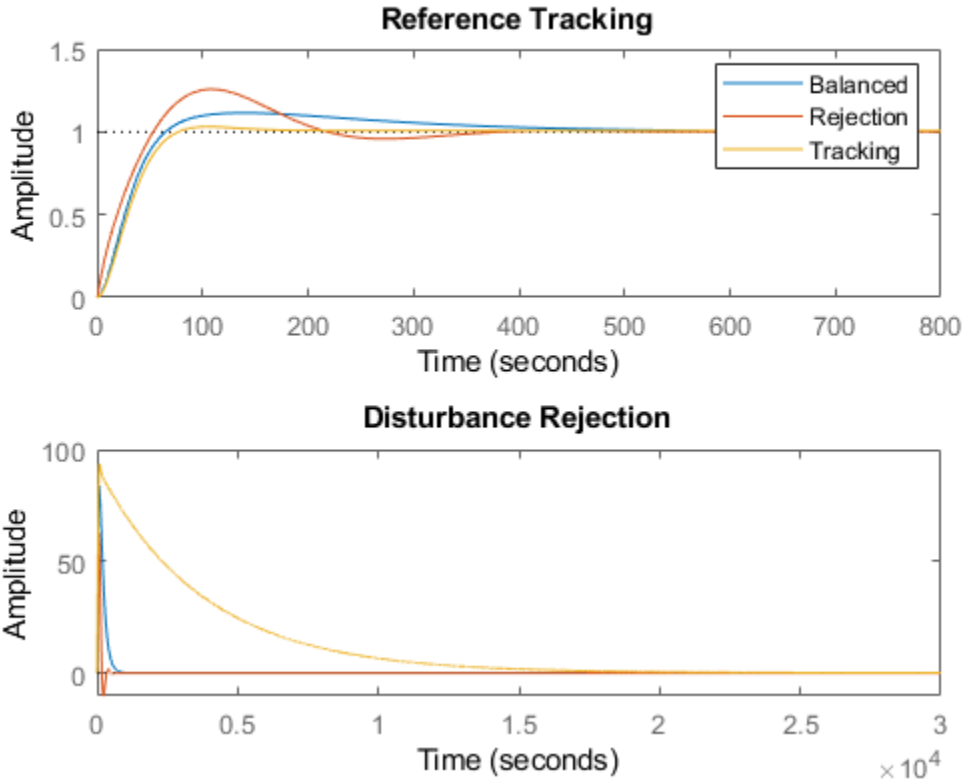
When you use the `DesignFocus` option to favor reference tracking or disturbance rejection in the tuned control system, you can still adjust phase margin for further fine tuning of the balance between these two measures of performance. Use `DesignFocus` and `PhaseMargin` together to achieve the performance balance that best meets your design requirements.

The effect of both options on system performance depends strongly on the properties of your plant. For some plants, changing the `PhaseMargin` or `DesignFocus` options has little or no effect.

See Also

More About

- "PID Tuning Algorithm" on page 4-65
- "PID Controller Design at the Command Line"
- "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)" on page 4-28

Interactively Estimate Plant Parameters from Response Data

This example shows how to use **PID Tuner** to fit a linear model to measured SISO response data.

If you have System Identification Toolbox software, you can use **PID Tuner** to estimate the parameters of a linear plant model based on time-domain response data measured from your system. **PID Tuner** then tunes a PID controller for the resulting estimated model. **PID Tuner** gives you several techniques to graphically, manually, or automatically adjust the estimated model to match your response data. This example illustrates some of those techniques.

In this example, you load measured response data from a data file into the MATLAB workspace you represent the plant as an LTI model. For information about generating simulated data from a Simulink model, see "Interactively Estimate Plant from Measured or Simulated Response Data" (Simulink Control Design).

Import Response Data for Identification

1 Open **PID Tuner** and load measured response data into the MATLAB workspace.

```
pidTuner(tf(1),'PI')
load PIDPlantMeasuredIOData
```

When you import response data, **PID Tuner** assumes that your measured data represents a plant connected to the PID controller in a negative-feedback loop. In other words, **PID Tuner** assumes the following structure for your system. **PID Tuner** assumes that you injected a step signal at the plant input u and measured the system response at y, as shown.
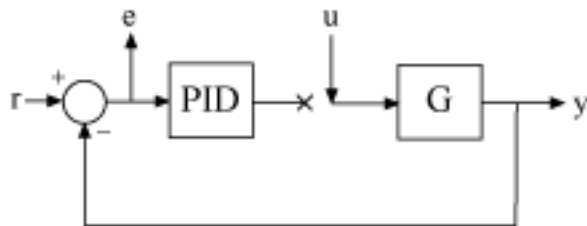
The sample data file for this example, contains three variables, each of which is a 501-by-1 array. inputu is the unit step function injected at u to obtain the response

data. `outputy` is the measured response of the system at `y`. The time vector `t`, runs from 0 to 50 s with a 0.1 s sample time. Comparing `inputu` to `t` shows that the step occurs at `t` = 5 s.

> **Tip** You can import response data stored as a numeric array (as in this example), a `timeseries` object, or an `iddata` object.

2   In **PID Tuner**, in the **Plant** menu, select `Identify New Plant`.
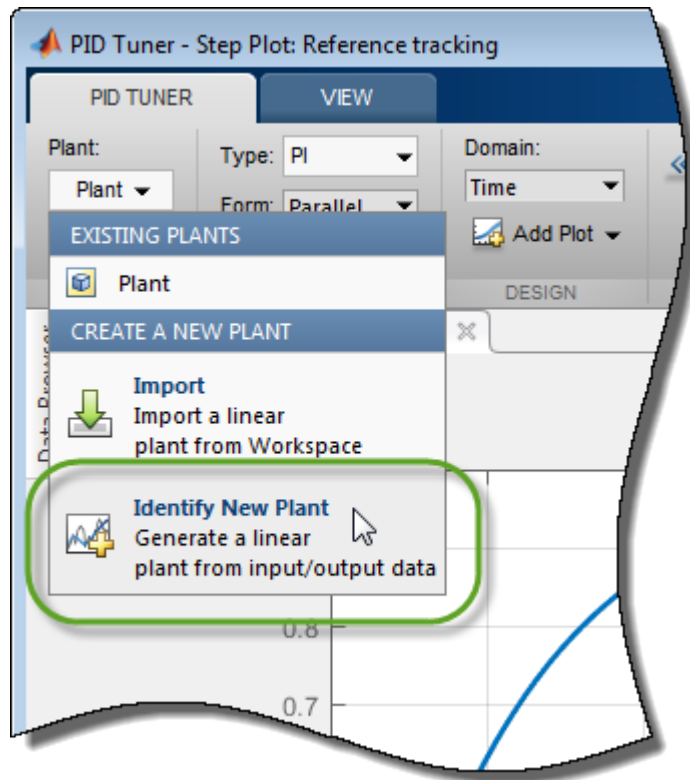
3   In the **Plant Identification** tab, click ➕ **Get I/O data** and select **Step Response**. This action opens the **Import Step Response** dialog box.

Enter information about the response data. The output signal is the measured system response, `outputy`. The input step signal is parametrized as shown in the diagram in

the dialog box. Here, enter 5 for the **Onset Lag**, and 0.1 for **Sample Time**. Then, click ⬇ **Import**.
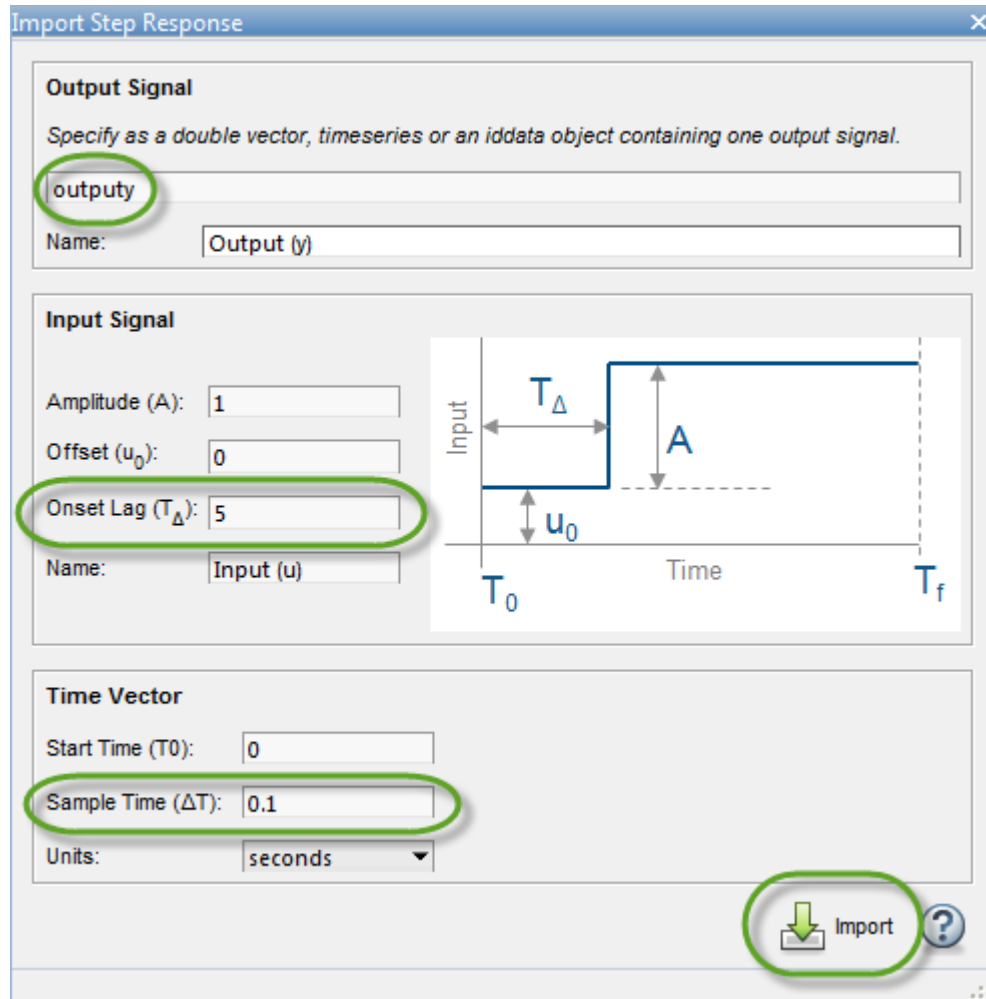
The **Plant Identification** plot displays the response data and the response of an initial estimated plant.
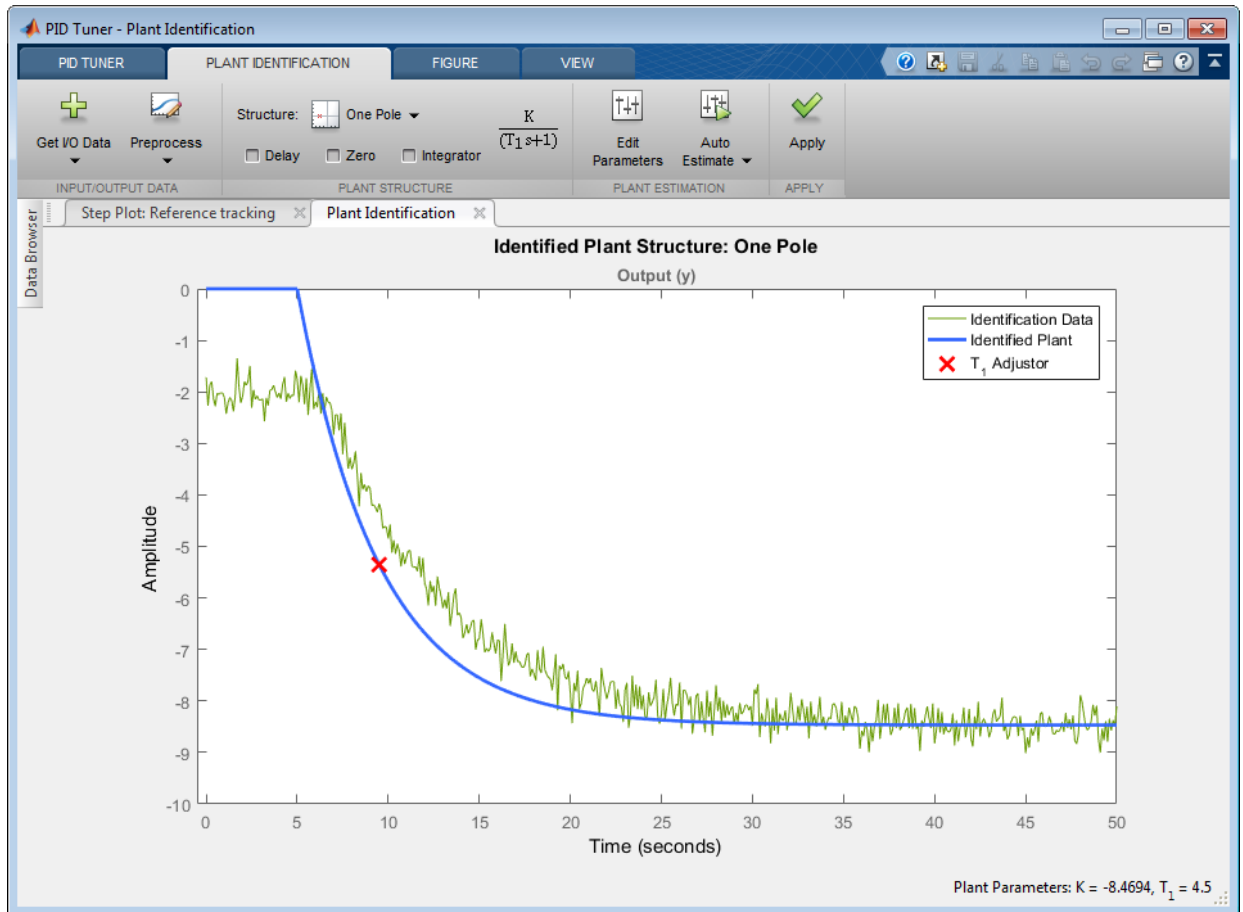
Preprocess Data

Depending on the quality and features of your response data, you might want to perform some preprocessing on the data to improve the estimated plant results. **PID Tuner** provides several options for preprocessing response data, such as removing offsets, filtering, or extracting a subset of the data. In this example, the response data has an offset. It is important for good identification results to remove data offsets. Use the **Preprocess** menu to do so. (For information about other data preprocessing options, see "Preprocess Data" on page 4-60.)

1

On the **Plant Identification** tab, click **Preprocess** and select Remove Offset. The **Remove Offset** tab opens, displaying time plots of the response data and corresponding input signal.

2

Select **Remove offset from signal** and choose the response, Output (y). In the **Offset to remove** text box, specify a value of 2. You can also select the signal initial value or signal mean, or enter a numerical value. The plot updates with an additional trace showing the signal with the offset applied.
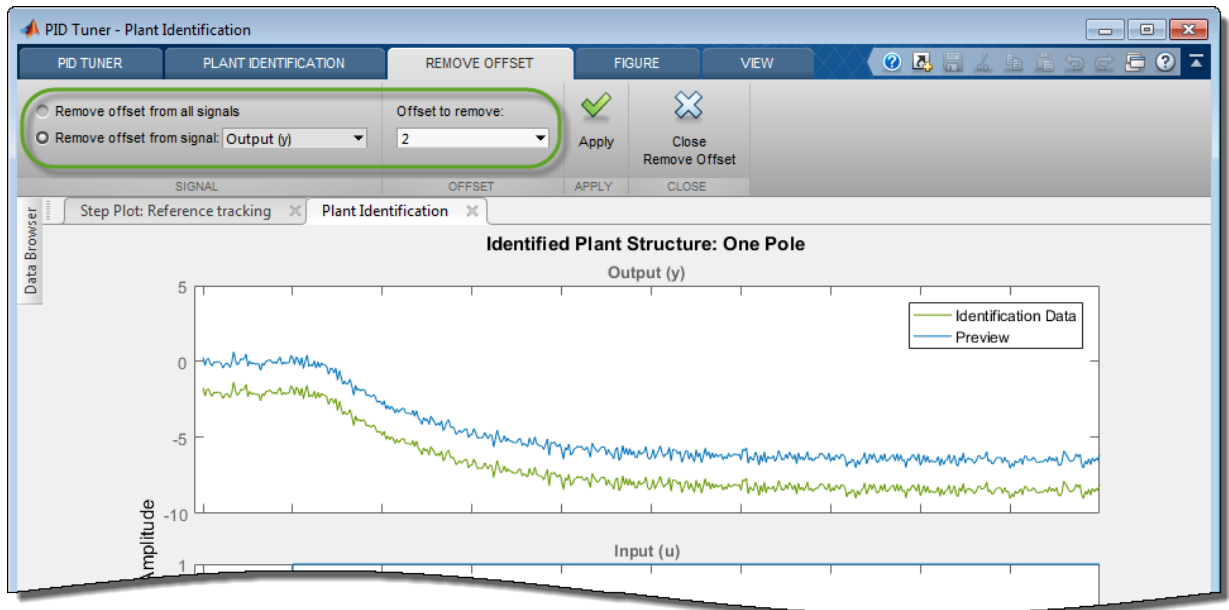
3

Click **Apply** to save the change to the signal. Click **Close Remove Offset** to return to the **Plant Identification** tab.

PID Tuner automatically adjusts the plant parameters to create a new initial guess for the plant based on the preprocessed response signal.

Adjust Plant Structure and Parameters

PID Tuner allows you to specify a plant structure, such as **One Pole**, **Underdamped Pair**, or **State-Space Model**. In the **Structure** menu, choose the plant structure that best matches your response. You can also add a transfer delay, a zero, or an integrator to

your plant. For this example, the one-pole structure gives the qualitatively correct response. You can make further adjustments to the plant structure and parameter values to make the estimated system's response a better match to the measured response data.

PID Tuner gives you several ways to adjust the plant parameters:

- Graphically adjust the response of the estimated system by dragging the adjustors on the plot. In this example, drag the red x to adjust the estimated plant time constant. **PID Tuner** recalculates system parameters as you do so. As you change the estimated system's response, it becomes apparent that there is some time delay between the application of the step input at t = 5 s, and the response of the system to that step input.

To add a transport delay to the estimated plant model, in the **Plant Structure** section, check **Delay**. A vertical line appears on the plot, indicating the current value of the delay. Drag the line left or right to change the delay, and make further adjustments to the system response by dragging the red x.

- Adjust the numerical values of system parameters such as gains, time constants, and time delays. To numerically adjust the values of system parameters, click ⊤⧾⊤ **Edit Parameters**.

  Suppose that you know from an independent measurement that the transport delay in your system is 1.5 seconds. In the **Plant Parameters** dialog box, enter 1.5 for τ. Check **Fix** to fix the parameter value. When you check **Fix** for a parameter, neither graphical nor automatic adjustments to the estimated plant model affect that parameter value.
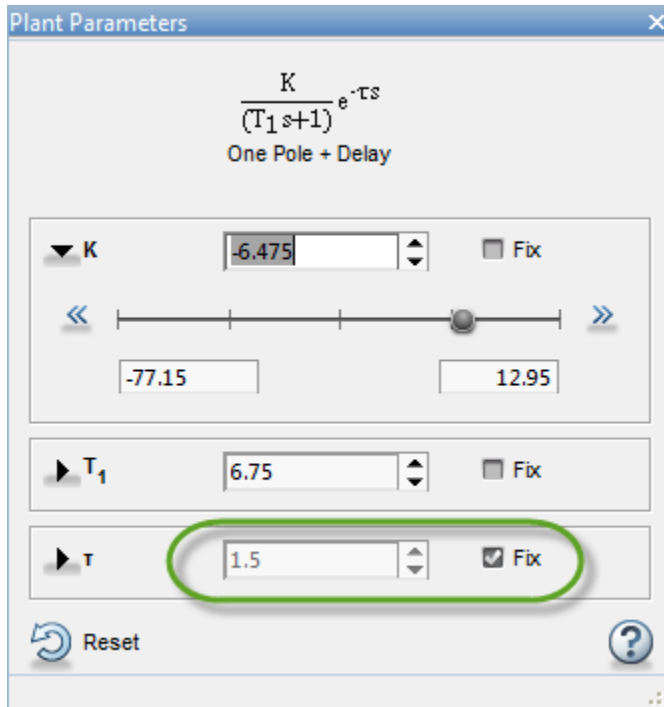
- Automatically optimize the system parameters to match the measured response data.

  Click ⧾⧾ **Auto Estimate** to update the estimated system parameters using the current values as an initial guess.

- Graphically adjust the response of the estimated system by dragging the adjustors on the plot. In this example, drag the red x to adjust the estimated plant time constant.

PID Tuner recalculates system parameters as you do so. As you change the estimated system's response, it becomes apparent that there is some time delay between the application of the step input at t = 5 s, and the response of the system to that step input.

To add a transport delay to the estimated plant model, in the **Plant Structure** section, check **Delay**. A vertical line appears on the plot, indicating the current value of the delay. Drag the line left or right to change the delay, and make further adjustments to the system response by dragging the red x.

- Adjust the numerical values of system parameters such as gains, time constants, and

  time delays. To numerically adjust the values of system parameters, click ⊞ **Edit Parameters**.

Suppose that in this example you know from an independent measurement that the transport delay in your system is 1.5 s. In the **Plant Parameters** dialog box, enter 1.5 for τ. To fix the delay value, check **Fix**. When you check **Fix** for a parameter, neither

graphical nor automatic adjustments to the estimated plant model affect the parameter value.

- Automatically optimize the system parameters to match the measured response data.

  Click ![icon] **Auto Estimate** to update the estimated system parameters using the current values as an initial guess.

You can continue to iterate using any of these methods to adjust plant structure and parameter values until the response of the estimated system adequately matches the measured response.

Save Plant and Tune PID Controller

When you are satisfied with the fit, click ![check] **Apply**. Doing so saves the estimated plant, Plant1, to the **PID Tuner** workspace. **PID Tuner** automatically designs a PI controller for Plant1 and, in the **Step Plot: Reference Tracking** plot, displays a new closed-loop

response. The **Plant** menu reflects that `Plant1` is selected for the current controller design.
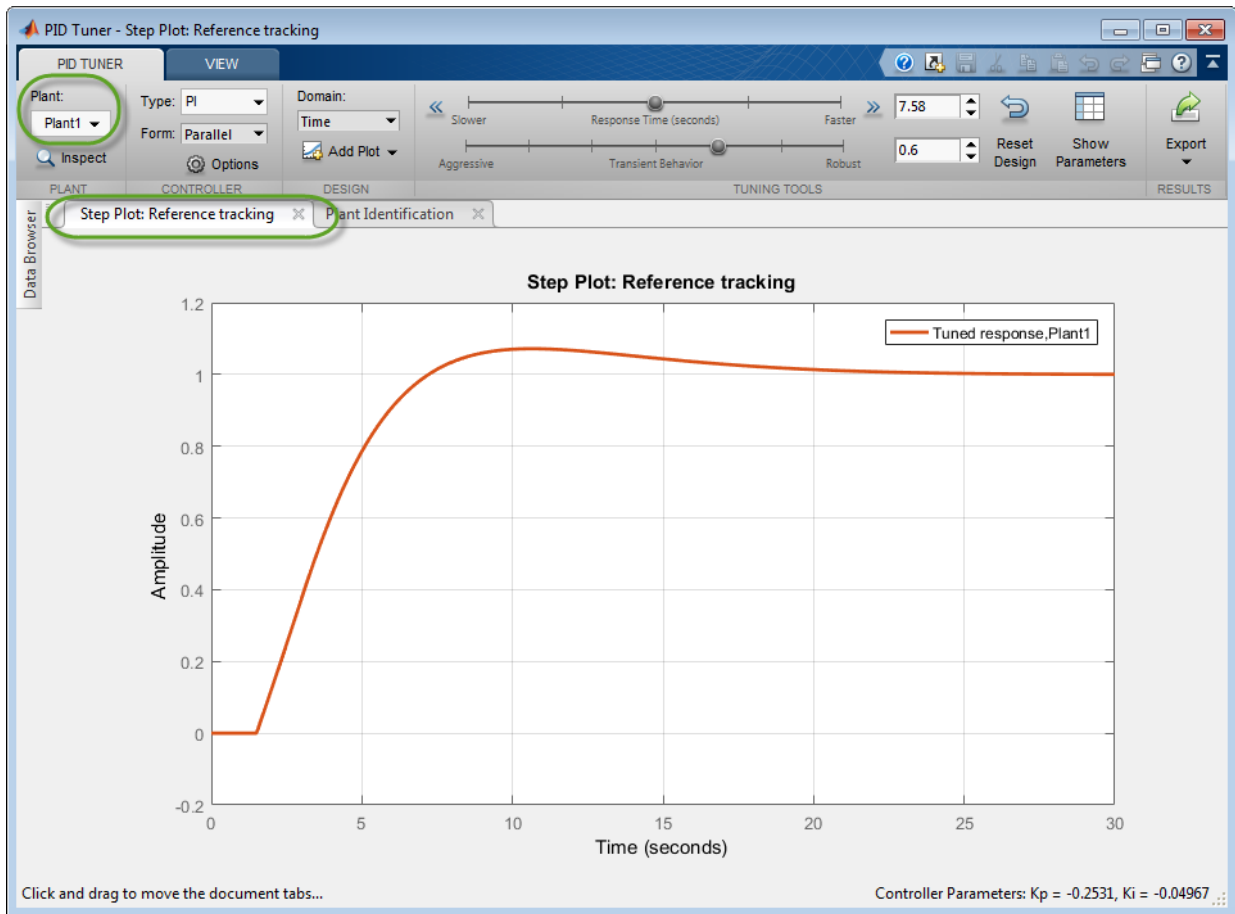
Tip To examine variables stored in the **PID Tuner** workspace, open the **Data Browser**.
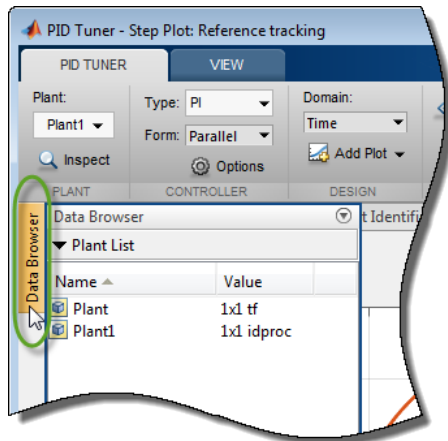
You can now use the **PID Tuner** tools to refine the controller design for the estimated plant and examine tuned system responses.

You can also export the identified plant from the **PID Tuner** workspace to the MATLAB workspace for further analysis. On the **PID Tuner** tab, click **Export**. Check the plant model you want to export to the MATLAB workspace. For this example, export `Plant1`, the plant you identified from response data. You can also export the tuned PID controller.

Click ✔ **OK**. The models you selected are saved to the MATLAB workspace.

Identified plant models are saved as identified LTI models, such as `idproc` or `idss`.

Tip Alternatively, right-click a plant in the **Data Browser** to select it for tuning or export it to the MATLAB workspace.
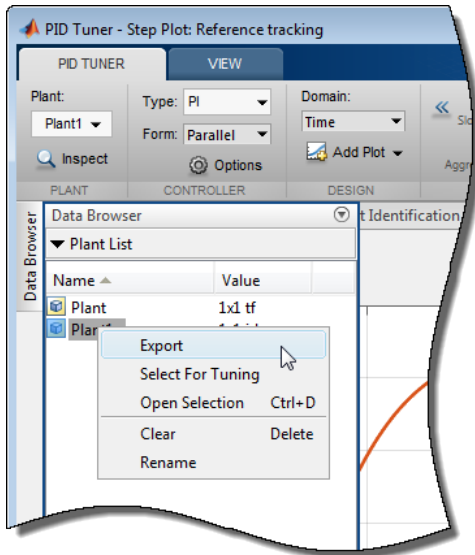
See Also

More About

- "Input/Output Data for Identification" on page 4-70
- "Preprocess Data" on page 4-60
- "Choosing Identified Plant Structure" on page 4-71
- "System Identification for PID Control" on page 4-66

Preprocess Data

Ways to Preprocess Data

In **PID Tuner**, you can preprocess plant data before you use it for estimation. After you import I/O data, on the **Plant Identification** tab, use the **Preprocess** menu to select a preprocessing operation.
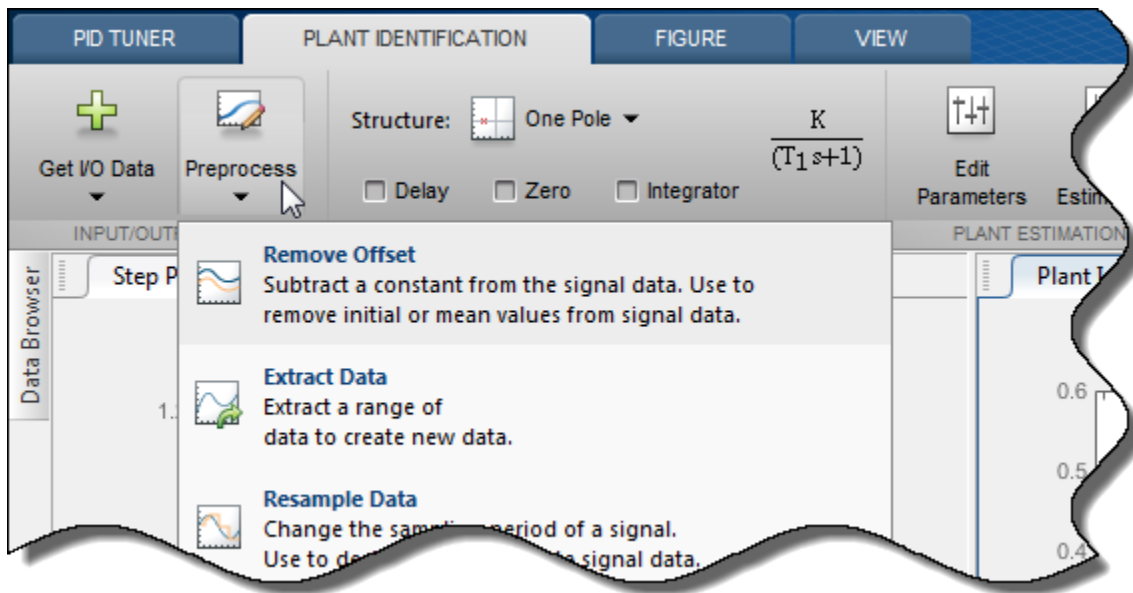
- "Remove Offset" on page 4-61 — Remove mean values, a constant value, or an initial value from the data.

- "Scale Data" on page 4-61 — Scale data by a constant value, signal maximum value, or signal initial value.

- "Extract Data" on page 4-62 — Select a subset of the data to use in the identification. You can graphically select the data to extract, or enter start and end times in the text boxes.

- "Filter Data" on page 4-62 — Process data using a low-pass, high-pass, or band-pass filter.

- "Resample Data" on page 4-62 –– Resample data using zero-order hold or linear interpolation.
- "Replace Data" on page 4-63 –– Replace data with a constant value, region initial value, region final value, or a line. You can use this functionality to replace outliers.

You can perform as many preprocessing operations on your data as are required for your application. For instance, you can both filter the data and remove an offset.

Remove Offset

It is important for good identification results to remove data offsets. In the **Remove Offset** tab, you can remove offset from all signals at once or select a particular signal using the **Remove offset from signal** drop down list. Specify the value to remove using the **Offset to remove** drop down list. The options are:

- A constant value. Enter the value in the box. (Default: 0)
- Mean of the data, to create zero-mean data.
- Signal initial value.

As you change the offset value, the modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking .

Scale Data

In the **Scale Data** tab, you can choose to scale all signals or specify a signal to scale. Select the scaling value from the **Scale to use** drop-down list. The options are:

- A constant value. Enter the value in the box. (Default: 1)
- Signal maximum value.
- Signal initial value.

As you change the scaling, the modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking .

Extract Data

Select a subset of data to use in **Extract Data** tab. You can extract data graphically or by specifying start time and end time. To extract data graphically, click and drag the vertical bars to select a region of the data to use.

Data outside the region that is highlighted yellow is discarded when you click .

Filter Data

You can filter your data using a low-pass, high-pass, or band-pass filter. A low-pass filter blocks high frequency signals, a high-pass filter blocks low frequency signals, and a band-pass filter combines the properties of both low- and high-pass filters.

On the **Low-Pass Filter**, **High-Pass Filter**, or **Band-Pass Filter** tab, you can choose to filter all signals or specify a particular signal. For the low-pass and high-pass filtering, you can specify the normalized cutoff frequency of the signal. Where, a normalized frequency of 1 corresponds to half the sampling rate. For the band-pass filter, you can specify the normalized start and end frequencies. Specify the frequencies by either entering the value in the associated field on the tab. Alternatively, you can specify filter frequencies graphically, by dragging the vertical bars in the frequency-domain plot of your data.

Click **Options** to specify the filter order, and select zero-phase shift filter.

After making choices, update the existing data with the preprocessed data by clicking .

Resample Data

In the **Resample Data** tab, specify the sampling period using the **Resample with sample period:** field. You can resample your data using one of the following interpolation methods:

- Zero-order hold — Fill the missing data sample with the data value immediately preceding it.
- Linear interpolation — Fill the missing data using a line that connects the two data points.

By default, the resampling method is set to `zero-order hold`. You can select the `linear interpolation` method from the **Resample Using** drop-down list.

The modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking .

Replace Data

In the **Replace Data** tab, select data to replace by dragging across a region in the plot. Once you select data, choose how to replace it using the **Replace selected data** drop-down list. You can replace the data you select with one of these options:

- A constant value
- Region initial value
- Region final value
- A line

The replaced preview data changes color and the replacement data appears on the plot. At any time before updating, click **Clear preview** to clear the data you replaced and start over.

After making choices, update the existing data with the preprocessed data by clicking .

Replace Data can be useful, for example, to replace outliers. Outliers can be defined as data values that deviate from the mean by more than three standard deviations. When estimating parameters from data containing outliers, the results may not be accurate. Hence, you might choose to replace the outliers in the data before you estimate the parameters.

See Also

More About
- "Input/Output Data for Identification" on page 4-70

PID Tuning Algorithm

Typical PID tuning objectives include:

- Closed-loop stability — The closed-loop system output remains bounded for bounded input.
- Adequate performance — The closed-loop system tracks reference changes and suppresses disturbances as rapidly as possible. The larger the loop bandwidth (the frequency of unity open-loop gain), the faster the controller responds to changes in the reference or disturbances in the loop.
- Adequate robustness — The loop design has enough gain margin and phase margin to allow for modeling errors or variations in system dynamics.

MathWorks algorithm for tuning PID controllers meets these objectives by tuning the PID gains to achieve a good balance between performance and robustness. By default, the algorithm chooses a crossover frequency (loop bandwidth) based on the plant dynamics, and designs for a target phase margin of 60°. When you interactively change the response time, bandwidth, transient response, or phase margin using the **PID Tuner** interface, the algorithm computes new PID gains.

For a given robustness (minimum phase margin), the tuning algorithm chooses a controller design that balances the two measures of performance, reference tracking and disturbance rejection. You can change the design focus to favor one of these performance measures. To do so, use the `DesignFocus` option of `pidtune` at the command line or the **Options** dialog box in **PID Tuner**.

When you change the design focus, the algorithm attempts to adjust the gains to favor either reference tracking or disturbance rejection, while achieving the same minimum phase margin. The more tunable parameters there are in the system, the more likely it is that the PID algorithm can achieve the desired design focus without sacrificing robustness. For example, setting the design focus is more likely to be effective for PID controllers than for P or PI controllers. In all cases, fine-tuning the performance of the system depends strongly on the properties of your plant. For some plants, changing the design focus has little or no effect.

System Identification for PID Control

Plant Identification

In many situations, a dynamic representation of the system you want to control is not readily available. One solution to this problem is to obtain a dynamical model using identification techniques. The system is excited by a measurable signal and the corresponding response of the system is collected at some sample rate. The resulting input-output data is then used to obtain a model of the system such as a transfer function or a state-space model. This process is called system identification or estimation. The goal of system identification is to choose a model that yields the best possible fit between the measured system response to a particular input and the model's response to the same input.

If you have a Simulink model of your control system, you can simulate input/output data instead of measuring it. The process of estimation is the same. The system response to some known excitation is simulated, and a dynamical model is estimated based upon the resulting simulated input/output data.

Whether you use measured or simulated date for estimation, once a suitable plant model is identified, you impose control objectives on the plant based on your knowledge of the desired behavior of the system that the plant model represents. You then design a feedback controller to meet those objectives.

If you have System Identification Toolbox software, you can use **PID Tuner** for both plant identification and controller design in a single interface. You can import input/output data and use it to identify one or more plant models. Or, you can obtain simulated input/output data from a Simulink model and use that to identify one or more plant models. You can then design and verify PID controllers using these plants. **PID Tuner** also allows you to directly import plant models, such as one you have obtained from an independent identification task.

For an overview of system identification, see About System Identification (System Identification Toolbox).

Linear Approximation of Nonlinear Systems for PID Control

The dynamical behavior of many systems can be described adequately by a linear relationship between the system's input and output. Even when behavior becomes

nonlinear in some operating regimes, there are often regimes in which the system dynamics are linear. For example, the behavior of an operational amplifier or the lift-vs-force dynamics of aerodynamic bodies can be described by linear models, within a certain limited operating range of inputs. For such a system, you can perform an experiment (or a simulation) that excites the system only in its linear range of behavior and collect the input/output data. You can then use the data to estimate a linear plant model, and design a PID controller for the linear model.

In other cases, the effects of nonlinearities are small. In such a case, a linear model can provide a good approximation, such that the nonlinear deviations are treated as disturbances. Such approximations depend heavily on the input profile, the amplitude and frequency content of the excitation signal.

Linear models often describe the deviation of the response of a system from some equilibrium point, due to small perturbing inputs. Consider a nonlinear system whose output, $y(t)$, follows a prescribed trajectory in response to a known input, $u(t)$. The dynamics are described by $dx(t)/dt = f(x, u)$, $y = g(x,u)$. Here, x is a vector of internal states of the system, and y is the vector of output variables. The functions f and g, which can be nonlinear, are the mathematical descriptions of the system and measurement dynamics. Suppose that when the system is at an equilibrium condition, a small perturbation to the input, Δu, leads to a small perturbation in the output, Δy:

$$\Delta \dot{x} = \frac{\partial f}{\partial x} \Delta x + \frac{\partial f}{\partial u} \Delta u,$$

$$\Delta y = \frac{\partial g}{\partial x} \Delta x + \frac{\partial g}{\partial u} \Delta u.$$

For example, consider the system of the following Simulink block diagram:
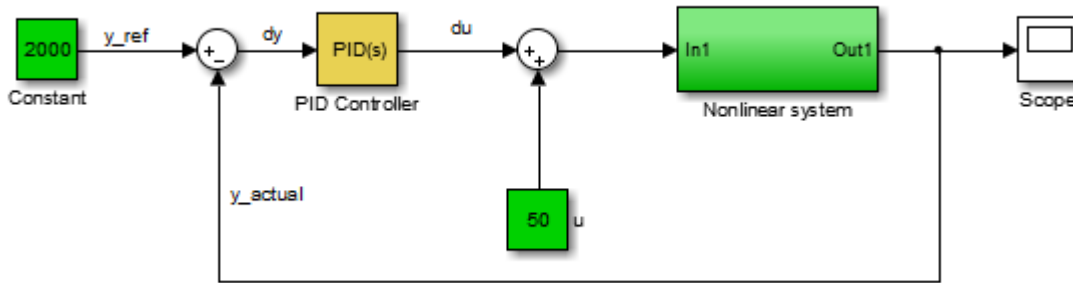
When operating in a disturbance-free environment, the nominal input of value 50 keeps the plant along its constant trajectory of value 2000. Any disturbances would cause the plant to deviate from this value. The PID Controller's task is to add a small correction to the input signal that brings the system back to its nominal value in a reasonable amount of time. The PID Controller thus needs to work only on the linear deviation dynamics even though the actual plant itself might be nonlinear. Thus, you might be able to achieve effective control over a nonlinear system in some regimes by designing a PID controller for a linear approximation of the system at equilibrium conditions.

Linear Process Models

A common use case is designing PID controllers for the steady-state operation of manufacturing plants. In these plants, a model relating the effect of a measurable input variable on an output quantity is often required in the form of a SISO plant. The overall system may be MIMO in nature, but the experimentation or simulation is carried out in a way that makes it possible to measure the incremental effect of one input variable on a selected output. The data can be quite noisy, but since the expectation is to control only the dominant dynamics, a low-order plant model often suffices. Such a proxy is obtained by collecting or simulating input-output data and deriving a process model (low order transfer function with unknown delay) from it. The excitation signal for deriving the data can often be a simple bump in the value of the selected input variable.

Advanced System Identification Tasks

In **PID Tuner**, you can only identify single-input, single output, continuous-time plant models. Additionally, **PID Tuner** cannot perform the following system identification tasks:

- Identify transfer functions of arbitrary number of poles and zeros. (**PID Tuner** can identify transfer functions up to three poles and one zero, plus an integrator and a time delay. **PID Tuner** can identify state-space models of arbitrary order.)
- Estimate the disturbance component of a model, which can be useful for separating measured dynamics from noise dynamics.
- Validate estimation by comparing the plant response against an independent dataset.
- Perform residual analysis.

If you need these enhanced identification features, import your data into the **System Identification** app (**System Identification**). Use the **System Identification** app to perform model identification and export the identified model to the MATLAB workspace. Then import the identified model into **PID Tuner** for PID controller design.

For more information about the System Identification Tool, see "Identify Linear Models Using System Identification App" (System Identification Toolbox).

See Also
System Identification

More About
- "Input/Output Data for Identification" on page 4-70
- "Choosing Identified Plant Structure" on page 4-71
- "Interactively Estimate Plant Parameters from Response Data" on page 4-48

Input/Output Data for Identification

Data Preparation

Identification of a plant model for PID tuning requires a single-input, single-output data set.

If you have measured data, use the data import dialogs to bring in identification data. Some common sources of identification data are transient tests such as bump test and impact test. For such data, **PID Tuner** provides dedicated dialogs that require you to specify data for only the output signal while characterizing the input by its shape. For an example, see "Interactively Estimate Plant Parameters from Response Data" on page 4-48.

If you want to obtain input/output data by simulating a Simulink model, the **PID Tuner** interface lets you specify the shape of the input stimulus used to generate the response. For an example, see the Simulink Control Design™ example "Design a PID Controller Using Simulated I/O Data."

Data Preprocessing

PID Tuner lets you preprocess your imported or simulated data. **PID Tuner** provides various options for detrending, scaling, and filtering the data.

It is strongly recommended to remove any equilibrium-related signal offsets from the input and output signals before proceeding with estimation. You can also filter the data to focus the signal contents to the frequency band of interest.

Some data processing actions can alter the nature of the data, which can result in transient data (step, impulse or wide pulse responses) to be treated as arbitrary input/output data. When that happens the identification plot does not show markers for adjusting the model time constants and damping coefficient.

For an example that includes a data-preprocessing step, see: "Interactively Estimate Plant Parameters from Response Data" on page 4-48.

For further information about data-preprocessing options, see "Preprocess Data" on page 4-60.

Choosing Identified Plant Structure

PID Tuner provides two types of model structures for representing the plant dynamics: process models and state-space models.

Use your knowledge of system characteristics and the level of accuracy required by your application to pick a model structure. In absence of any prior information, you can gain some insight into the order of dynamics and delays by analyzing the experimentally obtained step response and frequency response of the system. For more information see the following in the System Identification Toolbox documentation:

- "Correlation Models" (System Identification Toolbox)
- "Frequency-Response Models" (System Identification Toolbox)

Each model structure you choose has associated dynamic elements, or model parameters. You adjust the values of these parameters manually or automatically to find an identified model that yields a satisfactory match to your measured or simulated response data. In many cases, when you are unsure of the best structure to use, it helps to start with the simplest model structure, transfer function with one pole. You can progressively try identification with higher-order structures until a satisfactory match between the plant response and measured output is achieved. The state-space model structure allows an automatic search for optimal model order based on an analysis of the input-output data.

When you begin the plant identification task, a transfer function model structure with one real pole is selected by default. This default set up is not sensitive to the nature of the data and may not be a good fit for your application. It is therefore recommended that you choose a suitable model structure before performing parameter identification.

Process Models

Process models are transfer functions with 3 or fewer poles, and can be augmented by addition of zero, delay and integrator elements. Process models are parameterized by model parameters representing time constants, gain, and time delay. In **PID Tuner**, choose a process model in the **Plant Identification** tab using the **Structure** menu.
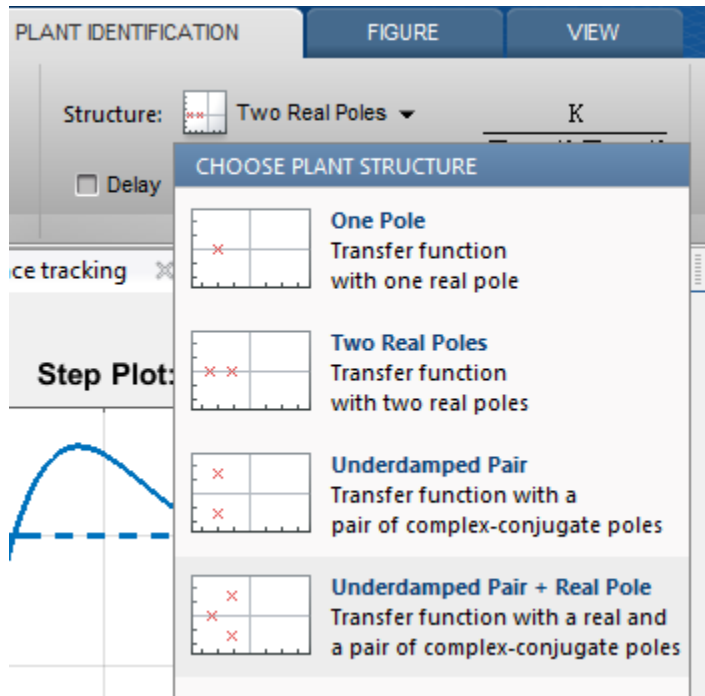
For any chosen structure you can optionally add a delay, a zero and/or an integrator element using the corresponding checkboxes. The model transfer function configured by these choices is displayed next to the **Structure** menu.
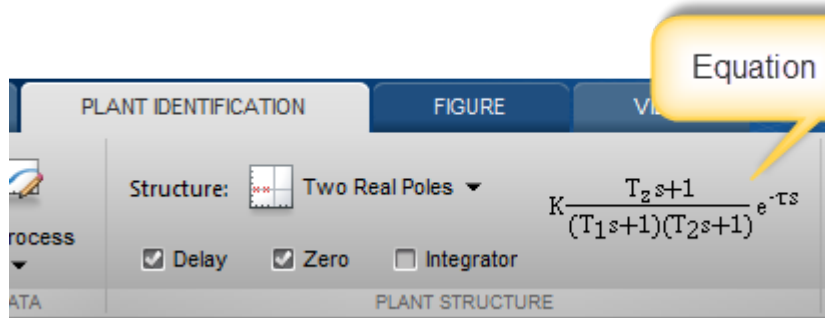
The simplest available process model is a transfer function with one real pole and no zero or delay elements:

$$H(s) = \frac{K}{T_1 s + 1}.$$

This model is defined by the parameters K, the gain, and T_1, the first time constant. The most complex process-model structure choose has three poles, an additional integrator, a zero, and a time delay, such as the following model, which has one real pole and one complex conjugate pair of poles:

$$H(s) = K \frac{T_z s + 1}{s(T_1 s + 1)\left(T_\omega^2 s^2 + 2\zeta T_\omega s + 1\right)} e^{-\tau s}.$$

In this model, the configurable parameters include the time constants associated with the poles and the zero, T_1, T_ω, and T_z. The other parameters are the damping coefficient ζ, the gain K, and the time delay τ.

When you select a process model type, **PID Tuner** automatically computes initial values for the plant parameters and displays a plot showing both the estimated model response and your measured or simulated data. You can edit the parameter values graphically using indicators on the plot, or numerically using the Plant Parameters editor. For an example illustrating this process, see "Interactively Estimate Plant Parameters from Response Data" on page 4-48.

The following table summarizes the various parameters that define the available types of process models.

Parameter	Used By	Description
K — Gain	All transfer functions	Can take any real value. In the plot, drag the plant response curve (blue) up or down to adjust K.
T_1 — First time constant	Transfer function with one or more real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the red x left (towards zero) or right (towards T) to adjust T_1.

Parameter	Used By	Description
T_2— Second time constant	Transfer function with two real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the magenta x left (towards zero) or right (towards T) to adjust T_2.
T_ω — Time constant associated with the natural frequency ω_n, where $T_\omega = 1/\omega_n$	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust T_ω.
ζ — Damping coefficient	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and 1. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust ζ.
τ — Transport delay	Any transfer function	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the orange vertical bar left (towards zero) or right (towards T) to adjust τ.

Parameter	Used By	Description
T_z — Model zero	Any transfer function	Can take any value between –T and T, the time span of measured or simulated data. In the plot, drag the red circle left (towards –T) or right (towards T) to adjust T_z.
Integrator	Any transfer function	Adds a factor of 1/s to the transfer function. There is no associated parameter to adjust.

State-Space Models

The state-space model structure for identification is primarily defined by the choice of number of states, the model order. Use the state-space model structure when higher order models than those supported by process model structures are required to achieve a satisfactory match to your measured or simulated I/O data. In the state-space model structure, the system dynamics are represented by the state and output equations:

$$\dot{x} = Ax + Bu,$$
$$y = Cx + Du.$$

x is a vector of state variables, automatically chosen by the software based on the selected model order. u represents the input signal, and y the output signals.

To use a state-space model structure, in the **Plant Identification** tab, in the **Structure** menu, select `State-Space Model`. Then click **Configure Structure** to open the **State-Space Model Structure** dialog box.

State Space Model Structure   ✕

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

Plant Order:

○ Specify value: |1|

○ Pick best value in the range: |1:10|

Input delay: |0|

☐ Allow feedthrough (nonzero D matrix)

                                                    ⑦
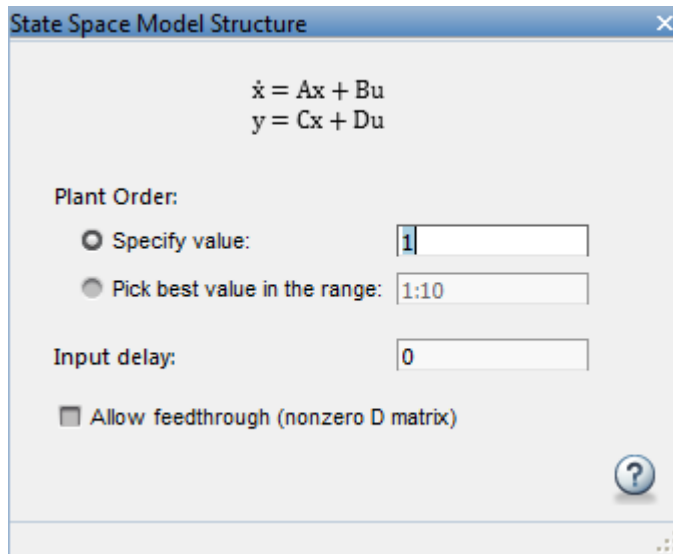
Use the dialog box to specify model order, delay and feedthrough characteristics. If you are unsure about the order, select **Pick best value in the range**, and enter a range of orders. In this case, when you click **Estimate** in the **Plant Estimation** tab, the software displays a bar chart of Hankel singular values. Choose a model order equal to the number of Hankel singular values that make significant contributions to the system dynamics.

When you choose a state-space model structure, the identification plot shows a plant response (blue) curve only if a valid estimated model exists. For example, if you change structure after estimating a process model, the state-space equivalent of the estimated model is displayed. If you change the model order, the plant response curve disappears until a new estimation is performed.

When using the state-space model structure, you cannot directly interact with the model parameters. The identified model should thus be considered unstructured with no physical meaning attached to the state variables of the model.

However, you can graphically adjust the input delay and the overall gain of the model. When you select a state-space model with a time delay, the delay is represented on the plot by a vertical orange bar is shown on the plot. Drag this bar horizontally to change the delay value. Drag the plant response (blue) curve up and down to adjust the model gain.

Existing Plant Models

Any previously imported or identified plant models are listed the **Plant List** section of the Data Browser.
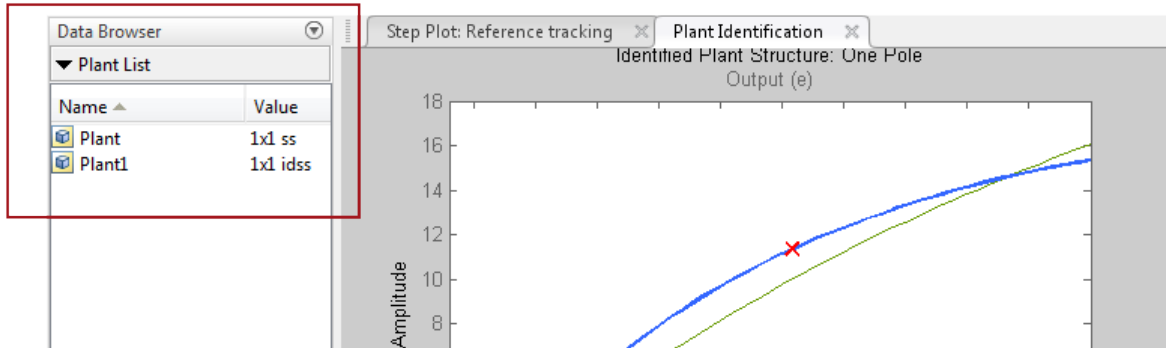
You can define the model structure and initialize the model parameter values using one of these plants. To do so, in the **Plant Identification** tab, in the **Structure** menu, select the linear plant model you want to use for structure an initialization.
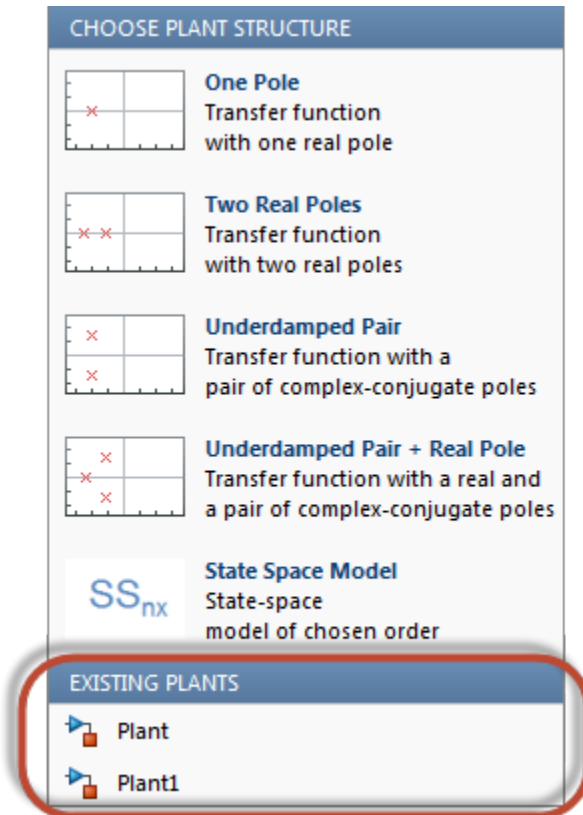
If the plant you select is a process model (`idproc` object), **PID Tuner** uses its structure. If the plant is any other model type, **PID Tuner** uses the state-space model structure.

Switching Between Model Structures

When you switch from one model structure to another, the software preserves the model characteristics (pole/zero locations, gain, delay) as much as possible. For example, when you switch from a one-pole model to a two-pole model, the existing values of T_1, T_z, τ and K are retained, T_2 is initialized to a default (or previously assigned, if any) value.

Estimating Parameter Values

Once you have selected a model structure, you have several options for manually or automatically adjusting parameter values to achieve a good match between the estimated model response and your measured or simulated input/output data. For an example that illustrates all these options, see:

- "Interactively Estimate Plant Parameters from Response Data" on page 4-48 (Control System Toolbox)
- "Interactively Estimate Plant from Measured or Simulated Response Data" (Simulink Control Design) Simulink Control Design)

PID Tuner does not perform a smart initialization of model parameters when a model structure is selected. Rather, the initial values of the model parameters, reflected in the plot, are arbitrarily-chosen middle of the range values. If you need a good starting point before manually adjusting the parameter values, use the **Initialize and Estimate** option from the **Plant Identification** tab.

Handling Initial Conditions

In some cases, the system response is strongly influenced by the initial conditions. Thus a description of the input to output relationship in the form of a transfer function is insufficient to fit the observed data. This is especially true of systems containing weakly damped modes. **PID Tuner** allows you to estimate initial conditions in addition to the model parameters such that the sum of the initial condition response and the input response matches the observed output well. Use the **Estimation Options** dialog box to specify how the initial conditions should be handled during automatic estimation. By default, the initial condition handling (whether to fix to zero values or to estimate) is automatically performed by the estimation algorithm. However, you can enforce a certain choice by using the Initial Conditions menu.

Initial conditions can only be estimated with automatic estimation. Unlike the model parameters, they cannot be modified manually. However, once estimated they remain fixed to their estimated values, unless the model structure is changed or new identification data is imported.

If you modify the model parameters after having performed an automatic estimation, the model response will show a fixed contribution (i.e., independent of model parameters) from initial conditions. In the following plot, the effects of initial conditions were identified to be particularly significant. When the delay is adjusted afterwards, the portion

of the response to the left of the input delay marker (the τ Adjustor) comes purely from initial conditions. The portion to the right of the τ Adjustor contains the effects of both the input signal as well as the initial conditions.
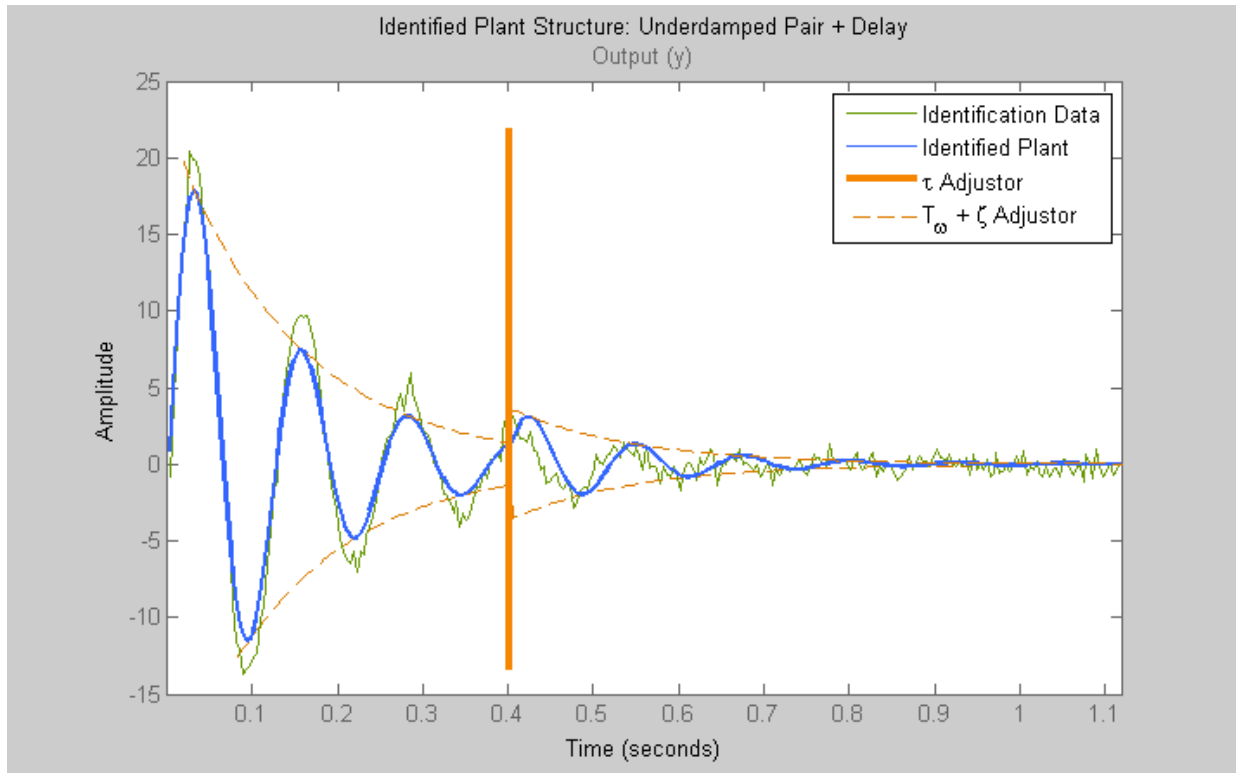
See Also

More About

- "System Identification for PID Control" on page 4-66
- "Interactively Estimate Plant Parameters from Response Data" on page 4-48

Pole Placement

Closed-loop pole locations have a direct impact on time response characteristics such as rise time, settling time, and transient oscillations. Root locus uses compensator gains to move closed-loop poles to achieve design specifications for SISO systems. You can, however, use state-space techniques to assign closed-loop poles. This design technique is known as *pole placement*, which differs from root locus in the following ways:

- Using pole placement techniques, you can design dynamic compensators.
- Pole placement techniques are applicable to MIMO systems.

Pole placement requires a state-space model of the system (use `ss` to convert other model formats to state space). In continuous time, such models are of the form

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where u is the vector of control inputs, x is the state vector, and y is the vector of measurements.

State-Feedback Gain Selection

Under state feedback $u = -Kx$, the closed-loop dynamics are given by

$$\dot{x} = (A - BK)x$$

and the closed-loop poles are the eigenvalues of *A-BK*. Using the `place` function, you can compute a gain matrix *K* that assigns these poles to any desired locations in the complex plane (provided that (*A*,*B*) is controllable).

For example, for state matrices *A* and *B*, and vector *p* that contains the desired locations of the closed loop poles,

```
K = place(A,B,p);
```

computes an appropriate gain matrix *K*.

State Estimator Design

You cannot implement the state-feedback law $u = -Kx$ unless the full state x is measured.

However, you can construct a state estimate ξ such that the law $u = -K\xi$ retains similar pole assignment and closed-loop properties. You can achieve this by designing a state estimator (or observer) of the form

$$\dot{\xi} = A\xi + Bu + L(y - C\xi - Du)$$

The estimator poles are the eigenvalues of *A-LC*, which can be arbitrarily assigned by proper selection of the estimator gain matrix *L*, provided that (*C, A*) is observable. Generally, the estimator dynamics should be faster than the controller dynamics (eigenvalues of *A-BK*).

Use the `place` function to calculate the *L* matrix

```
L = place(A',C',q).'
```

where `A` and `C` are the state and output matrices, and `q` is the vector containing the desired closed-loop poles for the observer.

Replacing *x* by its estimate ξ in $u = -Kx$ yields the dynamic output-feedback compensator

$$\dot{\xi} = [A - LC - (B - LD)K]\xi + Ly$$
$$u = -K\xi$$

Note that the resulting closed-loop dynamics are

$$\begin{bmatrix} \dot{x} \\ \dot{e} \end{bmatrix} = \begin{bmatrix} A - BK & BK \\ 0 & A - LC \end{bmatrix} \begin{bmatrix} x \\ e \end{bmatrix}, \quad e = x - \xi$$

Hence, you actually assign all closed-loop poles by independently placing the eigenvalues of *A-BK* and *A-LC*.

Example

Given a continuous-time state-space model

```
sys_pp = ss(A,B,C,D)
```

with seven outputs and four inputs, suppose you have designed

- A state-feedback controller gain K using inputs 1, 2, and 4 of the plant as control inputs
- A state estimator with gain L using outputs 4, 7, and 1 of the plant as sensors
- Input 3 of the plant as an additional known input

You can then connect the controller and estimator and form the dynamic compensator using this code:

```
controls = [1,2,4];
sensors = [4,7,1];
known = [3];
regulator = reg(sys_pp,K,L,sensors,known,controls)
```

Pole Placement Tools

You can use functions to

- Compute gain matrices K and L that achieve the desired closed-loop pole locations.
- Form the state estimator and dynamic compensator using these gains.

The following table summarizes the functions for pole placement.

Functions	Description
estim	Form state estimator given estimator gain
place	Pole placement design
reg	Form output-feedback compensator given state-feedback and estimator gains

Caution

Pole placement can be badly conditioned if you choose unrealistic pole locations. In particular, you should avoid:

- Placing multiple poles at the same location.

- Moving poles that are weakly controllable or observable. This typically requires high gain, which in turn makes the entire closed-loop eigenstructure very sensitive to perturbation.

See Also

`estim` | `place` | `reg`

Linear-Quadratic-Gaussian (LQG) Design

Linear-quadratic-Gaussian (LQG) control is a modern state-space technique for designing optimal dynamic regulators and servo controllers with integral action (also known as *setpoint trackers*). This technique allows you to trade off regulation/tracker performance and control effort, and to take into account process disturbances and measurement noise.

To design LQG regulators and setpoint trackers, you perform the following steps:

1   Construct the LQ-optimal gain.

2   Construct a Kalman filter (state estimator).

3   Form the LQG design by connecting the LQ-optimal gain and the Kalman filter.

For more information about using LQG design to create LQG regulators , see "Linear-Quadratic-Gaussian (LQG) Design for Regulation" on page 4-85.

For more information about using LQG design to create LQG servo controllers, see "Linear-Quadratic-Gaussian (LQG) Design of Servo Controller with Integral Action" on page 4-90.

These topics focus on the continuous-time case. For information about discrete-time LQG design, see the `dlqr` and `kalman` reference pages.

Linear-Quadratic-Gaussian (LQG) Design for Regulation

You can design an LQG regulator to regulate the output *y* around zero in the following model.
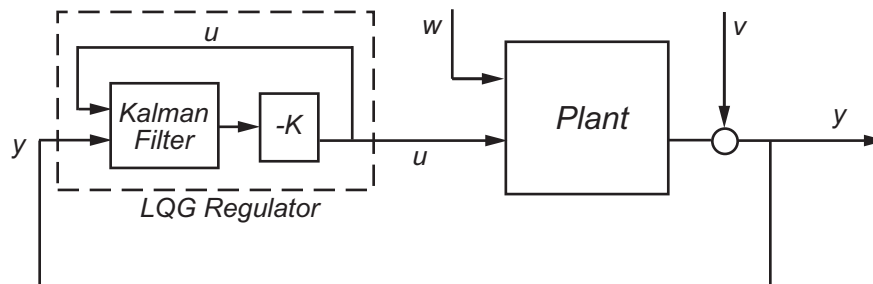
The plant in this model experiences disturbances (process noise) *w* and is driven by controls *u*. The regulator relies on the noisy measurements *y* to generate these controls. The plant state and measurement equations take the form of

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + Hw + v$$

and both *w* and *v* are modeled as white noise.

Note LQG design requires a state-space model of the plant. You can use `ss` to convert other model formats to state space.

To design LQG regulators, you can use the design techniques shown in the following table.

To design an LQG regulator using...	Use the following commands:
A quick, one-step design technique when the following is true: • You need the optimal LQG controller and either $E(wv')$ or H is nonzero. • All known (deterministic) inputs are control inputs and all outputs are measured. • Integrator states are weighted independently of states of plants and control inputs.	`lqg`

To design an LQG regulator using...	Use the following commands:
A more flexible, three-step design technique that allows you to specify: • Arbitrary *G* and *H*. • Known (deterministic) inputs that are not controls and/or outputs that are not measured. • A flexible weighting scheme for integrator states, plant states, and controls.	lqr, kalman, and lqgreg For more information, see • "Constructing the Optimal State-Feedback Gain for Regulation" on page 4-87 • "Constructing the Kalman State Estimator" on page 4-87 • "Forming the LQG Regulator" on page 4-89

Constructing the Optimal State-Feedback Gain for Regulation

You construct the LQ-optimal gain from the following elements:

• State-space system matrices
• Weighting matrices Q, R, and N, which define the tradeoff between regulation performance (how fast $x(t)$ goes to zero) and control effort.

To construct the optimal gain, type the following command:

K= lqr(A,B,Q,R,N)

This command computes the optimal gain matrix K, for which the state feedback law

$u = -Kx$ minimizes the following quadratic cost function for continuous time:

$$J(u) = \int_0^\infty \{x^T Q x + 2x^T N u + u^T R u\} dt$$

The software computes the gain matrix *K* by solving an algebraic Riccati equation.

For information about constructing LQ-optimal gain, including the cost function that the software minimizes for discrete time, see the lqr reference page.
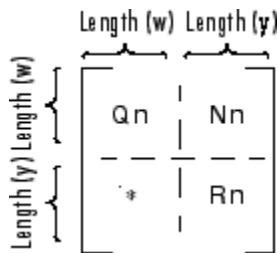
Constructing the Kalman State Estimator

You need a Kalman state estimator for LQG regulation and servo control because you cannot implement optimal LQ-optimal state feedback without full state measurement.

You construct the state estimate $\hat{x}$ such that $u = -K\hat{x}$ remains optimal for the output-feedback problem. You construct the Kalman state estimator gain from the following elements:

- State-space plant model `sys`

- Noise covariance data, `Qn`, `Rn`, and `Nn`

    The following figure shows the required dimensions for `Qn`, `Rn`, and `Nn`. If `Nn` is 0, you can omit it.

Required Dimensions for Qn, Rn, and Nn

Note You construct the Kalman state estimator in the same way for both regulation and servo control.

To construct the Kalman state estimator, type the following command:

```
[kest,L,P] = kalman(sys,Qn,Rn,Nn);
```

This command computes a Kalman state estimator, `kest` with the following plant equations:

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + Hw + v$$

where *w* and *v* are modeled as white noise. `L` is the Kalman gain and `P` the covariance matrix.

The software generates this state estimate using the Kalman filter

$$\frac{d}{dt}\hat{x} = A\hat{x} + Bu + L(y - C\hat{x} - Du)$$

with inputs u (controls) and y (measurements). The noise covariance data

$$E(ww^T) = Q_n, \quad E(vv^T) = R_n, \quad E(wv^T) = N_n$$

determines the Kalman gain L through an algebraic Riccati equation.

The Kalman filter is an optimal estimator when dealing with Gaussian white noise. Specifically, it minimizes the asymptotic covariance

$$\lim_{t \to \infty} E\left((x - \hat{x})(x - \hat{x})^T\right)$$

of the estimation error $x - \hat{x}$.
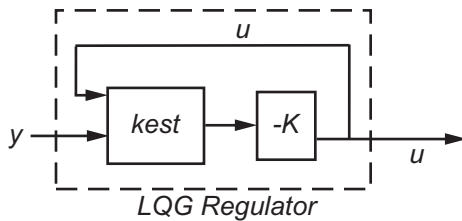

For more information, see the `kalman` reference page. For a complete example of a Kalman filter implementation, see Kalman Filtering.

Forming the LQG Regulator

To form the LQG regulator, connect the Kalman filter `kest` and LQ-optimal gain `K` by typing the following command:

```
regulator = lqgreg(kest, K);
```

This command forms the LQG regulator shown in the following figure.

The regulator has the following state-space equations:

$$\frac{d}{dt}\hat{x} = [A - LC - (B - LD)K]\hat{x} + Ly$$
$$u = -K\hat{x}$$

For more information on forming LQG regulators, see `lqgreg` and "LQG Regulation: Rolling Mill Case Study".

Linear-Quadratic-Gaussian (LQG) Design of Servo Controller with Integral Action

You can design a servo controller with integral action for the following model:

LQG Servo Controller
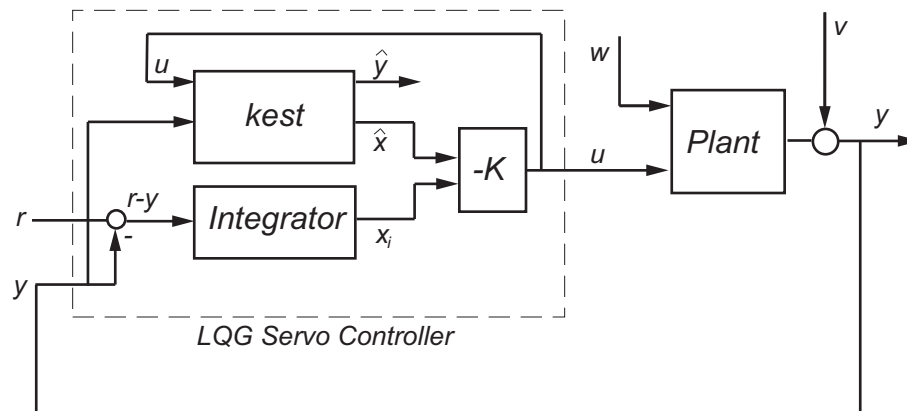
The servo controller you design ensures that the output *y* tracks the reference command *r* while rejecting process disturbances *w* and measurement noise *v*.

The plant in the previous figure is subject to disturbances *w* and is driven by controls *u*. The servo controller relies on the noisy measurements *y* to generate these controls. The plant state and measurement equations are of the form

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + Hw + v$$

and both *w* and *v* are modeled as white noise.

Note LQG design requires a state-space model of the plant. You can use `ss` to convert other model formats to state space.

To design LQG servo controllers, you can use the design techniques shown in the following table.

To design an LQG servo controller using...	Use the following commands:
A quick, one-step design technique when the following is true: • You need the optimal LQG controller and either *E*(*wv*') or *H* is nonzero. • All known (deterministic) inputs are control inputs and all outputs are measured. • Integrator states are weighted independently of states of plants and control inputs.	`lqg`
A more flexible, three-step design technique that allows you to specify: • Arbitrary *G* and *H*. • Known (deterministic) inputs that are not controls and/or outputs that are not measured. • A flexible weighting scheme for integrator states, plant states, and controls.	`lqi`, `kalman`, and `lqgtrack` For more information, see • "Constructing the Optimal State-Feedback Gain for Servo Control" on page 4-91 • "Constructing the Kalman State Estimator" on page 4-92 • "Forming the LQG Servo Control" on page 4-94

Constructing the Optimal State-Feedback Gain for Servo Control

You construct the LQ-optimal gain from the

• State-space plant model `sys`
• Weighting matrices `Q`, `R`, and `N`, which define the tradeoff between tracker performance and control effort

To construct the optimal gain, type the following command:

```
K= lqi(sys,Q,R,N)
```

This command computes the optimal gain matrix K, for which the state feedback law

$u = -Kz = -K[x; x_i]$ minimizes the following quadratic cost function for continuous time:

$$J(u) = \int_0^\infty \left\{ z^T Q z + u^T R u + 2 z^T N u \right\} dt$$

The software computes the gain matrix K by solving an algebraic Riccati equation.

For information about constructing LQ-optimal gain, including the cost function that the software minimizes for discrete time, see the lqi reference page.
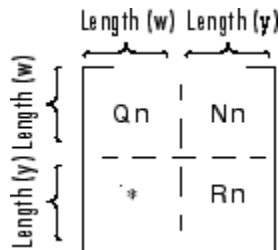
Constructing the Kalman State Estimator

You need a Kalman state estimator for LQG regulation and servo control because you cannot implement LQ-optimal state feedback without full state measurement.

You construct the state estimate $\hat{x}$ such that $u = -K\hat{x}$ remains optimal for the output-feedback problem. You construct the Kalman state estimator gain from the following elements:

- State-space plant model sys
- Noise covariance data, Qn, Rn, and Nn

  The following figure shows the required dimensions for Qn, Rn, and Nn. If Nn is 0, you can omit it.

Required Dimensions for Qn, Rn, and Nn

Note You construct the Kalman state estimator in the same way for both regulation and servo control.

To construct the Kalman state estimator, type the following command:

```
[kest,L,P] = kalman(sys,Qn,Rn,Nn);
```

This command computes a Kalman state estimator, `kest` with the following plant equations:

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + Hw + v$$

where w and v are modeled as white noise. `L` is the Kalman gain and `P` the covariance matrix.

The software generates this state estimate using the Kalman filter

$$\frac{d}{dt}\hat{x} = A\hat{x} + Bu + L(y - C\hat{x} - Du)$$

with inputs u (controls) and y (measurements). The noise covariance data

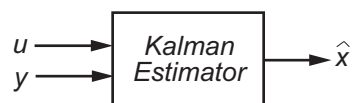$$E(ww^T) = Q_n, \quad E(vv^T) = R_n, \quad E(wv^T) = N_n$$

determines the Kalman gain L through an algebraic Riccati equation.

The Kalman filter is an optimal estimator when dealing with Gaussian white noise. Specifically, it minimizes the asymptotic covariance

$$\lim_{t \to \infty} E\left(\left(x - \hat{x} \right)\left(x - \hat{x} \right)^T \right)$$

of the estimation error $x - \hat{x}$.

For more information, see the `kalman` reference page. For a complete example of a Kalman filter implementation, see Kalman Filtering.

Forming the LQG Servo Control

To form a two-degree-of-freedom LQG servo controller, connect the Kalman filter `kest` and LQ-optimal gain `K` by typing the following command:

```
servocontroller = lqgtrack(kest, K);
```

This command forms the LQG servo controller shown in the following figure.

LQG Servo Controller
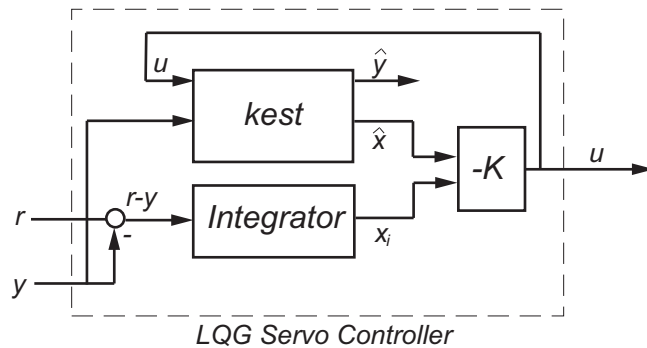
The servo controller has the following state-space equations:

$$
\begin{bmatrix} \dot{\hat{x}} \\ \dot{x}_i \end{bmatrix} = \begin{bmatrix} A - BK_x - LC + LDK_x & -BK_i + LDK_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix} + \begin{bmatrix} 0 & L \\ I & -I \end{bmatrix} \begin{bmatrix} r \\ y \end{bmatrix}
$$

$$
u = \begin{bmatrix} -K_x & -K_i \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix}
$$

For more information on forming LQG servo controllers, including how to form a one-degree-of-freedom LQG servo controller, see the `lqgtrack` reference page.

See Also
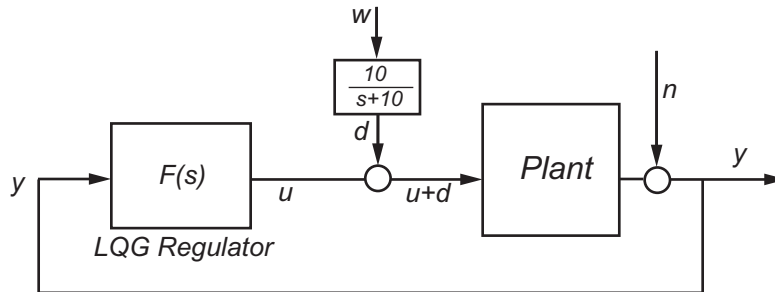
kalman | lqg | lqgreg | lqgtrack | lqi | lqr

Related Examples

- Kalman Filtering
- "Design an LQG Regulator" on page 4-96
- "Design an LQG Servo Controller" on page 4-100
- "Design an LQR Servo Controller in Simulink" on page 4-103

Design an LQG Regulator

As an example of LQG design, consider the following regulation problem.

The goal is to regulate the plant output *y* around zero. The input disturbance *d* is low frequency with power spectral density (PSD) concentrated below 10 rad/s. For LQG design purposes, it is modeled as white noise driving a lowpass filter with a cutoff at 10 rad/s, shown in the following figure.
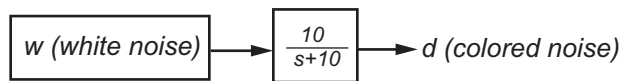
For simplicity, this noise is modeled as Gaussian white noise with variance of 1.

The following figure shows the Bode magnitude of the shaping filter.
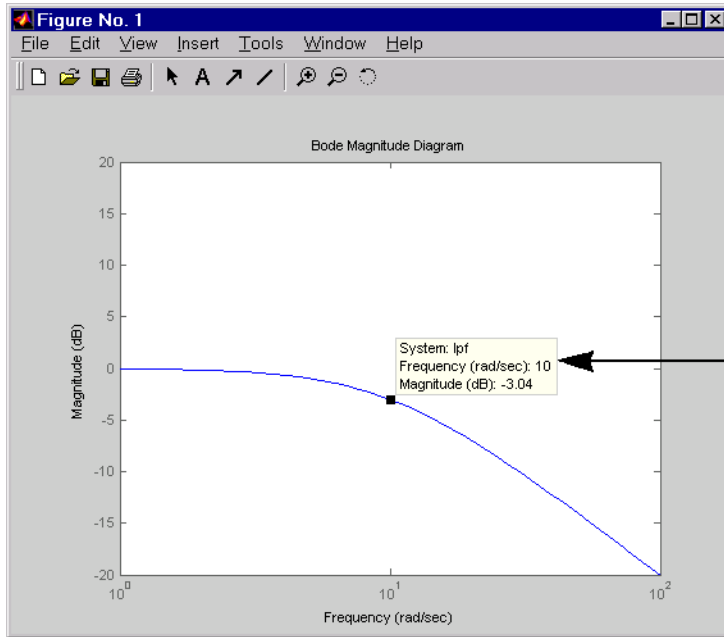
This data marker verifies that 10 rad/s is the -3 dB point.

Bode Magnitude of the Lowpass Filter

There is some measurement noise n, with noise intensity given by

$$E(n^2) = 0.01$$

Use the cost function

$$J(u) = \int_0^\infty (10y^2 + u^2)dt$$

to specify the tradeoff between regulation performance and cost of control. The following equations represent an open-loop state-space model:

$$\dot{x} = Ax + Bu + Bd \quad (state\,equations)$$
$$y = Cx + n \quad\quad\quad (measurements)$$

where (A,B,C) is a state-space realization of $100/(s^2 + s + 100)$.

The following commands design the optimal LQG regulator *F*(*s*) for this problem:

```
sys = ss(tf(100,[1 1 100])) % State-space plant model

% Design LQ-optimal gain K
K = lqry(sys,10,1)      % u = -Kx minimizes J(u)

% Separate control input u and disturbance input d
P = sys(:,[1 1]);
% input [u;d], output y

% Design Kalman state estimator Kest.
Kest = kalman(P,1,0.01)

% Form LQG regulator = LQ gain + Kalman filter.
F = lqgreg(Kest,K)
```

These commands returns a state-space model F of the LQG regulator *F*(*s*). The `lqry`, `kalman`, and `lqgreg` functions perform discrete-time LQG design when you apply them to discrete plants.

To validate the design, close the loop with `feedback`, create and add the lowpass filter in series with the closed-loop system, and compare the open- and closed-loop impulse responses by using the `impulse` function.

```
% Close loop
clsys = feedback(sys,F,+1)
% Note positive feedback.

% Create the lowpass filter and add it in series with clsys.
s = tf('s');
lpf= 10/(s+10) ;
clsys_fin = lpf*clsys;

% Open- vs. closed-loop impulse responses
impulse(sys,'r--',clsys_fin,'b-')
```

These commands produce the following figure, which compares the open- and closed-loop impulse responses for this example.
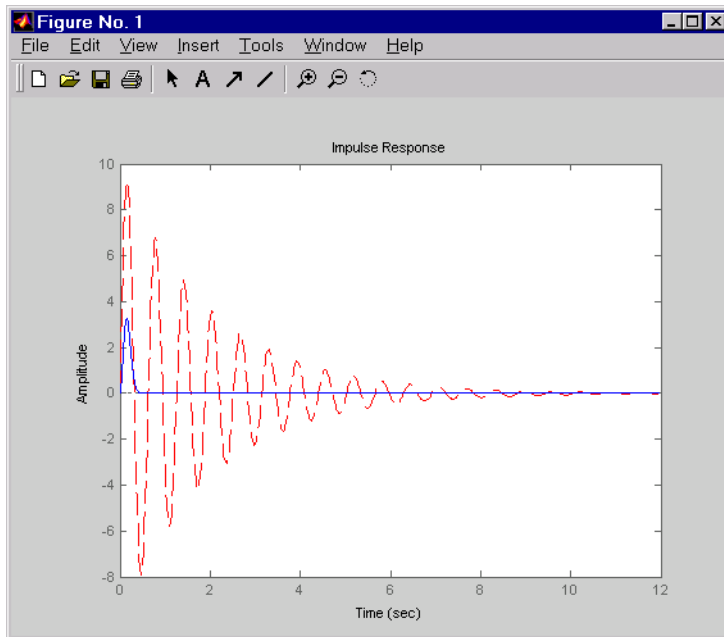
Comparison of Open- and Closed-Loop Impulse Response

See Also

`kalman` | `lqgreg`

Related Examples

- Kalman Filtering
- "Linear-Quadratic-Gaussian (LQG) Design" on page 4-85
- "Design an LQG Servo Controller" on page 4-100
- "Design an LQR Servo Controller in Simulink" on page 4-103

Design an LQG Servo Controller

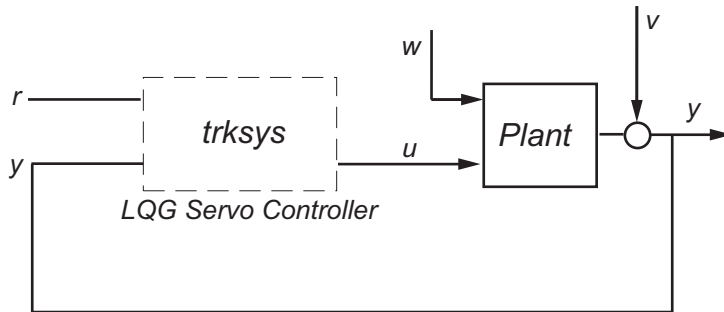This example shows you how to design a servo controller for the following system.

The plant has three states (x), two control inputs (u), two random inputs (w), one output (y), measurement noise for the output (v), and the following state and measurement equations:

$$\dot{x} = Ax + Bu + Gw$$
$$y = Cx + Du + Hw + v$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0.3 & 1 \\ 0 & 1 \\ -0.3 & 0.9 \end{bmatrix} \quad G = \begin{bmatrix} -0.7 & 1.12 \\ -1.17 & 1 \\ 0.14 & 1.5 \end{bmatrix}$$

$$C = \begin{bmatrix} 1.9 & 1.3 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 0.53 & -0.61 \end{bmatrix} \quad H = \begin{bmatrix} -1.2 & -0.89 \end{bmatrix}$$

The system has the following noise covariance data:

$$Q_n = E(ww^T) = \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix}$$
$$Rn = E(vv^T) = 0.7$$

Use the following cost function to define the tradeoff between tracker performance and control effort:

$$J(u) = \int_0^\infty \left(0.1x^T x + x_i^2 + u^T \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} u \right) dt$$

To design an LQG servo controller for this system:

1 Create the state space system by typing the following in the MATLAB Command Window:

```
A = [0 1 0;0 0 1;1 0 0];
B = [0.3 1;0 1;-0.3 0.9];
G = [-0.7 1.12; -1.17 1; .14 1.5];
C = [1.9 1.3 1];
D = [0.53 -0.61];
H = [-1.2 -0.89];
sys = ss(A,[B G],C,[D H]);
```

2 Construct the optimal state-feedback gain using the given cost function by typing the following commands:

```
nx = 3;     %Number of states
ny = 1;     %Number of outputs
Q = blkdiag(0.1*eye(nx),eye(ny));
R = [1 0;0 2];
K = lqi(ss(A,B,C,D),Q,R);
```

3 Construct the Kalman state estimator using the given noise covariance data by typing the following commands:

```
Qn = [4 2;2 1];
Rn = 0.7;
kest = kalman(sys,Qn,Rn);
```

4 Connect the Kalman state estimator and the optimal state-feedback gain to form the LQG servo controller by typing the following command:

```
trksys = lqgtrack(kest,K)
```

This command returns the following LQG servo controller:

```
>> trksys = lqgtrack(kest,K)

a =
            x1_e     x2_e     x3_e      xi1
    x1_e   -2.373   -1.062   -1.649    0.772
    x2_e   -3.443   -2.876   -1.335   0.6351
```

4-101

```
    x3_e  -1.963  -2.483  -2.043  0.4049
    xi1       0       0       0       0

b =
              r1      y1
    x1_e       0  0.2849
    x2_e       0  0.7727
    x3_e       0  0.7058
    xi1        1      -1

c =
           x1_e     x2_e     x3_e      xi1
    u1  -0.5388  -0.4173  -0.2481   0.5578
    u2   -1.492   -1.388   -1.131   0.5869

d =
        r1  y1
    u1   0   0
    u2   0   0

Input groups:
        Name        Channels
      Setpoint         1
    Measurement        2

Output groups:
       Name       Channels
      Controls      1,2

Continuous-time model.
```

See Also

kalman | lqgtrack | lqi

Related Examples

- Kalman Filtering
- "Linear-Quadratic-Gaussian (LQG) Design" on page 4-85
- "Design an LQG Regulator" on page 4-96
- "Design an LQR Servo Controller in Simulink" on page 4-103

Design an LQR Servo Controller in Simulink

The following figure shows a Simulink block diagram shows a tracking problem in aircraft autopilot design. To open this diagram, type `lqrpilot` at the MATLAB prompt.
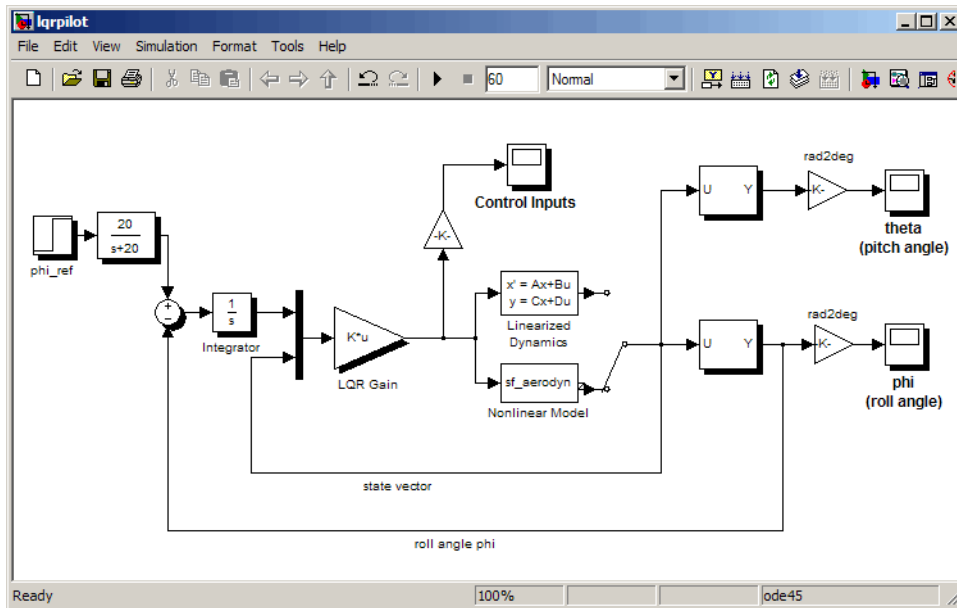
Key features of this diagram to note are the following:

- The Linearized Dynamics block contains the linearized airframe.
- `sf_aerodyn` is an S-Function block that contains the nonlinear equations for

  $(\theta, \phi) = (0, 15°)$.

-

  The error signal between ϕ and the ϕ_{ref} is passed through an integrator. This aids in driving the error to zero.
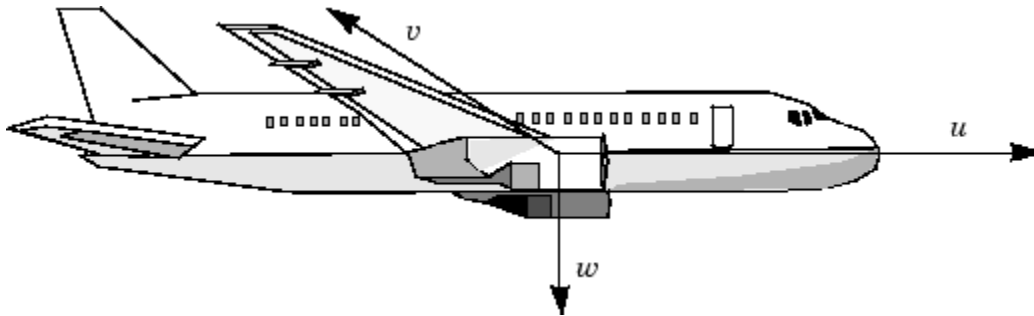
State-Space Equations for an Airframe

Beginning with the standard state-space equation

  $\dot{x} = Ax + Bu$

where

$$x = [u, v, w, p, q, r, \theta, \phi]^T$$

The variables u, v, and w are the three velocities with respect to the body frame, shown as follows.

Body Coordinate Frame for an Aircraft

The variables ϕ and θ are roll and pitch, and p, q, and r are the roll, pitch, and yaw rates, respectively.

The airframe dynamics are nonlinear. The following equation shows the nonlinear components added to the state space equation.

$$\dot{x} = Ax + Bu + \begin{bmatrix} -g\sin\theta \\ g\cos\theta\sin\phi \\ g\cos\theta\cos\phi \\ 0 \\ 0 \\ 0 \\ q\cos\phi - r\sin\phi \\ (q\sin\phi + r\cos\phi)\cdot\tan\theta \end{bmatrix}$$

Nonlinear Component of the State-Space Equation

To see the numerical values for A and B, type

```
load lqrpilot
A, B
```

at the MATLAB prompt.

Trimming

For LQG design purposes, the nonlinear dynamics are trimmed at $\phi = 15°$ and p, q, r, and θ set to zero. Since u, v, and w do not enter into the nonlinear term in the preceding

figure, this amounts to linearizing around $(\theta,\phi) = (0,15°)$ with all remaining states set to zero. The resulting state matrix of the linearized model is called A15.

Problem Definition

The goal to perform a steady coordinated turn, as shown in this figure.
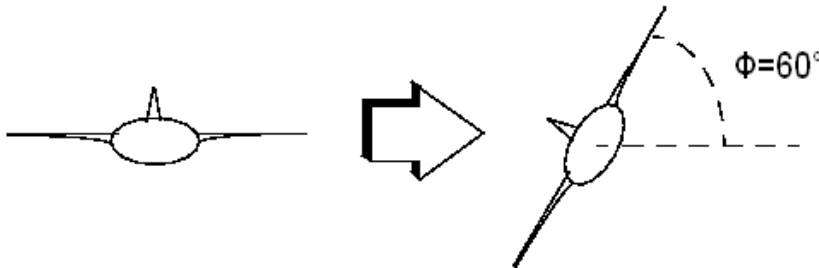
Aircraft Making a 60° Turn

To achieve this goal, you must design a controller that commands a steady turn by going through a 60° roll. In addition, assume that θ, the pitch angle, is required to stay as close to zero as possible.

Results

To calculate the LQG gain matrix, K, type

```
lqrdes
```

at the MATLAB prompt. Then, start the `lqrpilot` model with the nonlinear model, `sf_aerodyn`, selected.

This figure shows the response of ϕ to the 60° step command.
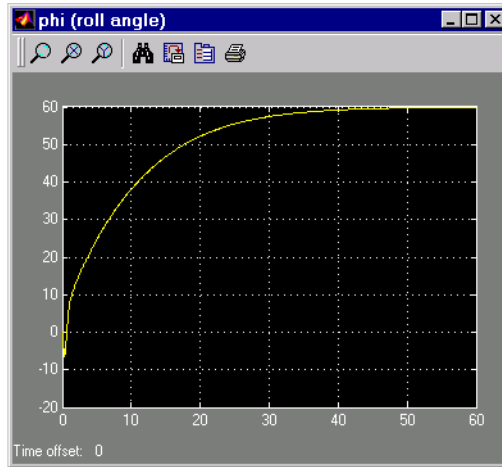
Tracking the Roll Step Command

As you can see, the system tracks the commanded 60° roll in about 60 seconds.

Another goal was to keep θ, the pitch angle, relatively small. This figure shows how well the LQG controller did.

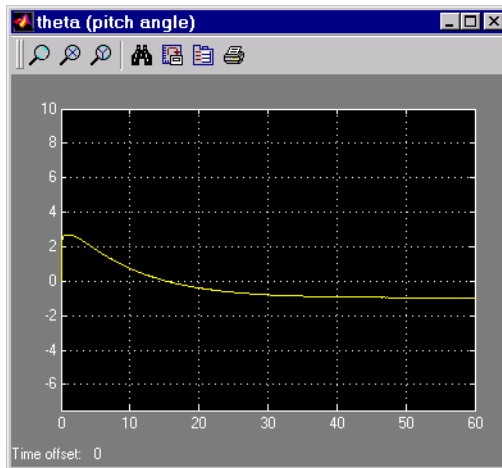**Minimizing the Displacement in the Pitch Angle, Theta**

Finally, this figure shows the control inputs.
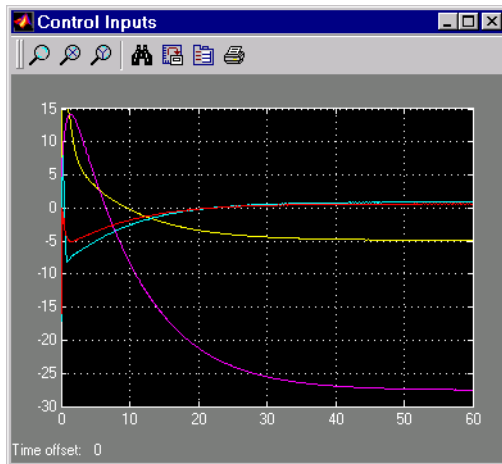
Control Inputs for the LQG Tracking Problem

Try adjusting the Q and R matrices in lqrdes.m and inspecting the control inputs and the system states, making sure to rerun lqrdes to update the LQG gain matrix K. Through trial and error, you may improve the response time of this design. Also, compare the

linear and nonlinear designs to see the effects of the nonlinearities on the system performance.

See Also

Related Examples
- Kalman Filtering
- "Linear-Quadratic-Gaussian (LQG) Design" on page 4-85
- "Design an LQG Regulator" on page 4-96
- "Design an LQG Servo Controller" on page 4-100

State Estimation Using Time-Varying Kalman Filter

This example shows how to estimate states of linear systems using time-varying Kalman filters in Simulink. You use the Kalman Filter block from the Control System Toolbox library to estimate the position and velocity of a ground vehicle based on noisy position measurements such as GPS sensor measurements. The plant model in Kalman filter has time-varying noise characteristics.

Introduction

You want to estimate the position and velocity of a ground vehicle in the north and east directions. The vehicle can move freely in the two-dimensional space without any constraints. You design a multi-purpose navigation and tracking system that can be used for any object and not just a vehicle.
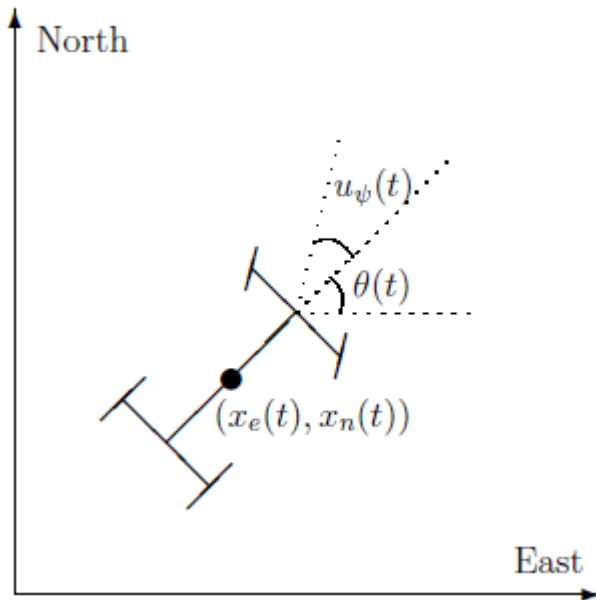
$x_e(t)$ and $x_n(t)$ are the vehicle's east and north positions from the origin, $\theta(t)$ is the vehicle orientation from east and $u_\psi(t)$ is the steering angle of the vehicle. t is the continuous-time variable.

The Simulink model consists of two main parts: Vehicle model and the Kalman filter. These are explained further in the following sections.

```
open_system('ctrlKalmanNavigationExample');
```
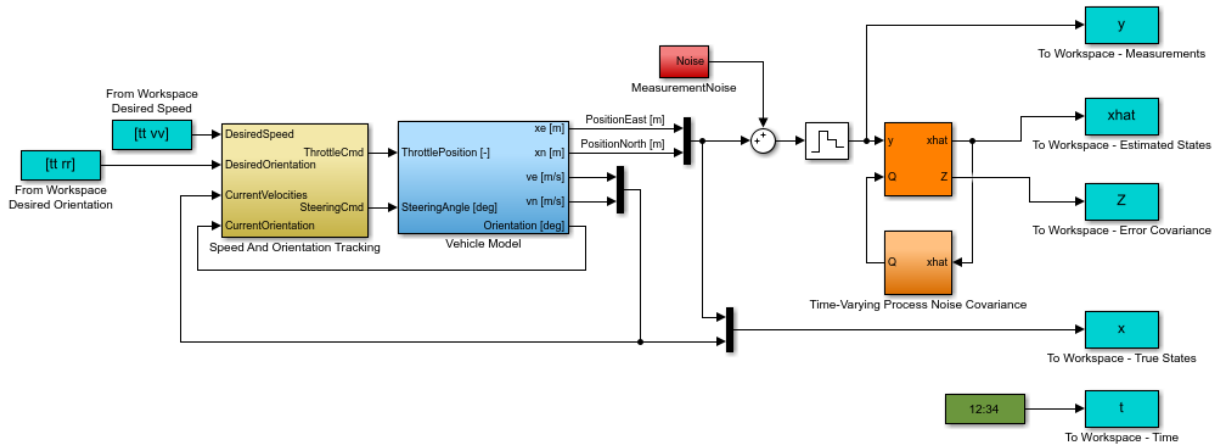
Vehicle Model

The tracked vehicle is represented with a simple point-mass model:

$$
\frac{d}{dt}
\begin{bmatrix}
x_e(t) \\
x_n(t) \\
s(t) \\
\theta(t)
\end{bmatrix}
=
\begin{bmatrix}
s(t)\cos(\theta(t)) \\
s(t)\sin(\theta(t)) \\
(P\,\frac{u_T(t)}{s(t)} - A\,C_d\,s(t)^2)/m \\
s(t)\tan(u_\psi(t))/L
\end{bmatrix}
$$

where the vehicle states are:

$x_e(t)$ East position $[m]$
$x_n(t)$ North position $[m]$
$s(t)$ Speed $[m/s]$
$\theta(t)$ Orientation from east $[deg]$

the vehicle parameters are:

$P = 100000$ Peak engine power $[W]$
$A = 1$ Frontal area $[m^2]$
$C_d = 0.3$ Drag coefficient $[Unitless]$
$m = 1250$ Vehicle mass $[kg]$
$L = 2.5$ Wheelbase length $[m]$

and the control inputs are:

$u_T(t)$ Throttle position in the range of -1 and 1 $[Unitless]$
$u_\psi(t)$ Steering angle $[deg]$

The longitudinal dynamics of the model ignore tire rolling resistance. The lateral dynamics of the model assume that the desired steering angle can be achieved instantaneously and ignore the yaw moment of inertia.

The car model is implemented in the `ctrlKalmanNavigationExample/Vehicle Model` subsystem. The Simulink model contains two PI controllers for tracking the desired orientation and speed for the car in the `ctrlKalmanNavigationExample/ Speed And Orientation Tracking` subsystem. This allows you to specify various operating conditions for the car and test the Kalman filter performance.

Kalman Filter Design

Kalman filter is an algorithm to estimate unknown variables of interest based on a linear model. This linear model describes the evolution of the estimated variables over time in response to model initial conditions as well as known and unknown model inputs. In this example, you estimate the following parameters/variables:

$$\hat{x}[n] = \begin{bmatrix} \hat{x}_e[n] \\ \hat{x}_n[n] \\ \hat{\dot{x}}_e[n] \\ \hat{\dot{x}}_n[n] \end{bmatrix}$$

where

$\hat{x}_e[n]$ East position estimate $[m]$
$\hat{x}_n[n]$ North position estimate $[m]$
$\hat{\dot{x}}_e[n]$ East velocity estimate $[m/s]$
$\hat{\dot{x}}_n[n]$ North velocity estimate $[m/s]$

The $\dot{x}$ terms denote velocities and not the derivative operator. n is the discrete-time index. The model used in the Kalman filter is of the form:

$$\hat{x}[n+1] = A\hat{x}[n] + Gw[n]$$
$$y[n] = C\hat{x}[n] + v[n]$$

where $\hat{x}$ is the state vector, y is the measurements, w is the process noise, and v is the measurement noise. Kalman filter assumes that w and v are zero-mean, independent random variables with known variances $E[ww^T] = Q$, $E[vv^T] = R$, and $E[wv^T] = N$. Here, the A, G, and C matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

where $T_s = 1\,[s]$

The third row of A and G model the east velocity as a random walk:
$\hat{\dot{x}}_e[n+1] = \hat{\dot{x}}_e[n] + w_1[n]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{\dot{x}}_e$. The first row of the A and G represent a discrete approximation to this kinematic relationship:
$(\hat{x}_e[n+1] - \hat{x}_e[n])/Ts = (\hat{\dot{x}}_e[n+1] + \hat{\dot{x}}_e[n])/2$. The second and fourth rows of the A and G represent the same relationship between the north velocity and position.

The C matrix represents that only position measurements are available. A position sensor, such as GPS, provides these measurements at the sample rate of 1Hz. The variance of the measurement noise v, the R matrix, is specified as $R = 50$. Since R is specified as a scalar, the Kalman filter block assumes that the matrix R is diagonal, its diagonals are 50

and is of compatible dimensions with y. If the measurement noise is Gaussian, R=50 corresponds to 68% of the position measurements being within $\pm\sqrt{50}\ m$ or the actual position in the east and north directions. However, this assumption is not necessary for the Kalman filter.

The elements of w capture how much the vehicle velocity can change over one sample time Ts. The variance of the process noise w, the Q matrix, is chosen to be time-varying. It captures the intuition that typical values of $w[n]$ are smaller when velocity is large. For instance, going from 0 to 10m/s is easier than going from 10 to 20m/s. Concretely, you use the estimated north and east velocities and a saturation function to construct Q[n]:

$$f_{sat}(z) = min(max(z, 25), 625)$$

$$Q[n] = \begin{bmatrix} 1 + \dfrac{250}{f_{sat}(\hat{x}_e^2)} & 0 \\ 0 & 1 + \dfrac{250}{f_{sat}(\hat{x}_n^2)} \end{bmatrix}$$

The diagonals of Q model the variance of w inversely proportional to the square of the estimated velocities. The saturation function prevents Q from becoming too large or small. The coefficient 250 is obtained from a least squares fit to 0-5, 5-10, 10-15, 15-20, 20-25m/s acceleration time data for a generic vehicle. Note that the diagonal Q choice represents a naive assumption that the velocity changes in the north and east direction are uncorrelated.

Kalman Filter Block Inputs and Setup

The 'Kalman Filter' block is in the `Control System Toolbox` library in Simulink. It is also in `System Identification Toolbox/Estimators` library. Configure the block parameters for discrete-time state estimation. Specify the following **Filter Settings** parameters:

- **Time domain:** Discrete-time. Choose this option to estimate discrete-time states.

- Select the **Use current measurement y[n] to improve xhat[n]** check box. This implements the "current estimator" variant of the discrete-time Kalman filter. This option improves the estimation accuracy and is more useful for slow sample times. However, it increases the computational cost. In addition, this Kalman filter variant has direct feedthrough, which leads to an algebraic loop if the Kalman filter is used in

a feedback loop that does not contain any delays (the feedback loop itself also has direct feedthrough). The algebraic loop can further impact the simulation speed.

Click the **Options** tab to set the block inport and outport options:

- Unselect the **Add input port u** check box. There are no known inputs in the plant model.

- Select the **Output state estimation error covariance Z** check box. The Z matrix provides information about the filter's confidence in the state estimates.

Click **Model Parameters** to specify the plant model and noise characteristics:

- **Model source:** Individual A, B, C, D matrices.

- **A**: A. The A matrix is defined earlier in this example.

- **C**: C. The C matrix is defined earlier in this example.

- **Initial Estimate Source**: Dialog

- **Initial states x[0]**: 0. This represents an initial guess of 0 for the position and velocity estimates at t=0s.

- **State estimation error covariance P[0]**: 10. Assume that the error between your initial guess x[0] and its actual value is a random variable with a standard deviation $\sqrt{10}$.

- Select the **Use G and H matrices (default G=I and H=0)** check box to specify a non-default G matrix.

- **G**: G. The G matrix is defined earlier in this example.

- **H**: 0. The process noise does not impact the measurements y entering the Kalman filter block.

- Unselect the **Time-invariant Q** check box. The Q matrix is time-varying and is supplied through the block inport Q. The block uses a time-varying Kalman filter due to this setting. You can select this option to use a time-invariant Kalman filter. A time-invariant Kalman filter performs slightly worse for this problem, but is easier to design and has a lower computational cost.

- **R**: R. This is the covariance of the measurement noise $v[n]$. The R matrix is defined earlier in this example.

- **N**: 0. Assume that there is no correlation between process and measurement noises.

- **Sample time (-1 for inherited)**: Ts, which is defined earlier in this example.

Kalman Filter

Estimate the states of a discrete-time or continuous-time linear
system. Time-varying systems are supported.

Filter Settings

Time domain: Discrete-Time ▼

☑ Use the current measurement y[n] to improve xhat[n]

| Model Parameters | Options |

System Model

Model source: Individual A, B, C, D matrices ▼

A: [1 0 Ts 0; 0 1 0 Ts; 0 0 1 0; 0 0 0 1]

C: [1 0 0 0; 0 1 0 0]

Initial Estimates

Source: Dialog ▼

Initial states x[0]: 0

State estimation error covariance P[0] 10

Noise Characteristics

☑ Use G and H matrices (default G=I and H=0)

G: [Ts/2 0; 0 Ts/2; 1 0; 0 1]   ☑ Time-invariant G

H: 0   ☑ Time-invariant H

Q: 0.05   ☐ Time-invariant Q

R: 50   ☑ Time-invariant R

N: 0   ☑ Time-invariant N

Sample time (-1 for inherited): Ts

4-117

Results

Test the performance of the Kalman filter by simulating a scenario where the vehicle makes the following maneuvers:

- At t = 0 the vehicle is at $x_e(0) = 0$, $x_n(0) = 0$ and is stationary.

- Heading east, it accelerates to 25m/s. It decelerates to 5m/s at t=50s.

- At t = 100s, it turns toward north and accelerates to 20m/s.

- At t = 200s, it makes another turn toward west. It accelerates to 25m/s.

- At t = 260s, it decelerates to 15m/s and makes a constant speed 180 degree turn.

Simulate the Simulink model. Plot the actual, measured and Kalman filter estimates of vehicle position.

```
sim('ctrlKalmanNavigationExample');

figure;
% Plot results and connect data points with a solid line.
plot(x(:,1),x(:,2),'bx',...
    y(:,1),y(:,2),'gd',...
    xhat(:,1),xhat(:,2),'ro',...
    'LineStyle','-');
title('Position');
xlabel('East [m]');
ylabel('North [m]');
legend('Actual','Measured','Kalman filter estimate','Location','Best');
axis tight;
```
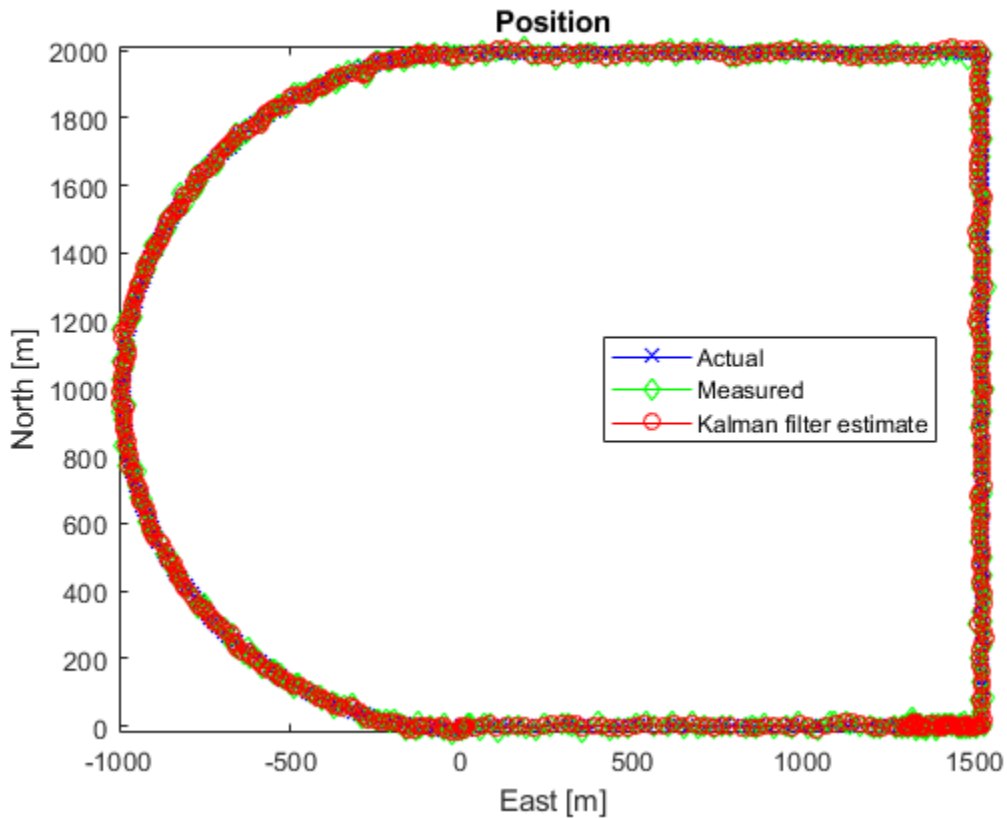
The error between the measured and actual position as well as the error between the Kalman filter estimate and actual position is:

```
% East position measurement error [m]
n_xe = y(:,1)-x(:,1);
% North position measurement error [m]
n_xn = y(:,2)-x(:,2);
% Kalman filter east position error [m]
e_xe = xhat(:,1)-x(:,1);
% Kalman filter north position error [m]
e_xn = xhat(:,2)-x(:,2);

figure;
% East Position Errors
```

4-119

```
subplot(2,1,1);
plot(t,n_xe,'g',t,e_xe,'r');
ylabel('Position Error - East [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xe,1)/numel(n_xe)),sprintf('Kalman f.: %.3f',norm(e_
axis tight;
% North Position Errors
subplot(2,1,2);
plot(t,y(:,2)-x(:,2),'g',t,xhat(:,2)-x(:,2),'r');
ylabel('Position Error - North [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xn,1)/numel(n_xn)),sprintf('Kalman f: %.3f',norm(e_x
axis tight;
```
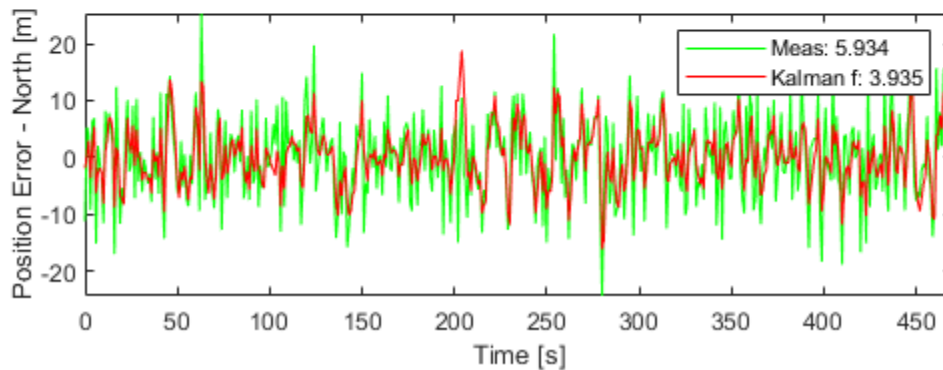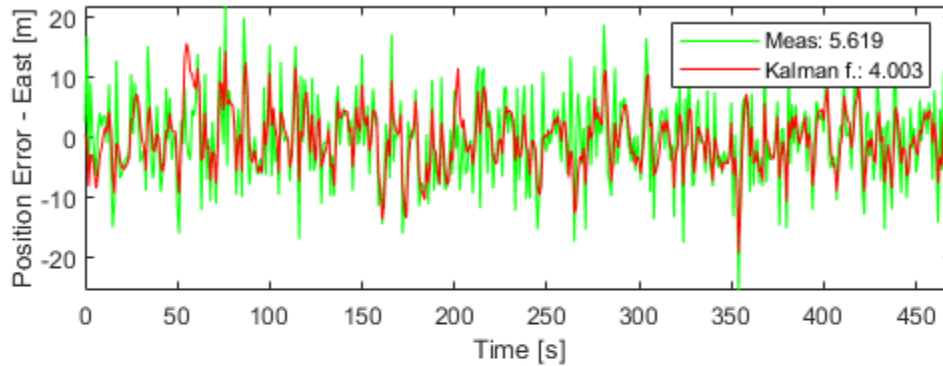
The plot legends show the position measurement and estimation error ($\|x_e - \hat{x}_e\|_1$ and $\|x_n - \hat{x}_n\|_1$) normalized by the number of data points. The Kalman filter estimates have about 25% percent less error than the raw measurements.

The actual velocity in the east direction and its Kalman filter estimate is shown below in the top plot. The bottom plot shows the estimation error.

```
e_ve = xhat(:,3)-x(:,3); % [m/s] Kalman filter east velocity error
e_vn = xhat(:,4)-x(:,4); % [m/s] Kalman filter north velocity error
figure;
% Velocity in east direction and its estimate
subplot(2,1,1);
plot(t,x(:,3),'b',t,xhat(:,3),'r');
ylabel('Velocity - East [m]');
xlabel('Time [s]');
legend('Actual','Kalman filter','Location','Best');
axis tight;
subplot(2,1,2);
% Estimation error
plot(t,e_ve,'r');
ylabel('Velocity Error - East [m]');
xlabel('Time [s]');
legend(sprintf('Kalman filter: %.3f',norm(e_ve,1)/numel(e_ve)));
axis tight;
```
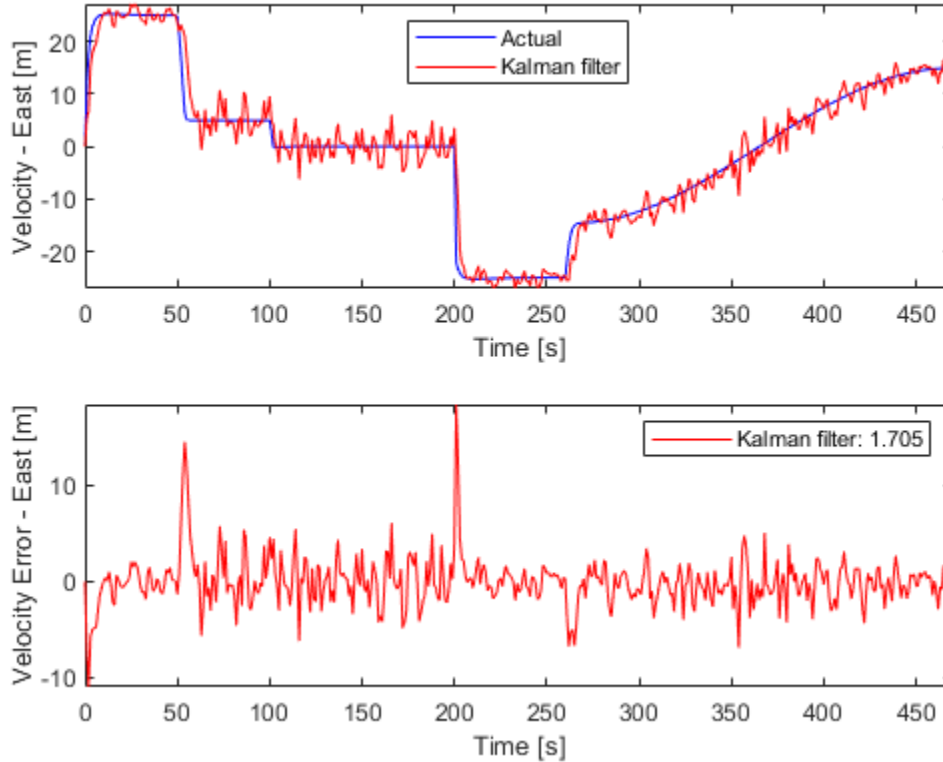
The legend on the error plot shows the east velocity estimation error $||\dot{x}_e - \hat{\dot{x}}_e||_1$ normalized by the number of data points.

The Kalman filter velocity estimates track the actual velocity trends correctly. The noise levels decrease when the vehicle is traveling at high velocities. This is in line with the design of the Q matrix. The large two spikes are at t=50s and t=200s. These are the times when the car goes through sudden deceleration and a sharp turn, respectively. The velocity changes at those instants are much larger than the predictions from the Kalman filter, which is based on its Q matrix input. After a few time-steps, the filter estimates catch up with the actual velocity.

Summary

You estimated the position and velocity of a vehicle using the Kalman filter block in Simulink. The process noise dynamics of the model were time-varying. You validated the filter performance by simulating various vehicle maneuvers and randomly generated measurement noise. The Kalman filter improved the position measurements and provided velocity estimates for the vehicle.

```
bdclose('ctrlKalmanNavigationExample');
```

See Also

Related Examples

- Kalman Filtering