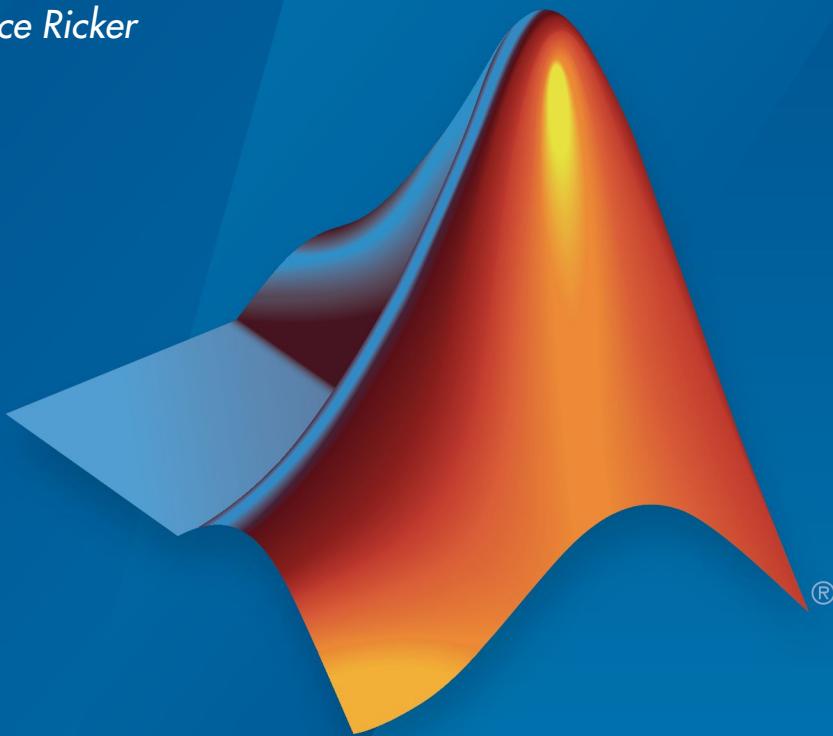


Model Predictive Control Toolbox™

Getting Started Guide

*Alberto Bemporad
Manfred Morari
N. Lawrence Ricker*



MATLAB®

R2016a

 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

Model Predictive Control Toolbox™ Getting Started Guide

© COPYRIGHT 2005–2016 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to 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

October 2004	First printing	New for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.2.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)
March 2010	Online only	Revised for Version 3.2 (Release 2010a)
September 2010	Online only	Revised for Version 3.2.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.1.3 (Release 2013b)
March 2014	Online only	Revised for Version 4.2 (Release R2014a)
October 2014	Online only	Revised for Version 5.0 (Release 2014b)
March 2015	Online only	Revised for Version 5.0.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.1 (Release 2015b)
March 2016	Online only	Revised for Version 5.2 (Release 2016a)

Introduction

1

Model Predictive Control Toolbox Product Description	1-2
Key Features	1-2
Acknowledgments	1-3
Bibliography	1-4

Building Models

2

MPC Modeling	2-2
Plant Model	2-2
Input Disturbance Model	2-4
Output Disturbance Model	2-5
Measurement Noise Model	2-6
Signal Types	2-8
Inputs	2-8
Outputs	2-8
Construct Linear Time Invariant (LTI) Models	2-9
Transfer Function Models	2-9
Zero/Pole/Gain Models	2-9
State-Space Models	2-10
LTI Object Properties	2-12
LTI Model Characteristics	2-15
Specify Multi-Input Multi-Output (MIMO) Plants	2-17

CSTR Model	2-20
Linearize Simulink Models	2-22
Linearization Using MATLAB Code	2-22
Linearization Using Linear Analysis Tool in Simulink Control Design	2-25
Linearize Simulink Models Using MPC Designer	2-32
Define MPC Structure By Linearization	2-32
Linearize Model	2-37
Specifying Operating Points	2-39
Identify Plant from Data	2-52
Design Controller for Identified Plant	2-54
Design Controller Using Identified Model with Noise Channel	2-56
Working with Impulse-Response Models	2-58
Bibliography	2-60

Designing Controllers Using MPC Designer

3

Design Controller Using MPC Designer	3-2
Test Controller Robustness	3-23
Design MPC Controller for Plant with Delays	3-35
Design MPC Controller for Nonsquare Plant	3-43
More Outputs Than Manipulated Variables	3-43
More Manipulated Variables Than Outputs	3-45

Designing Controllers Using the Command Line

4

Design MPC Controller at the Command Line	4-2
Simulate Controller with Nonlinear Plant	4-16
Nonlinear CSTR Application	4-16
Example Code for Successive Linearization	4-17
CSTR Results and Discussion	4-19
Control Based On Multiple Plant Models	4-23
A Two-Model Plant	4-23
Designing the Two Controllers	4-25
Simulating Controller Performance	4-26
Compute Steady-State Gain	4-32
Extract Controller	4-34
Signal Previewing	4-37
Run-Time Constraint Updating	4-39
Run-Time Weight Tuning	4-40

Designing and Testing Controllers in Simulink

5

Design MPC Controller in Simulink	5-2
Test an Existing Controller	5-23
Schedule Controllers at Multiple Operating Points	5-27
A Two-Model Plant	5-27
Animation of the Multi-Model Example	5-28
Designing the Two Controllers	5-29
Simulating Controller Performance	5-30

Introduction

- “Model Predictive Control Toolbox Product Description” on page 1-2
- “Acknowledgments” on page 1-3
- “Bibliography” on page 1-4

Model Predictive Control Toolbox Product Description

Design and simulate model predictive controllers

Model Predictive Control Toolbox™ provides functions, an app, and Simulink® blocks for systematically analyzing, designing, and simulating model predictive controllers. You can specify plant and disturbance models, horizons, constraints, and weights. The toolbox enables you to diagnose issues that could lead to run-time failures and provides advice on tuning weights to improve performance and robustness. By running different scenarios in linear and nonlinear simulations, you can evaluate controller performance.

You can adjust controller performance as it runs by tuning weights and varying constraints. You can implement adaptive model predictive controllers by updating the plant model at run time. For applications with fast sample times, you can develop explicit model predictive controllers. For rapid prototyping and embedded system design, the toolbox supports C-code and IEC 61131-3 Structured Text generation.

Key Features

- Design and simulation of model predictive controllers in MATLAB® and Simulink
- Customization of constraints and weights with advisory tools for improved performance and robustness
- Adaptive MPC control through run-time changes to internal plant model
- Explicit MPC control for applications with fast sample times using pre-computed solutions
- Control of plants over a wide range of operating conditions by switching between multiple model predictive controllers
- Specialized model predictive control quadratic programming (QP) solver optimized for speed, efficiency, and robustness
- Support for C-code generation with Simulink Coder™ and IEC 61131-3 Structured Text generation with Simulink PLC Coder™

Acknowledgments

MathWorks would like to acknowledge the following contributors to Model Predictive Control Toolbox.

Alberto Bemporad

Professor of Control Systems, IMT Institute for Advanced Studies Lucca, Italy.
Research interests include model predictive control, hybrid systems, optimization algorithms, and applications to automotive, aerospace, and energy systems. Fellow of the IEEE®. Author of the Model Predictive Control Simulink library and commands.

Manfred Morari

Professor at the Automatic Control Laboratory and former Head of Department of Information Technology and Electrical Engineering, ETH Zurich, Switzerland.
Research interests include model predictive control, hybrid systems, and robust control. Fellow of the IEEE, AIChE, and IFAC. Co-author of the first version of the toolbox.

N. Lawrence Ricker

Professor of Chemical Engineering, University of Washington, Seattle, USA.
Research interests include model predictive control and process optimization. Author of the quadratic programming solver and graphical user interface.

Bibliography

- [1] Allgower, F., and A. Zheng, *Nonlinear Model Predictive Control*, Springer-Verlag, 2000.
- [2] Camacho, E. F., and C. Bordons, *Model Predictive Control*, Springer-Verlag, 1999.
- [3] Kouvaritakis, B., and M. Cannon, *Non-Linear Predictive Control: Theory & Practice*, IEE Publishing, 2001.
- [4] Maciejowski, J. M., *Predictive Control with Constraints*, Pearson Education POD, 2002.
- [5] Prett, D., and C. Garcia, *Fundamental Process Control*, Butterworths, 1988.
- [6] Rossiter, J. A., *Model-Based Predictive Control: A Practical Approach*, CRC Press, 2003.

Building Models

- “MPC Modeling” on page 2-2
- “Signal Types” on page 2-8
- “Construct Linear Time Invariant (LTI) Models” on page 2-9
- “Specify Multi-Input Multi-Output (MIMO) Plants” on page 2-17
- “CSTR Model” on page 2-20
- “Linearize Simulink Models” on page 2-22
- “Linearize Simulink Models Using MPC Designer” on page 2-32
- “Identify Plant from Data” on page 2-52
- “Design Controller for Identified Plant” on page 2-54
- “Design Controller Using Identified Model with Noise Channel” on page 2-56
- “Working with Impulse-Response Models” on page 2-58
- “Bibliography” on page 2-60

MPC Modeling

In this section...

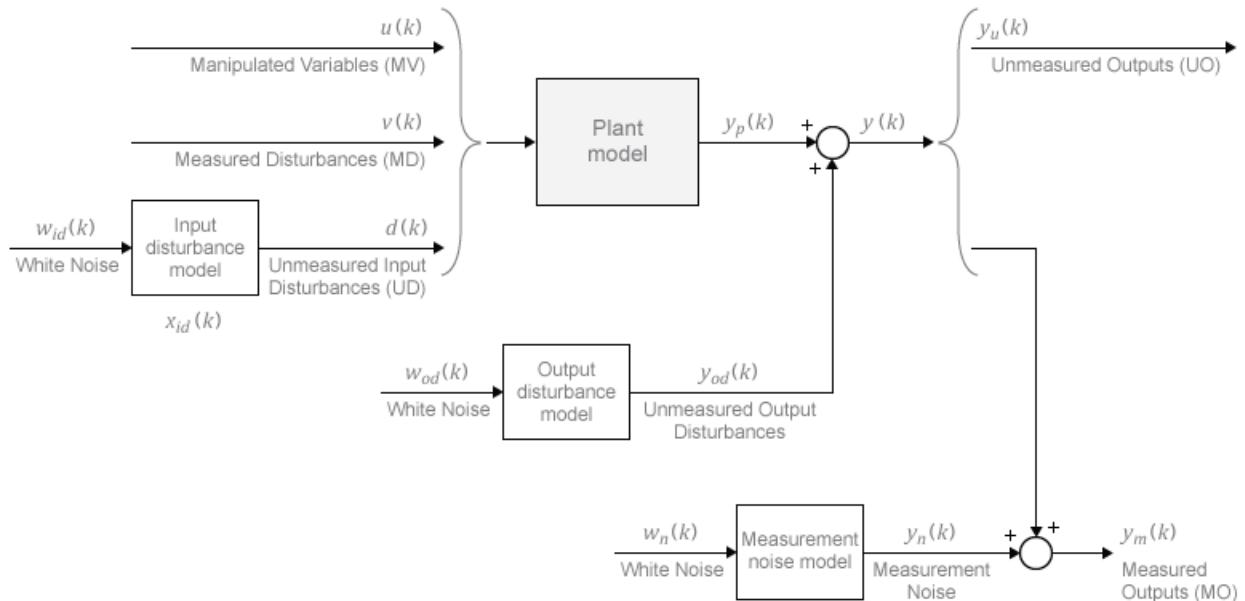
[“Plant Model” on page 2-2](#)

[“Input Disturbance Model” on page 2-4](#)

[“Output Disturbance Model” on page 2-5](#)

[“Measurement Noise Model” on page 2-6](#)

The model structure used in the MPC controller appears in the following illustration. This section explains how the models connect for prediction and state estimation.



Plant Model

You can specify the plant model in one of the following linear-time-invariant (LTI) formats:

- Numeric LTI models: Transfer function (`tf`), state space (`ss`), zero-pole-gain (`zpk`)

- Identified models (requires System Identification Toolbox™): `idss`, `idtf`, `idproc`, and `idpoly`

The MPC controller performs all estimation and optimization calculations using a discrete-time, delay-free, state-space system with dimensionless input and output variables. Therefore, when you specify a plant model in the MPC controller, the software performs the following, if needed:

- Conversion to state space. The `ss` command converts the supplied model to an LTI state-space model.
- Discretization or resampling. If the model sample time differs from the MPC controller sample time (defined in the `Ts` property), one of the following occurs:
 - If the model is continuous time, the `c2d` command converts it to a discrete-time LTI object using the controller sample time.
 - If the model is discrete time, the `d2d` command resamples it to generate a discrete-time LTI object using the controller sample time.
- Delay removal. If the discrete-time model includes any input, output, or internal delays, the `absorbDelay` command replaces them with the appropriate number of poles at $z = 0$, increasing the total number of discrete states. The `InputDelay`, `OutputDelay`, and `InternalDelay` properties of the resulting state space model are all zero.
- Conversion to dimensionless input and output variables. The MPC controller enables you to specify a scale factor for each plant input and output variable. If you do not specify scale factors, they default to 1. The software converts the plant input and output variables to dimensionless form as follows:

$$\begin{aligned}x_p(k+1) &= A_p x_p(k) + B S_i u_p(k) \\y_p(k) &= S_o^{-1} C x_p(k) + S_o^{-1} D S_i u_p(k).\end{aligned}$$

where A_p , B , C , and D are the constant state-space matrices determined in step 3, and:

- S_i is a diagonal matrix of input scale factors in engineering units.
- S_o is a diagonal matrix of output scale factors in engineering units.
- x_p is the state vector from step 3 in engineering units. No scaling is performed on state variables.

- u_p is a vector of dimensionless plant input variables.
- y_p is a vector of dimensionless plant output variables.

The resulting plant model has the following equivalent form:

$$\begin{aligned}x_p(k+1) &= A_p x_p(k) + B_{pu} u(k) + B_{pv} v(k) + B_{pd} d(k) \\y_p(k) &= C_p x_p(k) + D_{pu} u(k) + D_{pv} v(k) + D_{pd} d(k).\end{aligned}$$

Here, $C_p = S_o^{-1}C$, B_{pu} , B_{pv} , and B_{pd} are the corresponding columns of BS_i . Also,

D_{pu} , D_{pv} , and D_{pd} are the corresponding columns of $S_o^{-1}DS_i$. Finally, $u(k)$, $v(k)$, and $d(k)$ are the dimensionless manipulated variables, measured disturbances, and unmeasured input disturbances, respectively.

The MPC controller enforces the restriction of $D_{pu} = 0$, which means that the controller does not allow direct feedthrough from any manipulated variable to any plant output.

Input Disturbance Model

If your plant model includes unmeasured input disturbances, $d(k)$, the input disturbance model specifies the signal type and characteristics of $d(k)$. See “Controller State Estimation” for more information about the model.

The `getindist` command provides access to the model in use.

The input disturbance model is a key factor that influences the following controller performance attributes:

- Dynamic response to apparent disturbances. That is, the character of the controller response when the measured plant output deviates from its predicted trajectory, due to an unknown disturbance or modeling error.
- Asymptotic rejection of sustained disturbances. If the disturbance model predicts a sustained disturbance, controller adjustments continue until the plant output returns to its desired trajectory, emulating a classical integral feedback controller.

You can provide the input disturbance model as an LTI state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) object using `setindist`. The MPC controller

converts the input disturbance model to a discrete-time, delay-free, LTI state-space system using the same steps used to convert the plant model. The result is:

$$\begin{aligned}x_{id}(k+1) &= A_{id}x_{id}(k) + B_{id}w_{id}(k) \\d(k) &= C_{id}x_{id}(k) + D_{id}w_{id}(k).\end{aligned}$$

where A_{id} , B_{id} , C_{id} , and D_{id} are constant state-space matrices, and:

- $x_{id}(k)$ is a vector of $n_{xid} \geq 0$ input disturbance model states.
- $d_k(k)$ is a vector of n_d dimensionless unmeasured input disturbances.
- $w_{id}(k)$ is a vector of $n_{id} \geq 1$ dimensionless white noise inputs, assumed to have zero mean and unity variance.

If you do not provide an input disturbance model, then the controller uses a default model, which has integrators with dimensionless unity gain added to its outputs. An integrator is added for each unmeasured input disturbance, unless doing so would cause a violation of state observability. In this case, a static system with dimensionless unity gain is used instead.

Output Disturbance Model

The output disturbance model is a special case of the more general input disturbance model. Its output, $y_{od}(k)$, is directly added to the plant output rather than affecting the plant states. The output disturbance model specifies the signal type and characteristics of $y_{od}(k)$, and it is often used in practice. See “Controller State Estimation” for more details about the model.

The `getoutdist` command provides access to the output disturbance model in use.

You can specify a custom output disturbance model as an LTI state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) object using `setoutdist`. Using the same steps as for the plant model, the MPC controller converts the specified output disturbance model to a discrete-time, delay-free, LTI state-space system. The result is:

$$\begin{aligned}x_{od}(k+1) &= A_{od}x_{od}(k) + B_{od}w_{od}(k) \\y_{od}(k) &= C_{od}x_{od}(k) + D_{od}w_{od}(k).\end{aligned}$$

where A_{od} , B_{od} , C_{od} , and D_{od} are constant state-space matrices, and:

- $x_{od}(k)$ is a vector of $n_{xod} \geq 1$ output disturbance model states.
- $y_{od}(k)$ is a vector of n_y dimensionless output disturbances to be added to the dimensionless plant outputs.
- $w_{od}(k)$ is a vector of n_{od} dimensionless white noise inputs, assumed to have zero mean and unit variance.

If you do not specify an output disturbance model, then the controller uses a default model, which has integrators with dimensionless unity gain added to some or all of its outputs. These integrators are added according to the following rules:

- No disturbances are estimated, that is no integrators are added, for unmeasured plant outputs.
- An integrator is added for each measured output in order of decreasing output weight.
 - For time-varying weights, the sum of the absolute values over time is considered for each output channel.
 - For equal output weights, the order within the output vector is followed.
- For each measured output, an integrator is not added if doing so would cause a violation of state observability. Instead, a gain with a value of zero is used instead.

If there is an input disturbance model, then the controller adds any default integrators to that model before constructing the default output disturbance model.

Measurement Noise Model

One controller design objective is to distinguish disturbances, which require a response, from measurement noise, which should be ignored. The measurement noise model specifies the expected noise type and characteristics. See “Controller State Estimation” for more details about the model.

Using the same steps as for the plant model, the MPC controller converts the measurement noise model to a discrete-time, delay-free, LTI state-space system. The result is:

$$\begin{aligned} x_n(k+1) &= A_n x_n(k) + B_n w_n(k) \\ y_n(k) &= C_n x_n(k) + D_n w_n(k). \end{aligned}$$

Here, A_n , B_n , C_n , and D_n are constant state space matrices, and:

- $x_n(k)$ is a vector of $n_{xn} \geq 0$ noise model states.
- $y_n(k)$ is a vector of n_{ym} dimensionless noise signals to be added to the dimensionless measured plant outputs.
- $w_n(k)$ is a vector of $n_n \geq 1$ dimensionless white noise inputs, assumed to have zero mean and unit variance.

If you do not supply a noise model, the default is a unity static gain: $n_{xn} = 0$, D_n is an n_{ym} -by- n_{ym} identity matrix, and A_n , B_n , and C_n are empty.

For an `mpc` controller object, `MPCobj`, the property `MPCobj.Model.Noise` provides access to the measurement noise model.

Note: If the minimum eigenvalue of $D_n D_n^T$ is less than 1×10^{-8} , the MPC controller adds 1×10^{-4} to each diagonal element of D_n . This adjustment makes a successful default Kalman gain calculation more likely.

More About

- “Controller State Estimation”

Signal Types

Inputs

The *plant inputs* are the independent variables affecting the plant. As shown in “MPC Modeling” on page 2-2, there are three types:

Measured disturbances

The controller can't adjust them, but uses them for feedforward compensation.

Manipulated variables

The controller adjusts these in order to achieve its goals.

Unmeasured disturbances

These are independent inputs of which the controller has no direct knowledge, and for which it must compensate.

Outputs

The *plant outputs* are the dependent variables (outcomes) you wish to control or monitor. As shown in “MPC Modeling” on page 2-2, there are two types:

Measured outputs

The controller uses these to estimate unmeasured quantities and as feedback on the success of its adjustments.

Unmeasured outputs

The controller estimates these based on available measurements and the plant model. The controller can also hold unmeasured outputs at setpoints or within constraint boundaries.

You must specify the input and output types when designing the controller. See “Input and Output Types” on page 2-13 for more details.

More About

- “MPC Modeling” on page 2-2

Construct Linear Time Invariant (LTI) Models

Model Predictive Control Toolbox software supports the same LTI model formats as does Control System Toolbox™ software. You can use whichever is most convenient for your application. It's also easy to convert from one format to another. For more details, see the Control System Toolbox documentation.

In this section...

- “Transfer Function Models” on page 2-9
- “Zero/Pole/Gain Models” on page 2-9
- “State-Space Models” on page 2-10
- “LTI Object Properties” on page 2-12
- “LTI Model Characteristics” on page 2-15

Transfer Function Models

A transfer function (TF) relates a particular input/output pair. For example, if $u(t)$ is a plant input and $y(t)$ is an output, the transfer function relating them might be:

$$\frac{Y(s)}{U(s)} = G(s) = \frac{s+2}{s^2+s+10} e^{-1.5s}$$

This TF consists of a *numerator* polynomial, $s+2$, a *denominator* polynomial, s^2+s+10 , and a delay, which is 1.5 time units here. You can define G using Control System Toolbox `tf` function:

```
Gtf1 = tf([1 2], [1 1 10], OutputDelay ,1.5)
```

Control System Toolbox software builds and displays it as follows:

```
Transfer function:  
s + 2  
exp(-1.5*s) * -----  
s^2 + s + 10
```

Zero/Pole/Gain Models

Like the TF, the zero/pole/gain (ZPK) format relates an input/output pair. The difference is that the ZPK numerator and denominator polynomials are factored, as in

$$G(s) = 2.5 \frac{s + 0.45}{(s + 0.3)(s + 0.1 + 0.7i)(s + 0.1 - 0.7i)}$$

(zeros and/or poles are complex numbers in general).

You define the ZPK model by specifying the zero(s), pole(s), and gain as in

```
poles = [-0.3, -0.1+0.7*i, -0.1-0.7*i];
Gzpk1 = zpk(-0.45, poles, 2.5);
```

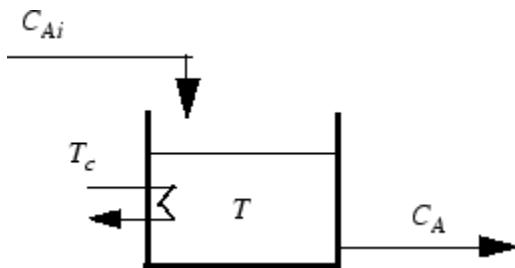
State-Space Models

The state-space format is convenient if your model is a set of LTI differential and algebraic equations. For example, consider the following linearized model of a continuous stirred-tank reactor (CSTR) involving an exothermic (heat-generating) reaction [1].

$$\frac{dC'_A}{dt} = a_{11}C'_A + a_{12}T' + b_{11}T_c' + b_{12}C'_{Ai}$$

$$\frac{dT'}{dt} = a_{21}C'_A + a_{22}T' + b_{21}T_c' + b_{22}C'_{Ai}$$

where C_A is the concentration of a key reactant, T is the temperature in the reactor, T_c is the coolant temperature, C_{Ai} is the reactant concentration in the reactor feed, and a_{ij} and b_{ij} are constants. See the process schematic in CSTR Schematic. The primes (e.g., C'_A) denote a deviation from the nominal steady-state condition at which the model has been linearized.



CSTR Schematic

Measurement of reactant concentrations is often difficult, if not impossible. Let us assume that T is a measured output, C_A is an unmeasured output, T_c is a manipulated variable, and C_{Ai} is an unmeasured disturbance.

The model fits the general state-space format

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

where

$$x = \begin{bmatrix} C'_A \\ T' \end{bmatrix}, u = \begin{bmatrix} T'_c \\ C'_{Ai} \end{bmatrix}, y = \begin{bmatrix} T' \\ C'_A \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The following code shows how to define such a model for some specific values of the a_{ij} and b_{ij} constants:

```
A = [-0.0285 -0.0014
      -0.0371 -0.1476];
B = [-0.0850 0.0238
      0.0802 0.4462];
C = [0 1
      1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);
```

This defines a *continuous-time* state-space model. If you do not specify a sampling period, a default sampling value of zero applies. You can also specify discrete-time state-space models. You can specify delays in both continuous-time and discrete-time models.

Note In the CSTR example, the D matrix is zero and the output does not instantly respond to change in the input. The Model Predictive Control Toolbox software prohibits direct (instantaneous) feedthrough from a manipulated variable to an output. For

example, the CSTR model could include direct feedthrough from the unmeasured disturbance, C_{Ai} , to either C_A or T but direct feedthrough from T_c to either output would violate this restriction. If the model had direct feedthrough from T_c , you can add a small delay at this input to circumvent the problem.

LTI Object Properties

The `ss` function in the last line of the above code creates a state space model, `CSTR`, which is an *LTI object*. The `tf` and `zpk` commands described in “Transfer Function Models” on page 2-9 and “Zero/Pole/Gain Models” on page 2-9 also create LTI objects. Such objects contain the model parameters as well as optional properties.

LTI Properties for the CSTR Example

The following code sets some of the `CSTR` model's optional properties:

```
CSTR.InputName = { T_c , C_A_i };  
CSTR.OutputName = { T , C_A };  
CSTR.StateName = { C_A , T };  
CSTRInputGroup.MV = 1;  
CSTRInputGroup.UD = 2;  
CSTROutputGroup.MO = 1;  
CSTROutputGroup.UO = 2;  
CSTR
```

The first three lines specify labels for the input, output and state variables. The next four specify the signal type for each input and output. The designations MV, UD, MO, and UO mean *manipulated variable*, *unmeasured disturbance*, *measured output*, and *unmeasured output*. (See “Signal Types” on page 2-8 for definitions.) For example, the code specifies that input 2 of model `CSTR` is an unmeasured disturbance. The last line causes the LTI object to be displayed, generating the following lines in the MATLAB Command Window:

```
a =  
      C_A          T  
C_A  -0.0285  -0.0014  
      T  -0.0371  -0.1476  
  
b =  
      T_c          C_Ai  
C_A  -0.085   0.0238  
      T  0.0802  0.4462
```

```

c =
      C_A      T
      T      0      1
      C_A     1      0

d =
      T_c   C_Ai
      T      0      0
      C_A    0      0

Input groups:
  Name    Channels
  MV        1
  UD        2

Output groups:
  Name    Channels
  MO        1
  UO        2

Continuous-time model

```

Input and Output Names

The optional `InputName` and `OutputName` properties affect the model displays, as in the above example. The software also uses the `InputName` and `OutputName` properties to label plots and tables. In that context, the underscore character causes the next character to be displayed as a subscript.

Input and Output Types

General Case

As mentioned in “Signal Types” on page 2-8, Model Predictive Control Toolbox software supports three input types and two output types. In a Model Predictive Control Toolbox design, designation of the input and output types determines the controller dimensions and has other important consequences.

For example, suppose your plant structure were as follows:

Plant Inputs	Plant Outputs
Two manipulated variables (MVs)	Three measured outputs (MOs)

Plant Inputs	Plant Outputs
One measured disturbance (MD)	Two unmeasured outputs (UOs)
Two unmeasured disturbances (UDs)	

The resulting controller has four inputs (the three MOs and the MD) and two outputs (the MVs). It includes feedforward compensation for the measured disturbance, and assumes that you wanted to include the unmeasured disturbances and outputs as part of the regulator design.

If you didn't want a particular signal to be treated as one of the above types, you could do one of the following:

- Eliminate the signal before using the model in controller design.
- For an output, designate it as unmeasured, then set its weight to zero.
- For an input, designate it as an unmeasured disturbance, then define a custom state estimator that ignores the input.

Note By default, the software assumes that unspecified plant inputs are manipulated variables, and unspecified outputs are measured. Thus, if you didn't specify signal types in the above example, the controller would have four inputs (assuming all plant outputs were measured) and five outputs (assuming all plant inputs were manipulated variables).

For model **CSTR**, the default Model Predictive Control Toolbox assumptions are incorrect. You must set its **InputGroup** and **OutputGroup** properties, as illustrated in the above code, or modify the default settings when you load the model into the MPC Designer app.

Use **setmpcsignals** to make type definition. For example:

```
CSTR = setmpcsignals(CSTR, UD, 2, UO, 2);
```

sets **InputGroup** and **OutputGroup** to the same values as in the previous example. The **CSTR** display would then include the following lines:

```
Input groups:  
Name      Channels  
Unmeasured    2  
Manipulated     1
```

```
Output groups:
  Name      Channels
Unmeasured    2
Measured      1
```

Notice that `setmpcsignals` sets unspecified inputs to **Manipulated** and unspecified outputs to **Measured**.

LTI Model Characteristics

Control System Toolbox software provides functions for analyzing LTI models. Some of the more commonly used are listed below. Type the example code at the MATLAB prompt to see how they work for the **CSTR** example.

Example	Intended Result
<code>dcgain(CSTR)</code>	Calculate gain matrix for the CSTR model's input/output pairs.
<code>impulse(CSTR)</code>	Graph CSTR model's unit-impulse response.
<code>linearSystemAnalyzer(CSTR)</code>	Open the Linear System Analyzer with the CSTR model loaded. You can then display model characteristics by making menu selections.
<code>pole(CSTR)</code>	Calculate CSTR model's poles (to check stability, etc.).
<code>step(CSTR)</code>	Graph CSTR model's unit-step response.
<code>zero(CSTR)</code>	Compute CSTR model's transmission zeros.

References

- [1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004, pp. 34–36 and 94–95.

See Also

`setmpcsignals` | `ss` | `tf` | `zpk`

Related Examples

- “Specify Multi-Input Multi-Output (MIMO) Plants” on page 2-17

Specify Multi-Input Multi-Output (MIMO) Plants

This example shows how to specify plants having multiple inputs and outputs.

Most MPC applications involve plants with multiple inputs and outputs. You can use `ss`, `tf`, and `zpk` to represent a MIMO plant model. For example, consider the following model of a distillation column [1], which has been used in many advanced control studies:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{12.8e^{-s}}{16.7s+1} & \frac{-18.9e^{-3s}}{21.0s+1} & \frac{3.8e^{-8.1s}}{14.9s+1} \\ \frac{6.6e^{-7s}}{10.9s+1} & \frac{-19.4e^{-3s}}{14.4s+1} & \frac{4.9e^{-3.4s}}{13.2s+1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

Outputs y_1 and y_2 represent measured product purities. The controller manipulates the inputs, u_1 and u_2 , to hold each output at a specified setpoint. These inputs represent the flow rates of reflux and reboiler steam, respectively. Input u_3 is a measured feed flow rate disturbance.

The model consists of six transfer functions, one for each input/output pair. Each transfer function is the first-order-plus-delay form often used by process control engineers.

Specify the individual transfer functions for each input/output pair. For example, `g12` is the transfer function from input u_1 to output y_2 .

```
g11 = tf( 12.8, [16.7 1], 'IODelay', 1.0, TimeUnit, minutes );
g12 = tf(-18.9, [21.0 1], 'IODelay', 3.0, TimeUnit, minutes );
g13 = tf( 3.8, [14.9 1], 'IODelay', 8.1, TimeUnit, minutes );
g21 = tf( 6.6, [10.9 1], 'IODelay', 7.0, TimeUnit, minutes );
g22 = tf(-19.4, [14.4 1], 'IODelay', 3.0, TimeUnit, minutes );
g23 = tf( 4.9, [13.2 1], 'IODelay', 3.4, TimeUnit, minutes );
```

Define a MIMO system by creating a matrix of transfer function models.

```
DC = [g11 g12 g13
      g21 g22 g23];
```

Define the input and output signal names and specify the third input as a measured input disturbance.

```
DC.InputName = { 'Reflux Rate', 'Steam Rate', 'Feed Rate' };
DC.OutputName = { 'Distillate Purity', 'Bottoms Purity' };
```

```
DC = setmpcsignals(DC, MD ,3);  
-->Assuming unspecified input signals are manipulated variables.
```

Review the resulting system.

```
DC
```

```
DC =
```

```
From input "Reflux Rate" to output...  
12.8  
Distillate Purity: exp(-1*s) * -----  
16.7 s + 1  
  
6.6  
Bottoms Purity: exp(-7*s) * -----  
10.9 s + 1  
  
From input "Steam Rate" to output...  
-18.9  
Distillate Purity: exp(-3*s) * -----  
21 s + 1  
  
-19.4  
Bottoms Purity: exp(-3*s) * -----  
14.4 s + 1  
  
From input "Feed Rate" to output...  
3.8  
Distillate Purity: exp(-8.1*s) * -----  
14.9 s + 1  
  
4.9  
Bottoms Purity: exp(-3.4*s) * -----  
13.2 s + 1  
  
Input groups:  
Name Channels  
Measured 3  
Manipulated 1,2  
  
Output groups:  
Name Channels  
Measured 1,2
```

Continuous-time transfer function.

References

- [1] Wood, R. K., and M. W. Berry, *Chem. Eng. Sci.*, Vol. 28, pp. 1707, 1973.

See Also

`setmpcsignals` | `ss` | `tf` | `zpk`

Related Examples

- “Construct Linear Time Invariant (LTI) Models” on page 2-9

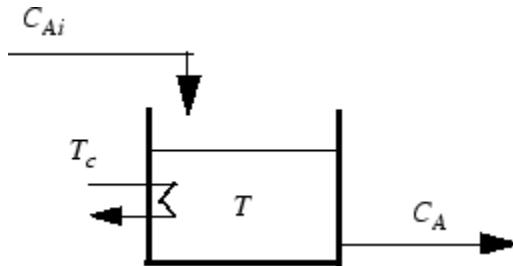
CSTR Model

The linearized model of a continuous stirred-tank reactor (CSTR) involving an exothermic (heat-generating) reaction is represented by the following differential equations:

$$\frac{dC'_A}{dt} = a_{11}C'_A + a_{12}T' + b_{11}T'_c + b_{12}C'_{Ai}$$

$$\frac{dT'}{dt} = a_{21}C'_A + a_{22}T' + b_{21}T'_c + b_{22}C'_{Ai}$$

where C_A is the concentration of a key reactant, T is the temperature in the reactor, T_c is the coolant temperature, C_{Ai} is the reactant concentration in the reactor feed, and a_{ij} and b_{ij} are constants. The primes (e.g., C'_A) denote a deviation from the nominal steady-state condition at which the model has been linearized.



Measurement of reactant concentrations is often difficult, if not impossible. Let us assume that T is a measured output, C_A is an unmeasured output, T_c is a manipulated variable, and C_{Ai} is an unmeasured disturbance.

The model fits the general state-space format

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

where

$$x = \begin{bmatrix} C'_A \\ T' \end{bmatrix}, u = \begin{bmatrix} T'_c \\ C'_{Ai} \end{bmatrix}, y = \begin{bmatrix} T' \\ C'_A \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The following code shows how to define such a model for some specific values of the a_{ij} and b_{ij} constants:

```
A = [-0.0285 -0.0014
      -0.0371 -0.1476];
B = [-0.0850 0.0238
      0.0802 0.4462];
C = [0 1
      1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);
```

The following code sets some of the CSTR model's optional properties:

```
CSTR.InputName = { 'T_c' , 'C_A_i' };
CSTR.OutputName = { 'T' , 'C_A' };
CSTR.StateName = { 'C_A' , 'T' };
CSTRInputGroup.MV = 1;
CSTRInputGroup.UD = 2;
CSTROutputGroup.MO = 1;
CSTROutputGroup.UO = 2;
```

To view the properties of CSTR, enter:

```
CSTR
```

Linearize Simulink Models

Generally, real systems are nonlinear. To design an MPC controller for a nonlinear system, you must model the plant in Simulink.

Although an MPC controller can regulate a nonlinear plant, the model used within the controller must be linear. In other words, the controller employs a linear approximation of the nonlinear plant. The accuracy of this approximation significantly affects controller performance.

To obtain such a linear approximation, you *linearize* the nonlinear plant at a specified *operating point*.

Note: Simulink Control Design™ software must be installed to linearize nonlinear Simulink models.

You can linearize a Simulink model:

- From the command line.
- Using the Linear Analysis Tool.
- Using the MPC Designer app.

Linearization Using MATLAB Code

This example shows how to obtain a linear model of a plant using a MATLAB script.

For this example the CSTR model, `CSTR_OpenLoop`, is linearized. The model inputs are the coolant temperature (manipulated variable of the MPC controller), limiting reactant concentration in the feed stream, and feed temperature. The model states are the temperature and concentration of the limiting reactant in the product stream. Both states are measured and used for feedback control.

Obtain Steady-State Operating Point

The operating point defines the nominal conditions at which you linearize a model. It is usually a steady-state condition.

Suppose that you plan to operate the CSTR with the output concentration, C_A , at 2 kmol/m^3 . The nominal feed concentration is 10 kmol/m^3 , and the nominal feed

temperature is 300 K. Create an operating point specification object to define the steady-state conditions.

```
opspec = operspec( CSTR_OpenLoop );
opspec = addoutputs(opspec, CSTR_OpenLoop/CSTR ,2);
opspec.Outputs(1).Known = true;
opspec.Outputs(1).y = 2;

op1 = findop( CSTR_OpenLoop ,opspec);
```

Operating Point Search Report:

Operating Report for the Model CSTR_OpenLoop.
(Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:

```
-----
(1.) CSTR_OpenLoop/CSTR/C_A
    x:          2      dx:     -4.6e-12 (0)
(2.) CSTR_OpenLoop/CSTR/T_K
    x:         373      dx:      5.49e-11 (0)
```

Inputs:

```
-----
(1.) CSTR_OpenLoop/Coolant Temperature
    u:          299      [-Inf Inf]
```

Outputs:

```
-----
(1.) CSTR_OpenLoop/CSTR
    y:          2      (2)
```

The calculated operating point is $C_A = 2 \text{ kmol/m}^3$ and $T_K = 373 \text{ K}$. Notice that the steady-state coolant temperature is also given as 299 K, which is the nominal value of the manipulated variable of the MPC controller.

To specify:

- Values of known inputs, use the `Input.Known` and `Input.u` fields of `opspec`
- Initial guesses for state values, use the `State.x` field of `opspec`

For example, the following code specifies the coolant temperature as 305 K and initial guess values of the C_A and T_K states before calculating the steady-state operating point:

```
opspec = operspec( CSTR_OpenLoop );
opspec.States(1).x = 1;
opspec.States(2).x = 400;
opspec.Inputs(1).Known = true;
opspec.Inputs(1).u = 305;

op2 = findop( CSTR_OpenLoop ,opspec);
```

Operating Point Search Report:

```
-----
Operating Report for the Model CSTR_OpenLoop.
(Time-Varying Components Evaluated at time t=0)
```

Operating point specifications were successfully met.

States:

```
-----
(1.) CSTR_OpenLoop/CSTR/C_A
      x:           1.78    dx:     -8.88e-15 (0)
(2.) CSTR_OpenLoop/CSTR/T_K
      x:           377    dx:     1.14e-13 (0)
```

Inputs:

```
-----
(1.) CSTR_OpenLoop/Coolant Temperature
      u:           305
```

Outputs: None

Specify Linearization Inputs and Outputs

If the linearization input and output signals are already defined in the model, as in **CSTR_OpenLoop**, then use the following to obtain the signal set.

```
io = getlinio( CSTR_OpenLoop );
```

Otherwise, specify the input and output signals as shown here.

```
io(1) = linio( CSTR_OpenLoop/Feed Concentration , 1, input );
io(2) = linio( CSTR_OpenLoop/Feed Temperature , 1, input );
io(3) = linio( CSTR_OpenLoop/Coolant Temperature , 1, input );
io(4) = linio( CSTR_OpenLoop/CSTR , 1, output );
io(5) = linio( CSTR_OpenLoop/CSTR , 2, output );
```

Linearize Model

Linearize the model using the specified operating point, `op1`, and input/output signals, `io`.

```
sys = linearize( CSTR_OpenLoop , op1, io)
```

```
sys =
```

```
A =
```

$$\begin{matrix} & \text{C_A} & \text{T_K} \\ \text{C_A} & -5 & -0.3427 \\ \text{T_K} & 47.68 & 2.785 \end{matrix}$$

```
B =
```

$$\begin{matrix} & \text{Feed Concent} & \text{Feed Tempera} & \text{Coolant Temp} \\ \text{C_A} & 1 & 0 & 0 \\ \text{T_K} & 0 & 1 & 0.3 \end{matrix}$$

```
C =
```

$$\begin{matrix} & \text{C_A} & \text{T_K} \\ \text{CSTR/1} & 0 & 1 \\ \text{CSTR/2} & 1 & 0 \end{matrix}$$

```
D =
```

$$\begin{matrix} & \text{Feed Concent} & \text{Feed Tempera} & \text{Coolant Temp} \\ \text{CSTR/1} & 0 & 0 & 0 \\ \text{CSTR/2} & 0 & 0 & 0 \end{matrix}$$

Continuous-time state-space model.

Linearization Using Linear Analysis Tool in Simulink Control Design

This example shows how to linearize a Simulink model using the Linear Analysis Tool, provided by the Simulink Control Design product.

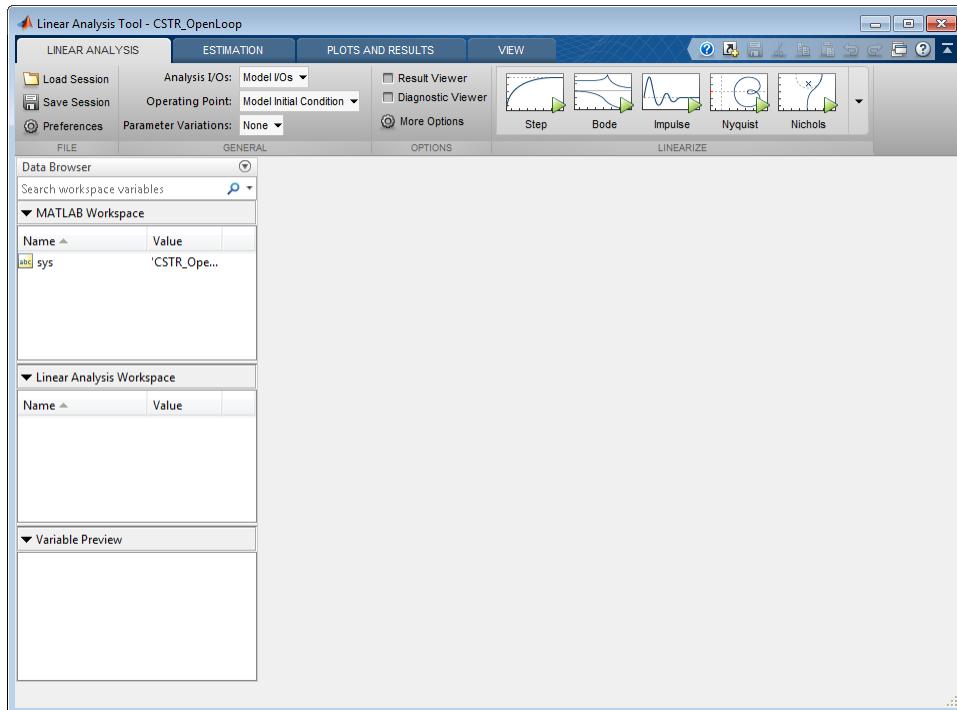
This example uses the CSTR model, `CSTR_OpenLoop`.

Open Simulink Model

```
sys = CSTR_OpenLoop ;
open_system(sys)
```

Open Linear Analysis Tool

In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.



Specify Linearization Inputs and Outputs

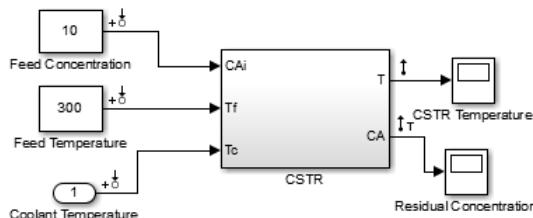
The linearization inputs and outputs are already specified for `CSTR_OpenLoop`. The input signals correspond to the outputs from the `Feed Concentration`, `Feed Temperature`, and `Coolant Temperature` blocks. The output signals are the inputs to the `CSTR Temperature` and `Residual Concentration` blocks.

To specify a signal as a:

- Linearization input, right-click the signal in the Simulink model window and select **Linear Analysis Points > Input Perturbation**.
- Linearization output, right-click the signal in the Simulink model window and select **Linear Analysis Points > Output Measurement**.

Specify Residual Concentration as Known Trim Constraint

In the Simulink model window, right-click the CA output signal from the CSTR block. Select **Linear Analysis Points > Trim Output Constraint**.

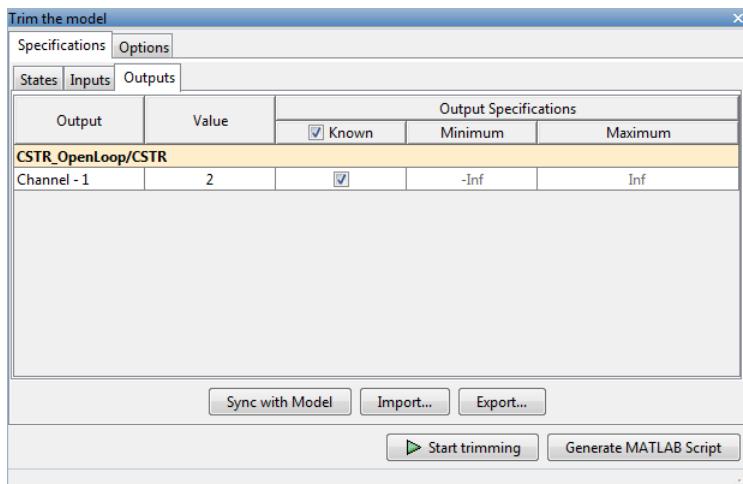


Copyright 1990-2012 The MathWorks, Inc.

In the Linear Analysis Tool, in the **Linear Analysis** tab, in the **Operating Point** drop-down list, select **Trim model**.

In the **Outputs** tab:

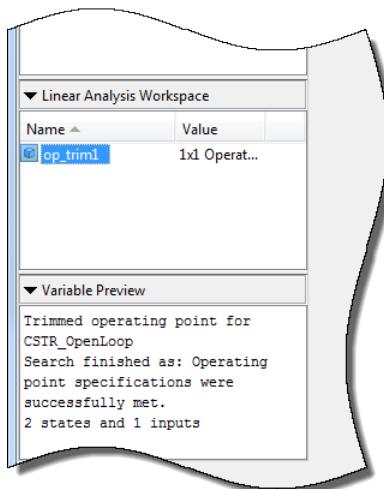
- Select the **Known** check box for Channel - 1 under **CSTR_OpenLoop/CSTR**.
- Set the corresponding **Value** to 2 kmol/m³.



Create and Verify Operating Point

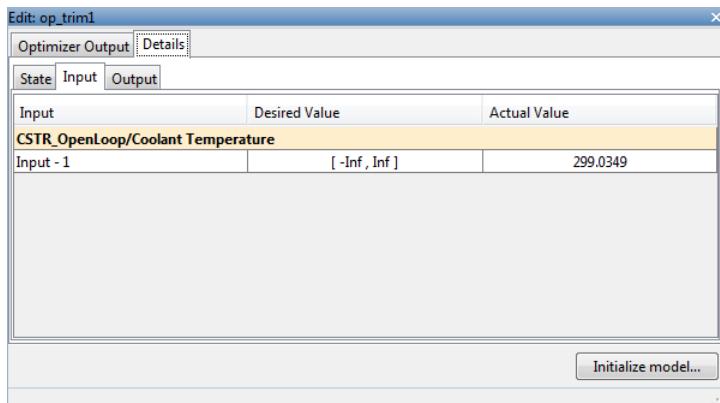
In the Trim the model dialog box, click **Start trimming**.

The operating point `op_trim1` displays in the **Linear Analysis Workspace**.



Double click `op_trim1` to view the resulting operating point.

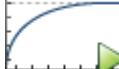
In the Edit dialog box, select the **Input** tab.



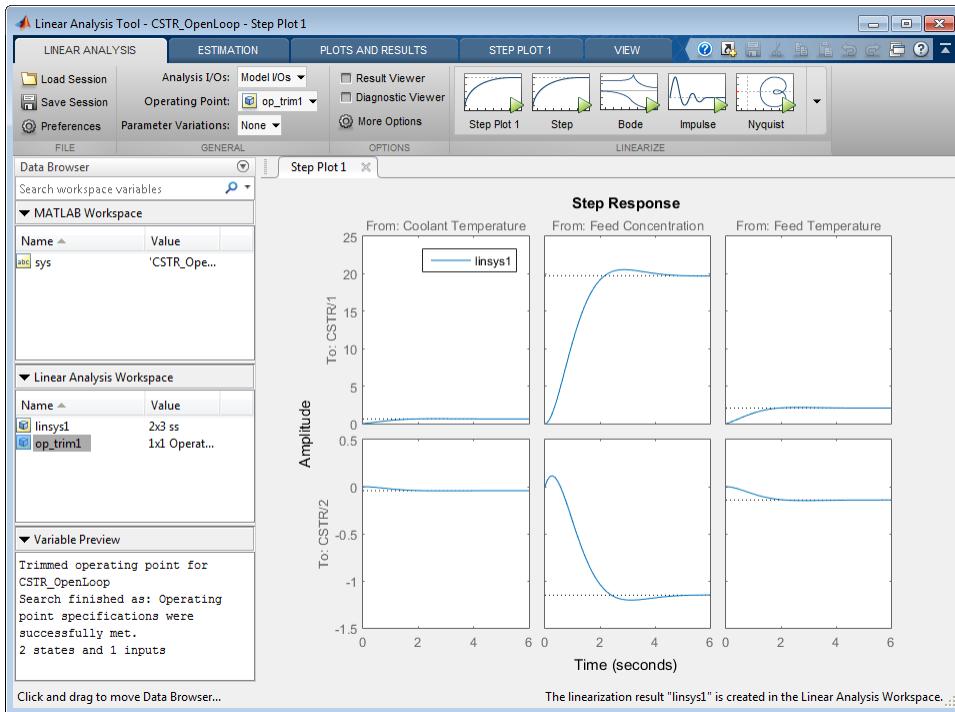
The coolant temperature at steady state is 299 K, as desired.

Linearize Model

In the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `op_trim1`.

Click  **Step** to linearize the model.

This option creates the linear model `linsys1` in the **Linear Analysis Workspace** and generates a step response for this model. `linsys1` uses `optrim1` as its operating point.



The step response from feed concentration to output **CSTR/2** displays an interesting inverse response. An examination of the linear model shows that **CSTR/2** is the residual CSTR concentration, C_A . When the feed concentration increases, C_A increases initially because more reactant is entering, which increases the reaction rate. This rate increase results in a higher reactor temperature (output **CSTR/1**), which further increases the reaction rate and C_A decreases dramatically.

Export Linearization Result

If necessary, you can repeat any of these steps to improve your model performance. Once you are satisfied with your linearization result, in the Linear Analysis Tool, drag and drop it from the **Linear Analysis Workspace** to the **MATLAB Workspace**. You can now use your linear model to design an MPC controller.

Related Examples

- “Design MPC Controller in Simulink” on page 5-2

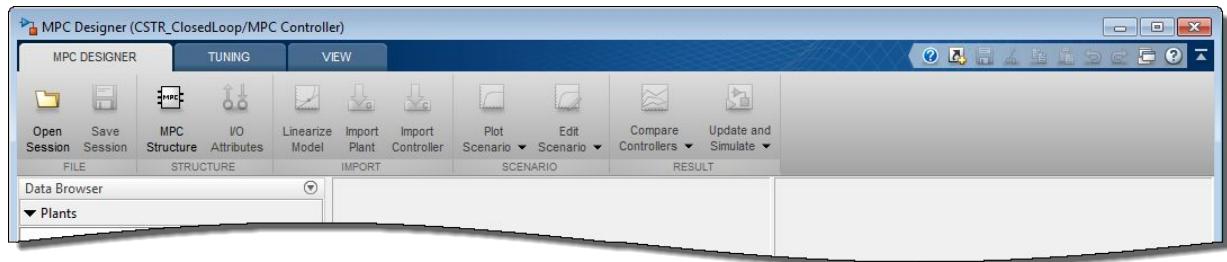
- “Design Controller Using MPC Designer” on page 3-2
- “Design MPC Controller at the Command Line” on page 4-2

Linearize Simulink Models Using MPC Designer

This topic shows how to linearize Simulink models using the MPC Designer app. To do so, start the app from a Simulink model that contains an **MPC Controller** block.

In the model window, double-click the **MPC Controller** block.

In the Block Parameters dialog box, ensure that the **MPC Controller** field is empty, and click **Design** to open MPC Designer.



Using the MPC Designer app, you can define the MPC structure by linearizing the Simulink model. After you define the initial MPC structure, you can also linearize the model at different operating points and import the linearized plants.

Note: If a controller from the MATLAB workspace is specified in the **MPC Controller** field, the app imports the specified controller. In this case, the MPC structure is derived from the imported controller. In this case, you can still linearize the Simulink model and import the linearized plants.

Define MPC Structure By Linearization

This example shows how to define the plant input/output structure in the MPC Designer app by linearizing a Simulink model.

On the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

Define MPC Structure By Linearization

MPC Structure

```

graph LR
    SD[Setpoints (reference)] --> MPC[MPC]
    UDI[Unmeasured Disturbances] --> MPC
    MPC -- Manipulated Variables --> P[Plant]
    MPC -- Measured Disturbances --> P
    P -- Unmeasured --> O[Outputs]
    P -- Measured --> O
    P -- Unmeasured --> MD[Measured Disturbances]
    MD --> MPC
  
```

Controller Sample Time
Specify MPC controller sample time (default sample time in the MPC block):

Simulink Operating Point
Choose an operating point at which plant model is linearized and nominal values are computed:

Simulink Signals for Plant Inputs

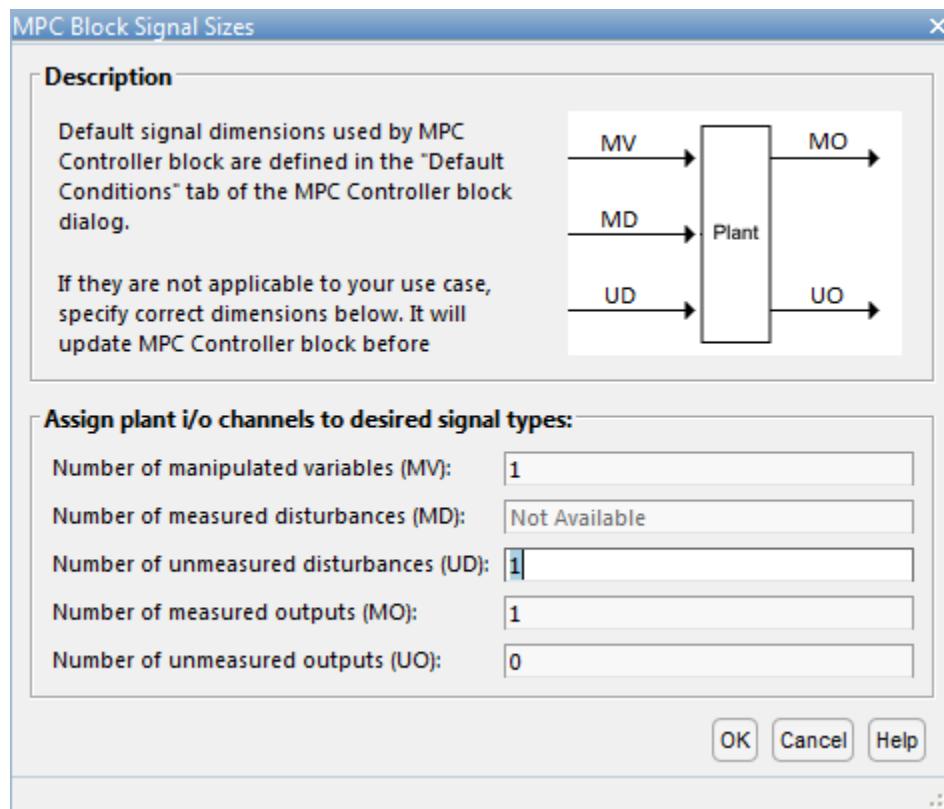
Selected	Type	Block Path
<input checked="" type="radio"/>	Manipulated Variables (MV)	CSTR_ClosedLoop/MPC_Controller:1

Simulink Signals for Plant Outputs

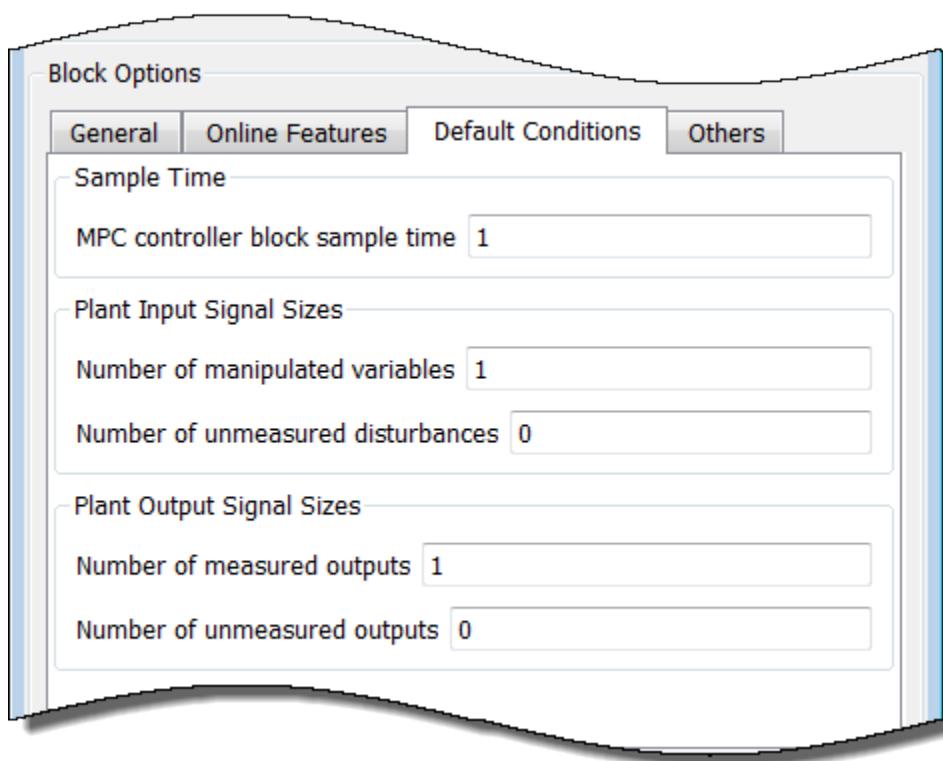
Selected	Type	Block Path
<input checked="" type="radio"/>	Measured Outputs (MO)	CSTR_ClosedLoop/CSTR:2

Specify Signal Dimensions

In the Define MPC Structure By Linearization dialog box, in the **MPC Structure** section, if the displayed signal dimensions do not match your model, click **Change I/O Sizes** to configure the dimensions. Any unmeasured disturbances or unmeasured outputs in your model are not detected by the **MPC Controller** block. Specify the dimensions for these signals.



Tip In the MPC Controller Block Parameters dialog box, in the **Default Conditions** tab, you can define the controller sample time and signal dimensions before opening MPC Designer.

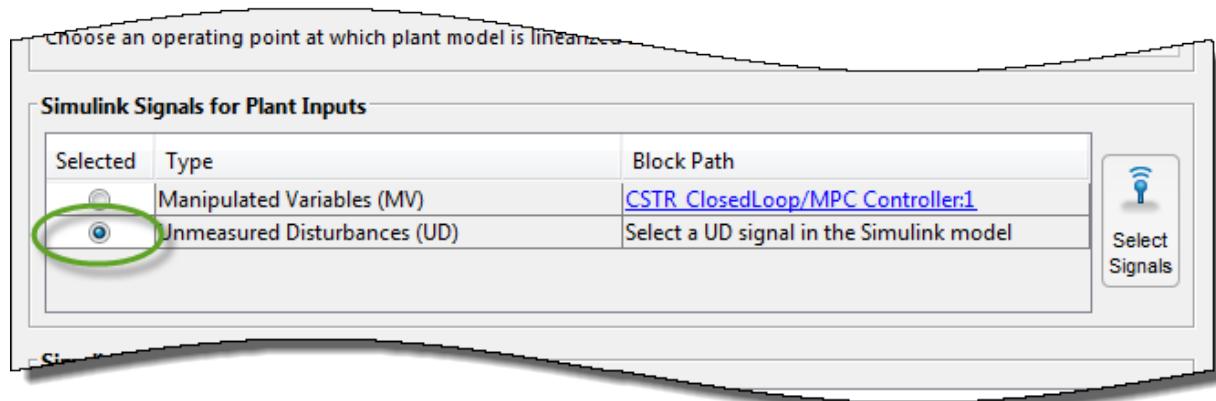


Select Plant Input/Output Signals

Before linearizing the model, assign Simulink signal lines to each MPC signal type in your model. The app uses these signals as linearization inputs and outputs.

In the **Simulink Signals for Plant Inputs** and **Simulink Signals for Plant Outputs** sections, the **Block Path** is automatically defined for manipulated variables, measured outputs, and measured disturbances. MPC Designer detects these signals since they are connected to the **MPC Controller** block. If your application has unmeasured disturbances or unmeasured outputs, select their corresponding Simulink signal lines.

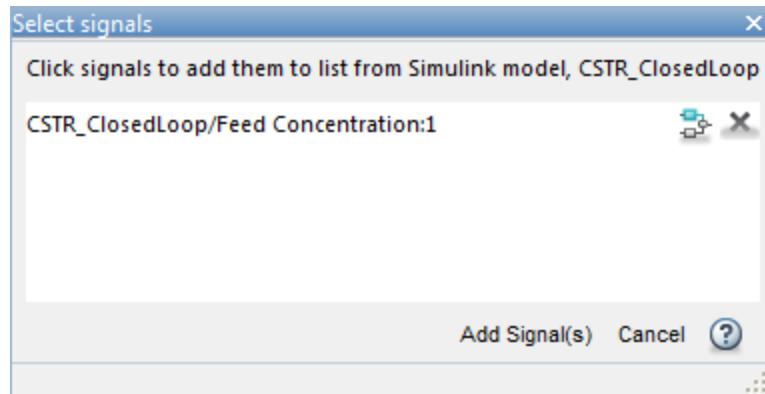
To choose a signal type, use the **Selected** option buttons.



Click **Select Signals**.

In the Simulink model window, click the signal line corresponding to the selected signal type.

The signal is highlighted, and its block path is added to the Select signals dialog box.



In the Select signals dialog box, click **Add Signal(s)**.

In the Define MPC Structure By Linearization dialog box, the **Block Path** for the selected signal type updates.

Specify Operating Point

In the **Simulink Operating Point** section, in the drop-down list, select an operating point at which to linearize the model.

For information on the different operating point options, see “Specifying Operating Points” on page 2-39.

Note: If you select an option that generates multiple operating points for linearization, MPC Designer uses only the first operating point to define the plant structure and linearize the model.

Define Structure and Linearize Model

Click **Define and Linearize**.

The app linearizes the Simulink model at the specified operating point using the specified input/output signals, and adds the linearized plant to the **Data Browser**.

Also, a default controller, which uses the linearized plant as its internal model, and a default simulation scenario are created.

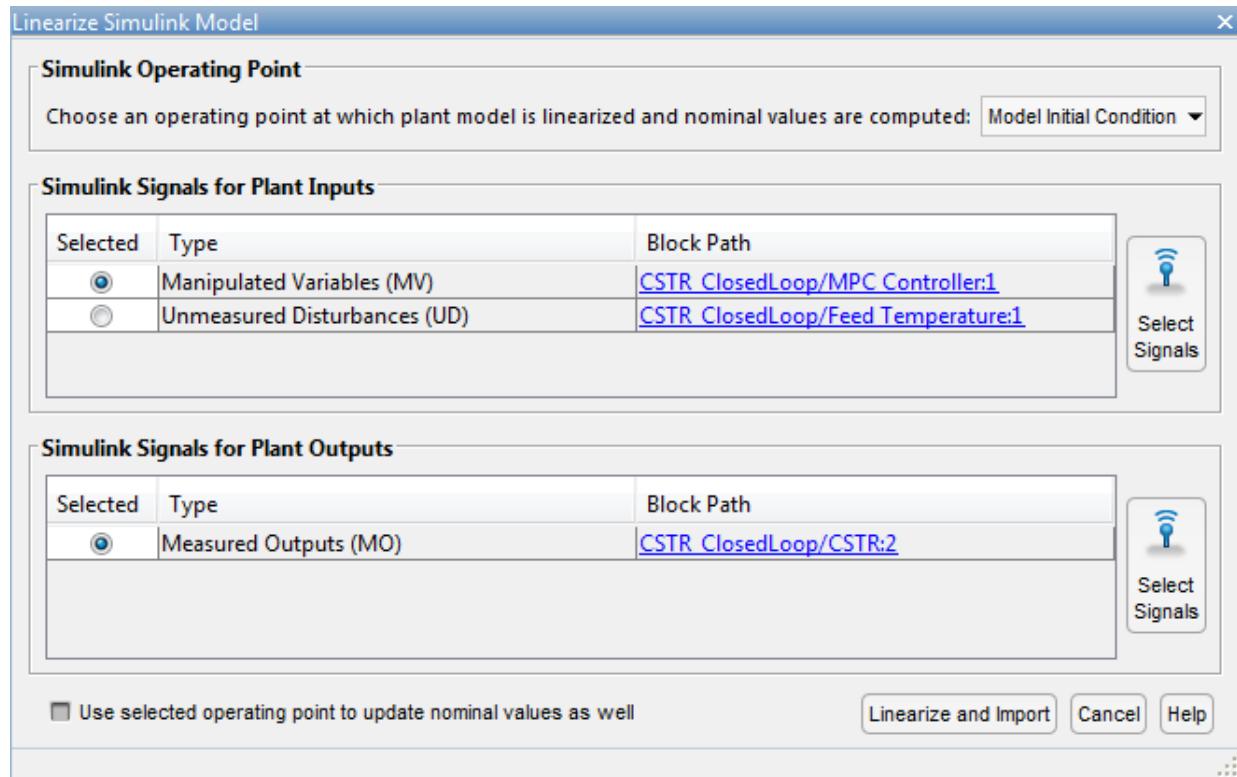
MPC Designer uses the input/output signal values at the selected operating point as nominal values.

Linearize Model

This example shows how to linearize and import a Simulink model using the MPC Designer app.

After you define the initial MPC structure, you can linearize the Simulink model at different operating points and import the linearized plants. Doing so is useful for validating controller performance against modeling errors.

On the **MPC Designer** tab, in the **Import** section, click **Linearize Model**.



Select Plant Input/Output Signals

In the **Simulink Signals for Plant Inputs** and **Simulink Signals for Plant Outputs** sections, the input/output signal configuration is the same as you specified when initially defining the MPC structure.

You cannot change the signal types and dimensions once the structure has been defined. However, for each signal type, you can select different signal lines from your Simulink model. The selected lines must have the same dimensions as defined in the current MPC structure.

Specify Operating Point

In the **Simulink Operating Point** section, in the drop-down list, select the operating points at which to linearize the model.

For information on the different operating point options, see “Specifying Operating Points” on page 2-39.

Linearize Model and Import Plant

Click **Linearize and Import**.

MPC Designer linearizes the Simulink model at the defined operating point using the specified input/output signals, and adds the linearized plant to the **Data Browser**.

If you select the **Use selected operating point to update nominal values as well** option, the app updates the controller nominal values using the operating point signal values.

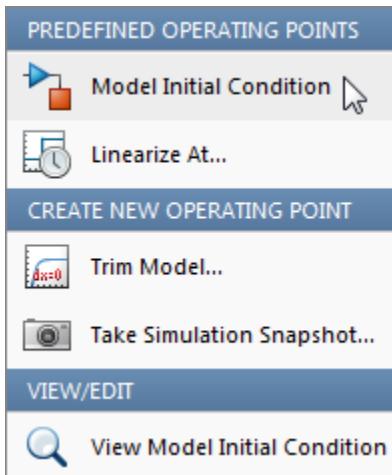
If you select an option that generates multiple operating points for linearization, the app linearizes the model at all the specified operating points. The linearized plants are added to the **Data Browser** in the same order in which their corresponding operating points are defined. If you choose to update the nominal values, the app uses the signal values from the first operating point.

Specifying Operating Points

In the **Simulink Operating Point** section, in the drop-down list, you can select or create operating points for model linearization. For more information on finding steady-state operating points, see “About Operating Points” and “Computing Steady-State Operating Points” in the Simulink Control Design documentation.

Select Model Initial Condition

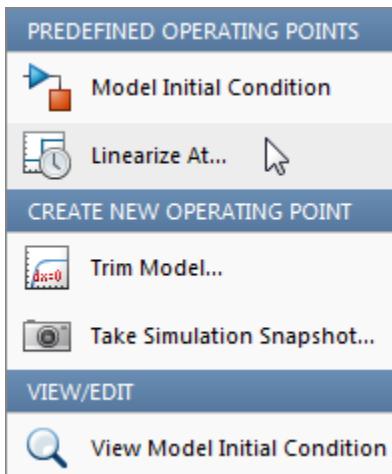
To linearize the model using the initial conditions specified in the Simulink model as the operating point, select **Model Initial Condition**.



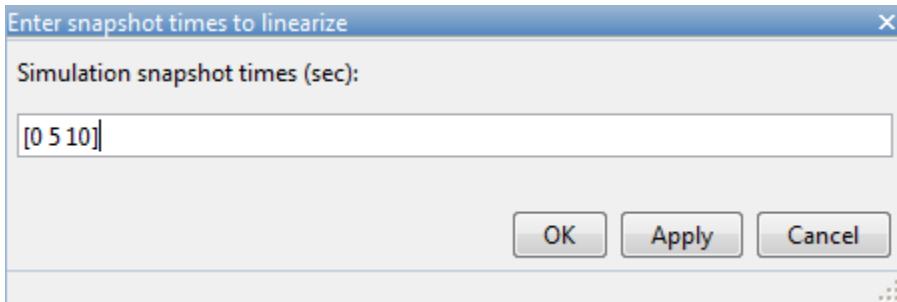
The model initial condition is the default operating point for linearization in the MPC Designer app.

Linearize at Simulation Snapshot Times

To linearize the model at specified simulation snapshot times, select **Linearize At**. Linearizing at snapshot times is useful when you know that your model reaches an equilibrium state after a certain simulation time.



In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



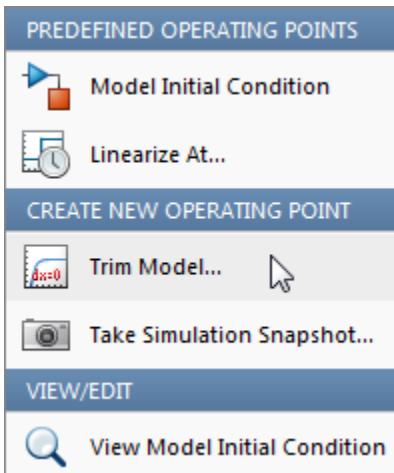
Click **OK**.

If you enter multiple snapshot times, and you selected **Linearize At** from the:

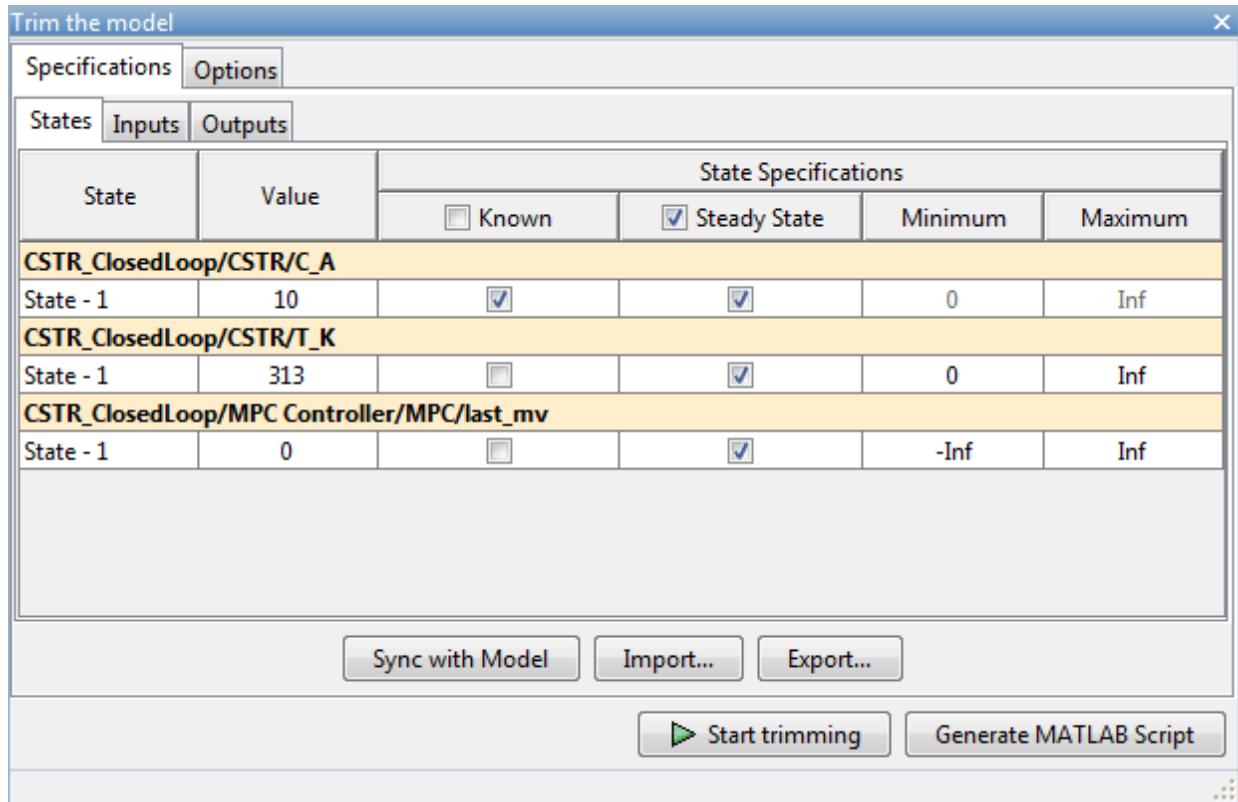
- Define MPC Structure By Linearization dialog box, MPC Designer linearizes the model using only the first snapshot time. The nominal values of the MPC controller are defined using the input/output signal values for this snapshot.
- Linearize Simulink Model dialog box, MPC Designer linearizes the model at all the specified snapshot times. The linearized plant models are added to the **Data Browser** in the order specified in the snapshot time array. If you selected the **Use selected operating point to update nominal values as well** option, the nominal values are set using the input/output signal values from the first snapshot.

Compute Steady-State Operating Point

To compute a steady-state operating point using numerical optimization methods to meet your specifications, select **Trim Model**.



In the Trim the model dialog box, enter the specifications for the steady-state state values at which you want to find an operating point.



Click **Start Trimming**.

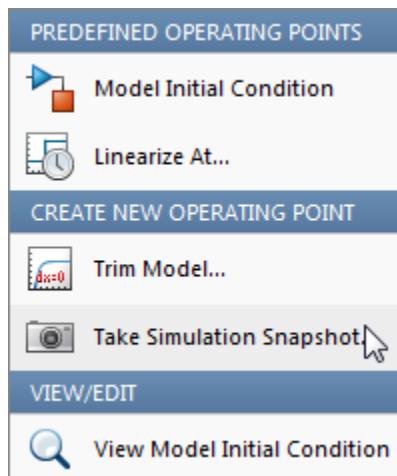
MPC Designer creates an operating point for the given specifications. The computed operating point is added to the **Simulink Operating Point** drop-down list and is selected.

For examples showing how to specify the conditions for a steady-state operating point search, see “Compute Steady-State Operating Points from State Specifications” and “Compute Steady-State Operating Point to Meet Output Specification” in the Simulink Control Design documentation.

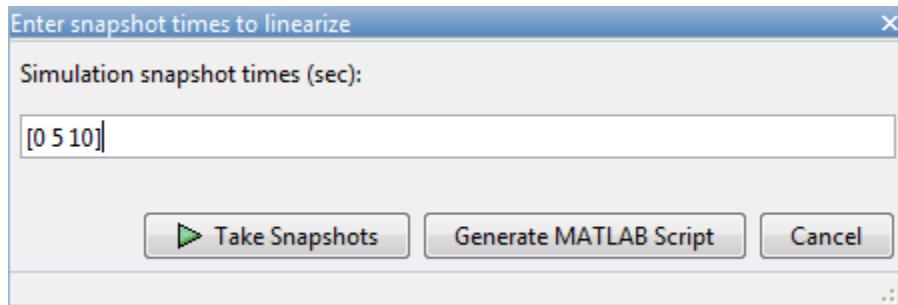
Compute Operating Point at Simulation Snapshot Time

To compute operating points using simulation snapshots, select **Take Simulation Snapshot**. Linearizing the model using operating points computed from simulation

snapshots is useful when you know that your model reaches an equilibrium state after a certain simulation time.



In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



Click **Take Snapshots**.

MPC Designer simulates the Simulink model. At each snapshot time, the current state of the model is used to create an operating point, which is added to the drop-down list and selected.

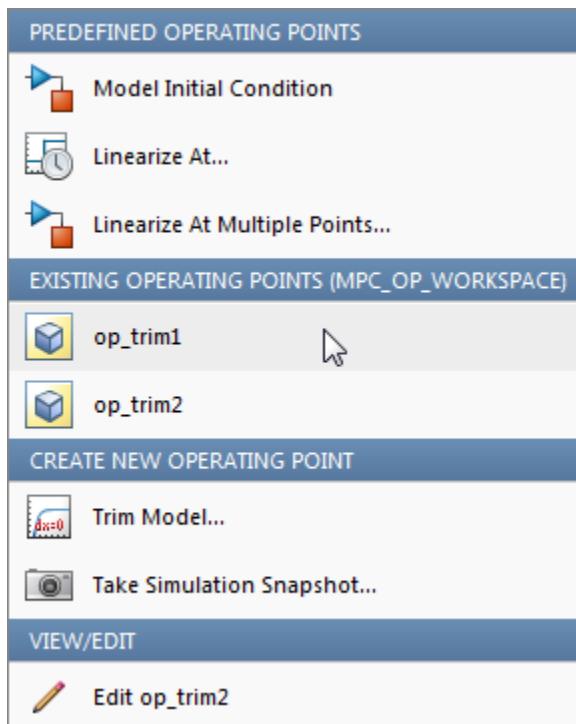
If you entered multiple snapshot times, the operating points are stored together as an array. If you selected **Take Simulation Snapshot** from the:

- Define MPC Structure By Linearization dialog box, MPC Designer linearizes the model using only the first operating point in the array. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.
- Linearize Simulink Model dialog box, MPC Designer linearizes the model at all the operating points in the array. The linearized plant models are added to the **Data Browser** in the same order as the operating point array.

In the MPC Designer app, the **Linearize At** and **Take Simulation Snapshot** options generally produce the same linearized plant and nominal signal values. However, since the **Take Simulation Snapshot** option first computes an operating point from the snapshot before linearization, the results can differ.

Select Existing Operating Point

Under **Existing Operating Points**, select a previously defined operating point at which to linearize the Simulink model. This option is available if one or more previously created operating points are available in the drop-down list.

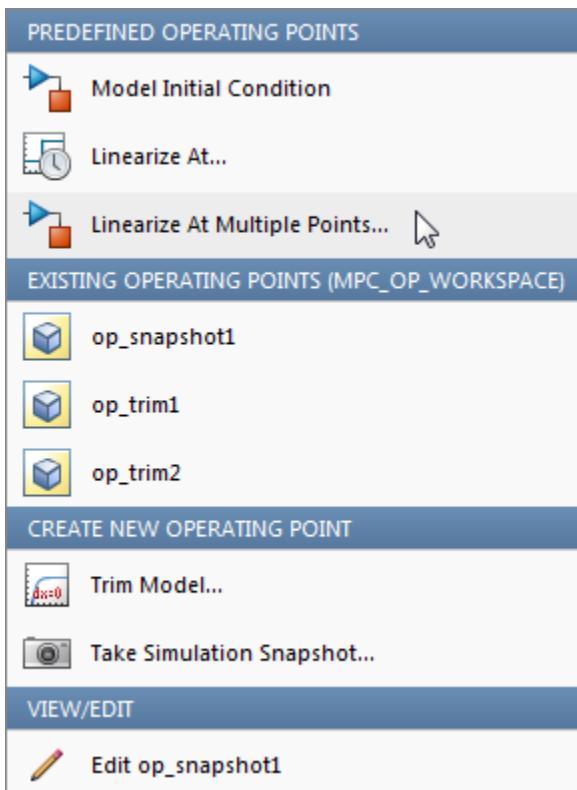


If the selected operating point represents an operating point array created using multiple snapshot times, and you selected an operating point from the:

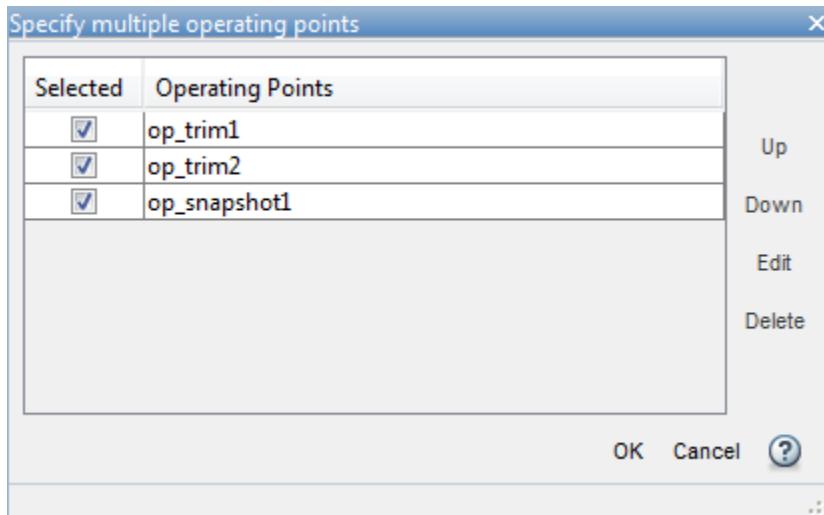
- Define MPC Structure By Linearization dialog box, MPC Designer linearizes the model using only the first operating point in the array. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.
- Linearize Simulink Model dialog box, MPC Designer linearizes the model at all the operating points in the array. The linearized plant models are added to the **Data Browser** in the same order as the operating point array.

Select Multiple Operating Points

To linearize the Simulink model at multiple existing operating points, select **Linearize at Multiple Points**. This option is available if there are more than one previously created operating points in the drop-down list.



In the Specify multiple operating points dialog box, select the operating points at which to linearize the model.



To change the operating point order, click an operating point in the list and click **Up** or **Down** to move the highlighted operating point within the list.

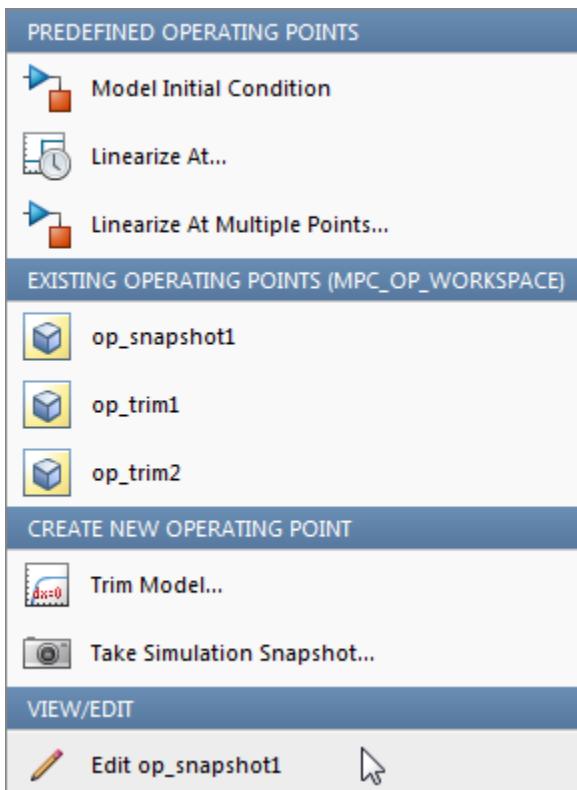
Click **OK**.

If you selected **Linearize at Multiple Points** from the:

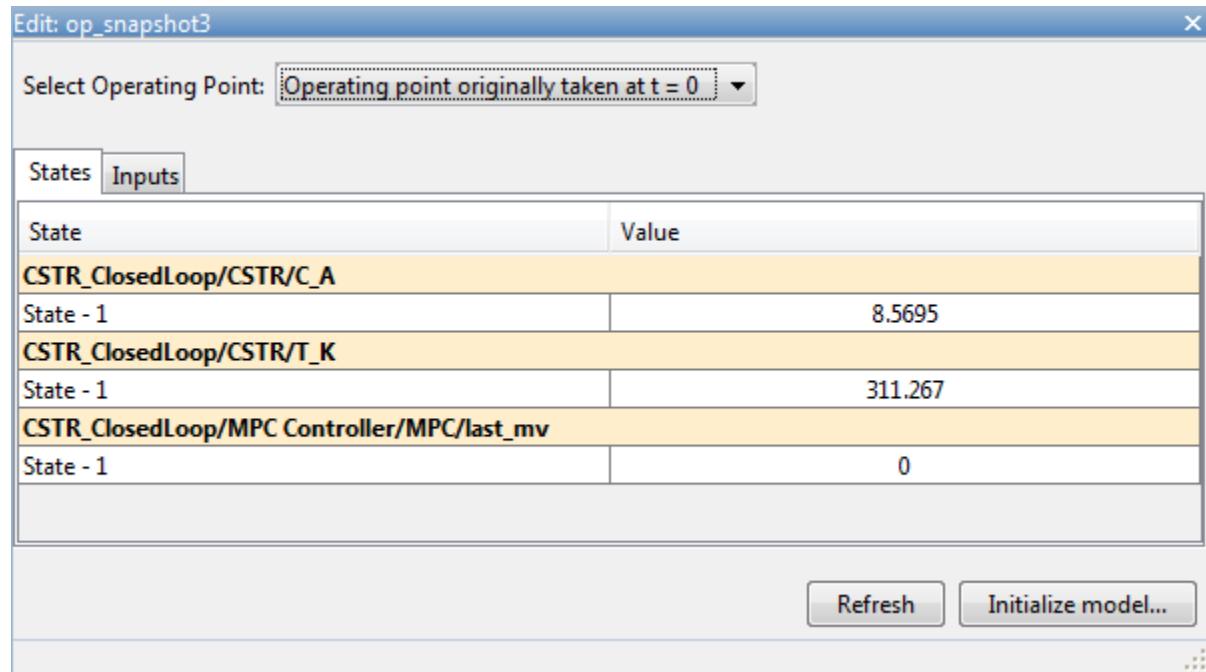
- Define MPC Structure By Linearization dialog box, MPC Designer linearizes the model using only the first specified operating point. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.
- Linearize Simulink Model dialog box, MPC Designer linearizes the model at all the specified operating points. The linearized plant models are added to the **Data Browser** in the order specified in the Specify multiple operating points dialog box.

View/Edit Operating Point

To view or edit the selected operating point, under **View/Edit**, click the **Edit** option.



In the Edit dialog box, if you created the selected operating point from a simulation snapshot, you can edit the operating point values.



If the selected operating point represents an operating point array, in the **Select Operating Point** drop-down list, select an operating point to view.

If you obtained the operating point by trimming the model, you can only view the operating point values.

Edit: op_trim1				
Optimizer Output		Details		
State	Input	Output		
State	Desired Value	Actual Value	Desired dx	Actual dx
CSTR_ClosedLoop/CSTR/C_A				
State - 1	10	10	0	-3.5732e-08
CSTR_ClosedLoop/CSTR/T_K				
State - 1	[0 , Inf]	161.9721	0	4.3283e-07
CSTR_ClosedLoop/MPC Controller/MPC/last_mv				
State - 1	[-Inf , Inf]	-298.1209	0	0

Initialize model...

To simulate the Simulink model at the selected operating point, click **Initialize model** to set the model initial conditions to the states in the operating point.

See Also

MPC Designer

Related Examples

- “Linearize Simulink Models” on page 2-22
- “Design MPC Controller in Simulink” on page 5-2

Identify Plant from Data

This example shows how to identify a linear plant model using measured data.

When you have measured plant input/output data, you can use System Identification Toolbox™ software to estimate a linear plant model. Then, you can use the estimated plant model to create a model predictive controller.

You can estimate the plant model either programmatically or by using the System Identification app.

This example requires a System Identification Toolbox license.

Load the measured data.

```
load dryer2
```

The variables `u2` and `y2`, which contain the data measured for a temperature-control application, are loaded into the MATLAB® workspace. `u2` is the plant input, and `y2` is the plant output. The sampling period for `u2` and `y2` is 0.08 seconds.

Create an `iddata` object, `dry_data`, to store the measured values of the plant inputs and outputs, sampling time, channel names, etc.

```
Ts = 0.08;  
dry_data = iddata(y2,u2,Ts);
```

You can optionally assign channel names and units for the input/output signals. To do so, use the `InputName`, `OutputName`, `InputUnit` and `OutputUnit` properties of an `iddata` object. For example:

```
dry_data.InputName = 'Power';  
dry_data.OutputName = 'Temperature';
```

Detrend the measured data.

Before estimating the plant model, preprocess the measured data to increase the accuracy of the estimated model. For this example, `u2` and `y2` contain constant offsets that you eliminate.

```
dry_data_detrended = detrend(dry_data);
```

Estimate a linear plant model.

You can use System Identification Toolbox software to estimate a linear plant model in one of the following forms:

- State-space model
- Transfer function model
- Polynomial model
- Process model
- Grey-box model

For this example, estimate a third-order, linear state-space plant model using the detrended data.

```
plant = ssest(dry_data_detrended,3);
```

To view the estimated parameters and estimation details, at the MATLAB command prompt, type `plant`.

See Also

`detrend` | `iddata` | `ssest`

Related Examples

- “Identify Linear Models Using System Identification App”
- “Design Controller for Identified Plant” on page 2-54
- “Design Controller Using Identified Model with Noise Channel” on page 2-56

More About

- “Handling Offsets and Trends in Data”
- “Working with Impulse-Response Models” on page 2-58

Design Controller for Identified Plant

This example shows how to design a model predictive controller using an identified plant model. The internal plant model of the controller uses only the measured input and output of the identified model.

This example requires a System Identification Toolbox™ license.

Load the input/output data for identification.

```
load dryer2  
Ts = 0.08;
```

Create an `iddata` object from the input

```
dry_data = iddata(y2,u2,Ts);  
dry_data_detrended = detrend(dry_data);
```

Estimate a linear state-space plant model.

```
plant_idss = ssest(dry_data_detrended,3);
```

`plant_idss` is a third-order, identified state-space model that contains one measured input and one unmeasured noise component.

You can convert the identified model to an `ss`, `tf`, or `zpk` model. For this example, convert the identified state-space model to a numeric state-space model.

```
plant_ss = ss(plant_idss);
```

`plant_ss` contains the measured input and output of `plant_idss`. The software discards the noise component of `plant_idss` when it creates `plant_ss`.

Design a model predictive controller for the numeric plant model.

```
controller = mpc(plant_ss,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

`controller` is an `mpc` object in which:

- The measured input of `plant_ss` is a manipulated variable.
- The output of `plant_ss` is a measured output.

To view the structure of the model predictive controller, type `controller` at the MATLAB® command prompt.

See Also

`mpc`

Related Examples

- “Identify Plant from Data” on page 2-52
- “Design Controller Using Identified Model with Noise Channel” on page 2-56

More About

- “About Identified Linear Models”

Design Controller Using Identified Model with Noise Channel

This example shows how to design a model predictive controller using an identified plant model with a nontrivial noise component.

This example requires a System Identification Toolbox™ license.

Load the input/output data for identification.

```
load dryer2  
Ts = 0.08;
```

Create an `iddata` object from the input

```
dry_data = iddata(y2,u2,Ts);  
dry_data_detrended = detrend(dry_data);
```

Estimate a linear state-space plant model.

```
plant_idss = ssest(dry_data_detrended,3);
```

`plant_idss` is a third-order, identified state-space model that contains one measured input and one unmeasured noise component.

Design a model predictive controller for the identified plant model.

```
controller = mpc(plant_idss,Ts);  
  
-->Converting "idmodel" object to state-space.  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

`controller` is an `mpc` object in which:

- The measured input of `plant_idss` is a manipulated variable.
- The noise component of `plant_idss` is an unmeasured disturbance.
- The output of `plant_idss` is a measured output.

To view the structure of the model predictive controller, at the MATLAB® command prompt, type `controller`.

You can change the treatment of plant input signals in one of two ways:

- Programmatic - Use the `setmpcsignals` command to set the signal types.
- MPC Designer - Specify the signal types when defining the MPC structure using an imported plant.

You can also design a model predictive controller using:

```
plant_ss = ss(plant_idss, augmented );
controller2 = mpc(plant_ss,Ts);
```

```
-->All input signals are labeled as "Measured" or "Noise". The model was probably converted from a Simulink model. All "Measured" inputs will be treated as manipulated variables, all "Noise" inputs as disturbances. Type "help setmpcsignals" for more information.
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon=1.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default values.
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default values.
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default values.
```

When you use the `augmented` input argument, `ss` creates two input groups, `Measured` and `Noise`, for the measured and noise inputs of `plant_idss`. `mpc` handles the measured and noise components of `plant_ss` and `plant_idss` identically.

See Also

`mpc`

Related Examples

- “Identify Plant from Data” on page 2-52
- “Design Controller for Identified Plant” on page 2-54

More About

- “About Identified Linear Models”

Working with Impulse-Response Models

You can use System Identification Toolbox software to estimate finite step-response or finite impulse-response (FIR) plant models using measured data. Such models, also known as *nonparametric models* (see [1] for example), are easy to determine from plant data ([2] and [3]) and have intuitive appeal.

You use the `impulseest` function to estimate an FIR model from measured data. The function returns an identified transfer function model, `idtf`. To design a model predictive controller for the plant, you can convert the identified FIR plant model to a numeric LTI model. However, this conversion usually yields a high-order plant, which can degrade the controller design. This result is particularly an issue for MIMO systems. For example, the estimator design can be affected by numerical precision issues with high-order plants.

Model predictive controllers work best with low-order parametric models. See [4] for an example. Therefore, to design a model predictive controller using measured plant data, you can use a parametric estimator, such as `ssest`. Then estimate a low-order parametric plant model. Alternatively, you can initially identify a nonparametric model using the data and then estimate a low-order parametric model for the nonparametric model's response. See [5] for an example.

References

- [1] Prett, D., and C. Garcia, *Fundamental Process Control*, Butterworths, 1988.
- [2] Cutler, C., and F. Yocum, "Experience with the DMC inverse for identification," *Chemical Process Control — CPC IV*(Y. Arkun and W. H. Ray, eds.), CACHE, 1991.
- [3] Ricker, N. L., "The use of bias least-squares estimators for parameters in discrete-time pulse response models," *Ind. Eng. Chem. Res.*, Vol. 27, pp. 343, 1988.
- [4] Maciejowski, J. M., *Predictive Control with Constraints*, Pearson Education POD, 2002.
- [5] Wang, L., P. Gawthrop, C. Chessari, T. Podsiadly, and A. Giles, "Indirect approach to continuous time system identification of food extruder," *J. Process Control*, Vol. 14, Number 6, pp. 603–615, 2004.

See Also

`impulseest | ssest`

Related Examples

- “Identify Plant from Data” on page 2-52
- “Design Controller for Identified Plant” on page 2-54
- “Design Controller Using Identified Model with Noise Channel” on page 2-56

Bibliography

- [1] Allgower, F., and A. Zheng, *Nonlinear Model Predictive Control*, Springer-Verlag, 2000.
- [2] Camacho, E. F., and C. Bordons, *Model Predictive Control*, Springer-Verlag, 1999.
- [3] Kouvaritakis, B., and M. Cannon, *Non-Linear Predictive Control: Theory & Practice*, IEE Publishing, 2001.
- [4] Maciejowski, J. M., *Predictive Control with Constraints*, Pearson Education POD, 2002.
- [5] Prett, D., and C. Garcia, *Fundamental Process Control*, Butterworths, 1988.
- [6] Rossiter, J. A., *Model-Based Predictive Control: A Practical Approach*, CRC Press, 2003.
- [7] Wood, R. K., and M. W. Berry, *Chem. Eng. Sci.*, Vol. 28, pp. 1707, 1973.

Designing Controllers Using MPC Designer

- “Design Controller Using MPC Designer” on page 3-2
- “Test Controller Robustness” on page 3-23
- “Design MPC Controller for Plant with Delays” on page 3-35
- “Design MPC Controller for Nonsquare Plant” on page 3-43

Design Controller Using MPC Designer

This example shows how to design a model predictive controller for a continuous stirred-tank reactor (CSTR) using the MPC Designer app.

CSTR Model

The following differential equations represent the linearized model of a continuous stirred-tank reactor (CSTR) involving an exothermic reaction:

$$\frac{dC'_A}{dt} = a_{11}C'_A + a_{12}T' + b_{11}T'_c + b_{12}C'_{Ai}$$

$$\frac{dT'}{dt} = a_{21}C'_A + a_{22}T' + b_{21}T'_c + b_{22}C'_{Ai}$$

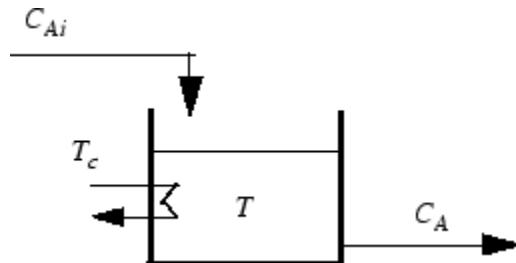
where the inputs are:

- C_{Ai} — Concentration of reagent A in the feed stream (kgmol/m^3)
- T_c — Reactor coolant temperature (degrees C)

and the outputs are:

- T — Reactor temperature (degrees C)
- C_A — Residual concentration of reagent A in the product stream (kgmol/m^3)

The prime terms, such as C'_A , denote a deviation from the nominal steady-state condition at which the model has been linearized.



Measurement of reagent concentrations is often difficult. For this example, assume that:

- T_c is a manipulated variable.
- C_{Ai} is an unmeasured disturbance.
- T is a measured output.
- C_A is an unmeasured output.

The model can be described in state-space format:

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

where,

$$x = \begin{bmatrix} C'_A \\ T' \end{bmatrix}, u = \begin{bmatrix} T'_c \\ C'_{Ai} \end{bmatrix}, y = \begin{bmatrix} T' \\ C'_A \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

For this example, the coolant temperature has a limited range of ± 10 degrees from its nominal value and a limited rate of change of ± 4 degrees per sample period.

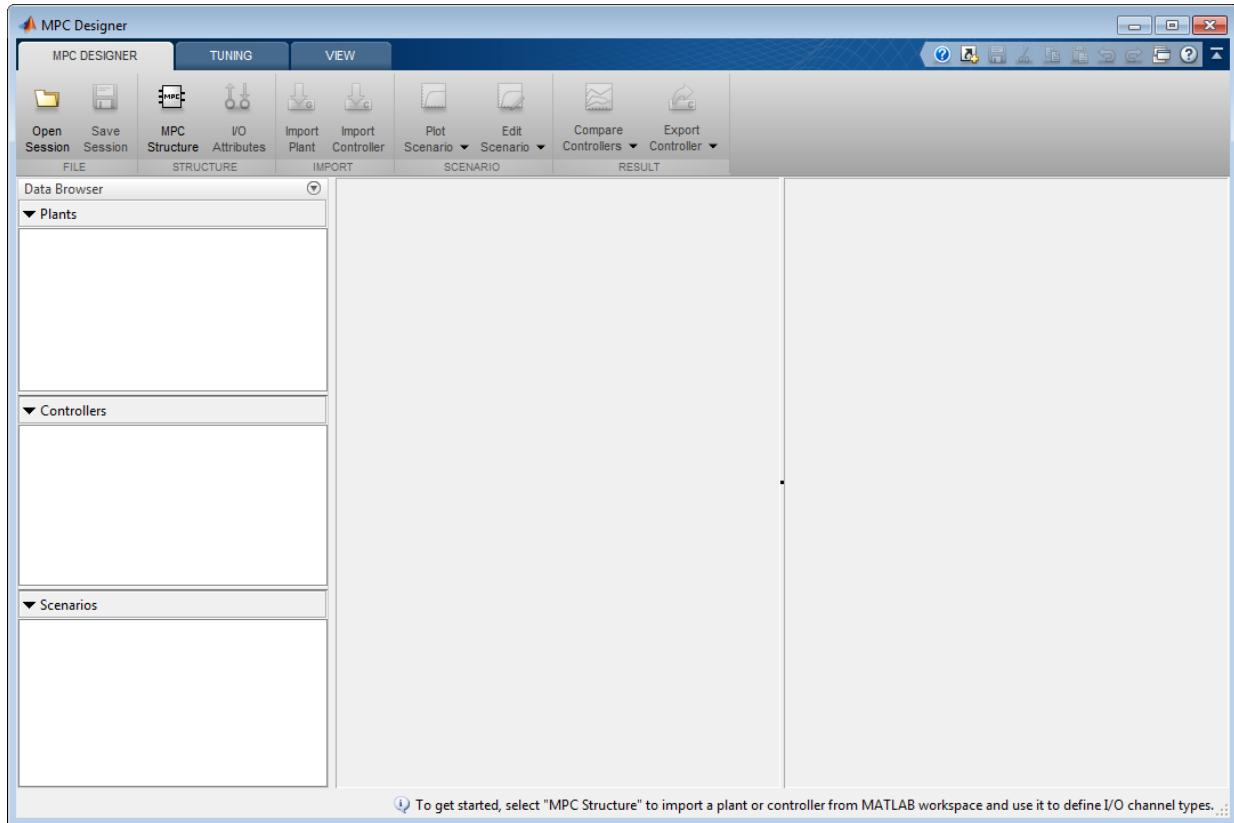
Define Plant Model

Create a state-space model of a CSTR system.

```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);
```

Open MPC Designer App

```
mpcDesigner
```



Import Plant and Define MPC Structure

On the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

In the Define MPC Structure By Importing dialog box, in the **Select a plant model or an MPC controller** table, select the **CSTR** model.

By default, all plant inputs are defined as manipulated variables and all plant outputs as measured outputs. In the **Assign plant i/o channels** section, assign the input and output channel indices such that:

- The first input, coolant temperature, is a manipulated variable.
- The second input, feed concentration, is an unmeasured disturbance.
- The first output, reactor temperature, is a measured output.
- The second output, reactant concentration, is an unmeasured output.

Define MPC Structure By Importing

MPC Structure

Select a plant model or an MPC controller from MATLAB Workspace:

Select	Name	Type	Order	Inputs	Outputs
<input checked="" type="radio"/>	CSTR	ss	2	2	2

Controller Sample Time

Specify MPC controller sample time:

Assign plant i/o channels to desired signal types:

Manipulated variable (MV) channel indices:	<input type="text" value="1"/>
Measured disturbance (MD) channel indices:	<input type="text"/>
Unmeasured disturbance (UD) channel indices:	<input type="text" value="2"/>
Measured output (MO) channel indices:	<input type="text" value="1"/>
Unmeasured output (UO) channel indices:	<input type="text" value="2"/>

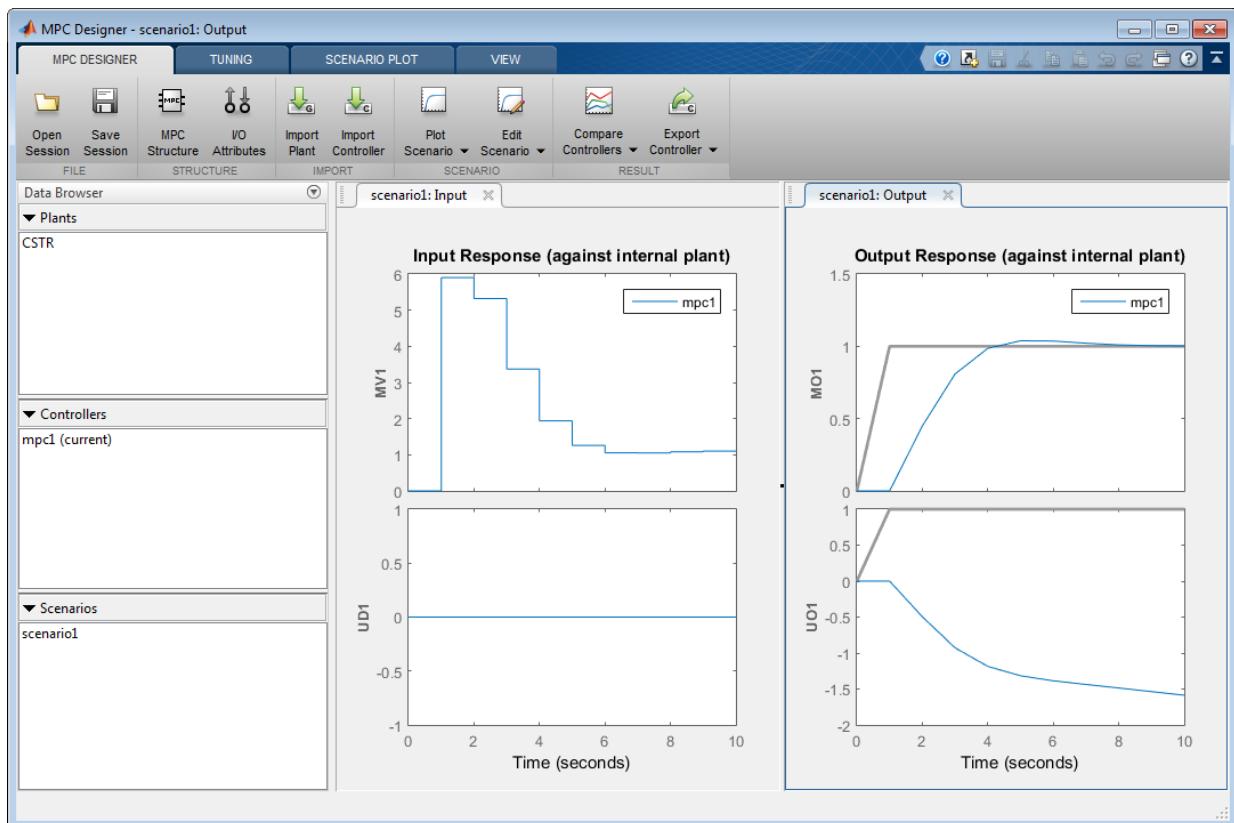
Refresh workspace Define and Import Cancel Help

Click **Define and Import**.

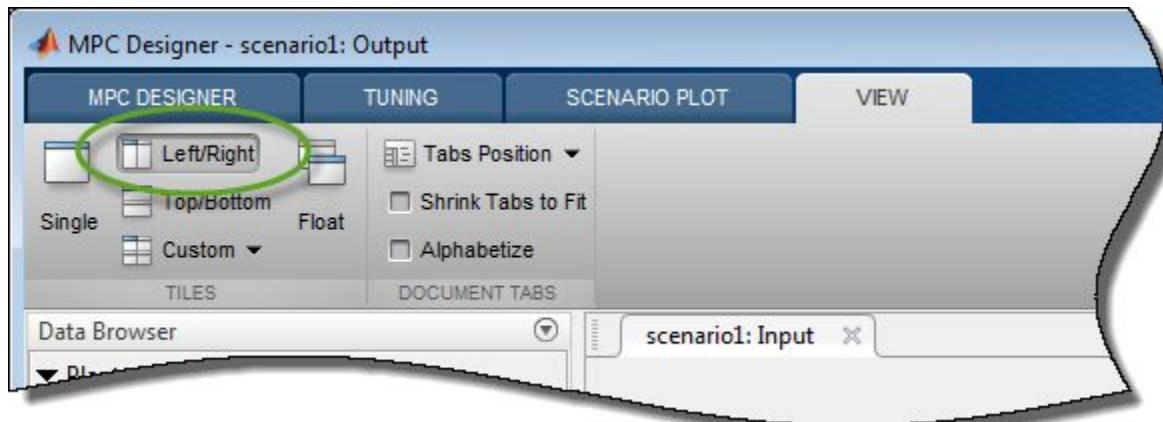
The app imports the CSTR plant to the **Data Browser**. The following are also added to the **Data Browser**:

- **mpc1** — Default MPC controller created using **sys** as its internal model. Since the CSTR plant is a continuous-time model, the default sample time of the controller is 1 second.
- **scenario1** — Default simulation scenario.

The app runs the default simulation scenario and updates the **Input Response** and **Output Response** plots.



Tip To view the response plots side-by-side, on the **View** tab, in the **Tiles** section, click **Left/Right**.



Once you define the MPC structure, you cannot change it within the current MPC Designer session. To use a different channel configuration, start a new session of the app.

Define Input and Output Channel Attributes

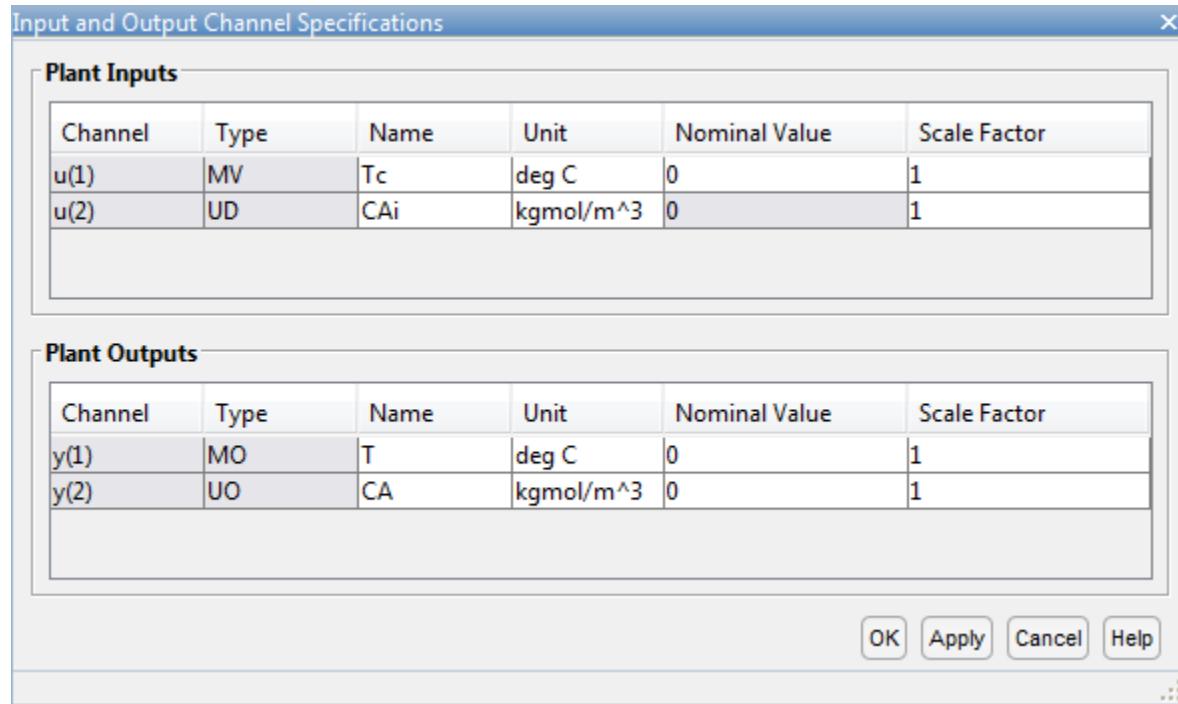
On the **MPC Designer** tab, select **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, in the **Name** column, specify a meaningful name for each input and output channel.

In the **Unit** column, optionally specify the units for each channel.

Since the state-space model is defined using deviations from the nominal operating point, set the **Nominal Value** for each input and output channel to 0.

Keep the **Scale Factor** for each channel at the default value of 1.



Click **OK**.

The **Input Response** and **Output Response** plot labels update to reflect the new signal names and units.

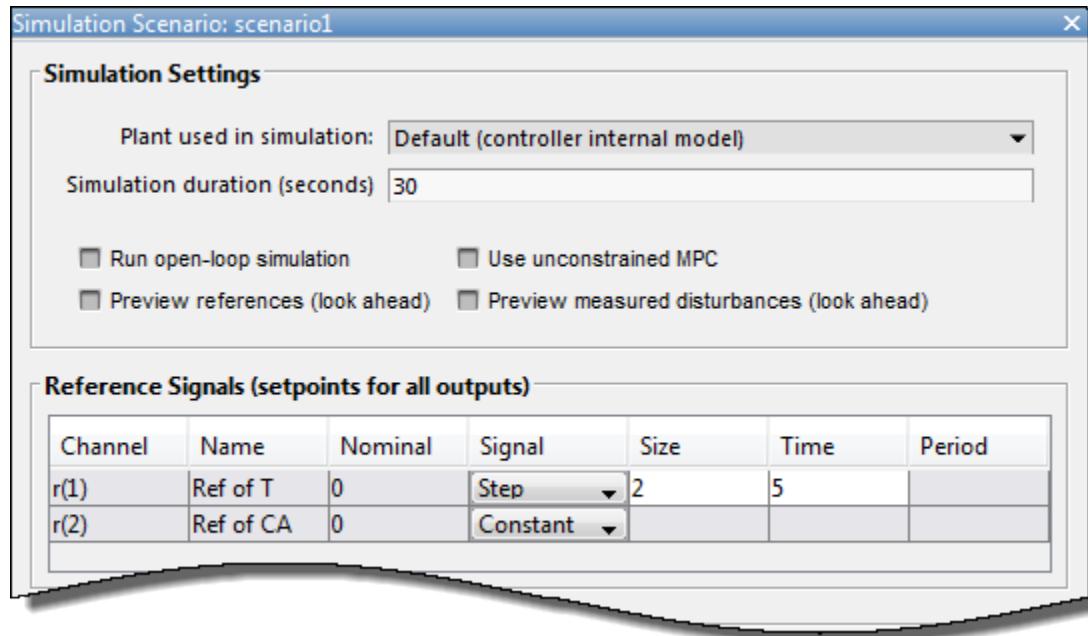
Configure Step Response Simulation Scenario

On the **MPC Designer** tab, in the **Scenario** Section, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, increase the **Simulation duration** to 30 seconds.

In the **Reference Signals** table, in the first row, specify a step **Size** of 2.

In the **Signal** column, in the second row, select a **Constant** reference to hold the concentration setpoint at its nominal value.

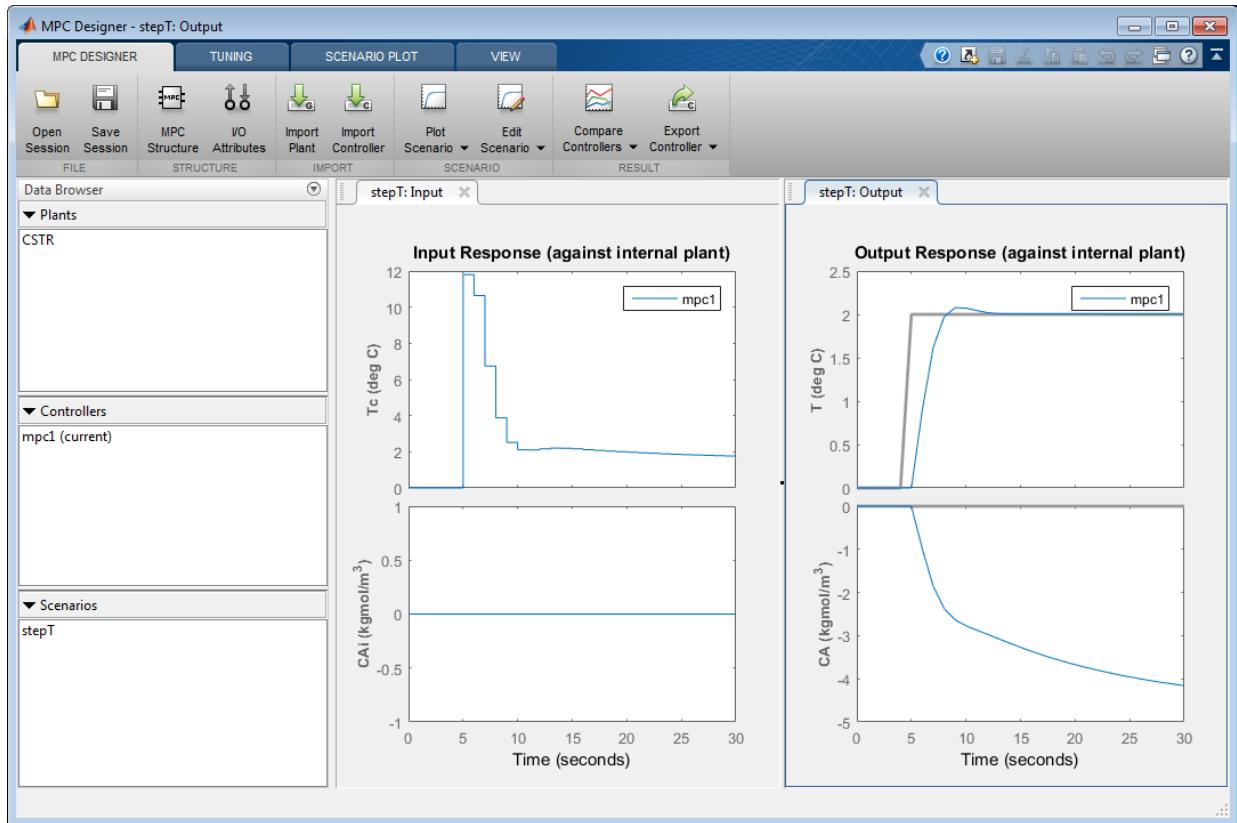


The default scenario is configured to simulate a step change of 2 degrees in the reactor temperature, T at a time of 5 seconds.

Click **OK**.

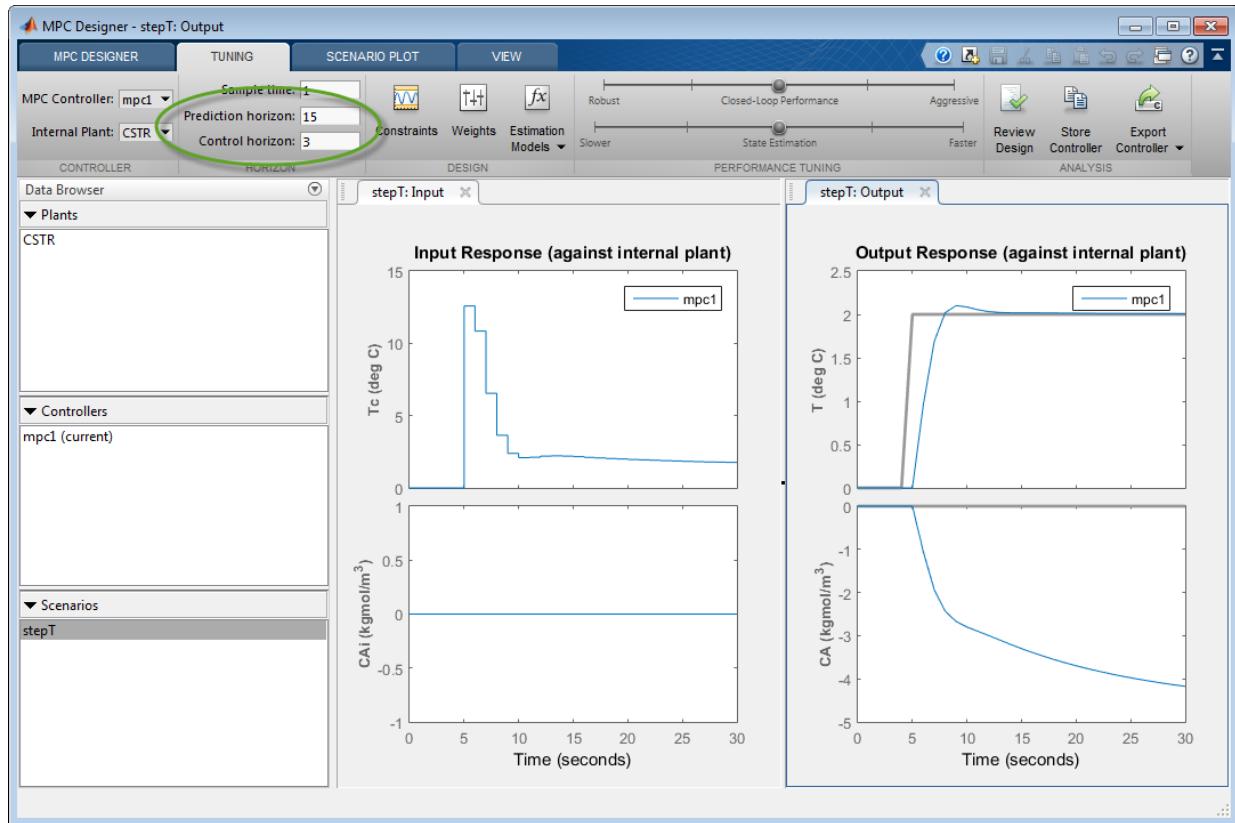
The response plots update to reflect the new simulation scenario configuration.

In the **Data Browser**, in the **Scenarios** section, double-click **scenario1**, and rename the scenario as **stepT**.



Configure Controller Horizons

On the **Tuning** tab, in the **Horizons** section, specify a **Prediction horizon** of 15 and a **Control horizon** of 3.



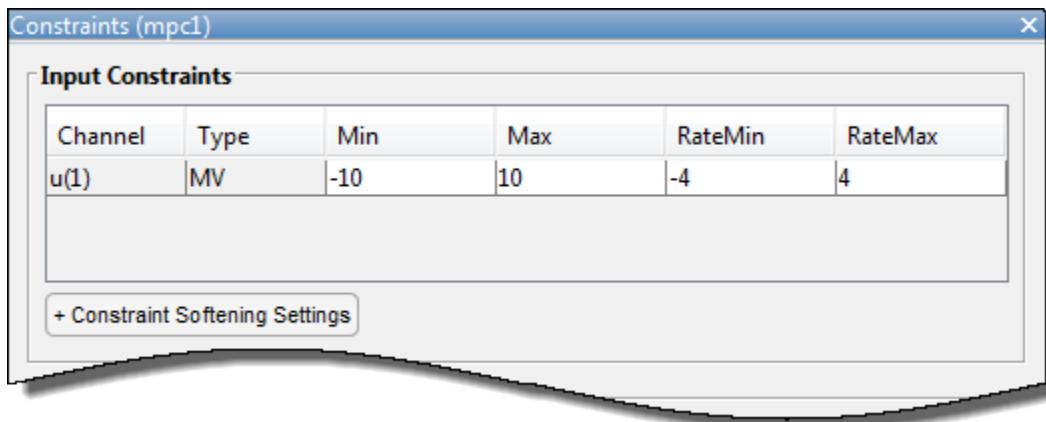
The response plots update to reflect the new horizons. The **Input Response** plot shows that the control actions for the manipulated variable violate the required coolant temperature constraints.

Define Manipulated Variable Input Constraints

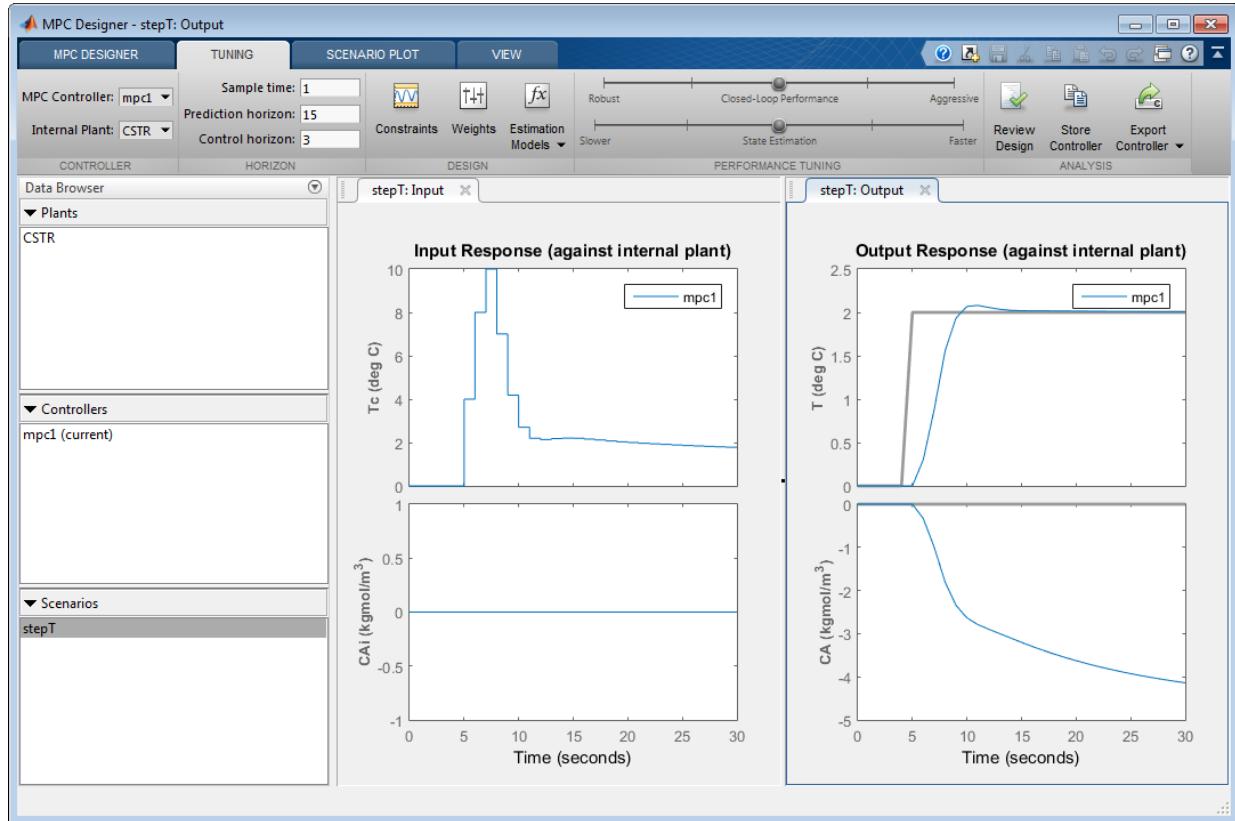
In the **Design** section, click **Constraints**.

In the Constraints dialog box, in the **Input Constraints** section, enter the coolant temperature upper and lower bounds in the **Min** and **Max** columns respectively.

Specify the rate of change limits in the **RateMin** and **RateMax** columns.



Click **OK**.



The **Input Response** plot shows the constrained manipulated variable control actions. Even with the constrained rate of change, the coolant temperature rises quickly to its maximum limit within three control intervals.

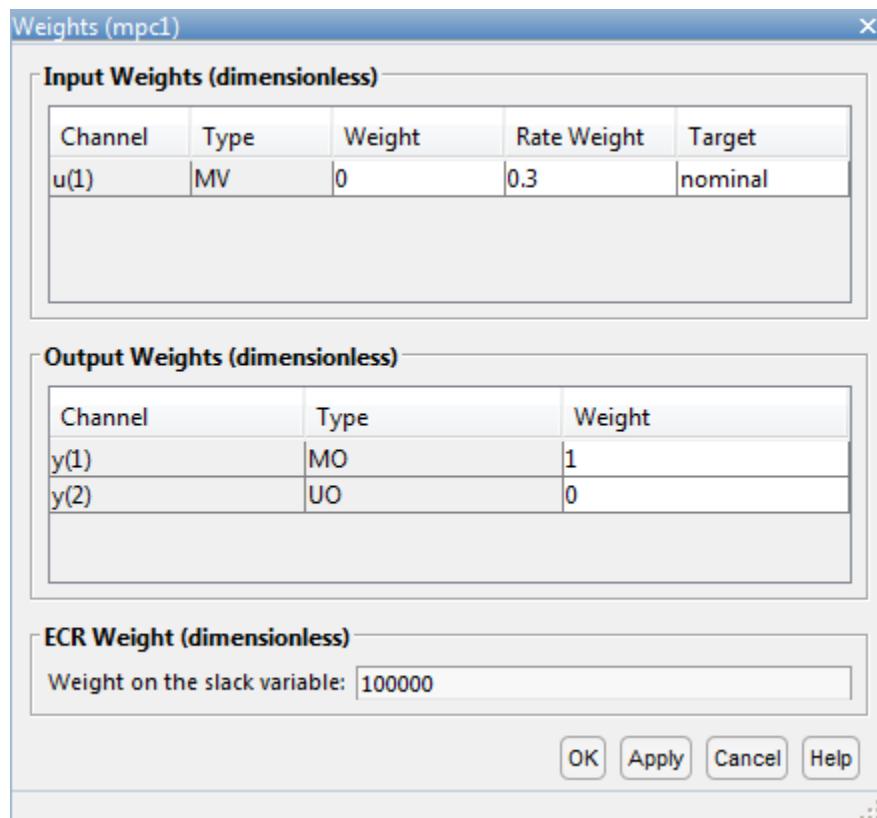
Specify Controller Tuning Weights

On the **Tuning** tab, in the **Design** section, click **Weights**.

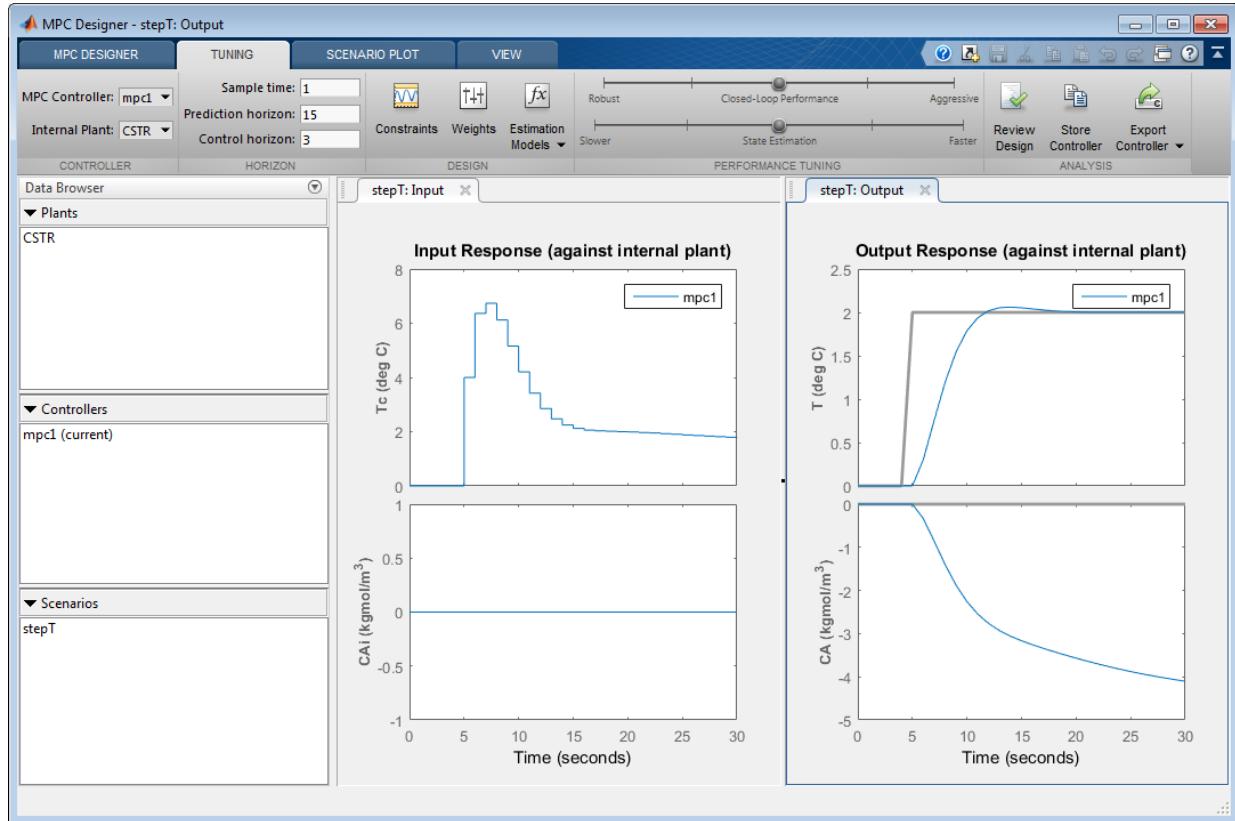
In the **Input Weights** table, increase the manipulated variable (MV)**Rate Weight** to **0.3**. Increasing the MV rate weight penalizes large MV changes in the controller optimization cost function.

In the **Output Weights** table, keep the default **Weight** values. By default, all unmeasured outputs have zero weights.

Since there is only one manipulated variable, if the controller tries to hold both outputs at specific setpoints, one or both outputs will exhibit steady-state error in their responses. Since the controller ignores setpoints for outputs with zero weight, setting the concentration output weight to zero allows reactor temperature setpoint tracking with zero steady-state error.



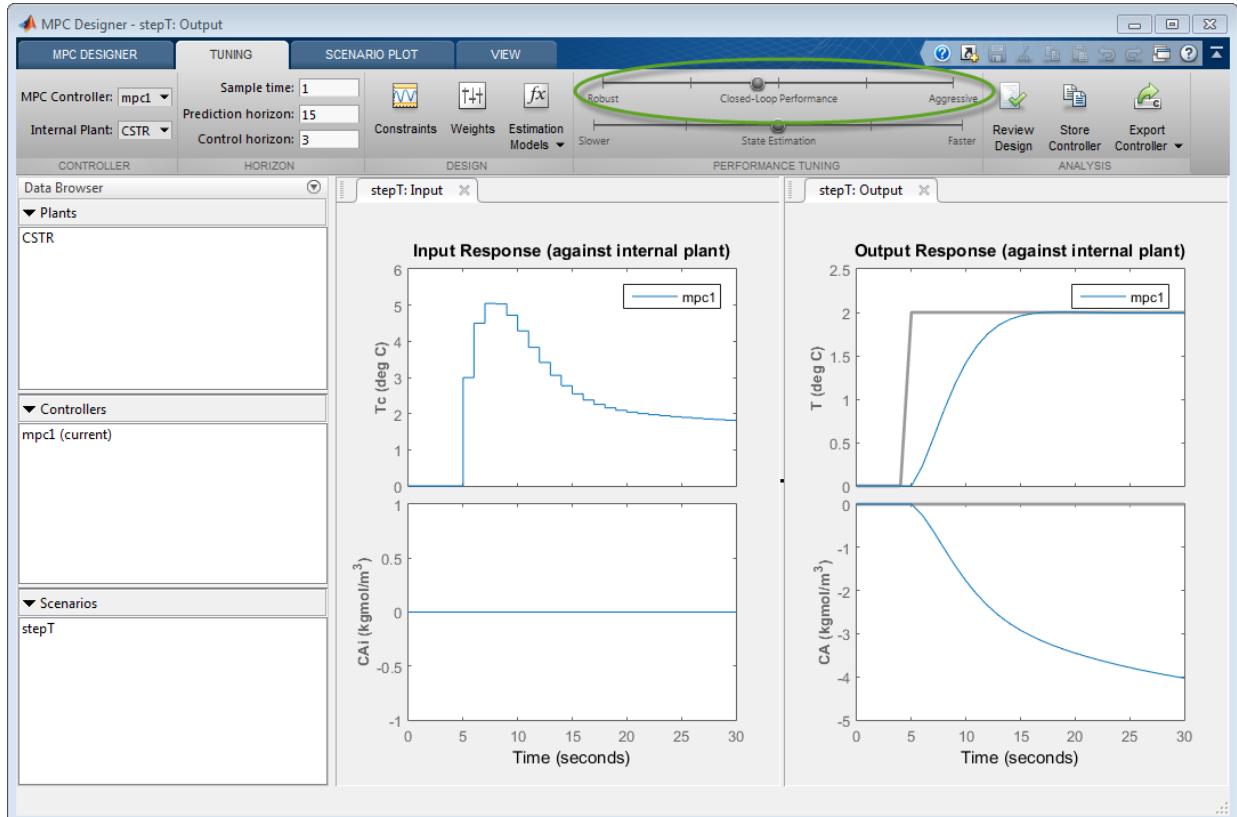
Click **OK**.



The **Input Response** plot shows the more conservative control actions, which result in a slower **Output Response**.

Eliminate Output Overshoot

Suppose the application demands zero overshoot in the output response. On the **Performance Tuning** tab, drag the **Closed-Loop Performance** slider to the left until the **Output Response** has no overshoot. Moving this slider to the left simultaneously reduces the manipulated variable rate weight and increases the output variable weight, producing a more robust controller.



Test Controller Disturbance Rejection

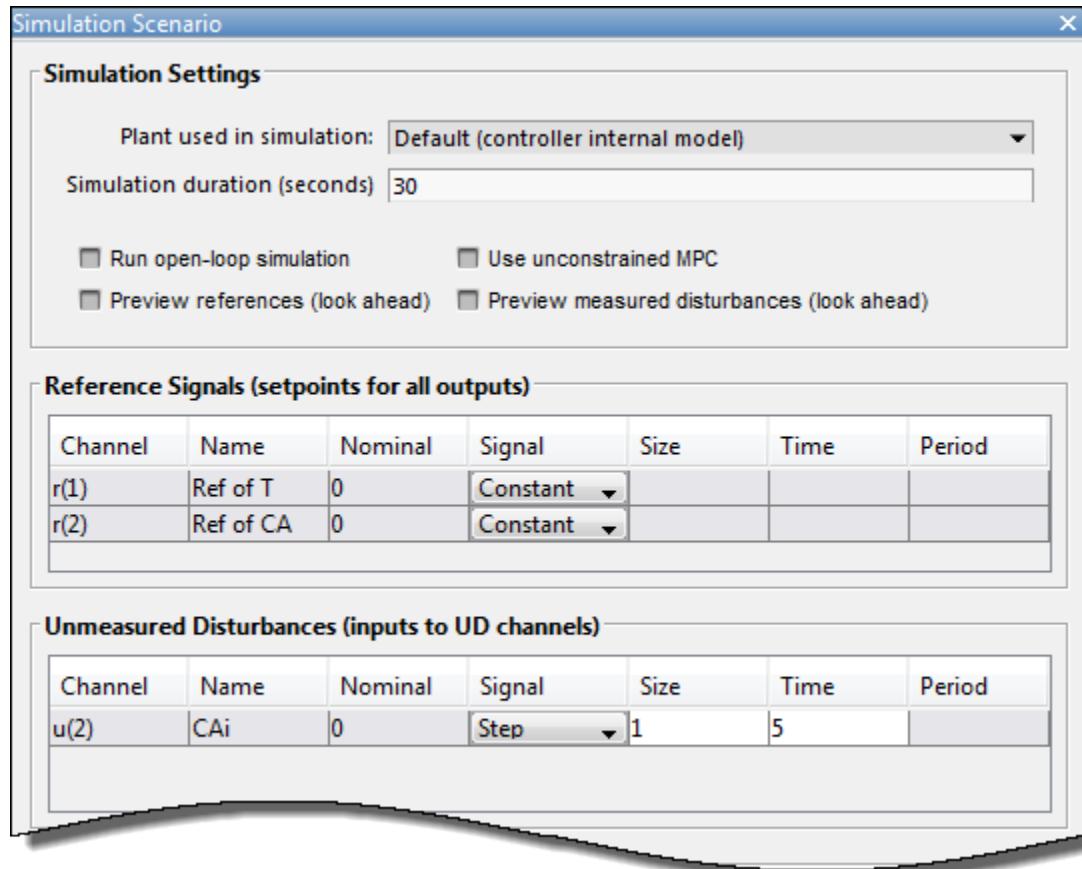
In a process control application, disturbance rejection is often more important than setpoint tracking. Simulate the controller response to a step change in the feed concentration.

On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > New Scenario**.

In the Simulation Scenario dialog box, set the **Simulation duration** to 30 seconds.

In the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step**.

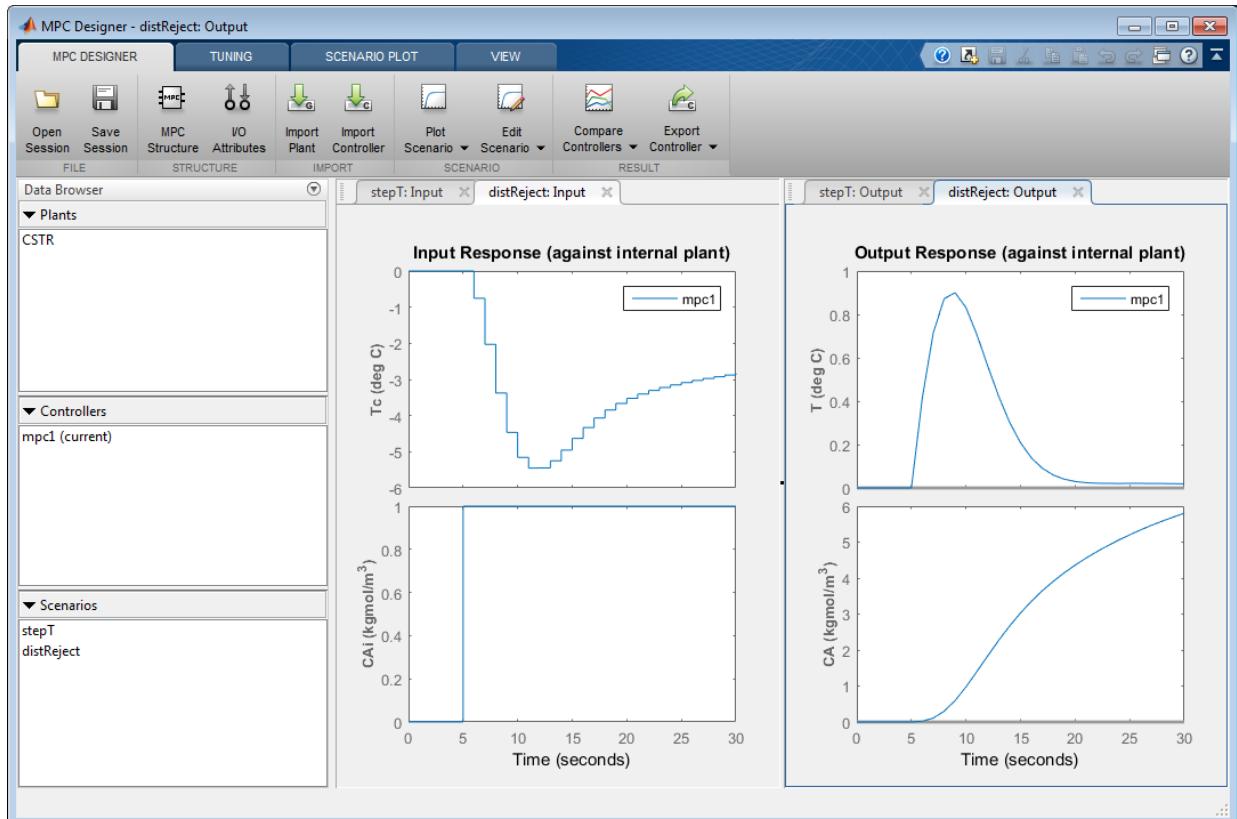
In the **Time** column, specify a step time of **5** seconds.



Click **OK**.

The app adds new scenario to the **Data Browser** and creates new corresponding **Input Response** and **Output Response** plots.

In the **Data Browser**, in the **Scenarios** section, double-click **NewScenario**, and rename it **distReject**.



In the **Output Response** plots, the controller returns the reactor temperature, T , to a value near its setpoint as expected. However, the required control actions cause an increase in the output concentration, CA to 6 kgmol/m^3 .

Specify Concentration Output Constraint

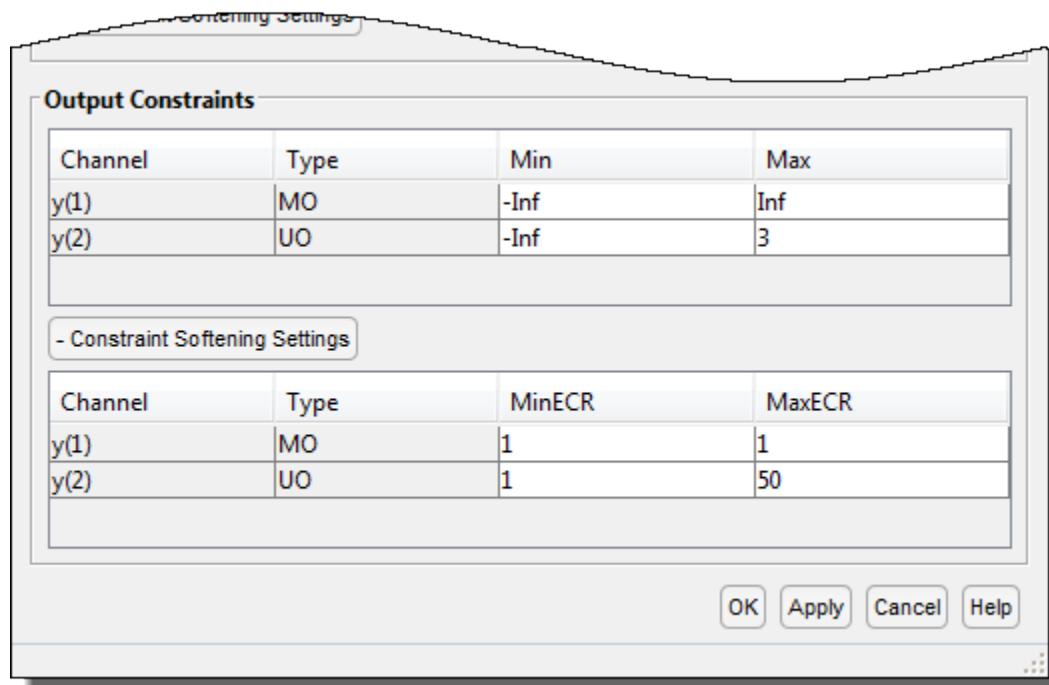
Previously, you defined the controller tuning weights to achieve the primary control objective of tracking the reactor temperature setpoint with zero steady-state error. Doing so enables the unmeasured reactor concentration to vary freely. Suppose that unwanted reactions occur once the reactor concentration exceeds a 3 kgmol/m^3 . To limit the reactor concentration, specify an output constraint.

On the **Tuning** tab, in the **Design** section, click **Constraints**.

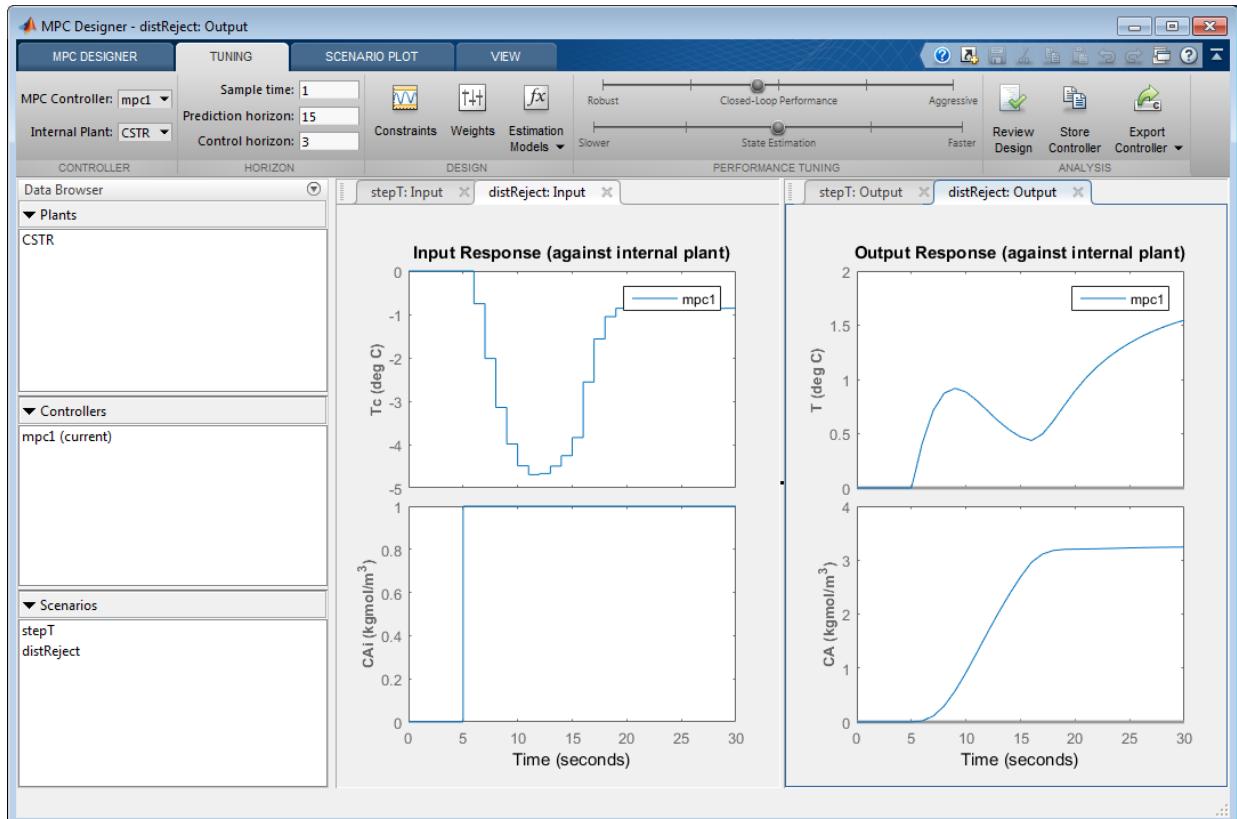
In the Constraints dialog box, in the **Output Constraints** section, the second row of the table, specify a **Max** unmeasured output (UO) value of 3.

In the **Output Constraints** section, click **Constraint Softening Settings**.

By default, all output constraints are soft, meaning that their **MinECR** and **MaxECR** values are greater than zero. To soften the unmeasured output (UO) constraint further, increase its **MaxECR** value.



Click **OK**.



In the **Output Response** plots, once the reactor concentration, **CA**, approaches 3 kgmol/m³, the reactor temperature, **T**, starts to increase. Since there is only one manipulated variable, the controller makes a compromise between the two competing control objectives: Temperature control and constraint satisfaction. A softer output constraint enables the controller to sacrifice the constraint requirement more to achieve improved temperature tracking.

Since the output constraint is soft, the controller maintains adequate temperature control by allowing a small concentration constraint violation. In general, depending on your application requirements, you can experiment with different constraint settings to achieve an acceptable control objective compromise.

Export Controller

In the **Tuning** tab, in the **Analysis** section, click **Export Controller**  to save the tuned controller, `mpc1`, to the MATLAB workspace.

References

- [1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004, pp. 34–36 and 94–95.

See Also

MPC Designer

Related Examples

- “Design MPC Controller in Simulink” on page 5-2

More About

- “Specifying Constraints”
- “Tuning Weights”

Test Controller Robustness

This example shows how to test the sensitivity of your model predictive controller to prediction errors using simulations.

It is good practice to test the robustness of your controller to prediction errors. Classical phase and gain margins are one way to quantify robustness for a SISO application. Robust Control Toolbox™ software provides sophisticated approaches for MIMO systems. It can also be helpful to run simulations.

Define Plant Model

For this example, use the CSTR model described in “Design Controller Using MPC Designer” on page 3-2.

```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);
```

Specify the signal names and signal types for the plant.

```
CSTR.InputName = { 'T_c' , 'C_A_i' };
CSTR.OutputName = { 'T' , 'C_A' };
CSTR.StateName = { 'C_A' , 'T' };
CSTR = setmpcsignals(CSTR, 'MV', 1, 'UD', 2, 'MO', 1, 'UO', 2);
```

Open MPC Designer App

Open the MPC Designer app, and import the plant model.

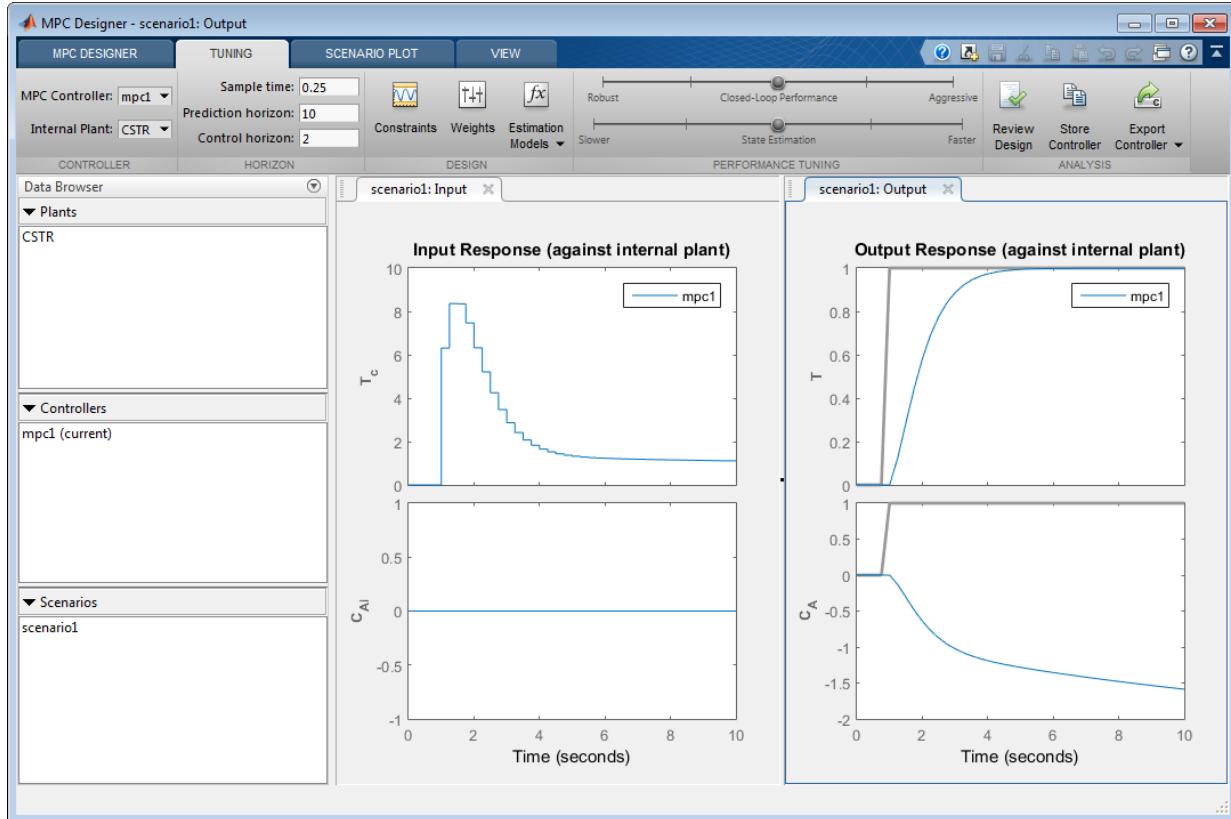
```
mpcDesigner(CSTR)
```

The app imports the plant model and adds it to the **Data Browser**. It also creates a default controller and a default simulation scenario.

Design Controller

Typically, you would design your controller by specifying scaling factors, defining constraints, and adjusting tuning weights. For this example, modify the controller sample time, and keep the other controller settings at their default values.

In the MPC Designer app, on the **Tuning** tab, in the **Horizon** section, specify a **Sample time** of **0.25** seconds.



The **Input Response** and **Output Response** plots update to reflect the new sample time.

Configure Simulation Scenario

To test controller setpoint tracking and unmeasured disturbance rejection, modify the default simulation scenario.

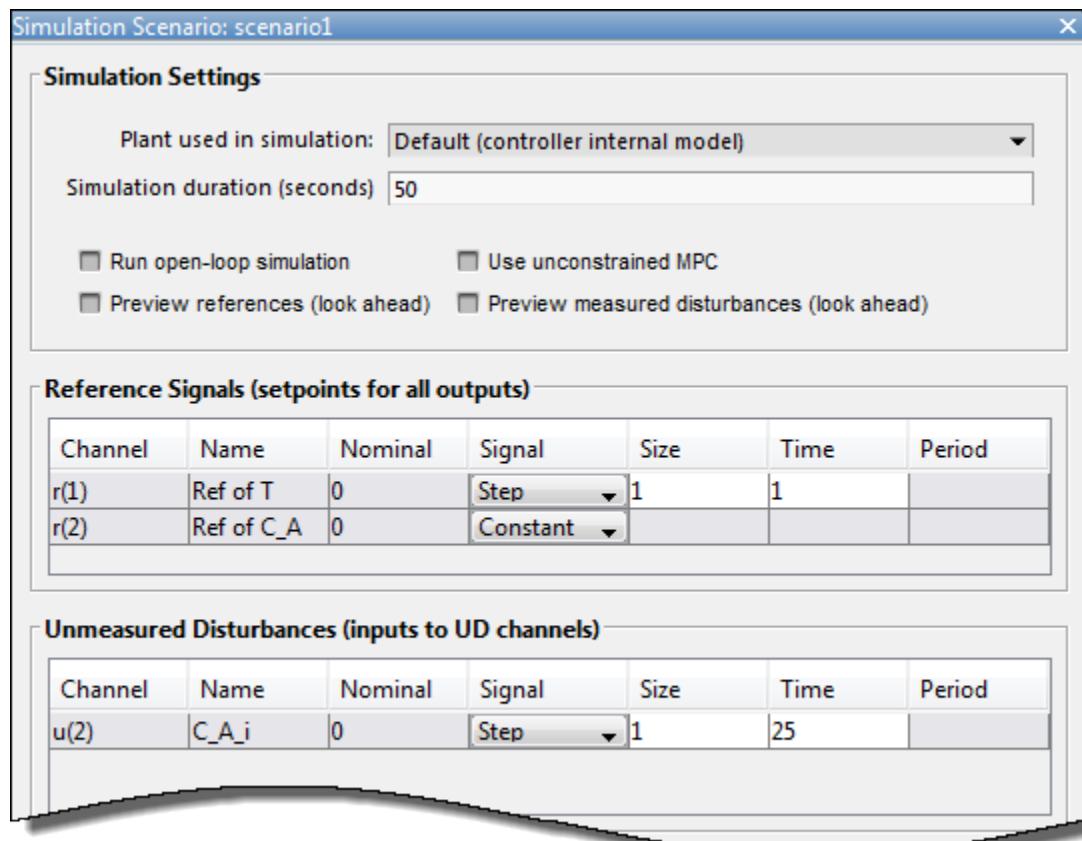
In the **Data Browser**, in the **Scenarios** sections, right-click **scenario1**, and select **Edit**.

In the Simulation Scenario dialog box, specify a **Simulation duration** of **50** seconds.

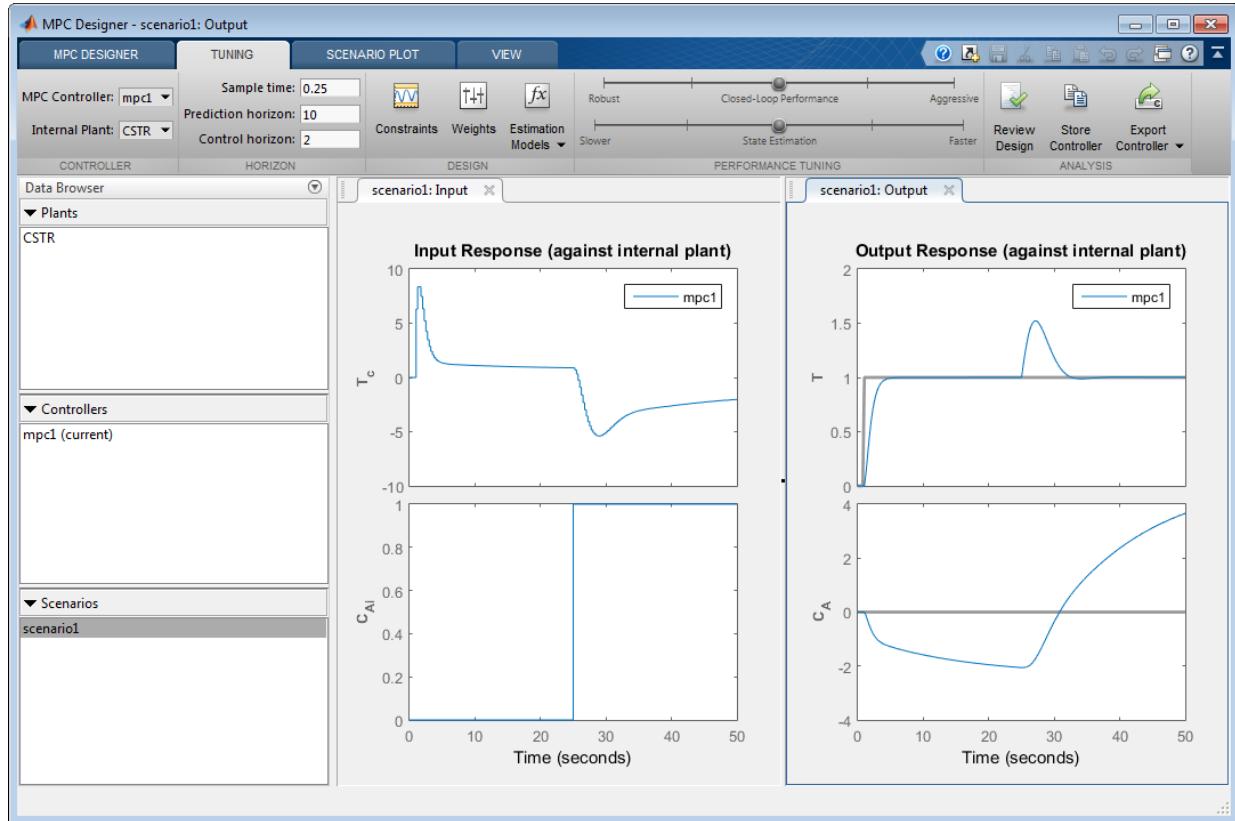
In the **Reference Signals** table, keep the default **Ref of T** setpoint configuration, which simulates a unit-step change in the reactor temperature.

To hold the concentration setpoint at its nominal value, in the second row, in the **Signal** drop-down list, select **Constant**.

Simulate a unit-step unmeasured disturbance at a time of 25 seconds. In the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step**, and specify a **Time** of 25.



Click **OK**.



The app runs the simulation scenario, and updates the response plots to reflect the new simulation settings. For this scenario, the internal model of the controller is used in the simulation. Therefore, the simulation results represent the controller performance when there are no prediction errors.

Define Perturbed Plant Models

Suppose that you want to test the sensitivity of your controller to plant changes that modify the effect of the coolant temperature on the reactor temperature. You can simulate such changes by perturbing element B(2,1) of the CSTR input-to-state matrix.

In the MATLAB Command Window, specify the perturbation matrix.

```
dB = [0 0;0.05 0];
```

Create the two perturbed plant models.

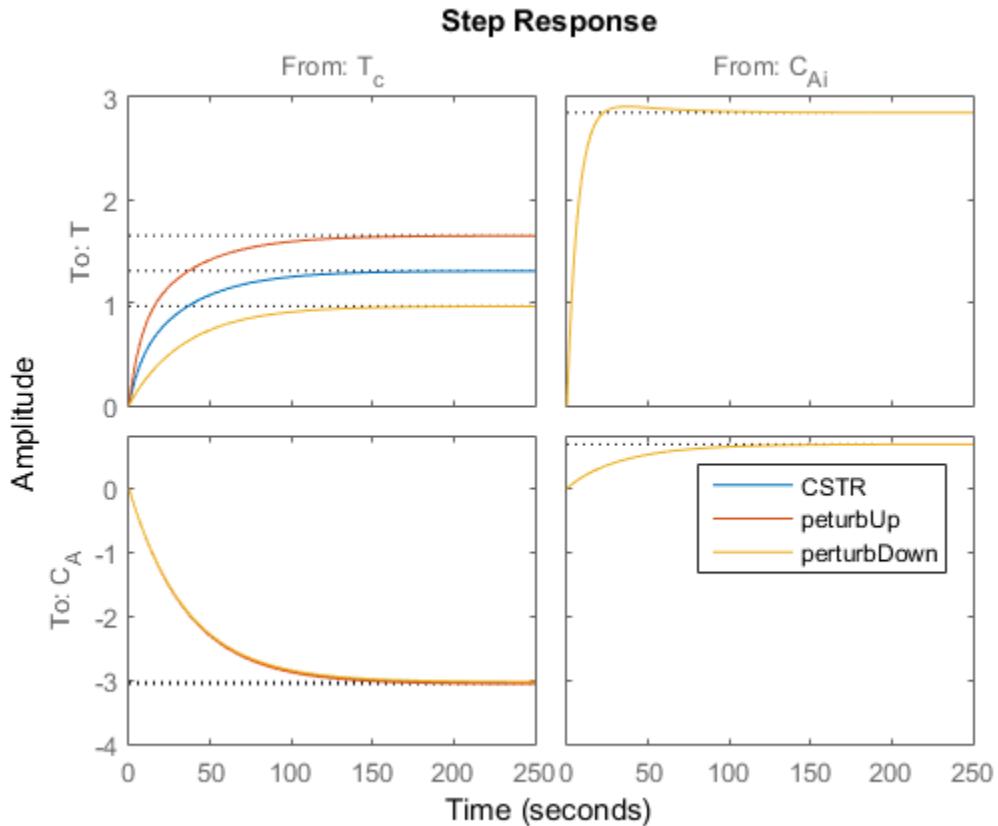
```
perturbUp = CSTR;
perturbUp.B = perturbUp.B + dB;

perturbDown = CSTR;
perturbDown.B = perturbDown.B - dB;
```

Examine Step Responses of Perturbed Plants

To examine the effects of the plant perturbations, plot the plant step responses.

```
step(CSTR,perturbUp,perturbDown)
legend( CSTR , perturbUp , perturbDown )
```

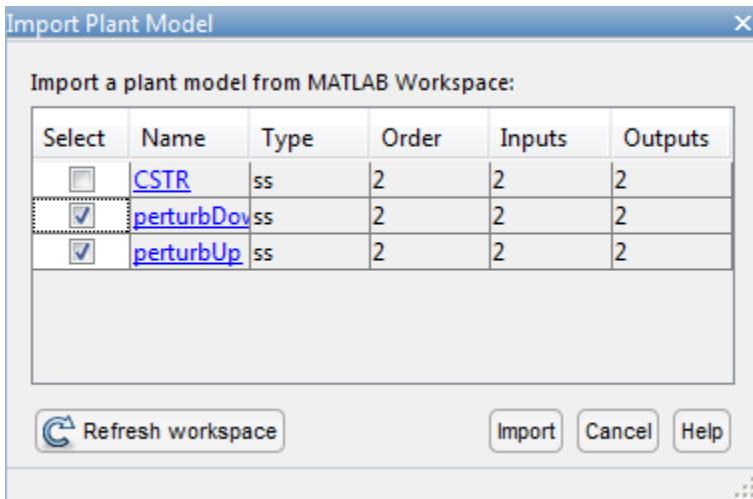


Perturbing element B(2, 1) of the CSTR plant changes the magnitude of the response of the reactor temperature, T , to changes in the coolant temperature, T_c .

Import Perturbed Plants

In the MPC Designer app, on the **MPC Designer** tab, in the **Import** section, click **Import Plant**.

In the Import Plant Model dialog box, select the **perturbUp** and **perturbDown** models.



Click **Import**.

The app imports the models and adds them to the **Data Browser**.

Define Perturbed Plant Simulation Scenarios

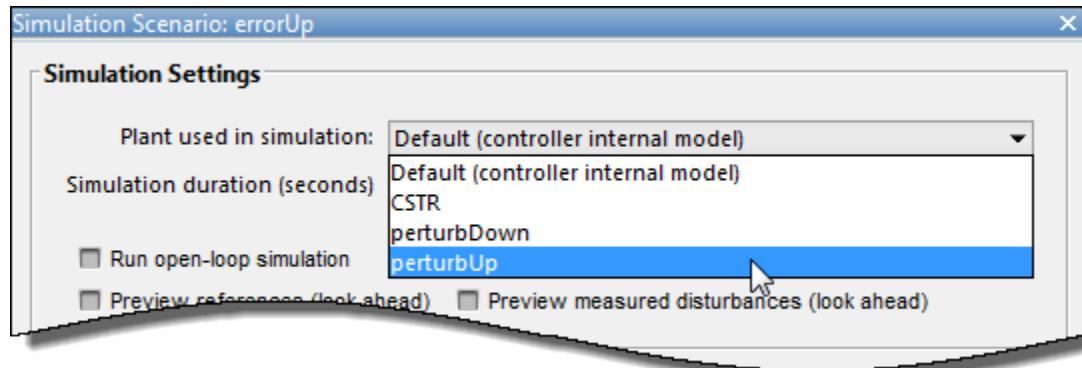
Create two simulation scenarios that use the perturbed plant models.

In the **Data Browser**, in the **Scenarios** section, double-click `scenario1`, and rename it `accurate`.

Right-click `accurate`, and click **Copy**. Rename `accurate_Copy` to `errorUp`.

Right-click `errorUp`, and select **Edit**.

In the Simulation Scenario dialog box, in the **Plant used in simulation** drop-down list, select `perturbUp`.

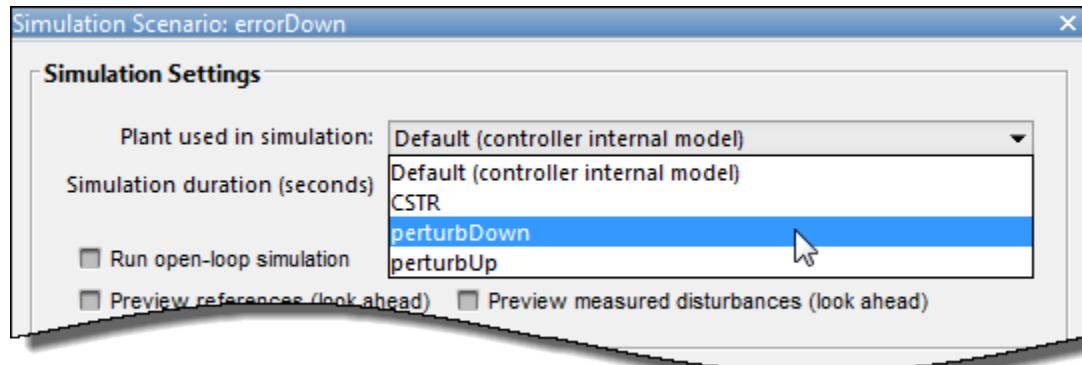


Click **OK**.

Repeat this process for the second perturbed plant.

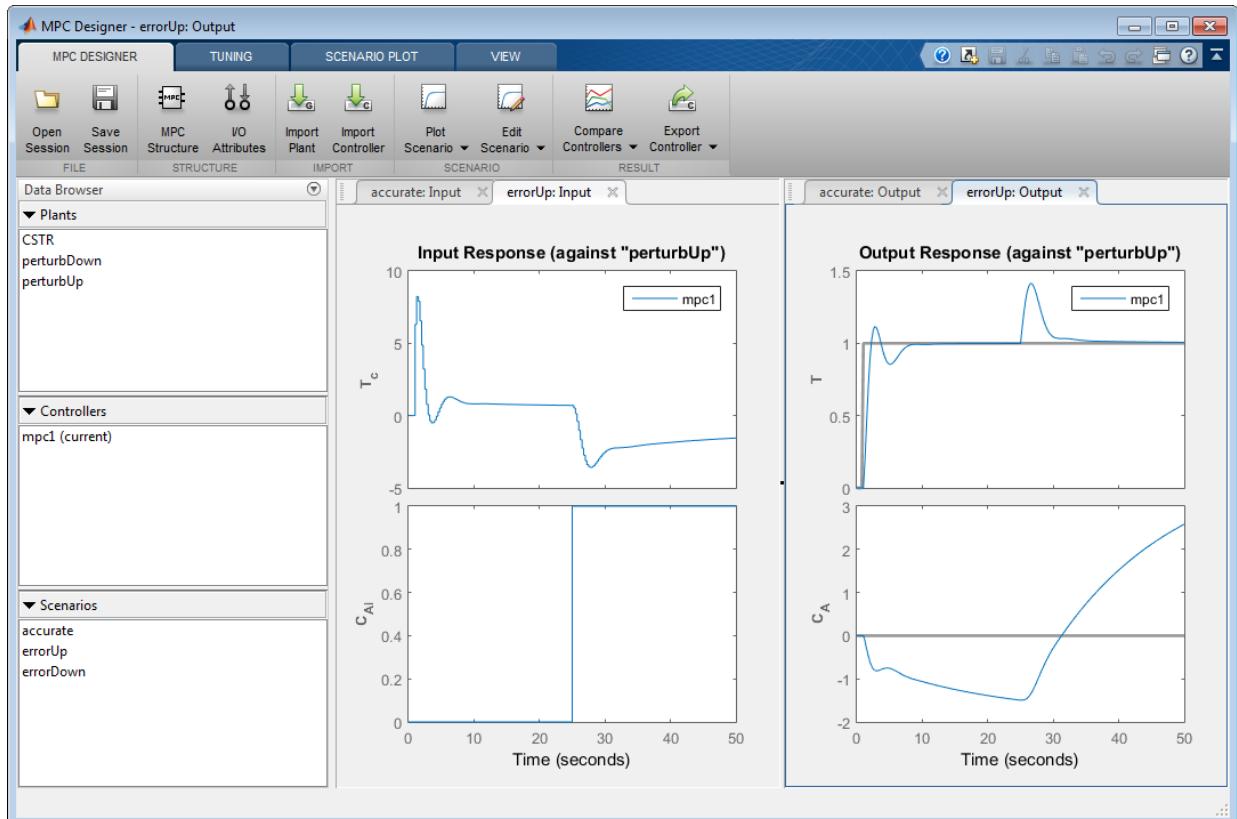
Copy the **accurate** scenario and rename it to **errorDown**.

Edit **errorDown**, selecting the **perturbDown** plant.



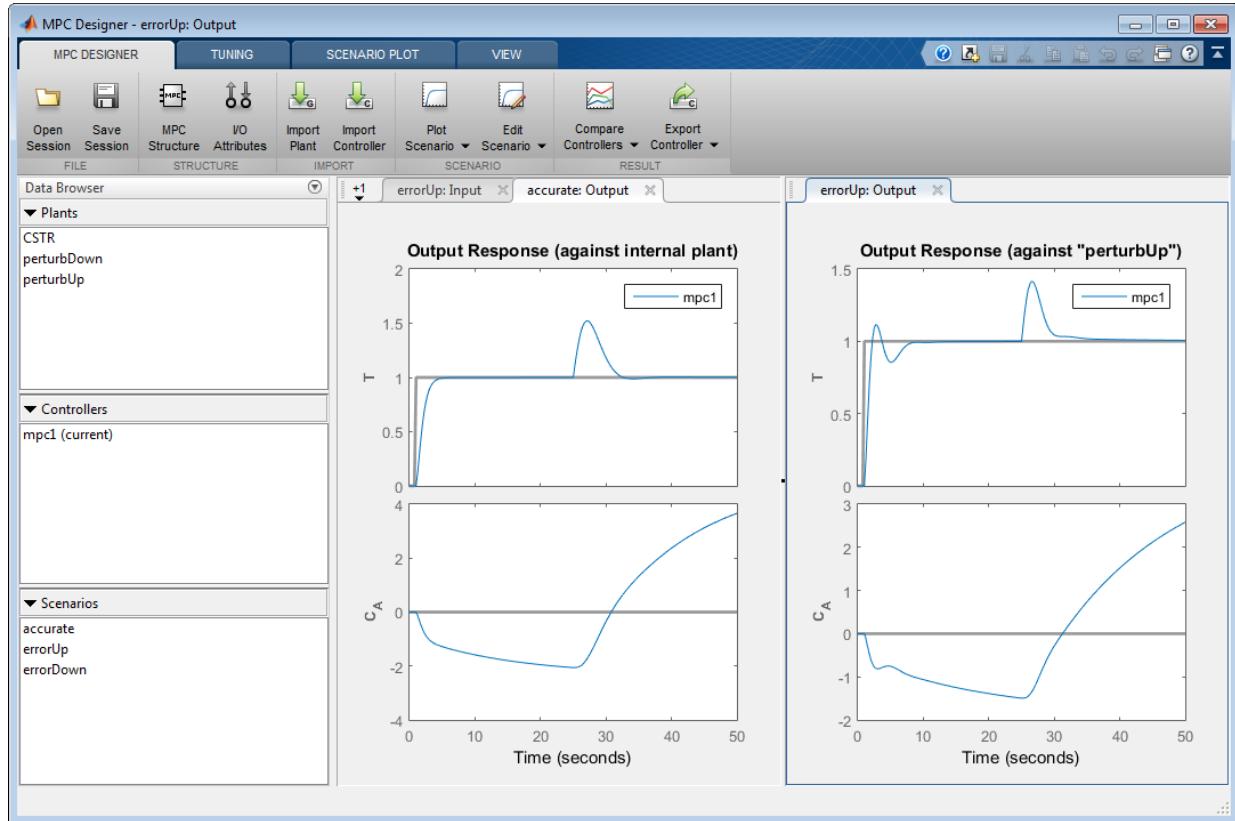
Examine **errorUp** Simulation Response

On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > errorUp**.



The app creates the **errorUp: Input** and **errorUp: Output** tabs, and displays the simulation response.

To view the **accurate** and **errorUp** responses side-by-side, drag the **accurate: Output** tab into the left plot panel.



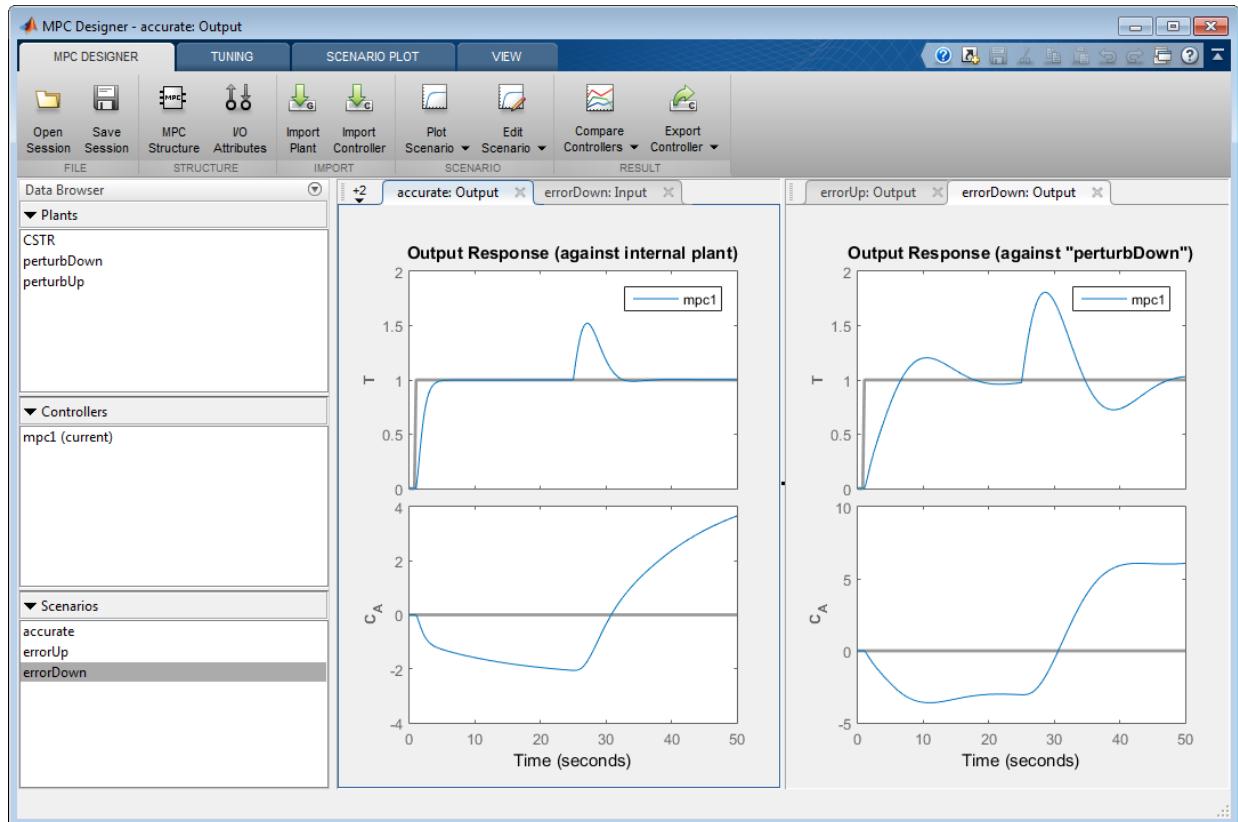
The perturbation creates a plant, **perturbUp**, that responds faster to manipulated variable changes than the controller predicts. On the **errorUp: Output** tab, in the **Output Response** plot, the T setpoint step response has about 10% overshoot with a longer settling time. Although this response is worse than the response of the **accurate** simulation, it is still acceptable. The faster plant response leads to a smaller peak error due to the unmeasured disturbance. Overall, the controller is able to control the **perturbUp** plant successfully despite the internal model prediction error.

Examine **errorDown** Simulation Response

On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > errorDown**.

The app creates the **errorDown: Input** and **errorDown: Output** tabs, and displays the simulation response.

To view the **accurate** and **errorDown** responses side-by-side, click the **accurate: Output** tab in the left display panel.



The perturbation creates a plant, **perturbDown**, that responds slower to manipulated variable changes than the controller predicts. On the **errorDown: Output** tab, in the **Output Response** plot, the setpoint tracking and disturbance rejection are worse than for the unperturbed plant.

Depending on the application requirements and the real-world potential for such plant changes, the degraded response for the `perturbDown` plant may require modifications to the controller design.

See Also

`mpc` | MPC Designer

Related Examples

- “Design Controller Using MPC Designer” on page 3-2
- “Test an Existing Controller” on page 5-23

Design MPC Controller for Plant with Delays

This example shows how to design an MPC controller for a plant with delays using the MPC Designer app.

Plant Model

An example of a plant with delays is the distillation column model:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{12.8e^{-s}}{16.7s+1} & \frac{-18.9e^{-3s}}{21.0s+1} & \frac{3.8e^{-8.1s}}{14.9s+1} \\ \frac{6.6e^{-7s}}{10.9s+1} & \frac{-19.4e^{-3s}}{14.4s+1} & \frac{4.9e^{-3.4s}}{13.2s+1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

Outputs y_1 and y_2 represent measured product purities. The model consists of six transfer functions, one for each input/output pair. Each transfer function is a first-order system with a delay. The longest delay in the model is 8.1 minutes.

Define Plant Model

Specify the individual transfer functions for each input/output pair. For example, g_{12} is the transfer function from input u_2 to output y_1 .

```
g11 = tf(12.8,[16.7 1], 'IOdelay', 1.0, 'TimeUnit', 'minutes');
g12 = tf(-18.9,[21.0 1], 'IOdelay', 3.0, 'TimeUnit', 'minutes');
g13 = tf(3.8,[14.9 1], 'IOdelay', 8.1, 'TimeUnit', 'minutes');
g21 = tf(6.6,[10.9 1], 'IOdelay', 7.0, 'TimeUnit', 'minutes');
g22 = tf(-19.4,[14.4 1], 'IOdelay', 3.0, 'TimeUnit', 'minutes');
g23 = tf(4.9,[13.2 1], 'IOdelay', 3.4, 'TimeUnit', 'minutes');
DC = [g11 g12 g13;
      g21 g22 g23];
```

Specify Input and Output Signal Names

```
DC.InputName = { 'Reflux Rate', 'Steam Rate', 'Feed Rate' };
DC.OutputName = { 'Distillate Purity', 'Bottoms Purity' };
```

Alternatively, you can specify the signal names in the MPC Designer app, on the **MPC Designer** tab, by clicking **I/O Attributes**.

Define Signal Configuration

Specify the third input, the feed rate, as a measured disturbance (MD).

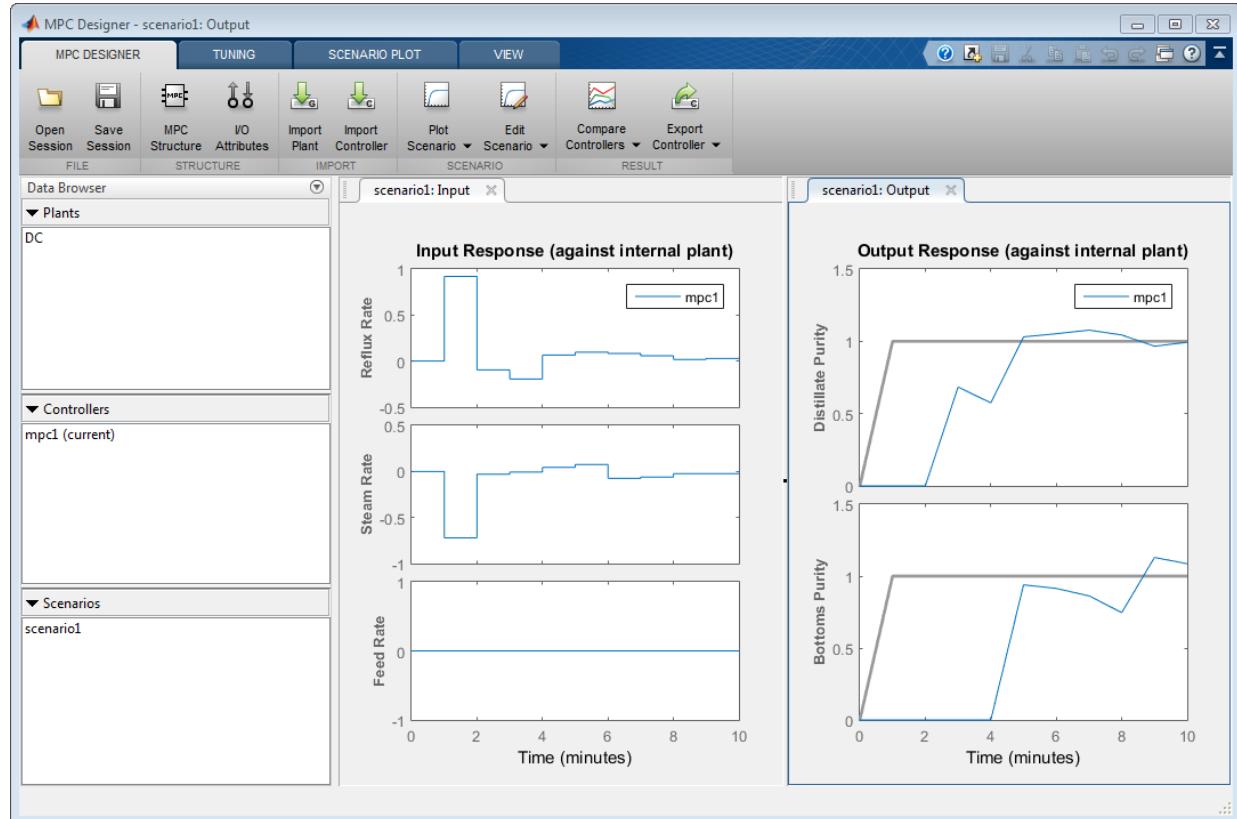
```
DC = setmpcsignals(DC, MD ,3);
```

Since they are not explicitly specified in `setmpcsignals`, all other input signals are configured as manipulated variables (MV), and all output signals are configured as measured outputs (MO) by default.

Open MPC Designer App

Open the MPC Designer app importing the plant model.

```
mpcDesigner(DC)
```



When launched with a continuous-time plant model, such as DC, the default controller sample time is 1 in the time units of the plant. If the plant is discrete time, the controller sample time is the same as the plant sample time.

The MPC Designer app imports the specified plant to the **Data Browser**. The following are also added to the **Data Browser**:

- `mpc1` — Default MPC controller created using DC as its internal model.
- `scenario1` — Default simulation scenario.

The app runs the simulation scenario and generates input and output response plots.

Specify Prediction and Control Horizons

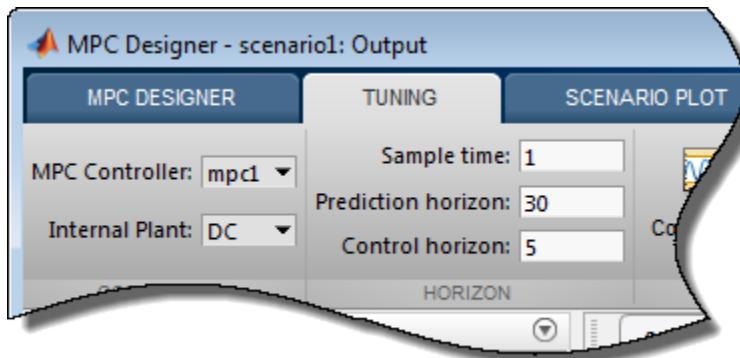
For a plant with delays, it is good practice to specify the prediction and control horizons such that

$$P - M \gg t_{d,max} / \Delta t$$

where,

- P is the prediction horizon.
- M is the control horizon.
- $t_{d,max}$ is the maximum delay, which is 8.1 minutes for the DC model.
- Δt is the controller **Sample time**, which is 1 minute by default.

On the **Tuning** tab, in the **Horizon** section, specify a **Prediction horizon** of 30 and a **Control horizon** of 5.



After you change the horizons, the **Input Response** and **Output Response** plots for the default simulation scenario are automatically updated.

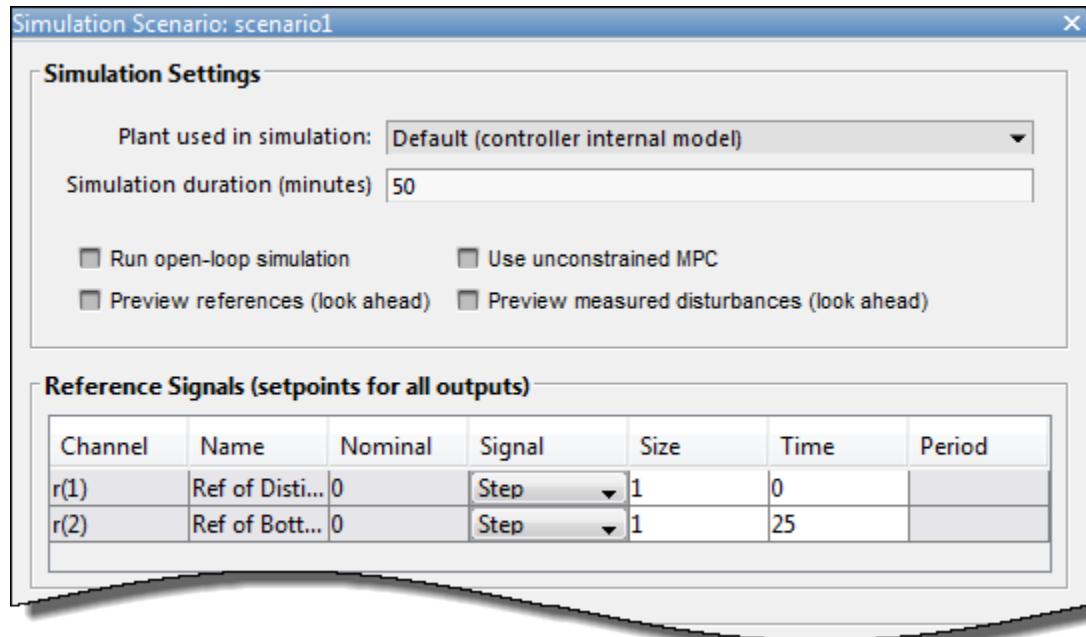
Simulate Controller Step Responses

On the **MPC Designer** tab, in the **Scenario** section, click **Edit Scenario > scenario1**. Alternatively, in the **Data Browser**, right-click **scenario1** and select **Edit**.

In the Simulation Scenario dialog box, specify a **Simulation duration** of 50 minutes.

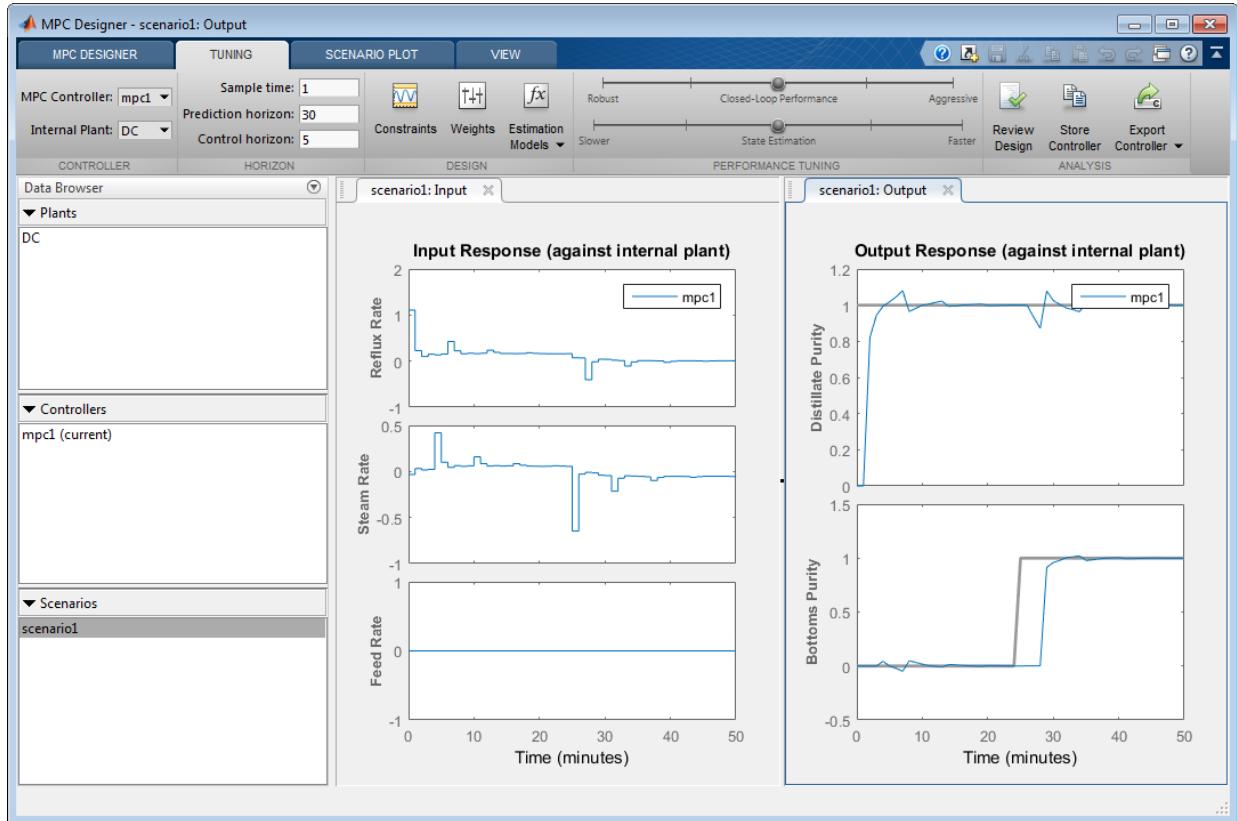
In the **Reference Signals** table, in the **Signal** drop-down list, select **Step** for both outputs to simulate step changes in their setpoints.

Specify a step **Time** of 0 for reference **r(1)**, the distillate purity, and a step time of 25 for **r(2)**, the bottoms purity.



Click **OK**.

The app runs the simulation with the new scenario settings and updates the input and output response plots.



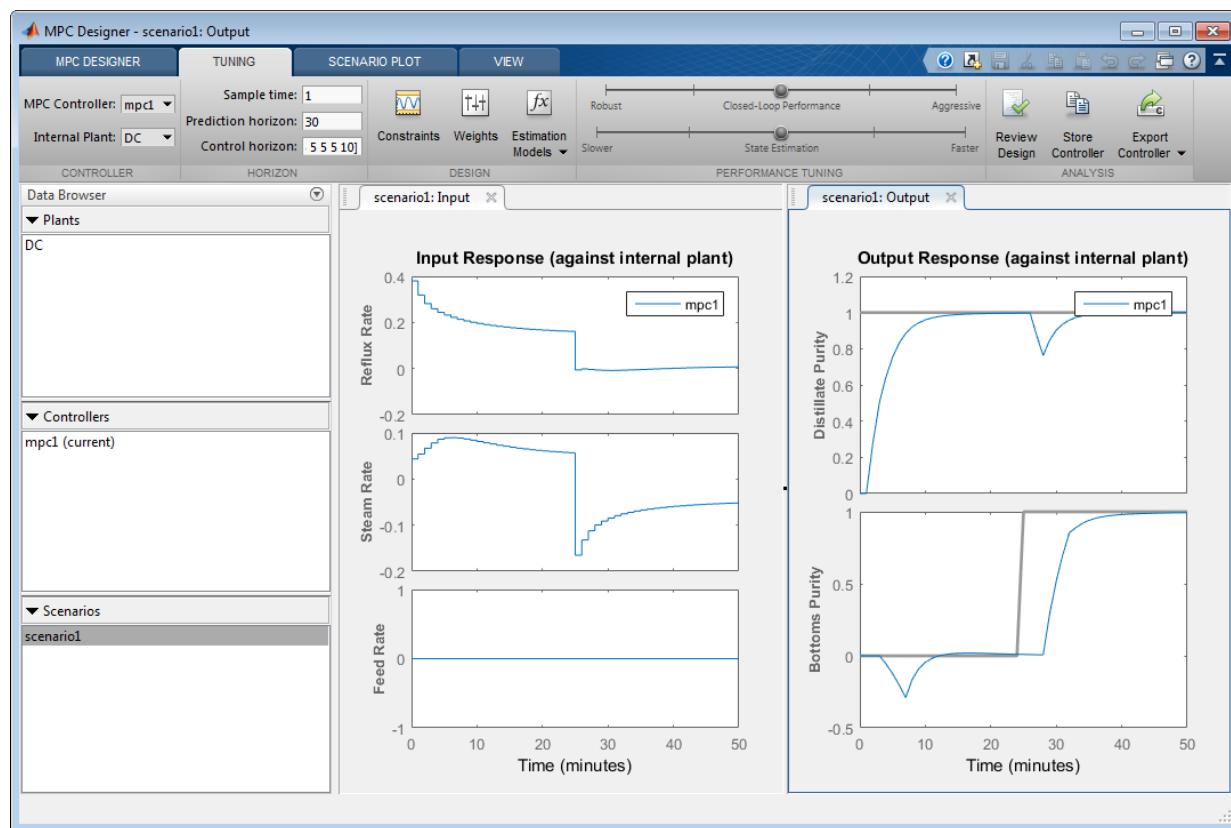
The **Input Response** plots show the optimal control moves generated by the controller. The controller reacts immediately in response to the setpoint changes, changing both manipulated variables. However, due to the plant delays, the effects of these changes are not immediately reflected in the **Output Response** plots. The **Distillate Purity** output responds after 1 minute, which corresponds to the minimum delay from g_{11} and g_{12} . Similarly, the **Bottoms Purity** output responds 3 minutes after the step change, which corresponds to the minimum delay from g_{21} and g_{22} . After the initial delays, both signals reach their setpoints and settle quickly. Changing either output setpoint disturbs the response of the other output. However, the magnitudes of these interactions are less than 10% of the step size.

Additionally, there are periodic pulses in the manipulated variable control actions as the controller attempts to counteract the delayed effects of each input on the two outputs.

Improve Performance Using Manipulated Variable Blocking

Use manipulated variable blocking to divide the prediction horizon into blocks, during which manipulated variable moves are constant. This technique produces smoother manipulated variable adjustments with less oscillation and smaller move sizes.

To use manipulated variable blocking, on the **Tuning** tab, specify the **Control horizon** as a vector of block sizes, [5 5 5 5 10].



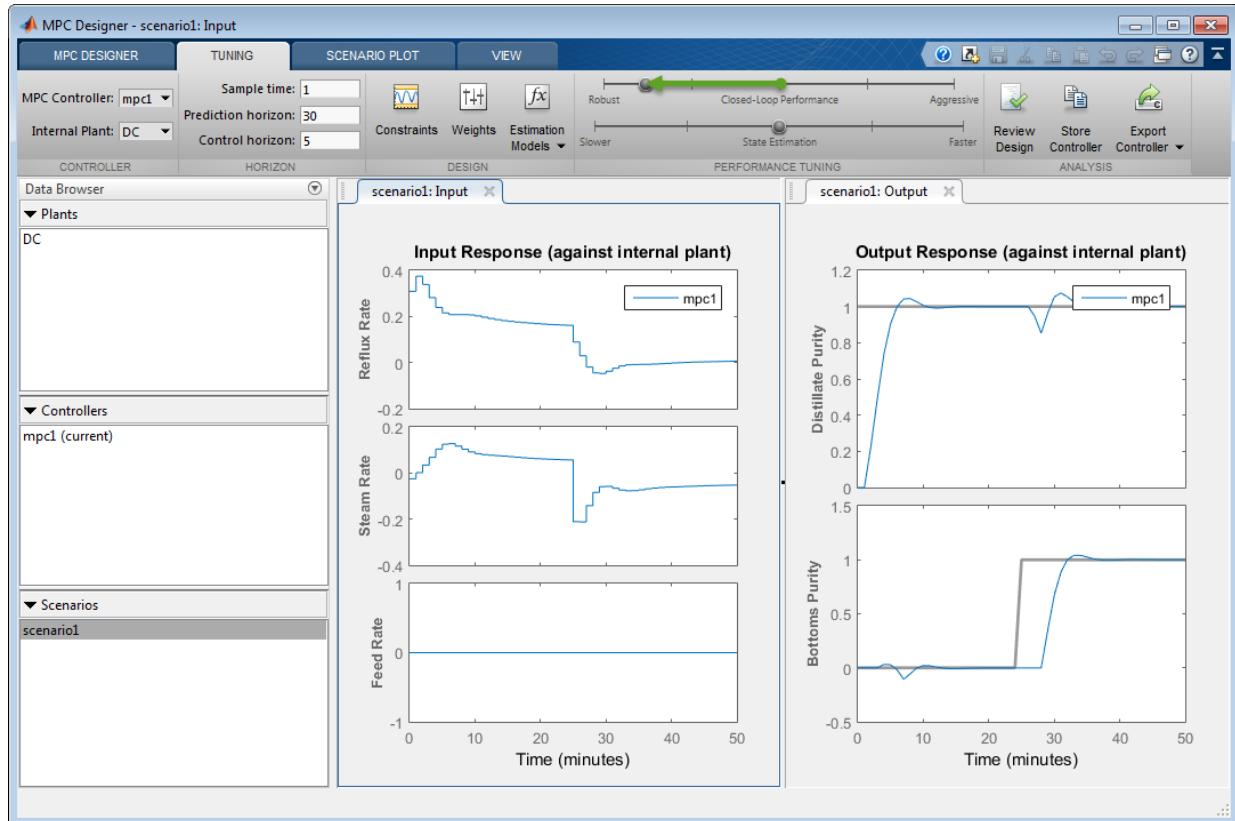
The initial manipulated variable moves are much smaller and the moves are less oscillatory. The trade-off is a slower output response, with larger interactions between the outputs.

Improve Performance By Tuning Controller Weights

Alternatively, you can produce smooth manipulated variable moves by adjusting the tuning weights of the controller.

Set the **Control horizon** back to the previous value of 5.

In the **Performance Tuning** section, drag the **Closed-Loop Performance** slider to the left towards the **Robust** setting.



As you move the slider to the left, the manipulated variable moves become smoother and the output response becomes slower.

References

- [1] Wood, R. K., and M. W. Berry, *Chem. Eng. Sci.*, Vol. 28, pp. 1707, 1973.

See Also

MPC Designer

Related Examples

- “Design Controller Using MPC Designer” on page 3-2
- “Specify Multi-Input Multi-Output (MIMO) Plants” on page 2-17

More About

- “Manipulated Variable Blocking”

Design MPC Controller for Nonsquare Plant

This topic shows how to configure an MPC controller for a nonsquare plant with unequal numbers of manipulated variables and outputs. Model Predictive Control Toolbox software supports plants with an excess of manipulated variables or plant with an excess of outputs.

More Outputs Than Manipulated Variables

When there are excess outputs, you cannot hold each at a setpoint. In this case, you have two options:

- Specify that certain outputs do not need to be held at setpoints by setting their tuning weights to zero.

The controller does not enforce setpoints on outputs with zero weight, and the outputs are free to vary. If the plant has N_e more outputs than manipulated variables, setting N_e output weights to zero enables the controller to hold the remaining outputs at their setpoints. If any manipulated variables are constrained, one or more output responses can still exhibit steady-state error, depending on the magnitudes of reference and disturbance signals.

Outputs with zero tuning weights can still be useful. If measured, the controller can use the outputs to help estimate the state of the plant. The outputs can also be used as performance indicators or held within an operating region defined by output constraints.

- Enforce setpoints on all outputs by specifying nonzero tuning weights for all of them.

The controller tries to hold all outputs at their respective setpoints. However, due to the limited number of manipulated variables, all output responses exhibit some degree of steady-state error.

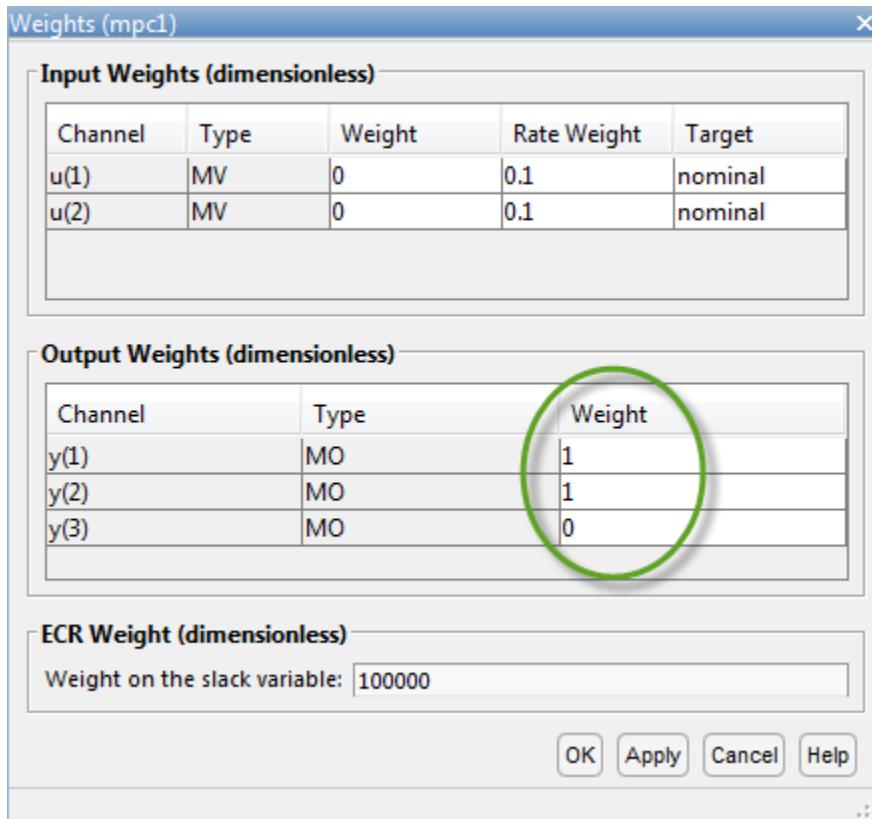
You can change the error magnitudes by adjusting the relative values of the output weights. Increasing an output weight decreases the steady-state error in that output at the expense of increased error in the other outputs.

You can configure the output tuning weights at the command line by setting the `Weights.OutputVariables` property of the controller.

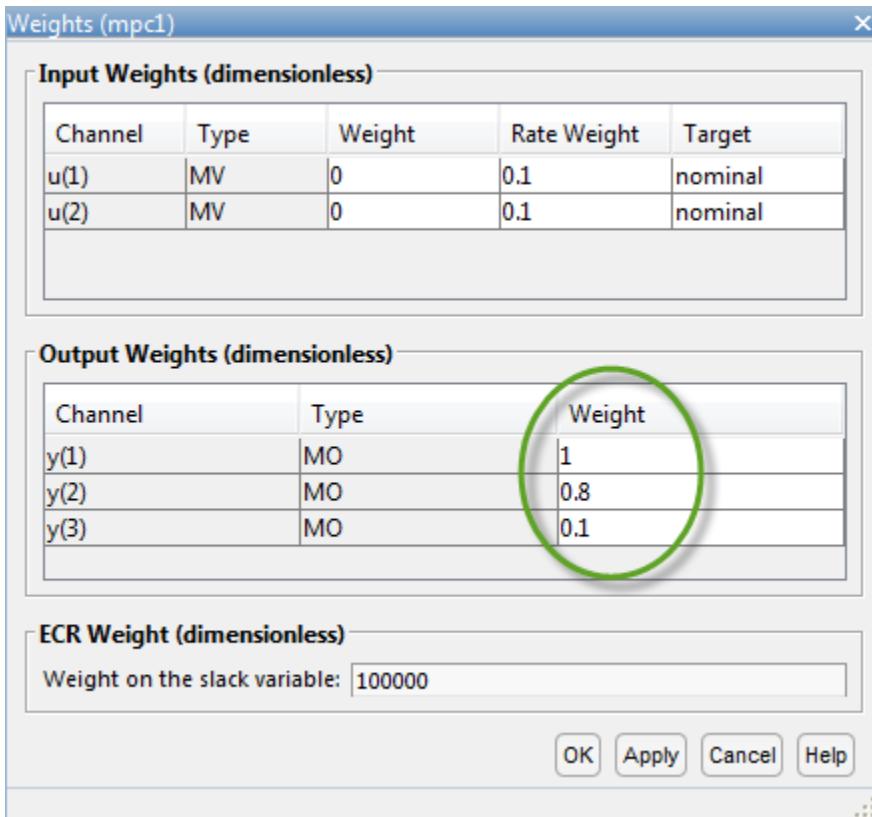
To configure output tuning weights in the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Weights** to open the Weights dialog box.

In the **Output Weights** section, specify the **Weight** for each output variable. For example, if your plant has two manipulated variables and three outputs, you can:

- Set one of the output weights to zero.



- Set all the weights to nonzero values. Outputs with higher weights exhibit less steady-state error.



More Manipulated Variables Than Outputs

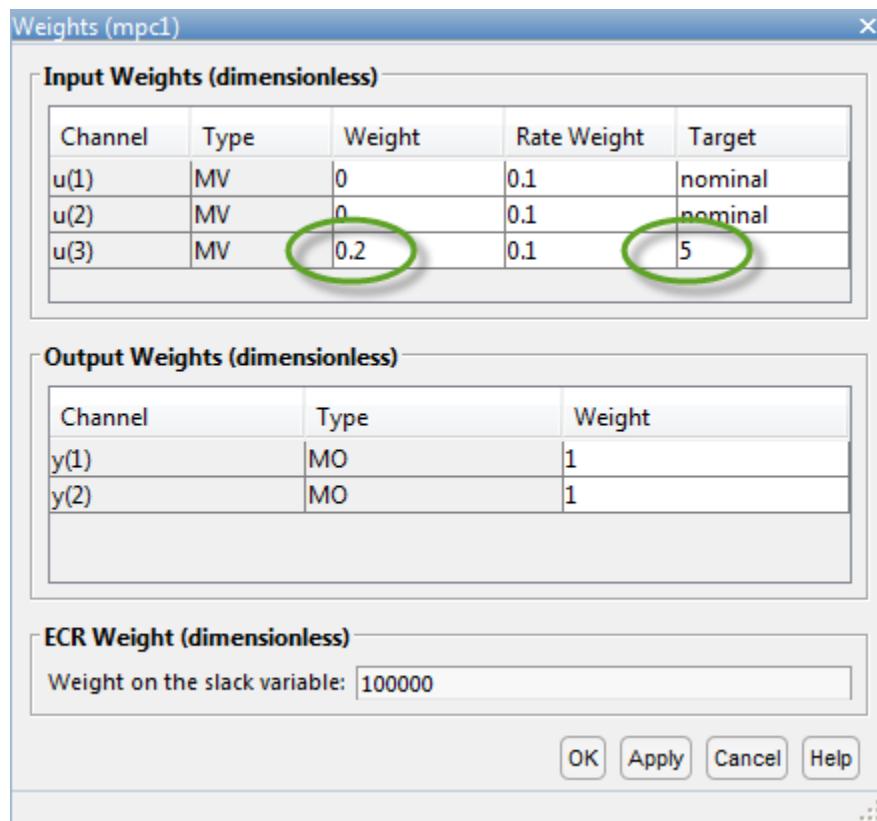
When there are excessive manipulated variables, the default MPC controller settings allow for error-free output setpoint tracking. However, the manipulated variables values can drift. You can prevent this drift by setting manipulated variable setpoints. If there are N_e excess manipulated variables, and you hold N_e of them at target values for economic or operational reasons, the remaining manipulated variables attain the values required to eliminate output steady-state error.

To configure a manipulated variable setpoint at the command line, use the `ManipulatedVariables.Target` controller property. Then specify an input tuning weight using the controller `Weights.ManipulatedVariables` property.

To define a manipulated variable setpoint in the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Weights**.

In the Weights dialog box, in the **Input Weights** section, specify a nonzero **Weight** value for the manipulated variable.

Specify a **Target** value for the manipulated variable.



By default, the manipulated variable **Target** is **nominal**, which means that it tracks the nominal value specified in the controller properties.

Note: Since nominal values apply to all controllers in an MPC Designer session, changing a **Nominal Value** updates all controllers in the app. The **Target** value, however, is specific to each individual controller.

The magnitude of the manipulated variable weight indicates how much the input can deviate from its setpoint. However, there is a trade-off between manipulated variable target tracking and output reference tracking. If you want to have better output setpoint tracking performance, use a relatively small input weight. If you want the manipulated variable to stay close to its target value, increase its input weight relative to the output weight.

You can also avoid drift by constraining one or more manipulated variables to a narrow operating region using hard constraints. To define constraints in the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Constraints** to open the Constraints dialog box.

In the **Input Constraints** section, specify **Max** and **Min** constraints values.

See Also

`mpc` | MPC Designer

Related Examples

- “Specify Multi-Input Multi-Output (MIMO) Plants” on page 2-17
- “Setting Targets for Manipulated Variables”

More About

- “Tuning Weights”

Designing Controllers Using the Command Line

- “Design MPC Controller at the Command Line” on page 4-2
- “Simulate Controller with Nonlinear Plant” on page 4-16
- “Control Based On Multiple Plant Models” on page 4-23
- “Compute Steady-State Gain” on page 4-32
- “Extract Controller” on page 4-34
- “Signal Previewing” on page 4-37
- “Run-Time Constraint Updating” on page 4-39
- “Run-Time Weight Tuning” on page 4-40

Design MPC Controller at the Command Line

This example shows how to create and test a model predictive controller from the command line.

Define Plant Model

This example uses the plant model described in “Design Controller Using MPC Designer”. Create a state space model of the plant and set some of the optional model properties.

```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);

CSTR.InputName = { T_c , C_A_i };
CSTR.OutputName = { T , C_A };
CSTR.StateName = { C_A , T };
CSTR.InputGroup.MV = 1;
CSTR.InputGroup.UD = 2;
CSTR.OutputGroup.MO = 1;
CSTR.OutputGroup.UO = 2;
```

Create Controller

To improve the clarity of the example, suppress Command Window messages from the MPC controller.

```
old_status = mpcverbosity( off );
```

Create a model predictive controller with a control interval, or sample time, of 1 second, and with all other properties at their default values.

```
Ts = 1;
MPCobj = mpc(CSTR,Ts);
```

Display the controller properties in the Command Window.

```
display(MPCobj)
```

```
MPC object (created on 15-Feb-2016 14:33:35):
```

```

-----
Sampling time:      1 (seconds)
Prediction Horizon: 10
Control Horizon:    2
Model:
    Plant: [2x2 ss]
    Disturbance: []
    Noise: []
    Nominal: [1x1 struct]

    Output disturbance model: default method (type "getoutdist(MPCobj)" for details)
    Input disturbance model: default method (type "getindist(MPCobj)" for details)

Details on Plant model:
-----
  1 manipulated variable(s)  -->| 2 states |--> 1 measured output(s)
  0 measured disturbance(s)  -->| 2 inputs |--> 1 unmeasured output(s)
  1 unmeasured disturbance(s) -->| 2 outputs |
-----
Indices:
  (input vector)      Manipulated variables: [1 ]
                    Unmeasured disturbances: [2 ]
  (output vector)     Measured outputs: [1 ]
                    Unmeasured outputs: [2 ]

Weights: (default)
  ManipulatedVariables: 0
  ManipulatedVariablesRate: 0.1000
  OutputVariables: [1 0]
  ECR: 100000

State Estimation: Default Kalman gain

Unconstrained

```

View and Modify Controller Properties

Display a list of the controller properties and their current values.

```
get(MPCobj)
```

```
ManipulatedVariables (MV): [1x1 struct]
```

```
OutputVariables (OV): [1x2 struct]
DisturbanceVariables (DV): [1x1 struct]
Weights (W): [1x1 struct]
Model: [1x1 struct]
Ts: 1
Optimizer: [1x1 struct]
PredictionHorizon (P): 10
ControlHorizon: 2
History: [2.02e+03 2 15 14 33 35.1]
Notes: {}
UserData: []
```

The displayed **History** value will be different for your controller, since it depends on when the controller was created. For a description of the editable properties of an MPC controller, enter **mpcprops** at the command line.

Use dot notation to modify these properties. For example, change the prediction horizon to 15.

```
MPCobj.PredictionHorizon = 15;
```

You can abbreviate property names provided that the abbreviation is unambiguous.

Many of the controller properties are structures containing additional fields. Use dot notation to view and modify these field values. For example, you can set the measurement units for the controller output variables. The **OutputUnit** property is for display purposes only and is optional.

```
MPCobj.Model.Plant.OutputUnit = { Deg C , kmol/m^3 };
```

By default, the controller has no constraints on manipulated variables and output variables. You can view and modify these constraints using dot notation. For example, set constraints for the controller manipulated variable.

```
MPCobj.MV.Min = -10;
MPCobj.MV.Max = 10;
MPCobj.MV.RateMin = -3;
MPCobj.MV.RateMax = 3;
```

You can also view and modify the controller tuning weights. For example, modify the weights for the manipulated variable rate and the output variables.

```
MPCobj.W.ManipulatedVariablesRate = 0.3;
```

```
MPCobj.W.OutputVariables = [1 0];
```

You can also define time-varying constraints and weights over the prediction horizon, which shifts at each time step. Time-varying constraints have a nonlinear effect when they are active. For example, to force the manipulated variable to change more slowly towards the end of the prediction horizon, enter:

```
MPCobj.MV.RateMin = [-4; -3.5; -3; -2.5];
```

```
MPCobj.MV.RateMax = [4; 3.5; 3; 2.5];
```

The **-2.5** and **2.5** values are used for the fourth step and beyond.

Similarly, you can specify different output variable weights for each step of the prediction horizon. For example, enter:

```
MPCobj.W.OutputVariables = [0.1 0; 0.2 0; 0.5 0; 1 0];
```

You can also modify the disturbance rejection characteristics of the controller. See `setEstimator`, `setindist` and `setoutdist` for more information.

Review Controller Design

Generate a report on potential run-time stability and performance issues.

```
review(MPCobj)
```

The screenshot shows a web browser window with the title "Web Browser - Review MPC Object "MPCObj"" and the sub-tab "Review MPC Object "MPCObj"" selected. The main content area displays a large heading "Design Review for Model Predictive Controller "MPCObj"" followed by a section titled "Summary of Performed Tests". Below this, a table lists various tests and their status. The table has two columns: "Test" and "Status". The "Test" column contains links to individual test results, and the "Status" column indicates whether each test passed or failed.

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Pass
Hard MV Constraints	Warning
Other Hard Constraints	Pass
Soft Constraints	Pass
Memory Size for MPC Data	Pass

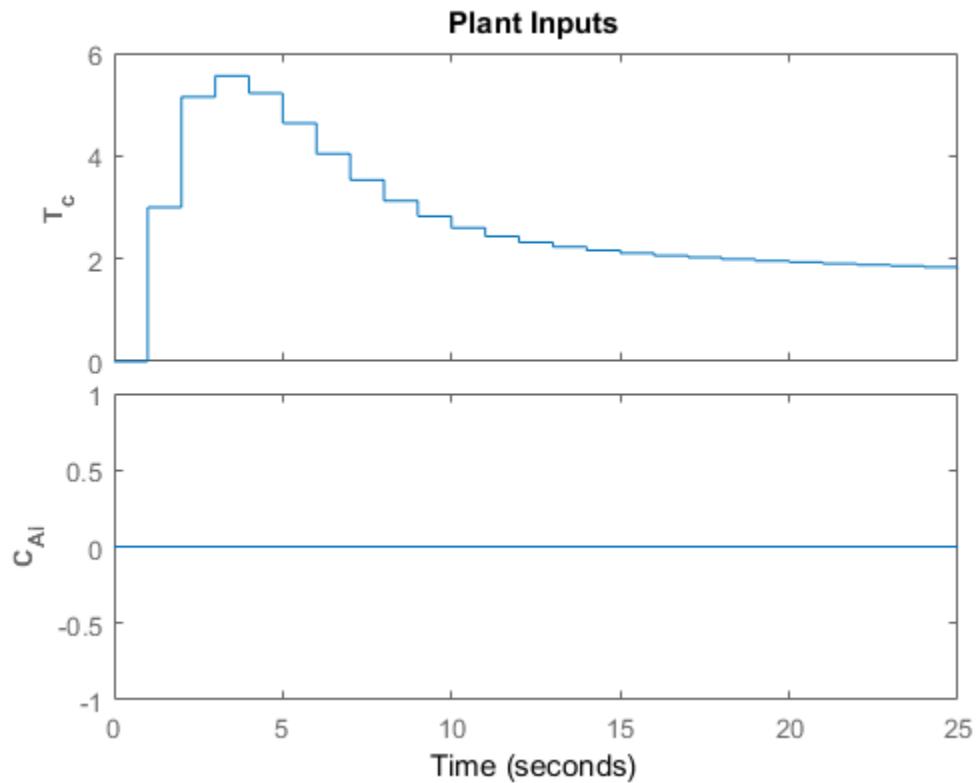
Individual Test Result

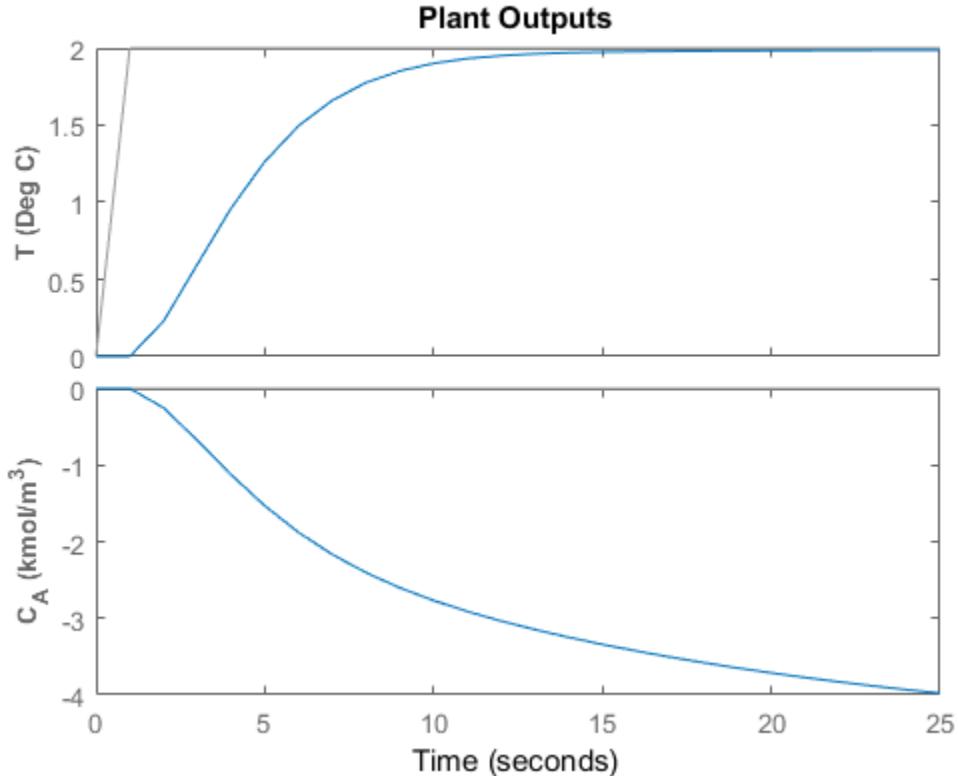
In this example, the review command found two potential issues with the design. The first warning asks whether the user intends to have a weight of zero on the C_A output. The second warning advises the user to avoid having hard constraints on both MV and MVRate.

Perform Linear Simulations

Use the sim function to run a linear simulation of the system. For example, simulate the closed-loop response of MPCobj for 26 control intervals. Specify setpoints of 2 and 0 for the reactor temperature and the residual concentration respectively. The setpoint for the residual concentration is ignored because the tuning weight for the second output is zero.

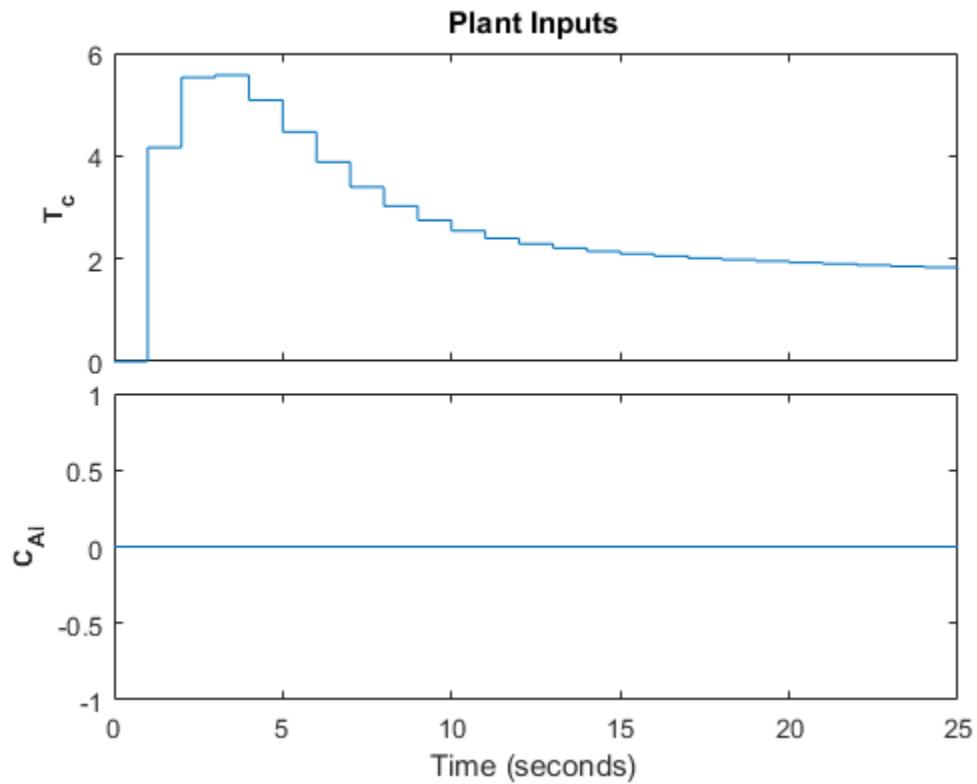
```
T = 26;  
r = [0 0; 2 0];  
sim(MPCobj,T,r)
```

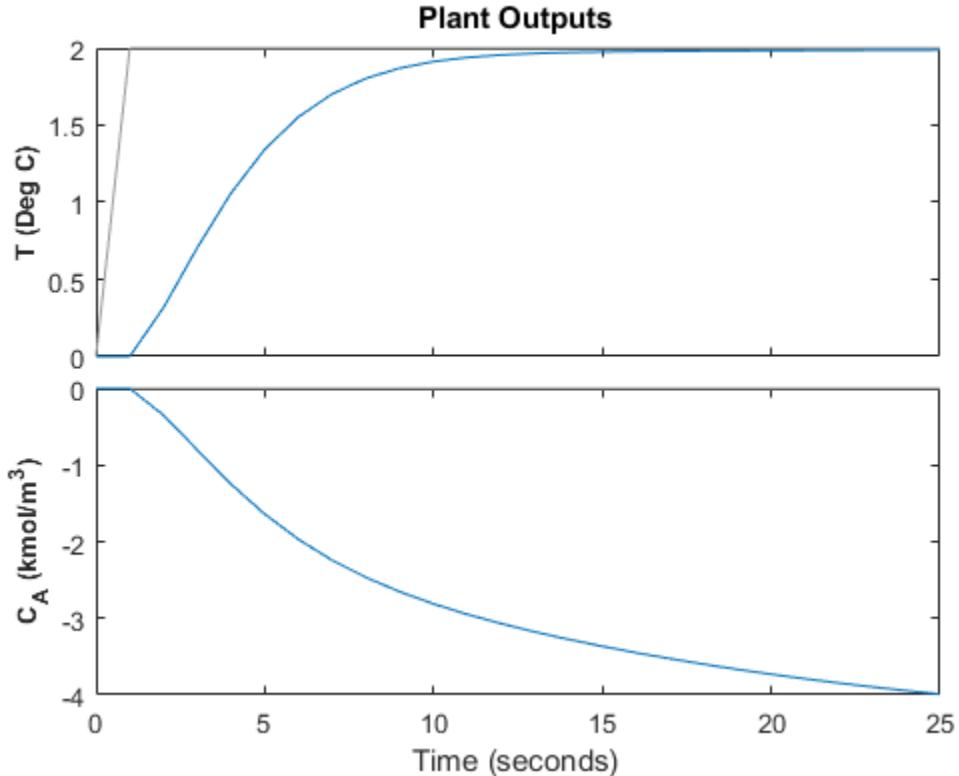




You can modify the simulation options using `mpcsimopt`. For example, run a simulation with the manipulated variable constraints turned off.

```
MPCopts = mpcsimopt;
MPCopts.Constraints = off ;
sim(MPCobj,T,r,MPCopts)
```

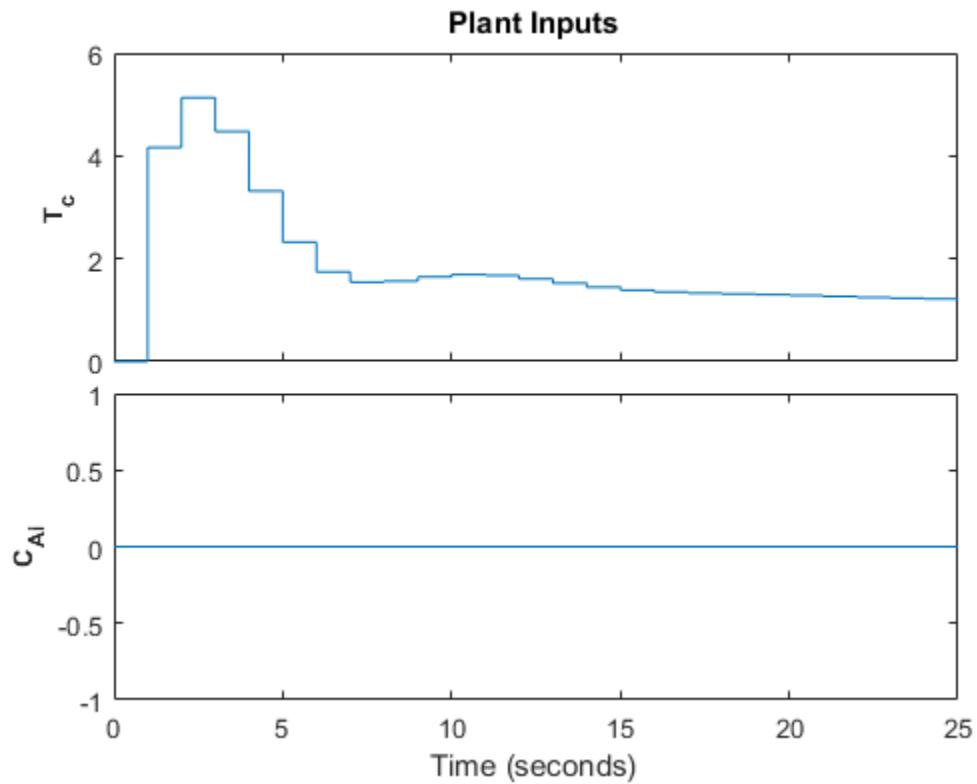


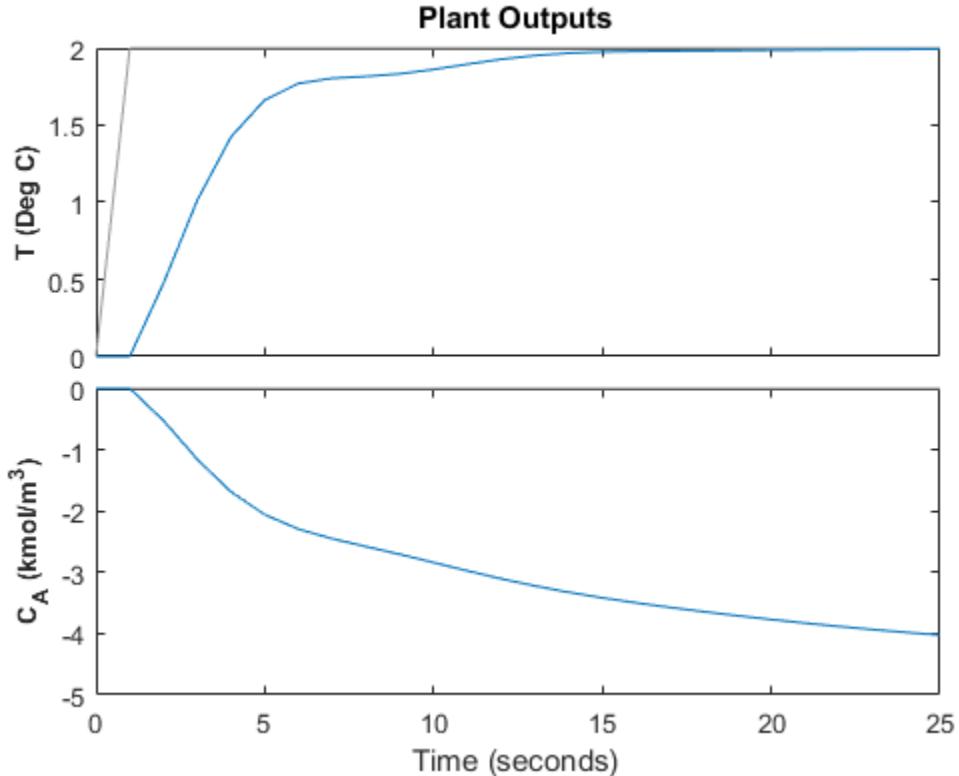


The first move of the manipulated variable now exceeds the specified 3-unit rate constraint.

You can also perform a simulation with a plant/model mismatch. For example, define a plant with 50% larger gains than those in the model used by the controller.

```
Plant = 1.5*CSTR;
MPCopts.Model = Plant;
sim(MPCobj,T,r,MPCopts)
```





The plant/model mismatch degrades controller performance slightly. Degradation can be severe and must be tested on a case-by-case basis.

Other options include the addition of a specified noise sequence to the manipulated variables or measured outputs, open-loop simulations, and a look-ahead option for better setpoint tracking or measured disturbance rejection.

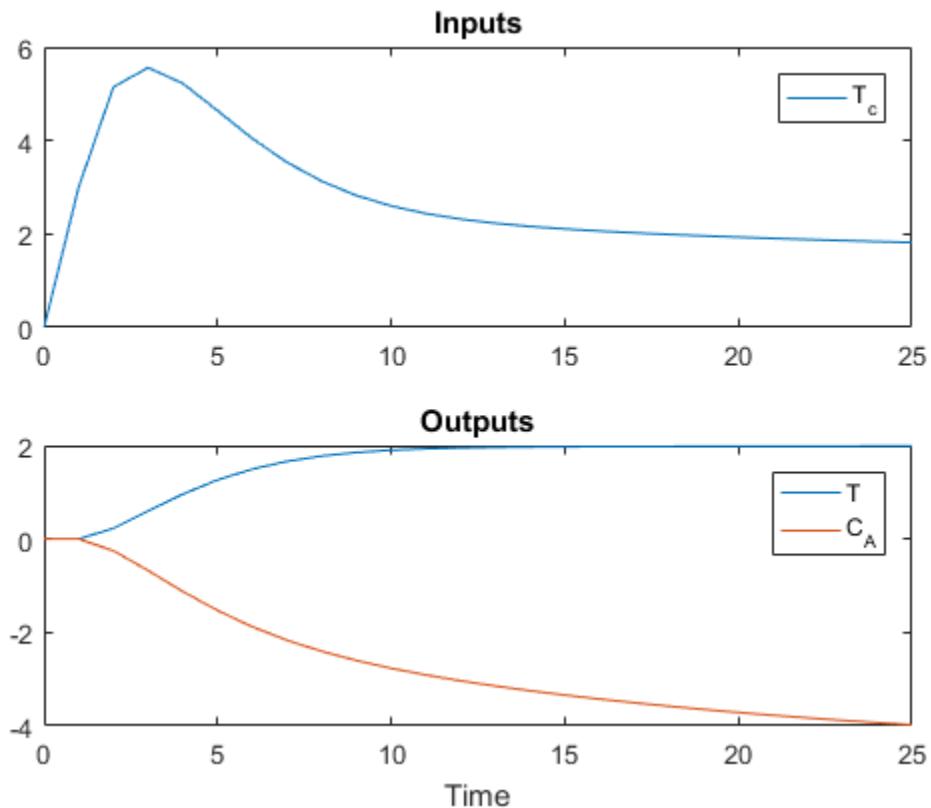
Store Simulation Results

Store the simulation results in the MATLAB Workspace.

```
[y,t,u] = sim(MPCobj,T,r);
```

This syntax suppresses automatic plotting and returns the simulation results. You can use the results for other tasks, including custom plotting. For example, plot the manipulated variable and both output variables in the same figure.

```
figure
subplot(2,1,1)
plot(t,u)
title( Inputs )
legend( T_c )
subplot(2,1,2)
plot(t,y)
title( Outputs )
legend( T , C_A )
xlabel( Time )
```



Restore the `mpcverbosity` setting.

```
mpcverbosity(old_status);
```

See Also

`mpc` | `review` | `sim`

Related Examples

- “Design Controller Using MPC Designer” on page 3-2
- “Design MPC Controller in Simulink” on page 5-2

More About

- “MPC Modeling” on page 2-2

Simulate Controller with Nonlinear Plant

You can use `sim` to simulate a closed-loop system consisting of a linear plant model and an MPC controller.

If your plant is a nonlinear Simulink model, you must linearize the plant (see “Linearization Using Linear Analysis Tool in Simulink Control Design” on page 2-25) and design a controller for the linear model (see “Design MPC Controller in Simulink” on page 5-2). To simulate the system, specify the controller in the MPC block parameter **MPC Controller** field and run the closed-loop Simulink model.

Alternatively, your nonlinear model might be a MEX-file, or you might want to include features unavailable in the MPC block, such as a custom state estimator. The `mpcmove` function is the Model Predictive Control Toolbox computational engine, and you can use it in such cases. The disadvantage is that you must duplicate the infrastructure that the `sim` function and the MPC block provide automatically.

The rest of this section covers the following topics:

- “Nonlinear CSTR Application” on page 4-16
- “Example Code for Successive Linearization” on page 4-17
- “CSTR Results and Discussion” on page 4-19

Nonlinear CSTR Application

The CSTR model described in “Linearize Simulink Models” on page 2-22 is a strongly nonlinear system. As shown in “Design MPC Controller in Simulink” on page 5-2, a controller can regulate this plant, but degrades (and might even become unstable) if the operating point changes significantly.

The objective of this example is to redefine the predictive controller at the beginning of each control interval so that its predictive model, though linear, represents the latest plant conditions as accurately as possible. This will be done by linearizing the nonlinear model repeatedly, allowing the controller to adapt as plant conditions change. See references [1] and [2] for more details on this approach.

Example Code for Successive Linearization

In the following code, the simulation begins at the nominal operating point of the CSTRmodel (concentration = 8.57) and moves to a lower point (concentration = 2) where the reaction rate is much higher. The required code is as follows:

```
[sys, xp] = CSTR_INOUT([],[],[], sizes );
up = [10 298.15 298.15];
u = up(3);
t save = [];
usave = [];
ysave = [];
r save = [];
Ts = 1;
t = 0;
while t < 40
    yp = xp;
    % Linearize the plant model at the current conditions
    [a,b,c,d] = linmod( CSTR_INOUT , xp, up);
    Plant = ss(a,b,c,d);
    Plant.InputGroup.ManipulatedVariables = 3;
    Plant.InputGroup.UnmeasuredDisturbances = [1 2];
    Model.Plant = Plant;

    % Set nominal conditions to the latest values
    Model.Nominal.U = [0 0 u];
    Model.Nominal.X = xp;
    Model.Nominal.Y = yp;

    dt = 0.001;

    simOptions.StartTime = num2str(t);
    simOptions.StopTime = num2str(t+dt);
    simOptions.LoadInitialState = on ;
    simOptions.InitialState = xp ;
    simOptions.SaveTime = on ;
    simOptions.SaveState = on ;
    simOptions.LoadExternalInput = on ;
    simOptions.ExternalInput = [t up; t+dt up] ;

    simOut = sim( CSTR_INOUT ,simOptions);

    T = simOut.get( tout );
    XP = simOut.get( xout );
```

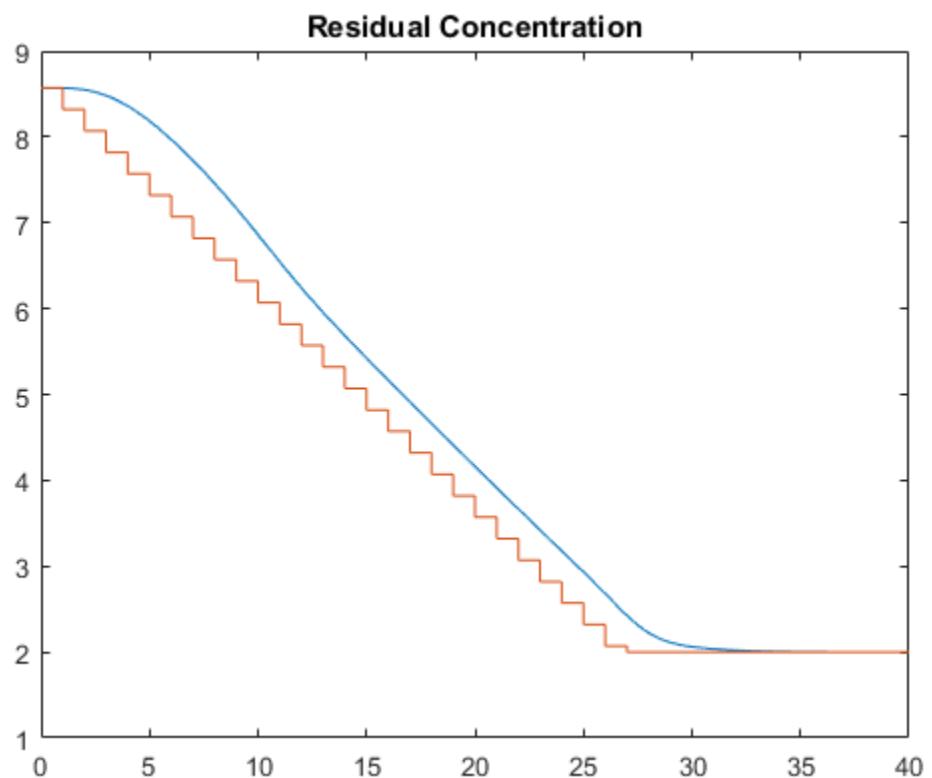
```
YP = simOut.get( yout );  
  
Model.Nominal.DX = (1/dt)*(XP(end,:)-xp(:));  
  
% Define MPC Toolbox controller for the latest model  
MPCobj = mpc(Model, Ts);  
MPCobj.W.Output = [0 1];  
  
% Ramp the setpoint  
r = max([8.57 - 0.25*t, 2]);  
  
% Compute the control action  
if t <= 0  
    xd = [0; 0];  
    x = mpcstate(MPCobj, xp, xd, [], u);  
end  
  
u = mpcmove(MPCobj, x, yp, [0 r], []);  
  
% Simulate the plant for one control interval  
up(3) = u;  
  
simOptions.StartTime = num2str(t);  
simOptions.StopTime = num2str(t+Ts);  
simOptions.InitialState = xp ;  
simOptions.ExternalInput = [t up; t+Ts up] ;  
  
simOut = sim( CSTR_INOUT ,simOptions);  
  
T = simOut.get( tout );  
XP = simOut.get( xout );  
YP = simOut.get( yout );  
  
% Save results for plotting  
tsave = [tsave; T];  
ysave = [ysave; YP];  
useave = [useave; up(ones(length(T),1),:)];  
rsave = [rsave; r(ones(length(T),1),:)];  
  
xp = XP(end,:);  
  
t = t + Ts;  
end
```

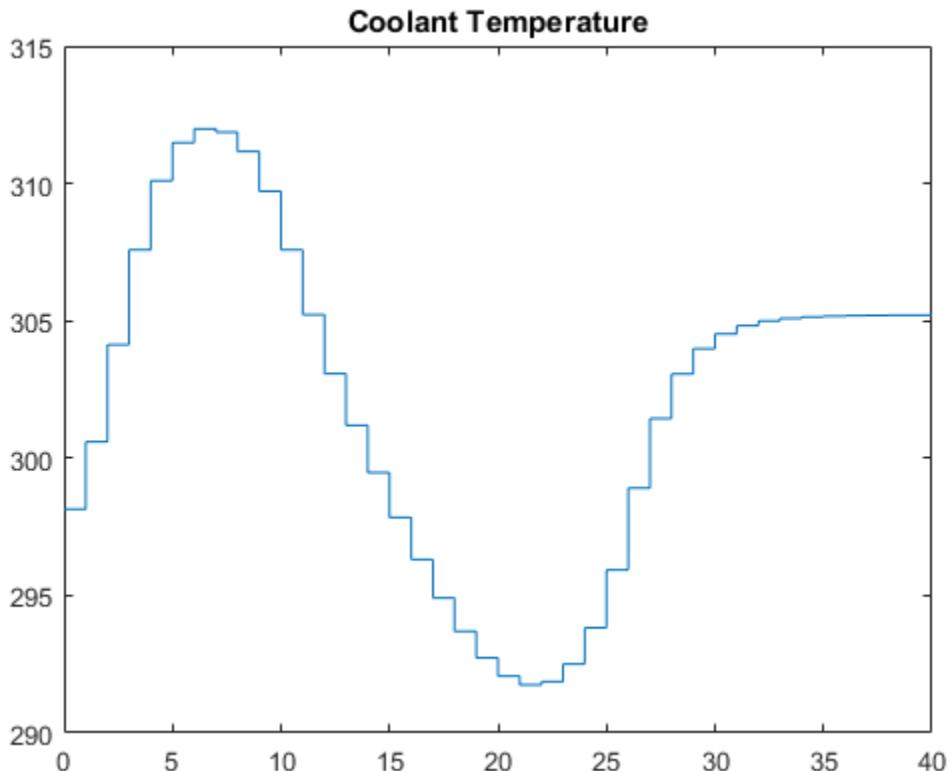
```
figure(1)
plot(tsave,[ysave(:,2) rsave])
title( Residual Concentration )
figure(2)
plot(tsave,useave(:,3))
title( Coolant Temperature )
```

CSTR Results and Discussion

The plotted results appear below. Note the following points:

- The setpoint is being ramped from the initial concentration to the desired final value (see the step-wise changes in the reactor concentration plot below). The reactor concentration tracks this ramp smoothly with some delay (see the smooth curve), and settles at the final state with negligible overshoot. The controller works equally well (and achieves the final concentration more rapidly) for a step-wise setpoint change, but it makes unrealistically rapid changes in coolant temperature (not shown).
- The final steady state requires a coolant temperature of 305.20 K (see the coolant temperature plot below). An interesting feature of this nonlinear plant is that if one starts at the initial steady state (coolant temperature = 298.15 K), stepping the coolant temperature to 305.20 and holding will not achieve the desired final concentration of 2. In fact, under this simple strategy the reactor concentration stabilizes at a final value of 7.88, far from the desired value. A successful controller must increase the reactor temperature until the reaction “takes off,” after which it must reduce the coolant temperature to handle the increased heat load. The relinearization approach provides such a controller (see following plots).





- Function `linearize` relinearizes the plant as its state evolves. This function was discussed previously in “Linearization Using MATLAB Code” on page 2-22.
- The code also resets the linear model’s nominal conditions to the latest values. Note, however, that the first two input signals, which are unmeasured disturbances in the controller design, always have nominal zero values. As they are unmeasured, the controller cannot be informed of the true values. A non-zero value would cause an error.
- Function `mpc` defines a new controller based on the relinearized plant model. The output weight tuning ignores the temperature measurement, focusing only on the concentration.
- At $t = 0$, the `mpcstate` function initializes the controller’s extended state vector, \mathbf{x} , which is an *mpcstate object*. Thereafter, the `mpcmove` function updates it

automatically using the controller's default state estimator. It would also be possible to use an Extended Kalman Filter (EKF) as described in [1] and [2], in which case the EKF would reset the `mpcstate` input variables at each step.

- The `mpcmove` function uses the latest controller definition and state, the measured plant outputs, and the setpoints to calculate the new coolant temperature at each step.
- The Simulink `sim` function simulates the nonlinear plant from the beginning to the end of the control interval. Note that the final condition from the previous step is being used as the initial plant state, and that the plant inputs are being held constant during each interval.

Remember that a conventional feedback controller or a fixed Model Predictive Control Toolbox controller tuned to operate at the initial condition would become unstable as the plant moves to the final condition. Periodic model updating overcomes this problem automatically and provides excellent control under all conditions.

References

- [1] Lee, J. H. and N. L. Ricker, "Extended Kalman Filter Based Nonlinear Model Predictive Control," *Ind. Eng. Chem. Res.*, Vol. 33, No. 6, pp. 1530–1541 (1994).
- [2] Ricker, N. L., and J. H. Lee "Nonlinear Model Predictive Control of the Tennessee Eastman Challenge Process," *Computers & Chemical Engineering*, Vol. 19, No. 9, pp. 961–981 (1995).

Control Based On Multiple Plant Models

The “Nonlinear CSTR Application” on page 4-16 shows how updates to the prediction model can improve MPC performance. In that case the model is nonlinear and you can obtain frequent updates by linearization.

A more common situation is that you have several linear plant models, each of which applies at a particular operating condition and you can design a controller based on each linear model. If the models cover the entire operating region and you can define a criterion by which you switch from one to another as operating conditions change, the controller set should be able to provide better performance than any individual controller.

The Model Predictive Control Toolbox includes a Simulink block that performs this function. It is the *Multiple MPC Controllers* block. The rest of this section is an illustrative example organized as follows:

- “A Two-Model Plant” on page 4-23
- “Designing the Two Controllers” on page 4-25
- “Simulating Controller Performance” on page 4-26

A Two-Model Plant

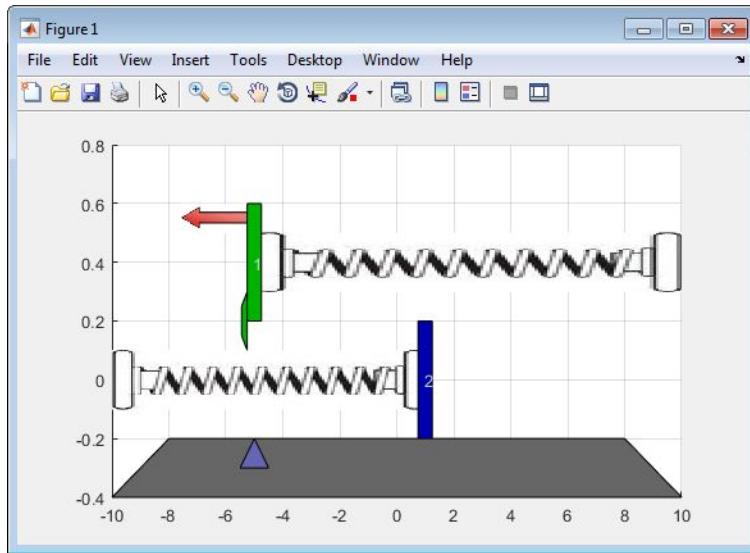
Note Switching MPC Controllers with Multiple MPC Controllers Block provides an animated version of the plant described below.

Animation of the Multi-Model Example is a stop-action snapshot of the subject plant. It consists of two masses, M_1 and M_2 . A spring connects M_1 to a rigid wall and pulls it to the right. An applied force, shown as a red arrow in Animation of the Multi-Model Example, opposes this spring, pulling M_1 to the left.

When the two masses are detached, as in Animation of the Multi-Model Example, mass M_2 is uncontrollable and responds only to the spring pulling it to the left.

If the two masses collide, however, they stick together (the collision is completely inelastic) until a change in the applied force separates them.

The control objective is to move M_1 in response to a command signal. The blue triangle in Animation of the Multi-Model Example represents the desired location. At the instant shown, the desired location is -5 .



Animation of the Multi-Model Example

In order to achieve its objective, the controller can adjust the applied force magnitude (the length of the red arrow). It receives continuous feedback on the M_1 location. There is also a contact sensor to signal collisions. The M_2 location is unmeasured.

If M_1 were isolated, this would be a routine control problem. The challenge is that the relationship between the applied force and the M_1 movement changes dramatically when M_2 attaches to M_1 .

The following code defines the model. First define the system parameters as follows:

```
%% Model Parameters
M1 = 1;          % mass
M2 = 5;          % mass
k1 = 1;          % spring constant
k2 = 0.1;         % spring constant
b1 = 0.3;         % friction coefficient
b2 = 0.8;         % friction coefficient
yeq1 = 10;        % wall mount position
yeq2 = -10;       % wall mount position
```

Next define a model of M_1 when the masses are separated. Its states are the M_1 position and velocity. Its inputs are the applied force, which will be the controller's manipulated

variable, and a spring constant calibration signal, which is a measured disturbance input.

```
A1 = [0 1;-k1/M1 -b1/M1];
B1 = [0 0;-1/M1 k1*yeq1/M1];
C1 = [1 0];
D1 = [0 0];
sys1 = ss(A1,B1,C1,D1);
sys1 = setmpcsignals(sys1, MV ,1, MD ,2);
```

The `setmpcsignals` command specifies the input type for the two inputs.

We need another model (with the same input/output structure) to predict movement when the two masses are joined, as follows:

```
A2 = [0 1;-(k1+k2)/(M1+M2) -(b1+b2)/(M1+M2)];
B2 = [0 0;-1/(M1+M2) (k1*yeq1+k2*yeq2)/(M1+M2)];
C2 = [1 0];
D2 = [0 0];
sys2 = ss(A2,B2,C2,D2);
sys2 = setmpcsignals(sys2, MV ,1, MD ,2);
```

Designing the Two Controllers

Next we define controllers for each case. Both use a sample time of 0.2 , a prediction horizon of $P = 20$, a control horizon of $M = 1$, and the default values for all other controller design parameters. The only difference in the controllers is the prediction model.

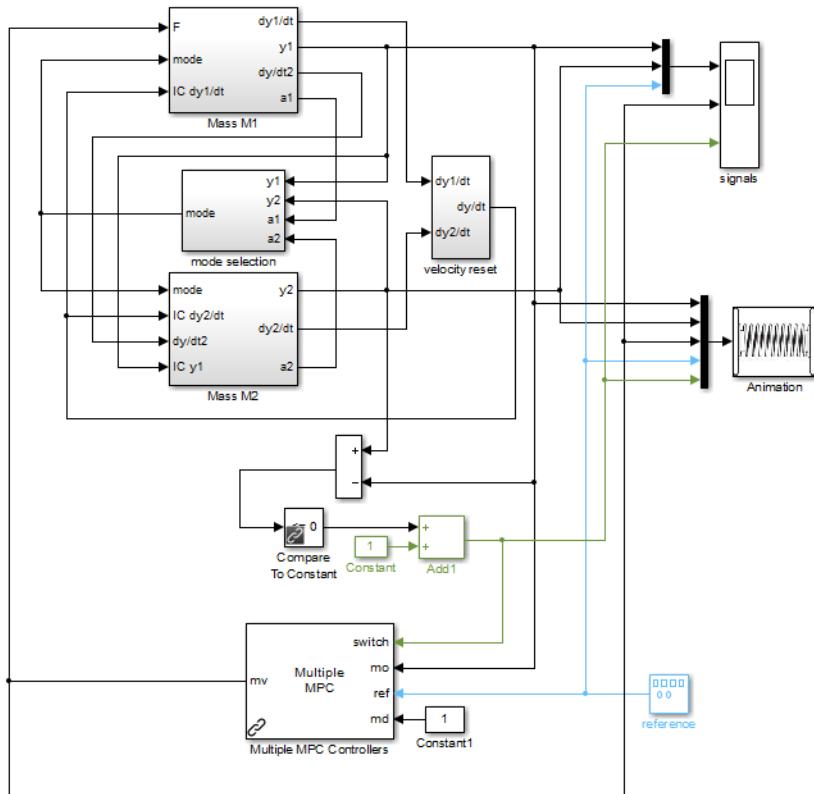
```
Ts = 0.2; % sampling time
p = 20; % prediction horizon
m = 1; % control horizon
MPC1 = mpc(sys1,Ts,p,m); % Controller for M1 detached from M2
MPC2 = mpc(sys2,Ts,p,m); % Controller for M1 connected to M2
```

The applied force also has the same constraints in each case. Its lower bound is zero (it can't reverse direction), and its maximum rate of change is 1000 per second (increasing or decreasing).

```
MPC1.MV = struct( Min ,0, RateMin ,-1e3, RateMax ,1e3);
MPC2.MV = struct( Min ,0, RateMin ,-1e3, RateMax ,1e3);
```

Simulating Controller Performance

Block Diagram of the Two-Model Example shows the Simulink block diagram for this example. The upper portion simulates the movement of the two masses, plots the signals as a function of time, and animates the example.



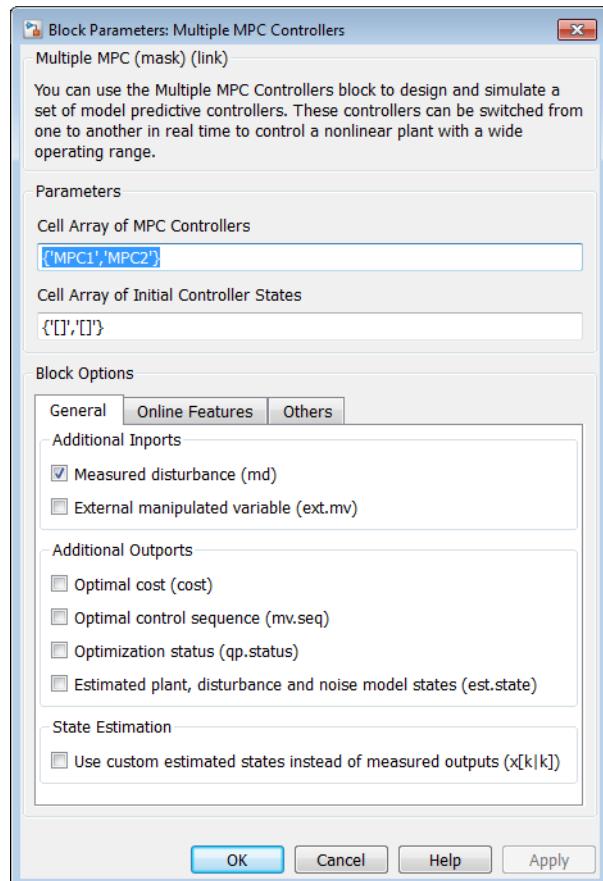
Block Diagram of the Two-Model Example

The lower part contains three key elements:

- A pulse generator that supplies the desired M_1 position (the controller reference signal). Its output is a square wave varying between -5 and 5 with a frequency of 0.015 per second.

- A simulation of a contact sensor. When the two masses have the same position, the **Compare to Constant** block evaluates to **true**, and the **Add1** block converts this to a 2. Otherwise, the **Add1** output is 1.
- The Multiple MPC Controller block. It has four inputs. The measured output (**mo**), reference (**ref**), and measured disturbance (**md**) inputs are as for a standard MPC Controller block. The distinctive feature is the **switch** input.

The figure below shows the Multiple MPC Controller block mask for this example (obtained by double clicking on the controller block).

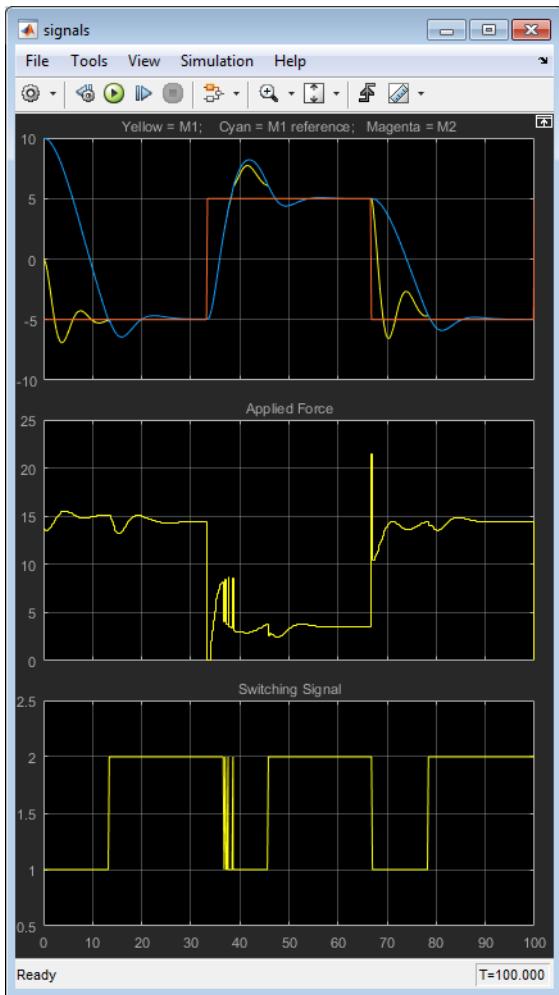


When the **switch** input is 1 the block automatically activates the first controller listed (**MPC1**), which is appropriate when the masses are separated. When the **switch** input is 2 the block automatically enables the second controller (**MPC2**).

The following code simulates the controller performance

```
Tstop = 100; % Simulation time  
y1initial = 0; % Initial M1 and M2 Positions  
y2initial = 10;  
open( mpc_switching )  
sim( mpc_switching ,Tstop)
```

The figure below shows the **signals** scope output.



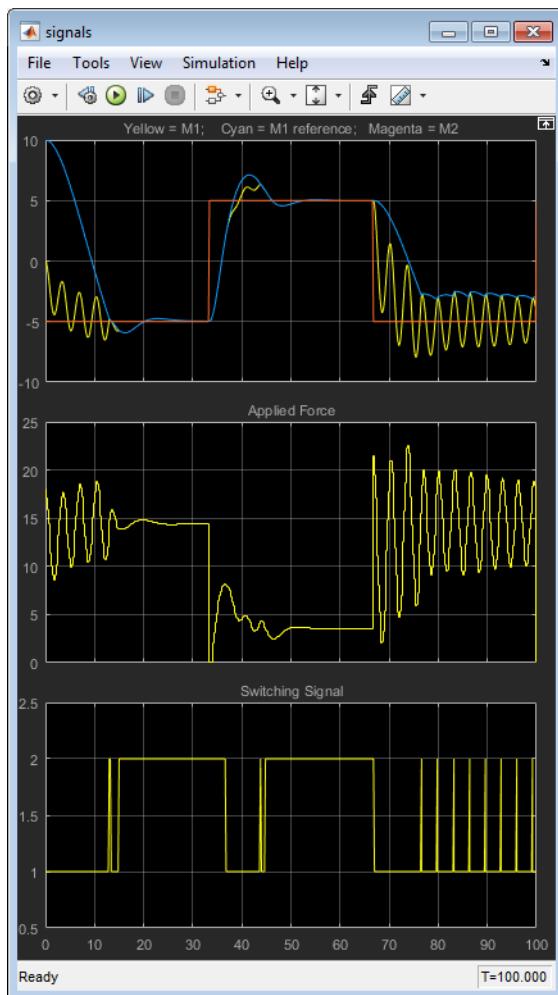
In the upper plot, the cyan curve is the desired position. It starts at -5 . The M_1 position (yellow) starts at 0 and under the control of MPC1, M_1 moves rapidly toward the desired position. M_2 (magenta) starts at 10 and begins moving in the same direction. At about $t = 13$ seconds, M_2 collides with M_1 . The switching signal (lower plot) changes at this instant from 1 to 2 , so controller MPC2 has taken over.

The collision moves M_1 away from its desired position and M_2 remains joined to M_1 . Controller MPC2 adjusts the applied force (middle plot) so M_1 quickly returns to the desired position.

When the desired position changes step-wise to 5, the two masses separate briefly (with appropriate switching to MPC1) but for the most part move together and settle rapidly at the desired position. The transition back to -5 is equally well behaved.

Now suppose we force MPC2 to operate under all conditions. The figure below shows the result. When the masses are separated, as at the start, MPC2 applies excessive force and then over-compensates, resulting in oscillatory behavior. Once the masses join, the movement smooths out, as would be expected.

The oscillations are especially severe in the last transition. The masses collide frequently and M_1 never reaches the desired position.



If we put MPC1 in charge exclusively, we instead see sluggish movements that fail to settle at the desired position before the next transition occurs (not shown but you can run `mpcswitching`).

In this case, at least, two controllers are better than one.

Compute Steady-State Gain

This example shows how to analyze a model predictive controller using `cloffset`. This function computes the closed-loop, steady-state gain for each output when a sustained, 1-unit disturbance is added to each output. It assumes that no constraints are active.

Define a state-space plant model.

```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);

CSTR.InputGroup.MV = 1;
CSTR.InputGroup.UD = 2;
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(CSTR,1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Specify tuning weights for the measured output signals.

```
MPCobj.W.OutputVariables = [1 0];
```

Compute the closed-loop, steady-state gain for this controller.

```
DCgain = cloffset(MPCobj)

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #2 is integrated white noise.
-->Assuming output disturbance added to measured output channel #1 is integrated white
    Assuming no disturbance added to measured output channel #2.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

```
DCgain =
```

```
0.0000    -0.0000
2.3272    1.0000
```

`DCgain(i,j)` represents the gain from the sustained, 1-unit disturbance on output *j* to measured output *i*.

The second column of `DCgain` shows that the controller does not react to a disturbance applied to the second output. This disturbance is ignored because the tuning weight for this channel is 0.

Since the tuning weight for the first output is nonzero, the controller reacts when a disturbance is applied to this output, removing the effect of the disturbance (`DCgain(1,1) = 0`). However, since the tuning weight for the second output is 0, this controller reaction introduces a gain for output 2 (`DCgain(2,1) = 2.3272`).

See Also

`cloffset` | `mpc`

More About

- “MPC Modeling” on page 2-2

Extract Controller

This example shows how to obtain an LTI representation of an unconstrained MPC controller using `ss`. You can use this to analyze the frequency response and performance of the controller.

Define a plant model. For this example, use the CSTR model described in “Design Controller Using MPC Designer”.

```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);

CSTR.InputGroup.MV = 1;
CSTR.InputGroup.UD = 2;
CSTR.OutputGroup.MO = 1;
CSTR.OutputGroup.UO = 2;
```

Create an MPC controller for the defined plant using the same sample time, prediction horizon, and tuning weights described in “Design MPC Controller at the Command Line”.

```
MPCobj = mpc(CSTR,1,15);
MPCobj.W.ManipulatedVariablesRate = 0.3;
MPCobj.W.OutputVariables = [1 0];

-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Extract the LTI state-space representation of the controller.

```
MPCss = ss(MPCobj);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #2 is integrated white noise.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Convert the original CSTR model to discrete form using the same sample time as the MPC controller.

```
CSTRd = c2d(CSTR,MPCss.Ts);
```

Create an LTI model of the closed-loop system using feedback. Use the manipulated variable and measured output for feedback, indicating a positive feedback loop. Using negative feedback would lead to an unstable closed-loop system, because the MPC controller is designed to use positive feedback.

```
CLsys = feedback(CSTRd,MPCss,1,1,1);
```

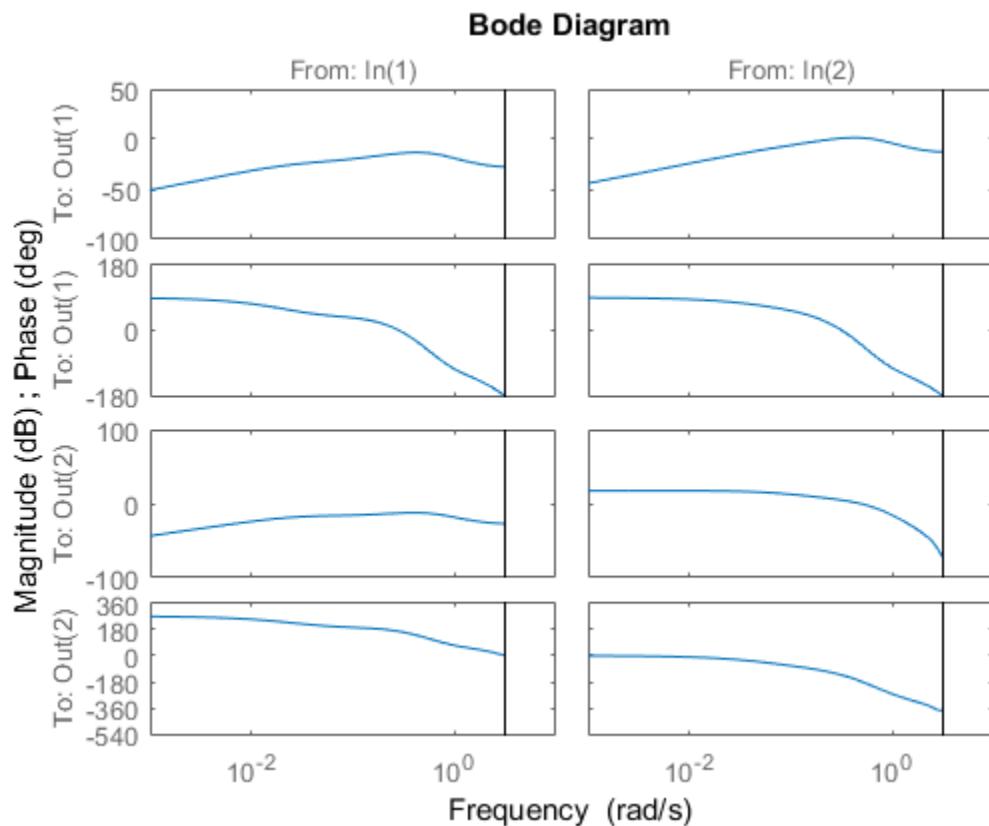
You can then analyze the resulting feedback system. For example, verify that all closed-loop poles are within the unit circle.

```
poles = eig(CLsys)
```

```
poles =  
0.5513 + 0.2700i  
0.5513 - 0.2700i  
0.6131 + 0.1110i  
0.6131 - 0.1110i  
0.9738 + 0.0000i  
0.9359 + 0.0000i
```

You can also view the system frequency response.

```
bode(CLsys)
```



See Also

[feedback](#) | [mpc](#) | [ss](#)

Related Examples

- “Design MPC Controller at the Command Line” on page 4-2

Signal Previewing

By default, a model predictive controller assumes that the current reference and measured disturbance signals remain constant during the controller prediction horizon. By doing so, the controller emulates a conventional feedback controller.

However, as shown in “Optimization Problem”, these signals can vary within the prediction horizon. If your application allows you to anticipate trends in such signals, you can use signal previewing with an MPC controller to improve reference tracking, measured disturbance rejection, or both.

The following Model Predictive Control Toolbox commands provide previewing options:

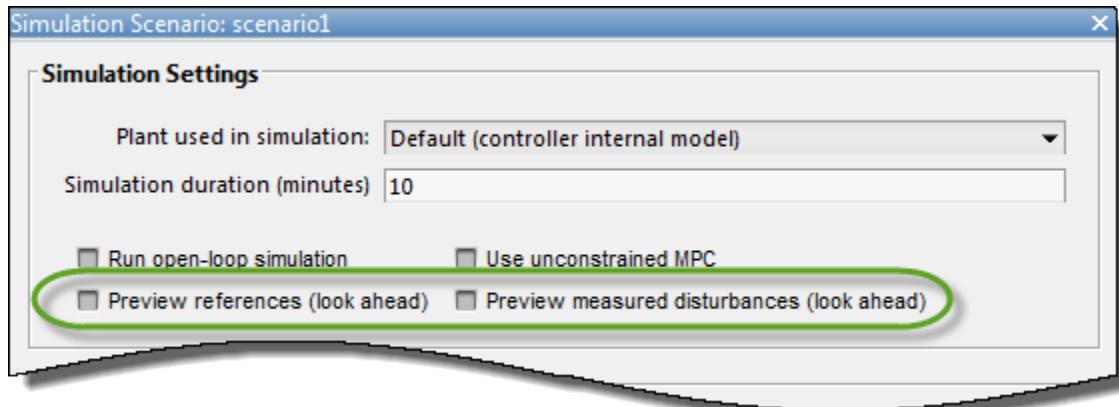
- `sim`
- `mpcmove`
- `mpcmoveAdaptive`

For Simulink, the following blocks support previewing:

- **MPC Controller**
- **Adaptive MPC Controller**
- **Multiple MPC Controllers**

Previewing for explicit MPC controllers will be supported in a future release.

In the MPC Designer app, you can specify whether simulation scenarios use previewing. When editing a scenario in the Simulation Scenario dialog box, select the **Preview references** or **Preview measured disturbances** options.



Related Examples

- Improving Control Performance with Look-Ahead (Previewing)

More About

- “Run-Time Constraint Updating” on page 4-39

Run-Time Constraint Updating

Constraint bounds can change during controller operation. The `mpcmove`, `mpcmoveAdaptive`, and `mpcmoveExplicit` commands allow for this possibility. In Simulink, all the MPC Controller blocks allow for it as well. Both of these simulation approaches give you the option to update all specified bounds at each control interval. You cannot constrain a variable for which the corresponding controller object property is unbounded. If you already defined time-varying constraints in the `mpc` controller object, the new value is applied to the first finite value in the prediction horizon. All other prediction horizon values adjust to maintain the same profile, that is they update by the same amount.

If your controller uses custom constraints on linear combinations of inputs and outputs, you can update these constraints at each simulation iteration by calling `setconstraint` before `mpcmove`. However, updating the custom constraint matrices at run time is not supported in Simulink. To deploy an MPC controller with run-time updating of custom constraints, use MATLAB Compiler™ to generate the executable code, and deploy it using the MATLAB Runtime. In this case, the controller sample time must be large, since run-time MPC regeneration is slow.

See Also

`mpcmove` | `mpcmoveAdaptive` | `mpcmoveExplicit` | `setconstraint`

Related Examples

- Varying Input and Output Constraints

More About

- “Run-Time Weight Tuning” on page 4-40
- “Constraints on Linear Combinations of Inputs and Outputs”

Run-Time Weight Tuning

There are two ways to perform tuning experiments using Model Predictive Control Toolbox software:

- Modify your controller object off line (by changing weights, etc.) and then test the modified object.
- Change tuning weights as the controller operates.

This topic describes the second approach.

You can adjust the following tuning weights as the controller operates (see “Tuning Weights”):

- Plant output variable (OV) reference tracking, w^y .
- Manipulated variable (MV) reference tracking, w^u .
- MV increment suppression, $w^{\Delta u}$.
- Global constraint softening, ρ_ϵ .

In each case, the weight applies to the entire prediction horizon. If you are using time-varying weights, you must use offline modification of the weight and then test the modified controller.

In Simulink, the following blocks support online (run-time) tuning:

- **MPC Controller**
- **Adaptive MPC Controller**
- **Multiple MPC Controllers.** In this case, the tuning signals apply to the active controller object, which might switch as the control system operates. If the objects in your set employ different weights, you should tune them off line.

The **Explicit MPC Controller** block cannot support online tuning because a weight change requires a complete revision of the explicit MPC control law, which is computationally intensive.

For command-line testing, the `mpcmove` and `mpcmoveAdaptive` commands include options to mimic the online tuning behavior available in Simulink.

Related Examples

- Tuning Controller Weights

More About

- “Signal Previewing” on page 4-37

Designing and Testing Controllers in Simulink

- “Design MPC Controller in Simulink” on page 5-2
- “Test an Existing Controller” on page 5-23
- “Schedule Controllers at Multiple Operating Points” on page 5-27

The Model Predictive Control Toolbox provides a controller block library for use in Simulink.

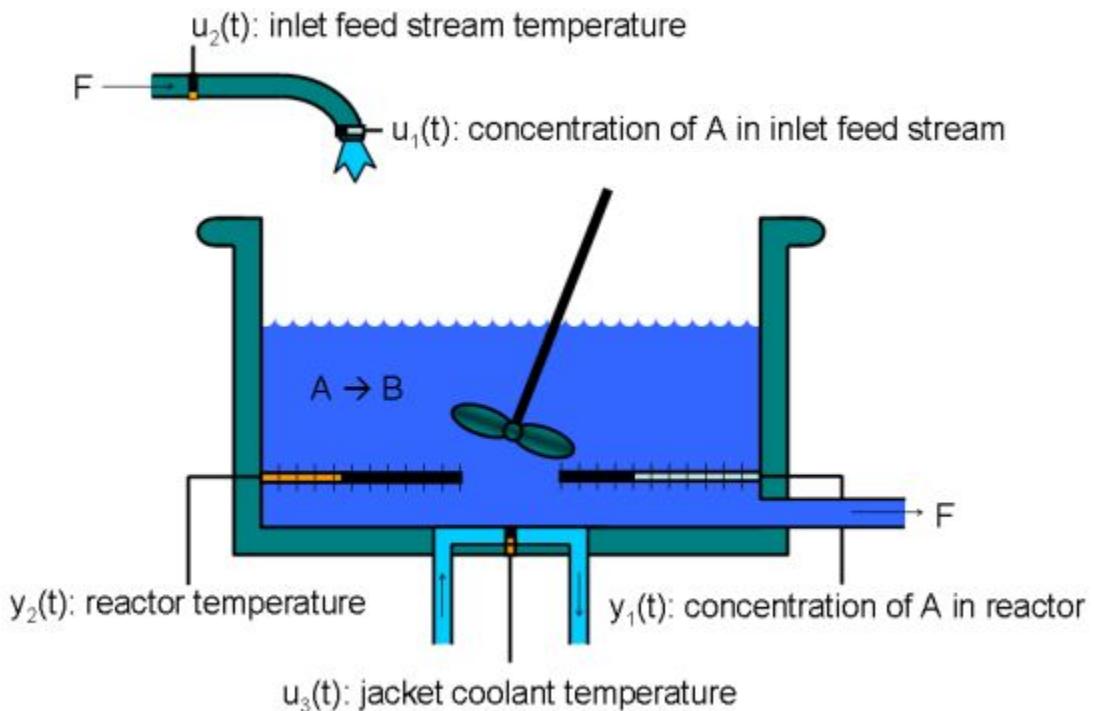
Design MPC Controller in Simulink

This example shows how to design a model predictive controller for a continuous stirred-tank reactor (CSTR) in Simulink using the MPC Designer app.

This example requires Simulink Control Design software to define the MPC structure by linearizing a nonlinear Simulink model.

CSTR Model

A CSTR is a jacketed nonadiabatic tank reactor commonly used in the process industry.



An inlet stream of reagent A feeds into the tank at a constant rate. A first-order, irreversible, exothermic reaction takes place to produce the product stream, which exits the reactor at the same rate as the input stream.

The CSTR model has three inputs:

- Feed Concentration (CA_i) — The concentration of reagent A in the feed stream (kgmol/m^3)
- Feed Temperature (T_i) — Feed stream temperature (K)
- Coolant Temperature (T_c) — Reactor coolant temperature (K)

The two model outputs are:

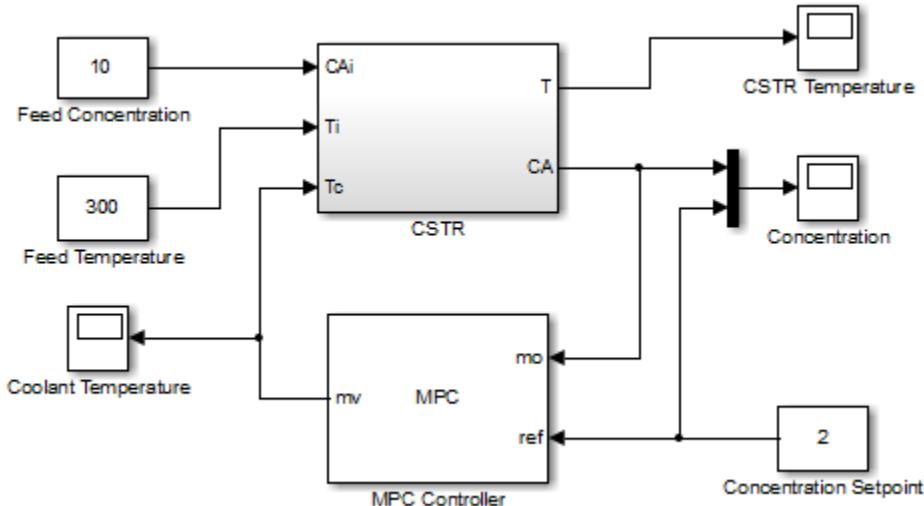
- CSTR Temperature (T) — Reactor temperature (K)
- Concentration (CA) — Concentration of reagent A in the product stream, also referred to as the *residual concentration* (kgmol/m^3)

The control objective is to maintain the residual concentration, CA , at its nominal setpoint by adjusting the coolant temperature, T_c . Changes in the feed concentration, CA_i , and feed temperature, T_i , cause disturbances in the CSTR reaction.

The reactor temperature, T , is usually controlled. However, for this example, ignore the reactor temperature, and assume that the residual concentration is measured directly.

Open Simulink Model

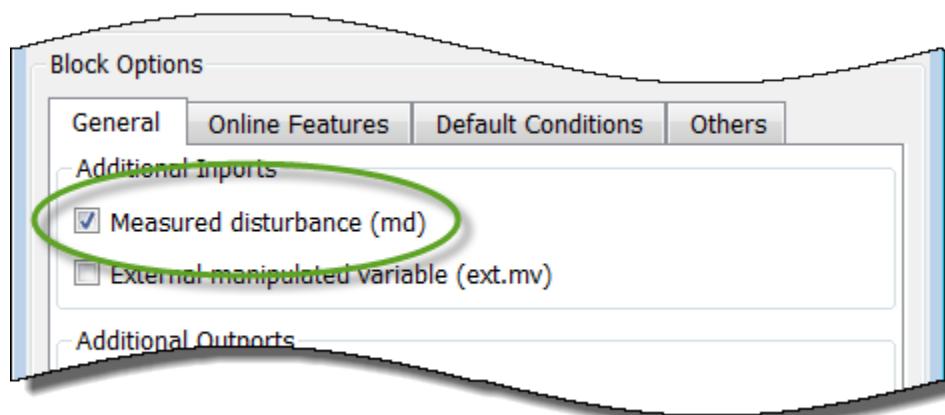
```
open_system( 'CSTR_ClosedLoop' );
```



Connect Measured Disturbance To MPC Controller Block

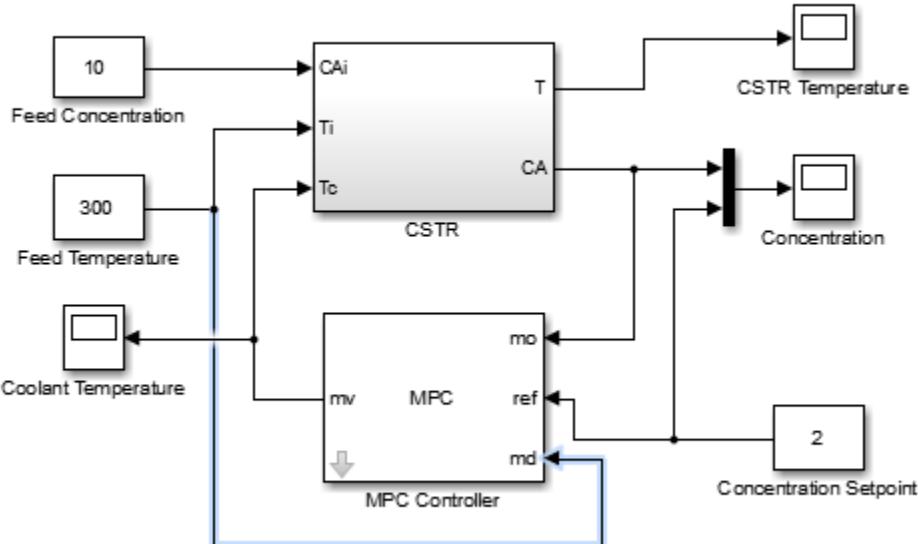
In the Simulink model window, double-click the MPC Controller block.

In the Block Parameters dialog box, on the **General** tab, in the **Additional Imports** section, check the **Measured disturbance (md)** option.



Click **Apply** to add the **md** import to the controller block.

In the Simulink model window, connect the **Feed Temperature** block output to the **md** import.



Open MPC Designer App

In the MPC Controller Block Parameters dialog box, click **Design** to open MPC Designer.

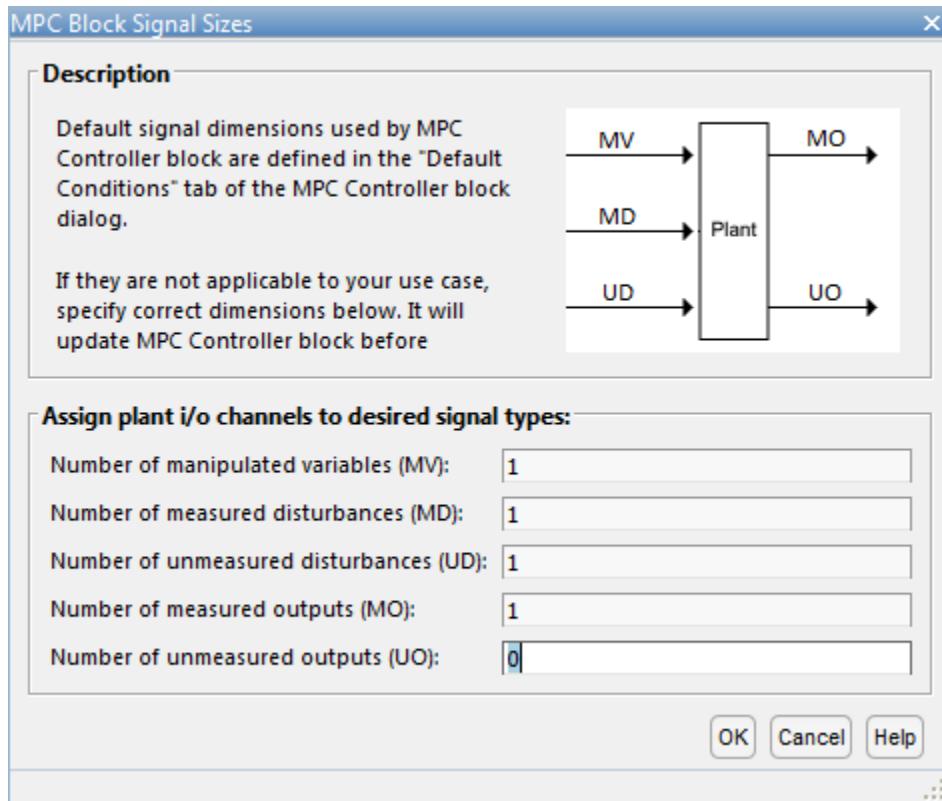
Define MPC Structure

In the MPC Designer app, on the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

In the Define MPC Structure By Linearization dialog box, in the **Controller Sample Time** section, specify a sample time of **0.1**.

In the **MPC Structure** section, click **Change I/O Sizes** to add the unmeasured disturbance and measured disturbance signal dimensions.

In the MPC Block Signal Sizes dialog box, specify the number of input/output channels of each type:



Click **OK**.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Signals for Plant Inputs** section, the app adds a row for **Unmeasured Disturbances (UD)**.

Define MPC Structure By Linearization

MPC Structure

```

graph LR
    Setpoints[Setpoints  
(reference)] --> MPC[MPC]
    UD[Unmeasured Disturbances] --> MPC
    MPC -- Manipulated Variables --> Plant[Plant]
    MPC -- Measured Disturbances --> Plant
    Plant -- Unmeasured --> Out[Outputs]
    Plant -- Measured --> Out
    Inputs[Inputs] -- Unmeasured Disturbances --> Plant
  
```

Controller Sample Time
Specify MPC controller sample time (default sample time in the MPC block):

Simulink Operating Point
Choose an operating point at which plant model is linearized and nominal values are computed:

Simulink Signals for Plant Inputs

Selected	Type	Block Path
<input checked="" type="radio"/>	Manipulated Variables (MV)	CSTR_ClosedLoop/MPC_Controller:1
<input type="radio"/>	Measured Disturbances (MD)	CSTR_ClosedLoop/Feed Temperature:1
<input type="radio"/>	Unmeasured Disturbances (UD)	Select a UD signal in the Simulink model

Select Signals

Simulink Signals for Plant Outputs

Selected	Type	Block Path
<input checked="" type="radio"/>	Measured Outputs (MO)	CSTR_ClosedLoop/CSTR:2

Select Signals

Buttons

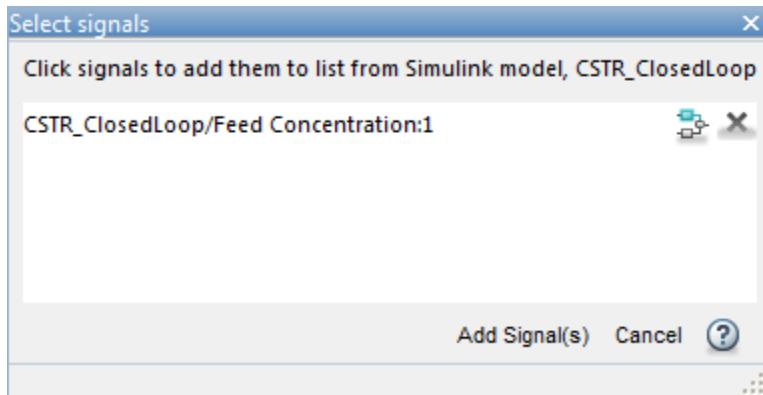
-
-
-

The manipulated variable, measured disturbance, and measured output are already assigned to their respective Simulink signal lines, which are connected to the MPC Controller block.

In the **Simulink Signals for Plant Inputs** section, select the **Unmeasured Disturbances (UD)** row, and click **Select Signals**.

In the Simulink model window, click the output signal from the **Feed Concentration** block.

The signal is highlighted and its block path is added to the Select Signal dialog box.



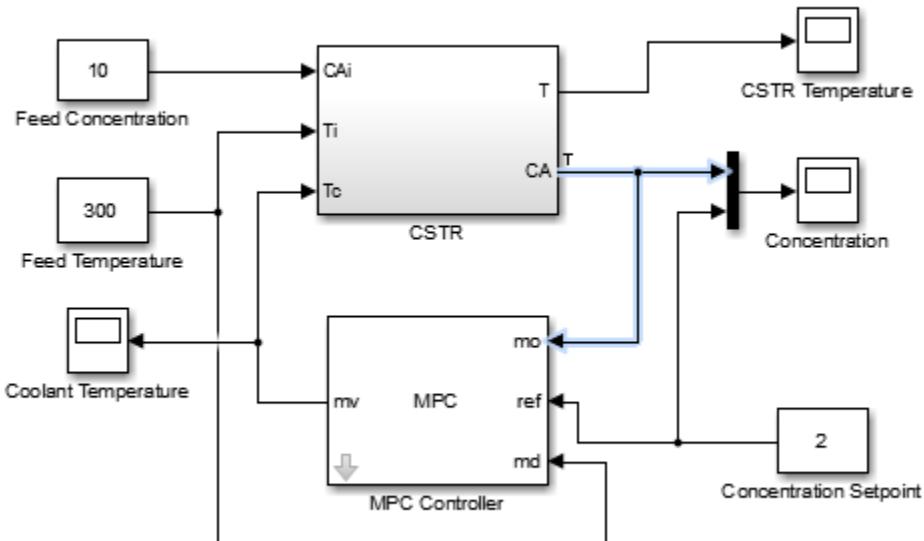
In the Select Signals dialog box, click **Add Signal(s)**.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Signals for Plant Inputs** table, the **Block Path** for the unmeasured disturbance signal is updated.

Linearize Simulink Model

Linearize the Simulink model at a steady-state equilibrium operating point where the residual concentration is 2 kgmol/m^3 . To compute such an operating point, add the CA signal as a trim output constraint, and specify its target constraint value.

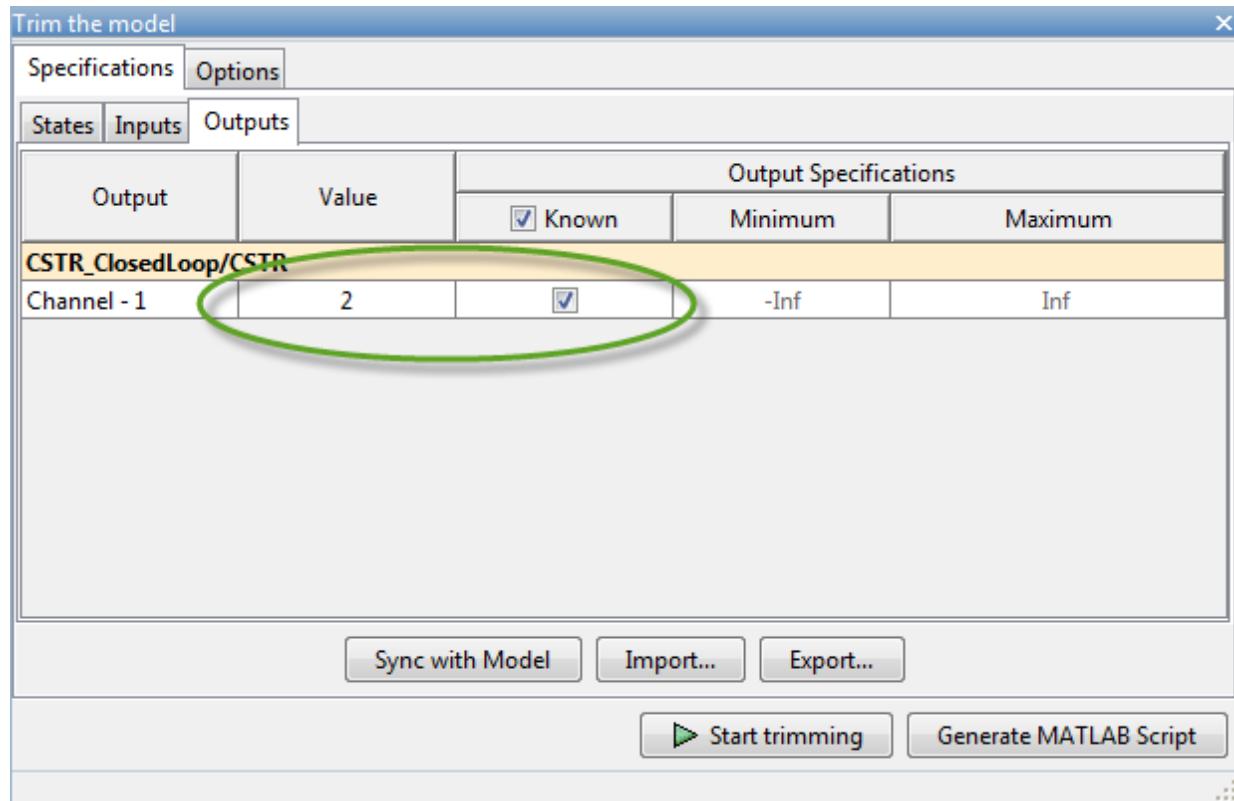
In the Simulink model window, right-click the signal line connected to CA outport of the CSTR block, and click **Linear Analysis Points > Trim Output Constraint**.



The CA signal can now be used to define output specifications for calculating a model steady-state operating point.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Operating Point** section, in the drop-down list, under **Create New Operating Point**, click **Trim Model**.

In the Trim the model dialog box, in the **Outputs** tab, check the box in the **Known** column for **Channel-1** and specify a **Value** of 2.



This setting constrains the value of the output signal during the operating point search to a known value.

Click **Start Trimming**.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Operating Point** section, the computed operating point, `op_trim1`, is added to the drop-down list and selected.

In the drop-down list, under **View/Edit**, click **Edit op_trim1**.

Edit: op_trim1

Optimizer Output **Details**

State **Input** **Output**

State	Desired Value	Actual Value	Desired dx	Actual dx
CSTR_ClosedLoop/CSTR/C_A				
State - 1	[0 , Inf]	2	0	-4.5972e-12
CSTR_ClosedLoop/CSTR/T_K				
State - 1	[0 , Inf]	373.1311	0	5.4897e-11
CSTR_ClosedLoop/MPC Controller/MPC/last_mv				
State - 1	[-Inf , Inf]	299.0349	0	0

Initialize model...

In the Edit dialog box, on the **State** tab, in the **Actual dx** column, the near-zero derivative values indicate that the computed operating point is at steady-state.

Click **Initialize model** to set the initial states of the Simulink model to the operating point values in the **Actual Values** column. Doing so enables you to later simulate the Simulink model at the computed operating point rather than at the default model initial conditions.

In the Initialize Model dialog box, click **OK**.

Close the Edit dialog box.

In the Define MPC Structure By Linearization dialog box, click **Define and Linearize** to linearize the model.

The linearized plant model is added to the MPC Designer **Data Browser**. Also, the following are added to the **Data Browser**:

- A default MPC controller created using the linearized plant as an internal prediction model

- A default simulation scenario

Define Input/Output Channel Attributes

On the **MPC Designer** tab, in the Structure section, click **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, in the **Name** column, specify meaningful names for each input and output channel.

In the **Unit** column, specify appropriate units for each signal.

Input and Output Channel Specifications					
Plant Inputs					
Channel	Type	Name	Unit	Nominal Value	Scale Factor
u(1)	MV	Tc	deg K	299.034880275325	1
u(2)	MD	Ti	deg K	300	1
u(3)	UD	CAi	kgmol/m^3	0	1

Plant Outputs					
Channel	Type	Name	Unit	Nominal Value	Scale Factor
y(1)	MO	CA	kgmol/m^3	2	1

The **Nominal Value** for each signal is the corresponding steady-state value at the computed operating point.

Click **OK**.

Define Disturbance Rejection Simulation Scenarios

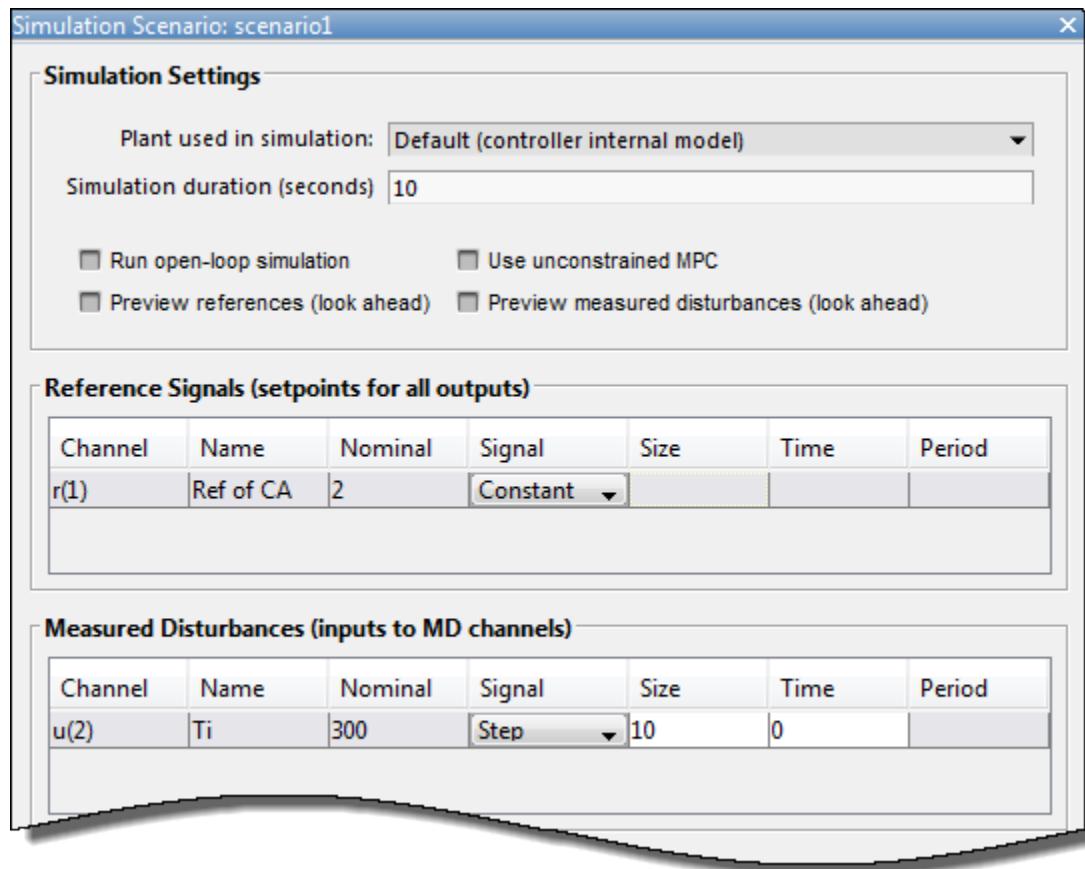
The primary objective of the controller is to hold the residual concentration, CA , at the nominal value of 2 kgmol/m^3 . To do so, the controller must reject both measured and unmeasured disturbances.

In the **Scenario** section, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, in the **Reference Signals** table, in the **Signal** drop-down list select **Constant** to hold the output setpoint at its nominal value.

In the **Measured Disturbances** table, in the **Signal** drop-down list, select **Step**.

Specify a step **Size** of 10 and a step **Time** of 0.



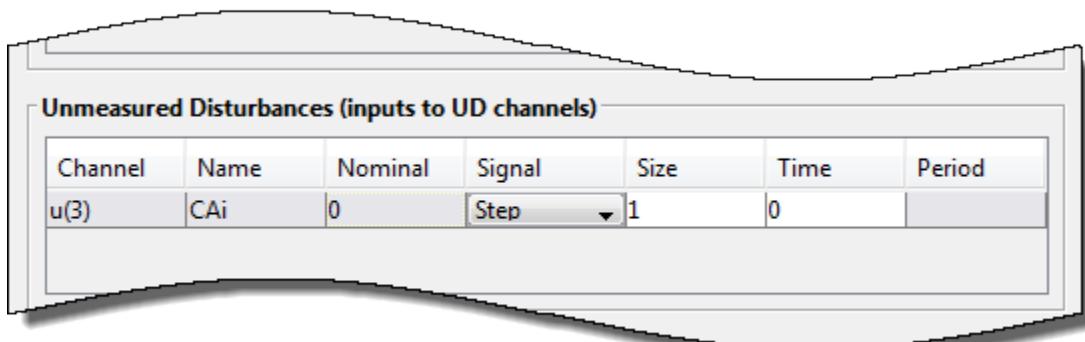
Click **OK**.

In the **Data Browser**, under **Scenarios**, double-click **scenario1** and rename it **MD_reject**.

In the **Scenario** section, click **Plot Scenario > New Scenario**.

In the Simulation Scenario dialog box, in the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step**.

Specify a step **Size** of 1 and a step **Time** of 0.



Click **OK**.

In the **Data Browser**, under **Scenarios**, double-click **NewScenario** and rename it **UD_reject**.

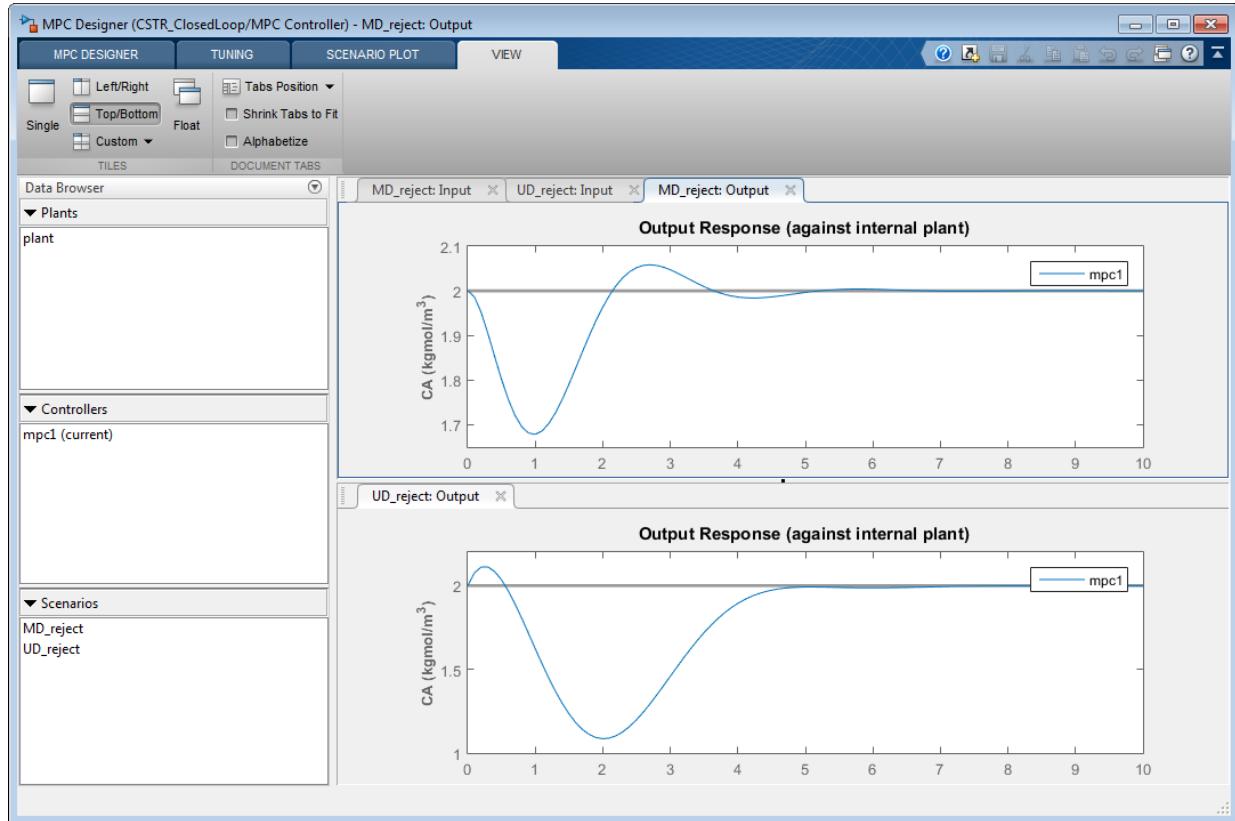
Arrange Output Response Plots

To make viewing the tuning results easier, arrange the plot area to display the Output Response plots for both scenarios at the same time.

On the **View** tab, in the **Tiles** section, click **Top/Bottom**.

The plot display area changes to display the Input Response plots above the Output Response plots.

Drag the **MD_reject: Output** tab up to the top plot.



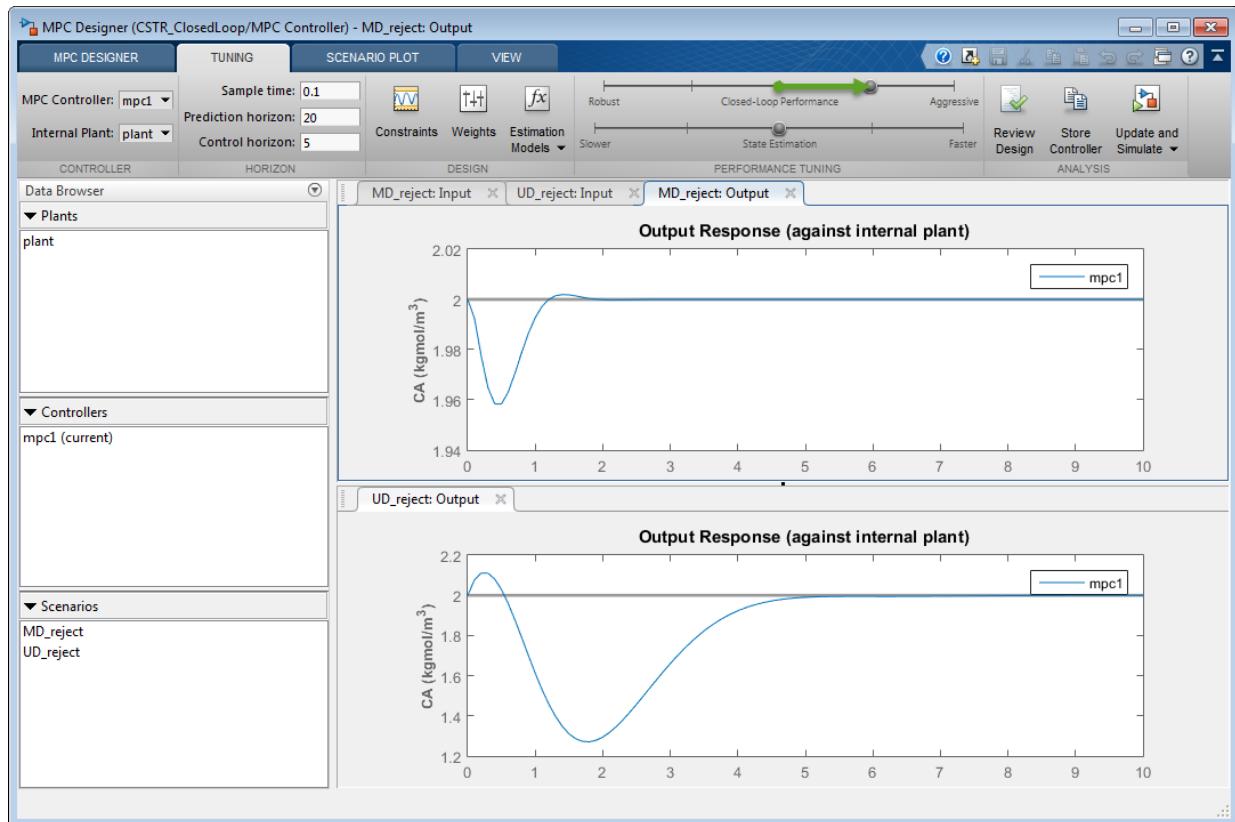
Tune Controller Performance

In the **Tuning** tab, in the **Horizon** section, specify a **Prediction horizon** of 20 and a **Control horizon** of 5.

The **Output Response** plots update based on the new horizon values.

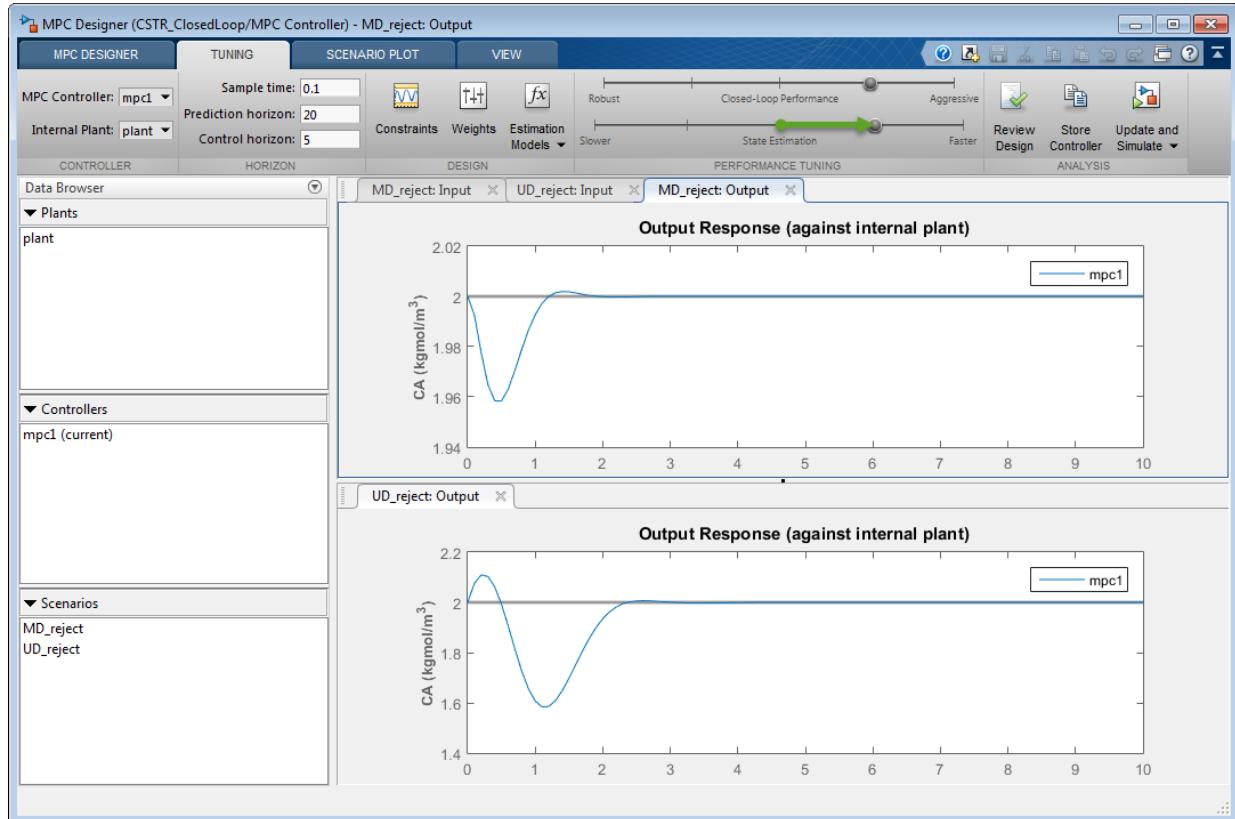
Use the default controller constraint and weight configurations.

In the **Performance Tuning** section, drag the **Closed-Loop Performance** slider to the right, which leads to tighter control of outputs and more aggressive control moves. Drag the slider until the **MD_reject: Output** response reaches steady state in less than 2 seconds.



Drag the **State Estimation** slider to the right, which leads to more aggressive unmeasured disturbance rejection. Drag the slider until the **UD_reject: Output** response reaches steady state in less than 3 seconds.

5 Designing and Testing Controllers in Simulink



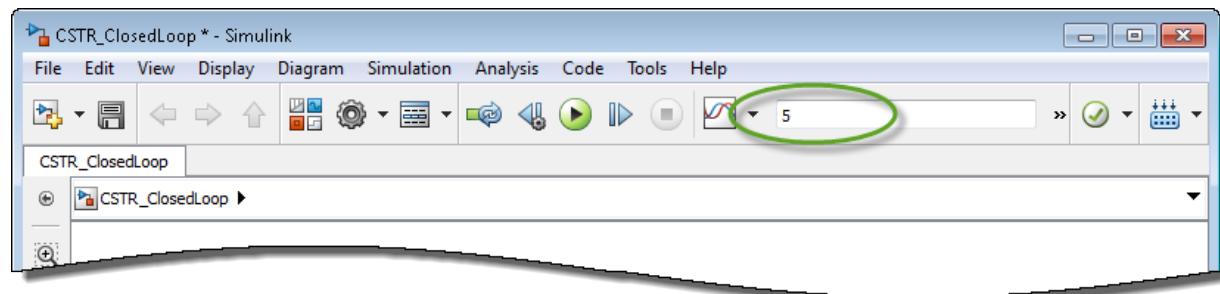
Update Simulink Model with Tuned Controller

In the Analysis section, click the **Update and Simulate** arrow ▾.

Under **Update and Simulate**, click **Update Block Only**. The app exports the tuned controller, `mpc1`, to the MATLAB workspace. In the Simulink model, the **MPC Controller** block is updated to use the exported controller.

Simulate Unmeasured Disturbance Rejection

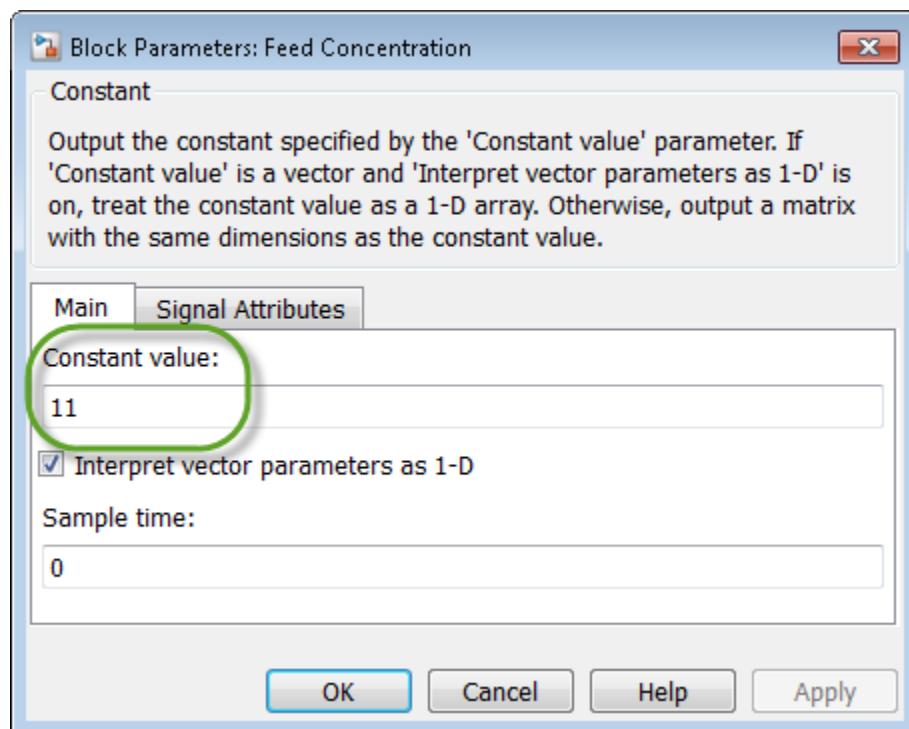
In the Simulink model window, change the simulation duration to 5 seconds.



The model initial conditions are set to the nominal operating point used for linearization.

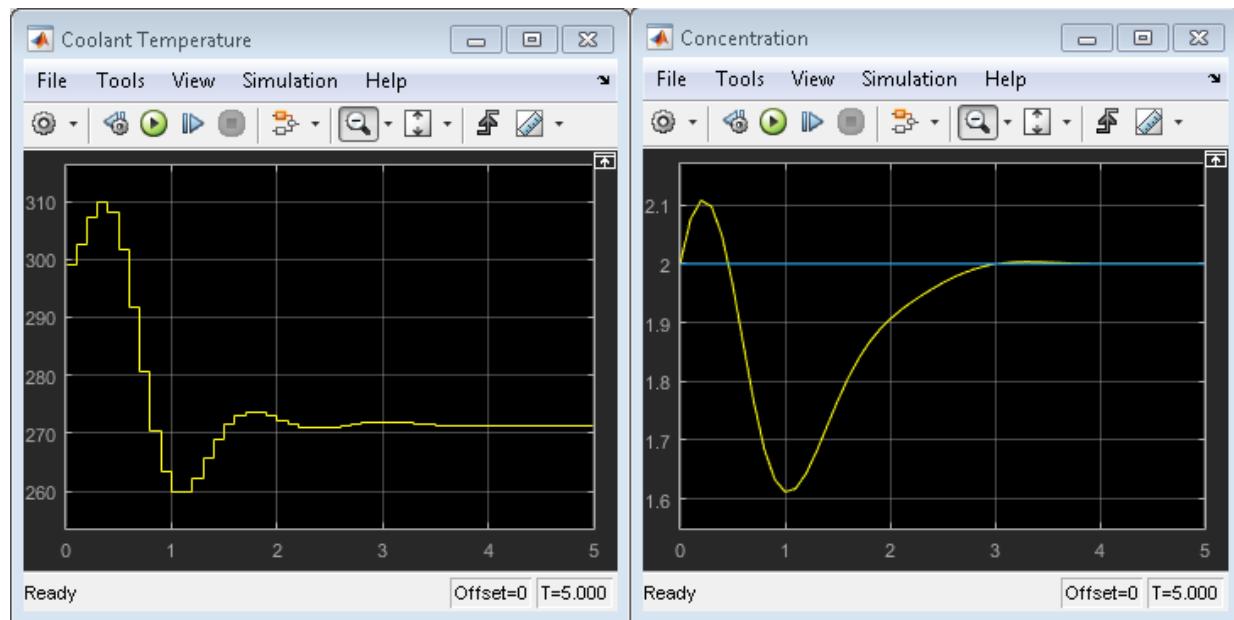
Double-click the **Feed Concentration** block.

In the Block Parameters dialog box, enter a **Constant Value** of 11 to simulate a unit step at time zero.



Click **Apply**.

In the Simulink model window, click **Run**.



The **Concentration** output response is similar to the **UD_reject** response, however the settling time is around 1 second later. The different result is due to the mismatch between the linear plant used in the MPC Designer simulation and the nonlinear plant in the Simulink model.

Simulate Measured Disturbance Rejection

In the Block Parameters: Feed Concentration dialog box, enter a **Constant Value** of 10 to return the feed concentration to its nominal value.

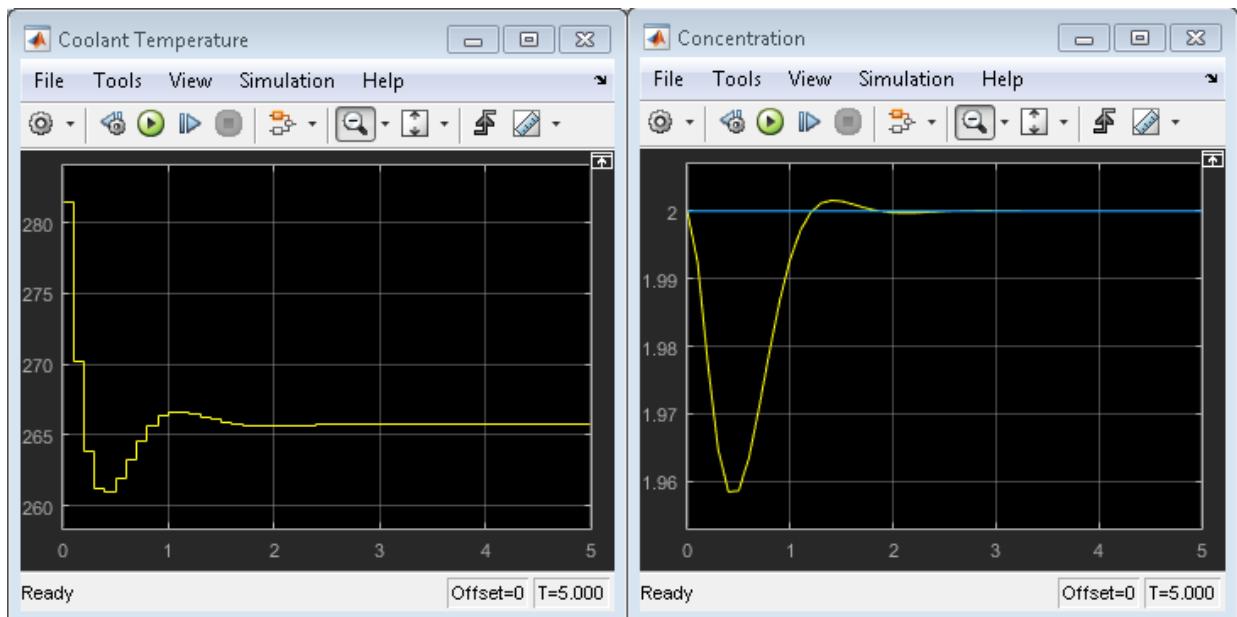
Click **OK**.

In the Simulink model window, double-click the **Feed Temperature** block.

In the Block Parameters: Feed Temperature dialog box, enter a **Constant Value** of 310 to simulate a step change of size 10 at time zero.

Click **OK**.

In the Simulink model window, click **Run**.



The **Concentration** output response is similar to the **MD_reject** response from the MPC Designer simulation.

References

- [1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004, pp. 34–36 and 94–95.

See Also

[MPC Controller](#) | [MPC Designer](#)

Related Examples

- “Linearize Simulink Models” on page 2-22
- “Design Controller Using MPC Designer” on page 3-2

More About

- “Tuning Weights”

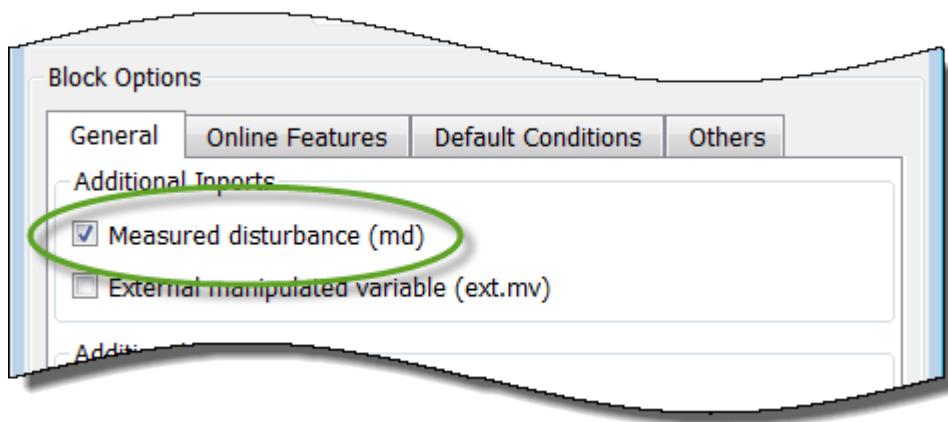
Test an Existing Controller

This topic shows how to test an existing model predictive controller by adding it to a Simulink model.

- 1 Open your Simulink model.
- 2 Add an MPC Controller block to the model.
- 3 If your controller includes measured disturbances, add the md import to the MPC Controller block.

Double-click the MPC Controller block.

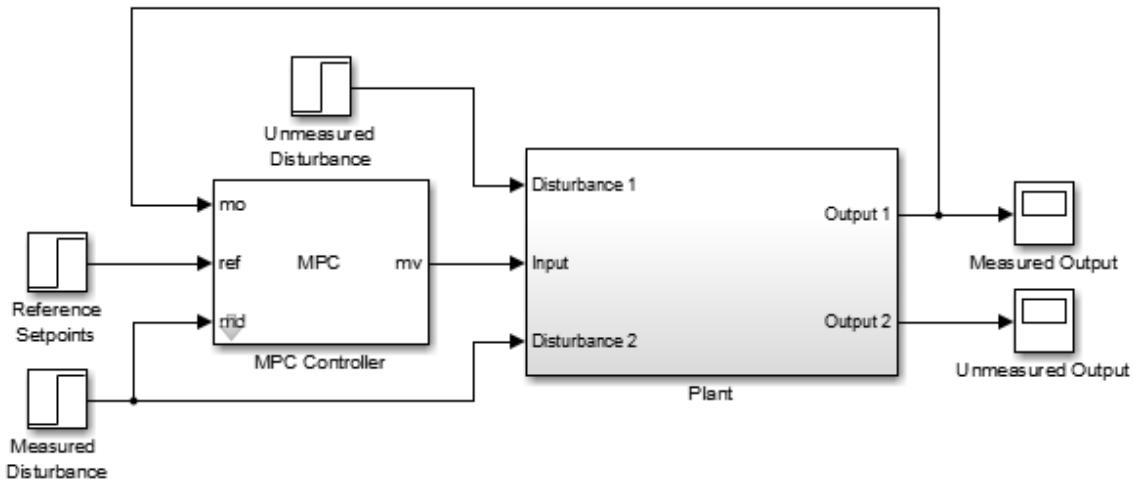
In the Block Parameters dialog box, on the **General** tab, select **Measured disturbance (md)**.



Click **OK**.

- 4 Connect the plant and controller signals in the Simulink model. Connect:
 - The plant inputs to the manipulated variable (mv) import of the MPC Controller block.
 - The plant measured outputs to the measured output (mo) import of the MPC Controller block.
 - The measured disturbances, if any, to the plant and to the measured disturbance (md) import of the MPC Controller block.

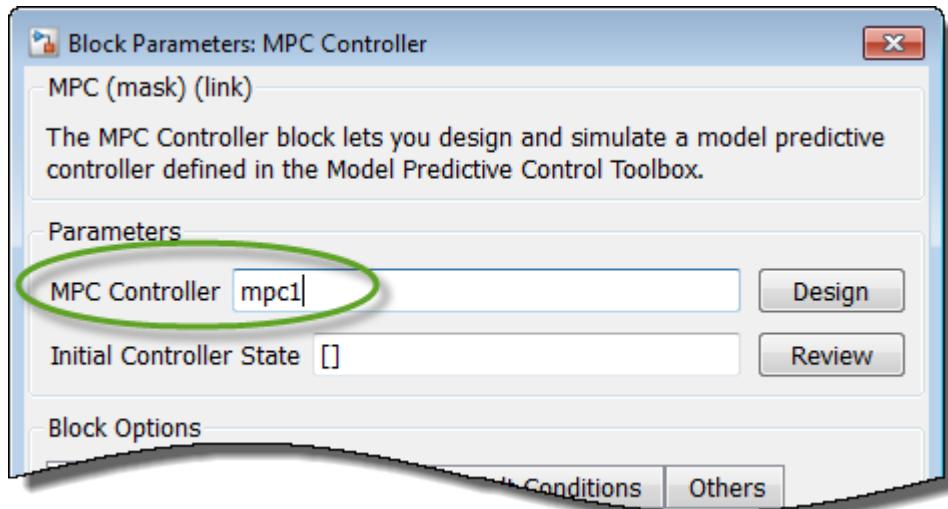
- Any unmeasured disturbances or unmeasured outputs to their corresponding plant import and outport.
- The reference signals to the reference (`ref`) import of the **MPC Controller** block.



5 Specify the controller.

Double-click the **MPC Controller** block.

In the Block Parameters dialog box, in the **MPC Controller** field, specify the name of an `mpc` controller from the MATLAB workspace.



Click **OK**.

6 (Optional) Modify the controller.

After specifying a controller in the **MPC Controller** block, you can modify the controller:

- Using the MPC Designer app:
 - In the Block Parameters dialog box, click **Design**.
 - In the MPC Designer app, tune the controller parameters.
 - In the **MPC Designer** tab, in the **Result** section, click **Update and Simulate > Update Block Only**.

The app exports the updated controller to the MATLAB workspace.

- Using commands to modify the controller object in the MATLAB workspace.

7 Run the Simulink model.

Tip If you do not have a Simulink model of your plant, you can generate one that uses your MPC controller to control its internal plant model. For more information, see “Generate Simulink Model from MPC Designer”.

See Also

[mpc](#) | [MPC Controller](#) | [MPC Designer](#)

Related Examples

- “Design MPC Controller in Simulink” on page 5-2
- “Design MPC Controller at the Command Line” on page 4-2
- “Generate Simulink Model from MPC Designer”

Schedule Controllers at Multiple Operating Points

In this section...

- “A Two-Model Plant” on page 5-27
- “Animation of the Multi-Model Example” on page 5-28
- “Designing the Two Controllers” on page 5-29
- “Simulating Controller Performance” on page 5-30

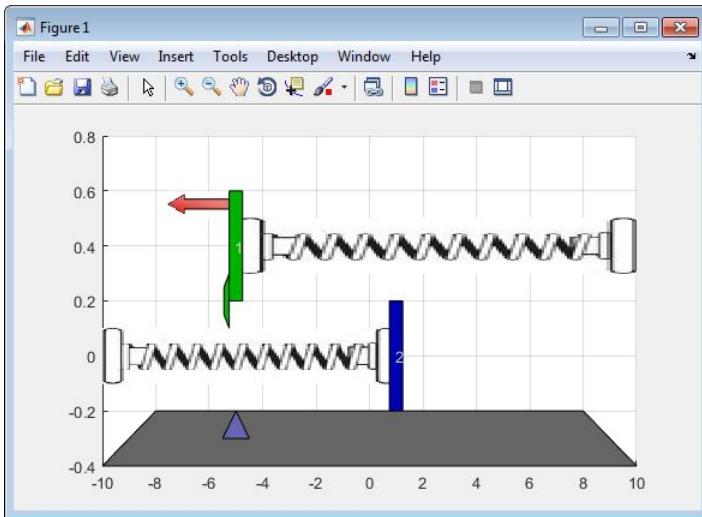
If your plant is nonlinear, a controller designed to operate in a particular target region may perform poorly in other regions. The common way to compensate is to design multiple controllers, each intended for a particular combination of operating conditions. Switch between them in real-time as conditions change. Gain scheduling is a conventional example of this technique. The following example shows how to coordinate multiple model predictive controllers for this purpose.

The rest of this section is an illustrative example organized as follows:

- “A Two-Model Plant” on page 5-27 describes the example application involving two distinct operating conditions.
- “Animation of the Multi-Model Example” on page 5-28 explains how to simulate the plant.
- “Designing the Two Controllers” on page 5-29 shows how to obtain the controllers for each operating condition.
- “Simulating Controller Performance” on page 5-30 shows how to use the **Multiple MPC Controllers** block in Simulink to coordinate the two controllers.

A Two-Model Plant

The figure below is a stop-action snapshot of the plant. It consists of two masses, M1 (upper) and M2 (lower). A spring connects M1 to a rigid wall and pulls it to the right. An applied force, shown as a red arrow, opposes the spring pulling M1 to the left.



In the above, mass M2 is uncontrollable. It responds solely to the spring pulling it to the left.

If the two masses collide, however, they stick together until a change in the applied force separates them.

The control objective is to make M1 track a reference signal. The blue triangle in the above indicates the desired location, which varies with time. At the instant shown, the desired numeric location is -5.

For an animated version of the plant, see [Scheduling Controllers for a Plant with Multiple Operating Points](#).

Animation of the Multi-Model Example

In order to achieve its objective, the controller can adjust the applied force magnitude (the length of the red arrow). It receives continuous measurements of the M1 location. There is also a contact sensor to signal collisions. The M2 location is unmeasured.

If M1 were isolated, this would be a routine control problem. The challenge is that the relationship between the applied force and M1's movement changes dramatically when M2 attaches to M1.

Define the model.

Define the system parameters.

```
M1 = 1; % mass
M2 = 5; % mass
k1 = 1; % spring constant
k2 = 0.1; % spring constant
b1 = 0.3; % friction coefficient
b2 = 0.8; % friction coefficient
y eq1 = 10; % wall mount position
y eq2 = -10; % wall mount position
```

Define a model of M1 when the masses are separated. Its states are the position and velocity of M1. Its inputs are the applied force, which will be the controller's manipulated variable (MV), and a spring constant calibration signal, which a measured disturbance (MD) input.

```
A1 = [0 1;-k1/M1 -b1/M1];
B1 = [0 0;-1/M1 k1*y eq1/M1];
C1 = [1 0];
D1 = [0 0];
sys1 = ss(A1,B1,C1,D1);
sys1 = setmpcsignals(sys1, MV ,1, MD ,2);
```

The call to `setmpcsignals` specifies the input type for the two inputs.

Define another model (with the same input/output structure) to predict movement when the two masses are joined.

```
A2 = [0 1;-(k1+k2)/(M1+M2) -(b1+b2)/(M1+M2)];
B2 = [0 0;-1/(M1+M2) (k1*y eq1+k2*y eq2)/(M1+M2)];
C2 = [1 0];
D2 = [0 0];
sys2 = ss(A2,B2,C2,D2);
sys2 = setmpcsignals(sys2, MV ,1, MD ,2);
```

Designing the Two Controllers

Next, define controllers for each case. Both controllers employ a 0.2 second sampling period (T_s), a prediction horizon of $p = 20$, a control horizon of $m = 1$ and default values for all other controller parameters.

The only property that distinguishes the two controllers is the prediction model.

```

Ts = 0.2;
p = 20;
m = 1;
MPC1 = mpc(sys1,Ts,p,m); % Controller for M1 detached from M2
MPC2 = mpc(sys2,Ts,p,m); % Controller for M1 connected to M2

```

The applied force also have the same constraints in each case. The lower bound for the applied force is zero because it cannot reverse direction. The maximum rate of change for the applied force is 1000 per second (increasing or decreasing).

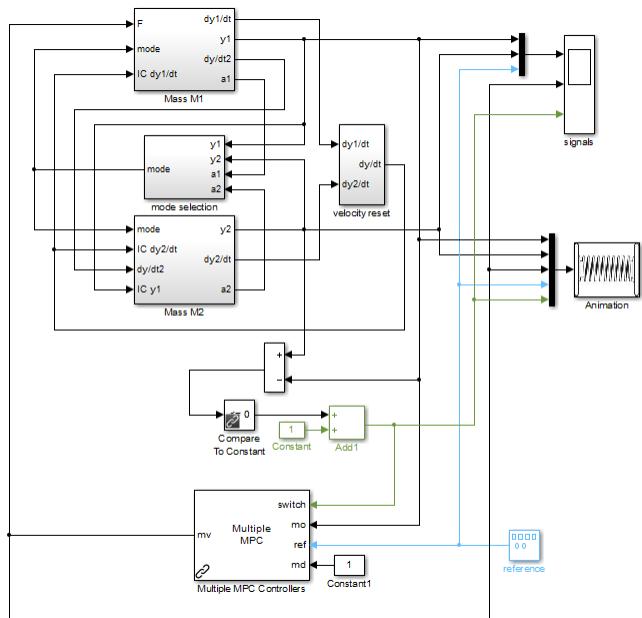
```

MPC1.MV = struct( Min ,0, RateMin ,-1e3, RateMax ,1e3);
MPC2.MV = struct( Min ,0, RateMin ,-1e3, RateMax ,1e3);

```

Simulating Controller Performance

The following is the Simulink block diagram for this example.

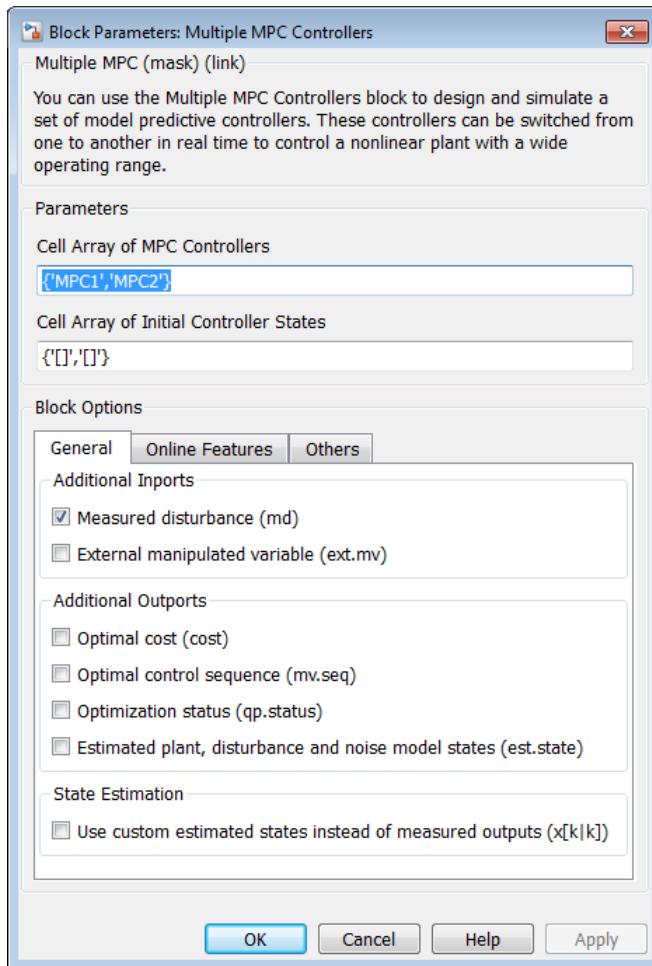


The upper portion simulates the movement of the two masses, plots the signals as a function of time and animates the plant.

The lower part contains the following key elements:

- A pulse generator that supplies the desired M1 position (the controller reference signal). The output of the pulse generator is a square wave that varies between —5 and 5, with a frequency of 0.015 per second.
- Simulation of a contact sensor. When the two masses have the same position, the **Compare to Constant** block evaluates to **true** and the **Add1** block converts this to a 2. Else, the output of **Add1** is 1.
- The **Multiple MPC Controllers** block, which has four inputs. The measured output (**mo**), reference (**ref**) and measured disturbance (**md**) inputs are as for the **MPC Controller** block. The distinctive feature of the **Multiple MPC Controllers** block is the **switch** input.

The figure below shows the **Multiple MPC Controllers** block mask for this example.



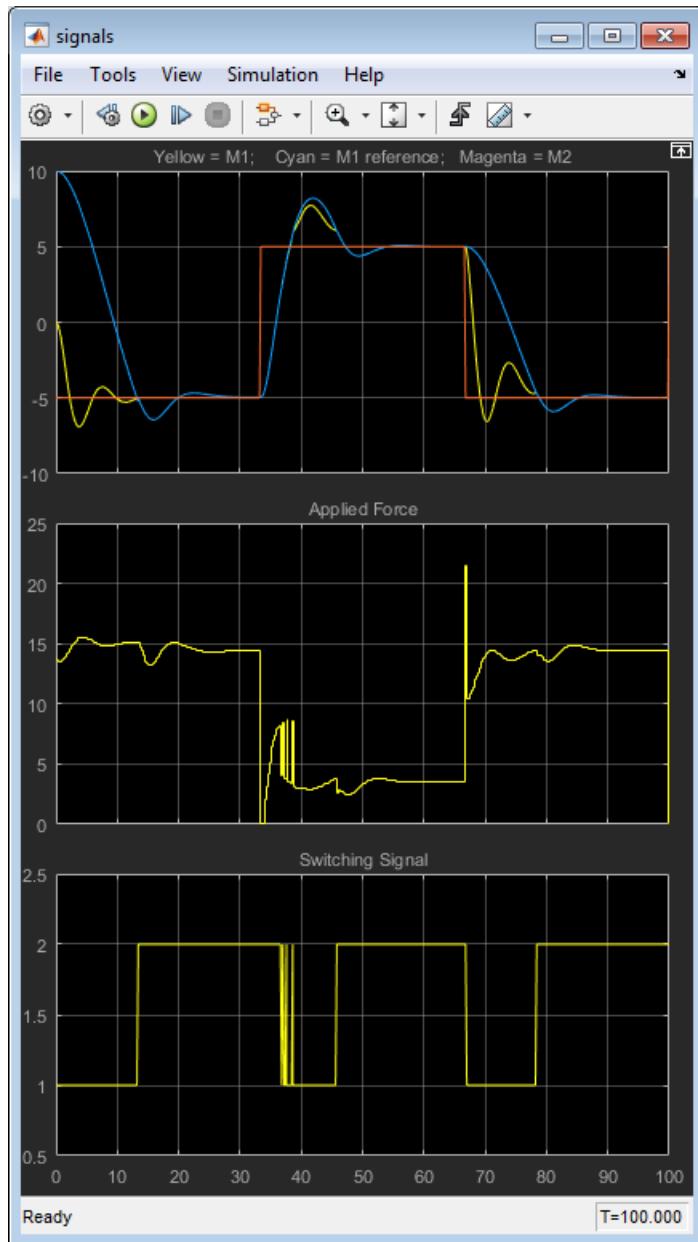
When the **switch** input is 1, the block automatically activates the first controller that is listed (**MPC1**), which is appropriate when the masses are separated. When the **switch** input is 2, the block automatically enables the second controller (**MPC2**).

Simulate the controller performance using Simulink commands.

```
Tstop = 100; % Simulation time  
y1initial = 0; % Initial M1 and M2 Positions
```

```
y2initial = 10;  
open( mpc_switching )  
sim( mpc_switching ,Tstop)
```

The figure below shows the **signals** scope output.



In the upper plot, the cyan curve is the desired position. It starts at -5. The M1 position (yellow) starts at 0. Under the control of MPC1, M1 moves rapidly towards the desired position. M2 (magenta) starts at 10 and begins moving in the same direction.

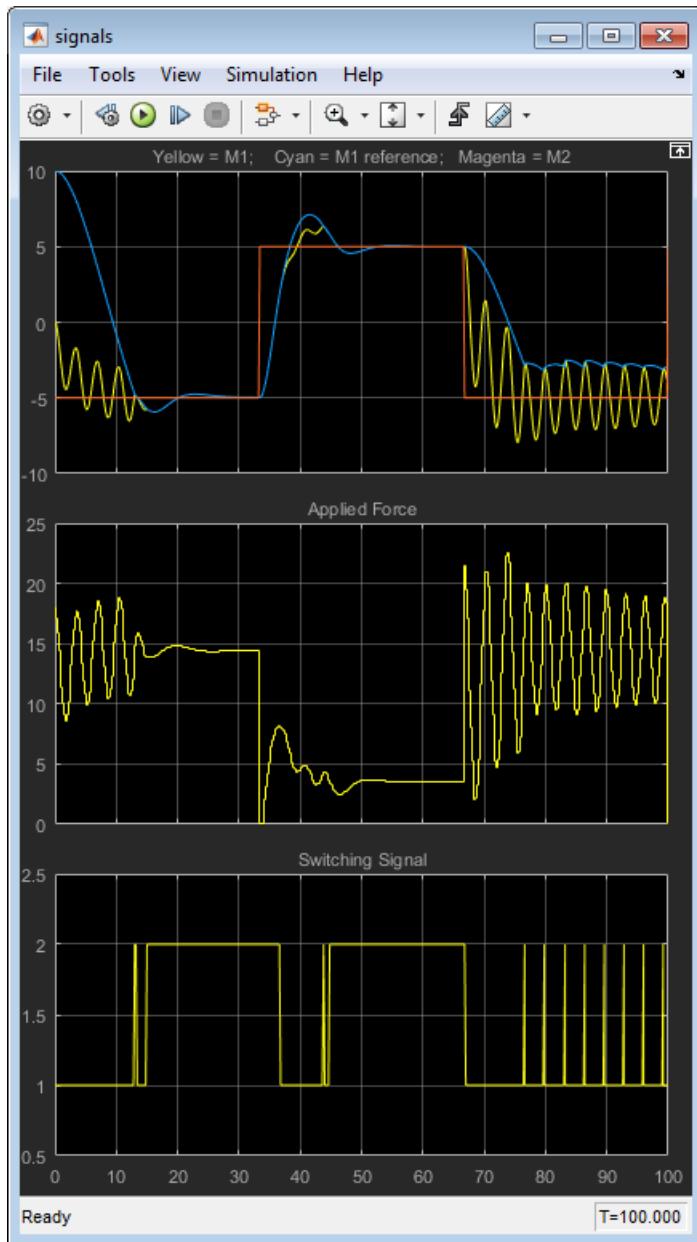
At about $t = 13$ seconds, M2 collides with M1. The switching signal (lower plot) changes at this instant from 1 to 2, so MPC2 takes over.

The completely inelastic collision moves M1 away from its desired position and M2 remains joined to M1. Controller MPC2 adjusts the applied force (middle plot) so M1 quickly returns to the desired position.

When the desired position changes step-wise to 5, the two masses separate briefly, with the controller switching to MPC1 appropriately. But, for the most part, the two masses move together and settle rapidly at the desired location. The transition back to -5 is equally well behaved.

Suppose we force MPC2 to operate under all conditions.

The figure below shows the result.



When the masses are disconnected, as at the start, MPC2 applies excessive force and then overcompensates, resulting in oscillatory behavior. Once the masses join, the move more smoothly, as would be expected.

The last transition causes especially severe oscillations. The masses collide frequently and M1 never reaches the desired position.

If MPC1 is in charge exclusively, the masses move sluggishly and fail to settle at the desired position before the next transition occurs. For more information, see the Scheduling Controllers for a Plant with Multiple Operating Points.

In this application, at least, two controllers are, indeed, better than one.

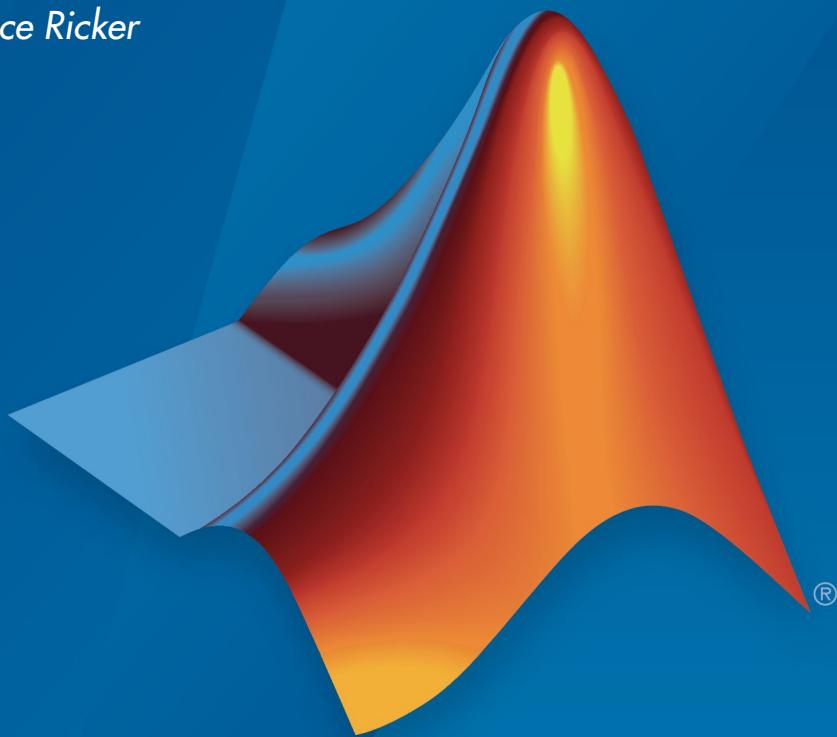
Model Predictive Control Toolbox™

User's Guide

Alberto Bemporad

Manfred Morari

N. Lawrence Ricker



MATLAB®

R2016a

 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

Model Predictive Control Toolbox™ User's Guide

© COPYRIGHT 2005–2016 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to 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

October 2004	First printing	New for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.2.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)
March 2010	Online only	Revised for Version 3.2 (Release 2010a)
September 2010	Online only	Revised for Version 3.2.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.2 (Release R2013a)
September 2013	Online only	Revised for Version 4.1.3 (Release R2013b)
March 2014	Online only	Revised for Version 4.2 (Release R2014a)
October 2014	Online only	Revised for Version 5.0 (Release R2014b)
March 2015	Online only	Revised for Version 5.0.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.1 (Release 2015b)
March 2016	Online only	Revised for Version 5.2 (Release 2016a)

Introduction

1

Specifying Scale Factors	1-2
Overview	1-2
Defining Scale Factors	1-2
Choosing Sample Time and Horizons	1-6
Sample Time	1-6
Prediction Horizon	1-7
Control Horizon	1-8
Defining Sample Time and Horizons	1-8
Specifying Constraints	1-10
Input and Output Constraints	1-10
Constraint Softening	1-12
Tuning Weights	1-16
Initial Tuning	1-16
Testing and Refinement	1-18
Robustness	1-19

Model Predictive Control Problem Setup

2

Optimization Problem	2-2
Overview	2-2
Standard Cost Function	2-2
Alternative Cost Function	2-6
Constraints	2-7
QP Matrices	2-8
Unconstrained Model Predictive Control	2-13

Adjusting Disturbance and Noise Models	2-15
Overview	2-15
Output Disturbance Model	2-16
Measurement Noise Model	2-18
Input Disturbance Model	2-20
Restrictions	2-22
Disturbance Rejection Tuning	2-23
Custom State Estimation	2-25
Time-Varying Weights and Constraints	2-26
Time-Varying Weights	2-26
Time-Varying Constraints	2-28
Terminal Weights and Constraints	2-30
Constraints on Linear Combinations of Inputs and Outputs	2-33
Manipulated Variable Blocking	2-35
QP Solver	2-38
Custom QP Application	2-39
Custom QP Solver	2-39
Controller State Estimation	2-42
Controller State Variables	2-42
State Observer	2-43
State Estimation	2-44
Built-in Steady-State Kalman Gains Calculation	2-46
Output Variable Prediction	2-47

Model Predictive Control Simulink Library

3

MPC Library	3-2
Relationship of Multiple MPC Controllers to MPC Controller Block	3-3
Listing the controllers	3-3

Designing the controllers	3-3
Defining controller switching	3-3
Improving prediction accuracy	3-4
Generate Code and Deploy Controller to Real-Time Targets	3-5
Code Generation in Simulink	3-5
Code Generation in MATLAB	3-5

Case-Study Examples

4

Design MPC Controller for Position Servomechanism	4-2
Design MPC Controller for Paper Machine Process	4-24
Bumpless Transfer Between Manual and Automatic Operations	4-50
Open Simulink Model	4-50
Define Plant and MPC Controller	4-51
Configure MPC Block Settings	4-52
Examine Switching Between Manual and Automatic Operation	4-53
Turn off Manipulated Variable Feedback	4-55
Switching Controller Online and Offline with Bumpless Transfer	4-58
Coordinate Multiple Controllers at Different Operating Points	4-64
Use Custom Constraints in Blending Process	4-72
Providing LQR Performance Using Terminal Penalty	4-83
References	4-88
Real-Time Control with OPC Toolbox	4-89
Simulation and Code Generation Using Simulink Coder ..	4-94

Simulation and Structured Text Generation Using PLC Coder	4-104
Generate Code To Compute Optimal MPC Moves in MATLAB	4-108
Setting Targets for Manipulated Variables	4-116
Specifying Alternative Cost Function with Off-Diagonal Weight Matrices	4-120
Review Model Predictive Controller for Stability and Robustness Issues	4-125
Control of an Inverted Pendulum on a Cart	4-144
Simulate MPC Controller with a Custom QP Solver	4-155

Adaptive MPC Design

5

Adaptive MPC	5-2
When to Use Adaptive MPC	5-2
Plant Model	5-3
Nominal Operating Point	5-4
State Estimation	5-4
Model Updating Strategy	5-6
Overview	5-6
Other Considerations	5-6
Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization	5-8
Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation	5-21
Time-Varying MPC	5-34
When to Use Time-Varying MPC	5-34
Time-Varying Prediction Models	5-34

Time-Varying Nominal Conditions	5-36
State Estimation	5-37
Time-Varying MPC Control of a Time-Varying Plant	5-39

Explicit MPC Design

6

Explicit MPC	6-2
Design Workflow for Explicit MPC	6-4
Traditional (Implicit) MPC Design	6-4
Explicit MPC Generation	6-5
Explicit MPC Simplification	6-6
Implementation	6-6
Simulation	6-7
Explicit MPC Control of a Single-Input-Single-Output Plant	6-9
Explicit MPC Control of an Aircraft with Unstable Poles ..	6-21
Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output	6-30
Explicit MPC Control of an Inverted Pendulum on a Cart ..	6-42

Gain Scheduling MPC Design

7

Gain-Scheduled MPC	7-2
Design Workflow for Gain Scheduling	7-3
General Design Steps	7-3
Tips	7-3

Gain Scheduled MPC Control of Nonlinear Chemical Reactor	7-5
Gain Scheduled MPC Control of Mass-Spring System	7-28
Gain Scheduled MPC Control of an Inverted Pendulum on a Cart	7-39

Reference for MPC Designer App

8

Generate MATLAB Code from MPC Designer	8-2
Generate Simulink Model from MPC Designer	8-4
Compare Multiple Controller Responses Using MPC Designer	8-6

Introduction

- “Specifying Scale Factors” on page 1-2
- “Choosing Sample Time and Horizons” on page 1-6
- “Specifying Constraints” on page 1-10
- “Tuning Weights” on page 1-16

Specifying Scale Factors

In this section...

“Overview” on page 1-2

“Defining Scale Factors” on page 1-2

Overview

Recommended practice includes specification of scale factors for each plant input and output variable, which is especially important when certain variables have much larger or smaller magnitudes than others.

The scale factor should equal (or approximate) the span of the variable. *Span* is the difference between its maximum and minimum value in engineering units, that is, the unit of measure specified in the plant model. Internally, MPC divides each plant input and output signal by its scale factor to generate dimensionless signals.

The potential benefits of scaling are as follows:

- Default MPC tuning weights work best when all signals are of order unity. Appropriate scale factors make the default weights a good starting point for controller tuning and refinement.
- When choosing cost function weights, you can focus on the relative priority of each term rather than a combination of priority and signal scale.
- Improved numerical conditioning. When values are scaled, round-off errors have less impact on calculations.

Once you have tuned the controller, changing a scale factor is likely to affect performance and the controller may need retuning. Best practice is to establish scale factors at the beginning of controller design and hold them constant thereafter.

Defining Scale Factors

To identify scale factors, estimate the span of each plant input and output variable in engineering units.

- If the signal has known bounds, use the difference between the upper and lower limit.

- If you do not know the signal bounds, consider running open-loop plant model simulations. You can vary the inputs over their likely ranges, and record output signal spans.
- If you have no idea, use the default scale factor (=1).

You can define scale factors at the command line and using the MPC Designer app.

Once you have set the scale factors and have begun to tune the controller performance, hold the scale factors constant.

Using Commands

After you create the MPC controller object using the `mpc` command, set the scale factor property for each plant input and output variable.

For example, the following commands create a random plant, specify the signal types, and define a scale factor for each signal.

```
% Random plant for illustrative purposes: 5 inputs, 3 outputs
Plant = drss(4,3,5);
Plant.InputName = { MV1 , UD1 , MV2 , UD2 , MD };
Plant.OutputName = { UO , M01 , M02 };

% Example signal spans
Uspan = [2, 20, 0.1, 5, 2000];
Yspan = [0.01, 400, 75];

% Example signal type specifications
iMV = [1 3];
iMD = 5;
iUD = [2 4];
iDV = [iMD,iUD];
Plant = setmpcsignals(Plant, MV ,iMV, MD ,iMD, UD ,iUD, ...
    MO ,[2 3], UO ,1);
Plant.d(:,iMV) = 0; % MPC requires zero direct MV feed-through

% Controller object creation. Ts = 0.3 for illustration.
MPCobj = mpc(Plant, 0.3);

% Override default scale factors using specified spans
for i = 1:2
    MPCobj.MV(i).ScaleFactor = Uspan(iMV(i));
end
```

```
% NOTE: DV sequence is MD followed by UD
for i = 1:3
    MPCobj.DV(i).ScaleFactor = Uspan(iDV(i));
end
for i = 1:3
    MPCobj.OV(i).ScaleFactor = Yspan(i);
end
```

Using MPC Designer App

After opening MPC Designer and defining the initial MPC structure, in the MPC

Designer tab, click **I/O Attributes**.



In the Input and Output Channel Specifications dialog box, specify a **Scale Factor** for each input and output signal.

Input and Output Channel Specifications

Plant Inputs					
Channel	Type	Name	Unit	Nominal Value	Scale Factor
u(1)	MV	T_c		0	1
u(2)	UD	C_A_i		0	1

Plant Outputs					
Channel	Type	Name	Unit	Nominal Value	Scale Factor
y(1)	MO	T		0	1
y(2)	UO	C_A		0	1

OK **Apply** **Cancel** **Help**

Click **OK** to update the controller settings.

See Also

[mpc](#) | MPC Designer

Related Examples

- Using Scale Factor to Facilitate Weight Tuning

More About

- “Choosing Sample Time and Horizons” on page 1-6

Choosing Sample Time and Horizons

In this section...

- “Sample Time” on page 1-6
- “Prediction Horizon” on page 1-7
- “Control Horizon” on page 1-8
- “Defining Sample Time and Horizons” on page 1-8

Sample Time

Duration

Recommended practice is to choose the control interval duration (controller property T_s) initially, and then hold it constant as you tune other controller parameters. If it becomes obvious that the original choice was poor, you can revise T_s . If you do so, you might then need to retune other settings.

Qualitatively, as T_s decreases, rejection of unknown disturbance usually improves and then plateaus. The T_s value at which performance plateaus depends on the plant dynamic characteristics.

However, as T_s becomes small, the computational effort increases dramatically. Thus, the optimal choice is a balance of performance and computational effort.

In Model Predictive Control, the prediction horizon, p is also an important consideration. If one chooses to hold the prediction horizon duration (the product p^*T_s) constant, p must vary inversely with T_s . Many array sizes are proportional to p . Thus, as p increases, the controller memory requirements and QP solution time increase.

Consider the following when choosing T_s :

- As a rough guideline, set T_s between 10% and 25% of your minimum desired closed-loop response time.
- Run at least one simulation to see whether unmeasured disturbance rejection improves significantly when T_s is halved. If so, consider revising T_s .
- For process control, $T_s \gg 1$ s is common, especially when MPC supervises lower-level single-loop controllers. Other applications, such as automotive or aerospace), can

require $T_s < 1$ s. If the time needed for solving the QP in real time exceeds the desired control interval, consider the “Explicit MPC” on page 6-2 option.

- For plants with delays, the number of state variables needed for modeling delays is inversely proportional to T_s .
- For open-loop unstable plants, if p^*T_s is too large, such that the plant step responses become infinite during this amount of time, key parameters needed for MPC calculations become undefined, generating an error message.

Units

The controller inherits its time unit from the plant model. Specifically, the controller uses the `TimeUnit` property of the plant model LTI object. This property defaults to seconds.

Prediction Horizon

Suppose that the current control interval is k . The *prediction horizon*, p , is the number of future control intervals the MPC controller must evaluate by prediction when optimizing its MVs at control interval k .

Tips

- Recommended practice is to choose p early in the controller design and then hold it constant while tuning other controller settings, such as the cost function weights. In other words, do not use p adjustments for controller tuning. Rather, the value of p should be such that the controller is internally stable and anticipates constraint violations early enough to allow corrective action.
- If the desired closed-loop response time is T and the control interval is T_s , try p such that $T \approx pT_s$.
- Plant delays impose a lower bound on the possible closed-loop response times. Choose p accordingly. To check for a violation of this condition, use the `review` command.
- Recommended practice is to increase p until further increases have a minor impact on performance. If the plant is open-loop unstable, the maximum p is the number of control intervals required for the open-loop step response of the plant to become infinite. $p > 50$ is rarely necessary unless T_s is too small.
- Unfavorable plant characteristics combined with a small p can generate an internally unstable controller. To check for this condition, use the `review` command, and increase p if possible. If p is already large, consider the following:
 - Increase T_s .

- Increase the cost function weights on MV increments.
- Modify the control horizon or use MV blocking (see “Manipulated Variable Blocking” on page 2-35).
- Use a small p with terminal weighting to approximate LQR behavior (See “Terminal Weights and Constraints” on page 2-30).

Control Horizon

The control horizon, m , is the number of MV moves to be optimized at control interval k . The control horizon falls between 1 and the prediction horizon p . The default is $m = 2$. Regardless of your choice for m , when the controller operates, the optimized MV move at the beginning of the horizon is used and any others are discarded.

Tips

Reasons to keep $m \ll p$ are as follows:

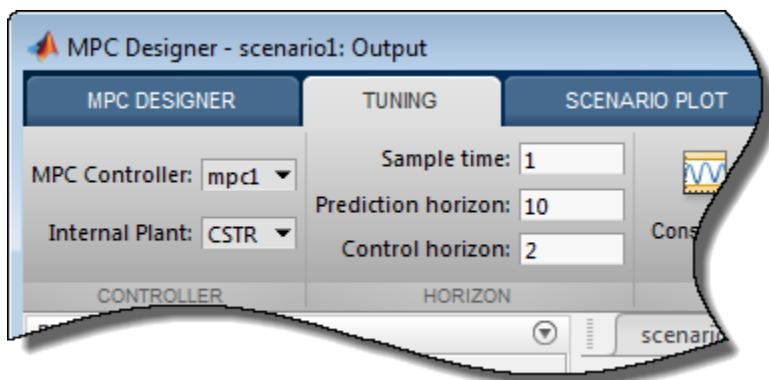
- Small m means fewer variables to compute in the QP solved at each control interval, which promotes faster computations.
- If the plant includes delays, $m < p$ is essential. Otherwise, some MV moves might not affect any of the plant outputs before the end of the prediction horizon, leading to a singular QP Hessian matrix. To check for a violation of this condition, use the `review` command.
- Small m promotes (but does not guarantee) an internally stable controller.

Defining Sample Time and Horizons

You can define the sample time, prediction horizon, and control horizon when creating an `mpc` controller at the command line. After creating a controller, `mpcObj`, you can modify the sample time and horizons by setting the following controller properties:

- Sample time — `mpcObj.Ts`
- Prediction horizon — `mpcObj.p`
- Control horizon — `mpcObj.m`

Also, when designing an MPC controller using the MPC Designer app, in the **Tuning** tab, in the **Horizon** section, you can modify the sample time and horizons.



See Also

[mpc](#) | MPC Designer

More About

- “Specifying Constraints” on page 1-10

Specifying Constraints

In this section...

[“Input and Output Constraints” on page 1-10](#)

[“Constraint Softening” on page 1-12](#)

Input and Output Constraints

By default, when you create a controller object using the `mpc` command, no constraints exist. To include a constraint, set the appropriate controller property. The following table summarizes the controller properties used to define most MPC Toolbox constraints. (MV = plant manipulated variable; OV = plant output variable; MV increment = $u(k) - u(k - 1)$).

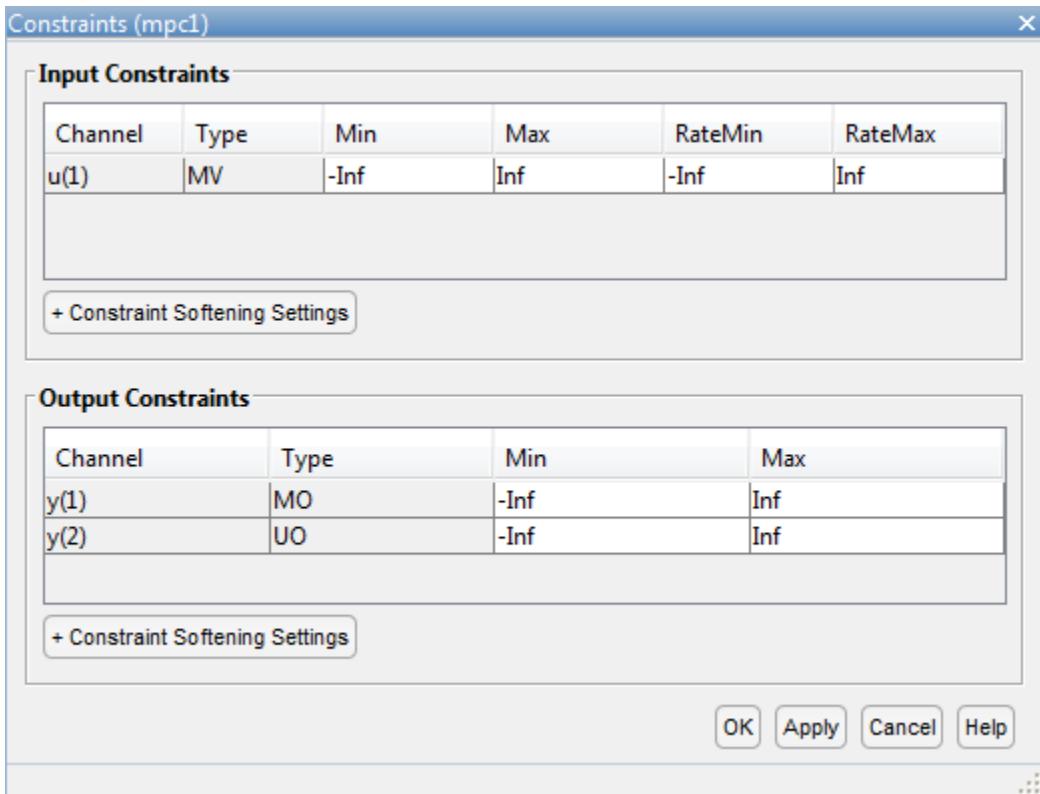
To include this constraint	Set this controller property	Softens constraint by setting
Lower bound on <i>i</i> th MV	<code>MV(i).Min > -Inf</code>	<code>MV(i).MinECR > 0</code>
Upper bound on <i>i</i> th MV	<code>MV(i).Max < Inf</code>	<code>MV(i).MaxECR > 0</code>
Lower bound on <i>i</i> th OV	<code>OV(i).Min > -Inf</code>	<code>OV(i).MinECR > 0</code>
Upper bound on <i>i</i> th OV	<code>OV(i).Max < Inf</code>	<code>OV(i).MaxECR > 0</code>
Lower bound on <i>i</i> th MV increment	<code>MV(i).RateMin > -Inf</code>	<code>MV(i).RateMinECR > 0</code>
Upper bound on <i>i</i> th MV increment	<code>MV(i).RateMax < Inf</code>	<code>MV(i).RateMaxECR > 0</code>

To set the controller constraint properties using the MPC Designer app, in the **Tuning**



tab, click **Constraints**. In the Constraints dialog box, specify the constraint values.

See “Constraints” on page 2-7 for the equations describing the corresponding constraints.



Tips

For MV bounds:

- Include known physical limits on the plant MVs as hard MV bounds.
- Include MV increment bounds when there is a known physical limit on the rate of change, or your application requires you to prevent large increments for some other reason.
- Do not include both hard MV bounds and hard MV increment bounds on the same MV, as they can conflict. If both types of bounds are important, soften one.

For OV bounds:

- Do not include OV bounds unless they are essential to your application. As an alternative to setting an OV bound, you can define an OV reference and set its cost function weight to keep the OV close to its setpoint.
- All OV constraints should be softened.
- Consider leaving the OV unconstrained for some prediction horizon steps. See “Time-Varying Weights and Constraints” on page 2-26.
- Consider a time-varying OV constraint that is easy to satisfy early in the horizon, gradually tapering to a more strict constraint. See “Time-Varying Weights and Constraints” on page 2-26.
- Do not include OV constraints that are impossible to satisfy. Even if soft, such constraints can cause unexpected controller behavior. For example, consider a SISO plant with five sampling periods of delay. An OV constraint before the sixth prediction horizon step is, in general, impossible to satisfy. You can use the `review` command to check for such impossible constraints, and use a time-varying OV bound instead. See “Time-Varying Weights and Constraints” on page 2-26.

Constraint Softening

Hard constraints are constraints that the quadratic programming (QP) solution must satisfy. If it is mathematically impossible to satisfy a hard constraint at a given control interval, k , the QP is *infeasible*. In this case, the controller returns an error status, and sets the manipulated variables (MVs) to $u(k) = u(k-1)$, i.e., no change. If the condition leading to infeasibility is not resolved, infeasibility can continue indefinitely, leading to a loss of control.

Disturbances and prediction errors are inevitable in practice. Therefore, a constraint violation could occur in the plant even though the controller predicts otherwise. A feasible QP solution does not guarantee that all hard constraints will be satisfied when the optimal MV is used in the plant.

If the only constraints in your application are bounds on MVs, the MV bounds can be hard constraints, as they are by default. MV bounds alone cannot cause infeasibility. The same is true when the only constraints are on MV increments.

However, a hard MV bound with a hard MV increment constraint can lead to infeasibility. For example, an upset or operation under manual control could cause the actual MV used in the plant to exceed the specified bound during interval $k-1$. If the controller is in automatic during interval k , it must return the MV to a value within the hard bound. If the MV exceeds the bound by too much, the hard increment constraint can make correcting the bound violation in the next interval impossible.

When there are hard constraints on plant outputs, or hard custom constraints (on linear combinations of plant inputs and outputs, and the plant is subject to disturbances, QP infeasibility is a distinct possibility.

All Model Predictive Control Toolbox™ constraints (except slack variable nonnegativity) can be *soft*. When a constraint is soft, the controller can deem an MV optimal even though it predicts a violation of that constraint. If all plant output, MV increment, and custom constraints are soft (as they are by default), QP infeasibility does not occur. However, controller performance can be substandard.

To soften a constraint, set the corresponding ECR value to a positive value (zero implies a hard constraint). The larger the ECR value, the more likely the controller will deem it optimal to violate the constraint in order to satisfy your other performance goals. The Model Predictive Control Toolbox software provides default ECR values but, as for the cost function weights, you might need to tune the ECR values in order to achieve acceptable performance.

To understand how constraint softening works, suppose that your cost function uses $w_{i,j}^u = w_{i,j}^{\Delta u} = 0$, giving both the MV and MV increments zero weight in the cost function. Only the output reference tracking and constraint violation terms are nonzero. In this case, the cost function is:

$$J(z_k) = \sum_{j=1}^{n_y} \sum_{i=1}^p \left\{ \frac{w_{i,j}^y}{s_j^y} [r_j(k+i|k) - y_j(k+i|k)] \right\}^2 + \rho_k^2.$$

Suppose that you have also specified hard MV bounds with $V_{j,min}^u(i) = 0$ and $V_{j,max}^u(i) = 0$. Then these constraints simplify to:

$$\frac{u_{j,min}(i)}{s_j^u} \leq \frac{u_j(k+i-1|k)}{s_j^u} \leq \frac{u_{j,max}(i)}{s_j^u}, \quad i = 1:p, \quad j = 1:n_u.$$

Thus, the slack variable, ϵ_k , no longer appears in the above equations. You have also specified soft constraints on plant outputs with $V_{j,min}^y(i) > 0$ and $V_{j,max}^y(i) > 0$.

$$\frac{y_{j,min}(i)}{s_j^y} - {}_k V_{j,min}^y(i) \leq \frac{y_j(k+i|k)}{s_j^y} \leq \frac{y_{j,max}(i)}{s_j^y} + {}_k V_{j,max}^y(i), \quad i = 1:p, \quad j = 1:n_y.$$

Now, suppose that a disturbance has pushed a plant output above its specified upper bound, but the QP with hard output constraints would be feasible, that is, all constraint violations could be avoided in the QP solution. The QP involves a trade-off between output reference tracking and constraint violation. The slack variable, ϵ_k , must be nonnegative. Its appearance in the cost function discourages, but does not prevent, an optimal $\epsilon_k > 0$. A larger ρ_ϵ weight, however, increases the likelihood that the optimal ϵ_k will be small or zero.

If the optimal $\epsilon_k > 0$, at least one of the bound inequalities must be active (at equality). A relatively large $V_{j,max}^y(i)$ makes it easier to satisfy the constraint with a small ϵ_k . In that case,

$$\frac{y_j(k+i|k)}{s_j^y}$$

can be larger, without exceeding

$$\frac{y_{j,max}(i)}{s_j^y} + {}_k V_{j,max}^y(i).$$

Notice that $V_{j,max}^y(i)$ does not set an upper limit on the constraint violation. Rather, it is a tuning factor determining whether a soft constraint is easy or difficult to satisfy.

Tips

- Use of dimensionless variables simplifies constraint tuning. Define appropriate scale factors for each plant input and output variable. See “Specifying Scale Factors” on page 1-2.
- To indicate the relative magnitude of a tolerable violation, use the ECR parameter associated with each constraint. Rough guidelines are as follows:
 - 0 — No violation allowed (hard constraint)

- 0.05 — Very small violation allowed (nearly hard)
- 0.2 — Small violation allowed (quite hard)
- 1 — average softness
- 5 — greater-than-average violation allowed (quite soft)
- 20 — large violation allowed (very soft)
- Use the overall constraint softening parameter of the controller (controller object property: `Weights. ECR`) to penalize a tolerable soft constraint violation relative to the other cost function terms. Set the `Weights. ECR` property such that the corresponding penalty is 1–2 orders of magnitude greater than the typical sum of the other three cost function terms. If constraint violations seem too large during simulation tests, try increasing `Weights. ECR` by a factor of 2–5.

Be aware, however, that an excessively large `Weights. ECR` distorts MV optimization, leading to inappropriate MV adjustments when constraint violations occur. To check for this, display the cost function value during simulations. If its magnitude increases by more than 2 orders of magnitude when a constraint violation occurs, consider decreasing `Weights. ECR`.

- Disturbances and prediction errors can lead to unexpected constraint violations in a real system. Attempting to prevent these violations by making constraints harder often degrades controller performance.

See Also

[review](#)

More About

- “Time-Varying Weights and Constraints” on page 2-26
- “Terminal Weights and Constraints” on page 2-30
- “Optimization Problem” on page 2-2

Tuning Weights

In this section...

- “Initial Tuning” on page 1-16
- “Testing and Refinement” on page 1-18
- “Robustness” on page 1-19

A model predictive controller design usually requires some tuning of the cost function weights. This topic provides tuning tips. See “Optimization Problem” on page 2-2 for details on the cost function equations.

Initial Tuning

- Before tuning the cost function weights, specify scale factors for each plant input and output variable. Hold these scale factors constant as you tune the controller. See “Specifying Scale Factors” on page 1-2 for more information.
- During tuning, use the **sensitivity** and **review** commands to obtain diagnostic feedback. The **sensitivity** command is intended to help with cost function weight selection.
- Change a weight by setting the appropriate controller property, as follows:

To change this weight	Set this controller property	Array size
OV reference tracking (w^y)	Weights.OV	p -by- n_y
MV reference tracking (w^u)	Weights.MV	p -by- n_u
MV increment suppression ($w^{\Delta u}$)	Weights.MVRate	p -by- n_u

Here, MV is a plant manipulated variable, and n_u is the number of MVs. OV is a plant output variable, and n_y is the number of OVs. Finally, p is the number of steps in the prediction horizon.

If a weight array contains $n < p$ rows, the controller duplicates the last row to obtain a full array of p rows. The default ($n = 1$) minimizes the number of parameters to be tuned, and is therefore recommended. See “Time-Varying Weights and Constraints” on page 2-26 for an alternative.

Tips for Setting OV Weights

- Considering the n_y OVs, suppose that n_{yc} must be held at or near a reference value (setpoint). If the i th OV is not in this group, set `Weights.OV(:, i) = 0`.
- If $n_u \geq n_{yc}$, it is usually possible to achieve zero OV tracking error at steady state, if at least n_{yc} MVs are not constrained. The default `Weights.OV = ones(1, ny)` is a good starting point in this case.

If $n_u > n_{yc}$, however, you have excess degrees of freedom. Unless you take preventive measures, therefore, the MVs may drift even when the OVs are near their reference values.

- The most common preventive measure is to define reference values (targets) for the number of excess MVs you have, $n_u - n_{yc}$. Such targets can represent economically or technically desirable steady-state values.
- An alternative measure is to set $w_{\Delta u} > 0$ for at least $n_u - n_{yc}$ MVs to discourage the controller from changing them.
- If $n_u < n_{yc}$, you do not have enough degrees of freedom to keep all required OVs at a setpoint. In this case, consider prioritizing reference tracking. To do so, set `Weights.OV(:, i) > 0` to specify the priority for the i th OV. Rough guidelines for this are as follows:
 - 0.05 — Low priority: Large tracking error acceptable
 - 0.2 — Below-average priority
 - 1 — Average priority – the default. Use this value if $n_{yc} = 1$.
 - 5 — Above average priority
 - 20 — High priority: Small tracking error desired

Tips for Setting MV Weights

By default, `Weights.MV = zeros(1, nu)`. If some MVs have targets, the corresponding MV reference tracking weights must be nonzero. Otherwise, the targets are ignored. If the number of MV targets is less than $(n_u - n_{yc})$, try using the same weight for each. A suggested value is 0.2, the same as below-average OV tracking. This value allows the MVs to move away from their targets temporarily to improve OV tracking.

Otherwise, the MV and OV reference tracking goals are likely to conflict. Prioritize by setting the `Weights.MV(:, i)` values in a manner similar to that suggested for

`Weights.OV` (see above). Typical practice sets the average MV tracking priority lower than the average OV tracking priority (e.g., $0.2 < 1$).

If the i th MV does not have a target, set `Weights.MV(:, i) = 0` (the default).

Tips for Setting MVRate Weights

- By default, `Weights.MVRate = 0.1*ones(1, nu)`. The reasons for this default include:
 - If the plant is open-loop stable, large increments are unnecessary and probably undesirable. For example, when model predictions are imperfect, as is always the case in practice, more conservative increments usually provide more robust controller performance, but poorer reference tracking.
 - These values force the QP Hessian matrix to be positive-definite, such that the QP has a unique solution if no constraints are active.

To encourage the controller to use even smaller increments for the i th MV, increase the `Weights.MVRate(:, i)` value.

- If the plant is open-loop unstable, you might need to decrease the average `Weight.MVRate` value to allow sufficiently rapid response to upsets.

Tips for Setting ECR Weights

See “Constraint Softening” on page 1-12 for tips regarding the `Weights.ECR` property.

Testing and Refinement

To focus on tuning individual cost function weights, perform closed-loop simulation tests under the following conditions:

- No constraints.
- No prediction error. The controller prediction model should be identical to the plant model. Both the MPC Designer app and the `sim` function provide the option to simulate under these conditions.

Use changes in the reference and measured disturbance signals (if any) to force a dynamic response. Based on the results of each test, consider changing the magnitudes of selected weights.

One suggested approach is to use constant `Weights.OV(:, i) = 1` to signify “average OV tracking priority,” and adjust all other weights to be relative to this value. Use the

sensitivity command for guidance. Use the **review** command to check for typical tuning issues, such as lack of closed-loop stability.

See “Adjusting Disturbance and Noise Models” on page 2-15 for tests focusing on the disturbance rejection ability of the controller.

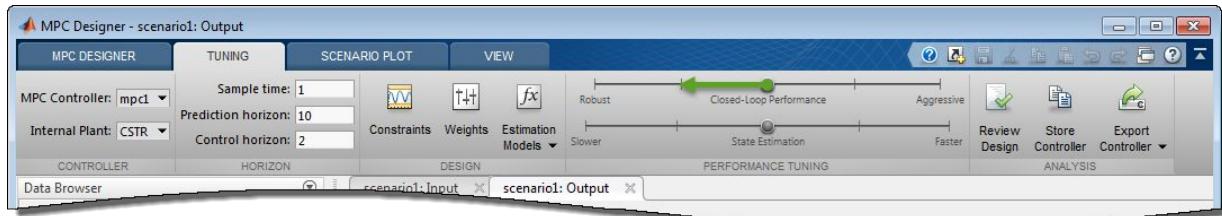
Robustness

Once you have weights that work well under the above conditions, check for sensitivity to prediction error. There are several ways to do so:

- If you have a nonlinear plant model of your system, such as a Simulink® model, simulate the closed-loop performance at operating points other than that for which the LTI prediction model applies.
- Alternatively, run closed-loop simulations in which the LTI model representing the plant differs (such as in structure or parameter values) from that used at the MPC prediction model. Both the MPC Designer app and the **sim** function provide the option to simulate under these conditions. See “Test Controller Robustness” for an example.

If controller performance seems to degrade significantly in comparison to tests with no prediction error, for an open-loop stable plant, consider making the controller less aggressive.

In the MPC Designer app, on the **Tuning** tab, you can do so using the **Closed-Loop Performance** slider.



Moving towards more robust control decreases OV/MV weights and increases MV Rate weights, which leads to relaxed control of outputs and more conservative control moves.

At the command line, you can make the following changes to decrease controller aggressiveness:

- Increase all `Weight.MVRate` values by a multiplicative factor of order 2.
- Decrease all `Weight.OV` and `Weight.MV` values by dividing by the same factor.

After adjusting the weights, reevaluate performance both with and without prediction error.

- If both are now acceptable, stop tuning the weights.
- If there is improvement but still too much degradation with model error, increase the controller robustness further.
- If the change does not noticeably improve performance, restore the original weights and focus on state estimator tuning (see “Adjusting Disturbance and Noise Models” on page 2-15).

Finally, if tuning changes do not provide adequate robustness, consider one of the following options:

- “Adaptive MPC” on page 5-2
- “Gain-Scheduled MPC” on page 7-2

Related Examples

- Tuning Controller Weights
- “Setting Targets for Manipulated Variables” on page 4-116

More About

- “Optimization Problem” on page 2-2
- “Specifying Constraints” on page 1-10
- “Adjusting Disturbance and Noise Models” on page 2-15

Model Predictive Control Problem Setup

- “Optimization Problem” on page 2-2
- “Adjusting Disturbance and Noise Models” on page 2-15
- “Custom State Estimation” on page 2-25
- “Time-Varying Weights and Constraints” on page 2-26
- “Terminal Weights and Constraints” on page 2-30
- “Constraints on Linear Combinations of Inputs and Outputs” on page 2-33
- “Manipulated Variable Blocking” on page 2-35
- “QP Solver” on page 2-38
- “Controller State Estimation” on page 2-42

Optimization Problem

In this section...

- “Overview” on page 2-2
- “Standard Cost Function” on page 2-2
- “Alternative Cost Function” on page 2-6
- “Constraints” on page 2-7
- “QP Matrices” on page 2-8
- “Unconstrained Model Predictive Control” on page 2-13

Overview

Model Predictive Control solves an optimization problem – specifically, a quadratic program (QP) – at each control interval. The solution determines the manipulated variables (MVs) to be used in the plant until the next control interval.

This QP problem includes the following features:

- The objective, or “cost”, function — A scalar, nonnegative measure of controller performance to be minimized.
- Constraints — Conditions the solution must satisfy, such as physical bounds on MVs and plant output variables.
- Decision — The MV adjustments that minimizes the cost function while satisfying the constraints.

The following sections describe these features in more detail.

Standard Cost Function

The standard cost function is the sum of four terms, each focusing on a particular aspect of controller performance, as follows:

$$J(z_k) = J_y(z_k) + J_u(z_k) + J_{\Delta u}(z_k) + J(z_k).$$

Here, z_k is the QP decision. As described below, each term includes weights that help you balance competing objectives. MPC controller provides default weights but you will usually need to adjust them to tune the controller for your application.

Output Reference Tracking

In most applications, the controller must keep selected plant outputs at or near specified reference values. MPC controller uses the following scalar performance measure:

$$J_y(z_k) = \sum_{j=1}^{n_y} \sum_{i=1}^p \left\{ \frac{w_{i,j}^y}{s_j^y} [r_j(k+i|k) - y_j(k+i|k)] \right\}^2.$$

Here,

- k — Current control interval.
- p — Prediction horizon (number of intervals).
- n_y — Number of plant output variables.
- z_k — QP decision, given by:

$$z_k^T = [u(k|k)^T \quad u(k+1|k)^T \quad \dots \quad u(k+p-1|k)^T \quad k].$$

- $y_j(k+i|k)$ — Predicted value of j th plant output at i th prediction horizon step, in engineering units.
- $r_j(k+i|k)$ — Reference value for j th plant output at i th prediction horizon step, in engineering units.
- s_j^y — Scale factor for j th plant output, in engineering units.
- $w_{i,j}^y$ — Tuning weight for j th plant output at i th prediction horizon step (dimensionless).

The values n_y , p , s_j^y , and $w_{i,j}^y$ are controller specifications, and are constant. The controller receives $r_j(k+i|k)$ values for the entire prediction horizon. The controller uses the state observer to predict the plant outputs. At interval k , the controller state estimates and MD values are available. Thus, J_y is a function of z_k only.

Manipulated Variable Tracking

In some applications, i.e. when there are more manipulated variables than plant outputs, the controller must keep selected manipulated variables (MVs) at or near specified target values. MPC controller uses the following scalar performance measure:

$$J_u(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^u}{s_j^u} [u_j(k+i|k) - u_{j,target}(k+i|k)] \right\}^2.$$

Here,

- k — Current control interval.
- p — Prediction horizon (number of intervals).
- n_u — Number of manipulated variables.
- z_k — QP decision, given by:

$$z_k^T = [u(k|k)^T \quad u(k+1|k)^T \quad \dots \quad u(k+p-1|k)^T \quad k].$$

- $u_{j,target}(k+i|k)$ — Target value for j th MV at i th prediction horizon step, in engineering units.
- s_j^u — Scale factor for j th MV, in engineering units.
- $w_{i,j}^u$ — Tuning weight for j th MV at i th prediction horizon step (dimensionless).

The values n_u , p , s_j^u , and $w_{i,j}^u$ are controller specifications, and are constant. The controller receives $u_{j,target}(k+i|k)$ values for the entire horizon. The controller uses the state observer to predict the plant outputs. Thus, J_u is a function of z_k only.

Manipulated Variable Move Suppression

Most applications prefer small MV adjustments (*moves*). MPC uses the following scalar performance measure:

$$J_{\Delta u}(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^{\Delta u}}{s_j^u} [u_j(k+i|k) - u_j(k+i-1|k)] \right\}^2.$$

Here,

Here,

- k — Current control interval.
- p — Prediction horizon (number of intervals).
- n_u — Number of manipulated variables.
- z_k — QP decision, given by:

$$z_k^T = [u(k|k)^T \quad u(k+1|k)^T \quad \dots \quad u(k+p-1|k)^T \quad k].$$

- s_j^u — Scale factor for j th MV, in engineering units.
- $w_{i,j}^{\Delta u}$ — Tuning weight for j th MV movement at i th prediction horizon step (dimensionless).

The values n_u , p , s_j^u , and $w_{i,j}^{\Delta u}$ are controller specifications, and are constant. $u(k-1|k) = u(k-1)$, which are the known MVs from the previous control interval. $J_{\Delta u}$ is a function of z_k only.

In addition, a control horizon $m < p$ (or MV blocking) constrains certain MV moves to be zero.

Constraint Violation

In practice, constraint violations might be unavoidable. Soft constraints allow a feasible QP solution under such conditions. MPC controller employs a dimensionless, nonnegative slack variable, ε_k , which quantifies the worst-case constraint violation. (See “Constraints” on page 2-7) The corresponding performance measure is:

$$J(z_k) = \rho \cdot \frac{2}{k}.$$

Here,

- z_k — QP decision, given by:

$$z_k^T = \begin{bmatrix} u(k|k)^T & u(k+1|k)^T & \cdots & u(k+p-1|k)^T & \varepsilon_k \end{bmatrix}.$$

- ε_k — Slack variable at control interval k (dimensionless).
- ρ_ε — Constraint violation penalty weight (dimensionless).

Alternative Cost Function

You can elect to use the following alternative to the standard cost function:

$$J(z_k) = \sum_{i=0}^{p-1} \left\{ [e_y^T(k+i) Q e_y(k+i)] + [e_u^T(k+i) R_u e_u(k+i)] + [\Delta u^T(k+i) R_{\Delta u} \Delta u(k+i)] \right\} + \rho \frac{\varepsilon_k^2}{k}.$$

Here, Q (n_y -by- n_y), R_u , and $R_{\Delta u}$ (n_u -by- n_u) are positive-semi-definite weight matrices, and:

$$\begin{aligned} e_y(i+k) &= S_y^{-1} [r(k+i+1|k) - y(k+i+1|k)] \\ e_u(i+k) &= S_u^{-1} [u_{target}(k+i|k) - u(k+i|k)] \\ \Delta u(k+i) &= S_u^{-1} [u(k+i|k) - u(k+i-1|k)]. \end{aligned}$$

Also,

- S_y — Diagonal matrix of plant output variable scale factors, in engineering units.
- S_u — Diagonal matrix of MV scale factors in engineering units.
- $r(k+1|k)$ — n_y plant output reference values at the i th prediction horizon step, in engineering units.
- $y(k+1|k)$ — n_y plant outputs at the i th prediction horizon step, in engineering units.
- z_k — QP decision, given by:

$$z_k^T = \begin{bmatrix} u(k|k)^T & u(k+1|k)^T & \cdots & u(k+p-1|k)^T & \varepsilon_k \end{bmatrix}.$$

- $u_{target}(k+i|k)$ — n_u MV target values corresponding to $u(k+i|k)$, in engineering units.

Output predictions use the state observer, as in the standard cost function.

The alternative cost function allows off-diagonal weighting, but requires the weights to be identical at each prediction horizon step.

The alternative and standard cost functions are identical if the following conditions hold:

- The standard cost functions employs weights $w_{i,j}^y$, $w_{i,j}^u$, and $w_{i,j}^{\Delta u}$ that are constant with respect to the index, $i = 1:p$.
- The matrices Q , R_u , and $R_{\Delta u}$ are diagonal with the squares of those weights as the diagonal elements.

Constraints

Certain constraints are implicit. For example, a control horizon $m < p$ (or MV blocking) forces some MV increments to be zero, and the state observer used for plant output prediction is a set of implicit equality constraints. Explicit constraints that you can configure are described below.

Bounds on Plant Outputs, MVs, and MV Increments

The most common MPC constraints are bounds, as follows.

$$\begin{aligned}\frac{y_{j,min}(i)}{s_j^y} - {}_k V_{j,min}^y(i) &\leq \frac{y_j(k+i|k)}{s_j^y} \leq \frac{y_{j,max}(i)}{s_j^y} + {}_k V_{j,max}^y(i), \quad i = 1:p, \quad j = 1:n_y \\ \frac{u_{j,min}(i)}{s_j^u} - {}_k V_{j,min}^u(i) &\leq \frac{u_j(k+i-1|k)}{s_j^u} \leq \frac{u_{j,max}(i)}{s_j^u} + {}_k V_{j,max}^u(i), \quad i = 1:p, \quad j = 1:n_u \\ \frac{\Delta u_{j,min}(i)}{s_j^u} - {}_k V_{j,min}^{\Delta u}(i) &\leq \frac{\Delta u_j(k+i-1|k)}{s_j^u} \leq \frac{\Delta u_{j,max}(i)}{s_j^u} + {}_k V_{j,max}^{\Delta u}(i), \quad i = 1:p, \quad j = 1:n_u.\end{aligned}$$

Here, the V parameters (ECR values) are dimensionless controller constants analogous to the cost function weights but used for constraint softening (see “Constraint Softening” on page 1-12). Also,

- ϵ_k — Scalar QP slack variable (dimensionless) used for constraint softening.

- s_j^y — Scale factor for j th plant output, in engineering units.
- s_j^u — Scale factor for j th MV, in engineering units.
- $y_{j,\min}(i), y_{j,\max}(i)$ — lower and upper bounds for j th plant output at i th prediction horizon step, in engineering units.
- $u_{j,\min}(i), u_{j,\max}(i)$ — lower and upper bounds for j th MV at i th prediction horizon step, in engineering units.
- $\Delta u_{j,\min}(i), \Delta u_{j,\max}(i)$ — lower and upper bounds for j th MV increment at i th prediction horizon step, in engineering units.

Except for the slack variable non-negativity condition, all of the above constraints are optional and are inactive by default (i.e., initialized with infinite limiting values). To include a bound constraint, you must specify a finite limit when you design the controller.

QP Matrices

This section describes the matrices associated with the model predictive control optimization problem described in “Optimization Problem” on page 2-2.

Prediction

Assume that the disturbance models described in “Input Disturbance Model” is unit gain, for example, $d(k)=n_d(k)$ is a white Gaussian noise). You can denote this problem as

$$x \leftarrow \begin{bmatrix} x \\ x_d \end{bmatrix}, A \leftarrow \begin{bmatrix} A & B_d \bar{C} \\ 0 & \bar{A} \end{bmatrix}, B_u \leftarrow \begin{bmatrix} B_u \\ 0 \end{bmatrix}, B_v \leftarrow \begin{bmatrix} B_v \\ 0 \end{bmatrix}, B_d \leftarrow \begin{bmatrix} B_d & \bar{D} \\ \bar{B} & \bar{B} \end{bmatrix} C \leftarrow \begin{bmatrix} C & D_d \bar{C} \end{bmatrix}$$

Then, the prediction model is:

$$x(k+1) = Ax(k) + B_u u(k) + B_v v(k) + B_d n_d(k)$$

$$y(k) = Cx(k) + D_v v(k) + D_d n_d(k)$$

Next, consider the problem of predicting the future trajectories of the model performed at time $k=0$. Set $n_d(i)=0$ for all prediction instants i , and obtain

$$y(i \mid 0) = C \left[A^i x(0) + \sum_{h=0}^{i-1} A^{i-1} \left(B_u \left(u(-1) + \sum_{j=0}^h \Delta u(j) \right) + B_v v(h) \right) \right] + D_v v(i)$$

This equation gives the solution

$$\begin{bmatrix} y(1) \\ \dots \\ y(p) \end{bmatrix} = S_x x(0) + S_{u1} u(-1) + S_u \begin{bmatrix} \Delta u(0) \\ \dots \\ \Delta u(p-1) \end{bmatrix} + H_v \begin{bmatrix} v(0) \\ \dots \\ v(p) \end{bmatrix}$$

where

$$S_x = \begin{bmatrix} CA \\ CA^2 \\ \dots \\ CA^p \end{bmatrix} \in \Re^{pn_y \times n_x}, S_{u1} = \begin{bmatrix} CB_u \\ CB_u + CAB_u \\ \dots \\ \sum_{h=0}^{p-1} CA^h B_u \end{bmatrix} \in \Re^{pn_y \times n_u}$$

$$S_u = \begin{bmatrix} CB_u & 0 & \dots & 0 \\ CB_u + CAB_u & CB_u & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \sum_{h=0}^{p-1} CA^h B_u & \sum_{h=0}^{p-2} CA^h B_u & \dots & CB_u \end{bmatrix} \in \Re^{pn_y \times pn_u}$$

$$H_v = \begin{bmatrix} CB_v & D_v & 0 & \dots & 0 \\ CAB_v & CB_v & D_v & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ CA^{p-1} B_v & CA^{p-2} B_v & CA^{p-3} B_v & \dots & D_v \end{bmatrix} \in \Re^{pn_y \times (p+1)n_v}$$

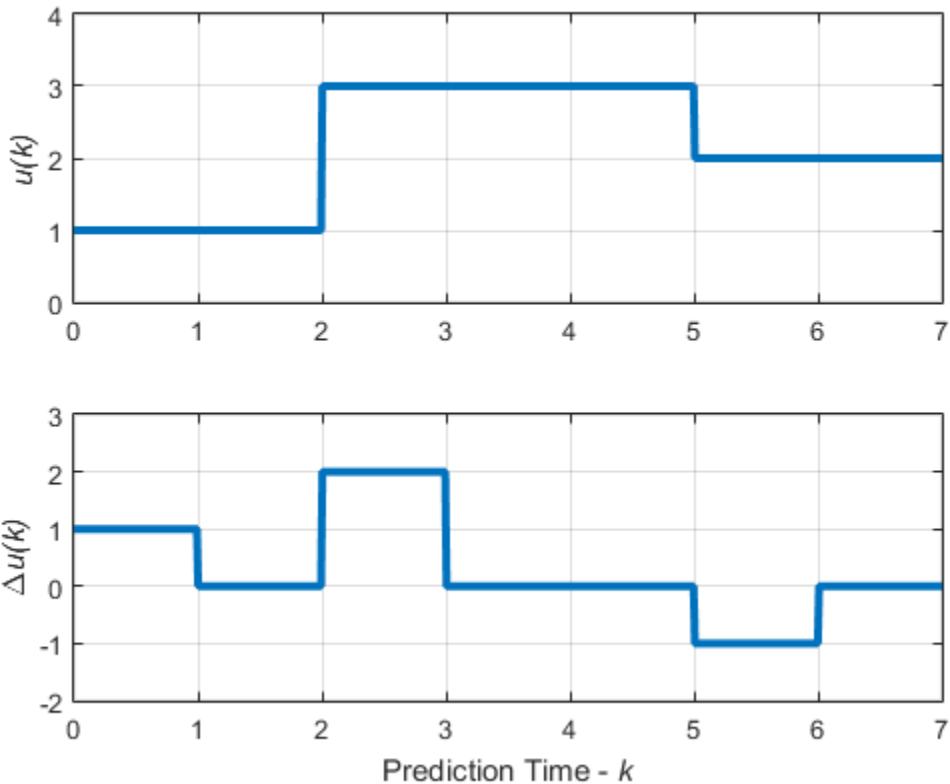
Optimization Variables

Let m be the number of free control moves, and let $z = [z_0; \dots; z_{m-1}]$. Then,

$$\begin{bmatrix} \Delta u(0) \\ \dots \\ \Delta u(p-1) \end{bmatrix} = J_M \begin{bmatrix} z_0 \\ \dots \\ z_{m-1} \end{bmatrix}$$

where J_M depends on the choice of blocking moves. Together with the slack variable ε , vectors z_0, \dots, z_{m-1} constitute the free optimization variables of the optimization problem. In the case of systems with a single manipulated variables, z_0, \dots, z_{m-1} are scalars.

Consider the blocking moves depicted in the following graph.



Blocking Moves: Inputs and Input Increments for moves = [2 3 2]

This graph corresponds to the choice $\text{moves}=[2 \ 3 \ 2]$, or, equivalently,
 $u(0)=u(1)$, $u(2)=u(3)=u(4)$, $u(5)=u(6)$, $\Delta u(0)=z_0$, $\Delta u(2)=z_1$, $\Delta u(5)=z_2$, $\Delta u(1)=\Delta u(3)=\Delta u(4)=\Delta u(6)=0$.

Then, the corresponding matrix J_M is

$$J_M = \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{bmatrix}$$

Cost Function

- “Standard Form” on page 2-11
- “Alternative Cost Function” on page 2-12

Standard Form

The function to be optimized is

$$\begin{aligned} J(z, \varepsilon) = & \left(\begin{bmatrix} u(0) \\ \dots \\ u(p-1) \end{bmatrix} - \begin{bmatrix} u_{target}(0) \\ \dots \\ u_{target}(p-1) \end{bmatrix} \right)^T W_u^2 \left(\begin{bmatrix} u(0) \\ \dots \\ u(p-1) \end{bmatrix} - \begin{bmatrix} u_{target}(0) \\ \dots \\ u_{target}(p-1) \end{bmatrix} \right) + \left[\begin{bmatrix} \Delta u(0) \\ \dots \\ \Delta u(p-1) \end{bmatrix} \right]^T W_{\Delta u}^2 \begin{bmatrix} \Delta u(0) \\ \dots \\ \Delta u(p-1) \end{bmatrix} \\ & + \left(\begin{bmatrix} y(1) \\ \dots \\ y(p) \end{bmatrix} - \begin{bmatrix} r(1) \\ \dots \\ r(p) \end{bmatrix} \right)^T W_y^2 \left(\begin{bmatrix} y(1) \\ \dots \\ y(p) \end{bmatrix} - \begin{bmatrix} r(1) \\ \dots \\ r(p) \end{bmatrix} \right) + \rho_\varepsilon \varepsilon^2 \end{aligned}$$

where

$$W_u = \text{diag}(w_{0,1}^u, w_{0,2}^u, \dots, w_{0,n_u}^u, \dots, w_{p-1,1}^u, w_{p-1,2}^u, \dots, w_{p-1,n_u}^u)$$

$$W_{\Delta u} = \text{diag}(w_{0,1}^{\Delta u}, w_{0,2}^{\Delta u}, \dots, w_{0,n_u}^{\Delta u}, \dots, w_{p-1,1}^{\Delta u}, w_{p-1,2}^{\Delta u}, \dots, w_{p-1,n_u}^{\Delta u})$$

$$W_y = \text{diag}(w_{1,1}^y, w_{1,2}^y, \dots, w_{1,n_y}^y, \dots, w_{p,1}^y, w_{p,2}^y, \dots, w_{p,n_y}^y)$$

Finally, after substituting $u(k)$, $\Delta u(k)$, $y(k)$, $J(z)$ can be rewritten as

$$J(z, \varepsilon) = \rho_\varepsilon \varepsilon^2 + z^T K_{\Delta u} z + 2 \left(\begin{bmatrix} r(1) \\ \vdots \\ r(p) \end{bmatrix}^T K_r + \begin{bmatrix} v(0) \\ \vdots \\ v(p) \end{bmatrix}^T K_v + u(-1)^T K_u + \begin{bmatrix} u_{target}(0) \\ \vdots \\ u_{target}(p-1) \end{bmatrix}^T K_{ut} + x(0)^T K_x \right) z + \text{constant}$$

Note You may want the QP problem to remain strictly convex. If the condition number of the Hessian matrix $K_{\Delta u}$ is larger than 10^{12} , add the quantity $10 * \text{sqrt}(\text{eps})$ on each diagonal term. You can use this solution only when all input rates are unpenalized ($W^{\Delta u} = 0$) (see “Weights” in the Model Predictive Control Toolbox reference documentation).

Alternative Cost Function

If you are using the alternative cost function shown in “Alternative Cost Function” on page 2-6, Equation 2-3, then Equation 2-2 is replaced by the following:

$$\begin{aligned} W_u &= \text{blkdiag}(R_u, \dots, R_u) \\ W_{\Delta u} &= \text{blkdiag}(R_{\Delta u}, \dots, R_{\Delta u}) \\ W_y &= \text{blkdiag}(Q, \dots, Q) \end{aligned}$$

In this case, the block-diagonal matrices repeat p times, for example, once for each step in the prediction horizon.

You also have the option to use a combination of the standard and alternative forms. See “Weights” in the Model Predictive Control Toolbox reference documentation for more details.

Constraints

Next, consider the limits on inputs, input increments, and outputs along with the constraint $\varepsilon \geq 0$.

$$\begin{bmatrix} y_{\min}(1) - \varepsilon V_{\min}^y(1) \\ \dots \\ y_{\min}(p) - \varepsilon V_{\min}^y(p) \\ u_{\min}(0) - \varepsilon V_{\min}^u(0) \\ \dots \\ u_{\min}(p-1) - \varepsilon V_{\min}^u(p-1) \\ \Delta u_{\min}(0) - \varepsilon V_{\min}^{\Delta u}(0) \\ \dots \\ \Delta u_{\min}(p-1) - \varepsilon V_{\min}^{\Delta u}(p-1) \end{bmatrix} \leq \begin{bmatrix} y(1) \\ \dots \\ y(p) \\ u(0) \\ \dots \\ u(p-1) \\ \Delta u(0) \\ \dots \\ \Delta u(p-1) \end{bmatrix} \leq \begin{bmatrix} y_{\max}(1) + \varepsilon V_{\max}^y(1) \\ \dots \\ y_{\max}(p) + \varepsilon V_{\max}^y(p) \\ u_{\max}(0) + \varepsilon V_{\max}^u(0) \\ \dots \\ u_{\max}(p-1) + \varepsilon V_{\max}^u(p-1) \\ \Delta u_{\max}(0) + \varepsilon V_{\max}^{\Delta u}(0) \\ \dots \\ \Delta u_{\max}(p-1) + \varepsilon V_{\max}^{\Delta u}(p-1) \end{bmatrix}$$

Note To reduce computational effort, the controller automatically eliminates extraneous constraints, such as infinite bounds. Thus, the constraint set used in real time may be much smaller than that suggested in this section.

Similar to what you did for the cost function, you can substitute $u(k)$, $\Delta u(k)$, $y(k)$, and obtain

$$M_z z + M_\varepsilon \varepsilon \leq M_{\lim} + M_v \begin{bmatrix} v(0) \\ \dots \\ v(p) \end{bmatrix} + M_u u(-1) + M_x x(0)$$

In this case, matrices $M_z, M_\varepsilon, M_{\lim}, M_v, M_u, M_x$ are obtained from the upper and lower bounds and ECR values.

Unconstrained Model Predictive Control

The optimal solution is computed analytically

$$z^* = -K_{\Delta u}^{-1} \left(\begin{bmatrix} r(1) \\ \dots \\ r(p) \end{bmatrix}^T K_r + \begin{bmatrix} v(0) \\ \dots \\ v(p) \end{bmatrix}^T K_v + u(-1)^T K_u + \begin{bmatrix} u_{target}(0) \\ \dots \\ u_{target}(p-1) \end{bmatrix}^T K_{ut} + x(0)^T K_x \right)^T$$

and the model predictive controller sets $\Delta u(k) = z^*_0$, $u(k) = u(k-1) + \Delta u(k)$.

More About

- “Adjusting Disturbance and Noise Models” on page 2-15
- “Time-Varying Weights and Constraints” on page 2-26
- “Terminal Weights and Constraints” on page 2-30

Adjusting Disturbance and Noise Models

A model predictive control requires the following to reject unknown disturbances effectively:

- Application-specific disturbance models
- Measurement feedback to update the controller state estimates

You can modify input and output disturbance models, and the measurement noise model using the MPC Designer app and at the command line. You can then adjust controller tuning weights to improve disturbance rejection.

In this section...

- “Overview” on page 2-15
- “Output Disturbance Model” on page 2-16
- “Measurement Noise Model” on page 2-18
- “Input Disturbance Model” on page 2-20
- “Restrictions” on page 2-22
- “Disturbance Rejection Tuning” on page 2-23

Overview

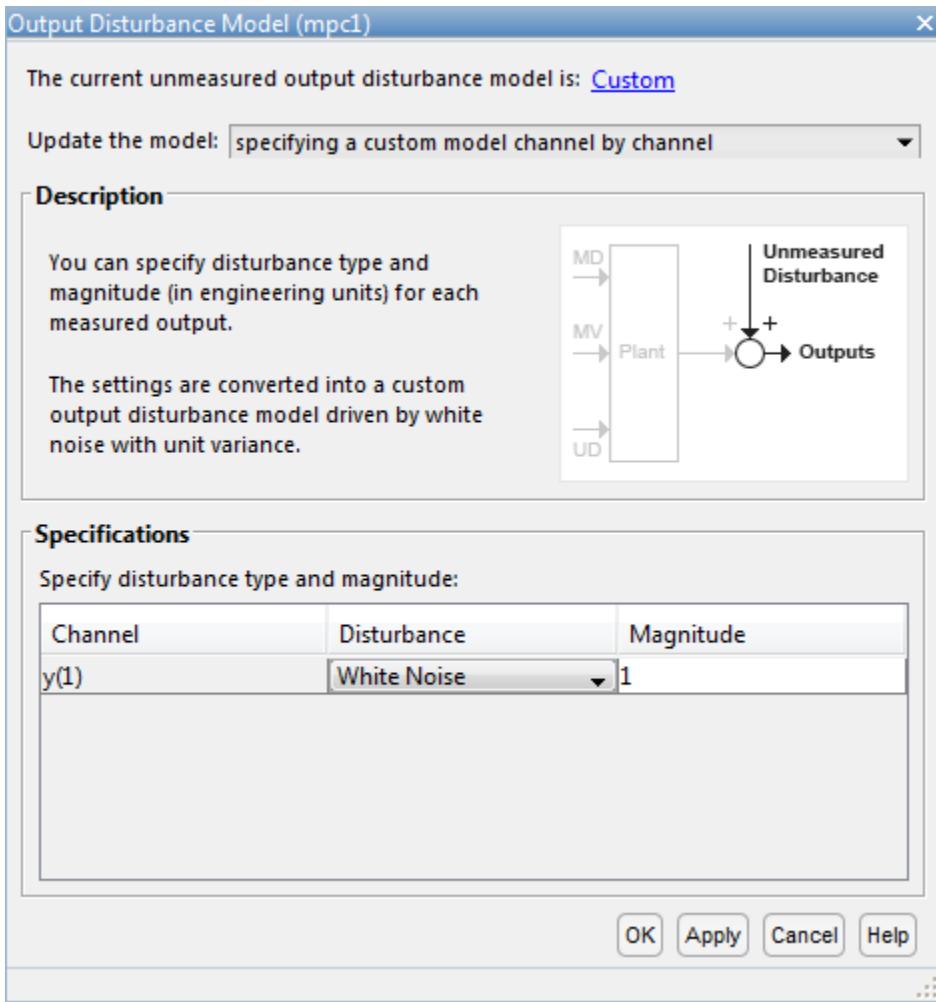
MPC attempts to predict how known and unknown events affect the plant output variables (OVs). Known events are changes in the measured plant input variables (MV and MD inputs). The plant model of the controller predicts the impact of these events, and such predictions can be quite accurate. For more information, see “MPC Modeling”.

The impacts of unknown events appear as errors in the predictions of known events. These errors are, by definition, impossible to predict accurately. However, an ability to anticipate trends can improve disturbance rejection. For example, suppose that the control system has been operating at a near-steady condition with all measured OVs near their predicted values. There are no known events, but one or more of these OVs suddenly deviates from its prediction. The controller disturbance and measurement models allow you to provide guidance on how to handle such errors.

Output Disturbance Model

Suppose that your plant model includes no unmeasured disturbance inputs. The MPC controller then models unknown events using an *output disturbance model*. As shown in “MPC Modeling”, the output disturbance model is independent of the plant, and its output adds directly to that of the plant model.

Using the MPC Designer app, you can specify the type of noise that is expected to affect each plant OV. In the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Estimation Models > Output Disturbance Model**. In the Output Disturbance Model dialog box, in the **Update the model** drop-down list, select **specifying a custom model channel by channel**.



In the **Specifications** section, in the **Disturbance** column, select one of the following disturbance models for each output:

- **White Noise** — Prediction errors are due to random zero-mean white noise. This option implies that the impact of the disturbance is short-lived, and therefore requires a modest, short-term controller response.

- **Random Step-like** — Prediction errors are due to a random step-like disturbance, which lasts indefinitely, maintaining a roughly constant magnitude. Such a disturbance requires a more aggressive, sustained controller response.
- **Random Ramp-like** — Prediction errors are due to a random ramp-like disturbance, which lasts indefinitely and tends to grow with time. Such a disturbance requires an even more aggressive controller response.

Model Predictive Control Toolbox software represents each disturbance type as a model in which white noise, with zero mean and unit variance, enters a SISO dynamic system consisting of one of the following:

- A static gain — For a white noise disturbance
- An integrator in series with a static gain — For a step-like disturbance
- Two integrators in series with a static gain — For a ramp-like disturbance

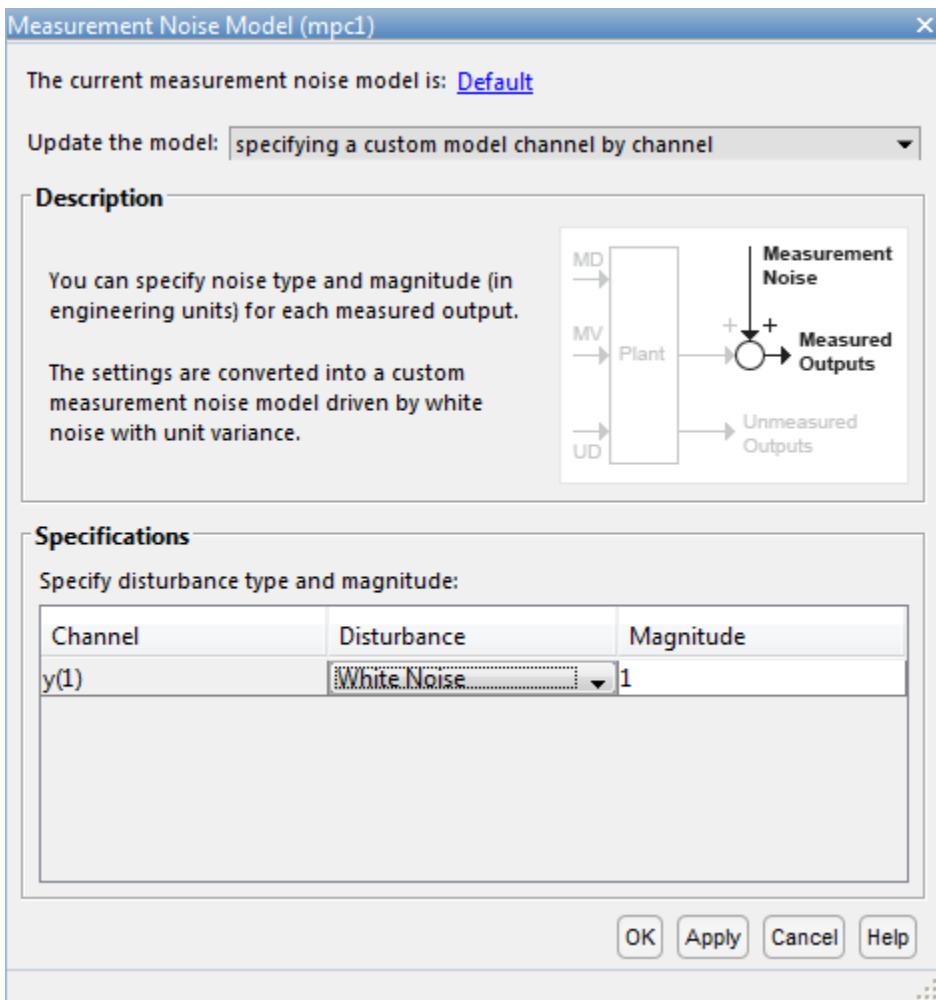
You can also specify the white noise input **Magnitude** for each disturbance model, overriding the assumption of unit variance. As you increase the noise magnitude, the controller responds more aggressively to a given prediction error. The specified noise magnitude corresponds to the static gain in the SISO model for each type of noise.

You can also view or modify the output disturbance model from the command line using `getoutdist` and `setoutdist` respectively.

Measurement Noise Model

MPC also attempts to distinguish disturbances, which require a controller response, from measurement noise, which the controller should ignore. Using the MPC Designer app, you can specify the expected measurement noise magnitude and character. In the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Estimation Models > Measurement Noise Model**. In the Model Noise Model dialog box, in the **Update the model** drop-down list, select **specifying a custom model channel by channel**.

In the **Specifications** section, in the **Disturbance** column, select a noise model for each measured output channel. The noise options are the same as the output disturbance model options.



White Noise is the default option and, in nearly all applications, should provide adequate performance.

When you include a measurement noise model, the controller considers each prediction error to be a combination of disturbance and noise effects. Qualitatively, as you increase the specified noise **Magnitude**, the controller attributes a larger fraction of each prediction error to noise, and it responds less aggressively. Ultimately, the controller

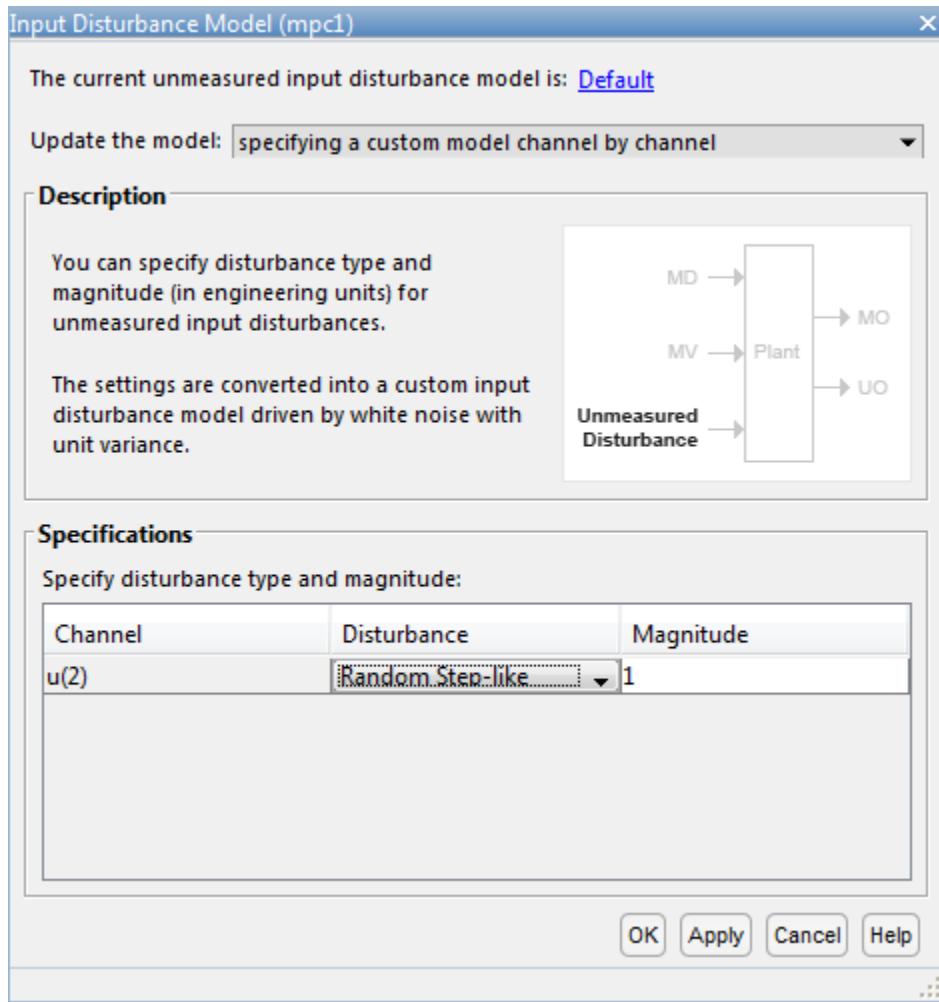
stops responding to prediction errors and only changes its MVs when you change the OV or MV reference signals.

Input Disturbance Model

When your plant model includes unmeasured disturbance (UD) inputs, the controller can use an *input disturbance model* in addition to the standard output disturbance model. The former provides more flexibility and is generated automatically by default. If the chosen input disturbance model does not appear to allow complete elimination of sustained disturbances, an output disturbance model is also added by default.

As shown in “MPC Modeling”, the input disturbance model consists of one or more white noise signals, with unit variance and zero mean, entering a dynamic system. The outputs of this system are the UD inputs to the plant model. In contrast to the output disturbance model, input disturbances affect the plant outputs in a more complex way as they pass through the plant model dynamics.

As with the output disturbance model, you can use the MPC Designer app to specify the type of disturbance you expect for each UD input. In the MPC Designer app, on the **Tuning** tab, in the **Design** section, click **Estimation Models > Input Disturbance Model**. In the Input Disturbance Model dialog box, in the **Update the model** drop-down list, select **specifying a custom model channel by channel**.



In the **Specifications** section, in the **Disturbance** column, select a noise model for each measured output channel. The input disturbance model options are the same as the output disturbance model options.

A common approach is to model unknown events as disturbances adding to the plant MVs. These disturbances, termed *load disturbances* in many texts, are realistic in that some unknown events are failures to set the MVs to the values requested by the controller. You can create a load disturbance model as follows:

- 1 Begin with an LTI plant model, `Plant`, in which all inputs are known (MVs and MDs).

- 2 Obtain the state-space matrices of `Plant`. For example:

```
[A,B,C,D] = ssdata(Plant);
```

- 3 Suppose that there are n_u MVs. Set B_u = columns of B corresponding to the MVs. Also, set D_u = columns of D corresponding to the MVs.

- 4 Redefine the plant model to include n_u additional inputs. For example:

```
Plant.b = [B Bu];  
Plant.d = [D Du]);
```

- 5 To indicate that the new inputs are unmeasured disturbances, use `setmpcsignals`, or set the `Plant.InputGroup` property.

This procedure adds load disturbance inputs without increasing the number of states in the plant model.

By default, given a plant model containing load disturbances, the Model Predictive Control Toolbox software creates an input disturbance model that generates n_{ym} step-like load disturbances. If $n_{ym} > n_u$, it also creates an output disturbance model with integrated white noise adding to $(n_{ym} - n_u)$ measured outputs. If $n_{ym} < n_u$, the last $(n_u - n_{ym})$ load disturbances are zero by default. You can modify these defaults using the MPC Designer app.

You can also view or modify the input disturbance model from the command line using `getindist` and `setindist` respectively.

Restrictions

As discussed in “Controller State Estimation” on page 2-42, the plant, disturbance, and noise models combine to form a state observer, which must be detectable using the measured plant outputs. If not, the software displays a command-window error message when you attempt to use the controller.

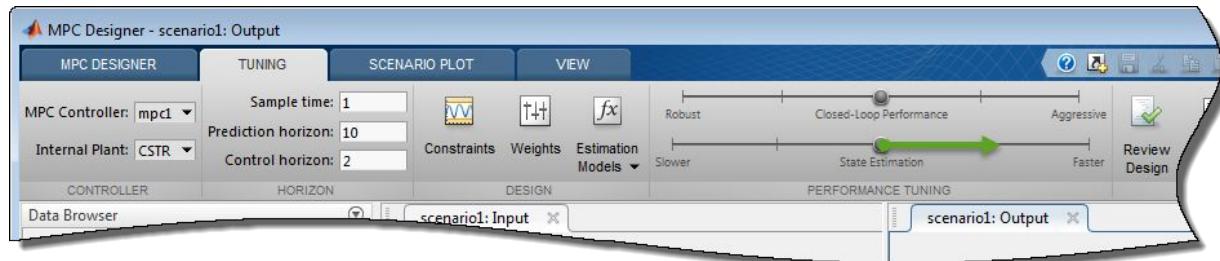
This limitation restricts the form of the disturbance and noise models. If any models are defined as anything other than white noise with a static gain, their model states must be detectable. For example, an integrated white noise disturbance adding to an unmeasured OV would be undetectable. The MPC Designer app prevents you from choosing such a model. Similarly, the number of measured disturbances, n_{ym} , limits the number of step-like UD inputs from an input disturbance model.

By default, the Model Predictive Control Toolbox software creates detectable models. If you modify the default assumptions (or change n_{ym}) and encounter a detectability error, you can revert to the default case.

Disturbance Rejection Tuning

During the design process, you can tune the disturbance rejection properties of the controller.

- 1 Before any controller tuning, define scale factors for each plant input and output variable (see “Specifying Scale Factors” on page 1-2). In the context of disturbance and noise modeling, this makes the default assumption of unit-variance white noise inputs more likely to yield good performance.
- 2 Initially, keep the disturbance models in their default configuration.
- 3 After tuning the cost function weights (see “Tuning Weights” on page 1-16), test your controller response to an unmeasured disturbance input other than a step disturbance at the plant output. Specifically, if your plant model includes UD inputs, simulate a disturbance using one or more of these. Otherwise, simulate one or more load disturbances, that is, a step disturbance added to a designated MV. Both the MPC Designer app and the `sim` command support such simulations.
- 4 If the response in the simulations is too sluggish, try one or more of the following to produce more aggressive disturbance rejection:
 - Increase all disturbance model gains by a multiplicative factor. In the MPC Designer app, do this by increasing the magnitude of each disturbance. If this helps but is insufficient, increase the magnitude further.
 - Decrease the measurement noise gains by a multiplicative factor. In the MPC Designer app, do this by increasing the measurement noise magnitude. If this helps but is insufficient, increase the magnitude further.
 - In the MPC Designer app, in the **Tuning** tab, drag the **State Estimation** slider to the right. Moving towards **Faster** state estimation simultaneously increases the gains for disturbance models and decreases the gains for noise models.



If this helps but is insufficient, drag the slider further to the right.

- Change one or more disturbances to model that requires a more aggressive controller response. For example, change the model from white noise disturbance to a step-like disturbance.

Note: Changing the disturbances in this way adds states to disturbance model, which can cause violations of the state observer detectability restriction.

- 5 If the response is too aggressive, and in particular, if the controller is not robust when its prediction of known events is inaccurate, try reversing the previous adjustments.

See Also

`getindist` | `getoutdist` | `MPC Designer` | `setindist` | `setmpcsignals` | `setoutdist`

Related Examples

- “Design Controller Using MPC Designer”

More About

- “MPC Modeling”
- “Controller State Estimation” on page 2-42

Custom State Estimation

The Model Predictive Control Toolbox software allows the following alternatives to the default state estimation approach:

- You can override the default Kalman gains, L and M . Obtain the default values using `getEstimator`. Then, use `setEstimator` to override those values. These commands assume that the columns of L and M are in the engineering units for the measured plant outputs. Internally, the software converts them to dimensionless form.
- You can use the custom estimation option. This skips all Kalman gain calculations. When the controller operates, at each control interval you must use an external procedure to estimate the controller states, $\mathbf{xc}(k|k)$, providing this to the controller.

Note: You cannot use custom state estimation with the MPC Designer app.

Related Examples

- Using Custom State Estimation

More About

- “Controller State Estimation” on page 2-42

Time-Varying Weights and Constraints

In this section...

[“Time-Varying Weights” on page 2-26](#)

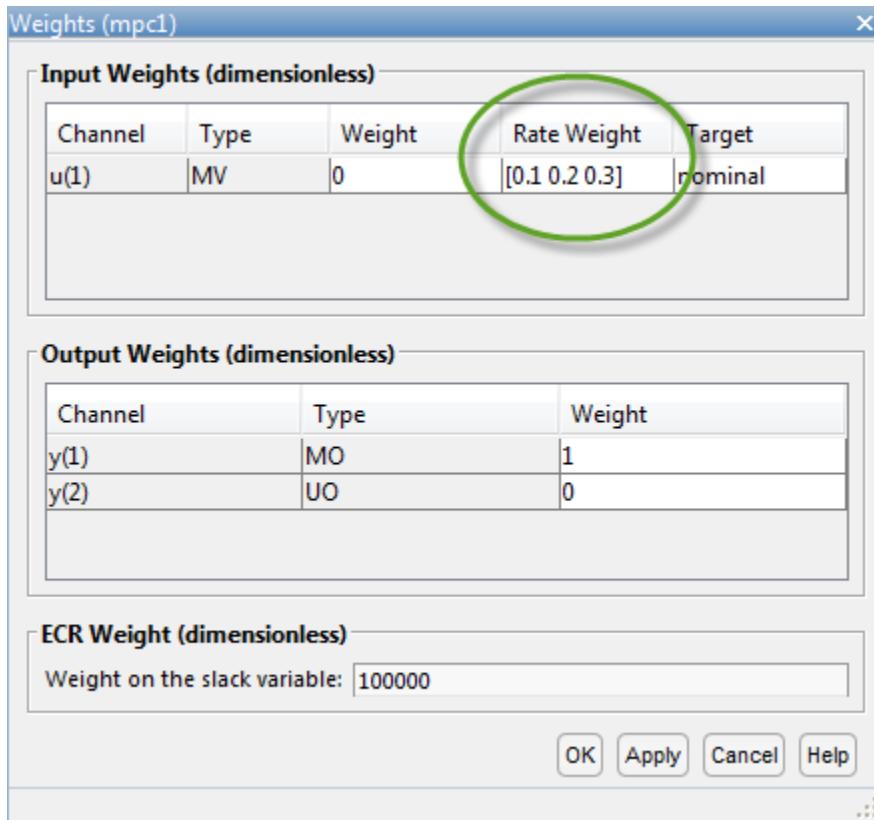
[“Time-Varying Constraints” on page 2-28](#)

Time-Varying Weights

As explained in “Optimization Problem” on page 2-2, the w^y , w^u , and $w^{\Delta u}$ weights can change from one step in the prediction horizon to the next. Such a *time-varying weight* is an array containing p rows, where p is the prediction horizon, and either n_y or n_u columns (number of OVs or MVs).

Using time-varying weights provides additional tuning possibilities. However, it complicates tuning. Recommended practice is to use constant weights unless your application includes unusual characteristics. For example, an application requiring terminal weights must employ time-varying weights. See “Terminal Weights and Constraints” on page 2-30.

You can specify time-varying weights in the MPC Designer app. In the Weights dialog box, specify a time-varying weight as a vector. Each element of the vector corresponds to one step in the prediction horizon. If the length of the vector is less than p , the last weight value applies for the remainder of the prediction horizon.



Note: For any given input channel, you can specify different vector lengths for **Rate Weight** and **Weight**. However, if you specify a time-varying **Weight** for any input channel, you must specify a time-varying **Weight** for all inputs using the same length weight vectors. Similarly, all input **Rate Weight** values must use the same vector length.

Also, if you specify a time-varying **Weight** for any output channel, you must specify a time-varying **Weight** for all output using the same length weight vectors.

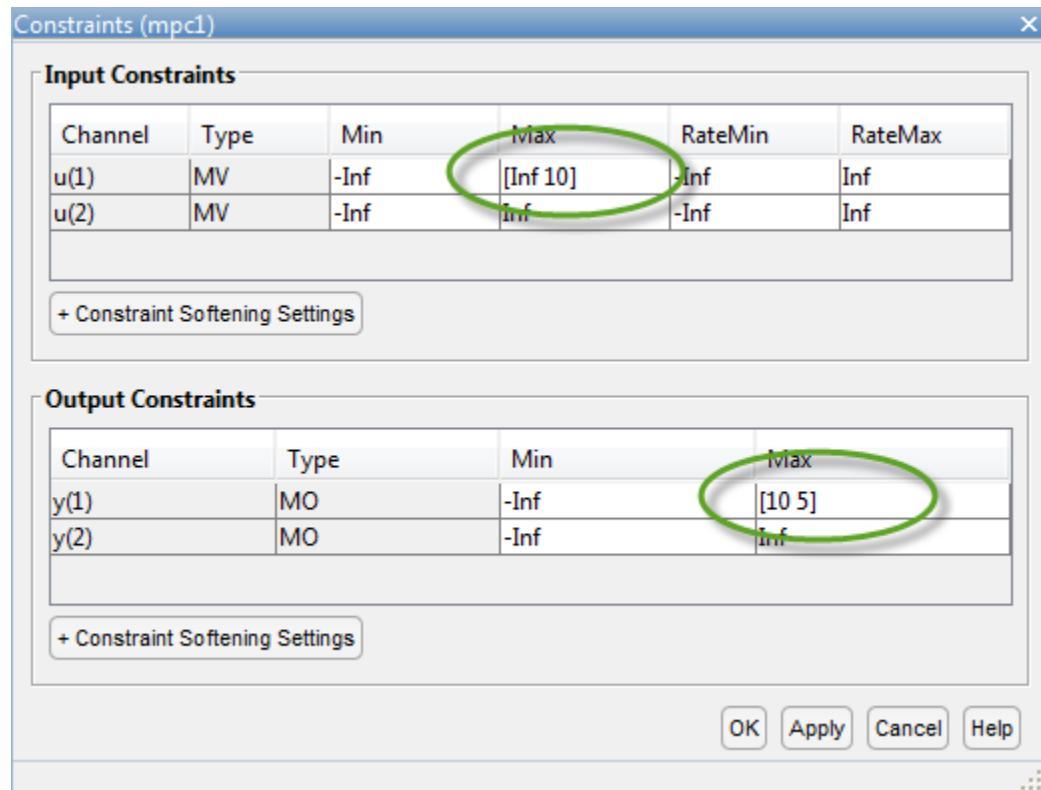
Time-Varying Constraints

When bounding an MV, OV, or MV increment, you can use a different bound value at each prediction-horizon step. To do so, specify the bound as a vector of up to p values, where p is the prediction horizon length (number of control intervals). If you specify $n < p$ values, the n th value applies for the remaining $p - n$ steps.

You can remove constraints at selected steps by specifying `Inf` (or `-Inf`).

If plant delays prevent the MVs from affecting an OV during the first d steps of the prediction horizon and you must include bounds on that OV, leave the OV unconstrained for the first d steps.

You can specify time-varying constraints in the MPC Designer app. In the Constraints dialog box, specify a vector for each time-varying constraint.



Related Examples

- Varying Input and Output Constraints

More About

- “Optimization Problem” on page 2-2
- “Terminal Weights and Constraints” on page 2-30

Terminal Weights and Constraints

Terminal weights are the quadratic weights W_y on $y(t+p)$ and W_u on $u(t+p-1)$. The variable p is the prediction horizon. You apply the quadratic weights at time $k+p$ only, such as the prediction horizon's final step. Using terminal weights, you can achieve infinite horizon control that guarantees closed-loop stability. However, before using terminal weights, you must distinguish between problems with and without constraints.

Terminal constraints are the constraints on $y(t+p)$ and $u(t+p-1)$, where p is the prediction horizon. You can use terminal constraints as an alternative way to achieve closed-loop stability by defining a terminal region.

Note: You can use terminal weights and constraints only at the command line. See [setterminal](#).

For the relatively simple unconstrained case, a terminal weight can make the finite-horizon Model Predictive Controller behave as if its prediction horizon were infinite. For example, the MPC controller behavior is identical to a linear-quadratic regulator (LQR). The standard LQR derives from the cost function:

$$J(u) = \sum_{i=1}^{\infty} x(k+i)^T Q x(k+i) + u(k+i-1)^T R u(k+i-1)$$

where x is the vector of plant states in the standard state-space form:

$$x(k+1) = Ax + Bu(k)$$

The LQR provides nominal stability provided matrices Q and R meet certain conditions. You can convert the LQR to a finite-horizon form as follows:

$$J(u) = \sum_{i=1}^{p-1} [x(k+i)^T Q x(k+i) + u(k+i-1)^T R u(k+i-1)] + x(k+p)^T Q_p x(k+p)$$

where Q_p , the terminal penalty matrix, is the solution of the Riccati equation:

$$Q_p = A^T Q_p A - A^T Q_p B (B^T Q_p B + R)^{-1} B^T Q_p A + Q$$

You can obtain this solution using the `lqr` command in Control System Toolbox™ software.

In general, Q_p is a full (symmetric) matrix. You cannot use the “Standard Cost Function” on page 2-2 to implement the LQR cost function. The only exception is for the first $p - 1$ steps if Q and R are diagonal matrices. Also, you cannot use the “Alternative Cost Function” on page 2-6 because it employs identical weights at each step in the horizon. Thus, by definition, the terminal weight differs from those in steps 1 to $p - 1$. Instead, use the following steps:

- 1** Augment the model (Equation 2-7) to include the weighted terminal states as auxiliary outputs:

$$y_{aug}(k) = Q_c x(k)$$

where Q_c is the Cholesky factorization of Q_p such that $Q_p = Q_c^T Q_c$.

- 2** Define the auxiliary outputs y_{aug} as unmeasured, and specify zero weight to them.
- 3** Specify unity weight on y_{aug} at the last step in the prediction horizon using `setterminal`.

To make the Model Predictive Controller entirely equivalent to the LQR, use a control horizon equal to the prediction horizon. In an unconstrained application, you can use a short horizon and still achieve nominal stability. Thus, the horizon is no longer a parameter to be tuned.

When the application includes constraints, the horizon selection becomes important. The constraints, which are usually softened, represent factors not considered in the LQR cost function. If a constraint becomes active, the control action deviates from the LQR (state feedback) behavior. If this behavior is not handled correctly in the controller design, the controller may destabilize the plant.

For an in-depth discussion of design issues for constrained systems see [1]. Depending on the situation, you might need to include terminal constraints to force the plant states into a defined region at the end of the horizon, after which the LQR can drive the plant signals to their targets. Use `setterminal` to add such constraints to the controller definition.

The standard (finite-horizon) Model Predictive Controller provides comparable performance, if the prediction horizon is long. You must tune the other controller parameters (weights, constraint softening, and control horizon) to achieve this performance.

Tip Robustness to inaccurate model predictions is usually a more important factor than nominal performance in applications.

References

- [1] Rawlings, J. B., and David Q. Mayne “Model Predictive Control: Theory and Design” Nob Hill Publishing, 2010.

See Also

`setterminal`

Related Examples

- “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”
- “Providing LQR Performance Using Terminal Penalty” on page 4-83

Constraints on Linear Combinations of Inputs and Outputs

You can constrain linear combinations of plant input and output variables. For example, you can constrain a particular manipulated variable (MV) to be greater than a linear combination of two other MVs. The general form of such constraints is the following:

$$Eu(k+i|k) + Fy(k+i|k) + Sv(k+i|k) \leq G + \epsilon_k V.$$

- ϵ_k — QP slack variable used for constraint softening (See “Constraint Softening” on page 1-12)
- $u(k+i|k)$ — n_u MV values, in engineering units
- $y(k+i|k)$ — n_y predicted plant outputs, in engineering units
- $v(k+i|k)$ — n_v measured plant disturbance inputs, in engineering units
- E, F, S, G , and V are constants

As with the QP cost function, output prediction using the state observer makes these constraints a function of the QP decision.

Custom constraints are dimensional by default.

You can update custom constraints at run time by calling `setconstraint` before calling `mpcmove`. However, updating the custom constraint matrices at run time is not supported in Simulink.

Note: Using custom constraints is not supported in the MPC Designer app.

See Also

`getconstraint` | `setconstraint`

Related Examples

- Using Custom Input and Output Constraints
- Nonlinear Blending Process with Custom Constraints

More About

- “Optimization Problem” on page 2-2

- “Run-Time Constraint Updating”

Manipulated Variable Blocking

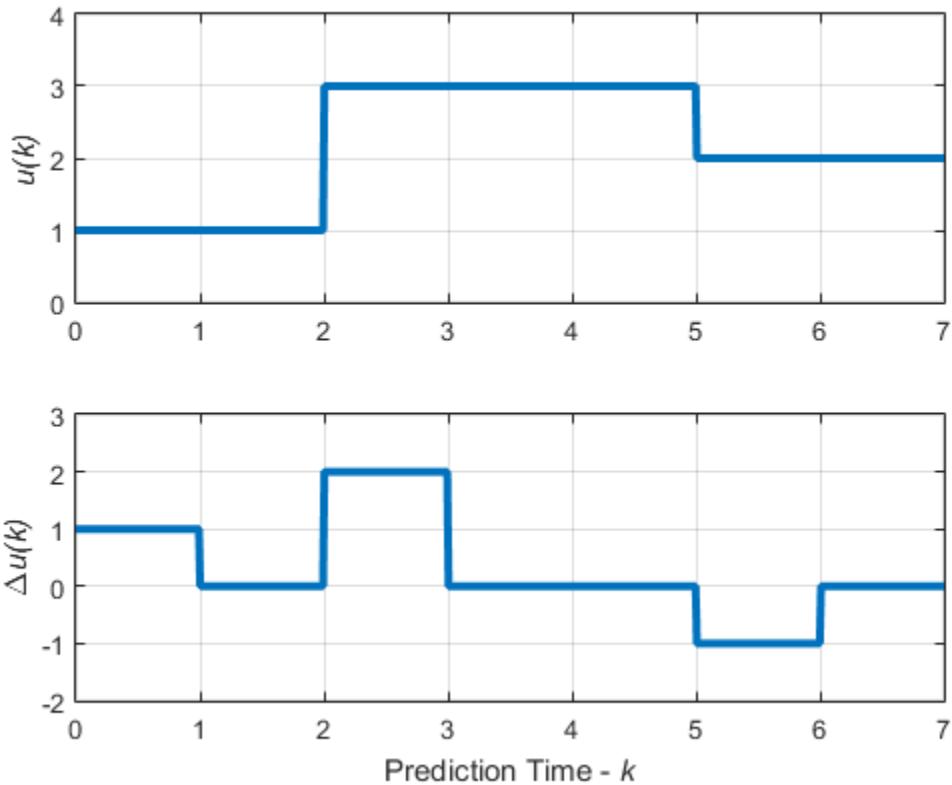
Blocking is an alternative to the simpler control horizon concept (see “Choosing Sample Time and Horizons” on page 1-6). It has many of the same benefits. It also provides more tuning flexibility and potential to smooth MV adjustments. To manipulated variable blocking, you divide the prediction horizon into a series of blocks. The controller then holds the manipulated variable constant within each block.

A recommended approach to blocking is as follows:

- Divide the prediction horizon into 3-5 blocks.
- Try the following alternatives
 - Equal block sizes (one-fifth to one-third of the prediction horizon, p)
 - Block sizes increasing. Example, with $p = 20$, three blocks of duration 3, 7 and 10 intervals.

To use manipulated variable blocking, specify your control horizon as a vector of block sizes. For example, the following figure represent control moves for a control horizon of $p = [2 3 2]$:

2 Model Predictive Control Problem Setup



For each block, the manipulated variable, u , is constant, that is:

- $u(0) = u(1)$
- $u(2) = u(3) = u(4)$
- $u(5) = u(6)$

Test the resulting controller in the same way that you test cost function weights. See “Tuning Weights” on page 1-16.

Related Examples

- “Design MPC Controller for Plant with Delays”

More About

- “Optimization Problem” on page 2-2
- “Tuning Weights” on page 1-16

QP Solver

The model predictive controller QP solver converts an MPC optimization problem to the general form QP problem

$$\underset{x}{\text{Min}} \left(\frac{1}{2} x^T H x + f^T x \right)$$

subject to the linear inequality constraints

$$Ax \geq b$$

where

- x is the solution vector.
- H is the Hessian matrix. This matrix is constant when using implicit MPC without online weight changes.
- A is a matrix of linear constraint coefficients. This matrix is constant when using implicit MPC.
- b and f are vectors.

At the beginning of each control interval, the controller computes H , f , A , and b or, if they are constant, retrieves their precomputed values.

The toolbox uses the KWIK algorithm [1] to solve the QP problem, which requires the Hessian to be positive definite. In the first control step, KWIK uses a *cold start*, in which the initial guess is the unconstrained solution described in “Unconstrained Model Predictive Control” on page 2-13. If x satisfies the constraints, it is the optimal QP solution, x^* , and the algorithm terminates. Otherwise, at least one of the linear inequality constraints must be satisfied as an equality. In this case, KWIK uses an efficient, numerically robust strategy to determine the active constraint set satisfying the standard optimization conditions. In the following control steps, KWIK uses a *warm start*. In this case, the active constraint set determined at the previous control step becomes the initial guess for the next.

Although KWIK is robust, consider the following:

- One or more linear constraints can be violated slightly due to numerical round-off errors. The toolbox employs a nonadjustable relative tolerance. This tolerance allows constraint violations of 10^{-6} times the magnitude of each term. Such violations are considered normal and do not generate warning messages.
- The toolbox also uses a nonadjustable tolerance when testing for an optimal solution.
- The search for the active constraint set is an iterative process. If the iterations reach a problem-dependent maximum, the algorithm terminates. For some controller configurations, the default maximum iterations can be very large, which can make the QP solver appear to stop responding (see “Optimizer”).
- If your problem includes hard constraints, these constraints can be *infeasible* (impossible to satisfy). If the algorithm detects infeasibility, it terminates immediately.

In the last two situations, with an abnormal outcome to the search, the controller retains the last successful control output. For more information, see, the `mpcmove` command. You can detect an abnormal outcome and override the default behavior as you see fit.

Custom QP Application

To access the built-in KWIK solver for applications that require solving online QP problems, use the `mpcqpsolver` command. This option is useful for:

- Advanced MPC applications that are beyond the scope of Model Predictive Control Toolbox software.
- Custom QP applications, including applications that require code generation.

Custom QP Solver

The Model Predictive Control Toolbox enables you to specify a custom QP solver for your MPC controller. This solver is called in place of the built-in `qpkwik` solver at each control interval. This option is useful for:

- Validating your simulation results with a third-party solver.
- Large MPC problems where the built-in KWIK solver runs slowly or fails to find a feasible solution.

To use a custom QP solver, you must:

- Set the `Optimizer.CustomSolver` property of your MPC controller to `true`.
- Provide an `mpcCustomSolver.m` file on the MATLAB® path that solves the general form QP optimization problem. To view a function template, at the MATLAB command line, enter:

```
edit mpcCustomSolver.txt
```

Your custom QP solver function declaration must match the following:

```
function [x,status] = mpcCustomSolver(H,f,A,b,x0)
```

where

- `H` is a Hessian matrix, specified as an n -by- n symmetric positive definite matrix, where n is the number of optimization variables.
- `f` is the multiplier of objective function linear term, specified as a column vector of length n .
- `A` is a matrix of linear inequality constraint coefficients, specified as an m -by- n matrix, where m is the number of constraints.
- `b` is the right-hand side of inequality constraints, specified as a column vector of length m .
- `x0` is an initial guess for the solution, specified as a column vector of length n .
- `x` is the optimal solution, returned as a column vector of length n .
- `status` is a solution validity indicator, returned as an integer according to the following:

Value	Description
> 0	<code>x</code> is optimal. <code>status</code> represents the number of iterations performed during optimization.
0	The maximum number of iterations was reached. The solution, <code>x</code> , may be suboptimal or infeasible.
-1	The problem appears to be infeasible, that is, the constraint $Ax \geq b$ cannot be satisfied.
-2	An unrecoverable numerical error occurred.

For more information, see “Simulate MPC Controller with a Custom QP Solver” on page 4-155.

Note: Code generation is not supported when using a custom QP solver.

References

- [1] Schmid, C. and L.T. Biegler, “Quadratic programming methods for reduced Hessian SQP,” *Computers & Chemical Engineering*, Vol. 18, Number 9, 1994, pp. 817–832.

See Also

`mpc` | `mpcmmove` | `mpcqpsolver`

More About

- “Optimization Problem” on page 2-2
- “Simulate MPC Controller with a Custom QP Solver” on page 4-155

Controller State Estimation

In this section...

- “Controller State Variables” on page 2-42
- “State Observer” on page 2-43
- “State Estimation” on page 2-44
- “Built-in Steady-State Kalman Gains Calculation” on page 2-46
- “Output Variable Prediction” on page 2-47

Controller State Variables

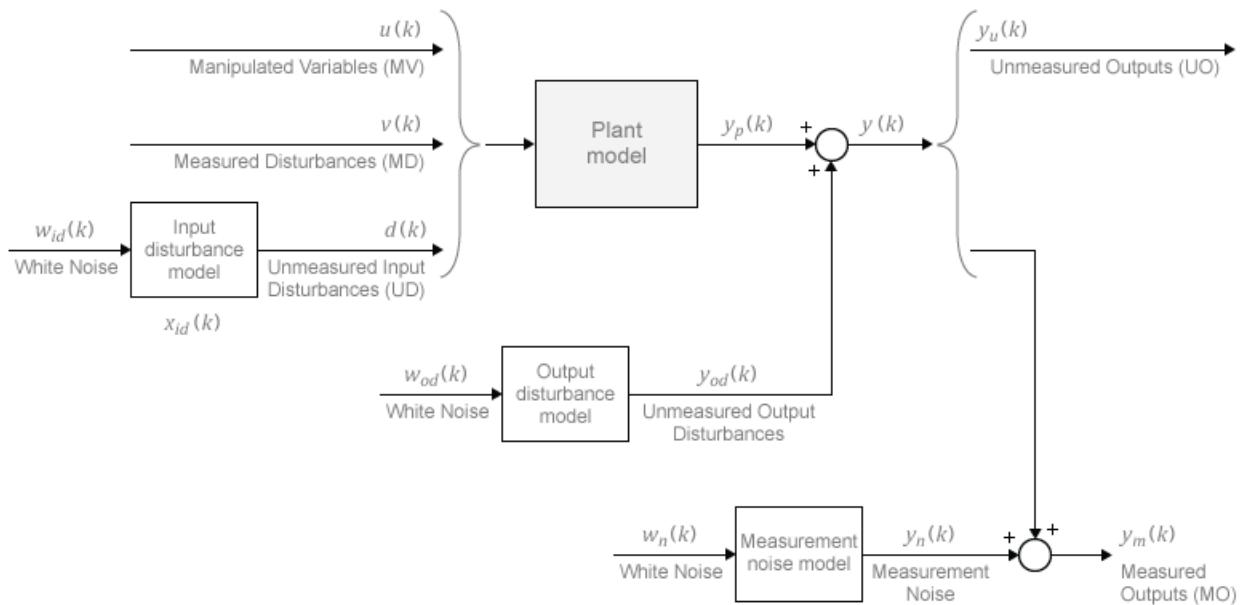
As the controller operates, it uses its current state, x_c , as the basis for predictions. By definition, the state vector is the following:

$$x_c^T(k) = \begin{bmatrix} x_p^T(k) & x_{id}^T(k) & x_{od}^T(k) & x_n^T(k) \end{bmatrix}.$$

Here,

- x_c is the controller state, comprising $n_{xp} + n_{xid} + n_{xod} + n_{xn}$ state variables.
- x_p is the plant model state vector, of length n_{xp} .
- x_{id} is the input disturbance model state vector, of length n_{xid} .
- x_{od} is the output disturbance model state vector, of length n_{xod} .
- x_n is the measurement noise model state vector, of length n_{xn} .

Thus, the variables comprising x_c represent the models appearing in the following diagram of the MPC system.



Some of the state vectors may be empty. If not, they appear in the sequence defined within each model.

By default, the controller updates its state automatically using the latest plant measurements. See “State Estimation” on page 2-44 for details. Alternatively, the custom state estimation feature allows you to update the controller state using an external procedure, and then supply these values to the controller. See “Custom State Estimation” on page 2-25 for details.

State Observer

Combination of the models shown in the diagram yields the state observer:

$$\begin{aligned} x_c(k+1) &= Ax_c(k) + Bu_o(k) \\ y(k) &= Cx_c(k) + Du_o(k). \end{aligned}$$

MPC controller uses the state observer in the following ways:

- To estimate values of unmeasured states needed as the basis for predictions (see “State Estimation” on page 2-44).

- To predict how the controller's proposed manipulated variable (MV) adjustments will affect future plant output values (see "Output Variable Prediction" on page 2-47).

The observer's input signals are the dimensionless plant manipulated and measured disturbance inputs, and the white noise inputs to the disturbance and noise models:

$$u_o^T(k) = [u^T(k) \ v^T(k) \ w_{id}^T(k) \ w_{od}^T(k) \ w_n^T(k)].$$

The observer's outputs are the n_y dimensionless plant outputs.

In terms of the parameters defining the four models shown in the diagram, the observer's parameters are:

$$A = \begin{bmatrix} A_p & B_{pd}C_{id} & 0 & 0 \\ 0 & A_{id} & 0 & 0 \\ 0 & 0 & A_{od} & 0 \\ 0 & 0 & 0 & A_n \end{bmatrix}, \quad B = \begin{bmatrix} B_{pu} & B_{pv} & B_{pd}D_{id} & 0 & 0 \\ 0 & 0 & B_{id} & 0 & 0 \\ 0 & 0 & 0 & B_{od} & 0 \\ 0 & 0 & 0 & 0 & B_n \end{bmatrix},$$

$$C = \begin{bmatrix} C_p & D_{pd}C_{id} & C_{od} & \begin{bmatrix} C_n \\ 0 \end{bmatrix} \end{bmatrix}, \quad D = \begin{bmatrix} 0 & D_{pv} & D_{pd}D_{id} & D_{od} & \begin{bmatrix} D_n \\ 0 \end{bmatrix} \end{bmatrix}.$$

Here, the plant and output disturbance models are resequenced so that the measured outputs precede the unmeasured outputs.

State Estimation

In general, the controller states are unmeasured and must be estimated. By default, the controller uses a steady state Kalman filter that derives from the state observer. (See "State Observer" on page 2-43.)

At the beginning of the k th control interval, the controller state is estimated with the following steps:

- 1 Obtain the following data:

- $x_c(k|k-1)$ — Controller state estimate from previous control interval, $k-1$
- $u^{act}(k-1)$ — Manipulated variable (MV) actually used in the plant from $k-1$ to k (assumed constant)

- $u^{opt}(k-1)$ — Optimal MV recommended by MPC and assumed to be used in the plant from $k-1$ to k
- $v(k)$ — Current measured disturbances
- $y_m(k)$ — Current measured plant outputs
- B_u, B_v — Columns of observer parameter B corresponding to $u(k)$ and $v(k)$ inputs
- C_m — Rows of observer parameter C corresponding to measured plant outputs
- D_{mv} — Rows and columns of observer parameter D corresponding to measured plant outputs and measured disturbance inputs
- L, M — Constant Kalman gain matrices

Plant input and output signals are scaled to be dimensionless prior to use in calculations.

- 2** Revise $x_c(k|k-1)$ when $u^{act}(k-1)$ and $u^{opt}(k-1)$ are different:

$$x_c^{rev}(k|k-1) = x_c(k|k-1) + B_u [u^{act}(k-1) - u^{opt}(k-1)].$$

- 3** Compute the innovation:

$$e(k) = y_m(k) - [C_m x_c^{rev}(k|k-1) + D_{mv} v(k)].$$

- 4** Update the controller state estimate to account for the latest measurements.

$$x_c(k|k) = x_c^{rev}(k|k-1) + M e(k).$$

Then, the software uses the current state estimate $x_c(k|k)$ to solve the quadratic program at interval k . The solution is $u^{opt}(k)$, the MPC-recommended manipulated-variable value to be used between control intervals k and $k+1$.

Finally, the software prepares for the next control interval assuming that the unknown inputs, $w_{id}(k)$, $w_{od}(k)$, and $w_n(k)$ assume their mean value (zero) between times k and $k+1$. The software predicts the impact of the known inputs and the innovation as follows:

$$x_c(k+1|k) = Ax_c^{rev}(k|k-1) + B_u u^{opt}(k) + B_v v(k) + Le(k). \quad \text{2-45}$$

Built-in Steady-State Kalman Gains Calculation

Model Predictive Control Toolbox software uses the `kalman` command to calculate Kalman estimator gains L and M . The following assumptions apply:

- State observer parameters A, B, C, D are time-invariant.
- Controller states, x_c , are detectable. (If not, or if the observer is numerically close to undetectability, the Kalman gain calculation fails, generating an error message.)
- Stochastic inputs $w_{id}(k)$, $w_{od}(k)$, and $w_n(k)$ are independent white noise, each with zero mean and identity covariance.
- Additional white noise $w_u(k)$ and $w_v(k)$ with the same characteristics adds to the dimensionless $u(k)$ and $v(k)$ inputs respectively. This improves estimator performance in certain cases, such as when the plant model is open-loop unstable.

Without loss of generality, set the $u(k)$ and $v(k)$ inputs to zero. The effect of the stochastic inputs on the controller states and measured plant outputs is:

$$\begin{aligned}x_c(k+1) &= Ax_c(k) + Bw(k) \\y_m(k) &= C_m x_c(k) + D_m w(k).\end{aligned}$$

Here,

$$w^T(k) = \begin{bmatrix} w_u^T(k) & w_v^T(k) & w_{id}^T(k) & w_{od}^T(k) & w_n^T(k) \end{bmatrix}.$$

Inputs to the `kalman` command are the state observer parameters A, C_m , and the following covariance matrices:

$$\begin{aligned}Q &= E\{Bw w^T B^T\} = BB^T \\R &= E\{D_m w w^T D_m^T\} = D_m D_m^T \\N &= E\{Bw w^T D_m^T\} = BD_m^T.\end{aligned}$$

Here, $E\{\dots\}$ denotes the expectation.

Output Variable Prediction

Model Predictive Control requires prediction of noise-free future plant outputs used in optimization. This is a key application of the state observer (see “State Observer” on page 2-43).

In control interval k , the required data are as follows:

- p — Prediction horizon (number of control intervals, which is greater than or equal to 1)
- $x_c(k|k)$ — Controller state estimates (see “State Estimation” on page 2-44)
- $v(k)$ — Current measured disturbance inputs (MDs)
- $v(k+i|k)$ — Projected future MDs, where $i=1:p-1$. If you are not using MD previewing, then $v(k+i|k) = v(k)$.
- A, B_u, B_v, C, D_v — State observer constants, where B_u, B_v , and D_v denote columns of the B and D matrices corresponding to inputs u and v . D_u is a zero matrix because of no direct feedthrough

Predictions assume that unknown white noise inputs are zero (their expectation). Also, the predicted plant outputs are to be noise-free. Thus, all terms involving the measurement noise states disappear from the state observer equations. This is equivalent to zeroing the last $n \times n$ elements of $x_c(k|k)$.

Given the above data and simplifications, for the first step the state observer predicts:

$$x_c(k+1|k) = Ax_c(k|k) + B_u u(k|k) + B_v v(k).$$

Continuing for successive steps, $i = 2:p$, the state observer predicts:

$$x_c(k+i|k) = Ax_c(k+i-1|k) + B_u u(k+i-1|k) + B_v v(k+i-1|k).$$

At any step, $i = 1:p$, the predicted noise-free plant outputs are:

$$y(k+i|k) = Cx_c(k+i|k) + D_v v(k+i|k).$$

All of these equations employ dimensionless plant input and output variables. See “Specifying Scale Factors” on page 1-2. The equations also assume zero offsets. Inclusion of nonzero offsets is straightforward.

For faster computations, the MPC controller uses an alternative form of the above equations in which constant terms are computed and stored during controller initialization. See “QP Matrices” on page 2-8.

More About

- “MPC Modeling”
- “Optimization Problem” on page 2-2

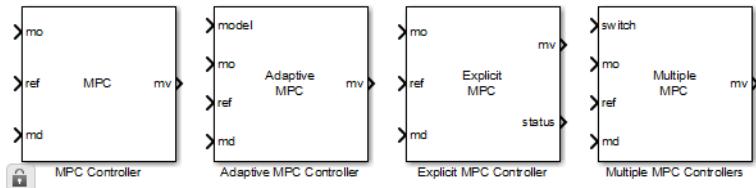
Model Predictive Control Simulink Library

- “MPC Library” on page 3-2
- “Relationship of Multiple MPC Controllers to MPC Controller Block” on page 3-3
- “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5

MPC Library

The MPC Simulink Library provides two blocks you can use to implement MPC control in Simulink, MPC Controller, and Multiple MPC Controllers.

Access the library using the Simulink Library Browser or by typing `mpcLib` at the command prompt. The library contains the following blocks:



MPC Simulink Library

For more information on each block, see their respective block reference pages:

- [MPC Controller](#)
- [Adaptive MPC Controller](#)
- [Explicit MPC Controller](#)
- [Multiple MPC Controllers](#)

Once you have access to the library, you can add one of its blocks to your Simulink model by clicking-and-dragging or copying-and-pasting.

Relationship of Multiple MPC Controllers to MPC Controller Block

The key difference between the **Multiple MPC Controllers** and the **MPC Controller** blocks is the way in which you designate the controllers to be used.

Listing the controllers

You must provide an ordered list containing N names, where N is the number of controllers and each name designates a valid MPC object in your base workspace. Each named controller must use the identical set of plant signals (for example, the same measured outputs and manipulated variables). See the **Multiple MPC Controllers** reference for more information on creating lists.

Designing the controllers

Use your knowledge of the process to identify distinct operating regions and design a controller for each. One convenient approach is to use the Simulink Control Design™ software to calculate each nominal operating point (typically a steady-state condition). Then, obtain a linear prediction model at this condition. To learn more, see the Simulink Control Design documentation. You must have a Simulink Control Design license to use this approach.

After the prediction models have been defined for each operating region, design each corresponding MPC Controller and give it a unique name in your base workspace.

Defining controller switching

Next, define the switching mechanism that will select among the controllers in real time. Add this mechanism to your Simulink model. For example, you could use one or more selected plant measurements to determine when each controller becomes active.

Your mechanism must define a scalar switching signal in the range 1 to N , where N is the number of controllers in your list. Connect this signal to the block's switch import. Set it to 1 when you want the first controller in your list to become active, to 2 when the second is to become active, and so on.

Note: The **Multiple MPC Controllers** block automatically rounds the switching signal to the nearest integer. If the signal is outside the range 1 to N , none of the controllers activate and the block output is zero.

Improving prediction accuracy

During operation, all inactive controllers receive the current manipulated variable and measured output signals so they can update their state estimates. These updates minimize bumps during controller transitions. See “Bumpless Transfer Between Manual and Automatic Operations” on page 4-50 for more information. It is good practice to enable the **Externally supplied MV signal** option and feedback the actual manipulated variables measured in the plant to the `ext.mv` inport. This signal is provided to all the controllers in the block’s list.

Generate Code and Deploy Controller to Real-Time Targets

Model Predictive Control Toolbox provides code generation functionality for controllers designed in Simulink and MATLAB.

Code Generation in Simulink

After designing a controller in Simulink software using the **MPC Controller** block, you can generate code and deploy it for real-time control. You can deploy the controller to all targets supported by the following products:

- Simulink Coder™
- Embedded Coder®
- Simulink PLC Coder™
- Simulink Real-Time™

The sampling rate that a controller can achieve in real-time environment is system-dependent. For example, for a typical small MIMO control application running on Simulink Real-Time, the sampling rate can go as low as 1–10 ms. To determine the sampling rate, first test a less aggressive controller whose sampling rate produces acceptable performance on the target. Next, increase the sampling rate and monitor the execution time used by the controller. You can further decrease the sampling rate as long as the optimization safely completes within each sampling period under the normal plant operations.

For more information, see “Simulation and Code Generation Using Simulink Coder” on page 4-94 and “Simulation and Structured Text Generation Using PLC Coder” on page 4-104.

Note: The **MPC Controller** block is implemented using the MATLAB Function block. To see the structure, right-click the block and select **Mask > Look Under Mask**. Open the **MPC** subsystem underneath.

Code Generation in MATLAB

After designing an MPC controller in MATLAB, you can generate C code using MATLAB Coder and deploy it for real-time control.

To generate code for computing optimal MPC control moves:

- 1 Generate data structures from an MPC or Explicit MPC controller using `getCodeGenerationData`.
- 2 To verify that your controller produces the expected closed-loop results, simulate it using `mpcmovetCodeGeneration` in place of `mpcmove`.
- 3 Generate code for `mpcmovetCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

For more information, see “Generate Code To Compute Optimal MPC Moves in MATLAB” on page 4-108.

See Also

[MPC Controller](#) | [mpcmovetCodeGeneration](#) | [Multiple MPC Controllers](#) | [review](#)

Related Examples

- “Simulation and Code Generation Using Simulink Coder” on page 4-94
- “Simulation and Structured Text Generation Using PLC Coder” on page 4-104
- “Generate Code To Compute Optimal MPC Moves in MATLAB” on page 4-108

Case-Study Examples

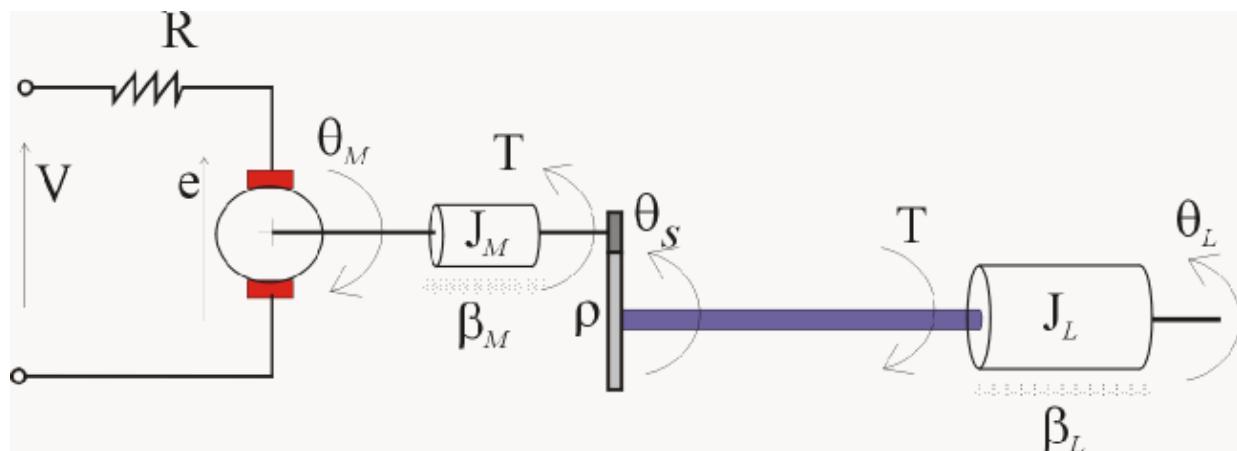
- “Design MPC Controller for Position Servomechanism” on page 4-2
- “Design MPC Controller for Paper Machine Process” on page 4-24
- “Bumpless Transfer Between Manual and Automatic Operations” on page 4-50
- “Switching Controller Online and Offline with Bumpless Transfer” on page 4-58
- “Coordinate Multiple Controllers at Different Operating Points” on page 4-64
- “Use Custom Constraints in Blending Process” on page 4-72
- “Providing LQR Performance Using Terminal Penalty” on page 4-83
- “Real-Time Control with OPC Toolbox” on page 4-89
- “Simulation and Code Generation Using Simulink Coder” on page 4-94
- “Simulation and Structured Text Generation Using PLC Coder” on page 4-104
- “Generate Code To Compute Optimal MPC Moves in MATLAB” on page 4-108
- “Setting Targets for Manipulated Variables” on page 4-116
- “Specifying Alternative Cost Function with Off-Diagonal Weight Matrices” on page 4-120
- “Review Model Predictive Controller for Stability and Robustness Issues” on page 4-125
- “Control of an Inverted Pendulum on a Cart” on page 4-144
- “Simulate MPC Controller with a Custom QP Solver” on page 4-155

Design MPC Controller for Position Servomechanism

This example shows how to design a model predictive controller for a position servomechanism using the MPC Designer app.

System Model

A position servomechanism consists of a DC motor, gearbox, elastic shaft, and load.



The differential equations representing this system are

$$\begin{aligned}\dot{\omega}_L &= -\frac{k_T}{J_L} \left(\theta_L - \frac{\theta_M}{\rho} \right) - \frac{\beta_L}{J_L} \omega_L \\ \dot{\omega}_M &= \frac{k_M}{J_M} \left(\frac{V - k_M \omega_M}{R} \right) - \frac{\beta_M \omega_M}{J_M} + \frac{k_T}{\rho J_M} \left(\theta_L - \frac{\theta_M}{\rho} \right)\end{aligned}$$

where,

- V is the applied voltage.
- T is the torque acting on the load.
- $\omega_L = \dot{\theta}_L$ is the load angular velocity.

- $\omega_M = \dot{\theta}_M$ is the motor shaft angular velocity.

The remaining terms are constant parameters.

Constant Parameters for Servomechanism Model

Symbol	Value (SI Units)	Definition
k_T	1280.2	Torsional rigidity
k_M	10	Motor constant
J_M	0.5	Motor inertia
J_L	$50J_M$	Load inertia
ρ	20	Gear ratio
β_M	0.1	Motor viscous friction coefficient
β_L	25	Load viscous friction coefficient
R	20	Armature resistance

If you define the state variables as

$$x_p = [\theta_L \quad \omega_L \quad \theta_M \quad \omega_M]^T,$$

then you can model the servomechanism as an LTI state-space system.

$$\begin{aligned} \dot{x}_p &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{k_T}{J_L} & -\frac{\beta_L}{J_L} & \frac{k_T}{\rho J_L} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{k_T}{\rho J_M} & 0 & -\frac{k_T}{\rho^2 J_M} & -\frac{\beta_M + \frac{k_M^2}{R}}{J_M} \end{bmatrix} x_p + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{k_M}{R J_M} \end{bmatrix} V \\ \theta_L &= [1 \quad 0 \quad 0 \quad 0] x_p \\ T &= \begin{bmatrix} k_T & 0 & -\frac{k_T}{\rho} & 0 \end{bmatrix} x_p \end{aligned}$$

The controller must set the angular position of the load, θ_L , at a desired value by adjusting the applied voltage, V .

However, since the elastic shaft has a finite shear strength, the torque, T , must stay within the range $|T| \leq 78.5$ Nm. Also, the voltage source physically limits the applied voltage to the range $|V| \leq 220$ V.

Construct Plant Model

Specify the model constants.

```
Kt = 1280.2; % Torsional rigidity
Km = 10; % Motor constant
Jm = 0.5; % Motor inertia
Jl = 50*Jm; % Load inertia
N = 20; % Gear ratio
Bm = 0.1; % Rotor viscous friction
Bl = 25; % Load viscous friction
R = 20; % Armature resistance
```

Define the state-space matrices derived from the model equations.

```
A = [ 0 1 0 0;
      -Kt/Jl -Bl/Jl Kt/(N*Jl) 0;
      0 0 0 1;
      Kt/(Jm*N) 0 -Kt/(Jm*N^2) -(Bm+Km^2/R)/Jm];
B = [0; 0; 0; Km/(R*Jm)];
C = [ 1 0 0 0;
      Kt 0 -Kt/N 0];
D = [0; 0];
```

Create a state-space model.

```
plant = ss(A,B,C,D);
```

Open MPC Designer App

```
mpcDesigner
```

Import Plant and Define Signal Configuration

In the MPC Designer app, in the **MPC Designer** tab, select **MPC Structure**.

In the Define MPC Structure By Importing dialog box, select the **plant** plant model, and assign the plant I/O channels to the following signal types:

- Manipulated variable — Voltage, V
- Measured output — Load angular position, θ_L
- Unmeasured output — Torque, T

Define MPC Structure By Importing

MPC Structure

Select a plant model or an MPC controller from MATLAB Workspace:

Select	Name	Type	Order	Inputs	Outputs
<input checked="" type="radio"/>	plant	ss	4	1	2

Controller Sample Time
Specify MPC controller sample time:

Assign plant i/o channels to desired signal types:

Manipulated variable (MV) channel indices:	<input type="text" value="1"/>
Measured disturbance (MD) channel indices:	<input type="text"/>
Unmeasured disturbance (UD) channel indices:	<input type="text"/>
Measured output (MO) channel indices:	<input type="text" value="1"/>
Unmeasured output (UO) channel indices:	<input type="text" value="2"/>

Refresh workspace Define and Import Cancel Help

Click **Define and Import**.

The MPC Designer app imports the specified plant to the **Data Browser**. The following are also added to the **Data Browser**:

- `mpc1` — Default MPC controller created using `plant` as its internal model.
- `scenario1` — Default simulation scenario. The results of this simulation are displayed in the **Input Response** and **Output Response** plots.

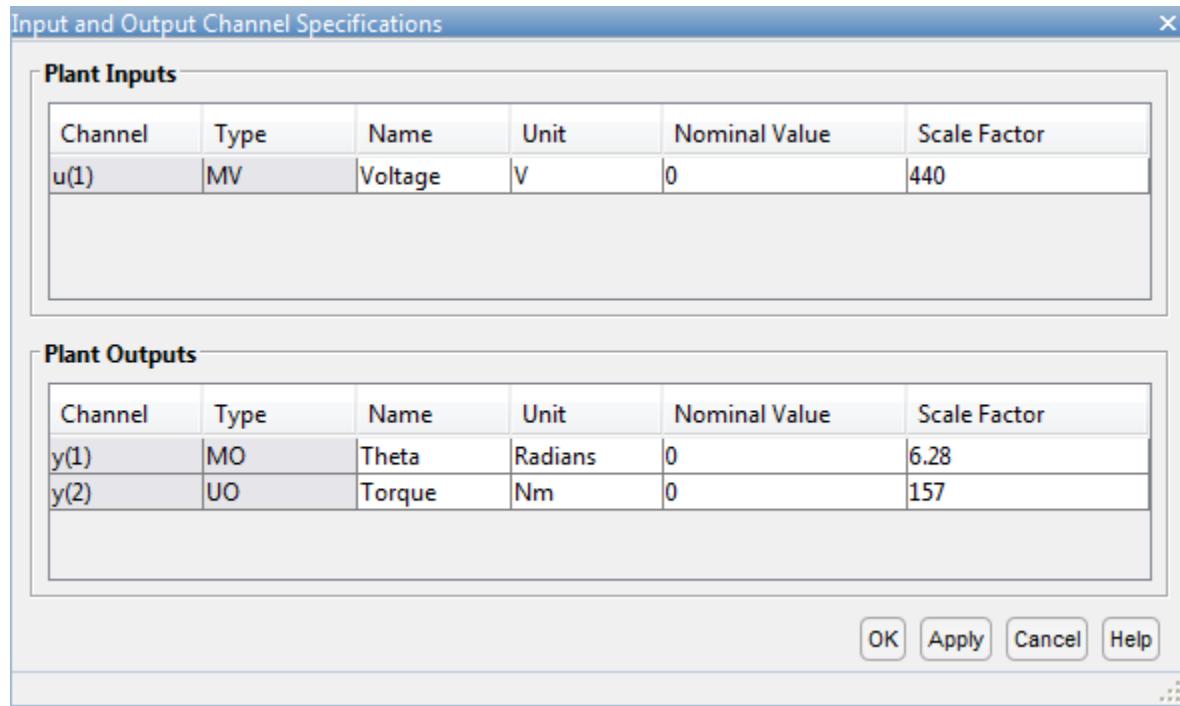
Define Input and Output Channel Attributes

On the **MPC Designer** tab, in the **Structure** section, click **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, for each input and output channel:

- Specify a meaningful **Name** and **Unit**.
- Keep the **Nominal Value** at its default value of 0.
- Specify a **Scale Factor** for normalizing the signal. Select a value that approximates the predicted operating range of the signal:

Channel Name	Minimum Value	Maximum Value	Scale Factor
Voltage	-220 V	220 V	440
Theta	- π radians	π radians	6.28
Torque	-78.5 Nm	78.5 Nm	157



Click **OK** to update the channel attributes and close the dialog box.

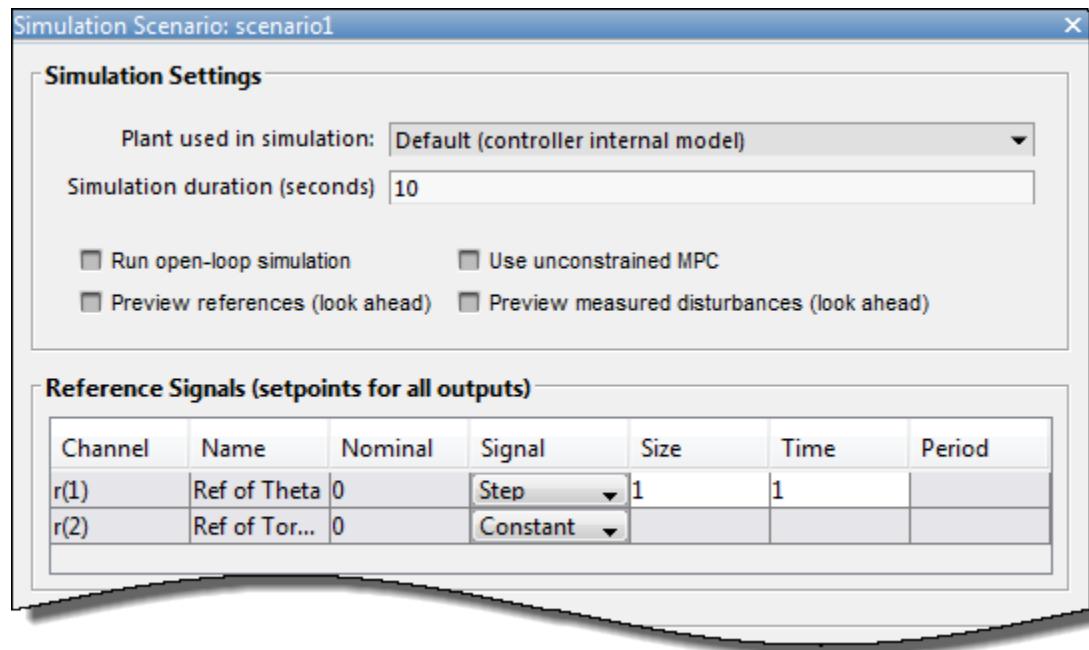
Modify Scenario To Simulate Angular Position Step Response

In the **Scenario** section, **Edit Scenario** drop-down list, select `scenario1` to modify the default simulation scenario.

In the Simulation Scenario dialog box, specify a **Simulation duration** of 10 seconds.

In the **Reference Signals** table, keep the default configuration for the first channel. These settings create a **Step** change of 1 radian in the angular position setpoint at a **Time** of 1 second.

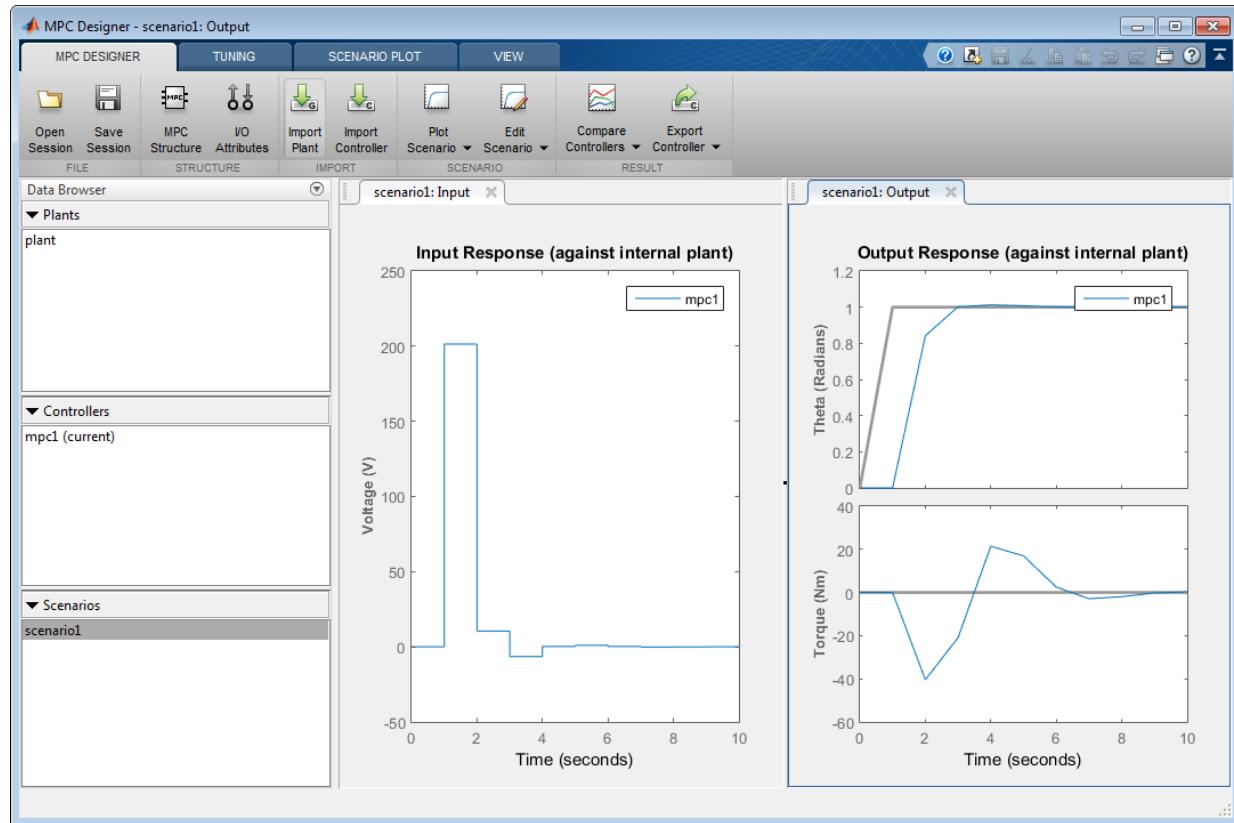
For the second output, in the **Signal** drop-down list, select **Constant** to keep the torque setpoint at its nominal value.



Click **OK**.

The app runs the simulation with the new scenario settings and updates the **Input Response** and **Output Response** plots.

4 Case-Study Examples



Specify Controller Sample Time and Horizons

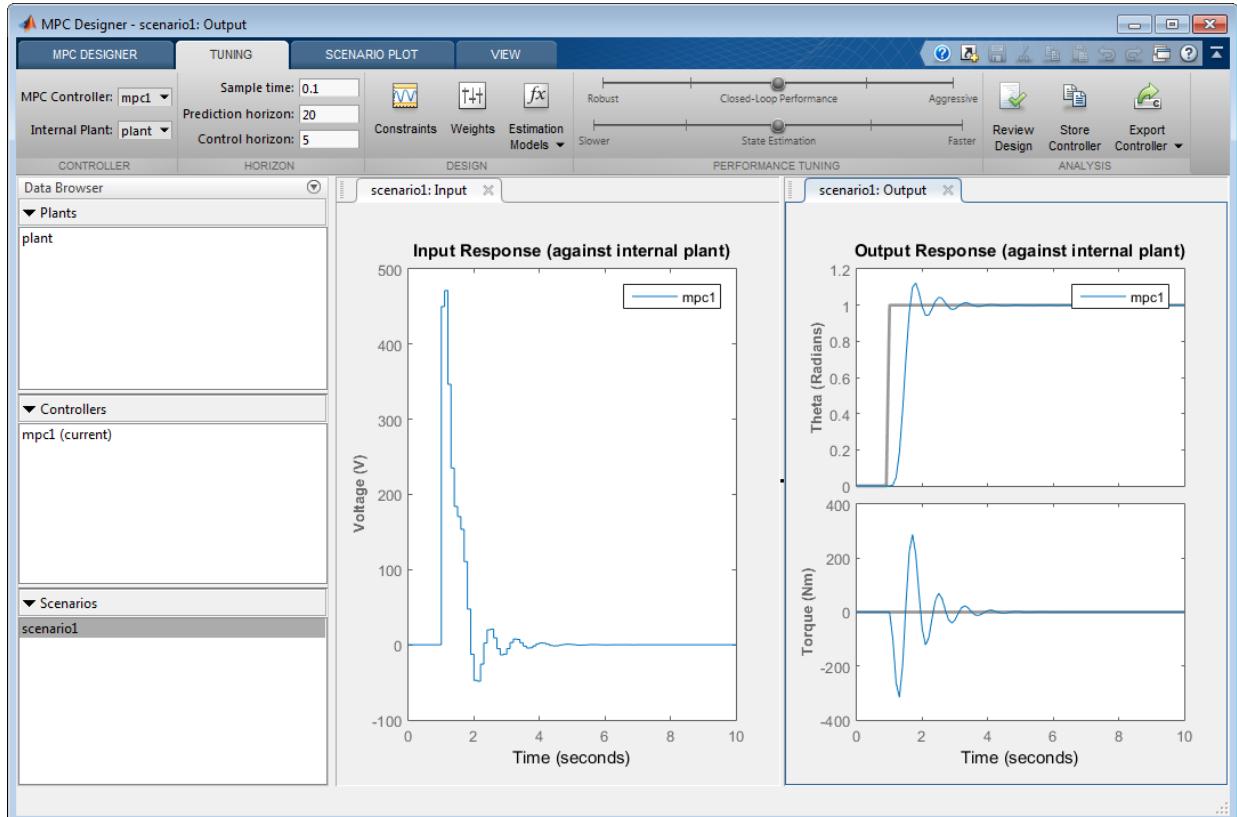
On the **Tuning** tab, in the **Horizon** section, specify a **Sample time** of 0.1 seconds.

For the specified sample time, T_s , and a desired response time of $T_r = 2$ seconds, select a prediction horizon, p , such that:

$$T_r \approx pT_s.$$

Therefore, specify a **Prediction horizon** of 20.

Specify a **Control horizon** of 5.



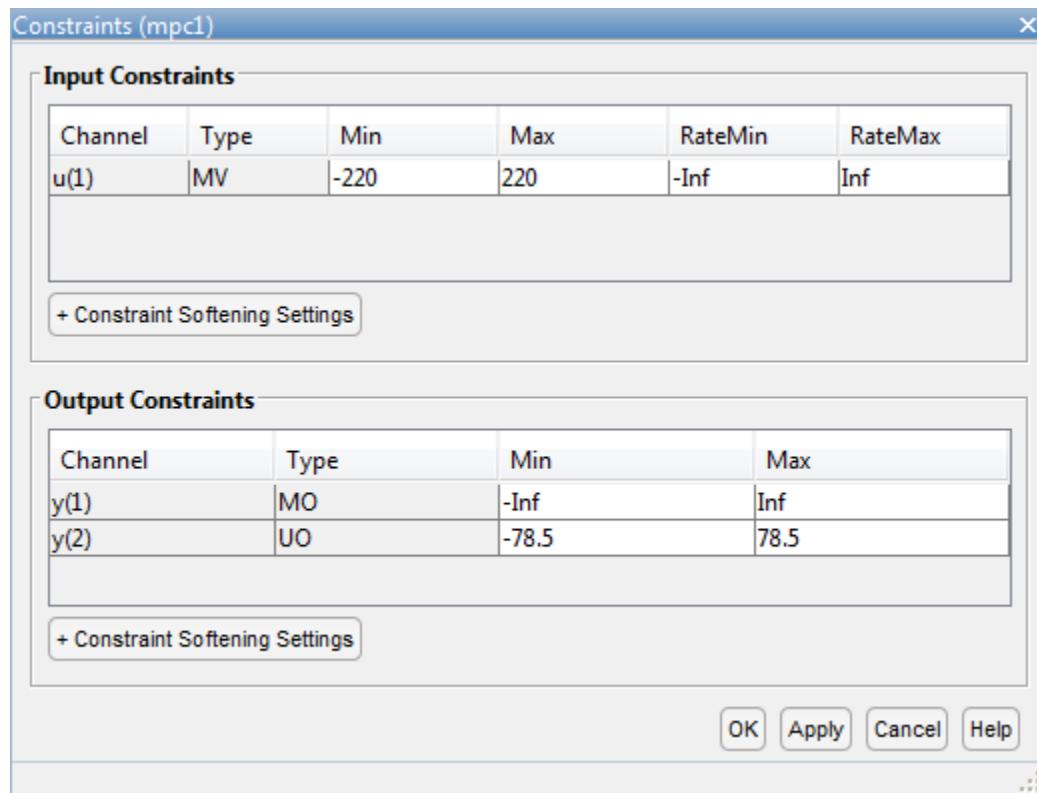
As you update the sample time and horizon values, the **Input Response** and **Output Response** plots update automatically. Both the input voltage and torque values exceed the constraints defined in the system model specifications.

Specify Constraints

In the **Design** section, select **Constraints**.

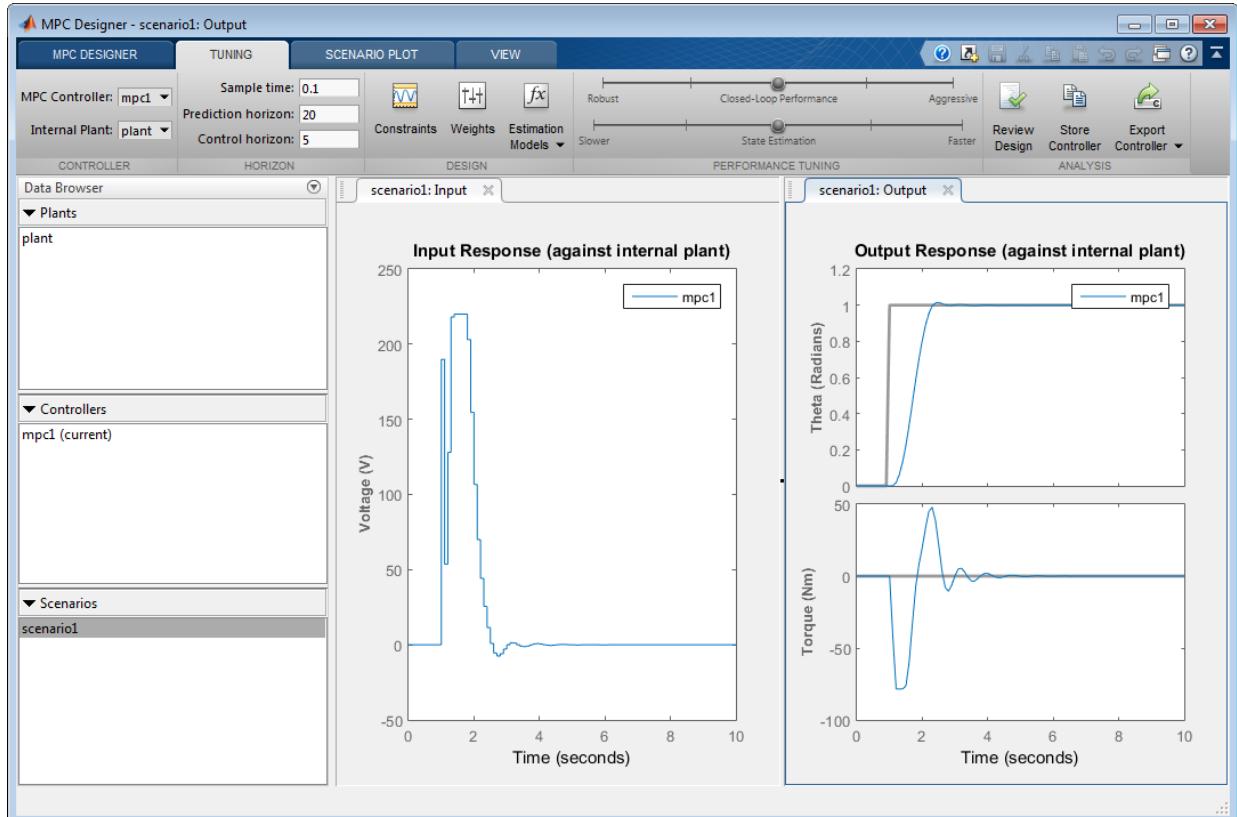
In the Constraints dialog box, in the **Input Constraints** section, specify the **Min** and **Max** voltage values for the manipulated variable (MV).

In the **Output Constraints** section, specify **Min** and **Max** torque values for the unmeasured output (UO).



There are no additional constraints, that is the other constraints remain at their default maximum and minimum values, $-\text{Inf}$ and Inf respectively

Click **OK**.

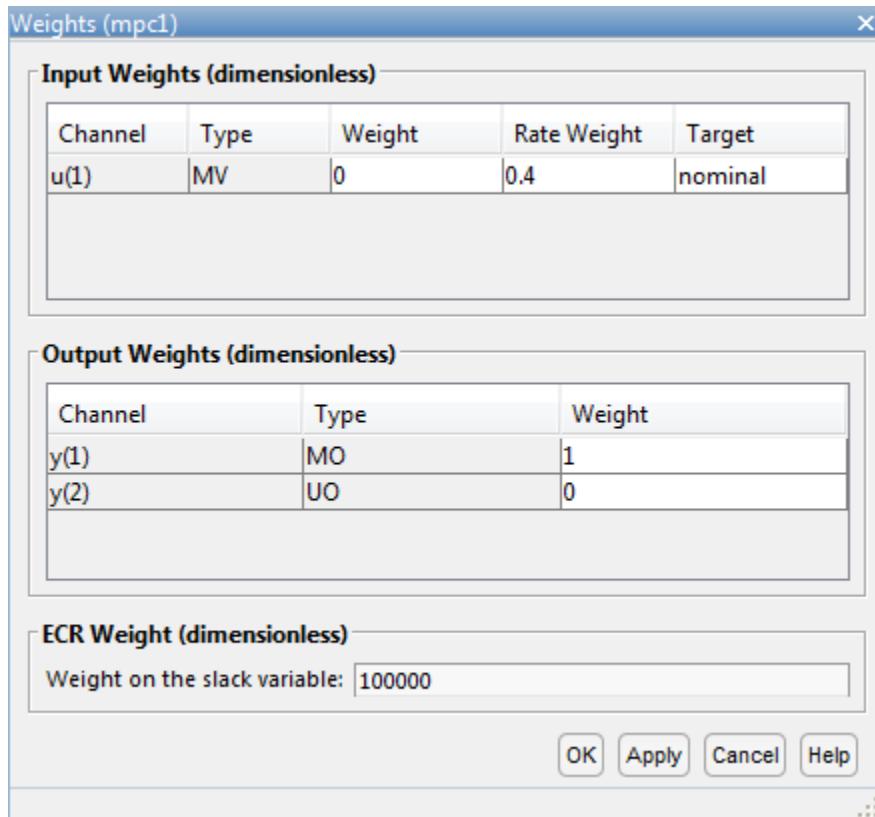


The response plots update to reflect the new constraints. In the **Input Response** plot, there are undesirable large changes in the input voltage.

Specify Tuning Weights

In the **Design** section, select **Weights**.

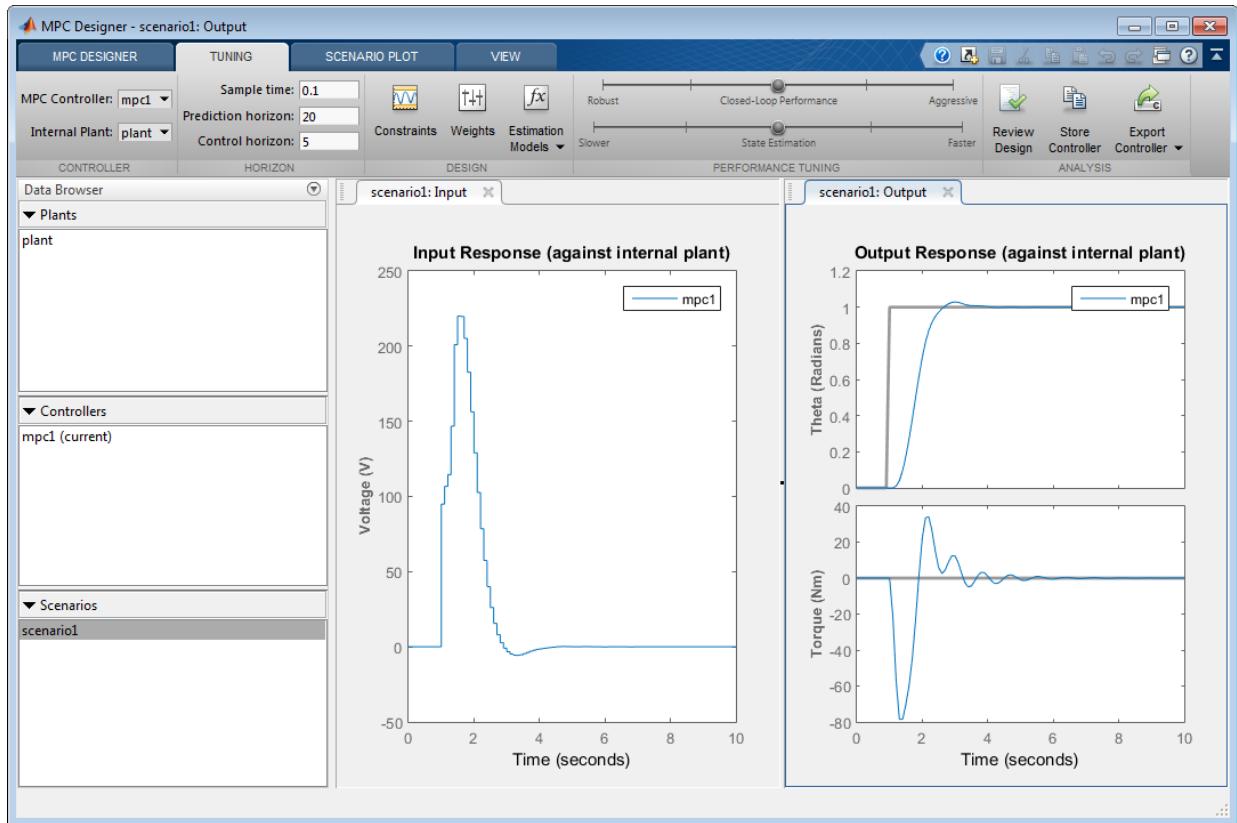
In the Weights dialog box, in the **Input Weights** table, increase the manipulated variable **Rate Weight**.



The tuning **Weight** for the manipulated variable (MV) is 0. This weight indicates that the controller can allow the input voltage to vary within its constrained range. The increased **Rate Weight** limits the size of manipulated variable changes.

Since the control objective is for the angular position of the load to track its setpoint, the tuning **Weight** on the measured output is 1. There is no setpoint for the applied torque, so the controller can allow the second output to vary within its constraints. Therefore, the **Weight** on the unmeasured output (UO) is 0, which enables the controller to ignore the torque setpoint.

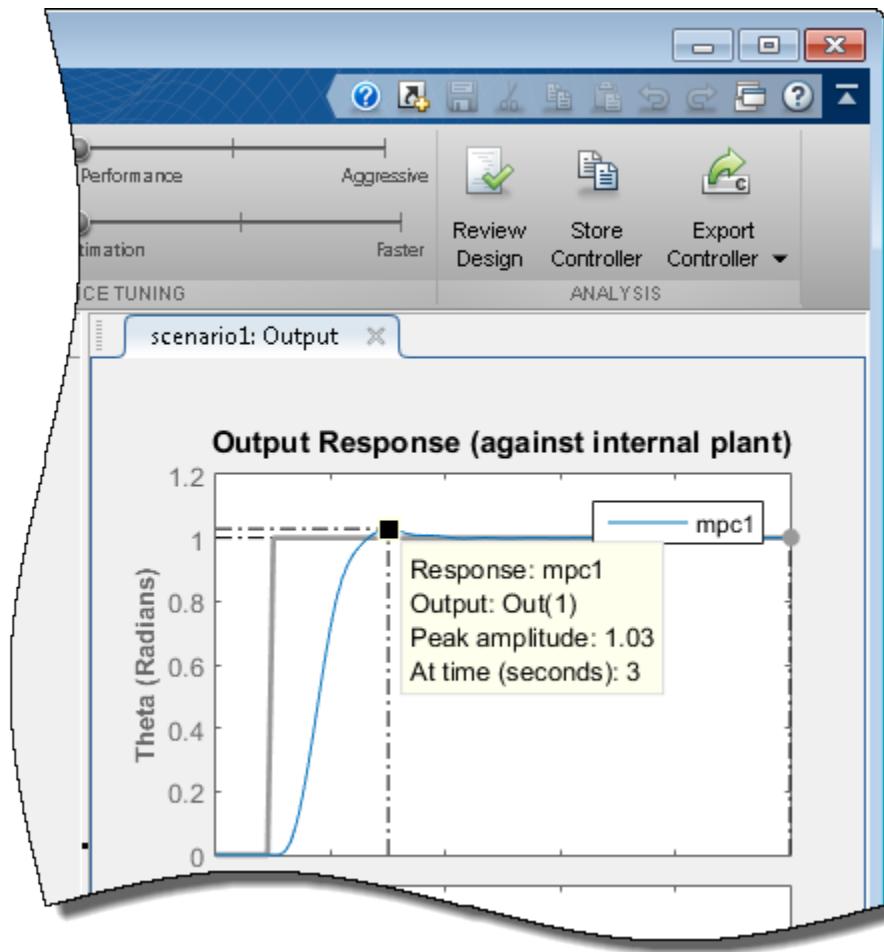
Click **OK**.



The response plots update to reflect the increased rate weight. The **Input Response** is smoother with smaller voltage changes.

Examine Output Response

In the **Output Response** plot, right-click the **Theta** plot area, and select **Characteristics > Peak Response**.

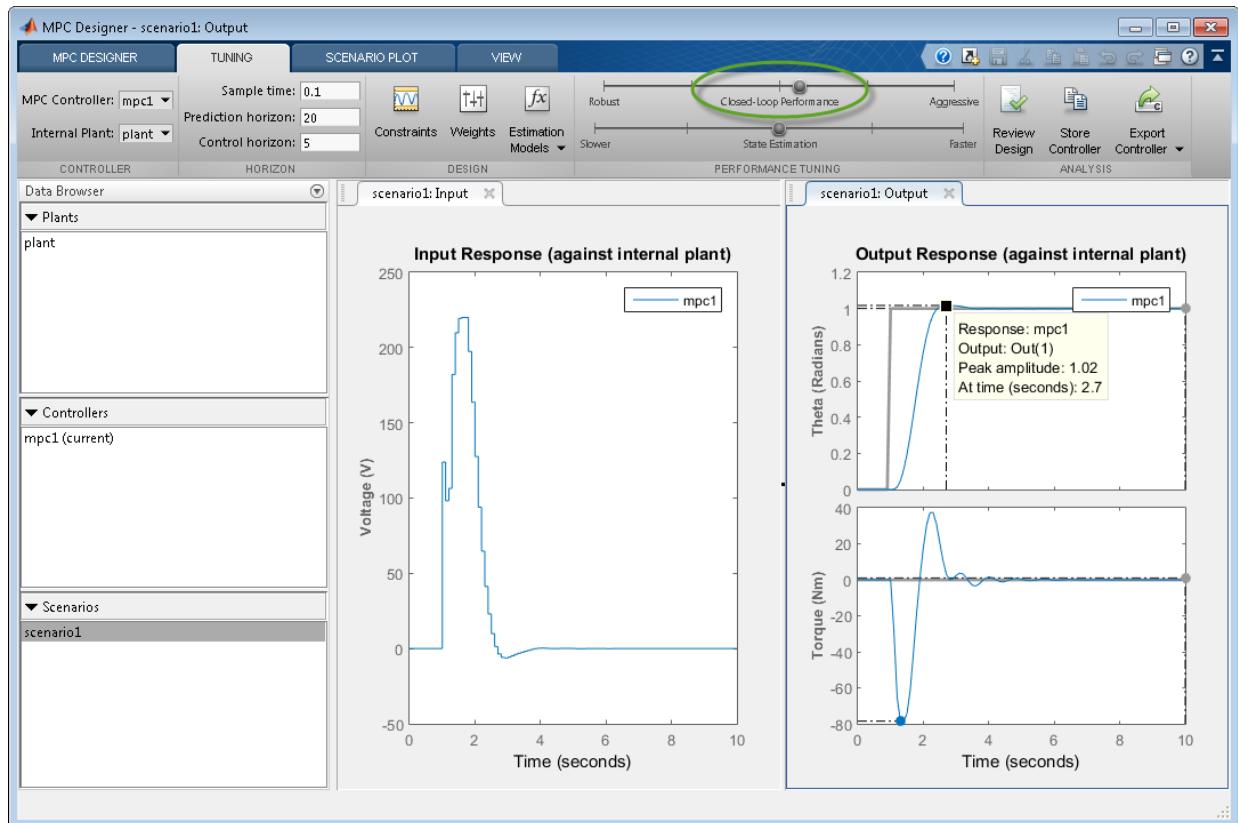


The peak output response occurs at time of 3 seconds with a maximum overshoot of 3%. Since the reference signal step change is at 1 second, the controller has a peak time of 2 seconds.

Improve Controller Response Time

Click and drag the **Closed-Loop Performance** slider to the right to produce a more **Aggressive** response. The further you drag the slider to the right, the faster the

controller responds. Select a slider position such that the peak response occurs at 2.7 seconds.



The final controller peak time is 1.7 seconds. Reducing the response time further results in overly-aggressive input voltage changes.

Generate and Run MATLAB Script

In the **Analysis** section, click the **Export Controller** arrow ▾.

Under **Export Controller**, click **Generate Script**.

In the Generate MATLAB Script dialog box, check the box next to **scenario1**.

Click **Generate Script**.

The app exports a copy of the plant model, `plant_C`, to the MATLAB workspace, along with simulation input and reference signals.

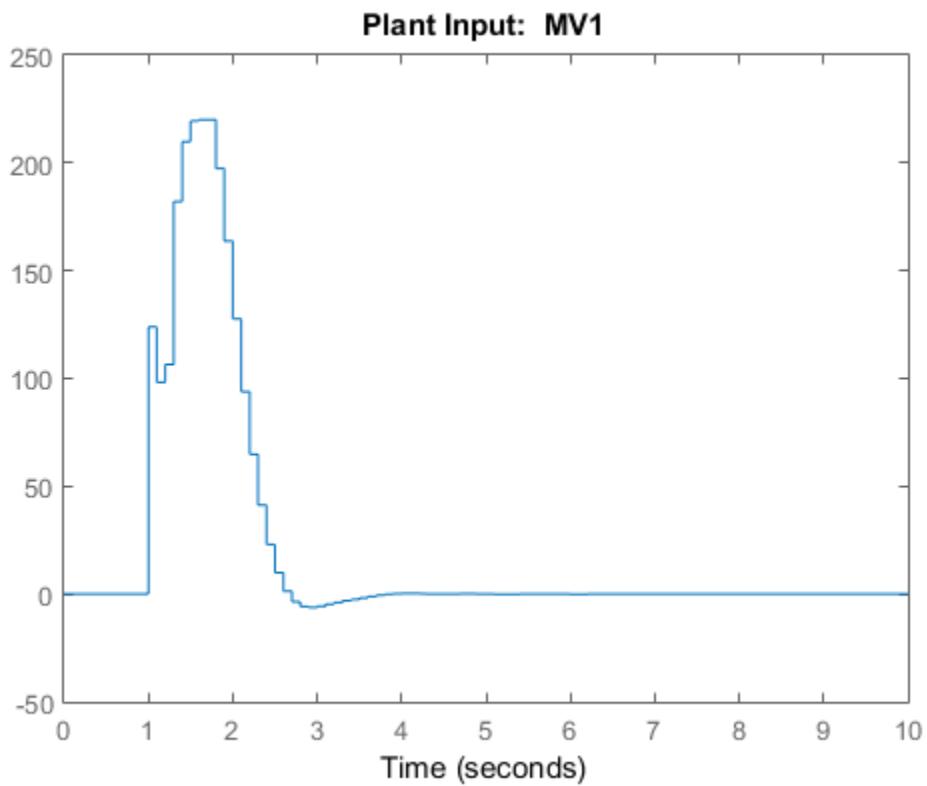
Additionally, the app generates the following code in the MATLAB Editor.

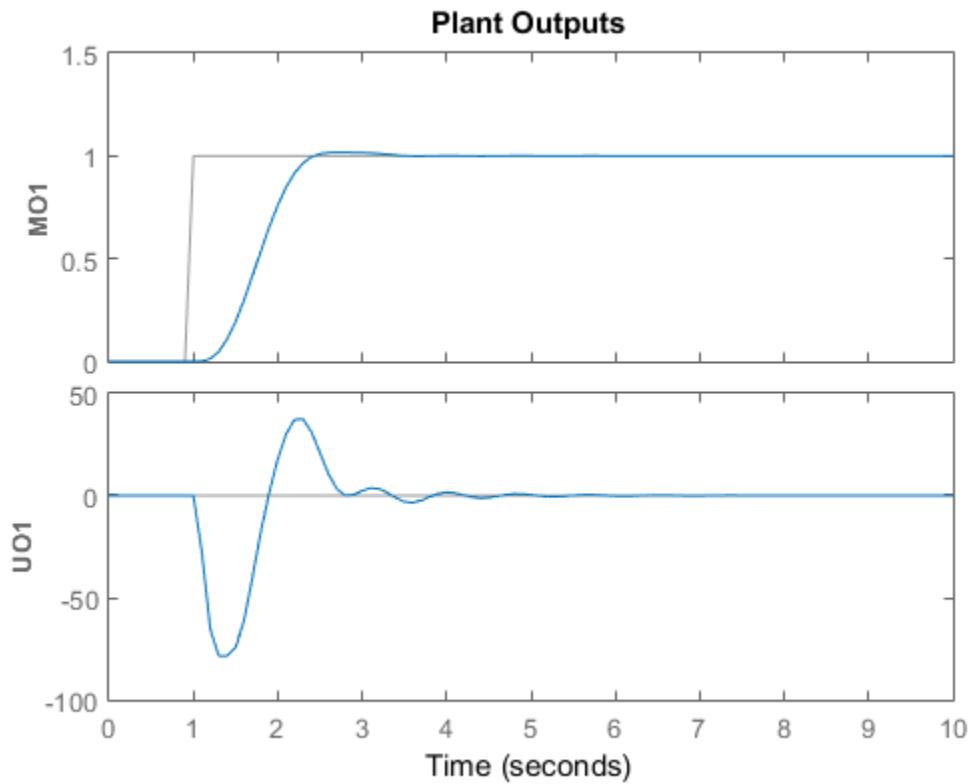
```
%> create MPC controller object with sample time
mpc1 = mpc(plant_C, 0.1);
%> specify prediction horizon
mpc1.PredictionHorizon = 20;
%> specify control horizon
mpc1.ControlHorizon = 5;
%> specify nominal values for inputs and outputs
mpc1.Model.Nominal.U = 0;
mpc1.Model.Nominal.Y = [0;0];
%> specify scale factors for inputs and outputs
mpc1.MV(1).ScaleFactor = 440;
mpc1.OV(1).ScaleFactor = 6.28;
mpc1.OV(2).ScaleFactor = 157;
%> specify constraints for MV and MV Rate
mpc1.MV(1).Min = -220;
mpc1.MV(1).Max = 220;
%> specify constraints for OV
mpc1.OV(2).Min = -78.5;
mpc1.OV(2).Max = 78.5;
%> specify overall adjustment factor applied to weights
beta = 1.2712;
%> specify weights
mpc1.Weights.MV = 0*beta;
mpc1.Weights.MVRate = 0.4/beta;
mpc1.Weights.OV = [1 0]*beta;
mpc1.Weights.ECR = 100000;
%> specify simulation options
options = mpcsimopt();
options.RefLookAhead = off ;
options.MDLookAhead = off ;
options.Constraints = on ;
options.OpenLoop = off ;
%> run simulation
sim(mpc1, 101, mpc1_RefSignal, mpc1_MDSignal, options);
```

In the MATLAB Window, in the **Editor** tab, select **Save**.

Complete the Save dialog box and then click **Save**.

In the **Editor** tab, click **Run**.



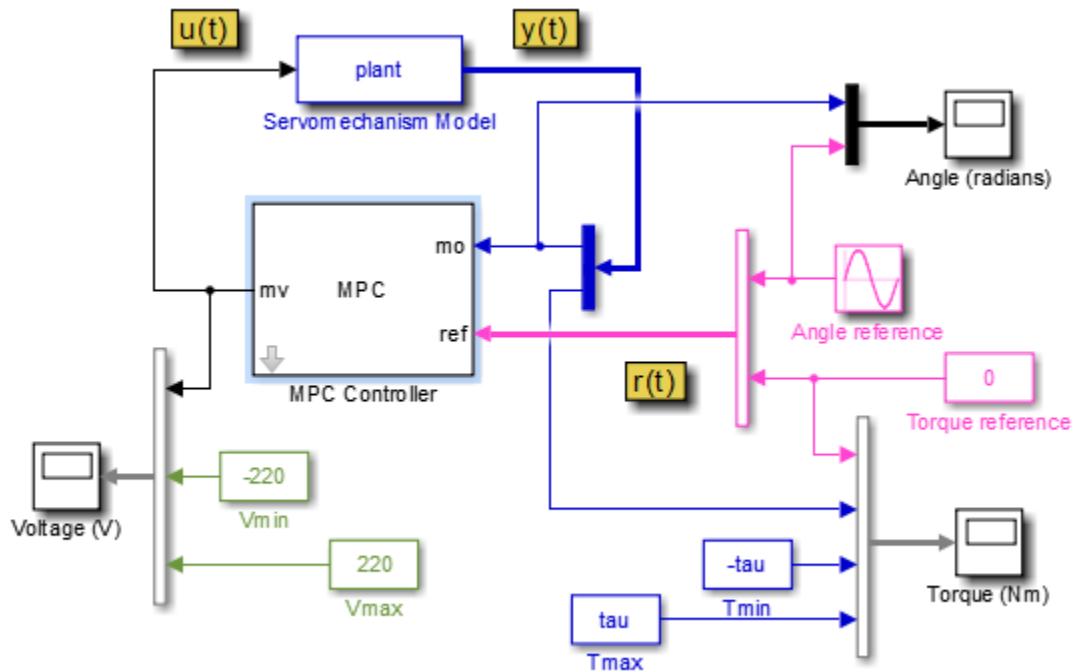


The script creates the controller, `mpc1`, and runs the simulation scenario. The input and output responses match the simulation results from the app.

Validate Controller Performance In Simulink

Open the servomechanism Simulink model.

```
open_system( 'mpc_motor' );
```

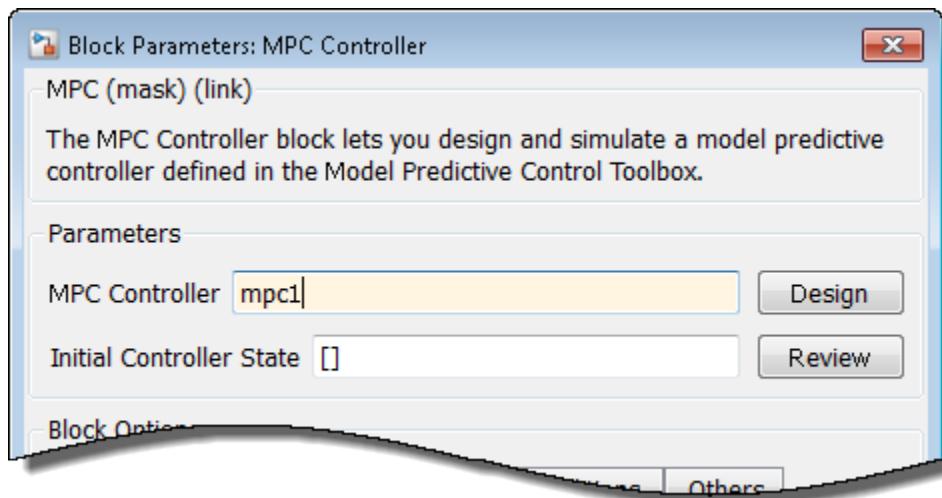


This model uses an **MPC Controller** block to control a servomechanism plant. The **Servomechanism Model** block is already configured to use the **plant** model from the MATLAB workspace.

The **Angle reference** source block creates a sinusoidal reference signal with a frequency of 0.4 rad/sec and an amplitude of π .

Double-click the **MPC Controller** block.

In the MPC Controller Block Parameters dialog box, specify an **MPC Controller** from the MATLAB workspace. Use the `mpc1` controller created using the generated script.



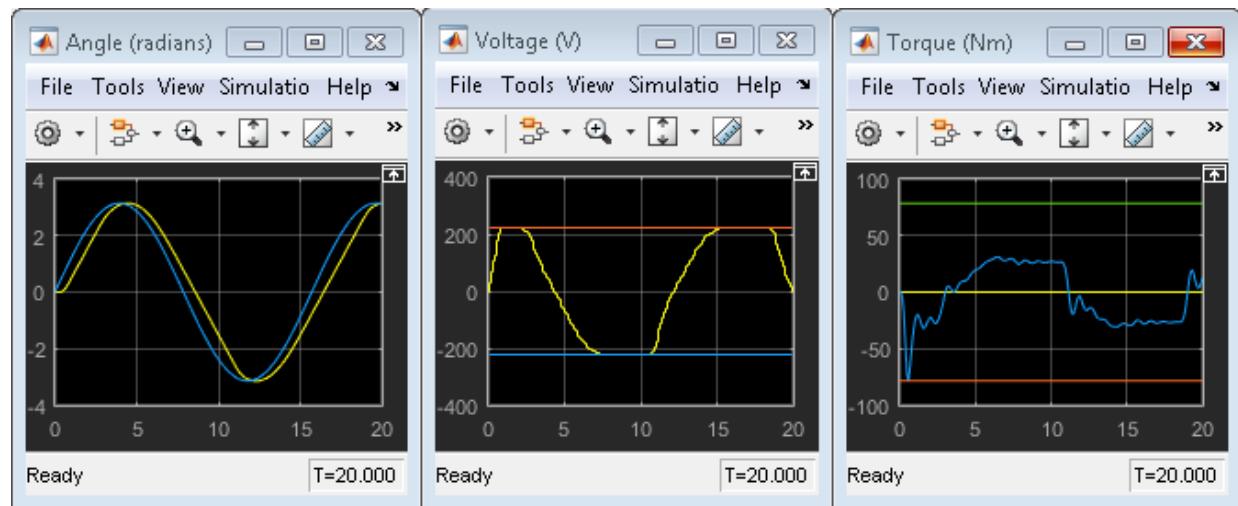
Click **OK**.

At the MATLAB command line, specify a torque magnitude constraint variable.

```
tau = 78.5;
```

The model uses this value to plot the constraint limits on the torque output scope.

In the Simulink model window, click **Run** to simulate the model.



In the **Angle** scope, the output response, yellow, tracks the angular position setpoint, blue, closely.

See Also

[mpc](#) | [MPC Controller](#) | [MPC Designer](#)

Related Examples

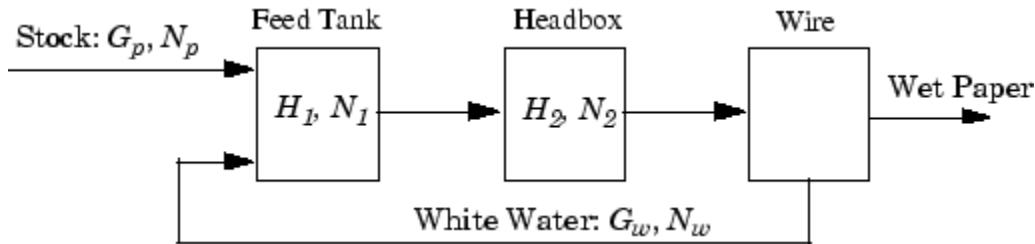
- “Design Controller Using MPC Designer”
- “Design MPC Controller at the Command Line”

Design MPC Controller for Paper Machine Process

This example shows how to design a model predictive controller for a nonlinear paper machine process using the MPC Designer app.

System Model

Ying *et al.* studied the control of consistency (percentage of pulp fibers in aqueous suspension) and liquid level in a paper machine headbox.



The process is nonlinear and has three outputs, two manipulated inputs, and two disturbance inputs, one of which is measured for feedforward control.

The process model is a set of ordinary differential equations (ODEs) in bilinear form. The states are

$$x = [H_1 \quad H_2 \quad N_1 \quad N_2]^T$$

- H_1 — Feed tank liquid level
- H_2 — Headbox liquid level
- N_1 — Feed tank consistency
- N_2 — Headbox consistency

The primary control objective is to hold H_2 and N_2 at their setpoints by adjusting the manipulated variables:

- G_p — Flow rate of stock entering the feed tank
- G_w — Flow rate of recycled white water

The consistency of stock entering the feed tank, N_p , is a measured disturbance, and the white water consistency, N_w , is an unmeasured disturbance.

All signals are normalized with zero nominal steady-state values and comparable numerical ranges. The process is open-loop stable.

The measured outputs are H_2 , N_1 , and N_2 .

The Simulink S-function, `mpc_pmmode1` implements the nonlinear model equations. To view this S-function, enter the following.

```
edit mpc_pmmode1
```

Construct Plant Model

To design a controller for a nonlinear plant using MPC Designer, you must first obtain a linear model of the plant. The paper machine headbox model can be linearized analytically.

At the MATLAB command line, enter the state-space matrices for the linearized model.

```
A = [-1.9300      0      0      0
      0.3940   -0.4260      0      0
              0      0   -0.6300      0
      0.8200   -0.7840   0.4130  -0.4260];
B = [1.2740    1.2740      0      0
      0      0      0      0
      1.3400   -0.6500   0.2030  0.4060
      0      0      0      0];
C = [0    1.0000      0      0
      0      0    1.0000      0
      0      0      0    1.0000];
D = zeros(3,4);
```

Create a continuous-time LTI state-space model.

```
PaperMach = ss(A,B,C,D);
```

Specify the names of the input and output channels of the model.

```
PaperMach.InputName = { G_p , G_w , N_p , N_w };
PaperMach.OutputName = { H_2 , N_1 , N_2 };
```

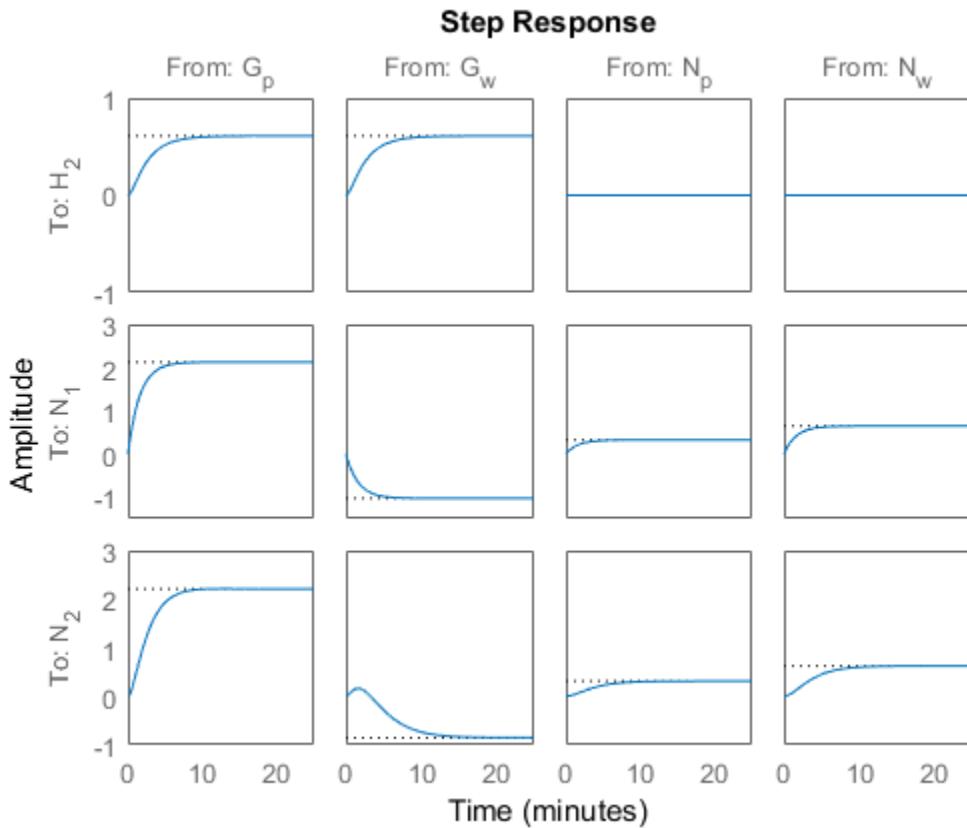
Specify the model time units.

```
PaperMach.TimeUnit = minutes ;
```

Plot Linear Model Step Response

Examine the open-loop response of the plant.

```
step(PaperMach);
```



The step response shows that:

- Both manipulated variables, G_p and G_w , affect all three outputs.
- The manipulated variables have nearly identical effects on H_2 .
- The response from G_w to N_2 is an inverse response.

These features make it difficult to achieve accurate, independent control of H_2 and N_2 .

Open MPC Designer App

```
mpcDesigner
```

Import Plant Model and Define Signal Configuration

In the MPC Designer app, on the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

In the Define MPC Structure By Importing dialog box, select the **PaperMach** plant model and assign the plant I/O channels to the following signal types:

- Manipulated variables — G_p and G_w
- Measured disturbance — N_p
- Unmeasured disturbance — N_w
- Measured outputs — H_2 , N_2 , and H_2

Define MPC Structure By Importing

MPC Structure

```
graph LR; Setpoints[Setpoints (reference)] --> MPC[MPC]; Disturbances1[Measured Disturbances] --> MPC; Disturbances2[Unmeasured Disturbances] --> MPC; MPC --> ManipulatedVariables[Manipulated Variables]; MPC --> Disturbances3[Unmeasured Disturbances]; ManipulatedVariables --> Inputs[Inputs]; Disturbances3 --> Inputs; Inputs --> Plant[Plant]; Plant --> Unmeasured[Unmeasured]; Plant --> Measured[Measured]; Unmeasured --> Outputs[Outputs]; Measured --> Outputs;
```

Select a plant model or an MPC controller from MATLAB Workspace:

Select	Name	Type	Order	Inputs	Outputs
<input checked="" type="radio"/>	PaperMach	ss	4	4	3

Controller Sample Time
Specify MPC controller sample time:

Assign plant i/o channels to desired signal types:

Manipulated variable (MV) channel indices:	<input type="text" value="1;2"/>
Measured disturbance (MD) channel indices:	<input type="text" value="3"/>
Unmeasured disturbance (UD) channel indices:	<input type="text" value="4"/>
Measured output (MO) channel indices:	<input type="text" value="1;2;3"/>
Unmeasured output (UO) channel indices:	<input type="text"/>

Refresh workspace Define and Import Cancel Help

Tip To find the correct channel indices, click the **PaperMachine** model **Name** to view additional model details.

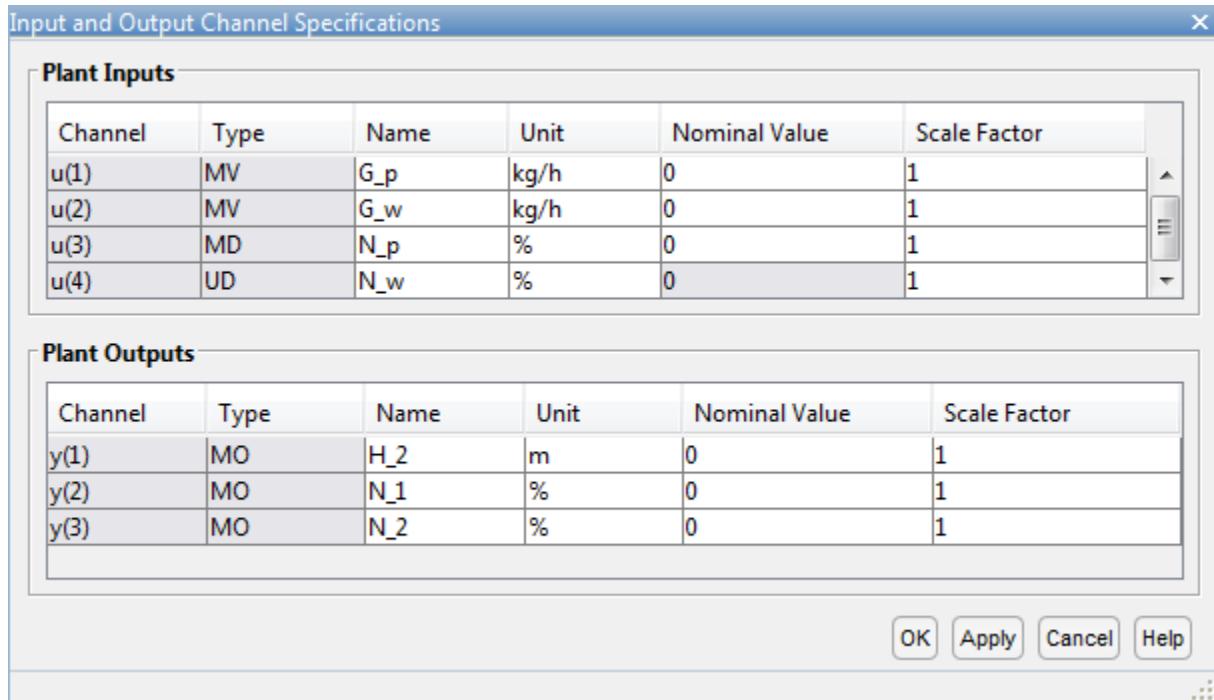
Click **Define and Import**.

The app imports the plant to the **Data Browser** and creates a default MPC controller using the imported plant.

Define Input and Output Channel Attributes

In the **Structure** section, select **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, in the **Unit** column, define the units for each channel. Since all the signals are normalized with zero nominal steady-state values, keep the **Nominal Value** and **Scale Factor** for each channel at their default values.



Click **OK** to update the channel attributes and close the dialog box.

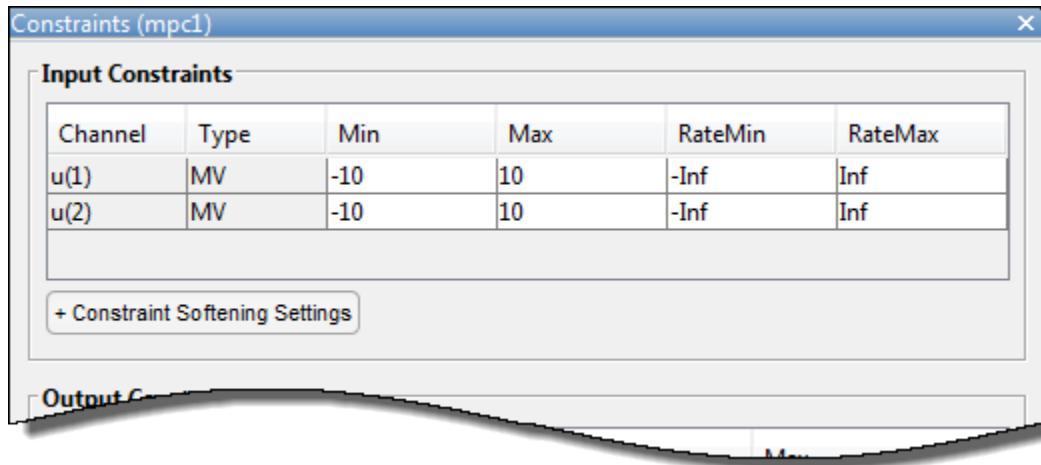
Specify Controller Sample Time and Horizons

On the **Tuning** tab, in the **Horizon** section, keep the **Sample time**, **Prediction Horizon**, and **Control Horizon** at their default values.

Specify Manipulated Variable Constraints

In the **Design** section, click **Constraints**.

In the Constraints dialog box, in the **Input Constraints** section, specify value constraints, **Min** and **Max**, for both manipulated variables.



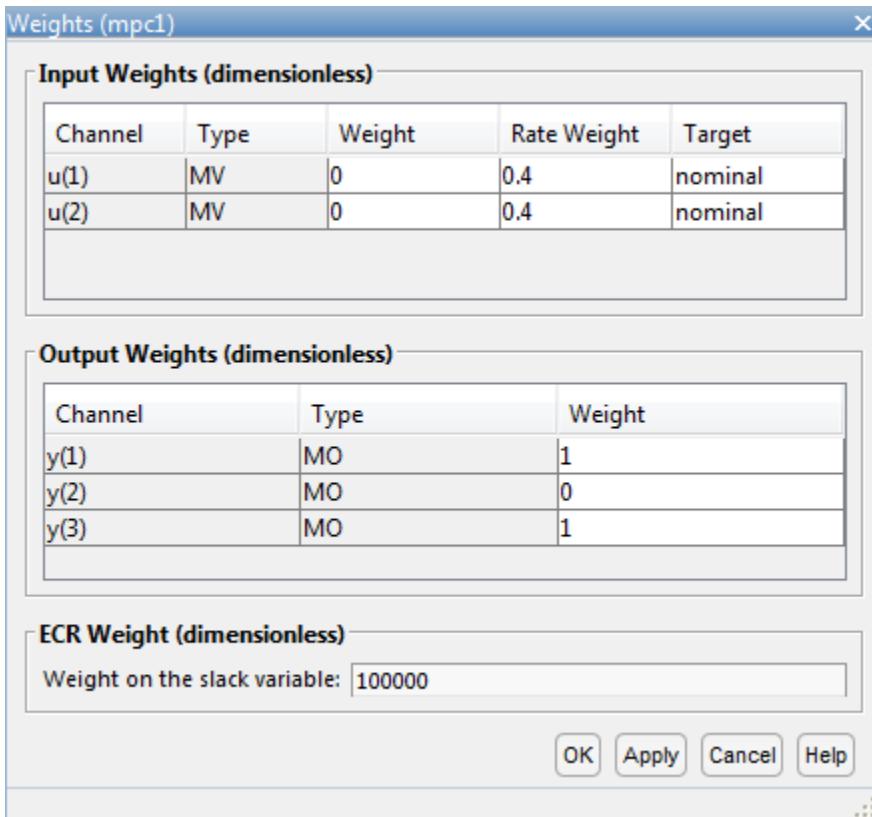
Click **OK**.

Specify Initial Tuning Weights

In the **Design** section, click **Weights**.

In the Weights dialog box, in the **Input Weights** section, increase the **Rate Weight** to **0.4** for both manipulated variables.

In the **Output Weights** section, specify a **Weight** of **0** for the second output, N_1 , and a **Weight** of **1** for the other outputs.



Increasing the rate weight for manipulated variables prevents overly-aggressive control actions resulting in a more conservative controller response.

Since there are two manipulated variables, the controller cannot control all three outputs completely. A weight of zero indicates that there is no setpoint for N_1 . As a result, the controller can hold H_2 and N_2 at their respective setpoints.

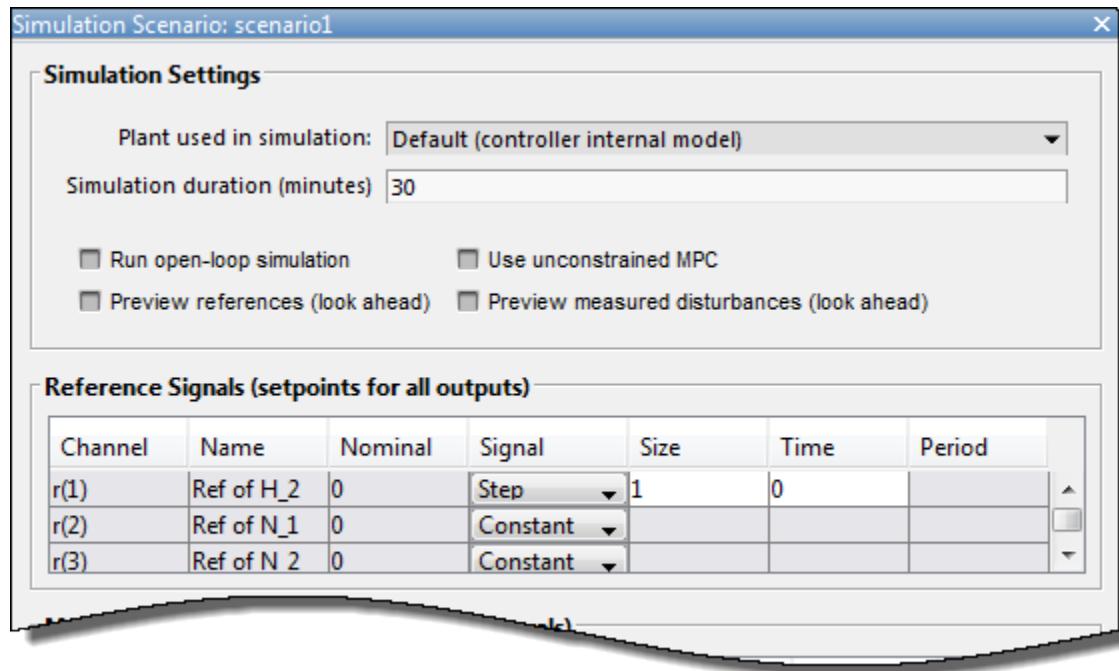
Simulate H_2 Setpoint Step Response

On the **MPC Designer** tab, in the **Scenario** section, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, specify a **Simulation duration** of 30 minutes.

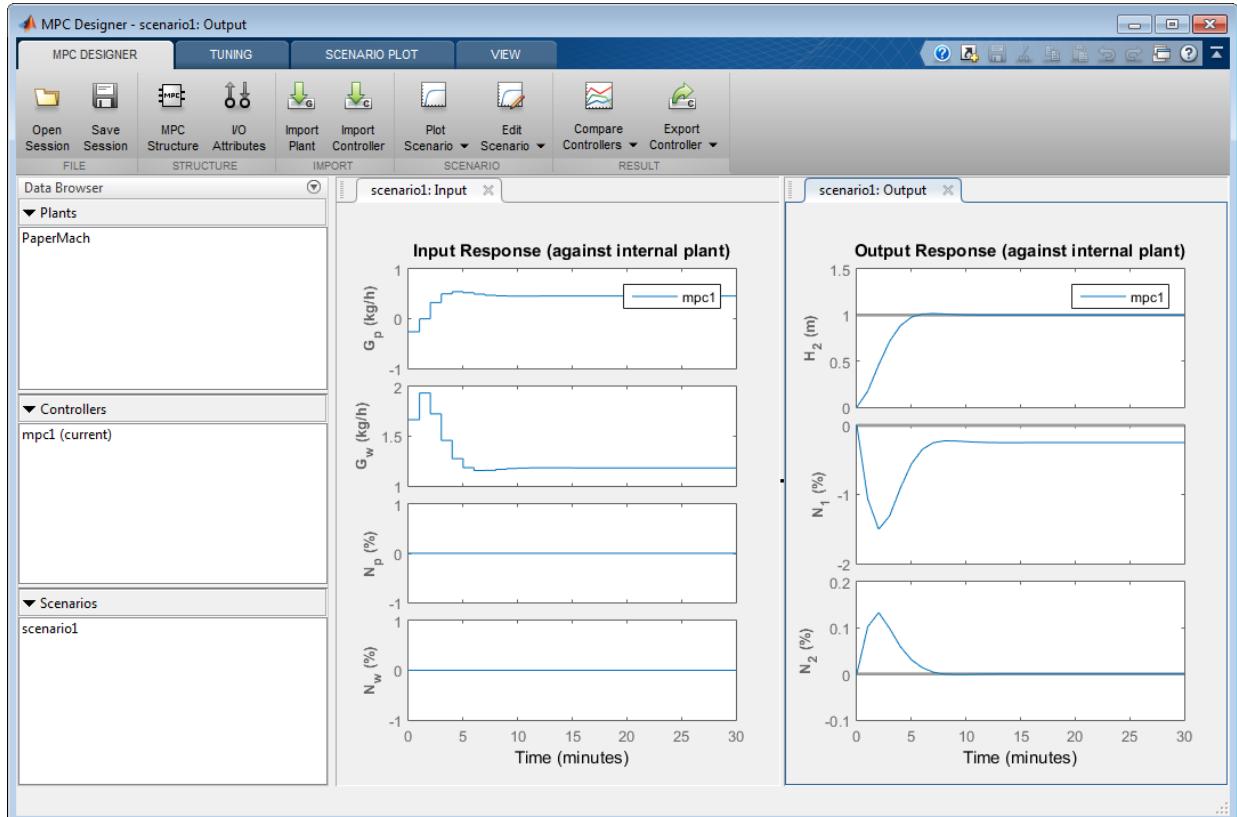
In the **Reference Signals** table, in the **Signal** drop-down list, select **Step** for the first output. Keep the step **Size** at 1 and specify a step **Time** of 0.

In the **Signal** drop-down lists for the other output reference signals, select **Constant** to hold the values at their respective nominal values. The controller ignores the setpoint for the second output since the corresponding tuning weight is zero.



Click **OK**.

The app runs the simulation with the new scenario settings and updates the **Input Response** and **Output Response** plots.



The initial design uses a conservative control effort to produce a robust controller. The response time for output H_2 is about 7 minutes. To reduce this response time, you can decrease the sample time, reduce the manipulated variable rate weights, or reduce the manipulated variable rate constraints.

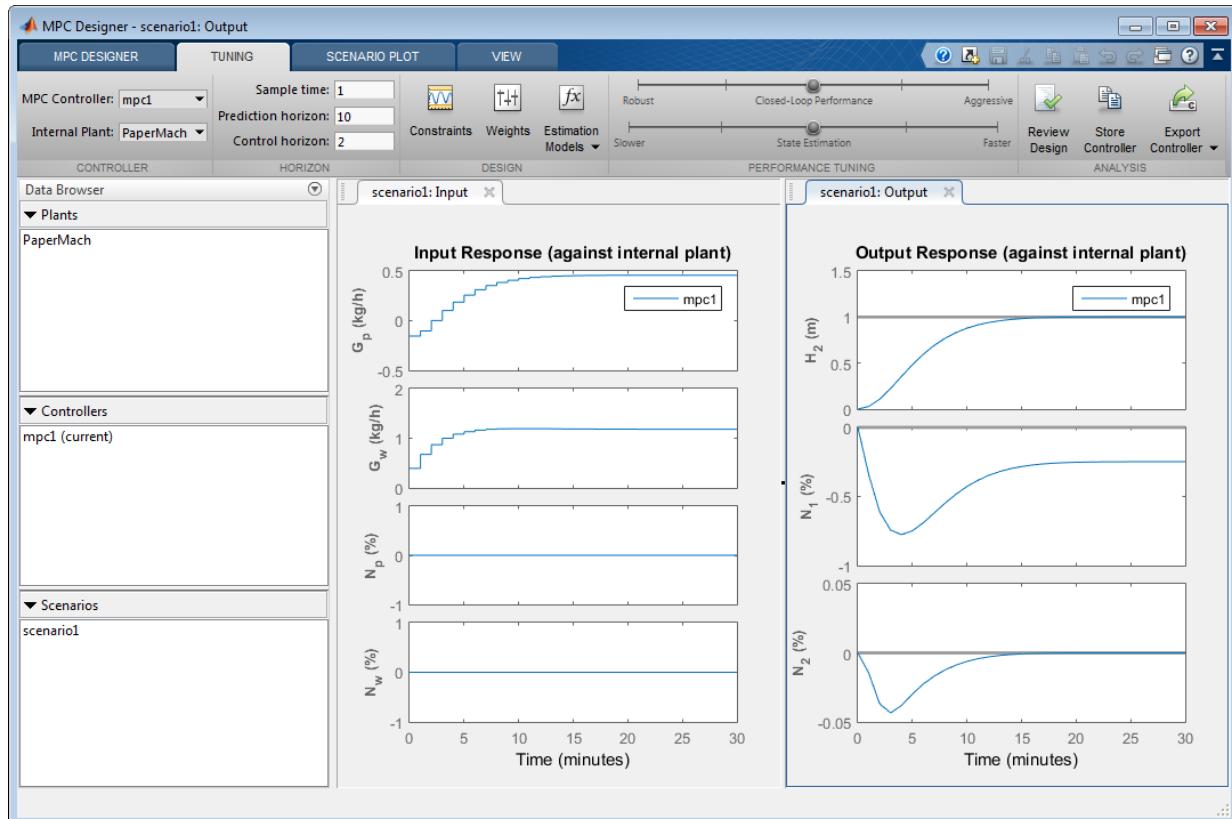
Since the tuning weight for output N_1 is zero, its output response shows a steady-state error of about -0.25 .

Adjust Weights to Emphasize Feed Tank Consistency Control

On the **Tuning** tab, in the **Design** section, select **Weights**.

In the **Weights** dialog box, in the **Output Weights** section, specify a **Weight** of **0.2** for the first output, H_2 .

4 Case-Study Examples



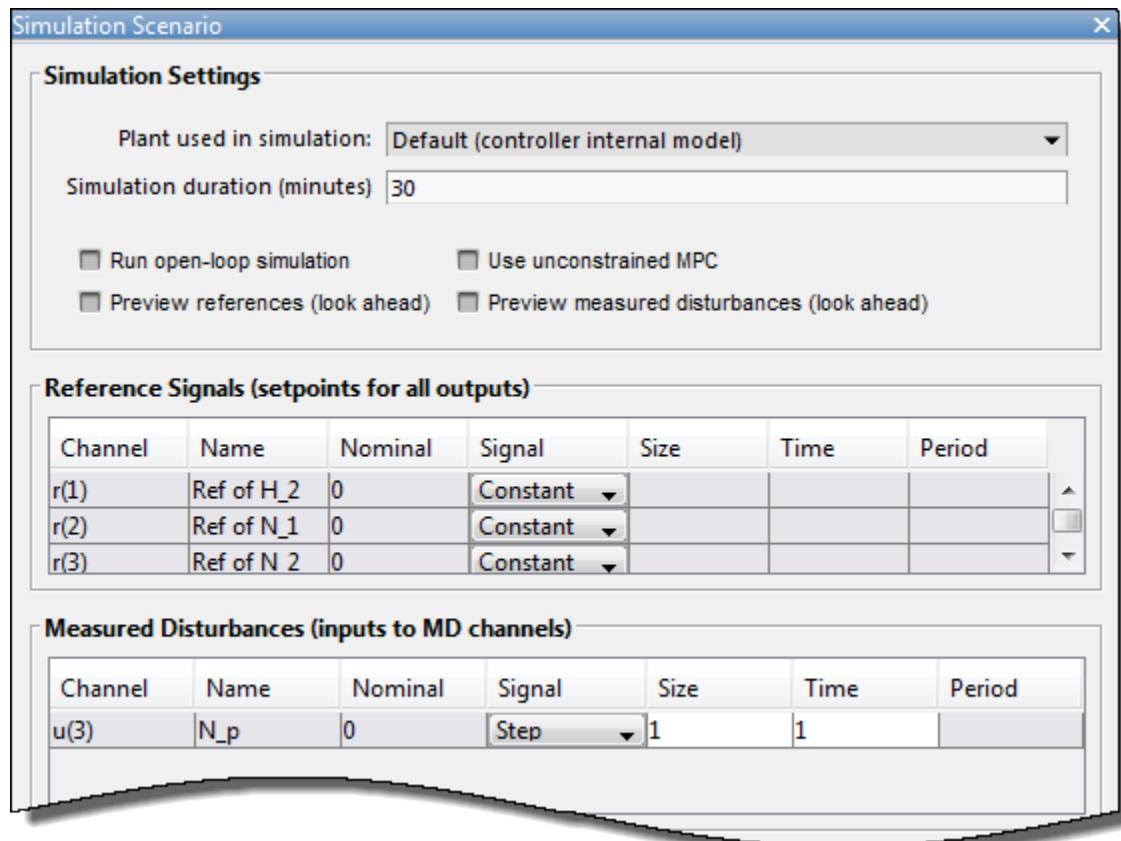
The controller places more emphasis on eliminating errors in feed tank consistency, N_2 , which significantly decreases the peak absolute error. The trade-off is a longer response time of about 17 minutes for the feed tank level, H_2 .

Test Controller Feedforward Response to Measured Disturbances

On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > New Scenario**.

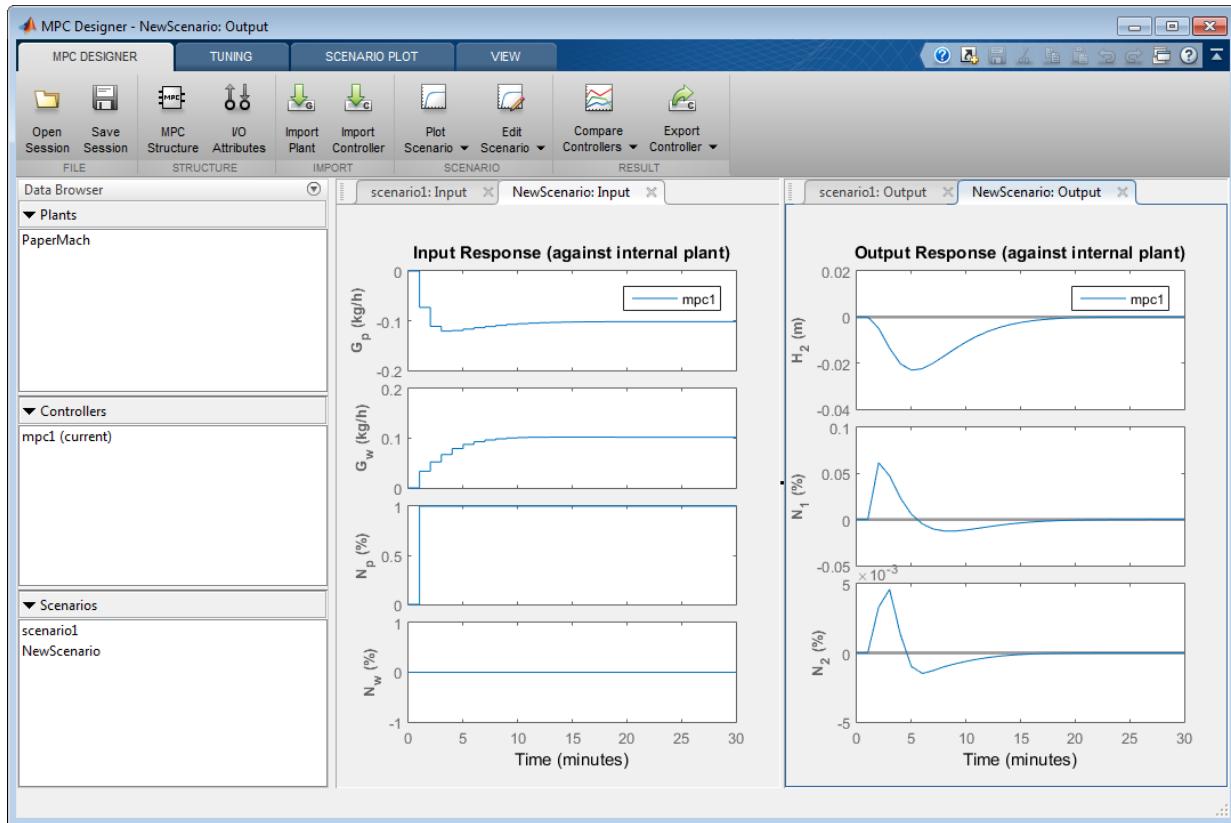
In the Simulation Scenario dialog box, specify a **Simulation duration** of 30 minutes.

In the **Measured Disturbances** table, specify a step change in measured disturbance, N_p , with a **Size** of 1 and a step **Time** of 1. Keep all output setpoints constant at their nominal values.



Click **OK** to run the simulation and display the input and output response plots.

4 Case-Study Examples



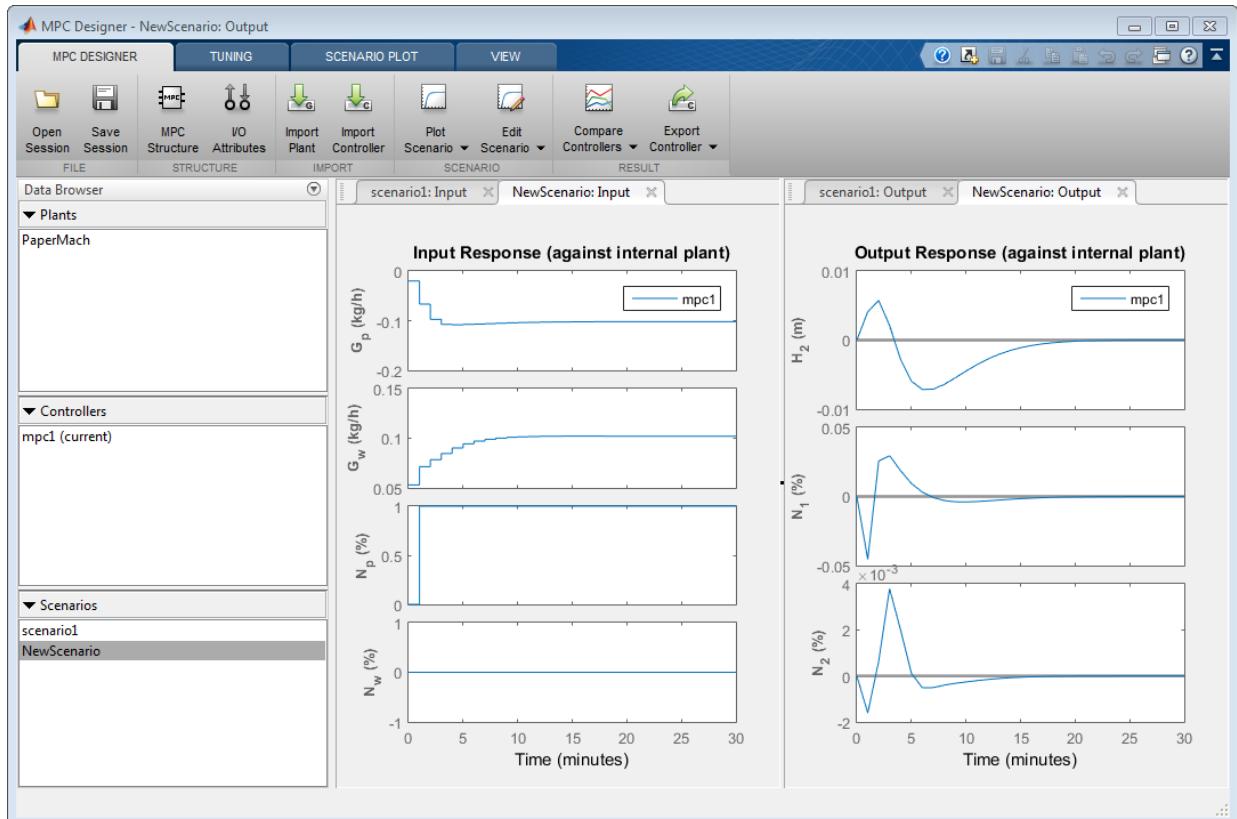
As shown in the **NewScenario: Output** plot, both H_2 and N_2 deviate little from their setpoints.

Experiment with Signal Previewing

In the **Data Browser**, in the **Scenarios** section, right-click **NewScenario**, and select **Edit**.

In the Simulation Scenario dialog box, in the **Simulation Settings** section, check the **Preview measured disturbances** option.

Click **Apply**.



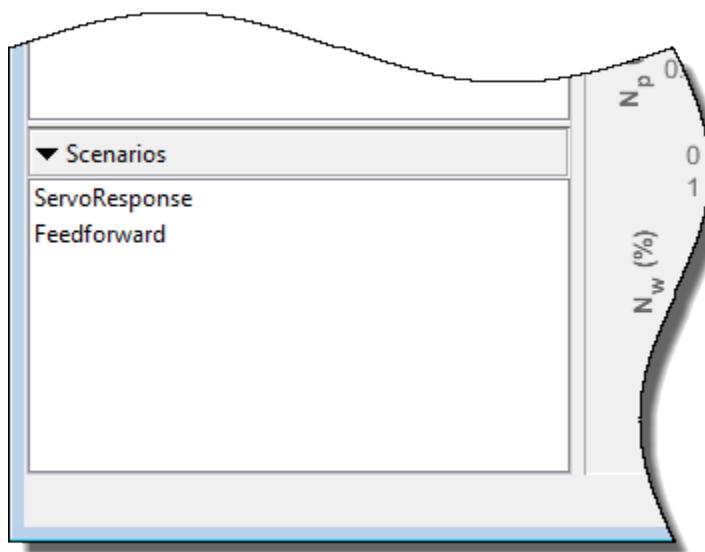
The manipulated variables begin changing before the measured disturbance occurs because the controller uses the known future disturbance value when computing its control action. The output disturbance values also begin changing before the disturbance occurs, which reduces the magnitude of the output errors. However, there is no significant improvement over the previous simulation result.

In the Simulation Scenario dialog box, clear the **Preview measured disturbances** option.

Click **OK**.

Rename Scenarios

With multiple scenarios, it is helpful to provide them with meaningful names. In the **Data Browser**, in the **Scenarios** section, double-click each scenario to rename them as shown:



Test Controller Feedback Response to Unmeasured Disturbances

In the **Data Browser**, in the **Scenarios** section, right-click **Feedforward**, and select **Copy**.

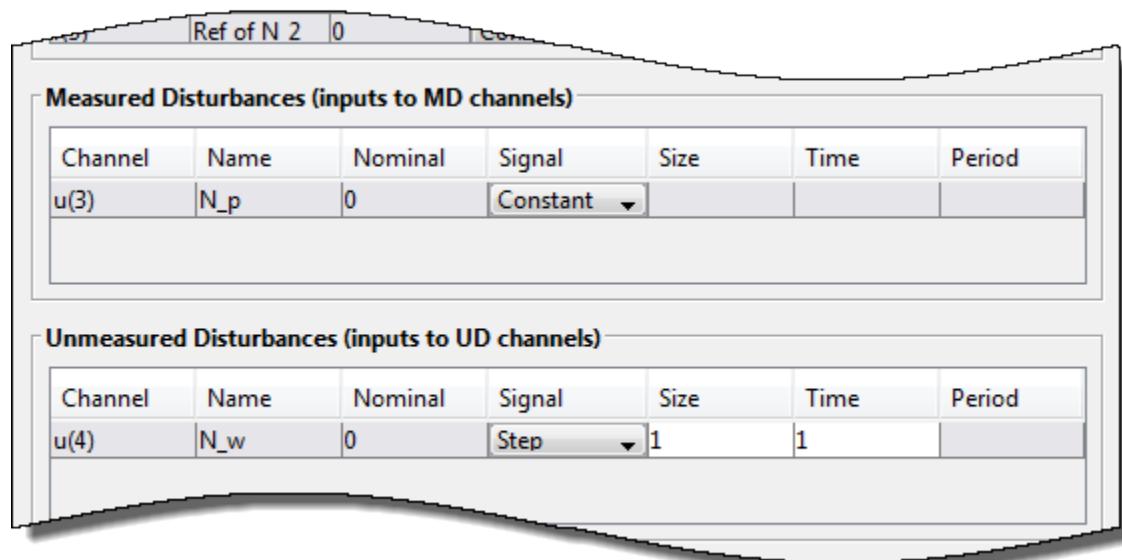
Double-click the new scenario, and rename it **Feedback**.

Right-click the **Feedback** scenario, and select **Edit**.

In the Simulation Scenario dialog box, in the **Measured Disturbances** table, in the **Signal** drop-down list, select **Constant** to remove the measured disturbance.

In the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step** to simulate a sudden, sustained unmeasured input disturbance.

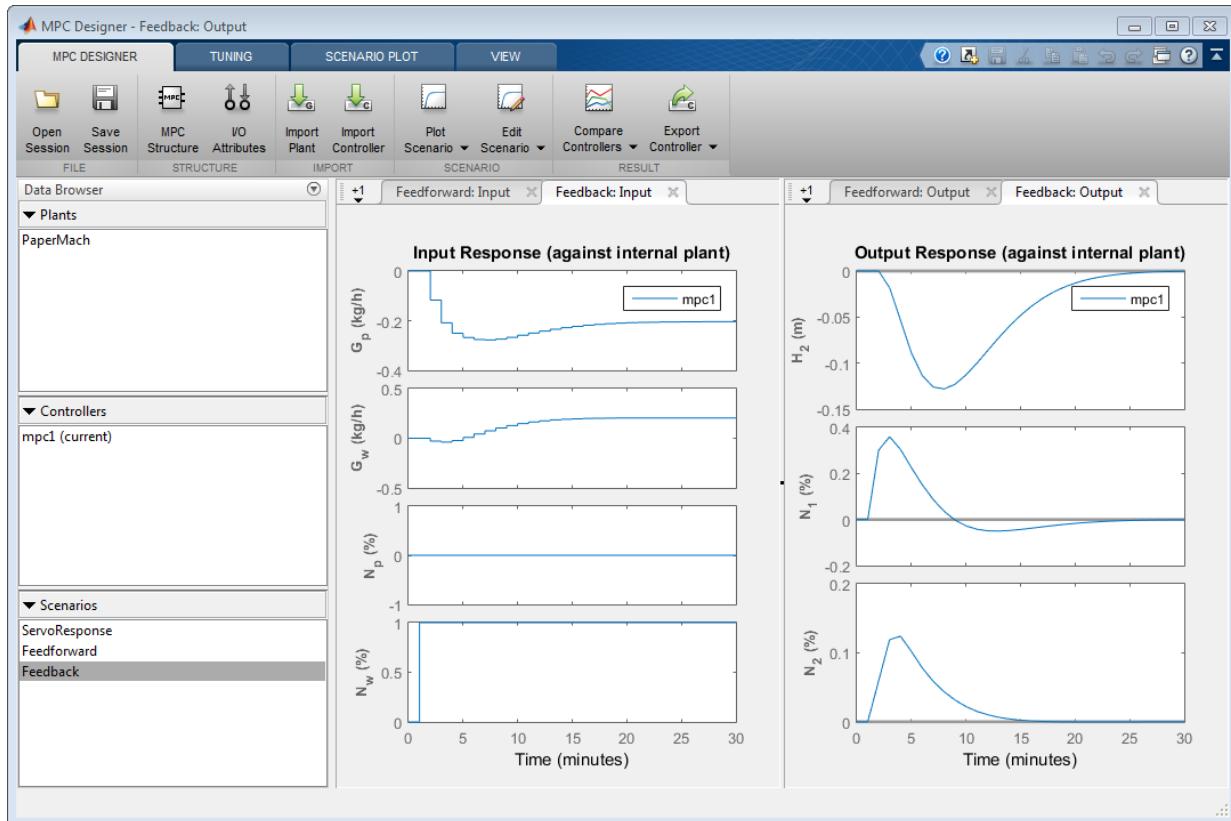
Set the step **Size** to 1 and the step **Time** to 1.



Click **OK** to update the scenario settings, and run the simulation.

In the **Data Browser**, in the **Scenarios** section, right-click **Feedback**, and select **Plot**.

4 Case-Study Examples



The controlled outputs, H_2 and N_2 , both exhibit relatively small deviations from their setpoints. The settling time is longer than for the original servo response, which is typical.

On the **Tuning** tab, in the **Analysis** section, click **Review Design** to check the controller for potential run-time stability or numerical problems.

The review report opens in a new window.

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Warning
Hard MV Constraints	Pass
Other Hard Constraints	Pass
Soft Constraints	Pass
Memory Size for MPC Data	Pass

The review flags two warnings about the controller design. Click the warning names to determine whether they indicate problems with the controller design.

The **Closed-Loop Steady-State Gains** warning indicates that the plant has more controlled outputs than manipulated variables. This input/output imbalance means that the controller cannot eliminate steady-state error for all of the outputs simultaneously. To meet the control objective of tracking the setpoints of H_2 and N_2 , you previously set the output weight for N_1 to zero. This setting causes the **QP Hessian Matrix Validity** warning, which indicates that one of the output weights is zero.

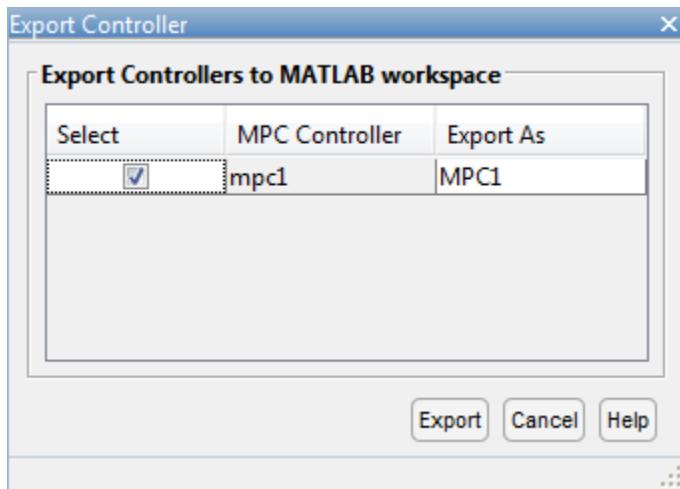
Since the input/output imbalance is a known feature of the paper machine plant model, and you intentionally set one of the output weights to zero to correct for the imbalance, neither warning indicates an issue with the controller design.

Export Controller to MATLAB Workspace

On the **MPC Designer** tab, in the **Result** section, click **Export Controller** .

In the Export Controller dialog box, check the box in the **Select** column.

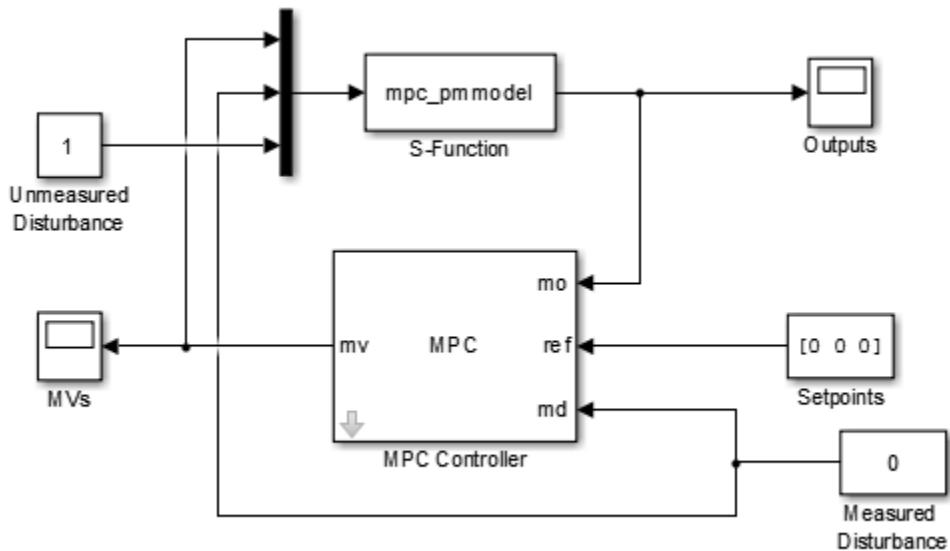
In the **Export As** column, specify MPC1 as the controller name.



Click **Export** to save a copy of the controller to the MATLAB workspace.

Open Simulink Model

```
open_system( mpc_papermachine )
```

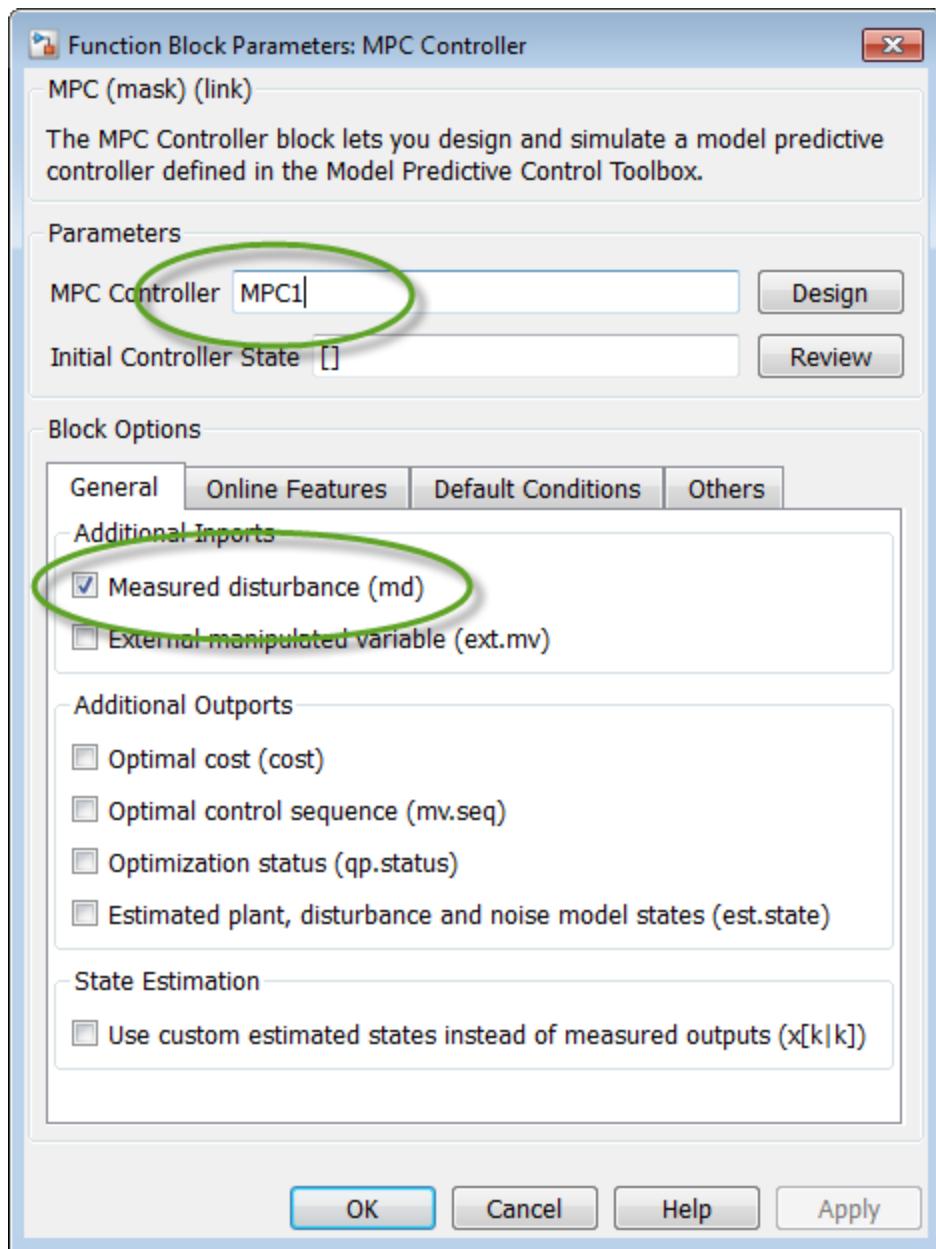


The **MPC Controller** block controls the nonlinear paper machine plant model, which is defined using the S-Function `mpc_pmmmodel`.

The model is configured to simulate a sustained unmeasured disturbance of size 1.

Configure MPC Controller Block

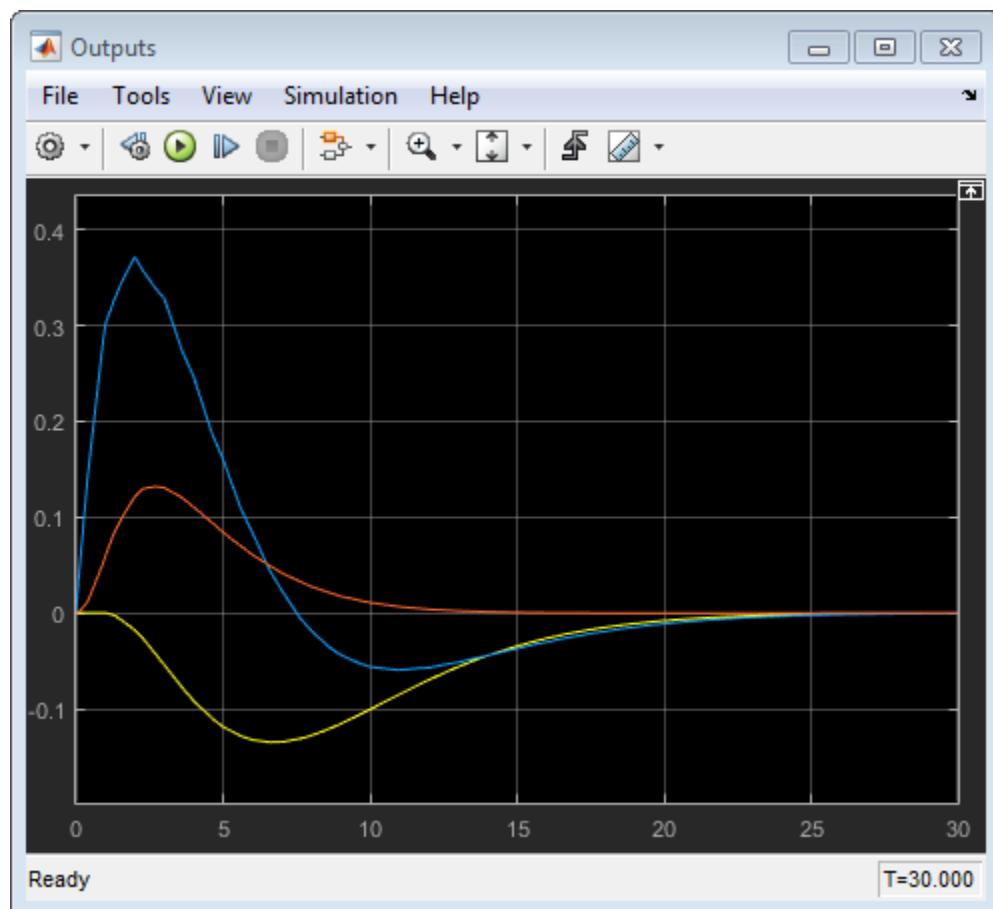
Double-click the **MPC Controller** block.



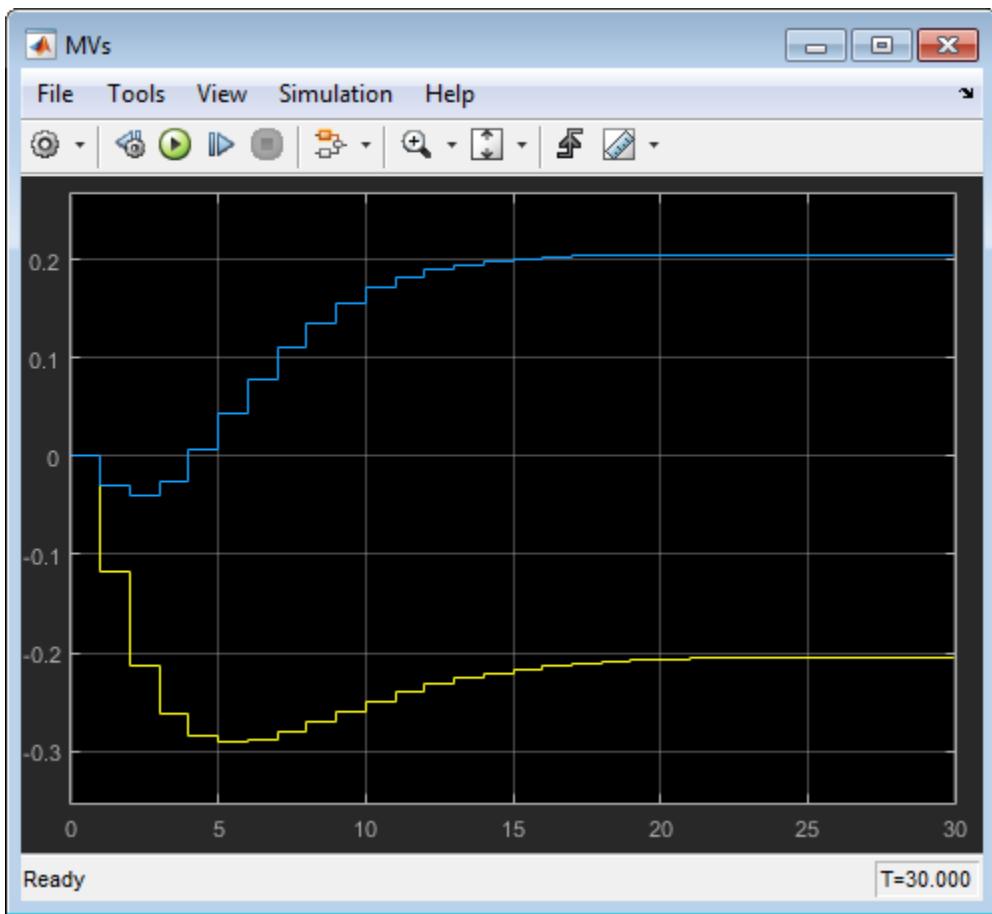
The **MPC Controller** block is already configured to use the **MPC1** controller that was previously exported to the MATLAB workspace.

Also, the **Measured disturbance** option is selected to add the **md** import to the controller block.

Simulate the model



In the **Outputs** plot, the responses are almost identical to the responses from the corresponding simulation in MPC Designer. The yellow curve is H_2 , the blue is N_1 , and the red is N_2 .



Similarly, in the **MVs** scope, the manipulated variable moves are almost identical to the moves from corresponding simulation in MPC Designer. The yellow curve is G_p and the blue is G_w .

These results show that there are no significant prediction errors due to the mismatch between the linear prediction model of the controller and the nonlinear plant. Even increasing the unmeasured disturbance magnitude by a factor of four produces similarly shaped response curves. However, as the disturbance size increases further, the effects of nonlinearities become more pronounced.

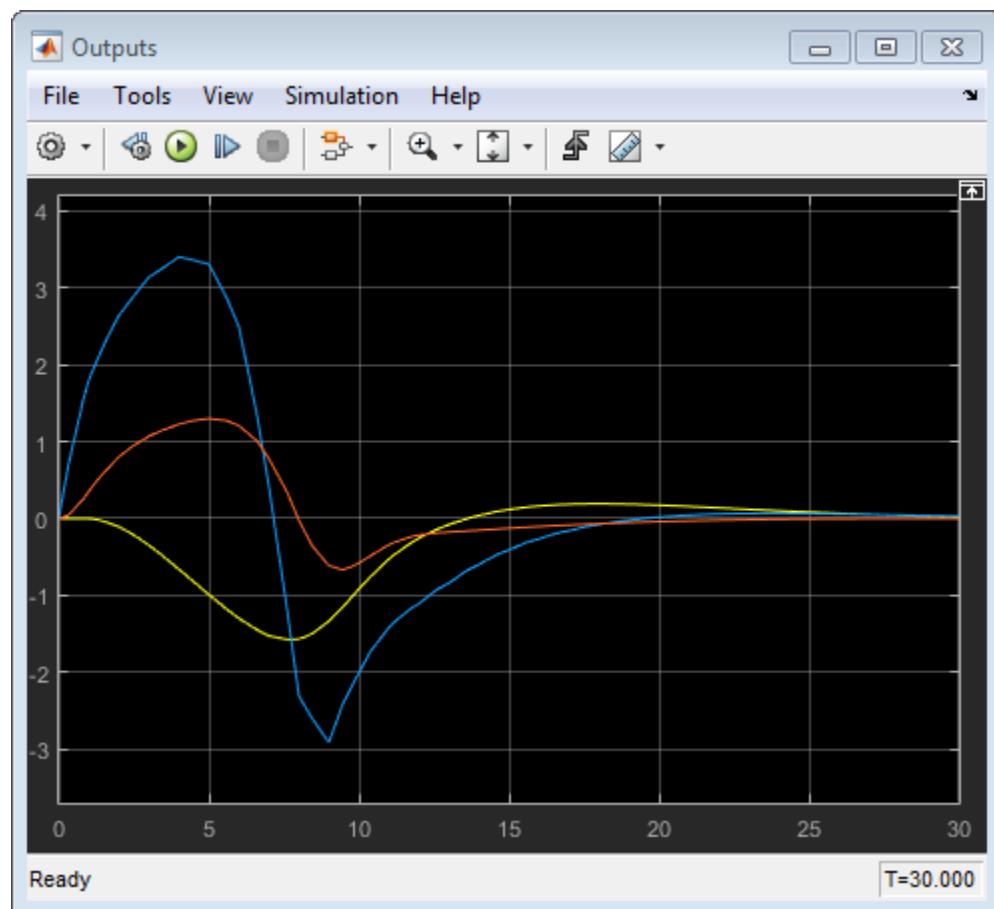
Increase Unmeasured Disturbance Magnitude

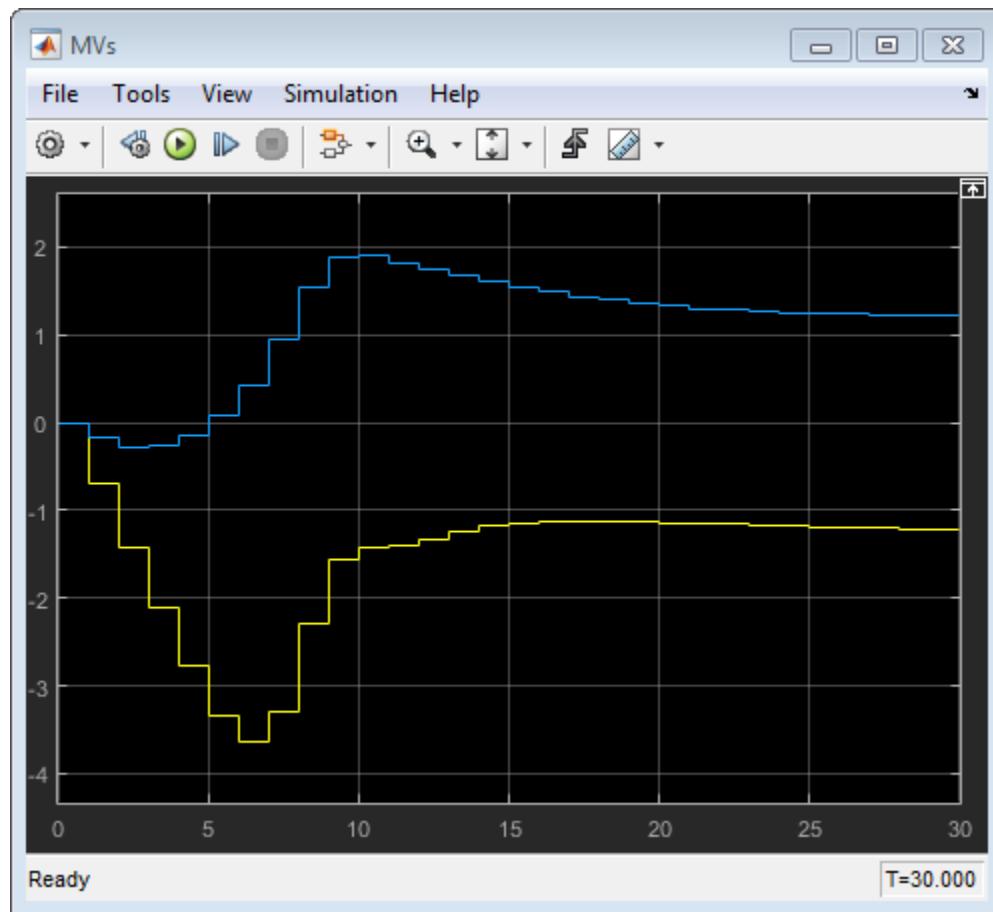
In the Simulink model window, double-click the **Unmeasured Disturbance** block.

In the Unmeasured Disturbance properties dialog box, specify a **Constant value** of **6.5**.

Click **OK**.

Simulate the model.





The mismatch between the prediction model and the plant now produces output responses with significant differences. Increasing the disturbance magnitude further results in large setpoint deviations and saturated manipulated variables.

References

- [1] Ying, Y., M. Rao, and Y. Sun "Bilinear control strategy for paper making process," *Chemical Engineering Communications* (1992), Vol. 111, pp. 13–28.

See Also

[MPC Controller](#) | [MPC Designer](#)

Related Examples

- “[Design Controller Using MPC Designer](#)”

Bumpless Transfer Between Manual and Automatic Operations

In this section...

- “Open Simulink Model” on page 4-50
- “Define Plant and MPC Controller” on page 4-51
- “Configure MPC Block Settings” on page 4-52
- “Examine Switching Between Manual and Automatic Operation” on page 4-53
- “Turn off Manipulated Variable Feedback” on page 4-55

This example shows how to bumplessly transfer between manual and automatic operations of a plant.

During startup of a manufacturing process, operators adjust key actuators manually until the plant is near the desired operating point before switching to automatic control. If not done correctly, the transfer can cause a *bump*, that is, large actuator movement.

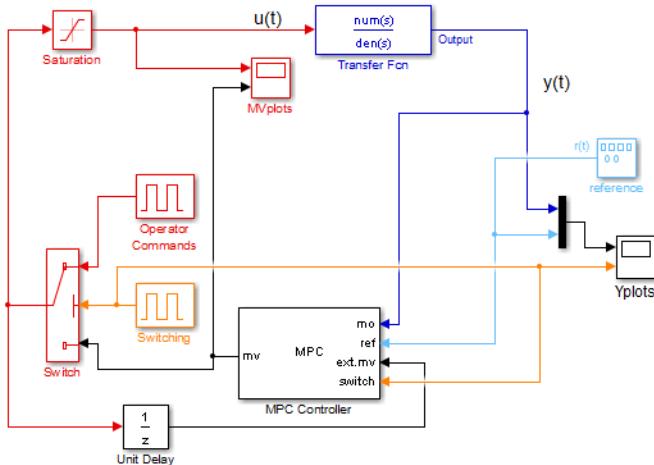
In this example, you simulate a Simulink model that contains a single-input single-output LTI plant and an **MPC Controller** block.

A model predictive controller monitors all known plant signals, even when it is not in control of the actuators. This monitoring improves its state estimates and allows a bumpless transfer to automatic operation.

Open Simulink Model

Open the Simulink model.

```
open_system( 'mpc_bumpless' )
```



To simulate switching between manual and automatic operation, the **Switching** block sends either 1 or 0 to control a switch. When it sends 0, the system is in automatic mode, and the output from the **MPC Controller** block goes to the plant. Otherwise, the system is in manual mode, and the signal from the **Operator Commands** block goes to the plant.

In both cases, the actual plant input feeds back to the controller **ext.mv** import, unless the plant input saturates at -1 or 1. The controller constantly monitors the plant output and updates its estimate of the plant state, even when in manual operation.

This model also shows the optimization switching option. When the system switches to manual operation, a nonzero signal enters the **switch** import of the controller block. The signal turns off the optimization calculations, which reduces computational effort.

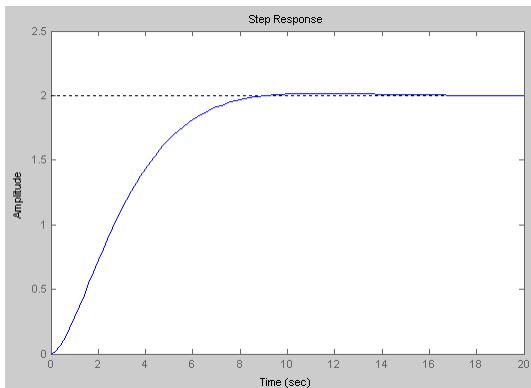
Define Plant and MPC Controller

Create the plant model.

```
num = [1 1];
den = [1 3 2 0.5];
sys = tf(num,den);
```

The plant is a stable single-input single-output system as seen in its step response.

```
step(sys)
```



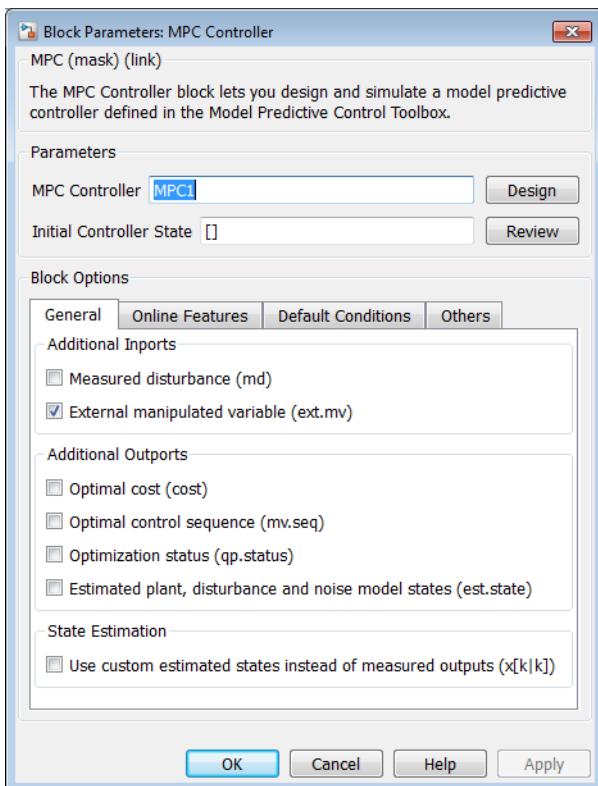
Create an MPC controller.

```
Ts = 0.5; % sampling time (seconds)
p = 15; % prediction horizon
m = 2; % control horizon
MPC1 = mpc(sys,Ts,p,m);
MPC1.Weights.Output = 0.01;
MPC1.MV = struct( Min ,-1, Max ,1);
Tstop = 250;
```

Configure MPC Block Settings

Open the Function Block Parameters: MPC Controller dialog box.

- Specify MPC1 in the **MPC Controller** box.
- Verify that the **External Manipulated Variable (ext.mv)** option in the **General** tab is selected. This option adds the `ext.mv` import to the block to enable the use of external manipulated variables.
- Verify that the **Use external signal to enable or disable optimization (switch)** option in the **Others** tab is selected. This option adds the `switch` import to the controller block to enable switching off the optimization calculations.

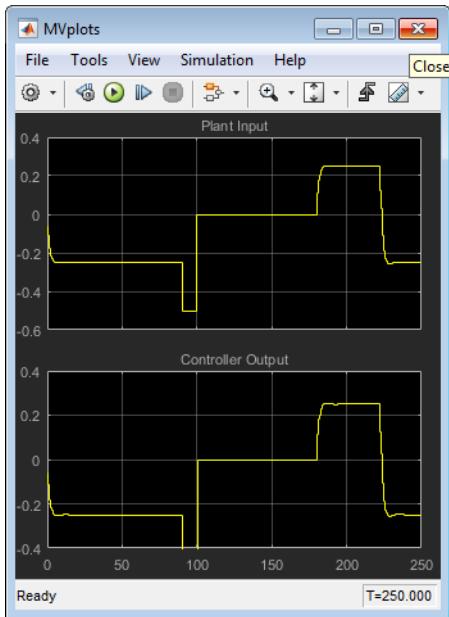
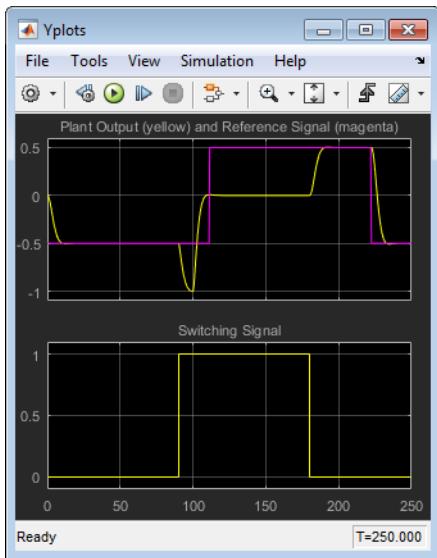


Click **OK**.

Examine Switching Between Manual and Automatic Operation

Click **Run** in the Simulink model window to simulate the model.

4 Case-Study Examples



For the first 90 time units, the **Switching Signal** is 0, which makes the system operate in automatic mode. During this time, the controller smoothly drives the controlled plant output from its initial value, 0, to the desired reference value, -0.5.

The controller state estimator has zero initial conditions as a default, which is appropriate when this simulation begins. Thus, there is no bump at startup. In general, start the system running in manual mode long enough for the controller to acquire an accurate state estimate before switching to automatic mode.

At time 90, the **Switching Signal** changes to 1. This change switches the system to manual operation and sends the operator commands to the plant. Simultaneously, the nonzero signal entering the **switch** import of the controller turns off the optimization calculations. While the optimization is turned off, the **MPC Controller** block passes the current **ext.mv** signal to the **Controller Output**.

Once in manual mode, the operator commands set the manipulated variable to -0.5 for 10 time units, and then to 0. The **Plant Output** plot shows the open-loop response between times 90 and 180 when the controller is deactivated.

At time 180, the system switches back to automatic mode. As a result, the plant output returns to the reference value smoothly, and a similar smooth adjustment occurs in the controller output.

Turn off Manipulated Variable Feedback

Delete the signals entering the **ext.mv** and **switch** imports of the controller block.

Delete the **Unit Delay** block and the signal line entering its import.

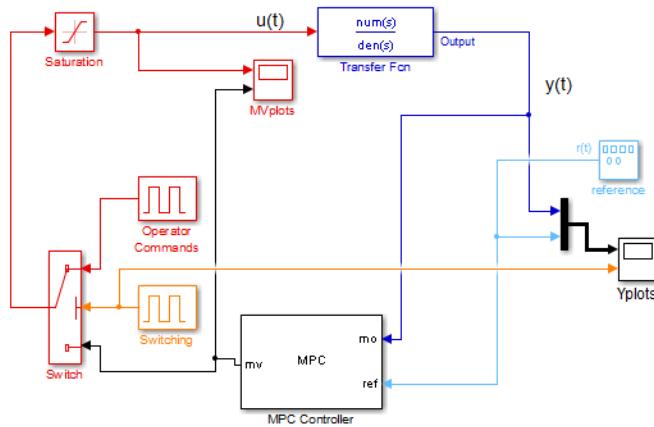
Open the Function Block Parameters: MPC Controller dialog box.

Deselect the **External Manipulated Variable (ext.mv)** option in the **General** tab to remove the **ext.mv** import from the controller block.

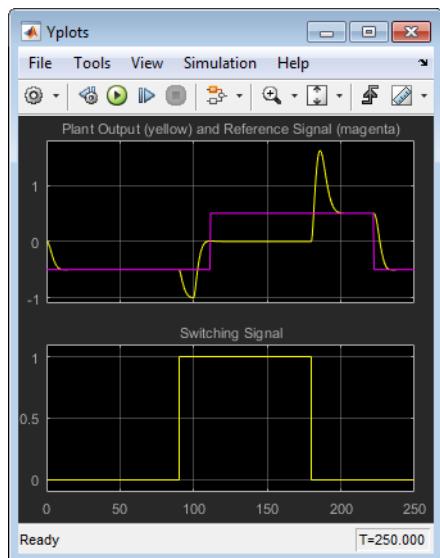
Deselect the **Use external signal to enable or disable optimization (switch)** option in the **Others** tab to remove the **switch** import from the controller block.

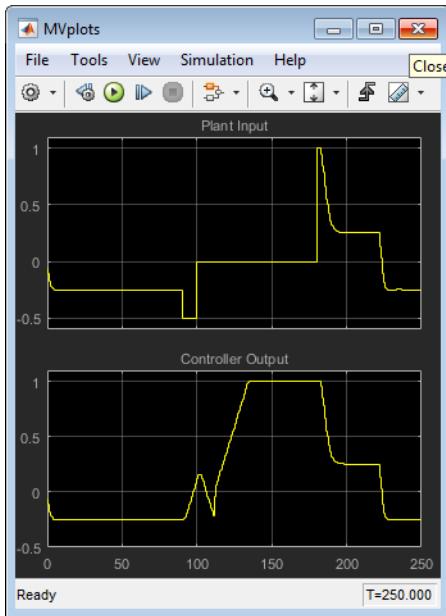
Click **OK**. The Simulink model now resembles the following figure.

4 Case-Study Examples



Click **Run** to simulate the model.





The behavior is identical to the original case for the first 90 time units.

When the system switches to manual mode at time 90, the plant behavior is the same as before. However, the controller tries to hold the plant at the setpoint. So, its output increases and eventually saturates, as seen in **Controller Output**. Since the controller assumes that this output is going to the plant, its state estimates become inaccurate. Therefore, when the system switches back to automatic mode at time 180, there is a large bump in the **Plant Output**.

Such a bump creates large actuator movements within the plant. By smoothly transferring from manual to automatic operation, a model predictive controller eliminates such undesired movements.

Related Examples

- “Switching Controller Online and Offline with Bumpless Transfer” on page 4-58

Switching Controller Online and Offline with Bumpless Transfer

This example shows how to obtain bumpless transfer when switching model predictive controller from manual to automatic operation or vice versa.

In particular, it shows how the EXT.MV input signal to the MPC block can be used to keep the internal MPC state up to date when the operator or another controller is in control.

Define Plant Model

The linear open-loop dynamic plant model is as follows:

```
num = [1 1];
den = [1 3 2 0.5];
sys = tf(num,den);
```

Design MPC Controller

Construct MPC controller

Create an MPC controller with plant model, sample time and horizons.

```
Ts = 0.5; % Sampling time
p = 15; % Prediction horizon
m = 2; % Control horizon
mpcobj = mpc(sys,Ts,p,m);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

MV Constraints

Define constraints on the manipulated variable.

```
mpcobj.MV=struct( Min ,-1, Max ,1);
```

Weights

Change the output weight.

```
mpcobj.Weights.Output=0.01;
```

Simulate Using Simulink

To run this example, Simulink® is required.

```

if ~mpcchecktoolboxinstalled( simulink )
    disp( 'Simulink(R) is required to run this example. ' )
    return
end

```

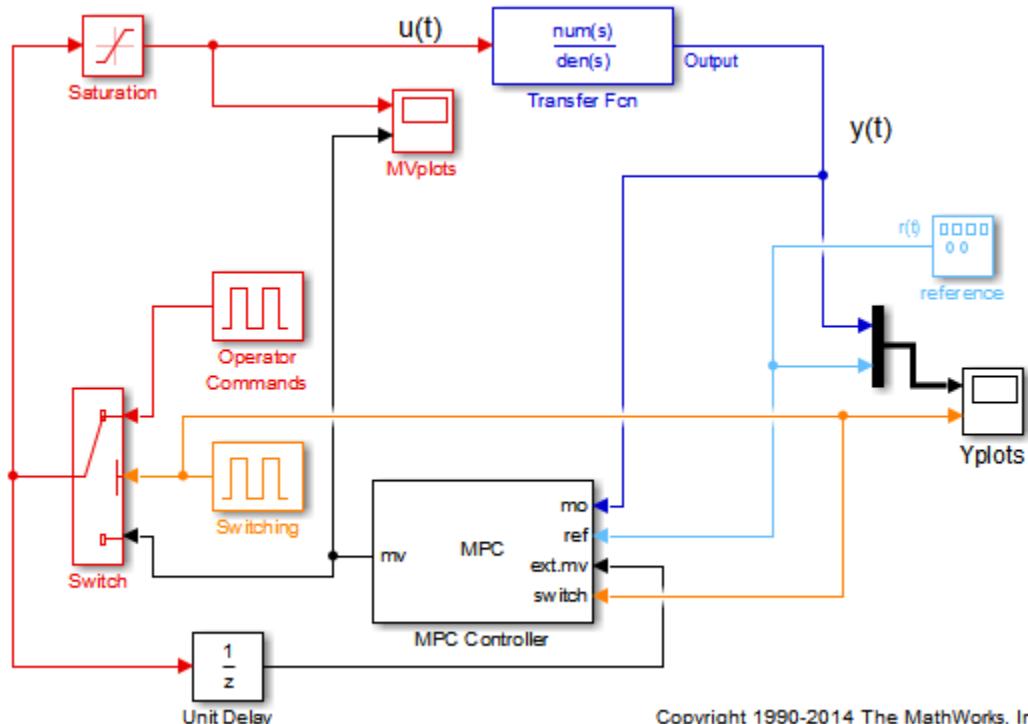
Simulate closed-loop control of the linear plant model in Simulink. Controller "mpcobj" is specified in the block dialog.

```

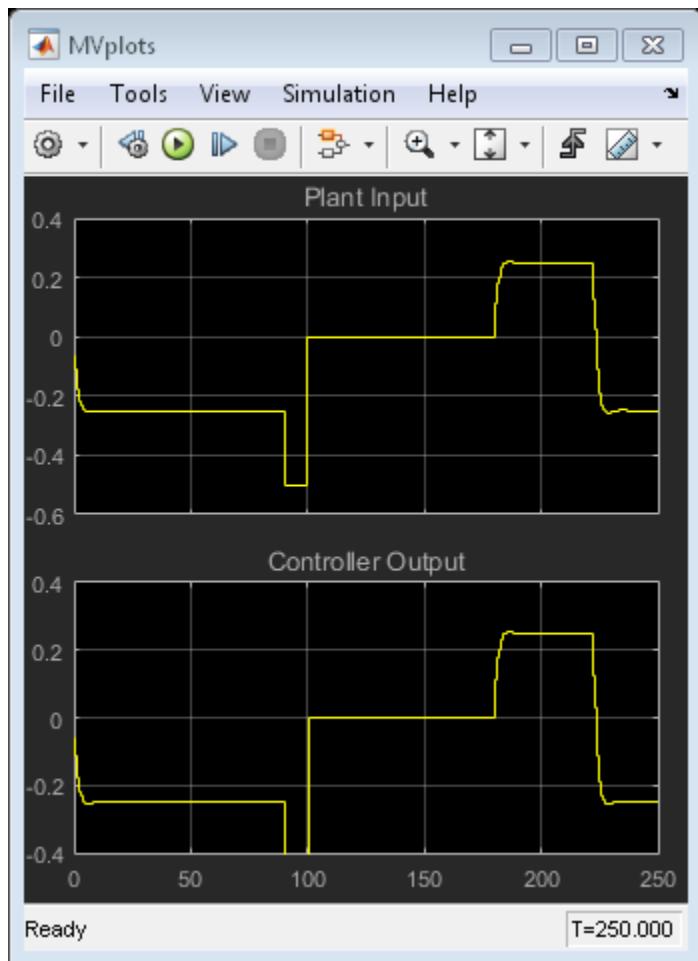
mdl = mpc_bumpless ;
open_system(mdl)
sim(mdl)

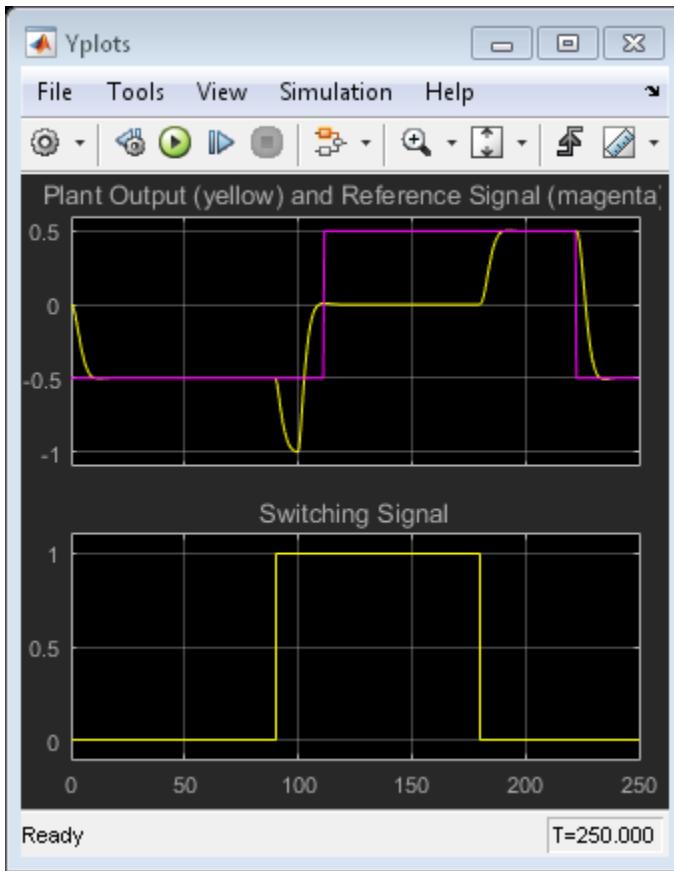
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each

```



Copyright 1990-2014 The MathWorks, Inc.





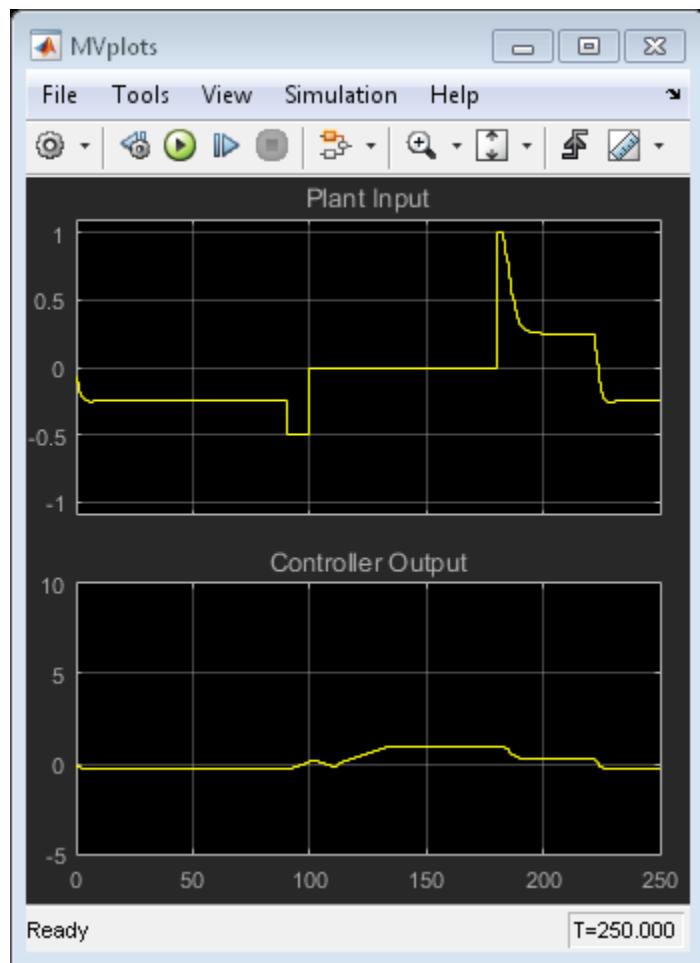
Simulate without Using External MV Signal

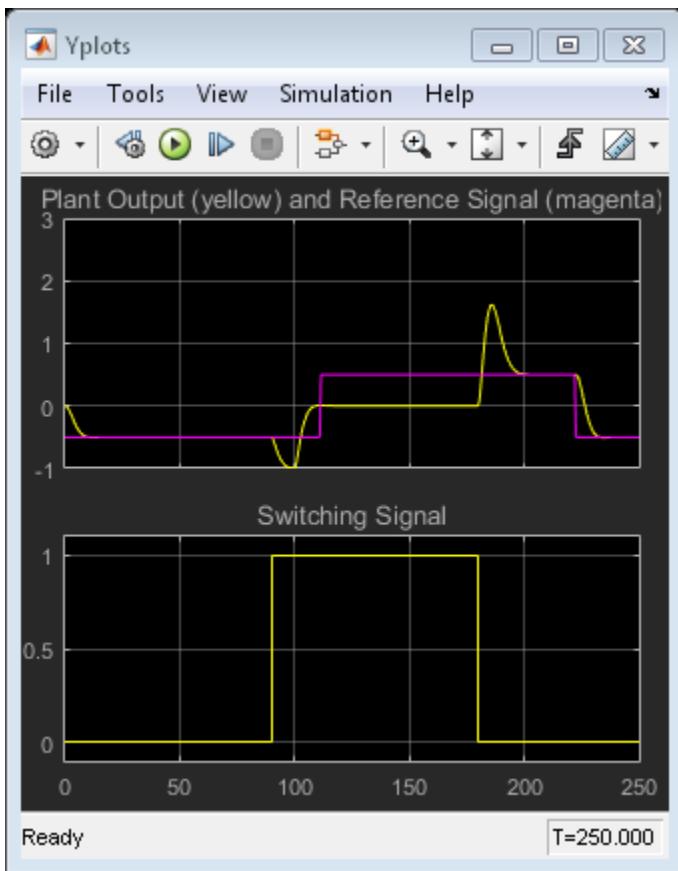
Without using the external MV signal, MPC controller is no longer able to provide bumpless transfer because the internal controller states are not estimated correctly.

```
delete_line(mdl, Switch/1 , Unit Delay/1 );
delete_line(mdl, Unit Delay/1 , MPC Controller/3 );
delete_block([mdl /Unit Delay ]);
delete_line(mdl, Switching/1 , MPC Controller/4 );
set_param([mdl /MPC Controller ], mv_inport , off );
set_param([mdl /MPC Controller ], switch_inport , off );
set_param([mdl /Yplots ], Ymin , -1~-0.1 )
set_param([mdl /Yplots ], Ymax , 3~1.1 )
```

4 Case-Study Examples

```
set_param([mdl '/MVplots'], Ymin , -1.1~-5 )
set_param([mdl '/MVplots'], Ymax , 1.1~10 )
sim(mdl);
```





Now the transition from manual to automatic control is much less smooth. Note the large "bump" between time = 180 and 200.

```
bdclose(md1)
```

Related Examples

- “Bumpless Transfer Between Manual and Automatic Operations” on page 4-50

Coordinate Multiple Controllers at Different Operating Points

Chemical reactors can exhibit strongly nonlinear behavior due to the exponential effect of temperature on reaction rate. If the primary reaction is exothermic, an increase in reaction rate causes an increase in reactor temperature. This positive feedback can lead to open-loop unstable behavior.

Reactors operate in either a continuous or a batch mode. In batch mode, operating conditions can change dramatically during a batch as the reactants disappear. Although continuous reactors typically operate at steady state, they must often move to a new steady state. In other words, both batch and continuous reactors need to operate safely and efficiently over a range of conditions.

If the reactor behaves nonlinearly, a single linear controller might not be able to manage such transitions. One approach is to develop linear models that cover the anticipated operating range, design a controller based on each model, and then define a criterion by which the control system switches from one such controller to another. Gain scheduling is an established technique. The challenge is to move the reactor operating conditions from an initial steady-state point to a much different condition. The transition passes through a region in which the plant is open-loop unstable. This example illustrates an alternative — coordination of multiple MPC controllers. The solution uses the Simulink Multiple MPC Controller block to coordinate the use of three controllers, each of which has been designed for a particular operating region.

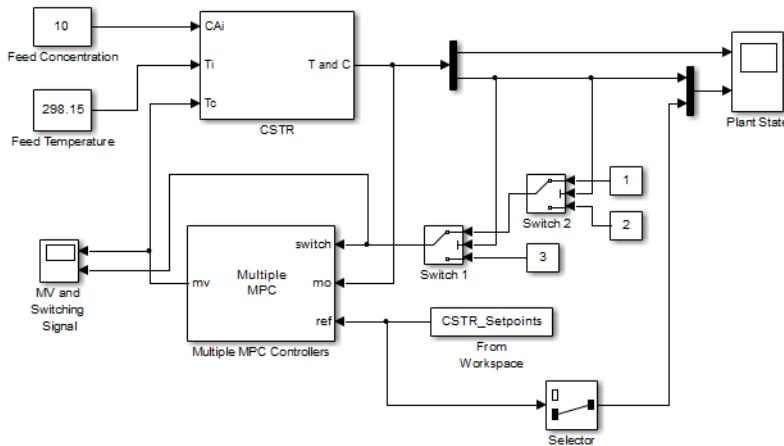
The subject process is a constant-volume continuous stirred-tank reactor (CSTR). The model consists of two nonlinear ordinary differential equations (see [1]). The model states are the reactor temperature and the rate-limiting reactant concentration. For the purposes of this example, both are assumed to be measured plant outputs.

There are three inputs:

- Concentration of the limiting reactant in the reactor feed stream, kmol/m^3
- The reactor feed temperature, K
- The coolant temperature, K

The control system can adjust the coolant temperature in order to regulate the reactor state and the rate of the exothermic main reaction. The other two inputs are independent unmeasured disturbances.

The Simulink diagram for this example appears below. The CSTR model is a masked subsystem. The feed temperature and composition are constants. As discussed above, the control system adjusts the coolant temperature (the T_c input on the CSTR block).



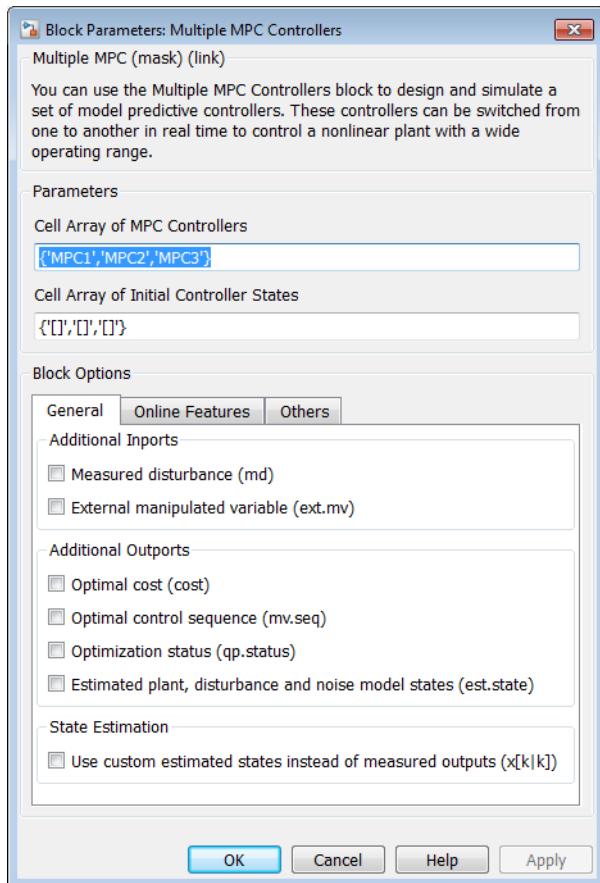
The two CSTR outputs are the reactor temperature and composition respectively. These are being sent to a scope display and to the control system as feedback.

The reference signal (i.e. setpoint) is coming from variable `CSTR_Setpoints`, which is in the base workspace. As there is only one manipulated variable (the coolant temperature) the control objective is to force the *reactor concentration* to track a specified trajectory. The concentration setpoint also goes to the `Plant State` scope for plotting. The control system receives a setpoint for the reactor temperature too but the controller design ignores it.

In that case why supply the temperature measurement to the controller? The main reason is to improve state estimation. If this were not done, the control system would have to infer the temperature value from the concentration measurement, which would introduce an estimation error and degrade the model's predictive accuracy.

The rationale for the `Switch 1` and `Switch 2` blocks appears below.

The figure below shows the `Multi MPC Controller` mask. The block is coordinating three controllers (`MPC1`, `MPC2` and `MPC3` in that sequence). It is also receiving the setpoint signal from the workspace, and the **Look ahead** option is active. This allows the controller to anticipate future setpoint values and usually improves setpoint tracking.



In order to designate which one of the three controllers is active at each time instant, we send the **Multi MPC Controllers** block a switching signal (connected to its **switch** input port). If it is 1, **MPC1** is active. If it is 2, **MPC2** is active, and so on.

In the diagram, **Switch 1** and **Switch 2** perform the controller selection function as follows:

- If the reactor concentration is 8 kmol/m^3 or greater, **Switch 1** sends the constant 1 to its output. Otherwise it sends the constant 2.
- If the reactor concentration is 3 kmol/m^3 or greater, **Switch 2** passes through the signal coming from **Switch 1** (either 1 or 2). Otherwise it sends the constant 3.

Thus, each controller handles a particular composition range. The simulation begins with the reactor at an initial steady state of 311K and 8.57 kmol/m³. The feed concentration is 10 kmol/m³ so this is a conversion of about 15%, which is low. The control objective is to transition smoothly to 80% conversion with the reactor concentration at 2 kmol/m³. The simulation will start with MPC1 active, transition to MPC2, and end with MPC3.

We decide to design the controllers around linear models derived at the following three reactor compositions (and the corresponding steady-state temperature): 8.5, 5.5, and 2 kmol/m³.

In practice, you would probably obtain the three models from data. This example linearizes the nonlinear model at the above three conditions (for details see “Using Simulink to Develop LTI Models” in the Getting Started Guide).

Note As shown later, we need to retain at the unmeasured plant inputs in the model. This prevents us from using the Model Predictive Control Toolbox automatic linearization feature. In the current toolbox, the automatic linearization feature can linearize with respect to manipulated variable and measured disturbance inputs only.

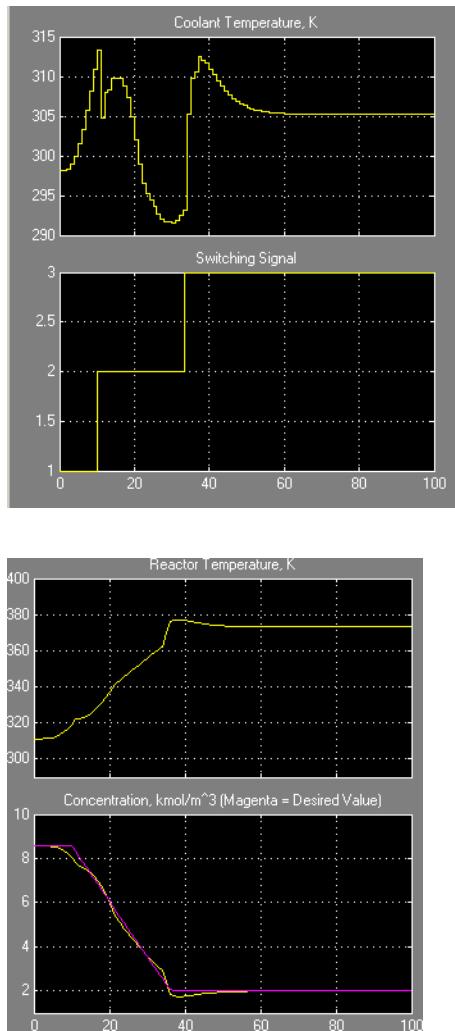
The following code obtains the linear models and designs the three controllers

```
[sys, xp] = CSTR_INOUT([],[],[], sizes );
up = [10 298.15 298.15] ;
yp = xp;
Ts = 1;
Nc = 3;
Controllers = cell(1,3);
Concentrations = [8.5 5.5 2];
Y = yp;
for i = 1:Nc
    clear Model
    Y(2) = Concentrations(i);
    [X,U,Y,DX] = trim( CSTR_INOUT ,xp(:),up(:),Y(:,[],[1,2] ,2)
    [a,b,c,d] = linmod( CSTR_INOUT , X, U );
    Plant = ss(a,b,c,d);
    Plant.InputGroup.MV = 3;
    Plant.InputGroup.UD = [1,2];
    Model.Plant = Plant;
    Model.Nominal.U = [0; 0; up(3)];
```

```
Model.Nominal.X = xp;
Model.Nominal.Y = yp;
MPCobj = mpc(Model, Ts);
MPCobj.Weight.OV = [0 1];
D = ss(getindist(MPCobj));
D.b = D.b*10;
set(D, InputName,[], OutputName,[], InputGroup,[], ...
    OutputGroup,[])
setindist(MPCobj, model, D)
Controllers{i} = MPCobj;
end
MPC1 = Controllers{1};
MPC2 = Controllers{2};
MPC3 = Controllers{3}
```

The key points regarding the designs are as follows:

- All three controllers use the same nominal condition, the values of the plant inputs and outputs at the initial steady-state. Exception: all unmeasured disturbance inputs must have zero nominal values.
- Each controller employs a different prediction model. The model structure is the same in each case (input and outputs are identical in number and type) but each model represents a particular steady-state reactor composition.
- It turns out that the MPC2 plant model obtained at 5 kmol/m³ is open-loop unstable. We must use a model structure that promotes a stable Kalman state estimator. If we include the unmeasured disturbance inputs in the prediction model, the default estimator assumes integrated white noise at each such input, which produces a stable estimator in this case.
- The default estimator signal-to-noise settings are inappropriate, however. If you use them and monitor the state estimates (not shown), the internally estimated temperature and composition can be far from the measured values. To overcome this, we increase the signal-to-noise ratio in each disturbance channel. See the use of **getindist** and **setindist** above. The default signal to noise is being increased by a factor of 10.
- We are using a zero weight on the measured temperature. See the above discussion of control objectives for the rationale.

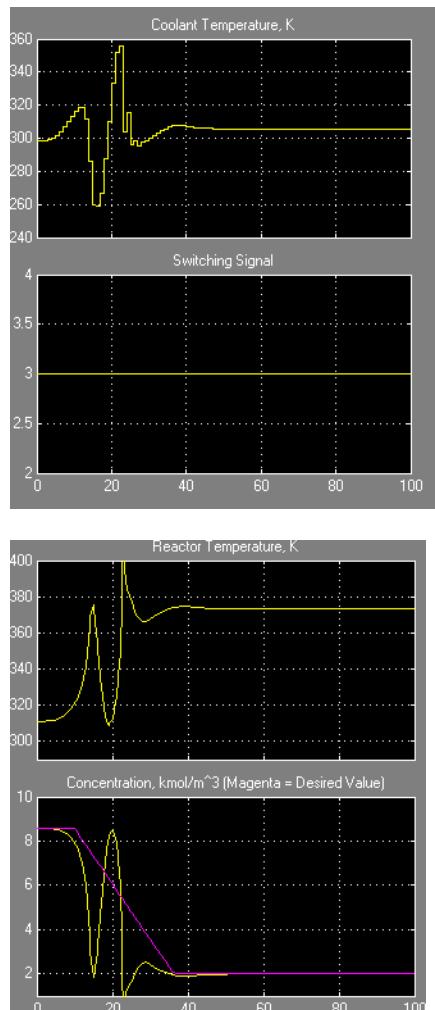


The above plots show the simulation results. The Multi MPC Controller block uses the three controllers sequentially as expected (see the switching signal). Tracking of the concentration setpoint is excellent and the reactor temperature is also controlled well.

To achieve this, the control system starts by increasing the coolant temperature, causing the reaction rate to increase. Once the reaction has achieved a high rate, it generates substantial heat and the coolant temperature must decrease to keep the

reactor temperature under control. As the reactor concentration depletes, the reaction rate slows and the control system must again raise the coolant temperature, finally settling at 305 K, about 7 K above the initial condition.

For comparison the plots below show the results for the same scenario if we force MPC3 to be active for the entire simulation. The CSTR eventually stabilizes at the desired steady-state but both the reactor temperature and composition exhibit large excursions away from the desired conditions.



References

- [1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp *Process Dynamics and Control*, 2nd Edition (2004), Wiley, pp. 34–36.

Use Custom Constraints in Blending Process

This example shows how to design an MPC controller for a blending process using custom input and output constraints.

Blending Process

A continuous blending process combines three feeds in a well-mixed container to produce a blend having desired properties. The dimensionless governing equations are:

$$\begin{aligned}\frac{dv}{d\tau} &= \sum_{i=1}^3 \phi_i - \phi \\ V \frac{d\gamma_j}{d\tau} &= \sum_{i=1}^3 (\gamma_{ij} - \gamma_j) \phi_i\end{aligned}$$

where

- V is the mixture inventory (in the container).
- ϕ_i is the plow rate for the i th feed.
- ϕ is the rate at which the blend is being removed from inventory, that is the demand.
- γ_{ij} is the concentration of constituent j in feed i .
- γ_j is the concentration of constituent j in the blend.
- τ is time.

In this example, there are two important constituents, $j = 1$ and 2 .

The control objectives are targets for the two constituent concentrations in the blend, and the mixture inventory. The challenge is that the demand, ϕ , and feed compositions, γ_{ij} , vary. The inventory, blend compositions, and demand are measured, but the feed compositions are unmeasured.

At the nominal operating condition:

- Feed 1, ϕ_1 , (mostly constituent 1) is 80% of the total inflow.
- Feed 2, ϕ_2 , (mostly constituent 2) is 20%.
- Feed 3, ϕ_3 , (pure constituent 1) is not used.

The process design allows manipulation of the total feed entering the mixing chamber, ϕ_T , and the individual rates of feeds 2 and 3. In other words, the rate of feed 1 is:

$$\phi_1 = \phi_T - \phi_2 - \phi_3$$

Each feed has limited availability:

$$0 \leq \phi_i \leq \phi_{i,\max}$$

The equations are normalized such that, at the nominal steady state, the mean residence time in the mixing container is $\tau = 1$.

The constraint $\phi_{1,\max} = 0.8$ is imposed by an upstream process, and the constraints $\phi_{2,\max} = \phi_{3,\max} = 0.6$ are imposed by physical limits.

Define Linear Plant Model

The blending process is mildly nonlinear, however you can derive a linear model at the nominal steady state. This approach is quite accurate unless the (unmeasured) feed compositions change. If the change is sufficiently large, the steady-state gains of the nonlinear process change sign and the closed-loop system can become unstable.

Specify the number of feeds, `ni`, and the number of constituents, `nc`.

```
ni = 3;
nc = 2;
```

Specify the nominal flow rates for the three input streams and the output stream, or demand. At the nominal operating condition, the output flow rate is equal to the sum of the input flow rates.

```
Fin_nom = [1.6,0.4,0];
F_nom = sum(Fin_nom);
```

Define the nominal constituent compositions for the input feeds, where `cin_nom(i,j)` represents the composition of constituent *i* in feed *j*.

```
cin_nom = [0.7 0.2 0.8;0.3 0.8 0];
```

Define the nominal constituent compositions in the output feed.

```
cout_nom = cin_nom*Fin_nom /F_nom;
```

Normalize the linear model such that the target demand is 1 and the product composition is 1.

```
fin_nom = Fin_nom/F_nom;
gij = [cin_nom(1,:)/cout_nom(1); cin_nom(2,:)/cout_nom(2)];
```

Create a state-space model with feed flows F1, F2, and F3 as MVs:

```
A = [zeros(1,nc+1); zeros(nc,1) -eye(nc)];
Bu = [ones(1,ni); gj];
```

Change the MV definition to [FT, F2, F3] where $F_1 = FT - F_2 - F_3$

```
Bu = [Bu(:,1), Bu(:,2)-Bu(:,1), Bu(:,3)-Bu(:,1)];
```

Add the measured disturbance, blend demand, as the 4th model input.

```
Bv = [-1; zeros(nc,1)];
B = [Bu Bv];
```

Define all of the states as measurable. The states consist of the mixture inventory and the constituent concentrations.

```
C = eye(nc+1);
```

Specify that there is no direct feed-through from the inputs to the outputs.

```
D = zeros(nc+1,ni+1);
```

Construct the linear plant model.

```
Model = ss(A,B,C,D);
Model.InputName = { F_T , F_2 , F_3 , F };
ModelInputGroup.MV = 1:3;
ModelInputGroup.MD = 4;
Model.OutputName = { V , c_1 , c_2 };
```

Create MPC Controller

Specify the sample time, prediction horizon, and control horizon.

```
Ts = 0.1;
p = 10;
```

```
m = 3;
```

Create the controller.

```
mpcobj = mpc(Model,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

The outputs are the inventory, $y(1)$, and the constituent concentrations, $y(2)$ and $y(3)$. Specify nominal values of unity after normalization for all outputs.

```
mpcobj.Model.Nominal.Y = [1 1 1];
```

Specify the normalized nominal values the manipulated variables, $u(1)$, $u(2)$ and $u(3)$, and the measured disturbance, $u(4)$.

```
mpcobj.Model.Nominal.U = [1 fin_nom(2) fin_nom(3) 1];
```

Specify output tuning weights. Larger weights are assigned to the first two outputs because we want to pay more attention to controlling the inventory, and the composition of the first constituent.

```
mpcobj.Weights.OV = [1 1 0.5];
```

Specify the hard bounds (physical limits) on the manipulated variables.

```
umin = [0 0 0];
umax = [2 0.6 0.6];
for i = 1:3
    mpcobj.MV(i).Min = umin(i);
    mpcobj.MV(i).Max = umax(i);
    mpcobj.MV(i).RateMin = -0.1;
    mpcobj.MV(i).RateMax = 0.1;
end
```

The total feed rate and the rates of feed 2 and feed 3 have upper bounds. Feed 1 also has an upper bound, determined by the upstream unit supplying it.

Specify Custom Constraints

Given the specified upper bounds on the feed 2 and 3 rates (0.6), it is possible that their sum could be as much as 1.2. Since the nominal total feed rate is 1.0, the controller can

request a physically impossible condition, where the sum of feeds 2 and 3 exceeds the total feed rate, which implies a negative feed 1 rate.

The following constraint prevents the controller from requesting an unrealistic ϕ_1 value.

$$0 \leq \phi_1 = \phi_T - \phi_2 - \phi_3 \leq 0.8$$

Specify this constraint in the form $Eu + Fy \leq g$.

```
E = [-1 1 1; 1 -1 -1];
g = [0;0.8];
```

Since no outputs are specified in the mixed constraints, set their coefficients to zero.

```
F = zeros(2,3);
```

Specify that both constraints are hard (ECR = 0).

```
v = zeros(2,1);
```

Specify zero coefficients for the measured disturbance.

```
h = zeros(2,1);
```

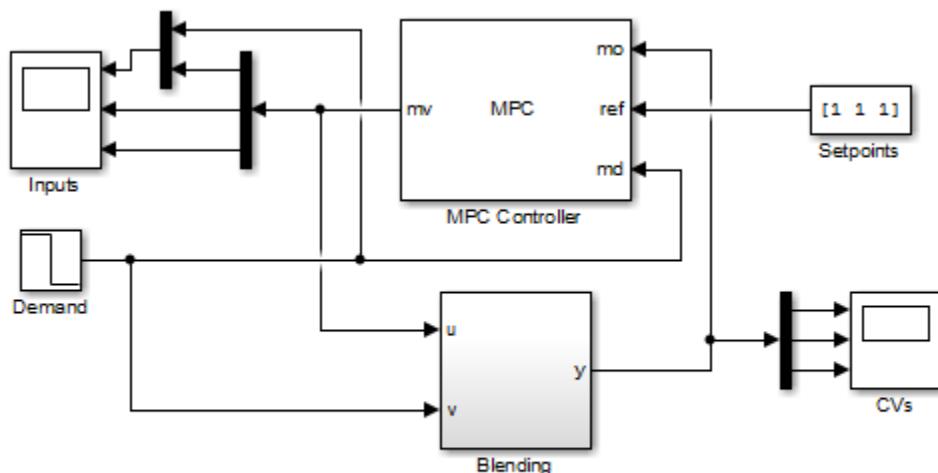
Set the custom constraints in the MPC controller.

```
setconstraint(mpcobj,E,F,g,v,h)
```

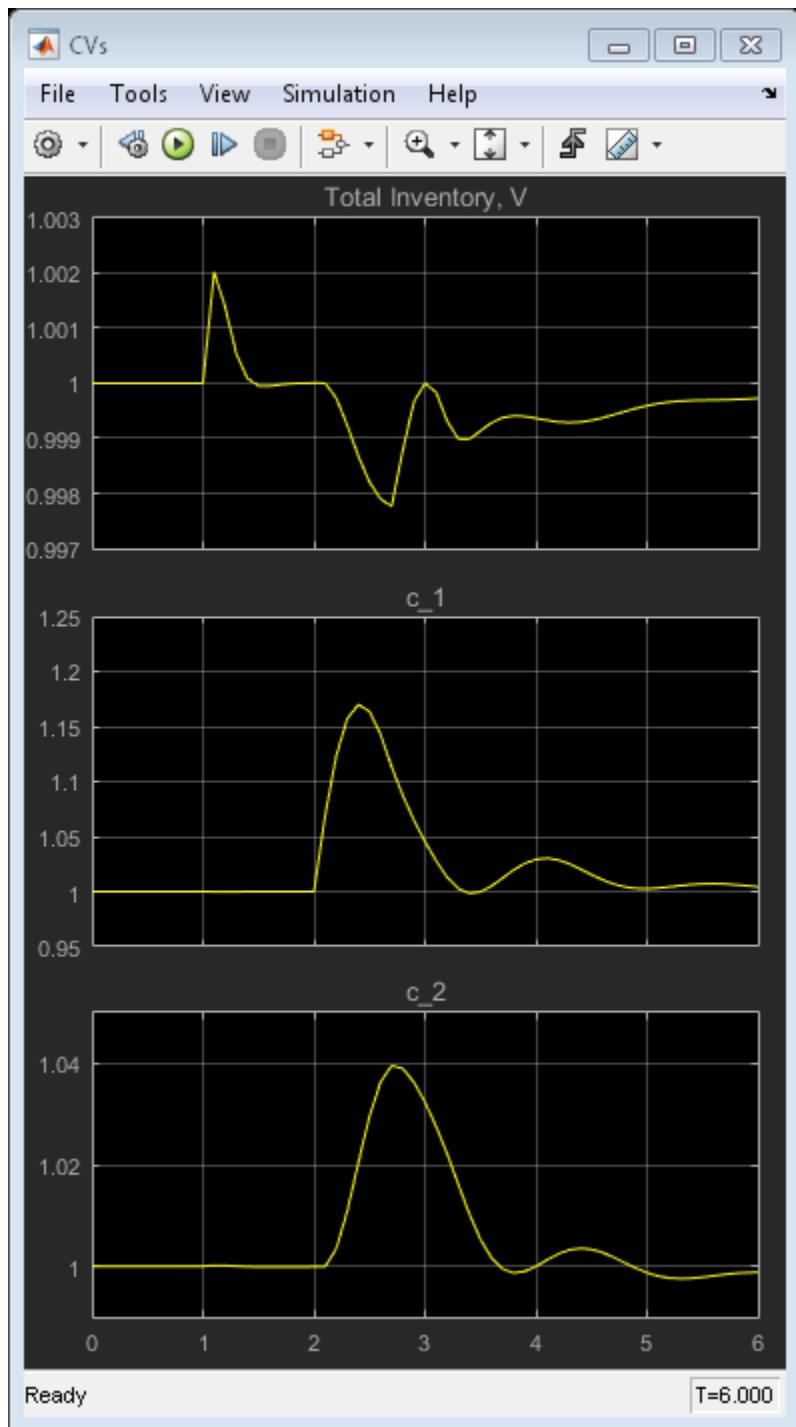
Open and Simulate Model in Simulink

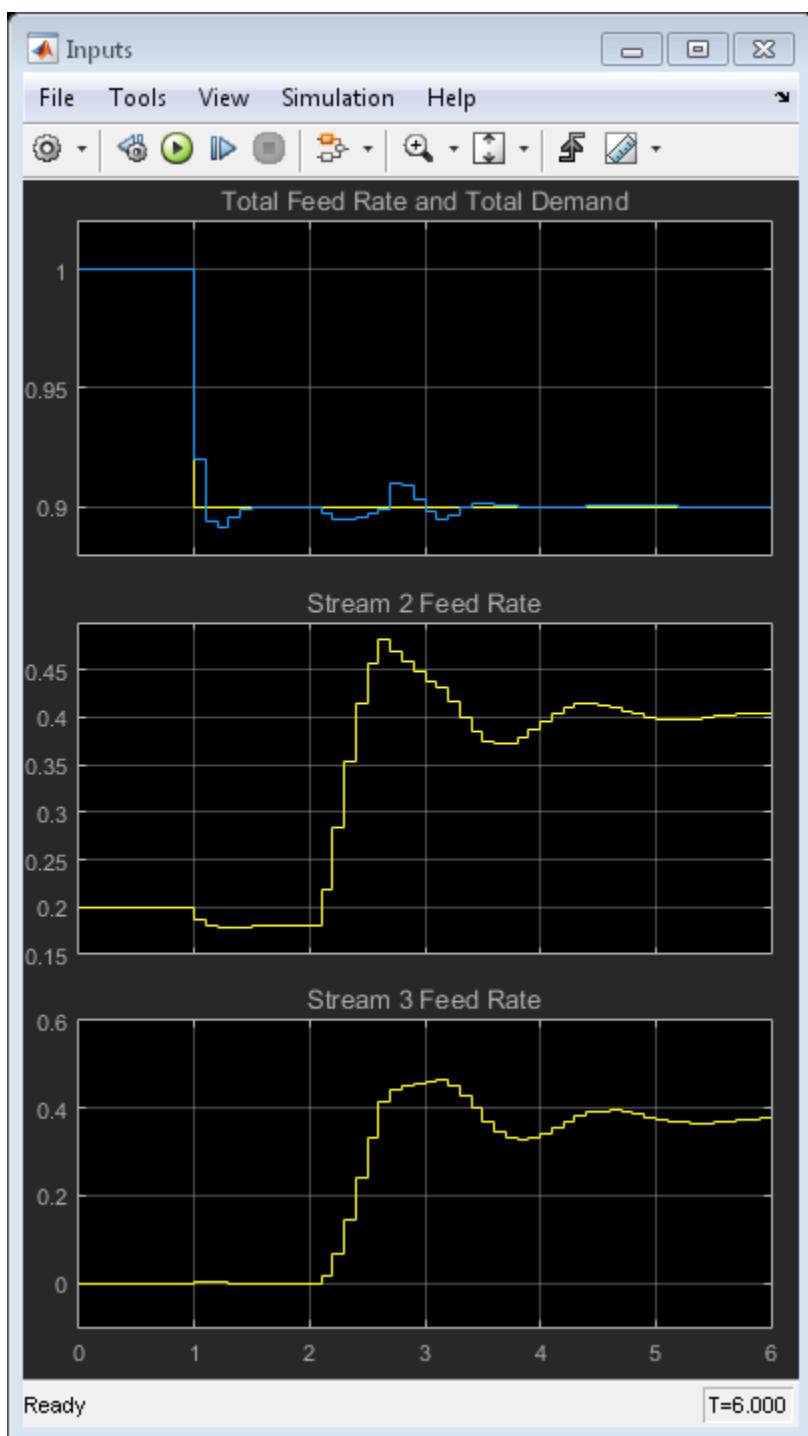
```
sys = mpc_blendingprocess ;
open_system(sys)
sim(sys)

-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```



Copyright 1990-2014 The MathWorks, Inc.





The MPC controller controls the blending process. The block labeled **Blending** incorporates the previously described model equations and includes an unmeasured step disturbance in the constituent 1 feed composition.

The **Demand**, ϕ , is modeled as a measured disturbance. The operator can vary the demand value, and the resulting signal goes to both the process and the controller.

The model simulates the following scenario:

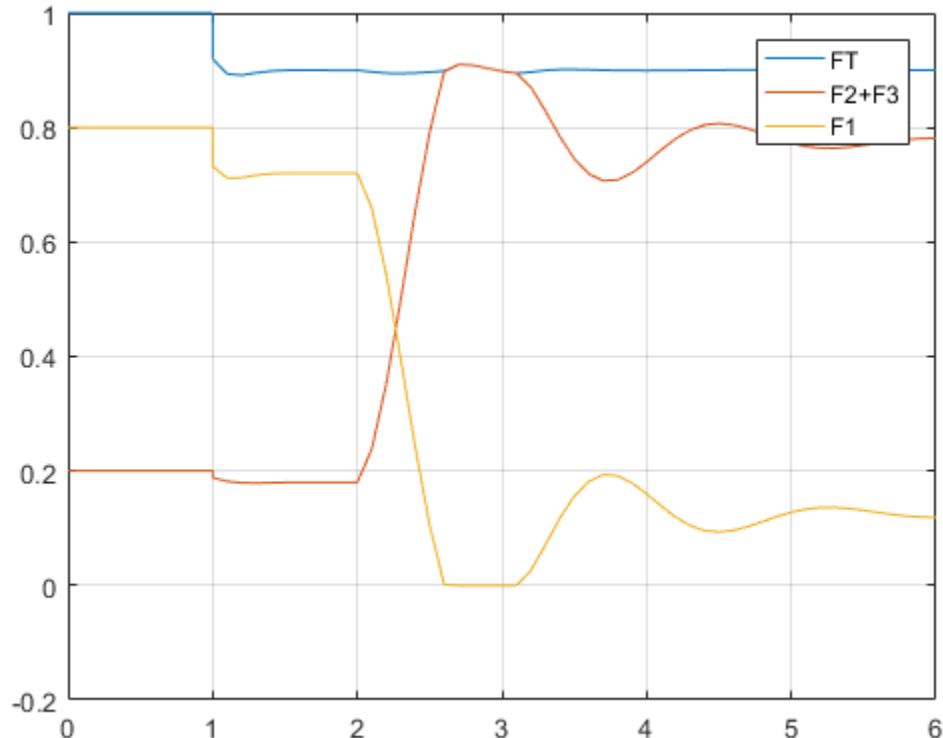
- At $\tau = 0$, the process is operating at steady state.
- At $\tau = 1$, the **Total Demand** decreases from $\phi = 1.0$ to $\phi = 0.9$.
- At $\tau = 2$, there is a large step increase in the concentration of constituent 1 in feed 1, from 1.17 to 2.17.

The controller maintains the inventory very close to its setpoint, but the severe disturbance in the feed composition causes a prediction error and a large disturbance in the blend composition, especially for constituent 1, c_1 . However, the controller recovers and drives the blend composition back to its setpoint.

Verify Effect of Custom Constraints

Plot the feed rate signals.

```
figure
plot(MVs.time,[MVs.signals(1).values(:,2), ...
    (MVs.signals(2).values + MVs.signals(3).values), ...
    (MVs.signals(1).values(:,2)-MVs.signals(2).values-MVs.signals(3).values)])
grid
legend( FT , F2+F3 , F1 )
```



The total feed rate, **FT**, and the sum of feed rates **F2** and **F3** coincide for $1.7 \leq \tau \leq 2.2$. If the custom input constraints had not been included, the controller would have requested an impossible negative feed 1 rate, **F1**, during this period.

```
bdclose(sys)
```

See Also

[setconstraint](#)

Related Examples

- MPC Control with Constraints on a Combination of Input and Output Signals

- MPC Control of a Nonlinear Blending Process

More About

- “Constraints on Linear Combinations of Inputs and Outputs” on page 2-33

Providing LQR Performance Using Terminal Penalty

This example, from Scokaert and Rawlings [1], shows how to make a finite-horizon Model Predictive Controller equivalent to an infinite-horizon linear quadratic regulator (LQR).

The “Standard Cost Function” on page 2-2 is similar to that used in an LQR controller with output weighting, as shown in the following equation:

$$J(u) = \sum_{i=1}^{\infty} y(k+i)^T Q y(k+i) + u(k+i-1)^T R u(k+i-1)$$

The LQR and MPC cost functions differ in the following ways:

- The LQR cost function forces y and u towards zero whereas the MPC cost function forces y and u toward nonzero setpoints.
You can shift the MPC prediction model’s origin to eliminate this difference and achieve zero setpoints at nominal condition.
- The LQR cost function uses an infinite prediction horizon in which the manipulated variable changes at each sampling instant. In the standard MPC cost function, the horizon length is p , and the manipulated variable changes m times, where m is the control horizon.

The two cost functions are equivalent if the MPC cost function is:

$$J(u) = \sum_{i=1}^{p-1} y(k+i)^T Q y(k+i) + u(k+i-1)^T R u(k+i-1) + x(k+p)^T Q_p x(k+p)$$

where Q_p is a penalty applied at the last (i.e., terminal) prediction horizon step, and the prediction and control horizons are equal, i.e., $p = m$. The required Q_p is the Riccati matrix that you can calculate using the Control System Toolbox `lqr` and `lqry` commands. The value is a positive definite symmetric matrix.

The following procedure shows how to design an unconstrained MPC controller that provides performance equivalent to a LQR controller:

- 1 Define a plant with one input and two outputs.

The plant is a double-integrator, represented as a state-space model in discrete-time with sampling interval 0.1 seconds.

```

A = [1 0;0.1 1];
B = [0.1;0.005];
C = eye(2);
D = zeros(2,1);
Ts = 0.1;
Plant = ss(A,B,C,D,Ts);
Plant.InputName = { u };
Plant.OutputName = { x_1 , x_2 };

```

- 2** Design an LQR controller with output feedback for the plant.

```

Q = eye(2);
R = 1;
[K,Qp] = lqry(Plant,Q,R);

```

Q and R are output and input weight matrices, respectively. Q_p is the Riccati matrix.

- 3** Design an MPC controller equivalent to the LQR controller.

To implement Equation 4-2, compute L , the Cholesky decomposition of Q_p , such that $L^T L = Q_p$. Then, define auxiliary unmeasured output variables $y_a(k) = Lx(k)$ such that $y_a^T y_a = x^T Q_p x$. For the first $p - 1$ prediction horizon steps, the standard Q and R weights apply to the original u and y , and y_a has a zero penalty. On step p , the original u and y have zero penalties, and y_a has a unity penalty.

- a** Augment the plant model, and specify the augmented outputs as unmeasured.

```

NewPlant = Plant;
cholP = chol(Qp);
set(NewPlant, C ,[C;cholP], D ,[D;zeros(2,1)],...
    OutputName ,{ x_1 , x_2 , Cx_1 , Cx_2 });
NewPlant.InputGroup.MV = 1;
NewPlant.OutputGroup.MO = [1 2];
NewPlant.OutputGroup.UO = [3 4];

```

- b** Create an MPC controller with equal prediction and control horizons.

```

P = 3;
M = 3;
MPCobj = mpc(NewPlant,Ts,P,M);

```

```

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming empty manipulated variables
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming empty manipulated variables rate
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming empty output variables

```

```
for output(s) y1 and zero weight for output(s) y2 y3 y4
```

When there are no constraints, you can use a rather short horizon (in this case, $p \geq 1$ gives identical results).

- c Specify weights for manipulated variables (MV) and output variables (OV).

```
ywt = sqrt(diag(Q)) ;
uwt = sqrt(diag(R)) ;
MPCobj.Weights.OV = [ywt 0 0];
MPCobj.Weights.MV = uwt;
MPCobj.Weights.MVrate = 1e-6;
```

The two augmented outputs have zero weights during the prediction horizon.

- d Specify terminal weights.

To obtain the desired effect, define unity weights for these at the final point in the horizon.

```
U = struct( Weight , uwt);
Y = struct( Weight , [0 0 1 1]);
setterminal(MPCobj, Y, U)
```

The first two states receive zero weight at the terminal point, and the input weight is unchanged.

- e Remove default state estimator.

The model states are measured directly, so the default MPC state estimator is unnecessary.

```
setoutdist(MPCobj, model ,tf(zeros(4,1)))
setEstimator(MPCobj,[],C)
```

The **setoutdist** command removes the output disturbances from the output channels, and the **setEstimator** command sets the controller state estimates equal to the measured output values.

- 4 Compare the control performance of LQR, MPC with terminal weights, and a standard MPC.

- a Compute closed-loop response with LQR controller.

```
clsys = feedback(Plant,K);
```

```
Tstop = 6;
x0 = [0.2;0.2];
[yLQR,tLQR] = initial(clsys,x0,Tstop);
```

- b** Compute closed-loop response with MPC with terminal weights.

```
SimOptions = mpccsimopt(MPCobj);
SimOptions.PlantInitialState = x0;
r = zeros(1,4);
[y,t,u] = sim(MPCobj,ceil(Tstop/Ts),r,SimOptions);
Cost = sum(sum(y(:,1:2)*diag(ywt).*y(:,1:2))) + sum(u*diag(uwt).*u);

-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise
```

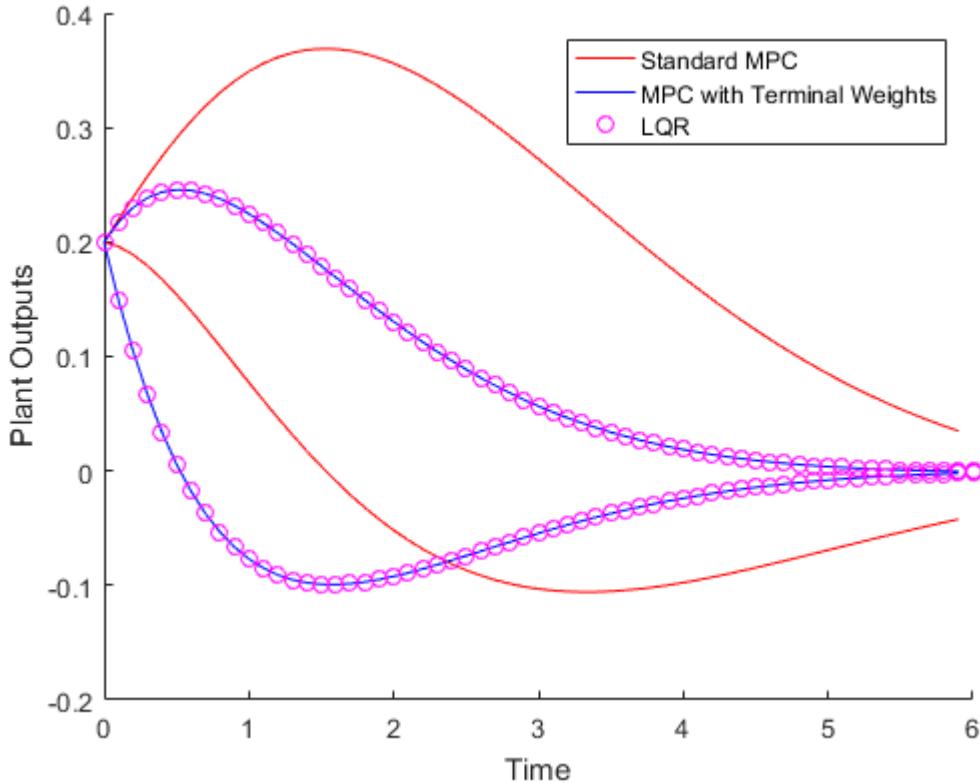
- c** Compute closed-loop response with standard MPC controller.

```
MPCobjSTD = mpc(Plant,Ts); % Default P = 10, M = 2
MPCobjSTD.Weights.MV = uwt;
MPCobjSTD.Weights.MVrate = 1e-6;
MPCobjSTD.Weights.OV = ywt;
SimOptions = mpccsimopt(MPCobjSTD);
SimOptions.PlantInitialState = x0;
r = zeros(1,2);
[ySTD,tSTD,uSTD] = sim(MPCobjSTD,ceil(Tstop/Ts),r,SimOptions);
CostSTD = sum(sum(ySTD*diag(ywt).*ySTD)) + sum(uSTD*uwt.*uSTD);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying Prediction
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. As
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming de
    for output(s) y1 and zero weight for output(s) y2
-->Assuming output disturbance added to measured output channel #1 is integrated
    Assuming no disturbance added to measured output channel #2.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise
```

- d** Compare the responses.

```
figure
h1 = line(tSTD,ySTD, color , r );
h2 = line(t,y(:,1:2), color , b );
h3 = line(tLQR,yLQR, color , m , marker , o , linestyle , none );
xlabel( Time )
ylabel( Plant Outputs )
legend([h1(1) h2(1) h3(1)], Standard MPC , MPC with Terminal Weights , LQR , Lo
```



The plot shows that the MPC controller with the terminal weights provides faster settling to the origin than the standard MPC. The LQR controller and MPC with terminal weights provide identical control performance.

As reported by Scokaert and Rawlings [1], the computed **Cost** value is 2.23, identical to that provided by the LQR controller. The computed **CostSTD** value for the standard MPC is 4.82, more than double compared to **Cost**.

You can improve the standard MPC by retuning. For example, use the same state estimation strategy. If the prediction and control horizons are then increased, it provides essentially the same performance.

This example shows that using a terminal penalty can eliminate the need to tune the MPC prediction and control horizons for the unconstrained case. If your application includes constraints, using a terminal weight is insufficient to guarantee nominal stability. You must also choose appropriate horizons and possibly add terminal constraints. For an in-depth discussion, see Rawlings and Mayne [2].

Although you can design and implement such a controller in Model Predictive Control Toolbox software, you might find designing the standard MPC controller more convenient.

References

- [1] Scokaert, P. O. M. and J. B. Rawlings “Constrained linear quadratic regulation” *IEEE Transactions on Automatic Control* (1998), Vol. 43, No. 8, pp. 1163-1169.
- [2] Rawlings, J. B., and David Q. Mayne “Model Predictive Control: Theory and Design” Nob Hill Publishing, 2010.

Related Examples

- “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”

More About

- “Terminal Weights and Constraints” on page 2-30

Real-Time Control with OPC Toolbox

This example shows how to implement an online model predictive controller application using the OPC client supplied with the OPC Toolbox™.

The example uses the Matrikon™ Simulation OPC server to simulate the behavior of an industrial process on Windows® operating system.

Download the Matrikon™ OPC Simulation Server from "www.matrikon.com"

Download and install the server and set it running either as a service or as an application.

This example needs OPC Toolbox™.

```
if ~mpcchecktoolboxinstalled( opc )
    disp( 'The example needs OPC Toolbox(TM).')
end
```

The example needs OPC Toolbox(TM).

Establish a Connection to the OPC Server

Use OPC Toolbox commands to connect to the Matrikon OPC Simulation Server.

```
if mpcchecktoolboxinstalled( opc )
    % Clear any existing opc connections.
    opcreset
    % Flush the callback persistent variables.
    clear mpcopcPlantStep;
    clear mpcopcMPCStep;
    try
        h = opcda( 'localhost' , 'Matrikon.OPC.Simulation.1' );
        connect(h);
    catch ME
        disp( 'The Matrikon(TM) OPC Simulation Server must be running on the local machine.' );
        return
    end
end
```

Set up the Plant OPC I/O

In practice the plant would be a physical process, and the OPC tags which define its I/O would already have been created on the OPC server. However, since in this case

a simulation OPC server is being used, the plant behavior must be simulated. This is achieved by defining tags for the plant manipulated and measured variables and creating a callback (mpcopcPlantStep) to simulate plant response to changes in the manipulated variables. Two OPC groups are required, one to represent the two manipulated variables to be read by the plant simulator and another to write back the two measured plant outputs storing the results of the plant simulation.

```

if mpcchecktoolboxinstalled( opc )
    % Build an opc group for 2 plant inputs and initialize them to zero.
    plant_read = addgroup(h, plant_read );
    imv1 = additem(plant_read, Bucket Brigade.Real8 , double );
    writeasync(imv1,0);
    imv2 = additem(plant_read, Bucket Brigade.Real4 , double );
    writeasync(imv2,0);
    % Build an opc group for plant outputs.
    plant_write = addgroup(h, plant_write );
    opv1 = additem(plant_write, Bucket Brigade.Time , double );
    opv2 = additem(plant_write, Bucket Brigade.Money , double );
    plant_write.WriteAsyncFcn = [];% Suppress command line display.
end

```

Specify the MPC Controller Which Will Control the Simulated Plant

Create plant model.

```

plant_model = ss([- .2 -.1; 0 -.05],eye(2,2),eye(2,2),zeros(2,2));
disc_plant_model = c2d(plant_model,1);
% We assume no model mismatch, a control horizon 6 samples and
% prediction horizon 20 samples.
mpcobj = mpc(disc_plant_model,1,20,6);
mpcobj.Weights.ManipulatedVariablesRate = [1 1];
% Build an internal MPC object structure so that the MPC object
% is not rebuilt each callback execution.
state = mpcstate(mpcobj);
y1 = mpcmove(mpcobj,state,[1;1],[1 1] );

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each

```

Build the OPC I/O for the MPC Controller

Build two OPC groups, one to read the two measured plant outputs and the other to write back the two manipulated variables.

```
if mpcchecktoolboxinstalled( opc )
    % Build an opc group for MPC inputs.
    mpc_read = addgroup(h, mpc_read );
    impcpv1 = additem(mpc_read, Bucket Brigade.Time , double );
    writeasync(impcpv1,0);
    impcpv2 = additem(mpc_read, Bucket Brigade.Money , double );
    writeasync(impcpv2,0);
    impcref1 = additem(mpc_read, Bucket Brigade.Int2 , double );
    writeasync(impcref1,1);
    impcref2 = additem(mpc_read, Bucket Brigade.Int4 , double );
    writeasync(impcref2,1);
    % Build an opc group for mpc outputs.
    mpc_write = addgroup(h, mpc_write );
    additem(mpc_write, Bucket Brigade.Real8 , double );
    additem(mpc_write, Bucket Brigade.Real4 , double );
    % Suppress command line display.
    mpc_write.WriteAsyncFcn = [];
end
```

Build OPC Groups to Trigger Execution of the Plant Simulator & Controller

Build two opc groups based on the same external opc timer to trigger execution of both plant simulation and MPC execution when the contents of the OPC time tag changes.

```
if mpcchecktoolboxinstalled( opc )
    gtime = addgroup(h, time );
    time_tag = additem(gtime, Triangle Waves.Real8 );
    gtime.UpdateRate = 1;
    gtime.DataChangeFcn = {@mpcopcPlantStep plant_read plant_write disc_plant_model};
    gmpctime = addgroup(h, mpctime );
    additem(gmpctime, Triangle Waves.Real8 );
    gmpctime.UpdateRate = 1;
    gmpctime.DataChangeFcn = {@mpcopcMPCStep mpc_read mpc_write mpcobj};
end
```

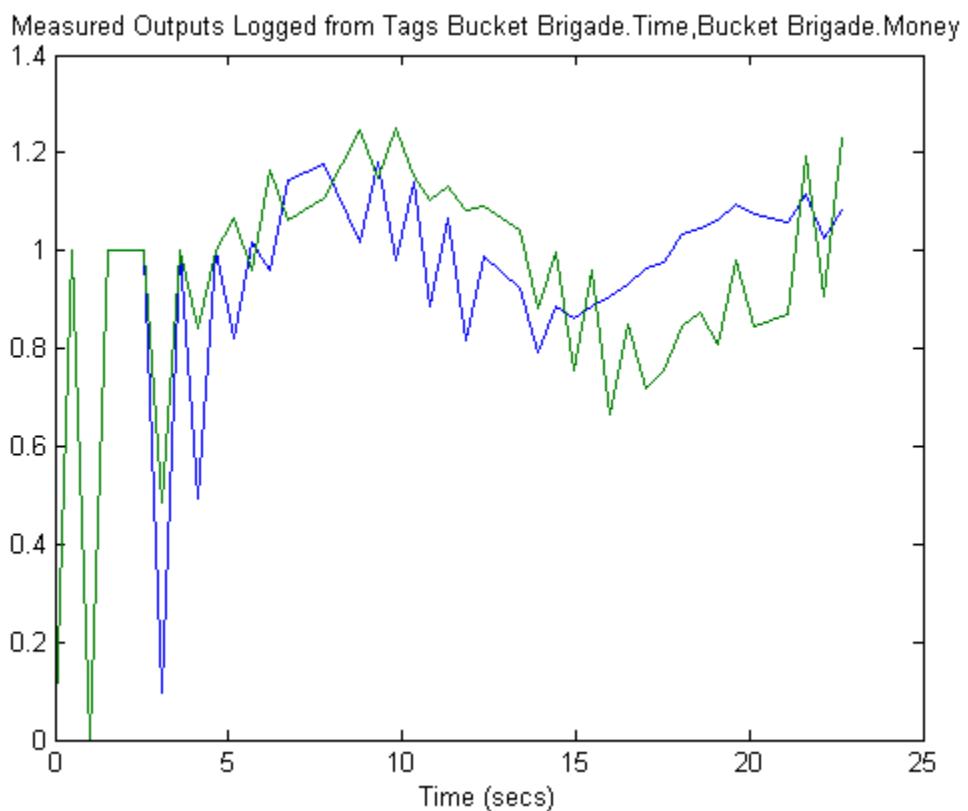
Log Data from the Plant Measured Outputs

Log the plant measured outputs from tags 'Bucket Brigade.Money' and 'Bucket Brigade.Money'.

```
if mpcchecktoolboxinstalled( opc )
    mpc_read.RecordsToAcquire = 40;
    start(mpc_read);
    while mpc_read.RecordsAcquired < mpc_read.RecordsToAcquire
        pause(3)
        fprintf( Logging data: Record %d / %d ,mpc_read.RecordsAcquired,mpc_read.Records
    end
    stop(mpc_read);
end
```

Extract and Plot the Logged Data

```
if mpcchecktoolboxinstalled( opc )
    [itemID, value, quality, timeStamp, eventTime] = getdata(mpc_read, double );
    plot((timeStamp(:,1)-timeStamp(1,1))*24*60*60,value)
    title( Measured Outputs Logged from Tags Bucket Brigade.Time,Bucket Brigade.Money )
    xlabel( Time (secs) );
end
```



Simulation and Code Generation Using Simulink Coder

This example shows how to simulate and generate real-time code for an MPC Controller block with Simulink Coder. Code can be generated in both single and double precisions.

Required Products

To run this example, Simulink® and Simulink® Coder™ are required.

```
if ~mpcchecktoolboxinstalled( 'simulink' )
    disp( 'Simulink(R) is required to run this example. ' )
    return
end
if ~mpcchecktoolboxinstalled( 'simulinkcoder' )
    disp( 'Simulink(R) Coder(TM) is required to run this example. ' );
    return
end
```

Setup Environment

You must have write-permission to generate the relevant files and the executable. So, before starting simulation and code generation, change the current directory to a temporary directory.

```
cwd = pwd;
tmpdir = tempname;
mkdir(tmpdir);
cd(tmpdir);
```

Define Plant Model and MPC Controller

Define a SISO plant.

```
plant = ss(tf([3 1],[1 0.6 1]));
```

Define the MPC controller for the plant.

```
Ts = 0.1; %Sampling time
p = 10; %Prediction horizon
m = 2; %Control horizon
Weights = struct( 'MV ',0, 'MVRate ',0.01, 'OV ',1); % Weights
MV = struct( 'Min ',-Inf, 'Max ',Inf, 'RateMin ', -100, 'RateMax ',100); % Input constraints
OV = struct( 'Min ',-2, 'Max ',2); % Output constraints
```

```
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

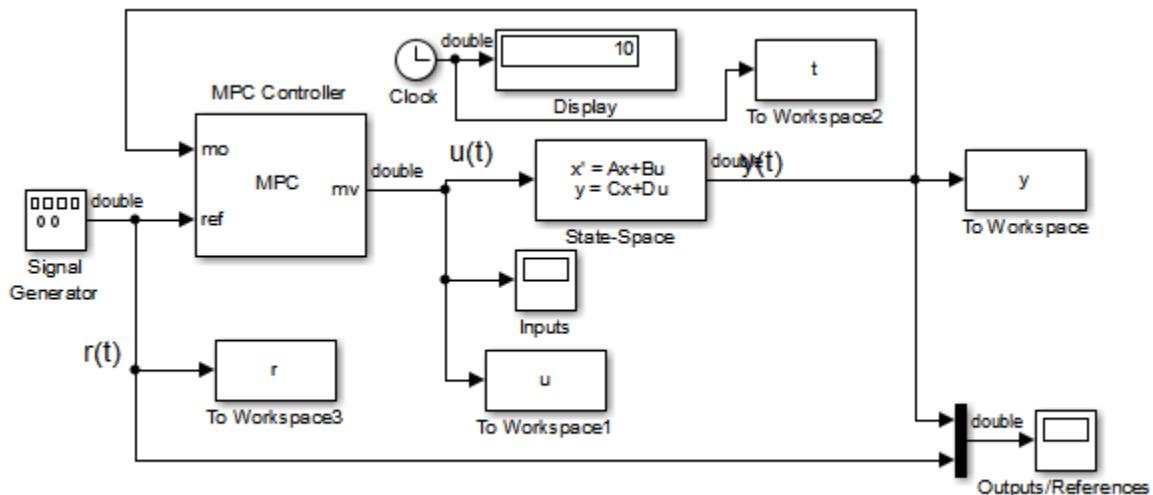
Simulate and Generate Code in Double-Precision

By default, MPC Controller blocks use double-precision in simulation and code generation.

Simulate the model in Simulink.

```
mdl1 = mpc_rtwdemo ;
open_system(mdl1);
sim(mdl1);

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```



Copyright 1990-2014 The MathWorks, Inc.

The controller effort and the plant output are saved into base workspace as variables **u** and **y**, respectively.

Build the model with the **rtwbuild** command.

```
disp( Generating C code... Please wait until it finishes. );
```

```
set_param(mdl1, RTWVerbose , off );
rtwbuild(mdl1);

Generating C code... Please wait until it finishes.
### Starting build procedure for model: mpc_rtwdemo
### Successful completion of build procedure for model: mpc_rtwdemo
```

On a Windows system, an executable file named "mpc_rtwdemo.exe" appears in the temporary directory after the build process finishes.

Run the executable.

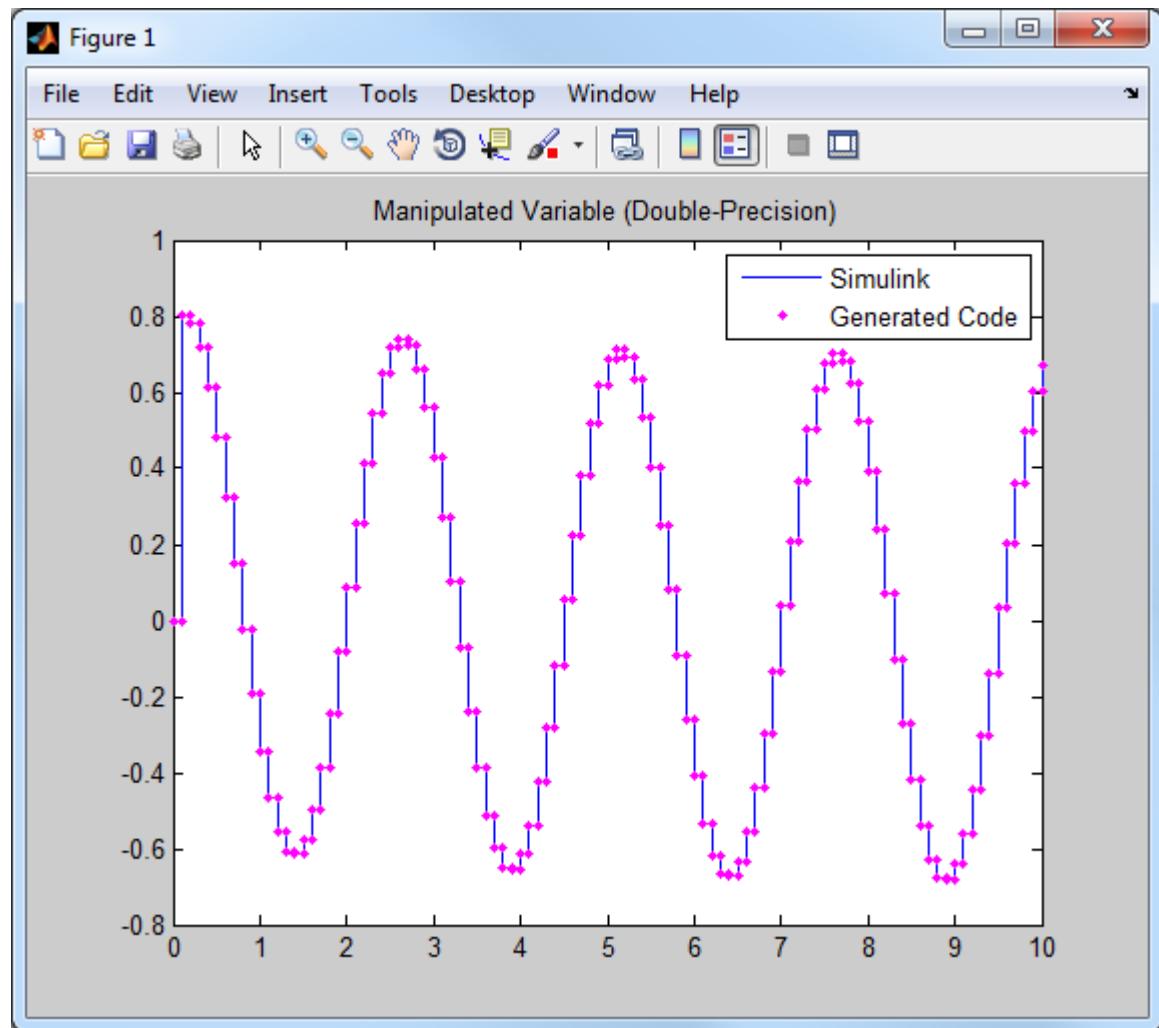
```
if ispc
    disp( Running executable... );
    status = system(mdl1);
else
    disp( The example only runs the executable on Windows system. );
end

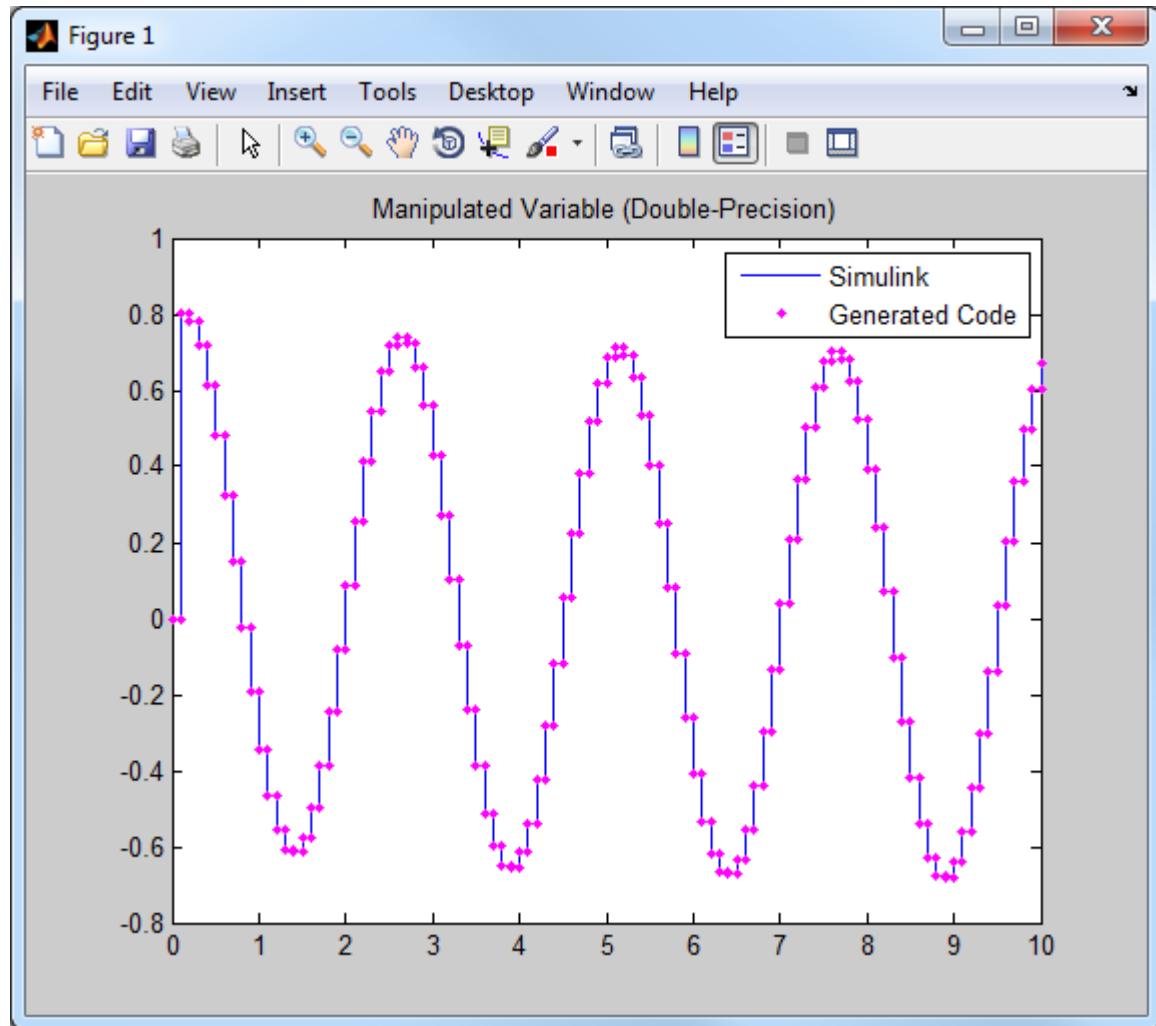
Running executable...

** starting the model **
** created mpc_rtwdemo.mat **
```

After the executable completes successfully (status=0), a data file named "mpc_rtwdemo.mat" appears in the temporary directory.

Compare the responses from the generated code (**rt_u** and **rt_y**) with the responses from the previous simulation in Simulink (**u** and **y**).





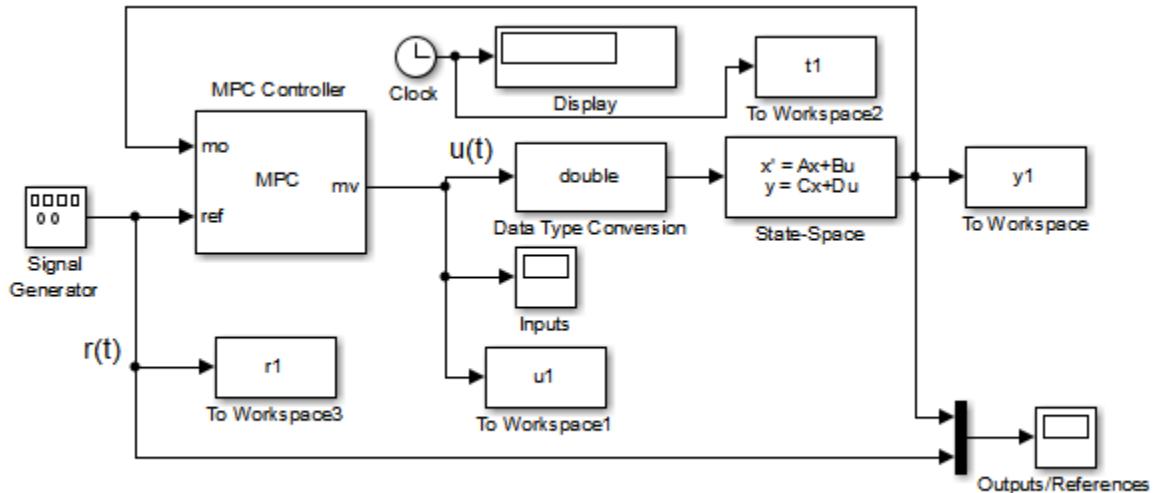
They are numerically equal.

Simulate and Generate Code in Single-Precision

You can also configure the MPC block to use single-precision in simulation and code generation.

```
mdl2 = mpc_rtwdemo_single ;
```

```
open_system(mdl2);
```



Copyright 1990-2014 The MathWorks, Inc.

To do that, open the MPC block dialog and select "single" as the "output data type" at the bottom of the dialog.

```
open_system([mdl2 /MPC_Controller ]);
```

Simulate the model in Simulink.

```
close_system([mdl2 /MPC_Controller ]);
sim(mdl2);
```

The controller effort and the plant output are saved into base workspace as variables **u1** and **y1**, respectively.

Build the model with the **rtwbuild** command.

```
disp( Generating C code... Please wait until it finishes. );
set_param(mdl2, RTWVerbose , off );
rtwbuild(mdl2);
```

Generating C code... Please wait until it finishes.

```
### Starting build procedure for model: mpc_rtwdemo_single
### Successful completion of build procedure for model: mpc_rtwdemo_single
```

On a Windows system, an executable file named "mpc_rtwdemo_single.exe" appears in the temporary directory after the build process finishes.

Run the executable.

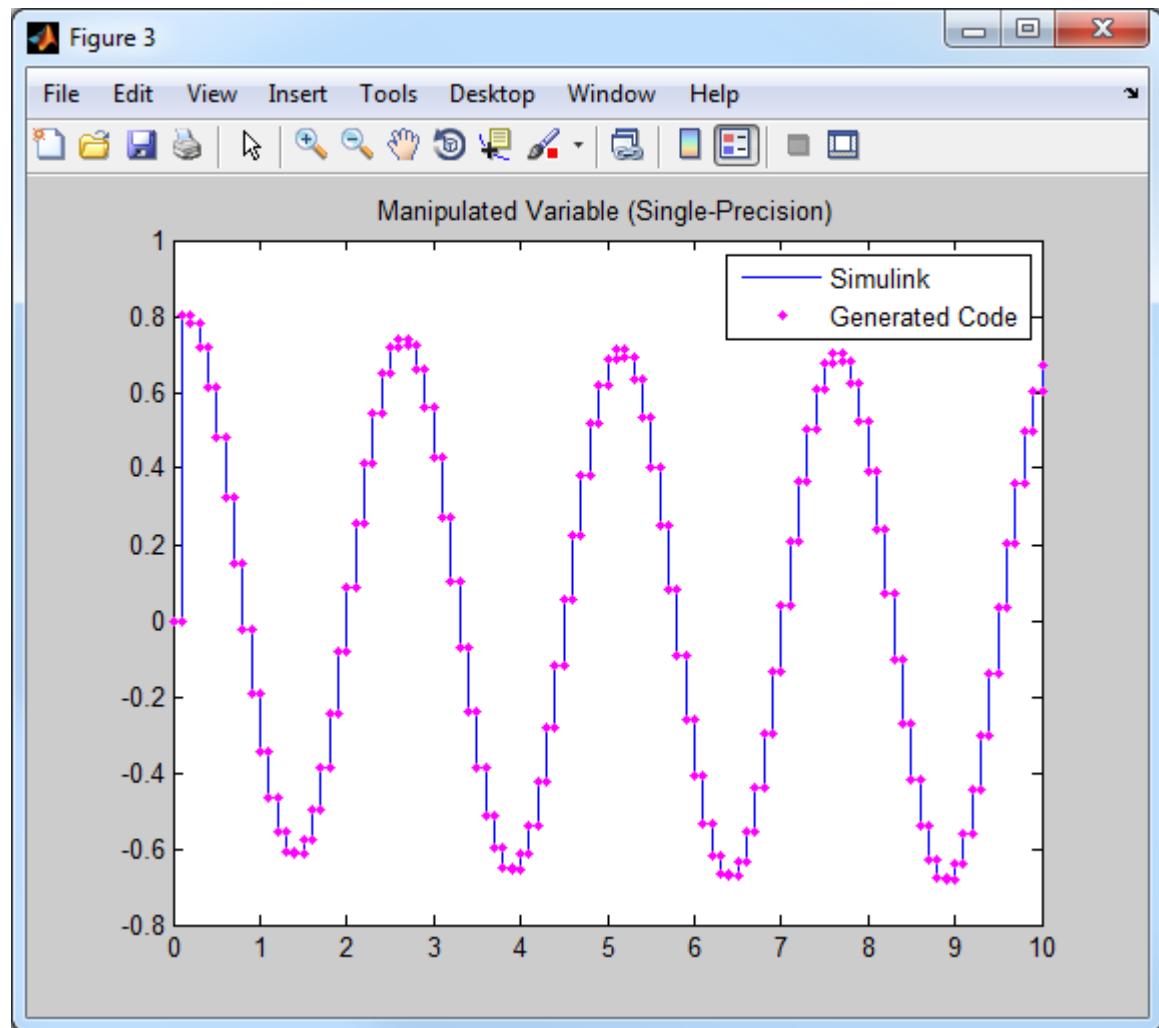
```
if ispc
    disp( Running executable... );
    status = system(md12);
else
    disp( The example only runs the executable on Windows system. );
end

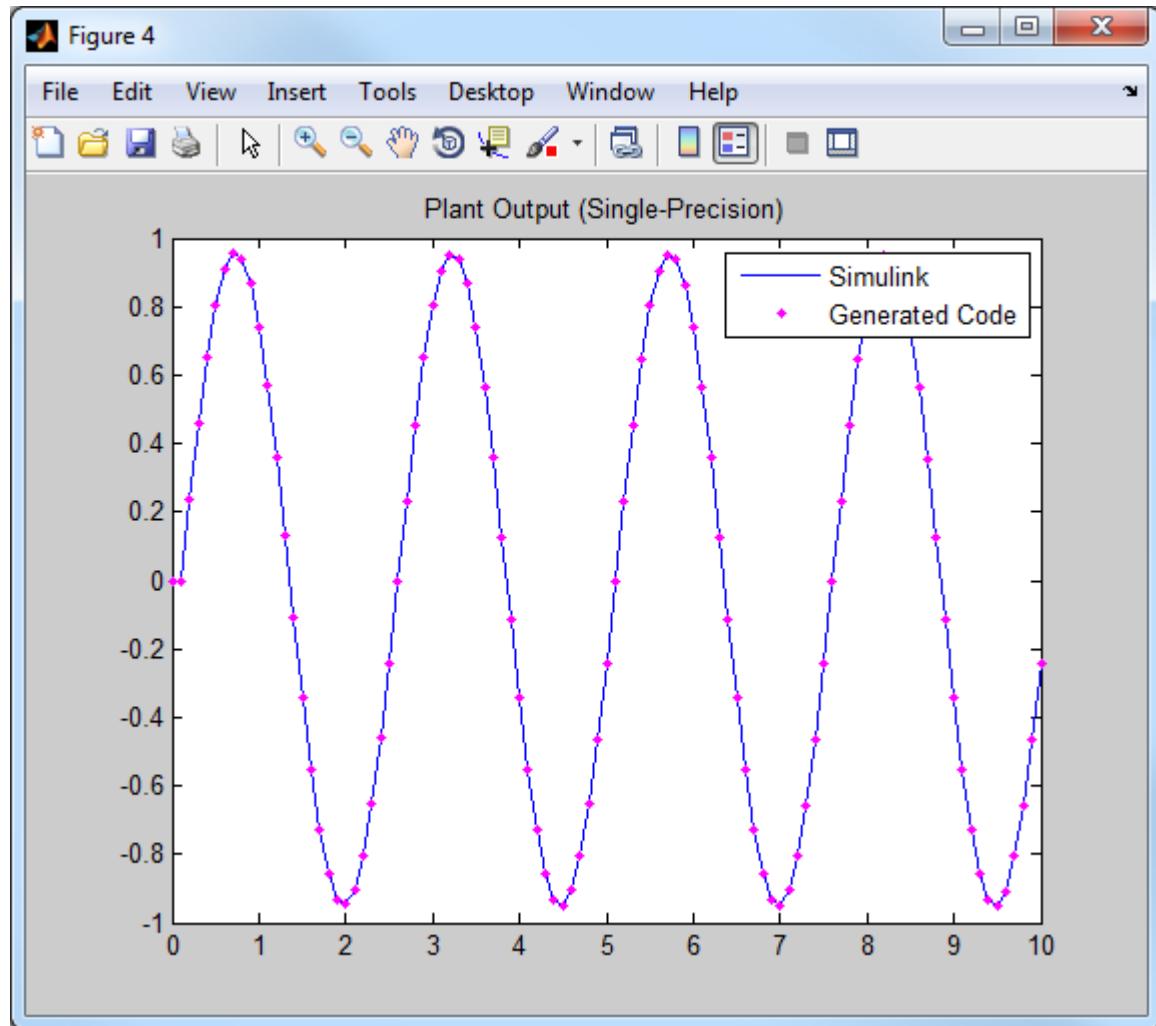
Running executable...

** starting the model **
** created mpc_rtwdemo_single.mat **
```

After the executable completes successfully (status=0), a data file named "mpc_rtwdemo_single.mat" appears in the temporary directory.

Compare the responses from the generated code (**rt_u1** and **rt_y1**) with the responses from the previous simulation in Simulink (**u1** and **y1**).





They are numerically equal.

Close the Simulink model.

```
bdclose(mdl1);  
bdclose(mdl2);
```

```
cd(cwd)
```

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5

Simulation and Structured Text Generation Using PLC Coder

This example shows how to simulate and generate Structured Text for an MPC Controller block using PLC Coder software. The generated code uses single-precision.

Required Products

To run this example, Simulink® and Simulink® PLC Coder™ are required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end
if ~mpcchecktoolboxinstalled( plccoder )
    disp( Simulink(R) PLC Coder(TM) is required to run this example. );
    return
end
```

Simulink(R) PLC Coder(TM) is required to run this example.

Setup Environment

You must have write-permission to generate the relevant files and the executable. So, before starting simulation and code generation, change the current directory to a temporary directory.

```
cwd = pwd;
tmpdir = tempname;
mkdir(tmpdir);
cd(tmpdir);
```

Define Plant Model and MPC Controller

Define a SISO plant.

```
plant = ss(tf([3 1],[1 0.6 1]));
```

Define the MPC controller for the plant.

```
Ts = 0.1; %Sampling time
p = 10; %Prediction horizon
m = 2; %Control horizon
Weights = struct( MV ,0, MVRate ,0.01, OV ,1); % Weights
```

```
MV = struct( Min ,-Inf, Max ,Inf, RateMin ,-100, RateMax ,100); % Input constraints  
OV = struct( Min ,-2, Max ,2); % Output constraints  
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Simulate and Generate Structured Text

Open the Simulink model.

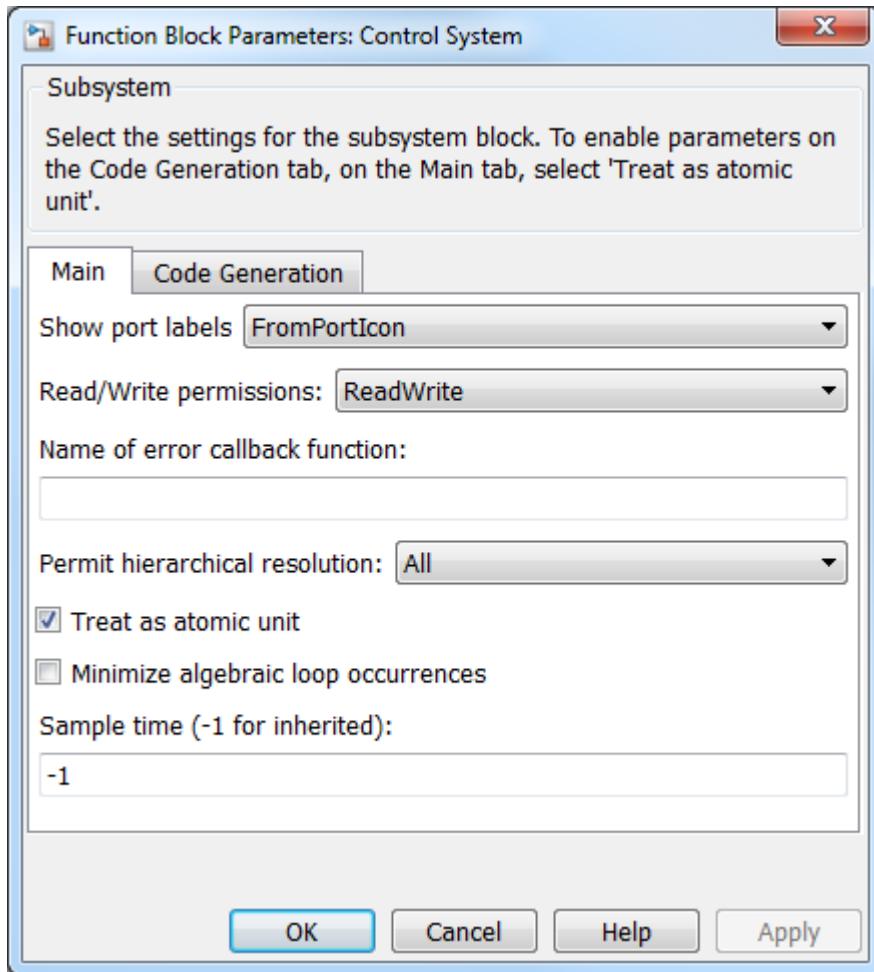
```
mdl = mpc_plcdemo ;  
open_system(mdl);
```

To generate structured text for the MPC Controller block, complete the following two steps:

- Configure the MPC block to use single precision. Select "single" in the "Output data type" combo box in the MPC block dialog.

```
open_system([mdl /Control System/MPC Controller ]);
```

- Put MPC block inside a subsystem block and treat the subsystem block as an atomic unit. Select the "Treat as atomic unit" checkbox in the subsystem block dialog.



Simulate the model in Simulink.

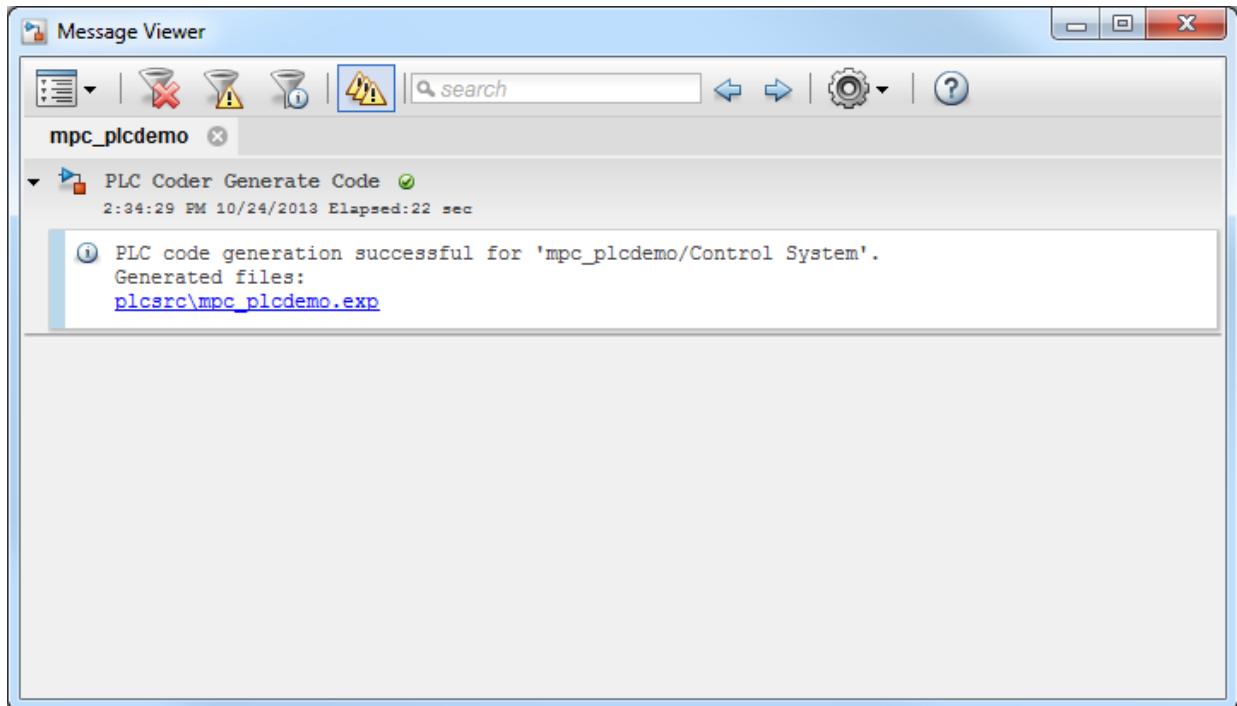
```
close_system([mdl '/Control System/MPC Controller ']);
open_system([mdl '/Outputs//References ']);
open_system([mdl '/Inputs ']);
sim(mdl);
```

To generate code with the PLC Coder, use the `plcgeneratecode` command.

```
disp( Generating PLC structure text... Please wait until it finishes. );
```

```
plcgeneratecode([mdl /Control System ]);
```

The Message Viewer dialog box shows that PLC code generation was successful.



Close the Simulink model.

```
bdclose(mdl);  
cd(cwd)
```

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5

Generate Code To Compute Optimal MPC Moves in MATLAB

This example shows how to use the `mpcmoveCodeGeneration` command to generate C code to compute optimal MPC control moves for real-time applications.

Plant Model

The plant is a single-input, single-output, stable, 2nd order linear plant.

```
plant = tf(5,[1 0.8 3]);
```

Convert the plant to discrete-time, state-space form, and specify a zero initial states vector.

```
Ts = 1;
plant = ss(c2d(plant,Ts));
x0 = zeros(size(plant.B,1),1);
```

MPC Controller Design

Create an MPC controller with default horizons.

```
mpcobj = mpc(plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming defau
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify controller tuning weights.

```
mpcobj.Weights.MV = 0;
mpcobj.Weights.MVrate = 0.5;
mpcobj.Weights.OV = 1;
```

Specify initial constraints on the manipulated variable and plant output. These constraints will be updated at run-time.

```
mpcobj.MV.Min = -1;
mpcobj.MV.Max = 1;
mpcobj.OV.Min = -1;
mpcobj.OV.Max = 1;
```

Simulating Online Constraint Changes with `mpcmove` Command

In the closed-loop simulation, constraints are updated and fed into the `mpcmove` command at each control interval.

```
yMPCMOVE = [];
uMPCMOVE = [];
```

Set the simulation time.

```
Tsim = 20;
```

Initialize the online constraint data.

```
MVMinData = -0.2-[1 0.95 0.9 0.85 0.8 0.75 0.7 0.65 0.6 0.55 0.5 ...
    0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1];
MVMaxData = 0.2+[1 0.95 0.9 0.85 0.8 0.75 0.7 0.65 0.6 0.55 0.5 ...
    0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1];
OVMinData = -0.2-[1 0.95 0.9 0.85 0.8 0.75 0.7 0.65 0.6 0.55 0.5 ...
    0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1];
OVMaxData = 0.2+[1 0.95 0.9 0.85 0.8 0.75 0.7 0.65 0.6 0.55 0.5 ...
    0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1];
```

Initialize plant states.

```
x = x0;
```

Initialize MPC states.

```
xmpc = mpcstate(mpcobj);
```

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each

Run a closed-loop simulation by calling `mpcmove` in a loop.

```
options = mpcmoveopt;
for ct = 1:round(Tsim/Ts)+1
    % Update and store plant output.
    y = plant.C*x;
    yMPCMOVE = [yMPCMOVE y];
    % Update constraints.
    options.MVMin = MVMinData(ct);
    options.MVMax = MVMaxData(ct);
```

```
options.OutputMin = OVMInData(ct);
options.OutputMax = OVMMaxData(ct);
% Compute control actions.
u = mpcmove(mpcobj,xmpc,y,1,[],options);
% Update and store plant state.
x = plant.A*x + plant.B*u;
uMPCMOVE = [uMPCMOVE u];
end
```

Validate Simulation Results with `mpcmoveCodeGeneration` Command

To prepare for generating code that computes optimal control moves from MATLAB, it is recommended to reproduce the same control results with the `mpcmoveCodeGeneration` command before using the `codegen` command from the MATLAB Coder product.

```
yCodeGen = [];
uCodeGen = [];
```

Initialize plant states.

```
x = x0;
```

Use `getCodeGenerationData` to create data structures to use with `mpcmoveCodeGeneration`.

```
[coredata, statedata, onlinedata] = getCodeGenerationData(mpcobj);
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

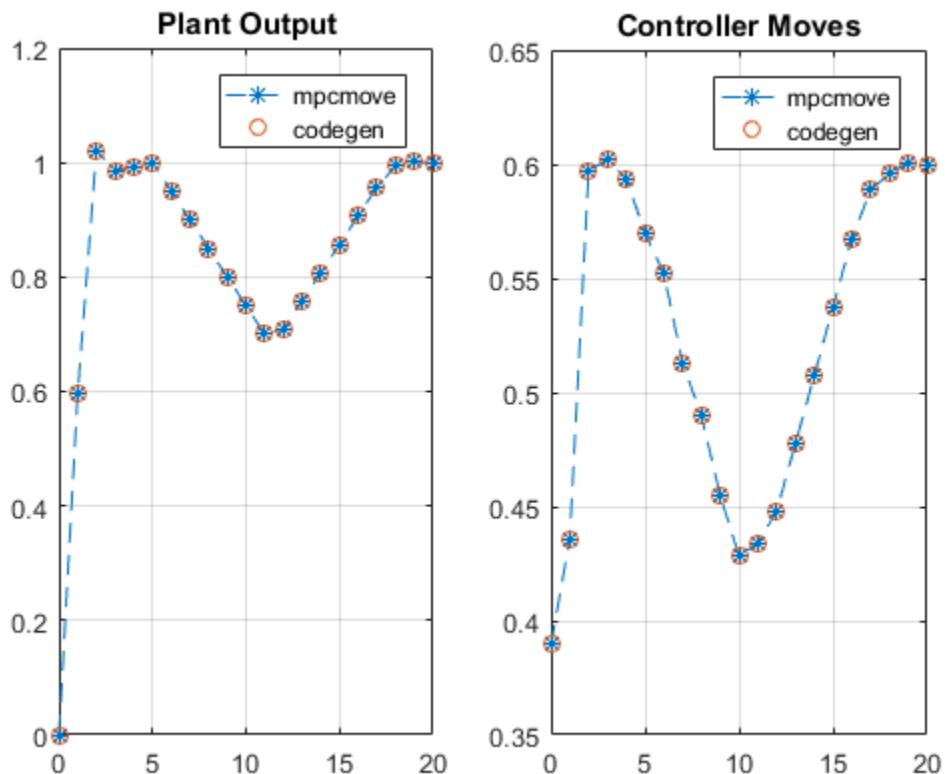
Run a closed-loop simulation by calling `mpcmoveCodeGeneration` in a loop.

```
for ct = 1:round(Tsim/Ts)+1
    % Update and store plant output.
    y = plant.C*x;
    yCodeGen = [yCodeGen y];
    % Update measured output in online data.
    onlinedata.signals.ym = y;
    % Update reference in online data.
    onlinedata.signals.ref = 1;
    % Update constraints in online data.
    onlinedata.limits.umin = MVMinData(ct);
    onlinedata.limits.umax = MVMaxData(ct);
    onlinedata.limits.ymin = OVMInData(ct);
```

```
onlinedata.limits.ymax = OVMaxData(ct);
% Compute control actions.
[u, statedata] = mpcmoveCodeGeneration(coredata, statedata, onlinedata);
% Update and store plant state.
x = plant.A*x + plant.B*u;
uCodeGen = [uCodeGen u];
end
```

The simulation results are identical to those using `mpcmove`.

```
t = 0:Ts:Tsim;
figure;
subplot(1,2,1)
plot(t,yMPCMOVE, --*, t,yCodeGen, o );
grid
legend( mpcmove , codegen )
title( Plant Output )
subplot(1,2,2)
plot(t,uMPCMOVE, --*, t,uCodeGen, o );
grid
legend( mpcmove , codegen )
title( Controller Moves )
```



Generating MEX Function From `mpcmoveCodeGeneration` Command

To generate C code from the `mpcmoveCodeGeneration` command, use the `codegen` command from the MATLAB Coder product. In this example, generate a MEX function `mpcmoveMEX` to reproduce the simulation results in MATLAB. You can change the code generation target to C/C++ static library, dynamic library, executable, etc. by using a different set of `coder.config` settings.

When generating C code for the `mpcmoveCodeGeneration` command:

- Since no data integrity checks are performed on the input arguments, you must make sure that all the input data has the correct types, dimensions, and values.

- You must define the first input argument, `mpcmove_struct`, as a constant when using `codegen` command.
- The second input argument, `mpcmove_state`, is updated by the command and returned as the second output. In most cases, you do not need to modify its contents and should simply pass it back to the command in the next control interval. The only exception is when custom state estimation is enabled, in which case you must provide the current state estimation with this argument.

```
if ~license( 'test' , 'MATLAB_Coder' )
    disp( 'MATLAB Coder(TM) is required to run this example.' )
    return
end
```

Generate MEX function.

```
fun = 'mpcmoveCodeGeneration';
funOutput = 'mpcmoveMEX';
Cfg = coder.config( 'mex' );
Cfg.DynamicMemoryAllocation = 'off';
codegen( '-config', Cfg, fun, '-o', funOutput, '-args', ...
    {coder.Constant(coredata), statedata, onlinedata});
```

Initialize data storage.

```
yMEX = [];
uMEX = [];
% Initialize plant states.
x = x0;
```

Use `getCodeGenerationData` to create data structures to use with `mpcmoveCodeGeneration`.

```
[coredata, statedata, onlinedata] = getCodeGenerationData(mpcoobj);

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

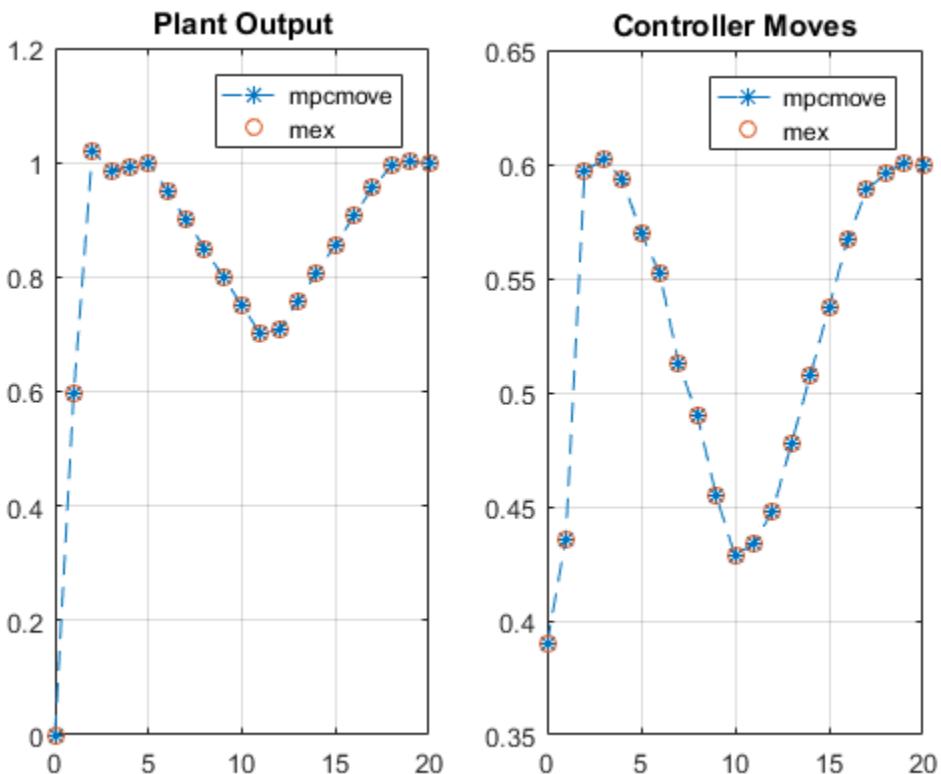
Run a closed-loop simulation by calling the generated `mpcmoveMEX` functions in a loop.

```
for ct = 1:round(Tsim/Ts)+1
    % Update and store the plant output.
    y = plant.C*x;
```

```
yMEX = [yMEX y];
% Update measured output in online data.
onlinedata.signals.ym = y;
% Update reference in online data.
onlinedata.signals.ref = 1;
% Update constraints in online data.
onlinedata.limits.umin = MVMinData(ct);
onlinedata.limits.umax = MVMaxData(ct);
onlinedata.limits.ymin = OVMinData(ct);
onlinedata.limits.ymax = OVMaxData(ct);
% Compute control actions.
[u, statedata] = mpmoveMEX(coredata, statedata, onlinedata);
% Update and store the plant state.
x = plant.A*x + plant.B*u;
uMEX = [uMEX u];
end
```

The simulation results are identical to those using `mpcmove`.

```
figure;
subplot(1,2,1)
plot(t,yMPCMOVE, --*, t,yMEX, o );
grid
legend( mpmove , mex )
title( Plant Output )
subplot(1,2,2)
plot(t,uMPCMOVE, --*, t,uMEX, o );
grid
legend( mpmove , mex )
title( Controller Moves )
```



See Also

[getCodeGenData](#) | [mpcmoveCodeGeneration](#)

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5

Setting Targets for Manipulated Variables

This example shows how to design a model predictive controller for a plant with two inputs and one output with target set-point for a manipulated variable.

Define Plant Model

The linear plant model has two inputs and two outputs.

```
N1 = [3 1];
D1 = [1 2*.3 1];
N2 = [2 1];
D2 = [1 2*.5 1];
plant = ss(tf({N1,N2},{D1,D2}));
A = plant.a;
B = plant.b;
C = plant.c;
D = plant.d;
x0 = [0 0 0 0] ;
```

Design MPC Controller

Create MPC controller.

```
Ts = 0.4; % Sampling time
mpcobj = mpc(plant,Ts,20,5);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify weights.

```
mpcobj.weights.manipulated = [0.3 0]; % weight difference MV#1 - Target#1
mpcobj.weights.manipulatedrate = [0 0];
mpcobj.weights.output = 1;
```

Define input specifications.

```
mpcobj.MV = struct( RateMin ,{-0.5;-0.5}, RateMax ,{0.5;0.5});
```

Specify target set-point u=2 for the first manipulated variable.

```
mpcobj.MV(1).Target=2;
```

Simulation Using Simulink

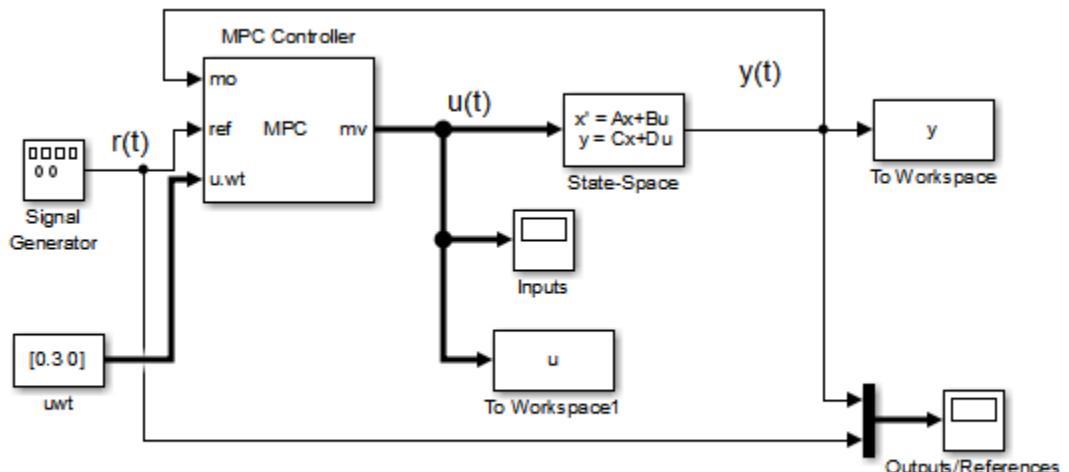
To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled( 'simulink' )
    disp( 'Simulink(R) is required to run this example.' )
    return
end
```

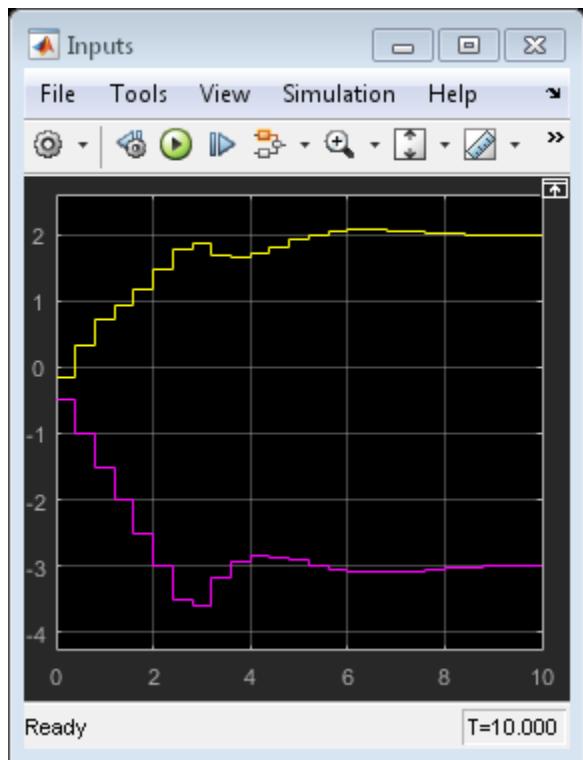
Simulate.

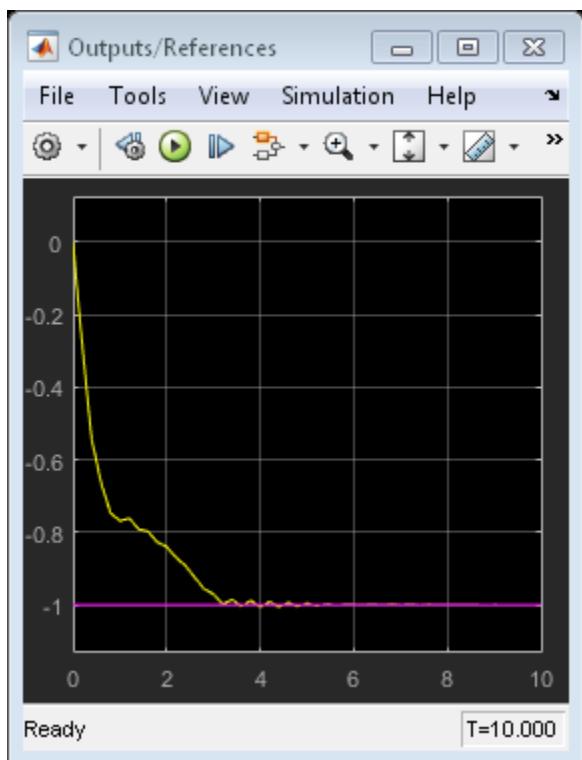
```
mdl = 'mpc_utarget';
open_system(mdl); % Open Simulink(R) Model
sim(mdl); % Start Simulation

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```



Copyright 1990-2014 The MathWorks, Inc.





```
bdclose(mdl)
```

Specifying Alternative Cost Function with Off-Diagonal Weight Matrices

This example shows how to use non-diagonal weight matrices in a model predictive controller.

Define Plant Model and MPC Controller

The linear plant model has two inputs and two outputs.

```
plant = ss(tf({1,1;1,2}, {[1 .5 1], [.7 .5 1]; [1 .4 2], [1 2]}));
[A,B,C,D] = ssdata(plant);
Ts = 0.1; % sampling time
plant = c2d(plant,Ts); % convert to discrete time
```

Create MPC controller.

```
p=20; % prediction horizon
m=2; % control horizon
mpcobj = mpc(plant,Ts,p,m);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default: 0
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default: 0
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default: 0
```

Define constraints on the manipulated variable.

```
mpcobj.MV = struct( Min ,{-3;-2}, Max ,{3;2}, RateMin ,{-100;-100}, RateMax ,{100;100} );
```

Define non-diagonal output weight. Note that it is specified inside a cell array.

```
OW = [1 -1]*[1 -1];
% Non-diagonal output weight, corresponding to ((y1-r1)-(y2-r2))^2
mpcobj.Weights.OutputVariables = {OW};
% Non-diagonal input weight, corresponding to (u1-u2)^2
mpcobj.Weights.ManipulatedVariables = {0.5*OW};
```

Simulate Using SIM Command

Specify simulation options.

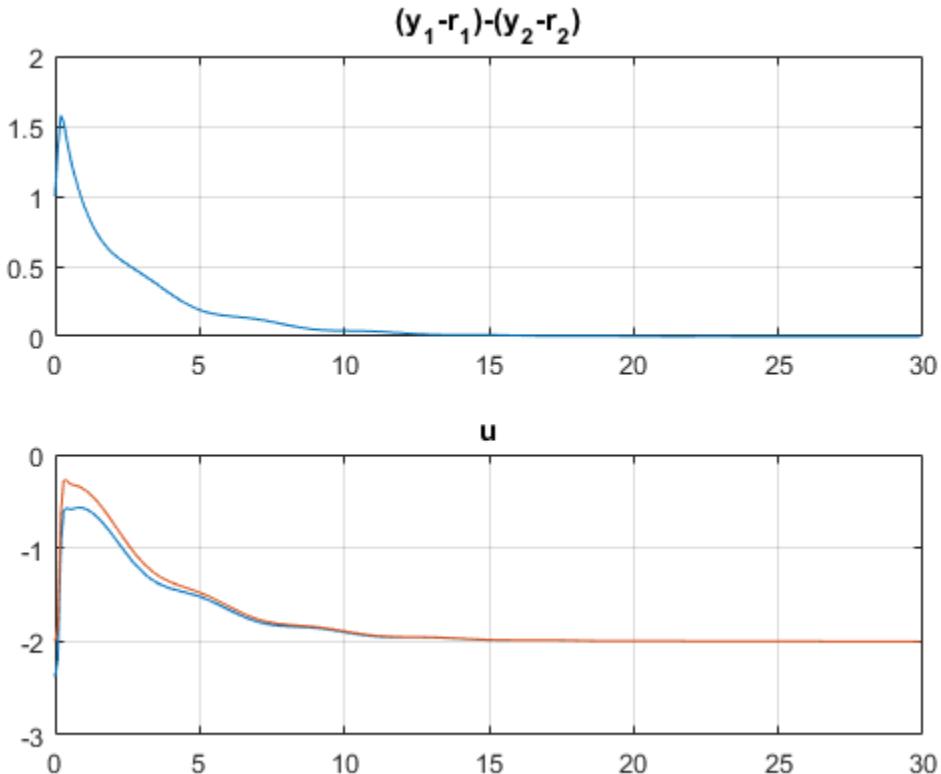
```
Tstop = 30; % simulation time
Tf = round(Tstop/Ts); % number of simulation steps
```

```
r = ones(Tf,1)*[1 2]; % reference trajectory
```

Run the closed-loop simulation and plot results.

```
[y,t,u] = sim(mpcobj,Tf,r);
subplot(211)
plot(t,y(:,1)-r(1,1)-y(:,2)+r(1,2));grid
title( (y_1-r_1)-(y_2-r_2) );
subplot(212)
plot(t,u);grid
title( u );

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```



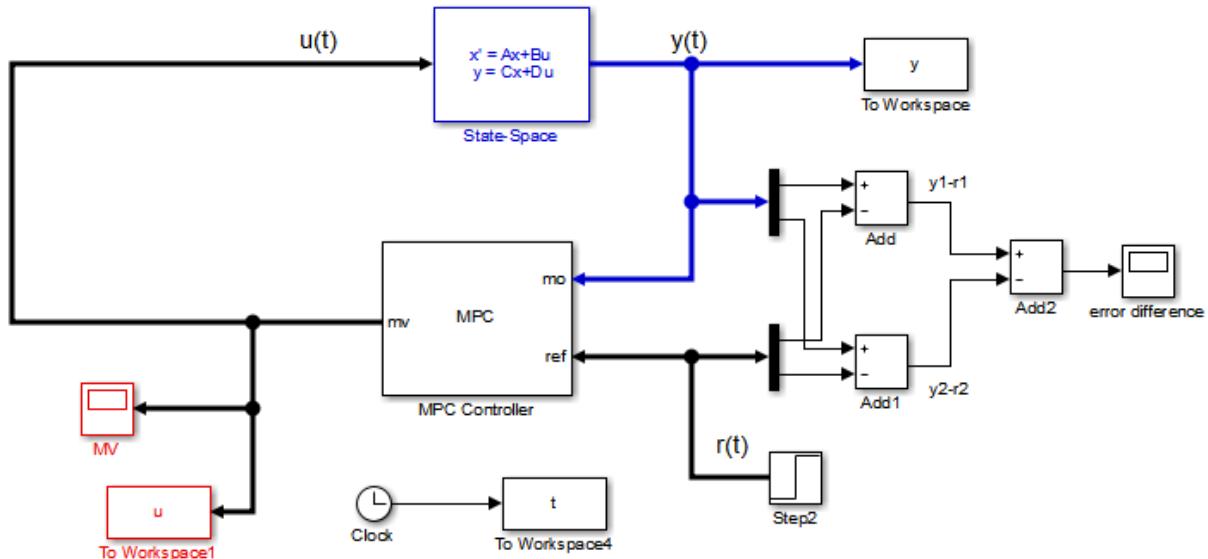
Simulate Using Simulink

To run this example, Simulink® is required.

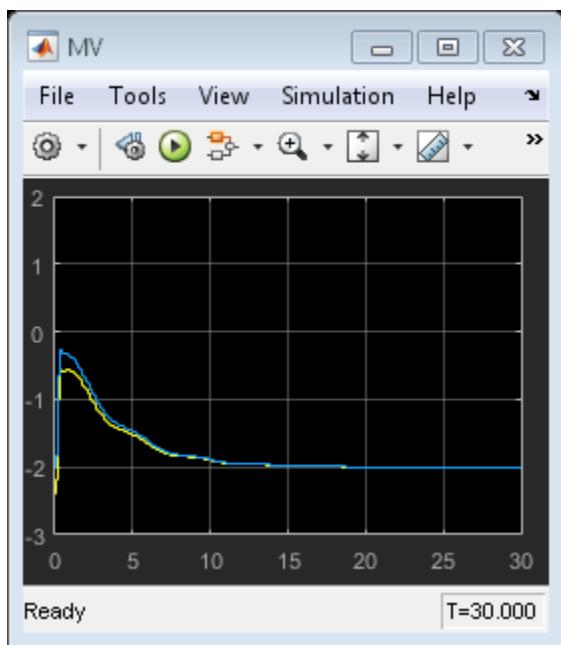
```
if ~mpcchecktoolboxinstalled( simulink )
    disp( 'Simulink(R) is required to run this part of the example.' )
    return
end
```

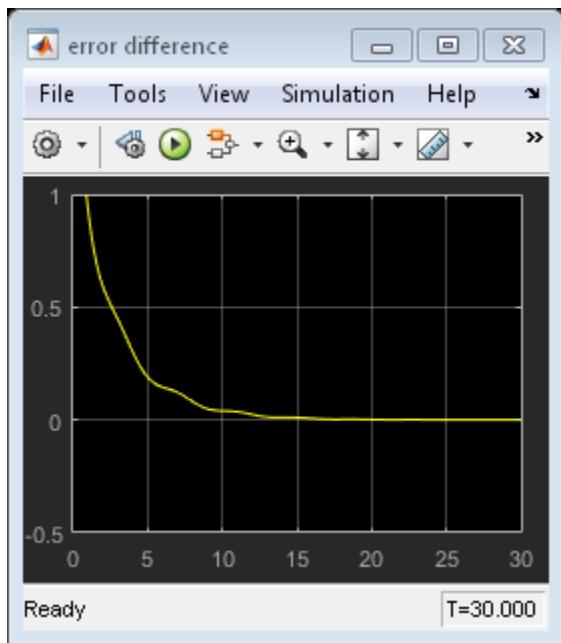
Now simulate closed-loop MPC in Simulink®.

```
mdl = mpc_weightsdemo ;
open_system(mdl);
sim(mdl)
```



Copyright 1990-2012 The MathWorks, Inc.





```
bdclose(md1);
```

Review Model Predictive Controller for Stability and Robustness Issues

This example shows how to use the `review` command to detect potential issues with a model predictive controller design.

The Fuel Gas Blending Process

The example application is a fuel gas blending process. The objective is to blend six gases to obtain a fuel gas, which is then burned to provide process heating. The fuel gas must satisfy three quality standards in order for it to burn reliably and with the expected heat output. The fuel gas header pressure must also be controlled. Thus, there are four controlled output variables. The manipulated variables are the six feed gas flow rates.

Inputs:

1. Natural Gas (NG)
2. Reformed Gas (RG)
3. Hydrogen (H₂)
4. Nitrogen (N₂)
5. Tail Gas 1 (T1)
6. Tail Gas 2 (T2)

Outputs:

1. High Heating Value (HHV)
2. Wobbe Index (WI)
3. Flame Speed Index (FSI)
4. Header Pressure (P)

The fuel gas blending process was studied by Muller et al.: "Modeling, validation, and control of an industrial fuel gas blending system", C.J. Muller, I.K. Craig, N.L. Ricker, J. of Process Control, in press, 2011.

Linear Plant Model

Use the following linear plant model as the prediction model for the controller. This state-space model, applicable at a typical steady-state operating point, uses the time unit of hours.

```
a = diag([-28.6120, -28.6822, -28.5134, -0.0281, -23.2191, -23.4266, ...])
```

```
-22.9377, - 0.0101, -26.4877, -26.7950, -27.2210, -0.0083, ...
-23.0890, -23.0062, -22.9349, -0.0115, -25.8581, -25.6939, ...
-27.0793, -0.0117, -22.8975, -22.8233, -21.1142, -0.0065]);
b = zeros(24,6);
b( 1: 4,1) = [4, 4, 8, 32] ;
b( 5: 8,2) = [2, 2, 4, 32] ;
b( 9:12,3) = [2, 2, 4, 32] ;
b(13:16,4) = [4, 4, 8, 32] ;
b(17:20,5) = [2, 2, 4, 32] ;
b(21:24,6) = [1, 2, 1, 32] ;
c = [diag([-6.1510, 7.6785, -5.9312, 34.2689]), ...
      diag([-2.2158, -3.1204, 2.6220, 35.3561]), ...
      diag([-2.5223, 1.1480, 7.8136, 35.0376]), ...
      diag([-3.3187, -7.6067, -6.2755, 34.8720]), ...
      diag([-1.6583, -2.0249, 2.5584, 34.7881]), ...
      diag([-1.6807, -1.2217, 1.0492, 35.0297])];
d = zeros(4,6);
Plant = ss(a, b, c, d);
```

By default, all the plant inputs are manipulated variables.

```
Plant.InputName = { NG , RG , H2 , N2 , T1 , T2 };
```

By default, all the plant outputs are measured outputs.

```
Plant.OutputName = { HHV , WI , FSI , P };
```

Transport delay is added to plant outputs to reflect the delay in the sensors.

```
Plant.OutputDelay = [0.00556 0.0167 0.00556 0];
```

Initial Controller Design

Construct an initial model predictive controller based on design requirements.

Specify sampling time, horizons and steady-state values.

The sampling time is that of the sensors (20 seconds). The prediction horizon is approximately equal to the plant settling time (39 intervals). The control horizon uses four blocked moves that have lengths of 2, 6, 12 and 19 intervals respectively. The nominal operating conditions are non-zero. The output measurement noise is white noise with magnitude of 0.001.

```
MPC_verbosity = mpcverbosity( off ); % Disable MPC message displaying at command line
```

```

Ts = 20/3600; % Time units are hours.
Obj = mpc(Plant, Ts, 39, [2, 6, 12, 19]);
Obj.Model.Noise = ss(0.001*eye(4));
Obj.Model.Nominal.Y = [16.5, 25, 43.8, 2100];
Obj.Model.Nominal.U = [1.4170, 0, 2, 0, 0, 26.5829];

```

Specify lower and upper bounds on manipulated variables.

Since all the manipulated variables are flow rates of gas streams, their lower bounds are zero. All the MV constraints are hard (MinECR and MaxECR = 0) by default.

```

MVmin = zeros(1,6);
MVmax = [15, 20, 5, 5, 30, 30];
for i = 1:6
    Obj.MV(i).Min = MVmin(i);
    Obj.MV(i).Max = MVmax(i);
end

```

Specify lower and upper bounds on manipulated variable increments.

The bounds are set large enough to allow full range of movement in one interval. All the MV rate constraints are hard (RateMinECR and RateMaxECR = 0) by default.

```

for i = 1:6
    Obj.MV(i).RateMin = -MVmax(i);
    Obj.MV(i).RateMax = MVmax(i);
end

```

Specify lower and upper bounds on plant outputs.

All the OV constraints are soft (MinECR and MaxECR = 0) by default.

```

OVmin = [16.5, 25, 39, 2000];
OVmax = [18.0, 27, 46, 2200];
for i = 1:4
    Obj.OV(i).Min = OVmin(i);
    Obj.OV(i).Max = OVmax(i);
end

```

Specify weights on manipulated variables.

MV weights are specified based on the known costs of each feed stream. This tells MPC controller how to move the six manipulated variables in order to minimize the cost of the

blended fuel gas. The weights are normalized so the maximum weight is approximately 1.0.

```
Obj.Weights.MV = [54.9, 20.5, 0, 5.73, 0, 0]/55;
```

Specify weights on manipulated variable increments.

They are small relative to the maximum MV weight so the MVs are free to vary.

```
Obj.Weights.MVrate = 0.1*ones(1,6);
```

Specify weights on plant outputs.

The OV weights penalize deviations from specified setpoints and would normally be "large" relative to the other weights. Let us first consider the default values, which equal the maximum MV weight specified above.

```
Obj.Weights.OV = [1, 1, 1, 1];
```

Using the review Command to Improve the Initial Design

Review the initial controller design.

```
review(Obj)
```

The screenshot shows a Microsoft Internet Explorer window with the title bar "Web Browser - Review MPC Object "Obj"" and a tab labeled "Review MPC Object "Obj"".

Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Warning
Hard MV Constraints	Warning
Other Hard Constraints	Pass
Soft Constraints	Fail
Memory Size for MPC Data	Pass

The summary table shown above lists three warnings and one error. Let's consider these in turn. Click **QP Hessian Matrix Validity** and scroll down to display the warning. It indicates that the plant signal magnitudes differ significantly. Specifically, the pressure response is much larger than the others.

Scale Factors

Scaling converts the relationship between output variables and manipulated variables to dimensionless form. Scale factor specifications can improve QP numerical accuracy. They also make it easier to specify tuning weight magnitudes.

In order for the outputs to be controllable, each must respond to at least one manipulated variable within the prediction horizon. If the plant is well scaled, the maximum absolute value of such responses should be of order unity.

Outputs whose maximum absolute scaled responses are outside the range [0.1,10] appear below. The table shows the maximum absolute response of each such OV with respect to each MV.

	NG	RG	H2	N2	T1	T2
P	236.876	244.868	242.709	241.478	240.892	242.702

Warning: at least one output variable response indicates poor scaling. Consider adjusting MV and OV ScaleFactors.

Examination of the specified OV bounds shows that the spans are quite different, and the pressure span is two orders of magnitude larger than the others. It is good practice to specify MPC scale factors to account for the expected differences in signal magnitudes. We are already weighting MVs based on relative cost, so we focus on the OVs only.

Calculate OV spans

```
OVspan = OVmax - OVmin;
%
% Use these as the specified scale factors
for i = 1:4
    Obj.OV(i).ScaleFactor = OVspan(i);
end
% Use review to verify that the scale factor warning has disappeared.
review(Obj);
%
% <<.../reviewDemo03.png>>
```

The next warning indicates that the controller does not drive the OVs to their targets at steady state. Click **Closed-Loop Steady-State Gains** to see a list of the non-zero gains.

Closed-Loop Steady-State Gains

`cloffset` is used to determine whether the controller forces all controlled output variables to their targets at steady state, in the absence of constraints.

The command calculates the impact of a sustained disturbance on each measured output variable (OV) in terms of an input/output gain. If a gain is zero, the controller eliminates steady-state tracking error for that disturbance-to-output mapping.

The gains with magnitudes exceeding 1e-05 are as follows:

Disturbed OV	Affected OV	Gain
HHV	HHV	0.0860281
WI	HHV	-0.0344992
FSI	HHV	0.0665757
HHV	WI	-0.036145
WI	WI	0.014495
FSI	WI	-0.027972
HHV	FSI	0.279361
WI	FSI	-0.11203
FSI	FSI	0.216193
HHV	P	0.0468767
WI	P	-0.0187986
FSI	P	0.036277

Warning: your design allows non-zero steady-state tracking errors in at least one controlled output. If this was not your intent, possible causes are as follows:

- Zero penalty weight on a plant output. Check the `Weights.OV` property.
- Non-zero penalty weight on a manipulated variable. Check the `Weights.MV` property.
- State estimator that does not include integration of output tracking error. The default estimator includes integration. If you have modified or replaced it, review your estimator design.

The first entry in the list shows that adding a sustained disturbance of unit magnitude to the HHV output would cause the HHV to deviate 0.0860 units from its steady-state target, assuming no constraints are active. The second entry shows that a unit disturbance in WI would cause a steady-state deviation ("offset") of -0.0345 in HHV, etc.

Since there are six MVs and only four OVs, excess degrees of freedom are available and you might be surprised to see non-zero steady-state offsets. The non-zero MV weights we have specified in order to drive the plant toward the most economical operating condition are causing this.

Non-zero steady-state offsets are often undesirable but are acceptable in this application because:

- # The primary objective is to minimize the blend cost. The gas quality (HHV, etc.) can vary freely within the specified OV limits.
- # The small offset gain magnitudes indicate that the impact of disturbances would be small.
- # The OV limits are soft constraints. Small, short-term violations are acceptable.

View the second warning by clicking **Hard MV Constraints**, which indicates a potential hard-constraint conflict.

Web Browser - Review MPC Object "Obj"

Review MPC Object "Obj" +

Hard MV Constraints

The controller should always satisfy hard bounds on a manipulated variable *OR* its rate-of-change. If you specify both constraint types simultaneously, however, they might conflict during real-time use.

For example, if an event pushes an MV outside a specified hard bound and the hard MV rate bounds are too small, the resulting QP will be *infeasible*.

Avoid such conflicts by specifying hard MV bounds *OR* hard MV rate bounds, but not both. Or if you want to specify both, soften the lower-priority constraint by setting its ECR to a value greater than zero.

Warning: your constraint definitions may conflict. The following table lists potential conflicts for each MV. The tabular entries show the location of each conflict in the prediction horizon and the type of conflict.

MV name	Horizon k	Conflict Type
NG	1	Min & RateMax
NG	1	Max & RateMin
RG	1	Min & RateMax
RG	1	Max & RateMin
H2	1	Min & RateMax
H2	1	Max & RateMin
N2	1	Min & RateMax
N2	1	Max & RateMin
T1	1	Min & RateMax
T1	1	Max & RateMin
T2	1	Min & RateMax
T2	1	Max & RateMin

[Return to list of tests](#)

If an external event causes the NG to go far below its specified minimum, the constraint on its rate of increase might make it impossible to return the NG within bounds in one interval. In other words, when you specify both MV.Min and MV.RateMax, the controller would not be able to find an optimal solution if the most recent MV value is less than (MV.Min - MV.RateMax). Similarly, there is a potential conflict when you specify both MV.Max and MV.RateMin.

An MV constraint conflict would be extremely unlikely in the gas blending application, but it's good practice to eliminate the possibility by softening one of the two constraints. Since the MV minimum and maximum values are physical limits and the increment bounds are not, we soften the increment bounds as follows:

```
for i = 1:6
    Obj.MV(i).RateMinECR = 0.1;
    Obj.MV(i).RateMaxECR = 0.1;
end
```

Review the new controller design.

```
review(Obj)
```

The screenshot shows a Microsoft Internet Explorer window with the title bar "Web Browser - Review MPC Object "Obj"" and the tab "Review MPC Object "Obj"".

Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Warning
Hard MV Constraints	Pass
Other Hard Constraints	Pass
Soft Constraints	Fail
Memory Size for MPC Data	Pass

The MV constraint conflict warning has disappeared. Now click **Soft Constraints** to view the error message.

Impact of delays

Delays can make it impossible to satisfy output constraints. The presence of unattainable constraints usually degrades performance. Let j be the location (within the prediction horizon) of the first finite constraint value (Min or Max) for $OV(i)$. If all delays for $OV(i)$ exceed j , the constraint is unattainable.

The following table lists each output constraint that is impossible to satisfy. The first column is the location (within the prediction horizon) of the first finite constraint value. The second column is the minimum delay for that output variable.

Constraint	Begins	Delay
WI.Min	1	3
WI.Max	1	3

Error: at least one output variable constraint is impossible to satisfy.

We see that the delay in the WI output makes it impossible to satisfy bounds on that variable until at least three control intervals have elapsed. The WI bounds are soft but it is poor practice to include unattainable constraints in a design. We therefore modify the WI bound specifications such that it is unconstrained until the 4th prediction horizon step.

```
Obj.OV(2).Min = [-Inf(1,3), OVmin(2)];  
Obj.OV(2).Max = [ Inf(1,3), OVmax(2)];
```

```
% Ee-issuing the review command to verifies that this eliminates the  
% error message (see the next step).
```

Diagnosing the Impact of Zero Output Weights

Given that the design requirements allow the OVs to vary freely within their limits, consider zeroing their penalty weights:

```
Obj.Weights.OV = zeros(1,4);
```

Review the impact of this design change.

```
review(Obj)
```

The screenshot shows a web browser window with the title "Web Browser - Review MPC Object "Obj"" and a tab labeled "Review MPC Object "Obj"".

Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Warning
Hard MV Constraints	Pass
Other Hard Constraints	Pass
Soft Constraints	Pass
Memory Size for MPC Data	Pass

A new warning regarding QP Hessian Matrix Validity has appeared. Click **QP Hessian Matrix Validity** warning to see the details.

The screenshot shows a Windows-style application window titled "Web Browser - Review MPC Object "Obj"". The window has a toolbar with icons for back, forward, and search. Below the toolbar, the title bar says "Review MPC Object "Obj"" and there is a "+" button. The main content area contains the following text and table:

Penalty Weights On Output Variables

Your output variable (OV) penalty weights also affect the Hessian. Non-zero values emphasize the importance of OV target tracking, making a unique QP solution more likely.

The following table lists the minimum weight for each OV along the prediction horizon.

OV	Weights.OV
HHV	0
WI	0
FSI	0
P	0

Warning: at least one OV weight is zero or very small.

The review has flagged the zero weights on all four output variables. Since the zero weights are consistent with the design requirement and the other Hessian tests indicate that the quadratic programming problem has a unique solution, this warning can be ignored.

Click **Closed-Loop Steady-State Gains** to see the second warning. It shows another consequence of setting the four OV weights to zero. When an OV is not penalized by a weight, any output disturbance added to it will be ignored, passing through with no attenuation.

Closed-Loop Steady-State Gains

`cloffset` is used to determine whether the controller forces all controlled output variables to their targets at steady state, in the absence of constraints.

The command calculates the impact of a sustained disturbance on each measured output variable (OV) in terms of an input/output gain. If a gain is zero, the controller eliminates steady-state tracking error for that disturbance-to-output mapping.

The gains with magnitudes exceeding 1e-05 are as follows:

Disturbed OV Affected OV Gain

HHV	HHV	1
WI	WI	1
FSI	FSI	1
P	P	1

Since it is a design requirement, non-zero steady-state offsets are acceptable provided that MPC is able to hold all the OVs within their specified bounds. It is therefore a good idea to examine how easily the soft OV constraints can be violated when disturbances are present.

Reviewing Soft Constraints

Click **Soft Constraints** to see a list of soft constraints -- in this case an upper and lower bound on each OV.

Soft Constraints

ECR Parameters

This test evaluates the constraint ECR parameters to help you achieve the proper balance of using hard and soft constraints. If a constraint is too soft, an unacceptable violation may occur. If it is too hard, the controller might pay it too much attention. Moreover, making a constraint harder cannot prevent a violation if the constraint is fundamentally infeasible.

You have defined 8 soft constraints. The table below lists these and shows potential violations based on specified variable bounds and other factors.

Impact Factor: the increase in the MPC cost function caused by this constraint violation relative to the average such increase. Rows are sorted in order of decreasing impact.

Sensitivity Ratio: the increase in the MPC cost function caused by this constraint violation relative to the typical cost function magnitude when there are no violations.

We consider a possible constraint violation equal to 10% of the nominal OV range. It then estimates the impact of such a violation on the MPC objective function relative to the impact of other violations. A large impact factor indicates a high-priority controller objective, and vice versa.

Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: P	20	1509	1000
Upper limit: P	20	1509	1000
Lower limit: FSI	0.7	1.849	1.225
Upper limit: FSI	0.7	1.849	1.225
Lower limit: WI	0.2	0.1509	0.1
Upper limit: WI	0.2	0.1509	0.1
Lower limit: HHV	0.15	0.08491	0.05625
Upper limit: HHV	0.15	0.08491	0.05625

A sensitivity ratio greater than $1e+08$ may degrade QP solution accuracy.

The Impact Factor column shows that using the default MinECR and MaxECR values give the pressure (P) a much higher priority than the other OVs. If we want the priorities to be more comparable, we should increase the pressure constraint ECR values and adjust the others too. For example, we consider

```
Obj.OV(1).MinECR = 0.5;
Obj.OV(1).MaxECR = 0.5;
Obj.OV(3).MinECR = 3;
Obj.OV(3).MaxECR = 3;
Obj.OV(4).MinECR = 80;
Obj.OV(4).MaxECR = 80;
```

Review the impact of this design change.

```
review(Obj)
```

Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: HHV	0.15	1.539	0.225
Upper limit: HHV	0.15	1.539	0.225
Lower limit: P	20	1.069	0.1563
Upper limit: P	20	1.069	0.1563
Lower limit: FSI	0.7	0.9311	0.1361
Upper limit: FSI	0.7	0.9311	0.1361
Lower limit: WI	0.2	0.6841	0.1
Upper limit: WI	0.2	0.6841	0.1

Notice from the Sensitivity Ratio column that all the sensitivity ratios are now less than unity. This means that the soft constraints will receive less attention than other terms in the MPC objective function, such as deviations of the MVs from their target values. Thus, it is likely that an output constraint violation would occur.

In order to give the output constraints higher priority than other MPC objectives, increase the Weights.ECR parameter from default 1e5 to a higher value to harden all the soft OV constraints.

```
Obj.Weights.ECR = 1e8;
```

Review the impact of this design change.

```
review(Obj)
```

Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: HHV	0.15	1.539	225
Upper limit: HHV	0.15	1.539	225
Lower limit: P	20	1.069	156.3
Upper limit: P	20	1.069	156.3
Lower limit: FSI	0.7	0.9311	136.1
Upper limit: FSI	0.7	0.9311	136.1
Lower limit: WI	0.2	0.6841	100
Upper limit: WI	0.2	0.6841	100

The controller is now a factor of 100 more sensitive to output constraint violations than to errors in target tracking.

Reviewing Data Memory Size

Click **Memory Size for MPC Data** to see the estimated memory size needed to store the MPC data matrices used on the hardware.

The screenshot shows a Microsoft Internet Explorer window with the title bar 'Web Browser - Review MPC Object "Obj"'. The main content area contains a heading 'Memory Size for MPC Data' and a paragraph explaining the purpose of the test. Below this is a table comparing memory requirements for different MPC configurations. At the bottom of the page is a link 'Return to list of tests'.

Type	Single Precision (kB)	Double Precision (kB)
MPC	250	500
MPC with Online Tuning	350	700

[Return to list of tests](#)

In this example, if the controller is running using single precision, it requires 250 KB of memory to store its matrices. If the controller memory size exceeds the memory available on the target system, you must redesign the controller to reduce its memory requirements. Alternatively, increase the memory available on the target system.

```
mpcverbosity(MPC_verbosity);  
[~, hWebBrowser] = web;  
close(hWebBrowser);
```

See Also

`review`

Control of an Inverted Pendulum on a Cart

This example uses a model predictive controller (MPC) to control an inverted pendulum on a cart.

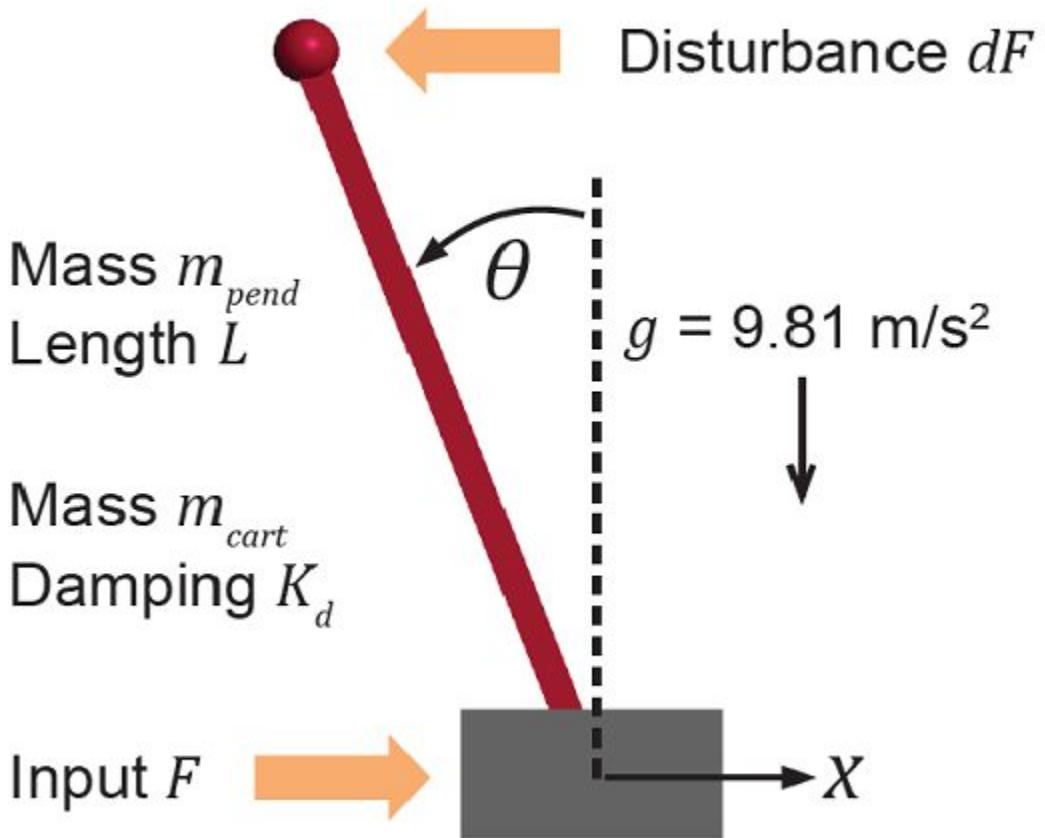
Product Requirement

This example requires Simulink® Control Design™ software to define the MPC structure by linearizing a nonlinear Simulink model.

```
if ~mpcchecktoolboxinstalled( 'slcontrol' )
    disp( 'Simulink Control Design(TM) is required to run this example.' )
    return
end
```

Pendulum/Cart Assembly

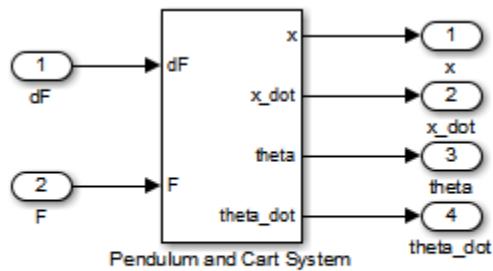
The plant for this example is the following cart/pendulum assembly, where x is the cart position and θ is the pendulum angle.



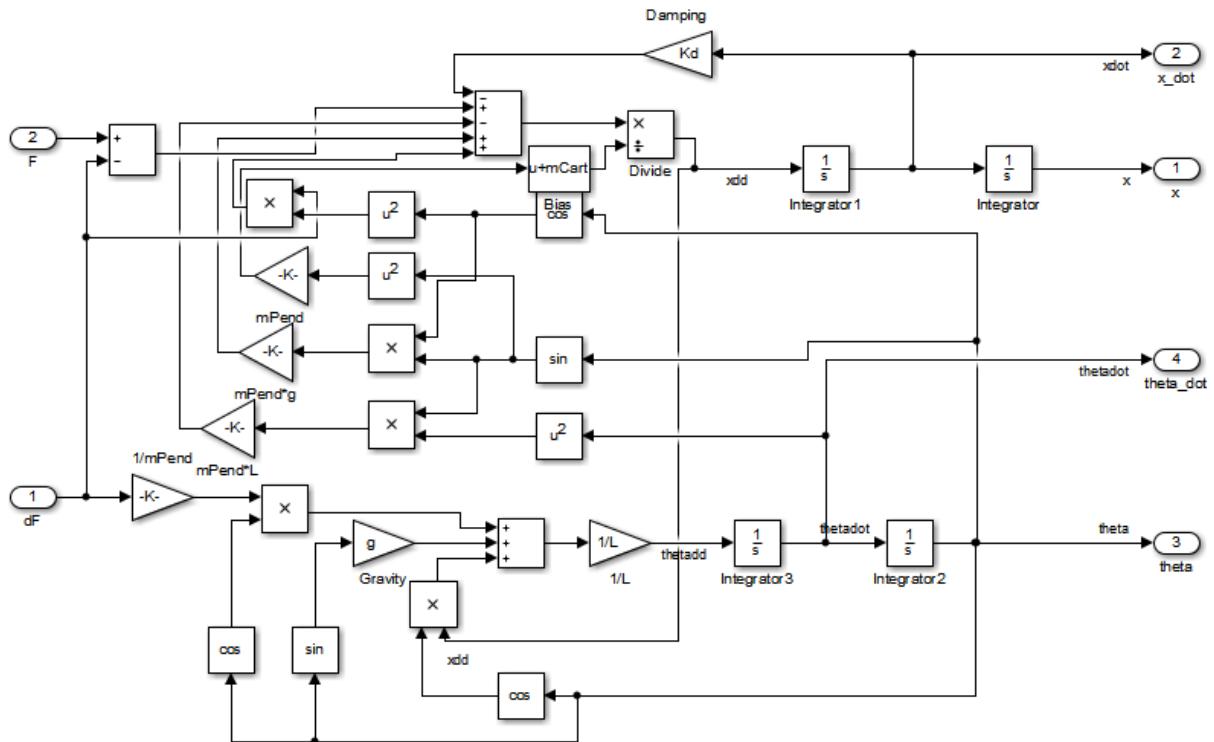
This system is controlled by exerting a variable force F on the cart. The controller needs to keep the pendulum upright while moving the cart to a new position or when the pendulum is nudged forward by an impulse disturbance dF applied at the upper end of the inverted pendulum.

This plant is modeled in Simulink with commonly used blocks.

```
mdlPlant = mpc_pendcartPlant ;
load_system(mdlPlant);
open_system([mdlPlant /Pendulum and Cart System ], force );
```



Copyright 1990-2015 The MathWorks, Inc.



Control Objectives

Assume the following initial conditions for the cart/pendulum assembly:

- The cart is stationary at $x = 0$.
- The inverted pendulum is stationary at the upright position $\theta = 0$.

The control objectives are:

- Cart can be moved to a new position between -10 and 10 with a step setpoint change.
- When tracking such a setpoint change, the rise time should be less than 4 seconds (for performance) and the overshoot should be less than 5 percent (for robustness).
- When an impulse disturbance of magnitude of 2 is applied to the pendulum, the cart should return to its original position with a maximum displacement of 1. The pendulum should also return to the upright position with a peak angle displacement of 15 degrees (0.26 radian).

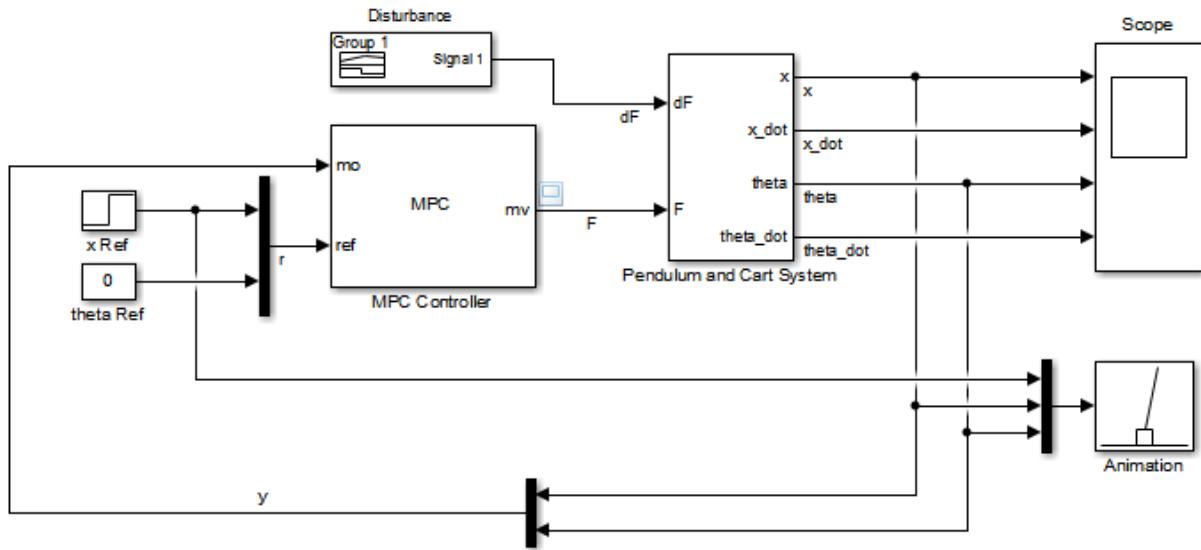
The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

Control Structure

For this example, use a single MPC controller with:

- One manipulated variable: Variable force F .
- Two measured outputs: Cart position x and pendulum angle θ .
- One unmeasured disturbance: Impulse disturbance dF .

```
mdlMPC = mpc_pendcartImplicitMPC ;
open_system(mdlMPC);
```



Copyright 1990-2015 The MathWorks, Inc.

Although cart velocity x_dot and pendulum angular velocity $theta_dot$ are available from the plant model, to make the design case more realistic, they are excluded as MPC measurements.

While the cart position setpoint varies (step input), the pendulum angle setpoint is constant (0 = upright position).

Linear Plant Model

Since the MPC controller requires a linear time-invariant (LTI) plant model for prediction, linearize the Simulink plant model at the initial operating point.

Specify linearization input and output points.

```
io(1) = linio([mdlPlant '/dF'],1, openinput );
io(2) = linio([mdlPlant '/F'],1, openinput );
io(3) = linio([mdlPlant '/Pendulum and Cart System'],1, openoutput );
io(4) = linio([mdlPlant '/Pendulum and Cart System'],3, openoutput );
```

Create operating point specifications for the plant initial conditions.

```
opspec = operspec(mdlPlant);
```

The first state is cart position x , which has a known initial state of 0.

```
opspec.States(1).Known = true;
opspec.States(1).x = 0;
```

The third state is pendulum angle θ , which has a known initial state of 0.

```
opspec.States(3).Known = true;
opspec.States(3).x = 0;
```

Compute operating point using these specifications.

```
options = findopOptions( DisplayReport ,false);
op = findop(mdlPlant,opspec,options);
```

Obtain the linear plant model at the specified operating point.

```
plant = linearize(mdlPlant,op,io);
plant.InputName = { dF ; F };
plant.OutputName = { x ; theta };
```

Examine the poles of the linearized plant.

```
pole(plant)
```

```
ans =
```

```
0
-11.9115
-3.2138
5.1253
```

The plant has an integrator and an unstable pole.

```
bdclose(mdlPlant);
```

MPC Design

The plant has two inputs, dF and F , and two outputs, x and θ . In this example, dF is specified as an unmeasured disturbance used by the MPC controller for better disturbance rejection. Set the plant signal types.

```
plant = setmpcsignals(plant, ud ,1, mv ,2);
```

To control an unstable plant, the controller sample time cannot be too large (poor disturbance rejection) or too small (excessive computation load). Similarly, the prediction horizon cannot be too long (the plant unstable mode would dominate) or too short (constraint violations would be unforeseen). Use the following parameters for this example:

```
Ts = 0.01;
PredictionHorizon = 50;
ControlHorizon = 5;
mpcobj = mpc(plant,Ts,PredictionHorizon,ControlHorizon);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 1
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

There is a limitation on how much force can be applied to the cart, which is specified as hard constraints on manipulated variable F .

```
mpcobj.MV.Min = -200;
mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by 100.

```
mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from 0.1 to 1.

```
mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position x and the second weight is associated with angle θ .

```
mpcobj.Weights.OV = [1.2 1];
```

To achieve more aggressive disturbance rejection, increase the state estimator gain by multiplying the default disturbance model gains by a factor of 10.

Update the input disturbance model.

```
disturbance_model = getindist(mpcobj);
setindist(mpcobj, model ,disturbance_model*10);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #1 is integrated white noise.
    Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Update the output disturbance model.

```
disturbance_model = getoutdist(mpcobj);
setoutdist(mpcobj, model ,disturbance_model*10);

-->Converting model to discrete time.
    Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

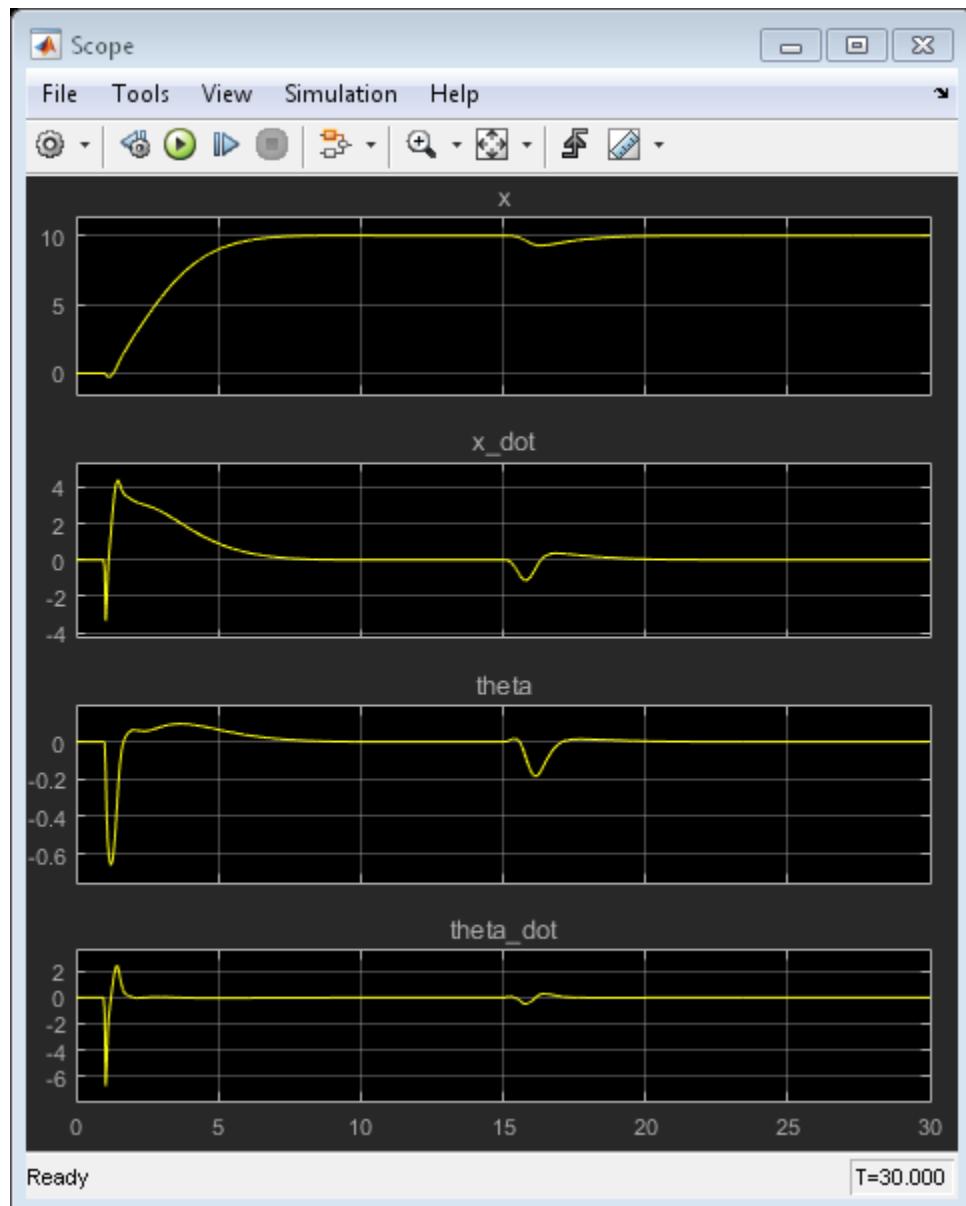
Closed-Loop Simulation

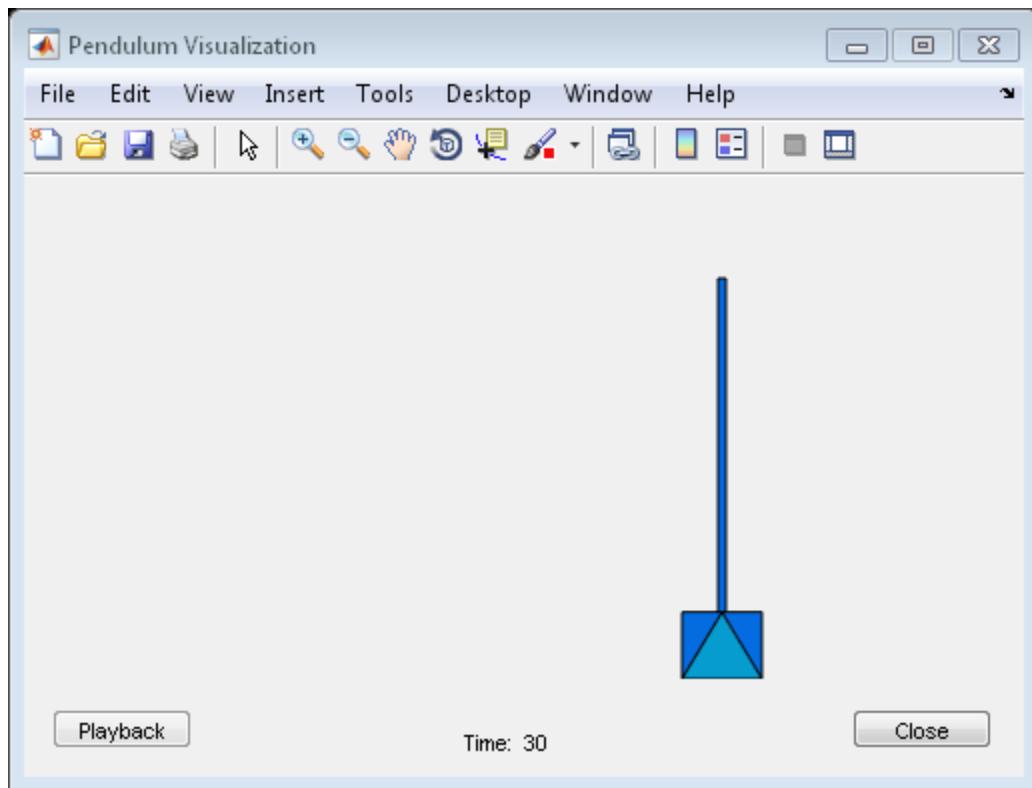
Validate the MPC design with a closed-loop simulation in Simulink.

```
open_system([mdlMPC /Scope ]);
sim(mdlMPC);

-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

4 Case-Study Examples





In the nonlinear simulation, all the control objectives are successfully achieved.

Discussion

It is important to point out that the designed MPC controller has its limitations. For example, if you increase the step setpoint change to 15, the pendulum fails to recover its upright position during the transition.

To reach the longer distance within the same rise time, the controller applies more force to the cart at the beginning. As a result, the pendulum is displaced from its upright position by a larger angle such as 60 degrees. At such angles, the plant dynamics differ significantly from the LTI predictive model obtained at $\theta = 0$. As a result, errors in the prediction of plant behavior exceed what the built-in MPC robustness can handle, and the controller fails to perform properly.

A simple workaround to avoid the pendulum falling is to restrict pendulum displacement by adding soft output constraints to *theta* and reducing the ECR weight on constraint softening.

```
mpcobj.OV(2).Min = -pi/2;  
mpcobj.OV(2).Max = pi/2;  
mpcobj.Weights.ECR = 100;
```

However, with these new controller settings, it is no longer possible to reach the longer distance within the required rise time. In other words, controller performance is sacrificed to avoid violation of soft output constraints.

To reach longer distances within the same rise time, the controller needs more accurate models at different angle to improve prediction. Another example “Gain Scheduled MPC Control of an Inverted Pendulum on a Cart” shows how to use gain scheduling MPC to achieve the longer distances.

```
bdclose(md1MPC);
```

More About

- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-42
- “Gain Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 7-39

Simulate MPC Controller with a Custom QP Solver

This example shows how to simulate the closed-loop response of an MPC controller with a custom quadratic programming (QP) solver in Simulink®.

We use an on-line monitoring example, first solving it by using the MPC Toolbox™ built-in solver, then using the quadprog solver from the Optimization Toolbox™.

Introduction

In the on-line monitoring example, the `qp.status` output of the MPC Controller block returns a positive integer whenever the controller obtains a valid solution of the current run-time QP problem and sets the `mv` output. The `qp.status` value corresponds to the number of iterations used to solve this QP.

If the QP is infeasible for a given control interval, the controller fails to find a solution. In that case, the `mv` outport stays at its most recent value and the `qp.status` outport returns -1. Similarly, if the maximum number of iterations is reached during optimization (rare), the `mv` outport also freezes and the `qp.status` outport returns 0.

Real-time MPC applications can detect whether the controller is in a "failure" mode (0 or -1) by monitoring the `qp.status` outport. If a failure occurs, a backup control plan should be activated. This is essential if there is any chance that the QP could become infeasible, because the default action (freezing MVs) may lead to unacceptable system behavior, such as instability. Such a backup plan is, necessarily, application-specific.

MPC Application with Online Monitoring

The plant used in this example is a single-input, single-output system with hard limits on both the manipulated variable (MV) and the controlled output (OV). The control objective is to hold the OV at a setpoint of 0. An unmeasured load disturbance is added to the OV. This disturbance is initially a ramp increase. The controller response eventually saturates the MV at its hard limit. Once saturation occurs, the controller can do nothing more, and the disturbance eventually drives the OV above its specified hard upper limit. When the controller predicts that it is impossible to force the OV below this upper limit, the run-time QP becomes infeasible.

Define the plant as a first-order SISO system with unity gain.

```
Plant = tf(1,[2 1]);
```

Define the unmeasured load disturbance. The signal ramps up from 0 to 2 between 1 and 3 seconds, then ramps back down from 2 to 0 between 3 and 5 seconds.

```
LoadDist = [0 0; 1 0; 3 2; 5 0; 7 0];
```

Design MPC Controller

Create an MPC object using the model of the test plant. The chosen control interval is about one tenth of the dominant plant time constant.

```
Ts = 0.2;
Obj = mpc(Plant, Ts);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define hard constraints on plant input (MV) and output (OV). By default, all the MV constraints are hard and OV constraints are soft.

```
Obj.MV.Min = -0.5;
Obj.MV.Max = 1;
Obj.OV.Min = -1;
Obj.OV.Max = 1;
Obj.OV.MinECR = 0; % change OV lower limit from soft to hard
Obj.OV.MaxECR = 0; % change OV upper limit from soft to hard
```

Generally, hard OV constraints are discouraged and are used here only to illustrate how to detect an infeasible QP. Hard OV constraints make infeasibility likely, in which case a backup control plan is essential. This example does not include a backup plan. However, as shown in the simulation, the default action of freezing the single MV is the best response in this simple case.

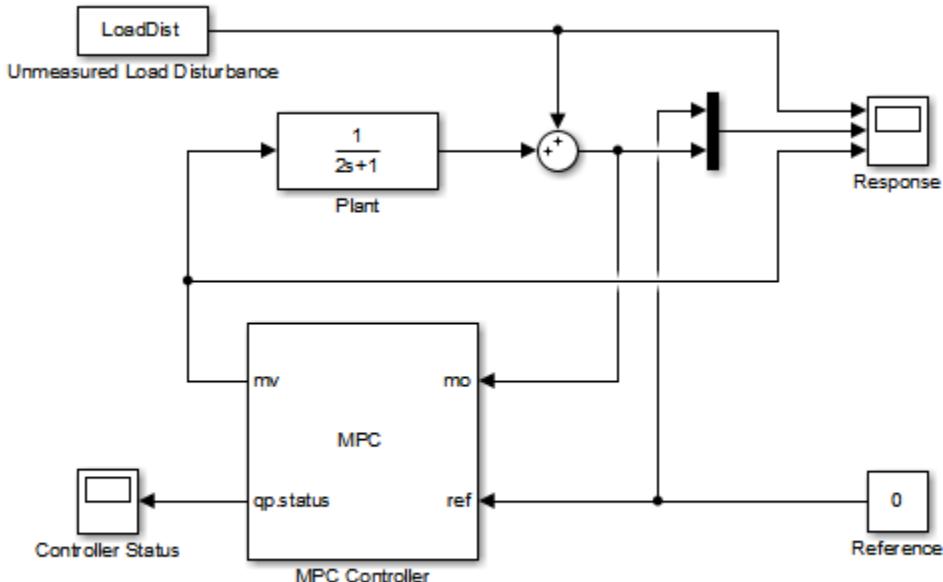
Simulate Using Simulink with Built-in QP Solver

To run this example, Simulink and the Optimization Toolbox are required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink is required to run this example. )
    return
end
if ~mpcchecktoolboxinstalled( optim )
    disp( The Optimization Toolbox is required to run this example. )
    return
end
```

Build the control system in a Simulink model and enable the `qp.status` outport from the controller block dialog. Its run-time value is displayed in a Simulink Scope block.

```
mdl = mpc_onlinemonitoring ;
open_system(mdl);
```



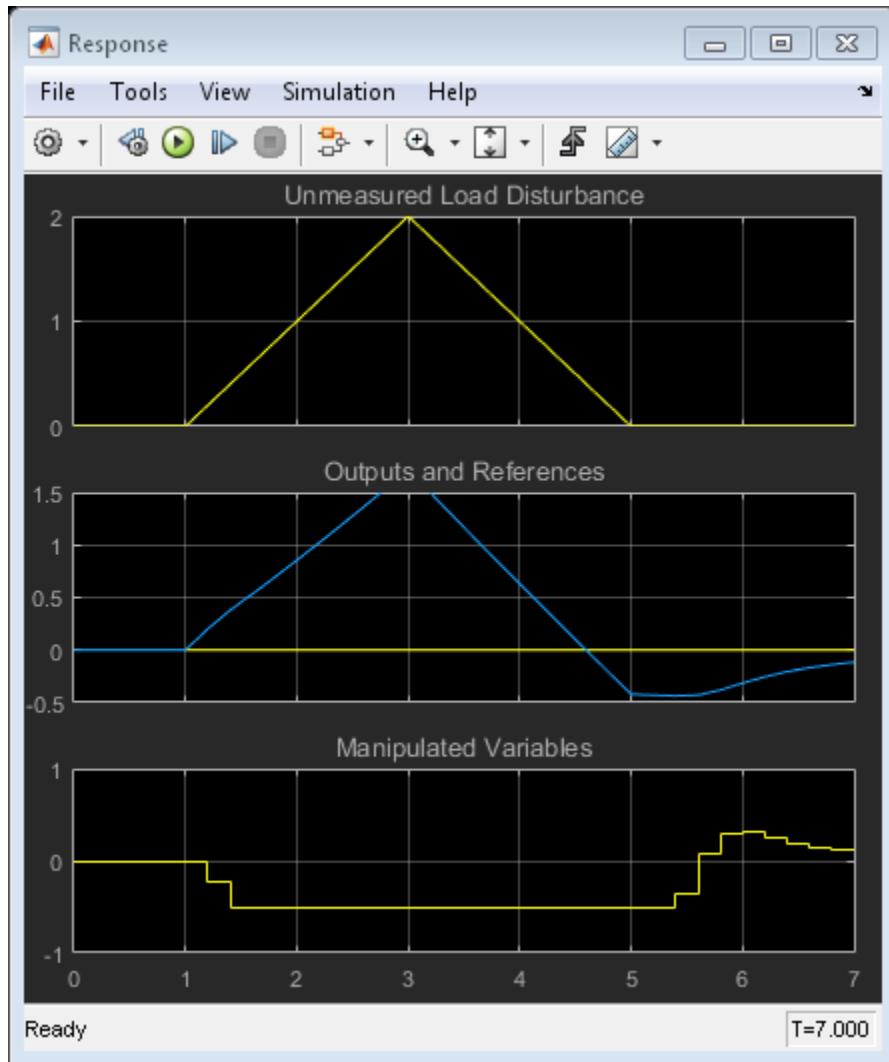
Copyright 1990-2014 The MathWorks, Inc.

Simulate the closed-loop response using the default Model Predictive Control Toolbox QP solver.

```
open_system([mdl /Controller Status ]);
open_system([mdl /Response ]);
sim(mdl);

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```





Explanation of the Closed-Loop Response

As shown in the response scope, at 1.4 seconds, the increasing disturbance causes the MV to saturate at its lower bound of -0.5, which is the QP solution under these conditions (because the controller is trying to hold the OV at its setpoint of 0).

The OV continues to increase due to the ramp disturbance and, at 2.2 seconds, exceeds the specified hard upper bound of 1.0. Since the QP is formulated in terms of predicted outputs, the controller still predicts that it can bring OV back below 1.0 in the next move and therefore the QP problem is still feasible.

Finally, at $t = 3.2$ seconds, the controller predicts that it can no longer move the OV below 1.0 within the next control interval, and the QP problem becomes infeasible and `qp.status` changes to -1 at this time.

After three seconds, the disturbance is decreasing. At 3.8 seconds, the QP becomes feasible again. The OV is still well above its setpoint, however, and the MV remains saturated until 5.4 seconds, when the QP solution is to increase the MV as shown. From then on, the MV is not saturated, and the controller is able to drive the OV back to its setpoint.

When the QP is feasible, the built-in solver finds the solution in three iterations or less.

Simulate with a Custom QP Solver

To examine how the custom solver behaves under the same conditions, activate the custom solver option by setting a property in the MPC controller.

```
Obj.Optimizer.CustomSolver = true;
```

You must also provide a MATLAB® function that satisfies all the following requirements:

- Function name must be `mpcCustomSolver`.
- Function input and output arguments must comply (see the example below for details).
- Function must be on the MATLAB path.

For this example, use the custom solver defined in `mpcCustomSolver.txt`, which uses the `quadprog` command from the Optimization Toolbox as the custom QP solver:

Save the function in your working directory as a `.m` file.

```
src = which( 'mpcCustomSolver.txt' );
dest = fullfile(pwd, 'mpcCustomSolver.m');
copyfile(src,dest, f );
```

Review the saved `mpcCustomSolver.m` file.

```

function [x, status] = mpcCustomSolver(H, f, A, b, x0)
% mpcCustomSolver allows user to specify a custom quadratic programming
% (QP) solver to solve the QP problem formulated by MPC controller. When
% the "mpcobj.Optimizer.CustomSolver" property is set true, instead of
% using the built-in QP solver, MPC controller will now use the customer QP
% solver defined in this function for simulations in MATLAB and Simulink.
%
% The MPC QP problem is defined as follows:
%   Find an optimal solution, x, that minimizes the quadratic objective
%   function, J = 0.5*x *H*x + f *x, subject to linear inequality
%   constraints, A*x >= b.
%
% Inputs (provided by MPC controller at run-time):
%   H: a n-by-n Hessian matrix, which is symmetric and positive definite.
%   f: a n-by-1 column vector.
%   A: a m-by-n matrix of inequality constraint coefficients.
%   b: a m-by-1 vector of the right-hand side of inequality constraints.
%   x0: a n-by-1 vector of the initial guess of the optimal solution.
%
% Outputs (fed back to MPC controller at run-time):
%   x: must be a n-by-1 vector of optimal solution.
%   status: must be an finite integer of:
%           positive value: number of iterations used in computation
%                           0: maximum number of iterations reached
%                           -1: QP is infeasible
%                           -2: Failed to find a solution due to other reasons
% Note that even if solver failed to find an optimal solution, "x" must be
% returned as a n-by-1 vector (i.e. set it to the initial guess x0)
%
% DO NOT CHANGE LINES ABOVE

% The following code is an example of how to implement the custom QP solver
% in this function. It requires Optimization Toolbox to run.

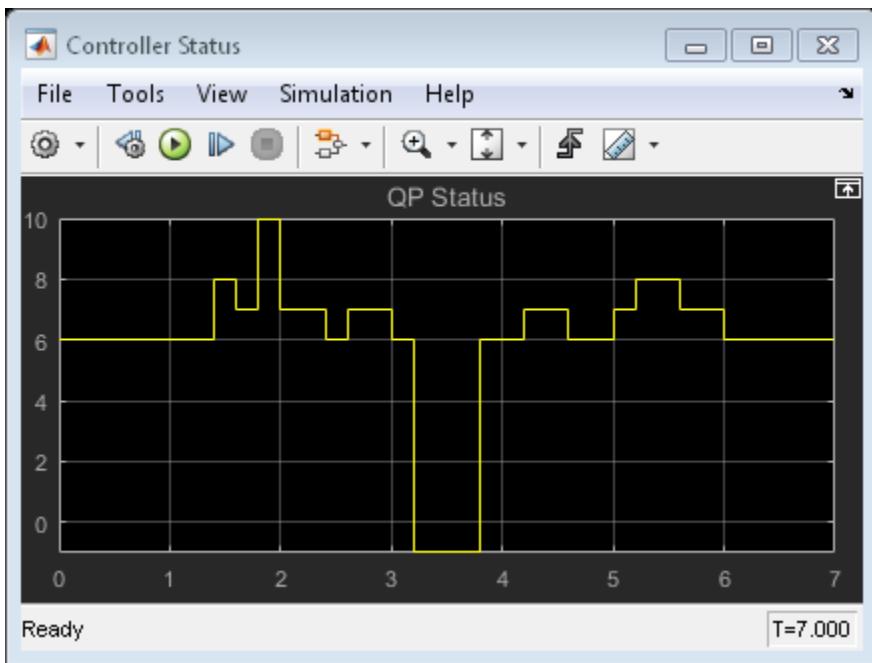
% Define QUADPROG options and turn off display of optimization results in
% Command window.
options = optimoptions( quadprog );
options.Display = none ;
%
% By definition, constraints required by "quadprog" solver is defined as
% A*x <= b. However, in our MPC QP problem, the constraints are defined as
% A*x >= b. Therefore, we need to implement some conversion here:
A_custom = -A;
b_custom = -b;
%
% Compute the QP s optimal solution. Note that the default algorithm used

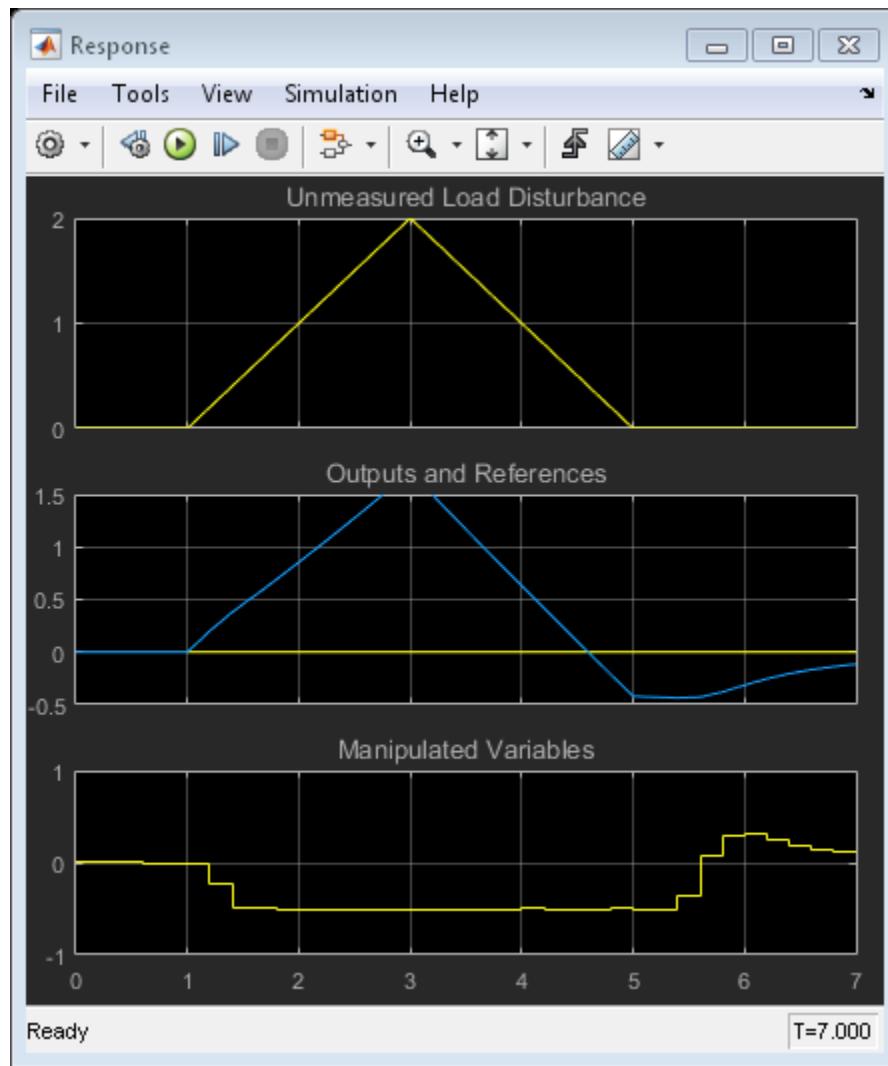
```

```
% by "quadprog" ( interior-point-convex ) ignores x0. "x0" is used here as
% an input argument for illustration only.
H = (H+H)/2; % ensure Hessian is symmetric
[x, ~, Flag, Output] = quadprog(H, f, A_custom, b_custom, [], [], [], x0, options)
% Converts the "flag" output to "status" required by the MPC controller.
switch Flag
    case 1
        status = Output.iterations;
    case 0
        status = 0;
    case -2
        status = -1;
    otherwise
        status = -2;
end
% Always return a non-empty x of the correct size. When the solver fails,
% one convenient solution is to set x to the initial guess.
if status <= 0
    x = x0;
end
```

Repeat the simulation.

```
set_param([mdl /Controller Status ], ymax , 10 );
sim(mdl)
```





The plant input and output signals are identical to those obtained using the built-in Model Predictive Control Toolbox solver, but the `qp.status` shows that `quadprog` does not take the same number of iterations to find a solution. However, it does detect the same infeasibility time period.

```
bdclose(mdl);
```

More About

- “QP Solver” on page 2-38

Adaptive MPC Design

- “Adaptive MPC” on page 5-2
- “Model Updating Strategy” on page 5-6
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21
- “Time-Varying MPC” on page 5-34
- “Time-Varying MPC Control of a Time-Varying Plant” on page 5-39

Adaptive MPC

In this section...

- “When to Use Adaptive MPC” on page 5-2
- “Plant Model” on page 5-3
- “Nominal Operating Point” on page 5-4
- “State Estimation” on page 5-4

When to Use Adaptive MPC

MPC control predicts future behavior using a linear-time-invariant (LTI) dynamic model. In practice, such predictions are never exact, and a key tuning objective is to make MPC insensitive to prediction errors. In many applications, this approach is sufficient for robust controller performance.

If the plant is strongly nonlinear or its characteristics vary dramatically with time, LTI prediction accuracy might degrade so much that MPC performance becomes unacceptable. Adaptive MPC can address this degradation by adapting the prediction model for changing operating conditions. As implemented in the Model Predictive Control Toolbox software, adaptive MPC uses a fixed model structure, but allows the models parameters to evolve with time. Ideally, whenever the controller requires a prediction (at the beginning of each control interval) it uses a model appropriate for the current conditions.

After you design an MPC controller for the average or most likely operating conditions of your control system, you can implement an adaptive MPC controller based on that design. For information about designing that initial controller, see “Controller Creation”.

At each control interval, the adaptive MPC controller updates the plant and model and nominal conditions. Once updated, the model and conditions remain constant over the prediction horizon. If you can predict how the plant and nominal conditions vary in the future, you can use “Time-Varying MPC” on page 5-34 to specify a model that changes over the prediction horizon.

An alternative option for controlling a nonlinear or time-varying plant is to use gain-scheduled MPC control. See “Gain-Scheduled MPC” on page 7-2.)

Plant Model

The plant model used as the basis for adaptive MPC must be an LTI discrete-time, state-space model. See “Basic Models” in the Control System Toolbox documentation or “Linearization Basics” in the Simulink Control Design documentation for information about creating and modifying such systems. The plant model structure is as follows:

$$\begin{aligned}x(k+1) &= Ax(k) + B_u u(k) + B_v v(k) + B_d d(k) \\y(k) &= Cx(k) + D_v v(k) + D_d d(k).\end{aligned}$$

Here, the matrices A , B_u , B_v , B_d , C , D_v and D_d are the parameters that can vary with time. The other variables in the expression are:

- k — Time index (current control interval).
- x — n_x plant model states.
- u — n_u manipulated inputs (MVs). These are the one or more inputs that are adjusted by the MPC controller.
- v — n_v measured disturbance inputs.
- d — n_d unmeasured disturbance inputs.
- y — n_y plant outputs, including n_{ym} measured and n_{yu} unmeasured outputs. The total number of outputs, $n_y = n_{ym} + n_{yu}$. Also, $n_{ym} \geq 1$ (there is at least one measured output).

Additional requirements for the plant model in adaptive MPC control are:

- Sample time (T_s) is a constant and identical to the MPC control interval.
- Time delay (if any) is absorbed as discrete states (see, e.g., the Control System Toolbox `absorbDelay` command).
- n_x , n_u , n_y , n_d , n_{ym} , and n_{yu} are all constants.
- Adaptive MPC prohibits direct feed-through from any manipulated variable to any plant output. Thus, $D_u = 0$ in the above model.
- The input and output signal configuration remains constant.

For more details about creation of plant models for MPC control, see “Plant Specification”.

Nominal Operating Point

A traditional MPC controller includes a nominal operating point at which the plant model applies, such as the condition at which you linearize a nonlinear model to obtain the LTI approximation. The `Model.Nominal` property of the controller contains this information.

In adaptive MPC, as time evolves you should update the nominal operating point to be consistent with the updated plant model.

You can write the plant model in terms of deviations from the nominal conditions:

$$\begin{aligned}x(k+1) &= \bar{x} + A(x(k) - \bar{x}) + B(u_t(k) - \bar{u}_t) + \bar{\Delta x} \\y(k) &= \bar{y} + C(x(k) - \bar{x}) + D(u_t(k) - \bar{u}_t).\end{aligned}$$

Here, the matrices A , B , C , and D are the parameter matrices to be updated. u_t is the combined plant input variable, comprising the u , v , and d variables defined above. The nominal conditions to be updated are:

- \bar{x} — n_x nominal states
- $\bar{\Delta x}$ — n_x nominal state increments
- \bar{u}_t — n_{ut} nominal inputs
- \bar{y} — n_y nominal outputs

State Estimation

By default, MPC uses a static Kalman filter (KF) to update its controller states, which include the n_{xp} plant model states, n_d (≥ 0) disturbance model states, and n_n (≥ 0) measurement noise model states. This KF requires two gain matrices, L and M . By default, the MPC controller calculates them during initialization. They depend upon the plant, disturbance, and noise model parameters, and assumptions regarding the stochastic noise signals driving the disturbance and noise models. For more details about state estimation in traditional MPC, see “Controller State Estimation” on page 2-42.

Adaptive MPC uses a Kalman filter and adjusts the gains, L and M , at each control interval to maintain consistency with the updated plant model. The result is a linear-time-varying Kalman filter (LTVKF):

$$\begin{aligned}L_k &= \left(A_k P_{k|k-1} C_{m,k}^T + N \right) \left(C_{m,k} P_{k|k-1} C_{m,k}^T + R \right)^{-1} \\M_k &= P_{k|k-1} C_{m,k}^T \left(C_{m,k} P_{k|k-1} C_{m,k}^T + R \right)^{-1} \\P_{k+1|k} &= A_k P_{k|k-1} A_k^T - \left(A_k P_{k|k-1} C_{m,k}^T + N \right) L_k^T + Q.\end{aligned}$$

Here, Q , R , and N are constant covariance matrices defined as in MPC state estimation. A_k and $C_{m,k}$ are state-space parameter matrices for the entire controller state, defined as for traditional MPC but with the portions affected by the plant model updated to time k . The value $P_{k|k-1}$ is the state estimate error covariance matrix at time k based on information available at time $k-1$. Finally, L_k and M_k are the updated KF gain matrices. For details on the KF formulation used in traditional MPC, see “Controller State Estimation” on page 2-42. By default, the initial condition, $P_{0|-1}$, is the static KF solution prior to any model updates.

The KF gain and the state error covariance matrix depend upon the model parameters and the assumptions leading to the constant Q , R , and N matrices. If the plant model is constant, the expressions for L_k and M_k converge to the equivalent static KF solution used in traditional MPC.

The equations for the controller state evolution at time k are identical to the KF formulation of traditional MPC described in “Controller State Estimation” on page 2-42, but with the estimator gains and state space matrices updated to time k .

You have the option to update the controller state using a procedure external to the MPC controller, and then supply the updated state to MPC at each control instant, k . In this case, the MPC controller skips all KF and LTVKF calculations.

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

More About

- “Model Updating Strategy” on page 5-6
- “Controller State Estimation” on page 2-42

Model Updating Strategy

In this section...

“Overview” on page 5-6

“Other Considerations” on page 5-6

Overview

Typically, to implement “Adaptive MPC” on page 5-2 control, you employ one of the following model-updating strategies:

- **Successive linearization** — Given a mechanistic plant model, e.g., a set of nonlinear ordinary differential and algebraic equations, derive its LTI approximation at the current operating condition. For example, Simulink Control Design software provides linearization tools for this purpose.
- **Using a Linear Parameter Varying (LPV) model** — Control System Toolbox software provides a LPV System Simulink block that allows you to specify an array of LTI models with scheduling parameters. You can perform batch linearization offline to obtain an array of plant models at the desired operating points and then use them in the LPV System block to provide model updating to the Adaptive MPC Controller Simulink block.
- **Online parameter estimation** — Given an empirical model structure and initial estimates of its parameters, use the available real-time plant measurements to estimate the current model parameters. For example, the System Identification Toolbox™ software provides real-time parameter estimation tools.

To implement “Time-Varying MPC” on page 5-34 control, you need to obtain LTI plants for the future prediction horizon steps. In this case, you can use the successive linearization and LPV model approaches as long as each model is a function of time

Other Considerations

There are several factors to keep in mind when designing and implementing an adaptive MPC controller.

- Before attempting adaptive MPC, define and tune an MPC controller for the most typical (nominal) operating condition. Make sure the system can tolerate some

prediction error. Test this tolerance via simulations in which the MPC prediction model differs from the plant. See “MPC Design”.

- An adaptive MPC controller requires more real-time computations than traditional MPC. In addition to the state estimation calculation, you must also implement and test a model-updating strategy, which might be computationally intensive.
- You must determine MPC tuning constants that provide robust performance over the expected range of model parameters. See “Tuning Weights” on page 1-16.
- Model updating via online parameter estimation is most effective when parameter variations occur gradually.
- When implementing adaptive MPC control, adapt only parameters defining the `Model.Plant` property of the controller. The disturbance and noise models, if any, remain constant.

See Also

[Adaptive MPC Controller](#)

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

More About

- “Adaptive MPC” on page 5-2

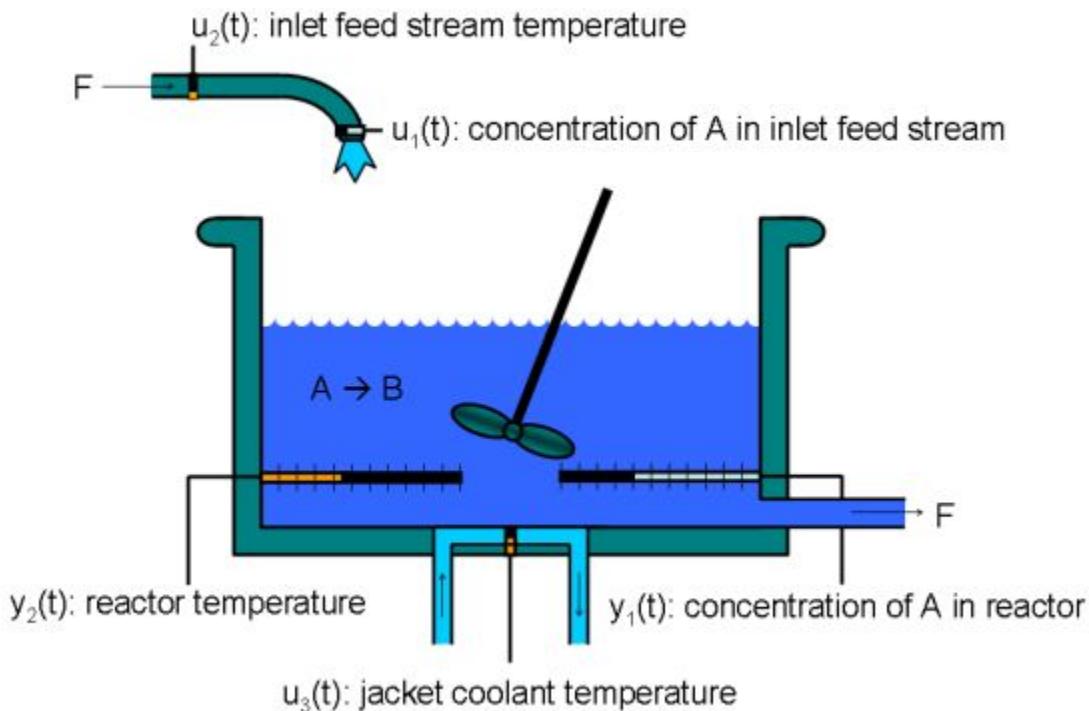
Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization

This example shows how to use an Adaptive MPC controller to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

A first principle nonlinear plant model is available and being linearized at each control interval. The adaptive MPC controller then updates its internal predictive model with the linearized plant model and achieves nonlinear control successfully.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:



This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A \rightarrow B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{aligned} u_1 &= CA_i && \text{Concentration of A in inlet feed stream} [kgmol/m^3] \\ u_2 &= T_i && \text{Inlet feed stream temperature} [K] \\ u_3 &= T_c && \text{Jacket coolant temperature} [K] \end{aligned}$$

and the outputs ($y(t)$), which are also the states of the model ($x(t)$), are:

$$\begin{aligned} y_1 &= x_1 = CA && \text{Concentration of A in reactor tank} [kgmol/m^3] \\ y_2 &= x_2 = T && \text{Reactor temperature} [K] \end{aligned}$$

The control objective is to maintain the concentration of reagent A, CA at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference as well as reject the measured disturbance arising from the inlet feed stream temperature T_i . The inlet feed stream concentration, CA_i , is assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

We also assume that direct measurements of concentrations are unavailable or infrequent, which is the usual case in practice. Instead, we use a "soft sensor" to estimate CA based on temperature measurements and the plant model.

About Adaptive Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical operating range. Next you can choose one of the two approaches to implement MPC control strategy:
 - (1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based on a desired scheduling strategy. See “Gain Scheduled MPC Control of Nonlinear Chemical Reactor” for more details. Use this approach when the plant models have different orders or time delays.
 - (2) Design one MPC controller offline at the initial operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.
- If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:
 - (1) Use successive linearization as shown in this example. Use this approach when a nonlinear plant model is available and can be linearized at run time.
 - (2) Use online estimation to identify a linear model when loop is closed. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” for more details. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

To linearize the plant, Simulink® and Simulink Control Design® are required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( 'Simulink(R) is required to run this example.' )
    return
end
if ~mpcchecktoolboxinstalled( slcontrol )
    disp( 'Simulink Control Design(R) is required to run this example.' )
    return
end
```

To implement an adaptive MPC controller, first you need to design a MPC controller at the initial operating point where CAi is 10 kgmol/m³, Ti and Tc are 298.15 K.

Create operating point specification.

```
plant_mdl = mpc_cstr_plant ;
op = operspec(plant_mdl);
```

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

```
[op_point, op_report] = findop(plant_mdl,op);
```

Operating Point Search Report:

```
-----
Operating Report for the Model mpc_cstr_plant.
(Time-Varying Components Evaluated at time t=0)
```

Operating point specifications were successfully met.
States:

```
-----
(1.) mpc_cstr_plant/CSTR/Integrator
      x:          311      dx:     8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:          8.57      dx:    -6.87e-12 (0)
```

Inputs:

```
-----
(1.) mpc_cstr_plant/CAi
```

```
      u:          10
(2.) mpc_cstr_plant/Ti
      u:         298
(3.) mpc_cstr_plant/Tc
      u:         298

Outputs:
-----
(1.) mpc_cstr_plant/T
      y:         311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:        8.57    [-Inf Inf]
```

Obtain nominal values of x, y and u.

```
x0 = [op_report STATES(1).x;op_report STATES(2).x];
y0 = [op_report Outputs(1).y;op_report Outputs(2).y];
u0 = [op_report Inputs(1).u;op_report Inputs(2).u;op_report Inputs(3).u];
```

Obtain linear plant model at the initial condition.

```
sys = linearize(plant_mdl, op_point);
```

Drop the first plant input CAi because it is not used by MPC.

```
sys = sys(:,2:3);
```

Discretize the plant model because Adaptive MPC controller only accepts a discrete-time plant model.

```
Ts = 0.5;
plant = c2d(sys,Ts);
```

Design MPC Controller

You design an MPC at the initial operating condition. When running in the adaptive mode, the plant model is updated at run time.

Specify signal types used in MPC.

```
plant.InputGroup.MeasuredDisturbances = 1;
plant.InputGroup.ManipulatedVariables = 2;
plant.OutputGroup.Measured = 1;
```

```
plant.OutputGroup.Unmeasured = 2;
plant.InputName = { Ti , Tc };
plant.OutputName = { T , CA };
```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Set nominal values in the controller

```
mpcobj.Model.Nominal = struct( X , x0, U , u0(2:3), Y , y0, DX , [0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude

```
Uscale = [30 50];
Yscale = [50 10];
mpcobj.DV(1).ScaleFactor = Uscale(1);
mpcobj.MV(1).ScaleFactor = Uscale(2);
mpcobj.OV(1).ScaleFactor = Yscale(1);
mpcobj.OV(2).ScaleFactor = Yscale(2);
```

Let reactor temperature T float (i.e. with no setpoint tracking error penalty), because the objective is to control reactor concentration CA and only one manipulated variable (coolant temperature Tc) is available.

```
mpcobj.Weights.OV = [0 1];
```

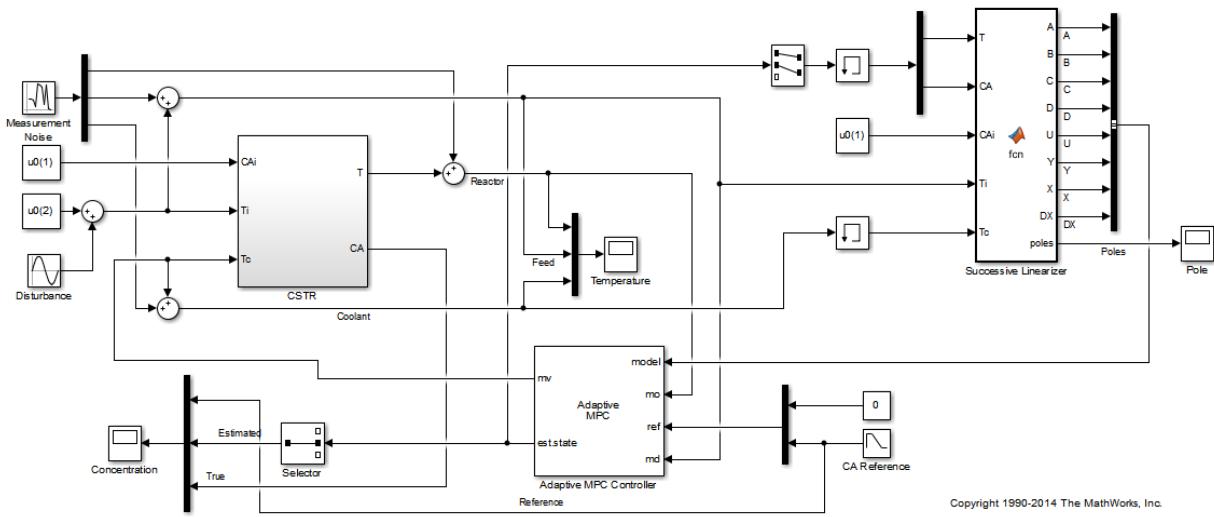
Due to the physical constraint of coolant jacket, Tc rate of change is bounded by degrees per minute.

```
mpcobj.MV.RateMin = -2;
mpcobj.MV.RateMax = 2;
```

Implement Adaptive MPC Control of CSTR Plant in Simulink (R)

Open the Simulink model.

```
mdl = ampc_cstr_linearization ;
open_system(md1);
```



Copyright 1990-2014 The MathWorks, Inc.

The model includes three parts:

- 1 The "CSTR" block implements the nonlinear plant model.
- 2 The "Adaptive MPC Controller" block runs the designed MPC controller in the adaptive mode.
- 3 The "Successive Linearizer" block in a MATLAB Function block that linearizes a first principle nonlinear CSTR plant and provides the linear plant model to the "Adaptive MPC Controller" block at each control interval. Double click the block to see the MATLAB code. You can use the block as a template to develop appropriate linearizer for your own applications.

Note that the new linear plant model must be a discrete time state space system with the same order and sample time as the original plant model has. If the plant has time delay, it must also be same as the original time delay and absorbed into the state space model.

Validate Adaptive MPC Control Performance

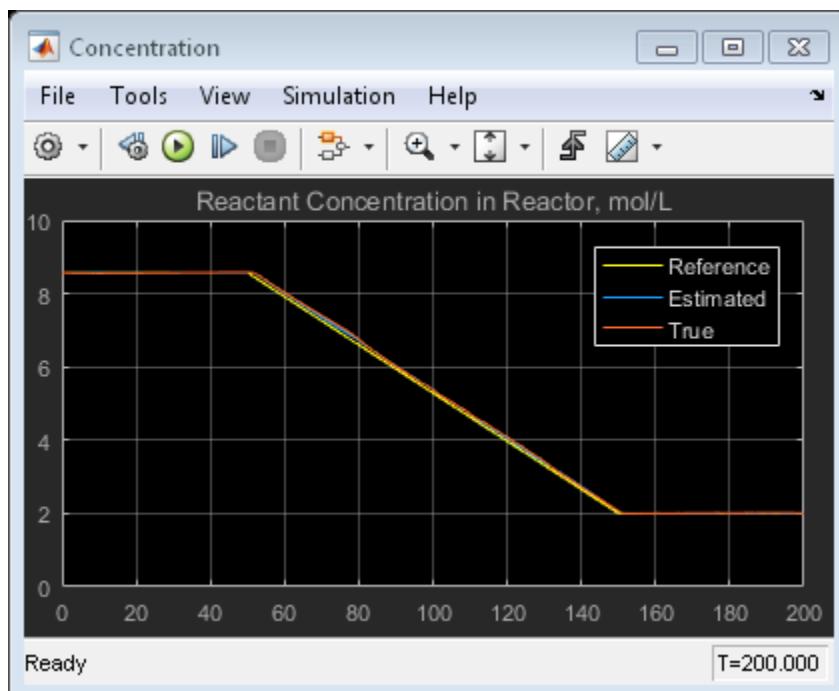
Controller performance is validated against both setpoint tracking and disturbance rejection.

- Tracking: reactor concentration CA setpoint transitions from original 8.57 (low conversion rate) to 2 (high conversion rate) kgmol/m³. During the transition, the plant first becomes unstable then stable again (see the poles plot).
- Regulating: feed temperature Ti has slow fluctuation represented by a sine wave with amplitude of 5 degrees, which is a measured disturbance fed to the MPC controller.

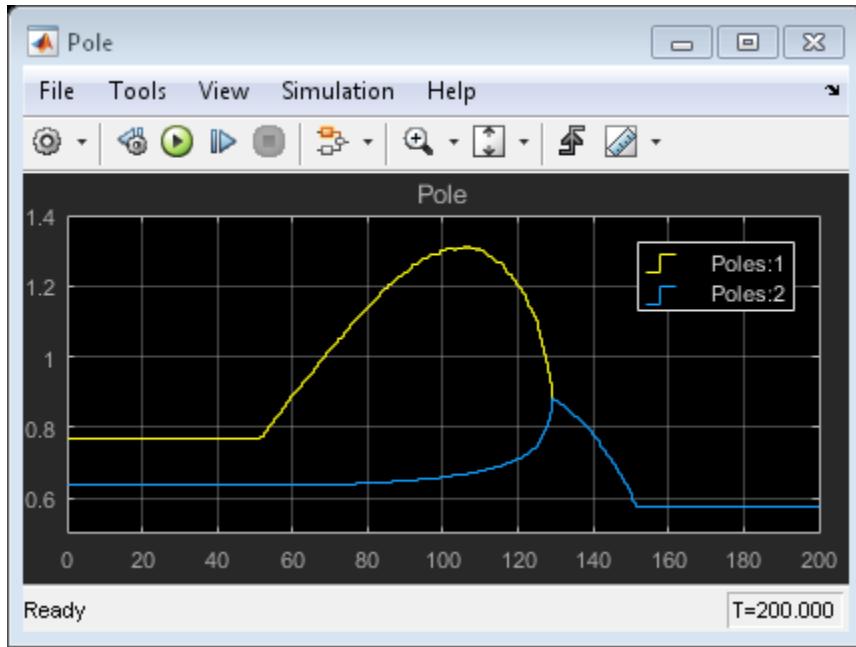
Simulate the closed-loop performance.

```
open_system([mdl '/Concentration'])
open_system([mdl '/Temperature'])
open_system([mdl '/Pole'])
sim(mdl);

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```





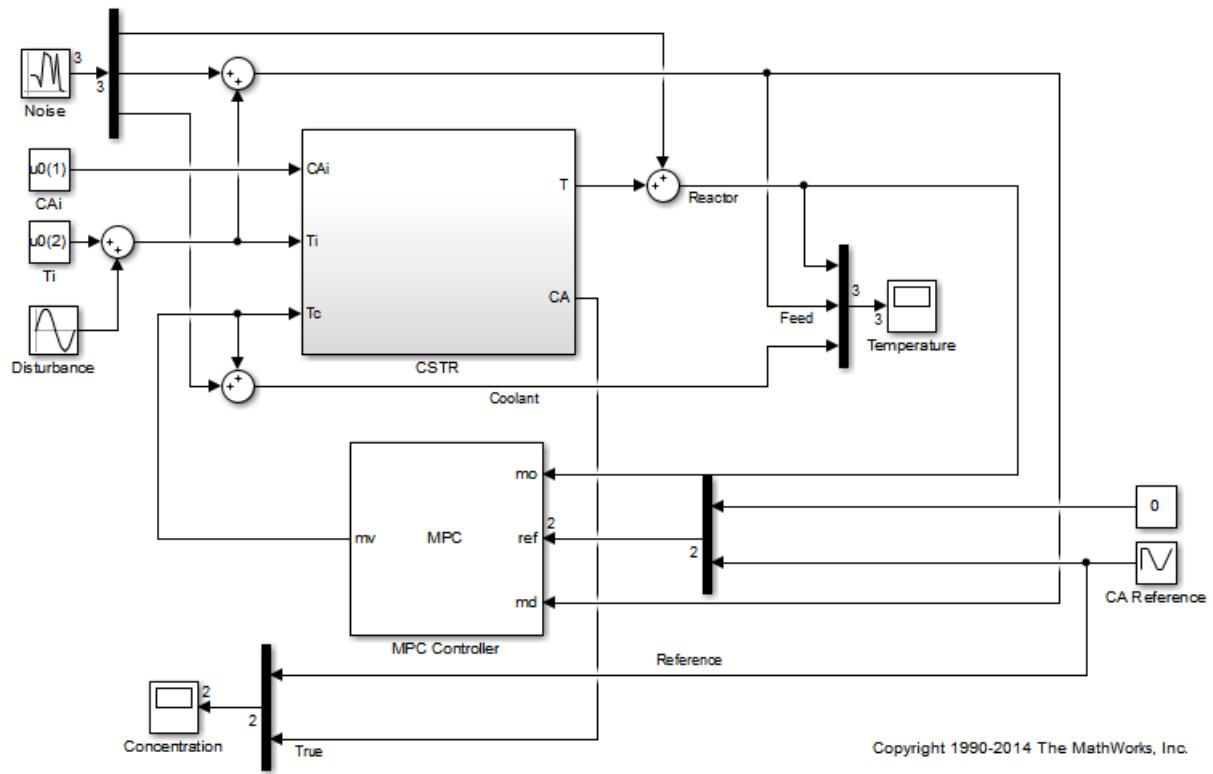


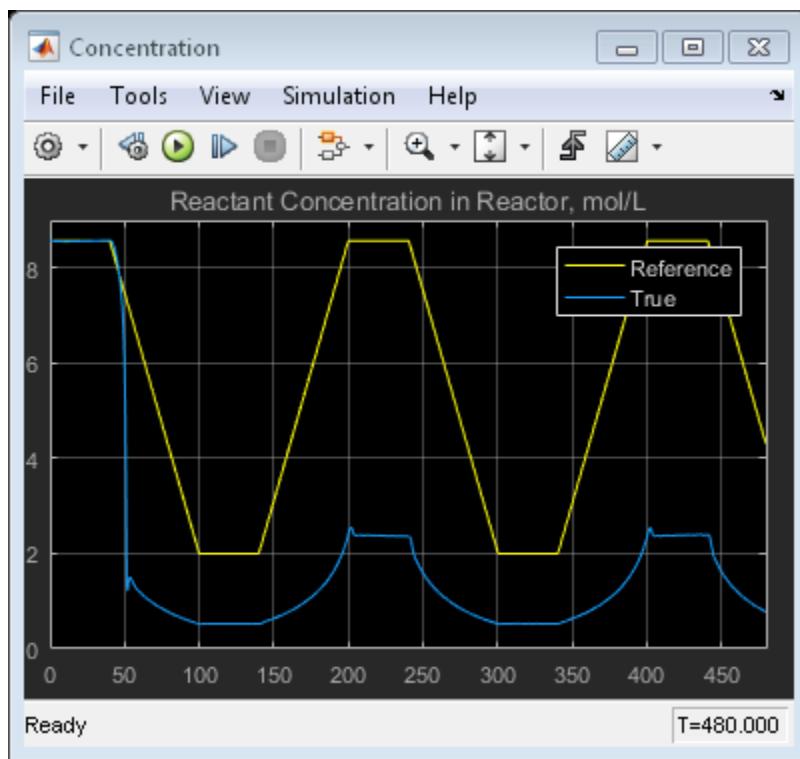
The tracking and regulating performance is very satisfactory. In an application to a real reactor, however, model inaccuracies and unmeasured disturbances could cause poorer tracking than shown here. Additional simulations could be used to study these effects.

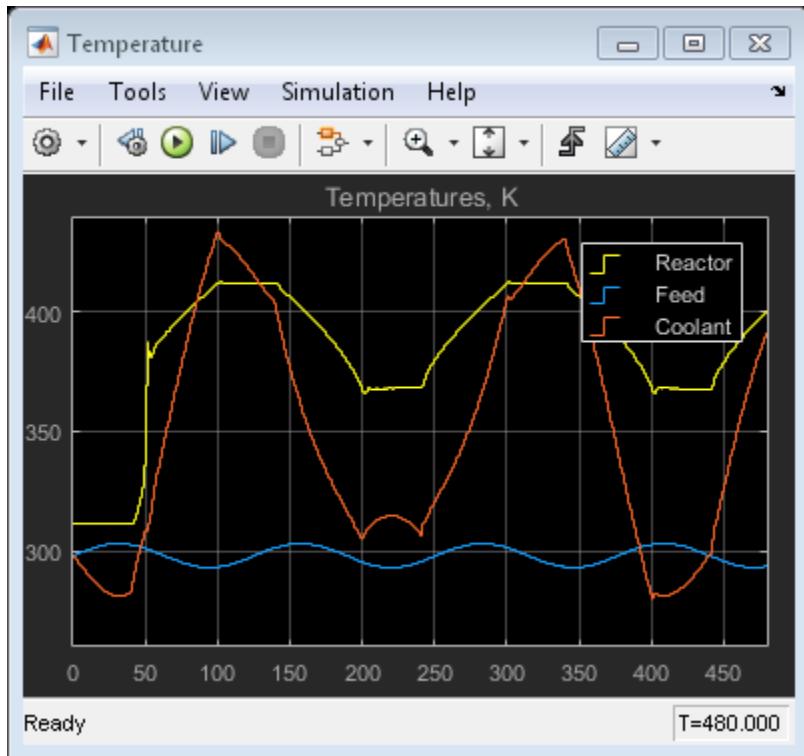
Compare with Non-Adaptive MPC Control

Adaptive MPC provides superior control performance than a non-adaptive MPC. To illustrate this point, the control performance of the same MPC controller running in the non-adaptive mode is shown below. The controller is implemented with a MPC Controller block.

```
mdl1 = 'ampc_cstr_no_linearization';
open_system(mdl1);
open_system([mdl1 '/Concentration'])
open_system([mdl1 '/Temperature'])
sim(mdl1);
```







As expected, the tracking and regulating performance is unacceptable.

```
bdclose(md1)  
bdclose(md11)
```

See Also

[Adaptive MPC Controller](#)

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

More About

- “Adaptive MPC” on page 5-2

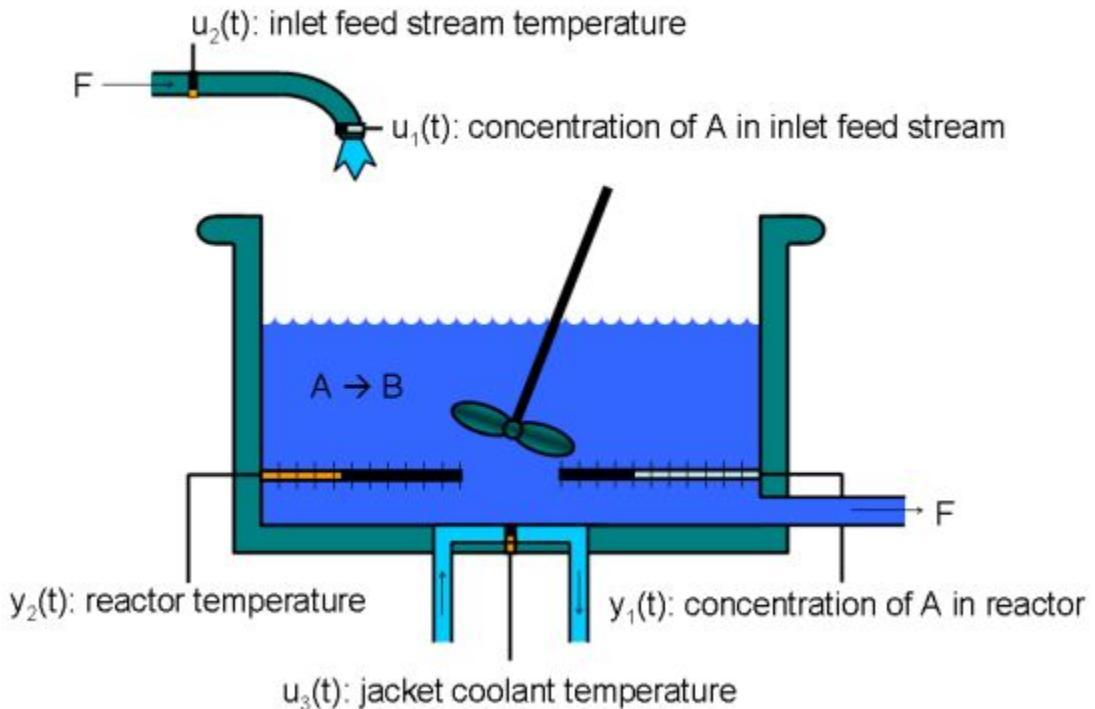
Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation

This example shows how to use an Adaptive MPC controller to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

A discrete time ARX model is being identified online by the Recursive Polynomial Model Estimator block at each control interval. The adaptive MPC controller uses it to update internal plant model and achieves nonlinear control successfully.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:



This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A \rightarrow B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{aligned} u_1 &= CA_i && \text{Concentration of A in inlet feed stream [kgmol/m}^3\text{]} \\ u_2 &= T_i && \text{Inlet feed stream temperature [K]} \\ u_3 &= T_c && \text{Jacket coolant temperature [K]} \end{aligned}$$

and the outputs ($y(t)$), which are also the states of the model ($x(t)$), are:

$$\begin{aligned} y_1 = x_1 &= CA && \text{Concentration of A in reactor tank [kgmol/m}^3\text{]} \\ y_2 = x_2 &= T && \text{Reactor temperature [K]} \end{aligned}$$

The control objective is to maintain the reactor temperature T at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference as well as reject the measured disturbance arising from the inlet feed stream temperature T_i . The inlet feed stream concentration, CA_i , is assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

About Adaptive Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical

operating range. Next you can choose one of the two approaches to implement MPC control strategy:

- (1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based on a desired scheduling strategy. See “Gain Scheduled MPC Control of Nonlinear Chemical Reactor” for more details. Use this approach when the plant models have different orders or time delays.
- (2) Design one MPC controller offline at the initial operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.
 - If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:
 - (1) Use successive linearization. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” for more details. Use this approach when a nonlinear plant model is available and can be linearized at run time.
 - (2) Use online estimation to identify a linear model when loop is closed, as shown in this example. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

To linearize the plant, Simulink® and Simulink Control Design® are required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end
if ~mpcchecktoolboxinstalled( slcontrol )
    disp( Simulink Control Design(R) is required to run this example. )
    return
end
```

To implement an adaptive MPC controller, first you need to design a MPC controller at the initial operating point where CAi is 10 kgmol/m³, Ti and Tc are 298.15 K.

Create operating point specification.

```
plant_mdl = mpc_cstr_plant ;
op = operspec(plant_mdl);
```

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

```
[op_point, op_report] = findop(plant_mdl,op);
```

Operating Point Search Report:

```
-----
Operating Report for the Model mpc_cstr_plant.
(Time-Varying Components Evaluated at time t=0)
```

Operating point specifications were successfully met.

States:

```
-----
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           311      dx:     8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:           8.57      dx:    -6.87e-12 (0)
```

Inputs:

```
-----
(1.) mpc_cstr_plant/CAi
      u:           10
(2.) mpc_cstr_plant/Ti
```

```

      u:          298
(3.) mpc_cstr_plant/Tc
      u:          298

Outputs:
-----
(1.) mpc_cstr_plant/T
    y:          311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
    y:          8.57    [-Inf Inf]

```

Obtain nominal values of x, y and u.

```

x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];

```

Obtain linear plant model at the initial condition.

```
sys = linearize(plant_mdl, op_point);
```

Drop the first plant input CA_i and second output CA because they are not used by MPC.

```
sys = sys(1,2:3);
```

Discretize the plant model because Adaptive MPC controller only accepts a discrete-time plant model.

```
Ts = 0.5;
plant = c2d(sys,Ts);
```

Design MPC Controller

You design an MPC at the initial operating condition. When running in the adaptive mode, the plant model is updated at run time.

Specify signal types used in MPC.

```

plant.InputGroup.MeasuredDisturbances = 1;
plant.InputGroup.ManipulatedVariables = 2;
plant.OutputGroup.Measured = 1;
plant.InputName = { Ti , Tc };
plant.OutputName = { T };

```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant);  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.
```

Set nominal values in the controller

```
mpcobj.Model.Nominal = struct( X , x0, U , u0(2:3), Y , y0(1), DX , [0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude

```
Uscale = [30 50];  
Yscale = 50;  
mpcobj.DV.ScalerFactor = Uscale(1);  
mpcobj.MV.ScalerFactor = Uscale(2);  
mpcobj.OV.ScalerFactor = Yscale;
```

Due to the physical constraint of coolant jacket, Tc rate of change is bounded by 2 degrees per minute.

```
mpcobj.MV.RateMin = -2;  
mpcobj.MV.RateMax = 2;
```

Reactor concentration is not directly controlled in this example. If reactor temperature can be successfully controlled, the concentration will achieve desired performance requirement due to the strongly coupling between the two variables.

Implement Adaptive MPC Control of CSTR Plant in Simulink (R)

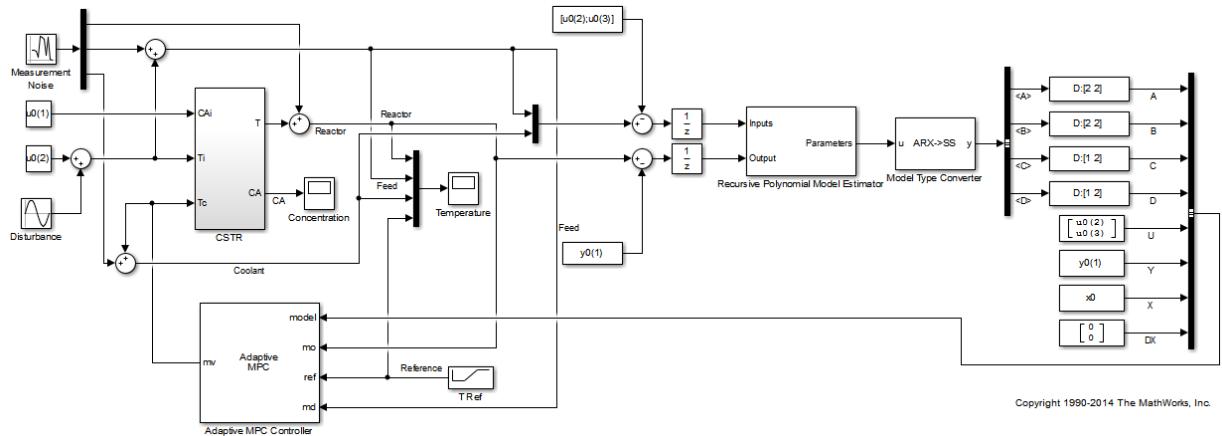
To run this example with online estimation, System Identification® is required.

```
if ~mpcchecktoolboxinstalled( ident )  
    disp( System Identification(R) is required to run this example. )  
    return  
end
```

Open the Simulink model.

```
mdl = ampc_cstr_estimation ;
```

```
open_system(md1);
```



The model includes three parts:

- 1 The "CSTR" block implements the nonlinear plant model.
- 2 The "Adaptive MPC Controller" block runs the designed MPC controller in the adaptive mode.
- 3 The "Recursive Polynomial Model Estimator" block estimates a two-input (T_i and T_c) and one-output (T) discrete time ARX model based on the measured temperatures. The estimated model is then converted into state space form by the "Model Type Converter" block and fed to the "Adaptive MPC Controller" block at each control interval.

In this example, the initial plant model is used to initialize the online estimator with parameter covariance matrix set to 1. The online estimation method is "Kalman Filter" with noise covariance matrix set to 0.01. The online estimation result is sensitive to these parameters and you can further adjust them to achieve better estimation result.

Both "Recursive Polynomial Model Estimator" and "Model Type Converter" are provided by System Identification Toolbox. You can use the two blocks as a template to develop appropriate online model estimation for your own applications.

The initial value of $A(q)$ and $B(q)$ variables are populated with the numerator and denominator of the initial plant model.

```
[num, den] = tfdata(plant);
Aq = den{1};
Bq = num;
```

Note that the new linear plant model must be a discrete time state space system with the same order and sample time as the original plant model has. If the plant has time delay, it must also be same as the original time delay and absorbed into the state space model.

Validate Adaptive MPC Control Performance

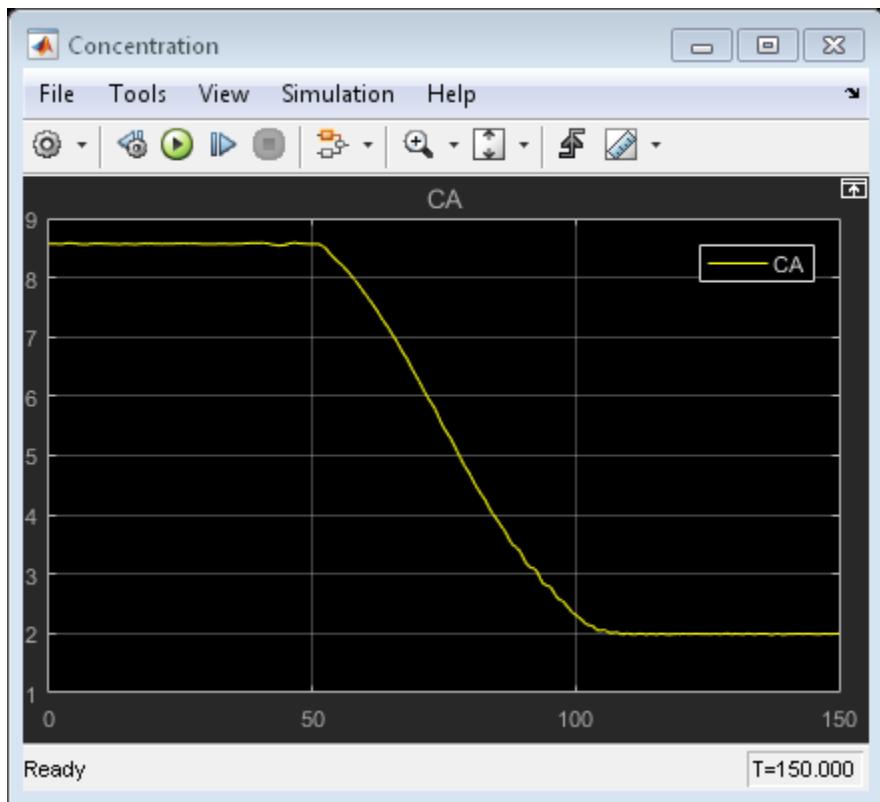
Controller performance is validated against both setpoint tracking and disturbance rejection.

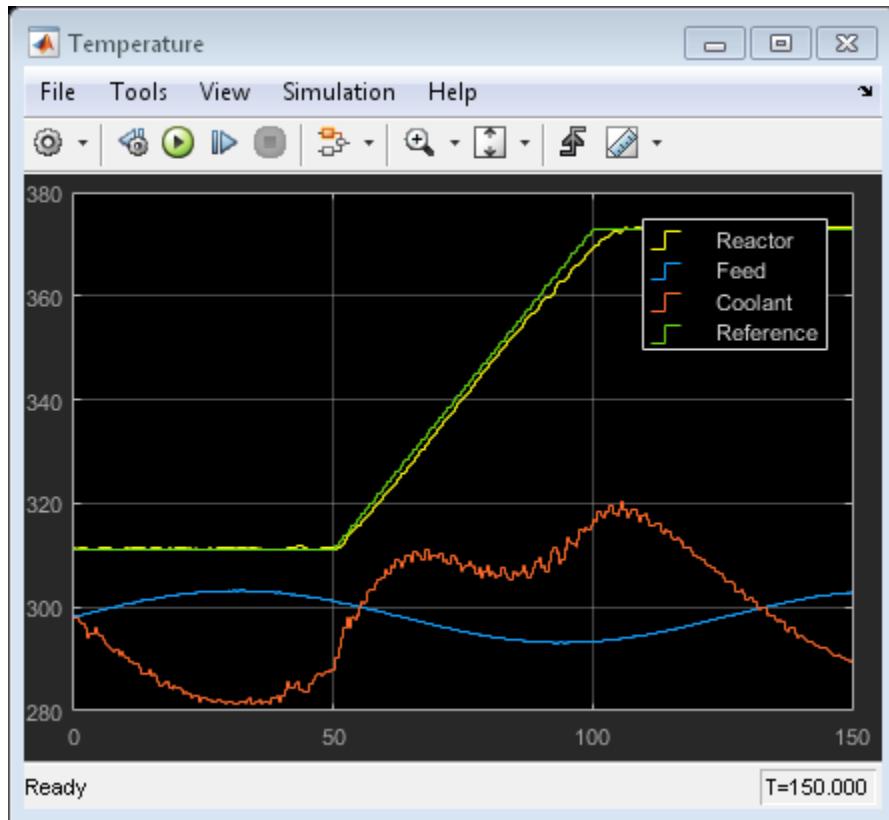
- Tracking: reactor temperature T setpoint transitions from original 311 K (low conversion rate) to 377 K (high conversion rate) kgmol/m³. During the transition, the plant first becomes unstable then stable again (see the poles plot).
- Regulating: feed temperature Ti has slow fluctuation represented by a sine wave with amplitude of 5 degrees, which is a measured disturbance fed to MPC controller.

Simulate the closed-loop performance.

```
open_system([mdl /Concentration ])
open_system([mdl /Temperature ])
sim(mdl);

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```



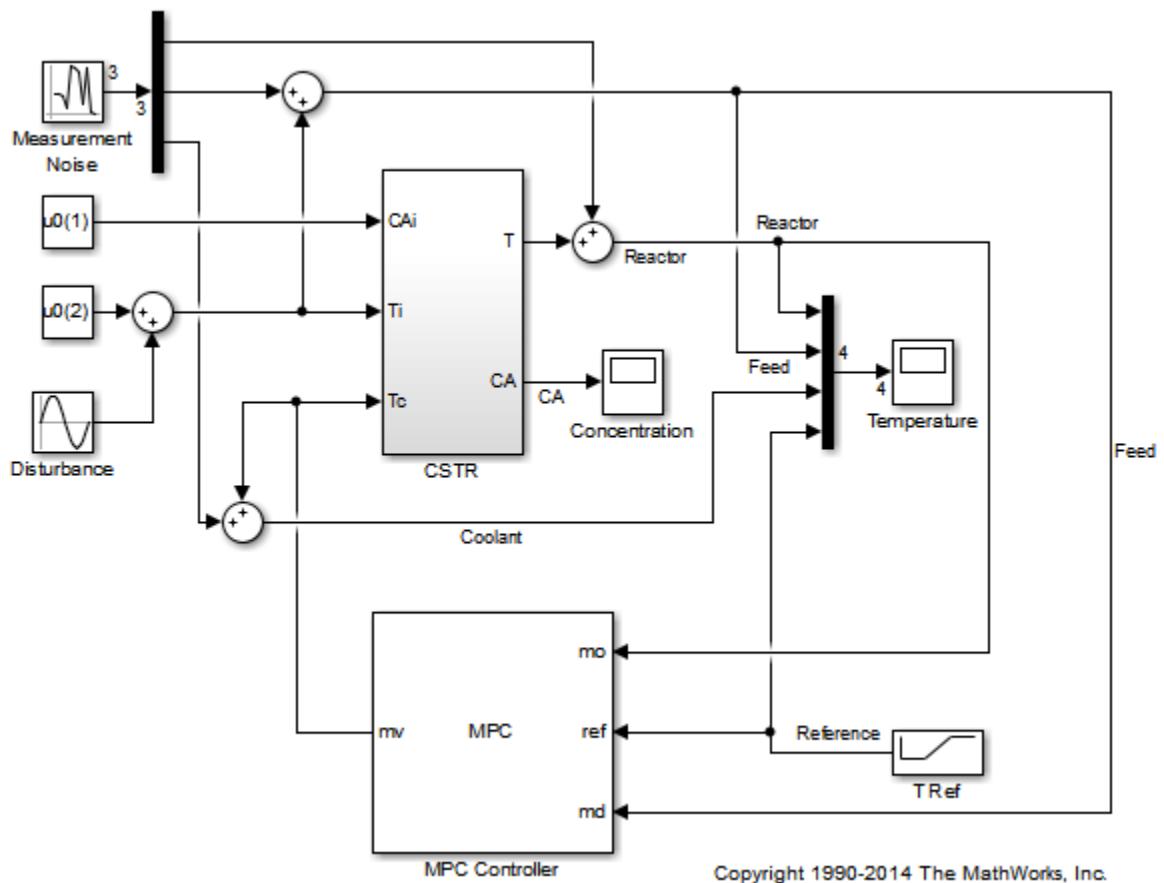


The tracking and regulating performance is very satisfactory.

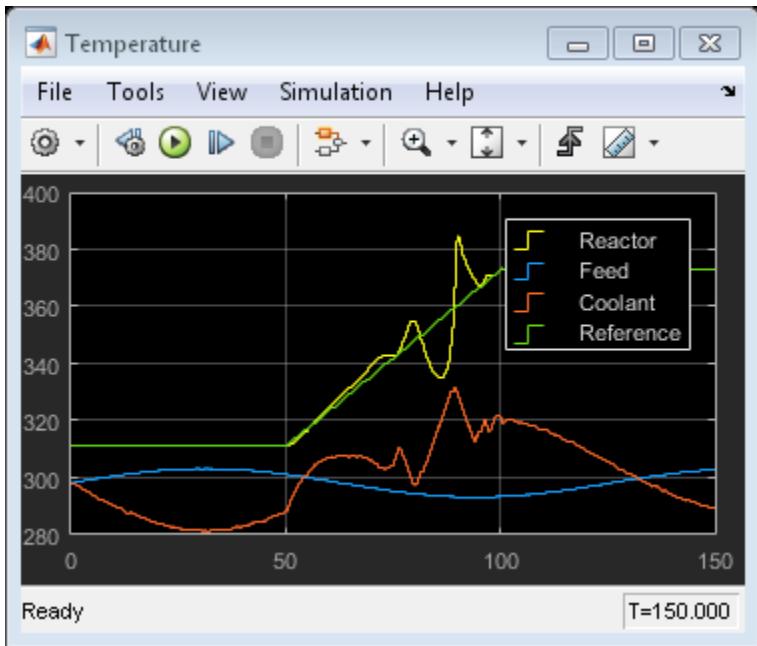
Compare with Non-Adaptive MPC Control

Adaptive MPC provides superior control performance than a non-adaptive MPC. To illustrate this point, the control performance of the same MPC controller running in the non-adaptive mode is shown below. The controller is implemented with a MPC Controller block.

```
mdl1 = 'ampc_cstr_no_estimation';
open_system(mdl1);
open_system([mdl1 '/Concentration']);
open_system([mdl1 '/Temperature']);
sim(mdl1);
```







As expected, the tracking and regulating performance is unacceptable.

```
bdclose(mdl)
bdclose(mdl1)
```

See Also

Adaptive MPC Controller

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8

More About

- “Adaptive MPC” on page 5-2

Time-Varying MPC

In this section...

[“When to Use Time-Varying MPC” on page 5-34](#)

[“Time-Varying Prediction Models” on page 5-34](#)

[“Time-Varying Nominal Conditions” on page 5-36](#)

[“State Estimation” on page 5-37](#)

When to Use Time-Varying MPC

To adapt to changing operating conditions, adaptive MPC supports updating the prediction model and its associated nominal conditions at each control interval. However, the updated model and conditions remain constant over the prediction horizon. If you can predict how the plant and nominal conditions vary in the future, you can use time-varying MPC to specify a model that changes over the prediction horizon. Such a linear time-varying (LTV) model is useful when controlling periodic systems or nonlinear systems that are linearized around a time-varying nominal trajectory.

To use time-varying MPC, specify arrays for the `Plant` and `Nominal` input arguments of `mpcmoveAdaptive`. For an example of time-varying MPC, see “[Time-Varying MPC Control of a Time-Varying Plant](#)” on page 5-39.

Time-Varying Prediction Models

Consider the LTV prediction model

$$\begin{aligned}x(k+1) &= A(k)x(k) + B_u(k)u(k) + B_v(k)v(k) \\y(k) &= C(k)x(k) + D_v(k)v(k)\end{aligned}$$

where A , B_u , B_v , C , and D are discrete-time state-space matrices that can vary with time. The other model parameters are:

- k — Current control interval time index
- x — Plant model states

- u — Manipulated variables
- v — Measured disturbance inputs
- y — Measured and unmeasured plant outputs

Since time-varying MPC extends adaptive MPC, the plant model requirements are the same; that is, for each model in the `Plant` array:

- Sample time (`Ts`) is constant and identical to the MPC controller sample time.
- Any time delays are absorbed as discrete states.
- The input and output signal configuration remains constant.
- There is no direct feed-through from the manipulated variables to the plant outputs.

For more information, see “Plant Model” on page 5-3.

The prediction of future trajectories for p steps into the future, where p is the prediction horizon, is the same as for the adaptive MPC case:

$$\begin{bmatrix} y(1) \\ \vdots \\ y(p) \end{bmatrix} = S_x x(0) + S_{u1} u(-1) + S_u \begin{bmatrix} \Delta u(0) \\ \vdots \\ \Delta u(p-1) \end{bmatrix} + H_v \begin{bmatrix} v(0) \\ \vdots \\ v(p) \end{bmatrix}$$

However, for an LTV prediction model, the matrices S_x , S_{u1} , S_u , and H_v are:

$$\begin{aligned}
 S_x &= \begin{bmatrix} C(1)A(0) \\ C(2)A(1)A(0) \\ \vdots \\ C(p)\prod_{i=0}^{p-1} A(i) \end{bmatrix} \\
 S_{u1} &= \begin{bmatrix} C(1)B_u(0) \\ C(2)[B_u(1) + A(1)B_u(0)] \\ \vdots \\ C(p)\sum_{k=0}^{p-1} \left[\left(\prod_{i=k+1}^{p-1} A(i) \right) B_u(k) \right] \end{bmatrix} \\
 S_u &= \begin{bmatrix} S_{u1} & \begin{matrix} 0 & \cdots & 0 \\ C(2)B_u(1) & \cdots & 0 \\ \vdots & & \end{matrix} \\ & \begin{matrix} C(p)\sum_{k=1}^{p-1} \left[\left(\prod_{i=k+1}^{p-1} A(i) \right) B_u(k) \right] & \cdots & \cdots & C(p)B_u(p-1) \end{matrix} \end{bmatrix} \\
 H_v &= \begin{bmatrix} C(1)B_v(0) & D_v(1) & 0 & \cdots & 0 \\ C(2)A(1)B_v(0) & C(2)B_v(1) & D_v(2) & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ C(p)\left(\prod_{i=1}^{p-1} A(i)\right)B_v(0) & \cdots & \cdots & C(p)B_v(p-1) & D_v(p) \end{bmatrix}
 \end{aligned}$$

where $\prod_{i=k_1}^{k_2} A(i) \triangleq A(k_2)A(k_2-1)\dots A(k_1)$ if $k_2 \geq k_1$, or I otherwise.

For more information on the prediction matrices for implicit MPC and adaptive MPC, see “QP Matrices” on page 2-8.

Time-Varying Nominal Conditions

Linear models are often obtained by linearizing nonlinear dynamics around time-varying nominal trajectories. For example, consider the following LTI model, obtained by linearizing a nonlinear system at the time-varying nominal offsets x_{off} , u_{off} , v_{off} , and y_{off} :

$$\begin{aligned}x(k+1) - \bar{x}_{off}(k+1) &= A(x(k) - \bar{x}_{off}(k)) + B_u(u(k) - \bar{u}_{off}(k)) + \\&\quad B_v(v(k) - \bar{v}_{off}(k)) + \Delta x_{off}(k) \\y(k) - \bar{y}_{off}(k) &= C(x(k) - \bar{x}_{off}(k)) + D_v(v(k) - \bar{v}_{off}(k))\end{aligned}$$

If we define

$$\begin{aligned}\overline{x_{off}} &\triangleq x(0), \quad \overline{u_{off}} \triangleq u(0) \\ \overline{v_{off}} &\triangleq v(0), \quad \overline{y_{off}} \triangleq y(0)\end{aligned}$$

as standard nominal values that remain constant over the prediction horizon, we can transform the LTI model into the following LTV model:

$$\begin{aligned}x(k+1) - \overline{x_{off}} &= A(x(k) - \overline{x_{off}}) + B_u(u(k) - \overline{u_{off}}) + B_v(v(k) - \overline{v_{off}}) + \bar{B}_v(k) \\y(k) - \overline{y_{off}} &= C(x(k) - \overline{x_{off}}) + D_v(v(k) - \overline{v_{off}}) + \bar{D}_v(k)\end{aligned}$$

where

$$\begin{aligned}\bar{B}_v(k) &\triangleq \Delta x_{off}(k) + A(\overline{x_{off}} - x_{off}(k)) + B_u(\overline{u_{off}} - u_{off}(k)) + \\&\quad B_v(\overline{v_{off}} - v_{off}(k)) + \Delta x_{off}(k+1) \\ \bar{D}_v(k) &\triangleq C(\overline{x_{off}} - x_{off}(k)) + D_v(\overline{v_{off}} - v_{off}(k)) - \overline{y_{off}}\end{aligned}$$

If the original linearized model is already LTV, the same transformation applies.

State Estimation

As with adaptive MPC, time-varying MPC uses a time-varying Kalman filter based on $A(0)$, $B(0)$, $C(0)$, and $D(0)$ from the initial prediction step; that is, the current time at which the state is estimated. For more information, see “State Estimation” on page 5-4.

See Also

`mpcmoveAdaptive`

More About

- “Adaptive MPC” on page 5-2
- “Optimization Problem” on page 2-2
- “Time-Varying MPC Control of a Time-Varying Plant” on page 5-39

Time-Varying MPC Control of a Time-Varying Plant

This example shows how the Model Predictive Control Toolbox™ can use time-varying prediction models to achieve better performance when controlling a time-varying plant.

The following MPC controllers are compared:

- 1** Linear MPC controller based on a time-invariant average model
- 2** Linear MPC controller based on a time-invariant model, which is updated at each time step.
- 3** Linear MPC controller based on a time-varying prediction model.

Time-Varying Linear Plant

In this example, the plant is a single-input-single-output 3rd order time-varying linear system with poles, zeros and gain that vary periodically with time.

$$G = \frac{5s + 5 + 2\cos(2.5t)}{s^3 + 3s^2 + 2s + 6 + \sin(5t)}$$

The plant poles move between being stable and unstable at run time, which leads to a challenging control problem.

Generate an array of plant models at $t = 0, 0.1, 0.2, \dots, 10$ seconds.

```
Models = tf;
ct = 1;
for t = 0:0.1:10
    Models(:,:,ct) = tf([5 5+2*cos(2.5*t)],[1 3 2 6+sin(5*t)]);
    ct = ct + 1;
end
```

Convert the models to state-space format and discretize them with a sample time of 0.1 second.

```
Ts = 0.1;
Models = ss(c2d(Models,Ts));
```

MPC Controller Design

The control objective is to track a step change in the reference signal. First, design an MPC controller for the average plant model. The controller sample time is 0.1 second.

```
sys = ss(c2d(tf([5 5],[1 3 2 6]),Ts)); % prediction model
```

```
p = 3; % prediction horizon
m = 3; % control horizon
mpcobj = mpc(sys,Ts,p,m);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Set hard constraints on the manipulated variable and specify tuning weights.

```
mpcobj.MV = struct( Min ,-2, Max ,2);
mpcobj.Weights = struct( MV ,0, MVRate ,0.01, Output ,1);
```

Set the initial plant states to zero.

```
x0 = zeros(size(sys.B));
```

Closed-Loop Simulation with Implicit MPC

Run a closed-loop simulation to examine whether the designed implicit MPC controller can achieve the control objective without updating the plant model used in prediction.

Set the simulation duration to 5 seconds.

```
Tstop = 5;
```

Use the `mpcmove` command in a loop to simulate the closed-loop response.

```
yyMPC = [];
uuMPC = [];
x = x0;
xmpc = mpcstate(mpcobj);
fprintf( Simulating MPC controller based on average LTI model.\n );
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyMPC = [yyMPC,y];
    % Compute and store the MPC optimal move.
    u = mpcmove(mpcobj,xmpc,y,1);
    uuMPC = [uuMPC,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end

-->Assuming output disturbance added to measured output channel #1 is integrated white
```

-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output.

Simulating MPC controller based on average LTI model.

Closed-Loop Simulation with Adaptive MPC

Run a second simulation to examine whether an adaptive MPC controller can achieve the control objective.

Use the `mpcmoveAdaptive` command in a loop to simulate the closed-loop response. Update the plant model for each control interval, and use the updated model to compute the optimal control moves. The `mpcmoveAdaptive` command uses the same prediction model across the prediction horizon.

```
yyAMPC = [];
uuAMPC = [];
x = x0;
xmpc = mpcstate(mpcobj);
nominal = mpcobj.Model.Nominal;
fprintf( Simulating MPC controller based on LTI model, updated at each time step t.\n );
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyAMPC = [yyAMPC, y];
    % Compute and store the MPC optimal move.
    u = mpcmoveAdaptive(mpcobj,xmpc,real_plant,nominal,y,1);
    uuAMPC = [uuAMPC,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end
```

Simulating MPC controller based on LTI model, updated at each time step t.

Closed-Loop Simulation with Time-Varying MPC

Run a third simulation to examine whether a time-varying MPC controller can achieve the control objective.

The controller updates the prediction model at each control interval and also uses time-varying models across the prediction horizon, which gives MPC controller the best knowledge of plant behavior in the future.

Use the `mpcmovAdaptive` command in a loop to simulate the closed-loop response. Specify an array of plant models rather than a single model. The controller uses each model in the array at a different prediction horizon step.

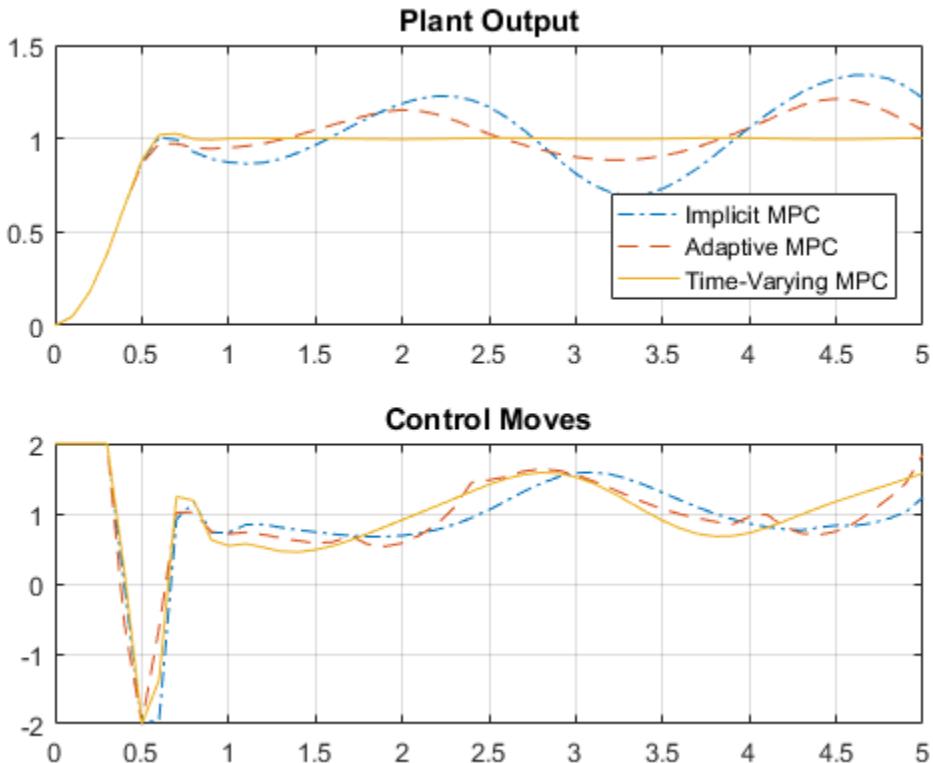
```
yyLTVMP = [];
uuLTVMP = [];
x = x0;
xmpc = mpccstate(mpcobj);
Nominals = repmat(nominal,3,1); % Nominal conditions are constant over the prediction
fprintf( Simulating MPC controller based on time-varying model, updated at each time s
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyLTVMP = [yyLTVMP, y];
    % Compute and store the MPC optimal move.
    u = mpcmovAdaptive(mpcobj,xmpc,Models(:,:,ct:ct+p),Nominals,y,1);
    uuLTVMP = [uuLTVMP,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end
```

Simulating MPC controller based on time-varying model, updated at each time step t.

Performance Comparison of MPC Controllers

Compare the closed-loop responses.

```
t = 0:Ts:Tstop;
figure
subplot(2,1,1);
plot(t,yyMPC, -. ,t,yyAMPC, -- ,t,yyLTVMP);
grid
legend( Implicit MPC , Adaptive MPC , Time-Varying MPC , Location , SouthEast )
title( Plant Output );
subplot(2,1,2);
plot(t,uuMPC, -. ,t,uuAMPC, -- ,t,uuLTVMP)
grid
title( Control Moves );
```

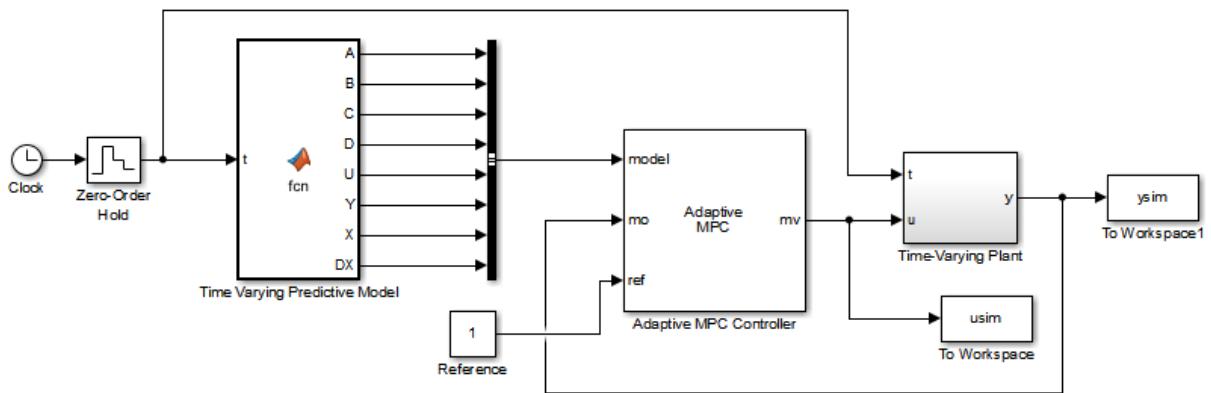


Only the time-varying MPC controller is able to bring the plant output close enough to the desired setpoint.

Closed-Loop Simulation of Time-Varying MPC in Simulink

To simulate time-varying MPC control in Simulink, pass the time-varying plant models to `model` input of the Adaptive MPC Controller block.

```
xmpc = mpcstate(mpcobj);
mdl = mpc_timevarying ;
open_system(mdl);
```



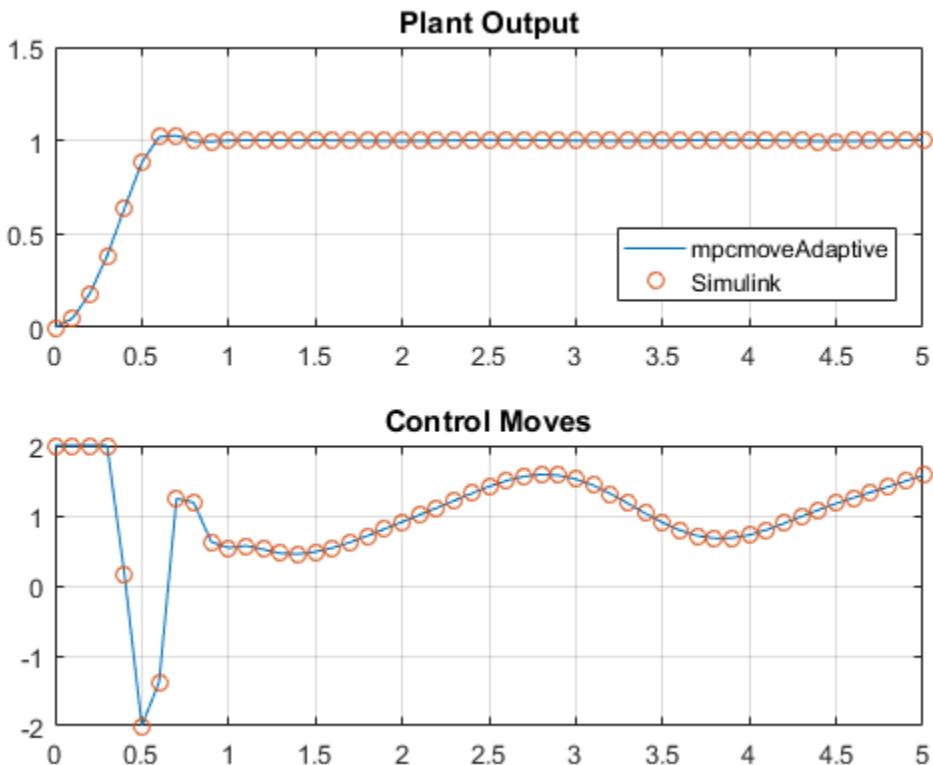
Run the simulation.

```
sim(mdl,Tstop);
fprintf( Simulating MPC controller based on LTV model in Simulink.\n );
```

Simulating MPC controller based on LTV model in Simulink.

Plot the MATLAB and Simulink time-varying simulation results.

```
figure
subplot(2,1,1)
plot(t,yyLTVMPc,t,ysim, o );
grid
legend( mpcmoveAdaptive , Simulink , Location , SouthEast )
title( Plant Output );
subplot(2,1,2)
plot(t,uuLTVMPc,t,usim, o )
grid
title( Control Moves );
```



The closed-loop responses in MATLAB and Simulink are identical.

```
bdclose(mdl);
```

See Also

`mpcmoveAdaptive`

More About

- “Time-Varying MPC” on page 5-34

Explicit MPC Design

- “Explicit MPC” on page 6-2
- “Design Workflow for Explicit MPC” on page 6-4
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-9
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-21
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30
- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-42

Explicit MPC

A traditional model predictive controller solves a quadratic program (QP) at each control interval to determine the optimal manipulated variable (MV) adjustments. These adjustments are the solution of the implicit nonlinear function $u=f(x)$.

The vector x contains the current controller state and other independent variables affecting the QP solution, such as the current output reference values. The Model Predictive Control Toolbox software imposes restrictions that force a unique QP solution.

Finding the optimal MV adjustments can be time consuming, and the required time can vary significantly from one control interval to the next. In applications that require a solution within a certain consistent time, which could be on the order of microseconds, the implicit MPC approach might be unsuitable.

As shown in “Optimization Problem” on page 2-2, if no QP inequality constraints are active for a given x vector, then the optimal MV adjustments become a linear function of x :

$$u = Fx + G.$$

where, F and G are constants. Similarly, if x remains in a region where a fixed subset of inequality constraints is active, the QP solution is also a linear function of x , but with different F and G constants.

Explicit MPC uses offline computations to determine all polyhedral regions where the optimal MV adjustments are a linear function of x , and the corresponding control-law constants. When the controller operates in real time, the explicit MPC controller performs the following steps at each control instant, k :

- 1 Estimate the controller state using available measurements, as in traditional MPC.
- 2 Form $x(k)$ using the estimated state and the current values of the other independent variables.
- 3 Identify the region in which $x(k)$ resides.
- 4 Looks up the predetermined F and G constants for this region.
- 5 Evaluate the linear function $u(k) = Fx(k) + G$.

You can establish a tight upper bound for the time required in each step. If the number of regions is not too large, the total computational time can be small. However, as the

number of regions increases, the time required in step 3 dominates. Also, the memory required to store all the linear control laws and polyhedral regions becomes excessive. The number of regions characterizing $u = f(x)$ depends primarily on the QP inequality constraints that could be active at the solution. If an explicit MPC controller has many constraints, and thus requires significant computational effort or memory, a traditional (implicit) implementation may be preferable.

Related Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-9
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-21
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

More About

- “Design Workflow for Explicit MPC” on page 6-4

Design Workflow for Explicit MPC

In this section...

- “Traditional (Implicit) MPC Design” on page 6-4
- “Explicit MPC Generation” on page 6-5
- “Explicit MPC Simplification” on page 6-6
- “Implementation” on page 6-6
- “Simulation” on page 6-7

To create an explicit MPC controller, you must first design a traditional (implicit) MPC controller. You then generate an explicit MPC controller based on the traditional controller design.

Traditional (Implicit) MPC Design

First design a traditional (implicit) MPC for your application and test it in simulations. Key considerations are as follows:

- The Model Predictive Control Toolbox software currently supports the following as independent variables for explicit MPC:
 - n_{xc} controller state variables (plant, disturbance, and measurement noise model states).
 - n_y (≥ 1) output reference values, where n_y is the number of plant output variables.
 - n_v (≥ 0) measured plant disturbance signals.

Thus, you must fix most MPC design parameters prior to determining an explicit MPC. Fixed parameters include prediction models (plant, disturbance and measurement noise), scale factors, horizons, penalty weights, manipulated variable targets, and constraint bounds.

For information about designing a traditional MPC controller, see “Controller Creation”.

For information about tuning traditional MPC controllers, see “Refinement”.

- Reference and measured disturbance previewing are not supported. At each control interval, the current n_y reference and n_v measured disturbance signals apply for the entire prediction horizon.

- To limit the number of regions needed by explicit MPC, include only essential constraints.
 - When including a constraint on a manipulated variable (MV) use a short control horizon or MV blocking. See “Choosing Sample Time and Horizons” on page 1-6.
 - Avoid constraints on plant outputs. If such a constraint is essential, consider imposing it for selected prediction horizon steps rather than the entire prediction horizon.
- Establish upper and lower bounds for each of the $n_x = n_{xc} + n_y + n_v$ independent variables. You might know some of these bounds a priori. However, you must run simulations that record at least the n_{xc} controller states as the system operates over the range of expected conditions. It is very important that you not underestimate this range, because the explicit MPC control function is not defined for independent variables outside the range.

For information about specifying bounds, see `generateExplicitRange`.

For information about simulating a traditional MPC controller, see “Simulation”.

Explicit MPC Generation

Given the constant MPC design parameters and the n_x upper and lower bounds on the control law's independent variables, i.e.,

$$x_l \leq x(k) \leq x_u,$$

the `generateExplicitMPC` command determines n_r regions. Each of these regions is defined by an inequality constraint and the corresponding control law constants:

$$\begin{aligned} H_i x(k) &\leq K_i, \quad i = 1, n_r \\ u(k) &= F_i x(k) + G_i, \quad i = 1, n_r. \end{aligned}$$

The Explicit MPC Controller object contains the constants H_i , K_i , F_i , and G_i for each region. The Explicit MPC Controller object also holds the original (implicit) design and independent variable bounds. Provided that $x(k)$ stays within the specified bounds and you retain all n_r regions, the explicit MPC object should provide the same optimal MV adjustments, $u(k)$, as the equivalent implicit MPC object.

For details about explicit MPC, see [1]. For details about how the explicit MPC controller is generated, see [2].

Explicit MPC Simplification

Even a relatively simple explicit MPC controller might need $n_r \gg 100$ to characterize the QP solution completely. If the number of regions is large, consider the following:

- Visualize the solution using the `plotSection` command.
- Use the `simplify` command to reduce the number of regions. In some cases, this can be done with no (or negligible) impact on control law optimality. For example, pairs of adjacent regions might employ essentially the same F_i and K_i constants. If so, and if the union of the two regions forms a convex set, they can be merged into a single region.

Alternatively, you can eliminate relatively small regions or retain selected regions only. If during operation the current $x(k)$ is not contained in any of the retained regions, the explicit MPC will return a suboptimal $u(k)$, as follows:

$$u(k) = F_j x(k) + G_j.$$

Here, j is the index of the region whose bounding constraint, $H_j x(k) \leq K_j$, is least violated.

Implementation

During operation, for a given $x(k)$, the explicit MPC controller performs the following steps:

- 1 Verifies that $x(k)$ satisfies the specified bounds, $x_l \leq x(k) \leq x_u$. If not, the controller returns an error status and sets $u(k) = u(k-1)$.
- 2 Beginning with region $i = 1$, tests the regions one by one to determine whether $x(k)$ belongs. If $H_i x(k) \leq K_i$, then $x(k)$ belongs to region i . If $x(k)$ belongs to region i , then the controller:
 - Obtains F_i and G_i from memory, and computes $u(k) = F_i x(k) + G_i$.
 - Signals successful completion, by returning a status code and the index i .
 - Returns without testing the remaining regions.

If $x(k)$ does not belong to region i , the controller:

- Computes the violation term v_i , which is the largest (positive) component of the vector $(H_i x(k) - K_i)$.
 - If v_i is the minimum violation for this $x(k)$, the controller sets $j = i$, and sets $v_{min} = v_i$.
 - The controller then increments i and tests the next region.
- 3** If all regions have been tested and $x(k)$ does not belong to any region (for example, due to a numerical precision issue), the controller:
- Obtains F_j and G_j from memory, and computes $u(k) = F_j x(k) + G_j$.
 - Sets status to indicate a suboptimal solution and returns.

Thus, the maximum computational time per control interval is that needed to test each region, computing the violation term in each case, and then calculating the suboptimal control adjustment.

Simulation

You can perform command-line simulations using the `sim` or `mpcmovExplicit` commands.

You can use the `Explicit MPC Controller` block to connect an explicit MPC to a plant modeled in Simulink.

References

- [1] A. Bemporad, M. Morari, V. Dua, and E.N. Pistikopoulos, “The explicit linear quadratic regulator for constrained systems,” *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.
- [2] A. Bemporad, “A multi-parametric quadratic programming algorithm with polyhedral computations based on nonnegative least squares,” 2014, Submitted for publication.

See Also

`Explicit MPC Controller` | `generateExplicitMPC` | `mpcmovExplicit`

Related Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-9
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-21
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

More About

- “Explicit MPC” on page 6-2

Explicit MPC Control of a Single-Input-Single-Output Plant

This example shows how to control a double integrator plant under input saturation in Simulink® using explicit MPC.

See also MPCDOUBLEINT.

Define Plant Model

The linear open-loop dynamic model is a double integrator:

```
plant = tf(1,[1 0 0]);
```

Design MPC Controller

Create the controller object with sampling period, prediction and control horizons:

```
Ts = 0.1;
p = 10;
m = 3;
mpcobj = mpc(plant, Ts, p, m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify actuator saturation limits as MV constraints.

```
mpcobj.MV = struct( Min ,-1, Max ,1);
```

Generate Explicit MPC Controller

Explicit MPC executes the equivalent explicit piecewise affine version of the MPC control law defined by the traditional MPC. To generate an Explicit MPC from a traditional MPC, you must specify range for each controller state, reference signal, manipulated variable and measured disturbance so that the multi-parametric quadratic programming problem is solved in the parameter space defined by these ranges.

Obtain a range structure for initialization

Use `generateExplicitRange` command to obtain a range structure where you can specify range for each parameter afterwards.

```
range = generateExplicitRange(mpcobj);  
-->Converting the "Model.Plant" property of "mpc" object to state-space.  
-->Converting model to discrete time.  
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Specify ranges for controller states

MPC controller states include states from plant model, disturbance model and noise model in that order. Setting the range of a state variable is sometimes difficult when the state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

```
range.State.Min(:) = [-10;-10];  
range.State.Max(:) = [10;10];
```

Specify ranges for reference signals

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate Explicit MPC must be at least as large as the practical range.

```
range.Reference.Min = -2;  
range.Reference.Max = 2;
```

Specify ranges for manipulated variables

If manipulated variables are constrained, the ranges used to generate Explicit MPC must be at least as large as these limits.

```
range.ManipulatedVariable.Min = -1.1;  
range.ManipulatedVariable.Max = 1.1;
```

Construct the Explicit MPC controller

Use `generateExplicitMPC` command to obtain the Explicit MPC controller with the parameter ranges previously specified.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range);  
display(mpcobjExplicit);
```

```
Regions found / unexplored:      19/      0
```

Explicit MPC Controller

```
Controller sample time: 0.1 (seconds)
```

```
Polyhedral regions: 19
```

```
Number of parameters: 4
```

```
Is solution simplified: No
```

```
State Estimation: Default Kalman gain
```

```
Type mpcobjExplicit.MPC for the original implicit MPC design.
```

```
Type mpcobjExplicit.Range for the valid range of parameters.
```

```
Type mpcobjExplicit.OptimizationOptions for the options used in multi-parametric QP
```

```
Type mpcobjExplicit.PiecewiseAffineSolution for regions and gain in each solution.
```

Use **simplify** command with the "exact" method to join pairs of regions whose corresponding gains are the same and whose union is a convex set. This practice can reduce memory footprint of the Explicit MPC controller without sacrifice any performance.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, exact );
display(mpcobjExplicitSimplified);
```

```
Regions to analyze:      15/      15
```

Explicit MPC Controller

```
Controller sample time: 0.1 (seconds)
```

```
Polyhedral regions: 15
```

```
Number of parameters: 4
```

```
Is solution simplified: Yes
```

```
State Estimation: Default Kalman gain
```

```
Type mpcobjExplicitSimplified.MPC for the original implicit MPC design.
```

```
Type mpcobjExplicitSimplified.Range for the valid range of parameters.
```

```
Type mpcobjExplicitSimplified.OptimizationOptions for the options used in multi-parametric QP
```

```
Type mpcobjExplicitSimplified.PiecewiseAffineSolution for regions and gain in each solution.
```

The number of piecewise affine region has been reduced.

Plot Piecewise Affine Partition

You can review any 2-D section of the piecewise affine partition defined by the Explicit MPC control law.

Obtain a plot parameter structure for initialization

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicitSimplified);
```

Specify parameters for a 2-D plot

In this example, you plot the 1th state variable vs. the 2nd state variable. All the other parameters must be fixed at a value within its range.

```
params.State.Index = [];
params.State.Value = [];
```

Fix other reference signals

```
params.Reference.Index = 1;
params.Reference.Value = 0;
```

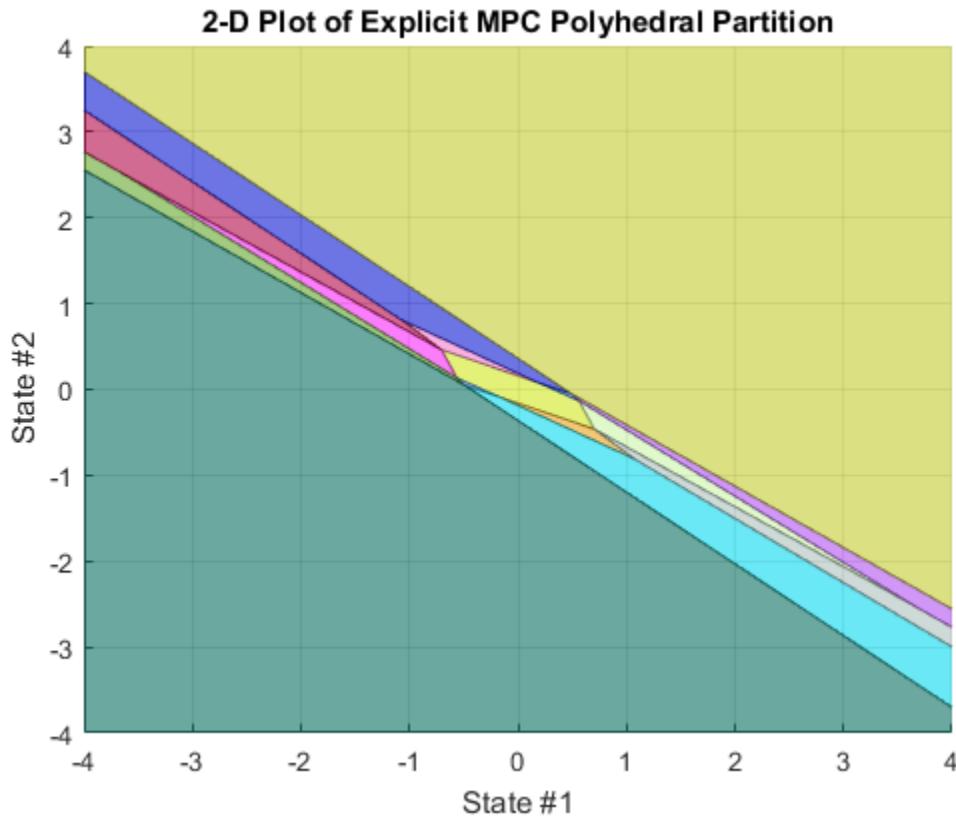
Fix manipulated variables

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Plot the 2-D section

Use `plotSection` command to plot the 2-D section defined previously.

```
plotSection(mpcobjExplicitSimplified, params);
axis([-4 4 -4 4]);
grid
xlabel( State #1 );
ylabel( State #2 );
```



Simulate Using MPCMOVE Command

Compare closed-loop simulation between tradition MPC (as referred as Implicit MPC) and Explicit MPC using `mpcmove` and `mpcmoveExplicit` commands respectively.

Prepare to store the closed-loop MPC responses.

```
Tf = round(5/Ts);
YY = zeros(Tf,1);
YYExplicit = zeros(Tf,1);
UU = zeros(Tf,1);
UUExplicit = zeros(Tf,1);
```

Prepare the real plant used in simulation

```
sys = c2d(ss(plant),Ts);
xsys = [0;0];
xsysExplicit = xsys;
```

Use MPCSTATE object to specify the initial states for both controllers

```
xmpc = mpcstate(mpcobj);
xmpcExplicit = mpcstate(mpcobjExplicitSimplified);
```

Simulate closed-loop response in each iteration.

```
for t = 0:Tf
    % update plant measurement
    ysys = sys.C*xsys;
    ysysExplicit = sys.C*xsysExplicit;
    % compute traditional MPC action
    u = mpcmove(mpcobj,xmpc,ysys,1);
    % compute Explicit MPC action
    uExplicit = mpcmoveExplicit(mpcobjExplicit,xmpcExplicit,ysysExplicit,1);
    % store signals
    YY(t+1)=ysys;
    YYExplicit(t+1)=ysysExplicit;
    UU(t+1)=u;
    UUEExplicit(t+1)=uExplicit;
    % update plant state
    xsys = sys.A*xsys + sys.B*u;
    xsysExplicit = sys.A*xsysExplicit + sys.B*uExplicit;
end
fprintf( \nDifference between traditional and Explicit MPC responses using MPCMOVE command is %f
```

Difference between traditional and Explicit MPC responses using MPCMOVE command is 4.08e-005

Simulate Using SIM Command

Compare closed-loop simulation between tradition MPC and Explicit MPC using **sim** commands respectively.

```
Tf = 5/Ts; % simulation iterations
[y1,t1,u1] = sim(mpcobj,Tf,1); % simulation with tradition MPC
[y2,t2,u2] = sim(mpcobjExplicitSimplified,Tf,1); % simulation with Explicit MPC

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
```

```
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.  
-->Converting the "Model.Plant" property of "mpc" object to state-space.  
-->Converting model to discrete time.  
    Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.  
-->Converting the "Model.Plant" property of "mpc" object to state-space.  
-->Converting model to discrete time.  
    Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

The simulation results are identical.

```
fprintf( \nDifference between traditional and Explicit MPC responses using SIM command
```

Difference between traditional and Explicit MPC responses using SIM command is 4.08204e-005

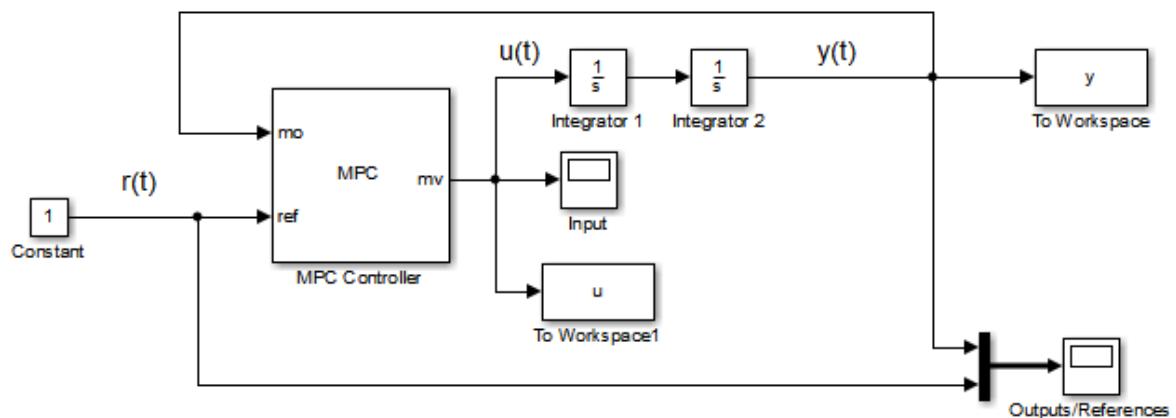
Simulate Using Simulink

To run this example, Simulink® is required.

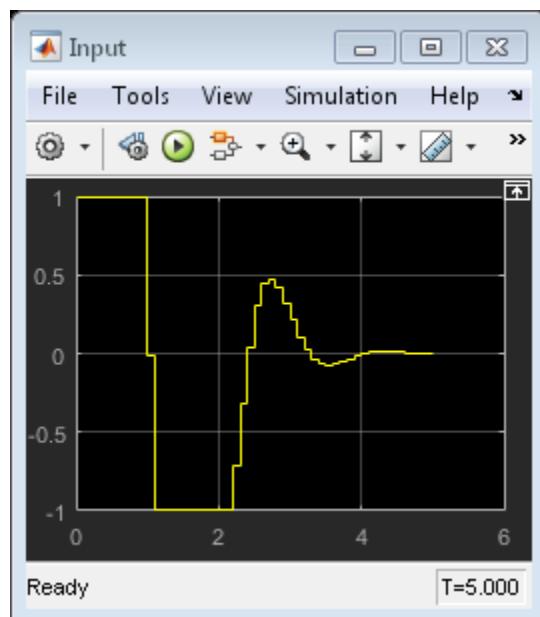
```
if ~mpcchecktoolboxinstalled( simulink )  
    disp( Simulink(R) is required to run this example. )  
    return  
end
```

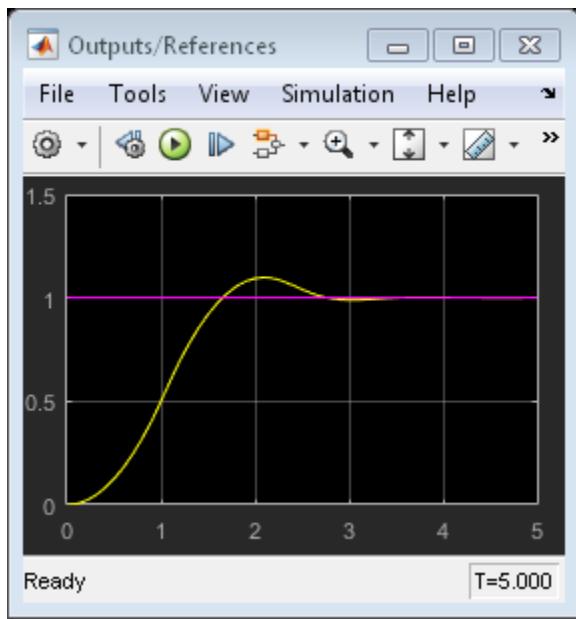
Simulate with traditional MPC controller in Simulink. Controller "mpcobj" is specified in the block dialog.

```
mdl = mpc_doubleint ;  
open_system(mdl);  
sim(mdl);
```



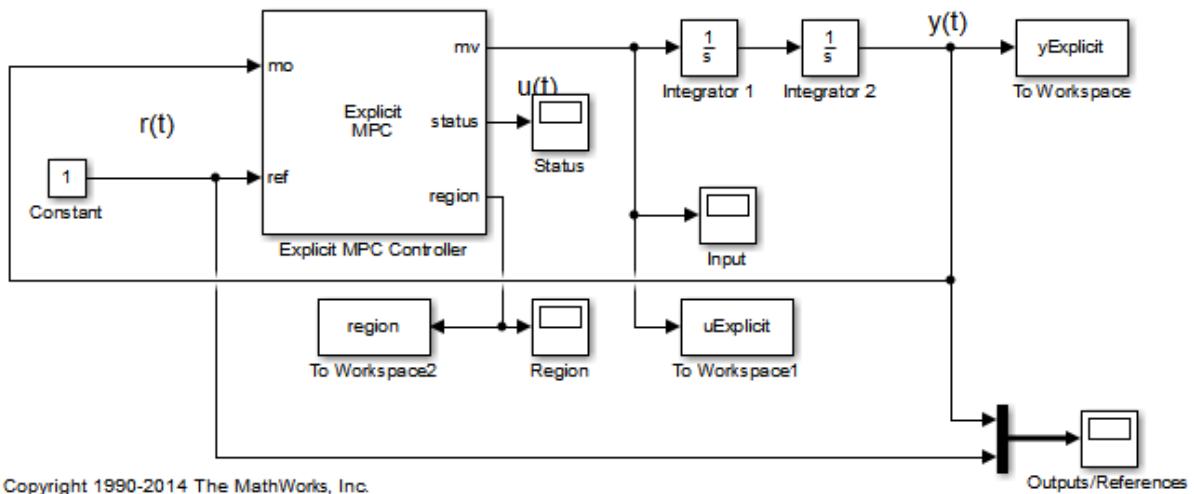
Copyright 1990-2014 The MathWorks, Inc.



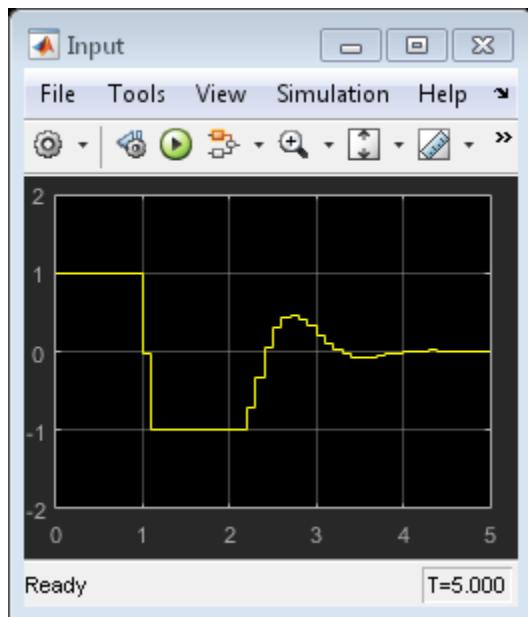


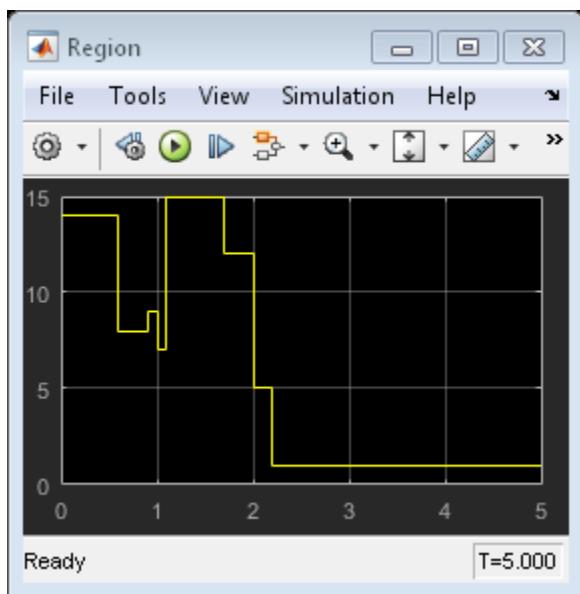
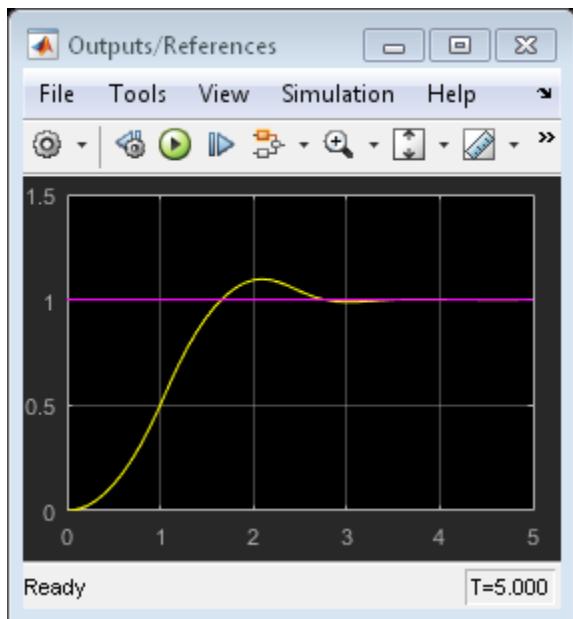
Simulate with Explicit MPC controller in Simulink. Controller "mpcobjExplicitSimplified" is specified in the block dialog.

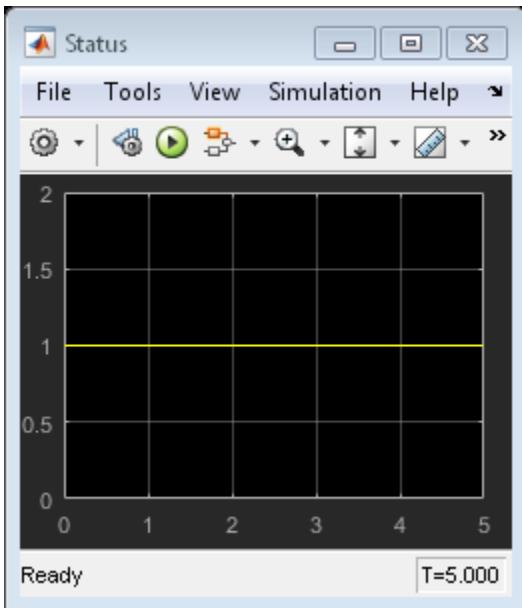
```
mdlExplicit = 'empc_doubleint';
open_system(mdlExplicit);
sim(mdlExplicit);
```



Copyright 1990-2014 The MathWorks, Inc.







The closed-loop responses are identical.

```
fprintf( \nDifference between traditional and Explicit MPC responses in Simulink is %g
```

```
Difference between traditional and Explicit MPC responses in Simulink is 3.98624e-13
```

```
bdclose(mdl)  
bdclose(mdlExplicit)
```

Related Examples

- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-21
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

More About

- “Explicit MPC” on page 6-2

Explicit MPC Control of an Aircraft with Unstable Poles

This example shows how to control an unstable aircraft with saturating actuators using Explicit MPC.

Reference:

- [1] P. Kapasouris, M. Athans and G. Stein, "Design of feedback control systems for unstable plants with saturating actuators", Proc. IFAC Symp. on Nonlinear Control System Design, Pergamon Press, pp.302--307, 1990
- [2] A. Bemporad, A. Casavola, and E. Mosca, "Nonlinear control of constrained linear systems via predictive reference management", IEEE® Trans. Automatic Control, vol. AC-42, no. 3, pp. 340-349, 1997.

See also MPCAIRCRAFT.

Define Aircraft Model

The linear open-loop dynamic model is as follows:

```
A = [-0.0151 -60.5651 0 -32.174;
      -0.0001 -1.3411 0.9929 0;
      0.00018 43.2541 -0.86939 0;
      0         0         1         0];
B = [-2.516 -13.136;
      -0.1689 -0.2514;
      -17.251 -1.5766;
      0         0];
C = [0 1 0 0;
      0 0 0 1];
D = [0 0;
      0 0];
plant = ss(A,B,C,D);
x0 = zeros(4,1);
```

The manipulated variables are the elevator and flaperon angles, the attack and pitch angles are measured outputs to be regulated.

The open-loop response of the system is unstable.

```
pole(plant)
```

```
ans =  
-7.6636 + 0.0000i  
5.4530 + 0.0000i  
-0.0075 + 0.0556i  
-0.0075 - 0.0556i
```

Design MPC Controller

To obtain an Explicit MPC controller, you must first design a traditional MPC (also referred as Implicit MPC) that is able to achieves your control objectives.

```
% *MV Constraints*
```

Both manipulated variables are constrained between +/- 25 degrees. Since the plant inputs and outputs are of different orders of magnitude, you also use scale factors to faciliate MPC tuning. Typical choices of scale factor are the upper/lower limit or the operating range.

```
MV = struct( Min ,{-25,-25} , Max ,{25,25} , ScaleFactor ,{50,50});
```

OV Constraints

Both plant outputs have constraints to limit undershoots at the first prediction horizon. You also specify scale factors for outputs.

```
OV = struct( Min ,{[-0.5;-Inf],[-100;-Inf]}, Max ,{[0.5;Inf],[100;Inf]}, ScaleFactor ,
```

Weights

The control task is to get zero offset for piecewise-constant references, while avoiding instability due to input saturation. Because both MV and OV variables are already scaled in MPC controller, MPC weights are dimensionless and applied to the scaled MV and OV values. In this example, you penalize the two outputs equally with the same OV weights.

```
Weights = struct( MV ,[0 0] , MVRate ,[0.1 0.1] , OV ,[10 10]);
```

Construct the traditional MPC controller

Create an MPC controller with plant model, sample time and horizons.

```
Ts = 0.05; % Sampling time  
p = 10; % Prediction horizon  
m = 2; % Control horizon  
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Generate Explicit MPC Controller

Explicit MPC executes the equivalent explicit piecewise affine version of the MPC control law defined by the traditional MPC. To generate an Explicit MPC from a traditional MPC, you must specify range for each controller state, reference signal, manipulated variable and measured disturbance so that the multi-parametric quadratic programming problem is solved in the parameter space defined by these ranges.

Obtain a range structure for initialization

Use `generateExplicitRange` command to obtain a range structure where you can specify range for each parameter afterwards.

```
range = generateExplicitRange(mpcobj);
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Specify ranges for controller states

MPC controller states include states from plant model, disturbance model and noise model in that order. Setting the range of a state variable is sometimes difficult when the state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

```
range.State.Min(:) = -10000;
range.State.Max(:) = 10000;
```

Specify ranges for reference signals

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate Explicit MPC must be at least as large as the practical range.

```
range.Reference.Min = [-1;-11];
range.Reference.Max = [1;11];
```

Specify ranges for manipulated variables

If manipulated variables are constrained, the ranges used to generate Explicit MPC must be at least as large as these limits.

```
range.ManipulatedVariable.Min = [MV(1).Min; MV(2).Min] - 1;
range.ManipulatedVariable.Max = [MV(1).Max; MV(2).Max] + 1;
```

Construct the Explicit MPC controller

Use `generateExplicitMPC` command to obtain the Explicit MPC controller with the parameter ranges previously specified.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range);
display(mpcobjExplicit);
```

```
Regions found / unexplored:      483/      0
```

Explicit MPC Controller

```
-----
```

```
Controller sample time:    0.05 (seconds)
```

```
Polyhedral regions:        483
```

```
Number of parameters:     10
```

```
Is solution simplified:   No
```

```
State Estimation:          Default Kalman gain
```

```
-----
```

```
Type mpcobjExplicit.MPC for the original implicit MPC design.
```

```
Type mpcobjExplicit.Range for the valid range of parameters.
```

```
Type mpcobjExplicit.OptimizationOptions for the options used in multi-parametric QP
```

```
Type mpcobjExplicit.PiecewiseAffineSolution for regions and gain in each solution.
```

Use `simplify` command with the "exact" method to join pairs of regions whose corresponding gains are the same and whose union is a convex set. This practice can reduce memory footprint of the Explicit MPC controller without sacrifice any performance.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, exact );
display(mpcobjExplicitSimplified);
```

```
Regions to analyze:      471/      471
```

Explicit MPC Controller

```
-----
```

```
Controller sample time:    0.05 (seconds)
```

```

Polyhedral regions:      471
Number of parameters:   10
Is solution simplified: Yes
State Estimation:       Default Kalman gain
-----
Type mpcobjExplicitSimplified.MPC for the original implicit MPC design.
Type mpcobjExplicitSimplified.Range for the valid range of parameters.
Type mpcobjExplicitSimplified.OptimizationOptions for the options used in multi-param
Type mpcobjExplicitSimplified.PiecewiseAffineSolution for regions and gain in each so

```

The number of piecewise affine region has been reduced.

Plot Piecewise Affine Partition

You can review any 2-D section of the piecewise affine partition defined by the Explicit MPC control law.

Obtain a plot parameter structure for initialization

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicitSimplified);
```

Specify parameters for a 2-D plot

In this example, you plot the pitch angle (the 4th state variable) vs. its reference (the 2nd reference signal). All the other parameters must be fixed at a value within its range.

Fix other state variables

```
params.State.Index = [1 2 3 5 6];
params.State.Value = [0 0 0 0 0];
```

Fix other reference signals

```
params.Reference.Index = 1;
params.Reference.Value = 0;
```

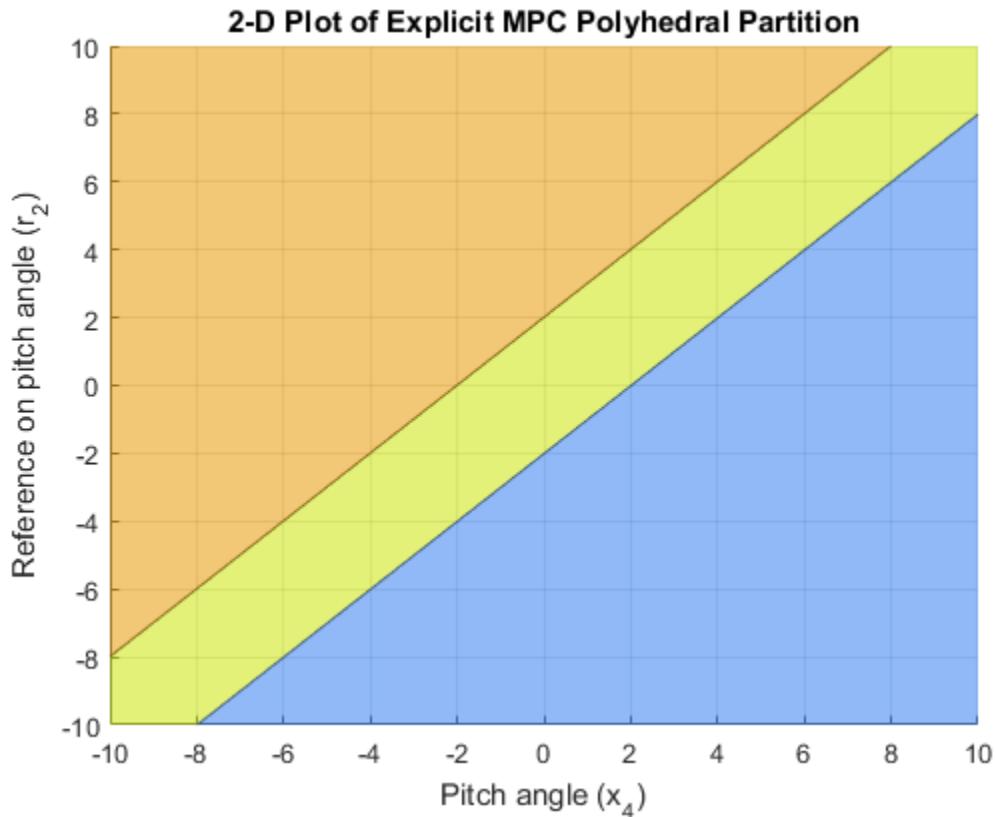
Fix manipulated variables

```
params.ManipulatedVariable.Index = [1 2];
params.ManipulatedVariable.Value = [0 0];
```

Plot the 2-D section

Use `plotSection` command to plot the 2-D section defined previously.

```
plotSection(mpcoObjExplicitSimplified, params);
axis([-10 10 -10 10]);
grid;
xlabel( Pitch angle ( $x_4$ ) );
ylabel( Reference on pitch angle ( $r_2$ ) );
```



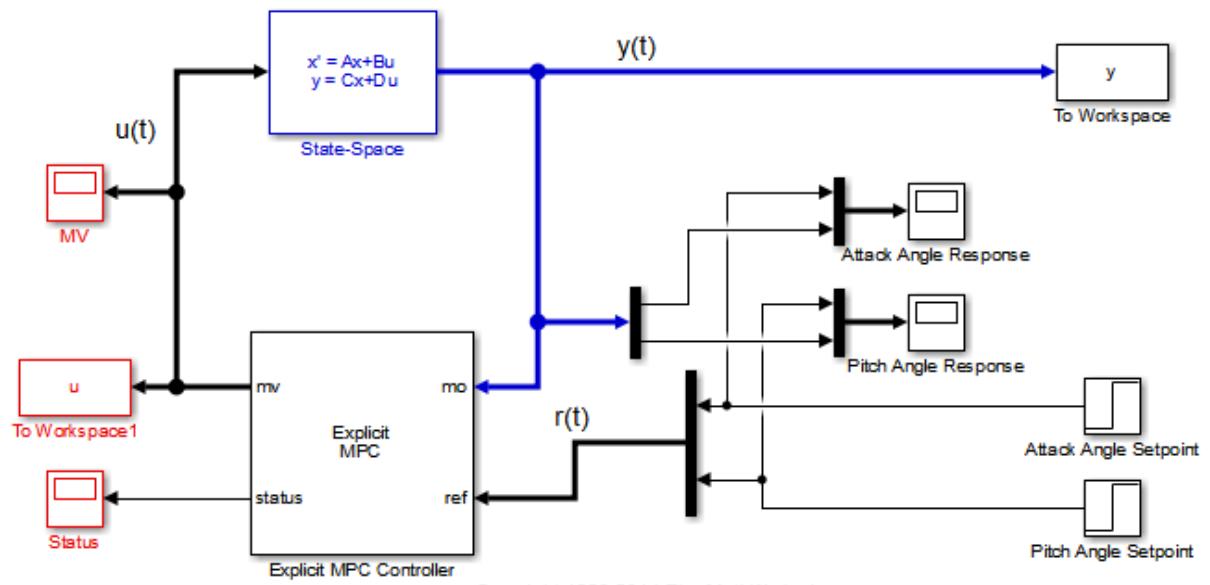
Simulate Using Simulink

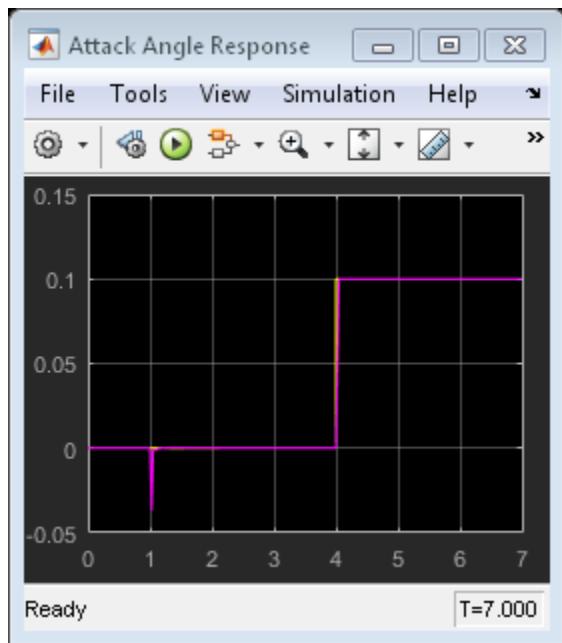
To run this example, Simulink® is required.

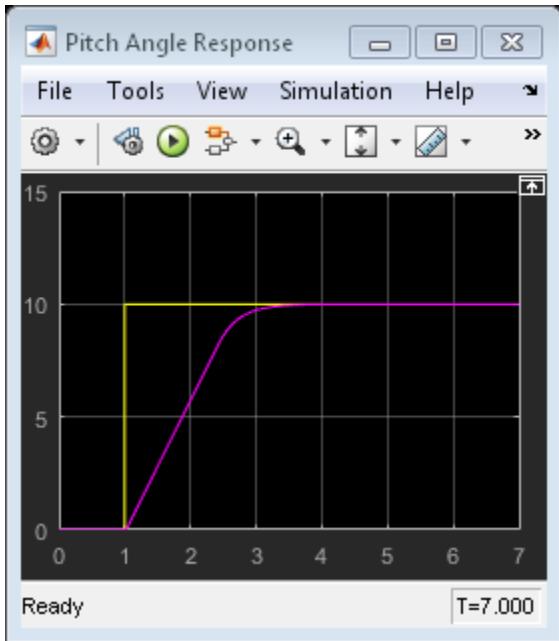
```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end
```

Simulate closed-loop control of the linear plant model in Simulink, using the Explicit MPC Controller block. Controller "mpcobjExplicitSimplified" is specified in the block dialog.

```
mdl = 'empc_aircraft';
open_system(mdl)
sim(mdl)
```







The closed-loop response is identical to the traditional MPC controller designed in the "mpcaircraft" example.

```
bdclose(mdl)
```

Related Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-9
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

More About

- “Explicit MPC” on page 6-2

Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output

This example shows how to use Explicit MPC to control DC servomechanism under voltage and shaft torque constraints.

Reference

[1] A. Bemporad and E. Mosca, "Fulfilling hard constraints in uncertain linear systems by reference managing," *Automatica*, vol. 34, no. 4, pp. 451-461, 1998.

See also MPCMOTOR.

Define DC-Servo Motor Model

The linear open-loop dynamic model is defined in "plant". Variable "tau" is the maximum admissible torque to be used as an output constraint.

```
[plant, tau] = mpcmotormodel;
```

Design MPC Controller

Specify input and output signal types for the MPC controller. The second output, torque, is unmeasurable.

```
plant = setmpcsignals(plant, MV ,1, MO ,1, UO ,2);
```

MV Constraints

The manipulated variable is constrained between +/- 220 volts. Since the plant inputs and outputs are of different orders of magnitude, you also use scale factors to facilitate MPC tuning. Typical choices of scale factor are the upper/lower limit or the operating range.

```
MV = struct( Min ,-220, Max ,220, ScaleFactor ,440);
```

OV Constraints

Torque constraints are only imposed during the first three prediction steps to limit the complexity of the explicit MPC design.

```
OV = struct( Min ,{Inf, [-tau;-tau;-tau;-Inf]}, Max ,{Inf, [tau;tau;tau;Inf]}, ScaleFa
```

Weights

The control task is to get zero tracking offset for the angular position. Since you only have one manipulated variable, the shaft torque is allowed to float within its constraint by setting its weight to zero.

```
Weights = struct( MV ,0, MVRate ,0.1, OV ,[0.1 0]);
```

Construct MPC controller

Create an MPC controller with plant model, sample time and horizons.

```
Ts = 0.1; % Sampling time
p = 10; % Prediction horizon
m = 2; % Control horizon
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Generate Explicit MPC Controller

Explicit MPC executes the equivalent explicit piecewise affine version of the MPC control law defined by the traditional MPC. To generate an Explicit MPC from a traditional MPC, you must specify the range for each controller state, reference signal, manipulated variable and measured disturbance so that the multi-parametric quadratic programming problem is solved in the parameter sets defined by these ranges.

Obtain a range structure for initialization

Use `generateExplicitRange` command to obtain a range structure where you can specify the range for each parameter afterwards.

```
range = generateExplicitRange(mpcobj);

-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on ea
```

Specify ranges for controller states

MPC controller states include states from plant model, disturbance model and noise model in that order. Setting the range of a state variable is sometimes difficult when the

state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

```
range.State.Min(:) = -1000;  
range.State.Max(:) = 1000;
```

Specify ranges for reference signals

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate Explicit MPC must be at least as large as the practical range. Note that the range for torque reference is fixed at 0 because it has zero weight.

```
range.Reference.Min = [-5;0];  
range.Reference.Max = [5;0];
```

Specify ranges for manipulated variables

If manipulated variables are constrained, the ranges used to generate Explicit MPC must be at least as large as these limits.

```
range.ManipulatedVariable.Min = MV.Min - 1;  
range.ManipulatedVariable.Max = MV.Max + 1;
```

Construct the Explicit MPC controller

Use `generateExplicitMPC` command to obtain the Explicit MPC controller with the parameter ranges previously specified.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range);  
display(mpcobjExplicit);
```

```
Regions found / unexplored:      75 /      0
```

```
Explicit MPC Controller
```

```
-----  
Controller sample time:    0.1 (seconds)  
Polyhedral regions:        75  
Number of parameters:     6
```

```

Is solution simplified:      No
State Estimation:          Default Kalman gain
-----
Type mpcobjExplicit.MPC for the original implicit MPC design.
Type mpcobjExplicit.Range for the valid range of parameters.
Type mpcobjExplicit.OptimizationOptions for the options used in multi-parametric QP
Type mpcobjExplicit.PiecewiseAffineSolution for regions and gain in each solution.

```

Plot Piecewise Affine Partition

You can review any 2-D section of the piecewise affine partition defined by the Explicit MPC control law.

Obtain a plot parameter structure for initialization

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicit);
```

Specify parameters for a 2-D plot

In this example, you plot the 1th state variable vs. the 2nd state variable. All the other parameters must be fixed at a value within its range.

Fix other state variables

```
params.State.Index = [3 4];
params.State.Value = [0 0];
```

Fix reference signals

```
params.Reference.Index = [1 2];
params.Reference.Value = [pi 0];
```

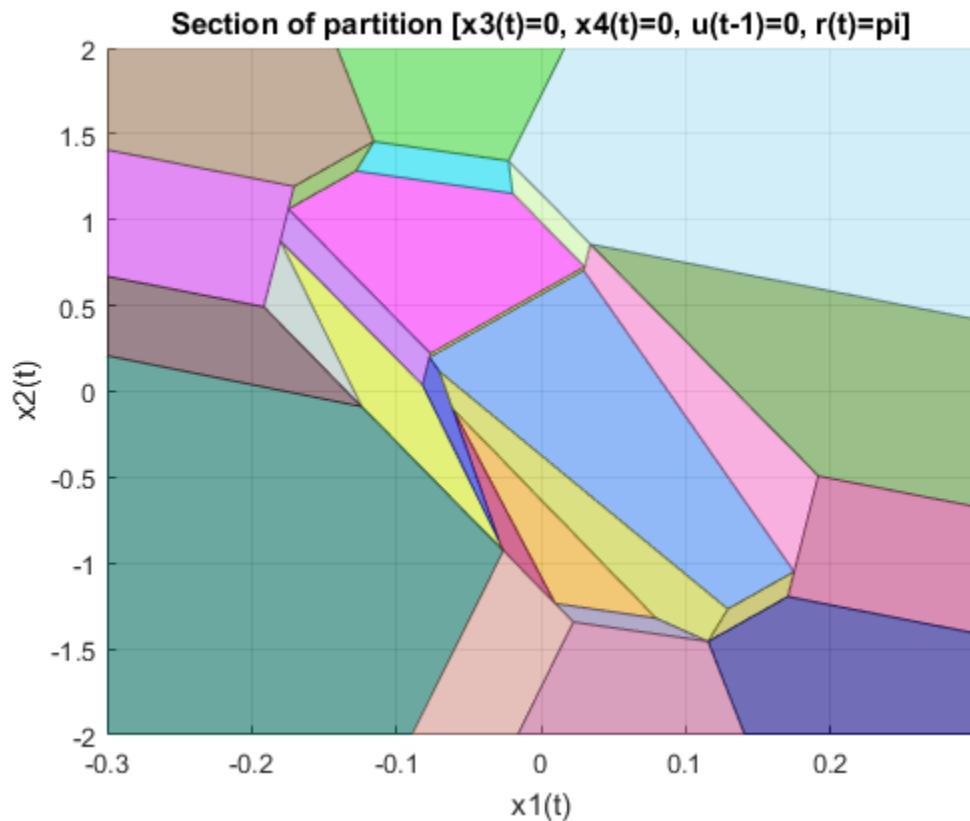
Fix manipulated variables

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Plot the 2-D section

Use `plotSection` command to plot the 2-D section defined previously.

```
plotSection(mpcoobjExplicit, params);
axis([- .3 .3 -2 2]);
grid
title( Section of partition [x3(t)=0, x4(t)=0, u(t-1)=0, r(t)=pi] )
xlabel( x1(t) );
ylabel( x2(t) );
```



Simulate Using SIM Command

Compare closed-loop simulation between traditional MPC (as referred as Implicit MPC) and Explicit MPC

```
Tstop = 8; % seconds
```

```

Tf = round(Tstop/Ts);           % simulation iterations
r = [pi 0];                   % reference signal
[y1,t1,u1] = sim(mpcobj,Tf,r); % simulation with traditional MPC
[y2,t2,u2] = sim(mpcobjExplicit,Tf,r); % simulation with Explicit MPC

-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

```

The simulation results are identical.

```

fprintf( SIM command: Difference between QP-based and Explicit MPC trajectories = %g\n
SIM command: Difference between QP-based and Explicit MPC trajectories = 4.82041e-12

```

Simulate Using Simulink

To run this example, Simulink® is required.

```

if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end

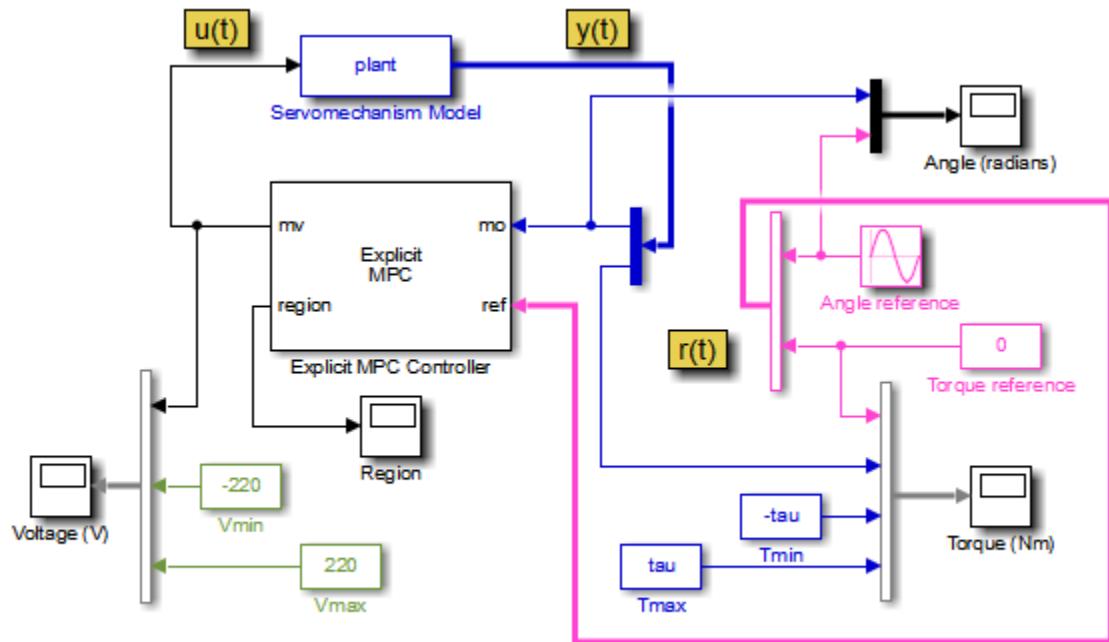
```

Simulate closed-loop control of the linear plant model in Simulink, using the Explicit MPC Controller block. Controller "mpcobjExplicit" is specified in the block dialog.

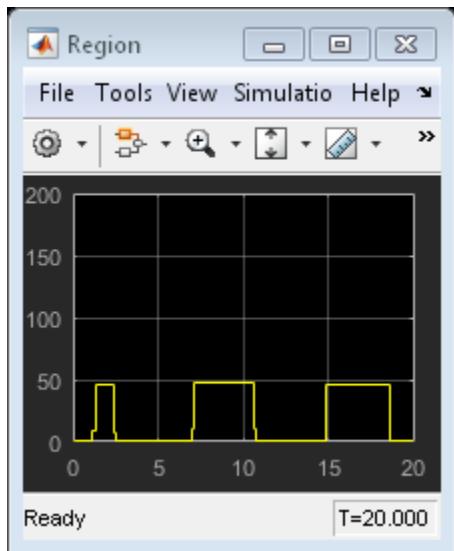
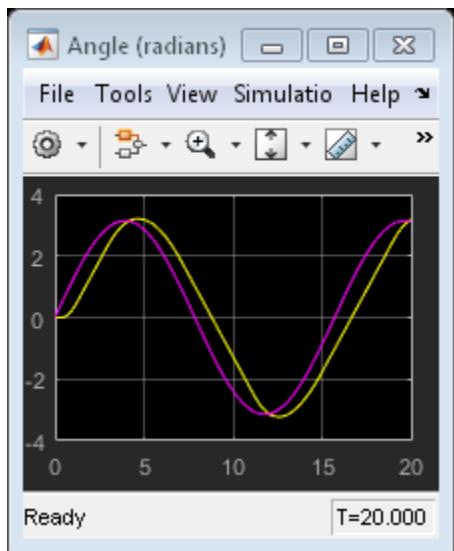
```

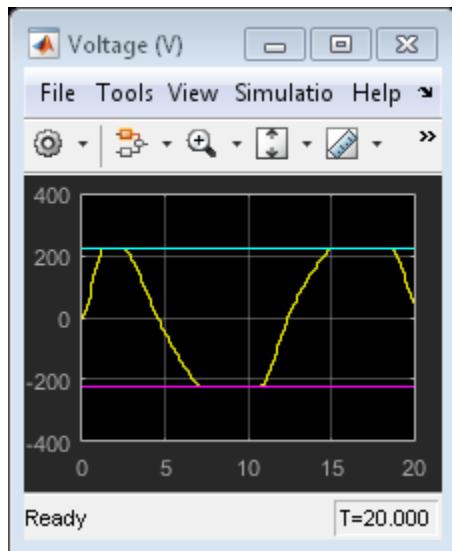
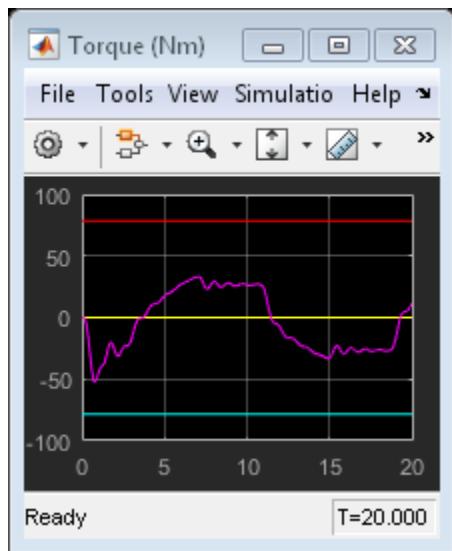
mdl = empc_motor ;
open_system(mdl)
sim(mdl);

```



Copyright 1990-2014 The MathWorks, Inc.





The closed-loop response is identical to the traditional MPC controller designed in the "mpcmotor" example.

Control Using Sub-optimal Explicit MPC

To reduce the memory footprint, you can use `simplify` command to reduce the number of piecewise affine solution regions. For example, you can remove regions whose Chebychev radius is smaller than .08. However, the price you pay is that the controller performance now becomes sub-optimal.

Use `simplify` command to generate Explicit MPC with sub-optimal solutions.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, radius, 0.08);
disp(mpcobjExplicitSimplified);
```

Regions to analyze: 75/ 75 --> 37 regions deleted.

explicitMPC with properties:

```
MPC: [1x1 mpc]
Range: [1x1 struct]
OptimizationOptions: [1x1 struct]
PiecewiseAffineSolution: [1x38 struct]
IsSimplified: 1
```

The number of piecewise affine regions has been reduced.

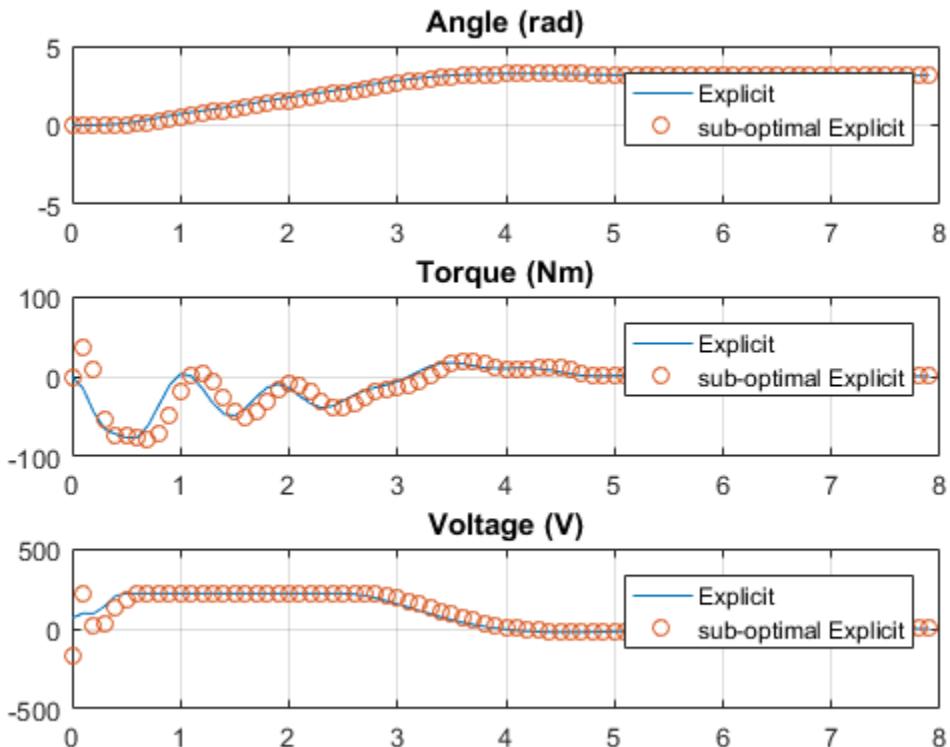
Compare closed-loop simulation between sub-optimal Explicit MPC and Explicit MPC.

```
[y3,t3,u3] = sim(mpcobjExplicitSimplified, Tf, r);
-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
-->Converting model to discrete time.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

The simulation results are not the same.

```
fprintf( SIM command: Difference between exact and suboptimal MPC trajectories = %g\n
SIM command: Difference between exact and suboptimal MPC trajectories = 439.399
Plot results.
figure;
```

```
subplot(3,1,1)
plot(t1,y1(:,1),t3,y3(:,1), o );
grid
title( Angle (rad) )
legend( Explicit , sub-optimal Explicit )
subplot(3,1,2)
plot(t1,y1(:,2),t3,y3(:,2), o );
grid
title( Torque (Nm) )
legend( Explicit , sub-optimal Explicit )
subplot(3,1,3)
plot(t1,u1,t3,u3, o );
grid
title( Voltage (V) )
legend( Explicit , sub-optimal Explicit )
```



The simulation result with the sub-optimal Explicit MPC is slightly worse.

```
bdclose(mdl)
```

Related Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-9
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-21

More About

- “Explicit MPC” on page 6-2

Explicit MPC Control of an Inverted Pendulum on a Cart

This example uses an explicit model predictive controller (explicit MPC) to control an inverted pendulum on a cart.

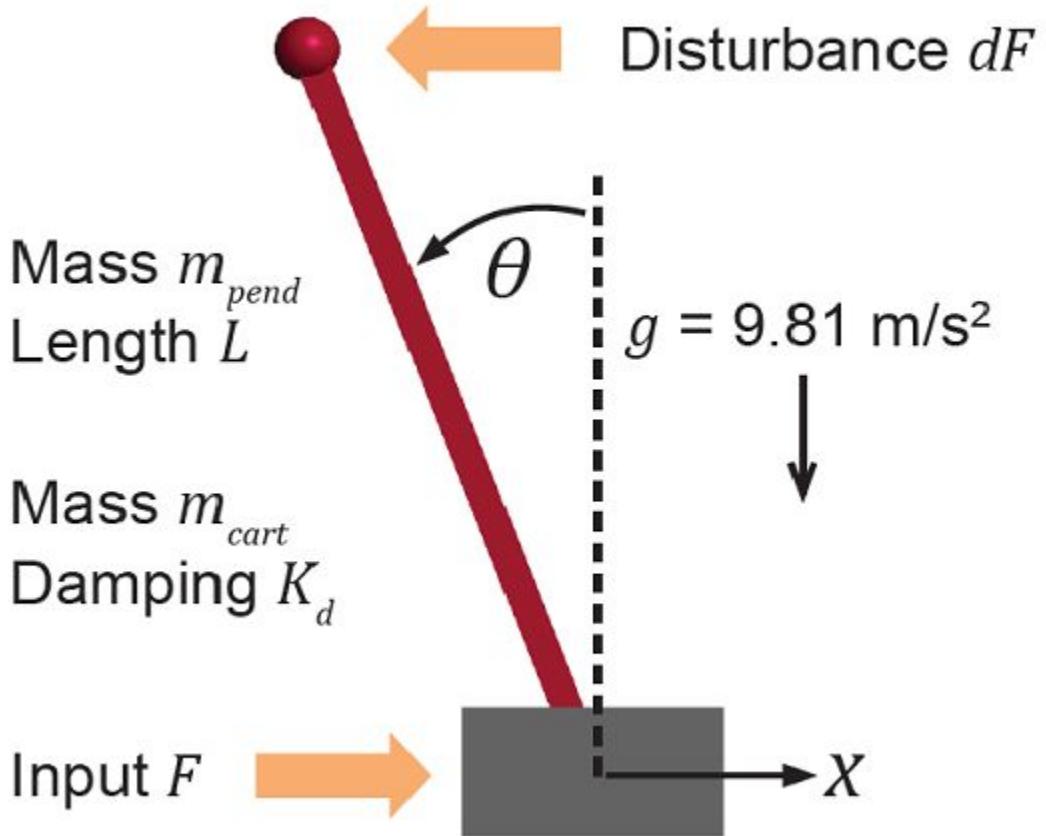
Product Requirement

This example requires Simulink Control Design™ software to define the MPC structure by linearizing a nonlinear Simulink model.

```
if ~mpcchecktoolboxinstalled( 'slcontrol' )
    disp( 'Simulink Control Design(R) is required to run this example.' )
    return
end
```

Pendulum/Cart Assembly

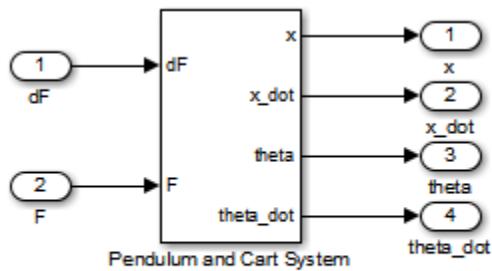
The plant for this example is the following cart/pendulum assembly, where x is the cart position and θ is the pendulum angle.



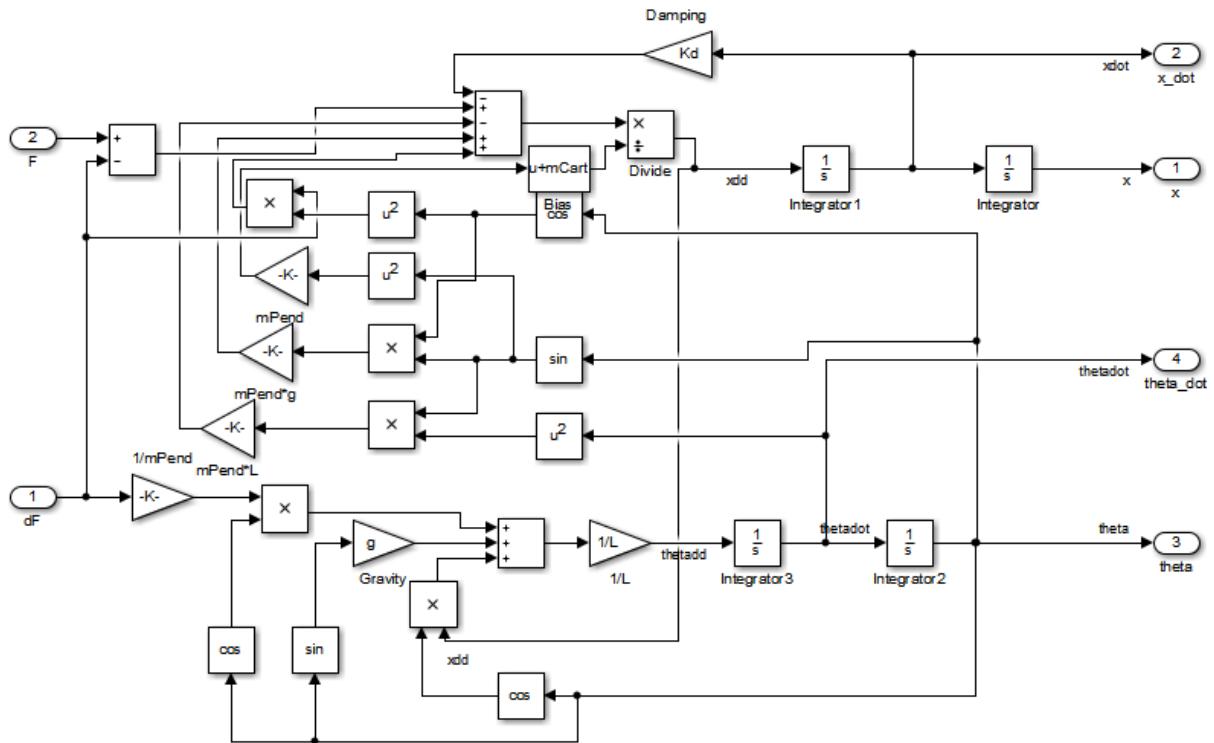
This system is controlled by exerting a variable force F on the cart. The controller needs to keep the pendulum upright while moving the cart to a new position or when the pendulum is nudged forward by an impulse disturbance dF applied at the upper end of the inverted pendulum.

This plant is modeled in Simulink with commonly used blocks.

```
mdlPlant = mpc_pendcartPlant ;
load_system(mdlPlant);
open_system([mdlPlant /Pendulum and Cart System ], force );
```



Copyright 1990-2015 The MathWorks, Inc.



Control Objectives

Assume the following initial conditions for the cart/pendulum assembly:

- The cart is stationary at $x = 0$.
- The inverted pendulum is stationary at the upright position $\theta = 0$.

The control objectives are:

- Cart can be moved to a new position between -10 and 10 with a step setpoint change.
- When tracking such a setpoint change, the rise time should be less than 4 seconds (for performance) and the overshoot should be less than 5 percent (for robustness).
- When an impulse disturbance of magnitude of 2 is applied to the pendulum, the cart should return to its original position with a maximum displacement of 1. The pendulum should also return to the upright position with a peak angle displacement of 15 degrees (0.26 radian).

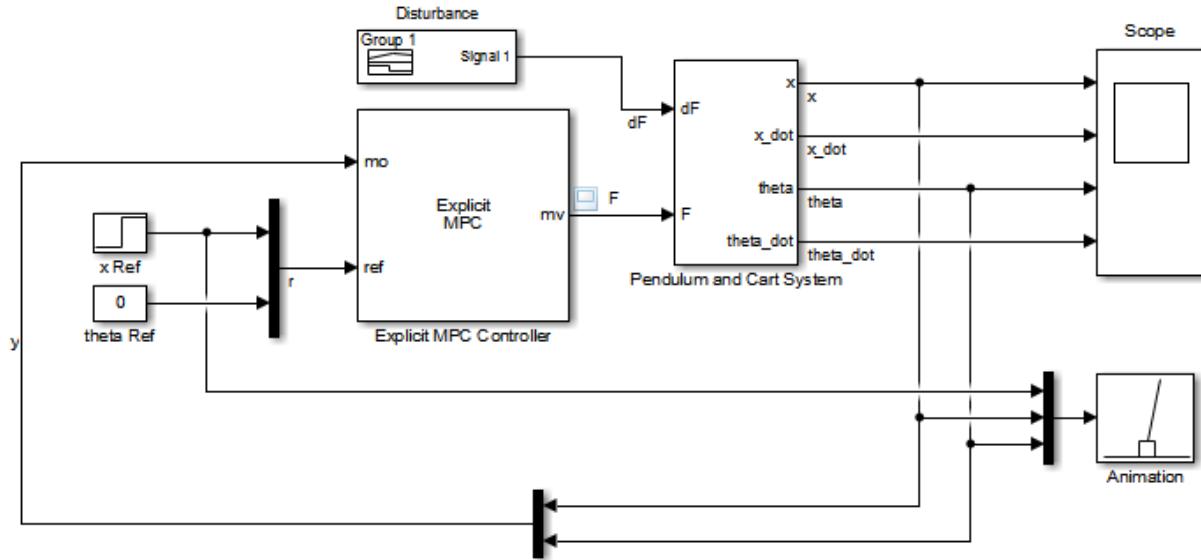
The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

Control Structure

For this example, use a single MPC controller with:

- One manipulated Variable: variable force F .
- Two measured outputs: Cart position x and pendulum angle θ .
- One unmeasured disturbance: Impulse disturbance dF .

```
mdlMPC = mpc_pendcartExplicitMPC ;
open_system(mdlMPC);
```



Copyright 1990-2015 The MathWorks, Inc.

Although cart velocity x_dot and pendulum angular velocity θeta_dot are available from the plant model, to make the design case more realistic, they are excluded as MPC measurements.

While the cart position setpoint varies (step input), the pendulum angle setpoint is constant (0 = upright position).

Linear Plant Model

Since the MPC controller requires a linear time-invariant (LTI) plant model for prediction, linearize the Simulink plant model at the initial operating point.

Specify linearization input and output points

```
io(1) = linio([mdlPlant '/dF'],1, openinput );
io(2) = linio([mdlPlant '/F'],1, openinput );
io(3) = linio([mdlPlant '/Pendulum and Cart System'],1, openoutput );
io(4) = linio([mdlPlant '/Pendulum and Cart System'],3, openoutput );
```

Create operating point specifications for the plant initial conditions.

```
opspec = operspec(mdlPlant);
```

The first state is cart position x , which has a known initial state of 0.

```
opspec.States(1).Known = true;
opspec.States(1).x = 0;
```

The third state is pendulum angle θ , which has a known initial state of 0.

```
opspec.States(3).Known = true;
opspec.States(3).x = 0;
```

Compute operating point using these specifications.

```
options = findopOptions( DisplayReport ,false);
op = findop(mdlPlant,opspec,options);
```

Obtain the linear plant model at the specified operating point.

```
plant = linearize(mdlPlant,op,io);
plant.InputName = { dF ; F };
plant.OutputName = { x ; theta };
```

Examine the poles of the linearized plant.

```
pole(plant)
```

```
ans =
```

```
0
-11.9115
-3.2138
5.1253
```

The plant has an integrator and an unstable pole.

```
bdclose(mdlPlant);
```

Traditional (Implicit) MPC Design

The plant has two inputs, dF and F , and two outputs, x and θ . In this example, dF is specified as an unmeasured disturbance used by the MPC controller for better disturbance rejection. Set the plant signal types.

```
plant = setmpcsignals(plant, ud ,1, mv ,2);
```

To control an unstable plant, the controller sample time cannot be too large (poor disturbance rejection) or too small (excessive computation load). Similarly, the prediction horizon cannot be too long (the plant unstable mode would dominate) or too short (constraint violations would be unforeseen). Use the following parameters for this example:

```
Ts = 0.01;
PredictionHorizon = 50;
ControlHorizon = 5;
mpcobj = mpc(plant,Ts,PredictionHorizon,ControlHorizon);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 1
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

There is a limitation on how much force we can apply to the cart, which is specified as hard constraints on manipulated variable F .

```
mpcobj.MV.Min = -200;
mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by 100.

```
mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from 0.1 to 1.

```
mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position x and the second weight is associated with angle θ .

```
mpcobj.Weights.OV = [1.2 1];
```

To achieve more aggressive disturbance rejection, increase the state estimator gain by multiplying the default disturbance model gains by a factor of 10.

Update the input disturbance model.

```
disturbance_model = getindist(mpcobj);
setindist(mpcobj, model ,disturbance_model*10);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #1 is integrated white noise.
    Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Update the output disturbance model.

```
disturbance_model = getoutdist(mpcobj);
setoutdist(mpcobj, model ,disturbance_model*10);

-->Converting model to discrete time.
    Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Explicit MPC Generation

A simple implicit MPC controller, without the need for constraint or weight changes at run-time, can be converted into an explicit MPC controller with the same control performance. The key benefit of using Explicit MPC is that it avoids real-time optimization, and as a result, is suitable for industrial applications that demand fast sample time. The tradeoff is that explicit MPC has a high memory footprint because optimal solutions for all feasible regions are pre-computed offline and stored for run-time access.

To generate an explicit MPC controller from an implicit MPC controller, define the ranges for parameters such as plant states, references, and manipulated variables. These ranges should cover the operating space for which the plant and controller are designed, to your best knowledge.

```
range = generateExplicitRange(mpcobj);
range.State.Min(:) = -20; % largest range comes from cart position x
range.State.Max(:) = 20;
range.Reference.Min = -20; % largest range comes from cart position x
range.Reference.Max = 20;
range.ManipulatedVariable.Min = -200;
```

```
range.ManipulatedVariable.Max = 200;  
-->Converting model to discrete time.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each manipulated variable.
```

Generate an explicit MPC controller for the defined ranges.

```
mpcobjExplicit = generateExplicitMPC(mpcobj,range);
```

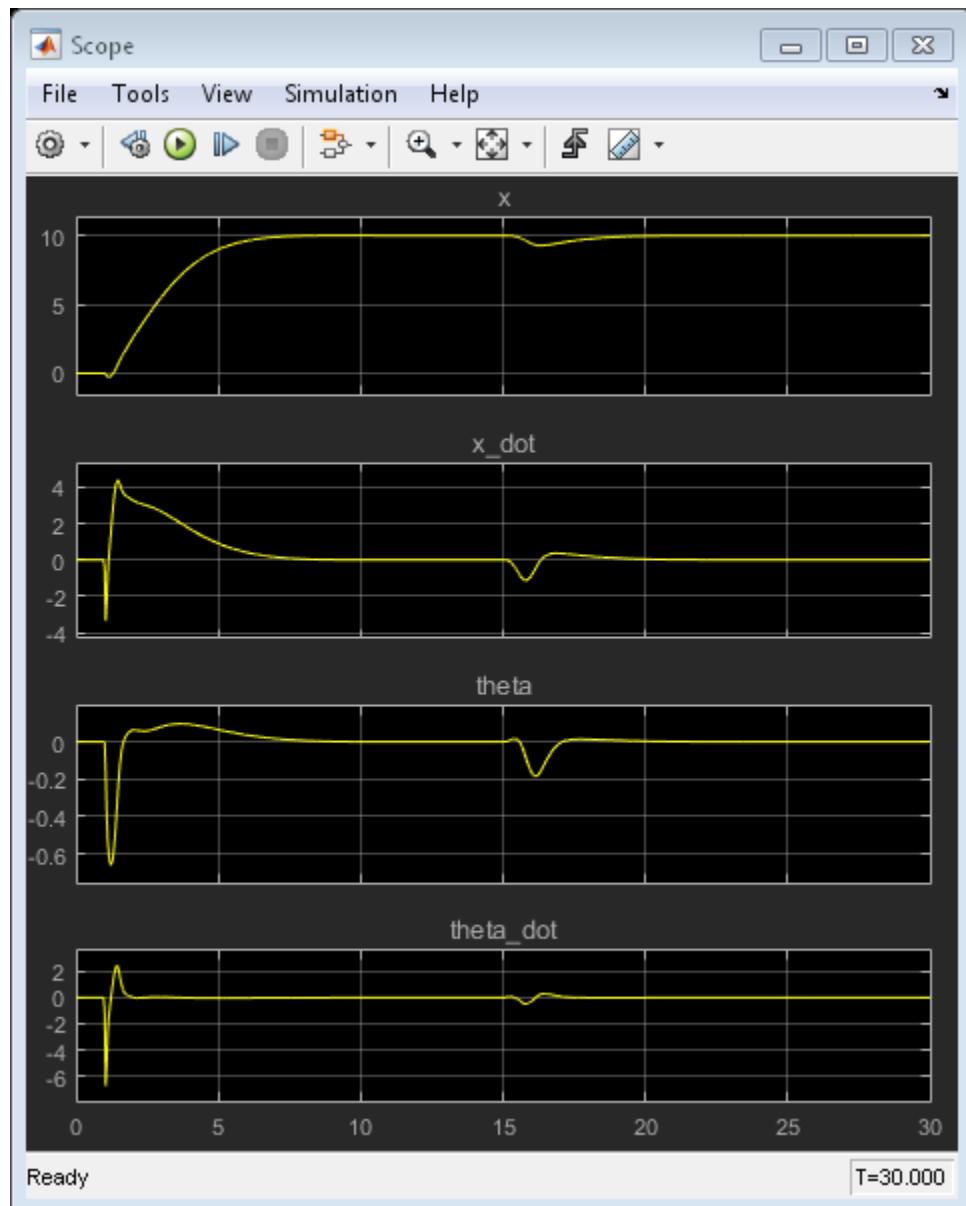
```
Regions found / unexplored:      92/      0
```

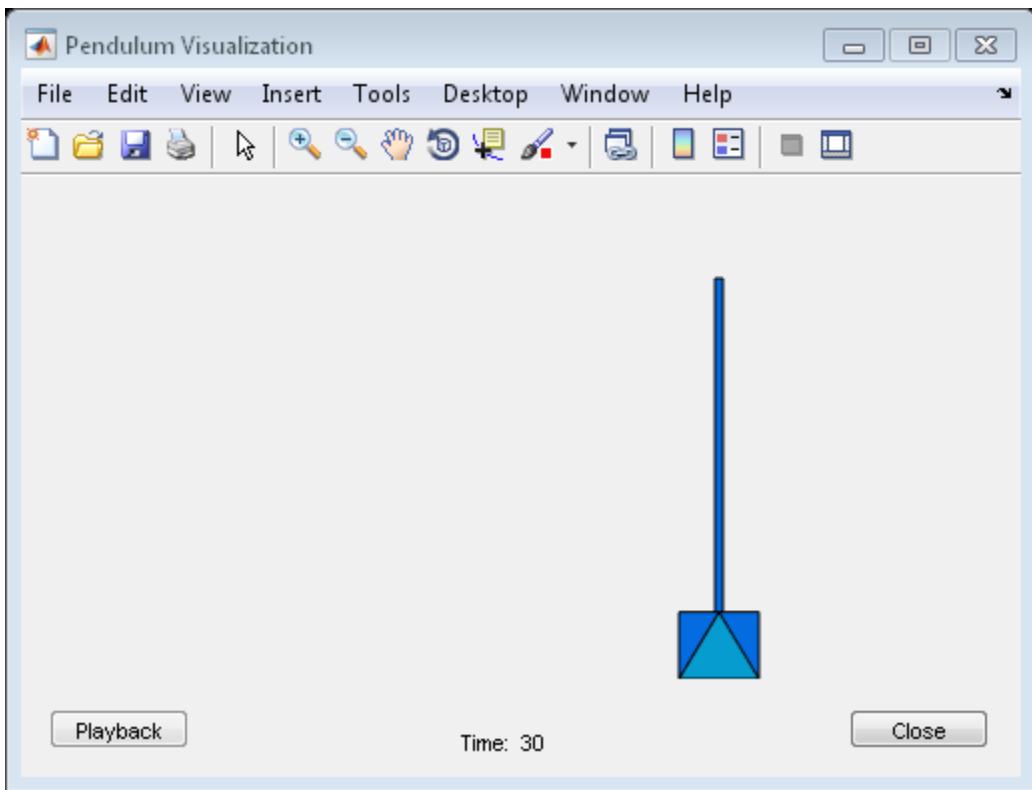
To use the explicit MPC controller in Simulink, specify it in the Explicit MPC Controller block dialog in your Simulink model.

Closed-Loop Simulation

Validate the MPC design with a closed-loop simulation in Simulink.

```
open_system([mdlMPC /Scope]);  
sim(mdlMPC);
```





In the nonlinear simulation, all the control objectives are successfully achieved.

Comparing with the results from “Control of an Inverted Pendulum on a Cart”, the implicit and explicit MPC controllers deliver identical performance as expected.

Discussion

It is important to point out that the designed MPC controller has its limitations. For example, if you increase the step setpoint change to 15, the pendulum fails to recover its upright position during the transition.

To reach the longer distance within the same rise time, the controller applies more force to the cart at the beginning. As a result, the pendulum is displaced from its upright position by a larger angle such as 60 degrees. At such angles, the plant dynamics differ significantly from the LTI predictive model obtained at $\theta = 0$. As a result, errors in

the prediction of plant behavior exceed what the built-in MPC robustness can handle, and the controller fails to perform properly.

A simple workaround to avoid the pendulum falling is to restrict pendulum displacement by adding soft output constraints to *theta* and reducing the ECR weight on constraint softening.

```
mpcobj.OV(2).Min = -pi/2;  
mpcobj.OV(2).Max = pi/2;  
mpcobj.Weights.ECR = 100;
```

However, with these new controller settings, it is no longer possible to reach the longer distance within the required rise time. In other words, controller performance is sacrificed to avoid violation of soft output constraints.

To reach longer distances within the same rise time, the controller needs more accurate models at different angle to improve prediction. Another example “Gain Scheduled MPC Control of an Inverted Pendulum on a Cart” shows how to use gain scheduling MPC to achieve the longer distances.

```
bdclose(mdlMPC);
```

More About

- “Explicit MPC” on page 6-2
- “Control of an Inverted Pendulum on a Cart” on page 4-144
- “Gain Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 7-39

Gain Scheduling MPC Design

- “Gain-Scheduled MPC” on page 7-2
- “Design Workflow for Gain Scheduling” on page 7-3
- “Gain Scheduled MPC Control of Nonlinear Chemical Reactor” on page 7-5
- “Gain Scheduled MPC Control of Mass-Spring System” on page 7-28
- “Gain Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 7-39

Gain-Scheduled MPC

The **Multiple MPC Controllers** block for Simulink allows you to switch between a defined set of MPC Controllers. You might need this feature if the plant operating characteristics change in a predictable way, and the change is such that a single prediction model cannot provide adequate accuracy. This approach is comparable to the use of gain scheduling in conventional feedback control.

The individual MPC controllers coordinate to make switching from one to another bumpless, avoiding a sudden change in the manipulated variables when the switch occurs.

You can perform command-line simulations using the `mpcmovemultiple` command.

More About

- “Design Workflow for Gain Scheduling” on page 7-3
- “Relationship of Multiple MPC Controllers to MPC Controller Block” on page 3-3

Design Workflow for Gain Scheduling

In this section...

[“General Design Steps” on page 7-3](#)

[“Tips” on page 7-3](#)

General Design Steps

- Define and tune a nominal MPC controller for the most likely (or average) operating conditions. (See “MPC Design”).
- Use simulations to determine an operating condition at which the nominal controller loses robustness. See “Simulation”.
- Identify a measurement (or combination of measurements) signaling when the nominal controller should be replaced.
- Determine a plant prediction model to be used at the new condition. Its input and output variables must be the same as in the nominal case.
- Define a new MPC controller based on the new prediction model. Use the nominal controller settings as a starting point, and test and retune controller settings if necessary.
- If two controllers are inadequate to provide robustness over the full operational range, consider adding another. If it appears that you need more than three controllers to provide robustness over the full range, consider using adaptive MPC instead. See “Adaptive MPC Design”.
- In your Simulink model, configure the **Multiple MPC Controllers** block. Specify the set of MPC controllers to be used, and specify the switching criterion.
- Test in closed-loop simulation over the full operating range to verify robustness and bumpless switching.

Tips

- Recommended MPC start-up practice is a warm-up period in which the plant operates under manual control while the controller initializes its state estimate. This typically requires 10-20 control intervals. A warm-up is especially important for the **Multiple MPC Controllers** block. Otherwise, switching between MPC controllers might upset the manipulated variables.

- If you select the **Multiple MPC Controllers** block's custom state estimation option, all MPC controllers in the set must have the same state dimension. This places implicit restrictions on plant and disturbance models.

See Also

`mpcmoveMultiple` | **Multiple MPC Controllers**

Related Examples

- “Schedule Controllers at Multiple Operating Points”
- “Coordinate Multiple Controllers at Different Operating Points” on page 4-64
- “Gain Scheduled MPC Control of Nonlinear Chemical Reactor” on page 7-5
- “Gain Scheduled MPC Control of Mass-Spring System” on page 7-28

More About

- “Relationship of Multiple MPC Controllers to MPC Controller Block” on page 3-3

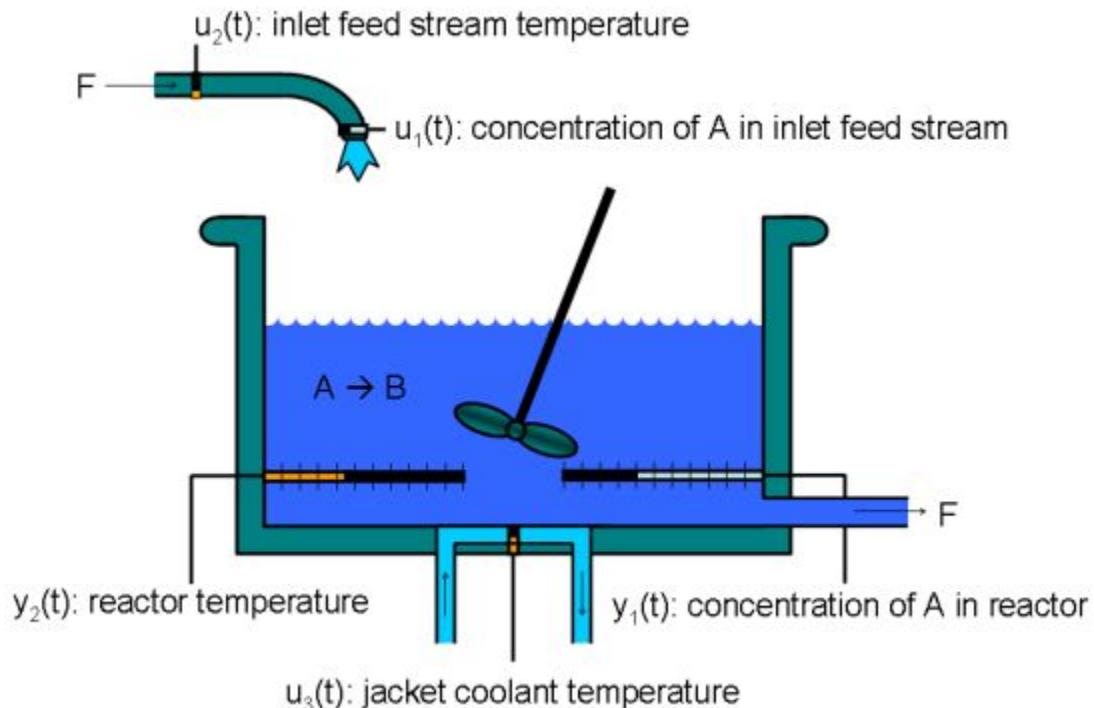
Gain Scheduled MPC Control of Nonlinear Chemical Reactor

This example shows how to use multiple MPC controllers to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

Multiple MPC Controllers are designed at different operating conditions and then implemented with the Multiple MPC Controller block in Simulink. At run time, a scheduling signal is used to switch controller from one to another.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:



This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be

perfectly mixed, and a single first-order exothermic and irreversible reaction, A \rightarrow B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{aligned} u_1 &= CA_i && \text{Concentration of A in inlet feed stream} [kgmol/m^3] \\ u_2 &= T_i && \text{Inlet feed stream temperature} [K] \\ u_3 &= T_c && \text{Jacket coolant temperature} [K] \end{aligned}$$

and the outputs ($y(t)$), which are also the states of the model ($x(t)$), are:

$$\begin{aligned} y_1 = x_1 &= CA && \text{Concentration of A in reactor tank} [kgmol/m^3] \\ y_2 = x_2 &= T && \text{Reactor temperature} [K] \end{aligned}$$

The control objective is to maintain the concentration of reagent A, CA at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference. The inlet feed stream concentration and temperature are assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

About Gain Scheduled Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical operating range. Next you can choose one of the two approaches to implement MPC control strategy:

(1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based on a desired scheduling strategy, as discussed in this example. Use this approach when the plant models have different orders or time delays.

(2) Design one MPC controller offline at a nominal operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.

- If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:

(1) Use successive linearization. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” for more details. Use this approach when a nonlinear plant model is available and can be linearized at run time.

(2) Use online estimation to identify a linear model when loop is closed. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” for more details. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

To run this example, Simulink® and Simulink Control Design® are required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end
if ~mpcchecktoolboxinstalled( slcontrol )
    disp( Simulink Control Design(R) is required to run this example. )
    return
end
```

First, a linear plant model is obtained at the initial operating condition, CAi is 10 kgmol/m³, Ti and Tc are 298.15 K. Functions from Simulink Control Design such as "operspec", "findop", "linearize", are used to generate the linear state space system from the Simulink model.

Create operating point specification.

```
plant_mdl = mpc_cstr_plant ;
op = operspec(plant_mdl);
```

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

```
[op_point, op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];
```

Operating Point Search Report:

```
-----
Operating Report for the Model mpc_cstr_plant.
(Time-Varying Components Evaluated at time t=0)
```

Operating point specifications were successfully met.

States:

```
-----
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           311      dx:     8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:           8.57      dx:    -6.87e-12 (0)
```

Inputs:

```
-----
(1.) mpc_cstr_plant/CAi
    u:          10
(2.) mpc_cstr_plant/Ti
    u:         298
(3.) mpc_cstr_plant/Tc
    u:         298

Outputs:
-----
(1.) mpc_cstr_plant/T
    y:          311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
    y:          8.57   [-Inf Inf]
```

Obtain linear model at the initial condition.

```
plant = linearize(plant_mdl, op_point);
```

Verify that the linear model is open-loop stable at this condition.

```
eig(plant)
```

```
ans =
-0.5223
-0.8952
```

Design MPC Controller for Initial Operating Condition

You design an MPC at the initial operating condition.

```
Ts = 0.5;
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```
plant.InputGroup.UnmeasuredDisturbances = [1 2];
plant.InputGroup.ManipulatedVariables = 3;
plant.OutputGroup.Measured = [1 2];
plant.InputName = { CAi , Ti , Tc };
```

```
plant.OutputName = { T , CA };
```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant, Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1  
    for output(s) y1 and zero weight for output(s) y2
```

Set nominal values in the controller. Note that nominal values for unmeasured disturbance must be zero.

```
mpcobj.Model.Nominal = struct( X , x0, U , [0;0;u0(3)], Y , y0, DX , [0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude

```
Uscale = [10;30;50];  
Yscale = [50;10];  
mpcobj.DV(1).ScaleFactor = Uscale(1);  
mpcobj.DV(2).ScaleFactor = Uscale(2);  
mpcobj.MV.ScaleFactor = Uscale(3);  
mpcobj.OV(1).ScaleFactor = Yscale(1);  
mpcobj.OV(2).ScaleFactor = Yscale(2);
```

The goal will be to track a specified transition in the reactor concentration. The reactor temperature will be measured and used in state estimation but the controller will not attempt to regulate it directly. It will vary as needed to regulate the concentration. Thus, set its MPC weight to zero.

```
mpcobj.Weights.OV = [0 1];
```

Plant inputs 1 and 2 are unmeasured disturbances. By default, the controller assumes integrated white noise with unit magnitude at these inputs when configuring the state estimator. Try increasing the state estimator signal-to-noise by a factor of 10 to improve disturbance rejection performance.

```
D = ss(getindist(mpcobj));  
D.b = eye(2)*10;
```

```

setindist(mpcobj, model , D);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
Assuming unmeasured input disturbance #1 is integrated white noise.
Assuming unmeasured input disturbance #2 is integrated white noise.
Assuming no disturbance added to measured output channel #2.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each

```

All other MPC parameters are at their default values.

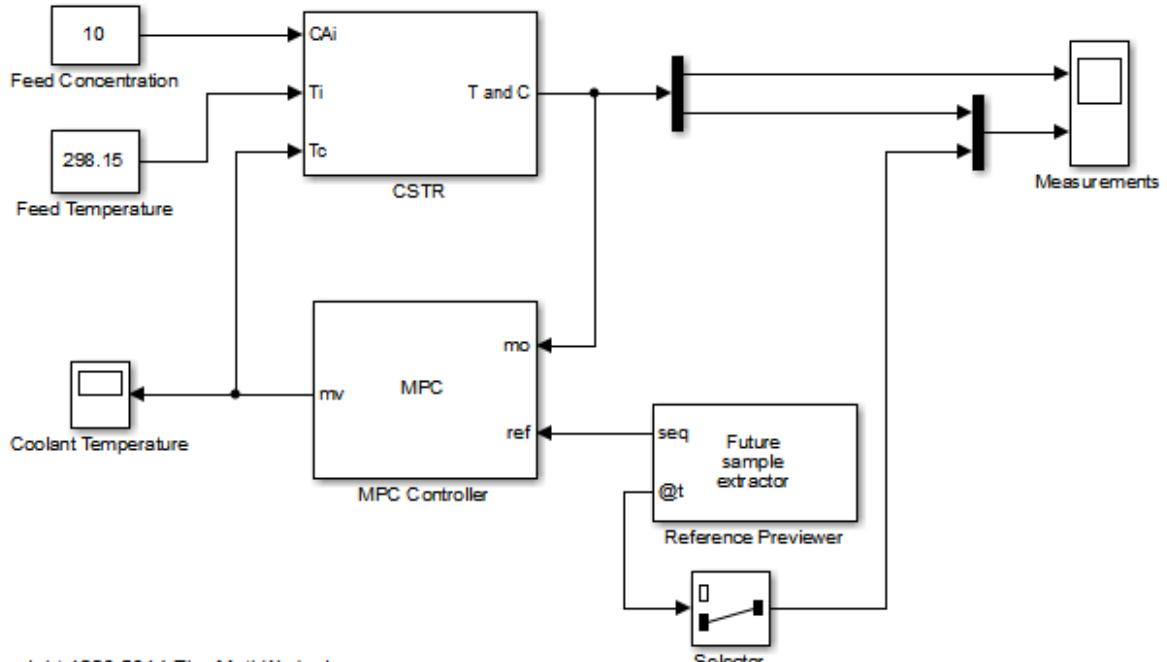
Test the Controller With a Step Disturbance in Feed Concentration

"mpc_cstr_single" contains a Simulink® model with CSTR and MPC Controller blocks in a feedback configuration.

```

mpc_mdl = mpc_cstr_single ;
open_system(mpc_mdl)

```



Note that the MPC Controller block is configured to look ahead (preview) the setpoint changes in the future, i.e., anticipating the setpoint transition. This generally improves setpoint tracking.

Define a constant setpoint for the output.

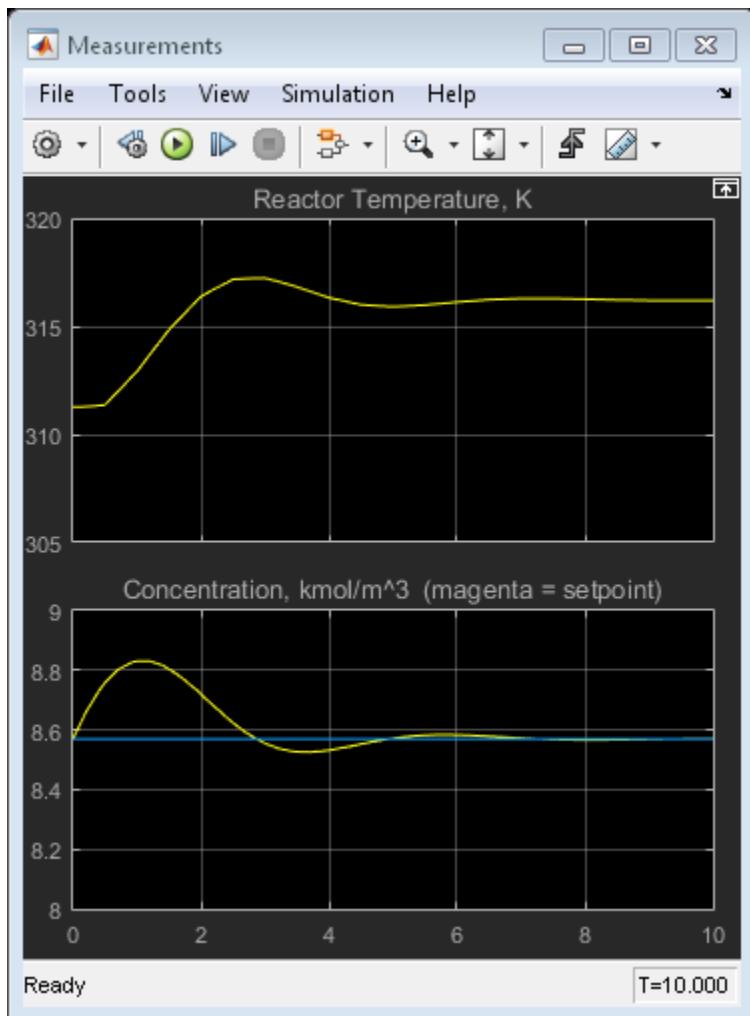
```
CSTR_Setpoints.time = [0; 60];  
CSTR_Setpoints.signals.values = [y0 y0] ;
```

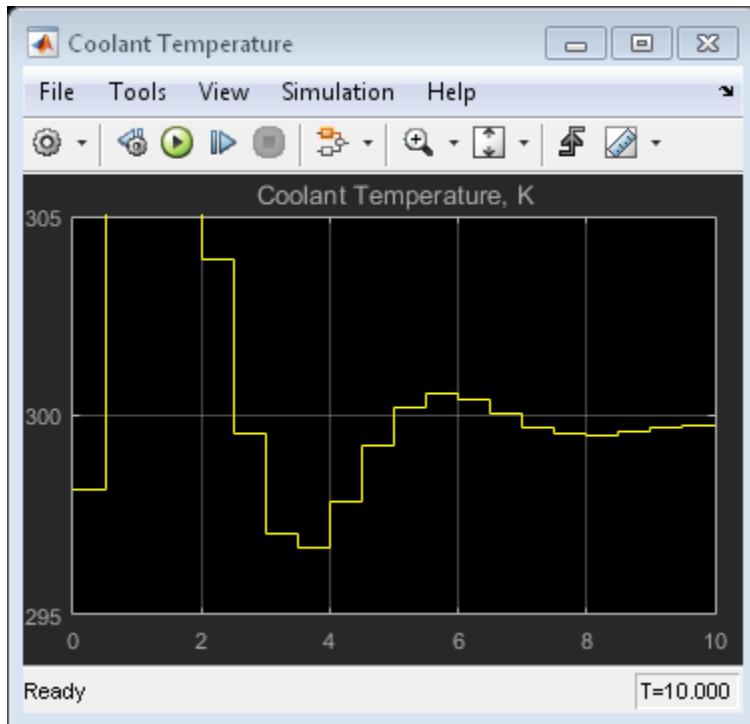
Test the response to a 5% increase in feed concentration.

```
set_param([mpc_mdl /Feed Concentration ], Value , 10.5 );
```

Set plot scales and simulate the response.

```
open_system([mpc_mdl /Measurements ])  
open_system([mpc_mdl /Coolant Temperature ])  
set_param([mpc_mdl /Measurements ], Ymin , 305~8 , Ymax , 320~9 )  
set_param([mpc_mdl /Coolant Temperature ], Ymin , 295 , Ymax , 305 )  
sim	mpc_mdl, 10);  
  
-->Converting model to discrete time.  
Assuming no disturbance added to measured output channel #2.  
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```





The closed-loop response is satisfactory.

Simulate Designed MPC Controller Using Full Transition

First, define the desired setpoint transition. After a 10-minute warm-up period, ramp the concentration setpoint downward at a rate of 0.25 per minute until it reaches 2.0 kmol/m³.

```
CSTR_Setpoints.time = [0 10 11:39] ;
CSTR_Setpoints.signals.values = [y0(1)*ones(31,1),[y0(2);y0(2);(y0(2):-0.25:2) ;2;2]];
```

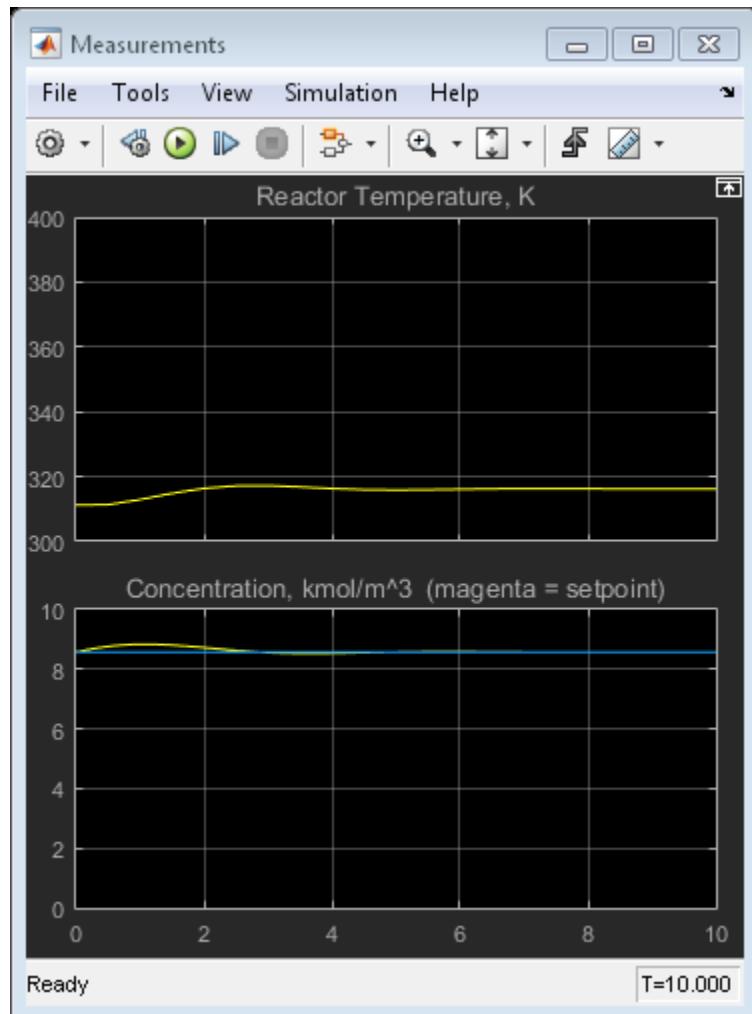
Remove the 5% increase in feed concentration used previously.

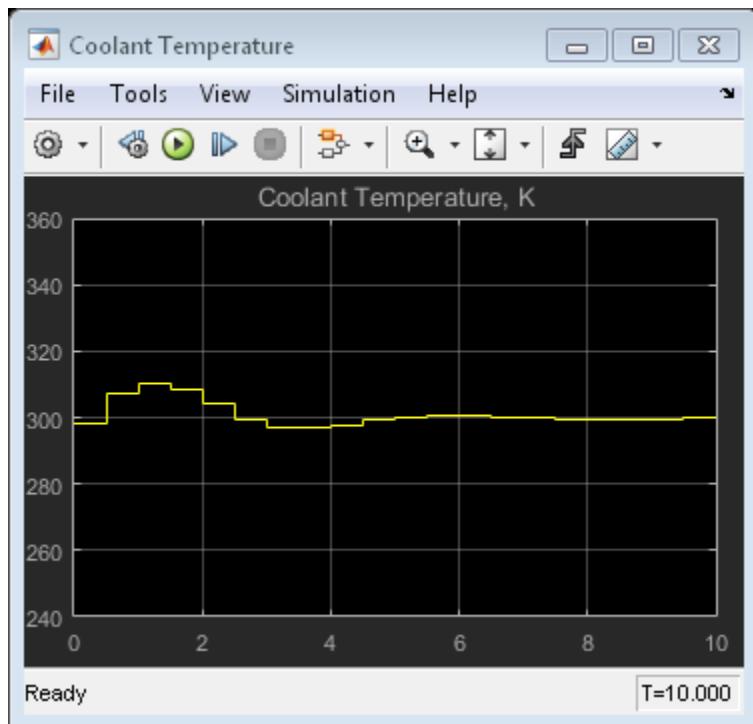
```
set_param([mpc_mdl /Feed Concentration ], Value , 10 )
```

Set plot scales and simulate the response.

```
set_param([mpc_mdl /Measurements ], Ymin , 300~0 , Ymax , 400~10 )
```

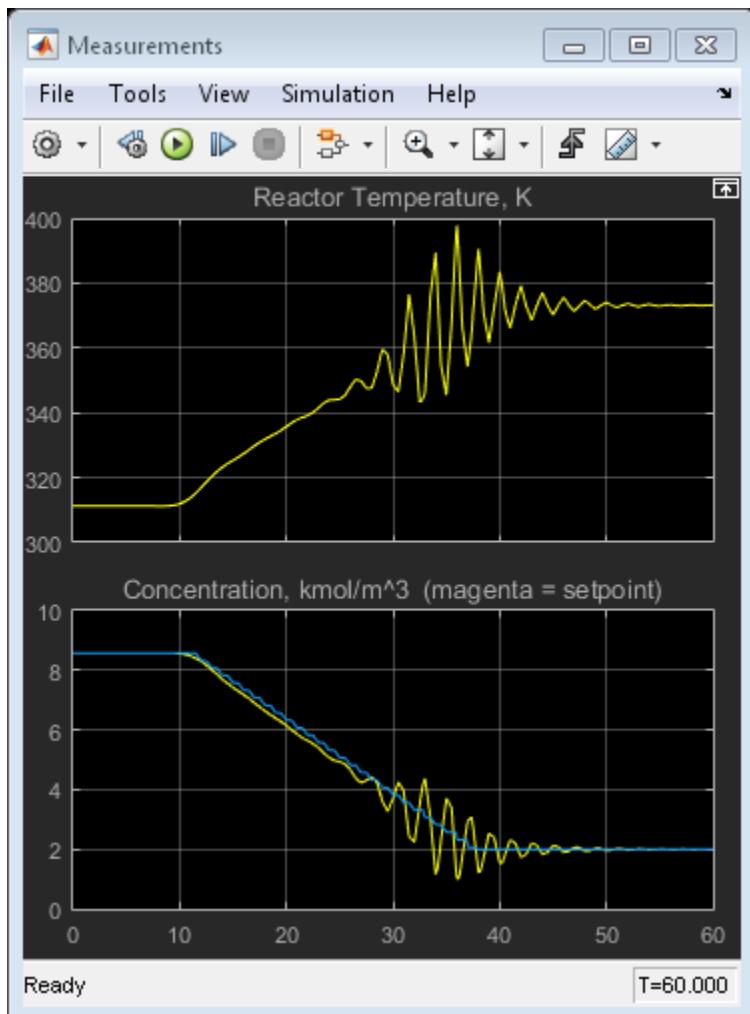
```
set_param([mpc_mdl /Coolant Temperature ], Ymin , 240 , Ymax , 360 )
```

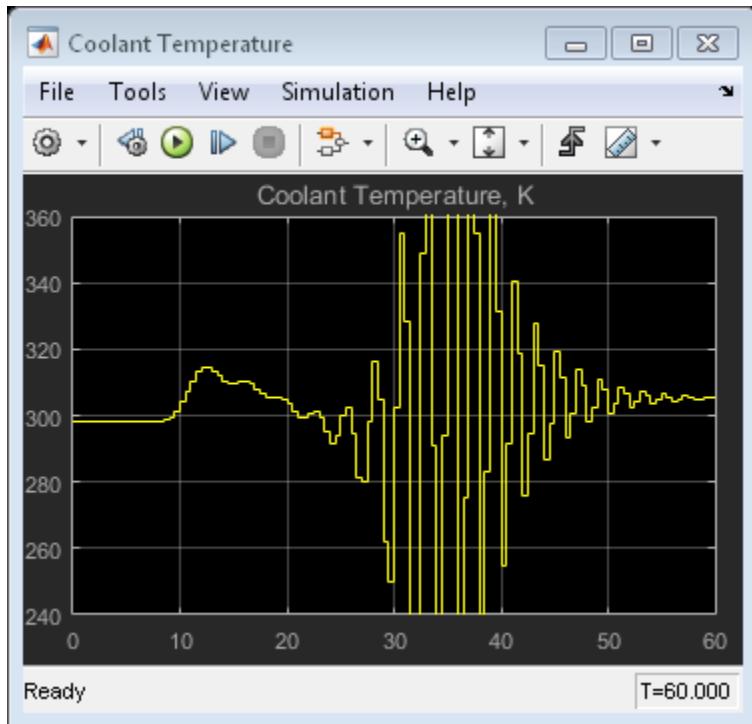




Simulate model.

```
sim(mpc_md1, 60)
```





The closed-loop response is unacceptable. Performance along the full transition can be improved if other MPC controllers are designed at different operating conditions along the transition path. In the next two section, two additional MPC controllers are design at intermediate and final transition stages respectively.

Design MPC Controller for Intermediate Operating Condition

Create operating point specification.

```
op = operspec(plant_md1);
```

Feed concentration is known.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known.

```
op.Inputs(2).u = 298.15;
```

```
op.Inputs(2).Known = true;
```

Reactor concentration is known

```
op.Outputs(2).y = 5.5;
op.Outputs(2).Known = true;
```

Find steady state operating condition.

```
[op_point, op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];
```

Operating Point Search Report:

Operating Report for the Model mpc_cstr_plant.
(Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.

States:

```
-----
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           339      dx:    3.42e-08 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:           5.5      dx:   -2.87e-09 (0)
```

Inputs:

```
-----
(1.) mpc_cstr_plant/CAi
      u:          10
(2.) mpc_cstr_plant/Ti
      u:         298
(3.) mpc_cstr_plant/Tc
      u:         298     [-Inf Inf]
```

Outputs:

```
-----
(1.) mpc_cstr_plant/T
      y:           339     [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:           5.5     (5.5)
```

Obtain linear model at the initial condition.

```
plant_intermediate = linearize(plant_mdl, op_point);
```

Verify that the linear model is open-loop unstable at this condition.

```
eig(plant_intermediate)
```

```
ans =
```

```
0.4941
-0.8357
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```
plant_intermediate.InputGroup.UnmeasuredDisturbances = [1 2];
plant_intermediate.InputGroup.ManipulatedVariables = 3;
plant_intermediate.OutputGroup.Measured = [1 2];
plant_intermediate.InputName = { CAi , Ti , Tc };
plant_intermediate.OutputName = { T , CA };
```

Create MPC controller with default prediction and control horizons

```
mpcobj_intermediate = mpc(plant_intermediate, Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Set nominal values, scale factors and weights in the controller

```
mpcobj_intermediate.Model.Nominal = struct( X , x0, U , [0;0;u0(3)], Y , y0, DX , [0
Uscale = [10;30;50];
Yscale = [50;10];
mpcobj_intermediate.DV(1).ScaleFactor = Uscale(1);
mpcobj_intermediate.DV(2).ScaleFactor = Uscale(2);
mpcobj_intermediate.MV.ScaleFactor = Uscale(3);
mpcobj_intermediate.OV(1).ScaleFactor = Yscale(1);
```

```

mpcobj_intermediate.OV(2).ScaleFactor = Yscale(2);
mpcobj_intermediate.Weights.OV = [0 1];
D = ss(getindist(mpcobj_intermediate));
D.b = eye(2)*10;
setindist(mpcobj_intermediate, model, D);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
Assuming unmeasured input disturbance #1 is integrated white noise.
Assuming unmeasured input disturbance #2 is integrated white noise.
Assuming no disturbance added to measured output channel #2.
Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

```

Design MPC Controller for Final Operating Condition

Create operating point specification.

```
op = operspec(plant_mdl);
```

Feed concentration is known.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Reactor concentration is known

```
op.Outputs(2).y = 2;
op.Outputs(2).Known = true;
```

Find steady state operating condition.

```
[op_point, op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];
```

Operating Point Search Report:

```
Operating Report for the Model mpc_cstr_plant.  
(Time-Varying Components Evaluated at time t=0)
```

```
Operating point specifications were successfully met.  
States:
```

```
-----  
(1.) mpc_cstr_plant/CSTR/Integrator  
      x:          373      dx:      5.57e-11 (0)  
(2.) mpc_cstr_plant/CSTR/Integrator1  
      x:           2      dx:     -4.6e-12 (0)
```

```
Inputs:
```

```
-----  
(1.) mpc_cstr_plant/CAi  
      u:          10  
(2.) mpc_cstr_plant/Ti  
      u:          298  
(3.) mpc_cstr_plant/Tc  
      u:          305      [-Inf Inf]
```

```
Outputs:
```

```
-----  
(1.) mpc_cstr_plant/T  
      y:          373      [-Inf Inf]  
(2.) mpc_cstr_plant/CA  
      y:           2      (2)
```

Obtain linear model at the initial condition.

```
plant_final = linearize(plant_mdl, op_point);
```

Verify that the linear model is again open-loop stable at this condition.

```
eig(plant_final)
```

```
ans =
```

```
-1.1077 + 1.0901i  
-1.1077 - 1.0901i
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```

plant_final.InputGroup.UnmeasuredDisturbances = [1 2];
plant_final.InputGroup.ManipulatedVariables = 3;
plant_final.OutputGroup.Measured = [1 2];
plant_final.InputName = { CAi , Ti , Tc };
plant_final.OutputName = { T , CA };

```

Create MPC controller with default prediction and control horizons

```
mpcobj_final = mpc(plant_final, Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Set nominal values, scale factors and weights in the controller

```

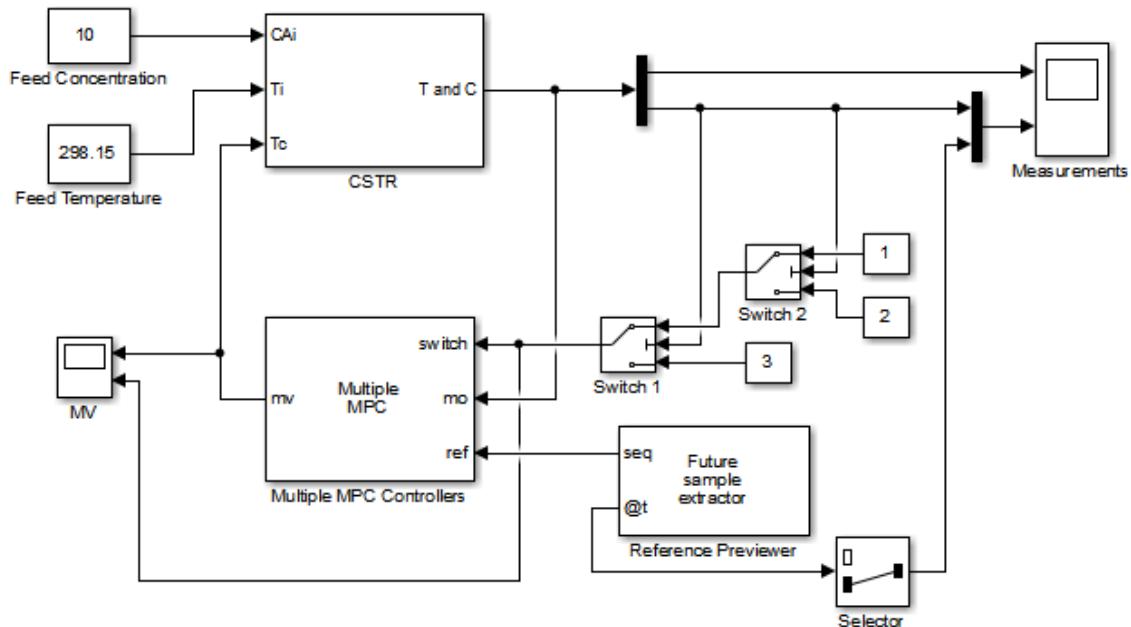
mpcobj_final.Model.Nominal = struct( X , x0, U , [0;0;u0(3)], Y , yo, DX , [0 0]);
Uscale = [10;30;50];
Yscale = [50;10];
mpcobj_final.DV(1).ScaleFactor = Uscale(1);
mpcobj_final.DV(2).ScaleFactor = Uscale(2);
mpcobj_final.MV.ScaleFactor = Uscale(3);
mpcobj_final.OV(1).ScaleFactor = Yscale(1);
mpcobj_final.OV(2).ScaleFactor = Yscale(2);
mpcobj_final.Weights.OV = [0 1];
D = ss(getindist/mpcobj_final));
D.b = eye(2)*10;
setindist/mpcobj_final, model , D);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #1 is integrated white noise.
    Assuming unmeasured input disturbance #2 is integrated white noise.
    Assuming no disturbance added to measured output channel #2.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Control the CSTR Plant With the Multiple MPC Controllers Block

The following model uses the Multiple MPC Controllers block to implement three MPC controllers across the operating range.

```
mmpc_mdl = mpc_cstr_multiple ;
open_system(mmpc_mdl);
```



Copyright 1990-2014 The MathWorks, Inc.

Note that it has been configured to use the three controllers in a sequence: mpcobj, mpcobj_intermediate and mpcobj_final.

```
open_system([mmpc_mdl /Multiple MPC Controllers]);
```

Note also that the two switches specify when to switch from one controller to another. The rules are: 1. If CSTR concentration ≥ 8 , use "mpcobj" 2. If $3 \leq$ CSTR concentration < 8 , use "mpcobj_intermediate" 3. If CSTR concentration < 3 , use "mpcobj_final"

Simulate with the Multiple MPC Controllers block

```
open_system([mmpc_mdl /Measurements]);
open_system([mmpc_mdl /MV]);
sim(mmpc_mdl)
```

-->Converting model to discrete time.

Assuming no disturbance added to measured output channel #2.

Assuming no disturbance added to measured output channel #1.

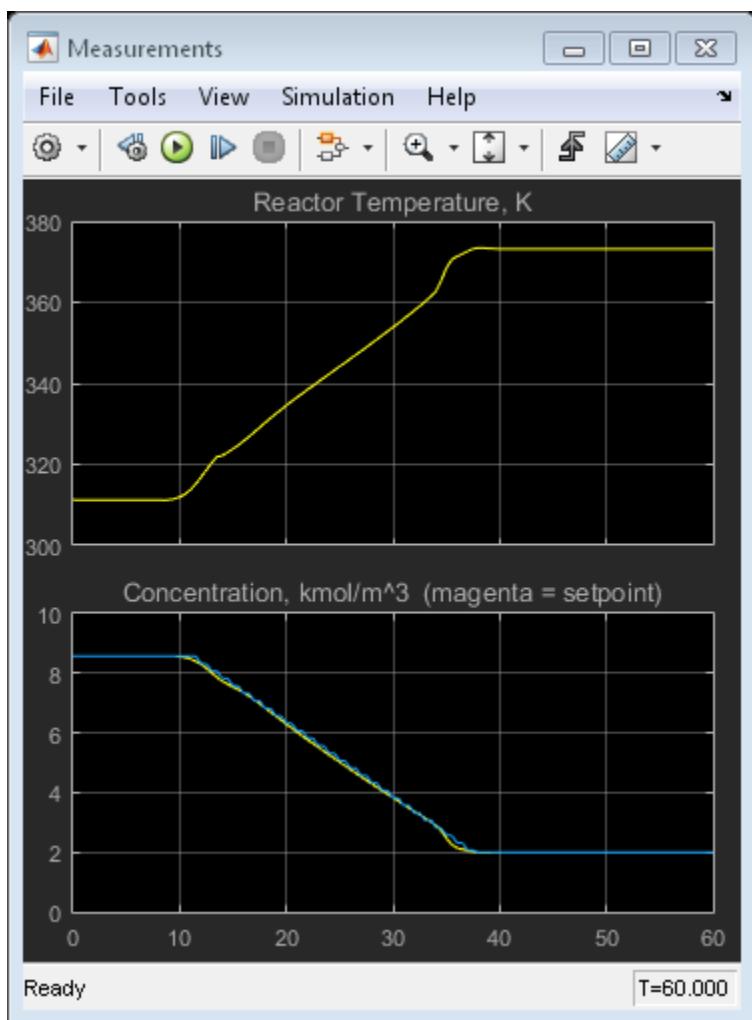
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

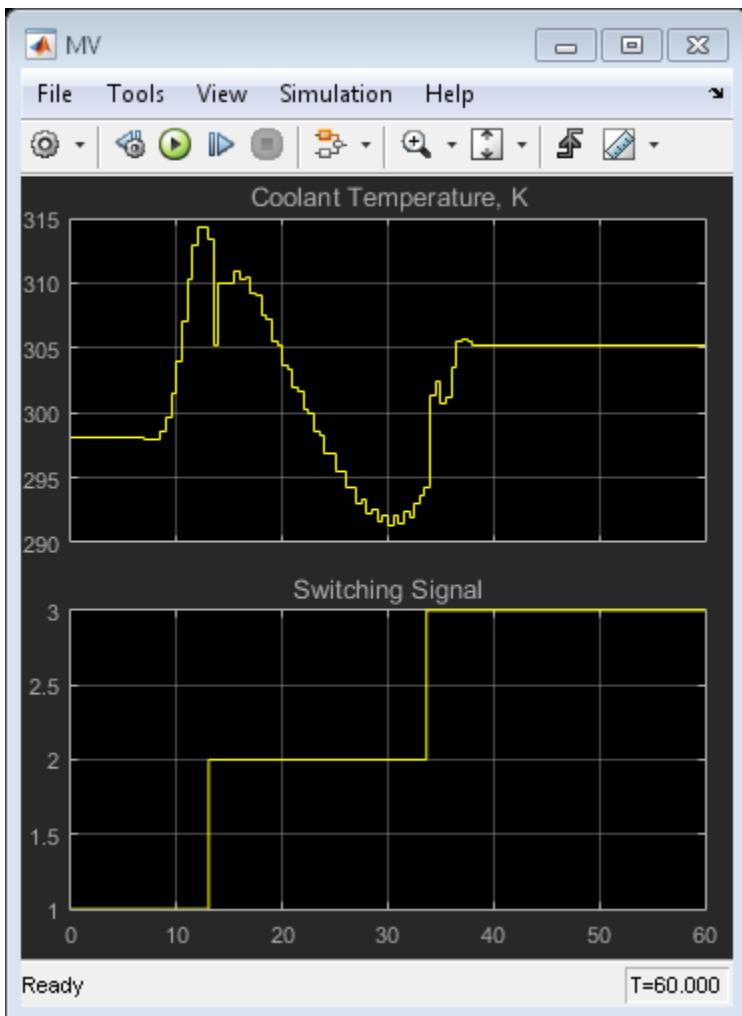
-->Converting model to discrete time.

Assuming no disturbance added to measured output channel #2.

Assuming no disturbance added to measured output channel #1.

-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.





The transition is now well controlled. The major improvement is in the transition through the open-loop unstable region. The plot of the switching signal shows when controller transitions occur. The MV character changes at these times because of the change in dynamic characteristics introduced by the new prediction model.

```
bdclose(plant_mdl)
bdclose(mpc_mdl)
```

```
bdclose('mmpc_md1')
```

Related Examples

- “Schedule Controllers at Multiple Operating Points”
- “Coordinate Multiple Controllers at Different Operating Points” on page 4-64
- “Gain Scheduled MPC Control of Mass-Spring System” on page 7-28

More About

- “Design Workflow for Gain Scheduling” on page 7-3

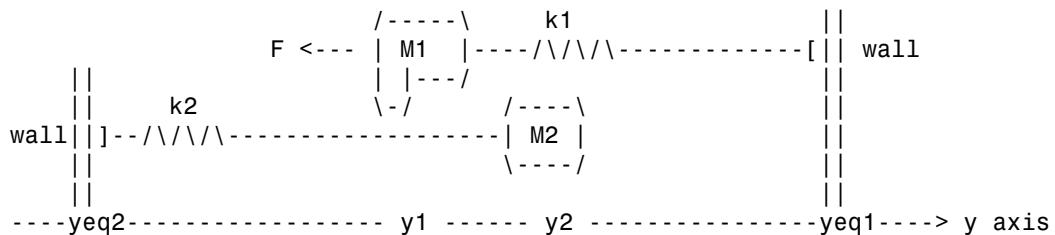
Gain Scheduled MPC Control of Mass-Spring System

This example shows how to use an Multiple MPC Controllers block to implement gain scheduled MPC control of a nonlinear plant.

System Description

The system is composed by two masses M1 and M2 connected to two springs k1 and k2 respectively. The collision is assumed completely inelastic. Mass M1 is pulled by a force F, which is the manipulated variable. The objective is to make mass M1's position y_1 track a given reference r .

The dynamics are twofold: when the masses are detached, M1 moves freely. Otherwise, M1+M2 move together. We assume that only M1 position and a contact sensor are available for feedback. The latter is used to trigger switching the MPC controllers. Note that position and velocity of mass M2 are not controllable.



The model is a simplified version of the model proposed in the following reference:

A. Bemporad, S. Di Cairano, I. V. Kolmanovsky, and D. Hrovat, "Hybrid modeling and control of a multibody magnetic actuator for automotive applications," in Proc. 46th IEEE® Conf. on Decision and Control, New Orleans, LA, 2007.

Model Parameters

```
M1=1; % mass
M2=5; % mass
k1=1; % spring constant
k2=0.1; % spring constant
b1=0.3; % friction coefficient
b2=0.8; % friction coefficient
yeq1=10; % wall mount position
yeq2=-10; % wall mount position
```

State Space Models

states: position and velocity of mass M1; manipulated variable: pull force F measured
 disturbance: a constant value of 1 which provides calibrates spring force to the right
 value measured output: position of mass M1

State-space model of M1 when masses are not in contact.

```
A1=[0 1;-k1/M1 -b1/M1];
B1=[0 0;-1/M1 k1*yeq1/M1];
C1=[1 0];
D1=[0 0];
sys1=ss(A1,B1,C1,D1);
sys1=setmpcsignals(sys1, MD ,2);

-->Assuming unspecified input signals are manipulated variables.
```

State-space model when the two masses are in contact.

```
A2=[0 1; -(k1+k2) / (M1+M2) -(b1+b2) / (M1+M2)];
B2=[0 0;-1 / (M1+M2) (k1*yeq1+k2*yeq2) / (M1+M2) ];
C2=[1 0];
D2=[0 0];
sys2=ss(A2,B2,C2,D2);
sys2=setmpcsignals(sys2, MD ,2);

-->Assuming unspecified input signals are manipulated variables.
```

Design MPC Controllers

Common parameters

```
Ts=0.2; % sampling time
p=20; % prediction horizon
m=1; % control horizon
```

Define MPC object for mass M1 detached from M2.

```
MPC1=mpc(sys1,Ts,p,m);
MPC1.Weights.OV=1;

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define constraints on the manipulated variable.

```
MPC1.MV=struct( Min ,0, Max ,Inf, RateMin ,-1e3, RateMax ,1e3);
```

Define MPC object for mass M1 and M2 stuck together.

```
MPC2=mpc(sys2,Ts,p,m);
MPC2.Weights.OV=1;
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define constraints on the manipulated variable.

```
MPC2.MV=MPC1.MV;
```

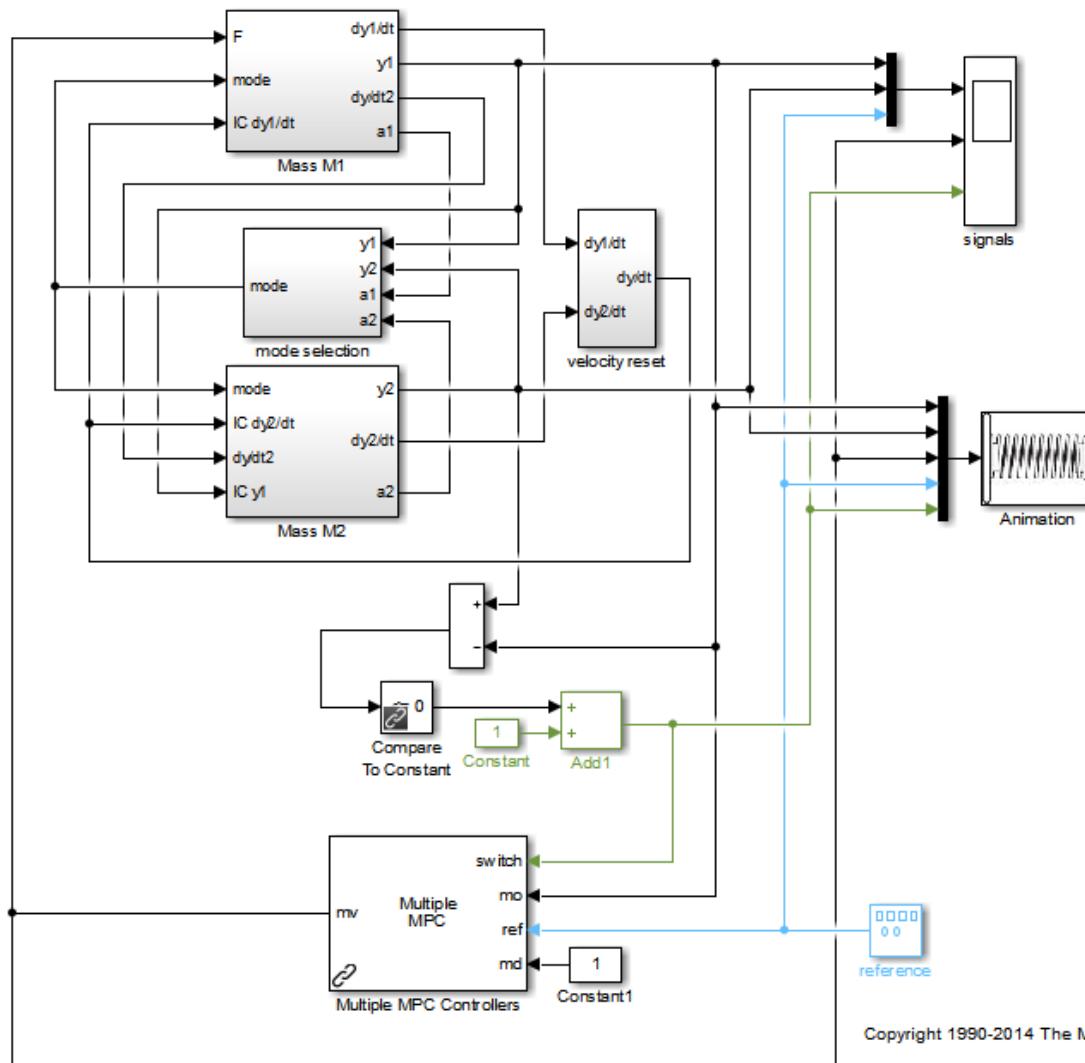
Simulate Gain Scheduled MPC in Simulink

To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled( simulink )
    disp( Simulink(R) is required to run this example. )
    return
end
mdl = mpc_switching ;
```

Simulate gain scheduled MPC control with Multiple MPC Controllers block.

```
y1initial=0; % Initial positions
y2initial=10;
open_system(mdl);
if exist( animationmpc_switchoff , var ) && animationmpc_switchoff
    close_system([mdl /Animation ]);
    clear animationmpc_switchoff
end
```



Copyright 1990-2014 The MathWorks, Inc.

```

disp( Start simulation by switching control between MPC1 and MPC2 ... );
disp( Control performance is satisfactory. );
open_system([mdl /signals ]);
sim(mdl);

```

Start simulation by switching control between MPC1 and MPC2 ...

Control performance is satisfactory.

-->Converting model to discrete time.

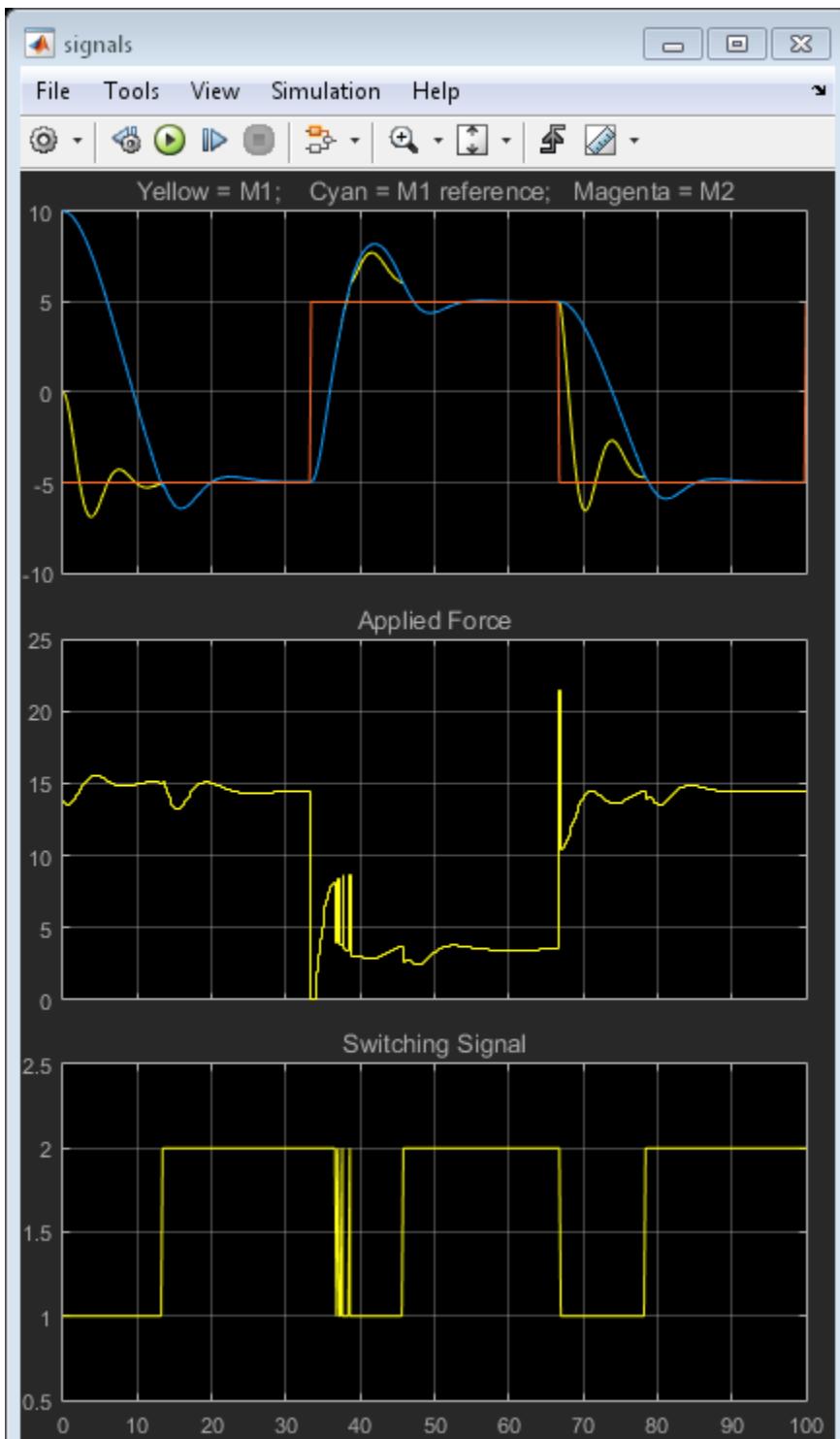
-->Assuming output disturbance added to measured output channel #1 is integrated white

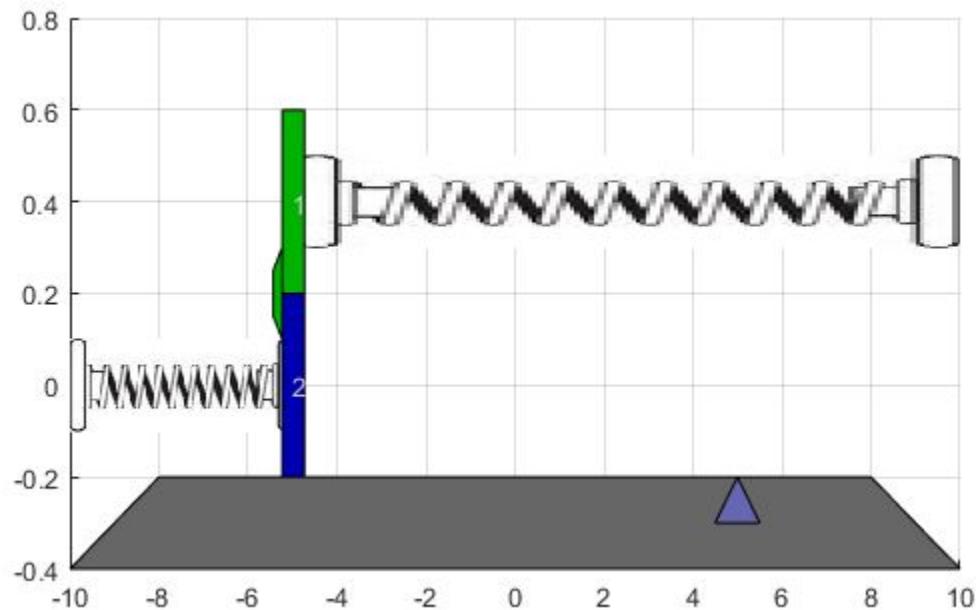
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each

-->Converting model to discrete time.

-->Assuming output disturbance added to measured output channel #1 is integrated white

-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each



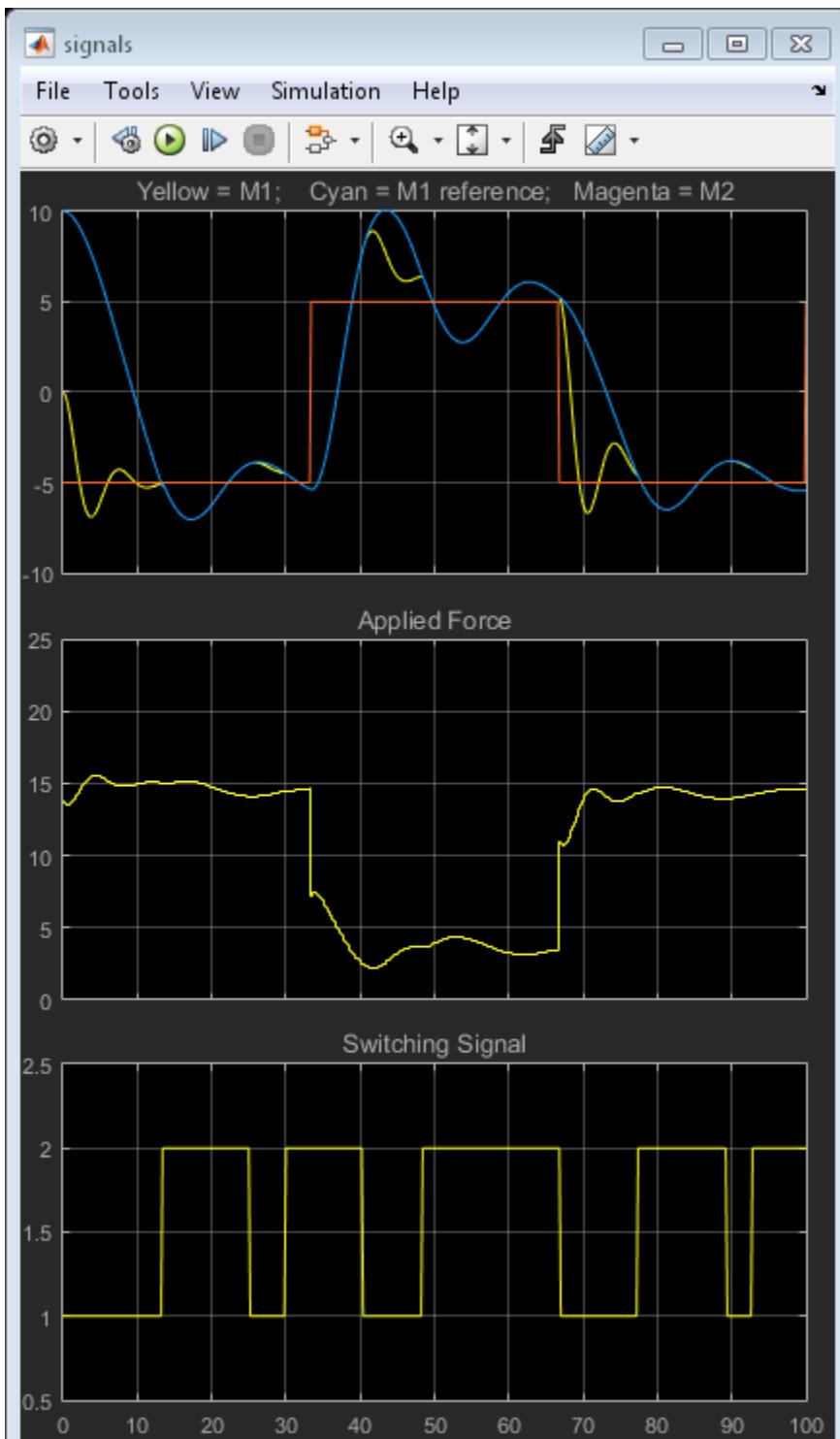


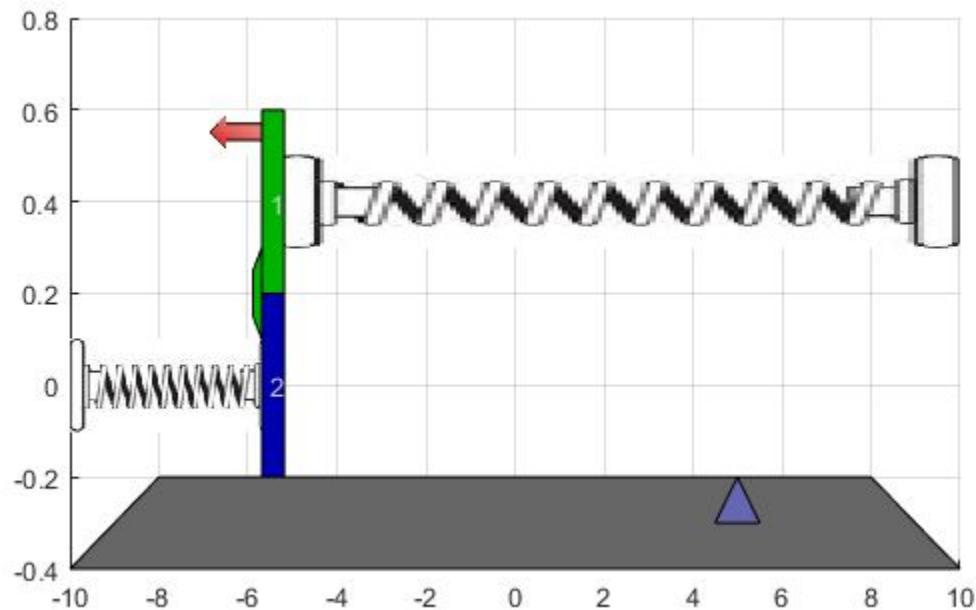
Use of two controllers provides good performance under all conditions.

Repeat Simulation Using MPC1 Only (Assumes Masses Never in Contact)

```
disp( Now repeat simulation by using only MPC1 ... );
disp( When two masses stick together, control performance deteriorates. );
MPC2save=MPC2;
MPC2=MPC1; %#ok<*NASGU>
sim(md1);
```

Now repeat simulation by using only MPC1 ...
When two masses stick together, control performance deteriorates.



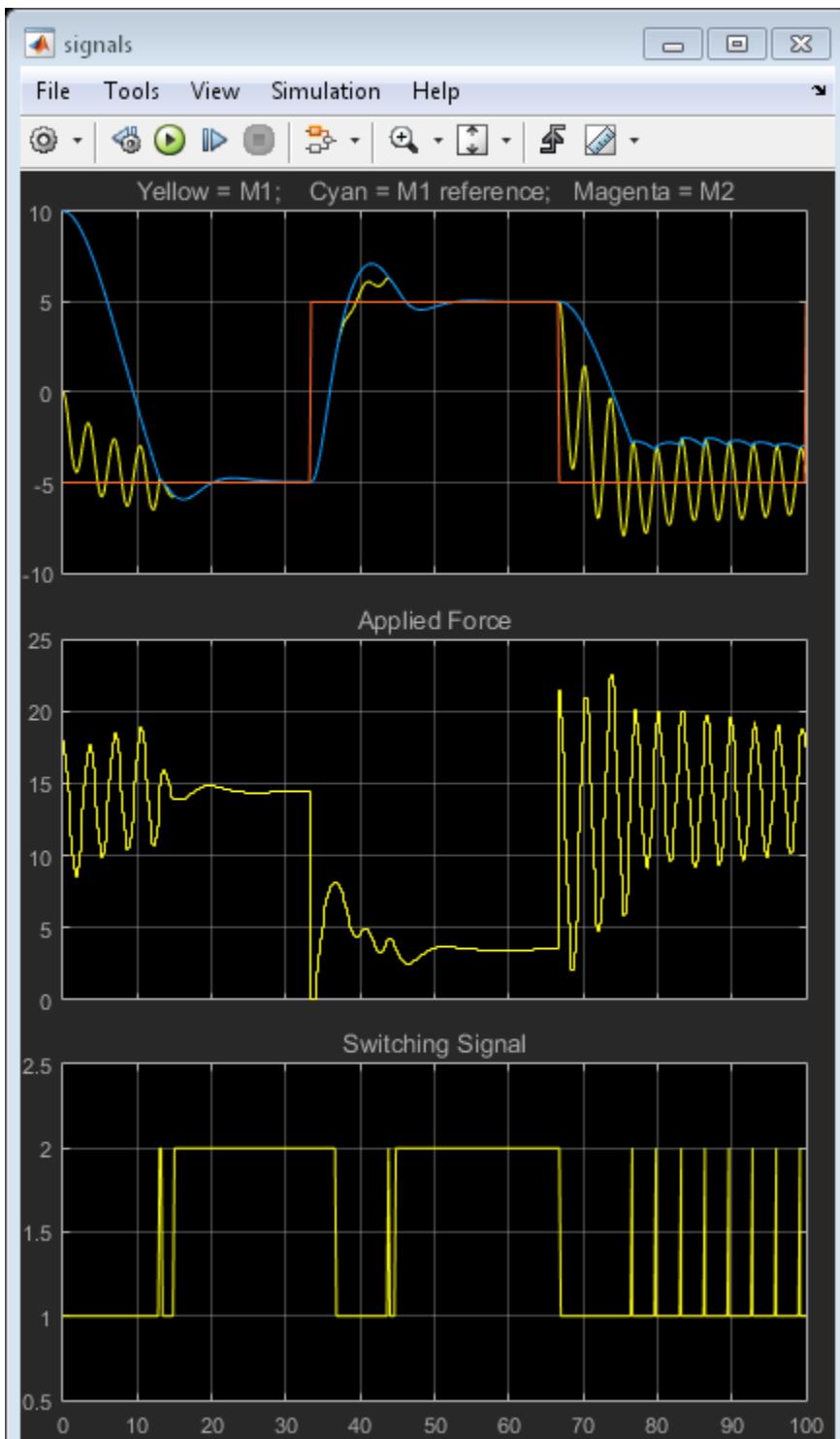


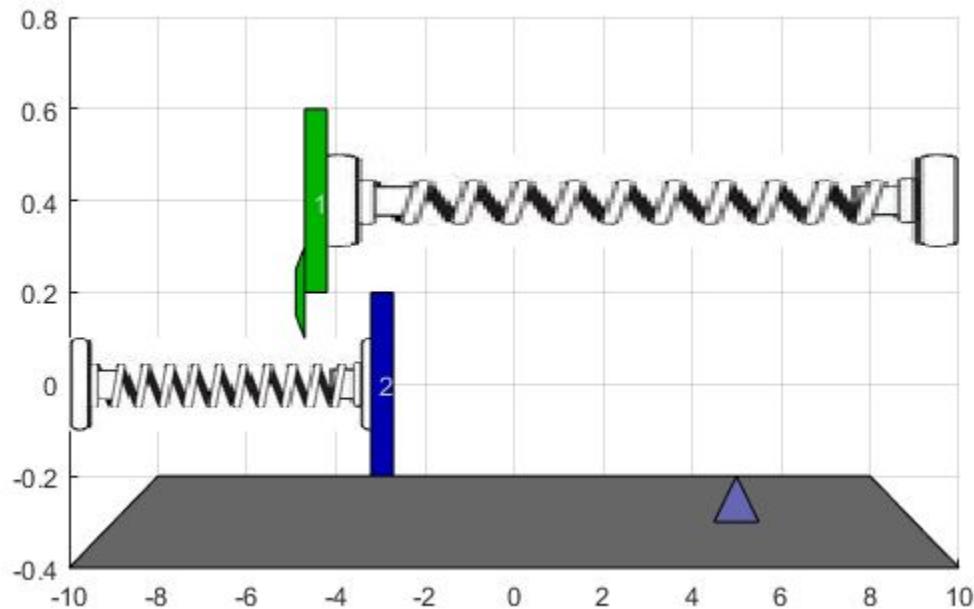
In this case, performance degrades whenever the two masses join.

Repeat Simulation Using MPC2 Only (Assumes Masses Always in Contact)

```
disp( Now repeat simulation by using only MPC2 ... );
disp( When two masses are detached, control performance deteriorates. );
MPC1=MPC2save;
MPC2=MPC1;
sim(md1);
```

Now repeat simulation by using only MPC2 ...
When two masses are detached, control performance deteriorates.





In this case, performance degrades when the masses separate, causing the controller to apply excessive force.

```
bdclose(md1)
close(findobj( Tag , mpc_switching_demo ))
```

Related Examples

- “Schedule Controllers at Multiple Operating Points”
- “Coordinate Multiple Controllers at Different Operating Points” on page 4-64
- “Gain Scheduled MPC Control of Nonlinear Chemical Reactor” on page 7-5

More About

- “Design Workflow for Gain Scheduling” on page 7-3

Gain Scheduled MPC Control of an Inverted Pendulum on a Cart

This example uses a gain scheduled model predictive controller to control an inverted pendulum on a cart.

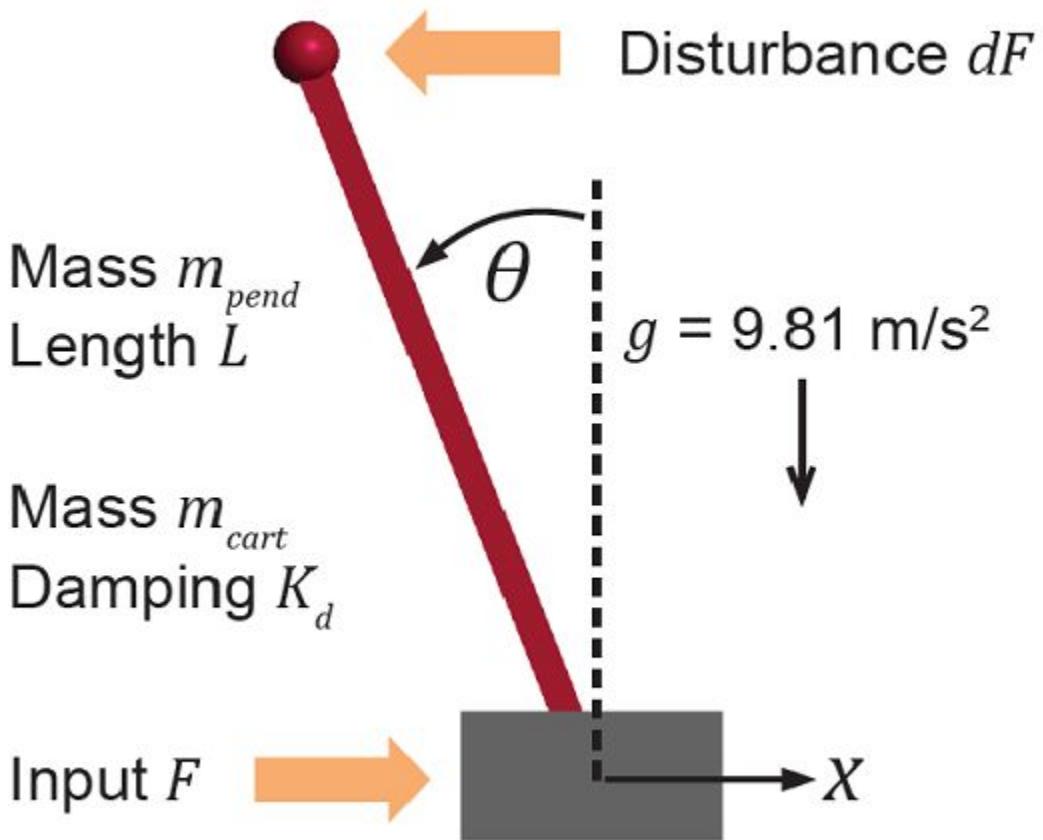
Product Requirement

This example requires Simulink® Control Design™ software to define the MPC structure by linearizing a nonlinear Simulink model.

```
if ~mpcchecktoolboxinstalled( 'slcontrol' )
    disp( 'Simulink Control Design(R) is required to run this example.' )
    return
end
```

Pendulum/Cart Assembly

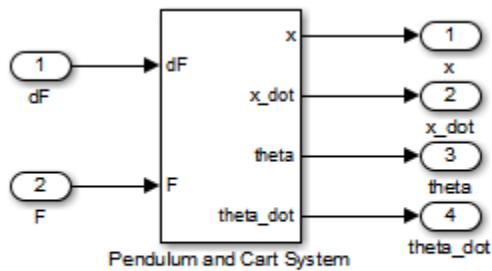
The plant for this example is the following cart/pendulum assembly, where x is the cart position and θ is the pendulum angle.



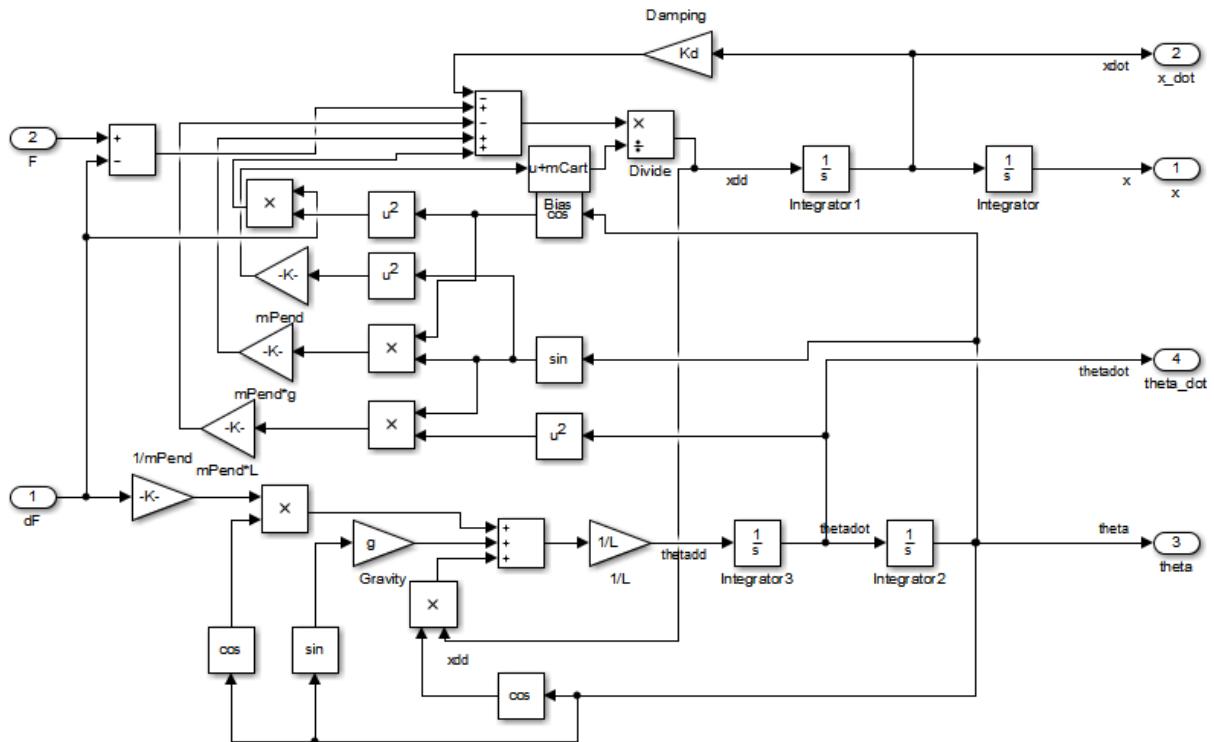
This system is controlled by exerting a variable force F on the cart. The controller needs to keep the pendulum upright while moving the cart to a new position or when the pendulum is nudged forward by an impulse disturbance dF applied at the upper end of the inverted pendulum.

This plant is modeled in Simulink with commonly used blocks.

```
mdlPlant = mpc_pendcartPlant ;
load_system(mdlPlant);
open_system([mdlPlant /Pendulum and Cart System ], force );
```



Copyright 1990-2015 The MathWorks, Inc.



Control Objectives

Assume the following initial conditions for the cart/pendulum assembly:

- The cart is stationary at $x = 0$.
- The inverted pendulum is stationary at the upright position $\theta = 0$.

The control objectives are:

- Cart can be moved to a new position between -15 and 15 with a step setpoint change.
- When tracking such a setpoint change, the rise time should be less than 4 seconds (for performance) and the overshoot should be less than 5 percent (for robustness).
- When an impulse disturbance of magnitude of 2 is applied to the pendulum, the cart should return to its original position with a maximum displacement of 1. The pendulum should also return to the upright position with a peak angle displacement of 15 degrees (0.26 radian).

The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

The Choice of Gain Scheduled MPC

In “Control of an Inverted Pendulum on a Cart”, a single MPC controller is able to move the cart to a new position between -10 and 10. However, if you increase the step setpoint change to 15, the pendulum fails to recover its upright position during the transition.

To reach the longer distance within the same rise time, the controller applies more force to the cart at the beginning. As a result, the pendulum is displaced from its upright position by a larger angle such as 60 degrees. At such angles, the plant dynamics differ significantly from the LTI predictive model obtained at $\theta = 0$. As a result, errors in the prediction of plant behavior exceed what the built-in MPC robustness can handle, and the controller fails to perform properly.

A simple workaround to avoid the pendulum falling is to restrict pendulum displacement by adding soft output constraints to θ and reducing the ECR weight on constraint softening.

```
mpcobj.OV(2).Min = -pi/2;  
mpcobj.OV(2).Max = pi/2;  
mpcobj.Weights.ECR = 100;
```

However, with these new controller settings it is no longer possible to reach the longer distance within the required rise time. In other words, controller performance is sacrificed to avoid violation of soft output constraints.

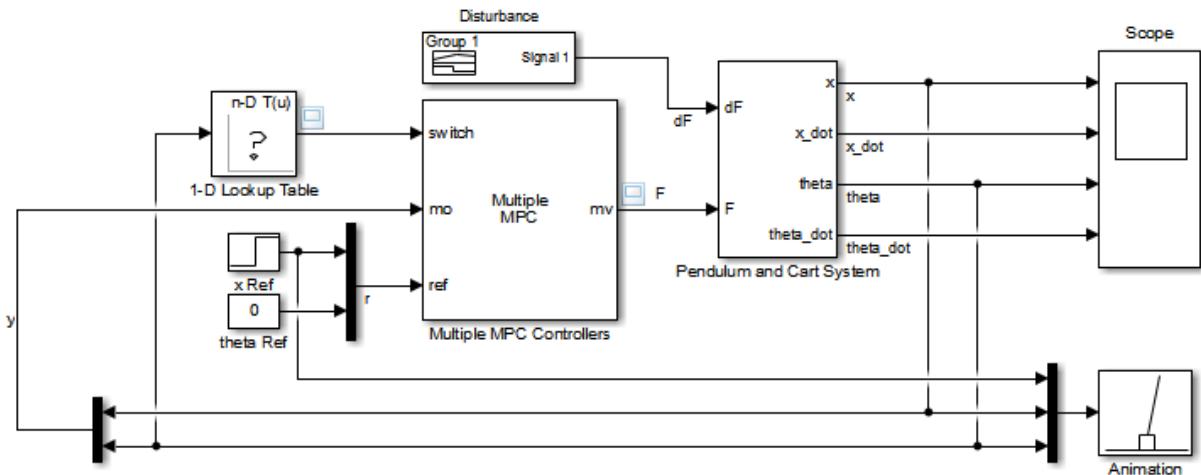
To move the cart to a new position between -15 and 15 while maintaining the same rise time, the controller needs to have more accurate models at different angles so that the controller can use them for better prediction. Gain scheduled MPC allows you to solve a nonlinear control problem by designing multiple MPC controllers at different operating points and switching between them at run time.

Control Structure

For this example, use a single MPC controller with:

- One manipulated variable: Variable force F .
- Two measured outputs: Cart position x and pendulum angle θ .
- One unmeasured disturbance: Impulse disturbance dF .

```
mdlMPC = mpc_pendcartGainSchedulingMPC ;
open_system(md1MPC);
```



Copyright 1990-2015 The MathWorks, Inc.

Although cart velocity x_dot and pendulum angular velocity θ_dot are available from the plant model, to make the design case more realistic, they are excluded as MPC measurements.

While the cart position setpoint varies (step input), the pendulum angle setpoint is constant (0 = upright position).

Linear Plant Model

Since the MPC controller requires a linear time-invariant (LTI) plant model for prediction, linearize the Simulink plant model at three different operating points.

Specify linearization input and output points

```
io(1) = linio([mdlPlant /dF ],1, openinput );
io(2) = linio([mdlPlant /F ],1, openinput );
io(3) = linio([mdlPlant /Pendulum and Cart System ],1, openoutput );
io(4) = linio([mdlPlant /Pendulum and Cart System ],3, openoutput );
```

Create specifications for the following three operating points where both cart and pendulum are stationary:

- Pendulum is horizontal, pointing right ($\theta = -\pi/2$)
- Pendulum is upright ($\theta = 0$)
- Pendulum is horizontal, pointing left ($\theta = \pi/2$)

```
angles = [-pi/2 0 pi/2];
for ct=1:length(angles)
```

create operating point specification

```
opspec(ct) = operspec(mdlPlant);
```

The first state is cart position x .

```
opspec(ct).States(1).Known = true;
opspec(ct).States(1).x = 0;
```

The second state is cart velocity x_dot (not at steady state).

```
opspec(ct).States(2).SteadyState = false;
```

The third state is pendulum angle θ .

```
opspec(ct).States(3).Known = true;
opspec(ct).States(3).x = angles(ct);
```

The forth state is angular velocity θ_dot (not at steady state)

```
opspec(ct).States(4).SteadyState = false;
```

```
end
```

Compute operating point using these specifications.

```
options = findopOptions( DisplayReport ,false);
[op, opresult] = findop(mdlPlant,opspec,options);
```

Obtain the linear plant model at the specified operating points.

```
plants = linearize(mdlPlant,op,io);
bdclose(mdlPlant);
```

Multiple MPC Designs

At each operating point, design an MPC controller with the corresponding linear plant model.

```
status = mpcverbosity( off );
for ct=1:length(angles)
```

Get a single plant model.

```
plant = plants(:,:,ct);
plant.InputName = { dF ; F };
plant.OutputName = { x ; theta };
```

The plant has two inputs, *dF* and *F*, and two outputs, *x* and *theta*. In this example, *dF* is specified as an unmeasured disturbance used by the MPC controller for prediction. Set the plant signal types.

```
plant = setmpcsignals(plant, ud ,1, mv ,2);
```

To control an unstable plant, the controller sample time cannot be too large (poor disturbance rejection) or too small (excessive computation load). Similarly, the prediction horizon cannot be too long (the plant unstable mode would dominate) or too short (constraint violations would be unforeseen). Use the following parameters for this example:

```
Ts = 0.01;
PredictionHorizon = 50;
ControlHorizon = 5;
mpcobj = mpc(plant,Ts,PredictionHorizon,ControlHorizon);
```

Specify nominal input and output values based on the operating point.

```
mpcobj.Model.Nominal.Y = [0;opresult(ct).States(3).x];
mpcobj.Model.Nominal.X = [0;0;opresult(ct).States(3).x;0];
mpcobj.Model.Nominal.DX = [0;opresult(ct).States(2).dx;0;opresult(ct).States(4).dx];
```

There is a limitation on how much force we can apply to the cart, which is specified as hard constraints on manipulated variable F .

```
mpcobj.MV.Min = -200;
mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by 100.

```
mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from 0.1 to 1.

```
mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position x and the second weight is associated with angle θ .

```
mpcobj.Weights.OV = [1.2 1];
```

To achieve more aggressive disturbance rejection, increase the state estimator gain by multiplying the default disturbance model gains by a factor of 10.

Update the input disturbance model.

```
disturbance_model = getindist(mpcobj);
setindist(mpcobj, model ,disturbance_model*10);
```

Update the output disturbance model.

```
disturbance_model = getoutdist(mpcobj);
setoutdist(mpcobj, model ,disturbance_model*10);
```

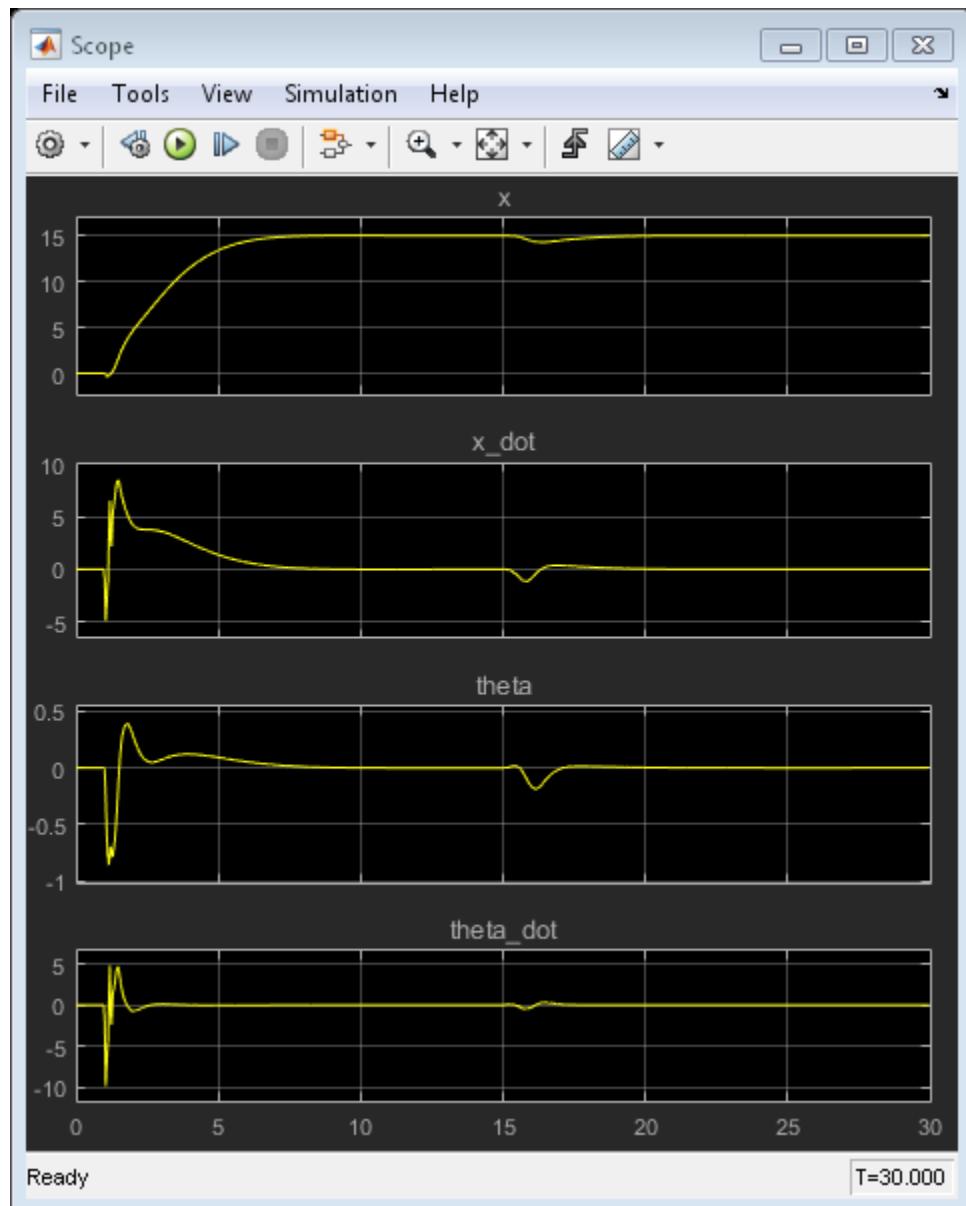
Save the MPC controller to the MATLAB workspace.

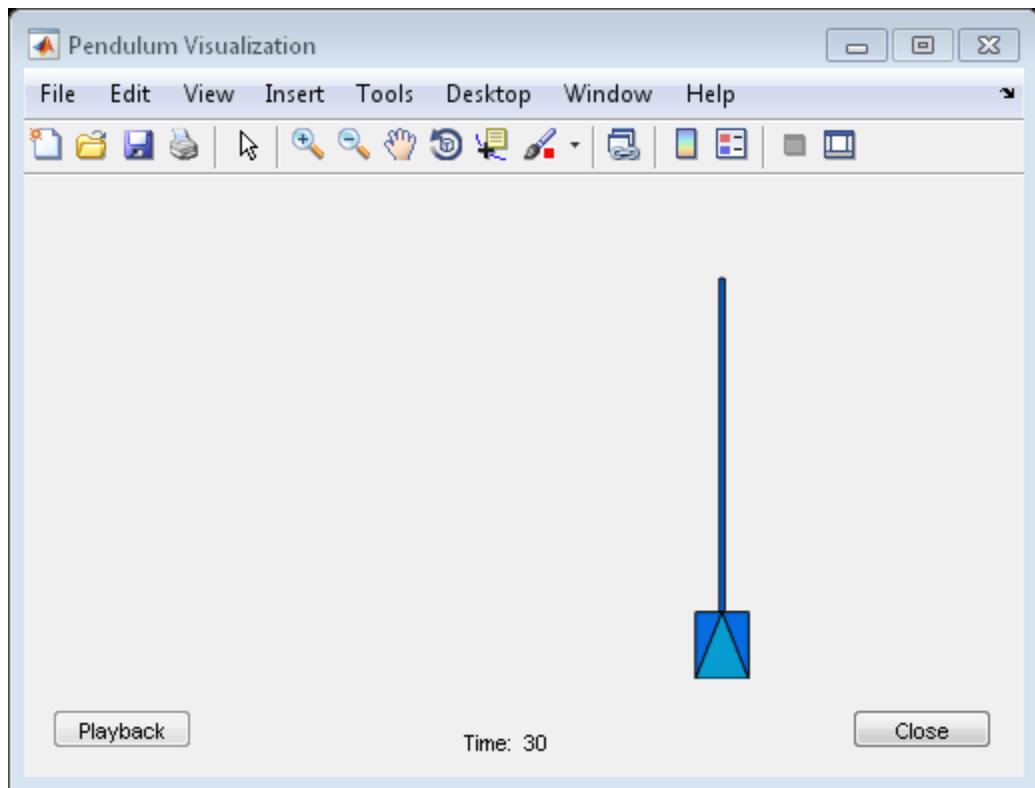
```
assignin( base ,[ mpc num2str(ct)] ,mpcobj);  
end  
mpcverbosity(status);
```

Closed-Loop Simulation

Validate the MPC design with a closed-loop simulation in Simulink.

```
open_system([mdlMPC /Scope ]);  
sim(mdlMPC);  
  
-->Converting model to discrete time.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output.
```





In the nonlinear simulation, all the control objectives are successfully achieved.

```
bdclose(mdlMPC);
```

More About

- “Gain-Scheduled MPC” on page 7-2
- “Control of an Inverted Pendulum on a Cart” on page 4-144
- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-42

Reference for MPC Designer App

This chapter is the reference manual for the Model Predictive Control Toolbox MPC Designer app.

- “Generate MATLAB Code from MPC Designer” on page 8-2
- “Generate Simulink Model from MPC Designer” on page 8-4
- “Compare Multiple Controller Responses Using MPC Designer” on page 8-6

Generate MATLAB Code from MPC Designer

This topic shows how to generate MATLAB code for creating and simulating model predictive controllers designed in the MPC Designer app. Generated MATLAB scripts are useful when you want to programmatically reproduce designs that you obtained interactively.

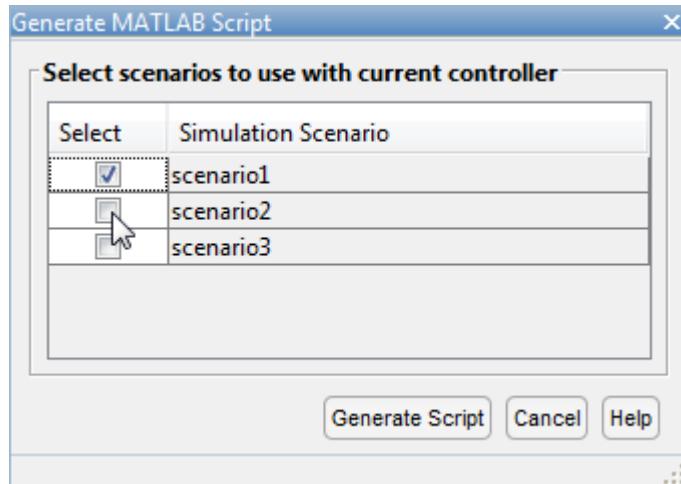
To create a MATLAB script:

- 1 In the MPC Designer app, interactively design and tune your model predictive controller.
- 2 On the **Tuning** tab, in the **Analysis** section, click the **Export Controller** arrow ▾.

Alternatively, **Export Controller** is on the **MPC Designer** tab, in the **Result** section.

Note: If you opened MPC Designer from Simulink, click the **Update and Simulate** arrow ▾.

- 3 Under **Export Controller** or **Update and Simulate**, click **Generate Script** .
- 4 In the Generate MATLAB Script dialog box, select one or more simulation scenarios to include in the generated script.



- 5 Click **Generate Script** to create the MATLAB script for creating the current MPC controller and running the selected simulation scenarios. The generated script opens in the MATLAB Editor.

In addition to generating a script, the app exports the following to the MATLAB workspace:

- A copy of the plant used to create the controller, that is the controller internal plant model
- Copies of the plants used in any simulation scenarios that do not use the default internal plant model
- The reference and disturbance signals specified for each simulation scenario

See Also

`mpc`

Related Examples

- “Generate Simulink Model from MPC Designer” on page 8-4

Generate Simulink Model from MPC Designer

This topic shows how to generate a Simulink model that uses the current model predictive controller to control its internal plant model.

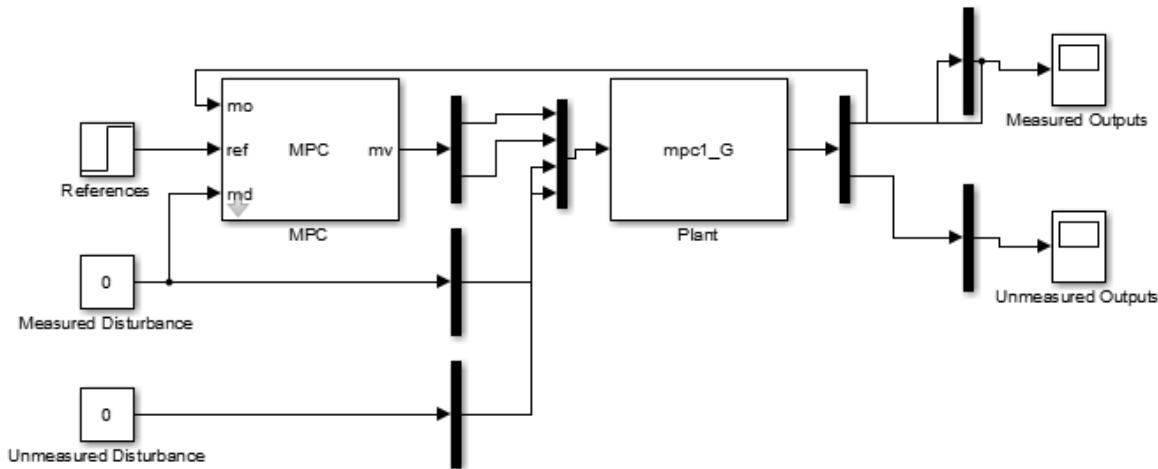
To create a Simulink model:

- 1 In the MPC Designer app, interactively design and tune your model predictive controller.
- 2 On the **Tuning** tab, in the **Analysis** section, click the **Export Controller** arrow ▾.

Alternatively, **Export Controller** is on the **MPC Designer** tab, in the **Result** section.

- 3

Under **Export Controller**, click **Generate Simulink Model** .



The app exports the current MPC controller and its internal plant model to the MATLAB workspace and creates a Simulink model that contains an **MPC Controller** block and a **Plant** block.

Also, default step changes in the output setpoints are added to the **References** block.

Use the generated model to validate your controller design. The generated model serves as a template for moving easily from the MATLAB design environment to the Simulink environment.

You can also use the Simulink model to generate code and deploy it for real-time control applications. For more information, see “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5.

See Also

[MPC Controller](#) | [MPC Designer](#)

Related Examples

- “Generate MATLAB Code from MPC Designer” on page 8-2

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 3-5
- “Design MPC Controller in Simulink”

Compare Multiple Controller Responses Using MPC Designer

This example shows how to compare multiple controller responses using the MPC Designer app. In particular, controllers with different output constraint configuration are compared.

Define Plant Model

Create a state-space model of your plant, and specify the MPC signal types.

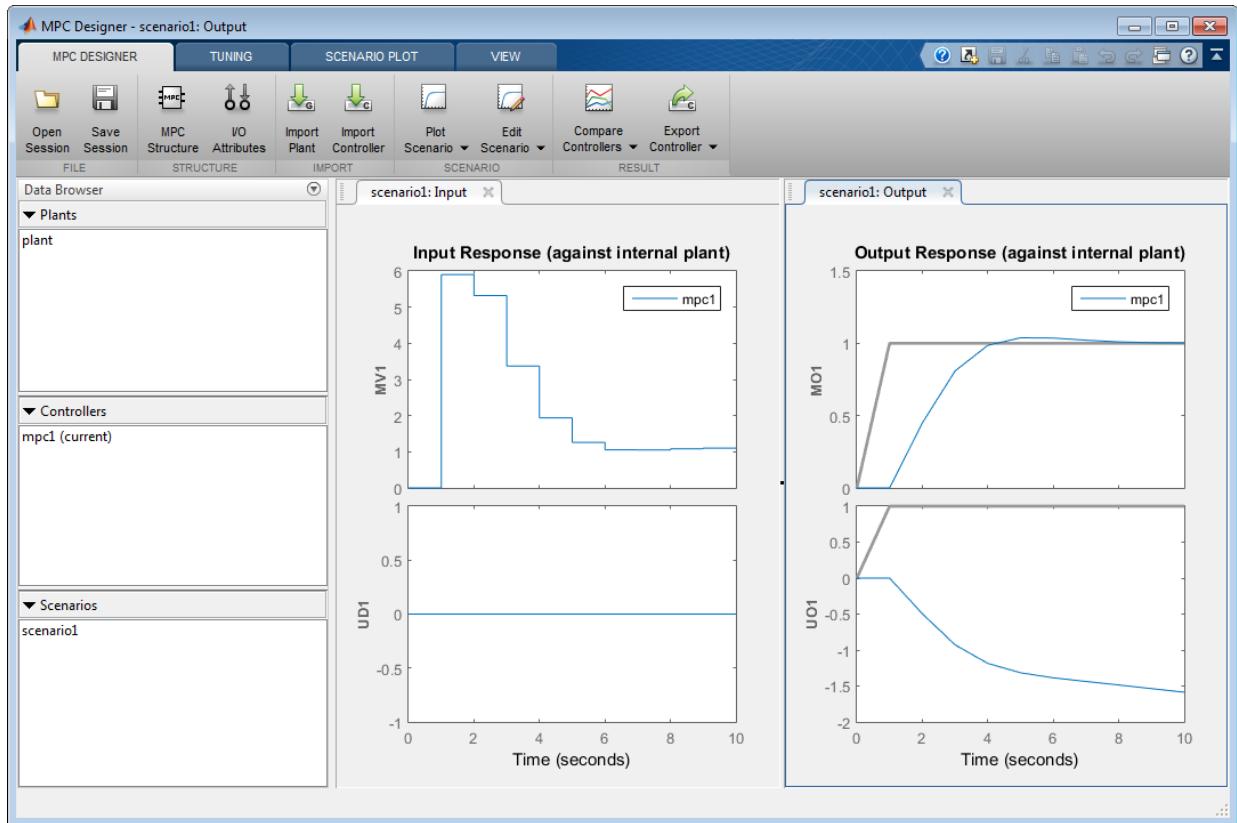
```
A = [-0.0285 -0.0014; -0.0371 -0.1476];
B = [-0.0850 0.0238; 0.0802 0.4462];
C = [0 1; 1 0];
D = zeros(2,2);

plant = ss(A,B,C,D);
plant = setmpcsignals(plant, MV ,1, UD ,2, MO ,1, UO ,2);
```

Open MPC Designer App

Open the MPC Designer app, and import the plant model.

```
mpcDesigner(plant)
```



The app adds the specified plant to the **Data Browser** along with a default controller, **mpc1**, and a default simulation scenario, **scenario1**.

Define Simulation Scenario

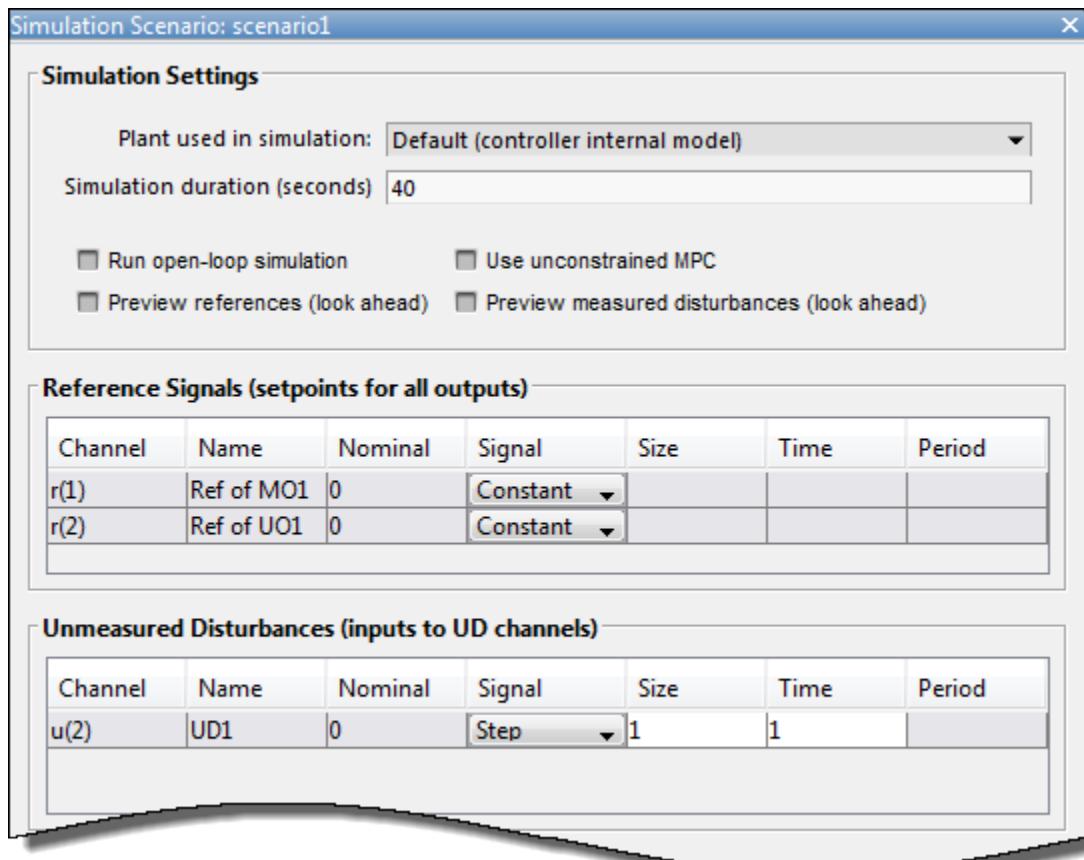
Configure a disturbance rejection simulation scenario.

In the MPC Designer app, on the MPC Designer tab, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, specify a **Simulation duration** of 40 seconds.

In the **Reference Signals** table, in the **Signal** drop-down lists, select **Constant** to hold the setpoints of both outputs at their nominal values.

In the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step**. Use the default **Time** and **Step** values.



This scenario simulates a unit step change in the unmeasured input disturbance at a time of 1 second.

Click **OK**.

The app runs the updated simulation scenario and updates the controller response plots. In the **Output Response** plots, the default controller returns the measured output, **MO1**, to its nominal value, however the control action causes an increase in the unmeasured output, **UO1**.

Create Controller with Hard Output Constraints

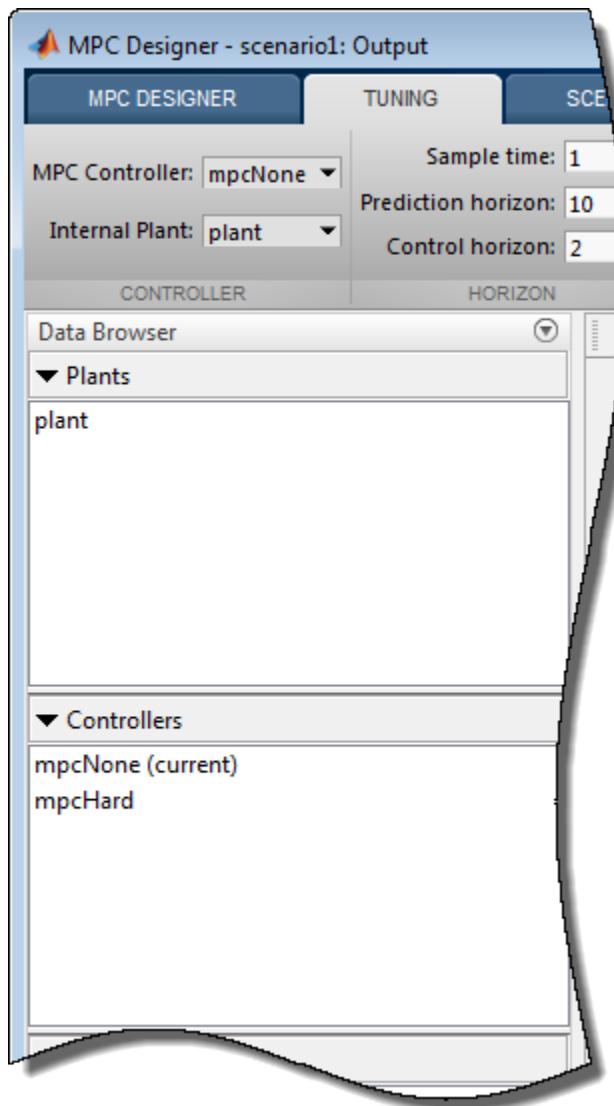
Suppose that the control specifications indicate that such an increase in the unmeasured disturbance is undesirable. To limit the effect of the unmeasured disturbance, create a controller with a hard output constraint.

Note: In practice, using hard output constraints is not recommended. Such constraints can create an infeasible optimization problem when the output variable moves outside of the constraint bounds due to a disturbance.

In the **Data Browser**, in the **Controllers** section, right-click `mpc1`, and select **Copy**.

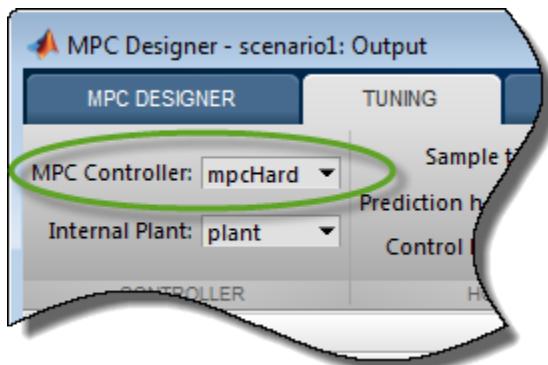
The app creates a copy of the default controller and adds it to the **Data Browser**.

Double-click each controller and rename them as follows.



Right-click the `mpcHard` controller, and select **Tune (make current)**. The app adds the `mpcHard` controller response to the **Input Response** and **Output Response** plots.

On the **Tuning** tab, in the **Controller** section, `mpcHard` is selected as the current **MPC Controller** being tuned.

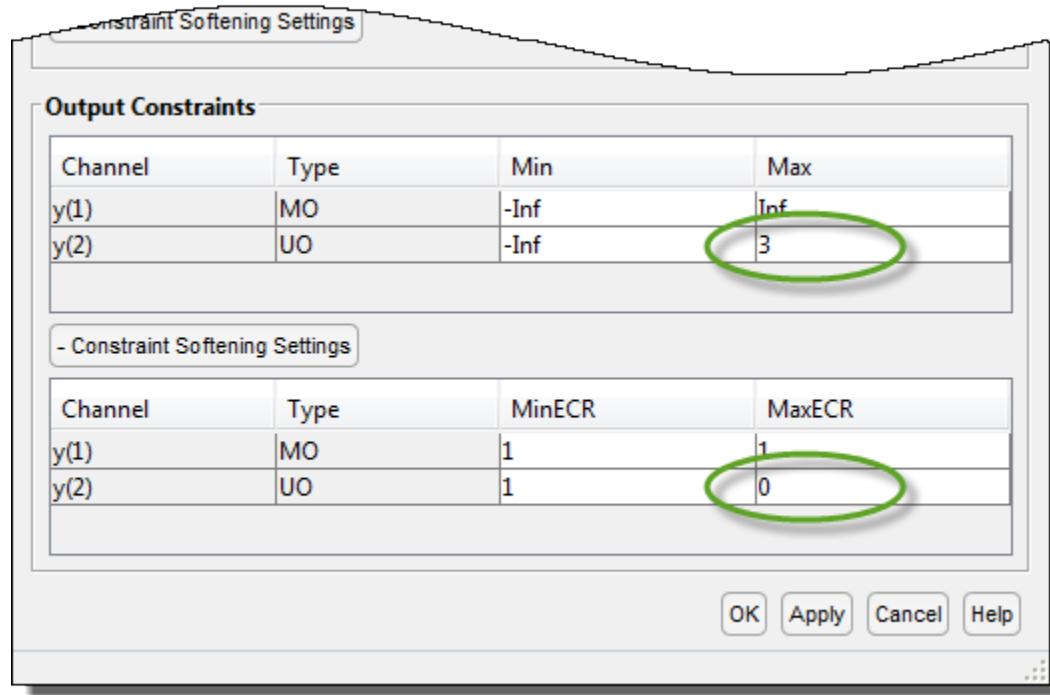


In the **Design** section, click **Constraints**.

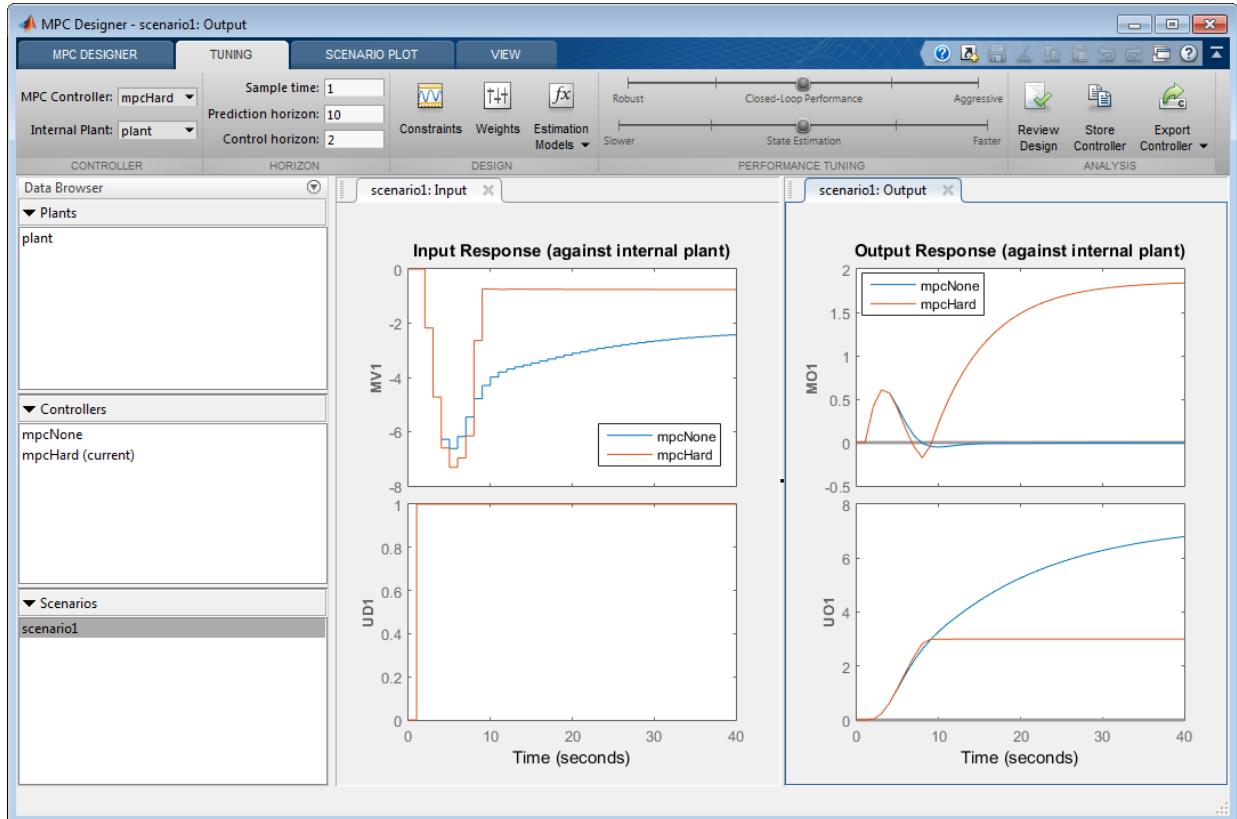
In the Constraints dialog box, in the **Output Constraints** section, in the **Max** column, specify a maximum output constraint of **3** for the unmeasured output (UO).

By default, all output constraints are soft, that is the controller can allow violations of the constraint when computing optimal control moves.

To make the unmeasured output constraint hard, click **Constraint Softening Settings**, and enter a **MaxECR** value of **0** for the UO. This setting places a strict limit on the controller output that cannot be violated.



Click **OK**.



The response plots update to reflect the new **mpcHard** configuration. In the **Output Response** plot, in the **UO1** plot, the **mpcHard** response is limited to a maximum of 3. As a trade-off, the controller cannot return the **MO1** response to its nominal value.

Tip If the plot legends are blocking the response signals, you can drag the legends to different locations.

Create Controller with Soft Output Constraints

Suppose the deviation of **MO1** from its nominal value is too large. You can soften the output constraint for a compromise between the two control objectives: **MO1** output tracking and **UO1** constraint satisfaction.

On the **Tuning** tab, in the **Analysis** section, click **Store Controller** to save a copy of `mpcHard` in the **Data Browser**.

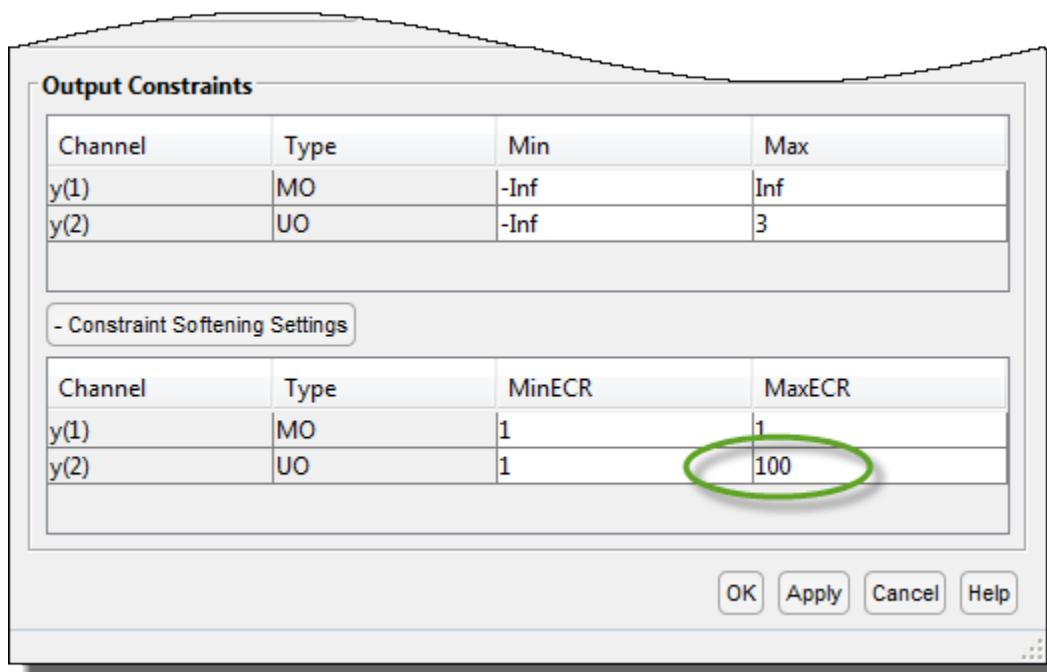
In the **Data Browser**, in the **Controllers** section, rename `mpcHard_Copy` to `mpcSoft`.

On the **Tuning** tab, in the **Controller** section, in the **MPC Controller** drop-down list, select `mpcSoft` as the current controller.

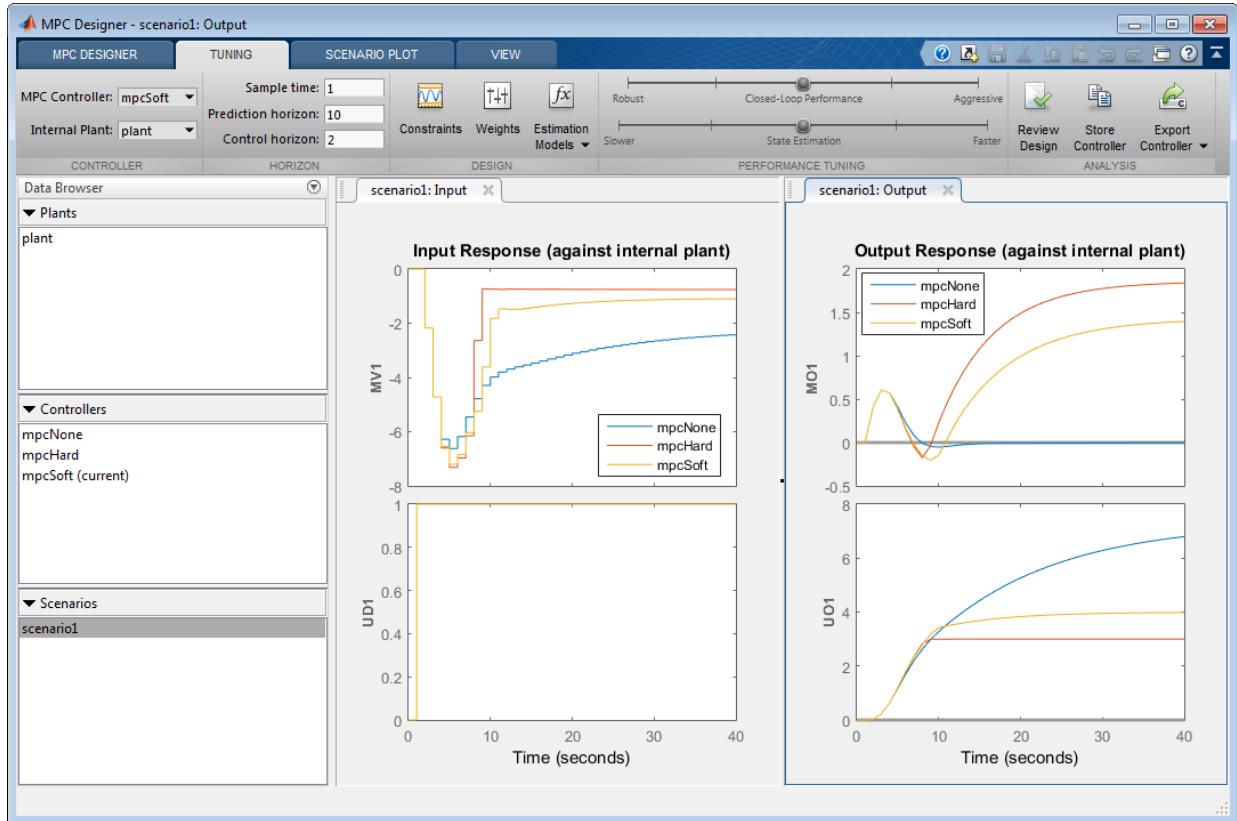
The app adds the `mpcSoft` controller response to the **Input Response** and **Output Response** plots.

In the **Design** section, click **Constraints**.

In the Constraints dialog box, in the **Output Constraints** section, enter a **MaxECR** value of 100 for the UO to soften the constraint.



Click **OK**.

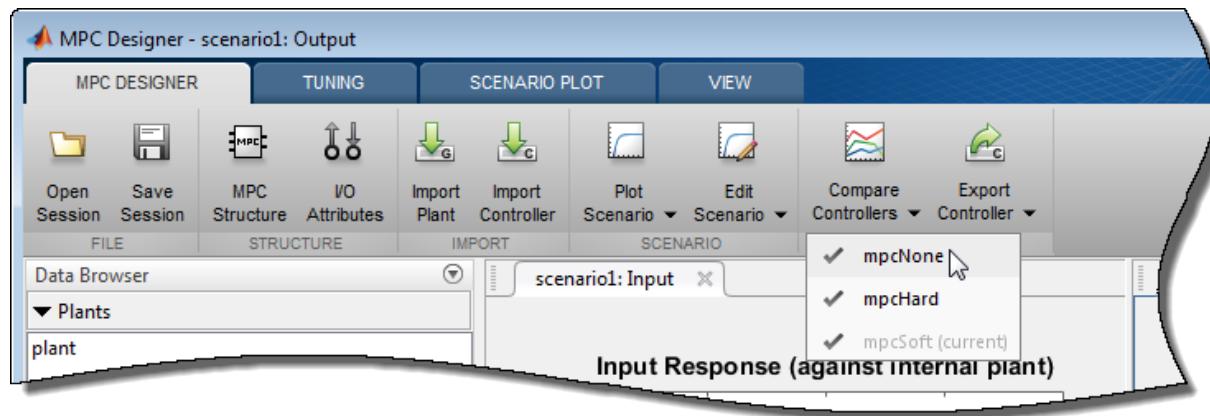


The response plots update to reflect the new `mpcSoft` configuration. In the **Output Response** plot, `mpcSoft` shows a compromise between the previous controller responses.

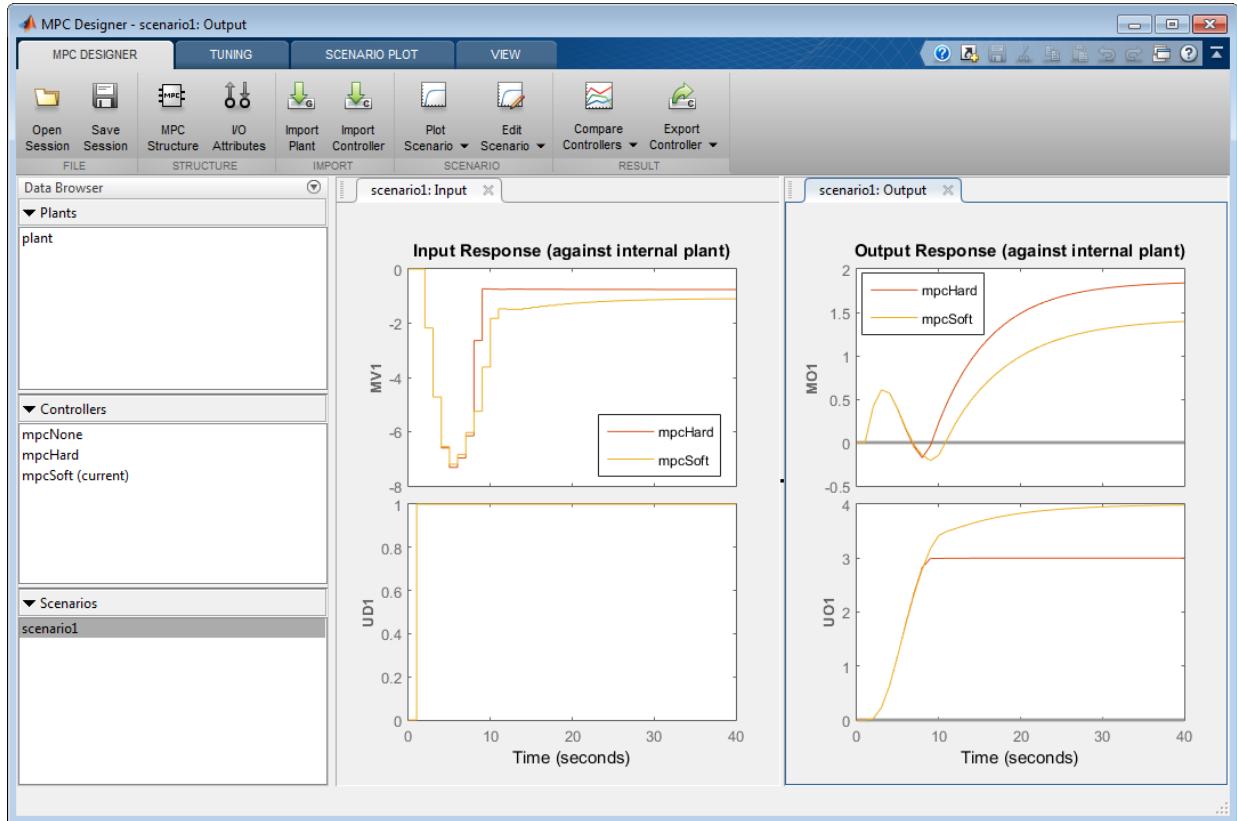
Remove Default Controller Response Plot

To compare the two constrained controllers only, you can remove the default unconstrained controller from the input and output response plots.

On the **MPC Designer** tab, in the **Result** section, click **Compare Controllers > mpcNone**.



The app removes the `mpcNone` responses from the **Input Response** and **Output Response** plots.



You can toggle the display of any controller in the **Data Browser** except for controller currently being tuned. Under **Compare Controllers**, the controllers with displayed responses are indicated with check marks.

See Also

MPC Designer

Related Examples

- “Design Controller Using MPC Designer”
- “Design MPC Controller in Simulink”

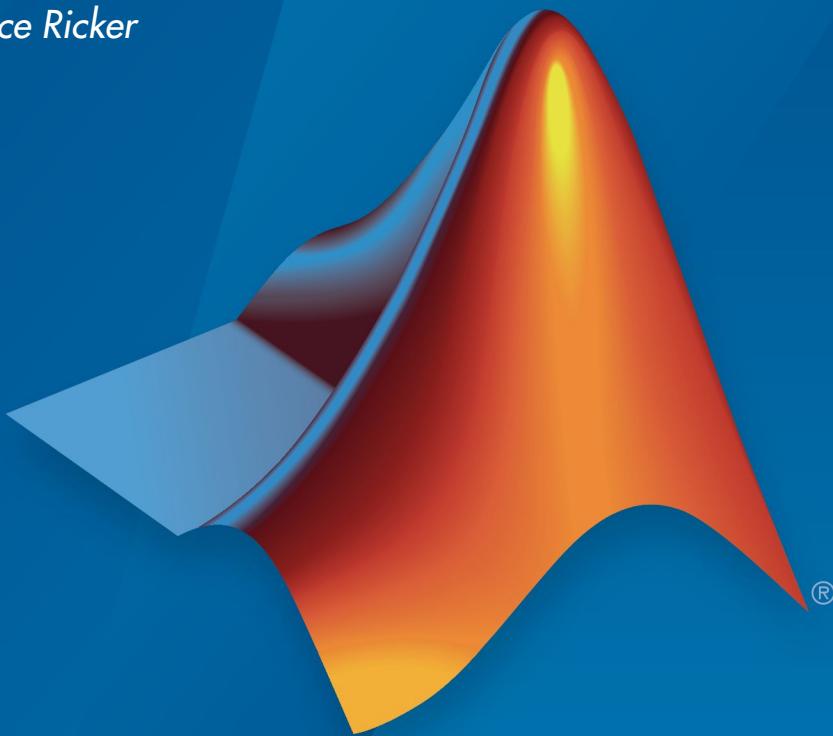
More About

- “Specifying Constraints” on page 1-10

Model Predictive Control Toolbox™

Reference

*Alberto Bemporad
Manfred Morari
N. Lawrence Ricker*



MathWorks®

R2016a

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

Model Predictive Control Toolbox™ Reference

© COPYRIGHT 2005–2016 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to 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

October 2004	First printing	New for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.2.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)
March 2010	Online only	Revised for Version 3.2 (Release 2010a)
September 2010	Online only	Revised for Version 3.2.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.1.3 (Release R2013b)
March 2014	Online only	Revised for Version 4.2 (Release R2014a)
October 2014	Online only	Revised for Version 5.0 (Release R2014b)
March 2015	Online only	Revised for Version 5.0.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.1 (Release 2015b)
March 2016	Online only	Revised for Version 5.2 (Release 2016a)

Functions – Alphabetical List

1

Block Reference

2

Object Reference

3

MPC Controller Object	3-2
ManipulatedVariables	3-2
OutputVariables	3-4
DisturbanceVariables	3-5
Weights	3-5
Model	3-7
Ts	3-9
Optimizer	3-9
PredictionHorizon	3-11
ControlHorizon	3-11
History	3-11
Notes	3-11
UserData	3-11
Construction and Initialization	3-12
MPC Simulation Options Object	3-13
MPC State Object	3-16

Explicit MPC Controller Object	3-18
Properties	3-18

Functions – Alphabetical List

cloffset

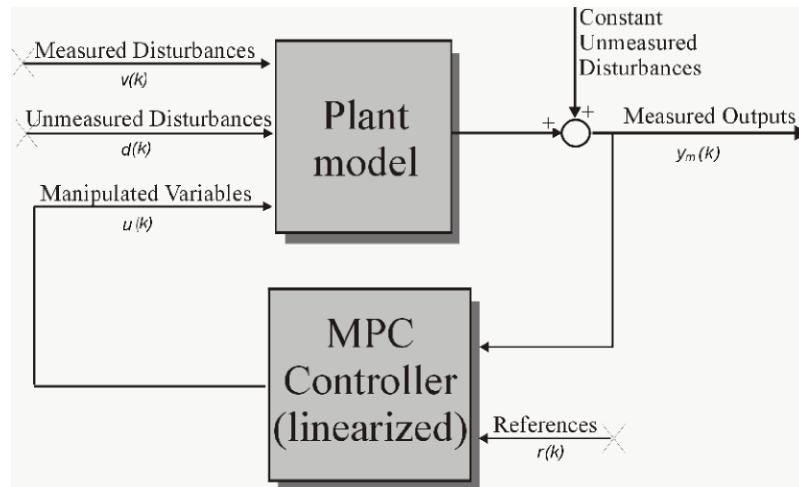
Compute MPC closed-loop DC gain from output disturbances to measured outputs assuming constraints are inactive at steady state

Syntax

```
DCgain = cloffset(MPCobj)
```

Description

The `cloff` function computes the DC gain from output disturbances to measured outputs, assuming constraints are not active, based on the feedback connection between `Model`.`Plant` and the linearized MPC controller, as depicted below.



Computing the Effect of Output Disturbances

By superposition of effects, the gain is computed by zeroing references, measured disturbances, and unmeasured input disturbances.

`DCgain = cloffset(MPCobj)` returns an n_{y_m} -by- n_{y_m} DC gain matrix `DCgain`, where n_{y_m} is the number of measured plant outputs. `MPCobj` is the MPC object specifying the

controller for which the closed-loop gain is calculated. `DCgain(i,j)` represents the gain from an additive (constant) disturbance on output `j` to measured output `i`. If row `i` contains all zeros, there will be no steady-state offset on output `i`.

See Also

`mpc` | `ss`

Related Examples

- “Compute Steady-State Gain”

Introduced before R2006a

compare

Compare two MPC objects

Syntax

```
yesno = compare(MPC1,MPC2)
```

Description

The compare function compares the contents of two MPC objects **MPC1**, **MPC2**. If the design specifications (models, weights, horizons, etc.) are identical, then **yesno** is equal to 1.

Note `compare` may return `yesno = 1` even if the two objects are not identical. For instance, **MPC1** may have been initialized while **MPC2** may have not, so that they may have different sizes in memory. In any case, if `yesno = 1`, the behavior of the two controllers will be identical.

See Also

`mpc`

Introduced before R2006a

d2d

Change MPC controller sample

Syntax

```
MPCobj = d2d(MPCobj,Ts)
```

Description

The **d2d** function changes the sample time of the MPC controller **MPCobj** to **Ts**. All models are sampled or resampled as soon as the QP matrices must be computed, for example when **sim** or **mpcmove** are called.

See Also

mpc | **set**

Introduced before R2006a

generateExplicitMPC

Convert implicit MPC controller to explicit MPC controller

Given a traditional Model Predictive Controller design in the implicit form, convert it to the explicit form for real-time applications requiring fast sample time.

Syntax

```
EMPCobj = generateExplicitMPC(MPCobj,range)
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

Description

`EMPCobj = generateExplicitMPC(MPCobj,range)` converts a traditional (implicit) MPC controller to the equivalent explicit MPC controller, using the specified parameter bounds. This calculation usually requires significant computational effort because a multi-parametric quadratic programming problem is solved during the conversion.

`EMPCobj = generateExplicitMPC(MPCobj,range,opt)` converts the MPC controller using additional optimization options.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;
```

```

p = 10;
m = 3;
MPCobj = mpc(plant,Ts,p,m);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default

```

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```

range = generateExplicitRange(MPCobj);

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;

```

Use the more robust reduction method for the computation. Use **generateExplicitOptions** to create a default options set, and then modify the **polyreduction** option.

```

opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;

```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

Explicit MPC Controller

```

-----
Controller sample time: 0.1 (seconds)
Polyhedral regions: 1
Number of parameters: 4
Is solution simplified: No
State Estimation: Default Kalman gain

```

Type `EMPCobj.MPC` for the original implicit MPC design.
Type `EMPCobj.Range` for the valid range of parameters.
Type `EMPCobj.OptimizationOptions` for the options used in multi-parametric QP computation.
Type `EMPCobj.PiecewiseAffineSolution` for regions and gain in each solution.

- “Explicit MPC Control of a Single-Input-Single-Output Plant”
- “Explicit MPC Control of an Aircraft with Unstable Poles”
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output”

Input Arguments

MPCobj — Traditional MPC controller

MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

range — Parameter bounds

structure

Parameter bounds, specified as a structure that you create with the `generateExplicitRange` command. This structure specifies the bounds on the parameters upon which the explicit MPC control law depends, such as state values, measured disturbances, and manipulated variables. See `generateExplicitRange` for detailed descriptions of these parameters.

opt — optimization options

structure

Optimization options for the conversion computation, specified as a structure that you create with the `mpcExplicitOptions` command. See `generateExplicitOptions` for detailed descriptions of these options.

Output Arguments

EMPCobj — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller that is equivalent to the input traditional controller, returned as an explicit MPC controller object. The properties of the explicit MPC controller object are summarized in the following table.

Property	Description
MPC	Traditional (implicit) controller object used to generate the explicit MPC controller. You create this MPC controller using the <code>mpc</code> command. It is the first argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See “MPC Controller Object” on page 3-2 or type <code>mpcprops</code> for details regarding the properties of the MPC controller.
Range	1-D structure containing the parameter bounds used to generate the explicit MPC controller. These determine the resulting controller’s valid operating range. This property is automatically populated by the <code>range</code> input argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See <code>generateExplicitRange</code> for details about this structure.
OptimizationOptions	1-D structure containing user-modifiable options used to generate the explicit MPC controller. This property is automatically populated by the <code>opt</code> argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See <code>generateExplicitOptions</code> for details about this structure.
PiecewiseAffineSolution	n_r -dimensional structure, where n_r is the number of piecewise affine (PWA) regions required to represent the control law. The i th element contains the details needed to compute the optimal manipulated variables

Property	Description
	when the solution lies within the <i>i</i> th region. See “Implementation”.
IsSimplified	Logical switch indicating whether the explicit control law has been modified using the simplify command such that the explicit control law approximates the base (implicit) MPC controller. If the control law has not been modified, the explicit controller should reproduce the base controller’s behavior exactly, provided both operate within the bounds described by the Range property.

More About

Tips

- Using Explicit MPC, you will most likely achieve best performance in small control problems, which involve small numbers of plant inputs/outputs/states as well as the number of constraints.
- Test the implicit controller thoroughly before attempting a conversion. This helps to determine the range of controller states and other parameters needed to generate the explicit controller.
- Simulate the explicit controller’s performance using the **sim** or **mpcmoveExplicit** commands, or the **Explicit MPC Controller** block in Simulink®.
- **generateExplicitMPC** displays progress messages in the command window. Use **mpcverbosity** to turn off the display.
- “Explicit MPC”
- “Design Workflow for Explicit MPC”

See Also

`generateExplicitOptions` | `generateExplicitRange` | `mpc` | `simplify`

Introduced in R2014b

generateExplicitOptions

Optimization options for explicit MPC generation

Syntax

```
opt = generateExplicitOptions(MPCobj)
```

Description

`opt = generateExplicitOptions(MPCobj)` creates a set of options to use when converting a traditional MPC controller, `MPCobj`, to explicit form using `generateExplicitMPC`. The options set is returned with all options set to default values. Use dot notation to modify the options.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;  
p = 10;  
m = 3;  
MPCobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(MPCobj);

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use **generateExplicitOptions** to create a default options set, and then modify the **polyreduction** option.

```
opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

```
Explicit MPC Controller
```

```
-----
```

```
Controller sample time: 0.1 (seconds)
```

```
Polyhedral regions: 1
```

```
Number of parameters: 4
```

```
Is solution simplified: No
```

```
State Estimation: Default Kalman gain
```

```
-----
```

```
Type EMPCobj.MPC for the original implicit MPC design.
```

```
Type EMPCobj.Range for the valid range of parameters.
```

```
Type EMPCobj.OptimizationOptions for the options used in multi-parametric QP computat
```

Type `EMPCobj.PiecewiseAffineSolution` for regions and gain in each solution.

Input Arguments

MPCobj — Traditional MPC controller

MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

Output Arguments

opt — Options for generating explicit MPC controller

structure

Options for generating explicit MPC controller, returned as a structure. When you create the structure, all the options are set to default values. Use dot notation to modify any options you want to change. The fields and their default values are as follows.

zerotol — Zero-detection tolerance

$1e-8$ (default) | positive scalar value

Zero-detection tolerance used by the NNLS solver, specified as a positive scalar value.

removetol — Redundant-inequality-constraint detection tolerance

$1e-4$ (default) | positive scalar value

Redundant-inequality-constraint detection tolerance, specified as a positive scalar value.

flattol — Flat region detection tolerance

$1e-5$ (default) | positive scalar value

Flat region detection tolerance, specified as a positive scalar value.

normalizetol — Constraint normalization tolerance

0.01 (default) | positive scalar value

Constraint normalization tolerance, specified as a positive scalar value.

maxiterNNLS – Maximum number of NNLS solver iterations

500 (default) | positive integer

Maximum number of NNLS solver iterations, specified as a positive integer.

maxiterQP – Maximum number of QP solver iterations

200 (default) | positive integer

Maximum number of QP solver iterations, specified as a positive integer.

maxiterBS – Maximum number of bisection method iterations

100 (default) | positive integer

Maximum number of bisection method iterations used to detect region flatness, specified as a positive integer.

polyreduction – Method for removing redundant inequalities

2 (default) | 1

Method used to remove redundant inequalities, specified as either 1 (robust) or 2 (fast).

See Also

`generateExplicitMPC`

Introduced in R2014b

generateExplicitRange

Bounds on explicit MPC control law parameters

Syntax

```
Range = generateExplicitRange(MPCobj)
```

Description

`Range = generateExplicitRange(MPCobj)` creates a structure of parameter bounds based upon a traditional (implicit) MPC controller object. The range structure is intended for use as an input argument to `generateExplicitMPC`. Usually, the initial range values returned by `generateExplicitRange` are not suitable for generating an explicit MPC controller. Therefore, use dot notation to set the values of the range structure before calling `generateExplicitMPC`.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;  
p = 10;  
m = 3;  
MPCobj = mpc(plant,Ts,p,m);
```

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau

```
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default value 1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default value 1
```

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(MPCobj);

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.

range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use **generateExplicitOptions** to create a default options set, and then modify the **polyreduction** option.

```
opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

Explicit MPC Controller

```
Controller sample time: 0.1 (seconds)
```

```
Polyhedral regions: 1
```

```
Number of parameters: 4
```

```
Is solution simplified: No
```

```
State Estimation: Default Kalman gain
```

```
Type EMPCobj.MPC for the original implicit MPC design.
```

```
Type EMPCobj.Range for the valid range of parameters.
```

```
Type EMPCobj.OptimizationOptions for the options used in multi-parametric QP computation.
```

Type `EMPCobj.PiecewiseAffineSolution` for regions and gain in each solution.

Input Arguments

MPCobj — Traditional MPC controller

MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

Output Arguments

Range — Parameter bounds

structure

Parameter bounds for generating an explicit MPC controller from `MPCobj`, returned as a structure.

Initially, each parameter's minimum and maximum bounds are identical. All such parameters are considered fixed. When you generate an explicit controller, any fixed parameters must be constant when the controller operates. This is unlikely to happen in general. Thus, you must specify valid bounds for all parameters. Use dot notation to set the values of the range structure as appropriate for your system.

The fields of the range structure are as follows.

State — Bounds on controller state values

structure

Bounds on controller state values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_x , where n_x is the number of controller states. `Range.State.Min` and `Range.State.Max` contain the minimum and maximum values, respectively, of all controller states. For example, suppose you are designing a two-state controller. You have determined that the range of the first controller state is $[-1000, 1000]$, and that of the second controller state is $[0, 2\pi]$. Set these bounds as follows:

```
Range.State.Min(:) = [-1000,0];
```

```
Range.State.Max(:) = [1000,2*pi];
```

MPC controller states include states from plant model, disturbance model, and noise model, in that order. Setting the range of a state variable is sometimes difficult when a state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

Reference – Bounds on controller reference signal values

structure

Bounds on controller reference signal values, specified as a structure containing fields **Min** and **Max**. Each of **Min** and **Max** is a vector of length n_y , where n_y is the number of plant outputs. **Range.Reference.Min** and **Range.Reference.Max** contain the minimum and maximum values, respectively, of all reference signal values. For example, suppose you are designing a controller for a two-output plant. You have determined that the range of the first plant output is $[-1000, 1000]$, and that of the second plant output is $[0, 2\pi]$. Set these bounds as follows:

```
Range.Reference.Min(:) = [-1000,0];
Range.Reference.Max(:) = [1000,2*pi];
```

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate the explicit MPC controller must be at least as large as the practical range.

MeasuredDisturbance – Bounds on measured disturbance values

structure

Bounds on measured disturbance values, specified as a structure containing fields **Min** and **Max**. Each of **Min** and **Max** is a vector of length n_{md} , where n_{md} is the number of measured disturbances. If your system has no measured disturbances, leave the generated values of this field unchanged.

Range.MeasuredDisturbance.Min and **Range.MeasuredDisturbance.Max** contain the minimum and maximum values, respectively, of all measured disturbance signals. For example, suppose you are designing a controller for a system with two measured disturbances. You have determined that the range of the first disturbance is $[-1, 1]$, and that of the second disturbance is $[0, 0.1]$. Set these bounds as follows:

```
Range.Reference.Min(:) = [-1,0];
Range.Reference.Max(:) = [1,0.1];
```

Usually you know the practical range of the measured disturbance signals being used at the nominal operating point in the plant. The ranges used to generate the explicit MPC controller must be at least as large as the practical range.

ManipulatedVariable — Bounds on manipulated variable values

structure

Bounds on manipulated variable values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_u , where n_u is the number of manipulated variables. `Range.ManipulatedVariable.Min` and `Range.ManipulatedVariable.Max` contain the minimum and maximum values, respectively, of all manipulated variables. For example, suppose your system has two manipulated variables. The range of the first manipulated variable is $[-1, 1]$, and that of the second variable is $[0, 0.1]$. Set these bounds as follows:

```
Range.ManipulatedVariable.Min(:) = [-1,0];  
Range.ManipulatedVariable.Max(:) = [1,0.1];
```

If manipulated variables are constrained, the ranges used to generate the explicit MPC controller must be at least as large as these limits.

See Also

[generateExplicitMPC](#) | [generateExplicitOptions](#) | [mpc](#)

Introduced in R2014b

generatePlotParameters

Parameters for plotSection

Syntax

```
plotParams = generatePlotParameters(EMPCobj)
```

Description

`plotParams = generatePlotParameters(EMPCobj)` creates a structure of parameters for a 2-D sectional plot of the explicit MPC control law of the explicit MPC controller, `EMPCobj`. You set the fields of this structure and use it to generate the plot using the `plotSection` command.

Examples

Specify Fixed Parameters for 2-D Plot of Explicit Control Law

Define a double integrator plant model and create a traditional implicit MPC controller for this plant. Constrain the manipulated variable to have an absolute value less than 1.

```
plant = tf(1,[1 0 0]);
MPCobj = mpc(plant,0.1,10,3);
MPCobj.MV = struct( Min ,-1, Max ,1);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define the parameter bounds for generating an explicit MPC controller.

```
range = generateExplicitRange(MPCobj);
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min(:) = -2;
range.Reference.Max(:) = 2;
range.ManipulatedVariable.Min(:) = -1.1;
range.ManipulatedVariable.Max(:) = 1.1;
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.  
-->Converting model to discrete time.  
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Create an explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range);
```

```
Regions found / unexplored: 19/ 0
```

Create a default plot parameter structure, which specifies that all of the controller parameters are fixed at their nominal values for plotting.

```
plotParams = generatePlotParameters(EMPCobj);
```

Allow the controller states to vary when creating a plot.

```
plotParams.State.Index = [];  
plotParams.State.Value = [];
```

Fix the manipulated variable and reference signal to 0 for plotting.

```
plotParams.ManipulatedVariable.Index(1) = 1;  
plotParams.ManipulatedVariable.Value(1) = 0;  
plotParams.Reference.Index(1) = 1;  
plotParams.Reference.Value(1) = 0;
```

Generate the 2-D section plot for the explicit MPC controller.

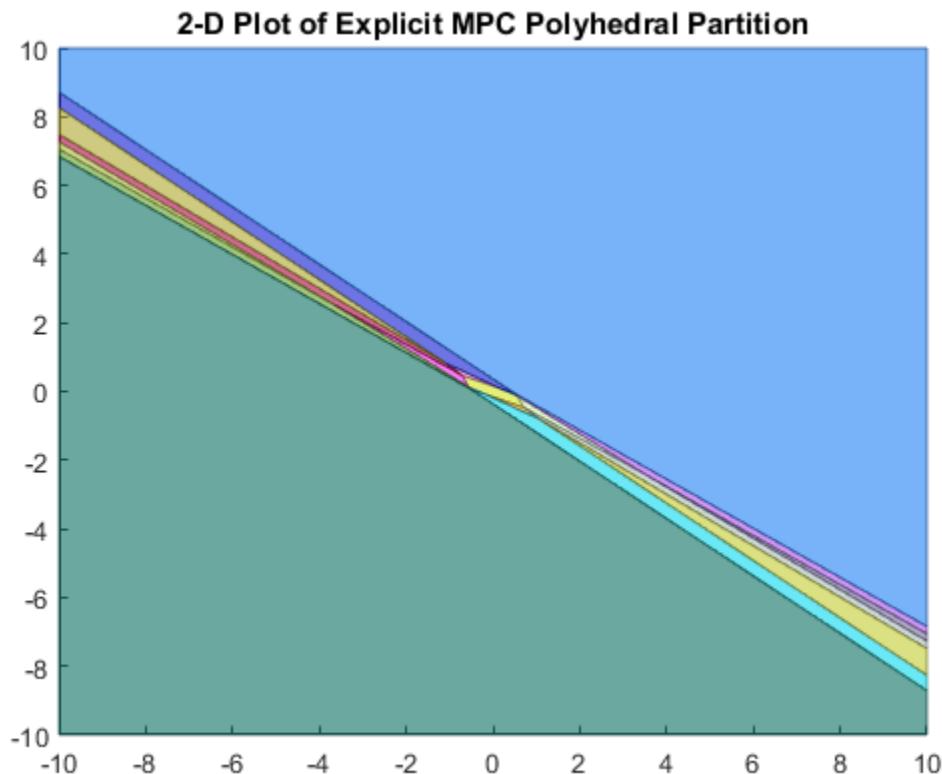
```
plotSection(EMPCobj,plotParams)
```

```
ans =
```

Figure (1: PiecewiseAffineSectionPlot) with properties:

```
Number: 1  
Name: PiecewiseAffineSectionPlot  
Color: [0.9400 0.9400 0.9400]  
Position: [360 502 560 420]  
Units: pixels
```

Use GET to show all properties



Input Arguments

EMPCobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller for which you want to create a 2-D sectional plot, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

Output Arguments

plotParams — Parameters for sectional plot

structure

Parameters for sectional plot of explicit MPC control law, returned as a structure.

As returned by `generatePlotParameters`, the `plotParams` structure command fixes all the control law's parameters at their nominal values. To obtain the desired plot, eliminate the `Index` and `Value` entries of the two parameters forming the plot axes, and modify fixed values as necessary. Then, use the `plotSection` command to display the 2-D sectional plot of the explicit control law's PWA regions with the remaining free parameters as the *x* and *y* axes.

The fields of the plot-parameters structure are as follows.

State — Fixed controller states

structure

Fixed controller states, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.State.Index` is a vector that contains the indices of the controller states to fix for the plot, and `plotParams.State.Value` contains the corresponding constant state values.

Modify the default value of `plotParams.State` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

Reference — Fixed reference signal values

structure

Fixed reference signal values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.Reference.Index` is a vector that contains the indices of the reference signals to fix for the plot, and `plotParams.Reference.Value` contains the corresponding constant reference signal values.

Modify the default value of `plotParams.Reference` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

MeasuredDisturbance — Fixed measured disturbance values

structure

Fixed measured disturbance values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.MeasuredDisturbance.Index` is a vector that contains the indices of the measured disturbances to fix for the plot, and `plotParams.MeasuredDisturbance.Value` contains the corresponding constant measured disturbance values.

Modify the default value of `plotParams.MeasuredDisturbance` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

ManipulatedVariable — Fixed manipulated variable values
structure

Fixed manipulated variable values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.ManipulatedVariable.Index` is a vector that contains the indices of the manipulated variables to fix for the plot, and `plotParams.ManipulatedVariable.Value` contains the corresponding constant manipulated variable values.

Modify the default value of `plotParams.ManipulatedVariable` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

See Also

`generateExplicitMPC` | `plotSection`

Introduced in R2014b

get

MPC property values

Syntax

```
Value = get(MPCobj,PropertyName )  
Struct = get(MPCobj)  
get(MPCobj)
```

Description

`Value = get(MPCobj,PropertyName)` returns the current value of the property `PropertyName` of the MPC controller `MPCobj`. The string `PropertyName` can be the full property name (for example, `UserData`) or any unambiguous case-insensitive abbreviation (for example, `user`). You can specify any generic MPC property.

`Struct = get(MPCobj)` converts the MPC controller `MPCobj` into a standard MATLAB® structure with the property names as field names and the property values as field values.

`get(MPCobj)` without a left-side argument displays all properties of `MPCobj` and their values.

More About

Tips

An alternative to the syntax

```
Value = get(MPCobj,PropertyName )
```

is the structure-like referencing

```
Value = MPCobj.PropertyName
```

For example,

`MPCobj.Ts`
`MPCobj.p`

return the values of the sampling time and prediction horizon of the MPC controller `MPCobj`.

See Also

`mpc` | `set`

Introduced before R2006a

getCodeGenerationData

Create data structures for `mpcmoveCodeGeneration`

Syntax

```
[configData,stateData,onlineData] = getCodeGenerationData(MPCobj)  
[___] = getCodeGenerationData(____,Name,Value)
```

Description

`[configData,stateData,onlineData] = getCodeGenerationData(MPCobj)`
creates data structures for use with `mpcmoveCodeGeneration`.

`[___] = getCodeGenerationData(____,Name,Value)` specifies additional options
using one or more `Name,Value` pair arguments.

Examples

Create MPC Code Generation Data Structures

Create a plant model, and define the MPC signal types.

```
plant = rss(3,2,2);  
plant.D = 0;  
plant = setmpcsignals(plant, mv ,1, ud ,2, mo ,1, uo ,2);
```

Create an MPC controller.

```
mpcObj = mpc(plant,0.1);  
  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1  
for output(s) y1 and zero weight for output(s) y2
```

Configure your controller parameters. For example, define bounds for the manipulated variable.

```
mpcObj.ManipulatedVariables.Min = -1;  
mpcObj.ManipulatedVariables.Max = 1;
```

Create code generation data structures.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj);  
  
-->Converting model to discrete time.  
-->The "Model.Disturbance" property of "mpc" object is empty:  
    Assuming unmeasured input disturbance #2 is integrated white noise.  
    Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each  
-->Converting model to discrete time.  
-->The "Model.Disturbance" property of "mpc" object is empty:  
    Assuming unmeasured input disturbance #2 is integrated white noise.  
    Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Specify Options for Creating MPC Code Generation Structures

Create a plant model and define the MPC signal types.

```
plant = rss(3,2,2);  
plant.D = 0;
```

Create an MPC controller.

```
mpcObj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 1.  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default values.  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default values.  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default values.
```

Create code generation data structures. Configure options to:

- Use single-precision floating-point values in the generated code
- Improve computational efficiency by not computing optimal sequence data.
- Use run your MPC controller in adaptive mode.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj,...  
    DataType , single , OnlyComputeCost ,true, IsAdaptive ,true);
```

```
-->Converting model to discrete time.
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Input Arguments

MPCobj — Model predictive controller

implicit MPC controller object | explicit MPC controller object

Model predictive controller, specified as one of the following:

- Implicit MPC controller object — To create an implicit MPC controller, use `mpc`.
- Explicit MPC controller object — To create an explicit MPC controller, design an implicit controller and then use `generateExplicitMPC`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`,`Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`,`Value1`,...,`NameN`,`ValueN`.

Example: `DataType` , `single` specifies that the generated code uses single-precision floating point values.

InitialState — Initial controller state

`mpcstate` object

Initial controller state, specified as the comma-separated pair consisting of `InitialState` and an `mpcstate` object. This state is used in place of the default state information from `MPCobj`.

DataType — Data type used in generated code

`double` (default) | `single`

Data type used in generated code, specified as specified as the comma-separated pair consisting of `DataType` and one of the following strings:

- `double` — Use double-precision floating point values.
- `single` — Use single-precision floating point values.

OnlyComputeCost — Toggle for computing only optimal cost
false (default) | true

Toggle for computing only optimal cost during simulation, specified as specified as the comma-separated pair consisting of `OnlyComputeCost` and either `true` or `false`. To reduce computational load by not calculating optimal sequence data, set `OnlyComputeCost` to `true`.

IsAdaptive — Adaptive MPC indicator
false (default) | true

Adaptive MPC indicator, specified as specified as the comma-separated pair consisting of `IsAdaptive` and either `true` or `false`. Set `IsAdaptive` to `true` if your controller is running in adaptive mode.

For more information on adaptive MPC, see “Adaptive MPC”.

Note: `IsAdaptive` and `IsLTV` cannot be `true` at the same time.

IsLTV — Time-varying MPC indicator
false (default) | true

Time-varying MPC indicator, specified as either `true` or `false`. Set `IsLTV` to `true` if your controller is running in time-varying mode.

For more information on time-varying MPC, see “Time-Varying MPC”.

Note: `IsAdaptive` and `IsLTV` cannot be `true` at the same time.

Output Arguments

configData — MPC configuration parameters
structure

MPC configuration parameters that are constant at run time, returned as a structure. These parameters are derived from the controller settings in `MPCobj`. When simulating your controller, pass `configData` to `mpcmoveCodeGeneration` without changing any parameters.

stateData — Initial controller states

structure

Initial controller states, returned as a structure. To initialize your simulation with the initial states defined in `MPCobj`, pass `stateData` to `mpcmoveCodeGeneration`. To use different initial conditions, modify `stateData`. You can specify non-default controller states using `InitialState`.

`stateData` has the following fields:

Plant — Plant model state estimates

`MPCobj` nominal plant states (default) | column vector of length n_{xp}

Plant model state estimates, returned as a column vector of length n_{xp} , where n_{xp} is the number of plant model states.

Disturbance — Unmeasured disturbance model state estimates

[] (default) | column vector of length n_{xd}

Unmeasured disturbance model state estimates, returned as a column vector of length n_{xd} , where n_{xd} is the number of unmeasured disturbance model states. `Disturbance` contains the input disturbance model states followed by the output disturbance model states.

To view the input and output disturbance models, use `getindist` and `getoutdist` respectively.

Noise — Output measurement noise model state estimates

[] (default) | column vector of length n_{xn}

Output measurement noise model state estimates, returned as a column vector of length n_{xn} , where n_{xn} is the number of noise model states.

LastMove — Manipulated variable control moves from previous control interval

`MPCobj` nominal MV values (default) | column vector of length n_{mv}

Manipulated variable control moves from previous control interval, returned as a column vector of length n_{mv} , where n_{mv} is the number of manipulated variables.

Covariance – Covariance matrix for controller state estimates

symmetrical n -by- n array

Covariance matrix for controller state estimates, returned as a symmetrical n -by- n array, where n is number of extended controller states; that is, the sum of n_{xp} , n_{xd} , and n_{xn} .

If the controller uses custom state estimation, Covariance is empty.

iA – Active inequality constraints

false (default) | logical vector of length m

Active inequality constraints, where the equal portion of the inequality is true, returned as a logical vector of length m . If `iA(i)` is true, then the i th inequality is active for the latest QP solver solution.

Note: Do not change the value of `iA`. Always use the values returned by either `getCodeGenerationData` or `mpcmoveCodeGeneration`.

onlineData – Online controller data

structure

Online controller data that you must update at each control interval, returned as a structure with the following fields:

Field	Description												
<code>signals</code>	Input and output signals, returned as a structure with the following fields. <table border="1"><thead><tr><th>Field</th><th>Description</th></tr></thead><tbody><tr><td><code>ym</code></td><td>Measured outputs</td></tr><tr><td><code>ref</code></td><td>Output references</td></tr><tr><td><code>md</code></td><td>Measured disturbances</td></tr><tr><td><code>mvTarget</code></td><td>Targets for manipulated variables</td></tr><tr><td><code>externalMV</code></td><td>Manipulated variables externally applied to the plant</td></tr></tbody></table>	Field	Description	<code>ym</code>	Measured outputs	<code>ref</code>	Output references	<code>md</code>	Measured disturbances	<code>mvTarget</code>	Targets for manipulated variables	<code>externalMV</code>	Manipulated variables externally applied to the plant
Field	Description												
<code>ym</code>	Measured outputs												
<code>ref</code>	Output references												
<code>md</code>	Measured disturbances												
<code>mvTarget</code>	Targets for manipulated variables												
<code>externalMV</code>	Manipulated variables externally applied to the plant												

Field	Description												
limits	Input and output constraints, returned as a structure with the following fields:												
	<table border="1"> <thead> <tr> <th>Field</th><th>Description</th></tr> </thead> <tbody> <tr> <td>ymin</td><td>Lower bounds on output signals</td></tr> <tr> <td>ymax</td><td>Upper bounds on output signals</td></tr> <tr> <td>umin</td><td>Lower bounds on input signals</td></tr> <tr> <td>umax</td><td>Upper bounds on input signals</td></tr> </tbody> </table>	Field	Description	ymin	Lower bounds on output signals	ymax	Upper bounds on output signals	umin	Lower bounds on input signals	umax	Upper bounds on input signals		
Field	Description												
ymin	Lower bounds on output signals												
ymax	Upper bounds on output signals												
umin	Lower bounds on input signals												
umax	Upper bounds on input signals												
weights	Updated QP optimization weights, returned as a structure with the following fields:												
	<table border="1"> <thead> <tr> <th>Field</th><th>Description</th></tr> </thead> <tbody> <tr> <td>ywt</td><td>Output weights</td></tr> <tr> <td>uwt</td><td>Manipulated variable weights</td></tr> <tr> <td>duwt</td><td>Manipulated variable rate weights</td></tr> <tr> <td>ecr</td><td>Weight on slack variable used for constraint softening</td></tr> </tbody> </table>	Field	Description	ywt	Output weights	uwt	Manipulated variable weights	duwt	Manipulated variable rate weights	ecr	Weight on slack variable used for constraint softening		
Field	Description												
ywt	Output weights												
uwt	Manipulated variable weights												
duwt	Manipulated variable rate weights												
ecr	Weight on slack variable used for constraint softening												
model	Updated plant and nominal values for adaptive MPC and time-varying MPC, returned as a structure with the following fields:												
	<table border="1"> <thead> <tr> <th>Field</th><th>Description</th></tr> </thead> <tbody> <tr> <td>A, B, C, D</td><td>State-space matrices of discrete-time state-space model.</td></tr> <tr> <td>X</td><td>Nominal plant states</td></tr> <tr> <td>U</td><td>Nominal plant inputs</td></tr> <tr> <td>Y</td><td>Nominal plant outputs</td></tr> <tr> <td>DX</td><td>Nominal plant state derivatives</td></tr> </tbody> </table>	Field	Description	A, B, C, D	State-space matrices of discrete-time state-space model.	X	Nominal plant states	U	Nominal plant inputs	Y	Nominal plant outputs	DX	Nominal plant state derivatives
Field	Description												
A, B, C, D	State-space matrices of discrete-time state-space model.												
X	Nominal plant states												
U	Nominal plant inputs												
Y	Nominal plant outputs												
DX	Nominal plant state derivatives												

`getCodeGenData` returns `onlineData` with empty matrices for all structure fields, except `signals.ref`, `signals.ym`, and `signals.md`. These fields contain the corresponding nominal signal values from `MPCobj`. If your controller does not have measured disturbances, `signals.md` is returned as an empty matrix.

For more information on configuring `onlineData` fields, see [mpcmoveCodeGeneration](#)

More About

- “Generate Code To Compute Optimal MPC Moves in MATLAB”
- “Generate Code and Deploy Controller to Real-Time Targets”

See Also

[mpcmoveCodeGeneration](#)

Introduced in R2016a

getconstraint

Set custom constraints on linear combinations of plant inputs and outputs

Syntax

```
[E,F,G,V,S] = getconstraint(MPCobj)
```

Description

`[E,F,G,V,S] = getconstraint(MPCobj)` returns the custom constraints previously defined for the MPC controller, `MPCobj`. The constraints are in the general form:

$$Eu(k+j|k) + Fy(k+j|k) + Sv(k+j|k) \leq G + \varepsilon V$$

where $j = 0, \dots, p$, and:

- p is the prediction horizon.
- k is the current time index.
- u is a column vector manipulated variables.
- y is a column vector of all plant output variables.
- v is a column vector of measured disturbance variables.
- ε is a scalar slack variable used for constraint softening (as in “Standard Cost Function”).
- E, F, G, V , and S are constant matrices.

Since the MPC controller does not optimize $u(k+p|k)$, `getconstraint` calculates the last constraint at time $k+p$ assuming that $u(k+p|k) = u(k+p-1|k)$.

Examples

Retrieve Custom Constraints from MPC Controller

Create a third-order plant model with two manipulated variables, one measured disturbance, and two measured outputs.

```
plant = rss(3,2,3);
```

```
plant.D = 0;
plant = setmpcsignals(plant, mv ,[1 2], md ,3);
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Assume that you have two soft constraints.

$$\begin{aligned} u_1 + u_2 &\leq 5 \\ y_2 + v &\leq 10 \end{aligned}$$

Set the constraints for the MPC controller.

```
E = [1 1; 0 0];
F = [0 0; 0 1];
G = [5;10];
V = [1;1];
S = [0;1];
setconstraint(MPCobj,E,F,G,V,S)
```

Retrieve the constraints from the controller.

```
[E,F,G,V,S] = getconstraint(MPCobj)
```

E =

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

F =

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

$G =$

5
10

$V =$

1
1

$S =$

0
1

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

E — Manipulated variable constraint constant

matrix

Manipulated variable constraint constant, returned as a matrix with:

- n_c rows, where n_c is the number of constraints.
- n_u columns, where n_u is the number of manipulated variables.

If `MPCobj` has no custom constraints, then `E` is empty, [].

F — Controlled output constraint constant

matrix

Controlled output constraint constant, returned as a matrix with:

- n_c rows, where n_c is the number of constraints.
- n_y columns, where n_y is the number of controlled outputs (measured and unmeasured).

If `MPCobj` has no custom constraints, then `F` is empty [].

G — Custom constraint constant

column vector

Custom constraint constant, returned as a column vector with n_c elements, where n_c is the number of constraints.

If `MPCobj` has no custom constraints, then `G` is empty [].

V — Constraint softening constant

column vector

Constraint softening constant representing the equal concern for the relaxation (ECR), returned as a column vector with n_c elements, where n_c is the number of constraints. If `MPCobj` has no custom constraints, then `V` is empty [].

If `V` is not specified, a default value of 1 is applied to all constraint inequalities and all constraints are soft. This behavior is the same as the default behavior for output bounds, as described in “Standard Cost Function”.

To make the i^{th} constraint hard, specify $V(i) = 0$.

To make the i^{th} constraint soft, specify $V(i) > 0$ in keeping with the constraint violation magnitude you can tolerate. The magnitude violation depends on the numerical scale of the variables involved in the constraint.

In general, as $V(i)$ decreases, the controller hardens the constraints by decreasing the constraint violation that is allowed.

S — Measured disturbance constraint constant

matrix

Measured disturbance constraint constant, returned as a matrix with:

- n_c rows, where n_c is the number of constraints.

- n_v columns, where n_v is the number of measured disturbances.

If there are no measured disturbances in the custom constraints, or `MPCobj` has no custom constraints, then `S` is empty [].

More About

- “Constraints on Linear Combinations of Inputs and Outputs”

See Also

`setconstraint`

Introduced in R2011a

getEstimator

Obtain Kalman gains and model for estimator design

Syntax

```
[L,M] = getEstimator(MPCobj)
[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj)
[L,M,model,index] = getEstimator(MPCobj, sys )
```

Description

`[L,M] = getEstimator(MPCobj)` extracts the Kalman gains used by the state estimator in a model predictive controller. The estimator updates the states of internal plant, disturbance, and noise models at the beginning of each controller interval.

`[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj)` also returns the system matrices used to calculate the estimator gains.

`[L,M,model,index] = getEstimator(MPCobj, sys)` returns an LTI state-space representation of the system used for state-estimator design and a structure summarizing the I/O signal types of the system.

Examples

Extract Parameters for State Estimation

The plant is a stable, discrete LTI ss model with four states, three inputs and three outputs. The manipulated variables are inputs 1 and 2. Input 3 is an unmeasured disturbance. Output 1 and 3 are measured. Output 2 is unmeasured.

Create a model of the plant and specify the signals for MPC.

```
rng(1253) % For repeatable results
Plant = drss(4,3,3);
```

```

Plant.Ts = 0.25;
Plant = setmpcsignals(Plant, MV ,[1,2], UD ,3, MO ,[1 3], UO , 2);
Plant.d(:,[1,2]) = 0;

```

The last command forces the plant to satisfy the assumption of no direct feedthrough.

Calculate the default Model Predictive Controller for this plant.

```

MPCobj = mpc(Plant);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 y3 and zero weight for output(s) y2

```

Obtain the parameters to be used in state estimation.

```

[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj);

-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #3 is integrated white noise.
-->Assuming output disturbance added to measured output channel #1 is integrated white
    Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
    output channel.

```

Based on the estimator state equation, the the estimator poles are given by the eigenvalues of $A - L^*C_m$. Calculate and display the poles.

```
Poles = eig(A - L*Cm)
```

```

Poles =

```

-0.7467
-0.5019
0.0769
0.4850
0.8825
0.8291

Confirm that the default estimator is asymptotically stable.

```
max(abs(Poles))
```

```
ans =
```

```
0.8825
```

This value is less than 1, so the estimator is asymptotically stable.

Verify that in this case, $L = A^*M$.

```
L = A*M
```

```
ans =
```

```
1.0e-16 *
```

0	0.5551
-0.1388	-0.2776
0.1388	-0.1388
0	0.2082
-0.2082	-0.2776
-0.2776	0

Input Arguments

MPCobj — MPC controller

MPC controller object

MPC controller, specified as an MPC controller object. Use the `mpc` command to create the MPC controller.

Output Arguments

L — Kalman gain matrix for time update

matrix

Kalman gain matrix for the time update, returned as a matrix. The dimensions of L are n_x -by- n_{ym} , where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-44.

M — Kalman gain matrix for measurement update
matrix

Kalman gain matrix for the measurement update, returned as a matrix. The dimensions of M are n_x -by- n_{ym} , where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-44.

A, Cm, Bu, Bv, Dvm — System matrices
matrices

System matrices used to calculate the estimator gains, returned as matrices of various dimensions. For definitions of these system matrices, see “State Estimator Equations” on page 1-44.

model — System used for state-estimator design
state-space model

System used for state-estimator design, returned as a state-space (`ss`) model. The input to `model` is a vector signal comprising the following components, concatenated in the following order:

- Manipulated variables
- Measured disturbance variables
- 1
- Noise inputs to disturbance models
- Noise inputs to measurement noise model

The number of noise inputs depends on the disturbance and measurement noise models within `MPCObj`. For the category noise inputs to disturbance models, inputs to the input disturbance model (if any) precede those entering the output disturbance model (if any). The constant input, 1, accounts for nonequilibrium nominal values (see “MPC Modeling”).

To make the calculation of gains L and M more robust, additive white noise inputs are assumed to affect the manipulated variables and measured disturbances (see “Controller State Estimation”). These white noise inputs are not included in `model`.

index – Locations of variables within model

structure

Locations of variables within the inputs and outputs of `model`. The structure summarizes these locations with the following fields and values.

Field Name	Value
<code>ManipulatedVariables</code>	Indices of manipulated variables within the input vector of <code>model</code> .
<code>MeasuredDisturbances</code>	Indices of measured input disturbances within the input vector of <code>model</code> .
<code>Offset</code>	Index of the constant input 1 within the input vector of <code>model</code> .
<code>WhiteNoise</code>	Indices of unmeasured disturbance inputs within the input vector of <code>model</code> .
<code>MeasuredOutputs</code>	Indices of measured outputs within the output vector of <code>model</code> .
<code>UnmeasuredOutputs</code>	Indices of unmeasured outputs within the output vector of <code>model</code> .

More About

State Estimator Equations

The following equations describe the state estimation. For more details, see “Controller State Estimation”.

$$\text{Output estimate: } y_m[n | n-1] = C_m x[n | n-1] + D_{vm} v[n].$$

$$\text{Measurement update: } x[n | n] = x[n | n-1] + M (y_m[n] - y_m[n | n-1]).$$

$$\text{Time update: } x[n+1 | n] = A x[n | n-1] + B_u u[n] + B_v v[n] + L (y_m[n] - y_m[n | n-1]).$$

$$\text{Estimator state: } x[n+1 | n] = (A - L C_m) x[n | n-1] + B_u u[n] + (B_v - L D_{vm}) v[n] + L y_m[n].$$

The estimator state is based on the current measurement of $y_m[n]$ and $v[n]$ as well as the optimal control action $u[n]$ computed at the current control interval.

The variables in these equations are summarized in the following table.

Symbol	Description
x	<p>Controller state vector, length n_x. It includes (in this sequence):</p> <ul style="list-style-type: none"> Plant model state estimates. Dimension obtained by conversion of <code>MPCobj.Model.Plant</code> to discrete LTI state-space form (if necessary), followed by use of <code>absorbDelay</code> to convert any delays to additional states. Input disturbance model state estimates (if any). Use the <code>getindist</code> command to review the input disturbance model structure. Output disturbance model state estimates (if any). Use the <code>getoutdist</code> command to review the output disturbance model structure. Output measurement noise states (if any) as specified by <code>MPCobj.Model.Noise</code>. <p>The length n_x is the sum of the number of states in the above four categories.</p>
y_m	Vector of measured outputs or an estimate of their true values, length n_{ym} .
u	Vector of manipulated variables, length n_u .
v	Vector of measured input disturbances, length n_v .
$[j k]$	Denotes an estimate of a state or output at time t_j based on data available at time t_k .
$[k]$	Denotes a quantity known at time t_k , i.e., not an estimate.
A	n_x -by- n_x state transition matrix.
B_u	n_x -by- n_u matrix mapping u to x .
B_v	n_x -by- n_x matrix mapping v to x .
C_m	n_{ym} -by- n_x matrix mapping x to y_m .
D_{vm}	n_{ym} -by- n_v matrix mapping v to y_m . Note that $D_{um} = 0$ because there can be no direct feedthrough between any manipulated variable and any measured output.

Symbol	Description
L	n_x -by- n_{ym} Kalman gain matrix for the time update. (See <code>kalmd</code> in the Control System Toolbox™ documentation.) Note that $L = A^*M$ minimizes the expected state estimation error for most combinations of plant and disturbance models used in MPC, but this is not true in general.
M	n_x -by- n_{ym} Kalman gain matrix for the measurement update. (See <code>kalmd</code> in the Control System Toolbox documentation.)

- “Controller State Estimation”
- “MPC Modeling”

See Also

`getindist` | `getoutdist` | `mpc` | `mpcstate` | `setEstimator`

Introduced in R2014b

getindist

Retrieve unmeasured input disturbance model

Syntax

```
indist = getindist(MPCobj)
[indist,channels] = getindist(MPCobj)
```

Description

`indist = getindist(MPCobj)` returns the input disturbance model, `indist`, used by the model predictive controller, `MPCobj`.

`[indist,channels] = getindist(MPCobj)` also returns the input channels to which integrated white noise has been added by default. For more information on the default model, see “MPC Modeling”.

Examples

Retrieve Input Disturbance Model

Define a plant model with no direct feedthrough.

```
plant = rss(3,1,2);
plant.d = 0;
```

Set the first input signal as a manipulated variable and the second input as an unmeasured disturbance.

```
plant = setmpcsignals(plant, MV ,[1], UD ,[2]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default value of 0.
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default value of 0.
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default value of 1.
```

Extract the input disturbance model.

```
indist = getindist(MPCobj);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #2 is integrated white noise.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Retrieve Input Disturbance Model Channels with Default Integrated White Noise

Define a plant model with no direct feedthrough.

```
plant = rss(3,1,3);
plant.d = 0;
```

Set the first input signal as a manipulated variable and the other two inputs as unmeasured disturbances.

```
plant = setmpcsignals(plant, MV ,[1], UD ,[2 3]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 1.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming ControlHorizon = 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default value of 0.
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default value of 0.
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default value of 1.
```

Extract the default output disturbance model.

```
[indist,channels] = getindist(MPCobj);

-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
    Assuming unmeasured input disturbance #2 is integrated white noise.
    Assuming unmeasured input disturbance #3 is white noise.
    Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Check which input disturbance channels have integrated white noise added by default.

```
channels
```

```
channels =
```

```
1
```

An integrator has been added only to the first unmeasured input disturbance. The other input disturbance uses a static unity gain to preserve state observability.

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

indist — Input disturbance model

discrete-time, delay-free, state-space model

Input disturbance model used by the model predictive controller, `MPCobj`, returned as a discrete-time, delay-free, state-space model.

The input disturbance model has:

- Unit-variance white noise input signals. By default, the number of inputs depends upon the number of unmeasured input disturbances and the need to maintain controller state observability. For custom input disturbance models, the number of inputs is your choice.
- n_d outputs, where n_d is the number of unmeasured disturbance inputs defined in `MPCobj.Model.Plant`. Each disturbance model output is sent to the corresponding plant unmeasured disturbance input.

If `MPCobj` does not have any unmeasured disturbance, `indist` is returned as an empty state-space model.

This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

channels — Input channels with integrated white noise

vector of input indices

Input channels with integrated white noise added by default, returned as a vector of input indices. If you set `indist` to a custom input disturbance model using `setindist`, `channels` is empty.

More About

Tips

- To specify a custom input disturbance model, use the `setindist` command.
- “MPC Modeling”
- “Controller State Estimation”

See Also

`getEstimator` | `getoutdist` | `mpc` | `setEstimator` | `setindist`

Introduced in R2006a

getname

Retrieve I/O signal names in MPC prediction model

Syntax

```
name = getname(MPCobj, input ,I)
name = getname(MPCobj, output ,I)
```

Description

`name = getname(MPCobj, input ,I)` returns the name of the I th input signal in variable `name`. This is equivalent to `name = MPCobj.Model.Plant.InputName{I}`. The name property is equal to the contents of the corresponding `Name` field of `MPCobj.DisturbanceVariables` or `MPCobj.ManipulatedVariables`.

`name = getname(MPCobj, output ,I)` returns the name of the I th output signal in variable `name`. This is equivalent to `name=MPCobj.Model.Plant.OutputName{I}`. The name property is equal to the contents of the corresponding `Name` field of `MPCobj.OutputVariables`.

See Also

`mpc` | `set` | `setname`

Introduced before R2006a

getoutdist

Retrieve unmeasured output disturbance model

Syntax

```
outdist = getoutdist(MPCobj)
[outdist,channels] = getoutdist(MPCobj)
```

Description

`outdist = getoutdist(MPCobj)` returns the output disturbance model, `outdist`, used by the model predictive controller, `MPCobj`.

`[outdist,channels] = getoutdist(MPCobj)` also returns the output channels to which integrated white noise has been added by default. For more information on the default model, see “MPC Modeling”.

Examples

Retrieve Output Disturbance Model

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,2,2);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Extract the output disturbance model.

```
outdist = getoutdist(MPCobj);

-->Converting model to discrete time.
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each channel.
```

Retrieve Output Disturbance Model Channels with Default Integrated White Noise

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 1.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming ControlHorizon = 1.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default manipulated variable weights.
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default manipulated variable rate weights.
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default output variable weights.
```

Extract the default output disturbance model.

```
[outdist,channels] = getoutdist(MPCobj);

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each channel.
```

Check which channels have default integrated white noise disturbances.

channels

```
channels =
```

```
1    2    3
```

Integrators have been added to all three output channels.

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

outdist — Output disturbance model

discrete-time, delay-free, state-space model

Output disturbance model used by the model predictive controller, `MPCObj`, returned as a discrete-time, delay-free, state-space model.

The output disturbance model has:

- n_y outputs, where n_y is the number of plant outputs defined in `MPCObj.Model.Plant`. Each disturbance model output is added to the corresponding plant output. By default, disturbance models corresponding to unmeasured output channels are zero.
- Unit-variance white noise input signals. By default, the number of inputs is equal to the number of default integrators added.

This model, in combination with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

channels — Output channels with integrated white noise

vector of output indices

Output channels with integrated white noise added by default, returned as a vector of output indices. If you set `outdist` to a custom output disturbance model using `setoutdist`, `channels` is empty.

More About

Tips

- To specify a custom output disturbance model, use the `setoutdist` command.
- “MPC Modeling”

- “Controller State Estimation”

See Also

[getEstimator](#) | [getindist](#) | [mpc](#) | [setEstimator](#) | [setoutdist](#)

Introduced before R2006a

gpc2mpc

Generate MPC controller using generalized predictive controller (GPC) settings

Syntax

```
mpc = gpc2mpc(plant)
gpcOptions = gpc2mpc
mpc = gpc2mpc(plant,gpcOptions)
```

Description

`mpc = gpc2mpc(plant)` generates a single-input single-output MPC controller with default GPC settings and sampling time of the plant, `plant`. The GPC is a nonminimal state-space representation described in [1]. `plant` is a discrete-time LTI model with sampling time greater than 0.

`gpcOptions = gpc2mpc` creates a structure `gpcOptions` containing default values of GPC settings.

`mpc = gpc2mpc(plant,gpcOptions)` generates an MPC controller using the GPC settings in `gpcOptions`.

Input Arguments

plant

Discrete-time LTI model with sampling time greater than 0.

Default:

gpcOptions

GPC settings, specified as a structure with the following fields.

N1	Starting interval in prediction horizon, specified as a positive integer. Default: 1.
----	---

N2	Last interval in prediction horizon, specified as a positive integer greater than N1. Default: 10.
NU	Control horizon, specified as a positive integer less than the prediction horizon. Default: 1.
Lam	Penalty weight on changes in manipulated variable, specified as a positive integer greater than or equal to 0. Default: 0.
T	Numerator of the GPC disturbance model, specified as a row vector of polynomial coefficients whose roots lie within the unit circle. Default: [1].
MVindex	Index of the manipulated variable for multi-input plants, specified as a positive integer. Default: 1.

Default:

Examples

Design an MPC controller using GPC settings:

```
% Specify the plant described in Example 1.8 of
% [1].
G = tf(9.8*[1 -0.5 6.3],conv([1 0.6565],[1 -0.2366 0.1493]));

% Discretize the plant with sample time of 0.6 seconds.
Ts = 0.6;
Gd = c2d(G, Ts);

% Create a GPC settings structure.
GPCOptions = gpc2mpc;

% Specify the GPC settings described in example 4.11 of
% [1].
% Hu
GPCOptions.NU = 2;
% Hp
```

```
GPCoptions.N2 = 5;
% R
GPCoptions.Lam = 0;
GPCoptions.T = [1 -0.8];

% Convert GPC to an MPC controller.
mpc = gpc2mpc(Gd, GPCoptions);

% Simulate for 50 steps with unmeasured disturbance between
% steps 26 and 28, and reference signal of 0.
SimOptions = mpccsimopt(mpc);
SimOptions.UnmeasuredDisturbance = [zeros(25,1); ...
-0.1*ones(3,1); 0];
sim(mpc, 50, 0, SimOptions);
```

More About

Tips

- For plants with multiple inputs, only one input is the manipulated variable, and the remaining inputs are measured disturbances in feedforward compensation. The plant output is the measured output of the MPC controller.
- Use the MPC controller with Model Predictive Control Toolbox™ software for simulation and analysis of the closed-loop performance.
- “Design Controller Using MPC Designer”
- “Design MPC Controller at the Command Line”

References

- [1] Maciejowski, J. M. *Predictive Control with Constraints*, Pearson Education Ltd., 2002, pp. 133–142.

See Also

“MPC Controller Object” on page 3-2

Introduced in R2010a

mpc

Create MPC controller

Syntax

```
MPCObj = mpc(Plant)
MPCObj = mpc(Plant,Ts)
MPCObj = mpc(Plant,Ts,p,m,W,MV,OV,DV)
MPCObj = mpc(Models,Ts,p,m,W,MV,OV,DV)
```

Description

`MPCObj = mpc(Plant)` creates a Model Predictive Controller object based on a discrete-time prediction model. The prediction model `Plant` can be either an LTI model with a specified sample time, or a System Identification model (see “Identify Plant from Data”). The controller, `MPCObj`, inherits its control interval from `Plant.Ts`, and its time unit from `Plant.TimeUnit`. All other controller properties are default values. After you create the MPC controller, you can set its properties using `MPCObj.PropertyName = PropertyValue`.

`MPCObj = mpc(Plant,Ts)` specifies a control interval of `Ts`. If `Plant` is a discrete-time LTI model with an unspecified sample time (`Plant.Ts = -1`), it inherits sample time `Ts` when used for predictions.

`MPCObj = mpc(Plant,Ts,p,m,W,MV,OV,DV)` specifies additional controller properties such as the prediction horizon (`p`), control horizon (`m`), and input, input increment, and output weights (`W`). You can also set the properties of manipulated variables (`MV`), output variables (`OV`), and input disturbance variables (`DV`). If any of these values are omitted or empty, the default values apply.

`MPCObj = mpc(Models,Ts,p,m,W,MV,OV,DV)` creates a Model Predictive Controller object based on a prediction model set, `Models`. This set includes plant, input disturbance, and measurement noise models along with the nominal conditions at which the models were obtained.

Input Arguments

Plant

Plant model to be used in predictions, specified as an LTI model (`tf`, `ss`, or `zpk`) or a System Identification Toolbox™ model. If the `Ts` input argument is unspecified, `Plant` must be a discrete-time LTI object with a specified sample time, or a System Identification Toolbox model.

Unless you specify otherwise, controller design assumes that all plant inputs are manipulated variables and all plant outputs are measured. Use the `setmpcsignals` command or the LTI `InputGroup` and `OutputGroup` properties to designate other signal types.

Ts

Controller sample time (control interval), specified as a positive scalar value.

p

Prediction horizon, specified as a positive integer. The control interval, `Ts`, determines the duration of each step. The default value is $10 + \text{maximum intervals of delay (if any)}$.

m

Control horizon, specified as a scalar integer, $1 \leq m \leq p$, or as a vector of blocking factors such that `sum(m) ≤ p` (see “Optimization Variables”). The default value is 2.

W

Controller tuning weights, specified as a structure. For details about how to specify this structure, see “Weights” on page 1-65.

MV

Bounds and other properties of manipulated variables, specified as a 1-by-`nu` structure array, where `nu` is the number of manipulated variables defined in the plant model. For details about how to specify this structure, see “ManipulatedVariables” on page 1-62.

OV

Bounds and other properties of the output variables, specified as a 1-by-`ny` structure array, where `ny` is the number of output variables defined in the plant model. For details about how to specify this structure, see “OutputVariables” on page 1-63.

DV

Scale factors and other properties of the disturbance inputs, specified as a 1-by-`nd` structure array, where `nd` is the number of disturbance inputs (measured + unmeasured) defined in the plant model. For details about how to specify this structure, see “DisturbanceVariables” on page 1-64.

Models

Plant, input disturbance, and measurement noise models, along with the nominal conditions at which these models were obtained, specified as a structure. For details about how to specify this structure, see “Model” on page 1-67.

Construction and Initialization

To minimize computational overhead, Model Predictive Controller creation occurs in two phases. The first happens at *construction* when you invoke the `mpc` command, or when you change a controller property. Construction involves simple validity and consistency checks, such as signal dimensions and non-negativity of weights.

The second phase is *initialization*, which occurs when you use the object for the first time in a simulation or analytical procedure. Initialization computes all constant properties required for efficient numerical performance, such as matrices defining the optimal control problem and state estimator gains. Additional, diagnostic checks occur during initialization, such as verification that the controller states are observable.

By default, both phases display informative messages in the command window. You can turn these messages on or off using the `mpcverbosity` command.

Properties

All of the parameters defining the traditional (implicit) MPC control law are stored in an MPC object, whose properties are listed in the following table.

MPC Controller Object

Property	Description
<code>ManipulatedVariables</code> (or <code>MV</code> or <code>Manipulated</code> or <code>Input</code>)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.

Property	Description
OutputVariables (or OV or Controlled or Output)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
DisturbanceVariables (or DV or Disturbance)	Disturbance scale factors, names, and units
Weights	Weights used in computing the performance (cost) function
Model	Plant, input disturbance, and output noise models, and nominal conditions.
Ts	Controller sample time
Optimizer	Parameters controlling the QP solver
PredictionHorizon	Prediction horizon
ControlHorizon	Number of free control moves or vector of blocking moves
History	Creation time
Notes	Text or comments about the MPC controller object
UserData	Any additional data

ManipulatedVariables

ManipulatedVariables (or MV or Manipulated or Input) is an n_u -dimensional array of structures (n_u = number of manipulated variables), one per manipulated variable. Each structure has the fields described in the following table, where p denotes the prediction horizon. Unless indicated otherwise, numerical values are in engineering units.

Manipulated Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this MV	1
Min	1 to p length vector of lower bounds on this MV	-Inf

Field Name	Content	Default
Max	1 to p length vector of upper bounds on this MV	Inf
MinECR	1 to p length vector of nonnegative parameters specifying the Min bound softness (0 = hard).	0 (dimensionless)
MaxECR	1 to p length vector of nonnegative parameters specifying the Max bound softness (0 = hard).	0 (dimensionless)
Target	1 to p length vector of target values for this MV	nominal
RateMin	1 to p length vector of lower bounds on the interval-to-interval change for this MV	-Inf
RateMax	1 to p length vector of upper bounds on the interval-to-interval change for this MV	Inf
RateMinECR	1 to p length vector of nonnegative parameters specifying the RateMin bound softness (0 = hard).	0 (dimensionless)
RateMaxECR	1 to p length vector of nonnegative parameters specifying the RateMax bound softness (0 = hard).	0 (dimensionless)
Name	Read-only MV signal name (character string)	InputName of LTI plant model
Units	Read-only MV signal units (character string)	InputUnit of LTI plant model

Note Rates refer to the difference $\Delta u(k) = u(k) - u(k-1)$. Constraints and weights based on derivatives du/dt of continuous-time input signals must be properly reformulated for the discrete-time difference $\Delta u(k)$, using the approximation $du/dt \cong \Delta u(k)/T_s$.

OutputVariables

`OutputVariables` (or `OV` or `Controlled` or `Output`) is an n_y -dimensional array of structures (n_y = number of outputs), one per output signal. Each structure has the

fields described in the following table. p denotes the prediction horizon. Unless specified otherwise, values are in engineering units.

Output Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this OV	1
Min	1 to p length vector of lower bounds on this OV	-Inf
Max	1 to p length vector of upper bounds on this OV	Inf
MinECR	1 to p length vector of nonnegative parameters specifying the Min bound softness (0 = hard).	1 (dimensionless)
MaxECR	1 to p length vector of nonnegative parameters specifying the Max bound softness (0 = hard).	1 (dimensionless)
Name	Read-only OV signal name (character string)	OutputName of LTI plant model
Units	Read-only OV signal units (character string)	OutputUnit of LTI plant model

In order to reject constant disturbances due, for instance, to gain nonlinearities, the default measured output disturbance model used in Model Predictive Control Toolbox software is integrated white noise (see “Output Disturbance Model”).

DisturbanceVariables

DisturbanceVariables (or DV or Disturbance) is an (n_v+n_d) -dimensional array of structures (n_v = number of measured input disturbances, n_d = number of unmeasured input disturbances). Each structure has the fields described in the following table.

Disturbance Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this DV	1
Name	Read-only DV signal name (character string)	InputName of LTI plant model

Field Name	Content	Default
Units	Read-only DV signal units (character string)	InputUnit of LTI plant model

The order of the disturbance signals within the array DV is the following: the first n_v entries relate to measured input disturbances, the last n_d entries relate to unmeasured input disturbances.

Weights

Weights is the structure defining the QP weighting matrices. It contains four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see “Standard Cost Function”) or the alternative cost function (see “Alternative Cost Function”).

Standard Cost Function

The following table lists the content of the four structure fields. In the table, p denotes the prediction horizon, n_u the number of manipulated variables, and n_y the number of output variables.

For the MV, MVRate and OV weights, if you specify fewer than p rows, the last row repeats automatically to form a matrix containing p rows.

Weights for the Standard Cost Function

Field Name (Abbreviations)	Content	Default (dimensionless)
ManipulatedVariables (or MV or Manipulated or Input)	(1 to p)-by- n_u dimensional array of nonnegative MV weights	<code>zeros(1,nu)</code>
ManipulatedVariablesRate (or MVRate or ManipulatedRate or InputRate)	(1 to p)-by- n_u dimensional array of MV-increment weights	<code>0.1*ones(1,nu)</code>
OutputVariables (or OV or Controlled or Output)	(1 to p)-by- n_y dimensional array of OV weights	1 (The default for output weights is the following: if $n_u \geq n_y$, all outputs are weighted with unit weight; if $n_u < n_y$, n_u outputs default to 1, with preference given

Field Name (Abbreviations)	Content	Default (dimensionless)
		to measured outputs, and the rest default to 0.)
ECR	Scalar weight on the slack variable ε used for constraint softening	1e5* (max weight)

Note If all **MVRate** weights are strictly positive, the resulting QP problem is strictly convex. If some **MVRate** weights are zero, the QP Hessian could be positive semidefinite. In order to keep the QP problem strictly convex, when the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12} , the quantity $10 * \text{sqrt}(\text{eps})$ is added to each diagonal term. See “Cost Function”.

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To do so, define the fields **ManipulatedVariables**, **ManipulatedVariablesRate**, and **OutputVariables** as cell arrays, each containing a single positive-semi-definite matrix of the appropriate size. Specifically, **OutputVariables** must be a cell array containing the n_y -by- n_y Q matrix, **ManipulatedVariables** must be a cell array containing the n_u -by- n_u R_u matrix, and **ManipulatedVariablesRate** must be a cell array containing the n_u -by- n_u $R_{\Delta u}$ matrix (see “Alternative Cost Function” and the **mpcweightsdemo** example). You can use diagonal weight matrices for one or more of these fields. If you omit a field, the MPC controller uses the defaults shown in the table above.

For example, you can specify off-diagonal weights, as follows

```
MPCObj.Weights.OutputVariables = {Q};
MPCObj.Weights.ManipulatedVariables = {Ru};
MPCObj.Weights.ManipulatedVariablesRate = {Rdu};
```

where $Q = Q$, $R_u = R_u$, and $R_{\Delta u} = R_{\Delta u}$ are positive semidefinite matrices.

Note You cannot specify non-diagonal weights that vary at each prediction horizon step. The same Q , R_u , and $R_{\Delta u}$ weights apply at each step.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in “Controller State Estimation”. It is a 1-D structure containing the following fields.

Models Used by MPC

Field Name	Content	Default									
Plant	LTI model or identified linear model of the plant	No default									
Disturbance	LTI model describing expected unmeasured input disturbances	[] (By default, input disturbances are expected to be integrated white noise. To model the signal, an integrator with dimensionless unity gain is added for each unmeasured input disturbance, unless the addition causes the controller to lose state observability. In that case, the disturbance is expected to be white noise, and so, a dimensionless unity gain is added to that channel instead.)									
Noise	LTI model describing expected noise for output measurements	[] (By default, measurement noise is expected to be white noise with unit variance. To model the signal, a dimensionless unity gain is added for each measured channel.)									
Nominal	Structure containing the state, input, and output values where <code>Model.Plant</code> is linearized	<p>The default values of the fields are shown in the following table:</p> <table border="1"> <thead> <tr> <th>Field</th><th>Description</th><th>Default</th></tr> </thead> <tbody> <tr> <td>X</td><td>Plant state at operating point</td><td>[]</td></tr> <tr> <td>U</td><td>Plant input at operating point, including manipulated variables and measured</td><td>[]</td></tr> </tbody> </table>	Field	Description	Default	X	Plant state at operating point	[]	U	Plant input at operating point, including manipulated variables and measured	[]
Field	Description	Default									
X	Plant state at operating point	[]									
U	Plant input at operating point, including manipulated variables and measured	[]									

Field Name	Content	Default		
		Field	Description	Default
			and unmeasured disturbances	
		Y	Plant output at operating point	[]
		DX	For continuous-time models, DX is the state derivative at operating point: $DX=f(X,U)$. For discrete-time models, $DX=x(k+1)-x(k)=f(X,U)-X$.	[]

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See “MPC Modeling”.

Specify input and output signal types via the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, use the `setmpcsignals` command. Valid signal types are listed in the following tables.

Input Groups in Plant Model

Name (Abbreviations)	Value
<code>ManipulatedVariables</code> (or MV or Manipulated or Input)	Indices of manipulated variables in <code>Model.Plant</code>
<code>MeasuredDisturbances</code> (or MD or Measured)	Indices of measured disturbances in <code>Model.Plant</code>
<code>UnmeasuredDisturbances</code> (or UD or Unmeasured)	Indices of unmeasured disturbances in <code>Model.Plant</code>

Output Groups in Plant Model

Name (Abbreviations)	Value
<code>MeasuredOutputs</code> (or MO or Measured)	Indices of measured outputs in <code>Model.Plant</code>

Name (Abbreviations)	Value
UnmeasuredOutputs (or U0 or Unmeasured)	Indices of unmeasured outputs in Model.Plant

By default, all `Model.Plant` inputs are manipulated variables, and all outputs are measured.

The structure `Nominal` contains the values (in engineering units) for states, inputs, outputs, and state derivatives/differences at the operating point where `Model.Plant` applies. This point is typically a linearization point. The fields are reported in the following table (see also “MPC Modeling”).

Nominal Values at Operating Point

Field	Description	Default
X	Plant state at operating point	[]
U	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	[]
Y	Plant output at operating point	[]
DX	For continuous-time models, DX is the state derivative at operating point: $DX=f(X,U)$. For discrete-time models, $DX=x(k+1)-x(k)=f(X,U)-X$.	[]

Ts

Sample time of the MPC controller. By default, if `Model.Plant` is a discrete-time model, `Ts = Model.Plant.ts`. For continuous-time plant models, specify a controller `Ts`. The `Ts` measurement unit is inherited from `Model.Plant.TimeUnit`.

Optimizer

Parameters for the QP optimization. `Optimizer` is a structure with the following fields:

Optimizer Properties

Field	Description	Default
MaxIter	Maximum number of iterations allowed in the QP solver, specified as one of the following:	Default

Field	Description	Default
	<ul style="list-style-type: none"> • Default — The MPC controller automatically computes the maximum number of QP solver iterations as: $\text{MaxIter} = 4(p \cdot n_{cy} + c(n_{cu} + n_{cr} + n_u) + n_{sv}) + n_{cr}$ <p>where</p> <ul style="list-style-type: none"> • p is the prediction horizon. • c is the control horizon. • n_{cy} is the number of OV constraints. • n_{cu} is the number of MV constraints. • n_{cr} is the number of MV rate constraints. • n_u is the number of MVs. • n_{sv} is the number of slack variables. • Positive integer — The QP solver stops after MaxIter iterations. 	
MinOutputECR	Minimum value allowed for OutputMinECR and OutputMaxECR , specified as a nonnegative scalar. A value of 0 indicates that hard output constraints are allowed. If either of the OutputVariables.MinECR or OutputVariables.MaxECR properties of an MPC controller are less than MinOutputECR , a warning is displayed and the value is raised to MinOutputECR during computation.	0
CustomSolver	Flag indicating whether to use a custom QP solver, specified as a logical value. If CustomSolver is true , the user must provide an mpcCustomSolver function on the MATLAB path. For information on how	false

Field	Description	Default
	to define the <code>mpcCustomSolver</code> function, see “Custom QP Solver”.	

Note: The default `MaxIter` value can be very large for some controller configurations, such as those with large prediction and control horizons. When simulating such controllers, if the QP solver cannot find a feasible solution, the simulation can appear to stop responding, since the solver continues searching for `MaxIter` iterations.

PredictionHorizon

`PredictionHorizon` is the integer number of prediction horizon steps. The control interval, `Ts`, determines the duration of each step. The default value is $10 + \text{maximum intervals of delay (if any)}$.

ControlHorizon

`ControlHorizon` is either a number of free control moves, or a vector of blocking moves (see “Optimization Variables”). The default value is 2.

History

`History` stores the time the MPC controller was created (read only).

Notes

`Notes` stores text or comments as a cell array of strings.

UserData

Any additional data stored within the MPC controller object.

Examples

Create MPC Controller with Specified Prediction and Control Horizons

Create a plant model with the transfer function $s + 1/(s^2 + 2s)$.

```
Plant = tf([1 1],[1 2 0]);
```

The plant is SISO, so its input must be a manipulated variable and its output must be measured. In general, it is good practice to designate all plant signal types using either the `setmpcsignals` command, or the LTI `InputGroup` and `OutputGroup` properties.

Specify a sample time for the controller.

```
Ts = 0.1;
```

Define bounds on the manipulated variable, u , such that $-1 \leq u \leq 1$.

```
MV = struct( Min , -1, Max , 1);
```

`MV` contains only the upper and lower bounds on the manipulated variable. In general, you can specify additional `MV` properties. When you do not specify other properties, their default values apply.

Specify a 20-interval prediction horizon, and a 3-interval control horizon.

```
p = 20;  
m = 3;
```

Create an MPC controller using the specified values. The fifth input argument is empty, so default tuning weights apply.

```
MPCobj = mpc(Plant,Ts,p,m,[],MV);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

- “Design MPC Controller at the Command Line”

More About

- “MPC Modeling”

See Also

`get` | `mpcprops` | `mpcverbosity` | `set` | `setmpcsignals`

Introduced before R2006a

MPC Designer

Design and simulate model predictive controllers

Description

The MPC Designer app lets you design and simulate model predictive controllers in MATLAB and Simulink.

Using this app, you can:

- Interactively design model predictive controllers and validate their performance using simulation scenarios
- Obtain linear plant models by linearizing Simulink models (requires Simulink Control Design™)
- Review controller designs for potential run-time stability or numerical issues
- Compare response plots for multiple model predictive controllers
- Generate Simulink models with an MPC controller and plant model
- Generate MATLAB scripts to automate MPC controller design and simulation tasks

Limitations

The following advanced MPC features are not available in the MPC Designer app:

- Explicit MPC design
- Adaptive MPC design
- Custom constraints (`setconstraint`)
- Terminal weight specification (`setterminal`)
- Custom state estimation (`setEstimator`)
- Sensitivity analysis (`sensitivity`)
- Alternative cost functions with off-diagonal weights
- Specification of initial plant and controller states for simulation
- Specification of nominal state values using `mpcObj.Model.Nominal.X` and `mpcObj.Model.Nominal.DX`

- Updating weights, constraints, MV targets, and external MV online during simulations

If your application requires any of these features, design your controller at the command line, and run simulations using `mpcmove` and `sim`. You can also run simulations in Simulink when using these features.

Open the MPC Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `mpcDesigner`.
- Simulink model editor: In the **MPC Controller Block** Parameters dialog box, click **Design**.

Examples

- “Design Controller Using MPC Designer”
- “Design MPC Controller in Simulink”
- “Compare Multiple Controller Responses Using MPC Designer”
- “Generate MATLAB Code from MPC Designer”
- “Generate Simulink Model from MPC Designer”

Programmatic Use

`mpcDesigner` opens the MPC Designer app. You can then import a plant or controller to start the design process, or open a saved design session.

`mpcDesigner(plant)` opens the MPC Designer app and creates a default MPC controller using `plant` as the internal prediction model. Specify `plant` as an `ss`, `tf`, or `zpk` LTI model.

By default, `plant` input and output signals are treated as manipulated variables and measured outputs respectively. To specify a different input/output channel configuration, use `setmpcsignals` before launching MPC Designer.

To use a System Identification Toolbox model as the plant, you must first convert it to an LTI model. For more information, see “Design Controller for Identified Plant” and “Design Controller Using Identified Model with Noise Channel”.

`mpcDesigner(MPCobj)` opens the MPC Designer app and imports the model predictive controller `MPCobj` from the MATLAB workspace. To create an MPC controller, use `mpc`.

`mpcDesigner(MPCobjs)` opens the MPC Designer app and imports multiple MPC controllers specified in the cell array `MPCobjs`. All of the controllers in `MPCobjs` must have the same input/output channel configuration.

`mpcDesigner(MPCobjs,names)` additionally specifies controller names when opening the app with multiple MPC controllers. Specify `names` as a cell array of strings with the same length as `MPCobjs`. Each element of `names` must be a unique string.

See Also

Functions
`mpc` | `sim`

Introduced in R2015b

mpcmove

Optimal control action

Syntax

```
u = mpcmove(MPCObj,x,ym,r,v)
[u,Info] = mpcmove(MPCObj,x,ym,r,v)
[u,Info] = mpcmove(MPCObj,x,ym,r,v,Options)
```

Description

`u = mpcmove(MPCObj,x,ym,r,v)` computes the optimal manipulated variable moves, $u(k)$, at the current time. $u(k)$ is calculated given the current estimated extended state, $x(k)$, the measured plant outputs, $y_m(k)$, the output references, $r(k)$, and the measured disturbances, $v(k)$, at the current time k . Call `mpcmove` repeatedly to simulate closed-loop model predictive control.

`[u,Info] = mpcmove(MPCObj,x,ym,r,v)` returns additional information regarding the model predictive controller in the second output argument `Info`.

`[u,Info] = mpcmove(MPCObj,x,ym,r,v,Options)` overrides default constraints and weights settings in `MPCObj` with the values specified by `Options`, an `mpcmoveopt` object. Use `Options` to provide run-time adjustment in constraints and weights during the closed-loop simulation.

Input Arguments

MPCObj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

x

`mpcstate` object that defines the current controller state.

Before you begin a simulation with `mpcmove`, initialize the controller state using `x = mpcstate(MPCobj)`. Then, modify the default properties of `x` as appropriate.

If you are using default state estimation, `mpcmove` expects `x` to represent `x[n|n-1]`. The `mpcmove` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a steady-state Kalman filter.

If you are using custom state estimation, `mpcmove` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmove` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

y_m

1-by- n_{ym} vector of current measured output values at time k , where n_{ym} is the number of measured outputs.

If you are using custom state estimation, set `ym = []`.

r

Plant output reference values, specified as a p -by- n_y array, where p is the prediction horizon of `MPCobj` and n_y is the number of outputs. Row `r(i, :)` defines the reference values at step i of the prediction horizon.

`r` must contain at least one row. If `r` contains fewer than p rows, `mpcmove` duplicates the last row to fill the p -by- n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, `r` must contain the anticipated variations, ideally for p steps.

v

Current and anticipated measured disturbances, specified as a p -by- n_{md} array, where p is the prediction horizon of `MPCobj` and n_{md} is the number of measured disturbances. Row `v(i, :)` defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use `v = []`.

v must contain at least one row. If v contains fewer than p rows, `mpcmove` duplicates the last row to fill the p -by- n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

Options

Override values for selected properties of `MPCObj`, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmove` time instant only. Using `Options` yields the same result as redefining or modifying `MPCObj` before each call to `mpcmove`, but involves considerably less overhead. Using `Options` is equivalent to using an `MPC Controller` Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

u — Optimal manipulated variable moves

row vector of length n_u

Optimal manipulated variable moves, returned as a row vector of length n_u , where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, u remains at its most recent successful solution, x .`LastMove`.

Info

Information regarding the model predictive controller, returned as a structure containing the following fields.

Uopt — Optimal manipulated variable adjustments

Optimal manipulated variable adjustments (moves), returned as a $p+1$ -by- n_u array, where p is the prediction horizon of `MPCObj` and n_u is the number of manipulated variables.

The first row of `Info.Uopt` is identical to the output argument `u`, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal values at time $k+i-1$, for $i = 1, \dots, p+1$. The `mpcmove` command does not calculate optimal control moves at time $k+p$, and therefore sets `Uopt(p+1,:)` to NaN.

Yopt

Predicted optimal output variable sequence, returned as a $p+1$ -by- n_y array, where p is the prediction horizon of `MPCobj` and n_y is the number of outputs.

The first row of `Info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time $k+i-1$, for $i = 1, \dots, p+1$.

Xopt – Optimal predicted state variable sequence

Optimal predicted state variable sequence, returned as a $p+1$ -by- n_x array, where p is the prediction horizon of `MPCobj` and n_x is the number of states.

The first row of `Info.Xopt` contains the current states at time `k` as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time $k+i-1$, for $i = 1, \dots, p+1$.

Topt – Time intervals

Time intervals, returned as a $p+1$ -by-a vector. `Topt(1) = 0`, representing the current time. Subsequent time steps `Topt(i)` are given by `Ts*(i-1)`, where `Ts = MPCobj.Ts`, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

Slack – Slack variable

Slack variable, ε , used in constraint softening, returned as 0 or a positive scalar value.

- $\varepsilon = 0$ — All constraints were satisfied for the entire prediction horizon.
- $\varepsilon > 0$ — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

Iterations — QP solution result

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- **Iterations > 0** — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- **Iterations = 0** — QP problem could not be solved in the allowed maximum number of iterations.
- **Iterations = -1** — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- **Iterations = -2** — Numerical error occurred when solving the QP problem.

QPCode — QP solution status

QP solution status, returned as one of the following strings:

- **feasible** — Optimal solution was obtained (**Iterations > 0**)
- **infeasible** — QP solver detected a problem with no feasible solution (**Iterations = -1**) or a numerical error occurred (**Iterations = -2**)
- **unreliable** — QP solver failed to converge (**Iterations = 0**)

Cost — Objective function cost

nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. See “Optimization Problem” for details.

The cost value is only meaningful when **QPCode = feasible**.

Examples

Analyze Closed-Loop Response

Perform closed-loop simulation of a plant with one MV and one measured OV.

Define a plant model and create a model predictive controller with MV constraints.

```
ts = 2;
Plant = ss(0.8,0.5,0.25,0,ts);
MPCobj = mpc(Plant);
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max = 2;

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Initialize an **mpcstate** object for simulation. Use the default state properties.

```
x = mpcstate(MPCobj);

-->Assuming output disturbance added to measured output channel #1 is integrated white noise
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each channel
```

Set the reference signal. There is no measured disturbance.

```
r = 1;
```

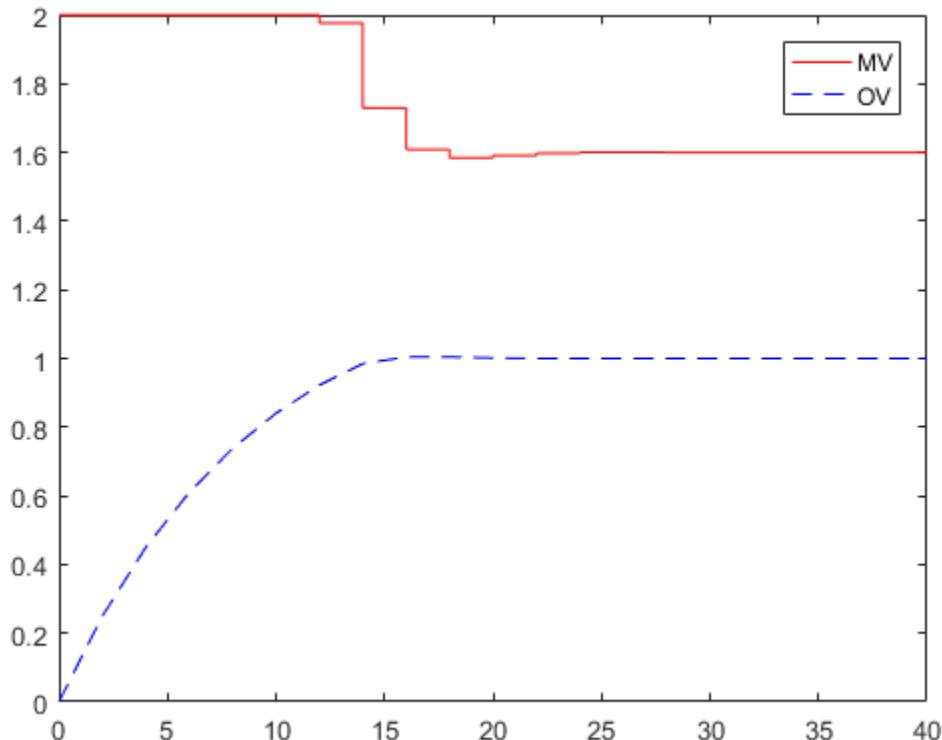
Simulate the closed-loop response by calling **mpcmove** iteratively.

```
t = [0:ts:40];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
    % simulated plant and predictive model are identical
    y(i) = 0.25*x.Plant;
    u(i) = mpcmove(MPCobj,x,y(i),r);
end
```

y and **u** store the OV and MV values.

Analyze the result.

```
[ts,us] = stairs(t,u);
plot(ts,us, r-, t,y, b-- )
legend( MV , OV )
```



Modify the MV upper bound as the simulation proceeds using an `mpcmoveopt` object.

```
MPCOpt = mpcmoveopt;
MPCOpt.MVMin = -2;
MPCOpt.MVMax = 2;
```

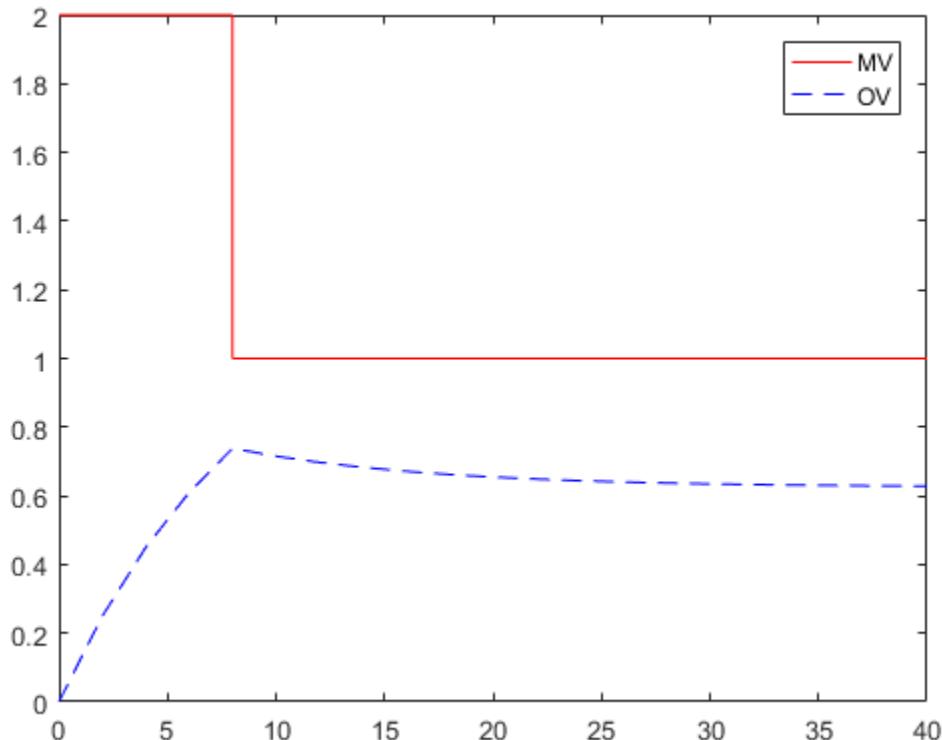
Simulate the closed-loop response and introduce the real-time upper limit change at eight seconds (the fifth iteration step).

```
x = mpcstate(MPCobj);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
    % simulated plant and predictive model are identical
```

```
y(i) = 0.25*x.Plant;
if i == 5
    MPCopt.MVMax = 1;
end
u(i) = mpmove(MPCobj,x,y(i),r,[],MPCopt);
end
```

Analyze the result.

```
[ts,us] = stairs(t,u);
plot(ts,us, r-, t,y, b-- )
legend( MV , OV )
```



Evaluate Scenario at Specific Time Instant

Define a plant model.

```
ts = 2;  
Plant = ss(0.8,0.5,0.25,0,ts);
```

Create a model predictive controller with MV and MVRate constraints. The prediction horizon is ten intervals. The control horizon is blocked.

```
MPCobj = mpc(Plant, ts, 10, [2 3 5]);  
MPCobj.MV(1).Min = -2;  
MPCobj.MV(1).Max = 2;  
MPCobj.MV(1).RateMin = -1;
```

```
MPCobj.MV(1).RateMax = 1;  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Initialize an mpcstate object for simulation from a particular state.

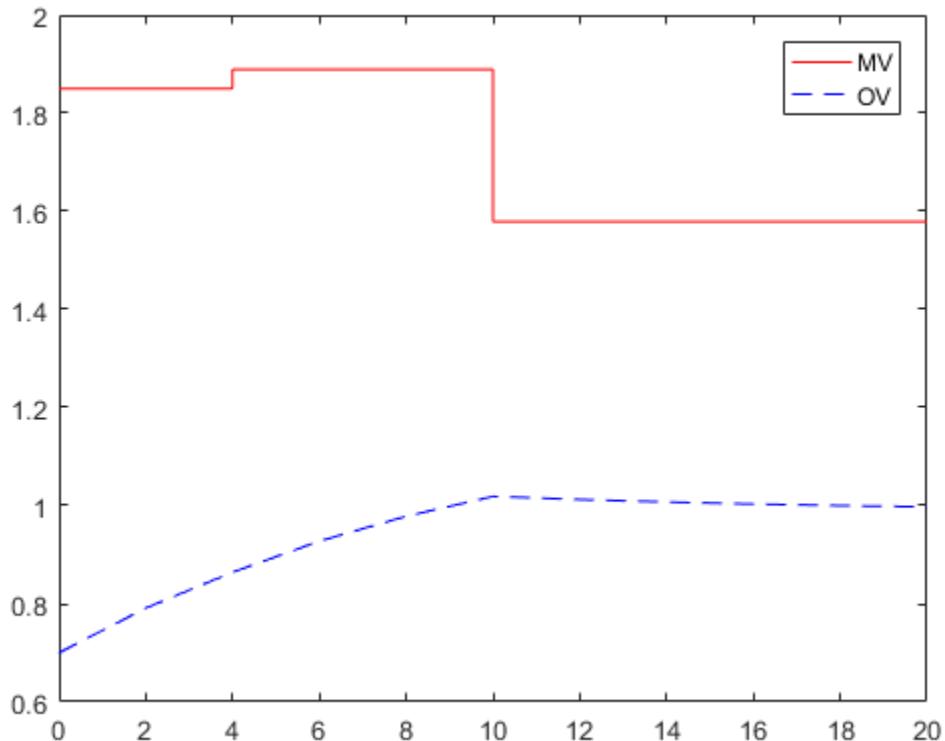
```
x = mpcstate(MPCobj);  
x.Plant = 2.8;  
x.LastMove = 0.85;  
-->Assuming output disturbance added to measured output channel #1 is integrated white noise  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each channel
```

Compute the optimal control at current time.

```
y = 0.25*x.Plant;  
r = 1;  
[u,Info] = mpmove(MPCobj,x,y,r);
```

Analyze the predicted optimal sequences.

```
[ts,us] = stairs(Info.Topt,Info.Uopt);  
plot(ts,us, r-, Info.Topt,Info.Yopt, b-- )  
legend( MV , OV )
```



plot ignores Info.Uopt(end) as it is NaN.

Examine the optimal cost.

Info.Cost

ans =

0.0793

- MPC Control with Anticipative Action (Look-Ahead)
- MPC Control with Input Quantization Based on Comparing the Optimal Costs

- Analysis of Control Sequences Optimized by MPC on a Double Integrator System

Alternatives

- Use `sim` for plant mismatch and noise simulation when not using run-time constraints or weight changes.
- Use the MPC Designer app to interactively design and simulate model predictive controllers.
- Use the `MPC Controller` block in Simulink and for code generation.

More About

Tips

- `mpcmove` updates `x`.
- If `ym`, `r` or `v` is specified as `[]`, `mpcmove` uses the appropriate `MPCObj.Model.Nominal` value instead.
- To view the predicted optimal behavior for the entire prediction horizon, plot the appropriate sequences provided in `Info`.
- To determine the optimization status, check `Info.Iterations` and `Info.QPCode`.

See Also

`mpc` | `mpcmoveopt` | `mpcstate` | `review` | `sim` | `setEstimator` | `getEstimator`

Introduced before R2006a

mpcmoveAdaptive

Compute optimal control with prediction model updating

Syntax

```
u = mpcmoveAdaptive(MPCObj,x,Plant,Nominal,ym,r,v)
[u,info] = mpcmoveAdaptive(MPCObj,x,Plant,Nominal,ym,r,v)
[___] = mpcmoveAdaptive(_____,opt)
```

Description

`u = mpcmoveAdaptive(MPCObj,x,Plant,Nominal,ym,r,v)` computes the optimal manipulated variable moves at the current time. This result depends on the properties contained in the MPC controller, the controller states, an updated prediction model, and the nominal values. The result also depends on the measured output variables, the output references (setpoints), and the measured disturbance inputs. `mpcmoveAdaptive` updates the controller state, `x`, when using default state estimation. Call `mpcmoveAdaptive` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveAdaptive(MPCObj,x,Plant,Nominal,ym,r,v)` returns additional details about the solution in a structure. To view the predicted optimal trajectory for the entire prediction horizon, plot the sequences provided in `info`. To determine whether the optimal control calculation completed normally, check `info.Iterations` and `info.QPCode`.

`[___] = mpcmoveAdaptive(_____,opt)` alters selected controller settings using options you specify with `mpcmoveopt`. These changes apply for the current time instant only, enabling a command-line simulation using `mpcmoveAdaptive` to mimic the `Adaptive MPC Controller` block in Simulink in a computationally efficient manner.

Input Arguments

MPCObj — MPC controller

MPC controller object

MPC controller, specified as an implicit MPC controller object. Use the `mpc` command to create the MPC controller.

x — Current MPC controller state
`mpcstate` object

Current MPC controller state, specified as an `mpcstate` object.

Before you begin a simulation with `mpcmoveAdaptive`, initialize the controller state using `x = mpcstate(MPCObj)`. Then, modify the default properties of `x` as appropriate.

If you are using default state estimation, `mpcmoveAdaptive` expects `x` to represent `x[n|n-1]`. The `mpcmoveAdaptive` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a linear time-varying Kalman filter.

If you are using custom state estimation, `mpcmoveAdaptive` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmoveAdaptive` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

Plant — Updated prediction model

discrete-time state-space model | model array

Updated prediction model, specified as one of the following:

- A delay-free, discrete-time state-space (`ss`) model. This plant is the update to `MPCObj.Model.Plant` and it must:
 - Have the same sample time as the controller; that is, `Plant.Ts` must match `MPCObj.Ts`
 - Have the same input and output signal configurations, such as type, order, and dimensions
 - Define the same states as the controller prediction model, `MPCObj.Model.Plant`
- An array of up to $p+1$ delay-free, discrete-time state-space models, where p is the prediction horizon of `MPCObj`. Use this option to vary the controller prediction model over the prediction horizon.

If `Plant` contains fewer than $p+1$ models, the last model repeats for the rest of the prediction horizon.

Tip If you use a plant other than a delay-free, discrete-time state-space model to define the prediction model in `MPCobj`, you can convert it to such a model to determine the prediction model structure.

If the original plant is	Then
Not a state-space model	Convert it to a state-space model using <code>ss</code> .
A continuous-time model	Convert it to a discrete-time model with the same sample time as the controller, <code>MPCobj.Ts</code> , using <code>c2d</code> with default forward Euler discretization.
A model with delays	Convert the delays to states using <code>absorbDelay</code> .

Nominal — Updated nominal conditions

structure | structure array | []

Updated nominal conditions, specified as one of the following:

- A structure of with the following fields:

Field	Description	Default
X	Plant state at operating point	[]
U	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	[]
Y	Plant output at operating point	[]
DX	For continuous-time models, DX is the state derivative at operating point: $DX=f(X,U)$. For discrete-time models, $DX=x(k+1)-x(k)=f(X,U)-X$.	[]

- An array of up to $p+1$ nominal condition structures, where p is the prediction horizon of `MPCObj`. Use this option to vary controller nominal conditions over the prediction horizon.

If `Nominal` contains fewer than $p+1$ structures, the last structure repeats for the rest of the prediction horizon.

If `Nominal` is empty, [], or if a field is missing or empty, `mpcmoveAdaptive` uses the corresponding `MPCObj.Model.Nominal` value.

y_m — Current measured outputs

ro vector of length n_{ym}

Current measured outputs, specified as a row vector of length n_{ym} vector, where n_{ym} is the number of measured outputs.

If you are using custom state estimation, `ym` is ignored. If you set `ym = []`, then `mpcmoveAdaptive` uses the appropriate nominal value.

r — Plant output reference values

p -by- n_y array | []

Plant output reference values, specified as a p -by- n_y array, where p is the prediction horizon of `MPCObj` and n_y is the number of outputs. Row `r(i, :)` defines the reference values at step i of the prediction horizon.

`r` must contain at least one row. If `r` contains fewer than p rows, `mpcmoveAdaptive` duplicates the last row to fill the p -by- n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

If you set `r = []`, then `mpcmoveAdaptive` uses the appropriate nominal value.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, `r` must contain the anticipated variations, ideally for p steps.

v — Current and anticipated measured disturbances

p -by- n_{md} array | []

Current and anticipated measured disturbances, specified as a p -by- n_{md} array, where p is the prediction horizon of `MPCObj` and n_{md} is the number of measured disturbances. Row `v(i, :)` defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use $v = []$.

v must contain at least one row. If v contains fewer than p rows, `mpcmoveAdaptive` duplicates the last row to fill the p -by- n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

If you set $v = []$, then `mpcmoveAdaptive` uses the appropriate nominal value.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

opt — Override values for selected controller properties

`mpcmoveopt` object

Override values for selected properties of `MPCObj`, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmoveAdaptive` time instant only. Using `opt` yields the same result as redefining or modifying `MPCObj` before each call to `mpcmoveAdaptive`, but involves considerably less overhead. Using `opt` is equivalent to using an **Adaptive MPC Controller** Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

u — Optimal manipulated variable moves

row vector of length n_u

Optimal manipulated variable moves, returned as a row vector of length n_u , where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, u remains at its most recent successful solution, x .`LastMove`.

info — Solution details

structure

Solution details, returned as a structure containing the following fields.

Uopt — Optimal manipulated variable adjustments (moves)

$(p+1)$ -by- n_u array

Optimal manipulated variable adjustments (moves), returned as a $(p+1)$ -by- n_u array, where p is the prediction horizon of `MPCobj` and n_u is the number of manipulated variables.

The first row of `info.Uopt` is identical to the output argument `u`, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal values at time $k+i-1$, for $i = 1, \dots, p+1$. The `mpcmoveAdaptive` command does not calculate optimal control moves at time $k+p$, and therefore sets `Uopt(p+1,:)` to NaN.

Yopt – Predicted output variable sequence

$(p+1)$ -by- n_y array

Predicted output variable sequence, returned as a $(p+1)$ -by- n_y array, where p is the prediction horizon of `MPCobj` and n_y is the number of outputs.

The first row of `info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time $k+i-1$, for $i = 1, \dots, p+1$.

Xopt – Predicted state variable sequence

$(p+1)$ -by- n_x array

Predicted state variable sequence, returned as a $(p+1)$ -by- n_x array, where p is the prediction horizon of `MPCobj` and n_x is the number of states.

The first row of `info.Xopt` contains the current states at time `k` as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time $k+i-1$, for $i = 1, \dots, p+1$.

Topt – Time intervals

column vector of length $p+1$

Time intervals, returned as a column vector of length $p+1$. `Topt(1) = 0`, representing the current time. Subsequent time steps, `Topt(i)`, are given by `Ts*(i-1)`, where `Ts = MPCobj.Ts`, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

Slack – Slack variable

0 | positive scalar

Slack variable, ε , used in constraint softening, returned as 0 or a positive scalar value.

- $\varepsilon = 0$ — All constraints were satisfied for the entire prediction horizon.
- $\varepsilon > 0$ — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

Iterations — QP solution result

positive integer | 0 | -1 | -2

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- **Iterations > 0** — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- **Iterations = 0** — QP problem could not be solved in the allowed maximum number of iterations.
- **Iterations = -1** — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- **Iterations = -2** — Numerical error occurred when solving the QP problem.

QPCode — QP solution status

feasible | infeasible | unreliable

QP solution status, returned as one of the following strings:

- **feasible** — Optimal solution was obtained (**Iterations > 0**)
- **infeasible** — QP solver detected a problem with no feasible solution (**Iterations = -1**) or a numerical error occurred (**Iterations = -2**)
- **unreliable** — QP solver failed to converge (**Iterations = 0**)

Cost — Objective function cost

nonnegative scalar

Objective function cost, returned as a nonnegative scalar. The cost quantifies the degree to which the controller has achieved its objectives. See “Optimization Problem” for details.

The cost value is only meaningful when **QPCode** = **feasible**.

More About

Tips

- If the prediction model is time-invariant, use `mpcmove`.
- Use the **Adaptive MPC Controller** Simulink block for simulations and code generation.
- “Adaptive MPC”
- “Time-Varying MPC”
- “Optimization Problem”

See Also

`getEstimator` | `mpc` | `mpcmove` | `mpcmoveopt` | `mpcstate` | `review` |
`setEstimator` | `sim`

Introduced in R2014b

mpcmoveCodeGeneration

Compute optimal control moves with code generation support

Syntax

```
[u,newStateData] = mpcmoveCodeGeneration(configData,stateData,  
onlineData)  
[___,info] = mpcmoveCodeGeneration(____)
```

Description

[u,newStateData] = mpcmoveCodeGeneration(configData,stateData, onlineData) computes optimal MPC control moves and supports code generation for deployment to real-time targets. The input data structures, generated using getCodeGenerationData, define the MPC controller to simulate.

mpcmoveCodeGeneration does not check input arguments for correct dimensions and data types.

[___,info] = mpcmoveCodeGeneration(____) returns additional information about the optimization result, including the number of iterations and the objective function cost.

Examples

Compute Optimal Control Moves Using Code Generation Data Structures

Create a proper plant model.

```
plant = rss(3,1,1);  
plant.D = 0;
```

Specify the controller sample time.

```
Ts = 0.1;
```

Create an MPC controller.

```
mpcObj = mpc(plant,Ts);  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Create code generation data structures.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj);  
-->Converting model to discrete time.  
-->Assuming output disturbance added to measured output channel #1 is integrated white  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each  
-->Converting model to discrete time.  
-->Assuming output disturbance added to measured output channel #1 is integrated white  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Initialize the plant states to zero to match the default states used by the MPC controller.

Run a closed-loop simulation. At each control interval, update the online data structure and call `mpcmovCodeGeneration` to compute the optimal control moves.

```
x = zeros(size(plant.B,1),1); % Initialize plant states to zero (|mpcObj| default).  
Tsim = 20;  
for i = 1:round(Tsim/Ts)+1  
    % Update plant output.  
    y = plant.C*x;  
    % Update measured output in online data.  
    onlineData.signals.ym = y;  
    % Update reference signal in online data.  
    onlineData.signals.ref = 1;  
    % Compute control actions.  
    [u,statedata] = mpcmoveCodeGeneration(configData,stateData,onlineData);  
    % Update plant state.  
    x = plant.A*x + plant.B*u;  
end
```

Generate MEX function with MATLAB® Coder™, specifying `configData` as a constant.

```
func = mpcmoveCodeGeneration ;  
funcOutput = mpcmoveMEX ;  
Cfg = coder.config( mex );  
Cfg.DynamicMemoryAllocation = off ;
```

```
codegen( -config ,Cfg,func, -o ,funcOutput, -args ,...
{coder.Constant(configData),stateData,onlineData});
```

Input Arguments

configData – MPC configuration parameters
structure

MPC configuration parameters that are constant at run time, specified as a structure generated using `getCodeGenData`.

Note: When using `codegen`, `configData` must be defined as `coder.Constant`.

stateData – Controller state
structure

Controller state at run time, specified as a structure. Generate the initial state structure using `getCodeGenData`. For subsequent control intervals, use the updated controller state from the previous interval. In general, use the `newStateData` structure directly.

If custom state estimation is enabled, you must manually update the state structure during each control interval. For more information, see “Using Custom State Estimation”.

onlineData – Online controller data
structure

Online controller data that you must update at run time, specified as a structure with the following fields:

signals – Updated input and output signals
structure

Updated input and output signals, specified as a structure with the following fields:

ym – Measured outputs
vector of length n_{ym}

Measured outputs, specified as a vector of length n_{ym} , where n_y is the number of measured outputs.

By default,`getCodeGenData` sets `ym` to the nominal measured output values from the controller.

ref – Output references

row vector of length n_y | p -by- n_y array

Output references, specified as a row vector of length n_y , where n_y is the number of outputs.

If you are using reference signal previewing with implicit or adaptive MPC, specify a p -by- n_y array, where p is the prediction horizon.

By default,`getCodeGenData` sets `ref` to the nominal output values from the controller.

md – Measured disturbances

row vector of length n_{md} | p -by- n_{md} array

Measured disturbances, specified as:

- A row vector of length n_{md} , where n_{md} is the number of measured disturbances.
- $A p$ -by- n_{md} array, if you are using signal previewing with implicit or adaptive MPC.

By default, if your controller has measured disturbances,`getCodeGenData` sets `md` to the nominal measured disturbance values from the controller. Otherwise, this field is empty and ignored by `mpcmoveCodeGen`.

mvTarget – Targets for manipulated variables

[] (default) | vector of length n_{mv}

Targets for manipulated variables, specified as:

- A vector of length n_{mv} , where n_{mv} is the number of manipulated variables.
- [] to use the default targets defined in the original MPC controller.

This field is ignored when using an explicit MPC controller.

externalMV – Manipulated variables externally applied to the plant

[] (default) | vector of length n_{mv}

Manipulated variables externally applied to the plant, specified as:

- A vector of length n_{mv} .
- [] to apply the optimal control moves to the plant.

limits — Updated input and output constraints

structure

Updated input and output constraints, specified as a structure. If you do not expect constraints to change at run time, ignore **limits**. This structure contains the following fields:

ymin — Lower bounds on output signals

column vector of length n_y | []

Lower bounds on output signals, specified as a column vector of length n_y .

If **ymin** is empty, [], the default bounds defined in the original MPC controller are used.

ymax — Upper bounds on output signals

column vector of length n_y | []

Upper bounds on output signals, specified as a column vector of length n_y .

If **ymax** is empty, [], the default bounds defined in the original MPC controller are used.

umin — Lower bounds on manipulated variables

column vector of length n_{mv} | []

Lower bounds on manipulated variables, specified as a column vector of length n_{mv} .

If **umin** is empty, [], the default bounds defined in the original MPC controller are used.

umax — Upper bounds on manipulated variables

column vector of length n_{mv} | []

Upper bounds on manipulated variables, specified as a column vector of length n_{mv} .

If **umax** is empty, [], the default bounds defined in the original MPC controller are used.

weights — Updated QP optimization weights

structure

Updated QP optimization weights, specified as a structure. If you do not expect tuning weights to change at run time, ignore **weights**. This structure contains the following fields:

ywt — Output weights

column vector of length n_y | []

Output weights, specified as a column vector of length n_y that contains nonnegative values.

If **ywt** is empty, [], the default weights defined in the original MPC controller are used.

uwt — Manipulated variable weights

column vector of length n_{mv} | []

Manipulated variable weights, specified as a column vector of length n_{mv} that contains nonnegative values.

If **uwt** is empty, [], the default weights defined in the original MPC controller are used.

duwt — Manipulated variable rate weights

column vector of length n_{mv} | []

Manipulated variable rate weights, specified as a column vector of length n_{mv} that contains nonnegative values.

If **duwt** is empty, [], the default weights defined in the original MPC controller are used.

ecr — Weight on slack variable used for constraint softening

nonnegative scalar | []

Weight on slack variable used for constraint softening, specified as a nonnegative scalar.

If **uwt** is empty, [], the default weight defined in the original MPC controller are used.

model — Updated plant and nominal values

structure

Updated plant and nominal values for adaptive MPC and time-varying MPC, specified as a structure. **model** is only available if you specify **isAdaptive** or **isLTV** as **true** when creating code generation data structures. This structure contains the following fields:

A — State matrix of discrete-time state-space plant model n_x -by- n_x array | n_x -by- n_x -by-($p+1$) array

State matrix of discrete-time state-space plant model, specified as an:

- n_x -by- n_x array when using adaptive MPC,
- n_x -by- n_x -by-($p+1$) array when using time-varying MPC,

where n_x is the number of plant states.**B — Input-to-state matrix of discrete-time state-space plant model** n_x -by- n_u array | n_x -by- n_u -by-($p+1$) array

Input-to-state matrix of discrete-time state-space plant model, specified as an:

- n_x -by- n_u array when using adaptive MPC,
- n_x -by- n_u -by-($p+1$) array when using time-varying MPC,

where n_u is the number of plant inputs.**C — State-to-output matrix of discrete-time state-space plant model** n_y -by- n_x array | n_y -by- n_x -by-($p+1$) array

State-to-output matrix of discrete-time state-space plant model, specified as an:

- n_y -by- n_x array when using adaptive MPC.
- n_y -by- n_x -by-($p+1$) array when using time-varying MPC.

D — Feedthrough matrix of discrete-time state-space plant model n_y -by- n_u array | n_y -by- n_u -by-($p+1$) array

Feedthrough matrix of discrete-time state-space plant model, specified as an:

- n_y -by- n_u array when using adaptive MPC.
- n_y -by- n_u -by-($p+1$) array when using time-varying MPC.

Since MPC controllers do not support plants with direct feedthrough, specify **D** as an array of zeros.**X — Nominal plant states**column vector of length n_x | n_x -by-1-by-($p+1$) array

Nominal plant states, specified as:

- A column vector of length n_x when using adaptive MPC.
- An n_x -by-1-by-($p+1$) array when using time-varying MPC.

U — Nominal plant inputs

column vector of length n_u | n_u -by-1-by-($p+1$) array

Nominal plant inputs, specified as:

- A column vector of length n_u when using adaptive MPC.
- An n_u -by-1-by-($p+1$) array when using time-varying MPC.

Y — Nominal plant outputs

column vector of length n_y | n_y -by-1-by-($p+1$) array

Nominal plant outputs, specified as:

- A column vector of length n_y when using adaptive MPC.
- An n_y -by-1-by-($p+1$) array when using time-varying MPC.

DX — Nominal plant state derivatives

column vector of length n_x | n_x -by-1-by-($p+1$) array

Nominal plant state derivatives, specified as:

- A column vector of length n_x when using adaptive MPC.
- An n_x -by-1-by-($p+1$) array when using time-varying MPC.

Output Arguments

u — Optimal manipulated variable moves

row vector of length n_u

Optimal manipulated variable moves, returned as a row vector of length n_u , where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, u remains at its most recent successful solution, x .`LastMove`.

newStateData — Updated controller state

structure

Updated controller state, returned as a structure. For subsequent control intervals, pass `newStateData` to `mpcmoveCodeGeneration` as `stateData`.

If custom state estimation is enabled, use `newStateData` to manually update the state structure before the next control interval. For more information, see “Using Custom State Estimation”.

info — Controller optimization information

structure

Controller optimization information, returned as a structure.

If you are using implicit or adaptive MPC, `info` contains the following fields:

Field	Description
Iterations	Number of QP solver iterations
QPCode	QP solver status code
Cost	Objective function cost
Uopt	Optimal manipulated variable adjustments
Yopt	Optimal predicted output variable sequence
Xopt	Optimal predicted state variable sequence
Topt	Time horizon intervals
Slack	Slack variable used in constraint softening

If `configData.OnlyComputeCost` is `true`, the optimal sequence information, `Uopt`, `Yopt`, `Xopt`, `Topt`, and `Slack`, is not available:

For more information, see `mpcmove` and `mpcmoveAdaptive`.

If you are using explicit MPC, `info` contains the following fields:

Field	Description
Region	Region in which the optimal solution was found
ExitCode	Solution status code

For more information, see `mpcmovExplicit`.

More About

- “Generate Code To Compute Optimal MPC Moves in MATLAB”
- “Generate Code and Deploy Controller to Real-Time Targets”

See Also

`getCodeGenerationData` | `mpcmov` | `mpcmovAdaptive` | `mpcmovExplicit`

Introduced in R2016a

mpcmoveExplicit

Compute optimal control using explicit MPC

Syntax

```
u = mpcmoveExplicit(EMPCobj,x,ym,r,v)
[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v)
[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v,MVused)
```

Description

`u = mpcmoveExplicit(EMPCobj,x,ym,r,v)` computes the optimal manipulated variable moves at the current time using an explicit model predictive control law. This result depends on the properties contained in the explicit MPC controller and the controller states. The result also depends on the measured output variables, the output references (setpoints), and the measured disturbance inputs. `mpcmoveExplicit` updates the controller state, `x`, when using default state estimation. Call `mpcmoveExplicit` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v)` returns additional details about the computation in a structure. To determine whether the optimal control calculation completed normally, check the data in `info`.

`[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v,MVused)` specifies the manipulated variable values used in the previous `mpcmoveExplicit` command, allowing a command-line simulation to mimic the **Explicit MPC Controller** Simulink block with the optional external MV input signal.

Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant”

Input Arguments

EMPCobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller to simulate, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

x — Current MPC controller state
`mpcstate` object

Current MPC controller state, specified as an `mpcstate` object.

Before you begin a simulation with `mpcmoveExplicit`, initialize the controller state using `x = mpcstate(EMPCobj)`. Then, modify the default properties of `x` as appropriate.

If you are using default state estimation, `mpcmoveExplicit` expects `x` to represent `x[n|n-1]`. The `mpcmoveExplicit` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a linear time-varying Kalman filter.

If you are using custom state estimation, `mpcmoveExplicit` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmoveExplicit` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

y_m — Current measured outputs
vector

Current measured outputs, specified as a 1-by- n_{ym} vector. n_{ym} is the number of measured outputs. If you are using custom state estimation, `ym` is ignored. If you set `ym = []`, then `mpcmoveExplicit` uses the appropriate nominal value.

r — Plant output reference values
vector

Plant output reference values, specified as a vector of n_y values. `mpcmoveExplicit` uses a constant reference for the entire prediction horizon. In contrast to `mpcmove` and `mpcmoveAdaptive`, `mpcmoveExplicit` does not support reference previewing.

If you set `r = []`, then `mpcmoveExplicit` uses the appropriate nominal value.

v — Current and anticipated measured disturbances
vector

Current and anticipated measured disturbances, specified as a vector of n_{md} values. In contrast to `mpcmove` and `mpcmoveAdaptive`, `mpcmoveExplicit` does not support

disturbance previewing. If your plant model does not include measured disturbances, use $v = []$.

MVused — Manipulated variable values from previous interval
vector

Manipulated variable values applied to the plant during the previous control interval, specified as a vector of n_u values. If this is the first `mpcmoveExplicit` command in a simulation sequence, omit this argument. Otherwise, if the MVs calculated by `mpcmoveExplicit` in the previous interval were overridden, set `MVused` to the correct values in order to improve the controller state estimation accuracy. If you omit `MVused`, `mpcmoveExplicit` assumes `MVused = x.LastMove`.

Output Arguments

u — Optimal manipulated variable moves
row vector of length n_u

Optimal manipulated variable moves, returned as a row vector of length n_u , where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, `u` remains at its most recent successful solution, `x.LastMove`.

info — Explicit MPC solution status
structure

Explicit MPC solution status, returned as a structure having the following fields.

ExitCode — Solution status code
1 | 0 | -1

Solution status code, returned as one of the following values:

- 1 — Successful solution.
- 0 — Failure. One or more controller input parameters is out of range.
- -1 — Undefined. Parameters are in range but an extrapolation must be used.

Region — Region to which current controller input parameters belong
positive integer | 0

Region to which current controller input parameters belong, returned as either a positive integer or 0. The integer value is the index of the polyhedron (region) to which the current controller input parameters belong. If the solution failed, `Region` = 0.

More About

Tips

- Use the `Explicit MPC Controller` Simulink block for simulations and code generation.
- “Explicit MPC”
- “Design Workflow for Explicit MPC”

See Also

`generateExplicitMPC`

Introduced in R2014b

mpcmoveMultiple

Compute gain-scheduling MPC control action at a single time instant

Syntax

```
u = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v,opt)
```

Description

`u = mpcmoveMultiple(MPCArray,states,index,ym,r,v)` computes the optimal manipulated variable moves at the current time using a model predictive controller selected by index from an array of MPC controllers. This results depends upon the properties contained in the MPC controller and the controller states. The result also depends on the measured plant outputs, the output references (setpoints), and the measured disturbance inputs. `mpcmoveMultiple` updates the controller state when default state estimation is used. Call `mpcmoveMultiple` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v)` returns additional details about the computation in a structure. To determine whether the optimal control calculation completed normally, check the data in `info`.

`[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v,opt)` alters selected controller settings using options you specify with `mpcmoveopt`. These changes apply for the current time instant only, allowing a command-line simulation using `mpcmoveMultiple` to mimic the `Multiple MPC Controllers` block in Simulink in a computationally efficient manner.

Input Arguments

MPCArray — MPC controllers
cell array of MPC controller objects

MPC controllers to simulate, specified as a cell array of traditional (implicit) MPC controller objects. Use the `mpc` command to create the MPC controllers.

All the controllers in `MPCArray` must use either default state estimation or custom state estimation. Mismatch is not permitted.

states — Current MPC controller states

cell array of `mpcstate` objects

Current controller states for each MPC controller in `MPCArray`, specified as a cell array of `mpcstate` objects.

Before you begin a simulation with `mpcmoveMultiple`, initialize each controller state using `x = mpcstate(MPCobj)`. Then, modify the default properties of each state as appropriate.

If you are using default state estimation, `mpcmoveAdaptive` expects `x` to represent `x[n|n-1]` (where `x` is one entry in `states`, the current state of one MPC controller in `MPCArray`). The `mpcmoveMultiple` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a steady-state Kalman filter.

If you are using custom state estimation, `mpcmoveMultiple` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmoveMultiple` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

index — Index of selected controller

positive integer

Index of selected controller in the cell array `MPCArray`, specified as a positive integer.

ym — Current measured outputs

vector

Current measured outputs, specified as a 1-by- n_{ym} vector. n_{ym} is the number of measured outputs. If you are using custom state estimation, `ym` is ignored. If you set `ym = []`, then `mpcmoveMultiple` uses the appropriate nominal value.

r — Plant output reference values

array

Plant output reference values, specified as a p -by- n_y array, where p is the prediction horizon of the selected controller and n_y is the number of outputs. Row $r(i,:)$ defines the reference values at step i of the prediction horizon.

r must contain at least one row. If r contains fewer than p rows, `mpcmoveMultiple` duplicates the last row to fill the p -by- n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

If you set $r = []$, then `mpcmoveMultiple` uses the appropriate nominal value.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, r must contain the anticipated variations, ideally for p steps.

v — Current and anticipated measured disturbances
array

Current and anticipated measured disturbances, specified as a p -by- n_{md} array, where p is the prediction horizon of the selected controller and n_{md} is the number of measured disturbances. Row $v(i,:)$ defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use $v = []$.

v must contain at least one row. If v contains fewer than p rows, `mpcmoveMultiple` duplicates the last row to fill the p -by- n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

If you set $v = []$, then `mpcmoveMultiple` uses the appropriate nominal value.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

opt — Override values for selected controller properties
`mpcmoveopt` object

Override values for selected properties of the selected MPC controller, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmoveMultiple` time instant only. Using `opt` yields the same result as redefining or modifying the selected controller before each call to `mpcmoveMultiple`, but

involves considerably less overhead. Using `opt` is equivalent to using a **Multiple MPC Controllers** Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

u — Optimal manipulated variable moves

row vector of length n_u

Optimal manipulated variable moves, returned as a row vector of length n_u , where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, `u` remains at its most recent successful solution, `x.LastMove`.

info — Solution details

structure

Solution details, returned as a structure containing the following fields.

Uopt — Optimal manipulated variable adjustments (moves)

array

Optimal manipulated variable adjustments (moves), returned as a $p+1$ -by- n_u array, where p is the prediction horizon of the selected controller and n_u is the number of manipulated variables.

The first row of `info.Uopt` is identical to the output argument `u`, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal values at time $k+i-1$, for $i = 1, \dots, p+1$. The `mpcmovemultiple` command does not calculate optimal control moves at time $k+p$, and therefore sets `Uopt(p+1,:)` to NaN.

Yopt — Predicted output variable sequence

array

Predicted output variable sequence, returned as a $p+1$ -by- n_y array, where p is the prediction horizon of the selected controller and n_y is the number of outputs.

The first row of `info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time $k+i-1$, for $i = 1, \dots, p+1$.

Xopt — Predicted state variable sequence

array

Predicted state variable sequence, returned as a $p+1$ -by- n_x array, where p is the prediction horizon of the selected controller and n_x is the number of states.

The first row of `info.Xopt` contains the current states at time k as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time $k+i-1$, for $i = 1, \dots, p+1$.

Topt — Time intervals

vector

Time intervals, returned as a $p+1$ -by-a vector. `Topt(1) = 0`, representing the current time. Subsequent time steps `Topt(i)` are given by $Ts^*(i-1)$, where $Ts = MPCobj.Ts$, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

Slack — Slack variable

0 | positive scalar

Slack variable, ε , used in constraint softening, returned as 0 or a positive scalar value.

- $\varepsilon = 0$ — All constraints were satisfied for the entire prediction horizon.
- $\varepsilon > 0$ — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

Iterations — QP solution result

positive integer | 0 | -1 | -2

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- `Iterations > 0` — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- `Iterations = 0` — QP problem could not be solved in the allowed maximum number of iterations.

- **Iterations = -1** — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- **Iterations = -2** — Numerical error occurred when solving the QP problem.

QPCode — QP solution status

`feasible | infeasible | unreliable`

QP solution status, returned as one of the following strings:

- **feasible** — Optimal solution was obtained (**Iterations > 0**)
- **infeasible** — QP solver detected a problem with no feasible solution (**Iterations = -1**) or a numerical error occurred (**Iterations = -2**)
- **unreliable** — QP solver failed to converge (**Iterations = 0**)

Cost — Objective function cost

`nonnegative scalar`

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. See “Optimization Problem” for details.

The cost value is only meaningful when **QPCode = feasible**.

More About

Tips

- Use the **Multiple MPC Controllers** Simulink block for simulations and code generation.

See Also

`generateExplicitMPC | getEstimator | mpcmove | mpcstate | review | setEstimator | sim`

Introduced in R2014b

mpcmoveopt

Options set for `mpcmove` and `mpcmoveAdaptive`

Syntax

```
options = mpcmoveopt
```

Description

`options = mpcmoveopt` creates an empty `mpcmoveopt` object. You can set one or more of its properties using dot notation, and then use the object with `mpcmove` or `mpcmoveAdaptive` to simulate run-time adjustment of selected controller properties, such as tuning weights and bounds.

`mpcmoveopt` property dimensions must be consistent with the number of manipulated variables (`nu`) and output variables (`ny`) defined in the `mpc` or `mpcAdaptive` controller you are simulating.

In general, if you do not specify a value for one of the `mpcmoveopt` properties, it defaults to the corresponding built-in value of the simulated controller.

Output Arguments

options

Options for the `mpcmove` or `mpcmoveAdaptive` command with the following fields:

- **OutputWeights** — Output variable tuning weights, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. These replace the controller's `Weight.OutputVariables` property. The weights must be nonnegative, finite real values.
- **MVWeights** — Manipulated variable tuning weights, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. These replace the controller's

`Weight.ManipulatedVariables` property. The weights must be nonnegative, finite real values.

- `MVRateWeights` — Manipulated variable rate tuning weights, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. These replace the controller's `Weight.ManipulatedVariablesRate` property. The weights must be nonnegative, finite real values.
- `ECRWeight` — Weight on the slack variable used for constraint softening, specified as a finite, real scalar. This value replaces the controller's `Weight.ECR` property.
- `OutputMin` — Lower bounds on the output variables, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. `OutputMin(i)` replaces the controller's `OutputVariables(i).Min` property, for $i = 1, \dots, ny$.
- `OutputMax` — Upper bounds on the output variables, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. `OutputMax(i)` replaces the controller's `OutputVariables(i).Max` property, for $i = 1, \dots, ny$.
- `MVMin` — Lower bounds on the manipulated variables, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. `MVMin(i)` replaces the controller's `ManipulatedVariables(i).Min` property, for $i = 1, \dots, nu$.
- `MVMax` — Upper bounds on the manipulated variables, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. `MVMax(i)` replaces the controller's `ManipulatedVariables(i).Max` property, for $i = 1, \dots, nu$.
- `MVUsed` — Manipulated variable values used in the plant during the previous control interval, specified as a 1-by-`nu` vector. This property mimics the `MPC Controller` or `Adaptive MPC Controller` Simulink blocks' external MV signal. If you do not provide an `MVUsed` value, the controller uses the `LastMove` property of `mpcstate`.
- `OnlyComputeCost` — Logical value that controls whether the optimal sequence is to be calculated and exported.
 - 0 (default) causes the controller to return the predicted optimal policy in addition to the objective function cost value.
 - 1 causes the controller to return the objective function cost only, which saves computational effort.

Examples

Simulation with Varying Controller Property

Vary a manipulated variable upper bound during a simulation.

Define the plant, which includes a 4-second input delay. Convert to a delay-free, discrete, state-space model using a 2-second control interval. Create the corresponding default controller and then specify MV bounds at +/-2.

```
ts = 2;
Plant = absorbDelay(c2d(ss(tf(0.8,[5 1], InputDelay ,4)),ts));
MPCObj = mpc(Plant, ts);
MPCObj.MV(1).Min = -2;
MPCObj.MV(1).Max = 2;

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Create an empty `mpcmoveopt` object. During simulation, you can set properties of the object to specify controller parameters.

```
options = mpcmoveopt;
```

Pre-allocate storage and initialize the controller state.

```
v = [];
t = [0:ts:20];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
x = mpcstate(MPCObj);

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Use `mpcmove` to simulate the following:

- Reference (setpoint) step change from initial condition $r = 0$ to $r = 1$ (servo response).
- MV upper bound step decrease from 2 to 1, occurring at $t = 10$.

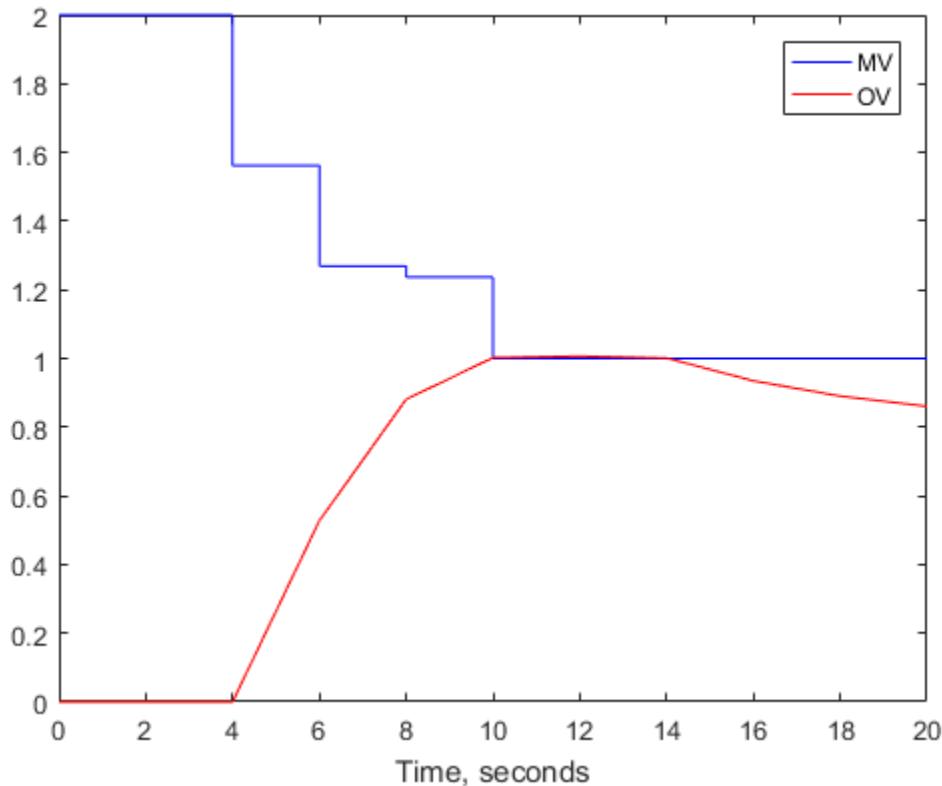
```
r = 1;
for i = 1:N
    y(i) = Plant.c*x.Plant;
    if t(i) >= 10
        options.MVMax = 1;
    end
```

```
[u(i),Info] = mpcmove(MPCobj,x,y(i),r,v,options);  
end
```

As the loop executes, the value of `options.MVMax` is reset to 1 for all iterations that occur after $t = 10$. Prior to that iteration, `options.MVMax` is empty. Therefore, the controller's value for `MVMax` is used, `MPCobj.MV(1).Max = 2`.

Plot the results of the simulation.

```
[ts,us] = stairs(t,u);  
plot(ts,us, b-,t,y, r-)  
legend( MV , OV )  
xlabel(sprintf( Time, %s ,Plant.TimeUnit))
```



From the plot, you can observe that the original MV upper bound is active until $t = 4$. After the input delay of 4 seconds, the output variable (OV) moves smoothly to its new target of $r = 1$. reaching the target at $t = 10$. The new MV bound imposed at $t = 10$ becomes avtive immediately. This forces the OV below its target, after the input delay elapses.

Now assume that you want to impose an OV upper bound at a specified location relative to the OV target. Consider the following constraint design command:

```
MPCobj.OV(1).Max = [Inf, Inf, 0.4, 0.3, 0.2];
```

This is a horizon-varying constraint. The known input delay makes it impossible for the controller to satisfy an OV constraint prior to the third prediction-horizon step. Therefore, a finite constraint during the first two steps would be poor practice. For illustrative purposes, the above constraint also decreases from 0.4 at step 3 to 0.2 at step 5 and thereafter.

The following commands produce the same results shown in the previous plot. The OV constraint is never active because it is being varied in concert with the setpoint, r .

```
x = mpcstate(MPCobj);
OPTobj = mpcmoveopt;
for i = 1:N
    y(i) = Plant.c*x.Plant;
    if t(i) >= 10
        OPTobj.MVMax = 1;
    end
    OPTobj.OutputMax = r + 0.4;
    [u(i), Info] = mpcmove(MPCobj, x, y(i), r, v, OPTobj);
end

-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

The scalar value $r + 0.4$ replaces the first finite value in the **MPCobj.OV(1).Max** vector, and the remaining finite values adjust to maintain the original profile, i.e., the numerical difference between these values is unchanged. $r = 1$ for the simulation, so the above use of the **mpcmoveopt** object is equivalent to the command

```
MPCobj.OV(1).Max = [Inf, Inf, 1.4, 1.3, 1.2];
```

The use of the `mpcmoveopt` object involves much less computational overhead, however.

Alternatives

The `mpcmoveopt` object is an optional feature of the `mpcmove` and `mpcmoveAdaptive` commands. The alternative is to redefine the controller and/or state object prior to each command invocation, but this involves considerable overhead.

More About

Tips

- `mpcmoveopt` cannot constrain a variable that was unconstrained in the controller creation step. The controller ignores any such specifications. Similarly, you cannot eliminate a constraint defined during controller creation, but you can change it to a very large (or small) value such that it is unlikely to become active.
- If the controller design includes a vector constraint, the run-time `mpcmoveopt` value replaces the first finite entry, and the remaining values shift to retain the same constraint profile. See “Simulation with Varying Controller Property” on page 1-118.

See Also

`mpc` | `mpcmove` | `setconstraint` | `setterminal`

Introduced in R2011b

mpcprops

Provide help on MPC controller properties

Syntax

`mpcprops`

Description

`mpcprops` displays details on the generic properties of MPC controllers. It provides a complete list of all the fields of MPC objects with a brief description of each field and the corresponding default values.

See Also

`set` | `get`

Introduced before R2006a

mpcqpsolver

Solve a quadratic programming problem using the KWIK algorithm

Syntax

```
[x,status] = mpqp solver(Linv,f,A,b,Aeq,beq,iA0,options)  
[x,status,iA,lambda] = mpqp solver(Linv,f,A,b,Aeq,beq,iA0,options)
```

Description

`[x,status] = mpqp solver(Linv,f,A,b,Aeq,beq,iA0,options)` finds an optimal solution, x , to a quadratic programming problem by minimizing the objective function:

$$J = \frac{1}{2} x^\top H x + f^\top x$$

subject to inequality constraints $Ax \geq b$, and equality constraints $A_{eq}x = b_{eq}$. `status` indicates the validity of x .

`[x,status,iA,lambda] = mpqp solver(Linv,f,A,b,Aeq,beq,iA0,options)` also returns the active inequalities, `iA`, at the solution, and the Lagrange multipliers, `lambda`, for the solution.

Examples

Solve Quadratic Programming Problem

Find the values of x that minimize

$$f(x) = 0.5x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2,$$

subject to the constraints

$$\begin{aligned}x_1 &\geq 0 \\x_2 &\geq 0 \\x_1 + x_2 &\leq 2 \\-x_1 + 2x_2 &\leq 2 \\2x_1 + x_2 &\leq 3.\end{aligned}$$

Specify the Hessian and linear multiplier vector for the objective function.

```
H = [1 -1; -1 2];
f = [-2; -6];
```

Specify the inequality constraint parameters.

```
A = [1 0; 0 1; -1 -1; 1 -2; -2 -1];
b = [0; 0; -2; -2; -3];
```

Define Aeq and beq to indicate that there are no equality constraints.

```
Aeq = [];
beq = zeros(0,1);
```

Find the lower-triangular Cholesky decomposition of H.

```
[L,p] = chol(H, lower );
Linv = inv(L);
```

It is good practice to verify that H is positive definite by checking if p = 0.

p

p =

0

Create a default option set for mpqp solver.

```
opt = mpqpOptions;
```

To cold start the solver, define all inequality constraints as inactive.

```
iA0 = false(size(b));
```

Solve the QP problem.

```
[x,status] = mpcqpssolver(Linv,f,A,b,Aeq,beq,iA0,opt);
```

Examine the solution, x .

```
x
```

```
x =
```

```
0.6667  
1.3333
```

Check Active Inequality Constraints for QP Solution

Find the values of x that minimize

$$f(x) = 3x_1^2 + 0.5x_2^2 - 2x_1x_2 - 3x_1 + 4x_2,$$

subject to the constraints

$$\begin{aligned}x_1 &\geq 0 \\x_1 + x_2 &\leq 5 \\x_1 + 2x_2 &\leq 7.\end{aligned}$$

Specify the Hessian and linear multiplier vector for the objective function.

```
H = [6 -2; -2 1];  
f = [-3; 4];
```

Specify the inequality constraint parameters.

```
A = [1 0; -1 -1; -1 -2];  
b = [0; -5; -7];
```

Define A_{eq} and b_{eq} to indicate that there are no equality constraints.

```
Aeq = [];  
beq = zeros(0,1);
```

Find the lower-triangular Cholesky decomposition of H .

```
[L,p] = chol(H, lower );  
Linv = inv(L);
```

Verify that H is positive definite by checking if $p = 0$.

p

$p =$

0

Create a default option set for `mpcqpsolver`.

```
opt = mpcqpsolverOptions;
```

To cold start the solver, define all inequality constraints as inactive.

```
iA0 = false(size(b));
```

Solve the QP problem.

```
[x,status,iA,lambda] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,opt);
```

Check the active inequality constraints. An active inequality constraint is at equality for the optimal solution.

iA

$iA =$

1
0
0

There is a single active inequality constraint.

View the Lagrange multiplier for this constraint.

```
lambda.ineqlin(1)
```

$ans =$

5.0000

- “Solve Custom MPC Quadratic Programming Problem and Generate Code”

Input Arguments

Linv — Inverse of lower-triangular Cholesky decomposition of Hessian matrix
n-by-*n* matrix

Inverse of lower-triangular Cholesky decomposition of Hessian matrix, specified as an *n*-by-*n* matrix, where *n* > 0 is the number of optimization variables. For a given Hessian matrix, *H*, Linv can be computed as follows:

```
[L,p] = chol(H, lower );  
Linv = inv(L);
```

H is an *n*-by-*n* matrix, which must be symmetric and positive definite. If *p*>0, then *H* is positive definite.

Note: The KWIK algorithm requires the computation of Linv instead of using *H* directly, as in the quadprog command.

f — Multiplier of objective function linear term
column vector

Multiplier of objective function linear term, specified as a column vector of length *n*.

A — Linear inequality constraint coefficients
m-by-*n* matrix | []

Linear inequality constraint coefficients, specified as an *m*-by-*n* matrix, where *m* is the number of inequality constraints.

If your problem has no inequality constraints, use [].

b — Right-hand side of inequality constraints
column vector of length *m*

Right-hand side of inequality constraints, specified as a column vector of length *m*.

If your problem has no inequality constraints, use `zeros(0,1)`.

Aeq — Linear equality constraint coefficients

q-by-*n* matrix | []

Linear equality constraint coefficients, specified as a *q*-by-*n* matrix, where *q* is the number of equality constraints, and *q* $\leq n$. Equality constraints must be linearly independent with `rank(Aeq) = q`.

If your problem has no equality constraints, use [].

beq — Right-hand side of equality constraints

column vector of length *q*

Right-hand side of equality constraints, specified as a column vector of length *q*.

If your problem has no equality constraints, use `zeros(0,1)`.

iAO — Initial active inequalities

logical vector of length *m*

Initial active inequalities, where the equal portion of the inequality is true, specified as a logical vector of length *m* according to the following:

- If your problem has no inequality constraints, use `false(0,1)`.
- For a *cold start*, `false(m,1)`.
- For a *warm start*, set `iAO(i) == true` to start the algorithm with the *i*th inequality constraint active. Use the optional output argument `iA` from a previous solution to specify `iAO` in this way. If both `iAO(i)` and `iAO(j)` are `true`, then rows *i* and *j* of `A` should be linearly independent. Otherwise, the solution can fail with `status = -2`.

options — Option set for mpqcqp solver

structure

Option set for `mpcqpsolver`, specified as a structure created using `mpcqpsolverOptions`.

Output Arguments

x — Optimal solution to the QP problem

column vector

Optimal solution to the QP problem, returned as a column vector of length n . `mpcqpsolver` always returns a value for `x`. To determine whether the solution is optimal or feasible, check the solution `status`.

status — Solution validity indicator

positive integer | 0 | -1 | -2

Solution validity indicator, returned as an integer according to the following:

Value	Description
> 0	<code>x</code> is optimal. <code>status</code> represents the number of iterations performed during optimization.
0	The maximum number of iterations was reached. The solution, <code>x</code> , may be suboptimal or infeasible.
-1	The problem appears to be infeasible, that is, the constraint $Ax \geq b$ cannot be satisfied.
-2	An unrecoverable numerical error occurred.

iA — Active inequalities

logical vector of length m

Active inequalities, where the equal portion of the inequality is true, returned as a logical vector of length m . If `iA(i) == true`, then the i th inequality is active for the solution `x`.

Use `iA` to *warm start* a subsequent `mpcqpsolver` solution.

lambda — Lagrange multipliers

structure

Lagrange multipliers, returned as a structure with the following fields:

Field	Description
<code>ineqlin</code>	Multipliers of the inequality constraints, returned as a vector of length n . When the solution is optimal, the elements of <code>ineqlin</code> are nonnegative.
<code>eqlin</code>	Multipliers of the equality constraints, returned as a vector of length q . There are no sign restrictions in the optimal solution.

More About

Tips

- The KWIK algorithm requires that the Hessian matrix, H , be positive definite. When calculating `Linv`, use:

```
[L, p] = chol(H, lower);
```

If $p = 0$, then H is positive definite. Otherwise, p is a positive integer.

- `mpcqpsolver` provides access to the QP solver used by Model Predictive Control Toolbox software. Use this command to solve QP problems in your own custom MPC applications. For an example of a custom MPC application using `mpcqpsolver`, see “Solve Custom MPC Quadratic Programming Problem and Generate Code”.
- You can also use `mpcqpsolver` as a general-purpose QP solver that supports code generation. Create a function, `myCode`, that uses `mpcqpsolver`.

```
function [out1,out2] = myCode(in1,in2)
 %#codegen
 ...
 [x,status] = mpqpsover(Linv,f,A,b,Aeq,Beq,iA0,options);
 ...
```

Generate C code with MATLAB Coder™.

```
func = myCode ;
cfg = coder.config( mex ); % or lib , dll
codegen( -config ,cfg,func, -o ,func);
```

When using `mpcqpsolver` for code generation, use the same precision for all real inputs, including options. Configure the precision as `double` or `single` using `mpcqpsolverOptions`.

Algorithms

`mpcqpsolver` solves the QP problem using an active-set method, the KWIK algorithm, based on [1]. For more information, see “QP Solver”.

The KWIK algorithm defines inequality constraints as $Ax \geq b$ rather than $Ax \leq b$, as in the `quadprog` command.

- “QP Solver”

References

- [1] Schmid, C., and L. T. Biegler. “Quadratic programming methods for reduced Hessian SQP.” *Computers & Chemical Engineering*. Vol. 18, No. 9, 1994, pp. 817–832.

See Also

`mpcqpsolverOptions` | `quadprog`

Introduced in R2015b

mpcqpsolverOptions

Create default option set for `mpcqpsolver`

Syntax

```
options = mpqpsoverOptions  
options = mpqpsoverOptions(type)
```

Description

`options = mpqpsoverOptions` creates a structure of default options for `mpqpsover`, which solves a quadratic programming (QP) problem using the KWIK algorithm.

`options = mpqpsoverOptions(type)` creates a default option set using the specified input data type. All real options are specified using this data type.

Examples

Create Default Option Set for MPC QP Solver

```
opt = mpqpsoverOptions;
```

Create and Modify Default MPC QP Solver Option Set

Create default option set.

```
opt = mpqpsoverOptions;
```

Specify the maximum number of iterations allowed during computation.

```
opt.MaxIter = 100;
```

Specify a feasibility tolerance for verifying that the optimal solution satisfies the inequality constraints.

```
opt.FeasibilityTol = 1.0e-3;
```

Create Option Set Specifying Input Argument Type

```
opt = mpcqpsolverOptions( single );
```

Input Arguments

type — MPC QP solver input argument data type

`double` (default) | `single`

MPC QP solver input argument data type, specified as either `double` or `single`. This data type is used for both simulation and code generation. All real options in the option set are specified using this data type, and all real input arguments to `mpcqpsolver` must match this type.

Output Arguments

options — Option set for `mpcqpsolver`

structure

Option set for `mpcqpsolver`, returned as a structure with the following fields:

Field	Description	Default
<code>DataType</code>	Input argument data type, specified as either <code>double</code> or <code>single</code> . This data type is used for both simulation and code generation, and all real input arguments to <code>mpcqpsolver</code> must match this type.	<code>double</code>
<code>MaxIter</code>	Maximum number of iterations allowed when computing the QP solution, specified as a positive integer.	200
<code>Feasibil</code>	Tolerance used to verify that inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger <code>FeasibilityTol</code> value allows for larger constraint violations.	<code>1.0e-6</code>
<code>Integrit</code>	Indicator of whether integrity checks are performed on the <code>mpcqpsolver</code> input data, specified as a logical value. If <code>IntegrityChecks</code> is <code>true</code> , then integrity checks are performed and diagnostic messages are displayed. Use <code>false</code> for code generation only.	<code>true</code>

See Also

[mpcqpsolver](#)

Introduced in R2015b

mpcsimopt

MPC simulation options

Syntax

```
options = mpcsimopt(MPCobj)
```

Description

`options = mpcsimopt(MPCobj)` creates an set of options for specifying additional parameters for simulating an `mpc` controller, `MPCobj`, with `sim`. Initially, `options` is empty. Use dot notation to change the options as needed for the simulation.

Output Arguments

options

Options for simulating an `mpc` controller using `sim`. `options` has the following properties.

MPC Simulation Options Properties

Property	Description
PlantInitialState	Initial state vector of the plant model generating the data.
ControllerInitialState	Initial condition of the MPC controller. This must be a valid <code>mpcstate</code> object. Note Nonzero values of <code>ControllerInitialState.LastMove</code> are only meaningful if there are constraints on the increments of the manipulated variables.
UnmeasuredDisturbance	Unmeasured disturbance signal entering the plant.

Property	Description
	An array with as many rows as simulation steps, and as many columns as unmeasured disturbances. Default: 0
InputNoise	<p>Noise on manipulated variables.</p> <p>An array with as many rows as simulation steps, and as many columns as manipulated variables. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0</p>
OutputNoise	<p>Noise on measured outputs.</p> <p>An array with as many rows as simulation steps, and as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0</p>
RefLookAhead	<p>Preview on reference signal (<code>on</code> or <code>off</code>). Default: <code>off</code></p>
MDLookAhead	<p>Preview on measured disturbance signal (<code>on</code> or <code>off</code>).</p>
Constraints	<p>Use MPC constraints (<code>on</code> or <code>off</code>). Default: <code>on</code></p>
Model	<p>Model used in simulation for generating the data.</p> <p>This property is useful for simulating the MPC controller under model mismatch. The LTI object specified in <code>Model</code> can be either a replacement for <code>Model.Plant</code>, or a structure with fields <code>Plant</code> and <code>Nominal</code>. By default, <code>Model</code> is equal to <code>MPCObj.Model</code> (no model mismatch). If <code>Model</code> is specified, then <code>PlantInitialState</code> refers to the initial state of <code>Model.Plant</code> and is defaulted to <code>Model.Nominal.x</code>.</p> <p>If <code>Model.Nominal</code> is empty, <code>Model.Nominal.U</code> and <code>Model.Nominal.Y</code> are inherited from <code>MPCObj.Model.Nominal</code>. <code>Model.Nominal.X</code>/</p>

Property	Description
	DX is only inherited if both plants are state-space objects with the same state dimension.
StatusBar	Display the wait bar (<code>on</code> or <code>off</code>). Default: <code>off</code>
MVSignal	Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action). An array with as many rows as simulation steps, and as many columns as manipulated variables. Default: 0
OpenLoop	Perform open-loop simulation (<code>on</code> or <code>off</code>). Default: <code>off</code>

Examples

Simulate MPC Control with Plant Model Mismatch

Simulate the MPC control of a multi-input, multi-output (MIMO) system with a mismatch between the predicted and actual plant models. The system has two manipulated variables, two unmeasured disturbances, and two measured outputs.

Define the predicted plant model.

```
p1 = tf(1,[1 2 1])*[1 1;0 1];
plantPredict = ss([p1 p1]);
plantPredict.InputName = { mv1 , mv2 , umd3 , umd4 };
```

Specify the MPC signal types.

```
plantPredict = setmpcsignals(plantPredict, MV ,[1 2], UD ,[3 4]);
```

Create the MPC controller.

```
mpcobj = mpc(plantPredict,1,40,2);
```

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define the unmeasured input disturbance model used by the controller.

```
distModel = eye(2,2)*ss(-0.5,1,1,0);
mpcobj.Model.Disturbance = distModel;
```

Define an actual plant model which differs from the predicted model and has unforeseen unmeasured disturbance inputs.

```
p2 = tf(1.5,[0.1 1 2 1])*[1 1;0 1];
plantActual = ss([p2 p2 tf(1,[1 1])*[0;1]]));
plantActual = setmpcsignals(plantActual, MV ,[1 2], UD ,[3 4 5]);
```

Configure the unmeasured disturbance and output reference trajectories.

```
dist = ones(1,3);
refs = [1 2];
```

Create and configure a simulation option set.

```
options = mpcsimopt(mpcobj);
options.UnmeasuredDisturbance = dist;
options.Model = plantActual;
```

Simulate the system.

```
sim(mpcobj,20,refs,options)
```

```
-->Converting model to discrete time.
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
```

```
-->Assuming output disturbance added to measured output channel #2 is integrated white
```

```
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output.
```

```
-->Converting model to discrete time.
```

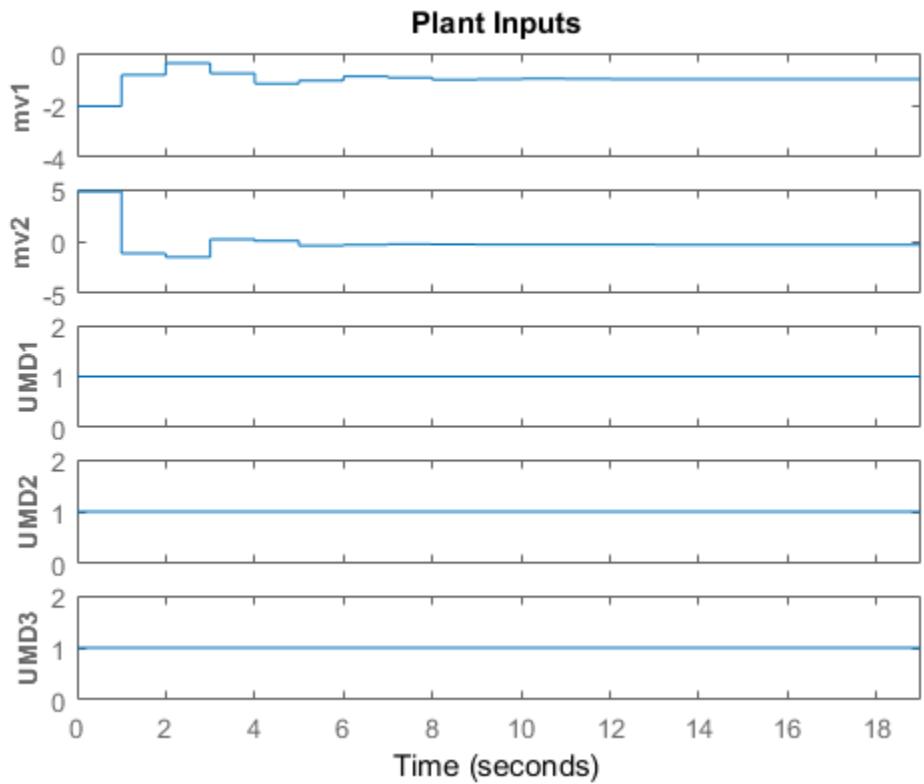
```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 2.
```

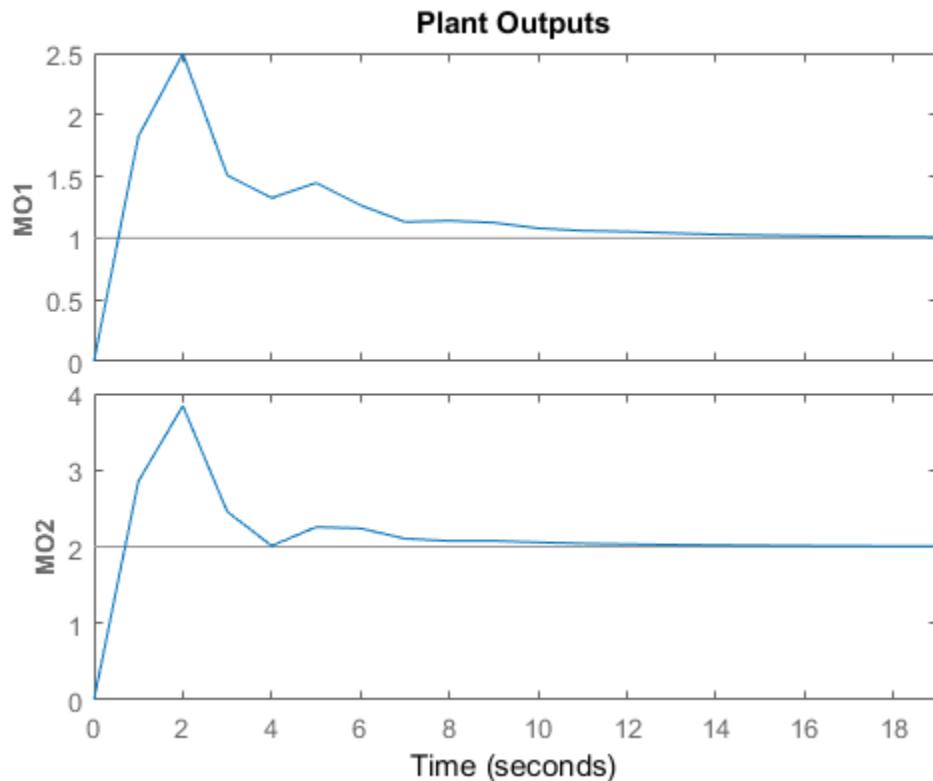
```
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 1.
```

```
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.
```

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.
```





- “Simulate Controller with Nonlinear Plant”

See Also

`sim`

Introduced before R2006a

mpcstate

Define MPC controller state

Syntax

```
xmpc = mpcstate(MPCObj)
xmpc = mpcstate(MPCObj, xp, xd, xn, u, p)
xmpc = mpcstate
```

Description

`xmpc = mpcstate(MPCObj)` creates a controller state object compatible with the controller object, `MPCObj`, in which all fields are set to their default values that are associated with the controller's nominal operating point.

`xmpc = mpcstate(MPCObj, xp, xd, xn, u, p)` sets the state fields of the controller state object to specified values. The controller may be an implicit or explicit controller object. Use this controller state object to initialize an MPC controller at a specific state other than the default state.

`xmpc = mpcstate` returns an `mpcstate` object in which all fields are empty.

`mpcstate` objects are updated by `mpcmove` through the internal state observer based on the extended prediction model. The overall state is updated from the measured output $y_m(k)$ by a linear state observer (see “State Observer”).

Input Arguments

MPCobj

MPC controller, specified as either a traditional MPC controller (`mpc`) or explicit MPC controller (`generateExplicitMPC`).

xp

Plant model state estimates, specified as a vector with N_{xp} elements, where N_{xp} is the number of states in the plant model.

xd

Disturbance model state estimates, specified as a vector with N_{xd} elements, where N_{xd} is the total number of states in the input and output disturbance models. The disturbance model states are ordered such that input disturbance model states are followed by output disturbance model state estimates.

xn

Measurement noise model state estimates, specified as a vector with N_{xn} elements, where N_{xn} is the number of states in the measurement noise model.

u

Values of the manipulated variables during the previous control interval, specified as a vector with N_u elements, where N_u is the number of manipulated variables.

p

Covariance matrix for the state estimates, specified as an N -by- N matrix, where N is the sum of N_{xp} , N_{xd} and N_{xn} .

Output Arguments

xmpc

MPC state object, containing the following properties.

Property	Description
Plant	<p>Vector of state estimates for the controller's plant model. Values are in engineering units and are absolute, i.e., they include state offsets.</p> <p>If the controller's plant model includes delays, the <code>Plant</code> field of the MPC state object includes states that model the delays. Therefore <code>length(Plant) > order of undelayed controller plant model.</code></p> <p>Default: controller's <code>Model.Nominal.X</code> property.</p>

Property	Description
Disturbance	<p>Vector of unmeasured disturbance model state estimates. This comprises the states of the input disturbance model followed by the states of the output disturbances model.</p> <p>Disturbance models may be created by default. Use the <code>getindist</code> and <code>getoutdist</code> commands to view the two disturbance model structures.</p> <p>Default: zero, or empty if there are no disturbance model states.</p>
Noise	<p>Vector of output measurement noise model state estimates.</p> <p>Default: zero, or empty if there are no noise model states.</p>
LastMove	<p>Vector of manipulated variables used in the previous control interval, $u(k-1)$. Values are absolute, i.e., they include manipulated variable offsets.</p> <p>Default: nominal values of the manipulated variables.</p>
Covariance	<p>n-by-n symmetrical covariance matrix for the controller state estimates, where n is the dimension of the extended controller state, i.e., the sum of the number states contained in the <code>Plant</code>, <code>Disturbance</code>, and <code>Noise</code> fields.</p> <p>Default: If the controller is employing default state estimation the default is the steady-state covariance computed according to the assumptions in “Controller State Estimation”. See also the description of the <code>P</code> matrix in the Control System Toolbox <code>kalmd</code> command. If the controller is employing custom state estimation, this field is empty (not used).</p>

Examples

Get Controller State Object

Create a Model Predictive Controller for a single-input-single-output (SISO) plant. For this example, the plant includes an input delay of 0.4 time units, and the control interval to 0.2 time units.

```

H = tf(1,[10 1], InputDelay ,0.4);
MPCobj = mpc(H,0.2);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1

```

Create the corresponding controller state object in which all states are at their default values.

```
xMPC = mpcstate(MPCobj)
```

```

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Converting delays to states.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
MPCSTATE object with fields
    Plant: [0 0 0]
    Disturbance: 0
    Noise: [1x0 double]
    LastMove: 0
    Covariance: [4x4 double]

```

The plant model, H, is a first-order, continuous-time transfer function. The **Plant** property of the **mpcstate** object contains two additional states to model the two intervals of delay. Also, by default the controller contains a first-order output disturbance model (an integrator) and an empty measured output noise model.

View the default covariance matrix.

```
xMPC.Covariance
```

```
ans =
```

0.0624	0.0000	0.0000	-0.0224
0.0000	1.0000	0.0000	-0.0000
0.0000	0.0000	1.0000	-0.0000
-0.0224	-0.0000	-0.0000	0.2301

See Also

`getoutdist` | `setindist` | `setoutdist` | `getEstimator` | `setEstimator` | `ss` |
`mpcmove`

Introduced before R2006a

mpcverbosity

Change toolbox verbosity level

Syntax

```
mpcverbosity on  
mpcverbosity off  
old_status = mpcverbosity(new_status)  
mpcverbosity
```

Description

`mpcverbosity on` enables messages displaying default operations taken by Model Predictive Control Toolbox software during the creation and manipulation of model predictive control objects.

`mpcverbosity off` turns messages off.

`old_status = mpcverbosity(new_status)` sets the verbosity level to the specified value, `new_status`. The function returns the original value of the verbosity level as `old_status`. Specify `new_status` as a string with the value of either `on` or `off`.

`mpcverbosity` just shows the verbosity status.

By default, messages are turned on.

See also “Construction and Initialization” on page 3-12 .

See Also

`mpc`

Introduced before R2006a

plot

Plot responses generated by MPC simulations

Syntax

```
plot(MPCObj,t,y,r,u,v,d)
```

Description

`plot(MPCObj,t,y,r,u,v,d)` plots the results of a simulation based on the MPC object `MPCObj`. `t` is a vector of length `Nt` of time values, `y` is a matrix of output responses of size `[Nt,Ny]` where `Ny` is the number of outputs, `r` is a matrix of setpoints and has the same size as `y`, `u` is a matrix of manipulated variable inputs of size `[Nt,Nu]` where `Nu` is the number of manipulated variables, `v` is a matrix of measured disturbance inputs of size `[Nt,Nv]` where `Nv` is the number of measured disturbance inputs, and `d` is a matrix of unmeasured disturbance inputs of size `[Nt,Nd]` where `Nd` is the number of unmeasured disturbances input.

See Also

`sim` | `mpc`

Introduced before R2006a

plotSection

Visualize explicit MPC control law as 2-D sectional plot

Syntax

```
plotsection(EMPCobj,plotParams)
```

Description

`plotsection(EMPCobj,plotParams)` displays a 2-D sectional plot of the piecewise affine regions used by an explicit MPC controller. All but two of the control law's free parameters are fixed, as specified by `plotParams`. The two remaining variables form the plot axes. By default, the `EMPCobj.Range` property sets the bounds for these axes.

Examples

Specify Fixed Parameters for 2-D Plot of Explicit Control Law

Define a double integrator plant model and create a traditional implicit MPC controller for this plant. Constrain the manipulated variable to have an absolute value less than 1.

```
plant = tf(1,[1 0 0]);
MPCobj = mpc(plant,0.1,10,3);
MPCobj.MV = struct( Min ,-1, Max ,1);

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define the parameter bounds for generating an explicit MPC controller.

```
range = generateExplicitRange(MPCobj);
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min(:) = -2;
range.Reference.Max(:) = 2;
range.ManipulatedVariable.Min(:) = -1.1;
range.ManipulatedVariable.Max(:) = 1.1;
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.  
-->Converting model to discrete time.  
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel.
```

Create an explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range);
```

```
Regions found / unexplored: 19/ 0
```

Create a default plot parameter structure, which specifies that all of the controller parameters are fixed at their nominal values for plotting.

```
plotParams = generatePlotParameters(EMPCobj);
```

Allow the controller states to vary when creating a plot.

```
plotParams.State.Index = [];  
plotParams.State.Value = [];
```

Fix the manipulated variable and reference signal to 0 for plotting.

```
plotParams.ManipulatedVariable.Index(1) = 1;  
plotParams.ManipulatedVariable.Value(1) = 0;  
plotParams.Reference.Index(1) = 1;  
plotParams.Reference.Value(1) = 0;
```

Generate the 2-D section plot for the explicit MPC controller.

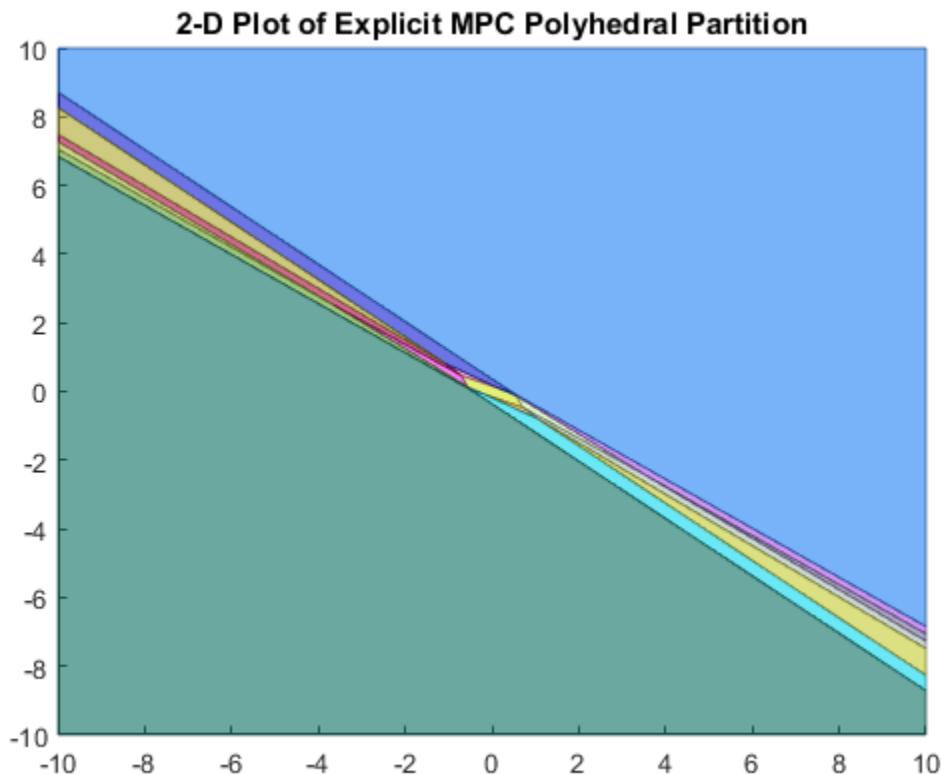
```
plotSection(EMPCobj,plotParams)
```

```
ans =
```

Figure (1: PiecewiseAffineSectionPlot) with properties:

```
Number: 1  
Name: PiecewiseAffineSectionPlot  
Color: [0.9400 0.9400 0.9400]  
Position: [360 502 560 420]  
Units: pixels
```

Use GET to show all properties



Input Arguments

EMPCobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller for which you want to create a 2-D sectional plot, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

plotParams — Parameters for sectional plot

structure

Parameters for sectional plot of explicit MPC control law, specified as a structure. Use `generatePlotParameters` to create an initial structure in which all the parameters of the controller are fixed at their nominal values. Then, modify this structure as necessary before invoking `plotSection`. See `generatePlotParameters` for more information.

See Also

`generateExplicitMPC` | `generatePlotParameters`

Introduced in R2014b

review

Examine MPC controller for design errors and stability problems at run time

Syntax

```
review(mpcobj)
```

Description

`review(mpcobj)` checks for potential design issues in the model predictive controller, `mpcobj`, and generates a report. `review` performs the following diagnostic tests:

- Is the optimal control problem well defined?
- Is the controller internally stable?
- Is the closed loop system stable when no constraints are active and there is no model mismatch?
- Is the controller able to eliminate steady-state tracking error when no constraints are active?
- Is there a likelihood that constraint definitions will result in an ill-conditioned or infeasible optimization problem?
- If the controller were used in a real-time environment, what memory capacity would be needed?

Use `review` iteratively to check your initial MPC design or whenever you make substantial changes to your controller. Make the recommended changes to your controller to eliminate potential problems. `review` does not modify `mpcobj`.

Input Arguments

mpcobj

Non-empty Model Predictive Controller (`mpc`) object

Examples

Examine MPC Controller for Design Errors or Stability Problems

Define a plant model and create an MPC controller.

```
plant = tf(1, [10 1]);
Ts = 2;
MPCobj = mpc(plant,Ts);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Set hard upper and lower bounds on the manipulated variable and its rate-of-change.

```
MV = MPCobj.MV;
MV.Min = -2;
MV.Max = 2;
MV.RateMin = -4;
MV.RateMax = 4;
MPCobj.MV = MV;
```

Review the controller design.

```
review(MPCobj)

-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Design Review for Model Predictive Controller "MPCObj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Pass
Hard MV Constraints	Warning
Other Hard Constraints	Pass
Soft Constraints	Pass
Memory Size for MPC Data	Pass

Individual Test Result

MPC Object Creation

Tests whether your specifications generate a valid MPC object. If not, the review terminates.

The MPC object is OK. Testing can proceed.

[Return to list of tests](#)

review flags the potential constraint conflict that could result if this controller was used to control a real process.

Web Browser - Review MPC Object "MPCObj"

Review MPC Object "MPCObj" +

Hard MV Constraints

The controller should always satisfy hard bounds on a manipulated variable *OR* its rate-of-change. If you specify both constraint types simultaneously, however, they might conflict during real-time use.

For example, if an event pushes an MV outside a specified hard bound and the hard MV rate bounds are too small, the resulting QP will be *infeasible*.

Avoid such conflicts by specifying hard MV bounds *OR* hard MV rate bounds, but not both. Or if you want to specify both, soften the lower-priority constraint by setting its ECR to a value greater than zero.

Warning: your constraint definitions may conflict. The following table lists potential conflicts for each MV. The tabular entries show the location of each conflict in the prediction horizon and the type of conflict.

MV name	Horizon k	Conflict Type
MV1	1	Min & RateMax
MV1	1	Max & RateMin

[Return to list of tests](#)

Other Hard Constraints

It can be impossible for your controller to satisfy all its hard constraints under all conditions

- Reviewing Model Predictive Controller Design for Potential Stability and Robustness Issues

Alternatives

`review` automates certain tests that you can perform at the command line.

- To test for steady-state tracking errors, use `cloffset`.
- To test the internal stability of a controller, check the eigenvalues of the `mpc` object. Use `ss` to convert the `mpc` object to a state-space model, and call `isstable`.

More About

Tips

- You can review your controller design in the MPC Designer app. On the **Tuning** tab, in the **Analysis** section, click **Review Design**.
- Test your controller design using techniques such as simulations, since `review` cannot detect all possible performance factors.
- “Simulation and Code Generation Using Simulink Coder”

See Also

`cloffset` | `mpc` | `ss`

Introduced in R2011b

sensitivity

Compute effect of controller tuning weights on performance

Syntax

```
[J,sens] =
sensitivity(MPCobj,PerfFunc,PerfWeights,Tstop,r,v,simopt,utarget)
[J,sens] = sensitivity(MPCobj, perf_fun ,param1,param2,...)
```

Description

The **sensitivity** function is a controller tuning aid. **J** specifies a scalar performance metric. **sensitivity** computes **J** and its partial derivatives with respect to the controller tuning weights. These *sensitivities* suggest tuning weight adjustments that should improve performance, that is, reduce **J**.

[J,sens] =
sensitivity(MPCobj,PerfFunc,PerfWeights,Tstop,r,v,simopt,utarget)
calculates the scalar performance metric, **J**, and sensitivities, **sens**, for the controller defined by the MPC controller object **MPCobj**.

PerfFunc must be one of the following strings:

ISE (integral squared error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uij})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right)$$

IAE (integral absolute error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} \left(\sum_{j=1}^{n_y} |w_j^y e_{yij}| + \sum_{j=1}^{n_u} (|w_j^u e_{uij}| + |w_j^{\Delta u} \Delta u_{ij}|) \right)$$

ITSE (integral of time-weighted squared error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} i\Delta t \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uji})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right)$$

$$J = \sum_{i=1}^{Tstop} i\Delta t \left(\sum_{j=1}^{n_y} |w_j^y e_{yij}| + \sum_{j=1}^{n_u} (|w_j^u e_{uji}| + |w_j^{\Delta u} \Delta u_{ij}|) \right)$$

ITAE (integral of time-weighted absolute error) for which the performance metric is

In the above expressions n_y is the number of controlled outputs and n_u is the number of manipulated variables. e_{yij} is the difference between output j and its setpoint (or reference) value at time interval i . e_{uji} is the difference between manipulated variable j and its target at time interval i .

The w parameters are nonnegative performance weights defined by the structure **PerfWeights**, which contains the following fields:

- **OutputVariables** — n_y element row vector that contains the w_j^y values
- **ManipulatedVariables** — n_u element row vector that contains the w_j^u values
- **ManipulatedVariablesRate** — n_u element row vector that contains the $w_j^{\Delta u}$ values

If **PerfWeights** is unspecified, it defaults to the corresponding weights in **MPCobj**. In general, however, the performance weights and those used in the controller have different purposes and should be defined accordingly.

Inputs **Tstop**, **r**, **v**, and **simopt** define the simulation scenario used to evaluate performance. See **sim** for details.

Tstop is the integer number of controller sampling intervals to be simulated. The final time for the simulations will be $Tstop \times \Delta t$, where Δt is the controller sampling interval specified in **MPCobj**.

The optional input **utarget** is a vector of n_u manipulated variable targets. Their defaults are the nominal values of the manipulated variables. Δu_{ij} is the change in manipulated variable j and its target at time interval i .

The structure variable **sens** contains the computed sensitivities (partial derivatives of **J** with respect to the **MPCobj** tuning weights.) Its fields are:

- **OutputVariables** — n_y element row vector of sensitivities with respect to **MPCobj.Weights.OutputVariables**
- **ManipulatedVariables** — n_u element row vector of sensitivities with respect to **MPCobj.Weights.ManipulatedVariables**
- **ManipulatedVariablesRate** — n_u element row vector of sensitivities with respect to **MPCobj.Weights.ManipulatedVariablesRate**

See “Weights” on page 1-65 for details on the tuning weights contained in **MPCobj**.

`[J,sens] = sensitivity(MPCobj, perf_fun ,param1,param2,...)` employs a performance function `perf_fun` to define **J**. Its function definition must be in the form

```
function J = perf_fun(MPCobj, param1, param2, ...)
```

That is, it must compute **J** for the given controller and optional parameters `param1`, `param2`, ... and it must be on the MATLAB path.

Note: While performing the sensitivity analysis, the software ignores time-varying, nondiagonal, and ECR slack variable weights.

Examples

Compute Controller Performance and Sensitivity

Define a third-order plant model with three manipulated variables and two controlled outputs.

```
plant = rss(3,2,3);
plant.d = 0;
```

Create an MPC controller for the plant.

```
MPCobj = mpc(plant,1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
```

```
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default value 0
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default value 1
```

Specify an integral absolute error performance function and set the performance weights.

```
PerfFunc = IAE ;
PerfWts.OutputVariables = [1 0.5];
PerfWts.ManipulatedVariables = zeros(1,3);
PerfWts.ManipulatedVariablesRate = zeros(1,3);
```

Define a 20 second simulation scenario with a unit step in the output 1 setpoint and a setpoint of zero for output 2.

```
Tstop = 20;
r = [1 0];
```

Define the nominal values of the manipulated variables to be zeros.

```
utarget = zeros(1,3);
```

Calculate the performance metric, J, and sensitivities, sens, for the specified controller and simulation scenario.

```
[J, sens] = sensitivity(MPCobj,PerfFunc,PerfWts,Tstop,r,[],[],utarget);

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white noise
-->Assuming output disturbance added to measured output channel #2 is integrated white noise
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output channel
```

See Also

[mpc](#) | [sim](#)

Introduced in R2009a

set

Set or modify MPC object properties

Syntax

```
set(MPCobj, Property ,Value)
set(MPCobj, Property1 ,Value1, Property2 ,Value2,...)
set(MPCobj, Property )
set(sys)
```

Description

The `set` function is used to set or modify the properties of an MPC controller (see “MPC Controller Object” on page 3-2 for background on MPC properties). Like its Handle Graphics® counterpart, `set` uses property name/property value pairs to update property values.

`set(MPCobj, Property ,Value)` assigns the value `Value` to the property of the MPC controller `MPCobj` specified by the string `Property`. This string can be the full property name (for example, `UserData`) or any unambiguous case-insensitive abbreviation (for example, `user`).

`set(MPCobj, Property1 ,Value1, Property2 ,Value2,...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`set(MPCobj, Property)` displays admissible values for the property specified by `Property`. See “MPC Controller Object” on page 3-2 for an overview of legitimate MPC property values.

`set(sys)` displays all assignable properties of `sys` and their admissible values.

See Also

`mpc` | `get` | `mpcprops`

Introduced before R2006a

setconstraint

Set custom constraints on linear combinations of plant inputs and outputs

Syntax

```
setconstraint(MPCobj,E,F,G)
setconstraint(MPCobj,E,F,G,V)
setconstraint(MPCobj,E,F,G,V,S)

setconstraint(MPCobj)
```

Description

`setconstraint(MPCobj,E,F,G)` specifies custom constraints of the following form for the MPC controller, `MPCobj`:

$$Eu(k+j|k) + Fy(k+j|k) \leq G + \varepsilon$$

where $j = 0, \dots, p$, and:

- p is the prediction horizon.
- k is the current time index.
- E , F , and G are constant matrices. Each row of E , F , and G represents a linear constraint to be imposed at each prediction horizon step.
- u is a column vector of manipulated variables.
- y is a column vector of all plant output variables.
- ε is a slack variable used for constraint softening (as in “Standard Cost Function”).

`setconstraint(MPCobj,E,F,G,V)` adds constraints of the following form:

$$Eu(k+j|k) + Fy(k+j|k) \leq G + \varepsilon V$$

where V is a constant vector representing the equal concern for the relaxation (ECR).

`setconstraint(MPCobj,E,F,G,V,S)` adds constraints of the following form:

$$Eu(k+j|k) + Fy(k+j|k) + Sv(k+j|k) \leq G + \varepsilon V$$

where:

- v is a column vector of measured disturbance variables.
- S is a constant matrix.

`setconstraint(MPCobj)` removes all custom constraints from the MPC controller, `MPCobj`.

Examples

Specify Custom Constraints on Linear Combination of Inputs and Outputs

Specify a constraint of the form $0 \leq u_2 - 2u_3 + y_2 \leq 15$ on an MPC controller.

Create a third-order plant model with three manipulated variables and two measured outputs.

```
plant = rss(3,2,3);
plant.d = 0;
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Formulate the constraint in the required form:

$$\begin{bmatrix} 0 & -1 & 2 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 15 \end{bmatrix} + \varepsilon \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Specify the constraint matrices.

```
E = [0 -1 2;0 1 -2];
F = [0 -1;0 1];
G = [0;15];
```

Set the constraints in the MPC controller.

```
setconstraint(MPCobj,E,F,G)
```

Specify Custom Hard Constraints for MPC Controller

Create a third-order plant model with two manipulated variables and two measured outputs.

```
plant = rss(3,2,2);  
plant.d = 0;
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Assume that you have two hard constraints.

$$\begin{aligned} u_1 + u_2 &\leq 5 \\ y_1 + y_2 &\leq 10 \end{aligned}$$

Specify the constraint matrices.

```
E = [1 1; 0 0];  
F = [0 0; 1 1];  
G = [5;10];
```

Specify the constraints as hard by setting V to zero for both constraints.

```
V = [0;0];
```

Set the constraints in the MPC controller.

```
setconstraint(MPCobj,E,F,G,V)
```

Specify Custom Constraints for MPC Controller with Measured Disturbances

Create a third-order plant model with two manipulated variables, two measured disturbances, and two measured outputs.

```
plant = rss(3,2,4);
plant.D = 0;
plant = setmpcsignals(plant, mv ,[1 2], md ,[3 4]);
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Assume that you have three soft constraints.

$$\begin{aligned} u_1 + u_2 &\leq 5 \\ y_1 + v_1 &\leq 10 \\ y_2 + v_2 &\leq 12 \end{aligned}$$

Specify the constraint matrices.

```
E = [1 1; 0 0; 0 0];
F = [0 0; 1 0; 0 1];
G = [5;10;12];
S = [0 0; 1 0; 0 1];
```

Set the constraints in the MPC controller using the default value for V.

```
setconstraint(MPCobj,E,F,G,[],S)
```

Remove All Custom Constraints from MPC Controller

Define a plant model and create an MPC controller.

```
plant = rss(3,2,2);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
```

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define controller custom constraints.

```
E = [-1 2; 1 -2];  
F = [0 1; 0 -1];  
G = [0; 10];  
setconstraint(MPCobj,E,F,G)
```

Remove the custom constraints.

```
setconstraint(MPCobj)  
-->Removing mixed input/output constraints.
```

- MPC Control with Constraints on a Combination of Input and Output Signals
- MPC Control of a Nonlinear Blending Process

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

E — Manipulated variable constraint constant

matrix of zeros (default) | matrix

Manipulated variable constraint constant, specified as a matrix with:

- n_c rows, where n_c is the number of constraints.
- n_u columns, where n_u is the number of manipulated variables.

F — Controlled output constraint constant

matrix of zeros (default) | matrix

Controlled output constraint constant, specified as a matrix with:

- n_c rows, where n_c is the number of constraints.

- n_y columns, where n_y is the number of controlled outputs (measured and unmeasured).

G — Custom constraint constant

column vector of zeros (default) | column vector

Custom constraint constant, specified as a column vector with n_c elements, where n_c is the number of constraints.

V — Constraint softening constant

column vector of ones (default) | column vector

Constraint softening constant representing the equal concern for the relaxation (ECR), specified as a column vector with n_c elements, where n_c is the number of constraints.

If V is not specified, a default value of 1 is applied to all constraint inequalities and all constraints are soft. This behavior is the same as the default behavior for output bounds, as described in “Standard Cost Function”.

To make the i^{th} constraint hard, specify $V(i) = 0$.

To make the i^{th} constraint soft, specify $V(i) > 0$ in keeping with the constraint violation magnitude you can tolerate. The magnitude violation depends on the numerical scale of the variables involved in the constraint.

In general, as $V(i)$ decreases, the controller hardens the constraints by decreasing the constraint violation that is allowed.

Note: If a constraint is difficult to satisfy, reducing its $V(i)$ value to make it harder may be counterproductive, and can lead to erratic control actions, instability, or failure of the QP solver that determines the control action.

S — Measured disturbance constraint constant

matrix of zeros (default) | matrix

Measured disturbance constraint constant, specified as a matrix with:

- n_c rows, where n_c is the number of constraints.
- n_v columns, where n_v is the number of measured disturbances.

More About

Tips

- The outputs, y , are being predicted using a model. If the model is imperfect, there is no guarantee that a constraint can be satisfied.
- Since the MPC controller does not optimize $u(k+p|k)$, the last constraint at time $k+p$ assumes that $u(k+p|k) = u(k+p-1|k)$.
- When simulating an MPC controller, you can update the custom constraints at each iteration by calling `setconstraint` before calling `mpcmove`.

To deploy an MPC controller with run-time updating of custom constraints, use MATLAB Compiler™ to generate the executable code, and deploy it using the MATLAB Runtime. In this case, the controller sample time must be large, since run-time MPC regeneration is slow.

Note: Updating the custom constraint matrices at each simulation iteration is not supported in Simulink.

- “Constraints on Linear Combinations of Inputs and Outputs”
- “Run-Time Constraint Updating”

See Also

`getconstraint` | `setterminal`

Introduced in R2011a

setEstimator

Modify a model predictive controller's state estimator

Syntax

```
setEstimator(MPCobj,L,M)
setEstimator(MPCobj, default )
setEstimator(MPCobj, custom )
```

Description

`setEstimator(MPCobj,L,M)` sets the gain matrices used for estimation of the states of an MPC controller. See “State Estimator Equations” on page 1-44. If `L` is empty, it defaults to $L = A^*M$, where A is the state transition matrix defined in “State Estimator Equations” on page 1-44. If `M` is omitted or empty, it defaults to a zero matrix, and the state estimator becomes a Luenberger observer.

`setEstimator(MPCobj, default)` restores the gain matrices `L` and `M` to their default values. The default values are the optimal static gains calculated by the Control System Toolbox function `kalmd` for the plant, disturbance, and measurement noise models specified in `MPCobj`.

`setEstimator(MPCobj, custom)` specifies that controller state estimation will be performed by a user-supplied procedure rather than the equations described in “State Estimator Equations” on page 1-44. This option suppresses calculation of `L` and `M`. When the controller is operating in this way, the procedure must supply the state estimate $x[n|n]$ to the controller at the beginning of each control interval.

Examples

Design State Estimator by Pole Placement

Design an estimator using pole placement, assuming the linear system $AM = L$ is solvable.

Create a plant model.

```
G = tf({1,1,1},{{1 .5 1],[1 1],[.7 .5 1]});
```

To improve the clarity of this example, call `mpcverbosity` to suppress messages related to working with an MPC controller.

```
old_status = mpcverbosity( off );
```

Create a model predictive controller for the plant. Specify the controller sample time as 0.2 seconds.

```
MPCobj = mpc(G, 0.2);
```

Obtain the default state estimator gain.

```
[~,M,A1,Cm1] = getEstimator(MPCobj);
```

Calculate the default observer poles.

```
e = eig(A1-A1*M*Cm1);
abs(e)
```

```
ans =
```

```
0.9402
0.9402
0.8816
0.8816
0.7430
0.9020
```

Specify faster observer poles.

```
new_poles = [.8 .75 .7 .85 .6 .81];
```

Compute a state-gain matrix that places the observer poles at `new_poles`.

```
L = place(A1 ,Cm1 ,new_poles) ;
```

`place` returns the controller-gain matrix, whereas you want to compute the observer-gain matrix. Using the principle of duality, which relates controllability to observability, you specify the transpose of `A1` and `Cm1` as the inputs to `place`. This function call yields the observer gain transpose.

Obtain the estimator gain from the state-gain matrix.

```
M = A1\L;
```

Specify M as the estimator for MPCobj.

```
setEstimator(MPCobj,L,M)
```

The pair, (A_1, C_{m1}), describing the overall state-space realization of the combination of plant and disturbance models must be observable for the state estimation design to succeed. Observability is checked in Model Predictive Control Toolbox software at two levels: (1) observability of the plant model is checked *at construction* of the MPC object, provided that the model of the plant is given in state-space form; (2) observability of the overall extended model is checked *at initialization* of the MPC object, after all models have been converted to discrete-time, delay-free, state-space form and combined together.

Restore mpcverbosity.

```
mpcverbosity(old_status);
```

Input Arguments

MPCobj — MPC controller

MPC controller object

MPC controller, specified as an MPC controller object. Use the `mpc` command to create the MPC controller.

L — Kalman gain matrix for time update

A^*M (default) | matrix

Kalman gain matrix for the time update, specified as a matrix. The dimensions of L are n_x -by- n_{ym} , where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-44.

If L is empty, it defaults to $L = A^*M$, where A is the state transition matrix defined in “State Estimator Equations” on page 1-44.

M — Kalman gain matrix for measurement update

0 (default) | matrix

Kalman gain matrix for the measurement update, specified as a matrix. The dimensions of L are n_x -by- n_{ym} , where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-44.

If M is omitted or empty, it defaults to a zero matrix, and the state estimator becomes a Luenberger observer.

More About

State Estimator Equations

The following equations describe the state estimation. For more details, see “Controller State Estimation”.

Output estimate: $y_m[n | n-1] = C_m x[n | n-1] + D_{vm} v[n]$.

Measurement update: $x[n | n] = x[n | n-1] + M (y_m[n] - y_m[n | n-1])$.

Time update: $x[n+1 | n] = A x[n | n-1] + B_u u[n] + B_v v[n] + L (y_m[n] - y_m[n | n-1])$.

Estimator state: $x[n+1 | n] = (A - L C_m) x[n | n-1] + B_u u[n] + (B_v - L D_{vm}) v[n] + L y_m[n]$.

The estimator state is based on the current measurement of $y_m[n]$ and $v[n]$ as well as the optimal control action $u[n]$ computed at the current control interval.

The variables in these equations are summarized in the following table.

Symbol	Description
x	<p>Controller state vector, length n_x. It includes (in this sequence):</p> <ul style="list-style-type: none"> Plant model state estimates. Dimension obtained by conversion of <code>MPCobj.Model.Plant</code> to discrete LTI state-space form (if necessary), followed by use of <code>absorbDelay</code> to convert any delays to additional states. Input disturbance model state estimates (if any). Use the <code>getindist</code> command to review the input disturbance model structure. Output disturbance model state estimates (if any). Use the <code>getoutdist</code> command to review the output disturbance model structure. Output measurement noise states (if any) as specified by <code>MPCobj.Model.Noise</code>. <p>The length n_x is the sum of the number of states in the above four categories.</p>

Symbol	Description
y_m	Vector of measured outputs or an estimate of their true values, length n_{ym} .
u	Vector of manipulated variables, length n_u .
v	Vector of measured input disturbances, length n_v .
$[j k]$	Denotes an estimate of a state or output at time t_j based on data available at time t_k .
$[k]$	Denotes a quantity known at time t_k , i.e., not an estimate.
A	n_x -by- n_x state transition matrix.
B_u	n_x -by- n_u matrix mapping u to x .
B_v	n_x -by- n_x matrix mapping v to x .
C_m	n_{ym} -by- n_x matrix mapping x to y_m .
D_{vm}	n_{ym} -by- n_v matrix mapping v to y_m . Note that $D_{um} = 0$ because there can be no direct feedthrough between any manipulated variable and any measured output.
L	n_x -by- n_{ym} Kalman gain matrix for the time update. (See <code>kalmd</code> in the Control System Toolbox documentation.) Note that $L = A^*M$ minimizes the expected state estimation error for most combinations of plant and disturbance models used in MPC, but this is not true in general.
M	n_x -by- n_{ym} Kalman gain matrix for the measurement update. (See <code>kalmd</code> in the Control System Toolbox documentation.)

See Also

`getEstimator` | `kalman` | `mpc` | `mpcstate`

Introduced in R2014b

setindist

Modify unmeasured input disturbance model

Syntax

```
setindist(MPCobj, model ,model)
setindist(MPCobj, integrators )
```

Description

`setindist(MPCobj, model ,model)` sets the input disturbance model used by the model predictive controller, `MPCobj`, to a custom model.

`setindist(MPCobj, integrators)` sets the input disturbance model to its default value. Use this syntax if you previously set a custom input disturbance model and you want to change back to the default model. For more information on the default input disturbance model, see “MPC Modeling”.

Examples

Specify Input Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.d = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant, MV ,1, UD ,[2 3 4]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
```

```
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Define disturbance models such that:

- Input disturbance 1 is random white noise with a magnitude of 2.
- Input disturbance 2 is random step-like noise with a magnitude of 0.5.
- Input disturbance 3 is random ramp-like noise with a magnitude of 1.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the input disturbance model using the above transfer functions. Use a separate noise input for each input disturbance.

```
indist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the input disturbance model in the MPC controller.

```
setindist(MPCobj, model ,indist)
```

View the controller input disturbance model.

```
getindist(MPCobj)
```

```
ans =
```

```
A =
      x1   x2   x3
x1    1    0    0
x2    0    1    0
x3    0   0.1   1

B =
      Noise#1  Noise#2  Noise#3
x1      0      0.05     0
x2      0        0     0.1
x3      0        0    0.005
```

```
C =
    x1  x2  x3
UD1  0  0  0
UD2  1  0  0
UD3  0  0  1

D =
    Noise#1  Noise#2  Noise#3
UD1      2        0        0
UD2      0        0        0
UD3      0        0        0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `indist`, into a discrete-time state-space model.

Remove Input Disturbance for Particular Channel

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.d = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant, MV ,1, UD ,[2 3 4]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Retrieve the default input disturbance model from the controller.

```
distMod = getindist(MPCobj);
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
Assuming unmeasured input disturbance #2 is integrated white noise.
Assuming unmeasured input disturbance #3 is integrated white noise.
Assuming unmeasured input disturbance #4 is integrated white noise.
-->Assuming output disturbance added to measured output channel #1 is integrated white
Assuming no disturbance added to measured output channel #2.
Assuming no disturbance added to measured output channel #3.
Assuming no disturbance added to measured output channel #4.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each output.
```

Remove the integrator from the second input disturbance. Construct the new input disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setindist(MPCobj, model ,distMod)
```

When removing an integrator from the input disturbance model in this way, use **sminreal** to make the custom model structurally minimal.

View the input disturbance model.

```
tf(getindist(MPCobj))
```

```
ans =
From input "UD1-wn" to output...
    0.1
UD1: -----
      z - 1
UD2: 0
UD3: 0
From input "UD3-wn" to output...
UD1: 0
UD2: 0
    0.1
UD3: -----
      z - 1
```

```
Sample time: 0.1 seconds
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The first and third channels of the input disturbance model remain at their default values as discrete-time integrators.

Set Input Disturbance Model to Default Value

Define a plant model with no direct feedthrough.

```
plant = rss(2,2,3);
plant.d = 0;
```

Set the second and third input signals as input disturbances.

```
plant = setmpcsignals(plant, MV ,1, UD ,[2 3]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming defau
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 and zero weight for output(s) y2
```

Set the input disturbance model to unity gain for both channels.

```
setindist(MPCobj, model ,tf(eye(2)))
```

Restore the default input disturbance model.

```
setindist(MPCobj, integrators )
```

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

model — Custom input disturbance model

[] (default) | `ss` object | `tf` object | `zpk` object

Custom input disturbance model, specified as a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting `model` or specifying `model` as [] is equivalent to using `setindist(MPCObj, integrators)`.

The input disturbance model has:

- Unit-variance white noise input signals. For custom input disturbance models, the number of inputs is your choice.
- n_d outputs, where n_d is the number of unmeasured disturbance inputs defined in `MPCObj.Model.Plant`. Each disturbance model output is sent to the corresponding plant unmeasured disturbance input.

This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`setindist` does not check custom input disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands generate an error.

This syntax is equivalent to `MPCObj.Model.Disturbance = model`.

More About

Tips

- To view the current input disturbance model, use the `getindist` command.
- “MPC Modeling”
- “Controller State Estimation”
- “Adjusting Disturbance and Noise Models”

See Also

`getEstimator` | `getoutdist` | `mpc` | `setEstimator` | `setindist`

Introduced before R2006a

setmpcsignals

Set signal types in MPC plant model

Syntax

```
P = setmpcsignals(P,SignalType1,Channels1,SignalType2,Channels2,...)
```

Description

The purpose of `setmpcsignals` is to configure the input/output channels of the MPC plant model `P`. `P` must be an LTI object. Valid signal types, their abbreviations, and the channel type they refer to are listed below.

Signal Type	Abbreviation	Channel
Manipulated	MV	Input
MeasuredDisturbances	MD	Input
UnmeasuredDisturbances	UD	Input
MeasuredOutputs	MO	Output
UnmeasuredOutputs	UO	Output

Unambiguous abbreviations of signal types are also accepted.

Note When using `setmpcsignals` to modify an existing MPC object, be sure that the fields `Weights`, `MV`, `OV`, `DV`, `Model.Noise`, and `Model.Disturbance` are consistent with the new I/O signal types.

`P=setmpcsignals(P)` sets channel assignments to default, namely all inputs are manipulated variables (MVs), all outputs are measured outputs (MOs). More generally, input signals that are not explicitly assigned are assumed to be MVs, while unassigned output signals are considered as MOs.

Examples

Set MPC Signal Types and Create MPC Controller

Create a four-input, two output state-space plant model. By default all input signals are manipulated variables and all outputs are measured outputs.

```
plant = rss(3,2,4);  
plant.d = 0;
```

Configure the plant input/output channels such that:

- The second and third inputs are measured disturbances.
- The fourth input is an unmeasured disturbance.
- The second output is unmeasured.

```
plant = setmpcsignals(plant, MD ,[2 3], UD ,4, UO ,2);  
-->Assuming unspecified input signals are manipulated variables.  
-->Assuming unspecified output signals are measured outputs.
```

Create an MPC controller.

```
MPCobj = mpc(plant,1);  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1  
    for output(s) y1 and zero weight for output(s) y2
```

More About

- “MPC Modeling”

See Also

[mpc](#) | [set](#)

Introduced before R2006a

setname

Set I/O signal names in MPC prediction model

Syntax

```
setname(MPCobj, input ,I,name)
setname(MPCobj, output ,I,name)
```

Description

`setname(MPCobj, input ,I,name)` changes the name of the I th input signal to `name`. This is equivalent to `MPCobj.Model.Plant.InputName{I}=name`. Note that `setname` also updates the read-only `Name` fields of `MPCobj.DisturbanceVariables` and `MPCobj.ManipulatedVariables`.

`setname(MPCobj, output ,I,name)` changes the name of the I th output signal to `name`. This is equivalent to `MPCobj.Model.Plant.OutputName{I} =name`. Note that `setname` also updates the read-only `Name` field of `MPCobj.OutputVariables`.

Note The `Name` properties of `ManipulatedVariables`, `OutputVariables`, and `DisturbanceVariables` are read-only. You must use `setname` to assign signal names, or equivalently modify the `Model.Plant.InputName` and `Model.Plant.OutputName` properties of the MPC object.

See Also

`getname` | `mpc` | `set`

Introduced before R2006a

setoutdist

Modify unmeasured output disturbance model

Syntax

```
setoutdist(MPCobj, model ,model)
setoutdist(MPCobj, integrators )
```

Description

`setoutdist(MPCobj, model ,model)` sets the output disturbance model used by the model predictive controller, `MPCobj`, to a custom model.

`setoutdist(MPCobj, integrators)` sets the output disturbance model to its default value. Use this syntax if you previously set a custom output disturbance model and you want to change back to the default model. For more information on the default output disturbance model, see “MPC Modeling”.

Examples

Specify Output Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming defau
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define disturbance models for each output such that the output disturbance for:

- Channel 1 is random white noise with a magnitude of 2.
- Channel 2 is random step-like noise with a magnitude of 0.5.
- Channel 3 is random ramp-like noise with a magnitude of 1.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the output disturbance model using these transfer functions. Use a separate noise input for each output disturbance.

```
outdist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the output disturbance model in the MPC controller.

```
setoutdist(MPCobj, model ,outdist)
```

View the controller output disturbance model.

```
getoutdist(MPCobj)
```

```
ans =
```

```
A =
      x1    x2    x3
x1    1    0    0
x2    0    1    0
x3    0   0.1    1
```

```
B =
      Noise#1  Noise#2  Noise#3
x1      0     0.05     0
x2      0      0     0.1
x3      0      0    0.005
```

```
C =
      x1  x2  x3
M01    0  0  0
M02    1  0  0
M03    0  0  1
```

```
D =
      Noise#1  Noise#2  Noise#3
```

```
M01      2      0      0
M02      0      0      0
M03      0      0      0
```

```
Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `outdist`, into a discrete-time state-space model.

Remove Output Disturbance from Particular Output Channel

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Retrieve the default output disturbance model from the controller.

```
distMod = getoutdist(MPCobj);

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```

Remove the integrator from the second output channel. Construct the new output disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setoutdist(MPCobj, model ,distMod)
```

When removing an integrator from the output disturbance model in this way, use `sminreal` to make the custom model structurally minimal.

View the output disturbance model.

```
tf(getoutdist(MPCobj))
```

```
ans =
```

```
From input "Noise#1" to output...
```

```
    0.1
M01: -----
      z - 1
```

```
M02: 0
```

```
M03: 0
```

```
From input "Noise#2" to output...
```

```
M01: 0
```

```
M02: 0
```

```
    0.1
M03: -----
      z - 1
```

```
Sample time: 0.1 seconds
```

```
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The disturbance models for channels 1 and 3 remain at their default values as discrete-time integrators.

Remove Output Disturbances from All Output Channels

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.d = 0;
MPCobj = mpc(plant,1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
```

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Set the output disturbance model to zero for all three output channels.

```
setoutdist(MPCobj, model ,tf(zeros(3,1)))
```

View the output disturbance model.

```
getoutdist(MPCobj)
```

```
ans =
```

```
D =
    Noise#1
M01      0
M02      0
M03      0
```

```
Static gain.
```

A static gain of 0 for all output channels indicates that the output disturbances were removed.

Set Output Disturbance Model to Default Value

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(2,2,2);
plant.d = 0;
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Remove the output disturbances for all channels.

```
setoutdist(MPCobj, model ,tf(zeros(2,1)))
```

Restore the default output disturbance model.

```
setoutdist(MPCobj, integrators )
```

Input Arguments

MPCobj — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

model — Custom output disturbance model

[] (default) | `ss` object | `tf` object | `zpk` object

Custom output disturbance model, specified as a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting `model` or specifying `model` as [] is equivalent to using `setoutdist(MPCobj, integrators)`.

The output disturbance model has:

- Unit-variance white noise input signals. For custom output disturbance models, the number of inputs is your choice.
- n_y outputs, where n_y is the number of plant outputs defined in `MPCobj.Model.Plant`. Each disturbance model output is added to the corresponding plant output.

This model, along with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`setoutdist` does not check custom output disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands will generate an error.

More About

Tips

- To view the current output disturbance model, use the `getoutdist` command.
- “MPC Modeling”
- “Controller State Estimation”
- “Adjusting Disturbance and Noise Models”

See Also

`getEstimator` | `getoutdist` | `mpc` | `setEstimator` | `setindist`

Introduced in R2006a

setterminal

Terminal weights and constraints

Syntax

```
setterminal(MPCobj,Y,U)
setterminal(MPCobj,Y,U,Pt)
```

Description

`setterminal(MPCobj,Y,U)` specifies diagonal quadratic penalty weights and constraints at the last step in the prediction horizon. The weights and constraints are on the terminal output $y(t+p)$ and terminal input $u(t+p - 1)$, where p is the prediction horizon of the MPC controller `MPCobj`.

`setterminal(MPCobj,Y,U,Pt)` specifies diagonal quadratic penalty weights and constraints from step Pt to the horizon end. By default, Pt is the last step in the horizon.

Input Arguments

MPCobj

MPC controller, specified as an MPC controller object

Default:

Y

Terminal weights and constraints for the output variables, specified as a structure with the following fields:

Weight	1-by- n_y vector of nonnegative weights
Min	1-by- n_y vector of lower bounds
Max	1-by- n_y vector of upper bounds

MinECR	1-by- n_y vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
MaxECR	1-by- n_y vector of constraint-softening ECR values for the upper bounds

n_y is the number of controlled outputs of the MPC controller.

If the **Weight**, **Min** or **Max** field is empty, the values in **MPCobj** are used at all prediction horizon steps including the last. For the standard bounds, if any element of the **Min** or **Max** field is infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in “Standard Cost Function”). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See [Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation](#).

By default, $\mathbf{Y}.\text{MinECR} = \mathbf{Y}.\text{MaxECR} = 1$ (soft output constraints).

Choose the **ECR** magnitudes carefully, accounting for the importance of each constraint and the numerical magnitude of a typical violation.

Default:

U

Terminal weights and constraints for the manipulated variables, specified as a structure with the following fields:

Weight	1-by- n_u vector of nonnegative weights
Min	1-by- n_u vector of lower bounds
Max	1-by- n_u vector of upper bounds
MinECR	1-by- n_u vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
MaxECR	1-by- n_u vector of constraint-softening ECR values for the upper bounds

n_u is the number of manipulated variables of the MPC controller.

If the **Weight**, **Min** or **Max** field is empty, the values in **MPCobj** are used at all prediction horizon steps including the last. For the standard bounds, if individual elements of

the `Min` or `Max` fields are infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in “Standard Cost Function”). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation.

By default, `U.MinECR = U.MaxECR = 0` (hard manipulated variable constraints)

Choose the ECR magnitudes carefully, accounting for the importance of each constraint and the numerical magnitude of a typical violation.

Default:

Pt

Step in the prediction horizon, specified as an integer between 1 and p , where p is the prediction horizon. The terminal values are applied to `Y` and `U` from prediction step `Pt` to the end.

Default: Prediction horizon p

Examples

Specify Constraints and Penalty Weights at Last Prediction Horizon Step

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);
plant.d = 0;
MPCobj = mpc(plant,0.1);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
    for output(s) y1 y2 and zero weight for output(s) y3
```

Specify a prediction horizon of 8.

```
MPCobj.PredictionHorizon = 8;
```

Define the following penalty weights and constraints:

- Diagonal penalty weights of 1 and 10 on the first two output variables
- Lower bounds of 0 and -1 on the first and third outputs respectively
- Upper bound of 2 on the second output
- Lower bound of 1 on the first manipulated variable

```
Y = struct( Weight ,[1,10,0], Min ,[0,-Inf,-1], Max ,[Inf,2,Inf]);  
U = struct( Min ,[1,-Inf]);
```

Specify the constraints and penalty weights at the last step of the prediction horizon.

```
setterminal(MPCobj,Y,U)
```

Specify Terminal Constraints For Final Prediction Horizon Range

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);  
plant.d = 0;  
MPCobj = mpc(plant,0.1);  
  
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon  
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.  
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default  
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default  
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1  
    for output(s) y1 y2 and zero weight for output(s) y3
```

Specify a prediction horizon of 10.

```
MPCobj.PredictionHorizon = 10;
```

Define the following terminal constraints:

- Lower bounds of 0 and -1 on the first and third outputs respectively
- Upper bound of 2 on the second output
- Lower bound of 1 on the first manipulated variable

```
Y = struct( Min ,[0,-Inf,-1], Max ,[Inf,2,Inf]);
```

```
U = struct( Min , [1,-Inf]);
```

Specify the constraints beginning at step 5 and ending at the last step of the prediction horizon.

```
setterminal(MPCobj,Y,U,5)
```

- “Providing LQR Performance Using Terminal Penalty”
- Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation

More About

Tips

- Advanced users can impose terminal polyhedral state constraints:

$$K_1 \leq Hx \leq K_2.$$

First, augment the plant model with additional artificial (unmeasured) outputs, $y = Hx$. Then specify bounds K_1 and K_2 on these y outputs.

- “Terminal Weights and Constraints”

See Also

`mpc` | `mpcprops` | `setconstraint`

Introduced in R2011a

sim

Simulate closed-loop/open-loop response to arbitrary reference and disturbance signals for implicit or explicit MPC

Syntax

```
sim(MPCobj,T,r)
sim(MPCobj,T,r,v)
sim(____,SimOptions)
[y,t,u,xp,xmpc,SimOptions] = sim(____)
```

Description

Use `sim` to simulate the implicit (traditional) or explicit MPC controller in closed loop with a linear time-invariant model, which, by default, is the plant model contained in `MPCobj.Model.Plant`. As an alternative, `sim` can simulate the open-loop behavior of the model of the plant, or the closed-loop behavior in the presence of a model mismatch, when the controller's prediction model differs from the actual plant model.

`sim(MPCobj,T,r)` simulates the closed-loop system formed by the plant model specified in `MPCobj.Model.Plant` and by the MPC controller specified by the MPC controller `MPCobj`, in response to the specified reference signal, `r`. The MPC controller can be either a traditional MPC controller (`mpc`) or explicit MPC controller (`explicitMPC`). The simulation runs for the specified number of simulation steps, `T`. `sim` plots the simulation results.

`sim(MPCobj,T,r,v)` also specifies the measured disturbance signal `v`.

`sim(____,SimOptions)` specifies additional simulation options, which you create with `mpcsimopt`. This syntax allows you to alter the default simulation options, such as initial states, input/output noise and unmeasured disturbances, plant mismatch, etc. You can use `SimOptions` with any of the previous input combinations.

`[y,t,u,xp,xmpc,SimOptions] = sim(____)` suppresses plotting and instead returns the sequence of plant outputs `y`, the time sequence `t` (equally spaced by `MPCobj.Ts`), the manipulated variables `u` generated by the MPC controller, the sequence `xp` of states

of the model of the plant used for simulation, the sequence `xmpc` of states of the MPC controller (provided by the state observer), and the simulation options, `SimOptions`. You can use this syntax with any of the allowed input argument combinations.

Input Arguments

MPCobj

MPC controller containing the parameters of the Model Predictive Control law to simulate, specified as either an implicit MPC controller (`mpc`) or an explicit MPC controller (`generateExplicitMPC`).

T

Number of simulation steps, specified as a positive integer.

If you omit `T`, the default value is the row size of whichever of the following arrays has the largest row size:

- The input argument `r`
- The input argument `v`
- The `UnmeasuredDisturbance` property of `SimOptions`, if specified
- The `OutputNoise` property of `SimOptions`, if specified

Default: The largest row size of `r`, `v`, `UnmeasuredDisturbance`, and `OutputNoise`

r

Reference signal, specified as an array. This array has `ny` columns, where `ny` is the number of plant outputs. `r` can have anywhere from 1 to `T` rows. If the number of rows is less than `T`, the missing rows are set equal to the last row.

Default: `MPCobj.Model.Nominal.Y`

v

Measured disturbance signal, specified as an array. This array has `nv` columns, where `nv` is the number of measured input disturbances. `v` can have anywhere from 1 to `T` rows. If the number of rows is less than `T`, the missing rows are set equal to the last row.

Default: Corresponding entries from `MPCobj.Model.Nominal.U`

SimOptions

Simulation options, specified as an options object you create using `mpcsimopt`.

Default: []

Output Arguments

y

Sequence of controlled plant outputs, returned as a T -by- Ny array, where T is the number of simulation steps and Ny is the number of plant outputs. The values in y do not include additive measurement noise, if any).

t

Time sequence, returned as a T -by-1 array, where T is the number of simulation steps. The values in t are equally spaced by `MPCobj.Ts`.

u

Sequence of manipulated variables generated by the MPC controller, returned as a T -by- Nu array, where T is the number of simulation steps and Nu is the number of manipulated variables.

xp

Sequence of plant model states, T -by- Nxp array, where T is the number of simulation steps and Nxp is the number of states in the plant model. The plant model is either `MPCobj.Model` or `SimOptions.Model`, if the latter is specified.

xmpc

Sequence of MPC controller state estimates, returned as a T -by-1 structure array. Each entry in the structure array has the same fields as an `mpcstate` object. The state estimates include plant, disturbance, and noise model states at each time step.

SimOptions

Simulation options used, returned as a `mpcsimopt` object.

Examples

Simulate MPC Control of MISO Plant

Simulate the MPC control of a MISO system. The system has one manipulated variable, one measured disturbance, one unmeasured disturbance, and one output.

Create the continuous-time plant model. This plant will be used as the prediction model for the MPC controller.

```
sys = ss(tf({1,1,1}, {[1 .5 1], [1 1], [.7 .5 1]}));
```

Discretize the plant model using a sampling time of 0.2 units.

```
Ts = 0.2;
sysd = c2d(sys,Ts);
```

Specify the MPC signal type for the plant input signals.

```
sysd = setmpcsignals(sysd, MV ,1, MD ,2, UD ,3);
```

Create an MPC controller for the `sysd` plant model. Use default values for the weights and horizons.

```
MPCobj = mpc(sysd);

-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming defau
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Constrain the manipulated variable to the [0 1] range.

```
MPCobj.MV = struct( Min ,0, Max ,1);
```

Specify the simulation stop time.

```
Tstop = 30;
```

Define the reference signal and the measured disturbance signal.

```
num_sim_steps = round(Tstop/Ts);
```

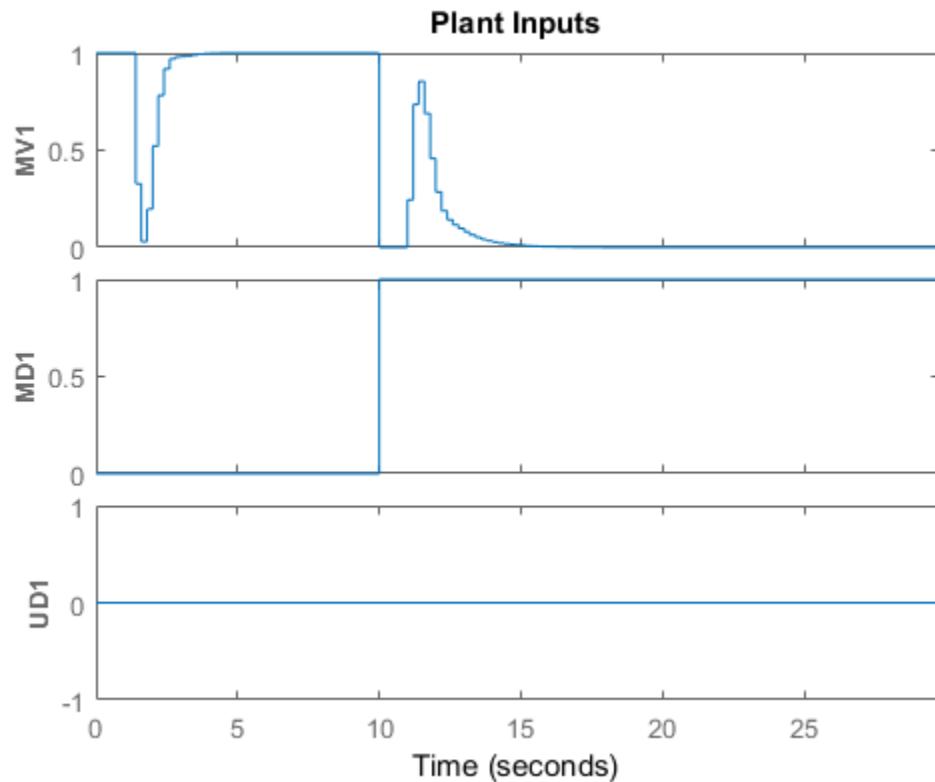
```
r = ones(num_sim_steps,1);  
v = [zeros(num_sim_steps/3,1); ones(2*num_sim_steps/3,1)];
```

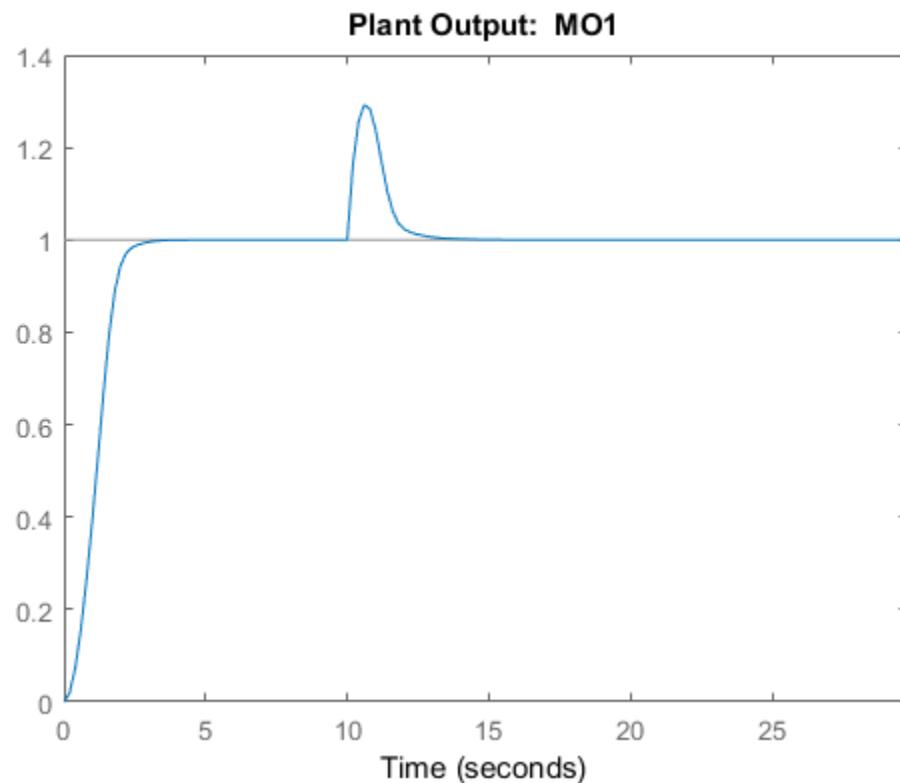
The reference signal, r , is a unit step. The measured disturbance signal, v , is a unit step, with a 10 unit delay.

Simulate the controller.

```
sim(MPCobj,num_sim_steps,r,v)
```

```
-->The "Model.Disturbance" property of "mpc" object is empty:  
Assuming unmeasured input disturbance #3 is integrated white noise.  
Assuming no disturbance added to measured output channel #1.  
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each
```



**See Also**

[mpcsimopt](#) | [mpc](#) | [mpcmove](#)

Introduced before R2006a

simplify

Reduce explicit MPC controller complexity and memory requirements

Syntax

```
EMPCreduced = simplify(EMPCobj, exact )
EMPCreduced = simplify(EMPCobj, exact ,uniteeps)
EMPCreduced = simplify(EMPCobj, radius ,r)
EMPCreduced = simplify(EMPCobj, sequence ,index)
simplify(EMPCobj, ___ )
```

Description

`EMPCreduced = simplify(EMPCobj, exact)` attempts to reduce the number of piecewise affine (PWA) regions in an explicit MPC controller by merging regions that have identical controller gains and whose union is a convex set. Reducing the number of PWA regions reduces memory requirements of the controller. This command returns a reduced controller, `EMPCreduced`.

`EMPCreduced = simplify(EMPCobj, exact ,uniteeps)` specifies the tolerance for identifying regions that can be merged.

`EMPCreduced = simplify(EMPCobj, radius ,r)` retains only regions whose Chebyshev radius (the radius of the largest ball contained in the region) is larger than `r`.

`EMPCreduced = simplify(EMPCobj, sequence ,index)` eliminates all regions except those specified in an index vector.

`simplify(EMPCobj, ___)` applies the reduction to the explicit MPC controller `EMPCobj`, rather than returning a new controller object. You can use this syntax with any of the previous reduction options.

Input Arguments

EMPCobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller to reduce, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

uniteeps — Tolerance for joining regions

0.001 (default) | positive scalar

Tolerance for joining PWA regions, specified as a positive scalar.

r — Minimum Chebyshev radius

0 (default) | nonnegative scalar

Minimum Chebyshev radius for retaining PWA regions, specified as a nonnegative scalar. When you use the `radius` option, `simplify` keeps only the regions whose Chebyshev radius is larger than `r`. The default value is 0, which causes all regions to be retained.

index — Indices of PWA regions to retain

1:nr (default) | vector

Indices of PWA regions to retain, specified as a vector. The default value is `[1:nr]`, where `nr` is the number of PWA regions in `EMPCobj`. Thus, by default, all regions are retained. You can obtain a sequence of regions to retain by performing simulations using `EMPCobj` and recording the indices of regions actually encountered.

Output Arguments

EMPCreduced — Reduced MPC controller

explicit MPC controller object

Reduced MPC controller, returned as an Explicit MPC controller object.

See Also

`generateExplicitMPC`

Introduced in R2014b

size

Size and order of MPC Controller

Syntax

```
mpc_obj_size = size(MPCobj)
mpc_obj_size = size(MPCobj,signal_type)
size(MPCobj)
```

Description

`mpc_obj_size = size(MPCobj)` returns a row vector specifying the number of manipulated inputs and measured controlled outputs of an MPC controller. This row vector contains the elements [$n_u \ n_{ym}$], where n_u is the number of manipulated inputs and n_{ym} is the number of measured controlled outputs.

`mpc_obj_size = size(MPCobj,signal_type)` returns the number of signals of the specified type that are associated with the MPC controller.

You can specify `signal_type` as one of the following strings:

- `uo` — Unmeasured controlled outputs
- `md` — Measured disturbances
- `ud` — Unmeasured disturbances
- `mv` — Manipulated variables
- `mo` — Measured controlled outputs

`size(MPCobj)` displays the size information for all the signal types of the MPC controller.

See Also

`mpc | set`

Introduced before R2006a

ss

Convert unconstrained MPC controller to state-space linear system

Syntax

```
sys = ss(MPCobj)
sys = ss(MPCobj,signals)
sys = ss(MPCobj,signals,ref_preview,md_preview)
[sys,ut] = ss(MPCobj)
```

Description

The `ss` command returns a linear controller in the state-space form. The controller is equivalent to the traditional (implicit) MPC controller `MPCobj` when no constraints are active. You can then use Control System Toolbox software for sensitivity analysis and other diagnostic calculations.

`sys = ss(MPCobj)` returns the linear discrete-time dynamic controller `sys`

$$x(k+1) = Ax(k) + By_m(k)$$

$$u(k) = Cx(k) + Dy_m(k)$$

where y_m is the vector of measured outputs of the plant, and u is the vector of manipulated variables. The sampling time of controller `sys` is `MPCobj.Ts`.

Note Vector x includes the states of the observer (plant + disturbance + noise model states) and the previous manipulated variable $u(k-1)$.

`sys = ss(MPCobj,signals)` returns the linearized MPC controller in its full form and allows you to specify the signals that you want to include as inputs for `sys`.

The full form of the MPC controller has the following structure:

$$x(k+1) = Ax(k) + By_m(k) + B_r r(k) + B_v v(k) + B_{ut} u_{target}(k) + B_{off}$$

$$u(k) = Cx(k) + Dy_m(k) + D_r r(k) + D_v v(k) + D_{ut} u_{target}(k) + D_{off}$$

Here, r is the vector of setpoints for both measured and unmeasured plant outputs, v is the vector of measured disturbances, u_{target} is the vector of preferred values for manipulated variables.

Specify **signals** as a single or multicharacter string constructed using any of the following:

- **r** — Output references
- **v** — Measured disturbances
- **o** — Offset terms
- **t** — Input targets

For example, to obtain a controller that maps $[y_m; r; v]$ to u , use:

```
sys = ss(MPCobj, rv);
```

In the general case of nonzero offsets, y_m (as well as r , v , and u_{target}) must be interpreted as the difference between the vector and the corresponding offset. Offsets can be nonzero if **MPCobj.Model.Nominal.Y** or **MPCobj.Model.Nominal.U** are nonzero.

Vectors B_{off} , D_{off} are constant terms. They are nonzero if and only if **MPCobj.Model.Nominal.DX** is nonzero (continuous-time prediction models), or **MPCobj.Model.Nominal.Dx-MPCobj.Model.Nominal.X** is nonzero (discrete-time prediction models). In other words, when **Nominal.X** represents an equilibrium state, B_{off} , D_{off} are zero.

Only the following fields of **MPCobj** are used when computing the state-space model: **Model**, **PredictionHorizon**, **ControlHorizon**, **Ts**, **Weights**.

sys = ss(MPCobj,signals,ref_preview,md_preview) specifies if the MPC controller has preview actions on the reference and measured disturbance signals. If the flag **ref_preview= on**, then matrices B_r and D_r multiply the whole reference sequence:

$$x(k+1) = Ax(k) + By_m(k) + B_r[r(k);r(k+1);...;r(k+p-1)] + \dots$$

$$u(k) = Cx(k) + Dy_m(k) + D_r[r(k);r(k+1);...;r(k+p-1)] + \dots$$

Similarly if the flag `md_preview= on`, then matrices B_v and D_v multiply the whole measured disturbance sequence:

$$x(k+1) = Ax(k) + \dots + B_v[v(k); v(k+1); \dots; v(k+p)] + \dots$$

$$u(k) = Cx(k) + \dots + D_v[v(k); v(k+1); \dots; v(k+p)] + \dots$$

`[sys,ut] = ss(MPCobj)` additionally returns the input target values for the full form of the controller.

`ut` is returned as a vector of doubles, `[utarget(k); utarget(k+1); ... utarget(k+h)]`.

Here:

- h — Maximum length of previewed inputs, that is, `h = max(length(MPCobj.ManipulatedVariables(:).Target))`
- `utarget` — Difference between the input target and corresponding input offsets, that is, `MPCobj.ManipulatedVariables(:).Targets - MPCobj.Model.Nominal.U`

Examples

Convert Unconstrained MPC Controller to State-Space Model

To improve the clarity of the example, suppress messages about working with an MPC controller.

```
old_status = mpcverbosity( off );
```

Create the plant model.

```
G = rss(5,2,3);
G.D = 0;
G = setmpcsignals(G, mv ,1, md ,2, ud ,3, mo ,1, uo ,2);
```

Configure the MPC controller with nonzero nominal values, weights, and input targets.

```
C = mpc(G,0.1);
C.Model.Nominal.U = [0.7 0.8 0];
C.Model.Nominal.Y = [0.5 0.6];
C.Model.Nominal.DX = rand(5,1);
```

```
C.Weights.MV = 2;
C.Weights.OV = [3 4];
C.MV.Target = [0.1 0.2 0.3];
```

C is an unconstrained MPC controller. Specifying **C.Model.Nominal.DX** as nonzero means that the nominal values are not at steady state. **C.MV.Target** specifies three preview steps.

Convert **C** to a state-space model.

```
sys = ss(C);
```

The output, **sys**, is a seventh-order SISO state-space model. The seven states include the five plant model states, one state from the default input disturbance model, and one state from the previous move, $u(k-1)$.

Restore **mpcverbosity**.

```
mpcverbosity(old_status);
```

See Also

mpc | **set** | **tf** | **zpk**

Introduced before R2006a

tf

Convert unconstrained MPC controller to linear transfer function

Syntax

```
sys=tf(MPCobj)
```

Description

The `tf` function computes the transfer function of the linear controller `ss(MPCobj)` as an LTI system in `tf` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `zpk`

Introduced before R2006a

trim

Compute steady-state value of MPC controller state for given inputs and outputs

Syntax

```
x = trim(MPCobj,y,u)
```

Description

The `trim` function finds a steady-state value for the plant state or the best approximation in a least squares sentence such that:

$$\begin{aligned}x - x_{off} &= A(x - x_{off}) + B(u - u_{off}) \\y - y_{off} &= C(x - x_{off}) + D(u - u_{off})\end{aligned}$$

Here, x_{off} , u_{off} , and y_{off} are the nominal values of the extended state x , input u , and output y .

x is returned as an `mpcstate` object. Specify y and u as doubles. y specifies the measured and unmeasured output values. u specifies the manipulated variable, measured disturbance, and unmeasured disturbance values. The values for unmeasured disturbances must be 0.

`trim` assumes the disturbance model and measurement noise model to be zero when computing the steady-state value. The software uses the extended state vector to perform the calculation.

See Also

`mpc` | `mpcstate`

Introduced before R2006a

zpk

Convert unconstrained MPC controller to zero/pole/gain form

Syntax

```
sys=zpk(MPCobj)
```

Description

The `zpk` function computes the zero-pole-gain form of the linear controller `ss(MPCobj)` as an LTI system in `zpk` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `tf`

Introduced before R2006a

Block Reference

MPC Controller

Compute MPC control law

Library

MPC Simulink Library

Description

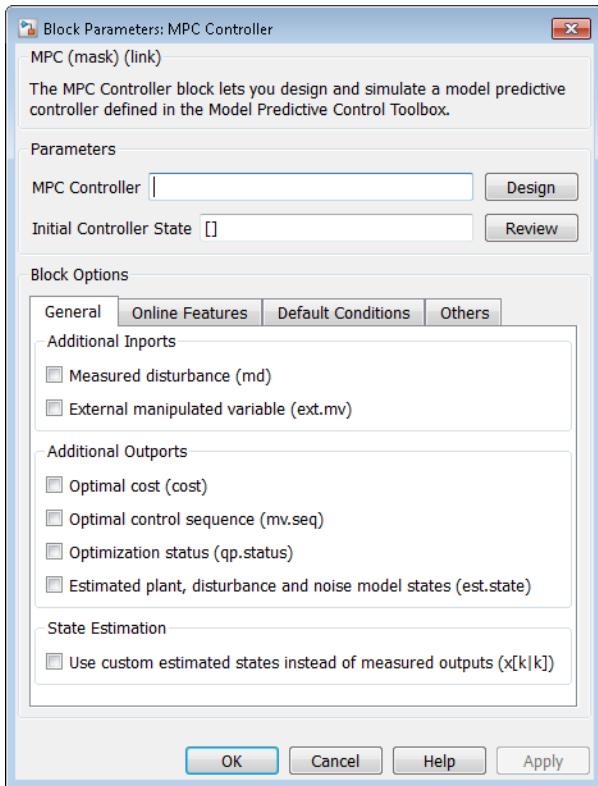


The **MPC Controller** block receives the current measured output signal (`mo`), reference signal (`ref`), and optional measured disturbance signal (`md`). The block computes the optimal manipulated variables (`mv`) by solving a quadratic program (QP).

To use the block in simulation and code generation, you must specify an `mpc` object, which defines a model predictive controller. This controller must have already been designed for the plant that it will control.

Because the **MPC Controller** block uses **MATLAB Function** blocks, it requires compilation each time you change the MPC object and block. Also, because MATLAB does not allow compiled code to reside in any MATLAB product folder, you must use a non-MATLAB folder to work on your Simulink model when you use MPC blocks.

Dialog Box



The MPC Controller block has the following parameter groupings:

- “Parameters” on page 2-4
- “Required Imports” on page 2-5
- “Required Outports” on page 2-6
- “Additional Imports (General tab)” on page 2-6
- “Additional Outports (General tab)” on page 2-8
- “State Estimation (General tab)” on page 2-10
- “Constraints (Online Features tab)” on page 2-11

- “Weights (Online Features tab)” on page 2-12
- “MV Targets (Online Features tab)” on page 2-14
- “Default Conditions tab” on page 2-14
- “Others tab” on page 2-15

Parameters

MPC controller

You must provide a traditional (implicit) `mpc` object that defines your controller using one of the following methods:

- Enter the name of an `mpc` object in the **MPC Controller** edit box. This object must be present in the MATLAB workspace.

If you want to modify the controller settings in a graphical environment, click **Design** to open the MPC Designer app. For example, you can:

- Import a new prediction model.
- Change horizons, constraints, and weights.
- Evaluate MPC performance with a linear plant.
- Export the updated controller to the MATLAB workspace.

To see how well the controller works for the nonlinear plant, run a closed-loop Simulink simulation.

- If you do not have an existing `mpc` object in the MATLAB workspace, leave the **MPC controller** field empty. With the **MPC Controller** block connected to the plant, click **Design** to open the MPC Designer app. Using the app, linearize the Simulink model at a specified operating point, and design your controller. For more information, see “Design MPC Controller in Simulink” and “Linearize Simulink Models Using MPC Designer”.

To use this design approach, you must have Simulink Control Design software.

If you specified a controller in the **MPC Controller** field, click **Review** to review your design for run-time stability and robustness issues. For more information, see “Review Model Predictive Controller for Stability and Robustness Issues”.

Initial controller state

Specifies the initial controller state. If this parameter is left blank, the block uses the nominal values that are defined in the `Model.Nominal` property of the `mpc` object. To override the default, create an `mpcstate` object in your workspace, and enter its name in the field.

Required Imports

Measured output or State estimate

If your controller uses default state estimation, this import is labeled `mo`. Connect this import to the measured signals from the plant.

If your controller uses custom state estimation, check **Use custom estimated states instead of measured outputs** in the **General** tab. Checking that option changes the label on this import to `x[k | k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k .

Reference

The `ref` dimension must not change from one control instant to the next. Each element must be a real number.

When `ref` is a 1-by- n_y signal, where n_y is the number of outputs, there is no reference signal previewing. All the current values are applied across the prediction horizon.

To use signal previewing, specify `ref` as an N -by- n_y signal, where N is the number of time steps for which you are specifying reference values. Here, $1 < N \leq p$, and p is the prediction horizon. Previewing usually improves performance since the controller can anticipate future reference signal changes. The first row of `ref` specifies the n_y references for the first step in the prediction horizon (at the next control interval $k = 1$), and so on for N steps. If $N < p$, the last row designates constant reference values for the remaining $p - N$ steps.

For example, suppose $n_y = 2$ and $p = 6$. At a given control instant, the signal connected to the `ref` import is:

```
[2 5 ← k=1
 2 6 ← k=2
```

```
2 7  ←  k=3
2 8] ←  k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step $k = 1$ are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step $k = 4$ in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

`mpcpreview` shows how to use reference previewing in a specific case. For calculation details on the use of the reference signal, see “Optimization Problem”.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport by solving a quadratic program at each control instant.

Additional Imports (General tab)

Measured disturbance

Add an import (`md`) to which you can connect a measured disturbance signal.

The `md` dimension must not change from one control instant to the next. Each element must be a real number.

When `md` is a 1-by- n_{md} signal, where $n_{md} \geq 1$ is the number of measured disturbances defined for your controller, there is no measured disturbance previewing. All the current values are applied across the prediction horizon.

To use disturbance previewing, specify `ref` as an N -by- n_{md} signal, where N is the number of time steps for which the MD is known. Here, $1 < N \leq p + 1$, and p is the prediction

horizon. Previewing usually improves performance since the controller can anticipate future disturbances. The first row of `md` specifies the n_{md} current disturbance values ($k=1$), with other rows specifying disturbances for subsequent control intervals. If $N < p + 1$, the last row designates constant reference values for the remaining $p - N + 1$ steps.

For example, suppose $n_{md} = 2$ and $p = 6$. At a given control instant, the signal connected to the `md` import is:

```
[2 5 ← k=0
 2 6 ← k=1
 2 7 ← k=2
 2 8] ← k=3
```

This signal informs the controller that:

- The current MD values are 2 and 5 at $k = 0$.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the third step $k = 3$ in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

`mpcpreview` shows how to use MD previewing in a specific case.

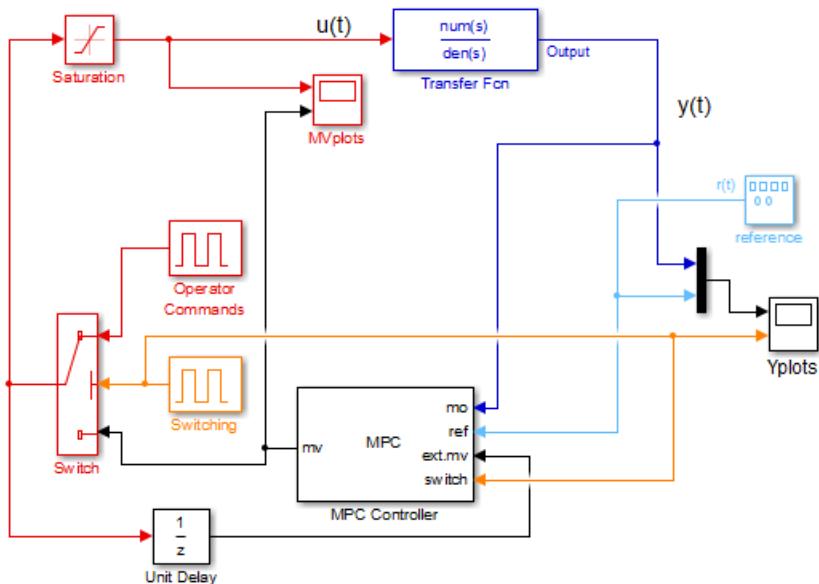
For calculation details, see “MPC Modeling” and “QP Matrices”.

External manipulated variable

Add an import (`ext.mv`), which you can connect to the actual manipulated variables (MV) used in the plant. The block uses these to update its internal state estimates.

Controller state estimation assumes that the MV is piecewise constant. At time t_k , the `ext.mv` value must be the effective MV between times t_{k-1} and t_k . For example, if the MV is actually varying over this interval, you might supply the time-averaged value evaluated at time t_k .

The following example, from the model `mpc_bumpless`, includes a switch that can override the controller’s output with a signal supplied by the operator. Also, the controller output may saturate. Feeding back the actual MV used in the plant (labeled `u(t)` in the example) improves the accuracy of controller state estimates.



If the external MV option is inactive or the `ext.mv` import is unconnected, the controller assumes that its MV output is used in the plant without modification.

Note There is direct feed through from the `ext.mv` import to the `mv` outputport. Thus, use of this option may cause an algebraic loop in the Simulink diagram. In the above examples, the insertion of a unit delay block avoids an algebraic loop.

Additional Outports (General tab)

Optimal cost

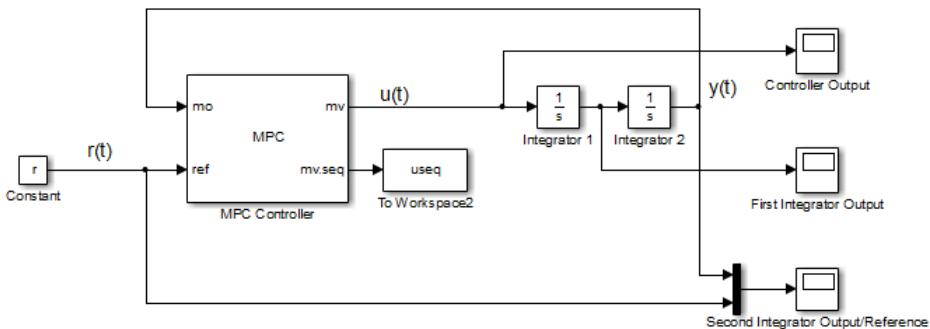
Add an outport (`cost`) that provides the optimal quadratic programming objective function value at the current time (a nonnegative scalar). If the optimization problem is infeasible, however, the value is meaningless. (See `qp.status`.)

Optimal control sequence

Add an outport (`mv.seq`) that provides the computed optimal MV sequence for the entire prediction horizon from $k=0$ to $k = p-1$. If n_u is the number of MVs and p is the length

of the prediction horizon, this signal is a p by n_u matrix. The first row represents $k=0$ and duplicates the block's MV outport.

The following block diagram (from Analysis of Control Sequences Optimized by MPC on a Double Integrator System) illustrates how to use this option. The diagram shows how to collect diagnostic data and send it to the **To Workspace2** block, which creates the variable, `useq`, in the workspace. Run the example to see how the optimal sequence evolves with time.



Optimization status

Add an outport (`qp.status`) that allows you to monitor the status of the QP solver.

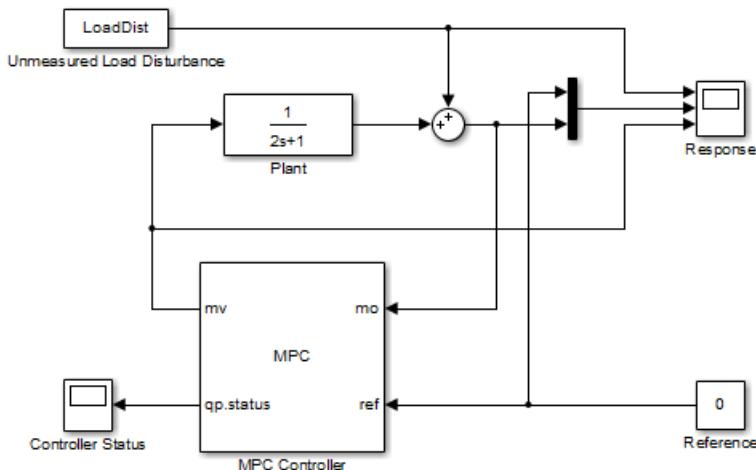
If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution at this time interval.

The QP solver may fail to find an optimal solution for the following reasons:

- `qp.status = 0` — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object.
- `qp.status = -1` — The QP solver detects an infeasible QP problem. See Monitoring Optimization Status to Detect Controller Failures for an example where a large, sustained disturbance drives the OV outside its specified bounds.
- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem.

For all the previous three failure modes, the MPC Controller block holds its `mv` output at the most recent successful solution. In a real-time application, you can use status indicator to set an alarm or take other special action.

The next diagram shows how to use the status indicator to monitor the MPC Controller block in real time. See Monitoring Optimization Status to Detect Controller Failures for more details.



Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the controller state estimates, $x[k|k]$, at each control instant. These include the plant, disturbance and noise model states.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

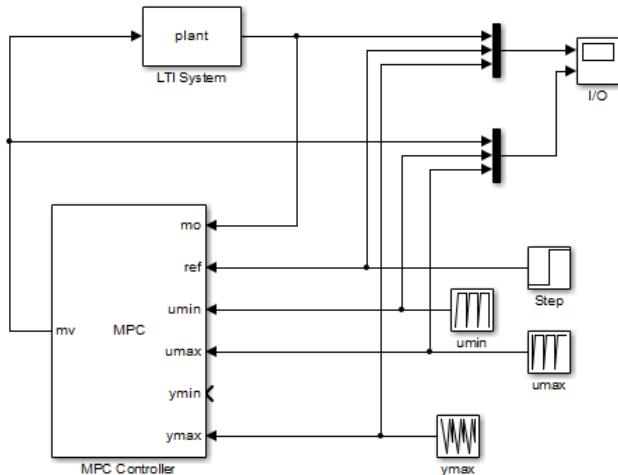
Replace `mo` with the $x[k|k]$ import for custom state estimation as described in “Required Imports” on page 2-5.

Constraints (Online Features tab)

Plant input and output limits

Add imports (u_{\min} , u_{\max} , y_{\min} , y_{\max}) that you can connect to run-time constraint signals.

u_{\min} and u_{\max} are vectors with n_u elements. y_{\min} and y_{\max} are vectors with n_y elements.



If any of these imports are unconnected, they are treated as unbounded signals. The corresponding variable in the `mpc` object must also be unbounded.

For connected imports, the following rules apply:

- All connected signals must be finite. Simulink does not support infinite signals.
- If a variable is unconstrained in the controller object, the connected value is ignored.

If this check box is not selected, the block uses the constant constraint values stored within its `mpc` object.

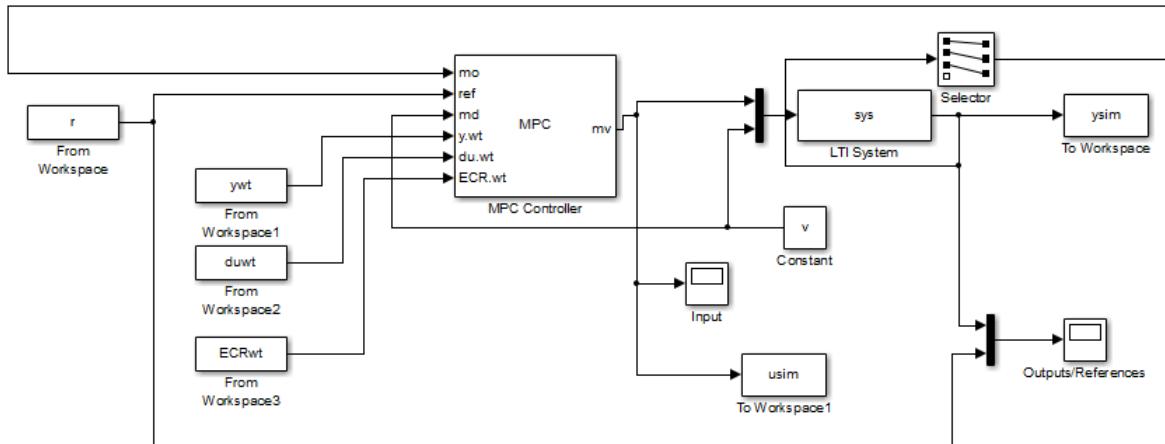
Note: You cannot specify time-varying constraints at run time using a matrix signal.

Weights (Online Features tab)

A controller intended for real-time applications should have “knobs” you can use to tune its performance when it operates with the real plant. This group of optional imports serves that purpose.

The diagram shown below shows three of the **MPC Controller** tuning imports. In this simulation context, the imports are tuned using pre-stored signals (the `ywt`, `duwt`, and `ECRwt` variables in the **From Workspace** blocks). In practice, you would connect a knob or similar manual adjustment.

Note: You cannot specify time-varying weights at run time using a matrix signal.



Weights on plant outputs

Add an import (`y.wt`) for a vector signal with n_y elements. Each element specifies a nonnegative tuning weight for each controlled output variable (OV). This signal overrides the `MPCobj.Weights.OV` property of the `mpc` object, which establishes the relative importance of OV reference tracking.

For example, if the preceding controller defined three OVs, the signal connected to the `y.wt` import should be a vector with three elements. If the second element is relatively large, the controller would place a relatively high priority on making OV(2) track the

$r(2)$ reference signal. Setting a $y.wt$ signal to zero turns off reference tracking for that OV.

If you do not connect a signal to the $y.wt$ import, the block uses the OV weights specified in your MPC object, and these values remain constant.

Weights on manipulated variables

Add an import ($u.wt$), whose input is a vector signal defining n_u nonnegative weights, where n_u is the number of manipulated variables (MVs). The input overrides the `MPCObj.Weights.MV` property of the `mpc` object, which establishes the relative importance of MV target tracking.

For example, if your controller defines four MVs and the second $u.wt$ element is relatively large, the controller would try to keep MV(2) close to its specified target (relative to other control objectives).

If you do not connect a signal to the $u.wt$ import, the block uses the `Weights.MV` weights property specified in your `mpc` object, and these values remain constant.

Weights on manipulated variable changes

Add an import ($du.wt$), for a vector signal defining n_u nonnegative weights, where n_u is the number of manipulated variables (MVs). The input overrides the `MPCObj.Weights.MVrate` property of the `mpc` object, which establishes the relative importance of MV changes.

For example, if your controller defines four MVs and the second $du.wt$ element is relatively large, the controller would use relatively small changes in the second MV. Such *move suppression* makes the controller less aggressive. However, too much suppression makes it sluggish.

If you do not connect a signal to the $du.wt$ import, the block uses the `Weights.MVrate` property specified in your `mpc` object, and these values remain constant.

Weight on overall constraint softening

Add an import ($ECR.wt$), for a scalar nonnegative signal that overrides the `mpc` controller's `MPCObj.Weights.ECR` property. This import has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero.

If there are soft constraints, increasing the `ECR.wt` value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

You may not be able to avoid violations of an output variable constraint. Thus, increasing the `ECR.wt` value is often counterproductive. Such an increase causes the controller to pay less attention to its other objectives and does not help reduce constraint violations. You usually need to tune `ECR.wt` to achieve the proper balance in relation to the other control objectives.

MV Targets (Online Features tab)

Targets for manipulated variables

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` import to which you can connect the target signal (dimension n_u , where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Default Conditions tab

Specify the default block sample time and signal dimensions for performing simulation, trimming, or linearization. In these cases, the `mv` output signal remains at zero. You must specify default condition values that are compatible with your Simulink model design.

Note: These default conditions apply only if the **MPC Controller** field is empty. If you specify a controller from the MATLAB workspace, the sample time and signal sizes from the specified controller are used.

Sample Time

Specify the default controller sample time.

Plant Input Signal Sizes

Specify the default signal dimensions for the following input signal types:

- Manipulated variables
- Unmeasured disturbances
- Measured disturbances

Note: You can specify the measured disturbances signal dimension only if, on the **General** tab, in the **Additional Imports** section, the **Measured disturbance** option is selected.

Plant Output Signal Sizes

Specify the default signal dimensions for the following output signal types:

- Measured outputs
- Unmeasured outputs

Others tab

Block data type

Specify the block data type as one of the following:

- **double** — Double-precision floating point (default).
- **single** — Single-precision floating point.

Specify the output data type as **single** if you are implementing the **MPC Controller** block on a single-precision target.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “[Simulation and Code Generation Using Simulink Coder](#)”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & Ports** **Port Data Types**.

Inherit sample time

Use the sample time inherited from the parent subsystem as the **MPC Controller** block’s sample time.

Inheriting the sample time allows you to conditionally execute the **MPC Controller** block inside the **Function-Call Subsystem** or **Triggered Subsystem** blocks.

For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note: When you place an **MPC controller** block inside a **Function-Call Subsystem** or **Triggered Subsystem** block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see "View Sample Time Information".

Use external signal to enable or disable optimization

Add an import (**switch**) whose input specifies whether the controller performs optimization calculations. If the input signal is zero, the controller behaves normally. If the input signal becomes nonzero, the **MPC Controller** block turns off the controller optimization calculations. This action reduces computational effort when the controller output is not needed, such as when the system is operating manually or another controller has taken over. The controller, however, continues to update its internal state estimate in the usual way. Thus, it is ready to resume optimization calculations whenever the **switch** signal returns to zero. While the controller optimization is turned off, the **MPC Controller** block passes the current **ext.mv** signal to the controller output. If the **ext.mv** import is not enabled, the controller output is held at whatever value it had when the optimization was disabled.

See Also

[mpc](#) | MPC Designer | [mpcstate](#) | Multiple MPC Controllers

Related Examples

- “Design MPC Controller in Simulink”
- MPC Control with Input Quantization Based on Comparing the Optimal Costs
- Analysis of Control Sequences Optimized by MPC on a Double Integrator System
- “Simulation and Code Generation Using Simulink Coder”
- “Simulation and Structured Text Generation Using PLC Coder”

More About

- “MPC Modeling”

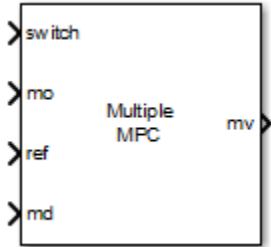
Multiple MPC Controllers

Simulate switching between multiple MPC controllers

Library

MPC Simulink Library

Description



As for the **MPC Controller** block, at each control instant the **Multiple MPC Controllers** block receives the current measured plant output, reference, and measured plant disturbance (if any). In addition, it receives a switching signal that selects the *active controller* from among a list of two or more candidates. The active controller then solves a quadratic program to determine the optimal plant manipulated variables for the current input signals.

The **Multiple MPC Controllers** block allows you to achieve better control when operating conditions change. In conventional feedback control, you might compensate for this by gain scheduling. In a similar manner, the **Multiple MPC Controllers** block allows you to transition between multiple MPC controllers in real-time based on the current conditions. Typically, you design each such controller for a particular region of the operating space. Using the available measurements, you detect the current operating region and choose the appropriate active controller.

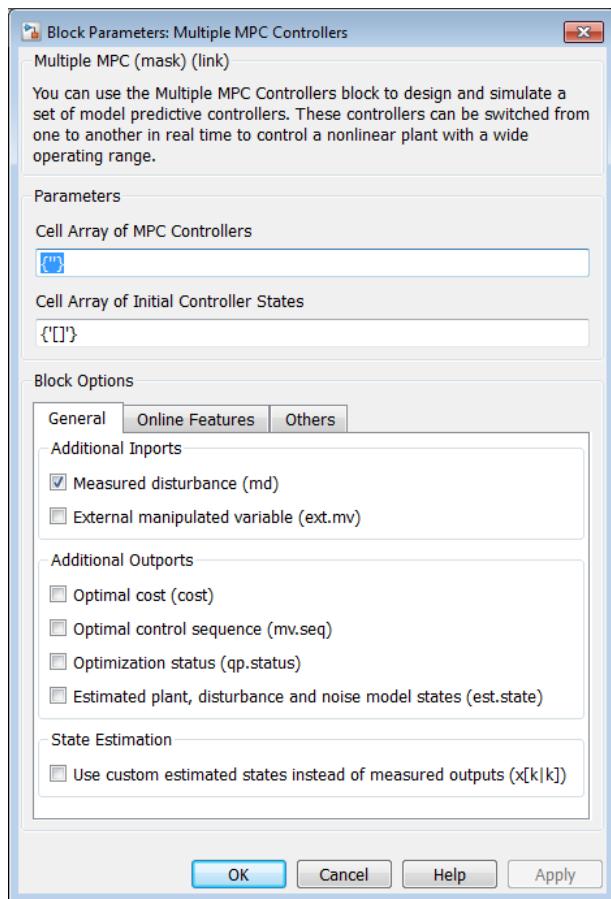
The **Adaptive MPC Controller** block compensates for operating point variations by modifying its prediction model. The advantages of the **Multiple MPC Controllers** block over **Adaptive MPC Controller** block are as follows:

- Simpler configuration – There is no need to identify prediction model parameters using online data.
- Its candidate controllers form a limited set that you can test thoroughly.

The **Multiple MPC Controllers** block lacks several optional features found in the **MPC Controller** block, as follows:

- You cannot disable optimization. One controller must always be active.
- You cannot initiate a controller design from within the block dialog, that is there is no **Design** button. You must design all candidate controllers before configuring the **Multiple MPC Controllers** block.
- Similarly, there is no **Review** button. Instead, use the **review** command or the MPC Designer app.

Dialog Box



The **MPC Controller** block has the following parameter groupings:

- “Parameters” on page 2-21
- “Required Imports” on page 2-21
- “Required Outports” on page 2-23
- “Additional Inputs (General tab)” on page 2-23
- “Additional Outputs (General tab)” on page 2-24

- “State Estimation (General tab)” on page 2-25
- “Constraints (Online Features tab)” on page 2-26
- “Weights (Online Features tab)” on page 2-26
- “MV Targets (Online Features tab)” on page 2-27
- “Others tab” on page 2-27

Parameters

Cell Array of MPC Controllers

Candidate controllers, specified as:

- A cell array of `mpc` objects.
- A cell array of strings, where each string is the name of an `mpc` object in the MATLAB workspace.

The specified array must contain at least two candidate controllers. The first entry in the cell array is the controller that corresponds to a switch input value of 1, the second corresponds to a switch input value of 2, and so on.

Cell Array of Initial Controller States

Optional initial states for each candidate controller, specified as:

- A cell array of `mpcstate` objects.
- A cell array of strings, where each string is the name of an `mpcstate` object in the MATLAB workspace.
- `{[],[],...}` or `{[],[],...}` — Use the nominal condition defined in `Model.Nominal` as the initial state for each controller. If you specify an empty object for one initial state, you must do so for all initial states.

Required Inputs

Controller Selection

The `switch` input signal must be a scalar integer between 1 and n_c , where n_c is the number of controllers listed in your block mask. At each control instant, this signal designates the active controller.

Measured output or State estimate

If all candidate controllers use default state estimation, this import is labeled `mo`. Connect the measured plant output variables.

If all candidate controllers use custom state estimation, check **Use custom estimated states instead of measured outputs** in the General tab. Checking that option changes the label on this import to `x[k|k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k .

All candidate controllers must use the same state estimation option, default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables ($n_y = n_{ym} + \text{number of unmeasured outputs}$). You have the option to specify future reference values (previewing).

The `ref` signal must be size N by n_y , where $N(1 \leq N \leq p)$ is the number of time steps for which you are specifying reference values and p is the prediction horizon. Each element must be a real number. The `ref` dimension must not change from one control instant to the next.

When $N=1$, you cannot preview. To specify future reference values, choose N such that $1 < N \leq p$ to enable previewing. Doing so usually improves performance via `feedforward` information. The first row specifies the n_y references for the first step in the prediction horizon (at the next control interval $k=1$), and so on for N steps. If $N < p$, the last row designates constant reference values to be used for the remaining $p - N$ steps.

For example, suppose $n_y=2$ and $p=6$. At a given control instant, the signal connected to the controller's `ref` import is

```
[2 5 ← k=1  
2 6 ← k=2  
2 7 ← k=3  
2 8] ← k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step ($k=1$) are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step ($k=4$) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

Required Outports

Manipulated Variables

The **mv** outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its **mv** outport by solving a quadratic program at each control instant.

Additional Imports (General tab)

Measured disturbance

Add an import (**md**) to which you can connect a measured disturbance signal.

Your measured disturbance signal (**MD**) must be size $N \times n_{md}$, where $n_{md} \geq 1$ is the number of measured disturbances defined in the active Model Predictive Controller and N ($1 \leq N \leq p+1$) is the number of time steps for which the **MD** is known. Each element must be a real, double-precision number. The signal dimensions must not change from one control instant to the next.

If $N = 1$, you cannot preview. At each control instant, the **MD** signal must contain the most recent measurements at the current time $k = 0$ (as a row vector, length n_{md}). The controller assumes that the **MDs** remain constant at their current values for the entire prediction horizon.

If you are able to predict future **MD** values, choose N such that $1 < N \leq p+1$ to enable previewing. Doing so usually improves performance via **feedforward**. In this case, the first row must contain the n_{md} current values at $k=0$, and the remaining rows designate variations over the next $N-1$ control instants. If $N < p+1$, the last row designates constant **MD** values to be used for the remaining $p+1-N$ steps of the prediction horizon.

For example, suppose $n_{md} = 2$ and $p = 6$. At a given control instant, the signal connected to the controller's `md` import is:

```
[2 5 ← k=0  
2 6 ← k=1  
2 7 ← k=2  
2 8] ← k=3
```

This signal informs the controller that:

- The current MDs are 2 and 5 at $k=0$.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the step 3 ($k=3$) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

`mpcpreview` shows how to use MD previewing in a specific case.

For calculation details, see “MPC Modeling” and “QP Matrices”.

Externally Supplied MV signals

Add an import (`ext.mv`) to which you connect the actual manipulated variables (MV) used in the plant. All candidate controllers use this when updating their controller state estimates.

For additional discussion and examples, see the `MPC Controller` block documentation.

Additional Outputs (General tab)

You may configure a number of optional output signals. At each sampling instant, the active controller determines their values. The following describes each briefly. For more details, see the `MPC Controller` block documentation.

Optimal cost

Add an outport (`cost`) that provides the optimal quadratic programming objective function value at the current time (a nonnegative scalar). If the controller is performing well and no constraints have been violated, the value should be small. If the optimization problem is infeasible, however, the value is meaningless. (See `qp.status`.)

Optimal control sequence

Add an outport (`mv.seq`) that provides the active controller's computed optimal MV sequence for the entire prediction horizon from $k=0$ to $k = p - 1$. If n_u is the number of MVs and p is the length of the prediction horizon, this signal is a p by n_u matrix. The first row represents $k=0$ and duplicates the block's MV outport.

Optimization status

Add an outport (`qp.status`) that allows you to monitor the status of the active controller's QP solver.

If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution at this time interval.

The QP solver may fail to find an optimal solution for the following reasons:

- `qp.status = 0` — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object.
- `qp.status = -1` — The QP solver detects an infeasible QP problem. See Monitoring Optimization Status to Detect Controller Failures for an example where a large, sustained disturbance drives the OV outside its specified bounds.
- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem.

For all the previous three failure modes, the `Multiple MPC Controllers` block holds its `MV` output at the most recent successful solution. In a real-time application, you can use status indicator to set an alarm or take other special action.

Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the active controller's state estimates, $x[k|k]$, at each control instant. These include the plant, disturbance and noise model states.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

Add the `x[k|k]` import for custom state estimation as described in “Required Imports” on page 2-21. All candidate controllers must use the same state estimation option,

default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Constraints (Online Features tab)

At each control instant, the optional features described below apply to the active controller.

Plant input and output limits

Add imports (`umin`, `umax`, `ymin`, `ymax`) that you can connect to run-time constraint signals. If this check box is not selected, the block uses the constant constraint values stored within the active controller.

An unconnected import is treated as an unbounded signal. The corresponding variable in the `mpc` object must be unbounded.

For connected imports, the following rules apply:

- All connected signals must be finite. Simulink does not support infinite signals.
- If a variable is unconstrained in the controller object, the connected value is ignored.

Weights (Online Features tab)

The optional inputs described below function as controller “tuning knobs.” By default (or when a signal is unconnected), the active controller’s stored tuning weights apply.

When using these online tuning features, you should usually prevent an unexpected change in the active controller. Otherwise, settings intended for a particular candidate controller may instead retune another.

Weights on plant outputs

Add an import (`y.wt`) for a vector signal containing a nonnegative weight for each controlled output variable (OV). This signal overrides the `MPCobj.Weights.OV` property of the active controller, which establishes the relative importance of OV reference tracking.

If you do not connect a signal to the `y.wt` import, the block uses the OV weights specified in the active controller, and these values remain constant.

Weights on manipulated variables

Add an import (`u.wt`), whose input is a vector signal defining nu nonnegative weights, where nu is the number of manipulated variables (MVs). The input overrides the `MPCObj.Weights.MV` property of the active controller, which establishes the relative importance of MV target tracking.

If you do not connect a signal to the `u.wt` import, the block uses the `Weights.MV` weights property specified in the active controller, and these values remain constant.

Weights on manipulated variable changes

Add an import (`du.wt`), for a vector signal defining nu nonnegative weights, where nu is the number of manipulated variables (MVs). The input overrides the `MPCObj.Weights.MVrate` property of the active controller, which establishes the relative importance of MV changes.

If you do not connect a signal to the `du.wt` import, the block uses the `Weights.MVrate` property specified in the active controller, and these values remain constant.

Weight on overall constraint softening

Add an import (`ECR.wt`), for a scalar nonnegative signal that overrides the active controller's `MPCObj.Weights.ECR` property. This import has no effect unless the active controller defines soft constraints whose associated ECR values are nonzero.

MV Targets (Online Features tab)

Targets for manipulated variables

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` import to which you can connect the target signal (dimension n_u , where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Others tab

Block data type

Specify the block data type of the manipulated variables as one of the following:

- **double** — Double-precision floating point (default).
- **single** — Single-precision floating point.

Specify the output data type as **single** if you are implementing the **MPC Controller** block on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see “[Simulation and Code Generation Using Simulink Coder](#)”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & Ports** **Port Data Types**.

Inherit sample time

Use the sample time inherited from the parent subsystem as the **Multiple MPC Controllers** block’s sample time.

Inheriting the sample time allows you to conditionally execute the **Multiple MPC Controllers** block inside the **Function-Call Subsystem** or **Triggered Subsystem** blocks. For an example, see [Using MPC Controller Block Inside Function-Call and Triggered Subsystems](#).

Note: When you place an MPC controller inside a **Function-Call Subsystem** or **Triggered Subsystem** block, you must execute the subsystem at the controller’s design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see “[View Sample Time Information](#)”.

See Also

`mpc` | **MPC Controller** | `mpcmove` | `mpcstate`

Related Examples

- Scheduling Controllers for a Plant with Multiple Operating Points
- Chemical Reactor with Multiple Operating Points
- “[Simulation and Code Generation Using Simulink Coder](#)”

- “Simulation and Structured Text Generation Using PLC Coder”

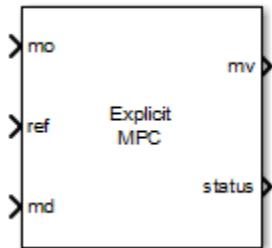
Explicit MPC Controller

Design and simulate explicit model predictive controller

Library

MPC Simulink Library

Description



Like the `MPC Controller` block, the `Explicit MPC Controller` block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

The key difference is that the `Explicit MPC Controller` block uses a table-lookup control law rather than solving a quadratic program during each control interval. The reduced online computational effort is advantageous in applications requiring a short control interval. The primary trade-off is a heavier offline computational effort needed to determine the control law and a larger memory footprint to store it. The combinatorial character of this computation restricts its use to applications with relatively few input, output, and state variables, a short prediction horizon, and few output constraints, if any.

The `Explicit MPC Controller` block also has fewer optional features than the other blocks in the MPC Simulink Library. In particular, it does not support the following:

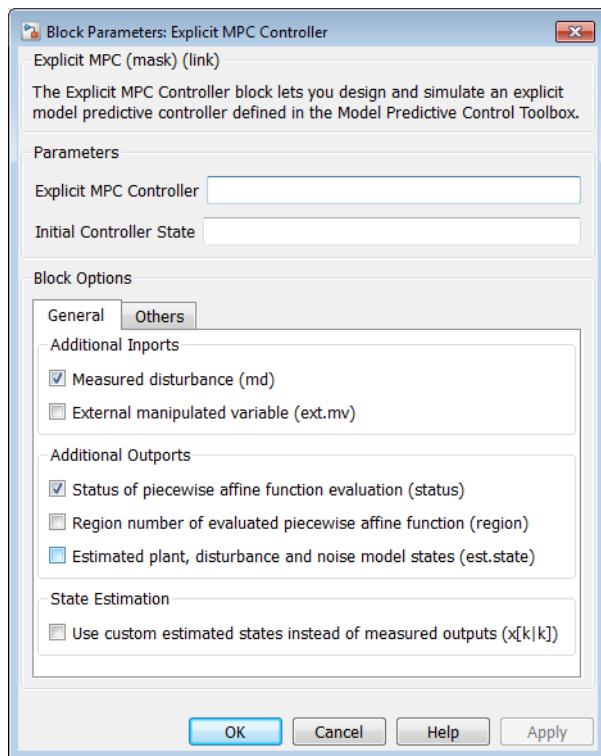
- Online tuning (i.e., penalty weight adjustments)
- Online constraint adjustments
- Manipulated variable target adjustments
- Reference and/or measured disturbance previewing
- Cost function outport
- Optimal control sequence outport

The block does support the following options common to most MPC Library blocks:

- Custom state estimation (default state estimation uses a static Kalman filter)
- Outport for state estimation results
- External manipulated variable feedback signal import
- Single-precision block data (default is double precision)
- Inherited sample time

It also provides two optional status outports unique to this block.

Dialog Box



The **Explicit MPC Controller** block has the following parameter groupings:

- “Parameters” on page 2-33
- “Required Imports” on page 2-33
- “Required Outports” on page 2-33
- “Additional Imports (General Tab)” on page 2-34
- “Additional Outports (General tab)” on page 2-34
- “Others tab” on page 2-35

Parameters

Explicit MPC Controller

An “Explicit MPC Controller Object” on page 3-18 object containing the control law to be used. It must exist in the workspace. Use the `generateExplicitMPC` command to create this object.

Initial Controller State

An optional `mpcstate` object specifying the initial controller state. By default the block uses the controller object’s `Model.Nominal` property.

Required Imports

Measured output or State estimate

If your controller uses default state estimation, this import is labeled `mo`. Connect the measured plant output variables.

If your controller uses custom state estimation, check **Use custom estimated states instead of measured outputs** in the General tab. Checking that option changes the label on this import to `x[k | k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k .

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables. Reference previewing is not supported. The block assumes each reference value is constant over the prediction horizon.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport at each control instant using the control law contained in the Explicit MPC controller object. If the control law evaluation fails, this signal is unchanged, i.e., held at the previous successful result.

Additional Imports (General Tab)

Measured disturbance

Add an import (`md`) to which you can connect a vector signal containing n_{md} elements, where n_{md} is the number of measured disturbances.

Measured disturbance previewing is not supported. The block assumes that each measured disturbance value is constant over the prediction horizon.

External manipulated variable

Add an import (`ext.mv`) to which you can connect a vector signal containing the n_u actual manipulated variables (MVs) used in the plant. The block uses this when updating its controller state estimates. Using this import improves state estimation accuracy when the MVs used in the plant differ from those calculated by the block, e.g., due to signal saturation or an override condition.

Controller state estimation assumes that the MV is piecewise constant. At time t_k , the `ext.mv` value must be the effective MV between times t_{k-1} and t_k . For example, if the MV is actually varying over this interval, you might supply the time-averaged value evaluated at time t_k .

If the external MV option is not selected or its import is unconnected, the block assumes that its MV output is used in the plant without modification.

For additional discussion and examples, see the corresponding section of the **MPC Controller** block reference page.

Note There is direct feed through from the `ext.mv` import to the `mv` outport. Thus, use of this option may cause an algebraic loop in the Simulink diagram. You might need to insert a **Memory** or **Unit Delay** block to prevent such algebraic loops.

Additional Outports (General tab)

Status of piecewise affine function evaluation

Add an outport (`status`) that indicates whether the latest explicit MPC control-law evaluation succeeded. The outport provides a scalar signal that has one of the following values:

- 1 — Successful explicit control law evaluation,
- 0 — Failure: one or more of the control law's parameters was out of range.
- -1 — Undefined: control law parameters were within the valid range but an extrapolation was necessary.

Region number of evaluated piecewise affine function

Add an outport (`region`) providing the index of the polyhedral region used in the latest explicit control law evaluation (a scalar). If the control law evaluation fails, the signal at this outport equals zero.

Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the controller state estimates, $x[k|k]$, at each control instant. These include the plant, disturbance and noise model states.

Others tab

Block data type

Specify the block data type as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

Specify the output data type as `single` if you are implementing the **MPC Controller** block on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see “[Simulation and Code Generation Using Simulink Coder](#)”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & Ports** **Port Data Types**.

Inherit sample time

Use the sample time inherited from the parent subsystem as the **MPC Controller** block's sample time.

Inheriting the sample time allows you to conditionally execute the **MPC Controller** block inside the **Function-Call Subsystem** or **Triggered Subsystem** blocks.

For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note: When you place an **Explicit MPC Controller** block inside a **Function-Call Subsystem** or **Triggered Subsystem** block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see “View Sample Time Information”.

See Also

`generateExplicitMPC` | `mpc` | **MPC Controller** | `mpcmoveExplicit` | `mpcstate`

Related Examples

- “Explicit MPC Control of a Single-Input-Single-Output Plant”
- “Explicit MPC Control of an Aircraft with Unstable Poles”
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output”

More About

- “Explicit MPC”
- “Design Workflow for Explicit MPC”

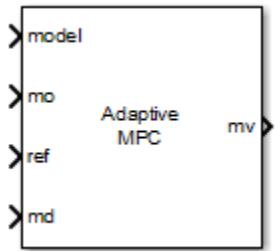
Adaptive MPC Controller

Design and simulate adaptive and time-varying model predictive controllers

Library

MPC Simulink Library

Description



Like the MPC Controller block, the Adaptive MPC Controller block uses the following input signals:

- Measured plant outputs (**mo**)
- Reference or setpoint (**ref**)
- Measured plant disturbance (**md**), if any

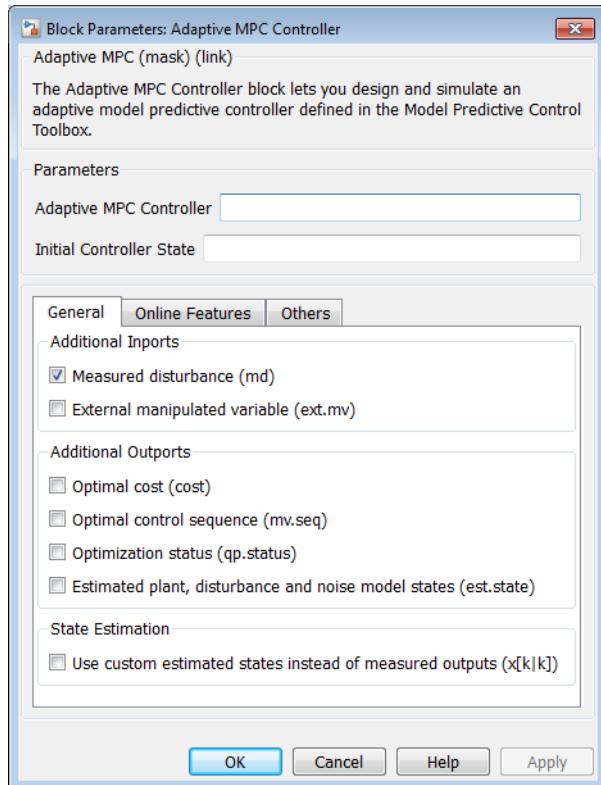
In addition, the required model input signal specifies the prediction model to be used when solving the quadratic program (QP) for the block output (**mv**), i.e., the optimal plant manipulated variables. The linear prediction model can change at each control interval in response to changes in the real plant at run-time. The prediction model can represent a single LTI plant used for all prediction steps (adaptive MPC mode) or an array of LTI plants for different prediction steps (time-varying MPC mode). Two common ways to modify this model are as follows:

- Given a nonlinear plant model, linearize it at the current operating point.
- Use plant data to estimate parameters in an empirical linear-time-varying model.

By default, the block estimates its prediction model states. Since the prediction model parameters change at run time, the static Kalman filter used in the **MPC Controller** block is inappropriate. Instead, the **Adaptive MPC Controller** block employs a linear-time-varying Kalman filter (LTVKF). See “Adaptive MPC” for details.

In all other ways the **Adaptive MPC Controller** block mimics the simpler **MPC Controller** block. As the adaptive version involves additional overhead, use the **MPC Controller** block unless you need to modify the prediction model. Another alternative is the **Multiple MPC Controllers** block, which allows you to use a finite, predetermined set of prediction models.

Dialog Box



The **Adaptive MPC Controller** block has the following parameter groupings:

- “Parameters” on page 2-39
- “Required Imports” on page 2-39
- “Required Outports” on page 2-41
- “Additional Imports (General tab)” on page 2-41
- “Additional Imports (General tab)” on page 2-41
- “State Estimation (General tab)” on page 2-41
- “Constraints (Online Features tab)” on page 2-42
- “MV Targets (Online Features tab)” on page 2-42
- “Others tab” on page 2-42

Parameters

Adaptive MPC Controller

A traditional (implicit) `mpc` controller object whose prediction model is to be modified at each control instant. By default, the block assumes all other controller object properties (e.g., tuning weights, constraints) are constant. You can override this assumption using the options in the **Online Features** tab.

The following restrictions apply to the `mpc` controller object:

- It must exist in your base workspace.
- Its prediction model must be an LTI discrete-time, state-space object with no delays. (Use the `absorbDelay` command to convert delays to discrete states.)

Initial Controller State

An optional `mpcstate` object specifying the initial controller state. If you leave this box blank, the block uses the nominal values defined in the controller object’s `Model.Nominal` property. The alternative is to create an `mpcstate` object in your workspace, initialize it to the desired state, and enter its name in the box.

Required Imports

Model

Connect a bus signal to the `model` import. This signal modifies the controller object `Model.Plant` and `Model.Nominal` properties at the beginning of each control interval.

The Adaptive MPC Controller requires `Model.Plant` to be an LTI discrete-time state-space object with no delays. The following command extracts the state-space matrices comprising such a model:

```
[A,B,C,D] = ssdata(MPCobj.Model.Plant)
```

The purpose of the `model` import is to replace these matrices with new ones having the same dimensions, and representing the same control interval. You must also retain the sequence in which the input, output, and state variables appear in `Model.Plant`.

When operating in:

- Adaptive MPC mode, the bus you connect to the `model` import must contain the following signals, each identified by the specified name:
 - `A` — n_x -by- n_x matrix signal, where n_x is the number of plant model states.
 - `B` — n_x -by- n_{utot} matrix signal, where n_{utot} is the total number of plant model inputs (i.e., manipulated variables, measured disturbances, and unmeasured disturbances).
 - `C` — n_y -by- n_x matrix signal, where n_y is the number of plant model outputs.
 - `D` — n_y -by- n_{utot} matrix signal.
 - `X` — Vector signal of length n_x , replacing the controller `Model.Nominal.X` property.
 - `Y` — Vector signal of length n_y , replacing the controller `Model.Nominal.Y` property.
 - `U` — Vector signal of length n_{utot} , replacing the controller `Model.Nominal.U` property.
 - `DX` — Vector signal of length n_x , replacing the controller `Model.Nominal.DX` property. It must be appropriate for use with a discrete-time model of the assumed control interval. For more information, see “Adaptive MPC”.
- Time-varying MPC mode, the bus you connect to the `model` import must contain the following 3-dimensional bus signals:
 - `A` — n_x -by- n_x -by- $(p+1)$ matrix signal
 - `B` — n_x -by- n_{utot} -by- $(p+1)$ matrix signal
 - `C` — n_y -by- n_x -by- $(p+1)$
 - `D` — n_y -by- n_{utot} -by- $(p+1)$ matrix signal

- X — n_x -by- $(p+1)$ matrix signal
- Y — n_y -by- $(p+1)$ matrix signal
- U — n_{utot} -by- $(p+1)$ matrix signal
- DX — n_x -by- $(p+1)$ matrix signal

Here, p is the controller prediction horizon. For each signal, specify $p+1$ values representing the model and nominal conditions at each step of the prediction horizon. For more information, see “Time-Varying MPC”.

One way to form the bus is to use a Simulink Bus Creator block.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport by solving a quadratic program at each control instant.

Additional Imports (General tab)

These optional imports are identical to the corresponding imports in the MPC Controller block. Please see the MPC Controller block reference page for information.

Additional Imports (General tab)

These optional outports are identical to the corresponding outports in the MPC Controller block. Please see the MPC Controller block reference page for information.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

Add the `x[k | k]` import for custom state estimation.

Constraints (Online Features tab)

These optional imports are identical to the corresponding imports in the MPC Controller block. Please see the MPC Controller block reference page for information.

MV Targets (Online Features tab)

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` import to which you can connect the target signal (dimension n_u , where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Others tab

These parameters are identical to the corresponding parameters in the MPC Controller block. Please see the MPC Controller block reference page for information.

See Also

`mpc` | MPC Controller | `mpcmoveAdaptive` | `mpcstate` | Multiple MPC Controllers

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization”
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation”
- “Time-Varying MPC Control of a Time-Varying Plant”

More About

- “Adaptive MPC”
- “Time-Varying MPC”

Object Reference

- “MPC Controller Object” on page 3-2
- “MPC Simulation Options Object” on page 3-13
- “MPC State Object” on page 3-16
- “Explicit MPC Controller Object” on page 3-18

MPC Controller Object

All of the parameters defining the traditional (implicit) MPC control law are stored in an MPC object, whose properties are listed in the following table.

MPC Controller Object

Property	Description
ManipulatedVariables (or MV or Manipulated or Input)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
OutputVariables (or OV or Controlled or Output)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
DisturbanceVariables (or DV or Disturbance)	Disturbance scale factors, names, and units
Weights	Weights used in computing the performance (cost) function
Model	Plant, input disturbance, and output noise models, and nominal conditions.
Ts	Controller sample time
Optimizer	Parameters controlling the QP solver
PredictionHorizon	Prediction horizon
ControlHorizon	Number of free control moves or vector of blocking moves
History	Creation time
Notes	Text or comments about the MPC controller object
UserData	Any additional data

ManipulatedVariables

ManipulatedVariables (or **MV** or **Manipulated** or **Input**) is an n_u -dimensional array of structures (n_u = number of manipulated variables), one per manipulated variable. Each structure has the fields described in the following table, where p denotes the

prediction horizon. Unless indicated otherwise, numerical values are in engineering units.

Manipulated Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this MV	1
Min	1 to p length vector of lower bounds on this MV	-Inf
Max	1 to p length vector of upper bounds on this MV	Inf
MinECR	1 to p length vector of nonnegative parameters specifying the Min bound softness (0 = hard).	0 (dimensionless)
MaxECR	1 to p length vector of nonnegative parameters specifying the Max bound softness (0 = hard).	0 (dimensionless)
Target	1 to p length vector of target values for this MV	nominal
RateMin	1 to p length vector of lower bounds on the interval-to-interval change for this MV	-Inf
RateMax	1 to p length vector of upper bounds on the interval-to-interval change for this MV	Inf
RateMinECR	1 to p length vector of nonnegative parameters specifying the RateMin bound softness (0 = hard).	0 (dimensionless)
RateMaxECR	1 to p length vector of nonnegative parameters specifying the RateMax bound softness (0 = hard).	0 (dimensionless)
Name	Read-only MV signal name (character string)	InputName of LTI plant model
Units	Read-only MV signal units (character string)	InputUnit of LTI plant model

Note Rates refer to the difference $\Delta u(k) = u(k) - u(k-1)$. Constraints and weights based on derivatives du/dt of continuous-time input signals must be properly reformulated for the discrete-time difference $\Delta u(k)$, using the approximation $du/dt \approx \Delta u(k)/T_s$.

OutputVariables

OutputVariables (or **OV** or **Controlled** or **Output**) is an n_y -dimensional array of structures (n_y = number of outputs), one per output signal. Each structure has the fields described in the following table. p denotes the prediction horizon. Unless specified otherwise, values are in engineering units.

Output Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this OV	1
Min	1 to p length vector of lower bounds on this OV	-Inf
Max	1 to p length vector of upper bounds on this OV	Inf
MinECR	1 to p length vector of nonnegative parameters specifying the Min bound softness (0 = hard).	1 (dimensionless)
MaxECR	1 to p length vector of nonnegative parameters specifying the Max bound softness (0 = hard).	1 (dimensionless)
Name	Read-only OV signal name (character string)	OutputName of LTI plant model
Units	Read-only OV signal units (character string)	OutputUnit of LTI plant model

In order to reject constant disturbances due, for instance, to gain nonlinearities, the default measured output disturbance model used in Model Predictive Control Toolbox software is integrated white noise (see “Output Disturbance Model”).

DisturbanceVariables

DisturbanceVariables (or DV or Disturbance) is an (n_v+n_d) -dimensional array of structures (n_v = number of measured input disturbances, n_d = number of unmeasured input disturbances). Each structure has the fields described in the following table.

Disturbance Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this DV	1
Name	Read-only DV signal name (character string)	InputName of LTI plant model
Units	Read-only DV signal units (character string)	InputUnit of LTI plant model

The order of the disturbance signals within the array DV is the following: the first n_v entries relate to measured input disturbances, the last n_d entries relate to unmeasured input disturbances.

Weights

Weights is the structure defining the QP weighting matrices. It contains four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see “Standard Cost Function”) or the alternative cost function (see “Alternative Cost Function”).

Standard Cost Function

The table below lists the content of the four structure fields. In the table, p denotes the prediction horizon, n_u the number of manipulated variables, and n_y the number of output variables.

For the MV, MVRate and OV weights, if you specify fewer than p rows, the last row repeats automatically to form a matrix containing p rows.

Weights for the Standard Cost Function

Field Name (Abbreviations)	Content	Default (dimensionless)
ManipulatedVariables (or MV or Manipulated or Input)	(1 to p)-by- n_u dimensional array of nonnegative MV weights	<code>zeros(1,nu)</code>

Field Name (Abbreviations)	Content	Default (dimensionless)
ManipulatedVariablesRate (or MVRate or ManipulatedRate or InputRate)	(1 to p)-by- n_u dimensional array of MV-increment weights	<code>0.1*ones(1,nu)</code>
OutputVariables (or OV or Controlled or Output)	(1 to p)-by- n_y dimensional array of OV weights	1 (The default for output weights is the following: if $n_u \geq n_y$, all outputs are weighted with unit weight; if $n_u < n_y$, n_u outputs default to 1, with preference given to measured outputs, and the rest default to 0.)
ECR	Scalar weight on the slack variable ε used for constraint softening	<code>1e5*(max weight)</code>

Note If all **MVRate** weights are strictly positive, the resulting QP problem is strictly convex. If some **MVRate** weights are zero, the QP Hessian might be positive semidefinite. In order to keep the QP problem strictly convex, when the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12} , the quantity `10*sqrt(eps)` is added to each diagonal term. See “Cost Function”.

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To do so, define the fields **ManipulatedVariables**, **ManipulatedVariablesRate**, and **OutputVariables** as cell arrays, each containing a single positive-semi-definite matrix of the appropriate size. Specifically, **OutputVariables** must be a cell array containing the n_y -by- n_y Q matrix, **ManipulatedVariables** must be a cell array containing the n_u -by- n_u R_u matrix, and **ManipulatedVariablesRate** must be a cell array containing the n_u -by- n_u $R_{\Delta u}$ matrix (see “Alternative Cost Function” and the `mpcweightsdemo` example). You can use diagonal weight matrices for one or more of these fields. If you omit a field, the MPC controller uses the defaults shown in the table above.

For example, you can specify off-diagonal weights, as follows

```
MPCCobj.Weights.OutputVariables = {Q};
```

```
MPCobj.Weights.ManipulatedVariables = {Ru};  
MPCobj.Weights.ManipulatedVariablesRate = {Rdu};
```

where $Q = Q$, $R_u = R_u$, and $R_{du} = R_{\Delta u}$ are positive semidefinite matrices.

Note You cannot specify non-diagonal weights that vary at each prediction horizon step. The same Q , R_u , and R_{du} weights apply at each step.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in “Controller State Estimation”. It is a 1-D structure containing the following fields.

Models Used by MPC

Field Name	Content	Default
<code>Plant</code>	LTI model or identified linear model of the plant	No default
<code>Disturbance</code>	LTI model describing expected unmeasured input disturbances	[] (By default, input disturbances are expected to be integrated white noise. To model the signal, an integrator with dimensionless unity gain is added for each unmeasured input disturbance, unless the addition causes the controller to lose state observability. In that case, the disturbance is expected to be white noise, and so, a dimensionless unity gain is added to that channel instead.)
<code>Noise</code>	LTI model describing expected noise for output measurements	[] (By default, measurement noise is expected to be white noise with unit variance. To model the signal, a dimensionless unity gain is added for each measured channel.)
<code>Nominal</code>	Structure containing the state, input, and output	The default values of the fields are shown in the following table:

Field Name	Content	Default		
		Field	Description	Default
	values where <code>Model.Plant</code> is linearized	X	Plant state at operating point	[]
		U	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	[]
		Y	Plant output at operating point	[]
		DX	For continuous-time models, DX is the state derivative at operating point: $DX=f(X,U)$. For discrete-time models, $DX=x(k+1)-x(k)=f(X,U)-X$.	[]

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See “MPC Modeling”.

Specify input and output signal types via the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, use the `setmpcsignals` command. Valid signal types are listed in the following tables.

Input Groups in Plant Model

Name (Abbreviations)	Value
<code>ManipulatedVariables</code> (or MV or Manipulated or Input)	Indices of manipulated variables in <code>Model.Plant</code>
<code>MeasuredDisturbances</code> (or MD or Measured)	Indices of measured disturbances in <code>Model.Plant</code>
<code>UnmeasuredDisturbances</code> (or UD or Unmeasured)	Indices of unmeasured disturbances in <code>Model.Plant</code>

Output Groups in Plant Model

Name (Abbreviations)	Value
MeasuredOutputs (or MO or Measured)	Indices of measured outputs in <code>Model.Plant</code>
UnmeasuredOutputs (or UO or Unmeasured)	Indices of unmeasured outputs in <code>Model.Plant</code>

By default, all `Model.Plant` inputs are manipulated variables, and all outputs are measured.

The structure `Nominal` contains the values (in engineering units) for states, inputs, outputs, and state derivatives/differences at the operating point where `Model.Plant` applies. This point is typically a linearization point. The fields are reported in the following table (see also “MPC Modeling”).

Nominal Values at Operating Point

Field	Description	Default
X	Plant state at operating point	[]
U	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	[]
Y	Plant output at operating point	[]
DX	For continuous-time models, DX is the state derivative at operating point: $DX=f(X,U)$. For discrete-time models, $DX=x(k+1)-x(k)=f(X,U)-X$.	[]

Ts

Sample time of the MPC controller. By default, if `Model.Plant` is a discrete-time model, `Ts = Model.Plant.ts`. For continuous-time plant models, you must specify a controller `Ts`. Its measurement unit is inherited from `Model.Plant.TimeUnit`.

Optimizer

Parameters for the QP optimization. `Optimizer` is a structure with the following fields:

Optimizer Properties

Field	Description	Default
MaxIter	<p>Maximum number of iterations allowed in the QP solver, specified as one of the following:</p> <ul style="list-style-type: none"> • Default — The MPC controller automatically computes the maximum number of QP solver iterations as: $\text{MaxIter} = 4(p \cdot n_{cy} + c(n_{cu} + n_{cr} + n_u)) + n_{sv}$ <p>where</p> <ul style="list-style-type: none"> • p is the prediction horizon. • c is the control horizon. • n_{cy} is the number of OV constraints. • n_{cu} is the number of MV constraints. • n_{cr} is the number of MV rate constraints. • n_u is the number of MVs. • n_{sv} is the number of slack variables. • Positive integer — The QP solver stops after MaxIter iterations. 	Default
MinOutputECR	Minimum value allowed for OutputMinECR and OutputMaxECR , specified as a nonnegative scalar. A value of 0 indicates that hard output constraints are allowed. If either of the OutputVariables.MinECR or OutputVariables.MaxECR properties of an MPC controller are less than MinOutputECR , a warning is displayed and the value is raised to MinOutputECR during computation.	0
CustomSolver	Flag indicating whether to use a custom QP solver, specified as a logical value. If	false

Field	Description	Default
	CustomSolver is true, the user must provide an <code>mpcCustomSolver</code> function on the MATLAB path. For information on how to define the <code>mpcCustomSolver</code> function, see “Custom QP Solver”.	

Note: The default `MaxIter` value can be very large for some controller configurations, such as those with large prediction and control horizons. When simulating such controllers, if the QP solver cannot find a feasible solution, the simulation can appear to stop responding, since the solver continues searching for `MaxIter` iterations.

PredictionHorizon

`PredictionHorizon` is the integer number of prediction horizon steps. The control interval, `Ts`, determines the duration of each step. The default value is 10 + maximum intervals of delay (if any).

ControlHorizon

`ControlHorizon` is either a number of free control moves, or a vector of blocking moves (see “Optimization Variables”). The default value is 2.

History

`History` stores the time the MPC controller was created (read only).

Notes

`Notes` stores text or comments as a cell array of strings.

UserData

Any additional data stored within the MPC controller object.

Construction and Initialization

To minimize computational overhead, Model Predictive Controller creation occurs in two phases. The first happens at *construction* when you invoke the `mpc` command, or when you change a controller property. Construction involves simple validity and consistency checks, such as signal dimensions and non-negativity of weights.

The second phase is *initialization*, which occurs when you use the object for the first time in a simulation or analytical procedure. Initialization computes all constant properties required for efficient numerical performance, such as matrices defining the optimal control problem and state estimator gains. Additional, diagnostic checks occur during initialization, such as verification that the controller states are observable.

By default, both phases display informative messages in the command window. You can turn these messages on or off using the `mpcverbosity` command.

MPC Simulation Options Object

The `mpcsimopt` object type contains various simulation options for simulating an MPC controller with the command `sim`. Its properties are listed in the following table.

MPC Simulation Options Properties

Property	Description
<code>PlantInitialState</code>	Initial state vector of the plant model generating the data.
<code>ControllerInitialState</code>	Initial condition of the MPC controller. This must be a valid <code>mpcstate</code> object. Note Nonzero values of <code>ControllerInitialState.LastMove</code> are only meaningful if there are constraints on the increments of the manipulated variables.
<code>UnmeasuredDisturbance</code>	Unmeasured disturbance signal entering the plant. An array with as many rows as simulation steps, and as many columns as unmeasured disturbances. Default: 0
<code>InputNoise</code>	Noise on manipulated variables. An array with as many rows as simulation steps, and as many columns as manipulated variables. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
<code>OutputNoise</code>	Noise on measured outputs. An array with as many rows as simulation steps, and as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
<code>RefLookAhead</code>	Preview on reference signal (<code>on</code> or <code>off</code>). Default: <code>off</code>

Property	Description
MDLookAhead	Preview on measured disturbance signal (<code>on</code> or <code>off</code>).
Constraints	Use MPC constraints (<code>on</code> or <code>off</code>). Default: <code>on</code>
Model	<p>Model used in simulation for generating the data.</p> <p>This property is useful for simulating the MPC controller under model mismatch. The LTI object specified in <code>Model</code> can be either a replacement for <code>Model.Plant</code>, or a structure with fields <code>Plant</code> and <code>Nominal</code>. By default, <code>Model</code> is equal to <code>MPCobj.Model</code> (no model mismatch). If <code>Model</code> is specified, then <code>PlantInitialState</code> refers to the initial state of <code>Model.Plant</code> and is defaulted to <code>Model.Nominal.x</code>.</p> <p>If <code>Model.Nominal</code> is empty, <code>Model.Nominal.U</code> and <code>Model.Nominal.Y</code> are inherited from <code>MPCobj.Model.Nominal</code>. <code>Model.Nominal.X/DX</code> is only inherited if both plants are state-space objects with the same state dimension.</p>
StatusBar	Display the wait bar (<code>on</code> or <code>off</code>). Default: <code>off</code>
MVSignal	<p>Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action).</p> <p>An array with as many rows as simulation steps, and as many columns as manipulated variables. Default: 0</p>
OpenLoop	Perform open-loop simulation (<code>on</code> or <code>off</code>). Default: <code>off</code>

The property `Model` is useful for simulating an MPC controller with “model mismatch”, i.e., when the controller’s prediction model only approximates the true plant behavior (inevitable in practice).

By default, `Model` is equal to `MPCObj.Model` (no model mismatch). If `Model` is specified, then `PlantInitialState` refers to the initial state of `Model.Plant` and defaults to `Model.Nominal.x`.

If `Model.Nominal` is empty, `Model.Nominal.U` and `Model.Nominal.Y` are inherited from `MPCObj.Model.Nominal`. `Model.Nominal.X/DX` is only inherited if both plants are state-space objects with the same state dimension.

MPC State Object

The `mpcstate` object type contains the state of an MPC controller. Create the MPC state object using `mpcstate`. Its properties are as follows.

Property	Description
Plant	<p>Vector of state estimates for the controller's plant model. Values are in engineering units and are absolute, i.e., they include state offsets.</p> <p>If the controller's plant model includes delays, the <code>Plant</code> field of the MPC state object includes states that model the delays. Therefore <code>length(Plant) > order of undelayed controller plant model.</code></p> <p>Default: controller's <code>Model.Nominal.X</code> property.</p>
Disturbance	<p>Vector of unmeasured disturbance model state estimates. This comprises the states of the input disturbance model followed by the states of the output disturbances model.</p> <p>Disturbance models may be created by default. Use the <code>getindist</code> and <code>getoutdist</code> commands to view the two disturbance model structures.</p> <p>Default: zero, or empty if there are no disturbance model states.</p>
Noise	<p>Vector of output measurement noise model state estimates.</p> <p>Default: zero, or empty if there are no noise model states.</p>
LastMove	<p>Vector of manipulated variables used in the previous control interval, $u(k-1)$. Values are absolute, i.e., they include manipulated variable offsets.</p> <p>Default: nominal values of the manipulated variables.</p>
Covariance	<p>n-by-n symmetrical covariance matrix for the controller state estimates, where n is the dimension of the extended controller state, i.e., the sum of the number states contained in the <code>Plant</code>, <code>Disturbance</code>, and <code>Noise</code> fields.</p>

Property	Description
	Default: If the controller is employing default state estimation the default is the steady-state covariance computed according to the assumptions in “Controller State Estimation”. See also the description of the P matrix in the Control System Toolbox <code>kalmd</code> command. If the controller is employing custom state estimation, this field is empty (not used).

Explicit MPC Controller Object

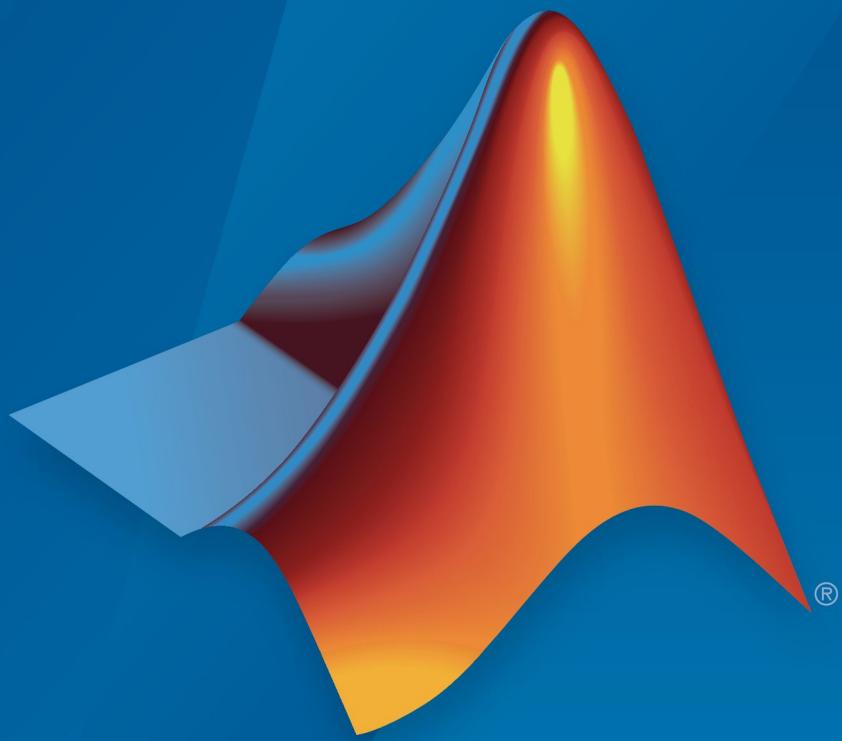
An explicit MPC object contains the explicit control law equivalent to the traditional (implicit) MPC controller object from which it derives. Use an explicit MPC controller to implement MPC in applications requiring very rapid computations, i.e., a short control interval. Use the `generateExplicitMPC` command to create the object. Its properties are as follows:

Properties

Property	Description
MPC	Traditional (implicit) controller object used to generate the explicit MPC controller. You create this MPC controller using the <code>mpc</code> command. It is the first argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See “MPC Controller Object” on page 3-2 or type <code>mpcprops</code> for details regarding the properties of the MPC controller.
Range	1-D structure containing the parameter bounds used to generate the explicit MPC controller. These determine the resulting controller’s valid operating range. This property is automatically populated by the <code>range</code> input argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See <code>generateExplicitRange</code> for details about this structure.
OptimizationOptions	1-D structure containing user-modifiable options used to generate the explicit MPC controller. This property is automatically populated by the <code>opt</code> argument to <code>generateExplicitMPC</code> when you create the explicit MPC controller. See <code>generateExplicitOptions</code> for details about this structure.

Property	Description
PiecewiseAffineSolution	n_r -dimensional structure, where n_r is the number of piecewise affine (PWA) regions required to represent the control law. The i th element contains the details needed to compute the optimal manipulated variables when the solution lies within the i th region. See “Implementation”.
IsSimplified	Logical switch indicating whether the explicit control law has been modified using the simplify command such that the explicit control law approximates the base (implicit) MPC controller. If the control law has not been modified, the explicit controller should reproduce the base controller’s behavior exactly, provided both operate within the bounds described by the Range property.

Model Predictive Control Toolbox™ Release Notes



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

Model Predictive Control Toolbox™ Release Notes

© COPYRIGHT 2005–2016 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to 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.

R2016a

Adaptive MPC with Time-Varying Prediction Models: Simulate adaptive MPC controllers with prediction models that change over the prediction horizon	1-2
mpcmoveCodeGeneration Command: Generate C code for computing optimal manipulated variable control moves	1-2
Custom QP Solver: Simulate model predictive controllers with a QP solver of your choice	1-2

R2015b

Redesigned MPC Designer App: Design model predictive controllers in MATLAB and Simulink using improved interactive workflows	2-2
MATLAB Script Generation from MPC Designer App: Automatically script model predictive controller design tasks	2-2
Simulink Model Generation from MPC Designer App: Automatically create a Simulink model with an MPC controller and plant model	2-3
mpcqpsolver Command: Develop and generate code for custom model predictive controllers using KWIK quadratic programming solver	2-3

Review model predictive controller design using MPC Designer app	2-3
Comparison of responses for multiple model predictive controllers in the same plot using MPC Designer app	2-3
Interactive tuning of model predictive controller performance objectives	2-4
mpctool command renamed to mpcDesigner	2-5
Functionality being removed or changed	2-5

R2015a

OutputVariables Integrator property of MPC controller being removed	3-2
setoutdist command remove syntax being removed	3-2
Functionality being removed or changed	3-4

R2014b

Explicit MPC control for applications with fast sample times using precomputed solutions	4-2
Adaptive MPC control through run-time changes to internal plant model	4-2
ScaleFactor property for MPC controllers, for making weight tuning independent of the engineering units of input and output variables	4-3

Option to use custom state estimation or measured state values instead of the built-in state estimation in MPC controllers	4-3
Option to specify manipulated variable target	4-3
Run-time weight tuning on manipulated variables	4-4
Run-time weight tuning and performance monitoring in Multiple MPC Controllers block	4-4
getEstimator and setEstimator commands to obtain and change state estimation parameters	4-4
Definition of external MV signal changed	4-5
Unconnected input and output limits imports default changed to match mpc object	4-5

R2014a

IEC 61131-3 Structured Text generation from MPC Controller and Multiple MPC Controllers blocks using Simulink PLC Coder	5-2
Reduced RAM usage for C code generated for MPC Controller and Multiple MPC Controllers blocks	5-2
Estimate of data memory size used by deployed MPC controller at run time	5-2

R2013b

Controller design for plant and disturbance models with internal delays	6-2
--	------------

Single-precision simulation and code generation using MPC Controller and Multiple MPC Controllers blocks	6-2
Conditional execution of MPC Controller and Multiple MPC Controllers blocks using Function-Call Subsystem and Triggered Subsystem blocks	6-3

R2013a

Bug Fixes

R2012b

Bug Fixes

R2012a

Run-Time Preview of Reference and Measured Disturbance Signals with MPC Controller Block	9-2
---	-----

R2011b

C Code Generation Improvements for All Targets with MPC Controller Block	10-2
Faster QP Solver Algorithm for Improving MPC Controller Performance	10-2

Run-Time Weight Tuning and Constraint Softening for MPC Controller	10-2
Run-Time Monitoring of MPC Controller Performance to Detect When an Optimal Solution Cannot Be Found	10-3
review Command for Diagnosing Issues with MPC Controller Parameters That Could Lead to Run-Time Failures	10-3
mpcmove Returns Aligned Time Horizons for Optimal Control, Predicted Output and Estimated State	10-3
Functionality Being Removed or Changed	10-4

R2011a

Support for Custom Constraints on MPC Controller Inputs and Outputs	11-2
Ability to Specify Terminal Constraints and Weights on MPC Controller	11-2
Ability to Access Optimal Cost and Optimal Control Sequence	11-2

R2010b

No New Features or Changes

R2010a

New Ability to Analyze SISO Generalized Predictive Controllers (GPC)	13-2
--	------

R2009b

Bug Fixes

R2009a

New Sensitivity Analysis to Determine Effect of Weights on Tuning MPC Controllers	15-2
---	------

R2008b

New Multiple MPC Controllers Block in the Model Predictive Control Toolbox Simulink Library	16-2
---	------

Tested Code Generation Support for Real-Time Workshop Target Systems	16-2
--	------

Ability to Design Controllers with Time-Varying Weights and Constraints Using the GUI	16-3
---	------

R2008a

No New Features or Changes

R2007b

New Option for Specifying Time-Varying Constraints	18-2
Ability to Specify Nondiagonal Q and R Weight Matrices in the Cost Function	18-2

R2007a

Bug Fixes

R2006b

No New Features or Changes

R2006a

Bumpless Transfer Added to MPC Block	21-2
New Bumpless Transfer Demo	21-2

R14SP3

No New Features or Changes

R14SP2

No New Features or Changes

R2016a

Version: 5.2

New Features

Bug Fixes

Adaptive MPC with Time-Varying Prediction Models: Simulate adaptive MPC controllers with prediction models that change over the prediction horizon

You can now specify prediction models and nominal conditions that change over the prediction horizon when using adaptive MPC. Use these options if you can predict how the plant and nominal conditions vary in the future.

To vary the prediction model, specify the `Plant` input argument of `mpcmovAdaptive` as an array of up to $p+1$ delay-free, discrete-time, state-space models, where p is the prediction horizon of your MPC controller. To vary the nominal conditions, specify the `Nominal` input argument of `mpcmovAdaptive` as an array of up to $p+1$ nominal condition structures.

For more information, see “Time-Varying MPC” and “Time-Varying MPC Control of a Time-Varying Plant”.

mpcmovCodeGeneration Command: Generate C code for computing optimal manipulated variable control moves

You can now generate C code for computing optimal manipulated variable control moves for any valid implicit or explicit MPC controller using the new `mpcmovCodeGeneration` command.

Use the new `getCodeGenerationData` command to create the input data structures for `mpcmovCodeGeneration`.

For an example of how to generate C code for computing optimal MPC control moves, see “Generate Code To Compute Optimal MPC Moves in MATLAB”.

Custom QP Solver: Simulate model predictive controllers with a QP solver of your choice

You can now define a custom QP solver for your MPC controller. To do so, you must provide a custom `mpcCustomSolver.m` file on the MATLAB® path that finds an optimal solution to the general form QP problem. For more information on the MPC QP problem and how to specify a custom solver, see “Custom QP Solver”.

For an example on how to use a custom QP solver, see “Simulate MPC Controller with a Custom QP Solver”.

R2015b

Version: 5.1

New Features

Bug Fixes

Compatibility Considerations

Redesigned MPC Designer App: Design model predictive controllers in MATLAB and Simulink using improved interactive workflows

The redesigned MPC Designer app streamlines MATLAB and Simulink® workflows for designing model predictive controllers. You can now:

- Generate MATLAB scripts for MPC controller design tasks. See “MATLAB Script Generation from MPC Designer App: Automatically script model predictive controller design tasks” on page 2-2.
- Generate a Simulink model with an MPC controller and plant model. See “Simulink Model Generation from MPC Designer App: Automatically create a Simulink model with an MPC controller and plant model” on page 2-3.
- Compare responses for multiple MPC controllers in the same plot. See “Comparison of responses for multiple model predictive controllers in the same plot using MPC Designer app” on page 2-3.
- Review MPC controllers for design and run-time stability issues. See “Review model predictive controller design using MPC Designer app” on page 2-3.
- Tune controller performance objectives using interactive sliders. See “Interactive tuning of model predictive controller performance objectives” on page 2-4.

To open the MPC Designer app, enter the following:

```
mpcDesigner
```

For examples of using the app from MATLAB and Simulink, see [Design Controller Using MPC Designer](#) and [Design MPC Controller in Simulink](#).

MATLAB Script Generation from MPC Designer App: Automatically script model predictive controller design tasks

You can now generate MATLAB scripts for creating and simulating model predictive controllers designed in the MPC Designer app. Generated MATLAB scripts are useful when you want to programmatically reproduce designs that you obtained interactively.

For more information, see [Generate MATLAB Code from MPC Designer](#).

Simulink Model Generation from MPC Designer App: Automatically create a Simulink model with an MPC controller and plant model

You can now generate a Simulink model that uses the current model predictive controller to control its internal plant model. You can then use the generated model to validate your controller design and generate code for real-time control applications.

For more information, see [Generate Simulink Model from MPC Designer](#).

mpcqpsolver Command: Develop and generate code for custom model predictive controllers using KWIK quadratic programming solver

You can use the new `mpcqpsolver` command to develop custom model predictive controllers. Use the new `mpcqpsolverOptions` command to specify additional solver options.

You can also use `mpcspSolver` as a general purpose QP solver that supports code generation.

For more information, see [Solve Custom MPC Quadratic Programming Problem](#) and [Generate Code](#).

Review model predictive controller design using MPC Designer app

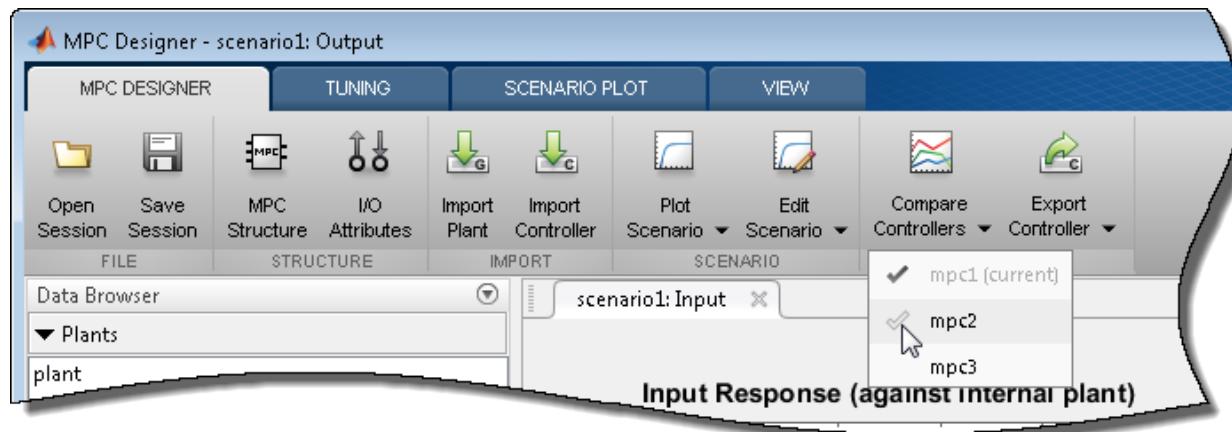
You can now review your model predictive controllers for potential run-time stability and numerical problems from within the MPC Designer app. To review the design of your

current controller, on the **Tuning** tab, click **Review Design** .

For more information on reviewing model predictive controller designs, see [review and Review Model Predictive Controller for Stability and Robustness Issues](#).

Comparison of responses for multiple model predictive controllers in the same plot using MPC Designer app

You can now simultaneously compare the response plots for multiple model predictive controllers using the MPC Designer app. On the **MPC Designer** tab, in the **Compare Controllers** drop-down list, select the controllers to compare.



You can add additional controllers to the MPC Designer **Data Browser** by:

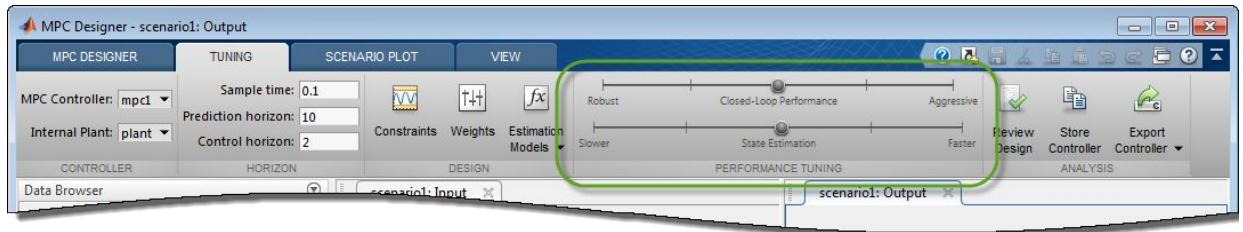
- Importing a controller from the MATLAB workspace — Select **Import Controller**
- Copying the current controller — Select **Store Controller**

For more information, see [Compare Multiple Controller Responses Using MPC Designer](#).

Interactive tuning of model predictive controller performance objectives

You can now tune controller performance objectives using interactive sliders. On the **Tuning** tab, use the **Performance Tuning** sliders to adjust the following:

- Closed-Loop Performance** objective — Moving towards more aggressive control simultaneously increases OV/MV weights and decreases MV Rate weights, which leads to tighter control of outputs and more aggressive control moves. Moving towards more robust control decreases OV/MV weights and increases MV Rate weights, which leads to relaxed control of outputs and more conservative control moves.
- State Estimation** speed — Moving towards faster state estimation simultaneously increases the gains for disturbance models and decreases the gains for noise models , which leads to more aggressive disturbance rejection. Moving towards slower state estimation decreases the gains for disturbance models and increases the gains for noise models, which leads to more conservative disturbance rejection.



mpctool command renamed to mpcDesigner

The `mpctool` command has been renamed. Starting in R2015b, open the MPC Designer app using the new `mpcDesigner` command.

For more information, see MPC Designer.

Compatibility Considerations

If you have scripts or functions that use `mpctool`, consider replacing those calls with `mpcDesigner`.

Functionality being removed or changed

Functionality	Result	Use This Instead	Compatibility Considerations
<code>mpctool</code>	Warns	<code>mpcDesigner</code>	Consider replacing <code>mpctool</code> with <code>mpcDesigner</code> .

R2015a

Version: 5.0.1

New Features

Bug Fixes

Compatibility Considerations

OutputVariables Integrator property of MPC controller being removed

The MPC controller property `OutputVariables(i).Integrator`, or `OV(i).Integrator`, is being removed. Previously, you specified custom integrator gains in the default output disturbance model using `OV(i).Integrator`. Starting in R2015a, you directly specify a custom output disturbance model as shown:

```
% Define a 2-by-2 plant model with no direct feedthrough
Plant = rss(2,2,2);
Plant.D = 0;
% Create an MPC object
MPCobj = mpc(Plant,1);
% Retrieve the default output disturbance model
Dmodel = getoutdist(MPCobj);
% Change the integrator gains
Dmodel = Dmodel * [2 0;0 3];
% Use new disturbance model in MPCobj
setoutdist(MPCobj, model ,Dmodel)
```

Compatibility Considerations

If your code uses the `OV(i).Integrator` property, you can update your code to use `setoutdist` and `getoutdist` for managing MPC controller output disturbance models.

For example, replace:

```
MPCobj.OV(1).Integrator = 2;
MPCobj.OV(2).Integrator = 3;
```

with:

```
Dmodel = getoutdist(MPCobj);
Dmodel = Dmodel * [2 0;0 3];
setoutdist(MPCobj, model ,Dmodel)
```

Use `tf(getoutdist(MPCobj))` to validate that the results are equivalent.

setoutdist command remove syntax being removed

The `setoutdist(MPCobj, remove ,channels)` syntax is being removed. Previously, you removed integrators from particular channels in the output disturbance model using

this syntax. Starting in R2015a, you directly specify a custom output disturbance model as shown:

```
% Define a 2-by-2 plant model with no direct feedthrough
Plant = rss(2,2,2);
Plant.D = 0;
% Create an MPC object
MPCobj = mpc(Plant,1);
% Retrieve the default output disturbance model
Dmodel = getoutdist(MPCobj);
% Remove the output disturbance model from output #1
Dmodel = sminreal([0;Dmodel(2,2)]);
% Use new disturbance model in MPCobj
setoutdist(MPCobj,'model',Dmodel)
```

When removing integrators from output disturbance channels, use `sminreal` to make the custom model structurally minimal.

Compatibility Considerations

If your code uses the `setoutdist(MPCobj, remove ,channels)` syntax, you can update your code to use `setoutdist` and `getoutdist` for managing MPC controller output disturbance models.

For example, replace:

```
setoutdist(MPCobj, remove ,1)
```

with:

```
Dmodel = getoutdist(MPCobj);
Dmodel = sminreal([0;Dmodel(2,2)]);
setoutdist(MPCobj,'model',Dmodel)
```

Use `tf(getoutdist(MPCobj))` to validate that the results are equivalent.

Functionality being removed or changed

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
<code>MPCObj.OV(i).-Integrator = value</code>	Warns	<code>setoutdist(MPCObj, model, sys)</code>	Use <code>setoutdist(MPCObj, model, sys)</code> to define custom output disturbance models. For more information, see “ OutputVariables Integrator property of MPC controller being removed” on page 3-2
<code>value = MPCObj.-OV(i).Integrator</code>	Warns	<code>sys = getoutdist(MPCObj)</code>	Use <code>getoutdist(MPCObj)</code> to retrieve MPC output disturbance models. For more information, see “ OutputVariables Integrator property of MPC controller being removed” on page 3-2
<code>setoutdist(MPCObj, remove, channels)</code>	Warns	<code>setoutdist(MPCObj, model, sys)</code>	Use <code>setoutdist(MPCObj, model, sys)</code> to define custom output disturbance models. For more information, see “ setoutdist command remove syntax being removed ” on page 3-2

R2014b

Version: 5.0

New Features

Bug Fixes

Compatibility Considerations

Explicit MPC control for applications with fast sample times using precomputed solutions

You can now design, simulate and deploy explicit MPC controllers for your plant. This functionality is useful for applications with fast sample times using pre-computed solutions.

To obtain an explicit MPC controller, you must first design a traditional MPC (also called implicit MPC) that is able to achieve your control objectives. Use the `generateExplicitMPC` command to design explicit MPC controllers. Use the `mpcmoveExplicit` command and the Explicit MPC Controller block to simulate explicit MPC controllers at the command-line and in Simulink, respectively.

For more information, see the following examples:

- Explicit MPC Control of a Single-Input-Single-Output Plant
- Explicit MPC Control of an Aircraft with Unstable Poles
- Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output

Adaptive MPC control through run-time changes to internal plant model

You can now simulate and deploy adaptive MPC controllers for your plant. This functionality helps you control a nonlinear plant across a wide operating range when the new linear plant model is available at run time.

To obtain an adaptive MPC controller, you must first design a traditional MPC (also called implicit MPC) that is able to achieve your control objectives at the initial operating condition. Then, update the internal plant model at each control interval at run time. Use the `mpcmoveAdaptive` command and the Adaptive MPC Controller block to simulate adaptive MPC controllers at the command-line and in Simulink, respectively.

For more information, see the following examples:

- Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization
- Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation

ScaleFactor property for MPC controllers, for making weight tuning independent of the engineering units of input and output variables

You can now specify scale factor in MPC controller in engineering units. The scale factors make weights dimensionless. Choosing proper scale factors, i.e. the operating ranges of the variable, makes weight tuning much easier. The default value of this property is 1.

For more information, see [Using Scale Factor to Facilitate Weight Tuning](#).

Option to use custom state estimation or measured state values instead of the built-in state estimation in MPC controllers

In addition to built-in state estimation, MPC controllers can now run custom state estimation. You can specify the state estimation mode by using `setEstimator(mpcobj, default)` and `setEstimator(mpcobj, custom)`, respectively.

When using custom state estimation, you can use the `Plant`, `Disturbance` and `Noise` properties of the controller state object `mpcstate` to provide custom state values at each control interval. The values can be from direct state measurements or your own state estimation algorithm. You must not programmatically change the `LastMV` property in the `mpcstate` object because it is still automatically updated by `mpcmmove`.

For more information, see [Using Custom State Estimation](#).

Compatibility Considerations

If your code changes the `LastMV` property of the state object to provide an external MV at run time, you must update the code to use `mpcmmoveopt` and specify the value in the `mpcmmoveopt.MVUsed` field instead.

If your code uses the `Plant`, `Disturbance` and `Noise` properties of the state object to provide external state values, you must use `setEstimator(mpcobj, custom)` to specify the controller to use the custom estimation mode before control starts.

Option to specify manipulated variable target

You can now specify targets on the manipulated variables during run time. At the command line, specify the value in the `MVTarget` field of the `mpcmmoveopt` object. In the

MPC controller blocks, select **Targets on manipulated variables (mv.target)** in the **Online Features** tab of the dialog box.

For more information, see Setting Targets for Manipulated Variables.

Run-time weight tuning on manipulated variables

You can now specify weights on the manipulated variables during run time. At the command line, specify the value in the **MVWeights** field of the **mpcmoveopt** object. In the MPC controller blocks, select **Weights on manipulated variables (u,wt)** in the **Online Features** tab of the dialog box.

For more information, see Setting Targets for Manipulated Variables.

Run-time weight tuning and performance monitoring in Multiple MPC Controllers block

You use the Multiple MPC Controllers block to implement gain-scheduled MPC control strategy by switching between multiple MPC controllers. You can now use this block to perform all the tasks that you perform with the MPC Controller block, such as online weight tuning, custom state estimation and performance monitoring.

getEstimator and setEstimator commands to obtain and change state estimation parameters

You can now use **getEstimator** to obtain the Kalman filter gains L and M, and additional parameters of the following observer equation used by the MPC controller:

$$\begin{aligned} y_m[n|n-1] &= C_m * x[n|n-1] + D_{vm} * v[n] \\ x[n|n] &= x[n|n-1] + M * (y_m[n] - y_m[n|n-1]) \\ x[n+1|n] &= A * x[n|n-1] + B_u * u[n] + B_v * v[n] + L * (y_m[n] - y_m[n|n-1]) \end{aligned}$$

Similarly, use **setEstimator** to change the parameters. For more information, see the **getEstimator** and **setEstimator** reference pages.

Compatibility Considerations

getestim and **setestim** commands warn and will be removed in a future release. Follow the instructions in the warning message to replace all instances with **getEstimator** and **setEstimator**.

Definition of external MV signal changed

The definition of externally supplied MV signals has been changed from $u[k]$ to $u[k-1]$. This implies that MPC controller now expects the external MV signal to be measured at the previous control interval $k-1$ and not at the current interval k .

Compatibility Considerations

If you enabled the `ext.mv` import in the **MPC Controller** or **Multiple MPC Controllers** block, do the following:

- If the connected signal does not come from the same MPC block, add a unit delay or memory block to the signal so that it is converted from $u[k]$ to $u[k-1]$.
- If the connected signal comes directly from the `mv` outport of the same MPC block, you see a warning about algebraic loop. To remove the warning, add a unit delay or memory block in the loop.

There is no incompatibility when you use `mpcmove` at the command prompt.

Unconnected input and output limits imports default changed to match mpc object

You can add imports (`umin`, `umax`, `ymin`, `ymax`) to the MPC controller blocks that you can connect to run-time constraint signals.

- If a channel is unconstrained in the `mpc` object, it remains unconstrained even if the import is connected and the provided value is ignored.
- If a channel is constrained, the original constraint specified in the `mpc` object is used when the corresponding import is unconnected.

Compatibility Considerations

Previously, when unconnected, **MPC Controller** block assumed the online constraint are unbounded ($+-\infty$). In this release, the simulation output may differ from previous releases because of the change in defaults

R2014a

Version: 4.2

New Features

Bug Fixes

IEC 61131–3 Structured Text generation from MPC Controller and Multiple MPC Controllers blocks using Simulink PLC Coder

The MPC Controller and Multiple MPC Controllers blocks support generation of IEC 61131–3 Structured Text using Simulink PLC Coder™. You can verify the generated code using the CoDeSys version 2.3 IDE.

For an example of Structured Text generation for an MPC controller see, Simulation and Structured Text Generation Using PLC Coder.

Reduced RAM usage for C code generated for MPC Controller and Multiple MPC Controllers blocks

The C code generated for the MPC Controller and Multiple MPC Controllers blocks reduces RAM usage. This change includes improved handling of memory allocation. For example, now the generated code does not use dynamic memory allocation, thereby extending support to targets that disallow dynamic memory allocation.

Estimate of data memory size used by deployed MPC controller at run time

You can determine if the data memory size required by an MPC controller exceeds the physical memory of the target system. The report generated by the review command now includes a platform-independent estimate of the data memory usage of an MPC controller at run time.

For an example, see Review Model Predictive Controller for Stability and Robustness Issues.

R2013b

Version: 4.1.3

New Features

Bug Fixes

Controller design for plant and disturbance models with internal delays

You can now design model predictive controllers for plant models, input (unmeasured) disturbance models, and output disturbance models that have internal delays. Previously, the software supported only input, output, or transport delays for plant and disturbance models.

When designing the MPC controller, the software discretizes the plant and disturbance models to the controller sample time. The software replaces each model delay of K sampling periods with K poles at $z = 0$. This delay absorption increases the model order, which increases the controller order.

If the models contain significant delays, you must specify an appropriate controller sample time. If the controller sample time is too large, you may not achieve the desired controller performance. However, if you sample a model that contains delays too fast, delay absorption leads to a high-order controller. Such a controller can have a large memory footprint, which can cause difficulty if you generate code for a real-time target. Also, high-order controllers can have numerical precision issues.

For more information regarding internal delays, see Internal Delays. To learn more about specifying a plant model, see Plant Specification. To specify the input disturbance model and the output disturbance model, see setindist and setoutdist.

Single-precision simulation and code generation using MPC Controller and Multiple MPC Controllers blocks

You can now specify the output data type for the **MPC Controller** and **Multiple MPC Controllers** blocks as **single**. This change provides the ability to simulate and generate code for model predictive controllers to be used on single-precision targets. Previously, these blocks supported only double-precision outputs.

To specify the output data type, in the block dialog, use the **Output data type** drop-down list.

For more information, see Simulation and Code Generation Using MPC Controller Block, MPC Controller, and Multiple MPC Controllers.

Conditional execution of MPC Controller and Multiple MPC Controllers blocks using Function-Call Subsystem and Triggered Subsystem blocks

MPC Controller and Multiple MPC Controllers blocks can now inherit the parent subsystem's sample time. Therefore, you can conditionally execute these blocks using the **Function-Call Subsystem** or **Triggered Subsystem** blocks.

To specify that the output sample time be inherited, in the block dialog, select the **Block uses inherited sample time (-1)** check box.

Note: When you place an MPC controller inside a **Function-Call Subsystem** or **Triggered Subsystem** block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

For more information, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems, MPC Controller, and Multiple MPC Controllers.

R2013a

Version: 4.1.2

Bug Fixes

R2012b

Version: 4.1.1

Bug Fixes

R2012a

Version: 4.1

New Features

Bug Fixes

Compatibility Considerations

Run-Time Preview of Reference and Measured Disturbance Signals with MPC Controller Block

This release introduces the ability to preview signals by using the **ref** and **md** imports of the MPC Controller block and the Multiple MPC Controllers block.

The **ref** import now accepts an N -by- N_{y} signal, where N is the number of previewing steps and N_{y} is the number of plant outputs.

The **md** import now accepts an N -by- N_{md} signal, where N is the number of previewing steps and N_{md} is the number of measured disturbances.

You cannot preview if the input signal is a vector, unless N_{y} or N_{md} , as appropriate, is 1.

For more information, see:

- Improving Control Performance with Look-Ahead (Previewing)
- Chemical Reactor with Multiple Operating Points

Compatibility Considerations

In the current release, if you have models with the MPC Controller block or the Multiple MPC Controllers block, you will see a warning if your blocks contain:

- A **custom reference signal** specified in the MATLAB workspace.
- A **custom disturbance signal** specified in the MATLAB workspace.

Custom Reference Signal Specified in MATLAB Workspace

You must clear this warning. If you ignore the warning, the block will assume that the **ref** signal is zero. This behavior is equivalent to leaving the **ref** import unconnected.

- **Without Look-Ahead (Previewing) Option.** To eliminate this warning:
 - 1 Add a **From Workspace** block to your model.
 - 2 Specify your reference signal variable name as the **Data** parameter of the **From Workspace** block.
 - 3 Connect the output of the **From Workspace** block to the **ref** import of the MPC Controller block or the Multiple MPC Controllers block.
- **With Look-Ahead (Previewing) Option.** To eliminate this warning:

-
- 1 Copy the **Reference Previewer** block from the `mpc_preview` model and place it in your model. See [Improving Control Performance with Look-Ahead \(Previewing\)](#) for more information.
 - 2 Specify your reference signal variable name as the **Signal** parameter of the **Reference Previewer** block. Also specify appropriate values for the **Sampling time** and **Number of previewing steps** parameters.
 - 3 Connect the output of the **Reference Previewer** block to the **ref** import of the MPC Controller block or the Multiple MPC Controllers block.

Custom Disturbance Signal Specified in MATLAB Workspace

You must clear this warning. If you ignore the warning, the block will assume that the `md` signal is zero. This behavior is equivalent to leaving the `md` import unconnected.

- **Without Look-Ahead (Previewing) Option.** To eliminate this warning:
 - 1 Add a **From Workspace** block to your model.
 - 2 Specify your disturbance signal variable name as the **Data** parameter of the **From Workspace** block.
 - 3 Connect the output of the **From Workspace** block to the **md** import of the MPC Controller block or the Multiple MPC Controllers block.
- **With Look-Ahead (Previewing) Option.** To eliminate this warning:
 - 1 Copy the **Measured Disturbance Previewer** block from the `mpc_preview` model, and place it in your model. See [Improving Control Performance with Look-Ahead \(Previewing\)](#) for more information.
 - 2 Specify your measured disturbance signal variable name as the **Signal** parameter of the **Reference Previewer** block. Also specify appropriate values for the **Sampling time** and **Number of previewing steps** parameters.
 - 3 Connect the output of the **Measured Disturbance Previewer** block to the **md** import of the MPC Controller block or the Multiple MPC Controllers block.

R2011b

Version: 4.0

New Features

Bug Fixes

Compatibility Considerations

C Code Generation Improvements for All Targets with MPC Controller Block

The MPC Controller block has been re-implemented using a MATLAB Function block and now supports code generation for all Simulink Coder™ targets.

For more information, see [Code Generation with Simulink Coder](#).

Faster QP Solver Algorithm for Improving MPC Controller Performance

This release implements a new quadratic problem (QP) solver that uses the KWIK algorithm. KWIK is faster and more numerically robust than the previous solver for ill-conditioned QP problems. You can use this solver without default constraints on decision variables.

For more information, see [MPC QP Solver](#).

Run-Time Weight Tuning and Constraint Softening for MPC Controller

This release introduces three new run-time tuning parameters for the MPC Controller block:

- **Weights on plant outputs**
- **Weights on manipulated variables rate**
- **Weight on overall constraints softening**

You can use these parameters to tune the weights on plant outputs, manipulated variables rate, and overall constraint softening. These capabilities are available in real time, without redesigning or re-implementing the MPC controller, and help adjust the controller performance.

For more information, see [Tuning Controller Weights](#).

You can also use an `mpcmovopt` object as an input to `mpcmove` to tune the weights and constraints.

For more information, see the following:

- [Switching Controllers Based on Optimal Costs](#)
- [Varying Input and Output Constraints](#)

Run-Time Monitoring of MPC Controller Performance to Detect When an Optimal Solution Cannot Be Found

This release introduces a new outport parameter—**Optimization status** in the MPC Controller block. You can use this outport to monitor the status of the optimization and take the necessary action when an optimal solution cannot be found. For more information, see [Monitoring Optimization Status to Detect Controller Failures](#).

You can also use the `Info.QPCode` field of the output of `mpcmove` to monitor the status of the optimization.

For more information, see the `mpcmove` reference page.

review Command for Diagnosing Issues with MPC Controller Parameters That Could Lead to Run-Time Failures

You can now use `review` to detect potential stability and robustness issues (both offline and at run time) with an MPC Controller design. The following aspects of the system are inspected:

- Stability of the model predictive controller and the closed loop
- Potential for contradictory settings in the specified constraints and mitigation of an ill-conditioned QP problem by softening constraints
- Validity of QP Hessian matrix

Use this command before implementing the MPC Controller, in conjunction with simulation.

For more information, see the following:

- [review reference](#)
- [Reviewing Model Predictive Controller Design for Potential Stability and Robustness Issues](#).

mpcmove Returns Aligned Time Horizons for Optimal Control, Predicted Output and Estimated State

`mpcmove` now returns `Info` with a time horizon of $t=k, \dots, k+p$, where k is the current time and p is the prediction horizon for the following fields:

- `Info.Uopt` — Optimal manipulated variable adjustments
- `Info.Yopt` — Predicted output
- `Info.Xopt` — Predicted state
- `Info.Topt` — Time horizon

You can now plot `Info.Uopt`, `Info.Yopt` and `Info.Xopt` using `Info.Topt` as the time vector.

For more information, see the `mpcmove` reference page.

Functionality Being Removed or Changed

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
<code>getmpcdata</code>	Still runs	<ul style="list-style-type: none">• <code>get</code>• <code>getconstraint</code>• <code>getestim</code>• <code>getindist</code>• <code>getoutdist</code>	Not applicable
<code>pack</code>	Still runs	Not applicable	Not applicable
<code>qpdantz</code>	Warns	<code>quadprog</code> (requires Optimization Toolbox™)	Replace all instances of <code>qpdantz</code> with <code>quadprog</code> .
<code>setmpcdata</code>	Still runs	<ul style="list-style-type: none">• <code>set</code>• <code>setconstraint</code>• <code>setestim</code>• <code>setindist</code>• <code>setoutdist</code>	Not applicable

R2011a

Version: 3.3

New Features

Bug Fixes

Support for Custom Constraints on MPC Controller Inputs and Outputs

In addition to upper and lower bounds, you can now specify constraints on linear combinations of an MPC controller inputs ($u(t)$) and outputs ($y(t)$). Specify custom constraints, such as $u1 + u2 < 1$ or $u + y < 2$, in the mpc object using setconstraint.

For more information, see:

- Custom Constraints on Inputs and Outputs
- Custom Constraints in a Blending Process
- MPC Control with Constraints on a Combination of Input and Output Signals demo

Ability to Specify Terminal Constraints and Weights on MPC Controller

You can now specify weights and constraints on the terminal predicted states of an MPC controller.

Using terminal weights, you can achieve infinite horizon control. For example, you can design an unconstrained MPC controller that behaves in exactly the same way as a Linear-Quadratic Regulator (LQR). You can use terminal constraints as an alternative way to achieve closed-loop stability by defining a terminal region.

You can specify both weights and constraints using the setterminal command.

For more information, see:

- Terminal Weights and Constraints
- Using Terminal Penalty to Provide LQR Performance
- Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation demo

Ability to Access Optimal Cost and Optimal Control Sequence

This release introduces two new parameters **Enable optimal cost outport** and **Enable control sequence outport** in the MPC Controller block. Using these parameters, you can access the optimal cost and control sequence along the prediction horizon. This information helps you analyze control performance.

You can also access the optimal cost and control sequence programmatically using the new **Cost** and **Yopt** fields, respectively, of the structure **info** returned by mpcremove.

For more information on using optimal cost and control sequence, see the following demos:

- MPC Control with Input Quantization Based on Comparing the Optimal Costs
- Analysis of Control Sequences Optimized by MPC on a Double Integrator System

R2010b

Version: 3.2.1

No New Features or Changes

R2010a

Version: 3.2

New Features

Bug Fixes

New Ability to Analyze SISO Generalized Predictive Controllers (GPC)

You can now use `gpc2mpc` to convert your SISO GPC controller to an MPC controller. Analyze and simulate the resulting MPC controller using available Model Predictive Control Toolbox™ commands.

For more information, see the `gpc2mpc` reference page.

R2009b

Version: 3.1.1

Bug Fixes

R2009a

Version: 3.1

New Features

Bug Fixes

New Sensitivity Analysis to Determine Effect of Weights on Tuning MPC Controllers

You can now perform sensitivity analysis to determine the effect of weights on the closed-loop performance of your system. You can perform sensitivity analysis using the following:

- MPC Tuning Advisor. See Tuning Advisor in the *Model Predictive Control User's Guide*.
- sensitivity command. See the sensitivity reference page.

R2008b

Version: 3.0

New Features

Bug Fixes

New Multiple MPC Controllers Block in the Model Predictive Control Toolbox Simulink Library

You can now use the Multiple MPC Controllers block in Simulink software to control a nonlinear process over a range of operating points. You include an MPC controller for each operating point in the Multiple MPC Controllers block and specify switching between these controllers in real-time based on the input scheduling signal to the block. If you need to change the design of a specific controller, you can open the MPC Design Tool GUI directly from the Multiple MPC Controllers block.

During model simulation, Model Predictive Control Toolbox provides bumpless transfer when the system transitions between operating points.

To learn more about configuring the new block, see the Multiple MPC Controllers block reference page.

Tested Code Generation Support for Real-Time Workshop Target Systems

After designing an MPC controller in Simulink software using the MPC Controller block, you can use Real-Time Workshop® software to build this controller and deploy it to the following target systems for real-time control:

- Generic Real-Time Target
- Real-Time Workshop Embedded Coder™
- Real-Time Windows Target
- Rapid Simulation Target
- Target Support Package FM5
- xPC Target
- dSpace Target
- Target for Infineon TriCore

The following target systems are either not supported or not recommended because they result in significant performance issues:

- Embedded Target for TI C2000 DSP
- Embedded Target for TI C6000 DSP
- Target Support Package IC1 (for Infineon C166)

-
- Tornado (VxWorks) Real-Time Target

Note The Multiple MPC Controllers block has not been tested with the target systems supported by Real-Time Workshop software.

Ability to Design Controllers with Time-Varying Weights and Constraints Using the GUI

While you design an MPC controller using the MPC Design Tool graphical user interface (GUI), you can specify time-varying weights and constraints for manipulated variables, rate of change of manipulated variables, and output variables. In the previous version, you could only specify the time-varying weights and constraints at the command line.

Furthermore, you can load an MPC controller with time-varying information from the command line into the MPC Design Tool GUI.

To learn more about the new options in the MPC Design Tool GUI, see the Model Predictive Control Toolbox documentation.

R2008a

Version: 2.3.1

No New Features or Changes

R2007b

Version: 2.3

New Features

New Option for Specifying Time-Varying Constraints

You can now configure the Model Predictive Controller block in Simulink to accept time-varying constraint signals that are generated by other blocks. To add imports to which you can connect time-varying constraint specifications, select the new **Enable input port for input and output limits** check box in the MPC Controller block. See also the [mpcvarbounds](#) demo.

In the previous version, you could only specify the constraints during the design phase and these constraints remained constant for the duration of the simulation.

For more information about the new **Enable input port for input and output limits** check box in the Model Predictive Controller block, see the MPC Controller block reference page.

Ability to Specify Nondiagonal Q and R Weight Matrices in the Cost Function

You can now specify off-diagonal weights in the cost function. In the previous release, only diagonal Q and R matrices were supported.

To learn more about specifying off-diagonal weights, see the discussion about weights in the MPC Controller block reference pages.

To access a new demo that shows how to use nondiagonal weight matrices, type the following command at the MATLAB prompt:

```
showdemo( mpcweightsdemo )
```

R2007a

Version: 2.2.4

Bug Fixes

R2006b

Version: 2.2.3

No New Features or Changes

R2006a

Version: 2.2.2

New Features

Bumpless Transfer Added to MPC Block

Bumpless transfer between manual and automatic operation or from one controller to another has been added to the Model Predictive Controller block in Simulink. This block now allows feedback of the true manipulated variable signals, which allows the controller to maintain an accurate state estimate during periods when its calculated adjustments are not being sent to the plant. For example, the controller's output might be ignored during a startup period or during temporary intervention by a (simulated) plant operator. If the controller assumes that its adjustments are being implemented (the default behavior), its state estimate will be incorrect, leading to a "bump" when the controller is reconnected to the plant. A tutorial example has been added to the documentation.

New Bumpless Transfer Demo

A new demo illustrating bumpless transfer has been added to the toolbox.

R14SP3

Version: 2.2.1

No New Features or Changes

R14SP2

Version: 2.2

No New Features or Changes

