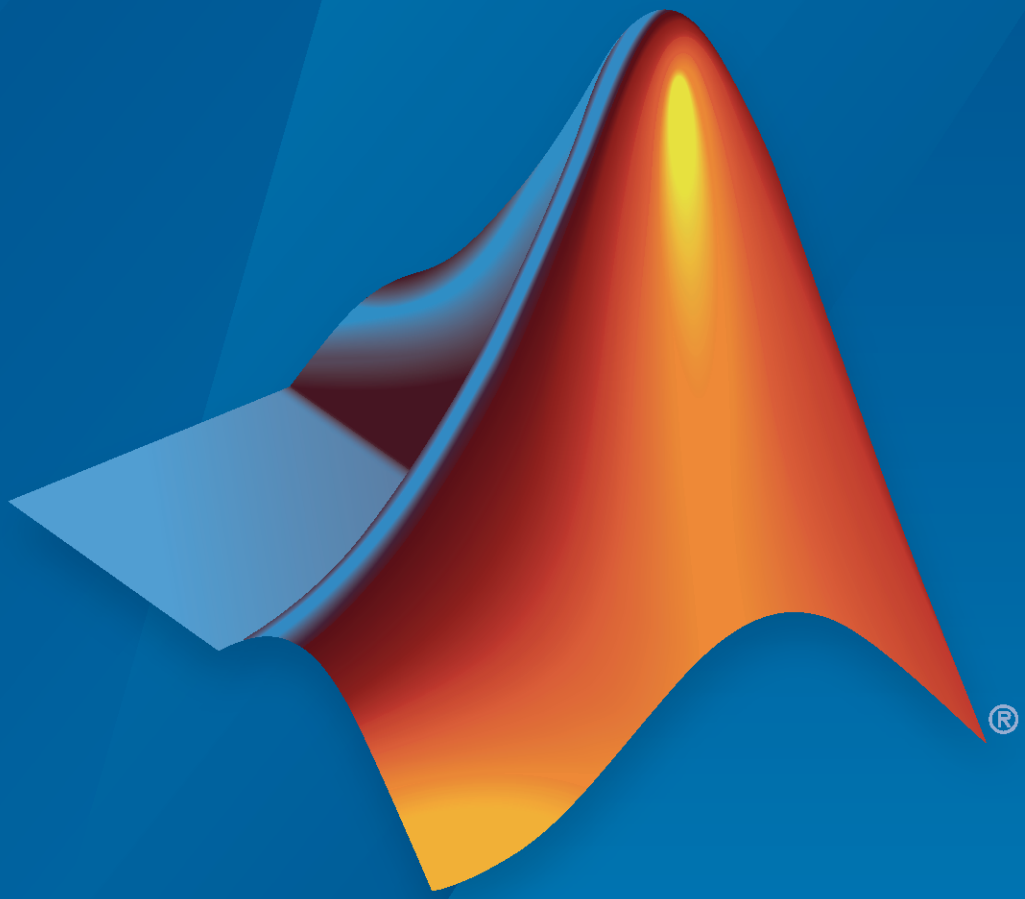
Model Predictive Control Toolbox™

User's Guide

Alberto Bemporad
N. Lawrence Ricker
Manfred Morari



MATLAB®

R2021b

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Model Predictive Control Toolbox™ User's Guide

© COPYRIGHT 2005–2021 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)
September 2016	Online only	Revised for Version 5.2.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2.2 (Release 2017a)
September 2017	Online only	Revised for Version 6.0 (Release 2017b)
March 2018	Online only	Revised for Version 6.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.2 (Release 2018b)
March 2019	Online only	Revised for Version 6.3 (Release 2019a)
September 2019	Online only	Revised for Version 6.3.1 (Release 2019b)
March 2020	Online only	Revised for Version 6.4 (Release 2020a)
September 2020	Online only	Revised for Version 7.0 (Release 2020b)
March 2021	Online only	Revised for Version 7.1 (Release 2021a)
September 2021	Online only	Revised for Version 7.2 (Release 2021b)

Contents

1 Controller Creation

Choose Sample Time and Horizons **1-2**
Sample Time **1-2**
Prediction Horizon **1-2**
Control Horizon **1-3**
Defining Sample Time and Horizons **1-3**

Specify Constraints **1-5**
Input and Output Constraints **1-5**
Constraint Softening **1-7**

DC Servomotor with Constraint on Unmeasured Output **1-10**

Discrete Control Set MPC **1-15**

Solve a Discrete Set MPC Problem in MATLAB **1-16**

Solve a Discrete Set MPC Problem in Simulink **1-21**

Surge Tank Control Using Discrete Control Set MPC **1-24**

Specify Scale Factors **1-29**
Determine Scale Factors **1-29**
Specify Scale Factors at Command Line **1-29**
Specify Scale Factors Using MPC Designer **1-30**

Using Scale Factors to Facilitate MPC Weights Tuning **1-32**

Tune Weights **1-43**
Initial Tuning **1-43**
Testing and Refinement **1-44**
Robustness **1-45**

Design Model Predictive Controller at Equilibrium Operating Point . . . **1-47**

Design MPC Controller for Plant with Delays **1-52**

Design MPC Controller for Nonsquare Plants **1-59**
More Outputs Than Manipulated Variables **1-59**
More Manipulated Variables Than Outputs **1-61**

Design MPC Controller for Identified Plant Model **1-64**
Design Controller for Identified Plant Using Apps **1-64**
Design Controller for Identified Plant at the Command Line **1-79**

Configure Noise Channels as Unmeasured Disturbances 1-84

Generate MATLAB Code from MPC Designer 1-89

Design MPC Controller for Position Servomechanism 1-91

Design MPC Controller for Paper Machine Process 1-110

Control of an Inverted Pendulum on a Cart 1-132

Thermo-Mechanical Pulping Process with Multiple Control Objectives 1-140

MPC Control of an Aircraft with Unstable Poles 1-148

2 Model Predictive Control Basics

Controller State Estimation 2-2
Controller State Variables 2-2
State Observer 2-3
State Estimation 2-3
Built-in Steady-State Kalman Gains Calculation 2-4
Output Variable Prediction 2-5

Optimization Problem 2-7
Overview 2-7
Standard Cost Function 2-7
Alternative Cost Function 2-9
Constraints 2-10
QP Matrices 2-11
Unconstrained Model Predictive Control 2-15

QP Solvers 2-17
Built-In QP Solvers 2-17
Custom QP Solver 2-19
Use quadprog as Custom QP Solver 2-22
Integration with FORCESPRO Solver 2-23

3 Controller Refinement

Setting Targets for Manipulated Variables 3-2

Constraints on Linear Combinations of Inputs and Outputs 3-5

Use Custom Constraints in Blending Process 3-9

Terminal Weights and Constraints 3-18

Provide LQR Performance Using Terminal Penalty Weights **3-20**

Adjust Disturbance and Noise Models **3-25**
Overview **3-25**
Output Disturbance Model **3-25**
Measurement Noise Model **3-27**
Input Disturbance Model **3-28**
Restrictions **3-30**
Disturbance Rejection Tuning **3-30**

Custom State Estimation **3-32**

Implement Custom State Estimator Equivalent to Built-In Kalman Filter **3-37**

Manipulated Variable Blocking **3-44**
Specify Blocking Interval Lengths **3-44**
Interpolate Block Moves for Nonlinear MPC **3-46**

Specifying Alternative Cost Function with Off-Diagonal Weight Matrices **3-48**

4 Controller Analysis

Review Model Predictive Controller for Stability and Robustness Issues **4-2**

Compute Steady-State Gain **4-17**

Extract Controller **4-20**

Compare Multiple Controller Responses Using MPC Designer **4-22**

Adjust Input and Output Weights Based on Sensitivity Analysis **4-31**

Understanding Control Behavior by Examining Optimal Control Sequence **4-37**

5 Controller Simulation

Simulating MPC Controller with Plant Model Mismatch **5-2**

Test MPC Controller Robustness using MPC Designer **5-5**

Generate Simulink Model from MPC Designer **5-14**

Test an Existing MPC Controller with Simulink **5-16**

Signal Previewing **5-19**

Improving Control Performance with Look-Ahead (Previewing) **5-20**

Update Constraints at Run Time **5-27**
Update Bounds on Input and Output Signals at Run Time **5-27**
Update Mixed Input/Output Constraints at Run Time **5-28**

Vary Input and Output Bounds at Run Time **5-30**

Tune Weights at Run Time **5-35**

Tuning MPC Controller Weights at Run-Time **5-36**

Setting Time-Varying Weights and Constraints with MPC Designer **5-42**
Time-Varying Weights **5-42**
Time-Varying Constraints **5-43**

Adjust Horizons at Run Time **5-45**
Adjust Horizons in MATLAB **5-45**
Adjust Horizons in Simulink **5-45**
Code Generation **5-45**
Effect on Time-Varying Controller Parameters and Signals **5-46**

Evaluate Control Performance Using Run-Time Horizon Adjustment . . . **5-48**

Switch Controller Online and Offline with Bumpless Transfer **5-57**

Switching Controllers Based on Optimal Costs **5-67**

Monitoring Optimization Status to Detect Controller Failures **5-73**

Simulate MPC Controller with a Custom QP Solver **5-77**

Use Suboptimal Solution in Fast MPC Applications **5-86**

Design and Cosimulate Control of High-Fidelity Distillation Tower with Aspen Plus Dynamics **5-93**

Simulate Linear MPC Controller with Nonlinear Plant using Successive Linearizations **5-111**
Nonlinear CSTR Application **5-111**
Example Code for Successive Linearization **5-111**
CSTR Results and Discussion **5-113**

6 Explicit MPC Design

Explicit MPC **6-2**

Design Workflow for Explicit MPC **6-4**
Traditional (Implicit) MPC Design **6-4**
Explicit MPC Generation **6-4**
Explicit MPC Simplification **6-5**
Implementation **6-5**
Simulation **6-6**

Explicit MPC Control of a Single-Input-Single-Output Plant **6-7**

Explicit MPC Control of an Aircraft with Unstable Poles **6-17**

Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output **6-24**

Explicit MPC Control of an Inverted Pendulum on a Cart **6-33**

7 Adaptive MPC Design

Adaptive MPC **7-2**
When to Use Adaptive MPC **7-2**
Plant Model **7-2**
Nominal Operating Point **7-3**
State Estimation **7-3**

Model Updating Strategy **7-5**
Overview **7-5**
Other Considerations **7-5**

Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization **7-7**

Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation **7-17**

Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter-Varying System **7-27**

Obstacle Avoidance Using Adaptive Model Predictive Control **7-38**

Time-Varying MPC **7-49**
When to Use Time-Varying MPC **7-49**
Time-Varying Prediction Models **7-49**
Time-Varying Nominal Conditions **7-50**
State Estimation **7-51**

Time-Varying MPC Control of a Time-Varying Plant **7-52**

Time-Varying MPC Control of an Inverted Pendulum on a Cart **7-58**

8 Gain Scheduling MPC Design

Gain-Scheduled MPC **8-2**
Design Workflow **8-2**

Schedule Controllers at Multiple Operating Points **8-4**

Gain-Scheduled MPC Control of Nonlinear Chemical Reactor **8-22**

Gain-Scheduled Implicit and Explicit MPC Control of Mass-Spring System **8-42**

Gain-Scheduled MPC Control of an Inverted Pendulum on a Cart **8-58**

9 Nonlinear MPC

Nonlinear MPC **9-2**
Generic Nonlinear MPC **9-2**
Multistage Nonlinear MPC **9-2**

Specify Prediction Model for Nonlinear MPC **9-5**
State Function **9-5**
Output Function **9-8**
Specify Optional Model Parameters **9-10**
Augment Prediction Model with Unmeasured Disturbances **9-10**

Specify Cost Function for Nonlinear MPC **9-12**
Custom Cost Function **9-12**
Cost Function Jacobian **9-16**

Specify Constraints for Nonlinear MPC **9-19**
Standard Linear Constraints **9-19**
Custom Constraints **9-20**
Custom Constraint Jacobians **9-24**

Configure Optimization Solver for Nonlinear MPC **9-27**
Solver Decision Variables **9-27**
Specify Initial Guesses **9-27**
Configure fmincon Options **9-27**
Specify Custom Solver **9-28**

Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC **9-32**

Generate Code to Plan and Execute Collision-Free Trajectories using KINOVA Gen3 Manipulator **9-43**

Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control **9-49**

Nonlinear Model Predictive Control of an Exothermic Chemical Reactor **9-61**

Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver **9-68**

Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant **9-76**

Optimization and Control of a Fed-Batch Reactor Using Nonlinear MPC **9-84**

Lane Following Using Nonlinear Model Predictive Control **9-94**

Lane Change Assist Using Nonlinear Model Predictive Control **9-101**

Control of Quadrotor Using Nonlinear Model Predictive Control **9-111**

Economic MPC **9-117**

Economic MPC Control of Ethylene Oxide Production **9-119**

Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC **9-127**

Land a Rocket Using Multistage Nonlinear MPC **9-141**

10 Code Generation

Generate Code and Deploy Controller to Real-Time Targets **10-2**
Code Generation in MATLAB **10-2**
Code Generation in Simulink **10-2**
Generate CUDA Code for Linear MPC Controllers **10-3**
Sampling Rate in Real-Time Environment **10-4**
QP Problem Construction for Generated C Code **10-4**
Code Generation for Custom QP Solvers **10-6**

Generate Code to Compute Optimal MPC Moves in MATLAB **10-7**

Simulation and Code Generation Using Simulink Coder **10-13**

Simulation and Structured Text Generation Using Simulink PLC Coder **10-20**

Use the GPU to Compute MPC Moves in MATLAB **10-25**

Use the GPU to Simulate an MPC Controller in Simulink **10-31**

Using MPC Controller Block Inside Function-Call and Triggered Subsystems **10-34**

Solve Custom MPC Quadratic Programming Problem and Generate Code **10-46**

Simulate and Generate Code for MPC Controller with Custom QP Solver **10-56**

Real-Time Control with OPC Toolbox **10-63**

Implement MPC Controllers using Embotech FORCESPRO Solvers . . . **10-67**
Embotech Quadratic Programming (QP) Solver **10-67**
Embotech Nonlinear Programming (NLP) Solver **10-68**

11 Automated Driving Applications

Automated Driving Using Model Predictive Control **11-2**
Simulation in Simulink **11-3**
Controller Customization **11-3**
Integration with Automated Driving Toolbox **11-4**

Adaptive Cruise Control System Using Model Predictive Control **11-5**

Adaptive Cruise Control with Sensor Fusion **11-10**

Lane Keeping Assist System Using Model Predictive Control **11-28**

Lane Keeping Assist with Lane Detection **11-33**

Lane Following Control with Sensor Fusion and Lane Detection **11-51**

Highway Lane Following **11-62**

Highway Lane Change **11-78**

Automate Testing for Highway Lane Following **11-89**

Highway Lane Following with Intelligent Vehicles **11-99**

Parking Valet Using Nonlinear Model Predictive Control **11-117**

Parallel Parking Using Nonlinear Model Predictive Control **11-126**

Parallel Parking Using RRT Planner and MPC Tracking Controller . . **11-139**

Traffic Light Negotiation **11-148**

Traffic Light Negotiation with Unreal Engine Visualization **11-162**

Controller Creation

- “Choose Sample Time and Horizons” on page 1-2
- “Specify Constraints” on page 1-5
- “DC Servomotor with Constraint on Unmeasured Output” on page 1-10
- “Discrete Control Set MPC” on page 1-15
- “Solve a Discrete Set MPC Problem in MATLAB ” on page 1-16
- “Solve a Discrete Set MPC Problem in Simulink” on page 1-21
- “Surge Tank Control Using Discrete Control Set MPC” on page 1-24
- “Specify Scale Factors” on page 1-29
- “Using Scale Factors to Facilitate MPC Weights Tuning” on page 1-32
- “Tune Weights” on page 1-43
- “Design Model Predictive Controller at Equilibrium Operating Point” on page 1-47
- “Design MPC Controller for Plant with Delays” on page 1-52
- “Design MPC Controller for Nonsquare Plants” on page 1-59
- “Design MPC Controller for Identified Plant Model” on page 1-64
- “Generate MATLAB Code from MPC Designer” on page 1-89
- “Design MPC Controller for Position Servomechanism” on page 1-91
- “Design MPC Controller for Paper Machine Process” on page 1-110
- “Control of an Inverted Pendulum on a Cart” on page 1-132
- “Thermo-Mechanical Pulping Process with Multiple Control Objectives” on page 1-140
- “MPC Control of an Aircraft with Unstable Poles” on page 1-148

Choose Sample Time and Horizons

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, $T_s >> 1$ 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 3-44).
  - Use a small p with terminal weighting to approximate LQR behavior (See “Terminal Weights and Constraints” on page 3-18).

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 $m << p$ are as follows:

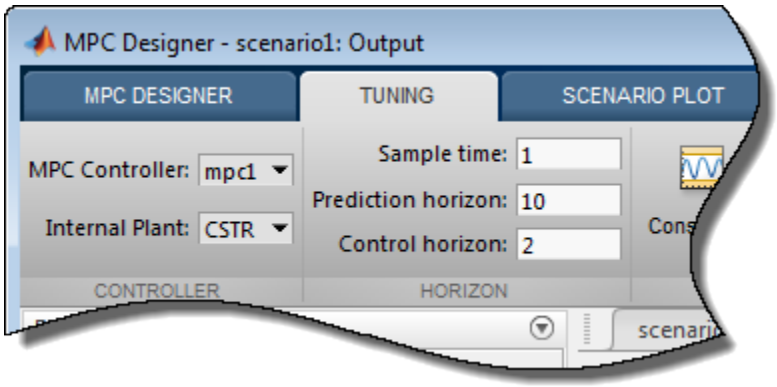
- Small m means fewer variables to compute in the QP solved at each control interval, which promotes faster computations.
- If the plant includes delays, $m < p$ is essential. Otherwise, some MV moves might not affect any of the plant outputs before the end of the prediction horizon, leading to a singular 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

- “Specify Constraints” on page 1-5

Specify Constraints

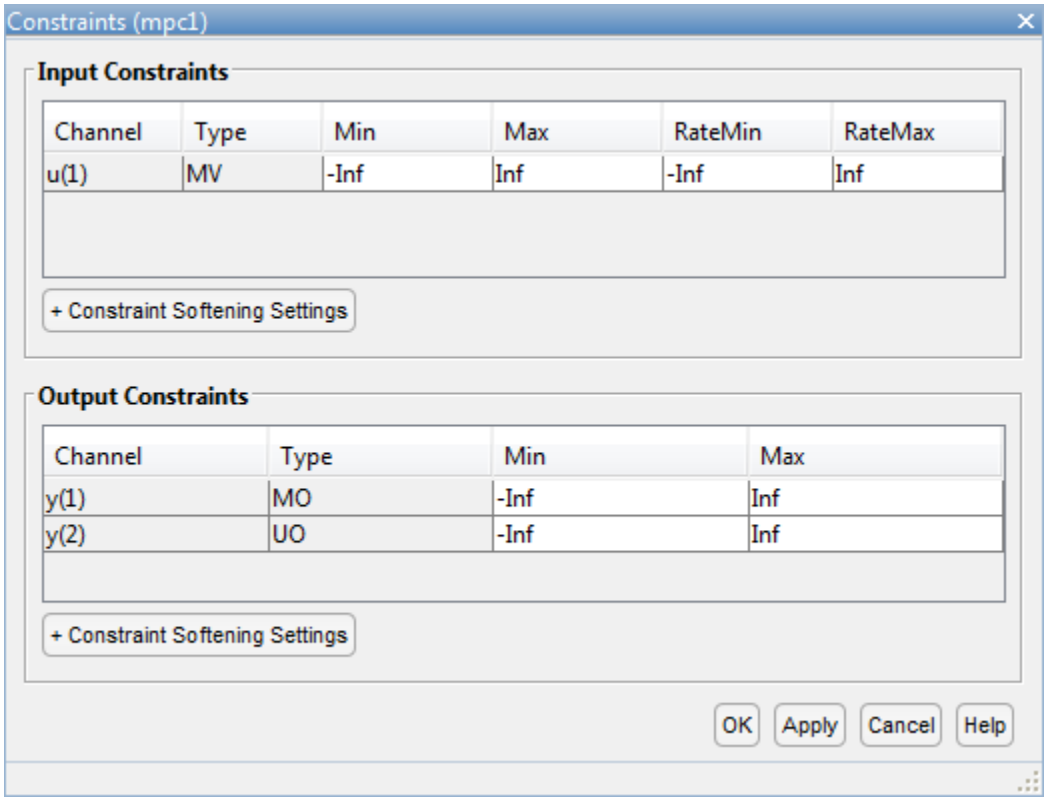
Input and Output Constraints

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 constraints. (MV = plant manipulated variable; OV = plant output variable; MV increment = $u(k) - u(k-1)$.

Constraint	Controller Property	Constraint Softening
Lower bound on *i*th MV	`MV(i).Min > -Inf`	`MV(i).MinECR > 0`
Upper bound on *i*th MV	`MV(i).Max < Inf`	`MV(i).MaxECR > 0`
Lower bound on *i*th OV	`OV(i).Min > -Inf`	`OV(i).MinECR > 0`
Upper bound on *i*th OV	`OV(i).Max < Inf`	`OV(i).MaxECR > 0`
Lower bound on *i*th MV increment	`MV(i).RateMin > -Inf`	`MV(i).RateMinECR > 0`
Upper bound on *i*th MV increment	`MV(i).RateMax < Inf`	`MV(i).RateMaxECR > 0`

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 2-10 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 “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42.
- Consider a time-varying OV constraint that is easy to satisfy early in the horizon, gradually tapering to a more strict constraint. See “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42.
- 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 “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42.

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 $u(k) = u(k–1)$, that is, 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.

If the plant is subject to disturbances and there are either hard output constraints or hard mixed input-output constraints, then 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 equal concern for relaxation (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 $w^u_{i,j} = w^{\Delta u}_{i,j} = 0$, 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:

$$J(z_k) = \sum_{j=1}^{n_y} \sum_{i=1}^{p} \left\{ \frac{w^y_{i,j}}{s^y_j} \left[r_j(k+i\,|k) - y_j(k+i\,|k) \right] \right\}^2 + \rho_\varepsilon \varepsilon_k^2 .$$

Suppose that you have also specified hard MV bounds with $V^u_{j,min}(i) = 0$ and $V^u_{j,max}(i) = 0$. Then these constraints simplify to:

$$\frac{u_{j,min}(i)}{s^u_j} \le \frac{u_j(k+i-1\,|k)}{s^u_j} \le \frac{u_{j,max}(i)}{s^u_j}, \qquad i = 1:p, \qquad j = 1:n_u .$$

Thus, the slack variable, ε_k, no longer appears in the above equations. You have also specified soft constraints on plant outputs with $V^y_{j,min}(i) > 0$ and $V^y_{j,max}(i) > 0$.

$$\frac{y_{j,min}(i)}{s^y_j} - \varepsilon_k V^y_{j,min}(i) \le \frac{y_j(k+i\,|k)}{s^y_j} \le \frac{y_{j,max}(i)}{s^y_j} + \varepsilon_k V^y_{j,max}(i), \qquad i = 1:p, \qquad j = 1:n_y .$$

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 $\varepsilon_k > 0$. A larger ρ_ε weight, however, increases the likelihood that the optimal ε_k will be small or zero.

If the optimal $\varepsilon_k > 0$, at least one of the bound inequalities must be active (at equality). A relatively large $V^y_{j,max}(i)$ makes it easier to satisfy the constraint with a small ε_k. In that case,

$$\frac{y_j(k+i\,|k)}{s^y_j}$$

can be larger, without exceeding

$$\frac{y_{j,max}(i)}{s^y_j} + \varepsilon_k V^y_{j,max}(i) .$$

Notice that $V^y_{j,max}(i)$ 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 “Specify Scale Factors” on page 1-29.
- 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

- “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42
- “Terminal Weights and Constraints” on page 3-18
- “Optimization Problem” on page 2-7
- “DC Servomotor with Constraint on Unmeasured Output” on page 1-10

DC Servomotor with Constraint on Unmeasured Output

This example shows how to design a model predictive controller for a DC servomechanism under voltage and shaft torque constraints.

For a similar example that uses explicit MPC, see “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24.

Define DC-Servo Motor Model

The `mpcmotormodel` function returns the plant model needed for the example. The linear open-loop dynamic model is defined in `plant`. The variable `tau` is the maximum admissible torque, this is going to be used as an output constraint.

```
[plant,tau] = mpcmotormodel;
```

The plant control input is the DC voltage, the four state variables are the angular position and velocities of the load and shaft. The measurable output is the angular position of the load. The second output, torque, is not measurable. Specify input and output signal types for the MPC controller.

```
plant = setmpcsignals(plant,'MV',1,'MO',1,'UO',2);
```

Specify 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);
```

Specify OV Constraints

Torque constraints of +|tau| and -|tau| are only imposed during the first three prediction steps.

```
OV = struct('Min',{-Inf, [-tau;-tau;-tau;-Inf]},...
            'Max',{Inf, [tau;tau;tau;Inf]},...
            'ScaleFactor',{2*pi, 2*tau});
```

Specify Tuning 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]);
```

Create MPC controller

Create an MPC controller with sample time `Ts`, prediction horizon `p`, and control horizon `m`.

```
Ts = 0.1;
p = 10;
m = 2;
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Simulate Controller Using `sim` Function

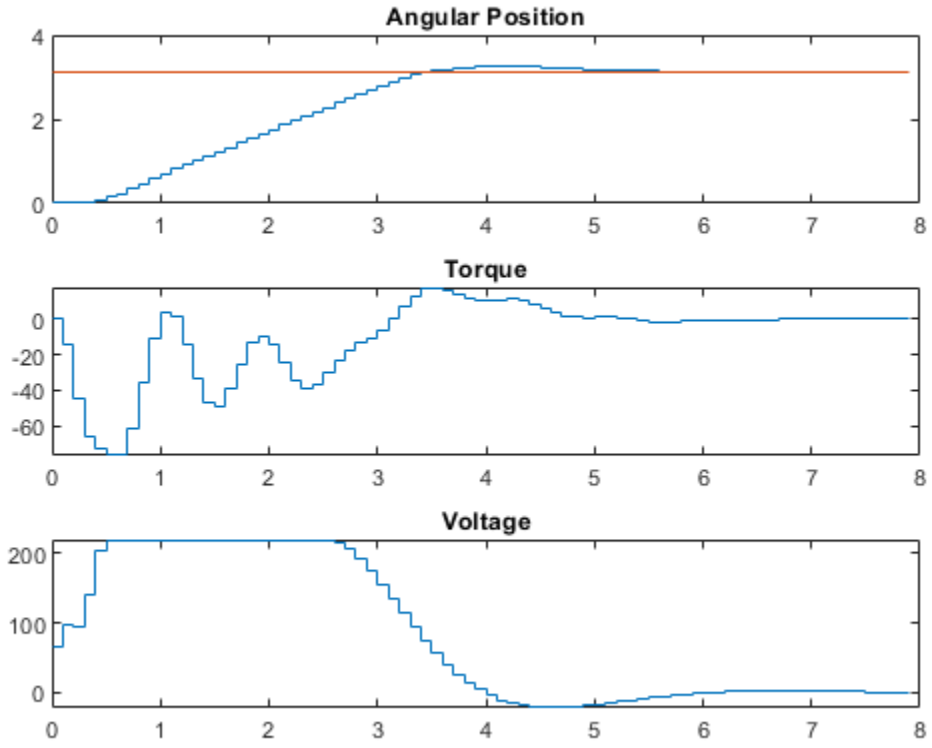
Use the `sim` function to simulate the closed-loop control of the linear plant model in MATLAB.

```
disp('Now simulating nominal closed-loop behavior');
Tstop = 8;                          % seconds
Tf = round(Tstop/Ts);               % simulation iterations
r = [pi*ones(Tf,1) zeros(Tf,1)];% reference signal
[y1,t1,u1] = sim(mpcobj,Tf,r);
```

```
Now simulating nominal closed-loop behavior
-->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 measured
```

Plot results.

```
subplot(3,1,1)
stairs(t1,y1(:,1))
hold on
stairs(t1,r(:,1))
hold off
title('Angular Position')
subplot(3,1,2)
stairs(t1,y1(:,2))
title('Torque')
subplot(3,1,3)
stairs(t1,u1)
title('Voltage')
```



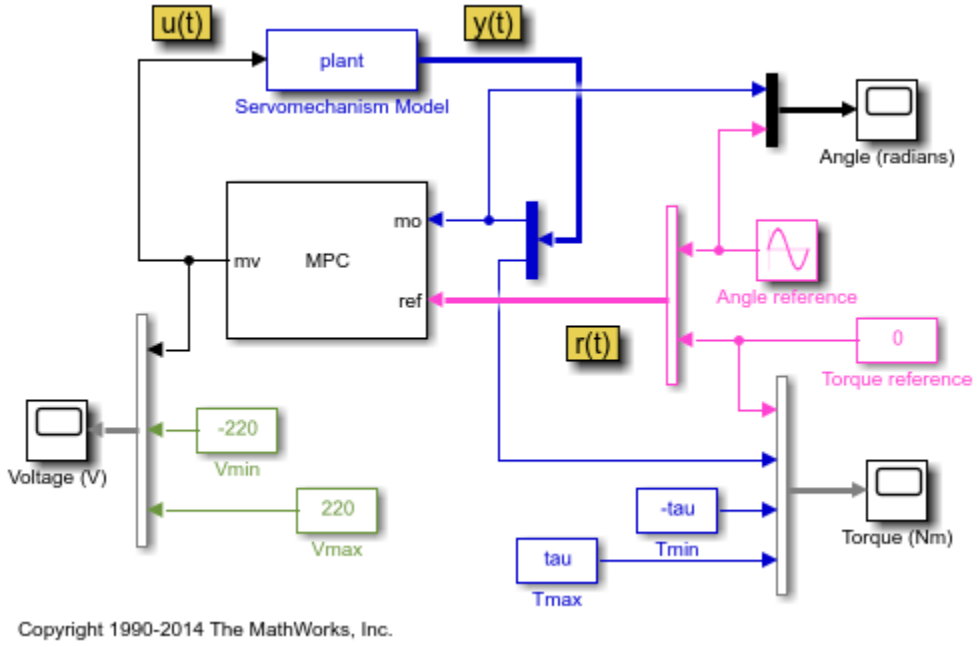
Simulate Using Simulink

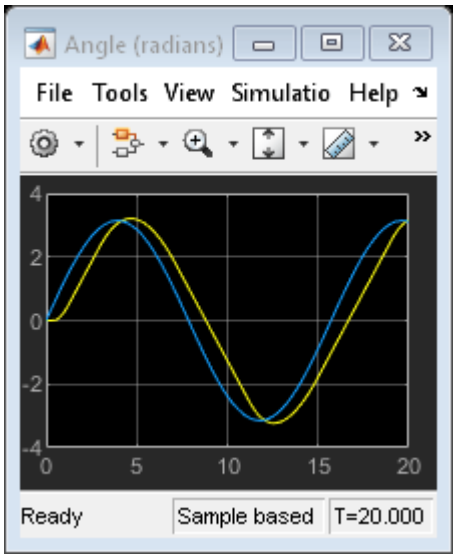
To run this example, Simulink® is required.

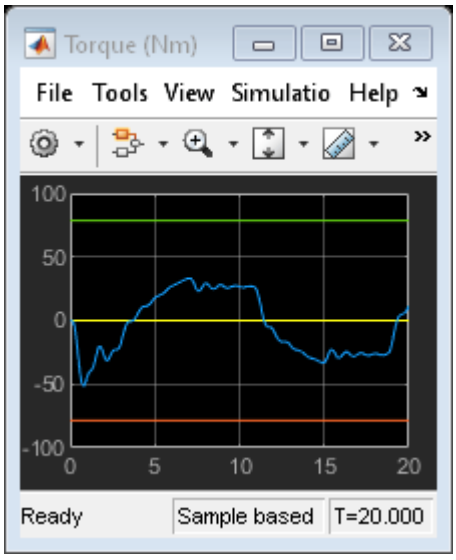
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

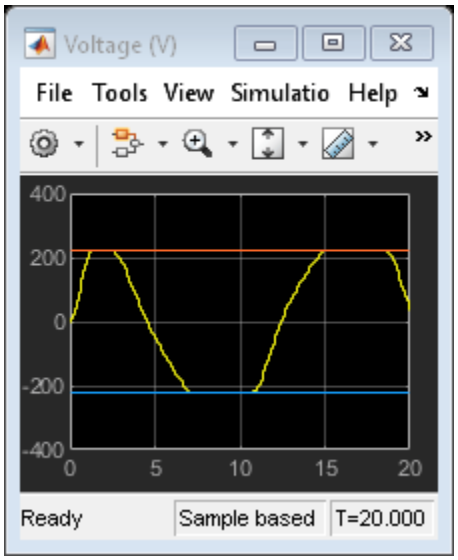
Simulate closed-loop control of the linear plant model in Simulink. The MPC Controller block is configured to use `mpcobj` as its controller.

```
mdl = 'mpc_motor';
open_system(mdl)
sim(mdl)
```




Angle (radians)
File Tools View Simulatio Help
4
2
0
-2
-4
0 5 10 15 20
Ready
Sample based
T=20.000


Torque (Nm)
File Tools View Simulatio Help
100
50
0
-50
-100
0 5 10 15 20
Ready
Sample based
T=20.000



The closed-loop response is identical to the simulation result in MATLAB.

References

[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.

```
bdclose(mdl)
```

See Also

More About

- “Specify Constraints” on page 1-5

Discrete Control Set MPC

MPC problems with discrete control sets are problems in which some or all manipulated variables belong to discrete sets. To handle these cases, for a given manipulated variable, specify the `Type` field of the corresponding `ManipulatedVariables` structure in the `mpc` object:

- `'binary'` — Restrict the manipulated variable to be either 0 or 1.
- `'integer'` — Restrict the manipulated variable to be an integer.
- Vector containing a discrete set of possible values — Restrict the manipulated variable to the specified values, for example `mpcobj.MV(2).Type=[-1,0,0.5,1,2];`.

By default, the type is set to `'continuous'`, indicating that the manipulated variable is continuous.

You can simulate the discrete control set linear MPC controller in:

- MATLAB® — using `sim`, `mpcmove`, `mpcmoveAdaptive` and `mpcmoveMultiple`.
- Simulink® — using the MPC Controller, Adaptive MPC Controller, and Multiple MPC Controllers blocks.

When simulating multiple controllers using `mpcmoveMultiple` or the Multiple MPC Controllers block, all candidate controllers must use the same manipulated variable type configuration.

Code generation from a controller with discrete control sets is supported in both MATLAB and Simulink.

A new built-in mixed-integer quadratic programming (MIQP) solver is used to solve the discrete control set MPC problem. You can use the new property `Optimizer.MixedIntegerOptions` of the `mpc` object to customize the options for this solver (like for example number of iterations and constraints tolerance).

See Also

Related Examples

- “Solve a Discrete Set MPC Problem in MATLAB” on page 1-16
- “Solve a Discrete Set MPC Problem in Simulink” on page 1-21
- “Surge Tank Control Using Discrete Control Set MPC” on page 1-24

Solve a Discrete Set MPC Problem in MATLAB

This example shows how to solve, in MATLAB, an MPC problem in which some manipulated variables belong to a discrete set.

Create a Plant Model

Fix the random generator seed for reproducibility.

```
rng(0);
```

Create a discrete-time strictly proper plant with 4 states, two inputs and one output.

```
plant = drss(4,1,2);
plant.D = 0;
```

Set the sampling time to 0.1s, and increase the control authority of the first input, to better illustrate its control contribution.

```
plant.Ts = 0.1;
plant.B(:,1)=plant.B(:,1)*2;
```

Design the MPC Controller

Create an MPC controller with one second sampling time, `20` steps prediction horizon and `5` steps control horizon.

```
mpcobj = mpc(plant,0.1,20,5);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify the first manipulated variable as belonging to a discrete set of seven possible values (you could also specify the type as an integer using the instruction `mpcobj.MV(1).Type = 'integer';`)

```
mpcobj.MV(1).Type = [-1 -0.7 -0.3 0 0.2 0.5 1];
```

Use rate limits to enforce maximum increment and decrement values for the first manipulated variable.

```
mpcobj.MV(1).RateMin = -0.5;
mpcobj.MV(1).RateMax = 0.5;
```

Set limits on the second manipulated variable, whose default type (continuous) has not been changed.

```
mpcobj.MV(2).Min = -2;
mpcobj.MV(2).Max = 2;
```

Simulate the Closed Loop Using the `sim` Command and Plot Results

Set the number of simulation steps.

```
simsteps = 50;
```

Create an output reference signal equal to zero from steps `20` to `35` and equal to `0.6` before and after.

```
r = ones(simsteps,1)*0.6;
r(20:35) = 0;
```

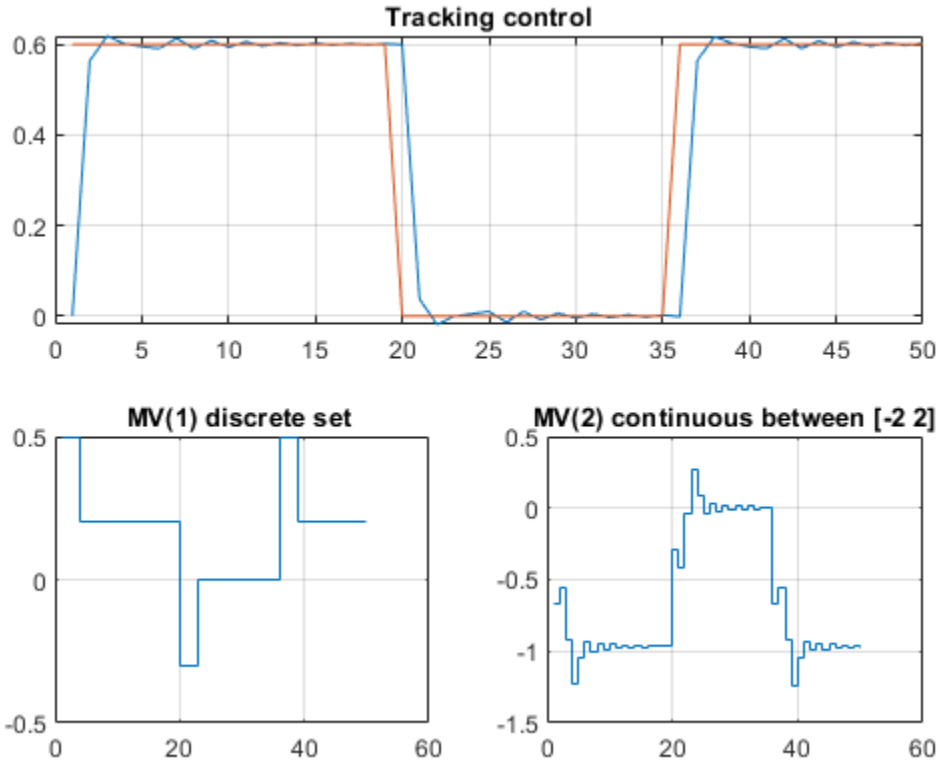
Simulate the closed loop using the `sim` command. Return the plant input and output signals.

```
[YY,~,UU,~,~,~,status] = sim(mpcobj,simsteps,r);
```

```
-->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 measured
```

Plot results.

```
figure(1)

subplot(211)    % plant output
plot([YY,r]);
grid
title("Tracking control");

subplot(223)    % first plant input
stairs(UU(:,1));
grid
%title("MV(1) discrete set "+ num2str(mpcobj.MV(1).Type','%0.1f '))
title("MV(1) discrete set ")

subplot(224)    % second plant input
stairs(UU(:,2));
grid
title("MV(2) continuous between [-2 2]")
```



As expected, the first manipulated variable is restricted to the values specified in the discrete set (with jumps less than the specified limit), while the second one can vary continuously between `-2` and 2. The plant output tracks the reference value after a few seconds.

Simulate the Closed Loop Using the `mpcmove` Command and Plot Results

Get handle to `mpcobj` state and initialize plant state.

```
xmpc = mpcstate(mpcobj);
x = xmpc.Plant;
```

Initialize arrays that store signals.

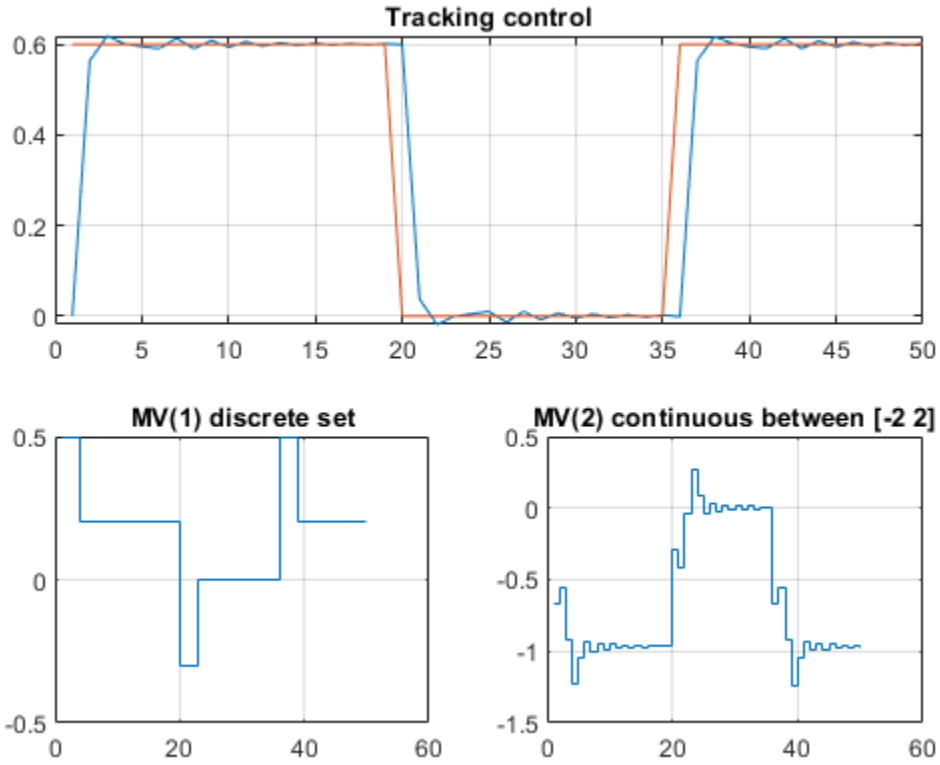
```
YY = []; RR = []; UU = []; XX = [];
```

Perform simulation using the `mpcmove` command to calculate the control actions.

```
for k = 1:simsteps
    XX = [XX;x']; % store plant state
    y = plant.C*x; % calculate plant output
    YY = [YY;y]; % store plant output
    RR = [RR;r(k)]; % store reference
    u = mpcmove(mpcobj,xmpc,y,r(k)); % calculate optimal mpc move
    UU = [UU;u']; % store plant input
    x = plant.A*x+plant.B*u; % update plant state
    % is the last line necessary since x=xmpc.Plant gets updated anyway?
end
```

Plot results.

```
figure(2)

subplot(211)    % plant output
plot([YY,r]);
grid
title("Tracking control");

subplot(223)    % first plant input
stairs(UU(:,1));
grid
title("MV(1) discrete set")

subplot(224)    % second plant input
stairs(UU(:,2));
grid
title("MV(2) continuous between [-2 2]")
```



The simulation results are identical as the ones achieved using the `sim` command.

See Also

“Discrete Control Set MPC” on page 1-15

Related Examples

- “Solve a Discrete Set MPC Problem in Simulink” on page 1-21
- “Surge Tank Control Using Discrete Control Set MPC” on page 1-24

Solve a Discrete Set MPC Problem in Simulink

This example shows how to solve, using Simulink, an MPC problem in which some manipulated variables belong to a discrete set.

Create a Plant Model

Fix the random generator seed for reproducibility.

```
rng(0);
```

Create a discrete-time strictly proper plant with 4 states, two inputs and one output.

```
plant = drss(4,1,2);
plant.D = 0;
```

Increase the control authority of the first input, to better illustrate its control contribution.

```
plant.B(:,1)=plant.B(:,1)*2;
```

Design the MPC Controller

Create an MPC controller with one second sampling time, `20` steps prediction horizon and `5` steps control horizon.

```
mpcobj = mpc(plant,0.1,20,5);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify the first manipulated variable as belonging to a discrete set of seven possible values. Note that you Could also specify it as an integer using the instruction `mpcobj.MV(1).Type = 'integer';` in which case the first manipulated variable will be constrained to be an integer.

```
mpcobj.MV(1).Type = [-1 -0.7 -0.3 0 0.2 0.5 1];
```

Use rate limits to enforce maximum increment and decrement values for the first manipulated variable.

```
mpcobj.MV(1).RateMin = -0.5;
mpcobj.MV(1).RateMax = 0.5;
```

Set limits on the second manipulated variable, whose default type (continuous) has not been changed.

```
mpcobj.MV(2).Min = -2;
mpcobj.MV(2).Max = 2;
```

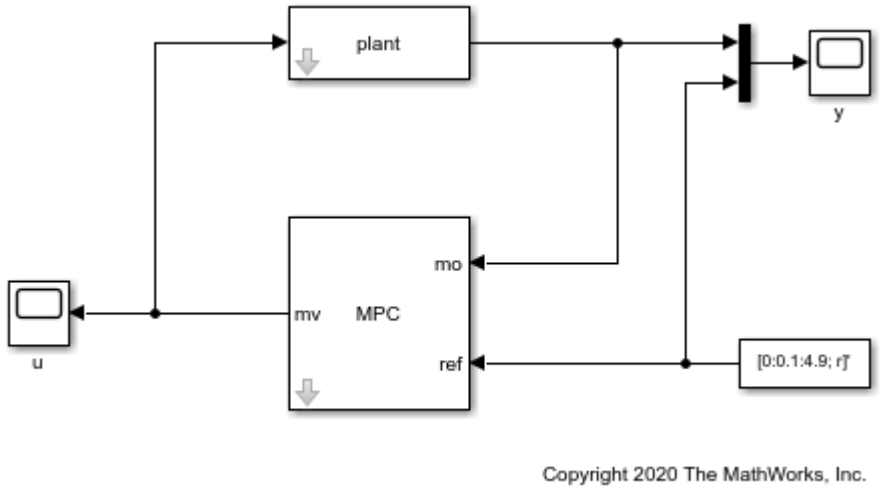
Control the Plant Model in Simulink

Create an output reference signal equal to zero from steps `20` to `35` and equal to `0.6` before and after.

```
r = ones(1,50)*0.6;
r(20:35) = 0;
```

Create a Simulink closed loop simulation using the **MPC Controller** block, with the `mpcobj` object passed as a parameter, to control the double integrator plant. For this example, open the pre-existing Simulink model `dcsdemo.slxc`.

```
open('dcsdemo.slx')
```

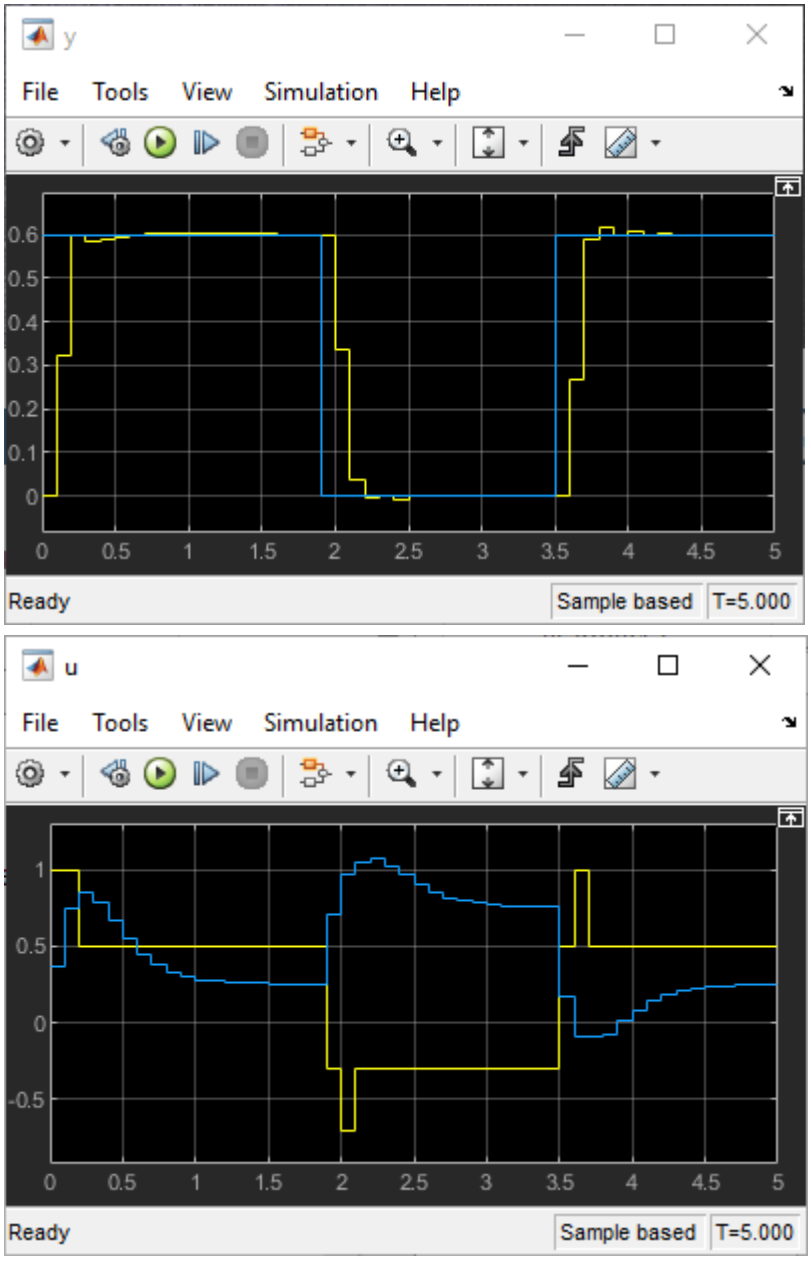


You can now run the model by clicking **Run** or by using the MATLAB command `sim`.

```
sim('dcsdemo.slx')
```

```
-->No sample time provided for plant model. Assuming sample time = controller's sample time = 0.1
-->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 measured

ans =
  Simulink.SimulationOutput:

                  tout: [51x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```

After the simulation, the plots of the two scopes show that the manipulated variable does not exceed the limit and the plant output tracks the reference signal after approximately half a second.



See Also

"Discrete Control Set MPC" on page 1-15

Related Examples

- "Solve a Discrete Set MPC Problem in MATLAB" on page 1-16
- "Surge Tank Control Using Discrete Control Set MPC" on page 1-24

Surge Tank Control Using Discrete Control Set MPC

This example shows how to use a linear MPC controller with both continuous- and discrete-set control actions to control the level of a surge tank in Simulink®.

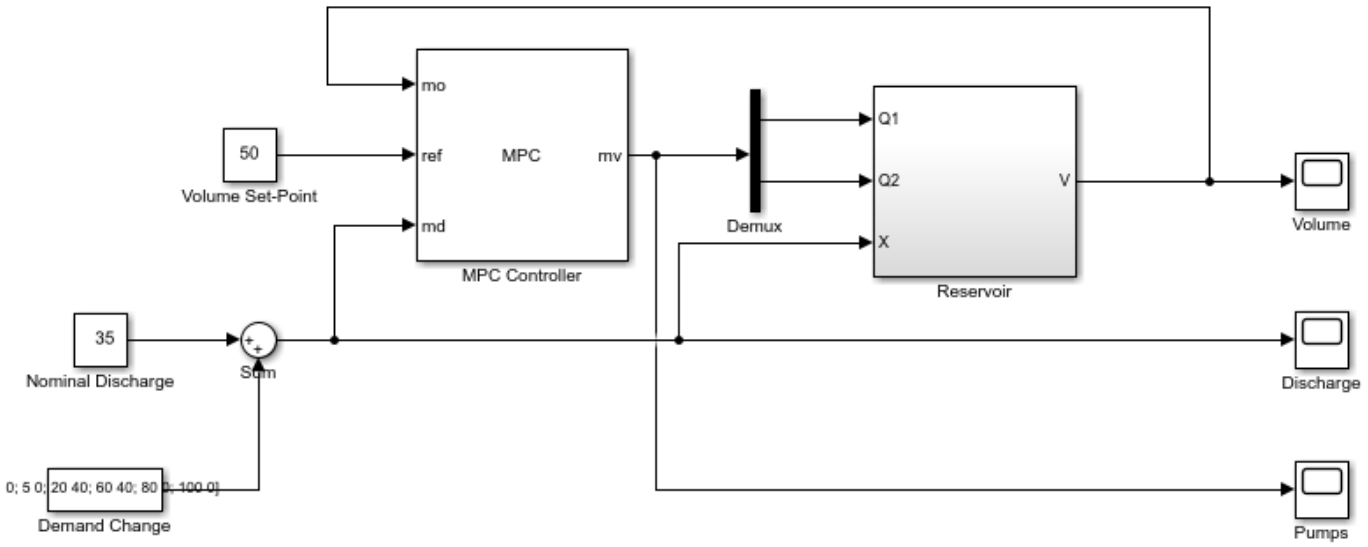
Overview

Many petrochemical processes include surge capacity to insulate downstream operations from upsets in upstream flows. This example considers a liquid surge tank supplied by two pumps.

- Pump 1 is variable-speed and can deliver a flow rate between 0 and 100 L/min. The rate of change of the flow rate is limited to 50 L/min per minute.
- Pump 2 is on-off and delivers 100 L/min when it is on.

This system also has a continuously adjustable valve that regulates the tank discharge to accommodate a downstream process.

```
mdl = 'ReservoirMPC';
open_system(mdl)
```



The control objective is to use the two pumps (manipulated variables) to maintain the tank volume at its setpoint when the discharge valve introduces a disturbance to the surge tank (measured disturbance).

When downstream demand is constant, pump 1 controls the surge tank level, ideally keeping it near 50%. When a disturbance occurs, both pumps can go into action to maintain the tank volume between 25% and 75% during the transient time. When downstream demand is high, however, Pump 2 must turn on.

In summary, the MPC controller manages the two pumps such that:

- The tank level stays near 50% when demand is constant.

- The tank level stays within the ideal range when Pump 2 turns on and off. Rapid Pump 2 on-off cycling is undesirable.

Create Linear Plant from Nonlinear Surge Tank Model

In the Simulink model, the surge tank model is in the Reservoir subsystem. It implements the following equations.

- `dV/dt = Qin - Qout`, where `V` is the tank volume between 0 and 100 (percent)
- `Qin = Q1 + Q2`, the sum of the two pump flow rates (L/min)
- `Qout = 0.2*sqrt(2)*X*sqrt(V)`, the discharge rate (L/min)
- `X`, the discharge valve, which can open between 0 and 100 (percent)

The plant starts running at its nominal steady-state operating point: pump 1 runs at 70 L/min, pump 2 is off, the discharge valve is at 35%, and the tank volume is at 50%. Create a linear model of the tank at this operating point. This model has three inputs: the manipulated variables `Q1` and `Q2`, and the measured disturbance|X|.

```
plant = ss(-0.7,[1 1 -2],1,[0 0 0]);
plant = setmpcsignals(plant,'MV',[1 2],'MD',3);
```

Design MPC Controller with Continuous and Discrete Control Actions

Create a linear MPC controller with a sample time of one second, default prediction and control horizons, and default cost function weights.

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

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Set the nominal values for the controller to match the steady-state operating point.

```
MPCobj.Model.Nominal.U = [70; 0; 35];
MPCobj.Model.Nominal.Y = 50;
MPCobj.Model.Nominal.X = 50;
MPCobj.Model.Nominal.DX = 0;
```

Choose the `MVRate` weights such that the controller adjusts pump 1 in preference to pump 2. That is, the controller penalizes pump 2 adjustments less than pump 1 adjustments.

```
MPCobj.Weights.MVrate = [0.1 0.2];
```

Specify safety bounds on the continuous input and output signals.

```
MPCobj.OV.Min = 0;
MPCobj.OV.Max = 100;
MPCobj.MV(1).Min = 0;
MPCobj.MV(1).Max = 100;
MPCobj.MV(1).RateMin = -50;
MPCobj.MV(1).RateMax = 50;
```

Since pump 2 has two discrete settings (0 and 100 L/min), specify a discrete set in the `Type` property of this controller. Depending on your application, the `Type` property can also be `binary` or `integer`.

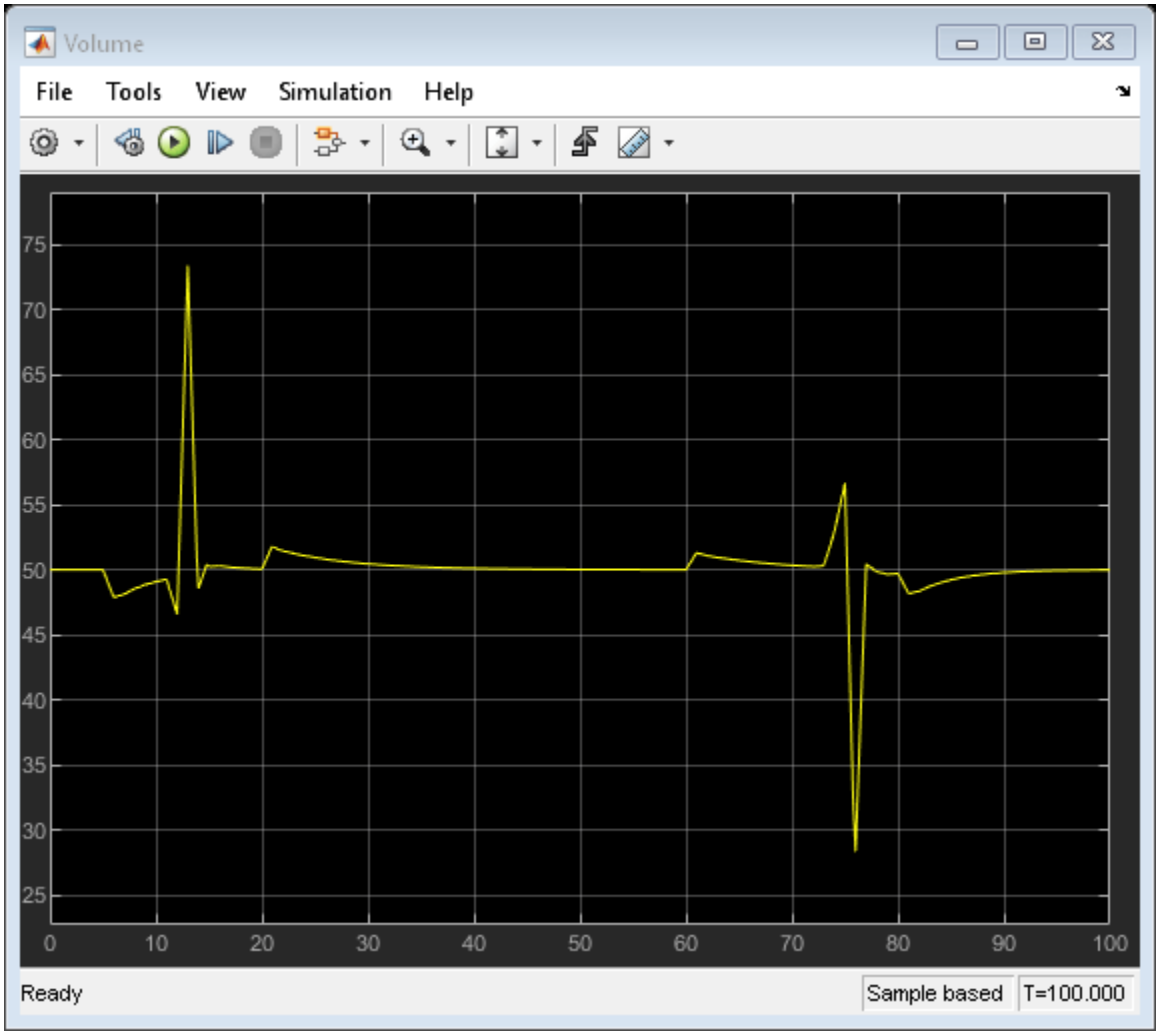
```
MPCobj.MV(2).Type = [0 100];
```

Simulate Closed-Loop Response with Discharge Disturbance

The simulation begins at the nominal condition. At t = 5, the discharge rate ramps up until pump 1 is at its full capacity and pump 2 must turn on. Pump 2 turns on at t = 12. Pump 1 must then decrease as rapidly as possible to keep the level in bounds, then establish a new steady-state operating point. At t = 60, the discharge begins to ramp down to the nominal state. The MPC controller keeps the volume within the ideal range, and pump 2 does not exhibit rapid cycling.

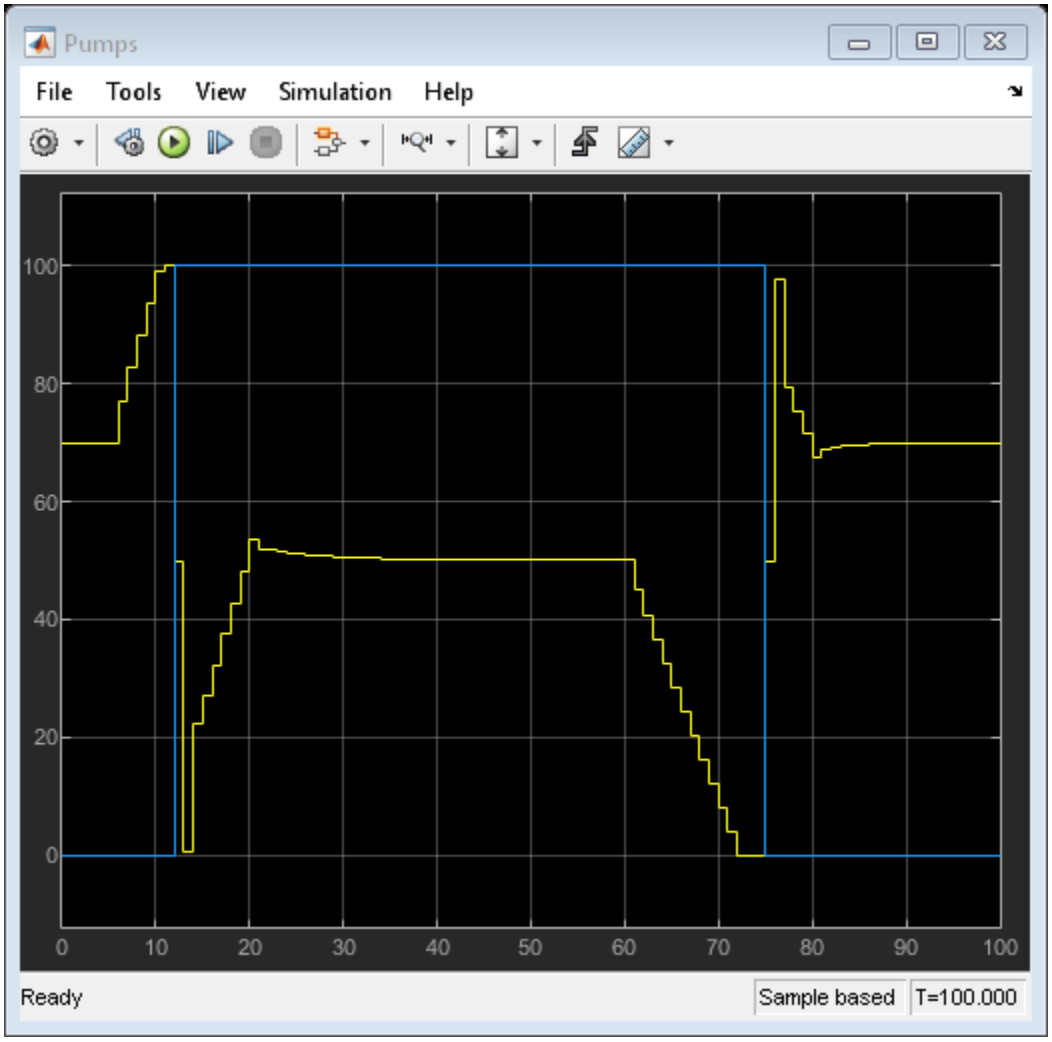
```
sim(mdl)
open_system([mdl '/Volume'])
```

```
-->Converting model to discrete time.
-->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 measured

ans =

  Simulink.SimulationOutput:
                  tout: [106x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```



The following figure shows the flow rates from two pumps.

```
open_system([mdl '/Pumps'])
```



As expected, the MPC controller turns pump 2 on and off.

```
bdclose(mdl) % close simulink model.
```

See Also

"Discrete Control Set MPC" on page 1-15

Related Examples

- "Solve a Discrete Set MPC Problem in MATLAB" on page 1-16
- "Solve a Discrete Set MPC Problem in Simulink" on page 1-21

Specify Scale Factors

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.

You can define scale factors at the command line and using the **MPC Designer** app.

Determine 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).

Specify Scale Factors at Command Line

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.

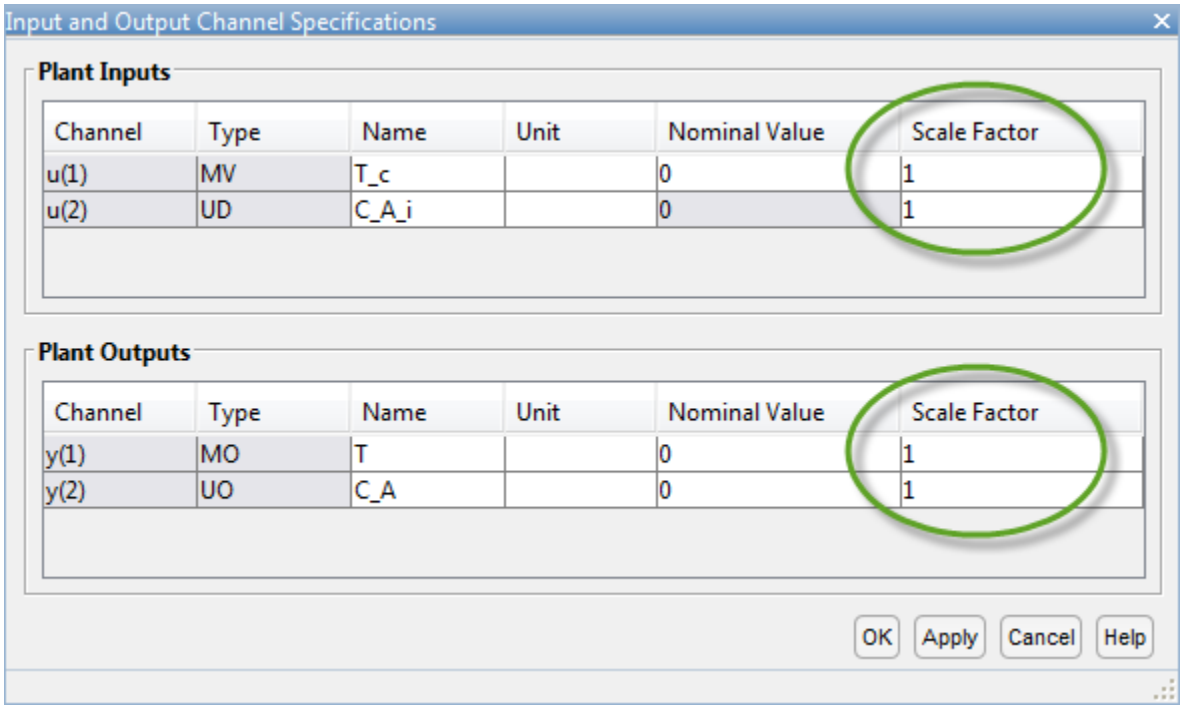
```
% Random plant for illustrative purposes: 5 inputs, 3 outputs
Plant = drss(4,3,5);
Plant.InputName = {'MV1','UD1','MV2','UD2','MD'};
Plant.OutputName = {'UO','MO1','MO2'};

% 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
```

Specify Scale Factors Using MPC Designer

After opening **MPC Designer** and defining the initial MPC structure, on 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.



To update the controller settings, click **OK**.

See Also

`mpc` | **MPC Designer**

More About

- “Choose Sample Time and Horizons” on page 1-2
- “Using Scale Factors to Facilitate MPC Weights Tuning” on page 1-32

Using Scale Factors to Facilitate MPC Weights Tuning

This example shows how to specify scale factors in an MPC controller to make weights tuning easier.

Define Plant Model

The discrete-time, linear, state-space plant model has 10 states, 5 inputs, and 3 outputs.

```
[plant,Ts] = mpcscalefactor_model;
[ny,nu] = size(plant.D);
```

The plant inputs include manipulated variable (MV), measured disturbance (MD) and unmeasured disturbance (UD). The plant outputs include measured outputs (MO) and unmeasured outputs (UO).

```
MVindex = [1, 3, 5];
MDindex = 4;
UDindex = 2;
MOindex = [1 3];
UOindex = 2;
plant = setmpcsignals(plant,'MV',MVindex,'MD',MDindex,'UD',UDindex,'MO',MOindex,'UO',UOindex);
```

The nominal values and operating ranges of plant model are as follows:

- Input 1: manipulated variable, nominal value is 100, range is [50 150]
- Input 2: unmeasured disturbance, nominal value is 10, range is [5 15]
- Input 3: manipulated variable, nominal value is 0.01, range is [0.005 0.015]
- Input 4: measured disturbance, nominal value is 0.1, range is [0.05 0.15]
- Input 5: manipulated variable, nominal value is 1, range is [0.5 1.5]
- Output 1: measured output, nominal value is 0.01, range is [0.005 0.015]
- Output 2: unmeasured output, nominal value is 1, range is [0.5 1.5]
- Output 3: measured output, nominal value is 100, range is [50 150]

Define and Analyze Open-Loop Plant Signals

Use `lsim` command to run an open loop linear simulation to verify that plant outputs are within the range and their average are close to the nominal values when input signals vary randomly around their nominal values.

```
% set input and output range
Urange = [100;10;0.01;0.1;1];
Yrange = [0.01;1;100];

% set input and output nominal values
% for this example nominal values are equivalent to ranges
Unominal = [100;10;0.01;0.1;1];
Ynominal = [0.01;1;100];

% define time intervals
t = (0:1000)'*Ts;
nt = length(t);

% define input and output signals
Uol = (rand(nt,nu)-0.5).*(ones(nt,1)*Urange');   % design input signal
Yol = lsim(plant,Uol,t);                         % compute plant output
```

```
fprintf('The difference between average output values and the nominal values are %.2f%%, %.2f%%,
    abs(mean(Yol(:,1)))/Ynominal(1)*100,abs(mean(Yol(:,2)))/Ynominal(2)*100,abs(mean(Yol(:,3)))/Y
```

```
The difference between average output values and the nominal values are 2.25%, 3.53%, 2.47% respe
```

Display means and standard deviations of input and output signals.

```
% input
mean(Uol)
```

```
ans = 1×5

    0.6682   -0.1202   -0.0000    0.0009    0.0025
```

```
std(Uol)
```

```
ans = 1×5

   28.3302    2.9136    0.0029    0.0281    0.2805
```

```
% output
mean(Yol)
```

```
ans = 1×3

   -0.0002    0.0353    2.4706
```

```
std(Yol)
```

```
ans = 1×3

    0.0038    0.5615   39.6285
```

Hrer, the mean value give a good idea of the actual nominal values and the standard deviation capture some idea of th range.

Evaluate MPC with Default MPC Weights

When plant input and output signals have different orders of magnitude, default MPC weight settings often give poor performance.

Create an MPC controller with default weights:

- `Weight.MV = 0`
- `Weight.MVRate = 0.1`
- `Weight.OV = 1`

```
% create mpc object
mpcobjUnscaled = mpc(plant);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.0000
```

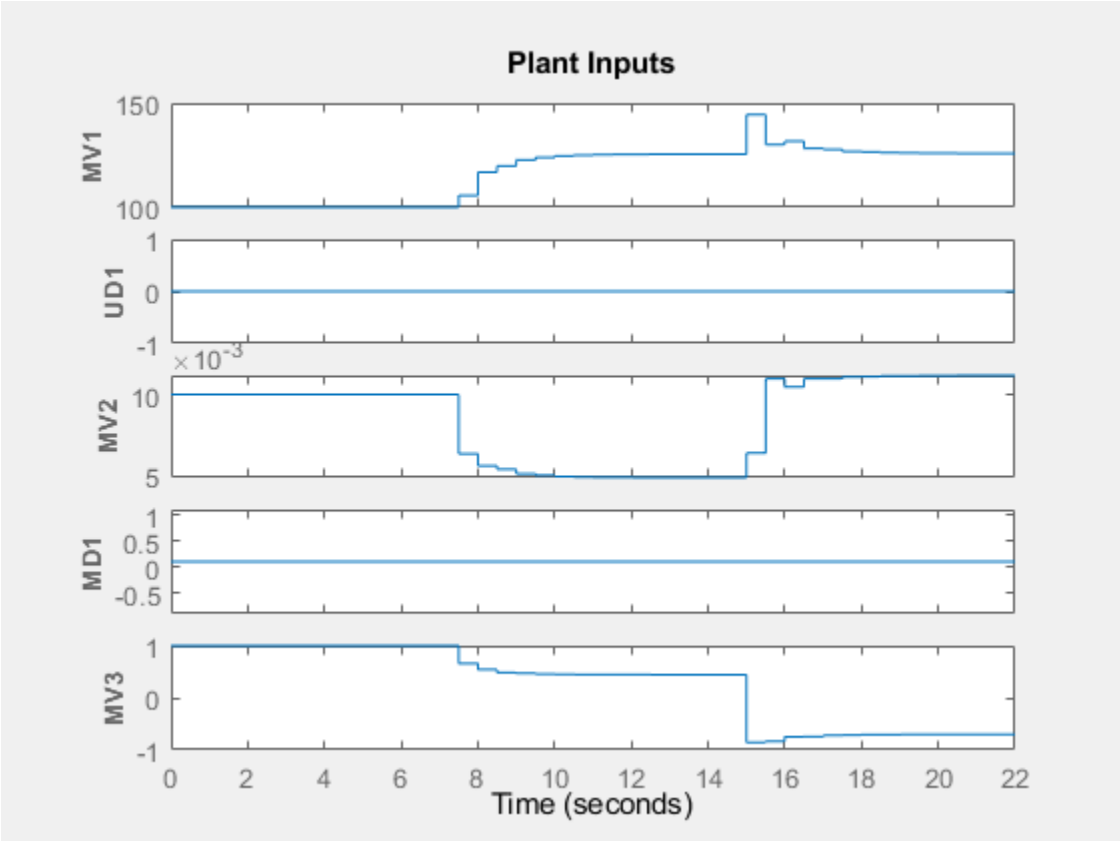
```
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.

% nominal values of the plant states
Xnominal = zeros(10,1);

% nominal values for unmeasured disturbance
Unominal(UDindex) = 0;  % Nominal values for unmeasured disturbance must be 0

% Set nominal values
mpcobjUnscaled.Model.Nominal = struct('X',Xnominal,'DX',Xnominal,'Y',Ynominal,'U',Unominal);
```

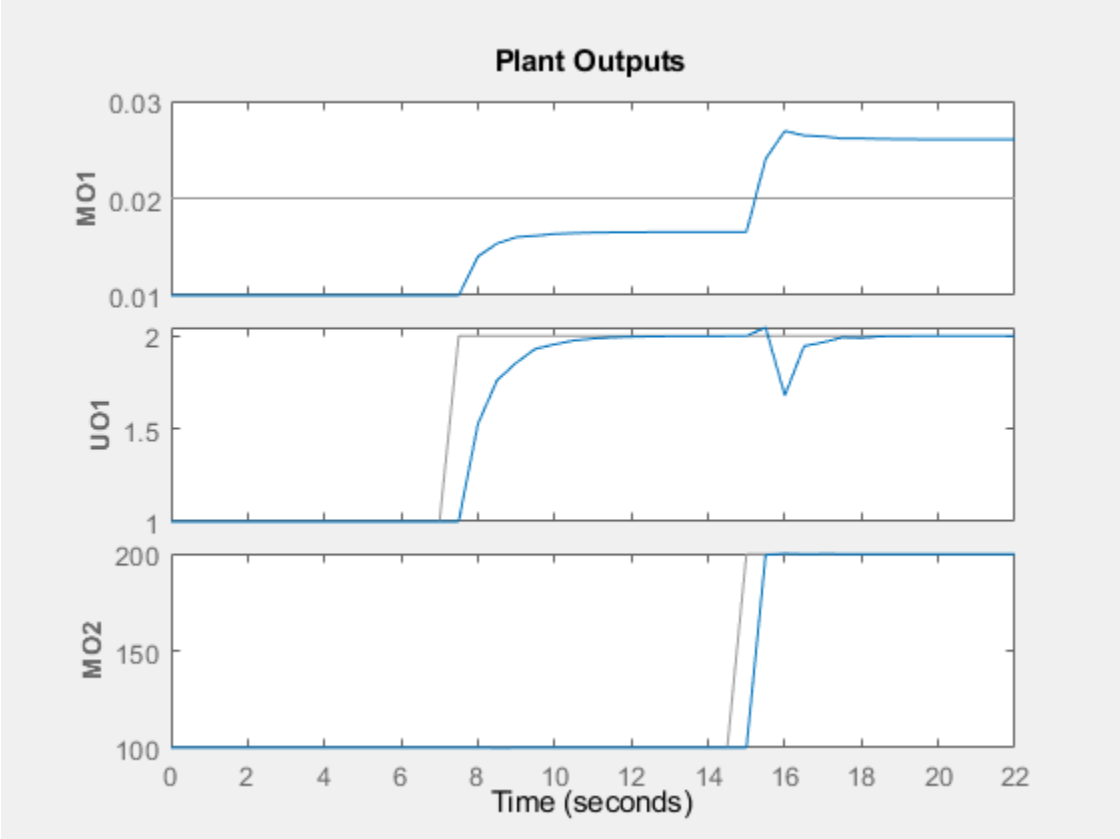
To calculate plant outputs, `sim` will subtract the nominal plant inputs from the inputs, and the nominal states from the current states, then apply the linear plant equations, and finally add the nominal output values to the calculated output.

To calculate the manipulated variables, it will remove the nominal output from the plant output signal, calculate the MPC control sequence, and add the nominal value of the manipulated variables to the calculated sequence.

First, test a sequence of step setpoint changes in three reference signals.

```
nStepLen = 15;                      % expected step response duration
Ns1 = nStepLen*ny;                  % calculate simulation time to accommodate ny steps
r1 = ones(Ns1,1)*Ynominal(:)';      % reference signal

% cycle through each output and define references at StepLen intervals
StepTime = 1;
for i = 1:ny
    r1(StepTime:end,i) = r1(StepTime:end,i) + Yrange(i);
    StepTime = StepTime + nStepLen;
end

% simulate closed loop for Ns1 steps and subject to reference r1
sim(mpcobjUnscaled,Ns1,r1)

-->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 noise.
   Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

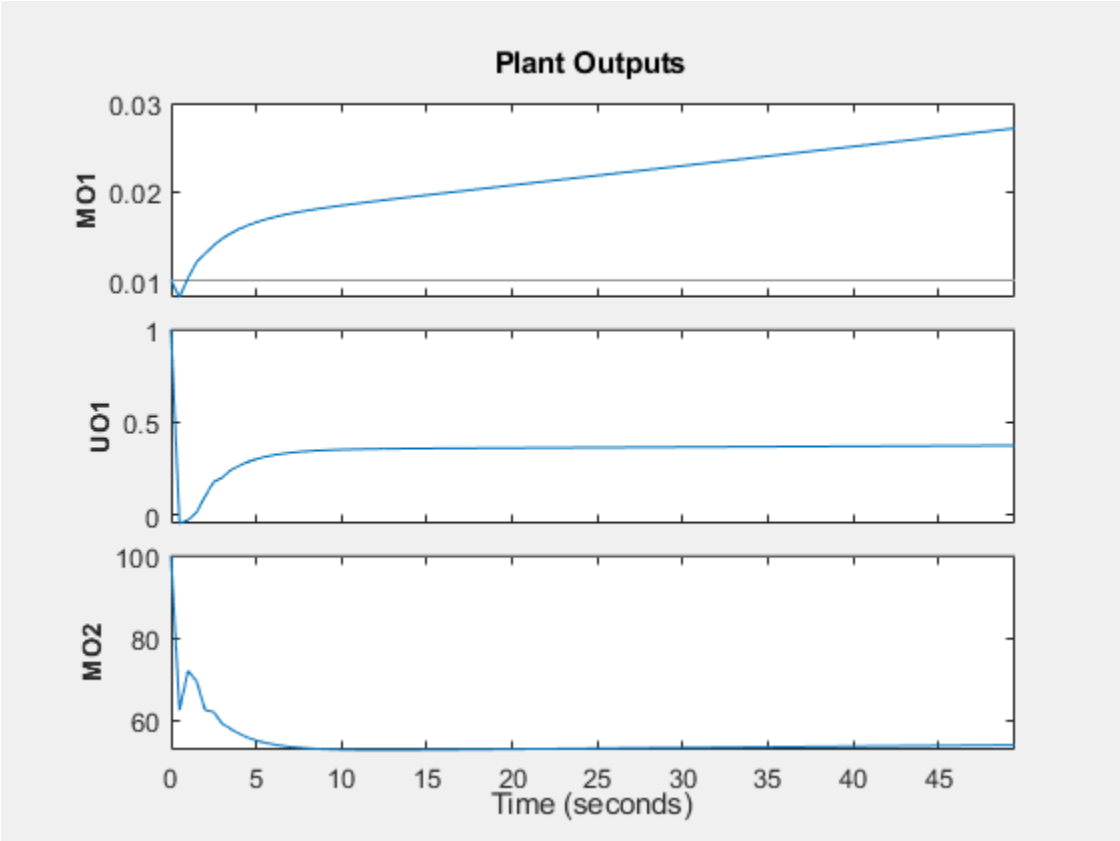

Plant Inputs
MV1
150
100
UD1
1
0
-1
×10^-3
MV2
10
5
MD1
1
0.5
0
-0.5
MV3
1
0
-1
0
2
4
6
8
10
12
14
16
18
20
22
Time (seconds)

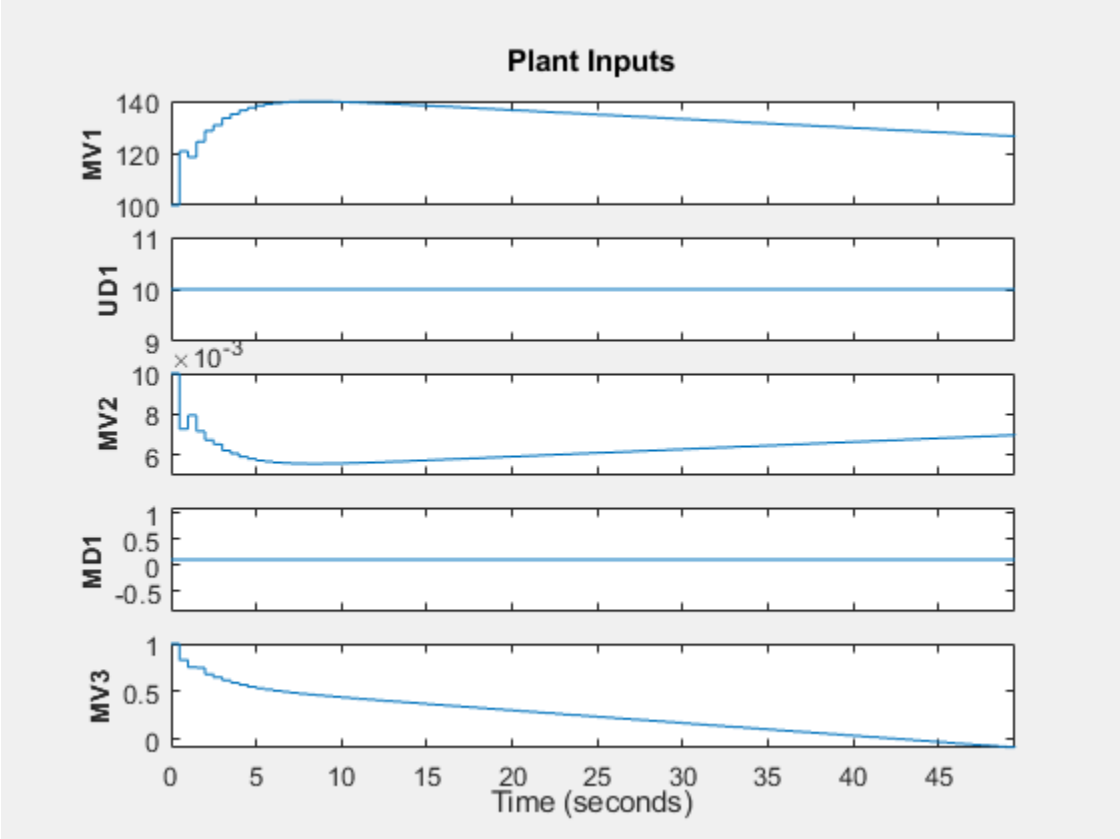


The tracking response of the first output is poor. The reason is that its range is small compared to the other outputs. If the default controller tuning weights are used, the MPC controller does not pay much attention to regulating this output because the associated penalty is so small compared to the other outputs in the objective function.

Second, test the unmeasured disturbance rejection.

```
% create simulation options object and set unmeasured disturbance
SimOpt = mpcsimopt;
SimOpt.UnmeasuredDisturbance = Urange(UDindex)';

% set number of simulation steps and reference signal
% reference signal is equal to the range values
Ns2 = 100;
r2 = ones(Ns2,1)*Yrange(:)';

% simulate the closed loop subject to reference r2
sim(mpcobjUnscaled,Ns2,r2,[],SimOpt)
```


Plant Outputs
MO1
0.03
0.02
0.01
UO1
1
0.5
0
MO2
100
80
60
0
5
10
15
20
25
30
35
40
45
Time (seconds)



The disturbance rejection response is also poor. None of the outputs return to their setpoints.

Evaluate MPC with Default MPC Weights After Specifying Scale Factors

Specifying input and output scale factors for the MPC controller:

- Improves the numerical quality of the optimization and state estimation calculations.
- Makes it more likely that the default tuning weights will achieve good controller performance.

Copy the MPC controller with default weights.

```
mpcobjScaled = mpcobjUnscaled;
```

To specify scale factors, it is good practice to use the expected operating range of each input and output.

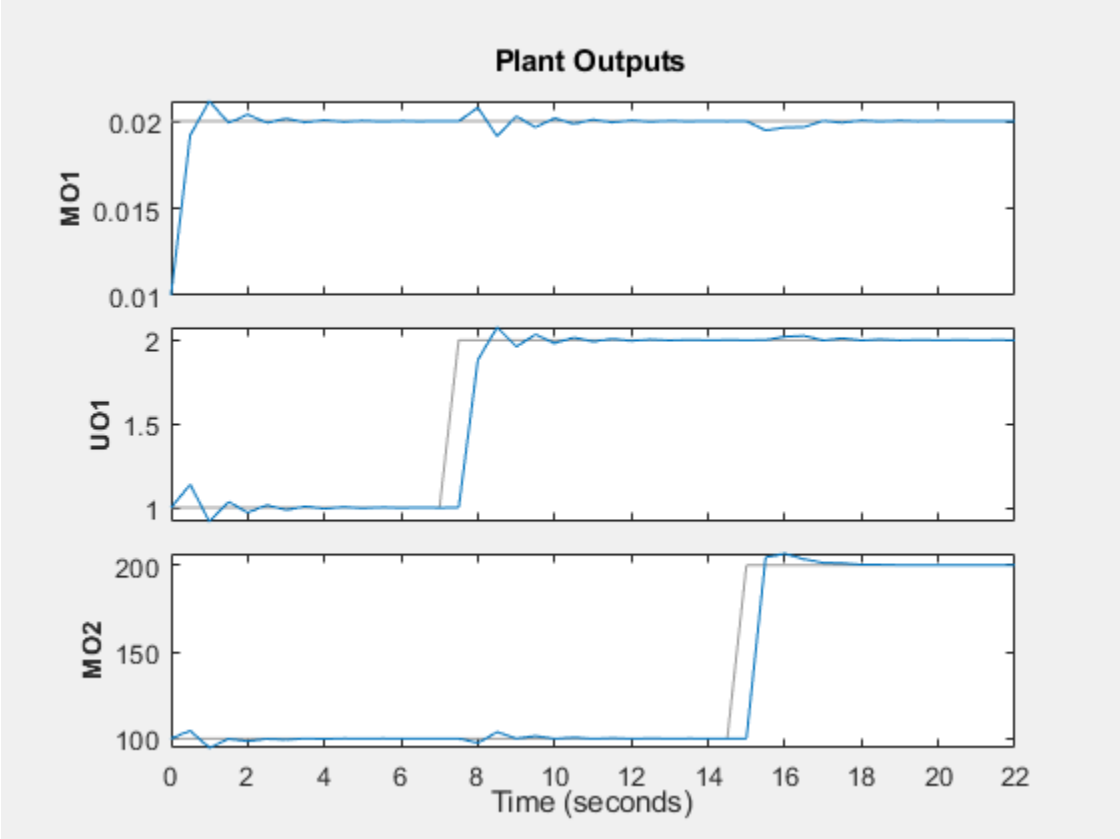
```
% scale manipulated variables
for i = 1:length(MVindex)
    mpcobjScaled.ManipulatedVariables(i).ScaleFactor = Urange(MVindex(i));
end

% scale measured disturbances
nmd = length(MDindex);
for i = 1:nmd
    mpcobjScaled.DisturbanceVariables(i).ScaleFactor = Urange(MDindex(i));
end
```

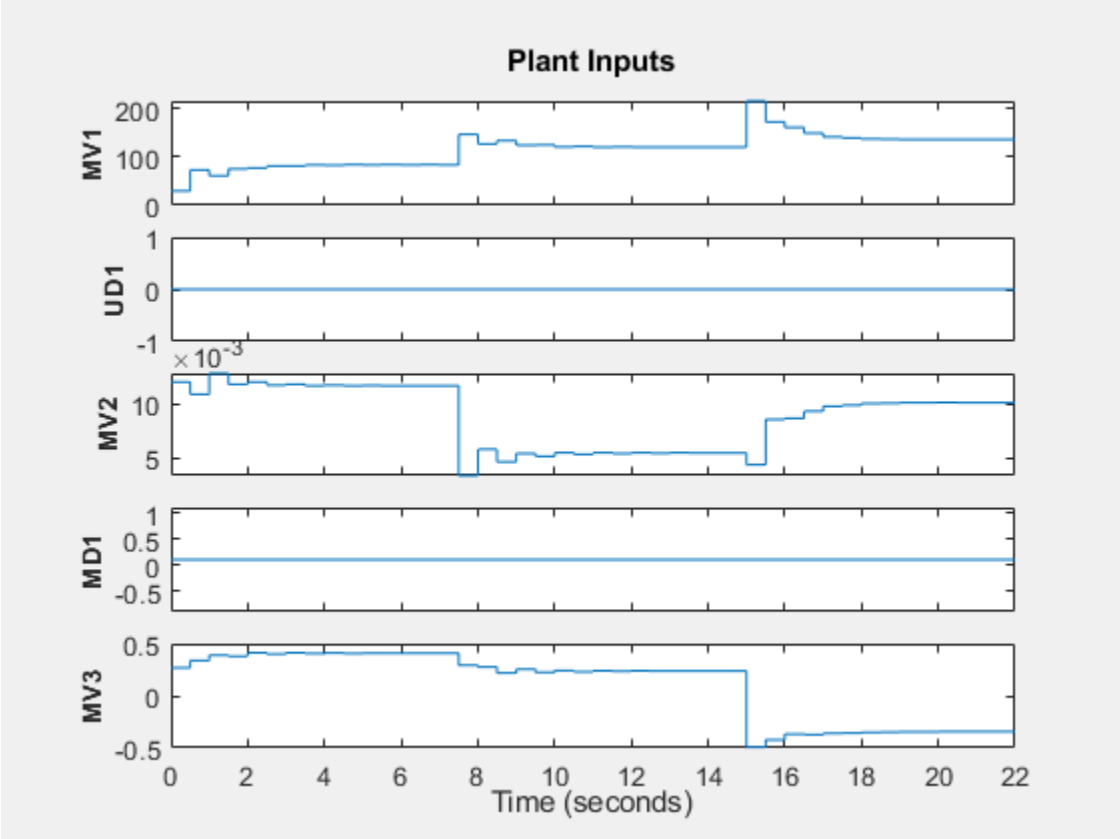
```
% scale unmeasured disturbances
for i = 1:length(UDindex)
    mpcobjScaled.DisturbanceVariables(i+nmd).ScaleFactor = Urange(UDindex(i));
end

% scale outputs
for i = 1:ny
    mpcobjScaled.OV(i).ScaleFactor = Yrange(i);
end
```

Repeat the first test, which is a sequence of step setpoint changes in three reference signals.

```
sim(mpcobjScaled,Ns1,r1)
```

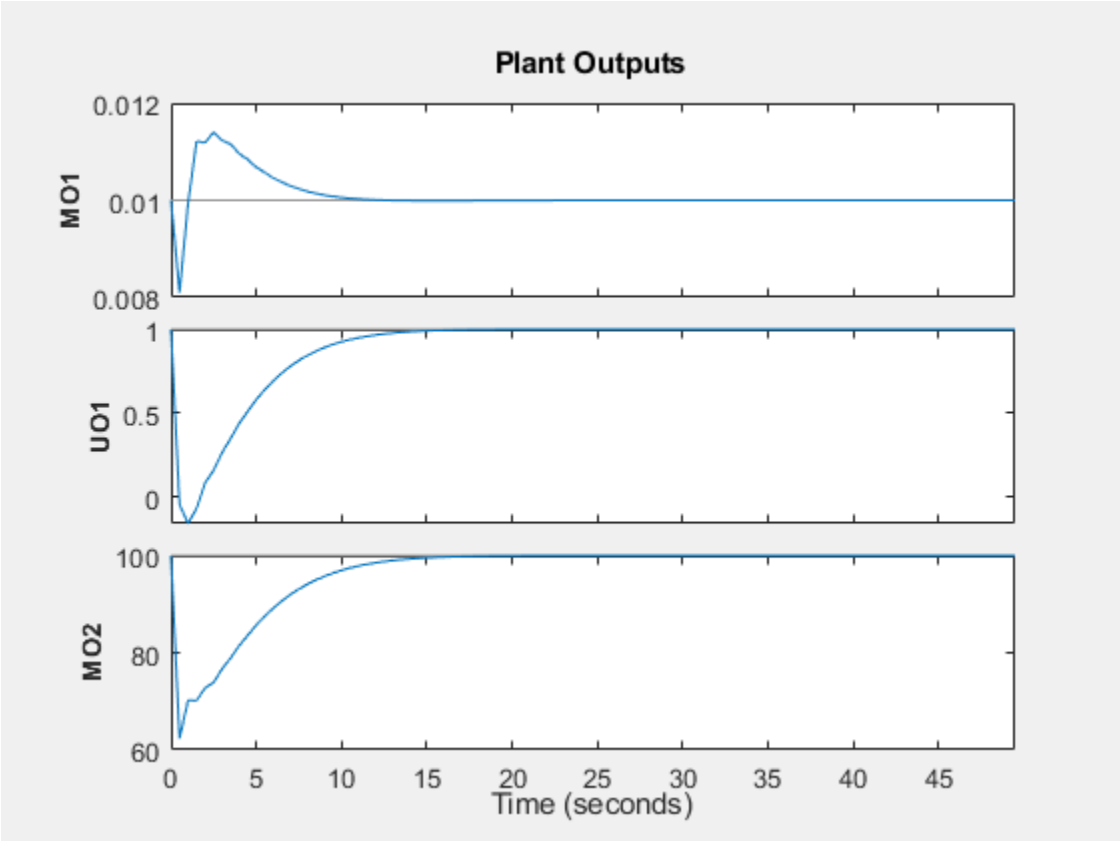
```
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #2 is integrated white noise.
-->Assuming output disturbance added to measured output channel #1 is integrated white noise.
   Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

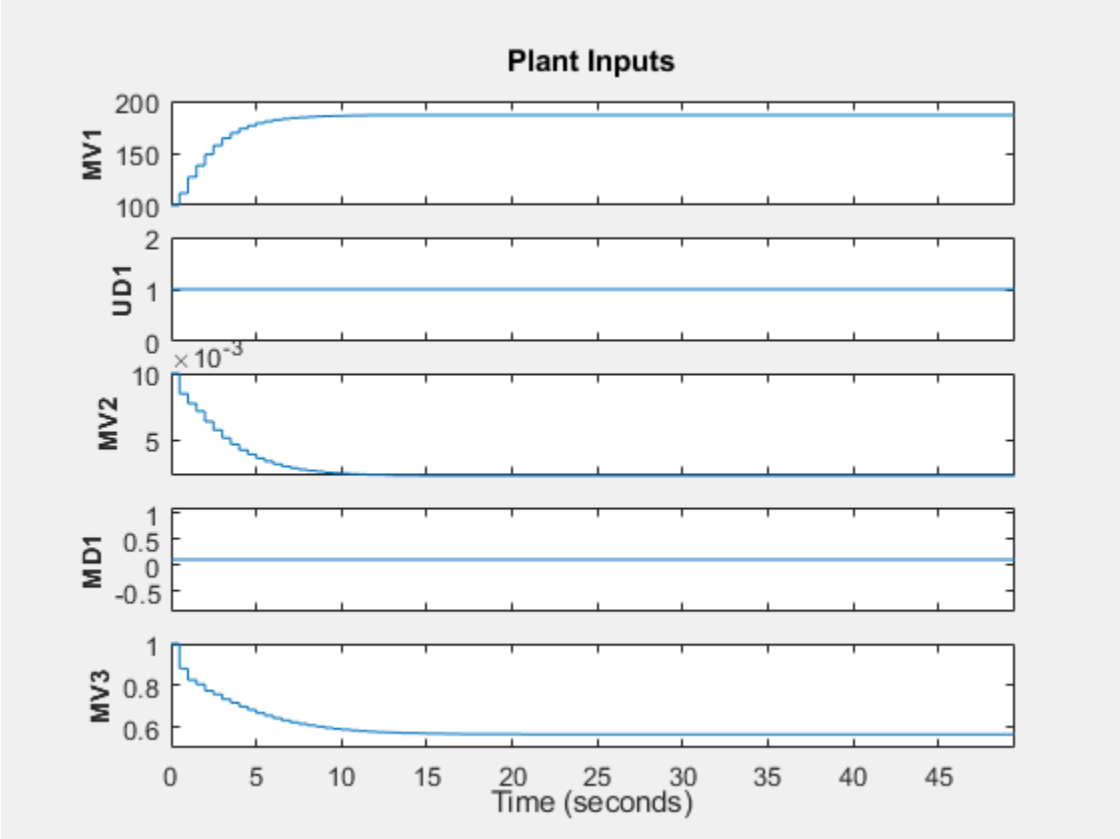




Repeat the second test, which is an unmeasured disturbance.

```
sim(mpcobjScaled,Ns2,r2,[],SimOpt)
```


Plant Outputs
MO1
0.012
0.01
0.008
UO1
1
0.5
0
MO2
100
80
60
0
5
10
15
20
25
30
35
40
45
Time (seconds)



Both setpoint tracking and disturbance rejection responses are good even without tuning MPC weights. This is because now the original weights apply to scaled signals, and therefore the weighting effect is not distorted.

See Also

`mpc` | **MPC Designer**

More About

- “Specify Scale Factors” on page 1-29

Tune Weights

A model predictive controller design usually requires some tuning of the cost function weights. This topic provides tuning tips. See “Optimization Problem” on page 2-7 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 “Specify Scale Factors” on page 1-29 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 “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42 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 ith 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 ith 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 MV Rate 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 1-7 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 "Adjust Disturbance and Noise Models" on page 3-25 for tests focusing on the disturbance rejection ability of the controller.

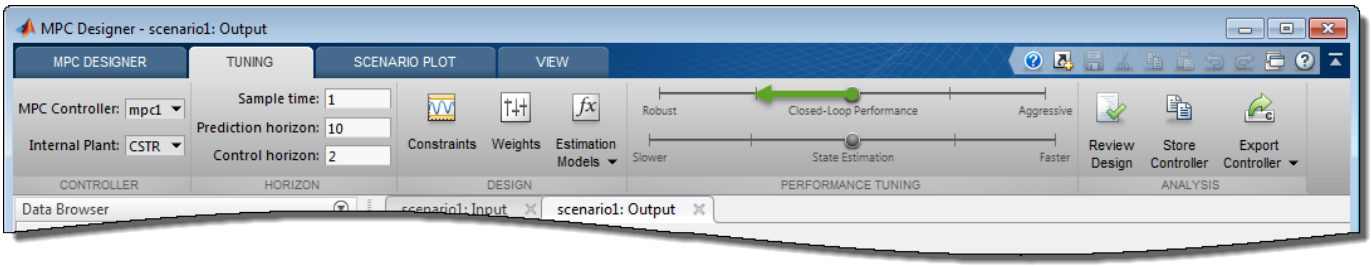
Robustness

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. For an example, see "Test MPC Controller Robustness using MPC Designer" on page 5-5.

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 **MPC Designer**, 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 "Adjust Disturbance and Noise Models" on page 3-25).

Finally, if tuning changes do not provide adequate robustness, consider one of the following options:

- Adaptive MPC control on page 7-2
- Gain-scheduled MPC control on page 8-2

See Also

More About

- “Optimization Problem” on page 2-7
- “Specify Constraints” on page 1-5
- “Adjust Disturbance and Noise Models” on page 3-25
- “Tuning MPC Controller Weights at Run-Time” on page 5-36
- “Setting Targets for Manipulated Variables” on page 3-2

Design Model Predictive Controller at Equilibrium Operating Point

This example shows how to design a model predictive controller with nonzero nominal values.

The plant model is obtained by linearization of a nonlinear plant in Simulink® at a nonzero steady-state operating point.

Linearize Nonlinear Plant Model

To run this example, Simulink and Simulink Control Design™ are required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

The nonlinear plant is implemented in Simulink model `mpc_nloffsets` and linearized at the default operating condition using the `linearize` function from Simulink Control Design.

Create operating point specification for the current model initial condition.

```
plant_mdl = 'mpc_nloffsets';
op = operspec(plant_mdl);
```

Compute the operating point for this initial condition.

```
[op_point, op_report] = findop(plant_mdl,op);
```

```
 Operating point search report:
---------------------------------

opreport =


 Operating point search report for the Model mpc_nloffsets.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</

(1.) mpc_nloffsets/Integrator
    -Inf    0.57514    Inf      0      -1.8208e-14      0
(2.) mpc_nloffsets/Integrator2
    -Inf     2.1503    Inf      0      -8.3844e-12      0

Inputs:
----------
    <strong>Min</strong>       <strong>u</strong>       <strong>Max</strong>
```

```
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>

(1.) mpc_nloffsets/In1
    -Inf    -1.252    Inf

Outputs:
----------
    <strong>Min</strong>        <strong>y</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>________</strong>    <strong>___</strong>

(1.) mpc_nloffsets/Out1
    -Inf    -0.52938    Inf
```

Extract nominal state, output, and input values from the computed operating point.

```
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = op_report.Outputs.y;
u0 = op_report.Inputs.u;
```

Linearize the plant at the initial condition.

```
plant = linearize(plant_mdl,op_point);
```

Design MPC Controller

Create an MPC controller object with a specified sample time `Ts`, prediction horizon `p`, and control horizon `m`.

```
Ts = 0.1;
p = 20;
m = 3;
mpcobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Set the nominal values in the controller.

```
mpcobj.Model.Nominal = struct('X',x0,'U',u0,'Y',y0);
```

Set the output measurement noise model (white noise, zero mean, variance = 0.01).

```
mpcobj.Model.Noise = 0.1;
```

Set the manipulated variable constraint.

```
mpcobj.MV.Max = 0.2;
```

Simulate Using Simulink

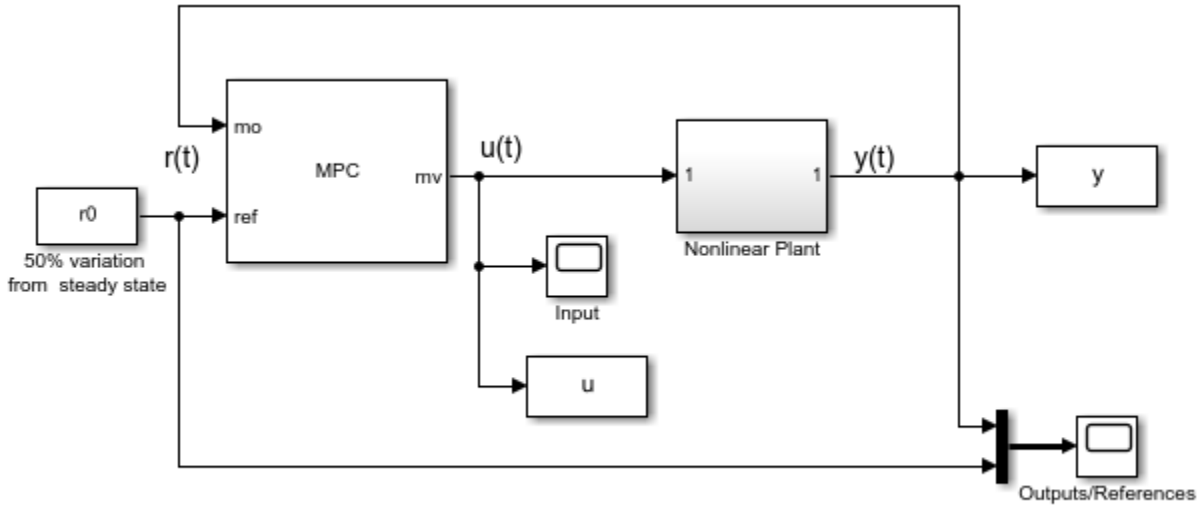
Specify the reference value for the output signal.

```
r0 = 1.5*y0;
```

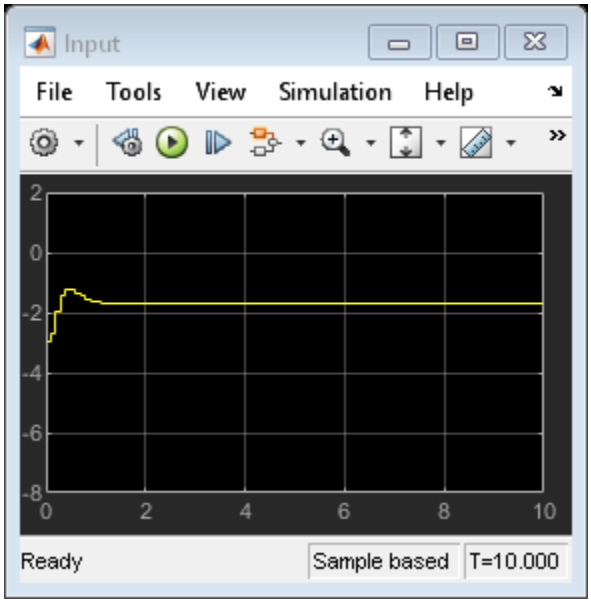
Open and simulate the model.

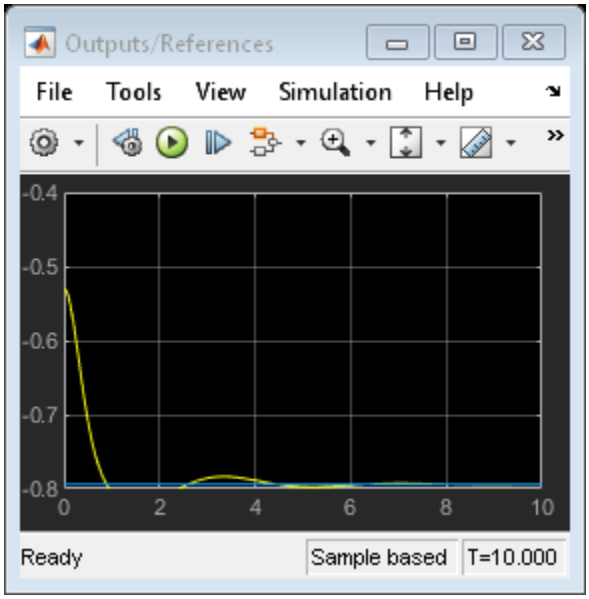
```
mdl = 'mpc_offsets';
open_system(mdl)
sim(mdl)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white noise.
```



Copyright 1990-2014 The MathWorks, Inc.





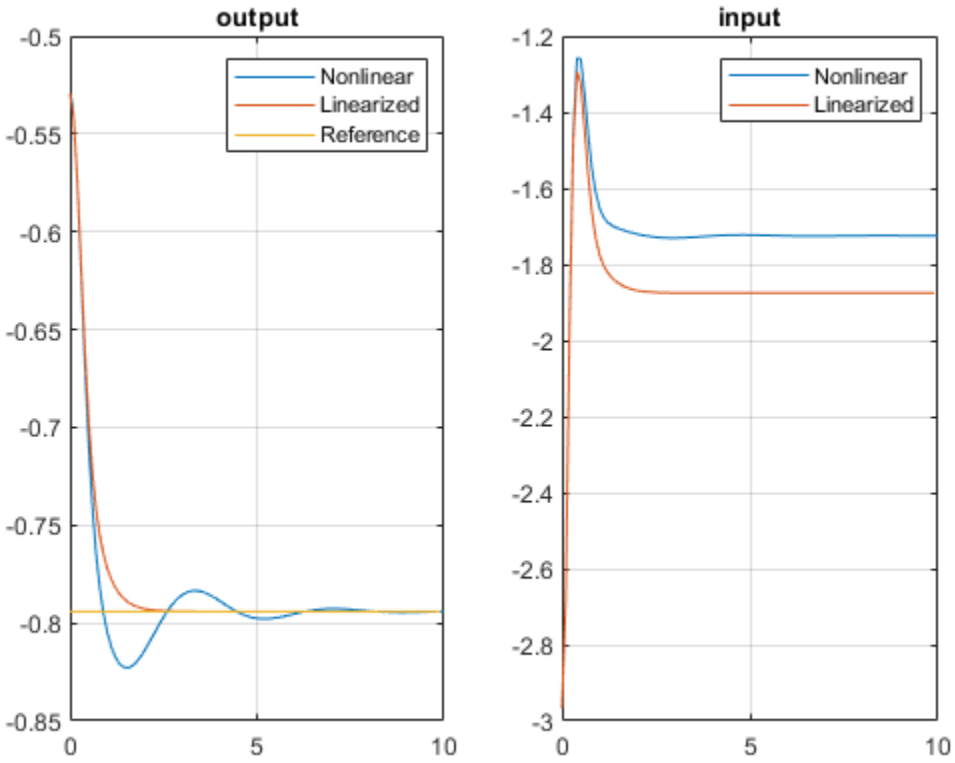
Simulate Using `sim` Command

Simulate the controller.

```
Tf = round(10/Ts);
r = r0*ones(Tf,1);
[y1,t1,u1,x1,xmpc1] = sim(mpcobj,Tf,r);
```

Plot and compare the simulation results.

```
subplot(1,2,1)
plot(y.time,y.signals.values,t1,y1,t1,r)
legend('Nonlinear','Linearized','Reference')
title('output')
grid
subplot(1,2,2)
plot(u.time,u.signals.values,t1,u1)
legend('Nonlinear','Linearized')
title('input')
grid
```




```
bdclose(plant_mdl)
bdclose(mdl)
```

See Also

`mpc` | MPC Controller

Design MPC Controller for Plant with Delays

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

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.

Specify the individual transfer functions for each input/output pair. For example, `g12` 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];
```

Configure Input and Output Signals

Define the 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 **MPC Designer**, on the **MPC Designer** tab, by clicking **I/O Attributes**.

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

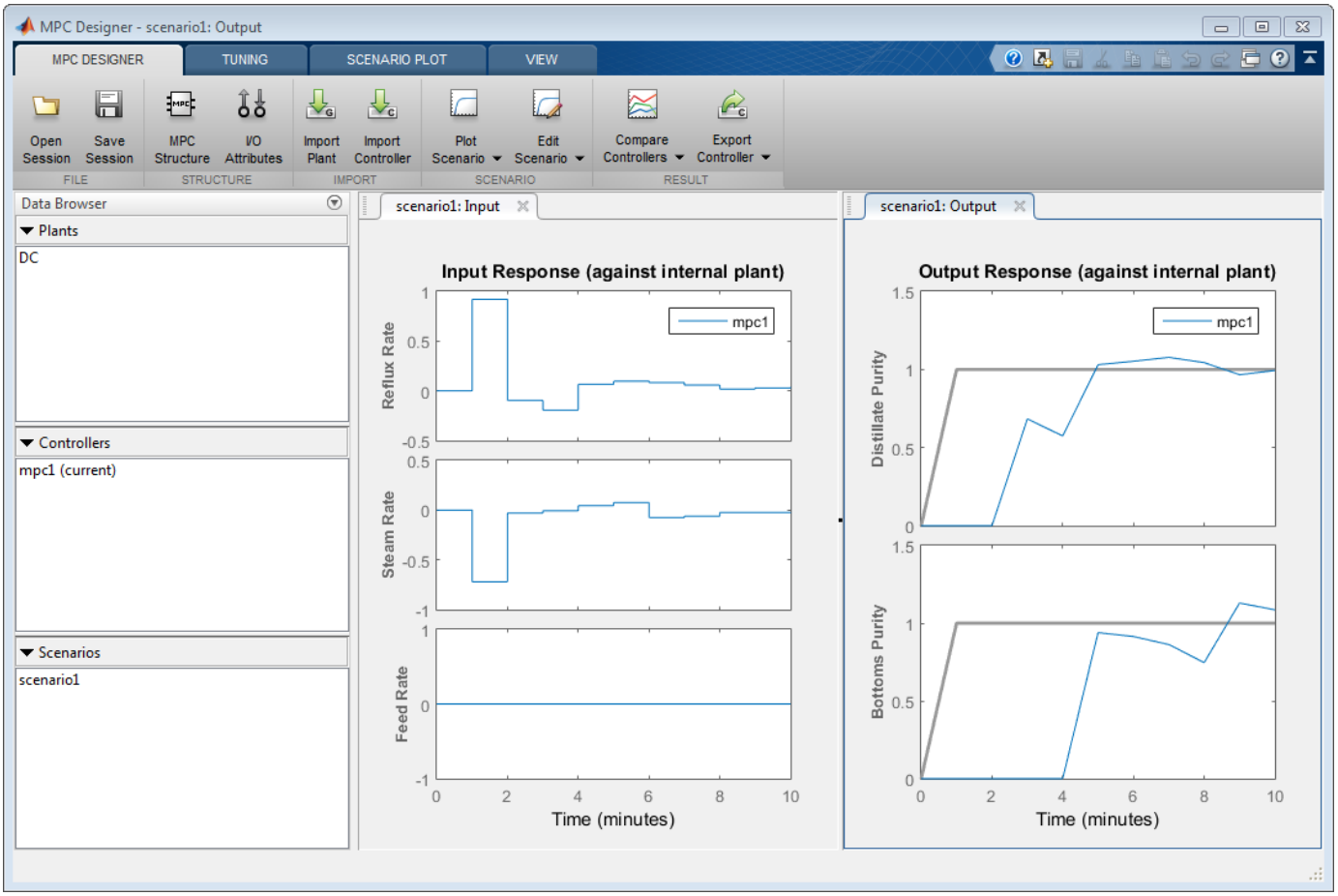
```
DC = setmpcsignals(DC,'MD',3);
```

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

Open **MPC Designer** 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.

MPC Designer 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

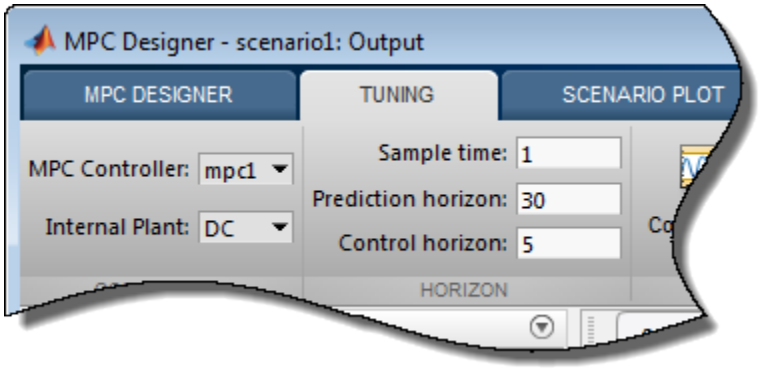
$$P - M \gg t_{d,max}/\Delta t$$

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.

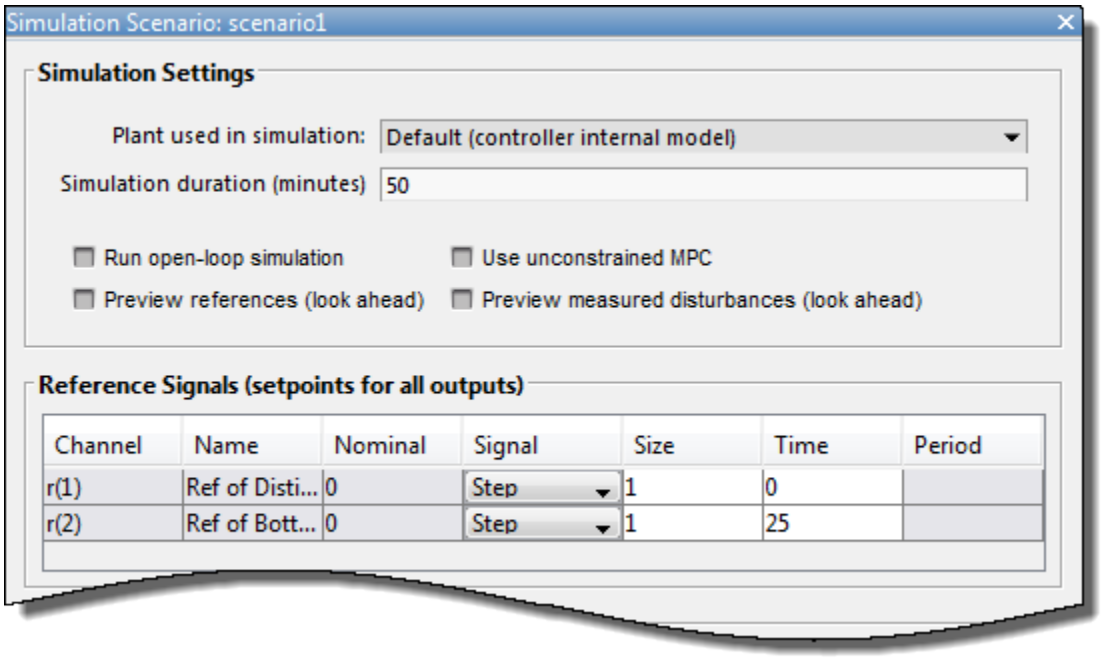
Simulate Controller Step Responses

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.

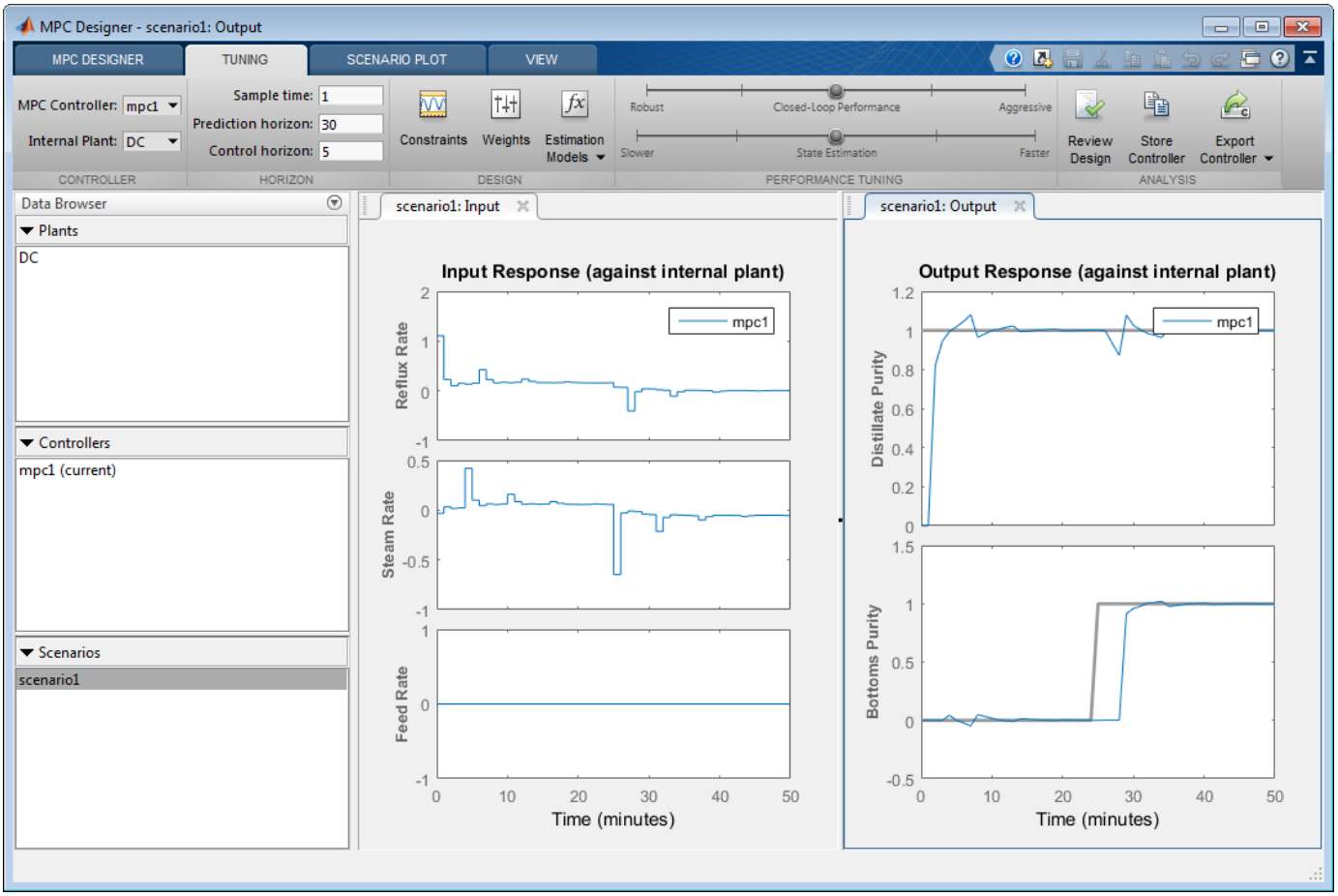
In the **Reference Signals** table, in the **Signal** drop-down list, select `Step` for both outputs to simulate step changes in their setpoints.

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.



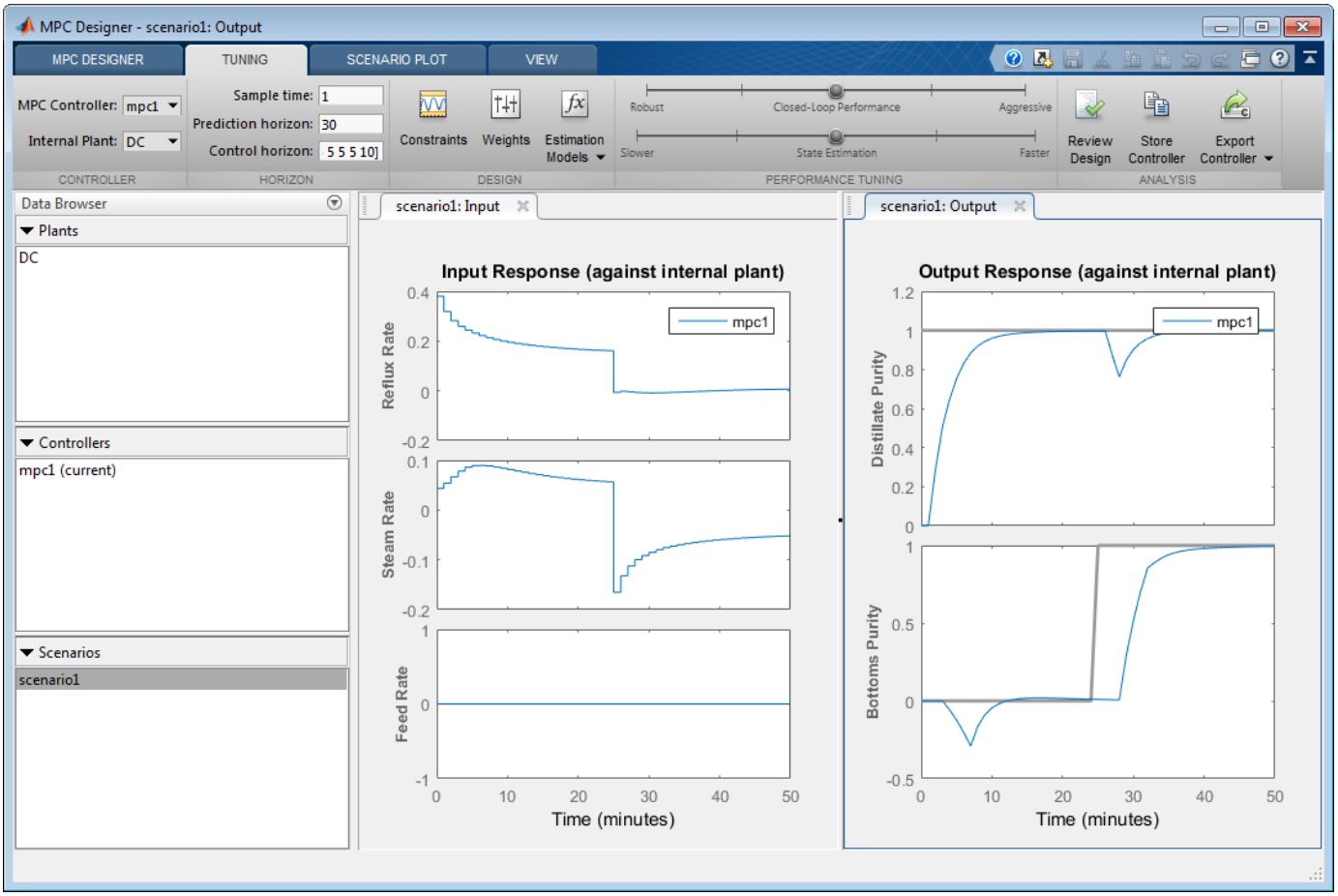
The **Input Response** plots show the optimal control moves generated by the controller. The controller reacts immediately in response to the setpoint changes, changing both manipulated variables. However, due to the plant delays, the effects of these changes are not immediately reflected in the **Output Response** plots. The **Distillate Purity** output responds after 1 minute, which corresponds to the minimum delay from `g11` and `g12`. Similarly, the **Bottoms Purity** output responds 3 minutes after the step change, which corresponds to the minimum delay from `g21` and `g22`. 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]`.



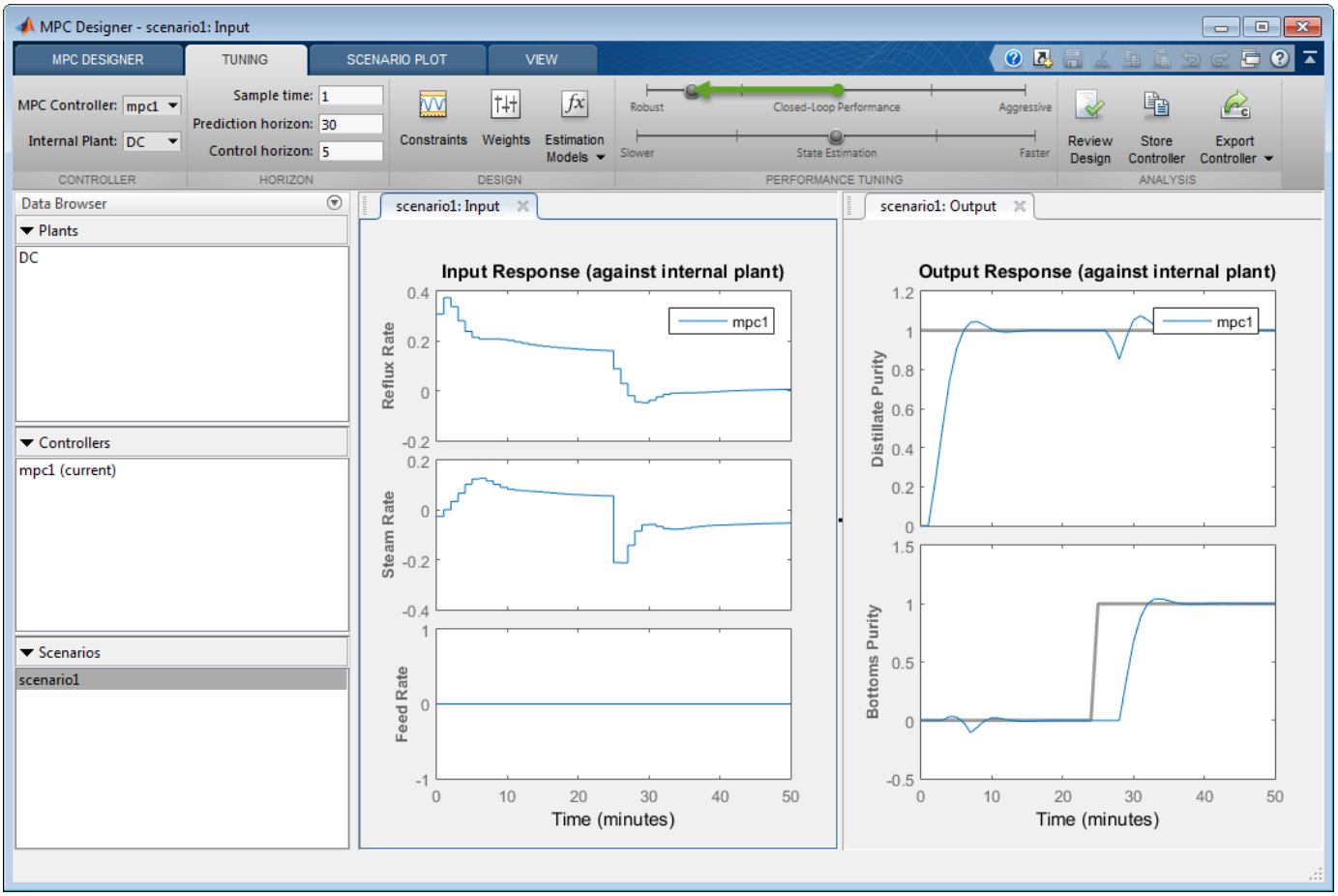
The initial manipulated variable moves are much smaller and the moves are less oscillatory. The trade-off is a slower output response, with larger interactions between the outputs.

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

More About

- “Manipulated Variable Blocking” on page 3-44
- “Design Controller Using MPC Designer”
- “Specify Multi-Input Multi-Output Plants”

Design MPC Controller for Nonsquare Plants

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 all of them at independently chosen setpoints. In this case, you have two options:

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

  The controller does not enforce setpoints on outputs with zero weight, so these outputs are free to vary. If the plant has N_e more outputs than manipulated variables, and the static gain matrix is full rank, setting N_e output weights to zero enables the controller to hold the remaining outputs at their setpoints. If any manipulated variables are constrained at steady state, 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.

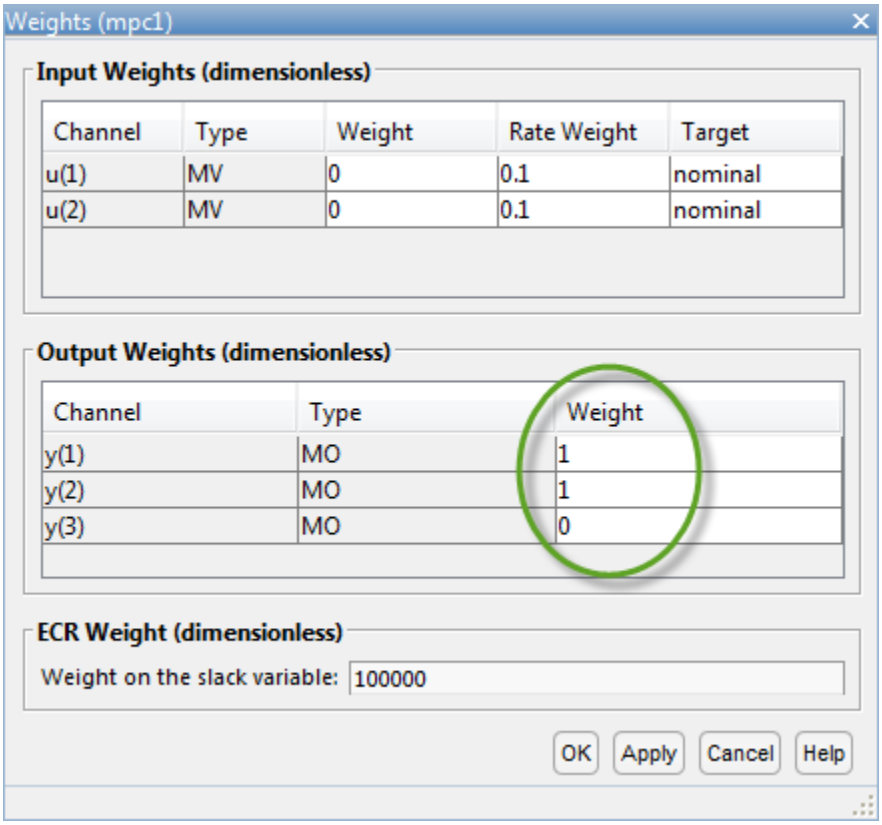
  You can change the error magnitudes by adjusting the relative values of the output weights. Increasing an output weight decreases the steady-state error in that output at the expense of increased error in the other outputs.

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

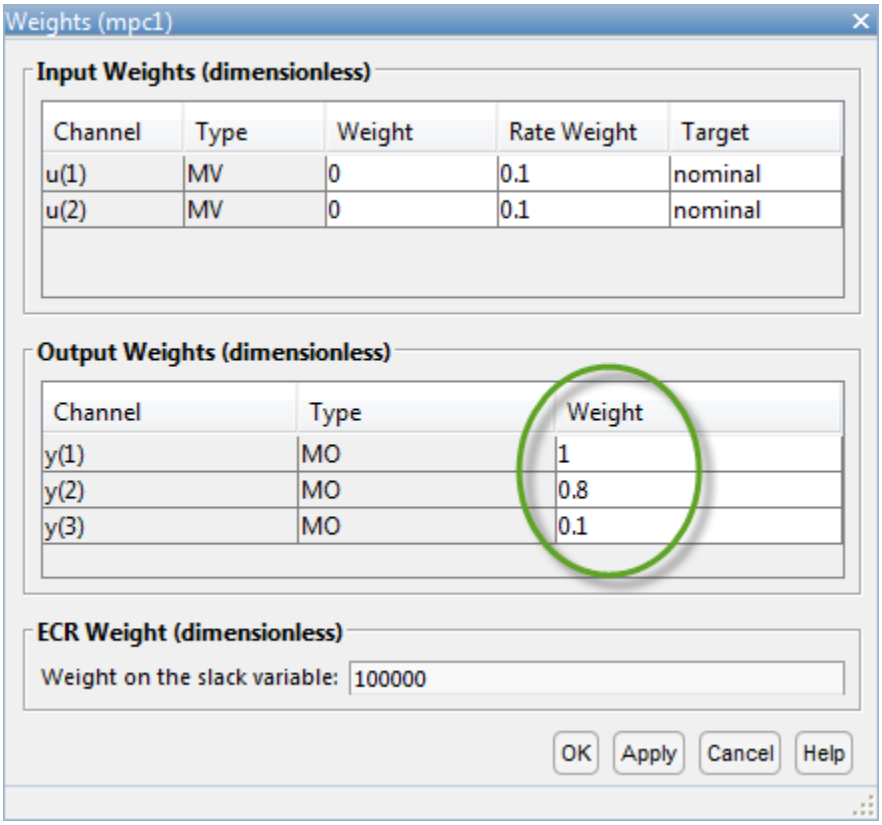
To configure output tuning weights in **MPC Designer**, on the **Tuning** tab, in the **Design** section, click **Weights** to open the Weights dialog box.

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

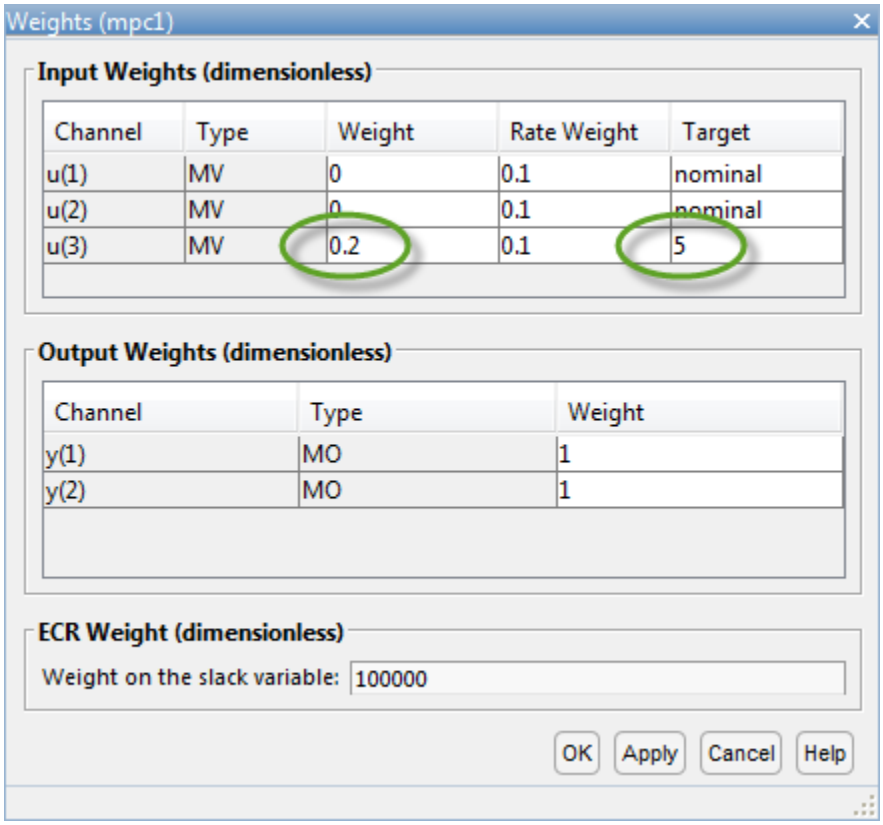
When there are excessive manipulated variables, the default MPC controller settings allow for error-free output setpoint tracking. However, the manipulated variables values can drift. You can prevent this drift by setting manipulated variable setpoints. If there are N_e excess manipulated variables, and the static gain matrix is full rank, you can hold N_e of them at target values for economic or operational reasons, and the remaining manipulated variables can 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 **MPC Designer**, 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 **MPC Designer**, 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

Apps
MPC Designer

Functions
`mpc`

More About

- “Tune Weights” on page 1-43
- “Specify Multi-Input Multi-Output Plants”
- “Setting Targets for Manipulated Variables” on page 3-2

Design MPC Controller for Identified Plant Model

You can define the internal plant model of your model predictive controller using a linear model identified using System Identification Toolbox software. You can identify the plant model and design the MPC controller interactively using apps or programmatically at the command line. For more information on identifying plant models, see “Identify Plant from Data”.

The **System Identification** app is not supported in MATLAB Online™.

Design Controller for Identified Plant Using Apps

This example shows how to interactively design a model predictive controller using an identified plant model. First, estimate the plant model from data using the **System Identification** app. Then design an MPC controller by importing the identified plant into **MPC Designer**.

Load Input/Output Data

Load the input and output data for identification.

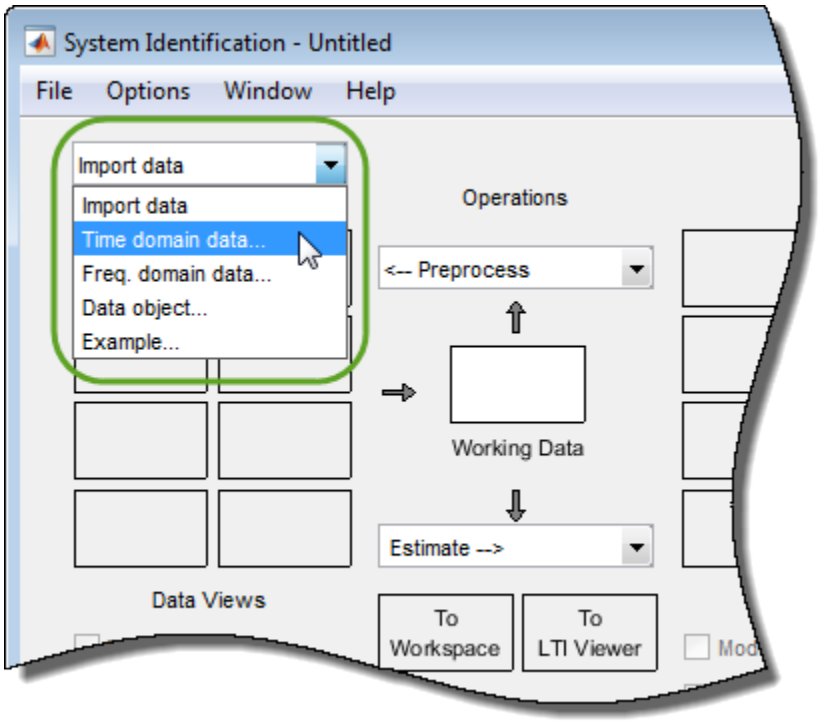
```
load(fullfile(matlabroot,'examples','mpc','plantIO'))
```

This command imports the plant input signal, `u`, output signal, `y`, and sample time, `Ts`, to the MATLAB workspace.

Open the **System Identification** app.

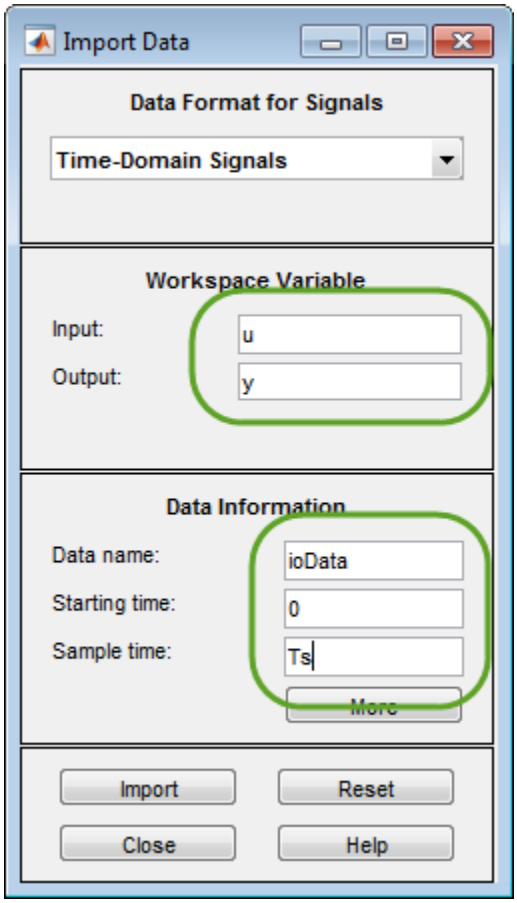
```
systemIdentification
```

In the **System Identification** app, under **Import data**, select `Time domain data`.

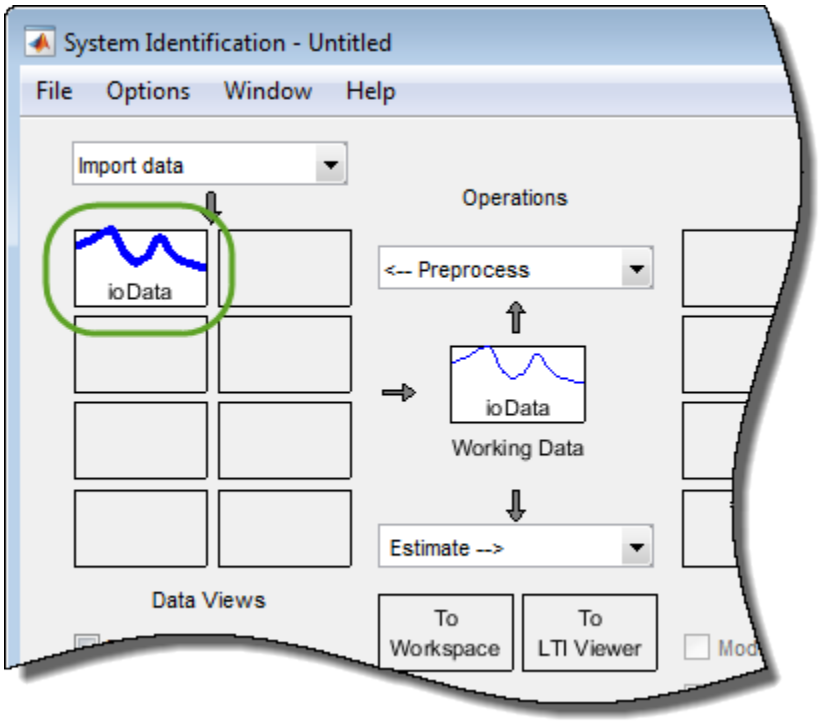


In the Import Data dialog box, specify the **Input**, **Output**, and **Sample time** using the data from the MATLAB workspace.

Also, specify the **Data name** as `ioData` and **Starting time** as `0`.

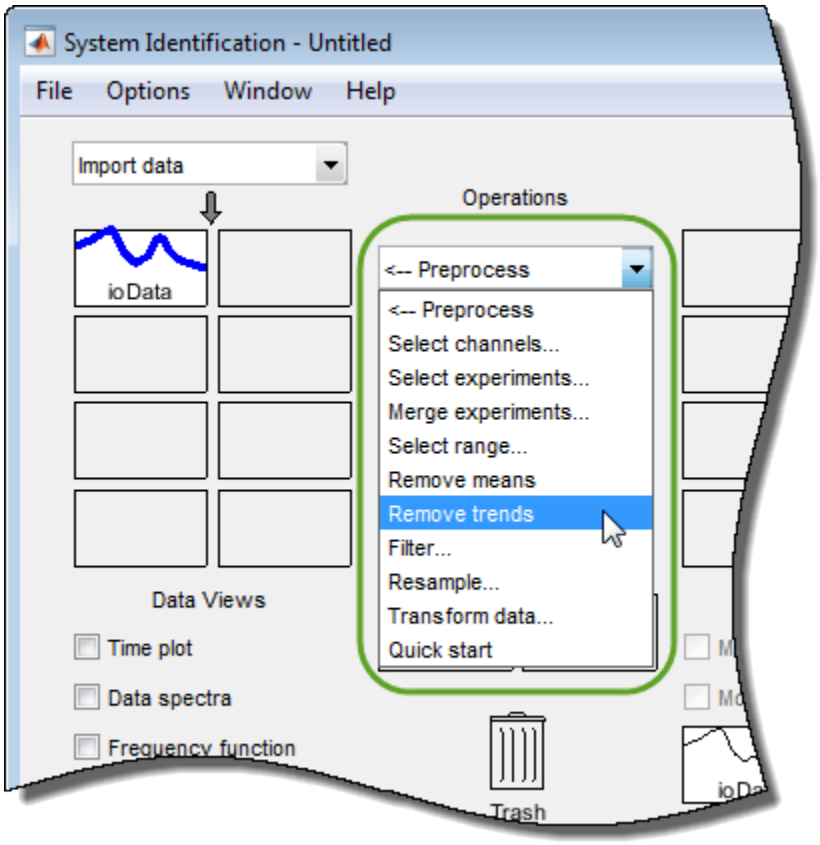


Click **Import**. The app imports the data, creates an `iddata` object with the specified name and signal properties, and adds this object to the **Data Views** area.



Preprocess Data

Typically, you must preprocess identification I/O data before estimating a model. For this example, remove the offsets from the input and output signals by detrending the data. In the **System Identification** app, under **Preprocess**, select `Remove trends`.

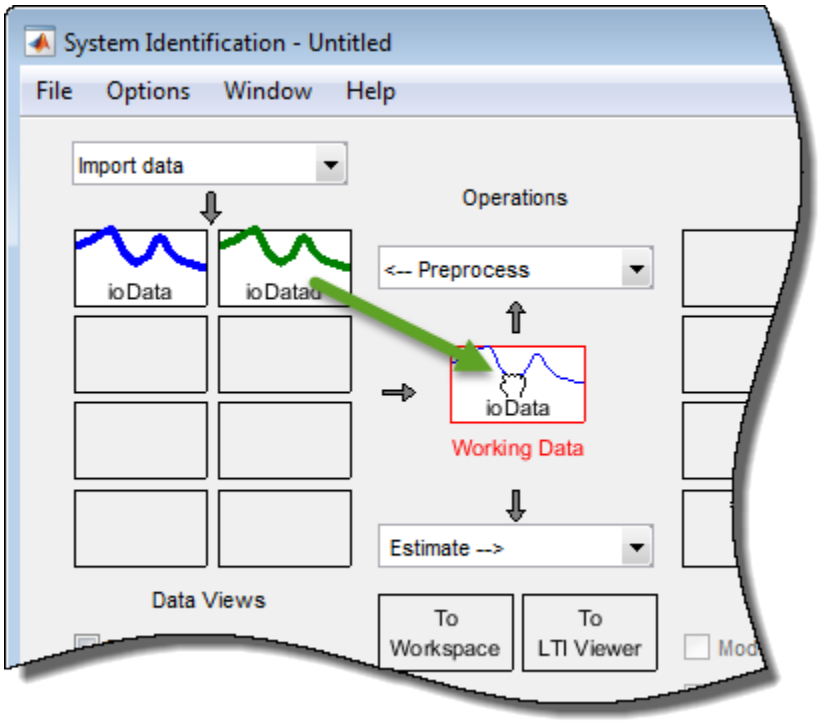


The app creates a data object, `ioDatad`, using the preprocessed data, and adds this object to the **Data Views** area.

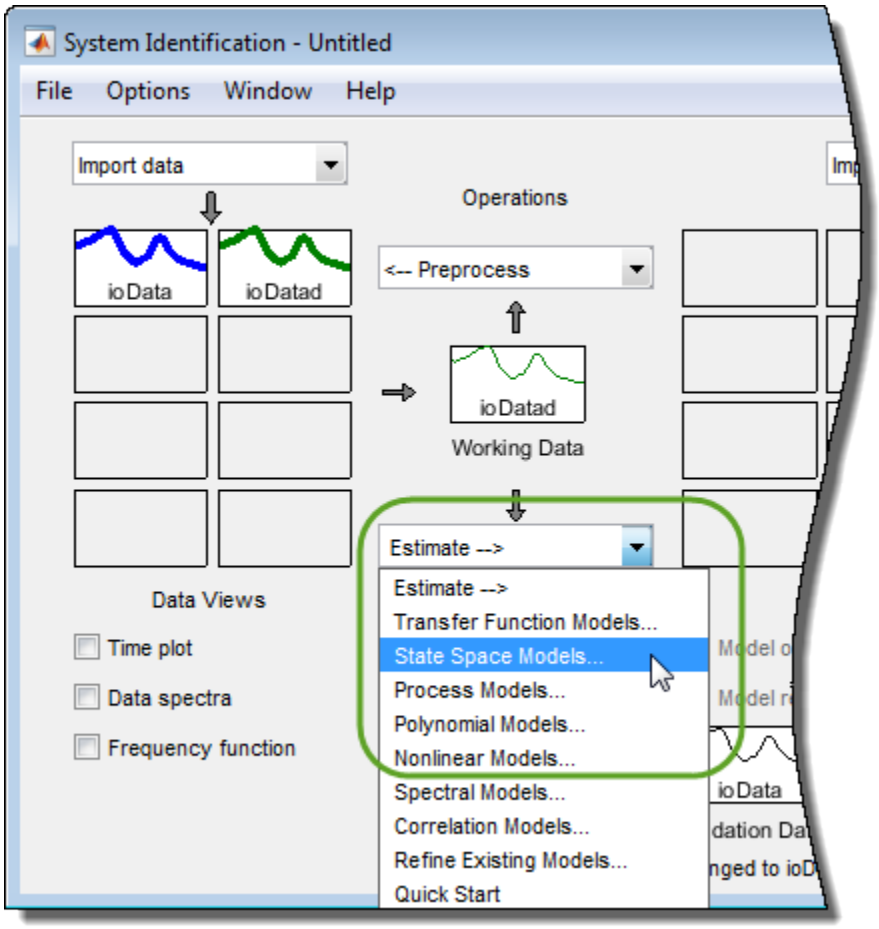
For more information on preprocessing identification data, see “Preprocess Data” (System Identification Toolbox).

Estimate Linear Model

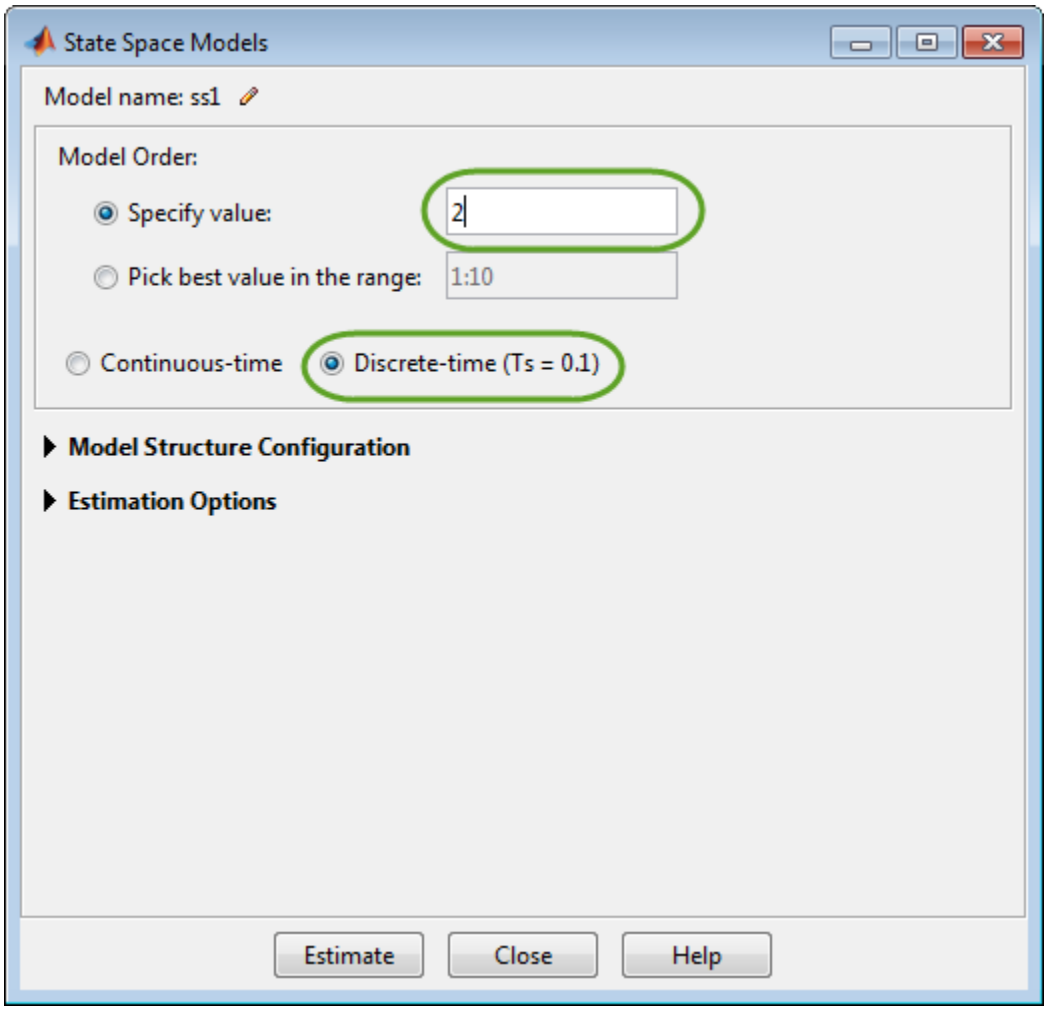
To use the detrended data, `ioDatad`, for model estimation, first drag the corresponding data object from the **Data Views** area to **Working Data**.



To estimate a state-space model, under **Estimate**, select `State Space Models`.

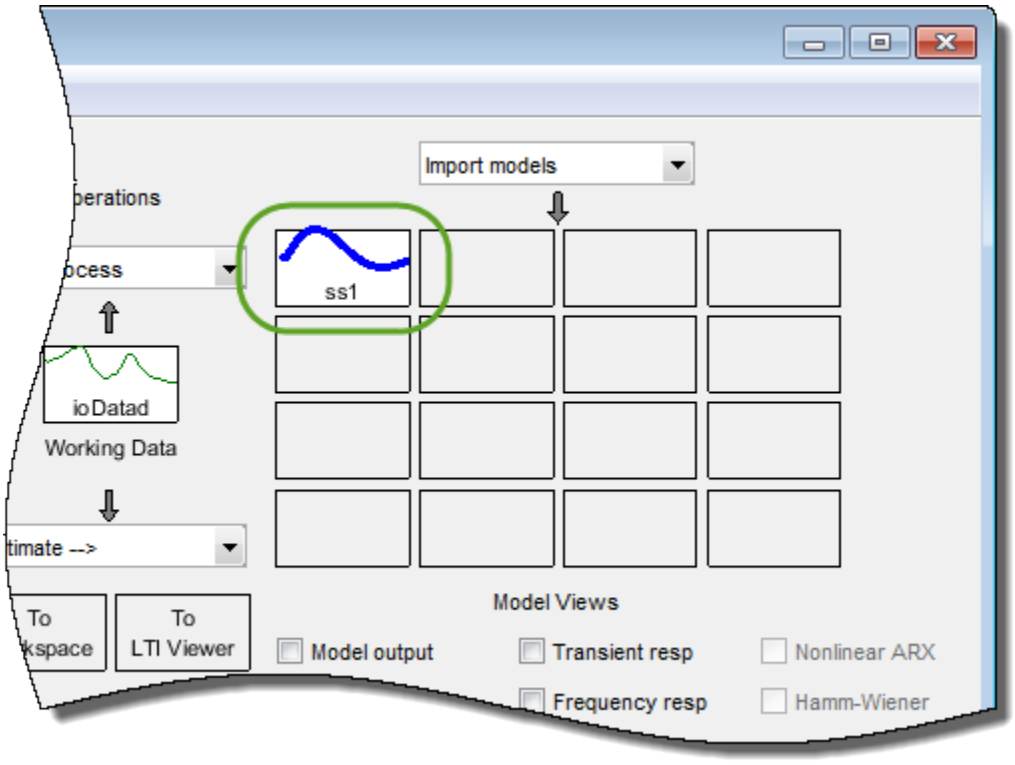


In the State Space Models dialog box, specify the properties of the estimated model and the estimation options. For this example, estimate a second-order, discrete-time model, leaving the other estimation options at their default values.



For more information on estimating state-space models, see “State-Space Models” (System Identification Toolbox).

Click **Estimate**. The app estimates a state-space model, `ss1`, and adds the model to the **Model Views** area.

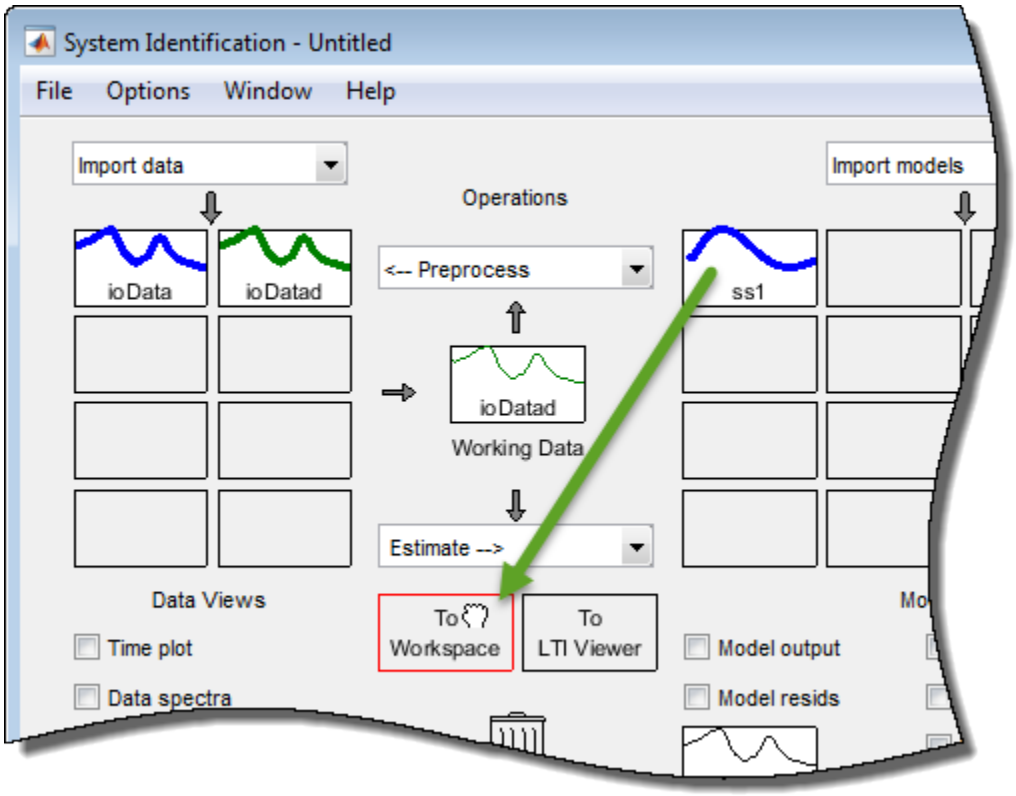


The estimated model has one measured input and one unmeasured noise component.

Import Identified Plant to MPC Designer

To use `ss1` for MPC control design, first export the model to the MATLAB workspace.

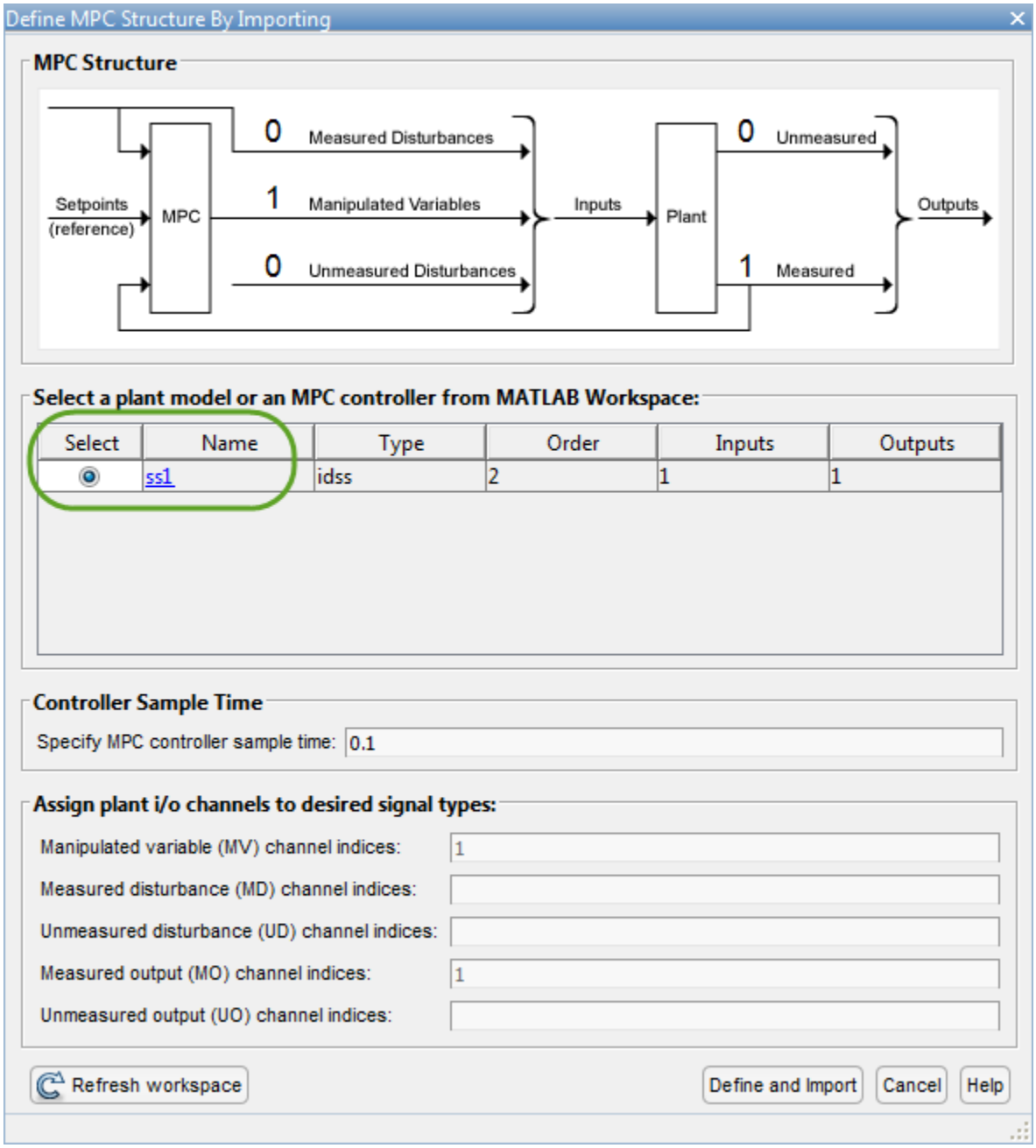
Drag `ss1` from the **Model Views** area to **To Workspace**.



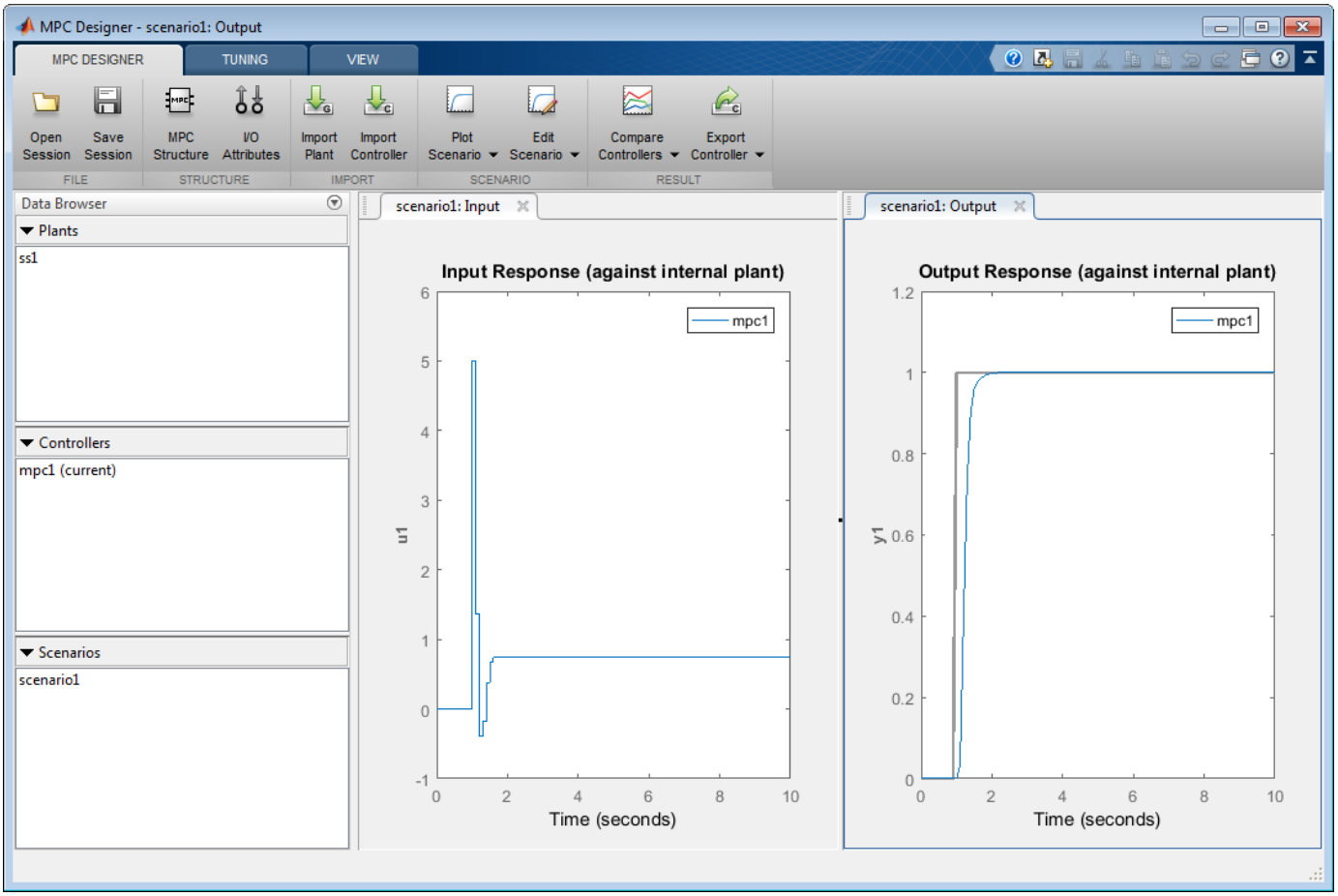
Open **MPC Designer**. At the MATLAB command line, type:

```
mpcDesigner
```

To import the identified model, in **MPC Designer**, click **MPC Structure**. In the Define MPC Structure By Importing dialog box, select `ss1` from the table.



Click **Define and Import**.



Tip You can also import the identified model when opening **MPC Designer**.

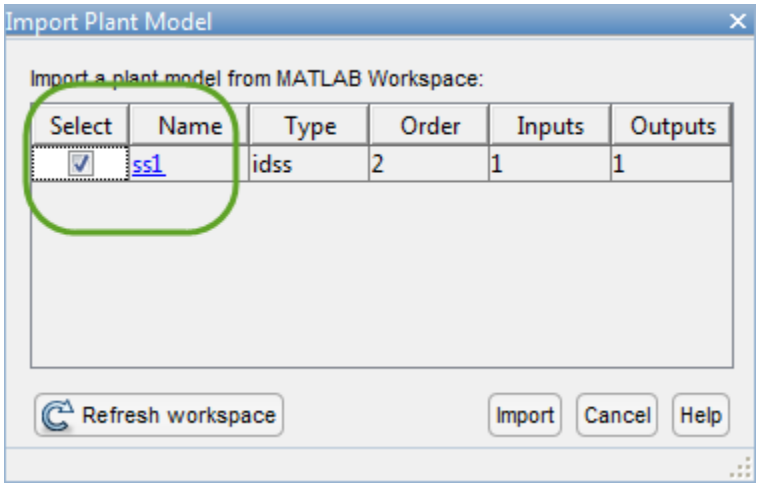
```
mpcDesigner(ss1)
```

The app converts the identified plant to a discrete-time, state-space model, if necessary, and creates a default MPC controller, `mpc1`, in which the:

- Measured input of the identified plant is a manipulated variable.
- Output of the identified plant is a measured output.

By default, the MPC controller discards the unmeasured noise component from your identified model. To configure noise channels as unmeasured disturbances, you must first create an augmented state-space model from your identified model. For more information, see “Configure Noise Channels as Unmeasured Disturbances” on page 1-84.

Note You can also import an identified linear model into an existing **MPC Designer** session. In **MPC Designer**, click **Import Plant**. In the Import Plant Model dialog box, select an identified model from the table.



Only identified models with an I/O configuration that is compatible with the current MPC structure are displayed in the Import Plant Model dialog box. If the current MPC structure includes unmeasured disturbances, any noise channels from the identified model are converted to unmeasured disturbances. Otherwise, the noise channels are discarded.

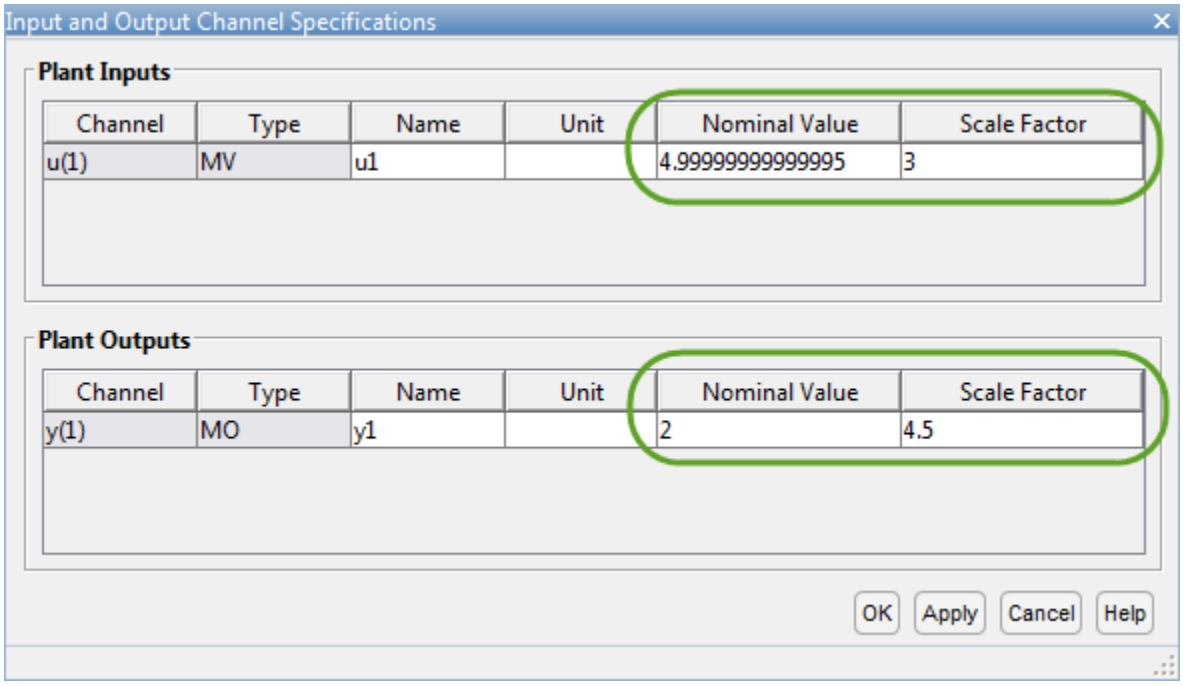
Specify I/O Attributes

To improve controller performance and simplify controller tuning, specify the following attributes for each input and output signal:

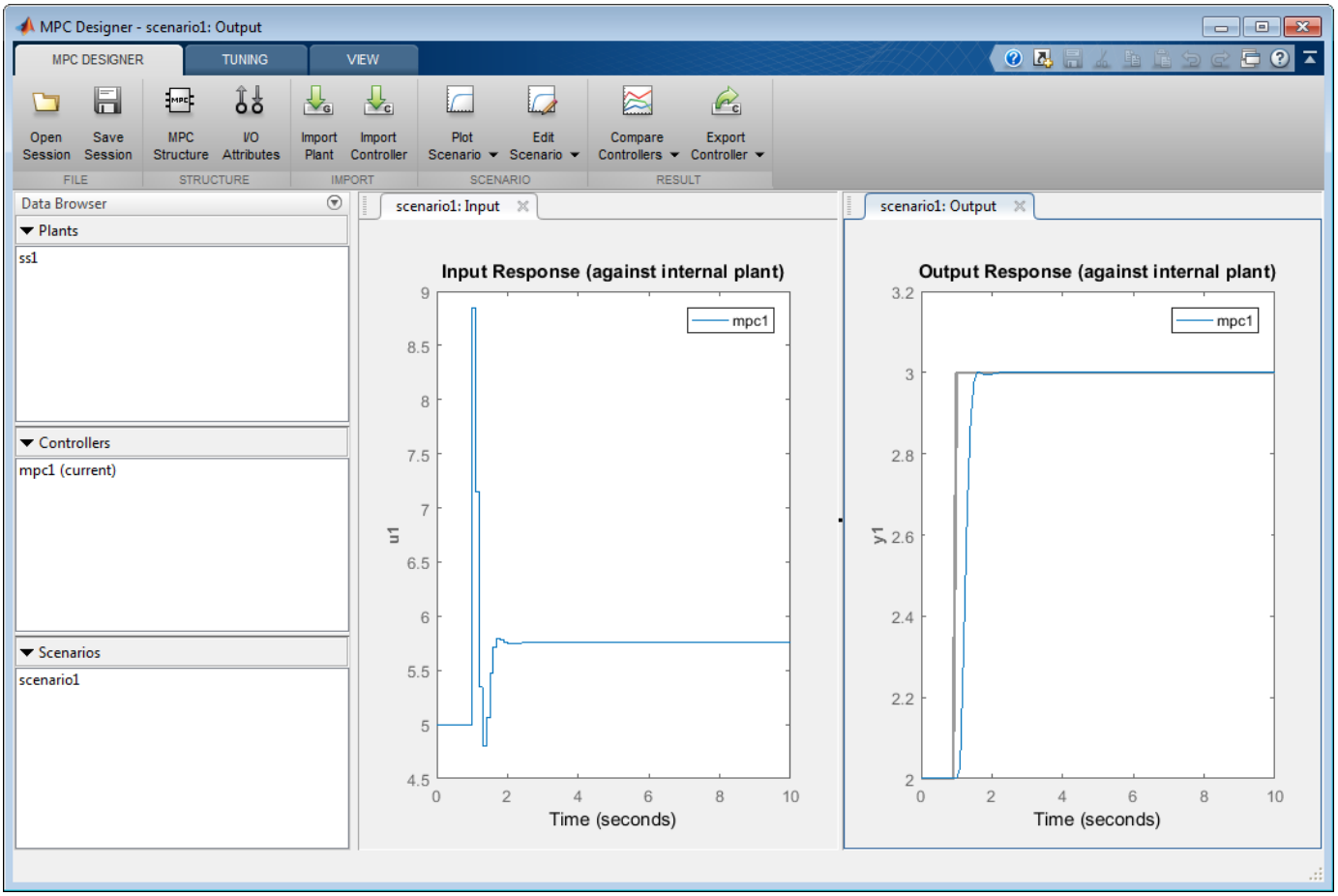
- **Scale Factor** — Scale each signal by a factor that approximates its span, which is the difference between its maximum and minimum values. Scaling simplifies controller weight tuning and improves the numerical conditioning of the controller. For more information, see “Specify Scale Factors” on page 1-29.
- **Nominal Value** — Apply an offset to each signal that corresponds to the nominal operating conditions under which you collected the identification data; that is the offsets removed by detrending the data. Specifying nominal values places the controller at the same operating point as the plant, which is important when the plant is a nonlinear system.

In **MPC Designer**, on the **MPC Designer** tab, click **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, specify the **Nominal Value** and **Scale Factor** for the input and output signals.

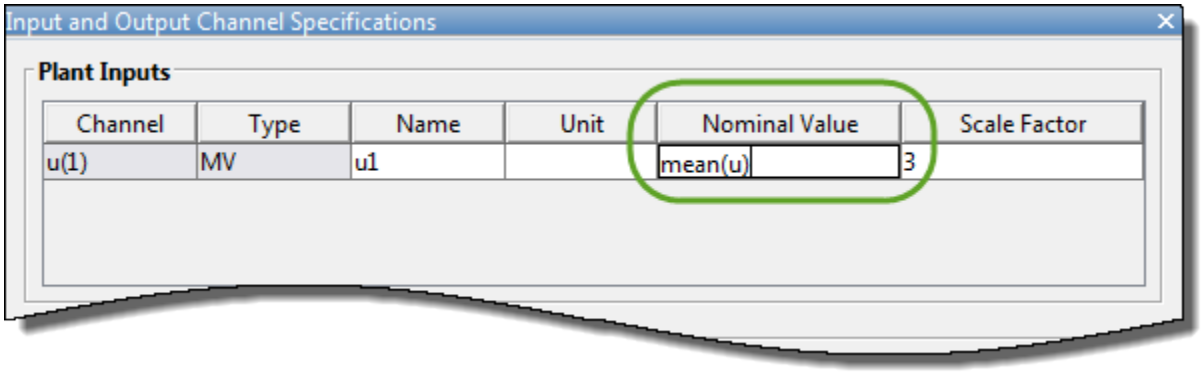

Input and Output Channel Specifications
Plant Inputs
Channel
Type
Name
Unit
Nominal Value
Scale Factor
u(1)
MV
u1
4.99999999999995
3
Plant Outputs
Channel
Type
Name
Unit
Nominal Value
Scale Factor
y(1)
MO
y1
2
4.5
OK
Apply
Cancel
Help

Click **OK**.



The default controller tracks the output reference value well, however the initial controller response is aggressive.

Tip You can specify the **Nominal Value** or **Scale Factor** using expressions such as `mean(u)` or `max(y) - min(y)` respectively, where `u` and `y` are the I/O signals from the MATLAB workspace.

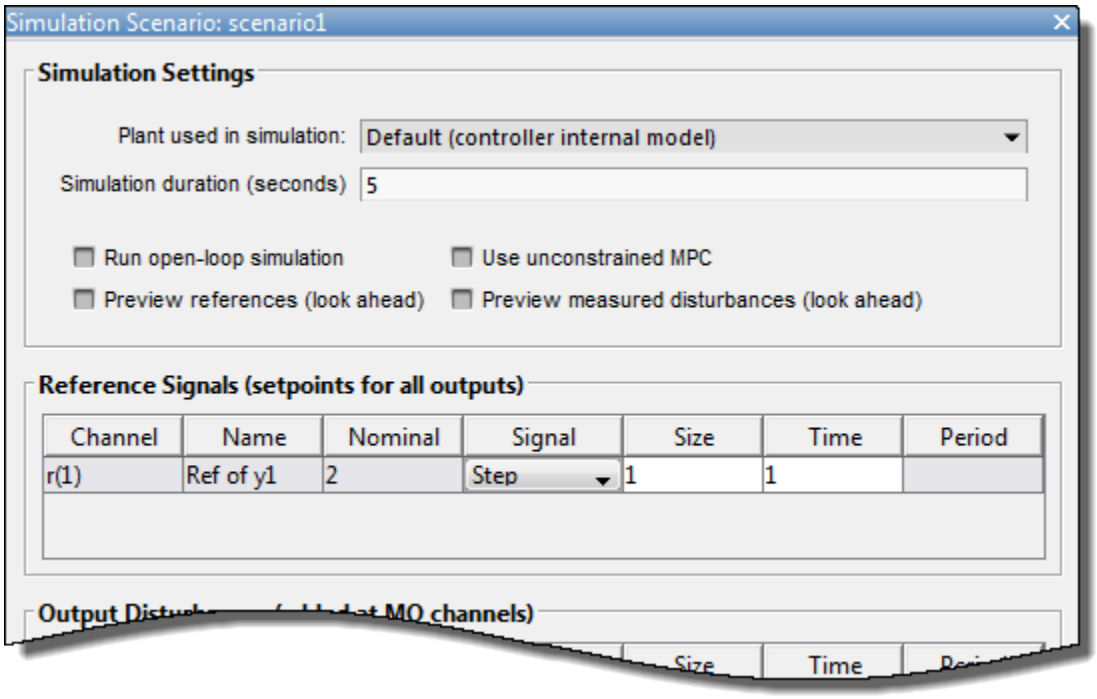


Configure Simulation Scenario

In **MPC Designer**, on the **Tuning** tab, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, specify a **Simulation duration** of 5 seconds.

In the **Reference Signals** section, keep the default step signal.

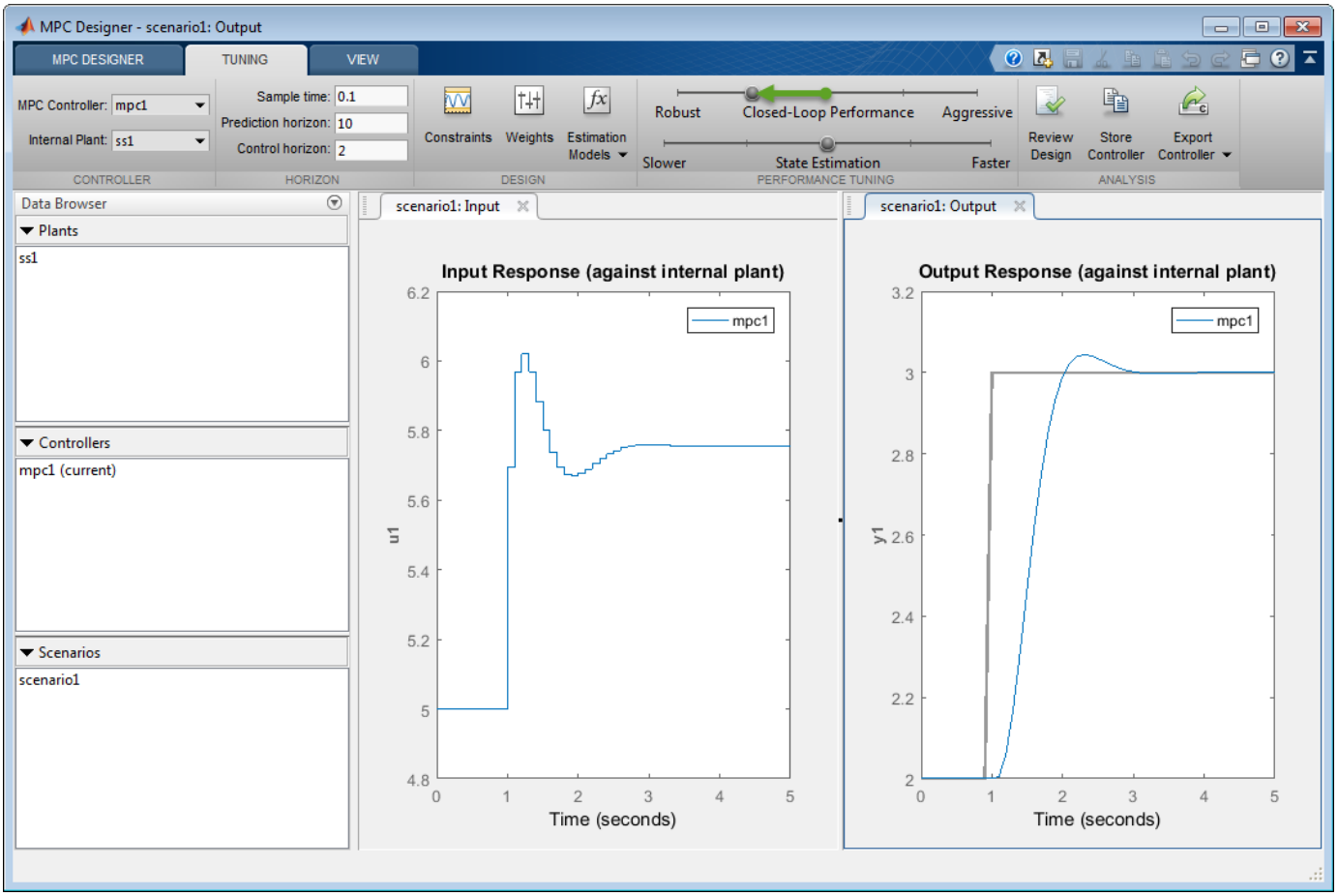


Click **OK.**

Tune Controller

Before tuning your controller, it is good practice to specify the controller sample time, prediction horizon, and control horizon. Since you identified a discrete-time plant model, the controller automatically derives its sample time from the identified model. For this example, use the default prediction and control horizons. For more information, see “Choose Sample Time and Horizons” on page 1-2.

To make the controller less aggressive, on the **Tuning** tab, drag the **Closed-Loop Performance** slider to the left. Doing so increases the cost function weight on the manipulated variable rate of change, and decreases the weight on the output variable.



The input response is now more conservative. The trade-offs are an increased overshoot and longer settling time.

For more information on tuning controller weights, see “Tune Weights” on page 1-43.

Note If your plant has known physical or safety constraints that limit the output range, input range, or input signal rate of change, you can specify these constraints in the MPC controller. If so, define the constraints before tuning your controller weights. For more information, see “Specify Constraints” on page 1-5.

Design Controller for Identified Plant at the Command Line

This example shows how to design a model predictive controller at the command line using an identified plant model.

Load the input/output data.

```
load plantIO
```

This command imports the plant input signal, `u`, plant output signal, `y`, and sample time, `Ts`, to the MATLAB® workspace.

Create an `iddata` object from the input and output data.

```
mydata = iddata(y,u,Ts);
```

Preprocess the I/O data by removing offsets (mean values) from the input and output signals.

```
mydatad = detrend(mydata);
```

You can also remove offsets by creating an `ssestOptions` object and specifying the `InputOffset` and `OutputOffset` options.

Estimate a second order, linear state-space model using the I/O data. Estimate a discrete-time model by specifying the sample time as `Ts`.

```
ss1 = ssest(mydatad,2,'Ts',Ts);
```

The estimated model has one measured input and one unmeasured noise component.

Create a default model predictive controller for the identified model, `ss1`.

```
mpcObj = mpc(ss1);
```

```
-->Converting linear model from System Identification Toolbox to state-space.
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

By default the controller discards the unmeasured noise component from your identified model.

To simplify the tuning process, specify input and output signal scaling factors.

```
mpcObj.MV(1).ScaleFactor = max(u) - min(u);
mpcObj.OV(1).ScaleFactor = max(y) - min(y);
```

Specify the nominal values for the input and output signals. Use the offsets that you previously removed from the I/O data.

```
nominalInput = mean(u);
nominalOutput = mean(y);
mpcObj.Model.Nominal.u = nominalInput;
mpcObj.Model.Nominal.y = nominalOutput;
```

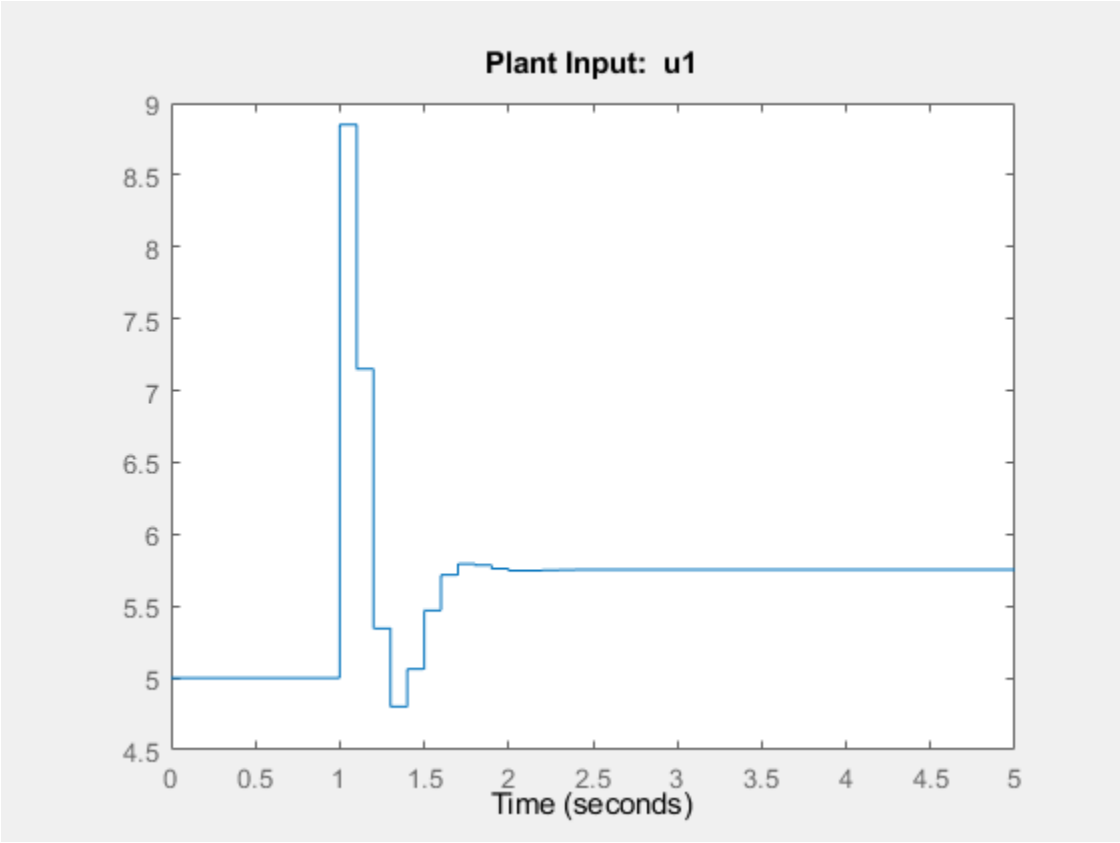
Configure the simulation reference signal. Specify a reference signal with a five-second duration and a unit step at a time of one second. The initial value of the reference signal is the nominal value of the output signal.

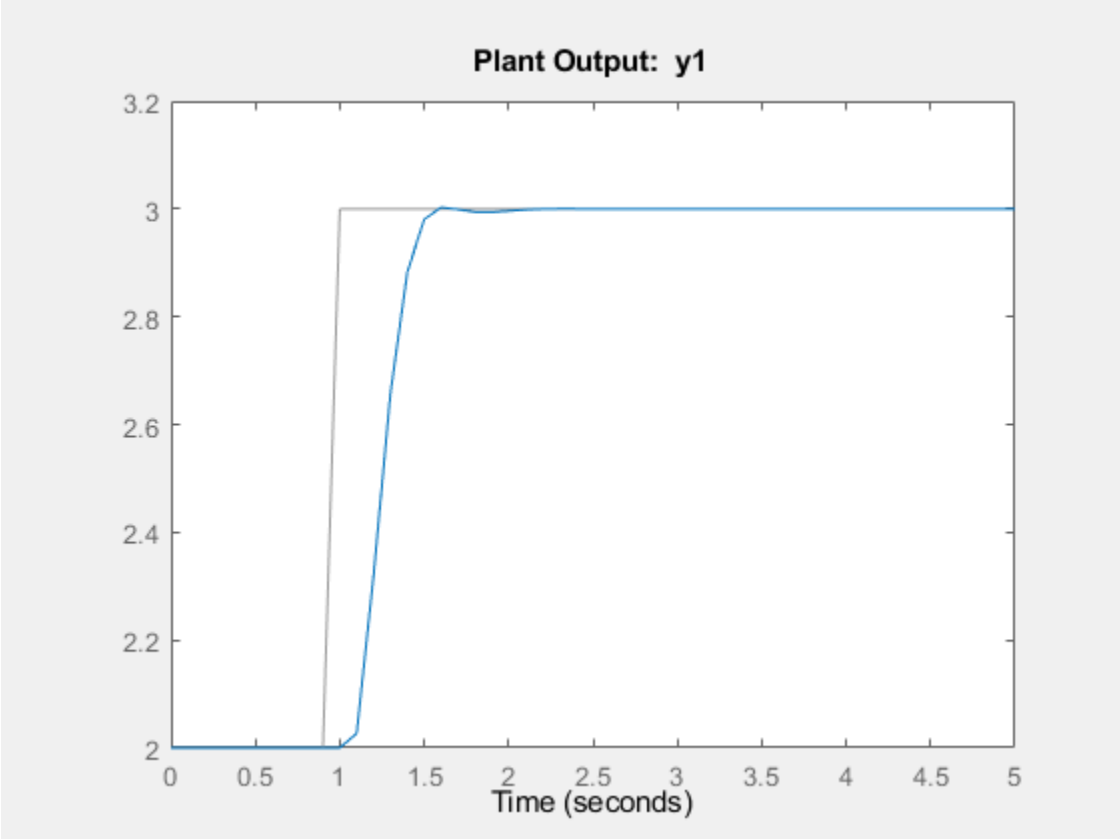
```
outputRef = [nominalOutput*ones(1/Ts,1);
             (nominalOutput+1)*ones(4/Ts+1,1)];
```

Before tuning the controller, simulate the initial controller performance.

```
sim(mpcObj,[],outputRef)
```

```
-->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 measured
```


Plant Input: u1
9
8.5
8
7.5
7
6.5
6
5.5
5
4.5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Time (seconds)



The default controller tracks the output reference value well, however the initial controller response is aggressive.

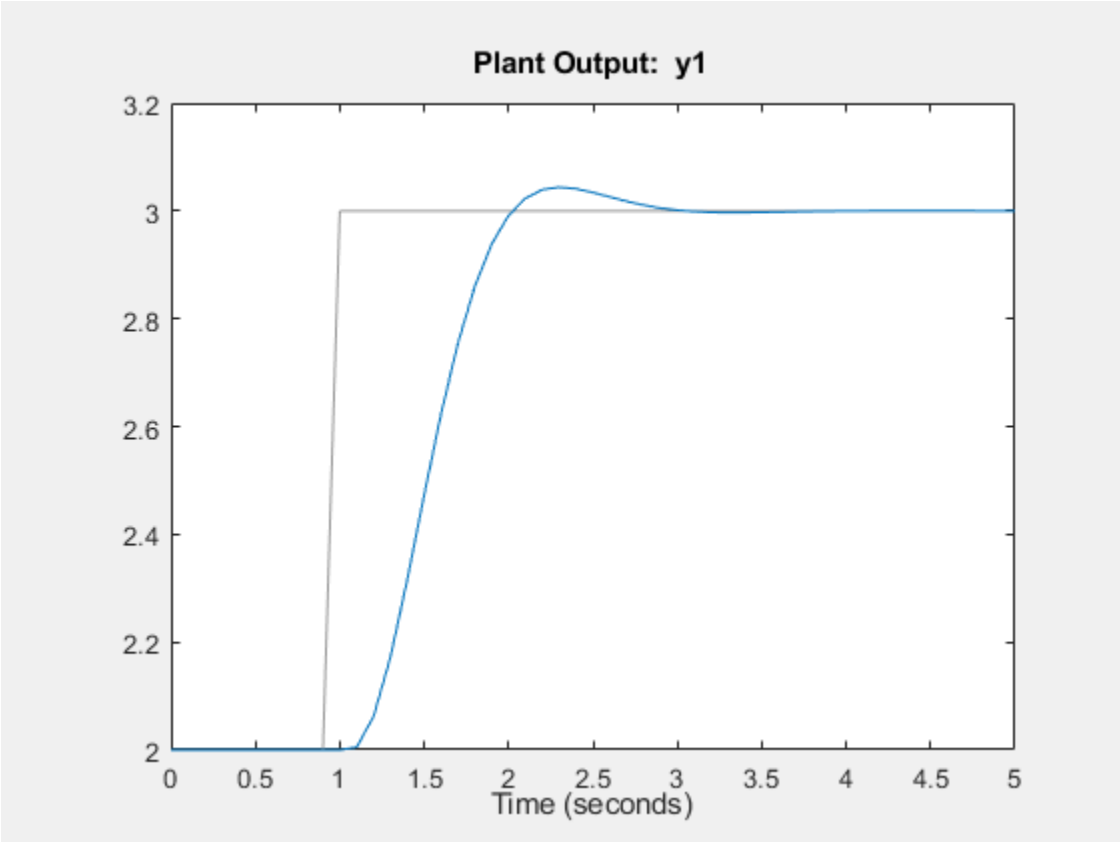
To make the controller less aggressive, simultaneously increase the tuning weight for the manipulated variable rate of change and decrease the tuning weight for the output variable.

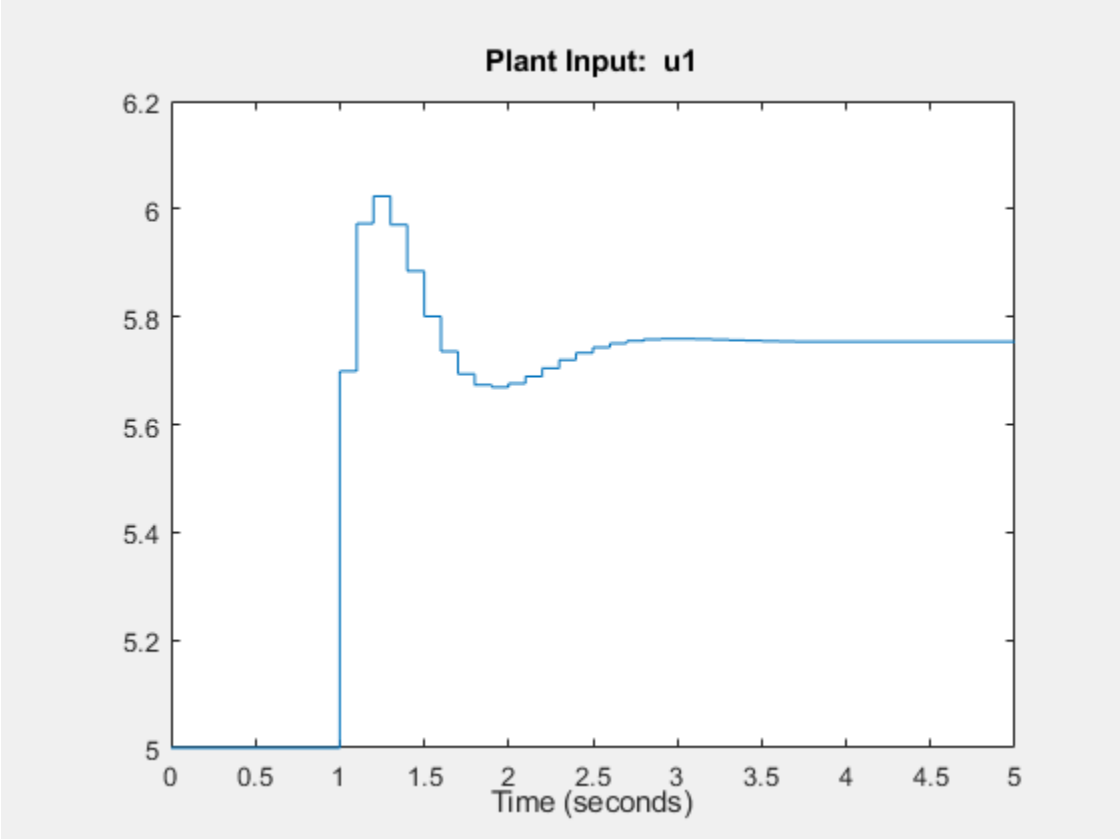
```
beta = 0.37;
mpcObj.Weights.MVRate = mpcObj.Weights.MVRate/beta;
mpcObj.Weights.OV = mpcObj.Weights.OV*beta;
```

Simulate the tuned controller response

```
sim(mpcObj,[],outputRef)
```

```
-->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 measured
```


Plant Output: y1
3.2
3
2.8
2.6
2.4
2.2
2
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Time (seconds)



The input response is now more conservative. The trade-offs are an increased overshoot and longer settling time.

Configure Noise Channels as Unmeasured Disturbances

When you create an MPC controller using an identified model, the software discards any noise channels from the model by default. You can configure the noise channels as unmeasured disturbances by augmenting the identified model.

Augment Identified Model with Noise Channels

To convert noise channels to unmeasured disturbances, first convert the identified model, `ss1`, to a state-space model using the `'augmented'` option. At the MATLAB command line, type:

```
ss2 = ss(ss1,'augmented');
```

This option creates a state-space model, `ss2`, with the following input groups:

- `Measured` — The input channels from the identified model.
- `Noise` — The noise channels from the identified model. The number of noise channels matches the number of outputs channels.

Note The System Identification Toolbox software assumes that the inputs to the `Noise` channels are unit-variance Gaussian noise. Therefore, the augmented model encapsulates any noise dynamics from the identified model, such as integration at the disturbance source.

You can then create an MPC controller using the augmented state-space model.

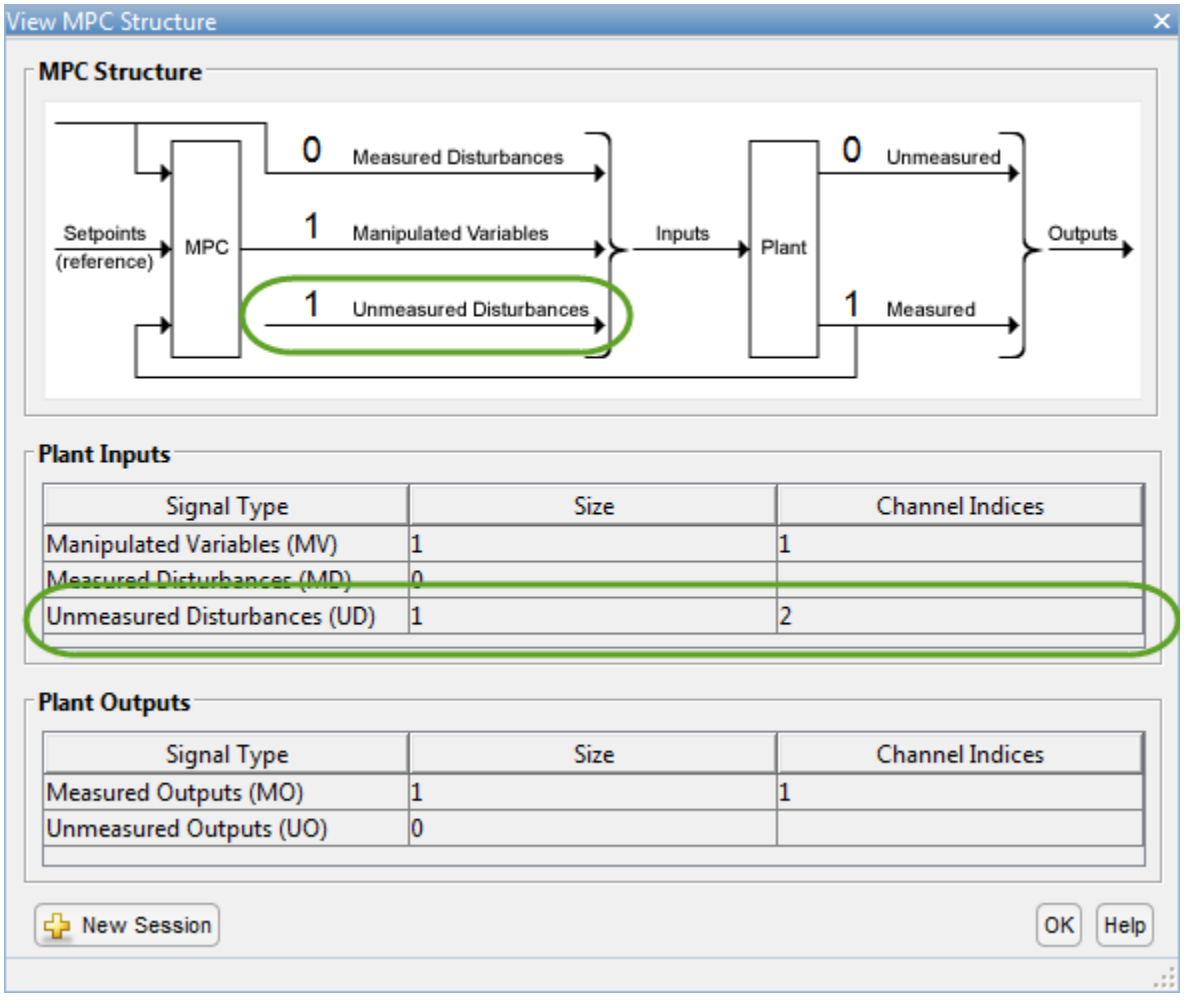
```
mpcObj = mpc(ss2);
```

The software configures the `Measured` inputs as manipulated variables and the `Noise` inputs as unmeasured disturbances.

You can also import the augmented model into **MPC Designer**.

```
mpcDesigner(ss2)
```

To view the MPC signal configuration, in **MPC Designer**, on the **MPC Designer** tab, click **MPC Structure**.



The View MPC Structure dialog box shows the noise channels as unmeasured disturbances.

Configure Input Disturbance Model

When you convert an identified model to an augmented state-space model, the System Identification Toolbox software assumes that noise sources are unit-variance Gaussian noise. However, by default, MPC controllers model unmeasured input disturbances as integrated Gaussian noise. When designing your controller, you can:

- Remove the integrators from the input disturbance model, which simplifies the controller. Use this option if the experimental identification data was collected under conditions that closely match the expected plant operating conditions. In this case, the augmented state-space model encapsulates any noise dynamics from the identified system.
- Keep the default integrated white noise input disturbance model, which leads to more aggressive disturbance rejection. Use this option if the experimental identification data was collected under controlled conditions that may not match the expected plant operating conditions. In this case, the controller compensates for noise dynamics that the augmented model does not encapsulate.

Note When using **MPC Designer**, you can tune your controller disturbance rejection properties by adjusting the **State Estimation** slider. For more information, see “Disturbance Rejection Tuning” on page 3-30.

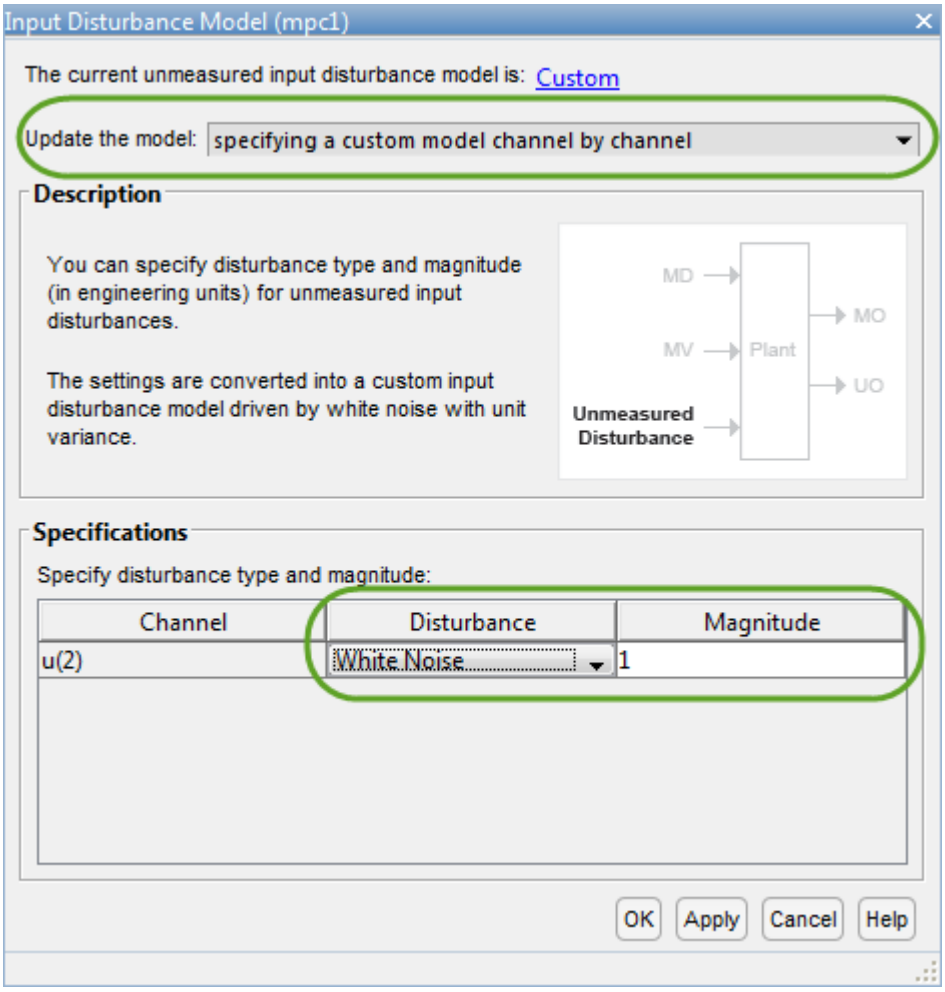
To remove an integrator from an input disturbance model channel, configure that channel as a static unit gain. For example, to remove the integrators from all input disturbance model channels, set the input disturbance model to a static gain identity matrix. At the MATLAB command line, type:

```
setindist(mpcObj,ss(eye(Nd)));
```

where `Nd` is the number of unmeasured disturbances.

To set the disturbance model for an unmeasured disturbance channel to a static unit gain using **MPC Designer**:

1. On the **Tuning** tab, select **Estimation Models** > **Input Disturbance Model**.
2. In the Input Disturbance Model dialog box, in the **Update the model** drop-down list, select `specifying a custom model channel by channel`.
3. In the **Specifications** table, in the **Disturbance** drop-down list, select `White Noise`.
4. Specify a **Magnitude** of `1`.



5 Repeat steps 3 and 4 for each unmeasured disturbance.

6 To apply the changes and update the input disturbance model, click **OK** or **Apply**.

For more information about changing the input disturbance model, see “Adjust Disturbance and Noise Models” on page 3-25.

Configure Simulation Scenario

You can simulate your MPC controller using unit-variance Gaussian noise unmeasured disturbance signals, as assumed by the System Identification Toolbox software. This scenario emulates the experimental conditions under which the data was collected for identification.

To configure unmeasured disturbance signals, create an MPC simulation option set for your controller using `mpcsimopt`. At the MATLAB command line, type:

```
opt = mpcsimopt(mpcObj);
```

Configure the `UnmeasuredDisturbance` option using `randn`.

```
opt.UnmeasuredDisturbance = randn(T,Nd);
```

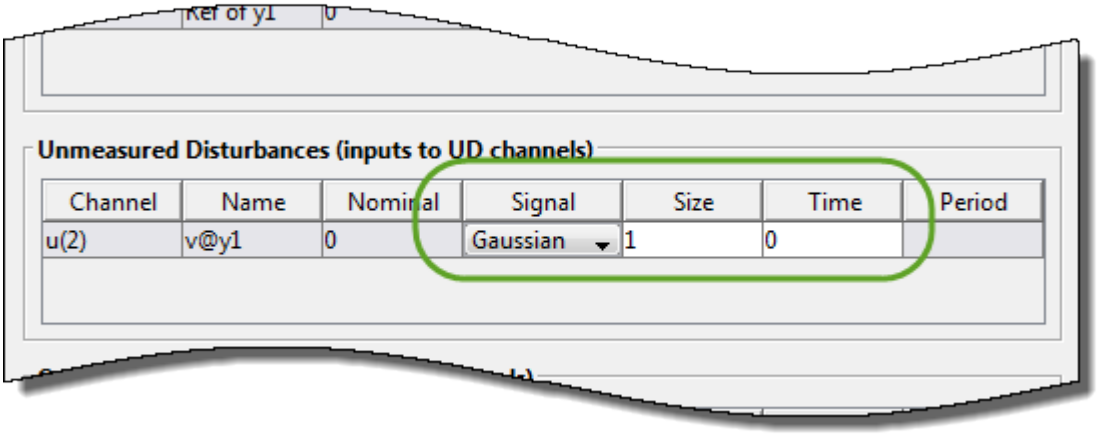
where `T` is the number of simulation steps and `Nd` is the number of unmeasured disturbances.

Simulate the controller using this option set and an output reference signal, `outputRef`.

```
y = sim(mpcObj,T,outputRef,opt);
```

To configure your simulation in **MPC Designer**:

1. On the **Tuning** tab, under **Edit Scenario** select the simulation scenario you want to edit.
2. In the Simulation Scenario dialog box, in the **Unmeasured Disturbances** section, under **Signal**, select `Gaussian`.
3. Specify a **Size** of `1`, which corresponds to a unit variance.
4. To apply the disturbance from the start of the simulation, specify a **Time** of `0`.



5. Repeat steps 2–4 for each unmeasured disturbance channel.
6. To apply the changes and update the **MPC Designer** response plots, click **OK** or **Apply**.

See Also

Apps
MPC Designer | **System Identification**

Functions
`mpc` | `ssest` | `ss` | `sim`

More About

- “About Identified Linear Models” (System Identification Toolbox)
- “Identify Plant from Data”
- “Design Controller Using MPC Designer”
- “Design MPC Controller at the Command Line”

Generate MATLAB Code from MPC Designer

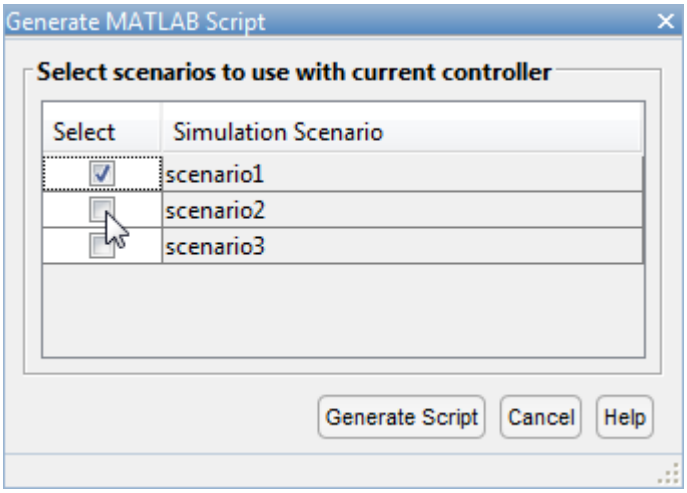
This topic shows how to generate MATLAB code for creating and simulating model predictive controllers designed in the **MPC Designer** app. Generated MATLAB scripts are useful when you want to programmatically reproduce designs that you obtained interactively.

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, on the **MPC Designer** tab, in the **Result**, click **Export Controller**.

   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`

More About

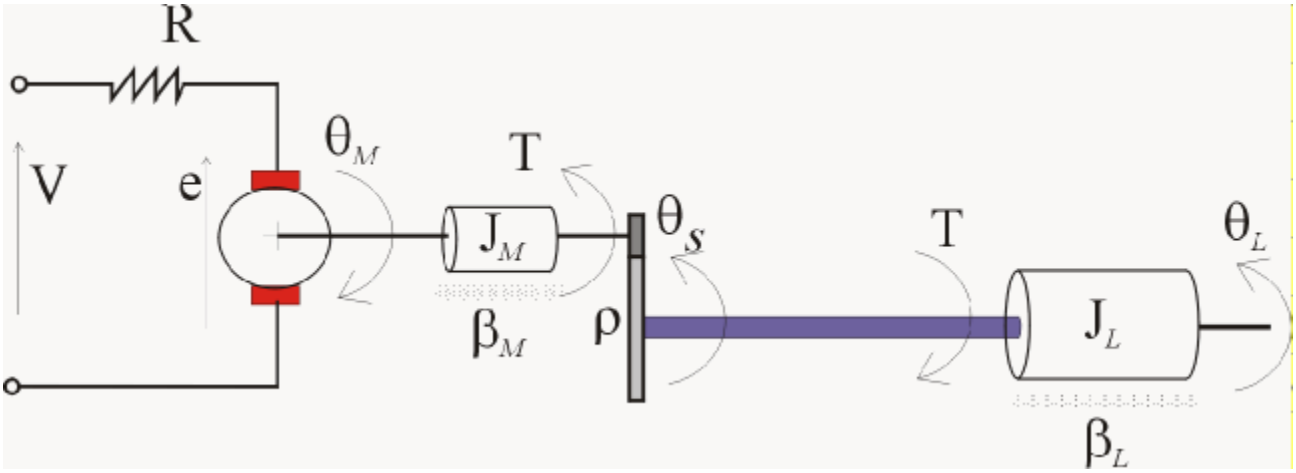
- “Generate Simulink Model from MPC Designer” on page 5-14

Design MPC Controller for Position Servomechanism

This example shows how to design a model predictive controller for a position servomechanism using **MPC Designer**.

System Model

A position servomechanism consists of a DC motor, gearbox, elastic shaft, and load.



The differential equations representing this system are

$$\dot{\omega}_L = -\frac{k_T}{J_L}\left(\theta_L - \frac{\theta_M}{\rho}\right) - \frac{\beta_L}{J_L}\omega_L$$

$$\dot{\omega}_M = \frac{k_M}{J_M}\left(\frac{V - k_M\omega_M}{R}\right) - \frac{\beta_M\omega_M}{J_M} + \frac{k_T}{\rho J_M}\left(\theta_L - \frac{\theta_M}{\rho}\right)$$

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

$$x_p = [\theta_L \ \omega_L \ \theta_M \ \omega_M]^T,$$

then you can model the servomechanism as an LTI state-space system.

$$\dot{x}_p = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{k_T}{J_L} & -\frac{\beta_L}{J_L} & \frac{k_T}{\rho J_L} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{k_T}{\rho J_M} & 0 & -\frac{k_T}{\rho^2 J_M} & -\frac{\beta_M + \frac{k_M^2}{R}}{J_M} \end{bmatrix} x_p + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{k_M}{R J_M} \end{bmatrix} V$$

$$\theta_L = [1 \ 0 \ 0 \ 0] x_p$$

$$T = \left[k_T \ 0 \ -\frac{k_T}{\rho} \ 0 \right] x_p$$

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 $|T| \le 78.5$ Nm. Also, the voltage source physically limits the applied voltage to the range $|V| \le 220$ 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

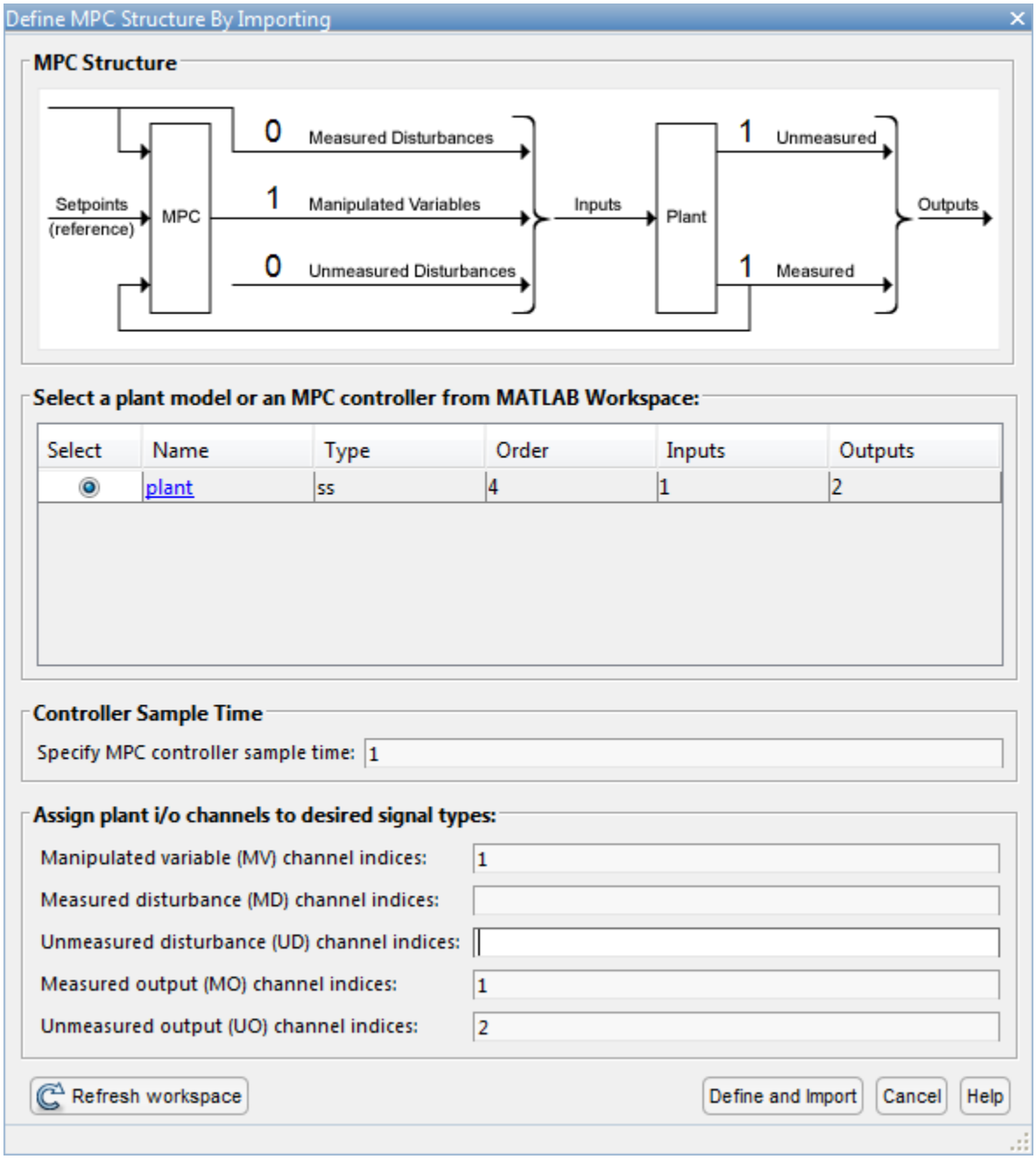
```
mpcDesigner
```

Import Plant and Define Signal Configuration

In **MPC Designer**, on 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



Click **Define and Import**.

MPC Designer 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.

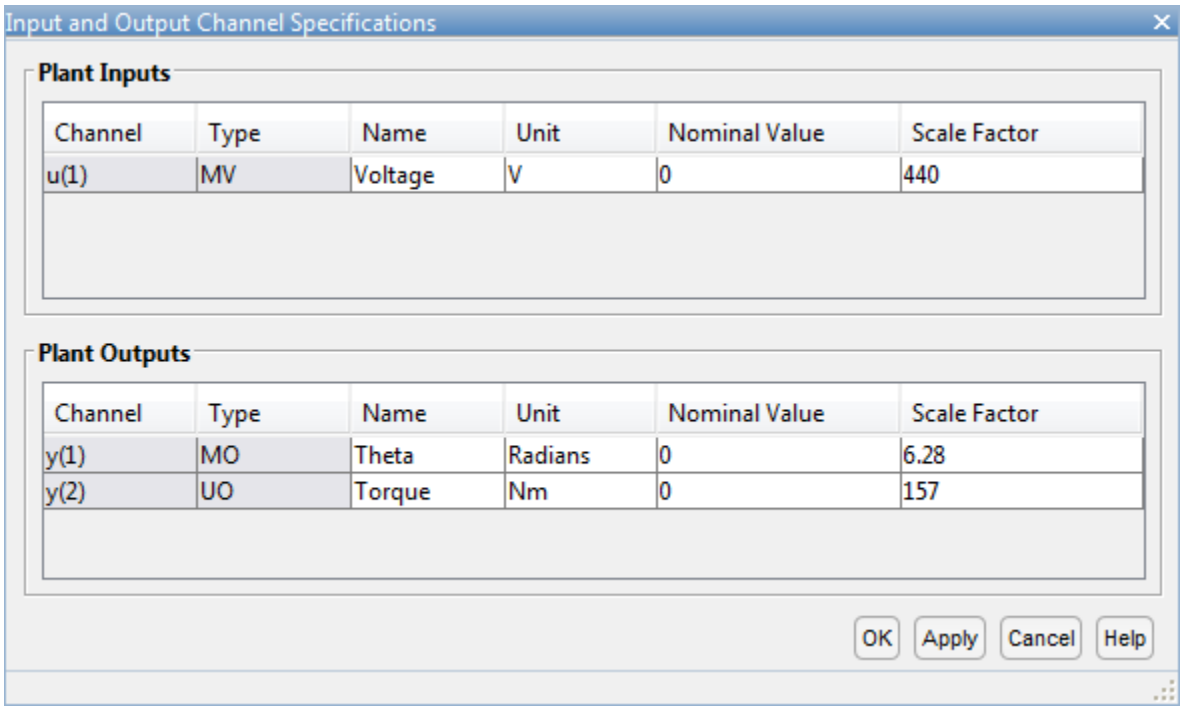
Define Input and Output Channel Attributes

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.

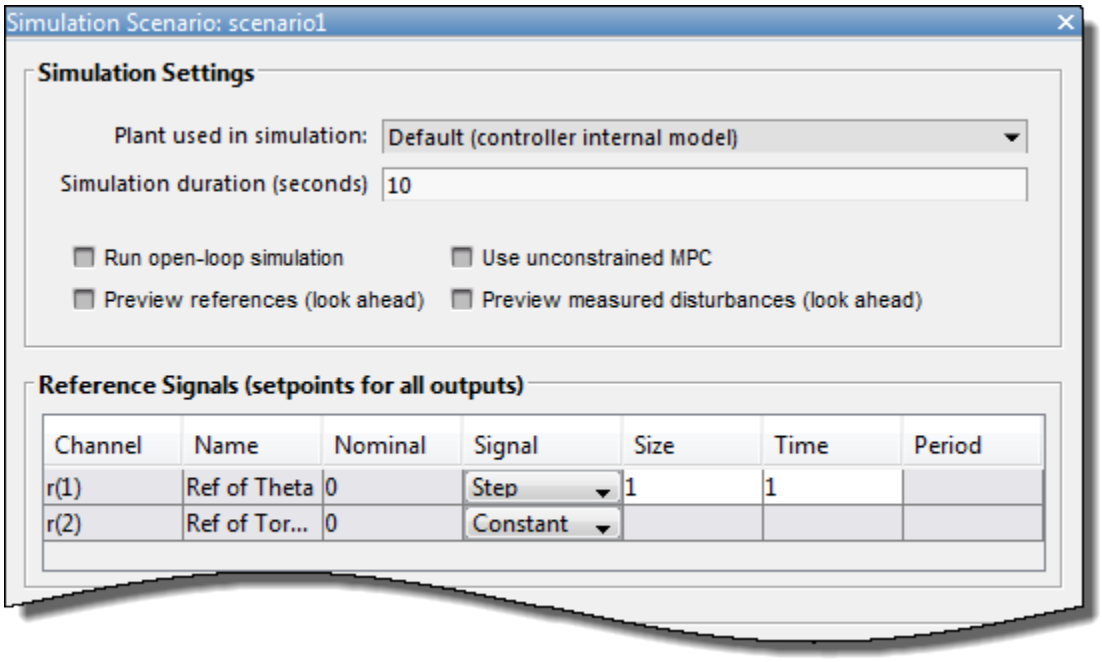
Modify Scenario To Simulate Angular Position Step Response

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.

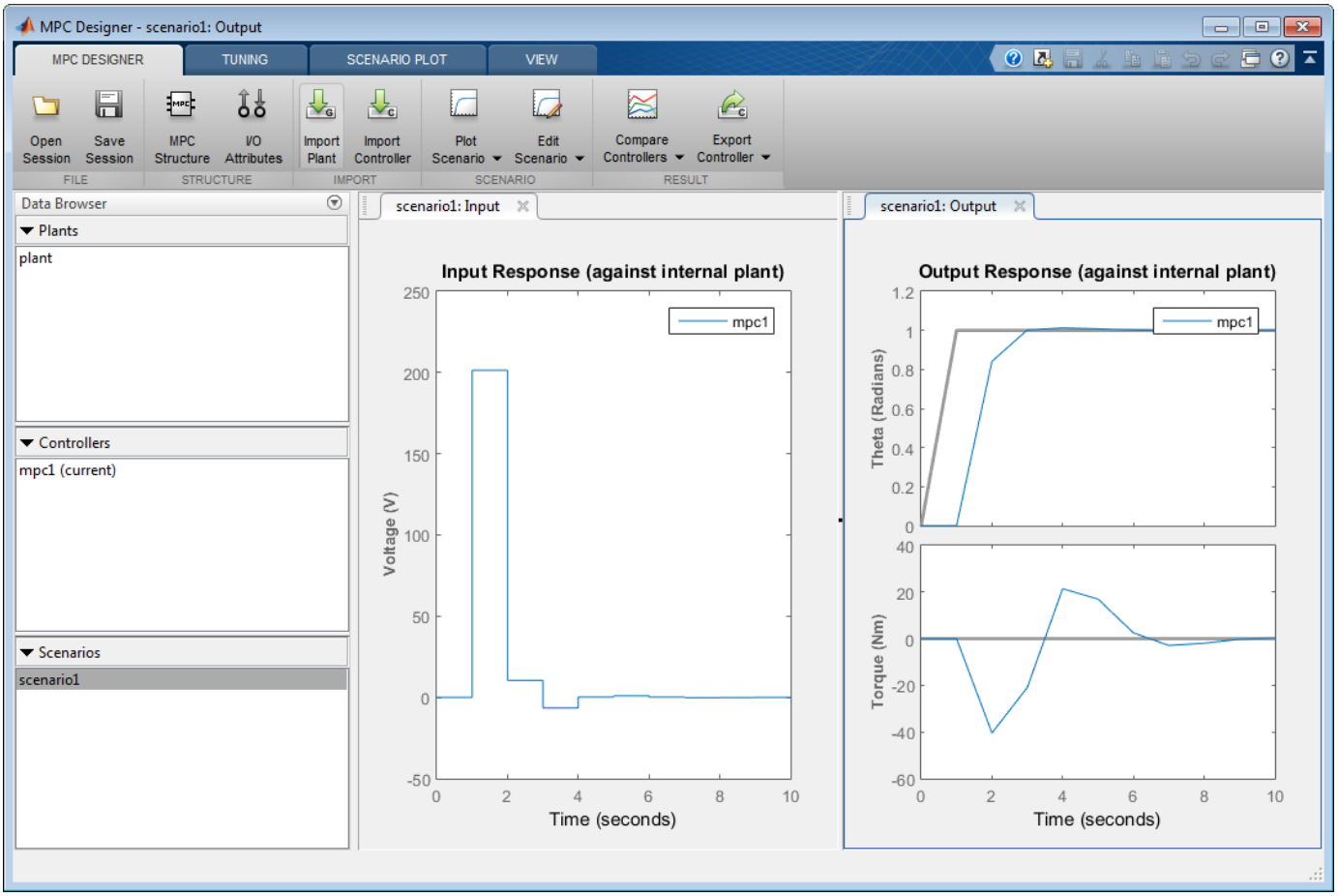
In the **Reference Signals** table, keep the default configuration for the first channel. These settings create a `Step` change of `1` radian in the angular position setpoint at a **Time** of `1` second.

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.



Specify Controller Sample Time and Horizons

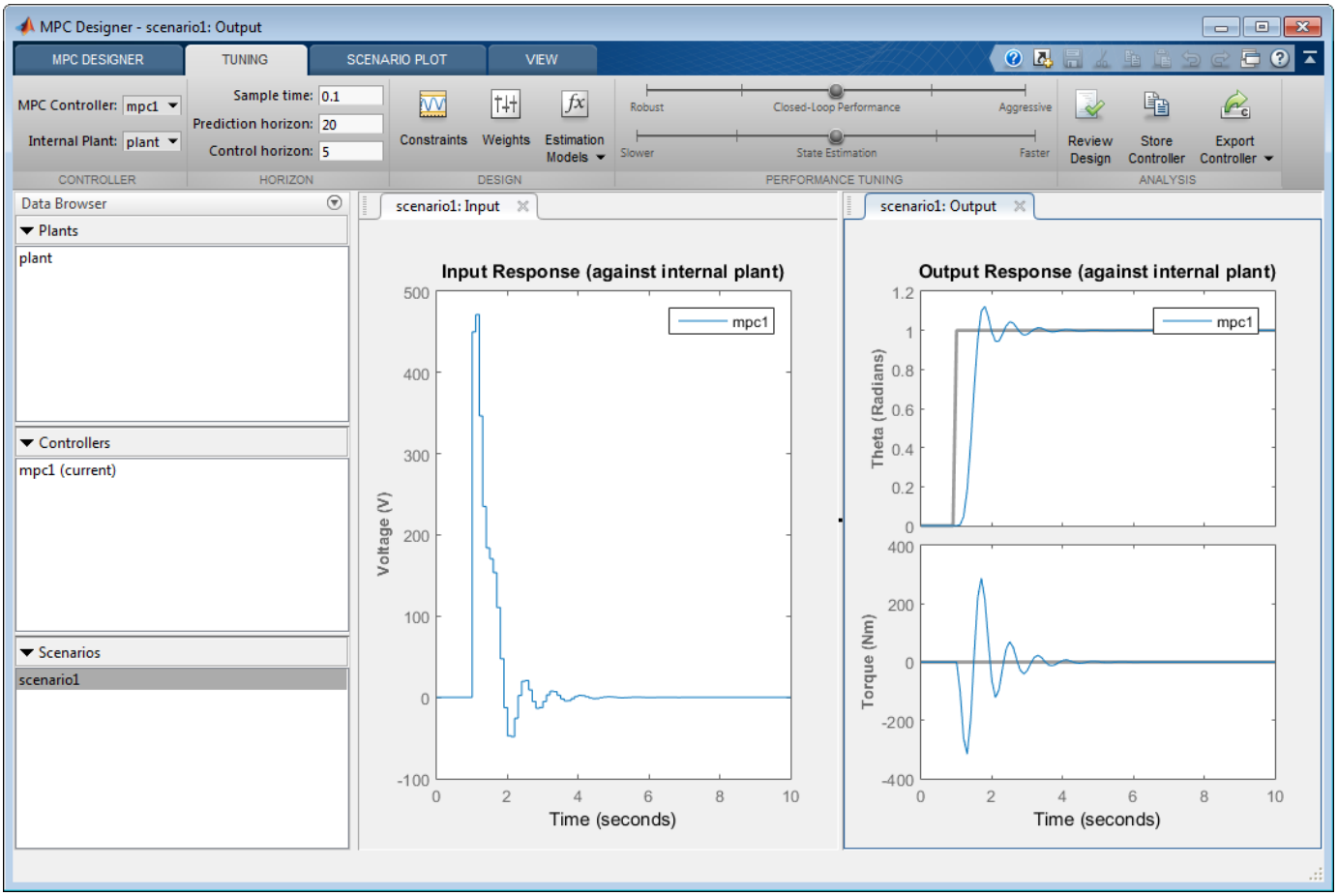
On the **Tuning** tab, in the **Horizon** section, specify a **Sample time** of `0.1` seconds.

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`.



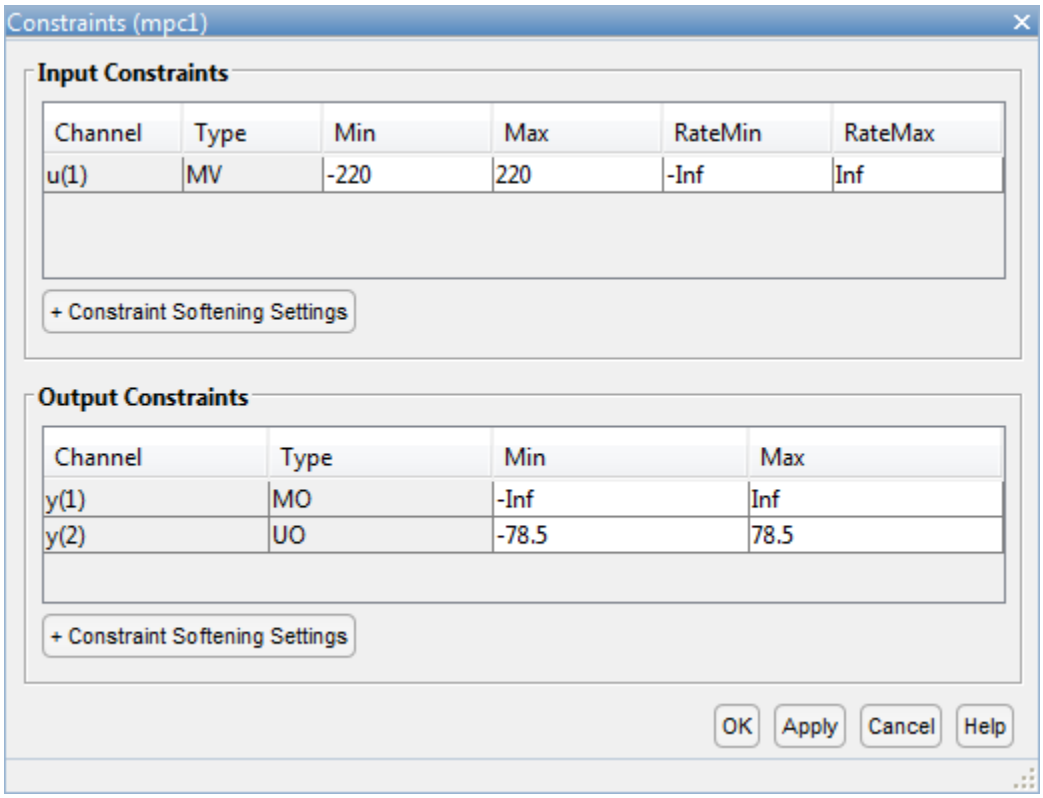
As you update the sample time and horizon values, the **Input Response** and **Output Response** plots update automatically. Both the input voltage and torque values exceed the constraints defined in the system model specifications.

Specify Constraints

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

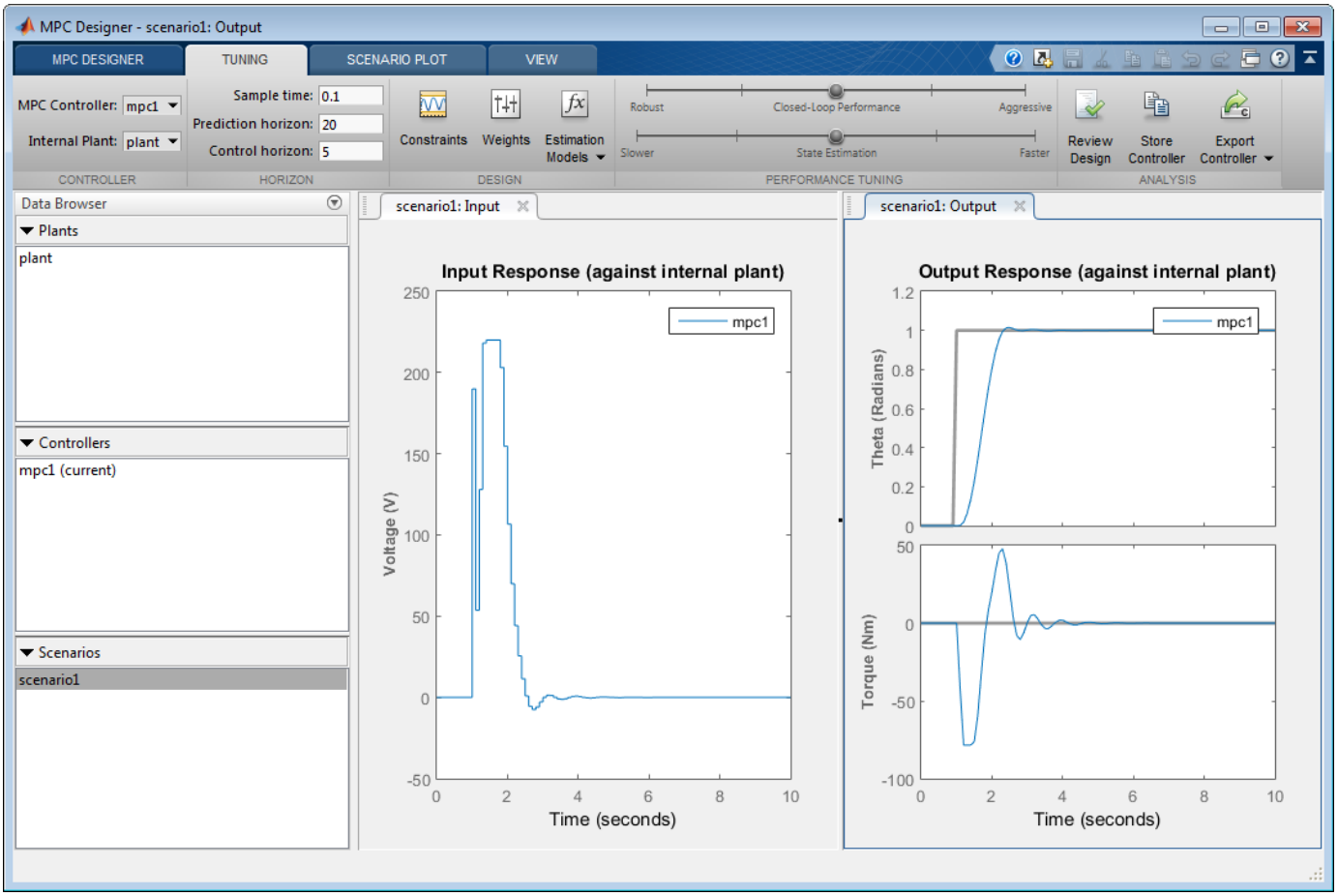
In the Constraints dialog box, in the **Input Constraints** section, specify the **Min** and **Max** voltage values for the manipulated variable (MV).

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, `–Inf` and `Inf` respectively

Click **OK**.

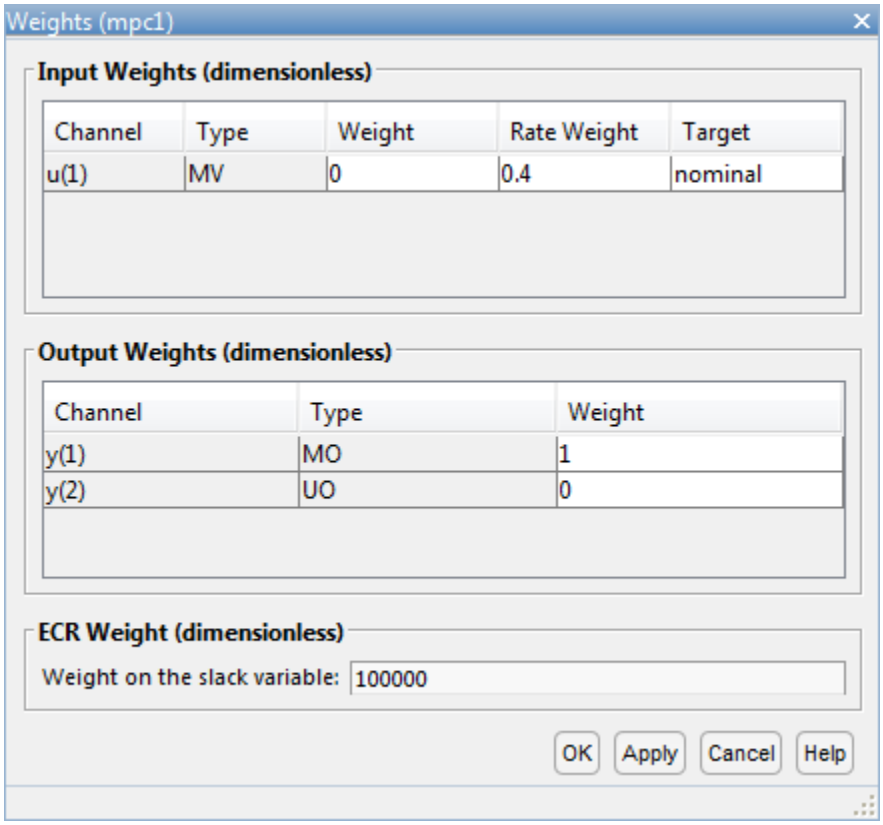


The response plots update to reflect the new constraints. In the **Input Response** plot, there are undesirable large changes in the input voltage.

Specify Tuning Weights

In the **Design** section, select **Weights**.

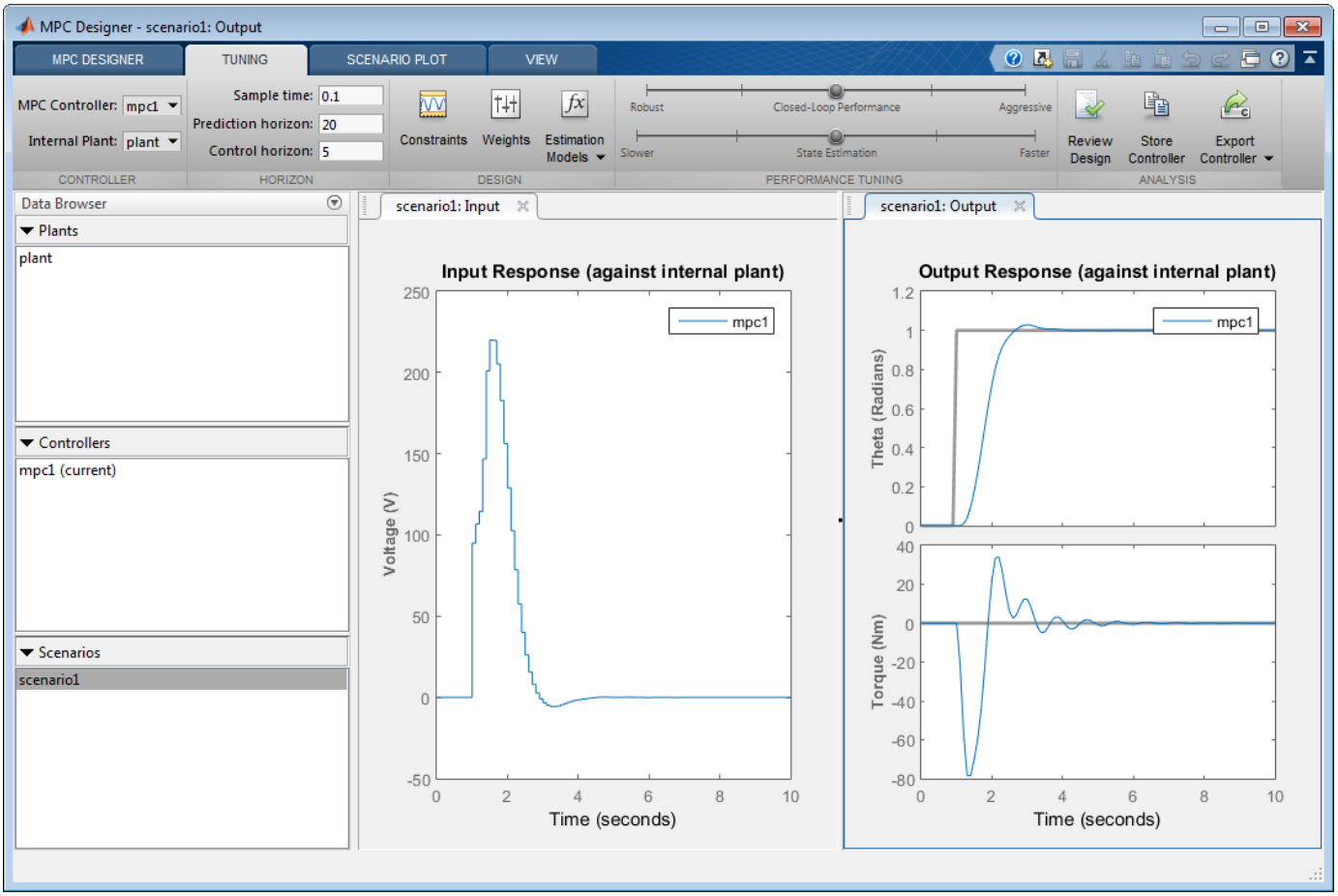
In the Weights dialog box, in the **Input Weights** table, increase the manipulated variable **Rate Weight**.



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.

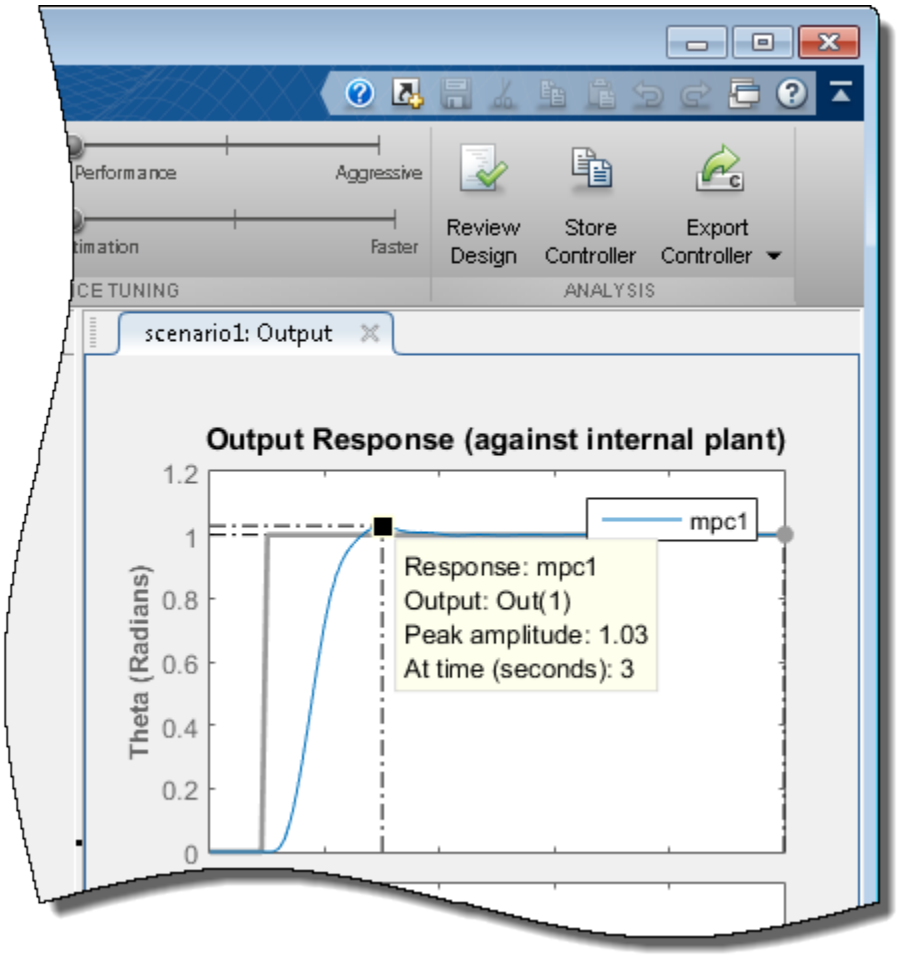
Click **OK**.



The response plots update to reflect the increased rate weight. The **Input Response** is smoother with smaller voltage changes.

Examine Output Response

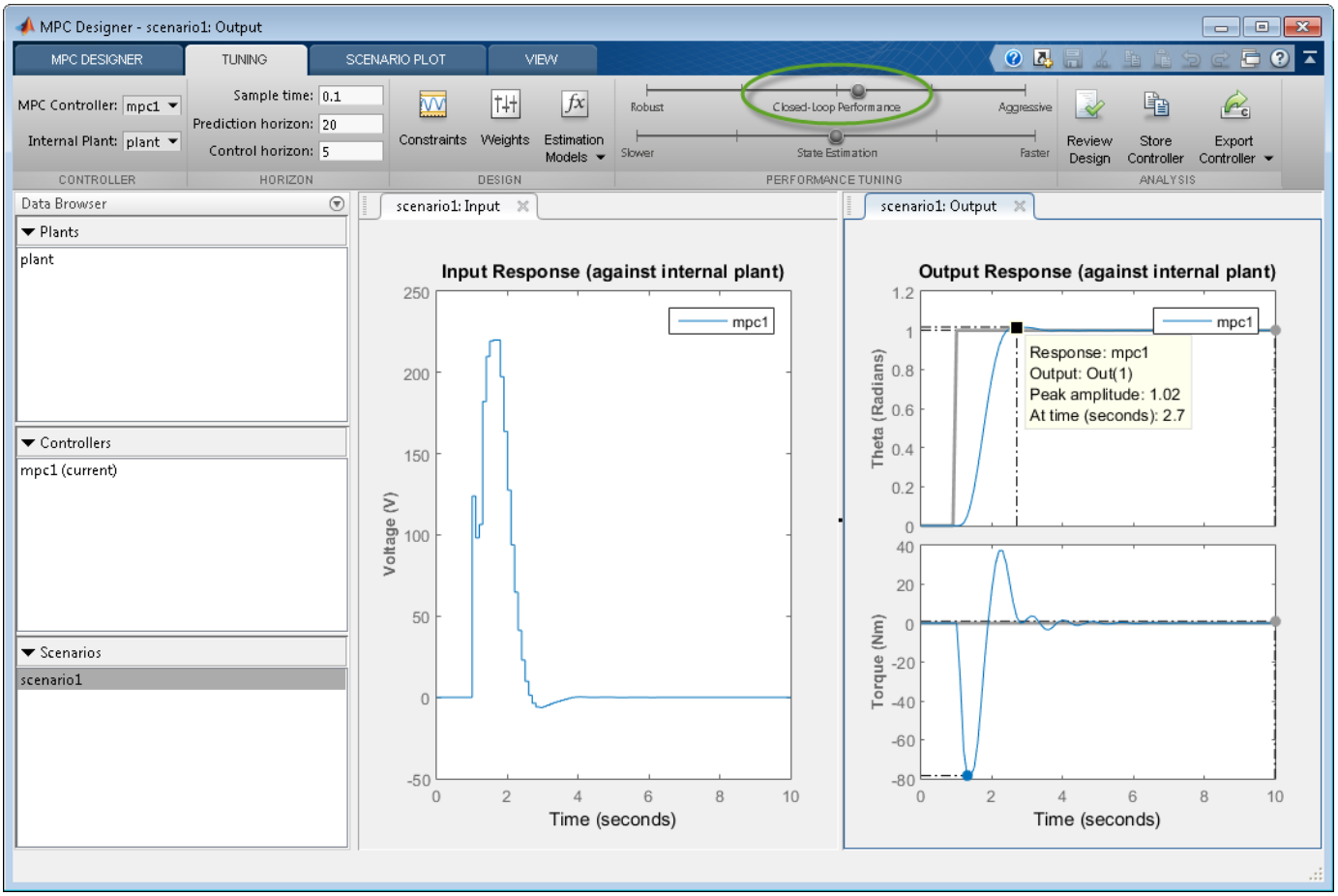
In the **Output Response** plot, right-click the **Theta** plot area, and select **Characteristics > Peak Response**.



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.

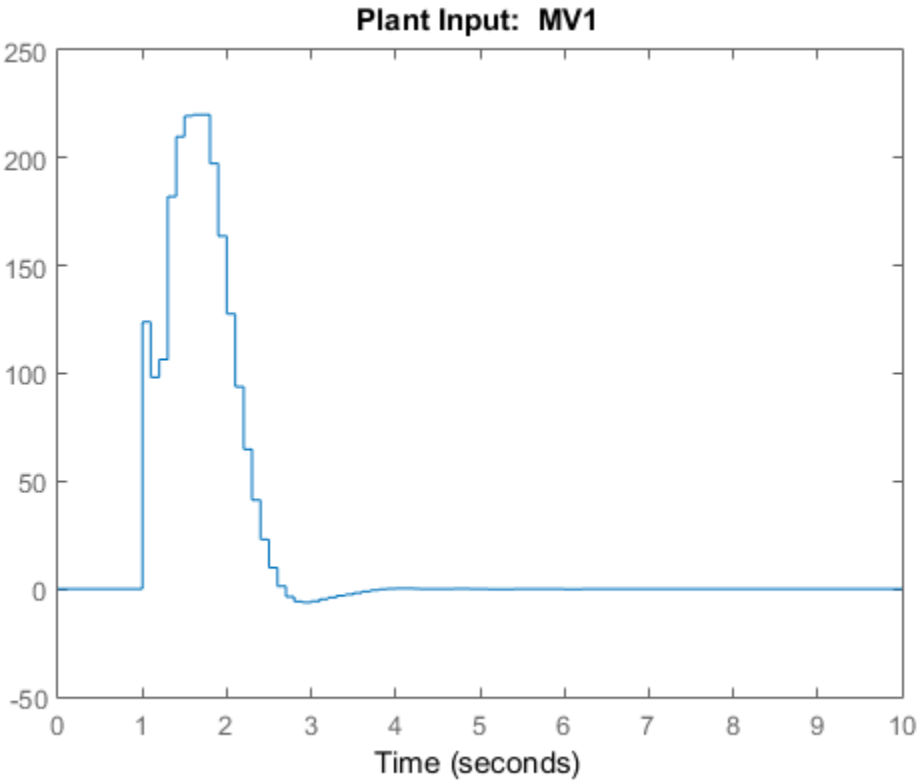
```
%% create MPC controller object with sample time
mpc1 = mpc(plant_C, 0.1);
%% specify prediction horizon
mpc1.PredictionHorizon = 20;
%% specify control horizon
mpc1.ControlHorizon = 5;
```

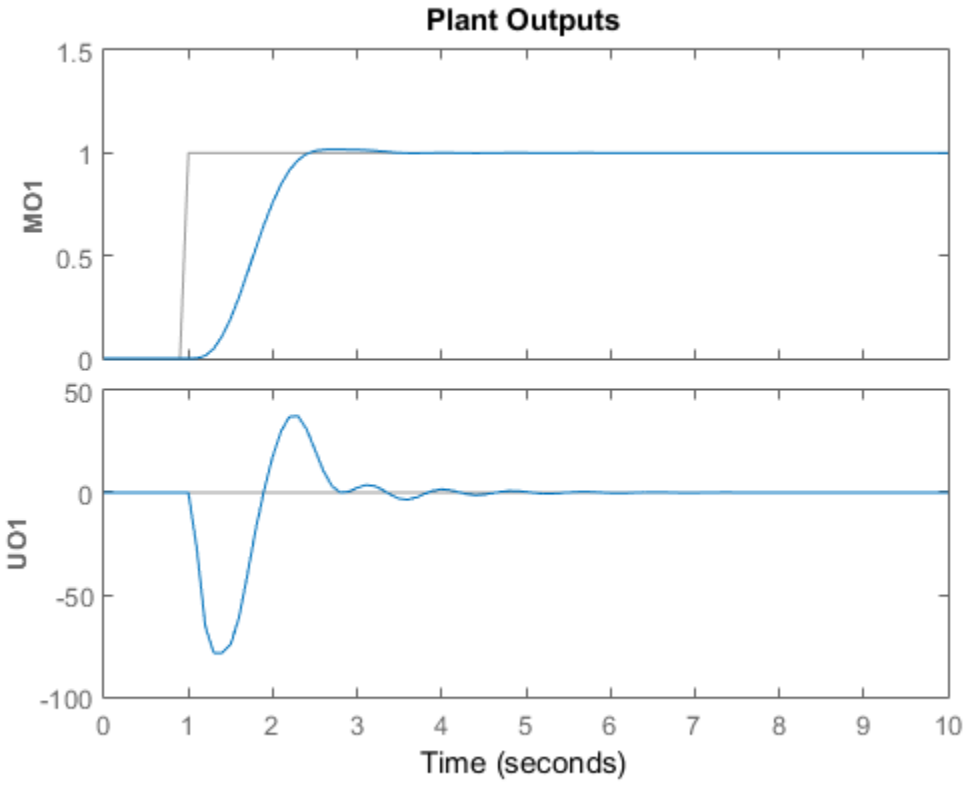
```
%% specify nominal values for inputs and outputs
mpc1.Model.Nominal.U = 0;
mpc1.Model.Nominal.Y = [0;0];
%% specify scale factors for inputs and outputs
mpc1.MV(1).ScaleFactor = 440;
mpc1.OV(1).ScaleFactor = 6.28;
mpc1.OV(2).ScaleFactor = 157;
%% specify constraints for MV and MV Rate
mpc1.MV(1).Min = -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**.


Plant Input: MV1
250
200
150
100
50
0
-50
0
1
2
3
4
5
6
7
8
9
10
Time (seconds)



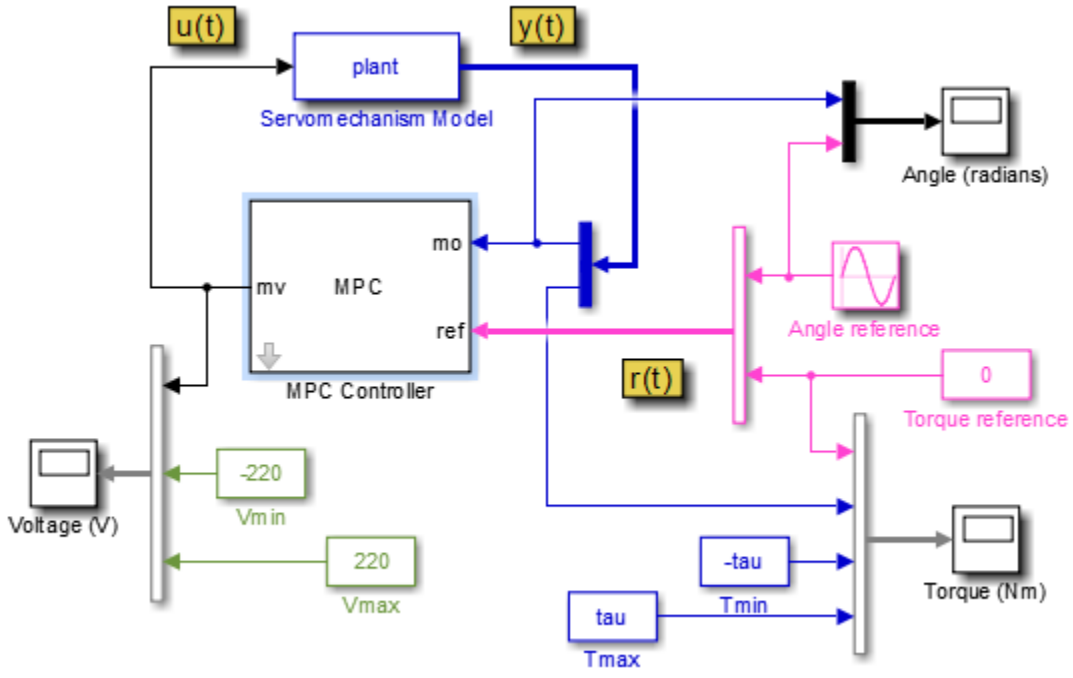
The script creates the controller, `mpc1`, and runs the simulation scenario. The input and output responses match the simulation results from the app.

Validate Controller Performance In Simulink

If you have a Simulink model of your system, you can simulate your controller and validate its performance. Simulink functionality is not supported in MATLAB Online.

Open the model.

```
open_system('mpc_motor')
```

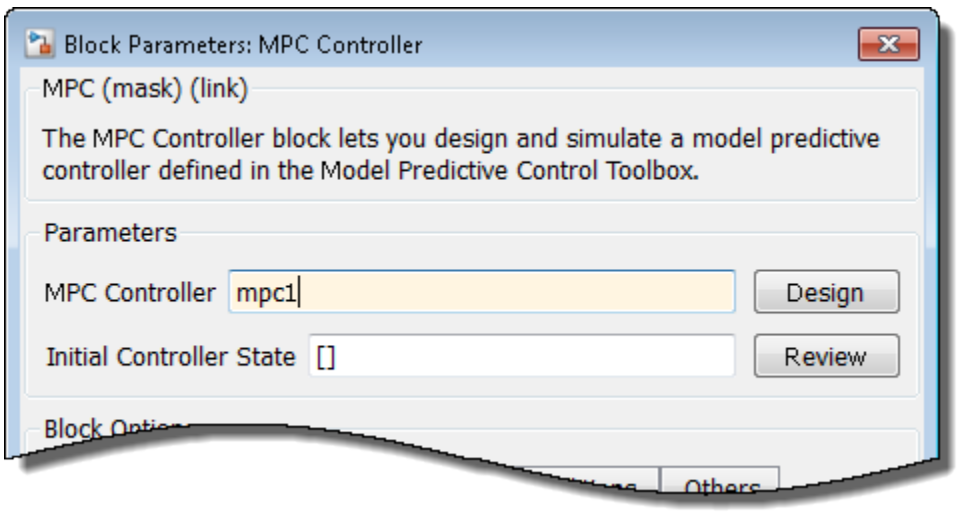


This model uses an MPC Controller block to control a servomechanism plant. The Servomechanism Model block is already configured to use the `plant` model from the MATLAB workspace.

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 Designer | `mpc` | MPC Controller

More About

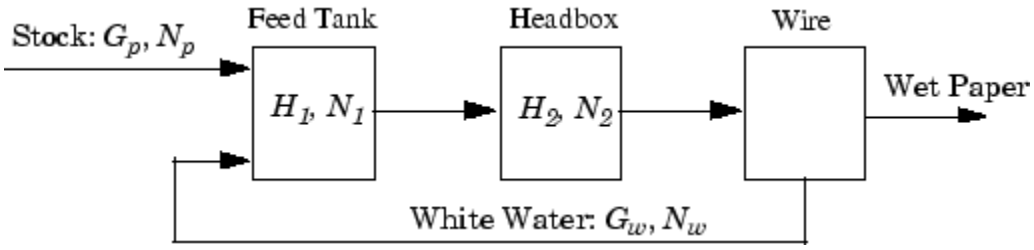
- “Design Controller Using MPC Designer”
- “Design MPC Controller at the Command Line”

Design MPC Controller for Paper Machine Process

This example shows how to design a model predictive controller for a nonlinear paper machine process using **MPC Designer**.

Plant 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

$$x = \left[H_1\ H_2\ N_1\ N_2\right]^T$$

- 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_pmmodel` implements the nonlinear model equations. To view this S-function, enter the following.

```
edit mpc_pmmodel
```

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.

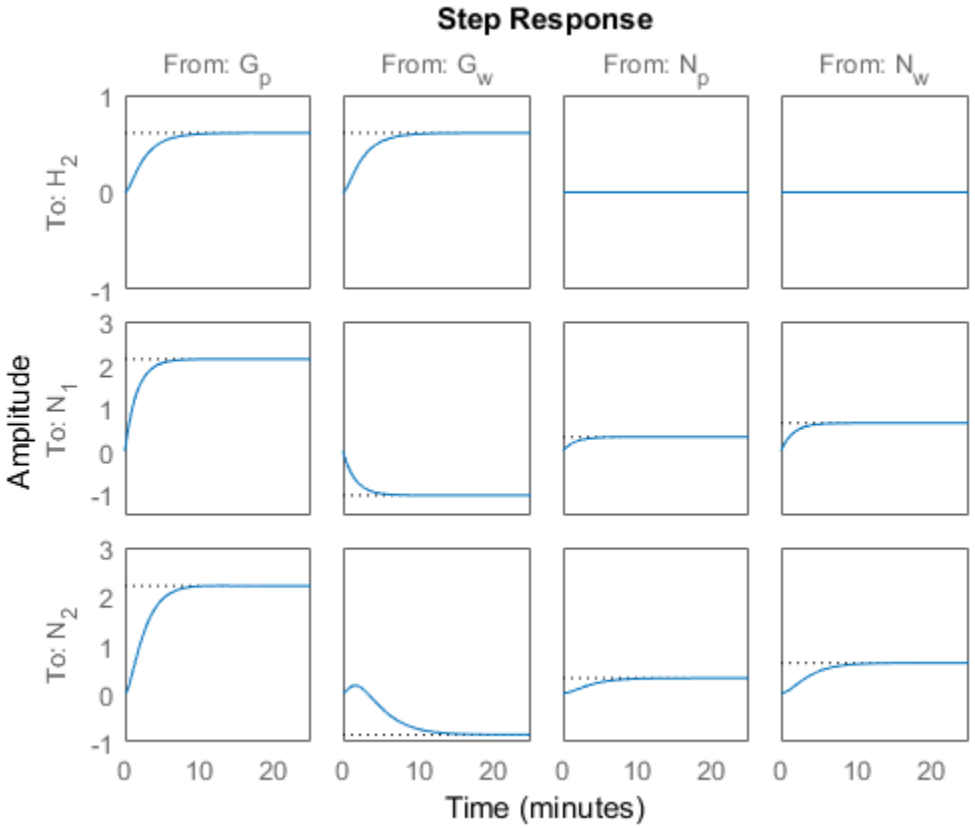
```
PaperMach.InputName = {'G_p','G_w','N_p','N_w'};
PaperMach.OutputName = {'H_2','N_1','N_2'};
```

Specify the model time units.

```
PaperMach.TimeUnit = 'minutes';
```

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.

Import Plant Model and Define Signal Configuration

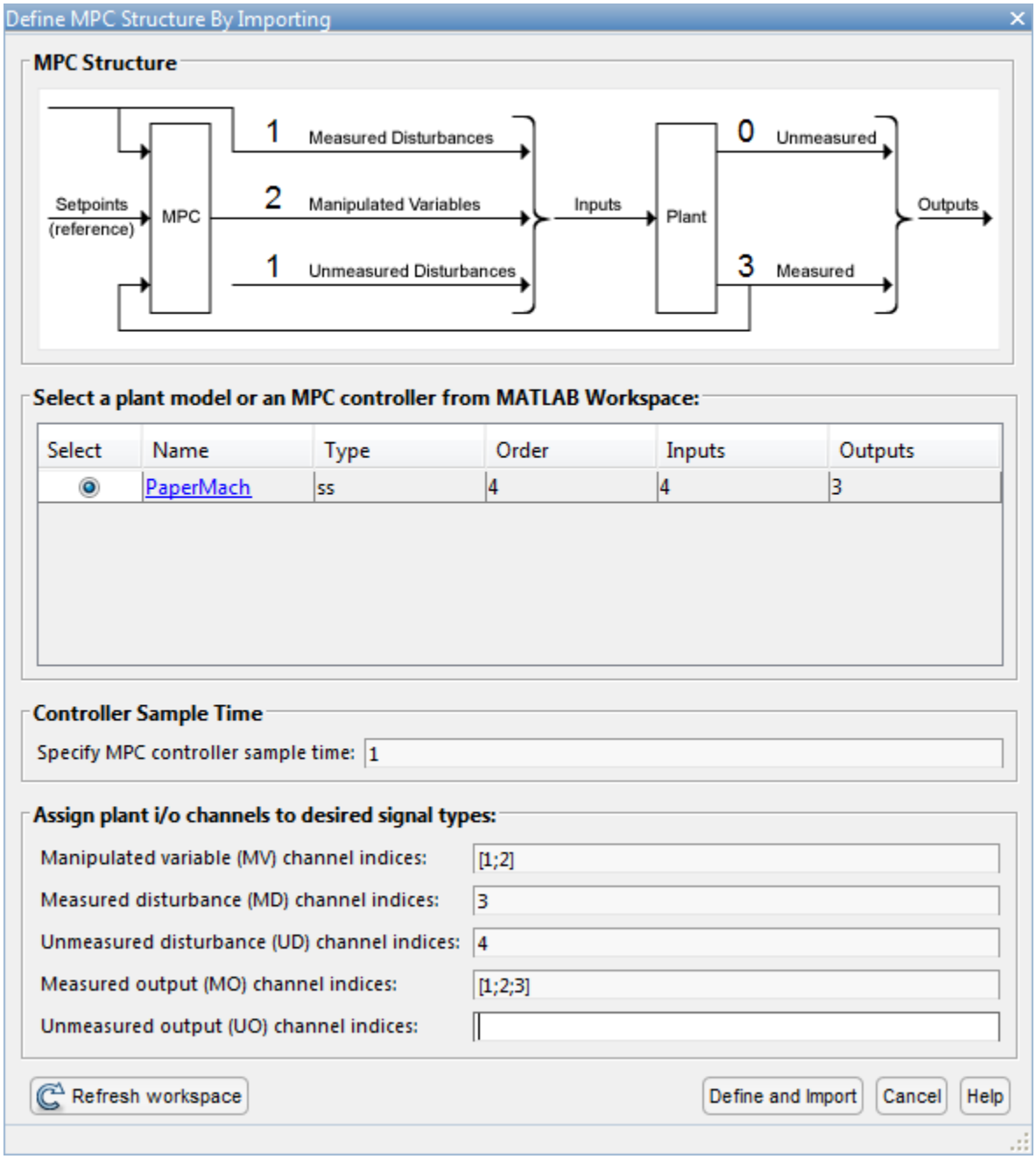
Open the **MPC Designer** app.

```
mpcDesigner
```

In **MPC Designer**, 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



Tip To find the correct channel indices, click the `PaperMach` model **Name** to view additional model details.

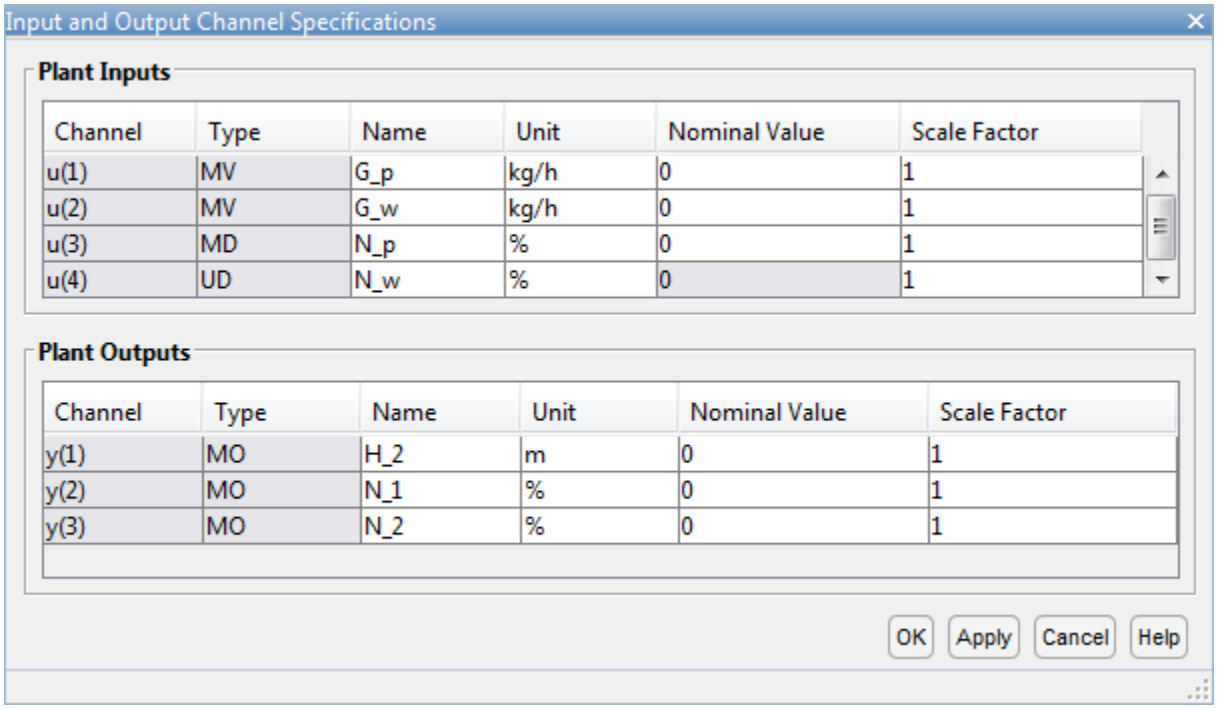
Click **Define and Import**.

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.

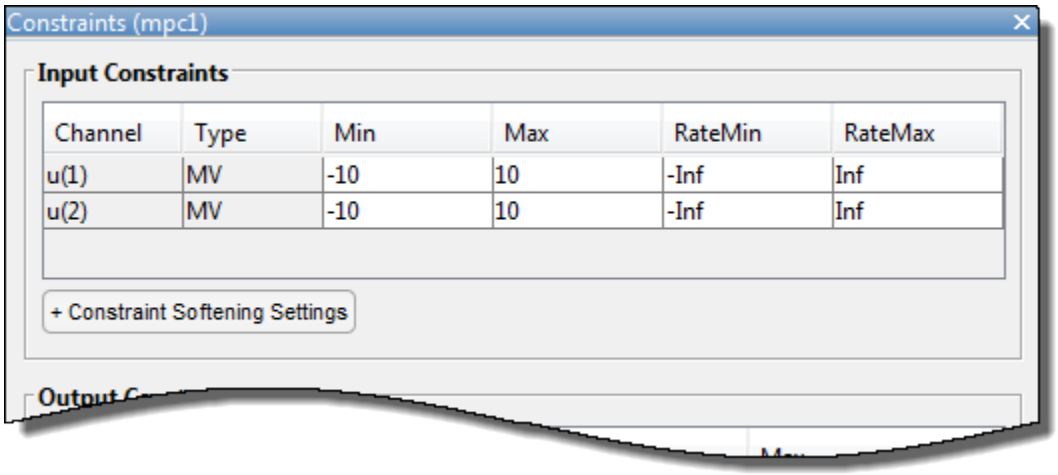
Specify Controller Sample Time and Horizons

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.



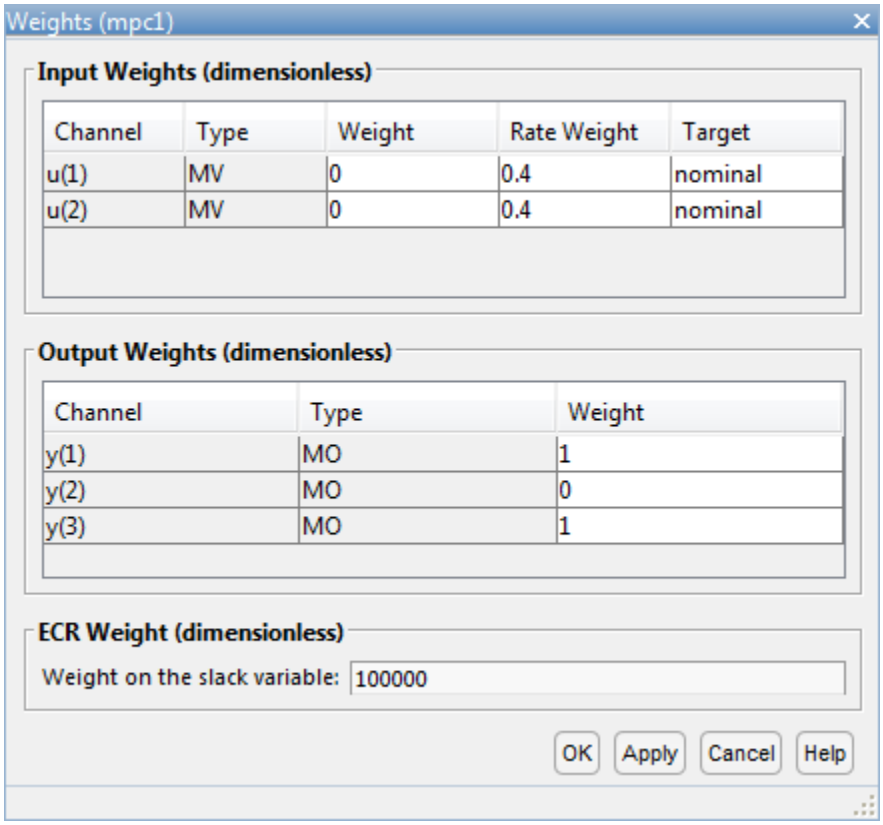
Click **OK**.

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.

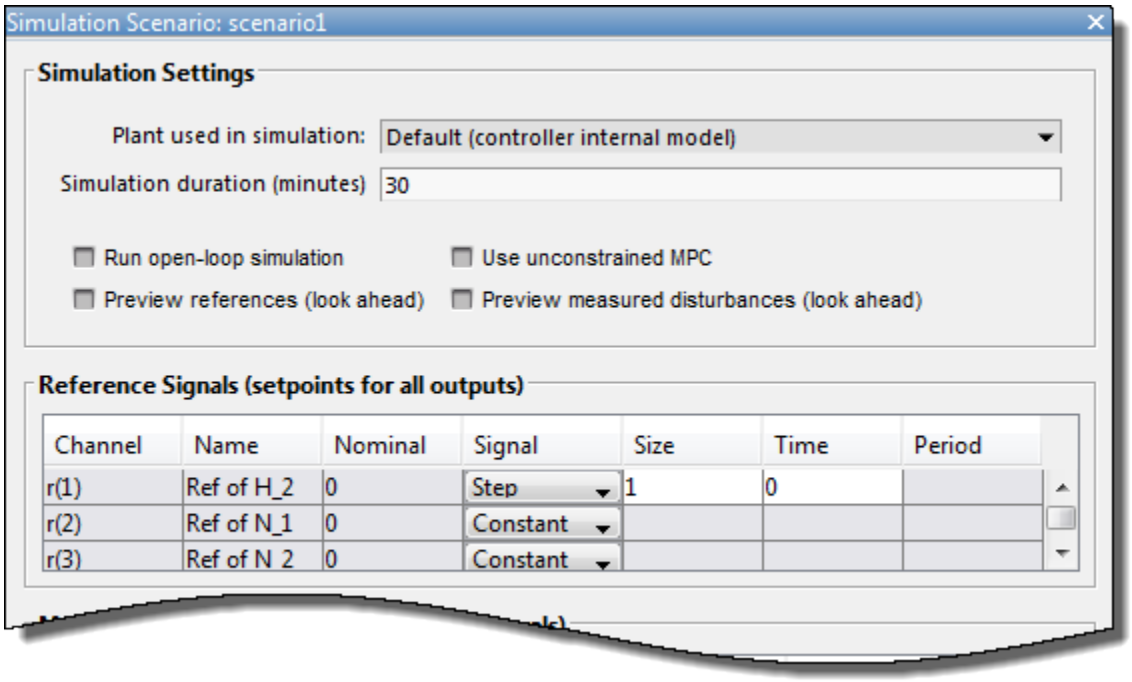
Simulate H_2 Setpoint Step Response

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.

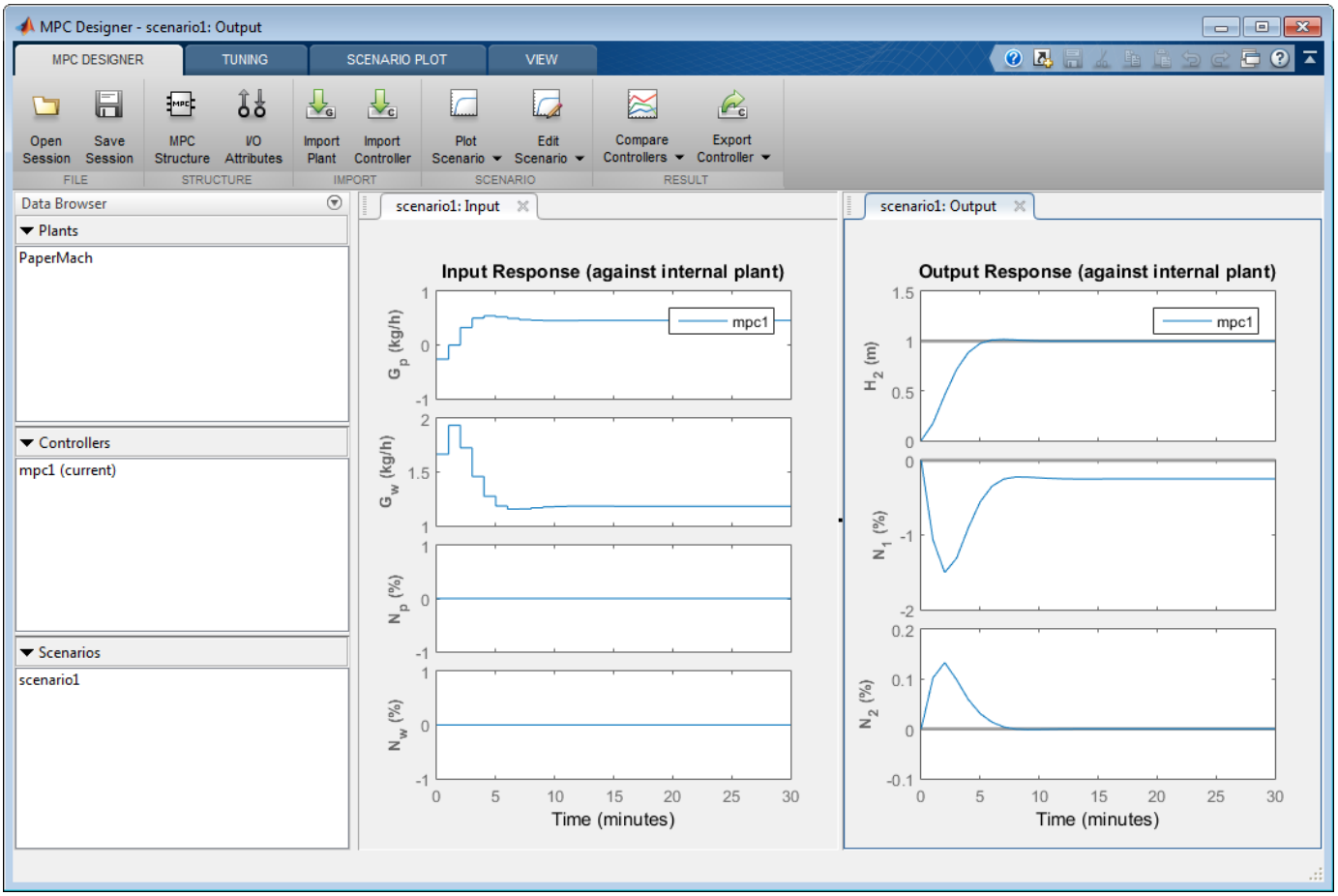
In the **Reference Signals** table, in the **Signal** drop-down list, select `Step` for the first output. Keep the step **Size** at `1` and specify a step **Time** of `0`.

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.



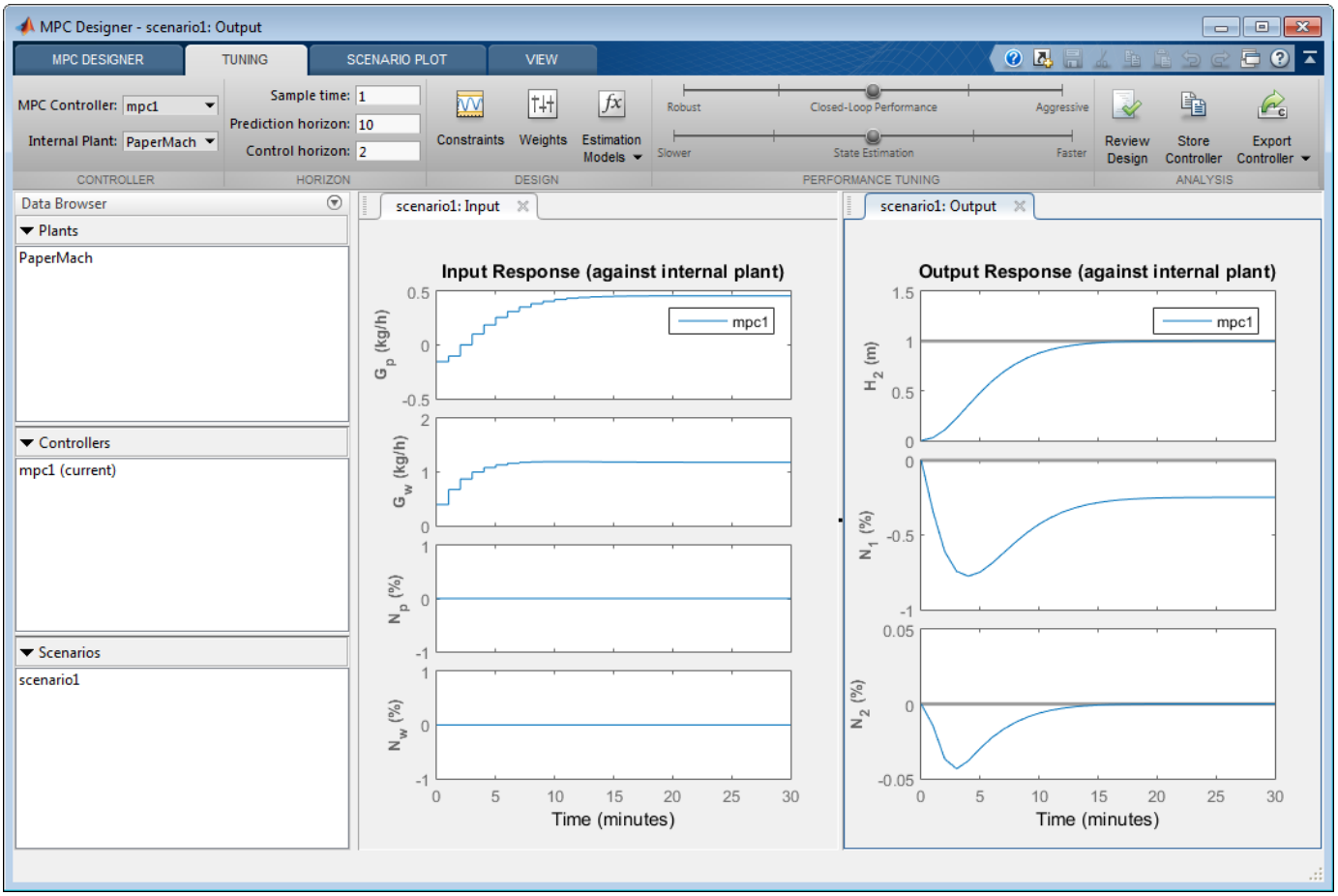
The initial design uses a conservative control effort to produce a robust controller. The response time for output H_2 is about 7 minutes. To reduce this response time, you can decrease the sample time, reduce the manipulated variable rate weights, or reduce the manipulated variable rate constraints.

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.



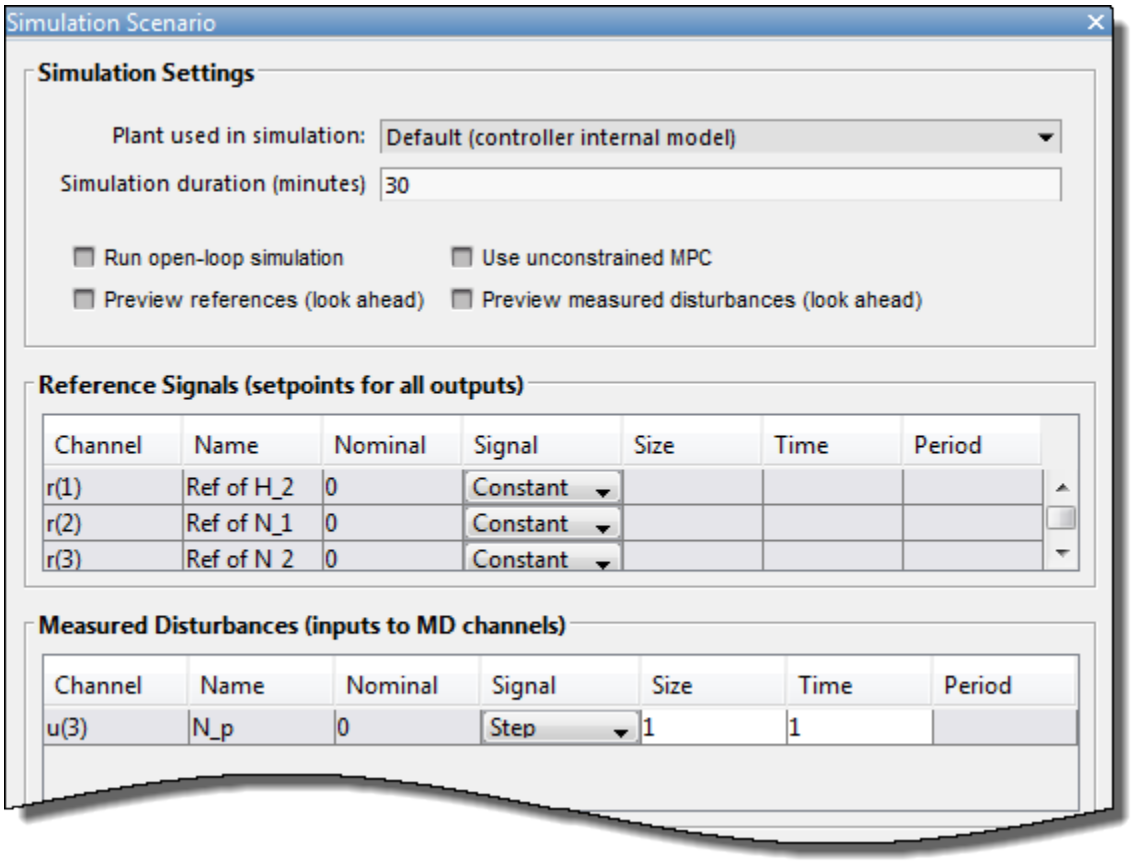
The controller places more emphasis on eliminating errors in feed tank consistency, N_2, which significantly decreases the peak absolute error. The trade-off is a longer response time of about 17 minutes for the feed tank level, H_2.

Test Controller Feedforward Response to Measured Disturbances

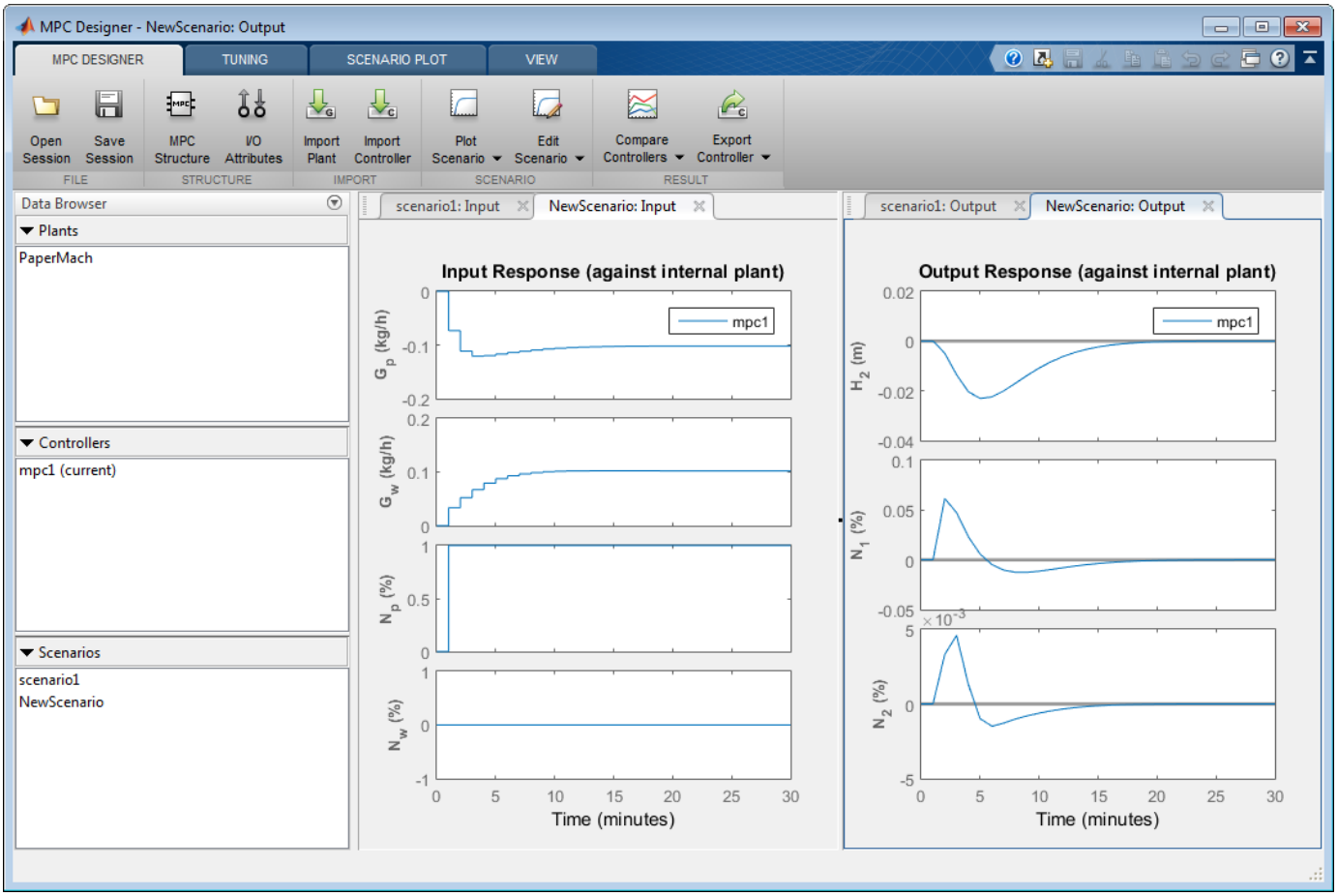
On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > New Scenario**.

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.



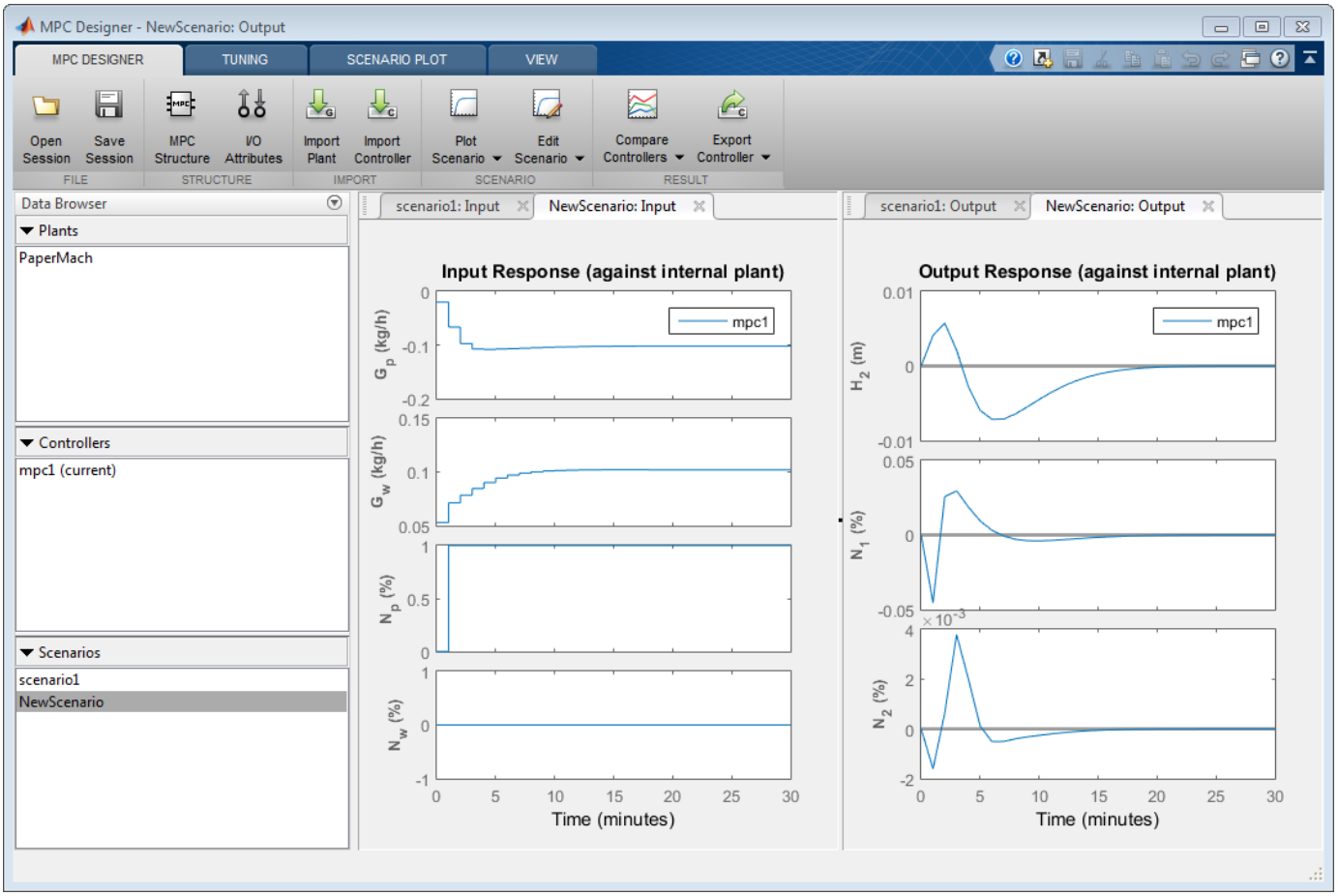
As shown in the **NewScenario: Output** plot, both H_2 and N_2 deviate little from their setpoints.

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**.



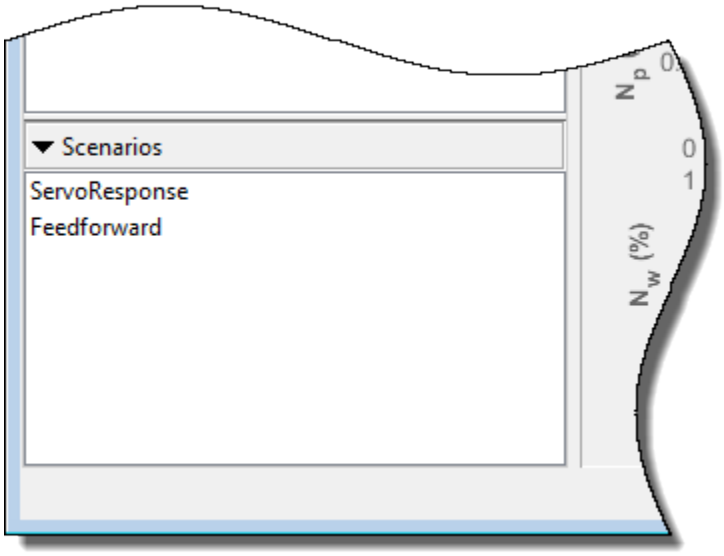
The manipulated variables begin changing before the measured disturbance occurs because the controller uses the known future disturbance value when computing its control action. The output disturbance values also begin changing before the disturbance occurs, which reduces the magnitude of the output 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**.

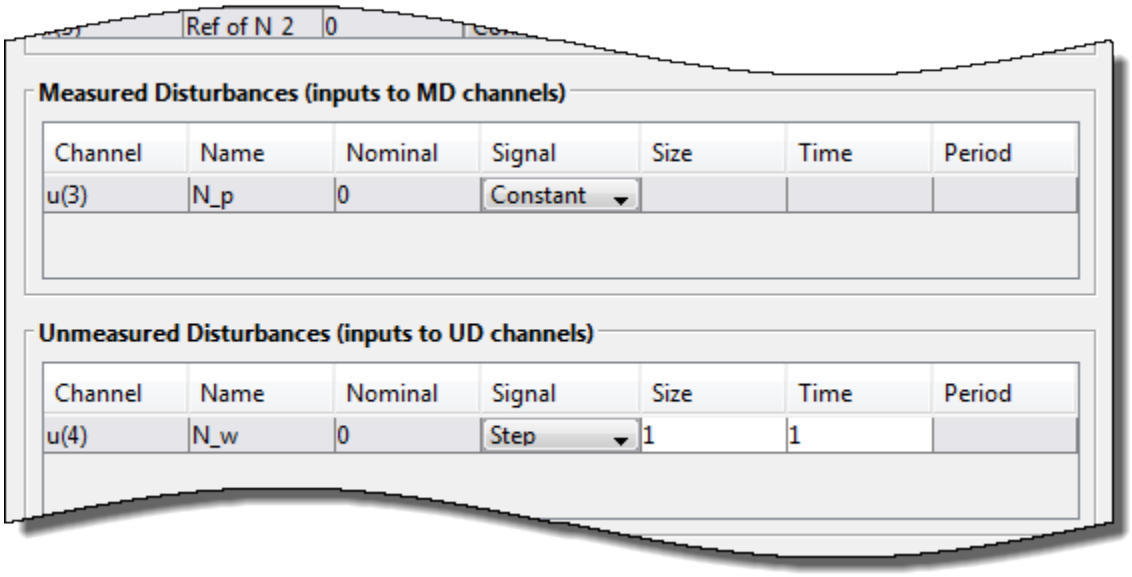
Double-click the new scenario, and rename it `Feedback`.

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.

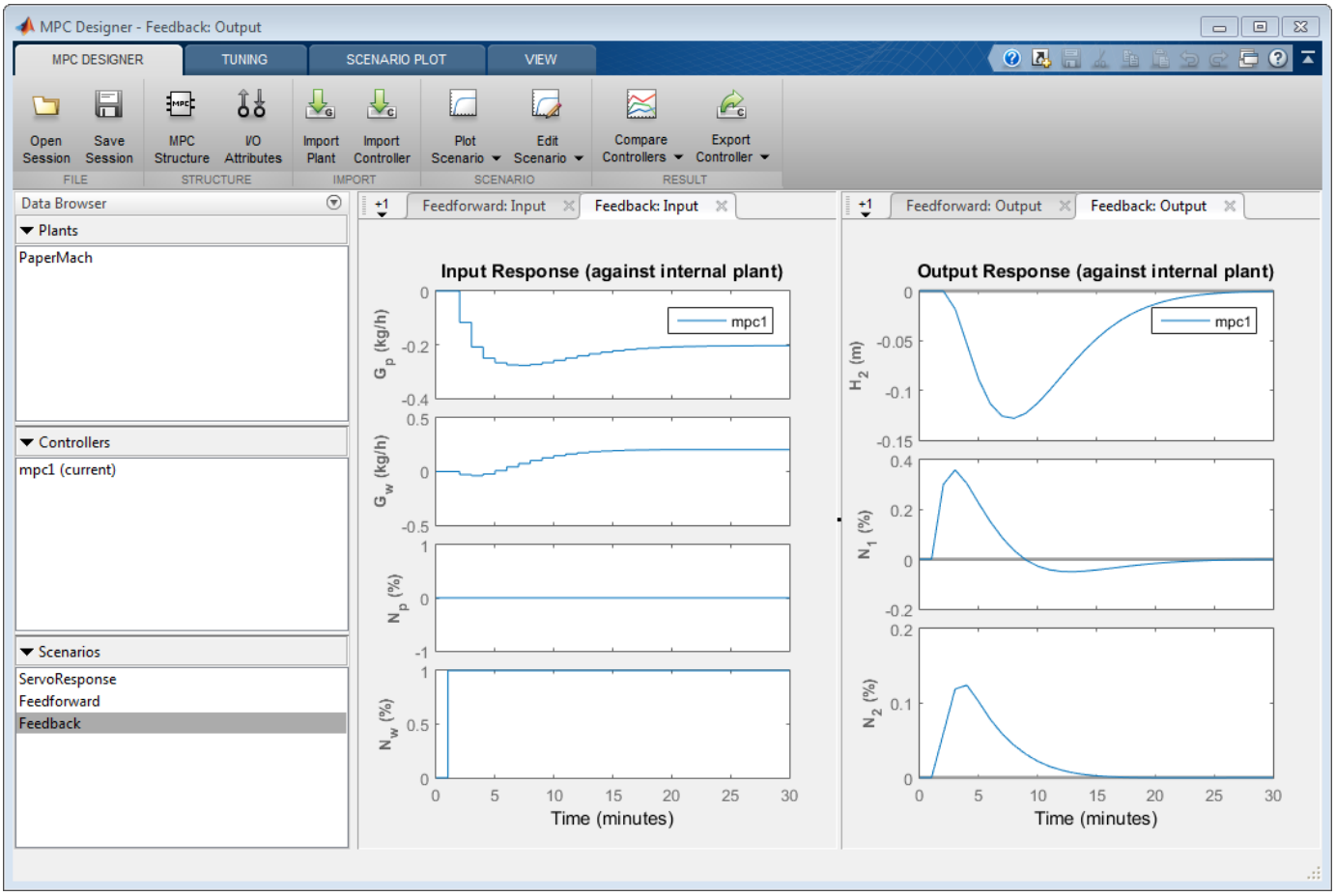
In the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select `Step` to simulate a sudden, sustained unmeasured input disturbance.

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**.



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.

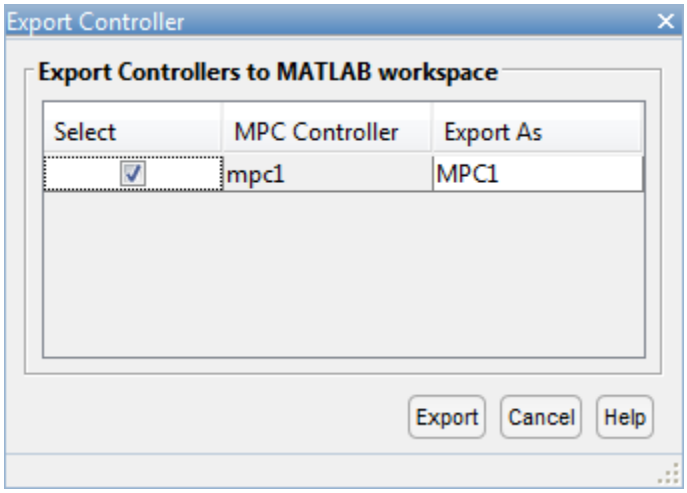
Since the input/output imbalance is a known feature of the paper machine plant model, and you intentionally set one of the output weights to zero to correct for the imbalance, neither warning indicates an issue with the controller design.

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.



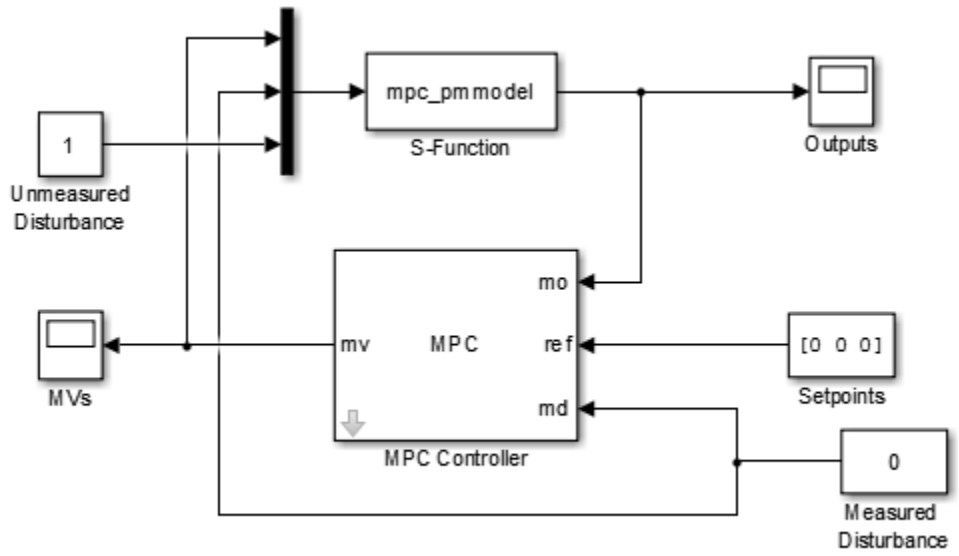
Click **Export** to save a copy of the controller to the MATLAB workspace.

Open and Simulate Simulink Model

If you have a Simulink model of your system, you can simulate your controller and validate its performance. Simulink functionality is not supported in MATLAB Online.

Open the model.

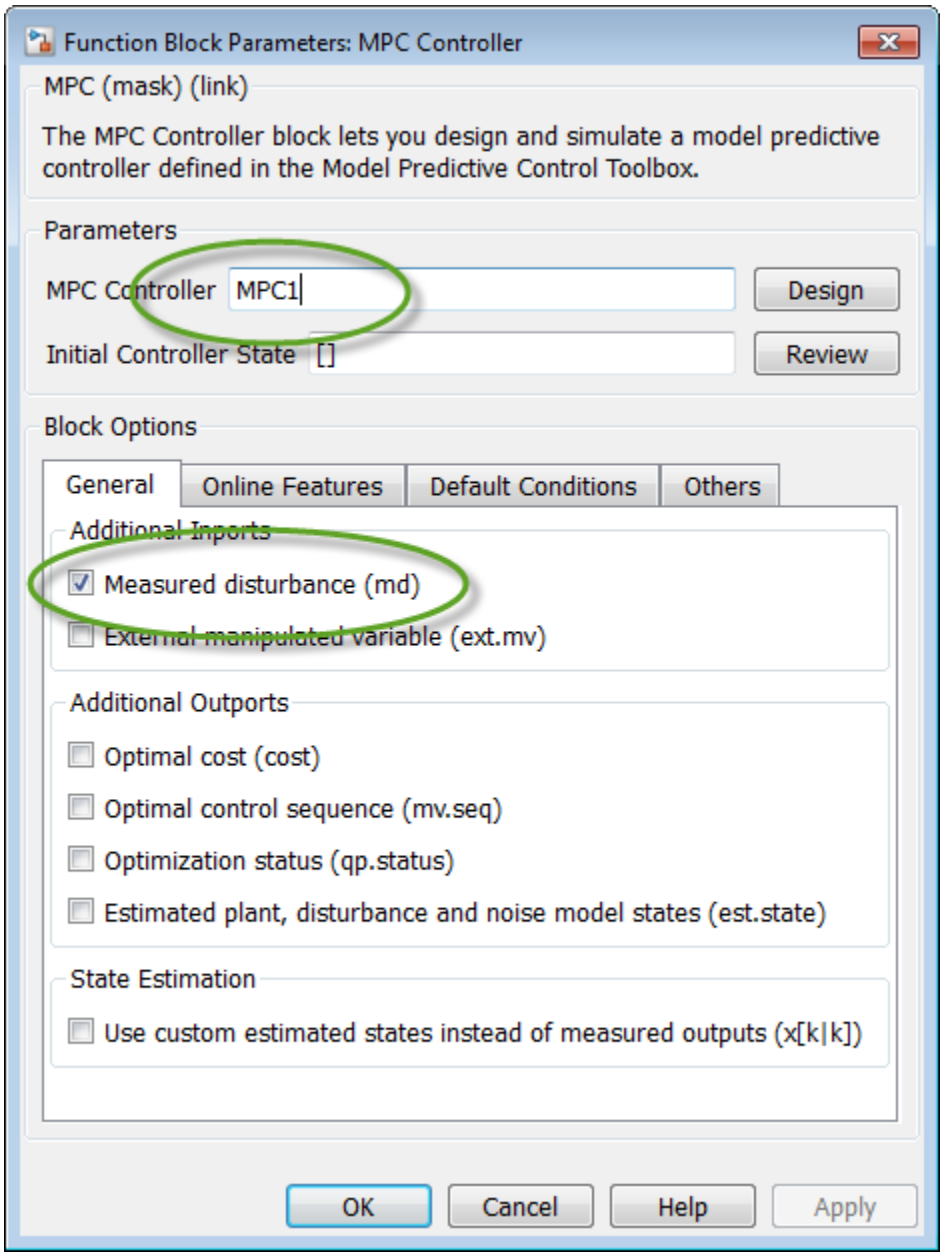
```
open_system('mpc_papermachine')
```



The MPC Controller block controls the nonlinear paper machine plant model, which is defined using the S-Function `mpc_pmmodel`.

The model is configured to simulate a sustained unmeasured disturbance of size `1`.

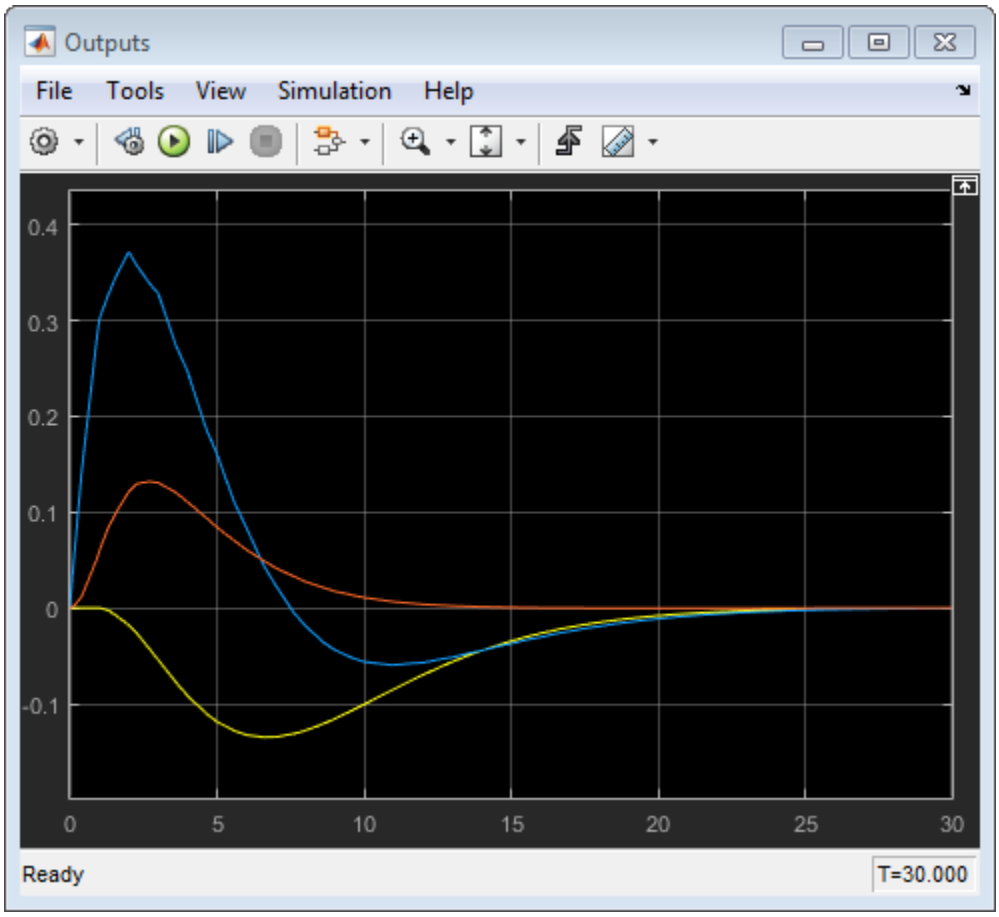
Double-click the MPC Controller block.



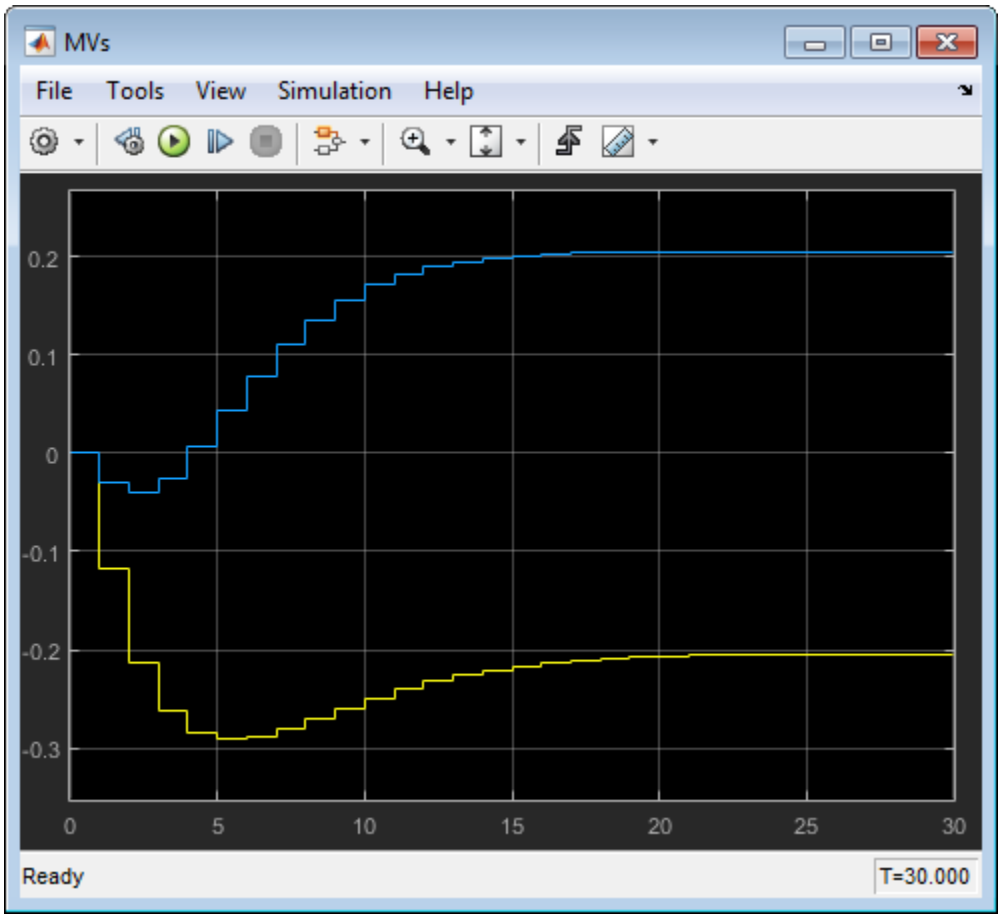
The MPC Controller block is already configured to use the `MPC1` controller that was previously exported to the MATLAB workspace.

Also, the **Measured disturbance** option is selected to add the `md` inport 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

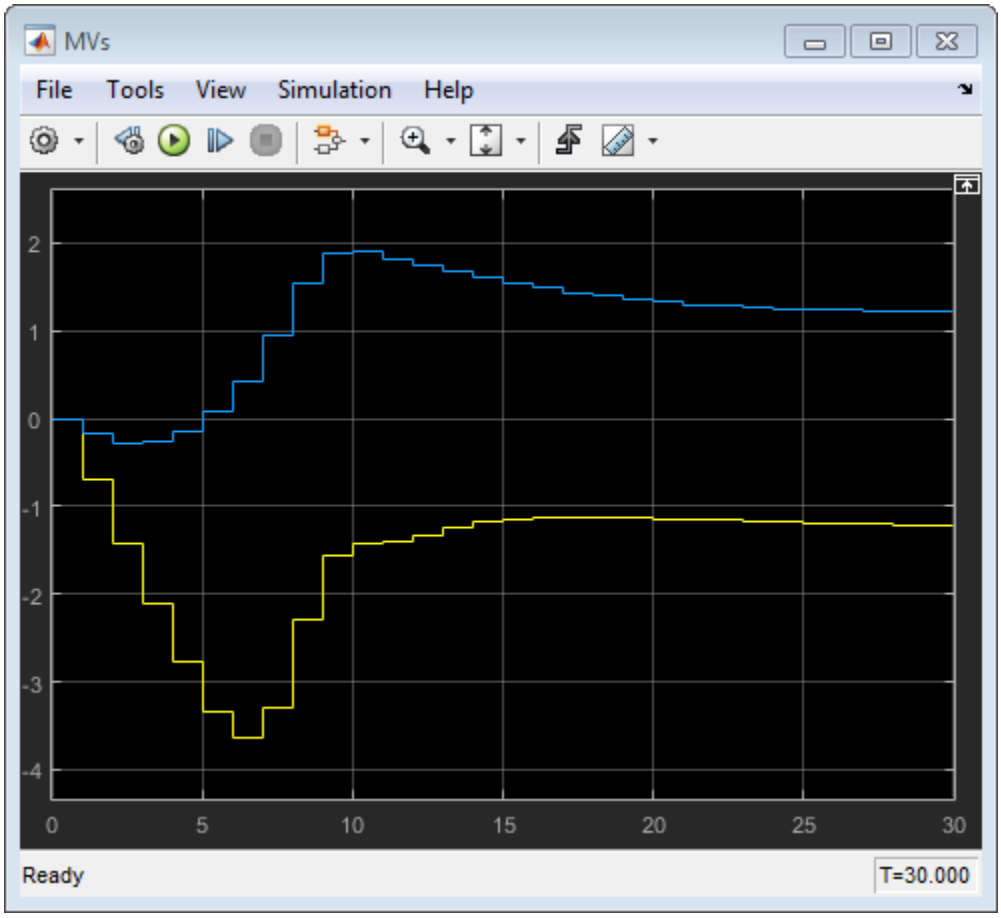
In the Simulink model window, double-click the Unmeasured Disturbance block.

In the Unmeasured Disturbance properties dialog box, specify a **Constant value** of `6.5`.

Click **OK**.

Simulate the model.

Outputs
File
Tools
View
Simulation
Help
4
3
2
1
0
-1
-2
-3
0
5
10
15
20
25
30
Ready
T=30.000



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 Designer | MPC Controller

More About

- “Design Controller Using MPC Designer”

Control of an Inverted Pendulum on a Cart

This example uses a model predictive controller (MPC) to control an inverted pendulum on a cart.

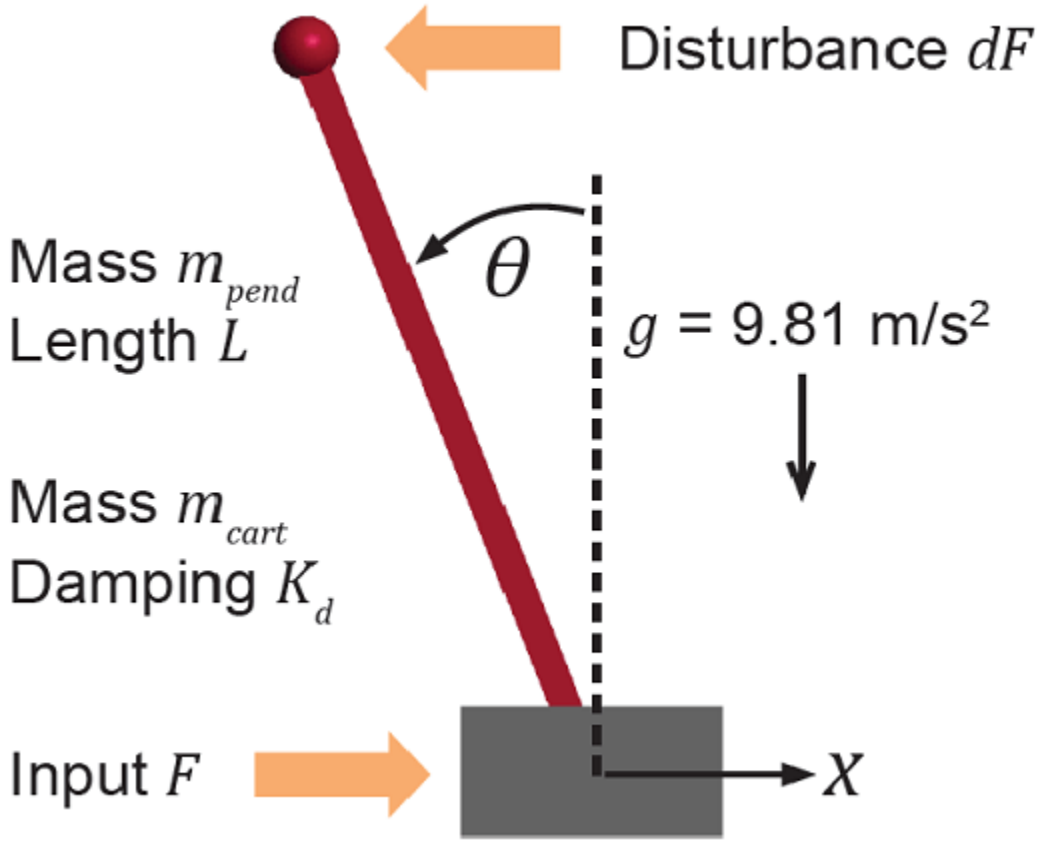
Product Requirement

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 is required to run this example.')
    return
end
```

Pendulum/Cart Assembly

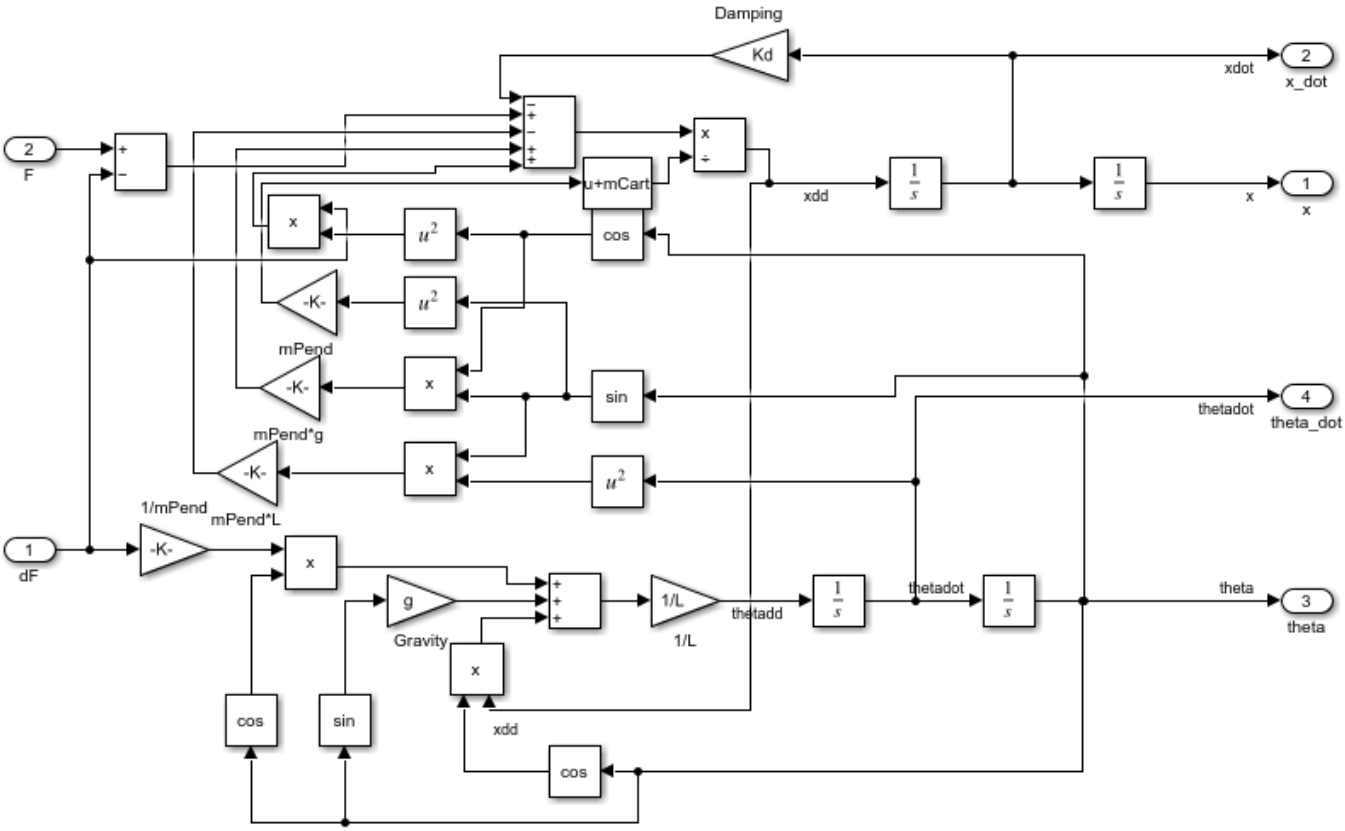
The plant for this example is the following cart/pendulum assembly, where *x* is the cart position and *theta* is the pendulum angle.



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')
```



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).

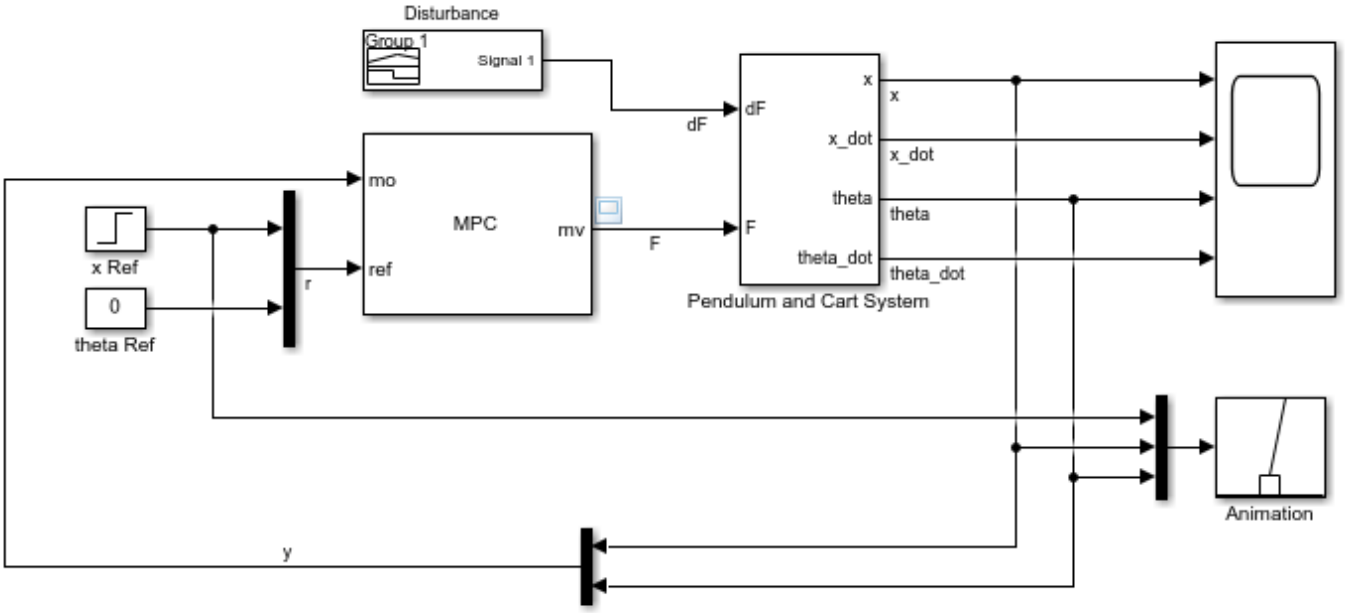
The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

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 *theta*.
- 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 *theta*, 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 *theta*. 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 0.0000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
```

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   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 *theta*.

```
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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Update the output disturbance model.

```
disturbance_model = getoutdist(mpcobj);
setoutdist(mpcobj,'model',disturbance_model*10);
```

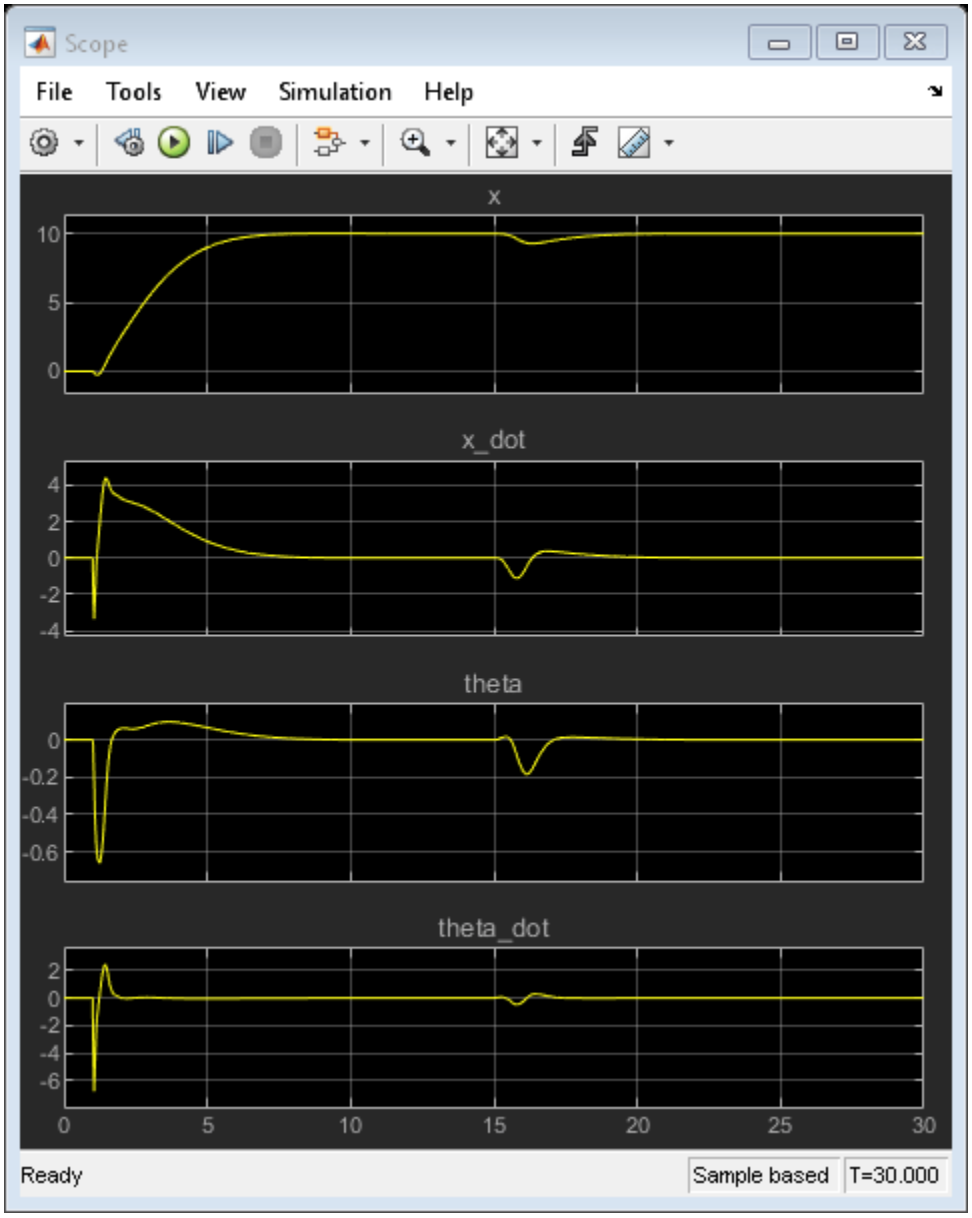
```
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

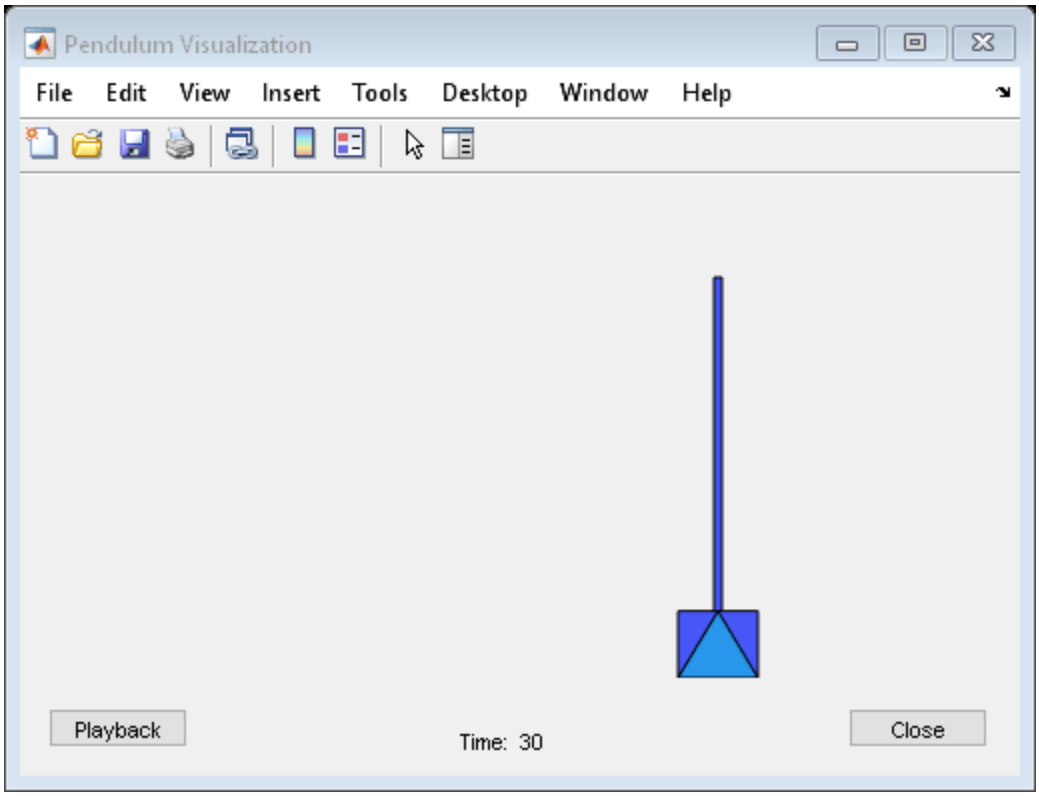
Closed-Loop Simulation

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 measured
```


Scope
File
Tools
View
Simulation
Help
x
10
5
0
x_dot
4
2
0
-2
-4
theta
0
-0.2
-0.4
-0.6
theta_dot
2
0
-2
-4
-6
0
5
10
15
20
25
30
Ready
Sample based
T=30.000



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” on page 8-58 shows how to use gain scheduling MPC to achieve the longer distances.

Close the Simulink model.

```
bdclose(mdlMPC)
```

See Also

More About

- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-33
- “Gain-Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 8-58

Thermo-Mechanical Pulping Process with Multiple Control Objectives

This example shows how to control a thermo-mechanical pulping (TMP) plant with a model predictive controller.

Plant Description

The following diagram shows a typical process arrangement for a two stage TMP operation. Two pressured refiners operate in sequence to produce a mechanical pulp suitable for making newsprint.

The primary objective of controlling the TMP plant is to regulate the energy applied to the pulp by the electric motors which drive each refiner to derive pulp with good physical properties without incurring excess energy costs.

The secondary control objective is to regulate the ratio of dry mass flow rate to overall mass flow rate (known as consistency) measured at the outlet of each refiner.

In practice, these objectives amount to regulating the primary and secondary refiner motor loads, and the primary and secondary refiner constancies, subject to the following output constraints:

(1) Maintain the power on each refiner below the maximum rated values.

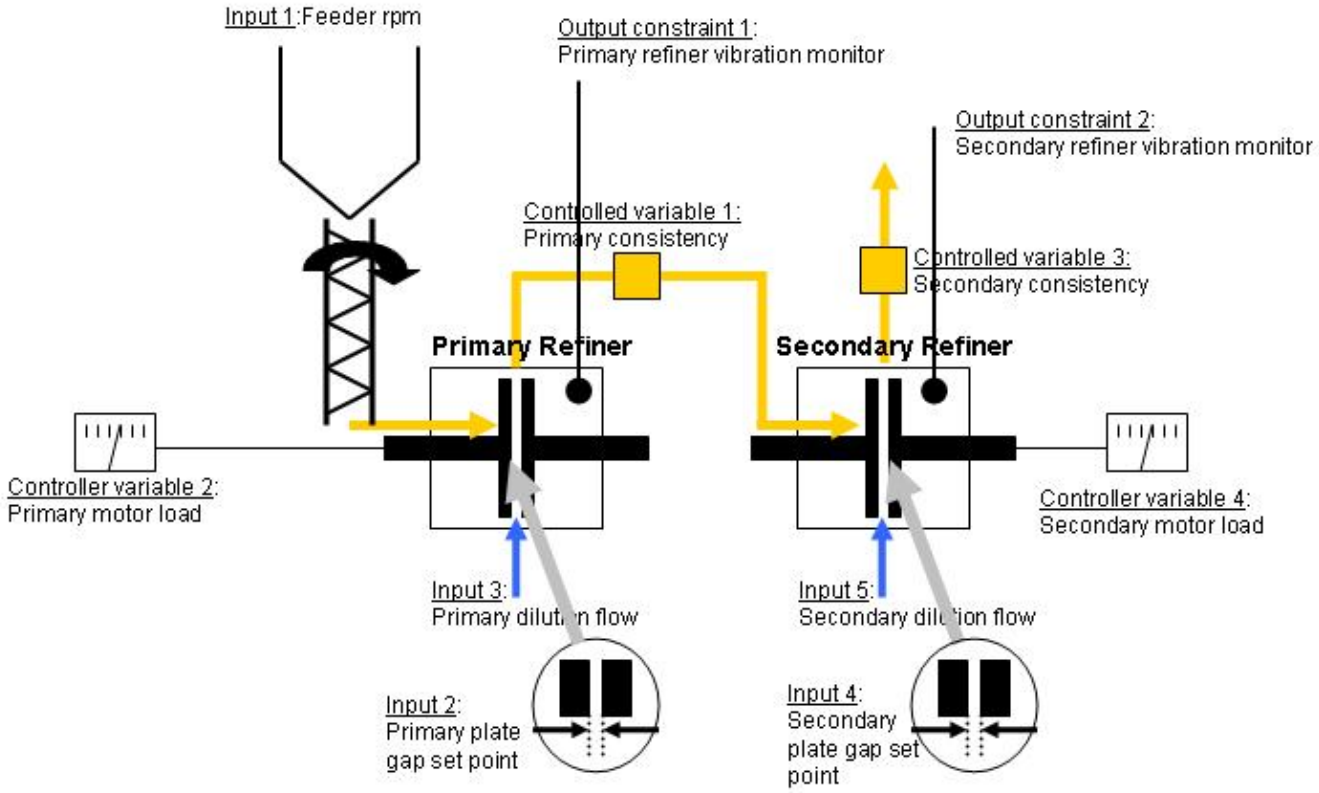
(2) Maintain the vibration level on the two refiners below a critical level to prevent refiner plate clash.

(3) Limit the measured consistency to prevent blow line plugging and fiber damage.

The manipulated variables for this plant include:

- Gap controller setpoints for regulating the distance between the refiner plates
- Dilution flow rates to the two refiners
- RPM of the screw feeder

Physical limits are also imposed on each of these inputs.



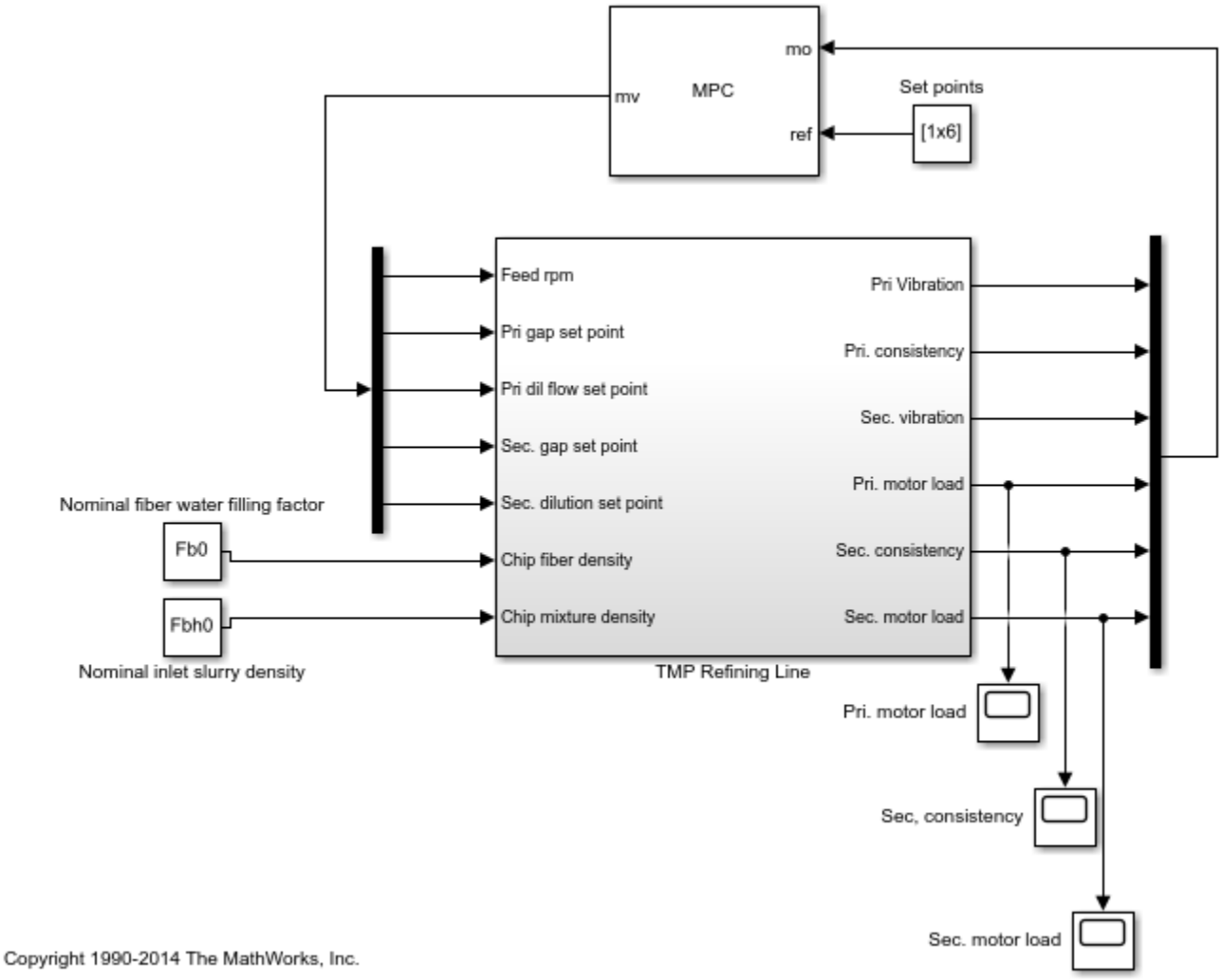
Modeling of the TMP Plant in Simulink®

To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

The following Simulink® model represents a TMP plant and an MPC Controller designed for the control objectives described above. The model is opened and plant data is initialized:

```
open_system('mpc_tmpdemo')
mpctmpinit;
```



The MPC controller is represented by an MPC object in the workspace. It is loaded from a previously saved design:

```
load mpc_tmpdemodata;
mpcobj
```

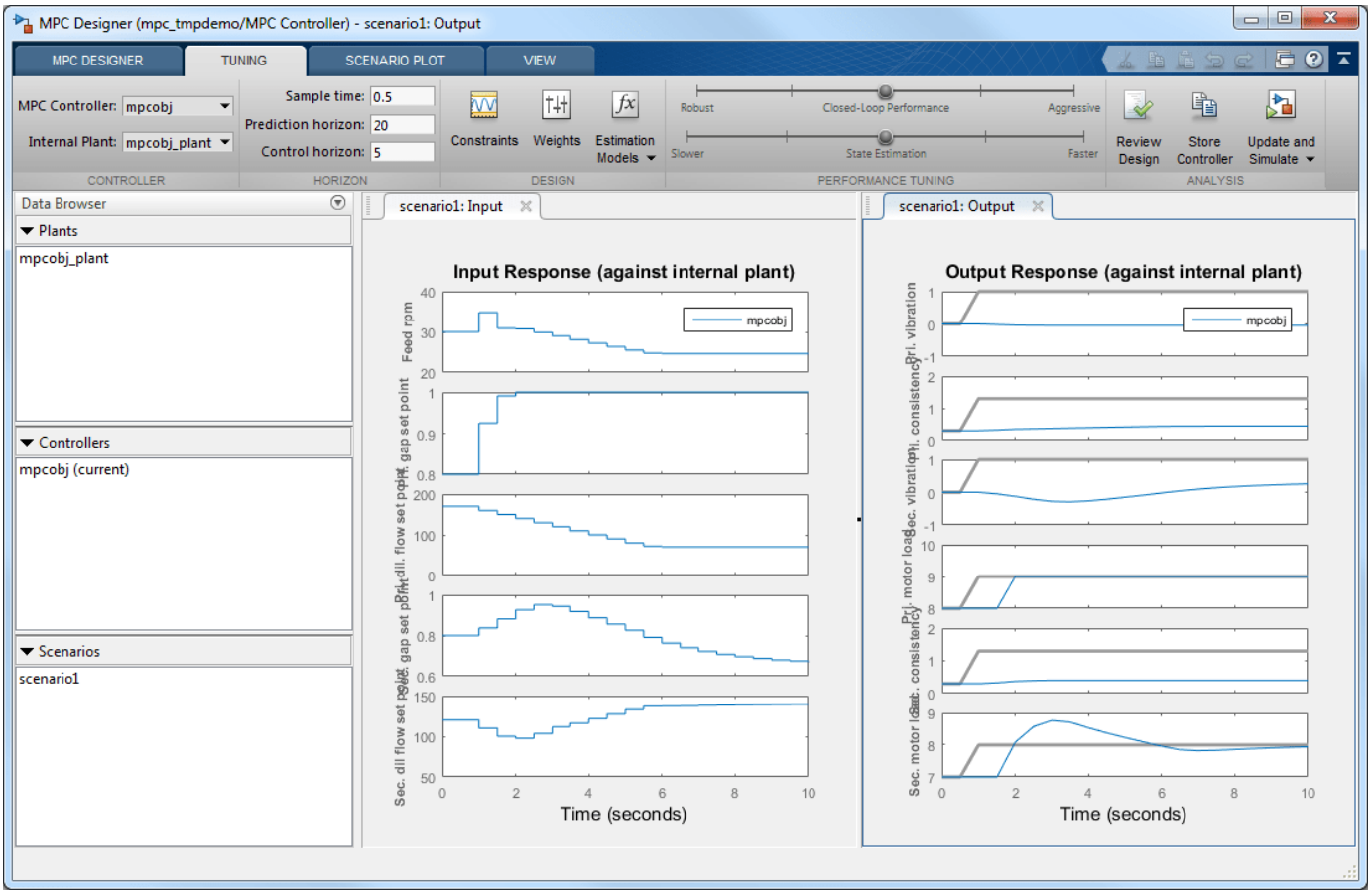
```
MPC object (created on 30-Mar-2004 17:20:31):
---------------------------------------------
Sampling time:      0.5 (seconds)
Prediction Horizon: 20
Control Horizon:    5

Plant Model:
                                    --------------
      5  manipulated variable(s)   -->|  7 states  |
                                     |             |-->  6 measured output(s)
      0  measured disturbance(s)   -->|  5 inputs  |
                                     |             |-->  0 unmeasured output(s)
      0  unmeasured disturbance(s) -->|  6 outputs |
```

```
                                   --------------
Disturbance and Noise Models:
        Output disturbance model: user specified (type "getoutdist(mpcobj)" for details)
         Measurement noise model: user specified (type "mpcobj.Model.Noise" for details)

Weights:
        ManipulatedVariables: [0 0 0 0 0]
    ManipulatedVariablesRate: [0.1000 10 0.1000 10 0.1000]
             OutputVariables: [0 10 0 1 10 1]
                         ECR: 1000000

State Estimation:  Default Kalman Filter (type "getEstimator(mpcobj)" for details)

Constraints:
                  0 <= Feed rpm <= 35,                -10 <= Feed rpm/rate <= Inf,       -Inf <=
          0 <= Pri. gap set point <= 1,       -10 <= Pri. gap set point/rate <= Inf, -Inf <= Pri
 70 <= Pri. dil. flow set point <= 250, -10 <= Pri. dil. flow set point/rate <= Inf,       -Inf <=
                       ...............                                 ...............
  70 <= Sec. dil flow set point <= 250,  -10 <= Sec. dil flow set point/rate <= Inf,     -Inf <=
```

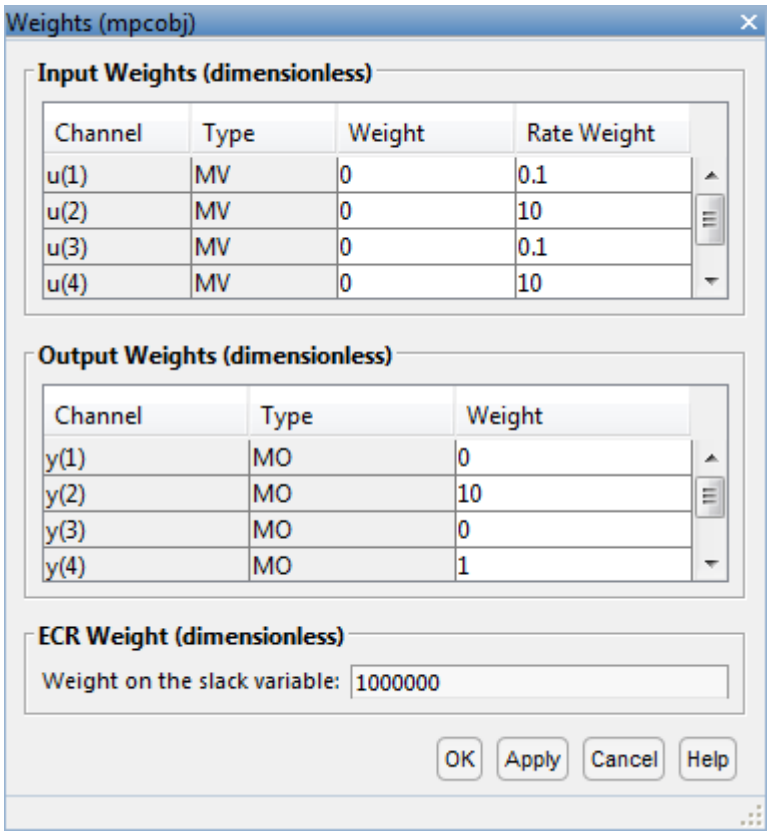
The controller was designed using the MPC Designer app.

Tuning the Controller Using the MPC Designer App

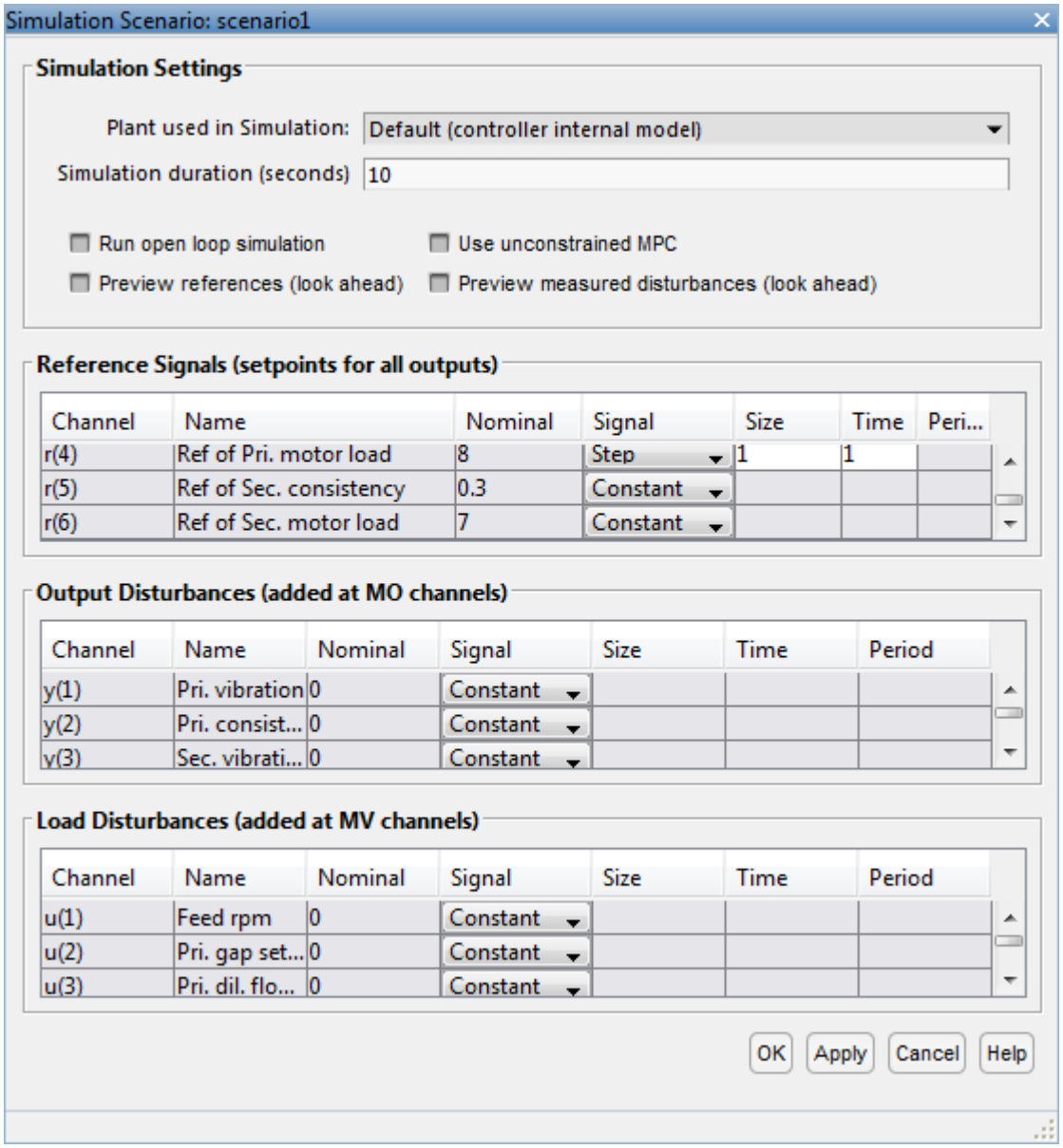
Click the "Design" button in the MPC Controller block dialog to launch the MPC Designer app.



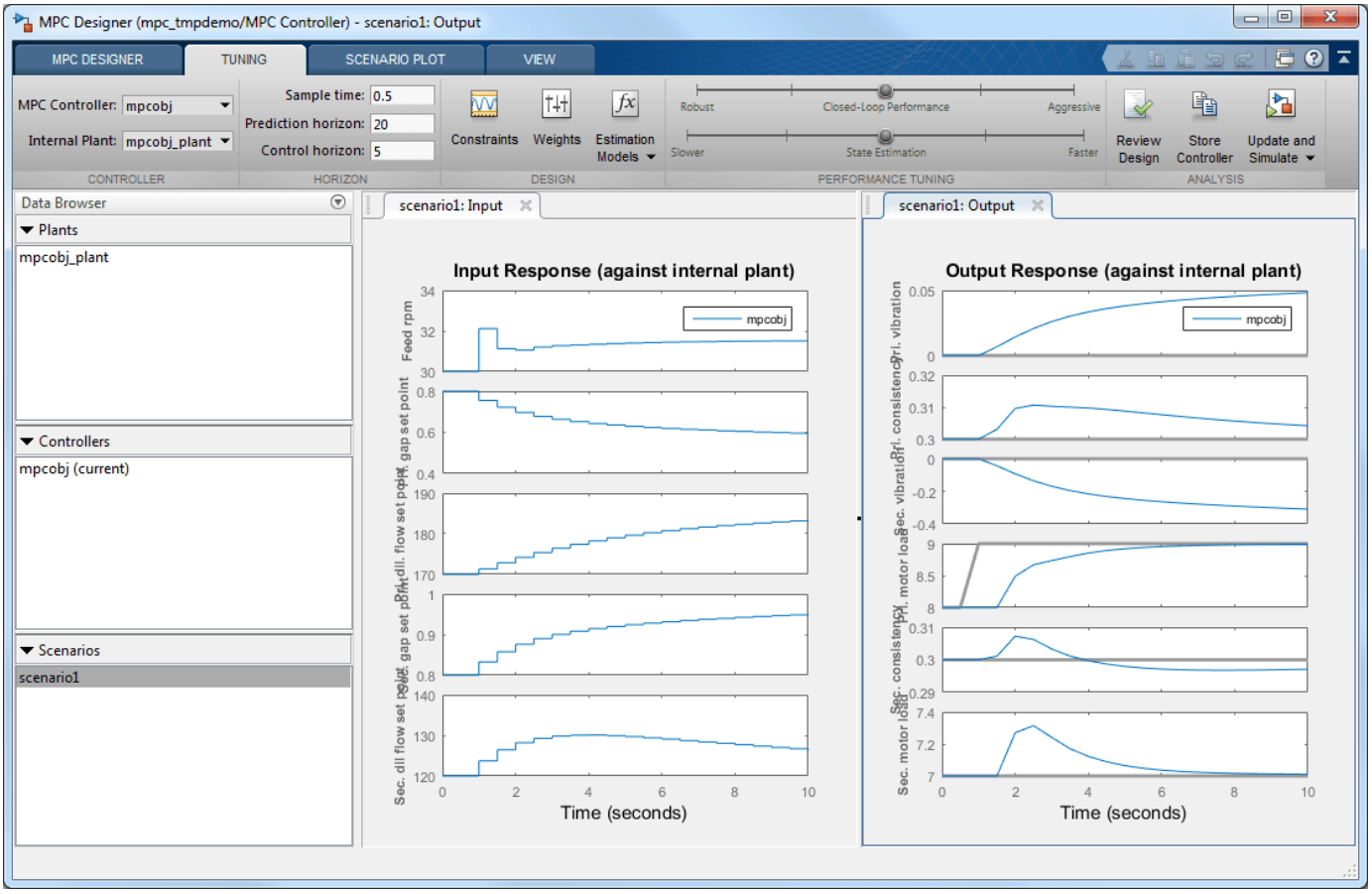
In the Tuning tab, click Weights to open the Weights dialog box. To put more emphasis on regulating primary and secondary refiner motor loads and constancies, specify the input and output weights as follows:



In the MPC Designer tab, click Edit Scenario to open the Simulation Scenario dialog box. To simulate a primary refiner motor load setpoint change from 8 to 9 MW without a model mismatch, specify the simulation scenario settings as follows:



The effect of design changes can be observed immediately in the response plots.

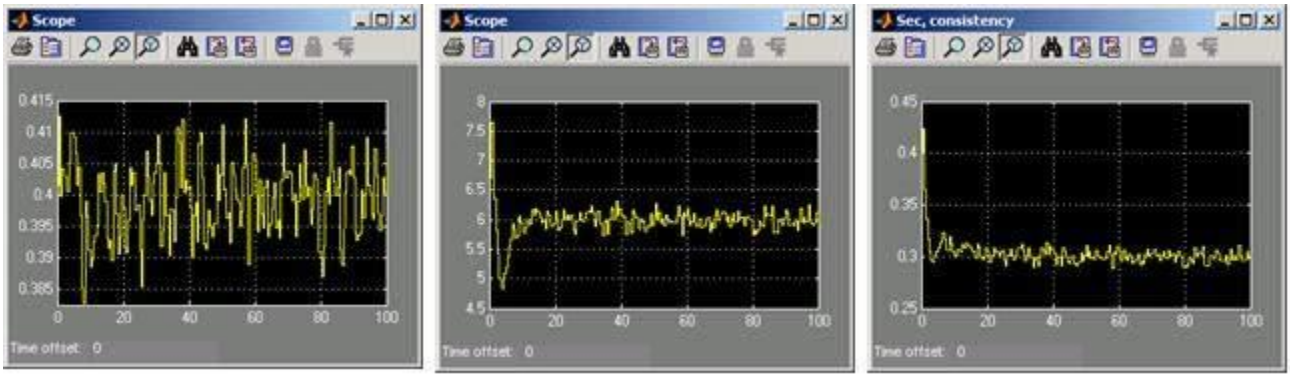


Simulating the Design in Simulink®

The controller can be tested on the non-linear plant by running the simulation in Simulink®. In the Tuning tab, in the Update and Simulate drop-down list, select Update Block and Run Simulation to export the current controller design to the MATLAB workspace and run the simulation in Simulink.

The output of the 3 scopes show the response to initial setpoints with:

- Primary consistency of 0.4
- Secondary motor load of 6 MW
- Secondary consistency of 0.3



```
bdclose('mpc_tmpdemo')
```

See Also

`mpc` | MPC Controller | **MPC Designer**

More About

- “Design MPC Controller in Simulink”

MPC Control of an Aircraft with Unstable Poles

This example shows how to use an MPC controller to control an unstable aircraft with saturating actuators.

For an example that controls the same plant using an explicit MPC controller, see “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-17.

Define Aircraft Model

The linearized open-loop model of the aircraft longitudinal dynamics has the following state-space matrices:

```
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];
```

Create the plant, and specify the initial states as zero.

```
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
```

Specify Controller 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 facilitate 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});
```

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',{1,200});
```

Specify Controller Tuning 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]);
```

Create MPC Controller

Create an MPC controller with the specified plant model, sample time, and horizons.

```
Ts = 0.05;          % Sample time
p = 10;             % Prediction horizon
m = 2;              % Control horizon
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Simulate Using Simulink®

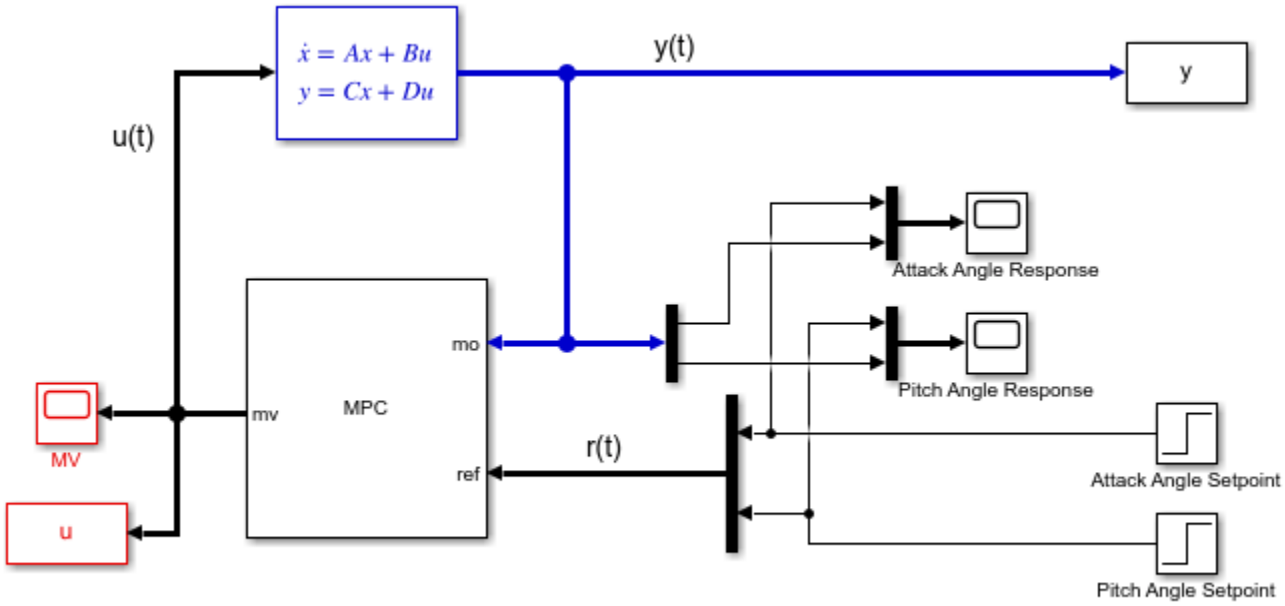
To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

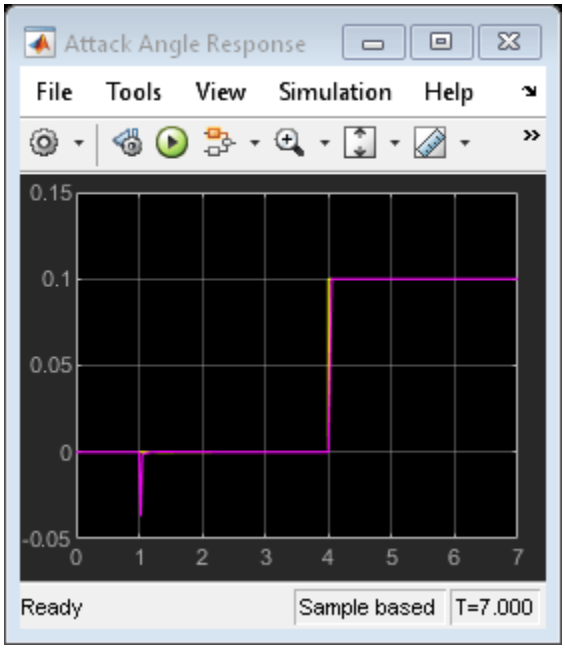
Simulate closed-loop control of the linear plant model in Simulink. To do so, for the MPC Controller block, set the **MPC Controller** property to `mpcobj`.

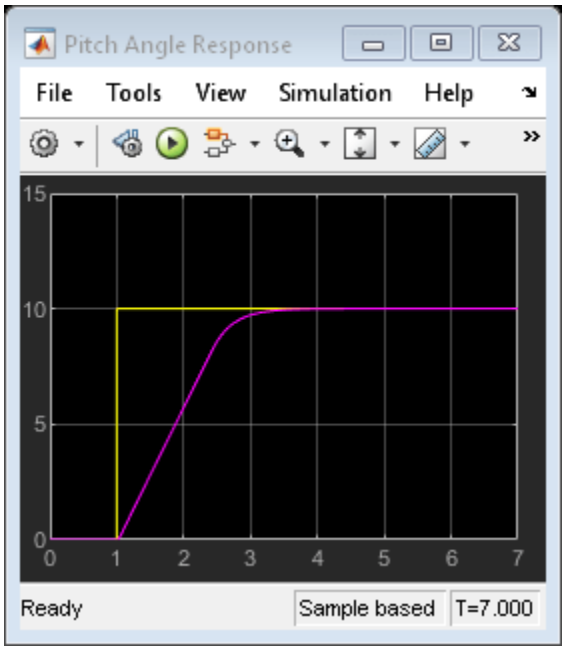
```
mdl = 'mpc_aircraft';
open_system(mdl)
sim(mdl)
```

```
-->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 measured
```



Copyright 1990-2014 The MathWorks, Inc.





The closed-loop response shows good setpoint tracking performance.

References

[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.

```
bdclose(mdl)
```

See Also

`mpc` | MPC Controller

More About

- “Design MPC Controller at the Command Line”

Model Predictive Control Basics

- “Controller State Estimation” on page 2-2
- “Optimization Problem” on page 2-7
- “QP Solvers” on page 2-17

Controller State Estimation

Controller State Variables

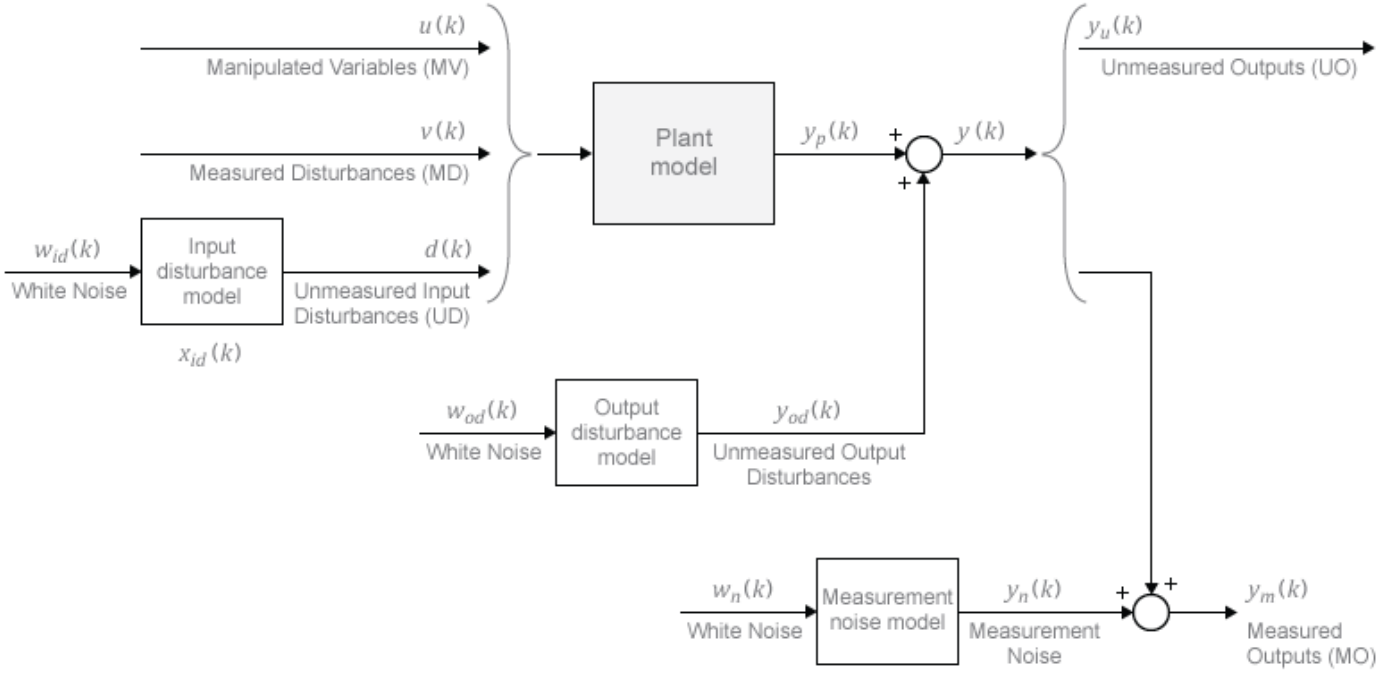
As the controller operates, it uses its current state, x_c, as the basis for predictions. By definition, the state vector is the following:

$$x_c^T(k) = \left[x_p^T(k)\ x_{id}^T(k)\ x_{od}^T(k)\ x_n^T(k)\right].$$

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 2-3 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 3-32 for details.

State Observer

Combination of the models shown in the diagram yields the state observer:

$$x_c(k+1) = Ax_c(k) + Bu_o(k)$$
$$y(k) = Cx_c(k) + Du_o(k)\,.$$

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 2-3).
- To predict how the controller’s proposed manipulated variable (MV) adjustments will affect future plant output values (see “Output Variable Prediction” on page 2-5).

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:

$$u_o^T(k) = \left[u^T(k)\ v^T(k)\ w_{id}^T(k)\ w_{od}^T(k)\ w_n^T(k)\right].$$

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:

$$A = \begin{bmatrix} A_p & B_{pd}C_{id} & 0 & 0 \\ 0 & A_{id} & 0 & 0 \\ 0 & 0 & A_{od} & 0 \\ 0 & 0 & 0 & A_n \end{bmatrix},\quad B = \begin{bmatrix} B_{pu} & B_{pv} & B_{pd}D_{id} & 0 & 0 \\ 0 & 0 & B_{id} & 0 & 0 \\ 0 & 0 & 0 & B_{od} & 0 \\ 0 & 0 & 0 & 0 & B_n \end{bmatrix},$$
$$C = \begin{bmatrix} C_p & D_{pd}C_{id} & C_{od} \begin{bmatrix} C_n \\ 0 \end{bmatrix} \end{bmatrix},\ D = \begin{bmatrix} 0 & D_{pv} & D_{pd}D_{id} & D_{od} \begin{bmatrix} D_n \\ 0 \end{bmatrix} \end{bmatrix}.$$

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.

At the beginning of the kth 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.

$$x_c^{rev}(k\,|k-1) = x_c(k\,|k-1) + B_u\left[u^{act}(k-1) - u^{opt}(k-1)\right]$$

3 Compute the innovation.

$$e(k) = y_m(k) - \left[C_m x_c^{rev}(k\,|k-1) + D_{mv}v(k)\right]$$

4 Update the controller state estimate to account for the latest measurements.

$$x_c(k\,|k) = x_c^{rev}(k\,|k-1) + Me(k)$$

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:

$$x_c(k+1\,|k) = Ax_c^{rev}(k\,|k-1) + B_u u^{opt}(k) + B_v v(k) + Le(k)$$

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:

$$x_c(k+1) = Ax_c(k) + Bw(k)$$
$$y_m(k) = C_m x_c(k) + D_m w(k)\,.$$

Here,

$$w^T(k) = \left[w_u^T(k)\ w_v^T(k)\ w_{id}^T(k)\ w_{od}^T(k)\ w_n^T(k) \right].$$

Inputs to the `kalman` command are the state observer parameters A, C_m, and the following covariance matrices:

$$\begin{aligned} Q &= E\left\{ Bww^TB^T \right\} = BB^T \\ R &= E\left\{ D_m ww^T D_m^T \right\} = D_m D_m^T \\ N &= \quad E\left\{ Bww^T D_m^T \right\} = BD_m^T . \end{aligned}$$

Here, $E\{...\}$ 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 2-3).

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 2-3)
- $v(k)$ — 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_{xn} 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 “Specify Scale Factors” on page 1-29. 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 2-11.

See Also

`kalman`

More About

- “MPC Prediction Models”
- “Optimization Problem” on page 2-7
- “Custom State Estimation” on page 3-32
- “Implement Custom State Estimator Equivalent to Built-In Kalman Filter” on page 3-37

Optimization Problem

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 minimize 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:

$$J(z_k) = J_y(z_k) + J_u(z_k) + J_{\Delta u}(z_k) + J_\varepsilon(z_k)\,.$$

Here, z_k is the QP decision. As described below, each term includes weights that help you balance competing objectives. While the MPC controller provides default weights, 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. An MPC controller uses the following scalar performance measure for output reference tracking:

$$J_y(z_k) = \sum_{j=1}^{n_y} \sum_{i=1}^{p} \left\{ \frac{w_{i,j}^y}{s_j^y} \left[r_j(k+i\,|k) - y_j(k+i\,|k) \right] \right\}^2 .$$

Here,

- k — Current control interval.
- p — Prediction horizon (number of intervals).
- n_y — Number of plant output variables.
- z_k — QP decision, given by:

  $$z_k^T = \left[u(k\,|k)^T \;\; u(k+1\,|k)^T \;\; \cdots \;\; u(k+p-1\,|k)^T \;\; \varepsilon_k \right].$$

- $y_j(k+i|k)$ — Predicted value of jth plant output at ith prediction horizon step, in engineering units.
- $r_j(k+i|k)$ — Reference value for jth plant output at ith prediction horizon step, in engineering units.

- s_j^y — Scale factor for jth plant output, in engineering units.
- $w_{i,j}^y$ — Tuning weight for jth plant output at ith prediction horizon step (dimensionless).

The values n_y, p, s_j^y, and $w_{i,j}^y$ are constant controller specifications. The controller receives reference values, $r_j(k+i|k)$, for the entire prediction horizon. The controller uses the state observer to predict the plant outputs, $y_j(k+i|k)$, which depend on manipulated variable adjustments (z_k), measured disturbances (MD), and state estimates. At interval k, the controller state estimates and MD values are available. Therefore, J_y is a function of z_k only.

Manipulated Variable Tracking

In some applications, such as when there are more manipulated variables than plant outputs, the controller must keep selected manipulated variables (MVs) at or near specified target values. An MPC controller uses the following scalar performance measure for manipulated variable tracking:

$$J_u(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^u}{s_j^u} \left[u_j(k+i\,|k) - u_{j,target}(k+i\,|k) \right] \right\}^2 .$$

Here,

- k — Current control interval.
- p — Prediction horizon (number of intervals).
- n_u — Number of manipulated variables.
- z_k — QP decision, given by:

  $$z_k^T = \left[u(k\,|k)^T \;\; u(k+1\,|k)^T \;\; \cdots \;\; u(k+p-1\,|k)^T \;\; \varepsilon_k \right].$$

- $u_{j,target}(k+i|k)$ — Target value for jth MV at ith prediction horizon step, in engineering units.
- s_j^u — Scale factor for jth MV, in engineering units.
- $w_{i,j}^u$ — Tuning weight for jth MV at ith prediction horizon step (dimensionless).

The values n_u, p, s_j^u, and $w_{i,j}^u$ are constant controller specifications. 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). An MPC constant uses the following scalar performance measure for manipulated variable move suppression:

$$J_{\Delta u}(z_k) = \sum_{j=1}^{n_u} \sum_{i=0}^{p-1} \left\{ \frac{w_{i,j}^{\Delta u}}{s_j^u} \left[u_j(k+i\,|k) - u_j(k+i-1\,|k) \right] \right\}^2 .$$

Here,

- k — Current control interval.

- p — Prediction horizon (number of intervals).
- n_u — Number of manipulated variables.
- z_k — QP decision, given by:

  $$z_k^T = \left[u(k|k)^T \; u(k+1|k)^T \; \cdots \; u(k+p-1|k)^T \; \varepsilon_k\right].$$

- s_j^u — Scale factor for jth MV, in engineering units.
- $w_{i,j}^{\Delta u}$ — Tuning weight for jth MV movement at ith prediction horizon step (dimensionless).

The values n_u, p, s_j^u, and $w_{i,j}^{\Delta u}$ are constant controller specifications. $u(k–1|k) = u(k–1)$, 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. An MPC controller employs a dimensionless, nonnegative slack variable, ε_k, which quantifies the worst-case constraint violation. (See “Constraints” on page 2-10) The corresponding performance measure is:

$$J_\varepsilon(z_k) = \rho_\varepsilon \varepsilon_k^2 .$$

Here,

- z_k — QP decision, given by:

  $$z_k^T = \left[u(k|k)^T \; u(k+1|k)^T \; \cdots \; u(k+p-1|k)^T \; \varepsilon_k\right].$$

- 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:

$$J(z_k) = \sum_{i=0}^{p-1} \left\{\left[e_y^T(k+i) Q e_y(k+i)\right] + \left[e_u^T(k+i) R_u e_u(k+i)\right] + \left[\Delta u^T(k+i) R_{\Delta u} \Delta u(k+i)\right]\right\} + \rho_\varepsilon \varepsilon_k^2 .$$

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:

$$e_y(i+k) = S_y^{-1}[r(k+i+1|k) - y(k+i+1|k)]$$
$$e_u(i+k) = S_u^{-1}\left[u_{target}(k+i|k) - u(k+i\big|k)\right]$$
$$\Delta u(k+i) = S_u^{-1}[u(k+i|k) - u(k+i-1|k)] .$$

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 ith prediction horizon step, in engineering units.
- $y(k+1|k)$ — n_y plant outputs at the ith prediction horizon step, in engineering units.
- z_k — QP decision, given by:

$$z_k^T = \left[u(k\,|\,k)^T \; u(k+1\,|\,k)^T \; \cdots \; u(k+p-1\,|\,k)^T \; \varepsilon_k\right].$$

- $u_{target}(k+i|k)$ — 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 $w^y_{i,j}$, $w^u_{i,j}$, and $w^{\Delta u}_{i,j}$ 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.

$$\frac{y_{j,min}(i)}{s^y_j} - \varepsilon_k V^y_{j,min}(i) \le \frac{y_j(k+i|k)}{s^y_j} \le \frac{y_{j,max}(i)}{s^y_j} + \varepsilon_k V^y_{j,max}(i), \qquad i = 1{:}p, \qquad j = 1{:}n_y$$

$$\frac{u_{j,min}(i)}{s^u_j} - \varepsilon_k V^u_{j,min}(i) \le \frac{u_j(k+i-1|k)}{s^u_j} \le \frac{u_{j,max}(i)}{s^u_j} + \varepsilon_k V^u_{j,max}(i), \qquad i = 1{:}p, \qquad j = 1{:}n_u$$

$$\frac{\Delta u_{j,min}(i)}{s^u_j} - \varepsilon_k V^{\Delta u}_{j,min}(i) \le \frac{\Delta u_j(k+i-1|k)}{s^u_j} \le \frac{\Delta u_{j,max}(i)}{s^u_j} + \varepsilon_k V^{\Delta u}_{j,max}(i), \qquad i = 1{:}p, \qquad j = 1{:}n_u\,.$$

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 1-7). Also,

- ε_k — Scalar QP slack variable (dimensionless) used for constraint softening.
- s^y_j — Scale factor for jth plant output, in engineering units.
- s^u_j — Scale factor for jth MV, in engineering units.
- $y_{j,\min}(i)$, $y_{j,\max}(i)$ — lower and upper bounds for jth plant output at ith prediction horizon step, in engineering units.
- $u_{j,\min}(i)$, $u_{j,\max}(i)$ — lower and upper bounds for jth MV at ith prediction horizon step, in engineering units.

- $\Delta u_{j,\min}(i)$, $\Delta u_{j,\max}(i)$ — lower and upper bounds for jth MV increment at ith 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 2-7.

Prediction

Assume that the disturbance models described in “Input Disturbance Model” are unit gains; that is, $d(k) = n_d(k)$ is white Gaussian noise. You can denote this problem as

$$x \leftarrow \begin{bmatrix} x \\ x_d \end{bmatrix},\ A \leftarrow \begin{bmatrix} A & B_d\bar{C} \\ 0 & \bar{A} \end{bmatrix},\ B_u \leftarrow \begin{bmatrix} B_u \\ 0 \end{bmatrix},\ B_v \leftarrow \begin{bmatrix} Bv \\ 0 \end{bmatrix},\ B_d \leftarrow \begin{bmatrix} B_d\bar{D} \\ \bar{B} \end{bmatrix},\ C \leftarrow \begin{bmatrix} C & D_d\bar{C} \end{bmatrix}$$

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

$$y(i\,|\,0) = C\left[A^i x(0) + \sum_{h=0}^{i-1} A^{i-1}\left(B_u\left(u(-1) + \sum_{j=0}^{h}\Delta u(j)\right) + B_v v(h)\right)\right] + D_v v(i)$$

This equation gives the solution

$$\begin{bmatrix} y(1) \\ \cdots \\ y(p) \end{bmatrix} = S_x x(0) + S_{u1} u(-1) + S_u \begin{bmatrix} \Delta u(0) \\ \cdots \\ \Delta u(p-1) \end{bmatrix} + H_v \begin{bmatrix} v(0) \\ \cdots \\ v(p) \end{bmatrix}$$

where

$$S_x = \begin{bmatrix} CA \\ CA^2 \\ \cdots \\ CA^p \end{bmatrix} \in \Re^{pn_y \times n_x}, S_{u1} = \begin{bmatrix} CB_u \\ CB_u + CAB_u \\ \cdots \\ \sum_{h=0}^{p-1} CA^h B_u \end{bmatrix} \in \Re^{pn_y \times n_u}$$

$$S_u = \begin{bmatrix} CB_u & 0 & \cdots & 0 \\ CB_u + CAB_u & CB_u & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{h=0}^{p-1} CA^h B_u & \sum_{h=0}^{p-2} CA^h B_u & \cdots & CB_u \end{bmatrix} \in \Re^{pn_y \times pn_u}$$

$$H_v = \begin{bmatrix} CB_v & D_v & 0 & \cdots & 0 \\ CAB_v & CB_v & D_v & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ CA^{p-1}B_v & CA^{p-2}B_v & CA^{p-3}B_v & \cdots & D_v \end{bmatrix} \in \Re^{pn_y \times (p+1)n_v}.$$

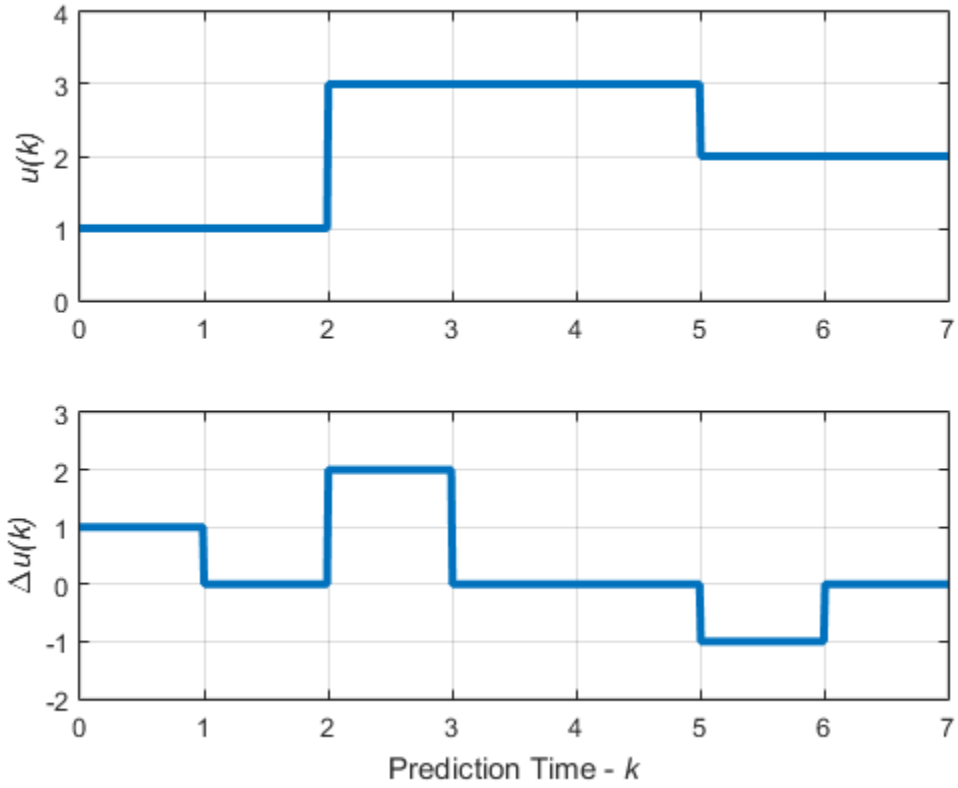
Optimization Variables

Let m be the number of free control moves, and let $z = [z_0; \ldots; z_{m-1}]$. Then,

$$\begin{bmatrix} \Delta u(0) \\ \cdots \\ \Delta u(p-1) \end{bmatrix} = J_M \begin{bmatrix} z_0 \\ \cdots \\ z_{m-1} \end{bmatrix}$$

where J_M depends on the choice of blocking moves. Together with the slack variable ε, vectors z_0, ..., z_{m-1} constitute the free optimization variables of the optimization problem. In the case of systems with a single manipulated variable, z_0, ..., 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 `moves=[2 3 2]`, or equivalently, $u(0)=u(1)$, $u(2)=u(3)=u(4)$, $u(5)=u(6)$, $\Delta u(0)=z0$, $\Delta u(2)=z1$, $\Delta u(5)=z2$, $\Delta u(1)=\Delta u(3)=\Delta u(4)=\Delta u(6)=0$.

Then, the corresponding matrix J_M is

$$J_M = \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{bmatrix}$$

For more information on manipulated variable blocking, see “Manipulated Variable Blocking” on page 3-44.

Cost Function

Standard Form

The function to be optimized is

$$J(z,\varepsilon) = \left(\begin{bmatrix} u(0) \\ \cdots \\ u(p-1) \end{bmatrix} - \begin{bmatrix} u_{target}(0) \\ \cdots \\ u_{target}(p-1) \end{bmatrix}\right)^T W_u^2 \left(\begin{bmatrix} u(0) \\ \cdots \\ u(p-1) \end{bmatrix} - \begin{bmatrix} u_{target}(0) \\ \cdots \\ u_{target}(p-1) \end{bmatrix}\right) + \begin{bmatrix} \Delta u(0) \\ \cdots \\ \Delta u(p-1) \end{bmatrix}^T W_{\Delta u}^2 \begin{bmatrix} \Delta u(0) \\ \cdots \\ \Delta u(p-1) \end{bmatrix}$$
$$+ \left(\begin{bmatrix} y(1) \\ \cdots \\ y(p) \end{bmatrix} - \begin{bmatrix} r(1) \\ \cdots \\ r(p) \end{bmatrix}\right)^T W_y^2 \left(\begin{bmatrix} y(1) \\ \cdots \\ y(p) \end{bmatrix} - \begin{bmatrix} r(1) \\ \cdots \\ r(p) \end{bmatrix}\right) + \rho_\varepsilon \varepsilon^2$$

where

$$\begin{aligned} W_u &= \mathrm{diag}\left(w_{0,1}^u, w_{0,2}^u, \ldots, w_{0,n_u}^u, \ldots, w_{p-1,1}^u, w_{p-1,2}^u, \ldots, w_{p-1,n_u}^u\right) \\ W_{\Delta u} &= \mathrm{diag}\left(w_{0,1}^{\Delta u}, w_{0,2}^{\Delta u}, \ldots, w_{0,n_u}^{\Delta u}, \ldots, w_{p-1,1}^{\Delta u}, w_{p-1,2}^{\Delta u}, \ldots, w_{p-1,n_u}^{\Delta u}\right) \\ W_y &= \mathrm{diag}\left(w_{1,1}^y, w_{1,2}^y, \ldots, w_{1,n_y}^y, \ldots, w_{p,1}^y, w_{p,2}^y, \ldots, w_{p,n_y}^y\right) \end{aligned} \tag{2-1}$$

Finally, after substituting $u(k)$, $\Delta u(k)$, $y(k)$, $J(z)$ can be rewritten as

$$J(z,\varepsilon) = \rho_\varepsilon \varepsilon^2 + z^T K_{\Delta u} z + 2\left(\begin{bmatrix} r(1) \\ \cdots \\ r(p) \end{bmatrix}^T K_r + \begin{bmatrix} v(0) \\ \cdots \\ v(p) \end{bmatrix}^T K_v + u(-1)^T K_u + \begin{bmatrix} u_{target}(0) \\ \cdots \\ u_{target}(p-1) \end{bmatrix}^T K_{ut} + x(0)^T K_x\right) z \tag{2-2}$$
$$+ c_y^T W_y c_y + c_u^T W_u c_u$$

where

$$c_y = S_x x(0) + S_{u1} u(-1) + H_v \begin{bmatrix} v(0) \\ \cdots \\ v(p) \end{bmatrix} - \begin{bmatrix} r(1) \\ \cdots \\ r(p) \end{bmatrix}$$
$$c_u = \begin{bmatrix} I_1 \\ \cdots \\ I_p \end{bmatrix} u(-1) - \begin{bmatrix} u_{target}(0) \\ \cdots \\ u_{target}(p-1) \end{bmatrix}$$

Here, $I_1 = \ldots = I_p$ are identity matrices of size n_u.

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*sqrt(eps)` on each diagonal term. You can use this solution only when all input rates are unpenalized ($W^{\Delta u}=0$) (see the `Weights` property of the `mpc` object).

Alternative Cost Function

If you are using the alternative cost function shown in “Alternative Cost Function” on page 2-9, then “Equation 2-1” is replaced by the following:

$$\begin{aligned} W_u &= \mathrm{blkdiag}(R_u, \ldots, R_u) \\ W_{\Delta u} &= \mathrm{blkdiag}(R_{\Delta u}, \ldots, R_{\Delta u}) \\ W_y &= \mathrm{blkdiag}(Q, \ldots, Q) \end{aligned} \tag{2-3}$$

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. For more information, see the `Weights` property of the `mpc` object.

Constraints

Next, consider the limits on inputs, input increments, and outputs along with the constraint $\varepsilon \geq 0$.

$$\begin{bmatrix} y_{\min}(1) - \varepsilon V^{y}_{\min}(1) \\ \cdots \\ y_{\min}(p) - \varepsilon V^{y}_{\min}(p) \\ u_{\min}(0) - \varepsilon V^{u}_{\min}(0) \\ \cdots \\ u_{\min}(p-1) - \varepsilon V^{u}_{\min}(p-1) \\ \Delta u_{\min}(0) - \varepsilon V^{\Delta u}_{\min}(0) \\ \cdots \\ \Delta u_{\min}(p-1) - \varepsilon V^{\Delta u}_{\min}(p-1) \end{bmatrix} \leq \begin{bmatrix} y(1) \\ \cdots \\ y(p) \\ u(0) \\ \cdots \\ u(p-1) \\ \Delta u(0) \\ \cdots \\ \Delta u(p-1) \end{bmatrix} \leq \begin{bmatrix} y_{\max}(1) + \varepsilon V^{y}_{\max}(1) \\ \cdots \\ y_{\max}(p) + \varepsilon V^{y}_{\max}(p) \\ u_{\max}(0) + \varepsilon V^{u}_{\max}(0) \\ \cdots \\ u_{\max}(p-1) + \varepsilon V^{u}_{\max}(p-1) \\ \Delta u_{\max}(0) + \varepsilon V^{\Delta u}_{\max}(0) \\ \cdots \\ \Delta u_{\max}(p-1) + \varepsilon V^{\Delta u}_{\max}(p-1) \end{bmatrix}$$

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

$$M_z z + M_\varepsilon \varepsilon \leq M_{\lim} + M_v \begin{bmatrix} v(0) \\ \cdots \\ v(p) \end{bmatrix} + M_u u(-1) + M_x x(0) \tag{2-4}$$

In this case, matrices M_z, M_ε, $M_{\lim}$, M_v, M_u, and M_x are obtained from the upper and lower bounds and ECR values.

Unconstrained Model Predictive Control

The optimal solution is computed analytically

$$z^* = -K_{\Delta u}^{-1}\left(\begin{bmatrix} r(1) \\ \cdots \\ r(p) \end{bmatrix}^T K_r + \begin{bmatrix} v(0) \\ \cdots \\ v(p) \end{bmatrix} K_v + u(-1)^T K_u + \begin{bmatrix} u_{target}(0) \\ \cdots \\ u_{target}(p-1) \end{bmatrix}^T K_{ut} + x(0)^T K_x \right)^T$$

and the model predictive controller sets $\Delta u(k) = z^*_0$, $u(k) = u(k-1) + \Delta u(k)$.

See Also

More About

- “Adjust Disturbance and Noise Models” on page 3-25
- “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42
- “Terminal Weights and Constraints” on page 3-18

QP Solvers

The model predictive controller QP solver converts a linear MPC optimization problem to the general form QP problem

$$\underset{x}{Min}(\frac{1}{2}x^{\top}Hx + f^{\top}x)$$

subject to the linear inequality constraints

$$Ax \leq b$$

where

- x is the solution vector.
- H is the Hessian matrix. This matrix is constant when your prediction model and tuning weights do not change at run time.
- A is a matrix of linear constraint coefficients. This matrix is constant when your prediction model does not change at run time.
- b and f are vectors.

At the beginning of each control interval, the controller computes H, f, A, and b. If H or A is constant, the controller retrieves their precomputed values.

Built-In QP Solvers

Model Predictive Control Toolbox software supports two built-in algorithms for solving the QP problem. Both solvers require the Hessian matrix to be positive definite.

- Active-set solver — This solver can provide fast and robust performance for small-scale and medium-scale optimization problems in both single and double precision. The active-set solver uses the KWIK algorithm from [1]. To use the active-set solver, set the `Optimizer.Algorithm` property of your MPC controller to `'active-set'`. To configure the algorithm settings, use the `Optimizer.ActiveSetOptions` property of your controller.
- Interior-point solver — This solver can provide superior performance for large-scale optimization problems, such as MPC applications that enforce constraints over large prediction and control horizons. This interior-point solver uses a primal-dual algorithm with a Mehrotra predictor-corrector. To use the interior-point solver, set the `Optimizer.Algorithm` property of your MPC controller to `'interior-point'`. To configure the algorithm settings, use the `Optimizer.InteriorPointOptions` property of your controller.

Solver Configuration

When selecting and configuring the QP solver for your application, consider the following:

- The size and configuration of the MPC problem affects the performance of the built-in QP solvers. To determine which solver is best for your application, consider simulating your controller across multiple simulation scenarios using both QP solvers.
- The interior-point solver is more sensitive to solver parameters than the active-set solver. Therefore, it can require more adjustment to find an optimal balance between performance and robustness.

- The active-set solver also uses a nonadjustable tolerance when testing for an optimal solution. You can adjust the optimality tolerances for the interior-point solver.
- One or more linear constraints can be violated slightly due to numerical round-off errors. Such violations are normal and do not generate warning messages. To adjust the tolerance for acceptable constraint violations, use the `ConstraintTolerance` setting for either the active-set or interior-point solver.
- The search for a QP solution is an iterative process. For either solver, you can specify the maximum number of iterations using the corresponding `MaxIterations` setting. If the number of iterations reaches the maximum, the algorithm terminates.
- The default maximum number of iterations for the active-set solver is $4(n_c + n_v)$, where n_c and n_v are the number of constraints and optimization variables across the prediction horizon, respectively. For some controller configurations, this value can be very large, which can make the QP solver appear to stop responding. This value has a lower bound of `120`.
- The default maximum number of iterations for the interior-point solver is `50`.
- If your MPC problem includes hard constraints after conversion to a QP problem, the QP inequality constraints can be infeasible (impossible to satisfy). If the QP solver detects infeasibility, it terminates immediately.

When the solver detects an infeasible problem or reaches the maximum number of iterations without finding an optimal solution, the controller retains the last successful control output. For more information, see `mpcmove`. You can detect an abnormal outcome and override the default behavior as you see fit.

In the first control step, the QP solvers use a cold start, in which the initial guess is the unconstrained solution described in “Unconstrained Model Predictive Control” on page 2-15. If x satisfies the constraints, it is the optimal QP solution and the algorithm terminates. Otherwise, at least one of the linear inequality constraints must be satisfied as an equality, and the solver computes the optimal solution. For subsequent control steps:

- The active-set solver uses a warm start where the active constraint set determined in the previous control step becomes the initial guess.
- The interior-point solver continues to use a cold start.

Suboptimal QP Solution

For a given MPC application with constraints, there is no way to predict how many QP solver iterations are required to find an optimal solution. Also, in real-time applications, the number of iterations can change dramatically from one control interval to the next. In such cases, the worst-case execution time can exceed the limit that is allowed on the hardware platform and determined by controller sample time.

You set a guaranteed worst-case execution time for your MPC controller by applying a suboptimal solution after the number of optimization iterations exceeds a specified maximum value. To set the worst-case execution time, first determine the time needed for a single optimization iteration by experimenting with your controller under nominal conditions. Then, set an upper bound on the number of iterations per control interval. For example, if it takes around 1 ms to compute each iteration on the hardware and the controller sample time is 10 ms, set the maximum number of iterations to no greater than `10`.

```
MPCobj.Optimizer.ActiveSetOptions.MaxIterations = 10;
```

By default, an MPC controller object has a lower bound of `120` on the maximum number of iterations for the active-set solver.

By default, when the solver reaches the maximum number of solver iterations without an optimal solution, the controller holds the manipulated variables at their previous values. To use the suboptimal solution reached after the final iteration, set the `UseSuboptimalSolution` option to `true`.

```
MPCobj.Optimizer.UseSuboptimalSolution = true;
```

While the solution is not optimal, the MPC controller adjusts the solution such that it satisfies all your specified constraints.

There is no guarantee that the suboptimal solution performs better than if you hold the controller output constant. You can simulate your system using both approaches, and select the configuration that provides better controller performance.

For an example, see “Use Suboptimal Solution in Fast MPC Applications” on page 5-86.

Custom QP Applications

To access the QP solvers for applications that require solving online QP problems, use the `mpcActiveSetSolver` and `mpcInteriorPointSolver` functions, which are 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

Model Predictive Control Toolbox software lets you specify a custom QP solver for your MPC controller. This solver is called in place of the built-in solvers at each control interval. This option is useful for:

- Validating your simulation results or generating code with an in-house third-party solver that you trust.
- Applications where the built-in solvers do not provide satisfactory performance for your specific problem.

You can define a custom solver for simulation or for code generation. In either instance, you define the custom solver using a custom function and configure your controller to use this custom function.

Task	Custom Solver Function	Affected MATLAB Functions	Affected Simulink Blocks
Simulation Set `Optimizer.CustomSolver` to `true`. `Optimizer.CustomSolverCodeGen` is ignored.	`mpcCustomSolver.m` Supports: • MATLAB code • MEX files	• `sim` • `mpcmove` • `mpcmoveAdaptive` • `mpcmoveMultiple` • `mpcmoveCodeGeneration`	• MPC Controller • Adaptive MPC Controller • Multiple MPC Controllers
Code Generation Set `Optimizer.CustomSolverCodeGen` to `true`. `Optimizer.CustomSolver` is ignored.	`mpcCustomSolverCodeGen.m` Supports: • MATLAB code suitable for code generation • C/C++ code	• `mpcMoveCodeGeneration`	

Custom Solver for Simulation

To simulate an MPC controller with a custom QP solver, perform the following steps.

1. Copy the solver template file to your working folder or anywhere on the MATLAB path, and rename it `mpcCustomSolver.m`. To copy the solver template to your current working folder, type the following at the MATLAB command line.

   ```
   src = which('mpcCustomSolver.txt');
   dest = fullfile(pwd,'mpcCustomSolver.m');
   copyfile(src,dest,'f');
   ```

2. Modify `mpcCustomSolver.m` by adding your own custom solver. Your solver must be able to run in MATLAB and be implemented in a MATLAB script or MEX file.
3. Configure your MPC controller `MPCobj` to use the custom solver.

   ```
   MPCobj.Optimizer.CustomSolver = true;
   ```

   The software now uses your custom solver for simulation in place of the built-in QP KWIK solver.
4. Simulate your controller. For more information, see “Simulation”.

For an example, see “Simulate MPC Controller with a Custom QP Solver” on page 5-77.

Custom Solver for Code Generation

You can generate code for MPC controllers that use a custom QP solver written in either C/C++ code or MATLAB code suitable for code generation.

- To do so at the command line, you must have MATLAB Coder™ software.
- To do so in Simulink you must have Simulink Coder or Simulink PLC Coder™ software.

To generate code for MPC controllers that use a custom QP solver, perform the following steps.

1. Copy the solver template file to your working folder or anywhere on the MATLAB path, and rename it `mpcCustomSolverCodeGen.m`. To copy the MATLAB code template to your current working folder, type the following at the MATLAB command line.

   ```
   src = which('mpcCustomSolverCodeGen_TemplateEML.txt');
   dest = fullfile(pwd,'mpcCustomSolverCodeGen.m');
   copyfile(src,dest,'f');
   ```

   Alternatively, you can use the C template.

   ```
   src = which('mpcCustomSolverCodeGen_TemplateC.txt');
   dest = fullfile(pwd,'mpcCustomSolverCodeGen.m');
   copyfile(src,dest,'f');
   ```

2. Modify `mpcCustomSolverCodeGen.m` by adding your own custom solver.
3. Configure your MPC controller `MPCobj` to use the custom solver.

   ```
   MPCobj.Optimizer.CustomSolverCodeGen = true;
   ```

   The software now uses your custom solver for code generation in place of the built-in QP KWIK solver.
4. Generate code for the controller. For more information, see “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2.

For an example, see “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 10-56.

Custom Solver for both Simulation and Code Generation

You can implement the same custom QP solver for both simulation and code generation. To do so:

- Set both `Optimizer.CustomSolver` and `Optimizer.CustomSolverCodeGen` to `true`.
- Create both `mpcCustomSolver.m` and `mpcCustomSolverCodeGen.m`.

During simulation, your controller uses the `mpcCustomSolver.m` custom function. For code generation, your controller uses the `mpcCustomSolverCodeGen.m` custom function.

You can specify the same MATLAB code in both custom solver functions, provided the code is suitable for code generation.

If you implement `mpcCustomSolverCodeGen.m` using C/C++ code, create a MEX file using the code. You can then call this MEX file from `mpcCustomSolver.m`. For more information on creating and using MEX files, see “C MEX File Applications”.

Custom Solver Function Implementation

When you implement a custom QP solver, your custom function must have one of the following signatures:

- Custom solver for simulation:

  ```
  function [x,status] = mpcCustomSolver(H,f,A,b,x0)
  ```

- Custom solver for code generation:

  ```
  function [x,status] = mpcCustomSolverCodeGen(H,f,A,b,x0)
  ```

For both simulation and code generation, your custom solver has the following input and output arguments.

- `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 the 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 side of the inequality constraint equation, 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 as shown in the following table.

Value	Description
`> 0`	`x` is optimal. `status` represents the number of iterations performed during optimization.
`0`	The maximum number of iterations was reached without finding an optimal solution. The solution `x` might be suboptimal or infeasible. If the `Optimizer.UseSuboptimalSolution` property of your controller is `true`, the controller uses the suboptimal solution in `x` when `status` is `0`.
`-1`	The problem appears to be infeasible, that is, the constraints cannot be satisfied.
`-2`	An unrecoverable numerical error occurred.

Note The MPC controller expects the custom solver functions to solve the QP problem subject to the linear inequality constraints $Ax \geq b$. If your custom solver uses $Ax \leq b$, you must change the sign of both `A` and `b` before passing them to your custom solver code.

Use quadprog as Custom QP Solver

You can configure an MPC object to use the active-set solver available with the `quadprog` function as a custom QP solver.

To automatically configure the MPC object `mpcobj` to use `quadprog` as custom QP solver for both simulation and code generation, you can use the `setCustomSolver` function. Specifically at the MATLAB command prompt, enter the following.

```
setCustomSolver(mpcobj,'quadprog')
```

This command generates, in the current folder, the files `mpcCustomSolver.m` and `mpcCustomSolverCodeGen.m`, which internally call `quadprog`. It then sets `mpcobj.Optimizer.CustomSolver` and `mpcobj.Optimizer.CustomSolverCodeGen` to `true`.

You can also further customize these functions, for example by adjusting the solver options, provided that you use the active-set solver (since other `quadprog` solvers are not supported for MPC problems).

To revert `mpcobj` back to use the built-in algorithm specified in `mpcobj.Optimizer.Algorithm` for both simulation and code generation, call `setCustomSolver` as follows.

```
setCustomSolver(mpcobj,'quadprog')
```

This command sets `mpcobj.Optimizer.CustomSolver` and `mpcobj.Optimizer.CustomSolverCodeGen` to `false`.

Integration with FORCESPRO Solver

You can use FORCESPRO, a real-time embedded optimization software tool developed by Embotech AG, to simulate and generate code for an MPC controller designed using Model Predictive Control Toolbox software. Starting with FORCESPRO 2.0, Embotech provides a plugin that leverages the design capabilities of Model Predictive Control Toolbox software and the computational performance of FORCESPRO. Using the plugin, you can generate a custom QP solver that allows deployment on real-time hardware and is highly optimized based on your specific MPC problem to achieve satisfactory real-time performance. Especially long-horizon MPC problems can be solved very efficiently.

For information on using the FORCESPRO solver together with Model Predictive Control Toolbox software, see “Implement MPC Controllers using Embotech FORCESPRO Solvers” on page 10-67.

References

[1] Schmid, C., and L.T. Biegler. "Quadratic Programming Methods for Reduced Hessian SQP." *Computers & Chemical Engineering* 18, no. 9 (September 1994): 817–32. https://doi.org/10.1016/0098-1354(94)E0001-4.

See Also

`mpc` | `mpcmove` | `mpcActiveSetSolver` | `mpcInteriorPointSolver`

More About

- “Optimization Problem” on page 2-7
- “Simulate MPC Controller with a Custom QP Solver” on page 5-77

Controller Refinement

- “Setting Targets for Manipulated Variables” on page 3-2
- “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5
- “Use Custom Constraints in Blending Process” on page 3-9
- “Terminal Weights and Constraints” on page 3-18
- “Provide LQR Performance Using Terminal Penalty Weights” on page 3-20
- “Adjust Disturbance and Noise Models” on page 3-25
- “Custom State Estimation” on page 3-32
- “Implement Custom State Estimator Equivalent to Built-In Kalman Filter” on page 3-37
- “Manipulated Variable Blocking” on page 3-44
- “Specifying Alternative Cost Function with Off-Diagonal Weight Matrices” on page 3-48

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 setpoint for one of the two manipulated variables.

Define Plant Model

The linear plant model has two inputs and one output. Define the plant as a transfer function, convert it to state space, specify the initial state, and extract the plant matrices for later use within the Simulink model.

```
plant = ss(tf({[3 1],[2 1]},{[1 2*.3 1],[1 2*.5 1]}));
x0 = [0 0 0 0]';
A = plant.A;
B = plant.B;
C = plant.C;
D = plant.D;
```

Design MPC Controller

Create an MPC controller with sampling time `0.4` s, and prediction and control horizons of `20` and `5` steps, respectively.

```
mpcobj = mpc(plant,0.4,20,5);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify weights for both manipulated variables and output.

```
mpcobj.weights.manipulated = [0.3 0]; % weight difference MV#1 - Target#1
mpcobj.weights.manipulatedrate = [0 0];
mpcobj.weights.output = 1;
```

Define constraints for the manipulated variable rate.

```
mpcobj.MV = struct('RateMin',{-0.5;-0.5},'RateMax',{0.5;0.5});
```

Set a target for one manipulated variable

Specify target setpoint `u = 2` for the first manipulated variable.

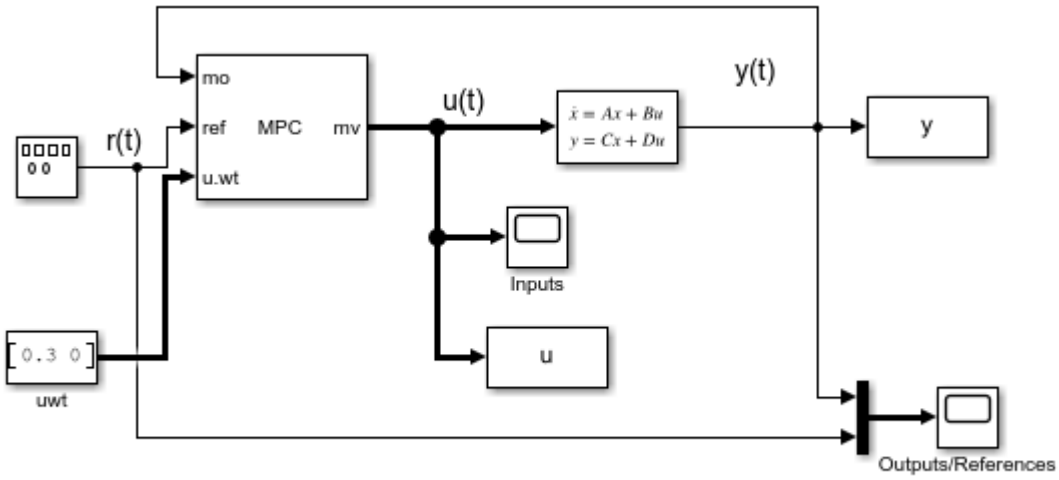
```
mpcobj.MV(1).Target=2;
```

Simulate with Simulink

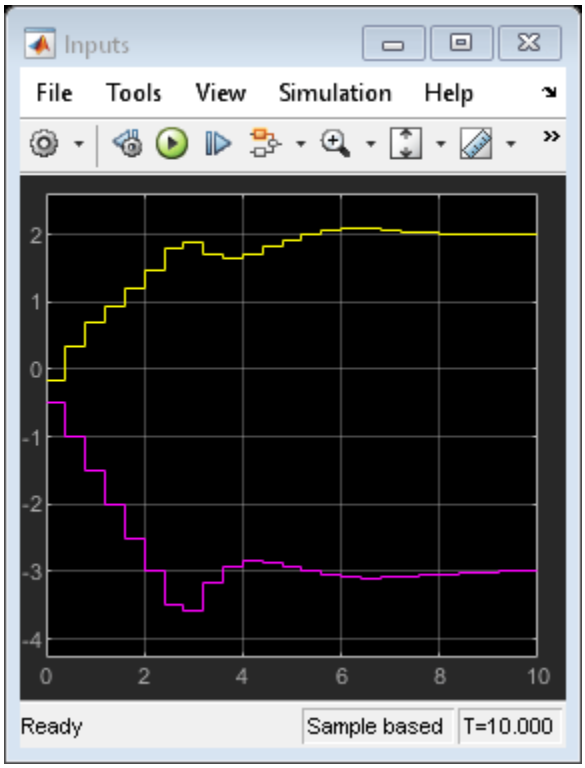
Define the model name and open the Simulink model. Note that the output reference is a square wave. Then simulate the model, using the `sim` command.

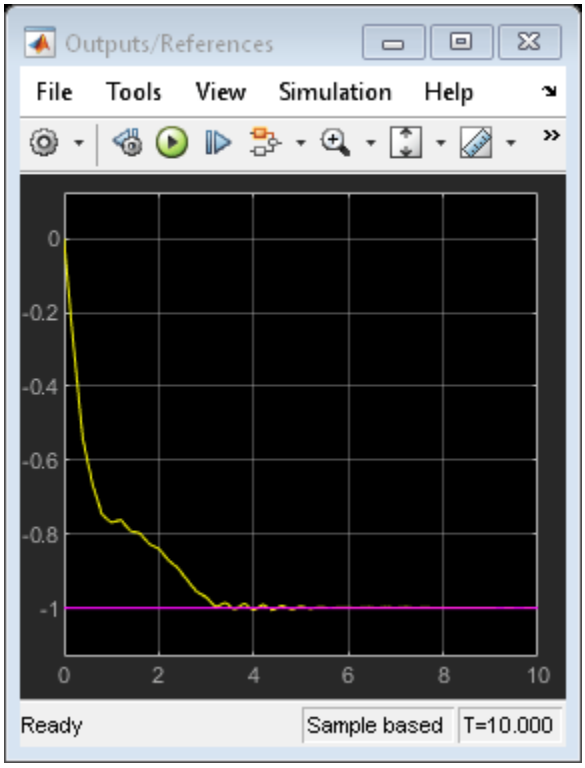
```
mdl = 'mpc_utarget';
open_system(mdl)
sim(mdl);
```

```
-->Converting model to discrete time.
-->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 measured
```



Copyright 1990-2014 The MathWorks, Inc.





The first plot shows that the first manipulated variable reaches its set point after about 6 seconds, while the plant output reaches its reference.

```
bdclose(mdl)          % close the Simulink model
```

See Also

`mpc` | MPC Controller

More About

- “MPC Signal Types”

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:

$$Eu\left(k+i\right)+Fy\left(k+i\right)+Sv\left(k+i\right)\le G+\varepsilon_k V$$

Here:

- ε_k is the QP slack variable used for constraint softening. For more information, see “Constraint Softening” on page 1-7.
- $u\left(k+i\right)$ are the N_{mv} manipulated variable values, in engineering units.
- $y\left(k+i\right)$ are the N_y predicted plant outputs, in engineering units.
- $v\left(k+i\right)$ are the N_{md} measured plant disturbance inputs, in engineering units.
- E, F, S, G, and V are constant matrices and vectors. For more information, see `setconstraint`.

As with the QP cost function, output prediction using the state observer makes these constraints a function of the QP decision variables.

To set the mixed input/output constraints of an MPC controller, use the `setconstraint` function. To obtain the existing constraints from a controller, use `getconstraint`.

When using mixed input/output constraints, consider the following:

- Mixed input/output constraints are dimensional by default.
- Run-time updating of mixed input/output constraints is supported at the command line and in Simulink®. For more information, see “Update Constraints at Run Time” on page 5-27.
- Using mixed input/output constraints is not supported in **MPC Designer**.

As an example, consider an MPC controller for a double-integrator plant with mixed input/output constraints.

Create Initial MPC Controller

The basic setup of the MPC controller includes:

- A double integrator as the prediction model
- Prediction horizon of 20
- Control horizon of 20
- Input constraints: $-1\le u\left(t\right)\le 1$

```
plant = tf(1,[1 0 0]);
Ts = 0.1;
p = 20;
m = 20;
mpcobj = mpc(plant,Ts,p,m);
mpcobj.MV = struct('Min',-1,'Max',1);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Define Mixed Input/Output Constraints

Constrain the sum of the input `u(t)` and output `y(t)` must be nonnegative and smaller than 1.2:

$$0 \le u(t) + y(t) \le 1.2$$

To impose this combined (mixed) I/O constraint, formulate it as a set of inequality constraints involving $u(t)$ and $y(t)$.

$$u(t) + y(t) \le 1.2$$
$$-u(t) - y(t) \le 0$$

To define these constraints using the `setconstraint` function, set the constraint constants as follows:

$$E = \begin{bmatrix} 1 \\ -1 \end{bmatrix},\ F = \begin{bmatrix} 1 \\ -1 \end{bmatrix},\ G = \begin{bmatrix} 1.2 \\ 0 \end{bmatrix}$$

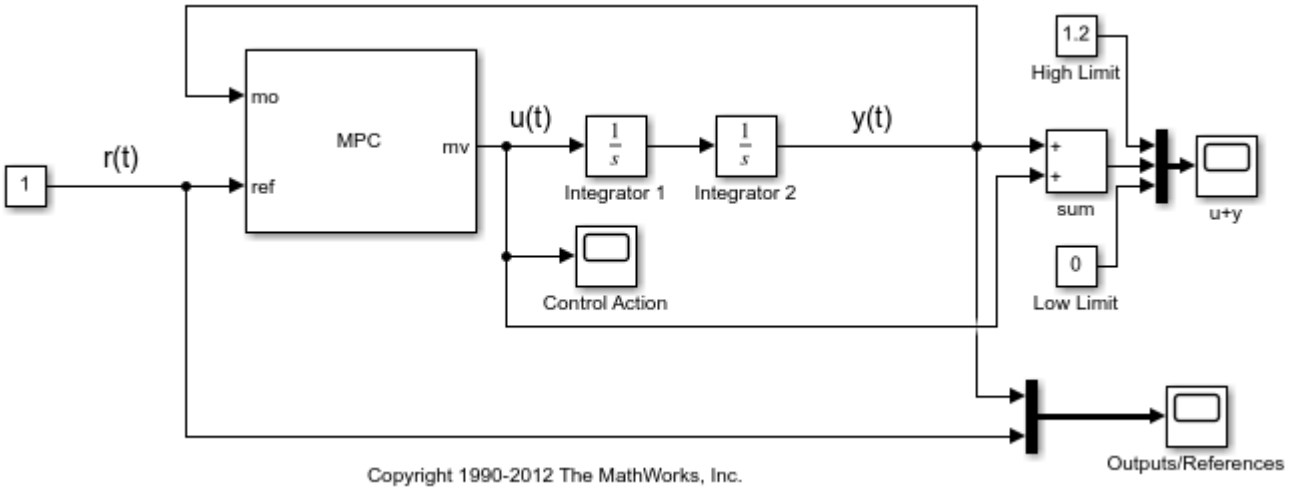
```
setconstraint(mpcobj,[1;-1],[1;-1],[1.2;0]);
```

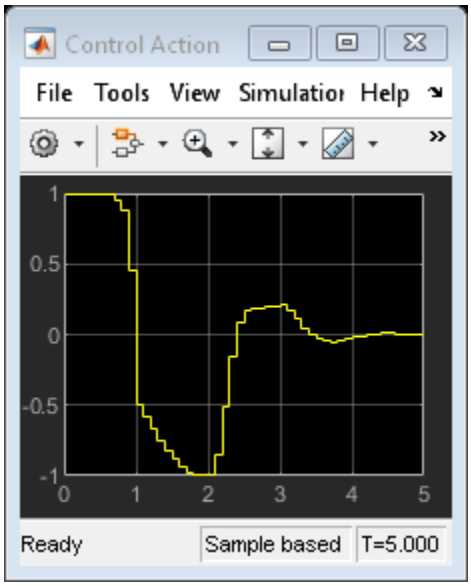
Simulate Controller

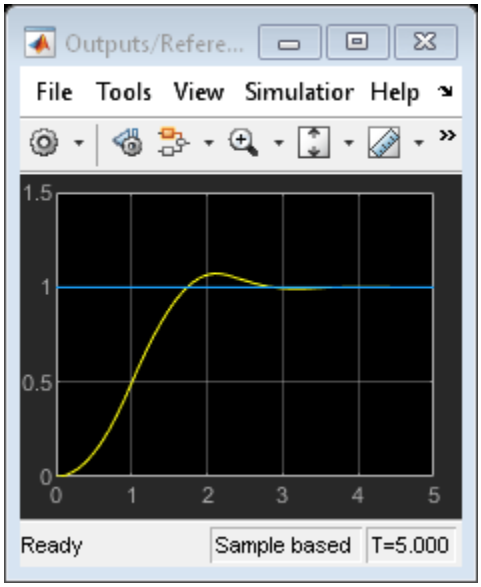
Simulate closed-loop control of the linear plant model in Simulink. The controller `mpcobj` is specified in the MPC Controller block.

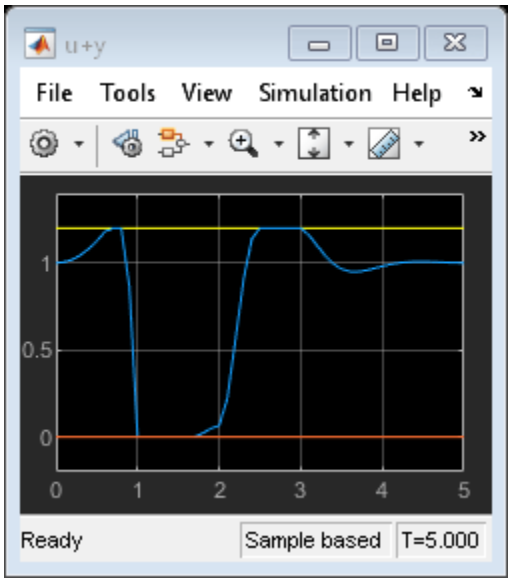
```
mdl = 'mpc_mixedconstraints';
open_system(mdl)
sim(mdl)
```

```
-->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 measured
```




Control Action
File Tools View Simulation Help
1
0.5
0
-0.5
-1
0 1 2 3 4 5
Ready
Sample based
T=5.000


Outputs/Refere...
File Tools View Simulation Help
1.5
1
0.5
0
0 1 2 3 4 5
Ready
Sample based
T=5.000



The MPC controller keeps the sum $u + y$ between 0 and 1.2 while tracking the reference signal, $r = 1$.

```
bdclose(mdl)
```

See Also

`setconstraint` | `getconstraint`

More About

- “Optimization Problem” on page 2-7
- “Update Constraints at Run Time” on page 5-27
- “Use Custom Constraints in Blending Process” on page 3-9

Use Custom Constraints in Blending Process

This example shows how to design an MPC controller for a blending process using custom mixed input/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:

$$\frac{dv}{d\tau} = \sum_{i=1}^{3} \phi_i - \phi$$
$$V\frac{d\gamma_j}{d\tau} = \sum_{i=1}^{3} (\gamma_{ij} - \gamma_j)\,\phi_i$$

where

- V is the mixture inventory (in the container).
- ϕ_i is the plow rate for feed i.
- 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:

$$\phi_1 = \phi_T - \phi_2 - \phi_2$$

Each feed has limited availability:

$$0 \le \phi_i \le \phi_{i,\max}$$

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); gij-1];
```

Change the MV definition to [FT, F2, F3] where F1 = FT - F2 - F3

```
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'};
Model.InputGroup.MV = 1:3;
Model.InputGroup.MD = 4;
Model.OutputName = {'V','c_1','c_2'};
```

Create MPC Controller

Specify the sample time, prediction horizon, and control horizon for the controller.

```
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 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

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 for 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. To pay more attention to controlling the inventory and the composition of the first constituent, use larger weights for the first two outputs.

```
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 Mixed 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.

$$0 \le \phi_1 = \phi_T - \phi_2 - \phi_3 \le 0.8$$

Specify this constraint in the form $Eu + Fy \le g$.

```
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)
```

Simulate Model in Simulink

The Simulink model contains a nonlinear model of the blending process and an unmeasured 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.

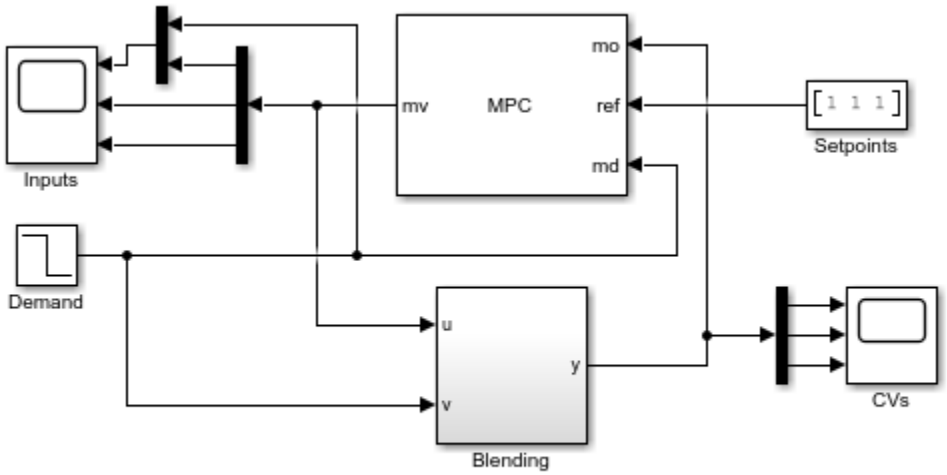
The model simulates the following scenario:

- 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.

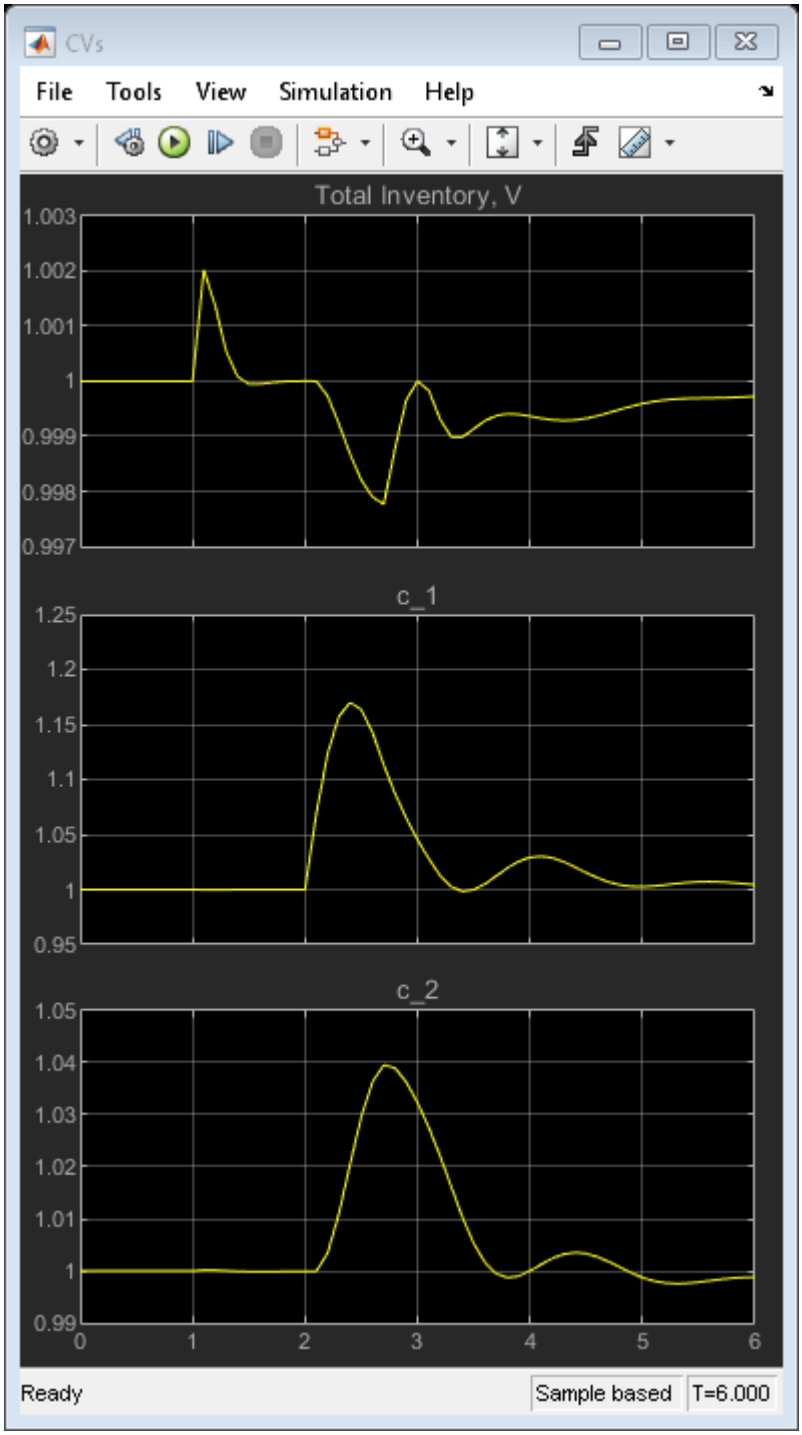
Open and simulate the Simulink model.

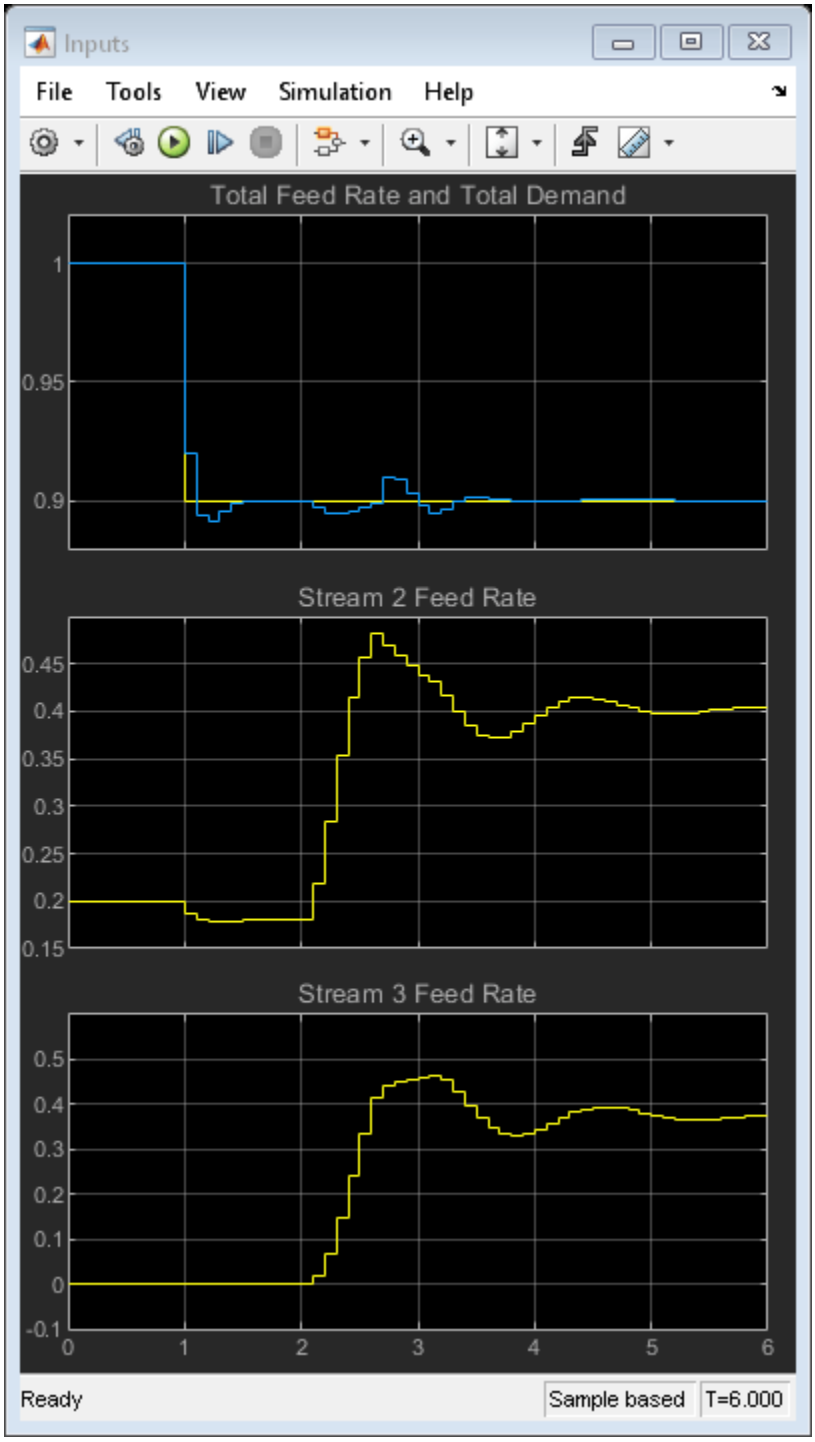
```
mdl = 'mpc_blendingprocess';
open_system(mdl)
sim(mdl)
```

```
-->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 noise.
-->Assuming output disturbance added to measured output channel #3 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



Copyright 1990-2014 The MathWorks, Inc.


CVs
File Tools View Simulation Help
Total Inventory, V
1.003
1.002
1.001
1
0.999
0.998
0.997
c_1
1.25
1.2
1.15
1.1
1.05
1
0.95
c_2
1.05
1.04
1.03
1.02
1.01
1
0.99
0 1 2 3 4 5 6
Ready
Sample based
T=6.000



In the simulation:

- At time 0, the plant operates steadily at the nominal conditions.
- At time 1, the demand decreases by 10%, and the controller maintains the inventory close to its setpoint.

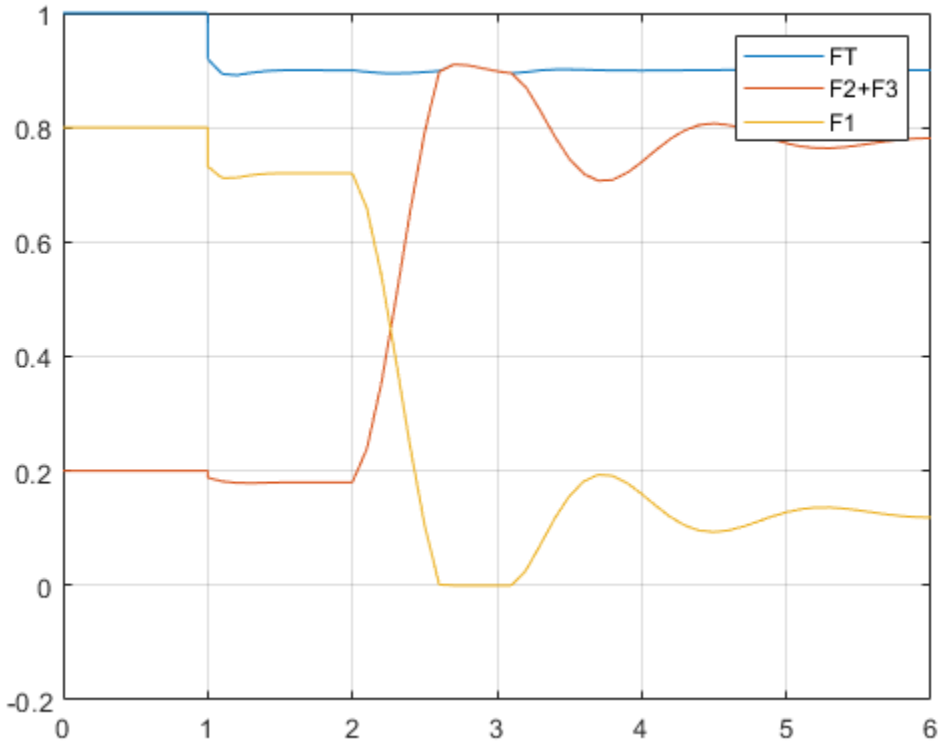
- At time 2, there is a large unmeasured increase in the concentration of constituent 1 contained in feed 1. This disturbance causes a prediction error and a large disturbance in the blend composition.

The disturbance is a nonlinear effect, but the linear MPC controller recovers well 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 unmeasured disturbance occurs at time 2, which requires the controller to decrease F1. During the transient, F1 becomes zero. If the mixed input/output constraint had not been included, F1 would have been negative. The controller requests for FT, F2, and F3 would have been impossible to satisfy, which would lead to a performance degradation. With the constraint included, the controller does its best given the physical limits of the system.

```
bdclose(mdl)
```

See Also

`setconstraint`

More About

- “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5

Terminal Weights and Constraints

Terminal weights are the quadratic weights Wy on $y(t+p)$ and Wu 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:

$$J(u) = \sum_{i=1}^{\infty} x(k+i)^T Q x(k+i) + u(k+i-1)^T R u(k+i-1) \tag{3-1}$$

where x is the vector of plant states in the standard state-space form:

$$x(k+1) = Ax + Bu(k) \tag{3-2}$$

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:

$$J(u) = \sum_{i=1}^{p-1} [x(k+i)^T Q x(k+i) + u(k+i-1)^T R u(k+i-1)] + x(k+p)^T Q_p x(k+p) \tag{3-3}$$

where Q_p , the terminal penalty matrix, is the solution of the Riccati equation:

$$Q_p = A^T Q_p A - A^T Q_p B (B^T Q_p B + R)^{-1} B^T Q_p A + Q \tag{3-4}$$

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 2-7 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 2-9 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 3-2") 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`

More About

• “Provide LQR Performance Using Terminal Penalty Weights” on page 3-20

Provide LQR Performance Using Terminal Penalty Weights

It is possible to make a finite-horizon model predictive controller equivalent to an infinite-horizon linear quadratic regulator (LQR) by setting tuning weights on the terminal predicted states.

The standard MPC cost function is similar to the cost function for 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 toward zero, whereas the MPC cost function forces y and u toward nonzero setpoints. You can shift the MPC prediction model origin to eliminate this difference and achieve zero nominal setpoints.
- The LQR cost function uses an infinite prediction horizon in which the manipulated variable changes at each sample time. 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:

$$J(u) = \sum_{i=1}^{p-1} \left(y(k+i)^T Q y(k+i) + u(k+i-1)^T R u(k+i-1) \right) + x(k+p)^T Q_p x(k+p)$$

Here, Q_p is a terminal penalty weight applied at the final prediction horizon step, and the prediction and control horizons are equal ($p = m$). The required Q_p is the Riccati matrix calculated using the `lqr` and `lqry` commands.

Define Plant Model

Specify the discrete-time open-loop dynamic plant model with a sample time of `0.1` seconds. For this model, make all states measurable outputs of the plant. This plant is the double integrator plant from [1].

```
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);
```

Design Infinite-Horizon LQR Controller

Compute the Riccati matrix `Qp` and state feedback gain `K` associated with the LQR problem with output weight `Q` and input weight `R`. For more information, see `lqry`.

```
Q = eye(2);
R = 1;
[K,Qp] = lqry(plant,Q,R);
```

Design Equivalent MPC Controller

To implement the MPC cost function, first compute L, the Cholesky decomposition of Q_p, such that $L^TL = Q_p$.

```
L = chol(Qp);
```

Next, define auxiliary unmeasured output variables $y_c = Lx$, such that $y_c^T y_c = x^T Q_p x$. Augment the output vector of the plant such that it includes these auxiliary outputs.

```
newPlant = plant;
set(newPlant,'C',[C;L],'D',[D;zeros(2,1)]);
```

Configure the state vector outputs as measured outputs and the auxiliary output signals as unmeasured outputs. By default, the input signal is the manipulated variable.

```
newPlant = setmpcsignals(newPlant,'MO',[1 2],'UO',[3 4]);
```

Create the controller object with the same sample time as the plant and equal prediction and control horizons.

```
p = 3;
m = p;
mpcobj = mpc(newPlant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Define tuning weights at each step of the prediction horizon for the manipulated variable and the measured outputs.

```
ywt = sqrt(diag(Q))';
uwt = sqrt(diag(R))';
mpcobj.Weights.OV = [sqrt(diag(Q))' 0 0];
mpcobj.Weights.MV = sqrt(R);
```

To make the QP problem associated with the MPC controller positive definite, include very small weights on manipulated variable increments.

```
mpcobj.Weights.MVRate = 1e-5;
```

Impose the terminal penalty $x^T(k+p)Q_p x(k+p)$ by specifying a unit weight on $y_c(k+p) = Lx(k+p)$. The terminal weight on `u(t+p-1)` remains the same.

```
Y = struct('Weight',[0 0 1 1]);
U = struct('Weight',uwt);
setterminal(mpcobj,Y,U);
```

Since the measured output vector contains the entire state vector, remove any additional output disturbance integrator inserted by the MPC controller.

```
setoutdist(mpcobj,'model',ss(zeros(4,1)));
```

Remove the state estimator by defining the following measurement update equation:

```
x[n|n] = x[n|n-1] + I * (x[n]-x[n|n-1]) = x[n]
```

Since the `setterminal` function resets the state estimator to its default value, call the `setEstimator` function after calling `setterminal`.

```
setEstimator(mpcobj,[],eye(2));
```

Compare MPC and LQR Controller Gains

Compute the gain of the MPC controller when the constraints are inactive (unconstrained MPC), and compare it to the LQR gain.

```
mpcgain = dcgain(ss(mpcobj));
```

```
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

```
fprintf('\n(unconstrained) MPC: u(k)=[%8.8g,%8.8g]*x(k)',mpcgain(1),mpcgain(2));
```

```
(unconstrained) MPC: u(k)=[-1.6355962,-0.91707456]*x(k)
```

```
fprintf('\n                LQR: u(k)=[%8.8g,%8.8g]*x(k)\n\n',-K(1),-K(2));
```

```
                LQR: u(k)=[-1.6355962,-0.91707456]*x(k)
```

The state feedback gains are exactly the same.

Compare Controller Performance

Compare the performance of the LQR controller, the MPC controller with terminal weights, and a standard MPC controller.

Compute the closed-loop response for the LQR controller.

```
clsys = feedback(plant,K);
Tstop = 6;
x0 = [0.2;0.2];
[yLQR,tLQR] = initial(clsys,x0,Tstop);
```

Compute the closed-loop response for the MPC controller with terminal weights.

```
simOpt = mpcsimopt(mpcobj);
simOpt.PlantInitialState = x0;
r = zeros(1,4);
[y,t,u] = sim(mpcobj,ceil(Tstop/Ts),r,simOpt);
```

Create a standard MPC controller with default prediction and control horizons (`p`=10, `m`=3). To match the other controllers, remove the output disturbance model and the default state estimator from the standard MPC controller.

```
mpcobjSTD = mpc(plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

```
mpcobjSTD.Weights.MV = uwt;
mpcobjSTD.Weights.OV = ywt;
```

```
setoutdist(mpcobjSTD,'model',tf(zeros(2,1)))
setEstimator(mpcobjSTD,[],C)
```

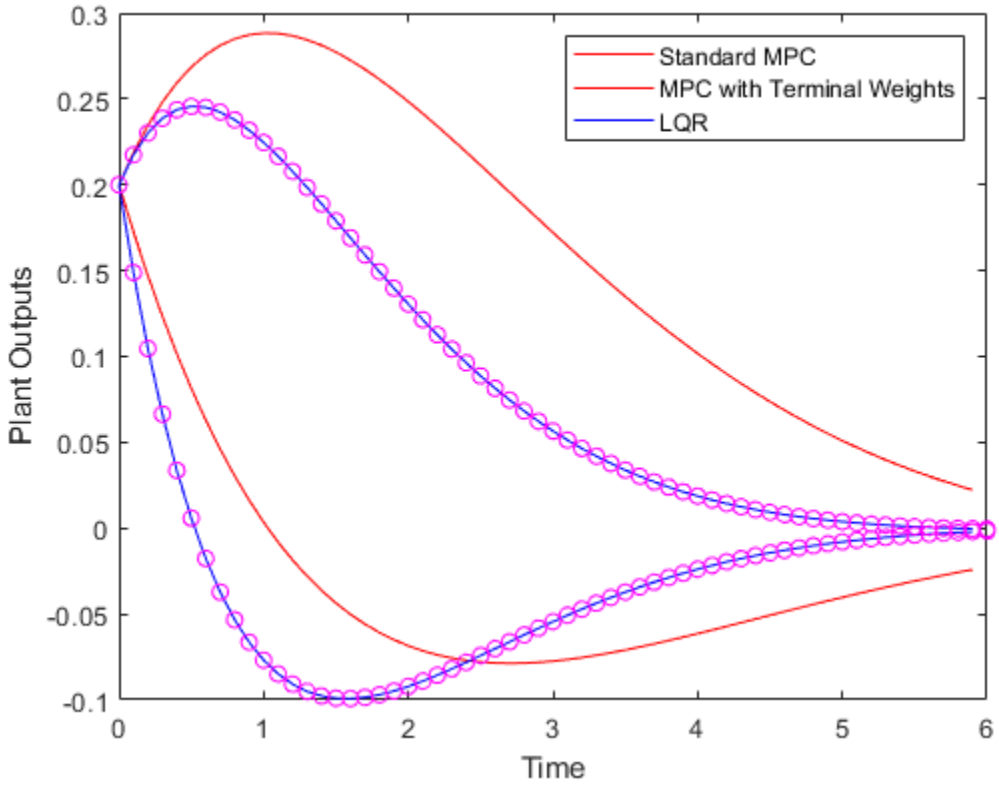
Compute the closed-loop response for the standard MPC controller.

```
simOpt = mpcsimopt(mpcobjSTD);
simOpt.PlantInitialState = x0;
r = zeros(1,2);
[ySTD,tSTD,uSTD] = sim(mpcobjSTD,ceil(Tstop/Ts),r,simOpt);
```

```
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Compare the controller responses.

```
plot(tSTD,ySTD,'r',t,y(:,1:2),'b',tLQR,yLQR,'mo')
xlabel('Time')
ylabel('Plant Outputs')
legend('Standard MPC','MPC with Terminal Weights','LQR','Location','NorthEast')
```



The MPC controller with terminal weights has a faster settling time compared to the standard MPC controller. The LQR controller and the MPC controller with terminal weights perform identically.

You can improve the standard MPC controller performance by adjusting the horizons. For example, if you increase the prediction and control horizons (`p=20`, `m=5`), the standard MPC controller performs almost identically to the MPC controller with terminal weights.

This example shows that using terminal penalty weights can eliminate the need to tune the prediction and control horizons for the unconstrained MPC 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 more information, see [2].

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 D. Q. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Publishing, 2010.

See Also

`setterminal`

More About

- “Optimization Problem” on page 2-7
- “Terminal Weights and Constraints” on page 3-18

Adjust Disturbance and Noise Models

A model predictive controller 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.

Overview

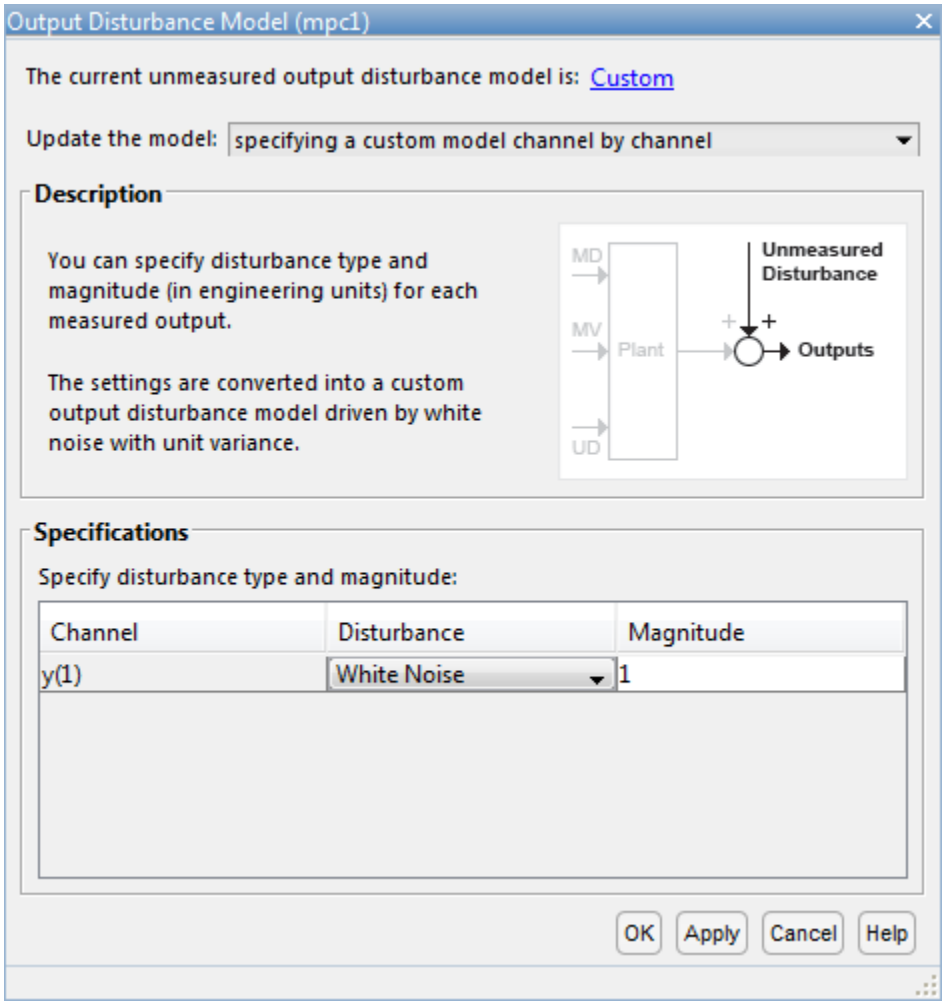
MPC attempts to predict how known and unknown events affect the plant output variables (OVs). Known events are changes in the measured plant input variables (MV and MD inputs). The plant model of the controller predicts the impact of these events, and such predictions can be quite accurate. For more information, see “MPC Prediction Models”.

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 noise 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 Prediction Models”, the output disturbance model is independent of the plant, and its output adds directly to that of the plant model.

Using **MPC Designer**, you can specify the type of noise that is expected to affect each plant OV. In the 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

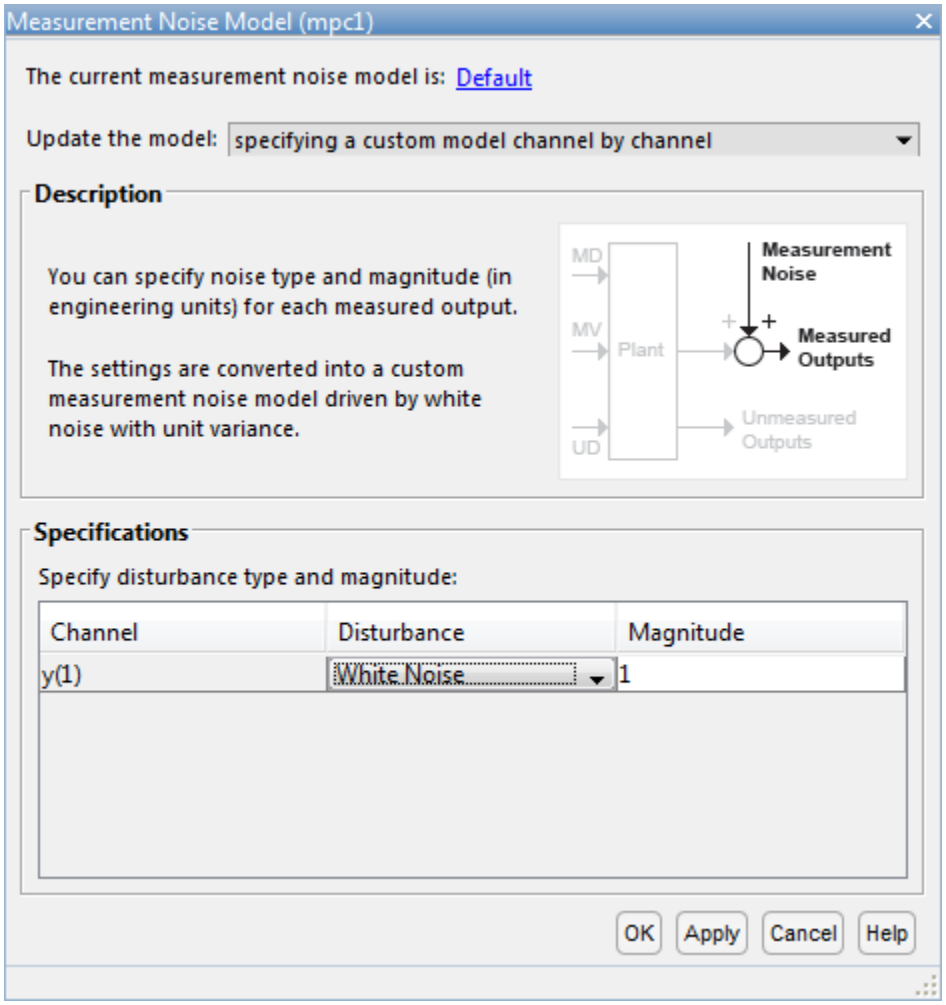
You can also specify the white noise input **Magnitude** for each disturbance model, overriding the assumption of unit variance. As you increase the noise magnitude, the controller responds more aggressively to a given prediction error. The specified noise magnitude corresponds to the static gain in 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 **MPC Designer**, you can specify the expected measurement noise magnitude and character. In the 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.

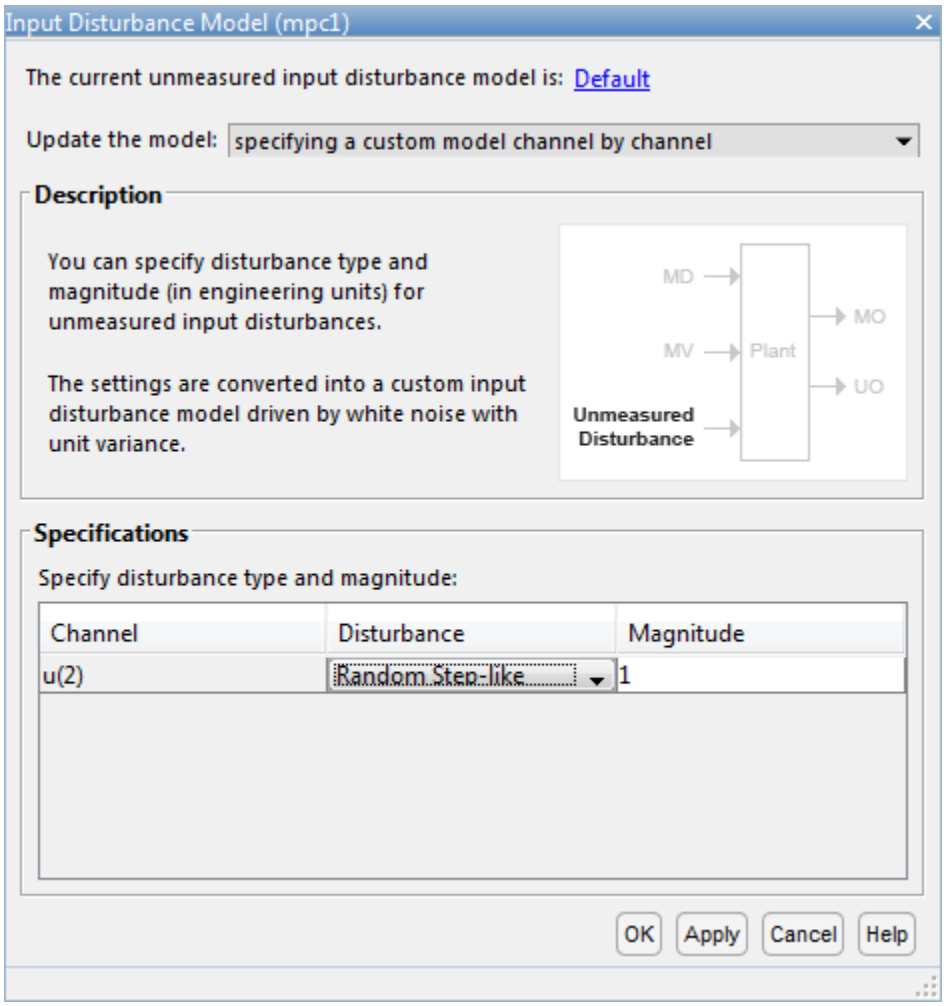
Input Disturbance Model

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 Prediction Models”, 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 **MPC Designer** to specify the type of disturbance you expect for each UD input. In the 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 **MPC Designer**.

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 2-2, 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. **MPC Designer** 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.

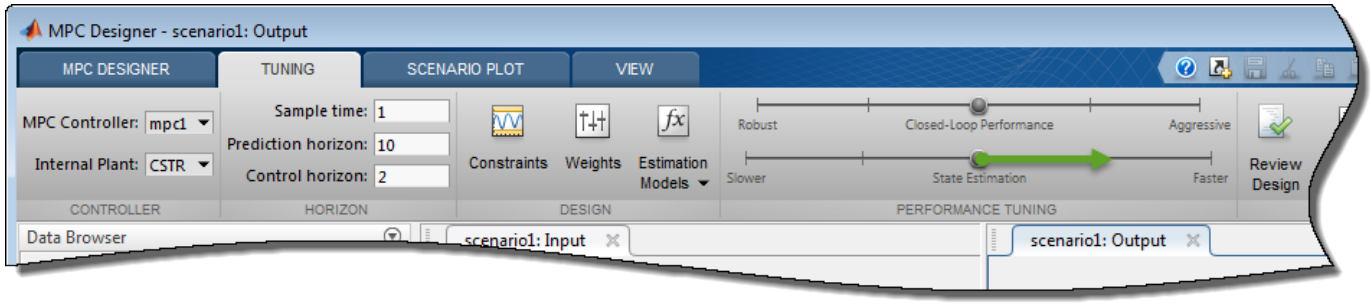
Disturbance Rejection Tuning

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 “Specify Scale Factors” on page 1-29). 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 “Tune Weights” on page 1-43), 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 **MPC Designer** 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 **MPC Designer**, 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 **MPC Designer**, do this by increasing the measurement noise magnitude. If this helps but is insufficient, increase the magnitude further.
- In **MPC Designer**, 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

Apps

MPC Designer

Functions

`setmpcsignals` | `getindist` | `setindist` | `getoutdist` | `setoutdist`

More About

- “MPC Prediction Models”
- “Controller State Estimation” on page 2-2
- “Design Controller Using MPC Designer”

Custom State Estimation

Model Predictive Control Toolbox™ software allows you to override the default controller state estimation method.

To do so, you can use the following methods:

- You can override the default Kalman gains, L and M, using the `setEstimator` function. To obtain the default values from the controller use `getEstimator`. These commands assume that the columns of L and M are in the engineering units of the measured plant outputs. Internally, the software converts them to dimensionless form.
- You can use the custom estimation option, which skips all Kalman gain calculations within the controller. When the controller operates, at each control interval you must use an external procedure to estimate the controller states and provide these state estimates to the controller.

Custom state estimation is not supported in **MPC Designer**. For more information see “Controller State Estimation” on page 2-2 and “Implement Custom State Estimator Equivalent to Built-In Kalman Filter” on page 3-37.

Define Plant Model

Consider the case of a double integrator plant for which all of the plant states are measurable. In such a case, you can provide the measured states to the MPC controller rather than have the controller estimate the states.

The linear open-loop plant model is a double integrator.

```
plant = tf(1,[1 0 0]);
```

Design MPC Controller

Create the controller object with a sample time of `0.1` seconds, a prediction horizon of `10` steps, and control horizon of 3 steps.

```
Ts=0.1;
mpcobj = mpc(plant,Ts,10,3);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify actuator saturation limits as manipulated variable constraints.

```
mpcobj.MV = struct('Min',-1,'Max',1);
```

Configure the controller to use custom state estimation.

```
setEstimator(mpcobj,'custom');
```

Simulate Controller

Initialize variables to store the closed-loop responses.

```
Tf = round(5/Ts);
YY = zeros(Tf,1);
UU = zeros(Tf,1);
```

Prepare the plant used in the simulation by converting it to a discrete-time model and setting the initial state.

```
sys = c2d(ss(plant),Ts);
xsys = [0;0];
```

Get an handle to the `mpcstate` object that is used to store the controller states.

```
xmpc = mpcstate(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 measured
```

Iteratively simulate the closed-loop response using the `mpcmove` function.

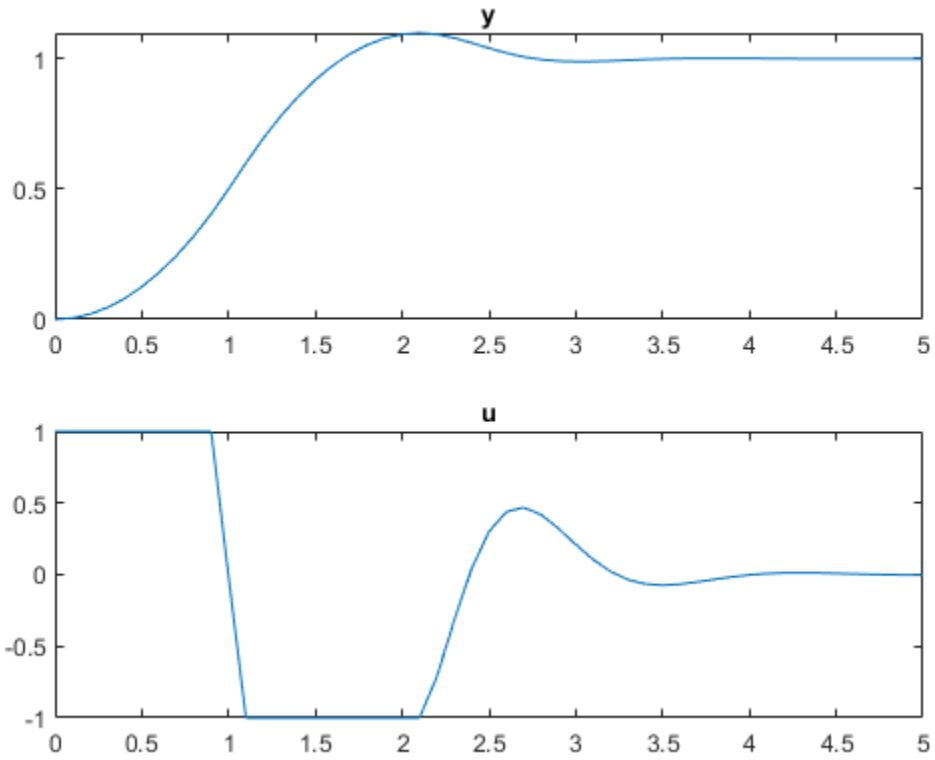
For each simulation step:

- Obtain the plant output, `y`, from the current state.
- Store the plant output.
- Set the plant state in the `mpcstate` object to the current measured state values, `xsys`, using the handle `xmpc`. For plants in which the state is not measurable, a state estimation must be provided by an observer. When using the built-in estimator, this is not needed.
- Compute the MPC control action, `u`, passing in the `mpcstate` object and the output reference, `1`.
- Store the control signal.
- Update the plant state.

```
for t = 0:Tf

    y = sys.C*xsys; % plant equations: output
    YY(t+1) = y;

    xmpc.Plant = xsys; % state estimation

    u = mpcmove(mpcobj,xmpc,[],1); % y is not needed
    UU(t+1) = u;

    xsys = sys.A*xsys + sys.B*u; % plant equations: next state
end
```

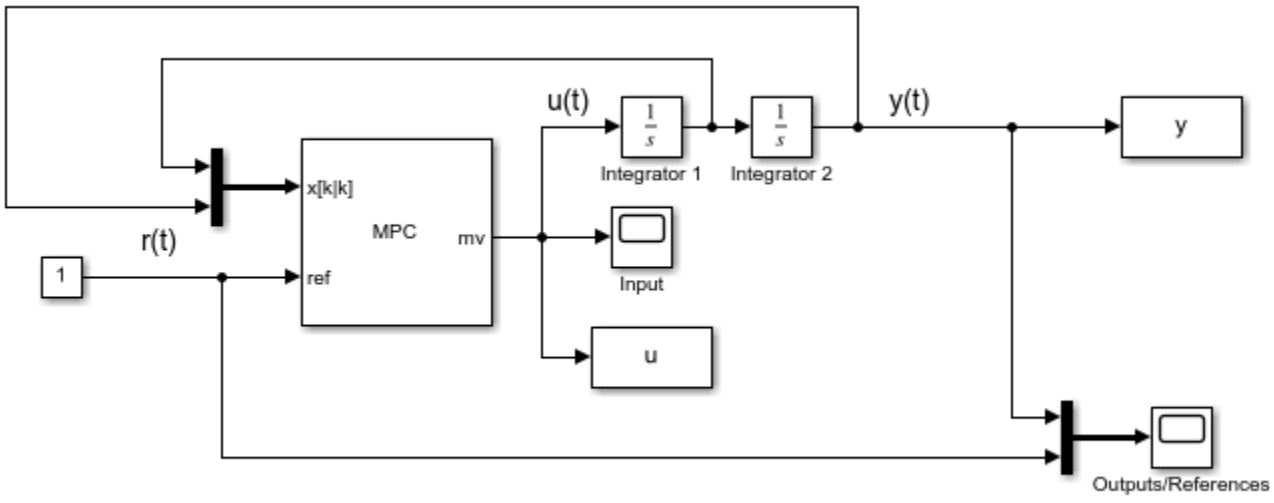
Plot the simulation results.

```
figure
subplot(2,1,1)
plot(0:Ts:5,YY)
title('y')
subplot(2,1,2)
plot(0:Ts:5,UU)
title('u')
```

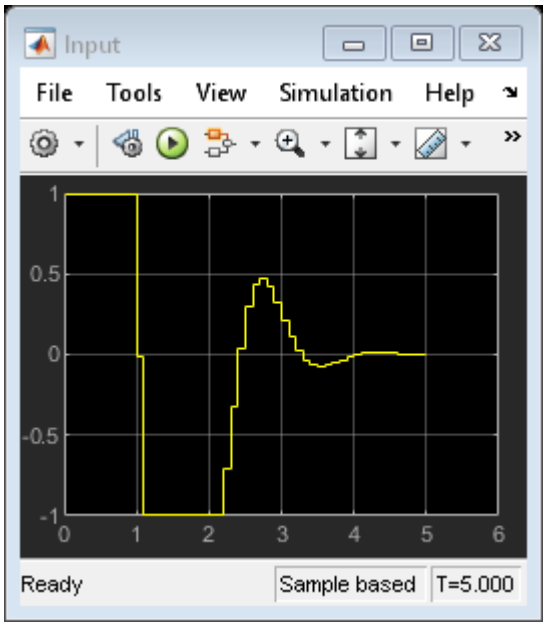


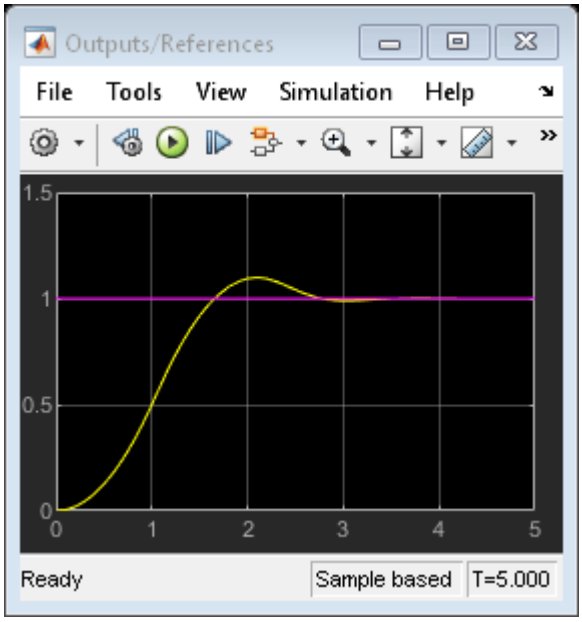
Simulate closed-loop control of the linear plant model in Simulink. For this model, the controller `mpcobj` is specified in the MPC Controller block.

```
mdl = 'mpc_customestimation';
open_system(mdl)
sim(mdl)
```



Copyright 1990-2014 The MathWorks, Inc.





The closed-loop responses for the MATLAB and Simulink simulations are identical.

```
fprintf('\nDifference between simulations in MATLAB and Simulink is %g\n',norm(UU-u));
```

```
Difference between simulations in MATLAB and Simulink is 6.77858e-14
```

Close the Simulink model.

```
bdclose(mdl)
```

See Also

`getEstimator` | `setEstimator`

More About

- “Controller State Estimation” on page 2-2
- “Implement Custom State Estimator Equivalent to Built-In Kalman Filter” on page 3-37

Implement Custom State Estimator Equivalent to Built-In Kalman Filter

This example shows how to design and implement a custom state estimator that is equivalent to the built-in Kalman Filter automatically designed by a linear MPC controller.

Overview

The linear MPC controller provided by Model Predictive Control Toolbox software includes a default state estimator to facilitate controller implementation. For a linear time-invariant (LTI) MPC controller whose prediction model never changes at run-time, the default state estimator is an LTI Kalman filter. This filter is automatically designed using the prediction model specified in the MPC object, and it generally works for many MPC applications. However, to achieve satisfactory results in some cases, you must customize or replace the Kalman filter. To do so, you can perform custom state estimation; that is, bypass the built-in estimator and provide estimated states directly to the MPC controller as run-time signals, `x[k|k]`.

One common approach to designing a custom state estimator is to first implement one that is equivalent to the built-in Kalman filter. You can then use it as a starting point for customization, for example by modifying its structure or tuning its parameters. In this example, you first explore a simple LTI MPC controller to understand how the built-in Kalman filter is generated. You then implement an equivalent estimator in Simulink using either core Simulink blocks or the Kalman Filter block included with Control System Toolbox software.

Design LTI MPC for Plant Running at Specific Operating Point

Assume that you have a continuous-time, single-input, single-output nonlinear plant. The plant has two states and you want to design an LTI MPC controller at a steady-state operating point.

Specify the state, input, and output values at this operating point.

```
x0 = [1.6931; 4.3863];
u0 = 7.7738;
y0 = 0.5;
```

You can obtain a linear plant model at the operating point in several ways, such as system identification and linearization. For this example, assume that `G` is the resulting linear plant model.

```
G = ss([-2 1;-9.7726 -1.3863],[0;1],[0.5 0],0);
```

To use the plant for custom state estimation, discretize the plant model with a sample time of 0.1.

```
Ts = 0.1;
Gd = c2d(G,Ts);
```

Create an LTI MPC controller using the linear plant model.

```
mpcobj = mpc(G,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Set the nominal values of the internal plant model to reflect the steady-state operating point.

```
mpcobj.Model.Nominal.Y = y0;
mpcobj.Model.Nominal.U = u0;
mpcobj.Model.Nominal.X = x0;
```

Make the controller less aggressive by reducing the output tuning weight from 1 to 0.2. Use default values for other controller parameters.

```
mpcobj.Weights.OutputVariables = 0.2;
```

Automatic Design of Built-in Kalman Filter

As a feedback controller, one important task of an MPC controller is to fully reject disturbances at run time, that is, the steady-state error at the plant output is zero in the presence of a disturbance. MPC controllers use a *disturbance model* to define the disturbance to be rejected. The states of the disturbance model reflect the presence of such a disturbance at run time.

Since the most common disturbance is an unmeasured step disturbance added at the plant output, MPC controller is configured to reject such a disturbance by default. To be more specific, MPC uses a discrete-time integrator with dimensionless unity gain as the default output disturbance model. Its input, `w`, is white noise with zero-mean and unit variance. You can examine the default disturbance model using the `getoutdist` function.

```
God = getoutdist(mpcobj);
```

```
-->Converting model to discrete time.
-->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 measured
```

In other words, MPC expects to reject a random step-like disturbance at the plant output with an expected magnitude of 1 (after scaling). In this example, keep this default output disturbance model.

To get the overall prediction model used by the MPC controller, `Gpred`, augment the plant model with the disturbance model. The measured output seen by MPC now becomes the sum of plant output and disturbance model output. Be aware that the white noise input value `w` is not an input to the prediction model because `w` is unmeasured.

```
A = blkdiag(Gd.A,God.A);
Bu = [Gd.B; 0];
Cm = [Gd.C God.C];
D = Gd.D;
Gpred = ss(A,Bu,Cm,D,Ts);
```

Since the MPC controller still needs to know all the prediction model states at run time, including the integrator state from the output disturbance model, an LTI Kalman Filter is automatically designed to estimate the three states at run time.

In the default design, the observer used by the Kalman filter is:

```
x[k+1] =  A*x[k] + w[k]
y[k] = Cm*x[k] + v[k]
```

The MPC controller also assumes the measurement noise at output `y` is white noise with zero mean and unit variance after scaling. Therefore, the default measurement noise model is `Gmn` shown below:

```
Gmn = ss(1,'Ts',Ts);
```

In this scenario, there are three additive noises in the Kalman filter design: (1) process noise added to the manipulated variable; (2) process noise added to the input of `God`; (3) measurement noise added to the input of `Gmn`:

```
        [ Gd.B(1)  0     0 ]   [ wn1 ]
 w[k] = [ Gd.B(2)  0     0 ] * [ wn2 ] = B_est * white noise
        [ 0        God.B 0 ]   [ wn3 ]
 v[k] = [ Gd.D God.D Gmn.D ] *   wn4   = D_est * white noise

B_est = [[Gd.B;0] [0;0;God.B] [0;0;0]];
D_est = [Gd.D God.D Gmn.D];
```

Therefore, to obtain the noise covariance matrices `Q`, `R`, and `N`, use the following equations.

```
Q = Expectation{w*ctranspose(w)} = B_est * ctranspose(B_est)
R = Expectation{v*ctranspose(v)} = D_est * ctranspose(D_est)
N = Expectation{w*ctranspose(v)} = B_est * ctranspose(D_est)

Q = B_est*B_est';
R = D_est*D_est';
N = B_est*D_est';
```

You can obtain the gains, `L` and {`M`}, using the `kalman` function.

```
G = eye(3);
H = zeros(1,3);
[~, L, ~, M] = kalman(ss(A,[Bu G],Cm,[D H],Ts),Q,R,N);
```

This default Kalman filter design process occurs automatically when you create an MPC controller object. To obtain the resulting default design, use the `getEstimator` function.

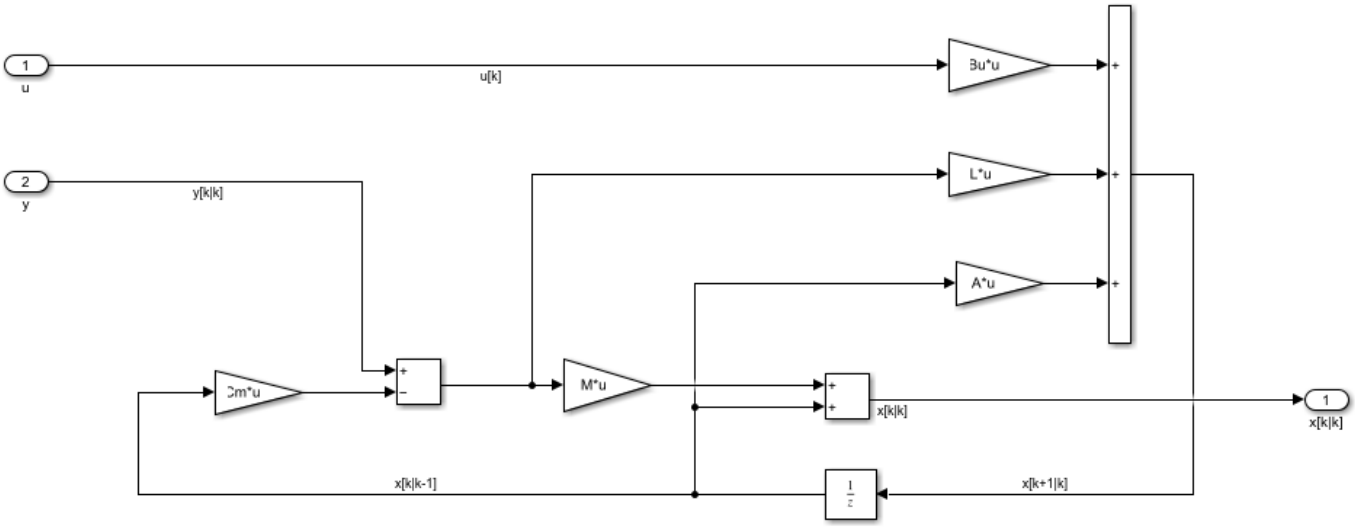
```
[L1,M1,A1,Cm1,Bu1] = getEstimator(mpcobj);
```

These estimator parameters are identical to the ones you previously derived manually.

Implement Equivalent State Estimator in Simulink

At this point, you can build your own Kalman filter in Simulink. One way to build this filter is to use core Simulink blocks based on the `L`, `M`, `A`, `Cm`, and `Bu`.

```
mdl = 'mpc_KalmanFilter';
load_system(mdl)
open_system([mdl '/State Estimator'])
```



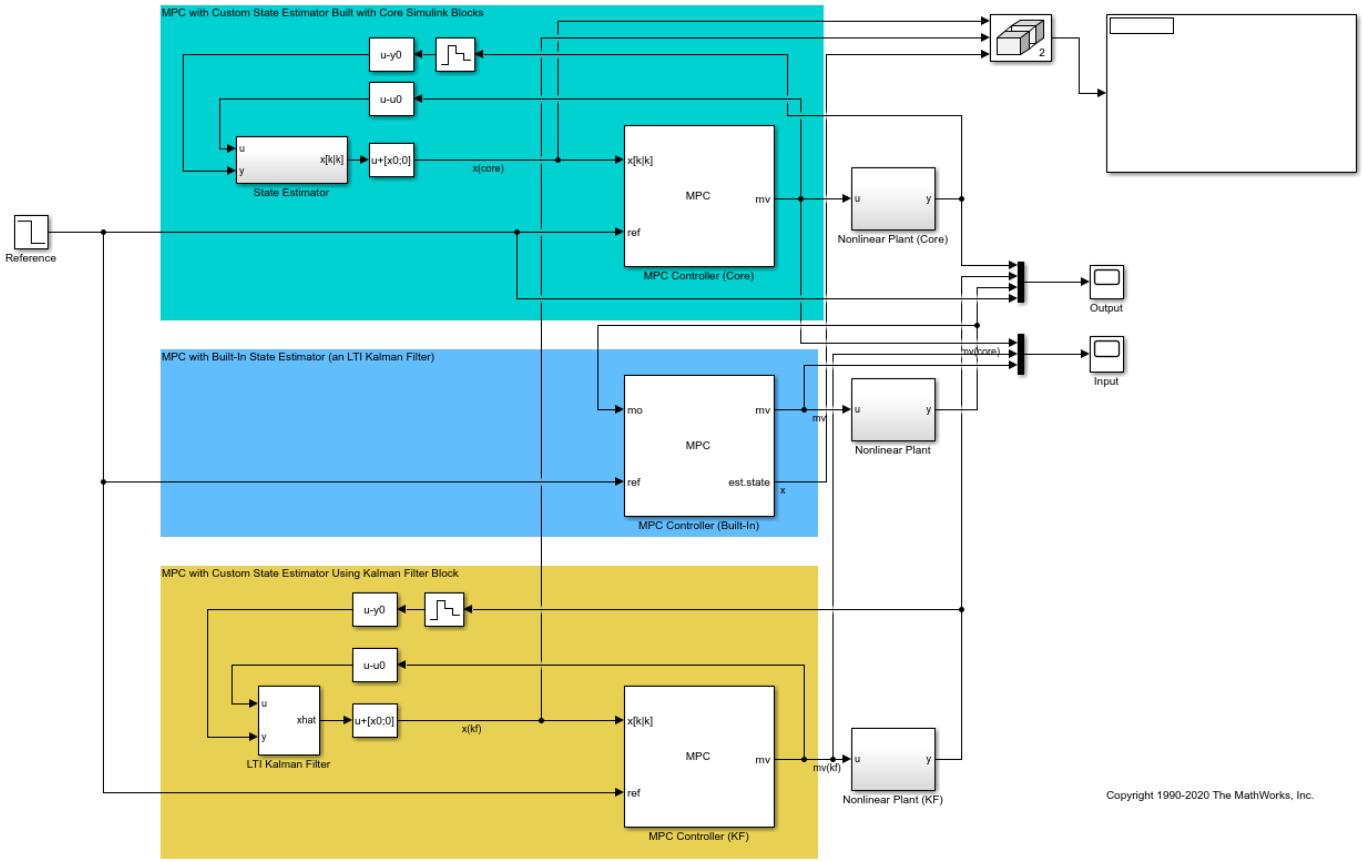
Alternatively, you can use the Kalman Filter block provided by Control System Toolbox software. This block uses the `Gpred` model as the system model as well as the noise covariance matrices `Q`, `R`, and `N`.

Since the Kalman filter is designed at the nominal operating point and uses deviation variables, the nominal values need to be subtracted from the Kalman filter input signals, `u` and `y`, and added to its output signal, `x[k|k]`.

Validate that Custom State Estimation Produces the Same Result

The Simulink model contains three MPC control loops that control the same plant.

```
open_system(mdl);
```



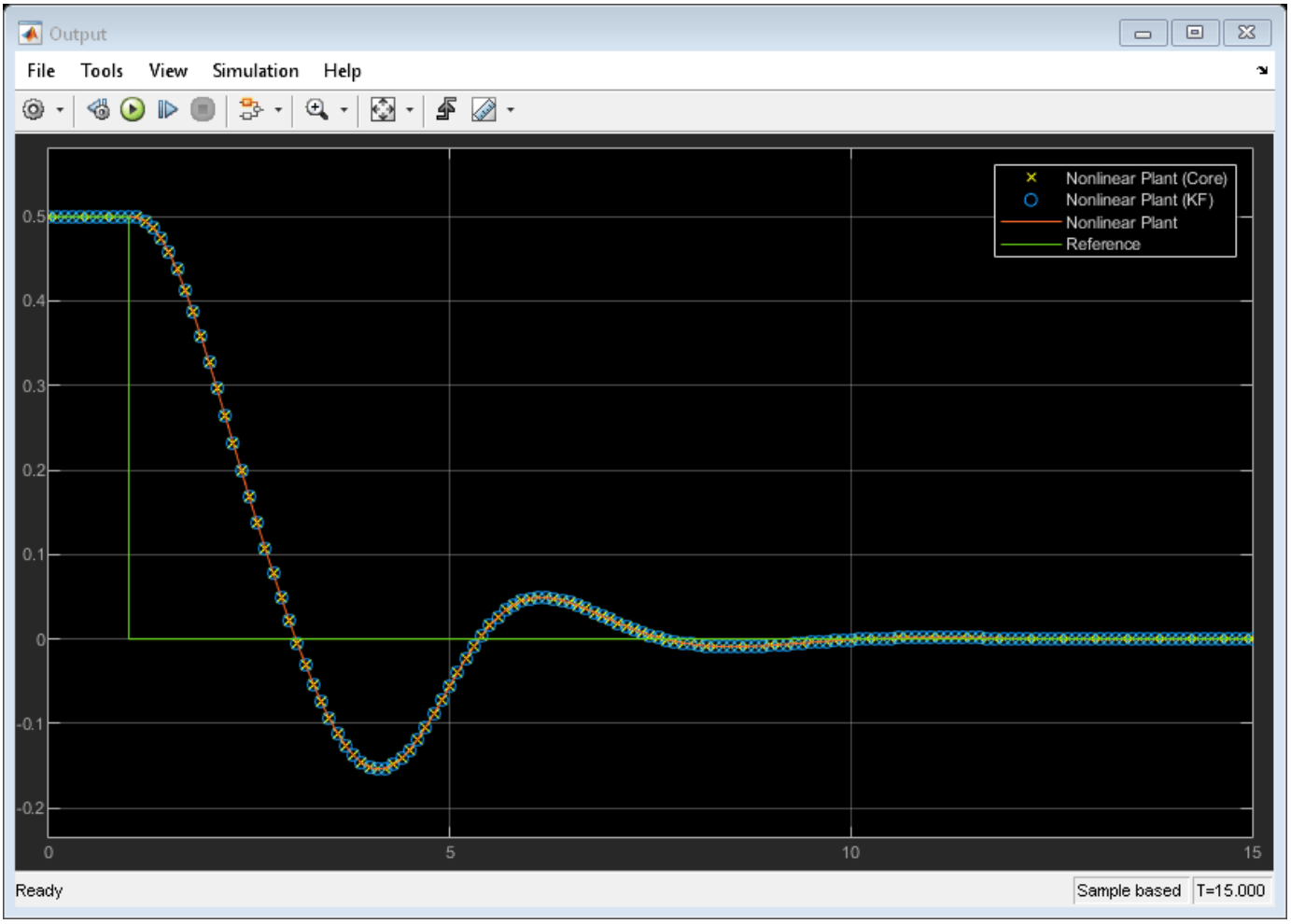
Create a second MPC controller that uses custom state estimation.

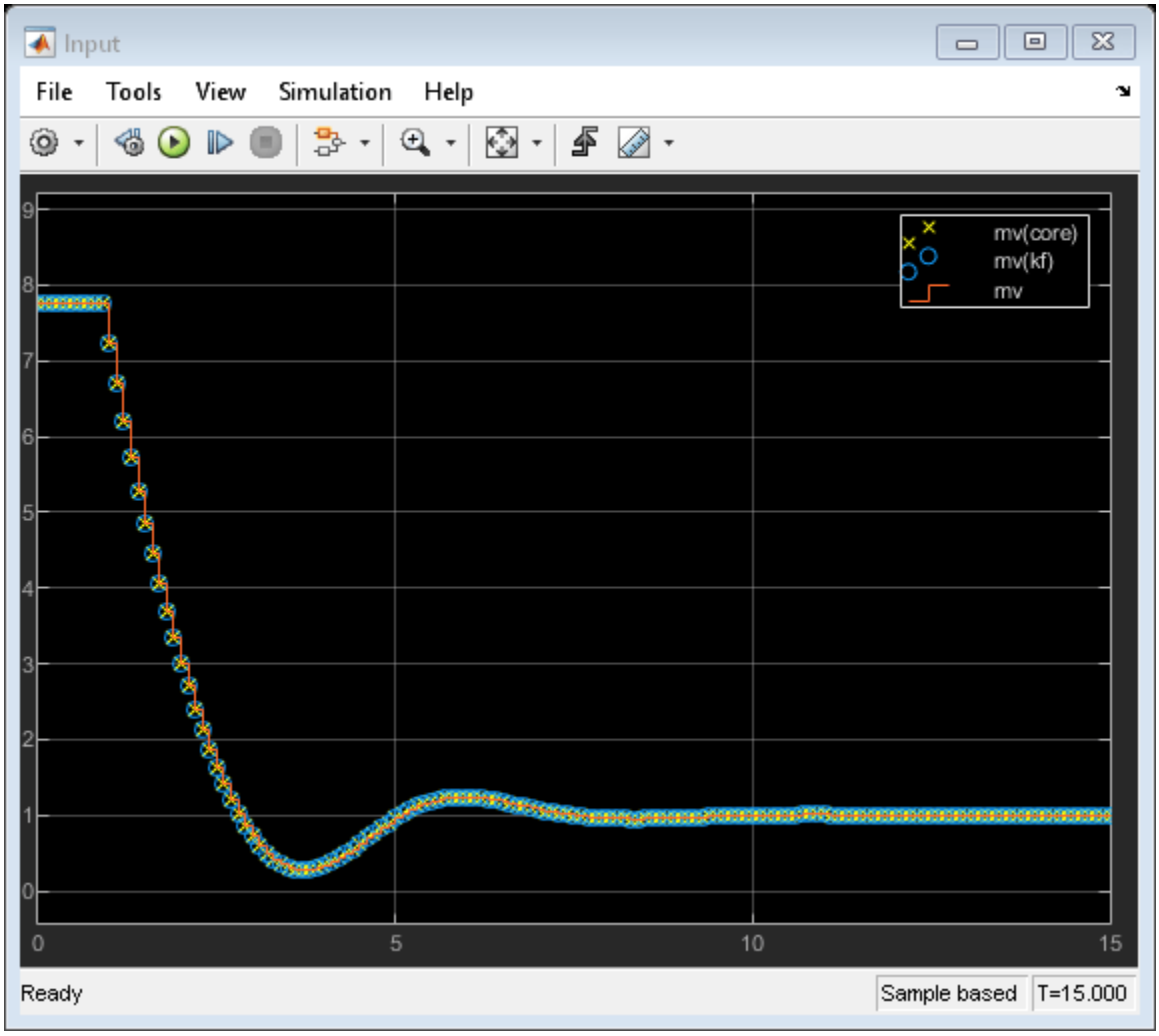
```
mpcobjCSE = mpcobj;
setEstimator(mpcobjCSE,'custom')
```

Simulate the model.

```
sim(mdl);
open_system([mdl '/Output']);
open_system([mdl '/Input']);
```

```
-->Converting model to discrete time.
-->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 measured
```


Output
File
Tools
View
Simulation
Help
Nonlinear Plant (Core)
Nonlinear Plant (KF)
Nonlinear Plant
Reference
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
0
5
10
15
Ready
Sample based
T=15.000



The simulation result shows that the built-in state estimator and custom state estimators produce the same result.

You can apply the state estimator derivation in this example to MIMO plants. In addition, when using adaptive of linear-time-varying (LTV) MPC, the built-in Kalman Filter is LTV where `L`, `M`, and the error covariance matrix `P` become time-varying. Therefore, the custom state estimator will be more complicated.

See Also

`mpc` | `getEstimator` | `setEstimator`

Related Examples

- “Adjust Disturbance and Noise Models” on page 3-25
- “Custom State Estimation” on page 3-32

Manipulated Variable Blocking

Manipulated variable blocking is an alternative to the simpler control horizon concept (see “Choose Sample Time and Horizons” on page 1-2), and it has many of the same benefits. Manipulated variable blocking:

- Provides more tuning flexibility
- Can smooth manipulated variable adjustments
- Can improve controller robustness

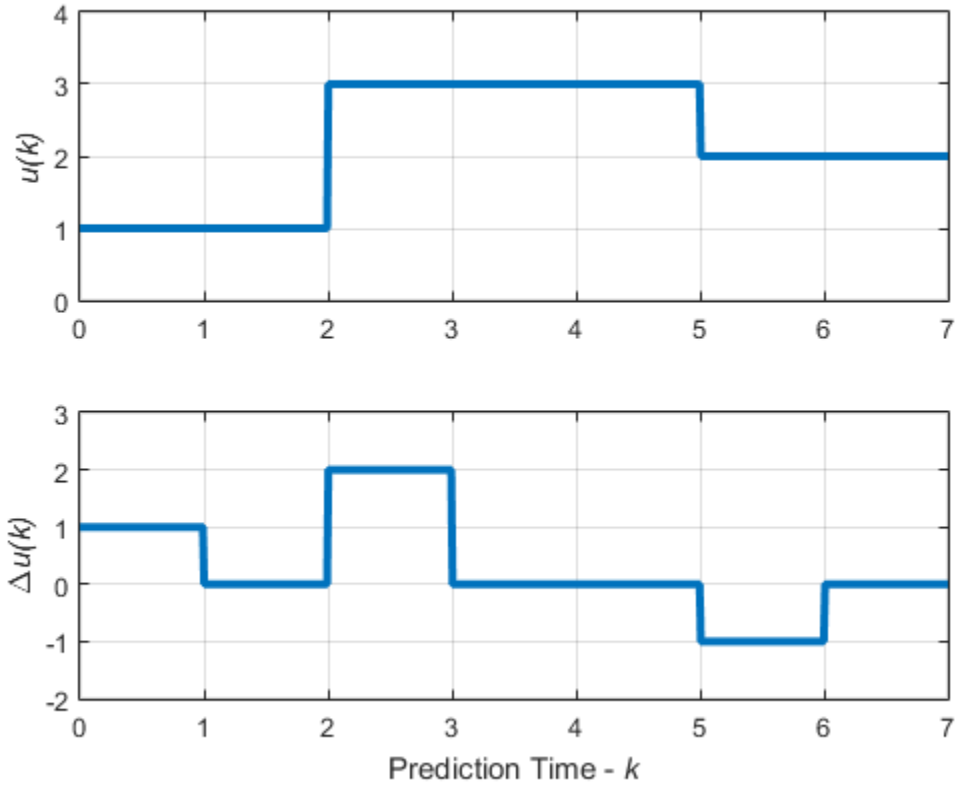
Specify Blocking Interval Lengths

To use manipulated variable blocking, divide the prediction horizon into a series of blocking intervals by specifying your control horizon as a vector of block sizes, [m_1, m_2, ...]. The sum of the block sizes must match the prediction horizon p. If you specify a vector whose sum is:

- Less than the prediction horizon, then the controller adds a blocking interval. The length of this interval is such that the sum of the interval lengths is p. For example, if p=`10` and you specify a control horizon of m=`[1 2 3]`, then the controller uses four intervals with lengths `[1 2 3 4]`.
- Greater than the prediction horizon, then the intervals are truncated until the sum of the interval lengths is equal to p. For example, if p=`10` and you specify a control horizon of `[1 2 3 6 7]`, then the controller uses four intervals with lengths `[1 2 3 4]`.

The controller computes M free moves, where M is the number of blocking intervals. The first free move applies to times k through $k+m_1-1$, the second free move applies from time $k+m_1$ through $k+m_1+m_2-1$, and so on. Here, k is the current control interval.

By default, the controller then holds the manipulated variable constant within each block; that is, the control moves are piecewise constant across each interval. For example, the following figure shows the optimal control moves for a control horizon of m=`[2 3 2]` and prediction horizon of p=`7`.



For each block, the manipulated variable, u, is constant, that is:

- $u(0) = u(1)$
- $u(2) = u(3) = u(4)$
- $u(5) = u(6)$

The recommended approach to blocking is to divide the prediction horizon into 3 to 5 blocks and use one of the following blocking alternatives:

- Equal block sizes (one-fifth to one-third of the prediction horizon, p)
- Block sizes increasing. For example, with p=`20`, you can try three blocks with intervals of length 3, 7, and 10.

To test the effects of different manipulated variable blocking configurations, perform closed-loop simulation tests under the following conditions:

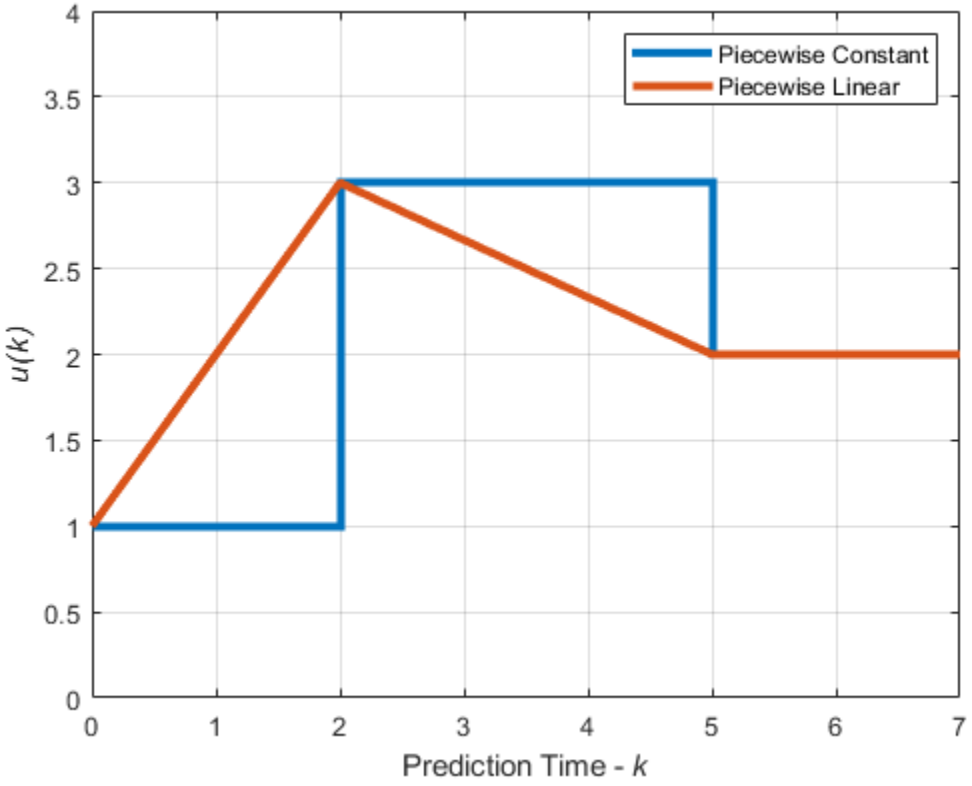
- No constraints
- No prediction error; that is, the controller prediction model should be identical to the plant model

To test each controller for stability and robustness issues, use the `review` function.

Interpolate Block Moves for Nonlinear MPC

As with a linear MPC controller, when you use manipulated variable blocking, a nonlinear MPC controller uses piecewise constant blocking intervals by default. This approach is often too restrictive for optimal path planning applications. To produce a less-restrictive, better-conditioned nonlinear programming problem, you can specify piecewise linear manipulated variable blocking intervals. To do so, set the `Optimization.MVInterpolationOrder` property of your `nlmpc` controller object to `1`.

The following figure shows the optimal control moves for control horizon m=`[2 3 2]` and prediction horizon p=`7`.



In the default piecewise constant case, the computed manipulated variable values of `1`, `3`, and `2` are constant over their respective blocking intervals.

In the piecewise linear case, the computed manipulated variable values are linearly interpolated for the first two blocking intervals and held constant for the final interval.

For more information on nonlinear MPC controllers, see “Nonlinear MPC” on page 9-2.

Note Linear interpolation of blocking moves is not supported for implicit, adaptive, or gain-scheduled MPC controllers.

See Also

`mpc` | `nlmpc`

More About

- “Optimization Problem” on page 2-7
- “Design MPC Controller for Plant with Delays” on page 1-52
- “Choose Sample Time and Horizons” on page 1-2
- “Design MPC Controller at the Command Line”
- “Design Controller Using MPC Designer”

Specifying Alternative Cost Function with Off-Diagonal Weight Matrices

This example shows how to use non-diagonal output 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 an MPC controller with prediction and control horizon of `20` and `2` steps, respectively.

```
mpcobj = mpc(plant,Ts,20,2);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Define constraints on the manipulated variables and their rates.

```
mpcobj.MV = struct('Min',{-3;-2},'Max',{3;2},'RateMin',{-100;-100},'RateMax',{100;100});
```

Specify non-diagonal output weights

To define non-diagonal output weights, you must select the alternative cost function instead of the standard cost function. The alternative cost function allows off-diagonal weighting, but requires the weights to be identical at each prediction horizon step. For more information on these cost function see “Optimization Problem” on page 2-7. To select the alternative cost function, you must specify the weight matrices in cell arrays. For more information, see the section on weights in `mpc`. Specify non-diagonal output weight, corresponding to ((y1-r1)-(y2-r2))^2.

```
OW = [1 -1]'*[1 -1];
mpcobj.Weights.OutputVariables = {OW};
```

Specify non-diagonal input weights

Non-diagonal input weight, corresponding to (u1-u2)^2.

```
mpcobj.Weights.ManipulatedVariables = {0.5*OW};
```

Simulate Using SIM Command

Specify simulation time and reference signal.

```
Tstop = 30;               % simulation time
Tf = round(Tstop/Ts);     % number of simulation steps
r = ones(Tf,1)*[1 2];     % reference trajectory
```

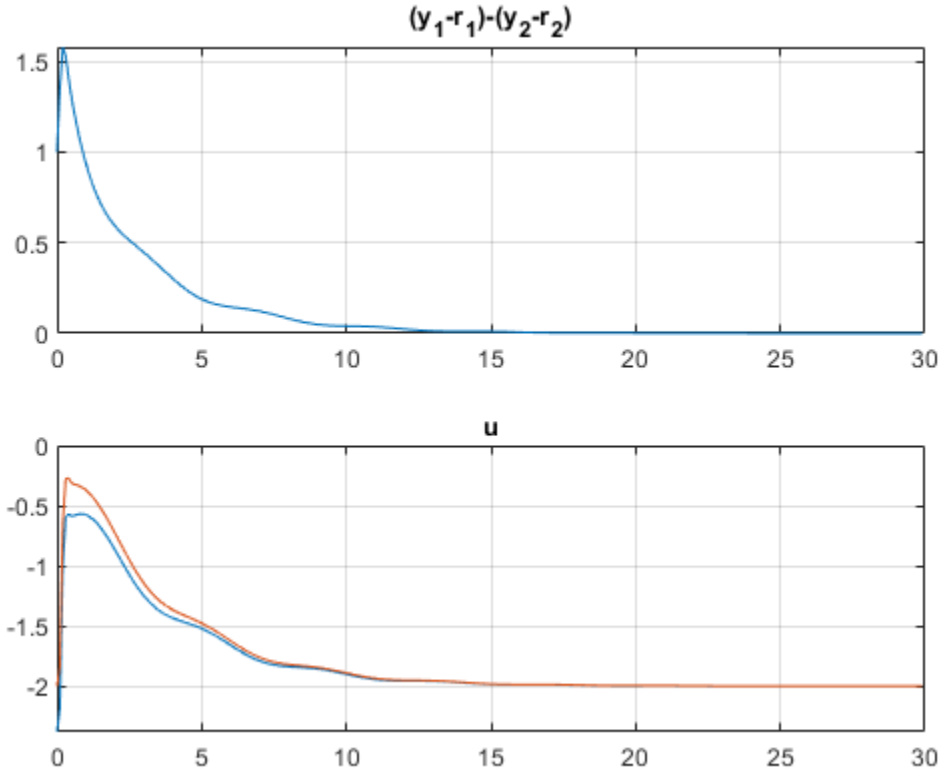
Run the closed-loop simulation.

```
[y,t,u] = sim(mpcobj,Tf,r);
```

```
-->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 measured
```

Plot the results.

```
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');
```

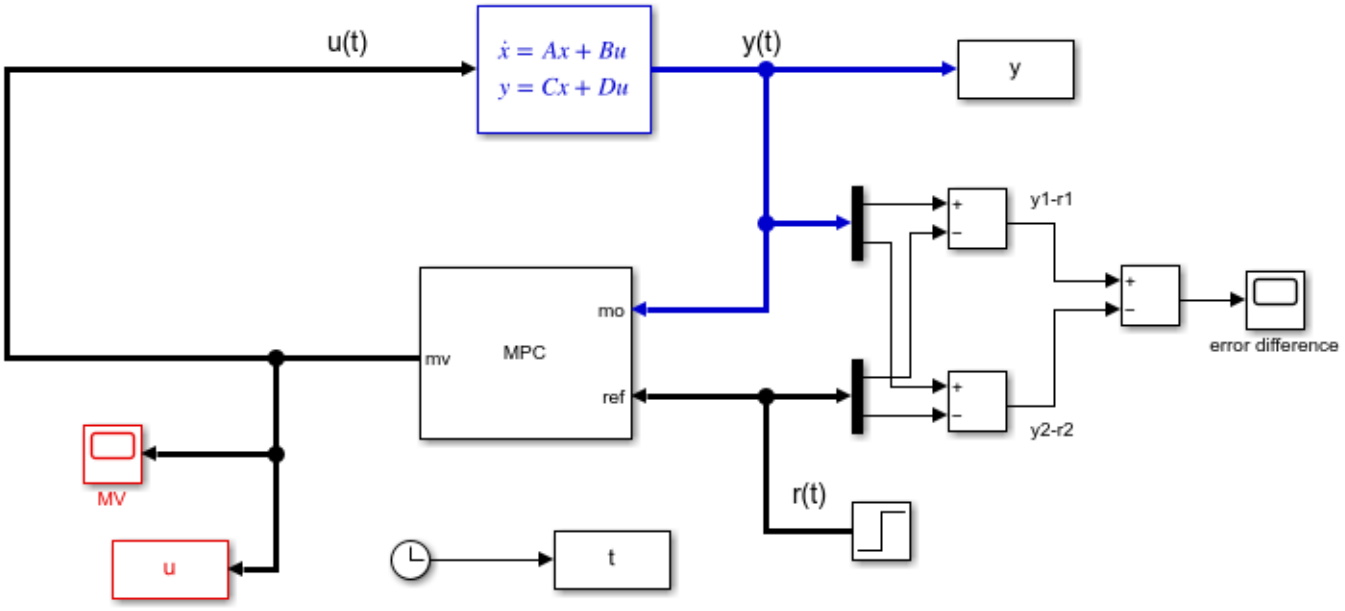


The difference between the two manipulated variable errors tends to zero.

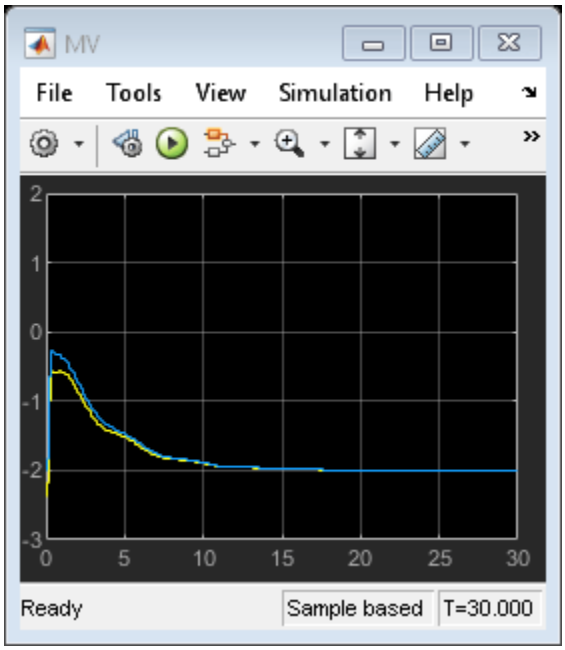
Simulate Using Simulink®

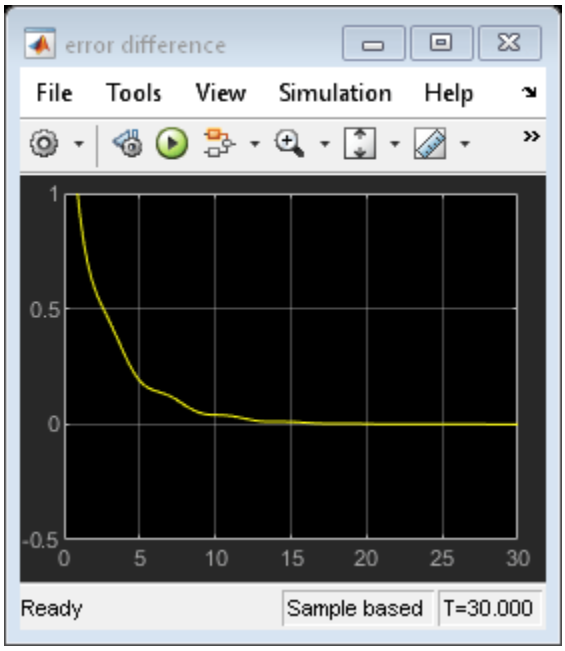
Now simulate closed-loop MPC in Simulink. As expected, results are identical.

```
mdl = 'mpc_weightsdemo';
open_system(mdl);
sim(mdl)
```



Copyright 1990-2012 The MathWorks, Inc.





Close the simulink model.

```
bdclose(mdl);
```

See Also

`mpc` | MPC Controller

More About

- “Optimization Problem” on page 2-7

Controller Analysis

- “Review Model Predictive Controller for Stability and Robustness Issues” on page 4-2
- “Compute Steady-State Gain” on page 4-17
- “Extract Controller” on page 4-20
- “Compare Multiple Controller Responses Using MPC Designer” on page 4-22
- “Adjust Input and Output Weights Based on Sensitivity Analysis” on page 4-31
- “Understanding Control Behavior by Examining Optimal Control Sequence” on page 4-37

Review Model Predictive Controller for Stability and Robustness Issues

You can review your model predictive controller design for potential stability and robustness problems. To do so:

- At the command line, use the `review` function.
- In MPC Designer, on the **Tuning** tab, in the **Analysis** section, click **Review Design**.

In both cases, the software generates a report that shows the results of the following tests:

- **MPC Object Creation** — Test whether the controller specifications generate a valid MPC controller. If the controller is invalid, additional tests are not performed.
- **QP Hessian Matrix Validity** — Test whether the MPC quadratic programming (QP) problem for the controller has a unique solution. You must choose cost function parameters (penalty weights) and horizons such that the QP Hessian matrix is positive-definite.
- **Closed-Loop Internal Stability** — Extract the `A` matrix from the state-space realization of the unconstrained controller, and then calculate its eigenvalues. If the absolute value of each eigenvalue is less than or equal to `1` and the plant is stable, then your feedback system is internally stable.
- **Closed-Loop Nominal Stability** — Extract the `A` matrix from the discrete-time state-space realization of the closed-loop system; that is, the plant and controller connected in a feedback configuration. Then calculate the eigenvalues of `A`. If the absolute value of each eigenvalue is less than or equal to `1`, then the nominal (unconstrained) system is stable.
- **Closed-Loop Steady-State Gains** — Test whether the controller forces all controlled output variables to their targets at steady state in the absence of constraints.
- **Hard MV Constraints** — Test whether the controller has hard constraints on both a manipulated variable and its rate of change, and if so, whether these constraints may conflict at run time.
- **Other Hard Constraints** — Test whether the controller has hard output constraints or hard mixed input/output constraints, and if so, whether these constraints may become impossible to satisfy at run time.
- **Soft Constraints** — Test whether the controller has the proper balance of hard and soft constraints by evaluating the constraint ECR parameters.
- **Memory Size for MPC Data** — Estimate the memory size required by the controller at run time.

You can also programmatically assess your controller design using the `review` function. In this case, the pass/fail testing results are returned as a structure and the testing report is suppressed.

The following example shows how to review your controller design at the command line and address potential design issues.

Plant Model

The application in this example 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 six manipulated variables are the feed gas flow rates.

The plant inputs are:

1. Natural gas (NG)
2. Reformed gas (RG)
3. Hydrogen (H2)
4. Nitrogen (N2)
5. Tail gas 1 (T1)
6. Tail gas 2 (T2)

The plant outputs are:

1. High heating value (HHV)
2. Wobbe index (WI)
3. Flame speed index (FSI)
4. Header pressure (P)

For more information on the fuel gas blending problem, see [1].

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 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'};
```

To reflect sensor delays, add transport delays to the plant outputs.

```
Plant.OutputDelay = [0.00556  0.0167  0.00556  0];
```

Initial Controller Design

Construct an initial model predictive controller based on the design requirements. First, for clarity, disable MPC command-window messages.

```
MPC_verbosity = mpcverbosity('off');
```

Create a controller with a:

- Sample time, `Ts`, of `20` seconds, specified in hours, which corresponds to the sample time of the sensors.
- Prediction horizon, `p`, of `39` control intervals, which is approximately equal to the plant settling time.
- Control horizon, `m`, that uses four blocked moves with lengths of `2`, `6`, `12`, and `19` control intervals.

```
Ts = 20/3600;
p = 39;
m = [2 6 12 19];
Obj = mpc(Plant,Ts,p,m);
```

Specify the output measurement noise and nonzero nominal operating point for the controller.

```
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 for each manipulated variable (MV). Since all the manipulated variables are flow rates of gas streams, their lower bounds are zero. By default, all the MV constraints are hard (`MinECR = 0` and `MaxECR = 0`).

```
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 for the manipulated variable increments. The bounds are set large enough to allow full range of movement in one interval. By default, all the MV rate constraints are hard (`RateMinECR = 0` and `RateMaxECR = 0`).

```
for i = 1:6
    Obj.MV(i).RateMin = -MVmax(i);
    Obj.MV(i).RateMax =  MVmax(i);
end
```

Specify lower and upper bounds for each plant output variable (OV). By default, all the OV constraints are soft (`MinECR = 1` and `MaxECR = 1`).

```
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 tuning weights for the manipulated variables. MV weights are specified based on the known costs of each feed stream. Doing so tells MPC controller how to move the six manipulated variables to minimize the cost of the blended fuel gas. The weights are normalized such that the maximum weight is approximately `1.0`.

```
Obj.Weights.MV = [54.9 20.5 0 5.73 0 0]/55;
```

Specify tuning weights for the manipulated variable increments. These weights are small relative to the maximum MV weight so that the MVs are free to vary.

```
Obj.Weights.MVrate = 0.1*ones(1,6);
```

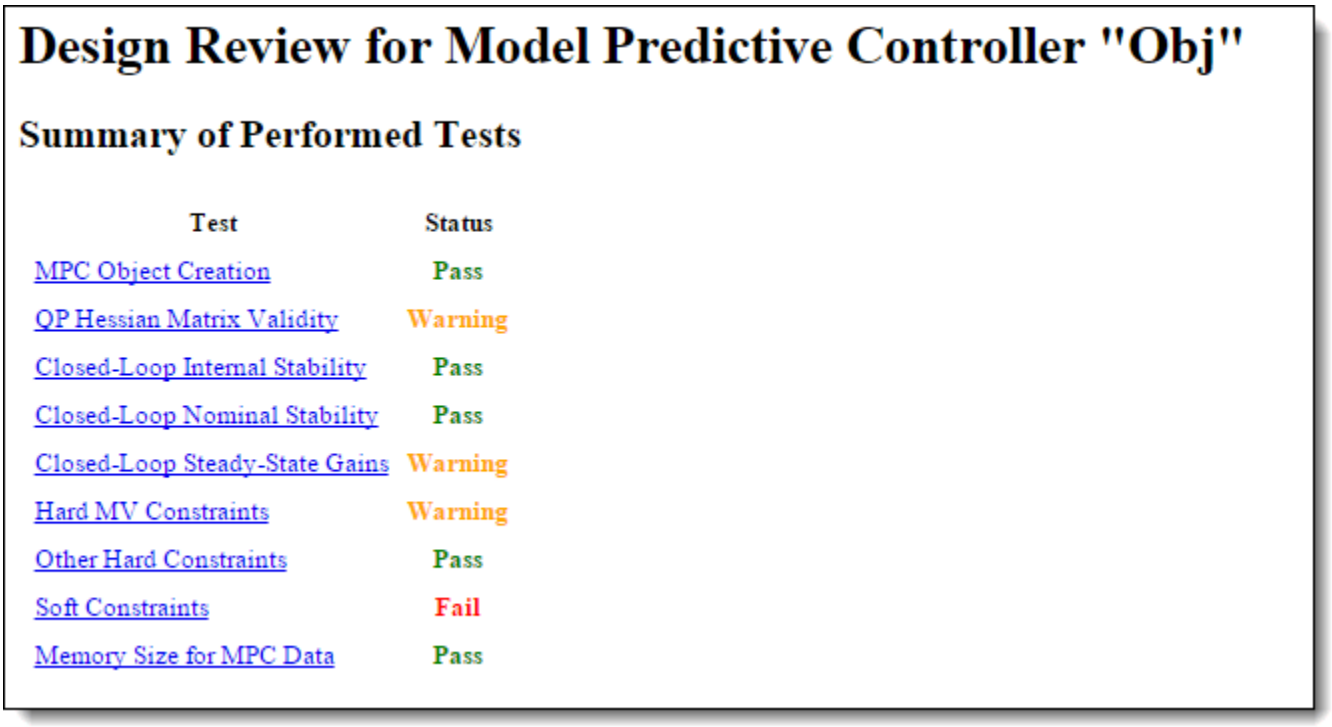
Specify tuning weights for the plant output variables. The OV weights penalize deviations from specified setpoints and would normally be large relative to the other weights. For this example, first consider the default values, which equal the maximum MV weight.

```
Obj.Weights.OV = [1,1,1,1];
```

Improve the Initial Design

Review the initial controller design. The `review` function generates and opens a report in the Web Browser window.

```
review(Obj)
```



Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Closed-Loop 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 review summary lists three warnings and one error. Review the warnings and error in order. Click **QP Hessian Matrix Validity** and scroll down to the warning, which indicates that the plant signal magnitudes differ significantly. Specifically, the pressure response is much larger than the other signals.

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.

The OV spans indicated by the specified OV bounds are quite different, and the pressure span is two orders of magnitude larger than the others. It is good practice to account for the expected differences in signal magnitudes by specifying MPC scale factors. Since the MVs are already weighted based on relative cost, specify scale factors only for the OVs.

Calculate OV spans.

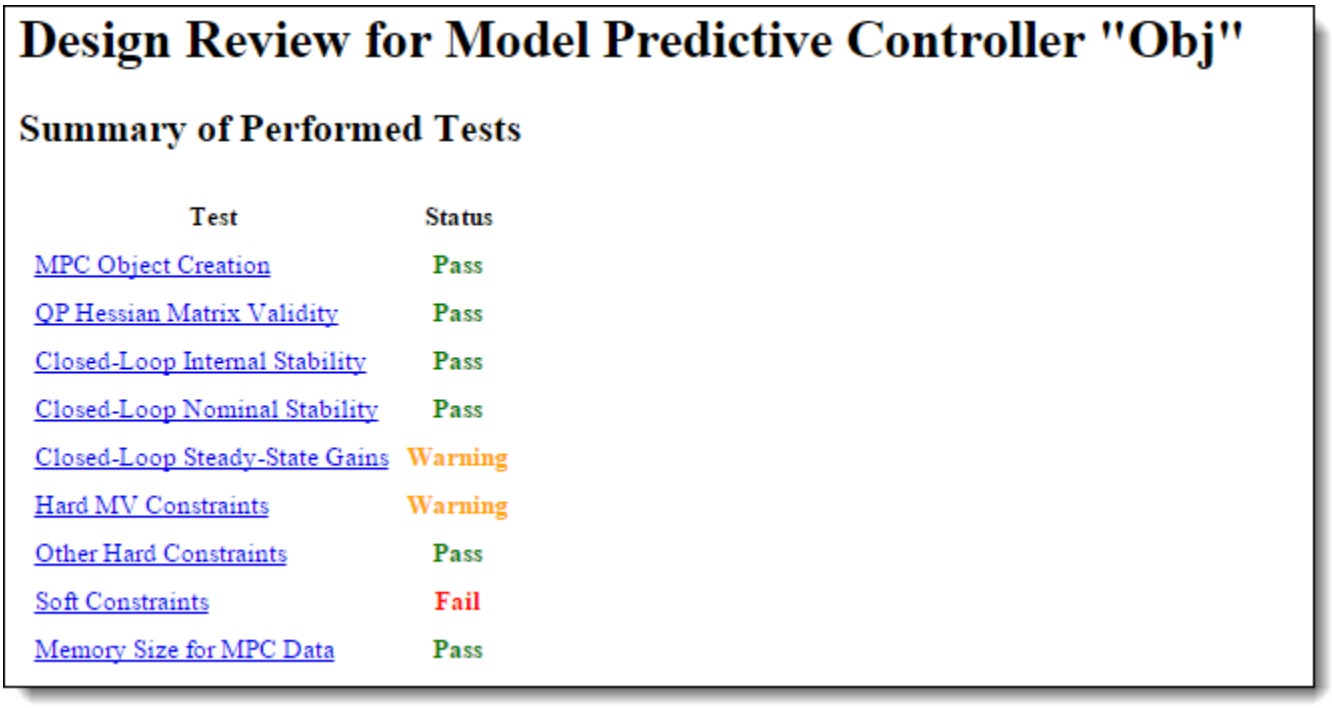
```
OVspan = OVmax - OVmin;
```

Use these spans as scale factors.

```
for i = 1:4
    Obj.OV(i).ScaleFactor = OVspan(i);
end
```

To verify that setting output scale factors fixes the warning, review the updated controller design.

```
review(Obj)
```



Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Closed-Loop 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 next warning indicates that the controller does not drive the OVs to their targets at steady state. To see a list of the nonzero gains, click **Closed-Loop Steady-State 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	0.0468766
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` output to deviate about 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, or offset, of about -0.0345 in `HHV`, and so on.

Since there are six MVs and only four OVs, excess degrees of freedom are available. Therefore, you might expect the controller to have no steady-state offsets. However, the specified nonzero MV weights, which were selected to drive the plant toward the most economical operating condition, are causing nonzero steady-state offsets.

Nonzero steady-state offsets are often undesirable but are acceptable in this application because:

1. The primary objective is to minimize the blend cost. The gas quality (`HHV`, and so on) can vary freely within the specified OV limits.
2. The small offset gain magnitudes indicate that the impact of disturbances is small.
3. The OV limits are soft constraints. Small, short-term violations are acceptable.

View the second warning by clicking **Hard MV Constraints**. This warning indicates a potential conflict in hard constraints.

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

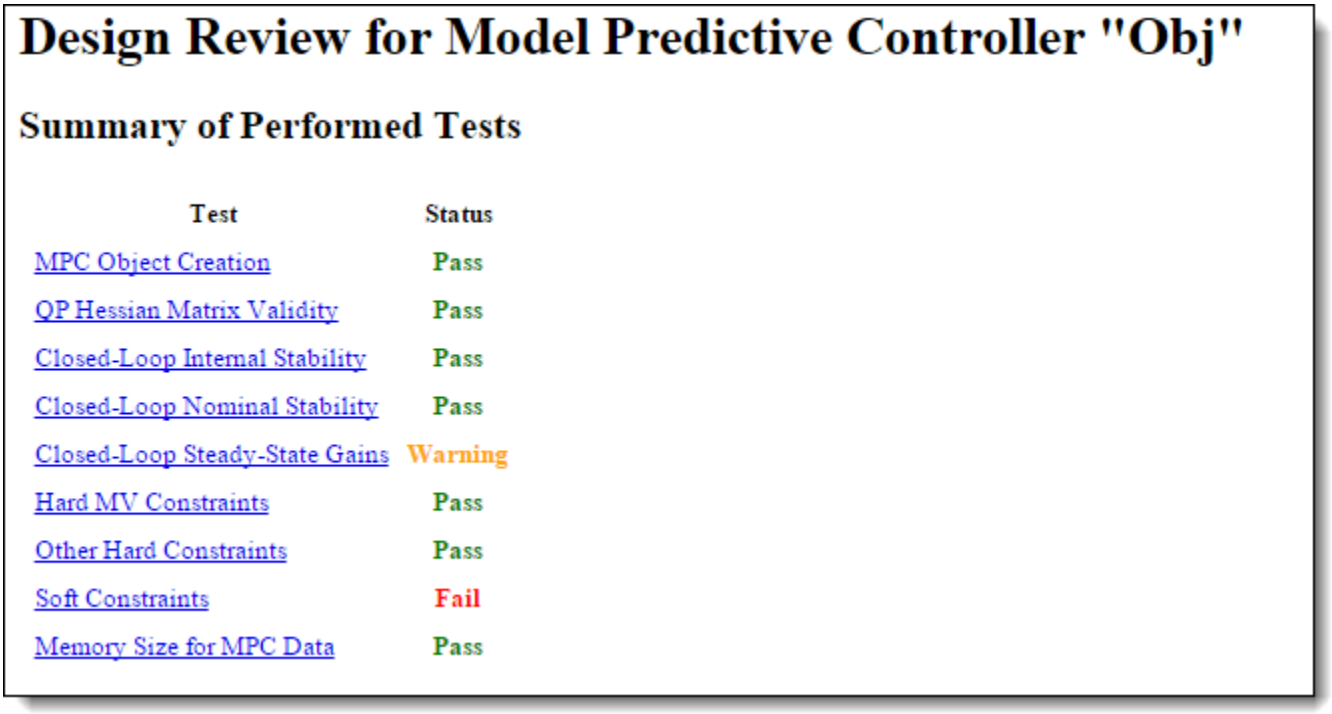
If an external event causes `NG` to go far below its specified minimum, the constraint on its rate of increase might make it impossible to return the `NG` within bounds in one control interval. In other words, if you specify both `MV.Min` and `MV.RateMax`, the controller would not be able to find an 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 unlikely in the gas blending application. However, it is 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, soften the increment bounds.

```
for i = 1:6
    Obj.MV(i).RateMinECR = 0.1;
    Obj.MV(i).RateMaxECR = 0.1;
end
```

Review the updated controller design.

```
review(Obj)
```



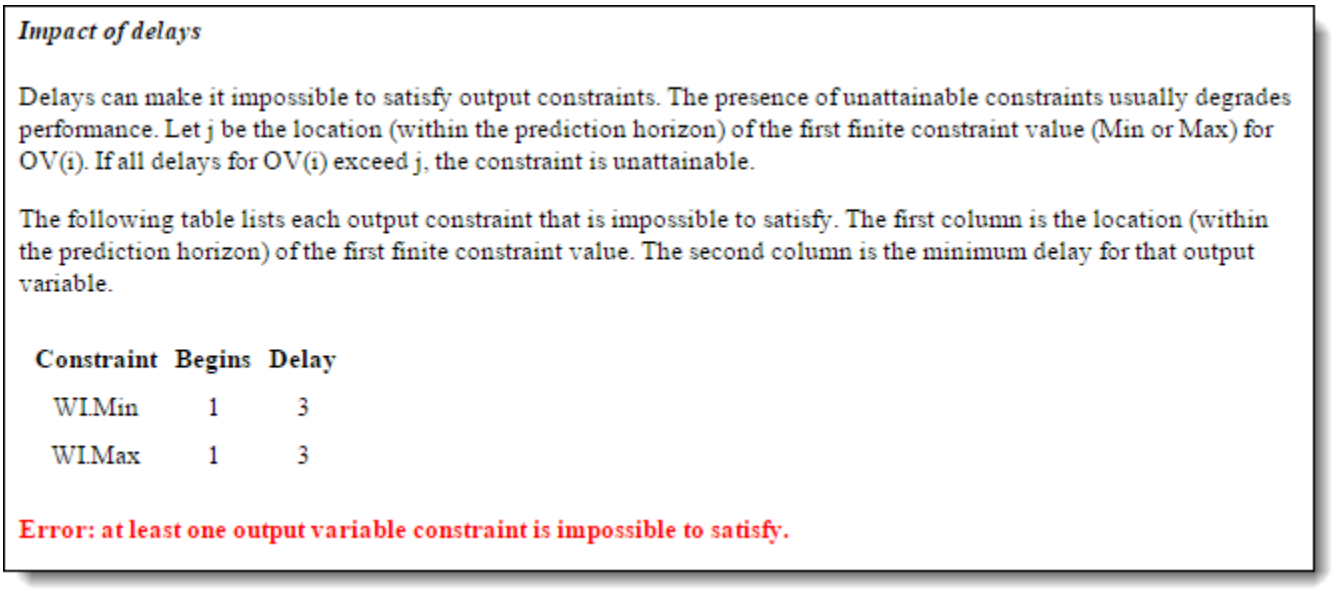
Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Closed-Loop 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 is fixed.

To view the error message, click **Soft Constraints**.



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.

The delay in the `WI` output makes it impossible to satisfy bounds on that variable within the first three control intervals. The `WI` bounds are soft, but it is poor practice to include unattainable constraints in a design. Therefore, modify the `WI` bound specifications such that it is unconstrained until the fourth prediction horizon step.

```
Obj.OV(2).Min = [-Inf(1,3) OVmin(2)];
Obj.OV(2).Max = [ Inf(1,3) OVmax(2)];
```

Rerunning the review command verifies that this change eliminates the error message, as shown in the next step.

Assess Impact of Zero Output Weights

Given that the design requirements allow the OVs to vary freely within their limits, consider removing their penalty weights.

```
Obj.Weights.OV = zeros(1,4);
```

Review the impact of this design change.

```
review(Obj)
```

Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Warning
Closed-Loop 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

There is a new warning regarding the QP Hessian matrix validity. To see the warning details, click **QP Hessian Matrix Validity**.

Penalty Weights On Output Variables

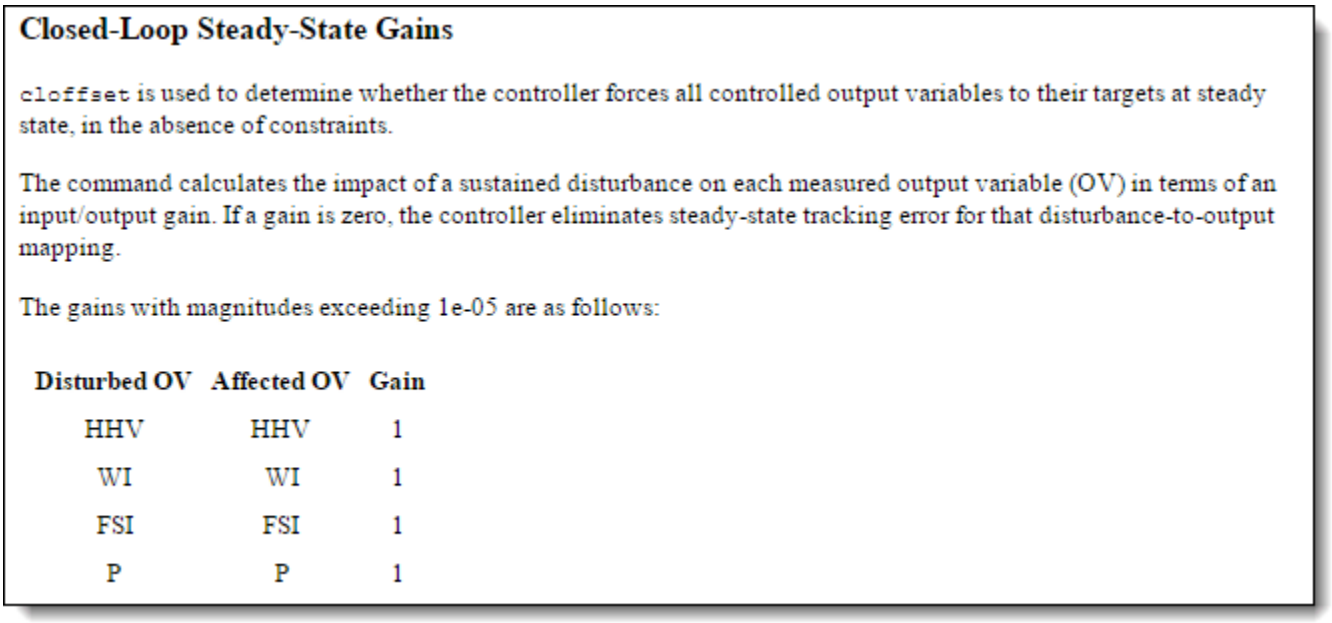
Your output variable (OV) penalty weights also affect the Hessian. Non-zero values emphasize the importance of OV target tracking, making a unique QP solution more likely.

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 flags the zero weights on all four output variables. Since the zero weights are consistent with the design requirements and the other Hessian tests indicate that the quadratic programming problem has a unique solution, this warning can be ignored. To see the second new warning, click **Closed-Loop Steady-State Gains**. The warning shows another consequence of setting the four OV weights to zero. When an OV is not penalized by a weight, the controller ignores any output disturbance added to the OV and passes the disturbance 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, nonzero steady-state offsets are acceptable as long as the controller is able to hold all the OVs within their specified bounds. Therefore, it is a good idea to examine how easily the soft OV constraints can be violated when disturbances are present.

Review Soft Constraints

To see a list of soft constraints, click **Soft Constraints**. In this example, the soft constraints are the 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.

Finding: Sensitivity ratios are acceptable.

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. To make the priorities more comparable, increase the pressure constraint ECR values, and adjust the others as well. For example:

```
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

In the **Sensitivity Ratio** column, all the sensitivity ratios are now less than unity, which means that the soft constraints receive less attention than other terms in the MPC objective function, such as deviations of the MVs from their target values. Therefore, it is likely that an output constraint violation would occur.

To give the output constraints higher priority than other MPC objectives, increase the `Weights.ECR` parameter from the default, `1e5`, to a higher value, which hardens all the soft OV constraints.

```
Obj.Weights.ECR = 1e8;
```

Review the impact of this design change.

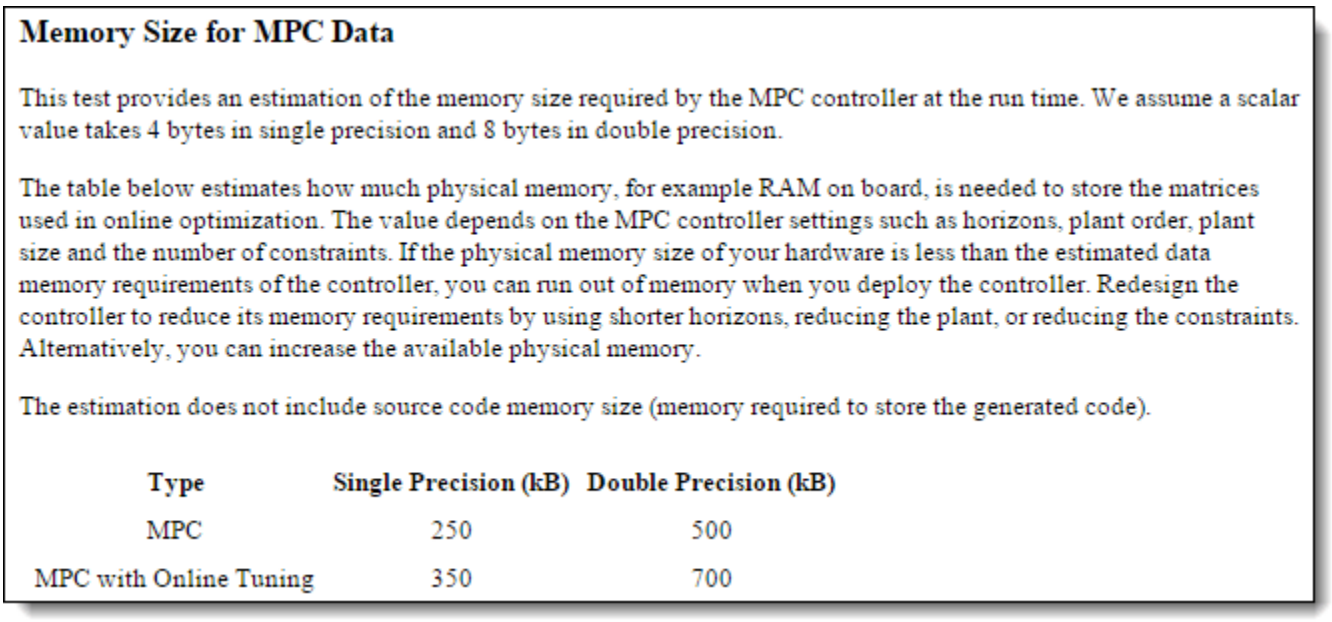
```
review(Obj)
```

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 more sensitive to output constraint violations than to errors in target tracking by a factor of `100`.

Review Data Memory Size

To see the estimated memory size required to store the MPC data matrices used on hardware, click **Memory Size for MPC Data**.



Memory Size for MPC Data

This test provides an estimation of the memory size required by the MPC controller at the run time. We assume a scalar value takes 4 bytes in single precision and 8 bytes in double precision.

The table below estimates how much physical memory, for example RAM on board, is needed to store the matrices used in online optimization. The value depends on the MPC controller settings such as horizons, plant order, plant size and the number of constraints. If the physical memory size of your hardware is less than the estimated data memory requirements of the controller, you can run out of memory when you deploy the controller. Redesign the controller to reduce its memory requirements by using shorter horizons, reducing the plant, or reducing the constraints. Alternatively, you can increase the available physical memory.

The estimation does not include source code memory size (memory required to store the generated code).

Type	Single Precision (kB)	Double Precision (kB)
MPC	250	500
MPC with Online Tuning	350	700

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, redesign the controller to reduce its memory requirements. Alternatively, increase the memory available on the target system.

Restore the MPC verbosity level.

```
mpcverbosity(MPC_verbosity);
```

References

[1] Muller C. J., I. K. Craig, and N. L. Ricker. "Modeling, validation, and control of an industrial fuel gas blending system." *Journal of Process Control*. Vol. 21, Number 6, 2011, pp. 852-860.

See Also

`review`

Compute Steady-State Gain

This example shows how to analyze the steady-state performance of a model predictive controller using `cloffset`. This command assumes that no constraint is active, and calculates the steady-state output sensitivity matrix of the closed loop, which is the DC gain from plant output disturbances (using a sustained 1-unit disturbance step) to controlled plant outputs. When this matrix is zero the controller is able to reject constant output disturbances and track constant setpoint with zero offsets in steady-state.

Define a state-space plant model with two inputs and two outputs. This plant represents the linearized model of a Continuously Stirred Tank Reactor (CSTR) where the first measured output is the concentration of a key reactant, the second measured output is the temperature in the reactor. For more information see “CSTR 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, with a sampling time of one second.

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

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

As the last output line specifies, the cost function default output weights is `1` for the first output and `0` for the second one:

```
MPCobj.W.OutputVariables
```

```
ans = 1×2

     1     0
```

The software automatically adds an integrator as output disturbance model for each measured output, in order of decreasing output weight, unless this causes the plant state to become unobservable. For this plant, only an integrator on the first output is added:

```
getoutdist(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 noise.
   Assuming no disturbance added to measured output channel #2.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

```
ans =

  A =
       x1
   x1   1

  B =
       u1
   x1   1

  C =
        x1
   MO1   1
   MO2   0

  D =
        u1
   MO1   0
   MO2   0

Sample time: 1 seconds
Discrete-time state-space model.
```

Compute the closed-loop, steady-state gain matrix for the closed loop system.

```
DCgain = cloffset(MPCobj)
```

```
DCgain = 2×2

    0.0000   -0.0000
    2.3272    1.0000
```

`DCgain(i,j)` is the closed loop static gain from output disturbance `j` to controlled plant output `i`. The first column of `DCgain` shows that a disturbance applied to the first measured output (key reactant concentration) only affects the second output (reactor temperature). The second column shows that a disturbance applied to the second measured output passes unmitigated through the closed loop and is fully measured at the second plant output. In other words, the closed loop is not able to compensate for a disturbance applied to the second output.

The fact that both entries in the first row of `DCgain` are zeros means that the controller is able to completely reject disturbances that affect this output (and therefore the tracking of any given setpoint reference on this output would be perfect). This happens because the cost function weight for the first output is nonzero, and because the built-in estimator includes the integrator added as disturbance model on the first output.

On the other hand, since the cost function weight for the second output is `0`, (and also because there is no integrator added as a disturbance model on this output) the controller does not try to reject disturbances affecting the second output, as shown by the second row of `DCgain`. This also means that the controller would be unable to track any reference setpoint on this output.

See Also

`cloffset` | `mpc`

More About

- “MPC Prediction Models”

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);
```

```
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

```
MPCobj.W.ManipulatedVariablesRate = 0.3;
MPCobj.W.OutputVariables = [1 0];
```

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 measured
```

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);
```

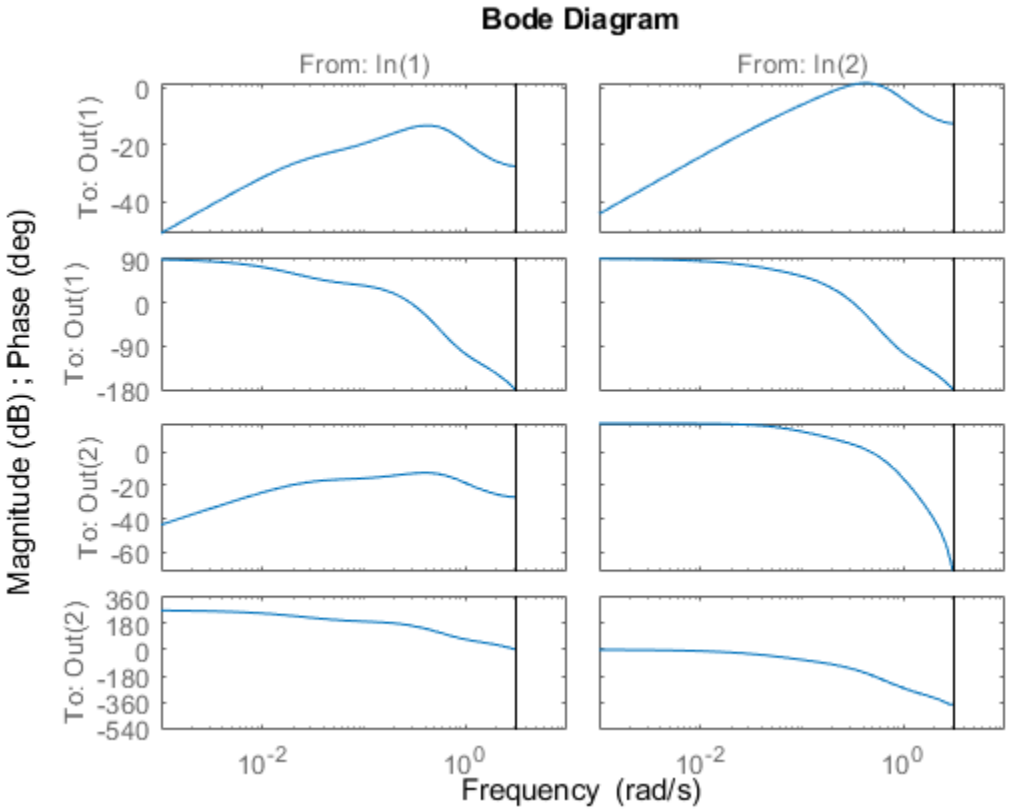
You can then analyze the resulting feedback system. For example, verify that all closed-loop poles are within the unit circle.

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

```
poles = 6×1 complex

   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

`ss` | `mpc` | `feedback`

More About

- “Design MPC Controller at the Command Line”

Compare Multiple Controller Responses Using MPC Designer

This example shows how to compare multiple controller responses using **MPC Designer**. In particular, controllers with different output constraint configurations are compared.

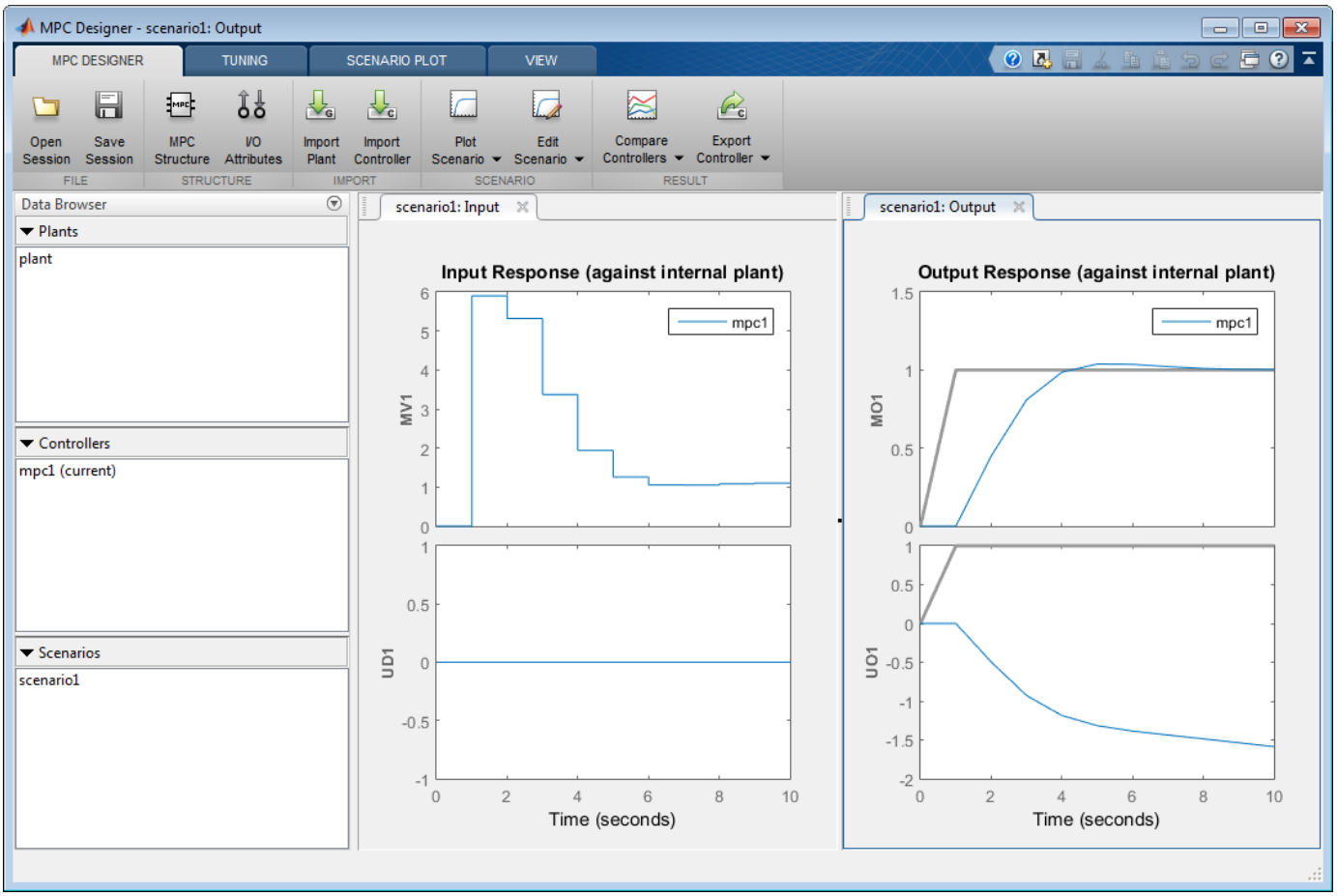
Define Plant Model

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**, 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

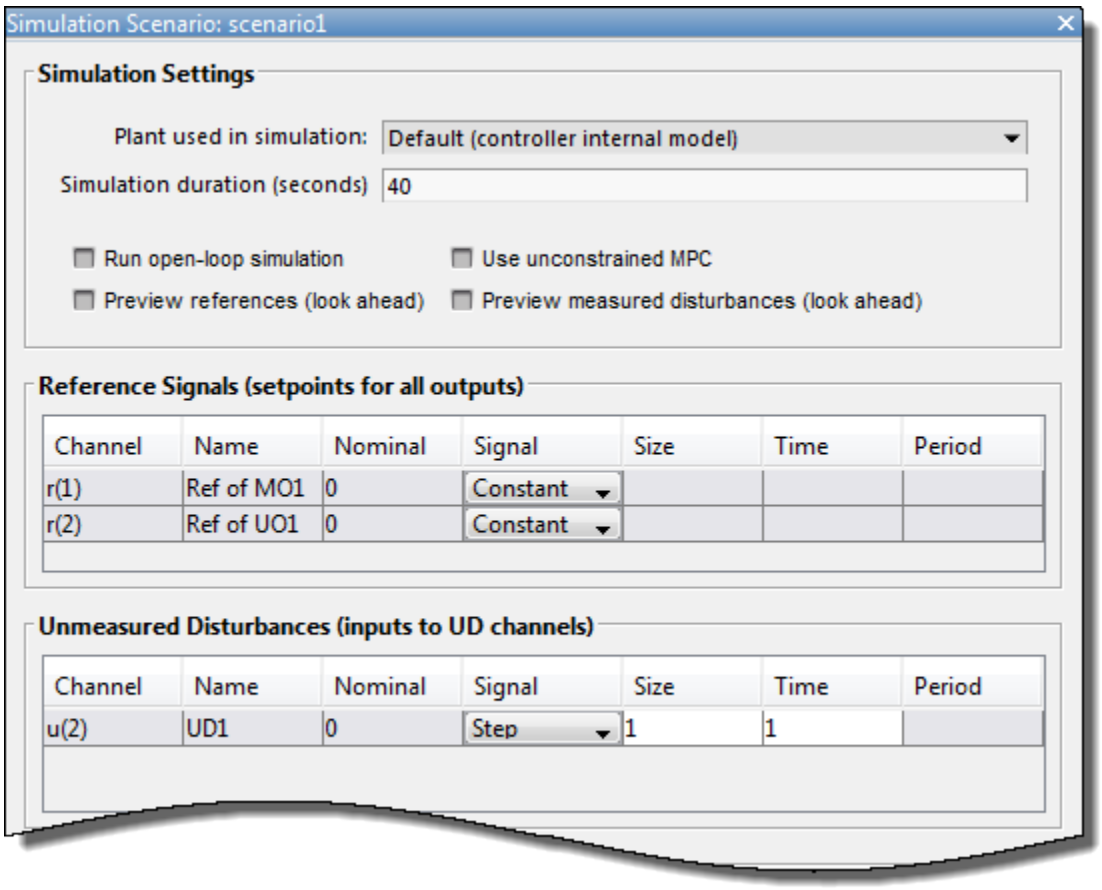
Configure a disturbance rejection simulation scenario.

In **MPC Designer**, 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

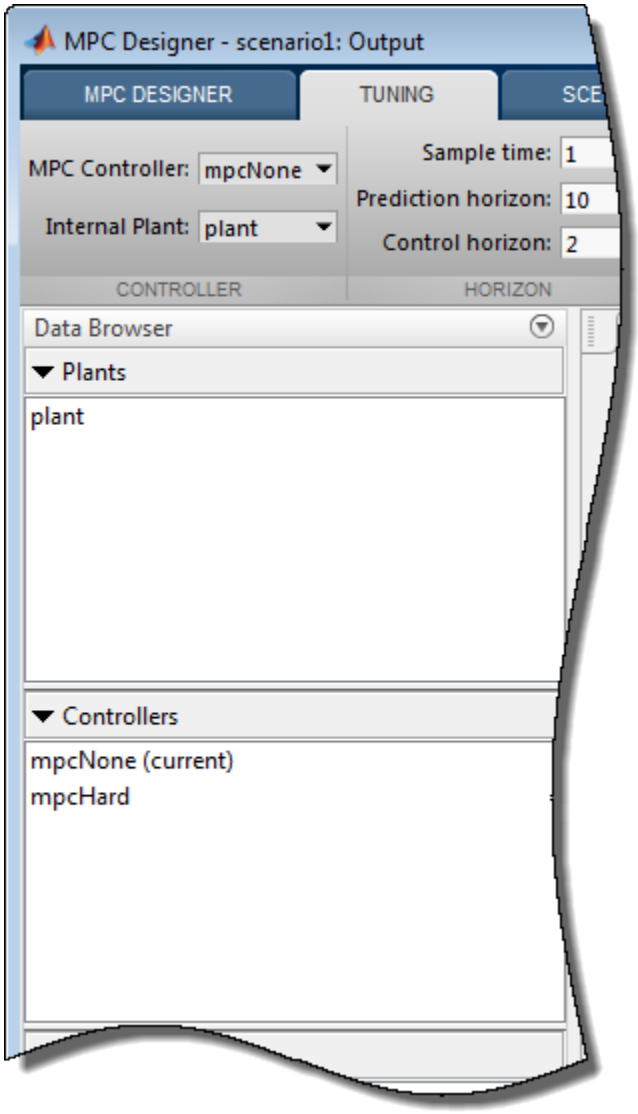
Suppose that the control specifications indicate that such an increase in the unmeasured disturbance is undesirable. To limit the effect of the unmeasured disturbance, create a controller with a hard output constraint.

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**.

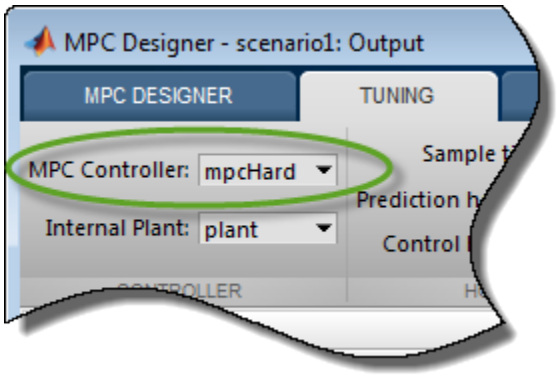
The app creates a copy of the default controller and adds it to the **Data Browser**.

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.

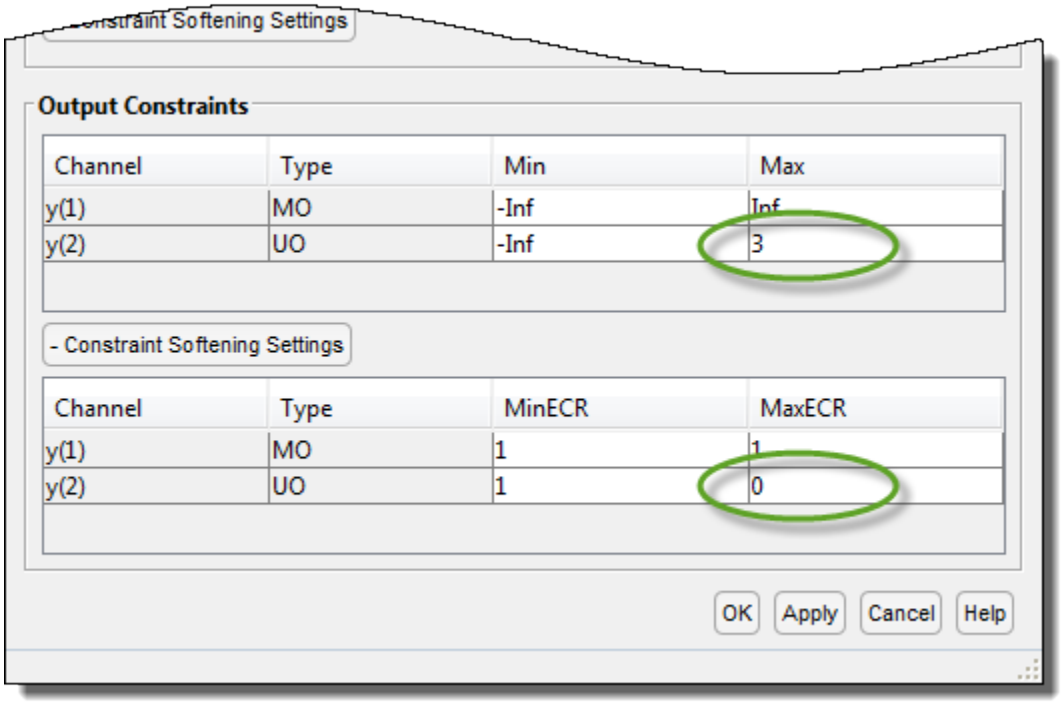


In the **Design** section, click **Constraints**.

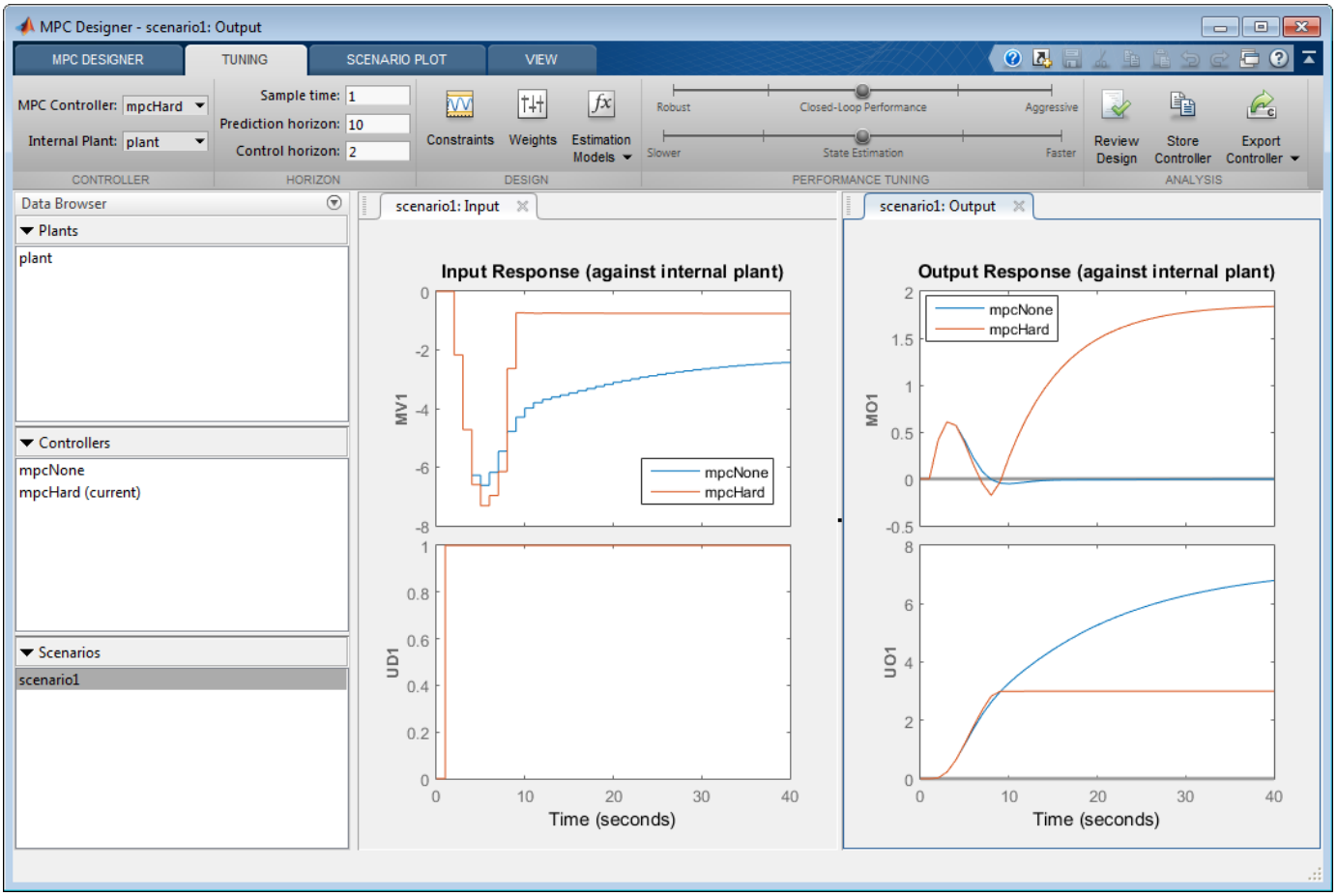
In the Constraints dialog box, in the **Output Constraints** section, in the **Max** column, specify a maximum output constraint of `3` for the unmeasured output (UO).

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**.

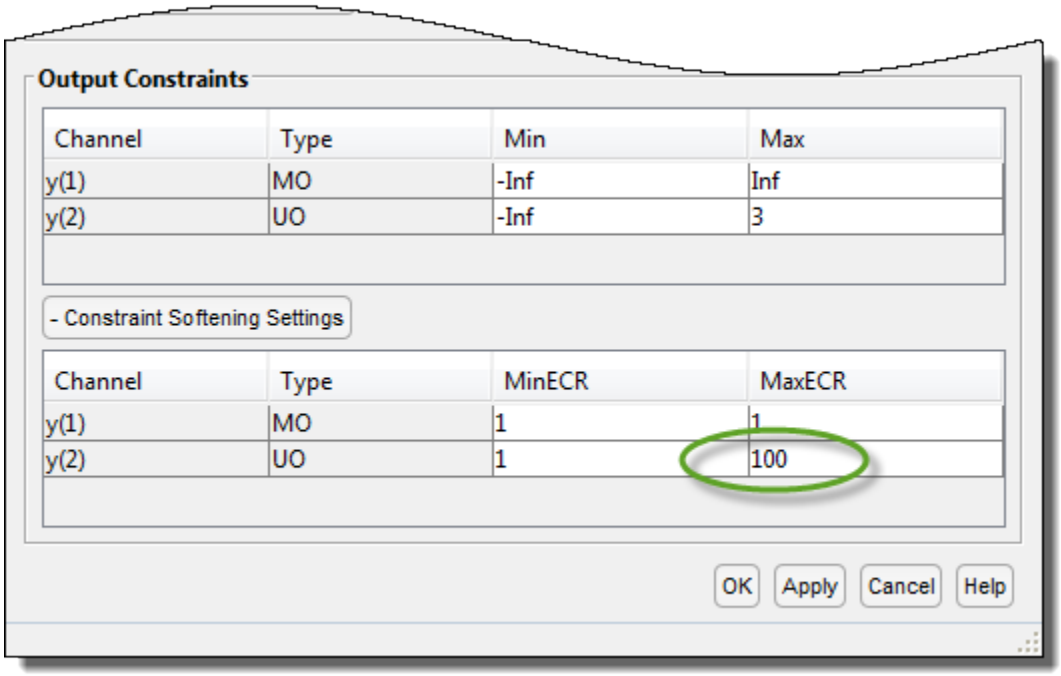
In the **Data Browser**, in the **Controllers** section, rename `mpcHard_Copy` to `mpcSoft`.

On the **Tuning** tab, in the **Controller** section, in the **MPC Controller** drop-down list, select `mpcSoft` as the current controller.

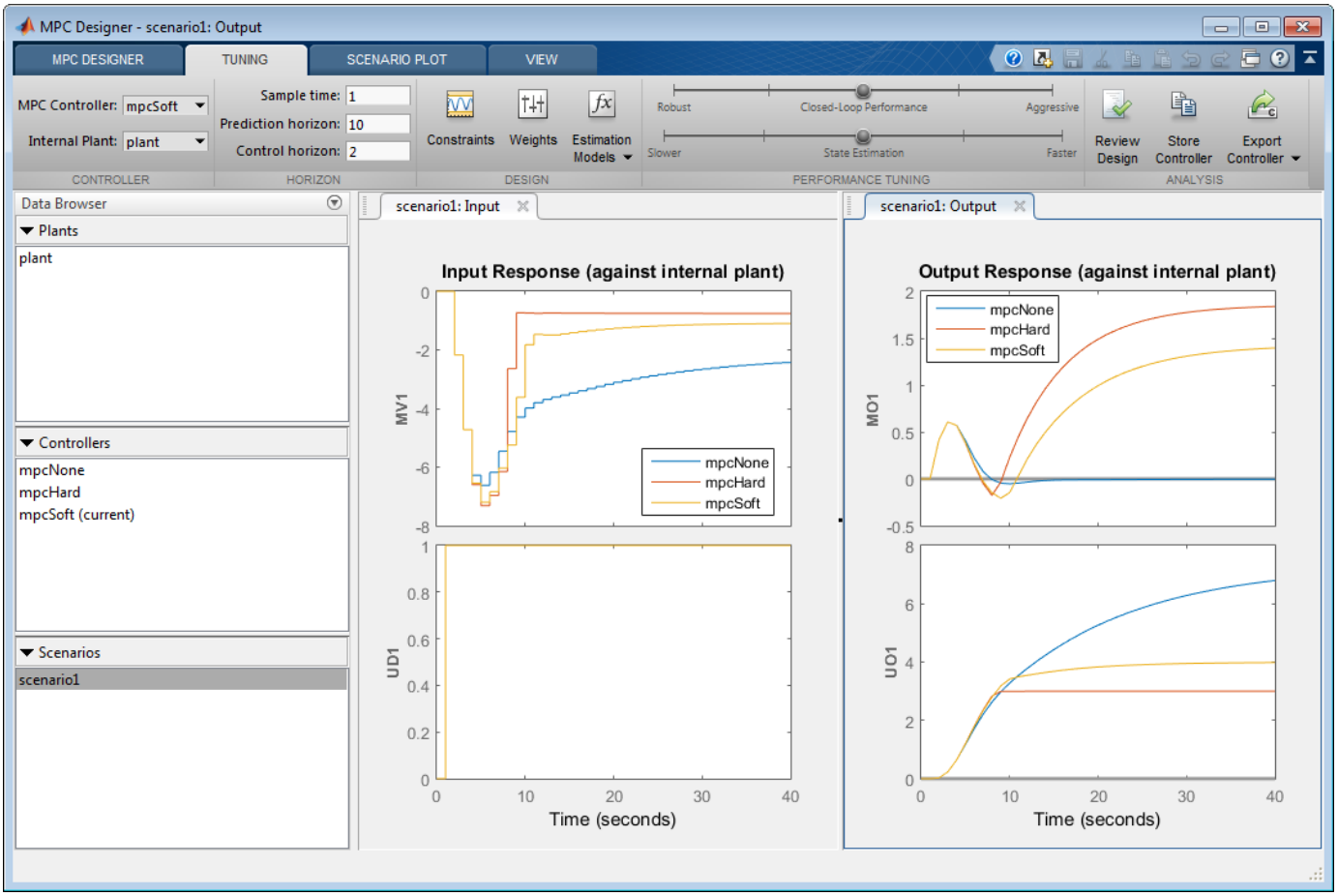
The app adds the `mpcSoft` controller response to the **Input Response** and **Output Response** plots.

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**.

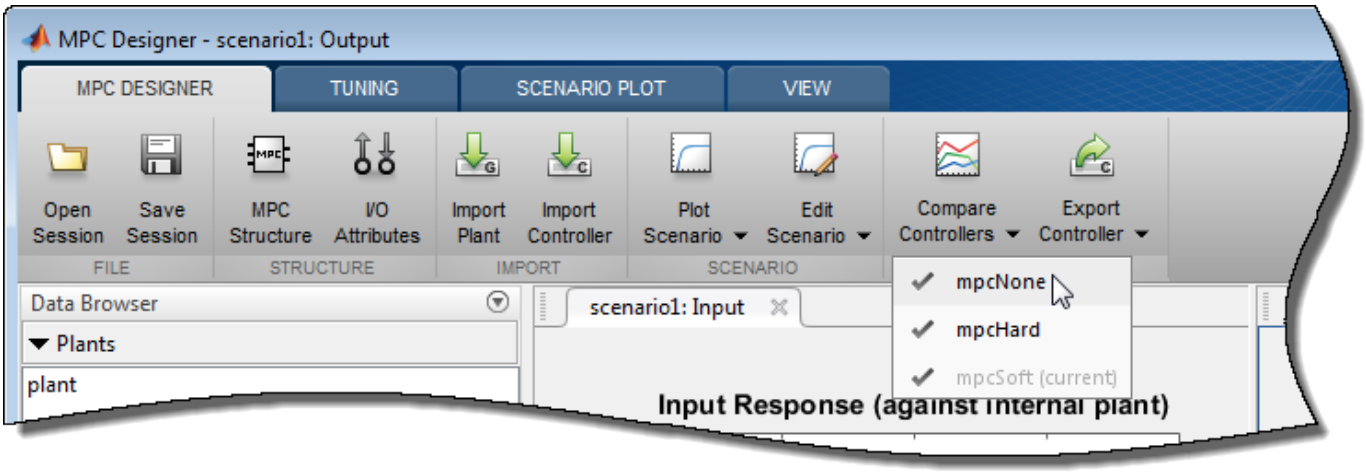


The response plots update to reflect the new `mpcSoft` configuration. In the **Output Response** plot, `mpcSoft` shows a compromise between the previous controller responses.

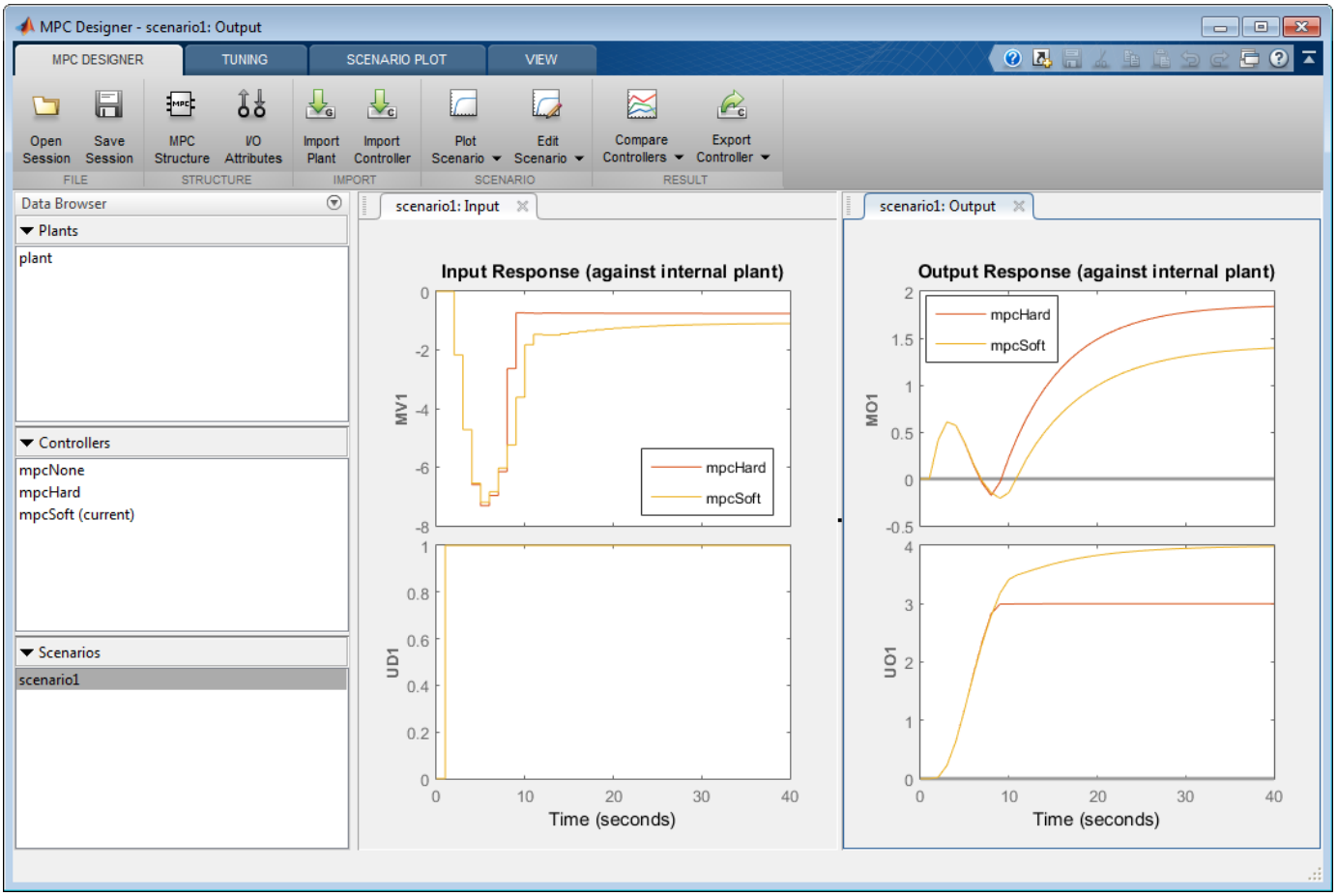
Remove Default Controller Response Plot

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

More About

- “Specify Constraints” on page 1-5
- “Design Controller Using MPC Designer”
- “Design MPC Controller in Simulink”

Adjust Input and Output Weights Based on Sensitivity Analysis

This example shows how to compute numerical derivatives of a cumulative performance index with respect to the weights of the MPC quadratic cost function, and use these derivatives to improve performance.

Define Plant Model

Create a state-space model for the plant.

```
plant = ss(tf({1,1,2;1 -1 -1},{[1 0 0],[1 0 0],[1 1];[1 2 8],[1 3],[1 1 3]}),'min');
```

The model is continuous-time and has 3 inputs (assumed to be manipulated variables), 2 outputs (assumed to be both measurable), and 8 state variables.

Design MPC Controller

Create an MPC controller with a sample time of `0.1`, a prediction horizon of `20` steps, and a control horizon of 3 steps.

```
mpcobj = mpc(plant,0.1,20,3);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Set constraints on manipulated variables and their rates of change.

```
for i = 1:3
    mpcobj.MV(i).Min = -2;
    mpcobj.MV(i).Max = 2;
    mpcobj.MV(i).RateMin = -4;
    mpcobj.MV(i).RateMax = 4;
end
```

Display the default cost function weights for the output variables, manipulated variables and manipulated variables rate.

```
mpcobj.Weights.OutputVariables
```

```
ans = 1×2

     1     1
```

```
mpcobj.Weights.ManipulatedVariables
```

```
ans = 1×3

     0     0     0
```

```
mpcobj.Weights.ManipulatedVariablesRate
```

```
ans = 1×3

    0.1000    0.1000    0.1000
```

Performance Evaluation Setup

Define a closed-loop cumulative performance index as the weighted Integral of the Square Error (ISE) between the plant signals and their references, calculated in the interval from `0` to `Tstop` seconds.

The weights, which reflect the desired closed-loop behavior, must be contained in a structure with the same fields as the `Weights` property of an MPC object.

```
PerformanceWeights = mpcobj.weights;
```

In this example, output tracking is more important than keeping the manipulated variable values low, therefore, define relatively higher weights on the output error, slightly higher weights on the manipulated variable rate, and keep the default weights on the manipulated variable values.

```
PerformanceWeights.OutputVariables = [100 100];
PerformanceWeights.ManipulatedVariablesRate = [1 1 1];
```

Note that `PerformanceWeights` is used only to calculate the cumulative performance index. It is not related to the weights specified inside the MPC controller object. Therefore the cumulative performance index is not related to the quadratic cost function that the MPC controller tries to minimize by choosing the manipulated variable values. Indeed, the performance index is based on a *closed loop* simulation until a time that is generally different than the prediction horizon, while the MPC controller calculates the moves which minimize its internal cost function up to the prediction horizon and in *open loop* fashion. Furthermore, even when the performance index is chosen to be of ISE type, its weights should be squared to match the weights defined in the MPC cost function.

Setup Simulation Parameters and Signals

In this example, you calculate the cumulative performance index sensitivity within a setpoint tracking scenario.

```
Tstop = 80;               % number of time steps to be simulated
r = ones(Tstop,1)*[1 1]; % set point reference signals
v = [];                   % no disturbance is added
simopt = mpcsimopt;       % create simulation options object
simopt.PlantInitialState = zeros(8,1); % set plant initial state
```

Calculate Sensitivities

Calculate the performance index value and its sensitivities to the `mpcobj` cost function weights, using the `sensitivity` function.

```
[J1, Sens1] = sensitivity(mpcobj, 'ISE', PerformanceWeights, Tstop, r, v, simopt);
```

```
-->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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Display sensitivities with respect to the weights for the output error signals, $\frac{\partial}{\partial W_y} J$.

```
Sens1.OutputVariables
```

```
ans = 1×2
10^4 ×
```

```
   -2.7346    2.7166
```

Display sensitivities with respect to the weights for the manipulated variable signals, $\frac{\partial}{\partial W_u} J$.

```
Sens1.ManipulatedVariables
```

```
ans = 1×3

    3.3375 -125.8266  -35.1067
```

Display sensitivities with respect to the weights for the manipulated variables rate signals, $\frac{\partial}{\partial W_{\Delta u}} J$.

```
Sens1.ManipulatedVariablesRate
```

```
ans = 1×3
10^4 ×

   -0.0007    1.0250   -0.8370
```

Adjust MPC Weights

Since you want to reduce the closed-loop cumulative performance index `J`, in this example the derivatives with respect to output weights show that the weight on `y1` should be increased, as the corresponding derivative is negative, while the weight on `y2` should be decreased.

Copy the MPC object to make modification on the new object.

```
mpcobj_new = mpcobj;
```

A negative sensitivity suggests increasing the first output weight from `1` to `2`.

```
mpcobj_new.Weights.OutputVariables(1) = 2;
```

A positive sensitivity suggests decreasing the second output weight from `1` to `0.2`.

```
mpcobj_new.Weights.OutputVariables(2) = 0.2;
```

Note that the sensitivity analysis only tells you in which direction to change the parameters, but not by how much. A trial and error procedure to select the appropriate magnitude of the change is expected.

Verify Performance Changes

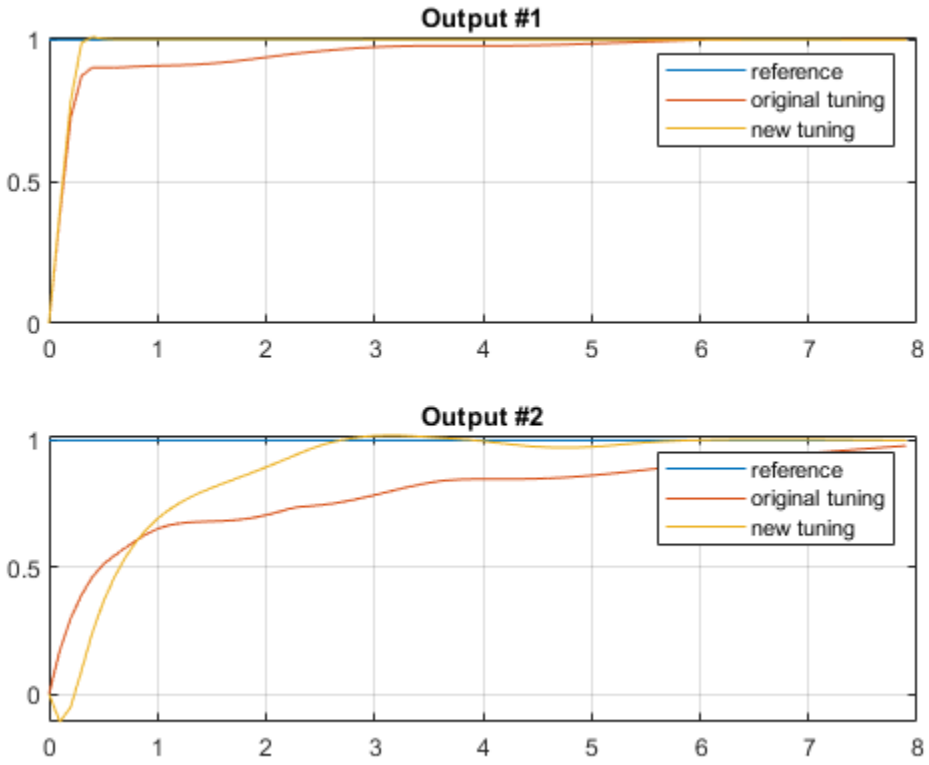
Simulate both MPC controllers.

```
[y1, t1, u1] = sim(mpcobj, Tstop, r, v, simopt);
[y2, t2, u2] = sim(mpcobj_new, Tstop, r, v, simopt);
```

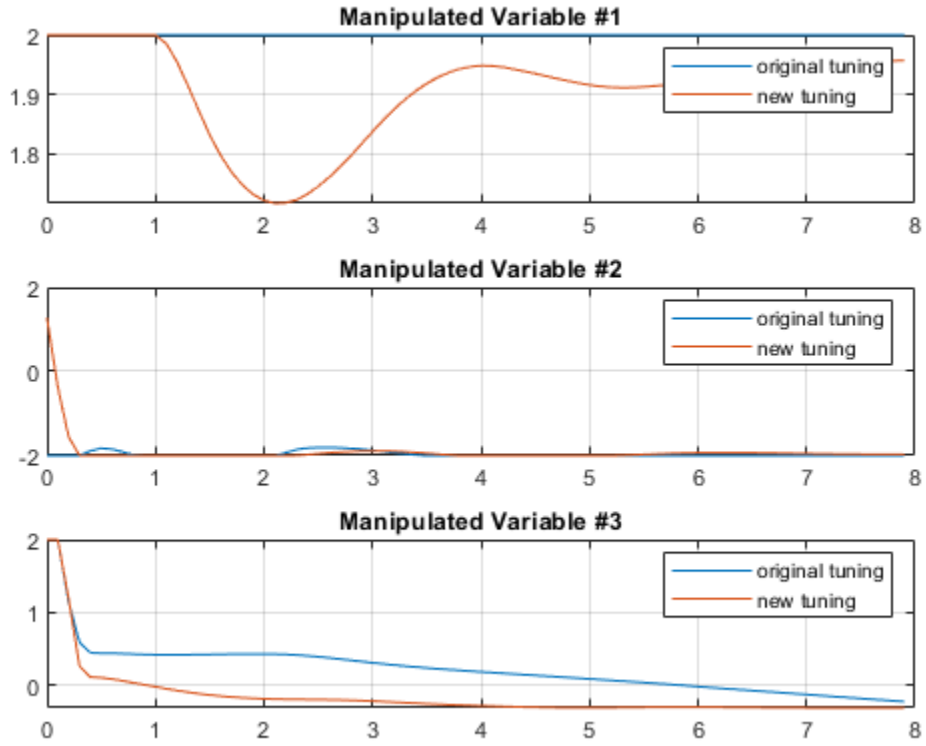
```
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Plot simulation results for both controllers.

```
% plot plant outputs
h1 = figure;
subplot(211)
plot(t2,r(:,1),t1,y1(:,1),t2,y2(:,1));grid
legend('reference','original tuning','new tuning')
title('Output #1')
subplot(212)
plot(t2,r(:,2),t1,y1(:,2),t2,y2(:,2));grid
legend('reference','original tuning','new tuning')
title('Output #2')
```




```
% plot manipulated variables
h2 = figure;
subplot(311)
plot(t1,u1(:,1),t2,u2(:,1));grid
legend('original tuning','new tuning')
title('Manipulated Variable #1')
subplot(312)
plot(t1,u1(:,2),t2,u2(:,2));grid
legend('original tuning','new tuning')
title('Manipulated Variable #2')
subplot(313)
plot(t1,u1(:,3),t2,u2(:,3));grid
legend('original tuning','new tuning')
title('Manipulated Variable #3')
```



Verify Cumulative Performance Index is Reduced

Compute the cumulative performance index for the new controller using the same performance measure.

```
J2 = sensitivity(mpcobj_new, 'ISE', PerformanceWeights, Tstop, r, v, simopt);
```

```
-->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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Previous Cumulative Performance Index.

```
J1
```

```
J1 = 1.2865e+05
```

New Cumulative Performance Index.

```
J2
```

```
J2 = 1.1623e+05
```

As expected the new value of the cumulative performance index is lower than the old value.

Use a User-Defined Performance Function

This is an example of how to write a user-defined performance function used by the `sensitivity` method. In this example, the custom function `mpc_performance_function.m` implements the standard ISE performance index based on `PerformanceWeights`.

Display the function.

```
type mpc_performance_function.m
```

```
function J = mpc_performance_function(MPCobj, Tsteps, r, PerformanceWeights)
% This is an example of how to write a user defined performance function
% used by the "sensitivity" method.  In this example, the code illustrate
% how we use performance weights to compute the cumulative performance index.

% Copyright 1990-2014 The MathWorks, Inc.

% Carry out simulation
[y,t,u] = sim(MPCobj, Tsteps, r);
du = [u(1,:);diff(u)];
% Get Weights in MPCobj
ny = size(MPCobj,'mo') + size(MPCobj,'uo');
nmv = size(MPCobj,'mv');
Wy = PerformanceWeights.OutputVariables(:);Wy=Wy(1:ny);
Wu = PerformanceWeights.ManipulatedVariables(:);Wu=Wu(1:nmv);
Wdu = PerformanceWeights.ManipulatedVariablesRate(:);Wdu=Wdu(1:nmv);
% Set mv target to 0
utarget=zeros(nmv,1);
% Compute J in ISE form
J=0;
aux=(y-r)*Wy;
J=J+aux'*aux;
aux=(u-ones(Tsteps,1)*utarget')*Wu;
J=J+aux'*aux;
aux=du*Wdu;
J=J+aux'*aux;
```

Use the custom function to calculate the performance index.

```
J3 = sensitivity(mpcobj,'mpc_performance_function',Tstop,r,PerformanceWeights)
```

```
J3 = 1.2865e+05
```

As expected, for `mpcobj`, user-defined Cumulative Performance Index `J3` has the same value as `J1`.

See Also

`sensitivity` | `mpc`

Understanding Control Behavior by Examining Optimal Control Sequence

This example shows how to inspect the optimized sequence of manipulated variables computed by a model predictive controller at each sample time.

The plant is a double integrator subject to input saturation.

Design MPC Controller

The basic setup of the MPC controller includes:

- A double integrator as the prediction model
- A prediction horizon of `20`
- A control horizon of `10`
- Input constraints `-1 <= u(t) <= 1`

Specify the MPC controller.

```
Ts = 0.1;
p = 20;
m = 10;
mpcobj = mpc(tf(1,[1 0 0]),Ts,p,m);
mpcobj.MV = struct('Min',-1,'Max',1);
nu = 1;
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Simulate Model in Simulink

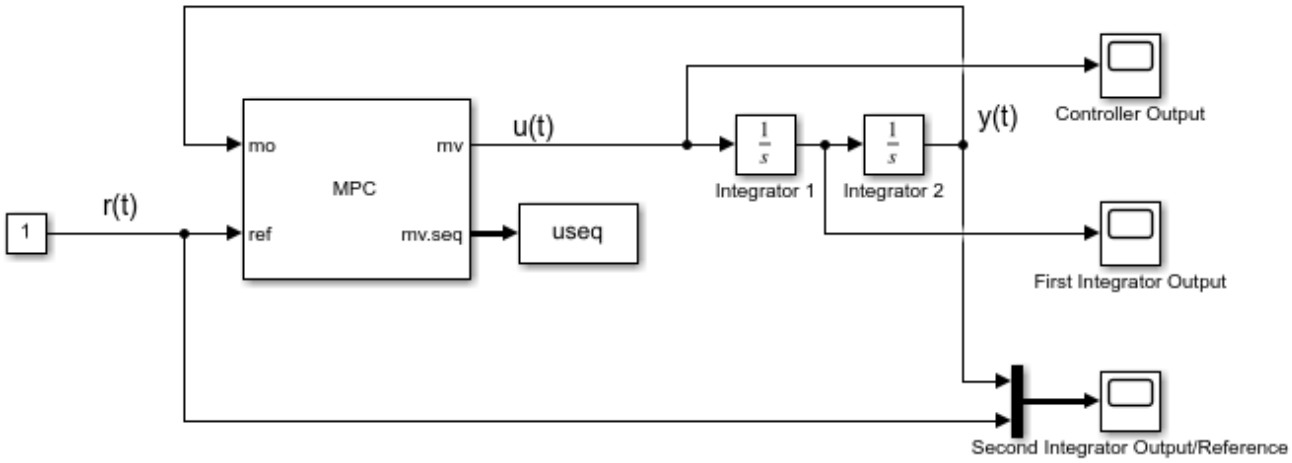
To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
```

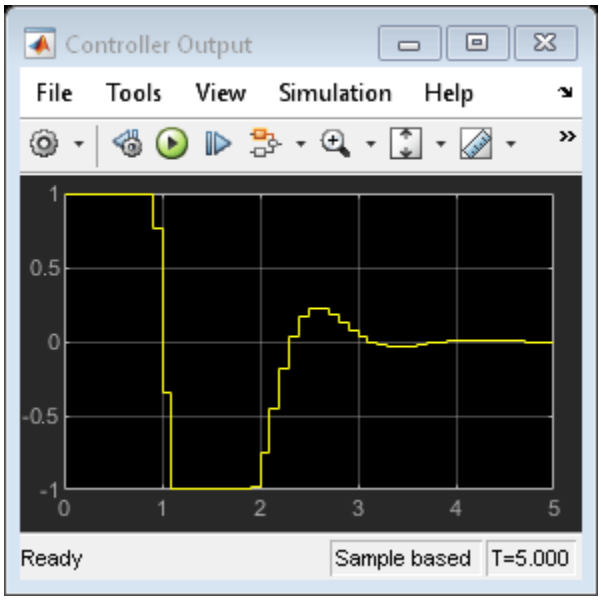
Open the Simulink model, and run the simulation.

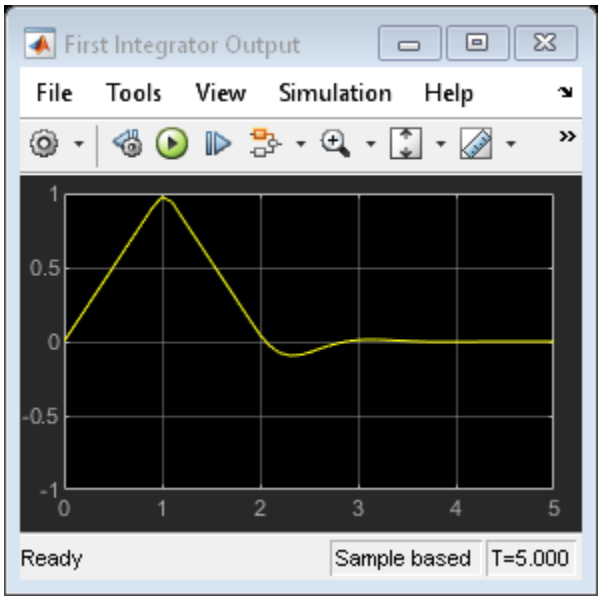
```
mdl = 'mpc_sequence';
open_system(mdl)
sim(mdl)
```

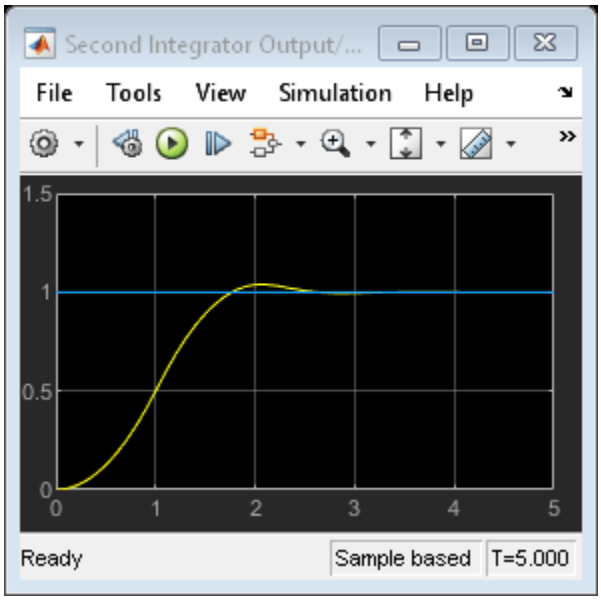
```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



Copyright 1990-2012 The MathWorks, Inc.







The MPC Controller block has an `mv.seq` output port, which is enabled by selecting the **Optimal control sequence** block parameter. This port outputs the optimal control sequence computed by the controller at each sample time. The output signal is an array with `p+1` rows and `Nmv` columns, where `p` is prediction horizon and `Nmv` is the number of manipulated variables.

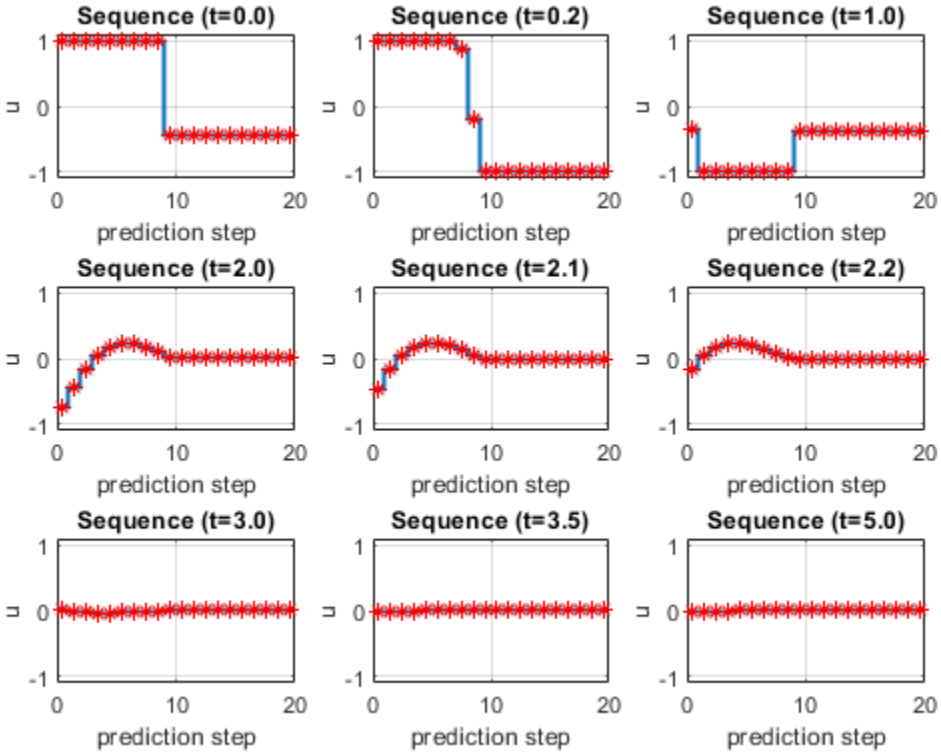
In a similar manner, the controller can output the optimal state sequence (`x.seq`) and the optimal output sequence (`y.seq`).

When the simulation stops, the `To Workspace` block connected to the `mv.seq` port exports this control sequence to the MATLAB® workspace, logging the data in the variable `useq`.

Analyze Optimal Control Sequences

Plot the optimal control sequence at specific time instants.

```
times = [0 0.2 1 2 2.1 2.2 3 3.5 5];
figure('Name','Optimal sequence history')
for t = 1:9
    ct = times(t)*10+1;
    subplot(3,3,t)
    h = stairs(0:p,useq.signals.values(ct,:));
    h.LineWidth = 1.5;
    hold on
    plot((0:p)+.5,useq.signals.values(ct,:),'*r')
    xlabel('prediction step')
    ylabel('u')
    title(sprintf('Sequence (t=%3.1f)',useq.time(ct)))
    grid
    axis([0 p -1.1 1.1])
    hold off
end
```



The MPC controller uses the first two seconds to bring the output very close to the set point. The controller output is at the upper limit (+1) for one second and switches to the lower limit (-1) for the next second, which is the best control strategy under the input constraints.

```
bdclose(mdl)
```

See Also

MPC Controller

Controller Simulation

- “Simulating MPC Controller with Plant Model Mismatch” on page 5-2
- “Test MPC Controller Robustness using MPC Designer” on page 5-5
- “Generate Simulink Model from MPC Designer” on page 5-14
- “Test an Existing MPC Controller with Simulink” on page 5-16
- “Signal Previewing” on page 5-19
- “Improving Control Performance with Look-Ahead (Previewing)” on page 5-20
- “Update Constraints at Run Time” on page 5-27
- “Vary Input and Output Bounds at Run Time” on page 5-30
- “Tune Weights at Run Time” on page 5-35
- “Tuning MPC Controller Weights at Run-Time” on page 5-36
- “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42
- “Adjust Horizons at Run Time” on page 5-45
- “Evaluate Control Performance Using Run-Time Horizon Adjustment” on page 5-48
- “Switch Controller Online and Offline with Bumpless Transfer” on page 5-57
- “Switching Controllers Based on Optimal Costs” on page 5-67
- “Monitoring Optimization Status to Detect Controller Failures” on page 5-73
- “Simulate MPC Controller with a Custom QP Solver” on page 5-77
- “Use Suboptimal Solution in Fast MPC Applications” on page 5-86
- “Design and Cosimulate Control of High-Fidelity Distillation Tower with Aspen Plus Dynamics” on page 5-93
- “Simulate Linear MPC Controller with Nonlinear Plant using Successive Linearizations” on page 5-111

Simulating MPC Controller with Plant Model Mismatch

This example shows how to simulate a model predictive controller with a mismatch between the predictive plant model and the actual plant, as well as measured and unmeasured disturbances, using the `sim` command.

The predictive plant model has 2 manipulated variables, 2 unmeasured input disturbances, and 2 measured outputs. The actual plant has different dynamics.

Define Plant Model

Define the parameters of the nominal plant which the MPC controller is based on. Systems from MV to MO and UD to MO are identical.

```
p1 = tf(1,[1 2 1])*[1 1; 0 1];
plant = ss([p1 p1],'minimal');
plant.InputName = {'mv1','mv2','ud3','ud4'};
```

Design MPC Controller

Define inputs 1 and 2 as manipulated variables, 3 and 4 as unmeasured disturbances.

```
plant = setmpcsignals(plant,'MV',[1 2],'UD',[3 4]);
% Create the controller object with sampling period, prediction and control
% horizons:
mpcobj = mpc(plant,1,40,2);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

For unmeasured input disturbances, the MPC controller will use the following unmeasured disturbance model.

```
distModel = eye(2,2)*ss(-.5,1,1,0);
mpcobj.Model.Disturbance = distModel;
```

Define the Real Plant Model Used in Simulation

Define the parameters of the actual plant in closed loop with the MPC controller.

```
p2 = tf(1.5,[0.1 1 2 1])*[1 1; 0 1];
psim = ss([p2 p2],'minimal');
psim = setmpcsignals(psim,'MV',[1 2],'UD',[3 4]);
```

Simulate Closed-Loop Response Using the SIM Command

Define reference trajectories and unmeasured disturbances entering the actual plant.

```
dist = ones(1,2);   % unmeasured disturbance signal
refs = [1 2];       % output reference signal
Tf = 20;            % total number of simulation steps
```

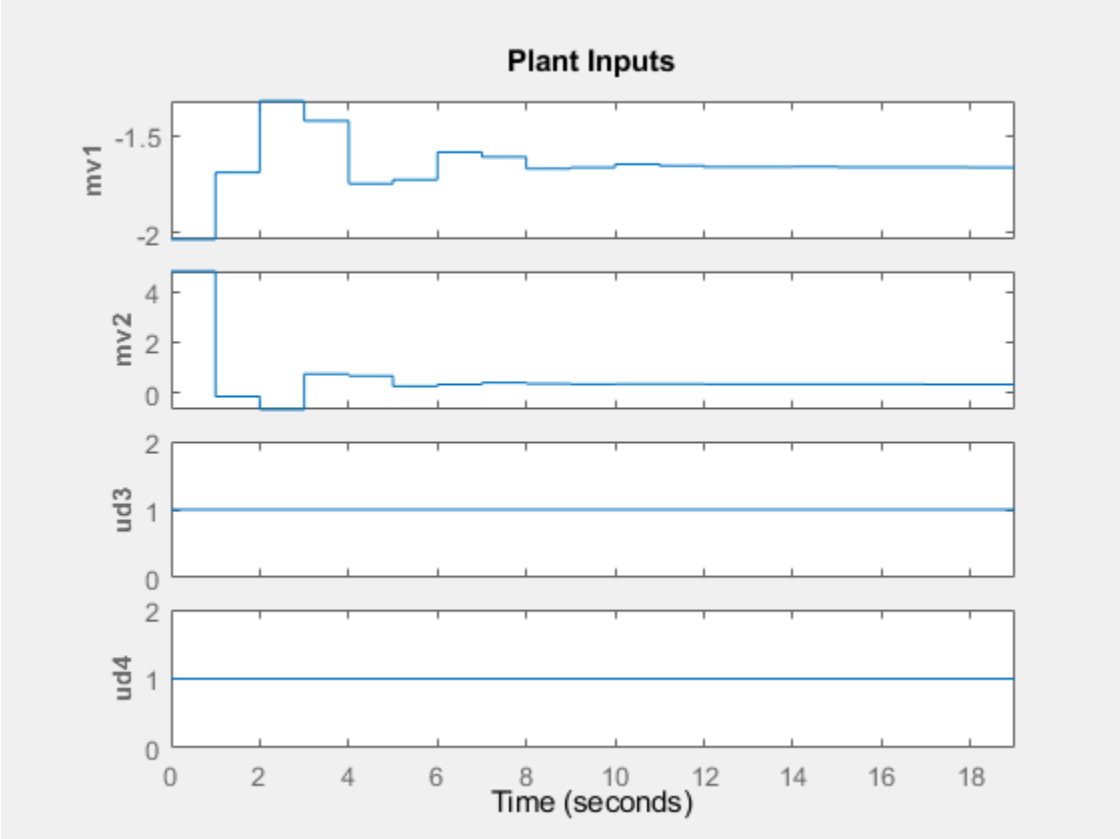
Create an MPC simulation options object. This allows you to define both unmeasured disturbances and a plant different than the one which the MPC controller uses as a prediction model.

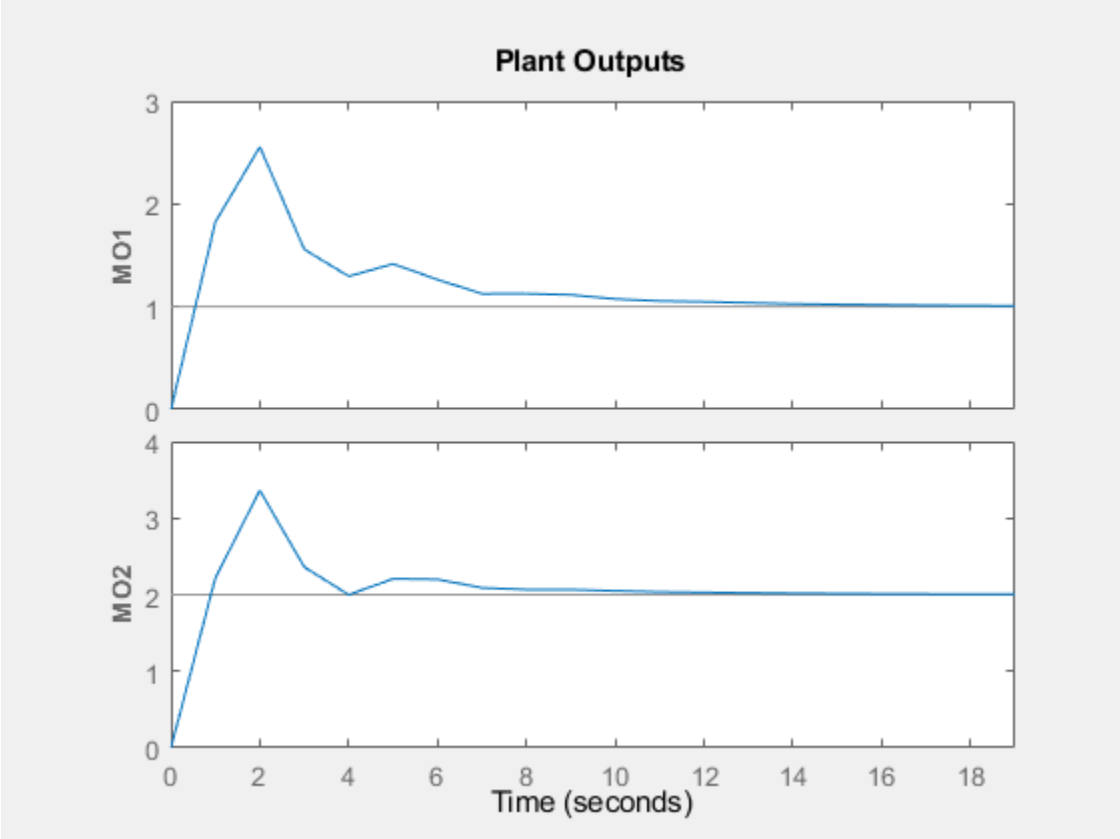
```
options = mpcsimopt(mpcobj);
options.unmeas = dist;  % unmeasured disturbance signal
options.model = psim;   % real plant model
```

Run the closed-loop MPC simulation with model mismatch and unforeseen unmeasured disturbance inputs.

```
sim(mpcobj,Tf,refs,options);
```

```
-->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 measured
-->Converting model to discrete time.
```





The closed loop tracking performance is acceptable despite the presence of model mismatches and unmeasured input disturbances.

See Also

`mpc`

More About

- “MPC Prediction Models”
- “Test MPC Controller Robustness using MPC Designer” on page 5-5

Test MPC Controller Robustness using MPC Designer

This example shows how to test the sensitivity of your model predictive controller to prediction errors using simulations, within **MPC Designer**.

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 more sophisticated approaches for MIMO systems. It can also be helpful to assess robustness by running simulations with selected model mismatches and disturbances.

Define 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);
```

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**, and import the plant model.

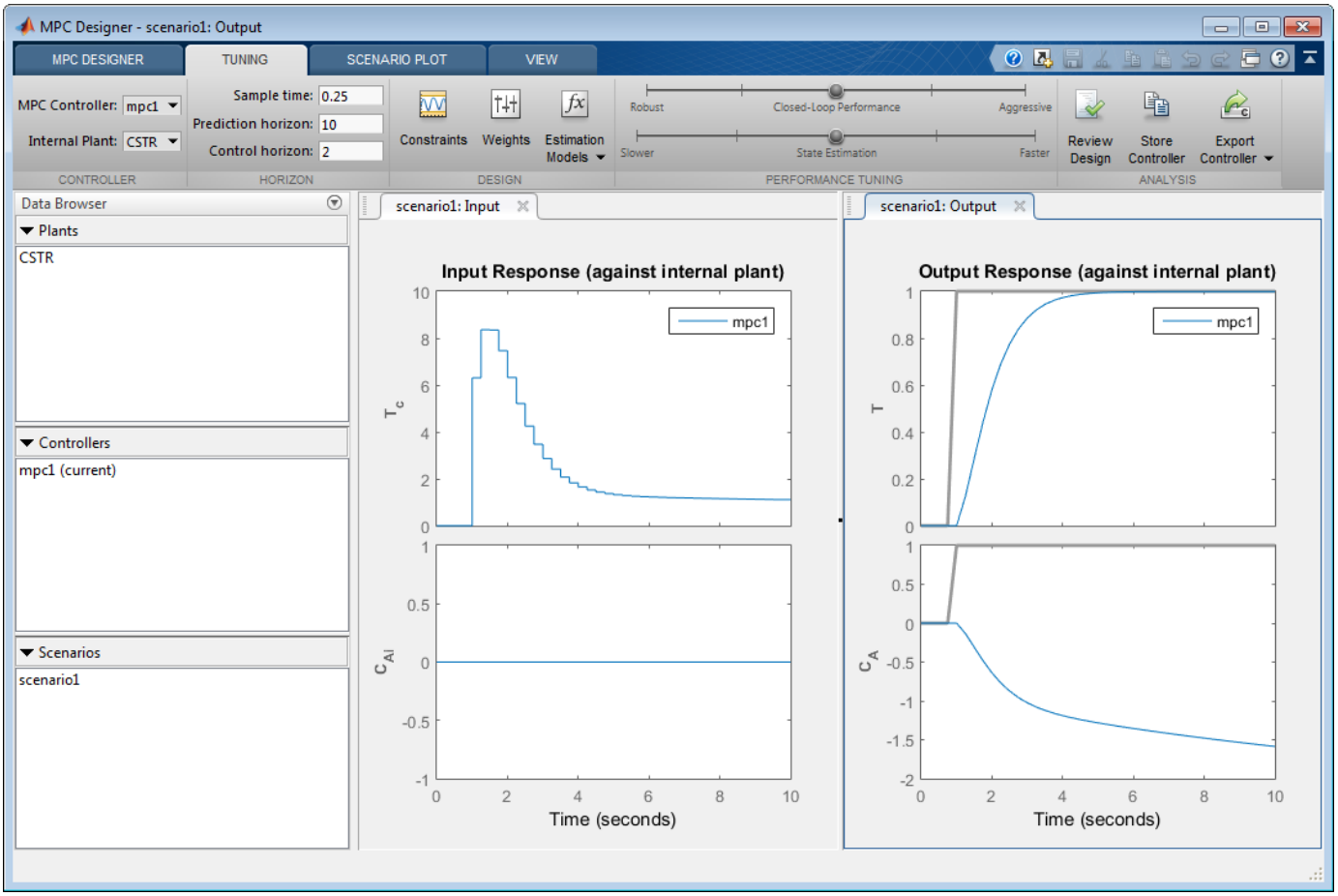
```
mpcDesigner(CSTR)
```

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 **MPC Designer**, 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.

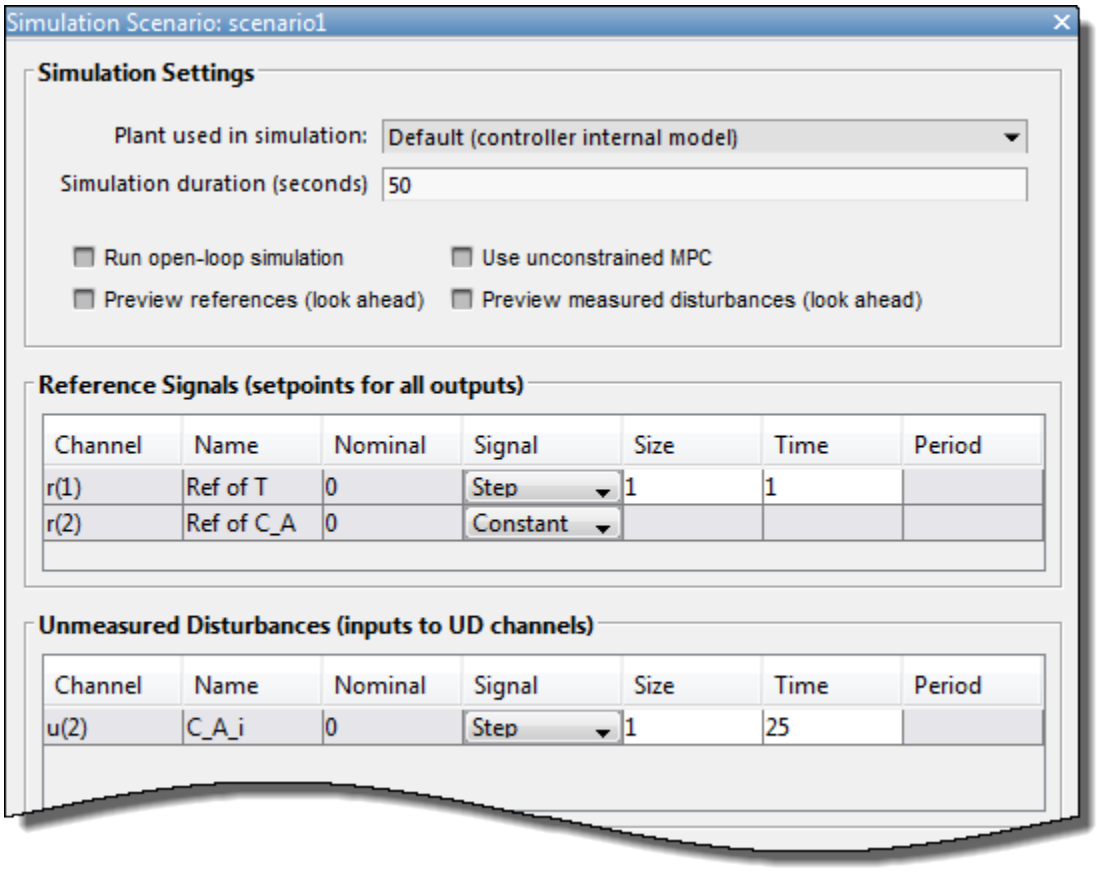
In the **Data Browser**, in the **Scenarios** sections, right-click `scenario1`, and select **Edit**.

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

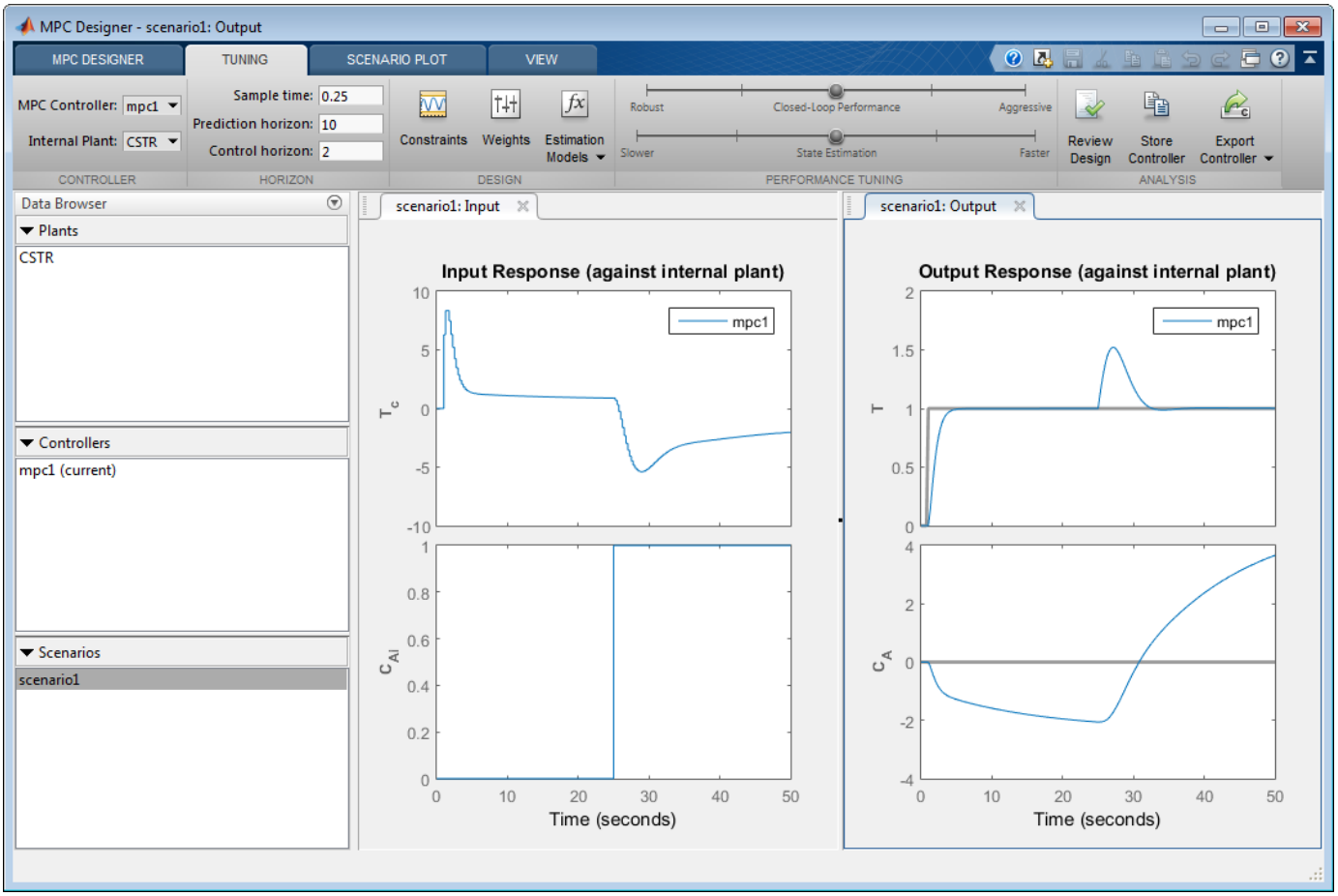
In the **Reference Signals** table, keep the default `Ref of T` setpoint configuration, which simulates a unit-step change in the reactor temperature.

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.


Simulation Scenario: scenario1
Simulation Settings
Plant used in simulation:
Default (controller internal model)
Simulation duration (seconds)
50
Run open-loop simulation
Use unconstrained MPC
Preview references (look ahead)
Preview measured disturbances (look ahead)
Reference Signals (setpoints for all outputs)
Channel
Name
Nominal
Signal
Size
Time
Period
r(1)
Ref of T
0
Step
1
1
r(2)
Ref of C_A
0
Constant
Unmeasured Disturbances (inputs to UD channels)
Channel
Name
Nominal
Signal
Size
Time
Period
u(2)
C_A_i
0
Step
1
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];
```

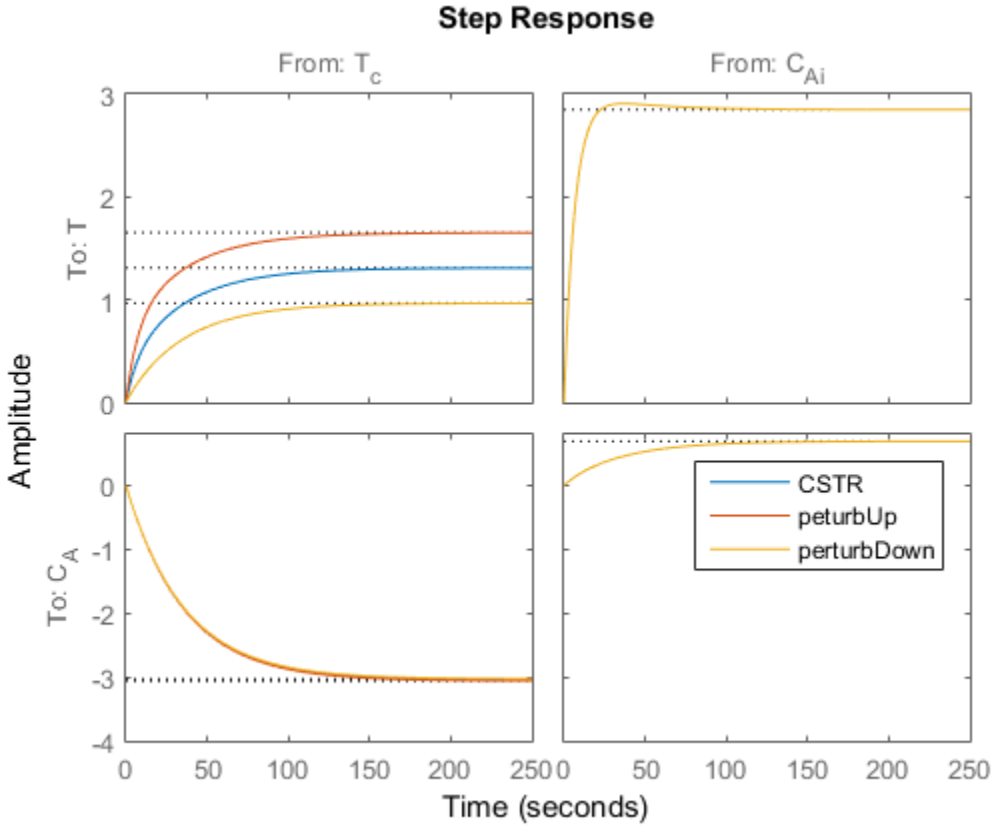
Create the two perturbed plant models.

```
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','peturbUp','perturbDown')
```

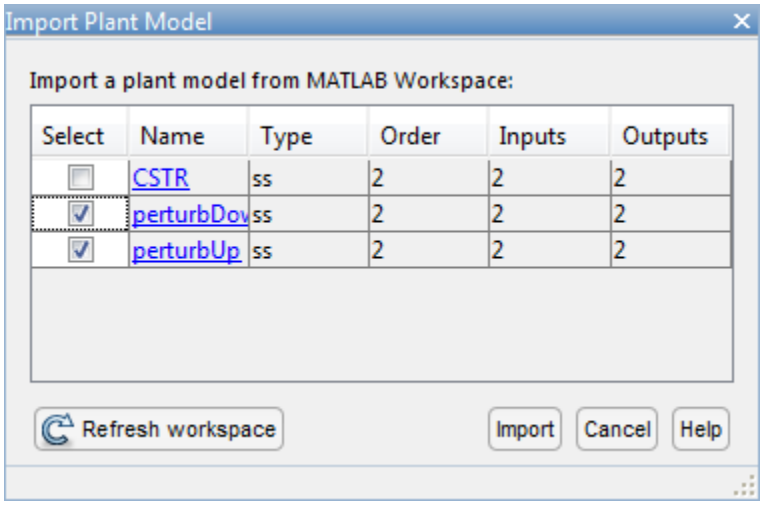


Perturbing element `B(2,1)` of the CSTR plant changes the magnitude of the response of the reactor temperature, `T`, to changes in the coolant temperature, `Tc`.

Import Perturbed Plants

In **MPC Designer**, 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

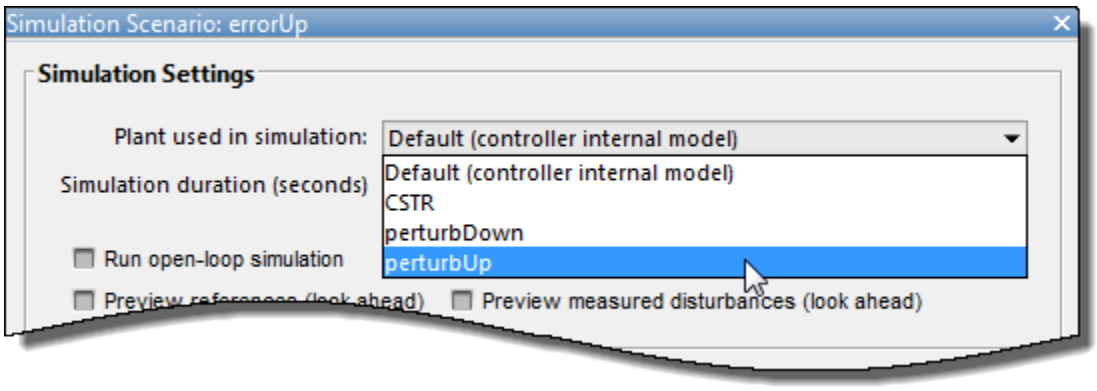
Create two simulation scenarios that use the perturbed plant models.

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`.

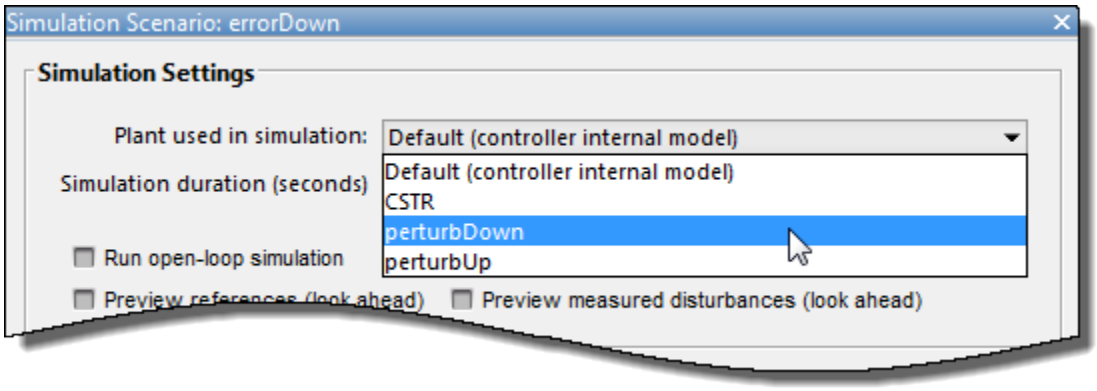


Click **OK**.

Repeat this process for the second perturbed plant.

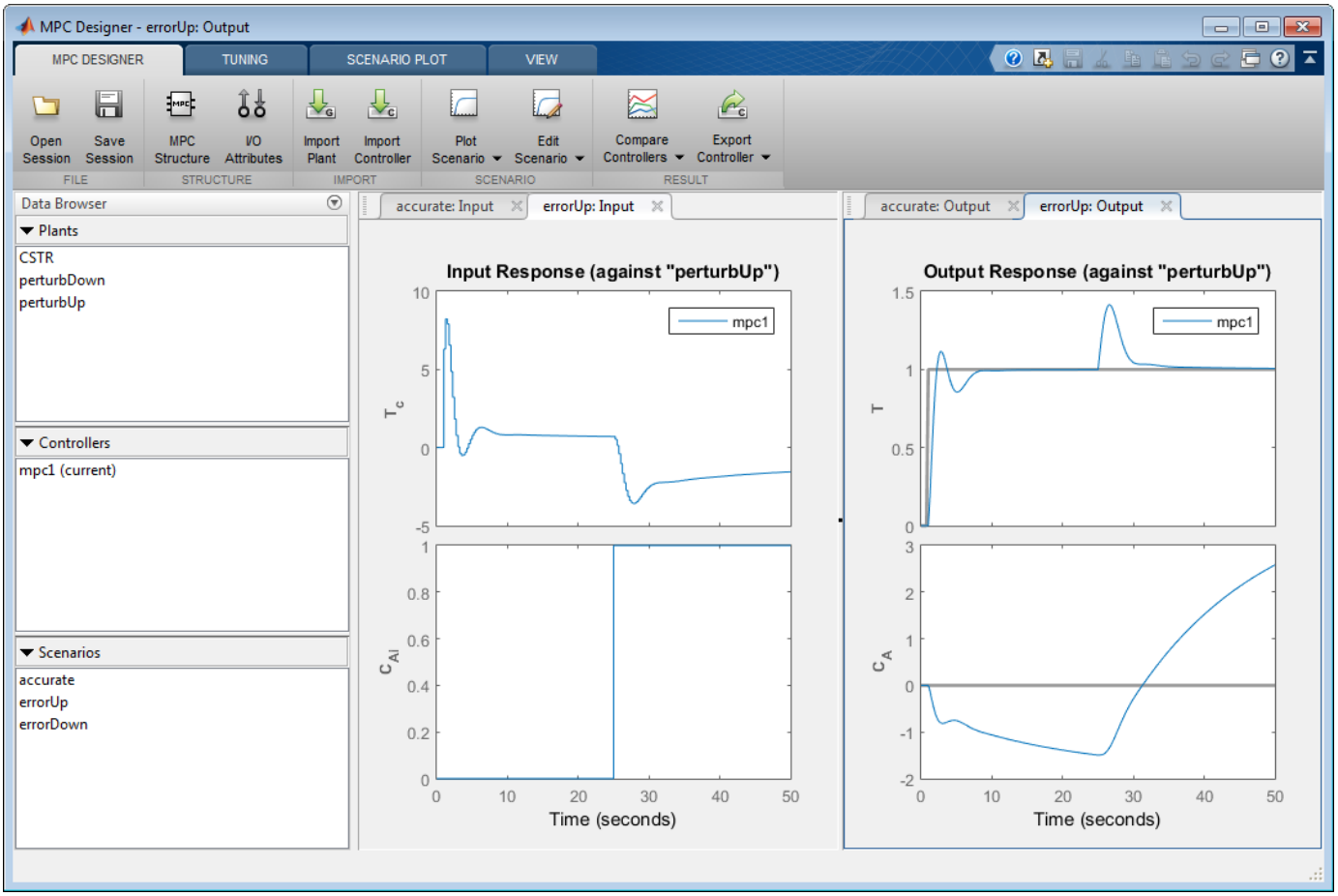
Copy the `accurate` scenario and rename it to `errorDown`.

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



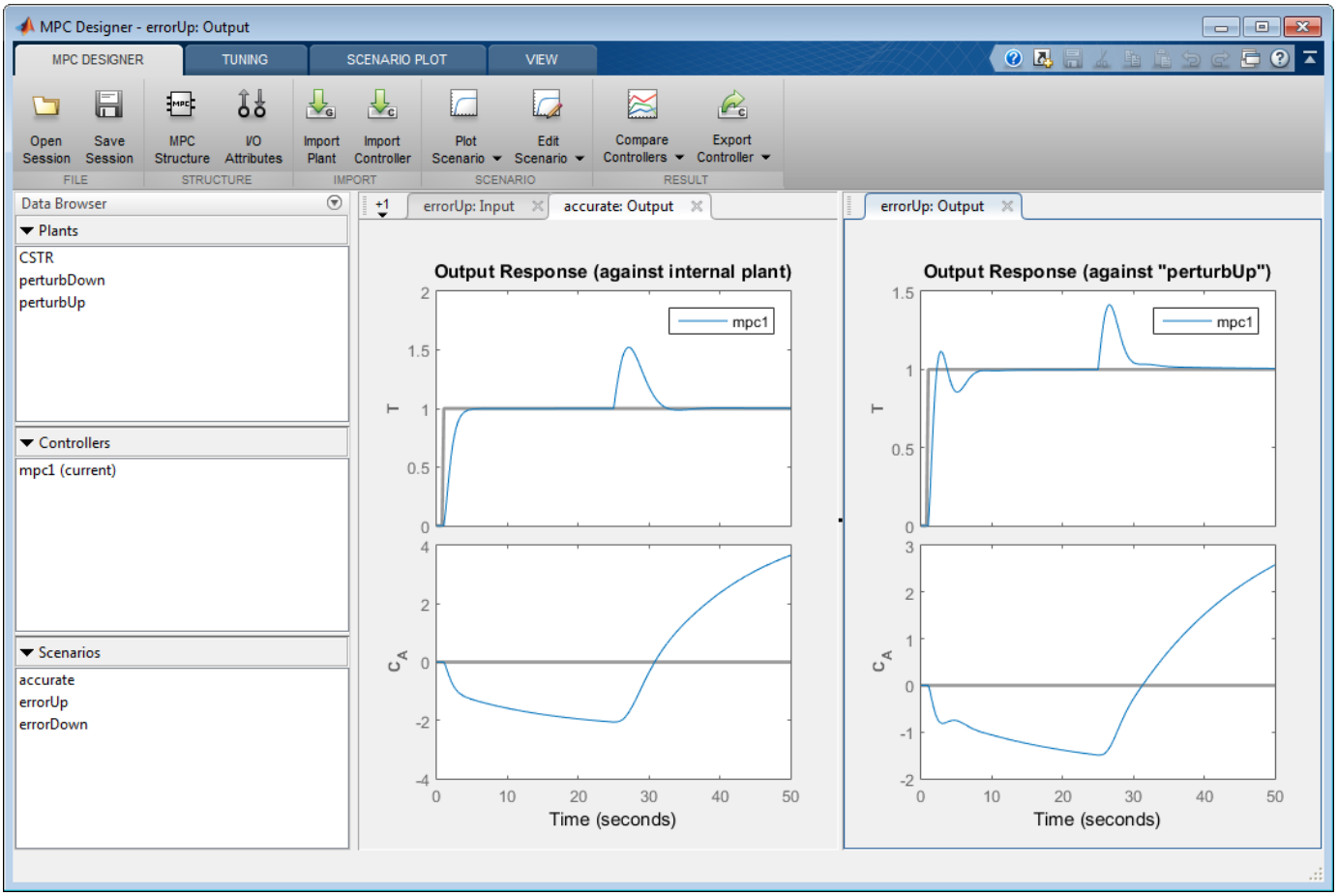
Examine `errorUp` Simulation Response

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.



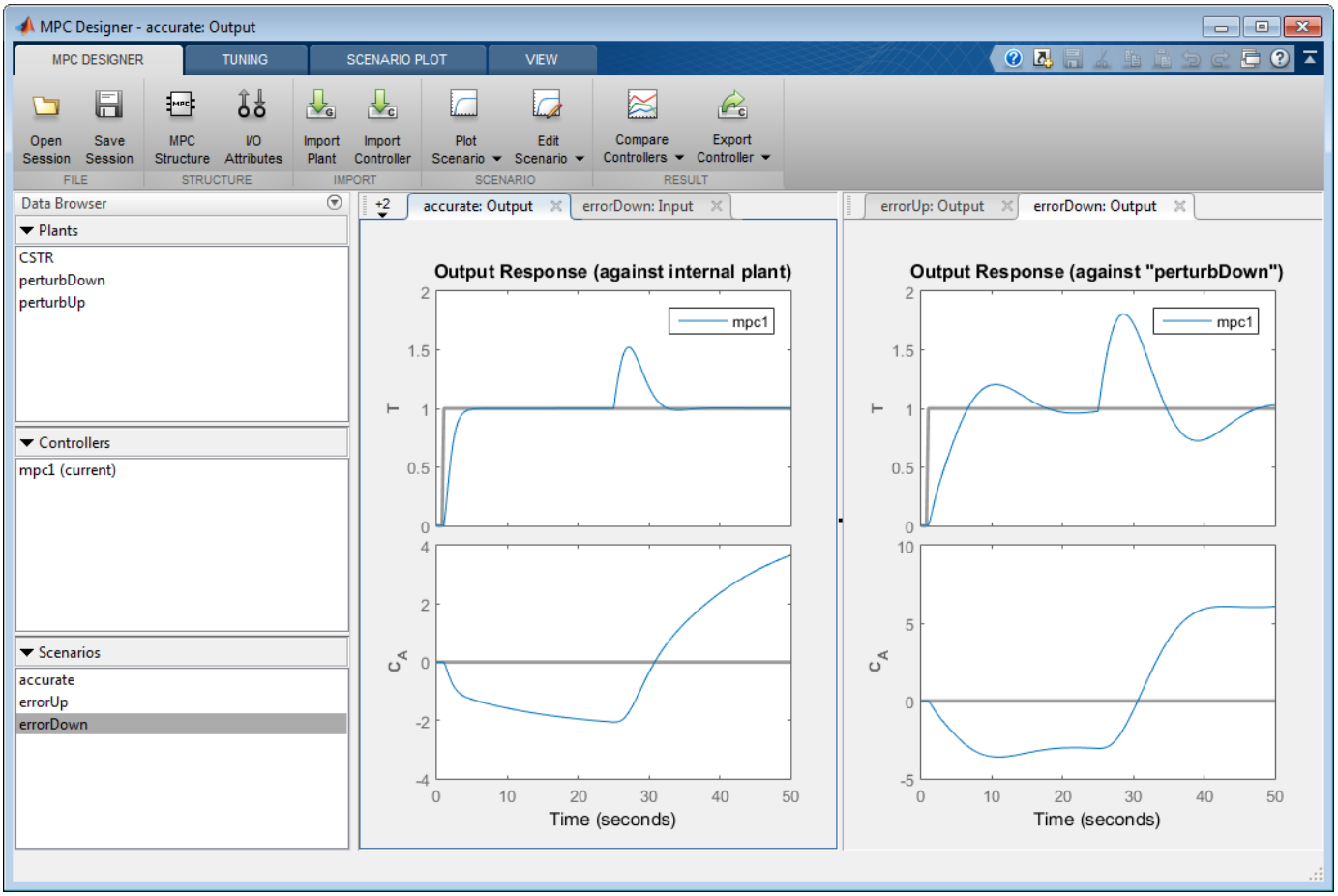
The perturbation creates a plant, `perturbUp`, that responds faster to manipulated variable changes than the controller predicts. On the **errorUp: Output** tab, in the **Output Response** plot, the **T** setpoint step response has about 10% overshoot with a longer settling time. Although this response 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**

More About

- “Design Controller Using MPC Designer”
- “Test an Existing MPC Controller with Simulink” on page 5-16
- “Simulating MPC Controller with Plant Model Mismatch” on page 5-2

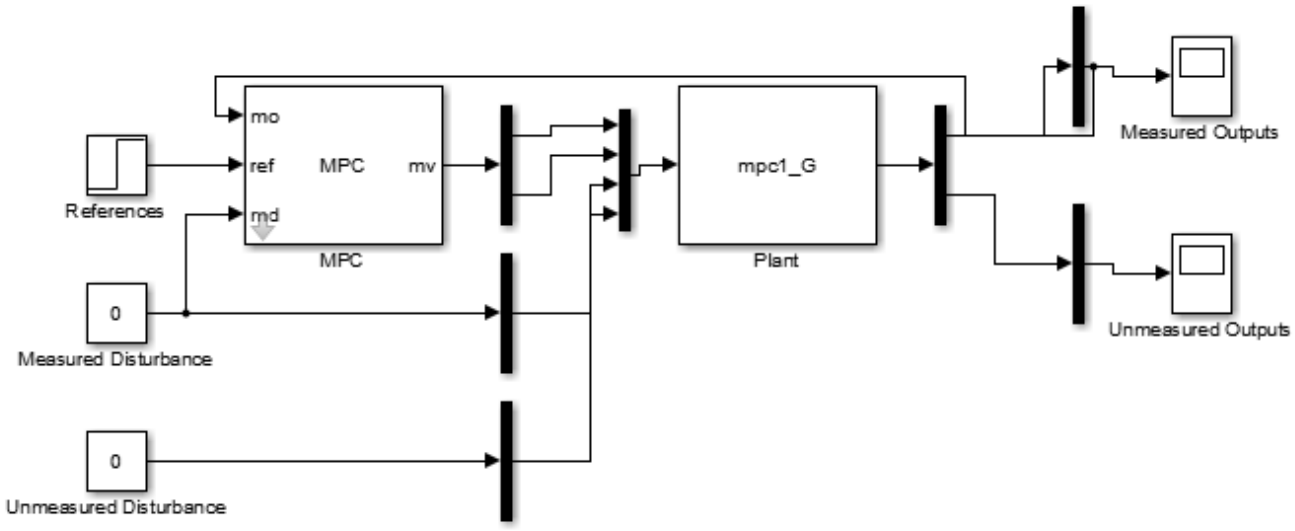
Generate Simulink Model from MPC Designer

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, on the **MPC Designer** tab, in the **Result** section, click **Export Controller**.
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 10-2.

See Also

MPC Designer | MPC Controller

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2
- “Design MPC Controller in Simulink”
- “Generate MATLAB Code from MPC Designer” on page 1-89

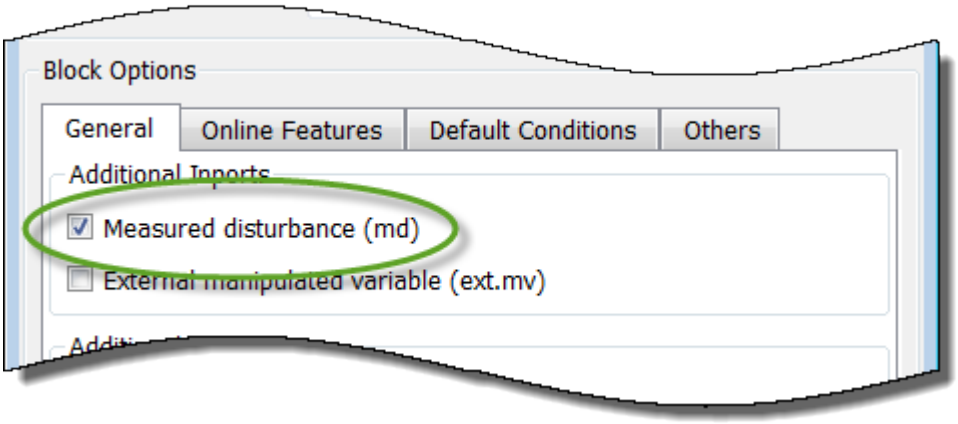
Test an Existing MPC Controller with Simulink

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

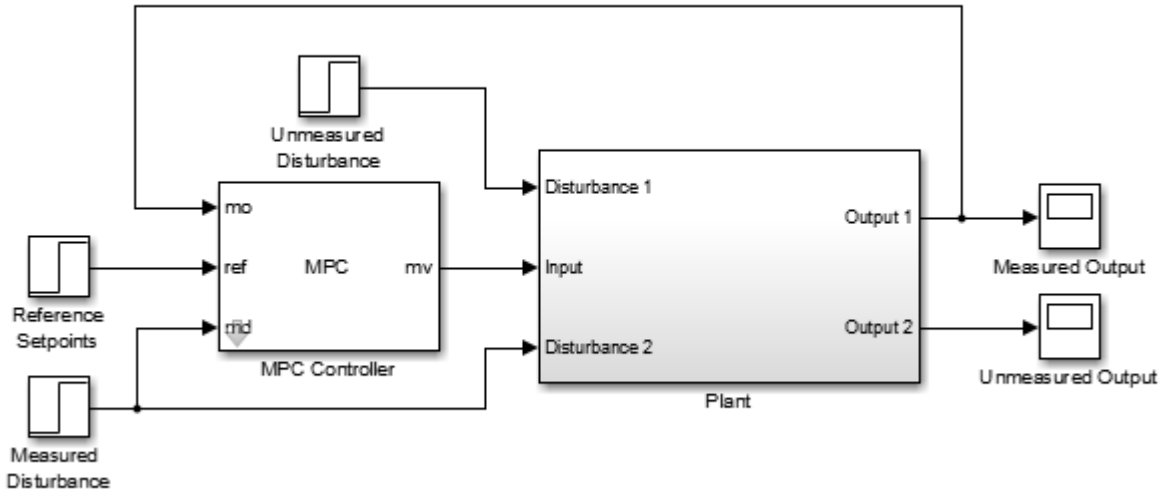
1. Open your Simulink model.
2. Add an MPC Controller block to the model.
3. If your controller includes measured disturbances, add the `md` inport to the MPC Controller block.

   Double-click the MPC Controller block.

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



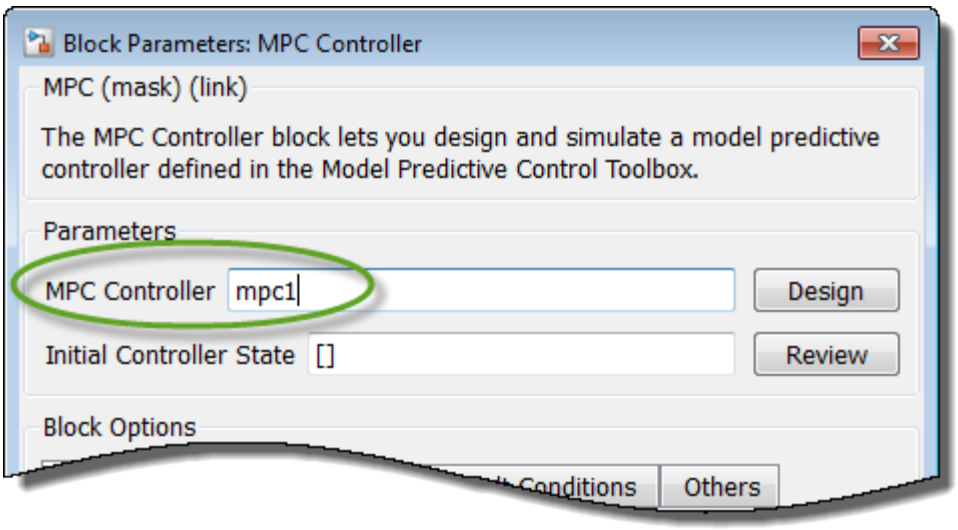
   Click **OK**.
4. Connect the plant and controller signals in the Simulink model. Connect:
   - The plant inputs to the manipulated variable (`mv`) inport of the MPC Controller block.
   - The plant measured outputs to the measured output (`mo`) inport of the MPC Controller block.
   - The measured disturbances, if any, to the plant and to the measured disturbance (`md`) inport of the MPC Controller block.
   - Any unmeasured disturbances or unmeasured outputs to their corresponding plant inport and outport.
   - The reference signals to the reference (`ref`) inport of the MPC Controller block.
   - Connect any signal that you would like to visualize (such for example the plant outputs) to Scope blocks.



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 **MPC Designer**:
  - In the Block Parameters dialog box, click **Design**.

- In **MPC Designer**, 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. Verify that the simulation terminates without errors and that the signals in the loop behave as expected.

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” on page 5-14.

See Also

`mpc` | MPC Controller | **MPC Designer**

More About

- “Design MPC Controller in Simulink”
- “Design MPC Controller at the Command Line”
- “Generate Simulink Model from MPC Designer” on page 5-14

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” on page 2-7, these signals can vary within the prediction horizon. If your application allows you to anticipate trends in such signals, an MPC controller with signal previewing can improve reference tracking, measured disturbance rejection, or both.

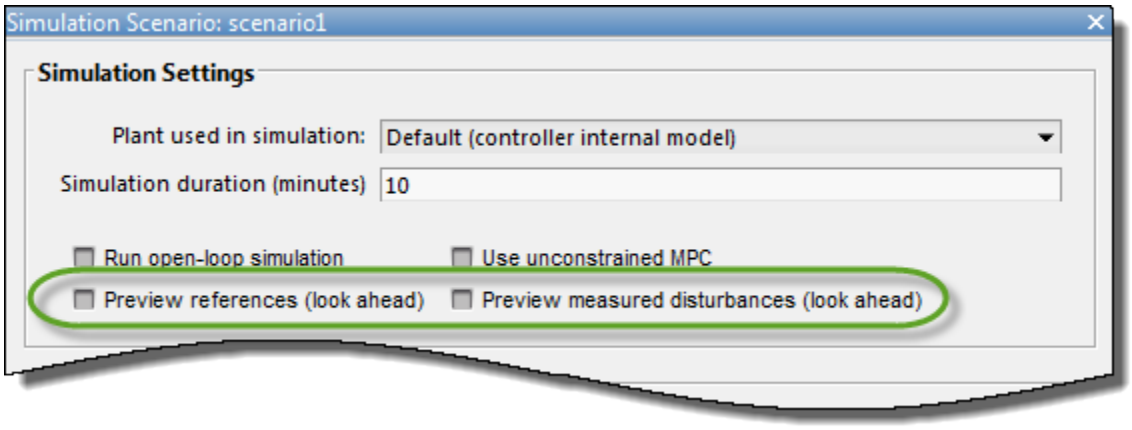
The following Model Predictive Control Toolbox commands provide previewing options:

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

For Simulink, the following blocks support previewing:

- MPC Controller
- Adaptive MPC Controller
- Multiple MPC Controllers

In **MPC Designer**, 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.



See Also

More About

- “Update Constraints at Run Time” on page 5-27
- “Improving Control Performance with Look-Ahead (Previewing)” on page 5-20

Improving Control Performance with Look-Ahead (Previewing)

This example shows how to design a model predictive controller with look-ahead (previewing) on reference and measured disturbance trajectories.

Define Plant Model

Define the plant model as a linear time invariant system with two inputs (one manipulated variable and one measured disturbance) and one output.

```
plant = ss(tf({1,1},{[1 .5 1],[1 1]}),'min');
```

Get the state-space matrices of the plant model, set a sampling time of 0.2 s, get the discrete-time matrices, and specify the initial condition.

```
[A,B,C,D] = ssdata(plant);              % continuous time ss realization
Ts = 0.2;                               % sampling time
[Ad,Bd,Cd,Dd] = ssdata(c2d(plant,Ts));  % discrete time ss realization
x0 = [0;0;0];                           % initial condition
```

Design Model Predictive Controller

Define type of input signals.

```
plant = setmpcsignals(plant,'MV',1,'MD',2);
```

Create the `mpc` object.

```
p = 20;                         % prediction horizon
m = 10;                         % control horizon
mpcobj = mpc(plant,Ts,p,m);
% Specify MV constraints.
mpcobj.MV = struct('Min',0,'Max',2);
% Specify weights
mpcobj.Weights = struct('MV',0,'MVRate',0.1,'Output',1);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Simulate Closed Loop Using the `sim` Command

```
Tstop = 30;                     % simulation time.
time = (0:Ts:(Tstop+p*Ts))';    % time vector
r = double(time>10);            % reference signal
v = -double(time>20);           % measured disturbance signal
```

Use the `mpcsimopt` object to turn on previewing feature in the closed-loop simulation.

```
params = mpcsimopt(mpcobj);
params.MDLookAhead='on';
params.RefLookAhead='on';
```

Simulate in MATLAB® with the MPC Toolbox `sim` command.

```
YY1 = sim(mpcobj,Tstop/Ts+1,r,v,params);
```

```
-->Converting model to discrete time.
-->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 measured
```

Simulate Using the `mpcmove` Command

Create array to store the closed-loop outputs.

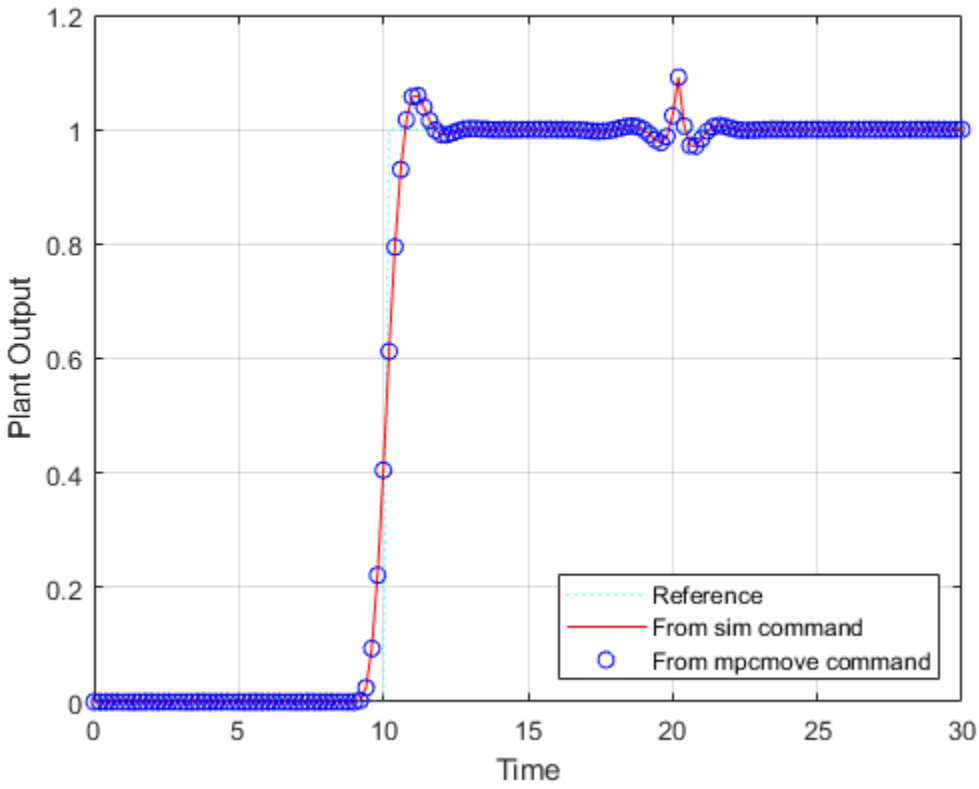
```
YY2 = [];
% Create variables to store current states of plant and controller
x = x0;                   % initial plant state
xmpc = mpcstate(mpcobj);  % pointer to current controller state
```

Start simulation loop

```
for ct=0:round(Tstop/Ts)
    % Plant equations: output update
    y = C*x + D(:,2)*v(ct+1);
    % Store output signals
    YY2 = [YY2,y]; %#ok<*AGROW>
    % Compute MPC law. Extracts references r(t+1),r(t+2),...,r(t+p) and
    % measured disturbances v(t),v(t+1),...,v(t+p) for previewing.
    u = mpcmove(mpcobj,xmpc,y,r(ct+2:ct+p+1),v(ct+1:ct+p+1));
    % Plant equations: state update
    x = Ad*x+Bd(:,1)*u+Bd(:,2)*v(ct+1);
end
```

Plot results.

```
figure
t = 0:Ts:Tstop;
plot(t,r(1:length(t)),'c:',t,YY1,'r-',t,YY2,'bo');
xlabel('Time');
ylabel('Plant Output');
legend({'Reference';'From sim command';'From mpcmove command'},'Location','SouthEast');
grid
```



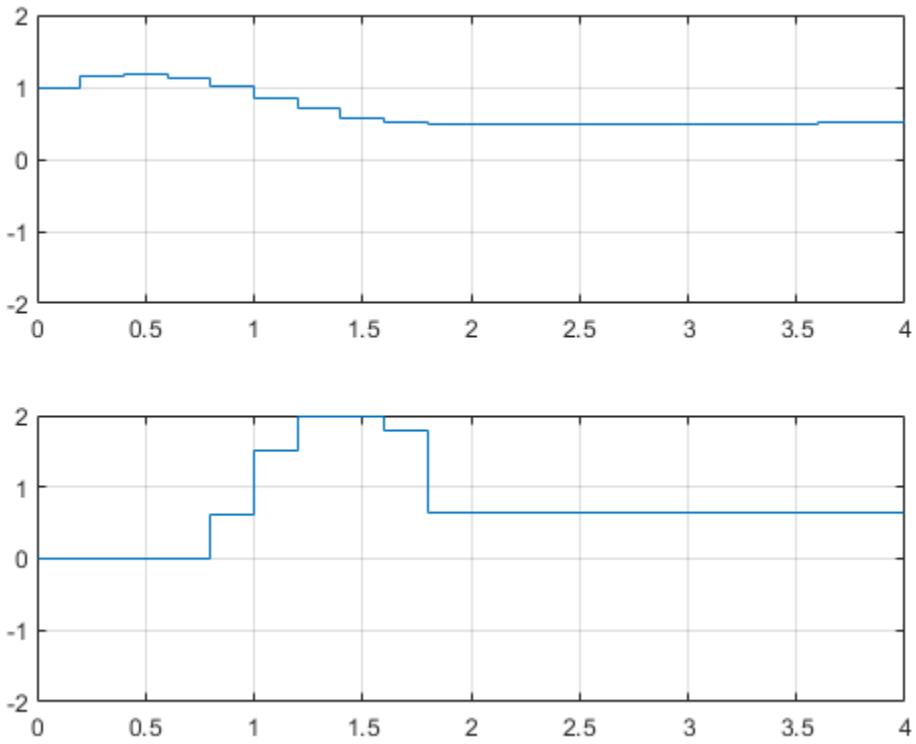
The responses are identical.

Optimal predicted trajectories are returned by `mpcmove`. Assume to you start from the current state and have a set-point change to 0.5 in 5 steps, and assume the measured disturbance has disappeared.

```
r1 = [ones(5,1);0.5*ones(p-5,1)];
v1 = zeros(p+1,1);
[~,Info] = mpcmove(mpcobj,xmpc,y,r1(1:p),v1(1:p+1));
```

Extract the optimal predicted trajectories and plot them.

```
topt = Info.Topt;
yopt = Info.Yopt;
uopt = Info.Uopt;
figure
subplot(211)
title('Optimal sequence of predicted outputs')
stairs(topt,yopt);
grid
axis([0 p*Ts -2 2]);
subplot(212)
title('Optimal sequence of manipulated variables')
stairs(topt,uopt);
axis([0 p*Ts -2 2]);
grid
```



Obtain LTI Representation of MPC Controller with Previewing

When the constraints are not active, the MPC controller behaves like a linear controller. You can get the state-space form of the MPC controller, with y, [r(t+1);r(t+2);...;r(t+p)], and [v(t);v(t+1);...;v(t+p)] as inputs to the controller.

Get state-space matrices of linearized controller.

```
LTI = ss(mpcobj,'rv','on','on');
[AL,BL,CL,DL] = ssdata(LTI);
```

Create array to store closed-loop outputs.

```
YY3 = [];
% Setup initial state of the MPC controller
x = x0;
xL = [x0;0;0];
```

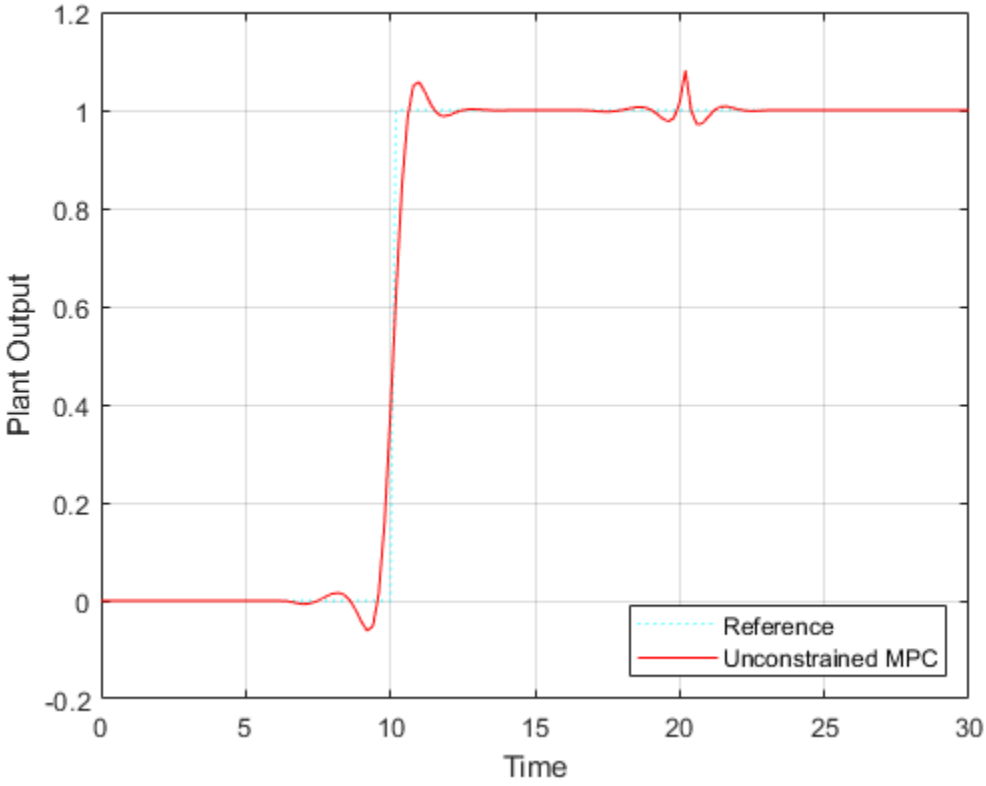
Start main simulation loop

```
for ct=0:round(Tstop/Ts)
    % Plant output update
    y = Cd*x + Dd(:,2)*v(ct+1);
    % Save output and refs value
    YY3 =[YY3,y];
    % Compute the linear MPC control action
    u = CL*xL + DL*[y;r(ct+2:ct+p+1);v(ct+1:ct+p+1)];
    % Note that the optimal move provided by MPC would be: mpcmove(MPCobj,xmpc,y,ref(t+2:t+p+1),v
```

```
    % Plant update
    x = Ad*x + Bd(:,1)*u + Bd(:,2)*v(ct+1);
    % Controller update
    xL = AL*xL + BL*[y;r(ct+2:ct+p+1);v(ct+1:ct+p+1)];
end
```

Plot results.

```
figure
plot(t,r(1:length(t)),'c:',t,YY3,'r-');
xlabel('Time');
ylabel('Plant Output');
legend({'Reference';'Unconstrained MPC'},'Location','SouthEast');
grid
```

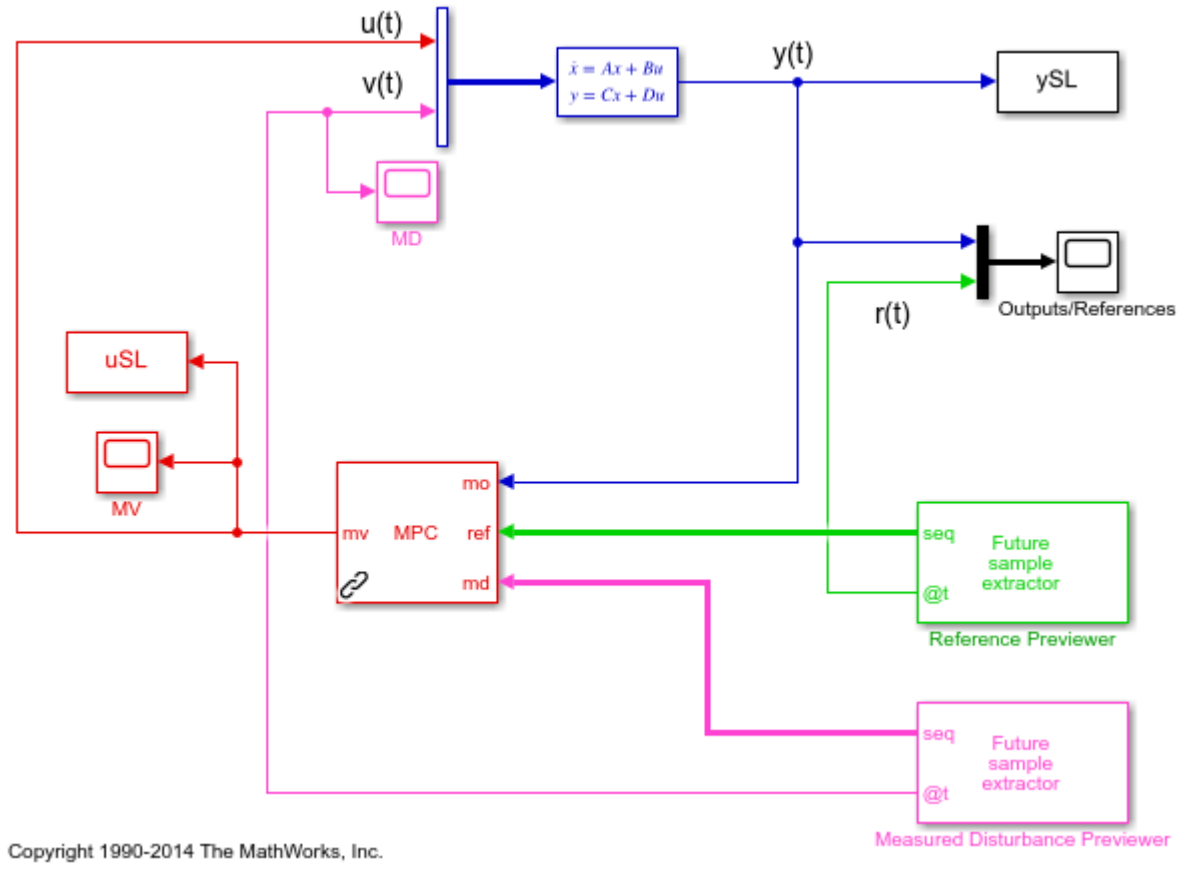


Simulate Using Simulink®

To run this example, Simulink® is required.

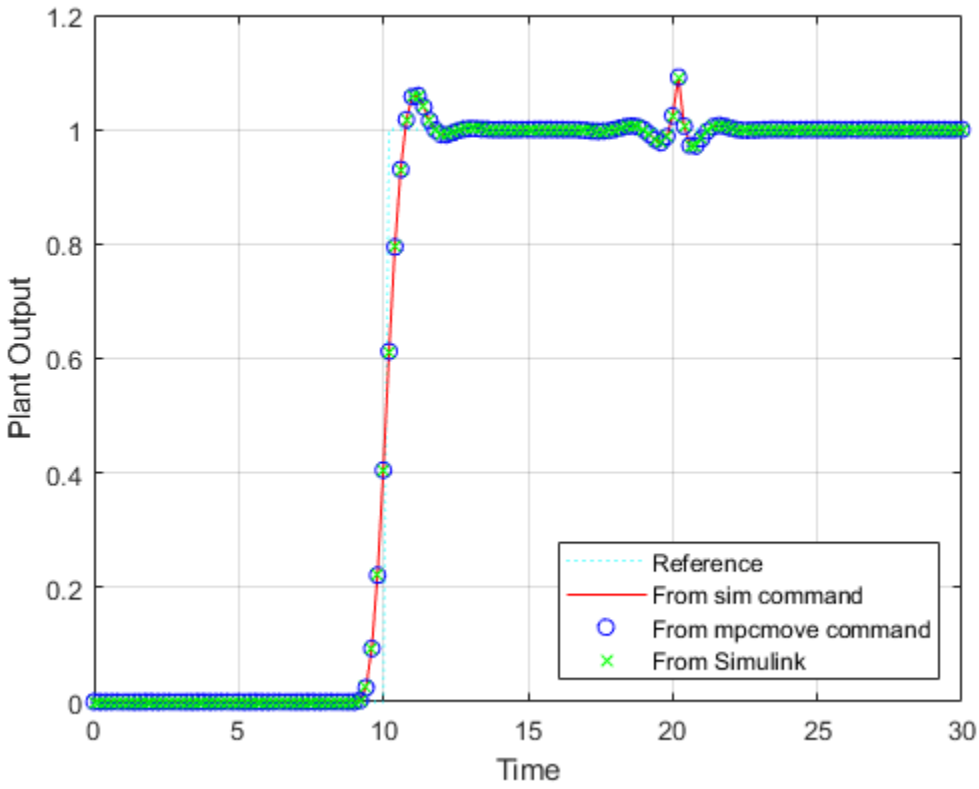
```
if ~mpccheckboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end

time = (0:Ts:(Tstop+p*Ts))'; % time vector
r = double(time>10); % reference signal
v = -double(time>20); % measured disturbance signal
```

```
% Define the reference signal in structure
ref.time = time;
ref.signals.values = r;

% Define the measured disturbance
md.time = time;
md.signals.values = v;

% Open Simulink model
mdl = 'mpc_preview';
open_system(mdl)

% Simulate the model using the Simulink |sim| command
sim(mdl,Tstop);
```



Plot results.

```
figure
t = 0:Ts:Tstop;
plot(t,r(1:length(t)),'c:',t,YY1,'r-',t,YY2,'bo',t,ySL,'gx');
xlabel('Time');
ylabel('Plant Output');
legend({'Reference';'From sim command';'From mpcmove command';'From Simulink'},'Location','SouthE
grid
```



The responses are identical.

```
bdclose('mpc_preview')
```

See Also

`mpc` | MPC Controller

More About

- “Signal Previewing” on page 5-19

Update Constraints at Run Time

To compensate for changing operating conditions, you can update constraints on plant inputs and outputs at run time. You can update the saturation limits for input and output signals as well as linear mixed input/output constraints.

Run-time constraint updating supports code generation.

Update Bounds on Input and Output Signals at Run Time

You can update the bounds on plant input and output signals at run time. To do so, first define initial signal bounds when designing your MPC controller. For more information, see “Specify Constraints” on page 1-5. If you do not specify initial bounds for a given signal, you cannot constrain that signal at run time.

To update signal bounds during a command-line simulation, at each control interval, set the corresponding properties of an `mpcmoveopt` object before calling `mpcmove`, `mpcmoveAdaptive`, or `mpcmoveMultiple`. To update:

- Manipulated variable lower and upper bounds, set the `MVMin` and `MVMax` properties, respectively.
- Output variable lower and upper bounds, set the `OutputMin` and `OutputMax` properties, respectively.

You can also update input and output bounds at run-time in Simulink for the MPC Controller, Adaptive MPC Controller, and Multiple MPC Controllers blocks. The following table lists the bounds, their associated block ports, and the block parameters to select to enable the ports.

Bounds	Port Name	Block Parameter
Lower bounds on manipulated variables	`umin`	**Lower MV Limits**
Upper bounds on manipulated variables	`umax`	**Upper MV Limits**
Lower bounds on output variables	`ymin`	**Lower OV Limits**
Upper bounds on output variables	`ymax`	**Upper OV Limits**

Connect signals to these ports that specify the run-time values of the bounds for each variable. If there is more than one manipulated variable or output variable, connect a vector signal to the corresponding ports. For example, if there are three output variables, connect a three-element vector signal to the `ymin` and `ymax` ports. If a variable is unconstrained in the controller object, then the connected signal value is ignored.

Tip For any constraint that you set to `-Inf` or `Inf`, either across the whole prediction horizon (uniform) or at individual prediction horizon steps (time-varying), the corresponding variable remains unconstrained at run time; that is, you cannot modify it.

However, you can keep a variable unconstrained, so as to not distort your offline controller design, while maintaining the ability to add constraints online. To do so, set the bound to a large value when

you create the controller. Do not use `realmax` as the large value, since doing so causes numerical issues at run time. You can then modify the constraint at run-time.

If you define time-varying constraints in your controller object, the new bounds are applied to the first finite values in the prediction horizon. All subsequent prediction horizon values adjust to maintain the same profile across the prediction horizon; that is, they change by the same amount.

For an example, see “Vary Input and Output Bounds at Run Time” on page 5-30.

Update Mixed Input/Output Constraints at Run Time

You can update mixed input/output constraints at run time. For more information on these constraints, see “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5. This feature is not supported for gain-scheduled MPC controllers.

You can update the following constraint matrices during your simulation:

- `E` — Manipulated variable constraint constant
- `F` — Controlled output constraint constant
- `G` — Mixed input/output constraint constant
- `S` — Measured disturbance constraint constant

To do so, first define initial constraints using the `setconstraint` command. You cannot add additional constraints at run time.

To update mixed input/output constraints during a command-line simulation, in each control interval set the `CustomConstraint` property of an `mpcmoveopt` object before calling `mpcmove` or `mpcmoveAdaptive`. Specify `CustomConstraint` as a structure with `E`, `F`, `G`, and `S` fields. Specify each field as an array with dimensions that match the initial constraint arrays specified using `setconstraint`.

To update mixed input/output constraints during a Simulink simulation, select the **Custom constraints** parameter of your MPC Controller or Adaptive MPC Controller block. Doing so adds `E`, `F`, `G`, and `S` input ports to the block. The `S` input port is added only if your controller has measured disturbances.

Connect matrix signals to these ports that specify the run-time values for each array. If you define `E`, `F`, `G`, or `S` in your MPC controller, you must connect a signal to the corresponding input port, and that signal must have the same dimensions as the array specified in the controller. If an array is not defined in the controller object, use a zero matrix with the correct size.

For an example that updates mixed input/output constraints for an adaptive MPC controller, see “Obstacle Avoidance Using Adaptive Model Predictive Control” on page 7-38.

See Also

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

More About

- “Tune Weights at Run Time” on page 5-35

- “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5
- “Vary Input and Output Bounds at Run Time” on page 5-30

Vary Input and Output Bounds at Run Time

This example shows how to vary input and output saturation limits in real-time control. For both command-line and Simulink® simulations, you specify updated input and output constraints at each control interval. The MPC controller then keeps the input and output signals within their specified bounds.

For more information on updating linear constraints at run time, see “Update Constraints at Run Time” on page 5-27.

Create Plant Model and MPC Controller

Define a SISO discrete-time plant with sample time `Ts`.

```
Ts = 0.1;
plant = c2d(tf(1,[1 .8 3]),Ts);
[A,B,C,D] = ssdata(plant);
```

Create an MPC controller with specified prediction horizon, `p`, control horizon, `c`, and sample time, `Ts`. Use `plant` as the internal prediction model.

```
p = 10;
m = 4;
mpcobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify controller tuning weights.

```
mpcobj.Weights.MV = 0;
mpcobj.Weights.MVrate = 0.5;
mpcobj.Weights.OV = 1;
```

For this example, the upper and lower bounds on the manipulated variable, and the upper bound on the output variable are varied at run time. To do so, you must first define initial dummy finite values for these constraints in the MPC controller object. Specify values for `MV.Min`, `MV.Max`, and `OV.Max`.

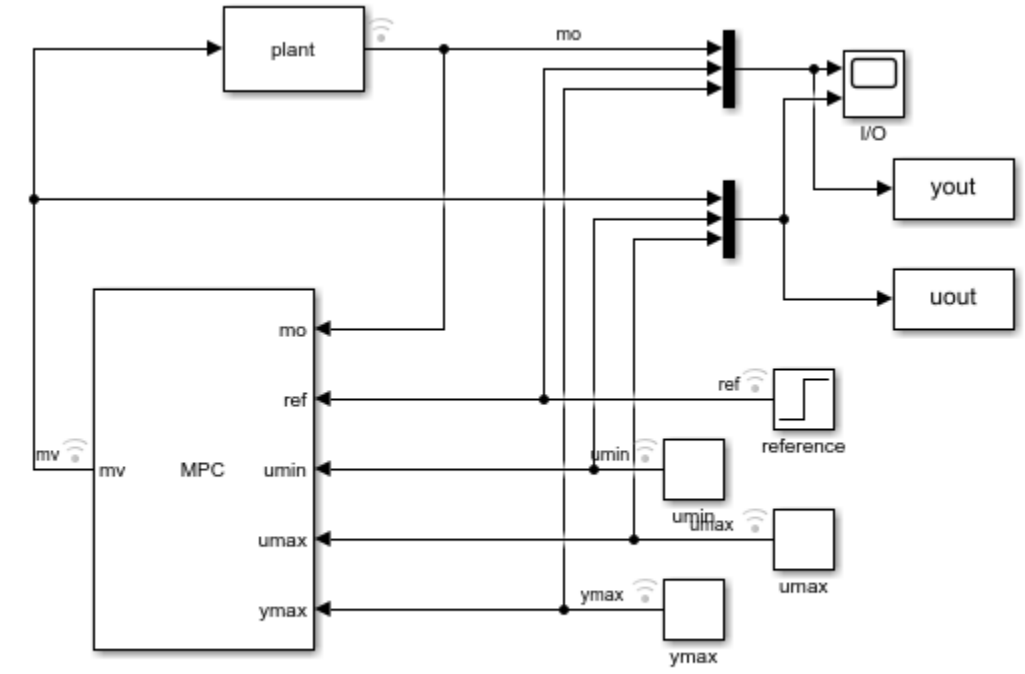
At run time, these constraints are changed using an `mpcmoveopt` object at the command line or corresponding input signals to the MPC Controller block.

```
mpcobj.MV.Min = 1;
mpcobj.MV.Max = 1;
mpcobj.OV.Max = 1;
```

Simulate Model Using Simulink

Open Simulink Model.

```
mdl = 'mpc_varbounds';
open_system(mdl)
```



Copyright 1990-2014 The MathWorks, Inc.

In this model, the input minimum and maximum constraint ports (`umin` and `umax`) and the output maximum constraint port (`ymax`)of the MPC Controller block are enabled. Since the minimum output bound is unconstrained, the `ymin` input port is disabled.

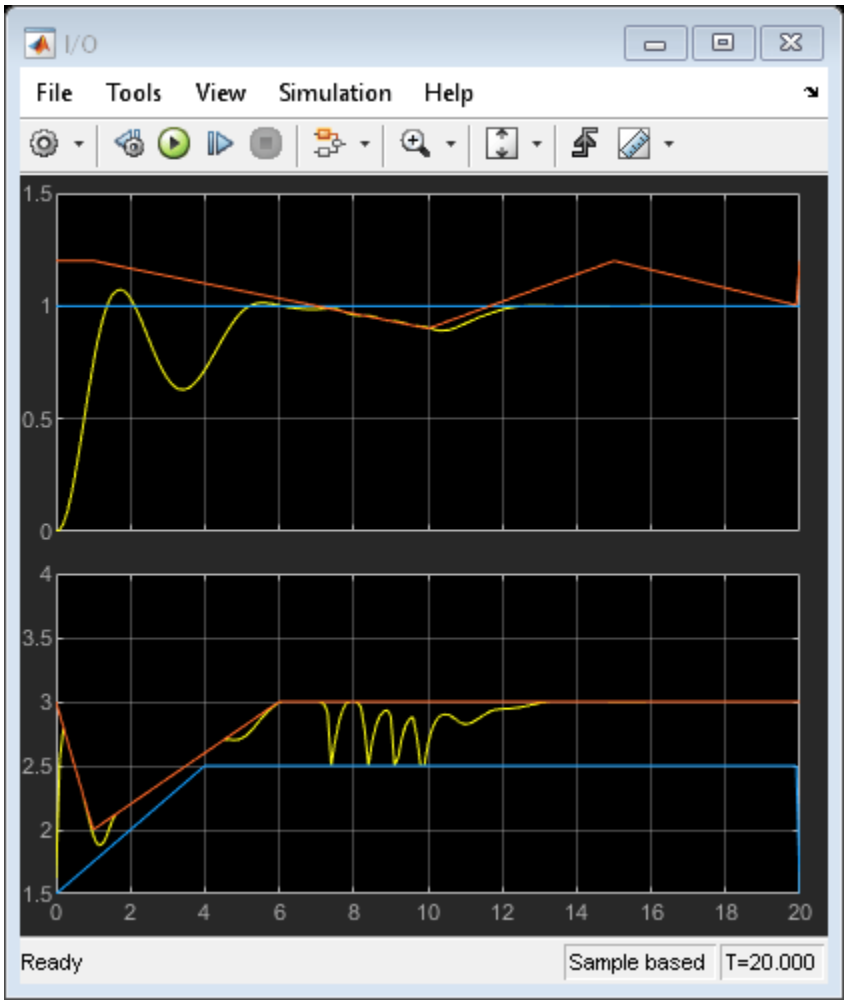
Configure the output setpoint, `ref`, and simulation duration, `Tsim`.

```
ref = 1;
Tsim = 20;
```

Run the simulation, and view the input and output responses in the I/O scope.

```
sim(mdl)
open_system([mdl '/I//O'])
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->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 measured
```



Simulate Model at Command Line

Specify the initial state of the plant and controller.

```
x = zeros(size(B,1),1);
xmpc = mpcstate(mpcobj);
```

Store the closed-loop output, manipulated variable, and state trajectories of the MPC controller in arrays YY, UU, and XX, respectively.

```
YY = [];
UU = [];
XX = [];
```

Create an `mpcmoveopt` object for specifying the run-time bound values.

```
options = mpcmoveopt;
```

Run the simulation loop.

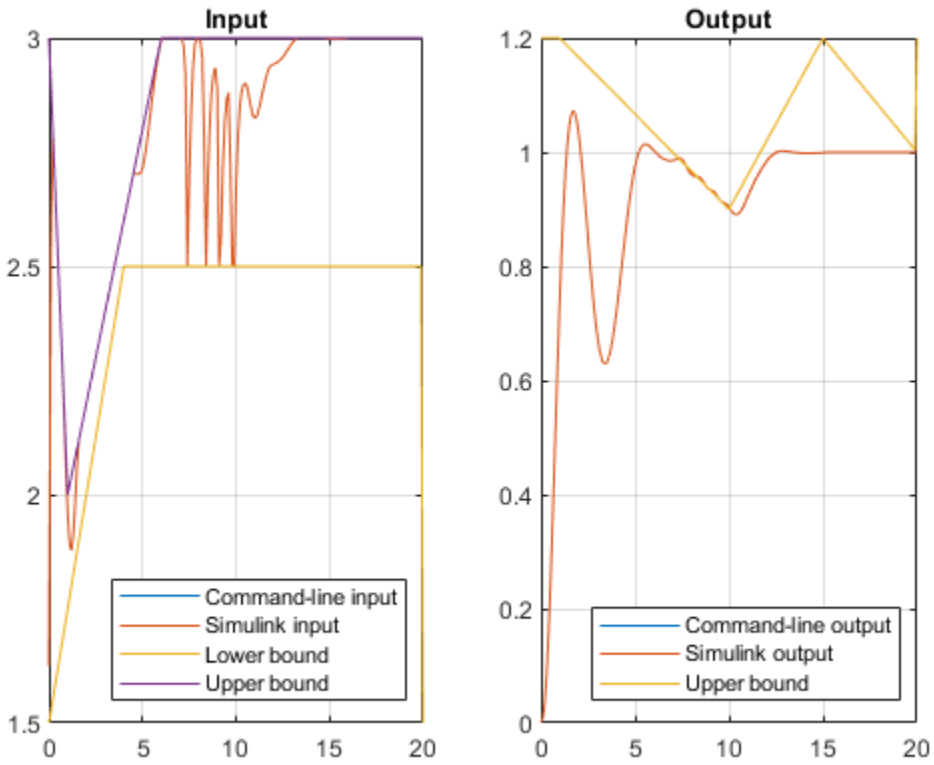
```
for t = 0:round(Tsim/Ts)
    % Store the plant state.
    XX = [XX; x];
```

```
    % Compute and store the plant output. There is no direct feedthrough
    % from the input to the output.
    y = C*x;
    YY = [YY; y'];

    % Get the reference signal value from the data output by the Simulink
    % simulation.
    ref = yout.Data(t+1,2);

    % Update the input and output bounds. For consistency, use the
    % constraint values output by the Simulink simulation.
    options.MVMin = uout.Data(t+1,2);
    options.MVMax = uout.Data(t+1,3);
    options.OutputMax = yout.Data(t+1,3);

    % Compute the MPC control action.
    u = mpcmove(mpcobj,xmpc,y,ref,[],options);

    % Update the plant state and store the input signal value.
    x = A*x + B*u;
    UU = [UU; u'];
end
```

Compare Simulation Results

Plot the input and output signals from both the Simulink and command-line simulations along with the changing input and output bounds.

```
figure
subplot(1,2,1)
plot(0:Ts:Tsim,[UU uout.Data(:,1) uout.Data(:,2) uout.Data(:,3)])
grid
title('Input')
legend('Command-line input','Simulink input','Lower bound',...
    'Upper bound','Location','Southeast')
subplot(1,2,2)
plot(0:Ts:Tsim,[YY yout.Data(:,1) yout.Data(:,3)])
grid
title('Output')
legend('Command-line output','Simulink output','Upper bound',...
    'Location','Southeast')
```



The results of the command-line and Simulink simulations are the same. The MPC controller keeps the input and output signals within the specified bounds as the constraints change throughout the simulation.

```
bdclose(mdl)
```

See Also

MPC Controller

More About

- “Update Constraints at Run Time” on page 5-27

Tune Weights at Run Time

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, as described in this topic.

In Simulink, the following blocks support online 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 and Multiple Explicit MPC Controllers blocks do not support online tuning because a weight change requires a complete revision of the explicit MPC control law, which is computationally intensive.

To tune weights during command-line simulations, first create an `mpcmoveopt` object, and specify the corresponding tuning weight properties. Then pass this object to either `mpcmove`, `mpcmoveAdaptive`, or `mpcmoveMultiple`.

This table lists the weights that you can tune at run time and their corresponding Simulink block ports and `mpcmoveopt` properties. For more information on tuning weights, including tuning tips, see “Tune Weights” on page 1-43.

Tune weights for	Simulink Block Port	mpcmoveopt Property
Output variables	`y.wt`	`OutputWeights`
Manipulated variables	`u.wt`	`MVWeights`
Manipulated variable increment	`du.wt`	`MVRateWeights`
Slack variable for constraint softening	`ecr.wt`	`ECRWeight`

For the output variable, manipulated variable, and manipulated variable increment weights, you can specify time-varying weights at run time; that is, tuning weights that vary over the prediction horizon. To do so, specify the tuning weights as arrays.

Note To vary weights at run time, you must specify time-varying weights when you create your MPC controller object. In other words, if you configure your controller to use constant weights over the prediction horizon, you cannot specify time-varying weights at run time.

See Also

More About

- “Signal Previewing” on page 5-19
- “Tuning MPC Controller Weights at Run-Time” on page 5-36

Tuning MPC Controller Weights at Run-Time

This example shows how to vary the weights on outputs, inputs, and ECR slack variable for soft constraints at run-rime, using either Simulink® or `mpcmove`.

The weights specified in the MPC object are overridden by the weights supplied to the MPC Controller block. If a weight signal is not connected to the MPC Controller block, then the corresponding weight is the one specified in the MPC object.

Define Plant Model

Define a multivariable discrete-time linear system with no direct I/O feedthrough, and assume input #4 is a measured disturbance and output #4 is unmeasured.

```
Ts = 0.1; % sampling time
plant = tf({1,[1 1],5,2;3,[1 5],1,0;0,0,1,[1 1];2,[1 -1],0,0},...
    {[1 1 1],[1 3 4 5],[1 10],[1 5];
      [1 1],[1 2],[1 2 8],[1 1];
      [1 2 1],[1 3 1 1],[1 1],[1 2];
      [1 1],[1 3 10 10],[1 10],[1 1]});
plant = c2d(ss(plant),Ts);
plant.D = 0;

% display size of the plant.
size(plant)

State-space model with 4 outputs, 4 inputs, and 17 states.
```

Design MPC Controller

Specify input and output signal types.

```
plant = setmpcsignals(plant,'MD',4,'UO',4);
% Create the controller object with sampling period, prediction and control
% horizons:
p = 20;                                     % Prediction horizon
m = 3;                                      % Control horizon
mpcobj = mpc(plant,Ts,p,m);

% note that default weights are assumed on inputs, input rates, and outputs

-->Assuming unspecified input signals are manipulated variables.
-->Assuming unspecified output signals are measured outputs.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 y2 y3 and zero weight for output(s) y4
```

Specify MV constraints.

```
mpcobj.MV(1).Min = -6;
mpcobj.MV(1).Max = 6;
mpcobj.MV(2).Min = -6;
mpcobj.MV(2).Max = 6;
mpcobj.MV(3).Min = -6;
mpcobj.MV(3).Max = 6;
```

Define Time-Varying Signals using Structure Format

Define reference signal.

```
Tstop = 10;
ref = [1 0 3 1];
r = struct('time',(0:Ts:Tstop)');
N = numel(r.time);
r.signals.values=ones(N,1)*ref;
```

Define measured disturbance.

```
v = 0.5;
```

OV weights are linearly increasing with time, except for output #2 that is not weighted.

```
ywt.time = r.time;
ywt.signals.values = (1:N)'*[.1 0 .1 .1];
```

MV rate weights are decreasing linearly with time.

```
duwt.time = r.time;
duwt.signals.values = (1-(1:N)/2/N)'*[.1 .1 .1];
```

ECR weight increases exponentially with time.

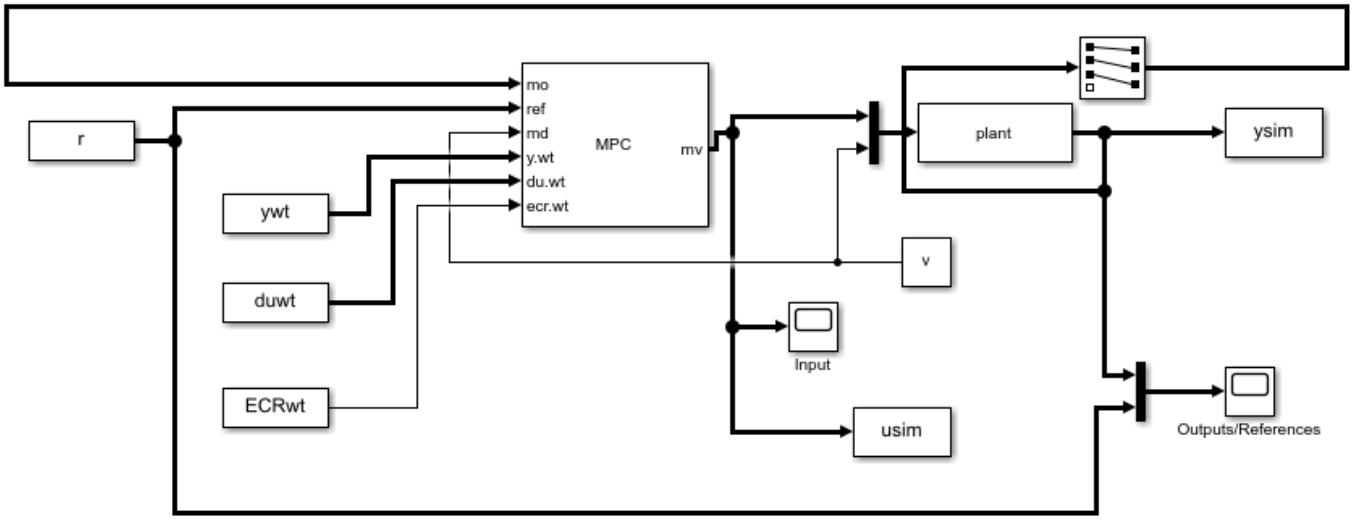
```
ECRwt.time = r.time;
ECRwt.signals.values = 10.^(2+(1:N)'/N);
```

Simulate Using Simulink®

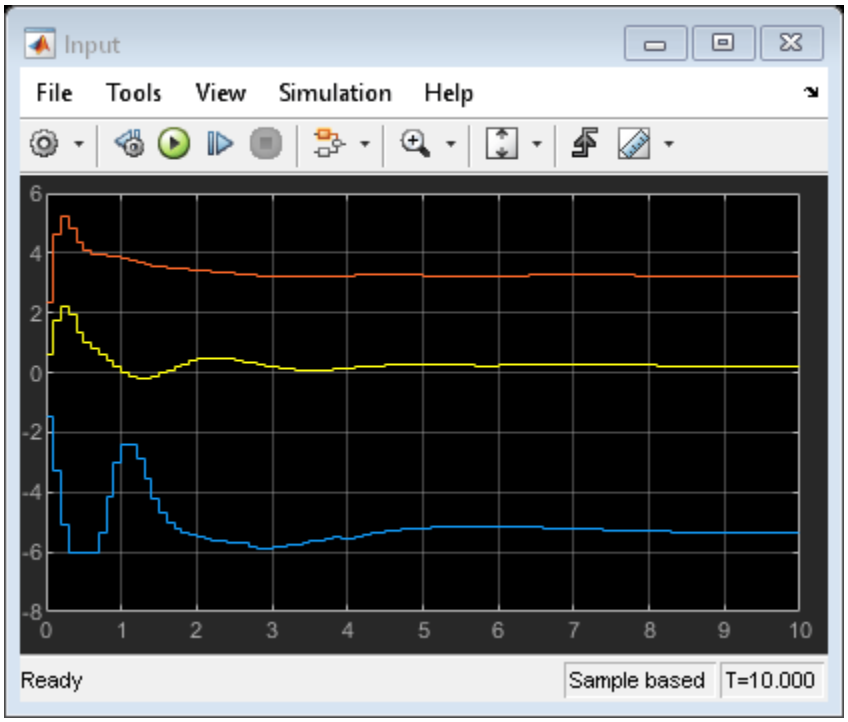
Start simulation.

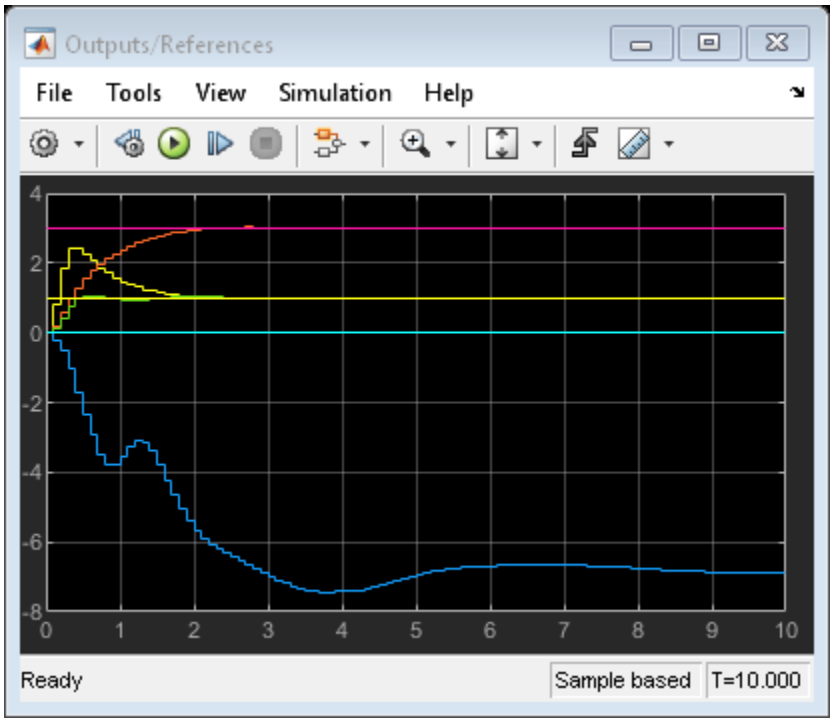
```
mdl = 'mpc_onlinetuning';
open_system(mdl);                    % Open Simulink(R) Model
sim(mdl);                            % Start Simulation
```

```
-->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.
-->Assuming output disturbance added to measured output channel #3 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



Copyright 1990-2014 The MathWorks, Inc.





Simulate Using MPCMOVE Command

Define real plant and MPC state object.

```
[A,B,C,D] = ssdata(plant);
x = zeros(size(plant.B,1),1);   % Initial state of the plant
xmpc = mpcstate(mpcobj);        % Handle to state of the MPC controller
```

Store the closed-loop MPC trajectories in arrays YY,UU,XX.

```
YY = []; UU = []; XX = [];
```

Use `mpcmoveopt` object to provide weights at run-time.

```
options = mpcmoveopt;
```

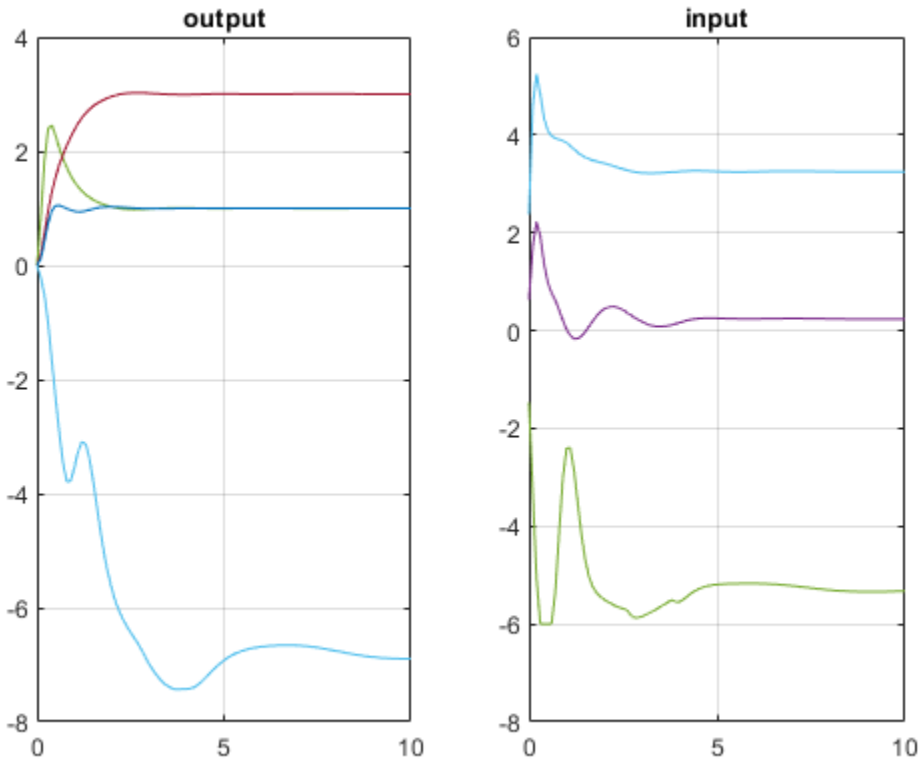
Start simulation.

```
for t = 0:N-1,
    % Store states
    XX = [XX,x]; %#ok<*AGROW>
    % Compute and store plant output (no feedthrough from MV to Y)
    y = C*x+D(:,4)*v;
    YY = [YY;y'];
    % Obtain reference signal
    ref = r.signals.values(t+1,:)';
    % Update |mpcmoveopt| object with run-time weights
    options.MVRateWeight = duwt.signals.values(t+1,:);
    options.OutputWeight = ywt.signals.values(t+1,:);
    options.ECRWeight = ECRwt.signals.values(t+1,:);
    % Compute and sore control action
    u = mpcmove(mpcobj,xmpc,y(1:3),ref,v,options);
```

```
    UU = [UU;u'];
    % Update plant states
    x = A*x + B(:,1:3)*u + B(:,4)*v;
end
```

Plot and Compare Simulation Results.

```
figure(1);
clf;
subplot(121)
plot(0:Ts:Tstop,[YY ysim])
grid
title('output')
subplot(122)
plot(0:Ts:Tstop,[UU usim])
grid
title('input')
```



Compare input and output signals.

```
norm(UU-usim)
norm(YY-ysim)

% results are identical except for numerical errors.


ans =
```

```
   5.5397e-11

ans =

   1.1288e-11

bdclose(mdl);
```

See Also

MPC Controller

More About

- “Tune Weights at Run Time” on page 5-35

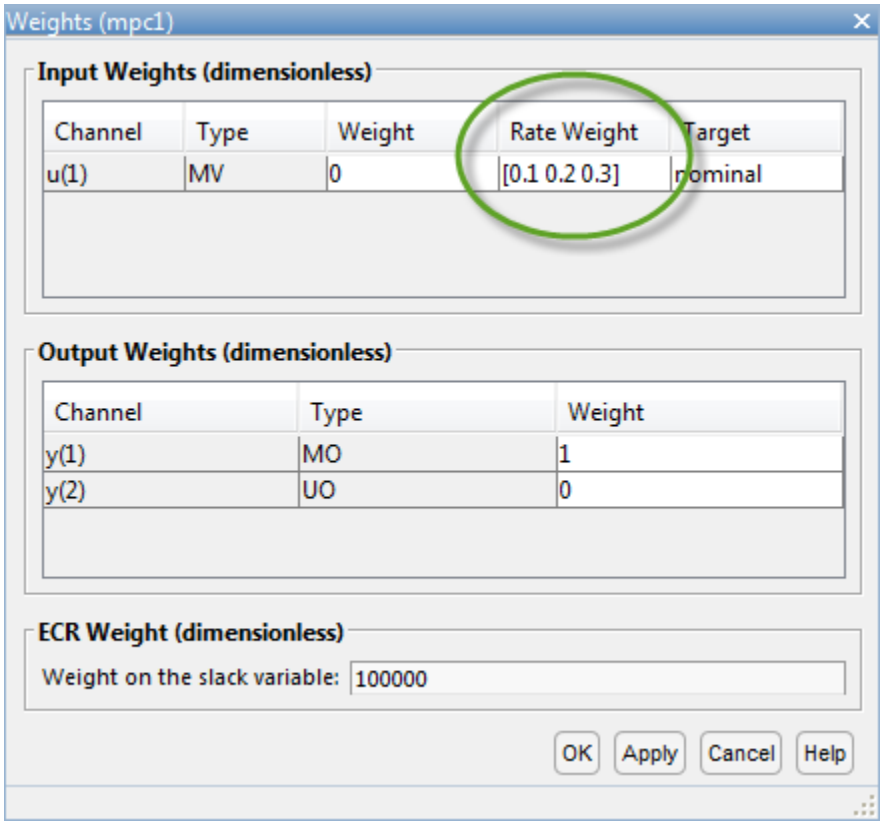
Setting Time-Varying Weights and Constraints with MPC Designer

Time-Varying Weights

As explained in “Optimization Problem” on page 2-7, 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 3-18.

You can specify time-varying weights in **MPC Designer**. 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.

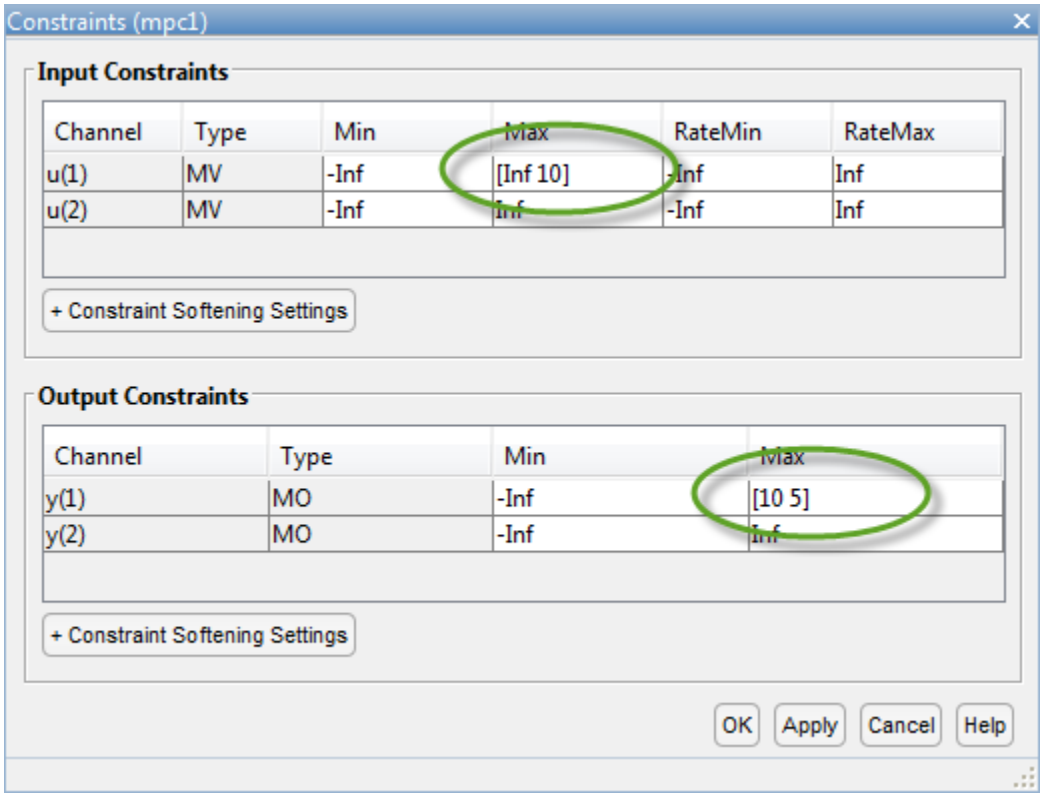
Time-Varying Constraints

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 nth 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 **MPC Designer**. In the Constraints dialog box, specify a vector for each time-varying constraint.



See Also

More About

- “Optimization Problem” on page 2-7
- “Terminal Weights and Constraints” on page 3-18

- “Tune Weights at Run Time” on page 5-35
- “Update Constraints at Run Time” on page 5-27
- “Vary Input and Output Bounds at Run Time” on page 5-30

Adjust Horizons at Run Time

For both implicit and adaptive MPC controllers, you can adjust the prediction and control horizons while the controller operates. Doing so can be useful for:

- Efficiently evaluating different horizon choices at run time during prototyping
- Adjusting horizons without redeployment after plant dynamics change significantly, such as in a batch process

Adjust Horizons in MATLAB

To adjust the horizons at run time from the command line, at each control interval, specify the following properties of the `mpcmoveopt` object.

- `PredictionHorizon` — Run-time prediction horizon signal, specified as a positive integer
- `ControlHorizon` — Run-time control horizon signal, specified as a positive integer or a vector of positive integers

You can then pass the `mpcmoveopt` object to either `mpcmove` or `mpcmoveAdaptive`.

Adjust Horizons in Simulink

In Simulink, to adjust the horizons for an MPC Controller or Adaptive MPC Controller block, select the **Adjust prediction horizon and control horizon at run time** parameter. Doing so adds the following input ports to the block:

- `p` — Run-time prediction horizon signal, specified as a scalar integer signal
- `m` — Run-time control horizon signal, specified as a scalar or vector signal

You must specify the maximum prediction horizon using the **Maximum prediction horizon** parameter. Doing so ensures that the optimal sequence output ports of the block (`mv.seq`, `x.seq`, and `y.seq`) have constant sizes with p_{max}+1 rows, where p_{max} is the maximum prediction horizon.

Code Generation

Run-time horizon tuning supports code generation in both MATLAB and Simulink. Generating code for a controller that supports run-time horizon changes allows you to tune your horizon values on your deployed controller hardware.

After tuning the horizon values, to improve the computational efficiency of your final deployed controller, you can generate code for a constant-horizon controller using the tuned values.

Deploying your controller with run-time horizon tuning enabled significantly increases the computational load and memory footprint of your MPC application. If you plan to use run-time horizon tuning only for prototyping to find the proper horizon values, after tuning, ensure that this feature is disabled. You can then generate code with a constant-horizon controller using the tuned values.

If your controller uses manipulated variable blocking and you generate code for your controller, the size of the control horizon vector must remain constant at run-time. In this case, you can still tune the values within the control horizon vector.

Note To generate code for a controller that uses run-time horizon tuning, your deployed hardware target must support dynamic memory allocation. For example, if your embedded system does not support the `malloc` C function, then the generated code will not run.

To generate code in MATLAB, set the `'UseVariableHorizon'` name-value argument of `getCodeGenerationData` to `true`. At each control interval, you can then specify the horizons before calling `mpcmoveCodeGeneration`.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcobj,'UseVariableHorizon',true);
...
onlineData.Horizons.p = 10;
onlineData.Horizons.m = 3;
[u,stateData] = mpcmoveCodeGeneration(configData,stateData,onlineData);
```

Effect on Time-Varying Controller Parameters and Signals

If your application uses controller parameters or signals that vary over the prediction horizon, adjusting the prediction horizon at run time affects the behavior of these time-varying parameters.

Constraints

If you define time-varying constraints in your controller object, the profile of the constraints across the prediction horizon does not change at run time. If your run-time prediction horizon value p_r is:

- Greater than the length of the constraint profile specified in your controller, then the controller uses the final value of the constraint profile for the remainder of the prediction horizon
- Less than the length of the constraint profile specified in your controller, then the controller truncates the constraint profile after p_r steps

For more information on adjusting constraints, see “Update Constraints at Run Time” on page 5-27 and “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42.

Tuning Weights and Preview Signals

To vary tuning weights or specify signal previews across the prediction horizon, you specify signal arrays where the rows correspond to prediction horizon steps.

If you adjust the prediction horizon at run time, it is best practice to define the weight and preview arrays with N rows, where N is based on the maximum prediction horizon p_{max} according to the following table.

At run time, you specify the first p_r rows of each signal array (p_r+1 rows for measured disturbances). The controller ignores any extra rows in the arrays.

Signal Type	Command-Line Usage	Block Input Port	Maximum Number of Rows *N*
MV tuning weights	`MVWeights` property of `mpcmoveopt`	`u.wt`	p_{ma} **1** x
MV rate weights	`MVRateWeights` property of `mpcmoveopt`	`du.wt`	p_{max}

Signal Type	Command-Line Usage	Block Input Port	Maximum Number of Rows *N*
Output weights	`OutputWeights` property of `mpcmoveopt`	`y.wt`	p_{max}
Reference preview	Input to `mpcmove` or `mpcmoveAdaptive`	`ref`	p_{max}
Measured disturbance preview	Input to `mpcmove` or `mpcmoveAdaptive`	`md`	p_{max}+1

If you specify any signal array with fewer than p_r rows (p_r+1 rows for measured disturbances), the controller uses the values from the final row for the remainder of the prediction horizon.

For more information on tuning weights, see “Tune Weights at Run Time” on page 5-35 and “Setting Time-Varying Weights and Constraints with MPC Designer” on page 5-42.

For more information on previewing reference or measured disturbance signals, see “Signal Previewing” on page 5-19.

Model and Nominal Conditions

For a linear time-varying MPC controller, you vary the plant model and nominal conditions across the prediction horizon by passing arrays to `mpcmoveAdaptive` or the Adaptive MPC Controller block, where the first element in each array is the current value and each additional array element corresponds to a prediction horizon step.

If you adjust the prediction horizon at run time, it is best practice to define the plant model and nominal condition arrays with p_{max}+1 elements. At run time, you specify the first p_r+1 elements of each array and the controller ignores the extra elements.

If you specify either array with fewer than p_r+1 elements, the controller uses the final element for the remainder of the prediction horizon.

For more information on linear time-varying MPC, see “Time-Varying MPC” on page 7-49.

See Also

Blocks
MPC Controller | Adaptive MPC Controller

Functions
`mpcmoveopt`

More About

- “Choose Sample Time and Horizons” on page 1-2
- “Manipulated Variable Blocking” on page 3-44
- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Evaluate Control Performance Using Run-Time Horizon Adjustment

This example shows how to adjust prediction and control horizons at run-time to evaluate controller performance without recreating the controller object or regenerating the code.

Overview of Prediction and Control Horizon Selection

Prediction and control horizons, together with controller sample time, are determined typically before other MPC settings such as constraints and weights are designed.

There are certain guidelines to help choose the sample time `Ts`, prediction horizon `p`, and control horizon `m`. For example, assume you want to determine how far the controller should look into the future. In theory, prediction time should be long enough to capture the dominant dynamic behavior of the plant but not any longer so as to avoid wasting resources used in computation. In practice, you often start with a small value and gradually increase it to see how control performance improves. When it plateaus, stop.

Control horizon determines how many decision variables MPC uses in optimization. If the value is too small, you don't have enough degrees of freedom to achieve a satisfactory performance. On the other hand, if the value is too large, both computation load and memory footprint increase significantly with little performance improvement. Therefore, it is another place you want to try different values and compare the results.

In this example, we demonstrate how to adjust prediction and control horizons of an MPC Controller block using its inports and compare control performance after multiple runs of simulation without recreating MPC controller object used by the block. If the block is running on an embedded system, you can adjust the horizons in real-time too, without regenerating and redeploying the code.

To run this example, Simulink® and Simulink Control Design™ are required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

Linearizing the Nonlinear Plant at Nominal Operating Point

The single-input-single-output nonlinear plant is implemented in Simulink model `mpc_nloffsets`. At the nominal operating point, the plant is at steady state with output of -0.5.

```
plant_mdl = 'mpc_nloffsets';
```

Use the `operspec` command from Simulink Control Design to create an operating point specification object with the desired output value fixed at steady state.

```
op = operspec(plant_mdl);
op.Outputs.Known = true;
op.Outputs.y = -0.5;
```

Use the `findop` command from Simulink Control Design to obtain the nominal operating point.

```
[op_point, op_report] = findop(plant_mdl,op);


 Operating point search report:
---------------------------------

opreport =


 Operating point search report for the Model mpc_nloffsets.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</s

(1.) mpc_nloffsets/Integrator
    -Inf    0.59453    Inf      0       1.0258e-13      0
(2.) mpc_nloffsets/Integrator2
    -Inf     2.1891    Inf      0      -1.0989e-09      0

Inputs:
----------
    <strong>Min</strong>        <strong>u</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>

(1.) mpc_nloffsets/In1
    -Inf    -1.1806    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>      <strong>Max</strong>
    <strong>____</strong>    <strong>____</strong>    <strong>____</strong>

(1.) mpc_nloffsets/Out1
    -0.5    -0.5    -0.5
```

Use the `linearize` command from Simulink Control Design to linearize the plant at the nominal operating condition.

```
plant = linearize(plant_mdl, op_point);
```

Obtain nominal plant states, output and input.

```
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = op_report.Outputs.y;
u0 = op_report.Inputs.u;
```

The linearized plant is underdamped second order system. Using the `damp` command, we can find out the dominant time constant of the plant, which is about 1.7 seconds.

```
damp(plant)

          Pole              Damping       Frequency      Time Constant
                                       (rad/seconds)      (seconds)
```

```
 -5.95e-01 + 1.84e+00i     3.07e-01       1.94e+00         1.68e+00
 -5.95e-01 - 1.84e+00i     3.07e-01       1.94e+00         1.68e+00
```

Designing Default MPC Controller

A simple guideline recommends that the prediction time should at least cover the dominant time constant (1.7 seconds) and control horizon is 10%~20% of the prediction horizon. Therefore, if we choose sample time of 0.1, the prediction horizon should be around 17. This gives us a starting point to choose the default horizons

```
Ts = 0.1;
p = 20;
m = 4;
mpcobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Set nominal values in the controller.

```
mpcobj.Model.Nominal = struct('X', x0, 'U', u0, 'Y', y0);
```

Set MV constraint.

```
mpcobj.MV.Max = 2;
mpcobj.MV.Min = -2;
```

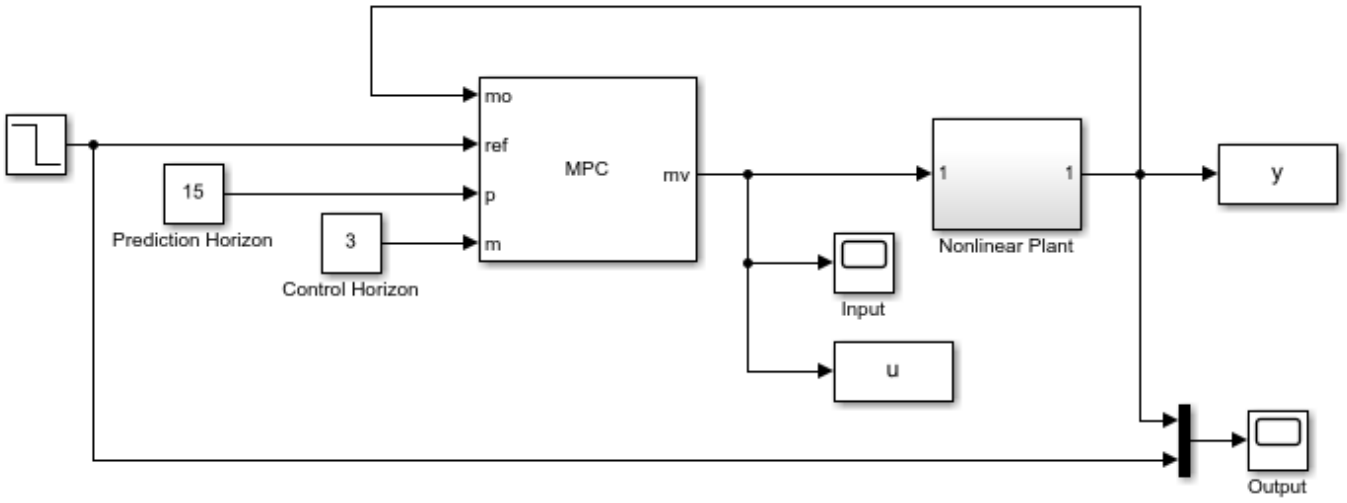
Since there is little noise in the plant, we reduce the noise model gain to make the default Kalman filter more aggressive.

```
mpcobj.Model.Noise = 0.1;
```

Comparing Performance Between Different Prediction Horizon Choices

The `mpc_onlineHorizons` model implements the closed-loop control system. Our goal to track a -0.2 step change in the reference signal with minimum overshoot. We also want the settling time to be less than 5 seconds.

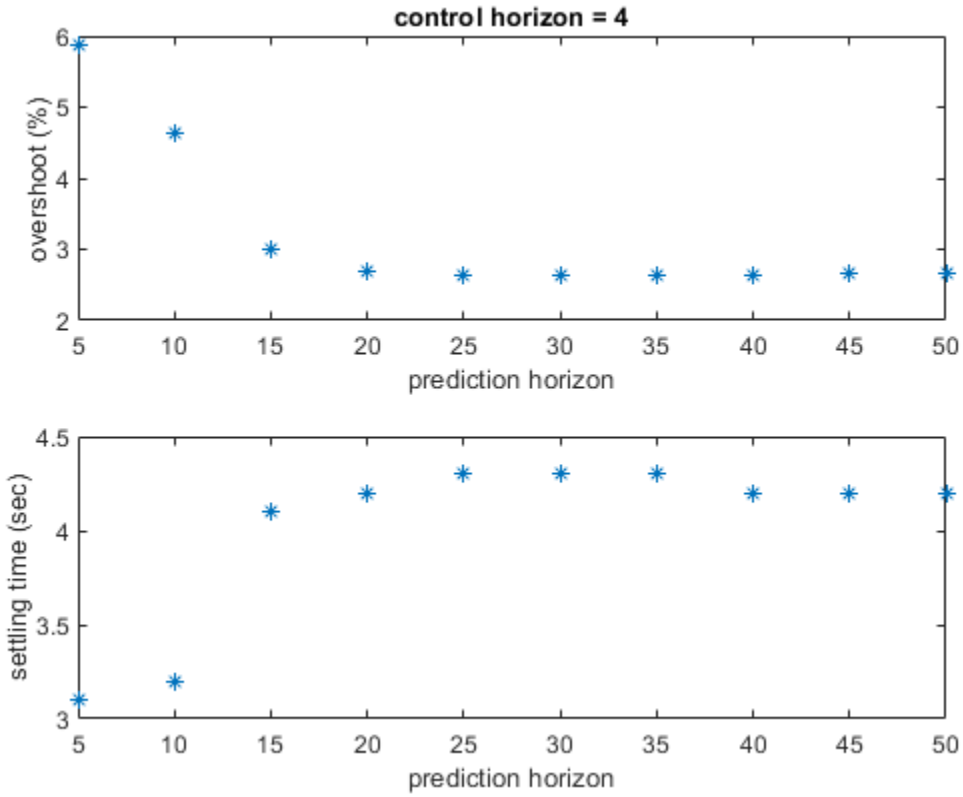
```
r0 = -0.7;
mdl = 'mpc_onlineHorizons';
open_system(mdl)
```



In the model, the MPC block has two inports where we can connect prediction horizon signal and control horizon signal. In each simulation, we vary the prediction horizon value (from 5 to 50) while keeping the control horizon at 4. We measure both the overshoot (%) and settling time (sec) from the saved simulation results. Note that the MPC controller object is not changed. Instead, the new horizon values are supplied as input signals at run-time.

```
p_choices = 5:5:50;
set_param([mdl '/Control Horizon'],'Value','4')
for p = p_choices
    set_param([mdl '/Prediction Horizon'],'Value',num2str(p))
    sim(mdl,20)
    settling_timeP(p/5) = ...
        find((abs(y.signals.values-r0)<0.01)&(abs([0;diff(y.signals.values)])<0.001),1,'first')*T
    if r0>y0
        overshootP(p/5) = abs((max(y.signals.values)-r0)/r0)*100;
    else
        overshootP(p/5) = abs((min(y.signals.values)-r0)/r0)*100;
    end
end
figure
subplot(2,1,1)
plot(p_choices,overshootP,'*')
xlabel('prediction horizon')
ylabel('overshoot (%)')
title('control horizon = 4')
subplot(2,1,2)
plot(p_choices,settling_timeP,'*')
ylabel('settling time (sec)')
xlabel('prediction horizon')
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white noise.
```



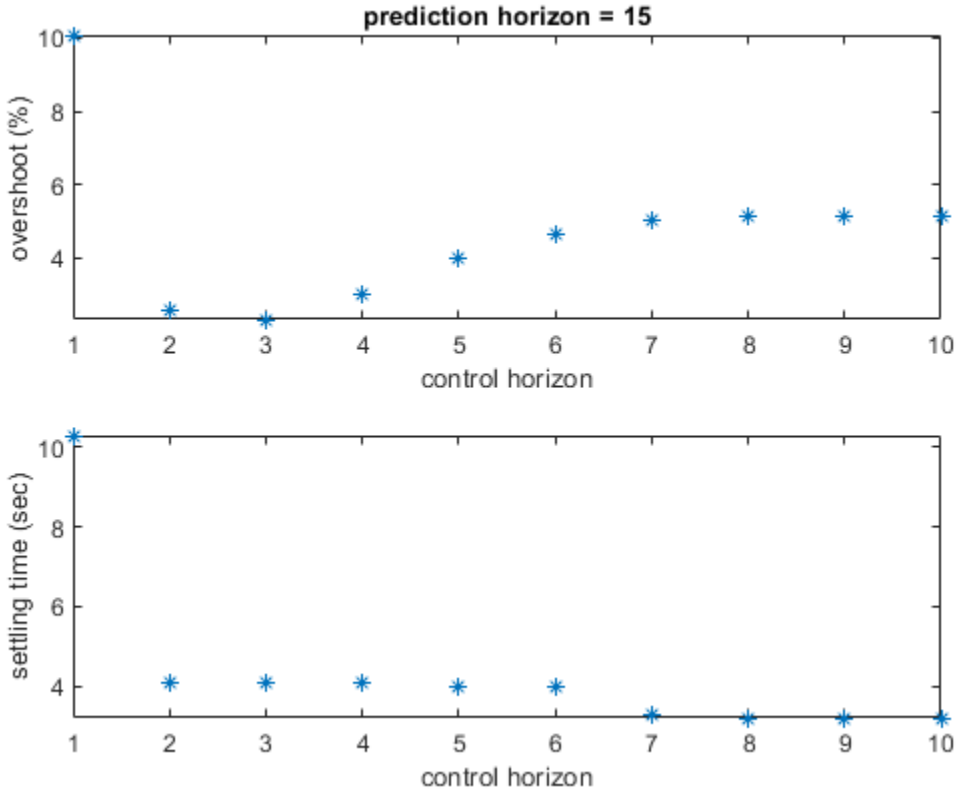
As the two plots show above, when prediction horizon increases from 5 to 15, the overshoot drops from 6% to 3% and settling time increases from 3 seconds to 4 seconds. After that, however, both overshoot and settling time remain more or less the same. In addition all the settling time values satisfy the upper bound of 5 seconds. Therefore, we choose the prediction horizon of 15, because it is the smallest value to achieve satisfactory performance by forming the smallest optimization problem.

Comparing Performance Between Different Control Horizon Choices

After we choose the prediction horizon, we use the same setup to evaluate different control horizon choices. In each simulation, we vary the control horizon (from 1 to 10) while keeping the prediction horizon at 15.

```
c_choices = 1:10;
set_param([mdl '/Prediction Horizon'],'Value','15')
for c = c_choices
    set_param([mdl '/Control Horizon'],'Value',num2str(c))
    sim(mdl,20)
    settling_timeC(c) = ...
        find((abs(y.signals.values-r0)<0.01)&(abs([0;diff(y.signals.values)])<0.001),1,'first')*T
    if r0>y0
        overshootC(c) = abs((max(y.signals.values)-r0)/r0)*100;
    else
        overshootC(c) = abs((min(y.signals.values)-r0)/r0)*100;
    end
end
figure
subplot(2,1,1)
```

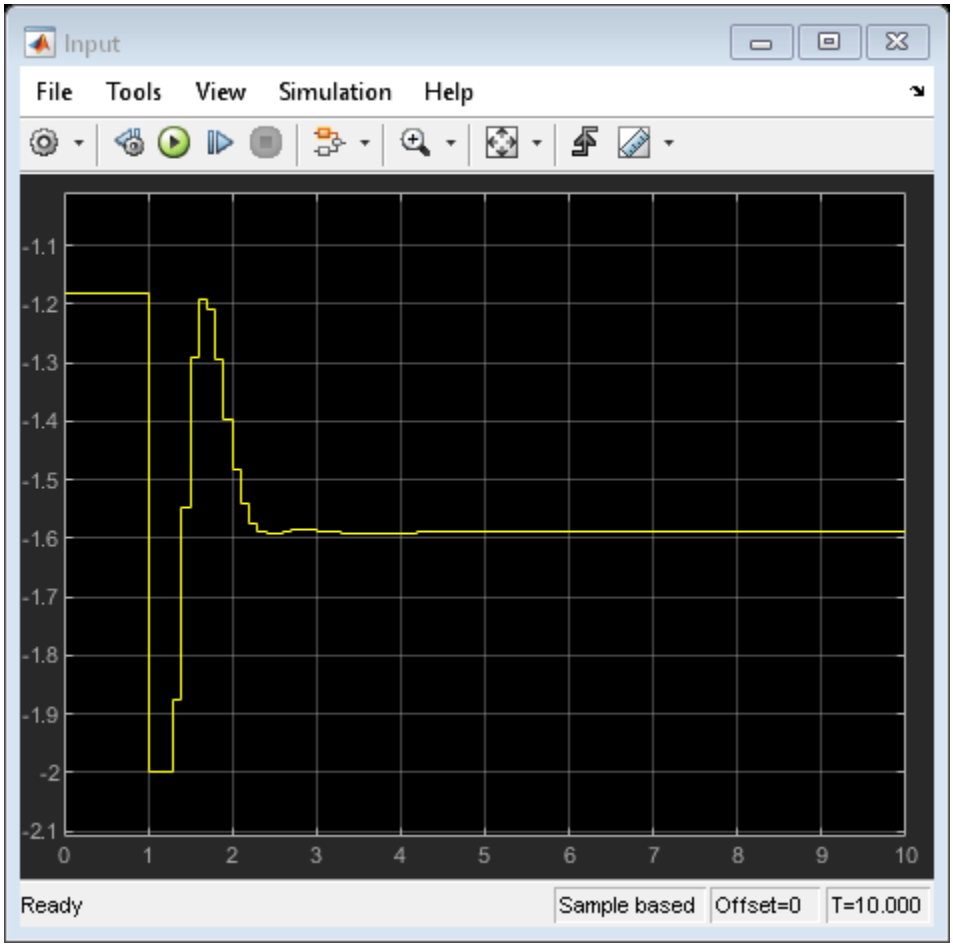
```
plot(c_choices,overshootC,'*')
xlabel('control horizon')
ylabel('overshoot (%)')
title('prediction horizon = 15')
subplot(2,1,2)
plot(c_choices,settling_timeC,'*')
xlabel('control horizon')
ylabel('settling time (sec)')
```

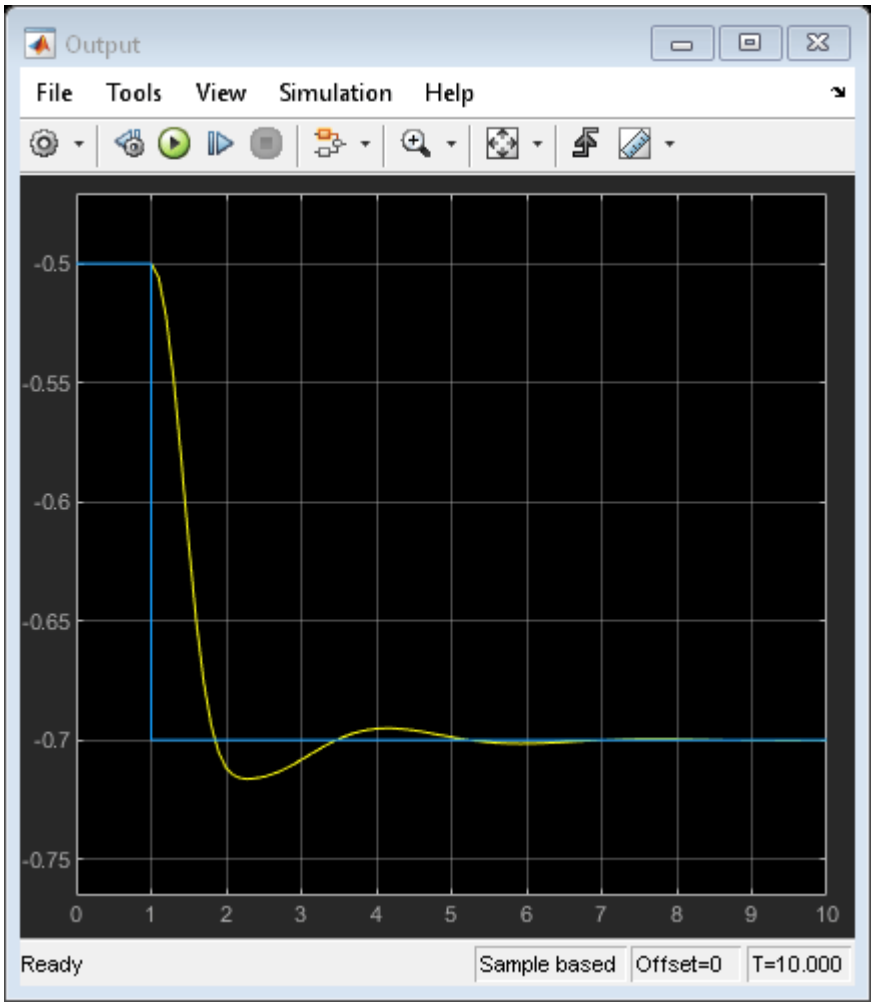


As the two plots show above, when control horizon increases from 1 to 3, the overshoot drops from 10% to 2%. After that, it increases back to 5% as control horizon grows from 4 to 10. The explanation is that when control horizon is 1, the controller doesn't have enough degrees of freedom to achieve reasonable response. When control horizon is 4 or beyond, the controller has more decision variables such that the first optimal move often becomes more aggressive and thus results in larger overshoot but shorter settling time. In this example, since the main control goal is to achieve minimum overshoot, we choose 3 as control horizon.

The model is simulated with prediction horizon = 15 and control horizon = 3. Recall that our original design choice is prediction horizon = 20 and control horizon = 4 based on a simple guideline, which is close to our final choice.

```
set_param([mdl '/Prediction Horizon'],'Value','15')
set_param([mdl '/Control Horizon'],'Value','3')
open_system([mdl '/Input'])
open_system([mdl '/Output'])
sim(mdl)
```


Input
File
Tools
View
Simulation
Help
-1.1
-1.2
-1.3
-1.4
-1.5
-1.6
-1.7
-1.8
-1.9
-2
-2.1
0
1
2
3
4
5
6
7
8
9
10
Ready
Sample based
Offset=0
T=10.000



Adjusting Horizons in Real-Time on Embedded Systems

The major benefit of using run-time prediction and control horizon inports in MPC and Adaptive MPC blocks is that you can evaluate and adjust controller performance in real-time without regenerating code and re-deploying it to the target system. This feature is very helpful at the prototyping stage.

To use run-time horizon adjustment in real time, the target system must support dynamic memory allocation because as horizons change, the sizes of all that matrices used to construct the optimization problem change at run-time as well.

You also need to specify the maximum prediction horizon in the block dialog to define the upper bound of the sizes of these matrices. Therefore, the memory footprint would be large. After finding the best horizon choices, it is recommended to disable the feature to have efficient code generation with fixed-size data.

```
bdclose(mdl)
```

See Also

MPC Controller

More About

- “Adjust Horizons at Run Time” on page 5-45

Switch Controller Online and Offline with Bumpless Transfer

This example shows how to obtain bumpless transfer when switching a model predictive controller from manual to automatic operation or vice versa.

During startup of a manufacturing process, before switching to automatic control, operators adjust key actuators manually until the plant is near the desired operating point. If not done correctly, the transfer can cause a bump; that is, a 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.

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.

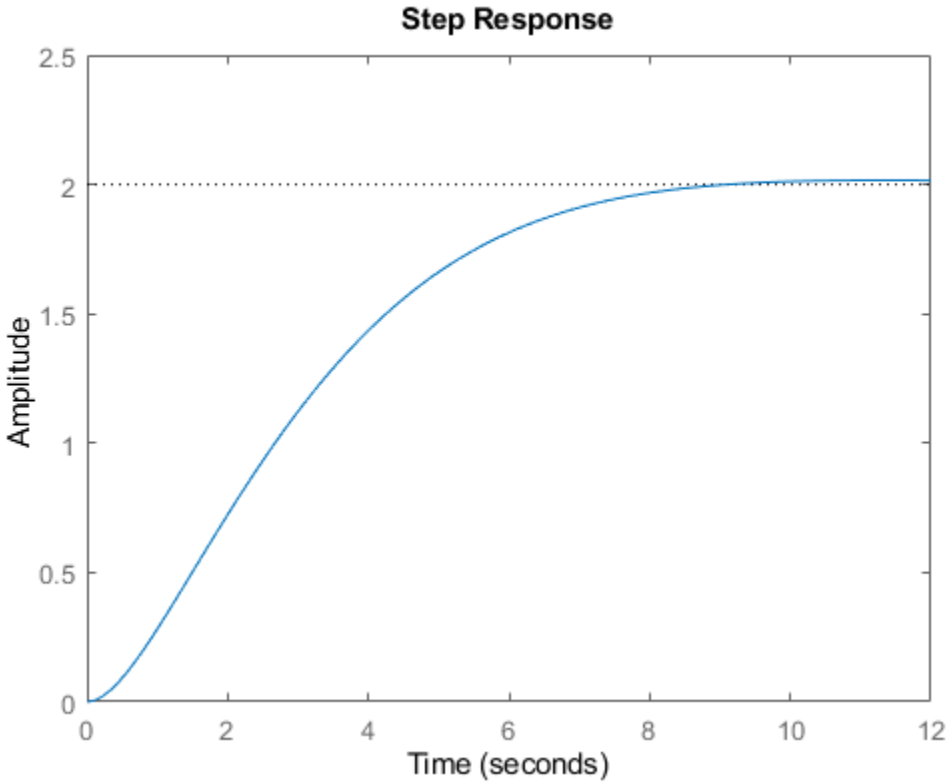
Define Plant Model

Define linear open-loop dynamic 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)
```



Design MPC Controller

Create an MPC controller, specifying the:

- Plant model
- Sample time `Ts`
- Prediction horizon `p`
- Control horizon `m`

```
Ts = 0.5;
p = 15;
m = 2;
mpcobj = mpc(sys,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Define constraints on the manipulated variable.

```
mpcobj.MV=struct('Min',-1,'Max',1);
```

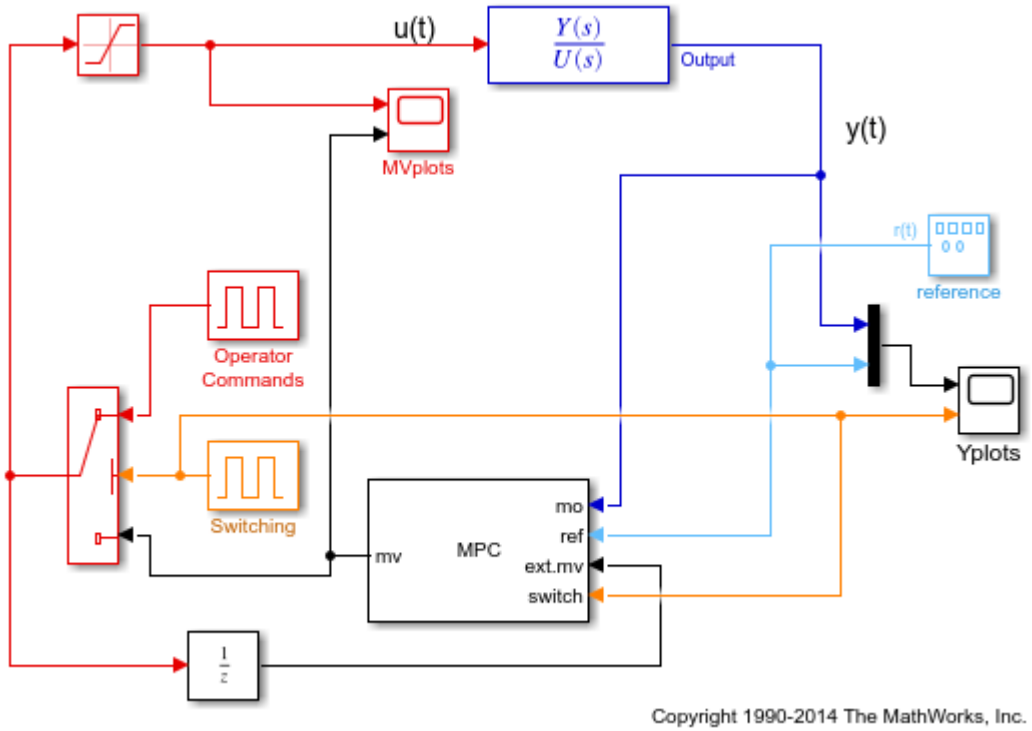
Specify the output tuning weight.

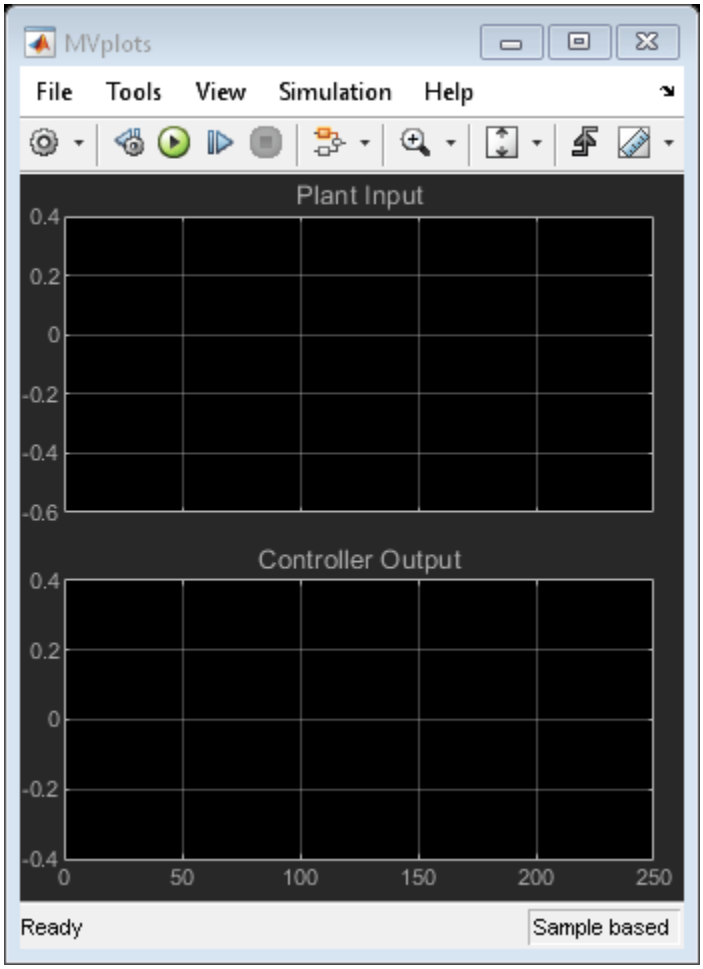
```
mpcobj.Weights.Output=0.01;
```

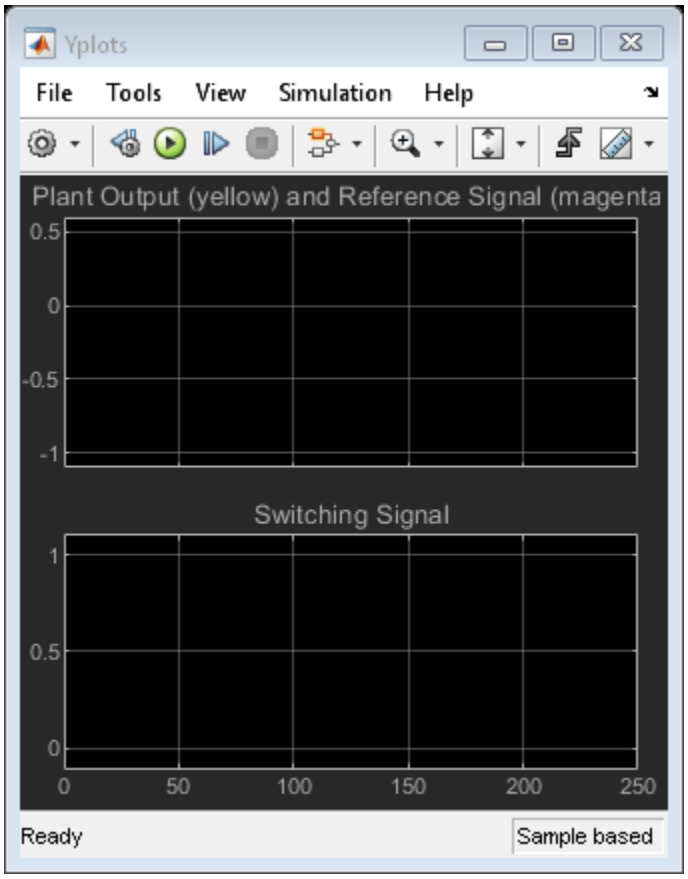
Open and Configure Simulink Model

Open the Simulink model.

```
mdl = 'mpc_bumpless';
open_system(mdl)
```




MVplots
File
Tools
View
Simulation
Help
Plant Input
0.4
0.2
0
-0.2
-0.4
-0.6
Controller Output
0.4
0.2
0
-0.2
-0.4
0
50
100
150
200
250
Ready
Sample based



In this model, the MPC Controller block is already configured for bumpless transfer using the following controller parameter settings.

- The **External manipulated variable** parameter is selected. This parameter enables the use of external manipulated variables by adding the `ext.mv` inport to the block.
- The **Use external signal to enable or disable optimization** is selected. This parameter adds a `switch` inport for switching off the controller optimization calculations.

To achieve bumpless transfer, the initial states of your plant and controller must be the same, which is the case for the plant and controller in this example. However, if the initial conditions for your system do not match, you can set the initial states of the controller to the plant initial states. To do so, extract the `mpcstate` object from your controller and set the initial state of the plant.

```
stateobj = mpcstate(MPC1);
stateobj.Plant = x0;
```

where `x0` is a vector of the initial plant states. Then, set the **Initial Controller State** parameter of the MPC Controller block to `stateobj`.

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` inport, 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.

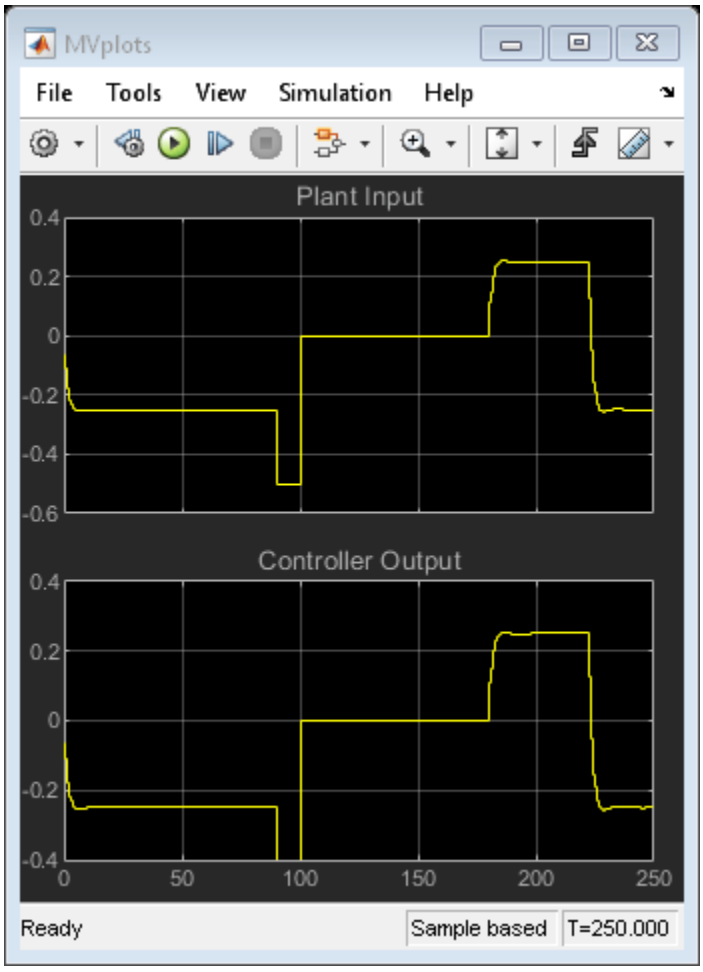
This model also shows the optimization switching option. When the system switches to manual operation, a nonzero signal enters the `switch` inport of the controller block. The signal turns off the optimization calculations of the controller, which reduces computational effort.

Simulate Controller in Simulink

Simulate closed-loop control of the linear plant model in Simulink.

```
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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



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 inport 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.

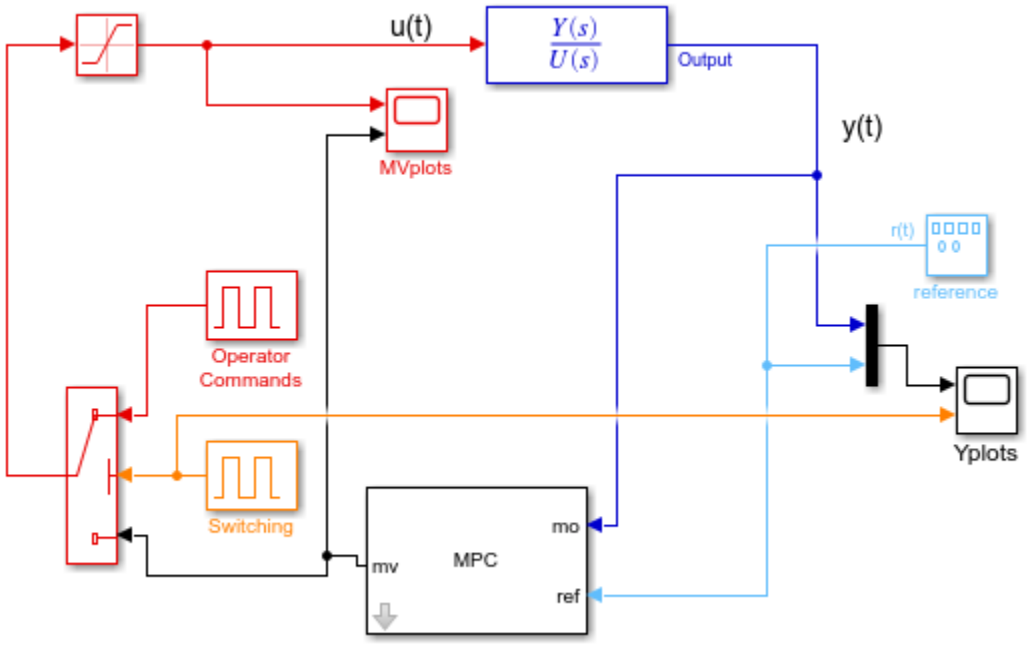
Turn Off Manipulated Variable Feedback

To examine the controller behavior without manipulated variable feedback, modify the model as follows:

- Delete the signals entering the `ext.mv` and switch `inports` of the MPC Controller block.
- Delete the Unit Delay block and the signal line entering its inport.
- For the MPC Controller block, clear the **External manipulated variable** and **Use external signal to enable or disable optimization** parameters.

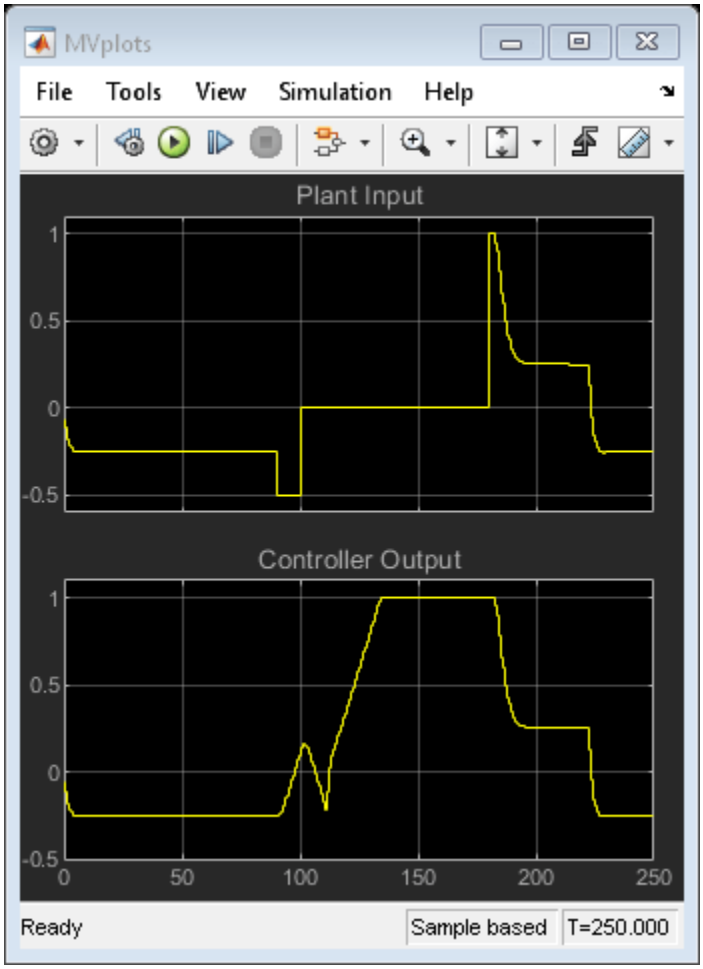
To perform these steps programmatically, use the following commands.

```
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');
```



Adjust the limits of the response plots, and simulate the model.

```
set_param([mdl '/Yplots'],'Ymin','-1.1~-0.1')
set_param([mdl '/Yplots'],'Ymax','2~1.1')
set_param([mdl '/MVplots'],'Ymin','-0.6~-0.5')
set_param([mdl '/MVplots'],'Ymax','1.1~1.1')
sim(mdl)
```


MVplots
File
Tools
View
Simulation
Help
Plant Input
1
0.5
0
-0.5
Controller Output
1
0.5
0
-0.5
0
50
100
150
200
250
Ready
Sample based
T=250.000

The behavior of the system 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 unwanted movements.

```
bdclose(mdl)
```

See Also

MPC Controller

Switching Controllers Based on Optimal Costs

This example shows how to use the "optimal cost" outport of the MPC Controller block to switch between multiple model predictive controllers whose outputs are restricted to discrete values.

Define Plant Model

The linear plant model is as follows::

```
plant = ss(tf({1,1},{[1 1.2 1],[1 1]}),'min');  % Plant with 2 inputs and 1 output
[A,B,C,D] = ssdata(plant);                      % Get state-space realization matrices, to be use
x0 = [0;0;0];                                   % Initial plant state
```

Design MPC Controller

Specify input and output signal types.

```
plant = setmpcsignals(plant,'MV',1,'MD',2);     % First input is manipulated, second is measured
```

Design two MPC controllers with the MV constraints of u=-1 and u=1, respectively. Only u at the current time is quantized. The subsequent calculated control actions may be any value between -1 and 1. The controller uses a receding horizon approach so these values don't actually go to the plants.

```
Ts = 0.2;                                           % Sampling time
p = 20;                                             % Prediction horizon
m = 10;                                             % Control horizon
mpc1 = mpc(plant,Ts,p,m);                             % First MPC object
mpc2 = mpc(plant,Ts,p,m);                             % Second MPC object
% Specify weights
mpc1.Weights = struct('MV',0,'MVRate',.3,'Output',1);   % Weights
mpc2.Weights = struct('MV',0,'MVRate',.3,'Output',1);   % Weights
% Specify constraints
mpc1.MV = struct('Min',[-1;-1],'Max',[-1;1]);       % Constraints on the manipulated variable: u
mpc2.MV = struct('Min',[1;-1],'Max',[1;1]);         % Constraints on the manipulated variable: u
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

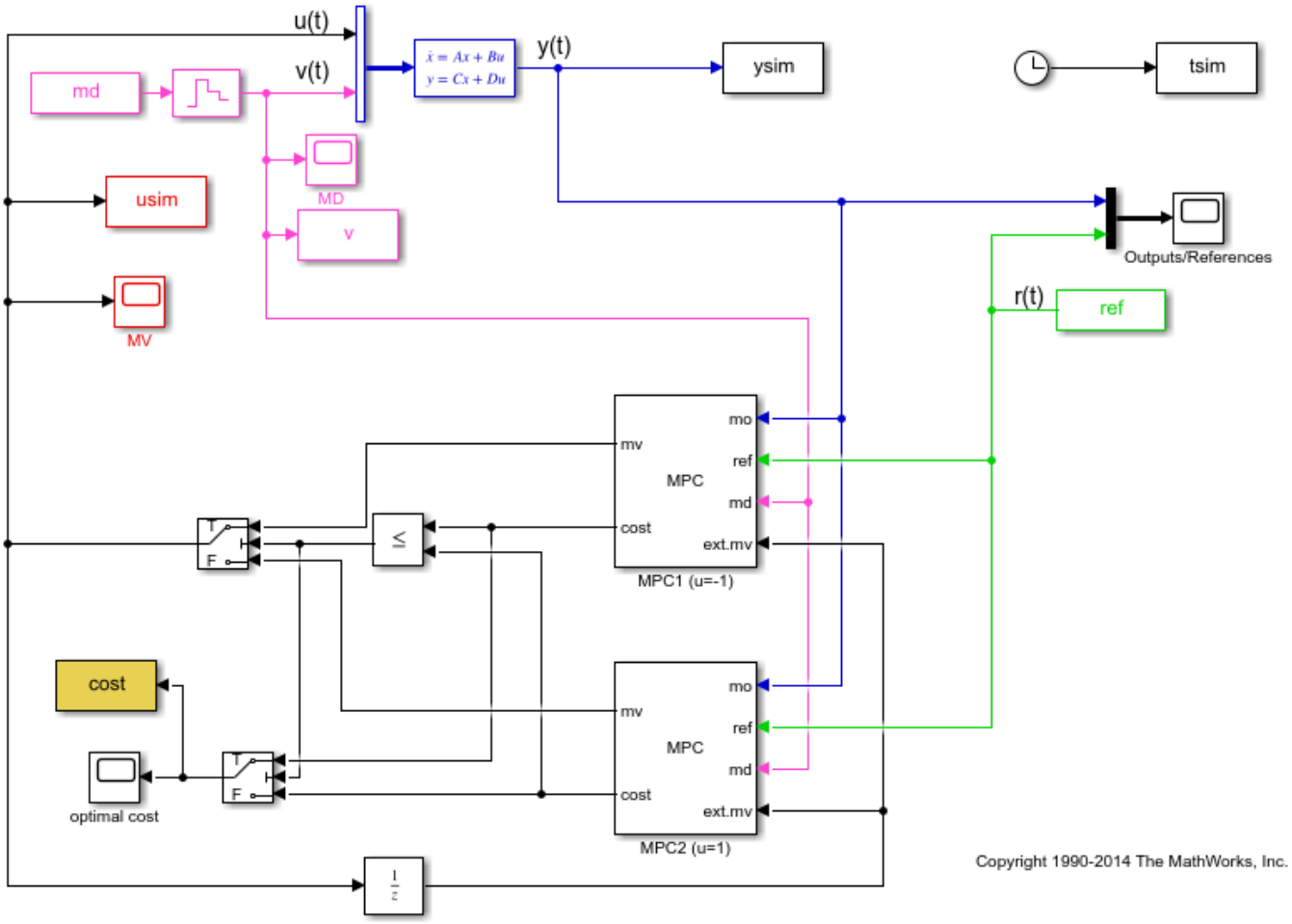
Simulate in Simulink®

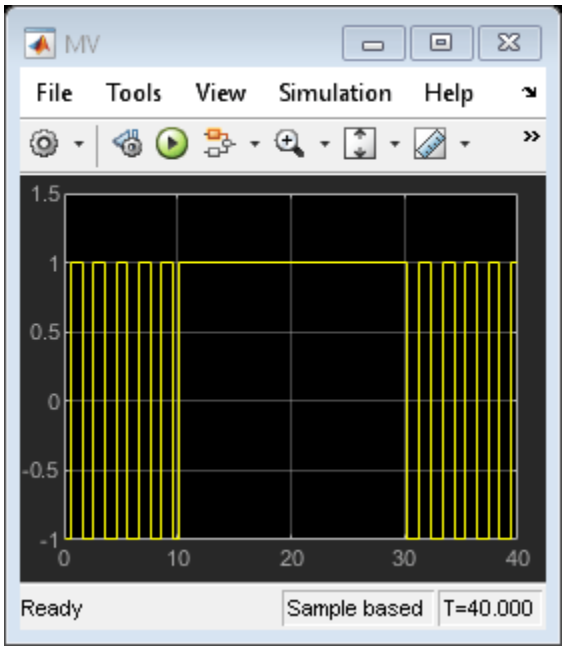
To run this example, Simulink® is required.

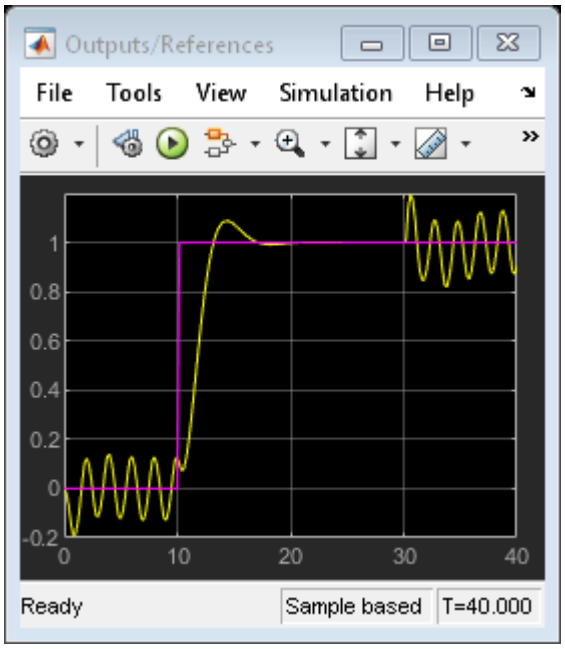
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
% Specify signals:
Tstop = 40;
ref.time = 0:Ts:(Tstop+p*Ts);
ref.signals.values = double(ref.time>10)';          % Step change in reference signal at time t=1
md.time = ref.time;
md.signals.values = double(md.time>30)';            % Step change in measured disturbance signal
```

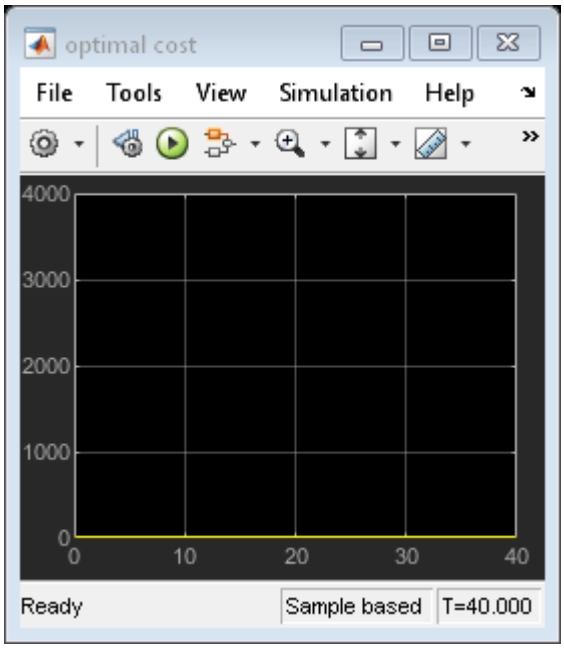
Open and simulate the Simulink® model:

```
mdl = 'mpc_optimalcost';
open_system(mdl);                    % Open Simulink(R) Model
sim(mdl,Tstop);                      % Start Simulation

-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
```




MV
File Tools View Simulation Help
1.5
1
0.5
0
-0.5
-1
0 10 20 30 40
Ready
Sample based
T=40.000


Outputs/References
File Tools View Simulation Help
1
0.8
0.6
0.4
0.2
0
-0.2
0 10 20 30 40
Ready
Sample based
T=40.000



Note that:

- From time 0 to time 10, the control action keeps switching between MPC1 (-1) and MPC2 (+1). This is because the reference signal is 0 and it requires a controller output at 0 to reach steady state, which cannot be achieved with either MPC controller.
- From time 10 to 30, MPC2 control output (+1) is chosen because the reference signal becomes +1 and it requires a controller output at +1 to reach steady state (plant gain is 1), which can be achieved by MPC2.
- From time 30 to 40, control action starts switching again. This is because with the presence of measured disturbance (+1), MPC1 leads to a steady state of 0 and MPC2 leads to a steady state of +2, while the reference signal still requires +1.

Simulate Using MPCMOVE Command

Use `mpcmove` to perform step-by-step simulation and compute current MPC control action:

```
[Ad,Bd,Cd,Dd] = ssdata(c2d(plant,Ts));              % Discrete-time dynamics
Nsteps = round(Tstop/Ts);                           % Number of simulation steps
```

Initialize matrices for storing simulation results

```
YY = zeros(Nsteps+1,1);
RR = zeros(Nsteps+1,1);
UU = zeros(Nsteps+1,1);
COST = zeros(Nsteps+1,1);
x = x0;                                             % Initial plant state
xt1 = mpcstate(mpc1);                               % Initial state of the MPC controller #1
xt2 = mpcstate(mpc2);                               % Initial state of the MPC controller #2
```

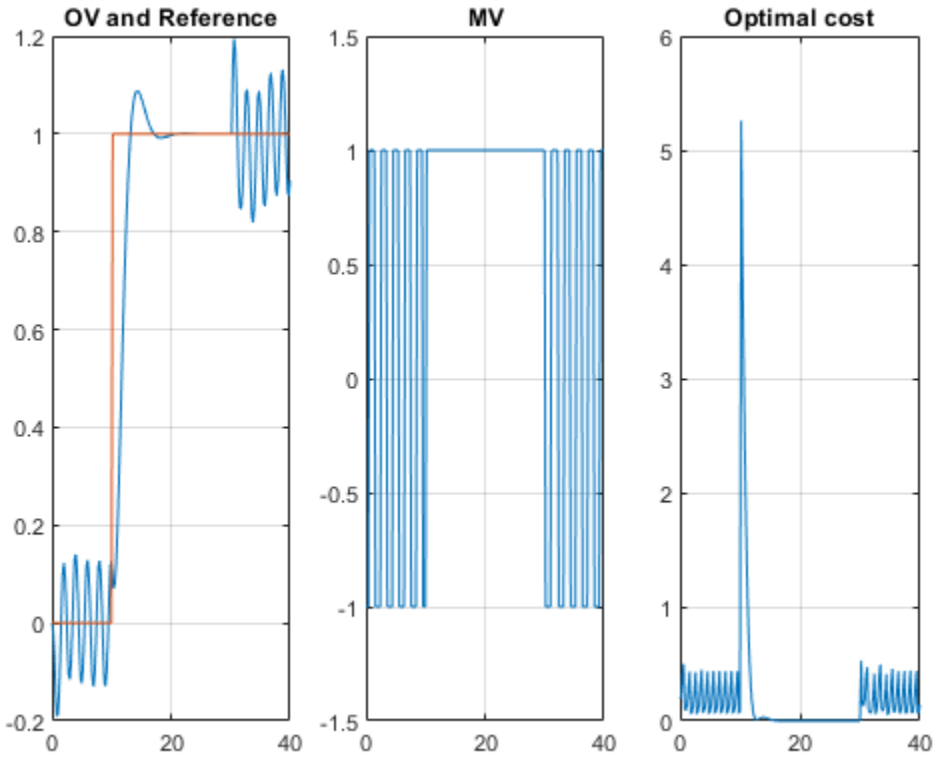
Start simulation.

```
for td=0:Nsteps
    % Construct signals
```

```
    v = md.signals.values(td+1);
    r = ref.signals.values(td+1);
    % Plant equations: output update
    y = Cd*x + Dd(:,2)*v;
    % Compute both MPC moves
    options = mpcmoveopt;
    options.OnlyComputeCost = true; % the last input argument tells "mpcmove" to only compute the
    [u1,Info1] = mpcmove(mpc1,xt1,y,r,v,options);
    [u2,Info2] = mpcmove(mpc2,xt2,y,r,v,options);
    % Compare the resulting optimal costs and choose the input value
    % corresponding to the smallest cost
    if Info1.Cost<=Info2.Cost
        u = u1;
        cost = Info1.Cost;
        % Update internal MPC state to the correct value
        xt2.Plant = xt1.Plant;
        xt2.Disturbance = xt1.Disturbance;
        xt2.LastMove = xt1.LastMove;
    else
        u = u2;
        cost = Info2.Cost;
        % Update internal MPC state to the correct value
        xt1.Plant = xt2.Plant;
        xt1.Disturbance = xt2.Disturbance;
        xt1.LastMove = xt2.LastMove;
    end
    % Store plant information
    YY(td+1) = y;
    RR(td+1) = r;
    UU(td+1) = u;
    COST(td+1) = cost;
    % Plant equations: state update
    x = Ad*x + Bd(:,1)*u + Bd(:,2)*v;
end
```

Plot the results of `mpcmove` to compare with the simulation results obtained in Simulink®:

```
subplot(131)
plot((0:Nsteps)*Ts,[YY,RR]);                    % Plot output and reference signals
grid
title('OV and Reference')
subplot(132)
plot((0:Nsteps)*Ts,UU);                         % Plot manipulated variable
grid
title('MV')
subplot(133)
plot((0:Nsteps)*Ts,COST);                       % Plot optimal MPC value function
grid
title('Optimal cost')
```



These plots resemble the plots in the scopes in the Simulink® model.

```
bdclose(mdl);
```

See Also

MPC Controller

More About

- “Optimization Problem” on page 2-7

Monitoring Optimization Status to Detect Controller Failures

This example shows how to use the `qp.status` outport of the MPC Controller block in Simulink® to detect controller failures in real time.

Overview of Run-Time Control Monitoring

The `qp.status` output from the MPC Controller block returns a positive integer when the controller finds an optimal control action by solving a quadratic programming (QP) problem. The integer value corresponds to the number of iterations used during optimization. If the QP problem formulated at a given sample interval is infeasible, the controller will fail to find a solution. In that case, the MV outport of controller block retains the most recent value and the `qp.status` outport returns -1. In a rare case when the maximum number of iteration is reached during optimization, the `qp.status` outport returns 0.

In industrial MPC applications, you can detect whether your model predictive controller is in a failure mode (0 or -1) or not by monitoring the `qp.status` outport. If an MPC failure occurs, you can use this signal to switch to a backup control plan.

For more information, see Lane Keeping Assist System and the `Optimizer` options in the `mpc`.

This example shows how to setup run-time controller status monitoring in Simulink.

Define Plant Model

The test plant is a single-input, single-output plant with hard limits on both manipulated variable and controlled output. A load disturbance is added at the plant output. The disturbance consists of a ramp signal that saturates manipulated variable due to the hard limit on the MV. After saturation occurs, you lose the control degree of freedom and the disturbance eventually forces the output outside its upper limit. When that happens, the QP problem formulated by the model predictive controller at run-time becomes infeasible.

Define the plant model as a simple SISO system with unity static 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 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 for `plant` with a sample time of `0.2` seconds.

```
mpcobj = mpc(plant, 0.2);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Define hard constraints on plant input (MV) and output (OV). By default, all the MV constraints are hard and OV constraints are soft.

```
mpcobj.MV.Min = -1;
mpcobj.MV.Max = 1;
mpcobj.OV.Min = -1;
mpcobj.OV.Max = 1;
```

Configure the upper and lower OV constraints as hard bounds.

```
mpcobj.OV.MinECR = 0;
mpcobj.OV.MaxECR = 0;
```

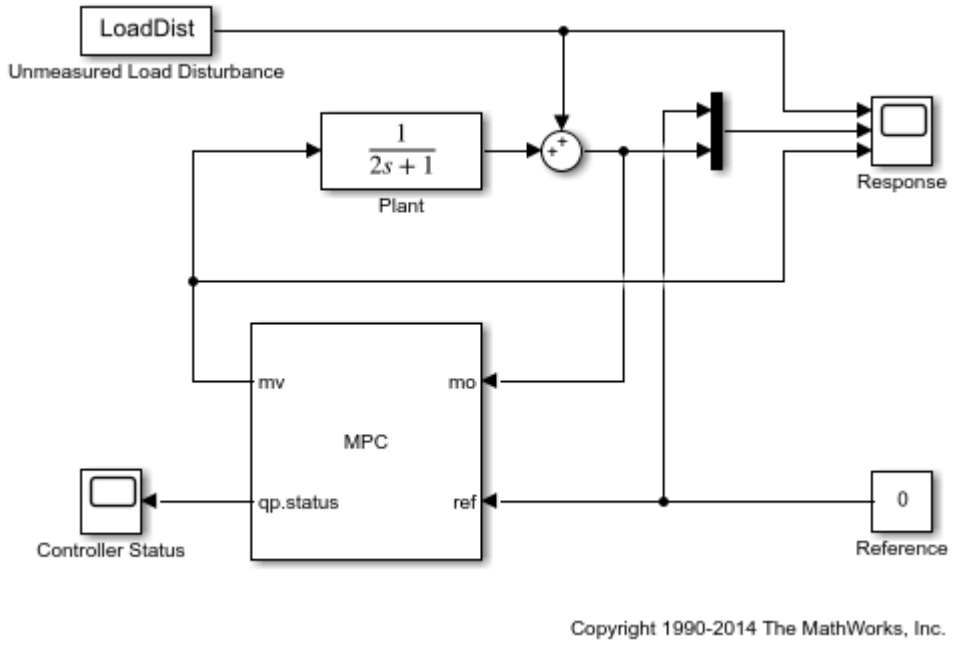
Override the default estimator. This high-gain estimator improves detection of an impending constraint violation.

```
setEstimator(mpcobj,[],[0;1])
```

Simulate Using Simulink

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)
```

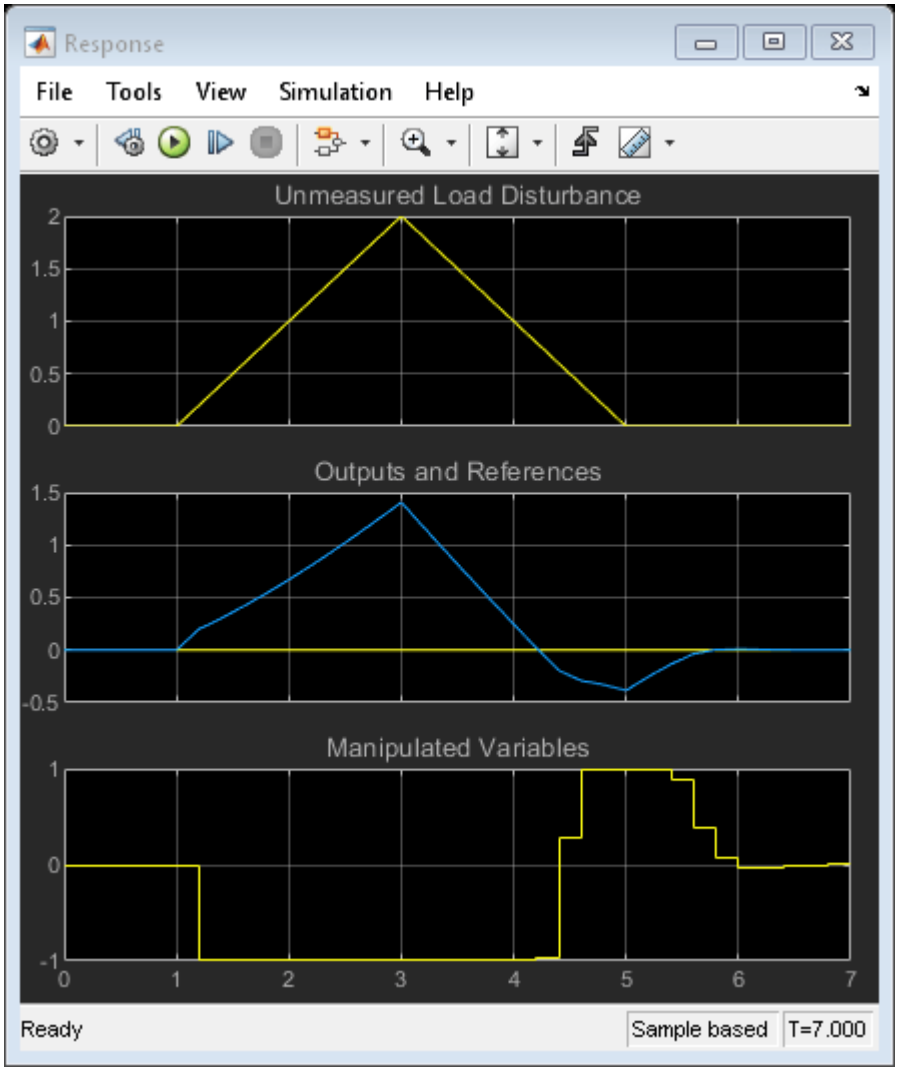


Simulate the closed-loop and display the response.

```
sim(mdl)
open_system([mdl '/Controller Status'])
open_system([mdl '/Response'])
```

```
-->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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Controller Status
File
Tools
View
Simulation
Help
QP Status
3
2
1
0
-1
0
1
2
3
4
5
6
7
Ready
Sample based
T=7.000



As shown in the response scope, the ramp-up disturbance signal causes the MV to saturate at its lower bound -1, which is the optimal solution for these situations. After the plant output exceeds the upper limit, at the next sampling interval (2.6 seconds), the controller realizes that it can no longer keep the output within bounds (because its MV is still saturated), so it signals controller failure due to an infeasible QP problem (-1 in the controller status scope). After the output comes back within bounds, the QP problem becomes feasible again (3.4 seconds). Once the MV is no longer saturated, normal control behavior returns.

Close the Simulink model.

```
bdclose(mdl)
```

See Also

Simulate MPC Controller with a Custom QP Solver

You can simulate the closed-loop response of an MPC controller with a custom quadratic programming (QP) solver in Simulink®.

This example uses an on-line monitoring application, first solving it using the Model Predictive Control Toolbox™ built-in solver, then using a custom solver that uses the `quadprog` (Optimization Toolbox) solver from the Optimization Toolbox™.

Implementing a custom QP solver in this way does not support code generation. For more information on generating code for a custom QP solver, see “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 10-56. For more information on QP Solvers, see “QP Solvers” on page 2-17.

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 = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Define hard constraints on the 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.

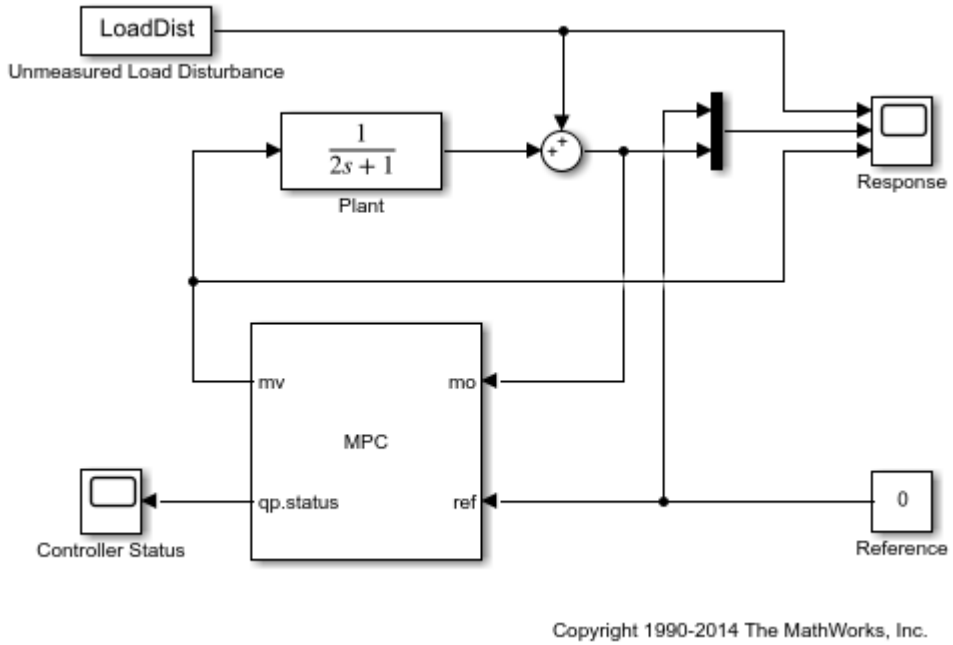
Simulate Using Simulink with Built-in QP Solver

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 by selecting the **Optimization status** parameter of the MPC Controller block. Display the run-time `qp.status` value in the Controller Status scope.

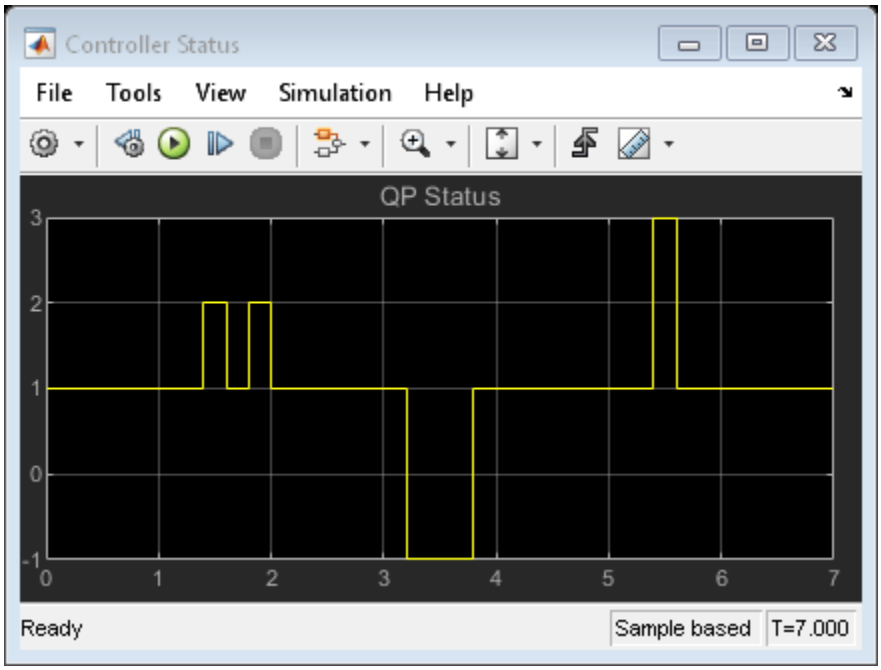
```
mdl = 'mpc_onlinemonitoring';
open_system(mdl)
```

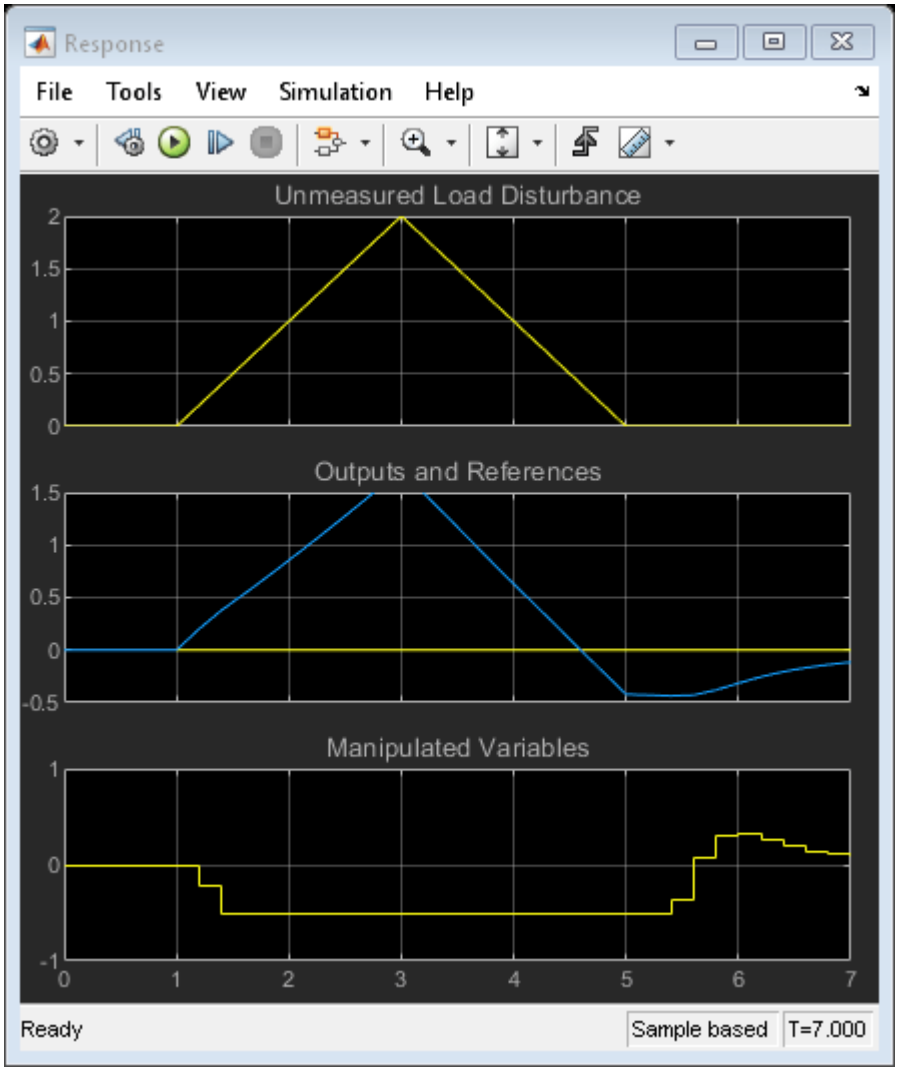


Simulate the closed-loop response using the default Model Predictive Control Toolbox QP solver.

```
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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```


Controller Status
File
Tools
View
Simulation
Help
QP Status
3
2
1
0
-1
0
1
2
3
4
5
6
7
Ready
Sample based
T=7.000



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 the `Optimizer.CustomSolver` property of 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 match those defined in the `mpcCustomSolver.txt` template file.
- 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. To implement your own custom QP solver, modify this file.

Save the function in your working folder as a `.m` file.

```
src = which('mpcCustomSolver.txt');
dest = fullfile(pwd,'mpcCustomSolver.m');
copyfile(src,dest,'f');
```

Review the saved `mpcCustomSolver.m` file.

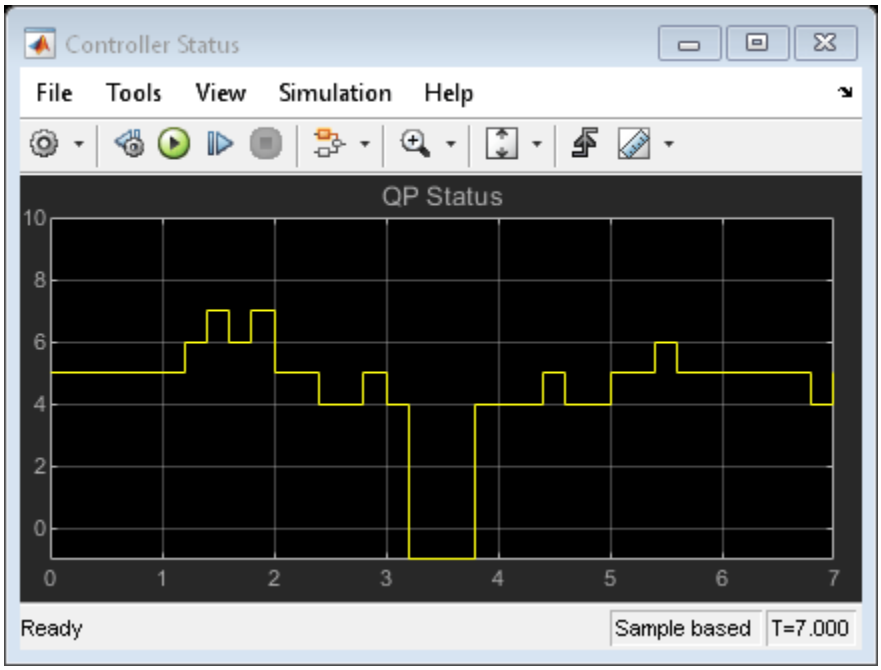
```
function [x, status] = mpcCustomSolver(H, f, A, b, x0)
% mpcCustomSolver allows user to specify a custom quadratic programming
% (QP) solver to solve the QP problem formulated by MPC controller.  When
% the "mpcobj.Optimizer.CustomSolver" property is set true, instead of
% using the built-in QP solver, 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)
```

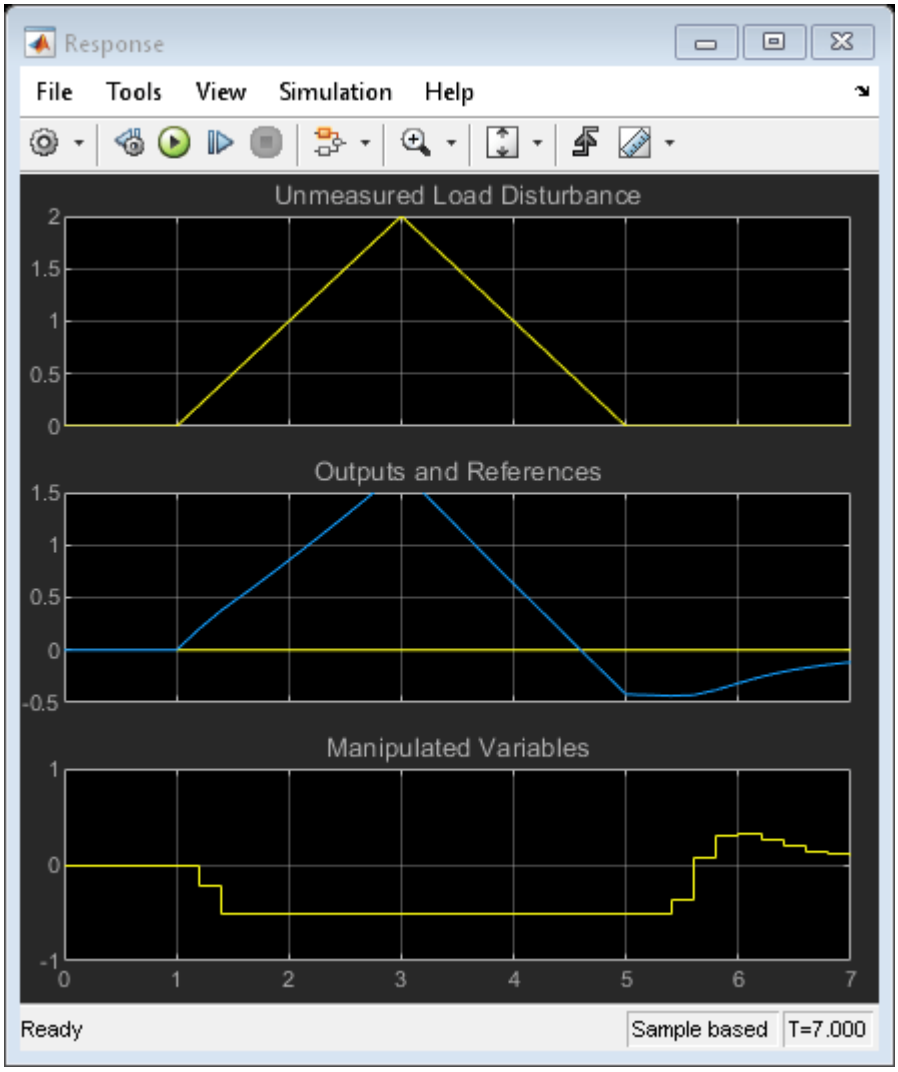
```
%
% DO NOT CHANGE LINES ABOVE

% 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)
```

```
-->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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```


Controller Status
File
Tools
View
Simulation
Help
QP Status
10
8
6
4
2
0
0
1
2
3
4
5
6
7
Ready
Sample based
T=7.000



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);
```

See Also

More About

- “QP Solvers” on page 2-17
- “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 10-56

Use Suboptimal Solution in Fast MPC Applications

This example shows how to guarantee the worst-case execution time of an MPC controller in real-time applications by using the suboptimal solution returned by the optimization solver.

What is a Suboptimal Solution?

Model predictive control (MPC) solves a quadratic programming (QP) problem at each control interval. The built-in QP solver uses an iterative active-set algorithm that is efficient for MPC applications. However, when constraints are present, there is no way to predict how many solver iterations are required to find an optimal solution. Also, in real-time applications, the number of iterations can change dramatically from one control interval to the next. In such cases, the worst-case execution time can exceed the limit that is allowed on the hardware platform and determined by the controller sample time.

You can guarantee the worst-case execution time for your MPC controller by applying a suboptimal solution after the number of optimization iterations exceeds a specified maximum value. To set the worst-case execution time, first determine the time needed for a single optimization iteration by experimenting with your controller under nominal conditions. Then, set a small upper bound on the number of iterations per control interval.

By default, when the maximum number of iterations is reached, an MPC controller does not use the suboptimal solution. Instead, the controller sets an error flag (`status = 0`) and freezes its output. Often, the solution available in earlier iterations is good enough, but requires refinement to find an optimal solution, which leads to many additional iterations.

This example shows how to configure your MPC controller to use the suboptimal solution. The suboptimal solution is a feasible solution available at the final iteration (modified, if necessary, to satisfy any hard constraints on the manipulated variables). To determine whether the suboptimal solution provides acceptable control performance for your application, run simulations across your operating range.

Define Plant Model

The plant model is a stable randomly generated state-space system. It has `10` states, `3` manipulated variables (MV), and `3` outputs (OV).

```
rng(1234);
nX = 10;
nOV = 3;
nMV = 3;
Plant = rss(nX,nOV,nMV);
Plant.d = 0;
Ts = 0.1;
```

Design MPC Controller with Constraints on MVs and OVs

Create an MPC controller with default values for all controller parameters except the constraints. Specify constraints on both the manipulated and output variables.

```
verbosity = mpcverbosity('off'); % Temporarily disable command line messages.
mpcobj = mpc(Plant, Ts);
for i = 1:nMV
    mpcobj.MV(i).Min = -1.0;
    mpcobj.MV(i).Max =  1.0;
```

```
end
for i = 1:nOV
    mpcobj.OV(i).Min = -1.0;
    mpcobj.OV(i).Max =  1.0;
end
```

Simultaneous constraints on both manipulated and output variables require a relatively large number of QP iterations to determine the optimal control sequence.

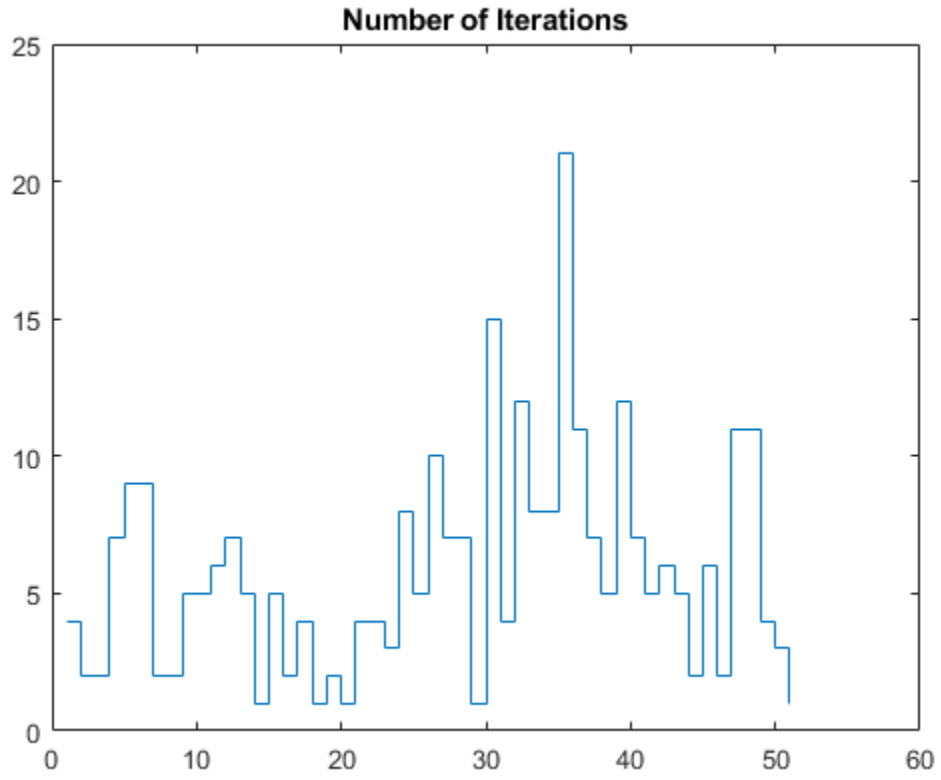
Simulate in MATLAB with Random Output Disturbances

First, simulate the MPC controller using the optimal solution in each control interval. To focus on only output disturbance rejection performance, set the output reference values to zero.

```
T = 5;
N = T/Ts + 1;
r = zeros(1,nOV);
SimOptions = mpcsimopt();
SimOptions.OutputNoise = 3*randn(N,nOV);
[y,t,u,~,~,~,status] = sim(mpcobj,N,r,[],SimOptions);
```

Plot the number of iterations used in each control interval.

```
figure
stairs(status)
hold on
title('Number of Iterations')
```



The largest number of iterations is 21, and the average is 5.8 iterations.

Create an MPC controller with the same settings, but configure it to use the suboptimal solution.

```
mpcobjSub = mpcobj;
mpcobjSub.Optimizer.UseSuboptimalSolution = true;
```

Reduce the maximum number of iterations for the default active-set QP solver to a small number.

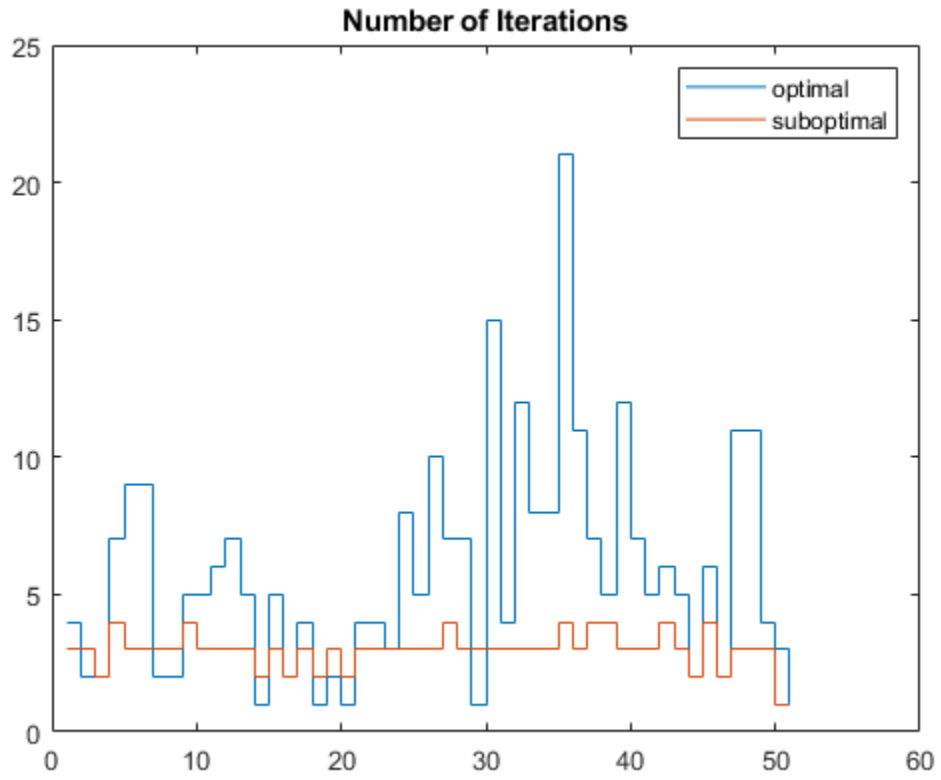
```
mpcobjSub.Optimizer.ActiveSetOptions.MaxIterations = 3;
```

Simulate the second controller with the same output disturbance sequence.

```
[ySub,tSub,uSub,~,~,~,statusSub] = sim(mpcobjSub,N,r,[],SimOptions);
```

Plot the number of iterations used in each control interval on the same plot. For any control interval in which the maximum number of iterations is reached, `statusSub` is zero. Before plotting the result, set the number of iterations for these intervals to `3`.

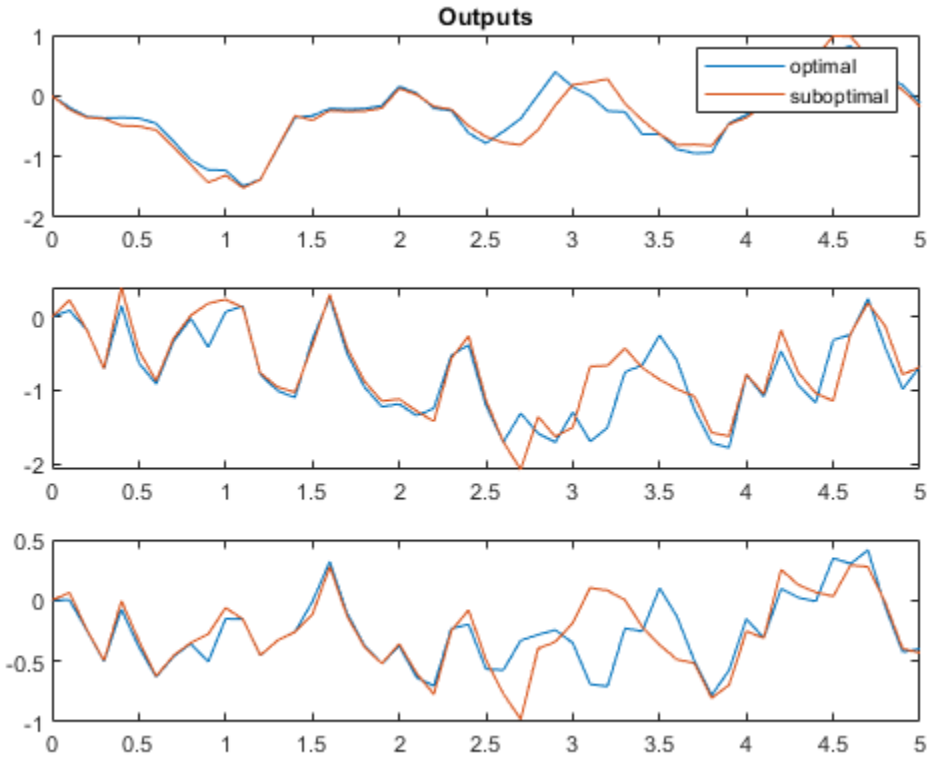
```
statusSub(statusSub == 0) = 3;
stairs(statusSub)
legend('optimal','suboptimal')
```



The largest number of iterations is now 3, and the average is 2.8 iterations.

Compare the performance of the two controllers. When the suboptimal solution is used, there is no significant deterioration in control performance compared to the optimal solution.

```
figure
for ct=1:3
    subplot(3,1,ct)
    plot(t,y(:,ct),t,ySub(:,ct))
end
subplot(3,1,1)
title('Outputs')
legend('optimal','suboptimal')
```

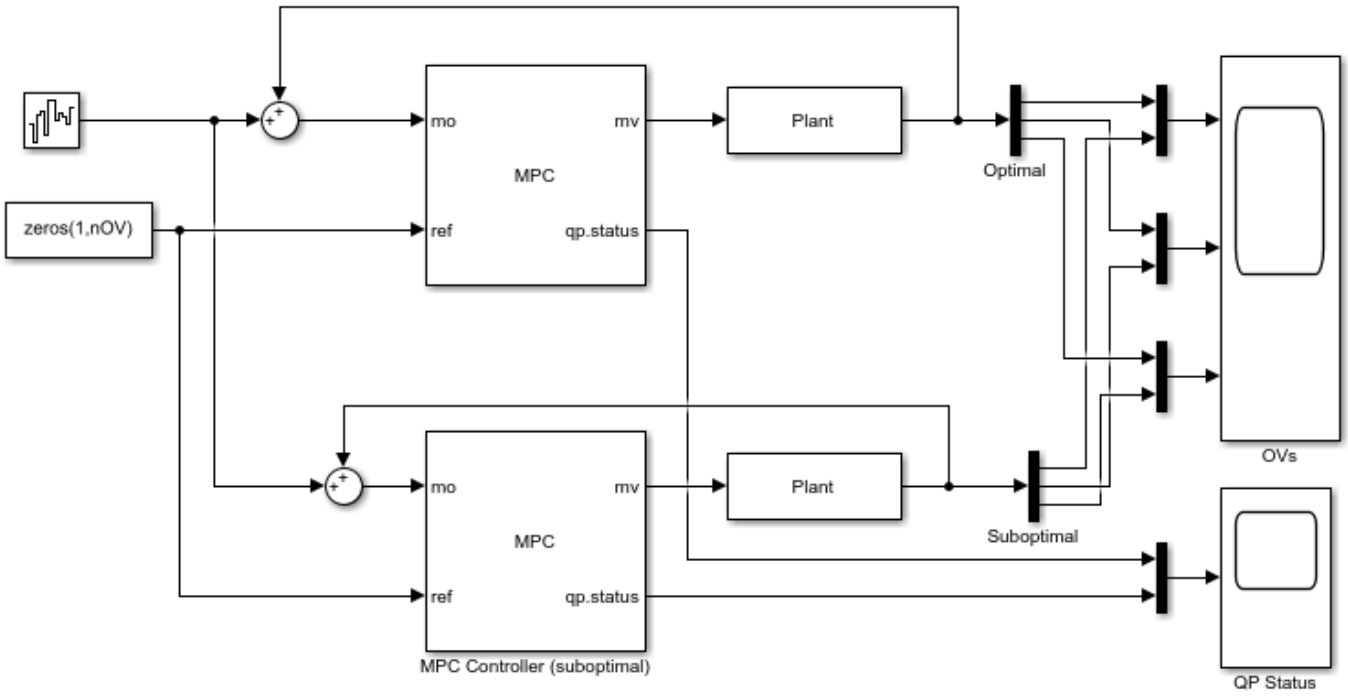


For a real-time application, as long as each solver iteration takes less than 30 milliseconds on the hardware, the worst-case execution time does not exceed the controller sample time (`0.1` seconds). In general, it is safe to assume that the execution time used by each iteration is more or less a constant.

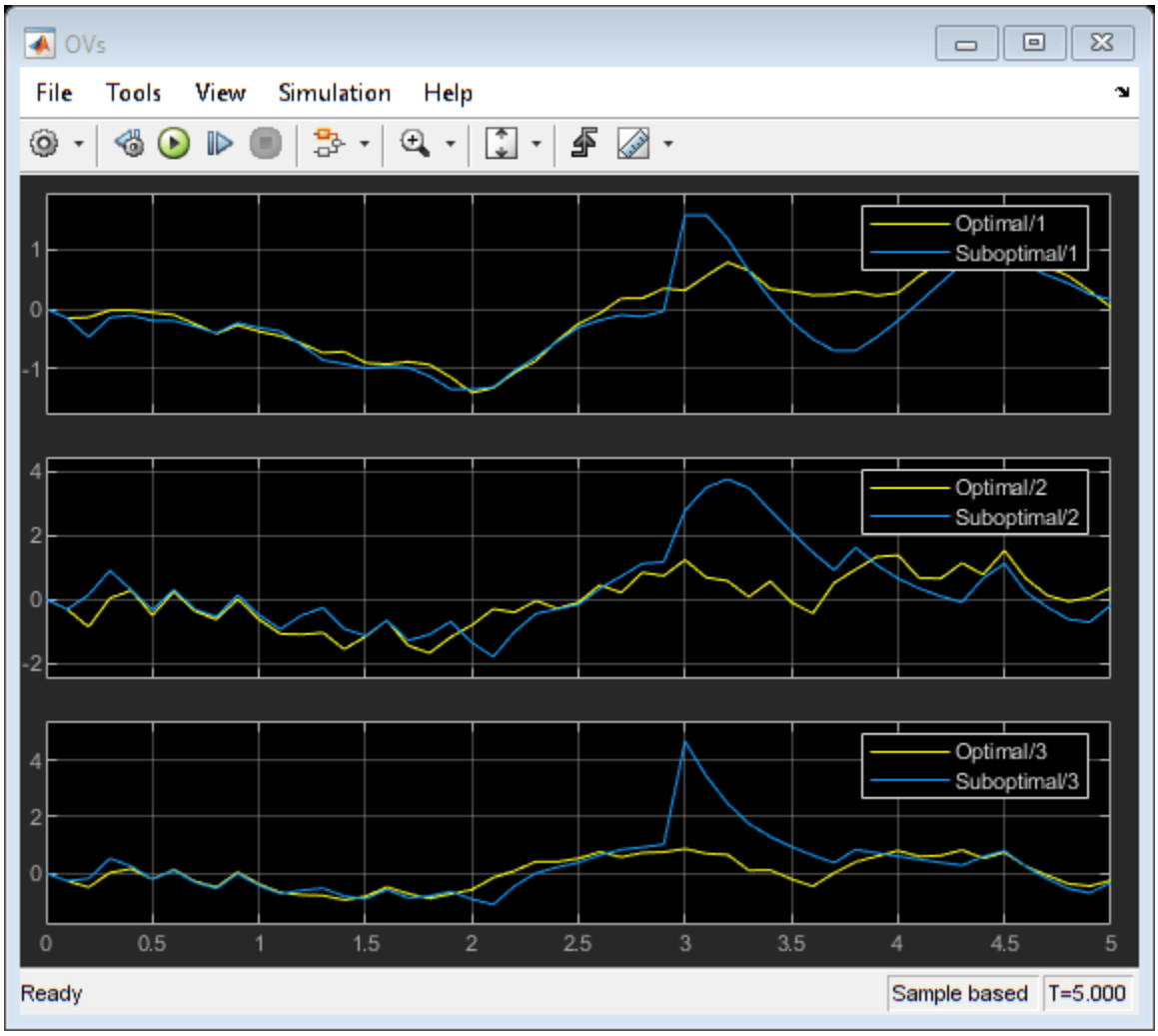
Simulate in Simulink with Random Output Disturbances

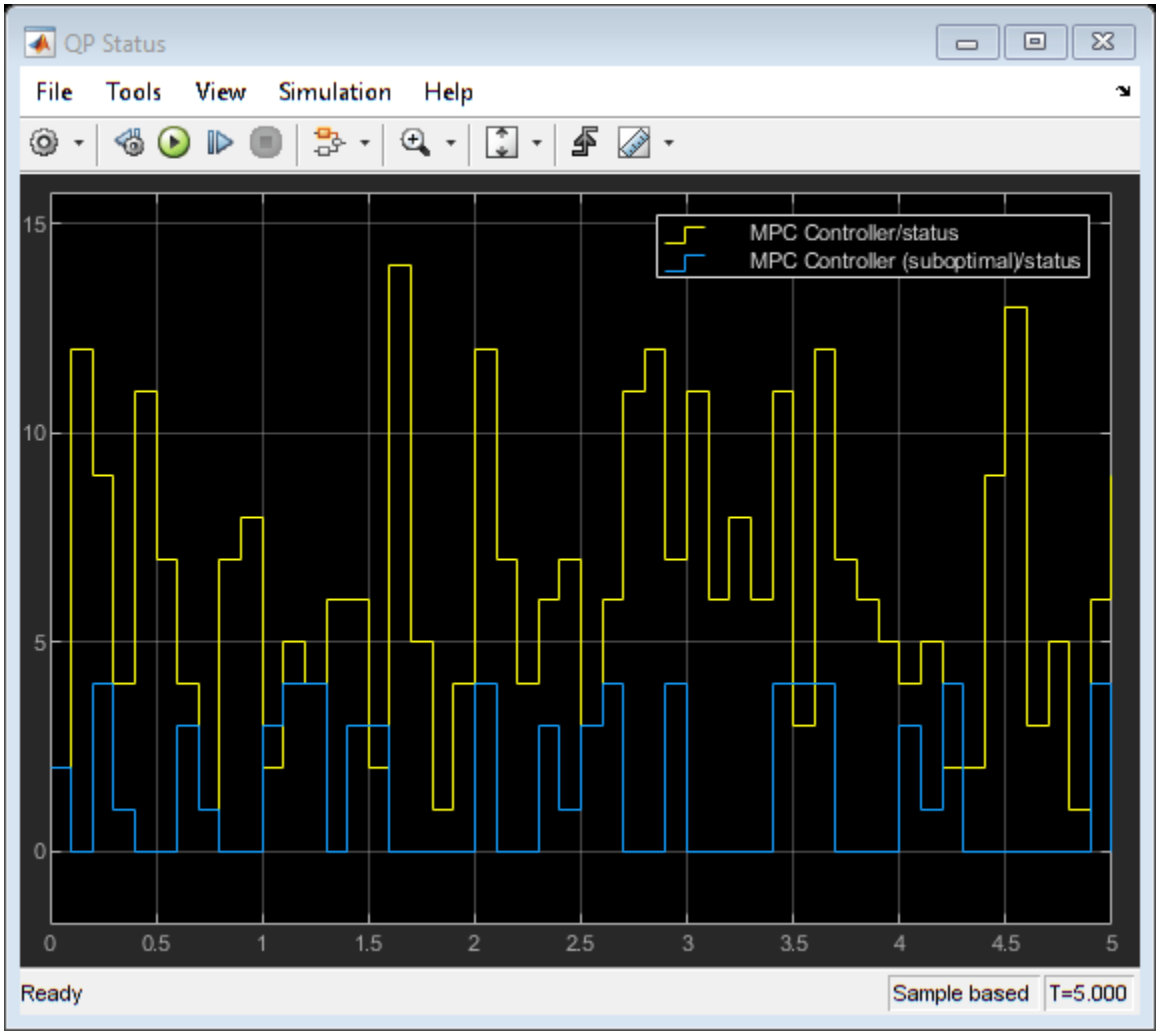
Simulate the controllers in Simulink®.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end

Model = 'mpc_SuboptimalSolution';
open_system(Model)
sim(Model)
```



Copyright 2017-2018 The MathWorks, Inc.


OVs
File
Tools
View
Simulation
Help
Optimal/1
Suboptimal/1
Optimal/2
Suboptimal/2
Optimal/3
Suboptimal/3
1
0
-1
4
2
0
-2
4
2
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
Sample based
T=5.000



As in the command-line simulation, the average number of QP iterations per control interval decreased without significantly affecting control performance.

```
mpcverbosity(verbosity); % Enable command line messages.
bdclose(Model)
```

See Also

Functions
`mpcmoveopt`

Blocks
MPC Controller | Adaptive MPC Controller

More About

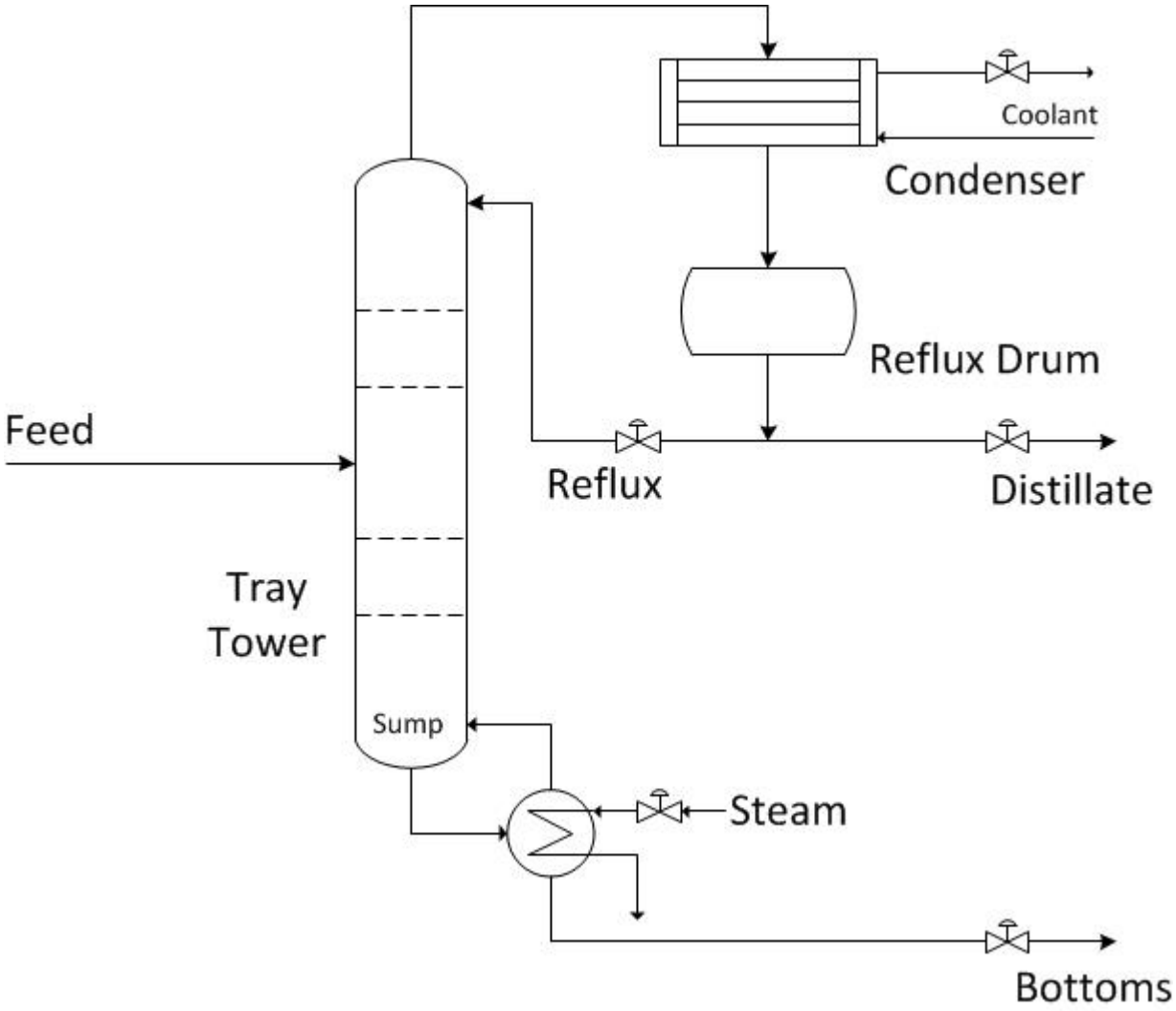
- “QP Solvers” on page 2-17

Design and Cosimulate Control of High-Fidelity Distillation Tower with Aspen Plus Dynamics

This example shows how to design a model predictive controller in MATLAB for a high-fidelity distillation tower model built in Aspen Plus Dynamics®. The controller performance is then verified through cosimulation between Simulink and Aspen Plus Dynamics.

Distillation Tower

The distillation tower uses 29 ideal stages to separate a mixture of benzene, toluene, and xylenes (represented by p-xylene). The distillation process is continuous. The equipment includes a reboiler and a total condenser as shown below:



The distillation tower operates at a nominal steady-state condition:

- The feed stream contains 30% of benzene, 40% of toluene and 30% of xylenes.
- The feed flow rate is 500 kmol/hour.

- To satisfy the distillate purity requirement, the distillate contains 95% of benzene.
- To satisfy the requirement of recovering 95% of benzene in the feed, the benzene impurity in the bottoms is 1.7%.

The control objectives are listed below, sorted by their importance:

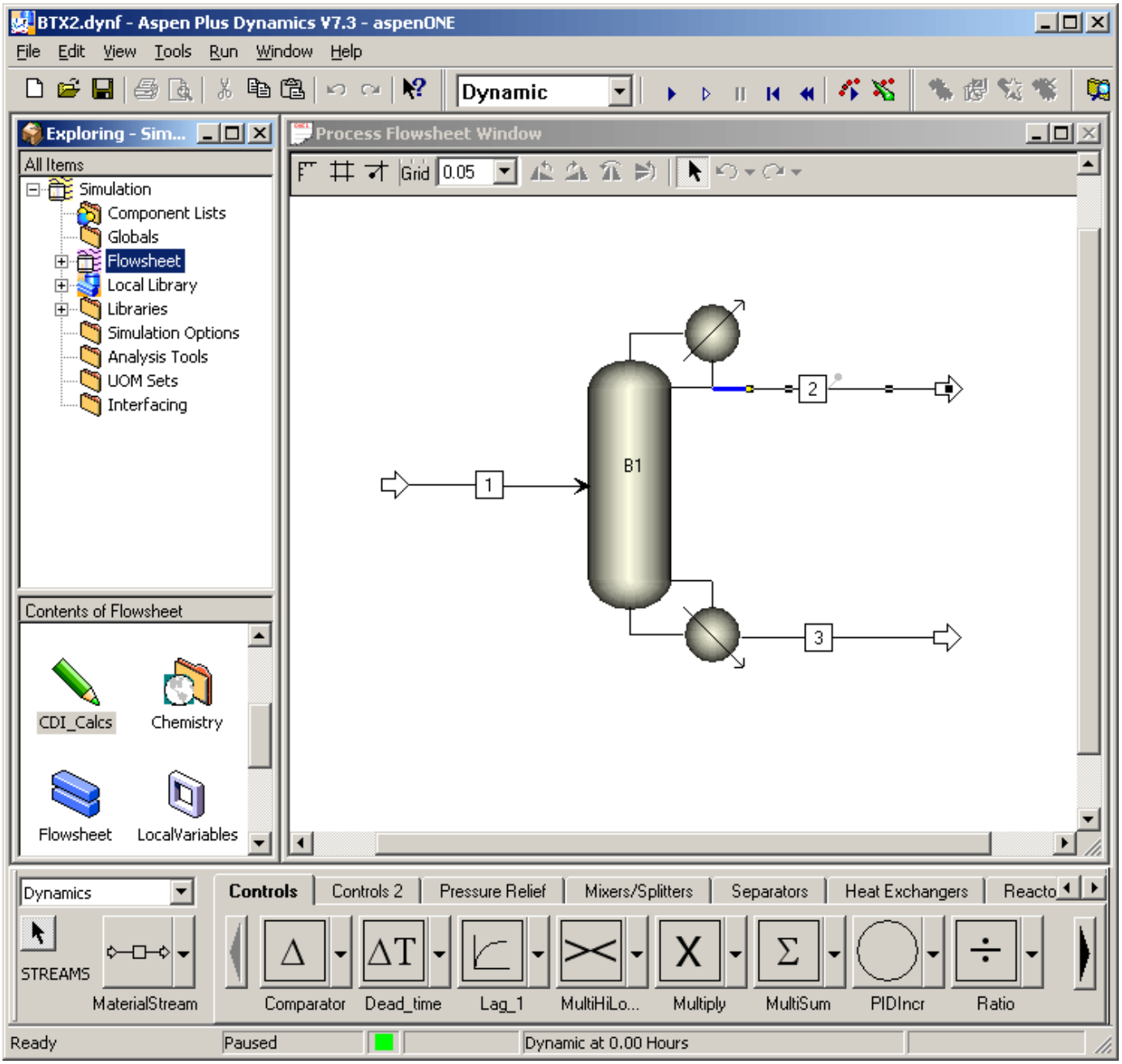
1. Hold the tower pressure constant.
2. Maintain 5% of toluene in the distillate (it is equivalent to maintain 95% of benzene in the distillate because the distillate only contains benzene and toluene).
3. Maintain 1.7% of the benzene in the bottoms.
4. Keep liquid levels in the sump and the reflux drum within specified limits.

Build High-Fidelity Plant Model in Aspen Plus Dynamics

Use an Aspen Plus RADFRAC block to define the tower's steady-state characteristics. In addition to the usual information needed for a steady-state simulation, you must specify tray hydraulics, tower sump geometry, and the reflux drum size. The trays are a sieve design spaced 18 inches apart. All trays have a 1.95 m in diameter with a 5 cm weir height. Nominal liquid depths are 0.67 m and 1.4875 m in the horizontal reflux drum and sump respectively.

The steady-state model is ported to Aspen Plus Dynamics (APD) for a flow-driven simulation. This neglects actuator dynamics and assumes accurate regulation of manipulated flow rates. By default, APD adds PI controllers to regulate the tower pressure and the two liquid levels. In this example, the default PI controllers are intentionally removed.

The APD model of the high-fidelity distillation tower is shown below:



Linearize Plant Using Aspen Plus Control Design Interface

Model Predictive Controller requires an LTI model of the plant. In this example, the plant inputs are:

1. Condenser duty (W)
2. Reboiler duty (W)
3. Reflux mass flow rate (kg/h)
4. Distillate mass flow rate (kg/h stream #2)
5. Bottoms mass flow rate (kg/h stream #3)

6 Feed molar flow rate (kmol/h stream #1)

The plant outputs are:

1 Tower pressure (in the condenser: stage 1, bar)
2 Reflux drum liquid level (m)
3 Sump liquid level (m)
4 Mass fraction toluene in the distillate
5 Mass fraction benzene in the bottoms

Aspen Plus Dynamics provides a Control Design Interface (CDI) tool that linearizes a dynamic model at a specified condition.

The following steps are taken to obtain the linear plant model in Aspen Plus Dynamics.

`Step 1`: Add a script to the APD model under the `Flowsheet` folder. In this example, the script name is `CDI_Calcs` (as shown above) and it contains the following APD commands:

```
Set Doc = ActiveDocument
set CDI = Doc.CDI
CDI.Reset
CDI.AddInputVariable "blocks(""B1"").condenser(1).QR"
CDI.AddInputVariable "blocks(""B1"").QrebR"
CDI.AddInputVariable "blocks(""B1"").Reflux.FmR"
CDI.AddInputVariable "streams(""2"").FmR"
CDI.AddInputVariable "streams(""3"").FmR"
CDI.AddInputVariable "streams(""1"").FR"
CDI.AddOutputVariable "blocks(""B1"").Stage(1).P"
CDI.AddOutputVariable "blocks(""B1"").Stage(1).Level"
CDI.AddOutputVariable "blocks(""B1"").SumpLevel"
CDI.AddOutputVariable "streams(""2"").Zmn(""TOLUENE"")"
CDI.AddOutputVariable "streams(""3"").Zmn(""BENZENE"")"
CDI.Calculate
```

`Step 2`: Initialize the APD model to the nominal steady-state condition.

`Step 3`: Invoke the script, which generates the following text files:

- `cdi_A.dat`, `cdi_B.dat`, `cdi_C.dat` define the A, B, and C matrices of a standard continuous-time LTI state-space model. D matrix is zero. The A, B, C matrices are sparse matrices.
- `cdi_list.lis` lists the model variables and their nominal values.
- `cdi_G.dat` defines the input/output static gain matrix at the nominal condition. The gain matrix is also a sparse matrix.

In this example, `cdi_list.lis` includes the following information:

```
A matrix computed, number of non-zero elements = 1408
B matrix computed, number of non-zero elements = 26
C matrix computed, number of non-zero elements = 20
G matrix computed, number of non-zero elements = 30
Number of state variables:    120
Number of input variables:      6
Number of output variables:     5
Input variables:
   1        -3690034.247458334  BLOCKS("B1").Condenser(1).QR
```

```
   2            3819023.193875  BLOCKS("B1").QRebR
   3            22135.96620144  BLOCKS("B1").Reflux.FmR
   4            11717.39655353  STREAMS("2").FmR
   5            34352.86345834  STREAMS("3").FmR
   6                       500  STREAMS("1").FR
Output variables:
   1         1.100022977953499  BLOCKS("B1").Stage(1).P
   2        0.6700005140605662  BLOCKS("B1").Stage(1).Level
   3                    1.4875  BLOCKS("B1").SumpLevel
   4       0.05002582161855798  STREAMS("2").Zmn("TOLUENE")
   5       0.01705308738356429  STREAMS("3").Zmn("BENZENE")
```

The nominal values of the state variables listed in the file are ignored because they are not needed in the MPC design.

Create Scaled and Reduced LTI State-Space Model

`Step 1`: Convert the CDI-generated sparse-matrices to a state-space model.

Load state-space matrices from the CDI data files to MATLAB workspace and convert the sparse matrices to full matrices.

```
load mpcdistillation_cdi_A.dat
load mpcdistillation_cdi_B.dat
load mpcdistillation_cdi_C.dat
A = full(spconvert(mpcdistillation_cdi_A));
B = full(spconvert(mpcdistillation_cdi_B));
C = full(spconvert(mpcdistillation_cdi_C));
D = zeros(5,6);
```

It is possible that an entire sparse matrix row or column is zero, in which case the above commands are insufficient. Use the following additional checks to make sure A, B, and C have the correct dimensions:

```
[nxAr,nxAc] = size(A);
[nxB,nu] = size(B);
[ny,nxC] = size(C);
nx = max([nxAr, nxAc, nxB, nxC]);
if nx > nxC
    C = [C, zeros(ny,nx-nxC)];
end
if nx > nxAc
    A = [A zeros(nxAr,nx-nxAc)];
end
if nx > nxAr
    nxAc = size(A,2);
    A = [A; zeros(nx-nxAr, nxAc)];
end
if nxB < nx
    B = [B; zeros(nx-nxB,nu)];
end
```

`Step 2`: Scale the plant signals.

It is good practice, if not essential, to convert plant signals from engineering units to a uniform dimensionless scale (e.g., 0-1 or 0-100%). One alternative is to define scale factors as part of a Model Predictive Controller design. This can simplify controller tuning significantly. For example, see, “Using Scale Factors to Facilitate MPC Weights Tuning” on page 1-32.

In the present example, however, we will use a model reduction procedure prior to controller design, and we therefore scale the plant model, using the scaled model in both model reduction and controller design. We define a *span* for each input and output, i.e., the difference between expected maximum and minimum values in engineering units. Also record the *nominal* and *zero* values in engineering units to facilitate subsequent conversions.

```
U_span = [2*(-3690034), 2*3819023, 2*22136, 2*11717, 2*34353, 2*500];
U_nom = 0.5*U_span;
U_zero = zeros(1,6);
Y_nom = [1.1, 0.67, 1.4875, 0.050026, 0.017053];
Y_span = [0.4, 2*Y_nom(2:5)];
Y_zero = [0.9, 0, 0, 0, 0];
```

Scale the B and C matrices such that all input/output variables are expressed as percentages.

```
B = B.*(ones(nx,1)*U_span);
C = C./(ones(nx,1)*Y_span)';
```

`Step 3`: Define the state-space plant model.

```
G = ss(A,B,C,D);
G.TimeUnit = 'hours';
G.u = {'Qc','Qr','R','D','B','F'};
G.y = {'P','RLev','Slev','xD','xB'};
```

`Step 4`: Reduce model order.

Model reduction speeds up the calculations with negligible effect on prediction accuracy. Use the `hsvd` command to determine which states can be safely discarded. Use the `balred` function to remove these states and reduce model order.

```
[hsv, baldata] = hsvd(G);
order = find(hsv>0.01,1,'last');
Options = balredOptions('StateElimMethod','Truncate');
G = balred(G,order,baldata,Options);
```

The original model has 120 states and the reduced model has only 16 states. Note that the `Truncate` option is used in the `balred` function to preserve a zero D matrix. The model has two poles at zero, which correspond to the two liquid levels.

Test Accuracy of the Linear Plant Model

Before continuing with the MPC design, it is good practice to verify that the scaled LTI model is accurate for small changes in the plant inputs. To do so, you need to compare the response of the nonlinear plant in APD and the response of linear model `G`.

`Step 1`: To obtain the response of the nonlinear plant, create a Simulink model and add the Aspen Modeler Block to it.

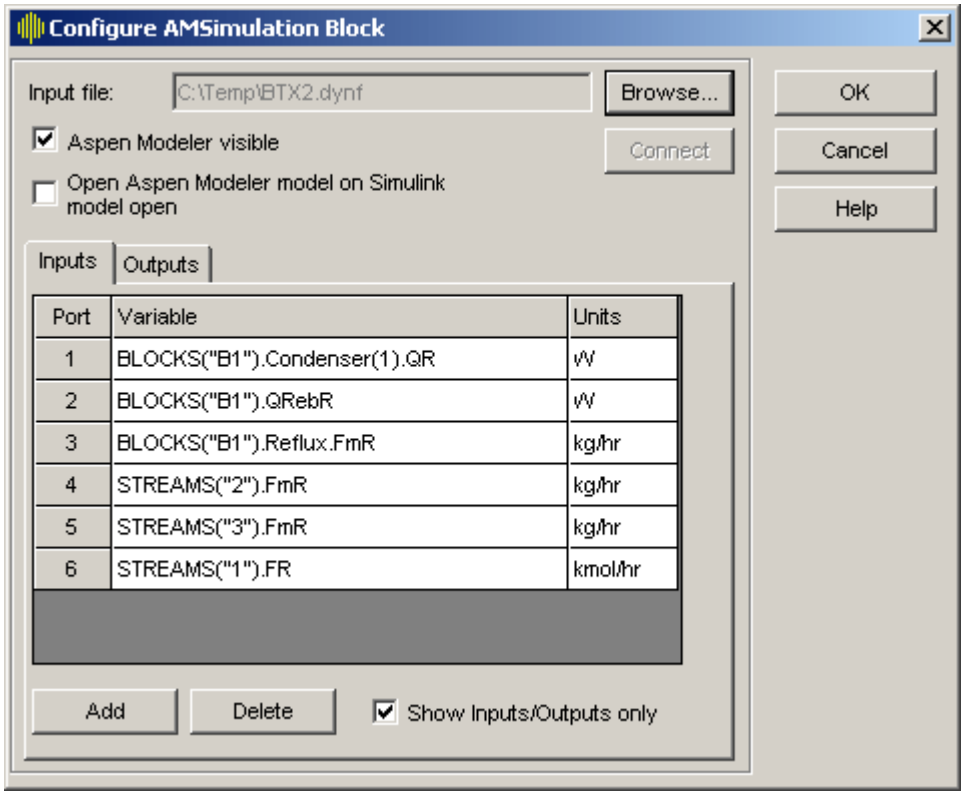
The block is provided by Aspen Plus Dynamics in their **AMSimulink** library.

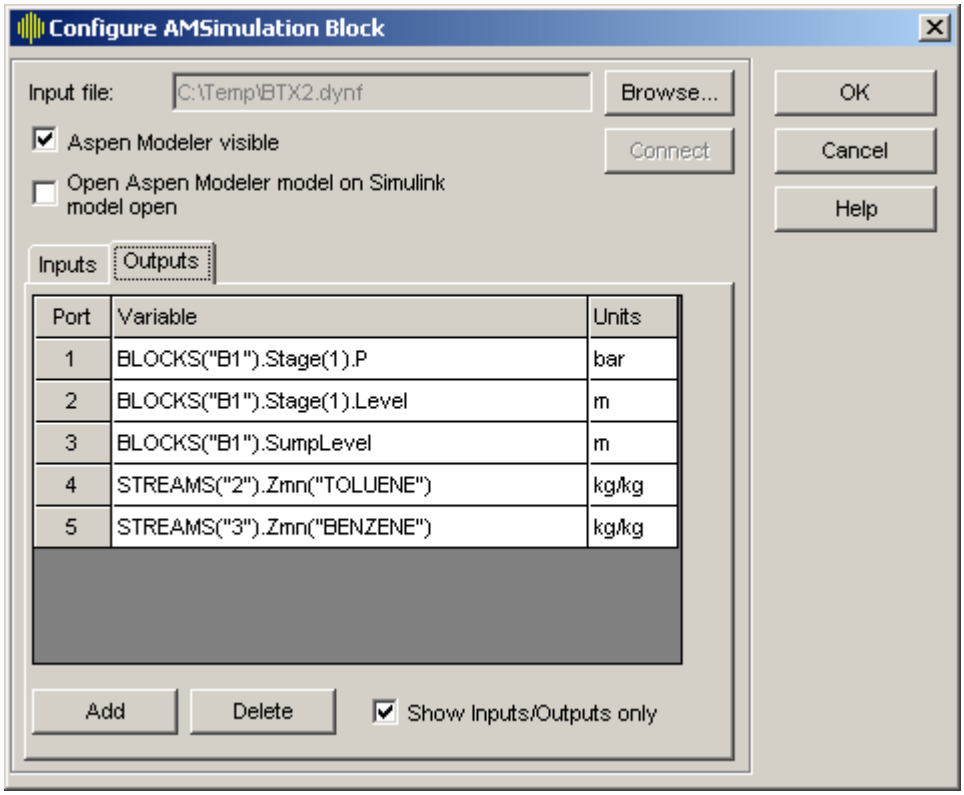
`Step 2`: Double-click the block and provide the location of the APD model.

The APD model information is then imported into Simulink. For large APD models, the importing process may take some time.

`Step 3`: Specify input and output signals in the **AMSimulation** block.

Use the same variable names and sequence as in the CDI script.

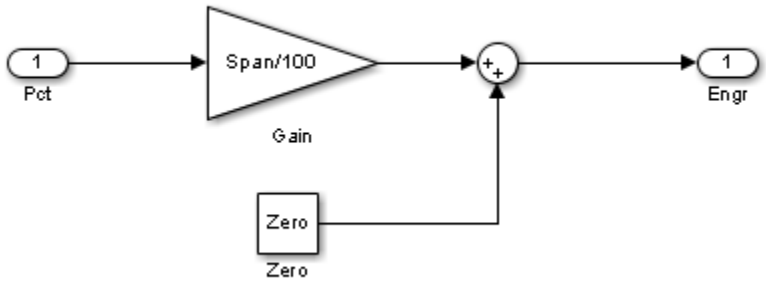




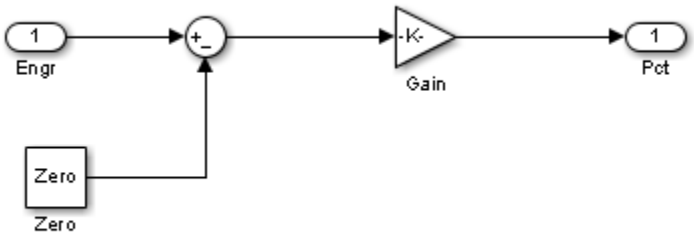
The block now shows inports and outports for each signal that you defined.

`Step 4`: Expand the Simulink model with an input signal coming from the variable `Umat` and an output signal saved to variable `Ypct_NL`. Both variables are created in Step 5.

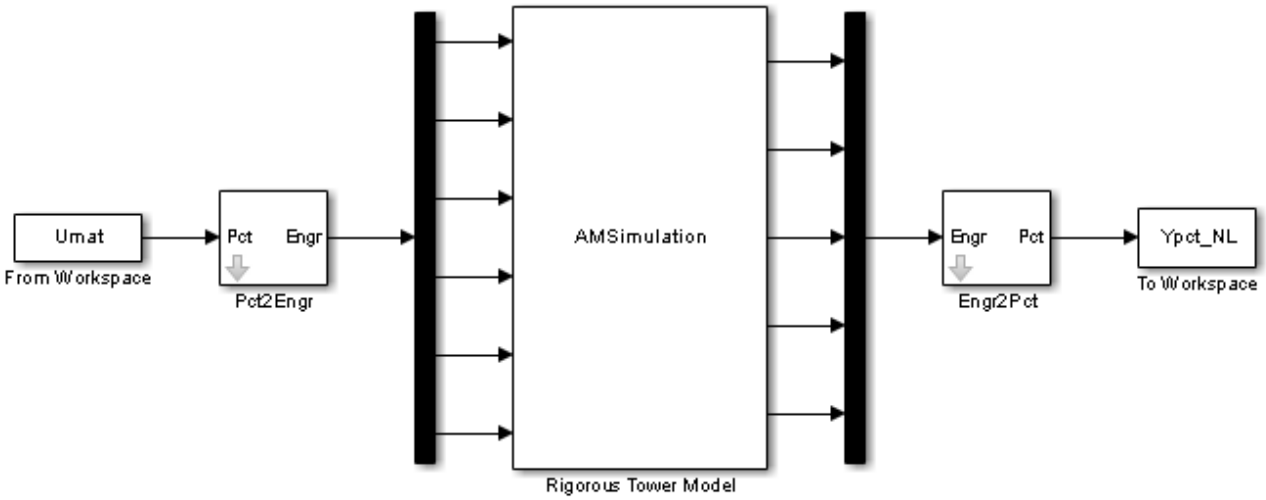
Since `Umat` is in the percentage units, the **Pct2Engr** block is implemented to convert from percentage units to engineering units.



Since `Ypct_NL` is in the percentage units, the **Engr2Pct** block is implemented to convert from engineering units to percentage units.



With everything connected and configured, the model appears as follows:



`Step 5`: Verify linear model with cosimulation.

In this example, 1 percent increase in the scaled reflux rate (input #3) is used as the excitation signal to the plant.

```
U_nom_pct = (U_nom - U_zero)*100./U_span;    % Convert nominal condition from engineering units to
Y_nom_pct = (Y_nom - Y_zero)*100./Y_span;
Tend = 1;                                    % Simulation duration (1 hour)
t = (0:1/60:Tend)';                          % Sample period is 1 minute
nT = length(t);
Upct = ones(nT,1)*U_nom_pct;
DUpct = zeros(nT,6);
DUpct(:,3) = ones(nT,1);                     % Input signal where step occurs in channel #3
```

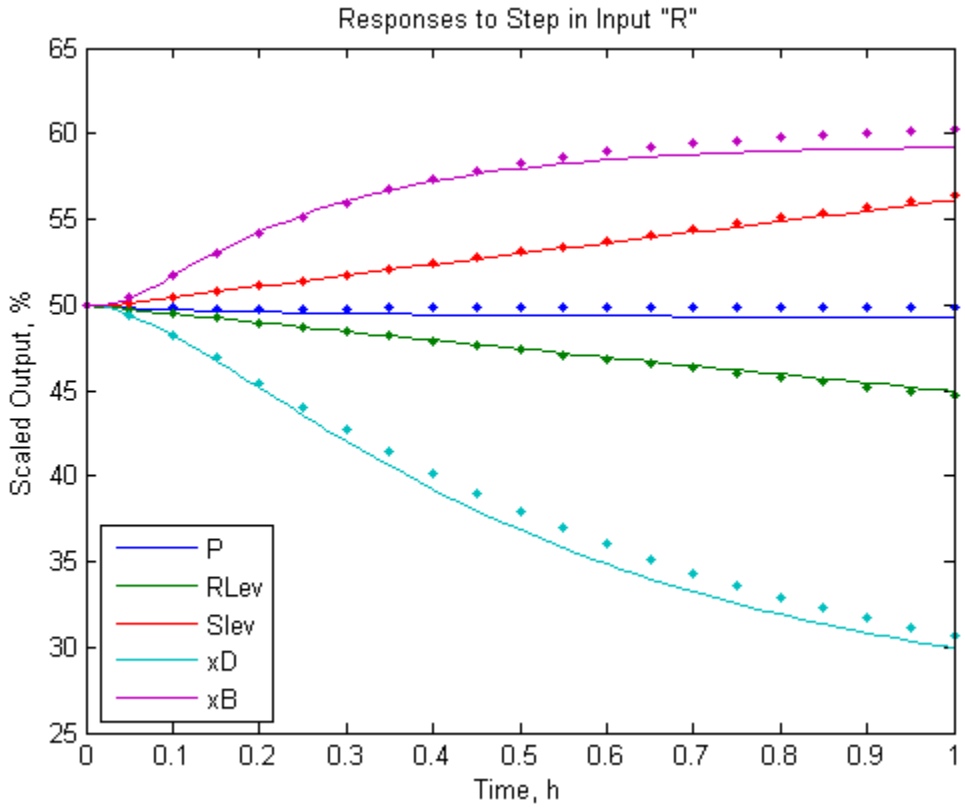
The response of the linear plant model is computed using the `lsim` command and stored in variable `Ypct_L`.

```
Ypct_L = lsim(G,DUpct,t);
Ypct_L = Ypct_L + ones(nT,1)*Y_nom_pct;
```

The response of the nonlinear plant is obtained through cosimulation between Simulink and Aspen Plus Dynamics. The excitation signal `Umat` is constructed as below. The result is stored in variable `Ypct_NL`.

```
Umat = [t, Upct+DUpct];
```

Compare the linear and nonlinear model responses.



The LTI model predictions track the nonlinear responses well. The amount of prediction error is acceptable. In any case, a Model Predictive Controller must be tuned to accommodate prediction errors, which are inevitable in applications.

You can repeat the above steps to verify similar agreement for the other five inputs.

Design Model Predictive Controller

Given an LTI prediction model, you are ready to design a Model Predictive Controller. In this example, the manipulated variables are the first five plant inputs. The sixth plant input (feed flow rate) is a measured disturbance for feed-forward compensation. All the plant outputs are measured.

`Step 1`: Augment the plant to model unmeasured load disturbances.

Lacking any more specific details regarding load disturbances, it is common practice to assume an unmeasured load disturbance occurring at each of the five inputs. This allows the MPC state estimator to eliminate offset in each controlled output when a load disturbance occurs.

In this example, 5 unmeasured load disturbances are added to the plant model `G`. In total, there are now 11 inputs to the prediction model `Gmpc`: 5 manipulated variables, 1 measured disturbance, and 5 unmeasured disturbances.

```
Gmpc = ss(G.A,G.B(:,[1:6,1:5]),G.C,zeros(5,11),'TimeUnit','hours');
InputName = cell(1,11);
for i = 1:5
    InputName{i} = G.InputName{i};
    InputName{i+6} = [G.InputName{i}, '-UD'];
```

```
end
InputName{6} = G.InputName{6};
Gmpc.InputName = InputName;
Gmpc.InputGroup = struct('MV',1:5,'MD',6,'UD',7:11);
Gmpc.OutputName = G.OutputName;
```

`Step 2`: Create an initial model predictive controller and specify sample time and horizons.

In this example, the controller sample period is 30 seconds. The prediction horizon is 60 intervals (30 minutes), which is large enough to make the controller performance insensitive to further increases of the prediction horizon. The control horizon is 4 intervals (2 minutes), which is relatively small to reduce computational effort.

```
Ts = 30/3600;       % sample time
PH = 60;            % prediction horizon
CH = 4;             % control horizon
MPCobj = mpc(Gmpc,Ts,PH,CH);  % MPC object
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

`Step 3`: Specify weights for manipulated variables and controlled outputs.

Weights are key tuning adjustments in MPC design and they should be chosen based on your control objectives.

There is no reason to hold a particular MV at a setpoint, so set the `Weights.ManipulatedVariables` property to zero:

```
MPCobj.Weights.ManipulatedVariables = [0, 0, 0, 0, 0];
```

The distillate product (MV #4) goes to storage. The only MV affecting downstream unit operations is the bottoms rate (MV #5). To discourage rapid changes in bottoms rate, retain the default weight of 0.1 for its rate of change. Reduce the other rate of change weights by a factor of 10:

```
MPCobj.Weights.ManipulatedVariablesRate = [0.01, 0.01, 0.01, 0.01, 0.1];
```

The control objectives provide guidelines to choose weights on controlled outputs:

1. The tower pressure must be regulated tightly for safety reasons and for minimizing upsets in tray temperatures and hydraulics. (objective #1)
2. The distillate composition must also be regulated tightly. (objective #2)
3. The bottoms composition can be regulated less tightly. (objective #3)
4. The liquid levels are even less important. (objective #4)

With these priorities in mind, weights on controlled outputs are chosen as follows::

```
MPCobj.Weights.OutputVariables = [10, 0.1, 0.1, 1, 0.5];
```

Scaling the model simplifies the choice of the optimization weights. Otherwise, in addition to the relative priority of each variable, you would also have to consider the relative magnitudes of the variables and choose weights accordingly.

`Step 4`: Specify nominal plant input/output values.

In this example, the nominal values are scaled as percentages. MPC controller demands that the nominal values for unmeasured disturbances must be zero.

```
MPCobj.Model.Nominal.U = [U_nom_pct'; zeros(5,1)];
MPCobj.Model.Nominal.Y = Y_nom_pct';
```

`Step 5`: Adjust state estimator gain.

Adjusting the state estimator gain affects the disturbance rejection performance. Increasing the state estimator gain (e.g. by increasing the gain of the input/output disturbance model) makes the controller respond more aggressively towards output changes (because the controller assumes the main source of the output changes is a disturbance, instead of measurement noise). On the other hand, decreasing the state estimator gain makes the closed-loop system more robust.

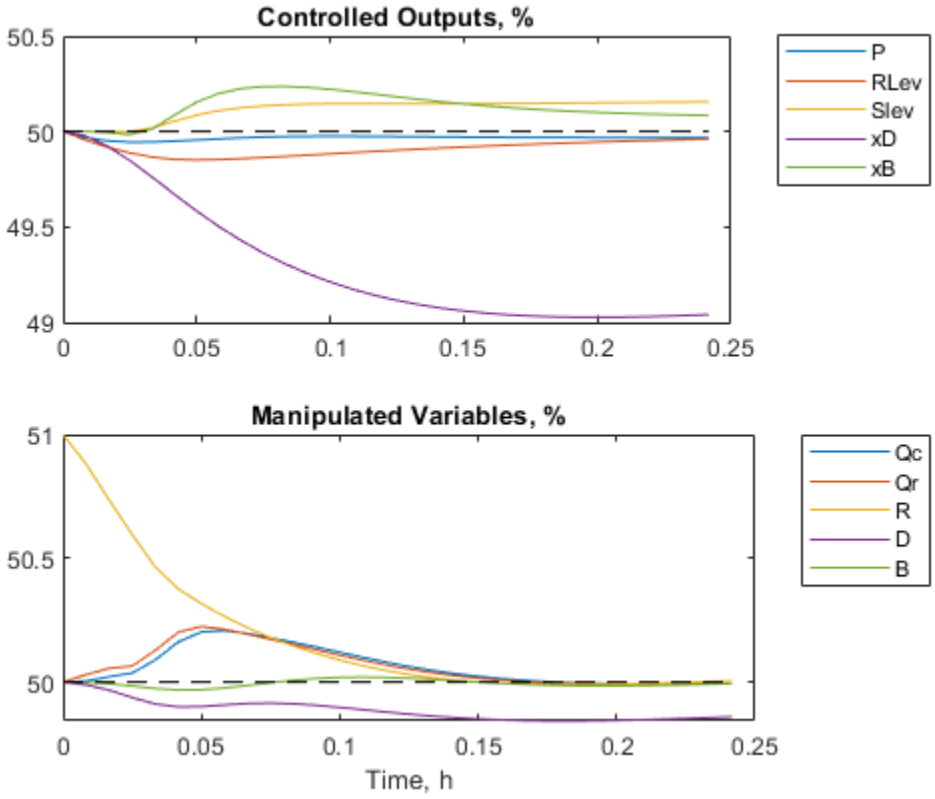
First, check whether using the default state estimator provides a decent disturbance rejection performance.

Simulate the closed-loop response to a 1% unit step in reflux (MV #3) in MATLAB. The simulation uses `G` as the plant, which implies no model mismatch.

```
T = 30;                                 % Simulation time
r = Y_nom_pct;                          % Nominal setpoints
v = U_nom_pct(6);                       % No measured disturbance
SimOptions = mpcsimopt(MPCobj);
SimOptions.InputNoise = [0 0 1 0 0];    % 1% unit step in reflux
[y_L,t_L,u_L] = sim(MPCobj, T, r, v, SimOptions); % Closed-loop simulation
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #7 is integrated white noise.
   Assuming unmeasured input disturbance #8 is integrated white noise.
   Assuming unmeasured input disturbance #9 is integrated white noise.
   Assuming unmeasured input disturbance #10 is integrated white noise.
   Assuming unmeasured input disturbance #11 is integrated white noise.
   Assuming no disturbance added to measured output channel #1.
   Assuming no disturbance added to measured output channel #4.
   Assuming no disturbance added to measured output channel #5.
   Assuming no disturbance added to measured output channel #2.
   Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

```
% plot responses
f1 = figure();
subplot(2,1,1);
plot(t_L,y_L,[0 t_L(end)],[50 50],'k--')
title('Controlled Outputs, %')
legend(Gmpc.OutputName,'Location','NorthEastOutside')
subplot(2,1,2);
plot(t_L,u_L(:,1:5),[0 t_L(end)],[50 50],'k--')
title('Manipulated Variables, %')
legend(Gmpc.InputName(1:5),'Location','NorthEastOutside')
xlabel('Time, h')
```



The default estimator provides sluggish load rejection. In particular, the critical xD output drops to 49% and has just begun to return to the setpoint after 0.25 hours.

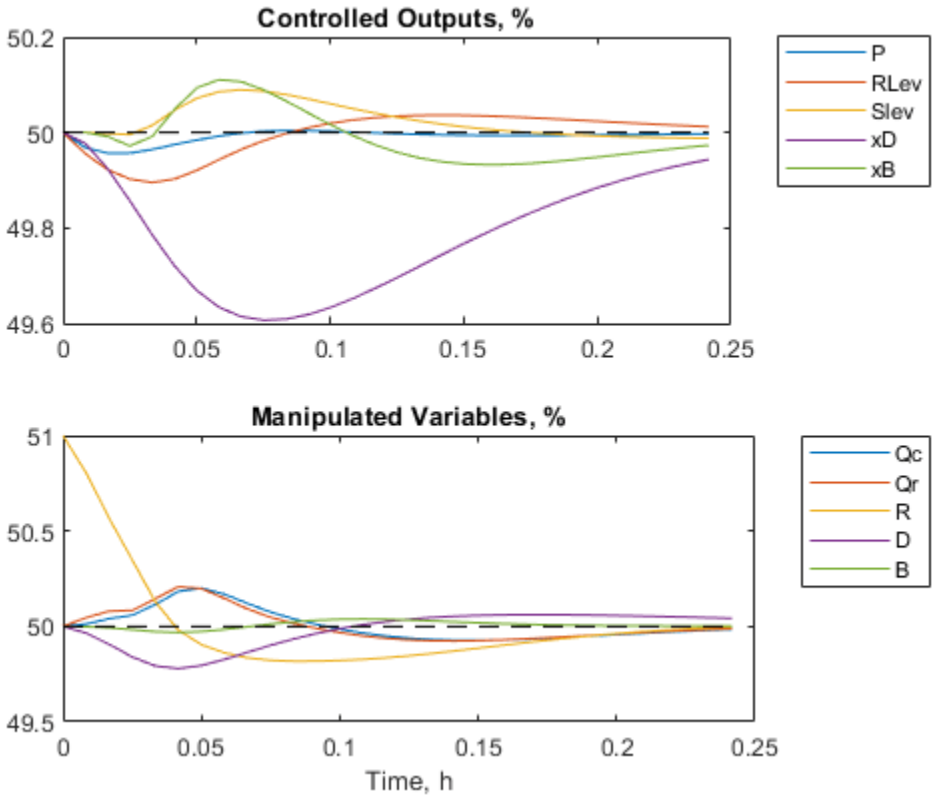
Secondly, increase the estimator gain by multiplying the default input disturbance model gain by a factor of 25.

```
EstGain = 25;                          % factor of 25
Gd = getindist(MPCobj);                % get default input disturbance model
Gd_new = EstGain*Gd;                   % create new input disturbance model
setindist(MPCobj,'Model',Gd_new); % set input disturbance model
[y_L,t_L,u_L] = sim(MPCobj,T,r,v,SimOptions); % Closed-loop simulation
```

```
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
   Assuming no disturbance added to measured output channel #4.
   Assuming no disturbance added to measured output channel #5.
   Assuming no disturbance added to measured output channel #2.
   Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

```
% plot responses
f2 = figure();
subplot(2,1,1);
plot(t_L,y_L,[0 t_L(end)],[50 50],'k--')
title('Controlled Outputs, %')
legend(Gmpc.OutputName,'Location','NorthEastOutside')
subplot(2,1,2)
plot(t_L,u_L(:,1:5),[0 t_L(end)],[50 50],'k--')
```

```
title('Manipulated Variables, %')
legend(Gmpc.InputName(1:5),'Location','NorthEastOutside')
xlabel('Time, h')
```



Now, the peak deviation in `xD` is 50% less than the default case and `xD` returns to its setpoint much faster. Other variables also respond more rapidly.

Thirdly, look at the reflux response (#3 in the "Manipulated Variables" plot). Because the disturbance is a 1% unit step, the response begins at 51% and its final value is 50% at steady state. The reflux response overshoots by 20% (reaching 49.8%) before settling. This amount of overshoot is acceptable.

If the estimator gain were increased further (e.g. by a factor of 50), the controller overshoot would increase too. However, such aggressive behavior is unlikely to be robust when applied to the nonlinear plant model.

You can introduce other load disturbances to verify that disturbance rejection is now rapid in all cases.

Scaling the model also simplifies disturbance model tuning. Otherwise, you would need to adjust the gain of each channel in the disturbance model to achieve good disturbance rejection for all loads.

Generally, you next check the response to setpoint changes. If the response is too aggressive, you can use setpoint filter to smooth it. Setpoint filter has no effect on load disturbance rejection and thus can be tuned independently.

Cosimulate MPC Controller and Nonlinear Plant

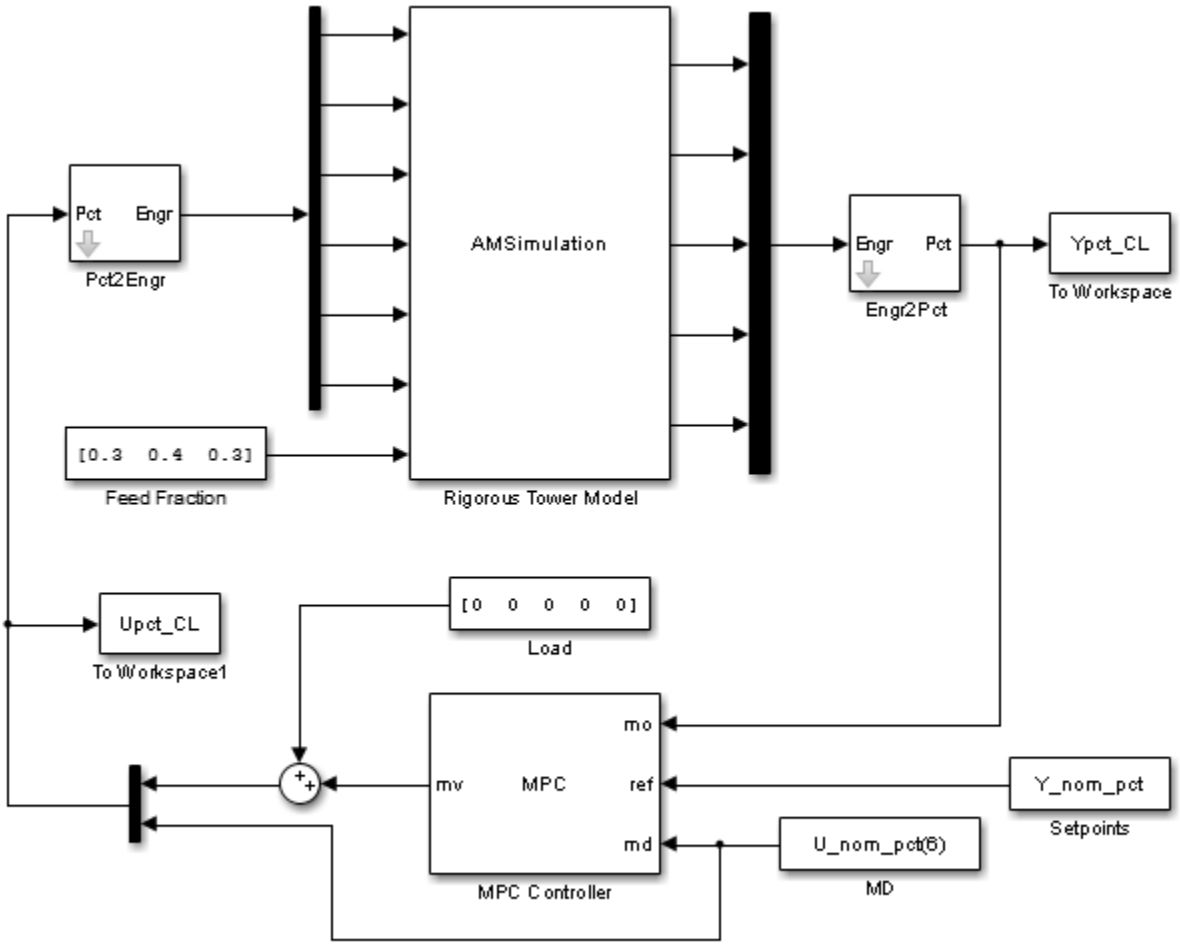
Use cosimulation to determine whether the MPC design is robust enough to control the nonlinear plant model.

`Step 1`: Add constraints to the MPC controller

Because the nonlinear plant model has input and output constraints during operation, MV and OV constraints are defined in the MPC controller as follows:

```
MV = MPCobj.MV;
OV = MPCobj.OV;
% Physical bounds on MVs at 0 and 100
for i = 1:5
    MV(i).Min = 0;
    MV(i).Max = 100;
end
MPCobj.MV = MV;
% Keep liquid levels greater than 25% and less than 75% of capacity.
for i = 2:3
    OV(i).Min = 25;
    OV(i).Max = 75;
end
MPCobj.OV = OV;
```

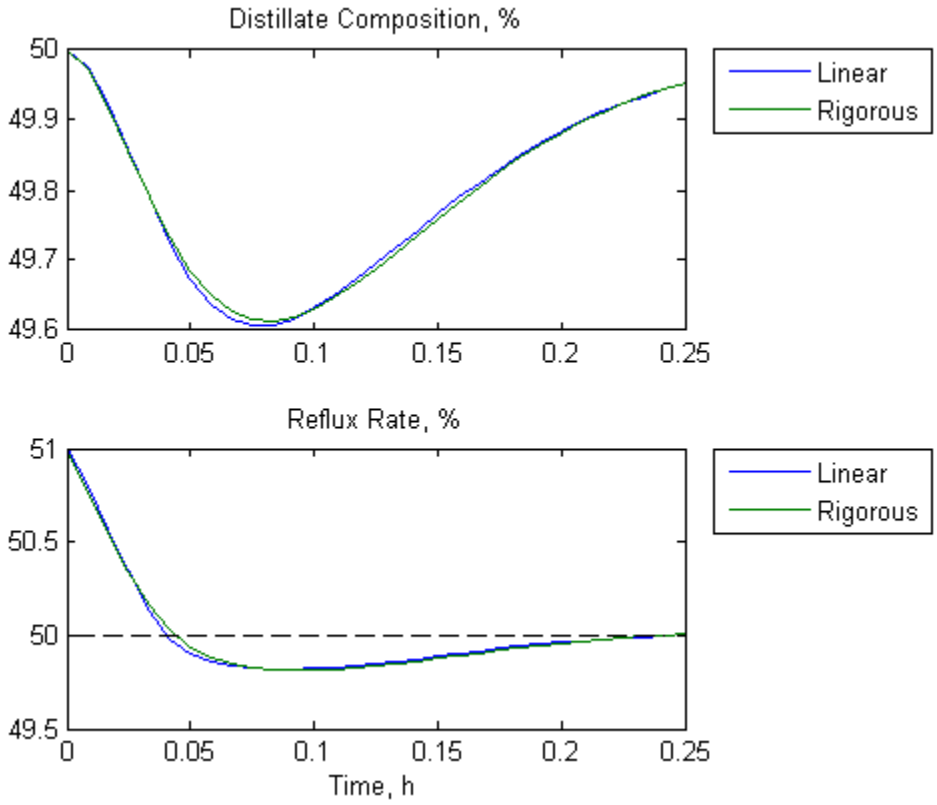
`Step 2`: Build Simulink model for cosimulation.



The model can simulate 1% unit step in reflux (MV #3). It can also simulate a change in feed composition, which is a common disturbance and differs from the load disturbances considered explicitly in the design.

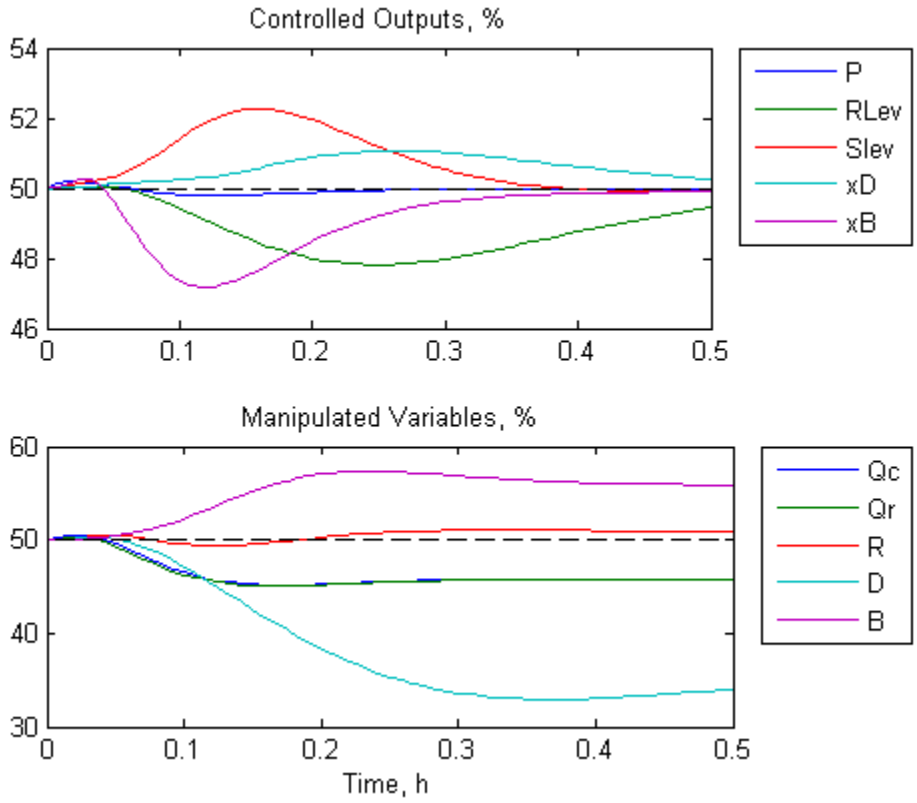
`Step 3`: Simulate 1% unit step in reflux (MV #3). Compare the closed-loop responses between using the linear plant model and using the nonlinear plant model.

Plot distillate product composition (`xD`) and the reflux rate (`R`):



In cosimulation, the model predictive controller rejects the small load disturbance in a manner almost identical to the linear simulation.

`Step 4`: Simulate a large decrease of benzene fraction (from 0.3 to 0.22) in the feed stream. Compare the closed-loop responses between using the linear and nonlinear plant models.



The drop in benzene fraction requires a sustained decrease in the distillate rate and a corresponding increase in the bottoms rate. There are also sustained drops in the heat duties and a minor increase in the reflux. All MV adjustments are smooth and all controlled outputs are nearly back to their setpoints within 0.5 hours.

See Also

`mpc`

Simulate Linear MPC Controller with Nonlinear Plant using Successive Linearizations

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 can linearize the plant (see “Linearization Using Model Linearizer in Simulink Control Design”) and design a controller for the linear model (see “Design MPC Controller in Simulink”). 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.

Nonlinear CSTR Application

The `CSTR` model described in “Linearize Simulink Models” is a strongly nonlinear system. As shown in “Design MPC Controller in Simulink”, 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. For more details on this approach, see [1] and [2].

Example Code for Successive Linearization

In the following code, the simulation begins at the nominal operating point of the `CSTR` model (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);
tsave = [];
usave = [];
ysave = [];
rsave = [];
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
```

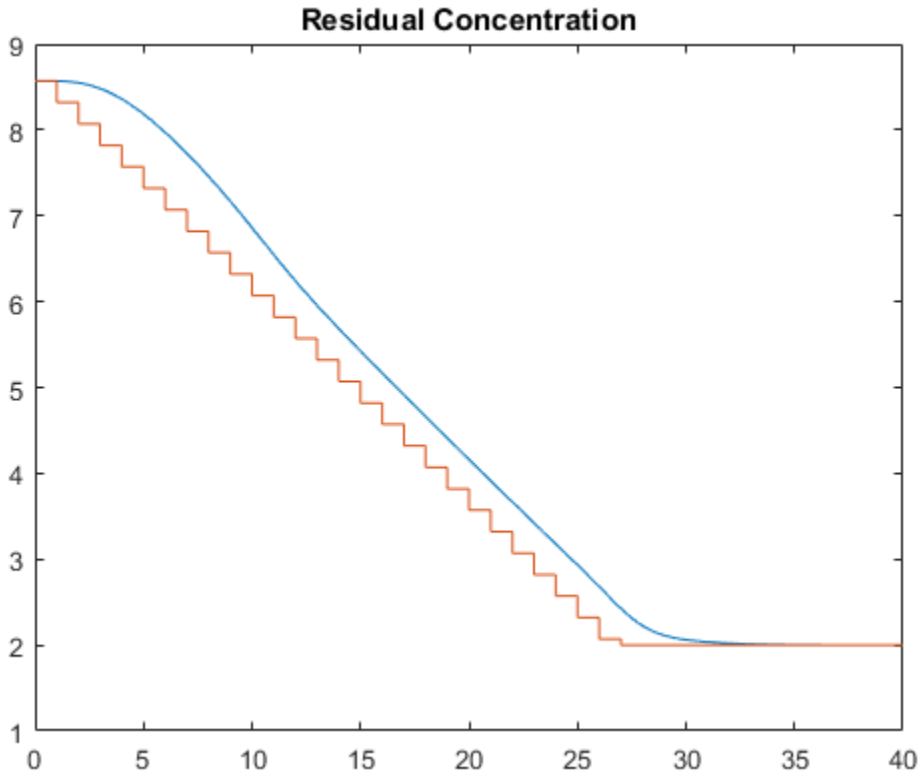
```
Model.Nominal.U = [0 0 u];
Model.Nominal.X = xp;
Model.Nominal.Y = yp;

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 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];
usave = [usave; up(ones(length(T),1),:)];
rsave = [rsave; r(ones(length(T),1),:)];
```

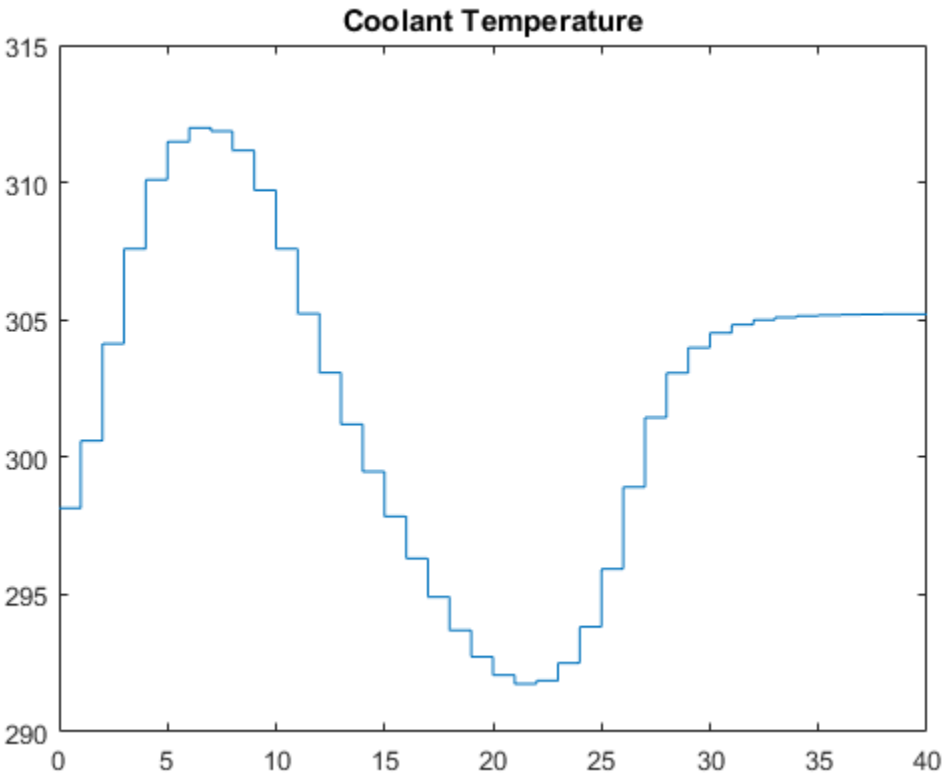
```
    xp = XP(end,:)';

    t = t + Ts;
end

figure(1)
plot(tsave,[ysave(:,2) rsave])
title('Residual Concentration')
figure(2)
plot(tsave,usave(:,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).


Residual Concentration
9
8
7
6
5
4
3
2
1
0
5
10
15
20
25
30
35
40


Coolant Temperature
315
310
305
300
295
290
0
5
10
15
20
25
30
35
40

- Function `linearize` relinearizes the plant as its state evolves. This function was discussed previously in “Linearization Using MATLAB Code”.
- 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 extended state vector of the controller, `x`. 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. 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).

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 6-7
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-17
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24
- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-33

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 can be unsuitable.

As shown in “Optimization Problem” on page 2-7, if no QP inequality constraints are active for a given x vector, then the optimal MV adjustments become a time invariant affine 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 an affine 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 affine functions 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, and 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 number of 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.

See Also

More About

- “Design Workflow for Explicit MPC” on page 6-4
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-7
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-17

- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24

Design Workflow for Explicit MPC

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 before creating an explicit MPC controller. 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 “Choose Sample Time and Horizons” on page 1-2.
  - 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 important that you do 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 independent variables of the control law, that is,

$$x_l \leq x(k) \leq x_u,$$

the `generateExplicitMPC` command determines n_r regions. Each of these regions is defined by an inequality constraint and the corresponding control law constants:

$$H_i x(k) \le K_i, \qquad i = 1 \text{ to } n_r$$
$$u(k) = F_i x(k) + G_i, \qquad i = 1 \text{ to } n_r$$

The `explicitMPC` 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. As long as $x(k)$ stays within the specified bounds and you retain all n_r regions, the explicit MPC object provides 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 require many regions ($n_r >> 100$) 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. Sometimes, this reduction 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. During operation, if the current $x(k)$ is not contained in any of the retained regions, the explicit MPC returns 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, $H_j x(k) \le K_j$, 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, $x_l \le \mathrm{x}(k) \le x_u$. 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 $H_i x(k) \le K_i$, 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 the time required 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 `mpcmoveExplicit` 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

`generateExplicitMPC` | `mpcmoveExplicit` | Explicit MPC Controller

More About

- “Explicit MPC” on page 6-2
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-7
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-17
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24

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.

For an example that controls a double integrator with an implicit MPC controller, see “Model Predictive Control of a Single-Input-Single-Output Plant”.

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 a sample period of 0.1 seconds, and prediction and control horizons of 10 and 3 steps respectively.

```
Ts = 0.1;
mpcobj = mpc(plant, Ts, 10, 3);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify actuator saturation limits as manipulated variable 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 traditional implicit MPC. To generate an explicit MPC object from a traditional MPC one, you must specify the range for each controller state, reference signal, manipulated variable and measured disturbance. This allows for the multi-parametric quadratic programming problem associated to the MPC problem to be set up and solved, once for all, in the parameter space defined by these ranges.

Use the `generateExplicitRange` function to obtain a range structure where you can specify 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 measured
```

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];
```

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 an explicit MPC controller must be at least as large as the practical range.

```
range.Reference.Min = -2;
range.Reference.Max = 2;
```

Specify the manipulated variable ranges. If the manipulated variables are constrained, the ranges used to generate the explicit MPC controller must be at least as large as these limits.

```
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use `generateExplicitMPC` command to obtain an explicit MPC controller with the specified parameter ranges.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range)
```

```
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 computation
Type 'mpcobjExplicit.PiecewiseAffineSolution' for regions and gain in each solution.
```

Use the `simplify` function with the `'exact'` method to join pairs of regions whose corresponding gains are the same and whose union is a convex set. Doing so can reduce memory footprint of the explicit MPC controller without sacrificing any performance.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, 'exact')
```

```
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 c
Type 'mpcobjExplicitSimplified.PiecewiseAffineSolution' for regions and gain in each solution.
```

The number of piecewise affine regions 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.

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicitSimplified)
```

```
params =

  struct with fields:

                 State: [1x1 struct]
             Reference: [1x1 struct]
    MeasuredDisturbance: [1x1 struct]
    ManipulatedVariable: [1x1 struct]
```

All but two of the control laws free parameters must be fixed, the remaining two variables form the plot axes. In this example, you plot the first state variable against the second state variable, so you need to fix all the other parameters at a value within their respective ranges.

Leave the states free to vary.

```
params.State.Index = [];
params.State.Value = [];
```

Fix the reference signal.

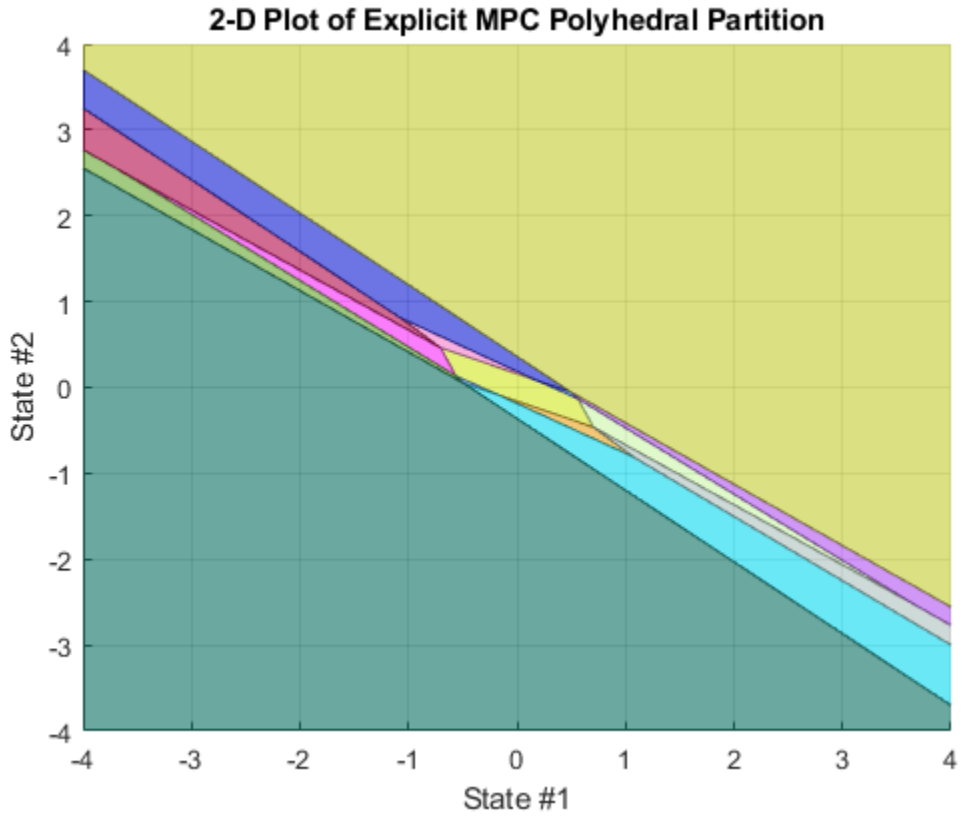
```
params.Reference.Index = 1;
params.Reference.Value = 0;
```

Fix the manipulated variable.

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Use `plotSection` command to plot the 2-D section defined by the two free parameters. For more information, see `plotSection`.

```
plotSection(mpcobjExplicitSimplified, params);
axis([-4 4 -4 4]);
grid
xlabel('State #1');
ylabel('State #2');
```



Simulate Using `mpcmove` Function

Compare closed-loop simulations for traditional implicit MPC and explicit MPC using the `mpcmove` and `mpcmoveExplicit` functions respectively.

Initialize variables 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 plant model used in simulation

```
sys = c2d(ss(plant),Ts);
xsys = [0;0];
xsysExplicit = xsys;
```

To obtain a pointer to the internal states for both controllers, use `mpcstate`.

```
xmpc = mpcstate(mpcobj);
xmpcExplicit = mpcstate(mpcobjExplicitSimplified);
```

Iteratively simulate the closed-loop response for both controllers.

```
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;
    UUExplicit(t+1)=uExplicit;

    % update plant state
    xsys = sys.A*xsys + sys.B*u;
    xsysExplicit = sys.A*xsysExplicit + sys.B*uExplicit;
end

% Norm of the differences between traditional and explicit controller signals.
fprintf('\nDifference between traditional and Explicit MPC responses using MPCMOVE command is %g\
    norm(UU-UUExplicit)+norm(YY-YYExplicit));


Difference between traditional and Explicit MPC responses using MPCMOVE command is 1.79044e-13
```

Simulate Using `sim` Function

Compare closed-loop simulations between traditional and explicit MPC using the `sim` command.

```
Tf = 5/Ts;                      % number of simulation iterations
[y1,t1,u1] = sim(mpcobj,Tf,1);  % simulate with traditional MPC
[y2,t2,u2] = sim(mpcobjExplicitSimplified,Tf,1); % simulate 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 measured
```

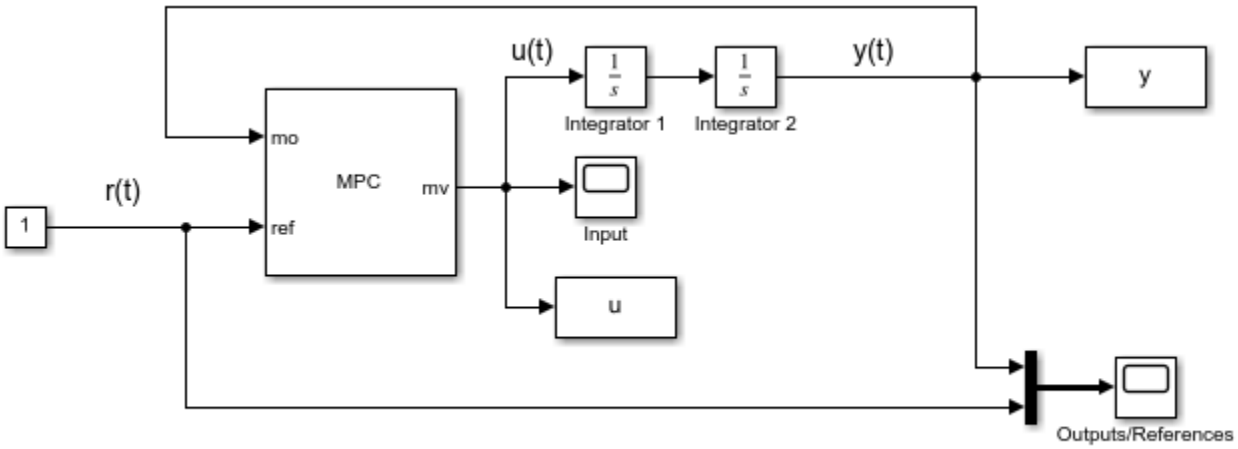
The simulation results are identical.

```
fprintf('\nDifference between traditional and Explicit MPC responses using SIM command is %g\n',
    norm(u2-u1)+norm(y2-y1));


Difference between traditional and Explicit MPC responses using SIM command is 1.79056e-13
```

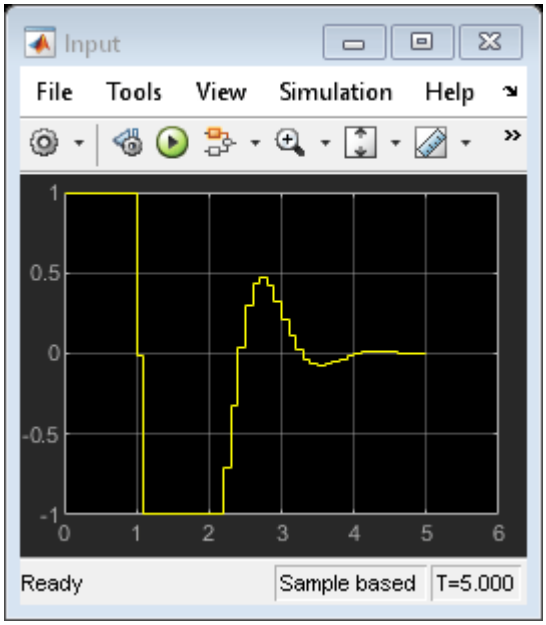
Simulate Using Simulink

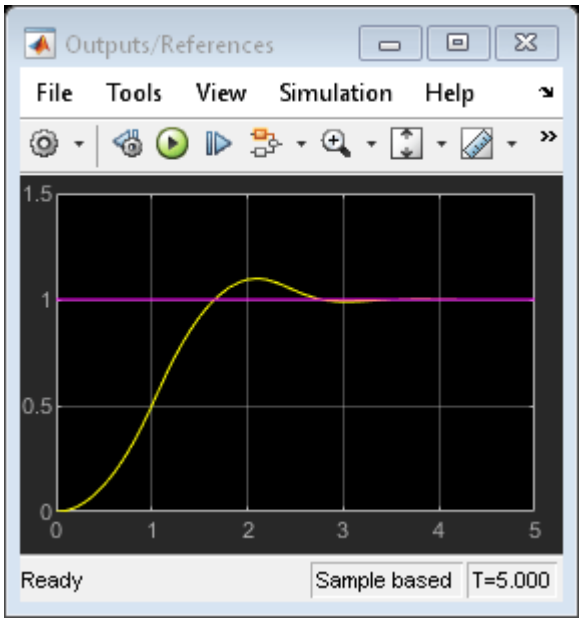
Simulate the traditional MPC controller in Simulink. The MPC Controller block is configured to use `mpcobj` as its controller.

```
mdl = 'mpc_doubleint';
open_system(mdl)
sim(mdl)
```



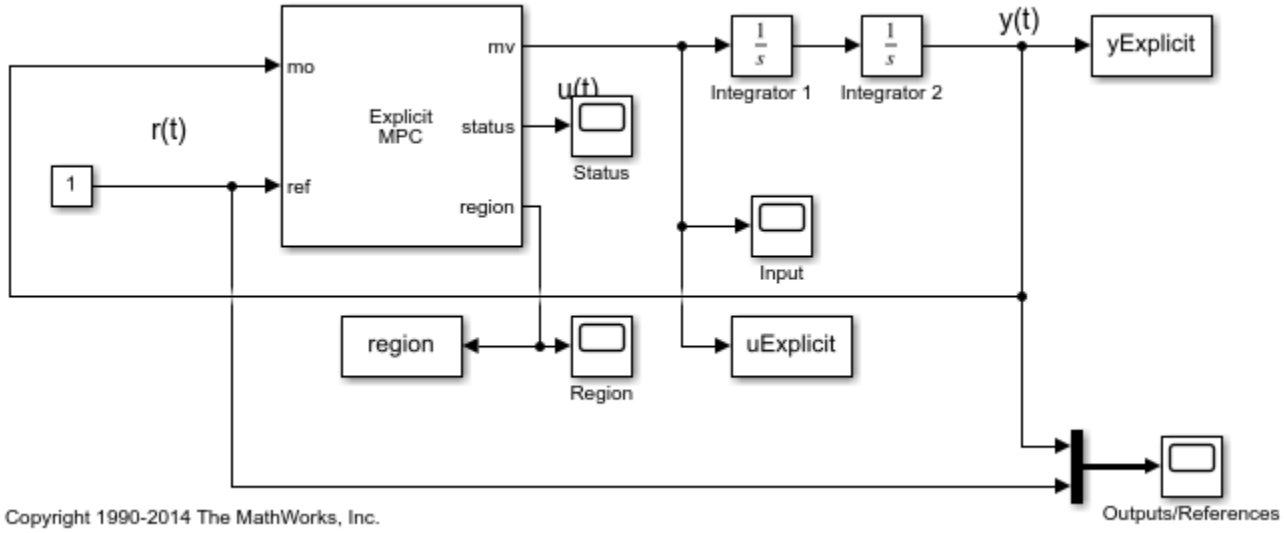
Copyright 1990-2014 The MathWorks, Inc.

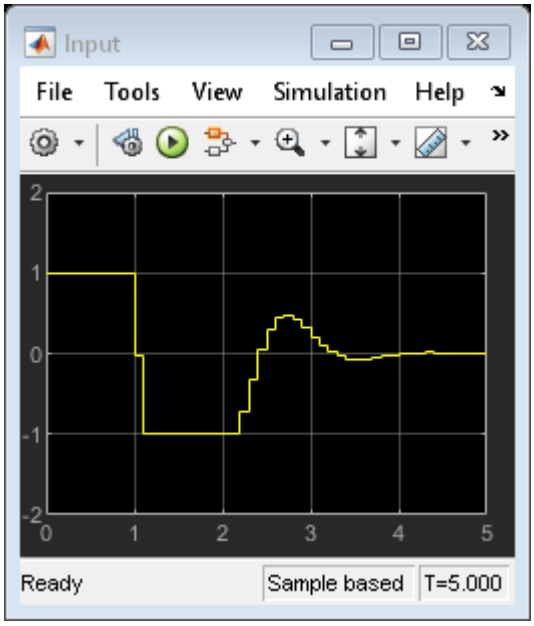


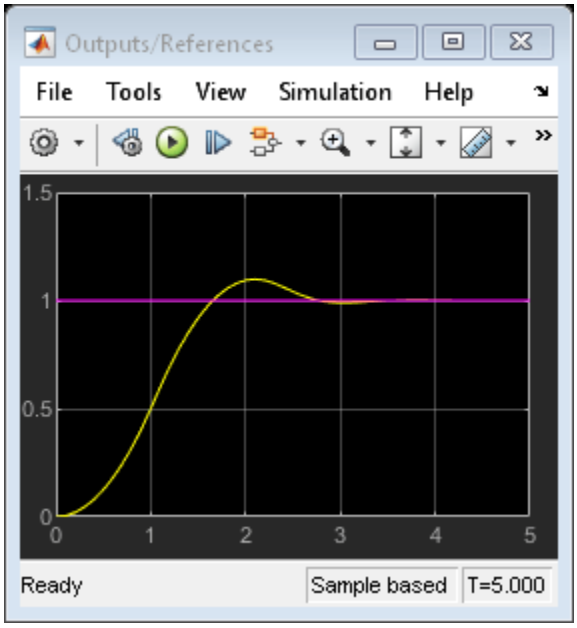


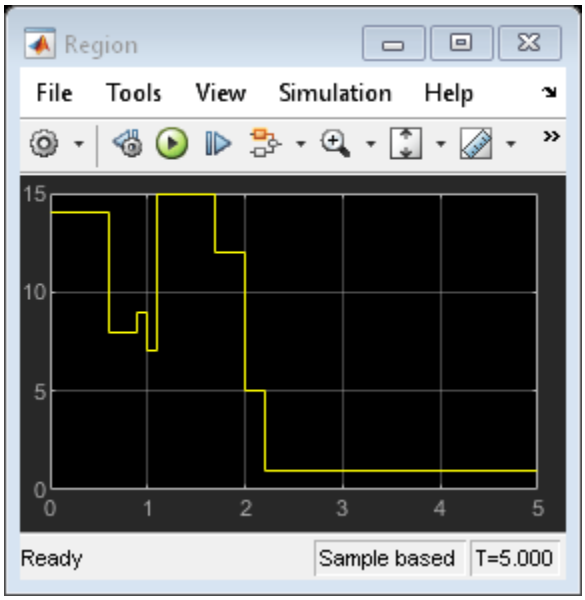
Simulate the explicit MPC controller in Simulink. The Explicit MPC Controller block is configured to use `mpcobjExplicitSimplified` as its controller.

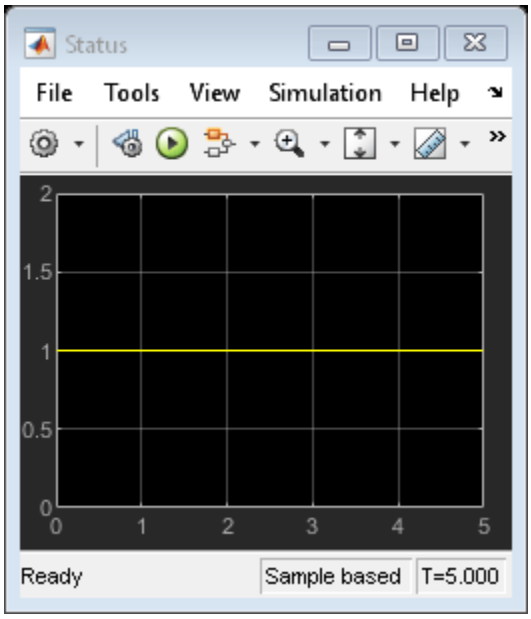
```
mdlExplicit = 'empc_doubleint';
open_system(mdlExplicit)
sim(mdlExplicit)
```




Input
File Tools View Simulation Help
2
1
0
-1
-2
0 1 2 3 4 5
Ready
Sample based
T=5.000


Outputs/References
File Tools View Simulation Help
1.5
1
0.5
0
0 1 2 3 4 5
Ready
Sample based
T=5.000





The closed-loop responses are identical.

```
fprintf('\nDifference between traditional and Explicit MPC responses in Simulink is %g\n',...
    norm(uExplicit-u)+norm(yExplicit-y));
```

```
Difference between traditional and Explicit MPC responses in Simulink is 1.64411e-13
```

Close both simulink models.

```
bdclose(mdl)
bdclose(mdlExplicit)
```

See Also

More About

- “Explicit MPC” on page 6-2
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-17
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24

Explicit MPC Control of an Aircraft with Unstable Poles

This example shows how to control an unstable aircraft with saturating actuators using explicit model predictive control.

For an example that controls the same plant using a traditional MPC controller, see “MPC Control of an Aircraft with Unstable Poles” on page 1-148.

Define Aircraft Model

The linear open-loop dynamic model of the aircraft has the following state-space matrices:

```
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];
```

Create the plant, and specify the initial states as zero.

```
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 (implicit) model predictive controller that is able to achieve 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 facilitate 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 undershoot at the first prediction horizon step. Also, specify scale factors for outputs.

```
OV = struct('Min',{[-0.5;-Inf],[-100;-Inf]},'Max',{[0.5;Inf],[100;Inf]},'ScaleFactor',{1,200});
```

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 Traditional MPC Controller

Create an MPC controller with the specified plant model, sample time, and horizons.

```
Ts = 0.05;          % Sample 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 controller. To generate an explicit MPC controller from a traditional MPC controller, you must specify the range for each controller state, reference signal, manipulated variable and measured disturbance. Doing so ensures that the multi-parametric quadratic programming problem is solved in the parameter space defined by these ranges.

Obtain a range structure for initialization

To obtain a range structure where you can specify the range for each parameter, use the `generateExplicitRange` command.

```
range = generateExplicitRange(mpcobj);
```

```
-->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 measured
```

Specify Ranges for Controller States

MPC controller states include states from the 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 an explicit MPC controller 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 an explicit MPC controller 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 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 computatio
Type 'mpcobjExplicit.PiecewiseAffineSolution' for regions and gain in each solution.
```

To join pairs of regions whose corresponding gains are the same and whose union is a convex set, use the `simplify` command with the `'exact'` method. This practice can reduce the memory footprint of the explicit MPC controller without sacrificing 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-parametric QP
Type 'mpcobjExplicitSimplified.PiecewiseAffineSolution' for regions and gain in each solution.
```

The number of piecewise affine regions 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

To obtain a parameter structure where you can specify which 2-D section to plot, use the `generatePlotParameters` function.

```
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 values within their respective ranges.

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;
```

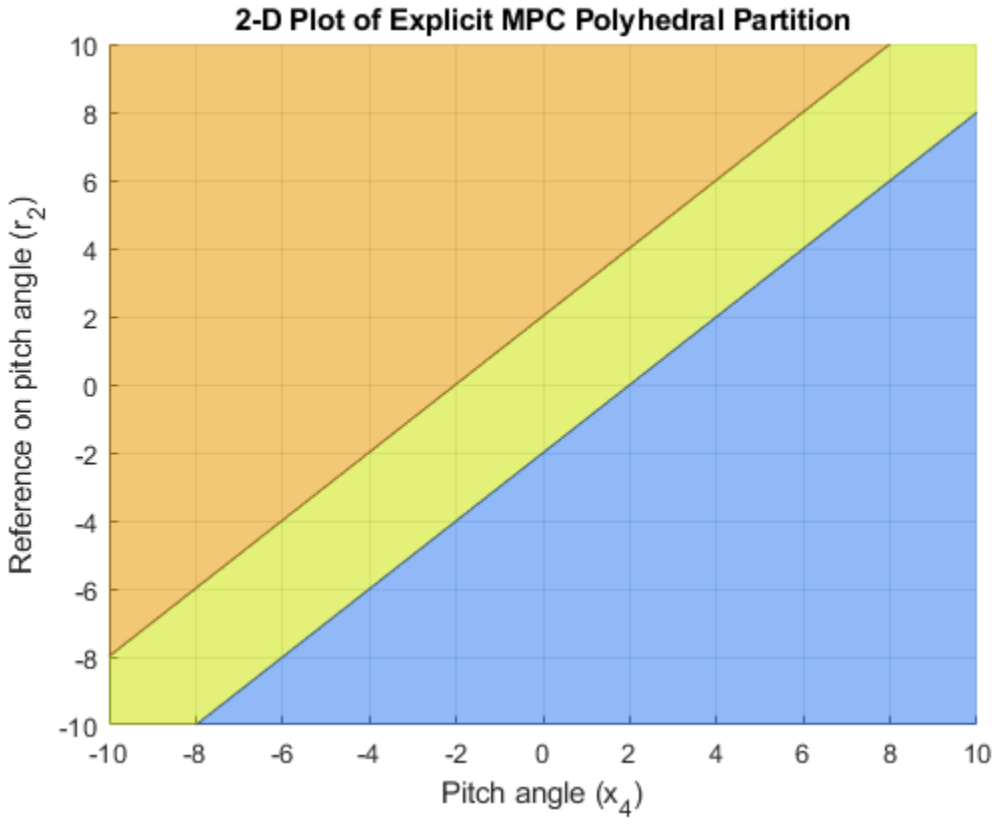
Fix manipulated variables.

```
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(mpcobjExplicitSimplified,params);
axis([-10 10 -10 10])
grid
xlabel('Pitch angle (x_4)')
ylabel('Reference on pitch angle (r_2)')
```



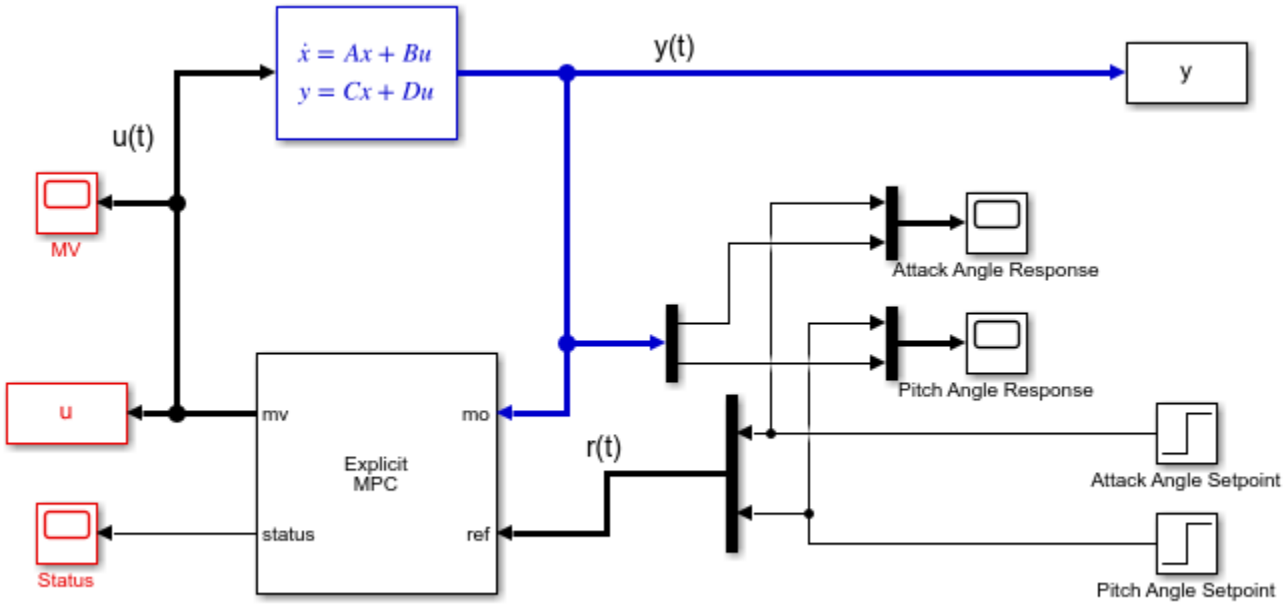
Simulate Using Simulink®

To run this example, Simulink® is required.

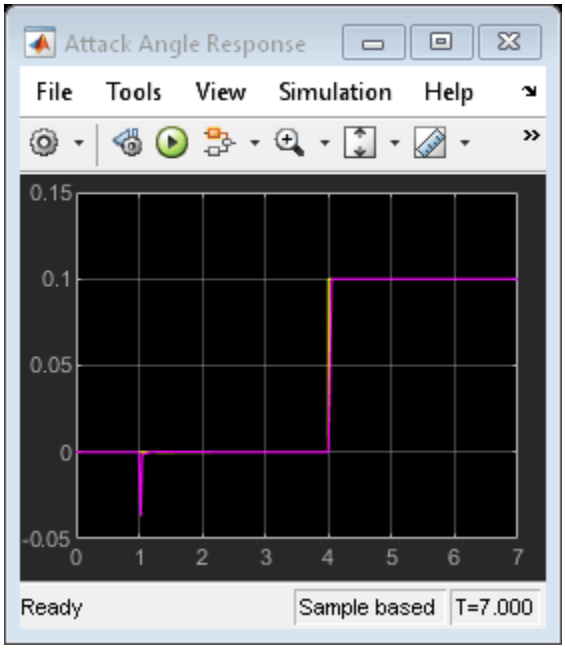
```
if ~mpccheckt oolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

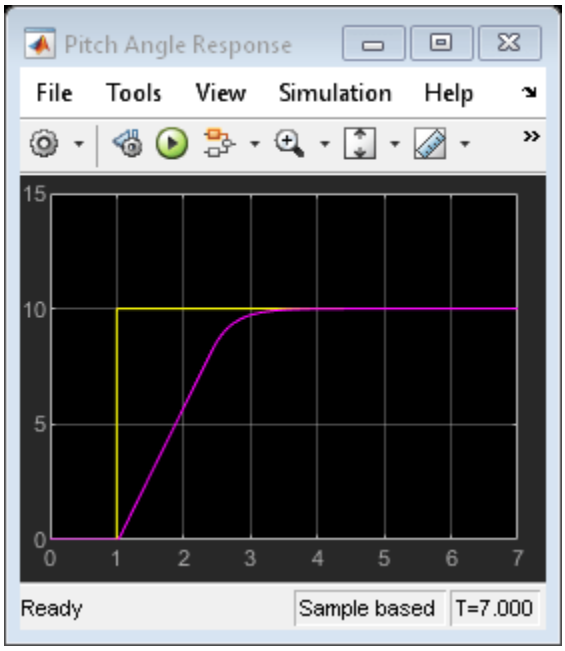
Simulate closed-loop control of the linear plant model in Simulink. To do so, for the MPC Controller block, set the **Explicit MPC Controller** property to `mpcobjExplicitSimplified`.

```
mdl = 'empc_aircraft';
open_system(mdl)
sim(mdl)
```



Copyright 1990-2014 The MathWorks, Inc.





The closed-loop response is identical to the traditional MPC controller designed in “MPC Control of an Aircraft with Unstable Poles” on page 1-148.

References

[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.

```
bdclose(mdl)
```

See Also

More About

- “Explicit MPC” on page 6-2
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-7
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-24

Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output

This example shows how to use explicit MPC to control a DC servomechanism under voltage and shaft torque constraints.

For a similar example that uses traditional implicit MPC, see “DC Servomotor with Constraint on Unmeasured Output” on page 1-10.

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;
```

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);
```

Specify 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);
```

Torque output 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;-Inf]},...
            'Max',{Inf, [tau;tau;tau;Inf]},...
            'ScaleFactor',{2*pi, 2*tau});
```

Specify Tuning 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]);
```

Create MPC controller

Create an MPC controller with sample time `Ts`, prediction horizon `p`, and control horizon `m`.

```
Ts = 0.1;
p = 10;
m = 2;
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 implicit MPC controller. To generate an explicit MPC controller from an implicit

MPC controller, 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.

Create 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 each measured
```

Specify ranges for controller states

MPC controller states include states from the 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, to collect state range data, running multiple open-loop plant simulations with typical reference and disturbance signals is recommended.

```
range.State.Min(:) = -1000;
range.State.Max(:) = 1000;
```

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. 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];
```

If manipulated variables are constrained, the ranges used to generate the explicit MPC controller must be at least as large as these limits.

```
range.ManipulatedVariable.Min = MV.Min - 1;
range.ManipulatedVariable.Max = MV.Max + 1;
```

Create an explicit MPC controller with the specified ranges.

```
mpcobjExplicit = generateExplicitMPC(mpcobj,range)
```

```
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 computation
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. To do so, first create a parameter structure where you can specify which 2-D section to plot.

```
params = generatePlotParameters(mpcobjExplicit);
```

In this example, you plot the first state variable against the second state variable. All the other parameters must be fixed at values within their respective ranges.

Fix other state variables.

```
params.State.Index = [3 4];
params.State.Value = [0 0];
```

Fix reference signals.

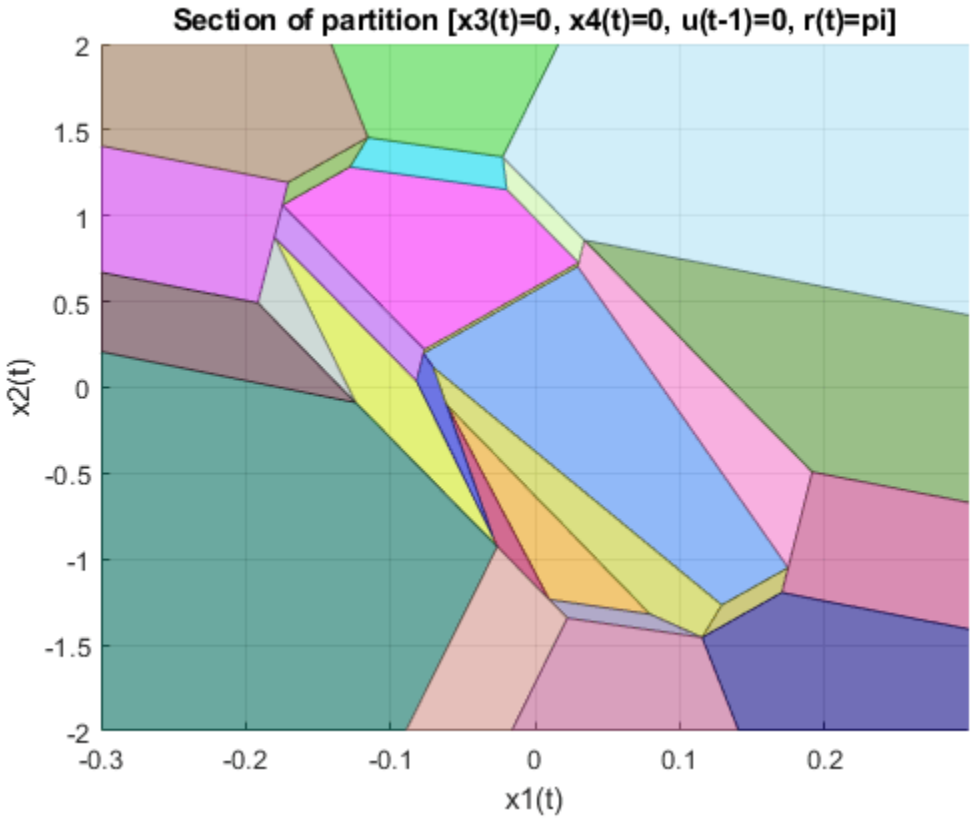
```
params.Reference.Index = [1 2];
params.Reference.Value = [pi 0];
```

Fix manipulated variables.

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Plot the specified 2-D section.

```
plotSection(mpcobjExplicit,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 Controller Using `sim` Function

Compare the closed-loop simulation results between the implicit MPC and explicit MPC controllers.

```
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 measured
```

The simulation results are identical.

```
fprintf('Difference between implicit and explicit MPC trajectories = %g\n',...
        norm(u2-u1)+norm(y2-y1));
```

```
Difference between implicit and explicit MPC trajectories = 8.68909e-12
```

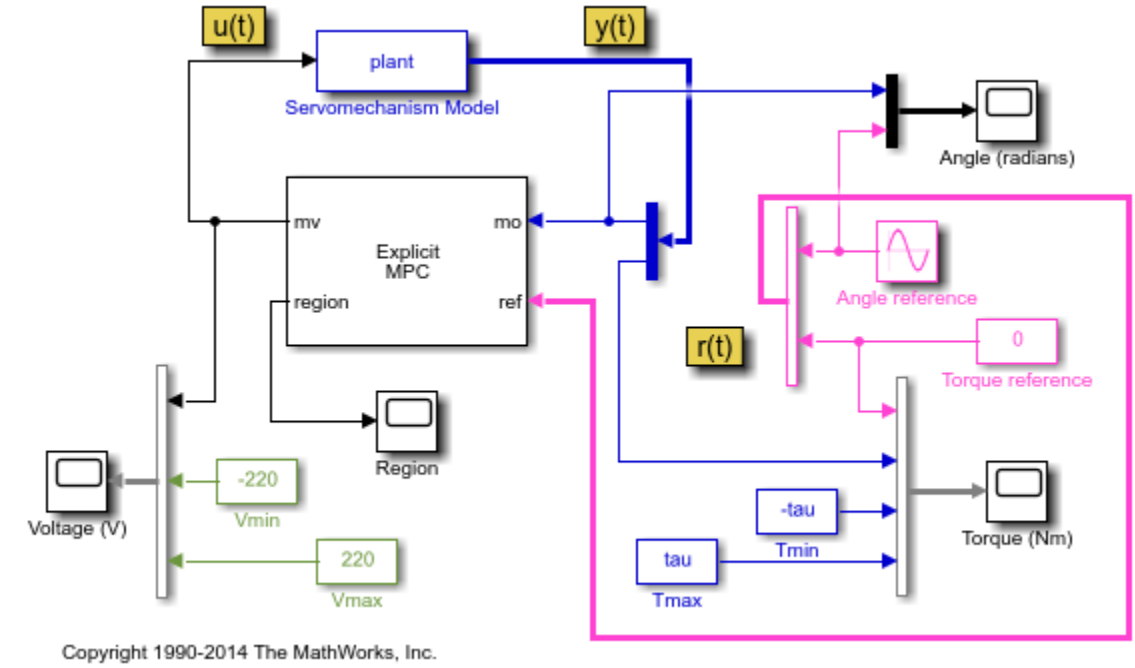
Simulate Using Simulink

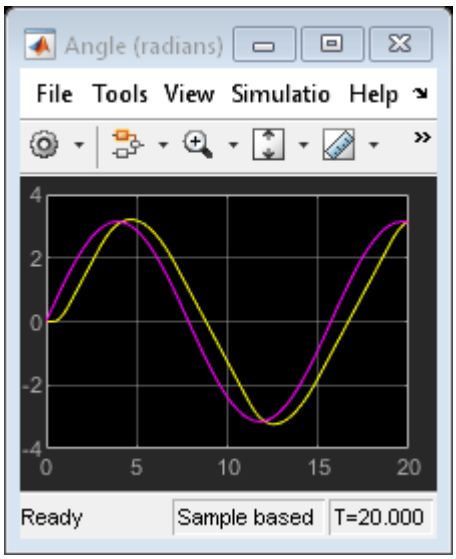
To run this example, Simulink® is required.

```
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. The Explicit MPC Controller block is configured to use `mpcobjExplicit` as its controller.

```
mdl = 'empc_motor';
open_system(mdl)
sim(mdl)
```





To specify the gain-scheduled controllers, double-click the Multiple MPC Controllers block. In the Block Parameters dialog box, specify the controllers as a cell array of controller names. Set the initial states for each controller to their respective nominal value by specifying the states as `{'[],'[]'}`.

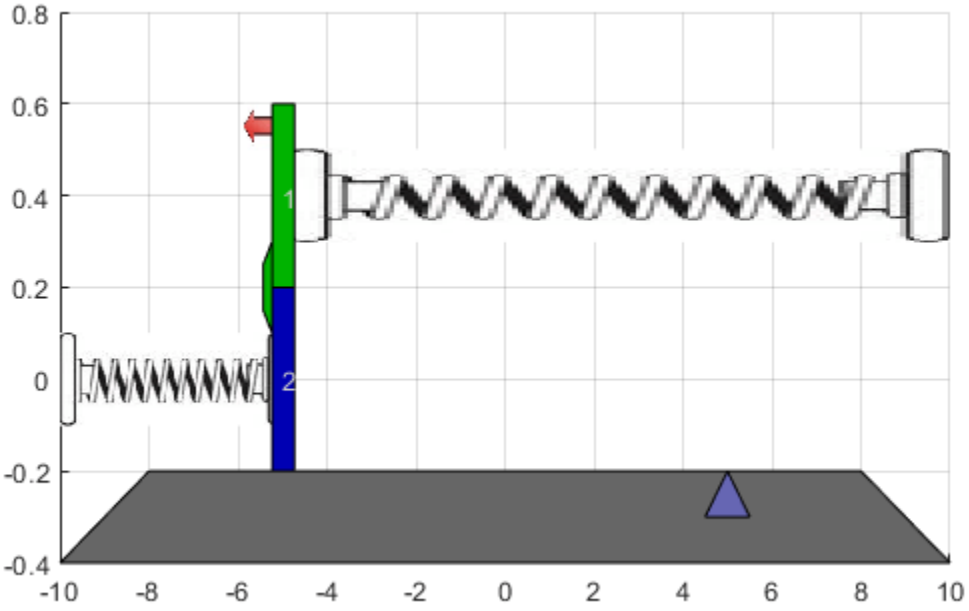
Click **OK**.

Run the simulation.

```
sim(mdl)
```

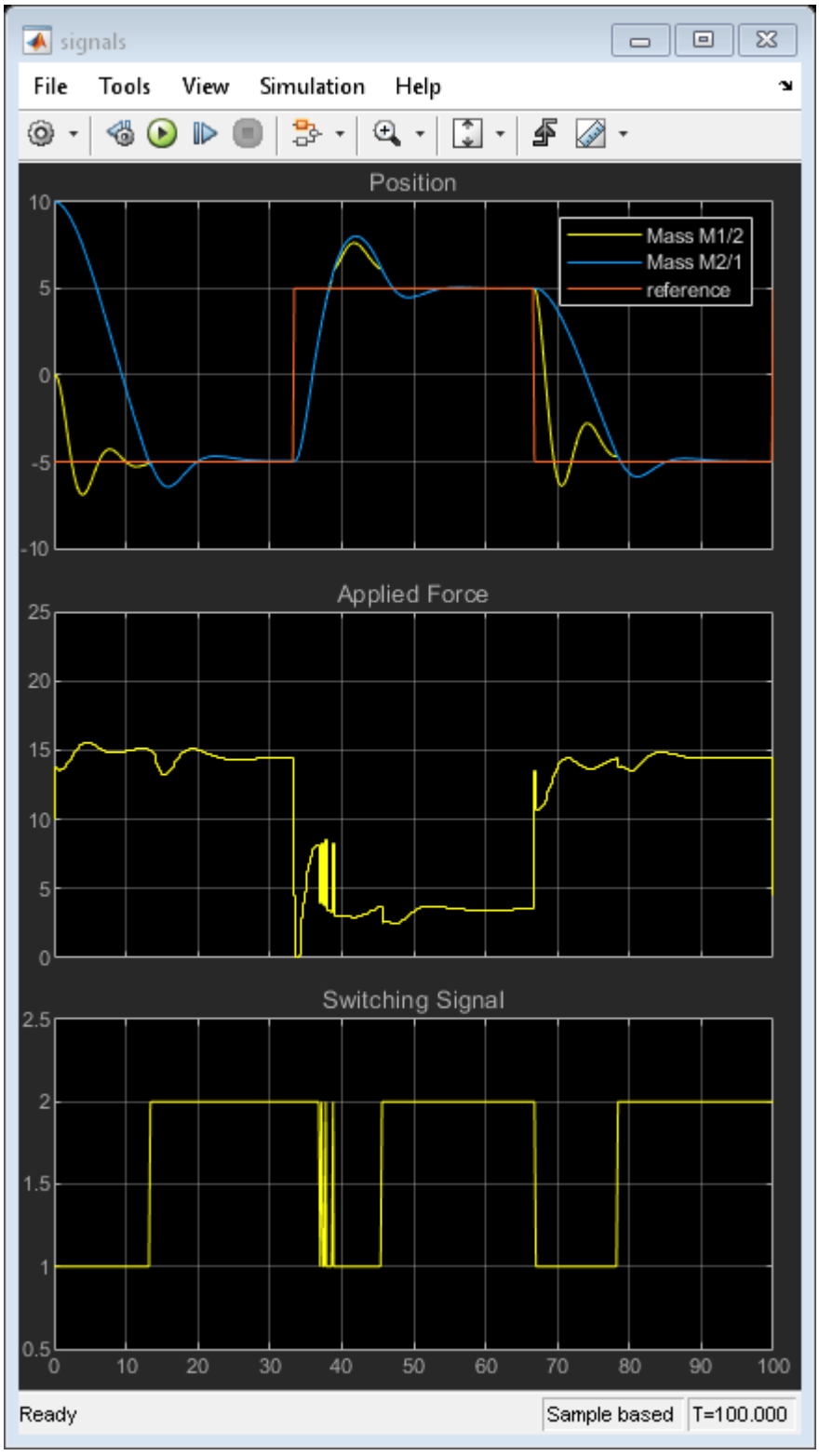
```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white noise.
```

```
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
```



To view the simulation results, open the signals scope.

```
open_system([mdl '/signals'])
```



Initially, `MPC1` moves mass `M1` to the reference setpoint. At about 13 seconds, `M2` collides with `M1`. The switching signal changes from 1 to 2, which switches control to `MPC2`.

The collision moves `M1` away from its setpoint and `MPC2` quickly returns the combined masses to the reference point.

During the subsequent reference signal transitions, when the masses separate and collide the Multiple MPC Controllers block switches between `MPC1` and `MPC2` accordingly. As a result, the combined masses settle rapidly to the reference points.

Compare with Single MPC Controller

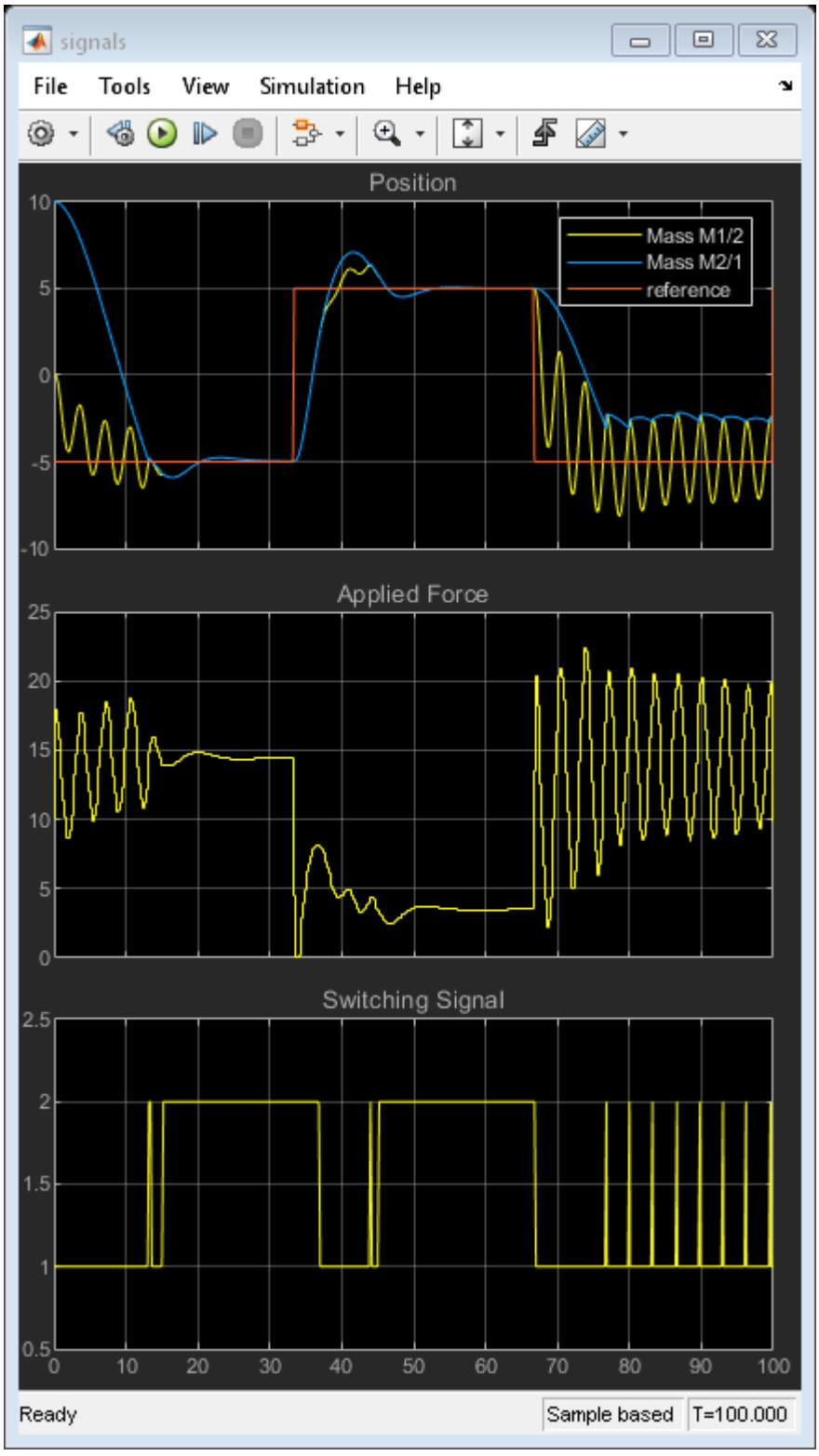
To demonstrate the benefit of using two MPC controllers for this application, simulate the system using just `MPC2`.

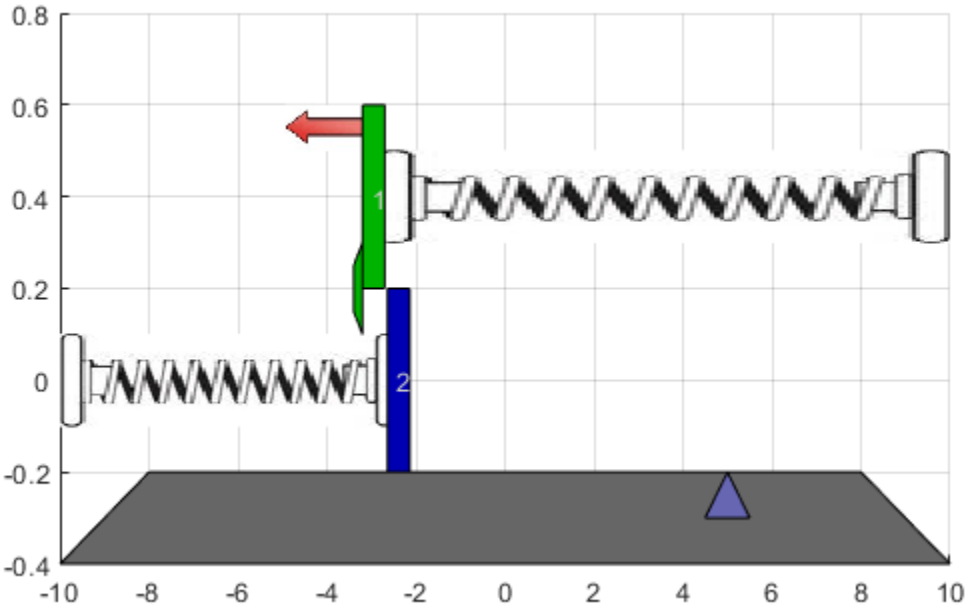
Change `MPC1` to match `MPC2`.

```
MPC1save = MPC1;
MPC1 = MPC2;
```

Run the simulation.

```
sim(mdl)
```


signals
File Tools View Simulation Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
Ready
Sample based
T=100.000



When the masses are not connected, `MPC2` applies excessive force since it expects a larger mass. This aggressive control action produces oscillatory behavior. Once the masses connect, the control performance improves, since the controller is designed for this condition.

Alternatively, changing `MPC2` to match `MPC1` results in sluggish control actions and long settling times when the masses are connected.

Set `MPC1` back to its original configuration.

```
MPC1 = MPC1save;
```

Create Explicit MPC Controllers

To reduce online computational effort, you can create an explicit MPC controller for each operating condition, and implement gain-scheduled explicit MPC control using the Multiple Explicit MPC Controllers block. For more information on explicit MPC controllers, see “Explicit MPC” on page 6-2.

To create an explicit MPC controller, first define the operating ranges for the controller states, input signals, and reference signals.

Create an explicit MPC range object using the corresponding traditional controller, `MPC1`.

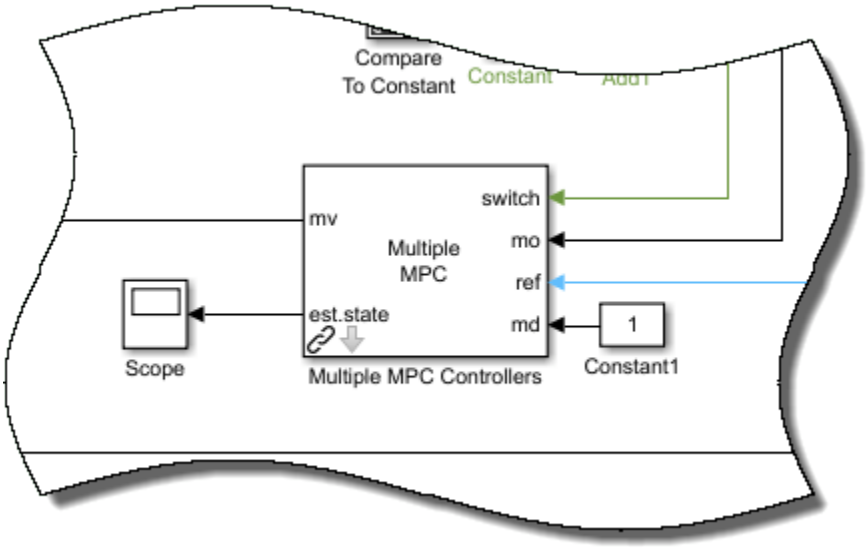
```
range = generateExplicitRange(MPC1);
```

Specify the ranges for the controller states. Both `MPC1` and `MPC2` contain states for:

- The position and velocity of mass `M1`.
- An integrator from the default output disturbance model.

When possible, use your knowledge of the plant to define the state ranges. For example, the first state corresponds to the position of `M1`, which has a range between -10 and 10.

Setting the range of a state variable can be difficult when the state does not correspond to a physical parameter, such as for the output disturbance model state. In that case, collect range information using simulations with typical reference and disturbance signals. For this system, you can activate the optional `est.state` outport of the Multiple MPC Controllers block, and view the estimated states using a scope. When simulating the controller responses, use a reference signal that covers the expected operating range.



Define the state ranges for the explicit MPC controllers based on the ranges of the estimated states.

```
range.State.Min(:) = [-10;-8;-3];
range.State.Max(:) = [10;8;3];
```

Define the range for the reference signal. Select a reference range that is smaller than the `M1` position range.

```
range.Reference.Min = -8;
range.Reference.Max = 8;
```

Specify the manipulated variable range using the defined MV constraints.

```
range.ManipulatedVariable.Min = 0;
range.ManipulatedVariable.Max = 30;
```

Define the range for the measured disturbance signal. Since the measured disturbance is constant, specify a small range around the constant value, `1`.

```
range.MeasuredDisturbance.Min = 0.9;
range.MeasuredDisturbance.Max = 1.1;
```

Create an explicit MPC controller that corresponds to `MPC1` using the specified range object.

```
expMPC1 = generateExplicitMPC(MPC1,range);
```

```
Regions found / unexplored:        4/       0
```

Create an explicit MPC controller that corresponds to `MPC2`. Since `MPC1` and `MPC2` operate over the same state and input ranges, and have the same constraints, you can use the same range object.

```
expMPC2 = generateExplicitMPC(MPC2,range);
```

```
Regions found / unexplored:        5/       0
```

In general, the explicit MPC ranges of different controllers may not match. For example, the controllers may have different constraints or state ranges. In such cases, create a separate explicit MPC range object for each controller.

Validate Explicit MPC Controllers

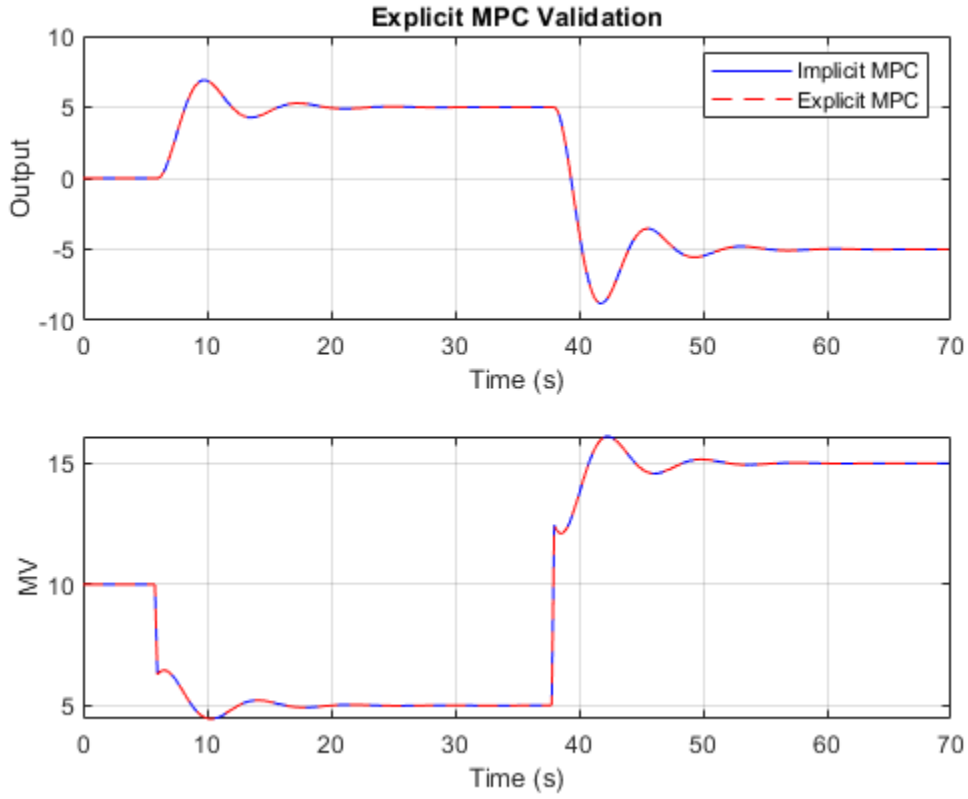
It is good practice to validate the performance of each explicit MPC controller before implementing gain-scheduled explicit MPC. For example, to compare the performance of `MPC1` and `expMPC1`, simulate the closed-loop response of each controller using `sim`.

```
r = [zeros(30,1); 5*ones(160,1); -5*ones(160,1)];
[Yimp,Timp,Uimp] = sim(MPC1,350,r,1);
[Yexp,Texp,Uexp] = sim(expMPC1,350,r,1);
```

```
-->Converting model to discrete time.
-->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 measured
```

Compare the plant output and manipulated variable sequences for the two controllers.

```
figure
subplot(2,1,1)
plot(Timp,Yimp,'b-',Texp,Yexp,'r--')
grid on
xlabel('Time (s)')
ylabel('Output')
title('Explicit MPC Validation')
legend('Implicit MPC','Explicit MPC')
subplot(2,1,2)
plot(Timp,Uimp,'b-',Texp,Uexp,'r--')
grid on
ylabel('MV')
xlabel('Time (s)')
```



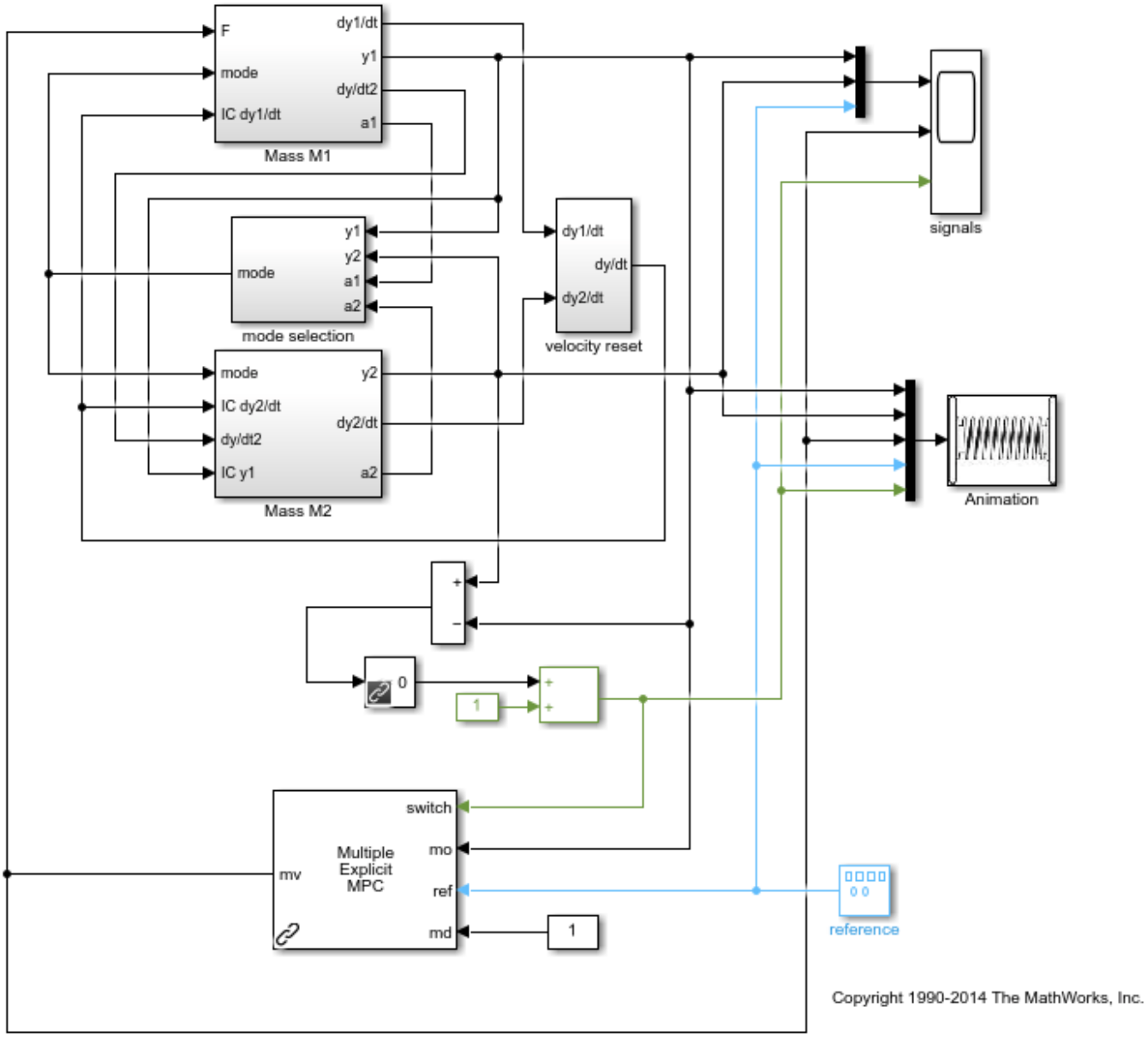
The closed-loop responses and manipulated variable sequences of the implicit and explicit controllers match. Similarly, you can validate the performance of `expMPC2` against that of `MPC2`.

If the responses of the implicit and explicit controllers do not match, adjust the explicit MPC ranges, and create a new explicit MPC controller.

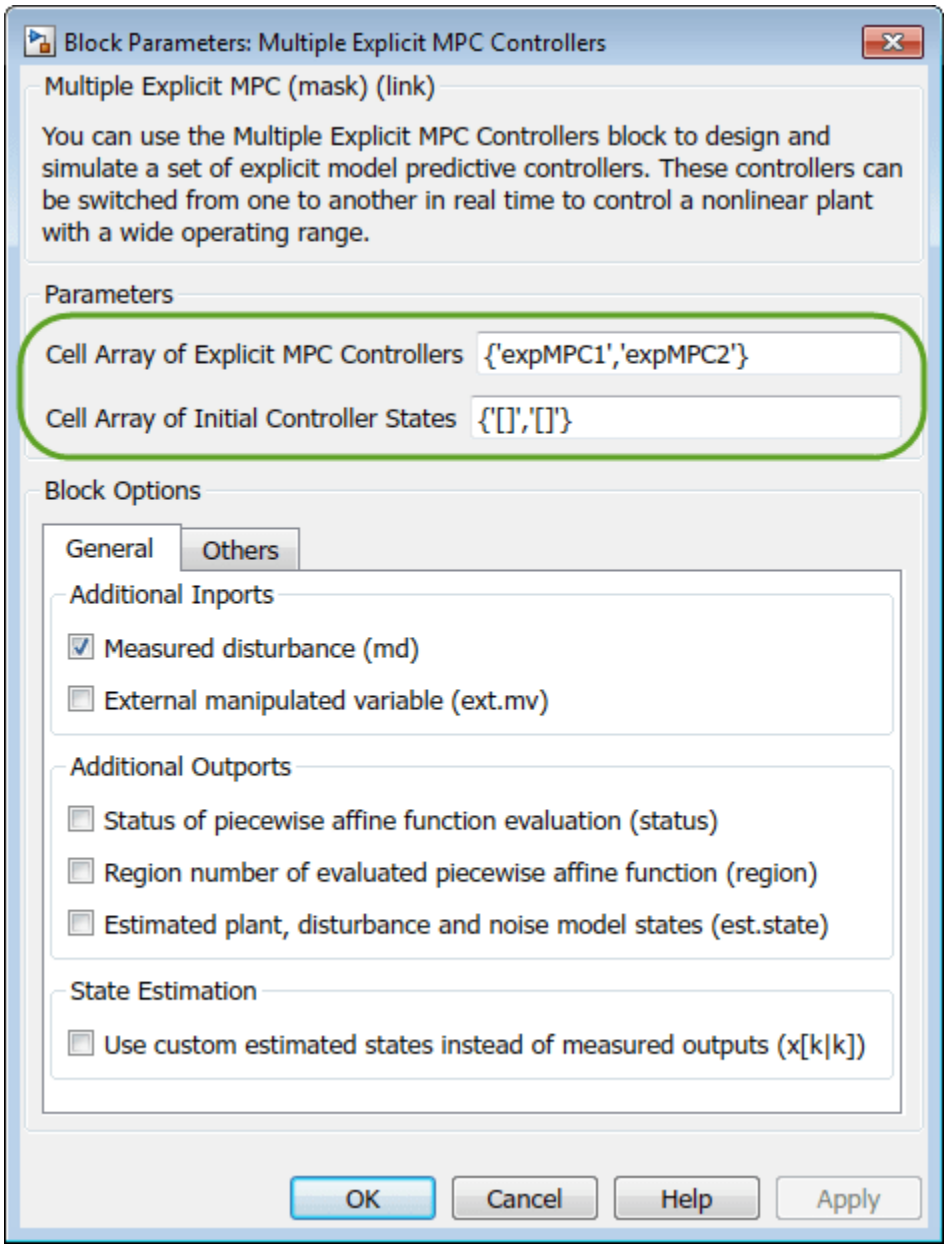
Simulate Gain-Scheduled Explicit MPC

To implement gain-scheduled explicit MPC control, replace the Multiple MPC Controllers block with the Multiple Explicit MPC Controllers block.

```
expModel = 'mpc_switching_explicit';
open_system(expModel)
```



To specify the explicit MPC controllers, double-click the Multiple Explicit MPC Controllers block. In the Block Parameters dialog box, specify the controllers as a cell array of controller names. Set the initial states for each controller to their respective nominal value by specifying the states as `{'[],'[]'}`.

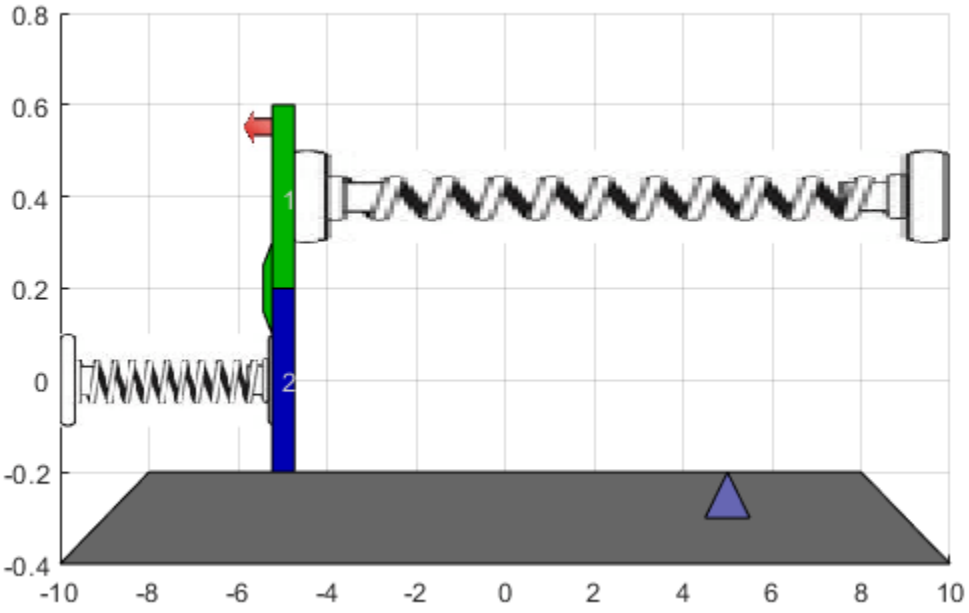


Click **OK**.

If you previously validated the your explicit MPC controllers, then substituting and configuring the Multiple Explicit MPC Controllers block should produce the same results as the Multiple MPC Controllers block.

Run the simulation.

```
sim(expModel)
```



To view the simulation results, open the signals scope.

```
open_system([expModel '/signals'])
```

The gain-scheduled explicit MPC controllers provide the same performance as the gain-scheduled implicit MPC controllers.

```
bdclose('all')
```

See Also

Multiple MPC Controllers | Multiple Explicit MPC Controllers

More About

- “Gain-Scheduled MPC” on page 8-2
- “Gain-Scheduled Implicit and Explicit MPC Control of Mass-Spring System” on page 8-42

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 a low conversion rate to high conversion rate.

Multiple MPC controllers are designed at different operating conditions and then implemented with the Multiple MPC Controllers block in Simulink. At run time, a scheduling signal is used to switch between controllers.

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 system is a jacketed non-adiabatic tank reactor described extensively in [1]. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> 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:

The outputs of the model, which are also the model states, are:

The control objective is to maintain the concentration of reagent A, at its desired setpoint, which changes over time when the reactor transitions from a low conversion rate to a high conversion rate. The coolant temperature 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 the MPC control strategy:

(1) Design several MPC controllers offline, one for each plant model. At run time, use the Multiple MPC Controller block, which switches between controllers 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 the Adaptive MPC Controller block together with a linear parameter-varying system (LPV System block). The Adaptive MPC Controller block updates the predictive model at each control interval, and the LPV System block supplies a linear plant model based on a scheduling strategy. For more details, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter-Varying System” on page 7-27. 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 the Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:

(1) Use successive linearization. For more details, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 7-7. 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. For more details, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 7-17. Use this approach when a 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™ software are required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
```

```
end
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

First, obtain a linear plant model at the initial operating condition, where `CAi` is 10 kgmol/m^3, and both `Ti` and `Tc` are 298.15 K. To generate the linear state-space system from the Simulink model, use functions such as `operspec`, `findop`, and `linearize` from Simulink Control Design.

Create operating point specification.

```
plant_mdl = 'mpc_cstr_plant';
op = operspec(plant_mdl);
```

Specify the known feed concentration at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Specify the known feed temperature at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Specify the known coolant temperature at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute the initial condition.

```
[op_point,op_report] = findop(plant_mdl,op);


 Operating point search report:
---------------------------------

opreport =


 Operating point search report for the Model mpc_cstr_plant.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>      <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</st
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</st

(1.) mpc_cstr_plant/CSTR/Integrator
     0     311.26    Inf      0       8.1176e-11      0
(2.) mpc_cstr_plant/CSTR/Integrator1
     0     8.5698    Inf      0      -6.8709e-12      0

Inputs:
----------
     <strong>Min</strong>        <strong>u</strong>        <strong>Max</strong>
    <strong>______</strong>    <strong>______</strong>    <strong>______</strong>
```

```
(1.) mpc_cstr_plant/CAi
        10         10         10
(2.) mpc_cstr_plant/Ti
    298.15    298.15    298.15
(3.) mpc_cstr_plant/Tc
    298.15    298.15    298.15

Outputs:
----------
    <strong>Min</strong>       <strong>y</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>

(1.) mpc_cstr_plant/T
    -Inf    311.26    Inf
(2.) mpc_cstr_plant/CA
    -Inf    8.5698    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 a 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

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 a specified sample time and default prediction and control horizons.

```
Ts = 0.5;
mpcobj = mpc(plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.0000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
```

```
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

Set nominal values in the controller. The nominal values for unmeasured disturbances must be zero.

```
mpcobj.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 0]);
```

Since the plant input and output signals have different orders of magnitude, specify scaling factors.

```
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 is to track a specified transition in the reactor concentration. The reactor temperature is 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.

```
Dist = ss(getindist(mpcobj));
Dist.B = eye(2)*10;
setindist(mpcobj,'model',Dist);
```

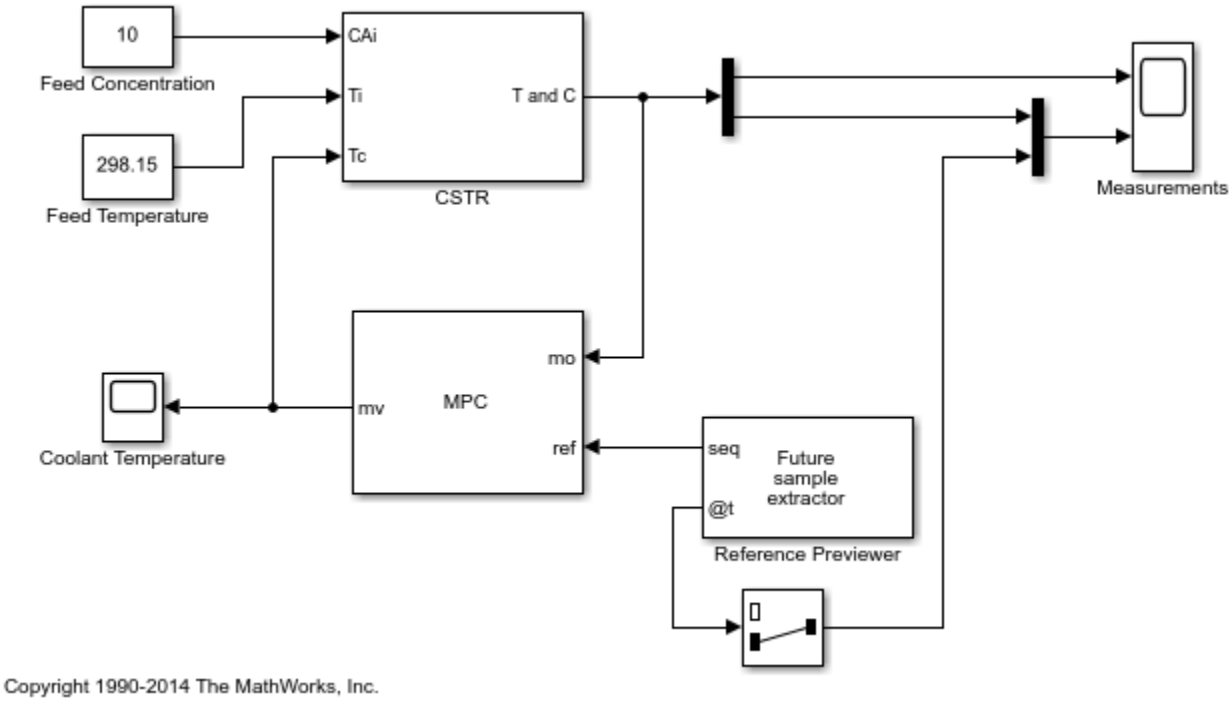
```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #1 is integrated white noise.
   Assuming unmeasured input disturbance #2 is integrated white noise.
   Assuming no disturbance added to measured output channel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

Keep all other MPC parameters at their default values.

Test the Controller With a Step Disturbance in Feed Concentration

The `mpc_cstr_single` Simulink model contains the CSTR plant and MPC controller in a feedback configuration.

```
mpc_mdl = 'mpc_cstr_single';
open_system(mpc_mdl)
```



Note that the MPC Controller block is configured to look ahead at (preview) setpoint changes in the future; that is, 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]';
```

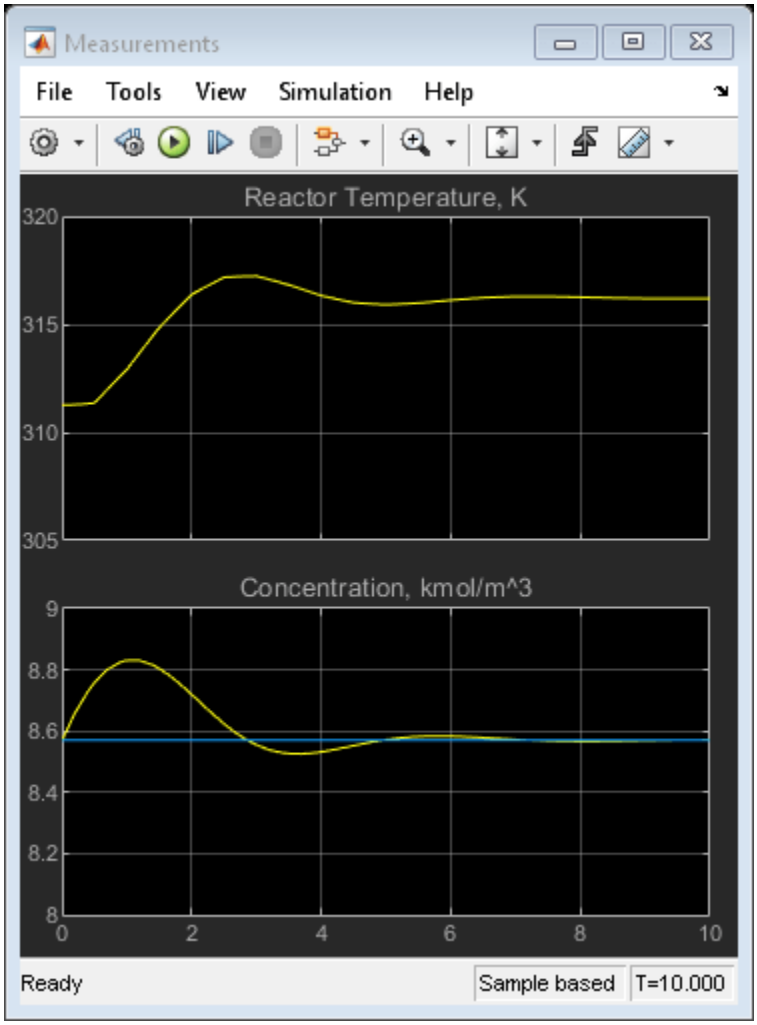
Test the response to a 5% increase in feed concentration.

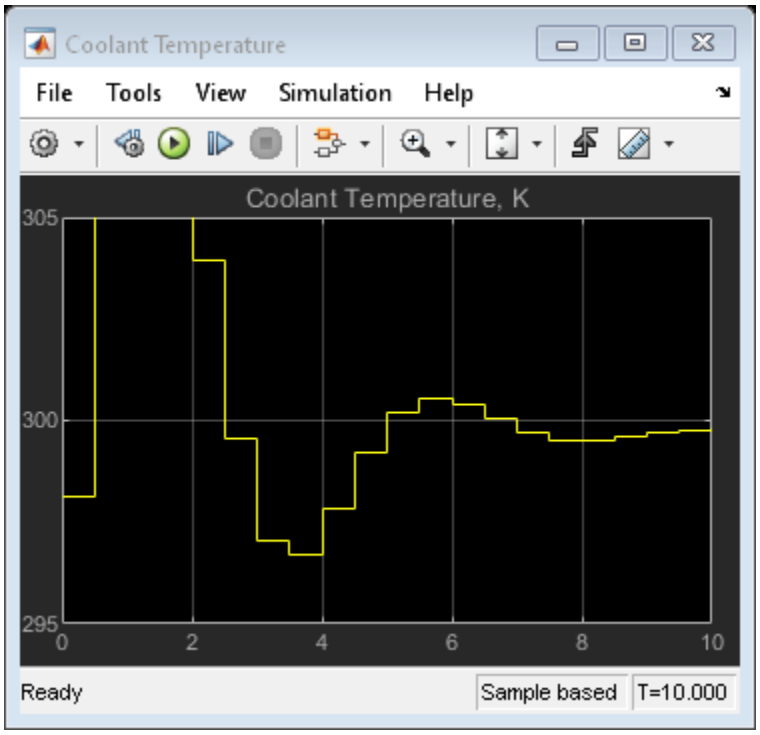
```
set_param([mpc_mdl '/Feed Concentration'],'Value','10.5');
```

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 measured
```


Measurements
File
Tools
View
Simulation
Help
Reactor Temperature, K
320
315
310
305
Concentration, kmol/m^3
9
8.8
8.6
8.4
8.2
8
0
2
4
6
8
10
Ready
Sample based
T=10.000



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/m^3.

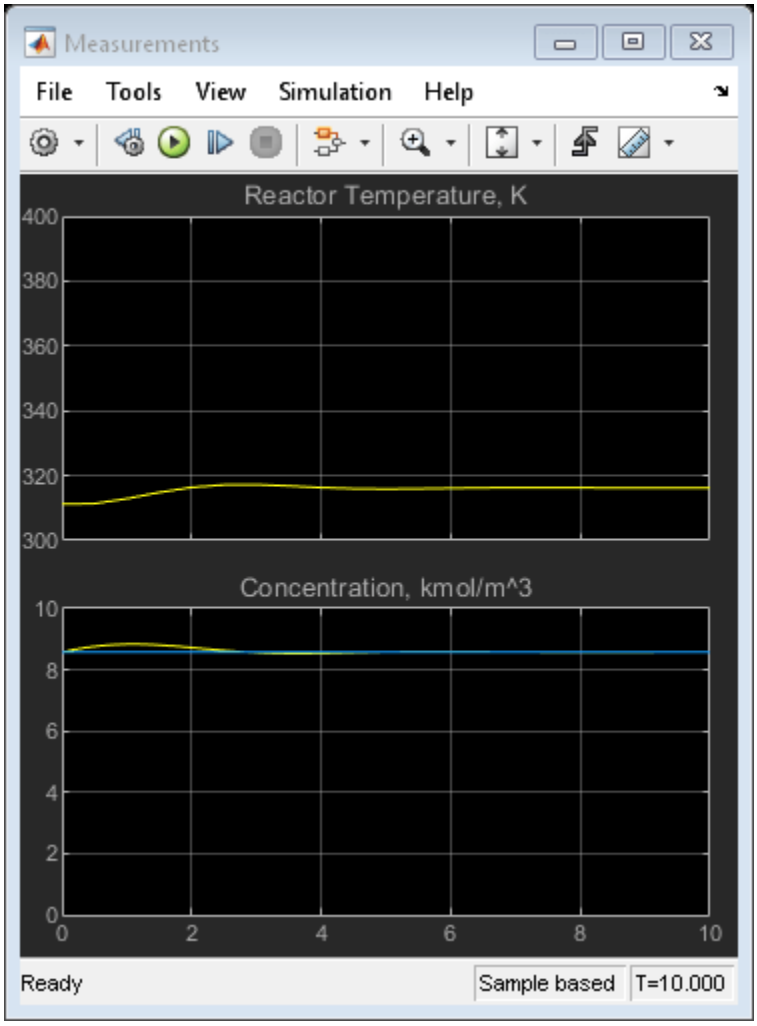
```
CSTR_Setpoints.time = [0 10 11:39]';
CSTR_Setpoints.signals.values = [y0(1)*ones(31,1),[y0(2);y0(2);(y0(2):-0.25:2)';2;2]];
```

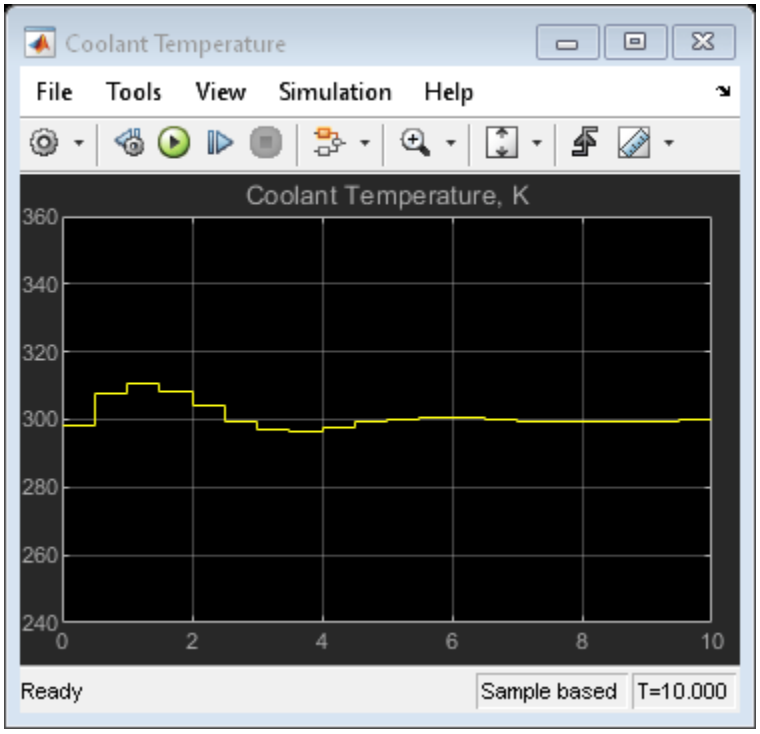
Remove the 5% increase in feed concentration used previously.

```
set_param([mpc_mdl '/Feed Concentration'],'Value','10')
```

Set plot scales and simulate the response.

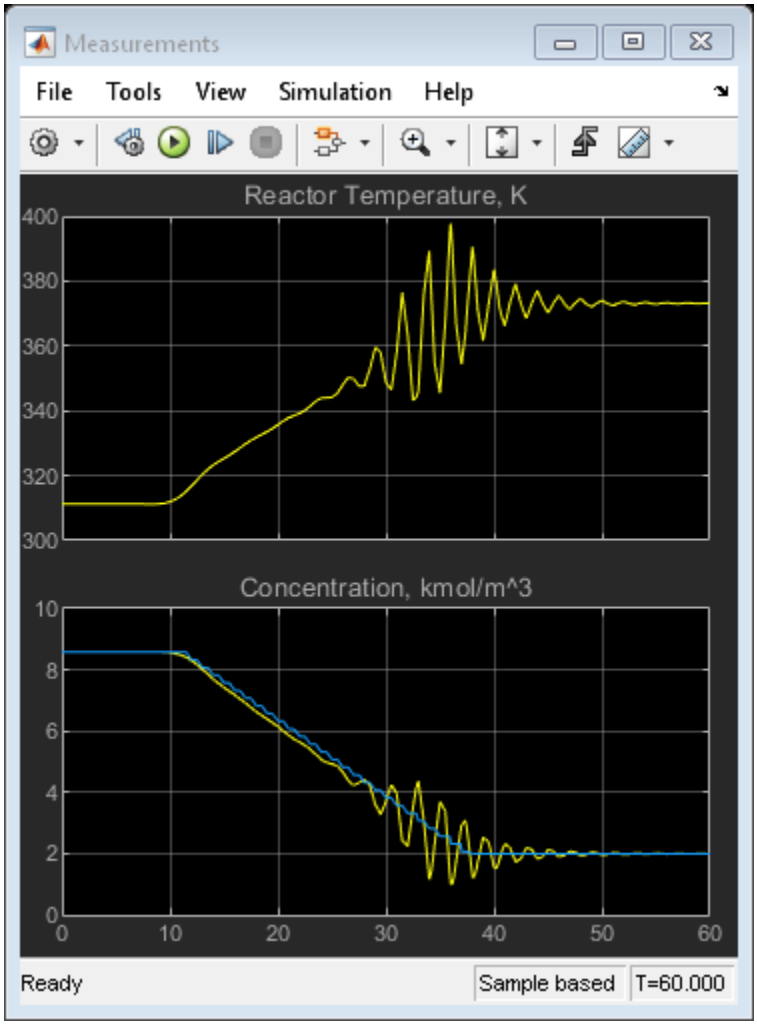
```
set_param([mpc_mdl '/Measurements'],'Ymin','300~0','Ymax','400~10')
set_param([mpc_mdl '/Coolant Temperature'],'Ymin','240','Ymax','360')
```

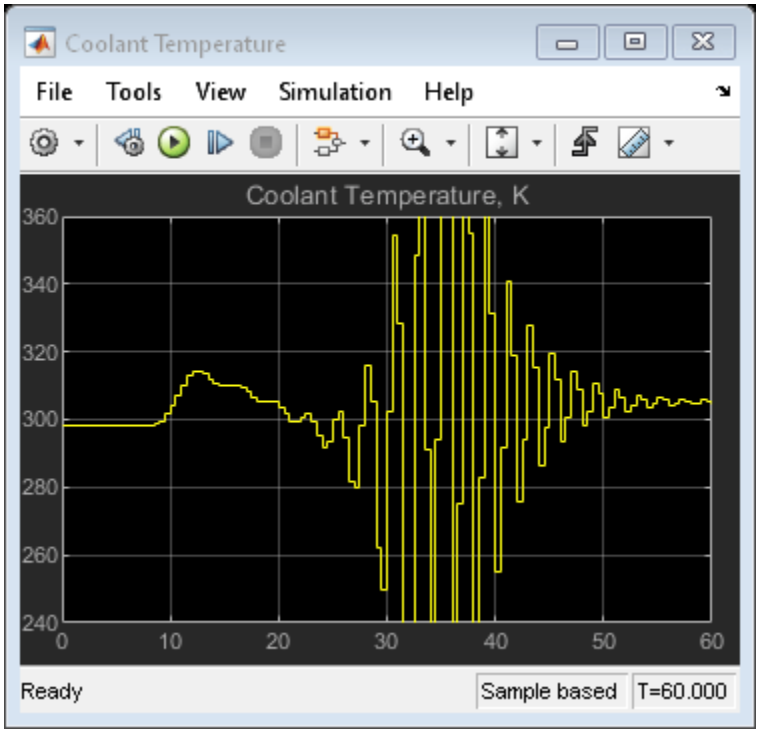

Measurements
File
Tools
View
Simulation
Help
Reactor Temperature, K
400
380
360
340
320
300
Concentration, kmol/m^3
10
8
6
4
2
0
0
2
4
6
8
10
Ready
Sample based
T=10.000



Simulate the model.

```
sim(mpc_mdl,60)
```


Measurements
File
Tools
View
Simulation
Help
Reactor Temperature, K
400
380
360
340
320
300
Concentration, kmol/m^3
10
8
6
4
2
0
0
10
20
30
40
50
60
Ready
Sample based
T=60.000



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 sections, you design MPC controllers for the intermediate and final transition stages.

Design MPC Controller for Intermediate Operating Condition

Create the operating point specification.

```
op = operspec(plant_mdl);
```

Specify the feed concentration.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Specify the feed temperature.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Specify the reactor concentration.

```
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:
---------------------------------

opreport =


 Operating point search report for the Model mpc_cstr_plant.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>      <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</str
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</str

(1.) mpc_cstr_plant/CSTR/Integrator
     0     339.43    Inf      0        3.416e-08      0
(2.) mpc_cstr_plant/CSTR/Integrator1
     0        5.5    Inf      0      -2.8663e-09      0

Inputs:
----------
     <strong>Min</strong>        <strong>u</strong>        <strong>Max</strong>
    <strong>______</strong>    <strong>______</strong>    <strong>______</strong>

(1.) mpc_cstr_plant/CAi
        10        10        10
(2.) mpc_cstr_plant/Ti
    298.15    298.15    298.15
(3.) mpc_cstr_plant/Tc
      -Inf    298.22       Inf

Outputs:
----------
    <strong>Min</strong>       <strong>y</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>

(1.) mpc_cstr_plant/T
    -Inf    339.43    Inf
(2.) mpc_cstr_plant/CA
     5.5       5.5    5.5
```

Obtain a 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 the MPC controller.

```
mpcobj_intermediate = mpc(plant_intermediate,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

Set nominal values, scaling factors, and weights in the controller.

```
mpcobj_intermediate.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 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];
Dist = ss(getindist(mpcobj_intermediate));
Dist.B = eye(2)*10;
setindist(mpcobj_intermediate,'model',Dist);
```

```
-->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 measured
```

Design MPC Controller for Final Operating Condition

Create the operating point specification.

```
op = operspec(plant_mdl);
```

Specify the feed concentration.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Specify the feed temperature.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Specify the reactor concentration.

```
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:
---------------------------------

opreport =


 Operating point search report for the Model mpc_cstr_plant.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>      <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</str
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</str

(1.) mpc_cstr_plant/CSTR/Integrator
     0     373.13    Inf      0       5.5692e-11      0
(2.) mpc_cstr_plant/CSTR/Integrator1
     0          2    Inf      0      -4.5972e-12      0

Inputs:
----------
     <strong>Min</strong>        <strong>u</strong>        <strong>Max</strong>
    <strong>______</strong>    <strong>______</strong>    <strong>______</strong>

(1.) mpc_cstr_plant/CAi
        10        10        10
(2.) mpc_cstr_plant/Ti
    298.15    298.15    298.15
(3.) mpc_cstr_plant/Tc
      -Inf     305.2       Inf

Outputs:
----------
    <strong>Min</strong>       <strong>y</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>

(1.) mpc_cstr_plant/T
    -Inf    373.13    Inf
(2.) mpc_cstr_plant/CA
       2         2      2
```

Obtain a 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 the MPC controller.

```
mpcobj_final = mpc(plant_final,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

Set nominal values, scaling factors, and weights in the controller.

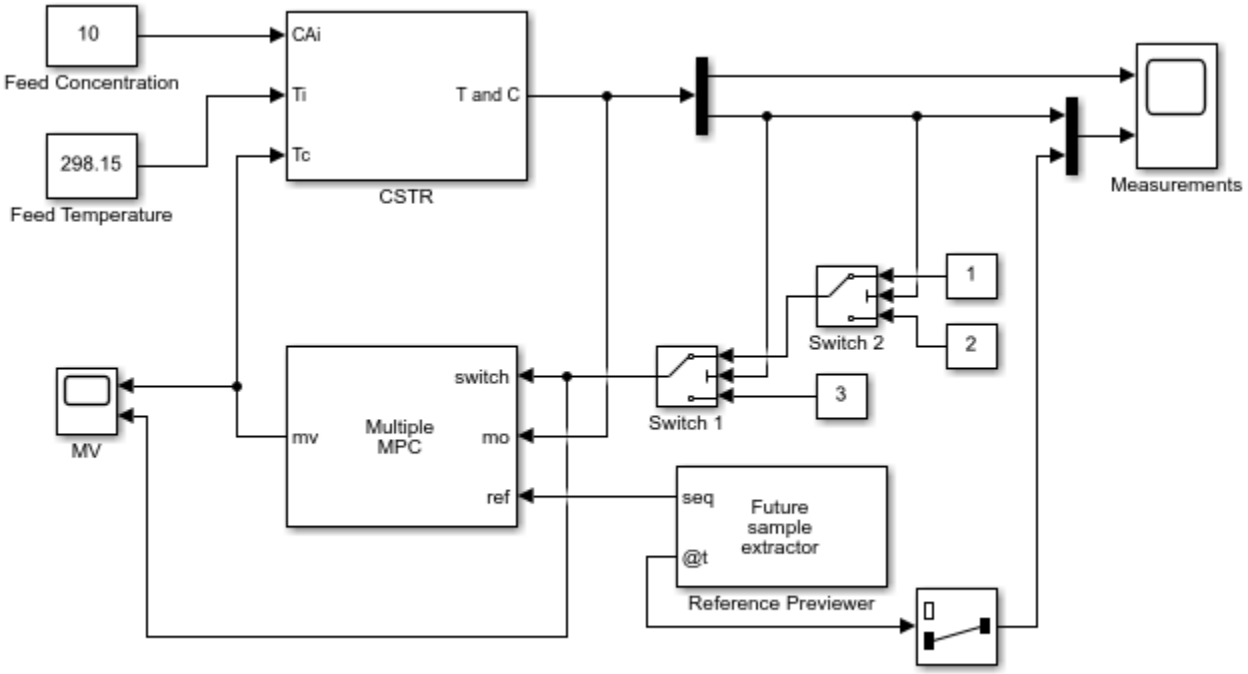
```
mpcobj_final.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 0]);
Uscale = [10;30;50];
Yscale = [50;10];
mpcobj_final.DV(1).ScaleFactor = Uscale(1);
mpcobj_final.DV(2).ScaleFactor = Uscale(2);
mpcobj_final.MV.ScaleFactor = Uscale(3);
mpcobj_final.OV(1).ScaleFactor = Yscale(1);
mpcobj_final.OV(2).ScaleFactor = Yscale(2);
mpcobj_final.Weights.OV = [0 1];
Dist = ss(getindist(mpcobj_final));
Dist.B = eye(2)*10;
setindist(mpcobj_final,'model',Dist);
```

```
-->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 measured
```

Control CSTR Plant Using Multiple MPC Controllers

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)
```



The model is configured to use the three controllers in a sequence: `mpcobj`, `mpcobj_intermediate`, and `mpcobj_final`.

```
open_system([mmpc_mdl '/Multiple MPC Controllers'])
```

Also, the two switches specify when to switch from one controller to another. The rules are:

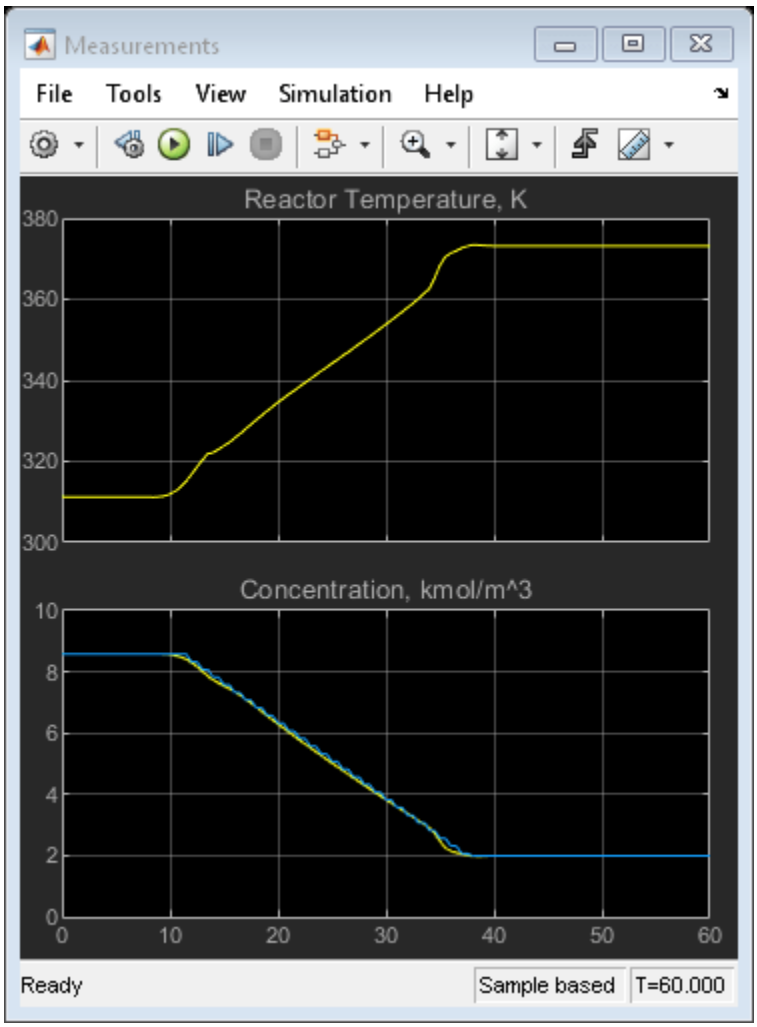
1. If CSTR concentration >= 8, use `mpcobj`.
2. If 3 <= CSTR concentration < 8, use `mpcobj_intermediate`.
3. If CSTR concentration < 3, use `mpcobj_final`.

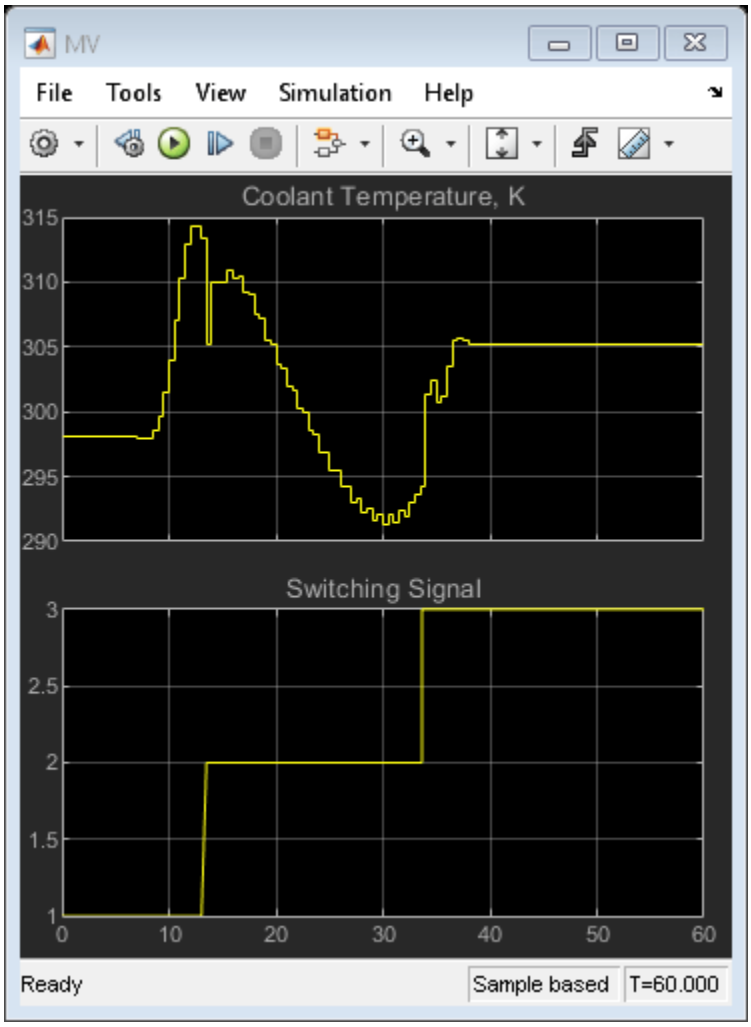
Simulate using the Multiple MPC Controllers block.

```
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 measured
-->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 measured
```





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.

References

[1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004.

```
bdclose(plant_mdl)
bdclose(mpc_mdl)
bdclose(mmpc_mdl)
```

See Also

MPC Controller | Multiple MPC Controllers

More About

- “Gain-Scheduled MPC” on page 8-2
- “Schedule Controllers at Multiple Operating Points” on page 8-4
- “Gain-Scheduled Implicit and Explicit MPC Control of Mass-Spring System” on page 8-42

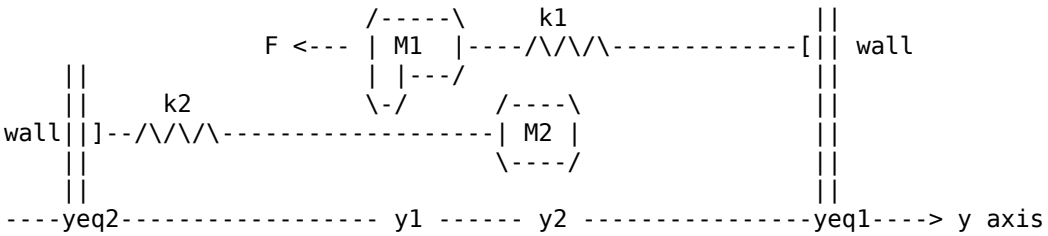
Gain-Scheduled Implicit and Explicit MPC Control of Mass-Spring System

This example shows how to use an Multiple MPC Controllers block and an Multiple Explicit 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 y1 track a given reference r.

The dynamics are twofold: when the masses are detached, `M1` moves freely. Otherwise, `M1` and `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 the position and velocity of mass `M2` are not controllable.



The model is a simplified version of the model proposed in [1].

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
```

Plant Models

The state-space plant models for this examples have the following input and output signals:

- States: Position and velocity of mass `M1`
- Manipulated variable: Pulling force `F`
- Measured disturbance: Constant value of `1` which calibrates the spring force to the correct value
- Measured output: Position of mass `M1`

Define the state-space model of `M1` when the 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.
```

Define the 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

Specify the controller sample time `Ts`, prediction horizon `p`, and control horizon `m`.

```
Ts = 0.2;
p = 20;
m = 1;
```

Design the first MPC controller for the case when mass `M1` detaches from `M2`.

```
MPC1 = mpc(sys1,Ts,p,m);
MPC1.Weights.OV = 1;

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify constraints on the manipulated variable.

```
MPC1.MV = struct('Min',0,'Max',30,'RateMin',-10,'RateMax',10);
```

Design the second MPC controller for the case when masses `M1` and `M2` are together.

```
MPC2 = mpc(sys2,Ts,p,m);
MPC2.Weights.OV = 1;

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify constraints on the manipulated variable.

```
MPC2.MV = struct('Min',0,'Max',30,'RateMin',-10,'RateMax',10);
```

Simulate Gain Scheduled MPC in Simulink®

Simulate gain-scheduled MPC control using the Multiple MPC Controllers block, which switches between `MPC1` and `MPC2`.

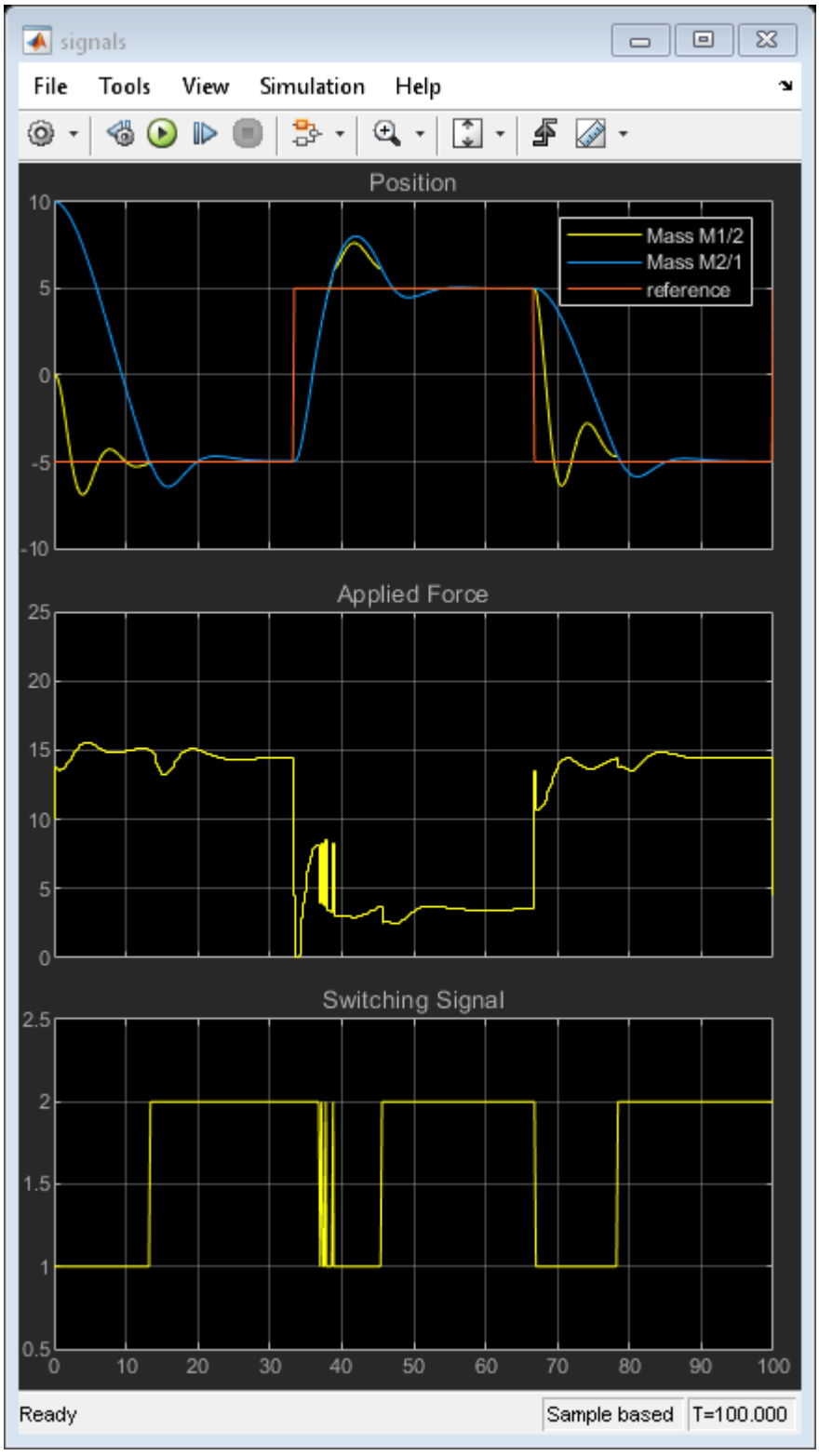
```
% To run this example, Simulink(R) is required.
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
```

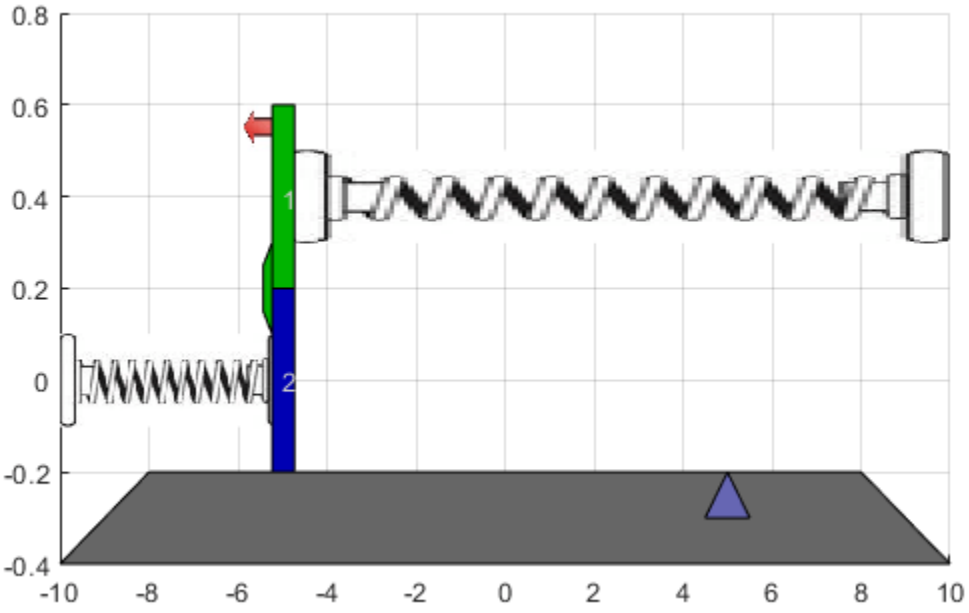
Open and configure the model.

```
y1initial = 0;      % Initial position of M1
y2initial = 10;     % Initial position of M2
mdl = 'mpc_switching';
open_system(mdl)
if exist('animationmpc_switchoff','var') && animationmpc_switchoff
    close_system([mdl '/Animation'])
    clear animationmpc_switchoff
end
```

Simulate the model.

```
open_system([mdl '/signals'])
sim(mdl)
MPC1saved = MPC1;
MPC2saved = MPC2;

-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
```


signals
File Tools View Simulation Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
Ready
Sample based
T=100.000

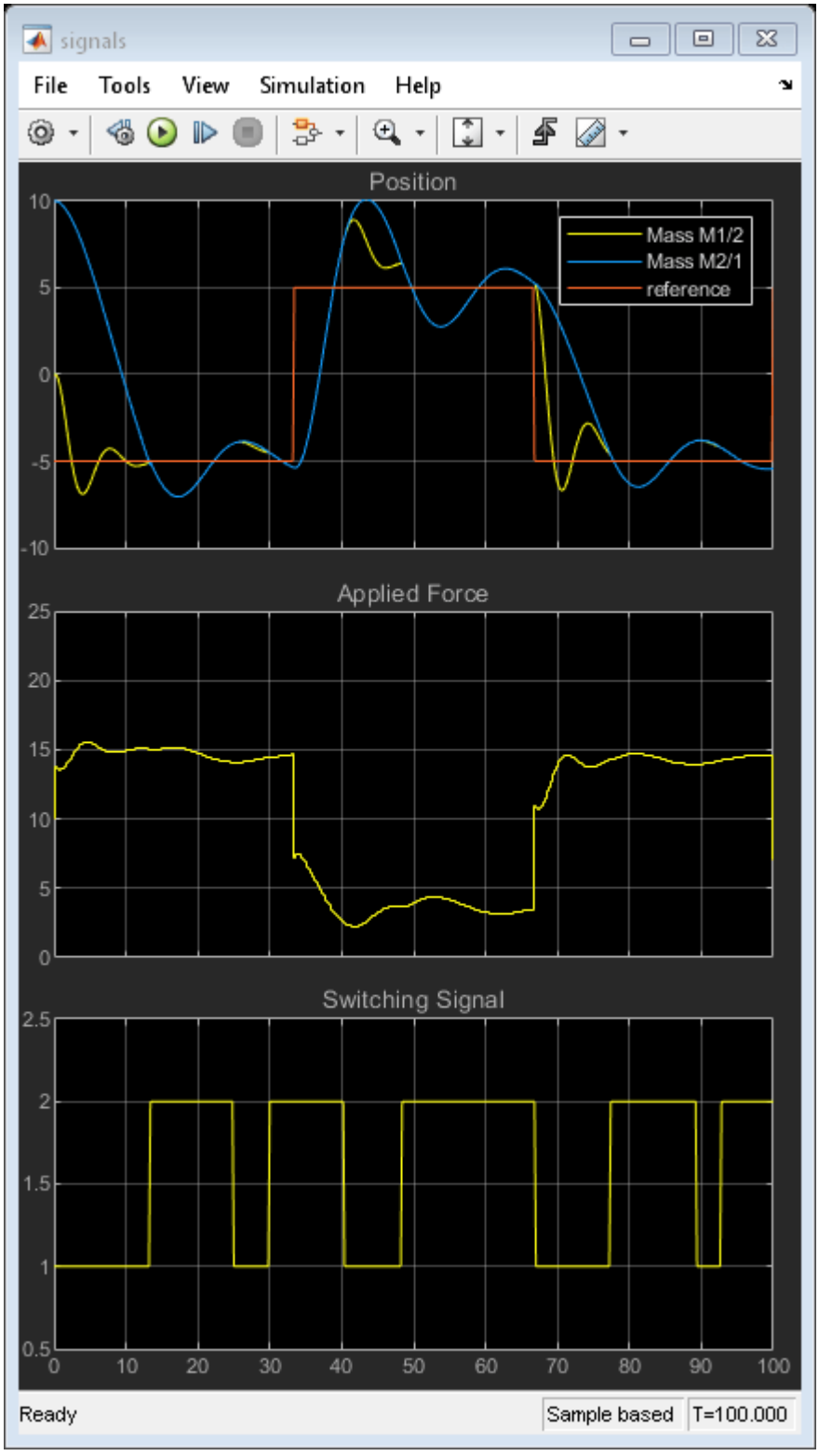


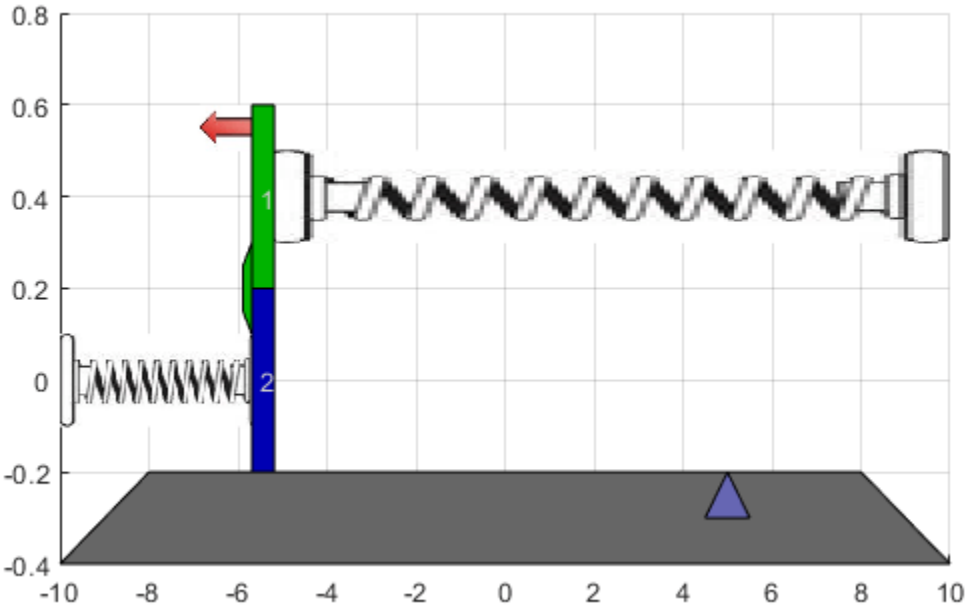
Using two controllers provides satisfactory performance under all conditions.

Repeat Simulation Using `MPC1` Only

Repeat the simulation assuming that the masses are never in contact; that is, using only controller `MPC1`.

```
MPC1 = MPC1saved;
MPC2 = MPC1saved;
sim(mdl)
```


signals
File Tools View Simulation Help
Position
Mass M1/2
Mass M2/1
reference
10
5
0
-5
-10
Applied Force
25
20
15
10
5
0
Switching Signal
2.5
2
1.5
1
0.5
0 10 20 30 40 50 60 70 80 90 100
Ready
Sample based
T=100.000

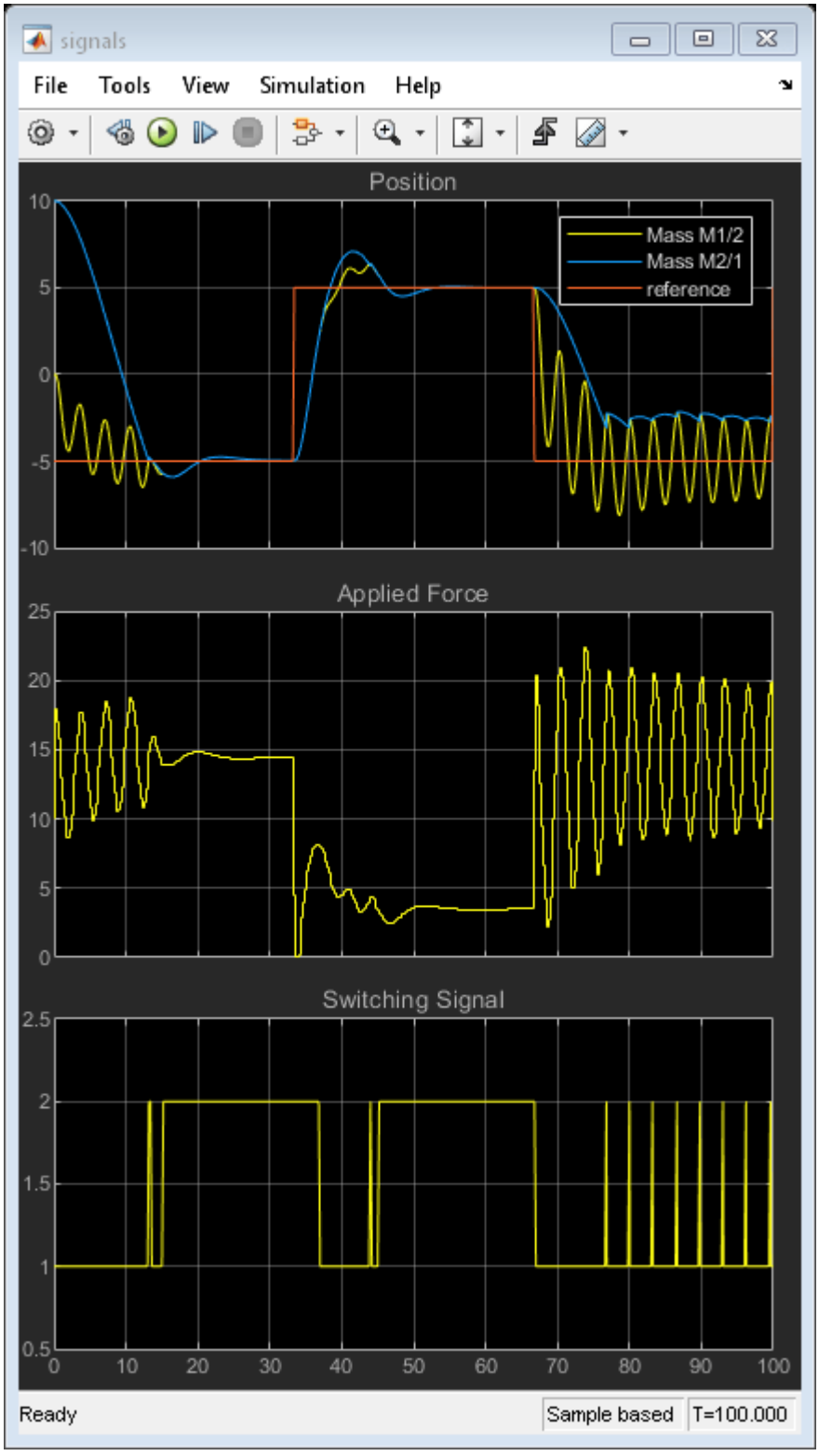


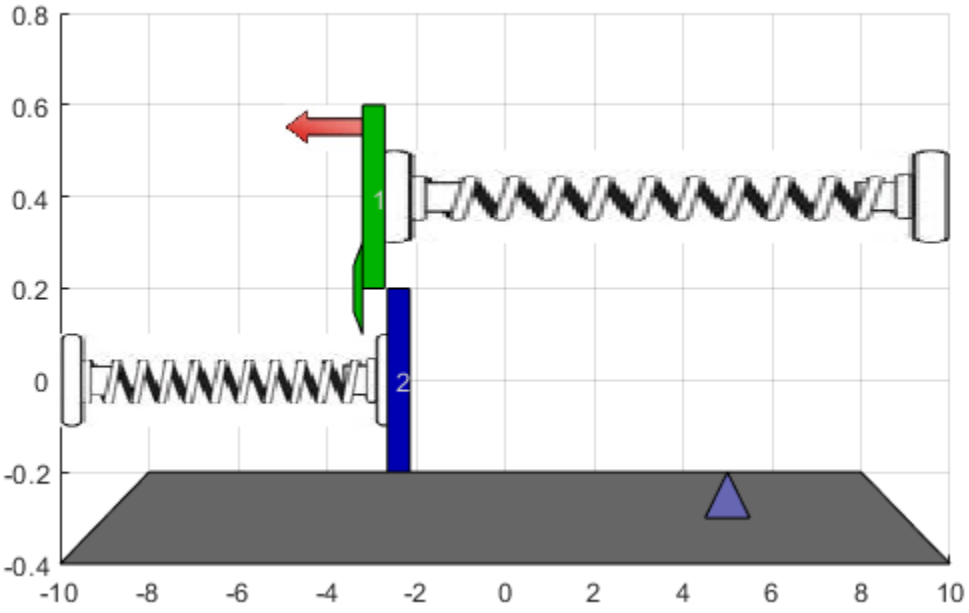
In this case, performance degrades whenever the two masses join.

Repeat Simulation Using `MPC2` Only

Repeat the simulation assuming that the masses are always in contact; that is, using only controller `MPC2`.

```
MPC1 = MPC2saved;
MPC2 = MPC2saved;
sim(mdl)
```


signals
File Tools View Simulation Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
Ready
Sample based
T=100.000



In this case, performance degrades when the masses separate, causing the controller to apply excessive force.

```
bdclose(mdl);
close(findobj('Tag','mpc_switching_demo'));
```

Design Explicit MPC Controllers

To reduce online computational effort, you can create an explicit MPC controller for each operating condition, and implement gain-scheduled explicit MPC control using the Multiple Explicit MPC Controllers block.

To create an explicit MPC controller, first define the operating ranges for the controller states, input signals, and reference signals. Create an explicit MPC range object using the corresponding traditional controller, `MPC1`.

```
range = generateExplicitRange(MPC1saved);
```

Specify the ranges for the controller states. Both `MPC1` and `MPC2` contain states for:

- The position and velocity of mass `M1`
- An integrator from the default output disturbance model

```
range.State.Min(:) = [-10;-8;-3];
range.State.Max(:) = [10;8;3];
```

When possible, use your knowledge of the plant to define the state ranges. However, it can be difficult when the state does not correspond to a physical parameter, such as for the output disturbance model state. In that case, collect range information using simulations with typical reference and disturbance signals. For this system, you can activate the optional `est.state` outport of the Multiple MPC Controllers block, and view the estimated states using a scope. When simulating the controller responses, use a reference signal that covers the expected operating range.

Define the range for the reference signal. Select a reference range that is smaller than the `M1` position range.

```
range.Reference.Min = -8;
range.Reference.Max = 8;
```

Specify the manipulated variable range using the defined MV constraints.

```
range.ManipulatedVariable.Min = 0;
range.ManipulatedVariable.Max = 30;
```

Define the range for the measured disturbance signal. Since the measured disturbance is constant, specify a small range around the constant value, `1`.

```
range.MeasuredDisturbance.Min = 0.9;
range.MeasuredDisturbance.Max = 1.1;
```

Create an explicit MPC controller that corresponds to `MPC1` using the specified range object.

```
expMPC1 = generateExplicitMPC(MPC1saved,range);
```

```
Regions found / unexplored:        4/       0
```

Create an explicit MPC controller that corresponds to `MPC2`. Since `MPC1` and `MPC2` operate over the same state and input ranges, and have the same constraints, you can use the same range object.

```
expMPC2 = generateExplicitMPC(MPC2saved,range);
```

```
Regions found / unexplored:        5/       0
```

In general, the explicit MPC ranges of different controllers may not match. For example, the controllers may have different constraints or state ranges. In such cases, create a separate explicit MPC range object for each controller.

Validate Explicit MPC Controllers

It is good practice to validate the performance of each explicit MPC controller before implementing gain-scheduled explicit MPC. For example, to compare the performance of `MPC1` and `expMPC1`, simulate the closed-loop response of each controller using `sim`.

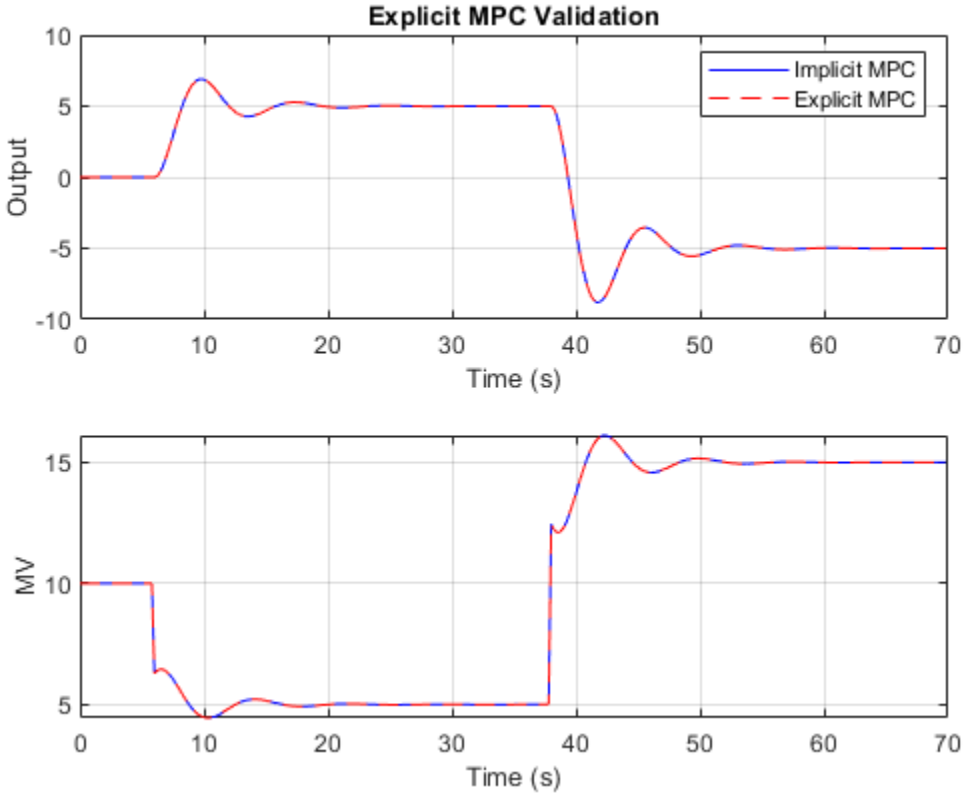
```
r = [zeros(30,1); 5*ones(160,1); -5*ones(160,1)];
[Yimp,Timp,Uimp] = sim(MPC1saved,350,r,1);
[Yexp,Texp,Uexp] = sim(expMPC1,350,r,1);
```

```
-->Converting model to discrete time.
-->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 measured
```

Compare the plant output and manipulated variable sequences for the two controllers.

```
figure
subplot(2,1,1)
plot(Timp,Yimp,'b-',Texp,Yexp,'r--')
```

```
grid on
xlabel('Time (s)')
ylabel('Output')
title('Explicit MPC Validation')
legend('Implicit MPC','Explicit MPC')
subplot(2,1,2)
plot(Timp,Uimp,'b-',Texp,Uexp,'r--')
grid on
ylabel('MV')
xlabel('Time (s)')
```



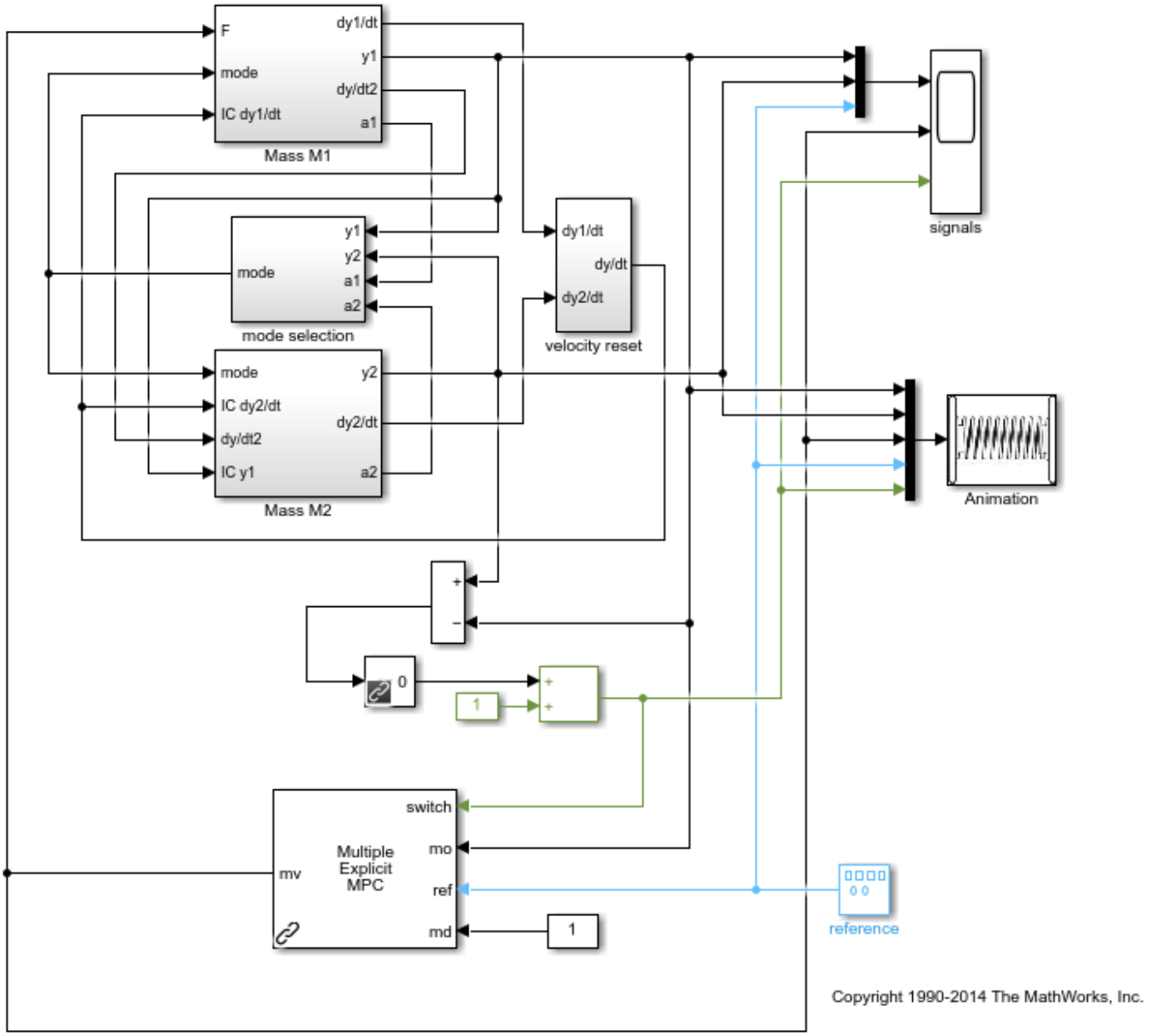
The closed-loop responses and manipulated variable sequences of the implicit and explicit controllers match. Similarly, you can validate the performance of `expMPC2` against that of `MPC2`.

If the responses of the implicit and explicit controllers do not match, adjust the explicit MPC ranges, and create a new explicit MPC controller.

Simulate Gain-Scheduled Explicit MPC

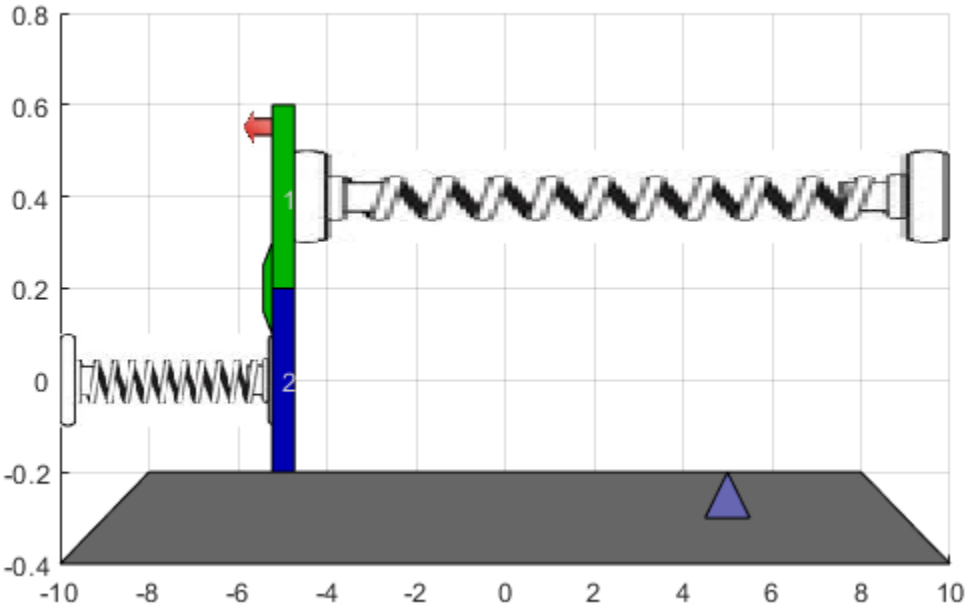
To implement gain-scheduled explicit MPC control, replace the Multiple MPC Controllers block with the Multiple Explicit MPC Controllers block.

```
expModel = 'mpc_switching_explicit';
open_system(expModel)
```



Run the simulation.

```
sim(expModel)
```



To view the simulation results, open the signals scope.

```
open_system([expModel '/signals'])
```

The gain-scheduled explicit MPC controllers provide the same performance as the gain-scheduled implicit MPC controllers.

References

[1] 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.

```
bdclose(expModel)
close(findobj('Tag','mpc_switching_demo'))
```

See Also

Multiple MPC Controllers | Multiple Explicit MPC Controllers

More About

- “Gain-Scheduled MPC” on page 8-2
- “Schedule Controllers at Multiple Operating Points” on page 8-4
- “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor” on page 8-22

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.

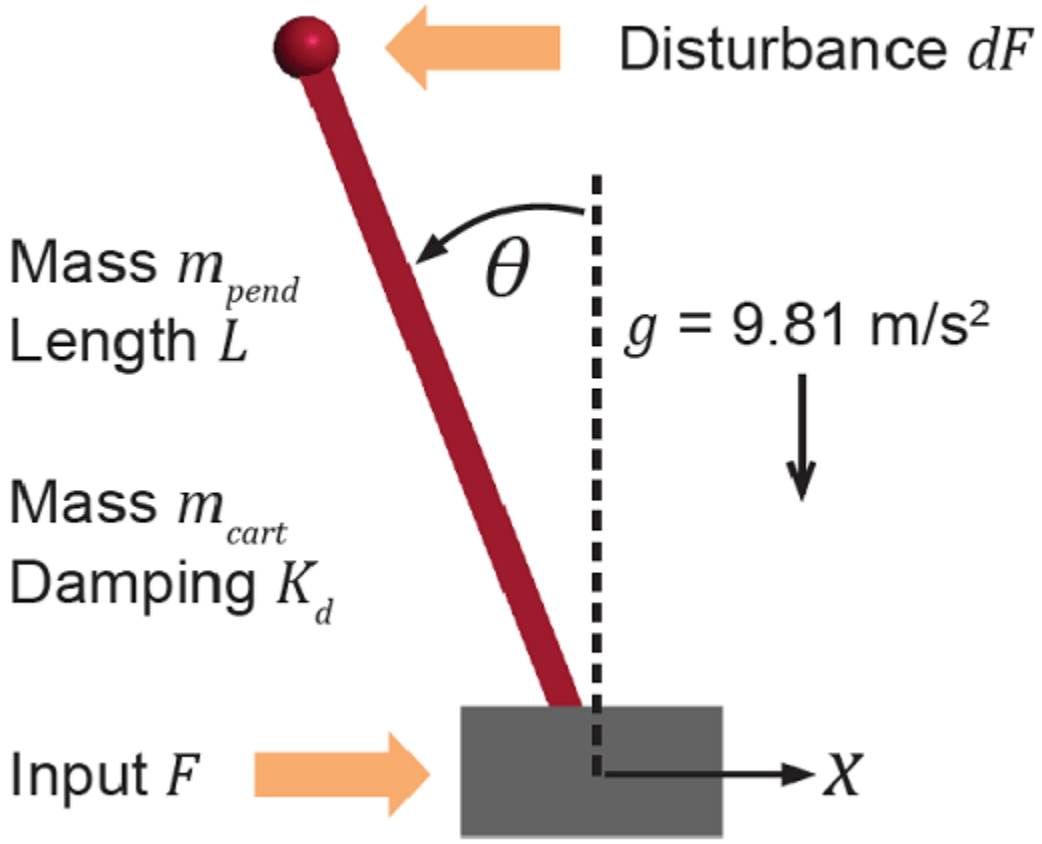
Product Requirement

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 is required to run this example.')
    return
end
```

Pendulum/Cart Assembly

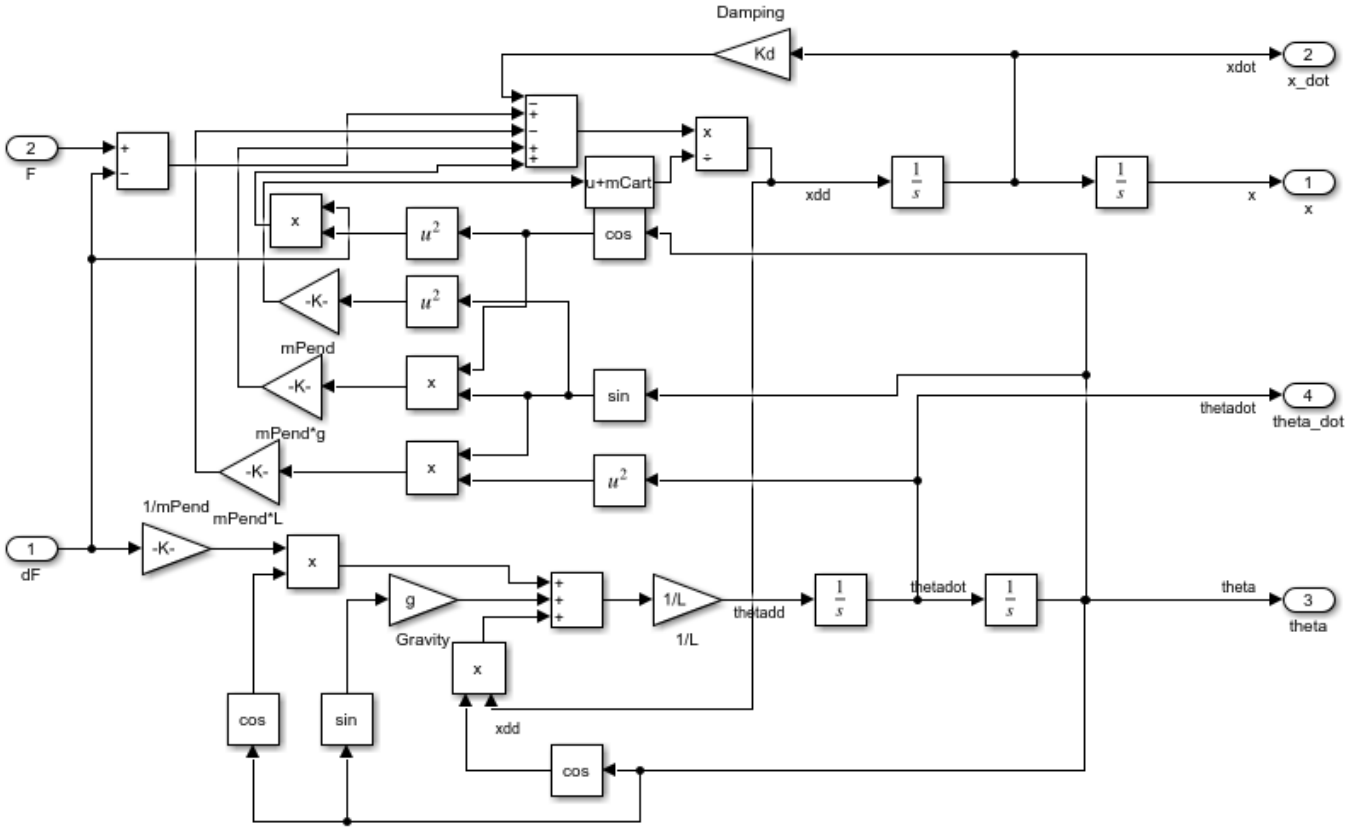
The plant for this example is the following cart/pendulum assembly, where *x* is the cart position and *theta* is the pendulum angle.



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')
```



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” on page 1-132, 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 *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.

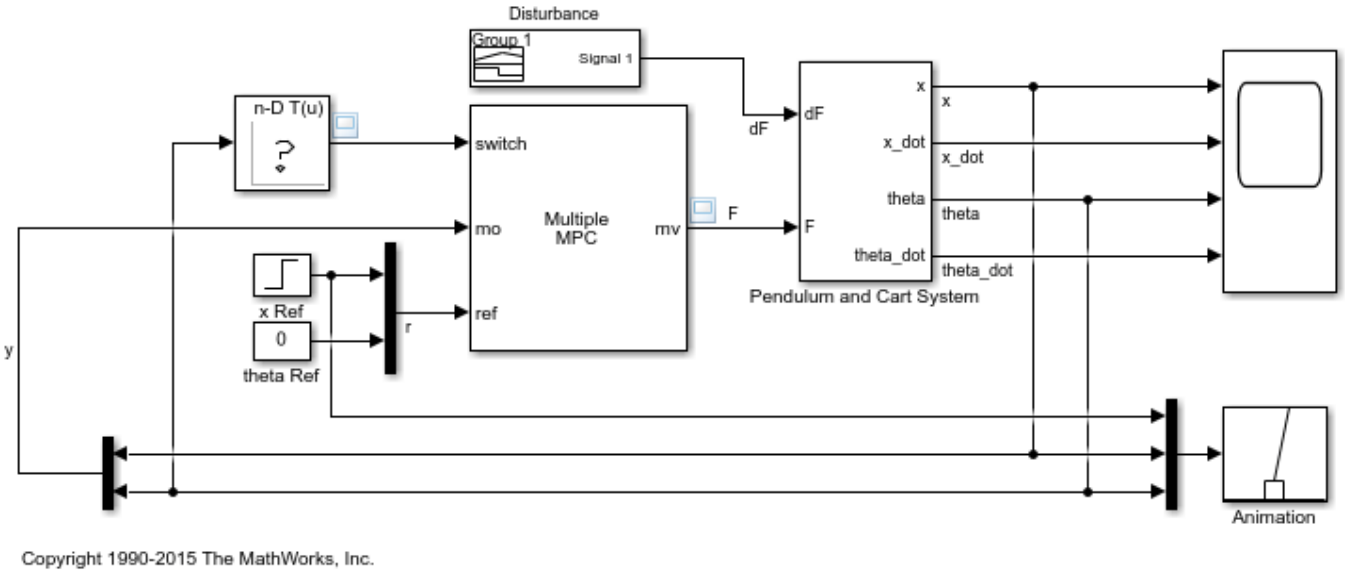
To move the cart to a new position between -15 and 15 while maintaining the same rise time, the controller needs to have more accurate models at different angles so that the controller can use them for better prediction. Gain-scheduled MPC allows you to solve a nonlinear control problem by designing 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 *theta*.
- One unmeasured disturbance: Impulse disturbance *dF*.

```
mdlMPC = 'mpc_pendcartGSMPC';
open_system(mdlMPC)
```



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 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 at 80 degrees, pointing right (*theta* = `-4*pi/9`)
- Pendulum is upright (*theta* = `0`)
- Pendulum is at 80 degrees, pointing left (*theta* = `4*pi/9`)

```
angles = [-4*pi/9 0 4*pi/9];
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 *theta*.

```
    opspec(ct).States(3).Known = true;
    opspec(ct).States(3).x = angles(ct);
```

The fourth state is angular velocity *theta_dot* (not at steady state).

```
    opspec(ct).States(4).SteadyState = false;

end
```

Compute operating points 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 *theta*.

```
    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);
```

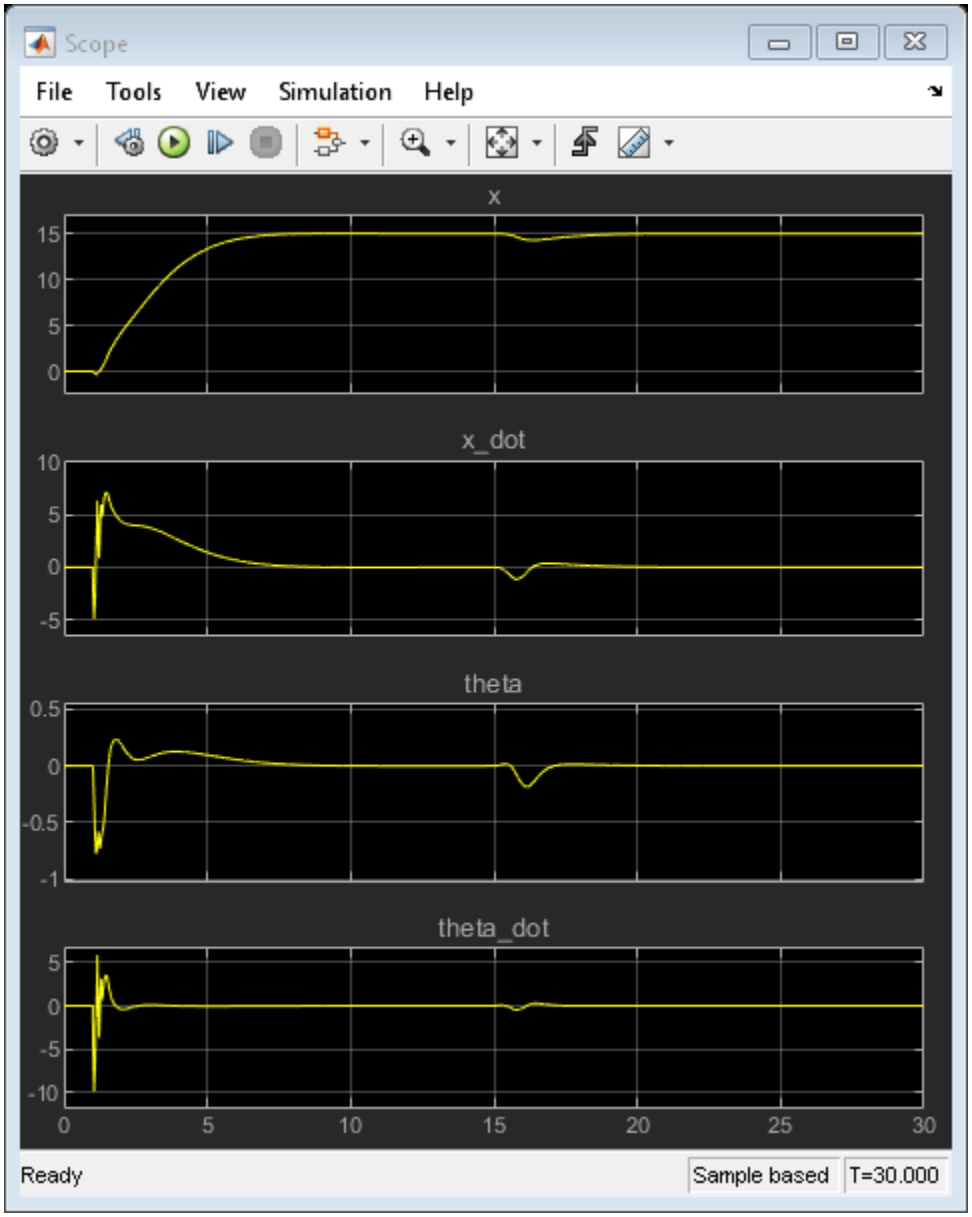
Closed-Loop Simulation

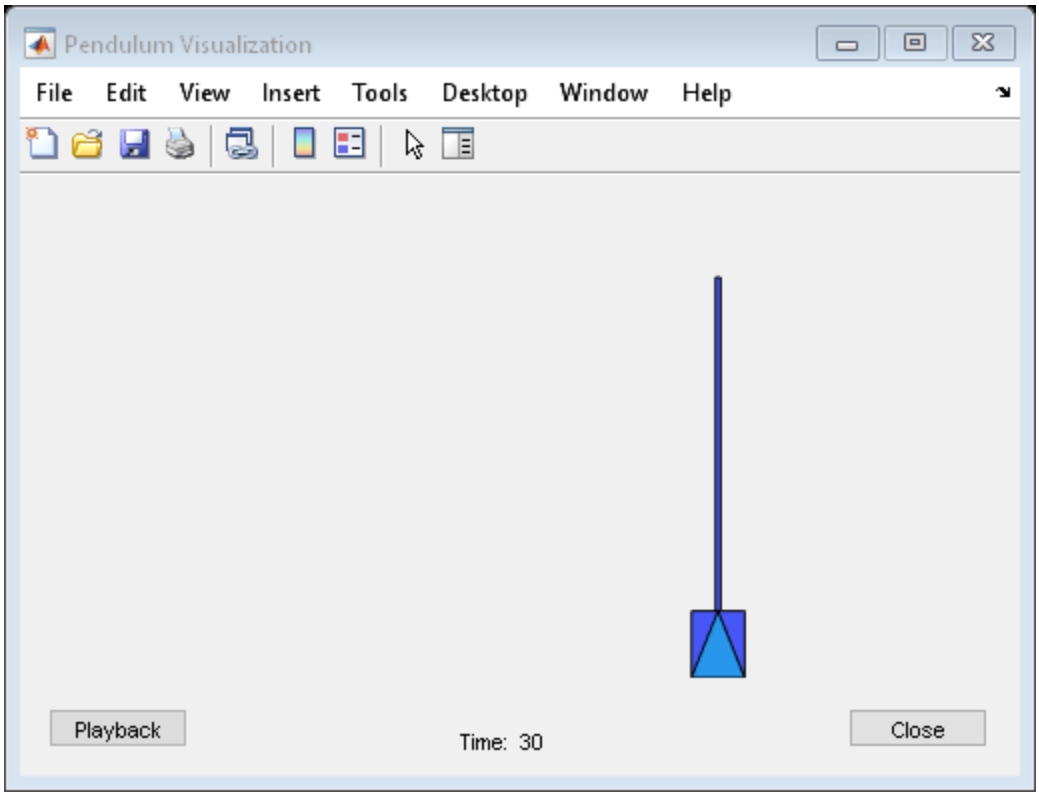
Validate the MPC design with a closed-loop simulation in Simulink.

```
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 measured
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

```
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```





In the nonlinear simulation, all the control objectives are successfully achieved.

Close Simulink model.

```
bdclose(mdlMPC)
```

See Also

Multiple MPC Controllers

More About

- “Gain-Scheduled MPC” on page 8-2
- “Control of an Inverted Pendulum on a Cart” on page 1-132
- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-33

Nonlinear MPC

- “Nonlinear MPC” on page 9-2
- “Specify Prediction Model for Nonlinear MPC” on page 9-5
- “Specify Cost Function for Nonlinear MPC” on page 9-12
- “Specify Constraints for Nonlinear MPC” on page 9-19
- “Configure Optimization Solver for Nonlinear MPC” on page 9-27
- “Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC” on page 9-32
- “Generate Code to Plan and Execute Collision-Free Trajectories using KINOVA Gen3 Manipulator” on page 9-43
- “Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control” on page 9-49
- “Nonlinear Model Predictive Control of an Exothermic Chemical Reactor” on page 9-61
- “Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver” on page 9-68
- “Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant” on page 9-76
- “Optimization and Control of a Fed-Batch Reactor Using Nonlinear MPC” on page 9-84
- “Lane Following Using Nonlinear Model Predictive Control” on page 9-94
- “Lane Change Assist Using Nonlinear Model Predictive Control” on page 9-101
- “Control of Quadrotor Using Nonlinear Model Predictive Control” on page 9-111
- “Economic MPC” on page 9-117
- “Economic MPC Control of Ethylene Oxide Production” on page 9-119
- “Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC” on page 9-127
- “Land a Rocket Using Multistage Nonlinear MPC” on page 9-141

Nonlinear MPC

As in traditional linear MPC, nonlinear MPC calculates control actions at each control interval using a combination of model-based prediction and constrained optimization. The key differences are:

- The prediction model can be nonlinear and include time-varying parameters.
- The equality and inequality constraints can be nonlinear.
- The scalar cost function to be minimized can be a nonquadratic (linear or nonlinear) function of the decision variables.

Using nonlinear MPC, you can:

- Simulate closed-loop control of nonlinear plants under nonlinear costs and constraints.
- Plan optimal trajectories by solving an open-loop constrained nonlinear optimization problem.

By default, nonlinear MPC controllers solve a nonlinear programming problem using the `fmincon` function with the SQP algorithm, which requires Optimization Toolbox™ software. If you do not have Optimization Toolbox software, you can specify your own custom nonlinear solver. For more information on configuring the `fmincon` solver and specifying a custom solver, see “Configure Optimization Solver for Nonlinear MPC” on page 9-27. For more information about using the FORCESPRO NLP solver with nonlinear MPC controllers, see “Implement MPC Controllers using Embotech FORCESPRO Solvers” on page 10-67.

Note The **MPC Designer** app does not support the design of nonlinear MPC controllers.

Generic Nonlinear MPC

To implement generic nonlinear MPC, create an `nlmpc` object, and specify:

- State and output functions that define your prediction model. For more information, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.
- A custom cost function that can replace or augment the standard MPC cost function. For more information, see “Specify Cost Function for Nonlinear MPC” on page 9-12.
- Standard bounds on inputs, outputs, and states.
- Additional custom equality and inequality constraints, which can include linear and nonlinear combinations of inputs, outputs, and states. For more information, see “Specify Constraints for Nonlinear MPC” on page 9-19.

You can simulate generic nonlinear MPC controllers:

- In Simulink using the Nonlinear MPC Controller block
- At the command line using `nlmpcmove`

Multistage Nonlinear MPC

A multistage MPC problem is an MPC problem in which cost and constraint functions are stage-based. Specifically, a multistage MPC controller with a prediction horizon of length p has $p+1$ stages, where the first stage corresponds to the current time and the last (terminal) stage corresponds to the last prediction step.

For a multistage MPC controller, each stage can have its own decision variables and parameters, as well as its own nonlinear cost and constraints. More importantly, cost and constraint functions at a specific stage are functions only of the decision variables and parameters at that stage. Other than make it easier to write Jacobian functions, this feature allows for a much more efficient data structure, which in turn significantly reduces computation times compared to the same problem solved using a generic NLMPC controller. The fact that slack variables are stage-based allows for more design flexibility, and disabling the use of manipulated variable rates as decision variables yields an even leaner problem formulation.

For these reason, if your nonlinear MPC problem has cost and constraint functions that do not involve cross-stage terms, use multistage nonlinear MPC controller in your design.

To implement a multistage nonlinear MPC controller, first create an `nlmpcMultistage` object, and then specify:

- State functions that define your prediction model. For discrete-time models, make sure `Model.IsContinuousTime` is set to `false`.
- Cost and constraint functions at the desired stages. You must specify the cost function for at least one stage.
- Hard upper and lower bounds on states, manipulated variables, and manipulated variable rates, if needed.

When designing your controller, consider the following points.

- Anonymous functions are not supported for `nlmpcMultistage` objects.
- Specifying Jacobians when they are available is the best practice, otherwise the solver must compute them numerically at each step.
- Unlike in generic nonlinear MPC, plant outputs, weights, ECR values, and scale factors are not present in an `nlmpcMultistage` object. You can implement them directly in your cost and constraint functions.
- The control horizon is also omitted in `nlmpcMultistage` objects. To implement block moves, set `RateMin` and `RateMax` to zero at desired prediction steps.

You can simulate multistage nonlinear MPC controllers:

- In Simulink using the Multistage Nonlinear MPC Controller block
- At the MATLAB command line using `nlmpcmove`

Code generation from a nonlinear multistage controller is supported in both MATLAB (using `mpcmoveCodeGeneration`) and Simulink.

For examples on how to create and use a multistage MPC controller, see “Create and Simulate Multistage Nonlinear MPC Controller”, “Simulate Multistage Nonlinear MPC Controller Using Initial Guesses”, and “Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC” on page 9-127.

See Also

Functions
`nlmpc` | `nlmpcMultistage` | `nlmpcmove`

Blocks
Nonlinear MPC Controller | Multistage Nonlinear MPC Controller

More About

- “Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC” on page 9-32
- “Economic MPC” on page 9-117

Specify Prediction Model for Nonlinear MPC

The prediction model of a nonlinear MPC controller consists of the following user-defined functions:

- State function — Predicts how the plant states evolve over time
- Output function — Calculates plant outputs in terms of state and input variables

You can specify either a continuous-time or a discrete-time prediction model.

Before simulating your controller, it is best practice to validate your custom functions, including the state function, output function, and their Jacobians using the `validateFcns` command.

State Function

You can specify either a continuous-time or a discrete-time state function. For a:

- Continuous-time prediction model, the state function is the state derivative function.

  $dx/dt = f(x, u)$
- Discrete-time prediction model, the state function is the state update function.

  $x(k + 1) = f(x(k), u(k))$

Since a nonlinear MPC controller is a discrete-time controller, if your state function is continuous-time, the controller automatically discretizes the model using the implicit trapezoidal rule. This method can handle moderately stiff models, and its prediction accuracy depends on the controller sample time; that is, a large sample time can potentially lead to inaccurate prediction.

If the default discretization method does not provide satisfactory prediction for your application, you can specify your own discrete-time prediction model that uses a different method. To do so, you can integrate a continuous-time state function from the given initial condition, x_k, to the next state, x_{k+1}. When doing so numerically, avoid approaches that require iterations, such as some variable-step-size methods, because these methods introduce numerical noise that degrades solver performance. An explicit multistep Euler method with sufficiently small step size is often the best method to try first. For an example, see “Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control” on page 9-49.

You can specify your state function in one of the following ways.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myStateFunction";
  ```
- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myStateFunction;
  ```
- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myStateFunction(x,u,params)
  ```

Your state function must have one of the following signatures.

- If your controller does not use optional parameters:

```
function z = myStateFunction(x,u)
```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

```
function z = myStateFunction(x,u,params)
```

This table describes the inputs and outputs of this function, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_u is the number of inputs, including all manipulated variables, measured disturbances, and unmeasured disturbances, and is equal to the `Dimensions.NumberOfInputs` property of the controller.

Argument	Input/Output	Description
`x`	Input	Current states, specified as a column vector of length N_x.
`u`	Input	Current inputs, specified as a column vector of length N_u.
`params`	Input	Optional parameters, specified as a comma-separated list (for example `p1,p2,p3`). The same parameters are passed to the prediction model, custom cost function, and custom constraint functions of the controller. If your model uses optional parameters, you must specify the number of parameters using the `Model.NumberOfParameters` property of the controller.
`z`	Output	State function output, returned as a column vector of length N_x. For a continuous-time prediction model, `z` contains the state derivatives, `dx/dt`, and for discrete-time prediction models, `z` contains the next states, `x(k+1)`.

As an example of a state function, consider the continuous-time model with the following state equations:

$$\dot{x}_1 = x_4$$
$$\dot{x}_2 = x_5$$
$$\dot{x}_3 = x_6$$
$$\dot{x}_4 = (u_1 - u_2 + u_3 - u_4)\cos(x_3)$$
$$\dot{x}_5 = (u_1 - u_2 + u_3 - u_4)\sin(x_3)$$
$$\dot{x}_6 = 0.2(u_1 - u_2 - u_3 + u_4)$$

You can specify the state function as follows:

```
z = zeros(6,1);
z(1) = x(4);
z(2) = x(5);
z(3) = x(6);
z(4) = (u(1) - u(2) + u(3) - u(4))*cos(x(3));
z(5) = (u(1) - u(2) + u(3) - u(4))*sin(x(3));
z(6) = 0.2*(u(1) - u(2) - u(3) + u(4));
```

State Function Jacobians

To improve computational efficiency, it is best practice to specify an analytical Jacobian for your state function. If you do not specify a Jacobian, the controller computes the Jacobian using numerical perturbation. To specify a Jacobian for your state function, set the `Jacobian.StateFcn` property of the controller to one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.StateFcn = "myStateJacobian";
  ```
- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.StateFcn = @myStateJacobian;
  ```
- Anonymous function

  ```
  Jacobian.StateFcn = @(x,u,params) myStateJacobian(x,u,params)
  ```

Your state Jacobian function must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function [A,Bmv] = myStateJacobian(x,u)
  ```
- If your controller uses parameters. Here `params` is a comma-separated list of parameters:

  ```
  function [A,Bmv] = myStateJacobian(x,u,params)
  ```

The input arguments of the state Jacobian function are the same as the inputs of the state function. This table describes the outputs of the Jacobian function, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller
- N_u is the number of inputs, including all manipulated variables, measured disturbances, and unmeasured disturbances, and is equal to the `Dimensions.NumberOfInputs` property of the controller

Argument	Description
`A`	Jacobian of the state function output, `z`, with respect to `x`, returned as an N_x-by-N_x array, where $\mathrm{A}(i,j) = \partial \mathrm{z}(i)/\partial \mathrm{x}(j)$.
`Bmv`	Jacobian of the state function output with respect to the manipulated variables, specified as an N_x-by-N_{mv} array, where $\mathrm{Bmv}(i,j) = \partial \mathrm{z}(i)/\partial \mathrm{u}(MV(j))$ and $MV(j)$ is the jth MV index in the `Dimensions.MVIndex` controller property. `Bmv` contains the gradients with respect to only the manipulated variables in `u`, since the measured and unmeasured disturbances are not decision variables.

Consider again, the state function with the following state equations:

$$\dot{x}_1 = x_4$$
$$\dot{x}_2 = x_5$$
$$\dot{x}_3 = x_6$$
$$\dot{x}_4 = (u_1 - u_2 + u_3 - u_4)\cos(x_3)$$
$$\dot{x}_5 = (u_1 - u_2 + u_3 - u_4)\sin(x_3)$$
$$\dot{x}_6 = 0.2(u_1 - u_2 - u_3 + u_4)$$

To find the Jacobians, compute the partial derivatives of the state equations with respect to the states and manipulated variables, assuming that all four inputs are manipulated variables.

```
A = zeros(6,6);
A(1,4) = 1;
A(2,5) = 1;
A(3,6) = 1;
A(4,3) = -(u(1) - u(2) + u(3) - u(4))*sin(x(3));
A(5,3) =  (u(1) - u(2) + u(3) - u(4))*cos(x(3));
B = zeros(6,4);
B(4,:) = cos(x(3))*[1 -1 1 -1];
B(5,:) = sin(x(3))*[1 -1 1 -1];
B(6,:) = 0.2*[1 -1 -1 1];
```

Output Function

The output function of your prediction model relates the states and inputs at the current control interval to the outputs. If the number of states and outputs of the prediction model are the same, you can omit `OutputFcn`, which implies that all states are measurable; that is, each output corresponds to one state.

Note `OutputFcn` cannot have direct feedthrough from any manipulated variable to any output at any time; in other words, nonlinear MPC always assumes `Dmv = 0`.

You can specify your output function in one of the following ways.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.OutputFcn = "myOutputFunction";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.OutputFcn = @myOutputFunction;
  ```

- Anonymous function

  ```
  Model.OutputFcn = @(x,u,params) myOutputFunction(x,u,params)
  ```

Your output function must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function y = myOutputFunction(x,u)
  ```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

  ```
  function y = myOutputFunction(x,u,params)
  ```

This table describes the inputs and outputs of this function, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_u is the number of inputs, including all manipulated variables, measured disturbances, and unmeasured disturbances, and is equal to the `Dimensions.NumberOfInputs` of the controller.

- N_Y is the number of outputs and is equal to the `Dimensions.NumberOfOutputs` property of the controller.

Argument	Input/Output	Description
`x`	Input	Current states, specified as a column vector of length N_x.
`u`	Input	Current inputs, specified as a column vector of length N_u.
`params`	Input	Optional parameters, specified as a comma-separated list (for example `p1,p2,p3`). The same parameters are passed to the prediction model, custom cost function, and custom constraint functions of the controller. If your model uses optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`.
`y`	Output	Current outputs, returned as a column vector of length N_y.

As an example of an output function, consider the following output equations. Recall, that your output function cannot have direct feedthrough from any manipulated variable to any output at any time.

$$y_1 = x_1$$
$$y_2 = x_2 + 0.2x_3$$
$$y_3 = x_3 \cdot x_4$$

You can specify the output function as follows:

```
y = zeros(6,1);
y(1) = x(1);
y(2) = x(2)+0.2*x(3);
y(3) = x(3)*x(4);
```

Output Function Jacobians

To improve computational efficiency, it is best practice to specify an analytical Jacobian for your output function. If you do not specify a Jacobian, the controller computes the Jacobian using numerical perturbation. To specify a Jacobian for your output function, set the `Jacobian.OutputFcn` property of the controller to one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.OutputFcn = "myOutputJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.OutputFcn = @myOutputJacobian;
  ```

- Anonymous function

  ```
  Jacobian.OutputFcn = @(x,u,params) myOutputJacobian(x,u,params)
  ```

Your output Jacobian function must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function C = myOutputJacobian(x,u)
  ```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

  ```
  function C = myOutputJacobian(x,u,params)
  ```

The input arguments of the output Jacobian function are the same as the inputs of the output function. This table describes the output of the Jacobian function. Since the output function cannot have direct feedthrough from any manipulated variable to any output, the output Jacobian function returns only the gradients of the output function with respect to the model states.

Argument	Description
`C`	Jacobian of the output function, returned as an N_y-by-N_x array, where $C(i, j) = \partial y(i)/\partial x(j)$.

Consider again, the model with the following output equations:

$$y_1 = x_1$$
$$y_2 = x_2 + 0.2x_3$$
$$y_3 = x_3 \cdot x_4$$

To find the Jacobians, compute the partial derivatives of the output equations with respect to the states. Since the output function cannot have direct feedthrough from any manipulated variable to any output at any time, you do not compute the Jacobian with respect to the manipulated variables.

```
C = zeros(3,4);
C(1,1) = 1;
C(2,2) = 1;
C(2,3) = 0.2;
C(3,3) = x(4);
C(3,4) = x(3);
```

Specify Optional Model Parameters

You can specify optional parameters for your nonlinear MPC prediction model, cost function, and custom constraints. To do so, pass a comma-separated list of parameter arguments to your custom functions (for example `p1,p2,p3`). These parameters must be numerical values. Also, you must specify the number of model parameters using the `Model.NumberOfParameters` property of the controller.

The same parameters are passed to the prediction model, custom cost function, custom constraint functions, and their respective Jacobians. For example, even if the state function uses only parameter `p1`, the constraint functions use only parameter `p2`, and the cost function uses only parameter `p3`, you must still define three parameters. All of these parameters are passed into all of these functions, and you must choose the correct parameter to use in each function.

For an example that specifies the sample time of a discrete-time state function as a parameter, see “Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control” on page 9-49.

Augment Prediction Model with Unmeasured Disturbances

You can augment your prediction model to include unmeasured disturbances. For example, if your plant does not have an integrator and you want to reject a certain unmeasured disturbance, augment your plan with a disturbance model. Doing so allows the external state estimator to detect the disturbance and thus give the nonlinear MPC controller enough information to reject it.

During simulation, the controller passes a zero to each unmeasured disturbance input channel, since the signals are unmeasured and assumed to be zero-mean by default.

To augment your prediction model, you must designate one or more input signals as unmeasured disturbances when creating a controller using `nlmpc`. For example, create a controller where the first two inputs are manipulated variables and the third input is an unmeasured disturbance (UD).

```
nlobj = nlmpc(nx,ny,'MV',[1 2],'UD',3);
```

Specify the unmeasured disturbance model in the state and output functions of your prediction model. This unmeasured disturbance model can be any arbitrary model that accurately captures the effect of the disturbance on your plant. For example:

- If you expect a step-like UD at a plant output, then specify the UD model as an integrator in your state function, and add the integrator state to the plant output in your output function.
- If you expect a ramp-like UD at a plant input, then specify the UD model as an integrator and add the integrator output to the input signal in your state function.

Any states that you add when specifying the unmeasured disturbance model are included in the prediction model state vector. The values in these unmeasured disturbance model states reflect the disturbance behavior during simulation. This state vector corresponds to the `x` input argument of your state function and the `X` input argument to your custom cost and constraint functions.

For an example that augments the prediction model for random step-like output disturbances, see “Nonlinear Model Predictive Control of an Exothermic Chemical Reactor” on page 9-61.

See Also

`nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Specify Cost Function for Nonlinear MPC” on page 9-12
- “Specify Constraints for Nonlinear MPC” on page 9-19

Specify Cost Function for Nonlinear MPC

While traditional linear MPC controllers optimize control actions to minimize a quadratic cost function, nonlinear MPC controllers support generic custom cost functions. For example, you can specify your cost function as a combination of linear or nonlinear functions of the system states and inputs. To improve computational efficiency, you can also specify an analytical Jacobian for your custom cost function.

Using a custom cost function, you can, for example:

- Maximize profitability
- Minimize energy consumption

When you specify a custom cost function for your nonlinear MPC controller, you can choose to either replace or augment the standard quadratic MPC cost function. By default, an `nlmpc` controller replaces the standard cost function with your custom cost function. In this case, the controller ignores the standard tuning weights in its `Weights` property.

To use an objective function that is the sum of the standard costs and your custom costs, set the `Optimization.ReplaceStandardCost` property of your `nlmpc` object to `false`. In this case, the standard tuning weights specified in the `Weights` property of the controller contribute to the cost function. However, you can eliminate any of the standard cost function terms by setting the corresponding penalty weight to zero. For more information on the standard MPC cost function, see “Standard Cost Function” on page 2-7.

Before simulating your controller, it is best practice to validate your custom functions, including the cost function and its Jacobian, using the `validateFcns` command.

Custom Cost Function

To configure your nonlinear MPC controller to use a custom cost function, set its `Optimization.CustomCostFcn` property to one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomCostFcn = "myCostFunction";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Optimization.CustomCostFcn = @myCostFunction;
  ```

- Anonymous function

  ```
  Optimization.CustomCostFcn = @(X,U,e,data,params) myCostFunction(X,U,e,data,params);
  ```

Your custom cost function must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function J = myCostFunction(X,U,e,data)
  ```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

  ```
  function J = myCostFunction(X,U,e,data,params)
  ```

This table describes the inputs and outputs of this function, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_u is the number of inputs, including all manipulated variables, measured disturbances, and unmeasured disturbances, and is equal to the `Dimensions.NumberOfInputs` property of the controller.
- p is the prediction horizon.
- k is the current time.

Argument	Input/Output	Description
`X`	Input	State trajectory from time k to time $k+p$, specified as a $(p+1)$-by-N_x array. The first row of `X` contains the current state values, which means that the solver does not use the values in `X(1,:)` as decision variables during optimization.
`U`	Input	Input trajectory from time k to time $k+p$, specified as a $(p+1)$-by-N_u array. The final row of `U` is always a duplicate of the preceding row; that is, `U(end,:) = U(end-1,:)`. Therefore, the values in the final row of `U` are not independent decision variables during optimization.
`e`	Input	Slack variable for constraint softening, specified as a nonnegative scalar. `e` is zero if there are no soft constraints in your controller. If you have nonlinear soft constraints defined in your inequality constraint function (`Model.CustomIneqConFcn`), use a positive penalty weight on `e` and make them part of the cost function.

<table>
<tr><th>Argument</th><th>Input/Output</th><th>Description</th></tr>
<tr><td>data</td><td>Input</td><td>Additional signals, specified as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Ts</td><td>Prediction model sample time, as defined in the Ts property of the controller</td></tr>
<tr><td>CurrentStates</td><td>Current prediction model states, as specified in the x input argument of nlmpcmove</td></tr>
<tr><td>LastMV</td><td>MV moves used in previous control interval, as specified in the lastmv input argument of nlmpcmove</td></tr>
<tr><td>References</td><td>Reference values for plant outputs, as specified in the ref input argument of nlmpcmove</td></tr>
<tr><td>MVTarget</td><td>Manipulated variable targets, as specified in the MVTarget property of an nlmpcmoveopt object</td></tr>
<tr><td>PredictionHorizon</td><td>Prediction horizon, as defined in the PredictionHorizon property of the controller</td></tr>
<tr><td>NumOfStates</td><td>Number of states, as defined in the Dimensions.NumberOfStates property of the controller</td></tr>
<tr><td>NumOfOutputs</td><td>Number of outputs, as defined in the Dimensions.NumberOfOutputs property of the controller</td></tr>
<tr><td>NumOfInputs</td><td>Number of inputs, as defined in the Dimensions.NumberOfInputs property of the controller</td></tr>
<tr><td>MVIndex</td><td>Manipulated variables indices, as defined in the Dimensions.MVIndex property of the controller</td></tr>
<tr><td>MDIndex</td><td>Measured disturbance indices, as defined in the Dimensions.MDIndex property of the controller</td></tr>
<tr><td>UDIndex</td><td>Unmeasured disturbance indices, as defined in the Dimensions.UDIndex property of the controller</td></tr>
</table>
</td></tr>
</table>

Argument	Input/Output	Description
`params`	Input	Optional parameters, specified as a comma-separated list (for example `p1,p2,p3`). The same parameters are passed to the prediction model, custom cost function, and custom constraint functions of the controller. For example, if the state function uses only parameter `p1`, the constraint functions use only parameter `p2`, and the cost function uses only parameter `p3`, then all three parameters are passed to all of these functions. If your model uses optional parameters, you must specify the number of parameters using the `Model.NumberOfParameters` property of the controller.
`J`	Output	Computed cost, returned as a scalar

Your custom cost function must:

- Be a continuous, finite function of `U`, `X`, and `e` and have finite first derivatives
- Increase as the slack variable `e` increases or be independent of it

To use output variable values in your cost function, you must first derive them from the state and input arguments using the prediction model output function, as specified in the `Model.OutputFcn` property of the controller. For example, to compute the output trajectory `Y` from time k to time $k+p$, use:

```
p = data.PredictionHorizon;
for i=1:p+1
    Y(i,:) = myOutputFunction(X(i,:)',U(i,:)',params)';
end
```

For more information on the prediction model output function, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.

Typically, you optimize control actions to minimize the cost function across the prediction horizon. Since the cost function value must be a scalar, you compute the cost function at each prediction horizon step and add the results together. For example, suppose that the stage cost function is:

$$J = 10u_1^2 + 5x_2^3 + x_1$$

That is, you want to minimize the difference between the first output and its reference value, and the product of the first manipulated variable and the second state. To compute the total cost function across the prediction horizon, use:

```
p = data.PredictionHorizon;
U1 = U(1:p,data.MVIndex(1));
X1 = X(2:p+1,1);
X2 = X(2:p+1,2);
J = 10*sum(sum(U1.^2)) + 5*sum(sum(X2.^3) + sum(sum(X1));
```

In general, for cost functions, do not use the following values, since they are not part of the decision variables used by the solver:

- `U(end,:)` — This row is a duplicate of the preceding row.
- `X(1,:)` — This row contains the current state values.

Since this example cost function is relatively simple, you can specify it using an anonymous function handle.

For relatively simple costs, you can specify the cost function using an anonymous function handle. For example, to specify an anonymous function that implements just the first term of the preceding cost function, use:

```
Optimization.CustomCostFcn = @(X,U,data) 10*sum(sum((U(1:end-1,data.MVIndex(1)).^2));
```

Cost Function Jacobian

To improve computational efficiency, it is best practice to specify an analytical Jacobian for your custom cost function. If you do not specify a Jacobian, the controller computes the Jacobian using numerical perturbation. To specify a Jacobian for your cost function, set the `Jacobian.CustomCostFcn` property of the controller to one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomCostFcn = "myCostJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomCostFcn = @myCostJacobian;
  ```

- Anonymous function

  ```
  Jacobian.CustomCostFcn = @(X,U,e,data,params) myCostJacobian(X,U,e,data,params)
  ```

Your cost Jacobian function must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function [G,Gmv,Ge] = myCostJacobian(X,U,e,data)
  ```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

  ```
  function [G,Gmv,Ge] = myCostJacobian(X,U,e,data,params)
  ```

The input arguments of the cost Jacobian function are the same as the inputs of the custom cost function. This table describes the outputs of the Jacobian function, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_{mv} is the number of manipulated variables.
- p is the prediction horizon.

Argument	Description
`G`	Jacobian of the cost function with respect to the state trajectories, returned as a p-by-N_x array, where $G(i, j) = \partial J / \partial X(i + 1, j)$. Compute `G` based on `X` from the second row to row p+1, ignoring the first row.

Argument	Description
`Gmv`	Jacobian of the cost function with respect to the manipulated variable trajectories, returned as a p-by-N_{mv} array, where $\mathrm{Gmv}(i,j) = \partial \mathrm{J}/\partial \mathrm{U}(i, MV(j))$ and $MV(j)$ is the jth MV index in `data.MVIndex`. Since the controller forces `U(p+1,:)` to equal `U(p,:)`, if your cost function uses `U(p+1,:)`, you must include the impact of both `U(p,:)` and `U(p+1,:)` in the Jacobian for `U(p,:)`.
`Ge`	Jacobian of the cost function with respect to the slack variable, `e`, returned as a scalar, where $\mathrm{Ge} = \partial \mathrm{J}/\partial \mathrm{e}$.

To use output variable values and their Jacobians in your cost Jacobian function, you must first derive them from the state and input arguments. To do so, use the Jacobian of the prediction model output function, as specified in the `Jacobian.OutputFcn` property of the controller. For example, to compute the output variables `Y` and their Jacobians `Yjacob` from time k to time $k+p$, use:

```
p = data.PredictionHorizon;
for i=1:p+1
    Y(i,:) = myOutputFunction(X(i,:)',U(i,:)',params)';
end
for i=1:p+1
    Yjacob(i,:) = myOutputJacobian(X(i,:)',U(i,:)',params)';
end
```

Since prediction model output functions do not support direct feedthrough from inputs to outputs, the output function Jacobian contains partial derivatives with respect to only the states in `X`. For more information on the output function Jacobian, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.

To find the Jacobians, compute the partial derivatives of the cost function with respect to the state trajectories, manipulated variable trajectories, and slack variable. For example, suppose that your cost function is as follows, where u_1 is the first manipulated variable.

$$J = 10u_1^2 + 5x_2^3 + x_1$$

To compute the Jacobian with respect to the state trajectories, use the following. Recall that you compute `G` based on `X` from the second row to row $p+1$, ignoring the first row.

```
p = data.PredictionHorizon;
Nx = data.NumOfStates;
U1 = U(1:p,data.MVIndex(1));
X2 = X(2:p+1,2);

G = zeros(p,Nx);
G(1:p,1) = 1;
G(1:p,2) = 15*X2.^2;
```

To compute the Jacobian with respect to the manipulated variable trajectories, use:

```
Nmv = length(data.MVIndex);

Gmv = zeros(p,Nmv);
Gmv(1:p,1) = 20*U1;
```

In this case, the derivative with respect to the slack variable is `Ge = 0`.

See Also

`nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Specify Prediction Model for Nonlinear MPC” on page 9-5
- “Specify Constraints for Nonlinear MPC” on page 9-19

Specify Constraints for Nonlinear MPC

When you create a nonlinear MPC controller using an `nlmpc` object, you can define any of the following constraints:

- Standard linear constraints on states, outputs, manipulated variables, and manipulated variable rates of change
- Custom equality constraints, specified as linear or nonlinear functions of the system states, inputs, and outputs
- Custom inequality constraints, specified as linear or nonlinear functions of the system states, inputs, and outputs

The controller optimizes its control moves to satisfy all of these constraints; that is, the custom constraints supplement the standard linear constraints.

To improve computational efficiency, you can also specify analytical Jacobians for your custom equality and inequality constraints.

By specifying custom equality or inequality constraints, you can, for example:

- Require the plant to reach a target state at the end of the prediction horizon
- Require cumulative resource consumption to stay within specified limits

Before simulating your controller, it is best practice to validate your custom functions, including the constraint functions and their Jacobians, using the `validateFcns` command.

Linear MPC controllers have properties for defining custom constraints on linear combinations of inputs and outputs, as discussed in “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5. These properties are not available for nonlinear MPC controllers. Instead, you implement such constraints within your custom equality or inequality constraint functions.

Standard Linear Constraints

The following table shows the standard linear constraints supported by nonlinear MPC controllers. For each of these constraints, you can specify a single bound that applies across the entire prediction horizon, or you can vary each constraint over the prediction horizon. For more information on setting controller linear constraint properties, see `nlmpc`.

Constraint	Controller Property	Constraint Softening
Lower bounds on state `i`	`States(i).Min > -Inf`	Not applicable. State bounds are always hard.
Upper bounds on state `i`	`States(i).Max < Inf`	Not applicable. State bounds are always hard.
Lower bounds on output variable `i`	`OutputVariables(i).Min > -Inf`	`OutputVariables(i).MinECR > 0` **Default:** 1 (soft)

Constraint	Controller Property	Constraint Softening
Upper bounds on output variable `i`	`OutputVariables(i).Max < Inf`	`OutputVariables(i).MaxECR > 0` **Default:** 1 (soft)
Lower bounds on manipulated variable `i`	`ManipulatedVariables(i).Min > -Inf`	`ManipulatedVariables(i).MinECR > 0` **Default:** `0` (hard)
Upper bounds on manipulated variable `i`	`ManipulatedVariables(i).Max < Inf`	`ManipulatedVariables(i).MaxECR > 0` **Default:** `0` (hard)
Lower bounds on manipulated variable `i` rate of change	`ManipulatedVariables(i).RateMin > -Inf`	`ManipulatedVariables(i).RateMinECR > 0` **Default:** `0` (hard)
Lower bounds on manipulated variable `i` rate of change	`ManipulatedVariables(i).RateMax < Inf`	`ManipulatedVariables(i).RateMaxECR > 0` **Default:** `0` (hard)

Custom Constraints

You can specify custom equality and inequality constraints for a nonlinear MPC controller. To configure your nonlinear MPC controller to use custom equality or inequality constraints, set its `Optimization.CustomEqConFcn` or `Optimization.CustomIneqConFcn` respectively. To do so, specify the custom functions as one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomEqConFcn = "myEqConFunction";
  Optimization.CustomIneqConFcn = "myIneqConFunction";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Optimization.CustomEqConFcn = @myEqConFunction;
  Optimization.CustomIneqConFcn = @myIneqConFunction;
  ```

- Anonymous function

  ```
  Optimization.CustomEqConFcn = ...
      @(X,U,data,params) myEqConFunction(X,U,data,params);
  Optimization.CustomIneqConFcn = ...
      @(X,U,e,data,params) myIneqConFunction(X,U,e,data,params);
  ```

Your constraint functions must have one of the following signatures.

- If your controller does not use optional parameters:

  ```
  function ceq = myEqConFunction(X,U,data)
  function cineq = myIneqConFunction(X,U,e,data)
  ```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

```
function ceq = myEqConFunction(X,U,data,params)
function cineq = myIneqConFunction(X,U,e,data,params)
```

This table describes the inputs and outputs of these functions, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_u is the number of inputs, including all manipulated variables, measured disturbances, and unmeasured disturbances, and is equal to the `Dimensions.NumberOfInputs` property of the controller.
- N_{ceq} is the number of equality constraints.
- N_{cineq} is the number of inequality constraints.
- p is the prediction horizon.
- k is the current time.

Argument	Input/Output	Description
`X`	Input	State trajectory from time k to time $k+p$, specified as a $(p+1)$-by-N_x array. The first row of `X` contains the current state values, which means that the solver does not use the values in `X(1,:)` as decision variables during optimization.
`U`	Input	Input trajectory from time k to time $k+p$, specified as a $(p+1)$-by-N_u array. The final row of `U` is always a duplicate of the preceding row; that is, `U(end,:) = U(end-1,:)`. Therefore, the values in the final row of `U` are not independent decision variables during optimization.
`e`	Input	Slack variable for constraint softening, specified as a positive scalar. Since all equality constraints are hard, this input argument applies to only the inequality constraint function.

<table>
<tr><th>Argument</th><th>Input/Output</th><th>Description</th></tr>
<tr><td>data</td><td>Input</td><td>Additional signals, specified as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Ts</td><td>Prediction model sample time, as defined in the Ts property of the controller</td></tr>
<tr><td>CurrentStates</td><td>Current prediction model states, as specified in the x input argument of nlmpcmove</td></tr>
<tr><td>LastMV</td><td>MV moves used in previous control, as specified in the lastmv input argument of nlmpcmove interval</td></tr>
<tr><td>References</td><td>Reference values for plant outputs, as specified in the ref input argument of nlmpcmove</td></tr>
<tr><td>MVTarget</td><td>Manipulated variable targets, as specified in the MVTarget property of an nlmpcmoveopt object</td></tr>
<tr><td>PredictionHorizon</td><td>Prediction horizon, as defined in the PredictionHorizon property of the controller</td></tr>
<tr><td>NumOfStates</td><td>Number of states, as defined in the Dimensions.NumberOfStates property of the controller</td></tr>
<tr><td>NumOfOutputs</td><td>Number of outputs, as defined in the Dimensions.NumberOfOutputs property of the controller</td></tr>
<tr><td>NumOfInputs</td><td>Number of inputs, as defined in the Dimensions.NumberOfInputs property of the controller</td></tr>
<tr><td>MVIndex</td><td>Manipulated variables indices, as defined in the Dimensions.MVIndex property of the controller</td></tr>
<tr><td>MDIndex</td><td>Measured disturbance indices, as defined in the Dimensions.MDIndex property of the controller</td></tr>
<tr><td>UDIndex</td><td>Unmeasured disturbance indices, as defined in the Dimensions.UDIndex property of the controller</td></tr>
</table>
</td></tr>
</table>

Argument	Input/Output	Description
`params`	Input	Optional parameters, specified as a comma-separated list (for example `p1,p2,p3`). The same parameters are passed to the prediction model, custom cost function, and custom constraint functions of the controller. For example, if the state function uses only parameter `p1`, the constraint functions use only parameter `p2`, and the cost function uses only parameter `p3`, then all three parameters are passed to all of these functions. If your model uses optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`.
`ceq`	Output	Computed equality constraint values, returned as a column vector of length N_{ceq}. An equality constraint is satisfied when the corresponding output is `0`.
`cineq`	Output	Computed inequality constraint values, returned as a column vector of length N_{cineq}. An inequality constraint is satisfied when the corresponding output is less than or equal to `0`.

To use output variable values in your constraint functions, you must first derive them from the state and input arguments using the prediction model output function, as specified in the `Model.OutputFcn` property of the controller. For example, to compute the output trajectory `Y` from time k to time $k+p$, use:

```
p = data.PredictionHorizon;
for i=1:p+1
    Y(i,:) = myOutputFunction(X(i,:)',U(i,:)',params)';
end
```

For more information on the prediction model output function, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.

In general:

- All equality constraints are hard.
- To define soft inequality constraints, use the slack variable input argument, `e`. For more information on constraint softening in MPC, see “Constraint Softening” on page 1-7.
- Equality constraints should be continuous and have continuous first derivatives with respect to the decision variables.

You can define custom constraints that apply across the entire prediction horizon. For example, suppose that you want to satisfy the following inequality constraints across the prediction horizon, where u_1 is the first manipulated variable:

$$2x_1^2 - 3x_2 - 10 \le 0$$

$$u_1^2 - 5 \le 0$$

To define the constraint values across the prediction horizon, use:

```
p = data.PredictionHorizon;
U1 = U(1:p,data.MVIndex(1));
X1 = X(2:p+1,1);
X2 = X(2:p+1,2);
```

```
cineq = [2*X1.^2 - 3*X2 - 10;
         U1.^2 - 5];
```

Applying these two constraints across p prediction horizon steps produces a column vector with 2*p inequality constraints. These inequality constraints are satisfied when the corresponding element of `cineq` is less than or equal to zero.

Alternatively, you can define constraints that apply at specific prediction horizon steps. For example, suppose that you want the states of a third-order plant to be:

$$x_1 = 5$$
$$x_2 = -3$$
$$x_3 = 0$$

To specify these state values as constraints on only the final prediction horizon step, use:

```
ceq = [X(p+1,1) - 5;
       X(p+1,2) + 3;
       X(p+1,3)];
```

These equality constraints are satisfied when the corresponding element of `ceq` is equal to zero.

For relatively simple constraints, you can specify the constraint function using an anonymous function handle. For example, to specify an anonymous function that implements the equality constraints, use:

```
Optimization.CustomEqConFcn = @(X,U,data) [X(p+1,1) - 5; X(p+1,2) + 3; X(p+1,3)];
```

Custom Constraint Jacobians

To improve computational efficiency, it is best practice to specify analytical Jacobians for your custom constraint functions. If you do not specify Jacobians, the controller computes the Jacobians using numerical perturbation.

To specify a Jacobian for your equality or inequality constraint functions, set the respective `Jacobian.CustomEqConFcn` or `Jacobian.CustomIneqConFcn` property of the controller to one of the following.

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomEqConFcn = "myEqConJacobian";
  Jacobian.CustomIneqConFcn = "myIneqConJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomEqConFcn = @myEqConJacobian;
  Jacobian.CustomIneqConFcn = @myIneqConJacobian;
  ```

- Anonymous function

  ```
  Jacobian.CustomEqConFcn = @(X,U,data,params) myEqConJacobian(X,U,data,params);
  Jacobian.CustomInqConFcn = @(X,U,e,data,params) myIneqConJacobian(X,U,e,data,params);
  ```

Your constraint Jacobian functions must have one of the following signatures.

- If your controller does not use optional parameters:

```
function [Geq,Gmv] = myEqConJacobian(X,U,data)
function [Geq,Gmv,Ge] = myIneqConJacobian(X,U,e,data)
```

- If your controller uses parameters. Here, `params` is a comma-separated list of parameters:

```
function [Geq,Gmv] = myEqConJacobian(X,U,data,params)
function [Geq,Gmv,Ge] = myIneqConJacobian(X,U,e,data,params)
```

The input arguments of the constraint Jacobian functions are the same as the inputs of their respective custom constraint functions. This table describes the outputs of the Jacobian functions, where:

- N_x is the number of states and is equal to the `Dimensions.NumberOfStates` property of the controller.
- N_{mv} is the number of manipulated variables.
- N_c is the number of constraints (either equality or inequality constraints, depending on the constraint function).
- p is the prediction horizon.

Argument	Description
`G`	Jacobian of the equality or inequality constraints with respect to the state trajectories, returned as a p-by-N_x-by-N_c array, where $\mathrm{G}(i,j,l) = \partial \mathrm{c}(l)/\partial \mathrm{X}(i+1,j)$. Compute `G` based on `X` from the second row to row $p+1$, ignoring the first row.
`Gmv`	Jacobian of the equality or inequality constraints with respect to the manipulated variable trajectories, returned as a p-by-N_{mv}-by-N_c array, where $\mathrm{Gmv}(i,j,l) = \partial \mathrm{c}(l)/\partial \mathrm{U}(i,MV(j))$ and $MV(j)$ is the jth MV index in `data.MVIndex`. Since the controller forces `U(p+1,:)` to equal `U(p,:)`, if your constraints use `U(p+1,:)`, you must include the impact of both `U(p,:)` and `U(p+1,:)` in the Jacobian for `U(p,:)`.
`Ge`	Jacobian of the inequality constraints with respect to the slack variable, `e`, returned as a row vector of length N_c, where $\mathrm{Ge}(l) = \partial \mathrm{c}(l)/\partial \mathrm{e}$

To use output variable Jacobians in your constraint Jacobian functions, you must first derive them from the state and input arguments using the Jacobian of the prediction model output function, as specified in the `Jacobian.OutputFcn` property of the controller. For example, to compute the output variable Jacobians `Yjacob` from time k to time $k+p$, use:

```
p = data.PredictionHorizon;
for i=1:p+1
    Y(i,:) = myOutputFunction(X(i,:)',U(i,:)',params)';
end
for i=1:p+1
    Yjacob(i,:) = myOutputJacobian(X(i,:)',U(i,:)',params)';
end
```

Since prediction model output functions do not support direct feedthrough from inputs to outputs, the output function Jacobian contains partial derivatives with respect to only the states in `X`. For more information on the output function Jacobian, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.

To find the Jacobians, compute the partial derivatives of the constraint functions with respect to the state trajectories, manipulated variable trajectories, and slack variable. For example, suppose that your constraint function is as follows, where u_1 is the first manipulated variable.

$$2x_1^2 - 3x_2 - 10 \le 0$$

$$u_1^2 - 5 \le 0$$

To compute the Jacobian with respect to the state trajectories, use:

```
Nx = data.NumOfStates;
Nc = 2*p;
G = zeros(p,Nx,Nc);
G(1:p,2,1:p) = diag(2*X1 - 3);
```

To compute the Jacobian with respect to the manipulated variable trajectories, use:

```
Nmv = length(data.MVIndex);
Gmv = zeros(p,Nmv,Nc);
Gmv(1:p,1,p+1:2*p) = diag(2*u(1:p,data.MVIndex(1)));
```

In this case, the derivative with respect to the slack variable is `Ge = zeros(20,1)`.

See Also

`nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Specify Prediction Model for Nonlinear MPC” on page 9-5
- “Specify Cost Function for Nonlinear MPC” on page 9-12

Configure Optimization Solver for Nonlinear MPC

By default, nonlinear MPC controllers solve a nonlinear programming problem using the `fmincon` function with the SQP algorithm, which requires Optimization Toolbox software. If you do not have Optimization Toolbox software, you can specify your own custom nonlinear solver.

Solver Decision Variables

For nonlinear MPC controllers at time t_k, the nonlinear optimization problem uses the following decision variables:

- Predicted state values from time t_{k+1} to t_{k+p}. These values correspond to rows 2 through p+1 of the `X` input argument of your cost, and constraint functions, where p is the prediction horizon.
- Predicted manipulated variables from time t_k to t_{k+p-1}. These values correspond to the manipulated variable columns in rows 1 through p of the `U` input argument of your cost, and constraint functions.

Therefore, the number of decision variables N_Z is equal to $p(N_x+N_{mv}) + 1$, N_x is the number of states, N_{mv} is the number of manipulated variables, and the +1 is for the global slack variable.

Specify Initial Guesses

A properly configured standard linear MPC optimization problem has a unique solution. However, nonlinear MPC optimization problems often allow multiple solutions (local minima), and finding a solution can be difficult for the solver. In such cases, it is important to provide a good starting point near the global optimum.

During closed-loop simulations, it is best practice to *warm start* your nonlinear solver. To do so, use the predicted state and manipulated variable trajectories from the previous control interval as the initial guesses for the current control interval. In Simulink, the Nonlinear MPC Controller block is configured to use these trajectories as initial guesses by default. To use these trajectories as initial guesses at the command line:

1. Return the `opt` output argument when calling `nlmpcmove`. This `nlmpcmoveopt` object contains any run-time options you specified in the previous call to `nlmpcmove`. It also includes the initial guesses for the state (`opt.X0`) and manipulated variable (`opt.MV0`) trajectories, and the global slack variable (`opt.Slack0`).
2. Pass this object in as the `options` input argument to `nlmpcmove` for the next control interval.

These command-line simulation steps are best practices, even if you do not specify any other run-time options.

Configure fmincon Options

By default, nonlinear MPC controllers optimize their control move using the `fmincon` function from the Optimization Toolbox. When you first create your controller, the `Optimization.SolverOptions` property of the `nlmpc` object contains the standard `fmincon` options with the following nondefault settings:

- Use the `SQP` algorithm (`SolverOptions.Algorithm = 'sqp'`)

- Use objective function gradients (`SolverOptions.SpecifyObjectiveGradient = 'true'`)
- Use constraint gradients (`SolverOptions.SpecifyConstraintGradient = 'true'`)
- Do not display optimization messages to the command window (`SolverOptions.Display = 'none'`)

These nondefault options typically improve the performance of the nonlinear MPC controller.

You can modify the solver options for your application. For example, to specify the maximum number of solver iterations for your application, set `SolverOptions.MaxIter`. For more information on the available solver options, see `fmincon`.

In general, you should not modify the `SpecifyObjectiveGradient` and `SpecifyConstraintGradient` solver options, since doing so can significantly affect controller performance. For example, the constraint gradient matrices are sparse, and setting `SpecifyConstraintGradient` to false would cause the solver to calculate gradients that are known to be zero.

Specify Custom Solver

If you do not have Optimization Toolbox software, you can specify your own custom nonlinear solver. To do so, create a custom wrapper function that converts the interface of your solver function to match the interface expected by the nonlinear MPC controller. Your custom function must be a MATLAB script or MAT-file on the MATLAB path. For an example that shows a template custom solver wrapper function, see “Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver” on page 9-68.

You can use the Nonlinear Programming solver developed by Embotech AG to simulate and generate code for nonlinear MPC controllers. For more information, see “Implement MPC Controllers using Embotech FORCESPRO Solvers” on page 10-67.

To configure your `nlmpc` object to use your custom solver wrapper function, set its `Optimization.CustomSolverFcn` property in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomSolverFcn = "myNLPSolver";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Optimization.CustomSolverFcn = @myNLPSolver;
  ```

Your custom solver wrapper function must have the signature:

```
function [zopt,cost,flag] = myNLPSolver(FUN,z0,A,B,Aeq,Beq,LB,UB,NLCON)
```

This table describes the inputs and outputs of this function, where:

- N_Z is the number of decision variables.
- M_{cineq} is the number of linear inequality constraints.
- M_{ceq} is the number of linear equality constraints.
- N_{cineq} is the number of nonlinear inequality constraints.
- N_{ceq} is the number of nonlinear equality constraints.

Argument	Input/Output	Description
`FUN`	Input	Nonlinear cost function to minimize, specified as a handle to a function with the signature: `[F,G] = FUN(z)` and arguments: • `z` — Decision variables, specified as a vector of length N_Z. • `F` — Cost, returned as a scalar. • `G` — Cost function gradients with respect to the decision variables, returned as a column vector of length N_Z, where G(i) = $\partial F / \partial z(i)$.
`z0`	Input	Initial guesses for decision variable values, specified as a vector of length N_Z
`A`	Input	Linear inequality constraint array, specified as an M_{cineq}-by-N_Z array. Together, `A` and `B` define constraints of the form $A \cdot z \le B$.
`B`	Input	Linear inequality constraint vector, specified as a column vector of length M_{cineq}. Together, `A` and `B` define constraints of the form $A \cdot z \le B$.
`Aeq`	Input	Linear equality constraint array, specified as an M_{ceq}-by-N_Z array. Together, `Aeq` and `Beq` define constraints of the form $Aeq \cdot z = Beq$.
`Beq`	Input	Linear equality constraint vector, specified as a column vector of length M_{ceq}. Together, `Aeq` and `Beq` define constraints of the form $Aeq \cdot z = Beq$.
`LB`	Input	Lower bounds for decision variables, specified as a column vector of length N_Z, where $LB(i) \le z(i)$.
`UB`	Input	Upper bounds for decision variables, specified as a column vector of length N_Z, where $z(i) \le UB(i)$.

Argument	Input/Output	Description
`NLCON`	Input	Nonlinear constraint function, specified as a handle to a function with the signature: `[cineq,c,Gineq,Geq] = NLCON(z)` and arguments: • `z` — Decision variables, specified as a vector of length N_Z. • `cineq` — Nonlinear inequality constraint values, returned as a column vector of length N_{cineq}. • `ceq` — Nonlinear equality constraint values, returned as a column vector of length N_{ceq}. • `Gineq` — Nonlinear inequality constraint gradients with respect to the decision variables, returned as an N_Z-by-N_{cineq} array, where $\text{Gineq}(i,j) = \partial \text{cineq}(j)/\partial z(i)$. • `Geq` — Nonlinear equality constraint gradients with respect to the decision variables, returned as an N_Z-by-N_{ceq} array, where $\text{Geq}(i,j) = \partial \text{ceq}(j)/\partial z(i)$.
`zopt`	Output	Optimal decision variable values, returned as a vector of length N_Z.
`cost`	Output	Optimal cost, returned as a scalar.
`flag`	Output	Exit flag, returned as one of the following: • `1` — Optimization successful (first order optimization conditions satisfied) • `0` — Suboptimal solution (maximum number of iterations reached) • Negative value — Optimization failed

When you implement your custom solver function, it is best practice to have your solver use the cost and constraint gradient information provided by the nonlinear MPC controller.

If you are unable to obtain a solution using your custom solver, try to identify a special condition for which you know the solution, and start the solver at this condition. If the solver diverges from this initial guess:

- Check the validity of the state and output functions in your prediction model.
- If you are using a custom cost function, make sure it is correct.
- If you are using the standard MPC cost function, verify the controller tuning weights.
- Make sure that all constraints are feasible at the initial guess.
- If you are providing custom Jacobian functions, validate your Jacobians using `validateFcns`.

See Also

`nlmpc` | `fmincon`

More About

- “Nonlinear MPC” on page 9-2

- “Specify Cost Function for Nonlinear MPC” on page 9-12
- “Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver” on page 9-68

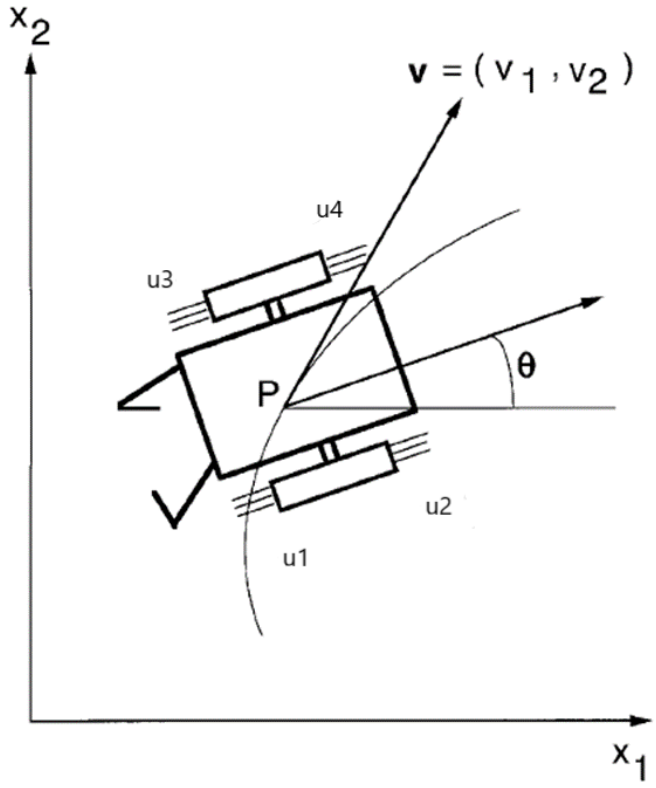
Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC

This example shows how to find the optimal trajectory that brings a flying robot from one location to another with minimum fuel cost using a nonlinear MPC controller. In addition, another nonlinear MPC controller, along with an extended Kalman filter, drives the robot along the optimal trajectory in closed-loop simulation.

Flying Robot

The flying robot in this example has four thrusters to move it around in a 2-D space. The model has six states:

- x(1) - `x` inertial coordinate of center of mass
- x(2) - `y` inertial coordinate of center of mass
- x(3) - `theta`, robot (thrust) direction
- x(4) - `vx`, velocity of x
- x(5) - `vy`, velocity of y
- x(6) - `omega`, angular velocity of theta



For more information on the flying robot, see [1]. The model in the paper uses two thrusts ranging from -1 to 1. However, this example assumes that there are four physical thrusts in the robot, ranging from 0 to 1, to achieve the same control freedom.

Trajectory Planning

The robot initially rests at `[-10,-10]` with an orientation angle of `pi/2` radians (facing north). The flying maneuver for this example is to move and park the robot at the final location `[0,0]` with an angle of `0` radians (facing east) in `12` seconds. The goal is to find the optimal path such that the total amount of fuel consumed by the thrusters during the maneuver is minimized.

Nonlinear MPC is an ideal tool for trajectory planning problems because it solves an open-loop constrained nonlinear optimization problem given the current plant states. With the availability of a nonlinear dynamic model, MPC can make more accurate decisions.

In this example, the target prediction time is `12` seconds. Therefore, specify a sample time of `0.4` seconds and prediction horizon of `30` steps. Create a multistage nonlinear MPC object with `6` states and `4` inputs. By default, all the inputs are manipulated variables (MVs).

```
Ts = 0.4;
p = 30;
nx = 6;
nu = 4;
nlobj = nlmpcMultistage(p,nx,nu);
nlobj.Ts = Ts;
```

For a path planning problem, it is typical to allow MPC to have free moves at each prediction step, which provides the maximum number of decision variables for the optimization problem. Since planning usually runs at a much slower sampling rate than a feedback controller, the extra computation load introduced by a larger optimization problem can be accepted.

Specify the prediction model state function using the function name. You can also specify functions using a function handle. For details on the state function, open `FlyingRobotStateFcn.m`. For more information on specifying the prediction model, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
```

Specify the Jacobian of the state function using a function handle. It is best practice to provide an analytical Jacobian for the prediction model. Doing so significantly improves simulation efficiency. For details on the Jacobian function, open `FlyingRobotStateJacobianFcn.m`.

```
nlobj.Model.StateJacFcn = @FlyingRobotStateJacobianFcn;
```

A trajectory planning problem usually involves a nonlinear cost function, which can be used to find the shortest distance, the maximal profit, or as in this case, the minimal fuel consumption. Because the thrust value is a direct indicator of fuel consumption, compute the fuel cost as the sum of the thrust values at each prediction step from stage 1 to stage p. Specify this cost function using a named function. For more information on specifying cost functions, see “Specify Cost Function for Nonlinear MPC” on page 9-12.

```
for ct = 1:p
    nlobj.Stages(ct).CostFcn = 'FlyingRobotCostFcn';
end
```

The goal of the maneuver is to park the robot at `[0,0]` with an angle of `0` radians at the 12th second. Specify this goal as the terminal state constraint, where every position and velocity state at the last prediction step (stage p+1) should be zero. For more information on specifying constraint functions, see “Specify Constraints for Nonlinear MPC” on page 9-19.

```
nlobj.Model.TerminalState = zeros(6,1);
```

It is best practice to provide analytical Jacobian functions for your stage cost and constraint functions as well. However, this example intentionally skips them so that their Jacobian is computed by the nonlinear MPC controller using the built-in numerical perturbation method.

Each thrust has an operating range between `0` and `1`, which is translated into lower and upper bounds on the MVs.

```
for ct = 1:nu
    nlobj.MV(ct).Min = 0;
    nlobj.MV(ct).Max = 1;
end
```

Specify the initial conditions for the robot.

```
x0 = [-10;-10;pi/2;0;0;0];  % robot parks at [-10, -10], facing north
u0 = zeros(nu,1);           % thrust is zero
```

It is best practice to validate the user-provided model, cost, and constraint functions and their Jacobians. To do so, use the `validateFcns` command.

```
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Analysis of user-provided model, cost, and constraint functions complete.
```

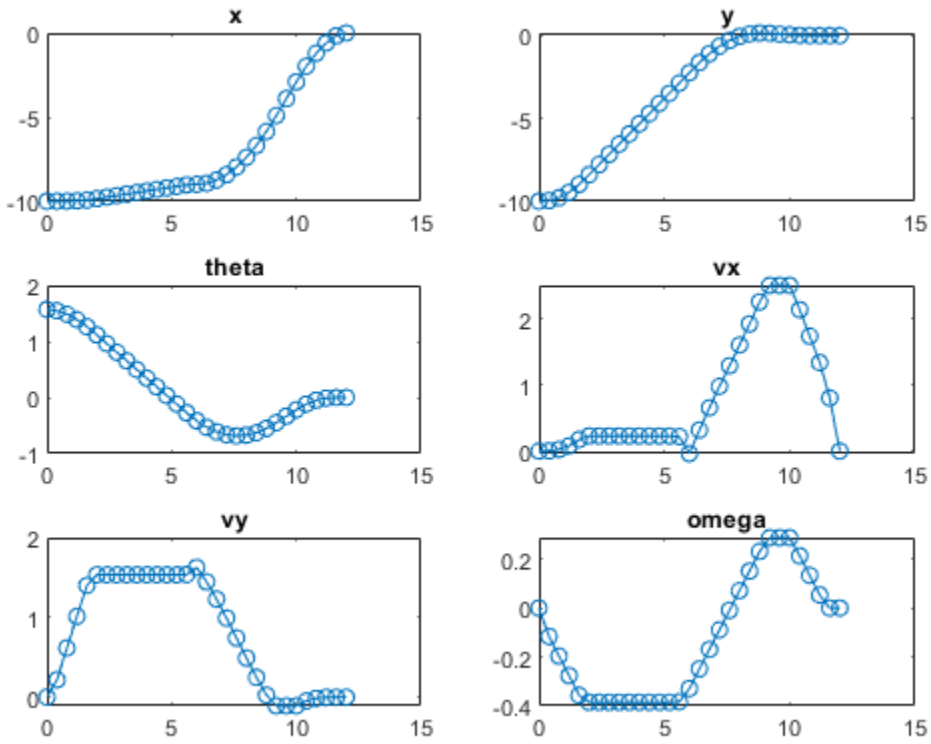
The optimal state and MV trajectories can be found by calling the `nlmpcmove` command once, given the current state `x0` and last MV `u0`. The optimal cost and trajectories are returned as part of the `info` output argument.

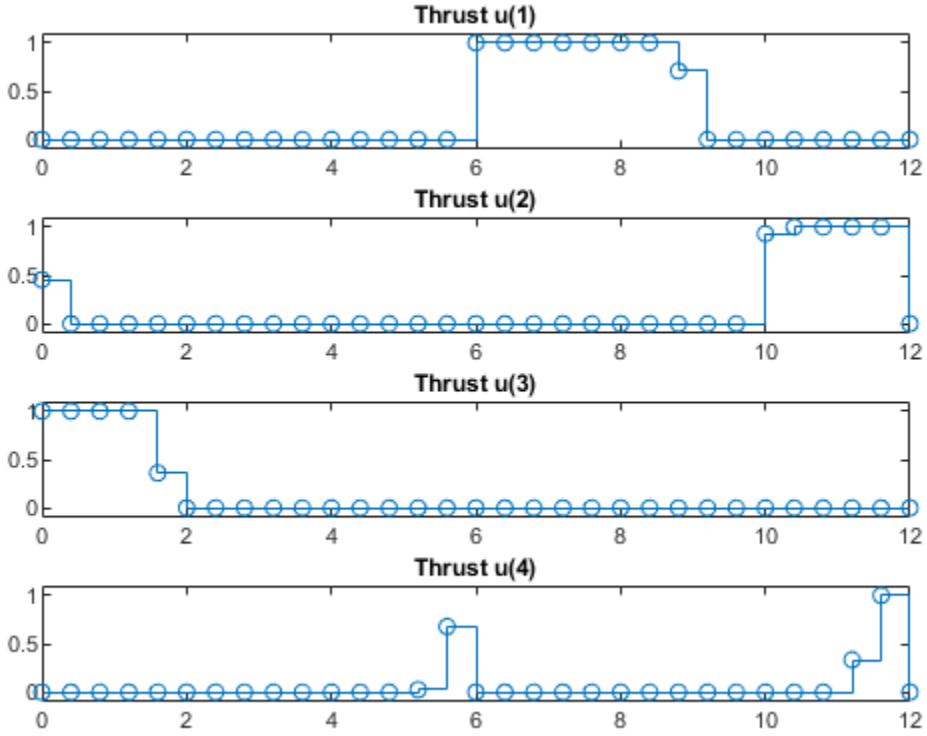
```
[~,~,info] = nlmpcmove(nlobj,x0,u0);
```

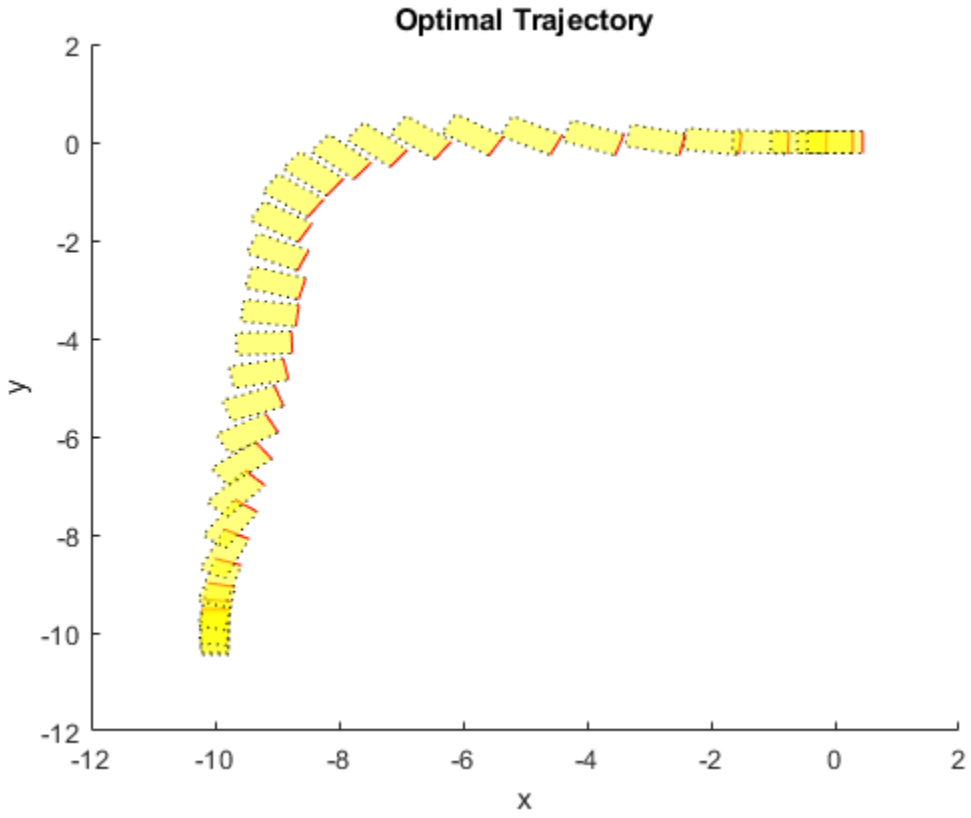
Plot the optimal trajectory. The optimal cost is 7.8.

```
FlyingRobotPlotPlanning(info,Ts);
```

```
Optimal fuel consumption =   7.797825
```


x
y
theta
vx
vy
omega


Thrust u(1)
Thrust u(2)
Thrust u(3)
Thrust u(4)
0
0.5
1
0
2
4
6
8
10
12



The first plot shows the optimal trajectory of the six robot states during the maneuver. The second plot shows the corresponding optimal MV profiles for the four thrusts. The third plot shows the X-Y position trajectory of the robot, moving from `[-10 -10 pi/2]` to `[0 0 0]`.

Feedback Control for Path Following

After the optimal trajectory is found, a feedback controller is required to move the robot along the path. In theory, you can apply the optimal MV profile directly to the thrusters to implement feed-forward control. However, in practice, a feedback controller is needed to reject disturbances and compensate for modeling errors.

You can use different feedback control techniques for tracking. In this example, you use a generic nonlinear MPC controller to move the robot to the final location. In this path tracking problem, you track references for all six states (the number of outputs equals the number of states).

```
ny = 6;
nlobj_tracking = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Use the same state function and its Jacobian function.

```
nlobj_tracking.Model.StateFcn = nlobj.Model.StateFcn;
nlobj_tracking.Jacobian.StateFcn = nlobj.Model.StateJacFcn;
```

For tracking control applications, reduce the computational effort by specifying shorter prediction horizon (no need to look far into the future) and control horizon (for example, free moves are allocated at the first few predictrion steps).

```
nlobj_tracking.Ts = Ts;
nlobj_tracking.PredictionHorizon = 10;
nlobj_tracking.ControlHorizon = 4;
```

The default cost function in nonlinear MPC is a standard quadratic cost function suitable for reference tracking and disturbance rejection. For tracking, tracking error has higher priority (larger penalty weights on outputs) than control efforts (smaller penalty weights on MV rates).

```
nlobj_tracking.Weights.ManipulatedVariablesRate = 0.2*ones(1,nu);
nlobj_tracking.Weights.OutputVariables = 5*ones(1,nx);
```

Set the same bounds for the thruster inputs.

```
for ct = 1:nu
    nlobj_tracking.MV(ct).Min = 0;
    nlobj_tracking.MV(ct).Max = 1;
end
```

Also, to reduce fuel consumption, it is clear that `u1` and `u2` cannot be positive at any time during the operation. Therefore, implement equality constraints such that `u(1)*u(2)` must be `0` for all prediction steps. Apply similar constraints for `u3` and `u4`.

```
nlobj_tracking.Optimization.CustomEqConFcn = ...
    @(X,U,data) [U(1:end-1,1).*U(1:end-1,2); U(1:end-1,3).*U(1:end-1,4)];
```

Validate your prediction model and custom functions, and their Jacobians.

```
validateFcns(nlobj_tracking,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Nonlinear State Estimation

In this example, only the three position states (x, y and angle) are measured. The velocity states are unmeasured and must be estimated. Use an extended Kalman filter (EKF) from Control System Toolbox™ for nonlinear state estimation.

Because an EKF requires a discrete-time model, you use the trapezoidal rule to transition from x(k) to x(k+1), which requires the solution of `nx` nonlinear algebraic equations. For more information, open `FlyingRobotStateFcnDiscreteTime.m`.

```
DStateFcn = @(xk,uk,Ts) FlyingRobotStateFcnDiscreteTime(xk,uk,Ts);
```

Measurement can help the EKF correct its state estimation. Only the first three states are measured.

```
DMeasFcn = @(xk) xk(1:3);
```

Create the EKF, and indicate that the measurements have little noise.

```
EKF = extendedKalmanFilter(DStateFcn,DMeasFcn,x0);
EKF.MeasurementNoise = 0.01;
```

Closed-Loop Simulation of Tracking Control

Simulate the system for 32 steps with correct initial conditions.

```
Tsteps = 32;
xHistory = x0';
uHistory = [];
lastMV = zeros(nu,1);
```

The reference signals are the optimal state trajectories computed at the planning stage. When passing these trajectories to the nonlinear MPC controller, the current and future trajectory is available for previewing.

```
Xopt = info.Xopt;
Xref = [Xopt(2:p+1,:);repmat(Xopt(end,:),Tsteps-p,1)];
```

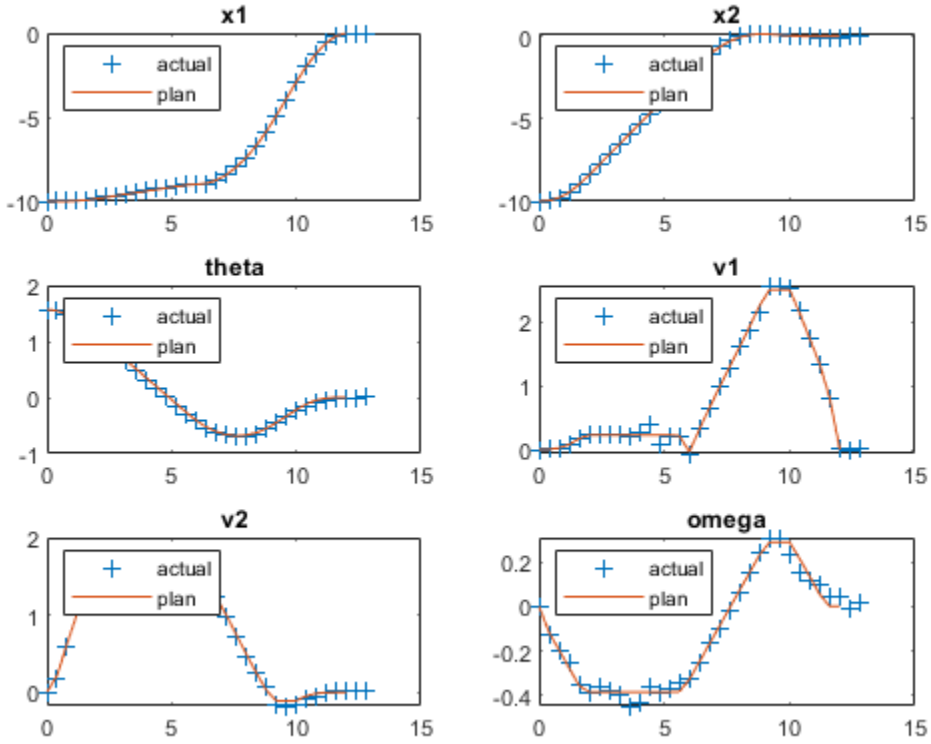
Use `nlmpcmove` and `nlmpcmoveopt` command for closed-loop simulation.

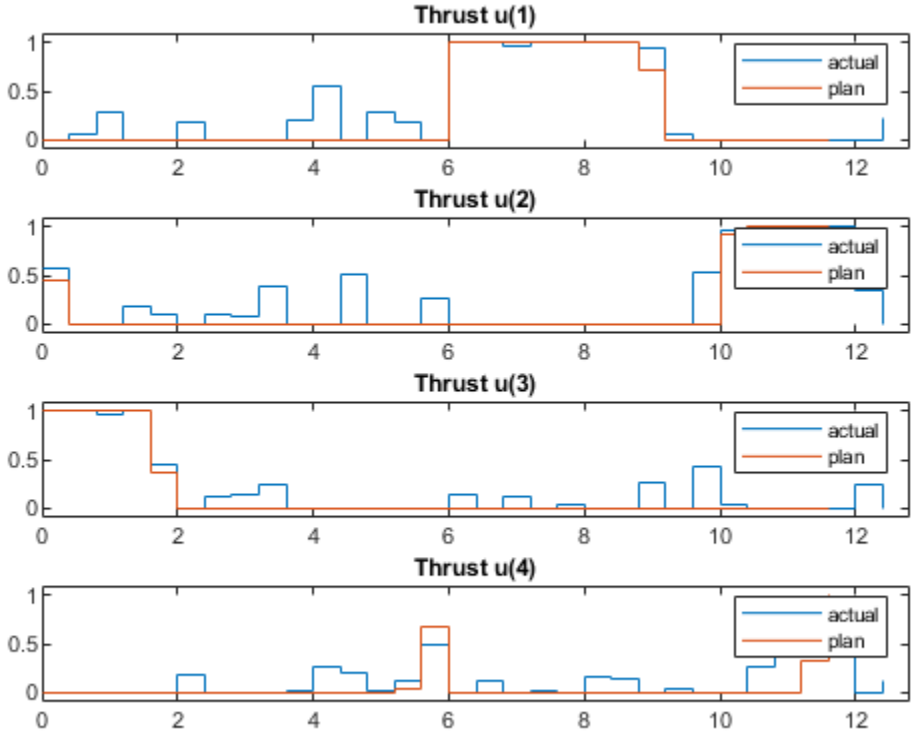
```
hbar = waitbar(0,'Simulation Progress');
options = nlmpcmoveopt;
for k = 1:Tsteps
    % Obtain plant output measurements with sensor noise.
    yk = xHistory(k,1:3)' + randn*0.01;
    % Correct state estimation based on the measurements.
    xk = correct(EKF, yk);
    % Compute the control moves with reference previewing.
    [uk,options] = nlmpcmove(nlobj_tracking,xk,lastMV,Xref(k:min(k+9,Tsteps),:),[],options);
    % Predict the state for the next step.
    predict(EKF,uk,Ts);
    % Store the control move and update the last MV for the next step.
    uHistory(k,:) = uk';
    lastMV = uk;
    % Update the real plant states for the next step by solving the
    % continuous-time ODEs based on current states xk and input uk.
    ODEFUN = @(t,xk) FlyingRobotStateFcn(xk,uk);
    [TOUT,YOUT] = ode45(ODEFUN,[0 Ts], xHistory(k,:)');
    % Store the state values.
    xHistory(k+1,:) = YOUT(end,:);
    % Update the status bar.
    waitbar(k/Tsteps, hbar);
end
close(hbar)
```

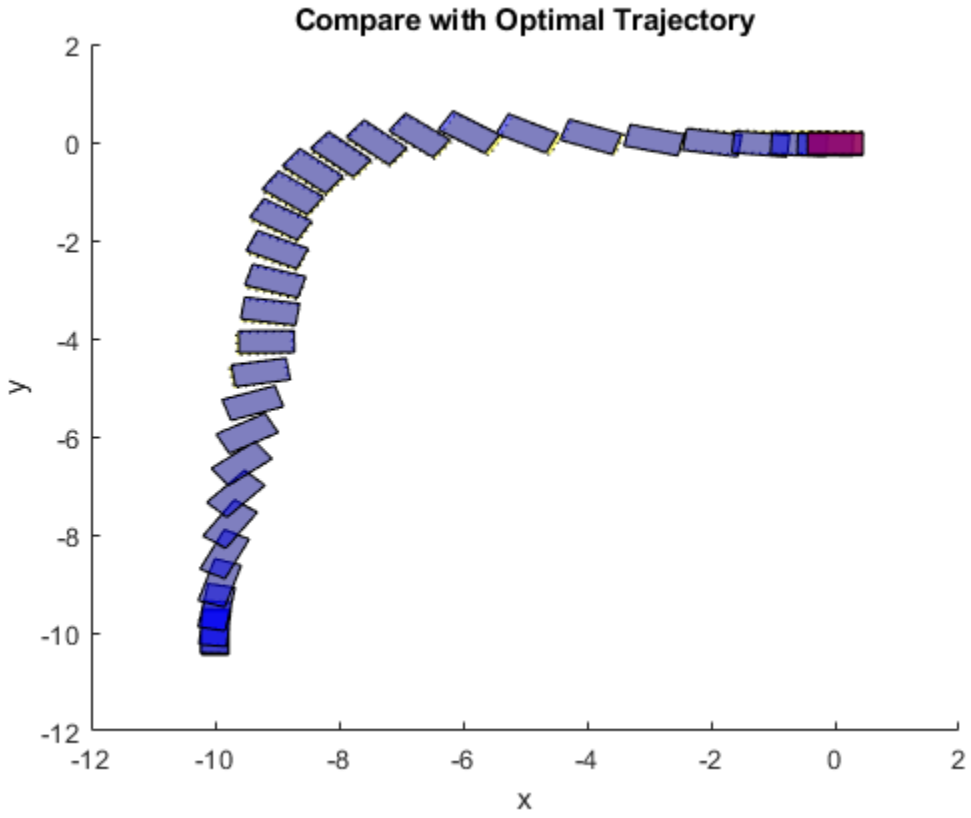
Compare the planned and actual closed-loop trajectories.

```
FlyingRobotPlotTracking(info,Ts,p,Tsteps,xHistory,uHistory);
```

```
Actual fuel consumption =  10.771121
```


x1
actual
plan
x2
actual
plan
theta
actual
plan
v1
actual
plan
v2
actual
plan
omega
actual
plan


Thrust u(1)
actual
plan
Thrust u(2)
actual
plan
Thrust u(3)
actual
plan
Thrust u(4)
actual
plan
0
0.5
1
0
2
4
6
8
10
12



The nonlinear MPC feedback controller successfully moves the robot (blue blocks), following the optimal trajectory (yellow blocks), and parks it at the final location (red block) in the last figure.

The actual fuel cost is higher than the planned cost. The main reason for this result is that, since we used shorter prediction and control horizons in the feedback controller, the control decision at each interval is suboptimal compared to the optimization problem used in the planning stage.

References

[1] Y. Sakawa. "Trajectory planning of a free-flying robot by using the optimal control." *Optimal Control Applications and Methods*, Vol. 20, 1999, pp. 235-248.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2

Generate Code to Plan and Execute Collision-Free Trajectories using KINOVA Gen3 Manipulator

This example shows how to generate code in order to speed up planning and execution of closed-loop collision-free robot trajectories using model predictive control (MPC). For more information on how to use MPC for motion planning of robot manipulators, see the example “Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator” (Robotics System Toolbox).

Modifications for code generation

To enable code generation, adapt the code in “Plan and Execute Collision-Free Trajectories Using KINOVA Gen3 Manipulator” (Robotics System Toolbox) following the steps below.

- Ensure all helper “Nonlinear MPC” on page 9-2 functions are stored as separate program files instead of “Anonymous Functions”. This example requires the following helper files.

- Custom cost function: `nlmpcCostFunctionKINOVACodeGen.m`

- Custom inequality constraints: `nlmpcIneqConFunctionKINOVACodeGen.m`

- State function: `nlmpcModelKINOVACodeGen.m`

- Output function: `nlmpcOutputKINOVACodeGen.m`

- Jacobian of custom cost function: `nlmpcJacobianCostKINOVACodeGen.m`

- Jacobian of custom inequality constraints: `nlmpcJacobianConstraintKINOVACodeGen.m`

- Jacobian of state function: `nlmpcJacobianModelKINOVACodeGen.m`

- Jacobian of output function: `nlmpcJacobianOutputKINOVACodeGen.m`

- Use the `Parameters` property of the `nlmpcmoveopt` object (and, when a mex file, the `onlineData` structure) to pass to the controller any parameter that might change at runtime (that is, at each time step) or between consecutive runs. For example, to enable the update of obstacles poses according to new sensor readings at each time step, the values of the current obstacle poses (`posesNow`) are added to the `params` cell variable. Similarly, to update the target robot pose online, the target pose (`poseFinal`) is added to the `Parameters` list. All the parameters must be numerical values. At each time step, the updated value of `params` is then copied into the `Parameters` property of either the `nlmpcmoveopt` object (if `nlmpcmove` is used) or in the `onlineData` structure (if a MEX file is used).
- Use `persistent` variables to pass in variables that are not numerical and are not externally modified at runtime. For example, the `rigidBodyTree` (Robotics System Toolbox) object `robot` and the cell array of `collisionMesh` (Robotics System Toolbox) objects `world` are created and saved as persistent variables the first time a helper function is executed using the following code.

```
persistent robot world
if isempty(robot)
    robot = loadrobot('kinovaGen3', 'DataFormat', 'column');
    [world, ~] = helperCreateObstaclesKINOVA(posesNow);
end
```

- Ensure all the functions used in helper files support code generation. For a list of built-in MATLAB functions supported for code generation, see this page. To test if a custom function supports code

generation, use `coder.screener` (Simulink). For example, try running `coder.screener('nlmpcIneqConFunctionKINOVA')`. This example requires that `loadrobot` (Robotics System Toolbox) and `checkCollision` (Robotics System Toolbox) support code generation.

- Use `getCodeGenerationData` and `buildMEX` to generate code for the `nlmpc` object with the specified properties. Replace `nlmpcmove` in the original example with the generated MEX file to compute optimal control actions.

Robot Description and Poses

Load the KINOVA Gen3 rigid body tree (RBT) model.

```
robot = loadrobot('kinovaGen3', 'DataFormat', 'column');
```

Get the number of joints.

```
numJoints = numel(homeConfiguration(robot));
```

Specify the robot frame where the end-effector is attached.

```
endEffector = "EndEffector_Link";
```

Specify initial and desired end-effector poses. Use inverse kinematics to solve for the initial robot configuration given a desired pose.

```
% Initial end-effector pose
taskInit = trvec2tform([[0.4 0 0.2]])*axang2tform([0 1 0 pi]);

% Compute current robot joint configuration using inverse kinematics
ik = inverseKinematics('RigidBodyTree', robot);
ik.SolverParameters.AllowRandomRestart = false;
weights = [1 1 1 1 1 1];
currentRobotJConfig = ik(endEffector, taskInit, weights, robot.homeConfiguration);
currentRobotJConfig = wrapToPi(currentRobotJConfig);

% Final (desired) end-effector pose
taskFinal = trvec2tform([0.35 0.55 0.35])*axang2tform([0 1 0 pi]);
anglesFinal = rotm2eul(taskFinal(1:3,1:3),'XYZ');
poseFinal = [taskFinal(1:3,4);anglesFinal']; % 6x1 vector for final pose: [x, y, z, phi, theta, p
```

Collision Meshes and Obstacles

To check for and avoid collisions during control, you must setup a collision `world` as a set of collision objects. This example uses `collisionSphere` (Robotics System Toolbox) objects as obstacles to avoid. To plan using static instead of moving objects, set `isMovingObst` to `false`.

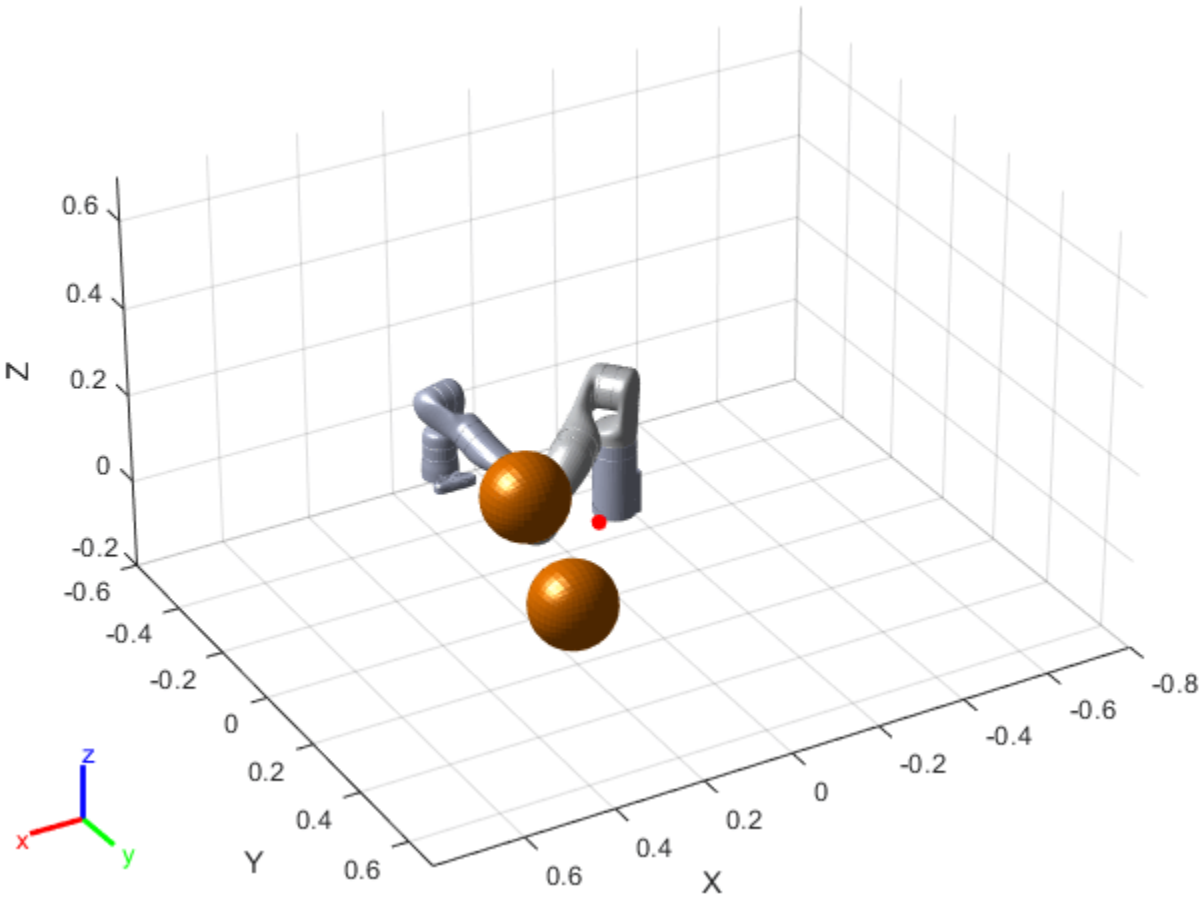
```
isMovingObst = true;
```

The obstacle sizes and locations are initialized in the `helperCreateMovingObstaclesKINOVACodeGen` helper function. To add more static obstacles, add collision objects in the `world` array.

```
if isMovingObst == true
    helperCreateMovingObstaclesKINOVACodeGen
else
    posesNow = [0.4 0.4 0.25 ; 0.3 0.3 0.4];
end
[world, numObstacles] = helperCreateObstaclesKINOVACodeGen(posesNow);
```

Visualize the robot at the initial configuration. You should see the obstacles in the environment as well.

```
x0 = [currentRobotJConfig', zeros(1,numJoints)];
helperInitialVisualizerKINOVACodeGen;
```



Specify a safety distance away from the obstacles. This value is used in the inequality constraint function of the nonlinear MPC controller.

```
safetyDistance = 0.01;
```

Design Nonlinear Model Predictive Controller

You can design the nonlinear model predictive controller using the `helperDesignNLMPCobjKINOVACodeGen` helper file, which creates an `nlmpc` controller object. To view the file, at the MATLAB command line, enter `edit helperDesignNLMPCobjKINOVACodeGen`.

```
helperDesignNLMPCobjKINOVACodeGen;
```

Closed-Loop Trajectory Planning

Simulate the robot for a maximum of 50 steps with correct initial conditions.

Initialize Simulation Parameters

```
maxIters = 50;
u0 = zeros(1,numJoints);
mv = u0';
time = 0;
goalReached = false;
```

Generate Code for `nlmpc` Object

```
useMex = true;
buildMex = true;
if useMex
    [coredata, onlinedata] = getCodeGenerationData(nlobj,x0',u0',paras);
    if buildMex
        mexfcn = buildMEX(nlobj,'kinovaMex',coredata,onlinedata);
    end
end
```

```
Generating MEX function "kinovaMex" from nonlinear MPC to speed up simulation.
Code generation successful.

MEX function "kinovaMex" successfully generated.
```

Initialize Data Array for Control

```
positions = zeros(numJoints,maxIters);
positions(:,1) = x0(1:numJoints)';

velocities = zeros(numJoints,maxIters);
velocities(:,1) = x0(numJoints+1:end)';

accelerations = zeros(numJoints,maxIters);
accelerations(:,1) = u0';

timestamp = zeros(1,maxIters);
timestamp(:,1) = time;
```

Generate Trajectory

Use the generated MEX file `kinovaMex` instead of the `nlmpcmove` function to speed up closed-loop trajectory generation. Specify the trajectory generation online options using the argument `onlineData`.

Each iteration calculates the position, velocity, and acceleration of the joints to avoid obstacles as they move towards the goal. The `helperCheckGoalReachedKINOVACodeGen` script checks if the robot has reached the goal. The `helperUpdateMovingObstaclesKINOVACodeGen` script moves the obstacle locations based on the time step.

```
options = nlmpcmoveopt;

for timestep=1:maxIters
    disp(['Calculating control at timestep ', num2str(timestep)]);
    % Optimize next trajectory point
    if ~useMex
        options.Parameters = paras;
        [mv,options,info] = nlmpcmove(nlobj,x0,mv,[],[], options);
    else
```

```
        onlinedata.Parameters = paras;
        [mv,onlinedata,info] = kinovaMex(x0',mv,onlinedata);
    end
    if info.ExitFlag < 0
        disp('Failed to compute a feasible trajectory. Aborting...')
        break;
    end
    % Update states and time for next iteration
    x0 = info.Xopt(2,:);
    time = time + nlobj.Ts;
    % Store trajectory points
    positions(:,timestep+1) = x0(1:numJoints)';
    velocities(:,timestep+1) = x0(numJoints+1:end)';
    accelerations(:,timestep+1) = info.MVopt(2,:)';
    timestamp(timestep+1) = time;
    % Check if goal is achieved
    helperCheckGoalReachedKINOVACodeGen;
    if goalReached
        break;
    end
    % Update obstacle pose if it is moving
    if isMovingObst
        helperUpdateMovingObstaclesKINOVACodeGen;
    end
end
```

```
Calculating control at timestep 1
Calculating control at timestep 2
Calculating control at timestep 3
Calculating control at timestep 4
Calculating control at timestep 5
Calculating control at timestep 6
Calculating control at timestep 7
Calculating control at timestep 8

Target configuration reached.
```

Execute Planned Trajectory Using Low-Fidelity Robot Control and Simulation

Trim the trajectory vectors based on the time steps calculated from the plan.

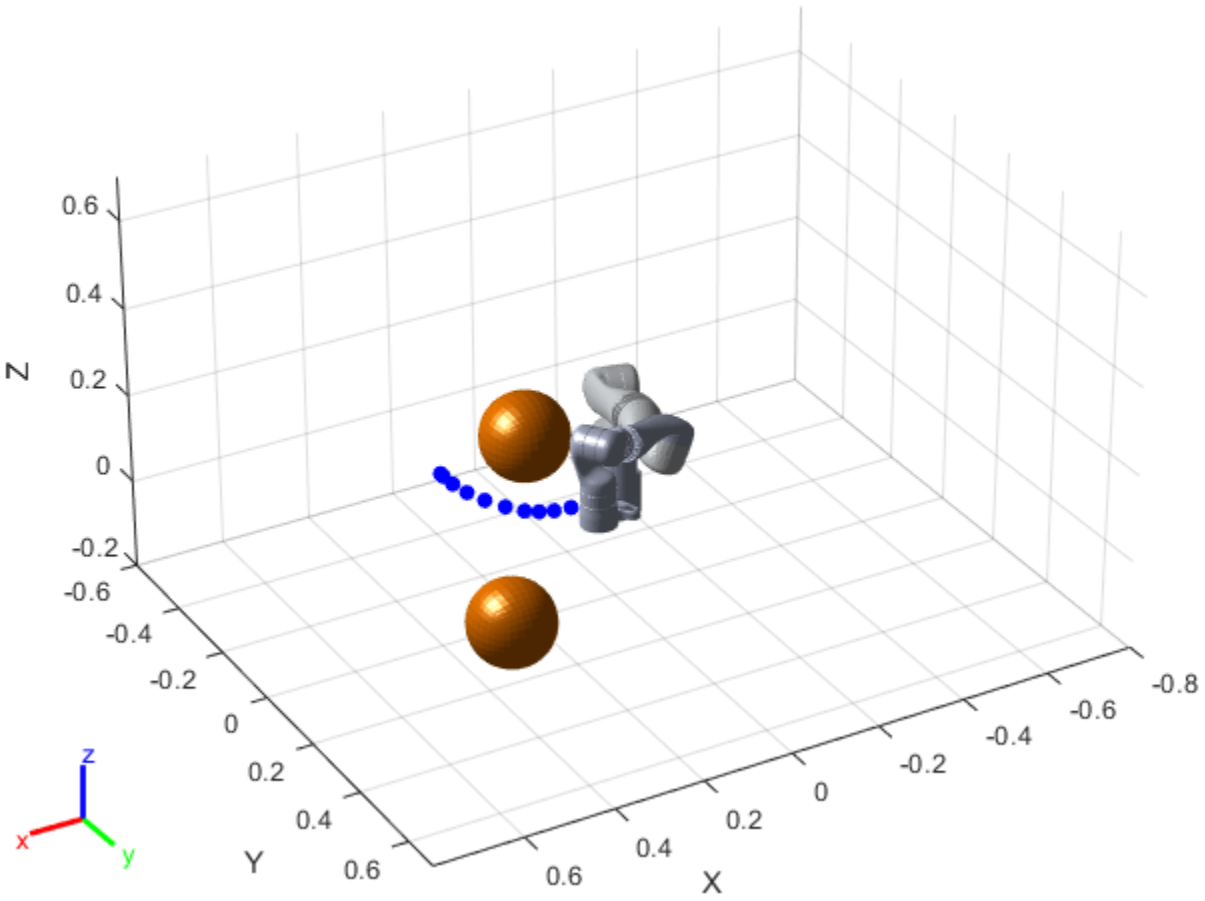
```
tFinal = timestep+1;
positions = positions(:,1:tFinal);
velocities = velocities(:,1:tFinal);
accelerations = accelerations(:,1:tFinal);
timestamp = timestamp(:,1:tFinal);

visTimeStep = 0.2;
```

Use a `jointSpaceMotionModel` object to track the trajectory with computed-torque control. The `helperTimeBasedStateInputsKINOVA` function generates the derivative inputs for the `ode15s` ODE solver to simulate the robot movement along the computed trajectory.

```
motionModel = jointSpaceMotionModel('RigidBodyTree', robot);

% Control robot to target trajectory points in simulation using low-fidelity model
initState = [positions(:,1);velocities(:,1)];
targetStates = [positions;velocities;accelerations]';
[t,robotStates] = ode15s(@(t,state) helperTimeBasedStateInputsKINOVA(motionModel, timestamp, tar
```

Visualize the robot motion.

```
helperFinalVisualizerKINOVACodeGen;
```



As expected, the robot successfully moves along the planned trajectory and avoids the obstacles.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2

Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control

This example uses a nonlinear model predictive controller object and block to achieve swing-up and balancing control of an inverted pendulum on a cart.

This example requires Optimization Toolbox™ software to provide the default nonlinear programming solver for nonlinear MPC to compute optimal control moves at each control interval.

Pendulum/Cart Assembly

The plant for this example is a pendulum/cart assembly, where *z* is the cart position and *theta* is the pendulum angle. The manipulated variable for this system is a variable force *F* acting on the cart. The range of the force is between `-100` and `100`. An impulsive disturbance *dF* can push the pendulum as well.

Control Objectives

Assume the following initial conditions for the pendulum/cart assembly.

- The cart is stationary at $z = 0$.
- The pendulum is in a downward equilibrium position where *theta* `= -pi`.

The control objectives are:

- Swing-up control: Initially swing the pendulum up to an inverted equilibrium position where $z = 0$ and *theta* = 0.
- Cart position reference tracking: Move the cart to a new position with a step setpoint change, keeping the pendulum inverted.
- Pendulum balancing: When an impulse disturbance of magnitude of 2 is applied to the inverted pendulum, keep the pendulum balanced, and return the cart to its original position.

The downward equilibrium position is stable, and the inverted equilibrium position is unstable, which makes swing-up control more challenging for a single linear controller, which nonlinear MPC handles easily.

Control Structure

In this example, the nonlinear MPC controller has the following I/O configuration.

- One manipulated variable: Variable force (*F*)
- Two measured outputs: Cart position (*z*) and pendulum angle (*theta*)

Two other states, cart velocity (*zdot*) and pendulum angular velocity (*thetadot*) are not measurable.

While the setpoint of the cart position, *z*, can vary, the setpoint of the pendulum angle, *theta*, is always 0 (inverted equilibrium position).

Create Nonlinear MPC Controller

Create a nonlinear MPC controller with the proper dimensions using an `nlmpc` object. In this example, the prediction model has 4 states, 2 outputs, and 1 input (MV).

```
nx = 4;
ny = 2;
nu = 1;
nlobj = nlmpc(nx, ny, nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

The prediction model has a sample time of `0.1` seconds, which is the same as the controller sample time.

```
Ts = 0.1;
nlobj.Ts = Ts;
```

Set the prediction horizon to `10`, which is long enough to capture major dynamics in the plant but not so long that it hurts computational efficiency.

```
nlobj.PredictionHorizon = 10;
```

Set the control horizon to 5, which is long enough to give the controller enough degrees of freedom to handle the unstable mode without introducing excessive decision variables.

```
nlobj.ControlHorizon = 5;
```

Specify Nonlinear Plant Model

The major benefit of nonlinear model predictive control is that it uses a nonlinear dynamic model to predict plant behavior in the future across a wide range of operating conditions.

This nonlinear model is usually a first principle model consisting of a set of differential and algebraic equations (DAEs). In this example, a discrete-time cart and pendulum system is defined in the `pendulumDT0` function. This function integrates the continuous-time model, `pendulumCT0`, between control intervals using a multistep forward Euler method. The same function is also used by the nonlinear state estimator.

```
nlobj.Model.StateFcn = "pendulumDT0";
```

To use a discrete-time model, set the `Model.IsContinuousTime` property of the controller to `false`.

```
nlobj.Model.IsContinuousTime = false;
```

The prediction model uses an optional parameter, `Ts`, to represent the sample time. Using this parameter means that, if you change the prediction sample time during the design, you do not have to modify the `pendulumDT0` file.

```
nlobj.Model.NumberOfParameters = 1;
```

The two plant outputs are the first and third state in the model, the cart position and pendulum angle, respectively. The corresponding output function is defined in the `pendulumOutputFcn` function.

```
nlobj.Model.OutputFcn = 'pendulumOutputFcn';
```

It is best practice to provide analytical Jacobian functions whenever possible, since they significantly improve the simulation speed. In this example, provide a Jacobian for the output function using an anonymous function.

```
nlobj.Jacobian.OutputFcn = @(x,u,Ts) [1 0 0 0; 0 0 1 0];
```

Since you do not provide Jacobian for the state function, the nonlinear MPC controller estimates the state function Jacobian during optimization using numerical perturbation. Doing so slows down simulation to some degree.

Define Cost and Constraints

Like linear MPC, nonlinear MPC solves a constrained optimization problem at each control interval. However, since the plant model is nonlinear, nonlinear MPC converts the optimal control problem into a nonlinear optimization problem with a nonlinear cost function and nonlinear constraints.

The cost function used in this example is the same standard cost function used by linear MPC, where output reference tracking and manipulated variable move suppression are enforced. Therefore, specify standard MPC tuning weights.

```
nlobj.Weights.OutputVariables = [3 3];
nlobj.Weights.ManipulatedVariablesRate = 0.1;
```

The cart position is limited to the range `-10` to `10`.

```
nlobj.OV(1).Min = -10;
nlobj.OV(1).Max = 10;
```

The force has a range between `-100` and `100`.

```
nlobj.MV.Min = -100;
nlobj.MV.Max = 100;
```

Validate Nonlinear MPC Controller

After designing a nonlinear MPC controller object, it is best practice to check the functions you defined for the prediction model, state function, output function, custom cost, and custom constraints, as well as their Jacobians. To do so, use the `validateFcns` command. This function detects any dimensional and numerical inconsistencies in these functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj,x0,u0,[],{Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Jacobian.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

State Estimation

In this example, only two plant states (cart position and pendulum angle) are measurable. Therefore, you estimate the four plant states using an extended Kalman filter. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn, @pendulumMeasurementFcn);
```

Closed-Loop Simulation in MATLAB®

Specify the initial conditions for simulations by setting the initial plant state and output values. Also, specify the initial state of the extended Kalman filter.

The initial conditions of the simulation areas follows.

- The cart is stationary at $z = 0$.
- The pendulum is in a downward equilibrium position, *theta* = `-pi`.

```
x = [0;0;-pi;0];
y = [x(1);x(3)];
EKF.State = x;
```

`mv` is the optimal control move computed at any control interval. Initialize `mv` to zero, since the force applied to the cart is zero at the beginning.

```
mv = 0;
```

In the first stage of the simulation, the pendulum swings up from a downward equilibrium position to an inverted equilibrium position. The state references for this stage are all zero.

```
yref1 = [0 0];
```

At a time of `10` seconds, the cart moves from position `0` to `5`. Set the state references for this position.

```
yref2 = [5 0];
```

Using the `nlmpcmove` command, compute optimal control moves at each control interval. This function constructs a nonlinear programming problem and solves it using the `fmincon` function from the Optimization Toolbox.

Specify the prediction model parameter using an `nlmpcmoveopt` object, and pass this object to `nlmpcmove`.

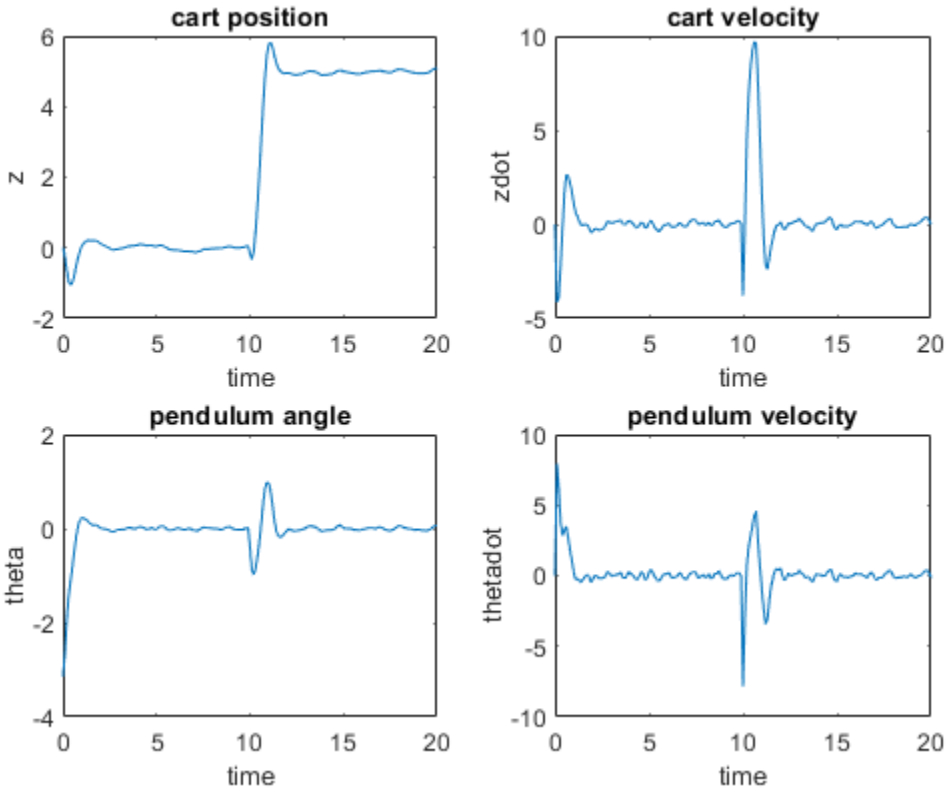
```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
```

Run the simulation for `20` seconds.

```
Duration = 20;
hbar = waitbar(0,'Simulation Progress');
xHistory = x;
for ct = 1:(20/Ts)
    % Set references
    if ct*Ts<10
        yref = yref1;
    else
        yref = yref2;
    end
    % Correct previous prediction using current measurement.
    xk = correct(EKF, y);
    % Compute optimal control moves.
    [mv,nloptions,info] = nlmpcmove(nlobj,xk,mv,yref,[],nloptions);
    % Predict prediction model states for the next iteration.
    predict(EKF, [mv; Ts]);
    % Implement first optimal control move and update plant states.
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data with some white noise.
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states for display.
    xHistory = [xHistory x]; %#ok<*AGROW>
    waitbar(ct*Ts/20,hbar);
end
close(hbar)
```

Plot the closed-loop response.

```
figure
subplot(2,2,1)
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```



The **pendulum angle** plot shows that the pendulum successfully swings up in two seconds. During the swing-up process, the cart is displaced with a peak deviation of -1, and returned to its original position around a time of 2 seconds.

The **cart position** plot shows that the cart successfully moves to $z = 5$ in two seconds. While the cart moves, the pendulum is displaced with a peak deviation of 1 radian (57 degrees) and returned to an inverted equilibrium position around a time of 12 seconds.

Closed-Loop Simulation with FORCESPRO Solver in MATLAB

You can easily use a third-party nonlinear programming solver together with the nonlinear MPC object designed using Model Predictive Control Toolbox software. For example, if you have FORCESPRO software from Embotech installed, you can use their **MPC Toolbox Plugin** to generate an efficient custom NLP solver from your `nlmpc` object and use the solver for simulation and code generation.

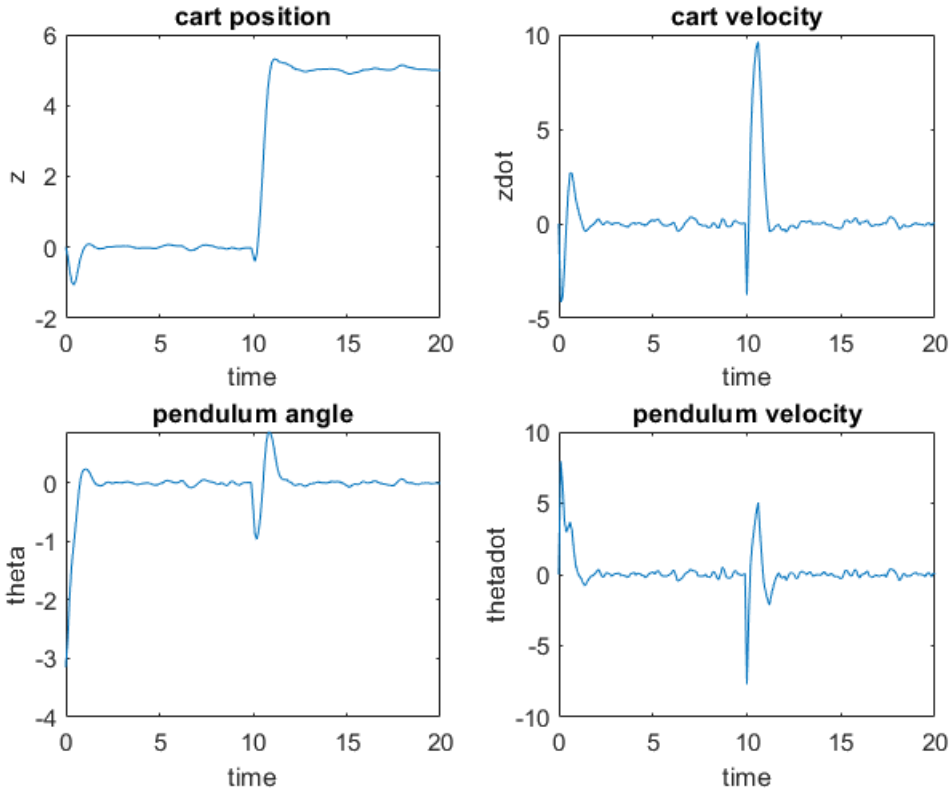
First, generate a custom solver using the `nlmpcToForces` command. You can choose using either an Interior-Point (IP) solver or a Sequential Quadratic Programming (SQP) solver using the `nlmpcToForcesOptions` command.

```
options = nlmpcToForcesOptions();
options.SolverName = 'MyIPSolver';
options.SolverType = 'InteriorPoint';
options.Parameter = Ts;
options.x0 = [0;0;-pi;0];
options.mv0 = 0;
[coredata, onlinedata] = nlmpcToForces(nlobj,options);
```

The `nlmpcToForces` function generates a custom MEX function `nlmpcmove_MyIPSolver`, which you can use to speed up closed-loop simulation.

```
x = [0;0;-pi;0];
mv = 0;
EKF.State = x;
y = [x(1);x(3)];
hbar = waitbar(0,'Simulation Progress');
xHistory = x;
for ct = 1:(20/Ts)
    % Set references
    if ct*Ts<10
        onlinedata.ref = repmat(yref1,10,1);
    else
        onlinedata.ref = repmat(yref2,10,1);
    end
    % Correct previous prediction using current measurement.
    xk = correct(EKF, y);
    % Compute optimal control moves using FORCESPRO solver.
    [mv,onlinedata,info] = nlmpcmove_MyIPSolver(xk,mv,onlinedata);
    % Predict prediction model states for the next iteration.
    predict(EKF, [mv; Ts]);
    % Implement first optimal control move and update plant states.
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data with some white noise.
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states for display.
    xHistory = [xHistory x]; %#ok<*AGROW>
    waitbar(ct*Ts/20,hbar);
end
close(hbar)
```

As expected, the closed-loop response is similar to the one obtained using `fmincon` as expected.

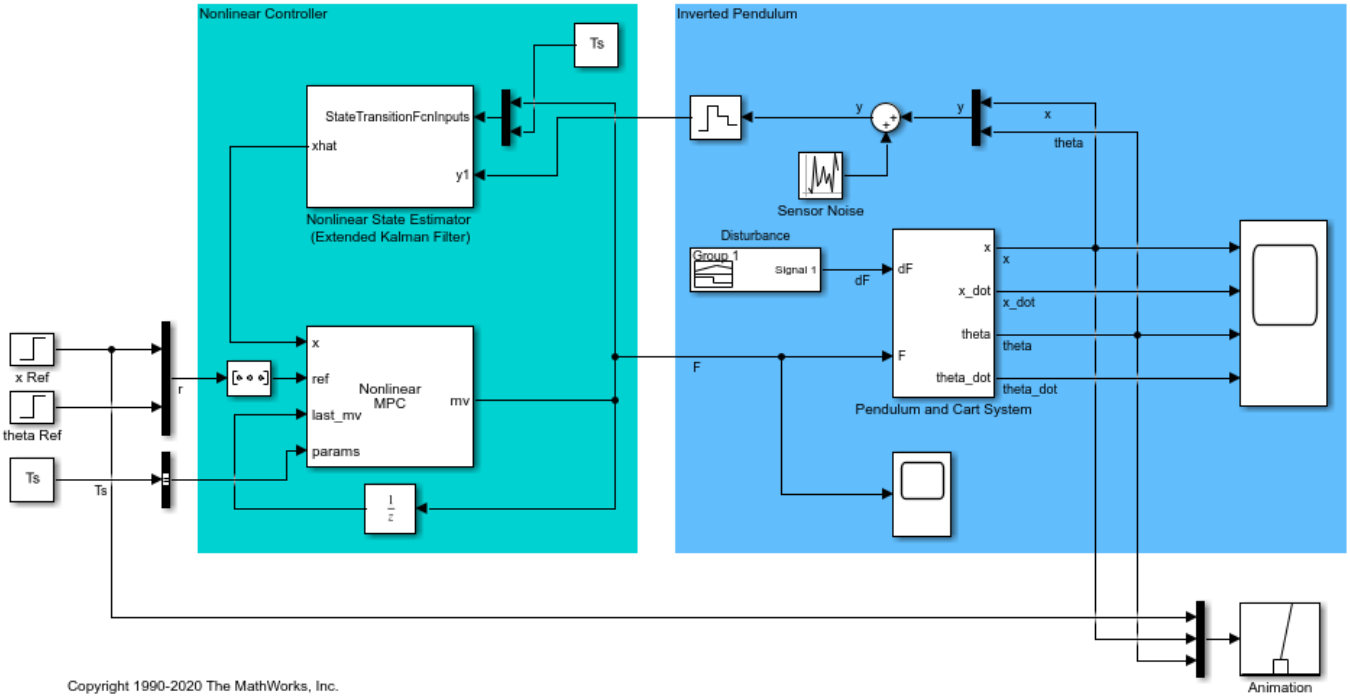


Closed-Loop Simulation in Simulink

Validate the nonlinear MPC controller with a closed-loop simulation in Simulink.

Open the Simulink model.

```
mdl = 'mpc_pendcartNMPC';
open_system(mdl)
```



In this model, the Nonlinear MPC Controller block is configured to use the previously designed controller, `nlobj`.

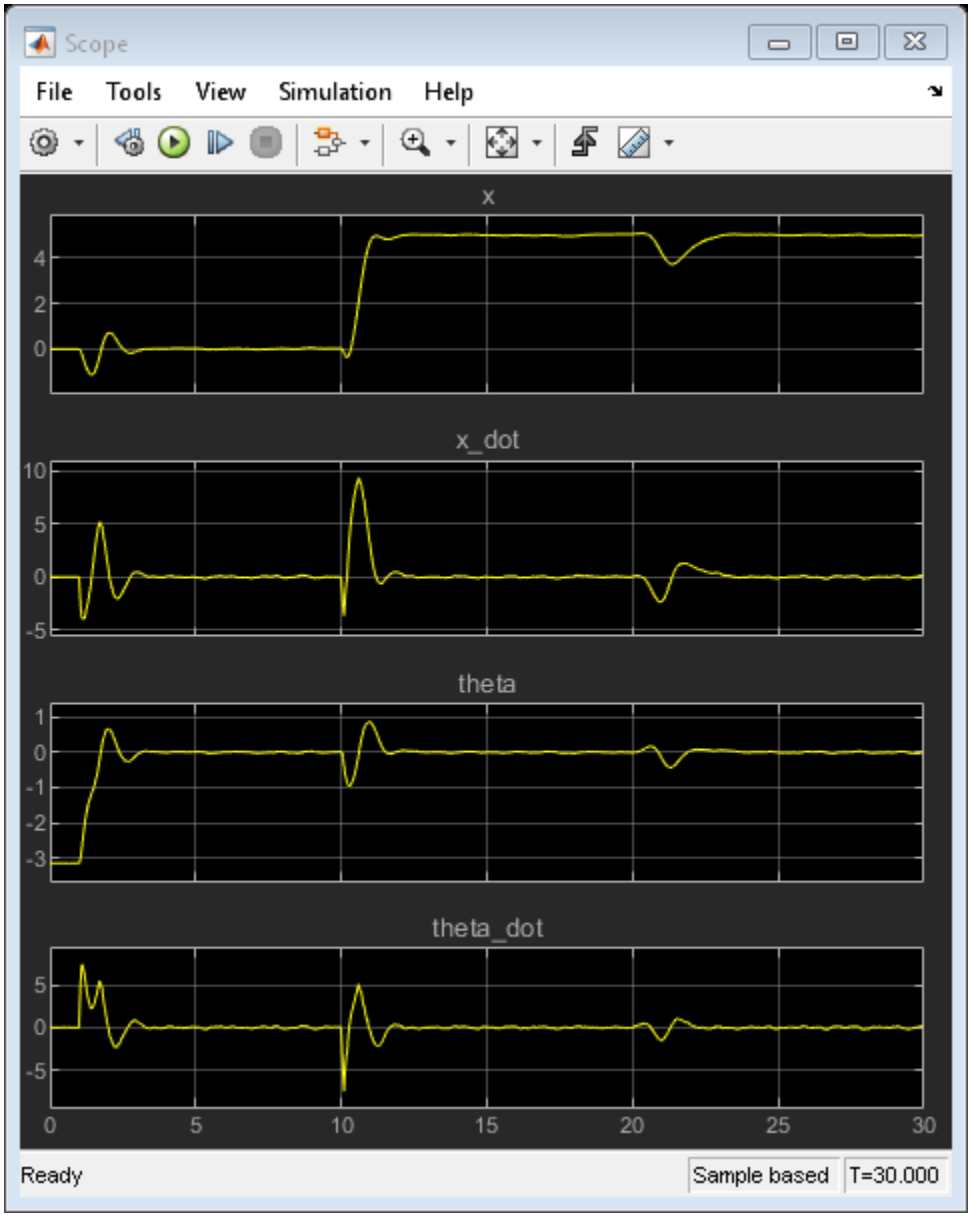
To use optional parameters in the prediction model, the model has a Simulink Bus block connected to the `params` input port of the Nonlinear MPC Controller block. To configure this bus block to use the `Ts` parameter, create a Bus object in the MATLAB workspace and configure the Bus Creator block to use this object. To do so, use the `createParameterBus` function. In this example, name the Bus object `'myBusObject'`.

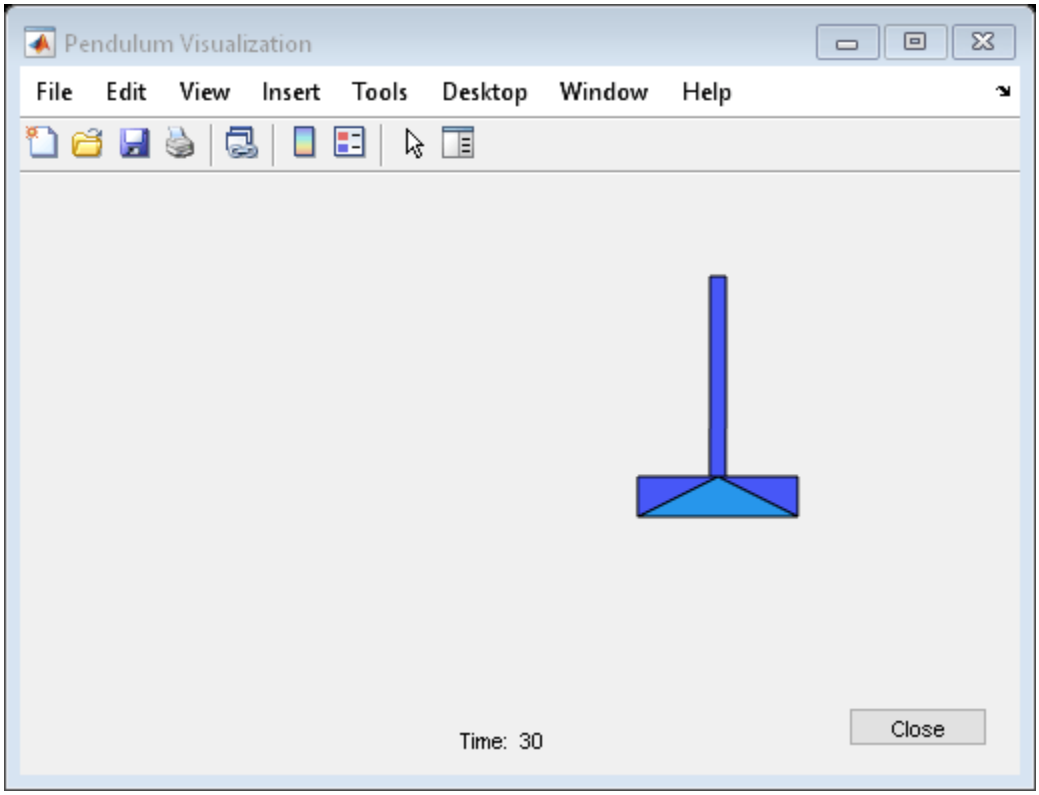
```
createParameterBus(nlobj,[mdl '/Nonlinear MPC Controller'],'myBusObject',{Ts});
```

```
A Simulink Bus object "myBusObject" created in the MATLAB Workspace, and Bus Creator block "mpc_
```

Run the simulation for `30` seconds.

```
open_system([mdl '/Scope'])
sim(mdl)
```


Scope
File
Tools
View
Simulation
Help
x
4
2
0
x_dot
10
5
0
-5
theta
1
0
-1
-2
-3
theta_dot
5
0
-5
0
5
10
15
20
25
30
Ready
Sample based
T=30.000

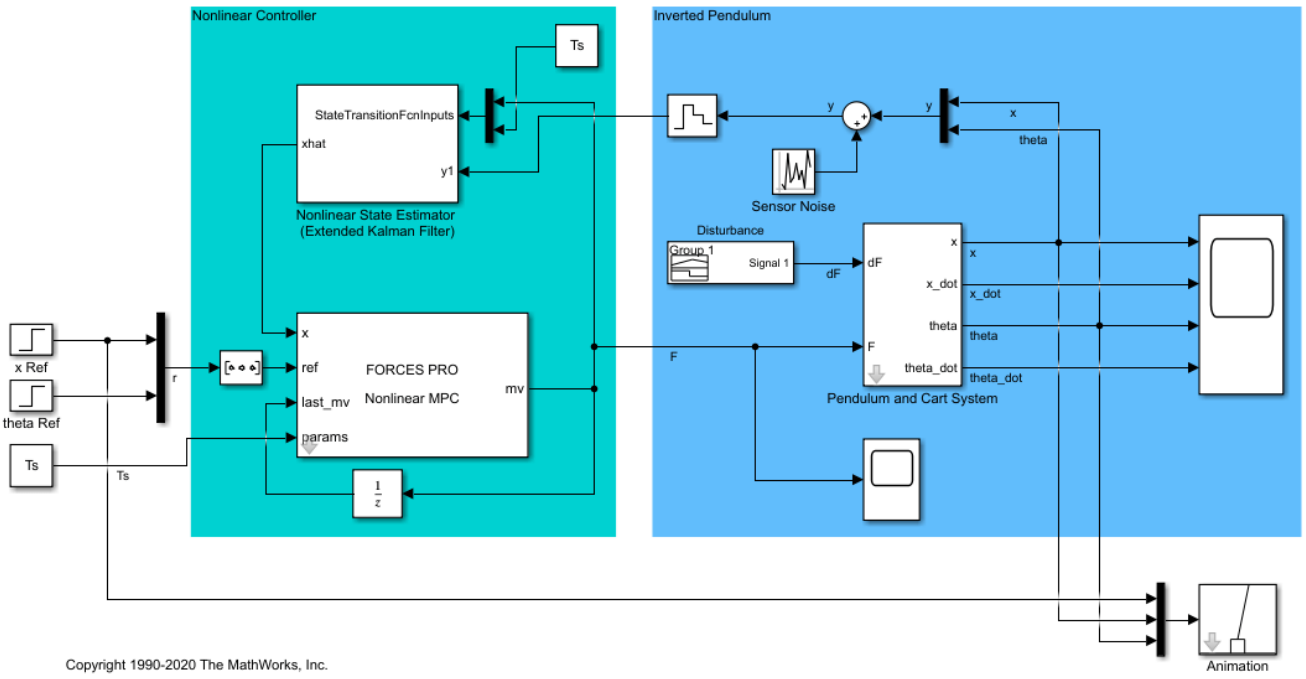


The nonlinear simulation in Simulink produces identical swing-up and cart position tracking results compared to the MATLAB simulation. Also, a push (impulse disturbance `dF`) is applied to the inverted pendulum at a time of `20` seconds. The nonlinear MPC controller successfully rejects the disturbance and returns the cart to $z = 5$ and the pendulum to an inverted equilibrium position.

Closed-Loop Simulation with FORCESPRO Solver in Simulink

You can also use the FORCES Nonlinear MPC block from the Embotech FORCESPRO software to simulate the nonlinear MPC using the generated custom NLP solver.

If you have FORCESPRO software installed, open the `mpc_pendcartFORCESPRO` model and run it to completion. The closed-loop response is similar to the one using `fmincon`.



Conclusion

This example illustrates a general workflow to design and simulate nonlinear MPC in MATLAB and Simulink using an `nlmpc` object and Nonlinear MPC Controller block, respectively. Depending on the specific nonlinear plant characteristics and control requirements, the implementation details can vary significantly. The key design challenges include:

- Choosing proper horizons, bounds, and weights
- Designing a nonlinear state estimator
- Designing a custom nonlinear cost function and constraint function
- Selecting solver options or choosing a custom NLP solver

You can use the functions and Simulink model in this example as templates for other nonlinear MPC design and simulation tasks.

Both the `nlmpcmoveCodeGeneration` command from Model Predictive Control Toolbox software and the `nlmpcmoveForces` command from FORCESPRO support code generation in MATLAB. Both the Nonlinear MPC block from Model Predictive Control Toolbox software and `FORCES Nonlinear MPC` block from FORCESPRO support code generation in Simulink.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2

Nonlinear Model Predictive Control of an Exothermic Chemical Reactor

This example shows how to use a nonlinear MPC controller to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from a low conversion rate to a high conversion rate.

About the Continuous Stirred Tank Reactor

A continuous stirred tank reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:

This system is a jacketed nonadiabatic tank reactor described extensively in [1]. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> 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 in the reactor is constant.

The inputs of the CSTR model are:

The outputs (y(t)), which are also the states of the model (x(t)), are:

$$y_1 = x_1 = T \quad \text{Reactor temperature}[K]$$
$$y_2 = x_2 = CA \quad \text{Concentration of A in reactor tank}[kgmol/m^3]$$

The control objective is to maintain the concentration of reagent A in the exit stream, , at its desired setpoint, which changes when the reactor transitions from a low conversion rate to a high conversion rate. The coolant temperature is the manipulated variable used by the controller to track the reference. The concentration of A in the feed stream and the feed stream temperature are measured disturbances.

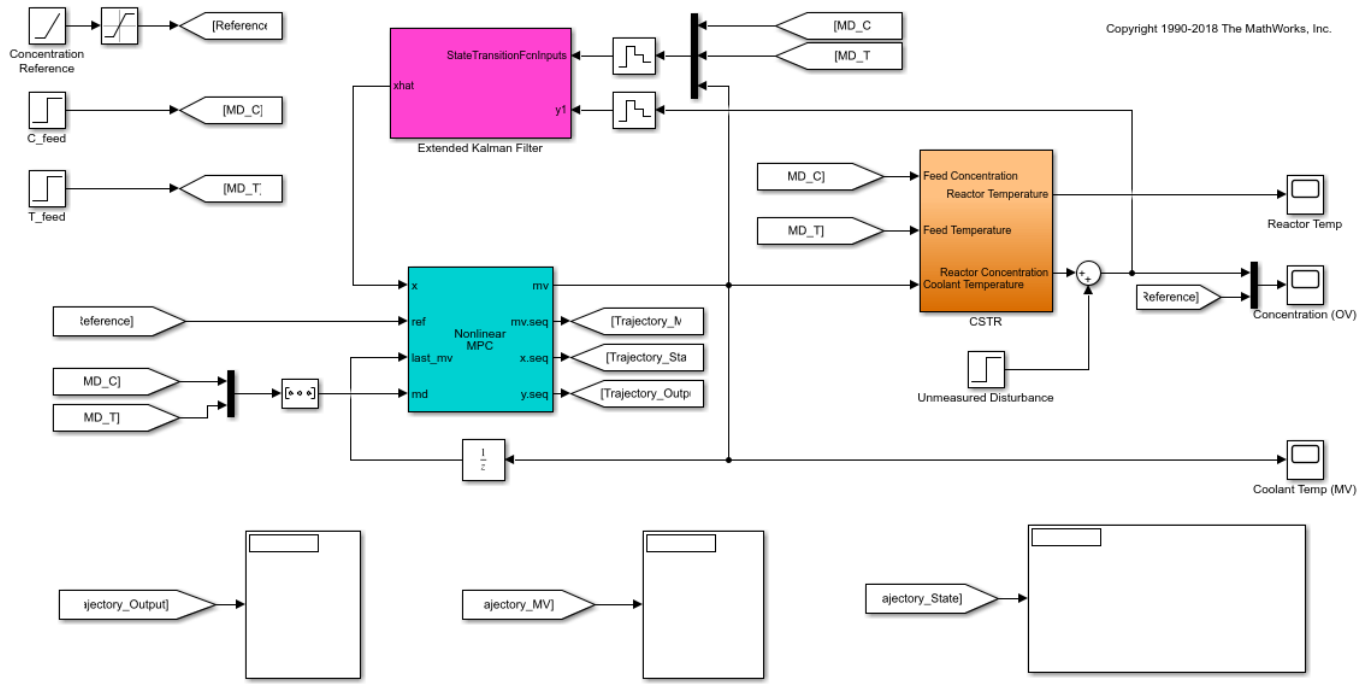
Simulink Model

To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
```

Open Simulink model.

```
mdl = 'mpc_cstr_nonlinear';
open_system(mdl)
```



Nonlinear Prediction Model

Nonlinear MPC requires a prediction model that describes the nonlinear behavior of your plant to your best knowledge. To challenge the controller, this example intentionally introduces modeling errors such that, as the temperature increases, the reaction rate of the prediction model exceeds that of the true plant. For details of the prediction model state function, see `exocstrStateFcnCT.m`.

In addition, to reject a random step-like unmeasured disturbance occurring in the concentration in the exit stream, the plant model is augmented with an integrator whose input is assumed to be zero-mean white noise. After augmentation, the prediction model has four states (T, CA and Dist) and four inputs (CA_i, T_i, T_c, WN).

Since you are only interested in controlling the concentration leaving the reactor, the output function returns a scalar value, which is the second state (CA) plus the third state (Dist). For details of the prediction model output function, see `exocstrOutputFcn.m`.

Nonlinear MPC

The control objective is to move the plant from the initial operating point with a low conversion rate (CA = 8.5698 kgmol/m^3) to the final operating point with a high conversion rate (CA = 2 kgmol/m^3). At the final steady state, the plant is open-loop unstable because cooling is no longer self-regulating. Therefore, the reactor temperature tends to *run away* from the operating point.

Create a nonlinear MPC controller object in MATLAB®. As mentioned previously, the prediction model has three states, one output, and four inputs. Among the inputs, the first two inputs (feed composition and feed temperature) are measured disturbances, the third input (coolant temperature) is the manipulated variable. The fourth input is the white noise going to the augmented integrator that represents an unmeasured output disturbance.

```
nlobj = nlmpc(3, 1,'MV',3,'MD',[1 2],'UD',4);
```

The prediction model sample time is the same as the controller sample time.

```
Ts = 0.5;
nlobj.Ts = Ts;
```

To reduce computational effort, use a short prediction horizon of 3 seconds (6 steps). Also, to increase robustness, use block moves in the control horizon.

```
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = [2 2 2];
```

Since the magnitude of the MV is of order 300 and that of the OV is order 1, scale the MV to make them compatible such that default tuning weights can be used.

```
nlobj.MV(1).ScaleFactor = 300;
```

Constrain the coolant temperature adjustment rate, which can only increase or decrease 5 degrees between two successive intervals.

```
nlobj.MV(1).RateMin = -5;
nlobj.MV(1).RateMax = 5;
```

It is good practice to scale the state to be of unit order. Doing so has no effect on the control strategy, but it can improve numerical behavior.

```
nlobj.States(1).ScaleFactor = 300;
nlobj.States(2).ScaleFactor = 10;
```

Specify the nonlinear state and output functions.

```
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

It is best practice to test your prediction model and any other custom functions before using them in a simulation. To do so, use the `validateFcns` command. In this case, use the initial operating point as the nominal condition for testing, setting the unmeasured disturbance state to `0`.

```
x0 = [311.2639; 8.5698; 0];
u0 = [10; 298.15; 298.15];
validateFcns(nlobj,x0,u0(3),u0(1:2)');
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Nonlinear State Estimation

The nonlinear MPC controller needs an estimate of three states (including the unmeasured disturbance state) at every sample time. To provide this estimate, use an Extended Kalman Filter (EKF) block. This block uses the same model as the nonlinear MPC controller except that the model is discrete-time. For details, see `exocstrStateFcnDT.m`.

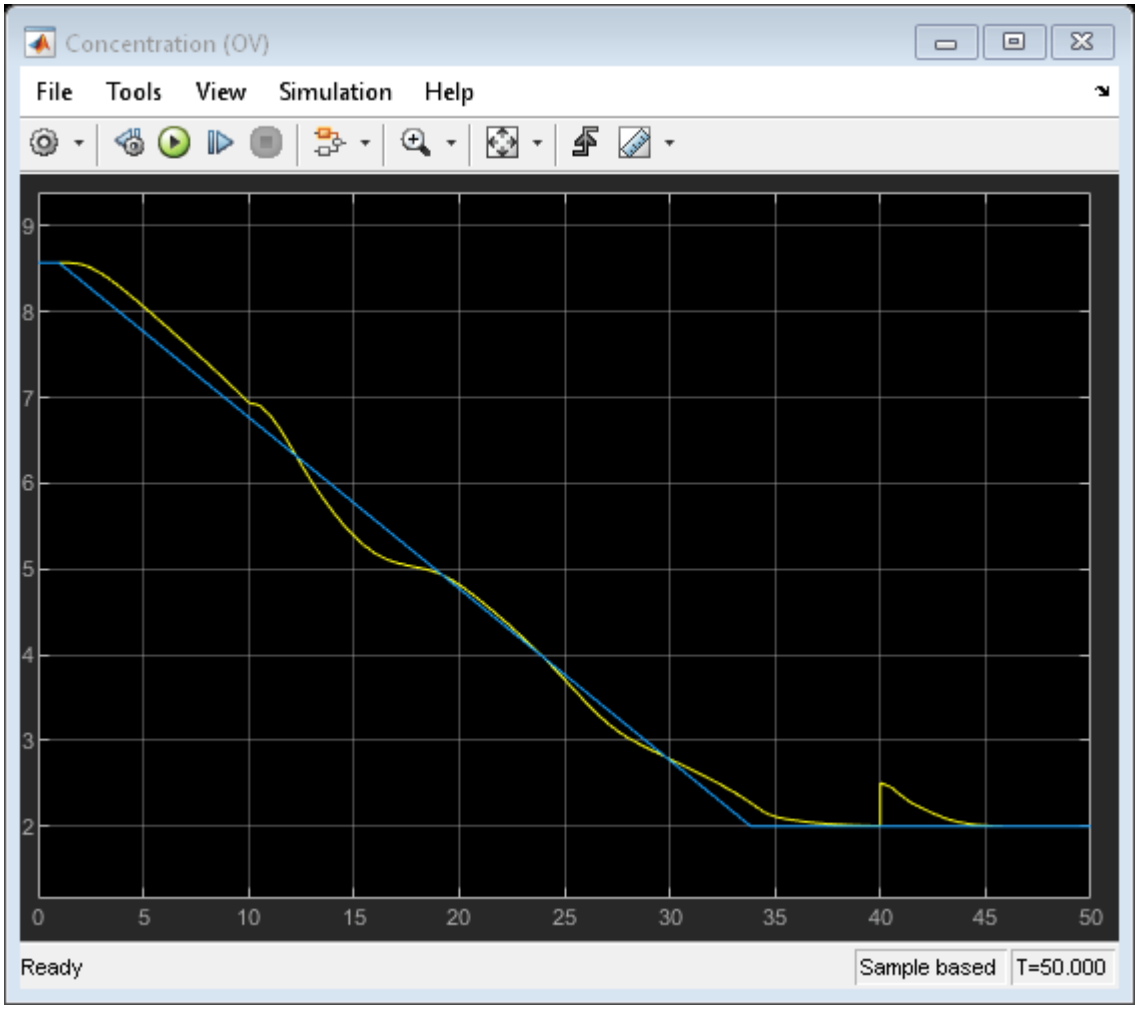
EKF measures the current concentration and uses it to correct the prediction from the previous interval. In this example, assume that the measurements are relatively accurate and use small covariance in the Extended Kalman Filter block.

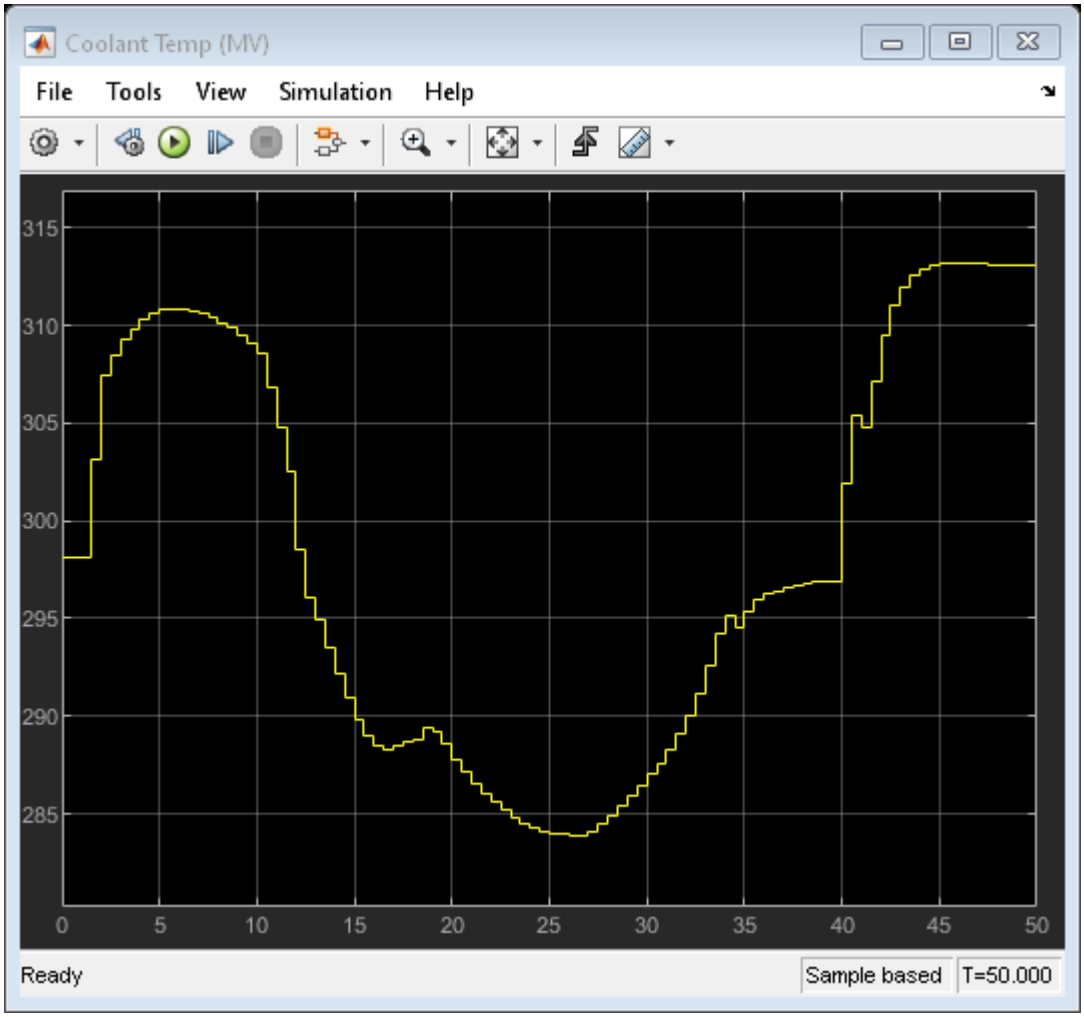
Closed-Loop Simulation

In the simulation, ramp up the target concentration rather than making an abrupt step change. You can use a much faster ramp rate because the nonlinear prediction model is used.

During the operating point transition, step changes in the two measured disturbance channels occur at `10` and `20` seconds, respectively. At time `40`, an unmeasured output disturbance (a step change in the concentration of the reactor exit) occurs as well.

```
open_system([mdl, '/Concentration (OV)'])
open_system([mdl, '/Coolant Temp (MV)'])
sim(mdl)
```


Concentration (OV)
File
Tools
View
Simulation
Help
9
8
7
6
5
4
3
2
0
5
10
15
20
25
30
35
40
45
50
Ready
Sample based
T=50.000



The concentration in the exit stream tracks its reference accurately and converges to the desired final value. Also, the controller rejects both measured disturbances and the unmeasured disturbance.

The initial controller moves are limited by the maximum rate-of-change in the coolant temperature. This could be improved by providing the controller MPC with a look-ahead reference signal, which informs the controller of the expected reference variation over the prediction horizon.

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.

```
bdclose(mdl)
```

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2

Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver

This example shows how to find the optimal policy to treat a population with two-strain tuberculosis (TB) by constructing a nonlinear MPC problem. The nonlinear MPC controller then uses both the default solver and a custom solver to calculate the optimal solution.

Two-Strain Tuberculosis Model

A two-strain tuberculosis model is introduced in [1]. The model is described as follows:

"In the absence of an effective vaccine, current control programs for TB have focused on chemotherapy. The antibiotic treatment for an active TB (with drug-sensitive strain) patient requires a much longer period of time and a higher cost than that for those who are infected with sensitive TB but have not developed the disease. Lack of compliance with drug treatments not only may lead to a relapse but to the development of antibiotic resistant TB - one of the most serious public health problems facing society today. The reduction in cases of drug sensitive TB can be achieved either by "case holding", which refers to activities and techniques used to ensure regularity of drug intake for a duration adequate to achieve a cure, or by "case finding", which refers to the identification (through screening, for example) of individuals latently infected with sensitive TB who are at high risk of developing the disease and who may benefit from preventive intervention Description of first code block. These preventive treatments will reduce the incidence (new cases per unit of time) of drug sensitive TB and hence indirectly reduce the incidence of drug resistant TB."

In the dynamic model used in this example, the total host population `N` (which is a constant) is divided into six distinct epidemiological classes. Five of these classes are defined as state variables:

- x(1) = `S`, number of susceptible individuals
- x(2) = `T`, number of effectively treated individuals
- x(3) = `L2`, latent, infected with resistant TB but not infectious
- x(4) = `I1`, infectious with typical TB
- x(5) = `I2`, infectious with resistant TB

The sixth class, `L1` (latent, infected with typical TB but not infectious) is calculated as `N - (S+T +L2+I1+I2)`.

You can reduce resistant TB cases using two manipulated variables (MVs):

- u(1) - "case finding", relatively inexpensive effort expended to identify those needing treatment.
- u(2) - "case holding", relatively costly effort to maintain effective treatment.

For more information on the dynamic model, see `ResistantTBStateFcn.m`.

Nonlinear Control Problem

The goal of TB treatment is to reduce the latent (L2) and infectious (I2) individuals with resistant TB during a five-year period while keeping the cost low. To achieve this goal, use a cost function that sums the following value over five years.

```
F = L2 + I2 + 0.5*B1*u1^2 + 0.5*B2*u2^2
```

Here, weight `B1` is `50`, and weight B2 is `500`. These weights emphasize a preference for case finding over case holding due to its cost impact.

The total population is `N = 30000`. At the initial condition:

`S` = 19000 `T` = 250 `L2` = 500 `I1` = 1000 `I2` = 250

which leaves `L1` = 9000.

```
N = 30000;
x0 = [76; 1; 2; 4; 1]*N/120;
```

In this example, find the optimal control policy using a nonlinear MPC controller. Create the `nlmpc` object with correct numbers of states, outputs, and inputs.

```
nx = 5;
ny = nx;
nu = 2;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Assume that the treatment policy can only be adjusted every three months. Therefore, set the controller sample time to 0.25 years. Since you want to find the optimal policy over five years, set the prediction horizon to 20 steps (5 years divided by 0.25).

```
Years = 5;
Ts = 0.25;
p = Years/Ts;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
```

For this planning problem, you want to use the maximum number of decision variables. To do so, set the control horizon equal to the prediction horizon.

```
nlobj.ControlHorizon = p;
```

The prediction model is defined in `ResistantTBStateFcn.m`. Specify this function as the controller state function.

```
nlobj.Model.StateFcn = 'ResistantTBStateFcn';
```

It is best practice to specify analytical Jacobian functions for prediction model and cost/constraint functions. For details on the Jacobian calculation for the state equations, see `ResistantTBStateJacFcn.m`. Set this file as the state Jacobian function.

```
nlobj.Jacobian.StateFcn = 'ResistantTBStateJacFcn';
```

Because all the states are numbers of individuals, they must be nonnegative values. Specify a minimum bound of `0` for all states.

```
for ct = 1:nx
    nlobj.States(ct).Min = 0;
end
```

Because there is a large population variation among the groups (states), scale the state variables using their respective nominal values. Doing so improves the numerical robustness of the optimization problem.

```
for ct = 1:nx
    nlobj.States(ct).ScaleFactor = x0(ct);
end
```

Both "finding" and "holding" controls have an operating range between `0.05` and `0.95`. Set these values as the lower and upper bounds for the MVs.

```
nlobj.MV(1).Min = 0.05;
nlobj.MV(1).Max = 0.95;
nlobj.MV(2).Min = 0.05;
nlobj.MV(2).Max = 0.95;
```

The cost function, which minimizes the TB population and the treatment cost, is defined in `ResistantTBCostFcn.m`. Since this planning problem does not require reference tracking or disturbance rejection, replace the standard cost using this cost function.

```
nlobj.Optimization.CustomCostFcn = "ResistantTBCostFcn";
nlobj.Optimization.ReplaceStandardCost = true;
```

Also, to speed up simulation, the analytical Jacobian of the cost is provided in `ResistantTBCostJacFcn`.

```
nlobj.Jacobian.CustomCostFcn = "ResistantTBCostJacFcn";
```

Since the `L1` population is defined as `N` minus the sum of all states, you must ensure that `(S+T+L2+I1+I2) - N < 0` is always satisfied. In the `nlmpc` object, specify this condition as an inequality constraint using an anonymous function.

```
nlobj.Optimization.CustomIneqConFcn = @(X,U,e,data) sum(X(2:end,:),2)-30000;
```

To check for potential numerical issues, validate your prediction model, custom functions, and Jacobians using the `validateFcns` command.

```
validateFcns(nlobj,x0,[0.5;0.5])
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Jacobian.CustomCostFcn is OK.
Optimization.CustomIneqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

To compute the optimal control policy, use the `nlmpcmove` function. At the initial condition, the MVs are zero. By default, `fmincon` from the Optimization Toolbox™ is used as the default NLP solver.

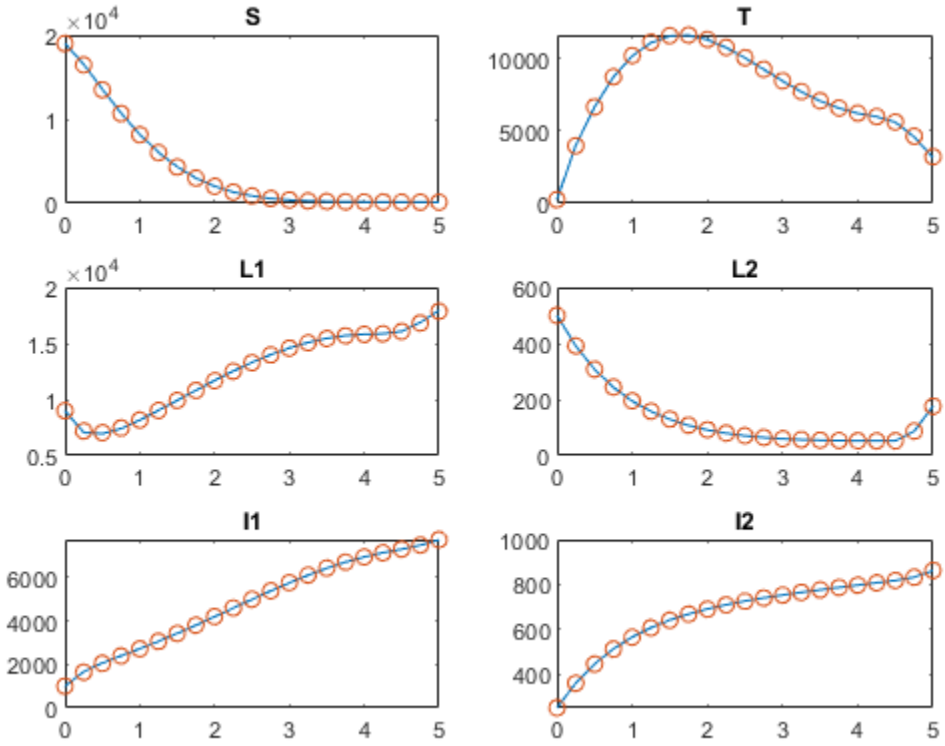
```
lastMV = zeros(nu,1);
[~,~,Info] = nlmpcmove(nlobj,x0,lastMV);
```

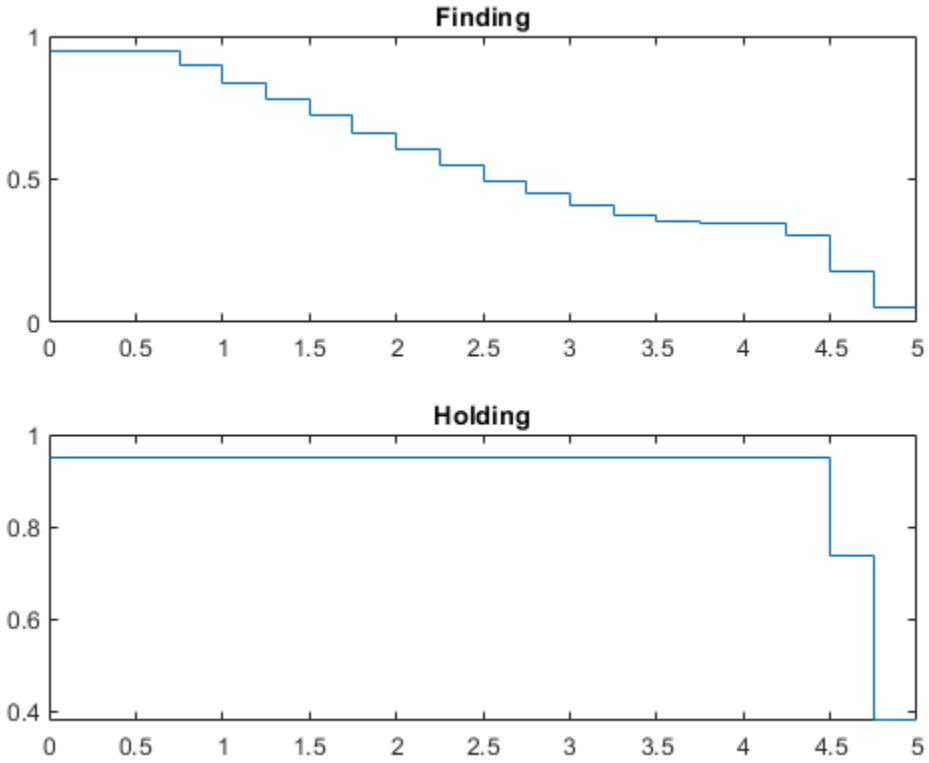
```
Slack variable unused or zero-weighted in your custom cost function. All constraints will be hard
```

Plot and examine the optimal solution.

```
ResistantTBPlot(Info,Ts)
```

```
Optimal cost =   5194.8
Final     L2 =    175.9
Final     I2 =    862.9
```


S
×10^4
2
1
0
0 1 2 3 4 5
T
10000
5000
0
0 1 2 3 4 5
L1
×10^4
2
1.5
1
0.5
0 1 2 3 4 5
L2
600
400
200
0
0 1 2 3 4 5
I1
6000
4000
2000
0
0 1 2 3 4 5
I2
1000
800
600
400
0 1 2 3 4 5



The optimal solution yields a cost of `5195`, and the total number of individuals infected with resistant TB at the final time is `L2 + I2 = 1037`.

Find Optimal Treatment Using Custom Nonlinear Programming Solver

If you want to use a third-party NLP solver in the simulation, write an interface file that converts the inputs defined by `nlmpc` into the inputs defined by your NLP solver, and specify it as the `CustomSolverFcn` in the `nlmpc` object.

In this example, assume that you have an "XYZ" solver that has a different user interface than `fmincon`. A `ResistantTBSolver.m` file is created to convert the optimization problem defined by `nlmpc` object to the proper interface required by the "XYZ" solver. Review the `ResistantTBSolver.m`.

```
function [zopt,cost,exitflag] = ResistantTBSolver(FUN,z0,A,b,Aeq,beq,lb,ub,NONLINCON)
% This is an interface function that calls a XYZ nonlinear programming
% solver to solve an NLP problem defined by an nlmpc controller. The
% input and output definitions of this function are identical to fmincon.
%
% This interface function converts fmincon inputs to the format required
% by the XYZ solver and converts the results back to the fmincon outputs.
%
% Inputs
%       FUN: nonlinear cost. FUN accepts input z (decision variabls) and
%            returns cost f (a scalar value evaluated at z) and its
%            gradient g (a nz-by-1 vector evaluated at z).
```

```
%        z0: initial guess of z
%       A,b: A*z<=b
%   Aeq,beq: Aeq*z==beq
%     lb,ub: lower and upper bounds of z
% NONLINCON: nonlinear constraints. NONLINCON accepts input z and returns
%            the vectors C and Ceq as the first two outputs, representing
%            the nonlinear inequalities and equalities respectively where
%            FUN is minimized subject to C(z) <= 0 and Ceq(z) = 0.
%            NONLINCON also returns Jacobian matrices of C (a nz-by-ncin
%            sparse matrix) and Ceq (a nz-by-nceq sparse matrix).
%
% Outputs
%     zopt:  optimal solution of z
%     cost:  optimal cost at optimal z
% exitflag:
%            1  First order optimality conditions satisfied.
%            0  Too many function evaluations or iterations.
%           -1  Stopped by output/plot function.
%           -2  No feasible point found.
%
% You can use this example as a template to implement an interface file to
% your own NLP solver. The solver must be an M file or a MEX file on the
% MATLAB path.
%
% DO NOT EDIT LINES ABOVE.
%
% Copyright 2018 The MathWorks, Inc.

%% Set dimensional information of linear and nonlinear constraints
num_lin_ineq = size(A,1);
num_lin_eq = size(Aeq,1);
[in,eq] = NONLINCON(z0);
num_non_ineq = length(in);
num_non_eq = length(eq);
total = num_non_ineq + num_non_eq + num_lin_ineq + num_lin_eq;
logicals_nlineq = false(total,1);
logicals_nlineq(1:num_non_ineq) = true;
logicals_nleq = false(total,1);
logicals_nleq(num_non_ineq+(1:num_non_eq)) = true;
logicals_ineq = false(total,1);
logicals_ineq(num_non_ineq+num_non_eq+(1:num_lin_ineq)) = true;
logicals_eq = false(total,1);
logicals_eq(num_non_ineq+num_non_eq+num_lin_ineq+(1:num_lin_eq)) = true;
options = struct('nlineq',logicals_nlineq,'nleq',logicals_nleq,...
                 'ineq',logicals_ineq,'eq',logicals_eq);
%% Set decision variable bounds
options.lb = lb;
options.ub = ub;
%% Set RHS of nonlinear constraints
options.cl = [-inf(num_non_ineq,1);zeros(num_non_eq,1)];
options.cu = [zeros(num_non_ineq,1);zeros(num_non_eq,1)];
%% Set RHS of linear constraints
options.rl = [-inf(num_lin_ineq,1);beq];
options.ru = [b;beq];
%% Set A matrix
options.A = sparse([A; Aeq]);
%% Set XYZ solver optons
options.algorithm = struct('print_level',0,'max_iter',200,...
```

```
                            'max_cpu_time',1000,'tol',1.0000e-06,...
                            'hessian_approximation','limited-memory');
%% Set function handles used by XYZ solver
Jstr = sparse(ones(num_non_ineq+num_non_eq,length(z0)));
funcs = struct('objective',@(x) fval(FUN,x),...
               'gradient',@(x) gval(FUN,x),...
               'constraints',@(x) conval(NONLINCON,x),...
               'jacobian',@(x) jacval(NONLINCON,x),...
               'jacobianstructure',@() Jstr...
               );
%% Call XYZ and return cost and status
[zopt,output] = XYZsolver(z0,funcs,options);
cost = FUN(zopt);
exitflag = convertStatustoExitflag(output.status);

%% Utility functions
function f = fval(fun,z)
% Return nonlinear cost
[f,~] = fun(z);

function g = gval(fun,z)
% Return cost gradient
[~,g] = fun(z);

function c = conval(nonlcon,z)
% Return nonlinear constraints
[in,eq] = nonlcon(z);
c = [in;eq];

function J = jacval(nonlcon,z)
% Return constraints Jacobian as nc-by-nz in sparse matrix
% Jin is nz-by-ncin sparse, Jeq is nz-by-nceq sparse
[~,~,Jin,Jeq] = nonlcon(z);
J = [Jin Jeq]';

function exitflag = convertStatustoExitflag(status)
switch(status)
    case 0
        %info.Status = 'Success';
        exitflag = 1;
    case 1
        %info.Status = 'Solved to Acceptable Level';
        exitflag = 1;
    case 2
        %info.Status = 'Infeasible';
        exitflag = -1;
    case 3
        %info.Status = 'Search Direction Becomes Too Small';
        exitflag = -2;
    case 4
        %info.Status = 'Diverging Iterates';
        exitflag = -2;
    case 6
        %info.Status = 'Feasible Point Found';
        exitflag = 1;
    case -1
        %info.Status = 'Exceeded Iterations';
        exitflag = 0;
```

```
    case -4
        %info.Status = 'Max Time Exceeded';
        exitflag = 0;
    otherwise
        %info.Status = 'IPOPT Error';
        exitflag = -2;
end
```

You can use this file as a template to implement an interface file to your own NLP solver. The solver must be a MATLAB script or MEX file on the MATLAB path.

You can plug in the solver by specifying it as the custom solver in the `nlmpc` object.

```
nlobj.Optimization.CustomSolverFcn = @ResistantTBSolver;
```

As long as the "XYZ" solver is reliable and its options are properly chosen, rerunning the simulation should produce similar results.

References

[1] Jung, E., S. Lenhart, and Z. Feng. "Optimal Control of Treatments in a Two-Strain Tuberculosis Model." *Discrete and Continuous Dynamical Systems*, Series B2, 2002, pp. 479-482.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Configure Optimization Solver for Nonlinear MPC” on page 9-27

Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant

This example shows how to use a nonlinear model predictive controller to control an ethylene oxidation plant as it transitions from one operating point to another. In addition, linear MPC controllers are generated directly from the nonlinear MPC controller to implement a gain-scheduled control scheme that produces comparable performance.

Ethylene Oxidation Plant

Oxidation of ethylene (C2H4) to ethylene oxide (C2H4O) occurs in a cooled, gas-phase catalytic reactor. Three reactions occur simultaneously in the well-mixed gas phase within the reactor:

```
C2H4 + 0.5*O2 -> C2H4O
```

```
C2H4 + 3*O2 -> 2*CO2 + 2*H2O
```

```
C2H4O + 2.5*O2 -> 2*CO2 + 2*H2O
```

A mixture of air and ethylene is fed continuously. The plant is described by a first-principle nonlinear dynamic model, implemented as a set of ordinary differential equations (ODEs) in the `oxidationStateFcn` function. For more information, see `oxidationStateFcn.m`.

The plant contains four states:

- Gas density in the reactor (*x1*)
- C2H4 concentration in the reactor (*x2*)
- C2H4O concentration in the reactor (*x3*)
- Temperature in the reactor (*x4*)

The plant has two inputs:

- Total volumetric feed flow rate (*u1*)
- C2H4 concentration of the feed (*u2*)

The plant has one output:

- C2H4O concentration in the effluent flow (*y*, equivalent to *x3*)

For convenience, all variables in the model are pre-scaled to be dimensionless. All the states are measurable. The plant equations and parameters are obtained from [1].

Control Objectives

In this example, the total volumetric feed flow rate (*u1*) is the manipulated variable (MV) and C2H4O concentration in the effluent flow (*y*) is the output variable (OV). Good tracking performance of *y* is required within an operating range from `0.03` to `0.05`. The corresponding *u1* values are `0.38` and `0.15`, respectively.

The C2H4 concentration in the feed flow (*u2*) is a measured disturbance. Its nominal value is `0.5`, and a typical disturbance has a size of `0.1`. The controller is required to reject such a disturbance.

The manipulated variable *u1* has a range from `0.0704` to `0.7042` due to actuator limitations.

Feedback Control with Nonlinear MPC

In general, using nonlinear MPC with an accurate nonlinear prediction model provides a benchmark performance; that is, the best control solution you can achieve. However, in practice, linear MPC control solutions, such as adaptive MPC or gain-scheduled MPC, are more computationally efficient than nonlinear MPC. If your linear control solution can deliver a comparable performance, there is no need to implement the nonlinear control solution, especially in a real-time environment.

In this example, you first design a nonlinear MPC controller to obtain the benchmark performance. Afterward, you generate several linear MPC objects from the nonlinear controller at different operating point using the `convertToMPC` function. Finally, you implement gain-scheduled MPC using these linear MPC objects and compare the performance.

Create a nonlinear MPC controller with 4 states, 1 output, 1 manipulated variable, and 1 measured disturbance.

```
nlobj = nlmpc(4,1,'MV',1,'MD',2);
```

Specify the controller sample time and the prediction and control horizons.

```
Ts = 5;
PredictionHorizon = 10;
ControlHorizon = 3;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = PredictionHorizon;
nlobj.ControlHorizon = ControlHorizon;
```

Specify the nonlinear prediction model using `oxidationStateFcn.m`, which is a continuous-time model.

```
nlobj.Model.StateFcn = 'oxidationStateFcn';
nlobj.States(1).Name  = 'Den';
nlobj.States(2).Name  = 'CE';
nlobj.States(3).Name  = 'CEO';
nlobj.States(4).Name  = 'Tc';
```

Specify the output function that returns the C2H4O concentration in the effluent flow (same as x3). Its scale factor is its typical operating range.

```
nlobj.Model.OutputFcn = @(x,u) x(3);
nlobj.OV.Name = 'CEOout';
nlobj.OV.ScaleFactor = 0.03;
```

Specify the MV constraints based on the controller actuator limitations. Its scale factor is its typical operating range.

```
nlobj.MV.Min = 0.0704;
nlobj.MV.Max = 0.7042;
nlobj.MV.Name = 'Qin';
nlobj.MV.ScaleFactor = 0.6;
```

Specify the measured disturbance name. Its scale factor is its typical operating range.

```
nlobj.MD.Name = 'CEin';
nlobj.MD.ScaleFactor = 0.5;
```

Initially the plant is at an equilibrium operating point with a low concentration of C2H4O (y = `0.03`) in the effluent flow. Find the initial values of the states and output using `fsolve` from the Optimization Toolbox.

```
options = optimoptions('fsolve','Display','none');
uLow = [0.38 0.5];
xLow = fsolve(@(x) oxidationStateFcn(x,uLow),[1 0.3 0.03 1],options);
yLow = xLow(3);
```

Validate that the prediction model functions do not have any numerical issues using the `validateFcns` command. Validate the functions at the initial state and output values.

```
validateFcns(nlobj,xLow,uLow(1),uLow(2));
```

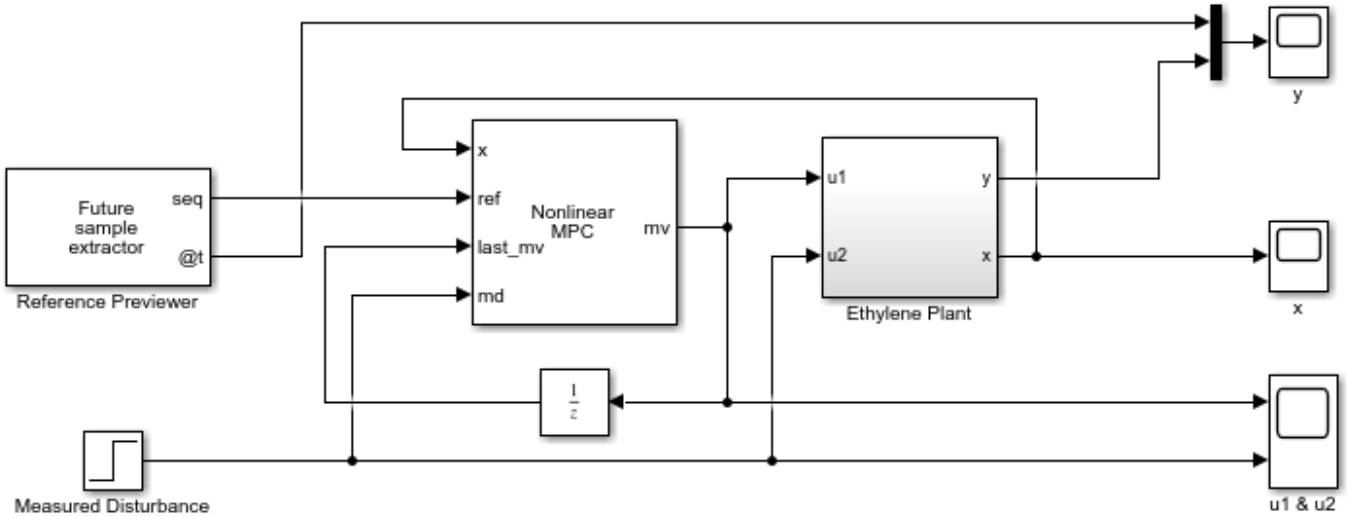
```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Specify the reference signal in a structure, where it ramps up from `0.03` to `0.05` in `50` seconds at time `100`.

```
Tstop = 300;
time = (0:Ts:(Tstop+PredictionHorizon*Ts))';
r = [yLow*ones(sum(time<100),1);linspace(yLow,yLow+0.02,11)';(yLow+0.02)*ones(sum(time>150),1)];
ref.time = time;
ref.signals.values = r;
```

To assess nonlinear MPC performance, use a Simulink model. The Nonlinear MPC Controller block in the model is configured to use `nlobj` as its controller.

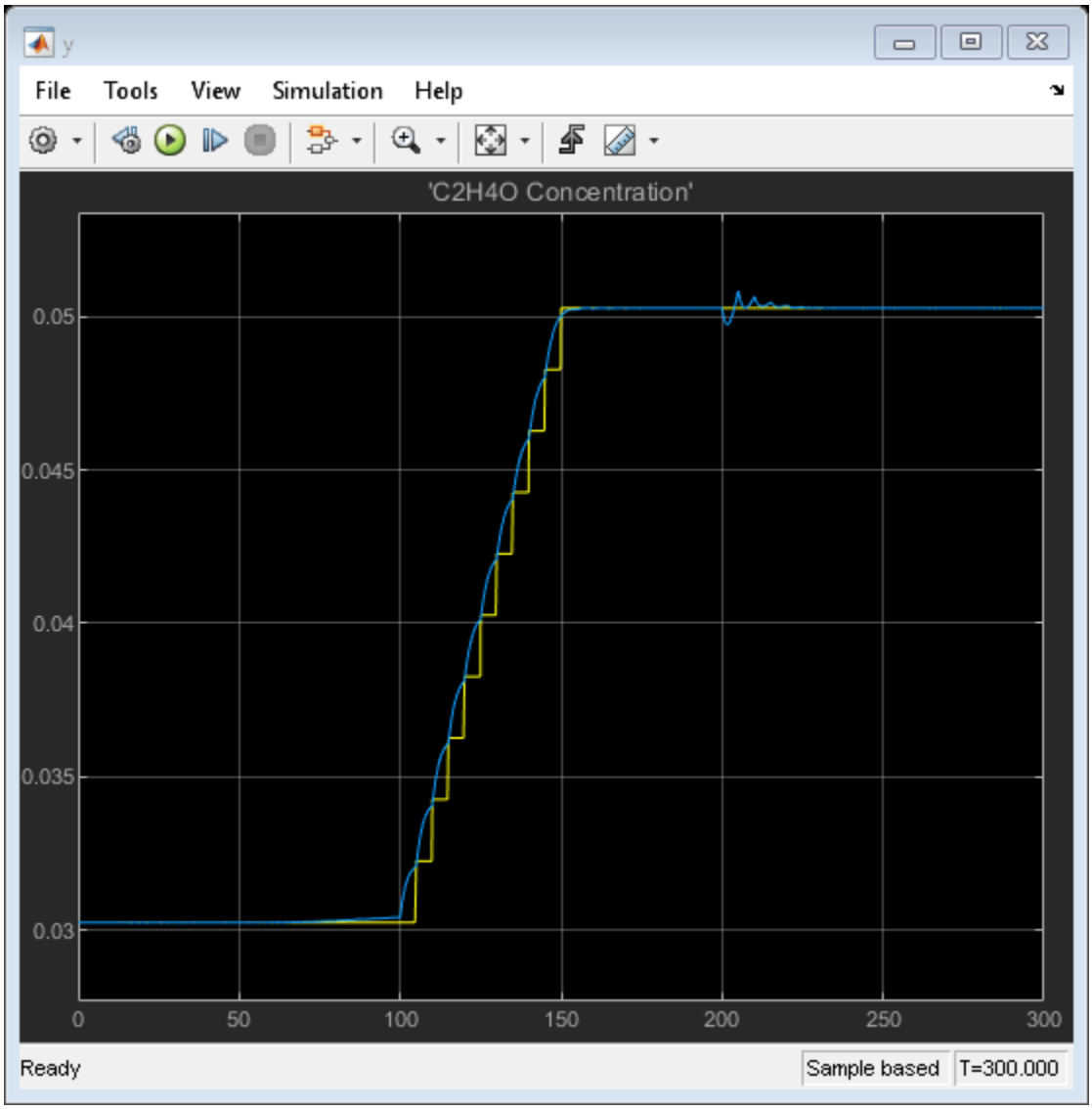
```
mdlNMPC = 'oxidationNMPC';
open_system(mdlNMPC)
```



Copyright 1990-2018 The MathWorks, Inc.

Run the simulation, and view the output.

```
sim(mdlNMPC)
open_system([mdlNMPC '/y'])
```



The nonlinear MPC controller produces good reference tracking and disturbance rejection performance, as expected.

Although a ramp-like set-point change in C2H4O concentration occurs between 100 and 150 seconds (the yellow stair curve in the plot), the controller knows about the change as early as at 50 seconds because of reference previewing. Since the objective is to minimize tracking errors across the whole horizon, the controller decides to move the plant in advance such that tracking error is the smallest across the prediction horizon. If previewing is disabled, the controller would start reacting at 100 seconds, which would produce a larger tracking error.

In this example, since all the states are measurable, full state feedback is used by the nonlinear MPC controller. In general, when there are unmeasurable states, you must design a nonlinear state estimator, such as an extended Kalman filter (EKF) or a moving horizon estimator (MHE).

Obtain Linear MPC Controllers from Nonlinear MPC Controller

In practice, when producing comparable performance, linear MPC is always preferred over nonlinear MPC due to its higher computation efficiency. Since you designed a nonlinear MPC controller as a benchmark, you can convert it into a linear MPC controller at a specific operating point.

In this example, you generate three linear MPC controllers with C2H4O concentrations at `0.03`, `0.04`, and `0.05`, respectively. During the conversion, the nonlinear plant model is linearized at the specified operating point. All the scale factors, linear constraints, and quadratic weights defined in the nonlinear MPC object are retained. However, any custom nonlinear cost function or custom nonlinear equality or inequality constraints are discarded.

Generate a linear MPC controller at an operating point with low C2H4O conversion rate `y = 0.03`. Specify the operating point using the corresponding state and input values, `xLow` and `uLow`, respectively.

```
mpcobjLow = convertToMPC(nlobj,xLow,uLow);
```

Generate a linear MPC controller at an operating point with medium C2H4O conversion rate `y = 0.04`.

```
uMedium = [0.24 0.5];
xMedium = fsolve(@(x) oxidationStateFcn(x,uMedium),[1 0.3 0.03 1],options);
mpcobjMedium = convertToMPC(nlobj,xMedium,uMedium);
```

Generate a linear MPC controller at an operating point with high C2H4O conversion rate `y = 0.05`.

```
uHigh = [0.15 0.5];
xHigh = fsolve(@(x) oxidationStateFcn(x,uHigh),[1 0.3 0.03 1],options);
mpcobjHigh = convertToMPC(nlobj,xHigh,uHigh);
```

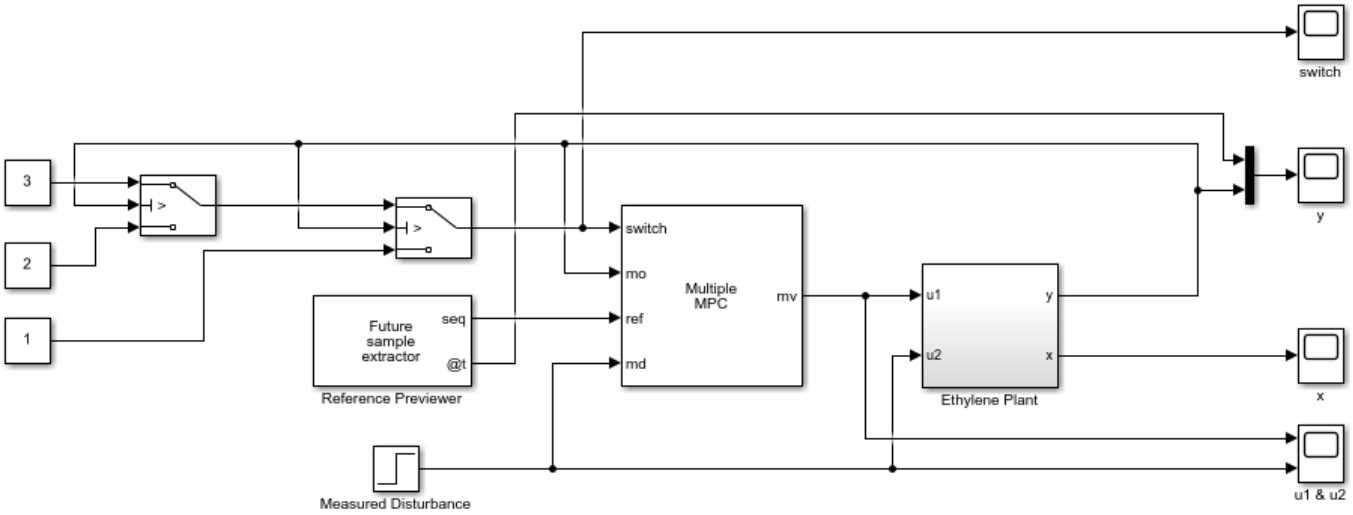
Feedback Control with Gain-Scheduled MPC

Implement a gain-scheduled MPC solution using the three generated linear MPC controllers. The scheduling scheme is:

- If y is lower than 0.035, use `mpcobjLow`.
- If y is higher than 0.045, use `mpcobjHigh`.
- Otherwise, use `mpcobjMedium`.

To assess the gain-scheduled controller performance, use another Simulink model.

```
mdlMPC = 'oxidationMPC';
open_system(mdlMPC)
```

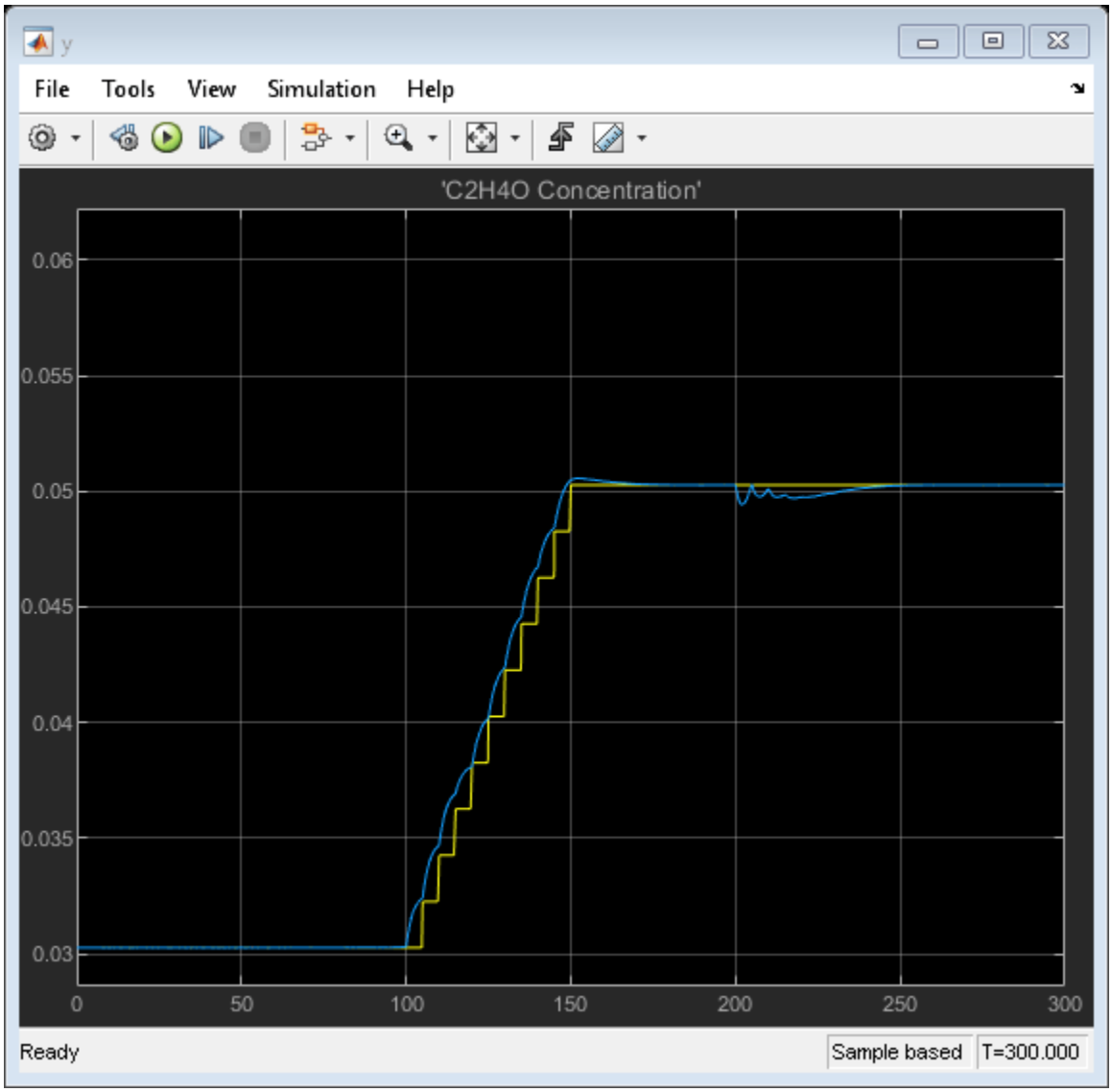


Copyright 1990-2018 The MathWorks, Inc.

Run the simulation, and view the output.

```
sim(mdlMPC)
open_system([mdlMPC '/y'])
```

```
-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
-->Converting model to discrete time.
-->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 measured
```



The gain-scheduled controller produces comparable reference tracking and disturbance rejection performance.

Conclusion

This example illustrates a general workflow to:

- Design and simulate a nonlinear MPC controller in MATLAB and Simulink for a benchmark control performance.
- Use the nonlinear MPC object to directly generate linear MPC controllers at desired operating points.
- Implement a gain-scheduled MPC control scheme using these controllers.

If the performance of the gain-scheduled controller is comparable to that of the nonlinear controller, you can feel confident implementing a linear control solution to a nonlinear control problem.

Close the Simulink model.

```
bdclose(mdlNMPC)
```

References

[1] H. Durand, M. Ellis, P. D. Christofides. "Economic model predictive control designs for input rate-of-change constraint handling and guaranteed economic performance." *Computers and Chemical Engineering*, Vol. 92, 2016, pp. 18-36.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2

Optimization and Control of a Fed-Batch Reactor Using Nonlinear MPC

This example shows how to use nonlinear model predictive control to optimize batch reactor operation. The example also shows how to run a nonlinear MPC controller as an adaptive MPC controller and a time-varying MPC controller to quickly compare their performance.

Fed-Batch Chemical Reactor

The following irreversible and exothermic reactions occur in the batch reactor [1]:

```
A + B => C  (desired product)
C => D      (undesired product)
```

The batch begins with the reactor partially filled with known concentrations of reactants `A` and `B`. The batch reacts for 0.5 hours, during which additional `B` can be added and the reactor temperature can be changed.

The nonlinear model of the batch reactor is defined in the `fedbatch_StateFcn` and `fedbatch_OutputFcn` functions. This system has the following inputs, states, and outputs.

Manipulated Variables

- `u1` = `u_B`, flow rate of `B` feed
- `u2` = `Tsp`, reactor temperature setpoint, deg C

Measured disturbance

- `u3` = `c_Bin`, concentration of `B` in the `B` feed flow

States

- `x1` = `V*c_A`, mol of `A` in the reactor
- `x2` = `V*(c_A + c_C)`, mol of `A + C` in the reactor
- `x3` = `V`, liquid volume in the reactor
- `x4` = `T`, reactor temperature, K

Outputs

- `y1` = `V*c_C`, amount of product `C` in the reactor, equivalent to `x2-x1`
- `y2` = `q_r`, heat removal rate, a nonlinear function of the states
- `y3` = `V`, liquid volume in reactor

The goal is to maximize the production of `C` (`y1`) at the end of the batch process. During the batch process, the following operating constraints must be satisfied:

1. Hard upper bound on heat removal rate (`y2`). Otherwise, temperature control fails.
2. Hard upper bound on liquid volume in reactor (`y3`) for safety.
3. Hard upper and lower bounds on `B` feed rate (`u_B`).
4. Hard upper and lower bounds on reactor temperature setpoint (`Tsp`).

Specify the nominal operating condition at the beginning of the batch process.

```
c_A0 = 10;
c_B0 = 1.167;
c_C0 = 0;
V0  = 1;
T0 = 50 + 273.15;
c_Bin = 20;
```

Specify the nominal states.

```
x0 = zeros(3,1);
x0(1) = c_A0*V0;
x0(2) = x0(1) + c_C0*V0;
x0(3) = V0;
x0(4) = T0;
```

Specify the nominal inputs.

```
u0 = zeros(3,1);
u0(2) = 40;
u0(3) = c_Bin;
```

Specify the nominal outputs.

```
y0 = fedbatch_OutputFcn(x0,u0);
```

Nonlinear MPC Design to Optimize Batch Operation

Create a nonlinear MPC object with 4 states, 3 outputs, 2 manipulated variables, and 1 measured disturbance.

```
nlmpcobj_Plan = nlmpc(4, 3, 'MV', [1,2], 'MD', 3);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Given the expected batch duration `Tf`, choose the controller sample time `Ts` and prediction horizon.

```
Tf = 0.5;
N = 50;
Ts = Tf/N;
nlmpcobj_Plan.Ts = Ts;
nlmpcobj_Plan.PredictionHorizon = N;
```

If you set the control horizon equal to the prediction horizon, there will be 50 free control moves, which leads to a total of 100 decision variables because the plant has two manipulated variables. To reduce the number of decision variables, you can specify control horizon using blocking moves. Divide the prediction horizon into 8 blocks, which represents 8 free control moves. Each of the first seven blocks lasts seven prediction steps. Doing so reduces the number of decision variables to 16.

```
nlmpcobj_Plan.ControlHorizon = [7 7 7 7 7 7 7 1];
```

Specify the nonlinear model in the controller. The function `fedbatch_StateFcnDT` converts the continuous-time model to discrete time using a multi-step Forward Euler integration formula.

```
nlmpcobj_Plan.Model.StateFcn = @(x,u) fedbatch_StateFcnDT(x,u,Ts);
nlmpcobj_Plan.Model.OutputFcn = @(x,u) fedbatch_OutputFcn(x,u);
nlmpcobj_Plan.Model.IsContinuousTime = false;
```

Specify the bounds for feed rate of `B`.

```
nlmpcobj_Plan.MV(1).Min = 0;
nlmpcobj_Plan.MV(1).Max = 1;
```

Specify the bounds for the reactor temperature setpoint.

```
nlmpcobj_Plan.MV(2).Min = 20;
nlmpcobj_Plan.MV(2).Max = 50;
```

Specify the upper bound for the heat removal rate. The true constraint is `1.5e5`. Since nonlinear MPC can only enforce constraints at the sampling instants, use a safety margin of `0.05e5` to prevent a constraint violation between sampling instants.

```
nlmpcobj_Plan.OV(2).Max = 1.45e5;
```

Specify the upper bound for the liquid volume in the reactor.

```
nlmpcobj_Plan.OV(3).Max = 1.1;
```

Since the goal is to maximize `y1`, the amount of `C` in the reactor at the end of the batch time, specify a custom cost function that replaces the default quadratic cost. Since `y1 = x2-x1`, define the custom cost to be minimized as `x1-x2` using an anonymous function.

```
nlmpcobj_Plan.Optimization.CustomCostFcn = @(X,U,e,data) X(end,1)-X(end,2);
nlmpcobj_Plan.Optimization.ReplaceStandardCost = true;
```

To configure the manipulated variables to vary linearly with time within each block, select piecewise linear interpolation. By default, nonlinear MPC keeps manipulated variables constant within each block, using piecewise constant interpolation, which might be too restrictive for an optimal trajectory planning problem.

```
nlmpcobj_Plan.Optimization.MVInterpolationOrder = 1;
```

Use the default nonlinear programming solver `fmincon` to solve the nonlinear MPC problem. For this example, set the solver step tolerance to help achieve first order optimality.

```
nlmpcobj_Plan.Optimization.SolverOptions.StepTolerance = 1e-8;
```

Before carrying out optimization, check whether all the custom functions satisfy NLMPC requirements using the `validateFcns` command.

```
validateFcns(nlmpcobj_Plan, x0, u0(1:2), u0(3));
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Optimization.CustomCostFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Analysis of Optimization Results

Find the optimal trajectories for the manipulated variables such that production of `C` is maximized at the end of the batch process. To do so, use the `nlmpcmove` function.

```
fprintf('\nOptimization started...\n');
[~,~,Info] = nlmpcmove(nlmpcobj_Plan,x0,u0(1:2),zeros(1,3),u0(3));
fprintf('   Expected production of C (y1) is %g moles.\n',Info.Yopt(end,1));
fprintf('   First order optimality is satisfied (Info.ExitFlag = %i).\n',...
    Info.ExitFlag);
fprintf('Optimization finished...\n');
```

```
Optimization started...
Slack variable unused or zero-weighted in your custom cost function. All constraints will be hard
   Expected production of C (y1) is 2.02353 moles.
   First order optimality is satisfied (Info.ExitFlag = 1).
Optimization finished...
```

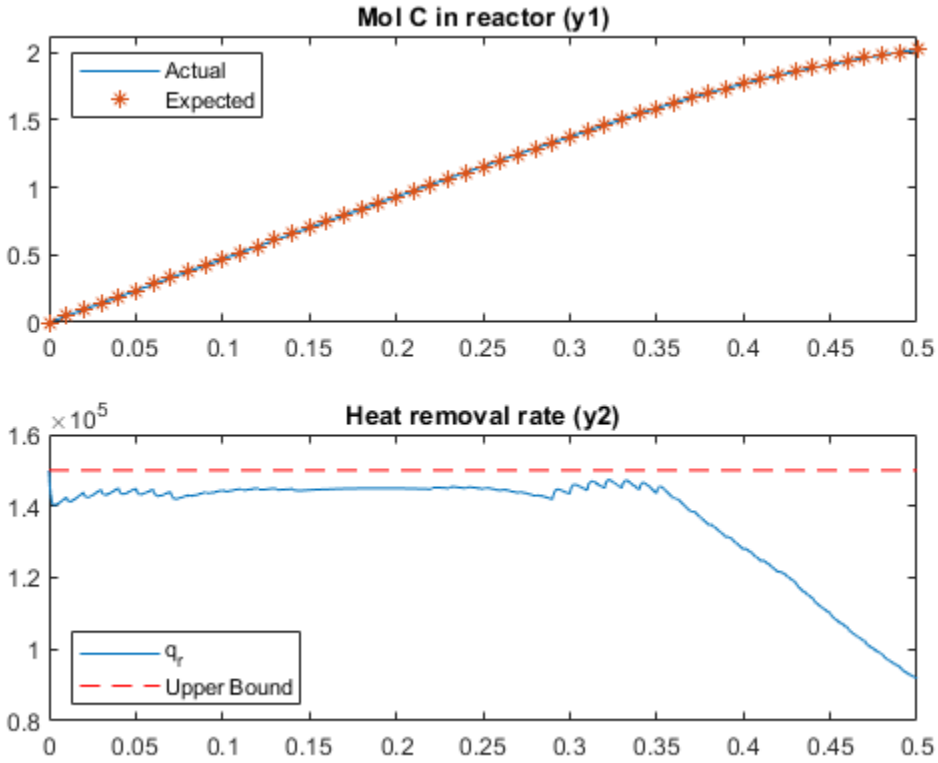
The discretized model uses a simple Euler integration, which could be inaccurate. To check this, integrate the model using the `ode15s` command for the calculated optimal MV trajectory.

```
Nstep = size(Info.Xopt,1) - 1;
t = 0;
X = x0';
t0 = 0;
for i = 1:Nstep
    u_in = [Info.MVopt(i,1:2)'; c_Bin];
    ODEFUN = @(t,x) fedbatch_StateFcn(x, u_in);
    TSPAN = [t0, t0+Ts];
    Y0 = X(end,:)';
    [TOUT,YOUT] = ode15s(ODEFUN,TSPAN,Y0);
    t = [t; TOUT(2:end)];
    X = [X; YOUT(2:end,:)];
    t0 = t0 + Ts;
end
nx = size(X,1);
Y = zeros(nx,3);
for i = 1:nx
    Y(i,:) = fedbatch_OutputFcn(X(i,:)',u_in)';
end
fprintf('\n   Actual Production of C (y1) is %g moles.\n',X(end,2)-X(end,1));
fprintf('   Heat removal rate (y2) satisfies the upper bound.\n');
```

```
   Actual Production of C (y1) is 2.0228 moles.
   Heat removal rate (y2) satisfies the upper bound.
```

In the top plot of the following figure, the actual production of `C` agrees with the expected production of `C` calculated from `nlmpcmove`. In the bottom plot, the heat removal rate never exceeds its hard constraint.

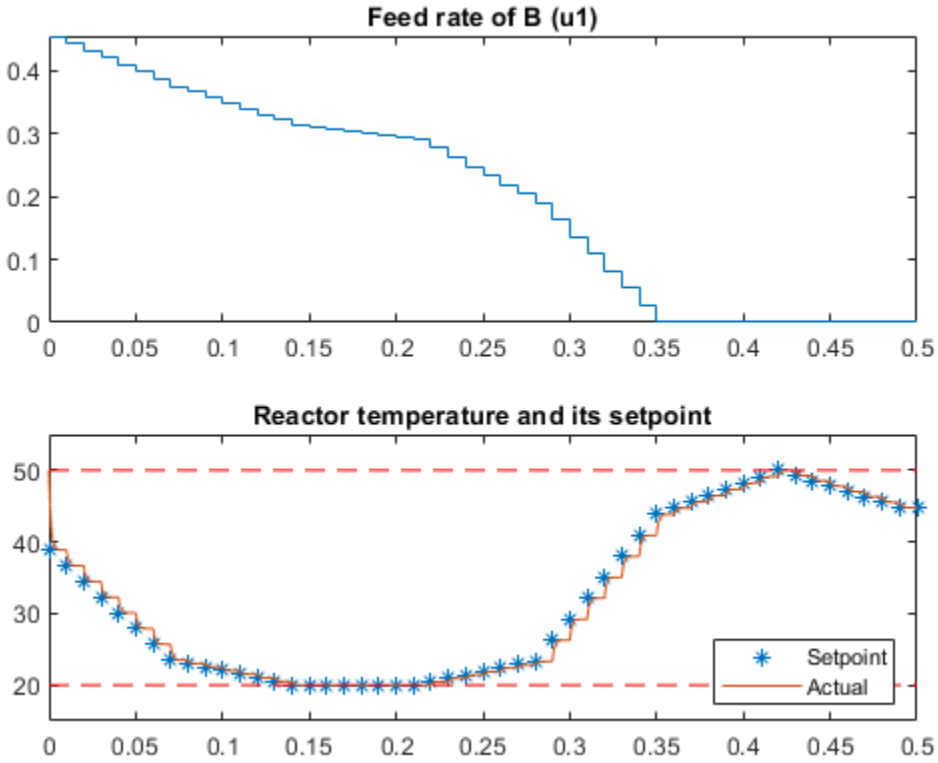
```
figure
subplot(2,1,1)
plot(t,Y(:,1),(0:Nstep)*Ts, Info.Yopt(:,1),'*')
axis([0 0.5 0 Y(end,1) + 0.1])
legend({'Actual','Expected'},'location','northwest')
title('Mol C in reactor (y1)')
subplot(2,1,2)
tTs = (0:Nstep)*Ts;
t(end) = 0.5;
plot(t,Y(:,2),'-',[0 tTs(end)],1.5e5*ones(1,2),'r--')
axis([0 0.5 0.8e5, 1.6e5])
legend({'q_r','Upper Bound'},'location','southwest')
title('Heat removal rate (y2)')
```



Close examination of the heat removal rate shows that it can exhibit peaks and valleys between the sampling instants as reactant compositions change. Consequently, the heat removal rate exceeds the specified maximum of `1.45e5` (around t = 0.35 h) but stays below the true maximum of `1.5e5`.

The following figure shows the optimal trajectory of planned adjustments in the B feed rate (`u1`), and the reactor temperature (`x4`) and its setpoint (`u2`).

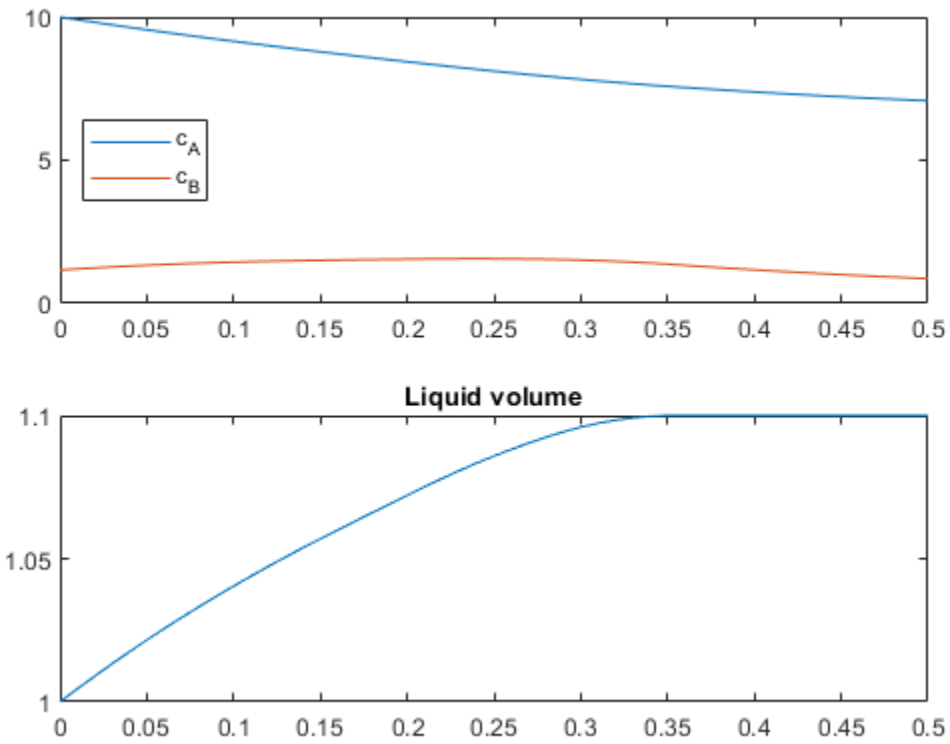
```
figure
subplot(2,1,1)
stairs(tTs,Info.MVopt(:,1))
title('Feed rate of B (u1)')
subplot(2,1,2)
plot(tTs,Info.MVopt(:,2),'*',t,X(:,4)-273.15,'-',...
    [0 0.5],[20 20],'r--',[0 0.5],[50 50],'r--')
axis([0 0.5 15 55])
title('Reactor temperature and its setpoint')
legend({'Setpoint','Actual'},'location','southeast')
```



The trajectory begins with a relatively high feed rate, which increases `c_B` and the resulting `C` production rate. To prevent exceeding the heat removal rate constraint, reactor temperature and feed rate must decrease. The temperature eventually hits its lower bound and stays there until the reactor is nearly full and the `B` feed rate must go to zero. The temperature then increases to its maximum (to increase `C` production) and finally drops slightly (to reduce `D` production, which is favored at higher temperatures).

The top plot of the following figure shows the consumption of `c_A`, which tends to reduce `C` production. To compensate, the plan first increases `c_B`, and when that is no longer possible (the reactor liquid volume must not exceed 1.1), the plan makes optimal use of the temperature. In the bottom plot of the following figure, the liquid volume never exceeds its upper bound.

```
figure
subplot(2,1,1)
c_A = X(:,1)./X(:,3);
c_B = (c_Bin*X(:,3) + X(:,1) + V0*(c_B0 - c_A0 - c_Bin))./X(:,3);
plot(t,[c_A, c_B])
legend({'c_A','c_B'}, 'location', 'west')
subplot(2,1,2)
plot(tTs,Info.Yopt(:,3))
title('Liquid volume')
```



Nonlinear MPC Design for Tracking the Optimal C Product Trajectory

To track the optimal trajectory of product C calculated above, you design another nonlinear MPC controller with the same prediction model and constraints. However, use the standard quadratic cost and default horizons for tracking purposes.

To simplify the control task, assume that the optimal trajectory of the B feed rate is implemented in the plant and the tracking controller considers it to be a measured disturbance. Therefore, the controller uses the reactor temperature setpoint as its only manipulated variable to track the desired y1 profile.

Create the tracking controller.

```
nlmpcobj_Tracking = nlmpc(4,3,'MV',2,'MD',[1,3]);
nlmpcobj_Tracking.Ts = Ts;
nlmpcobj_Tracking.Model = nlmpcobj_Plan.Model;
nlmpcobj_Tracking.MV = nlmpcobj_Plan.MV(2);
nlmpcobj_Tracking.OV = nlmpcobj_Plan.OV;
nlmpcobj_Tracking.Weights.OutputVariables = [1 0 0];        % track y1 only
nlmpcobj_Tracking.Weights.ManipulatedVariablesRate = 1e-6;  % agressive MV
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Obtain the C production (y1) reference signal from the optimal plan trajectory.

```
Cref = Info.Yopt(:,1);
```

Obtain the feed rate of B (u1) from the optimal plan trajectory. The feed concentration of B (u3) is a constant.

```
MD = [Info.MVopt(:,1) c_Bin*ones(N+1,1)];
```

First, run the tracking controller in nonlinear MPC mode.

```
[X1,Y1,MV1,et1] = fedbatch_Track(nlmpcobj_Tracking,x0,u0(2),N,Cref,MD);
fprintf('\nNonlinear MPC: Elapsed time = %g sec. Production of C = %g mol\n',et1,Y1(end,1));
```

```
Nonlinear MPC: Elapsed time = 3.65202 sec. Production of C = 2.01131 mol
```

Second, run the controller as an adaptive MPC controller.

```
nlmpcobj_Tracking.Optimization.RunAsLinearMPC = 'Adaptive';
[X2,Y2,MV2,et2] = fedbatch_Track(nlmpcobj_Tracking,x0,u0(2),N,Cref,MD);
fprintf('\nAdaptive MPC: Elapsed time = %g sec. Production of C = %g mol\n',et2,Y2(end,1));
```

```
Adaptive MPC: Elapsed time = 2.43837 sec. Production of C = 2.01567 mol
```

Third, run the controller as a time-varying MPC controller.

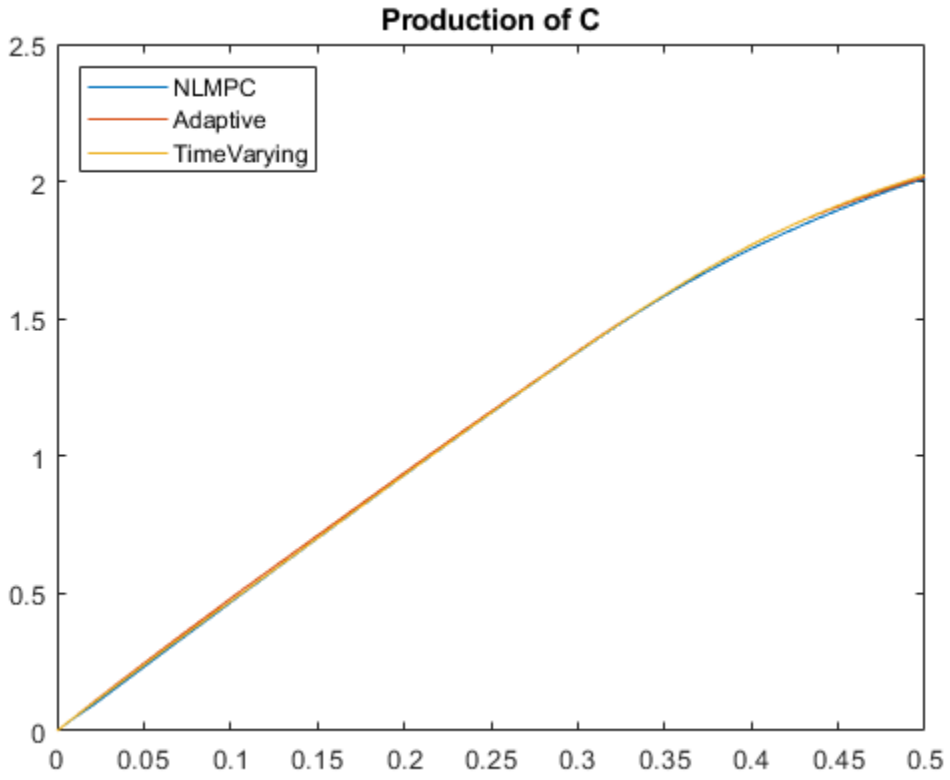
```
nlmpcobj_Tracking.Optimization.RunAsLinearMPC = 'TimeVarying';
[X3,Y3,MV3,et3] = fedbatch_Track(nlmpcobj_Tracking,x0,u0(2),N,Cref,MD);
fprintf('\nTime-varying MPC: Elapsed time = %g sec. Production of C = %g mol\n',et3,Y3(end,1));
```

```
Time-varying MPC: Elapsed time = 2.04578 sec. Production of C = 2.02349 mol
```

In the majority of MPC applications, linear MPC solutions, such as Adaptive MPC and Time-varying MPC, provide performance that is comparable to the nonlinear MPC solution, while consuming less resources and executing faster. In these cases, nonlinear MPC often represents the best control results that MPC can achieve. By running a nonlinear MPC controller as a linear MPC controller, you can assess whether implementing a linear MPC solution is good enough in practice.

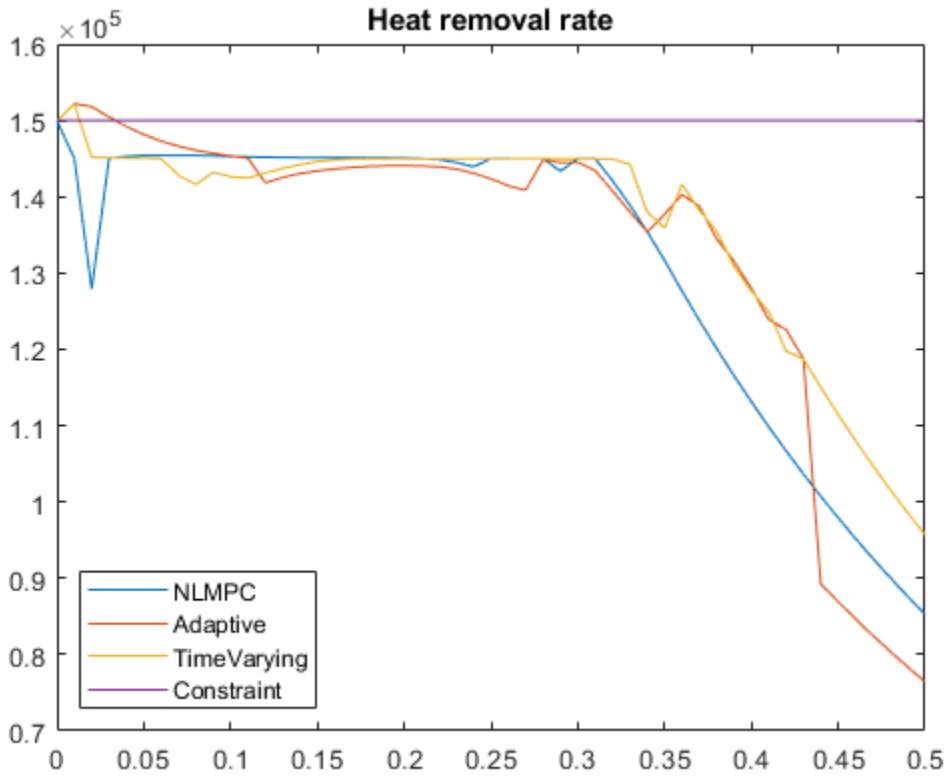
In this example, all three methods come close to the optimal C production obtained in the planning stage.

```
figure
plot(Ts*(0:N),[Y1(:,1) Y2(:,1) Y3(:,1)])
title('Production of C')
legend({'NLMPC','Adaptive','TimeVarying'},'location','northwest')
```



The unexpected result is that time-varying MPC produces more `C` than nonlinear MPC. The explanation is that the model linearization approaches used in the adaptive and time-varying modes result in a violation of the heat removal constraint, which results in a higher `C` production.

```
figure
plot(Ts*(0:N),[Y1(:,2) Y2(:,2) Y3(:,2) 1.5e5*ones(N+1,1)])
title('Heat removal rate')
legend({'NLMPC','Adaptive','TimeVarying','Constraint'},'location','southwest')
```



The adaptive MPC mode uses the plant states and inputs at the beginning of each control interval to obtain a single linear prediction model. This approach does not account for the known future changes in the feed rate, for example.

The time-varying method avoids this issue. However, at the start of the batch it assumes (by default) that the states will remain constant over the horizon. It corrects for this once it obtains its first solution (using data in the `opts` variable), but its initial choice of reactor temperature is too high, resulting in an early `q_r` constraint violation.

References

[1] Srinivasan, B., S. Palanki, and D. Bonvin, "Dynamic optimization of batch processes I. Characterization of the nominal solution", *Computers and Chemical Engineering*, vol. 27 (2003), pp. 1-26.

See Also

`nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Adaptive MPC” on page 7-2
- “Time-Varying MPC” on page 7-49

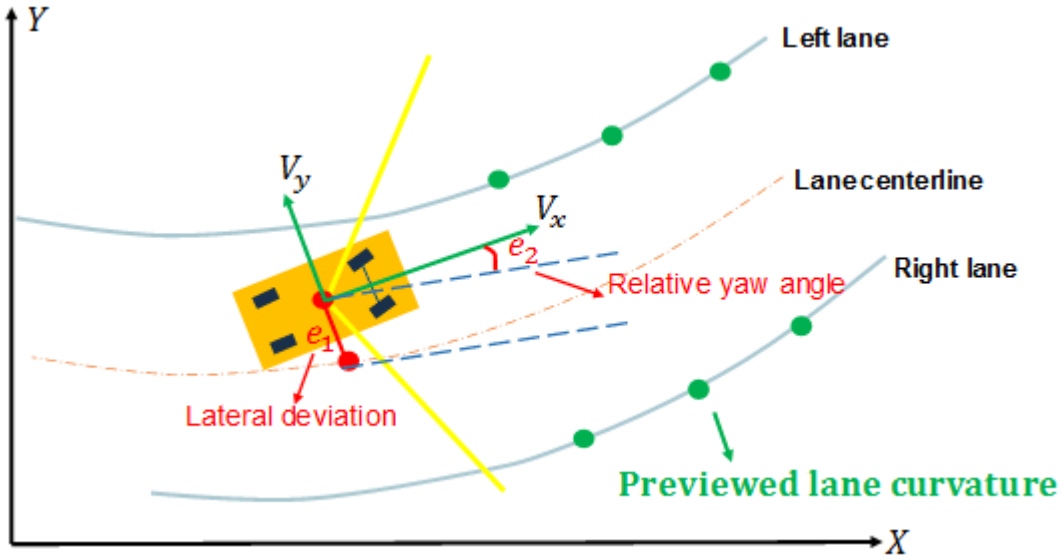
Lane Following Using Nonlinear Model Predictive Control

This example shows how to design a lane-following controller using the Nonlinear Model Predictive Controller block. In this example, you:

1. Design a nonlinear MPC controller (NLMPC) for lane following.
2. Compare the performance of NLMPC with adaptive MPC.

Introduction

A lane-following system is a control system that keeps the vehicle traveling along the centerline of a highway lane, while maintaining a user-set velocity. The lane-following scenario is depicted in the following figure.



A lane-following system manipulates both the longitudinal acceleration and front steering angle of the vehicle to:

- Keep the lateral deviation e_1 and relative yaw angle e_2 small.
- Keep the longitudinal velocity V_x close to a driver set velocity.
- Balance the above two goals when they cannot be met simultaneously.

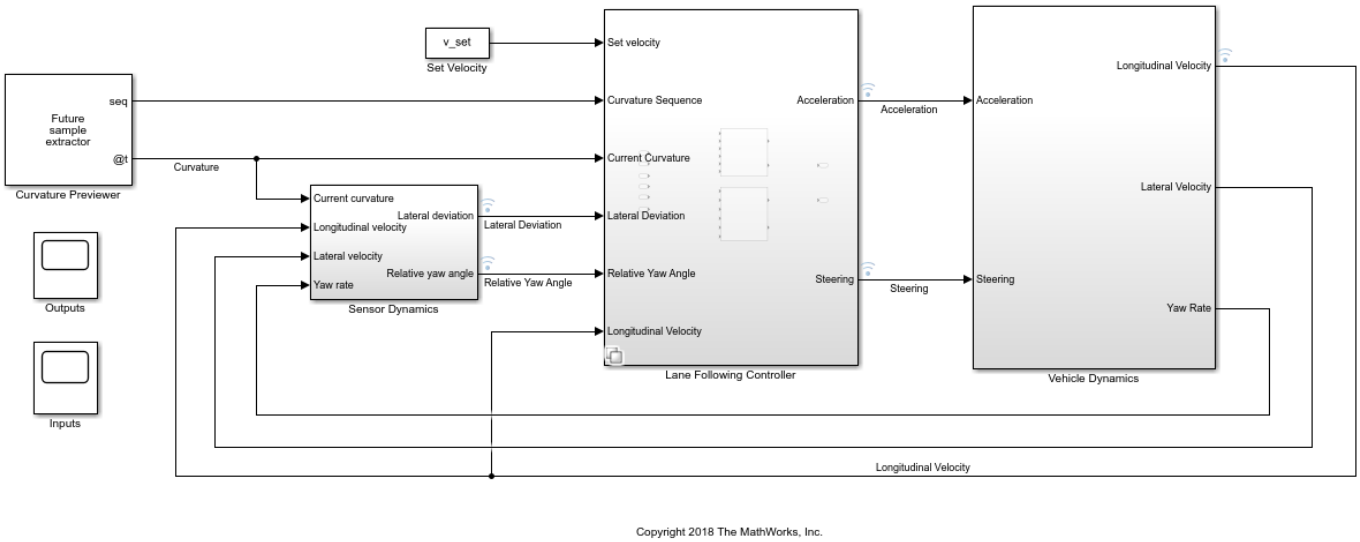
In a separate example of lane keeping assist, it is assumed that the longitudinal velocity is constant. For more information, see “Lane Keeping Assist System Using Model Predictive Control” on page 11-28. This restriction is relaxed in this example because the longitudinal acceleration varies in this MIMO control system.

Another example augments a lane-following system with spacing control, where a safe distance from a detected lead car is also maintained. For more information, see “Lane Following Control with Sensor Fusion and Lane Detection” on page 11-51.

Overview of Simulink Model

Open the Simulink model.

```
mdl = 'LaneFollowingNMPC';
open_system(mdl)
```



Copyright 2018 The MathWorks, Inc.

This model contains four main components:

1. Vehicle Dynamics: Apply the *bicycle mode* of lateral vehicle dynamics, and approximate the longitudinal dynamics using a time constant T.
2. Sensor Dynamics: Approximate a sensor such as a camera to calculate the lateral deviation and relative yaw angle.
3. Lane Following Controller: Simulate nonlinear MPC and adaptive MPC.
4. Curvature Previewer: Detect the curvature at the current time step and the curvature sequence over the prediction horizon of the MPC controller.

The vehicle dynamics and sensor dynamics are discussed in more details in “Adaptive Cruise Control with Sensor Fusion” on page 11-10. This example applies the same model for vehicle and sensor dynamics.

Parameters of Vehicle Dynamics and Road Curvature

The necessary Vehicle Dynamics and Road Curvature parameters are defined using the `LaneFollowingUsingNMPCData` script which is a `PreLoadFcn` callback of the model.

Design Nonlinear Model Predictive Controller

The continuous-time prediction model for NLMPC has the following state and output equations. The state equations are defined in `LaneFollowingStateFcn`.

$$\frac{d}{dt}\begin{bmatrix} V_y \\ \dot{\varphi} \\ V_x \\ \dot{V}_x \\ e_1 \\ e_2 \\ x_{od} \end{bmatrix} = \begin{bmatrix} -\frac{2C_f+2C_r}{mV_x} & -V_x-\frac{2C_fl_f-2C_rl_r}{mV_x} & 0 & 0 & 0 & 0 & 0 \\ -\frac{2C_fl_f-2C_rl_r}{I_zV_x} & -\frac{2C_fl_f^2+2C_rl_r^2}{I_zV_x} & 0 & 0 & 0 & 0 & 0 \\ \dot{\psi} & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-1}{\tau} & 0 & 0 & 0 \\ 1 & 0 & e_2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} V_y \\ \dot{\varphi} \\ V_x \\ \dot{V}_x \\ e_1 \\ e_2 \\ x_{od} \end{bmatrix} + \begin{bmatrix} 0 & \frac{2C_f}{m} \\ 0 & \frac{2C_fl_f}{I_z} \\ 0 & 0 \\ \frac{1}{\tau} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} a \\ \delta \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -V_x \\ 0 \end{bmatrix}[\rho] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}[w_{od}]$$

$$y = \begin{bmatrix} V_x & e_1 & e_2 + x_{od} \end{bmatrix}$$

The prediction model includes an unmeasured disturbance (UD) model. The UD model describes what type of unmeasured disturbance NLMPC expects to encounter and reject in the plant. In this example, the UD model is an integrator with its input assumed to be white noise. Its output is added to the relative yaw angle. Therefore, the controller expects a random step-like unmeasured disturbance occurring at the relative yaw angle output and is prepared to reject it when it happens.

Create a nonlinear MPC controller with a prediction model that has seven states, three outputs, and two inputs. The model has two MV signals: acceleration and steering. The product of the road curvature and the longitudinal velocity is modeled as a measured disturbance, and the unmeasured disturbance is modeled by white noise.

```
nlobj = nlmpc(7,3,'MV',[1 2],'MD',3,'UD',4);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Specify the controller sample time, prediction horizon, and control horizon.

```
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 2;
```

Specify the state function for the nonlinear plant model and its Jacobian.

```
nlobj.Model.StateFcn = @(x,u) LaneFollowingStateFcn(x,u);
nlobj.Jacobian.StateFcn = @(x,u) LaneFollowingStateJacFcn(x,u);
```

Specify the output function for the nonlinear plant model and its Jacobian. The output variables are:

- Longitudinal velocity
- Lateral deviation
- Sum of the yaw angle and yaw angle output disturbance

```
nlobj.Model.OutputFcn = @(x,u) [x(3);x(5);x(6)+x(7)];
nlobj.Jacobian.OutputFcn = @(x,u) [0 0 1 0 0 0 0;0 0 0 0 1 0 0;0 0 0 0 0 1 1];
```

Set the constraints for manipulated variables.

```
nlobj.MV(1).Min = -3;      % Maximum acceleration 3 m/s^2
nlobj.MV(1).Max = 3;       % Minimum acceleration -3 m/s^2
```

```
nlobj.MV(2).Min = -1.13;   % Minimum steering angle -65
nlobj.MV(2).Max = 1.13;    % Maximum steering angle 65
```

Set the scale factors.

```
nlobj.OV(1).ScaleFactor = 15;   % Typical value of longitudinal velocity
nlobj.OV(2).ScaleFactor = 0.5;  % Range for lateral deviation
nlobj.OV(3).ScaleFactor = 0.5;  % Range for relative yaw angle
nlobj.MV(1).ScaleFactor = 6;    % Range of steering angle
nlobj.MV(2).ScaleFactor = 2.26; % Range of acceleration
nlobj.MD(1).ScaleFactor = 0.2;  % Range of Curvature
```

Specify the weights in the standard MPC cost function. The third output, yaw angle, is allowed to float because there are only two manipulated variables to make it a square system. In this example, there is no steady-state error in the yaw angle as long as the second output, lateral deviation, reaches 0 at steady state.

```
nlobj.Weights.OutputVariables = [1 1 0];
```

Penalize acceleration change more for smooth driving experience.

```
nlobj.Weights.ManipulatedVariablesRate = [0.3 0.1];
```

Validate prediction model functions at an arbitrary operating point using the `validateFcns` command. At this operating point:

- `x0` contains the state values.
- `u0` contains the input values.
- `ref0` contains the output reference values.
- `md0` contains the measured disturbance value.

```
x0 = [0.1 0.5 25 0.1 0.1 0.001 0.5];
u0 = [0.125 0.4];
ref0 = [22 0 0];
md0 = 0.1;
validateFcns(nlobj,x0,u0,md0,{},ref0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
Model.OutputFcn is OK.
Jacobian.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

In this example, an extended Kalman filter (EKF) provides state estimation for the seven states. The state transition function for the EKF is defined in `LaneFollowingEKFStateFcn`, and the measurement function is defined in `LaneFollowingEKFMeasFcn`.

Design Adaptive Model Predictive Controller

An adaptive MPC (AMPC) controller is also designed using the Path Following Control System block in this example. This controller uses a linear model for the vehicle dynamics and updates the model online as the longitudinal velocity varies.

In practice, as long as a linear control solution such as adaptive MPC or gain-scheduled MPC can achieve comparable control performance against nonlinear MPC, you would implement the linear control solution because it is more computationally efficient.

Compare Controller Performance

To compare the results of NLMPC and AMPC, simulate the model and save the logged data.

First, simulate the model using nonlinear MPC. To do so, set `controller_type` to `1`.

```
controller_type = 1;
sim(mdl)
logsout1 = logsout;
```

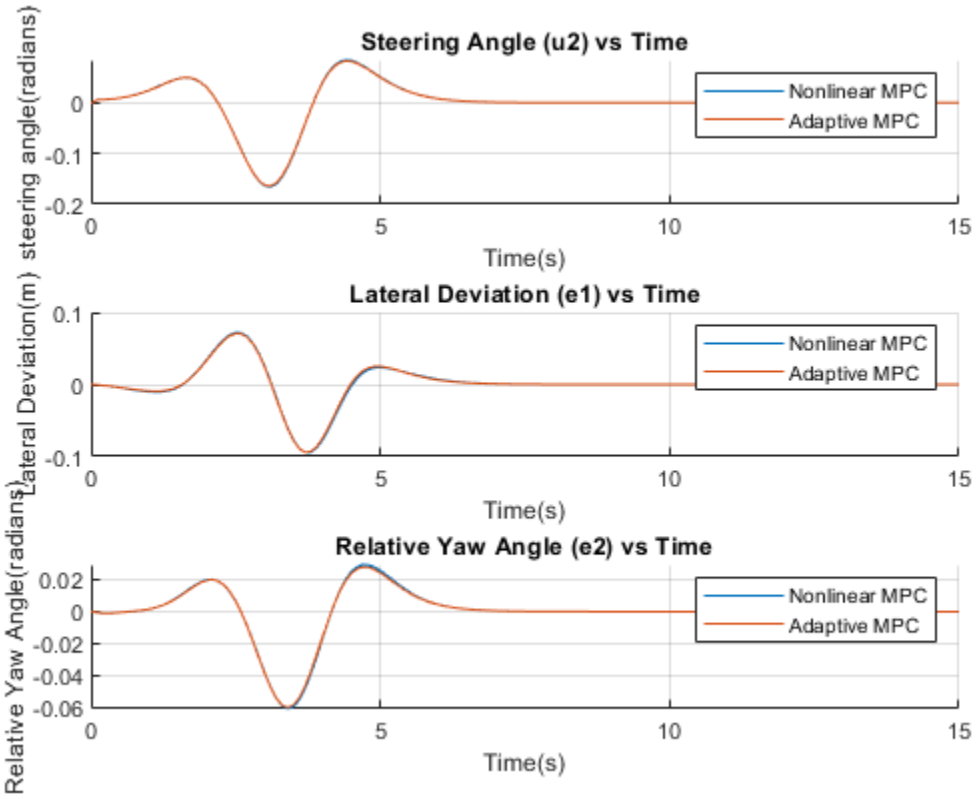
Second, simulate the model using adaptive MPC. To do so, set `controller_type` to `2`.

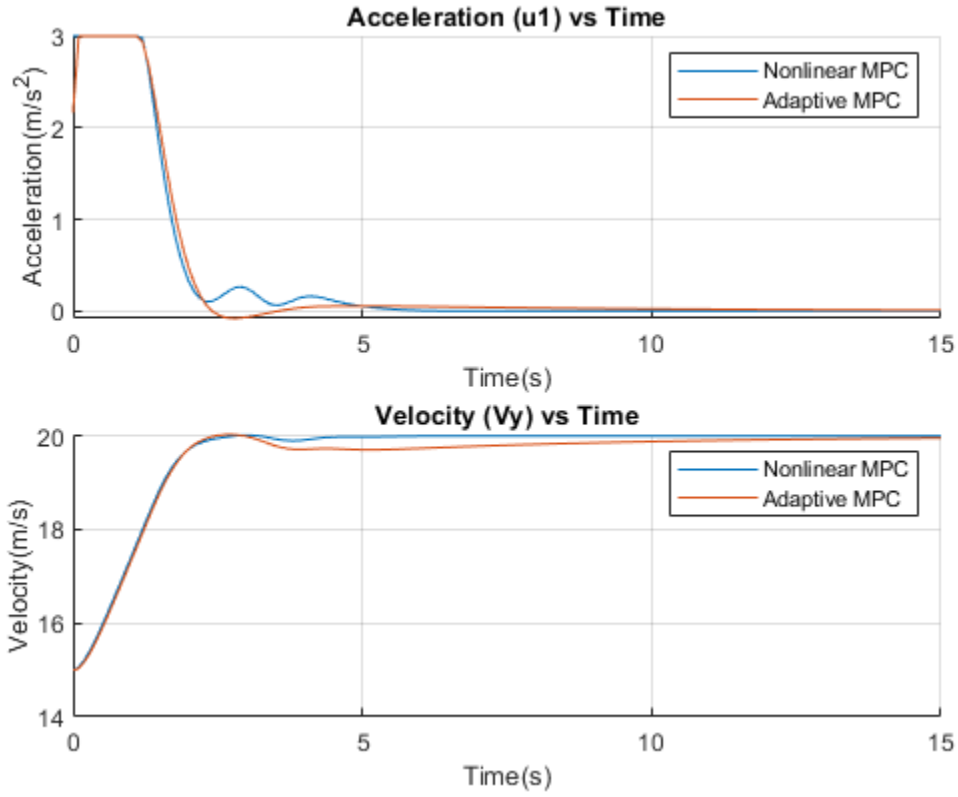
```
controller_type = 2;
sim(mdl)
logsout2 = logsout;
```

```
   Assuming no disturbance added to measured output channel #1.
   Assuming no disturbance added to measured output channel #2.
-->Assuming output disturbance added to measured output channel #3 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Plot and compare simulation results.

```
LaneFollowingCompareResults(logsout1,logsout2)
```





In the first plot, both nonlinear MPC and adaptive MPC give almost identical steering angle profiles. The lateral deviation and relative yaw angle are close to zero during the maneuver. This result implies that the vehicle is traveling along the desired path.

The longitudinal control command and performance for nonlinear and adaptive MPC are slightly different. The nonlinear MPC controller has smoother acceleration command and better tracking of set velocity, although the result from adaptive MPC is also acceptable.

You can also view the results via Scopes of Outputs and Inputs in the model.

Set the controller variant to nonlinear MPC.

```
controller_type = 1;
```

Conclusion

This example shows how to design a nonlinear model predictive controller for lane following. The performance of using nonlinear MPC and adaptive MPC is compared. You can select nonlinear MPC or adaptive MPC depending on the modeling information and computational power for your application.

```
% close Simulink model
bdclose(mdl)
```

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Lane Change Assist Using Nonlinear Model Predictive Control” on page 9-101

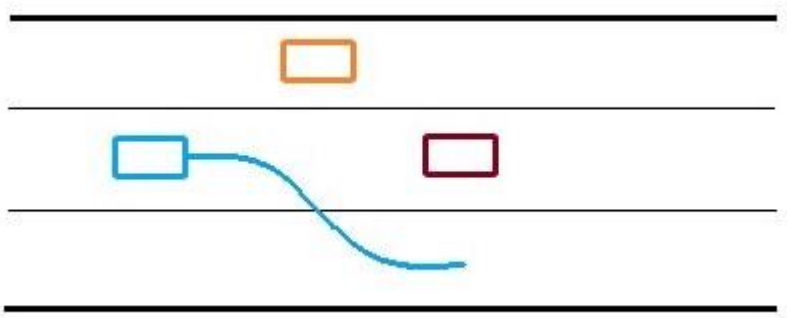
Lane Change Assist Using Nonlinear Model Predictive Control

This example shows how to design a lane-change controller using a multistage nonlinear model predictive control (MPC). In this example, you:

1. Review a control algorithm that combines a custom AStar path planning algorithm and a lane-change controller designed using the Model Predictive Control Toolbox™ software.
2. Design a multistage nonlinear MPC controller for autonomous lane changing.
3. Test the closed-loop control system in a Simulink® model using driving scenarios generated using Automated Driving Toolbox™ software.

Introduction

A lane change assist control system autonomously steers an ego vehicle to an adjacent lane when there is another vehicle moving slower in front of it, as shown in the following figure.



The lane change controller in this example is designed to work when the ego vehicle is driving on a straight road at a constant velocity, though it can be extended to other driving scenarios with appropriate modifications.

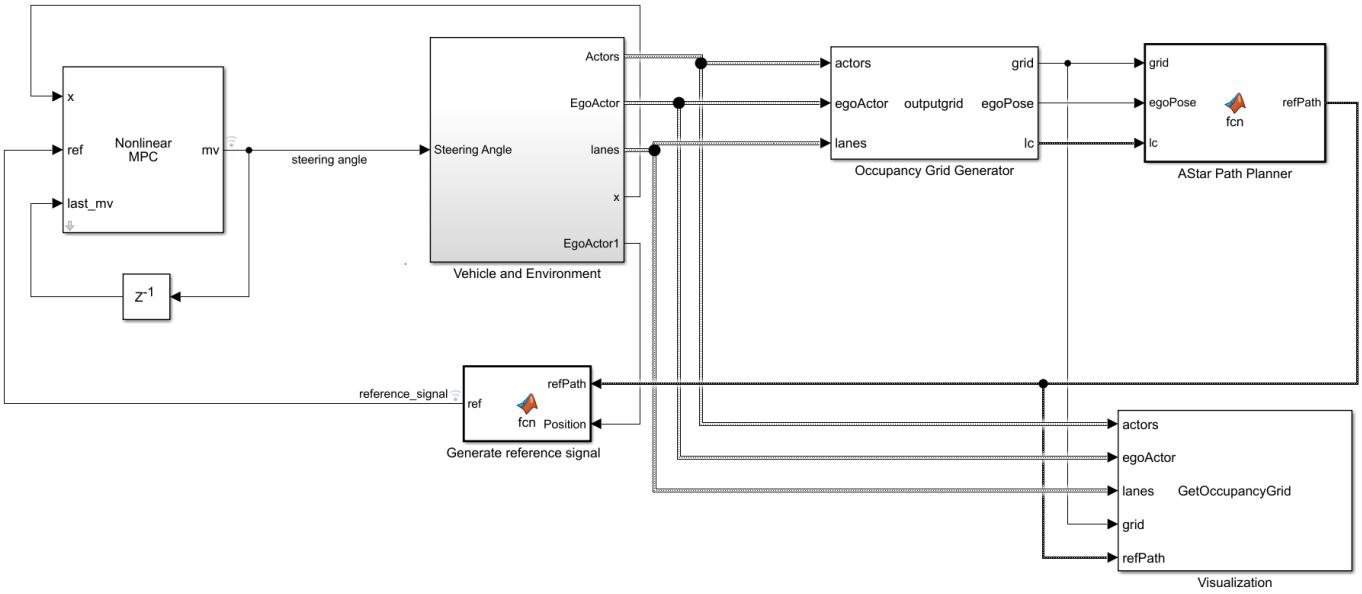
In this example:

1. A driving scenario is used to model the environment such that a situation requiring a lane change arises. The scenario was created and exported using the **Driving Scenario Designer** app from the Automated Driving Toolbox.
2. Based on this scenario, a discrete occupancy grid is populated, which is then used by the path planner to plan a collision-free reference path for the ego vehicle.
3. Once the reference path is generated, the controller performs the autonomous lane change maneuver by controlling the steering angle of the ego vehicle to track the lateral position of the planned path.

Overview of Simulink Model

Open the Simulink model.

```
mdl = 'LaneChangeExample';
open_system(mdl)
```



The model contains four main components:

1. Nonlinear MPC — Lane change controller, which controls the front steering angle of the ego vehicle
2. Vehicle and Environment — Models the motion of the ego vehicle and models the environment
3. Occupancy Grid Generator — Generates a discrete grid that contains information about the environment and cars surrounding the ego vehicle
4. AStar Path Planner — Plans a collision-free path for the ego vehicle considering the dynamic behavior of other cars

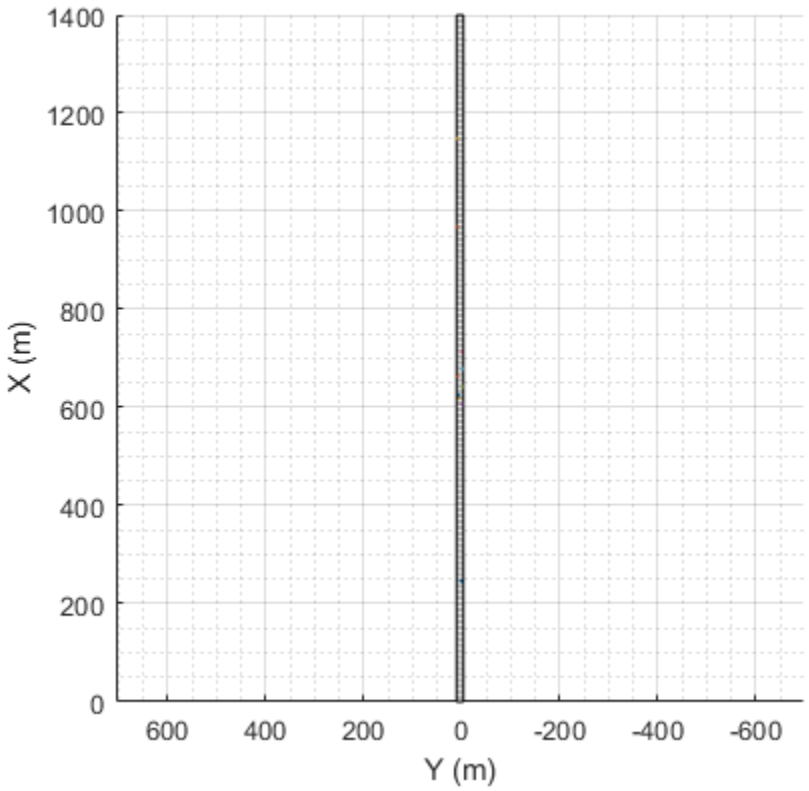
Inside the Vehicle and Environment subsystem, the Vehicle Dynamics subsystem models the vehicle dynamics using the Bicycle Model - Velocity Input block from the Automated Driving Toolbox.

Opening this model runs the `helperLCSetUp` script, which initializes the data used by the Simulink model, such as the vehicle model parameters, controller design parameters, road scenario, and surrounding cars.

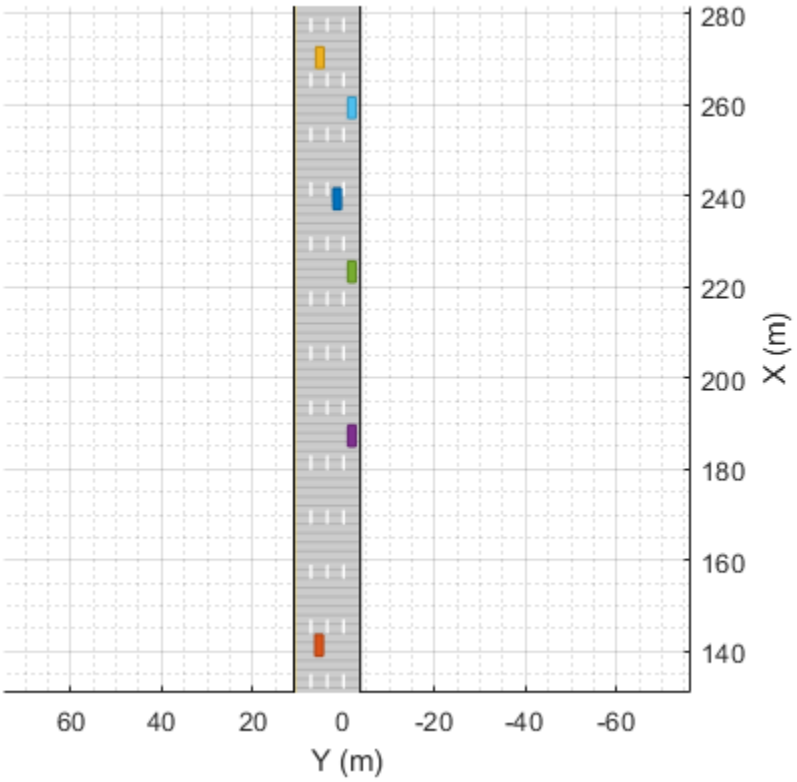
The multistage nonlinear MPC controller for this example is designed using the `createNLmpcObjLC` function, which is called from the `helperLCSetUp` script. This controller uses the state equations defined in `vehicleStateFcnLC.m` and controls the steering angle of the ego vehicle.

Plot the scenario with the road and the cars that the ego vehicle will encounter.

```
plot(scenario)
```

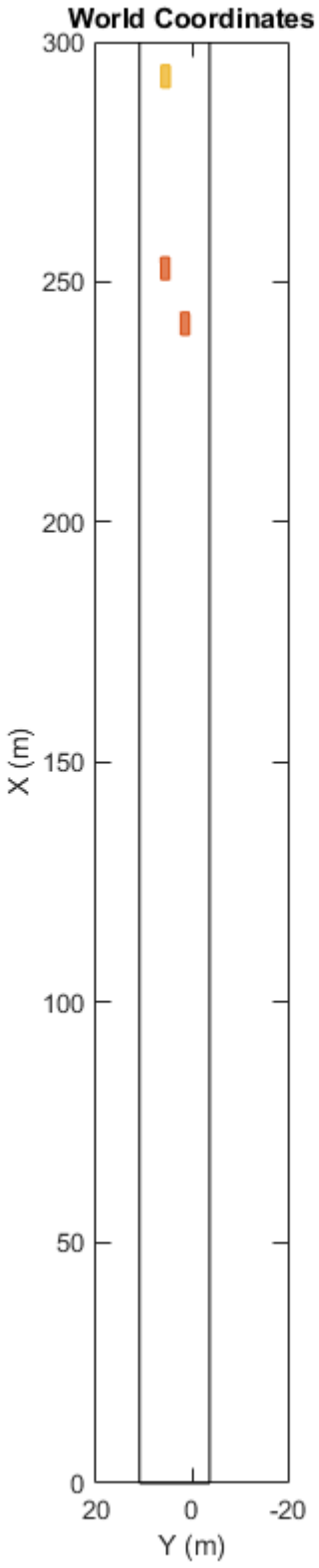


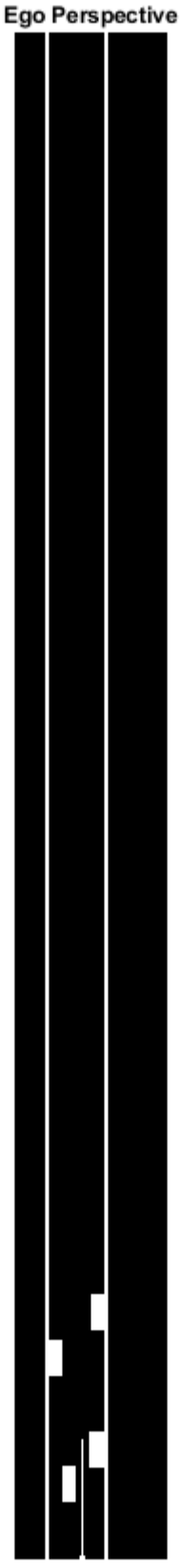
The following figure shows a zoomed-in portion of the road.



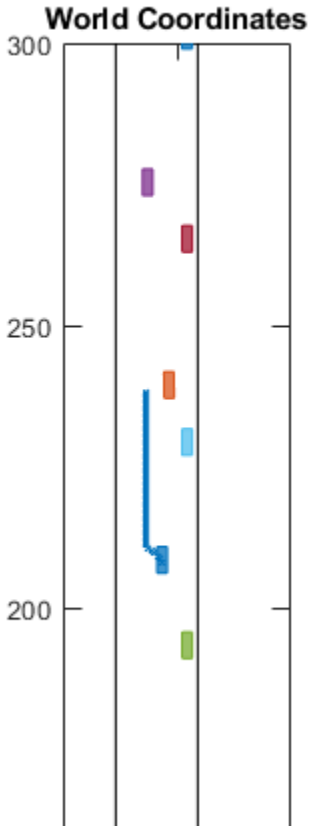
Simulate the model to the end of the scenario. Simulating the model opens the Bird's Eye Plot in **World Coordinates** and the occupancy grid in **Ego Perspective**. The occupancy grid shows a representation of the road and vehicles in front of the ego vehicle and includes the planned path as a white line.

```
out = sim(mdl);
```

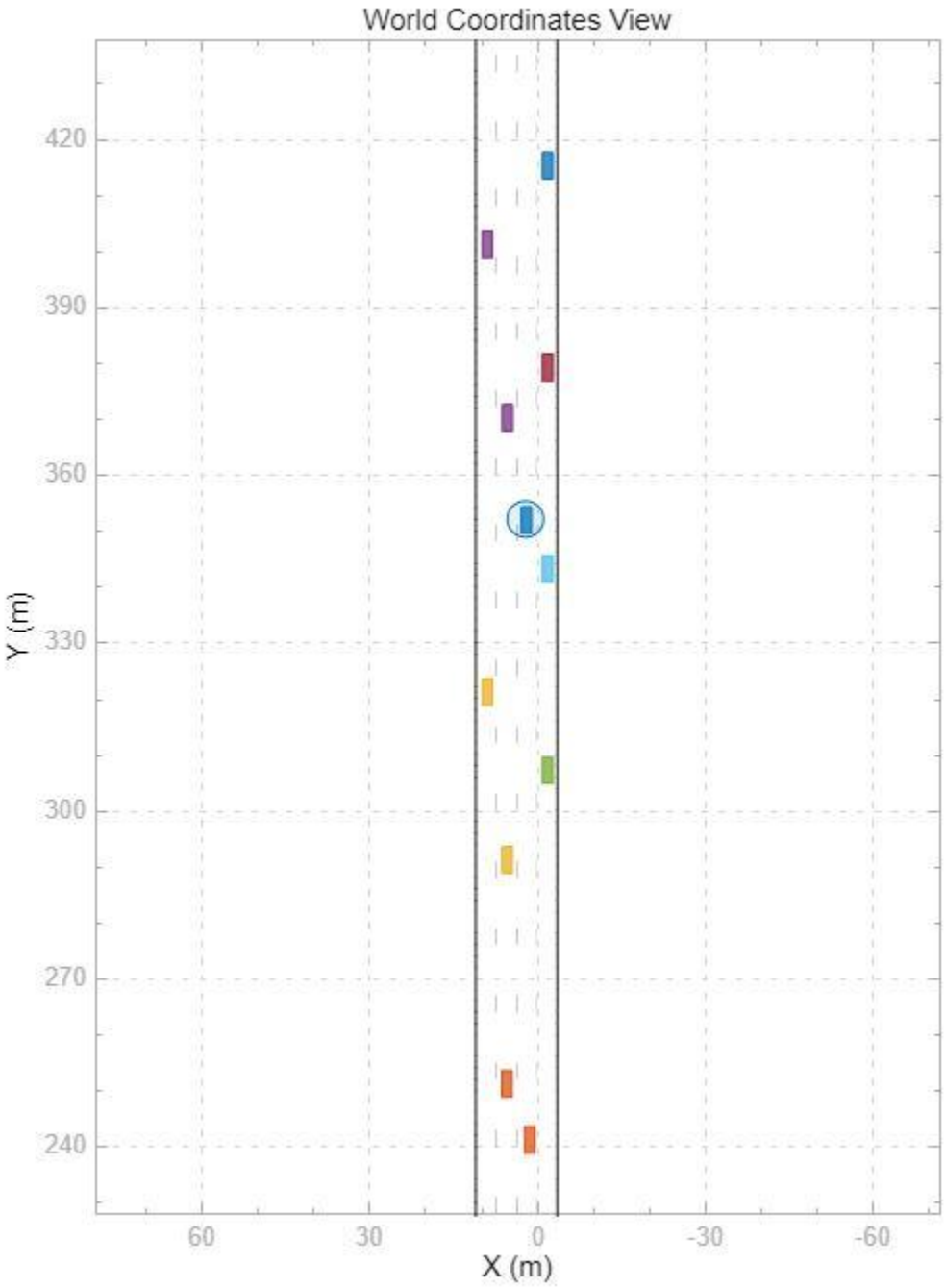

World Coordinates
300
250
200
150
100
50
0
X (m)
20
0
-20
Y (m)


Ego Perspective

During the simulation, the Bird's Eye Plot shows the planned path in blue.

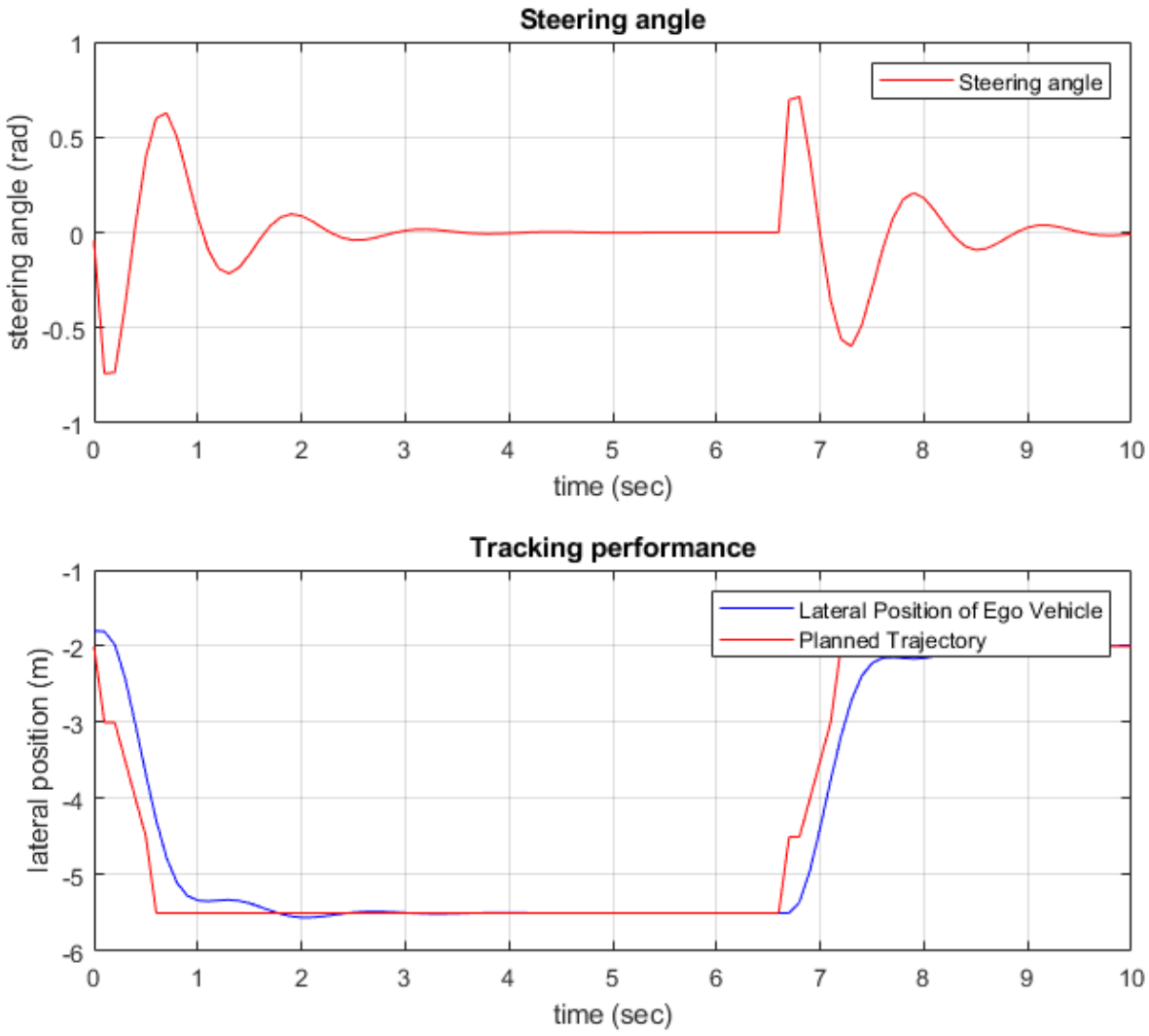


To plot the results of the simulation and depict the ego vehicle surroundings, you can also use the Bird's-Eye Scope (Automated Driving Toolbox). The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, set up the signals by clicking **Find Signals**. Once the signals are set up and the simulation is running, you can view the lane change maneuver performed by the ego vehicle in the **World Coordinates View** of the Bird's-Eye Scope.



Plot the controller performance.

```
plotLCResults
```



The figure shows the lane change performance of the controller.

- The **Steering angle** plot shows that the steering angle for the ego vehicle follows that of a standard lane change maneuver.
- The **Tracking performance** plot shows that the multistage nonlinear MPC controller does a satisfactory job tracking the lateral position of the reference path from the AStar path planner.

Run Controller for Multiple Test Scenarios

This example includes an additional test scenario. To verify the controller performance, you can test the controller for multiple scenarios and tune the controller parameters if the performance is not satisfactory. To do so:

1. Select the scenario by changing `scenarioId` in `helperLCSetUp`. To use the additional scenario, set `scenarioId = 2`.

2 Configure the simulation parameters by running `helperLCSetUp`.
3 Simulate the model with the selected scenario.
4 Evaluate the controller performance using `plotLCResults`
5 Tune the controller parameters if the performance is not satisfactory.

Conclusion

This example shows how to implement an integrated autonomous lane change controller on a straight road with a reference path generated from an AStar path planner and test it in Simulink using driving scenarios generated using Automated Driving Toolbox software.

See Also

Nonlinear MPC Controller | `nlmpc`

More About

- “Nonlinear MPC” on page 9-2
- “Lane Following Using Nonlinear Model Predictive Control” on page 9-94

Control of Quadrotor Using Nonlinear Model Predictive Control

This example shows how to design a trajectory tracking controller for a quadrotor using nonlinear model predictive control (MPC).

Quadrotor Model

The quadrotor has four rotors which are directed upwards. From the center of mass of the quadrotor, rotors are placed in a square formation with equal distance. The mathematical model for the quadrotor dynamics are derived from Euler-Lagrange equations [1].

The twelve states for the quadrotor are:

$[x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}]$,

where

- $[x, y, z]$denote the positions in the inertial frame
- Angle positions $[\phi, \theta, \psi]$ are in the order of roll, pitch, and yaw
- The remaining states are the velocities of the positions and angles

The control inputs for the quadrotor are the squared angular velocities of the four rotors:

$[\omega_1^2, \omega_2^2, \omega_3^2, \omega_4^2]$.

These control inputs create force, torque, and thrust in the direction of the body z-axis. In this example, every state is measurable, and the control inputs are constrained to be within $[0,12]$ $\left(\frac{\text{rad}}{s}\right)^2$.

The state function and state Jacobian function of the model are built and derived using Symbolic Math Toolbox™ software.

```
getQuadrotorDynamicsAndJacobian;
```

```
ans = function_handle with value:
    @QuadrotorStateFcn


ans = function_handle with value:
    @QuadrotorStateJacobianFcn
```

The `getQuadrotorDynamicsAndJacobian` script generates the following files:

- `QuadrotorStateFcn.m` — State function
- `QuadrotorStateJacobianFcn.m` — State Jacobian function

For details on either function, open the corresponding file.

Design Nonlinear Model Predictive Controller

Create a nonlinear MPC object with 12 states, 12 outputs, and 4 inputs. By default, all the inputs are manipulated variables (MVs).

```
nx = 12;
ny = 12;
```

```
nu = 4;
nlobj = nlmpc(nx, ny, nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

Specify the prediction model state function using the function name. You can also specify functions using a function handle.

```
nlobj.Model.StateFcn = "QuadrotorStateFcn";
```

Specify the Jacobian of the state function using a function handle. It is best practice to provide an analytical Jacobian for the prediction model. Doing so significantly improves simulation efficiency.

```
nlobj.Jacobian.StateFcn = @QuadrotorStateJacobianFcn;
```

Validate your prediction model, your custom functions, and their Jacobians.

```
rng(0)
validateFcns(nlobj,rand(nx,1),rand(nu,1));
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Analysis of user-provided model, cost, and constraint functions complete.
```

Specify a sample time of `0.1` seconds, prediction horizon of 18 steps, and control horizon of 2 steps.

```
Ts = 0.1;
p = 18;
m = 2;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = m;
```

Limit all four control inputs to be in the range [0,12].

```
nlobj.MV = struct('Min',{0;0;0;0},'Max',{12;12;12;12});
```

The default cost function in nonlinear MPC is a standard quadratic cost function suitable for reference tracking and disturbance rejection. In this example, the first 6 states $[x, y, z, \phi, \theta, \psi]$ are required to follow a given reference trajectory. Because the number of MVs (4) is smaller than the number of reference output trajectories (6), there are not enough degrees of freedom to track the desired trajectories for all output variables (OVs).

```
nlobj.Weights.OutputVariables = [1 1 1 1 1 1 0 0 0 0 0 0];
```

In this example, MVs also have nominal targets to keep the quadrotor floating, which can lead to conflict between the MV and OV reference tracking goals. To prioritize targets, set the average MV tracking priority lower than the average OV tracking priority.

```
nlobj.Weights.ManipulatedVariables = [0.1 0.1 0.1 0.1];
```

Also, penalize aggressive control actions by specifying tuning weights for the MV rates of change.

```
nlobj.Weights.ManipulatedVariablesRate = [0.1 0.1 0.1 0.1];
```

Closed-Loop Simulation

Simulate the system for 20 seconds with a target trajectory to follow.

```
% Specify the initial conditions
x = [7;-10;0;0;0;0;0;0;0;0;0;0];
% Nominal control that keeps the quadrotor floating
nloptions = nlmpcmoveopt;
nloptions.MVTarget = [4.9 4.9 4.9 4.9];
mv = nloptions.MVTarget;
```

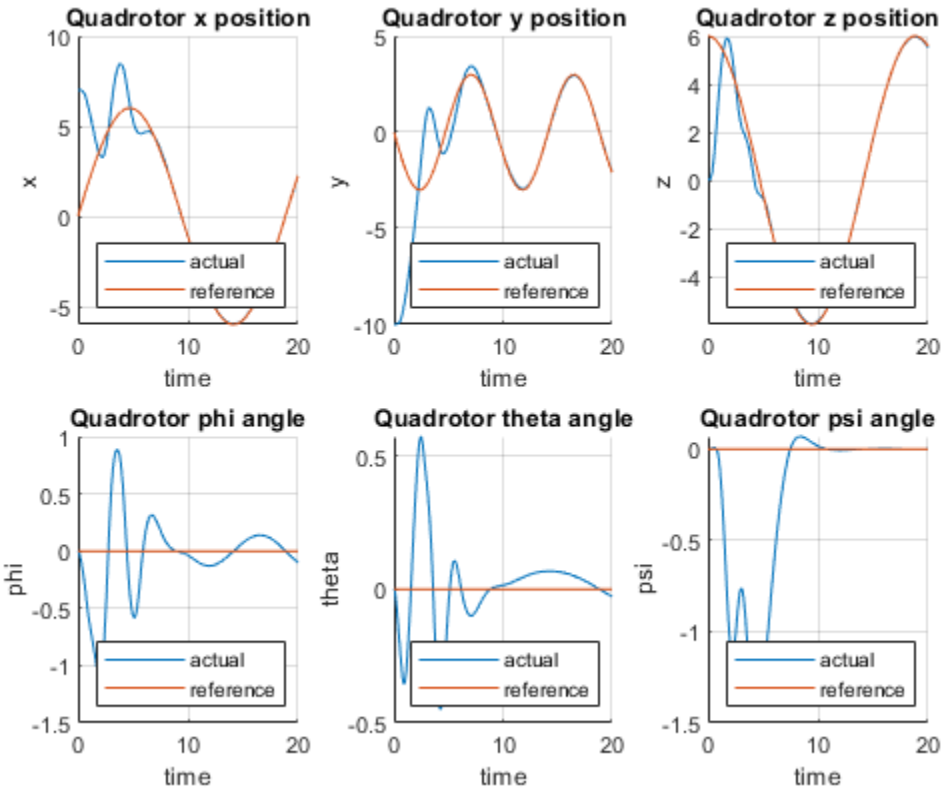
Simulate the closed-loop system using the `nlmpcmove` function, specifying simulation options using an `nlmpcmove` object.

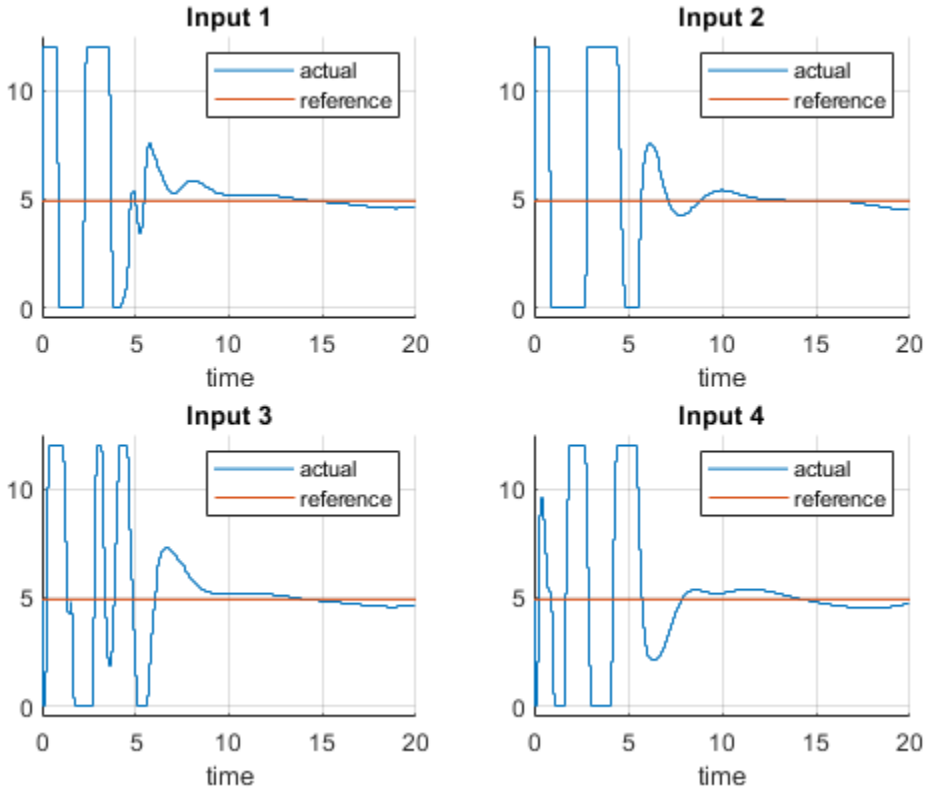
```
Duration = 20;
hbar = waitbar(0,'Simulation Progress');
xHistory = x';
lastMV = mv;
uHistory = lastMV;
for k = 1:(Duration/Ts)
    % Set references for previewing
    t = linspace(k*Ts, (k+p-1)*Ts,p);
    yref = QuadrotorReferenceTrajectory(t);
    % Compute the control moves with reference previewing.
    xk = xHistory(k,:);
    [uk,nloptions,info] = nlmpcmove(nlobj,xk,lastMV,yref',[],nloptions);
    uHistory(k+1,:) = uk';
    lastMV = uk;
    % Update states.
    ODEFUN = @(t,xk) QuadrotorStateFcn(xk,uk);
    [TOUT,YOUT] = ode45(ODEFUN,[0 Ts], xHistory(k,:)');
    xHistory(k+1,:) = YOUT(end,:);
    waitbar(k*Ts/Duration,hbar);
end
close(hbar)
```

Visualization and Results

Plot the results, and compare the planned and actual closed-loop trajectories.

```
plotQuadrotorTrajectory;
```


Quadrotor x position
x
time
actual
reference
Quadrotor y position
y
time
actual
reference
Quadrotor z position
z
time
actual
reference
Quadrotor phi angle
phi
time
actual
reference
Quadrotor theta angle
theta
time
actual
reference
Quadrotor psi angle
psi
time
actual
reference



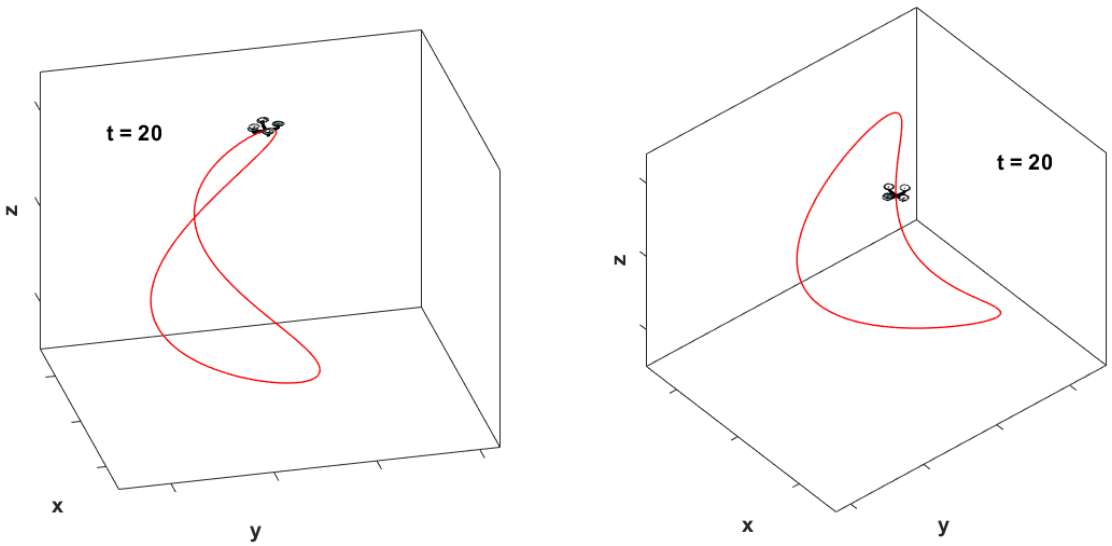
Because the number of MVs is smaller than the number of reference output trajectories, there are not enough degrees of freedom to track the desired trajectories for all OVs.

As shown in the figure for states $[x, y, z, \phi, \theta, \psi]$ and control inputs,

- The states $[x, y, z]$ match the reference trajectory very closely within 7 seconds.
- The states $[\phi, \theta, \psi]$ are driven to the neighborhood of zeros within 9 seconds.
- The control inputs are driven to the target value of 4.9 around 10 seconds.

You can animate the trajectory of the quadrotor. The quadrotor moves close to the "target" quadrotor which travels along the reference trajectory within 7 seconds. After that, the quadrotor follows closely the reference trajectory. The animation terminates at 20 seconds.

```
animateQuadrotorTrajectory;
```



Conclusion

This example shows how to design a nonlinear model predictive controller for trajectory tracking of a quadrotor. The dynamics and Jacobians of the quadrotor are derived using Symbolic Math Toolbox software. The quadrotor tracks the reference trajectory closely.

References

[1] T. Luukkonen, *Modelling and control of quadcopter*, Independent research project in applied mathematics, Espoo: Aalto University, 2011.

[2] E. Tzorakoleftherakis, and T. D. Murphey. "Iterative sequential action control for stable, model-based control of nonlinear systems." *IEEE Transactions on Automatic Control* (2018).

See Also

`nlmpc` | `nlmpcmove` | `nlmpcmoveopt`

More About

- “Nonlinear MPC” on page 9-2

Economic MPC

Economic model predictive controllers optimize control actions to satisfy generic economic or performance cost functions. The name *Economic MPC* derives from applications in which the cost function to minimize is the operating cost of the system under control.

Traditional implicit MPC controllers minimize a quadratic performance criterion (cost function) using a linear prediction model.

A quadratic cost function is adequate for tracking specified output and manipulated variable references. However, some applications can require optimizing for performance criteria, such as fuel consumption or production rates. Such performance criteria can be a combination of linear or nonlinear functions of the system states, inputs, or outputs.

An economic MPC controller:

- Can use a linear or nonlinear prediction model
- Uses your generic performance cost function instead of (or in addition to) the built-in quadratic cost function
- Computes optimal control moves by solving a nonlinear optimization problem using the SQP algorithm in `fmincon`

To implement an economic MPC controller, create a nonlinear MPC controller object, and specify:

- State and output functions that define your prediction model. For more information, see “Specify Prediction Model for Nonlinear MPC” on page 9-5.
- A generic performance-based cost function. For more information, see “Specify Cost Function for Nonlinear MPC” on page 9-12.

For more information on nonlinear MPC controller objects, see `nlmpc`.

You can simulate economic MPC controllers:

- In Simulink using the Nonlinear MPC Controller block
- At the command line using `nlmpcmove`

Note Designing an economic MPC controller using the **MPC Designer** app is not supported.

An economic MPC controller requires Optimization Toolbox software.

See Also

Functions
`nlmpc` | `nlmpcmove`

Blocks
Nonlinear MPC Controller

More About

- “Economic MPC Control of Ethylene Oxide Production” on page 9-119

Economic MPC Control of Ethylene Oxide Production

This example shows how to maximize the production of an ethylene oxide plant for profit using an economic MPC controller. This controller is implemented using a nonlinear MPC controller with a custom performance-based cost function.

Nonlinear Ethylene Oxidation Plant

Conversion of ethylene (C2H4) to ethylene oxide (C2H4O) occurs in a cooled, gas-phase catalytic reactor. Three reactions occur simultaneously in the well-mixed gas phase within the reactor:

```
C2H4 + 0.5*O2 -> C2H4O
```

```
C2H4 + 3*O2 -> 2*CO2 + 2*H2O
```

```
C2H4O + 2.5*O2 -> 2*CO2 + 2*H2O
```

The first reaction is wanted and the other two are unwanted because they reduce C2H4O production. A mixture of air and ethylene is continuously fed into the reactor. The first-principle nonlinear dynamic model of the reactor is implemented as a set of ordinary differential equations (ODEs) in the `oxidationPlantCT` function. For more information, see `oxidationPlantCT.m`.

The plant has four states:

- Gas density in the reactor (x_1)
- C2H4 concentration in the reactor (x_2)
- C2H4O concentration in the reactor (x_3)
- Temperature in the reactor (x_4)

The plant has three inputs:

- C2H4 concentration in the feed (u_1)
- Reactor cooling jacket temperature (u_2)
- C2H4 feed rate (u_3)

All variables in the model are scaled to be dimensionless and of unity order. The basic plant equations and parameters are obtained from [1] with some changes in input/output definitions and ordering.

The plant is asymptotically open-loop stable.

Control Objectives and Constraints

The primary control objective is to maximize the ethylene oxide (C2H4O) production rate (which in turn maximizes profit) at any steady-state operating point, given the availability of C2H4 in the feed stream.

The C2H4O production rate is defined as the product of the C2H4O concentration in the reactor (x_3) and the total volumetric flow rate exiting the reactor ($u_3/u_1 * x_4$).

The operating point is effectively determined by the three inputs. u_1 is the C2H4 concentration in the feed, which the MPC controller can manipulate. u_2 is the cooling jacket temperature, which keeps the temperature stable. u_3 is the C2H4 feed rate, which indicates the available ethylene coming from an upstream process. A higher feed rate increases the achievable C2H4O production rate. In this example, both u_2 and u_3 are measured disturbances.

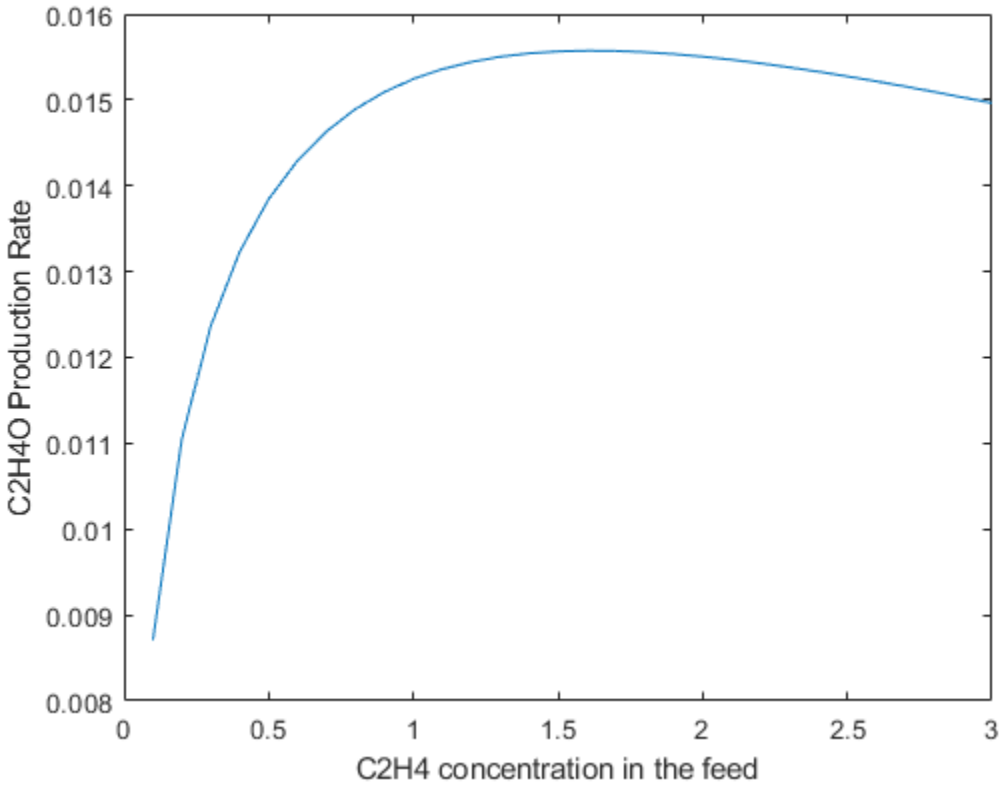
Optimal Production Rate at the Initial Operating Point

At the initial condition, the cooling jacket temperature is `1.1` and the C2H4 availability is `0.175`.

```
Tc = 1.1;
C2H4Avalability = 0.175;
```

Compute the optimal C2H4O production rate by sweeping through the operating range of the C2H4 concentration in the feed (u_1) using `fsolve`.

```
uRange = 0.1:0.1:3;
EORate = zeros(length(uRange),1);
optimopt = optimoptions('fsolve','Display','none');
for ct = 1:length(uRange)
    xRange = real(fsolve(@(x) oxidationPlantCT(x,[uRange(ct);Tc;C2H4Avalability]),rand(1,4),optim
    EORate(ct) = C2H4Avalability/uRange(ct)*xRange(3)*xRange(4);
end
figure
plot(uRange,EORate)
xlabel('C2H4 concentration in the feed')
ylabel('C2H4O Production Rate')
```



The optimal C2H4O production rate of 0.0156 is achieved at u_1 = 1.6. In other words, if the plant originally operates with a different C2H4 concentration in the feed, you expect the economic MPC controller to bring it to 1.6 such that the optimal C2H4O production rate is achieved.

Nonlinear MPC Design

Economic MPC can be implemented with a nonlinear MPC controller. The prediction model has four states and three inputs (one MV and two MDs). In this example, since you do not need an output function, assume y = x.

```
nlobj = nlmpc(4,4,'MV',1,'MD',[2 3]);
nlobj.States(1).Name = 'Den';
nlobj.States(1).Name = 'C2H4';
nlobj.States(1).Name = 'C2H4O';
nlobj.States(1).Name = 'Tc';
nlobj.MV.Name = 'CEin';
nlobj.MD(1).Name = 'Tc';
nlobj.MD(2).Name = 'Availability';
```

```
In standard cost function, zero weights are applied by default to one or more OVs because there a
```

The nonlinear plant model is defined in `oxidationPlantDT`. It is a discrete-time model where a multistep explicit Euler method is used for integration. While this example uses a nonlinear plant model, you can also implement economic MPC using linear plant models.

```
nlobj.Model.StateFcn = 'oxidationPlantDT';
nlobj.Model.IsContinuousTime = false;
```

In general, to improve computational efficiency, it is best practice to provide an analytical Jacobian function for the prediction model. In this example, you do not provide one because the simulation is fast enough.

The relatively large sample time of 25 seconds used here is appropriate when the plant is stable and the primary objective is economic optimization. Prediction horizon is 2, which gives a prediction time is 50 seconds.

```
Ts = 25;
nlobj.Ts = Ts;                                  % Sample time
nlobj.PredictionHorizon = 2;                    % Prediction horizon
nlobj.ControlHorizon = 1;                       % Control horizon
```

All the states in the prediction model must be positive based on first principles. Therefore, specify a minimum bound of zero for all states.

```
nlobj.States(1).Min = 0;
nlobj.States(2).Min = 0;
nlobj.States(3).Min = 0;
nlobj.States(4).Min = 0;
```

Plant input u_1 must stay within saturation limits between `0.1` and 3.

```
nlobj.MV.Min = 0.1;
nlobj.MV.Max = 3;
```

The rates of change of u_1 are also limited to +/- `0.02`/sec.

```
nlobj.MV.RateMin = -0.02*Ts;
nlobj.MV.RateMax = 0.02*Ts;
```

Custom Cost Function for Economic MPC

Instead of using the standard quadratic objective function, a custom cost function is used as the replacement. You want to maximize the C2H4O production rate at the end of the prediction horizon.

```
f = -(u3/u1*x3*x4)
```

The negative sign in `f` is used to maximize production, since the controller minimizes `f` during optimization. For more information, see `oxidationCostFcn.m`.

```
nlobj.Optimization.CustomCostFcn = 'oxidationCostFcn';
nlobj.Optimization.ReplaceStandardCost = true;
```

Validate Custom Functions

Assume that the plant initially operates at `u1 = 0.5`.

```
u0 = 0.5;
```

Find the states at the steady state using `fsolve`.

```
x0 = real(fsolve(@(x) oxidationPlantCT(x,[u0;Tc;C2H4Avalability]),rand(1,4),optimopt));
```

The C2H4O production rate is 0.0138, far away from the optimal condition of 0.0156.

```
EORate0 = C2H4Avalability/u0*x0(3)*x0(4);
```

Validate the state function and cost function at the initial condition.

```
validateFcns(nlobj,x0,u0,[Tc C2H4Avalability]);
```

```
Model.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

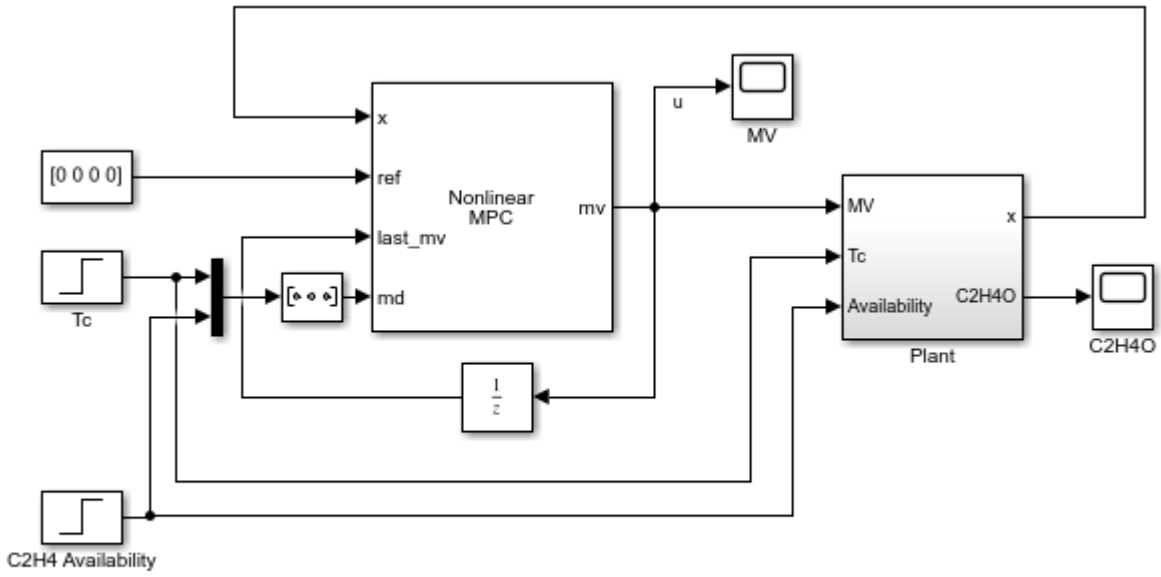
You can compute the first move using the `nlmpcmove` function. It returns an MV of 1.0, indicating that economic MPC will increase the MV from 0.5 to 1, limited by the manipulated variable rate constraint.

```
mv = nlmpcmove(nlobj,x0,u0,zeros(1,4),[Tc C2H4Avalability]);
```

Simulink Model with Economic MPC Controller

Open the Simulink model.

```
mdl = 'mpc_economicEO';
open_system(mdl)
```



Copyright 2017-2018 The MathWorks Inc.

The cooling jacket temperature is initially `1.1` and remains constant for the first 100 seconds. It then increases to `1.15`, and therefore, reduces the optimal C2H4O production rate from `0.0156` to `0.0135`.

The C2H4 availability is initially `0.175` and remains constant for the first 200 seconds. It then increases to `0.25`, and therefore, increases the optimal C2H4O production rate from `0.0135` to `0.0195`.

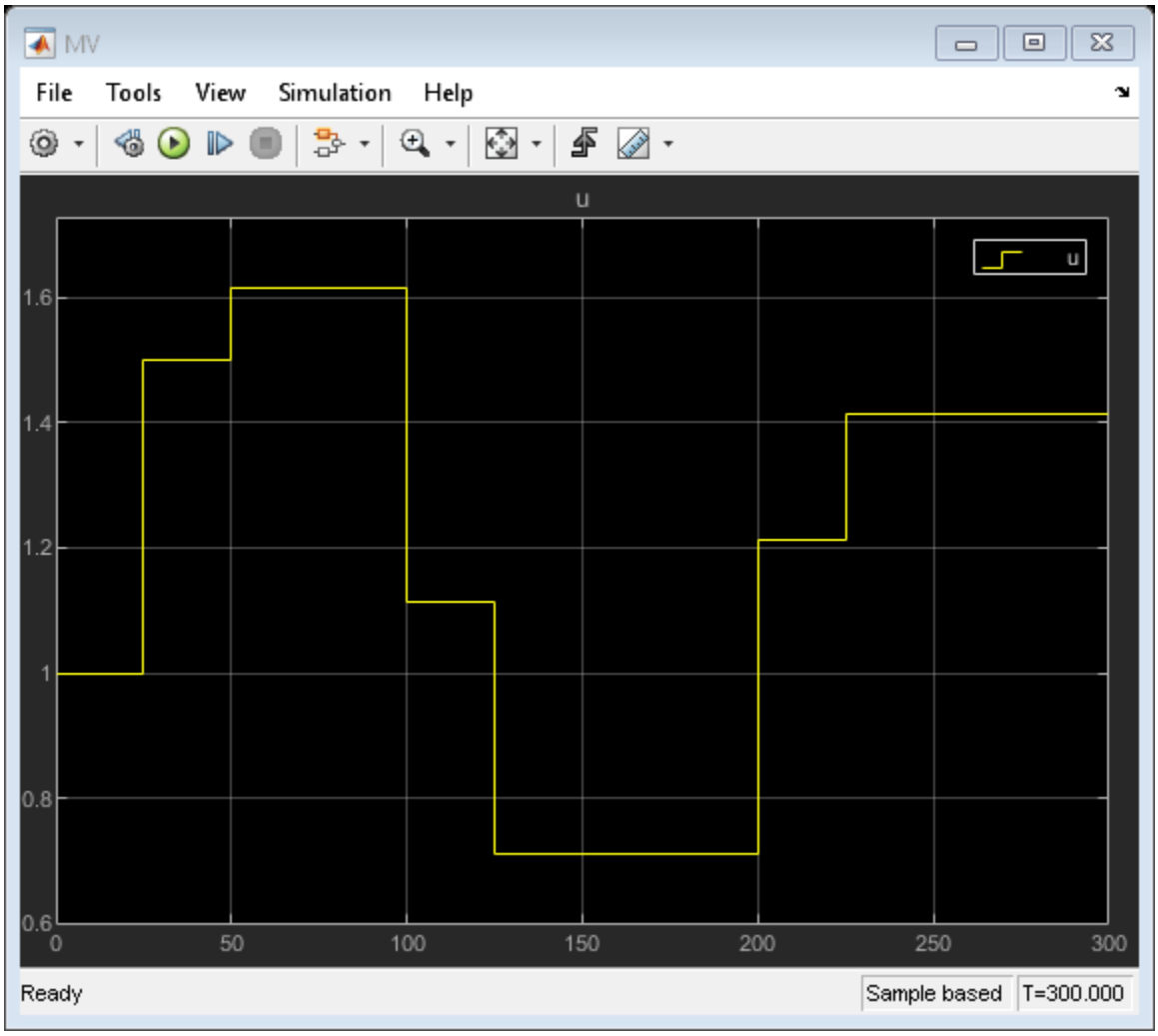
The model includes constant (zero) references for the four plant outputs. The Nonlinear MPC Controller block requires these reference signals, but they are ignored in the custom cost function.

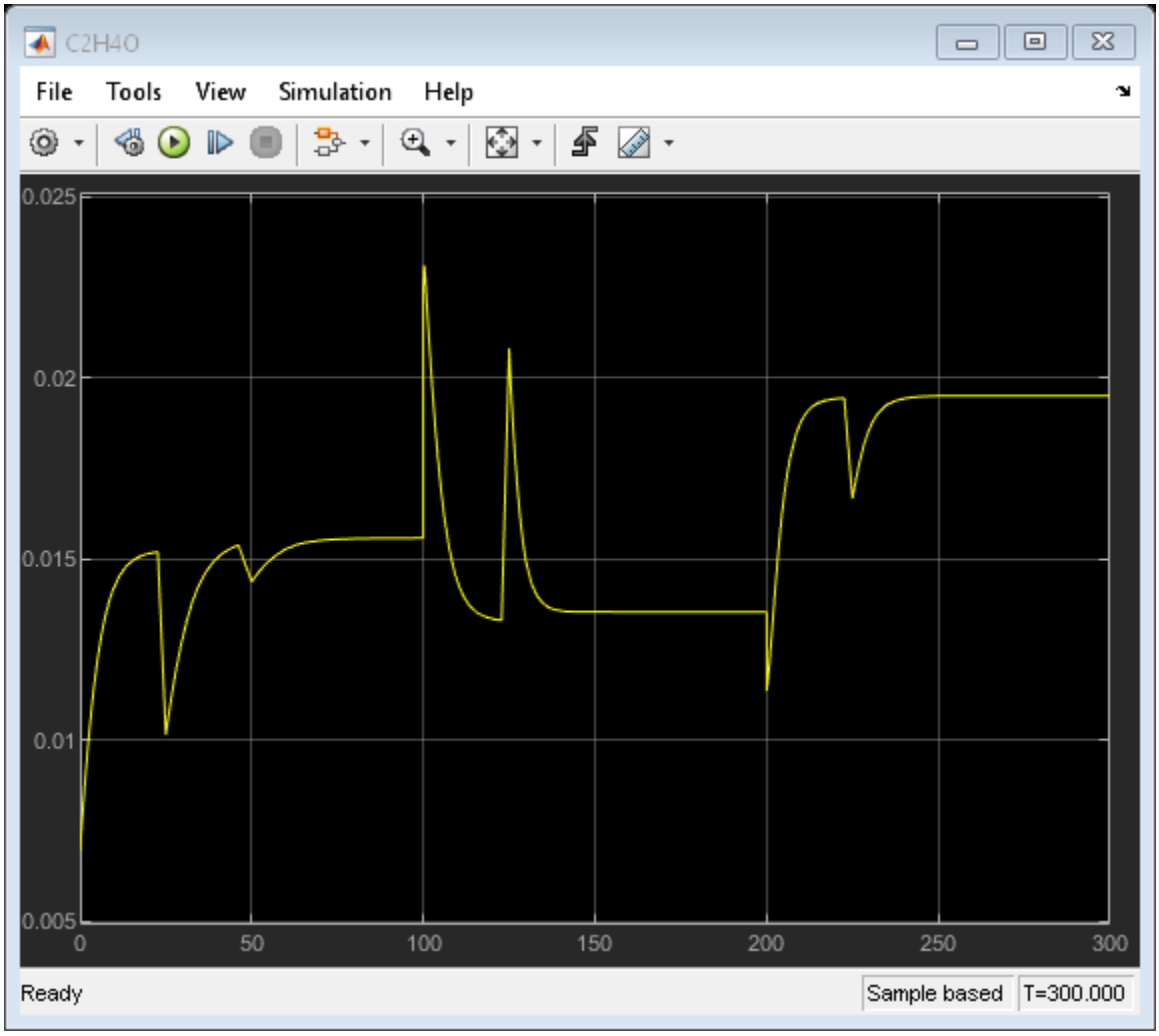
The Plant subsystem calculates the plant states by integrating the ODEs in `oxidationPlantCT.m`. Assume all the states are measurable such that you do not need to implement a nonlinear state estimator in this example. The `C2H4O` plant output is the instantaneous C2H4O production rate, which is used for display purposes.

Simulate Model and Analyze Results

Run the simulation.

```
open_system([mdl '/MV']);
open_system([mdl '/C2H4O']);
sim(mdl)
```


MV
File
Tools
View
Simulation
Help
u
u
1.6
1.4
1.2
1
0.8
0.6
0
50
100
150
200
250
300
Ready
Sample based
T=300.000



Because the C2H4O plant operating at the initial condition is not optimal, its profit can be improved. In the first 100 seconds, the economic MPC controller gradually moves the plant to the true optimal condition under the same cooling jacket temperature and C2H4 availability constraints. It improves C2H4O production rate by:

$$\left(0.0156 - 0.0138\right)/0.0138 = 13\%$$

which could be worth millions of dollars per year in large-scale production.

In the next 100 seconds, the cooling jacket temperature increases from `1.1` to `1.15`. The economic MPC controller moves the plant smoothly to the new optimal condition `0.0135` as expected.

In the next 100 seconds, the C2H4 availability increases from `0.175` to `0.25`. The economic MPC controller is again able to move the plant the new optimal steady state `0.0195`.

Close the Simulink model.

```
bdclose(mdl)
```

References

[1] H. Durand, M. Ellis, P. D. Christofides. "Economic model predictive control designs for input rate-of-change constraint handling and guaranteed economic performance." *Computers and Chemical Engineering*. Vol. 92,2016, pp 18-36.

See Also

More About

- “Economic MPC” on page 9-117

Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC

This example shows how to use multistage nonlinear model predictive control (MPC) as a path planner to find the optimal trajectory to automatically park a truck and trailer system in the presence of static obstacles.

Overview

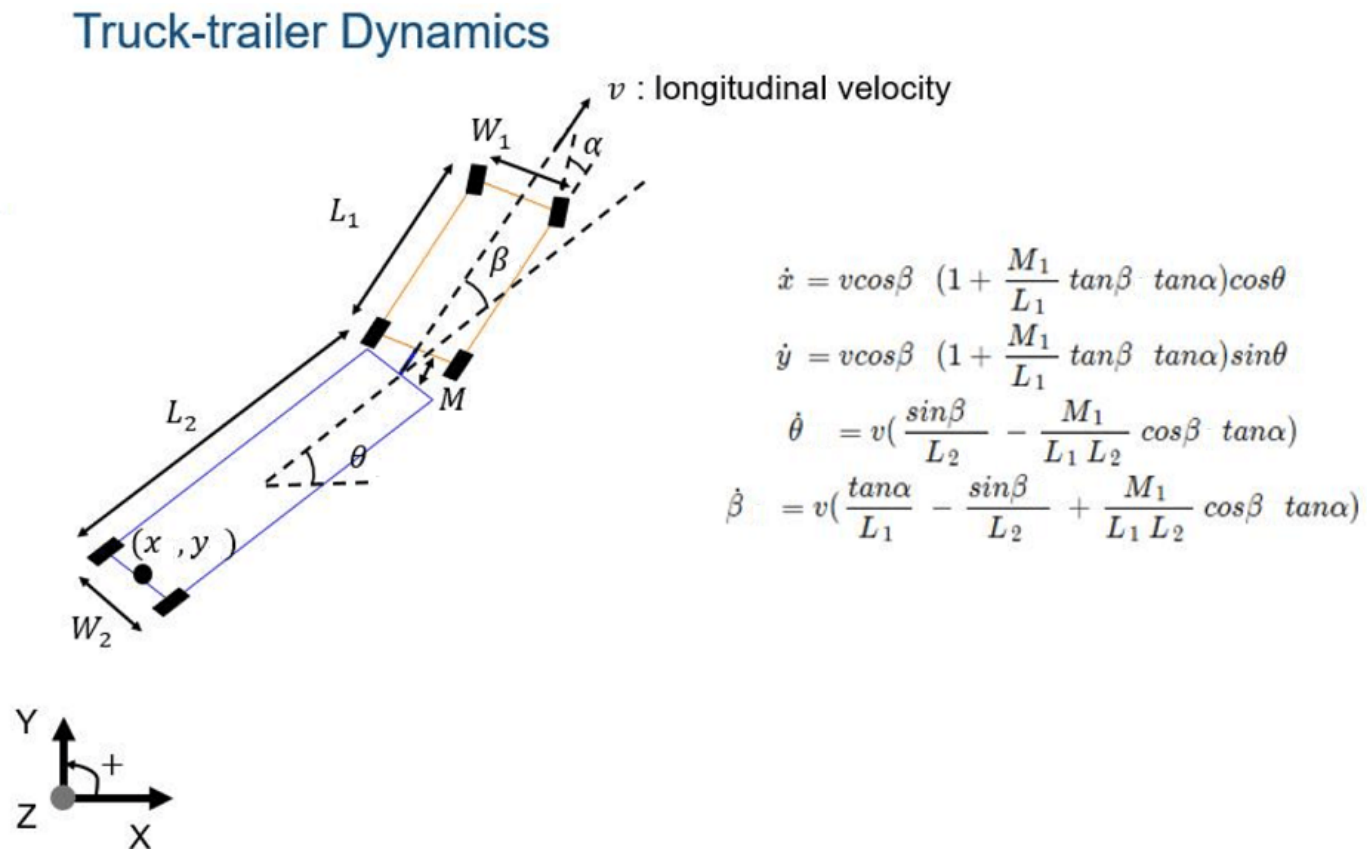
An MPC controller uses an internal model to predict plant behavior. Given the current states of the plant, based on the prediction model, MPC finds an optimal control sequence that minimizes cost and satisfies the constraints specified across the prediction horizon. Because MPC finds the plant future trajectory at the same time, it can work as a powerful tool to solve trajectory optimization problems, such as autonomous parking of a vehicle and motion planning of a robot arm.

In such trajectory optimization problems the plant, cost function, and constraints can often be nonlinear. Therefore, you need to use nonlinear MPC controller for problem formulation and solution. In this example, you design a nonlinear MPC controller that finds an optimal route to automatically park a truck with a single trailer from its initial position to its target position, which is between two static obstacles. You can then pass the generated path to a low-level controller as a reference signal, so that it can execute the parking maneuver in real time.

This example requires Optimization Toolbox™ and Robotics System Toolbox™ software.

Truck and Single Trailer System

The following figure shows the truck and trailer nonlinear dynamic system.



The states for this model are:

1. `x` (center of the trailer rear axle, global x position)
2. `y` (center of the trailer rear axle, global y position)
3. `theta` (trailer orientation, global angle, 0 = east)
4. `beta` (truck orientation with respect to trailer, 0 = aligned)

The inputs for this model are:

1. `alpha` (truck steering angle)
2. `v` (truck longitudinal velocity)

For this model, length and position are in meters, velocity is in m/s, and angles are in radians.

Define the following model parameters.

- `M` (hitch length)
- `L1` (truck length)
- `W1` (truck width)
- `L2` (trailer length)
- `W2` (trailer width)
- `Lwheel` (wheel diameter)

- `Wwheel` (wheel width)

```
params = struct('M', 1, ...
    'L1',6,...
    'W1',2.5,...
    'L2',10,...
    'W2',2.5,...
    'Lwheel',1,...
    'Wwheel',0.4);
```

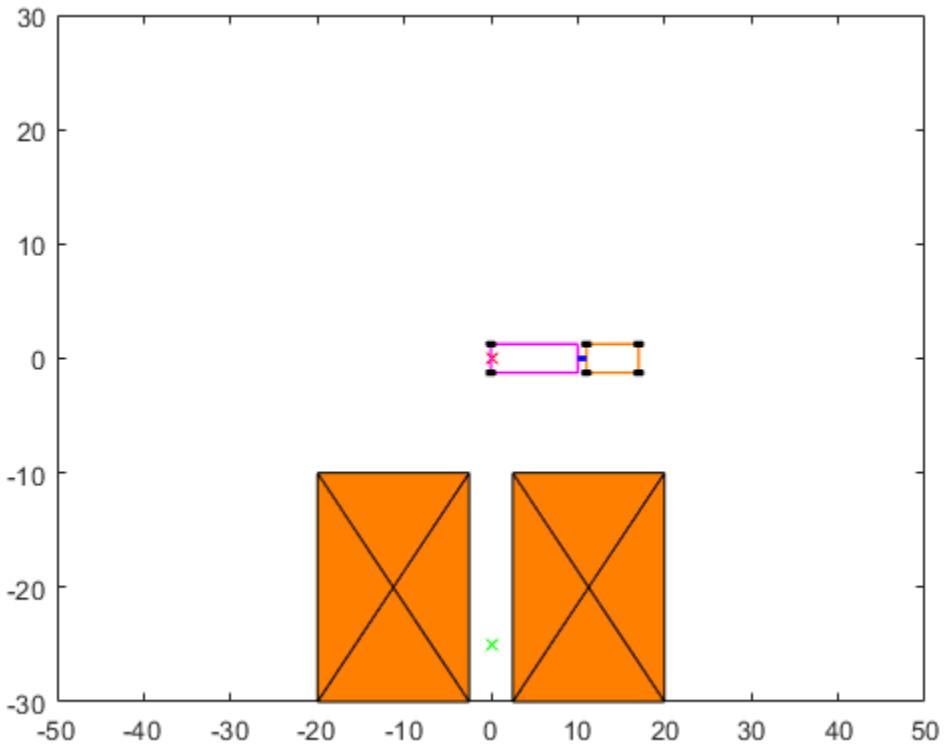
The nonlinear model is implemented in the `TruckTrailerStateFcn` function and its manually-derived analytical Jacobian (which is used to speed up optimization) is in the `TruckTrailerStateJacobianFcn` function .

In this example, since you use MPC as a path planner instead of a low-level path-following controller, the truck's longitudinal velocity is used as one of the manipulated variables instead of the acceleration.

Automatic Parking Problem

The parking lot is 100 meters wide and 60 meters long. The goal is to find a viable path that brings the truck and trailer system from any initial position to the target position (the green cross in the following figure) in 20 seconds using reverse parking. In the process, the planned path must avoid collisions with two obstacles next to the parking spot.

```
initialPose = [0;0;0;0];
targetPose = [0;-25;pi/2;0];
TruckTrailerPlot(initialPose,targetPose,params);
```



The initial plant inputs (steering angle and longitudinal velocity) are both `0`.

```
u0 = zeros(2,1);
```

The initial position must be valid. Use the inequality constraint function `TruckTrailerIneqConFcn` to check for validity. Details about this function are discussed in the next section. Here, collision detection is carried out by specific Robotics System Toolbox functions.

```
cineq = TruckTrailerIneqConFcn(1,initialPose,u0,...
    [params.M;params.L1;params.L2;params.W1;params.W2]);
if any(cineq>0)
    fprintf('Initial pose is not valid.\n');
    return
end
```

Path Planning Using Multistage Nonlinear MPC

Compared with the generic nonlinear MPC controller (`nlmpc` object), multistage nonlinear MPC provides you with a more flexible and efficient way to implement MPC with staged costs and constraints. This flexibility and efficiency is especially useful for trajectory planning.

A multistage nonlinear MPC controller with prediction horizon `p` defines `p+1` stages, representing time `k` (current time), `k+1`, ..., `k+p`. For each stage, you can specify stage-specific cost, inequality constraint, and equality constraint functions. These functions depend only on the plant state and input values at that stage. Given the current plant states `x[k]`, MPC finds the manipulated variable (MV) trajectory (from time `k` to `k+p-1`) to optimize the summed stage costs (from time `k` to `k+p`), satisfying all the stage constraints (from time `k` to `k+p`).

In this example, the plant has four states and two inputs (both MVs). Choose the prediction horizon `p` and sample time `Ts` such that p*Ts = 20.

Create the multistage nonlinear MPC controller.

```
p = 40;
nlobj = nlmpcMultistage(p,4,2);
nlobj.Ts = 0.5;
```

Specify the prediction model and its analytical Jacobian in the controller object. Since the model requires three parameters (`M`, `L1`, and `L2`), set `Model.ParameterLength` to `3`.

```
nlobj.Model.StateFcn = "TruckTrailerStateFcn";
nlobj.Model.StateJacFcn = "TruckTrailerStateJacobianFcn";
nlobj.Model.ParameterLength = 3;
```

Specify hard bounds on the two manipulated variables. The steering angle must remain in the range +/- 45 degrees. The maximum forward speed is 10 m/s and the maximum reverse speed is 10 m/s.

```
nlobj.MV(1).Min = -pi/4;     % Minimum steering angle
nlobj.MV(1).Max =  pi/4;     % Maximum steering angle
nlobj.MV(2).Min = -10;       % Minimum velocity (reverse)
nlobj.MV(2).Max =  10;       % Maximum velocity (forward)
```

Specify hard bounds on the fourth state, which is the angle between truck and trailer. It cannot go beyond +/-90 degrees due to mechanics limitations.

```
nlobj.States(4).Min = -pi/2;
nlobj.States(4).Max = pi/2;
```

You can use different ways to define the cost terms. For example, you might want to minimize time, fuel consumption, or parking speed. For this example, minimize parking speed in the quadratic format to promote safety.

Since MVs are only valid from stage 1 to stage `p`, you do not need to define any stage cost for the last stage, `p+1`. The five model settings, `M`, `L1`, `L2`, `W1`, and `W` are stage parameters for stages 1 to `p` and are used by the inequality constraint functions.

```
for ct=1:p
    nlobj.Stages(ct).CostFcn = "TruckTrailerCost";
    nlobj.Stages(ct).CostJacFcn = "TruckTrailerCostGradientFcn";
    nlobj.Stages(ct).ParameterLength = 5;
end
```

You can use inequality constraints to avoid collision during the parking process. Use the `TruckTrailerIneqConFcn` function to check whether the truck or the trailer collide with any of the two static obstacles at a specific stage. When either the truck or the trailer gets within the 1 m safety zone around the obstacles, a collision is detected.

In general, check for such collisions for all the prediction steps (from stage 2 to stage `p+1`). In this example, however, you only need to check stages from 2 to `p` because the last stage is already taken care of by the equality constraint function.

For this example, do not provide an analytical Jacobian for the inequality constraint functions because it is too complicated to derive manually. Therefore, the controller uses a finite-difference method (numerical perturbation) to estimate the Jacobian at run time.

```
for ct=2:p
    nlobj.Stages(ct).IneqConFcn = "TruckTrailerIneqConFcn";
end
```

Use terminal state for the last stage to ensure successful parking at the target position. In this example, the target position is provided as a run-time signal. Here we use a dummy finite value to let MPC know which states will have terminal values at run time.

```
nlobj.Model.TerminalState = zeros(4,1);
```

At the end of multistage nonlinear MPC design, you can use the `validateFcns` command with random initial plant states and inputs to check whether any of the user-defined state, cost, and constraint function as well as any analytical Jacobian function, has a problem.

You must provide all the defined state functions and stage parameters to the controller at run time. `StageParameter` contains all the stage parameters stacked into a single vector. We also use `TerminalState` to specify terminal state at run time.

```
simdata = getSimulationData(nlobj,'TerminalState');
simdata.StateFcnParameter = [params.M;params.L1;params.L2];
simdata.StageParameter = repmat([params.M;params.L1;params.L2;params.W1;params.W2],p,1);
simdata.TerminalState = targetPose;
validateFcns(nlobj,[2;3;0.5;0.4],[0.1;0.2],simdata);
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
"CostJacFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
"IneqConFcn" of the following stages [2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 2
Analysis of user-provided model, cost, and constraint functions complete.
```

Since the default nonlinear programming solver `fmincon` searches for a local minimum, you must provide a good initial guess for the decision variables, especially for trajectory optimization problems that usually involve a complicated (likely nonconvex) solution space.

This example has 244 decision variables, the plant states and inputs (6 in total) for each of the first p (40) stages and plant states (4) for the last stage `p+1`. The `TruckTrailerInitialGuess` function uses simple heuristics to generate the initial guess. For example, if the truck and trailer is initially positioned above the obstacles, it generates an initial guess with one waypoint; otherwise, it uses two waypoints (a waypoint is an intermediate point on a route of travel at which course is changed). The initial guess is displayed as dots in the following animation plot.

```
[simdata.InitialGuess, XY0] = TruckTrailerInitialGuess(initialPose,targetPose,u0,p);
```

Trajectory Planning and Simulation Result

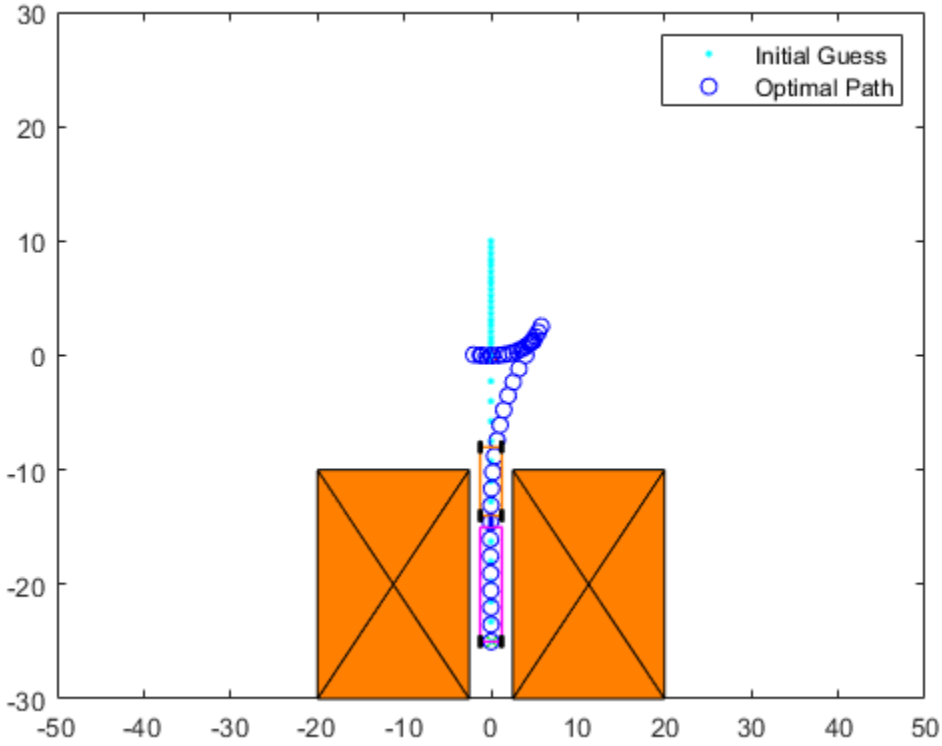
Use the `nlmpcmove` function to find the optimal parking path, which typically takes ten to twenty seconds, depending on the initial position.

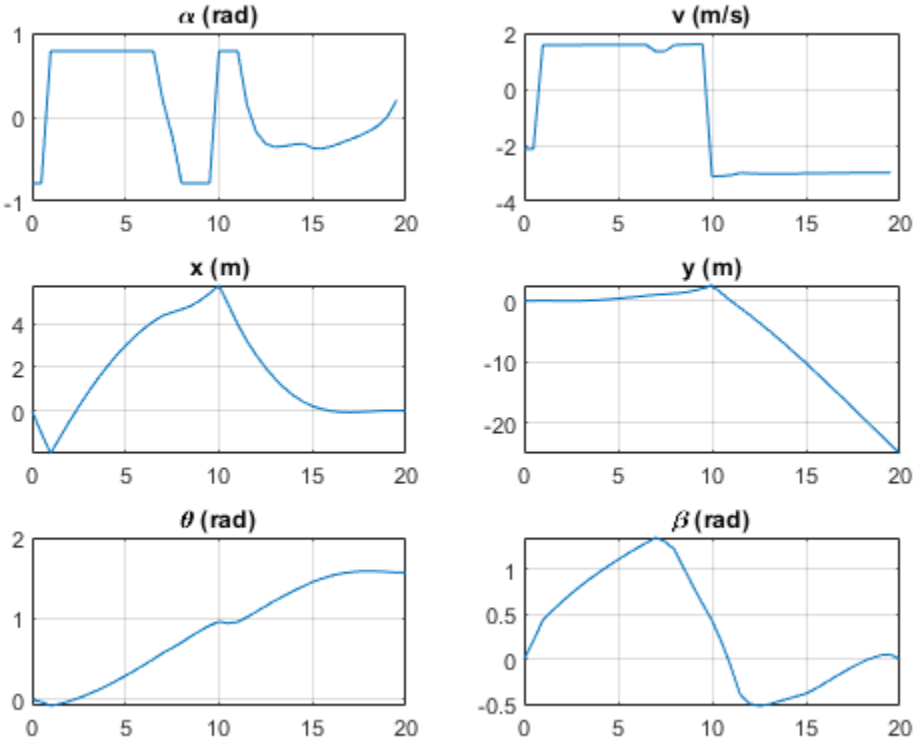
```
fprintf('Automated Parking Planner is running...\n');
tic;[~,~,info] = nlmpcmove(nlobj,initialPose,u0,simdata);t=toc;
fprintf('Calculation Time = %s; Objective cost = %s; ExitFlag = %s; Iterations = %s\n',...
    num2str(t),num2str(info.Cost),num2str(info.ExitFlag),num2str(info.Iterations));
```

```
Automated Parking Planner is running...
Calculation Time = 13.2531; Objective cost = 248.5784; ExitFlag = 1; Iterations = 51
```

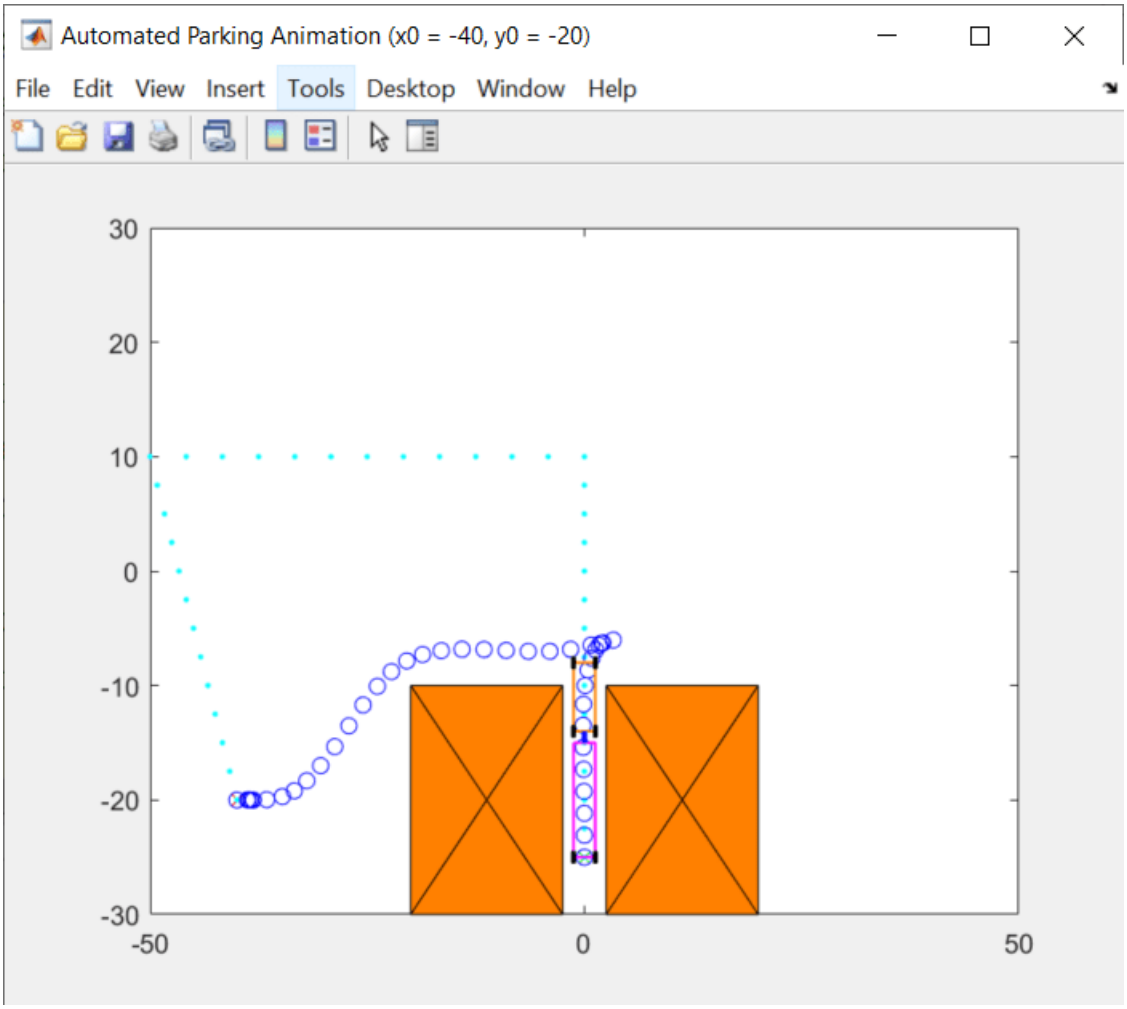
Two plots are generated. One is the animation of the parking process, where blue circles indicate the optimal path and the initial guess is shown as a dot. The other displays the optimal trajectory of plant states and control moves.

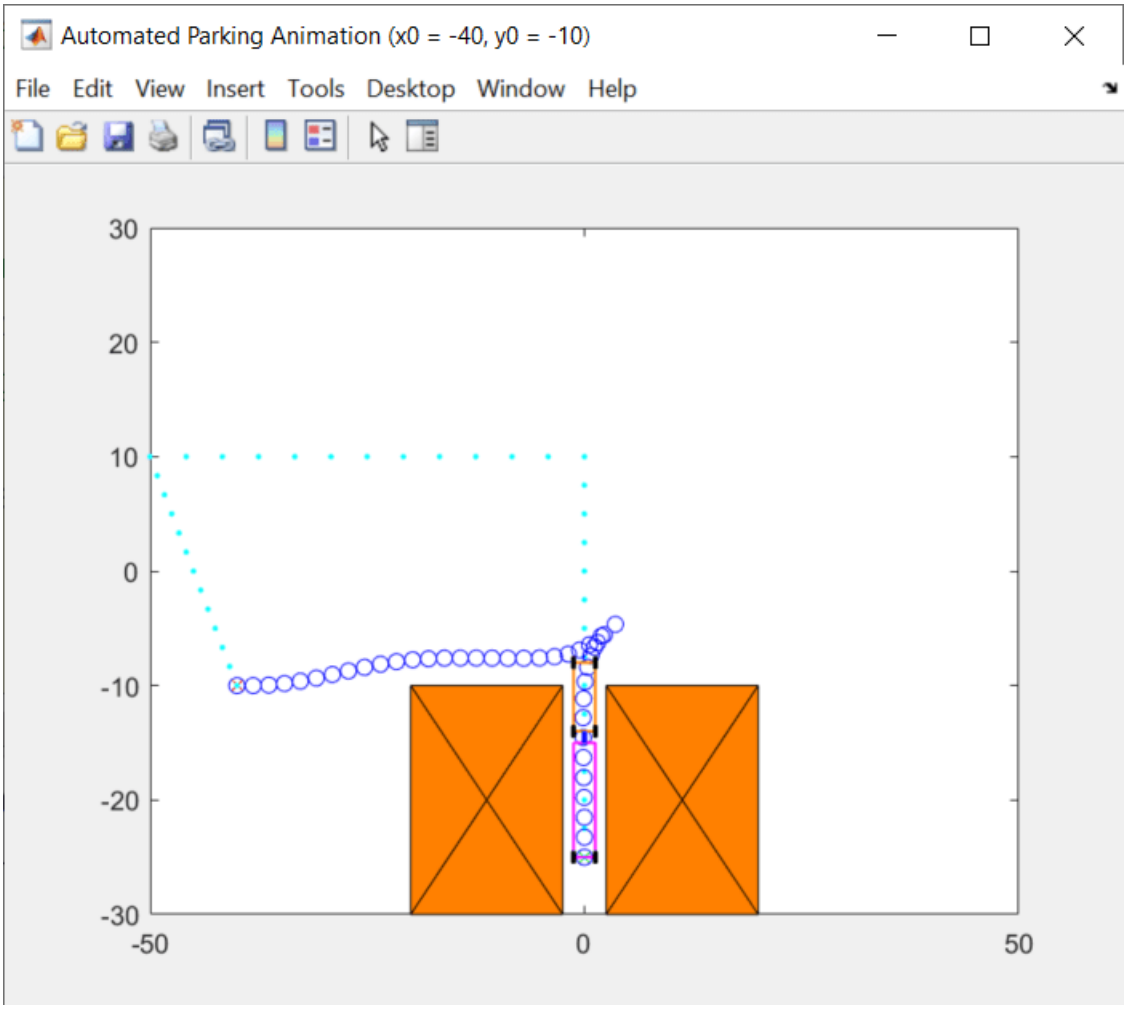
```
TruckTrailerPlot(initialPose, targetPose, params, info, XY0);
```

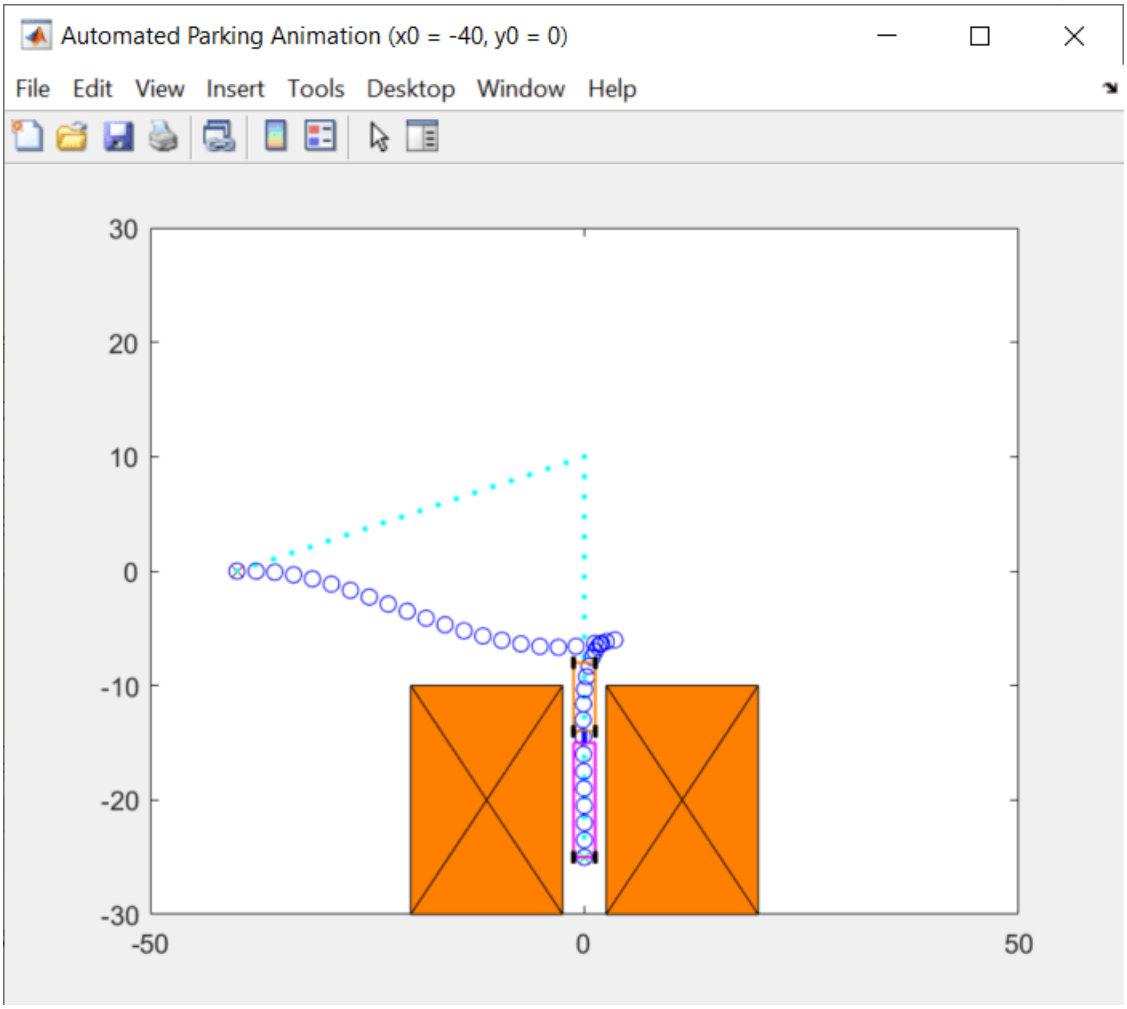


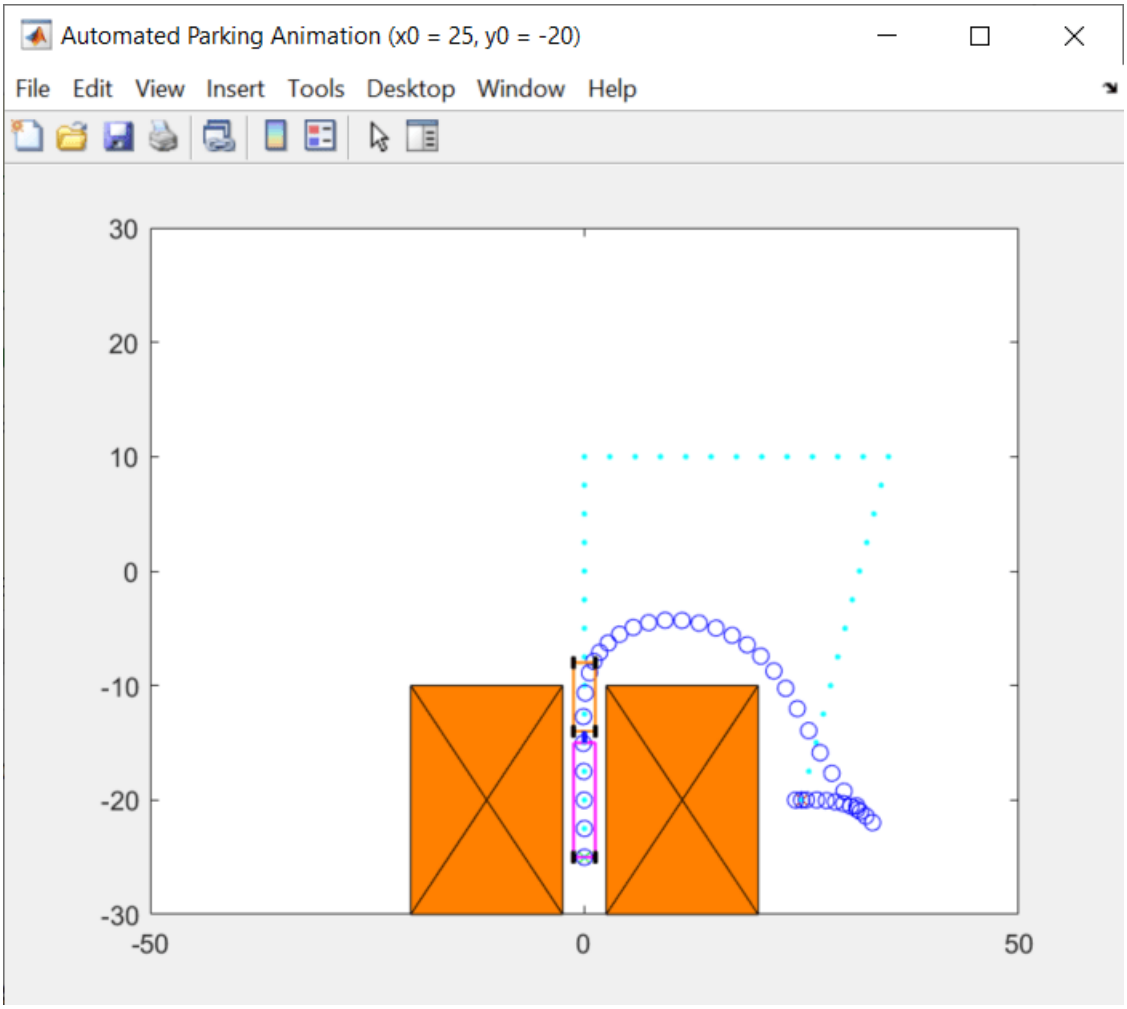


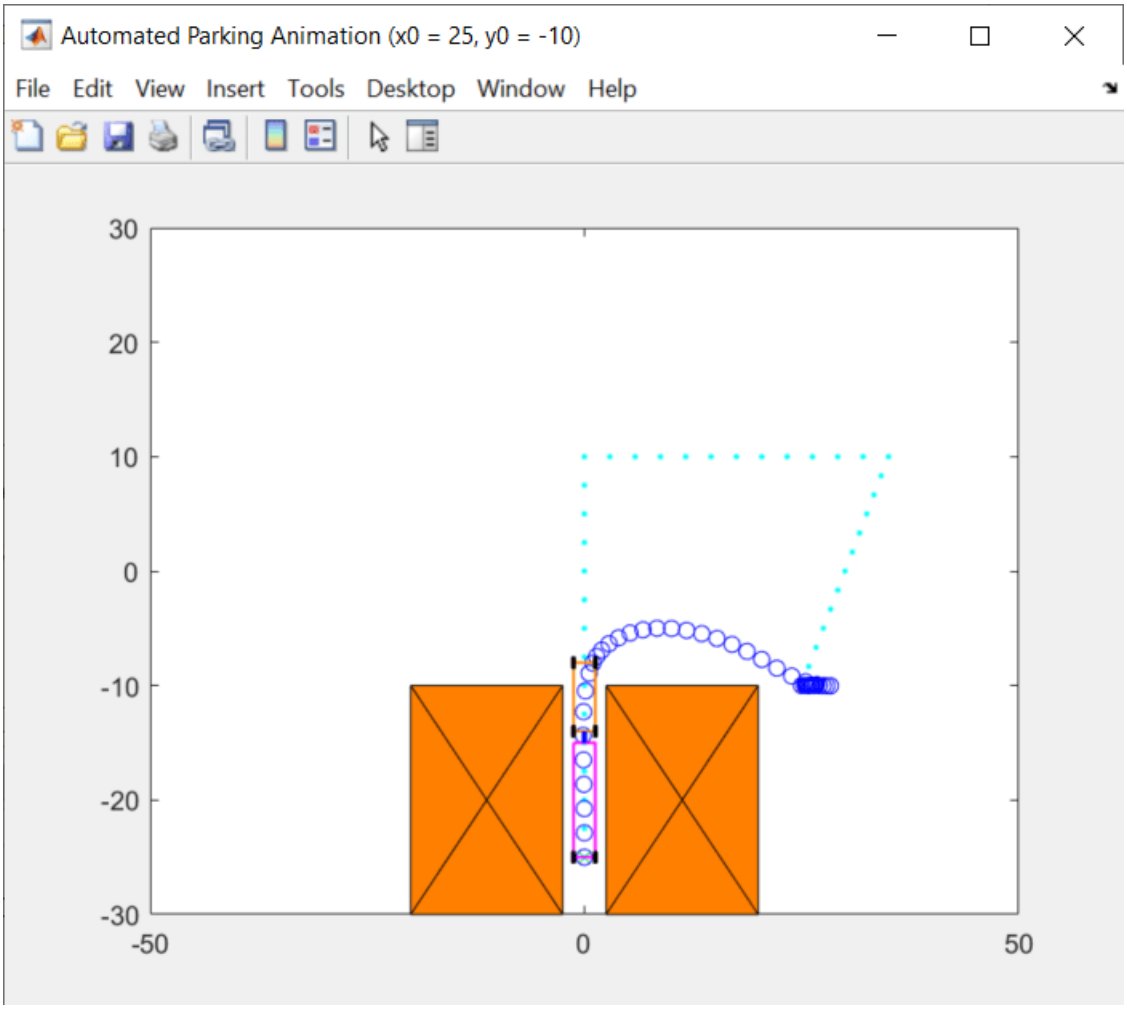
The following screenshots show the optimal trajectories found by MPC for different initial positions.

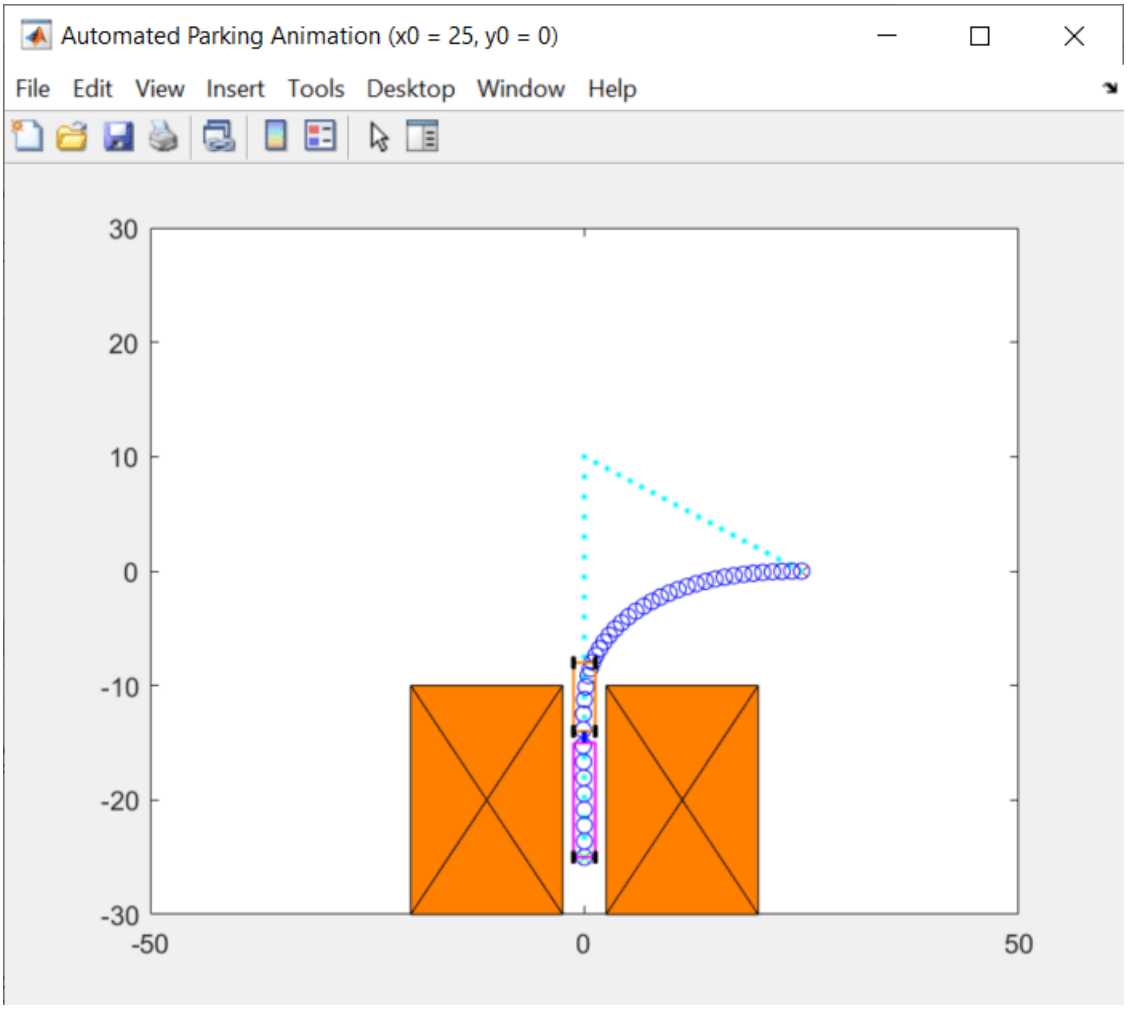

Automated Parking Animation (x0 = -40, y0 = -20)
File Edit View Insert Tools Desktop Window Help
30
20
10
0
-10
-20
-30
-50
0
50


Automated Parking Animation (x0 = -40, y0 = -10)
File Edit View Insert Tools Desktop Window Help
30
20
10
0
-10
-20
-30
-50
0
50


Automated Parking Animation (x0 = -40, y0 = 0)
File Edit View Insert Tools Desktop Window Help
30
20
10
0
-10
-20
-30
-50
0
50


Automated Parking Animation (x0 = 25, y0 = -20)
File Edit View Insert Tools Desktop Window Help
30
20
10
0
-10
-20
-30
-50
0
50


Automated Parking Animation (x0 = 25, y0 = -10)
File Edit View Insert Tools Desktop Window Help
30
20
10
0
-10
-20
-30
-50
0
50



You can try other initial X-Y positions in the Automatic Parking Problem section by changing the first two parameters of `initialPose`, as long as the positions are valid.

If `ExitFlag` is negative, the nonlinear MPC controller fails to find an optimal solution and you cannot trust the returned trajectory. In that case, you might need to provide a better initial guess and specify it in `simdata.InitialGuess` before calling `nlmpcmove`.

See Also

`nlmpcMultistage` | `nlmpcmove`

Related Examples

- “Nonlinear MPC” on page 9-2

Land a Rocket Using Multistage Nonlinear MPC

This example shows how to use a multistage nonlinear MPC controller as a planner to find an optimal path that safely lands a rocket on the ground and then use another multistage nonlinear MPC controller as a feedback controller to follow the generated path and carry out the landing maneuver.

The environment in this example is a 3-DOF rocket represented by a circular disc with mass. The rocket has two thrusters for forward and rotational motion. Gravity acts vertically downwards, and there are no aerodynamic drag forces. The goal is to first find a path that can safely land the robot on the ground at a specified location offline and then execute the landing maneuver at run time.

In this planning and control problem:

- Motion of the rocket is bounded in X (horizontal axis) from -100 to 100 meters and Y (vertical axis) from 0 to 120 meters.
- The goal position is at (0,0) with orientation at 0 radians.
- The maximum thrust applied by each thruster can be preconfigured.
- The rocket can have an arbitrary initial position and orientation.

```
x0 = [-30;60;0;0;0;0];
u0 = [0;0];
```

Obtain Nonlinear Dynamic Model of the Rocket

The first-principle nonlinear dynamic model of the rocket has six states and 2 inputs. Both inputs are manipulated variables.

States:

1. `x`: Horizontal position of the center of gravity in meters
2. `y`: Vertical position of the center of gravity in meters
3. `theta`: Tilt angle with respect to the center of gravity in radians
4. `dxdt`: Horizontal velocity in meters per second
5. `dydt`: Vertical velocity in meters per second
6. `dthetadt`: Angular velocity in radians per second

Inputs: # thrust on the left, in Newtons # thrust on the right, in Newtons

The continuous-time model of the rocket is implemented with the `RocketStateFcn` function. To speed up optimization, its analytical Jacobian is manually derived in the `RocketStateJacobianFcn` function. The model is valid only if the rocket is above or at the ground ($y \geq 10$).

At run time, you assume that all the states are measurable and there is a sensor reading that detects a rough landing (`-1`), soft landing (`1`), or airborne (`0`) condition.

Design Planner and Find Optimal Landing Path

MPC uses an internal model to predict plant behavior in the future. Given the current states of the plant, based on the prediction model, MPC finds an optimal control sequence that minimizes the cost and satisfies all the constraints specified across the prediction horizon. Since MPC finds the state trajectory of the plant in the future, you can use MPC to solve trajectory optimization problems. Such

problems include autonomous parking of a vehicle, motion planning of a robot arm, and finding a landing path for a rocket.

For trajectory optimization problems, the plant, cost function, and constraints are often nonlinear. Therefore, you must formulate and solve the planning problem using nonlinear MPC. You can pass the generated optimal path to a path-following controller as a reference signal, so that it can execute the planned maneuver.

Compared to a generic nonlinear MPC controller, implemented using an `nlmpc` object, multistage nonlinear MPC provides a more flexible and efficient implementation with staged costs and constraints. This flexibility and efficiency are especially useful for trajectory planning.

A multistage nonlinear MPC controller with prediction horizon `p` defines `p+1` stages, which represent times `k` (current time) through `k+p`. For each stage, you can specify stage-specific cost, inequality constraint, and equality constraint functions. Each function depends only on the plant state and input values at the corresponding stage.

Given the current plant states, `x[k]`, the MPC controller finds the manipulated variable (MV) trajectory (from time `k` to `k+p-1`) that optimizes the summed stage costs (from time `k` to `k+p`) while satisfying all the stage constraints (from time `k` to `k+p`).

In this example, select the prediction horizon `p` and sample time `Ts` such that the prediction time is `p*Ts = 10` seconds.

```
Ts = 0.2;
pPlanner = 50;
```

Create a multistage nonlinear MPC controller for the specified prediction horizon and sample time.

```
planner = nlmpcMultistage(pPlanner,6,2);
planner.Ts = Ts;
```

Specify prediction model and its analytical Jacobian.

```
planner.Model.StateFcn = 'RocketStateFcn';
planner.Model.StateJacFcn = 'RocketStateJacobianFcn';
```

Specify hard bounds on the two thrusters. You can adjust the maximum thrust and observe its impact on the landing strategy chosen by the planner. Typical maximum values are between 6 and 10 Newtons. If the maximum thrust is too small, you might not be able to land the rocket successfully if the initial position is challenging.

```
planner.MV(1).Min = 0;
planner.MV(1).Max = 8;
planner.MV(2).Min = 0;
planner.MV(2).Max = 8;
```

To avoid crashing, specify a hard lower bound on the vertical Y position.

```
planner.States(2).Min = 10;
```

There are different factors that you can include in your cost function. For example, you can minimize time, fuel consumption, or landing speed. For this example, you define a cost function that optimizes fuel consumption by minimizing the sum of the thrust values. To improve efficiency, you also supply the analytical Jacobian function for the cost.

Use the same cost function for all stages. Since MVs are only valid from stage 1 to stage `p`, you do not need to define a stage cost for the final stage, `p+1`.

```
for ct=1:pPlanner
    planner.Stages(ct).CostFcn = 'RocketPlannerCostFcn';
    planner.Stages(ct).CostJacFcn = 'RocketPlannerCostGradientFcn';
end
```

To ensure a successful landing at the target, specify terminal state for the final stage.

```
planner.Model.TerminalState = [0;10;0;0;0;0];
```

In this example, set the maximum number of iterations to a large value to accommodate the large search space and the nonideal default initial guess.

```
planner.Optimization.SolverOptions.MaxIterations = 1000;
```

After creating you nonlinear MPC controller, check whether there is any problem with your state, cost, and constraint functions, as well as their analytical Jacobian functions. To do so, call `validateFcns` functions with random initial plant states and inputs.

```
validateFcns(planner,rand(6,1),rand(2,1));
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
"CostJacFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Analysis of user-provided model, cost, and constraint functions complete.
```

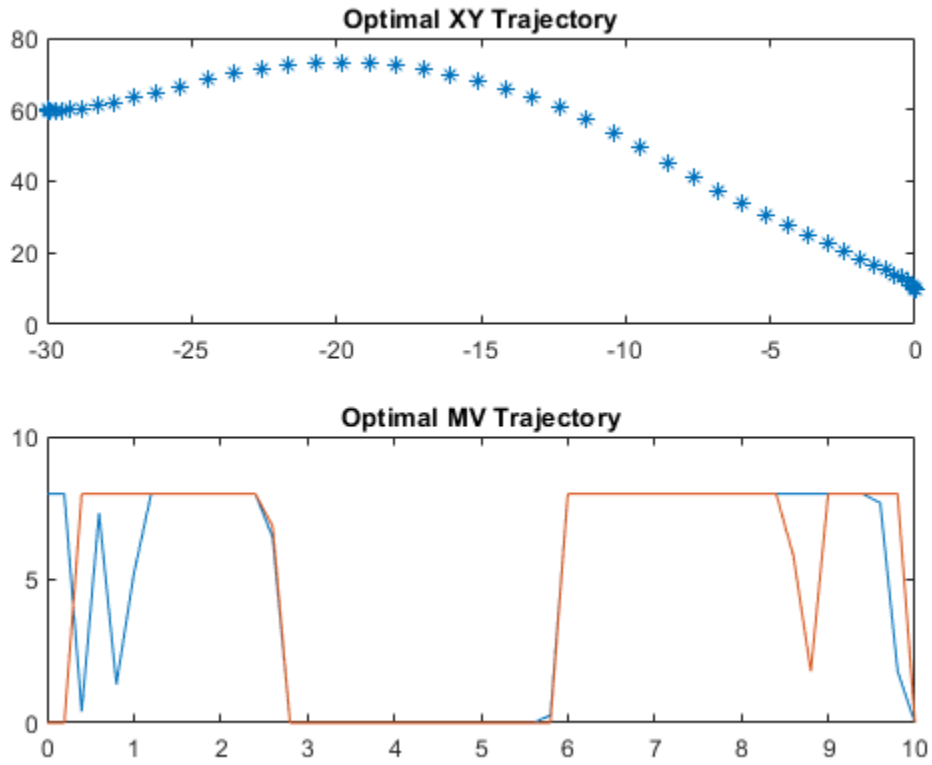
Compute the optimal landing path using `nlmpcmove`, which can typically take a few seconds, depending on the initial rocket position.

```
fprintf('Rocker landing planner running...\n');
tic;
[~,~,info] = nlmpcmove(planner,x0,u0);
t=toc;
fprintf('Calculation Time = %s\n',num2str(t));
fprintf('Objective cost = %s',num2str(info.Cost));
fprintf('ExitFlag = %s',num2str(info.Iterations));
fprintf('Iterations = %s\n',num2str(info.Iterations));
```

```
Rocker landing planner running...
Calculation Time = 25.1026
Objective cost = 492.9689ExitFlag = 381Iterations = 381
```

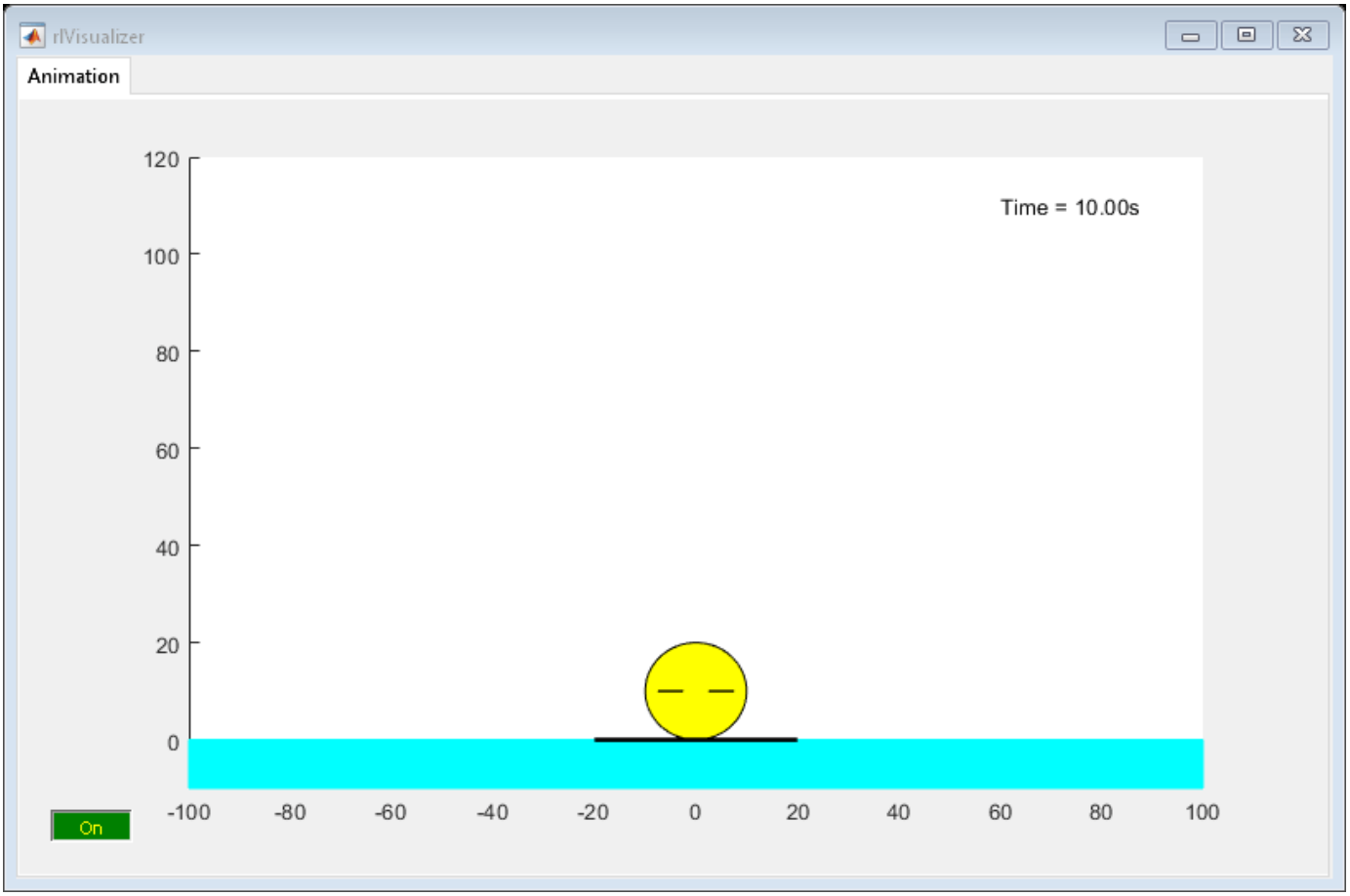
Extract the optimal trajectory from the `info` structure and plot the result.

```
figure
subplot(2,1,1)
plot(info.Xopt(:,1),info.Xopt(:,2),'*')
title('Optimal XY Trajectory')
subplot(2,1,2)
plot(info.Topt,info.MVopt(:,1),info.Topt,info.MVopt(:,2))
title('Optimal MV Trajectory')
```



Animate the planned optimal trajectory.

```
plotobj = RocketAnimation(6,2);
for ct=1:pPlanner+1
    updatePlot(plotobj,(ct-1)*planner.Ts,info.Xopt(ct,:),info.MVopt(ct,:));
    pause(0.1);
end
```



Design Lander and Follow the Optimal Path

Like generic nonlinear MPC, you can use multistage nonlinear MPC for reference tracking and disturbance rejection. In this example, you use it to track the optimal trajectory found by the planner. For a path-following problem, the lander does not require a long prediction horizon. Create the controller.

```
pLander = 10;
lander = nlmpcMultistage(pLander,6,2);
lander.Ts = Ts;
```

For the path-following controller, the lander has the same prediction model, thrust bounds, and minimum Y position.

```
lander.Model.StateFcn = 'RocketStateFcn';
lander.Model.StateJacFcn = 'RocketStateJacobianFcn';
lander.MV(1).Min = 0;
lander.MV(1).Max = 8;
lander.MV(2).Min = 0;
lander.MV(2).Max = 8;
lander.States(2).Min = 10;
```

The cost function for the lander is different from that of the planner. The lander uses quadratic cost terms to achieve both tight reference tracking (by penalizing the tracking error) and smooth control actions (by penalizing large changes in the control actions). This lander cost function is implemented

in the `RocketLanderCostFcn` function. The corresponding manually derived cost gradient function is implemented in `RocketLanderCostGradientFcn`.

At run time, you provide the six state trajectory references to the lander as stage parameters. Therefore, specify the number of parameters for each stage.

```
for ct=1:pLander+1
    lander.Stages(ct).CostFcn = 'RocketLanderCostFcn';
    lander.Stages(ct).CostJacFcn = 'RocketLanderCostGradientFcn';
    lander.Stages(ct).ParameterLength = 6;
end
```

Since changes in control actions are represented by `MVRate`, you must enable the multistage nonlinear MPC controller to use `MVRate` values in its in calculations.

```
lander.UseMVRate = true;
```

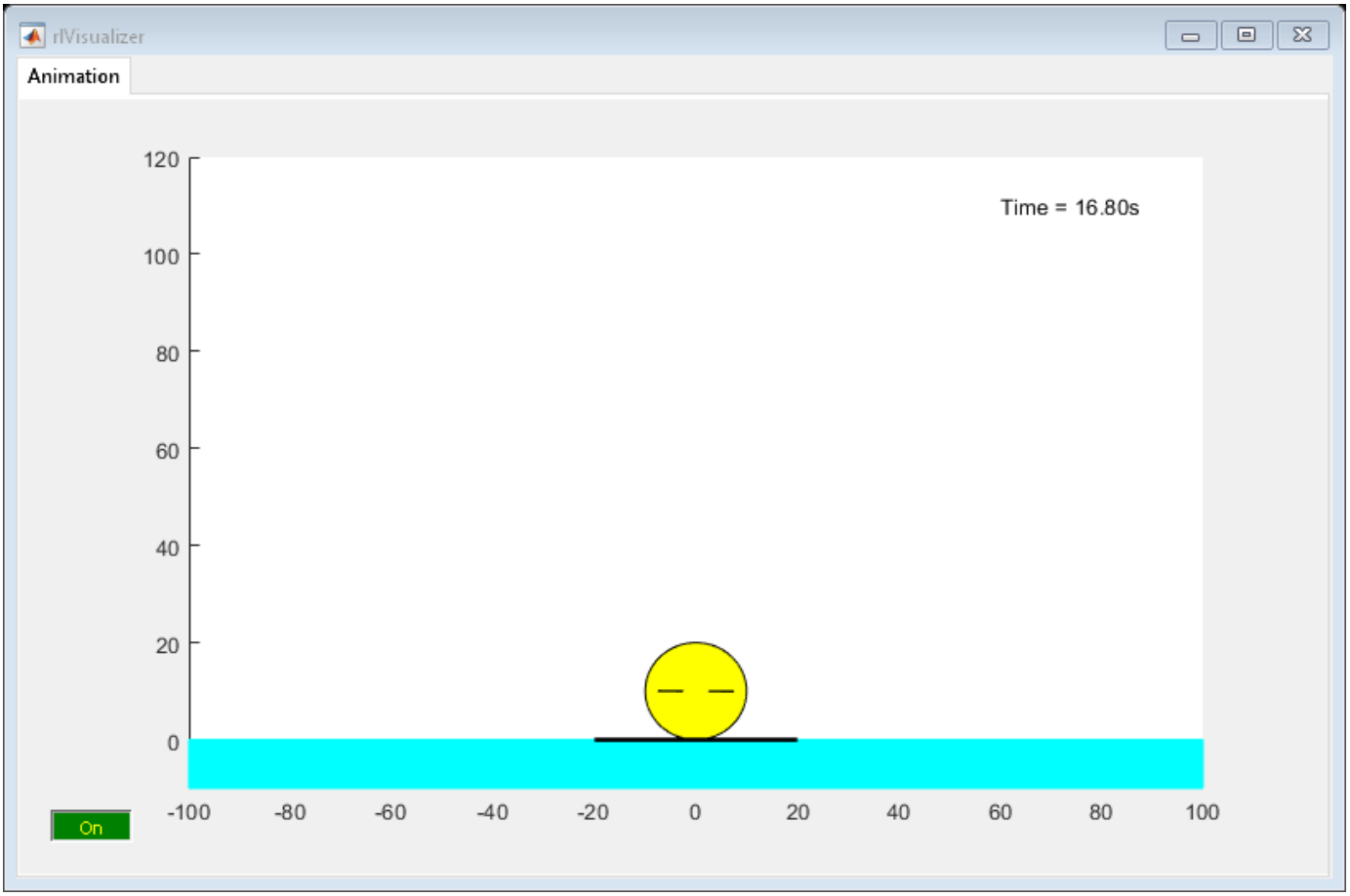
Validate the controller design.

```
simdata = getSimulationData(lander);
validateFcns(lander,rand(6,1),rand(2,1),simdata);
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11] are OK.
"CostJacFcn" of the following stages [1 2 3 4 5 6 7 8 9 10 11] are OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Simulate the landing maneuver in a closed-loop control scenario by iteratively calling `nlmpcmove`. This simulation assumes that all states are measured. Stop the simulation when the rocket states are close enough to the target states.

```
x = x0;
u = u0;
k = 1;
references = reshape(info.Xopt',(pPlanner+1)*6,1); % Extract reference signal as column vector.
while true
    % Obtain new reference signals.
    simdata.StageParameter = RocketLanderReferenceSignal(k,references,pLander);
    % Compute the control action.
    [u,simdata,infoLander] = nlmpcmove(lander,x,u,simdata);
    % Update the animation plot.
    updatePlot(plotobj,(k-1)*Ts,x,u);
    pause(0.1);
    % Simulate the plant to the next state using an ODE solver.
    [~,X] = ode45(@(t,x) RocketStateFcn(x,u),[0 Ts],x);
    x = X(end,:)';
    % Stop if rocket has landed.
    if max(abs(x-[0;10;0;0;0;0])) < 1e-2
        % Plot the the final rocket position.
        updatePlot(plotobj,k*Ts,x,zeros(2,1));
        break
    end
    % Move to the next simulation step.
    k = k + 1;
end
```

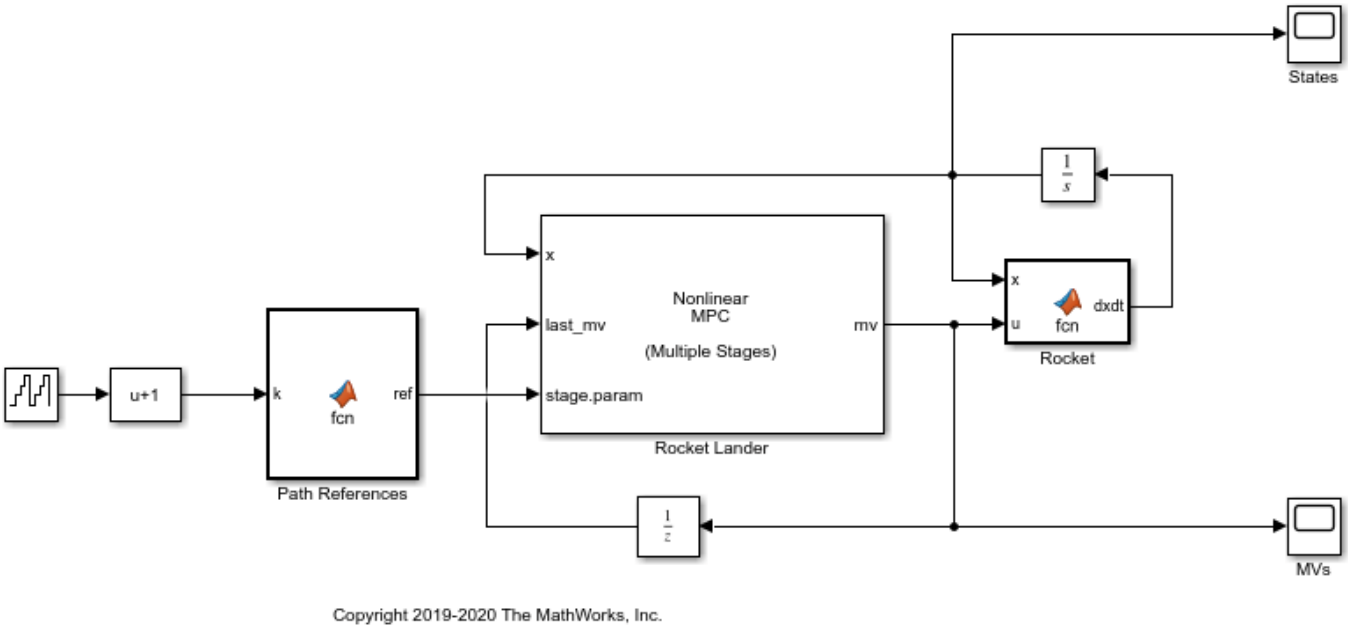


Due to the shorter horizon and different cost terms, the landing trajectory slightly differs from the planned trajectory and it takes longer to land. This result is often what happens in such a two-tier control framework with planning and regulation.

Simulate in Simulink Using Multistage Nonlinear MPC Block

You can implement the same closed-loop simulation in a Simulink model using the Multistage Nonlinear MPC block.

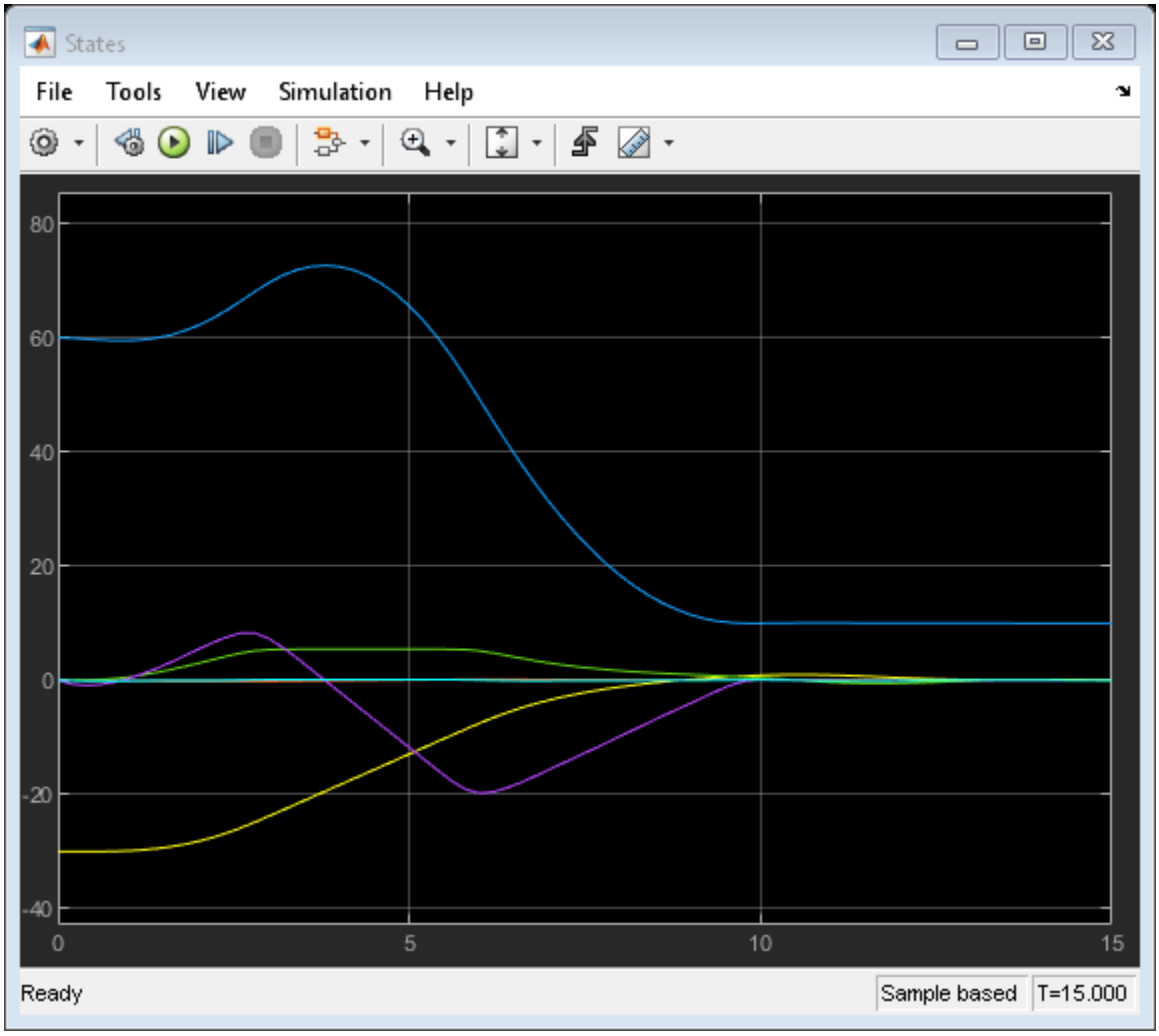
```
mdl = 'RocketLanderSimulation';
open_system(mdl)
```

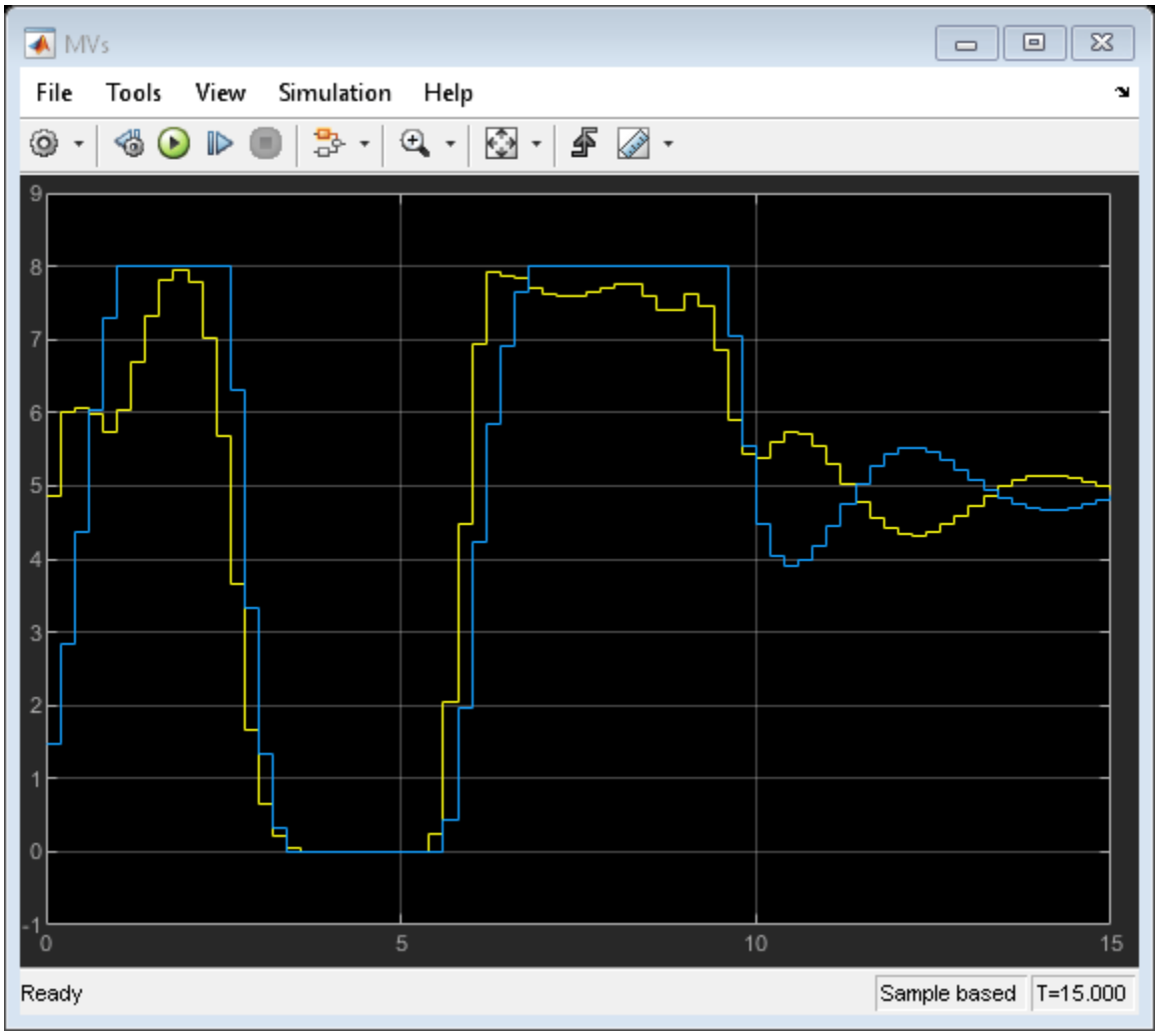


The States scope shows that the plant states are brought to the target states in a reasonable time.

```
sim(mdl)
open_system([mdl '/States'])
open_system([mdl '/MVs'])
```

```
ans =

  Simulink.SimulationOutput:
                  tout: [81x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```


States
File
Tools
View
Simulation
Help
80
60
40
20
0
-20
-40
0
5
10
15
Ready
Sample based
T=15.000



For real-time applications, you can generate code from the Multistage Nonlinear MPC block.

```
bdclose(mdl)
```

See Also

Functions
`nlmpcMultistage` | `nlmpcmove`

Blocks
Multistage Nonlinear MPC Controller

Related Examples

- “Nonlinear MPC” on page 9-2
- “Land a Rocket Using Multistage Nonlinear MPC” on page 9-141

Code Generation

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2
- “Generate Code to Compute Optimal MPC Moves in MATLAB” on page 10-7
- “Simulation and Code Generation Using Simulink Coder” on page 10-13
- “Simulation and Structured Text Generation Using Simulink PLC Coder” on page 10-20
- “Use the GPU to Compute MPC Moves in MATLAB” on page 10-25
- “Use the GPU to Simulate an MPC Controller in Simulink” on page 10-31
- “Using MPC Controller Block Inside Function-Call and Triggered Subsystems” on page 10-34
- “Solve Custom MPC Quadratic Programming Problem and Generate Code” on page 10-46
- “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 10-56
- “Real-Time Control with OPC Toolbox” on page 10-63
- “Implement MPC Controllers using Embotech FORCESPRO Solvers” on page 10-67

Generate Code and Deploy Controller to Real-Time Targets

Model Predictive Control Toolbox software provides code generation functionality for controllers designed in MATLAB or Simulink.

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 for an implicit or explicit linear MPC controller:

1. Generate data structures from an MPC controller or explicit MPC controller using `getCodeGenerationData`.
2. To verify that your controller produces the expected closed-loop results, simulate it using `mpcmoveCodeGeneration` in place of `mpcmove`.
3. Generate code for `mpcmoveCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

For an example, see “Generate Code to Compute Optimal MPC Moves in MATLAB” on page 10-7.

You can also generate code for nonlinear MPC controllers that use the default `fmincon` solver with the SQP algorithm. To generate code for computing optimal control moves for a nonlinear MPC controller:

1. Generate data structures from a nonlinear MPC controller using `getCodeGenerationData`.
2. To verify that your controller produces the expected closed-loop results, simulate it using `nlmpcmoveCodeGeneration` in place of `nlmpcmove`.
3. Generate code for `nlmpcmoveCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

Code Generation in Simulink

After designing a controller in Simulink using any of the MPC blocks, you can generate code and deploy it for real-time control. You can deploy controllers to all targets supported by the following products:

- Simulink Coder
- Embedded Coder®
- Simulink PLC Coder
- Simulink Real-Time™

You can generate code for any of the Model Predictive Control Toolbox Simulink blocks.

Types of Controllers	Blocks
Implicit MPC controllers	MPC Controller
Explicit MPC controllers	Explicit MPC Controller

Types of Controllers	Blocks
Gain-scheduled MPC controllers	Multiple MPC Controllers Multiple Explicit MPC Controllers
Adaptive MPC controllers	Adaptive MPC Controller
MPC controllers for automotive applications	Adaptive Cruise Control System Lane Keeping Assist System Path Following Control System
Nonlinear MPC controllers that use `fmincon` with SQP	Nonlinear MPC Controller

For more information on generating code, see “Simulation and Code Generation Using Simulink Coder” on page 10-13 and “Simulation and Structured Text Generation Using Simulink PLC Coder” on page 10-20.

Note The MPC Controller, Explicit MPC Controller, Adaptive MPC Controller, and Nonlinear MPC Controller blocks are implemented using the MATLAB Function (Simulink) block. To see the structure, right-click the block, and select **Mask > Look Under Mask**. Then, open the MPC subsystem underneath.

Note If your nonlinear MPC controller uses optional parameters, you must also generate code for the Bus Creator block connected to the **params** input port of the Nonlinear MPC Controller block. To do so, place the Nonlinear MPC Controller and Bus Creator blocks within a subsystem, and generate code for that subsystem.

Generate CUDA Code for Linear MPC Controllers

You can generate CUDA® code for your MPC controller using GPU Coder™. For more information on supported GPUs, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on installing and setting up the prerequisite product, see “Installing Prerequisite Products” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

To generate and use GPU code in MATLAB:

1. Design a linear controller using an `mpc` object.
2. Generate the structures for the core, states, and online data from your linear MPC controller using the `getCodeGenerationData` function.
3. Optionally simulate your closed loop iteratively using the `mpcmoveCodeGeneration` function and the data structures created in the previous step.
4. Create a coder configuration options object using the `coder.gpuConfig` function, and configure the code generation options.
5. Generate code for the `mpcmoveCodeGeneration` function using the `codegen` function and the coder configuration options object. Doing so generates a new function which uses code running on the GPU.
6. Simulate your controller using the new generated function and the data structures.

For an example on using GPU code in MATLAB, see “Use the GPU to Compute MPC Moves in MATLAB” on page 10-25

You can generate and use GPU code from the MPC Controller, Adaptive MPC Controller, or Explicit MPC Controller blocks.

To generate GPU code from a Simulink model containing any of these blocks, open the Configuration Parameters dialog box by clicking **Model Settings**. Then, in the **Code Generation** section, select **Generate GPU code**.

For details on how to configure your model for GPU code generation, see “Code Generation from Simulink Models with GPU Coder” (GPU Coder).

Sampling Rate in Real-Time Environment

The sampling rate that a controller can achieve in a real-time environment is system-dependent. For example, for a typical small MIMO control application running on Simulink Real-Time, the sample time can be as long as 1–10 ms for linear MPC and 100–1000 ms for nonlinear MPC. To determine the sample time, first test a less-aggressive controller whose sampling rate produces acceptable performance on the target. Next, decrease the sample time and monitor the execution time of the controller. You can further decrease the sample time as long as the optimization safely completes within each sampling period under normal plant operating conditions. To reduce the sample time, you can also consider using:

- Explicit MPC. While explicit MPC controllers have a faster execution time, they also have a larger memory footprint, since they store precomputed control laws. For more information, see “Explicit MPC Design”.
- A suboptimal QP solution after a specified number of maximum solver iterations. For more information, see “Suboptimal QP Solution” on page 2-18.

Tip A lower controller sample time does not necessarily provide better performance. In fact, you want to choose a sample time that is small enough to give you good performance but no smaller. For the same prediction time, smaller sample times result in larger prediction steps, which in turn produces a larger memory footprint and more complex optimization problem.

QP Problem Construction for Generated C Code

At each control interval, an implicit or adaptive MPC controller constructs a new QP problem, which is defined as:

$$\underset{x}{Min}(\frac{1}{2}x^{\top}Hx + f^{\top}x)$$

subject to the linear inequality constraints

$$Ax \leq b$$

where

- x is the solution vector.
- H is the Hessian matrix.

- A is a matrix of linear constraint coefficients.
- f and b are vectors.

In generated C code, the following matrices are used to provide H, A, f, and b. Depending on the type and configuration of the MPC controller, these matrices are either constant or regenerated at each control interval.

Constant Matrix	Size	Purpose	Implicit MPC	Implicit MPC with Online Weight Tuning	Adaptive MPC or LTV MPC
`Hinv`	N_M-by-N_M	Inverse of the Hessian matrix, H	Constant	Regenerated	Regenerated
`Linv`	N_M-by-N_M	Inverse of the lower-triangular Cholesky decomposition of H			
`Ac`	N_C-by-N_M	Linear constraint coefficients, A		Constant	
`Kx`	N_{xqp}-by-(N_M–1)	Used to generate f		Regenerated	
`Kr`	p*N_y-by-(N_M–1)				
`Kul`	N_{mv}-by-(N_M–1)				
`Kv`	(N_{md}+1)*(p+1)-by-(N_M–1)				
`Kut`	p*N_{mv}-by-(N_M–1)				
`Mlim`	N_C-by-1	Used to generate b		Constant	Constant, except when there are custom constraints
`Mx`	N_C-by-N_{xqp}				Regenerated
`Mul`	N_C-by-N_{mv}				
`Mv`	N_C-by-(N_{md}+1)*(p+1)				

Here:

- p is the prediction horizon.
- N_{mv} is the number of manipulated variables.
- N_{md} is the number of measured disturbances.
- N_y is the number of output variables.
- N_M is the number of optimization variables (m*N_{mv}+1, where m is the control horizon).
- N_{xqp} is the number of states used for the QP problem; that is, the total number of the plant states and disturbance model states.
- N_C is the total number of constraints.

At each control interval, the generated C code computes f and b as:

$$f = Kx^{\mathsf{T}} * x_q + Kr^{\mathsf{T}} * r_p + Ku1^{\mathsf{T}} * m_l + Kv^{\mathsf{T}} * v_p + Kut^{\mathsf{T}} * u_t$$

$$b = -\left(Mlim + Mx * x_q + Mu1 * m_l + Mv * v_p\right)$$

where

- x_q is the vector of plant and disturbance model states estimated by the Kalman filter.
- m_l is the manipulated variable move from the previous control interval.
- u_t is the manipulated variable target.
- v_p is the sequence of measured disturbance signals across the prediction horizon.
- r_p is the sequence of reference signals across the prediction horizon.

Note When generating code in MATLAB, the `getCodeGenerationData` command generates these matrices and returns them in `configData`.

Code Generation for Custom QP Solvers

You can generate code for linear MPC controllers that use a custom QP solver written in C or MATLAB code. The controller calls this solver in place of the built-in QP solver at each control interval.

For an example, see “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 10-56. For more information on custom QP solvers, see “Custom QP Solver” on page 2-19.

For information on using the FORCESPRO solver, see “Implement MPC Controllers using Embotech FORCESPRO Solvers” on page 10-67.

See Also

Functions
`mpcmoveCodeGeneration` | `nlmpcmoveCodeGeneration` | `review`

Blocks
MPC Controller | Multiple MPC Controllers | Explicit MPC Controller | Multiple Explicit MPC Controllers | Adaptive MPC Controller | Nonlinear MPC Controller

More About

- “Simulation and Code Generation Using Simulink Coder” on page 10-13
- “Simulation and Structured Text Generation Using Simulink PLC Coder” on page 10-20
- “Generate Code to Compute Optimal MPC Moves in MATLAB” on page 10-7

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.

After simulating the controller using `mpcmove`, you use `mpcmoveCodeGeneration` to simulate the controller using optimized data structures, reproducing the same results. Then you generate an executable having the same inputs and outputs as `mpcmoveCodeGeneration`. Finally, you use the generated executable to simulate the controller, using the same code and data structures you used for `mpcmoveCodeGeneration.`

Plant Model

The plant is a single-input, single-output, stable, 2nd order linear plant.

```
plant = tf(5,[1 0.8 3]);
```

Set a sampling time of one second, 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);
```

Design MPC Controller

Create an MPC controller with default horizons and the specified sampling time.

```
mpcobj = mpc(plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

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;
```

Simulate 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. Note that `xmpc` is an handle object pointing to the current (always updated) state of the controller.

```
xmpc = 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 measured
```

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 = OVMaxData(ct);
    % Compute control actions and store plant input.
    u = mpcmove(mpcobj,xmpc,y,1,[],options);
    uMPCMOVE = [uMPCMOVE u];
    % Update plant state.
    x = plant.A*x + plant.B*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;
```

Create data structures to use with `mpcmoveCodeGeneration` using `getCodeGenerationData`.

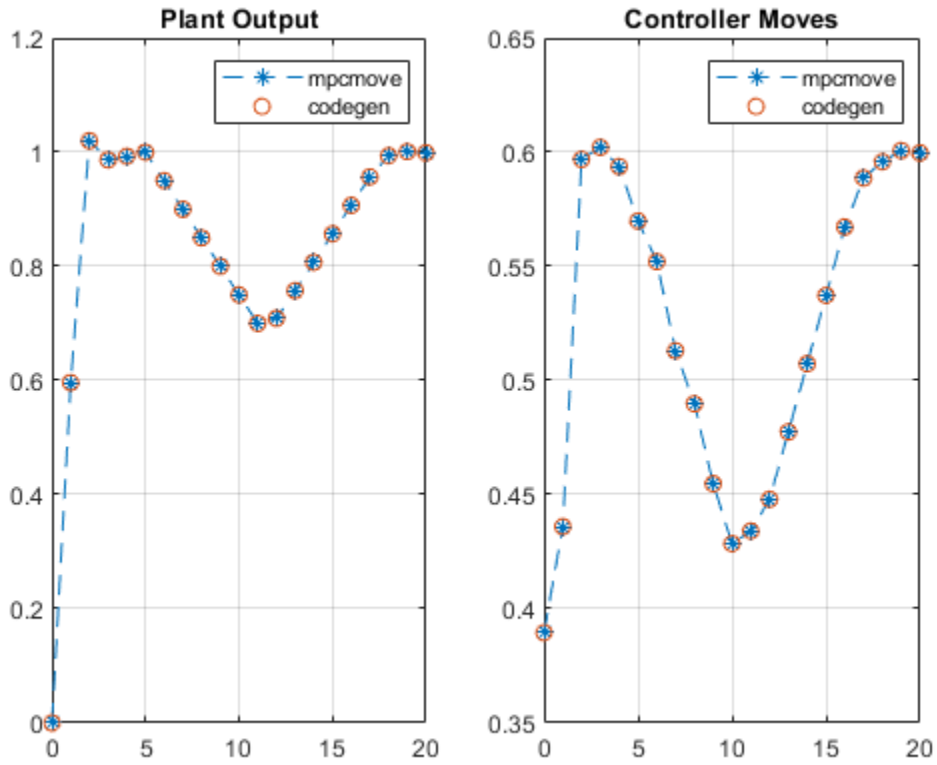
```
[coredata,statedata,onlinedata] = getCodeGenerationData(mpcobj);
```

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 and store control action.
    [u,statedata] = mpcmoveCodeGeneration(coredata,statedata,onlinedata);
    uCodeGen = [uCodeGen u];
    % Update plant state.
    x = plant.A*x + plant.B*u;
end
```

The simulation results are identical to those obtained 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')
```



Generate 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 the `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 using this argument.

```
% check MATLAB coder license
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});
```

```
Code generation successful.
```

Initialize data storage.

```
yMEX = [];
uMEX = [];
```

Initialize plant states.

```
x = x0;
```

Use `getCodeGenerationData` to create data structures to use with `mpcmoveCodeGeneration`.

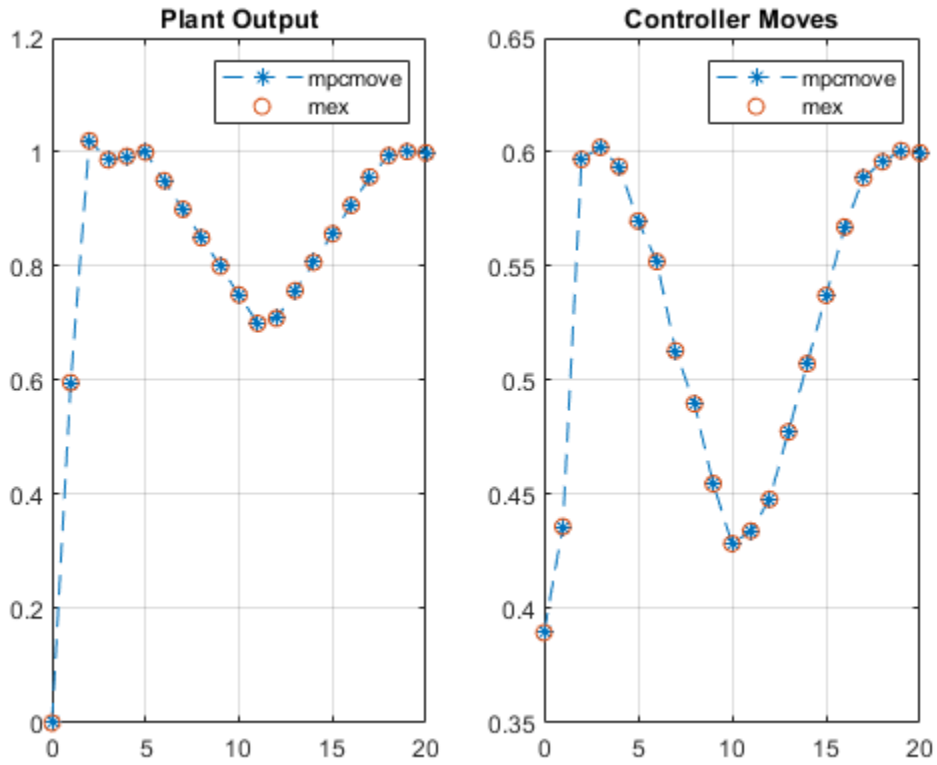
```
[coredata,statedata,onlinedata] = getCodeGenerationData(mpcobj);
```

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 and store control action.
    [u,statedata] = mpcmoveMEX(coredata,statedata,onlinedata);
    uMEX = [uMEX u];
    % Update plant state.
    x = plant.A*x + plant.B*u;
end
```

The simulation results are identical to those obtained using `mpcmove`.

```
figure
subplot(1,2,1)
plot(t,yMPCMOVE,'--*',t,yMEX,'o')
grid
legend('mpcmove','mex')
title('Plant Output')
subplot(1,2,2)
plot(t,uMPCMOVE,'--*',t,uMEX,'o')
grid
legend('mpcmove','mex')
title('Controller Moves')
```



See Also

`mpcmoveCodeGeneration` | `getCodeGenerationData`

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

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 is required to run this example.')
    return
end
if ~mpcchecktoolboxinstalled('simulinkcoder')
    disp('Simulink Coder is required to run this example.');
    return
end
```

Configure Environment

You must have write-permission to generate the relevant files and the executable. Therefore, 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;   %Sample 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);
```

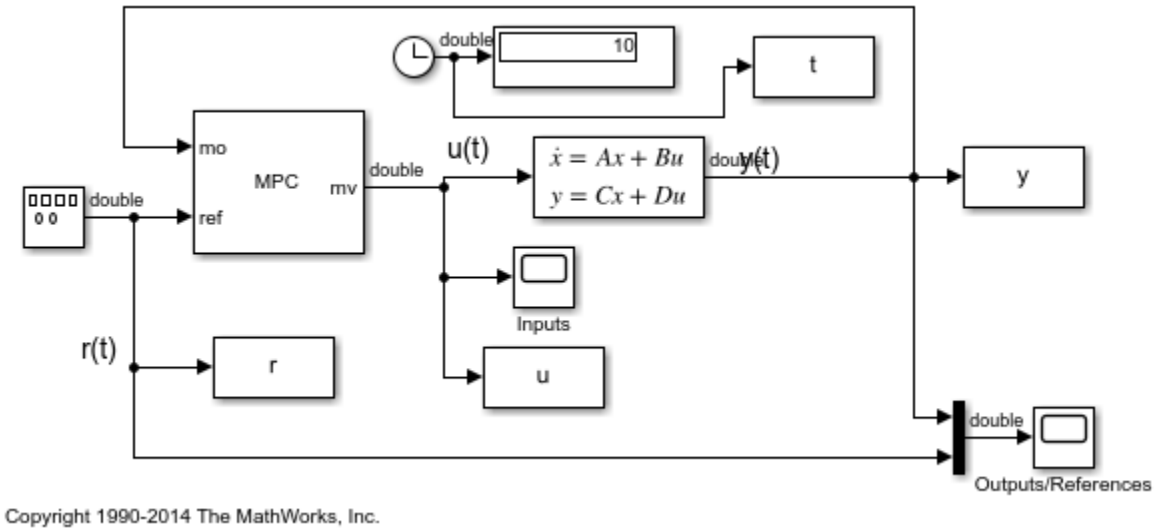
Simulate and Generate Code in Double-Precision

By default, MPC Controller blocks use double-precision data for 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 noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



The controller effort and the plant output are saved into base workspace as variables `u` and `y`, respectively.

Build the model with the `slbuild` command.

```
disp('Generating C code... Please wait until it finishes.')
set_param(mdl1,'RTWVerbose','off')
slbuild(mdl1);
```

```
Generating C code... Please wait until it finishes.
### Starting build procedure for: mpc_rtwdemo
### Successful completion of build procedure for: mpc_rtwdemo

Build Summary

Top model targets built:

Model        Action                        Rebuild Reason
========================================================================================================
mpc_rtwdemo  Code generated and compiled  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 39.397s
```

On a Windows® system, an executable file named `mpc_rtwdemo.exe` appears in the temporary directory after the build process finishes.

Run the executable.

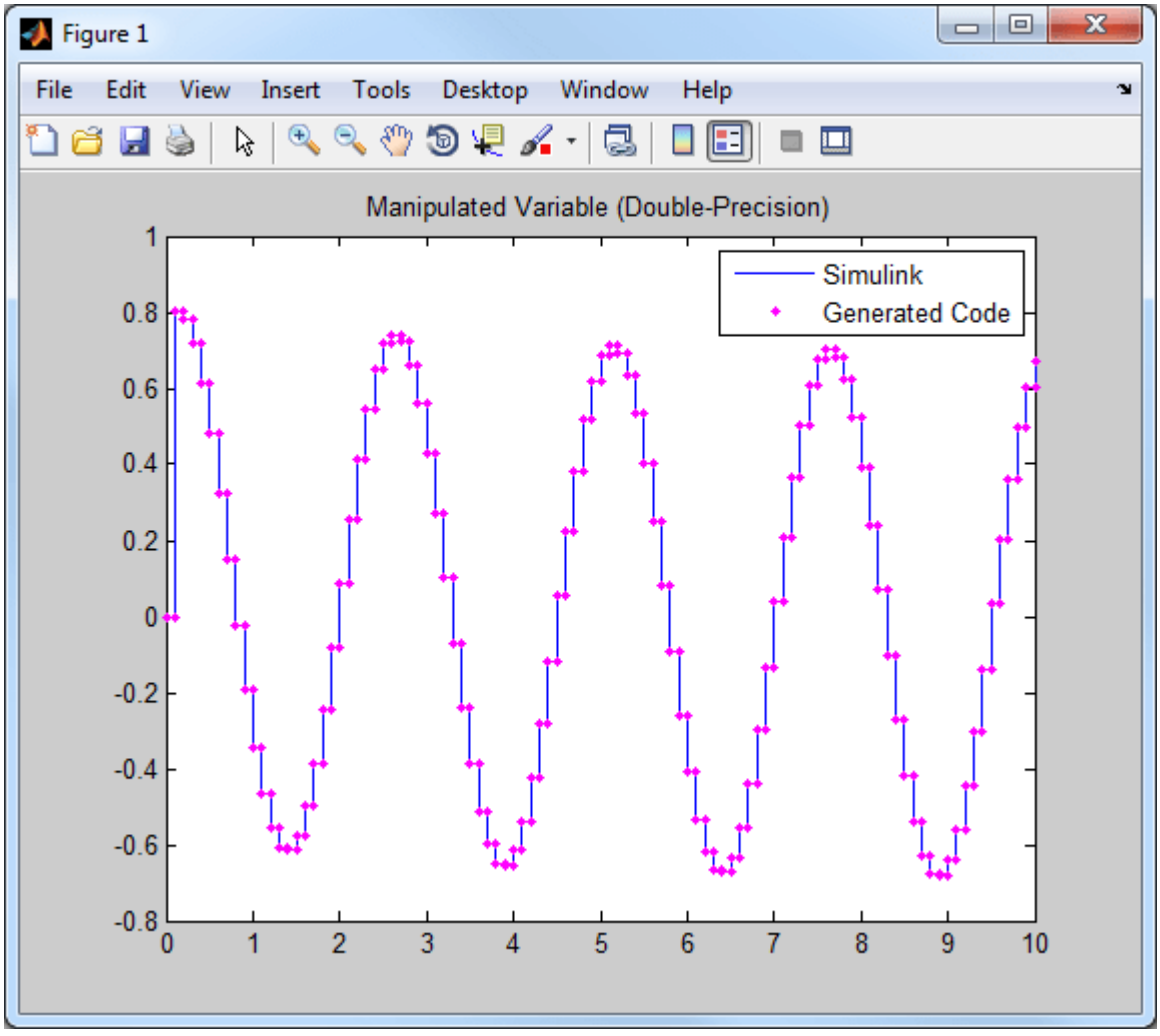
```
if ispc
    disp('Running executable...')
    status = system(mdl1);
else
    disp('The example only runs the executable on Windows system.')
end
```

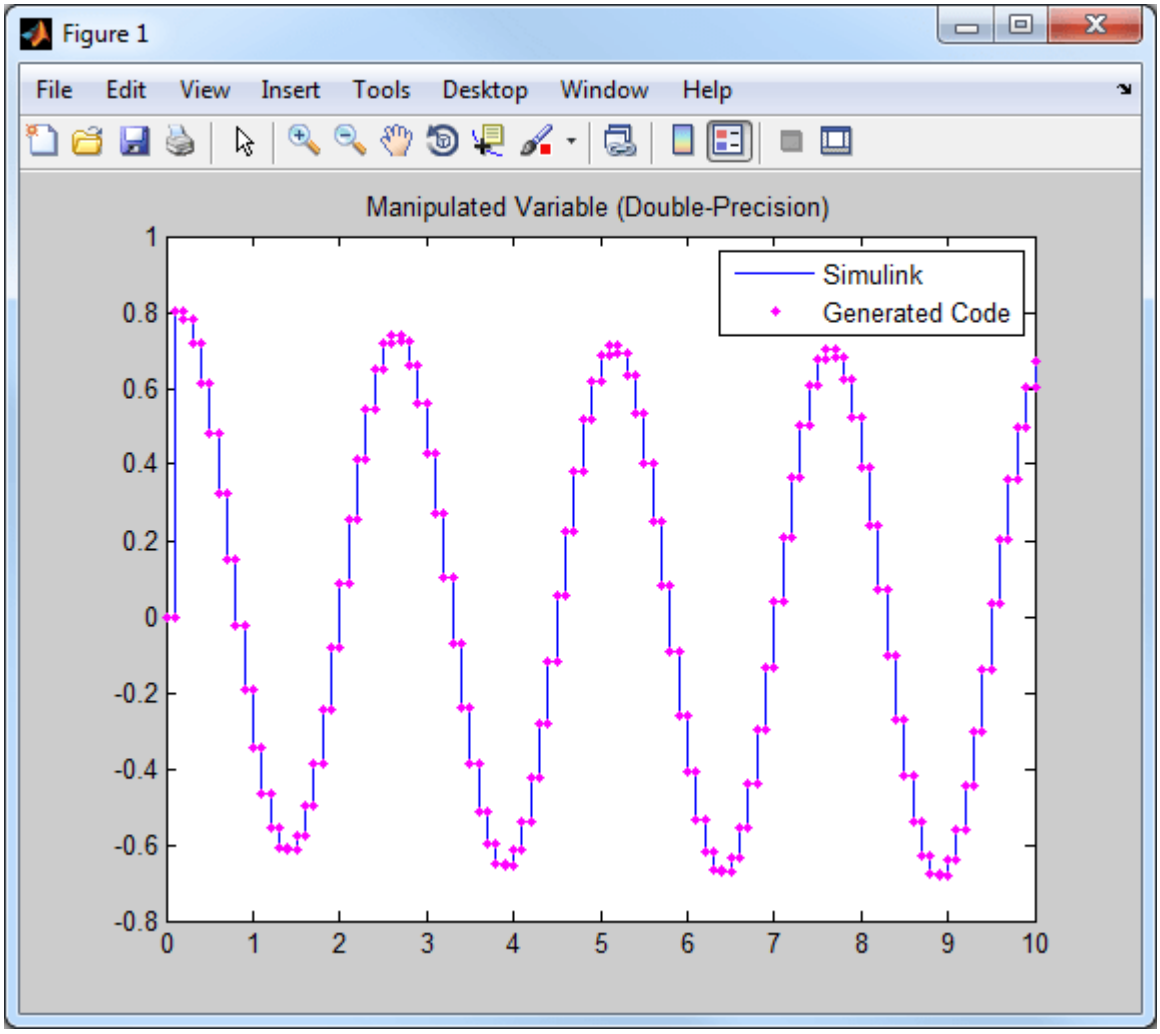
```
Running executable...
```

```
** 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`).



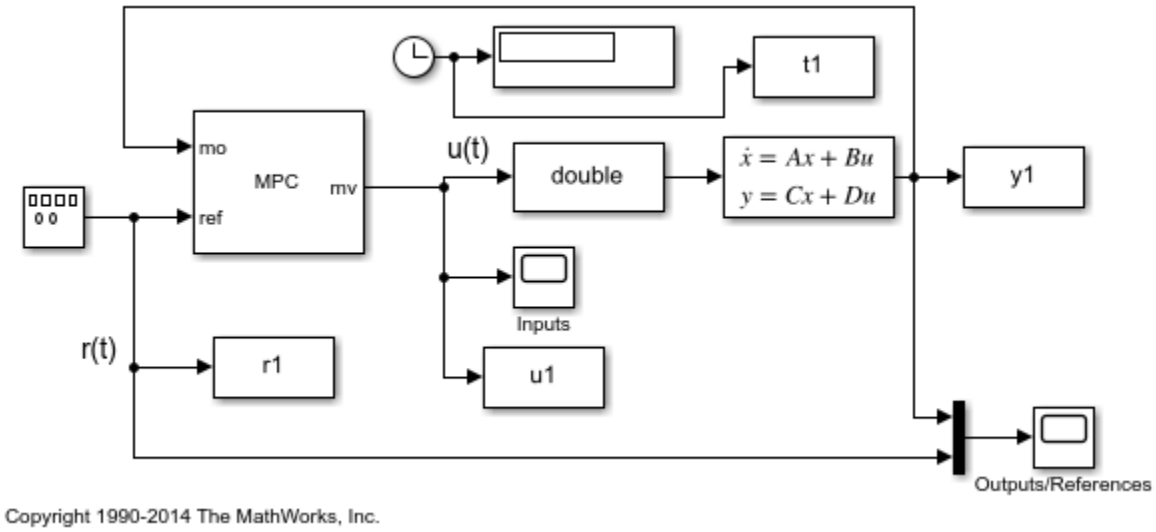


The responses are numerically equal.

Simulate and Generate Code in Single-Precision

You can also configure the MPC block to use single-precision data in simulation and code generation.

```
mdl2 = 'mpc_rtwdemo_single';
open_system(mdl2)
```



To do so, set the **Output data type** property of the MPC Controller block to `single`.

Simulate the model in Simulink.

```
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 `slbuild` command.

```
disp('Generating C code... Please wait until it finishes.')
set_param(mdl2,'RTWVerbose','off')
slbuild(mdl2);
```

```
Generating C code... Please wait until it finishes.
### Starting build procedure for: mpc_rtwdemo_single
### Successful completion of build procedure for: mpc_rtwdemo_single

Build Summary

Top model targets built:

Model               Action                        Rebuild Reason
========================================================================================================
mpc_rtwdemo_single  Code generated and compiled  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 37.678s
```

On a Windows system, an executable file named `mpc_rtwdemo_single.exe` appears in the temporary directory after the build process finishes

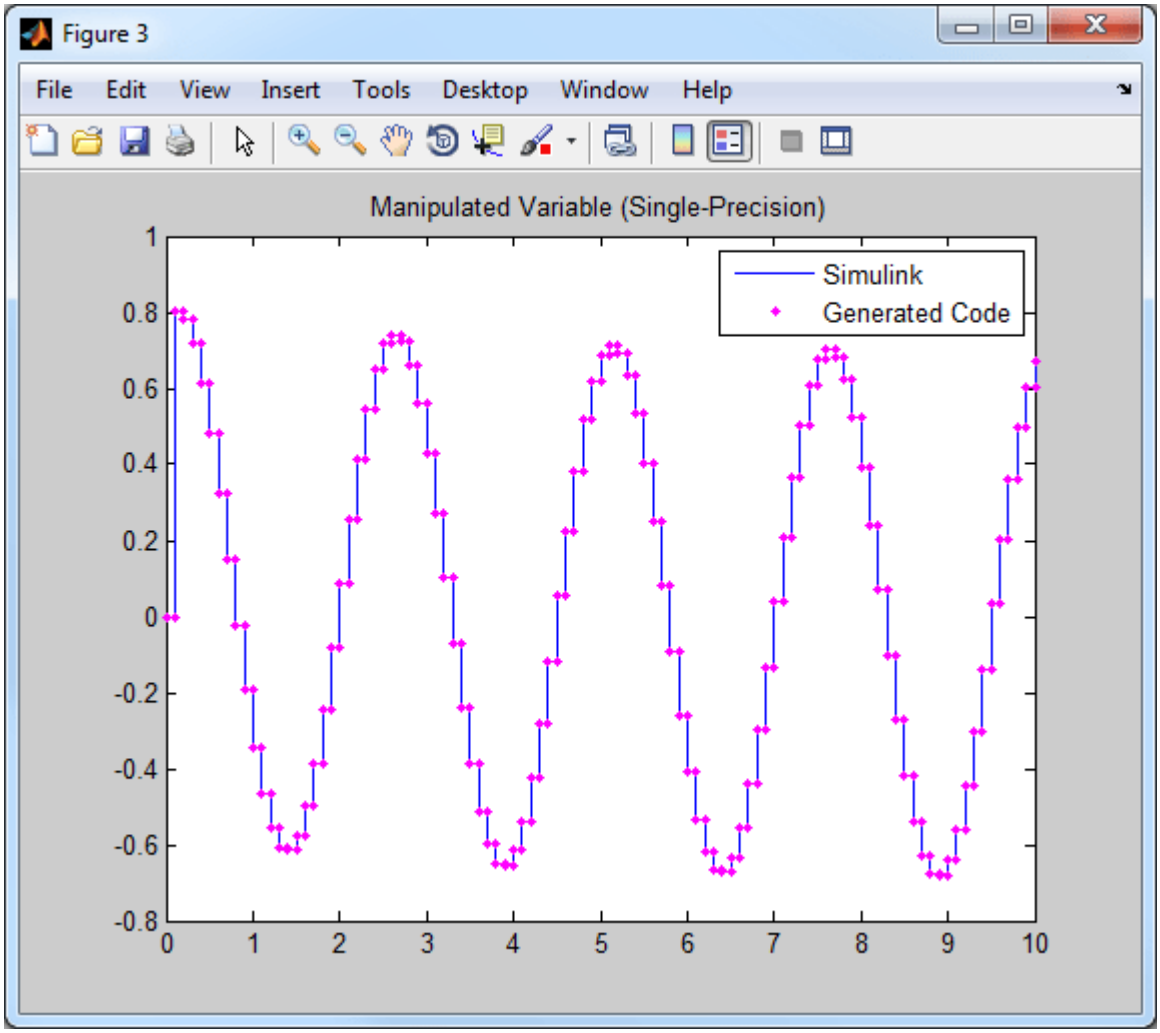
Run the executable.

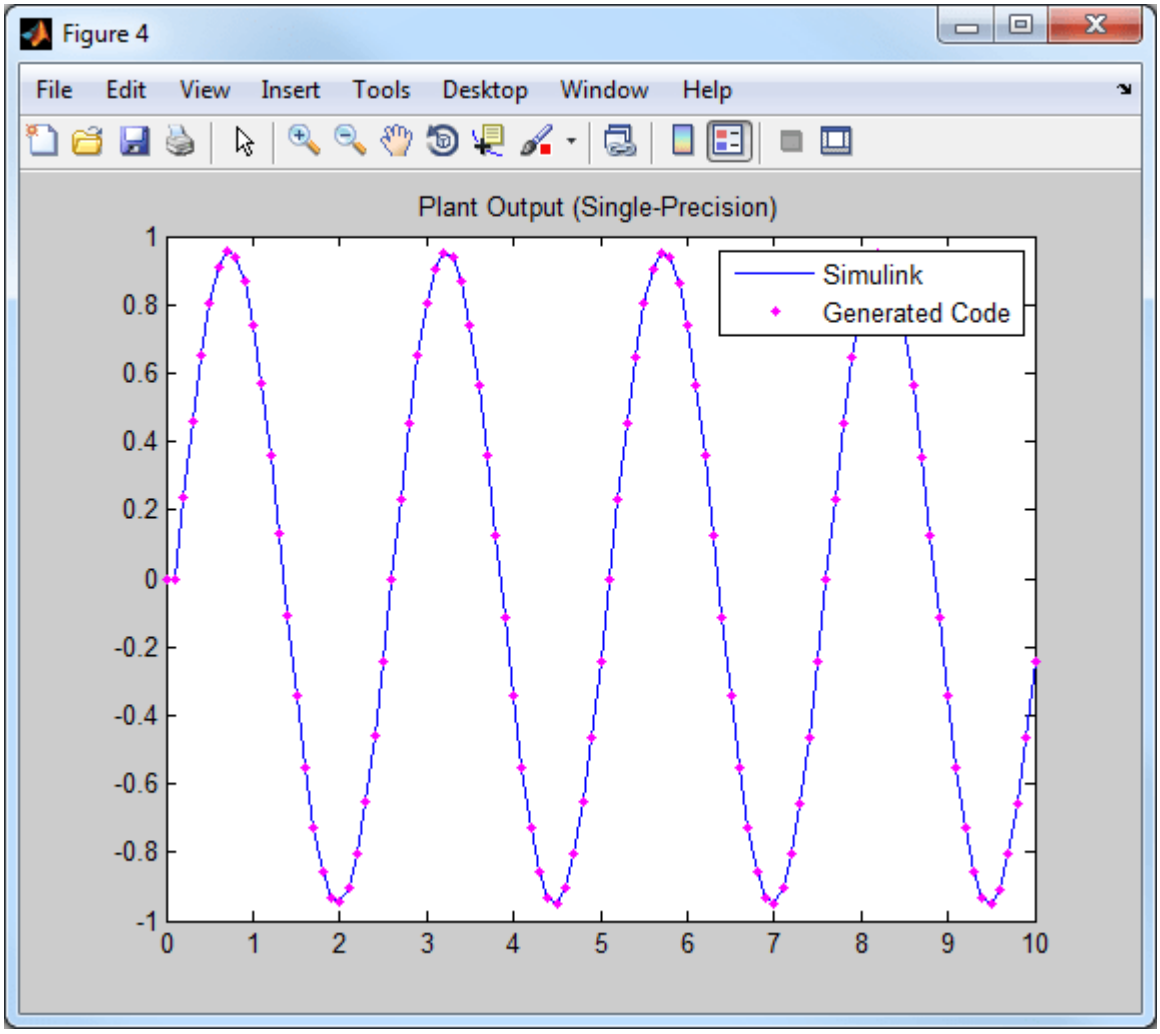
```
if ispc
    disp('Running executable...')
```

```
    status = system(mdl2);
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`).





The responses are numerically equal.

Close the Simulink models, and return to the original directory.

```
bdclose(mdl1)
bdclose(mdl2)
cd(cwd)
```

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Simulation and Structured Text Generation Using Simulink PLC Coder

This example shows how to simulate and generate Structured Text for an MPC Controller block using Simulink® 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 is required to run this example.')
    return
end
if ~mpcchecktoolboxinstalled('plccoder')
    disp('Simulink PLC Coder is required to run this example.');
    return
end
```

Setup Environment

You must have write-permission to generate the relevant files and the executable. Therefore, 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]));
```

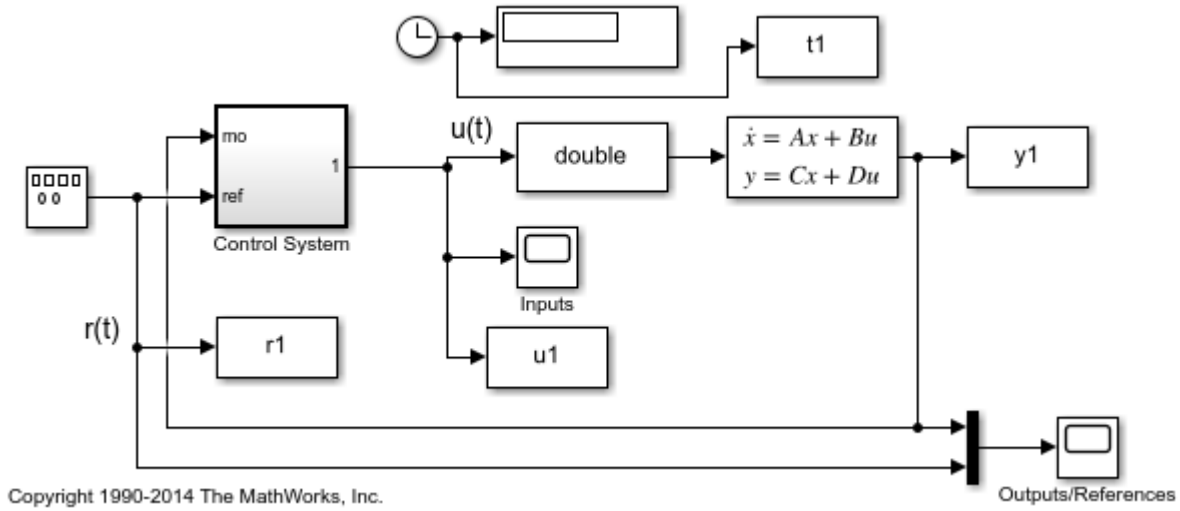
Define the MPC controller for the plant.

```
Ts = 0.1;   %Sample 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.

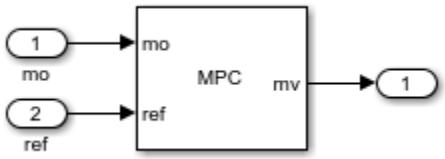
```
mdl = 'mpc_plcdemo';
open_system(mdl)
```



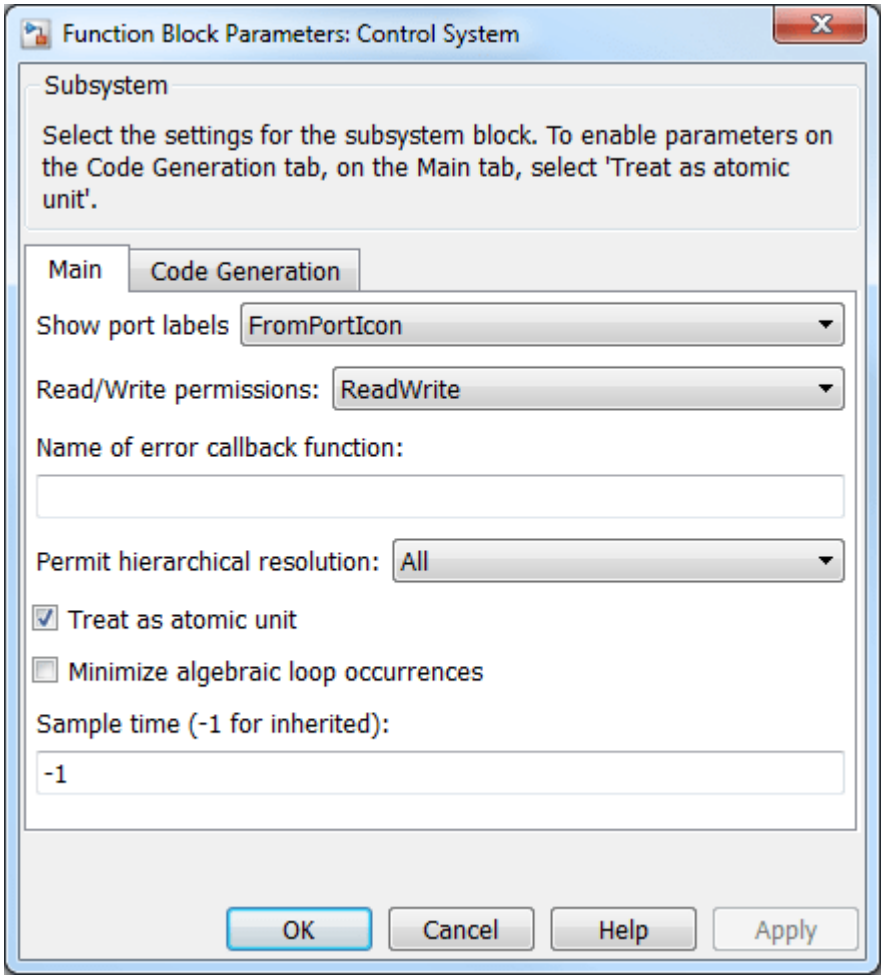
To generate structured text for the MPC Controller block, complete the following two steps:

- Configure the MPC block to use single-precision data. Set the **Output data type** property of the MPC Controller block to `single`.

```
open_system([mdl '/Control System/MPC Controller'])
```



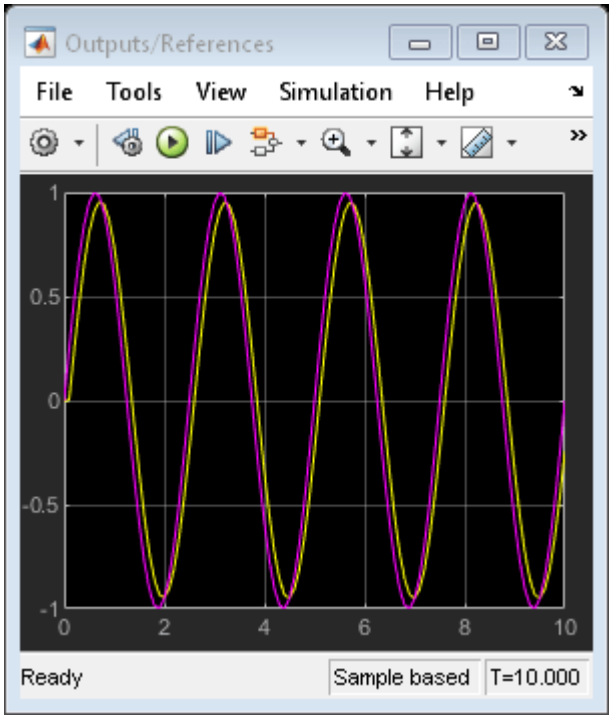
- Put the MPC block inside a subsystem block and treat the subsystem block as an atomic unit. Select the **Treat as atomic unit** property of the subsystem block.

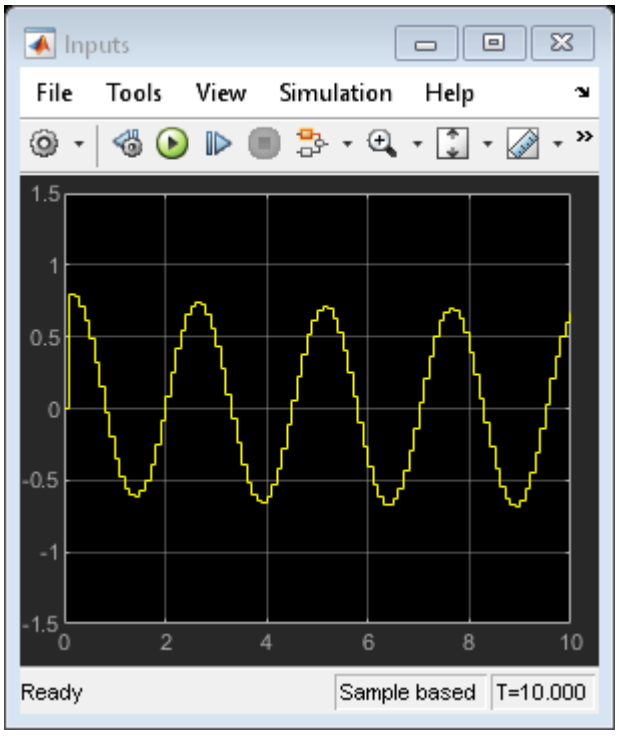


Simulate the model in Simulink.

```
close_system([mdl '/Control System/MPC Controller'])
open_system([mdl '/Outputs//References'])
open_system([mdl '/Inputs'])
sim(mdl)
```

```
-->Converting model to discrete time.
-->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 measured
```



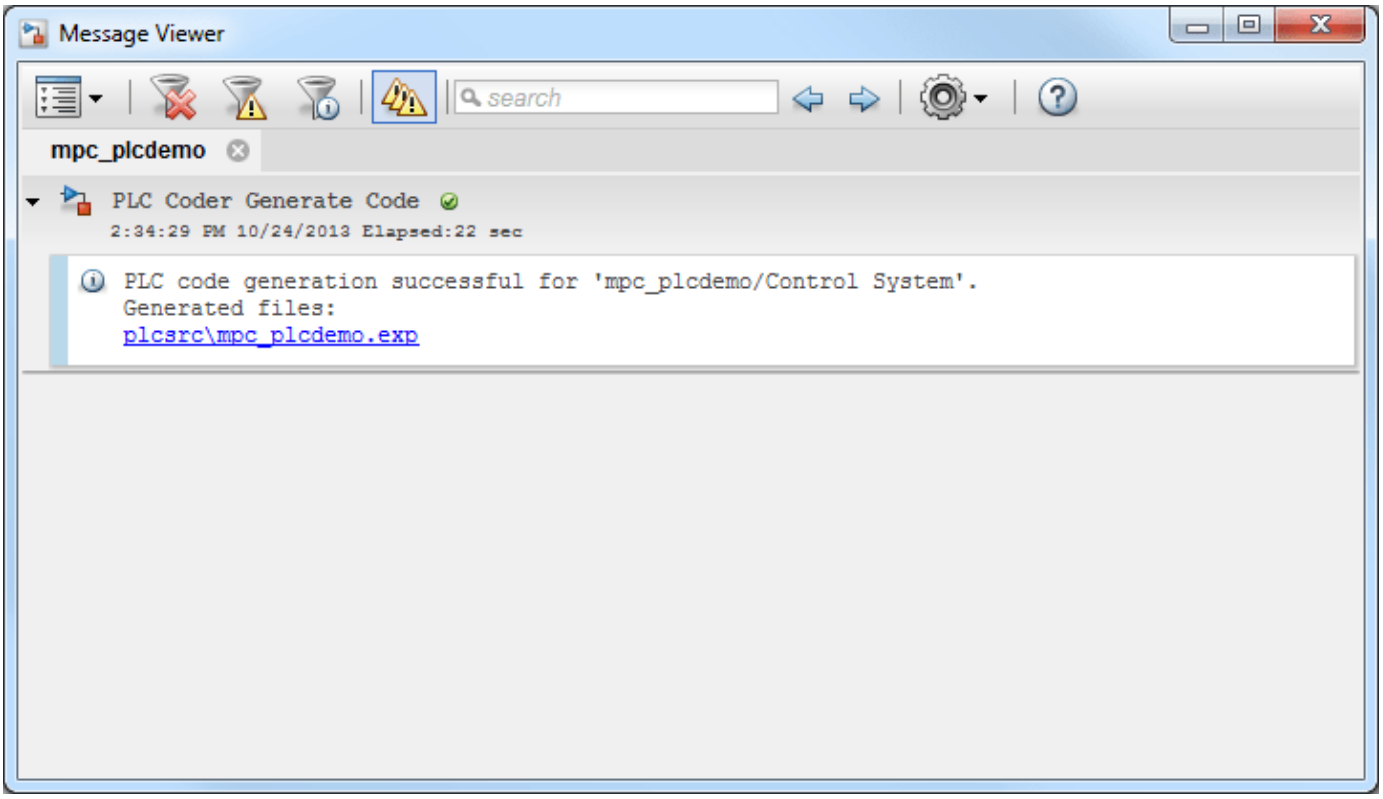


To generate code with the PLC Coder, use the `plcgeneratecode` command.

```
disp('Generating PLC structure text... Please wait until it finishes.')
plcgeneratecode([mdl '/Control System']);
```

```
Generating PLC structure text... Please wait until it finishes.
### Generating PLC code for 'mpc_plcdemo/Control System'.
### Using <a href="matlab:configset.showParameterGroup('mpc_plcdemo', { 'PLC Code Generation' } )
### Begin code generation for IDE <a href="matlab:configset.showParameterGroup('mpc_plcdemo', {
### Emit PLC code to file.
### Creating PLC code generation report <a href="matlab:web('C:\TEMP\Bdoc21b_1757077_3096\ib2EDA
### PLC code generation successful for 'mpc_plcdemo/Control System'.
### Generated files:
<a href="matlab: edit('plcsrc\mpc_plcdemo.exp')">plcsrc\mpc_plcdemo.exp</a>
```

The Message Viewer dialog box shows that PLC code generation was successful.



Close the Simulink model, and return to the original directory.

```
bdclose(mdl)
cd(cwd)
```

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Use the GPU to Compute MPC Moves in MATLAB

This example shows how to generate CUDA code and use the GPU to compute optimal MPC moves in MATLAB™, using the `mpcmoveCodeGeneration` function.

Create Plant Model and Design MPC Controller

Fix random generator seed for reproducibility.

```
rng(0);
```

Create a discrete-time strictly proper plant with 10 states, 3 inputs, and 3 outputs.

```
plant = drss(10,3,3);
plant.D = 0;
```

Create a random initial state for the plant, to be used later in simulation.

```
x0 = rand(10,1);
```

Create an MPC controller with sampling time 0.1 seconds and default prediction and control horizons.

```
mpcobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon = 10.
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

Specify random constraints on the manipulated and measured variables.

```
for ct=1:3
    mpcobj.MV(ct).Min = -1*rand;
    mpcobj.MV(ct).Max = 1*rand;
end

for ct=1:3
    mpcobj.OV(ct).Min = -10*rand;
    mpcobj.OV(ct).Max = 10*rand;
end
```

Get handle to `mpcobj` internal state.

```
xmpc=mpcstate(mpcobj);
```

```
-->No sample time provided for plant model. Assuming sample time = controller's sample time = 0.1
-->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.
-->Assuming output disturbance added to measured output channel #3 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```

Simulate Closed Loop Using `mpcmove`

Before generating the code, simulate the plant in closed loop using `mpcmove` to make sure the behavior is acceptable. For this example, the result of the simulation with `mpcmove` is stored so it can be later compared to the simulation using generated code.

Initialize arrays that will store moves and outputs for later plotting.

```
yMV = [];
uMV = [];
```

Initialize plant and controller states.

```
x = x0;
xmpc.Plant=x0;
```

Run a closed-loop simulation by calling `mpcmove` in a loop for 5 steps.

```
for ct = 1:5
    % Update and store the plant output.
    y = plant.C*x;
    yMV = [yMV y];
    % Compute control actions with ref = ones(1,3)
    u = mpcmove(mpcobj,xmpc,y,ones(1,3));
    % Update and store the plant state.
    x = plant.A*x + plant.B*u;
    uMV = [uMV u];
end
```

Create Data Structures for Code Generation and Simulation

Reset controller initial conditions.

```
xmpc.Plant=x0;
xmpc.Disturbance=zeros(1,3);
xmpc.LastMove=zeros(1,3);
```

Use `getCodeGenerationData` to create the three structures needed for code generation and simulation from the MPC object and its initial state. The `coredata` structure contains the main configuration parameters of the MPC controller that are constant at run time. The `statedata` structure contains the states of the MPC controller, such as for example the state of the plant model, the estimated disturbance, the covariance matrix, and the last control move. The `onlinedata` structure contains data that you must update at each control interval, such as measurement and reference signals, constraints, and weights.

```
[coredata,statedata,onlinedata] = getCodeGenerationData(mpcobj,'InitialState',xmpc);
```

Store the initial state data structure for later re-initialization.

```
statedata0=statedata;
```

Simulate Closed Loop Using `mpcmoveCodeGeneration`

The `mpcmoveCodeGeneration` function allows you to simulate the MPC controller in closed loop in a manner similar to `mpcmove`. It takes in the three controller data structures, where `statedata` and `onlinedata` represent the current values of the controller states and input signals, respectively, and calculates the optimal control move and the new value of the controller states. You can then generate code from `mpcmoveCodeGeneration` (in this example also CUDA code) and compile it to an executable file (in this example one running on the GPU) which has the same inputs and outputs and therefore can be called from MATLAB in exactly the same way.

Initialize arrays to store moves and outputs for later plotting.

```
yCDG = [];
uCDG = [];
```

Initialize plant and controller states.

```
x = x0;
statedata=statedata0;
```

Run a closed-loop simulation by calling `mpcmoveCodeGeneration` in a loop for 5 steps.

```
for ct = 1:5
    % Update and store the plant output.
    y = plant.C*x;
    yCDG = [yCDG y];
    % Update measured output and reference in online data.
    onlinedata.signals.ym = y;
    onlinedata.signals.ref = ones(1,3);
    % Compute control actions.
    [u,statedata] = mpcmoveCodeGeneration(coredata,statedata,onlinedata);
    % Update and store the plant state.
    x = plant.A*x + plant.B*u;
    uCDG = [uCDG u];
end
```

Generate MEX Function for GPU Execution

Create a GPU coder configuration option object using the `coder.gpuConfig` function, and configure the code generation options.

```
CfgGPU = coder.gpuConfig('mex');
CfgGPU.TargetLang = 'C++';
CfgGPU.EnableVariableSizing = false;
CfgCPU.ConstantInputs = 'IgnoreValues';
```

Generate the MEX function `mympcmoveGPU` from the `mpcmoveCodeGeneration` MATLAB function, using the `codegen` command. This command generates CUDA code and compiles it to obtain the MEX executable file `mympcmoveGPU` which runs on the GPU.

```
codegen('-config',CfgGPU,'mpcmoveCodeGeneration','-o','mympcmoveGPU','-args',{coder.Constant(core
```

```
Code generation successful.
```

Simulate Closed Loop Using `mympcmoveGPU`

Initialize arrays that will store moves and outputs for later plotting.

```
yGPU = [];
uGPU = [];
```

Initialize plant and controller states.

```
x = x0;
statedata=statedata0;
```

Run a closed-loop simulation by calling `mympcmoveGPU` in a loop for 5 steps.

```
for ct = 1:5
    % Update and store the plant output.
    y = plant.C*x;
```

```
    yGPU = [yGPU y];
    % Update measured output and reference in online data.
    onlinedata.signals.ym = y;
    onlinedata.signals.ref = ones(1,3);
    % Compute control actions.
    [u,statedata] = mympcmoveGPU(coredata,statedata,onlinedata);
    % Update and store the plant state.
    x = plant.A*x + plant.B*u;
    uGPU = [uGPU u];
end
```

Generate MEX Function for CPU Execution

Create a coder configuration option object using the `coder.Config` function, and configure the code generation options.

```
CfgCPU = coder.config('mex');
CfgCPU.DynamicMemoryAllocation='off';
CfgCPU.EnableVariableSizing = false;
CfgCPU.ConstantInputs = 'IgnoreValues';
```

Generate the MEX function `mympcmoveCPU` from the `mpcmoveCodeGeneration` MATLAB function, using the `codegen` command. This command generates C code and compiles it to obtain the MEX executable file `mympcmoveCPU` which runs on the CPU.

```
codegen('-config',CfgCPU,'mpcmoveCodeGeneration','-o','mympcmoveCPU','-args',{coder.Constant(core
```

```
Code generation successful.
```

Simulate Closed Loop Using `mympcmoveCPU`

Initialize arrays that will store moves and outputs for later plotting.

```
yCPU = [];
uCPU = [];
```

Initialize plant and controller states.

```
x = x0;
statedata=statedata0;
```

Run a closed-loop simulation by calling my`mpcmoveCPU` in a loop for 5 steps.

```
for ct = 1:5
    % Update and store the plant output.
    y = plant.C*x;
    yCPU = [yCPU y];
    % Update measured output and reference in online data.
    onlinedata.signals.ym = y;
    onlinedata.signals.ref = ones(1,3);
    % Compute control actions.
    [u,statedata] = mympcmoveCPU(coredata,statedata,onlinedata);
    % Update and store the plant state.
    x = plant.A*x + plant.B*u;
    uCPU = [uCPU u];
end
```

Compare MPC Moves

First, compare the plant inputs and outputs obtained from `mpcmove` and the ones obtained using the GPU.

```
uGPU-uMV
```

```
ans = 3×5
10^-15 ×

    0.3886    0.1110    0.3886   -0.2220   -0.7772
   -0.2776   -0.0555   -0.3886   -0.7494    0.0694
   -0.0035    0.0555    0.2359    0.0278   -0.0833
```

```
yGPU-yMV
```

```
ans = 3×5
10^-14 ×

         0   -0.1110   -0.0333   -0.1332   -0.0333
         0   -0.1776   -0.0444   -0.4219   -0.2665
         0   -0.0222    0.0222   -0.0666    0.1332
```

The simulation results are identical, except for negligible numerical errors, to those using `mpcmove`.

```
uCPU-uMV
```

```
ans = 3×5
10^-14 ×

    0.0278    0.0389   -0.0167   -0.0389   -0.0167
    0.0611   -0.0056   -0.0749   -0.1013    0.0291
         0    0.0333    0.0833   -0.0111   -0.0611
```

```
yCPU-yMV
```

```
ans = 3×5
10^-14 ×

         0    0.0111   -0.0777    0.1554   -0.0444
         0    0.2331         0   -0.2887   -0.1998
         0   -0.1332   -0.0444   -0.1332    0.0666
```

Similarly the difference between results obtained by running the `mpcmoveCodeGeneration` in MATLAB and running the generated code on the CPU is negligible.

```
uCPU-uCDG
```

```
ans = 3×5
10^-14 ×

         0   -0.1554   -0.1887   -0.1055   -0.0500
   -0.0444         0   -0.0111   -0.0985   -0.0180
         0    0.0278   -0.0111   -0.0444   -0.1027
```

```
yCPU-yCDG

ans = 3×5
10^-14 ×

         0   -0.0555    0.4663    0.3997    0.3553
         0   -0.1443    0.1332    0.0444   -0.2220
         0    0.0666    0.0888    0.0666    0.3331
```

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Use the GPU to Simulate an MPC Controller in Simulink

This example shows how to generate CUDA code and use the GPU to compute optimal MPC moves in Simulink™.

Create Plant Model and Design MPC Controller

Use a double integrator as a plant.

```
plant = tf(1,[1 0 0]);
```

Create an MPC object for the plant with a sampling time of 0.1 seconds, and prediction and control horizon of 10 and 3 steps, respectively.

```
mpcobj = mpc(plant, 0.1, 10, 3);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming default 0.00000
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming default 0.1
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1.00000.
```

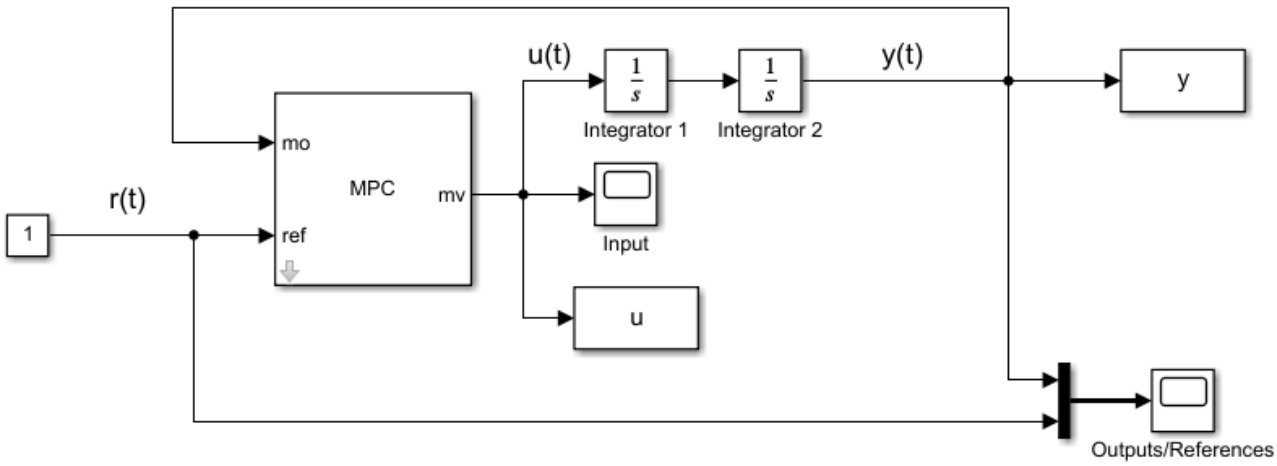
Limit the manipulated variable between –1 and 1.

```
mpcobj.MV = struct('Min',-1,'Max',1);
```

Control the Plant Model in Simulink

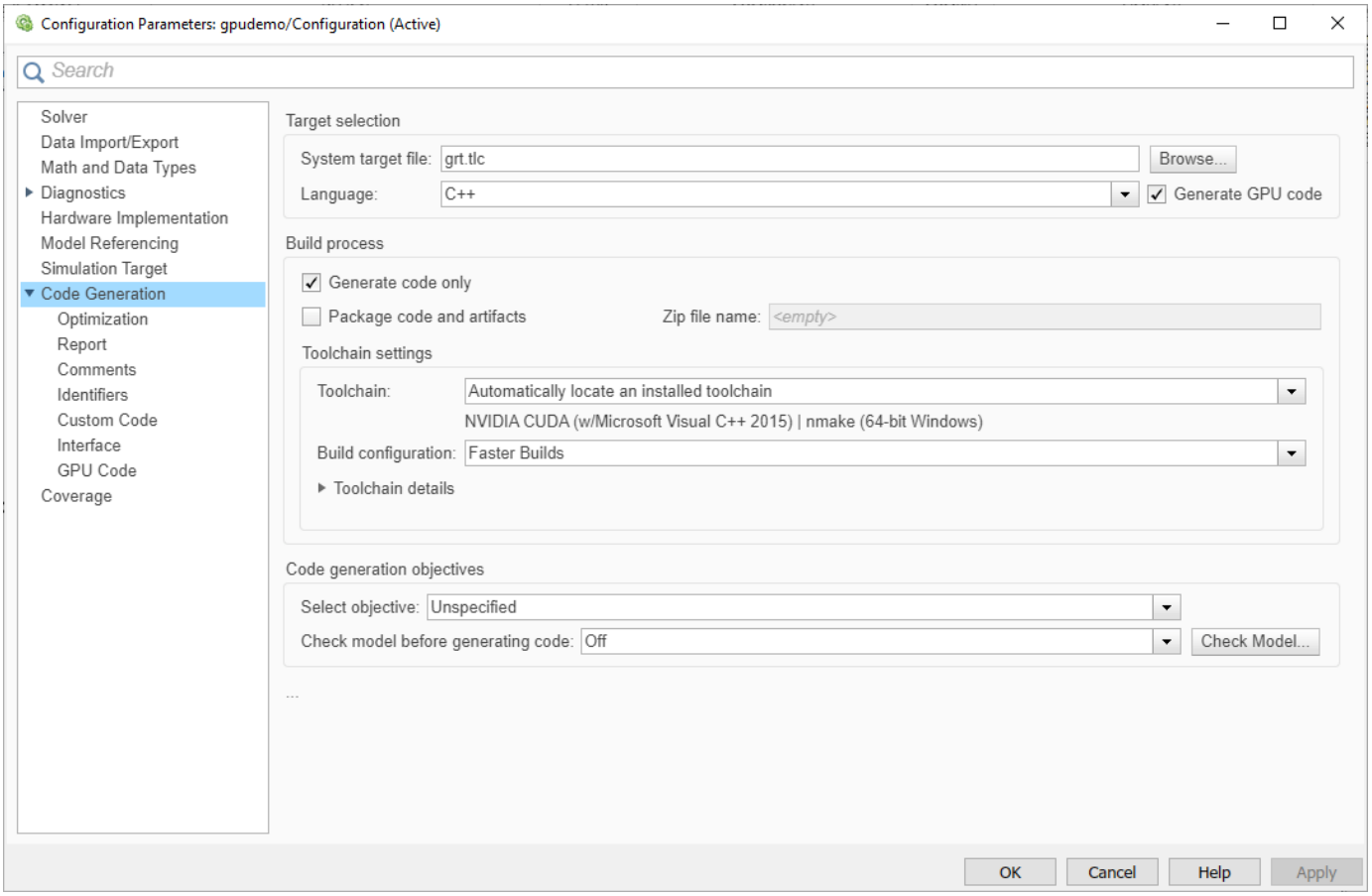
Create a Simulink closed loop simulation using the MPC Controller block, with the `mpcobj` object passed as a parameter, to control the double integrator plant. For this example, open the pre-existing `gpudemo` Simulink model.

```
open_system('gpudemo')
```



Copyright 1990-2014 The MathWorks, Inc.

Open the Configuration Parameters dialog box by clicking **Model Settings**. Then, in the **Code Generation** section, select **Generate GPU code**.

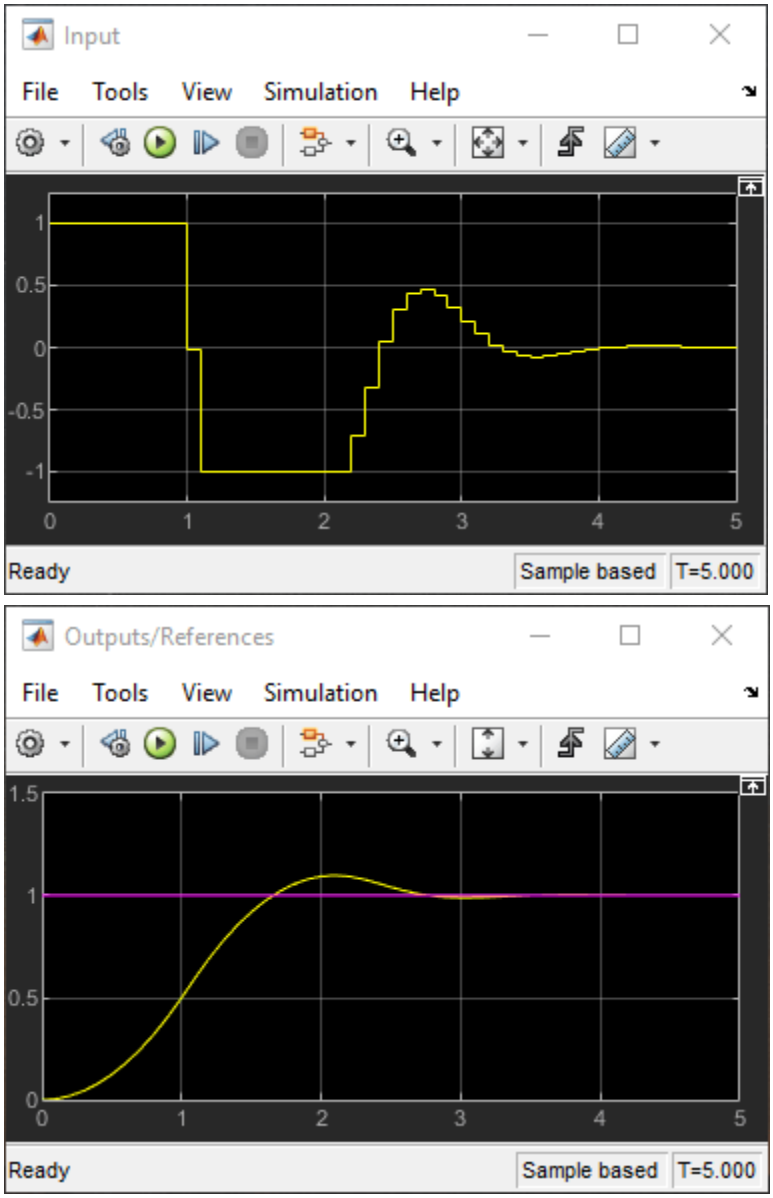


You can now run the model by clicking **Run** or by using the MATLAB command `sim`. Before running the simulation the model will generate CUDA code from the Simulink model and compile it to obtain a MEX executable. When the model is simulated, this file is called and the simulation is performed on the GPU.

```
sim('gpudemo')
```

```
-->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 measured
```

After the simulation, the plots of the two scopes show that the manipulated variable does not exceed the limit and the plant output tracks the reference signal after approximately 3 seconds.



Here, the entire Simulink model is executed on the GPU. To deploy only the MPC block on the GPU, you can create a model having only the MPC block inside. Typically in embedded control modules, the deployed model contains the controller block plus a few interface blocks for input/output signals.

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Using MPC Controller Block Inside Function-Call and Triggered Subsystems

This example shows how to configure and simulate MPC Controller blocks placed inside Function-Call and Triggered subsystems.

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);
```

Configure and Simulate MPC Controller Block Inside Function-Call Subsystem

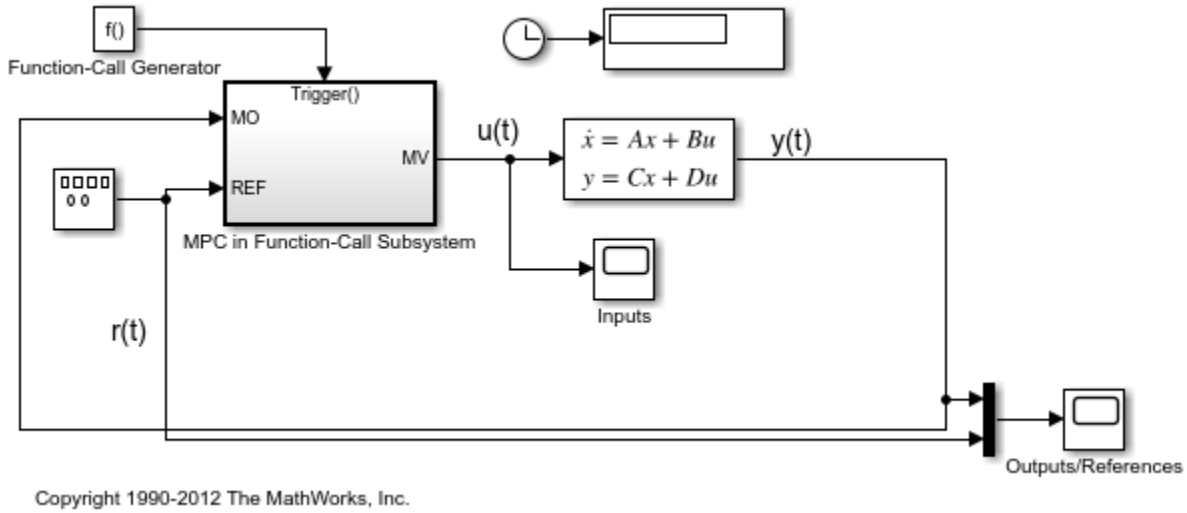
To run this example, Simulink® is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

Function-Call subsystem is invoked directly by another block during simulation. To ensure that the MPC controller works properly inside a Function-Call subsystem, you must configure the MPC Controller block to use inherited sample time and invoke the Function-Call subsystem periodically with the same sample time defined in the MPC controller object.

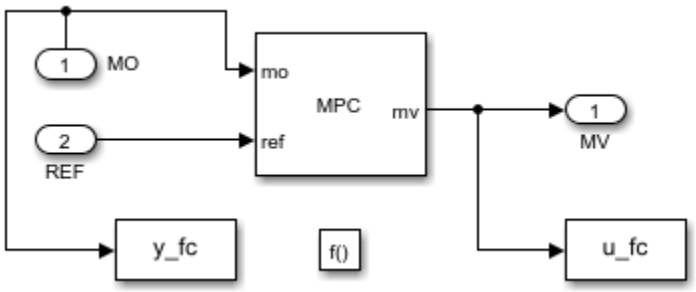
Open the model.

```
mdl1 = 'mpc_rtwdemo_functioncall';
open_system(mdl1)
```



The MPC Controller block is inside the MPC in Triggered Subsystem block.

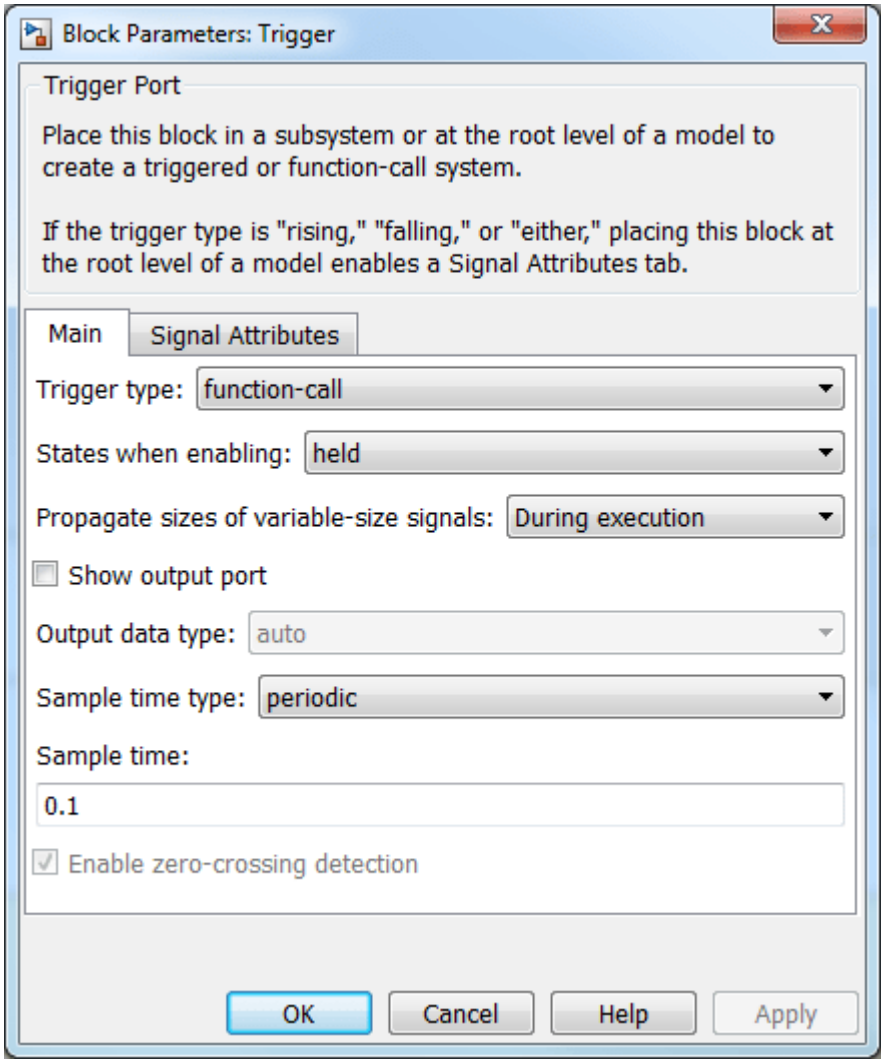
```
open_system([mdl1 '/MPC in Function-Call Subsystem'])
```



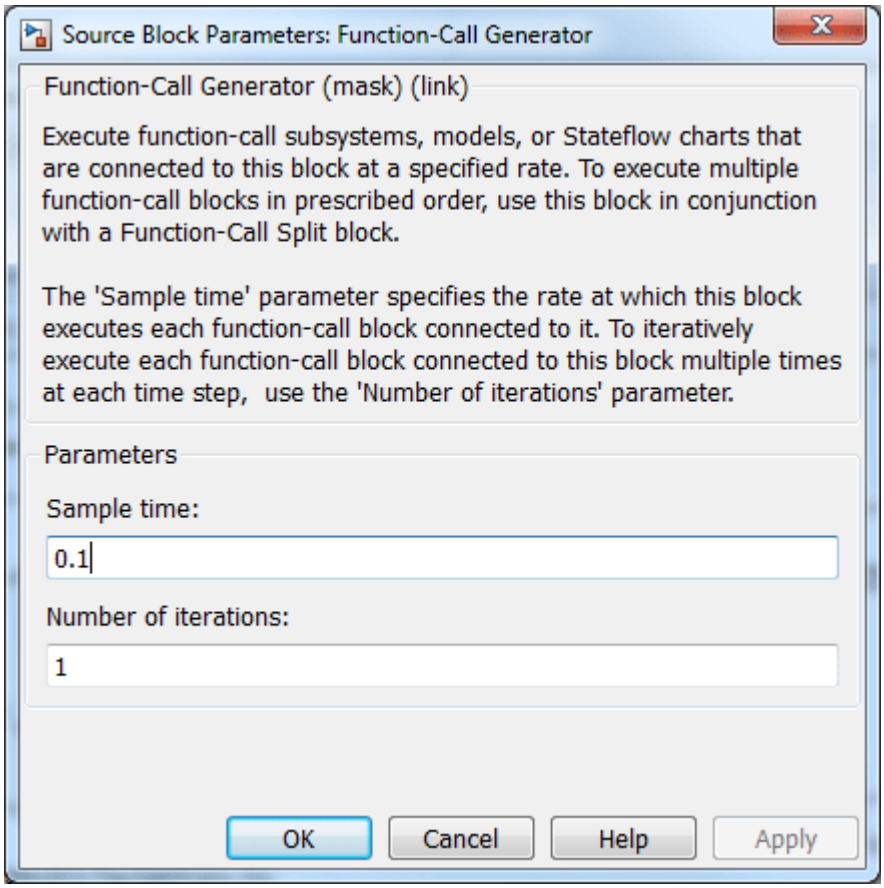
Configure the controller to use an inherited sample time. To do so, select the **Inherit sample time** property of the MPC Controller block.

Invoke the Function-Call subsystem periodically with the correct sample time.

For this example, since the controller has a sample time of 0.1 seconds, configure the trigger block inside the Function-Call subsystem to use the same sample time.



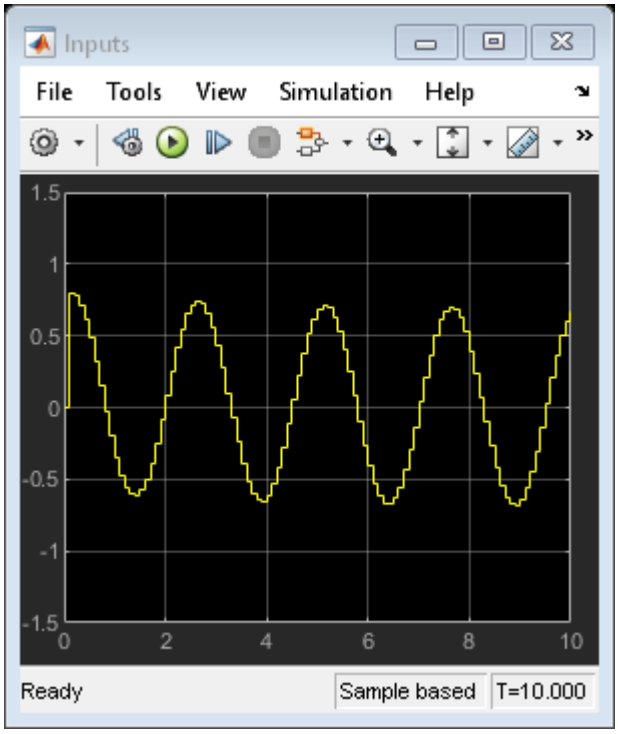
For this example, use the Function-Call Generator block to execute the Function-Call subsystem at the sample rate as 0.1 seconds.

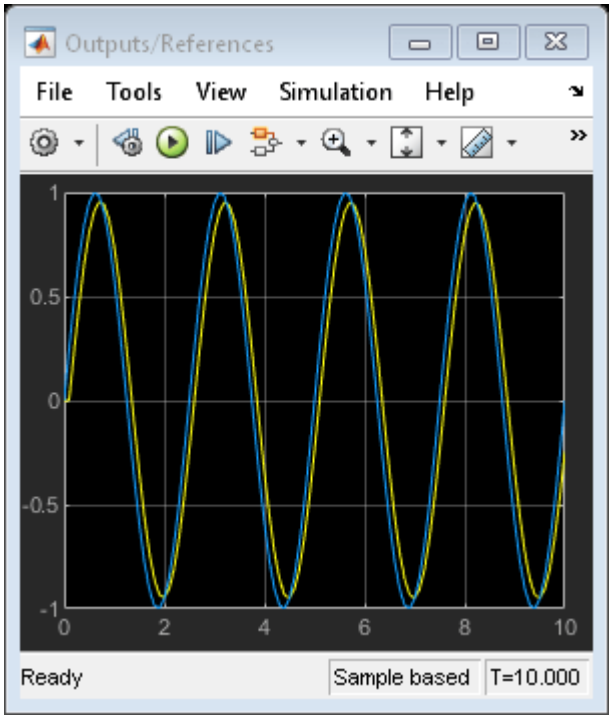


Simulate the model.

```
close_system([mdl1 '/MPC in Function-Call Subsystem/MPC Controller'])
open_system([mdl1 '/Inputs'])
open_system([mdl1 '/Outputs//References'])
sim(mdl1)
```

```
-->Converting model to discrete time.
-->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 measured
```





The controller effort and the plant output are saved into base workspace as variables `u_fc` and `y_fc`, respectively.

Close the Simulink model.

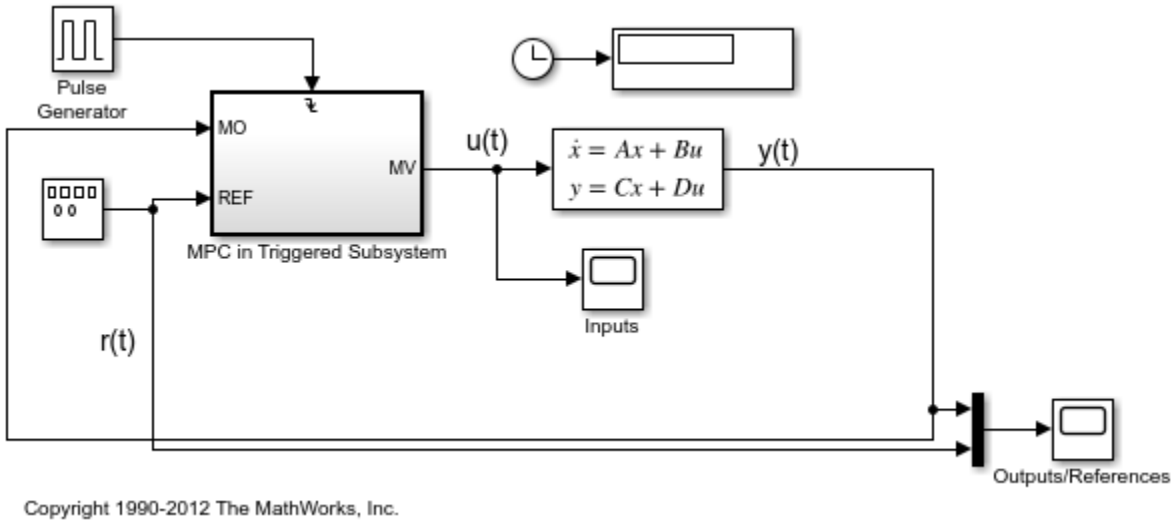
```
bdclose(mdl1)
```

Configure and Simulate MPC Controller Block Inside Triggered Subsystem

Triggered subsystem executes each time a trigger event occurs. To ensure that the MPC controller works properly inside a triggered subsystem, you must configure the MPC Controller block to use inherited sample time and invoke the Triggered subsystem periodically with the same sample time defined in the MPC controller object.

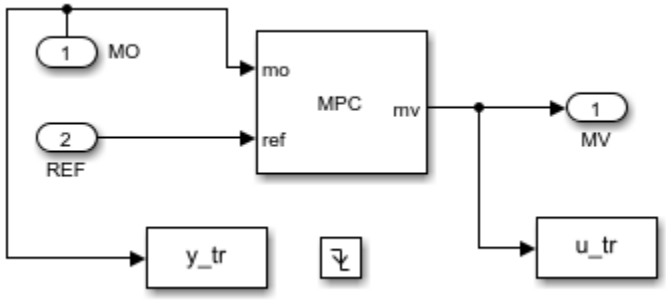
Open the model.

```
mdl2 = 'mpc_rtwdemo_triggered';
open_system(mdl2)
```



The MPC Controller block is in the MPC in Triggered Subsystem block.

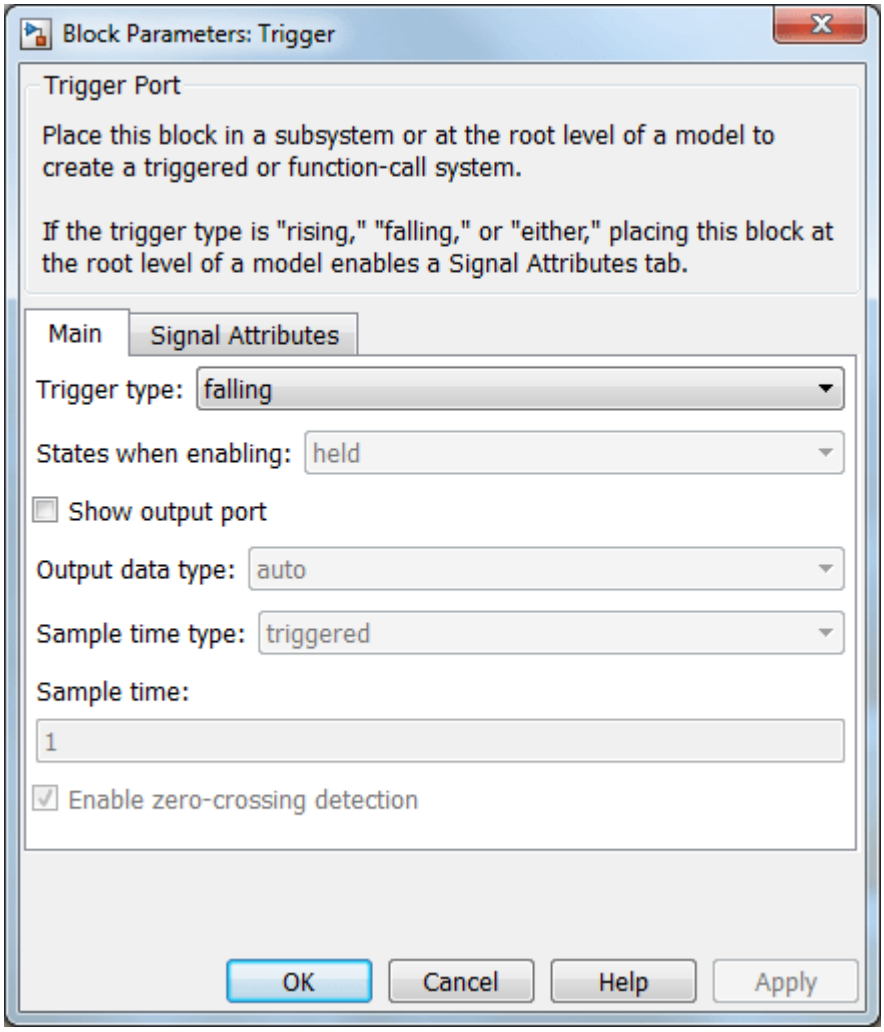
```
open_system([mdl2 '/MPC in Triggered Subsystem']);
```



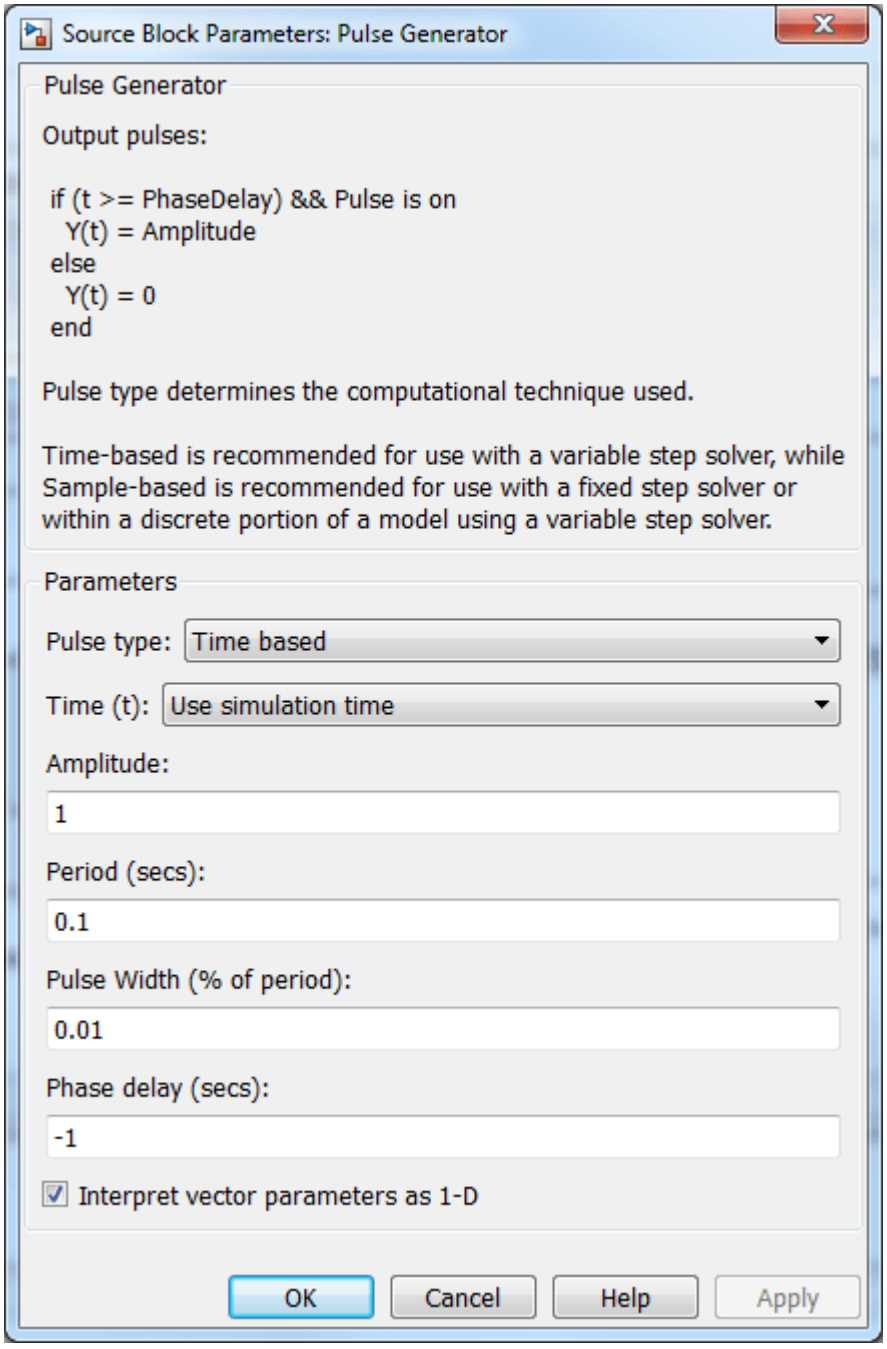
Configure the MPC block to use an inherited sample time, as you did for the function-call subsystem model.

Execute the Triggered subsystem periodically with the correct sample time.

For this example, configure the Trigger block inside the triggered subsystem to use a `falling` trigger type.

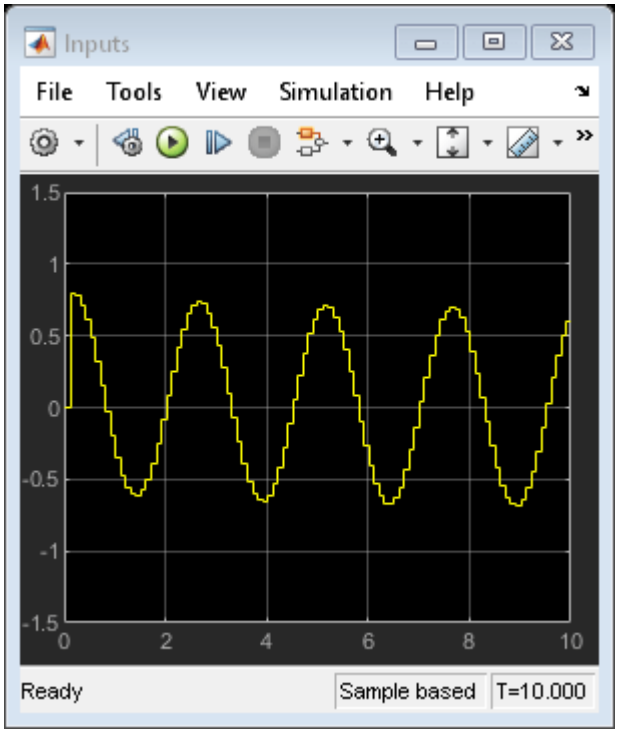


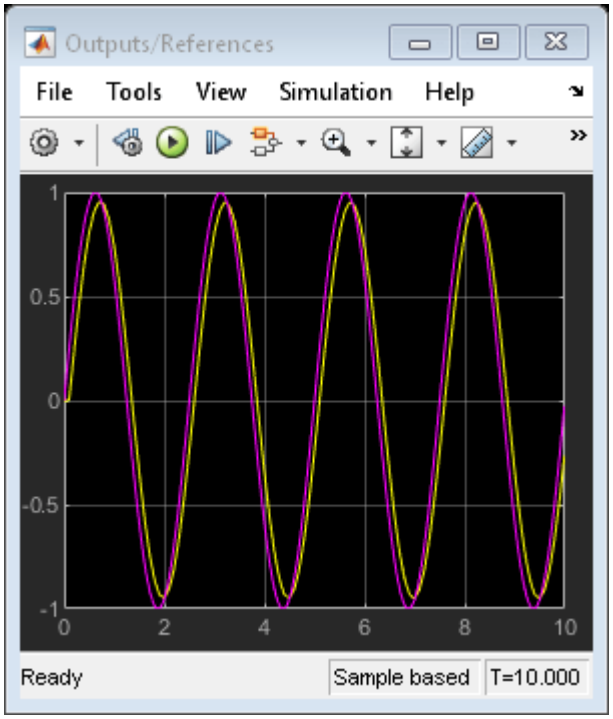
For this example, use the Pulse Generator block to provide a periodic triggering signal at the sample rate as 0.1 seconds.



Simulate the model.

```
close_system([mdl2 '/MPC in Triggered Subsystem/MPC Controller'])
open_system([mdl2 '/Inputs'])
open_system([mdl2 '/Outputs//References'])
sim(mdl2)
```





The controller effort and the plant output are saved into base workspace as variables `u_tr` and `y_tr`, respectively.

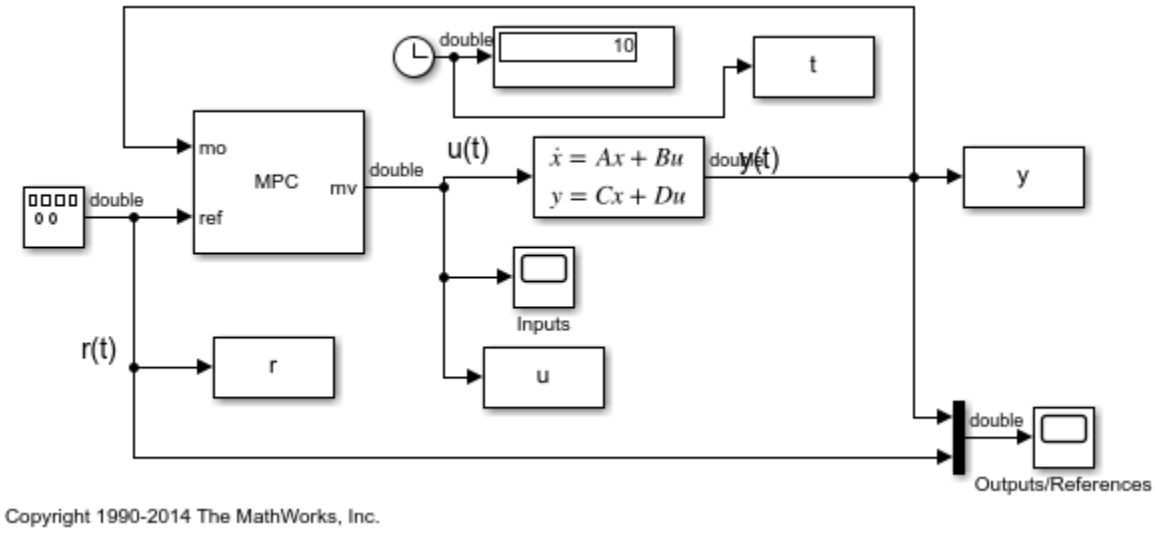
Close the Simulink model.

```
bdclose(mdl2)
```

Compare Responses

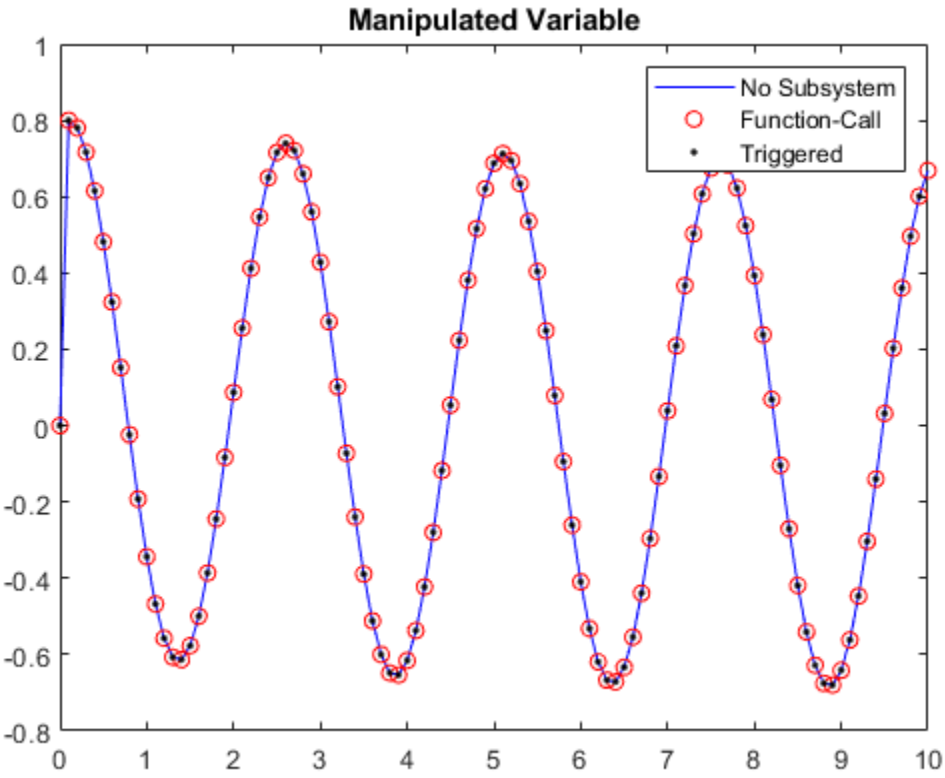
Compare the simulation results from the Function-Call subsystem and the Triggered subsystem with the result generated by an MPC Controller block that is not placed inside a subsystem and does not inherit sample time.

```
mdl = 'mpc_rtwdemo';
open_system(mdl)
sim(mdl)
```



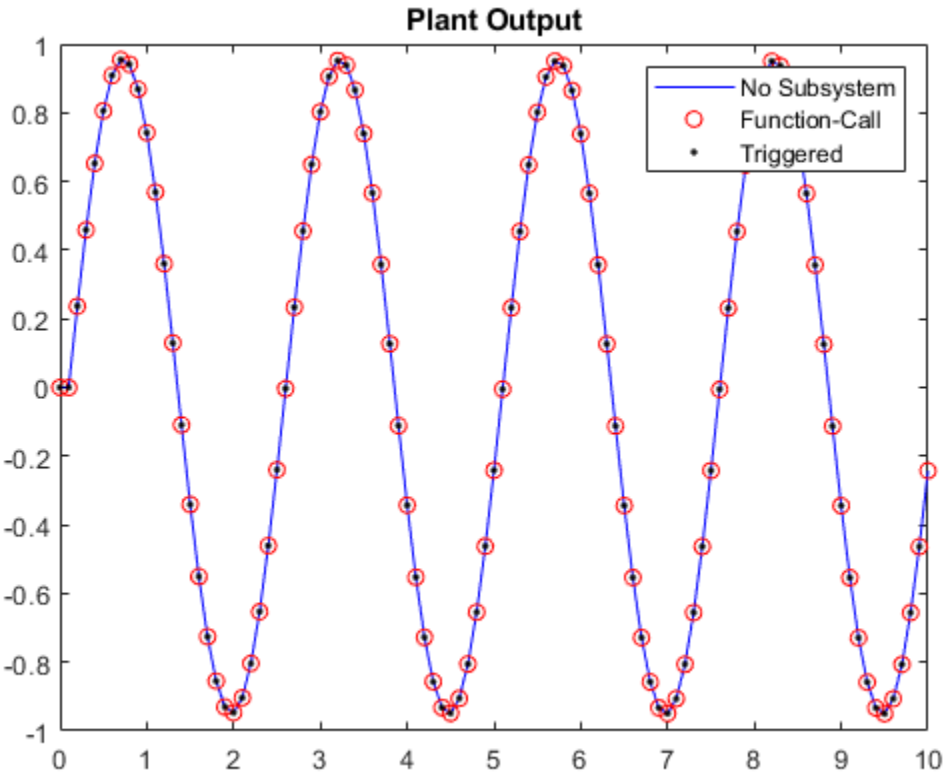
Compare the responses of the manipulated variable.

```
figure
plot(t,u,'b-',t,u_fc,'ro',t(1:end-1),u_tr,'k.')
title('Manipulated Variable')
legend('No Subsystem','Function-Call','Triggered')
```



Compare the responses the plant output.

```
figure
plot(t,y,'b-',t,y_fc,'ro',t(1:end-1),y_tr,'k.')
title('Plant Output')
legend('No Subsystem','Function-Call','Triggered')
```



The results of all three models are numerically equal.

Close the Simulink model.

```
bdclose(mdl)
```

See Also

Function-Call Subsystem | Triggered Subsystem

More About

- “Choose Sample Time and Horizons” on page 1-2

Solve Custom MPC Quadratic Programming Problem and Generate Code

This example shows how to use the built-in active-set QP solver to implement a custom MPC algorithm that supports C code generation in MATLAB.

Define Plant Model

The plant model is a discrete-time state-space system and it is open-loop unstable. We assume that all the plant states are measurable. Therefore, we avoid the need for designing a state estimator, which is beyond the scope of this example.

```
A = [1.1 2; 0 0.95];
B = [0; 0.0787];
C = [-1 1];
D = 0;
Ts = 1;
sys = ss(A,B,C,D,Ts);
x0 = [0.5;-0.5]; % initial states at [0.5 -0.5]
```

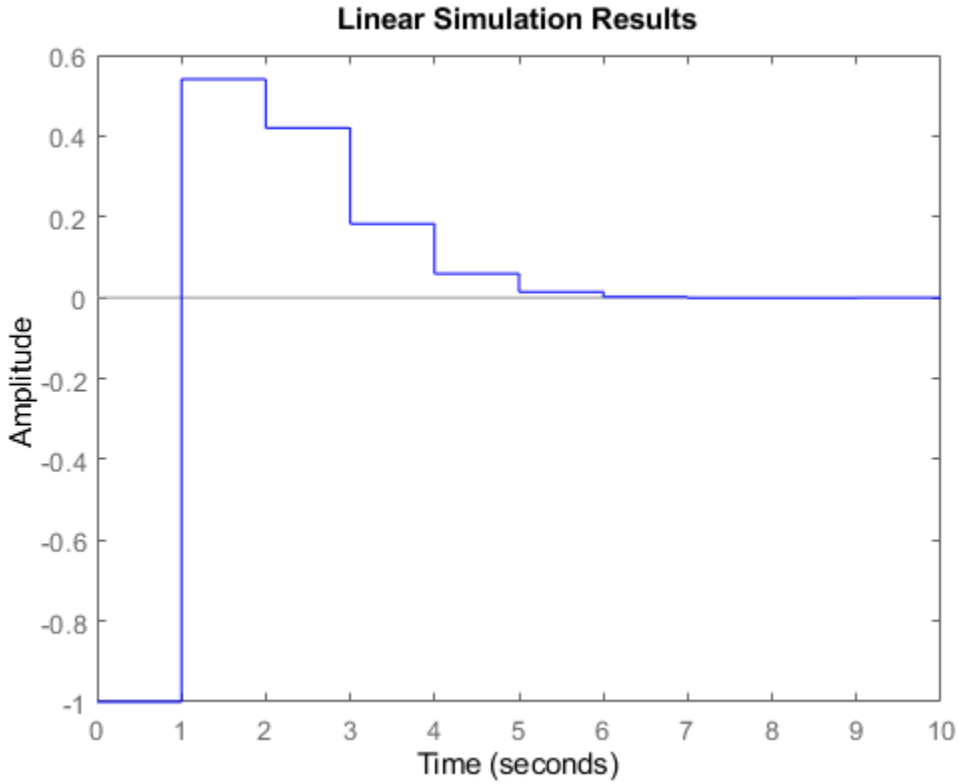
Design Unconstrained Linear Quadratic Regulator (LQR)

Design an unconstrained LQR with output weighting. This controller serves as the baseline to compare with the custom MPC algorithm. The LQ control law is `u(k) = -K_lqr*x(k)`.

```
Qy = 1;
R = 0.01;
K_lqr = lqry(sys,Qy,R);
```

Run a simulation with initial states at [0.5 -0.5]. The closed-loop response is stable.

```
t_unconstrained = 0:1:10;
u_unconstrained = zeros(size(t_unconstrained));
Unconstrained_LQR = tf([-1 1])*feedback(ss(A,B,eye(2),0,Ts),K_lqr);
lsim(Unconstrained_LQR,'-',u_unconstrained,t_unconstrained,x0);
hold on
```



Design Custom MPC Controller with Terminal Weight

Design a custom MPC controller with the terminal weight applied at the last prediction step.

The predicted state sequences, X(k), generated by the linear model and input sequence, U(k), can be formulated as: `X(k) = M*x(k) + CONV*U(k)`. In this example, use four prediction steps (N = 4).

```
M = [A;A^2;A^3;A^4];
CONV = [B zeros(2,1) zeros(2,1) zeros(2,1);...
        A*B B zeros(2,1) zeros(2,1);...
        A^2*B A*B B zeros(2,1);...
        A^3*B A^2*B A*B B];
```

The MPC objective function is `J(k) = sum(x(k)'*Q*x(k) + u(k)'*R*u(k) + x(k+N)'*Q_bar*x(k+N))`. To ensure that the MPC objective function has the same quadratic cost as the infinite horizon quadratic cost used by LQR, terminal weight `Q_bar` is obtained by solving the following Lyapunov equation:

```
Q = C'*C;
Q_bar = dlyap((A-B*K_lqr)', Q+K_lqr'*R*K_lqr);
```

Convert the MPC problem into a standard QP problem, which has the objective function `J(k) = U(k)'*H*U(k) + 2*x(k)'*F'*U(k)`.

```
Q_hat = blkdiag(Q,Q,Q,Q_bar);
R_hat = blkdiag(R,R,R,R);
H = CONV'*Q_hat*CONV + R_hat;
F = CONV'*Q_hat*M;
```

When there are no constraints, the optimal predicted input sequence U(k) generated by MPC controller is `-K*x`, where `K = inv(H)*F`.

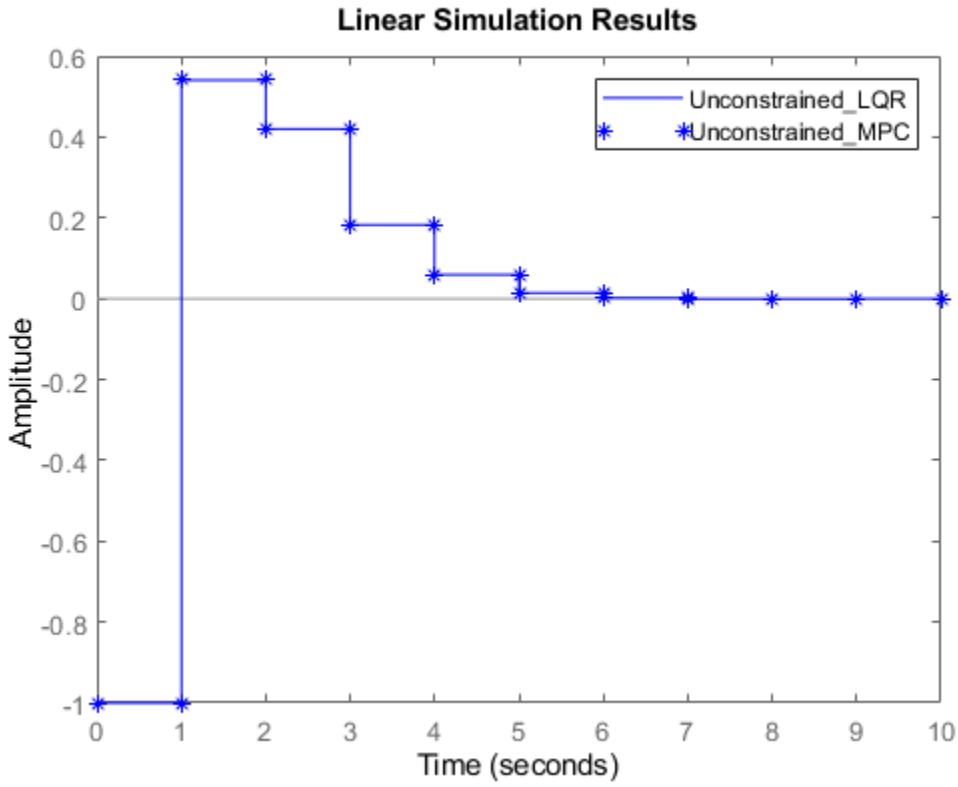
```
K = H\F;
```

In practice, only the first control move `u(k) = -K_mpc*x(k)` is applied to the plant (receding horizon control).

```
K_mpc = K(1,:);
```

Run a simulation with initial states at [0.5 -0.5]. The closed-loop response is stable.

```
Unconstrained_MPC = tf([-1 1])*feedback(ss(A,B,eye(2),0,Ts),K_mpc);
lsim(Unconstrained_MPC,'*',u_unconstrained,t_unconstrained,x0)
legend show
```



LQR and MPC controllers produce the same result because the control laws are the same.

```
K_lqr
K_mpc
```

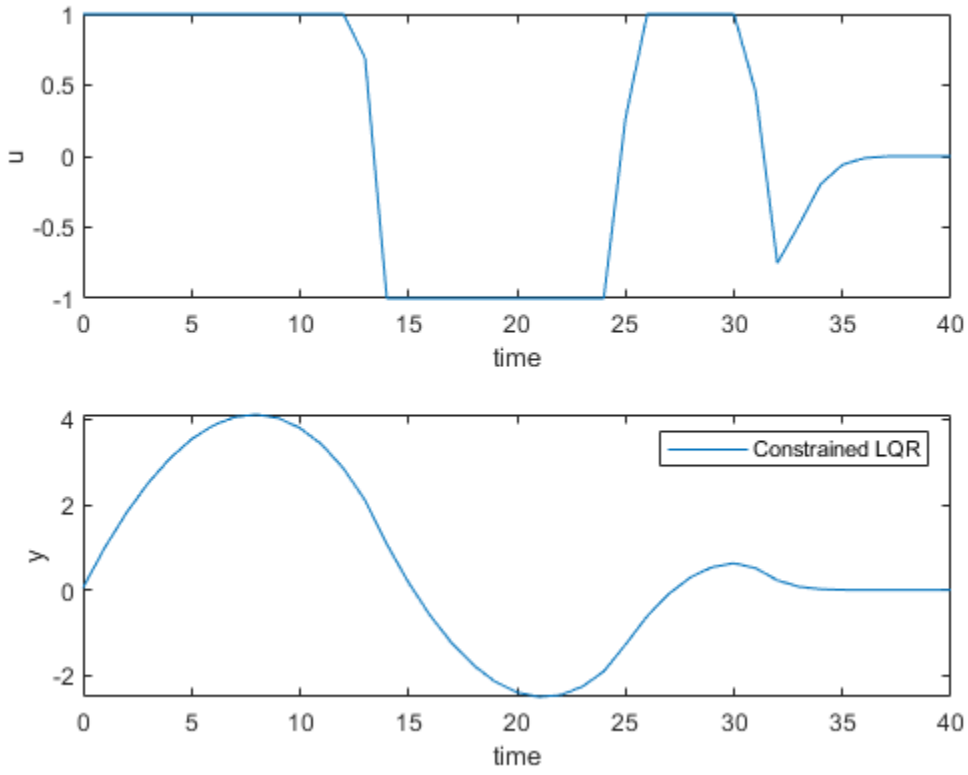
```
K_lqr =

    4.3608   18.7401


K_mpc =
```

```
    4.3608   18.7401
```

LQR Control Performance Deteriorates When Applying Constraints

Restrict the controller output, u(k), to be between -1 and 1. The LQR controller generates a slow and oscillatory closed-loop response due to saturation.

```
x = x0;
t_constrained = 0:40;
for ct = t_constrained
    uLQR(ct+1) = -K_lqr*x;
    uLQR(ct+1) = max(-1,min(1,uLQR(ct+1)));
    x = A*x+B*uLQR(ct+1);
    yLQR(ct+1) = C*x;
end
figure
subplot(2,1,1)
plot(t_constrained,uLQR)
xlabel('time')
ylabel('u')
subplot(2,1,2)
plot(t_constrained,yLQR)
xlabel('time')
ylabel('y')
legend('Constrained LQR')
```



MPC Controller Solves QP Problem Online When Applying Constraints

One of the major benefits of using MPC controller is that it handles input and output constraints explicitly by solving an optimization problem at each control interval.

Use the built-in KWIK QP solver, `mpcActiveSetSolver`, to implement the custom MPC controller designed above. The constraint matrices are defined as Ac*x>=b0.

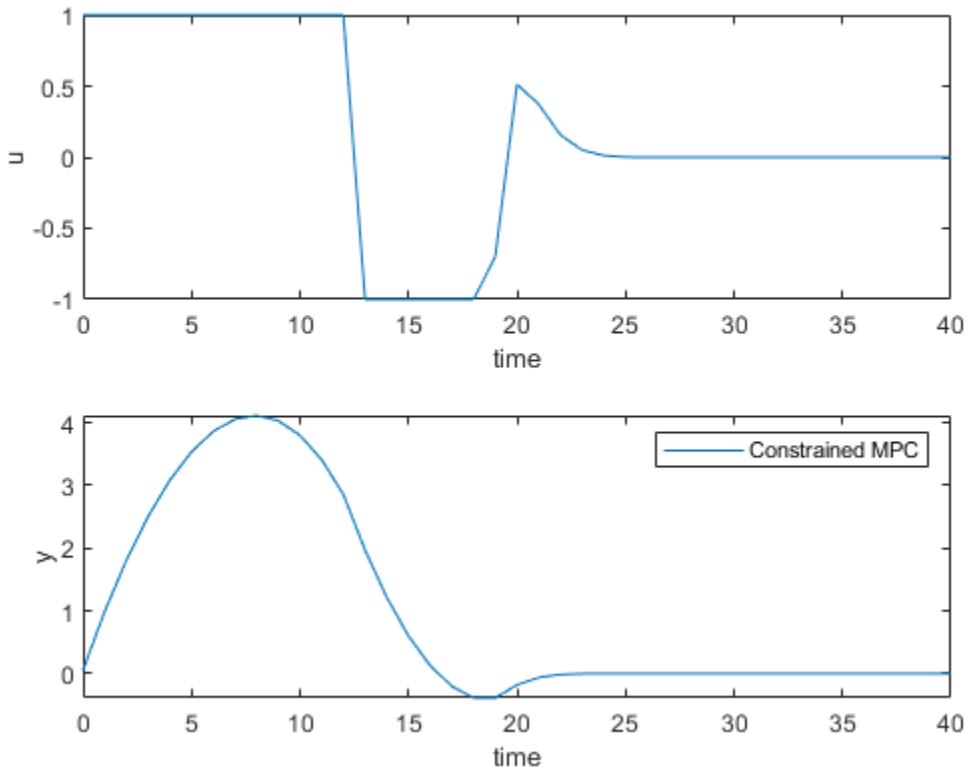
```
Ac = [1 0 0 0;...
      -1 0 0 0;...
       0 1 0 0;...
       0 -1 0 0;...
       0 0 1 0;...
       0 0 -1 0;...
       0 0 0 1;...
       0 0 0 -1];
b0 = [1;1;1;1;1;1;1;1];
```

Since in this case the Hessian matrix H is constant, you can precalculate the inverse of its lower-triangular Cholesky decomposition, and then pass it to the `mpcActiveSetSolver` function, instead of passing the Hessian matrix directly. As a result, `mpcActiveSetSolver` can avoid performing this computation at each time step.

```
L = chol(H,'lower');
Linv = L\eye(size(H,1));
```

Run a simulation by calling `mpcActiveSetSolver` at each simulation step. Initially all the inequalities are inactive (cold start).

```
x = x0;
iA = false(size(b0));

% create options for the solver, and specify non-hessian first input
opt = mpcActiveSetOptions;
opt.IntegrityChecks = false;
opt.UseHessianAsInput = false;

for ct = t_constrained
    [u,status,iA] = mpcActiveSetSolver(Linv,F*x,Ac,b0,[],zeros(0,1),iA,opt);
    uMPC(ct+1) = u(1);
    x = A*x+B*uMPC(ct+1);
    yMPC(ct+1) = C*x;
end
figure
subplot(2,1,1)
plot(t_constrained,uMPC)
xlabel('time')
ylabel('u')
subplot(2,1,2)
plot(t_constrained,yMPC)
xlabel('time')
ylabel('y')
legend('Constrained MPC')
```

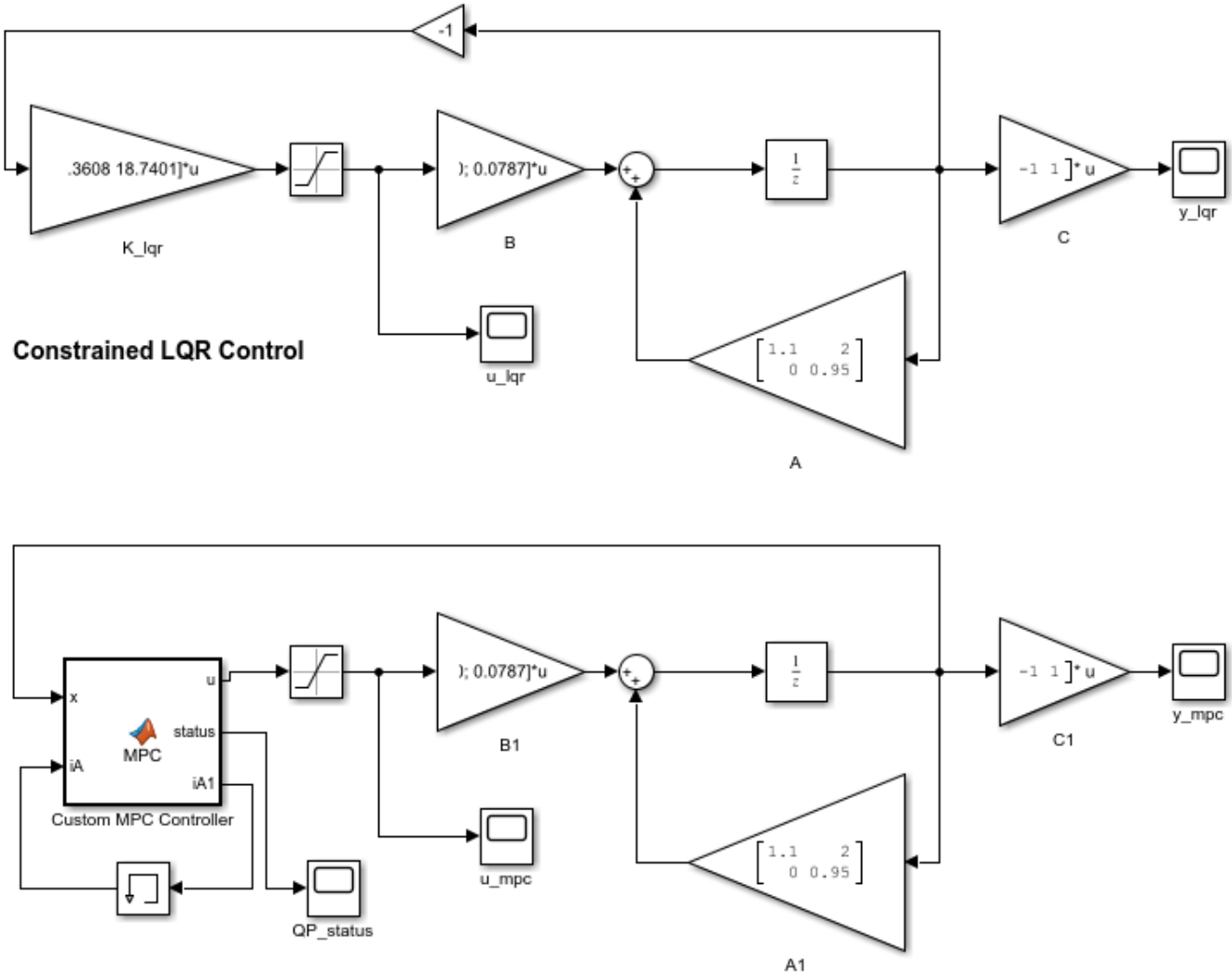


The MPC controller produces a closed-loop response with faster settling time and less oscillation.

Simulate Custom MPC Using MATLAB Function Block in Simulink

`mpcActiveSetSolver` can be used inside a MATLAB Function block to provide simulation and code generation in the Simulink environment.

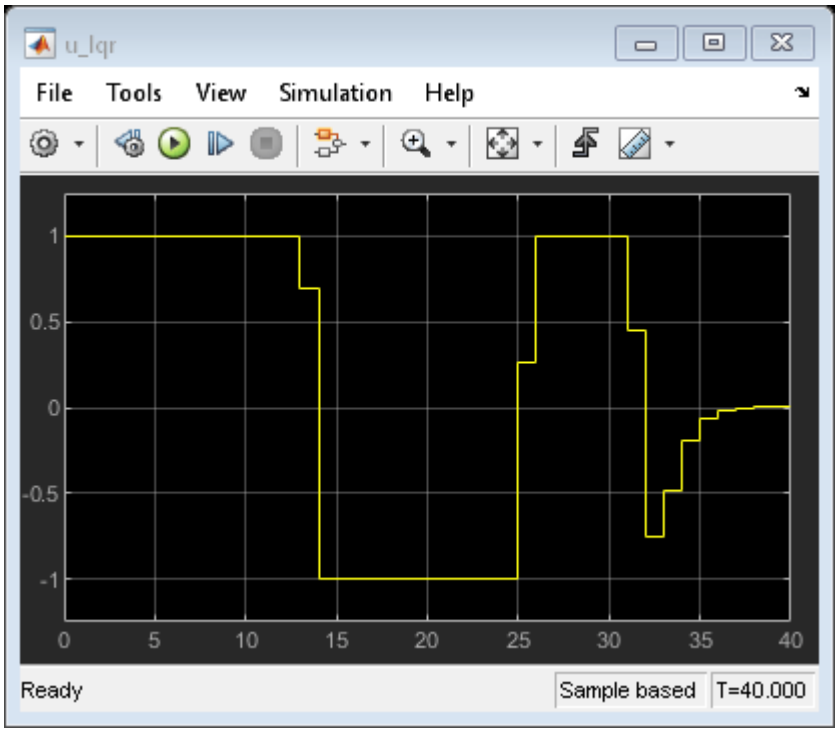
```
mdl = 'mpc_activesetqp';
open_system(mdl)
```

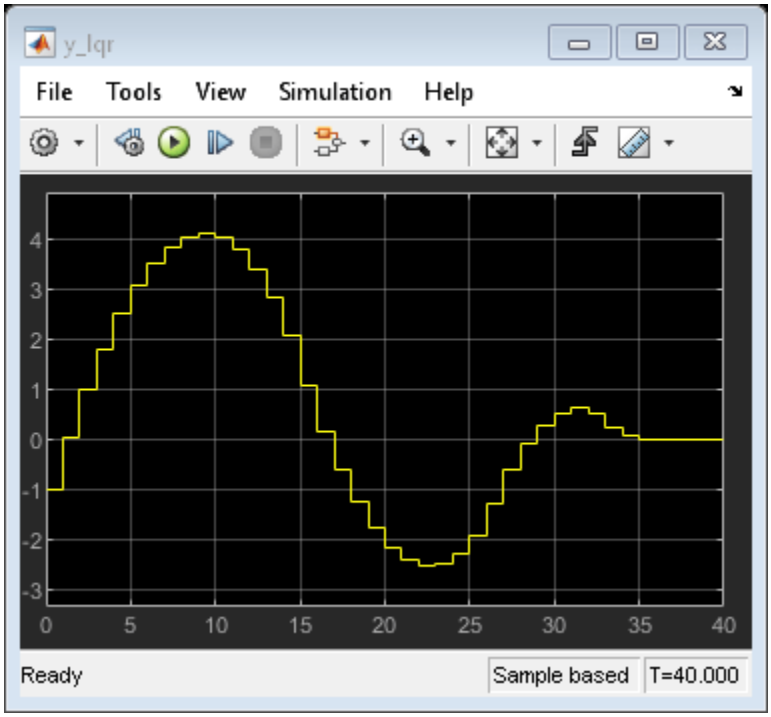


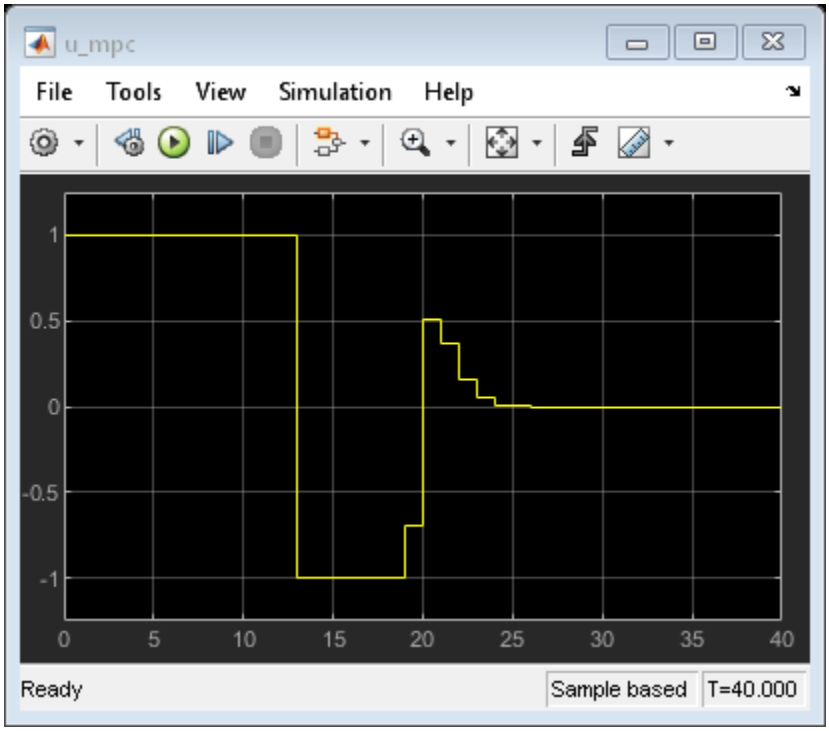
The Custom MPC Controller block is a MATLAB Function block. To examine the MATLAB code, double-click the block. Since `Linv`, `F`, `Ac`, `b0` matrices, and `opt` structure are constant, they are passed into the MATLAB Function block as parameters.

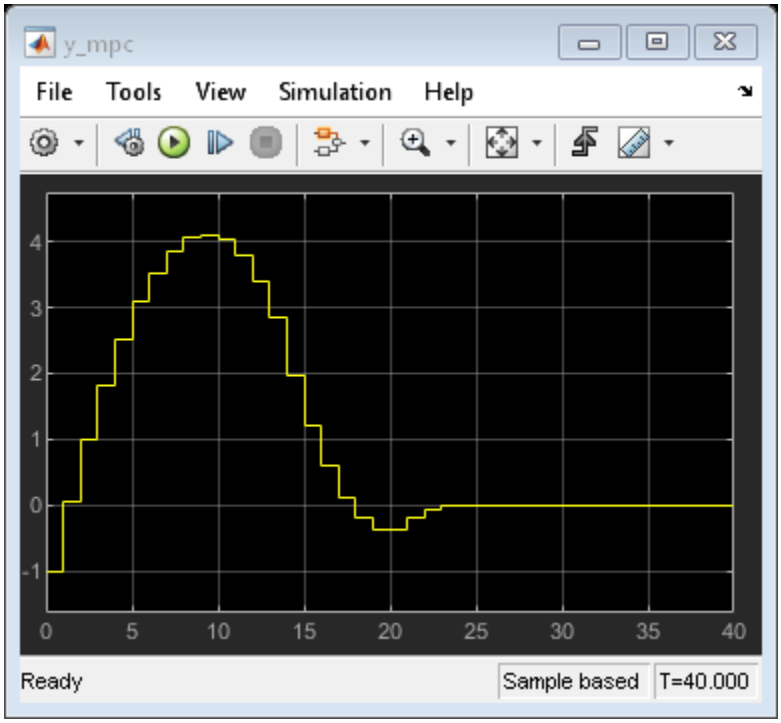
Run a simulation in Simulink. The closed-responses of LQR and MPC controllers are identical to their counterparts in the MATLAB simulation.

```
open_system([mdl '/u_lqr'])
open_system([mdl '/y_lqr'])
open_system([mdl '/u_mpc'])
open_system([mdl '/y_mpc'])
sim(mdl)
```


u_lqr
File Tools View Simulation Help
1
0.5
0
-0.5
-1
0 5 10 15 20 25 30 35 40
Ready
Sample based
T=40.000


y_lqr
File Tools View Simulation Help
4
3
2
1
0
-1
-2
-3
0 5 10 15 20 25 30 35 40
Ready
Sample based
T=40.000





Code Generation in MATLAB

`mpcActiveSetSolver` supports C code generation with MATLAB Coder. Assume you have a function, `mycode`, that is compatible with the code generation standard.

```
function [x,iter,iA1,lam] = mycode()
%#codegen
n = 5;
m = 10;
q = 2;
H = diag(10*rand(n,1));
f = randn(n,1);
A = randn(m,n);
b = randn(m,1);
Aeq = randn(q,n);
beq = randn(q,1);
Linv = chol(H,'lower')\eye(n);
iA = false(m,1);
Opt = mpcActiveSetOptions();
[x,iter,iA1,lam] = mpcActiveSetSolver(Linv,f,A,b,Aeq,beq,iA,Opt);
```

You can use following command to generate C code with MATLAB Coder:

```
fun = 'mycode';
Cfg = coder.config('mex'); % or 'lib', 'dll', etc.
codegen('-config',Cfg,fun,'-o',fun);
```

Acknowledgment

This example is inspired by Professor Mark Cannon's lecture notes for the Model Predictive Control class at University of Oxford. The plant model is the same one used in Example 2.1 in the "Prediction and optimization" section.

```
bdclose(mdl)
```

See Also

`mpcqpsolver` | `mpcqpsolverOptions`

More About

- “QP Solvers” on page 2-17

Simulate and Generate Code for MPC Controller with Custom QP Solver

This example shows how to simulate and generate code for a model predictive controller that uses a custom quadratic programming (QP) solver. The plant for this example is a dc-servo motor in Simulink®.

DC-Servo Motor Model

The dc-servo motor model is a linear dynamic system described in [1]. `plant` is the continuous-time state-space model of the motor. `tau` is the maximum admissible torque, which you use as an output constraint.

```
[plant,tau] = mpcmotormodel;
```

Design MPC Controller

The plant has one input, the motor input voltage. The MPC controller uses this input as a manipulated variable (`MV`). The plant has two outputs, the motor angular position and shaft torque. The angular position is a measured output (`MO`), and the shaft torque is unmeasured (`UO`).

```
plant = setmpcsignals(plant,'MV',1,'MO',1,'UO',2);
```

Constrain the manipulated variable to be between +/- `220` volts. Since the plant inputs and outputs are of different orders of magnitude, to facilitate tuning, use scale factors. Typical choices of scale factor are the upper/lower limit or the operating range.

```
MV = struct('Min',-220,'Max',220,'ScaleFactor',440);
```

There is no constraint on the angular position. Specify upper and lower bounds on shaft torque during the first three prediction horizon steps. To define these bounds, use `tau`.

```
OV = struct('Min',{-Inf, [-tau;-tau;-Inf]},...
    'Max',{Inf, [tau;tau;Inf]},'ScaleFactor',{2*pi, 2*tau});
```

The control task is to achieve zero tracking error for the angular position. Since you only have one manipulated variable, allow shaft torque to float within its constraint by setting its tuning weight to zero.

```
Weights = struct('MV',0,'MVRate',0.1,'OV',[0.1 0]);
```

Specify the sample time and horizons, and create the MPC controller, using `plant` as the predictive model.

```
Ts = 0.1;           % Sample time
p = 10;             % Prediction horizon
m = 2;              % Control horizon
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

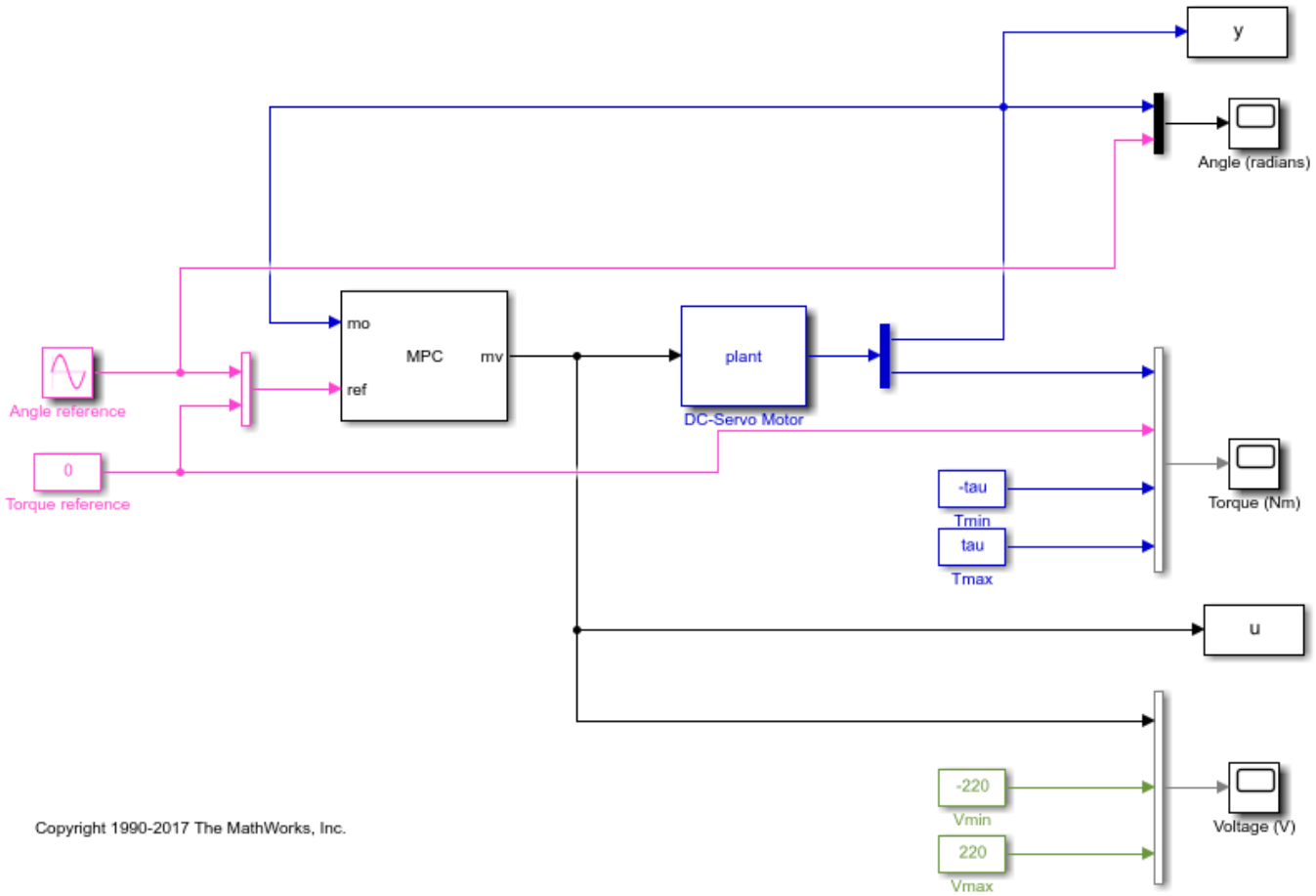
Simulate in Simulink with Built-In QP Solver

To run the remaining example, Simulink is required.

```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
```

Open a Simulink model that simulates closed-loop control of the dc-servo motor using the MPC controller. By default, MPC uses a built-in QP solver that uses the KWIK algorithm.

```
mdl = 'mpc_customQPcodegen';
open_system(mdl)
```



Run the simulation

```
sim(mdl)
```

```
-->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 measured
```

Store the plant input and output signals in the MATLAB workspace.

```
uKWIK = u;
yKWIK = y;
```

Simulate in Simulink with a Custom QP Solver

To examine how the custom solver behaves under the same conditions, enable the custom solver in the MPC controller.

```
mpcobj.Optimizer.CustomSolver = true;
```

You must also provide a MATLAB® function that satisfies the following requirements:

- Function name must be `mpcCustomSolver`.
- Input and output arguments must match the arguments in the template file.
- Function must be on the MATLAB path.

In this example, use the custom QP solver defined in the template file `mpcCustomSolverCodeGen_TemplateEML.txt`, which implements the `dantzig` algorithm and is suitable for code generation. Save the function in your working folder as `mpcCustomSolver.m`.

```
src = which('mpcCustomSolverCodeGen_TemplateEML.txt');
dest = fullfile(pwd,'mpcCustomSolver.m');
copyfile(src,dest,'f')
```

Simulate closed-loop control of the dc-servo motor, and save the plant input and output.

```
sim(mdl)
uDantzigSim = u;
yDantzigSim = y;
```

```
-->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 measured
```

Generate Code with Custom QP Solver

To run the remaining example, Simulink Coder product is required.

```
if ~mpcchecktoolboxinstalled('simulinkcoder')
    disp('Simulink(R) Coder(TM) is required to run this example.')
    return
end
```

To generate code from an MPC Controller block that uses a custom QP solver, enable the custom solver for code generation option in the MPC controller.

```
mpcobj.Optimizer.CustomSolverCodeGen = true;
```

You must also provide a MATLAB® function that satisfies all the following requirements:

- Function name must be `mpcCustomSolverCodeGen`.
- Input and output arguments must match the arguments in the template file.
- Function must be on the MATLAB path.

In this example, use the same custom solver defined in `mpcCustomSolverCodeGen_TemplateEML.txt`. Save the function in your working folder as `mpcCustomSolverCodeGen.m`.

```
src = which('mpcCustomSolverCodeGen_TemplateEML.txt');
dest = fullfile(pwd,'mpcCustomSolverCodeGen.m');
copyfile(src,dest,'f')
```

Review the saved `mpcCustomSolverCodeGen.m` file.

```
function [x, status] = mpcCustomSolverCodeGen(H, f, A, b, x0)
```

```
%#codegen
% mpcCustomSolverCodeGen allows the user to specify a custom (QP) solver
% written in MATLAB to be used by MPC controller in code generation.
%
% Workflow:
%   (1) Copy this template file to your work folder and rename it to
%       "mpcCustomSolverCodeGen.m".  The work folder must be on the path.
%   (2) Modify the "mpcCustomSolverCodeGen.m" to use your solver.
%       Note that your solver must use only fixed-size data.
%   (3) Set "mpcobj.Optimizer.CustomSolverCodeGen = true" to tell the MPC
%       controller to use the solver in code generation.
% To generate code:
%   In MATLAB, use "codegen" command with "mpcmoveCodeGeneration" (require MATLAB Coder)
%   In Simulink, generate code with MPC and Adaptive MPC blocks
%
% To use this solver for simulation in MATLAB and Simulink, you need to:
%   (1) Copy "mpcCustomSolver.txt" template file to your work folder and
%       rename it to "mpcCustomSolver.m".  The work folder must be on the path.
%   (2) Modify the "mpcCustomSolver.m" to use your solver.
%   (3) Set "mpcobj.Optimizer.CustomSolver = true" to tell the MPC
%       controller to use the solver in simulation.
%
% The MPC QP problem is defined as follows:
%
%       min J(x) = 0.5*x'*H*x + f'*x, s.t. 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 (sent back to MPC controller at run-time):
%       x: must be a n-by-1 vector of optimal solution.
%  status: must be an 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:
%   (1) When solver fails to find an optimal solution (status<=0), "x"
%       still needs to be returned.
%   (2) To use sub-optimal QP solution in MPC, return the sub-optimal "x"
%       with "status = 0".  In addition, you need to set
%       "mpcobj.Optimizer.UseSuboptimalSolution = true" in MPC controller.
%
% DO NOT CHANGE LINES ABOVE

% This template implements a showcase QP solver using "Dantzig" algorithm
% by G. B. Dantzig, A. Orden, and P. Wolfe, "The generalized simplex method
% for minimizing a linear form under linear inequality constraints",
% Pacific J. of Mathematics, 5:183–195, 1955.
%
% User is expected to modify this template and plug in own custom QP solver
% that replaces the "Dantzig" algorithm.

ZERO = zeros('like',H);
```

```
ONE = ones('like',H);
% xmin is a constant term that adds to the initial basis because "dantzig"
% requires positive optimization variables.  A fixed "xmin" does not work
% for all MPC problems.
xmin = -1e3*ones(size(f(:)))*ONE;

maxiter = 200*ONE;
nvar = length(f);
ncon = length(b);
a = -H*xmin(:);
H = H\eye(nvar);
rhsc = A*xmin(:) - b(:);
rhsa = a-f(:);
TAB = -[H H*A';A*H A*H*A'];
basisi = [H*rhsa; rhsc + A*H*rhsa];
ibi = -(1:nvar+ncon)'*ONE;
ili = -ibi*ONE;
%% Call EML function "qpdantzg"
[basis,ib,il,iter] = qpdantzg(TAB,basisi,ibi,ili,maxiter); %#ok<ASGLU>
%% status
if iter > maxiter
    status = ZERO;
elseif iter < ZERO
    status = -ONE;
else
    status = iter;
end
%% optimal variable
x = zeros(nvar,1,'like',H);
for j = 1:nvar
    if il(j) <= ZERO
        x(j) = xmin(j);
    else
        x(j) = basis(il(j))+xmin(j);
    end
end
```

Generate executable code from the Simulink model using the `slbuild` command from Simulink Coder.

```
slbuild(mdl)
```

```
### Starting build procedure for: mpc_customQPcodegen
-->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 measured
### Successful completion of build procedure for: mpc_customQPcodegen

Build Summary

Top model targets built:

Model                Action                       Rebuild Reason
=============================================================================================
mpc_customQPcodegen  Code generated and compiled  Code generation information file does not exist
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 58.326s
```

On a Windows system, after the build process finishes, the software adds the executable file `mpc_customQPcodegen.exe` to your working folder.

Run the executable. After the executable completes successfully (`status = 0`), the software adds the data file `mpc_customQPcodegen.mat` to your working folder. Load the data file into the MATLAB workspace, and obtain the plant input and output signals generated by the executable.

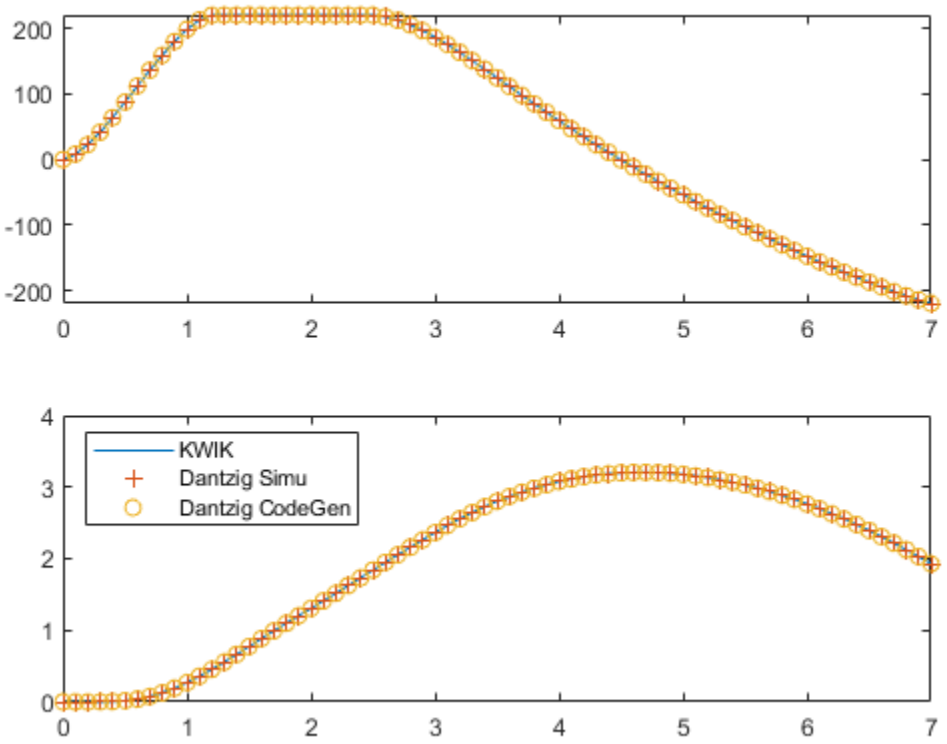
```
if ispc
    status = system(mdl);
    load(mdl)
    uDantzigCodeGen = u;
    yDantzigCodeGen = y;
else
    disp('The example only runs the executable on Windows system.');
end
```

```
** starting the model **
** created mpc_customQPcodegen.mat **
```

Compare Simulation Results

Compare the plant input and output signals from all the simulations.

```
if ispc
    figure
    subplot(2,1,1)
    plot(u.time,uKWIK.signals.values,u.time,uDantzigSim.signals.values,...
        '+',u.time,uDantzigCodeGen.signals.values,'o')
    subplot(2,1,2)
    plot(y.time,yKWIK.signals.values,y.time,yDantzigSim.signals.values,...
        '+',y.time,yDantzigCodeGen.signals.values,'o')
    legend('KWIK','Dantzig Simu','Dantzig CodeGen','Location','northwest')
else
    figure
    subplot(2,1,1)
    plot(u.time,uKWIK.signals.values,u.time,uDantzigSim.signals.values,'+')
    subplot(2,1,2)
    plot(y.time,yKWIK.signals.values,y.time,yDantzigSim.signals.values,'+')
    legend('KWIK','Dantzig Simu','Location','northwest')
end
```



The signals from all the simulations are identical.

References

[1] Bemporad, A. and Mosca, E. "Fulfilling hard constraints in uncertain linear systems by reference managing." *Automatica*, Vol. 34, Number 4, pp. 451-461, 1998.

```
bdclose(mdl)
```

See Also

Functions
`mpc`

Blocks
MPC Controller

More About

- “QP Solvers” on page 2-17

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.')
end
```

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
```

Configure 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, in this case, since a simulation OPC server is used, the plant behavior must be simulated. To do so, you define tags for the plant manipulated and measured variables and create a callback function (`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 MPC Controller to Control 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 steps, and prediction horizon of 20 steps.

```
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 during each callback execution.

```
state = mpcstate(mpcobj);
y1 = mpcmove(mpcobj,state,[1;1]',[1 1]');
```

Build OPC I/O for 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 Simulator and 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 change.

```
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
```

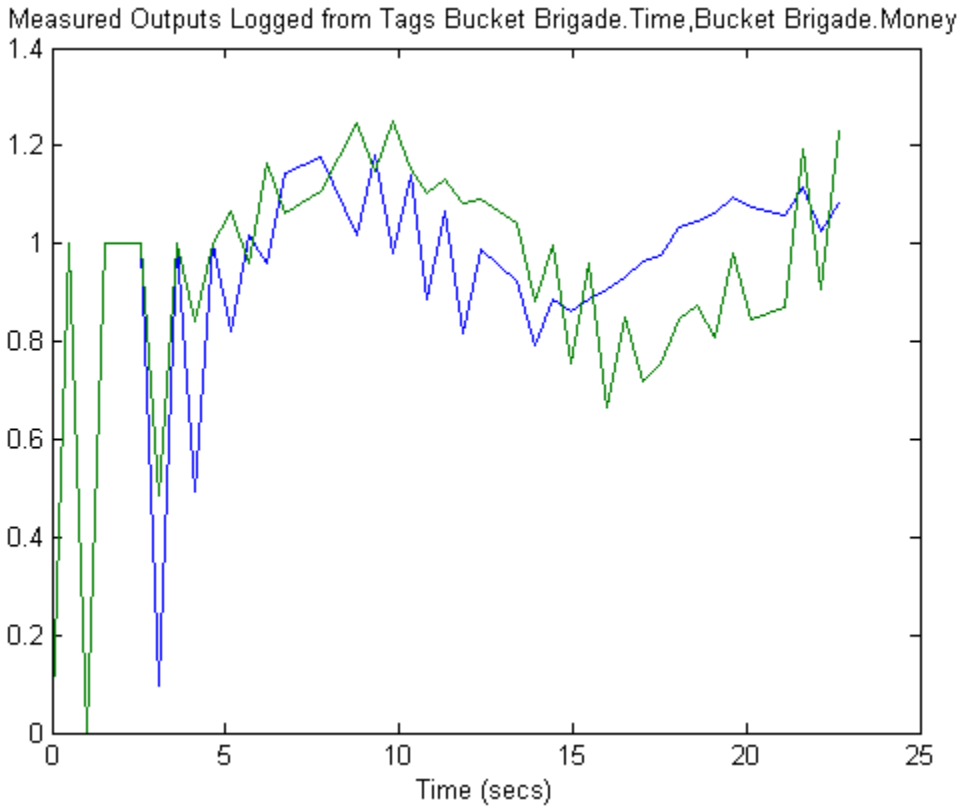
Log Data from Plant Measured Outputs

Log the plant measured outputs from tags `'Bucket Brigade.Time'` and `'Bucket Brigade.Money'`.

```
if mpccheckboxinstalled('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.RecordsToAcquire)
    end
    stop(mpc_read);
end
```

Extract and Plot Logged Data

```
if mpccheckboxinstalled('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
```


Measured Outputs Logged from Tags Bucket Brigade.Time,Bucket Brigade.Money
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
Time (secs)

See Also

Implement MPC Controllers using Embotech FORCESPRO Solvers

You can use FORCESPRO, a real-time embedded optimization software tool developed by Embotech AG, to simulate and generate code for linear and nonlinear MPC controllers designed using Model Predictive Control Toolbox software. Embotech provides a plugin that leverages the design capabilities of Model Predictive Control Toolbox software and the computational performance of FORCESPRO. Using the plugin, you can generate custom solvers that allow deployment on real-time hardware and that are highly optimized, based on your specific MPC problem, to achieve satisfactory real-time performance.

For more information on using the FORCESPRO MPC plugin, see the FORCESPRO Documentation. You can also use FORCESPRO solvers for other optimization applications in both MATLAB and Simulink. For more information, see the FORCESPRO Third Party Products and Services.

For information on generating code in MATLAB and Simulink for Model Predictive Control Toolbox controllers, see “Generate Code and Deploy Controller to Real-Time Targets” on page 10-2.

Embotech Quadratic Programming (QP) Solver

To design and simulate a linear time-invariant MPC controller (one in which the prediction model does not change at run time) in MATLAB using the Embotech FORCESPRO QP solver, follow these steps.

1. Design a linear controller using an `mpc` object.
2. Create a custom solver generation option object for the solver using `mpcToForcesOptions` with a string input argument that is either `"sparse"` (to build a sparse QP problem), or `"dense"` (to build a dense QP problem). Use `"sparse"` if your MPC problem has a long prediction horizon and a large number of constraints.
3. Generate the custom solver and the related variables containing structures for the core, state, and online data using `mpcToForces`.
4. If needed, adjust the controller state in the variables containing the state data structure, and specify run-time signals in the variable containing the online data structure.
5. Simulate the system by iteratively calling `mpcmoveFORCES`. For sparse QP problems, a MEX file is automatically generated and used to speed up the simulation.

You can also generate production code. For example, to generate a MEX file from `mpcmoveForces` with a dense QP formulation, where the variables `coredata`, `statedata`, and `onlinedata` were created by `mpcToForces`, you can use this code:

```
% configure code generation to create a MATLAB executable
cfg = coder.config('mex'); % or LIB, EXE, etc.
cfg.ConstantInputs = 'IgnoreValues';

% create an executable named myMex
codegen('-config',cfg,'mpcmoveForces','-o','myMex',...
        '-args',{coder.Constant(coredata), statedata, onlinedata});

% calculate the manipulated variables by calling the myMex executable
[mv, statedata, info] = myMex(coredata, statedata, onlinedata)
```

To design and simulate a linear time-invariant MPC controller in Simulink using the Embotech FORCESPRO QP solver solver, follow these steps.

1. Design a linear controller using an `mpc` object.
2. Create a custom solver generation option object for the solver using `mpcToForcesOptions` with a string input argument that is either `"sparse"` (to build a sparse QP problem), or `"dense"` (to build a dense QP problem). Use `"sparse"` if your MPC problem has a long prediction horizon and a large number of constraints.
3. Generate the custom solver and the related variables containing structures for the core, states, and online data using `mpcToForces`.
4. Add the appropriate block to your model:
   - For a sparse QP problem, open the Simulink library browser, find the FORCES MPC (Sparse QP) block under the FORCESPRO MPC Blocks category, and add it to your model.
   - For a dense QP problem, open the Simulink library browser, find the MPC Controller block under the Model Predictive Control Toolbox category, and add it to your model.
5. Specify structure variables in the block dialog:
   - For a sparse QP problem, specify the variables containing the core and states data structures.
   - For a dense QP problem, specify the `mpc` object.
6. Simulate the system.
7. When needed, generate code directly from the model or the block.

For more information on how to use QP solvers with Model Predictive Control Toolbox, see “QP Solvers” on page 2-17.

For more information on the FORCESPRO QP solver, see the Embotech FORCESPRO QP solver documentation.

Note Using the QP Embotech FORCESPRO Solver for Adaptive MPC controllers or MPC controllers with custom constraints is not supported.

Embotech Nonlinear Programming (NLP) Solver

To design and simulate a nonlinear MPC controller in MATLAB using the Embotech FORCESPRO NLP solver, follow these steps.

1. Design a nonlinear controller using an `nlmpc` or `nlmpcMultistage` object.
2. Specify custom solver generation options using `nlmpcToForcesOptions` (or `nlmpcMultistageToForcesOptions`, if you designed a multistage controller in the previous step). For an `nlmpc` object, you can choose to use the sequential quadratic programming (SQP) solver instead of the interior-point (IP) solver. Use the IP solver if your nonlinear MPC problem has long prediction horizon and a large number of constraints. For `nlmpcMultistage` objects only the IP solver is available.
3. Generate the custom solver and the related variables containing structures for the core, states, and online data using `nlmpcToForces`, (or `nlmpcMultistageToForces`).
4. Specify current controller states, last control action, and use the variable containing the online data structure to specify other run-time signals.
5. Simulate the system by iteratively calling `nlmpcmoveForces` (or `nlmpcmoveForcesMultistage`). A MEX file is automatically generated for the two functions to speed up the simulation in MATLAB.

To design and simulate a nonlinear MPC controller in Simulink using the Embotech FORCESPRO NLP solver, follow these steps.

1. Design a nonlinear controller using an `nlmpc` or `nlmpcMultistage` object.
2. Specify custom solver generation options using `nlmpcToForcesOptions` (or `nlmpcMultistageToForcesOptions`, if you designed a multistage controller in the previous step). For an `nlmpc` object, you can choose to use the sequential quadratic programming (SQP) solver instead of the interior-point (IP) solver. Use the IP solver if your nonlinear MPC problem has long prediction horizon and a large number of constraints. For `nlmpcMultistage` objects only the IP solver is available.
3. Generate the custom solver and the related variables containing structures for the core, states, and online data using `nlmpcToForces`, (or `nlmpcMultistageToForces`).
4. Open the Simulink library browser, find the FORCES Multistage Nonlinear MPC block under the FORCESPRO MPC Blocks category, and add it to your model.
5. Specify the variable containing the core data structure in the block dialog.
6. Simulate the system.
7. When needed, generate code directly from the model or the block.

You can also generate code for your Simulink model as described in the section “Code Generation in Simulink” on page 10-2.

Note Using the NLP Embotech FORCESPRO Solver is only supported when the state and output functions are compatible with MATLAB code generation and with CasADi. Additionally, for generic (that is non multistage) nonlinear MPC problems:

- You must not use the custom cost and constraint functions.
- If the nonlinear MPC controller uses multiple optional parameters, you must group them in a single column vector and set the `Model.NumberOfParameters` property of the controller to 1.

For an example on using the FORCES Nonlinear MPC block see “Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control” on page 9-49.

See Also

More About

- “QP Solvers” on page 2-17
- “Configure Optimization Solver for Nonlinear MPC” on page 9-27

Automated Driving Applications

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Adaptive Cruise Control System Using Model Predictive Control” on page 11-5
- “Adaptive Cruise Control with Sensor Fusion” on page 11-10
- “Lane Keeping Assist System Using Model Predictive Control” on page 11-28
- “Lane Keeping Assist with Lane Detection” on page 11-33
- “Lane Following Control with Sensor Fusion and Lane Detection” on page 11-51
- “Highway Lane Following” on page 11-62
- “Highway Lane Change” on page 11-78
- “Automate Testing for Highway Lane Following” on page 11-89
- “Highway Lane Following with Intelligent Vehicles” on page 11-99
- “Parking Valet Using Nonlinear Model Predictive Control” on page 11-117
- “Parallel Parking Using Nonlinear Model Predictive Control” on page 11-126
- “Parallel Parking Using RRT Planner and MPC Tracking Controller” on page 11-139
- “Traffic Light Negotiation” on page 11-148
- “Traffic Light Negotiation with Unreal Engine Visualization” on page 11-162

Automated Driving Using Model Predictive Control

Model predictive control (MPC) is a discrete-time multi-variable control architecture. At each control interval, an MPC controller uses an internal model to predict future plant behavior. Based on this prediction, the controller computes optimal control actions. For more information on model predictive control, see “MPC Design”.

You can use MPC in automated driving applications to improve vehicle responsiveness while maintaining passenger comfort and safety. Applications can include:

- Adaptive cruise control — For an example, see “Adaptive Cruise Control System Using Model Predictive Control” on page 11-5.
- Lane-keeping assist — For an example, see “Lane Keeping Assist System Using Model Predictive Control” on page 11-28.
- Lane-following control — For an example, see “Lane Following Control with Sensor Fusion and Lane Detection” on page 11-51.
- Parking — For an example, see “Parallel Parking Using Nonlinear Model Predictive Control” on page 11-126.
- Obstacle avoidance — For an example, see “Obstacle Avoidance Using Adaptive Model Predictive Control” on page 7-38.

MPC has several features that are useful for automated driving.

MPC Feature	Description	More Information
Explicitly handle input and output constraints	When computing optimal control moves, an MPC controller accounts for any input and output constraints on the system. For example, you can specify constraints for: • Speed limits • Safe following distance • Physical vehicle limits, such as maximum steering angle • Obstacles for the controller to avoid	• “Specify Constraints” on page 1-5 • “Constraints on Linear Combinations of Inputs and Outputs” on page 3-5
Predict ego vehicle behavior across a receding horizon	An MPC controller uses an internal model of the vehicle dynamics to predict how the vehicle will react to a given control action across a prediction horizon. This behavior is analogous to a human driver understanding and predicting the behavior of their vehicle.	• “Choose Sample Time and Horizons” on page 1-2 • “MPC Prediction Models”
Preview reference trajectories and disturbances across prediction horizon	If you can anticipate reference trajectories or disturbances across the prediction horizon, an MPC controller can incorporate this information when computing optimal control actions. This behavior is analogous to a human driver previewing the road ahead of their vehicle.	“Signal Previewing” on page 5-19

MPC Feature	Description	More Information
Update internal vehicle model at run time	If the dynamics of the ego vehicle vary over time, such as for velocity-dependent steering dynamics, you can update the controller internal model using adaptive MPC.	“Adaptive MPC” on page 7-2
Generate code	You can automatically generate code for deploying model predictive controllers.	“Generate Code and Deploy Controller to Real-Time Targets” on page 10-2

Simulation in Simulink

To simplify the initial development of automated driving controllers, Model Predictive Control Toolbox software provides Simulink blocks for adaptive cruise control, lane-keeping assistance, and path following. These blocks provide application-specific interfaces and options for designing an MPC controller.

Block	Description
Adaptive Cruise Control System	Track a set velocity and maintain a safe distance from a lead vehicle by adjusting the longitudinal acceleration of an ego vehicle.
Lane Keeping Assist System	Keep an ego vehicle traveling along the center of a straight or curved road by adjusting the front steering angle.
Path Following Control System	Keep an ego vehicle traveling along the center of a straight or curved road while tracking a set velocity and maintaining a safe distance from a lead vehicle. To do so, the controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle.

For other automated driving applications, such as obstacle avoidance, you can design and simulate controllers using the other model predictive control Simulink blocks, such as the MPC Controller, Adaptive MPC Controller, and Nonlinear MPC Controller blocks. For an example that uses an adaptive model predictive controller, see “Obstacle Avoidance Using Adaptive Model Predictive Control” on page 7-38.

Controller Customization

For the Adaptive Cruise Control System, Lane Keeping Assist System, and Path Following Control System blocks, you can generate a custom subsystem, which you can then modify for your application. This option is useful when you want to:

- Modify default MPC settings or use advanced MPC features
- Modify the default controller initial conditions
- Use different application settings, such as a custom safe following distance definition for adaptive cruise control

To create a custom subsystem, click the corresponding button for the block you are using. For example, to create a custom subsystem for an Adaptive Cruise Control System block, on the **Block** tab, click **Create ACC subsystem**. The software creates a Simulink model that contains a subsystem

with the same configuration as your original controller. You can modify this subsystem and directly substitute it back into your original model, replacing the controller block.

Integration with Automated Driving Toolbox

If you have Automated Driving Toolbox™ software, you can integrate your model predictive controller with systems for:

- Object detection and tracking
- Lane boundary detection
- Path planning
- Sensor fusion

For examples, see: “Adaptive Cruise Control with Sensor Fusion” on page 11-10, “Lane Keeping Assist with Lane Detection” on page 11-33, and “Lane Following Control with Sensor Fusion and Lane Detection” on page 11-51.

See Also

Blocks
Adaptive Cruise Control System | Lane Keeping Assist System | MPC Controller | Adaptive MPC Controller | Path Following Control System | Nonlinear MPC Controller

Adaptive Cruise Control System Using Model Predictive Control

This example shows how to use the Adaptive Cruise Control System block in Simulink® and demonstrates the control objectives and constraints of this block.

Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc','main'));
```

Adaptive Cruise Control System

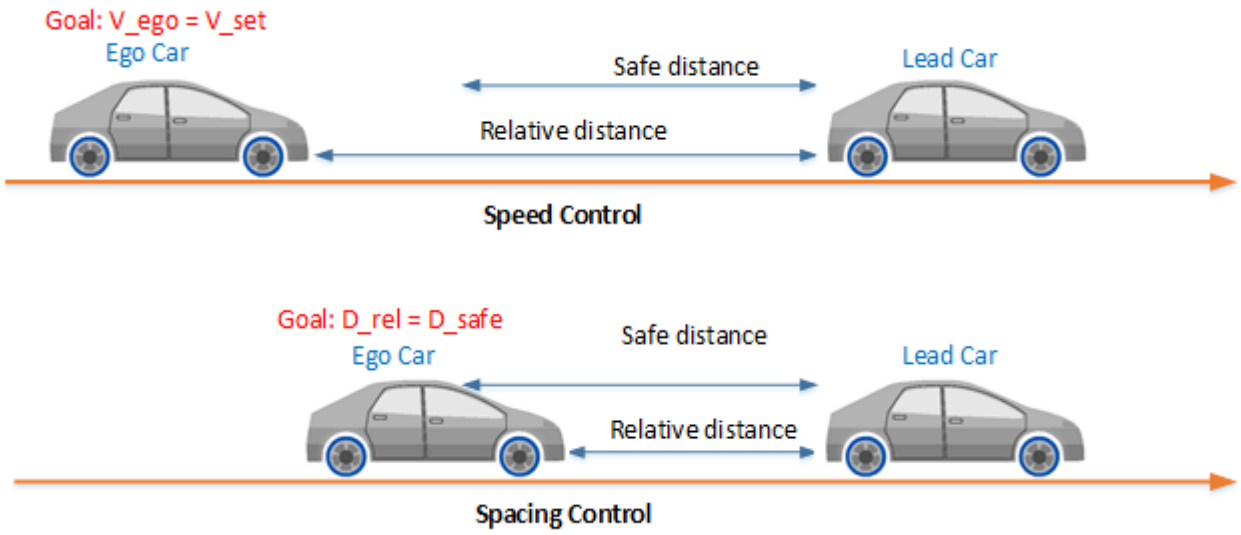
A vehicle (ego car) equipped with adaptive cruise control (ACC) has a sensor, such as radar, that measures the distance to the preceding vehicle in the same lane (lead car), D_{rel}. The sensor also measures the relative velocity of the lead car, V_{rel}. The ACC system operates in the following two modes:

- Speed control: The ego car travels at a driver-set speed.
- Spacing control: The ego car maintains a safe distance from the lead car.

The ACC system decides which mode to use based on real-time radar measurements. For example, if the lead car is too close, the ACC system switches from speed control to spacing control. Similarly, if the lead car is further away, the ACC system switches from spacing control to speed control. In other words, the ACC system makes the ego car travel at a driver-set speed as long as it maintains a safe distance.

The following rules are used to determine the ACC system operating mode:

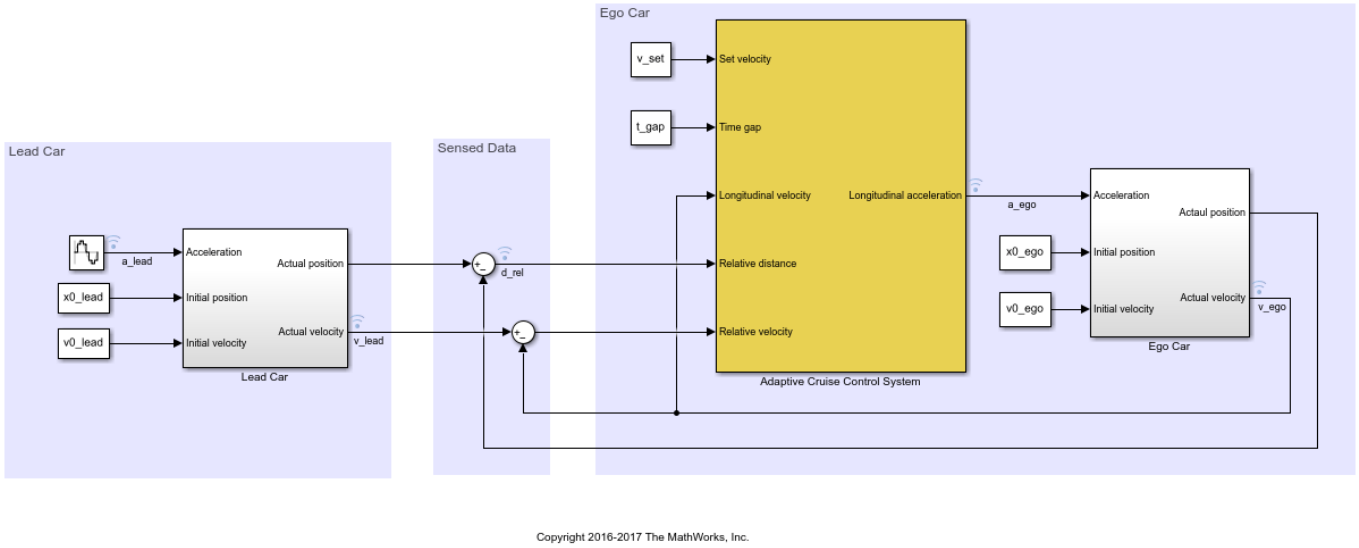
- If $D_{rel} \geq D_{safe}$, then speed control mode is active. The control goal is to track the driver-set velocity, V_{set}.
- If $D_{rel} < D_{safe}$, then spacing control mode is active. The control goal is to maintain the safe distance, D_{safe}.



Simulink Model for Lead Car and Ego Car

The dynamics for lead car and ego car are modeled in Simulink. Open the Simulink model.

```
mdl = 'mpcACCsystem';
open_system(mdl)
```



Copyright 2016-2017 The MathWorks, Inc.

To approximate a realistic driving environment, the acceleration of the lead car varies according to a sine wave during the simulation. The Adaptive Cruise Control System block outputs an acceleration control signal for the ego car.

Define the sample time, `Ts`, and simulation duration, `T`, in seconds.

```
Ts = 0.1;
T = 80;
```

For both the ego vehicle and the lead vehicle, the dynamics between acceleration and velocity are modeled as:

$$G = \frac{1}{s(0.5s + 1)}$$

which approximates the dynamics of the throttle body and vehicle inertia.

Specify the linear model for ego car.

```
G_ego = tf(1,[0.5,1,0]);
```

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 25;   % initial velocity for lead car (m/s)

x0_ego = 10;   % initial position for ego car (m)
v0_ego = 20;   % initial velocity for ego car (m/s)
```

Configuration of Adaptive Cruise Control System Block

The ACC system is modeled using the Adaptive Cruise Control System Block in Simulink. The inputs to the ACC system block are:

- Driver-set velocity V_{set}
- Time gap T_{gap}
- Velocity of the ego car V_{ego}
- Relative distance to the lead car D_{rel} (from radar)
- Relative velocity to the lead car V_{rel} (from radar)

The output for the ACC system is the acceleration of the ego car.

The safe distance between the lead car and the ego car is a function of the ego car velocity, V_{ego}:

$$D_{safe} = D_{default} + T_{gap} \times V_{ego}$$

where $D_{default}$ is the standstill default spacing and T_{gap} is the time gap between the vehicles. Specify values for $D_{default}$, in meters, and T_{gap}, in seconds.

```
t_gap = 1.4;
D_default = 10;
```

Specify the driver-set velocity in m/s.

```
v_set = 30;
```

Considering the physical limitations of the vehicle dynamics, the acceleration is constrained to the range `[-3,2]` (m/s^2).

```
amin_ego = -3;
amax_ego = 2;
```

For this example, the default parameters of the Adaptive Cruise Control System block match the simulation parameters. If your simulation parameters differ from the default values, then update the block parameters accordingly.

Simulation Analysis

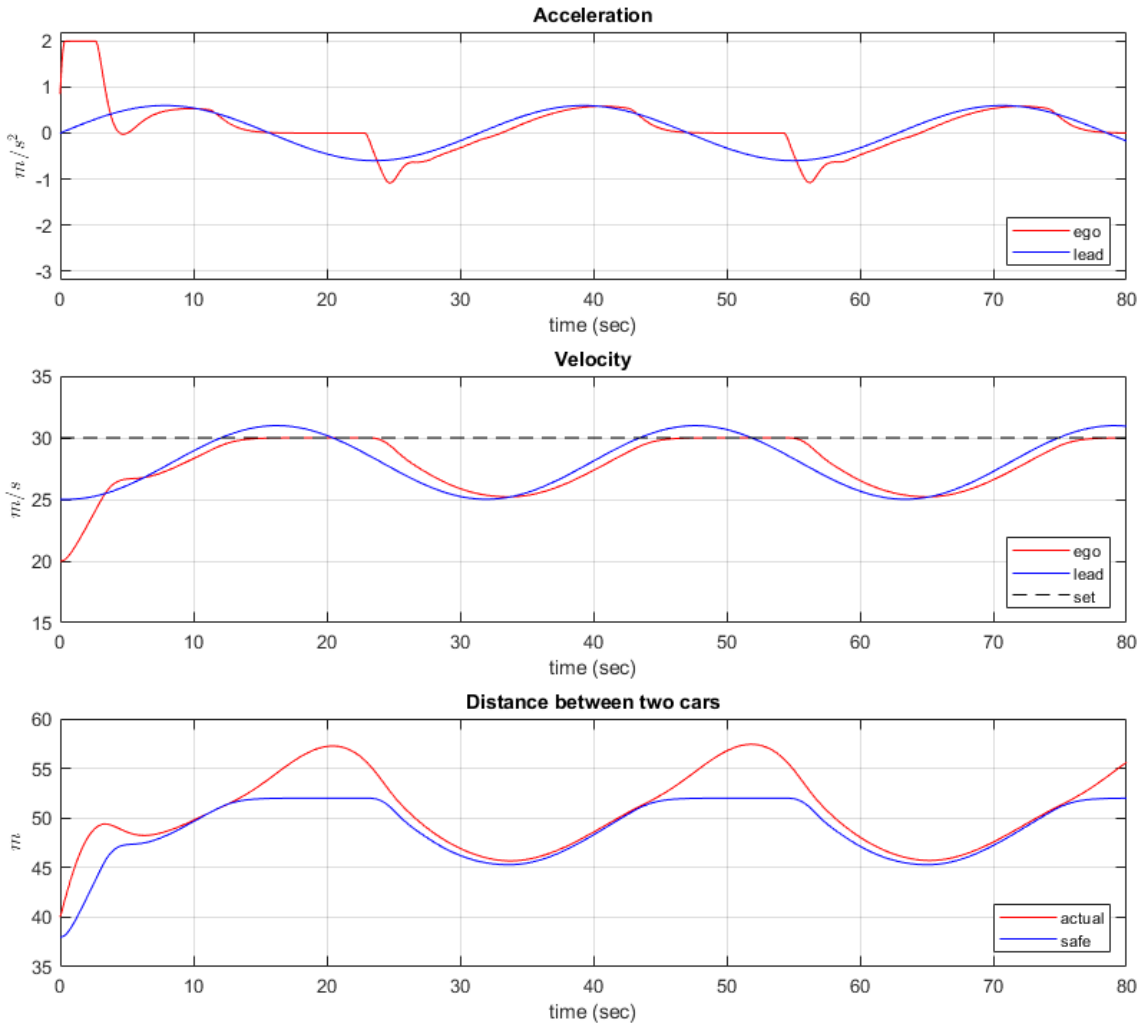
Run the simulation.

```
sim(mdl)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #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 measured
```

Plot the simulation result.

```
mpcACCplot(logsout,D_default,t_gap,v_set)
```



In the first 3 seconds, to reach the driver-set velocity, the ego car accelerates at full throttle.

From 3 to 13 seconds, the lead car accelerates slowly. As a result, to maintain a safe distance to the lead car, the ego car accelerates with a slower rate.

From 13 to 25 seconds, the ego car maintains the driver-set velocity, as shown in the **Velocity** plot. However, as the lead car reduces speed, the spacing error starts approaching 0 after 20 seconds.

From 25 to 45 seconds, the lead car slows down and then accelerates again. The ego car maintains a safe distance from the lead car by adjusting its speed, as shown in the **Distance** plots.

From 45 to 56 seconds, the spacing error is above `0`. Therefore, the ego car achieves the driver-set velocity again.

From 56 to 76 seconds, the deceleration/acceleration sequence from the 25 to 45 second interval is repeated.

Throughout the simulation, the controller ensures that the actual distance between the two vehicles is greater than the set safe distance. When the actual distance is sufficiently large, then the controller ensures that the ego vehicle follows the driver-set velocity.

Remove example file folder from MATLAB path, and close Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc','main'));
bdclose(mdl)
```

See Also

Blocks
Adaptive Cruise Control System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

Adaptive Cruise Control with Sensor Fusion

This example shows how to implement a sensor fusion-based automotive adaptive cruise controller for a vehicle traveling on a curved road using sensor fusion.

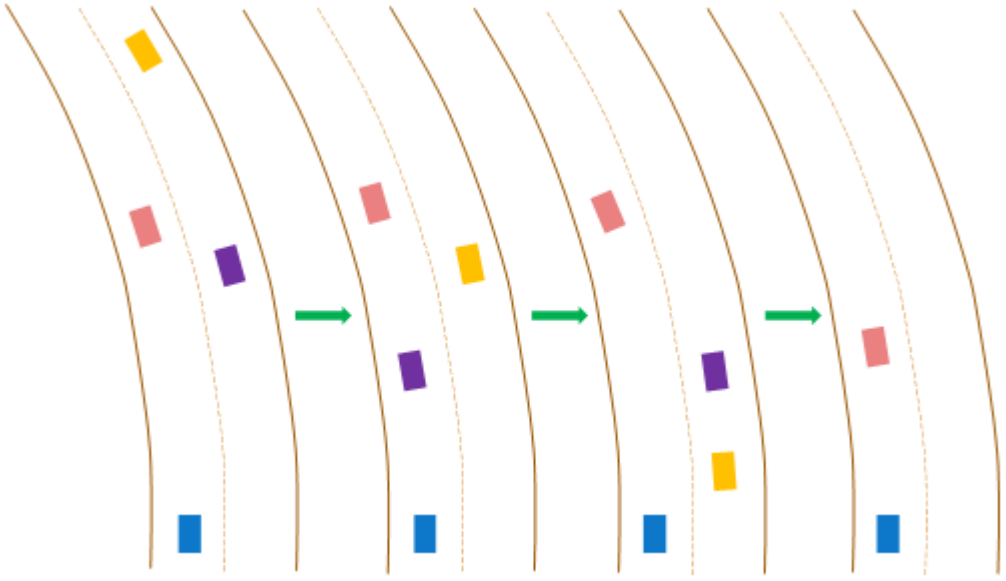
In this example, you:

1. Review a control system that combines sensor fusion and an adaptive cruise controller (ACC). Two variants of ACC are provided: a classical controller and an Adaptive Cruise Control System block from Model Predictive Control Toolbox.
2. Test the control system in a closed-loop Simulink model using synthetic data generated by the Automated Driving Toolbox.
3. Configure the code generation settings for software-in-the-loop simulation, and automatically generate code for the control algorithm.

Introduction

An adaptive cruise control system is a control system that modifies the speed of the ego vehicle in response to conditions on the road. As in regular cruise control, the driver sets a desired speed for the car; in addition, the adaptive cruise control system can slow the ego vehicle down if there is another vehicle moving slower in the lane in front of it.

For the ACC to work correctly, the ego vehicle must determine how the lane in front of it curves, and which car is the 'lead car', that is, in front of the ego vehicle in the lane. A typical scenario from the viewpoint of the ego vehicle is shown in the following figure. The ego vehicle (blue) travels along a curved road. At the beginning, the lead car is the pink car. Then the purple car cuts into the lane of the ego vehicle and becomes the lead car. After a while, the purple car changes to another lane, and the pink car becomes the lead car again. The pink car remains the lead car afterward. The ACC design must react to the change in the lead car on the road.



Current ACC designs rely mostly on range and range rate measurements obtained from radar, and are designed to work best along straight roads. An example of such a system is given in “Adaptive Cruise Control System Using Model Predictive Control” on page 11-5 and in “Automotive Adaptive

Cruise Control Using FMCW Technology” (Radar Toolbox). Moving from advanced driver-assistance system (ADAS) designs to more autonomous systems, the ACC must address the following challenges:

1. Estimating the relative positions and velocities of the cars that are near the ego vehicle and that have significant lateral motion relative to the ego vehicle.
2. Estimating the lane ahead of the ego vehicle to find which car in front of the ego vehicle is the closest one in the same lane.
3. Reacting to aggressive maneuvers by other vehicles in the environment, in particular, when another vehicle cuts into the ego vehicle lane.

This example demonstrates two main additions to existing ACC designs that meet these challenges: adding a sensor fusion system and updating the controller design based on model predictive control (MPC). A sensor fusion and tracking system that uses both vision and radar sensors provides the following benefits:

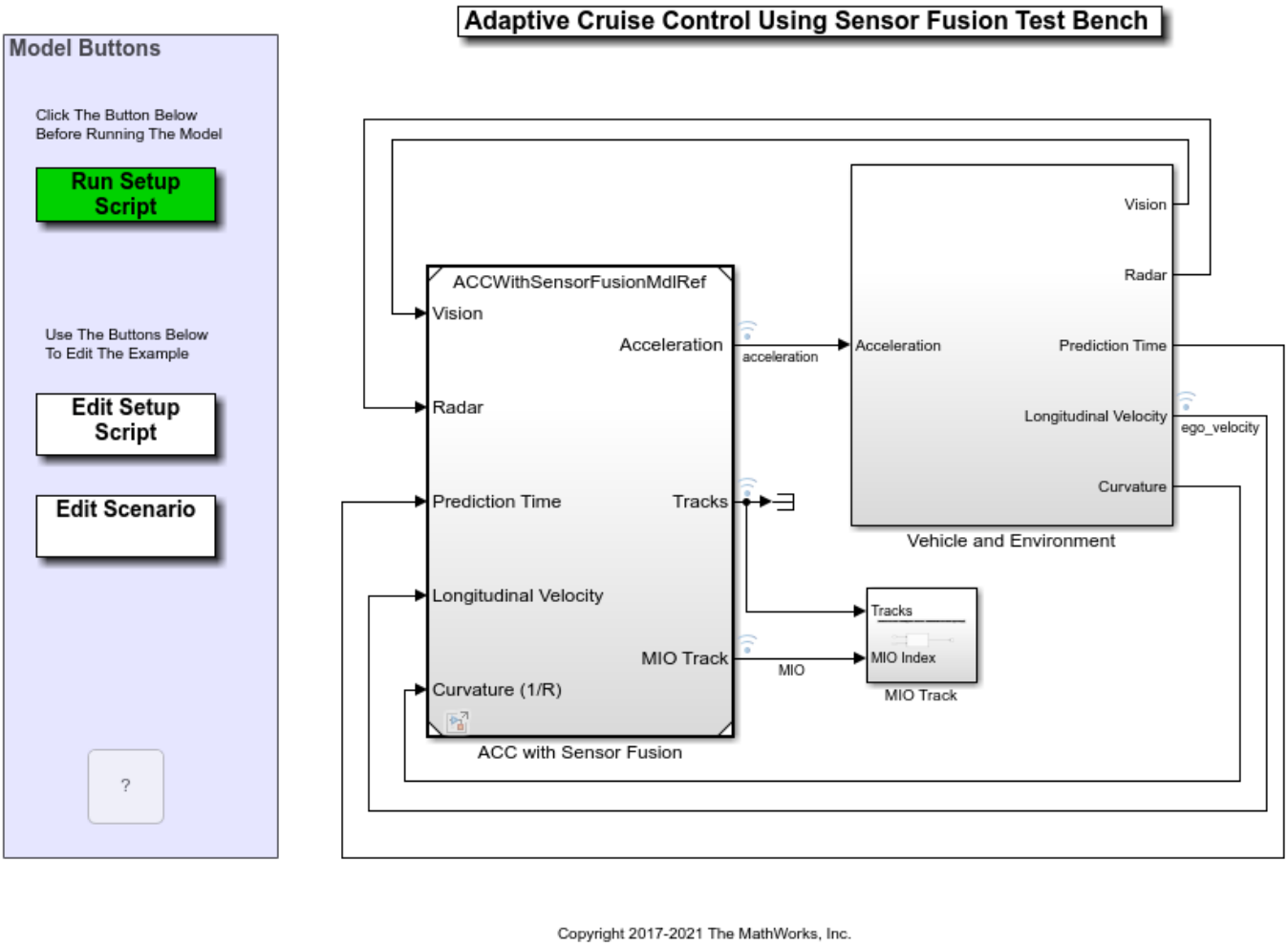
1. It combines the better lateral measurement of position and velocity obtained from vision sensors with the range and range rate measurement from radar sensors.
2. A vision sensor can detect lanes, provide an estimate of the lateral position of the lane relative to the ego vehicle, and position the other cars in the scene relative to the ego vehicle lane. This example assumes ideal lane detection.

An advanced MPC controller adds the ability to react to more aggressive maneuvers by other vehicles in the environment. In contrast to a classical controller that uses a PID design with constant gains, the MPC controller regulates the velocity of the ego vehicle while maintaining a strict safe distance constraint. Therefore, the controller can apply more aggressive maneuvers when the environment changes quickly in a similar way to what a human driver would do.

Overview of Test Bench Model and Simulation Results

To open the main Simulink model, use the following command:

```
open_system('ACCTestBenchExample')
```



The model contains two main subsystems:

1. ACC with Sensor Fusion, which models the sensor fusion and controls the longitudinal acceleration of the vehicle. This component allows you to select either a classical or model predictive control version of the design.
2. A Vehicle and Environment subsystem, which models the motion of the ego vehicle and models the environment. A simulation of radar and vision sensors provides synthetic data to the control subsystem.

To run the associated initialization script before running the model, in the Simulink model, click **Run Setup Script** or, at the command prompt, type the following:

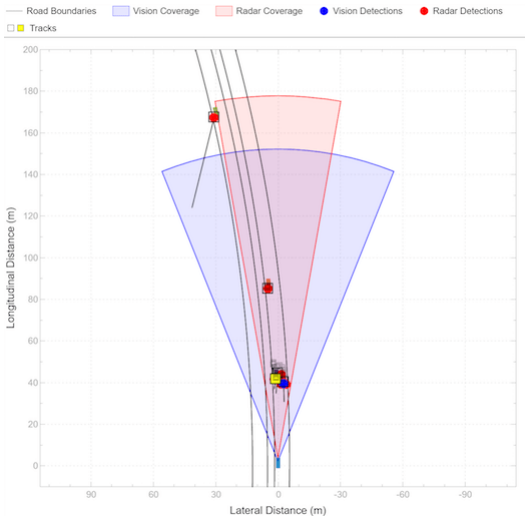
```
helperACCSetUp
```

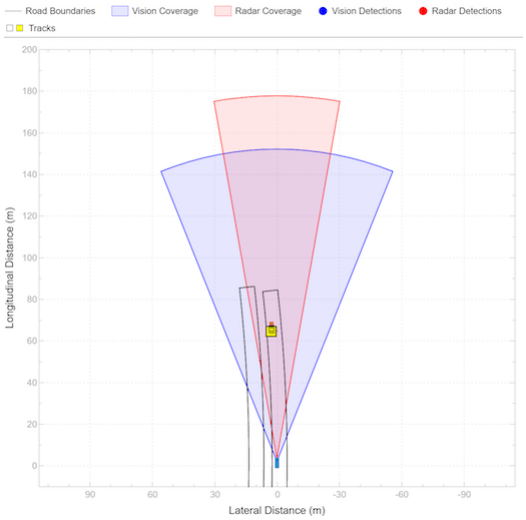
The script loads certain constants needed by the Simulink model, such as the scenario object, vehicle parameters, and ACC design parameters. The default ACC is the classical controller. The script also creates buses that are required for defining the inputs into and outputs for the control system referenced model. These buses must be defined in the workspace before model compilation. When the model compiles, additional Simulink buses are automatically generated by their respective blocks.

To plot the results of the simulation and depict the surroundings of the ego vehicle, including the tracked objects, use the Bird's-Eye Scope (Automated Driving Toolbox). The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, click **Find Signals** to set up the signals. The following commands run the simulation to 15 seconds to get a mid-simulation picture and run again all the way to end of the simulation to gather results.

```
sim('ACCTestBenchExample','StopTime','15') %Simulate 15 seconds
sim('ACCTestBenchExample') %Simulate to end of scenario
```

```
ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [151x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```





The Bird's-Eye Scope shows the results of the sensor fusion. It shows how the radar and vision sensors detect the vehicles within their sensors coverage areas. It also shows the tracks maintained by the Multi Object Tracker block. The yellow track shows the most important object (MIO): the closest track in front of the ego vehicle in its lane. We see that at the beginning of the scenario, the most important object is the fast-moving car ahead of the ego vehicle. When the passing car gets closer to the slow-moving car, it crosses to the left lane, and the sensor fusion system recognizes it to be the MIO. This car is much closer to the ego vehicle and much slower than it. Thus, the ACC must slow the ego vehicle.

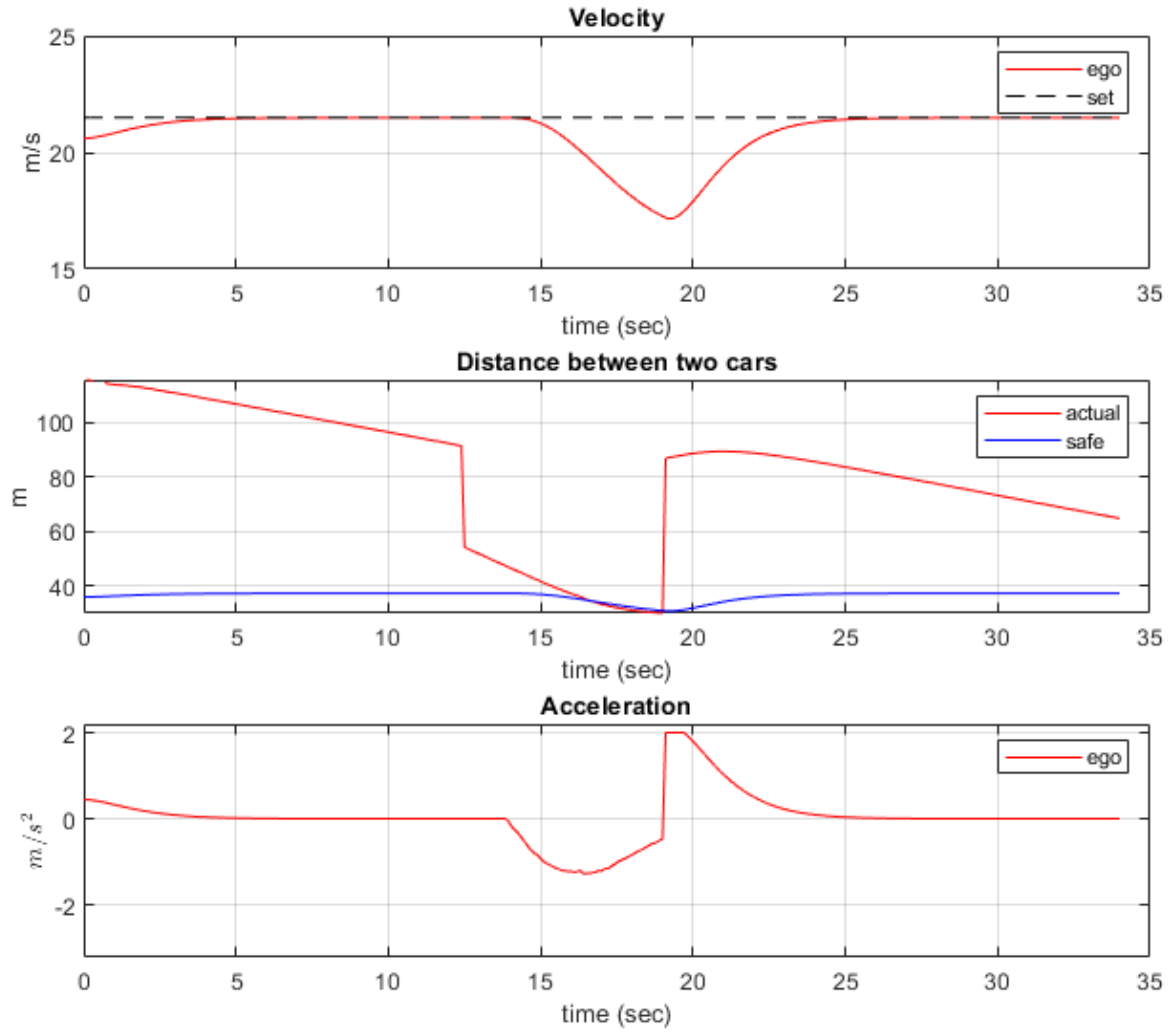
In the following results for the classical ACC system, the:

- Top plot shows the ego vehicle velocity.
- Middle plot shows the relative distance between the ego vehicle and lead car.
- Bottom plot shows the ego vehicle acceleration.

In this example, the raw data from the Tracking and Sensor Fusion system is used for ACC design without post-processing. You can expect to see some 'spikes' (middle plot) due to the uncertainties in the sensor model especially when another car cuts into or leaves the ego vehicle lane.

To view the simulation results, use the following command.

```
helperPlotACCResults(logsout,default_spacing,time_gap)
```

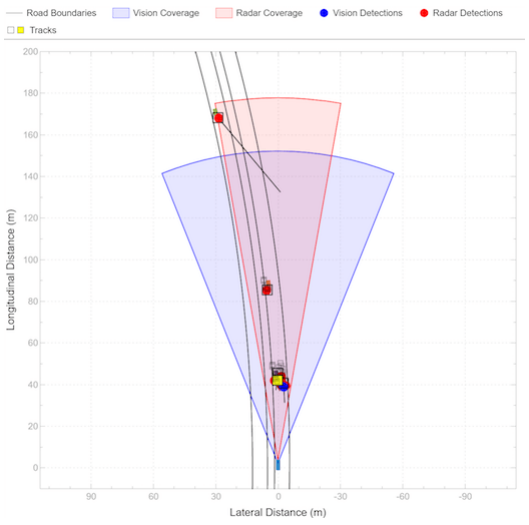


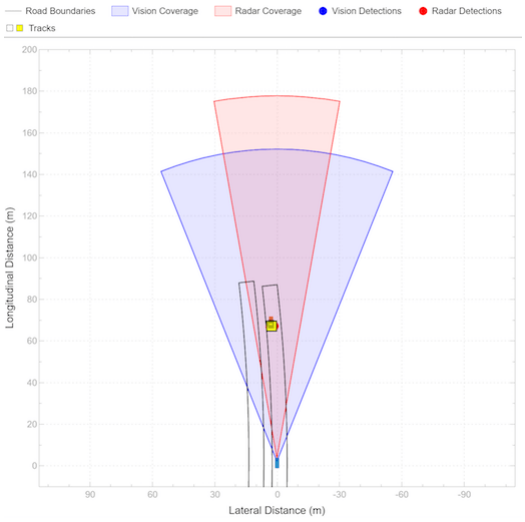
- In the first 11 seconds, the lead car is far ahead of the ego vehicle (middle plot). The ego vehicle accelerates and reaches the velocity set by the driver (top plot).
- Another car becomes the lead car from 11 to 20 seconds when the car cuts into the ego vehicle lane (middle plot). When the distance between the lead car and the ego vehicle is large (11-15 seconds), the ego vehicle still travels at the driver-set velocity. When the distance becomes small (15-20 seconds), the ego vehicle decelerates to maintain a safe distance from the lead car (top plot).
- From 20 to 34 seconds, the car in front moves to another lane, and a new lead car appears (middle plot). Because the distance between the lead car and the ego vehicle is large, the ego vehicle accelerates until it reaches the driver-set velocity at 27 seconds. Then, the ego vehicle continues to travel at the driver-set velocity (top plot).
- The bottom plot demonstrates that the acceleration is within the range [-3,2] m/s^2. The smooth transient behavior indicates that the driver comfort is satisfactory.

In the MPC-based ACC design, the underlying optimization problem is formulated by tracking the driver-set velocity subject to enforcing a safe distance from the lead car. The MPC controller design is described in the Adaptive Cruise Controller section. To run the model with the MPC design, first activate the MPC variant, and then run the following commands. This step requires Model Predictive Control Toolbox software. You can check the existence of this license using the following code. If no code exists, a sample of similar results is depicted.

```
hasMPCLicense = license('checkout','MPC_Toolbox');
if hasMPCLicense
    controller_type = 2;
    sim('ACCTestBenchExample','StopTime','15') %Simulate 15 seconds
    sim('ACCTestBenchExample') %Simulate to end of scenario
else
    load data_mpc
end
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #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 measured

ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [151x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #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 measured
```

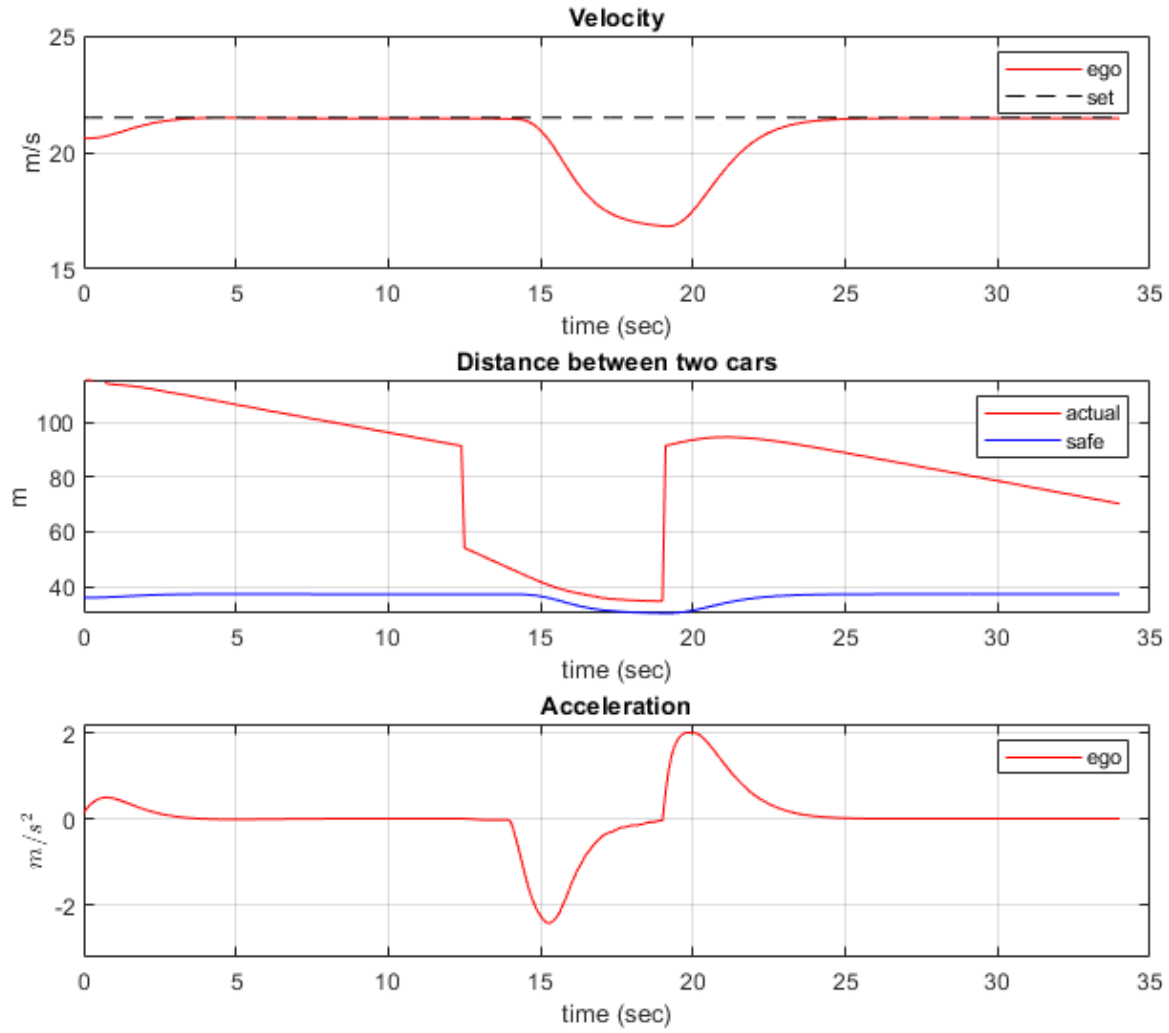




In the simulation results for the MPC-based ACC, similar to the classical ACC design, the objectives of speed and spacing control are achieved. Compared to the classical ACC design, the MPC-based ACC is more aggressive as it uses full throttle or braking for acceleration or deceleration. This behavior is due to the explicit constraint on the relative distance. The aggressive behavior may be preferred when sudden changes on the road occur, such as when the lead car changes to be a slow car. To make the controller less aggressive, open the mask of the Adaptive Cruise Control System block, and reduce the value of the **Controller Behavior** parameter. As previously noted, the spikes in the middle plot are due to the uncertainties in the sensor model.

To view the results of the simulation with the MPC-based ACC, use the following command.

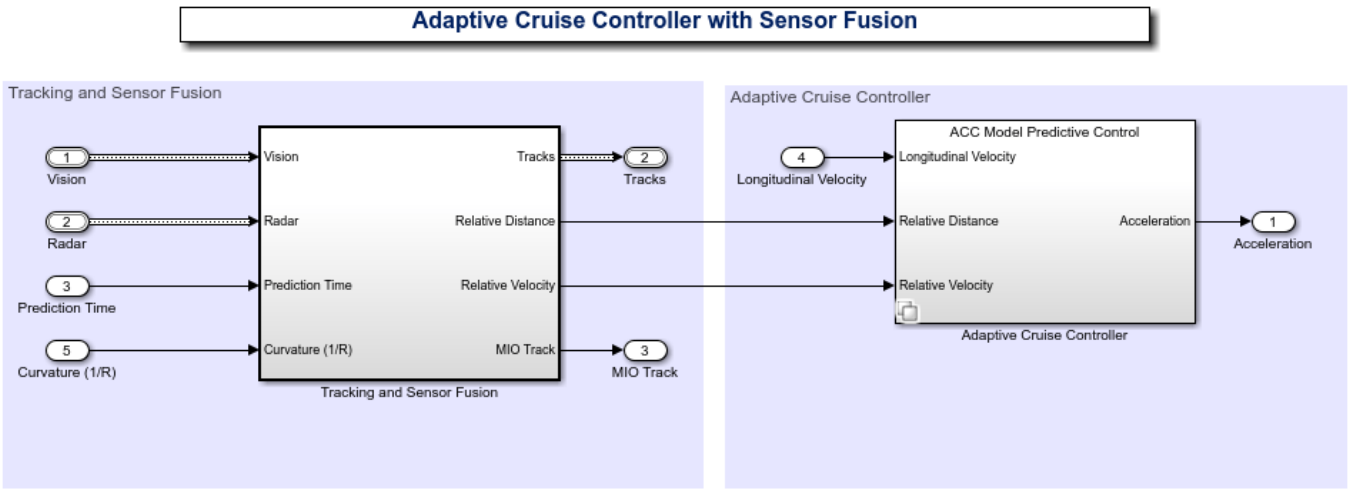
```
helperPlotACCResults(logsout,default_spacing,time_gap)
```



In the following, the functions of each subsystem in the Test Bench Model are described in more detail. The Adaptive Cruise Controller with Sensor Fusion subsystem contains two main components:

1. Tracking and Sensor Fusion subsystem
2. Adaptive Cruise Controller subsystem

```
open_system('ACCTestBenchExample/ACC with Sensor Fusion')
```

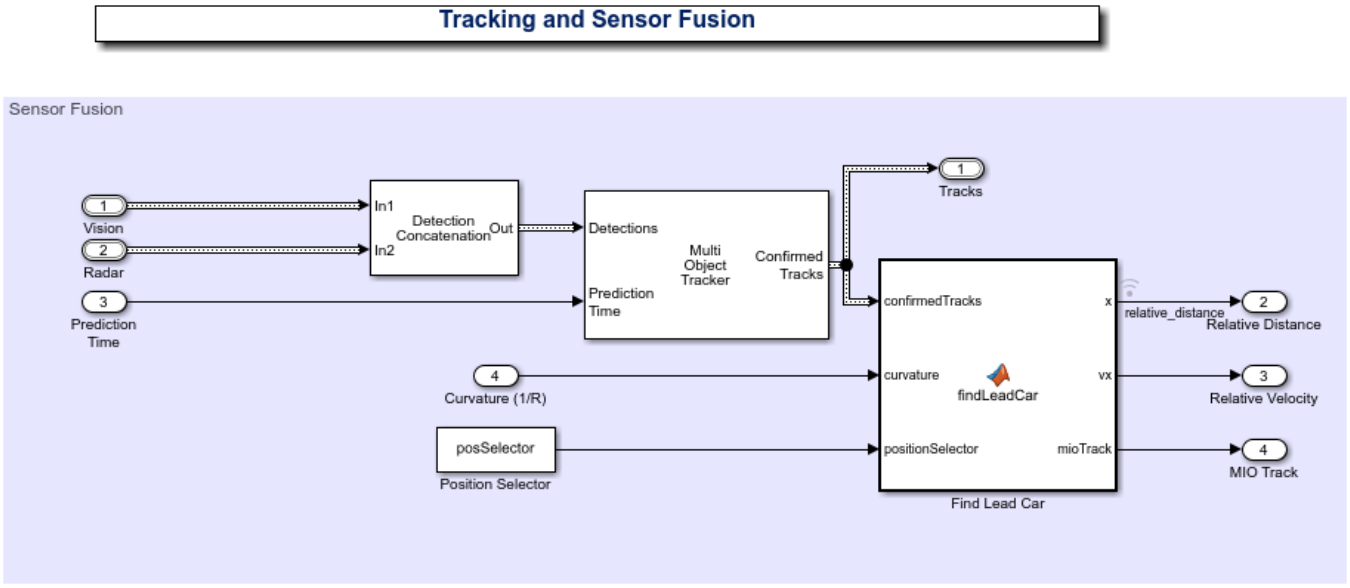


Copyright 2017-2021 The MathWorks, Inc.

Tracking and Sensor Fusion

The Tracking and Sensor Fusion subsystem processes vision and radar detections coming from the Vehicle and Environment subsystem and generates a comprehensive situation picture of the environment around the ego vehicle. Also, it provides the ACC with an estimate of the closest car in the lane in front of the ego vehicle.

```
open_system('ACCWithSensorFusionMdlRef/Tracking and Sensor Fusion')
```



The main block of the Tracking and Sensor Fusion subsystem is the Multi-Object Tracker (Automated Driving Toolbox) block, whose inputs are the combined list of all the sensor detections and the prediction time. The output from the Multi Object Tracker block is a list of confirmed tracks.

The Detection Concatenation (Automated Driving Toolbox) block concatenates the vision and radar detections. The prediction time is driven by a clock in the Vehicle and Environment subsystem.

The Detection Clustering block clusters multiple radar detections, since the tracker expects at most one detection per object per sensor.

The `findLeadCar` MATLAB function block finds which car is closest to the ego vehicle and ahead of it in same the lane using the list of confirmed tracks and the curvature of the road. This car is referred to as the lead car, and may change when cars move into and out of the lane in front of the ego vehicle. The function provides the position and velocity of the lead car relative to the ego vehicle and an index to the most important object (MIO) track.

Adaptive Cruise Controller

The adaptive cruise controller has two variants: a classical design (default) and an MPC-based design. For both designs, the following design principles are applied. An ACC equipped vehicle (ego vehicle) uses sensor fusion to estimate the relative distance and relative velocity to the lead car. The ACC makes the ego vehicle travel at a driver-set velocity while maintaining a safe distance from the lead car. The safe distance between lead car and ego vehicle is defined as

$$D_{safe} = D_{default} + T_{gap} \cdot V_x$$

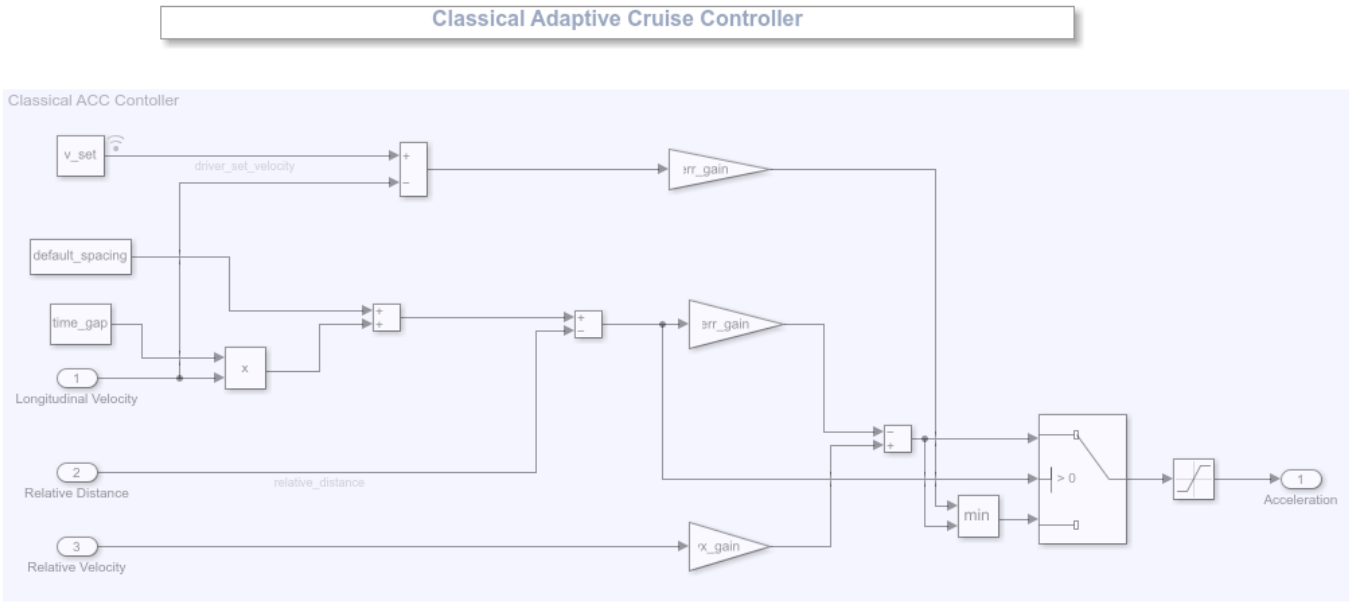
where the default spacing $D_{default}$, and time gap T_{gap} are design parameters and V_x is the longitudinal velocity of the ego vehicle. The ACC generates the longitudinal acceleration for the ego vehicle based on the following inputs:

- Longitudinal velocity of ego vehicle
- Relative distance between lead car and ego vehicle (from the Tracking and Sensor Fusion system)
- Relative velocity between lead car and ego vehicle (from the Tracking and Sensor Fusion system)

Considering the physical limitations of the ego vehicle, the longitudinal acceleration is constrained to the range [-3,2] m/s^2.

In the classical ACC design, if the relative distance is less than the safe distance, then the primary goal is to slow down and maintain a safe distance. If the relative distance is greater than the safe distance, then the primary goal is to reach driver-set velocity while maintaining a safe distance. These design principles are achieved through the Min and Switch blocks.

```
open_system('ACCWithSensorFusionMdlRef/Adaptive Cruise Controller/ACC Classical')
```

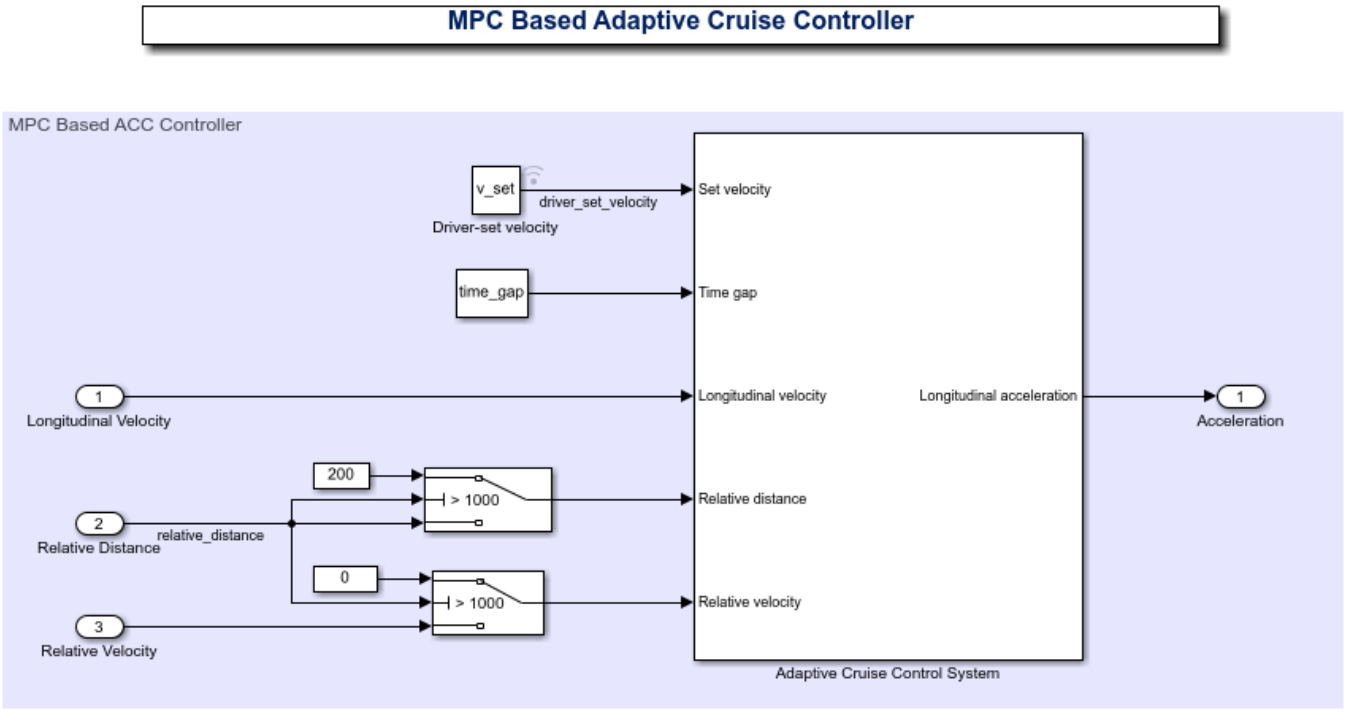


In the MPC-based ACC design, the underlying optimization problem is formulated by tracking the driver-set velocity subject to a constraint. The constraint enforces that relative distance is always greater than the safe distance.

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & |V - V_{set}|^2 \\ \text{subject to} \quad & D_{relative} - D_{safe} \geq 0 \\ & -3 \leq u \leq 2 \end{aligned}$$

To configure the Adaptive Cruise Control System block, use the parameters defined in the `helperACCSetUp` file. For example, the linear model for ACC design G, and G is obtained from vehicle dynamics. The two Switch blocks implement simple logic to handle large numbers from the sensor (for example, the sensor may return `Inf` when it does not detect an MIO).

```
open_system('ACCWithSensorFusionMdlRef/Adaptive Cruise Controller/ACC Model Predictive Control')
```



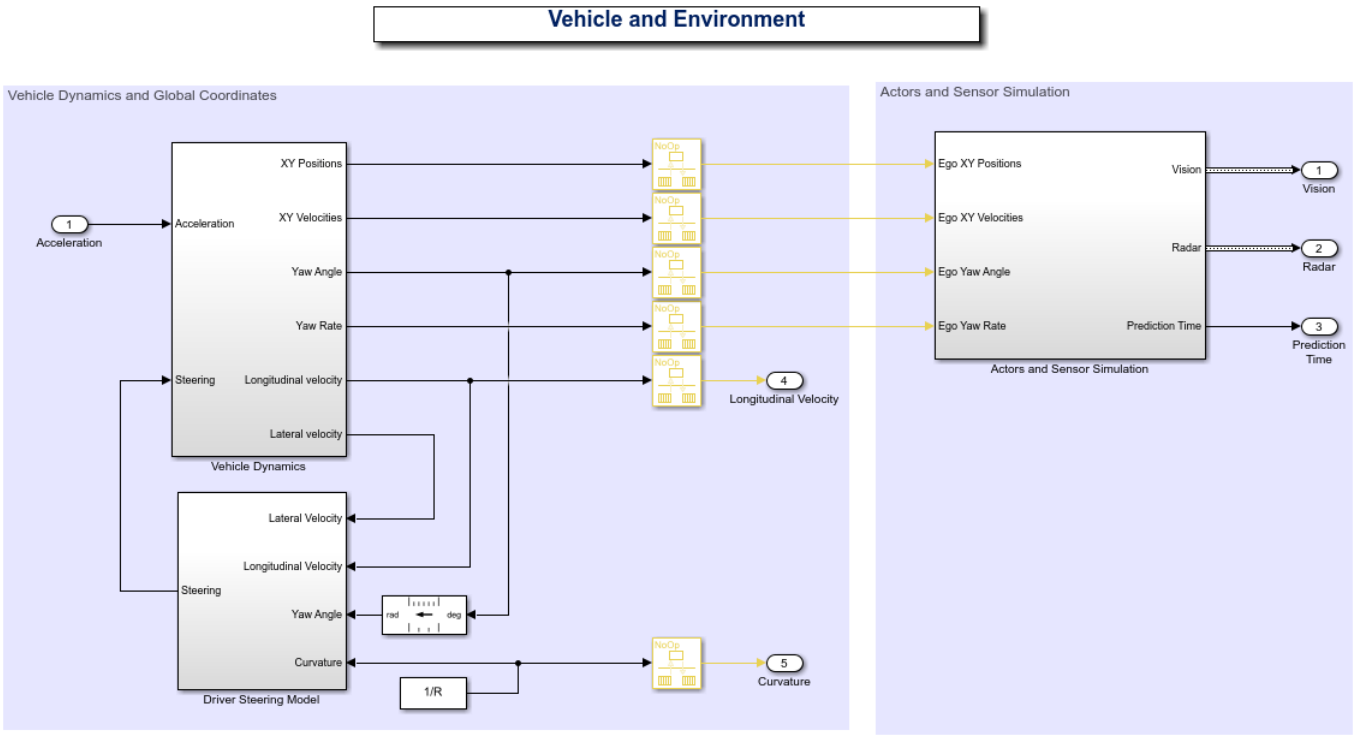
For more information on MPC design for ACC, see “Adaptive Cruise Control System Using Model Predictive Control” on page 11-5.

Vehicle and Environment

The Vehicle and Environment subsystem is composed of two parts:

1. Vehicle Dynamics and Global Coordinates
2. Actor and Sensor Simulation

```
open_system('ACCTestBenchExample/Vehicle and Environment')
```



The Vehicle Dynamics subsystem models the vehicle dynamics with the Bicycle Model - Force Input block from the Automated Driving Toolbox. The vehicle dynamics, with input u (longitudinal acceleration) and front steering angle , are approximated by:

$$\frac{d}{dt}\begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} = \begin{bmatrix} -\frac{2C_f+2C_r}{mV_x} & 0 & -V_x - \frac{2C_f\ell_f-2C_r\ell_r}{mV_x} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -\frac{2C_f\ell_f-2C_r\ell_r}{I_z V_x} & 0 & -\frac{2C_f\ell_f^2+2C_r\ell_r^2}{I_z V_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} + \begin{bmatrix} \frac{2C_f}{m} \\ 0 \\ \frac{2C_f\ell_f}{I_z} \\ 0 \\ 0 \end{bmatrix} \delta + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{\tau} \end{bmatrix} u + \begin{bmatrix} 0 \\ 0 \\ 0 \\ V_y\dot{\psi} \\ 0 \end{bmatrix}$$

In the state vector, V_y denotes the lateral velocity, V_x denotes the longitudinal velocity and ψ denotes the yaw angle. The vehicle parameters are provided in the `helperACCSetUp` file.

The outputs from the vehicle dynamics (such as longitudinal velocity V_x and lateral velocity V_y) are based on body fixed coordinates. To obtain the trajectory traversed by the vehicle, the body fixed coordinates are converted into global coordinates through the following relations:

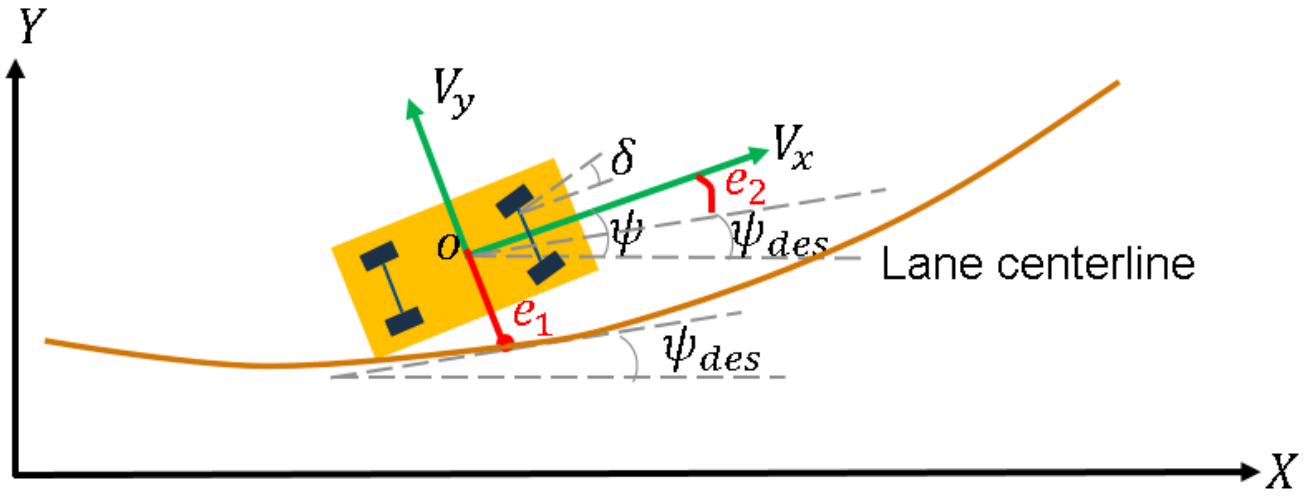
$$\dot{X} = V_x\cos(\psi) - V_y\sin(\psi), \quad \dot{Y} = V_x\sin(\psi) + V_y\cos(\psi)$$

The yaw angle ψ and yaw angle rate $\dot{\psi}$ are also converted into the units of degrees.

The goal for the driver steering model is to keep the vehicle in its lane and follow the curved road by controlling the front steering angle . This goal is achieved by driving the yaw angle error e_2 and lateral displacement error e_1 to zero (see the following figure), where

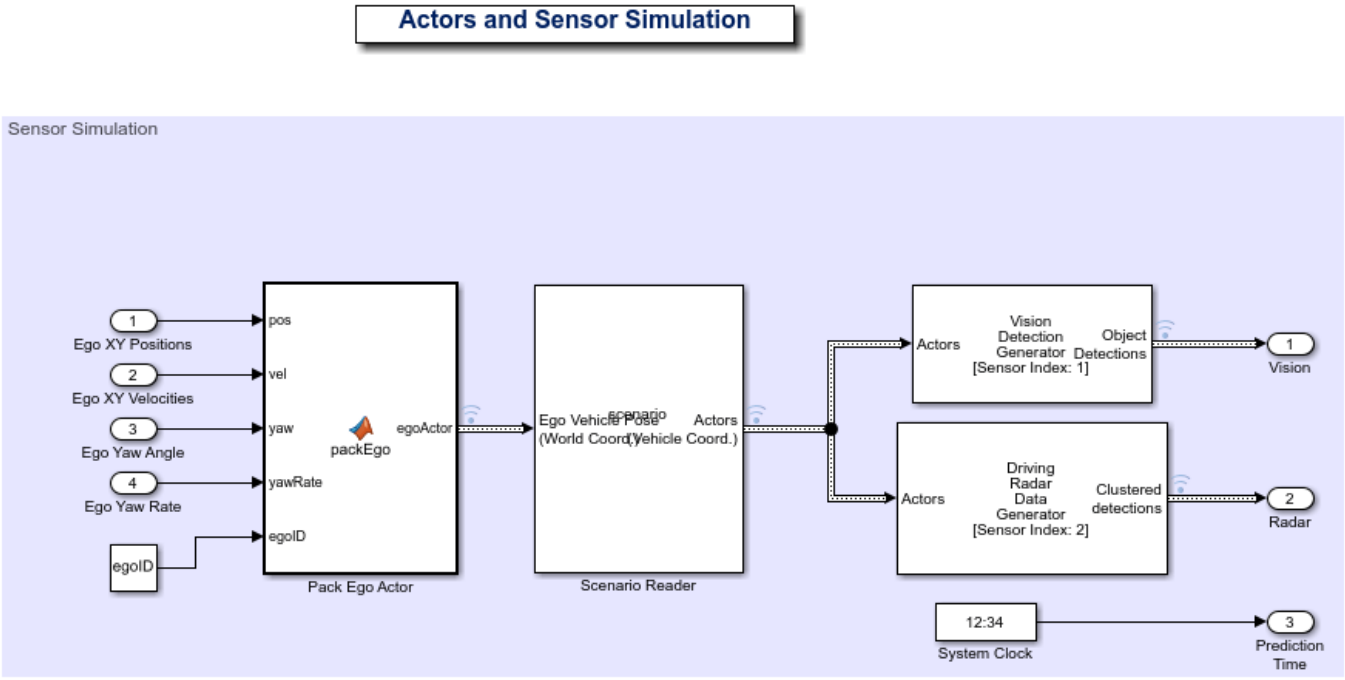
$$\dot{e}_1 = V_x e_2 + V_y, \quad e_2 = \psi - \psi_{des}$$

The desired yaw angle rate is given by Vx/R (R denotes the radius for the road curvature).



The Actors and Sensor Simulation subsystem generates the synthetic sensor data required for tracking and sensor fusion. Before running this example, the Driving Scenario Designer (Automated Driving Toolbox) app was used to create a scenario with a curved road and multiple actors moving on the road. The roads and actors from this scenario were then saved to the scenario file `ACCTestBenchScenario.mat`. To see how you can define the scenario, see the Scenario Creation section.

```
open_system('ACCTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```



The motion of the ego vehicle is controlled by the control system and is not read from the scenario file. Instead, the ego vehicle position, velocity, yaw angle, and yaw rate are received as inputs from the Vehicle Dynamics block and are packed into a single actor pose structure using the `packEgo` MATLAB function block.

The Scenario Reader (Automated Driving Toolbox) block reads the actor pose data from the scenario file `ACCTestBenchScenario.mat`. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. In this example, you use a Vision Detection Generator (Automated Driving Toolbox) block and Radar Detection Generator (Automated Driving Toolbox) block. Both sensors are long-range and forward-looking, and provide good coverage of the front of the ego vehicle, as needed for ACC. The sensors use the actor poses in ego vehicle coordinates to generate lists of vehicle detections in front of the ego vehicle. Finally, a clock block is used as an example of how the vehicle would have a centralized time source. The time is used by the Multi Object Tracker block.

Scenario Creation

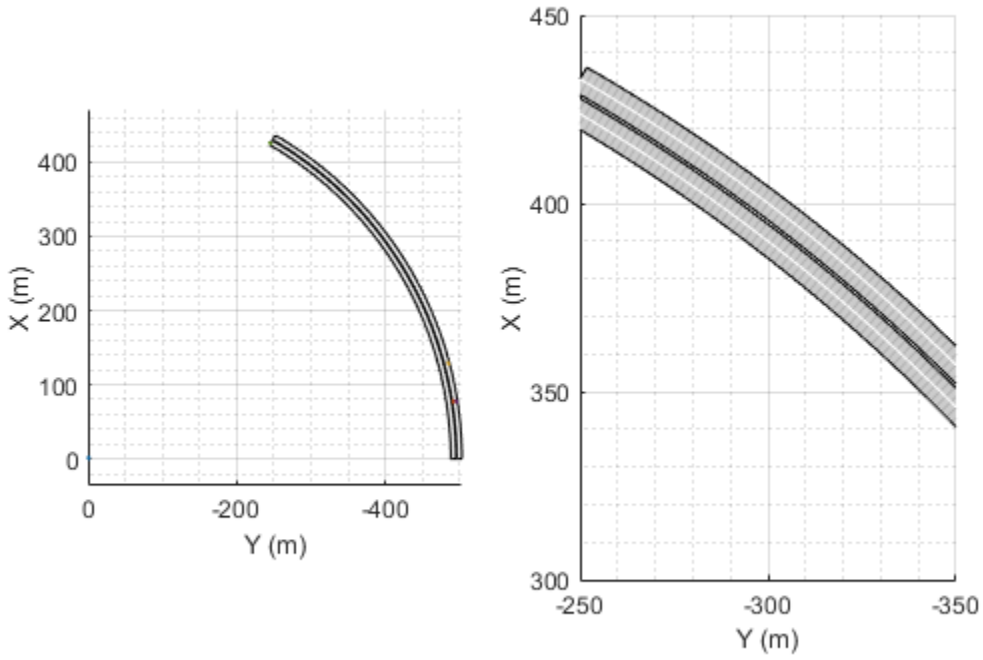
The **Driving Scenario Designer** app allows you to define roads and vehicles moving on the roads. For this example, you define two parallel roads of constant curvature. To define the road, you define the road centers, the road width, and banking angle (if needed). The road centers were chosen by sampling points along a circular arc, spanning a turn of 60 degrees of constant radius of curvature.

You define all the vehicles in the scenario. To define the motion of the vehicles, you define their trajectory by a set of waypoints and speeds. A quick way to define the waypoints is by choosing a subset of the road centers defined earlier, with an offset to the left or right of the road centers to control the lane in which the vehicles travel.

This example shows four vehicles: a fast-moving car in the left lane, a slow-moving car in the right lane, a car approaching on the opposite side of the road, and a car that starts on the right lane, but then moves to the left lane to pass the slow-moving car.

The scenario can be modified using the **Driving Scenario Designer** app and resaved to the same scenario file `ACCTestBenchScenario.mat`. The Scenario Reader block automatically picks up the changes when simulation is rerun. To build the scenario programmatically, you can use the `helperScenarioAuthoring` function.

```
plotACCScenario
```



Generating Code for the Control Algorithm

Although the entire model does not support code generation, the `ACCWithSensorFusionMdlRef` model is configured to support generating C code using Embedded Coder software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
    rtwbuild('ACCWithSensorFusionMdlRef')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `ACCWithSensorFusionMdlRef` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
    set_param('ACCTestBenchExample/ACC with Sensor Fusion',...
        'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `ACCTestBenchExample` model, code is generated, compiled, and executed for the `ACCWithSensorFusionMdlRef` model. This enables you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated adaptive cruise controller (ACC) on a curved road with sensor fusion, test it in Simulink using synthetic data generated by the Automated Driving Toolbox, componentize it, and automatically generate code for it.

```
bdclose all
```

See Also

Blocks

Adaptive Cruise Control System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

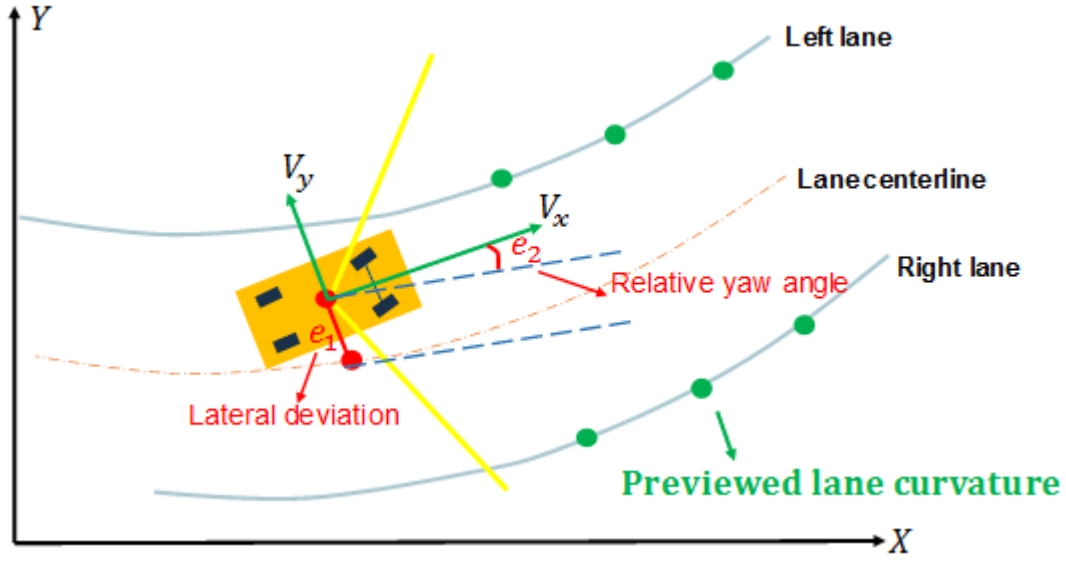
Lane Keeping Assist System Using Model Predictive Control

This example shows how to use the Lane Keeping Assist System block in Simulink® and demonstrates the control objectives and constraints of this block.

Lane Keeping Assist System

A vehicle (ego car) equipped with a lane-keeping assist (LKA) system has a sensor, such as camera, that measures the lateral deviation and relative yaw angle between the centerline of a lane and the ego car. The sensor also measures the current lane curvature and curvature derivative. Depending on the curve length that the sensor can view, the curvature in front of the ego car can be calculated from the current curvature and curvature derivative.

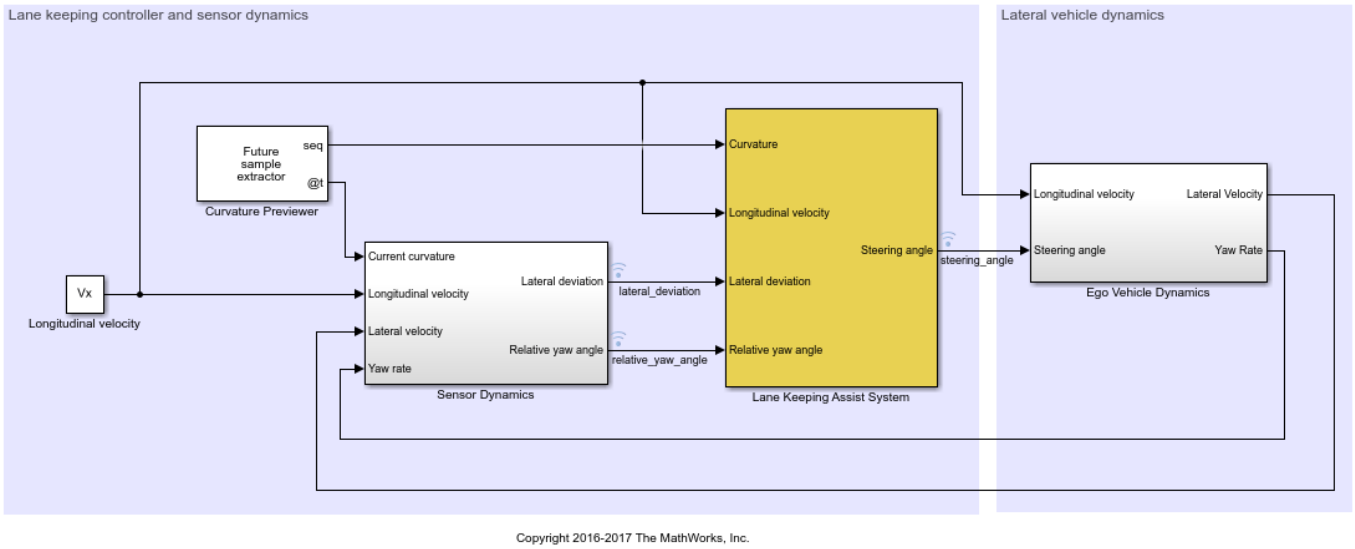
The LKA system keeps the ego car travelling along the centerline of the lanes on the road by adjusting the front steering angle of the ego car. The goal for lane keeping control is to drive both lateral deviation and relative yaw angle close to zero.



Simulink Model for Ego Car

The dynamics for ego car are modeled in Simulink. Open the Simulink model.

```
mdl = 'mpcLKAsystem';
open_system(mdl)
```



Define the sample time, `Ts`, and simulation duration, `T`, in seconds.

```
Ts = 0.1;
T = 15;
```

To describe the lateral vehicle dynamics, this example uses a *bicycle model* with the following parameters.

- `m` is the total vehicle mass (kg).
- `Iz` is the yaw moment of inertia of the vehicle (Kg*m^2).
- `lf` is the longitudinal distance from the center of gravity to the front tires (m).
- `lr` is the longitudinal distance from center of gravity to the rear tires (m).
- `Cf` is the cornering stiffness of the front tires (N/rad).
- `Cr` is the cornering stiffness of the rear tires (N/rad).

```
m = 1575;
Iz = 2875;
lf = 1.2;
lr = 1.6;
Cf = 19000;
Cr = 33000;
```

You can represent the lateral vehicle dynamics using a linear time-invariant (LTI) system with the following state, input, and output variables. The initial conditions for the state variables are assumed to be zero.

- State variables: Lateral velocity V_y and yaw angle rate r
- Input variable: Front steering angle
- Output variables: Same as state variables

In this example, the longitudinal vehicle dynamics are separated from the lateral vehicle dynamics. Therefore, the longitudinal velocity is assumed to be constant. In practice, the longitudinal velocity

can vary. The Lane Keeping Assist System block uses adaptive MPC to adjust the model of the lateral dynamics accordingly.

```
% Specify the longitudinal velocity in m/s.
Vx = 15;
```

Specify a state-space model, `G(s)`, of the lateral vehicle dynamics.

```
A = [-(2*Cf+2*Cr)/m/Vx, -Vx-(2*Cf*lf-2*Cr*lr)/m/Vx;...
     -(2*Cf*lf-2*Cr*lr)/Iz/Vx, -(2*Cf*lf^2+2*Cr*lr^2)/Iz/Vx];
B = [2*Cf/m, 2*Cf*lf/Iz]';
C = eye(2);
G = ss(A,B,C,0);
```

Sensor Dynamics and Curvature Previewer

In this example, the Sensor Dynamics block outputs the lateral deviation and relative yaw angle. The dynamics for relative yaw angle are $\dot{e}_2 = r - V_x\rho$, where ρ denotes the curvature. The dynamics for lateral deviation are $\dot{e}_1 = V_x e_2 + V_y$.

The Curvature Previewer block outputs the previewed curvature with a look-ahead time of one second. Therefore, given a sample time $Ts = 0.1$, the prediction horizon `10` steps. The curvature used in this example is calculated based on trajectories for a double lane change maneuver.

Specify the prediction horizon and obtain the previewed curvature.

```
PredictionHorizon = 10;

time = 0:0.1:15;
md = getCurvature(Vx,time);
```

Configuration of the Lane Keeping Assist System Block

The LKA system is modeled in Simulink using the Lane Keeping Assist System block. The inputs to the LKA system block are:

- Previewed curvature (from lane detections)
- Ego longitudinal velocity
- Lateral deviation (from lane detections)
- Relative yaw angle (from lane detections)

The output of the LKA system is the front steering angle of the ego car. Considering the physical limitations of the ego car, the steering angle is constrained to the range [-0.5,0.5] rad/s.

```
u_min = -0.5;
u_max = 0.5;
```

For this example, the default parameters of the Lane Keeping Assist System block match the simulation parameters. If your simulation parameters differ from the default values, update the block parameters accordingly.

Simulation Analysis

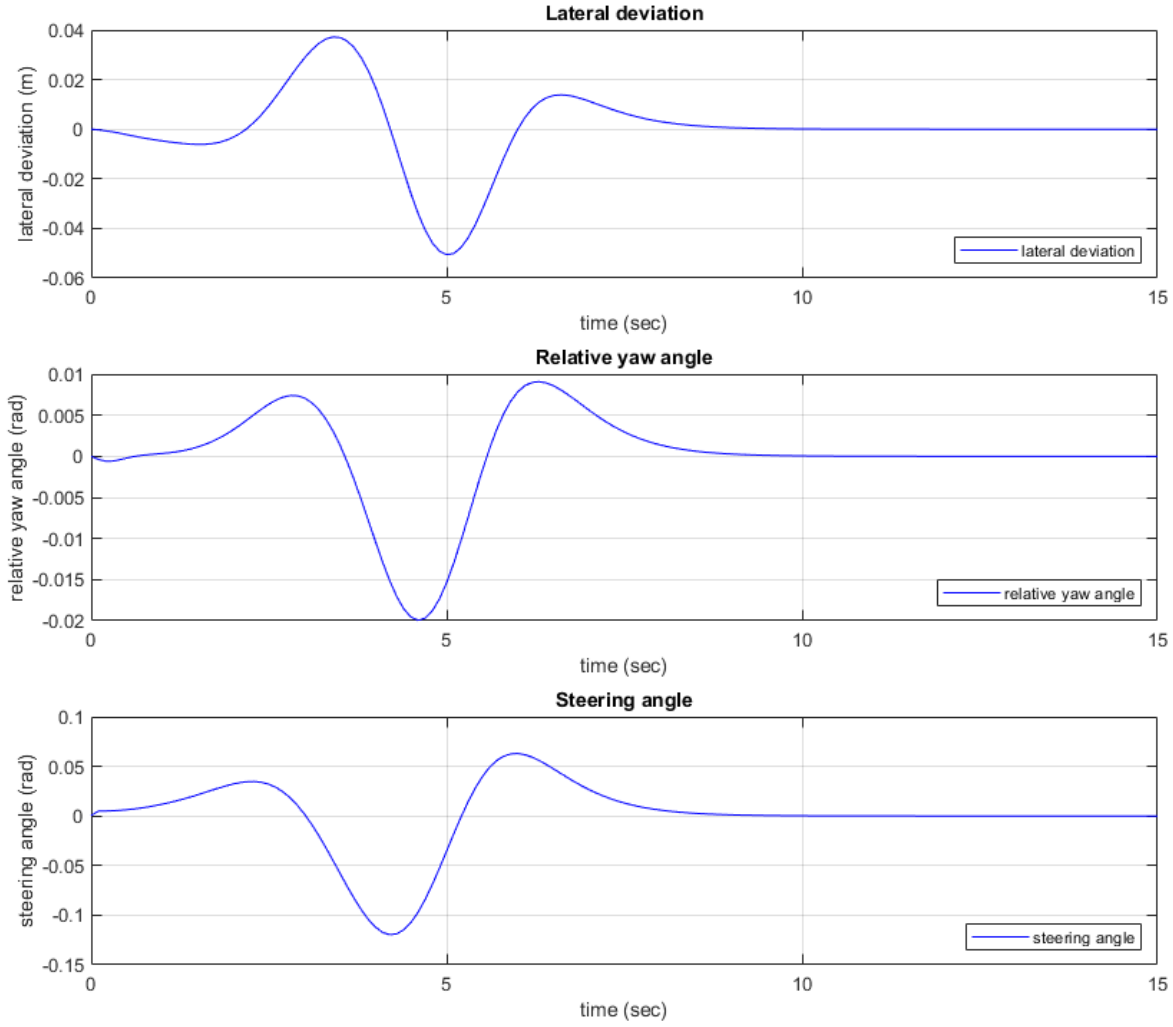
Run the model.

```
sim(mdl)
```

```
   Assuming no disturbance added to measured output channel #1.
-->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 measured
```

Plot the simulation results.

```
mpcLKAplot(logsout)
```



The lateral deviation and the relative yaw angle both converge to zero. That is, the ego car follows the road closely based on the previewed curvature.

```
% Close Simulink model.
bdclose(mdl)
```

See Also

Blocks
Lane Keeping Assist System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Lane Keeping Assist with Lane Detection” on page 11-33

Lane Keeping Assist with Lane Detection

This example shows how to simulate and generate code for an automotive lane keeping assist (LKA) controller.

In this example, you:

1. Review a control algorithm that combines data processing from lane detections and a lane keeping controller from the Model Predictive Control Toolbox™.
2. Test the control system in a closed-loop Simulink® model using synthetic data generated by the Automated Driving Toolbox™.
3. Configure the code generation settings for software-in-the-loop simulation and automatically generate code for the control algorithm.

Introduction

A lane keeping assist (LKA) system is a control system that aids a driver in maintaining safe travel within a marked lane of a highway. The LKA system detects when the vehicle deviates from a lane and automatically adjusts the steering to restore proper travel inside the lane without additional input from the driver. In this example, the LKA system switches between the driver steering command and lane keeping controller. This approach is sufficient to introduce a modeling architecture for an LKA system, however a real system would also provide haptic feedback to the steering wheel and enable the driver to override the LKA system by applying sufficient counter-torque.

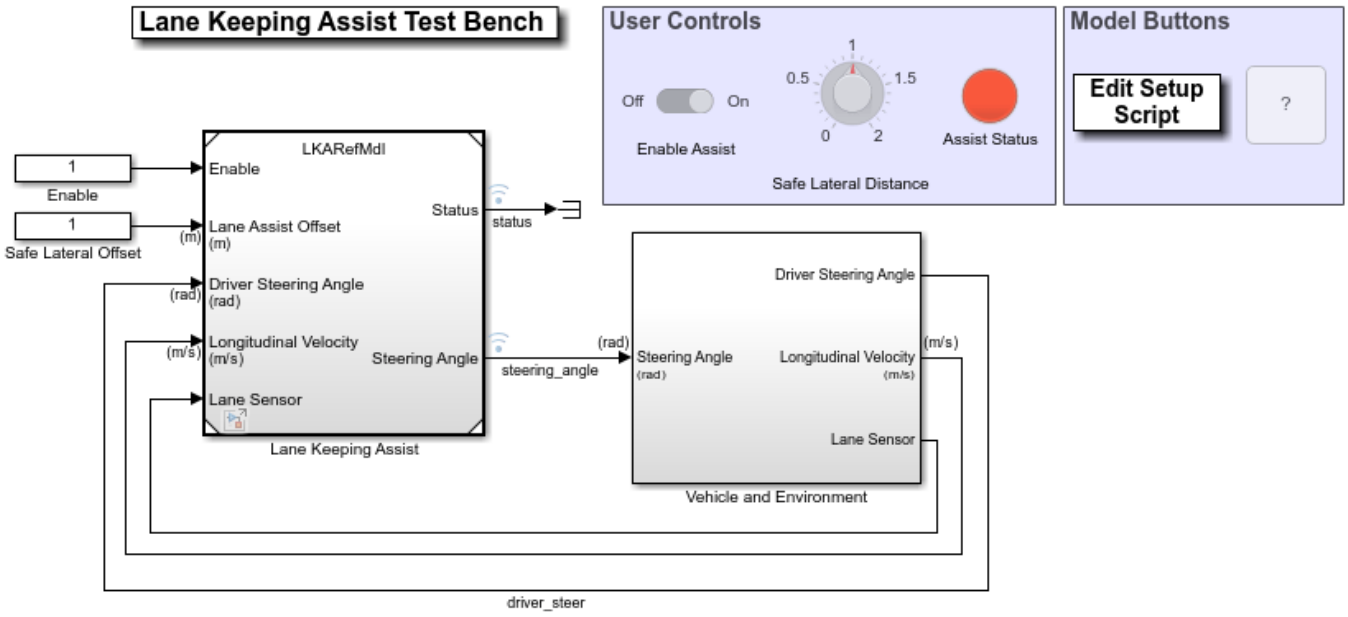
For the LKA to work correctly, the ego vehicle must determine the lane boundaries and how the lane in front of it curves. Idealized LKA designs rely mostly on the previewed curvature, the lateral deviation, and relative yaw angle between the centerline of the lane and the ego vehicle. An example of such a system is given in “Lane Keeping Assist System Using Model Predictive Control” on page 11-28. Moving from advanced drive-assistance system (ADAS) designs to more autonomous systems, the LKA must be robust to missing, incomplete, or inaccurate measurement readings from real-world lane detectors.

This example demonstrates a robust approach to the controller design when the data from lane detections may not be accurate. To do so, it uses data from a synthetic lane detector that simulates the impairments introduced by a wide-angle monocular vision camera. The controller makes decisions when the data from the sensor is invalid or outside a range. This provides a safety guard when the sensor measurement is false due to conditions in the environment, such as a sharp curve on the road.

Open Test Bench Model

To open the Simulink test bench model, use the following command.

```
open_system('LKATestBenchExample')
```



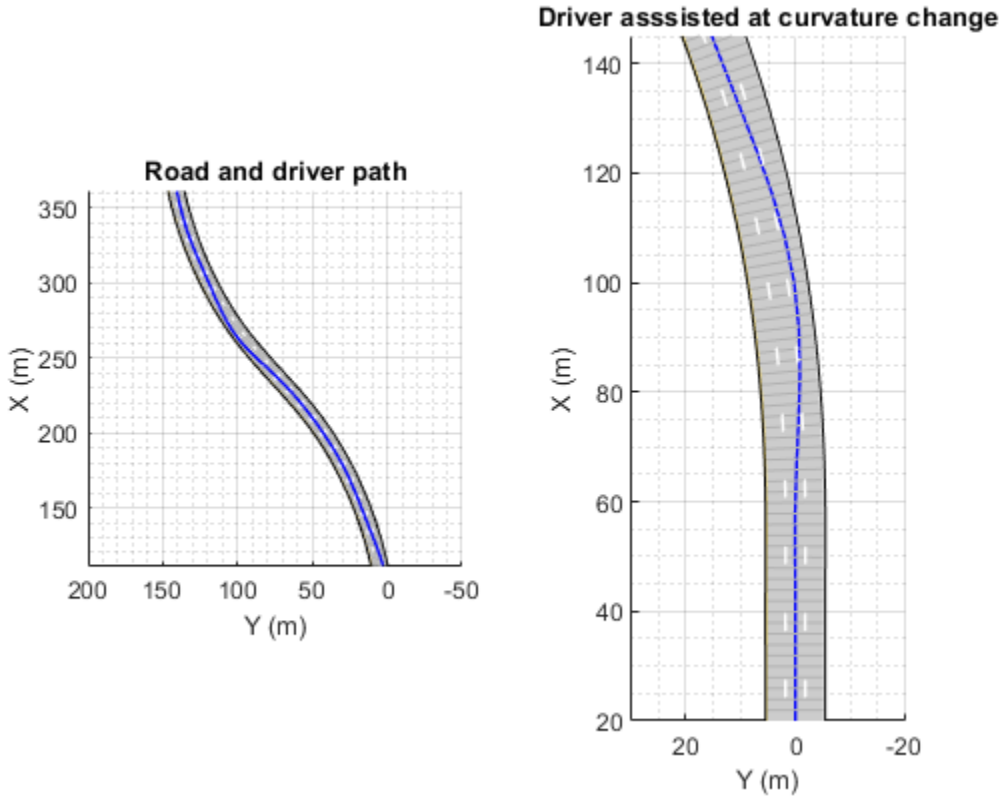
Copyright 2017-2020 The MathWorks, Inc.

The model contains two main subsystems:

1. Lane Keeping Assist, which controls the front steering angle of the vehicle
2. Vehicle and Environment subsystem, which models the motion of the ego vehicle and models the environment

Opening this model also runs the `helperLKASetUp` script, which initializes data used by the model. The script loads certain constants needed by the Simulink model, such as the vehicle model parameters, controller design parameters, road scenario, and driver path. You can plot the road and the path that the driver model will follow.

```
plotLKAInputs(scenario,driverPath)
```



Simulate Assisting a Distracted Driver

You can explore the behavior of the algorithm by enabling lane-keeping assistance and setting the safe lateral distance. In the Simulink model, in the **User Controls** section, switch the toggle to **On**, and set the **Safe Lateral Distance** to 1 meter. Alternatively, enable the lane-keeping assist and set the safe lateral distance.

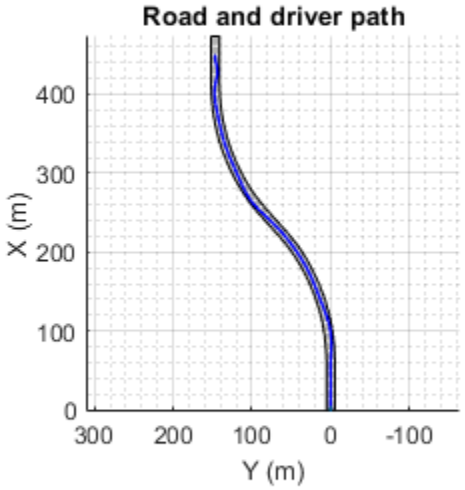
```
set_param('LKATestBenchExample/Enable','Value','1')
set_param('LKATestBenchExample/Safe Lateral Offset','Value','1')
```

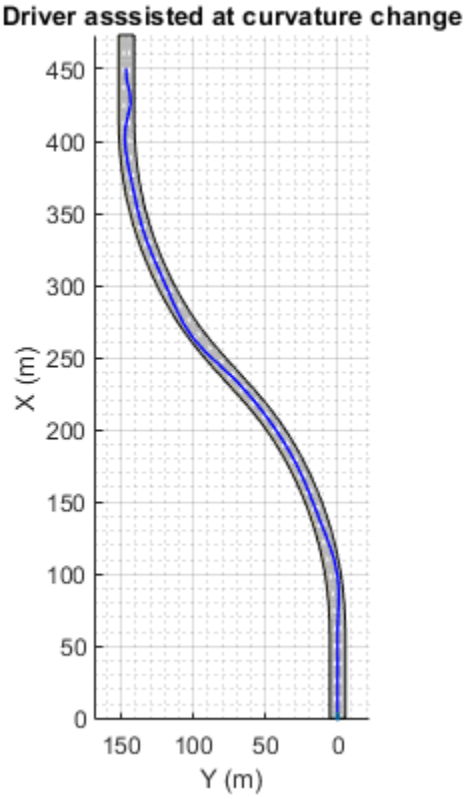
To plot the results of the simulation, use the Bird's-Eye Scope (Automated Driving Toolbox). The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation for 15 seconds, and explore the contents of the Bird's-Eye Scope.

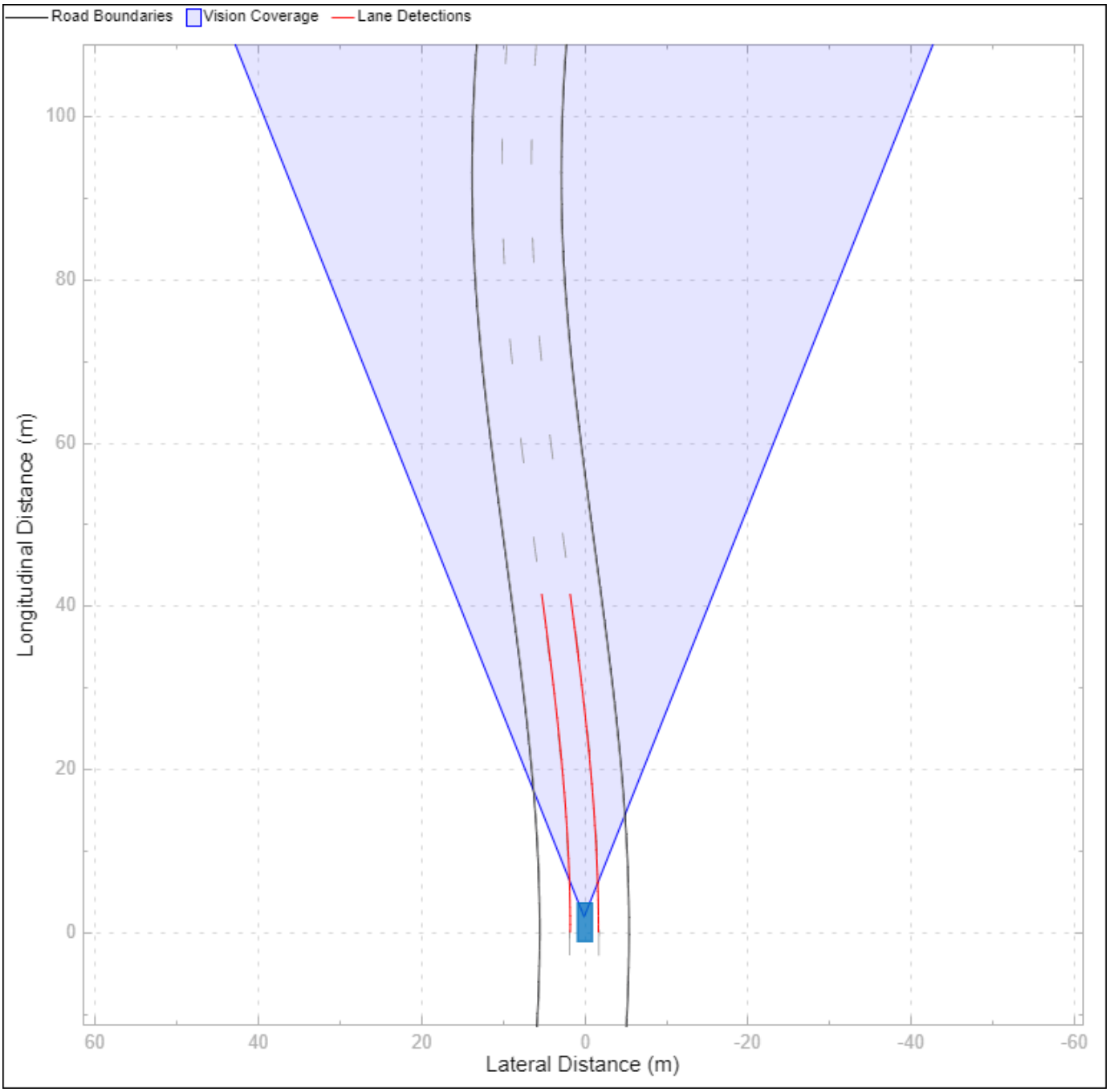
```
sim('LKATestBenchExample','StopTime','15') % Simulate 15 seconds
```

```
   Assuming no disturbance added to measured output channel #1.
-->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 measured

ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [4684x1 double]
```

```
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
      ErrorMessage: [0x0 char]
```





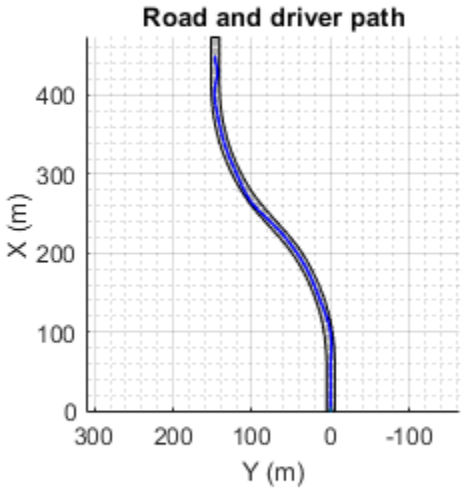


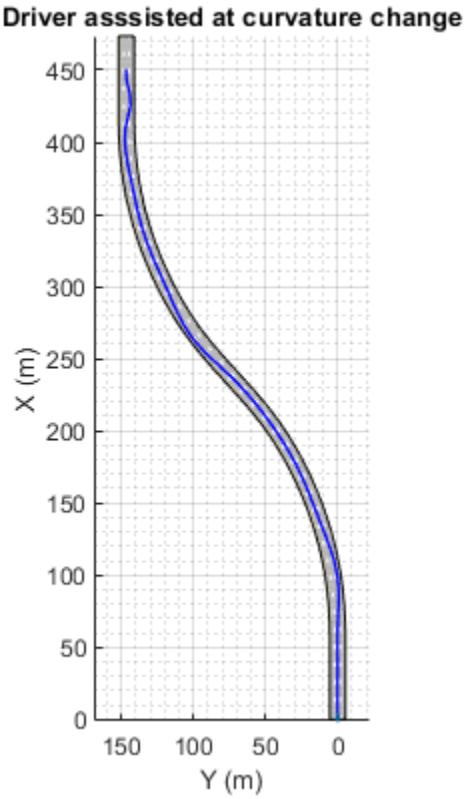
The Bird's-Eye Scope shows a symbolic representation of the road from the perspective of the ego vehicle. In this example, the Bird's-Eye Scope renders the coverage area of the synthetic vision detector as a shaded area. The ideal lane markings are additionally shown, as well as the synthetically detected left and right lane boundaries (shown here in red).

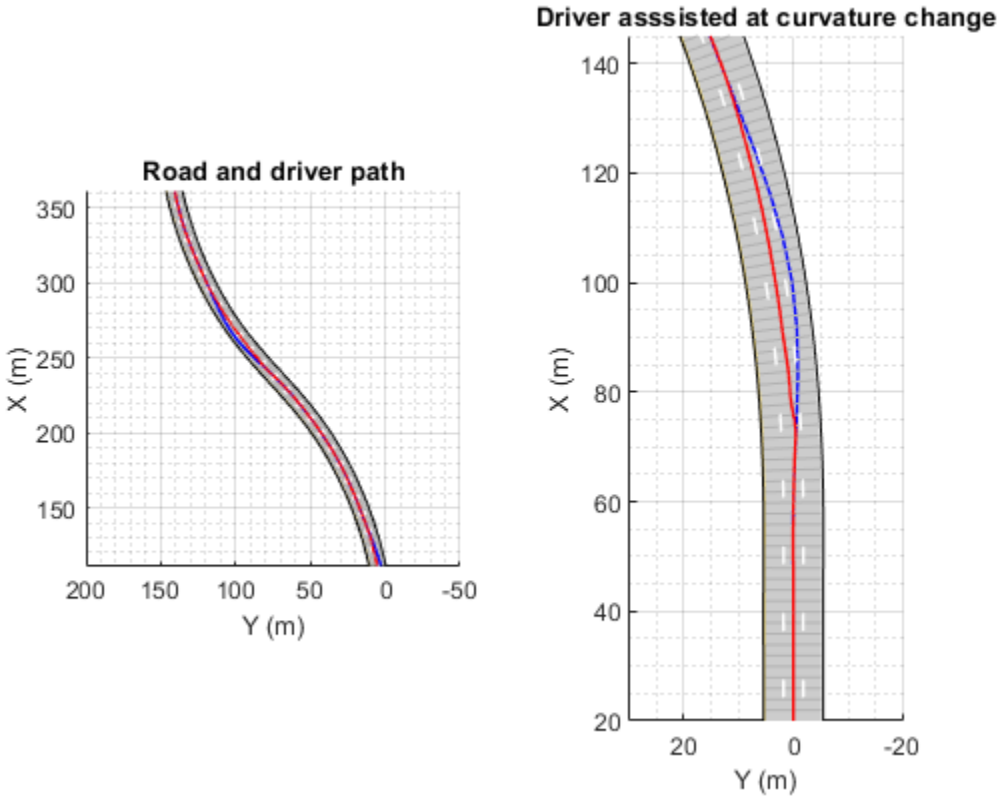
To run the full simulation and explore the results, use the following commands.

```
sim('LKATestBenchExample')                  % Simulate to end of scenario
plotLKAResults(scenario,logsout,driverPath)
```

```
   Assuming no disturbance added to measured output channel #1.
-->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 measured
```



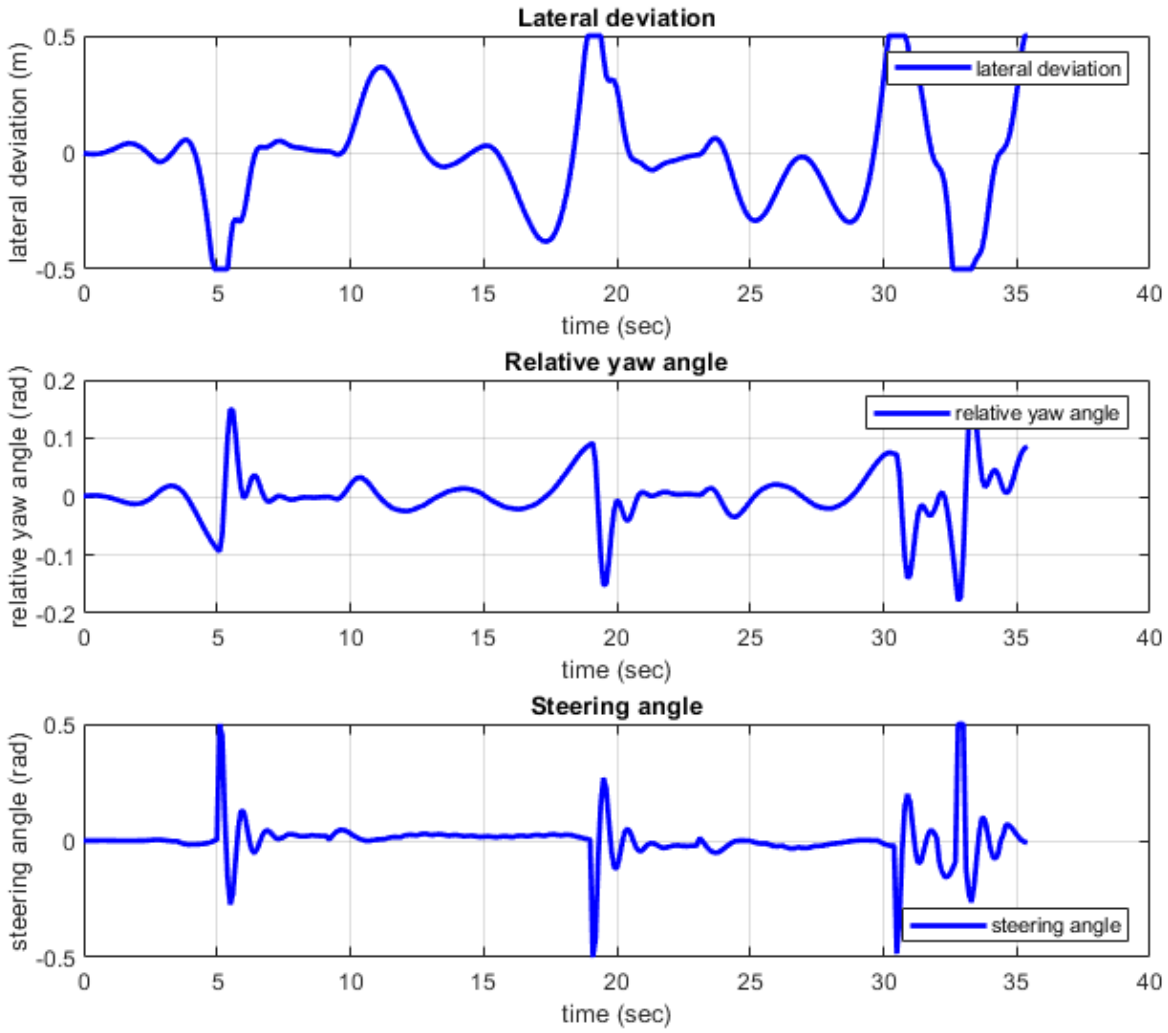




The blue curve for the driver path shows that the distracted driver may drive the ego vehicle to another lane when the road curvature changes. The red curve for the driver with Lane Keeping Assist shows that the ego vehicle remains in its lane when the road curvature changes.

To plot the controller performance, use the following command.

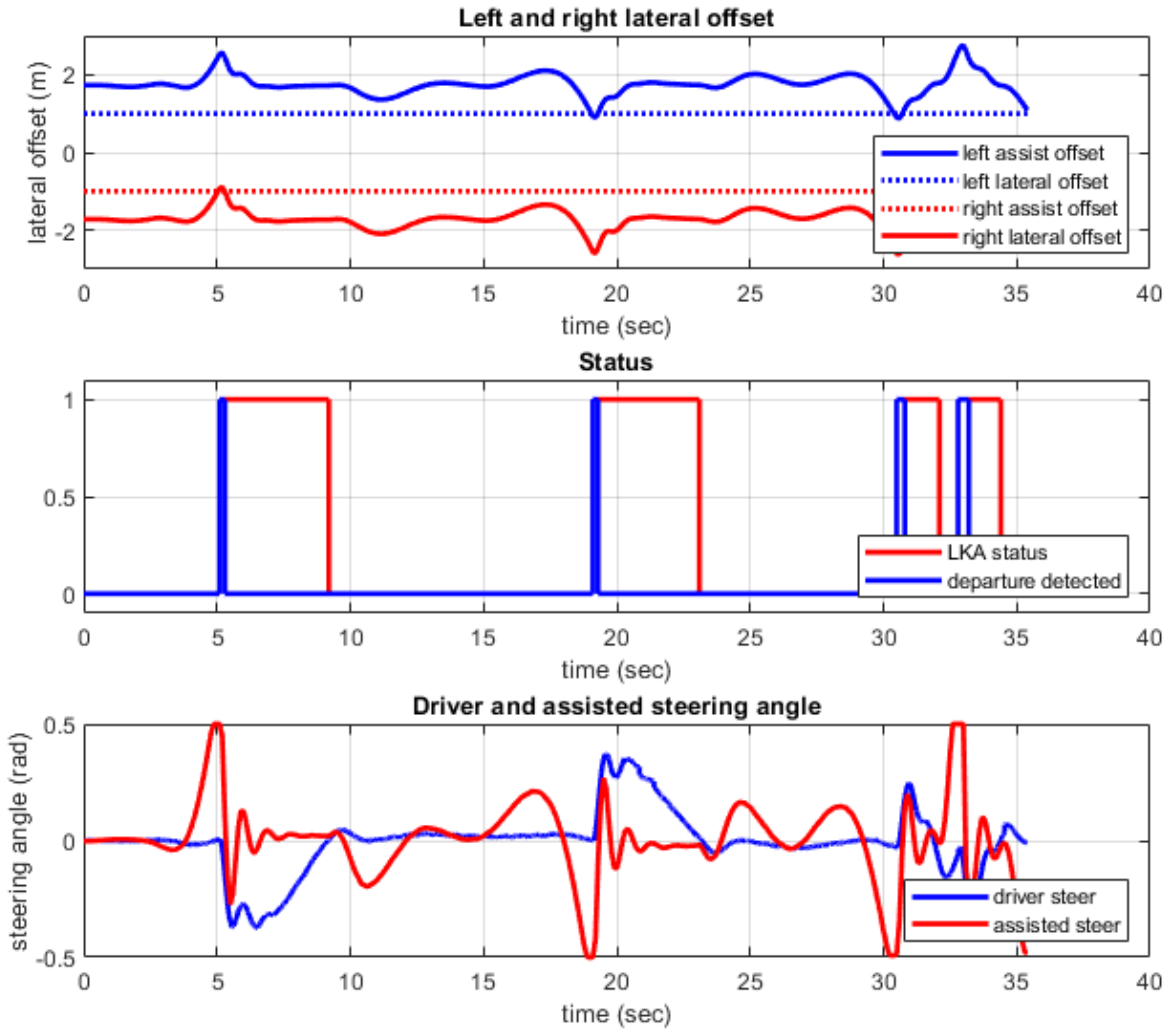
```
plotLKAPerformance(logsout)
```



- Top plot shows the lateral deviation relative to ego vehicle. The lateral deviation with LKA is within [-0.5,0.5] m.
- Middle plot shows the relative yaw angle. The relative yaw angle with LKA is within [-0.15,0.15] rad.
- Bottom plot shows the steering angle of the ego vehicle. The steering angle with LKA is within [-0.5,0.5] rad.

To view the controller status, use the following command.

```
plotLKAStatus(logsout)
```



- Top plot shows the left and right lane offset. Around 5.5 s, 19 s, 31 s, and 33 s, the lateral offset is within the distance set by the lane keeping assist. When this happens, the lane departure is detected.
- Middle plot shows the LKA status and the detection of lane departure. The departure detected status is consistent with the top plot. The LKA is turned on when the lane departure is detected, but the control is returned to the driver later when the driver can steer the ego vehicle correctly.
- Bottom plot shows the steering angle from driver and LKA. When the difference between the steering angle from driver and LKA is small, the LKA releases control to driver (for example, between 9 s to 17 s).

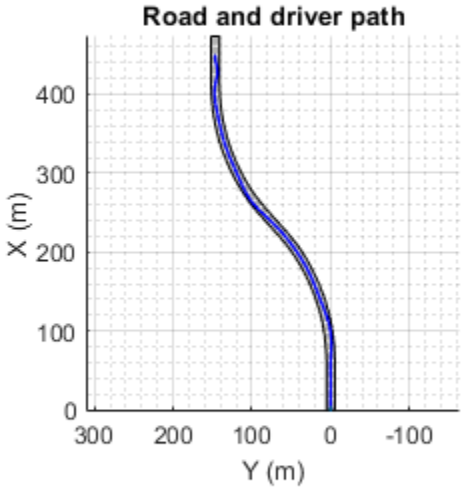
Simulate Lane Following

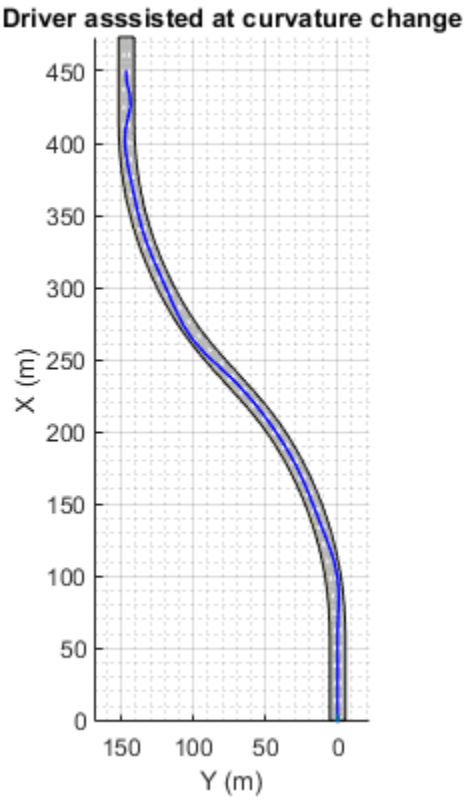
You can modify the value of Safe Lateral Offset for LKA to ignore the driver input, putting the controller into a pure lane following mode. By increasing this threshold, the lateral offset is always

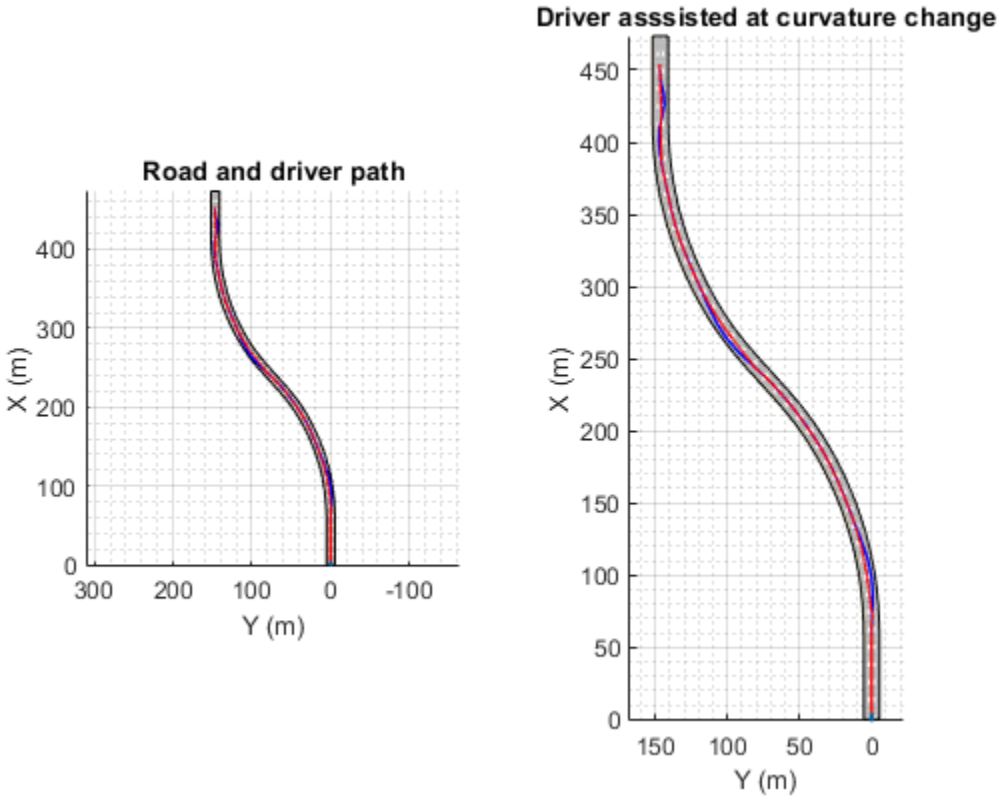
within the distance set by the lane keeping assist. Thus, the status for lane departure is on and the lane keeping assist takes control all the time.

```
set_param('LKATestBenchExample/Safe Lateral Offset','Value','2')
sim('LKATestBenchExample')                                    % Simulate to end of scenario
```

```
   Assuming no disturbance added to measured output channel #1.
-->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 measured
```

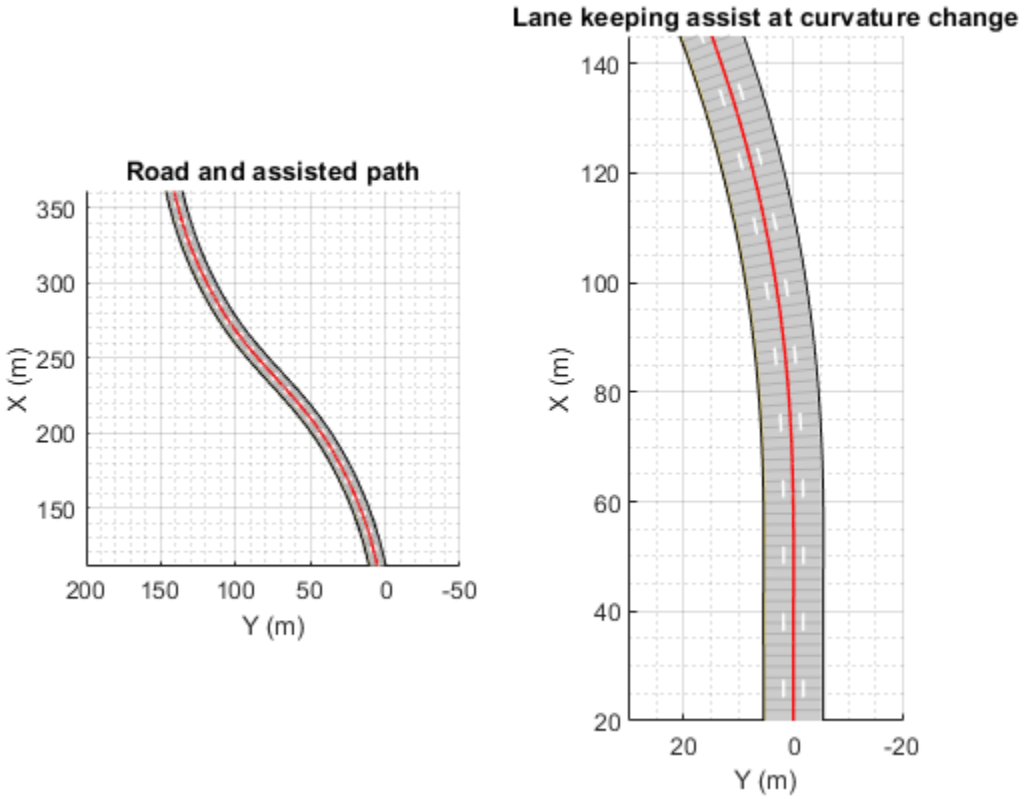






You can explore the results of the simulation using the following commands.

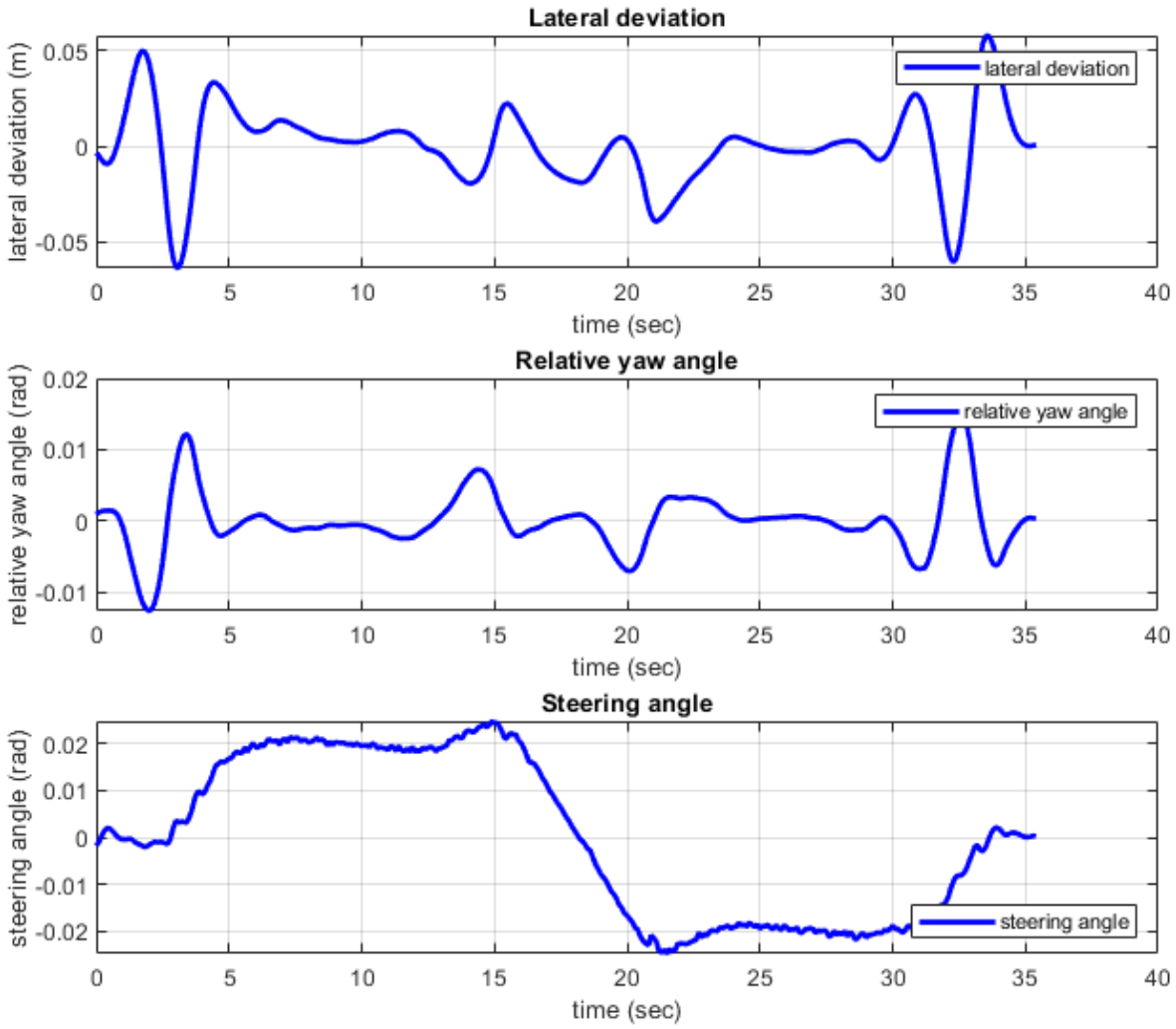
```
plotLKAResults(scenario,logsout)
```



The red curve shows that the Lane Keeping Assist on its own can keep the ego vehicle travelling along the centerline of its lane.

Use the following command to depict the controller performance.

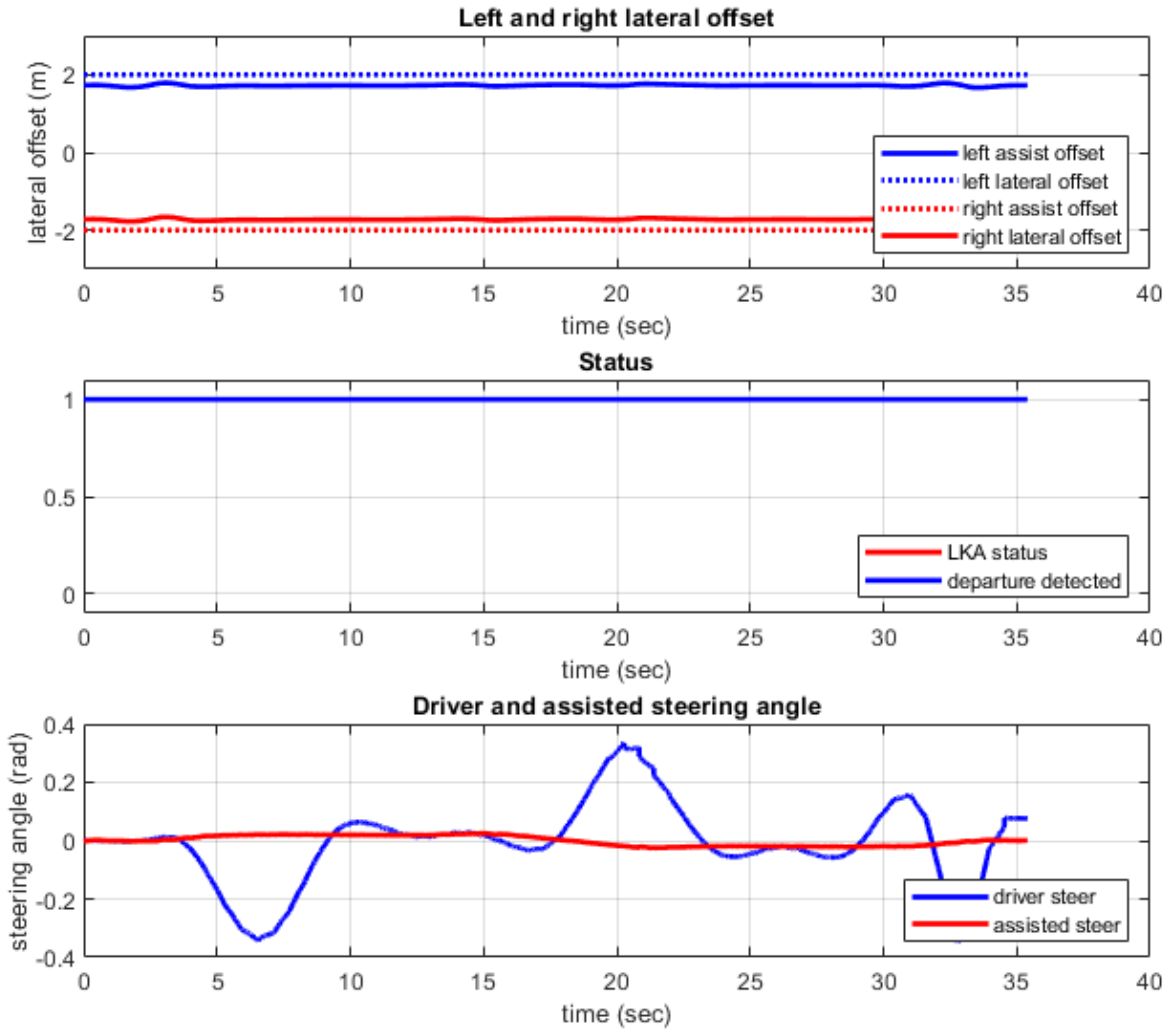
```
plotLKAPerformance(logsout)
```



- Top plot shows the lateral deviation relative to ego vehicle. The lateral deviation with LKA is within [-0.1,0.1] m.
- Middle plot shows the relative yaw angle. The relative yaw angle with LKA is within [-0.02,0.02] rad.
- Bottom plot shows the steering angle of the ego vehicle. The steering angle with LKA is within [-0.04,0.04] rad.

To view the controller status, use the following command.

```
plotLKAStatus(logsout)
```

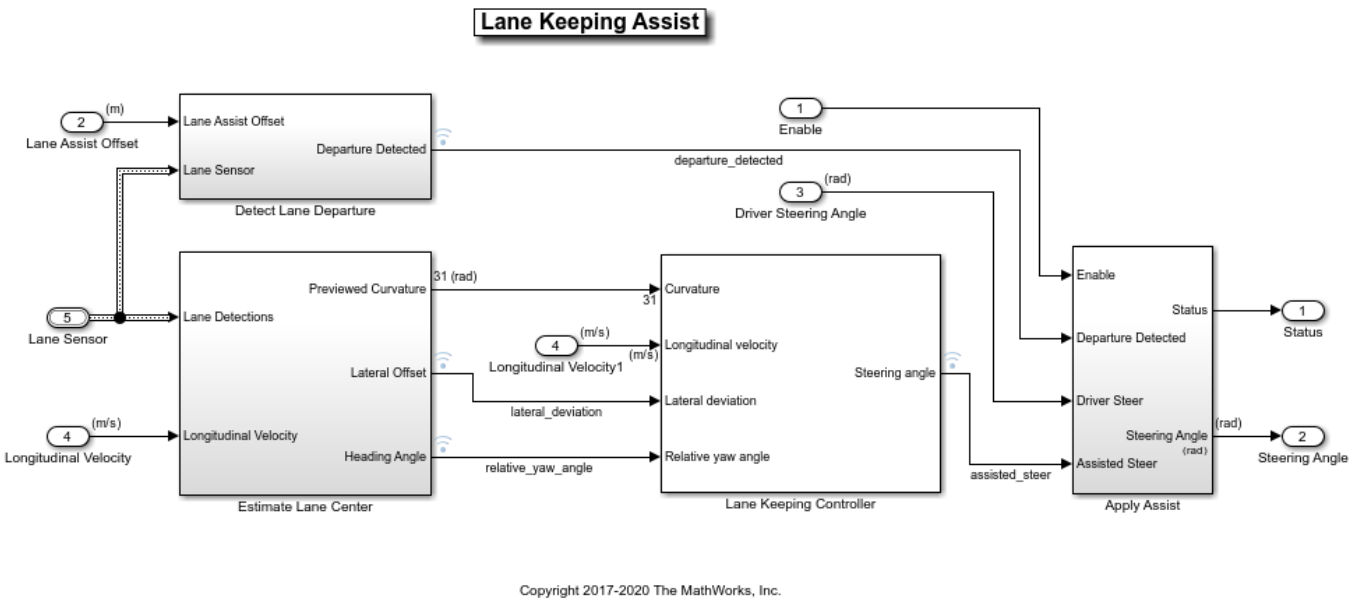


- Top plot shows the left and right lane offset. Since the lateral offset is never within the distance set by the lane keeping assist, the lane departure is not detected.
- Middle plot shows that the LKA status is always one, that is, the Lane Keeping Assist takes control all the time.
- Bottom plot shows the steering angle from driver and LKA. The steering angle from driver negotiating with the curved road is too aggressive. The small steering angle from LKA is sufficient for the curved road in this example.

Explore Lane Keeping Assist Algorithm

The Lane Keeping Assist model contains four main parts: 1) Estimate Lane Center 2) Lane Keeping Controller 3) Detect Lane Departure, and 4) Apply Assist.

```
open_system('LKATestBenchExample/Lane Keeping Assist')
```

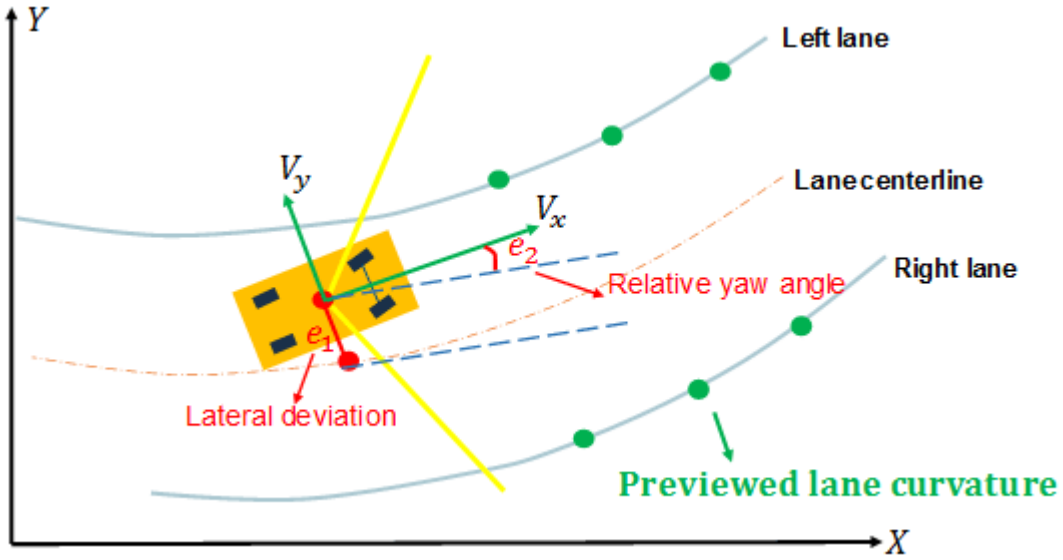


Copyright 2017-2020 The MathWorks, Inc.

The Detect Lane Departure subsystem outputs a signal that is true when the vehicle is too close to a detected lane. You detect a departure when the offset between the vehicle and lane boundary from the Lane Sensor is less than the Lane Assist Offset input.

The Estimate Lane Center subsystem outputs the data from lane sensors to the lane keeping controller. The detector in this example is configured to report the left and right lane boundaries of the current lane in the current field-of-view of the camera. Each boundary is modeled as a length of a curve whose curvature varies linearly with distance (clothoid curve). To feed this data to a controller, offset both of the detected curves toward the center of the lane by the width of the car and a small margin (1.8 m total). Weight each of the resulting centered curves by the strength of the detection and pass the averaged result to the controller. Also, The Estimate Lane Center subsystem provides finite values for inputs to the Lane Keeping Controller subsystem. The previewed curvature provides the centerline of lane curvature ahead of the ego vehicle. In this example, the ego vehicle can look ahead for three seconds, which is the product of the prediction horizon and sample time. This look-ahead time enables the controller to use previewed information for calculating steering angle for the ego vehicle, which improves the MPC controller performance.

The goal for the Lane Keeping Controller block is to keep the vehicle in its lane and follow the curved road by controlling the front steering angle . This goal is achieved by driving the lateral deviation e_1 and the relative yaw angle e_2 to be small (see the following figure).



The LKA controller calculates a steering angle for the ego vehicle based on the following inputs:

- Previewed curvature (derived from Lane Detections)
- Ego vehicle longitudinal velocity
- Lateral deviation (derived from Lane Detections)
- Relative yaw angle (derived from Lane Detections)

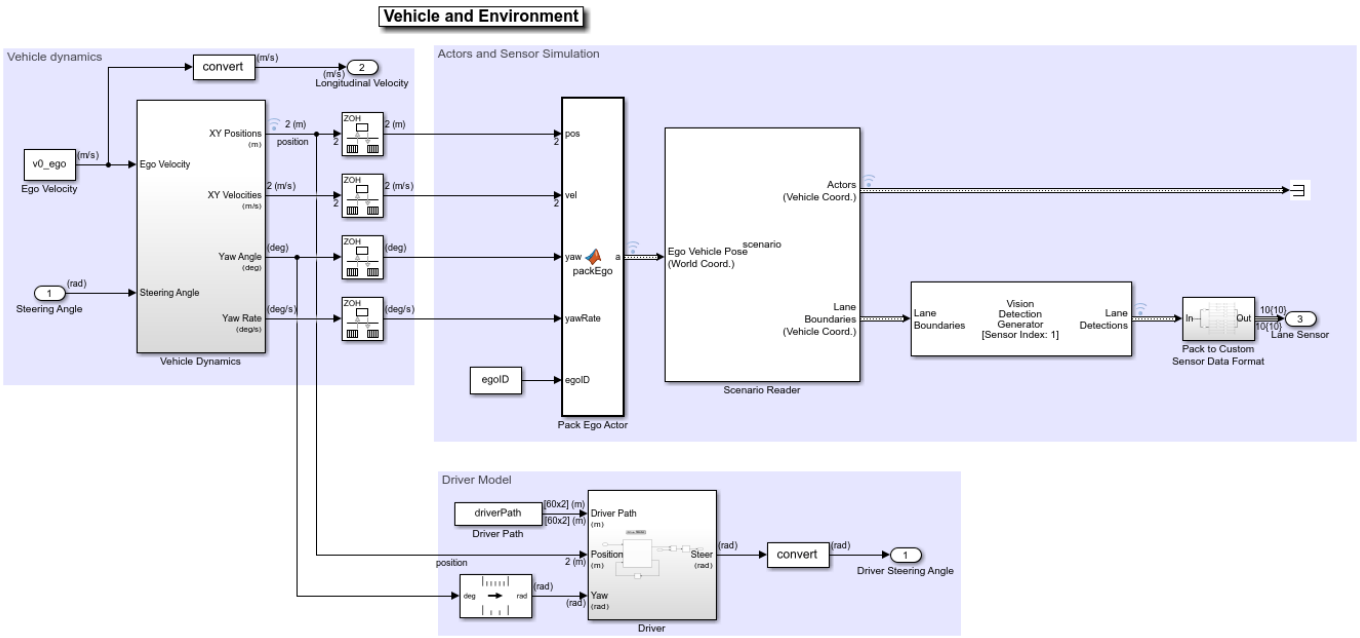
Considering physical limitations of the ego vehicle, the steering angle is constrained to be within [-0.5,0.5] rad. You can change the prediction horizon or move the **Controller Behavior** slider to adjust the performance of the controller.

The Apply Assist subsystem decides if the lane keeping controller or the driver takes control of the ego vehicle. The subsystem switches between the driver commanded steering and the assisted steering from the Lane Keeping Controller. The switch to assisted steering is initiated when a lane departure is detected. Control is returned to the driver when the driver begins steering within the lane again.

Explore Vehicle and Environment

The Vehicle and Environment subsystem enables closed loop simulation of the lane keeping assist controller.

```
open_system('LKATestBenchExample/Vehicle and Environment')
```



The Vehicle Dynamics subsystem models the vehicle dynamics with Vehicle Body 3DOF Single Track block from Vehicle Dynamics Blockset™.

The Scenario Reader (Automated Driving Toolbox) block generates the ideal left and right lane boundaries based on the position of the vehicle with respect to the scenario read from scenario file `LKATestBenchScenario.mat`.

The Vision Detection Generator block takes the ideal lane boundaries from the Scenario Reader block. The detection generator models the field of view of a monocular camera and determines the heading angle, curvature, curvature derivative, and valid length of each road boundary, accounting for any other obstacles.

The Driver subsystem generates the driver steering angle based on the driver path which was created in `helperLKASetUp`.

Generate Code for the Control Algorithm

The `LKARefMdl` model is configured to support generating C code using Embedded Coder® software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
    slbuild('LKARefMdl')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `LKARefMdl` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
    set_param('LKATestBenchExample/Lane Keeping Assist',...
```

```
        'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `LKATestBenchExample` model, code is generated, compiled, and executed for the `LKARefMdl` model. This allows you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated lane keeping assist (LKA) controller on a curved road with lane detection. It also shows how to test the controller in Simulink using synthetic data generated by the Automated Driving Toolbox, componentize it, and automatically generate code for it.

```
close all
bdclose all
```

See Also

Blocks
Lane Keeping Assist System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Lane Keeping Assist System Using Model Predictive Control” on page 11-28

Lane Following Control with Sensor Fusion and Lane Detection

This example shows how to simulate and generate code for an automotive lane-following controller.

In this example, you:

1. Review a control algorithm that combines sensor fusion, lane detection, and a lane following controller from the Model Predictive Control Toolbox™ software.
2. Test the control system in a closed-loop Simulink® model using synthetic data generated by Automated Driving Toolbox™ software.
3. Configure the code generation settings for software-in-the-loop simulation and automatically generate code for the control algorithm.

Introduction

A lane following system is a control system that keeps the vehicle traveling within a marked lane of a highway, while maintaining a user-set velocity or safe distance from the preceding vehicle. A lane following system includes combined longitudinal and lateral control of the ego vehicle:

- Longitudinal control - Maintain a driver-set velocity and keep a safe distance from the preceding car in the lane by adjusting the acceleration of the ego vehicle.
- Lateral control - Keep the ego vehicle traveling along the centerline of its lane by adjusting the steering of the ego vehicle

The combined lane following control system achieves the individual goals for longitudinal and lateral control. Further, the lane following control system can adjust the priority of the two goals when they cannot be met simultaneously.

For an example of longitudinal control using adaptive cruise control (ACC) with sensor fusion, see “Adaptive Cruise Control with Sensor Fusion” on page 11-10. For an example of lateral control using a lane keeping assist (LKA) system with lane detection, see “Lane Keeping Assist with Lane Detection” on page 11-33. The ACC example assumes ideal lane detection, and the LKA example does not consider surrounding vehicles.

In this example, both lane detection and surrounding cars are considered. The lane following system synthesizes data from vision and radar detections, estimates the lane center and lead car distance, and calculates the longitudinal acceleration and steering angle of the ego vehicle.

Define Scenario

Before opening the model, you can optionally change the scenario that the model simulates. This scenario selection is controlled by a callback function, `helperLFSetUp`, which runs when the model opens.

By default, the model simulates a cut-in scenario on a curved road. To change the default scenario used, either edit the setup script by clicking the **Edit Setup Script** button in the model or by calling `helperLFSetup` with a new input scenario. For example, the following syntax is equivalent to specifying the default scenario.

```
helperLFSetup('LFACC_04_Curve_CutInOut');
```

You can choose from the following scenarios.

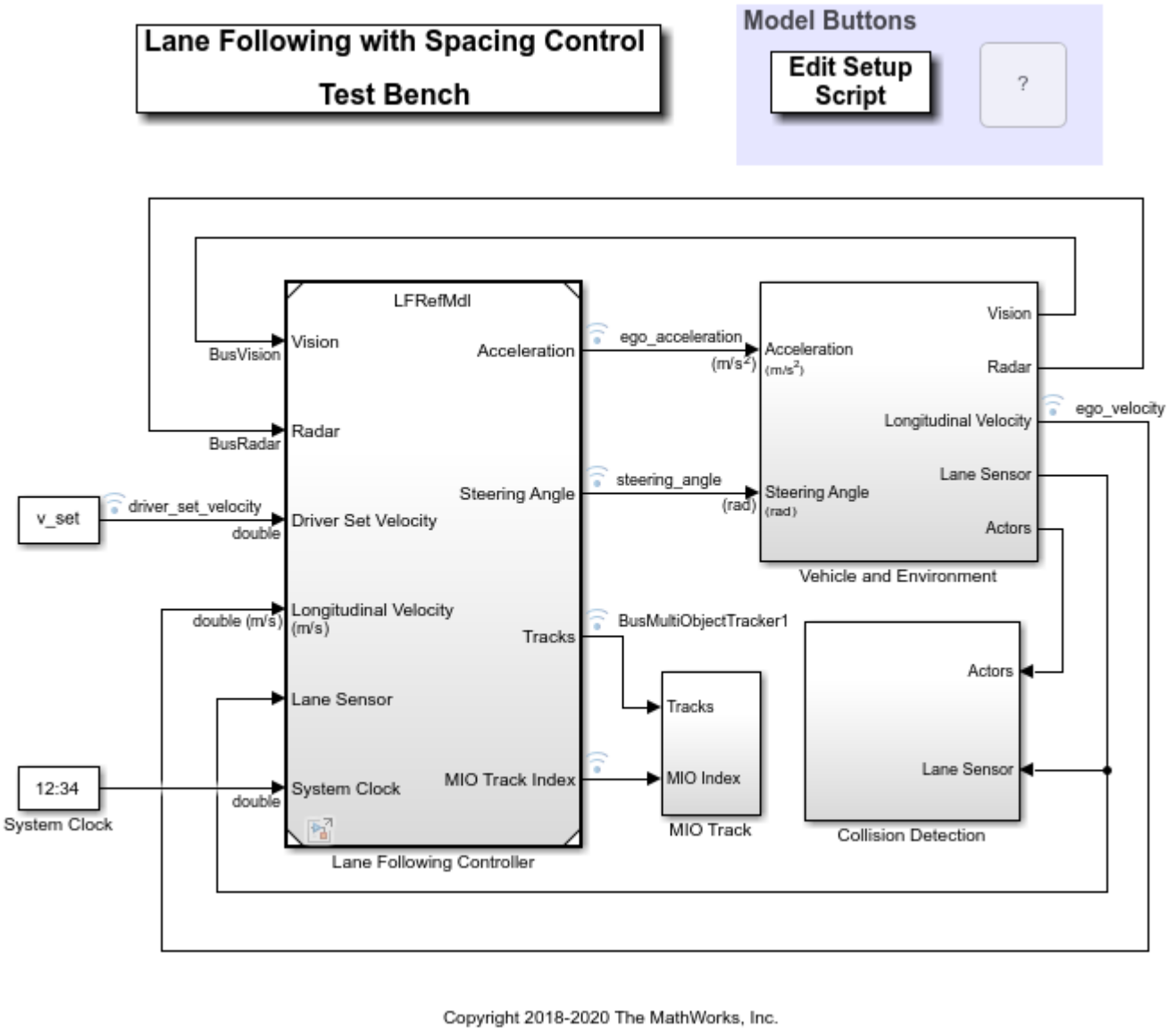
```
'ACC_01_ISO_TargetDiscriminationTest'
'ACC_02_ISO_AutoRetargetTest'
```

```
'ACC_03_ISO_CurveTest'
'ACC_04_StopnGo'
'LFACC_01_DoubleCurve_DecelTarget'
'LFACC_02_DoubleCurve_AutoRetarget'
'LFACC_03_DoubleCurve_StopnGo'
'LFACC_04_Curve_CutInOut'
'LFACC_05_Curve_CutInOut_TooClose'
```

Open Test Bench Model

Open the Simulink test bench model.

```
open_system('LaneFollowingTestBenchExample')
```



Copyright 2018-2020 The MathWorks, Inc.

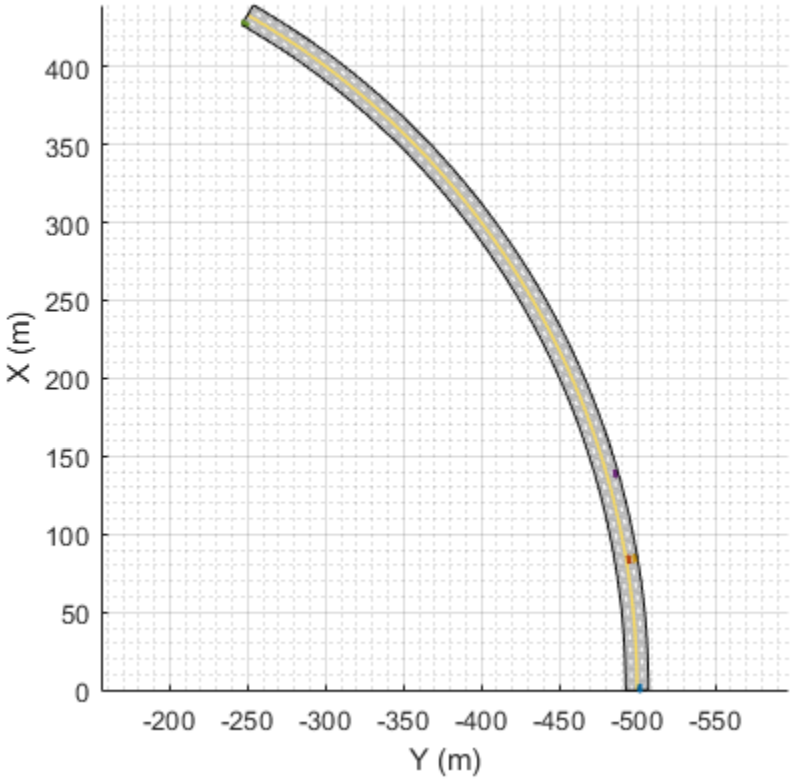
The model contains four main components:

1. Lane Following Controller - Controls both the longitudinal acceleration and front steering angle of the ego vehicle
2. Vehicle and Environment - Models the motion of the ego vehicle and models the environment
3. Collision Detection - Stops the simulation when a collision of the ego vehicle and lead vehicle is detected
4. MIO Track - Enables MIO track for display in the Bird's-Eye Scope.

Opening this model also runs the `helperLFSetUp` script, which initializes the data used by the model by running the scenario function and loading constants needed by the Simulink model, such as the vehicle model parameters, controller design parameters, road scenario, and surrounding cars.

Plot the road and the path that the ego vehicle will follow.

```
plot(scenario)
```

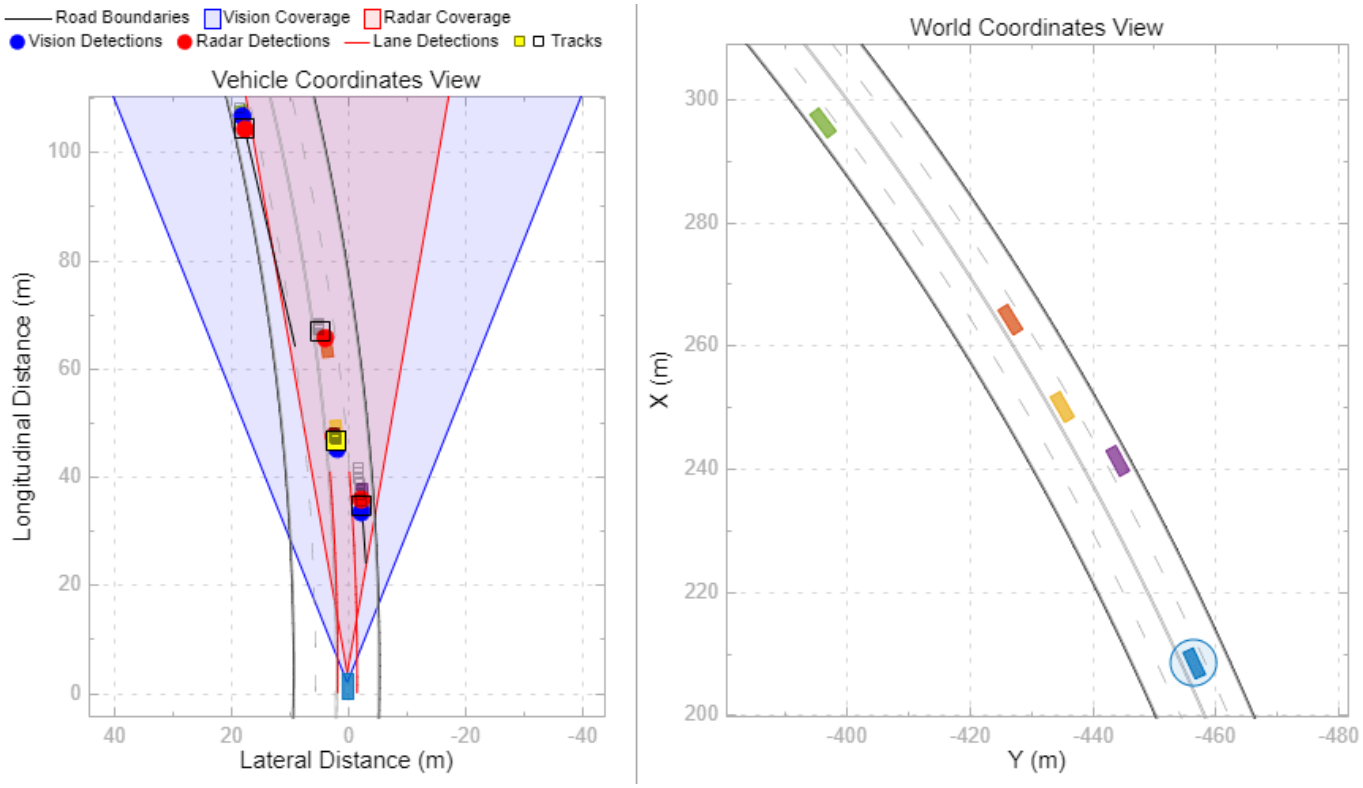


To plot the results of the simulation and depict the ego vehicle surroundings and tracked objects, use the Bird's-Eye Scope (Automated Driving Toolbox). The Bird's-Eye Scope is a model-level visualization tool that you can open from the Simulink toolstrip. On the **Simulation** tab, under **Review Results**, click **Bird's-Eye Scope**. After opening the scope, set up the signals by clicking **Find Signals**.

To get a mid-simulation view, simulate the model for `10` seconds.

```
sim('LaneFollowingTestBenchExample','StopTime','10')
```

After simulating the model for 10 seconds, open the Bird's-Eye Scope. In the scope toolstrip, to display the World Coordinates View of the scenario, click **World Coordinates**. In this view, the ego vehicle is circled. To display the legend for the Vehicle Coordinates View, click **Legend**.

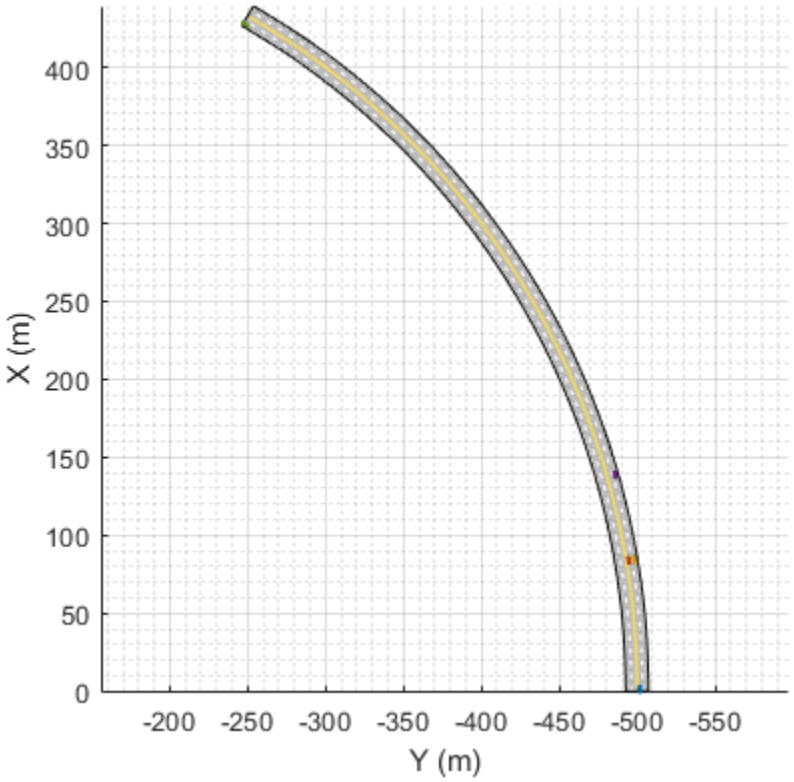


The Bird's-Eye Scope shows the results of the sensor fusion. It shows how the radar and vision sensors detect the vehicles within their coverage areas. It also shows the tracks maintained by the Multi-Object Tracker (Automated Driving Toolbox) block. The yellow track shows the most important object (MIO), which is the closest track in front of the ego vehicle in its lane. The ideal lane markings are also shown along with the synthetically detected left and right lane boundaries (shown in red).

Simulate the model to the end of the scenario.

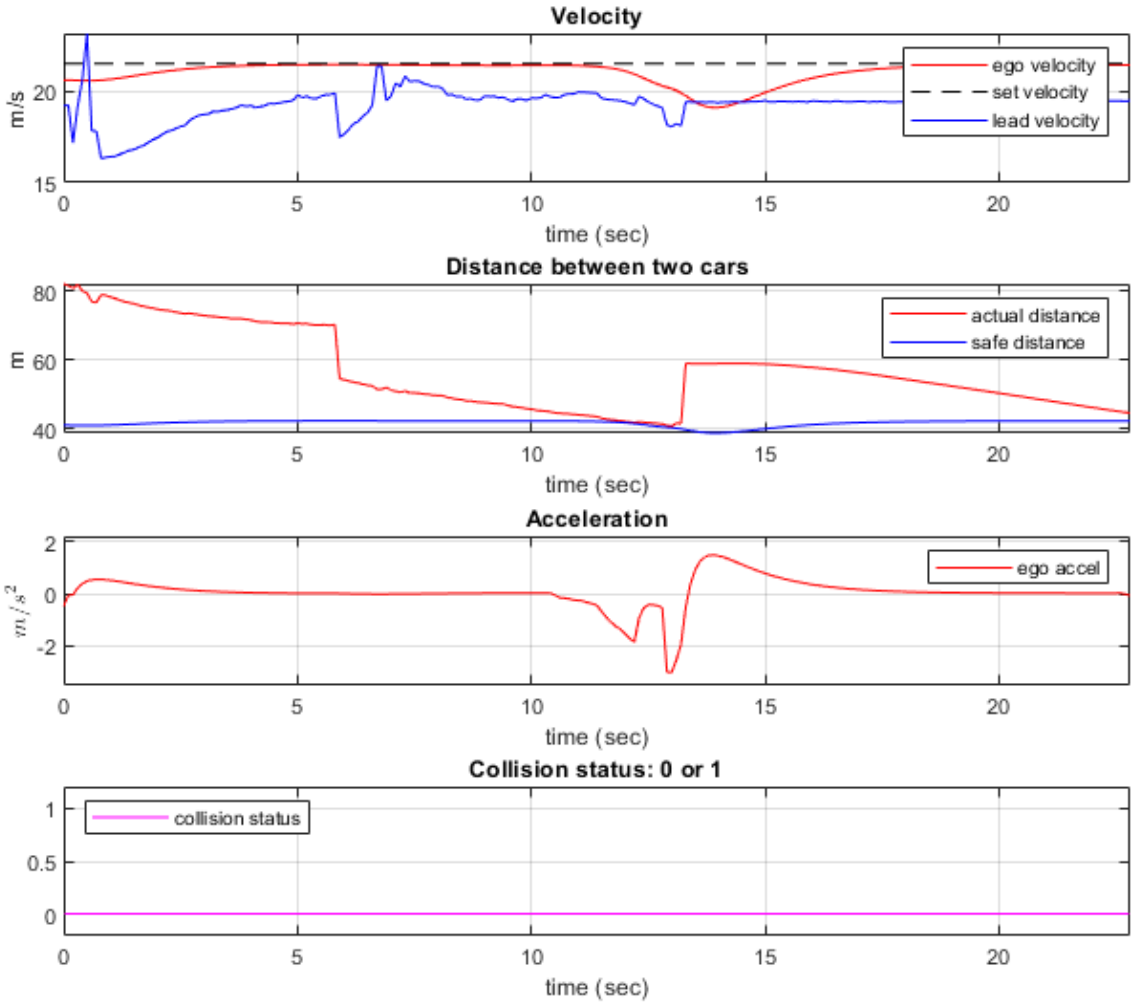
```
sim('LaneFollowingTestBenchExample')
```

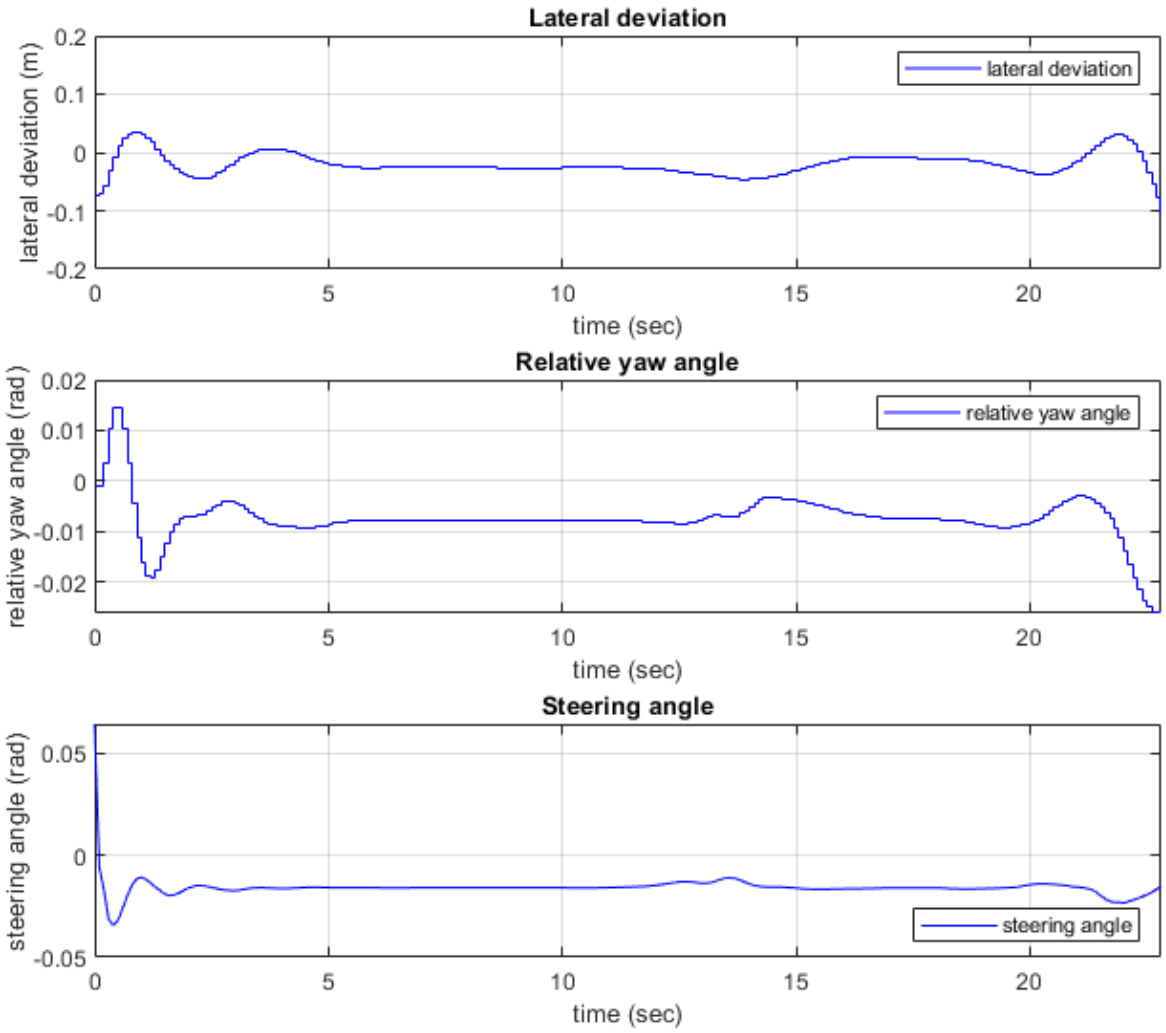
```
   Assuming no disturbance added to measured output channel #3.
-->Assuming output disturbance added to measured output channel #2 is integrated white noise.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #4 is integrated white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on each measured
```



Plot the controller performance.

```
plotLFResults(logsout,time_gap,default_spacing)
```


Velocity
m/s
20
15
0
5
10
15
20
time (sec)
ego velocity
set velocity
lead velocity
Distance between two cars
m
80
60
40
actual distance
safe distance
Acceleration
m/s^2
2
0
-2
ego accel
Collision status: 0 or 1
1
0.5
0
collision status



The first figure shows the following spacing control performance results.

- The **Velocity plot** shows that the ego vehicle maintains velocity control from 0 to 11 seconds, switches to spacing control from 11 to 16 seconds, then switches back to velocity control.
- The **Distance between two cars** plot shows that the actual distance between lead vehicle and ego vehicle is always greater than the safe distance.
- The **Acceleration** plot shows that the acceleration for ego vehicle is smooth.
- The **Collision status** plot shows that no collision between lead vehicle and ego vehicle is detected, thus the ego vehicle runs in a safe mode.

The second figure shows the following lateral control performance results.

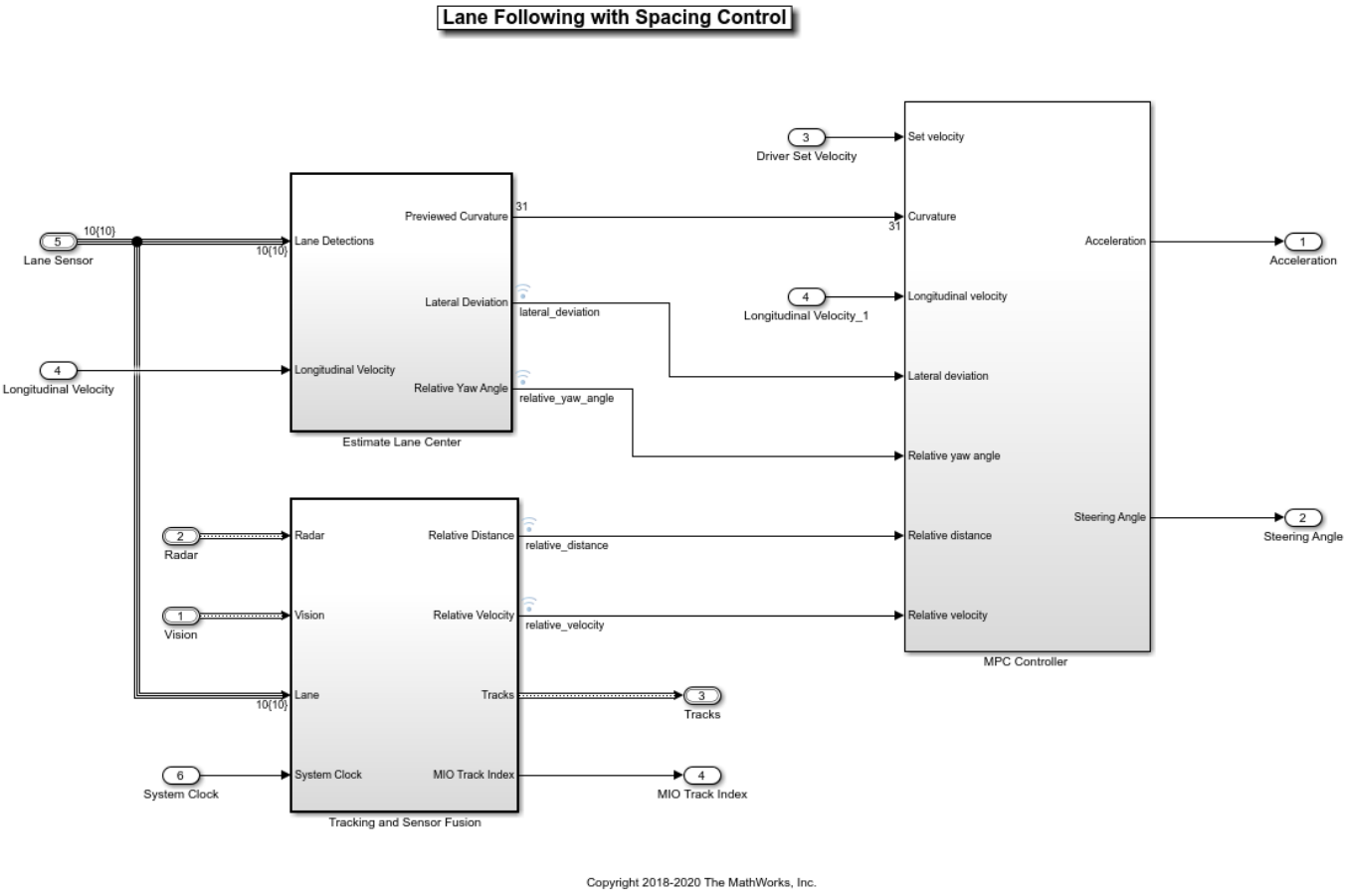
- The **Lateral deviation** plot shows that the distance to the lane centerline is within 0.2 m.
- The **Relative yaw angle** plot shows that the yaw angle error with respect to lane centerline is within 0.03 rad (less than 2 degrees).

- The **Steering angle** plot shows that the steering angle for ego vehicle is smooth.

Explore Lane Following Controller

The Lane Following Controller subsystem contains three main parts: 1) Estimate Lane Center 2) Tracking and Sensor Fusion 3) MPC Controller

```
open_system('LaneFollowingTestBenchExample/Lane Following Controller')
```

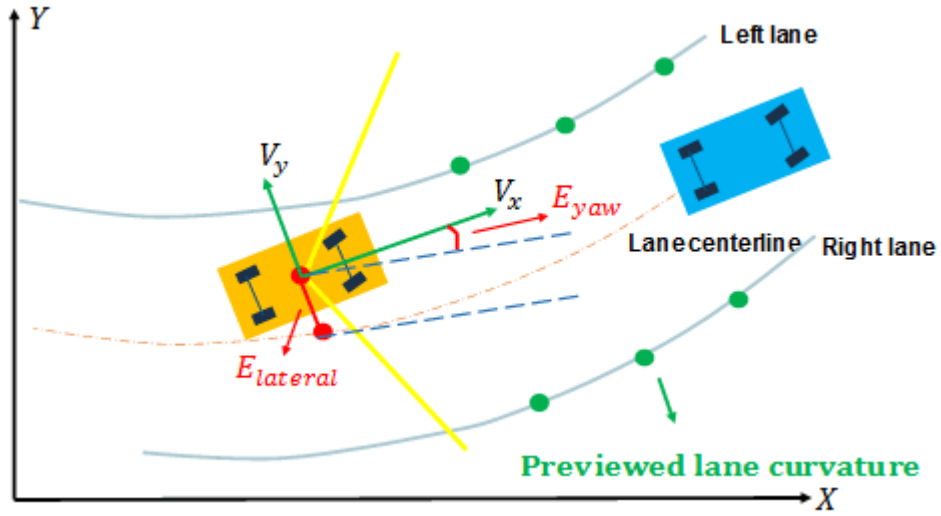


The Estimate Lane Center subsystem outputs the lane sensor data to the MPC controller. The previewed curvature provides the centerline of lane curvature ahead of the ego vehicle. In this example, the ego vehicle can look ahead for 3 seconds, which is the product of the prediction horizon and the controller sample time. The controller uses previewed information for calculating the ego vehicle steering angle, which improves the MPC controller performance. The lateral deviation measures the distance between the ego vehicle and the centerline of the lane. The relative yaw angle measures the yaw angle difference between the ego vehicle and the road. The ISO 8855 to SAE J670E block inside the subsystem converts the coordinates from Lane Detections, which use ISO 8855, to the MPC Controller which uses SAE J670E.

The Tracking and Sensor Fusion subsystem processes vision and radar detections coming from the Vehicle and Environment subsystem and generates a comprehensive situation picture of the environment around the ego vehicle. Also, it provides the lane following controller with an estimate of the closest vehicle in the lane in front of the ego vehicle.

The goals for the MPC Controller block are to:

- Maintain the driver-set velocity and keep a safe distance from lead vehicle. This goal is achieved by controlling the longitudinal acceleration.
- Keep the ego vehicle in the middle of the lane; that is reduce the lateral deviation $E_{lateral}$ and the relative yaw angle E_{yaw}, by controlling the steering angle.
- Slow down the ego vehicle when road is curvy. To achieve this goal, the MPC controller has larger penalty weights on lateral deviation than on longitudinal speed.



The MPC controller is designed within the Path Following Control (PFC) System block based on the entered mask parameters, and the designed MPC Controller is an adaptive MPC which updates the vehicle model at run time. The lane following controller calculates the longitudinal acceleration and steering angle for the ego vehicle based on the following inputs:

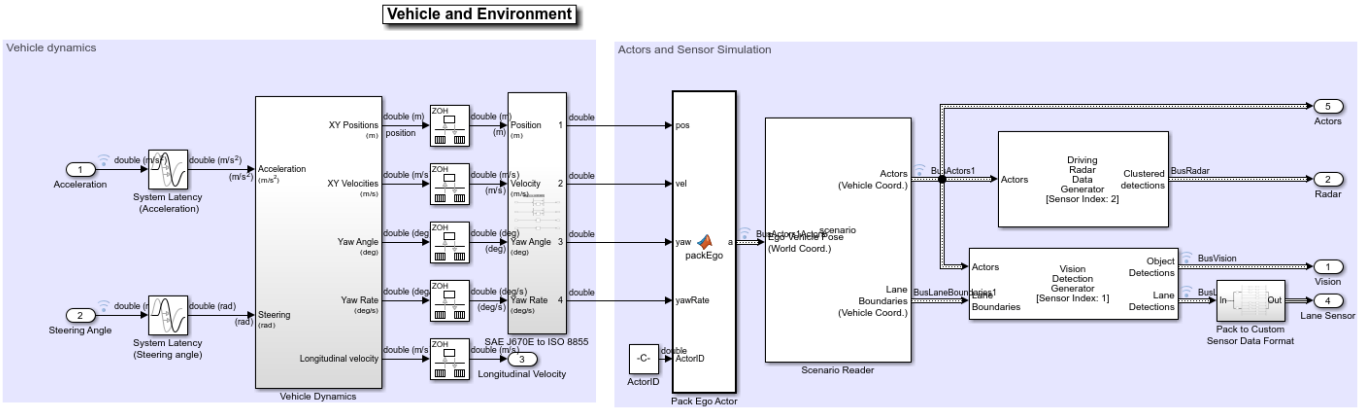
- Driver-set velocity
- Ego vehicle longitudinal velocity
- Previewed curvature (derived from Lane Detections)
- Lateral deviation (derived from Lane Detections)
- Relative yaw angle (derived from Lane Detections)
- Relative distance between lead vehicle and ego vehicle (from the Tracking and Sensor Fusion system)
- Relative velocity between lead vehicle and ego vehicle (from the Tracking and Sensor Fusion system)

Considering the physical limitations of the ego vehicle, the steering angle is constrained to be within [-0.26,0.26] rad, and the longitudinal acceleration is constrained to be within [-3,2] m/s^2.

Explore Vehicle and Environment

The Vehicle and Environment subsystem enables closed-loop simulation of the lane following controller.

```
open_system('LaneFollowingTestBenchExample/Vehicle and Environment')
```



The System Latency blocks model the latency in the system between model inputs and outputs. The latency can be caused by sensor delay or communication delay. In this example, the latency is approximated by one sample time $T_s = 0.1$ seconds.

The Vehicle Dynamics subsystem models the vehicle dynamics using a Bicycle Model - Force Input block from the Vehicle Dynamics Blockset™. The lower-level dynamics are modeled by a first-order linear system with a time constant of $\tau = 0.5$ seconds.

The SAE J670E to ISO 8855 subsystem converts the coordinates from Vehicle Dynamics, which uses SAE J670E, to Scenario Reader, which uses ISO 8855.

The Scenario Reader (Automated Driving Toolbox) block reads the actor poses data from the base workspace scenario variable. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. The Scenario Reader block also generates the ideal left and right lane boundaries based on the position of the vehicle with respect to the scenario used in `helperLFSetUp`.

The Vision Detection Generator (Automated Driving Toolbox) block takes the ideal lane boundaries from the Scenario Reader block. The detection generator models the field of view of a monocular camera and determines the heading angle, curvature, curvature derivative, and valid length of each road boundary, accounting for any other obstacles. The Driving Radar Data Generator (Automated Driving Toolbox) block generates clustered detections from the ground-truth data present in the field-of-view of the radar based on the radar cross-section defined in the scenario.

Run Controller for Multiple Test Scenarios

This example uses multiple test scenarios based on ISO standards and real-world scenarios. To verify the controller performance, you can test the controller for multiple scenarios and tune the controller parameters if the performance is not satisfactory. To do so:

1. Select the scenario by changing the scenario name input to `helperLFSetUp`.
2. Configure the simulation parameters by running `helperLFSetUp`.
3. Simulate the model with the selected scenario.
4. Evaluate the controller performance using `plotLFResults`
5. Tune the controller parameters if the performance is not satisfactory.

You can automate the verification and validation of the controller using Simulink Test™.

Generate Code for the Control Algorithm

The `LFRefMdl` model supports generating C code using Embedded Coder® software. To check if you have access to Embedded Coder, run:

```
hasEmbeddedCoderLicense = license('checkout','RTW_Embedded_Coder')
```

You can generate a C function for the model and explore the code generation report by running:

```
if hasEmbeddedCoderLicense
  rtwbuild('LFRefMdl')
end
```

You can verify that the compiled C code behaves as expected using software-in-the-loop (SIL) simulation. To simulate the `LFRefMdl` referenced model in SIL mode, use:

```
if hasEmbeddedCoderLicense
  set_param('LaneFollowingTestBenchExample/Lane Following Controller',...
            'SimulationMode','Software-in-the-loop (SIL)')
end
```

When you run the `LaneFollowingTestBenchExample` model, code is generated, compiled, and executed for the `LFRefMdl` model, which enables you to test the behavior of the compiled code through simulation.

Conclusions

This example shows how to implement an integrated lane following controller on a curved road with sensor fusion and lane detection, test it in Simulink using synthetic data generated using Automated Driving Toolbox software, componentize it, and automatically generate code for it.

```
close all
bdclose all
```

See Also

Blocks
Lane Keeping Assist System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Highway Lane Following” on page 11-62

Highway Lane Following

This example shows how to simulate a highway lane following application with vision processing, sensor fusion, and controller components. These components are tested in a 3D simulation environment that includes camera and radar sensor models.

Introduction

A highway lane following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance to a preceding vehicle in the same lane. The system typically uses vision processing algorithms to detect lanes and vehicles from a camera. The vehicle detections from the camera are then fused with detections from a radar to improve the robustness of perception. The controller uses the lane detections, vehicle detections, and set speed to control steering and acceleration.

This example demonstrates how to create a test bench model to test vision processing, sensor fusion, and controls in a 3D simulation environment. The test bench model can be configured for different scenarios to test the ability to follow lanes and avoid collisions with other vehicles. In this example, you:

1. **Partition the algorithm and test bench** — The model is partitioned into lane following algorithm models and a test bench model. The algorithm models implement the individual components. The test bench includes the integration of the algorithm models, and virtual testing framework.
2. **Explore the test bench model** — The test bench model contains the testing framework, which includes the scenario, ego vehicle dynamics model, and metrics assessment using ground truth. A cuboid scenario defines vehicle trajectories and specifies the ground truth. An equivalent Unreal Engine® scene is used to model detections from a radar sensor and images from a monocular camera sensor. A bicycle model is used to model the ego vehicle.
3. **Explore the algorithm models** — Algorithm models are reference models that implement vision processing, sensor fusion, decision logic, and controls components to build the lane following application.
4. **Visualize a test scenario** — The scenario contains a curved road with multiple vehicles.
5. **Simulate the test bench model** — The model is simulated to test integration of the vision processing, sensor fusion, and controls components.
6. **Explore additional scenarios** — These scenarios test the system under additional conditions.

Testing the integration of the controller and the perception algorithm requires a photorealistic simulation environment. In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is supported only on Microsoft', char(174), ' Windows', char(174), '.'])
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0)
```

Partition Algorithm and Test Bench

The model is partitioned into separate algorithm and test bench models.

- Algorithm models — Algorithm models are reference models that implement the functionality of individual components.
- Test bench model — The Highway Lane Following test bench specifies the stimulus and environment to test the algorithm models.

Explore Test Bench Model

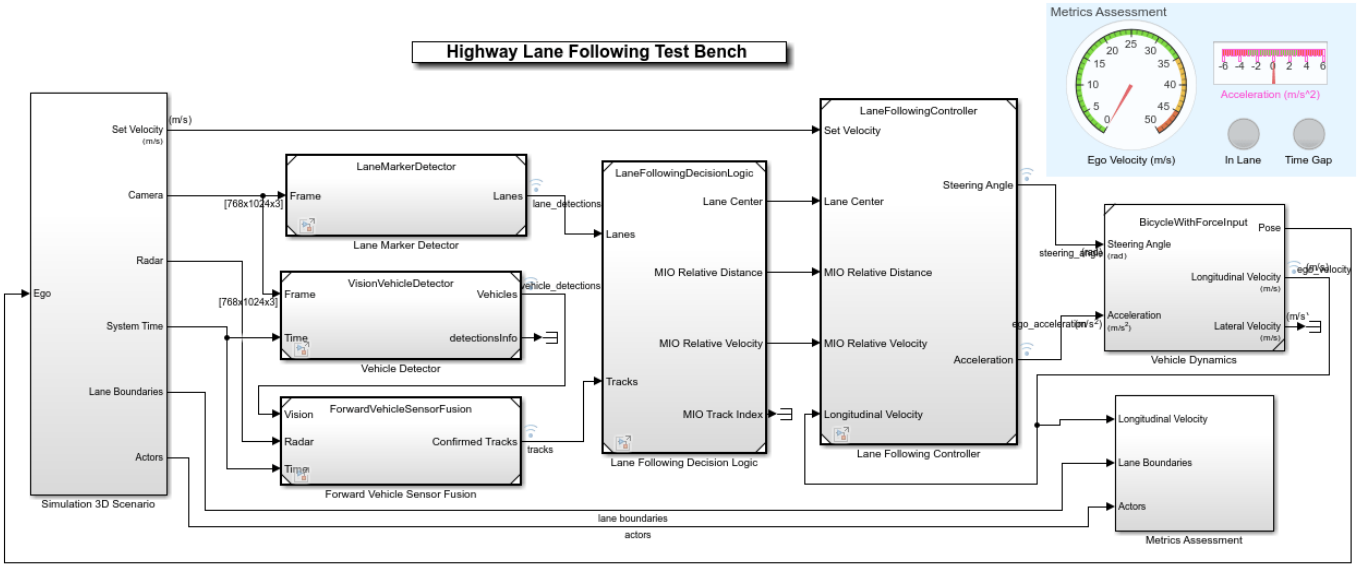
In this example, you use a system-level simulation test bench model to explore the behavior of the control and vision processing algorithms for the lane following system.

To explore the test bench model, open a working copy of the project example files. MATLAB® copies the files to an example folder so that you can edit them.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('HighwayLaneFollowing.zip', 'workDir', pwd);
```

Open the system-level simulation test bench model.

```
open_system("HighwayLaneFollowingTestBench")
```



Copyright 2019-2021 The MathWorks, Inc.

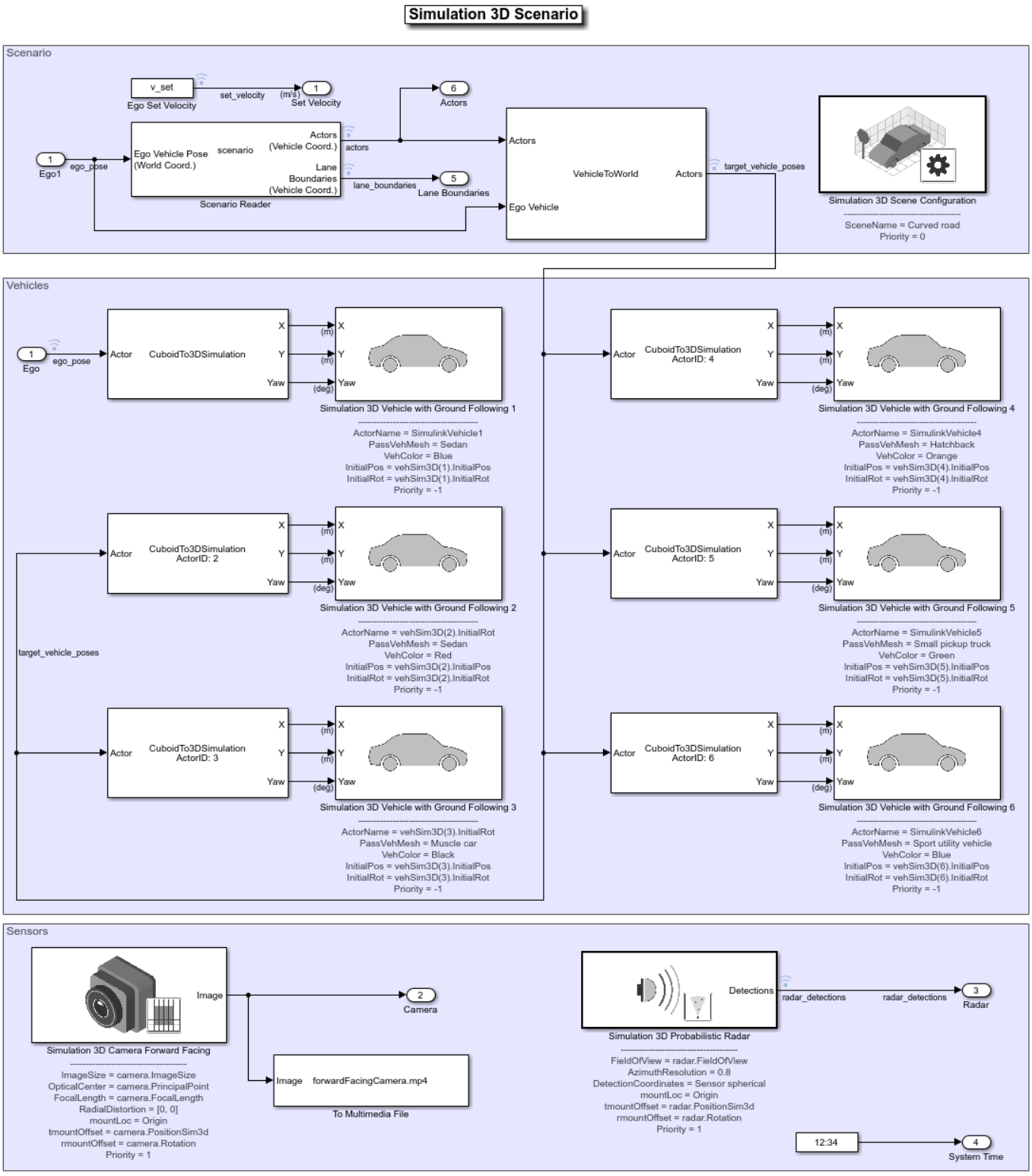
The test bench model contains these modules:

- Simulation 3D Scenario — Subsystem that specifies the road, vehicles, camera sensor, and radar sensor used for simulation.
- Lane Marker Detector — Algorithm model to detect the lane boundaries in the frame captured by the camera sensor.
- Vehicle Detector — Algorithm model to detect vehicles in the frame captured by the camera sensor.
- Forward Vehicle Sensor Fusion — Algorithm model to fuse vehicle detections from the camera and radar sensors.

- Lane Following Decision Logic — Algorithm model to specify the lateral and longitudinal decision logic that provides information related to the most important object (MIO) and lane center to the controller.
- Lane Following Controller — Algorithm model that specifies the steering angle and acceleration controls.
- Vehicle Dynamics — Subsystem that specifies the dynamic model of the ego vehicle.
- Metrics Assessment — Subsystem that assesses system-level behavior.

The Simulation 3D Scenario subsystem configures the road network, positions vehicles, and synthesizes sensors. Open the Simulation 3D Scenario subsystem.

```
open_system("HighwayLaneFollowingTestBench/Simulation 3D Scenario")
```



The scene and road network are specified by these parts of the subsystem:

- The Simulation 3D Scene Configuration (Automated Driving Toolbox) block has the **SceneName** parameter set to `Curved road`.
- The Scenario Reader (Automated Driving Toolbox) block is configured to use a driving scenario that contains a road network that closely matches a section of the road network from the `Curved road` scene.

The vehicle positions are specified by these parts of the subsystem:

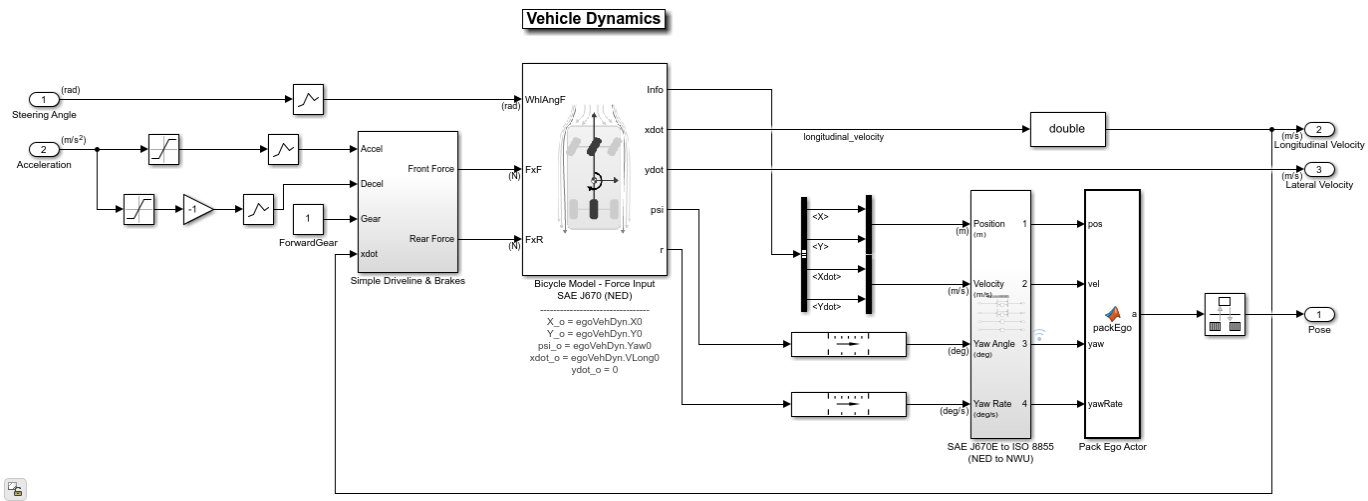
- The Ego input port controls the position of the ego vehicle, which is specified by the Simulation 3D Vehicle with Ground Following 1 block.
- The Vehicle To World (Automated Driving Toolbox) block converts actor poses from the coordinates of the ego vehicle to the world coordinates.
- The Scenario Reader (Automated Driving Toolbox) block outputs actor poses, which control the position of the target vehicles. These vehicles are specified by the other Simulation 3D Vehicle with Ground Following (Automated Driving Toolbox) blocks.
- The Cuboid To 3D Simulation (Automated Driving Toolbox) block converts the ego pose coordinate system (with respect to below the center of the vehicle rear axle) to the 3D simulation coordinate system (with respect to below the vehicle center).

The sensors attached to the ego vehicle are specified by these parts of the subsystem:

- The Simulation 3D Camera (Automated Driving Toolbox) block is attached to the ego vehicle to capture its front view. The output image from this block is processed by the Lane Marker Detector block to detect the lanes and Vehicle Detector block to detect the vehicles.
- The Simulation 3D Probabilistic Radar Configuration (Automated Driving Toolbox) block is attached to the ego vehicle to detect vehicles in 3D simulation environment.
- The Measurement Bias Center to Rear Axle block converts the coordinate system of the Simulation 3D Probabilistic Radar Configuration (Automated Driving Toolbox) block (with respect to below the vehicle center) to the ego pose coordinates (with respect to below the center of the vehicle rear axle).

The Vehicle Dynamics subsystem uses a Bicycle Model block to model the ego vehicle. Open the Vehicle Dynamics subsystem.

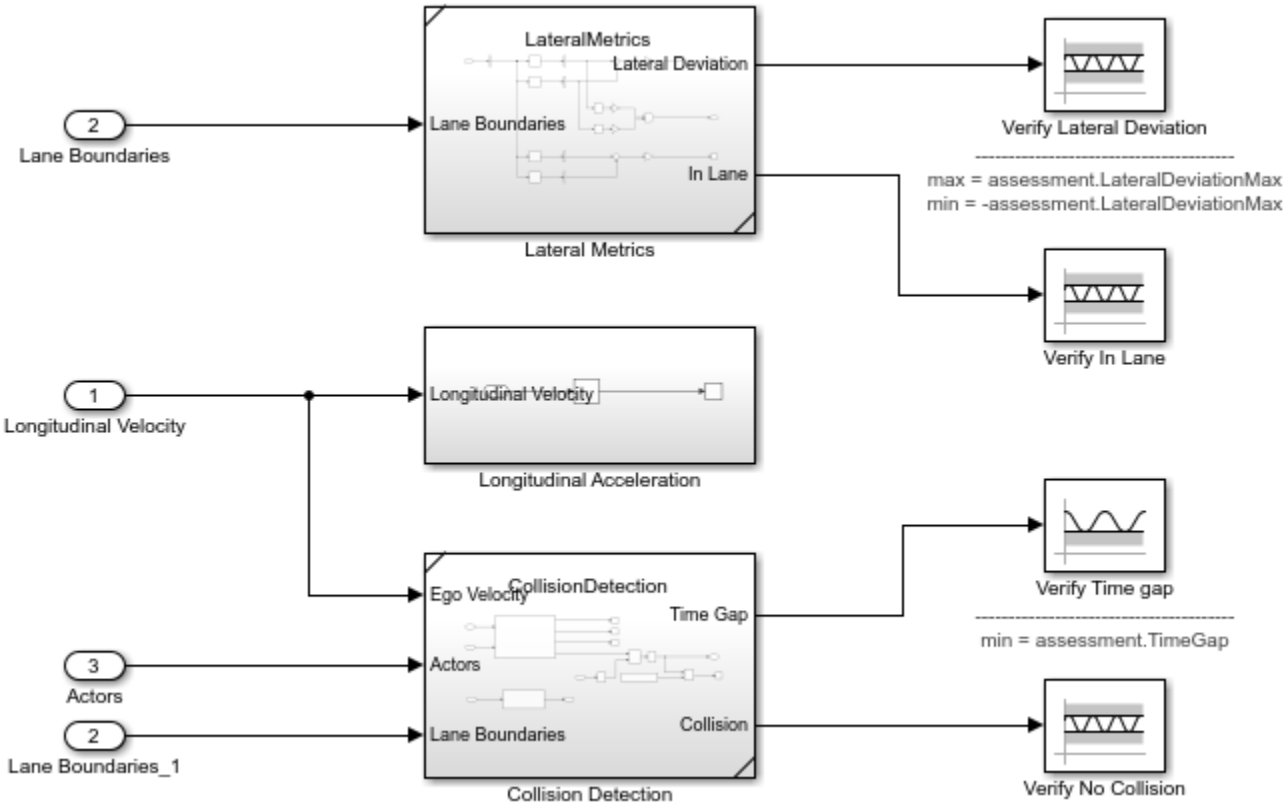
```
open_system("HighwayLaneFollowingTestBench/Vehicle Dynamics");
```



The Bicycle Model block implements a rigid two-axle single track vehicle body model to calculate longitudinal, lateral, and yaw motion. The block accounts for body mass, aerodynamic drag, and weight distribution between the axles due to acceleration and steering. For more details, see Bicycle Model (Automated Driving Toolbox).

The Metric Assessment subsystem enables system-level metric evaluations using the ground truth information from the scenario. Open the Metrics Assessment subsystem.

```
open_system("HighwayLaneFollowingTestBench/Metrics Assessment");
```



In this example, four metrics are used to assess the lane-following system.

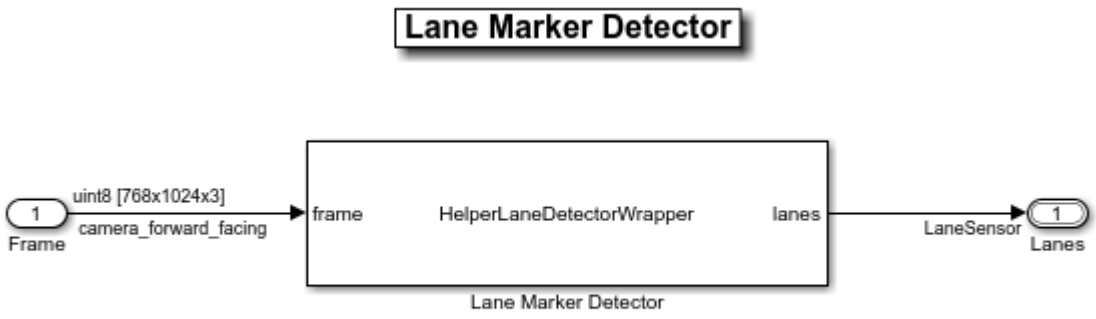
- Verify Lateral Deviation — This block verifies that the lateral deviation from the center line of the lane is within prescribed thresholds for the corresponding scenario. Define the thresholds when you author the test scenario.
- Verify In Lane — This block verifies that the ego vehicle is following one of the lanes on the road throughout the simulation.
- Verify Time gap — This block verifies that the time gap between the ego vehicle and the lead vehicle is more than 0.8 seconds. The time gap between the two vehicles is defined as the ratio of the calculated headway distance to the ego vehicle velocity.
- Verify No Collision — This block verifies that the ego vehicle does not collide with the lead vehicle at any point during the simulation. For more details on how to integrate these metrics with Simulink Test™ for enabling automatic regression testing, see “Automate Testing for Highway Lane Following” (Automated Driving Toolbox).

Explore Algorithm Models

The lane following system is developed by integrating the lane marker detector, vehicle detector, forward vehicle sensor fusion, lane following decision logic, and lane following controller components.

The lane marker detector algorithm model implements a perception module to analyze the images of roads. Open the Lane Marker Detector algorithm model.

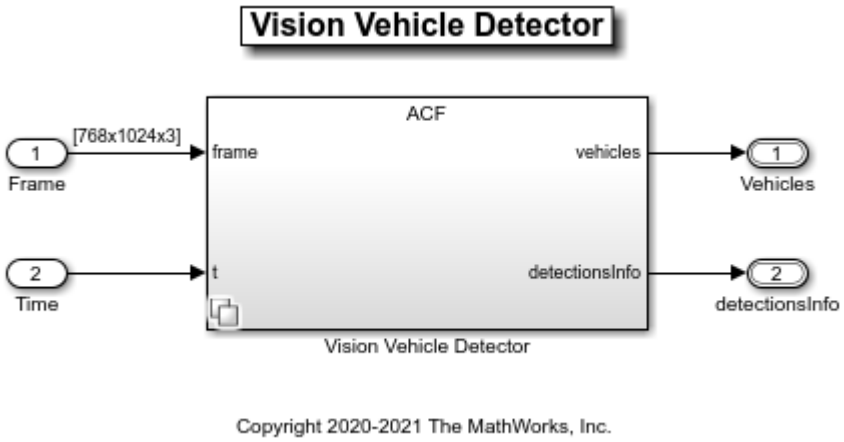
```
open_system("LaneMarkerDetector");
```



The lane marker detector takes the frame captured by a monocular camera sensor as input. It also takes in the camera intrinsic parameters through the mask. It detects the lane boundaries and outputs the lane information and marking type of each lane through the `LaneSensor` bus. For more details on how to design and evaluate a lane marker detector, see “Design Lane Marker Detector Using Unreal Engine Simulation Environment” (Automated Driving Toolbox) and “Generate Code for Lane Marker Detector” (Automated Driving Toolbox).

The vehicle detector algorithm model detects vehicles in the driving scenario. Open the Vehicle Detector algorithm model.

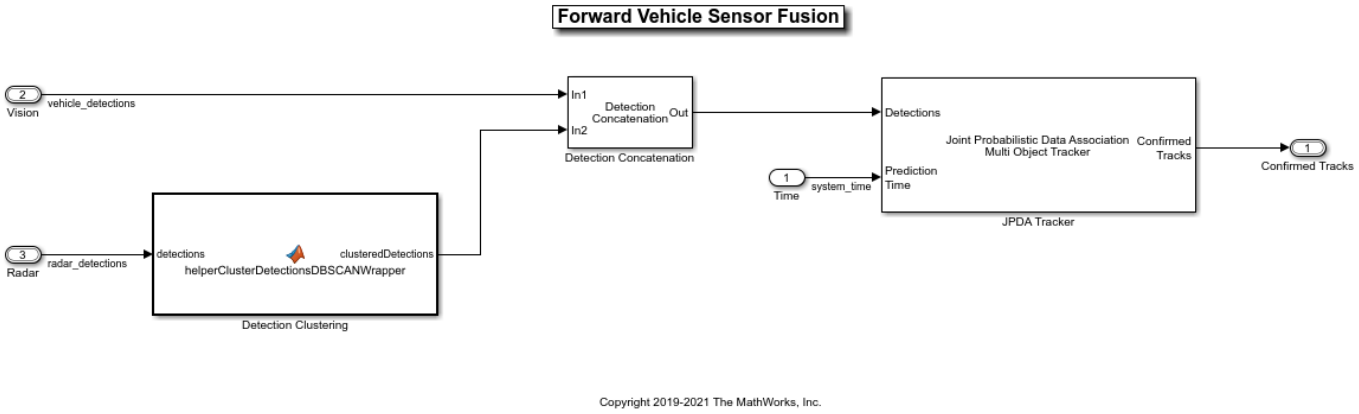
```
open_system("VisionVehicleDetector");
```



The vehicle detector takes the frame captured by a camera sensor as input. It also takes in the camera intrinsic parameters through the mask. It detects the vehicles and outputs the vehicle information as bounding boxes. For more details on how to design and evaluate a vehicle detector, see “Generate Code for Vision Vehicle Detector” (Automated Driving Toolbox).

The forward vehicle sensor fusion component fuses vehicle detections the from camera and radar sensors and tracks the detected vehicles using the central level tracking method. Open the Forward Vehicle Sensor Fusion algorithm model.

```
open_system("ForwardVehicleSensorFusion");
```

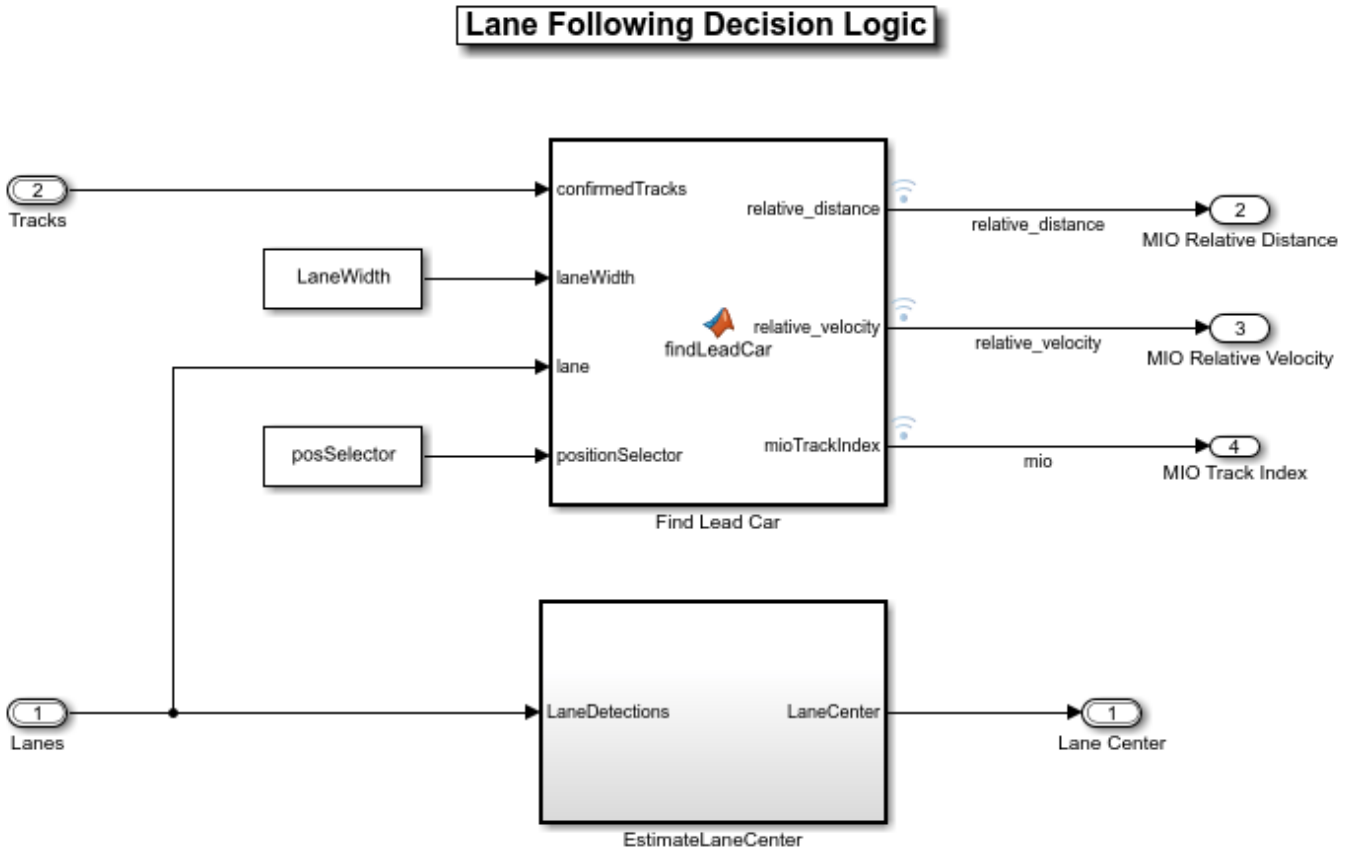


Copyright 2019-2021 The MathWorks, Inc.

The forward vehicle sensor fusion model takes in the vehicle detections from vision and radar sensors as inputs. The radar detections are clustered and then concatenated with vision detections. The concatenated vehicle detections are then tracked using a joint probabilistic data association tracker. This component outputs the confirmed tracks. For more details on forward vehicle sensor fusion, see “Forward Vehicle Sensor Fusion” (Automated Driving Toolbox).

The lane following decision logic algorithm model specifies lateral and longitudinal decisions based on the detected lanes and tracks. Open the Lane Following Decision Logic algorithm model.

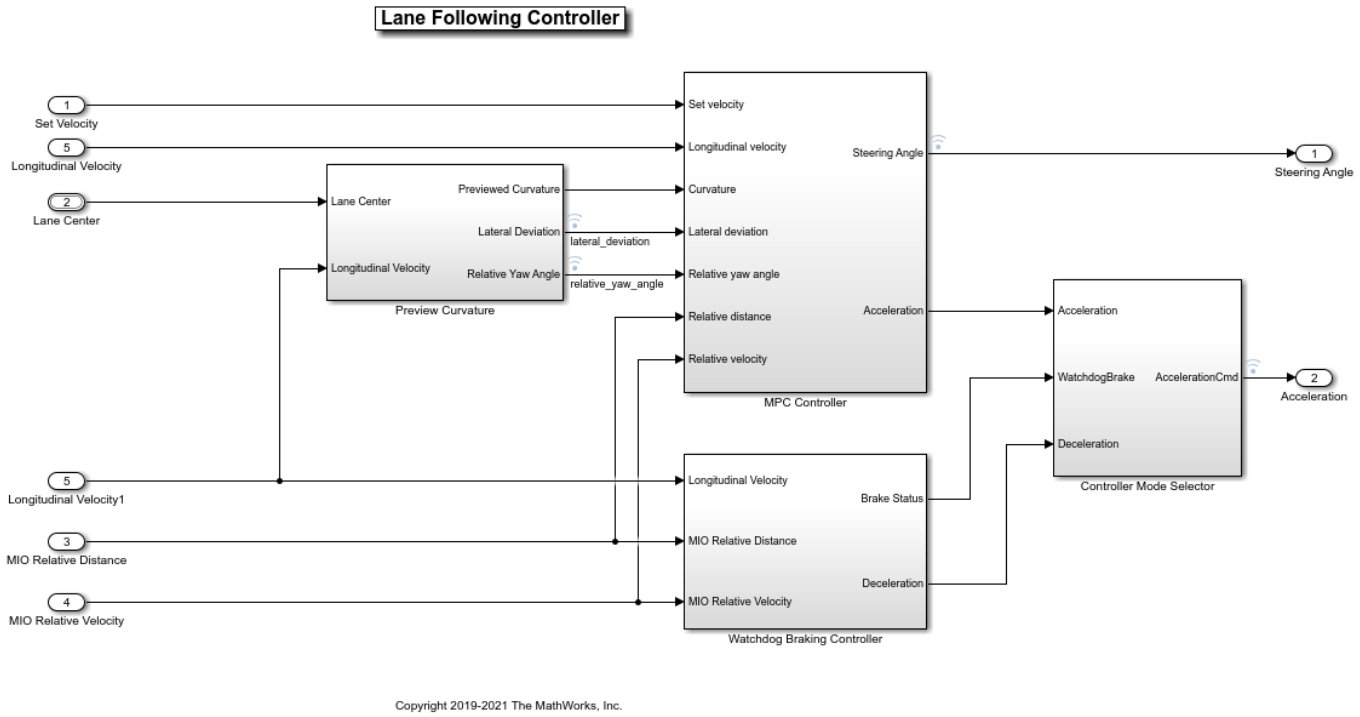
```
open_system("LaneFollowingDecisionLogic");
```



The lane following decision logic model takes the detected lanes from the lane marker detector and the confirmed tracks from the forward vehicle sensor fusion module as inputs. It estimates the lane center and also determines the MIO lead car traveling in the same lane as the ego vehicle. It outputs the relative distance and relative velocity between the MIO and ego vehicle.

The lane following controller specifies the longitudinal and lateral controls. Open the Lane Following Controller algorithm model.

```
open_system("LaneFollowingController");
```



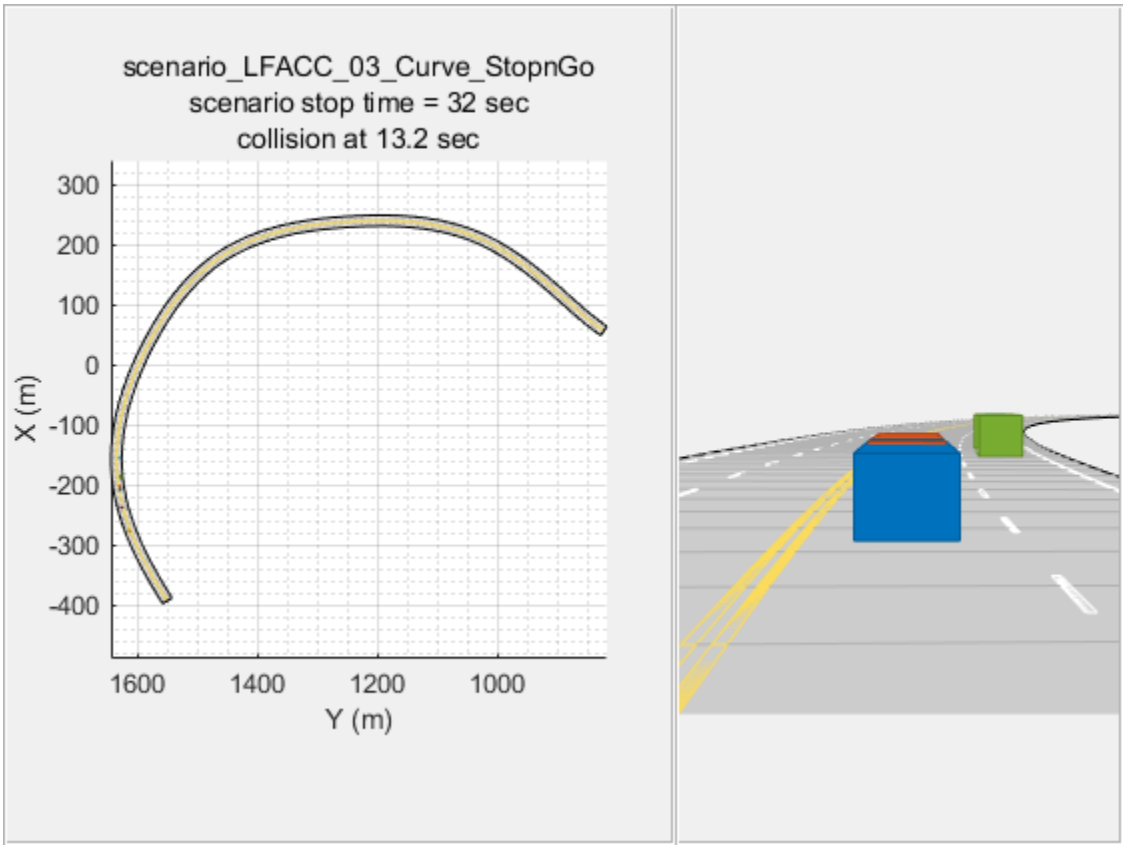
The controller takes the set velocity, lane center, and MIO information as inputs. It uses a path following controller to control the steering angle and acceleration for the ego vehicle. It also uses a watchdog braking controller to apply brakes as a fail-safe mode. The controller outputs the steering angle and acceleration command that determines whether to accelerate, decelerate, or apply brakes. The Vehicle Dynamics block uses these outputs for lateral and longitudinal control of the ego vehicle.

Visualize Test Scenario

The helper function `scenario_LFACC_03_Curve_StopnGo` generates a cuboid scenario that is compatible with the `HighwayLaneFollowingTestBench` model. This is an open-loop scenario containing multiple target vehicles on a curved road. The road centers, lane markings, and vehicles in this cuboid scenario closely match a section of the curved road scene provided with the 3D simulation environment. In this scenario, a lead vehicle slows down in front of the ego vehicle while other vehicles travel in adjacent lanes.

Plot the open-loop scenario to see the interactions of the ego vehicle and target vehicles.

```
hFigScenario = helperPlotLFScenario("scenario_LFACC_03_Curve_StopnGo");
```



The ego vehicle is not under closed-loop control, so a collision occurs with the slower moving lead vehicle. The goal of the closed-loop system is to follow the lane and maintain a safe distance from the lead vehicles. In the `HighwayLaneFollowingTestBench` model, the ego vehicle has the same initial velocity and initial position as in the open-loop scenario.

Close the figure.

```
close(hFigScenario)
```

Simulate the test bench model

Configure and test the integration of the algorithms in the 3D simulation environment. To reduce command-window output, turn off the MPC update messages.

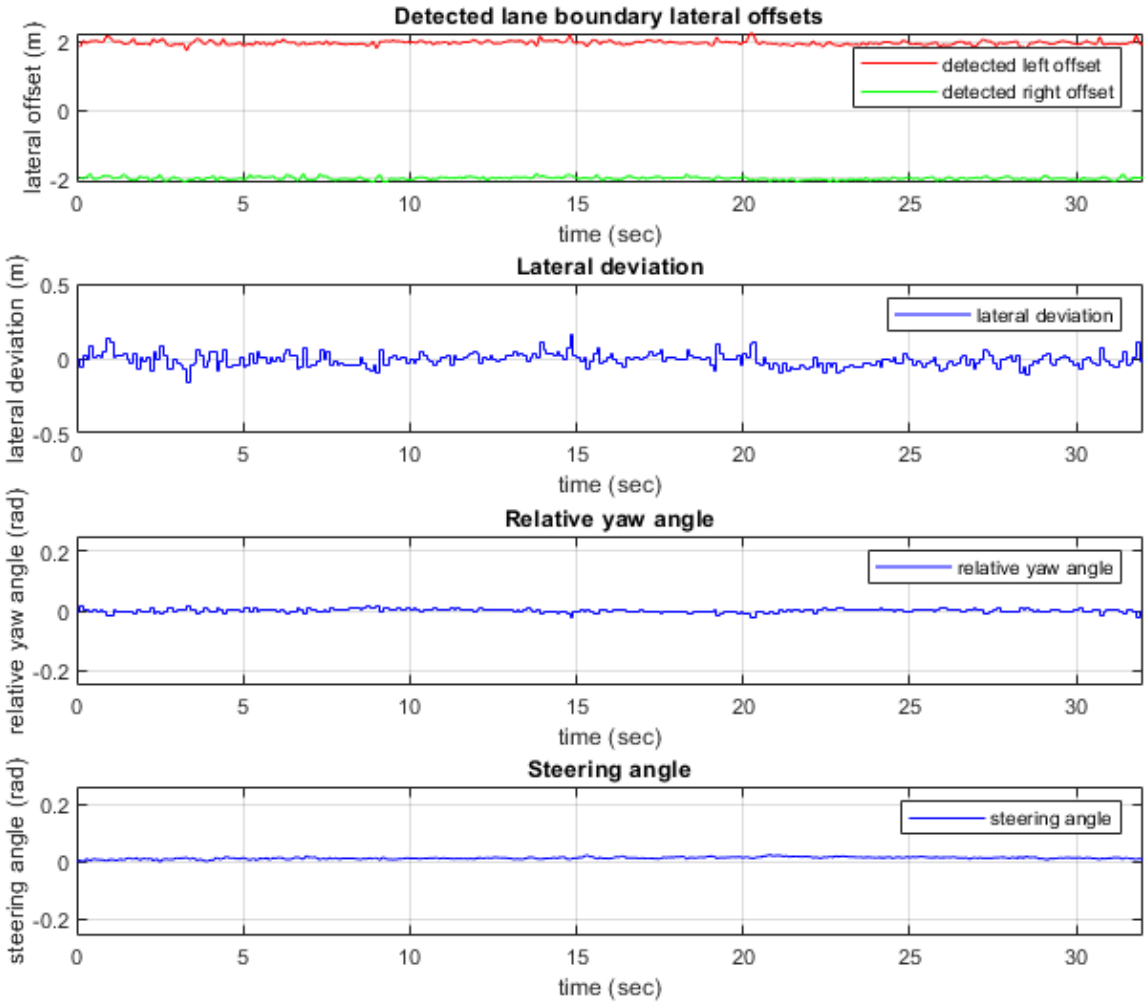
```
mpcverbosity('off');
```

Configure the test bench model to use the same scenario.

```
helperSLHighwayLaneFollowingSetup("scenarioFcnName",...
    "scenario_LFACC_03_Curve_StopnGo");
sim("HighwayLaneFollowingTestBench")
```

Plot the lateral controller performance results.

```
hFigLatResults = helperPlotLFLateralResults(logsout);
```



Close the figure.

```
close(hFigLatResults)
```

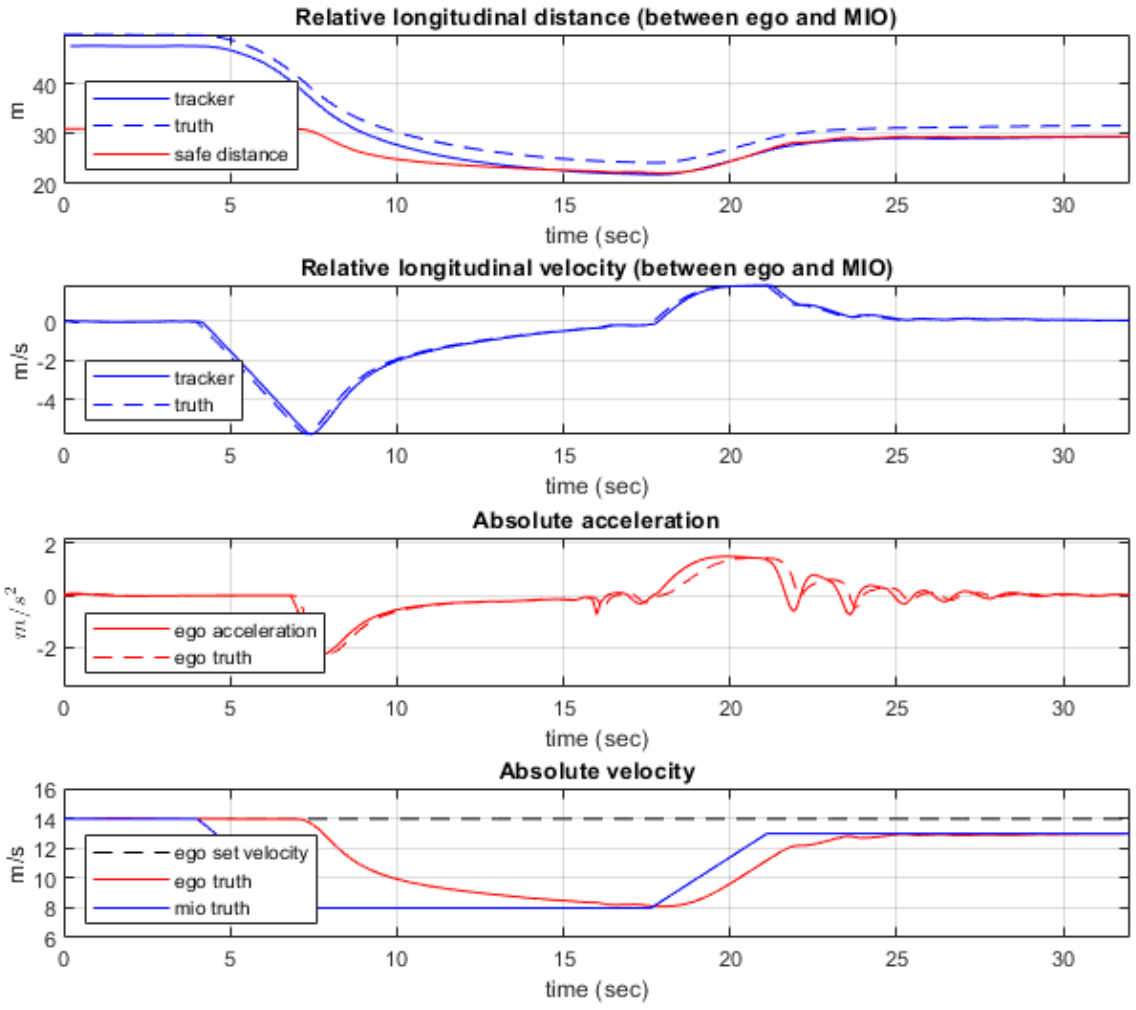
Examine the simulation results.

- The **Detected lane boundary lateral offsets** plot shows the lateral offsets of the detected left-lane and right-lane boundaries from the centerline of the lane. The detected values are close to the ground truth of the lane but deviate by small quantities.
- The **Lateral deviation** plot shows the lateral deviation of the ego vehicle from the centerline of the lane. Ideally, lateral deviation is zero meters, which implies that the ego vehicle exactly follows the centerline. Small deviations occur when the vehicle is changing velocity to avoid collision with another vehicle.
- The **Relative yaw angle** plot shows the relative yaw angle between ego vehicle and the centerline of the lane. The relative yaw angle is very close to zero radian, which implies that the heading angle of the ego vehicle matches the yaw angle of the centerline closely.

- The **Steering angle** plot shows the steering angle of the ego vehicle. The steering angle trajectory is smooth.

Plot the longitudinal controller performance results.

```
hFigLongResults = helperPlotLFLongitudinalResults(logsout,time_gap,...
    default_spacing);
```



Close the figure.

```
close(hFigLongResults)
```

Examine the simulation results.

- The **Relative longitudinal distance** plot shows the distance between the ego vehicle and the MIO. In this case, the ego vehicle approaches the MIO and gets close to it or exceeds the safe distance in some cases.

- The **Relative longitudinal velocity** plot shows the relative velocity between the ego vehicle and the MIO. In this example, the vehicle detector only detects positions, so the tracker in the control algorithm estimates the velocity. The estimated velocity lags the actual (ground truth) MIO relative velocity.
- The **Absolute acceleration** plot shows that the controller commands the vehicle to decelerate when it gets too close to the MIO.
- The **Absolute velocity** plot shows the ego vehicle initially follows the set velocity, but when the MIO slows down, to avoid a collision, the ego vehicle also slows down.

During simulation, the model logs signals to the base workspace as `logsout` and records the output of the camera sensor to `forwardFacingCamera.mp4`. You can use the `helperPlotLFDetectionResults` function to visualize the simulated detections similar to how recorded data is explored in the “Forward Collision Warning Using Sensor Fusion” (Automated Driving Toolbox) example. You can also record the visualized detections to a video file to enable review by others who do not have access to MATLAB.

Plot the detection results from logged data, generate a video, and open the Video Viewer (Image Processing Toolbox) app.

```
hVideoViewer = helperPlotLFDetectionResults(...
    logsout, "forwardFacingCamera.mp4" , scenario, camera, radar,...
    scenarioFcnName,...
    "RecordVideo", true,...
    "RecordVideoFileName", scenarioFcnName + "_VPA",...
    "OpenRecordedVideoInVideoViewer", true,...
    "VideoViewerJumpToTime", 10.6);
```

Play the generated video.

- **Front Facing Camera** shows the image returned by the camera sensor. The left lane boundary is plotted in red and the right lane boundary is plotted in green. These lanes are returned by the Lane Marker Detector model. Tracked detections are also overlaid on the video.
- **Bird's-Eye Plot** shows true vehicle positions, sensor coverage areas, probabilistic detections, and track outputs. The plot title includes the simulation time so that you can correlate events between the video and previous static plots.

Close the figure.

```
close(hVideoViewer)
```

Explore Additional Scenarios

The previous simulations tested the `scenario_LFACC_03_Curve_StopnGo` scenario. This example provides additional scenarios that are compatible with the `HighwayLaneFollowingTestBench` model:

```
scenario_LF_01_Straight_RightLane
scenario_LF_02_Straight_LeftLane
scenario_LF_03_Curve_LeftLane
scenario_LF_04_Curve_RightLane
scenario_LFACC_01_Curve_DecelTarget
scenario_LFACC_02_Curve_AutoRetarget
scenario_LFACC_03_Curve_StopnGo
```

```
scenario_LFACC_04_Curve_CutInOut
scenario_LFACC_05_Curve_CutInOut_TooClose
scenario_LFACC_06_Straight_StopandGoLeadCar
```

These scenarios represent two types of testing.

- Use scenarios with the `scenario_LF_` prefix to test lane-detection and lane-following algorithms without obstruction from other vehicles. The vehicles in the scenario are positioned such that they are not seen by the ego vehicle.
- Use scenarios with the `scenario_LFACC_` prefix to test lane-detection and lane-following algorithms with other vehicles that are within the sensor coverage area of the ego vehicle.

Examine the comments in each file for more details about the geometry of the road and vehicles in each scenario. You can configure the `HighwayLaneFollowingTestBench` model and workspace to simulate these scenarios using the `helperSLHighwayLaneFollowingSetup` function.

For example, while evaluating the effects of a camera-based lane detection algorithm on closed-loop control, it can be helpful to begin with a scenario that has a road but no vehicles. To configure the model and workspace for such a scenario, use the following code.

```
helperSLHighwayLaneFollowingSetup("scenarioFcnName",...
    "scenario_LF_04_Curve_RightLane");
```

Enable the MPC update messages again.

```
mpcverbosity('on');
```

Conclusion

This example showed how to integrate vision processing, sensor fusion and controller components to simulate a highway lane following system in a closed-loop 3D simulation environment. The example also demonstrated various evaluation metrics to validate the performance of the designed system. If you have license to Simulink Coder™ and Embedded Coder™, you can generate ready to deploy code of the algorithm models for embedded real-time target (ERT).

Copyright 2019-2020 The MathWorks, Inc.

See Also

Blocks

Lane Keeping Assist System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

Highway Lane Change

This example shows how to simulate an automated lane change maneuver system for highway driving scenario.

Introduction

An automated lane change maneuver (LCM) system enables the ego vehicle to automatically move from one lane to another lane. The LCM system models the longitudinal and lateral control dynamics for automated lane change. An LCM system senses the environment for most important objects (MIOs) using on-board sensors, identifies an optimal trajectory that avoids these objects, and steers the ego vehicle along this trajectory.

This example shows how to design and test the planner and controller components of an LCM system. In this example, the lane change planner uses ground truth information from the scenario to detect MIOs. It then generates a feasible trajectory to negotiate a lane change that is executed by the lane change controller. In this example, you:

- **Explore the test bench model** — The model contains planning, controls, vehicle dynamics, scenario, and metrics to assess functionality.
- **Model the lane change planner** — The reference model finds the MIO, samples terminal states of the ego vehicle, and generates an optimal trajectory.
- **Model the lane change controller** — This model generates control commands for the ego vehicle based on the generated trajectory.
- **Simulate and visualize system behavior** — The test bench model is configured to test the integration of planning and controls to perform lane change maneuvers on a curved road with multiple vehicles.
- **Explore other scenarios** — These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own planner and controller components of an LCM system.

Explore Test Bench Model

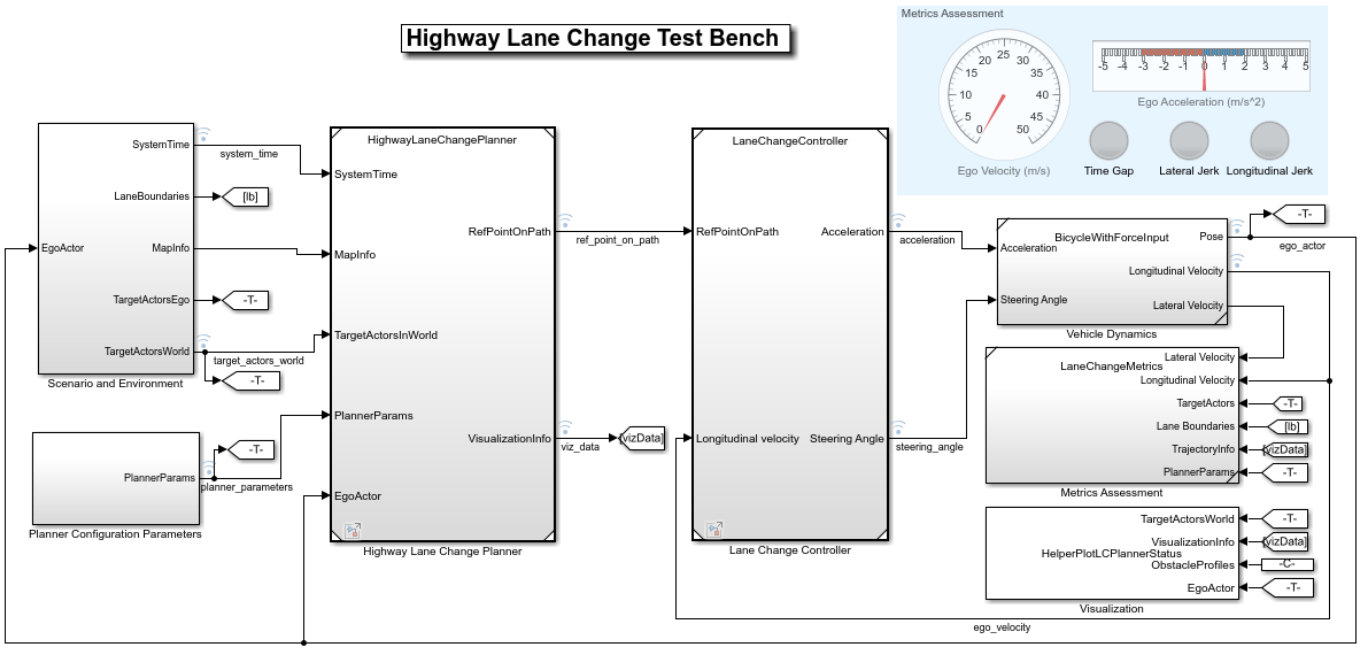
In this example, you use a system-level simulation test bench model to explore the behavior of the planner and controller components for a lane change maneuver system.

To explore the test bench model, open a working copy of the project example files. MATLAB® copies the files to an example folder so that you can edit them.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('HighwayLaneChange.zip', 'workDir', pwd);
```

Open the system-level simulation test bench model.

```
open_system('HighwayLaneChangeTestBench')
```



Copyright 2019-2021 The MathWorks, Inc.

Opening this model runs the `helperSLHighwayLaneChangeSetup` script that initializes the road scenario using the `drivingScenario` (Automated Driving Toolbox) object in the base workspace. It also configures the planner configuration parameters, controller design parameters, vehicle model parameters, and the Simulink® bus signals required for defining the inputs and outputs for the `HighwayLaneChangeTestBench` model.

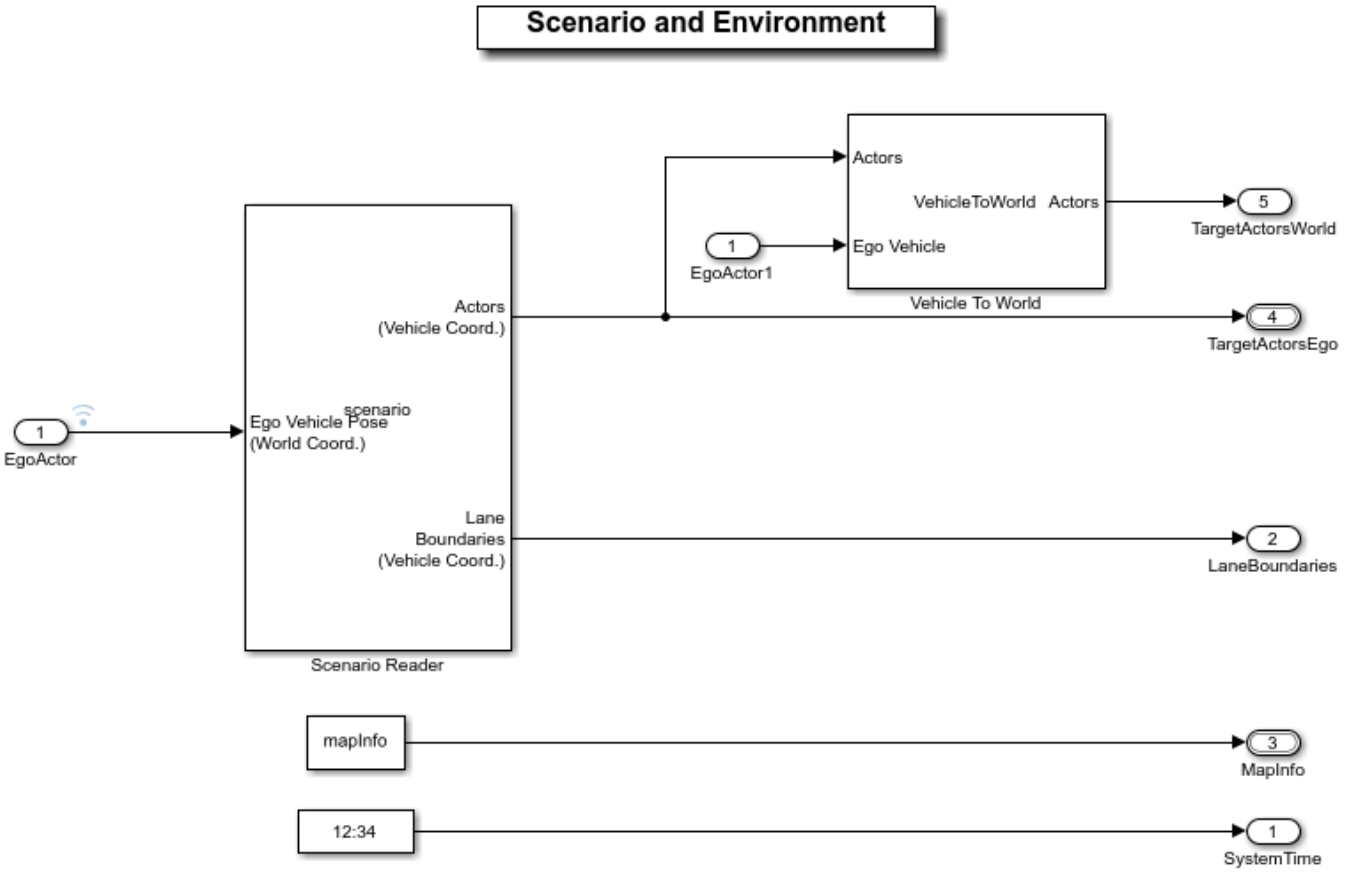
The test bench model contains the following subsystems.

- Scenario and Environment — Subsystem that specifies the scene, vehicles, and map data used for simulation.
- Planner Configuration Parameters — Subsystem that specifies the configuration parameters required for the planner algorithm.
- Highway Lane Change Planner — Subsystem that implements the lane change planner algorithm for highway.
- Lane Change Controller — Subsystem that specifies the path following controller that generates control commands to steer the ego vehicle along the generated trajectory.
- Vehicle Dynamics — Subsystem that specifies the dynamic model for the ego vehicle.
- Metrics Assessment — Subsystem that specifies metrics to assess system level behavior.

The Vehicle Dynamics subsystem models the ego vehicle using a Bicycle Model (Automated Driving Toolbox) and updates its state using commands received from the Lane Change Controller. For more details on Vehicle Dynamics subsystem, see “Highway Lane Following” (Automated Driving Toolbox) example.

The Scenario and Environment subsystem uses the Scenario Reader (Automated Driving Toolbox) block to provide road network and vehicle ground truth positions. This block also outputs map data required for the highway lane change planner algorithm. Open the Scenario and Environment subsystem.

```
open_system('HighwayLaneChangeTestBench/Scenario and Environment')
```



The Scenario Reader (Automated Driving Toolbox) block is configured to read the `drivingScenario` (Automated Driving Toolbox) object from the base workspace. It uses this object to read the actor data. It takes in ego vehicle information to perform a closed-loop simulation. This block outputs ground truth information of the lanes and actors in ego vehicle coordinates. The Vehicle To World (Automated Driving Toolbox) block is used to convert target vehicle positions from the vehicle coordinates to world coordinates. This subsystem reads map data from the base workspace and outputs information about the lanes and reference path.

The Planner Configuration Parameters subsystem reads base workspace variables using constant blocks and constructs a bus structure using the Bus Creator block. The bus created by this subsystem is used by the lane change planner.

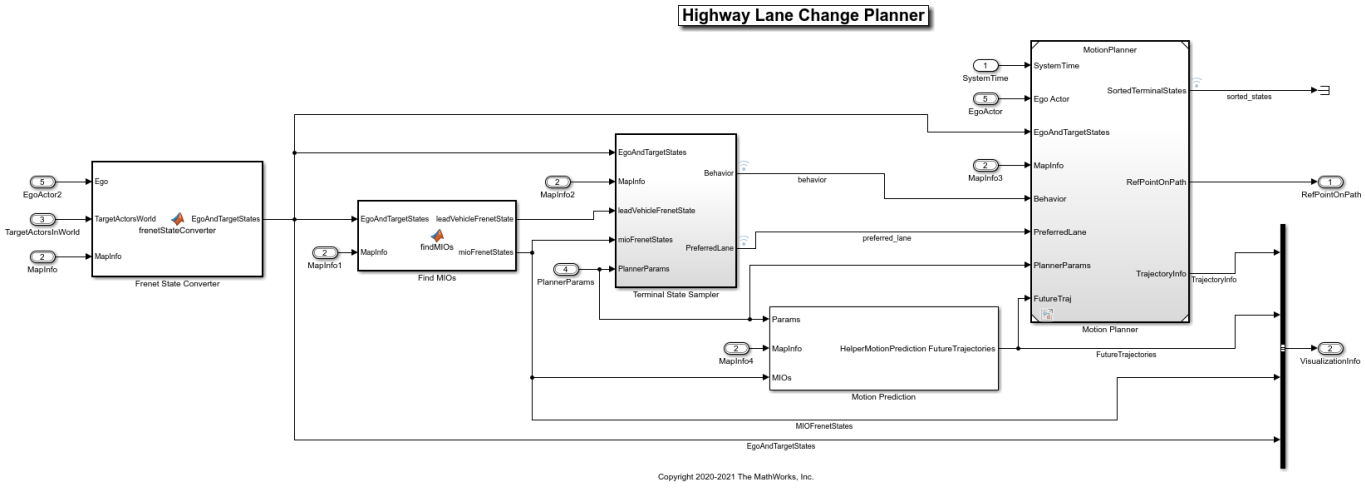
The Highway Lane Change Planner reference model uses ground truth actor positions in world coordinates, map data, and planner configuration parameters to perform trajectory planning for the automated lane change maneuver.

Model Highway Lane Change Planner

The Highway Lane Change Planner reference model implements the main algorithm for the highway lane change system. The model finds the MIOs surrounding the ego vehicle using the Fernet coordinate system. Subsequently, the model samples terminal states for different behaviors, predicts the motion of target actors, and generates multiple trajectories. Finally, the model evaluates the costs

of generated trajectories and checks for the possibility of collision and kinematic feasibility to estimate the optimal trajectory. Open the Highway Lane Change Planner reference model.

```
open_system('HighwayLaneChangePlanner')
```



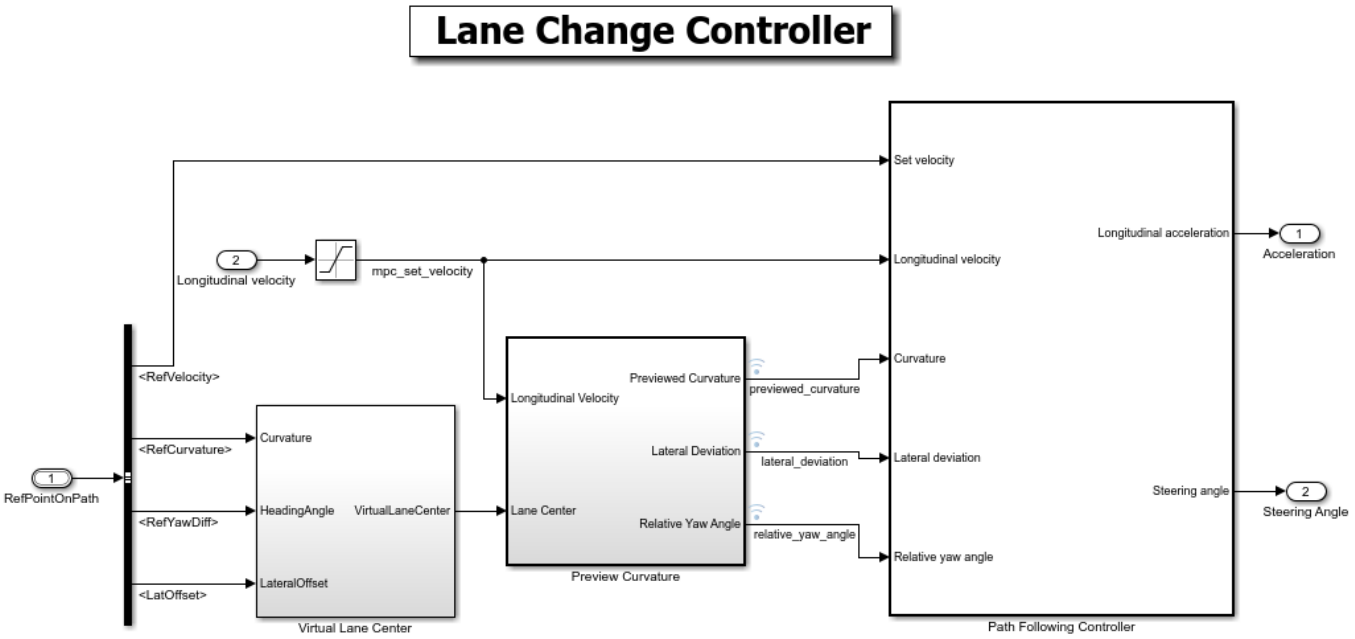
The Highway Lane Change Planner model contains the following blocks:

- The Frenet State Converter block converts the pose of the ego vehicle and other vehicles in the scenario into the Frenet coordinate system from world coordinates.
- The Find MIOs block identifies the most important objects (MIOs) surrounding the ego vehicle.
- The Terminal State Sampler block samples terminal states for cruise control, lead car following, and lane change behaviors. The Motion Prediction module predicts the motion of MIOs.
- The Motion Planner reference model generates an optimal trajectory from the sampled trajectories. This model checks the sampled trajectories for cost, feasibility, and the possibility of collision to identify the optimal trajectory. This block also computes the appropriate point on the trajectory for the ego vehicle to follow. For more information on the Highway Lane Change Planner, see “Generate Code for Highway Lane Change Planner” (Automated Driving Toolbox).

Model Lane Change Controller

The Lane Change Controller reference model simulates a path following control mechanism that keeps the ego vehicle traveling along the generated trajectory while tracking a set velocity. To do so, the controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle. The controller computes optimal control actions while satisfying velocity, acceleration, and steering angle constraints using adaptive model predictive control (MPC). Open the Lane Change Controller reference model.

```
open_system('LaneChangeController')
```



- The Virtual Lane Center subsystem creates a virtual lane from the path point. The virtual lane matches the format required by the Path Following Controller block.
- The Preview Curvature subsystem converts trajectory to curvature input required by Path Following Controller block.
- The Path Following Controller block uses the Path Following Control System block from the Model Predictive Control Toolbox™.

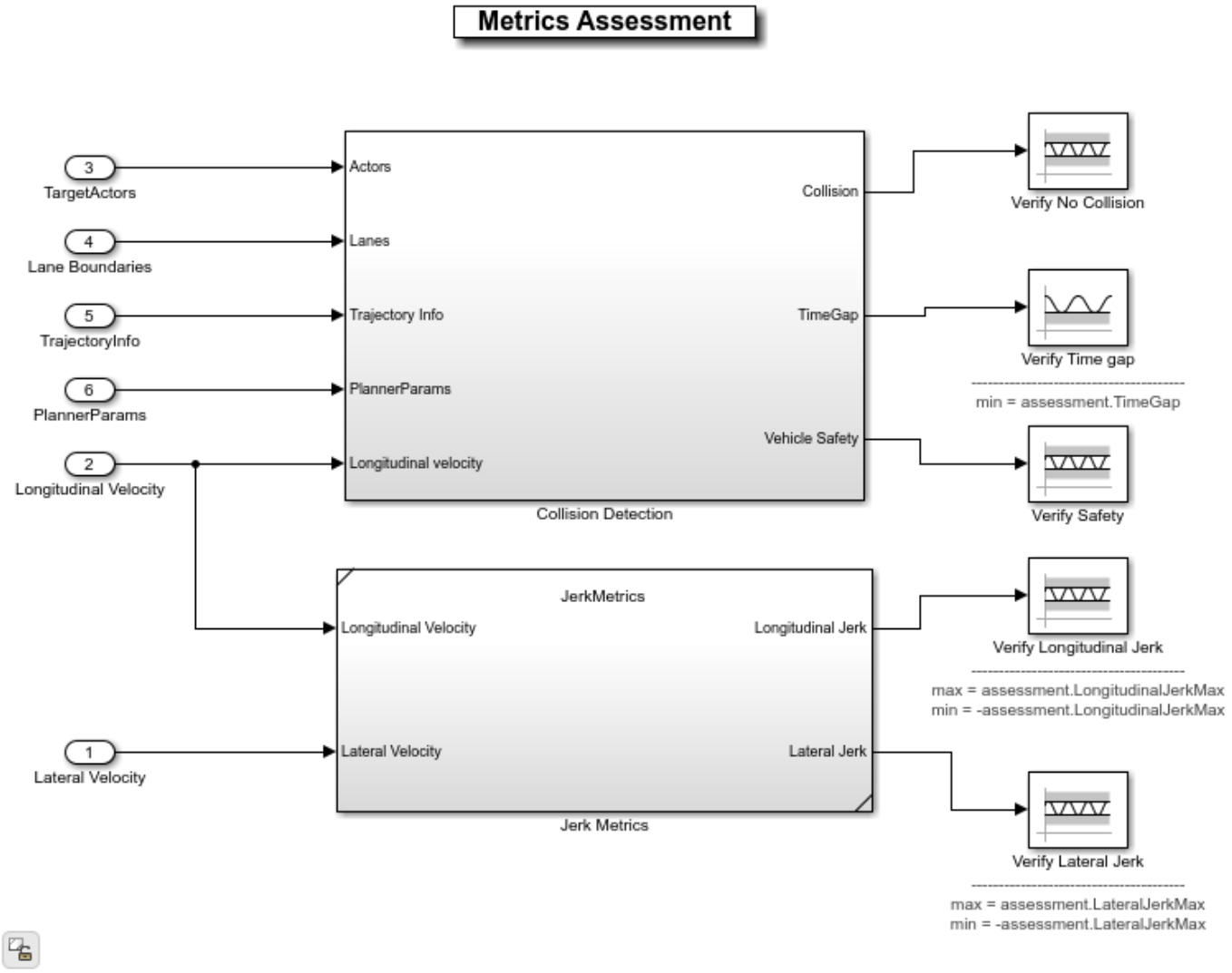
The Path Following Controller block keeps the vehicle traveling within a marked lane of a highway while maintaining a user-set velocity. This controller includes combined longitudinal and lateral control of the ego vehicle:

- Longitudinal control maintains a user-set velocity of the ego vehicle.
- Lateral control keeps the ego vehicle traveling along the center line of its lane by adjusting the steering of the ego vehicle.

Explore Metrics Assessment

The Metrics Assessment subsystem assesses system level behavior of the LCM system using the metrics mentioned below. Open the Metrics Assessment subsystem.

```
open_system('HighwayLaneChangeTestBench/Metrics Assessment')
```



- The **DetectCollision** block detects the collision of the ego vehicle with other vehicles and halts the simulation if a collision is detected.
- The **DetectLeadVehicle** block computes the headway between the ego and lead vehicles, which is used for computing the **TimeGap** value.
- The **TimeGap** value is calculated using the distance to the lead vehicle (headway) and the longitudinal velocity of the ego vehicle, and it is evaluated against prescribed limits.
- The **LongitudinalJerk** value is calculated using the longitudinal velocity and evaluated against prescribed limits.
- The **LateralJerk** value is calculated using the lateral velocity evaluated against prescribed limits.

Simulate and Visualize System Behavior

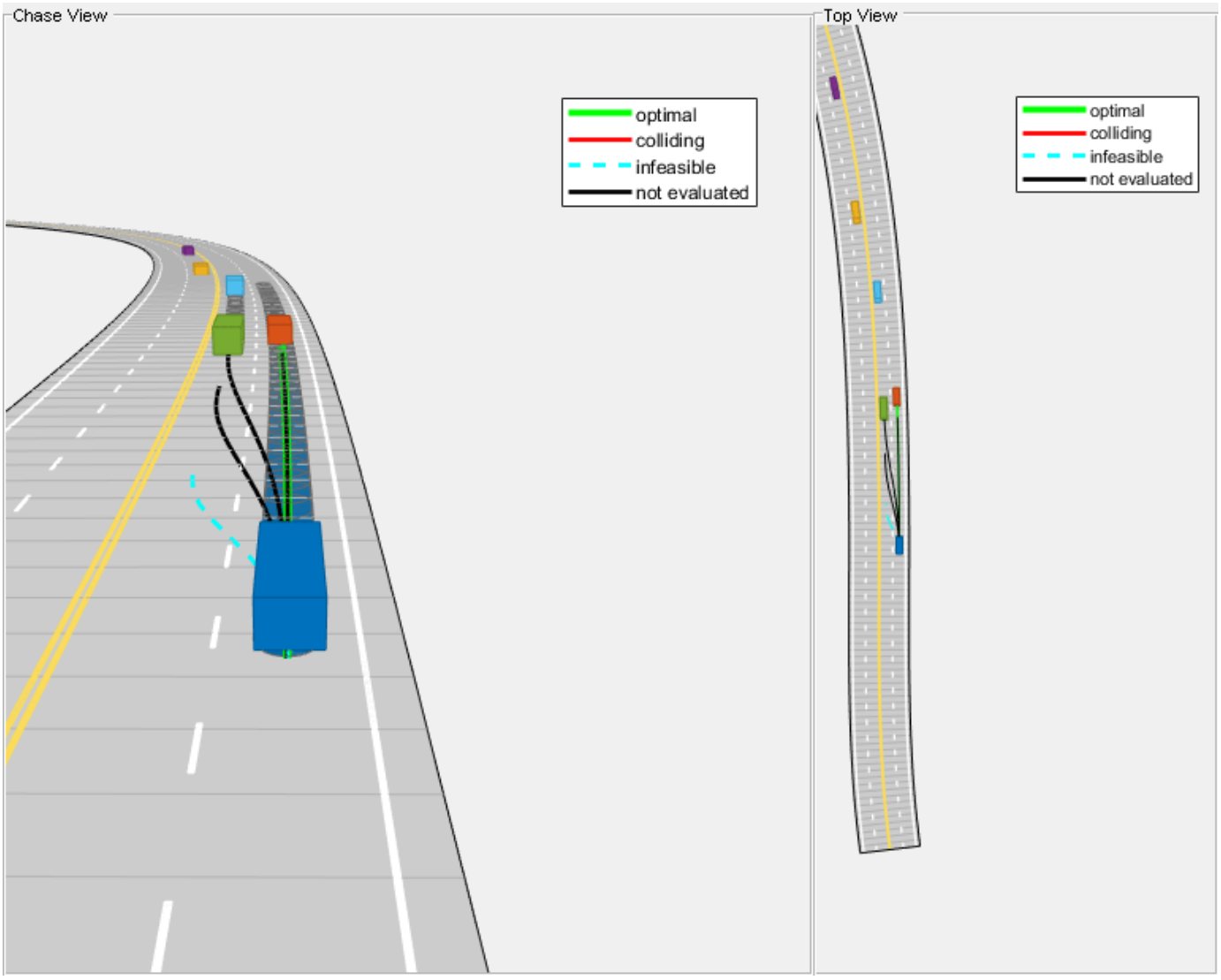
Set up and run the **HighwayLaneChangeTestBench** simulation model to visualize the behavior of the system during a lane change. The Visualization block in the model creates a MATLAB figure that shows the chase view and top view of the scenario and plots the ego vehicle, sampled trajectories,

capsule list, and other vehicles in the scenario. Configure the **HighwayLaneChangeTestBench** model to use the `scenario_LC_15_StopnGo_Curved` scenario.

```
helperSLHighwayLaneChangeSetup("scenarioFcnName","scenario_LC_15_StopnGo_Curved")
```

Simulate the model for 5 seconds. The lane change planner reference model generates a trajectory to navigate the vehicle in the scenario. To reduce command-window output, first turn off the MPC update messages.

```
mpcverbosity('off');
sim("HighwayLaneChangeTestBench","StopTime","5");
```

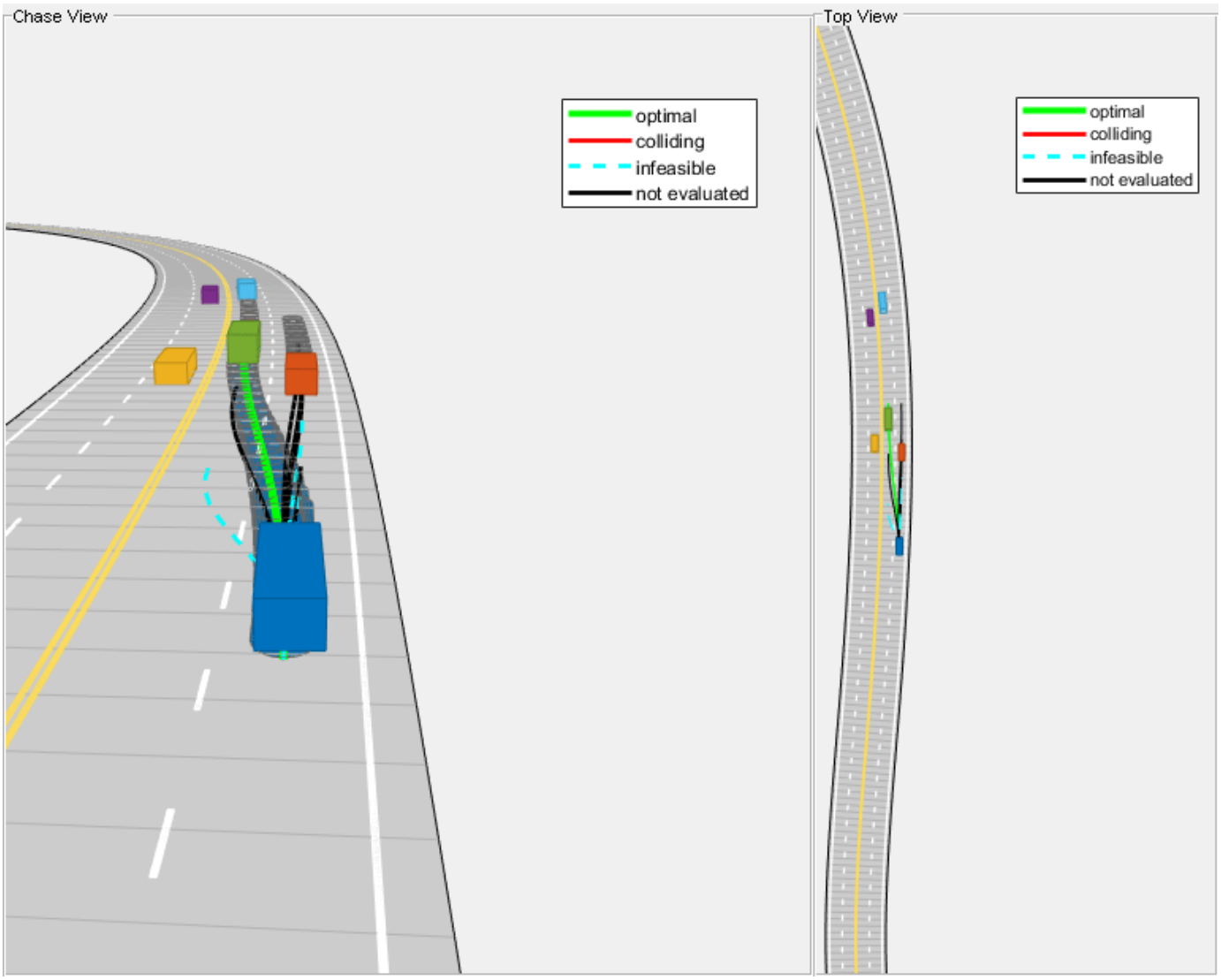


Close the figure.

```
hLCPlot = findobj( 'Type', 'Figure', 'Name', 'Lane Change Status Plot');
if ~isempty(hLCPlot)
    close(hLCPlot);
end
```

Run the simulation for 8 seconds. The lane change planner reference model generates a trajectory to navigate around a slower lead vehicle.

```
sim("HighwayLaneChangeTestBench","StopTime","8");
```

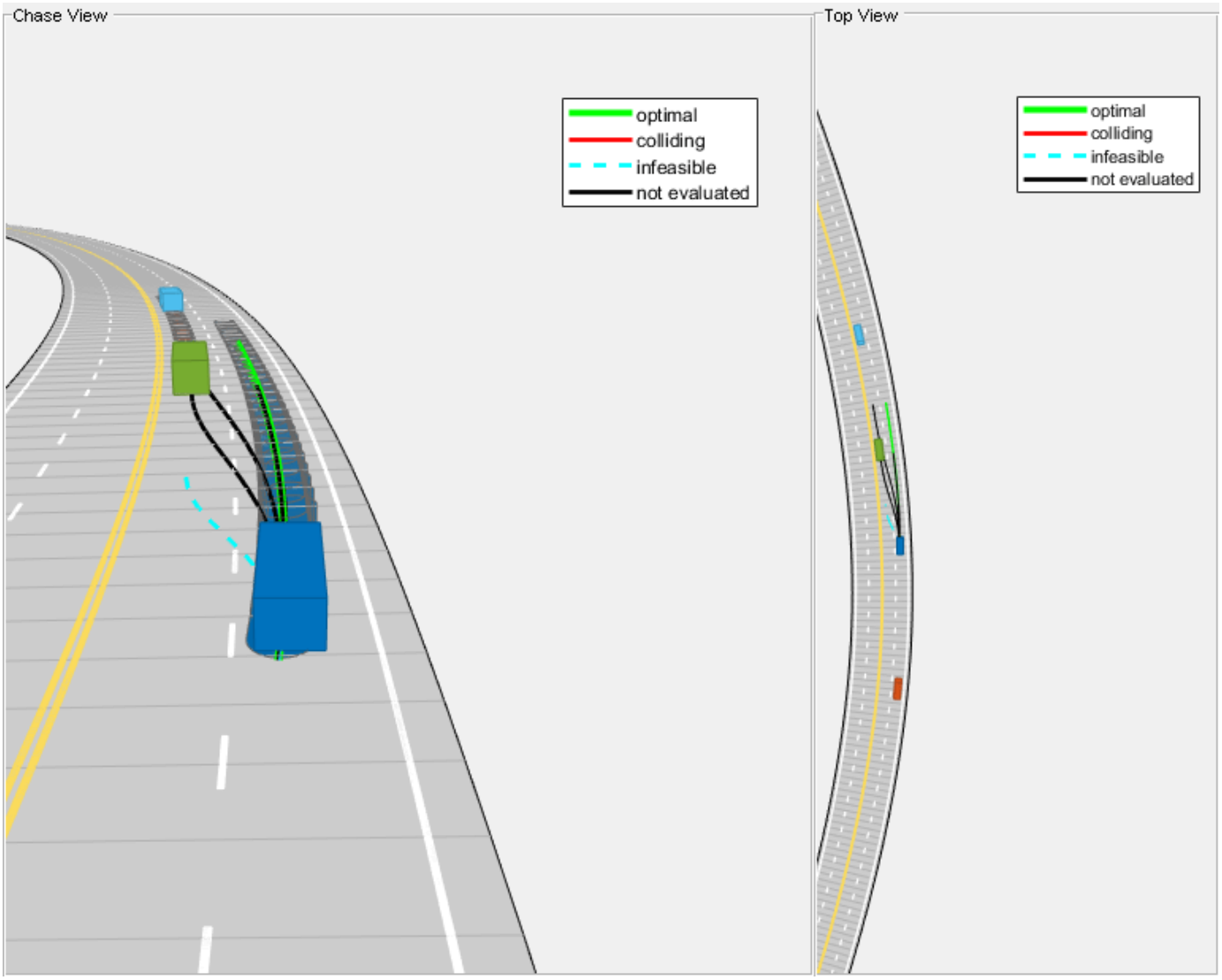


Close the figure.

```
hLCPlot = findobj( 'Type', 'Figure', 'Name', 'Lane Change Status Plot');
if ~isempty(hLCPlot)
    close(hLCPlot);
end
```

Run the simulation for 18 seconds. The lane change planner reference model generates a trajectory to navigate the vehicle to the left lane and then to the right lane to avoid collision with the slow moving lead vehicle. Observe that the ego vehicle performs a lane change twice to avoid collision while maintaining a set velocity.

```
simout = sim("HighwayLaneChangeTestBench","StopTime","18");
```



Close the figure.

```
hLCPlot = findobj( 'Type', 'Figure', 'Name', 'Lane Change Status Plot');
if ~isempty(hLCPlot)
    close(hLCPlot);
end
```

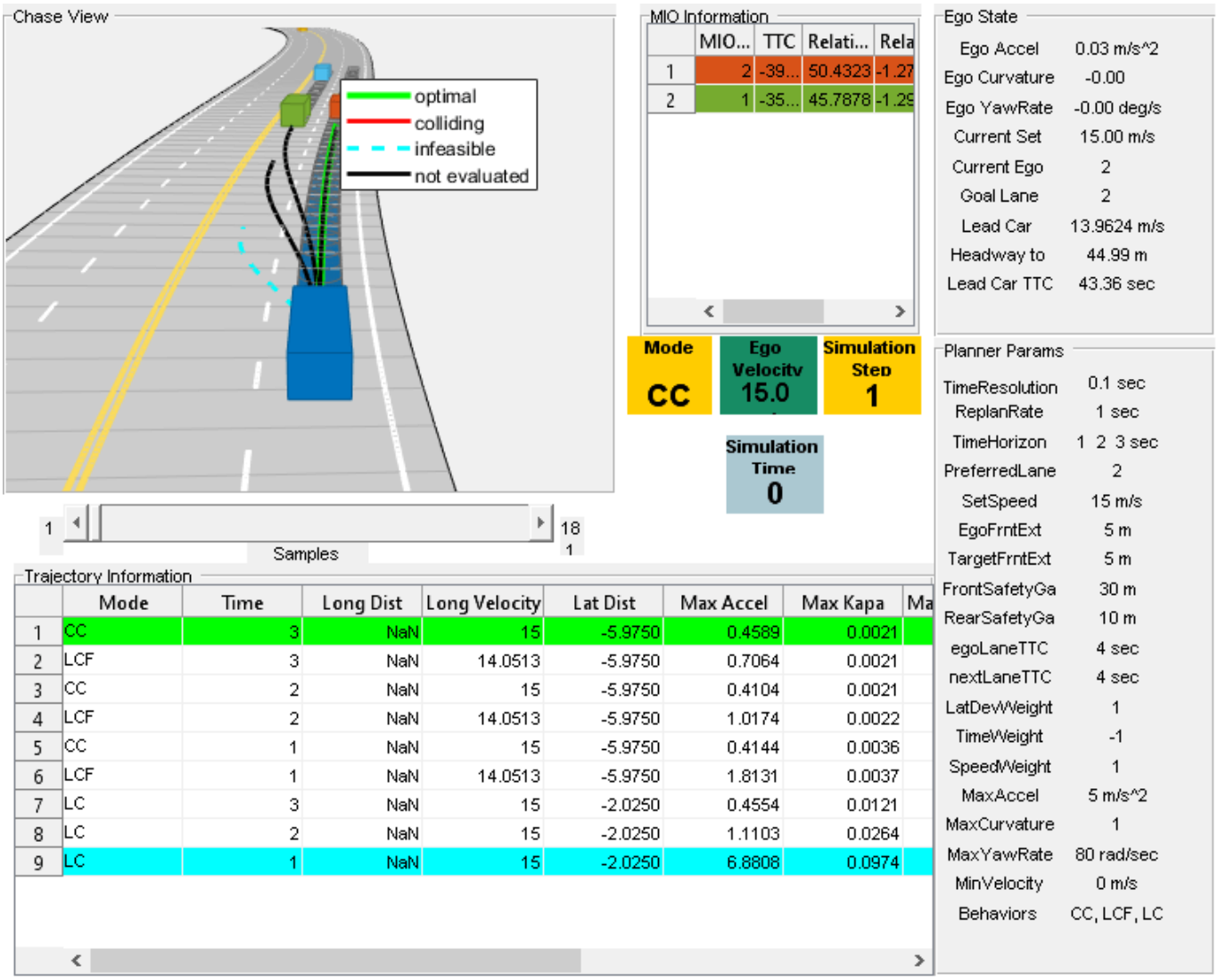
During the simulation, the model logs signals to base workspace as `logsout`. You can analyze the simulation results and debug any failures in the system behavior using the `helperAnalyzeLCSimulationResults` function. The function creates a MATLAB figure and plots chase view of the scenario. The slider in the figure enables you to select a desired simulation step to analyze different parameters shown in these panes:

- **Chase View** — Shows chase view of the scenario showing ego vehicle, sampled trajectories, capsule list, and other vehicles.
- **Trajectory Information** — Shows different attributes of sampled trajectories. The highlighted rows show the type of sampled trajectory by using the same color coding as shown in the Chase View.

- **MIO Information** — Shows different attributes of identified MIOs. The color of the row matches with the face color of the corresponding vehicle.
- **Mode** — Shows the selected behavior for the ego vehicle.
- **Ego Velocity** — Shows the velocity of ego vehicle. Units are in meters per second.
- **Simulation Step** — Shows the simulation step number set using the slider.
- **Simulation Time** — Shows time corresponding to simulation step. Units are in meters.
- **Ego State** — Shows parameters of the ego vehicle and identified lead vehicle.
- **Planner Parameters** — Shows configuration parameters for the planner.

Run the script and explore the plot.

```
helperAnalyzeLCSimulationResults(simout.logsout);
```



Explore Other Scenarios

In the previous section, you explored the system behavior for the `scenario_LC_15_StopnGo_Curved` scenario. Below is a list of scenarios that are compatible with the `HighwayLaneChangeTestBench` model.

```
scenario_LC_01_SlowMoving
scenario_LC_02_SlowMovingWithPassingCar
scenario_LC_03_DisabledCar
scenario_LC_04_CutInWithBrake
scenario_LC_05_SingleLaneChange
scenario_LC_06_DoubleLaneChange
scenario_LC_07_RightLaneChange
scenario_LC_08_SlowmovingCar_Curved
scenario_LC_09_CutInWithBrake_Curved
scenario_LC_10_SingleLaneChange_Curved
scenario_LC_11_MergingCar_HighwayEntry
scenario_LC_12_CutInCar_HighwayEntry
scenario_LC_13_DisabledCar_Ushape
scenario_LC_14_DoubleLaneChange_Ushape
scenario_LC_15_StopnGo_Curved [Default]
```

These scenarios are created using the Driving Scenario Designer (Automated Driving Toolbox) and are exported to a scenario file. Examine the comments in each file for more details on the road and vehicles in each scenario. You can configure the `HighwayLaneChangeTestBench` and workspace to simulate these scenarios using the `helperSLHighwayLaneChangeSetup` function. For example, you can configure the simulation for a curved road scenario.

```
helperSLHighwayLaneChangeSetup("scenarioFcnName","scenario_LC_10_SingleLaneChange_Curved");
```

Conclusion

This example shows how to simulate a highway lane change maneuver using ground truth vehicle positions.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

See Also

More About

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Automate Testing for Highway Lane Following” on page 11-89
- “Highway Lane Change” on page 11-78

Automate Testing for Highway Lane Following

This example shows how to assess the functionality of a lane-following application by defining scenarios based on requirements, automating testing of components and the generated code for those components. The components include lane-detection, sensor fusion, decision logic, and controls. This example builds on the “Highway Lane Following” (Automated Driving Toolbox) example.

Introduction

A highway lane-following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance from a preceding vehicle in the same lane. The system typically includes lane detection, sensor fusion, decision logic, and controls components. System-level simulation is a common technique for assessing functionality of the integrated components. Simulations are configured to test scenarios based on system requirements. Automatically running these simulations enables regression testing to verify system-level functionality.

The “Highway Lane Following” (Automated Driving Toolbox) example showed how to simulate a system-level model for lane-following. This example shows how to automate testing that model against multiple scenarios using Simulink Test™. The scenarios are based on system-level requirements. In this example, you will:

1. **Review requirements:** The requirements describe system-level test conditions. Simulation test scenarios are created to represent these conditions.
2. **Review the test bench model:** Review the system-level lane-following test bench model that contains metric assessments. These metric assessments integrate the test bench model with Simulink Test for the automated testing.
3. **Disable runtime visualizations:** Runtime visualizations are disabled to reduce execution time for the automated testing.
4. **Automate testing:** A test manager is configured to simulate each test scenario, assess success criteria, and report results. The results are explored dynamically in the test manager and exported to a PDF for external reviewers.
5. **Automate testing with generated code:** The lane detection, sensor fusion, decision logic, and controls components are configured to generate C++ code. The automated testing is run on the generated code to verify expected behavior.
6. **Automate testing in parallel:** Overall execution time for running the tests is reduced using parallel computing on a multi-core computer.

Testing the system-level model requires a photorealistic simulation environment. In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error("The 3D simulation environment requires a Windows 64-bit platform");
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```

Review Requirements

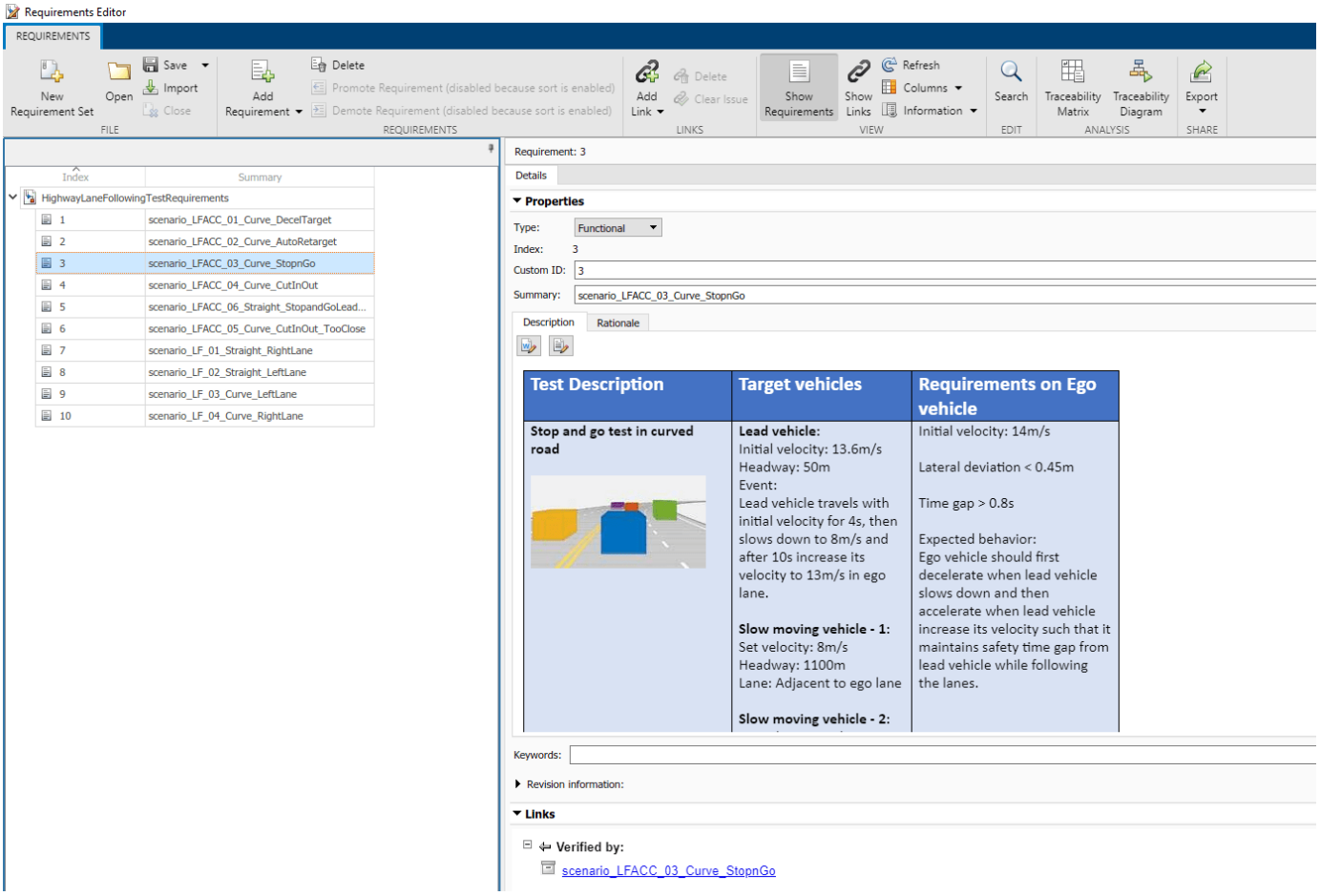
Simulink Requirements™ lets you author, analyze, and manage requirements within Simulink. This example contains ten test scenarios, with high-level testing requirements defined for each scenario. Open the requirement set.

To explore the test requirements and test bench model, open a working copy of the project example files. MATLAB copies the files to an example folder so that you can edit them. The `TestAutomation` folder contains the files that enables the automate testing.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('HighwayLaneFollowing.zip', 'workDir', pwd);
```

```
open('HighwayLaneFollowingTestRequirements.slreqx')
```

Alternatively, you can also open the file from the **Requirements** tab of the Requirements Manager app in Simulink.



Each row in this file specifies the requirements in textual and graphical formats for testing the lane-following system for a test scenario. The scenarios with the **scenario_LF_** prefix enable you to test lane-detection and lane-following algorithms without obstruction by other vehicles. The scenarios with the **scenario_LFACC_** prefix enable you to test lane-detection, lane-following, and ACC behavior with other vehicles on the road.

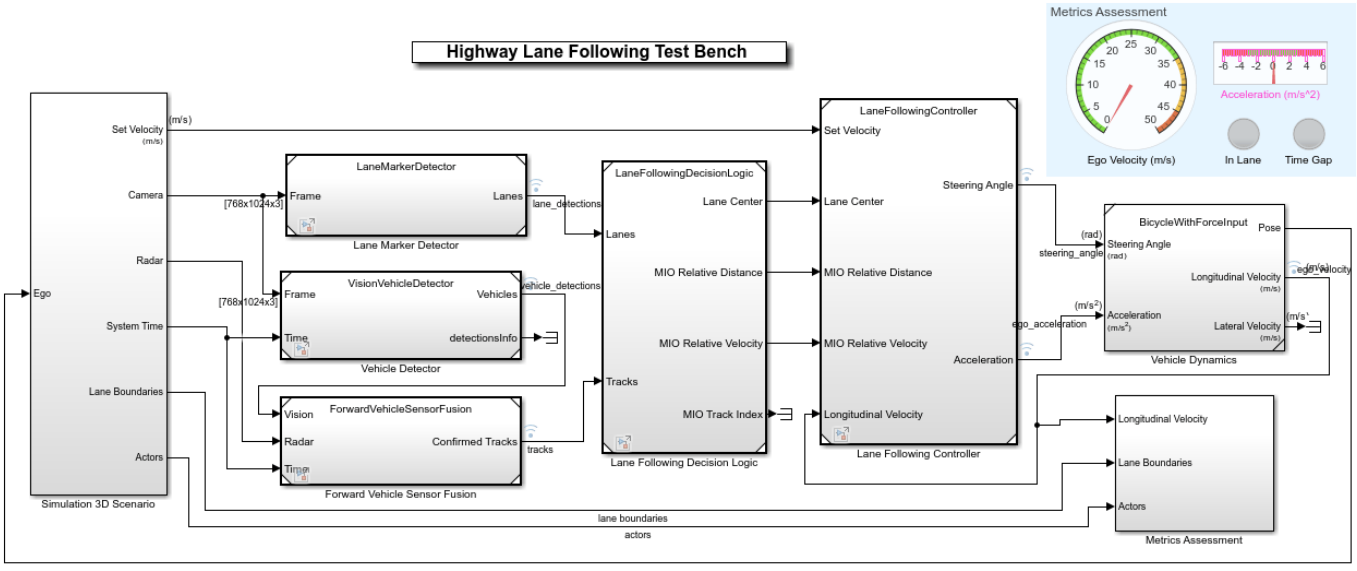
1. `scenario_LF_01_Straight_RightLane` — Straight road scenario with ego vehicle in right lane.
2. `scenario_LF_02_Straight_LeftLane` — Straight road scenario with ego vehicle in left lane.
3. `scenario_LF_03_Curve_LeftLane` — Curved road scenario with ego vehicle in left lane.
4. `scenario_LF_04_Curve_RightLane` — Curved road scenario with ego vehicle in right lane.
5. `scenario_LFACC_01_Curve_DecelTarget` — Curved road scenario with a decelerating lead vehicle in ego lane.
6. `scenario_LFACC_02_Curve_AutoRetarget` — Curved road scenario with changing lead vehicles in ego lane. This scenario tests the ability of the ego vehicle to retarget to a new lead vehicle while driving along a curve.
7. `scenario_LFACC_03_Curve_StopnGo` — Curved road scenario with a lead vehicle slowing down in ego lane.
8. `scenario_LFACC_04_Curve_CutInOut` — Curved road scenario with a fast moving car in the adjacent lane cuts into the ego lane and cuts out from ego lane.
9. `scenario_LFACC_05_Curve_CutInOut_TooClose` — Curved road scenario with a fast moving car in the adjacent lane cuts into the ego lane and cuts out from ego lane aggressively.
10. `scenario_LFACC_06_Straight_StopandGoLeadCar` — Straight road scenario with a lead vehicle that breaks down in ego lane.

These requirements are implemented as test scenarios with the same names as the scenarios used in the `HighwayLaneFollowingTestBench` model.

Review Test Bench Model

This example reuses the `HighwayLaneFollowingTestBench` model from the “Highway Lane Following” (Automated Driving Toolbox) example. Open the test bench model.

```
open_system("HighwayLaneFollowingTestBench");
```



Copyright 2019-2021 The MathWorks, Inc.

This test bench model has **Simulation 3D Scenario**, **Lane Marker Detector**, **Vehicle Detector**, **Forward Vehicle Sensor Fusion**, **Lane Following Decision Logic** and **Lane Following Controller** and **Vehicle Dynamics** components.

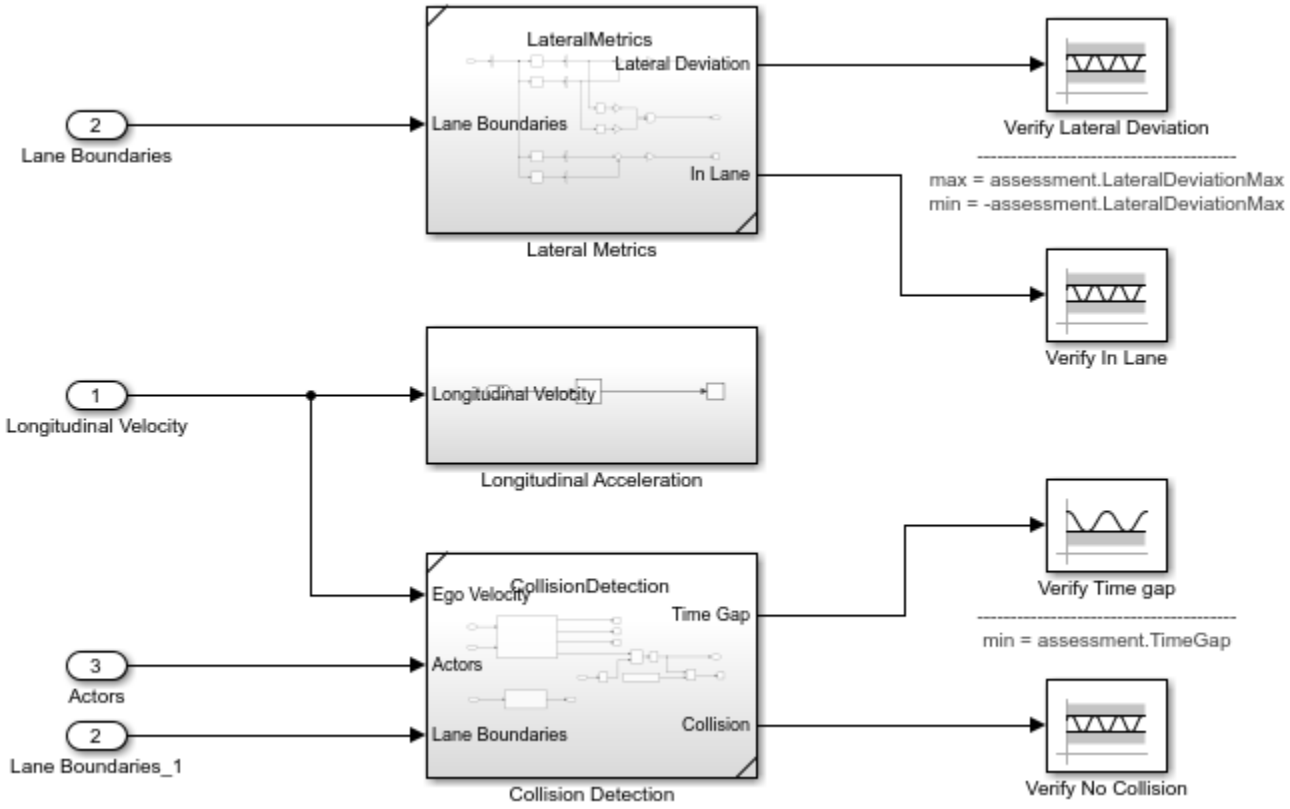
This test bench model is configured using the `helperSLHighwayLaneFollowingSetup` script. This setup script takes `scenarioName` as input. `scenarioName` can be any one of the previously described test scenarios. To run the setup script, use code:

```
scenarioName = "scenario_LFACC_03_Curve_StopnGo";
helperSLHighwayLaneFollowingSetup("scenarioFcnName",scenarioName);
```

You can now simulate the model and visualize the results. For more details on the analysis of the simulation results and the design of individual components in the test bench model, see the “Highway Lane Following” (Automated Driving Toolbox) example.

In this example, the focus is more on automating the simulation runs for this test bench model using Simulink Test for the different test scenarios. The **Metrics Assessment** subsystem enables integration of system-level metric evaluations with Simulink Test. This subsystem uses Check Static Range (Simulink) blocks for this integration. Open the **Metrics Assessment** subsystem.

```
open_system("HighwayLaneFollowingTestBench/Metrics Assessment");
```



In this example, four metrics are used to assess the lane-following system.

- **Verify Lateral Deviation**: Verifies that the lateral deviation from the centerline of the lane is within prescribed thresholds for the corresponding scenario. Prescribed thresholds are defined while authoring the test scenario.

- **Verify In Lane**: Verifies that the ego vehicle is following one of the lanes on the road throughout the simulation.
- **Verify Time gap**: Verifies that the time gap between the ego vehicle and the lead vehicle is above 0.8 seconds. The time gap between the two vehicles is defined as the ratio of the calculated headway distance to the ego vehicle velocity.
- **Verify No Collision**: Verifies that the ego vehicle does not collide with the lead vehicle at any point during the simulation.

Disable Runtime Visualizations

The system-level test bench model visualizes intermediate outputs during the simulation for the analysis of different components in the model. These visualizations are not required when the tests are automated. You can reduce execution time for the automated testing by disabling them.

Disable runtime visualizations for the **Lane Marker Detector** subsystem.

```
load_system('LaneMarkerDetector');
blk = 'LaneMarkerDetector/Lane Marker Detector';
set_param(blk,'EnableDisplays','off');
```

Disable runtime visualizations for the **Vehicle Detector** subsystem.

```
load_system('VisionVehicleDetector');
blk = 'VisionVehicleDetector/Vision Vehicle Detector/ACF/ACF';
set_param(blk,'EnableDisplay','off');
```

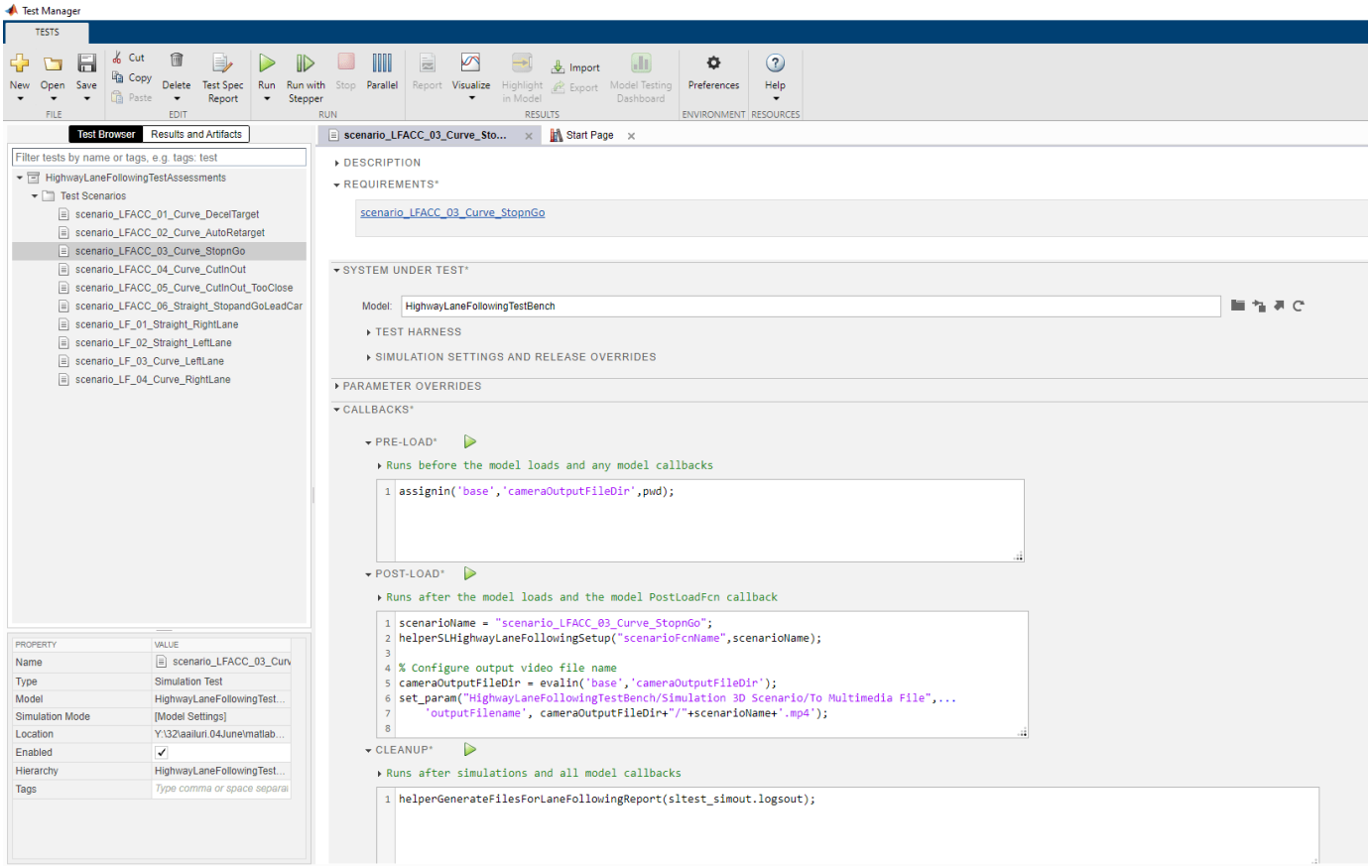
Configure the Simulation 3D Scene Configuration (Automated Driving Toolbox) block to run the Unreal Engine in headless mode, where the 3D simulation window is disabled.

```
blk = ['HighwayLaneFollowingTestBench/Simulation 3D Scenario/', ...
       'Simulation 3D Scene Configuration'];
set_param(blk,'EnableWindow','off');
```

Automate Testing

The Test Manager is configured to automate the testing of the lane-following application. Open the `HighwayLaneFollowingTestAssessments.mldatx` test file in the Test Manager.

```
sltestmgr;
testFile = sltest.testmanager.load('HighwayLaneFollowingTestAssessments.mldatx');
```



Observe the populated test cases that were authored previously in this file. Each test case is linked to the corresponding requirement in the Requirements Editor for traceability. Each test case uses the `POST-LOAD` callback to run the setup script with appropriate inputs and to configure the output video file name. After the simulation of the test case, it invokes `helperGenerateFilesForLaneFollowingReport` from the `CLEAN-UP` callback to generate the plots explained in the “Highway Lane Following” (Automated Driving Toolbox) example.

Run and explore results for a single test scenario:

To reduce command-window output, turn off the MPC update messages.

```
mpcverbosity('off');
```

To test the system-level model with the `scenario_LFACC_03_Curve_StopnGo` test scenario from Simulink Test, use this code:

```
testSuite = getTestSuiteByName(testFile,'Test Scenarios');
testCase = getTestCaseByName(testSuite,'scenario_LFACC_03_Curve_StopnGo');
resultObj = run(testCase);
```

To generate a report after the simulation, use this code:

```
sltest.testmanager.report(resultObj,'Report.pdf',...,
    'Title','Highway Lane Following',...
    'IncludeMATLABFigures',true,...
```

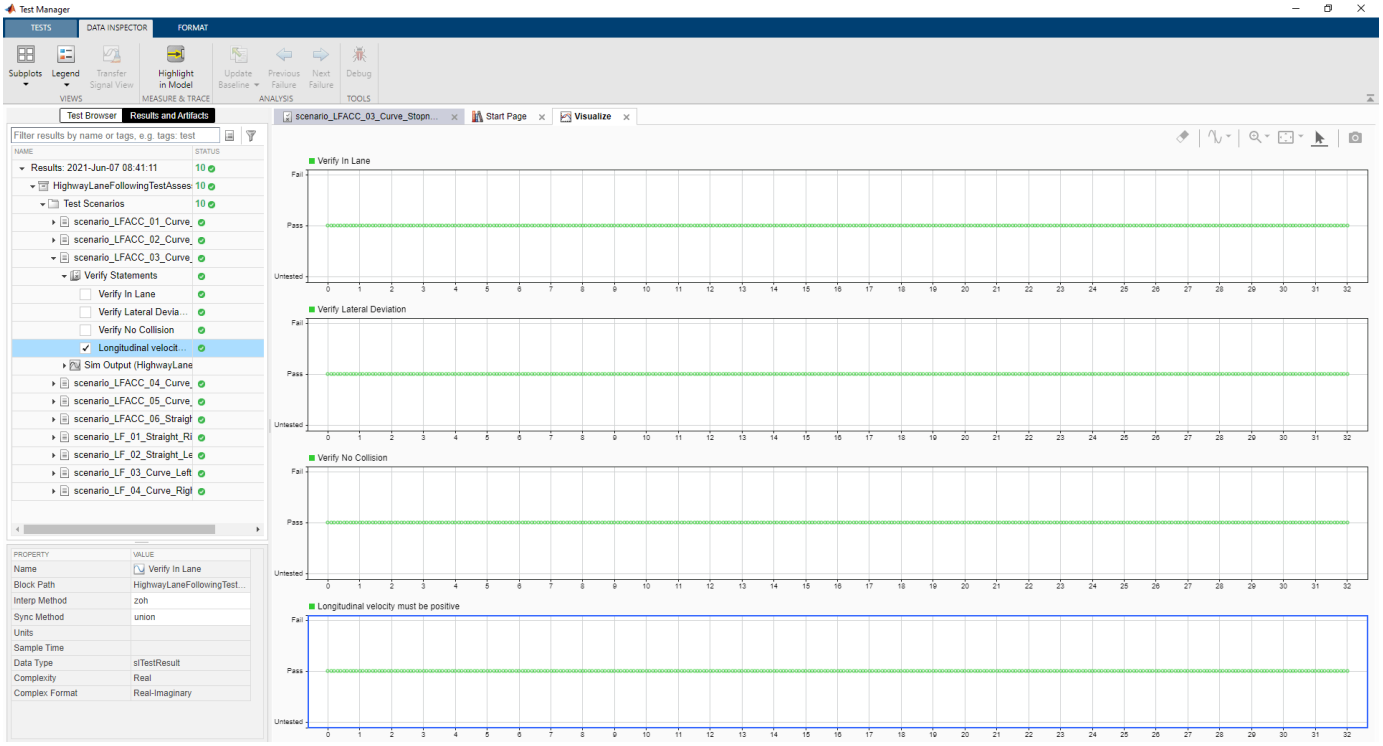
```
'IncludeErrorMessages',true,...
'IncludeTestResults',0,'LaunchReport',true);
```

Examine the report `Report.pdf`. Observe that the **Test environment** section shows the platform on which the test is run and the MATLAB® version used for testing. The **Summary** section shows the outcome of the test and duration of the simulation in seconds. The **Results** section shows pass/fail results based on the assessment criteria. This section also shows the plots logged from the `helperGenerateFilesForLaneFollowingReport` function.

Run and explore results for all test scenarios:

You can simulate the system for all the tests by using `sltest.testmanager.run`. Alternatively, you can simulate the system by clicking **Play** in the Test Manager app.

After completion of the test simulations, the results for all the tests can be viewed in the **Results and Artifacts** tab of the Test Manager. For each test case, the Check Static Range (Simulink) blocks in the model are associated with the Test Manager to visualize overall pass/fail results.



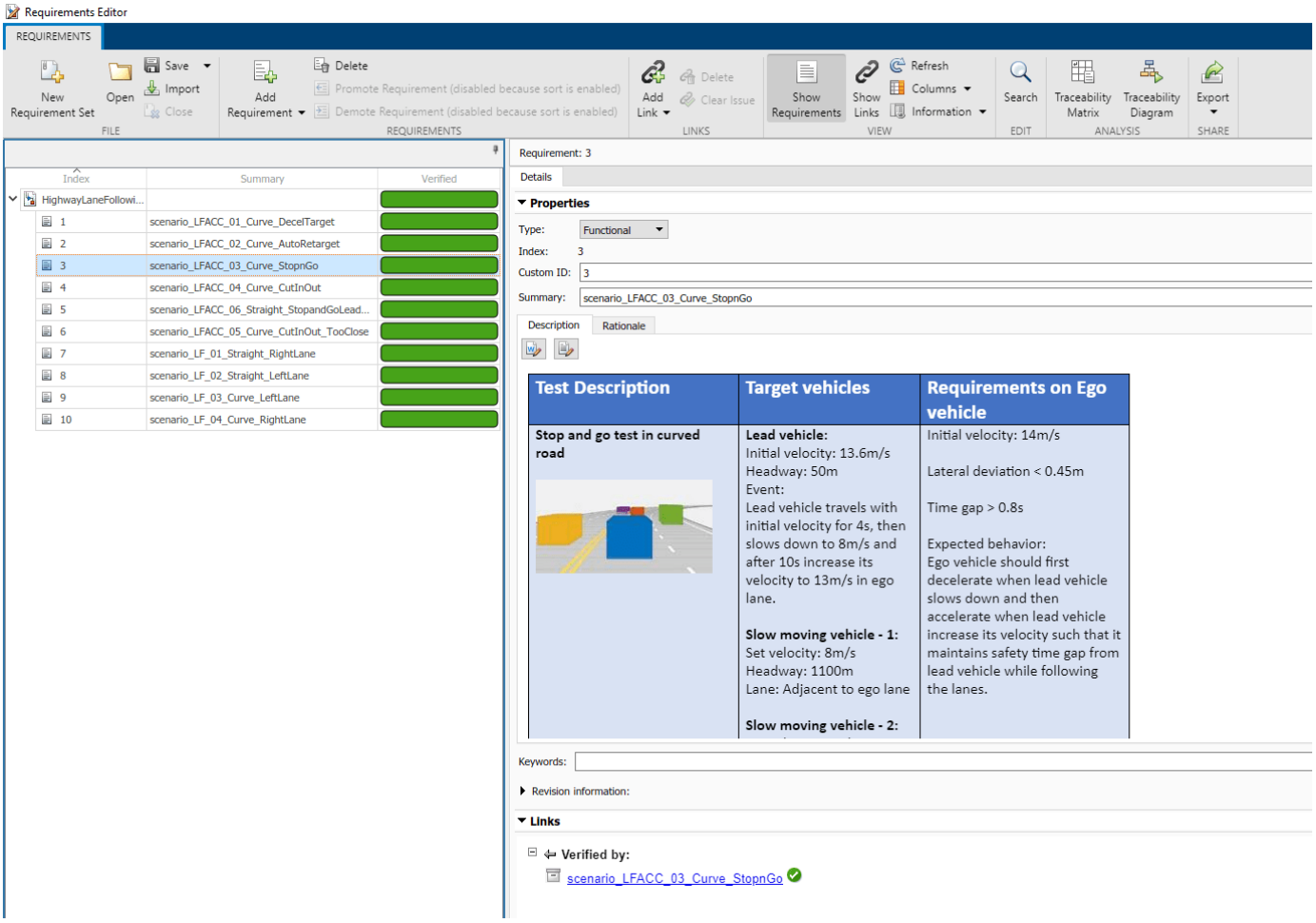
You can find the generated report in current working directory. This report contains a detailed summary of pass/fail statuses and plots for each test case.

Summary

Name	Outcome	Duration (Seconds)
HighwayLaneFollowingTestAssessments	10	2149.277
Test Scenarios	10	2149.278
scenario_LFACC_01_Curve_DecelTarget		448.803
scenario_LFACC_02_Curve_AutoRetarget		218.906
scenario_LFACC_03_Curve_StopnGo		268.746
scenario_LFACC_04_Curve_CutInOut		200.885
scenario_LFACC_05_Curve_CutInOut_TooClose		237.335
scenario_LFACC_06_Straight_StopandGoLeadCar		128.585
scenario_LF_01_Straight_RightLane		157.953
scenario_LF_02_Straight_LeftLane		145.923
scenario_LF_03_Curve_LeftLane		177.465
scenario_LF_04_Curve_RightLane		164.469

Verify test status in Requirements Editor:

Open the Requirements Editor and select **Display**. Then, select **Verification Status** to see a verification status summary for each requirement. Green and red bars indicate the pass/fail status of simulation results for each test.



Automate Testing with Generated Code

The `HighwayLaneFollowingTestBench` model enables integrated testing of **Lane Marker Detector**, **Vehicle Detector**, **Forward Vehicle Sensor Fusion**, **Lane Following Decision Logic**, and **Lane Following Controller** components. It is often helpful to perform regression testing of these components through software-in-the-loop (SIL) verification. If you have Embedded Coder™ Simulink Coder™ license, then you can generate code for these components. This workflow lets you verify that the generated code produces expected results that match the system-level requirements throughout simulation.

Set **Lane Marker Detector** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Marker Detector';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Vehicle Detector** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Vehicle Detector';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Forward Vehicle Sensor Fusion** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Forward Vehicle Sensor Fusion';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Lane Following Decision Logic** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Following Decision Logic';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Set **Lane Following Controller** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Following Controller';
set_param(model,'SimulationMode','Software-in-the-loop');
```

Now, run `sltest.testmanager.run` to simulate the system for all the test scenarios. After the completion of tests, review the plots and results in the generated report.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

Automate Testing in Parallel

If you have a Parallel Computing Toolbox™ license, then you can configure Test Manager to execute tests in parallel using a parallel pool. To run tests in parallel, save the models after disabling the runtime visualizations using `save_system('LaneMarkerDetector')`, `save_system('VisionVehicleDetector')` and `save_system('HighwayLaneFollowingTestBench')`. Test Manager uses the default Parallel Computing Toolbox cluster and executes tests only on the local machine. Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results. For more information on how to configure tests in parallel from the Test Manager, see “Run Tests Using Parallel Execution” (Simulink Test).

See Also

More About

- “Automated Driving Using Model Predictive Control” on page 11-2
- “Highway Lane Following” on page 11-62

Highway Lane Following with Intelligent Vehicles

This example shows how to simulate a lane following application in a scenario that contains intelligent target vehicles. The intelligent target vehicles are the non-ego vehicles in the scenario and are programmed to adapt their trajectories based on the behavior of its neighboring vehicles. In this example, you will:

1. Model the behavior of the target vehicles to dynamically adapt their trajectories in order to perform one of the following behaviors: velocity keeping, lane following, or lane change.

2. Simulate and test the lane following application in response to the dynamic behavior of the target vehicles on straight road and curved road scenarios.

You can also apply the modeling patterns used in this example to test your own lane following algorithms.

Introduction

The highway lane following system developed in this example steers the ego vehicle to travel within a marked lane. The system tests the lane following capability in the presence of other non-ego vehicles, which are the target vehicles. For regression testing, it is often sufficient for the target vehicles to follow a predefined trajectory. To randomize the behavior and identify edge cases like aggressive lane change in front of the ego vehicle, it is beneficial to add intelligence to the target vehicles.

This example builds on the “Highway Lane Following” (Automated Driving Toolbox) example that demonstrates lane following in the presence of target vehicles that follow predefined trajectories. This example modifies the scenario simulation framework of the “Highway Lane Following” (Automated Driving Toolbox) example by adding functionalities to model and simulate intelligent target vehicles. The intelligent target vehicles added to this example adapt their trajectories based on the behavior of the neighboring vehicles and the environment. In response, the lane following system automatically reacts to ensure that the ego vehicle stays in its lane.

In this example, you achieve system-level simulation through integration with the Unreal Engine® from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['Unreal simulation is only supported on Microsoft', char(174), ' Windows', char(174),
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```

In the rest of the example, you will:

1. **Explore the test bench model**: Explore the functionalities in the system-level test bench model that you use to assess lane following with intelligent target vehicles.
2. **Vehicle behaviors**: Explore vehicle behaviors that you can use to model the intelligent target vehicles.
3. **Model the intelligent target vehicles**: Model the target vehicles in the scenario for three different behaviors: velocity keeping, lane following, and lane changing.
4. **Simulate lane following with intelligent target vehicles on a straight road**: Simulate velocity keeping, lane following, and lane change behaviors of a target vehicle while testing lane following on a straight road.

5 **Simulate lane following with intelligent target vehicles on a curved road**: Simulate velocity keeping, lane following, and lane change behaviors of a target vehicle while testing lane following on a curved road.

6 **Test with other scenarios**: Test the model with other scenarios available with this example.

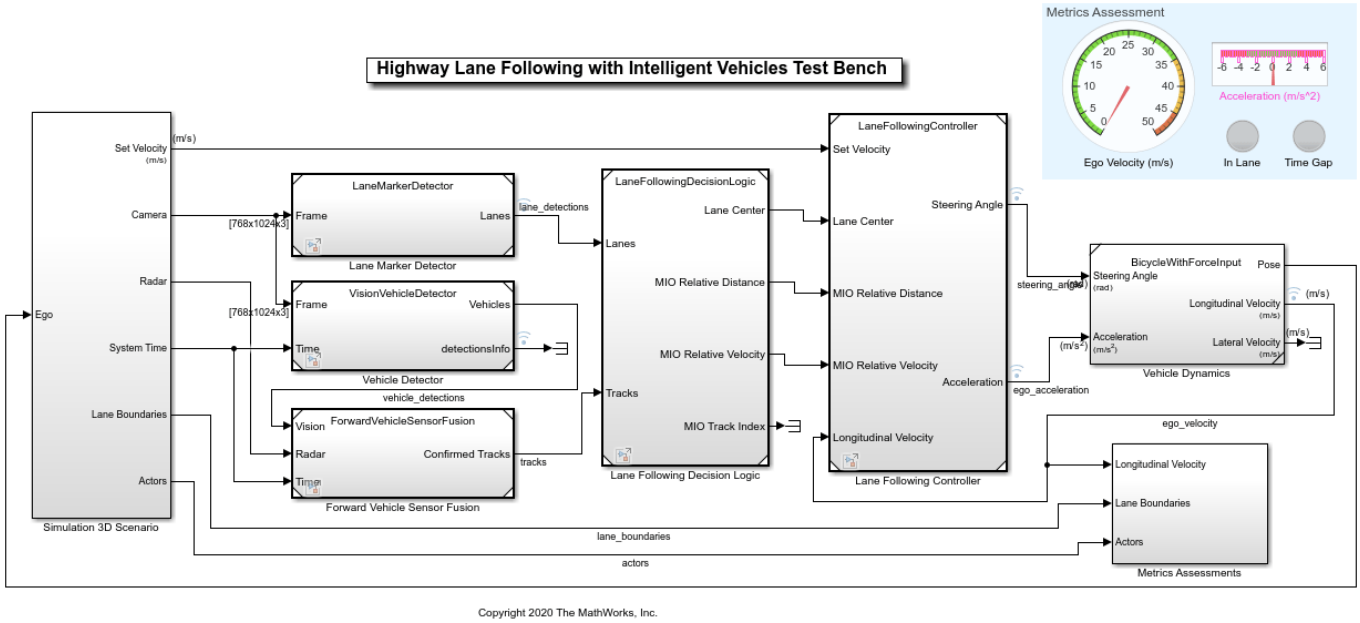
Explore Test Bench Model

To explore the test bench model, open a working copy of the project example files. MATLAB® copies the files to an example folder so that you can edit them.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('HLFIntelligentVehicles.zip', 'workDir', pwd);
```

Open the system-level simulation test bench model for the lane following application.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench")
```



The test bench model contains these modules:

1 **Simulation 3D Scenario**: Subsystem that specifies road, ego vehicle, intelligent target vehicles, camera, and radar sensors used for simulation.

2 **Lane Marker Detector**: Algorithm model to detect the lane boundaries in the frame captured by camera sensor.

3 **Vehicle Detector**: Algorithm model to detect to detect vehicles in the frame captured by camera sensor.

4 **Forward Vehicle Sensor Fusion**: Algorithm model that fuses the detections of vehicles in front of the ego vehicle that were obtained from vision and radar sensors.

5 **Lane Following Decision Logic**: Algorithm model that specifies lateral, longitudinal decision logic and provides lane center information and MIO related information to controller.

6 **Lane Following Controller**: Algorithm model that specifies the controls.

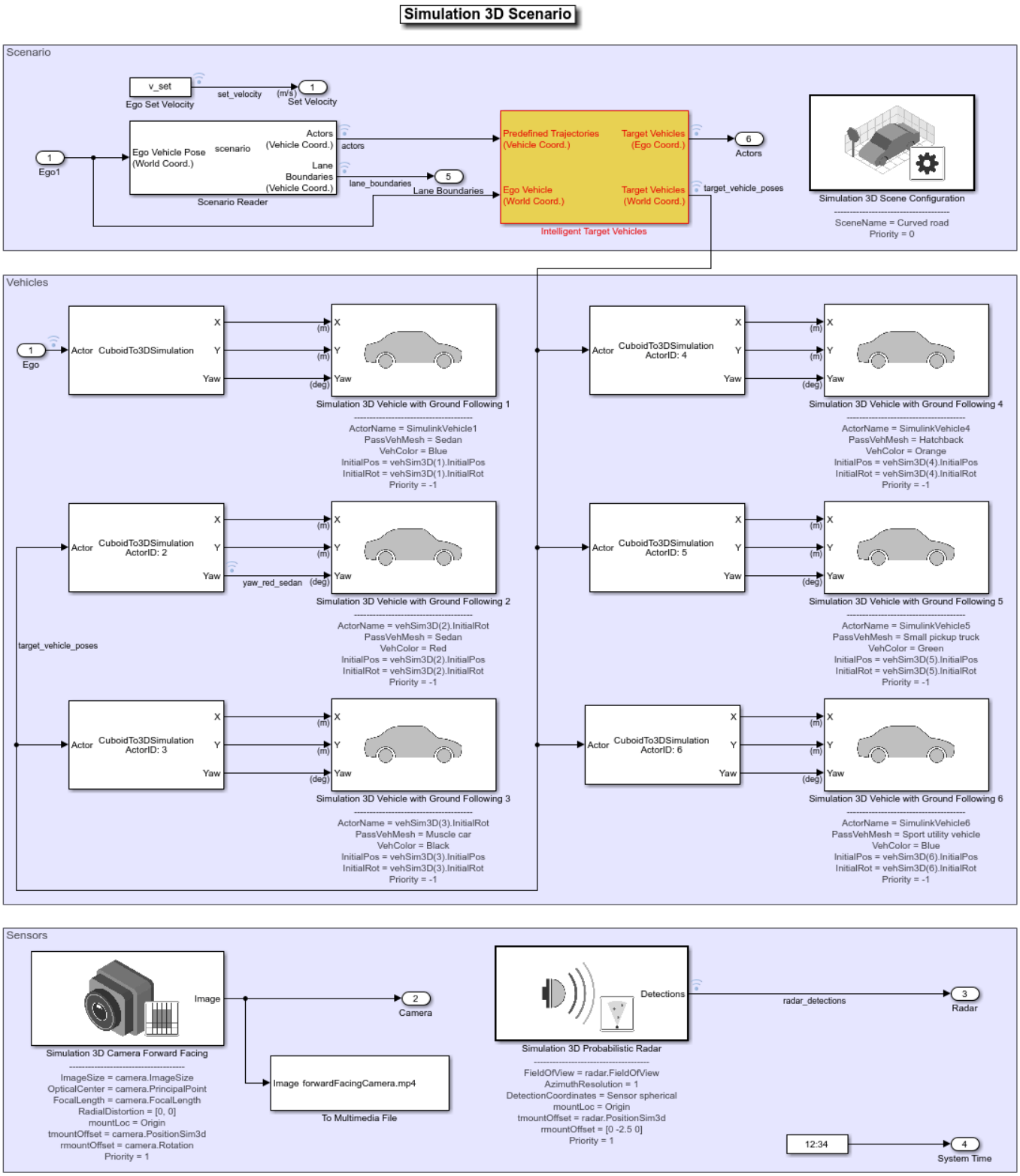
7 **Vehicle Dynamics**: Specifies the dynamics model for the ego vehicle.

8 **Metrics Assessment**: Assesses system-level behavior.

The Lane Marker Detector, Vehicle Detector, Forward Vehicle Sensor Fusion, Lane Following Decision Logic, Lane Following Controller, Vehicle Dynamics, and Metrics Assessment subsystems are based on the subsystems used in “Highway Lane Following” (Automated Driving Toolbox) (Automated Driving Toolbox). If you have license to Simulink® Coder™ and Embedded Coder™, you can generate deployable-ready embedded real-time code for the Lane Marker Detector, Vehicle Detector, Forward Vehicle Sensor Fusion, Lane Following Decision Logic, and Lane Following Controller algorithm models. This example focuses only on the **Simulation 3D Scenario** subsystem. An **Intelligent Target Vehicles** subsystem block is added to the **Simulation 3D Scenario** subsystem in order to configure the behavior of target vehicles in the scenario. The Lane Marker Detector, Vehicle Detector, Forward Vehicle Sensor Fusion, Lane Following Decision Logic, Lane Following Controller, Vehicle Dynamics, and Metrics Assessment subsystems steer the ego vehicle in response to the behavior of the target vehicles configured by the **Simulation 3D Scenario** subsystem.

Open the **Simulation 3D Scenario** subsystem and highlight the **Intelligent Target Vehicles** subsystem.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario")
hilite_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intel
```


Simulation 3D Scenario
Scenario
v_set
Ego Set Velocity
set_velocity
(m/s)
1
Set Velocity
1
Ego1
Ego Vehicle Pose
(World Coord.)
scenario
Actors
(Vehicle Coord.)
Lane
Boundaries
(Vehicle Coord.)
Scenario Reader
actors
lane_boundaries
5
Lane Boundaries
Predefined Trajectories
(Vehicle Coord.)
Target Vehicles
(Ego Coord.)
Ego Vehicle
(World Coord.)
Target Vehicles
(World Coord.)
Intelligent Target Vehicles
6
Actors
target_vehicle_poses
Simulation 3D Scene Configuration
SceneName = Curved road
Priority = 0
Vehicles
1
Ego
Actor
CuboidTo3DSimulation
X
Y
Yaw
(m)
(deg)
Simulation 3D Vehicle with Ground Following 1
ActorName = SimulinkVehicle1
PassVehMesh = Sedan
VehColor = Blue
InitialPos = vehSim3D(1).InitialPos
InitialRot = vehSim3D(1).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 4
Simulation 3D Vehicle with Ground Following 4
ActorName = SimulinkVehicle4
PassVehMesh = Hatchback
VehColor = Orange
InitialPos = vehSim3D(4).InitialPos
InitialRot = vehSim3D(4).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 2
yaw_red_sedan
Simulation 3D Vehicle with Ground Following 2
ActorName = vehSim3D(2).InitialRot
PassVehMesh = Sedan
VehColor = Red
InitialPos = vehSim3D(2).InitialPos
InitialRot = vehSim3D(2).InitialRot
Priority = -1
target_vehicle_poses
CuboidTo3DSimulation
ActorID: 5
Simulation 3D Vehicle with Ground Following 5
ActorName = SimulinkVehicle5
PassVehMesh = Small pickup truck
VehColor = Green
InitialPos = vehSim3D(5).InitialPos
InitialRot = vehSim3D(5).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 3
Simulation 3D Vehicle with Ground Following 3
ActorName = vehSim3D(3).InitialRot
PassVehMesh = Muscle car
VehColor = Black
InitialPos = vehSim3D(3).InitialPos
InitialRot = vehSim3D(3).InitialRot
Priority = -1
CuboidTo3DSimulation
ActorID: 6
Simulation 3D Vehicle with Ground Following 6
ActorName = SimulinkVehicle6
PassVehMesh = Sport utility vehicle
VehColor = Blue
InitialPos = vehSim3D(6).InitialPos
InitialRot = vehSim3D(6).InitialRot
Priority = -1
Sensors
Image
Simulation 3D Camera Forward Facing
ImageSize = camera.ImageSize
OpticalCenter = camera.PrincipalPoint
FocalLength = camera.FocalLength
RadialDistortion = [0, 0]
mountLoc = Origin
tmountOffset = camera.PositionSim3d
rmountOffset = camera.Rotation
Priority = 1
2
Camera
Image forwardFacingCamera.mp4
To Multimedia File
Detections
Simulation 3D Probabilistic Radar
FieldOfView = radar.FieldOfView
AzimuthResolution = 1
DetectionCoordinates = Sensor spherical
mountLoc = Origin
tmountOffset = radar.PositionSim3d
rmountOffset = [0 -2.5 0]
Priority = 1
radar_detections
3
Radar
12:34
4
System Time

The **Simulation 3D Scenario** subsystem configures the road network, models the target vehicles, sets vehicle positions, and synthesizes sensors. The subsystem is initialized by using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. This script defines the driving scenario for testing the highway lane following. This setup script defines the road network and sets the behavior for each target vehicle in the scenario.

- The Scenario Reader (Automated Driving Toolbox) block reads the roads and actors (ego and target vehicles) from a scenario file specified using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. The block outputs the poses of target vehicles and the lane boundaries with respect to the coordinate system of the ego vehicle.
- The **Intelligent Target Vehicles** is a function-call subsystem block that models the behavior of the actors in the driving scenario. The initial values for this subsystem block parameters are set by the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script. The Cuboid To 3D Simulation (Automated Driving Toolbox) and the Simulation 3D Vehicle with Ground Following (Automated Driving Toolbox) blocks set the actor poses for the 3D simulation environment.
- The Simulation 3D Scene Configuration (Automated Driving Toolbox) block implements a 3D simulation environment by using the road network and the actor positions.

This setup script also configures the controller design parameters, vehicle model parameters, and the Simulink® bus signals required for the `HighwayLaneFollowingWithIntelligentVehiclesTestBench` model. This script assigns an array of structures, `targetVehicles`, to the base workspace that contains the behavior type for each target vehicle.

Vehicle Behaviors

This example enables you to use four modes of vehicle behaviors for configuring the target vehicles using the `targetVehicles` structure.

- `Default`: In this mode, the target vehicles in the scenario follow predefined trajectories. The target vehicles are non-adaptive and are not configured for intelligent behavior.
- `VelocityKeeping`: In this mode, the target vehicles are configured to travel in a lane at a constant set velocity. Each target vehicle maintains the set velocity regardless of the presence of a lead vehicle in its current lane and does not check for collision.
- `LaneFollowing`: In this mode, the target vehicles are configured to travel in a lane by adapting their velocities in response to a lead vehicle. If a target vehicle encounters a lead vehicle in its current lane, the model performs collision checking and adjusts the velocity of the target vehicle. Collision checking ensures that the target vehicle maintains a safe distance from the lead vehicle.
- `LaneChange`: In this mode, the target vehicles are configured to travel in a lane at a particular velocity and follow the lead vehicle. If the target vehicle gets too close to the lead vehicle, then it performs a lane change. Before changing the lane, the model checks for potential forward and side collisions and adapts the velocity of the target vehicle to maintain a safe distance from other vehicles in the scenario.

Model Intelligent Target Vehicles

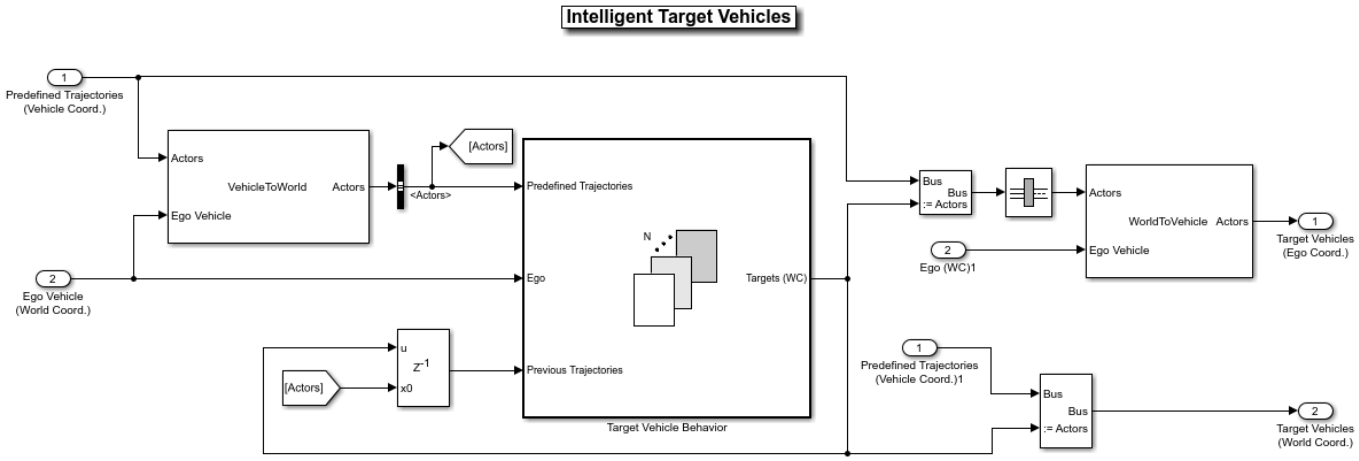
The **Intelligent Target Vehicles** subsystem dynamically updates the vehicle poses for all the target vehicles based on their predefined vehicle behavior. As mentioned already, the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` script defines the scenario and the behavior for each target vehicle in the scenario. The setup script stores the vehicle behavior

and other attributes as an array of structures, `targetVehicles`, to the base workspace. The structure stores these attributes:

- `ActorID`
- `Position`
- `Velocity`
- `Roll`
- `Pitch`
- `Yaw`
- `AngularVelocity`
- `InitialLaneID`
- `BehaviorType`

The **Intelligent Target Vehicles** subsystem uses a mask to load the configuration in `targetVehicles` from the base workspace. You can set the values of these attributes to modify the position, orientation, velocities, and behavior of target vehicles. Open the **Intelligent Target Vehicles** subsystem.

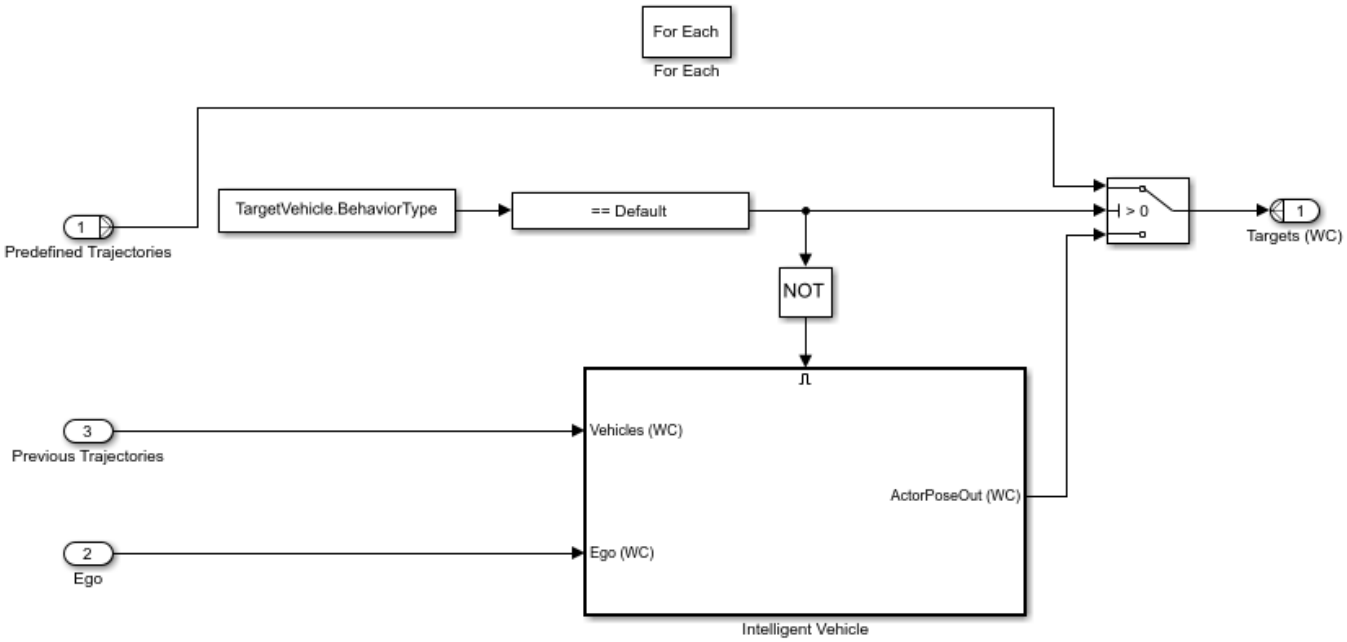
```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The Vehicle To World (Automated Driving Toolbox) block converts the predefined actor (ego and target vehicle) poses and trajectories from ego-vehicle coordinates to world coordinates. The **Target Vehicle Behavior** subsystem block computes the next state of the target vehicles by using predefined target vehicles poses, ego-vehicle pose, and current state of target vehicles. The subsystem outputs the target vehicles poses in world coordinates for navigating the vehicles in the 3D simulation environment.

Open the **Target Vehicle Behavior** subsystem.

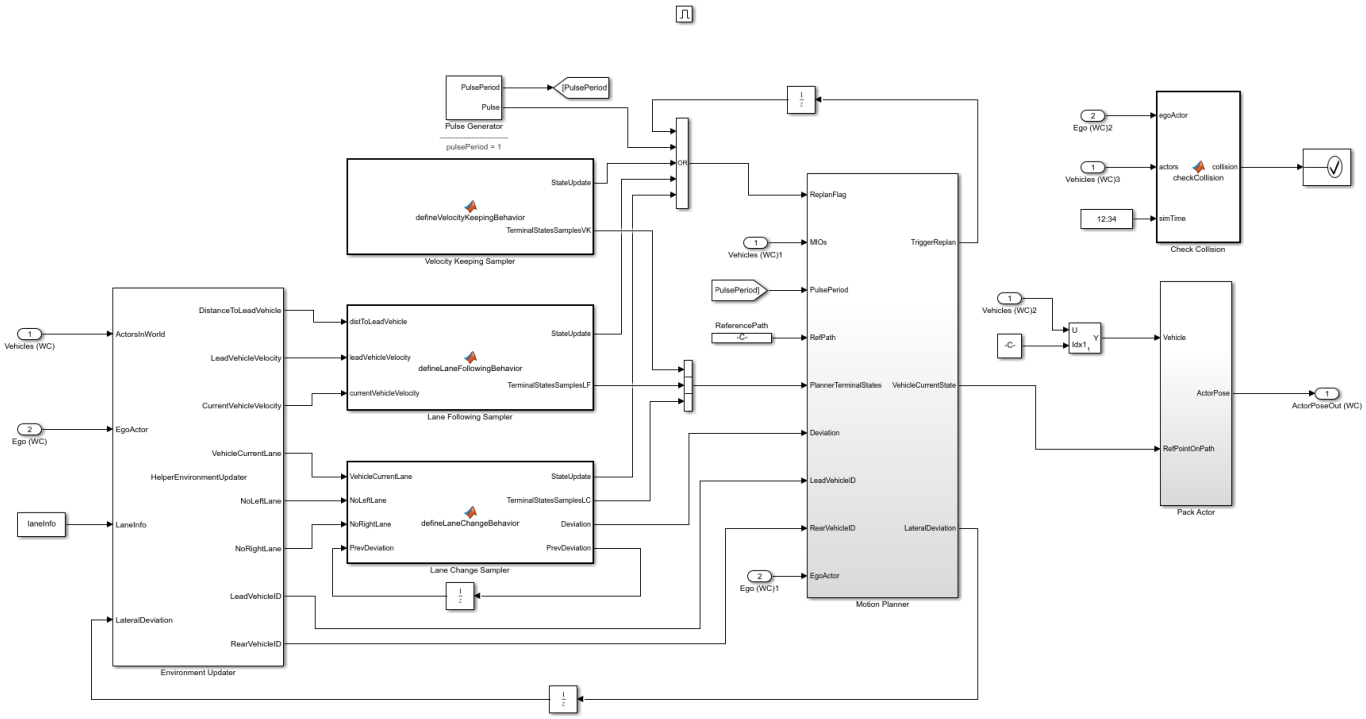
```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The **Target Vehicle Behavior** subsystem enables you to switch between the default and other vehicle behaviors. If the behavior type for a target vehicle is set to `Default`, the subsystem configures the target vehicles to follow predefined trajectories. Otherwise, the position of the vehicle is dynamically computed and updated using the **Intelligent Vehicle** subsystem block. The **Intelligent Vehicle** subsystem block configures the `VelocityKeeping`, `LaneFollowing`, and `LaneChange` behaviors for the target vehicles.

Open **Intelligent Vehicle** subsystem.

```
open_system("HighwayLaneFollowingWithIntelligentVehiclesTestBench/Simulation 3D Scenario/Intelli
```



The **Intelligent Vehicle** subsystem computes the pose of a target vehicle by using information about the neighboring vehicles and the vehicle behavior. The subsystem is similar to the **Lane Change Planner** component of the “Highway Lane Change” (Automated Driving Toolbox) example. The **Intelligent Vehicle** subsystem has these blocks:

- The **Environment Updater** block computes the lead and rear vehicle information, current lane number, and existence of adjacent lanes (`NoLeftLane`, `NoRightLane`) with respect to the current state of the target vehicle. This block is configured by the System object™ `HelperEnvironmentUpdater`.
- The **Velocity Keeping Sampler** block defines terminal states required for the `VelocityKeeping` behavior. This block reads the set velocity from the mask parameter `norm(TargetVehicle.Velocity)`.
- The **Lane Following Sampler** block defines terminal states required for the `LaneFollowing` behavior. This block reads the set velocity from the mask parameter `norm(TargetVehicle.Velocity)`.
- The **Lane Change Sampler** block defines terminal states required for the `LaneChange` behavior. This block also defines deviation offset from the reference path to keep the vehicle in a specific lane after a lane change. This block reads `TargetVehicle.Velocity`, `laneInfo`, and `TargetVehicle.InitialLaneID` from the base workspace by using mask parameters.

The table shows the configuration of terminal states and parameters for different vehicle behaviors:

Vehicle Behavior	Longitudinal States	Lateral States	Velocity States
Velocity Keeping	Constant set to 60 meters	Constant set to 0, to travel in the current lane	Configurable constant
Lane Following	Varies based on distance to the lead vehicle in the current lane of the vehicle	Constant set to 0, to travel in the current lane	Varies based on velocity of the lead vehicle in the current lane of the vehicle
Lane Change	Constant set to 60 meters	Varies based on the current lane of the lead vehicle and lane geometry	Configurable constant

- The **Check Collision** block checks for collision with any other vehicle in the scenario. The simulation stops if collision is detected.
- The **Pulse Generator** block defines the replan period for the **Motion Planner** subsystem. The default value is set to 1 second. Replanning can be triggered every pulse period, or if any of the samplers has a state update, or by the **Motion Planner** subsystem.
- The **MotionPlanner** subsystem generates trajectory for a target vehicle by using the terminal states defined by the vehicle behavior. It uses `trajectoryOptimalFrenet` (Navigation Toolbox) from Navigation Toolbox™ to generate a trajectory. The subsystem estimates the position of the vehicle along its trajectory at every simulation step. This subsystem internally uses the `HelperTrajectoryPlanner` System object™ to implement a fallback mechanism for different vehicle behaviors when the `trajectoryOptimalFrenet` function is unable to generate a feasible trajectory.
- If the vehicle behavior is set to `LaneChange`, the trajectory planner attempts to generate a trajectory with `LaneFollowing` behavior. If it is unable to generate a trajectory, then it stops the vehicle using its stop behavior.
- If the vehicle behavior is set to `LaneFollowing` or `VelocityKeeping`, the trajectory planner stops the vehicle using stop behavior.

The system implements the stop behavior by constructing a trajectory with the previous state of the vehicle, which results in an immediate stop of the target vehicle.

Simulate Intelligent Target Vehicle Behavior on Straight Road

This example uses a test scenario that has three target vehicles (red sedan, black muscle car, and orange hatchback) and one ego vehicle (blue sedan) traveling on a straight road with two lanes.

- The red sedan is the first target vehicle and travels in the lane adjacent to the ego lane.
- The orange hatchback is a lead vehicle for the ego vehicle in the ego lane.
- The black muscle car is slow moving and a lead vehicle for the red sedan in the adjacent lane of the ego vehicle. The figure shows the initial positions of these vehicles.

You can run the simulation any number of times by changing the behavior type for each vehicle during each run. This example runs the simulation three times and at each run the behavior type for the first target vehicle is modified.

Configure All Target Vehicles Behavior to Velocity Keeping and Run Simulation

Run the setup script to configure `VelocityKeeping` behavior for all target vehicles.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup(...
    "scenarioFcnName",...
    "scenario_LFACC_01_Straight_IntelligentVelocityKeeping");
```

Display the `BehaviorType` of all the target vehicles.

```
disp([targetVehicles(:).BehaviorType]');
```

```
    VelocityKeeping
    VelocityKeeping
    VelocityKeeping
    Default
    Default
```

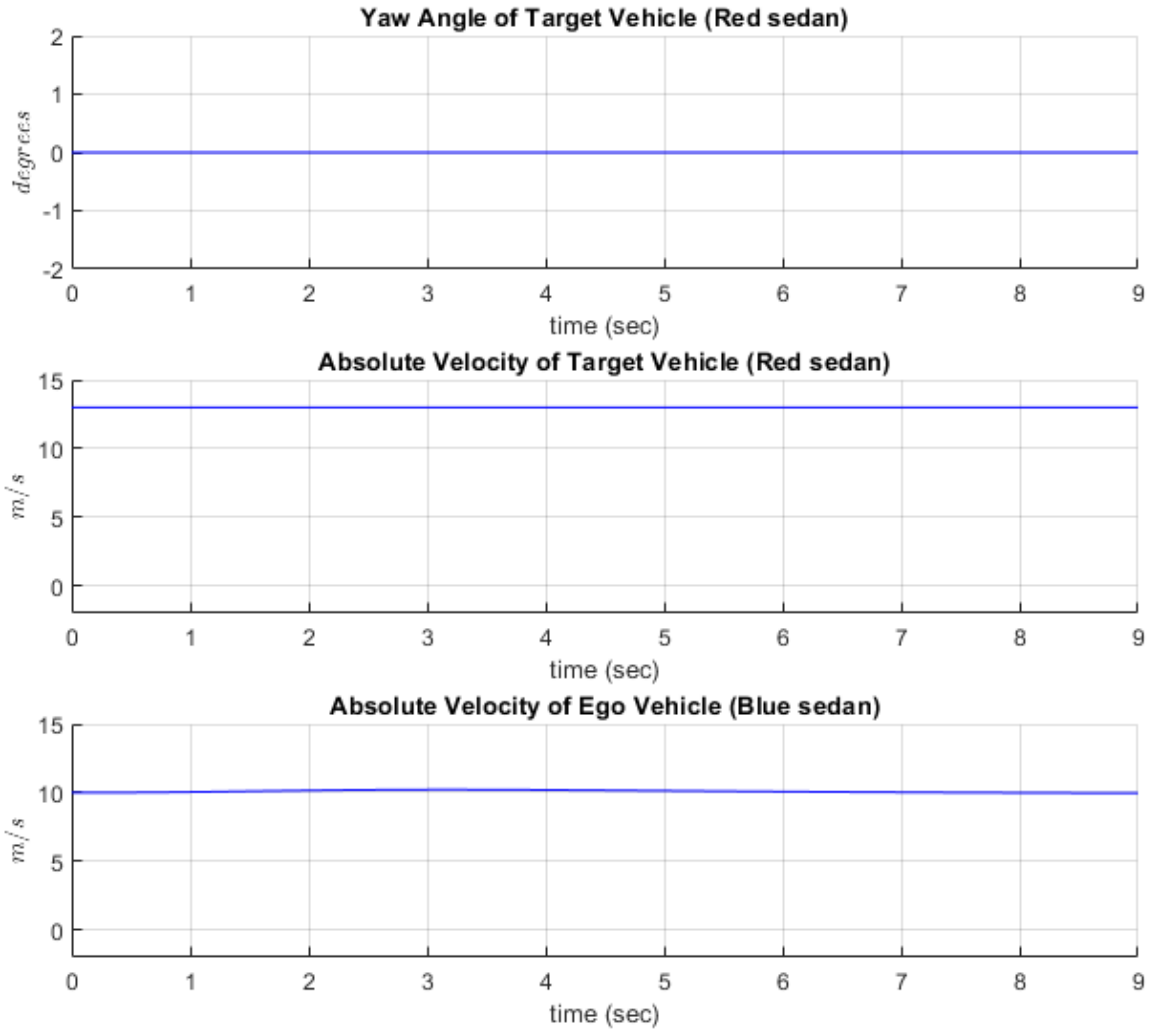
Run the simulation and visualize the results. The target vehicles in the scenario travel in their respective lanes at a constant velocity. The red sedan and the black muscle car maintain their velocity and do not check for collisions.

To reduce command-window output, turn off the model predictive control (MPC) update messages.

```
mpcverbosity('off');
% Run the model
simout = sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench","StopTime","9");
```

Plot the velocity profiles of ego and first target vehicle (red sedan) to analyze the results.

```
hFigVK = helperPlotEgoAndTargetVehicleProfiles(simout.logsout);
```



Close figure

```
close(hFigVK);
```

- The **Yaw Angle of Target Vehicle (Red sedan)** plot shows the yaw angle of the red sedan. There is no variation in the yaw angle as the vehicle travels on a straight lane road.

- The **Absolute Velocity of Target Vehicle (Red sedan)** plot shows the absolute velocity of the red sedan. The velocity profile of the vehicle is constant as the vehicle is configured to `VelocityKeeping` behavior.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows that there is no effect of the red sedan on the ego vehicle as both vehicles travel in adjacent lanes.

Configure First Target Vehicle Behavior to Lane Following and Run Simulation

Configure the behavior type for the first target vehicle (red sedan) to perform lane following. Display the updated values for the `BehaviorType` of target vehicles.

```
targetVehicles(1).BehaviorType = VehicleBehavior.LaneFollowing;
disp([targetVehicles(:).BehaviorType]');
```

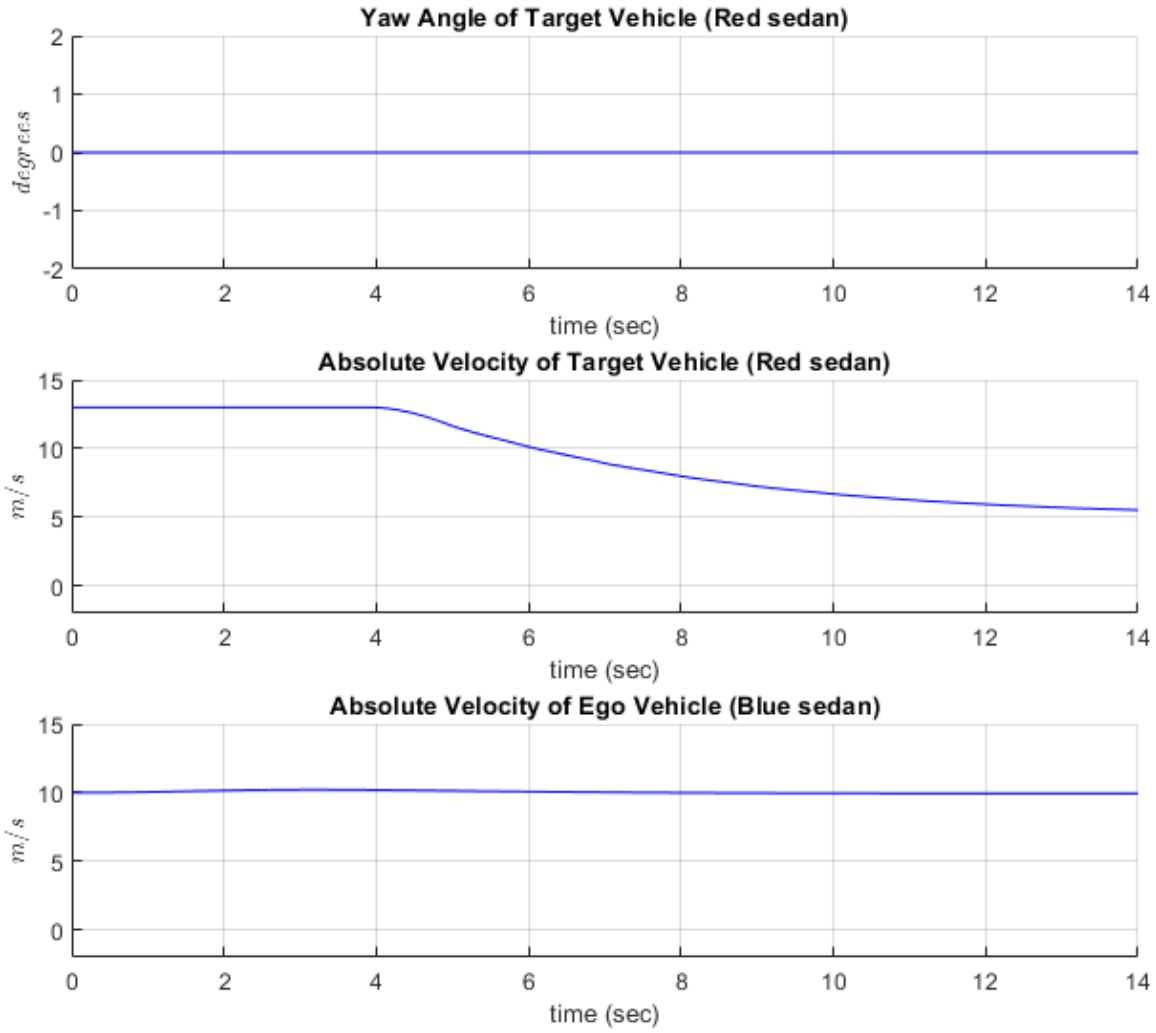
```
    LaneFollowing
    VelocityKeeping
    VelocityKeeping
    Default
    Default
```

Run the simulation and visualize the results. The target vehicles in the scenario are traveling in their respective lanes. The first target vehicle (red sedan) slows down to avoid colliding with the slow-moving black muscle car in its lane.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
```

Plot the velocity profiles of the ego and first target vehicle (red sedan) to analyze the results.

```
hFigLF = helperPlotEgoAndTargetVehicleProfiles(logsout);
```



Close figure

```
close(hFigLF);
```

- The **Yaw Angle of Target Vehicle (Red sedan)** plot is the same as the one obtained in the previous simulation. There is no variation in the yaw angle as the vehicle travels on a straight lane road.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot diverges from the previous simulation. The velocity of the red sedan gradually decreases from 13 m/s to 5 m/s to avoid colliding with the black muscle car and maintains a safety gap.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot is same as the one in the previous simulation. The ego vehicle is not affected by the change in the behavior of the red sedan.

Configure First Target Vehicle Behavior to Lane Changing and Run Simulation

```
targetVehicles(1).BehaviorType = VehicleBehavior.LaneChange;
```

Display the `BehaviorType` of all the target vehicles.

```
disp([targetVehicles(:).BehaviorType]');
```

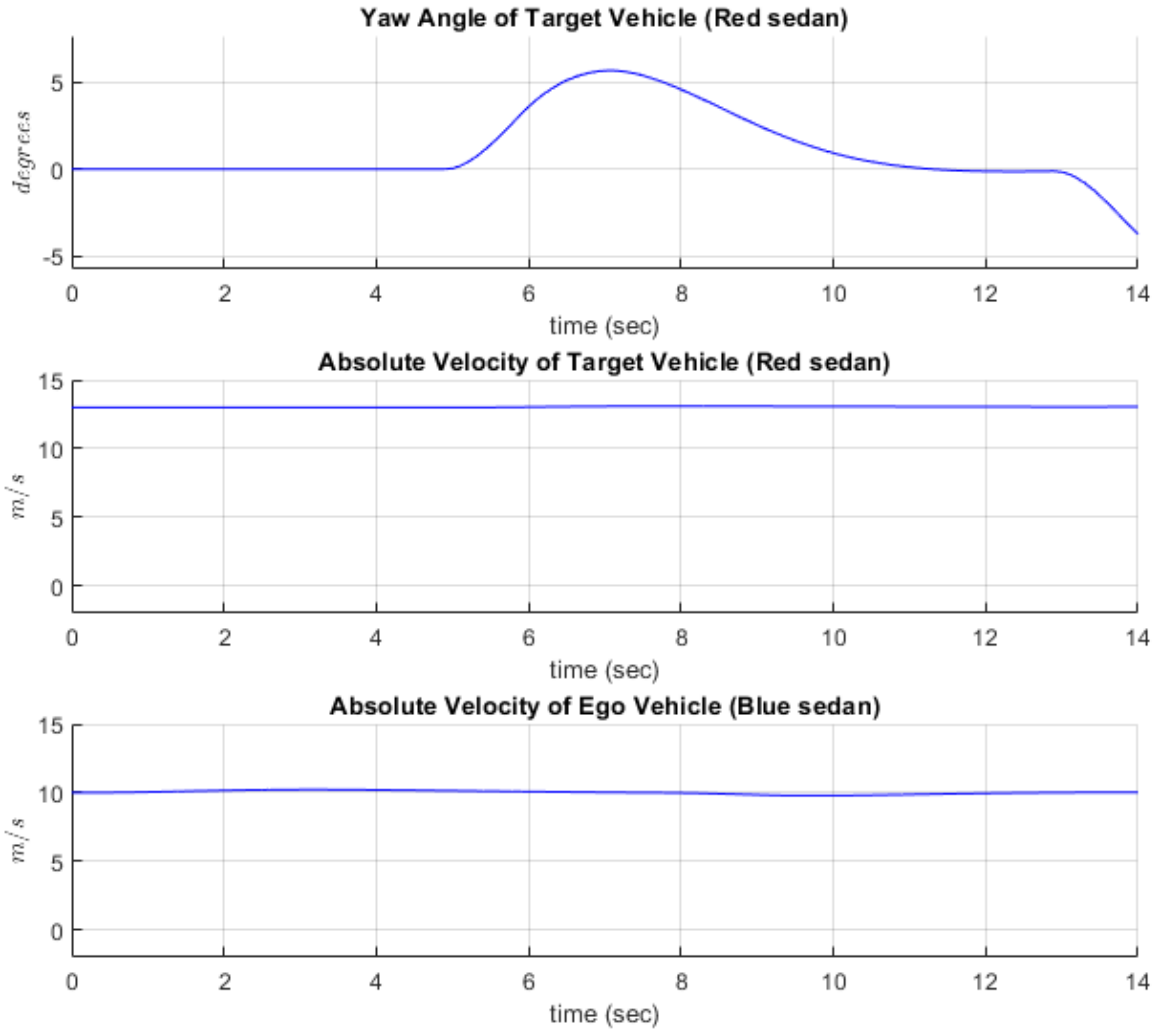
```
    LaneChange
    VelocityKeeping
    VelocityKeeping
    Default
    Default
```

Run the simulation and visualize the results. The orange hatchback and black muscle car are traveling at constant velocity in their respective lanes. The first target vehicle (red sedan) performs a lane change as it gets close to the black muscle car. It also does another lane change when it gets close to the orange hatchback.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
```

Plot the velocity profiles of the ego and the first target vehicle (red sedan) to analyze the results.

```
hFigLC = helperPlotEgoAndTargetVehicleProfiles(logsout);
```



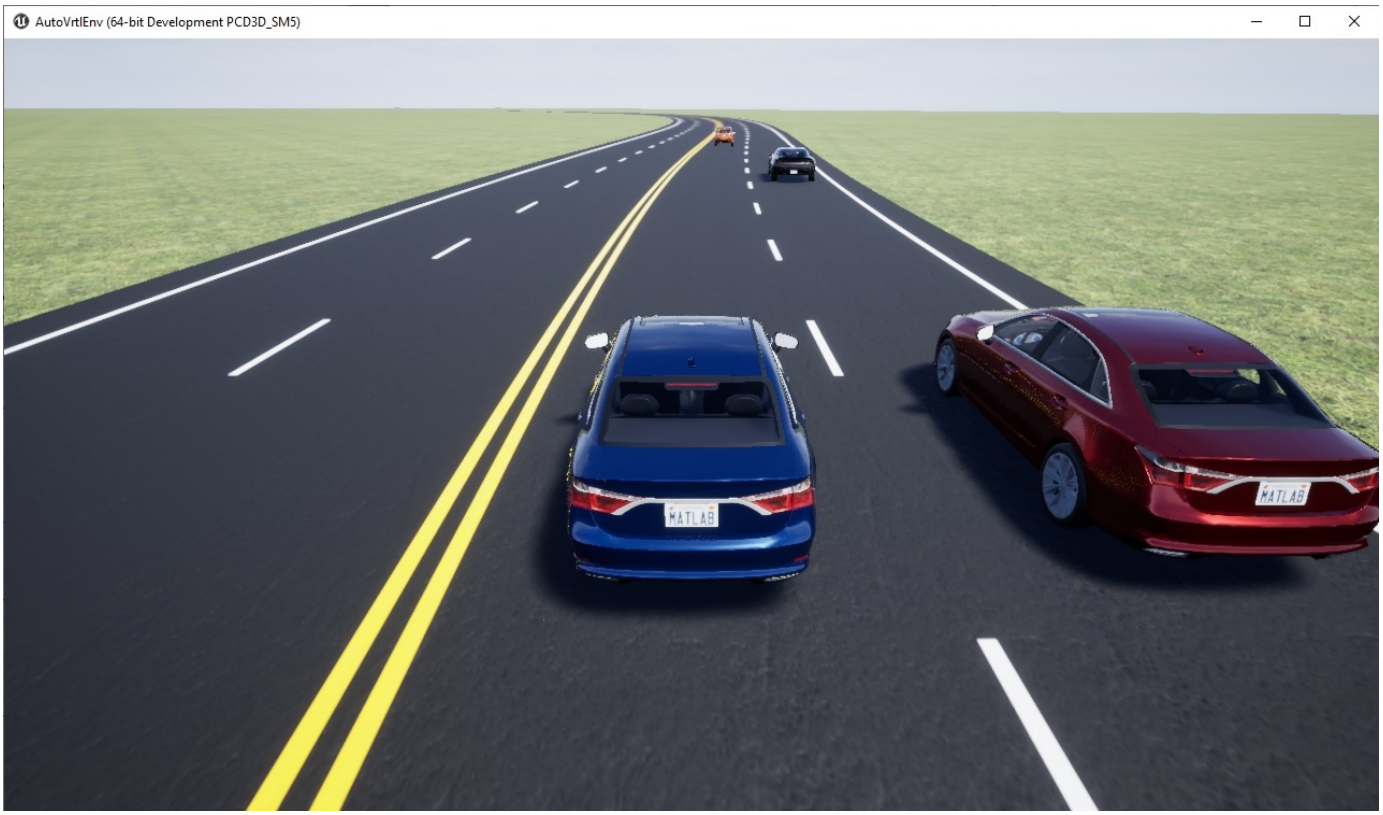
Close figure

```
close(hFigLC);
```

- The **Yaw Angle of Target Vehicle (Red sedan)** plot diverges from the previous simulation results. The yaw angle profile of the first target vehicle shows deviations as the vehicle performs a lane change.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot is similar to the `VelocityKeeping` behavior. The red sedan maintains a constant velocity even during the lane change.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows the ego vehicle response to the lane change maneuver by the first target vehicle (red sedan). The velocity of the ego vehicle decreases as the red sedan changes lanes. The red sedan moves to the ego lane and travels in front of the ego vehicle. The ego vehicle reacts by decreasing its velocity in order to travel in the same lane. Close all the figures.

Simulate Intelligent Target Vehicle Behavior on Curved Road

Test the model on a scenario with curved roads. The vehicle configuration and position of vehicles are similar to the previous simulation. The test scenario contains a curved road and the first target vehicle (Red sedan) is configured to `LaneChange` behavior. The other two target vehicles are configured to `VelocityKeeping` behavior. The figure below shows the initial positions of the vehicles in the curved road scene.

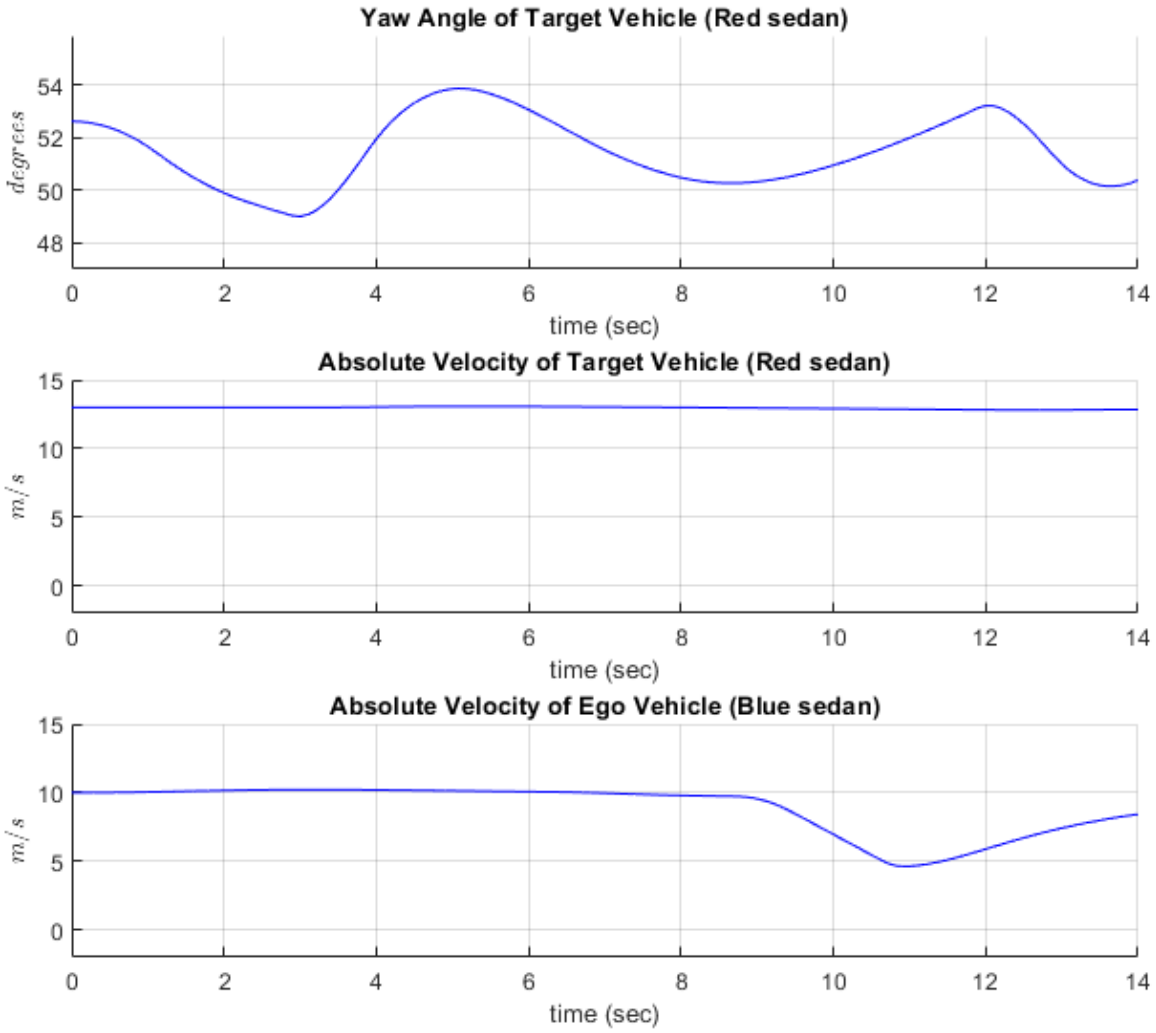


Run the setup script to configure the model parameters.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup(...
    "scenarioFcnName",...
        "scenario_LFACC_04_Curved_IntelligentLaneChange");
```

Run simulation and visualize the results. Plot the yaw angle and velocity profiles of ego and target vehicles.

```
sim("HighwayLaneFollowingWithIntelligentVehiclesTestBench");
hFigCurvedLC = helperPlotEgoAndTargetVehicleProfiles(logsout);
```



- The **Yaw Angle of Target Vehicle (Red sedan)** plot shows variation in the profile as the red sedan performs lane change on a curved road. The curvature of the road also impacts the yaw angle of the target vehicle.
- The **Absolute Velocity of Target Vehicle (Red sedan)** plot is similar to the `VelocityKeeping` behavior, as the red sedan maintains a constant velocity during lane change on a curved road.
- The **Absolute Velocity of Ego Vehicle (Blue sedan)** plot shows the response of the ego vehicle to the lane change maneuver by the red sedan. The ego vehicle reacts by decreasing its velocity in order to travel in the same lane.

Close the figure.

```
close(hFigCurvedLC);
```

Explore Other Scenarios

This example provides additional scenarios that are compatible with the `HighwayLaneFollowingWithIntelligentVehiclesTestBench` model. Below is a list of compatible scenarios that are provided with this example.

- `scenario_LFACC_01_Straight_IntelligentVelocityKeeping` function configures the test scenario such that all the target vehicles are configured to perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_02_Straight_IntelligentLaneFollowing` function configures the test scenario such that the red sedan performs `LaneFollowing` behavior while all other target vehicles perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_03_Straight_IntelligentLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a straight road.
- `scenario_LFACC_04_Curved_IntelligentLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a curved road. This is configured as the default scenario.
- `scenario_LFACC_05_Curved_IntelligentDoubleLaneChange` function configures the test scenario such that the red sedan performs `LaneChange` behavior while all other target vehicles perform `VelocityKeeping` behavior on a curved road. The placement of other vehicles in this scenario is such that the red sedan performs a double lane change during the simulation.

For more details on the road and target vehicle configurations in each scenario, view the comments in each file. You can configure the Simulink model and workspace to simulate these scenarios using the `helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup` function.

```
helperSLHighwayLaneFollowingWithIntelligentVehiclesSetup("scenarioFcnName","scenario_LFACC_05_Cu
```

Conclusion

This example demonstrates how to test the functionality of a lane following application in a scenario with an ego vehicle and multiple intelligent target vehicles.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

See Also

More About

- “Highway Lane Following” on page 11-62
- “Automate Testing for Highway Lane Following” on page 11-89
- “Automated Driving Using Model Predictive Control” on page 11-2

Parking Valet Using Nonlinear Model Predictive Control

This example shows how to generate a reference trajectory and track the trajectory for a parking valet using nonlinear model predictive control (NLMPC).

Parking Garage

In this example, the parking garage contains an ego vehicle and eight static obstacles. The obstacles are given by six parked vehicles, a reserved parking area, and the garage border. The goal of the ego vehicle is to park at a target pose without colliding with any of the obstacles. The reference point of the ego pose is located at the center of the rear axle.

Define the parameters of the ego vehicle.

```
vdims = vehicleDimensions;
egoWheelbase = vdims.Wheelbase;
distToCenter = 0.5*egoWheelbase;
```

Specify the initial ego vehicle pose.

```
% Ego initial pose: x(m), y(m) and yaw angle (rad)
egoInitialPose = [4,12,0];
```

Define the target pose for the ego vehicle. In this example, there are two possible parking directions. To park facing north, set `parkNorth` to `true`. To park facing south, set `parkNorth` to `false`.

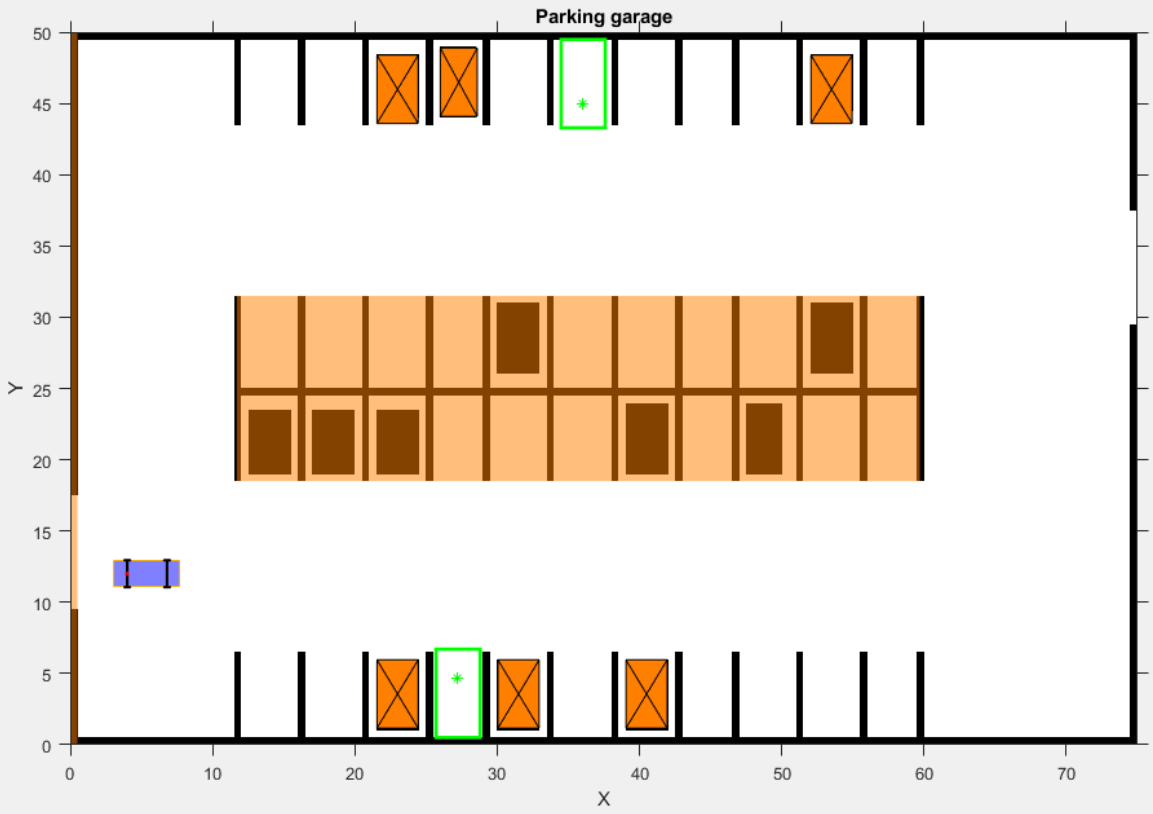
```
parkNorth = true;
if parkNorth
    egoTargetPose = [36,45,pi/2];
else
    egoTargetPose = [27.2,4.7,-pi/2];
end
```

The `helperSLCreateCostmap` function creates a static map of the parking lot that contains information about stationary obstacles, road markings, and parked cars. For more details, see the "Automated Parking Valet in Simulink" (Automated Driving Toolbox) example.

```
costmap = helperSLCreateCostmap();
centerToFront = distToCenter;
centerToRear = distToCenter;
helperSLCreateUtilityBus;
costmapStruct = helperSLCreateUtilityStruct(costmap);
```

Visualize the parking environment. Use a sample time of `0.1` for the visualizer.

```
Tv = 0.1;
helperSLVisualizeParkingValet(egoInitialPose, 0, costmapStruct);
```



The six parked vehicles are orange boxes on the top and bottom of the figure. The middle area represents the reserved parking area. The left border of the garage is also modeled as a static obstacle. The ego vehicle in blue has two axles and four wheels. The two green boxes represent the target parking spots for the ego vehicle, with the top spot facing north.

Generate a Trajectory Using Nonlinear Model Predictive Controller

In this example, a kinematic bicycle model with front steering angle is used. The motion of the ego vehicle can be described by the following equations.

$$\dot{x} = v \cdot \cos(\psi)$$
$$\dot{y} = v \cdot \sin(\psi)$$
$$\dot{\psi} = \frac{v}{b} \cdot \tan(\delta)$$

where (x, y) denotes the position of the vehicle and ψ denotes the yaw angle of the vehicle. The parameter b represents the wheelbase of the vehicle. (x, y, ψ) are the state variables of the vehicle state functions. The speed v and steering angle δ are the control variables of the vehicle state functions.

The parking valet trajectory from the NLMPC controller for is designed based on the analysis similar to “Parallel Parking Using Nonlinear Model Predictive Control” on page 11-126 example. The design of controller is implemented in the `createMPCForParkingValet` script.

- The speed of the ego vehicle is constrained to be within [-6.5,6.5] m/s (approximately with speed limit as 15 mph) and the steering angle of the ego vehicle is constrained to be within [-45,45] degrees.
- The cost function for `nlmpc` controller object is a custom cost function defined in a manner similar to a quadratic tracking cost plus a terminal cost. In the following custom cost function, $s(t)$ denotes the states of ego vehicle at time t, d represents the duration of simulation. s_{ref} is given by the target pose for the ego vehicle. The matrices Q_p, R_p, Q_t, and R_t are constant.

$$J = \int_0^d (s(t) - s_{ref})^T Q_p (s(t) - s_{ref}) + u(t)^T R_p u(t) dt + (s(d) - s_{ref})^T Q_t (s(d) - s_{ref}) + u(d)^T R_t u(d)$$

- To avoid collision with obstacles, the NLMPC controller must satisfy the following inequality constraints, where minimum distance to all obstacles $dist_{min}$ must be greater than a safe distance $dist_{safe}$. In this example, the ego vehicle and obstacles are modeled as `collisionBox` (Robotics System Toolbox) objects and the distance from the ego vehicle to obstacles is computed by the `checkCollision` (Robotics System Toolbox) function.

$$dist_{min} \geq dist_{safe}$$

- The initial guess for the solution path is given by two straight lines. The first line is from the initial ego vehicle pose to a middle point, and the second line is from the middle point to the ego vehicle target pose.

Select a middle point for the initial solution path guess.

```
if parkNorth
    midPoint = [4,34,pi/2];
else
    midPoint = [27,12,0];
end
```

Configure the parameters of the NLMPC controller. To plan an optimal trajectory over the entire prediction horizon, set the control horizon equal to the prediction horizon.

```
% Sample time
Ts = 0.1;
% Prediction horizon
p = 100;
% Control horizon
c = 100;
% Weight matrices for terminal cost
Qt = 0.5*diag([10 5 20]);
Rt = 0.1*diag([1 2]);
% Weight matrices for tracking cost
if parkNorth
    Qp = 1e-6*diag([2 2 0]);
    Rp = 1e-4*diag([1 15]);
else
    Qp = 0*diag([2 2 0]);
    Rp = 1e-2*diag([1 5]);
end
% Safety distance to obstacles (m)
safetyDistance = 0.1;
% Maximum iteration number
maxIter = 70;
```

```
% Disable message display
mpcverbosity('off');
```

Create the NLMPC controller using the specified parameters.

```
[nlobj,opt,paras] = createMPCForParkingValet(p,c,Ts,egoInitialPose,egoTargetPose,...
    maxIter,Qp,Rp,Qt,Rt,distToCenter,safetyDistance,midPoint);
```

Set the initial conditions for the ego vehicle.

```
x0 = egoInitialPose';
u0 = [0;0];
```

Generate the reference trajectory using the `nlmpcmove` function.

```
tic;
[mv,nloptions,info] = nlmpcmove(nlobj,x0,u0,[],[],opt);
timeVal = toc;
```

Obtain the reference trajectories for the states (`xRef`) and the control actions (`uRef`), which are the optimal trajectories computed of the prediction horizon.

```
xRef = info.Xopt;
uRef = info.MVopt;
```

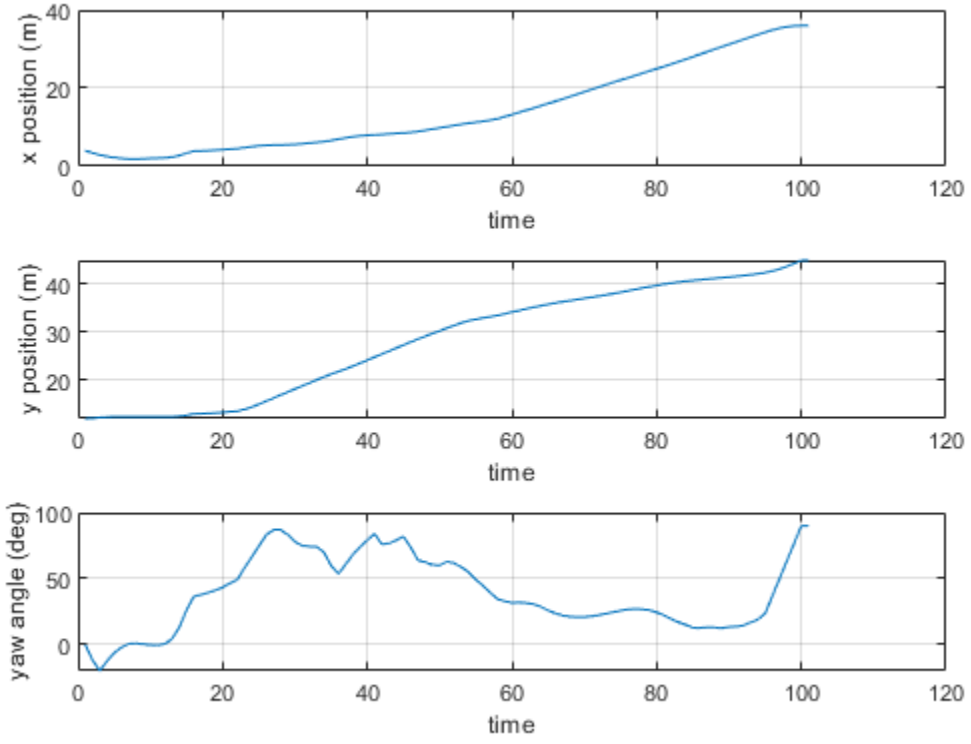
Analyze the planned trajectory.

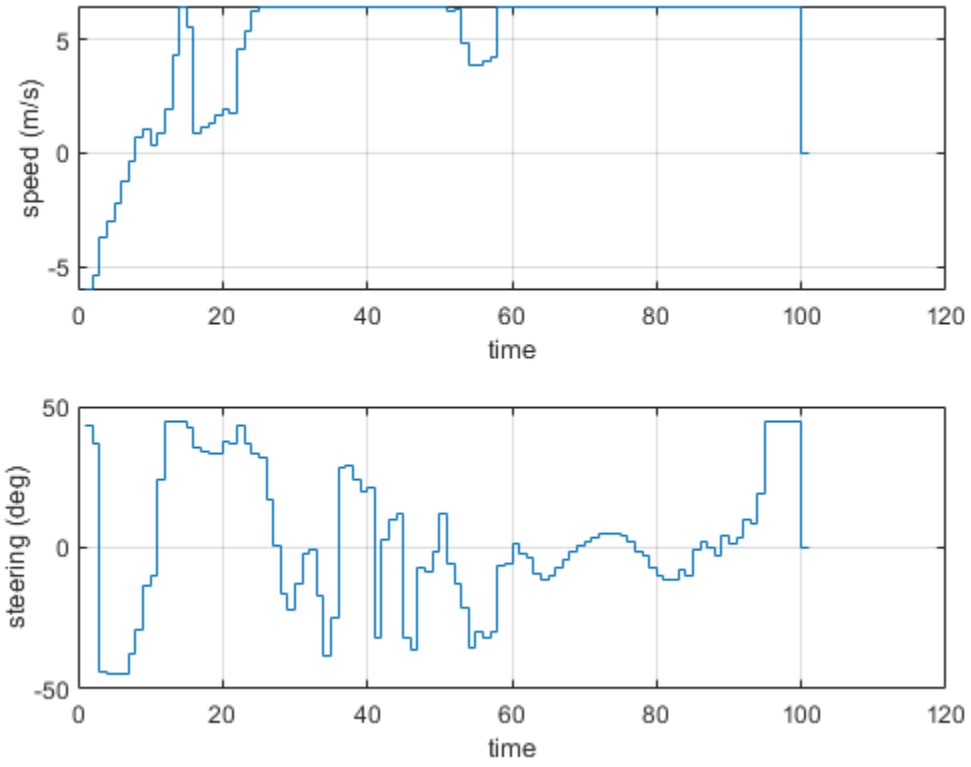
```
analyzeParkingValetResults(nlobj,info,egoTargetPose,Qp,Rp,Qt,Rt,...
    distToCenter,safetyDistance,timeVal)
```

```
Summary of results:
1) Valid results. No collisions.
2) Minimum distance to obstacles = 0.1184 (Valid when greater than safety distance 0.1000)
3) Optimization exit flag = 1 (Successful when positive)
4) Elapsed time (s) for nlmpcmove = 84.2429
5) Final states error in x (m), y (m) and theta (deg):  0.0018, 0.0023, 0.0050
6) Final control inputs speed (m/s) and steering angle (deg): 0.0212, -0.3379
```

As shown in the following plots, the planned trajectory successfully parks the ego vehicle in the target pose. The final control input values are close to zero.

```
plotTrajectoryParkingValet(xRef,uRef)
```


x position (m)
0
20
40
0
20
40
60
80
100
120
time
y position (m)
20
30
40
0
20
40
60
80
100
120
time
yaw angle (deg)
0
50
100
0
20
40
60
80
100
120
time



Track Reference Trajectory in Simulink Model

Design an NLMPC controller to track the reference trajectory.

First, set the simulation duration and update the reference trajectory based on the duration.

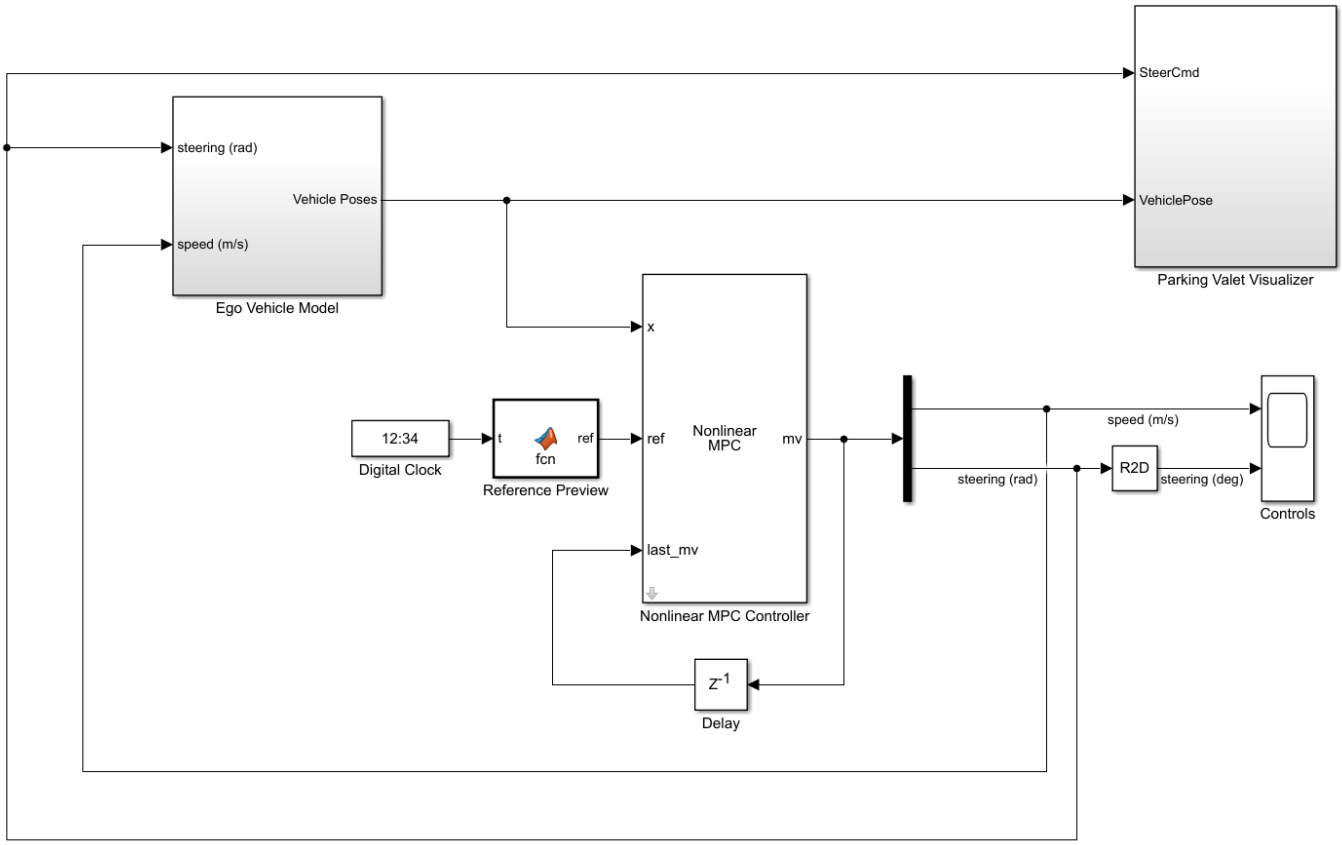
```
Duration = 12;
Tsteps = Duration/Ts;
Xref = [xRef(2:p+1,:);repmat(xRef(end,:),Tsteps-p,1)];
```

Create an NLMPC controller with a tracking prediction horizon (`pTracking`) of `10`.

```
pTracking = 10;
nlobjTracking = createMPCForTrackingParkingValet(pTracking,Xref);
```

Open the Simulink model.

```
mdl = 'mpcAutoParkingValet';
open_system(mdl)
```

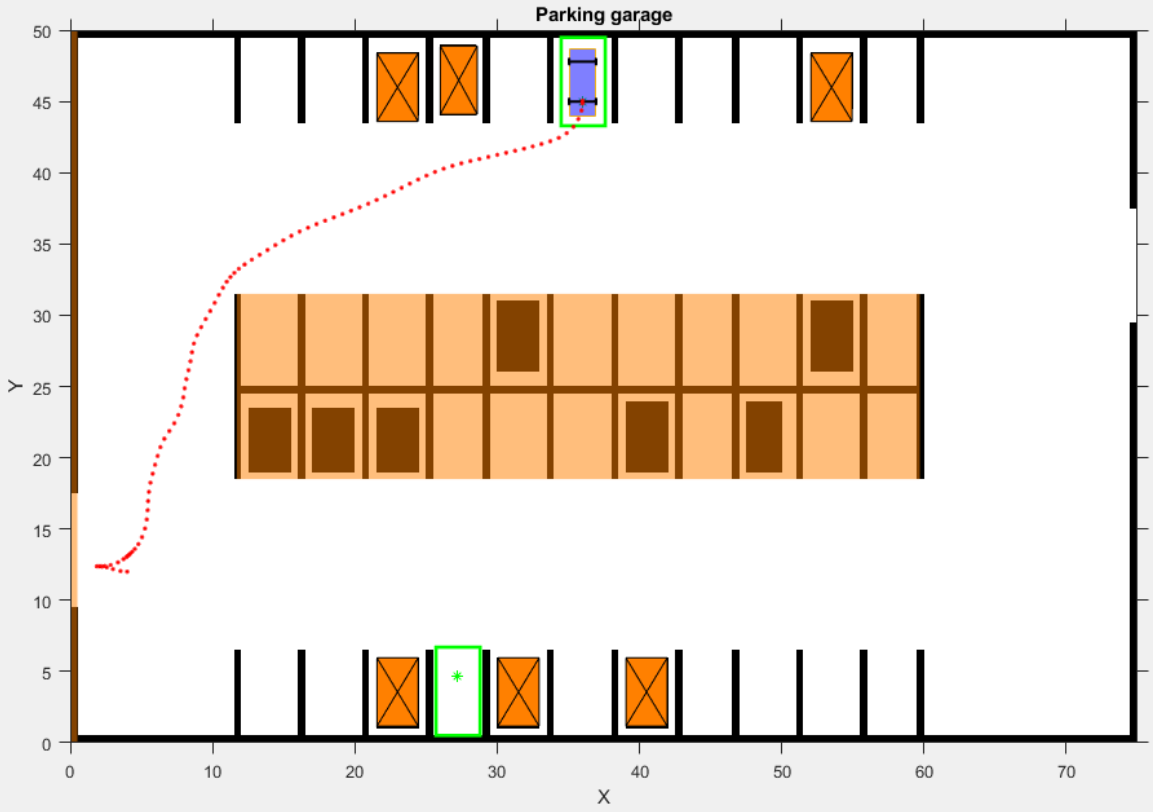


Close the animation plots before running the simulation.

```
f = findobj('Name','Automated Parking Valet');
close(f)
```

Simulate the model.

```
sim(mdl)
```




```
ans =
  Simulink.SimulationOutput:

                tout: [125x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```

The animation shows that the ego vehicle parks at the target pose successfully without any obstacle collisions. You can also view the ego vehicle and pose trajectories using the Ego Vehicle Pose and Controls scopes.

Conclusion

This example shows how to generate a reference trajectory and track the trajectory for parking valet using nonlinear model predictive control. The controller navigates the ego vehicle to the target parking spot without colliding with any obstacles.

```
mpcverbosity('on');
bdclose(mdl)
```

```
f = findobj('Name','Automated Parking Valet');
close(f)
```

See Also

Functions
`nlmpc` | `nlmpcmove`

Blocks
Nonlinear MPC Controller

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

Parallel Parking Using Nonlinear Model Predictive Control

This example shows how to design a parallel parking controller using nonlinear model predictive control (NLMPC).

Parking Environment

In this example, the parking environment contains an ego vehicle and six static obstacles. The obstacles include four parked vehicles, the road curbside, and a yellow line on the road. The goal of the ego vehicle is to park at a target pose without colliding with any of the obstacles. The reference point for the ego vehicle pose is located at the center of rear axle.

The ego vehicle has two axles and four wheels. Define the ego vehicle parameters.

```
vdims = vehicleDimensions;
egoWheelbase = vdims.Wheelbase;
distToCenter = 0.5*egoWheelbase;
```

The ego vehicle starts at the following initial pose.

- X position of `7` m
- Y position of `3.1` m
- Yaw angle `0` rad

```
egoInitialPose = [7,3.1,0];
```

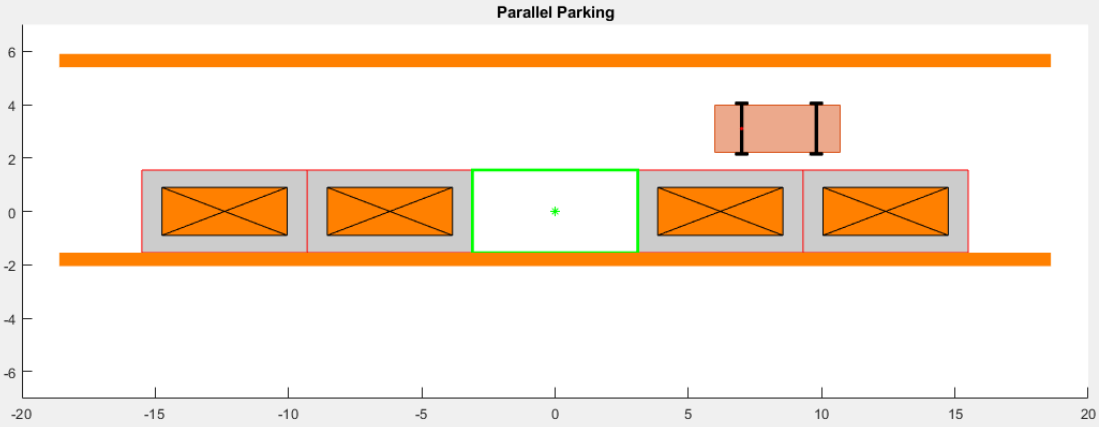
To park the center of the ego vehicle at the target location (X = `0`, Y = `0`) use the following target pose, which specifies the location of the rear-axle reference point.

- X position equal to half the wheelbase length in the negative X direction
- Y position of `0` m
- Yaw angle `0` rad

```
egoTargetPose = [-distToCenter,0,0];
```

Visualize the parking environment. Specify a visualizer sample time of `0.1` s.

```
Tv = 0.1;
helperSLVisualizeParking(egoInitialPose,0);
```



In the visualization, the four parked vehicles are the orange boxes in the middle. The bottom orange boundary is the road curbside and the top orange boundary is the yellow line on the road.

Ego Vehicle Model

For parking problems, the vehicle travels at low speeds. This example uses a kinematic bicycle model with front steering angle for the vehicle parking problem. The motion of the ego vehicle can be described using the following equations.

$$\dot{x} = v \cdot \cos(\psi)$$
$$\dot{y} = v \cdot \sin(\psi)$$
$$\dot{\psi} = \frac{v}{b} \cdot \tan(\delta)$$

Here, (x, y) denotes the position of the vehicle and ψ denotes the yaw angle of the vehicle. The parameter b represents the wheelbase of the vehicle. (x, y, ψ) are the state variables for the vehicle state functions. The speed v and steering angle δ are the control variables for the vehicle state functions. The vehicle state functions are implemented in `parkingVehicleStateFcn`.

Design Nonlinear Model Predictive Controller

The nonlinear model predictive controller for parking is designed based on the following analysis.

- The output of the vehicle state function is the same as the state of the vehicle (x, y, ψ). Therefore, the NLMPC controller object is created with three states, three outputs, and two manipulated variables.
- The speed of the ego vehicle is constrained to be between -2 and 2 m/s, and the steering angle of the ego vehicle is constrained to be between -45 and 45 degrees.
- The NLMPC controller uses a custom cost function, which is defined in a manner similar to a quadratic tracking cost plus a terminal cost. In the following custom cost function, $s(t)$ denotes the states of ego vehicle at time t, d represents the duration of simulation, and s_{ref} is the target pose of the ego vehicle. The weight matrices Q_p, R_p, Q_t, and R_t are constant.

$$J = \int_0^d (s(t) - s_{ref})^T Q_p (s(t) - s_{ref}) + u(t)^T R_p u(t) dt + (s(d) - s_{ref})^T Q_t (s(d) - s_{ref}) + u(d)^T R_t u(d)$$

- To avoid collisions with obstacles, the NLMPC controller must satisfy the following inequality constraints where the minimum distance to all obstacles $dist_{min}$ must be greater than a safe distance $dist_{safe}$. In this example, the ego vehicle and obstacles are modeled as `collisionBox` (Robotics System Toolbox) objects and the distance from ego vehicle to obstacles is computed using `checkCollision` (Robotics System Toolbox).

  $$dist_{min} \geq dist_{safe}$$

- To improve the simulation efficiency, the Jacobians of the state function, cost function, and inequality constraints are all provided to the NLMPC controller. The Jacobians of the inequality constraints are approximated based on [1].
- The initial guesses for the state solutions are defined by straight lines between the initial and target poses of the ego vehicle.

Specify the sample time (`Ts`), prediction horizon (`p`), and control horizon (`m`) for the nonlinear MPC controller.

```
Ts = 0.1;
p = 70;
c = 70;
```

Specify constant weight matrices for the controller. Define both the tracking weight matrices (`Qp` and `Rp`) and the terminal weight matrices (`Qt` and `Rt`).

```
Qp = diag([0.1 0.1 0]);
Rp = 0.01*eye(2);
Qt = diag([1 5 100]);
Rt = 0.1*eye(2);
```

Specify the safety distance of `0.1` m, which the controller uses when defining its constraints.

```
safetyDistance = 0.1;
```

Specify the maximum number of iterations for the NLMPC solver.

```
maxIter = 40;
```

Create the nonlinear MPC controller. For clarity, first disable the MPC command-window messages.

```
mpcverbosity('off');
```

Create the `nlmpc` controller object with three states, three outputs, and two inputs.

```
nx = 3;
ny = 3;
nu = 2;
nlobj = nlmpc(nx,ny,nu);
```

Specify the sample time (`Ts`), prediction horizon (`PredictionHorizon`), and control horizon (`ControlHorizon`) for the controller.

```
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Define constraints for the manipulated variables. Here, `MV(1)` is the ego vehicle speed in m/s, and `MV(2)` is the steering angle in radians.

```
nlobj.MV(1).Min = -2;
nlobj.MV(1).Max = 2;
nlobj.MV(2).Min = -pi/4;
nlobj.MV(2).Max = pi/4;
```

Specify the controller state function and the state-function Jacobian.

```
nlobj.Model.StateFcn = "parkingVehicleStateFcn";
nlobj.Jacobian.StateFcn = "parkingVehicleStateJacobianFcn";
```

Specify the controller cost function and the cost-function Jacobian.

```
nlobj.Optimization.CustomCostFcn = "parkingCostFcn";
nlobj.Optimization.ReplaceStandardCost = true;
nlobj.Jacobian.CustomCostFcn = "parkingCostJacobian";
```

Define custom inequality constraints for the controller and the constraint Jacobian. The custom constraint function computes the distance form the ego vehicle to all the obstacles in the environment and compares these distances to the safe distance.

```
nlobj.Optimization.CustomIneqConFcn = "parkingIneqConFcn";
nlobj.Jacobian.CustomIneqConFcn = "parkingIneqConFcnJacobian";
```

Configure the optimization solver of the controller.

```
nlobj.Optimization.SolverOptions.FunctionTolerance = 0.01;
nlobj.Optimization.SolverOptions.StepTolerance = 0.01;
nlobj.Optimization.SolverOptions.ConstraintTolerance = 0.01;
nlobj.Optimization.SolverOptions.OptimalityTolerance = 0.01;
nlobj.Optimization.SolverOptions.MaxIter = maxIter;
```

Define an initial guess for the optimal state solution. This initial guess is the straight line from the starting pose to the target pose. Also, specify the values for the ego vehicle parameters in the `nlmpcmoveopt` object.

```
opt = nlmpcmoveopt;
opt.X0 = [linspace(egoInitialPose(1),egoTargetPose(1),p)', ...
          linspace(egoInitialPose(2),egoInitialPose(2),p)'...
          zeros(p,1)];
opt.MV0 = zeros(p,nu);
```

Computing the cost function and inequality constraints, along with their Jacobians, requires passing parameters to the custom functions. Define the parameter vector and specify the number of parameters. Also, specify the parameter values in the `nlmpcmoveopt` object.

```
paras = {egoTargetPose,Qp,Rp,Qt,Rt,distToCenter,safetyDistance}';
nlobj.Model.NumberOfParameters = numel(paras);
opt.Parameters = paras;
```

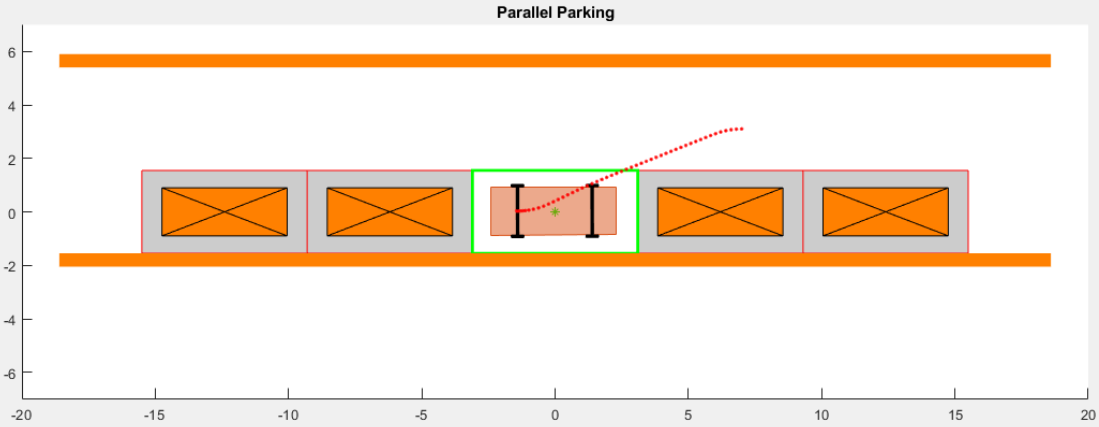
Simulate Controller in MATLAB

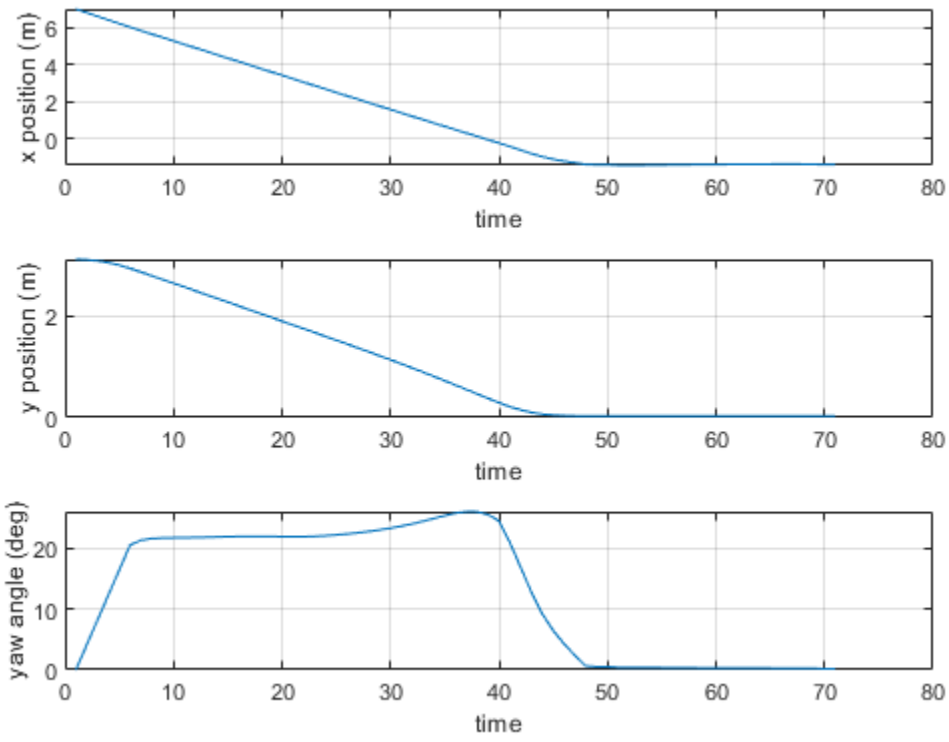
To simulate an NLMPC controller in MATLAB®, you can use one of the following options:

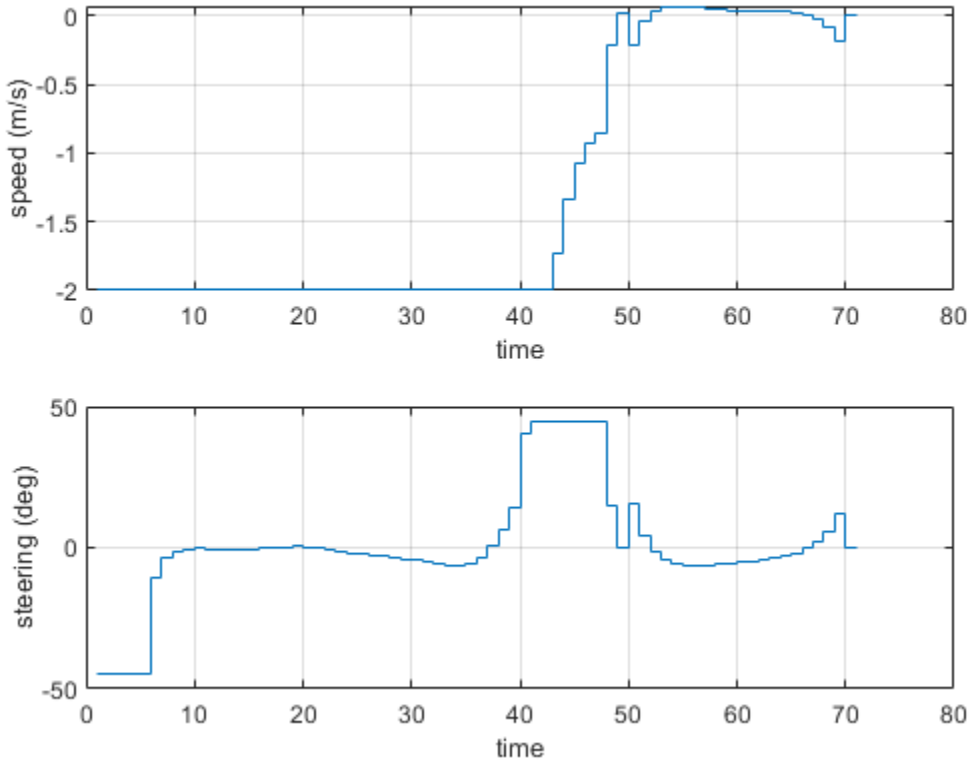
- Simulate the controller using the `nlmpcmove` function.
- Build a MEX file for the controller using the `buildMEX` function. Evaluating this MEX file improves the simulation efficiency compared to `nlmpcmove`.

Simulate the NLMPC controller for parking using the `runParkingAndPlot` script. For this simulation, do not build a MEX file (set `useMEX` to `0`).

```
useMex = 0;
runParkingAndPlot
```







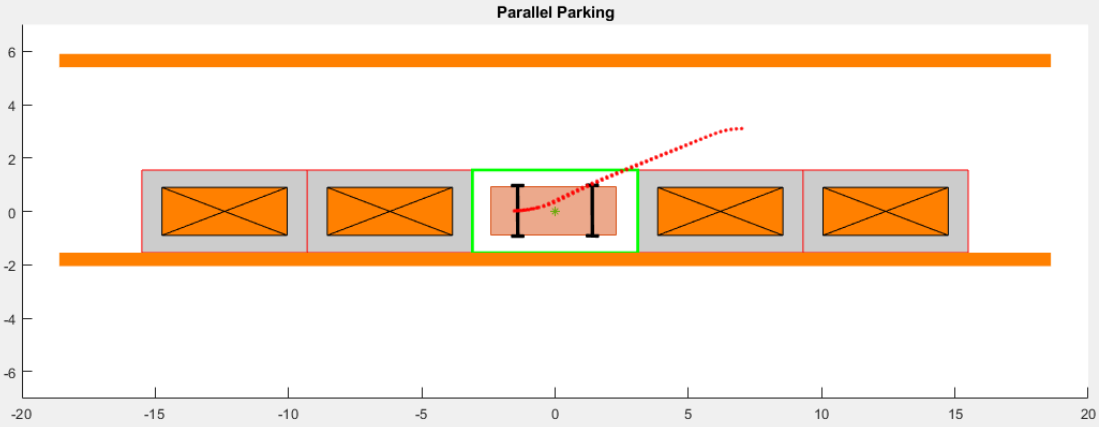

```
Summary of results:
1) Valid results. No collisions.
2) Minimum distance to obstacles = 0.1782 (Valid when greater than safety distance 0.1000)
3) Optimization exit flag = 1 (Successful when positive)
4) Elapsed time (s) for nlmpcmove = 29.7581
5) Final states error in x (m), y (m) and theta (deg):  -0.0087, 0.0294, 0.1698
6) Final control inputs speed (m/s) and steering angle (deg): -0.0006, -0.0180
```

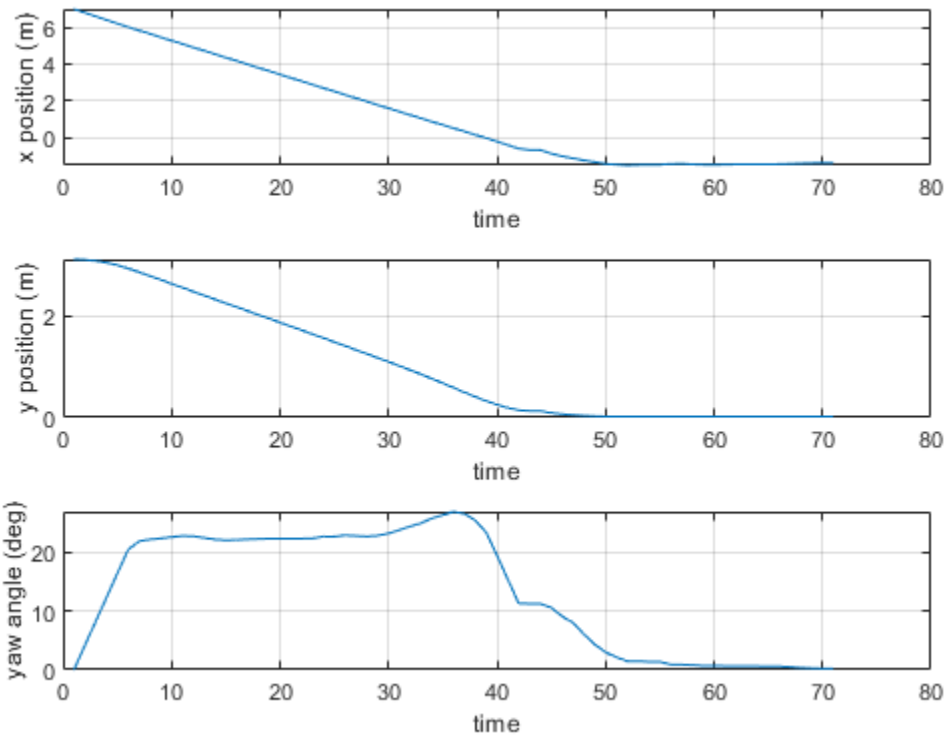
The ego vehicle parks in the target pose successfully. The final control input values are close to zero. In the animation and the ego vehicle does not collide with any obstacles at any time.

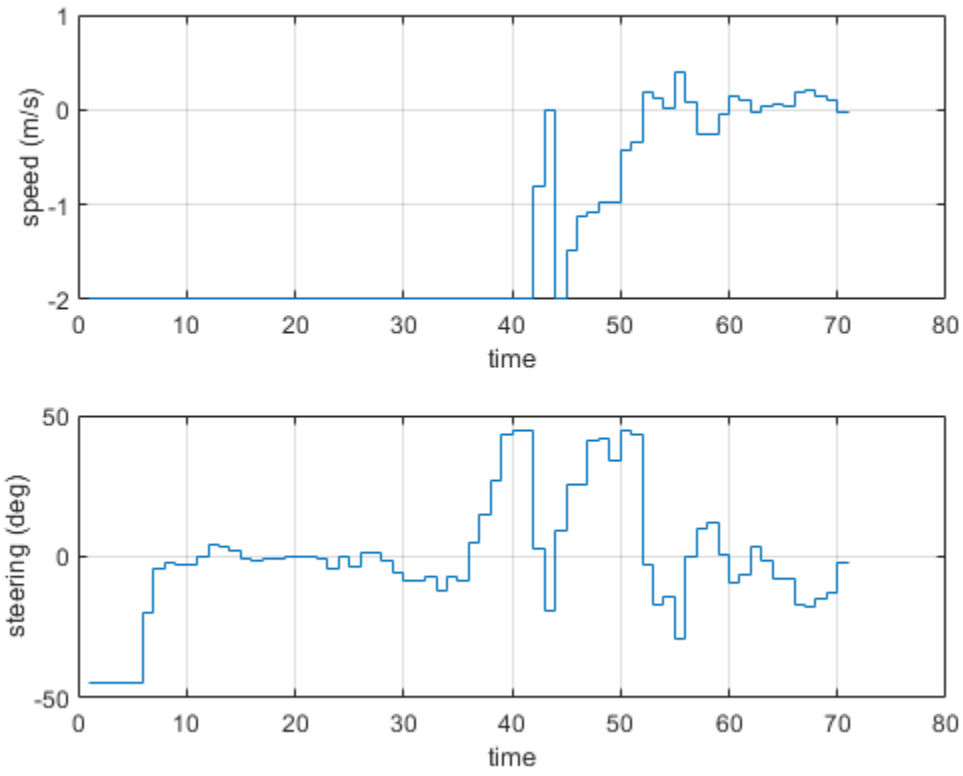
Build a MEX file for your controller and rerun the simulation.

```
useMex = 1;
runParkingAndPlot
```

```
Generating MEX function "parkingMex" from nonlinear MPC to speed up simulation.
Code generation successful.

MEX function "parkingMex" successfully generated.
```


Parallel Parking
6
4
2
0
-2
-4
-6
-20
-15
-10
-5
0
5
10
15
20


x position (m)
time
y position (m)
time
yaw angle (deg)
time
0
10
20
30
40
50
60
70
80




```
Summary of results:
1) Valid results. No collisions.
2) Minimum distance to obstacles = 0.1274 (Valid when greater than safety distance 0.1000)
3) Optimization exit flag = 0 (Successful when positive)
4) Elapsed time (s) for nlmpcmove = 6.2446
5) Final states error in x (m), y (m) and theta (deg):  -0.0042, 0.0195, 0.2166
6) Final control inputs speed (m/s) and steering angle (deg): -0.0284, -1.8098
```

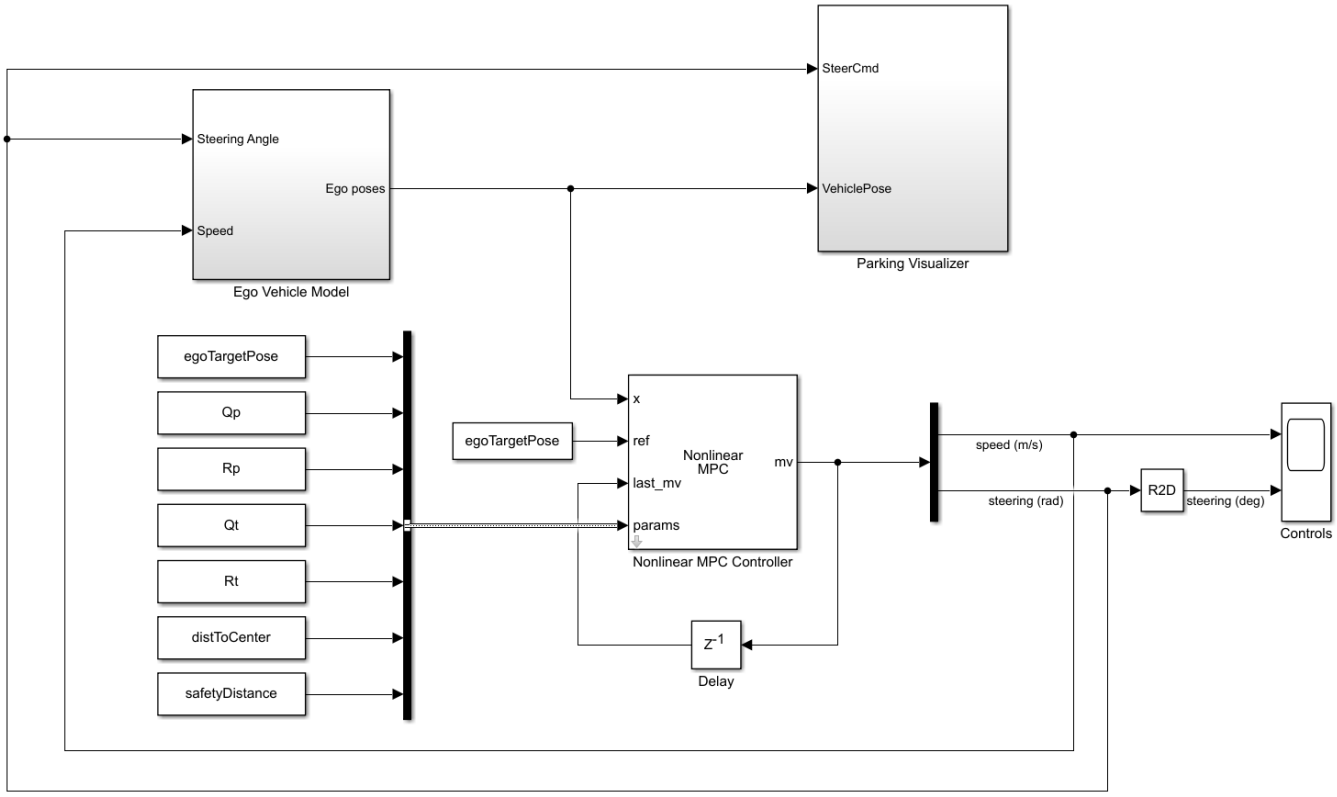
The simulation using the MEX file produces similar results and is significantly faster than the simulation using `nlmpcmove`.

Simulate Controller in Simulink

To simulate the NLMPC controller in Simulink ®, use the Nonlinear MPC Controller block. For this example, to simulate the ego vehicle, use the Vehicle Body 3DOF Lateral block, which is a Bicycle Model (Automated Driving Toolbox) block.

Specify the simulation duration and open the Simulink model.

```
Duration = p*Ts;
mdl = 'mpcVDAutoParking';
open_system(mdl)
```



To pass the ego vehicle parameters to the controller, you must create a parameter bus object.

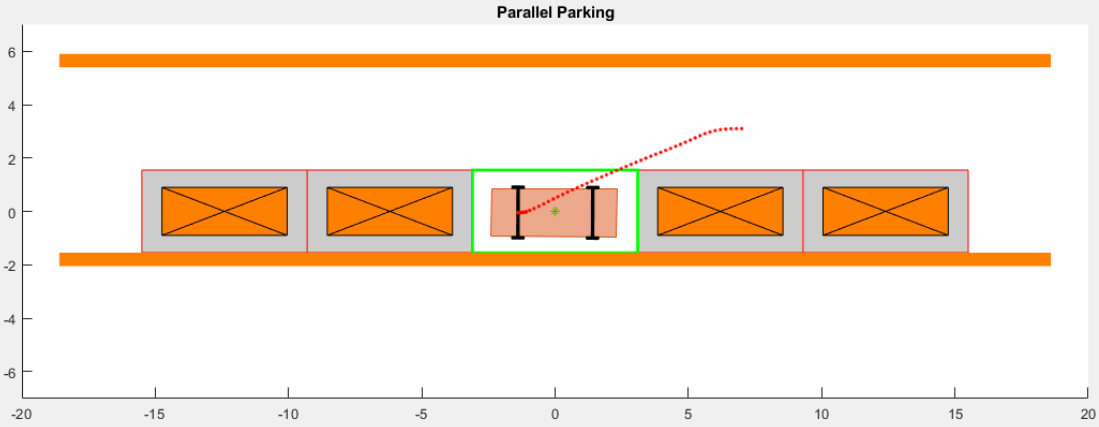
```
createParameterBus(nlobj,[mdl '/Nonlinear MPC Controller'],'parasBusObject',paras);
```

Close the animation plot before simulating the model.

```
f = findobj('Name','Automated Parallel Parking');
close(f)
```

Simulate the model.

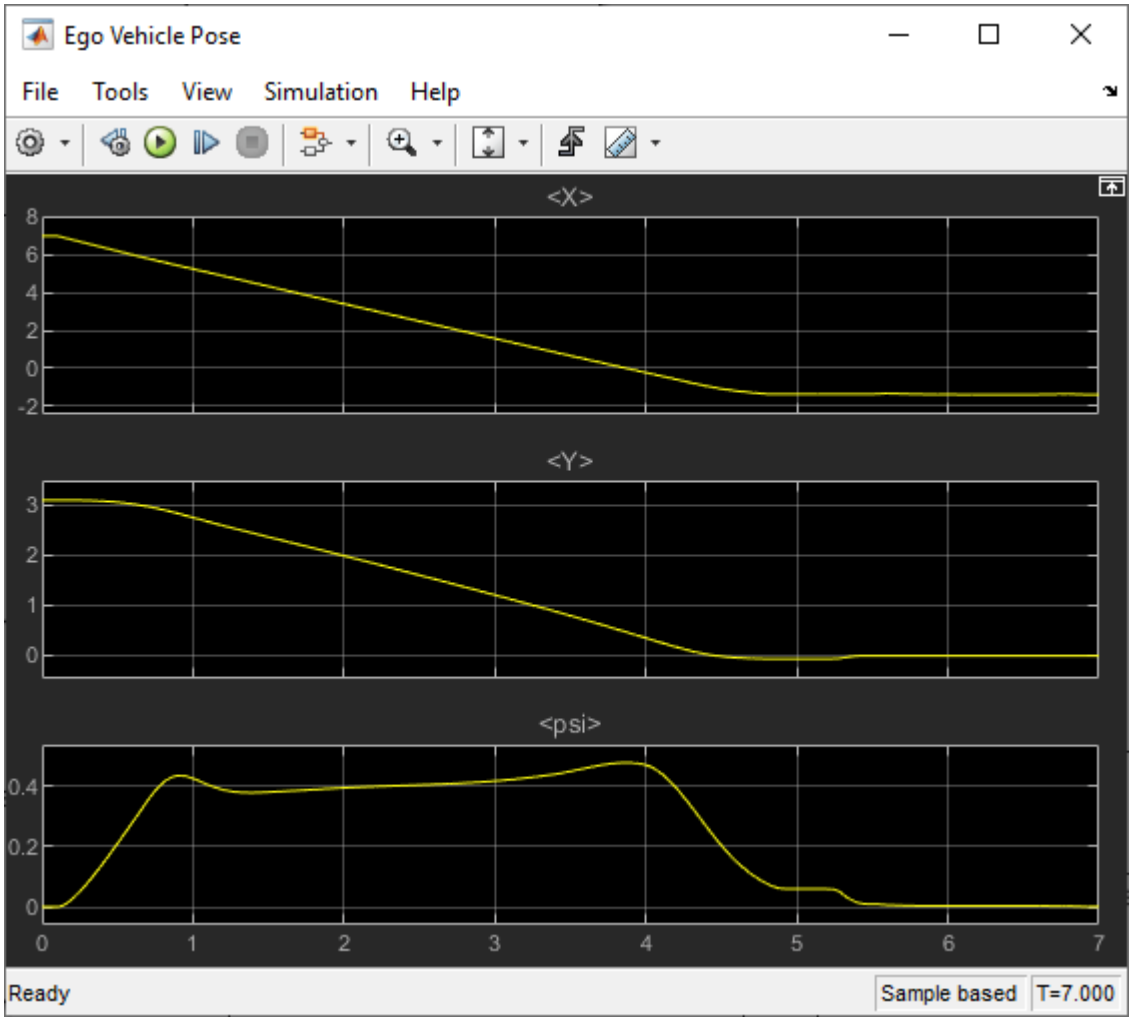
```
sim(mdl)
```

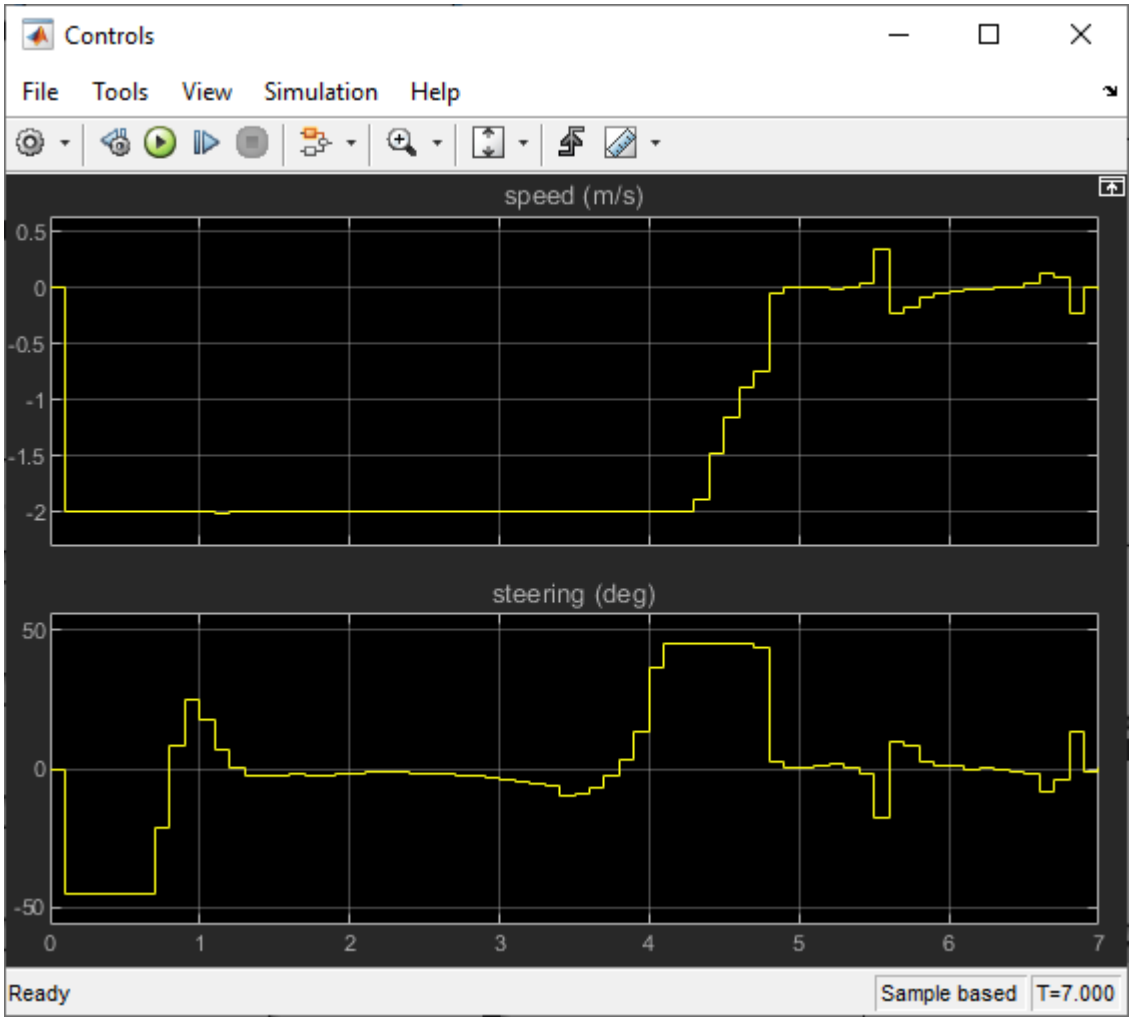



```
ans =
  Simulink.SimulationOutput:

                  tout: [2164x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```

Examine the Ego Vehicle Pose and Controls scopes.

```
open_system([mdl '/Ego Vehicle Model/Ego Vehicle Pose'])
open_system([mdl '/Controls'])
```


Ego Vehicle Pose
File
Tools
View
Simulation
Help
<X>
8
6
4
2
0
-2
<Y>
3
2
1
0
<psi>
0.4
0.2
0
0
1
2
3
4
5
6
7
Ready
Sample based
T=7.000



The simulation results are similar to the MATLAB simulation. The ego vehicle has parked at the target pose successfully without collisions with any obstacles.

Conclusion

This example shows how to design a nonlinear MPC controller for parallel parking. The controller navigates the ego vehicle to the target parking spot without colliding with any obstacles.

```
% Enable message display
mpcverbosity('on');
% Close Simulink model
bdclose(mdl)
% Close animation plots
f = findobj('Name','Automated Parallel Parking');
close(f)
```

References

[1] Schulman, John, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg, and Pieter Abbeel. ‘Motion Planning with Sequential Convex Optimization and

Convex Collision Checking’. *The International Journal of Robotics Research* 33, no. 9 (August 2014): 1251–70. https://doi.org/10.1177/0278364914528132.

See Also

Functions
`nlmpc` | `nlmpcmove`

Blocks
Nonlinear MPC Controller

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

Parallel Parking Using RRT Planner and MPC Tracking Controller

This example shows how to parallel park an ego car by generating a path using the RRT star planner and tracking the trajectory using nonlinear model predictive control (NLMPC).

Parking Environment

In this example, the parking environment contains an ego vehicle and six static obstacles. The obstacles include four parked vehicles, the road curbside, and a yellow line on the road. The goal of the ego vehicle is to park at a target pose without colliding with any of the obstacles. The reference point for the ego vehicle pose is located at the center of rear axle.

The ego vehicle has two axles and four wheels. Define the ego vehicle parameters.

```
vdims = vehicleDimensions;
egoWheelbase = vdims.Wheelbase;
distToCenter = 0.5*egoWheelbase;
```

The ego vehicle starts at the following initial pose.

- X position of `7` m
- Y position of `3.1` m
- Yaw angle `0` rad

```
egoInitialPose = [7,3.1,0];
```

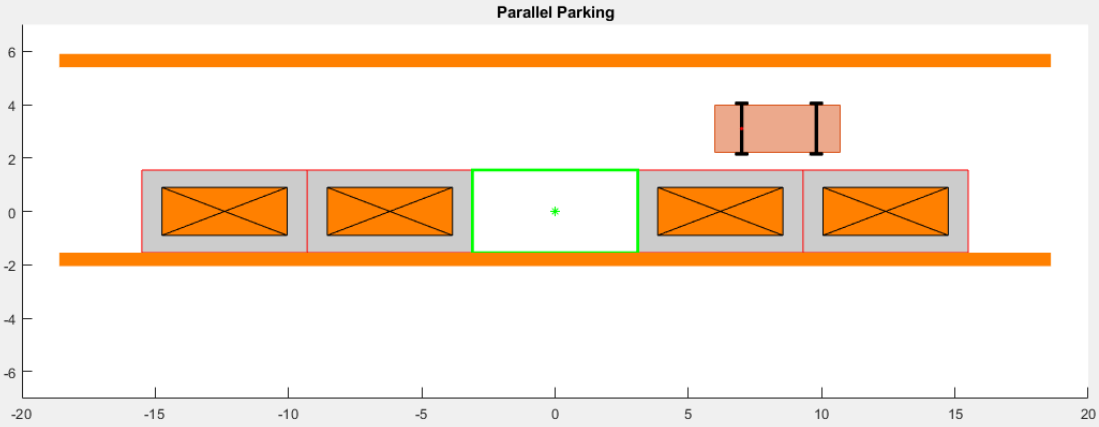
To park the center of the ego vehicle at the target location (X = `0`, Y = `0`) use the following target pose, which specifies the location of the rear-axle reference point.

- X position equal to half the wheelbase length
- Y position of `0` m
- Yaw angle `0` rad

```
egoTargetPose = [-distToCenter,0,0];
```

Visualize the parking environment. Specify a visualizer sample time of `0.1` s.

```
Tv = 0.1;
helperSLVisualizeParking(egoInitialPose,0);
```



In the visualization, the four parked vehicles are the orange boxes in the middle. The bottom orange boundary is the road curbside and the top orange boundary is the yellow line on the road.

Ego Vehicle Model

For parking problems, the vehicle travels at low speeds. This example uses a kinematic bicycle model with front steering angle for the vehicle parking problem. The motion of the ego vehicle can be described using the following equations.

$$\dot{x} = v \cdot \cos(\psi)$$
$$\dot{y} = v \cdot \sin(\psi)$$
$$\dot{\psi} = \frac{v}{b} \cdot \tan(\delta)$$

Here, (x, y) denotes the position of the vehicle and ψ denotes the yaw angle of the vehicle. The parameter b represents the wheelbase of the vehicle. (x, y, ψ) are the state variables for the vehicle state functions. The speed v and steering angle δ are the control variables for the vehicle state functions. The vehicle state functions are implemented in `parkingVehicleStateFcnRRT`.

Path Planning from RRT Star

Configure the state space for the planner. In this example, the state of the ego vehicle is a three-element vector, [x y theta], with the xy coordinates in meters and angle of rotation in radians.

```
xlim = [-10 10];
ylim = [-2 6];
yawlim = [-3.1416 3.1416];
bounds = [xlim;ylim;yawlim];
stateSpace = stateSpaceReedsShepp(bounds);
stateSpace.MinTurningRadius = 7;
```

Create a custom state validator. The planner requires a customized state validator to enable collision checking between the ego vehicle and obstacles.

```
stateValidator = parkingStateValidator(stateSpace);
```

Configure the path planner. Use `plannerRRTStar` as the planner and specify the state space and state validator. Specify additional parameters for the planner.

```
planner = plannerRRTStar(stateSpace,stateValidator);
planner.MaxConnectionDistance = 4;
planner.ContinueAfterGoalReached = true;
planner.MaxIterations = 2000;
```

Plan a path from the initial pose to the target pose using the configured path planner. Set the random number seed for repeatability.

```
rng(9, 'twister');
[pathObj,solnInfo] = plan(planner,egoInitialPose,egoTargetPose);
```

Plot the tree expansion on the parking environment.

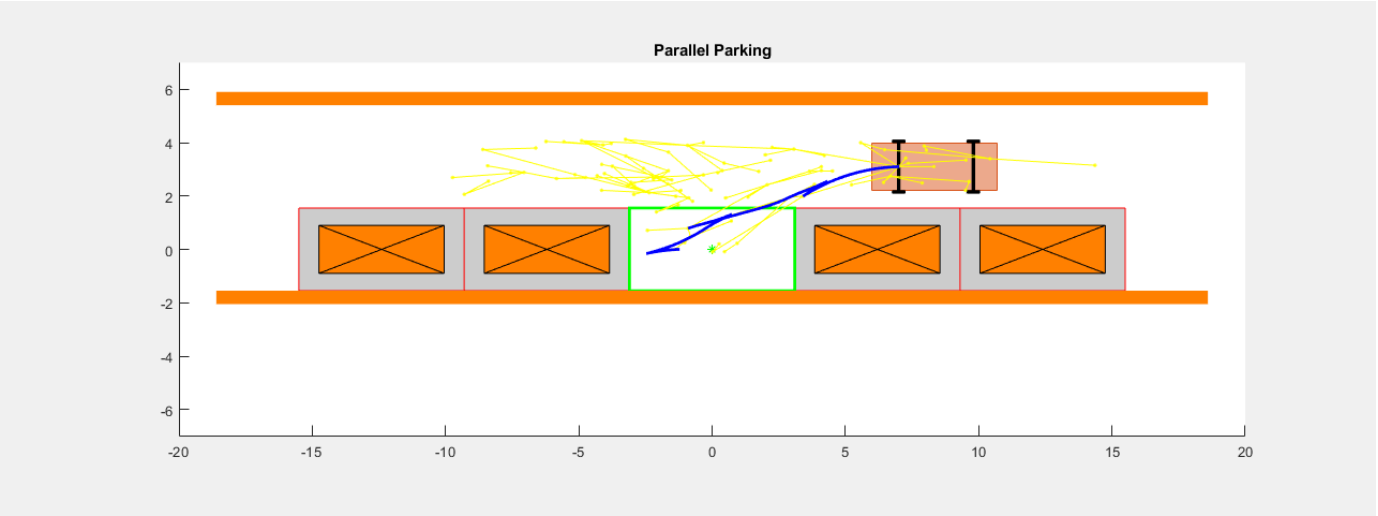
```
f = findobj('Name','Automated Parallel Parking');
ax = gca(f);
hold(ax, 'on');
plot(ax,solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'y.-'); % tree expansion
```

Generate a trajectory from `pathObj` by interpolating with an appropriate number of points.

```
p = 100;
pathObj.interpolate(p+1);
xRef = pathObj.States;
```

Draw the path on the environment.

```
plot(ax,xRef(:,1), xRef(:,2),'b-','LineWidth',2)
```



Design Nonlinear MPC Tracking Controller

Create the nonlinear MPC controller. For clarity, first disable the MPC command-window messages.

```
mpcverbosity('off');
```

Create the `nlmpc` controller object with three states, three outputs, and two inputs.

```
nlobjTracking = nlmpc(3,3,2);
```

Specify the sample time (`Ts`), prediction horizon (`PredictionHorizon`), and control horizon (`ControlHorizon`) for the controller.

```
Ts = 0.1;
pTracking = 10;
nlobjTracking.Ts = Ts;
nlobjTracking.PredictionHorizon = pTracking;
nlobjTracking.ControlHorizon = pTracking;
```

Define constraints for the manipulated variables. Here, `MV(1)` is the ego vehicle speed in m/s, and `MV(2)` is the steering angle in radians.

```
nlobjTracking.MV(1).Min = -2;
nlobjTracking.MV(1).Max = 2;
nlobjTracking.MV(2).Min = -pi/6;
nlobjTracking.MV(2).Max = pi/6;
```

Specify tuning weights for the controller.

```
nlobjTracking.Weights.OutputVariables = [1,1,3];
nlobjTracking.Weights.ManipulatedVariablesRate = [0.1,0.2];
```

The motion of ego vehicle is governed by a kinematic bicycle model. Specify the controller state function and the state-function Jacobian.

```
nlobjTracking.Model.StateFcn = "parkingVehicleStateFcnRRT";
nlobjTracking.Jacobian.StateFcn = "parkingVehicleStateJacobianFcnRRT";
```

Specify terminal constraints on control inputs. Both speed and steering angle are expected to be zero at the end.

```
nlobjTracking.Optimization.CustomEqConFcn = "parkingTerminalConFcn";
```

Validate the controller design.

```
validateFcns(nlobjTracking,randn(3,1),randn(2,1));
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Run Closed-loop Simulation in MATLAB

To speed up simulation, first generate a MEX function for the NLMPC controller.

Specify the initial ego vehicle state.

```
x = egoInitialPose';
```

Specify the initial control inputs.

```
u = [0;0];
```

Obtain code generation data for the NLMPC controller.

```
[coredata,onlinedata] = getCodeGenerationData(nlobjTracking,x,u);
```

Build a MEX function for simulating the controller.

```
mexfcn = buildMEX(nlobjTracking,'parkingRRTMex',coredata,onlinedata);
```

```
Generating MEX function "parkingRRTMex" from nonlinear MPC to speed up simulation.
Code generation successful.

MEX function "parkingRRTMex" successfully generated.
```

Initialize data before running simulation.

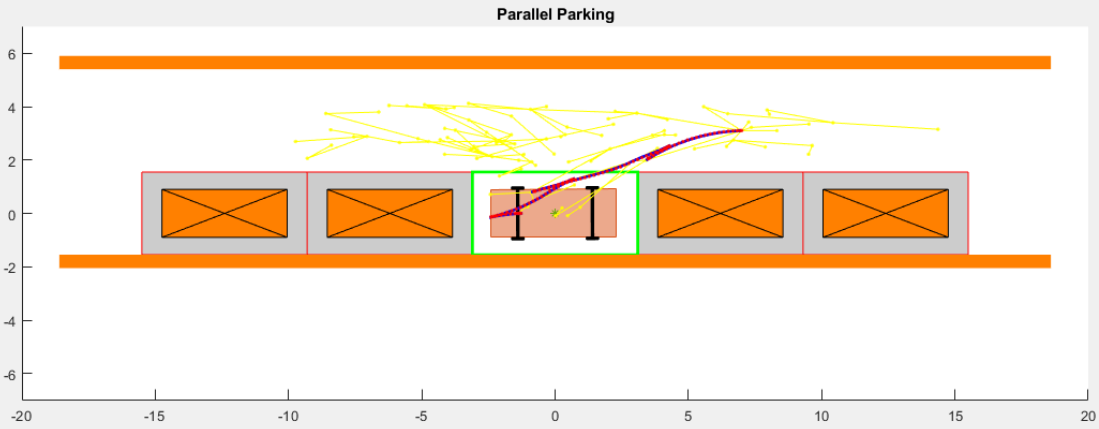
```
xTrackHistory = x;
uTrackHistory = u;
mv = u;
Duration = 14;
Tsteps = Duration/Ts;
Xref = [xRef(2:p+1,:);repmat(xRef(end,:),Tsteps-p,1)];
```

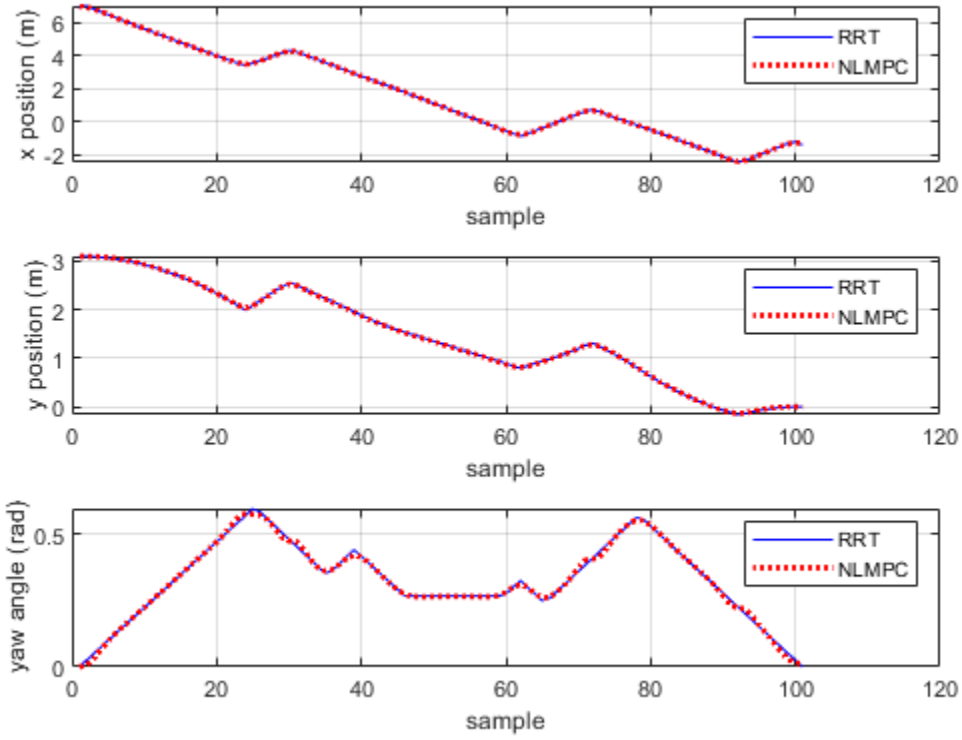
Run the closed-loop simulation in MATLAB using the MEX function.

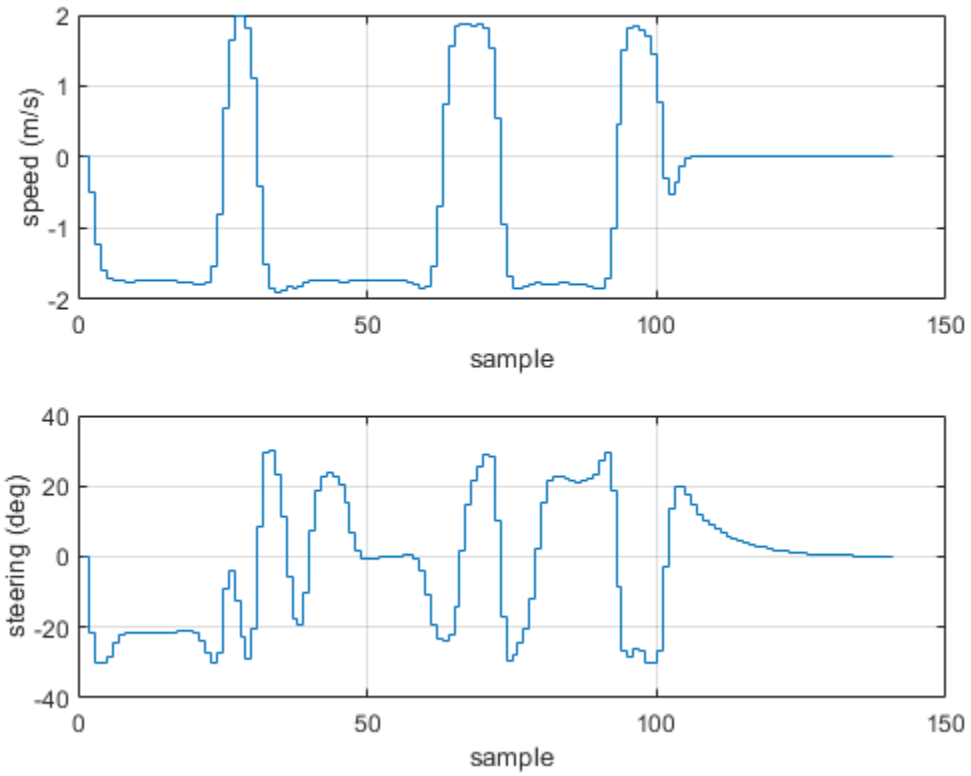
```
for ct = 1:Tsteps
    % States
    xk = x;
    % Compute optimal control moves with MEX function
    onlinedata.ref = Xref(ct:min(ct+pTracking-1,Tsteps),:);
    [mv,onlinedata,info] = mexfcn(xk,mv,onlinedata);
    % Implement first optimal control move and update plant states.
    ODEFUN = @(t,xk) parkingVehicleStateFcnRRT(xk,mv);
    [TOUT,YOUT] = ode45(ODEFUN,[0 Ts], xk);
    x = YOUT(end,:)';
    % Save plant states for display.
    xTrackHistory = [xTrackHistory x]; %#ok<*AGROW>
    uTrackHistory = [uTrackHistory mv];
end
```

Plot and animate the simulation results when using the NLMPC controller. The tracking results match the reference trajectory from the path planner.

```
plotAndAnimateParkingRRT(p,xRef,xTrackHistory,uTrackHistory);
```




x position (m)
6
4
2
0
-2
0
20
40
60
80
100
120
sample
RRT
NLMPC
y position (m)
3
2
1
0
0
20
40
60
80
100
120
sample
RRT
NLMPC
yaw angle (rad)
0.5
0
0
20
40
60
80
100
120
sample
RRT
NLMPC

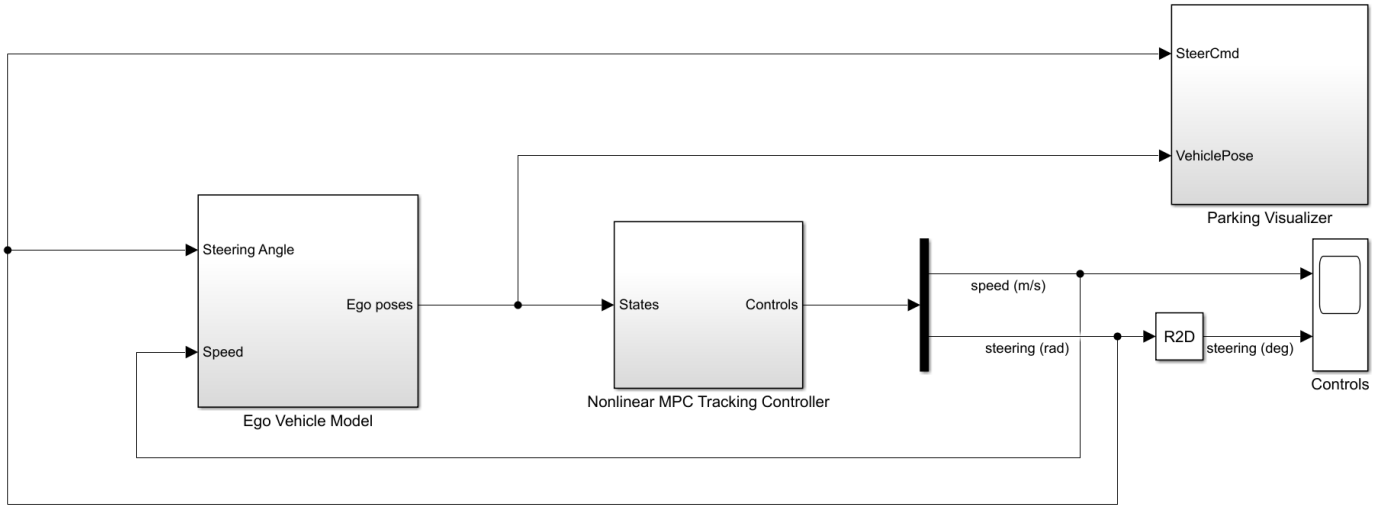



```
Tracking error infinity norm in x (m), y (m) and theta (deg): 0.0538, 0.0538, 1.3432
Final control inputs of speed (m/s) and steering angle (deg): 0.0001, 0.0981
```

Run Closed-loop Simulation in Simulink

To simulate the NLMPC controller in Simulink ®, use the Nonlinear MPC Controller block. For this example, to simulate the ego vehicle, use the Vehicle Body 3DOF Lateral block, which is a Bicycle Model (Automated Driving Toolbox) block.

```
mdl = 'mpcVDAutoParkingRRT';
open_system(mdl)
```

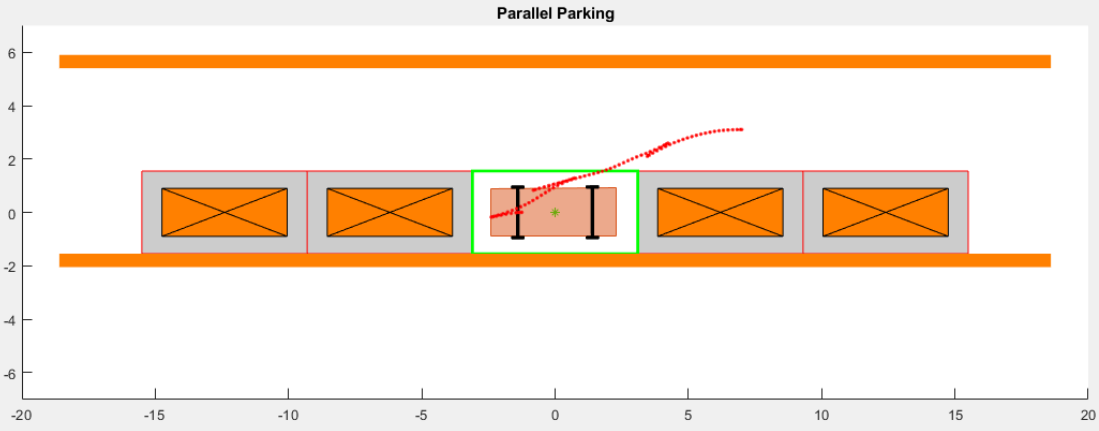


Close the animation plot before simulating the model.

```
f = findobj('Name','Automated Parallel Parking');
close(f)
```

Simulate the model.

```
sim(mdl)
```




```
ans =
  Simulink.SimulationOutput:

                    tout: [7661x1 double]

      SimulationMetadata: [1x1 Simulink.SimulationMetadata]
            ErrorMessage: [0x0 char]
```

Examine the Ego Vehicle Pose and Controls scopes. The simulation results are similar to the MATLAB simulation. The ego vehicle has parked at the target pose successfully without collisions with any obstacles.

Conclusion

This example shows how to how to parallel park an ego car by generating a path using an RRT star planner and tracking the trajectory using a nonlinear MPC controller. The controller navigates the ego vehicle to the target parking spot without colliding with any obstacles.

```
% Enable message display
mpcverbosity('on');
% Close Simulink model
bdclose(mdl)
% Close animation plots
f = findobj('Name','Automated Parallel Parking');
close(f)
```

See Also

Functions

`nlmpc` | `nlmpcmove`

Blocks

Nonlinear MPC Controller

More About

- “Automated Driving Using Model Predictive Control” on page 11-2

Traffic Light Negotiation

This example shows how to design and test decision logic for negotiating a traffic light at an intersection.

Introduction

Decision logic for negotiating traffic lights is a fundamental component of an automated driving application. The decision logic must react to inputs like the state of the traffic light and surrounding vehicles. The decision logic then provides the controller with the desired velocity and path. Since traffic light intersections are dangerous to test, simulating such driving scenarios can provide insight into the interactions of the decision logic and the controller.

This example shows how to design and test the decision logic for negotiating a traffic light. The decision logic in this example reacts to the state of the traffic light, distance to the traffic light, and distance to the closest vehicle ahead. In this example, you will:

1. **Explore the test bench model**: The model contains the traffic light sensors and environment, traffic light decision logic, controls, and vehicle dynamics.
2. **Model the traffic light decision logic**: The traffic light decision logic arbitrates between a lead vehicle and an upcoming traffic light. It also provides a reference path for the ego vehicle to follow at an intersection in the absence of lanes.
3. **Simulate a left turn with traffic light and a lead vehicle**: The model is configured to test the interactions between the traffic light decision logic and controls of the ego vehicle, while approaching an intersection in the presence of a lead vehicle.
4. **Simulate a left turn with traffic light and cross traffic**: The model is configured to test the interactions between the traffic light decision logic and controls of the ego vehicle when there is cross traffic at the intersection.
5. **Explore other scenarios**: These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own decision logic and controls to negotiate traffic lights.

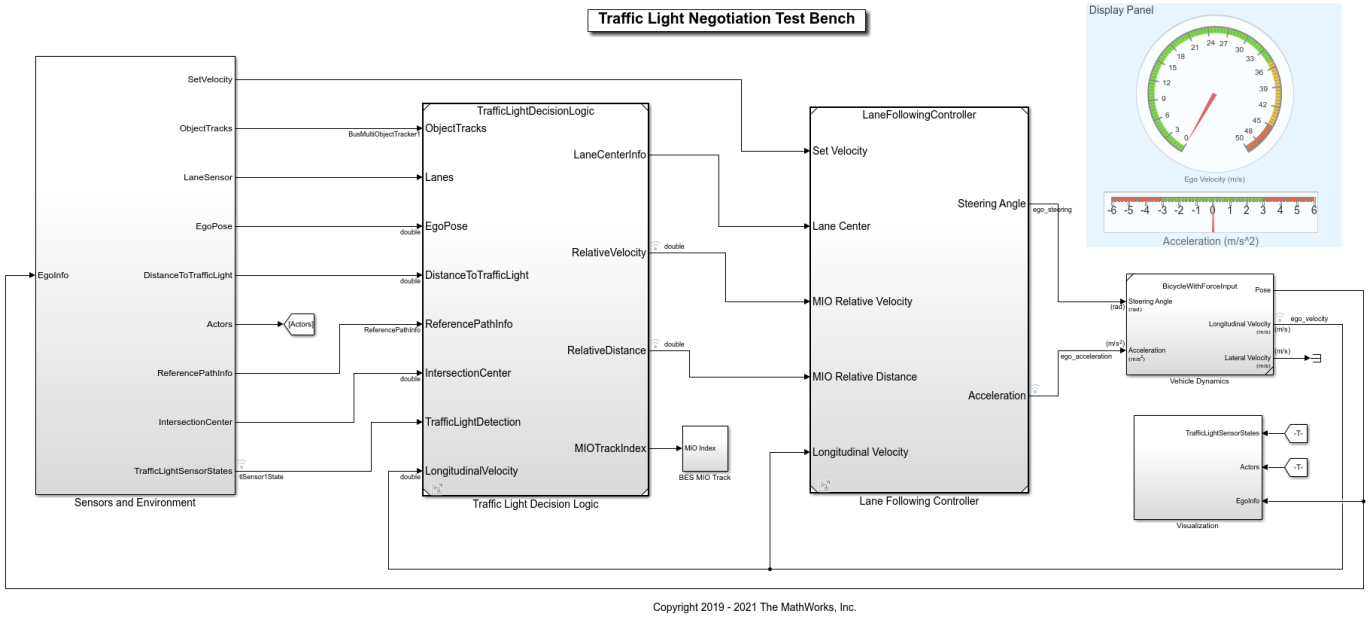
Explore Test Bench Model

To explore the test bench model, open a working copy of the project example files. MATLAB® copies the files to an example folder so that you can edit them.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('TrafficLightNegotiation.zip', 'workDir', pwd);
```

To explore the behavior of the traffic light negotiation system, open the simulation test bench model for the system.

```
open_system("TrafficLightNegotiationTestBench");
```



Opening this model runs the `helperSLTrafficLightNegotiationSetup` script that initializes the road scenario using the `drivingScenario` object in the base workspace. It runs the default test scenario, `scenario_02_TLN_left_turn_with_cross_over_vehicle`, that contains an ego vehicle and two other vehicles. This setup script also configures the controller design parameters, vehicle model parameters, and Simulink® bus signals required for defining the inputs and outputs for the `TrafficLightNegotiationTestBench` model.

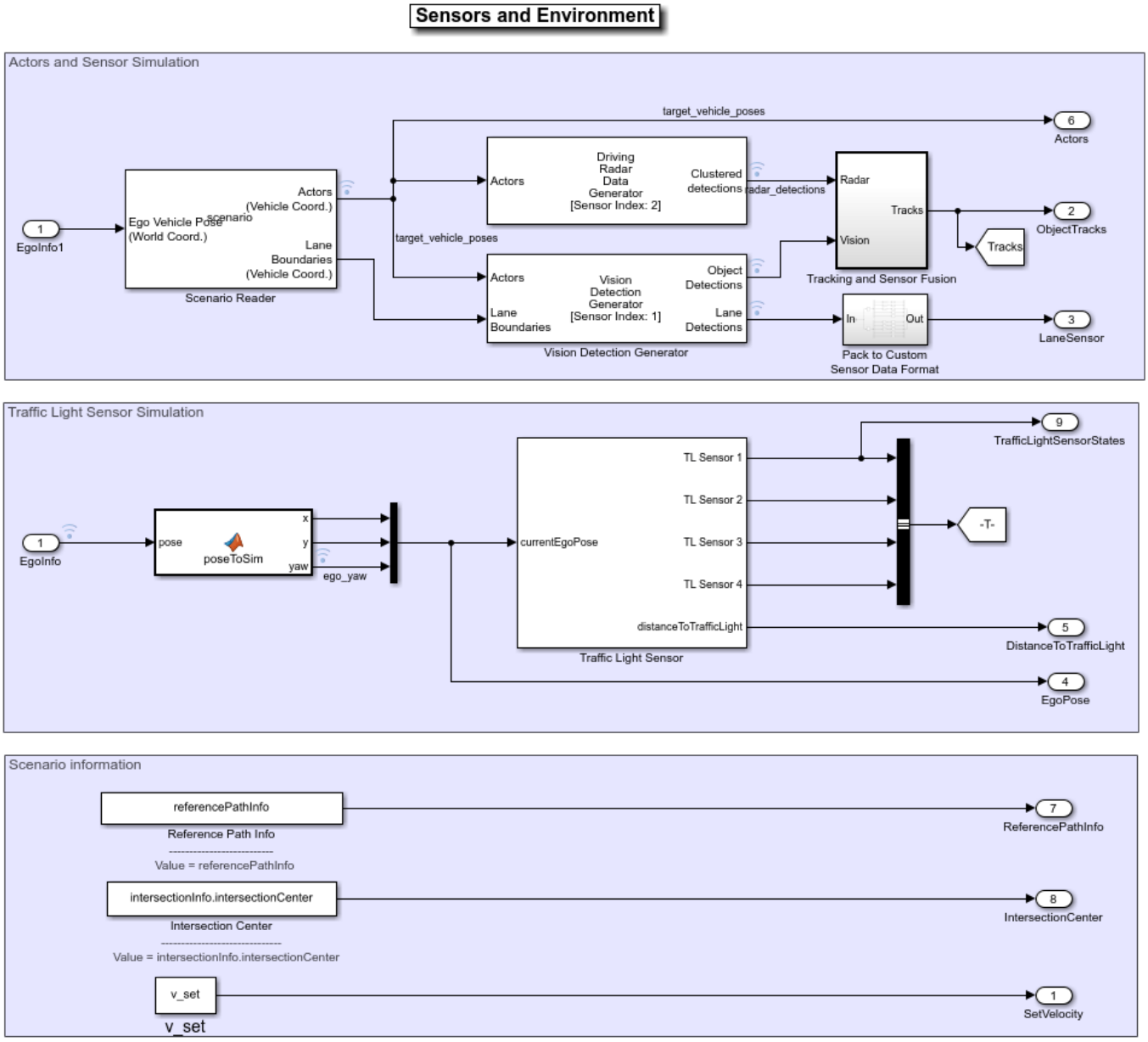
The test bench model contains the following subsystems:

1. **Sensors and Environment**: Models the traffic light sensor, road network, vehicles, and the camera and radar sensors used for simulation.
2. **Traffic Light Decision Logic**: Arbitrates between the traffic light and other lead vehicles or cross-over vehicles at the intersection.
3. **Lane-Following Controller**: Generates longitudinal and lateral controls.
4. **Vehicle Dynamics**: Models the ego vehicle using a Bicycle Model (Automated Driving Toolbox) block and updates its state using commands received from the **Lane Following Controller** subsystem.
5. **Visualization**: Plots the world coordinate view of the road network, vehicles, and the traffic light state during simulation.

The **Lane Following Controller** reference model and the **Vehicle Dynamics** subsystem are reused from the “Highway Lane Following” (Automated Driving Toolbox) example. This example focuses on the **Sensors and Environment** and **Traffic Light Decision Logic** subsystems.

The **Sensors and Environment** subsystem configures the road network, defines target vehicle trajectories, and synthesizes sensors. Open the **Sensors and Environment** subsystem.

```
open_system("TrafficLightNegotiationTestBench/Sensors and Environment");
```

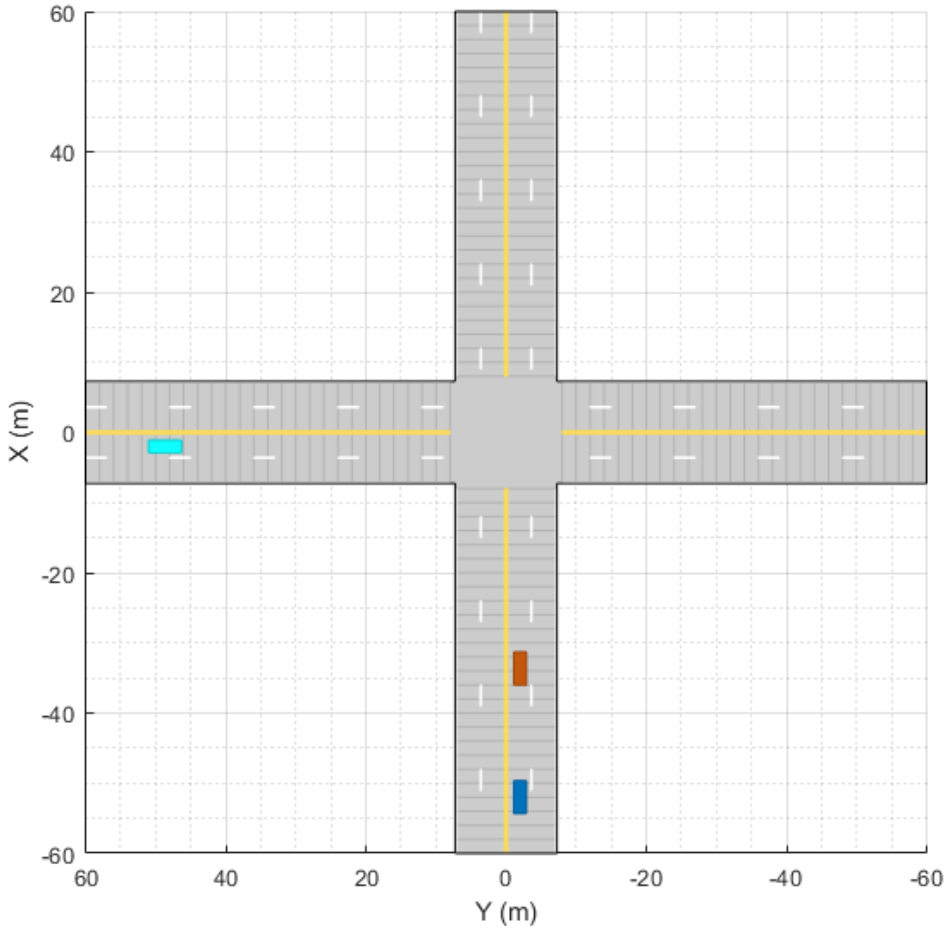


The scenario and sensors on the ego vehicle are specified by the following parts of the subsystem:

- The Scenario Reader (Automated Driving Toolbox) block is configured to take in ego vehicle information to perform a closed-loop simulation. It outputs ground truth information of lanes and actors in ego vehicle coordinates. This block reads the `drivingScenario` object variable, `scenario`, from the base workspace, which contains a road network compatible with the `TrafficLightNegotiationTestBench` model.

Plot the road network provided by the scenario.

```
hFigScenario = figure('Position', [1 1 800 600]);
plot(scenario, 'Parent', axes(hFigScenario));
```



This default scenario has one intersection with an ego vehicle, one lead vehicle, and one cross-traffic vehicle.

Close the figure.

```
close(hFigScenario);
```

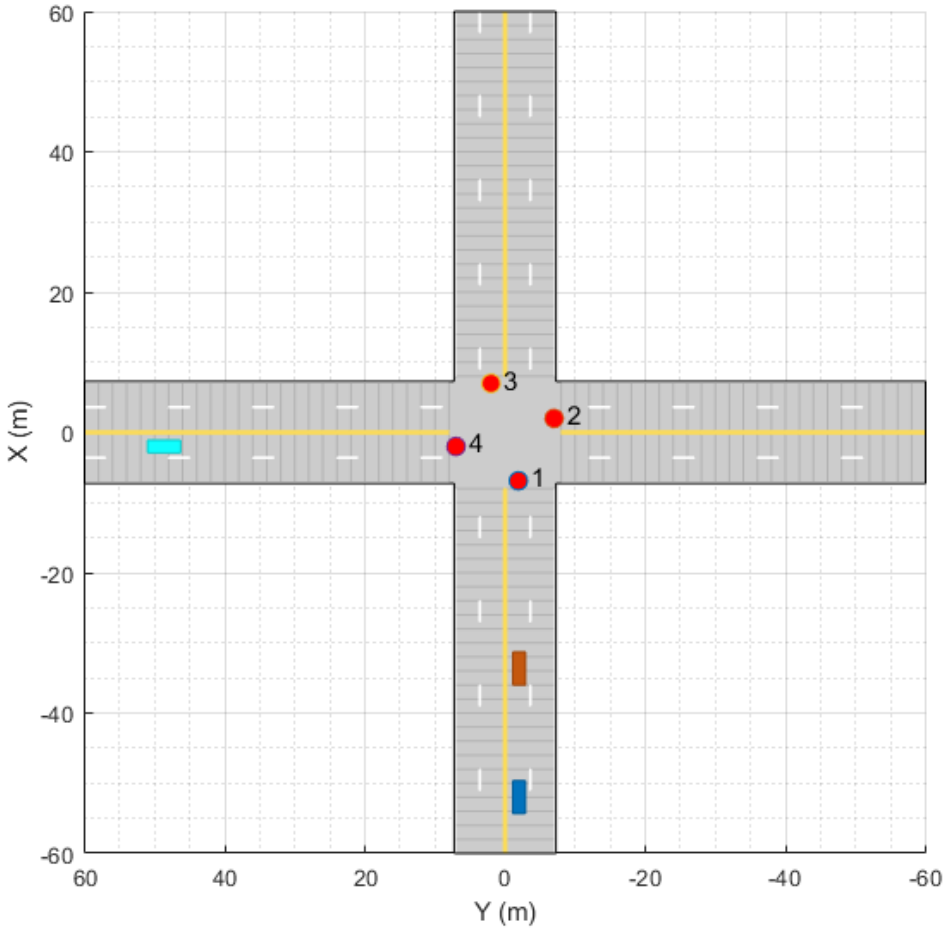
The **Tracking and Sensor Fusion** subsystem fuses vehicle detections from Driving Radar Data Generator (Automated Driving Toolbox) and Vision Detection Generator (Automated Driving Toolbox) blocks by using a Multi-Object Tracker (Automated Driving Toolbox) block to provide object tracks surrounding the ego vehicle.

The Vision Detection Generator block also provides lane detections with respect to the ego vehicle that helps in identifying vehicles present in the ego lane.

The **Traffic Light Sensor** subsystem simulates the traffic lights. It is configured to support four traffic light sensors at the intersection, **TL Sensor 1**, **TL Sensor 2**, **TL Sensor 3**, and **TL Sensor 4**.

Plot the scenario with traffic light sensors.

```
hFigScenario = helperPlotScenarioWithTrafficLights();
```



Observe that this is the same scenario as before, only with traffic light sensors added. These sensors are represented by red circles at the intersection, indicating red traffic lights. The labels for the traffic lights **1**, **2**, **3**, **4** correspond to **TL Sensor 1**, **TL Sensor 2**, **TL Sensor 3**, and **TL Sensor 4**, respectively.

Close the figure.

```
close(hFigScenario);
```

The test scenarios in `TrafficLightNegotiationTestBench` are configured such that the ego vehicle negotiates with **TL Sensor 1**. There are three modes in which you can configure this **Traffic Light Sensor** subsystem:

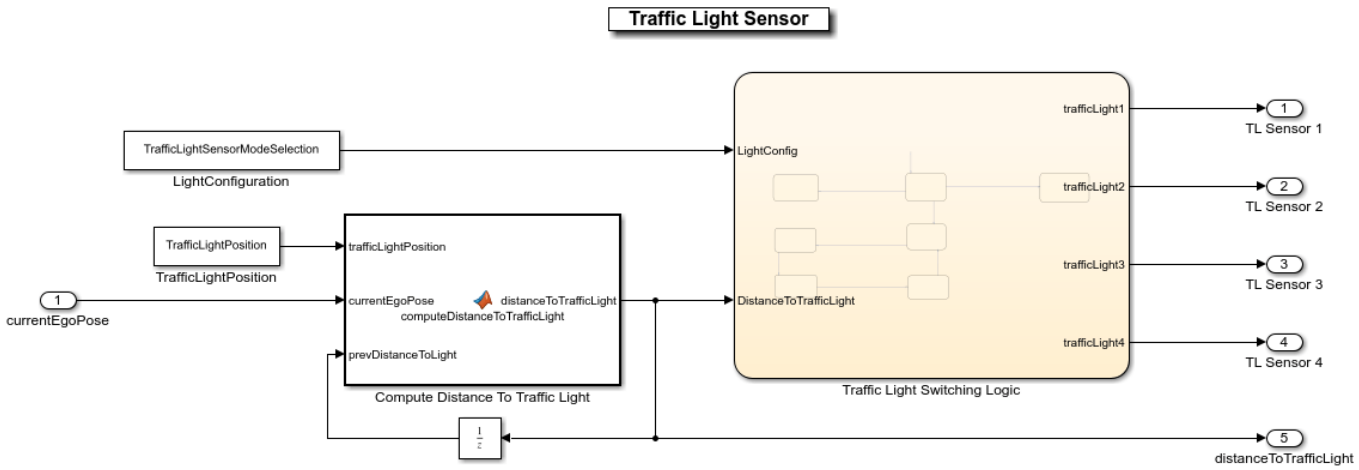
1. **Steady Red**: **TL Sensor 1** and **TL Sensor 3** are always in a red state. The other two traffic lights are always in a green state.
2. **Steady Green**: **TL Sensor 1** and **TL Sensor 3** are always in a green state. The other two traffic lights are always in a red state.

3 **Cycle** [Default]: **TL Sensor 1** and **TL Sensor 3** follow a cyclic pattern: green-yellow-red with predefined timings. Other traffic lights also follow a cyclic pattern: red-green-yellow with predefined timings to complement the **TL Sensor 1** and **TL Sensor 3**.

You can configure this subsystem in one of these modes by using the `Traffic Light Sensor Mode` mask parameter.

Open the **Traffic Light Sensor** subsystem.

```
open_system('TrafficLightNegotiationTestBench/Sensors and Environment/Traffic Light Sensor', 'fo
```



The **Traffic Light Switching Logic** Stateflow® chart implements the traffic light state change logic for the four traffic light sensors. The initial state for all the traffic lights is set to red. Transition to a different mode is based on a trigger condition defined by distance of the ego vehicle to the **TL Sensor 1** traffic light. This distance is defined by the variable `distanceToTrafficLight`. Traffic light transition is triggered if this distance is less than `trafficLightStateTriggerThreshold`. This threshold is currently set to 60 meters and can be changed in the `helperSLTrafficLightNegotiationSetup` script.

The Compute Distance To Traffic Light block calculates `distanceToTrafficLight` using the traffic light position of **TL Sensor 1**, defined by the variable `trafficLightPosition`. This is obtained from the `Traffic Light Position` mask parameter of the **Traffic Light Sensor** subsystem. The value of the mask parameter is set to `intersectionInfo.tlSensor1Position`, a variable set in the base workspace by the `helperSLTrafficLightNegotiationSetup` script. `intersectionInfo` structure is an output from the `helperGetTrafficLightScene` function. This function is used to create the test scenarios that are compatible with the `TrafficLightNegotiationTestBench` model.

The following inputs are needed by the traffic light decision logic and controller to implement their functionalities:

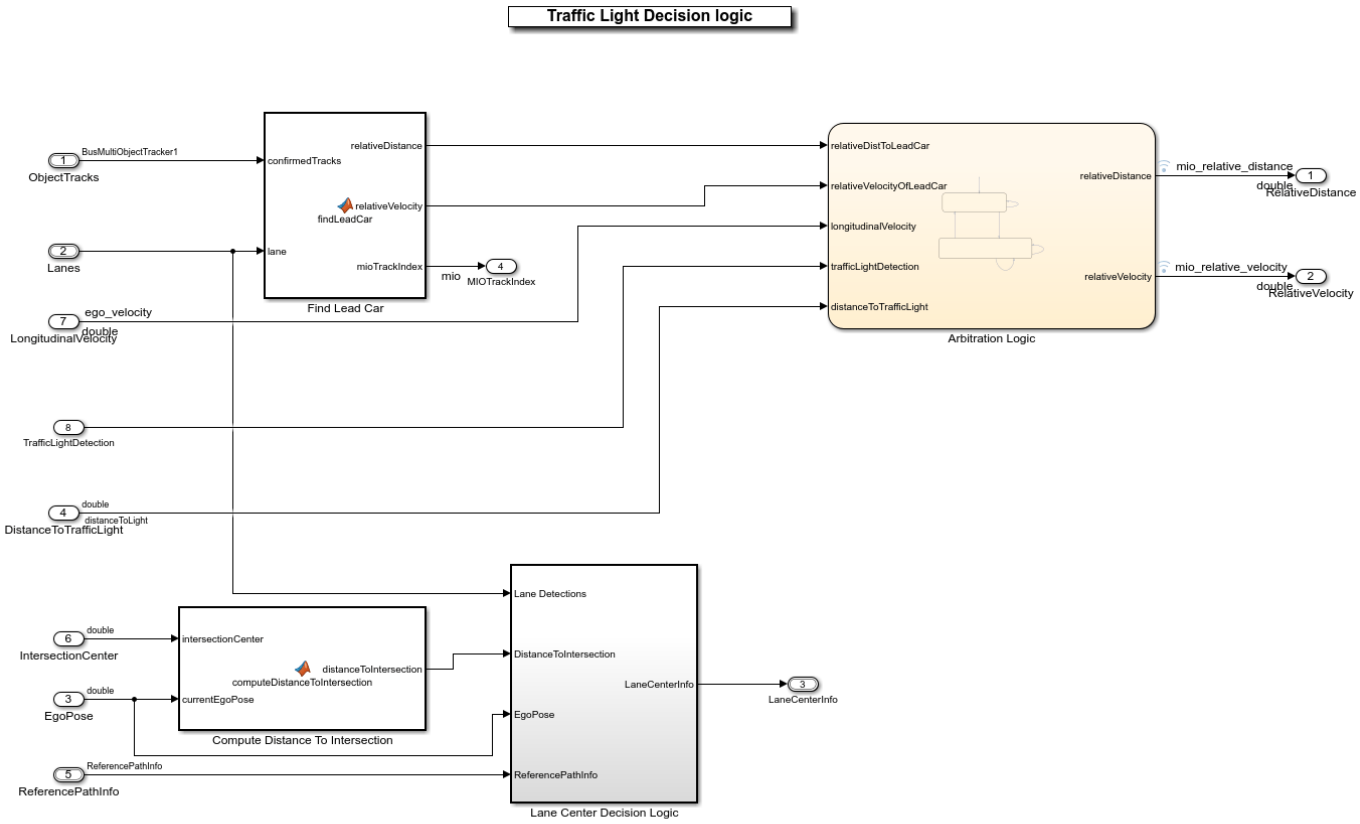
- **ReferencePathInfo** provides a predefined reference trajectory that can be used by the ego vehicle for navigation in absence of lane information. The ego vehicle can go straight, take a left turn, or a right turn at the intersection based on the reference path. This reference path is obtained using `referencePathInfo`, an output from `helperGetTrafficLightScene`. This function takes an input argument to specify the direction of travel at the intersection. The possible values are: `Straight`, `Left`, and `Right`.

- **IntersectionCenter** provides the position of the intersection center of the road network in the scenario. This is obtained using the `intersectionInfo`, an output from `helperGetTrafficLightScene`.
- **Set Velocity** defines the user-set velocity for the controller.

Model Traffic Light Decision Logic

The **Traffic Light Decision Logic** reference model arbitrates between the lead car and the traffic light. It also calculates the lane center information as required by the controller either using the detected lanes or a predefined path. Open the **Traffic Light Decision Logic** reference model.

```
open_system("TrafficLightDecisionLogic");
```

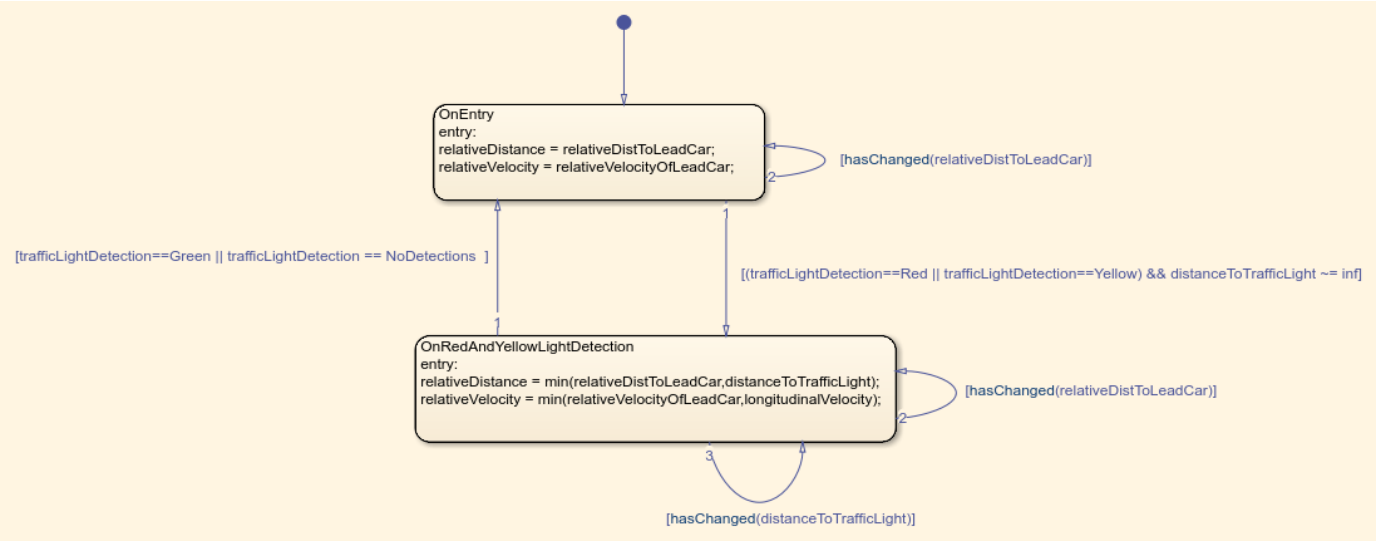


Copyright 2019 - 2021 The MathWorks, Inc.

The **Find Lead Car** subsystem finds the lead car in the current lane from input object tracks. It provides relative distance, **relativeDistToLeadCar**, and relative velocity, **relativeVelocityOfLeadCar**, with respect to the lead vehicle. If there is no lead vehicle, then this block considers the lead vehicle to be present at infinite distance.

The **Arbitration Logic** Stateflow chart uses the lead car information and implements the logic required to arbitrate between the traffic light and the lead vehicle at the intersection. Open the **Arbitration Logic** Stateflow chart.

```
open_system("TrafficLightDecisionLogic/Arbitration Logic");
```



The **Arbitration Logic** Stateflow chart consists of two states, `OnEntry` and `OnRedAndYellowLightDetection`. If the traffic light state is green or if there are no traffic light detections, the state remains in the `OnEntry` state. If the traffic light state is red or yellow, then the state transitions to the `OnRedAndYellowLightDetection` state. The control flow switches between these states based on `trafficLightDetection` and `distanceToTrafficLight` variables. In each state, relative distance and relative velocity with respect to the most important object (MIO) are calculated. The lead vehicle and the red traffic light are considered as MIOs.

OnEntry:

```
relativeDistance = relativeDistToLeadCar;

relativeVelocity = relativeVelocityOfLeadCar;
```

OnRedAndYellowLightDetection:

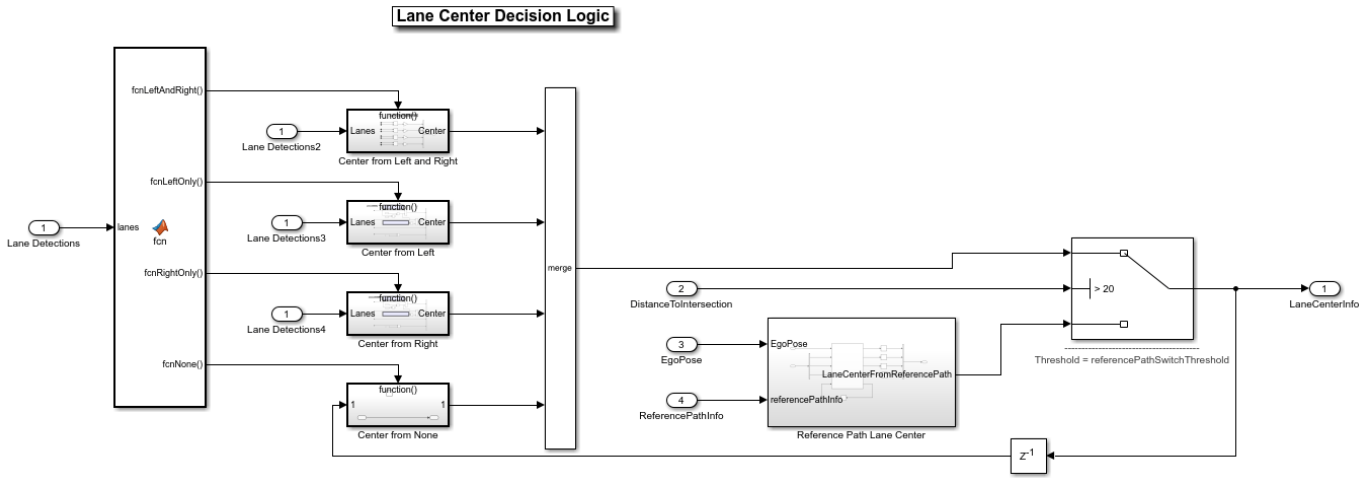
```
relativeDistance = min(relativeDistToLeadCar,distanceToTrafficLight);

relativeVelocity = min(relativeVelocityOfLeadCar,longitudinalVelocity);
```

The **longitudinalVelocity** represents the longitudinal velocity of the ego vehicle.

The Compute Distance To Intersection block computes the distance to the intersection center from the current ego position. Because the intersection has no lanes, the ego vehicle uses this distance to fall back to the predefined reference path at the intersection.

The **Lane Center Decision Logic** subsystem calculates the lane center information as required by the Path Following Control System. Open the **Lane Center Decision Logic** subsystem.

```
open_system("TrafficLightDecisionLogic/Lane Center Decision Logic");
```



The **Lane Center Decision Logic** subsystem primarily relies on the lane detections from the Vision Detection Generator (Automated Driving Toolbox) block to estimate lane center information like curvature, curvature derivative, lateral offset, and heading angle. However, there are no lane markings to detect at the intersection. In such cases, the lane center information can be estimated from a predefined reference path.

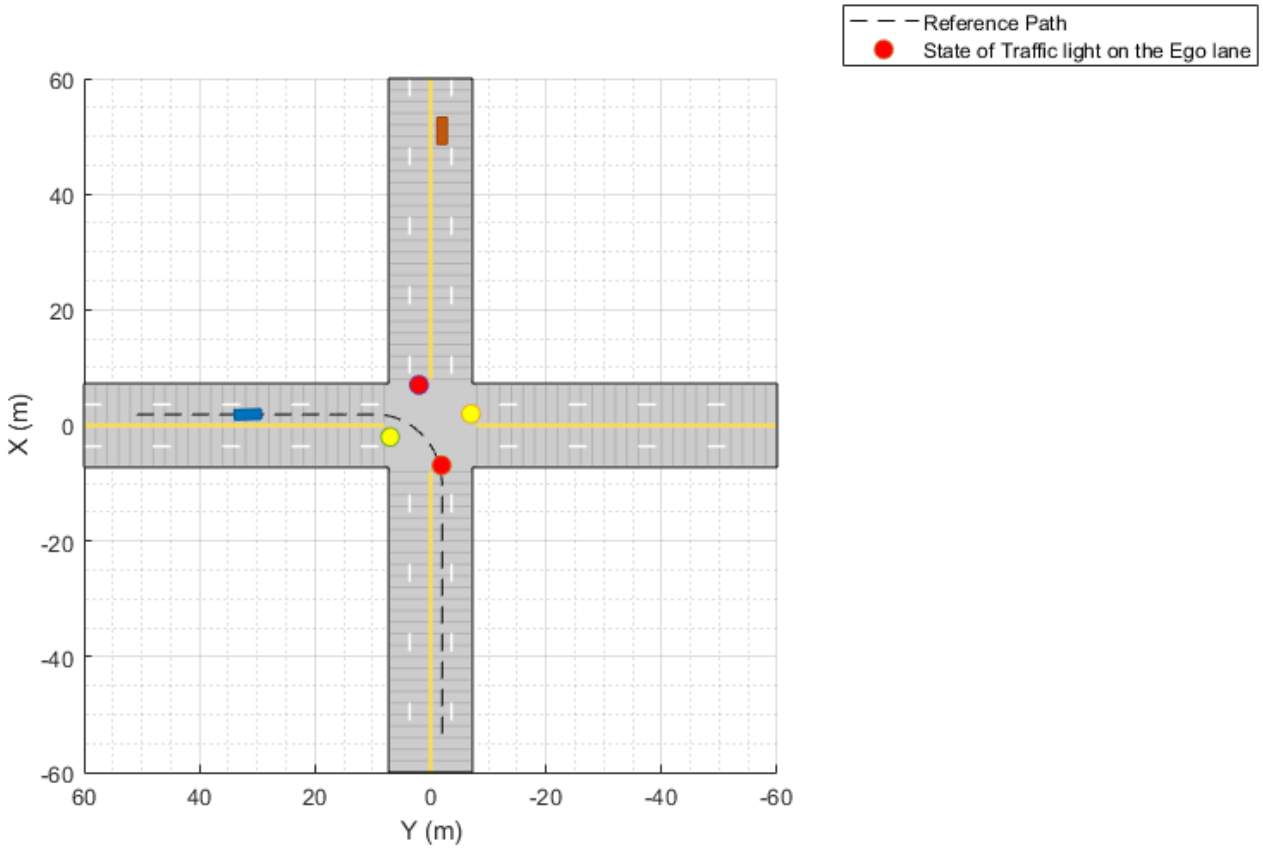
The **Reference Path Lane Center** subsystem computes lane center information based on the current ego pose and predefined reference path. A switch is configured to use **LaneCenterFromReferencePath** when **DistanceToIntersection** is less than `referencePathSwitchThreshold`. This threshold is currently set to 20 meters and can be changed in the `helperSLTrafficLightNegotiationSetup` script.

Simulate Left Turn with Traffic Light and Lead Vehicle

In this test scenario, a lead vehicle travels in the ego lane and crosses the intersection. The traffic light state keeps green for the lead vehicle and turns red for the ego vehicle. The ego vehicle is expected to follow the lead vehicle, negotiate the traffic light, and make a left turn.

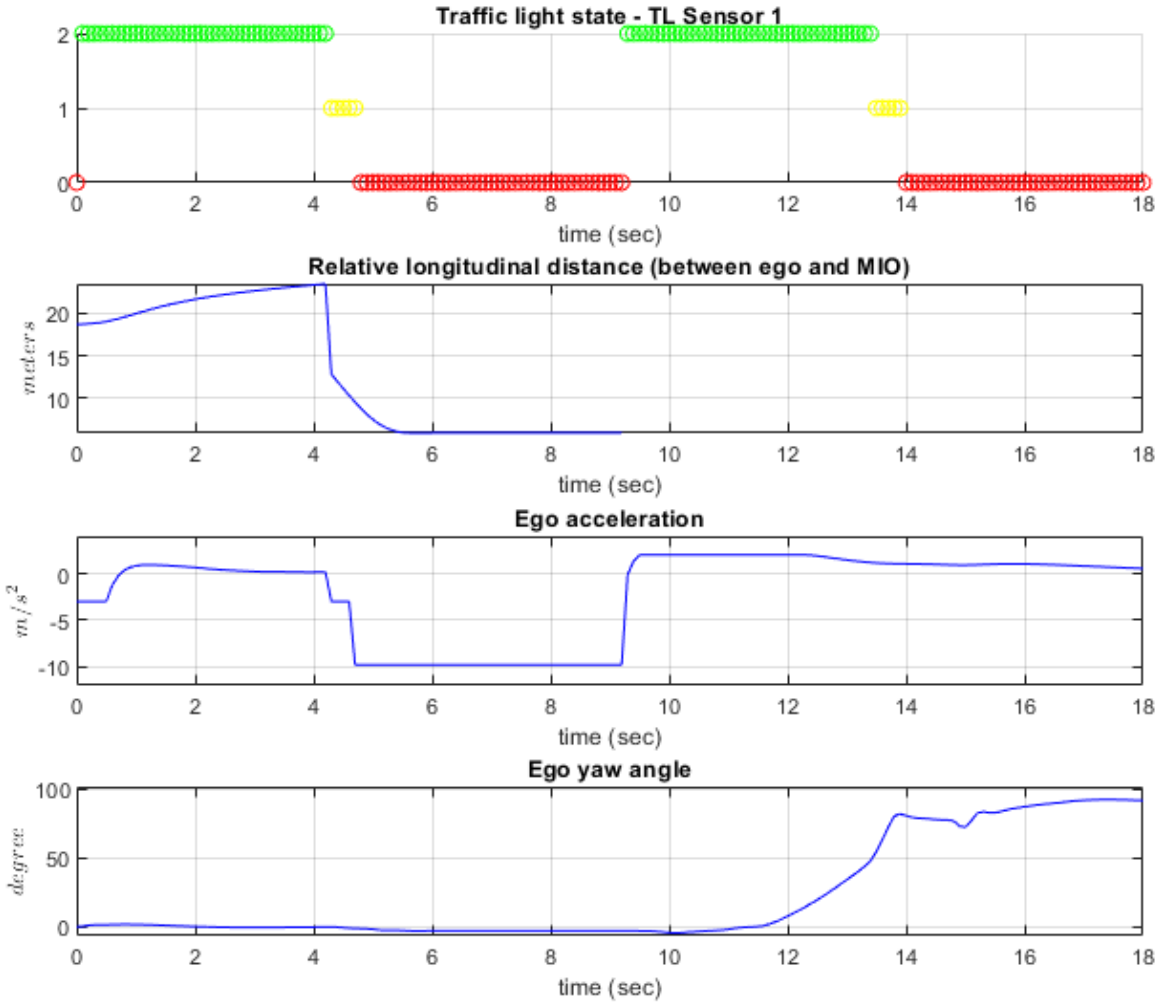
Configure the `TrafficLightNegotiationTestBench` model to use the `scenario_03_TLN_left_turn_with_lead_vehicle` scenario.

```
helperSLTrafficLightNegotiationSetup("scenario_03_TLN_left_turn_with_lead_vehicle");
% To reduce command-window output, first turn off the MPC update messages.
mpcverbosity('off');
% Simulate the model.
sim("TrafficLightNegotiationTestBench");
```



Plot the simulation results.

```
hFigResults = helperPlotTrafficLightNegotiationResults(logsout);
```



Examine the results.

- The **Traffic light state - TL Sensor 1** plot shows the traffic light sensor states of **TL Sensor 1**. It changes from green to yellow, then from yellow to red, and then repeats in **Cycle** mode.
- The **Relative longitudinal distance** plot shows the relative distance between the ego vehicle and the MIO. Notice that the ego vehicle follows the lead vehicle from 0 to 4.2 seconds by maintaining a safe distance from it. You can also observe that from 4.2 to 9 seconds, this distance reduces because the red traffic light is detected as an MIO. Also notice the gaps representing infinite distance when there is no MIO after the lead vehicle exceeds the maximum distance allowed for an MIO.
- The **Ego acceleration** plot shows the acceleration profile from the **Lane Following Controller**. Notice the negative acceleration from 4.2 to 4.7 seconds, in reaction to the detection of the red traffic light as an MIO. You can also observe the increase in acceleration after 9 seconds, in response to the green traffic light.

- The **Ego yaw angle** plot shows the yaw angle profile of the ego vehicle. Notice the variation in this profile after 12 seconds, in response to the ego vehicle taking a left turn.

Close the figure.

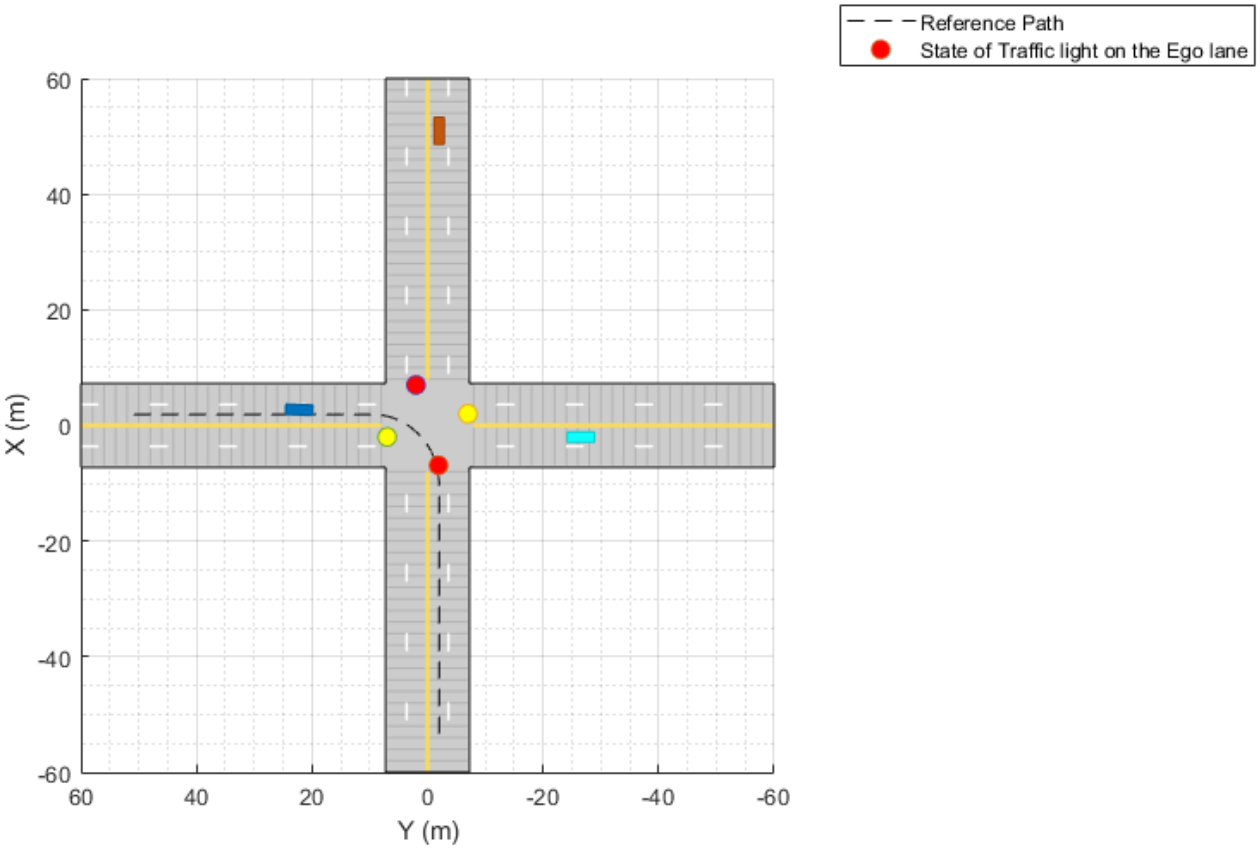
```
close(hFigResults);
```

Simulate Left Turn with Traffic Light and Cross Traffic

This test scenario is an extension to the previous scenario. In addition to the previous conditions, in this scenario, a slow-moving cross-traffic vehicle is in the intersection when the traffic light is green for the ego vehicle. The ego vehicle is expected to wait for the cross-traffic vehicle to pass the intersection before taking the left turn.

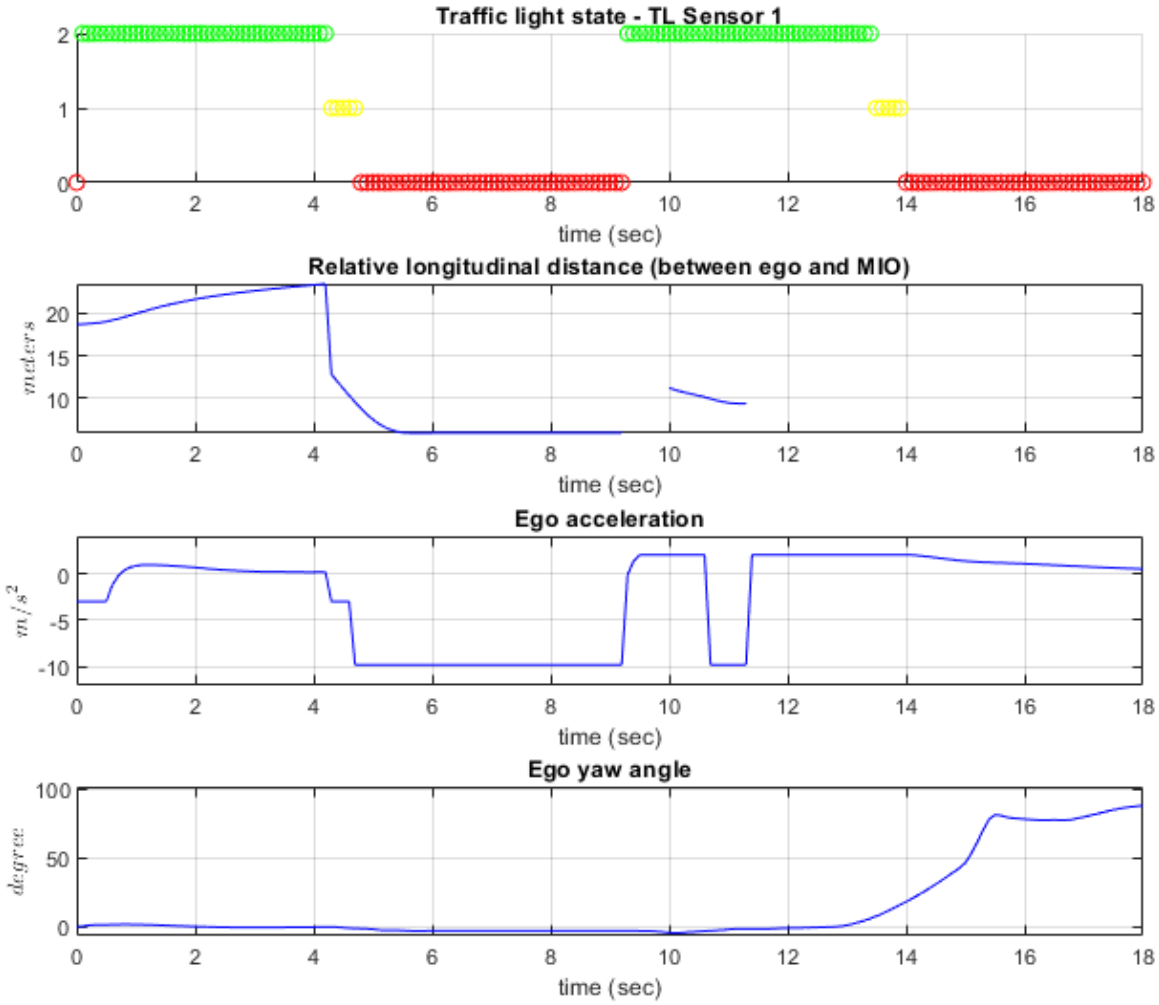
Configure the `TrafficLightNegotiationTestBench` model to use the `scenario_02_TLN_left_turn_with_cross_over_vehicle` scenario.

```
helperSLTrafficLightNegotiationSetup("scenario_02_TLN_left_turn_with_cross_over_vehicle");

% Simulate the model.
sim("TrafficLightNegotiationTestBench");
```



Plot the simulation results.

```
hFigResults = helperPlotTrafficLightNegotiationResults(logsout);
```



Examine the results.

- The **Traffic light state - TL Sensor 1** plot is same as the one from the previous simulation.
- The **Relative longitudinal distance** plot diverges from the previous simulation run from 10.5 seconds onward. Notice the detection of the cross-traffic vehicle as the MIO at 10 seconds at around 10 meters.
- The **Ego acceleration** plot also quickly responds to the cross-traffic vehicle at 10.6. You can notice a hard-braking profile in response to the cross-traffic vehicle at the intersection.
- The **Ego yaw angle** plot shows that the ego vehicle initiates a left turn after 14 seconds, in response to the cross-traffic vehicle leaving the intersection.

Close the figure.

```
close(hFigResults);
```

Explore Other Scenarios

In the previous sections, you explored the system behavior for the `scenario_03_TLN_left_turn_with_lead_vehicle` and `scenario_02_TLN_left_turn_with_cross_over_vehicle` scenarios. Below is a list of scenarios that are compatible with `TrafficLightNegotiationTestBench`.

```
  scenario_01_TLN_left_turn
  scenario_02_TLN_left_turn_with_cross_over_vehicle [Default]
  scenario_03_TLN_left_turn_with_lead_vehicle
  scenario_04_TLN_straight
  scenario_05_TLN_straight_with_lead_vehicle
```

Use these additional scenarios to analyze `TrafficLightNegotiationTestBench` under different conditions. For example, while learning about the interactions between the traffic light decision logic and controls, it can be helpful to begin with a scenario that has an intersection with a traffic light but no vehicles. To configure the model and workspace for such a scenario, use this code:

```
helperSLTrafficLightNegotiationSetup("scenario_04_TLN_straight");
```

Enable the MPC update messages.

```
mpcverbosity('on');
```

Conclusion

In this example, you implemented decision logic for traffic light negotiation and tested it with a lane following controller in a closed-loop Simulink model.

See Also

More About

- “Traffic Light Negotiation with Unreal Engine Visualization” on page 11-162
- “Automated Driving Using Model Predictive Control” on page 11-2

Traffic Light Negotiation with Unreal Engine Visualization

This example shows how to design and simulate a vehicle to negotiate traffic lights in the Unreal Engine® driving simulation environment.

Introduction

Decision logic for negotiating traffic lights is a fundamental component of automated driving applications. The decision logic interacts with a controller to steer the ego vehicle based on the state of the traffic light and other vehicles in the ego lane. Simulating real-world traffic scenarios with realistic conditions can provide more insight into the interactions between the decision logic and the controller. Automated Driving Toolbox™ provides a 3D simulation environment powered by Unreal Engine® from Epic Games®. You can use this engine to visualize the motion of a vehicle in a prebuilt 3D scene. This engine provides an intuitive way to analyze the performance of decision logic and control algorithms when negotiating a traffic light at an intersection.

For information on how to design the decision logic and controls for negotiating traffic lights in a cuboid environment, see the “Traffic Light Negotiation” (Automated Driving Toolbox) example. This example shows how to control a traffic light in an Unreal scene and then how to simulate and visualize vehicle behavior for different test scenarios. In this example, you will:

1. **Explore the architecture of the test bench model**: The model contains sensors and environment, traffic light decision logic, controls, and vehicle dynamics.
2. **Control traffic light in an Unreal scene**: The `Simulation 3D Traffic Light Controller` helper block configures the model to control the state of a traffic light in an Unreal scene by using Simulink®.
3. **Simulate vehicle behavior during green to red transition**: The model analyzes the interactions between the decision logic and the controller when the traffic light state transitions from green to red and the ego vehicle is at a distance of 10 meters from the stop line.
4. **Simulate vehicle behavior during red to green transition**: The model analyzes the interactions between the decision logic and the controller when the traffic light transitions from red to green and the ego vehicle is at a distance of 11 meters from stop line. In this case, the ego vehicle also negotiates traffic light as another vehicle crosses the intersection.
5. **Explore other scenarios**: These scenarios test the system under additional conditions.

You can apply the modeling patterns used in this example to test your own decision logic and controls to negotiate traffic lights in an Unreal scene.

In this example, you enable system-level simulation through integration with the Unreal Engine. This environment requires a Windows® 64-bit platform.

```
if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'
end
```

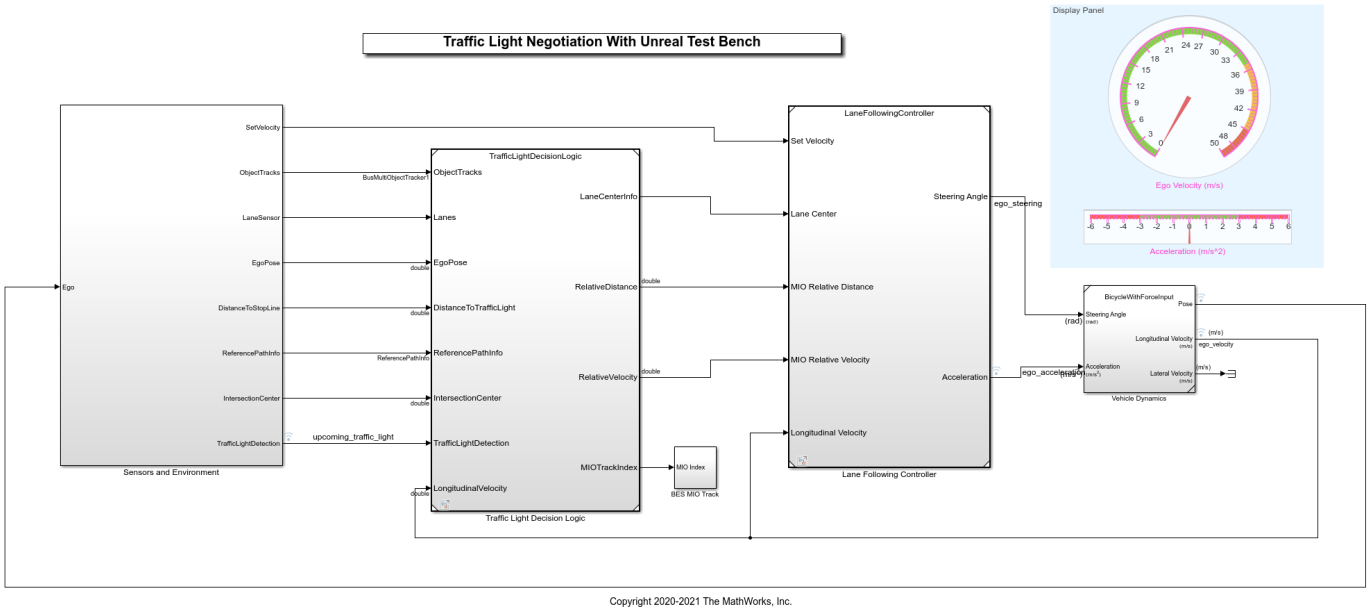
Explore Architecture of Test Bench Model

To explore the test bench model, copy the project example files to a working folder. Use `workDir` argument of the `helperDrivingProjectSetup` function to specify the file path. The length of file path must be less than 25 characters to avoid maximum character limit for Windows file path.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('TLNUnreal.zip', 'workDir', pwd);
```

To explore the behavior of the traffic light negotiation system, open the simulation test bench model for the system.

```
open_system("TrafficLightNegotiationWithUnrealTestBench");
```



Opening this model runs the `helperSLTrafficLightNegotiationWithUnrealSetup` script to initialize the test scenario stored as a `drivingScenario` (Automated Driving Toolbox) object in the base workspace. The default test scenario, `scenario_03_TLN_straight_greenToRed_with_lead_vehicle`, contains one ego vehicle and two non-ego vehicles. This setup script also configures the controller design parameters, vehicle model parameters, and Simulink® bus signals to define the inputs and outputs for the `TrafficLightNegotiationWithUnrealTestBench` model.

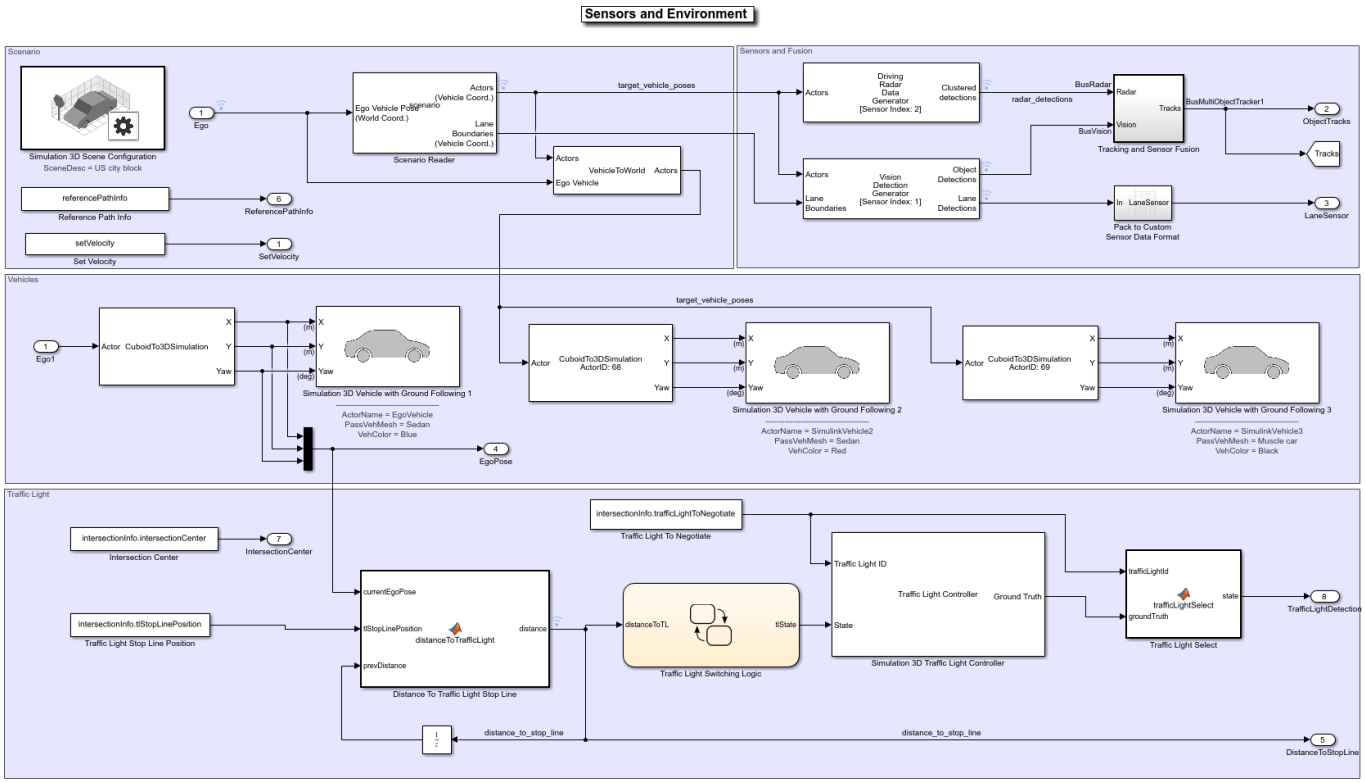
The test bench model contains the following subsystems:

1. **Sensors and Environment**: Models the road network, vehicles, camera, and radar sensors used for simulation. The subsystem uses the `Simulation 3D Traffic Light Controller` helper block to control the state of traffic lights in an Unreal scene.
2. **Traffic Light Decision Logic**: Arbitrates between the traffic light and other lead vehicles or cross-traffic vehicles at the intersection.
3. **Lane-Following Controller**: Generates longitudinal and lateral controls for the ego vehicle.
4. **Vehicle Dynamics**: Models the ego vehicle using a Bicycle Model (Automated Driving Toolbox) block and updates its state using commands received from the **Lane Following Controller** reference model.

The **Traffic Light Decision Logic**, **Lane Following Controller** reference models, and **Vehicle Dynamics** subsystem are reused from the “Traffic Light Negotiation” (Automated Driving Toolbox) example. This example modifies the **Sensors and Environment** subsystem to make it compatible for simulation with an Unreal scene.

The **Sensors and Environment** subsystem configures the road network, sets vehicle positions, synthesizes sensors, and fuses the vehicle detections from the radar and vision sensors. Open the **Sensors and Environment** subsystem.

```
open_system("TrafficLightNegotiationWithUnrealTestBench/Sensors and Environment");
```



Select Scenario

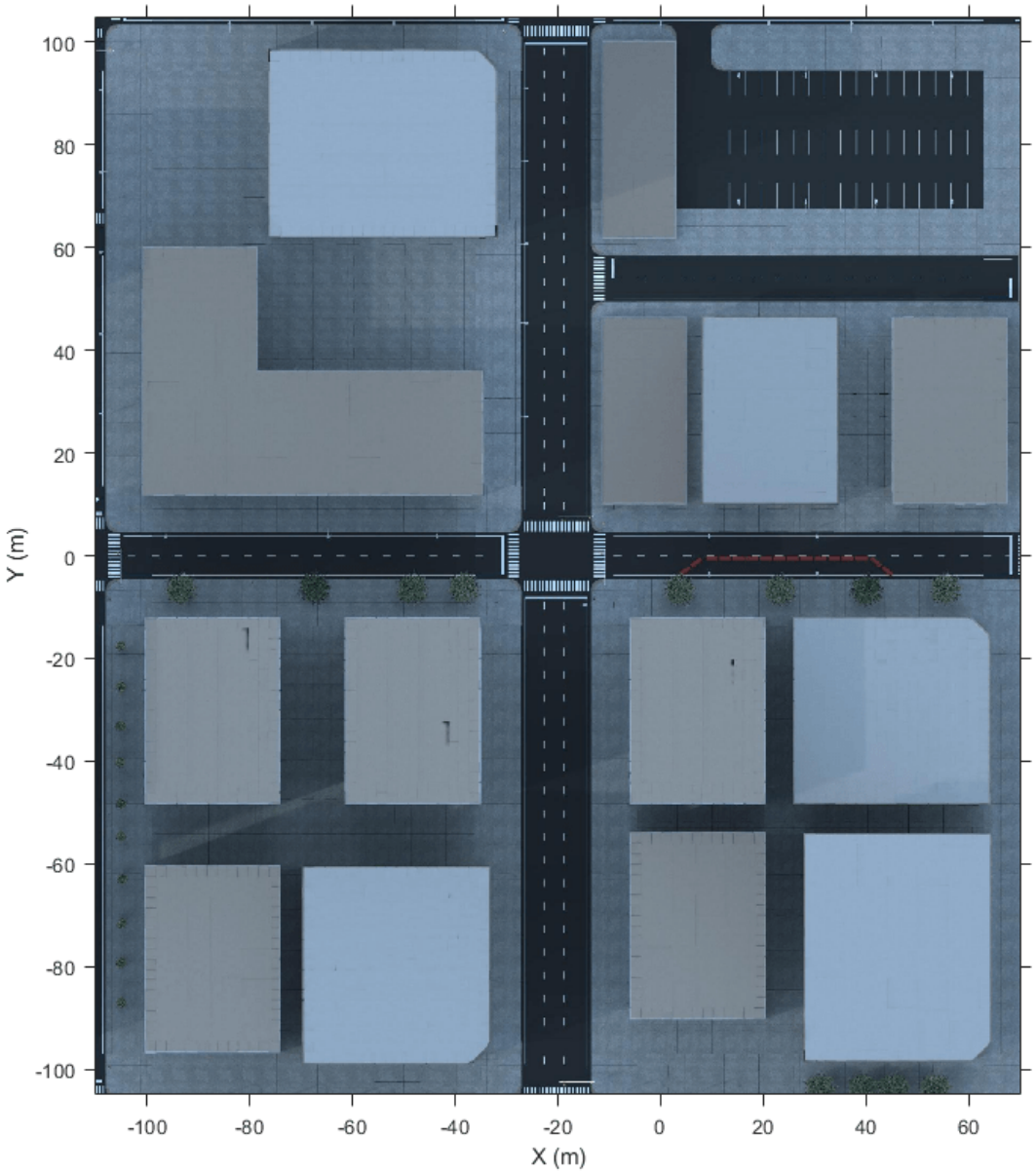
The scene and road network required for the test bench model are specified by the following parts of this subsystem:

- The scene name parameter `Scene name` of the Simulation 3D Scene Configuration (Automated Driving Toolbox) block is set to US City Block (Automated Driving Toolbox). The US city block road network consists of fifteen one-way intersections with two traffic lights at each intersection. This example uses a section of the US city block scene to test the model.
- The Scenario Reader (Automated Driving Toolbox) block takes the ego vehicle information as input and performs a closed-loop simulation. This block reads the `drivingScenario` object `scenario` from the base workspace. The scenario contains the desired road network. The road network closely matches with a section of the US city block scene and contains one intersection.

You can display the selected section of the US city block scene by using the `helperDisplayTrafficLightScene` function.

Specify the x and the y limits to select the desired scene area and plot the extracted scene.

```
xlimit = [-110 70];
ylimit = [-105 105];
hFigure = helperDisplayTrafficLightScene(xlimit, ylimit);
snapnow;
close(hFigure);
```



The `helperGetTrafficLightScenario` function specifies a reference path for the ego vehicle to follow when the lane information is not available. The **Reference Path Info** block reads the reference path stored in the base workspace variable `referencePathInfo`. The ego vehicle can either go straight or take a left turn at the intersection based on the reference trajectory. You can select one of these reference trajectories by setting the input values of `helperGetTrafficLightScenario` function. Set the value to

- `Straight` - To make the ego vehicle travel straight through the intersection.
- `Left` - To make the ego vehicle take a left turn at the intersection.

The **Set Velocity** block reads the velocity value from the base workspace variable `setVelocity` and gives as input to the controller.

Set Vehicle Positions

The scenario contains one ego vehicle and two non-ego vehicles. The positions for each vehicle in the scenario are specified by these parts of the subsystem:

- The Simulation 3D Vehicle with Ground Following (Automated Driving Toolbox) block provides an interface that changes the position and orientation of the vehicle in the 3D scene.
- The **Ego** input port controls the position of the ego vehicle, which is specified by the `Simulation 3D Vehicle with Ground Following 1` block. The `ActorName` mask parameter of `Simulation 3D Vehicle with Ground Following 1` block is specified as `EgoVehicle`.
- The Cuboid To 3D Simulation (Automated Driving Toolbox) block converts the ego pose coordinate system (with respect to below the center of the vehicle rear axle) to the 3D simulation coordinate system (with respect to below the vehicle center).
- The Scenario Reader (Automated Driving Toolbox) block also outputs ground truth information of lanes and actor poses in ego vehicle coordinates for the target vehicles. There are two target vehicles in this example, which are specified by the other Simulation 3D Vehicle with Ground Following blocks.
- The Vehicle To World (Automated Driving Toolbox) block converts the actor pose coordinates from ego vehicle coordinates to the world coordinates.

The **Tracking and Sensor Fusion** subsystem fuses vehicle detections from Driving Radar Data Generator (Automated Driving Toolbox) and Vision Detection Generator (Automated Driving Toolbox) blocks and tracks the fused detections using Multi-Object Tracker (Automated Driving Toolbox) block to provide object tracks surrounding the ego vehicle. The Vision Detection Generator block also provides lane detections with respect to the ego vehicle that helps in identifying vehicles present in the ego lane.

Control Traffic Light in Unreal Scene

This model uses the **Simulation 3D Traffic Light Controller** helper block to configure and control the state of traffic lights in an Unreal scene. The **Simulation 3D Traffic Light Controller** helper block controls the state of traffic lights by using `Timer-Based` or `State-Based` mode. You can select the desired mode by using the `Control mode` mask parameter. By default, this model uses `State-Based` mode. For information on `Timer-Based` mode, see the block mask description.

In `State-Based` mode, the block overwrites the state of a traffic light specified by the `Traffic Light ID` input port. The value for the `Traffic Light ID` input port is set by the `intersectionInfo.trafficLightToNegotiate` variable in the `helperGetTrafficLightScenario` function. In this model, the value for `Traffic Light ID` input port is set to 16. This implies that the block controls the traffic light with ID value 16 in the US city block scene. The states of all the traffic lights present in the US city block scene is returned by the `Ground Truth` output port of the `Simulation 3D Traffic Light Controller` helper block. The model tests the decision logic and controls by using the ground truth information and does not require perception-based traffic light detection.

The `Traffic Light Select` block extracts the state of the traffic light with ID value 16 from the `Ground Truth` output. The **Traffic Light Decision Logic** reference model uses the state value to arbitrate between the lead car and the traffic light. For more information about the **Traffic Light Decision Logic** reference model, see the “Traffic Light Negotiation” (Automated Driving Toolbox) example.

The **Traffic Light Stop Line Position** block provides the stop line position at the intersection corresponding to the selected traffic light `trafficLightToNegotiate`. The stop line position value is specified by `intersectionInfo.tlStopLinePosition`.

The **Intersection Center** block provides the position of the intersection center of the road network in the scenario. This is obtained using the `intersectionInfo`, an output from `helperGetTrafficLightScenario`.

It is often important to test the decision logic and controls when the ego vehicle is close to the traffic light and the traffic light changes its state. The model used in this example enables traffic lights to change state when the `EgoVehicle` is close to the traffic light.

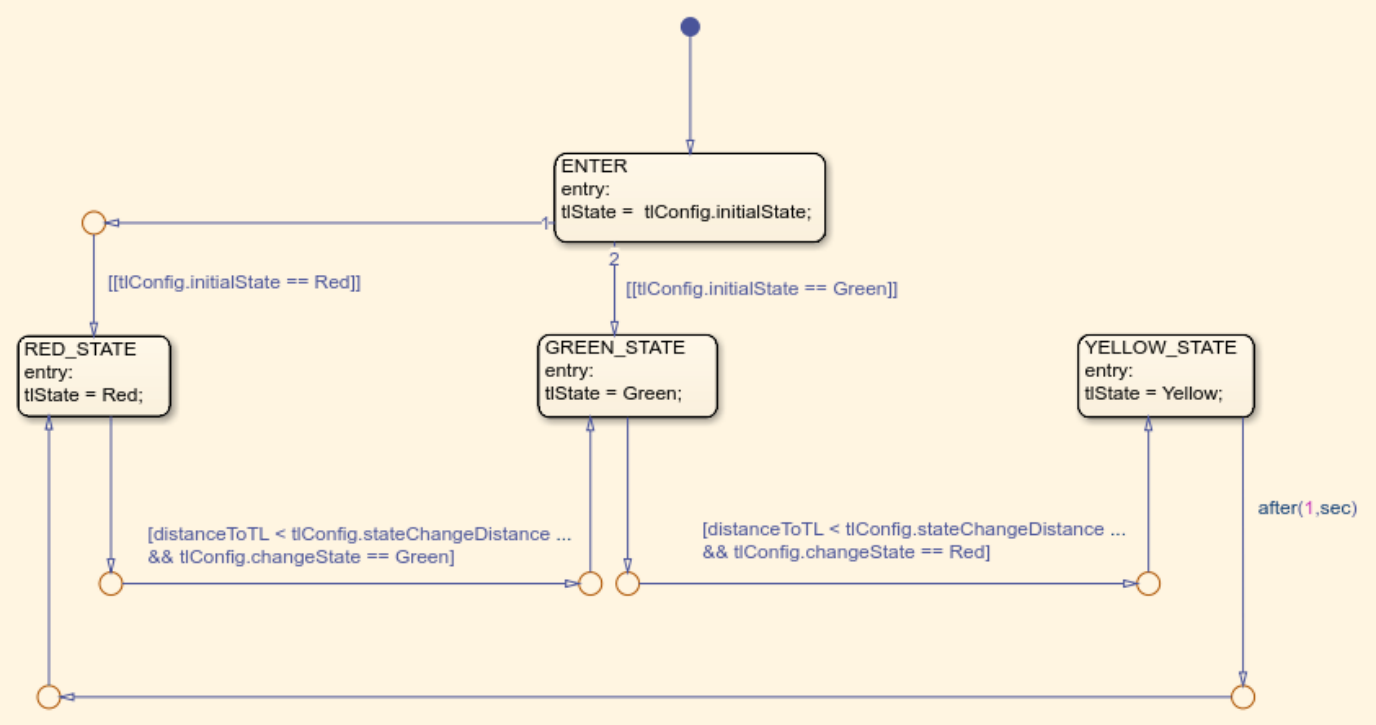
The `Distance To Traffic Light Stop Line` block calculates the Euclidean distance between the stop line corresponding to the selected traffic light `trafficLightToNegotiate` and the current ego vehicle position.

The **Traffic Light Decision Logic** uses the distance value to decide the most important object (MIO), the closest object in front of the ego vehicle. It can be the lead vehicle or traffic light in the ego lane.

The **Traffic Light Switching Logic** block outputs `tlState`, the state of the traffic light that needs to be set. This is implemented using Stateflow® and uses the distance value to trigger a state change when the `EgoVehicle` is closer to the traffic light than the specified distance.

Open the **Traffic Light Switching Logic** block.

```
open_system("TrafficLightNegotiationWithUnrealTestBench/Sensors and Environment/Traffic Light Sw
```



Traffic Light Switching Logic uses the `Configuration params` mask parameter to read the traffic light configuration, `trafficLightConfig`, from the base workspace. You can use the

`trafficLightConfig` structure to configure different test scenarios. This structure is defined in the test scenario function and has the following fields: `stateChangeDistance`, `initialState`, and `changeState`.

- `initialState` specifies the state of the traffic light before the state change.
- `stateChangeDistance` specifies the threshold distance of the `EgoVehicle` to the traffic light at which state change should happen.
- `changeState` specifies the state of the traffic light to be set after state change.

State switching happens based on the set configuration and when `EgoVehicle` reaches `stateChangeDistance`. When the `initialState` is `Red` and `changeState` is `Green` the Stateflow chart switches from `Red` state to `Green` state. Conversely, when the `initialState` is `Green` and `changeState` is `Red` the Stateflow chart is modeled such that the state transition happens from `Green` state to `Yellow` state and after one second, the traffic light switches to `Red` state.

Simulate Vehicle Behavior During Green To Red Transition

This section tests the decision logic when the ego vehicle is at a close distance to the traffic light and the traffic light state changes from green to red. In this test scenario, a lead vehicle travels in the ego lane and crosses the intersection. The traffic light state keeps green for the lead vehicle and turns red when the ego vehicle is at a distance of 10 meters from the stop line. The ego vehicle is expected to follow the lead vehicle, negotiate the state transition, and come to a complete halt before the stop line.

Configure the `TrafficLightNegotiationWithUnrealTestBench` model to use the `scenario_03_TLN_straight_greenToRed_with_lead_vehicle` test scenario.

```
helperSLTrafficLightNegotiationWithUnrealSetup(...
    "scenario_03_TLN_straight_greenToRed_with_lead_vehicle");
```

Display the `trafficLightConfig` structure parameters set for the test scenario.

```
disp(trafficLightConfig');
```

```
           initialState: 2
    stateChangeDistance: 10
            changeState: 0
```

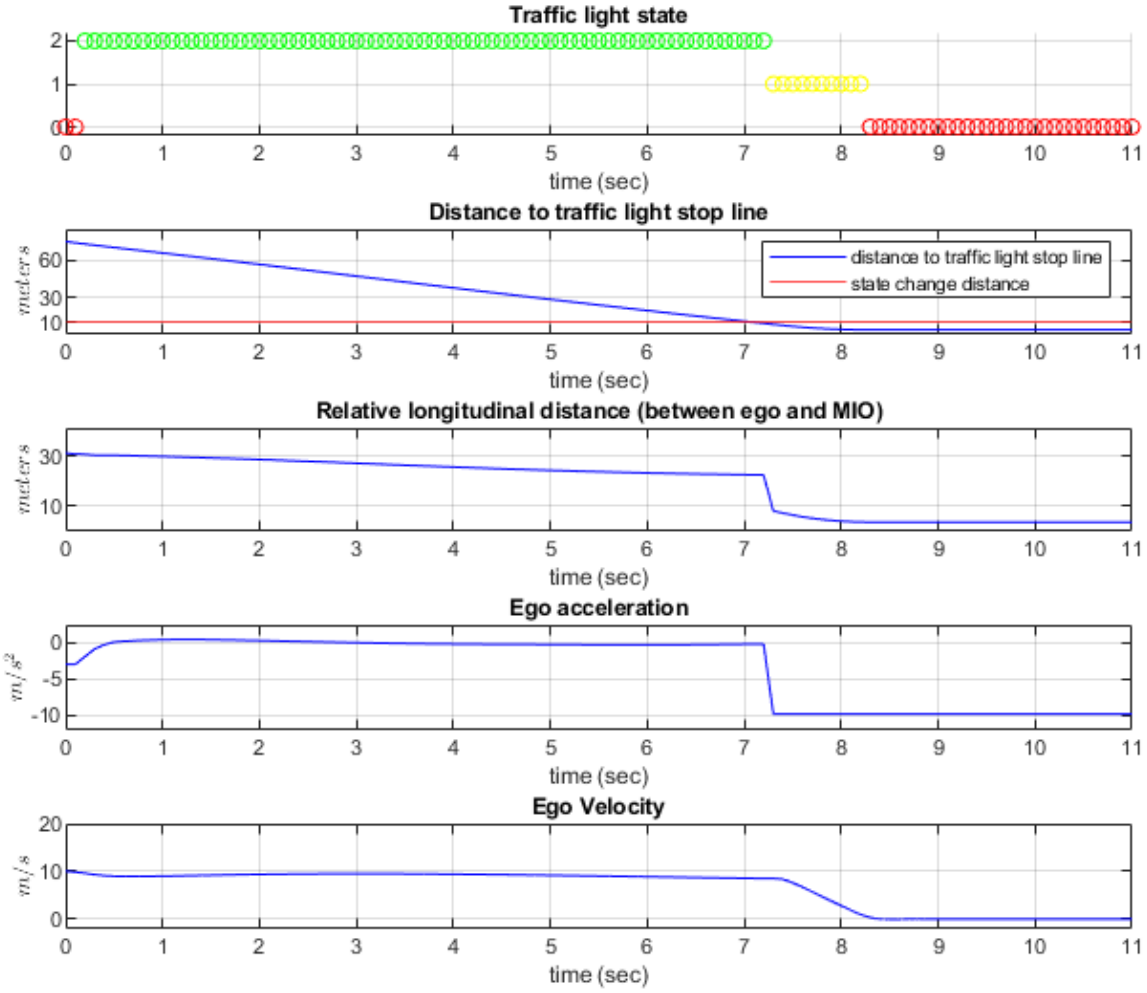
Simulate the model. During the simulation, the model logs the signals required for post simulation analysis to `logsout`.

To reduce command-window output, first turn off the MPC update messages.

```
mpcverbosity('off');
sim("TrafficLightNegotiationWithUnrealTestBench");
```

Plot the simulation results using `helperPlotTrafficLightControlAndNegotiationResults` function.

```
hFigResults = helperPlotTrafficLightControlAndNegotiationResults(logsout, trafficLightConfig.sta
```



Examine the results.

- The **Traffic light state** plot shows the state of the traffic light. The **Distance to traffic light stop line** plot shows the distance between the ego vehicle and the stop line corresponding to the traffic light. You can see that the initial state of the traffic light is green and the state changes from green to yellow as the ego vehicle approaches the stop line. The state changes from yellow to red when the ego vehicle is at a distance of 10 meters from the stop line.
- The **Relative longitudinal distance** plot shows the relative distance between the ego vehicle and the most important object (MIO). The MIO is the closest object in front of the ego vehicle. It can be a lead vehicle or a traffic light in the ego lane. The ego vehicle follows the lead vehicle and maintains a safe distance when the traffic light state is green. The distance between the ego and the lead vehicle decreases when the traffic light transitions from green to red. This is because, as the ego vehicle approaches the stop line, the traffic light is detected as an MIO. At this point of time, the traffic light state is either red or yellow.

- The **Ego acceleration** plot shows the acceleration profile from the **Lane Following Controller**. Notice that this closely follows the dip in the relative distance, in reaction to the detection of the red traffic light as an MIO.
- The **Ego velocity** plot shows the velocity profile of the ego vehicle. Notice that the ego velocity slows down in reaction to the yellow and red traffic lights and comes to a complete halt before the stop line. This can be verified by comparing the plot with **Distance to traffic light stop line**, when the velocity is zero.

You can refer to the “Traffic Light Negotiation” (Automated Driving Toolbox) example to learn more about this analysis and the interactions between the decision logic and the controller.

Close the figure.

```
close(hFigResults);
```

Simulate Vehicle Behavior During Red To Green Transition

This section tests the decision logic when the ego vehicle is at a close distance to the traffic light and the traffic light state changes from red to green. In addition, a cross-traffic vehicle is in the intersection when the traffic light is green for the ego vehicle. The traffic light state is initially red for the ego vehicle and turns green when the ego vehicle is at a distance of 11 meters from the stop line. The ego vehicle is expected to slow down as it approaches the traffic light when the state is red and must start accelerating when the traffic light state changes from red to green. It is also expected to wait for the cross-traffic vehicle to pass the intersection before accelerating to continue its travel.

The test scenario function `scenario_04_TLN_straight_redToGreen_with_cross_vehicle` implements this scenario. Configure the `TrafficLightNegotiationWithUnrealTestBench` model to use this scenario.

```
helperSLTrafficLightNegotiationWithUnrealSetup(...
    "scenario_04_TLN_straight_redToGreen_with_cross_vehicle");
```

Display the `trafficLightConfig` structure parameters that are set for this test scenario.

```
disp(trafficLightConfig');
```

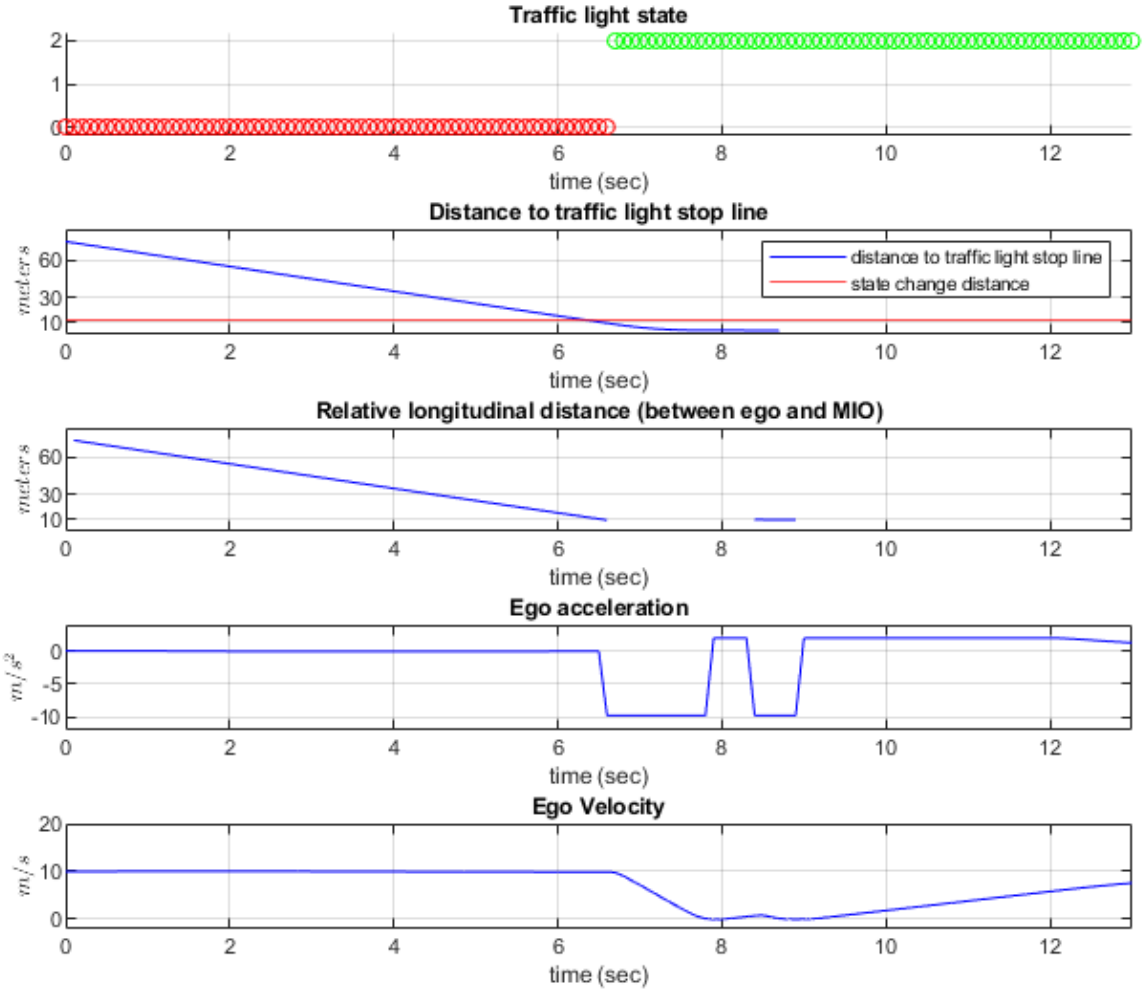
```
           initialState: 0
    stateChangeDistance: 11
            changeState: 2
```

Simulate the model.

```
sim("TrafficLightNegotiationWithUnrealTestBench");
```

Plot the simulation results.

```
hFigResults = helperPlotTrafficLightControlAndNegotiationResults(logsout, trafficLightConfig.sta
```



Examine the results.

- The **Traffic light state** plot shows that the initial traffic light state is red. The traffic light state changes from red to green when the ego vehicle is at a distance of 11 meters from the stop line.
- The **Relative longitudinal distance** plot closely follows the **Distance to traffic light stop line** plot because there is no lead vehicle. Notice the sudden dip in the relative distance in response to the detection of the cross-over vehicle.
- The **Ego acceleration** plot shows that the ego vehicle attempts to slow down on seeing the red traffic light. However, in response to the state change to green, you can observe an increase in acceleration. You can then notice a hard-braking profile in response to the cross-traffic vehicle at the intersection.
- The **Ego velocity** plot closely follows the **Ego acceleration** plot and shows a decrease in velocity as the ego vehicle approaches the intersection. You can also notice a slight increase in velocity in

response to green traffic light and subsequent decrease in velocity in response to the cross-traffic vehicle.

Close the figure.

```
close(hFigResults);
```

Explore Other Scenarios

In the previous sections, you explored the system behavior for the `scenario_03_TLN_straight_greenToRed_with_lead_vehicle` and `scenario_04_TLN_straight_redToGreen_with_cross_vehicle` scenarios. Below is a list of scenarios that are compatible with `TrafficLightNegotiationWithUnrealTestBench`.

```
scenario_01_TLN_left_redToGreen_with_lead_vehicle
scenario_02_TLN_straight_greenToRed
scenario_03_TLN_straight_greenToRed_with_lead_vehicle [Default]
scenario_04_TLN_straight_redToGreen_with_cross_vehicle
```

Use these additional scenarios to analyze `TrafficLightNegotiationWithUnrealTestBench` under different conditions.

Enable the MPC update messages.

```
mpcverbosity('on');
```

You can use the modeling patterns in this example to build your own traffic light negotiation application.

See Also

More About

- “Traffic Light Negotiation” on page 11-148
- “Automated Driving Using Model Predictive Control” on page 11-2