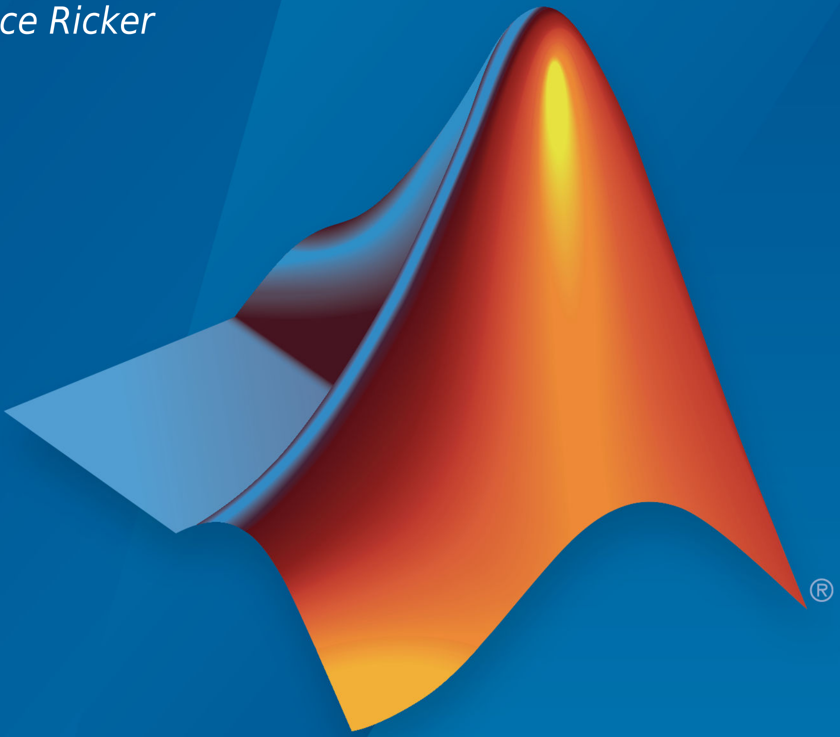
Model Predictive Control Toolbox™

User's Guide

Alberto Bemporad
Manfred Morari
N. Lawrence Ricker



MATLAB®

R2018**a**

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Model Predictive Control Toolbox ™ User's Guide

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)

Contents

Introduction

1

4 Case-Study Examples

5 Adaptive MPC Design

6 Explicit MPC Design

Introduction

Specify Scale Factors

Overview

Recommended practice includes specification of scale factors for each plant input and output variable, which is especially important when certain variables have much larger or smaller magnitudes than others.

The scale factor should equal (or approximate) the span of the variable. Span is the difference between its maximum and minimum value in engineering units, that is, the unit of measure specified in the plant model. Internally, MPC divides each plant input and output signal by its scale factor to generate dimensionless signals.

The potential benefits of scaling are as follows:

- Default MPC tuning weights work best when all signals are of order unity. Appropriate scale factors make the default weights a good starting point for controller tuning and refinement.
- When choosing cost function weights, you can focus on the relative priority of each term rather than a combination of priority and signal scale.
- Improved numerical conditioning. When values are scaled, round-off errors have less impact on calculations.

Once you have tuned the controller, changing a scale factor is likely to affect performance and the controller may need retuning. Best practice is to establish scale factors at the beginning of controller design and hold them constant thereafter.

Defining Scale Factors

To identify scale factors, estimate the span of each plant input and output variable in engineering units.

- If the signal has known bounds, use the difference between the upper and lower limit.
- If you do not know the signal bounds, consider running open-loop plant model simulations. You can vary the inputs over their likely ranges, and record output signal spans.
- If you have no idea, use the default scale factor (=1).

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

Once you have set the scale factors and have begun to tune the controller performance, hold the scale factors constant.

Using Commands

After you create the MPC controller object using the `mpc` command, set the scale factor property for each plant input and output variable.

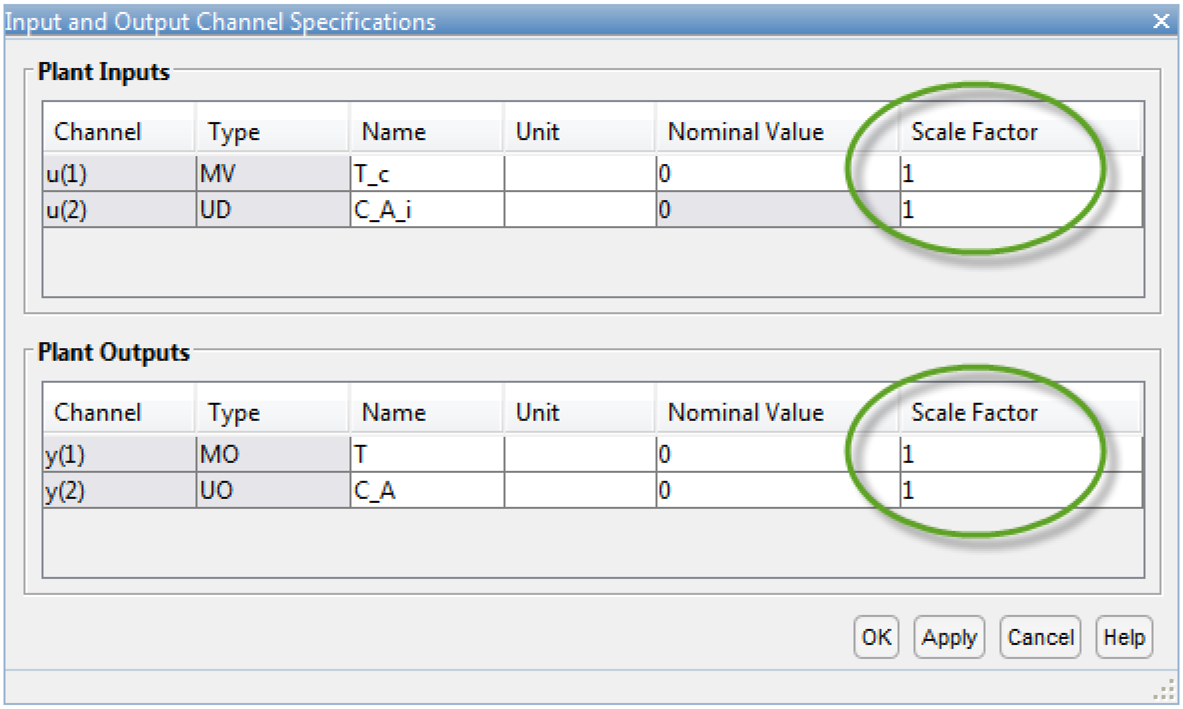
For example, the following commands create a random plant, specify the signal types, and define a scale factor for each signal.

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

Using MPC Designer App

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.



Input and Output Channel Specifications

Plant Inputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
u(1)	MV	T_c		0	1
u(2)	UD	C_A_i		0	1

Plant Outputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
y(1)	MO	T		0	1
y(2)	UO	C_A		0	1

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

See Also

MPC Designer | `mpc`

More About

- “Choose Sample Time and Horizons” on page 1-6
- “Using Scale Factors to Facilitate Weight Tuning”

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, ue the `review` command.
- Recommended practice is to increase p until further increases have a minor impact on performance. If the plant is open-loop unstable, the maximum p is the number of control intervals required for the open-loop step response of the plant to become infinite. $p > 50$ is rarely necessary unless T_s is too small.
- Unfavorable plant characteristics combined with a small p can generate an internally unstable controller. To check for this condition, use the `review` command, and increase p if possible. If p is already large, consider the following:
  - Increase T_s.
  - Increase the cost function weights on MV increments.
  - Modify the control horizon or use MV blocking (see “Manipulated Variable Blocking” on page 2-34).
  - Use a small p with terminal weighting to approximate LQR behavior (See “Terminal Weights and Constraints” on page 2-29).

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:

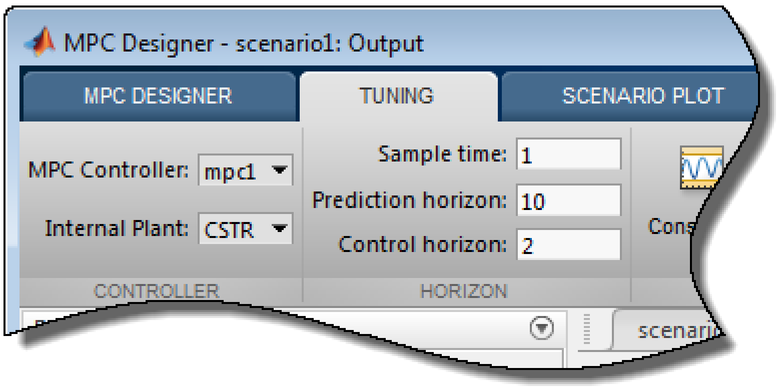
- Small m means fewer variables to compute in the QP solved at each control interval, which promotes faster computations.
- If the plant includes delays, $m < p$ is essential. Otherwise, some MV moves might not affect any of the plant outputs before the end of the prediction horizon, leading to a singular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



MPC Controller: mpc1

Internal Plant: CSTR

Sample time: 1

Prediction horizon: 10

Control horizon: 2

See Also

MPC Designer | `mpc`

More About

- “Specify Constraints” on page 1-10

Specify Constraints

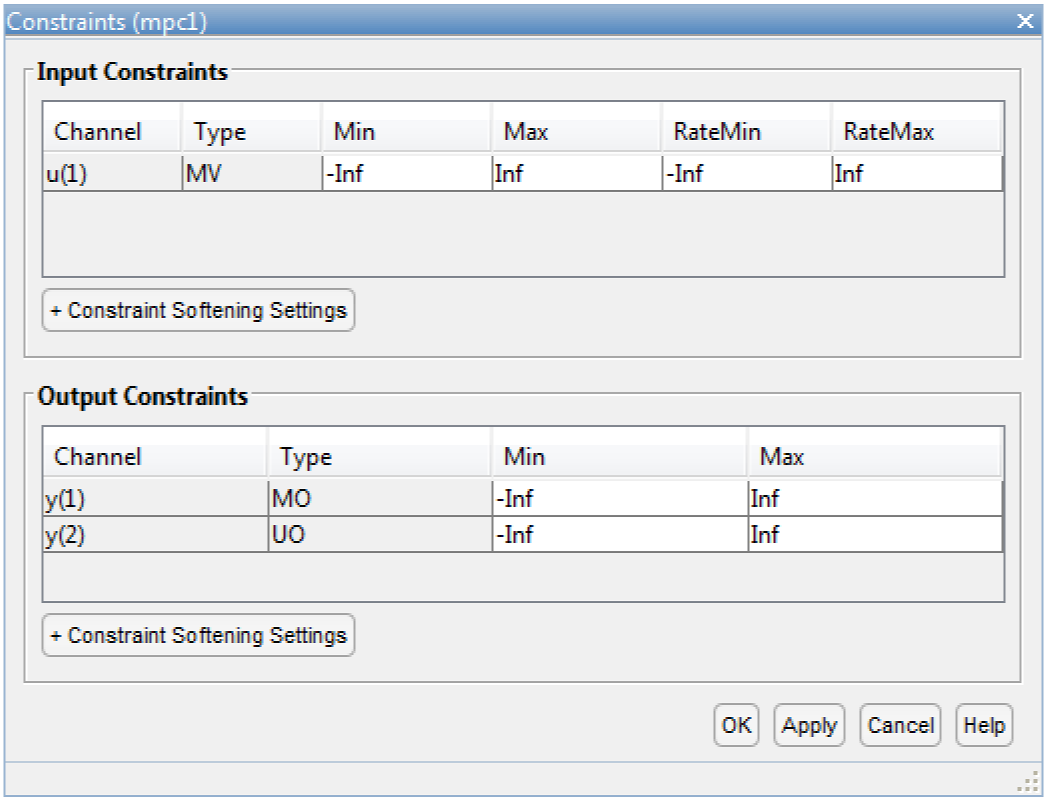
Input and Output Constraints

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 = $u(k) - u(k - 1)$).

To include this constraint	Set this controller property	Soften constraint by setting
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 2-6 for the equations describing the corresponding constraints.


Constraints (mpc1)
+ Constraint Softening Settings
+ Constraint Softening Settings
OK
Apply
Cancel
Help

Input Constraints

Channel	Type	Min	Max	RateMin	RateMax
u(1)	MV	-Inf	Inf	-Inf	Inf

Output Constraints

Channel	Type	Min	Max
y(1)	MO	-Inf	Inf
y(2)	UO	-Inf	Inf

Tips

For MV bounds:

- Include known physical limits on the plant MVs as hard MV bounds.
- Include MV increment bounds when there is a known physical limit on the rate of change, or your application requires you to prevent large increments for some other reason.
- Do not include both hard MV bounds and hard MV increment bounds on the same MV, as they can conflict. If both types of bounds are important, soften one.

For OV bounds:

- Do not include OV bounds unless they are essential to your application. As an alternative to setting an OV bound, you can define an OV reference and set its cost function weight to keep the OV close to its setpoint.
- All OV constraints should be softened.
- Consider leaving the OV unconstrained for some prediction horizon steps. See “Time-Varying Weights and Constraints” on page 2-25.
- Consider a time-varying OV constraint that is easy to satisfy early in the horizon, gradually tapering to a more strict constraint. See “Time-Varying Weights and Constraints” on page 2-25.
- Do not include OV constraints that are impossible to satisfy. Even if soft, such constraints can cause unexpected controller behavior. For example, consider a SISO plant with five sampling periods of delay. An OV constraint before the sixth prediction horizon step is, in general, impossible to satisfy. You can use the `review` command to check for such impossible constraints, and use a time-varying OV bound instead. See “Time-Varying Weights and Constraints” on page 2-25.

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.

When there are hard constraints on plant outputs, or hard custom constraints (on linear combinations of plant inputs and outputs, and the plant is subject to disturbances, QP infeasibility is a distinct possibility.

All Model Predictive Control Toolbox constraints (except slack variable nonnegativity) can be soft. When a constraint is soft, the controller can deem an MV optimal even though it predicts a violation of that constraint. If all plant output, MV increment, and custom constraints are soft (as they are by default), QP infeasibility does not occur. However, controller performance can be substandard.

To soften a constraint, set the corresponding ECR value to a positive value (zero implies a hard constraint). The larger the ECR value, the more likely the controller will deem it optimal to violate the constraint in order to satisfy your other performance goals. The Model Predictive Control Toolbox software provides default ECR values but, as for the cost function weights, you might need to tune the ECR values in order to achieve acceptable performance.

To understand how constraint softening works, suppose that your cost function uses

$w_{i,j}^{u} = w_{i,j}^{\Delta u} = 0$, giving both the MV and MV increments zero weight in the cost function. Only the output reference tracking and constraint violation terms are nonzero. In this case, the cost function is:

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

Suppose that you have also specified hard MV bounds with $V_{j,min}^{u}(i) = 0$ and

$V_{j,max}^{u}(i) = 0$. Then these constraints simplify to:

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

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

$$\frac{y_{j,min}(i)}{s_j^y} - \varepsilon_k V_{j,min}^{y}(i) \le \frac{y_j(k+i|k)}{s_j^y} \le \frac{y_{j,max}(i)}{s_j^y} + \varepsilon_k V_{j,max}^{y}(i),\ i = 1:p, \quad j = 1:n_y.$$

Now, suppose that a disturbance has pushed a plant output above its specified upper bound, but the QP with hard output constraints would be feasible, that is, all constraint violations could be avoided in the QP solution. The QP involves a trade-off between output reference tracking and constraint violation. The slack variable, ε_k, must be nonnegative. Its appearance in the cost function discourages, but does not prevent, an optimal $\varepsilon_k > 0$. 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 $V_{j,max}^{y}(i)$ makes it easier to satisfy the constraint with a small ε_k. In that case,

$$\frac{y_j(k+i|k)}{s_j^y}$$

can be larger, without exceeding

$$\frac{y_{j,max}(i)}{s_j^y} + \varepsilon_k V_{j,max}^{y}(i).$$

Notice that $V_{j,max}^{y}(i)$ does not set an upper limit on the constraint violation. Rather, it is a tuning factor determining whether a soft constraint is easy or difficult to satisfy.

Tips

- Use of dimensionless variables simplifies constraint tuning. Define appropriate scale factors for each plant input and output variable. See “Specify Scale Factors” on page 1-2.

- To indicate the relative magnitude of a tolerable violation, use the ECR parameter associated with each constraint. Rough guidelines are as follows:
  - 0 — No violation allowed (hard constraint)
  - 0.05 — Very small violation allowed (nearly hard)
  - 0.2 — Small violation allowed (quite hard)
  - 1 — average softness
  - 5 — greater-than-average violation allowed (quite soft)
  - 20 — large violation allowed (very soft)
- Use the overall constraint softening parameter of the controller (controller object property: `Weights.ECR`) to penalize a tolerable soft constraint violation relative to the other cost function terms. Set the `Weights.ECR` property such that the corresponding penalty is 1–2 orders of magnitude greater than the typical sum of the other three cost function terms. If constraint violations seem too large during simulation tests, try increasing `Weights.ECR` by a factor of 2–5.

  Be aware, however, that an excessively large `Weights.ECR` distorts MV optimization, leading to inappropriate MV adjustments when constraint violations occur. To check for this, display the cost function value during simulations. If its magnitude increases by more than 2 orders of magnitude when a constraint violation occurs, consider decreasing `Weights.ECR`.
- Disturbances and prediction errors can lead to unexpected constraint violations in a real system. Attempting to prevent these violations by making constraints harder often degrades controller performance.

See Also

`review`

More About

- “Time-Varying Weights and Constraints” on page 2-25
- “Terminal Weights and Constraints” on page 2-29
- “Optimization Problem” on page 2-2

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 2-2 for details on the cost function equations.

Initial Tuning

- Before tuning the cost function weights, specify scale factors for each plant input and output variable. Hold these scale factors constant as you tune the controller. See “Specify Scale Factors” on page 1-2 for more information.
- During tuning, use the `sensitivity` and `review` commands to obtain diagnostic feedback. The `sensitivity` command is intended to help with cost function weight selection.
- Change a weight by setting the appropriate controller property, as follows:

To change this weight	Set this controller property	Array size
OV reference tracking (w^y)	`Weights.OV`	p-by-n_y
MV reference tracking (w^u)	`Weights.MV`	p-by-n_u
MV increment suppression ($w^{\Delta u}$)	`Weights.MVRate`	p-by-n_u

Here, MV is a plant manipulated variable, and n_u is the number of MVs. OV is a plant output variable, and n_y is the number of OVs. Finally,p is the number of steps in the prediction horizon.

If a weight array contains $n < p$ rows, the controller duplicates the last row to obtain a full array of p rows. The default ($n = 1$) minimizes the number of parameters to be tuned, and is therefore recommended. See “Time-Varying Weights and Constraints” on page 2-25 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 *i*th OV. Rough guidelines for this are as follows:

  - 0.05 — Low priority: Large tracking error acceptable
  - 0.2 — Below-average priority
  - 1 — Average priority – the default. Use this value if $n_{yc} = 1$.
  - 5 — Above average priority
  - 20 — High priority: Small tracking error desired

Tips for Setting MV Weights

By default, `Weights.MV = zeros(1,nu)`. If some MVs have targets, the corresponding MV reference tracking weights must be nonzero. Otherwise, the targets are ignored. If the number of MV targets is less than ($n_u - n_{yc}$), try using the same weight for each. A suggested value is 0.2, the same as below-average OV tracking. This value allows the MVs to move away from their targets temporarily to improve OV tracking.

Otherwise, the MV and OV reference tracking goals are likely to conflict. Prioritize by setting the `Weights.MV(:,i)` values in a manner similar to that suggested for `Weights.OV` (see above). Typical practice sets the average MV tracking priority lower than the average OV tracking priority (e.g., 0.2 < 1).

If the *i*th MV does not have a target, set `Weights.MV(:,i)` = 0 (the default).

Tips for Setting MVRate Weights

- By default, `Weights.MVRate = 0.1*ones(1,nu)`. The reasons for this default include:

  - If the plant is open-loop stable, large increments are unnecessary and probably undesirable. For example, when model predictions are imperfect, as is always the case in practice, more conservative increments usually provide more robust controller performance, but poorer reference tracking.
  - These values force the QP Hessian matrix to be positive-definite, such that the QP has a unique solution if no constraints are active.

  To encourage the controller to use even smaller increments for the *i*th MV, increase the `Weights.MVRate(:,i)` value.
- If the plant is open-loop unstable, you might need to decrease the average `Weight.MVRate` value to allow sufficiently rapid response to upsets.

Tips for Setting ECR Weights

See “Constraint Softening” on page 1-12 for tips regarding the `Weights.ECR` property.

Testing and Refinement

To focus on tuning individual cost function weights, perform closed-loop simulation tests under the following conditions:

- No constraints.
- No prediction error. The controller prediction model should be identical to the plant model. Both the **MPC Designer** app and the `sim` function provide the option to simulate under these conditions.

Use changes in the reference and measured disturbance signals (if any) to force a dynamic response. Based on the results of each test, consider changing the magnitudes of selected weights.

One suggested approach is to use constant `Weights.OV(:,i) = 1` to signify “average OV tracking priority,” and adjust all other weights to be relative to this value. Use the `sensitivity` command for guidance. Use the `review` command to check for typical tuning issues, such as lack of closed-loop stability.

See “Adjust Disturbance and Noise Models” on page 2-15 for tests focusing on the disturbance rejection ability of the controller.

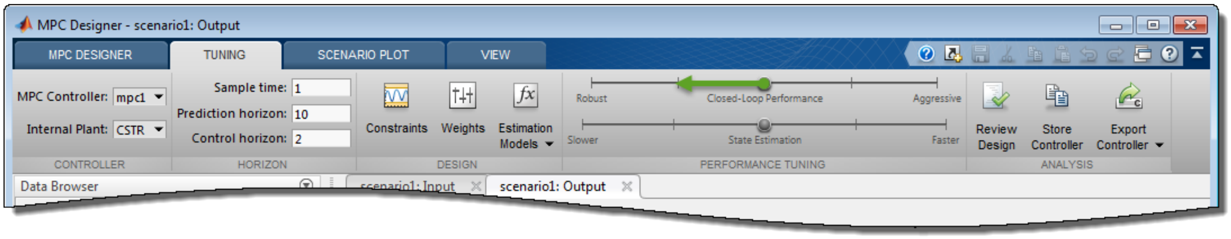
Robustness

Once you have weights that work well under the above conditions, check for sensitivity to prediction error. There are several ways to do so:

- If you have a nonlinear plant model of your system, such as a Simulink® model, simulate the closed-loop performance at operating points other than that for which the LTI prediction model applies.
- Alternatively, run closed-loop simulations in which the LTI model representing the plant differs (such as in structure or parameter values) from that used at the MPC prediction model. Both the **MPC Designer** app and the `sim` function provide the option to simulate under these conditions. See “Test Controller Robustness” for an example.

If controller performance seems to degrade significantly in comparison to tests with no prediction error, for an open-loop stable plant, consider making the controller less aggressive.

In **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 2-15).

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

- Adaptive MPC control on page 5-2
- Gain-scheduled MPC control on page 7-2

See Also

More About

- “Optimization Problem” on page 2-2
- “Specify Constraints” on page 1-10
- “Adjust Disturbance and Noise Models” on page 2-15
- “Tuning Controller Weights”
- “Setting Targets for Manipulated Variables” on page 4-75

Model Predictive Control Problem Setup

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:

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

- $y_j(k+i|k)$ — 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:

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

Here,

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

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

- $u_{j,target}(k+i|k)$ — 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:

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

Here,

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

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

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

The values n_u, p, s_j^u, and $w_{i,j}^{\Delta u}$ are 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 2-6) The corresponding performance measure is:

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

Here,

- z_k — QP decision, given by:

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

- 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:

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

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:

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

Also,

- S_y — Diagonal matrix of plant output variable scale factors, in engineering units.
- S_u — Diagonal matrix of MV scale factors in engineering units.
- $r(k+1|k)$ — n_y plant output reference values at the 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:

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

- $u_{target}(k+i|k)$ — 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 $w_{i,j}^{y}$, $w_{i,j}^{u}$, and $w_{i,j}^{\Delta u}$ that are constant with respect to the index, $i = 1{:}p$.
- The matrices Q, R_u, and $R_{\Delta u}$ are diagonal with the squares of those weights as the diagonal elements.

Constraints

Certain constraints are implicit. For example, a control horizon $m < p$ (or MV blocking) forces some MV increments to be zero, and the state observer used for plant output prediction is a set of implicit equality constraints. Explicit constraints that you can configure are described below.

Bounds on Plant Outputs, MVs, and MV Increments

The most common MPC constraints are bounds, as follows.

$$\frac{y_{j,min}(i)}{s_j^y} - \varepsilon_k V_{j,min}^y(i) \le \frac{y_j(k+i|k)}{s_j^y} \le \frac{y_{j,max}(i)}{s_j^y} + \varepsilon_k V_{j,max}^y(i), \quad i = 1:p, \quad j = 1:n_y$$

$$\frac{u_{j,min}(i)}{s_j^u} - \varepsilon_k V_{j,min}^u(i) \le \frac{u_j(k+i-1|k)}{s_j^u} \le \frac{u_{j,max}(i)}{s_j^u} + \varepsilon_k V_{j,max}^u(i), \ i = 1:p, \quad j = 1:n_u$$

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

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 1-12). Also,

- ε_k — Scalar QP slack variable (dimensionless) used for constraint softening.
- s_j^y — Scale factor for *j*th plant output, in engineering units.
- s_j^u — Scale factor for *j*th MV, in engineering units.
- $y_{j,\min}(i)$, $y_{j,\max}(i)$ — lower and upper bounds for *j*th plant output at *i*th prediction horizon step, in engineering units.
- $u_{j,\min}(i)$, $u_{j,\max}(i)$ — lower and upper bounds for *j*th MV at *i*th prediction horizon step, in engineering units.
- $\Delta u_{j,\min}(i)$, $\Delta u_{j,\max}(i)$ — lower and upper bounds for *j*th MV increment at *i*th prediction horizon step, in engineering units.

Except for the slack variable non-negativity condition, all of the above constraints are optional and are inactive by default (i.e., initialized with infinite limiting values). To include a bound constraint, you must specify a finite limit when you design the controller.

QP Matrices

This section describes the matrices associated with the model predictive control optimization problem described in “Optimization Problem” on page 2-2.

Prediction

Assume that the disturbance models described in “Input Disturbance Model” 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

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

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

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

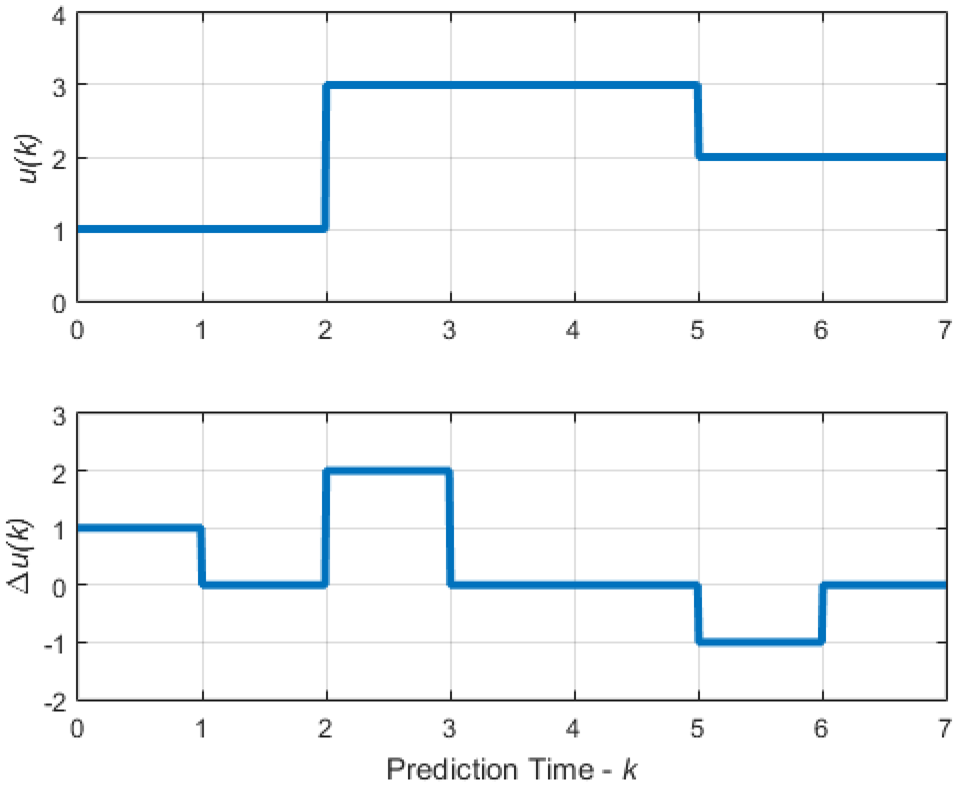
Optimization Variables

Let m be the number of free control moves, and let z= [z_0; ...; 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 variables, 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

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

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

$$W_u = \operatorname{diag}\left(w_{0,1}^u, w_{0,2}^u, \ldots, w_{0,n_u}^u, \ldots, w_{p-1,1}^u, w_{p-1,2}^u, \ldots, w_{p-1,n_u}^u\right)$$
$$W_{\Delta u} = \operatorname{diag}\left(w_{0,1}^{\Delta u}, w_{0,2}^{\Delta u}, \ldots, w_{0,n_u}^{\Delta u}, \ldots, w_{p-1,1}^{\Delta u}, w_{p-1,2}^{\Delta u}, \ldots, w_{p-1,n_u}^{\Delta u}\right)$$
$$W_y = \operatorname{diag}\left(w_{1,1}^y, w_{1,2}^y, \ldots, w_{1,n_y}^y, \ldots, w_{p,1}^y, w_{p,2}^y, \ldots, w_{p,n_y}^y\right)$$

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

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

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 "Weights").

Alternative Cost Function

If you are using the alternative cost function shown in "Alternative Cost Function" on page 2-5, "Equation 2-2" on page 2-11, then "Equation 2-1" on page 2-11 is replaced by the following:

$$W_u = \text{blkdiag}(R_u, \ldots, R_u)$$
$$W_{\Delta u} = \text{blkdiag}(R_{\Delta u}, \ldots, R_{\Delta u})$$
$$W_y = \text{blkdiag}(Q, \ldots, Q)$$

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 "Weights".

Constraints

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

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

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

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

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 2-15
- “Time-Varying Weights and Constraints” on page 2-25
- “Terminal Weights and Constraints” on page 2-29

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

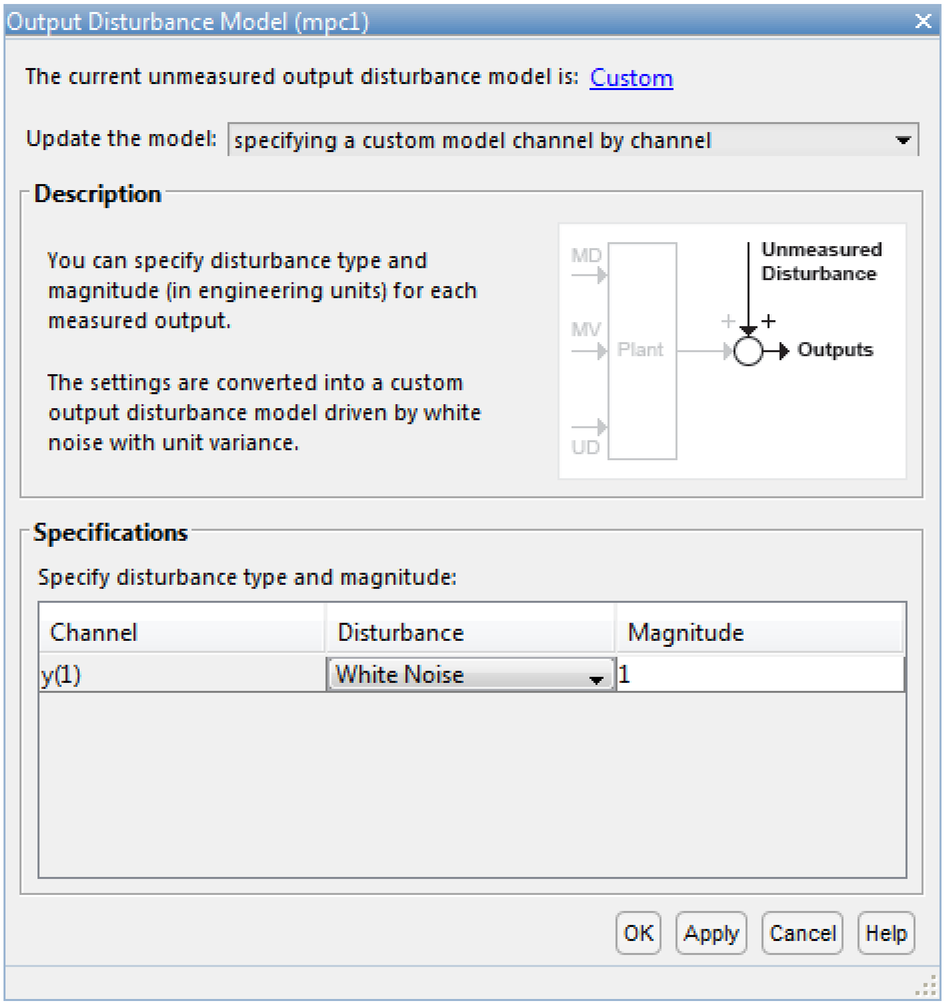
MPC attempts to predict how known and unknown events affect the plant output variables (OVs). Known events are changes in the measured plant input variables (MV and MD inputs). The plant model of the controller predicts the impact of these events, and such predictions can be quite accurate. For more information, see “MPC Modeling”.

The impacts of unknown events appear as errors in the predictions of known events. These errors are, by definition, impossible to predict accurately. However, an ability to anticipate trends can improve disturbance rejection. For example, suppose that the control system has been operating at a near-steady condition with all measured OVs near their predicted values. There are no known events, but one or more of these OVs suddenly deviates from its prediction. The controller disturbance and measurement 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 Modeling”, 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**.


Output Disturbance Model (mpc1)

The current unmeasured output disturbance model is: Custom

Update the model: specifying a custom model channel by channel

Description

You can specify disturbance type and magnitude (in engineering units) for each measured output.

The settings are converted into a custom output disturbance model driven by white noise with unit variance.

Specifications

Specify disturbance type and magnitude:

Channel	Disturbance	Magnitude
y(1)	White Noise	1

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

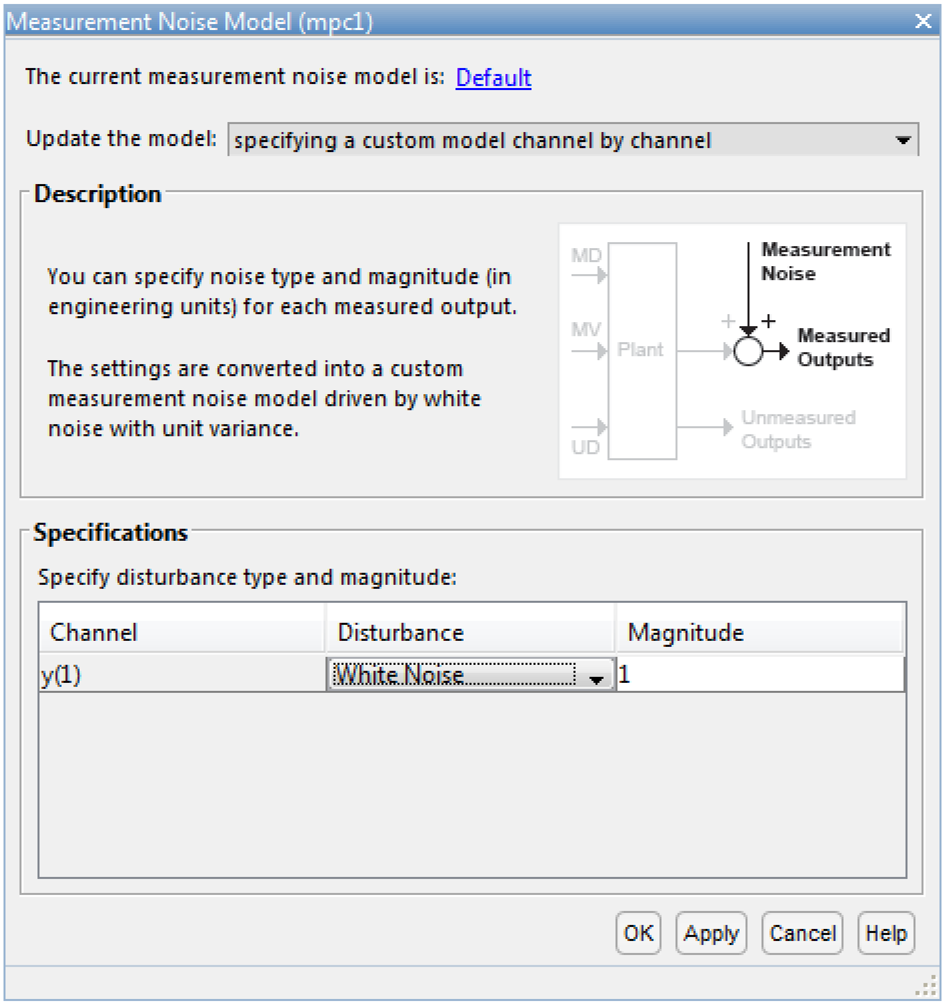
You can also specify the white noise input **Magnitude** for each disturbance model, overriding the assumption of unit variance. As you increase the noise magnitude, the controller responds more aggressively to a given prediction error. The specified noise magnitude corresponds to the static gain in 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.


Measurement Noise Model (mpc1)

The current measurement noise model is: Default

Update the model: specifying a custom model channel by channel

Description

You can specify noise type and magnitude (in engineering units) for each measured output.

The settings are converted into a custom measurement noise model driven by white noise with unit variance.

Specifications

Specify disturbance type and magnitude:

Channel	Disturbance	Magnitude
y(1)	White Noise	1

`White Noise` is the default option and, in nearly all applications, should provide adequate performance.

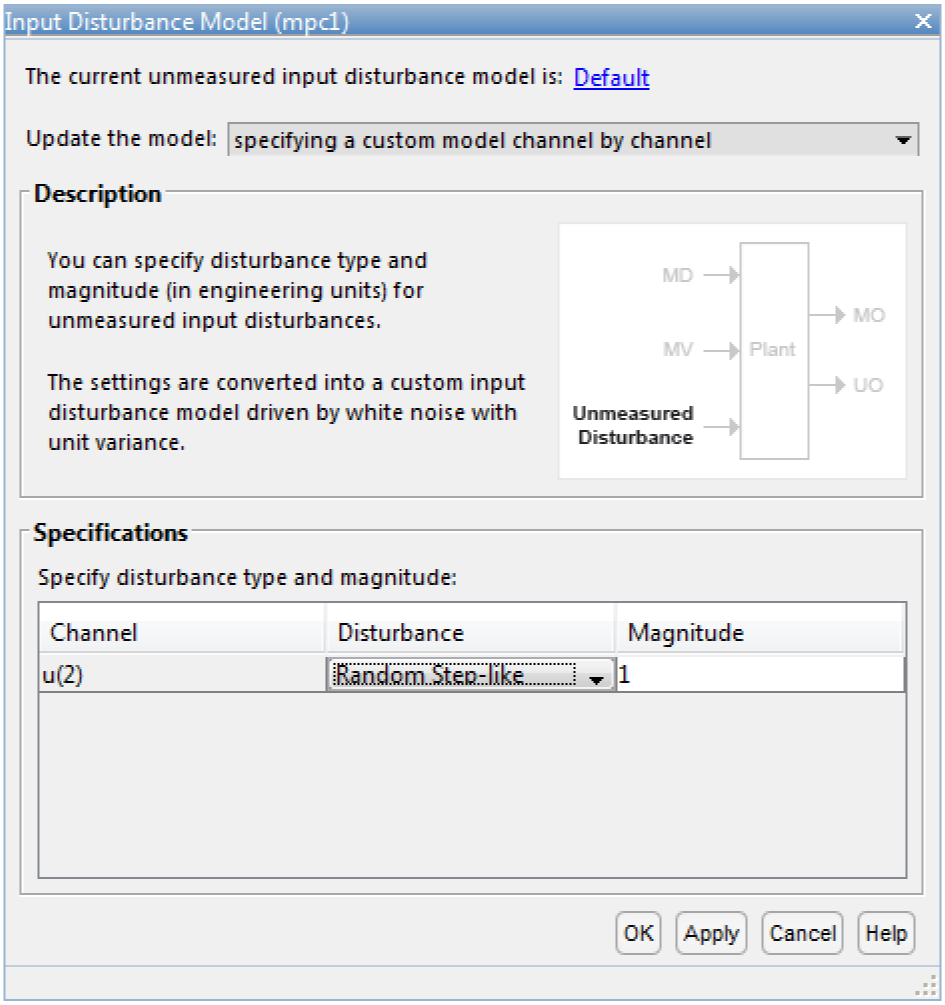
When you include a measurement noise model, the controller considers each prediction error to be a combination of disturbance and noise effects. Qualitatively, as you increase the specified noise **Magnitude**, the controller attributes a larger fraction of each prediction error to noise, and it responds less aggressively. Ultimately, the controller stops responding to prediction errors and only changes its MVs when you change the OV or MV reference signals.

Input Disturbance Model

When your plant model includes unmeasured disturbance (UD) inputs, the controller can use an input disturbance model in addition to the standard output disturbance model. The former provides more flexibility and is generated automatically by default. If the chosen input disturbance model does not appear to allow complete elimination of sustained disturbances, an output disturbance model is also added by default.

As shown in “MPC Modeling”, the input disturbance model consists of one or more white noise signals, with unit variance and zero mean, entering a dynamic system. The outputs of this system are the UD inputs to the plant model. In contrast to the output disturbance model, input disturbances affect the plant outputs in a more complex way as they pass through the plant model dynamics.

As with the output disturbance model, you can use **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**.


Input Disturbance Model (mpc1)

The current unmeasured input disturbance model is: Default

Update the model: specifying a custom model channel by channel

Description

You can specify disturbance type and magnitude (in engineering units) for unmeasured input disturbances.

The settings are converted into a custom input disturbance model driven by white noise with unit variance.

Specifications

Specify disturbance type and magnitude:

Channel	Disturbance	Magnitude
u(2)	Random Step-like	1

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 2-44, 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.

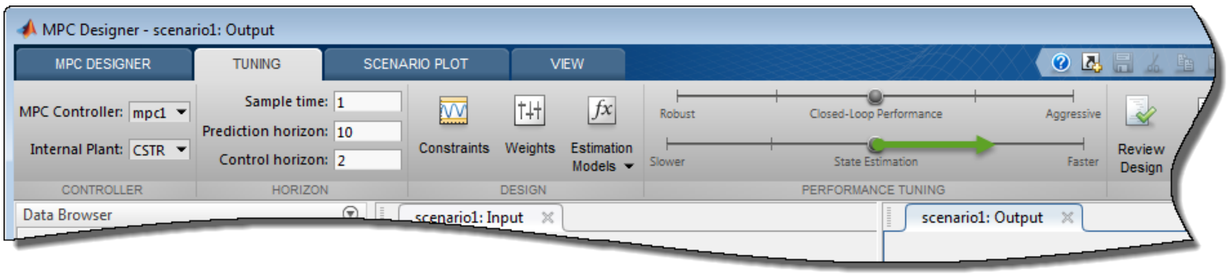
By default, the Model Predictive Control Toolbox software creates detectable models. If you modify the default assumptions (or change n_{ym}) and encounter a detectability error, you can revert to the default case.

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 1-2). In the context of disturbance and noise modeling, this makes the default assumption of unit-variance white noise inputs more likely to yield good performance.
2. Initially, keep the disturbance models in their default configuration.
3. After tuning the cost function weights (see “Tune Weights” on page 1-16), test your controller response to an unmeasured disturbance input other than a step disturbance at the plant output. Specifically, if your plant model includes UD inputs, simulate a disturbance using one or more of these. Otherwise, simulate one or more load disturbances, that is, a step disturbance added to a designated MV. Both **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.


MPC Designer - scenario1: Output

MPC DESIGNER | TUNING | SCENARIO PLOT | VIEW

MPC Controller: mpc1
Internal Plant: CSTR
Sample time: 1
Prediction horizon: 10
Control horizon: 2

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
`getindist` | `getoutdist` | `setindist` | `setmpcsignals` | `setoutdist`

More About

- “MPC Modeling”
- “Controller State Estimation” on page 2-44
- “Design Controller Using MPC Designer”

Custom State Estimation

The Model Predictive Control Toolbox software allows the following alternatives to the default state estimation approach:

- You can override the default Kalman gains, *L* and *M*. Obtain the default values using `getEstimator`. Then, use `setEstimator` to override those values. These commands assume that the columns of *L* and *M* are in the engineering units for the measured plant outputs. Internally, the software converts them to dimensionless form.
- You can use the custom estimation option. This skips all Kalman gain calculations. When the controller operates, at each control interval you must use an external procedure to estimate the controller states, `xc(k|k)`, providing this to the controller.

Note You cannot use custom state estimation with **MPC Designer**.

See Also

`getEstimator` | `setEstimator`

More About

- "Controller State Estimation" on page 2-44
- Using Custom State Estimation

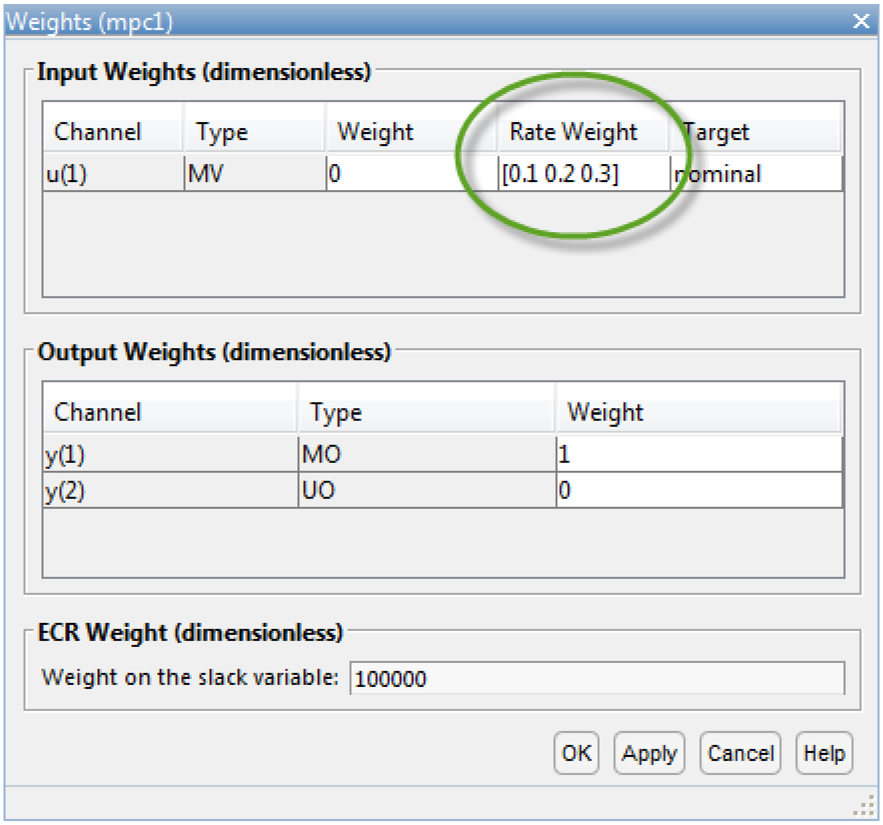
Time-Varying Weights and Constraints

Time-Varying Weights

As explained in “Optimization Problem” on page 2-2, the w^y, w^u, and $w^{\Delta u}$ weights can change from one step in the prediction horizon to the next. Such a time-varying weight is an array containing *p* rows, where *p* is the prediction horizon, and either n_y or n_u columns (number of OVs or MVs).

Using time-varying weights provides additional tuning possibilities. However, it complicates tuning. Recommended practice is to use constant weights unless your application includes unusual characteristics. For example, an application requiring terminal weights must employ time-varying weights. See “Terminal Weights and Constraints” on page 2-29.

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.


Weights (mpc1)

Input Weights (dimensionless)

Channel	Type	Weight	Rate Weight	Target
u(1)	MV	0	[0.1 0.2 0.3]	nominal

Output Weights (dimensionless)

Channel	Type	Weight
y(1)	MO	1
y(2)	UO	0

ECR Weight (dimensionless)

Weight on the slack variable: 100000

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.

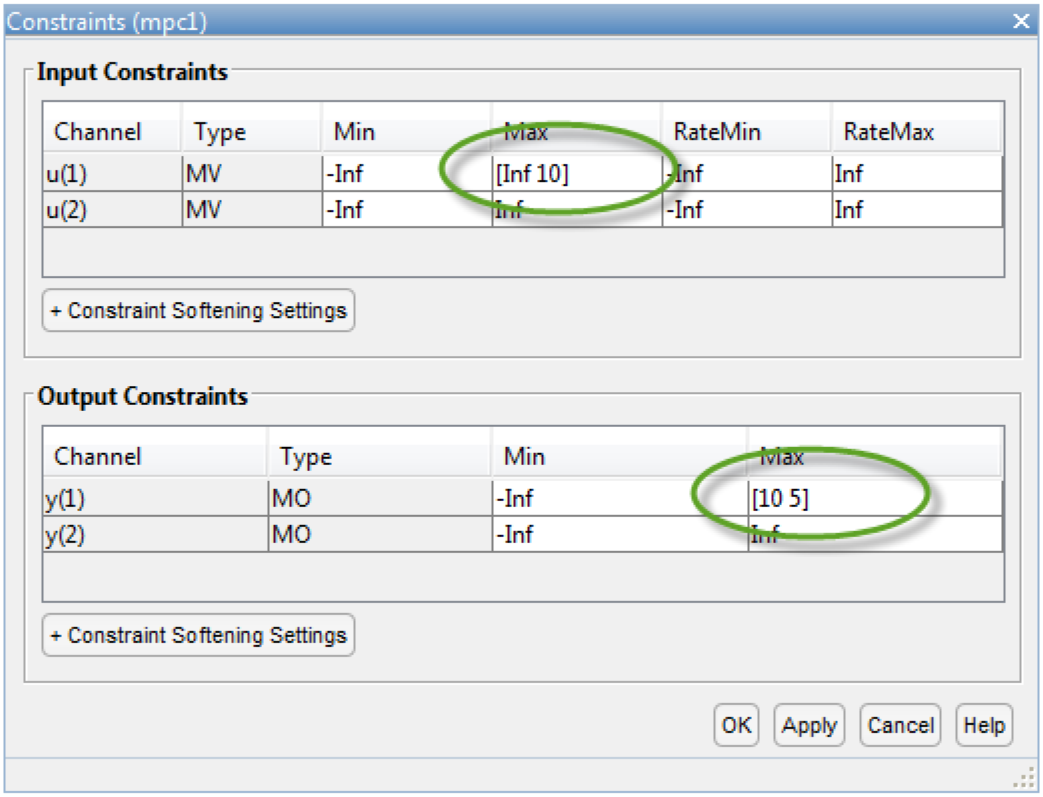
Time-Varying Constraints

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.


Constraints (mpc1)

Input Constraints

Channel	Type	Min	Max	RateMin	RateMax
u(1)	MV	-Inf	[Inf 10]	-Inf	Inf
u(2)	MV	-Inf	Inf	-Inf	Inf

+ Constraint Softening Settings

Output Constraints

Channel	Type	Min	Max
y(1)	MO	-Inf	[10 5]
y(2)	MO	-Inf	Inf

+ Constraint Softening Settings
OK
Apply
Cancel
Help

See Also

More About

- “Optimization Problem” on page 2-2
- “Terminal Weights and Constraints” on page 2-29
- “Vary Input and Output Bounds at Run Time”

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:

$$J(u) = \sum_{i=1}^{\infty} x(k+i)^T Qx(k+i) + u(k+i-1)^T Ru(k+i-1)$$

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

$$x(k+1) = Ax + Bu(k)$$

The LQR provides nominal stability provided matrices Q and R meet certain conditions. You can convert the LQR to a finite-horizon form as follows:

$$J(u) = \sum_{i=1}^{p-1} [x(k+i)^T Qx(k+i) + u(k+i-1)^T Ru(k+i-1)] + x(k+p)^T Q_p x(k+p)$$

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

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

You can obtain this solution using the `lqr` command in Control System Toolbox™ software.

In general, Q_p is a full (symmetric) matrix. You cannot use the "Standard Cost Function" on page 2-2 to implement the LQR cost function. The only exception is for the first $p - 1$ steps if Q and R are diagonal matrices. Also, you cannot use the alternative cost function on page 2-5 because it employs identical weights at each step in the horizon. Thus, by definition, the terminal weight differs from those in steps 1 to $p - 1$. Instead, use the following steps:

1. Augment the model ("Equation 2-6" on page 2-29) 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

- “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”
- “Provide LQR Performance Using Terminal Penalty Weights” on page 4-69

Constraints on Linear Combinations of Inputs and Outputs

You can constrain linear combinations of plant input and output variables. For example, you can constrain a particular manipulated variable (MV) to be greater than a linear combination of two other MVs. The general form of such constraints is the following:

$$Eu\left(k+i|k\right)+Fy\left(k+i|k\right)+Sv(k+i\mid k)\leq G+\varepsilon_k V.$$

- ε_k — QP slack variable used for constraint softening (See “Constraint Softening” on page 1-12)
- $u(k+i|k)$ — n_u MV values, in engineering units
- $y(k+i|k)$ — n_y predicted plant outputs, in engineering units
- $v(k+i|k)$ — n_v measured plant disturbance inputs, in engineering units
- E, F, S, G, and V are constants

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

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”.

Note Using mixed input/output constraints is not supported in **MPC Designer**.

See Also

`getconstraint` | `setconstraint`

More About

- “Optimization Problem” on page 2-2
- “Update Constraints at Run Time”
- “Using Custom Input and Output Constraints”

- “Nonlinear Blending Process with Custom Constraints”

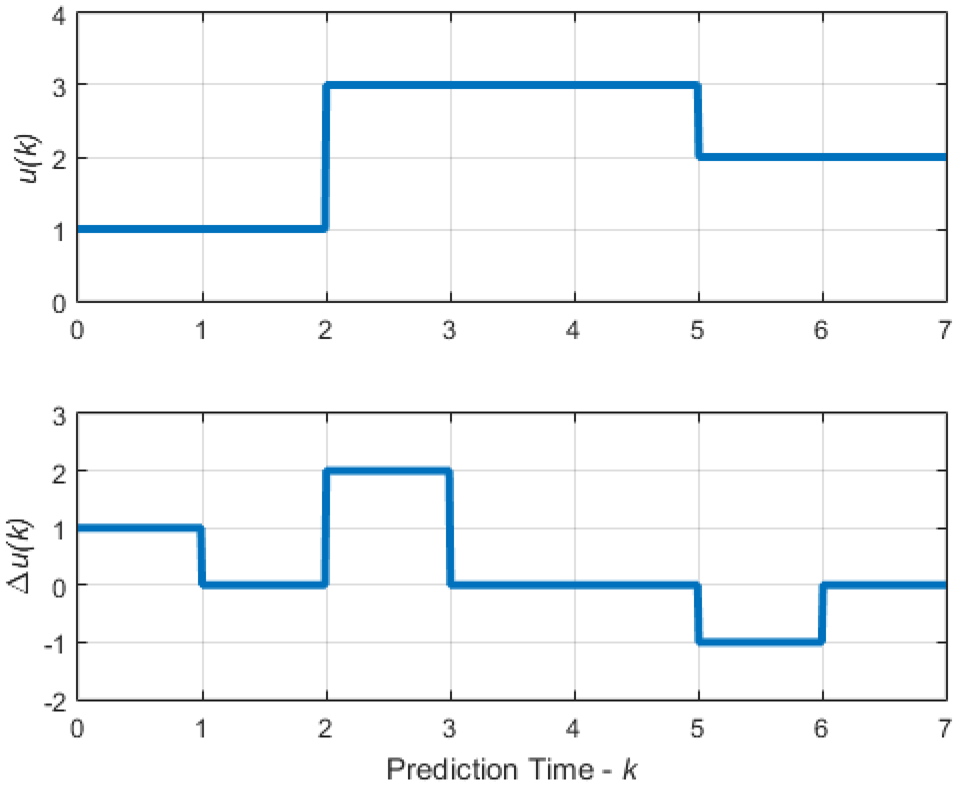
Manipulated Variable Blocking

Manipulated variable blocking is an alternative to the simpler control horizon concept (see “Choose Sample Time and Horizons” on page 1-6). It has many of the same benefits. It also provides more tuning flexibility and potential to smooth MV adjustments. To use manipulated variable blocking, you divide the prediction horizon into a series of blocks. The controller then holds the manipulated variable constant within each block.

A recommended approach to blocking is as follows:

- Divide the prediction horizon into 3-5 blocks.
- Try the following alternatives
  - Equal block sizes (one-fifth to one-third of the prediction horizon, *p*)
  - Block sizes increasing. Example, with *p* = 20, three blocks of duration 3, 7 and 10 intervals.

To use manipulated variable blocking, specify your control horizon as a vector of block sizes. For example, the following figure represent control moves for a control horizon of `p = [2 3 2]`:



For each block, the manipulated variable, *u*, is constant, that is:

- $u(0) = u(1)$
- $u(2) = u(3) = u(4)$
- $u(5) = u(6)$

Test the resulting controller in the same way that you test cost function weights. See “Tune Weights” on page 1-16.

See Also

`mpc` | `sim`

More About

- "Optimization Problem" on page 2-2
- "Tune Weights" on page 1-16
- "Design MPC Controller for Plant with Delays"
- "Choose Sample Time and Horizons" on page 1-6
- "Design MPC Controller at the Command Line"
- "Design Controller Using MPC Designer"

QP Solver

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

$$\underset{x}{Min}\left(\frac{1}{2}x'Hx + f'x\right)$$

subject to the linear inequality constraints

$$Ax \geq b$$

where

- *x* is the solution vector.
- *H* is the Hessian matrix. This matrix is constant when using implicit MPC without online weight changes.
- *A* is a matrix of linear constraint coefficients. This matrix is constant when using implicit MPC.
- *b* and *f* are vectors.

At the beginning of each control interval, the controller computes *H*, *f*, *A*, and *b* or, if they are constant, retrieves their precomputed values.

The toolbox uses the KWIK algorithm [1] to solve the QP problem, which requires the Hessian to be positive definite. In the first control step, KWIK uses a cold start, in which the initial guess is the unconstrained solution described in “Unconstrained Model Predictive Control” on page 2-13. If *x* satisfies the constraints, it is the optimal QP solution, x^*, and the algorithm terminates. Otherwise, at least one of the linear inequality constraints must be satisfied as an equality. In this case, KWIK uses an efficient, numerically robust strategy to determine the active constraint set satisfying the standard optimization conditions. In the following control steps, KWIK uses a warm start. In this case, the active constraint set determined at the previous control step becomes the initial guess for the next.

Although KWIK is robust, consider the following:

- One or more linear constraints can be violated slightly due to numerical round-off errors. The toolbox employs a nonadjustable relative tolerance. This tolerance allows constraint violations of 10^{-6} times the magnitude of each term. Such violations are considered normal and do not generate warning messages.

- The toolbox also uses a nonadjustable tolerance when testing for an optimal solution.
- The search for the active constraint set is an iterative process. If the iterations reach a problem-dependent maximum, the algorithm terminates. For some controller configurations, the default maximum iterations can be very large, which can make the QP solver appear to stop responding (see “Optimizer”).
- If your problem includes hard constraints, these constraints can be infeasible (impossible to satisfy). If the algorithm detects infeasibility, it terminates immediately.

In the last two situations, with an abnormal outcome to the search, the controller retains the last successful control output. For more information, see, the `mpcmove` command. You can detect an abnormal outcome and override the default behavior as you see fit.

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 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 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 be no greater than `10`.

```
MPCobj.Optimizer.MaxIter = 10;
```

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

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, when applied, it satisfies all your specified constraints.

There is no guarantee that the suboptimal solution performs better than holding 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 4-126.

Custom QP Application

To access the built-in KWIK solver for applications that require solving online QP problems, use the `mpcqpsolver` command. This option is useful for:

- Advanced MPC applications that are beyond the scope of Model Predictive Control Toolbox software.
- Custom QP applications, including applications that require code generation.

Custom QP Solver

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 `qpkwik` solver at each control interval. This option is useful for:

- Validating your simulation results or generating code with a third-party solver.
- Large MPC problems where the built-in KWIK solver runs slowly or fails to find a feasible solution.

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.

	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`	

Implement Custom Solver for Simulation

To simulate an MPC controller with a custom QP solver:

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 4-115.

Implement 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. Doing so:

- At the command line requires MATLAB Coder™ software.
- In Simulink requires Simulink Coder or Simulink PLC Coder™ software.

To generate code for MPC controllers that use a custom QP solver:

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 9-2.

For an example, see “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 9-32.

Implement Custom Solver for Simulation and Code Generation

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

- 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” (MATLAB).

Custom Solver Argument Descriptions

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

- When using a custom solver for simulation:

  ```
  function [x,status] = mpcCustomSolver(H,f,A,b,x0)
  ```
- When using a custom solver for code generation:

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

In both cases, 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 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 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 according to the following:

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`, may 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 constraint $Ax \geq b$ cannot be satisfied.
`-2`	An unrecoverable numerical error occurred.

References

[1] Schmid, C. and L.T. Biegler, “Quadratic programming methods for reduced Hessian SQP,” *Computers & Chemical Engineering*, Vol. 18, Number 9, 1994, pp. 817–832.

See Also

`mpc` | `mpcmove` | `mpcqpsolver`

More About

- “Optimization Problem” on page 2-2
- “Simulate MPC Controller with a Custom QP Solver” on page 4-115

Controller State Estimation

Controller State Variables

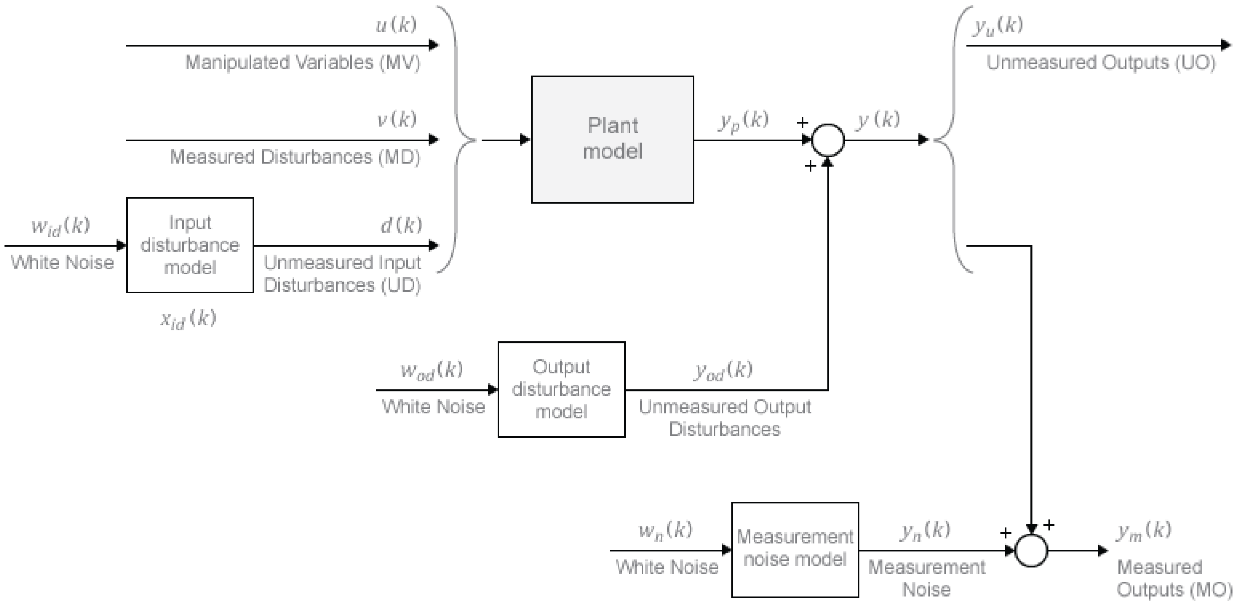
As the controller operates, it uses its current state, x_c, as the basis for predictions. By definition, the state vector is the following:

$$x_c^T(k) = \begin{bmatrix} x_p^T(k) & x_{id}^T(k) & x_{od}^T(k) & x_n^T(k) \end{bmatrix}.$$

Here,

- x_c is the controller state, comprising $n_{xp} + n_{xid} + n_{xod} + n_{xn}$ state variables.
- x_p is the plant model state vector, of length n_{xp}.
- x_{id} is the input disturbance model state vector, of length n_{xid}.
- x_{od} is the output disturbance model state vector, of length n_{xod}.
- x_n is the measurement noise model state vector, of length n_{xn}.

Thus, the variables comprising x_c represent the models appearing in the following diagram of the MPC system.



Some of the state vectors may be empty. If not, they appear in the sequence defined within each model.

By default, the controller updates its state automatically using the latest plant measurements. See “State Estimation” on page 2-46 for details. Alternatively, the custom state estimation feature allows you to update the controller state using an external procedure, and then supply these values to the controller. See “Custom State Estimation” on page 2-24 for details.

State Observer

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

$$\begin{aligned} x_c(k+1) &= Ax_c(k) + Bu_o(k) \\ y(k) &= Cx_c(k) + Du_o(k). \end{aligned}$$

MPC controller uses the state observer in the following ways:

- To estimate values of unmeasured states needed as the basis for predictions (see “State Estimation” on page 2-46).

- To predict how the controller's proposed manipulated variable (MV) adjustments will affect future plant output values (see "Output Variable Prediction" on page 2-49).

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:

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

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:

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

Here, the plant and output disturbance models are resequenced so that the measured outputs precede the unmeasured outputs.

State Estimation

In general, the controller states are unmeasured and must be estimated. By default, the controller uses a steady-state Kalman filter that derives from the state observer. (See "State Observer" on page 2-45.)

At the beginning of the kth control interval, the controller state is estimated with the following steps:

1 Obtain the following data:

   - $x_c(k|k\text{–}1)$ — Controller state estimate from previous control interval, k–1
   - $u^{act}(k\text{–}1)$ — Manipulated variable (MV) actually used in the plant from k–1 to k (assumed constant)

- $u^{opt}(k\text{–}1)$ — Optimal MV recommended by MPC and assumed to be used in the plant from k–1 to k
- $v(k)$ — Current measured disturbances
- $y_m(\mathrm{k})$ — 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 $x_c(k|k\text{–}1)$ when $u^{act}(k\text{–}1)$ and $u^{opt}(k\text{–}1)$ 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,

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

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

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

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 2-45).

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 2-46)
- $v(k)$ — 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 nxn 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 1-2. The equations also assume zero offsets. Inclusion of nonzero offsets is straightforward.

For faster computations, the MPC controller uses an alternative form of the above equations in which constant terms are computed and stored during controller initialization. See “QP Matrices” on page 2-7.

See Also

`kalman`

More About

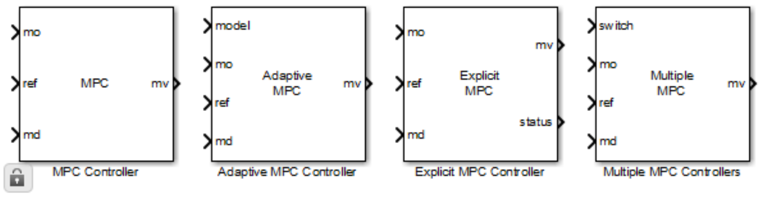
- “MPC Modeling”
- “Optimization Problem” on page 2-2
- “Custom State Estimation” on page 2-24

Model Predictive Control Simulink Library

MPC Library

The MPC Simulink Library provides two blocks you can use to implement MPC control in Simulink, MPC Controller, and Multiple MPC Controllers.

Access the library using the Simulink Library Browser or by typing `mpclib` at the command prompt. The library contains the following blocks:



MPC Simulink Library

For more information on each block, see their respective block reference pages:

- MPC Controller
- Adaptive MPC Controller
- Explicit MPC Controller
- Multiple MPC Controllers

Once you have access to the library, you can add one of its blocks to your Simulink model by clicking-and-dragging or copying-and-pasting.

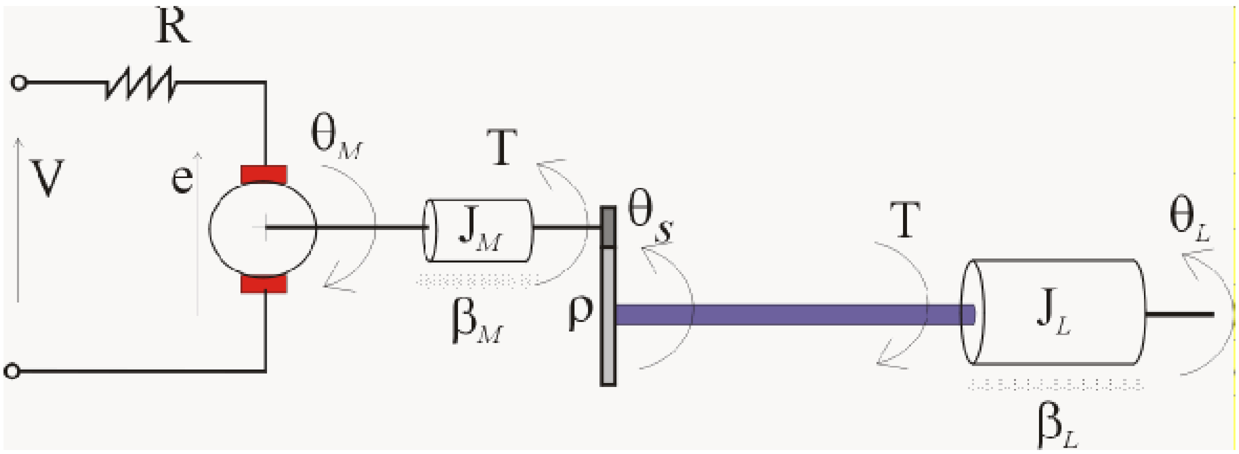
Case-Study Examples

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

$$x_p = \begin{bmatrix}\theta_L & \omega_L & \theta_M & \omega_M\end{bmatrix}^T,$$

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

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

$$\theta_L = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} x_p$$

$$T = \begin{bmatrix} k_T & 0 & -\dfrac{k_T}{\rho} & 0 \end{bmatrix} x_p$$

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

Construct Plant Model

Specify the model constants.

```
Kt = 1280.2;  % Torsional rigidity
Km = 10;      % Motor constant
Jm = 0.5;     % Motor inertia
Jl = 50*Jm;   % Load inertia
N = 20;       % Gear ratio
Bm = 0.1;     % Rotor viscous friction
Bl = 25;      % Load viscous friction
R = 20;       % Armature resistance
```

Define the state-space matrices derived from the model equations.

```
A = [        0       1             0                0;
        -Kt/Jl  -Bl/Jl     Kt/(N*Jl)                0;
             0       0             0                1;
     Kt/(Jm*N)       0  -Kt/(Jm*N^2)  -(Bm+Km^2/R)/Jm];
B = [0; 0; 0; Km/(R*Jm)];
C = [  1  0       0  0;
      Kt  0   -Kt/N  0];
D = [0; 0];
```

Create a state-space model.

```
plant = ss(A,B,C,D);
```

Open MPC Designer App

```
mpcDesigner
```

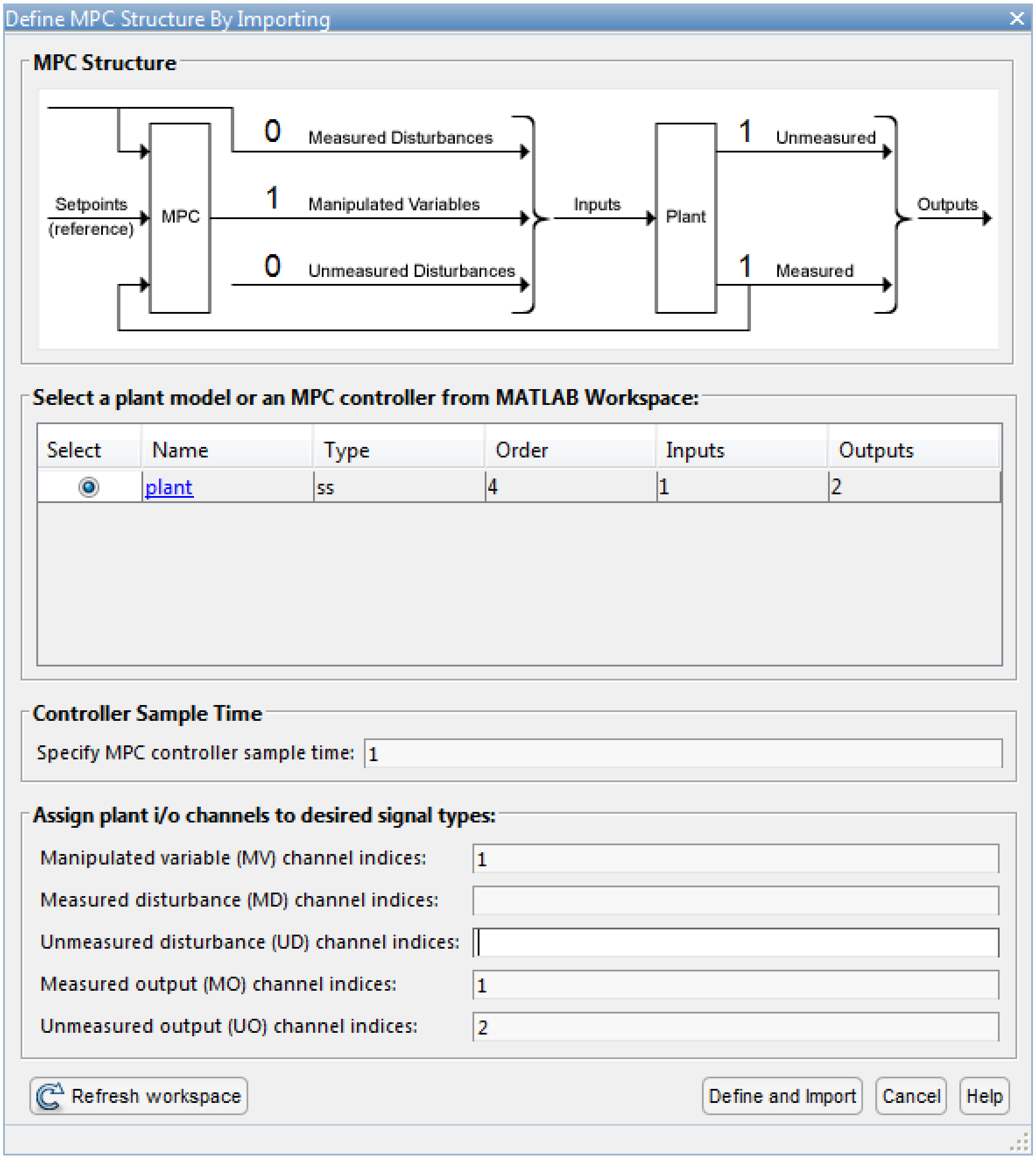
Import Plant and Define Signal Configuration

In **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



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

Select	Name	Type	Order	Inputs	Outputs
●	plant	ss	4	1	2

Controller Sample Time

Specify MPC controller sample time: 1

Assign plant i/o channels to desired signal types:

Manipulated variable (MV) channel indices:	1
Measured disturbance (MD) channel indices:	
Unmeasured disturbance (UD) channel indices:	
Measured output (MO) channel indices:	1
Unmeasured output (UO) channel indices:	2

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.

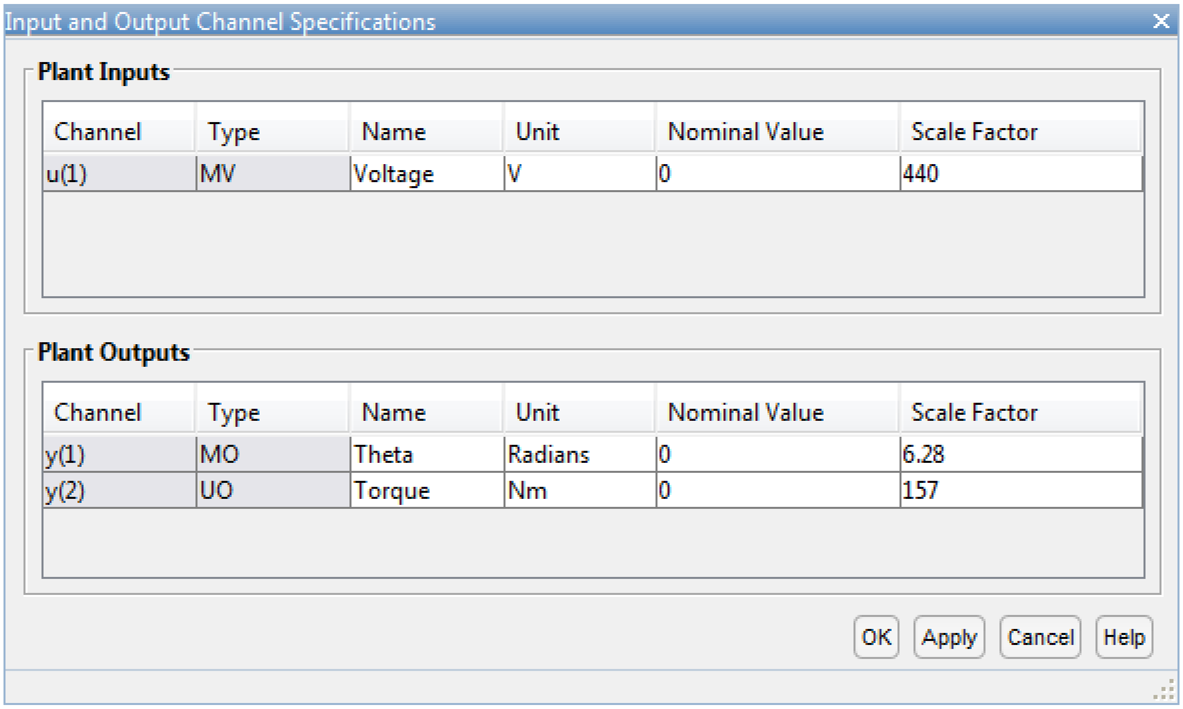
Define Input and Output Channel Attributes

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`



Plant Inputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
u(1)	MV	Voltage	V	0	440

Plant Outputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
y(1)	MO	Theta	Radians	0	6.28
y(2)	UO	Torque	Nm	0	157

Click **OK** to update the channel attributes and close the dialog box.

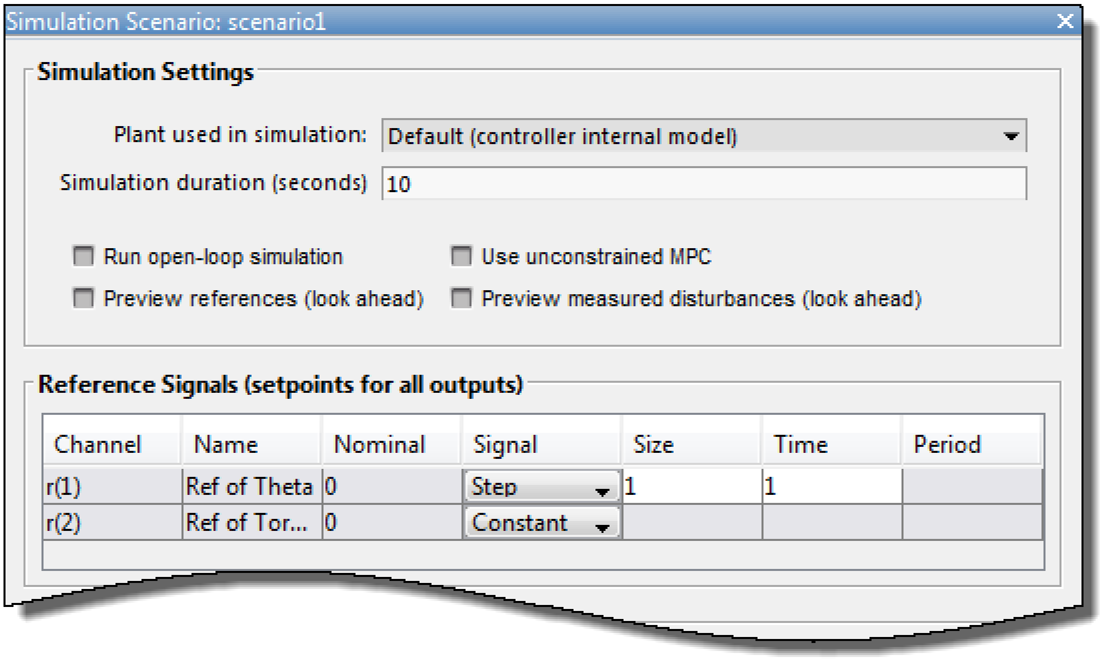
Modify Scenario To Simulate Angular Position Step Response

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.



Simulation Scenario: scenario1

Simulation Settings

Plant used in simulation: Default (controller internal model)

Simulation duration (seconds) 10

☐ Run open-loop simulation ☐ Use unconstrained MPC

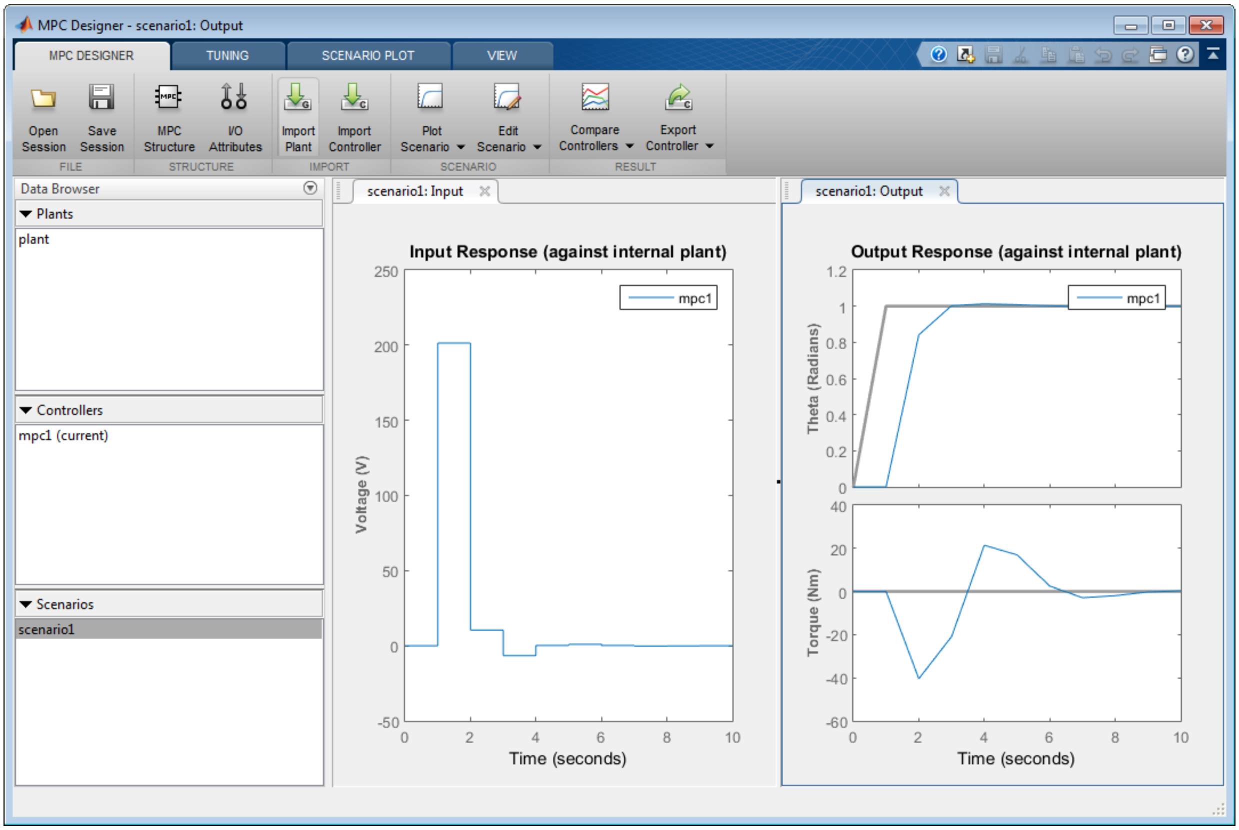
☐ Preview references (look ahead) ☐ Preview measured disturbances (look ahead)

Reference Signals (setpoints for all outputs)

Channel	Name	Nominal	Signal	Size	Time	Period
r(1)	Ref of Theta	0	Step	1	1	
r(2)	Ref of Tor...	0	Constant			

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

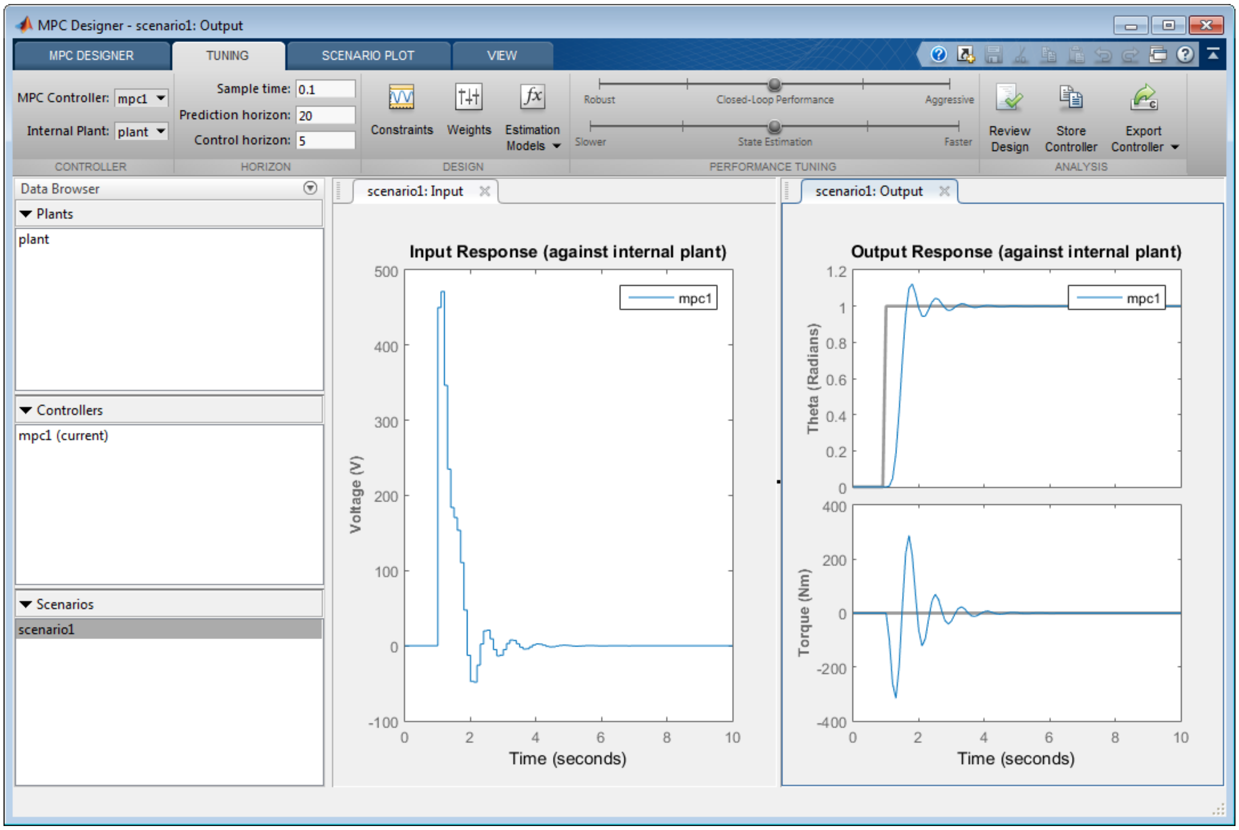
On the **Tuning** tab, in the **Horizon** section, specify a **Sample time** of `0.1` seconds.

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`.



MPC Controller: mpc1
Internal Plant: plant
Sample time: 0.1
Prediction horizon: 20
Control horizon: 5

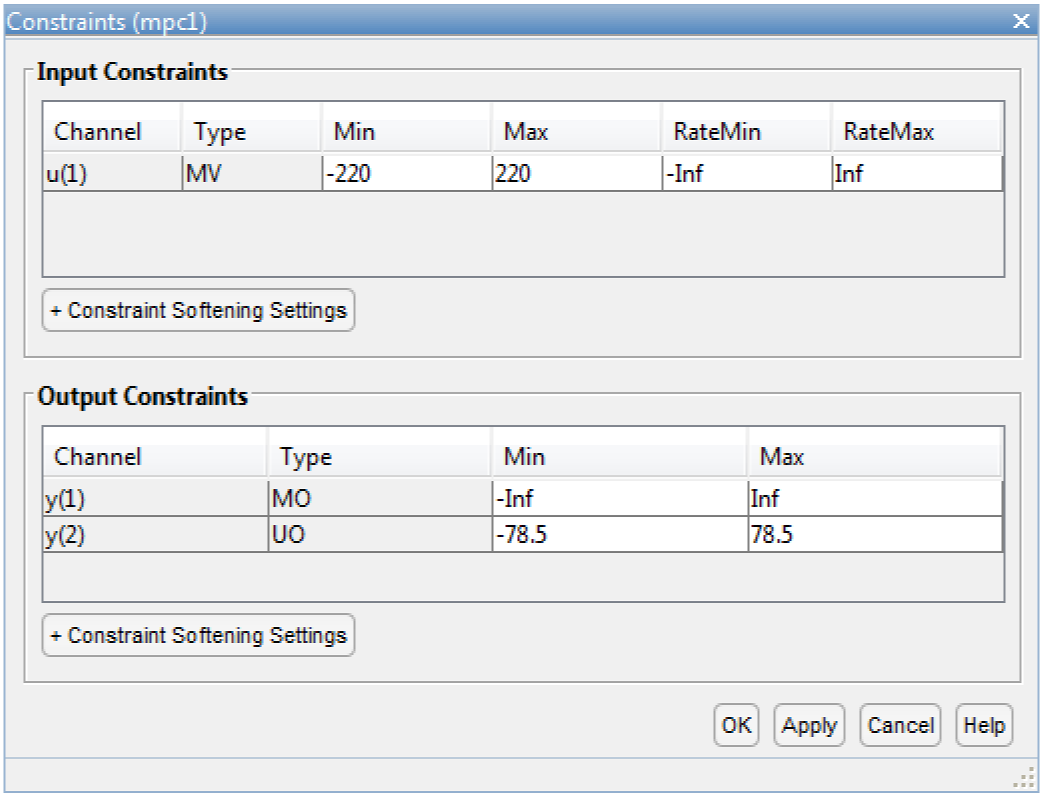
As you update the sample time and horizon values, the **Input Response** and **Output Response** plots update automatically. Both the input voltage and torque values exceed the constraints defined in the system model specifications.

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).


Constraints (mpc1)
+ Constraint Softening Settings
+ Constraint Softening Settings
OK
Apply
Cancel
Help

Input Constraints

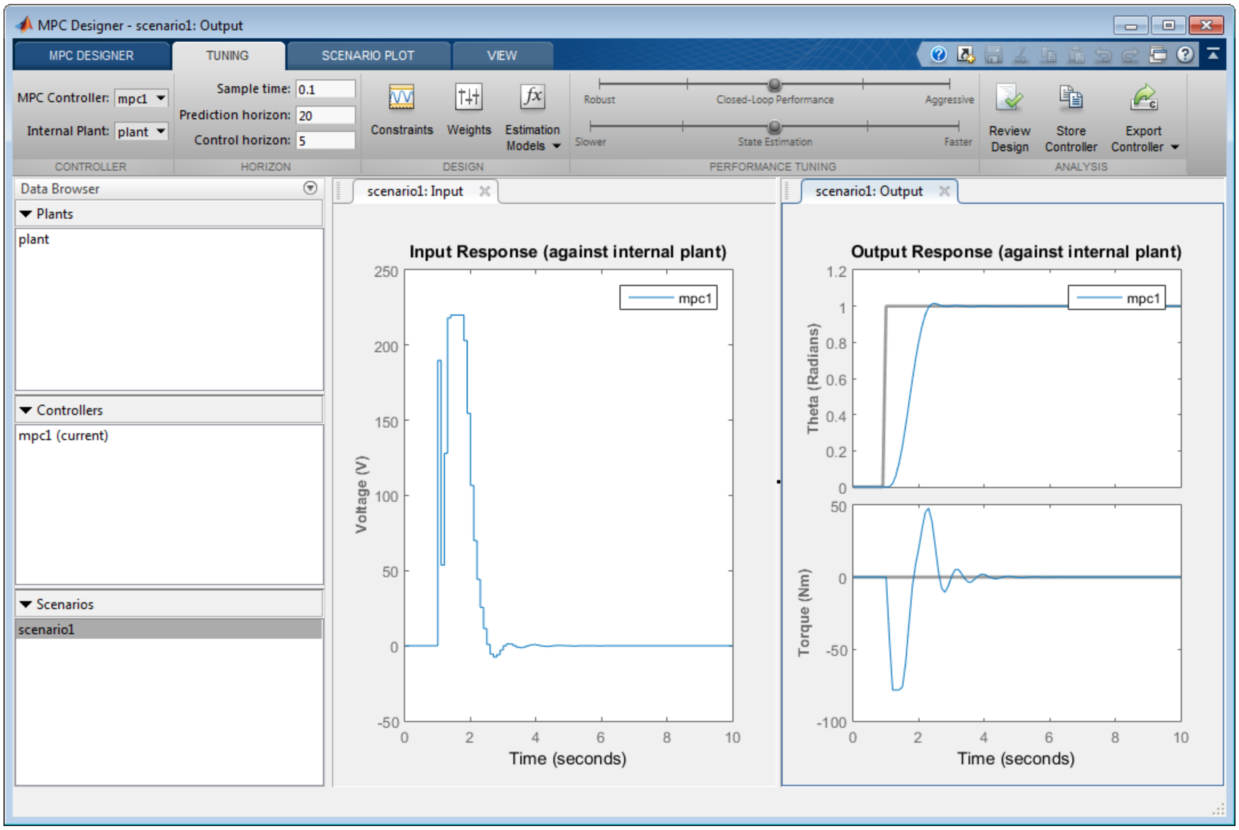
Channel	Type	Min	Max	RateMin	RateMax
u(1)	MV	-220	220	-Inf	Inf

Output Constraints

Channel	Type	Min	Max
y(1)	MO	-Inf	Inf
y(2)	UO	-78.5	78.5

There are no additional constraints, that is the other constraints remain at their default maximum and minimum values, –`Inf` and `Inf` respectively

Click **OK**.

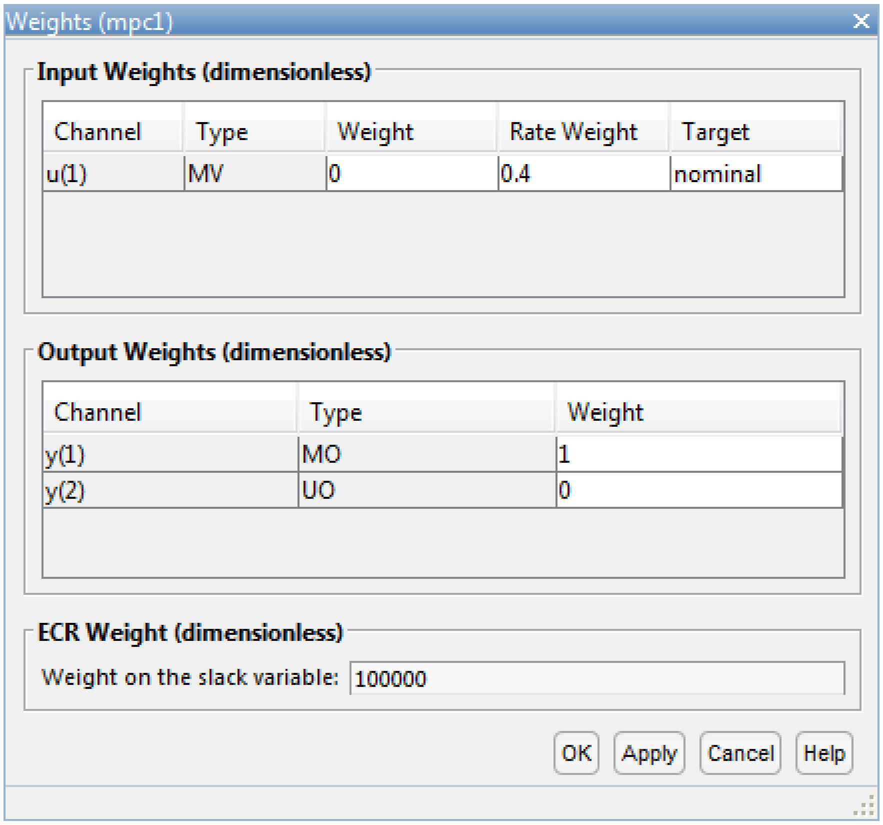


The response plots update to reflect the new constraints. In the **Input Response** plot, there are undesirable large changes in the input voltage.

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**.



Weights (mpc1)

Input Weights (dimensionless)

Channel	Type	Weight	Rate Weight	Target
u(1)	MV	0	0.4	nominal

Output Weights (dimensionless)

Channel	Type	Weight
y(1)	MO	1
y(2)	UO	0

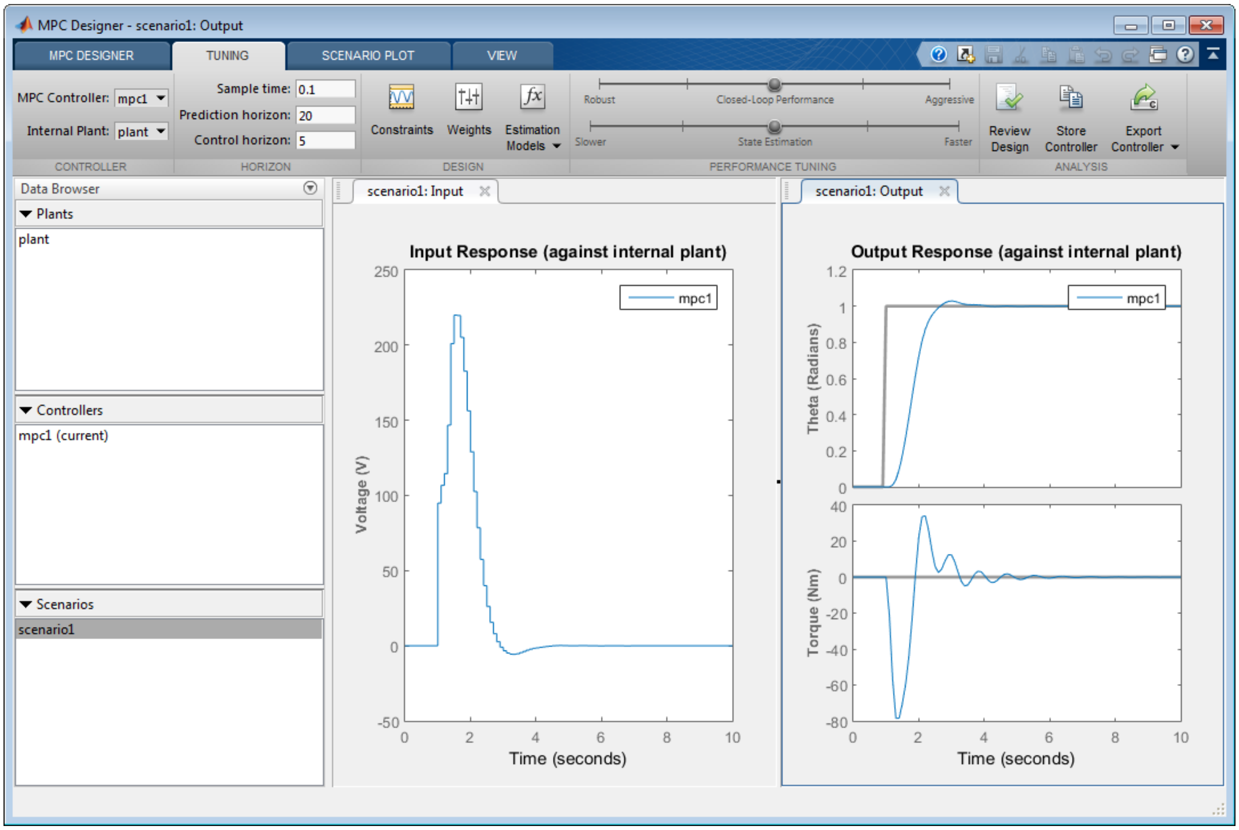
ECR Weight (dimensionless)

Weight on the slack variable: 100000

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.

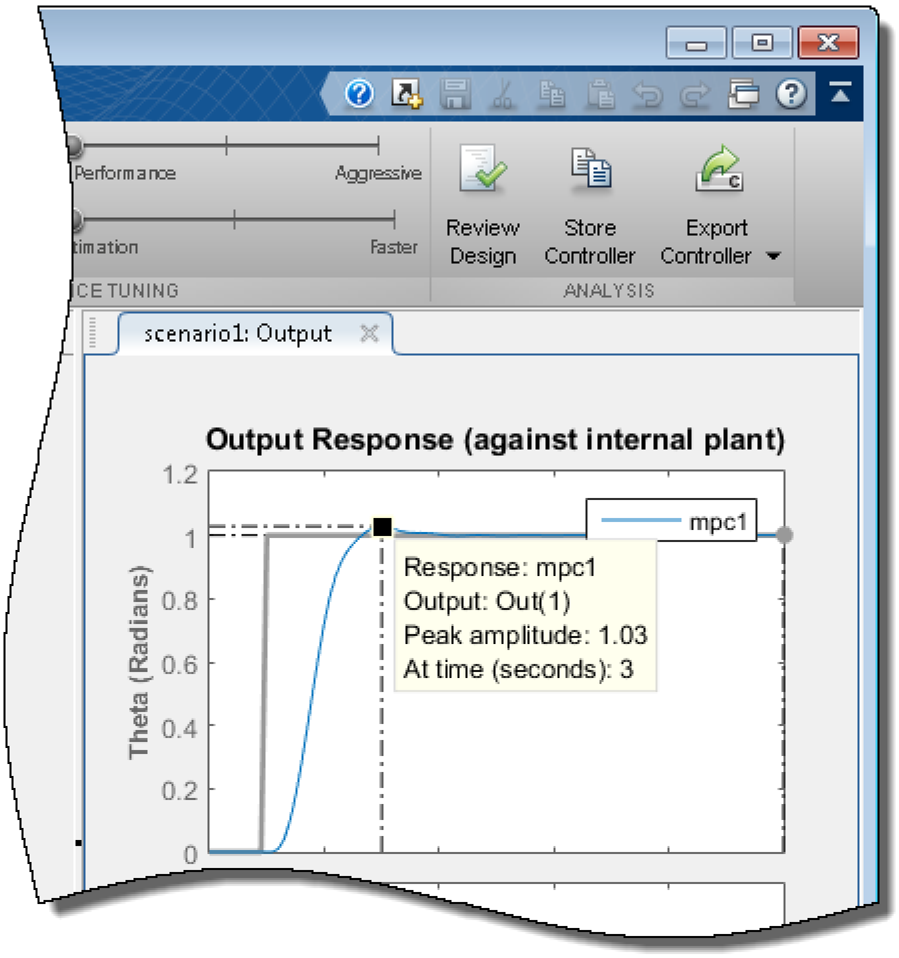
Click **OK**.



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**.

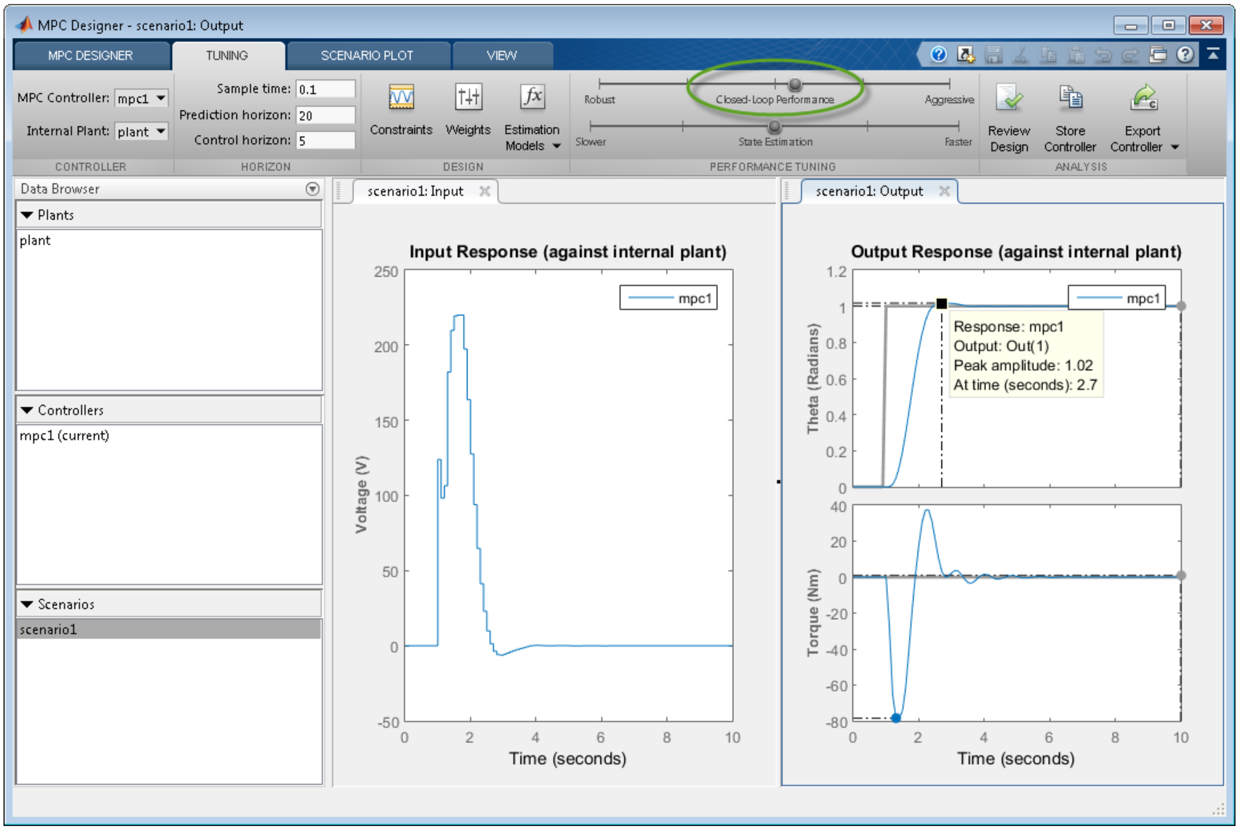


Response: mpc1
Output: Out(1)
Peak amplitude: 1.03
At time (seconds): 3

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.

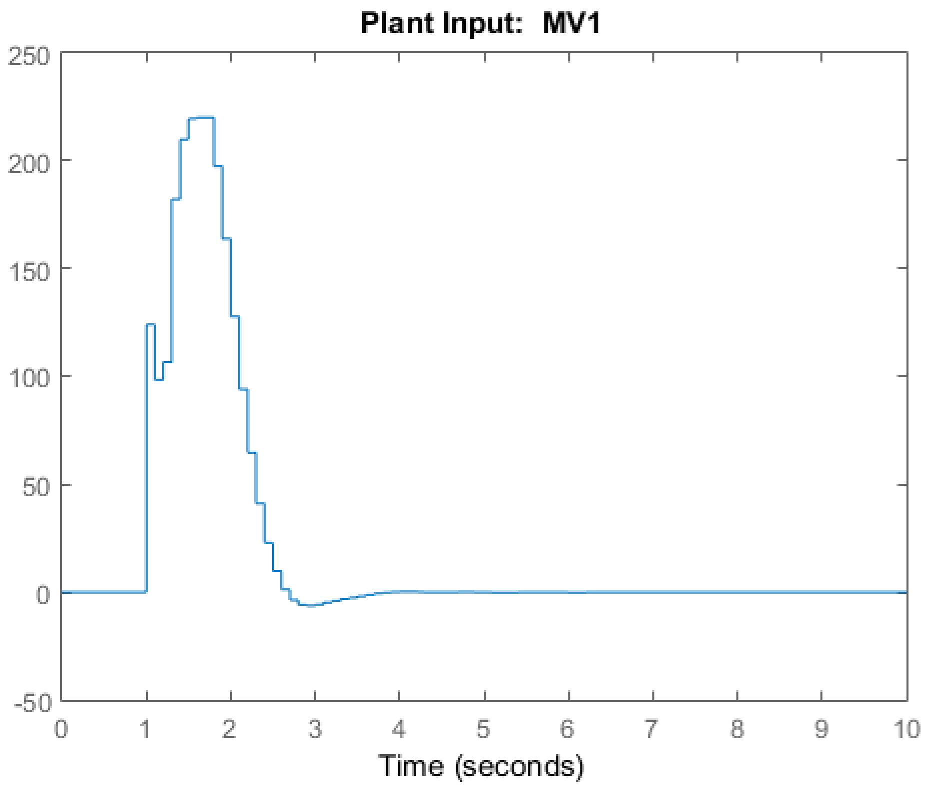
Additionally, the app generates the following code in the MATLAB Editor.

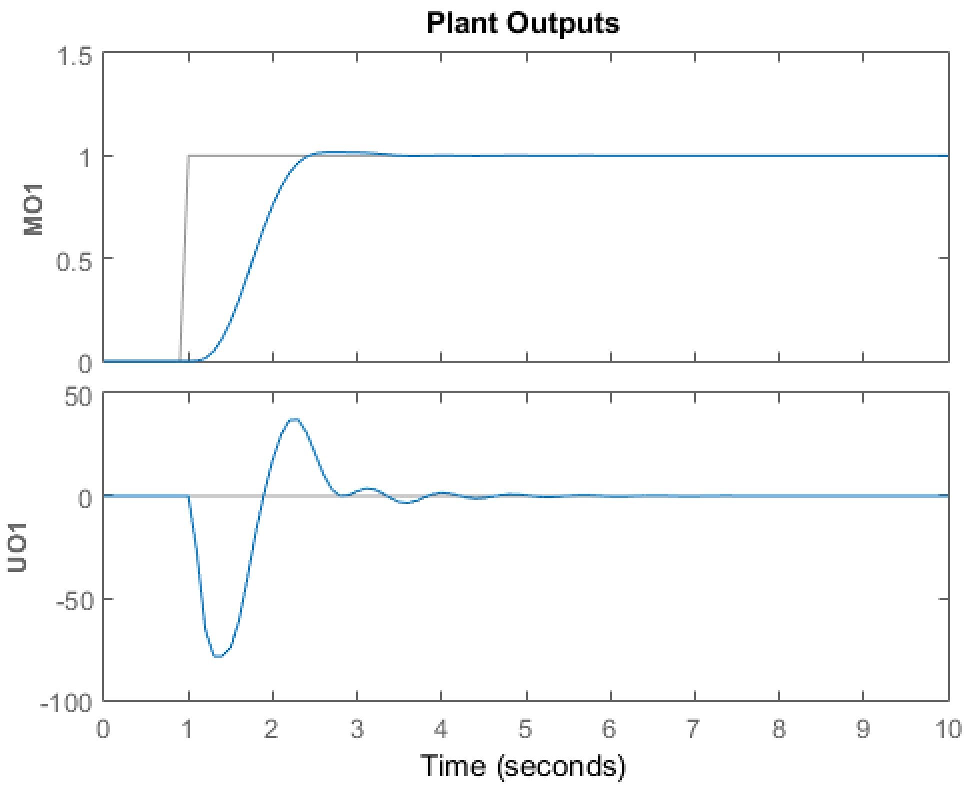
```
%% create MPC controller object with sample time
mpc1 = mpc(plant_C, 0.1);
%% specify prediction horizon
mpc1.PredictionHorizon = 20;
%% specify control horizon
mpc1.ControlHorizon = 5;
%% specify nominal values for inputs and outputs
mpc1.Model.Nominal.U = 0;
mpc1.Model.Nominal.Y = [0;0];
%% specify 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)

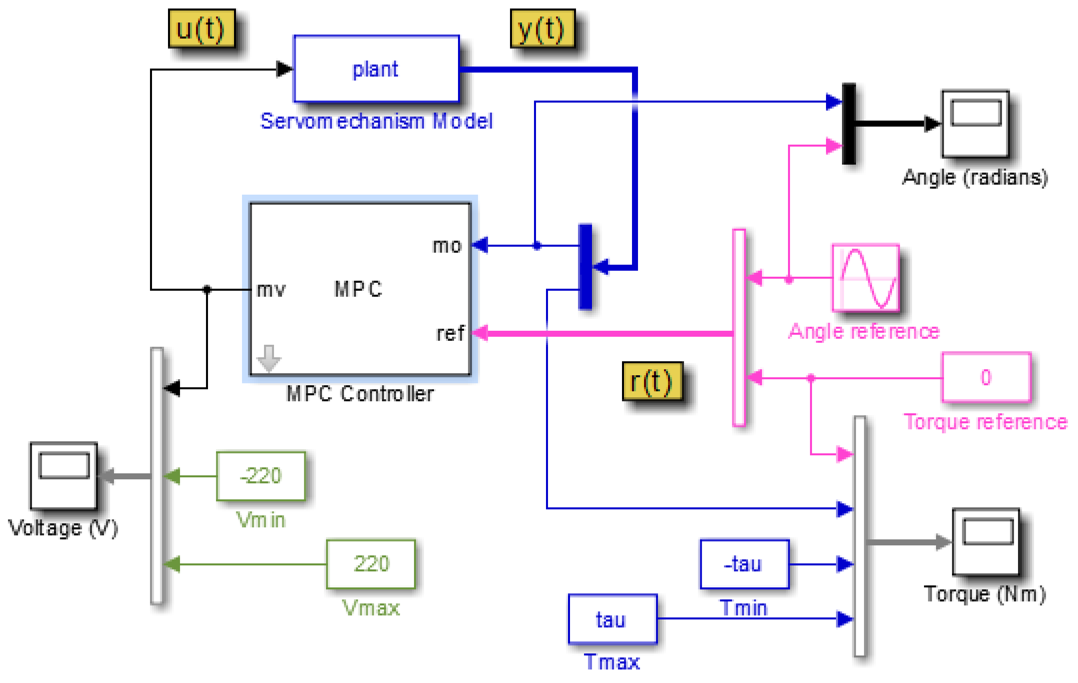


The script creates the controller, `mpc1`, and runs the simulation scenario. The input and output responses match the simulation results from the app.

Validate Controller Performance In Simulink

Open the servomechanism Simulink model.

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

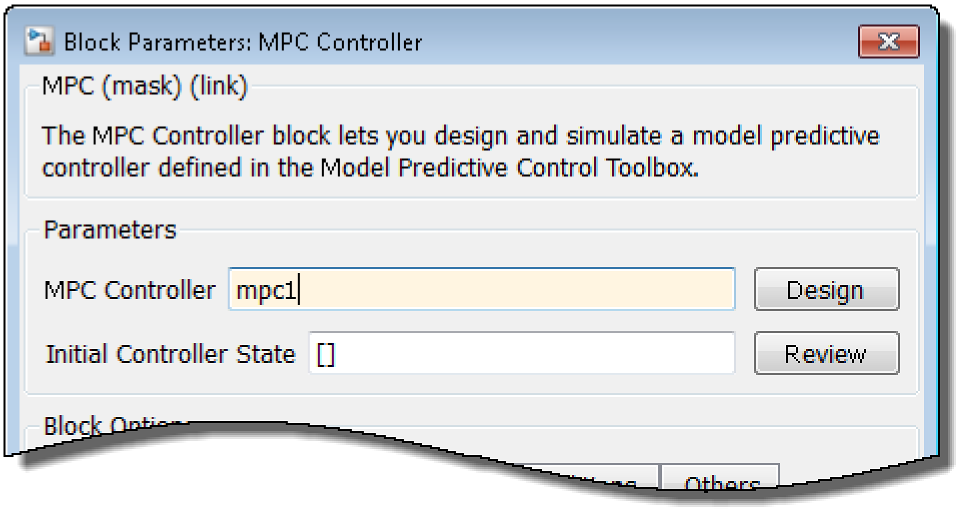


This model uses an MPC Controller block to control a servomechanism plant. The Servomechanism Model block is already configured to use the `plant` model from the MATLAB workspace.

The Angle reference source block creates a sinusoidal reference signal with a frequency of `0.4` rad/sec and an amplitude of π.

Double-click the MPC Controller block.

In the MPC Controller Block Parameters dialog box, specify an **MPC Controller** from the MATLAB workspace. Use the `mpc1` controller created using the generated script.


Block Parameters: MPC Controller

MPC (mask) (link)

The MPC Controller block lets you design and simulate a model predictive controller defined in the Model Predictive Control Toolbox.

Parameters

MPC Controller: mpc1

Initial Controller State: []

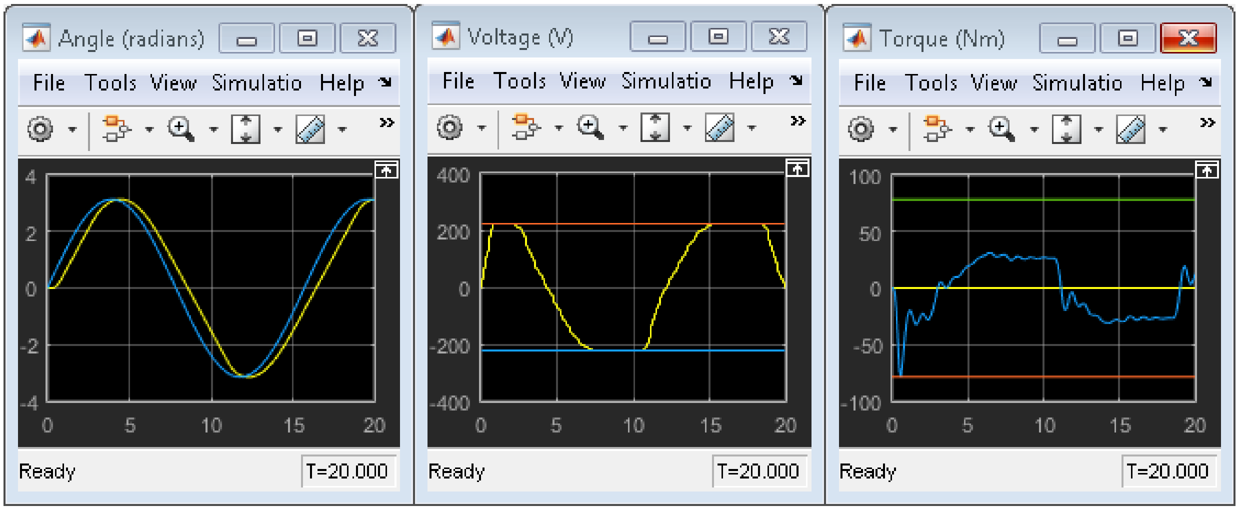
Click **OK**.

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

More About

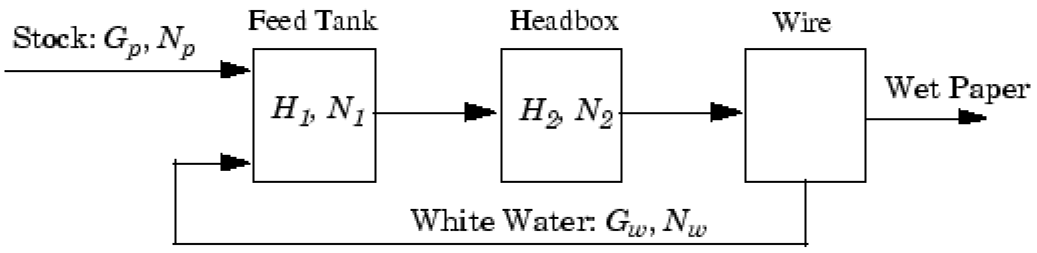
- “Design Controller Using MPC Designer”
- “Design MPC Controller at the Command Line”

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

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

- H_1 — Feed tank liquid level
- H_2 — Headbox liquid level
- N_1 — Feed tank consistency
- N_2 — Headbox consistency

The primary control objective is to hold H_2 and N_2 at their setpoints by adjusting the manipulated variables:

- G_p — Flow rate of stock entering the feed tank
- G_w — Flow rate of recycled white water

The consistency of stock entering the feed tank, N_p, is a measured disturbance, and the white water consistency, N_w, is an unmeasured disturbance.

All signals are normalized with zero nominal steady-state values and comparable numerical ranges. The process is open-loop stable.

The measured outputs are H_2, N_1, and N_2.

The Simulink S-function, `mpc_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.

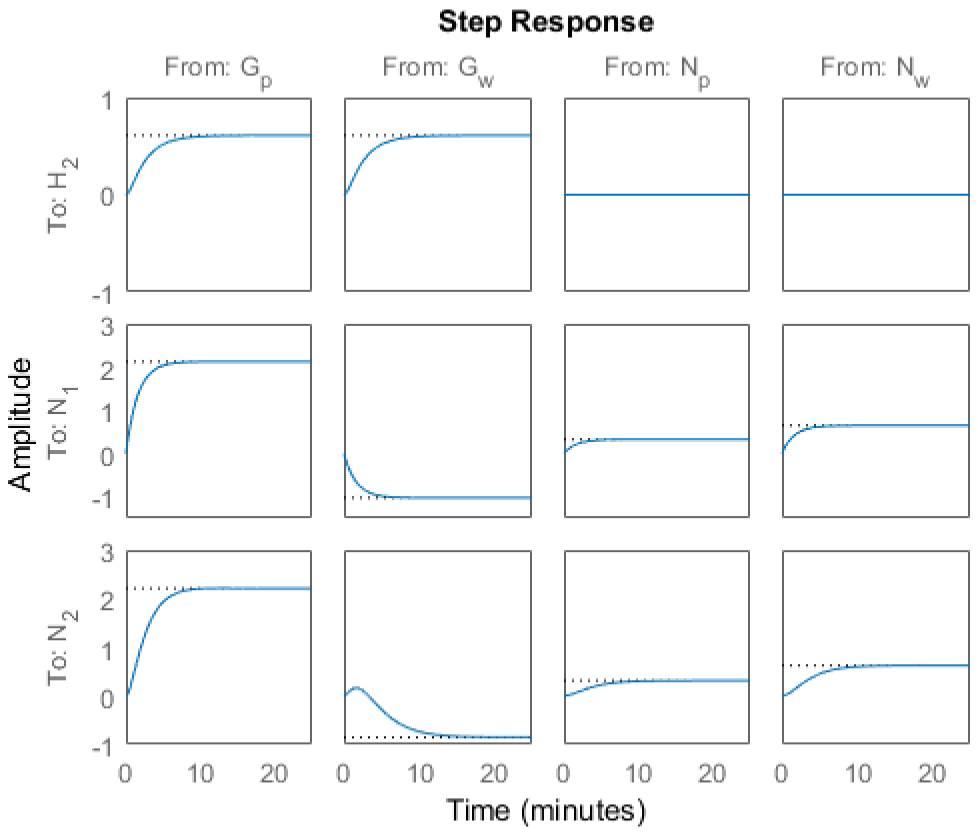
```
PaperMach.InputName = {'G_p','G_w','N_p','N_w'};
PaperMach.OutputName = {'H_2','N_1','N_2'};
```

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

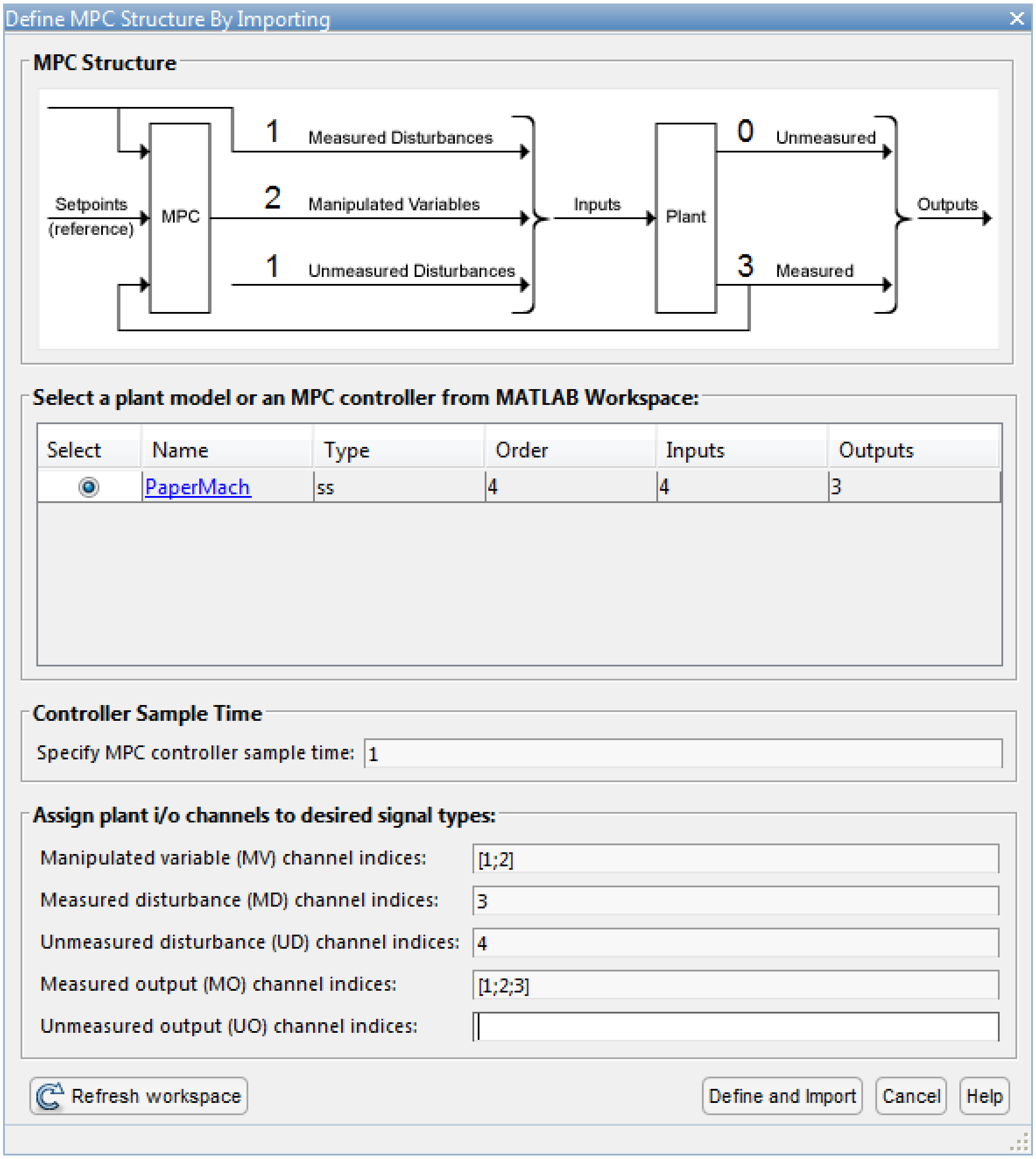
Open the **MPC Designer** app.

```
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



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

Select	Name	Type	Order	Inputs	Outputs
●	PaperMach	ss	4	4	3

Controller Sample Time

Specify MPC controller sample time:	1

Assign plant i/o channels to desired signal types:

Manipulated variable (MV) channel indices:	[1;2]
Measured disturbance (MD) channel indices:	3
Unmeasured disturbance (UD) channel indices:	4
Measured output (MO) channel indices:	[1;2;3]
Unmeasured output (UO) channel indices:	

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

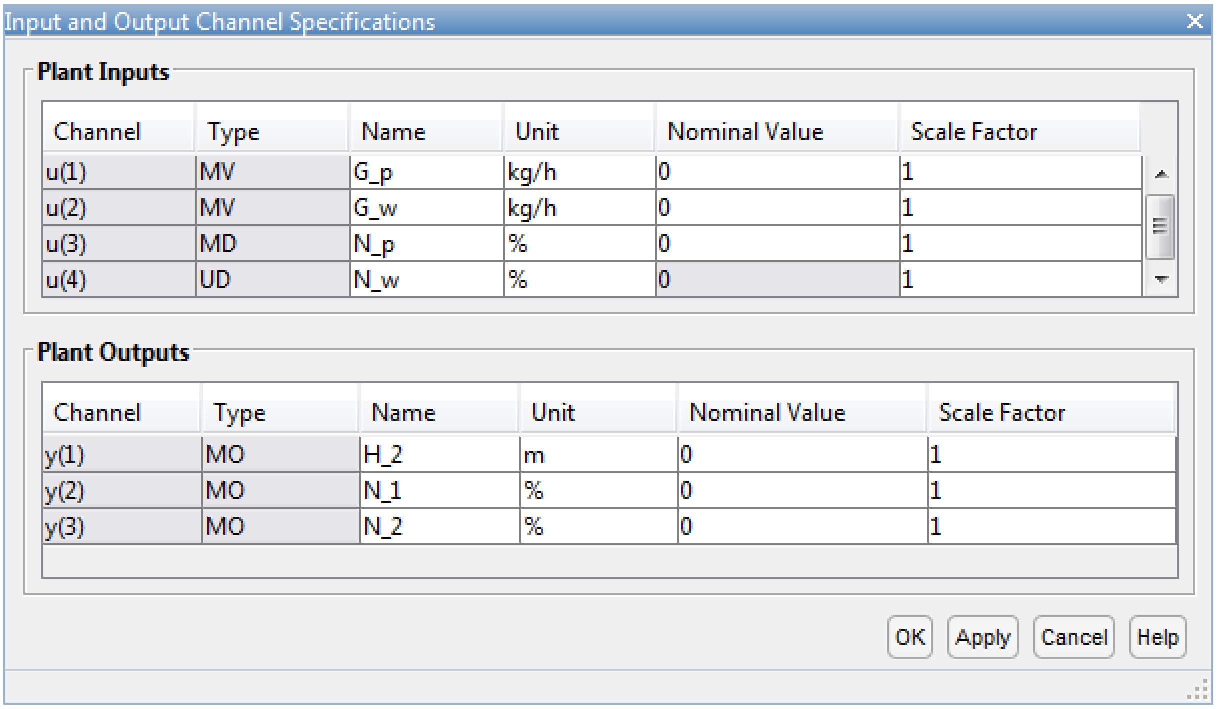
Click **Define and Import**.

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.



Plant Inputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
u(1)	MV	G_p	kg/h	0	1
u(2)	MV	G_w	kg/h	0	1
u(3)	MD	N_p	%	0	1
u(4)	UD	N_w	%	0	1

Plant Outputs

Channel	Type	Name	Unit	Nominal Value	Scale Factor
y(1)	MO	H_2	m	0	1
y(2)	MO	N_1	%	0	1
y(3)	MO	N_2	%	0	1

Click **OK** to update the channel attributes and close the dialog box.

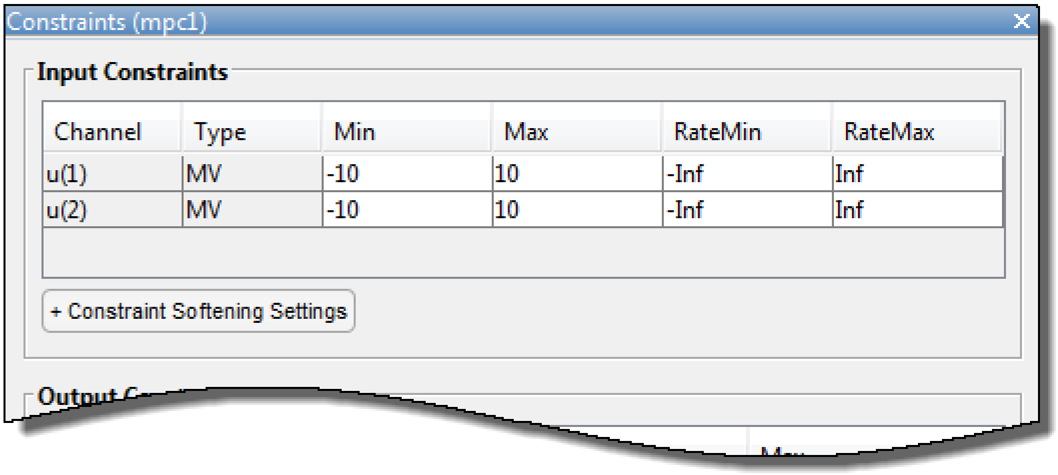
Specify Controller Sample Time and Horizons

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.



Channel	Type	Min	Max	RateMin	RateMax
u(1)	MV	-10	10	-Inf	Inf
u(2)	MV	-10	10	-Inf	Inf

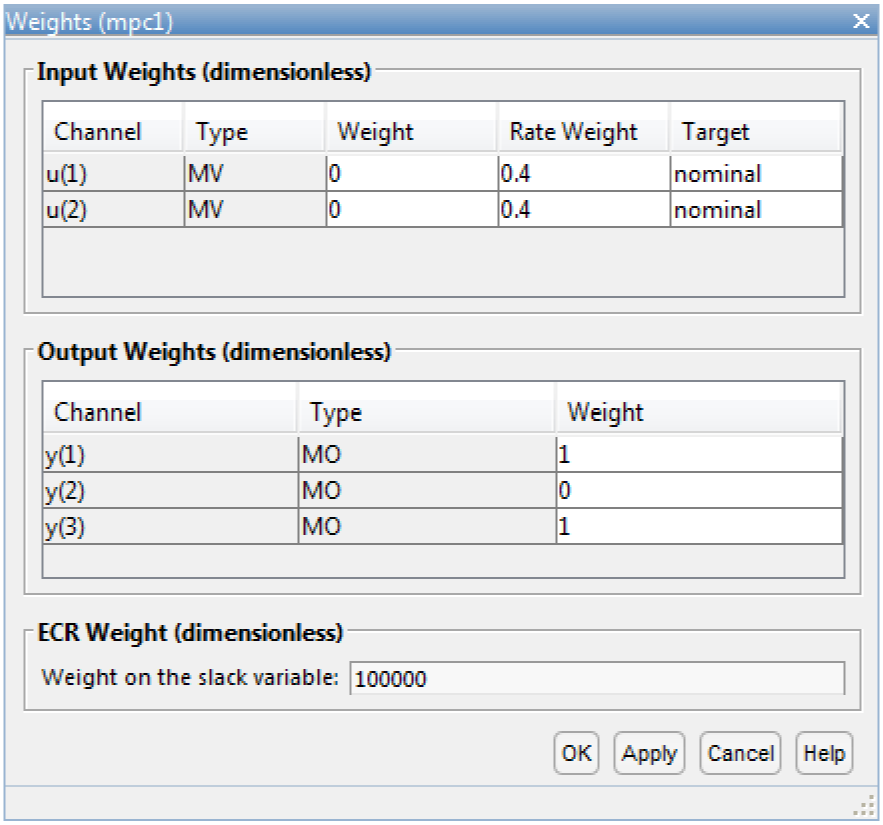
Click **OK**.

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.



Input Weights (dimensionless)

Channel	Type	Weight	Rate Weight	Target
u(1)	MV	0	0.4	nominal
u(2)	MV	0	0.4	nominal

Output Weights (dimensionless)

Channel	Type	Weight
y(1)	MO	1
y(2)	MO	0
y(3)	MO	1

ECR Weight (dimensionless)

Weight on the slack variable: 100000

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.

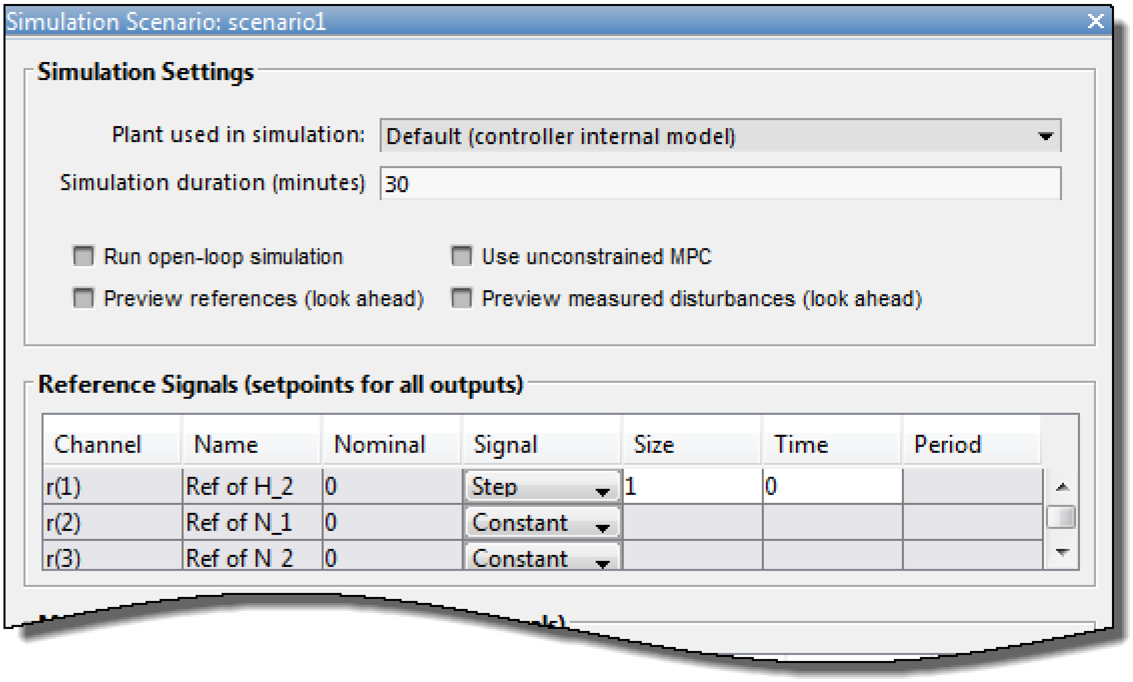
Simulate H_2 Setpoint Step Response

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.



Simulation Scenario: scenario1

Simulation Settings

Plant used in simulation: Default (controller internal model)

Simulation duration (minutes) 30

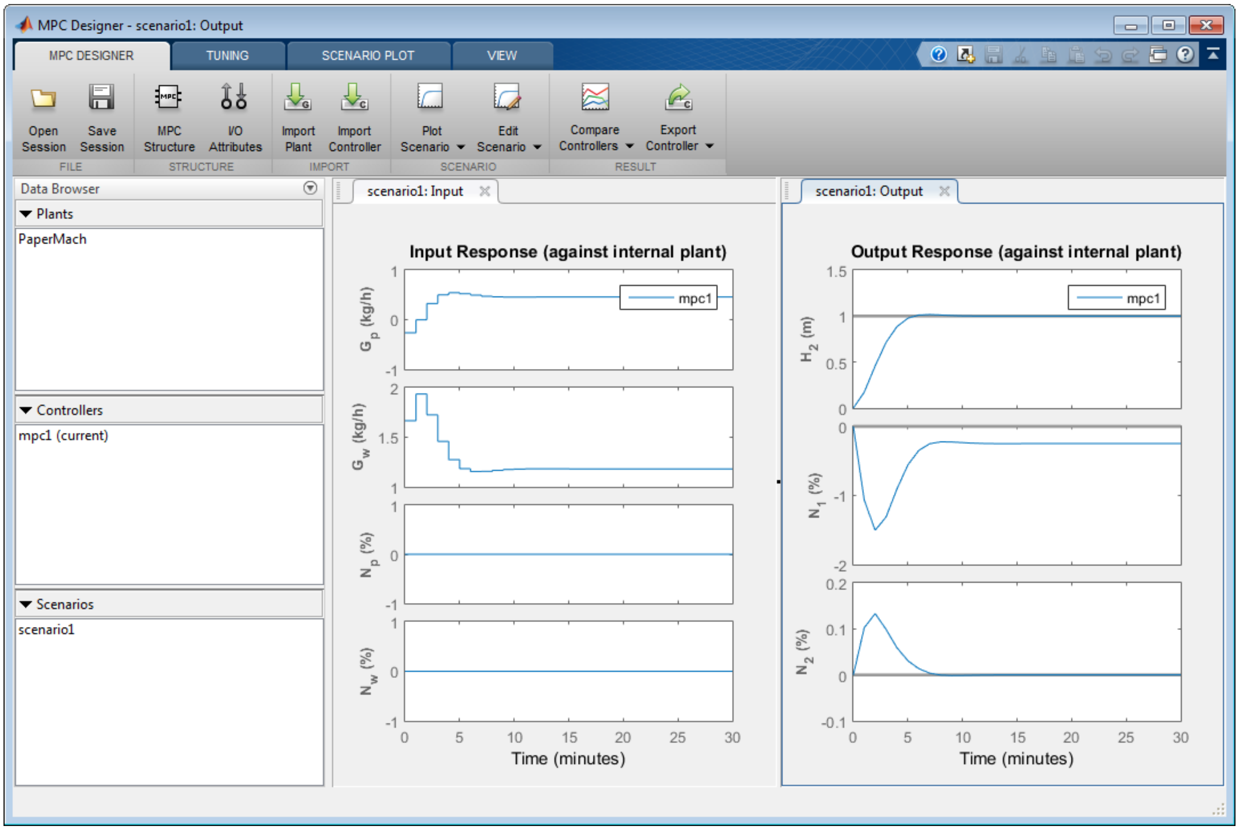
- [ ] Run open-loop simulation
- [ ] Use unconstrained MPC
- [ ] Preview references (look ahead)
- [ ] Preview measured disturbances (look ahead)

Reference Signals (setpoints for all outputs)

Channel	Name	Nominal	Signal	Size	Time	Period
r(1)	Ref of H_2	0	Step	1	0	
r(2)	Ref of N_1	0	Constant			
r(3)	Ref of N_2	0	Constant			

Click **OK**.

The app runs the simulation with the new scenario settings and updates the **Input Response** and **Output Response** plots.



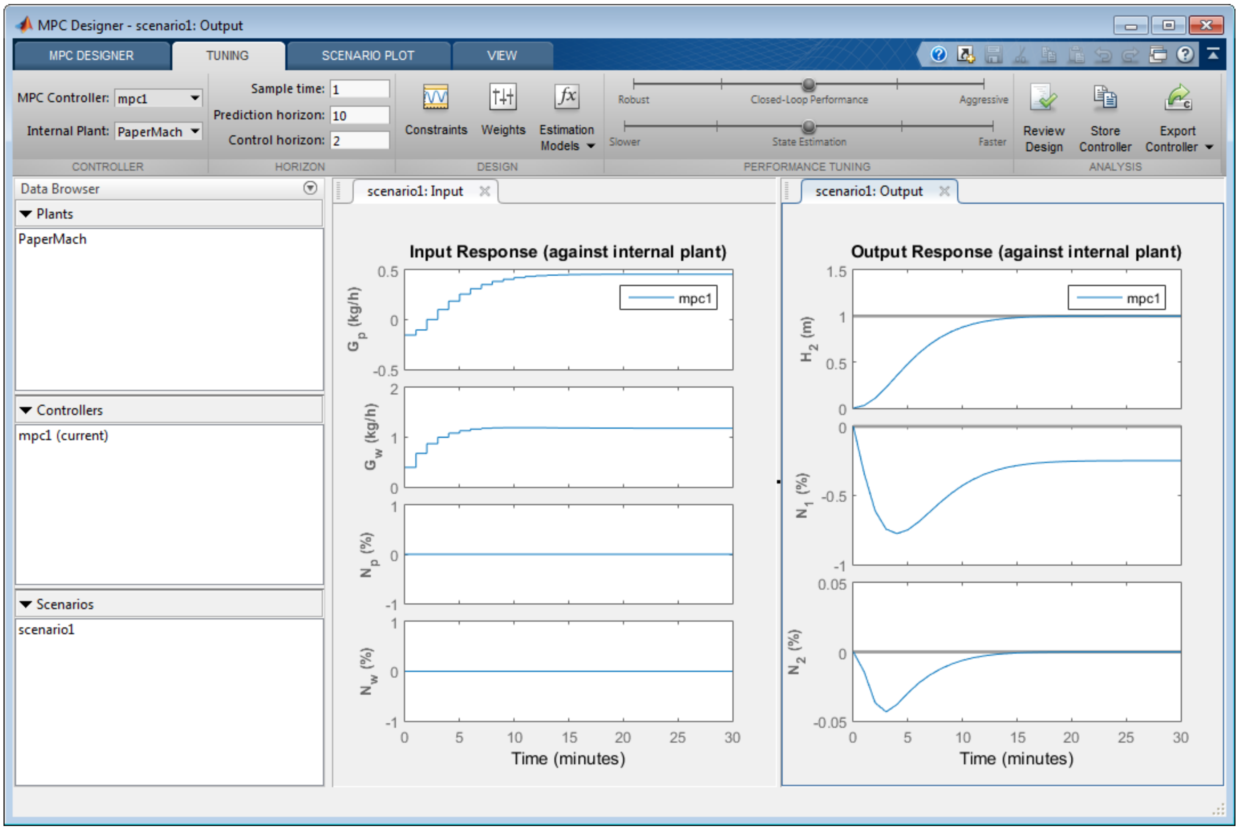
The initial design uses a conservative control effort to produce a robust controller. The response time for output H_2 is about 7 minutes. To reduce this response time, you can decrease the sample time, reduce the manipulated variable rate weights, or reduce the manipulated variable rate constraints.

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.



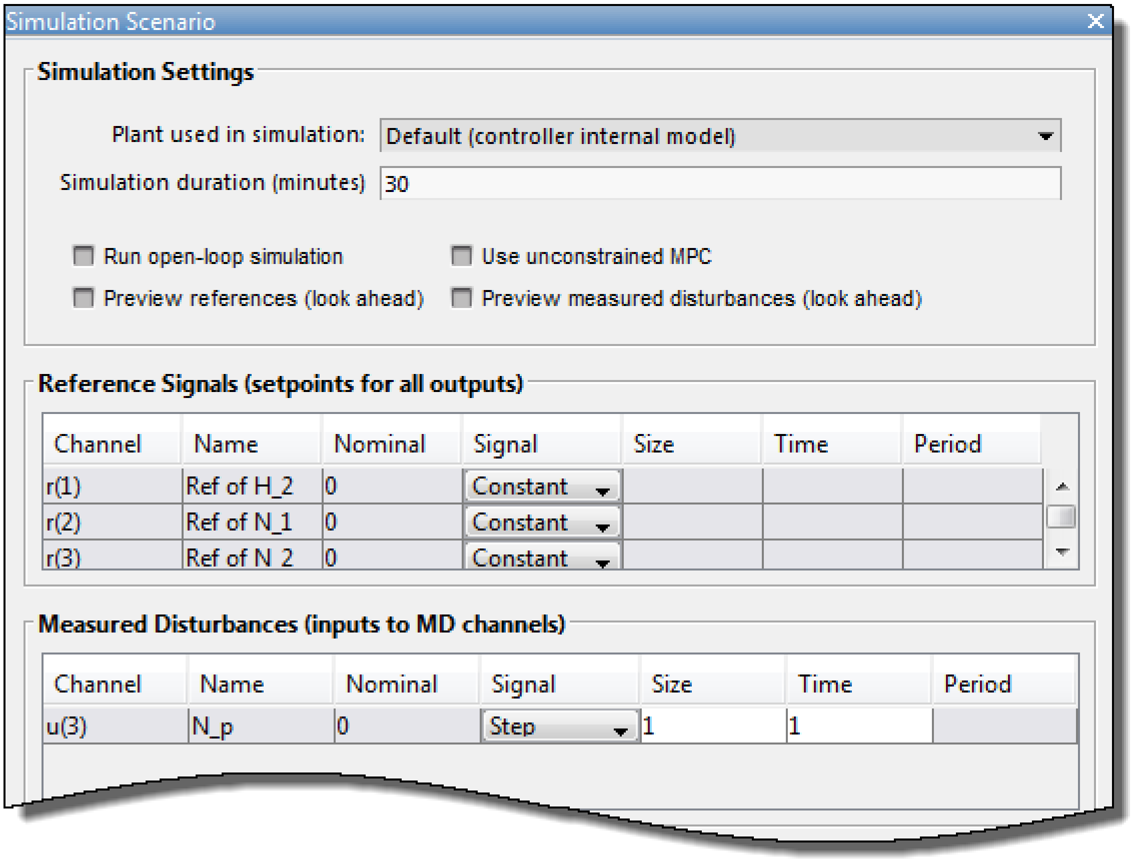
The controller places more emphasis on eliminating errors in feed tank consistency, N_2, which significantly decreases the peak absolute error. The trade-off is a longer response time of about 17 minutes for the feed tank level, H_2.

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.



Simulation Settings

Plant used in simulation: Default (controller internal model)

Simulation duration (minutes) 30

- [ ] Run open-loop simulation
- [ ] Use unconstrained MPC
- [ ] Preview references (look ahead)
- [ ] Preview measured disturbances (look ahead)

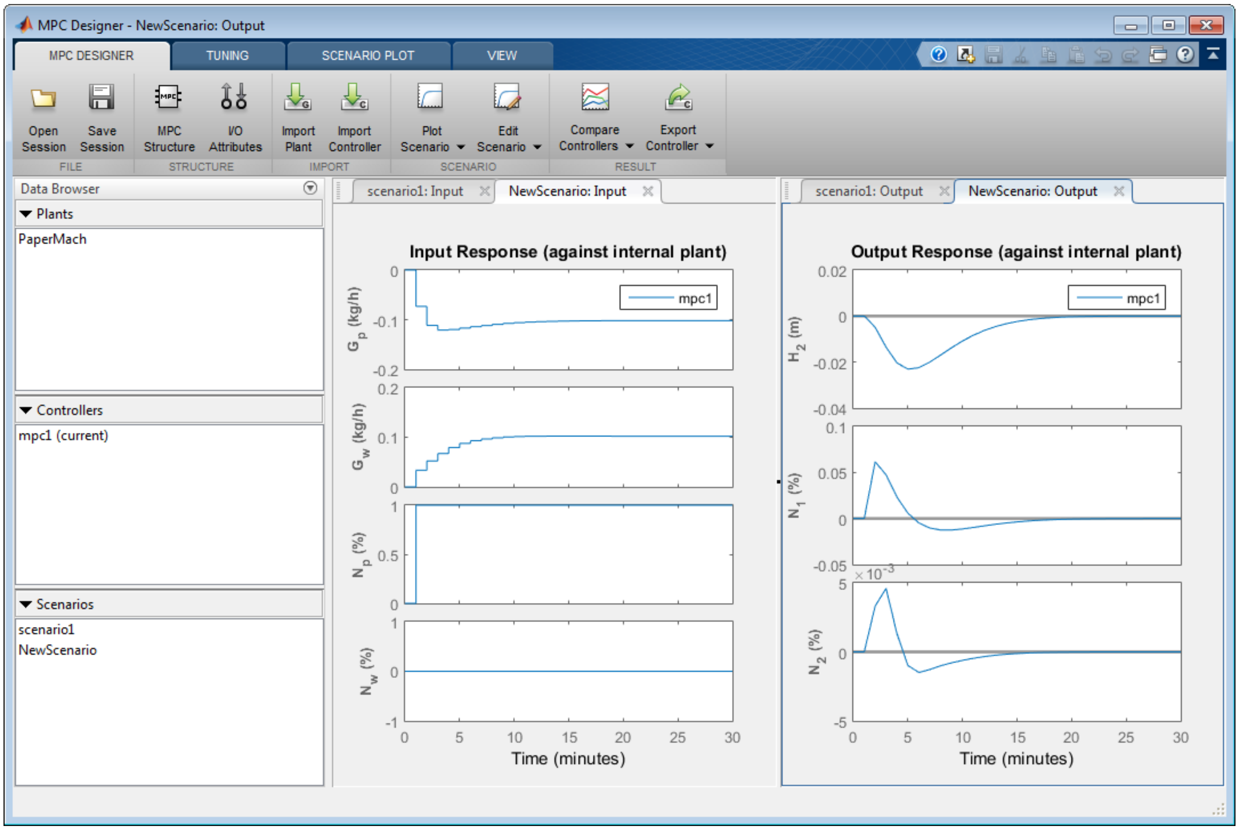
Reference Signals (setpoints for all outputs)

Channel	Name	Nominal	Signal	Size	Time	Period
r(1)	Ref of H_2	0	Constant			
r(2)	Ref of N_1	0	Constant			
r(3)	Ref of N_2	0	Constant			

Measured Disturbances (inputs to MD channels)

Channel	Name	Nominal	Signal	Size	Time	Period
u(3)	N_p	0	Step	1	1	

Click **OK** to run the simulation and display the input and output response plots.



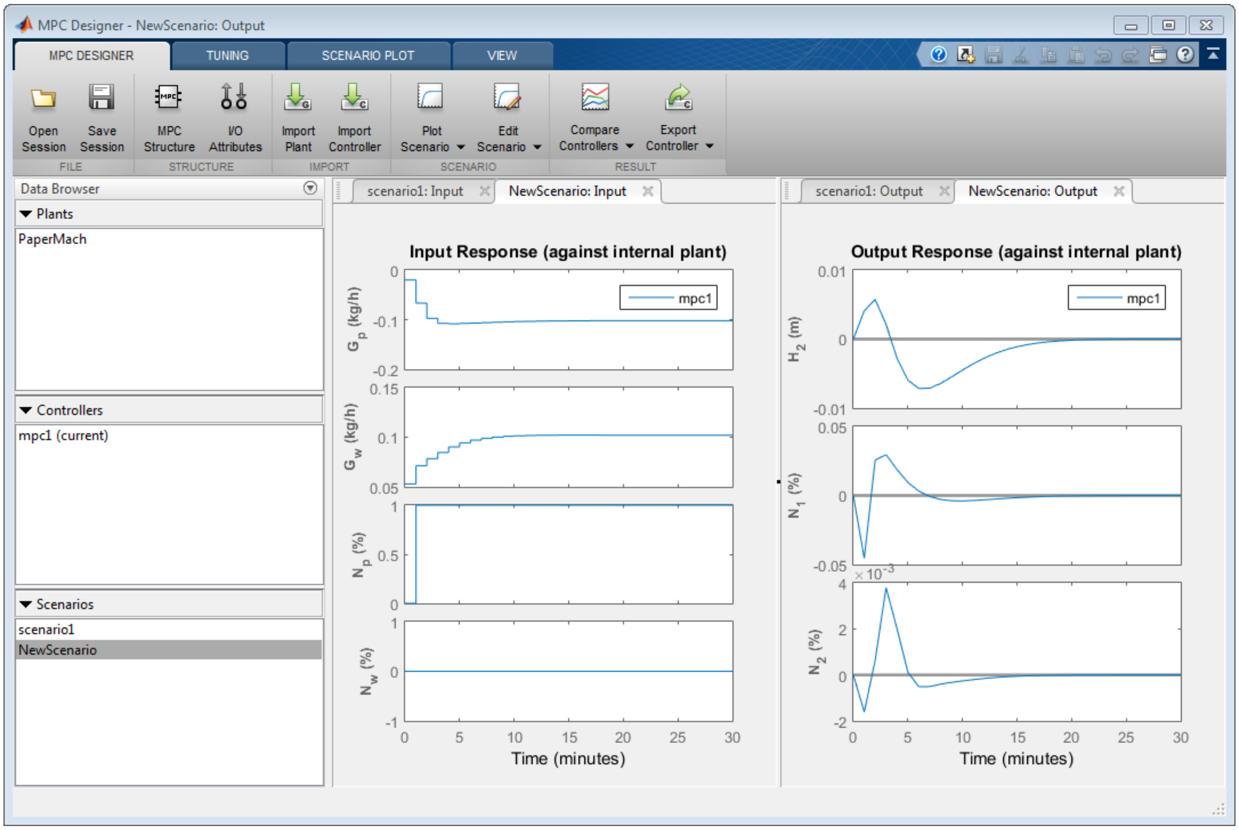
As shown in the **NewScenario: Output** plot, both H_2 and N_2 deviate little from their setpoints.

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**.



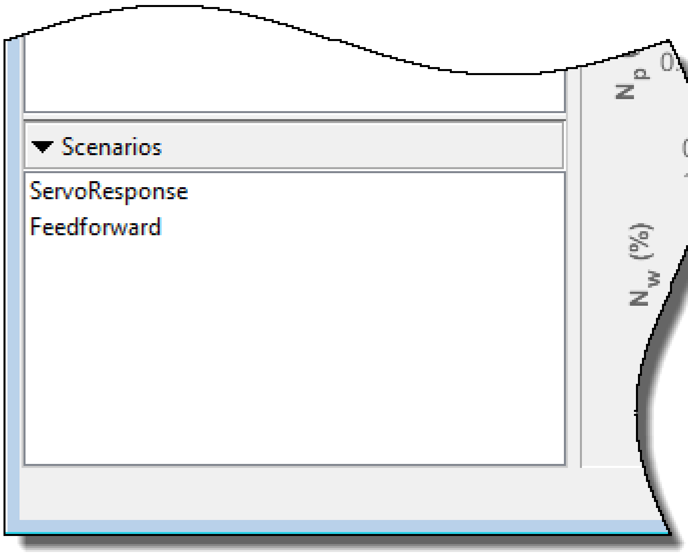
The manipulated variables begin changing before the measured disturbance occurs because the controller uses the known future disturbance value when computing its control action. The output disturbance values also begin changing before the disturbance occurs, which reduces the magnitude of the output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

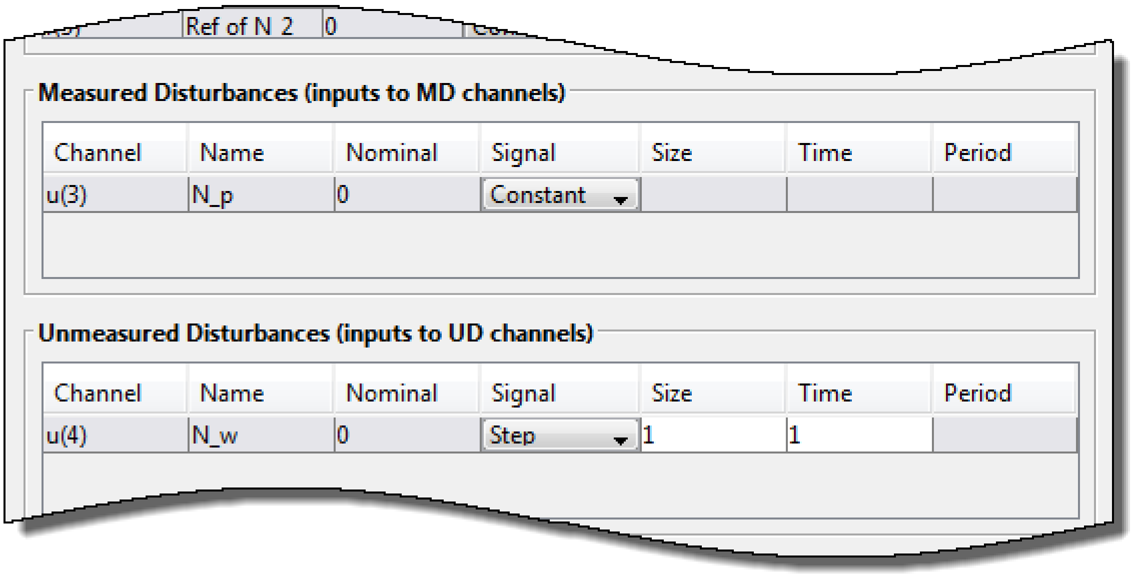
Double-click the new scenario, and rename it `Feedback`.

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`.



	Ref of N 2	0	

Measured Disturbances (inputs to MD channels)

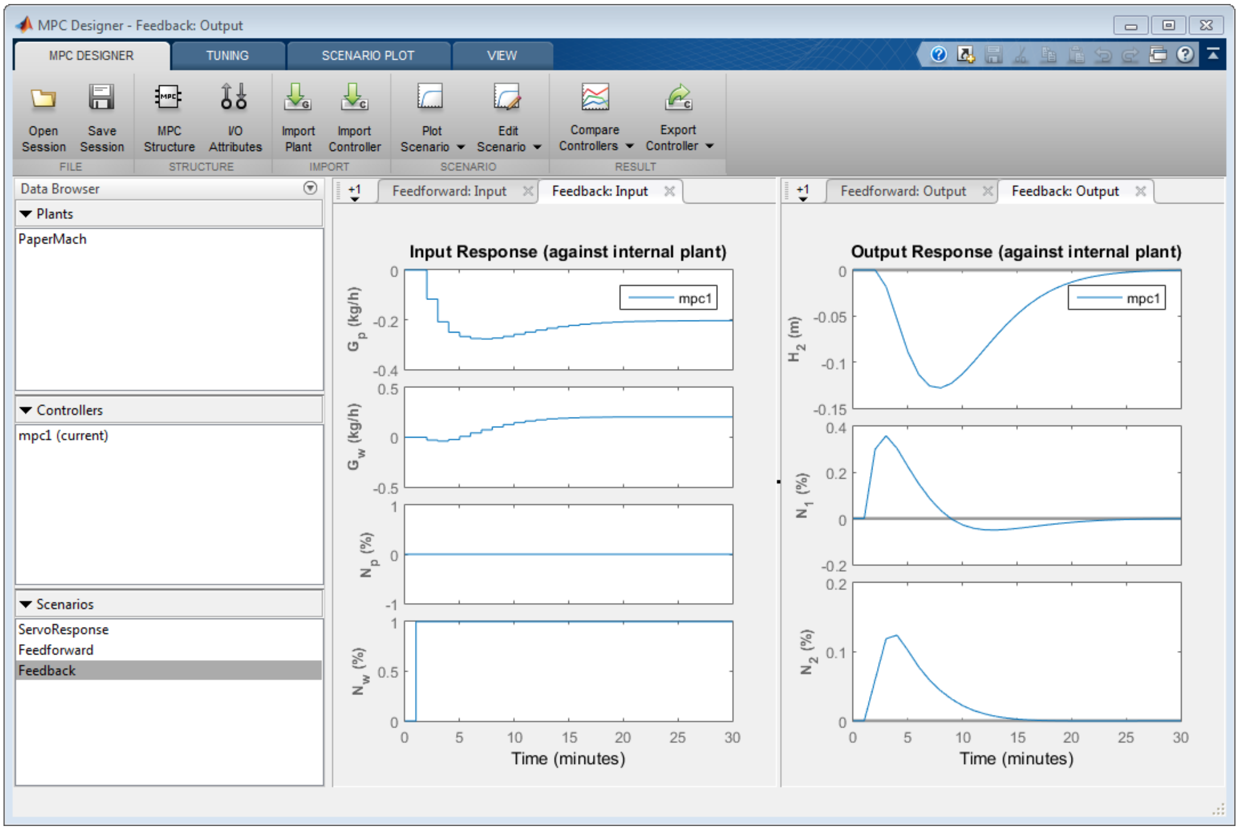
Channel	Name	Nominal	Signal	Size	Time	Period
u(3)	N_p	0	Constant			

Unmeasured Disturbances (inputs to UD channels)

Channel	Name	Nominal	Signal	Size	Time	Period
u(4)	N_w	0	Step	1	1	

Click **OK** to update the scenario settings, and run the simulation.

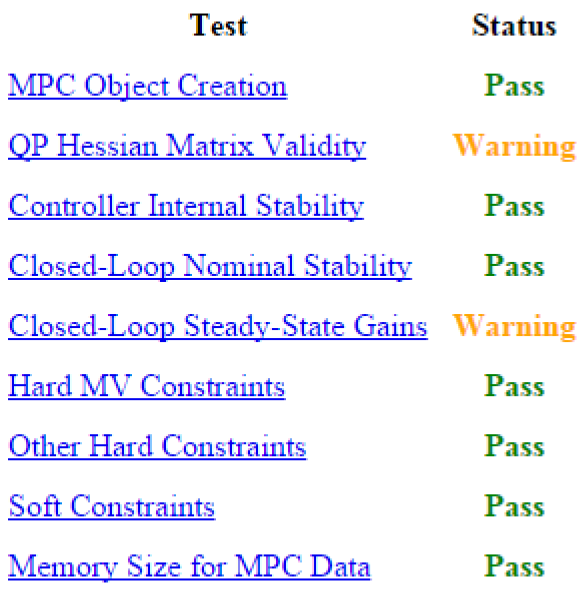
In the **Data Browser**, in the **Scenarios** section, right-click `Feedback`, and select **Plot**.



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.

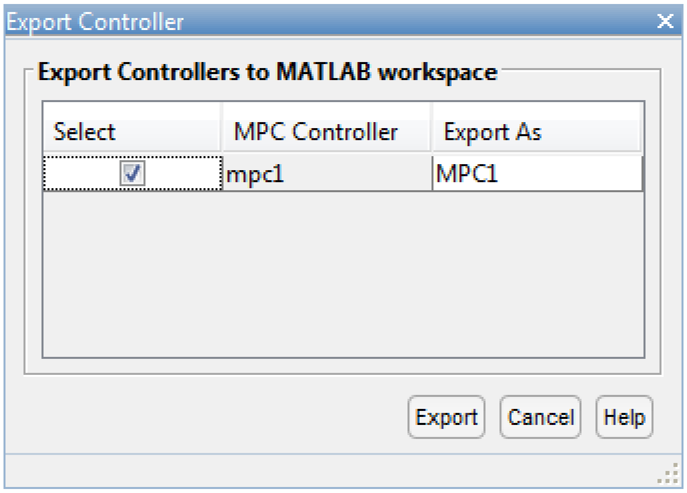
Since the input/output imbalance is a known feature of the paper machine plant model, and you intentionally set one of the output weights to zero to correct for the imbalance, neither warning indicates an issue with the controller design.

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.



Export Controller

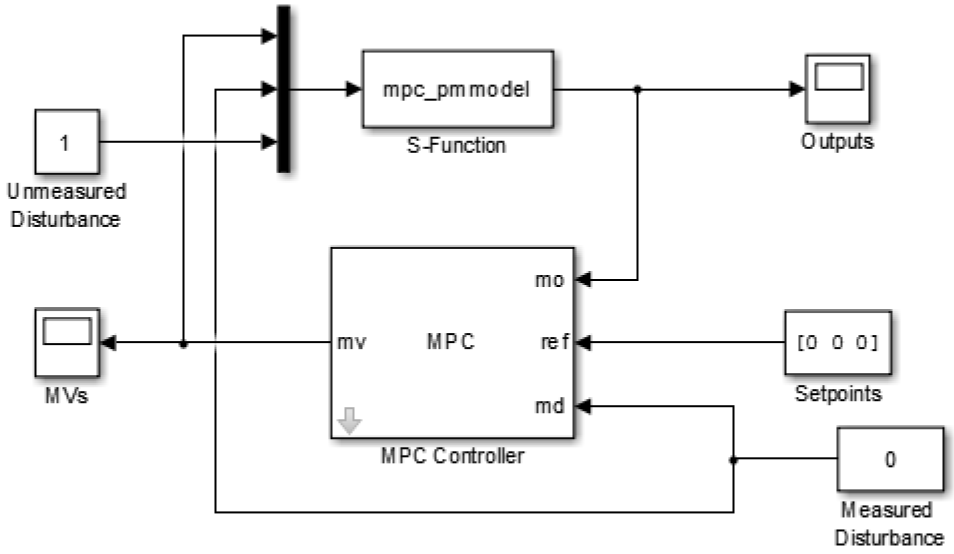
Export Controllers to MATLAB workspace

Select	MPC Controller	Export As
✓	mpc1	MPC1

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

Open and Simulate Simulink Model

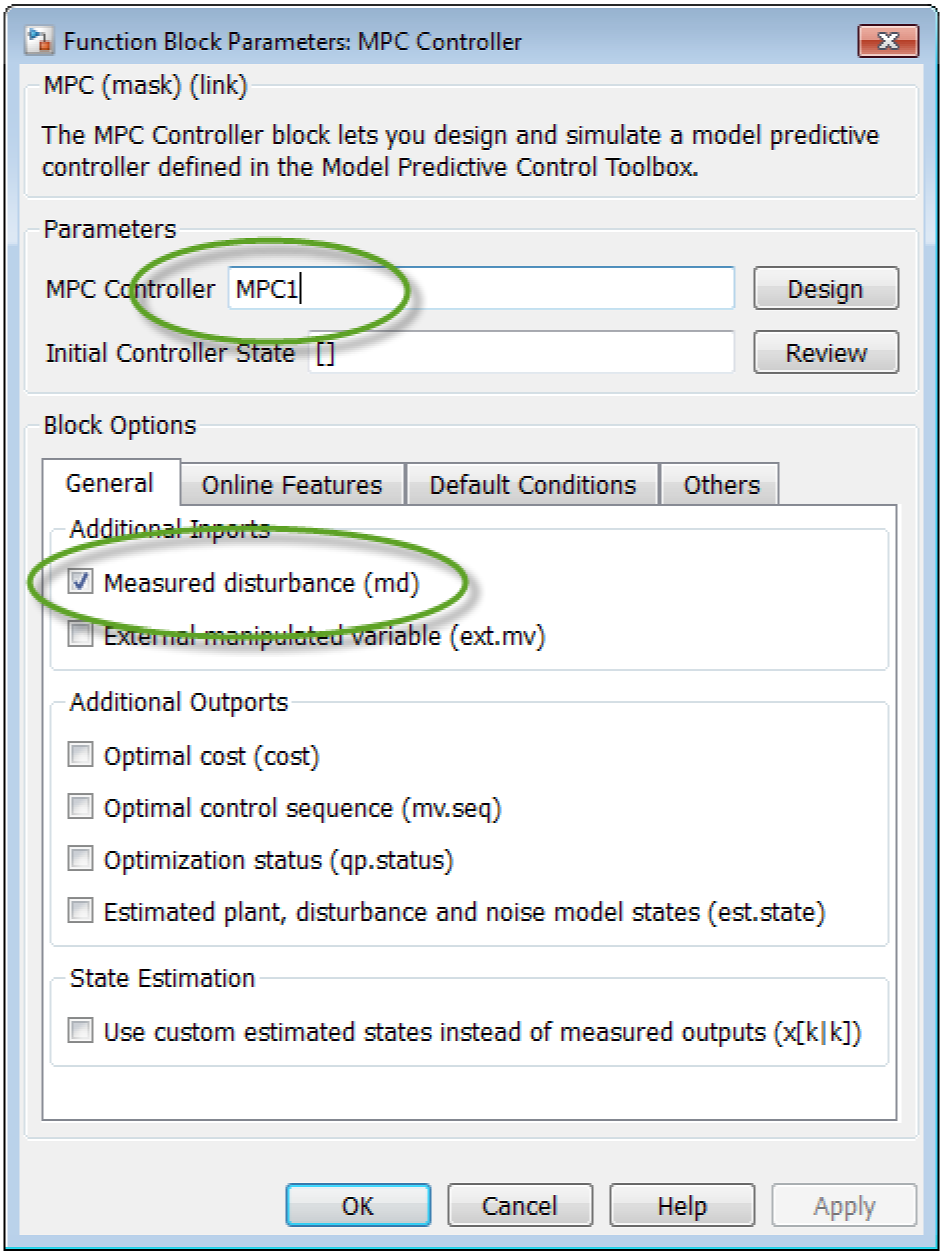
```
open_system('mpc_papermachine')
```



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

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

Double-click the MPC Controller block.


Function Block Parameters: MPC Controller

MPC (mask) (link)

The MPC Controller block lets you design and simulate a model predictive controller defined in the Model Predictive Control Toolbox.

Parameters

MPC Controller: MPC1

Initial Controller State: []

Block Options

General | Online Features | Default Conditions | Others

Additional Inports

[✓] Measured disturbance (md)

[ ] External manipulated variable (ext.mv)

Additional Outports

[ ] Optimal cost (cost)

[ ] Optimal control sequence (mv.seq)

[ ] Optimization status (qp.status)

[ ] Estimated plant, disturbance and noise model states (est.state)

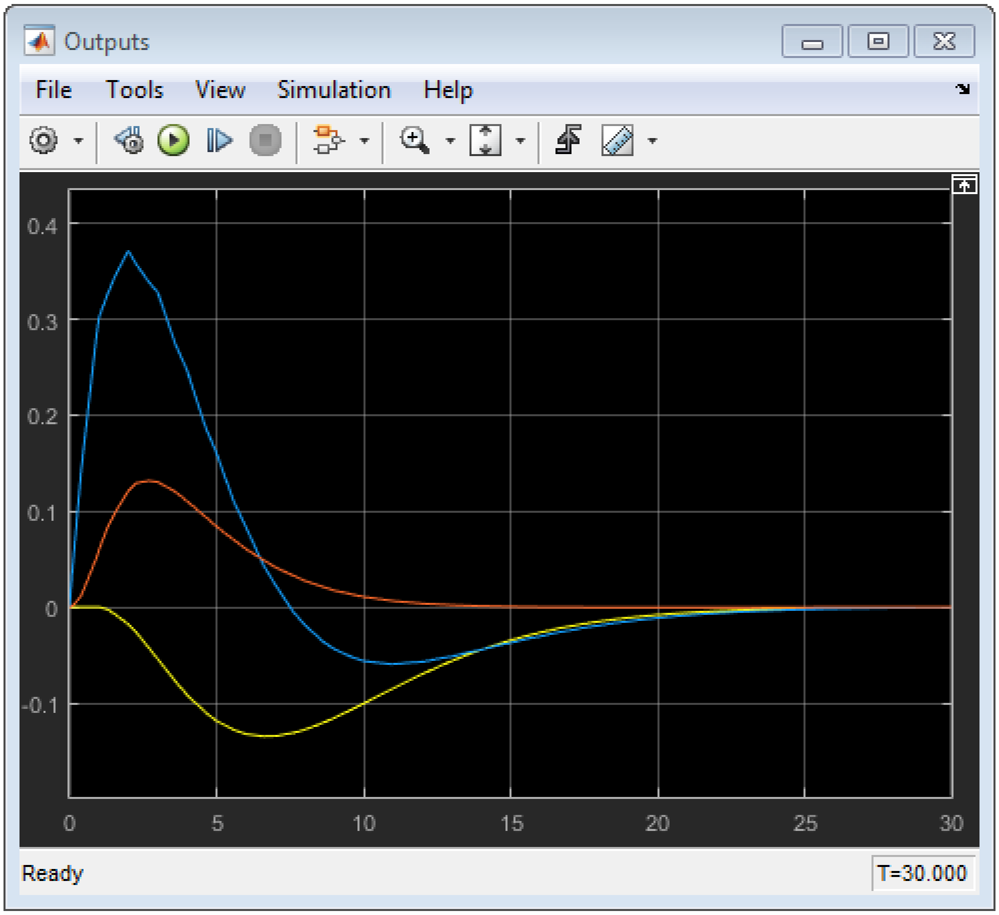
State Estimation

[ ] Use custom estimated states instead of measured outputs (x[k|k])

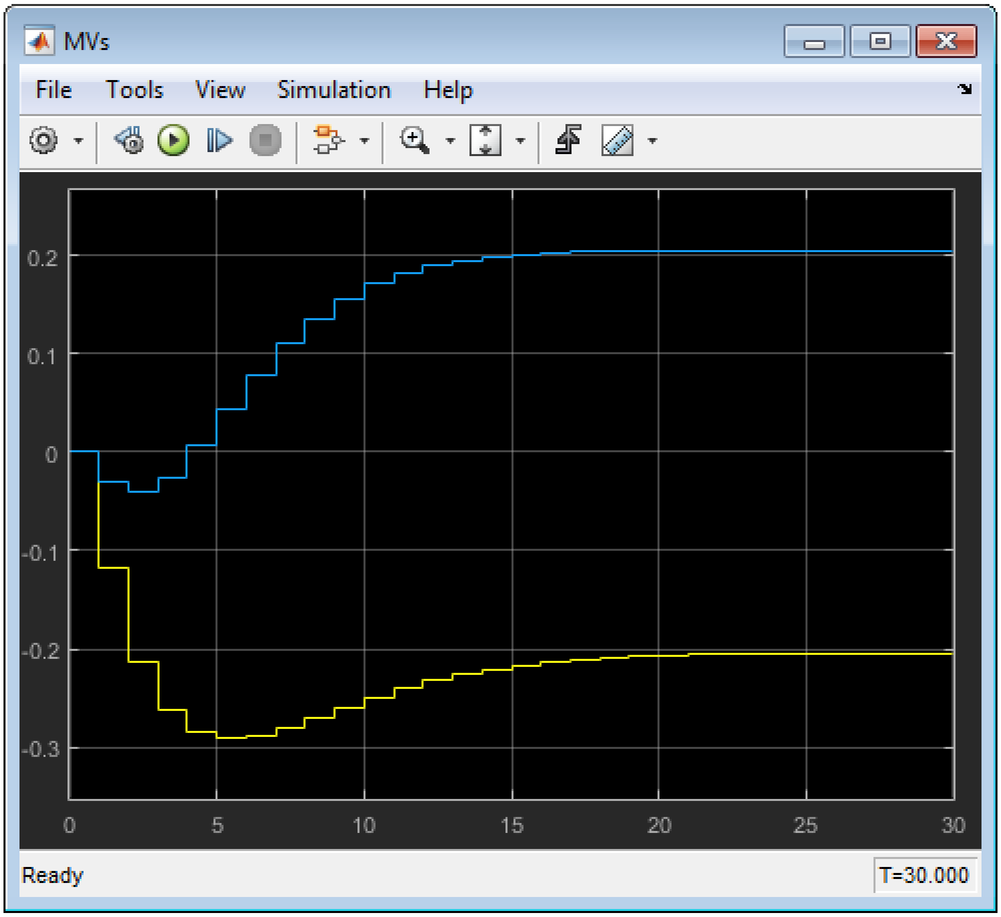
The MPC Controller block is already configured to use the `MPC1` controller that was previously exported to the MATLAB workspace.

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.

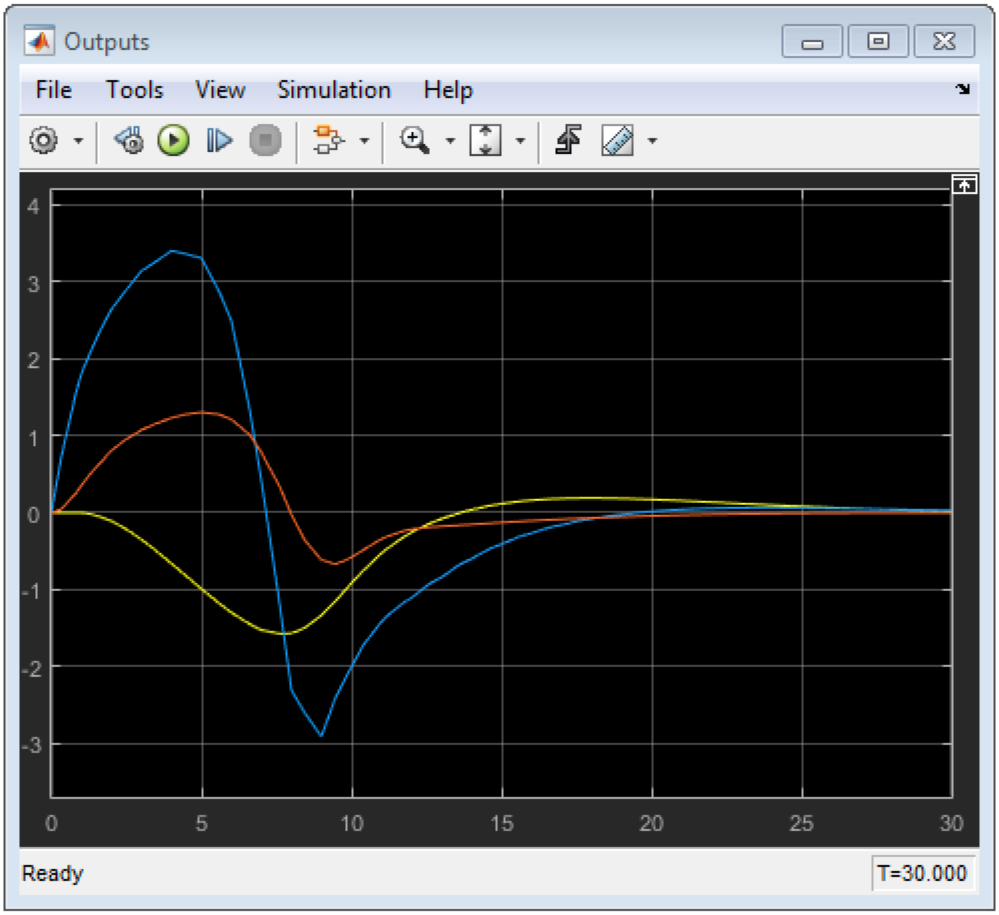
Increase Unmeasured Disturbance Magnitude

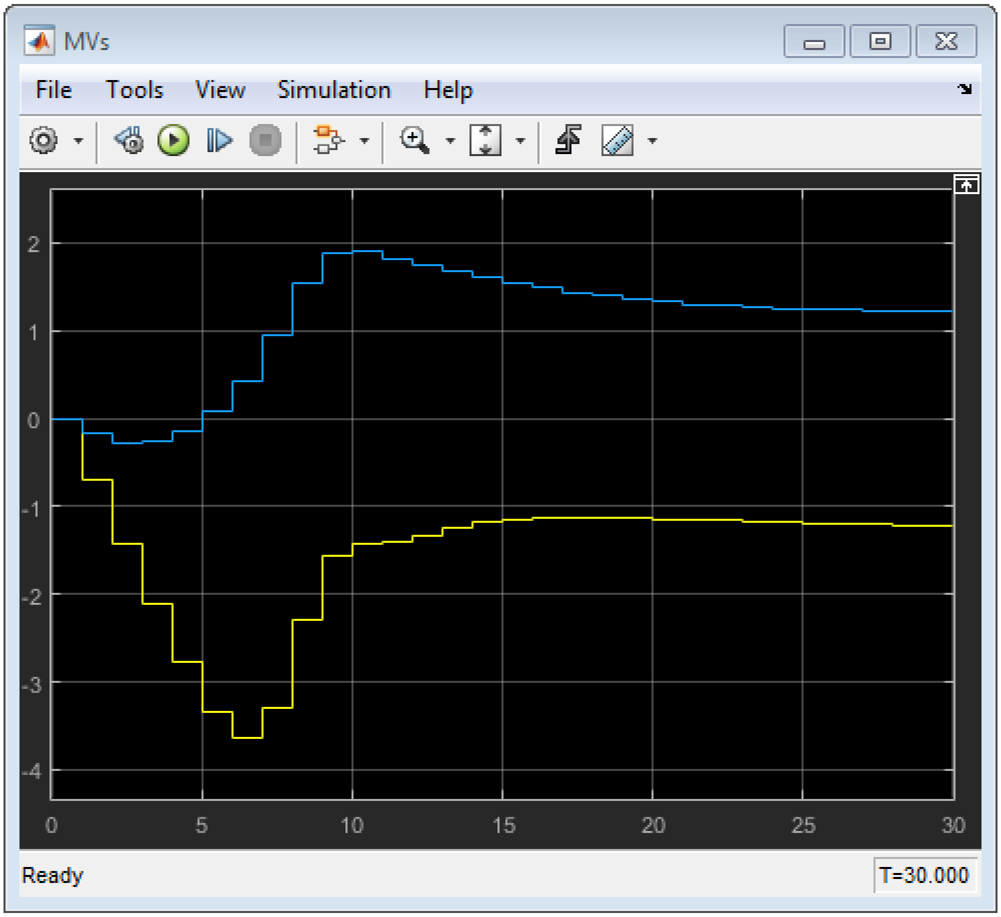
In the Simulink model window, double-click the Unmeasured Disturbance block.

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

Click **OK**.

Simulate the model.





The mismatch between the prediction model and the plant now produces output responses with significant differences. Increasing the disturbance magnitude further results in large setpoint deviations and saturated manipulated variables.

References

[1] Ying, Y., M. Rao, and Y. Sun “Bilinear control strategy for paper making process,” *Chemical Engineering Communications* (1992), Vol. 111, pp. 13–28.

See Also

MPC Controller | **MPC Designer**

More About

- “Design Controller Using MPC Designer”

Bumpless Transfer Between Manual and Automatic Operations

This example shows how to bumplessly transfer between manual and automatic operations of a plant.

During startup of a manufacturing process, operators adjust key actuators manually until the plant is near the desired operating point before switching to automatic control. If not done correctly, the transfer can cause a *bump*, that is, large actuator movement.

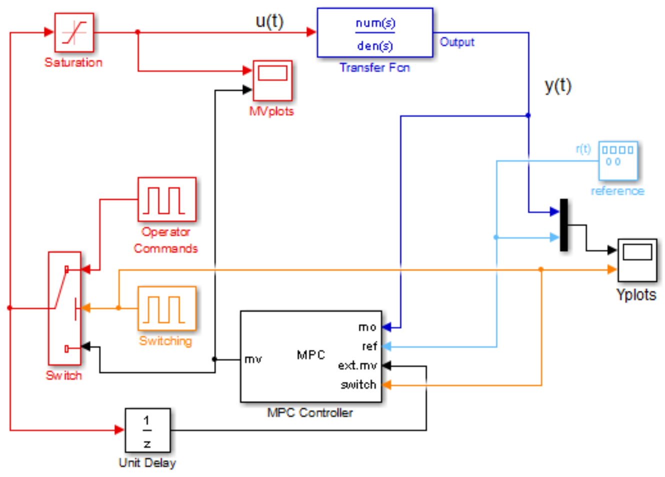
In this example, you simulate a Simulink model that contains a single-input single-output LTI plant and an MPC Controller block.

A model predictive controller monitors all known plant signals, even when it is not in control of the actuators. This monitoring improves its state estimates and allows a bumpless transfer to automatic operation.

Open Simulink Model

Open the Simulink model.

```
open_system('mpc_bumpless')
```



To simulate switching between manual and automatic operation, the Switching block sends either 1 or 0 to control a switch. When it sends 0, the system is in automatic mode, and the output from the MPC Controller block goes to the plant. Otherwise, the system is in manual mode, and the signal from the Operator Commands block goes to the plant.

In both cases, the actual plant input feeds back to the controller `ext.mv` 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, which reduces computational effort.

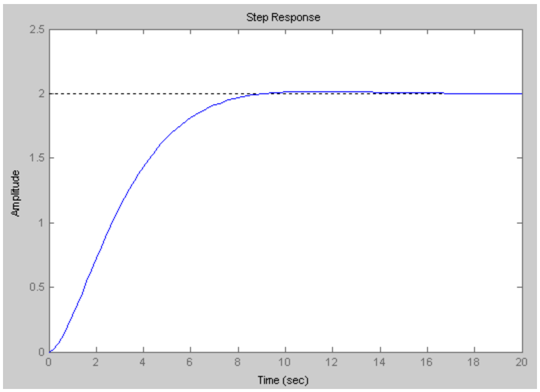
Define Plant and MPC Controller

Create the plant model.

```
num = [1 1];
den = [1 3 2 0.5];
sys = tf(num,den);
```

The plant is a stable single-input single-output system as seen in its step response.

```
step(sys)
```



Create an MPC controller.

```
Ts = 0.5;     % sampling time (seconds)
p = 15;       % prediction horizon
```

```
m = 2;          % control horizon
MPC1 = mpc(sys,Ts,p,m);
MPC1.Weights.Output = 0.01;
MPC1.MV = struct('Min',-1,'Max',1);
Tstop = 250;
```

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.

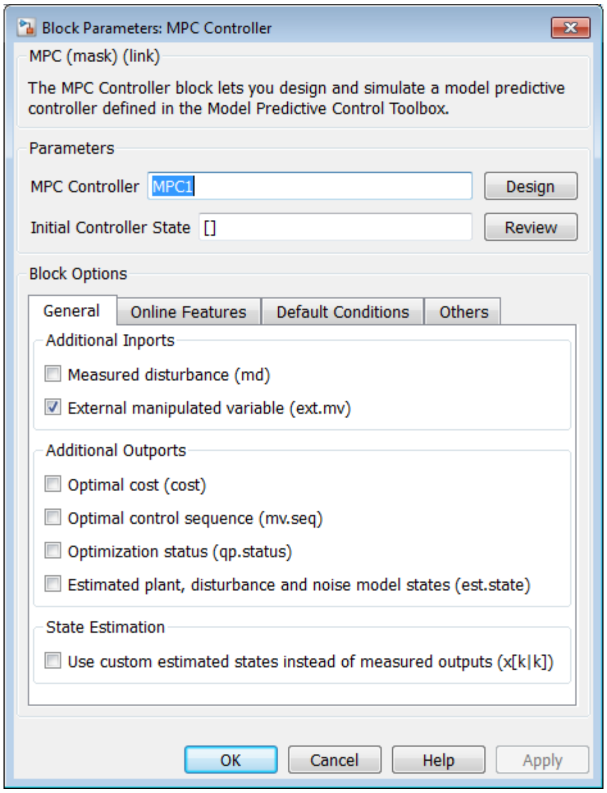
Configure MPC Block Settings

Open the Block Parameters dialog box for the MPC Controller block.

- In the **MPC Controller** box, specify `MPC1`.
- (optional) In the **Initial Controller State** box, specify the initial conditions for the controller. This step is not necessary if the controller and plant already have the same initial state, as is the case for this example. To specify initial conditions, first 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, specify `stateobj` in the **Initial Controller State** box.
- Verify that the **External Manipulated Variable (ext.mv)** option in the **General** tab is selected. This option adds the `ext.mv` inport to the block to enable the use of external manipulated variables.
- Verify that the **Use external signal to enable or disable optimization (switch)** option in the **Others** tab is selected. This option adds the `switch` inport to the controller block to enable switching off the optimization calculations.



Block Parameters: MPC Controller

MPC (mask) (link)

The MPC Controller block lets you design and simulate a model predictive controller defined in the Model Predictive Control Toolbox.

Parameters

MPC Controller: MPC1

Initial Controller State: []

Block Options

General | Online Features | Default Conditions | Others

Additional Inports

- [ ] Measured disturbance (md)
- [✓] External manipulated variable (ext.mv)

Additional Outports

- [ ] Optimal cost (cost)
- [ ] Optimal control sequence (mv.seq)
- [ ] Optimization status (qp.status)
- [ ] Estimated plant, disturbance and noise model states (est.state)

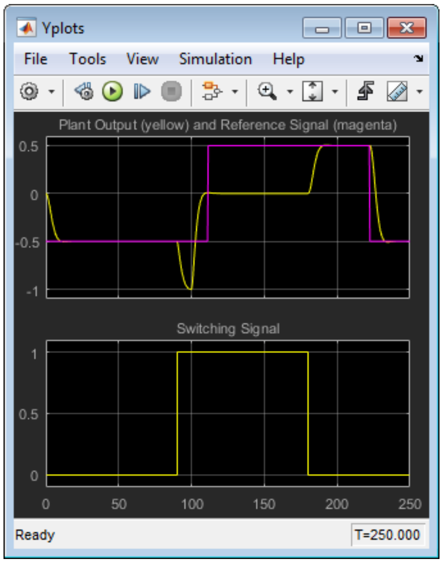
State Estimation

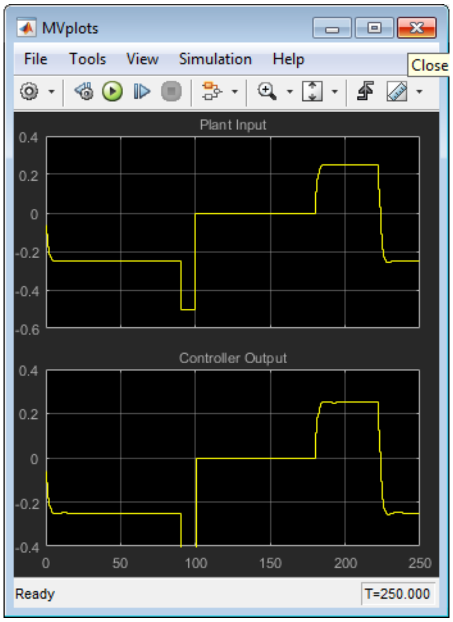
- [ ] Use custom estimated states instead of measured outputs (x[k|k])

Click **OK**.

Examine Switching Between Manual and Automatic Operation

Click **Run** in the Simulink model window to simulate the model.


Yplots
File Tools View Simulation Help
Plant Output (yellow) and Reference Signal (magenta)
0.5
0
-0.5
-1
Switching Signal
1
0.5
0
0
50
100
150
200
250
Ready
T=250.000


MVplots
File Tools View Simulation Help
Close
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
T=250.000

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

Delete the signals entering the `ext.mv` and `switch` inports of the controller block.

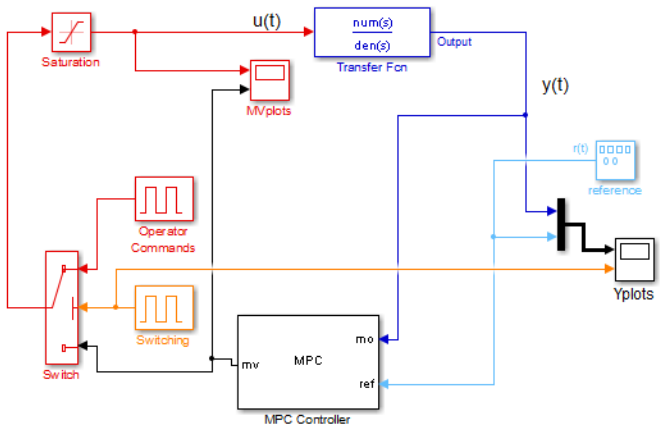
Delete the Unit Delay block and the signal line entering its inport.

Open the Function Block Parameters: MPC Controller dialog box.

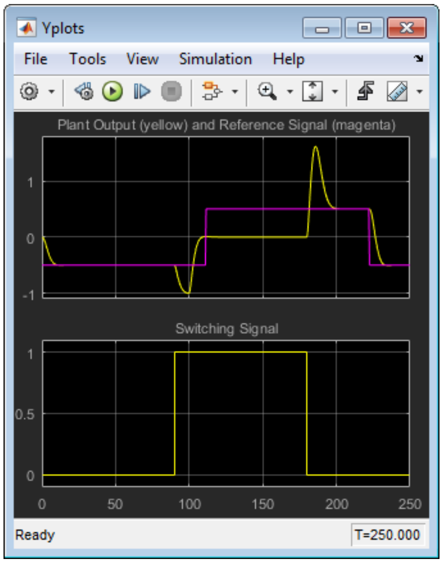
Deselect the **External Manipulated Variable (ext.mv)** option in the **General** tab to remove the `ext.mv` inport from the controller block.

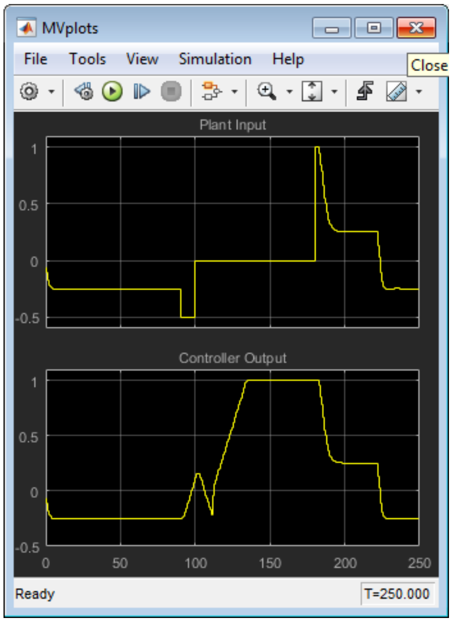
Deselect the **Use external signal to enable or disable optimization (switch)** option in the **Others** tab to remove the `switch` inport from the controller block.

Click **OK**. The Simulink model now resembles the following figure.



Click **Run** to simulate the model.


Yplots
File Tools View Simulation Help
Plant Output (yellow) and Reference Signal (magenta)
1
0
-1
Switching Signal
1
0.5
0
0 50 100 150 200 250
Ready
T=250.000



The behavior is identical to the original case for the first 90 time units.

When the system switches to manual mode at time 90, the plant behavior is the same as before. However, the controller tries to hold the plant at the setpoint. So, its output increases and eventually saturates, as seen in `Controller Output`. Since the controller assumes that this output is going to the plant, its state estimates become inaccurate. Therefore, when the system switches back to automatic mode at time 180, there is a large bump in the `Plant Output`.

Such a bump creates large actuator movements within the plant. By smoothly transferring from manual to automatic operation, a model predictive controller eliminates such undesired movements.

See Also

Blocks
MPC Controller

Use Custom Constraints in Blending Process

This example shows how to design an MPC controller for a blending process using custom input and output constraints.

Blending Process

A continuous blending process combines three feeds in a well-mixed container to produce a blend having desired properties. The dimensionless governing equations are:

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

where

- V is the mixture inventory (in the container).
- ϕ_i is the plow rate for the ith feed.
- ϕ is the rate at which the blend is being removed from inventory, that is the demand.
- γ_{ij} is the concentration of constituent j in feed i.
- γ_j is the concentration of constituent j in the blend.
- τ is time.

In this example, there are two important constituents, j = 1 and 2.

The control objectives are targets for the two constituent concentrations in the blend, and the mixture inventory. The challenge is that the demand, ϕ, and feed compositions, γ_{ij}, vary. The inventory, blend compositions, and demand are measured, but the feed compositions are unmeasured.

At the nominal operating condition:

- Feed 1, ϕ_1, (mostly constituent 1) is 80% of the total inflow.
- Feed 2, ϕ_2, (mostly constituent 2) is 20%.
- Feed 3, ϕ_3, (pure constituent 1) is not used.

The process design allows manipulation of the total feed entering the mixing chamber, ϕ_T, and the individual rates of feeds 2 and 3. In other words, the rate of feed 1 is:

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

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.

```
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 defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

The outputs are the inventory, `y(1)`, and the constituent concentrations, `y(2)` and `y(3)`. Specify nominal values of unity after normalization for all outputs.

```
mpcobj.Model.Nominal.Y = [1 1 1];
```

Specify the normalized nominal values the manipulated variables, `u(1)`, `u(2)` and `u(3)`, and the measured disturbance, `u(4)`.

```
mpcobj.Model.Nominal.U = [1 fin_nom(2) fin_nom(3) 1];
```

Specify output tuning weights. Larger weights are assigned to the first two outputs because we want to pay more attention to controlling the inventory, and the composition of the first constituent.

```
mpcobj.Weights.OV = [1 1 0.5];
```

Specify the hard bounds (physical limits) on the manipulated variables.

```
umin = [0 0 0];
umax = [2 0.6 0.6];
for i = 1:3
   mpcobj.MV(i).Min = umin(i);
   mpcobj.MV(i).Max = umax(i);
   mpcobj.MV(i).RateMin = -0.1;
   mpcobj.MV(i).RateMax =  0.1;
end
```

The total feed rate and the rates of feed 2 and feed 3 have upper bounds. Feed 1 also has an upper bound, determined by the upstream unit supplying it.

Specify Custom Constraints

Given the specified upper bounds on the feed 2 and 3 rates (0.6), it is possible that their sum could be as much as 1.2. Since the nominal total feed rate is 1.0, the controller can request a physically impossible condition, where the sum of feeds 2 and 3 exceeds the total feed rate, which implies a negative feed 1 rate.

The following constraint prevents the controller from requesting an unrealistic ϕ_1 value.

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

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

```
E = [-1 1 1; 1 -1 -1];
g = [0;0.8];
```

Since no outputs are specified in the mixed constraints, set their coefficients to zero.

```
F = zeros(2,3);
```

Specify that both constraints are hard (ECR = 0).

```
v = zeros(2,1);
```

Specify zero coefficients for the measured disturbance.

```
h = zeros(2,1);
```

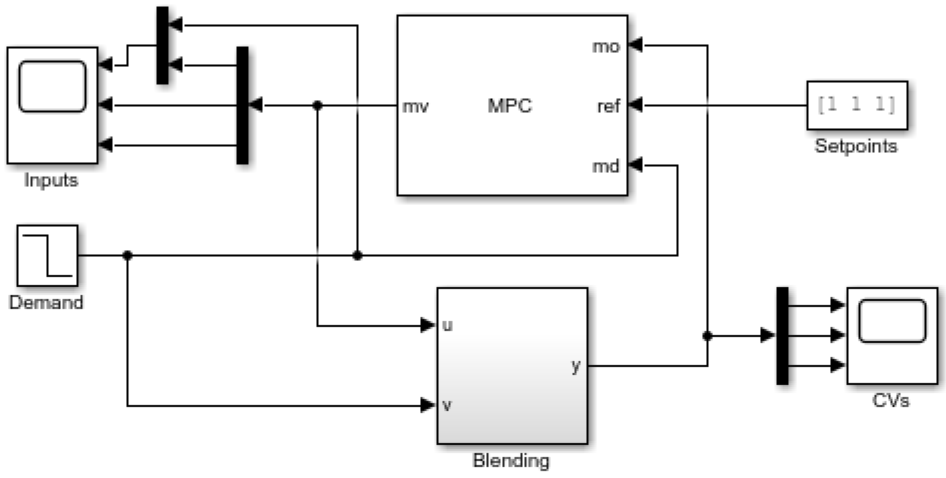
Set the custom constraints in the MPC controller.

```
setconstraint(mpcobj,E,F,g,v,h)
```

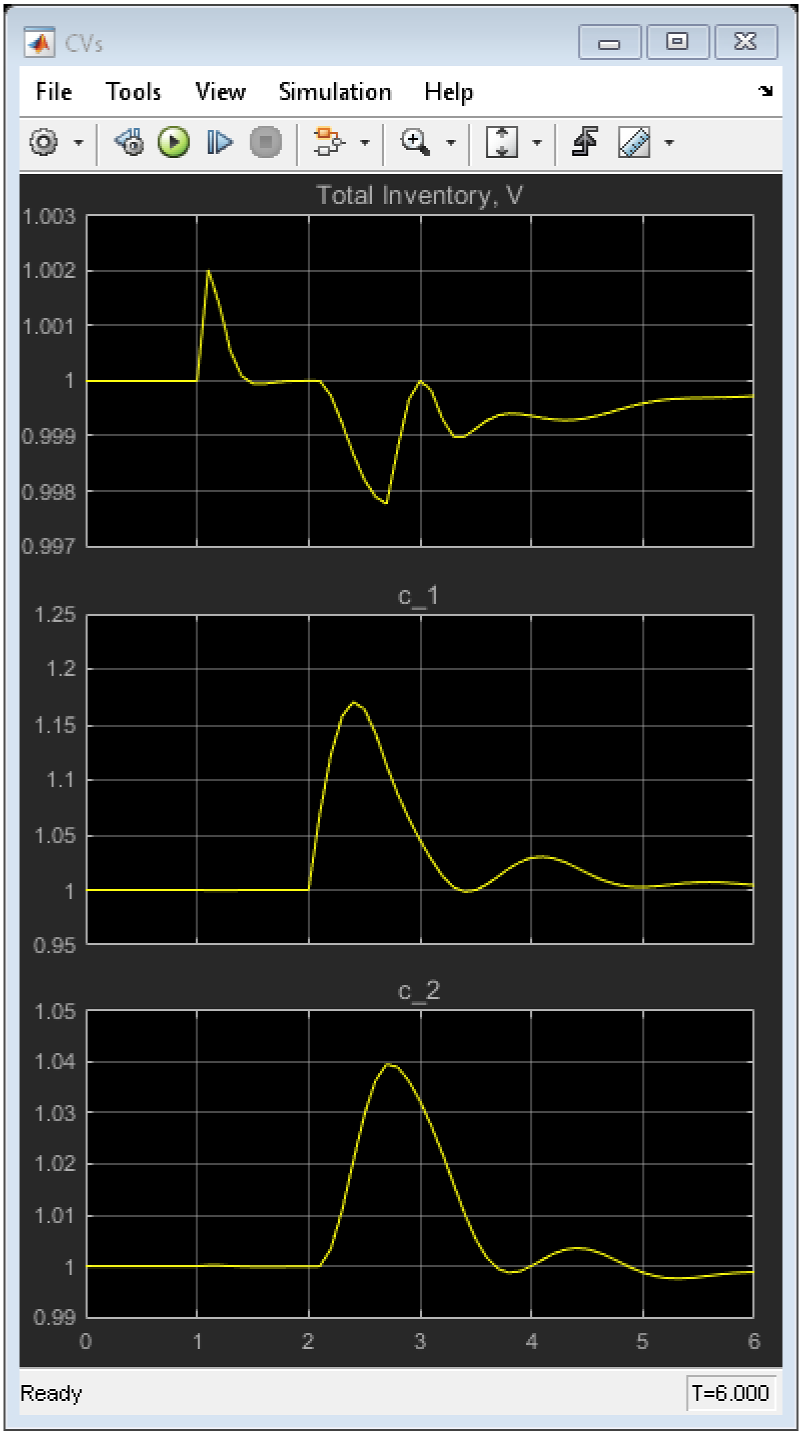
Open and Simulate Model in Simulink

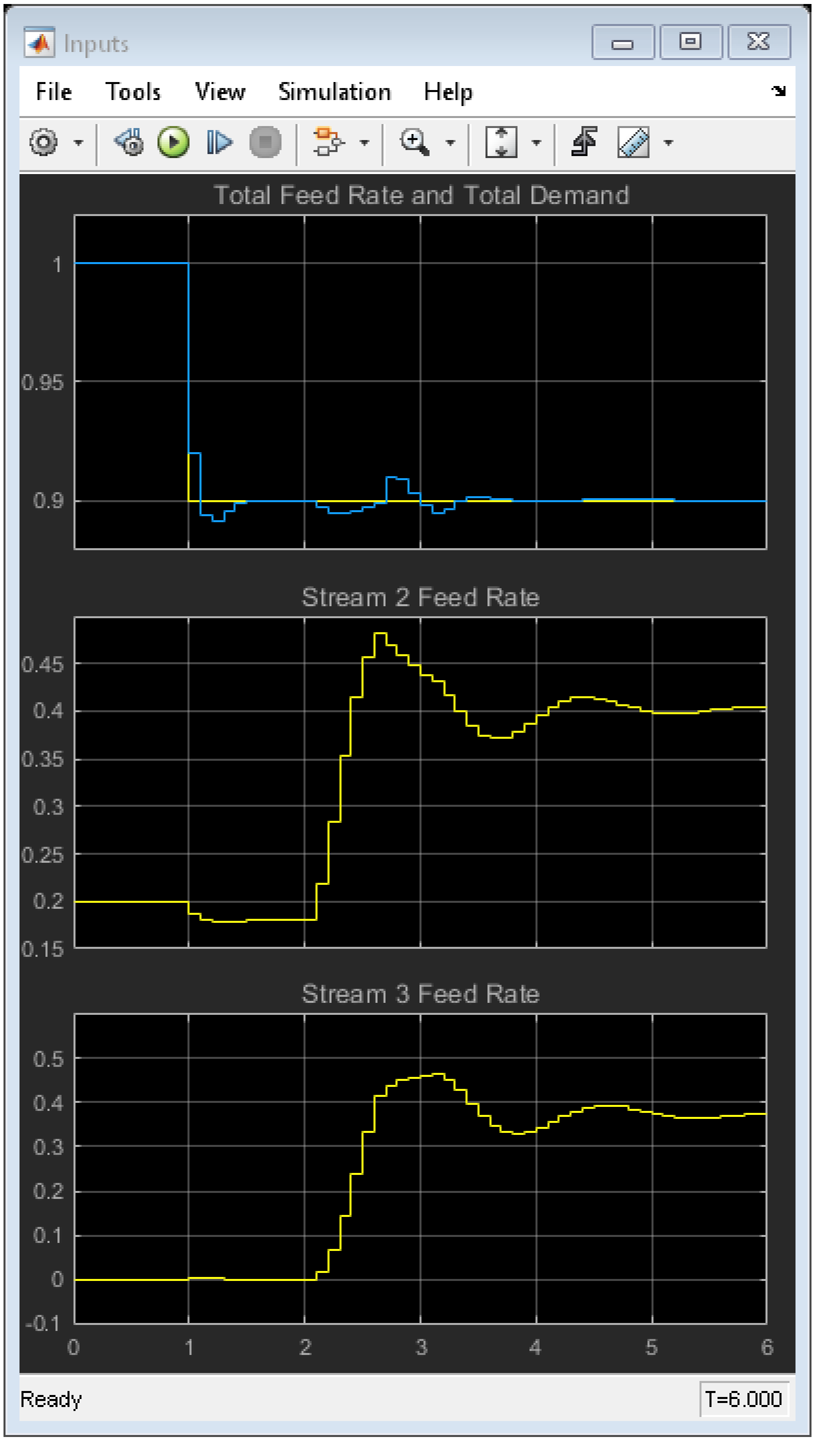
```
sys = 'mpc_blendingprocess';
open_system(sys)
sim(sys)
```

```
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



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
T=6.000


Inputs
File
Tools
View
Simulation
Help
Total Feed Rate and Total Demand
1
0.95
0.9
Stream 2 Feed Rate
0.45
0.4
0.35
0.3
0.25
0.2
0.15
Stream 3 Feed Rate
0.5
0.4
0.3
0.2
0.1
0
-0.1
0
1
2
3
4
5
6
Ready
T=6.000

The MPC controller controls the blending process. The block labeled `Blending` incorporates the previously described model equations and includes an unmeasured step disturbance in the constituent 1 feed composition.

The `Demand`, ϕ, is modeled as a measured disturbance. The operator can vary the demand value, and the resulting signal goes to both the process and the controller.

The model simulates the following scenario:

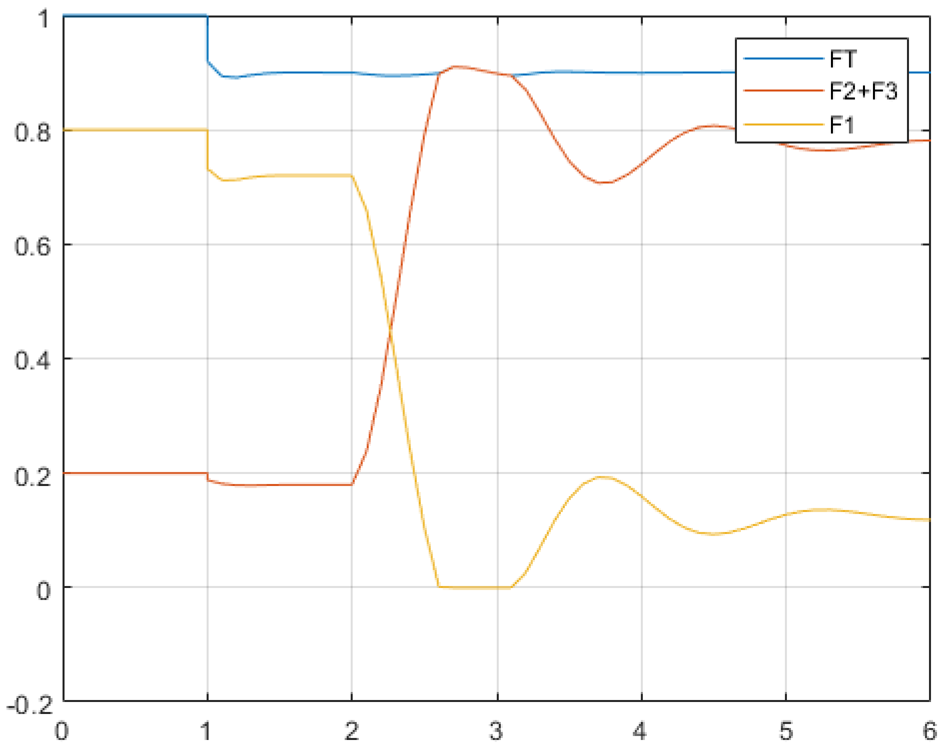
- At $\tau = 0$, the process is operating at steady state.
- At $\tau = 1$, the `Total Demand` decreases from $\phi = 1.0$ to $\phi = 0.9$.
- At $\tau = 2$, there is a large step increase in the concentration of constituent 1 in feed 1, from 1.17 to 2.17.

The controller maintains the inventory very close to its setpoint, but the severe disturbance in the feed composition causes a prediction error and a large disturbance in the blend composition, especially for constituent 1, `c_1`. However, the controller recovers and drives the blend composition back to its setpoint.

Verify Effect of Custom Constraints

Plot the feed rate signals.

```
figure
plot(MVs.time,[MVs.signals(1).values(:,2), ...
    (MVs.signals(2).values + MVs.signals(3).values), ...
    (MVs.signals(1).values(:,2)-MVs.signals(2).values-MVs.signals(3).values)])
grid
legend('FT','F2+F3','F1')
```



The total feed rate, FT, and the sum of feed rates F2 and F3 coincide for $1.7 \leq \tau \leq 2.2$. If the custom input constraints had not been included, the controller would have requested an impossible negative feed 1 rate, F1, during this period.

```
bdclose(sys)
```

See Also

`setconstraint`

More About

- "Constraints on Linear Combinations of Inputs and Outputs" on page 2-32
- "Using Custom Input and Output Constraints"
- "Nonlinear Blending Process with Custom Constraints"

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 using terminal penalty weights [1]. The standard MPC cost function on page 2-2 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 sampling instant. In the standard MPC cost function, the horizon length is *p*, and the manipulated variable changes *m* times, where *m* is the control horizon.

The two cost functions are equivalent if the MPC cost function is:

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

where Q_p is a 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 Ricatti matrix calculated using the `lqr` and `lqry` commands.

Design an MPC Controller Equivalent to LQR Controller

This example shows shows how to design an unconstrained MPC controller that provides performance equivalent to an LQR controller.

Define the Plant Model

The plant is a double integrator, represented as a state-space model in discrete time with a sample time of 0.1 seconds. In this case the two plant states are measurable at the plant outputs.

```
A = [1 0;0.1 1];
B = [0.1;0.005];
C = eye(2);
D = zeros(2,1);
Ts = 0.1;
Plant = ss(A,B,C,D,Ts);
Plant.InputName = {'u'};
Plant.OutputName = {'x_1','x_2'};
```

Design an LQR Controller

Design an LQR controller with output feedback for the plant.

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

`Q` and `R` are output and input weight matrices, respectively. `Qp` is the Ricatti matrix.

Design an Equivalent MPC Controller

To implement the MPC cost function, compute L, the Cholesky decomposition of Q_p, such that $L^T L = Q_p$. Then define auxilliary unmeasured output variables $y_a(k) = Lx(k)$, such that $y_a^T y_a = x^T Q_p x$.

```
NewPlant = Plant;
L = chol(Qp);
set(NewPlant,'C',[C;L],'D',[D;zeros(2,1)],...
    'OutputName',{'x_1','x_2','Cx_1','Cx_2'})
NewPlant.InputGroup.MV = 1;
NewPlant.OutputGroup.MO = [1 2];
NewPlant.OutputGroup.UO = [3 4];
```

Create an MPC controller with equal prediction and control horizons.

```
P = 3;
M = 3;
MPCobj = mpc(NewPlant,Ts,P,M);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Specify weights for the manipulated variable and output variables for the first $p-1$ prediction horizon steps. Use the square roots of the diagonal elements of the Q and R weight matrices from the LQR controller design. The standard Q weight matrix values apply to y, and y_a has a zero penalty.

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

Specify terminal weights for the final prediction horizon step. On step p, the original y has a zero penalty, and y_a has a unit penalty. The input weight remains the same for the terminal step.

```
U = struct('Weight',uwt);
Y = struct('Weight',[0 0 1 1]);
setterminal(MPCobj,Y,U)
```

Remove the default state estimator. Since the model states are measured directly, the default state estimator is unnecessary.

```
setoutdist(MPCobj,'model',tf(zeros(4,1)))
setEstimator(MPCobj,[],C)
```

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.

```
SimOptions = mpcsimopt(MPCobj);
SimOptions.PlantInitialState = x0;
r = zeros(1,4);
[y,t,u] = sim(MPCobj,ceil(Tstop/Ts),r,SimOptions);
```

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

```
Cost = sum(sum(y(:,1:2)*diag(ywt).*y(:,1:2))) + sum(u*diag(uwt).*u);
```

Compute the closed-loop response for a standard MPC controller with default prediction and control horizons ($p = 10$, $m = 3$). To match the other controllers, remove the default state estimator from the standard MPC controller.

```
MPCobjSTD = mpc(Plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
   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)
SimOptions = mpcsimopt(MPCobjSTD);
SimOptions.PlantInitialState = x0;
r = zeros(1,2);
[ySTD,tSTD,uSTD] = sim(MPCobjSTD,ceil(Tstop/Ts),r,SimOptions);
```

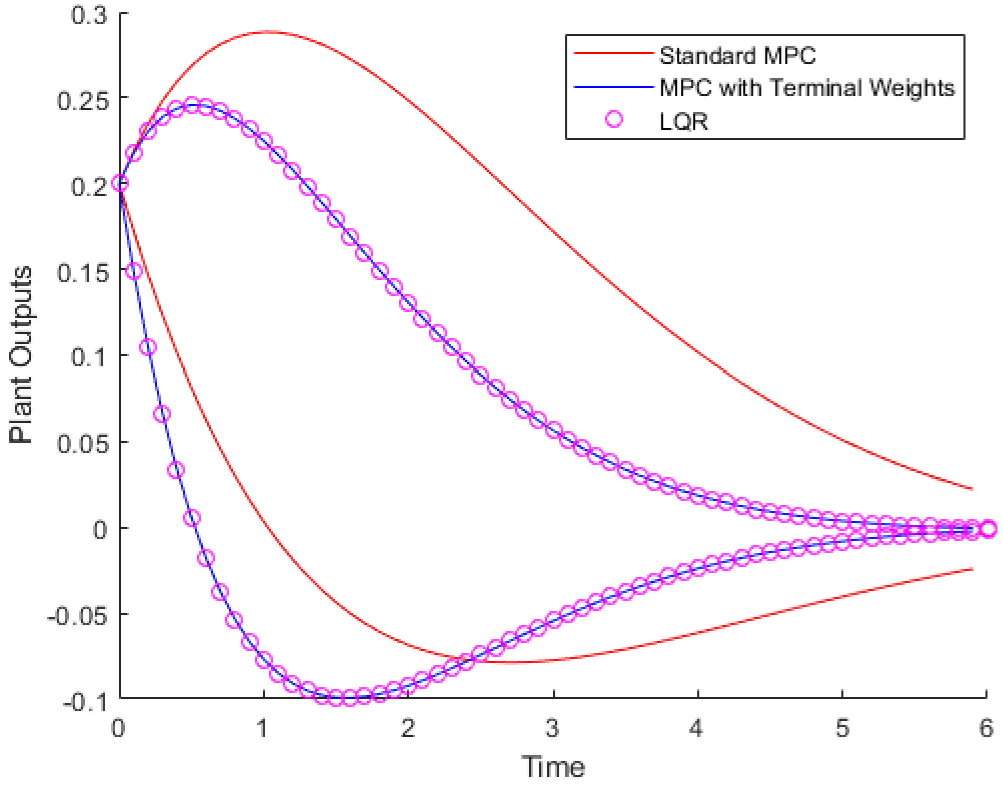
```
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

```
CostSTD = sum(sum(ySTD*diag(ywt).*ySTD)) + sum(uSTD*uwt.*uSTD);
```

Compare the controller responses.

```
figure
h1 = line(tSTD,ySTD,'color','r');
h2 = line(t,y(:,1:2),'color','b');
```

```
h3 = line(tLQR,yLQR,'color','m','marker','o','linestyle','none');
xlabel('Time')
ylabel('Plant Outputs')
legend([h1(1) h2(1) h3(1)],'Standard MPC','MPC with Terminal Weights','LQR','Location'
```



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.

As reported in [1], the computed `Cost` value of `2.23` for the MPC controller with terminal weights is identical to the LQR controller cost. The cost for the standard MPC controller, `CostSTD`, is `4.82`, more than double the value of `Cost`.

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 Rawlings and Mayne [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 David Q. Mayne “Model Predictive Control: Theory and Design” Nob Hill Publishing, 2010.

See Also

`setterminal`

More About

- “Optimization Problem” on page 2-2
- “Terminal Weights and Constraints” on page 2-29
- “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”

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 a manipulated variable.

Define Plant Model

The linear plant model has two inputs and two outputs.

```
N1 = [3 1];
D1 = [1 2*.3 1];
N2 = [2 1];
D2 = [1 2*.5 1];
plant = ss(tf({N1,N2},{D1,D2}));
A = plant.A;
B = plant.B;
C = plant.C;
D = plant.D;
x0 = [0 0 0 0]';
```

Design MPC Controller

Create MPC controller.

```
Ts = 0.4;                         % Sample time
mpcobj = mpc(plant,Ts,20,5);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify weights.

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

Define input specifications.

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

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

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

Simulation Using Simulink®

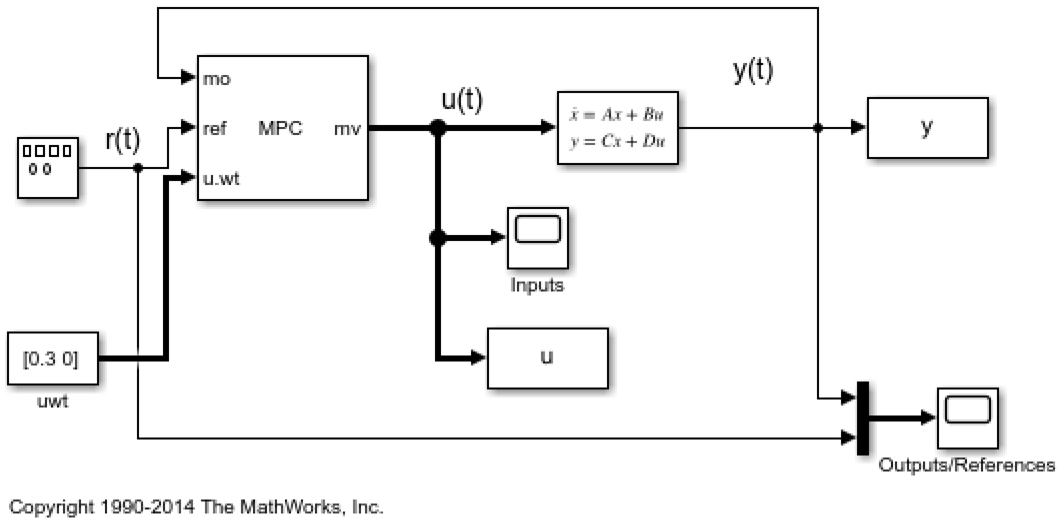
To run this example, Simulink® is required.

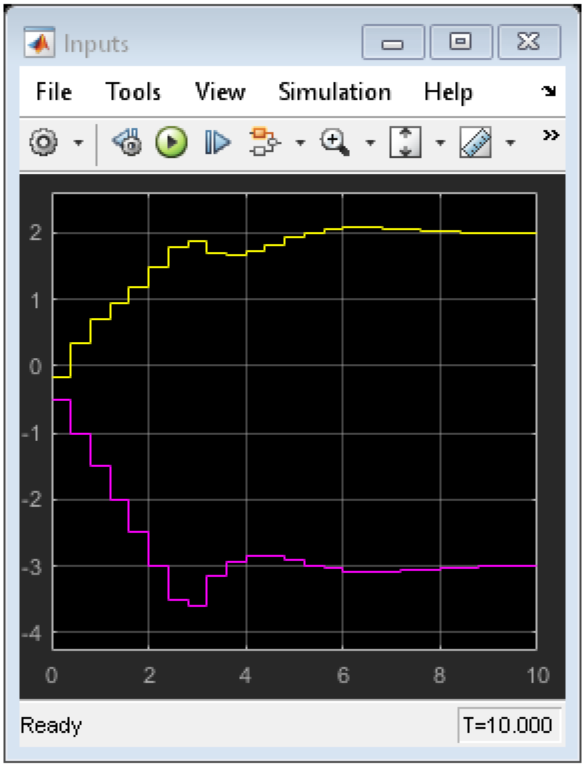
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

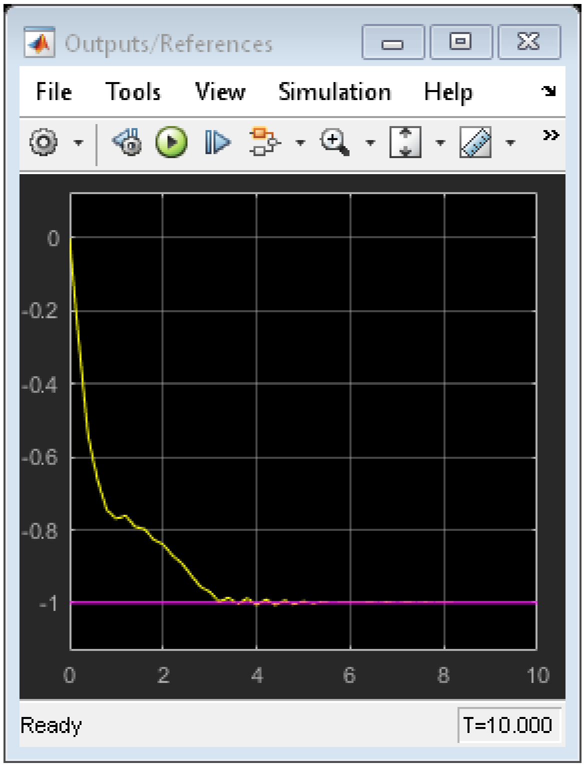
Simulate.

```
mdl = 'mpc_utarget';
open_system(mdl)      % Open Simulink(R) Model
sim(mdl);             % Start Simulation
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```




Inputs
File Tools View Simulation Help
2
1
0
-1
-2
-3
-4
0 2 4 6 8 10
Ready
T=10.000


Outputs/References
File
Tools
View
Simulation
Help
0
-0.2
-0.4
-0.6
-0.8
-1
0
2
4
6
8
10
Ready
T=10.000

```
bdclose(mdl)
```

Specifying Alternative Cost Function with Off-Diagonal Weight Matrices

This example shows how to use non-diagonal weight matrices in a model predictive controller.

Define Plant Model and MPC Controller

The linear plant model has two inputs and two outputs.

```
plant = ss(tf({1,1;1,2},{[1 .5 1],[.7 .5 1];[1 .4 2],[1 2]}));
[A,B,C,D] = ssdata(plant);
Ts = 0.1;                % sampling time
plant = c2d(plant,Ts);  % convert to discrete time
```

Create MPC controller.

```
p=20;       % prediction horizon
m=2;        % control horizon
mpcobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define constraints on the manipulated variable.

```
mpcobj.MV = struct('Min',{-3;-2},'Max',{3;2},'RateMin',{-100;-100},'RateMax',{100;100}
```

Define non-diagonal output weight. Note that it is specified inside a cell array.

```
OW = [1 -1]'*[1 -1];
% Non-diagonal output weight, corresponding to ((y1-r1)-(y2-r2))^2
mpcobj.Weights.OutputVariables = {OW};
% Non-diagonal input weight, corresponding to (u1-u2)^2
mpcobj.Weights.ManipulatedVariables = {0.5*OW};
```

Simulate Using SIM Command

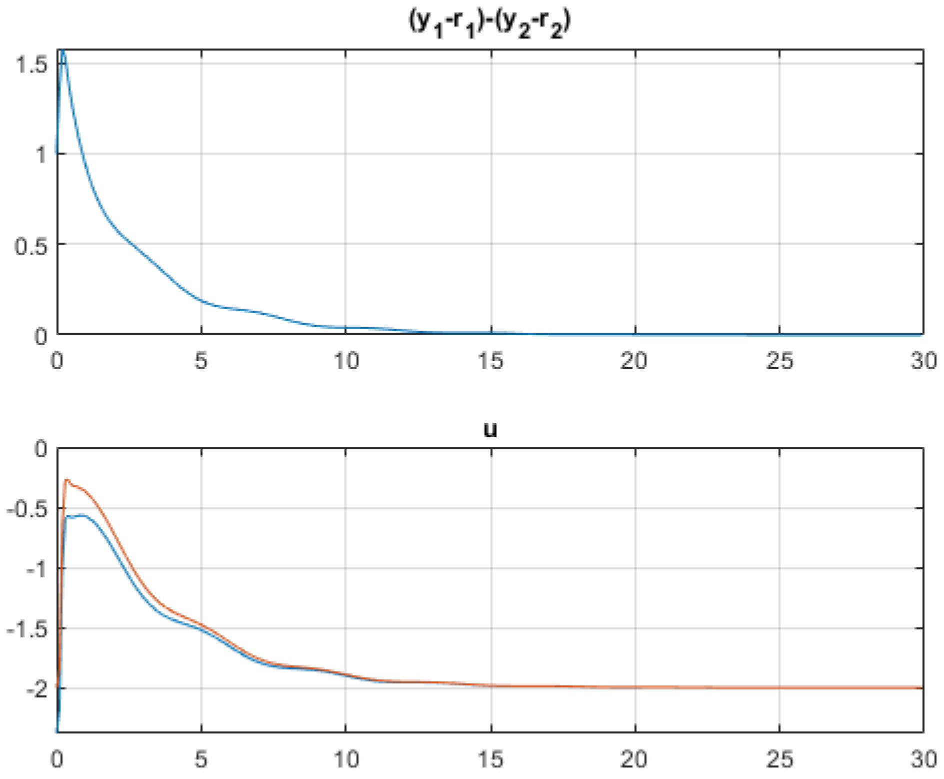
Specify simulation options.

```
Tstop = 30;               % simulation time
Tf = round(Tstop/Ts);     % number of simulation steps
r = ones(Tf,1)*[1 2];     % reference trajectory
```

Run the closed-loop simulation and plot results.

```
[y,t,u] = sim(mpcobj,Tf,r);
subplot(211)
plot(t,y(:,1)-r(1,1)-y(:,2)+r(1,2));grid
title('(y_1-r_1)-(y_2-r_2)');
subplot(212)
plot(t,u);grid
title('u');
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



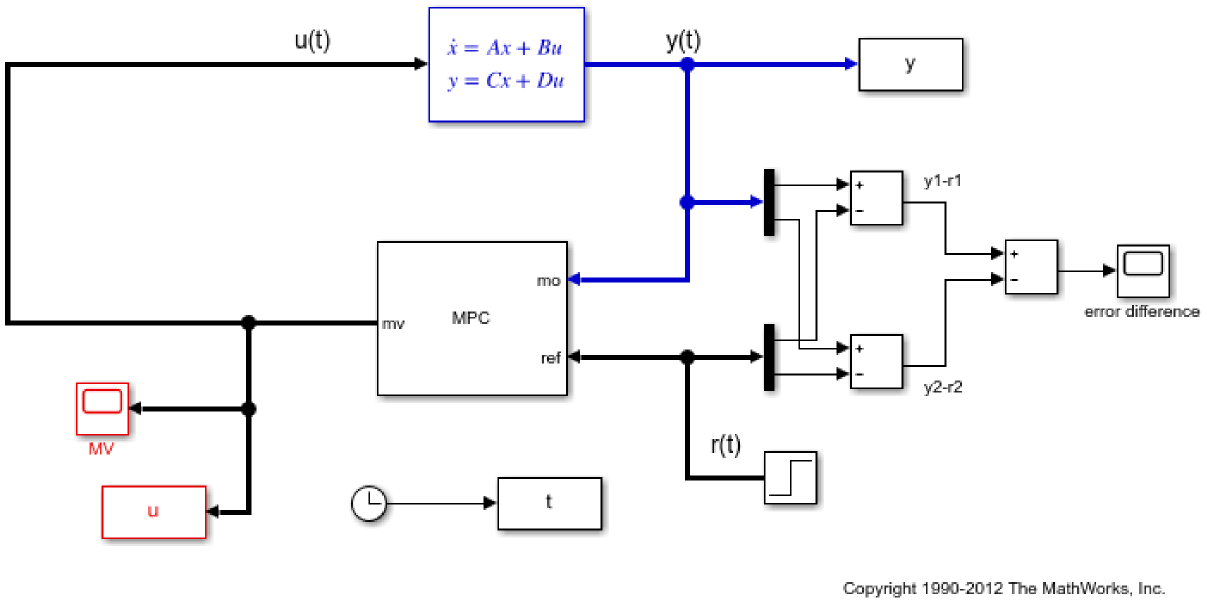
Simulate Using Simulink®

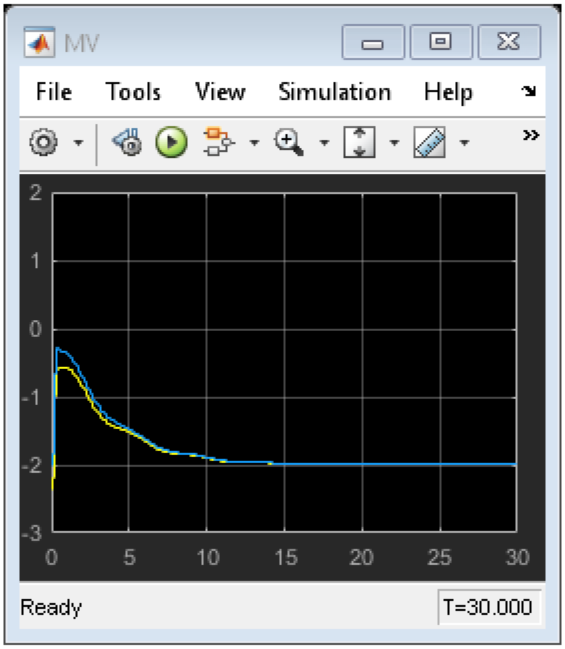
To run this example, Simulink® is required.

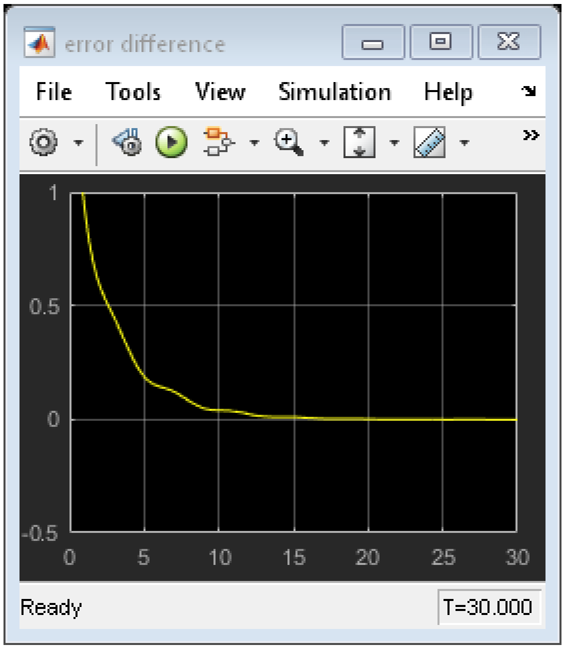
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this part of the example.')
    return
end
```

Now simulate closed-loop MPC in Simulink®.

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




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


error difference
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
0
5
10
15
20
25
30
Ready
T=30.000

```
bdclose(mdl);
```

Review Model Predictive Controller for Stability and Robustness Issues

This example shows how to use the `review` command to detect potential issues with a model predictive controller design.

Fuel Gas Blending Process

The example application is a fuel gas blending process. The objective is to blend six gases to obtain a fuel gas, which is then burned to provide process heating. The fuel gas must satisfy three quality standards in order for it to burn reliably and with the expected heat output. The fuel gas header pressure must also be controlled. Thus, there are four controlled output variables. The manipulated variables are the six feed gas flow rates.

Inputs:

```
1. Natural Gas (NG)
2. Reformed Gas (RG)
3. Hydrogen (H2)
4. Nitrogen (N2)
5. Tail Gas 1 (T1)
6. Tail Gas 2 (T2)
```

Outputs:

```
1. High Heating Value (HHV)
2. Wobbe Index (WI)
3. Flame Speed Index (FSI)
4. Header Pressure (P)
```

The fuel gas blending process was studied by Muller et al.: "Modeling, validation, and control of an industrial fuel gas blending system", C.J. Muller, I.K. Craig, N.L. Ricker, J. of Process Control, in press, 2011.

Linear Plant Model

Use the following linear plant model as the prediction model for the controller. This state-space model, applicable at a typical steady-state operating point, uses the time unit of hours.

```
a = diag([-28.6120, -28.6822, -28.5134,  -0.0281, -23.2191, -23.4266, ...
          -22.9377, - 0.0101, -26.4877, -26.7950, -27.2210,  -0.0083, ...
          -23.0890, -23.0062, -22.9349,  -0.0115, -25.8581, -25.6939, ...
          -27.0793,  -0.0117, -22.8975, -22.8233, -21.1142,  -0.0065]);
b = zeros(24,6);
b( 1: 4,1) = [4, 4, 8, 32]';
b( 5: 8,2) = [2, 2, 4, 32]';
b( 9:12,3) = [2, 2, 4, 32]';
b(13:16,4) = [4, 4, 8, 32]';
b(17:20,5) = [2, 2, 4, 32]';
b(21:24,6) = [1, 2, 1, 32]';
c = [diag([ 6.1510,  7.6785, -5.9312, 34.2689]), ...
     diag([-2.2158, -3.1204,  2.6220, 35.3561]), ...
     diag([-2.5223,  1.1480,  7.8136, 35.0376]), ...
     diag([-3.3187, -7.6067, -6.2755, 34.8720]), ...
     diag([-1.6583, -2.0249,  2.5584, 34.7881]), ...
     diag([-1.6807, -1.2217,  1.0492, 35.0297])];
d = zeros(4,6);
Plant = ss(a, b, c, d);
```

By default, all the plant inputs are manipulated variables.

```
Plant.InputName = {'NG', 'RG', 'H2', 'N2', 'T1', 'T2'};
```

By default, all the plant outputs are measured outputs.

```
Plant.OutputName = {'HHV', 'WI', 'FSI', 'P'};
```

Transport delay is added to plant outputs to reflect the delay in the sensors.

```
Plant.OutputDelay = [0.00556  0.0167  0.00556  0];
```

Initial Controller Design

Construct an initial model predictive controller based on design requirements.

Specify sampling time, horizons and steady-state values.

The sampling time is that of the sensors (20 seconds). The prediction horizon is approximately equal to the plant settling time (39 intervals). The control horizon uses four blocked moves that have lengths of 2, 6, 12 and 19 intervals respectively. The nominal operating conditions are non-zero. The output measurement noise is white noise with magnitude of 0.001.

```
MPC_verbosity = mpcverbosity('off'); % Disable MPC message displaying at command line
Ts = 20/3600;   % Time units are hours.
Obj = mpc(Plant, Ts, 39, [2, 6, 12, 19]);
Obj.Model.Noise = ss(0.001*eye(4));
Obj.Model.Nominal.Y = [16.5, 25, 43.8, 2100];
Obj.Model.Nominal.U = [1.4170, 0, 2, 0, 0, 26.5829];
```

Specify lower and upper bounds on manipulated variables.

Since all the manipulated variables are flow rates of gas streams, their lower bounds are zero. All the MV constraints are hard (`MinECR` and `MaxECR` = 0) by default.

```
MVmin = zeros(1,6);
MVmax = [15, 20, 5, 5, 30, 30];
for i = 1:6
    Obj.MV(i).Min = MVmin(i);
    Obj.MV(i).Max = MVmax(i);
end
```

Specify lower and upper bounds on manipulated variable increments.

The bounds are set large enough to allow full range of movement in one interval. All the MV rate constraints are hard (`RateMinECR` and `RateMaxECR` = 0) by default.

```
for i = 1:6
    Obj.MV(i).RateMin = -MVmax(i);
    Obj.MV(i).RateMax =  MVmax(i);
end
```

Specify lower and upper bounds on plant outputs.

All the OV constraints are soft (MinECR and MaxECR = 0) by default.

```
OVmin = [16.5, 25, 39, 2000];
OVmax = [18.0, 27, 46, 2200];
for i = 1:4
    Obj.OV(i).Min = OVmin(i);
    Obj.OV(i).Max = OVmax(i);
end
```

Specify weights on manipulated variables.

MV weights are specified based on the known costs of each feed stream. This tells MPC controller how to move the six manipulated variables in order to minimize the cost of the blended fuel gas. The weights are normalized so the maximum weight is approximately 1.0.

```
Obj.Weights.MV = [54.9, 20.5, 0, 5.73, 0, 0]/55;
```

Specify weights on manipulated variable increments.

They are small relative to the maximum MV weight so the MVs are free to vary.

```
Obj.Weights.MVrate = 0.1*ones(1,6);
```

Specify weights on plant outputs.

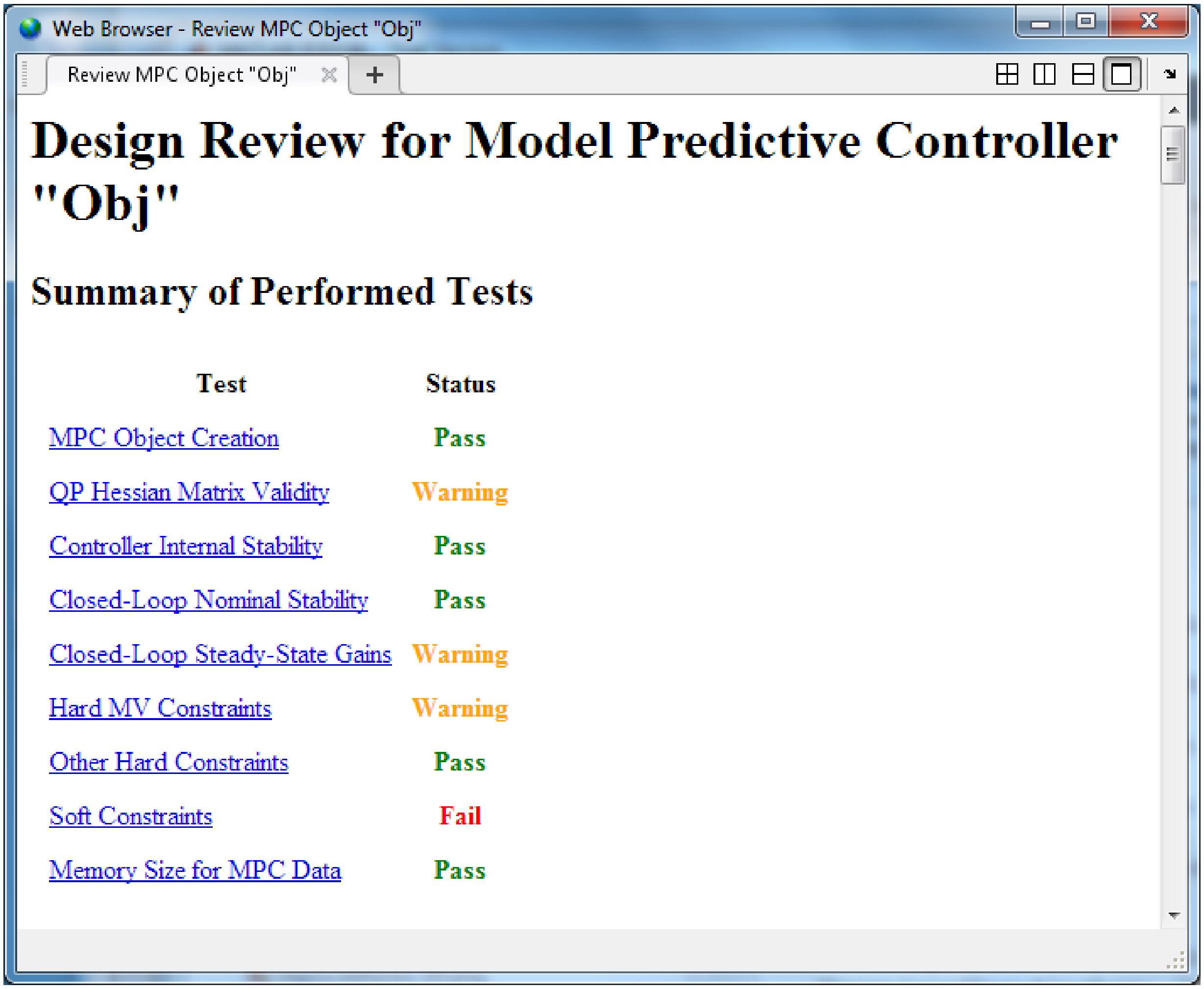
The OV weights penalize deviations from specified setpoints and would normally be "large" relative to the other weights. Let us first consider the default values, which equal the maximum MV weight specified above.

```
Obj.Weights.OV = [1, 1, 1, 1];
```

Improve the Initial Design

Review the initial controller design.

```
review(Obj)
```

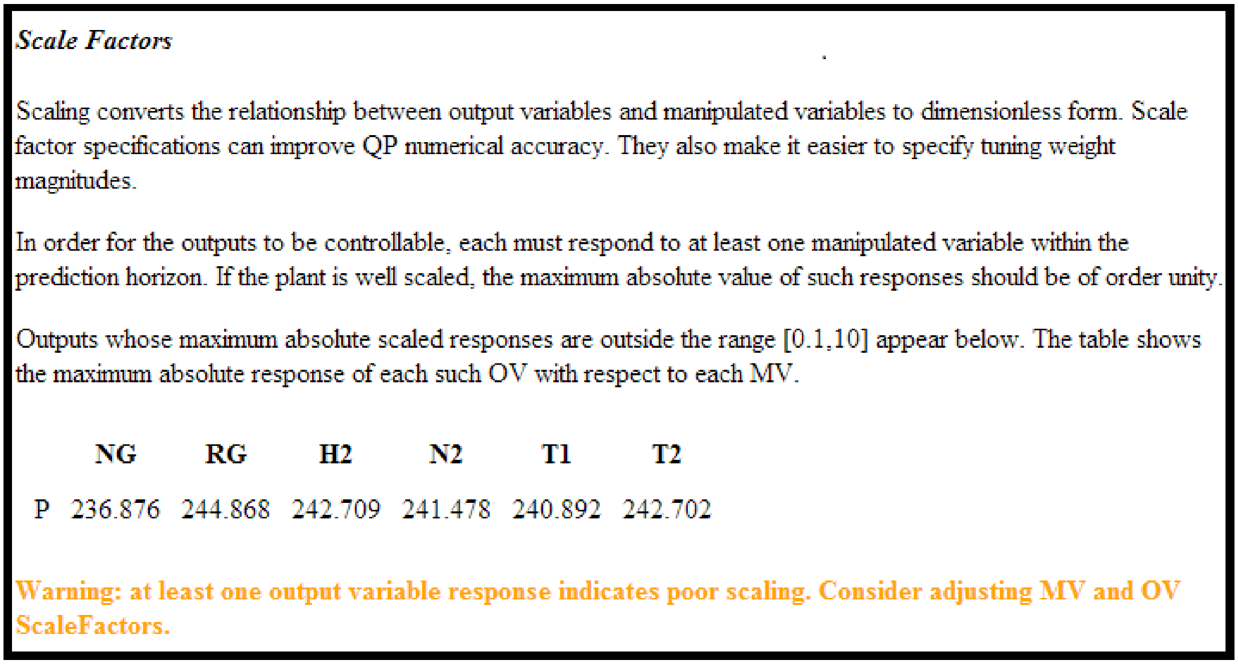


Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

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

The summary table shown above lists three warnings and one error. Consider these in turn. Click **QP Hessian Matrix Validity** and scroll down to display the warning It indicates that the plant signal magnitudes differ significantly. Specifically, the pressure response is much larger than the others.


Scale Factors

Scaling converts the relationship between output variables and manipulated variables to dimensionless form. Scale factor specifications can improve QP numerical accuracy. They also make it easier to specify tuning weight magnitudes.

In order for the outputs to be controllable, each must respond to at least one manipulated variable within the prediction horizon. If the plant is well scaled, the maximum absolute value of such responses should be of order unity.

Outputs whose maximum absolute scaled responses are outside the range [0.1,10] appear below. The table shows the maximum absolute response of each such OV with respect to each MV.

	NG	**RG**	**H2**	**N2**	**T1**	**T2**
P	236.876	244.868	242.709	241.478	240.892	242.702

Warning: at least one output variable response indicates poor scaling. Consider adjusting MV and OV ScaleFactors.

Examination of the specified OV bounds shows that the spans are quite different, and the pressure span is two orders of magnitude larger than the others. It is good practice to specify MPC scale factors to account for the expected differences in signal magnitudes. We are already weighting MVs based on relative cost, so we focus on the OVs only.

Calculate OV spans.

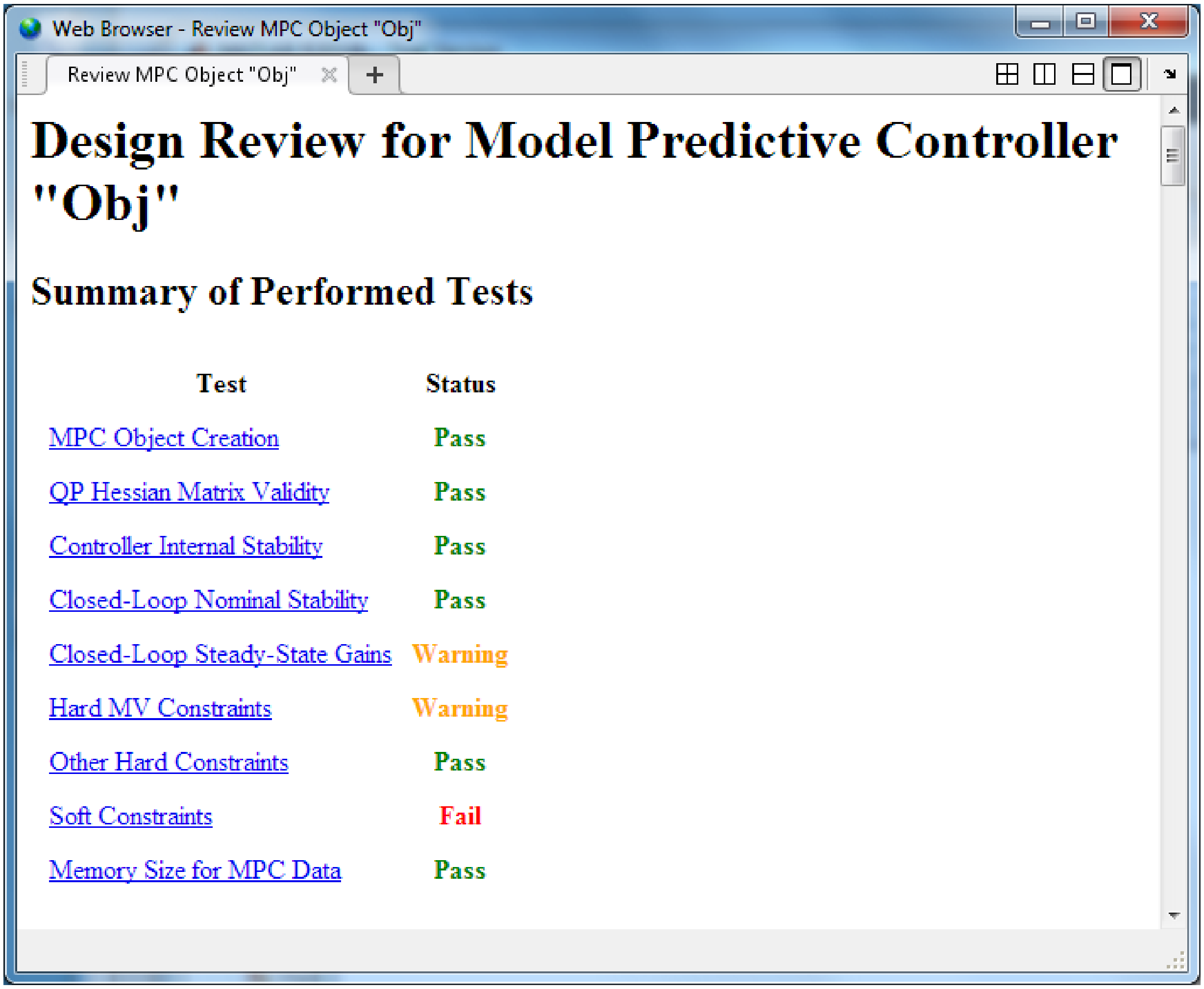
```
OVspan = OVmax - OVmin;
```

Use these as the specified scale factors.

```
for i = 1:4
    Obj.OV(i).ScaleFactor = OVspan(i);
end
```

Verify that the scale factor warning has disappeared.

```
review(Obj)
```

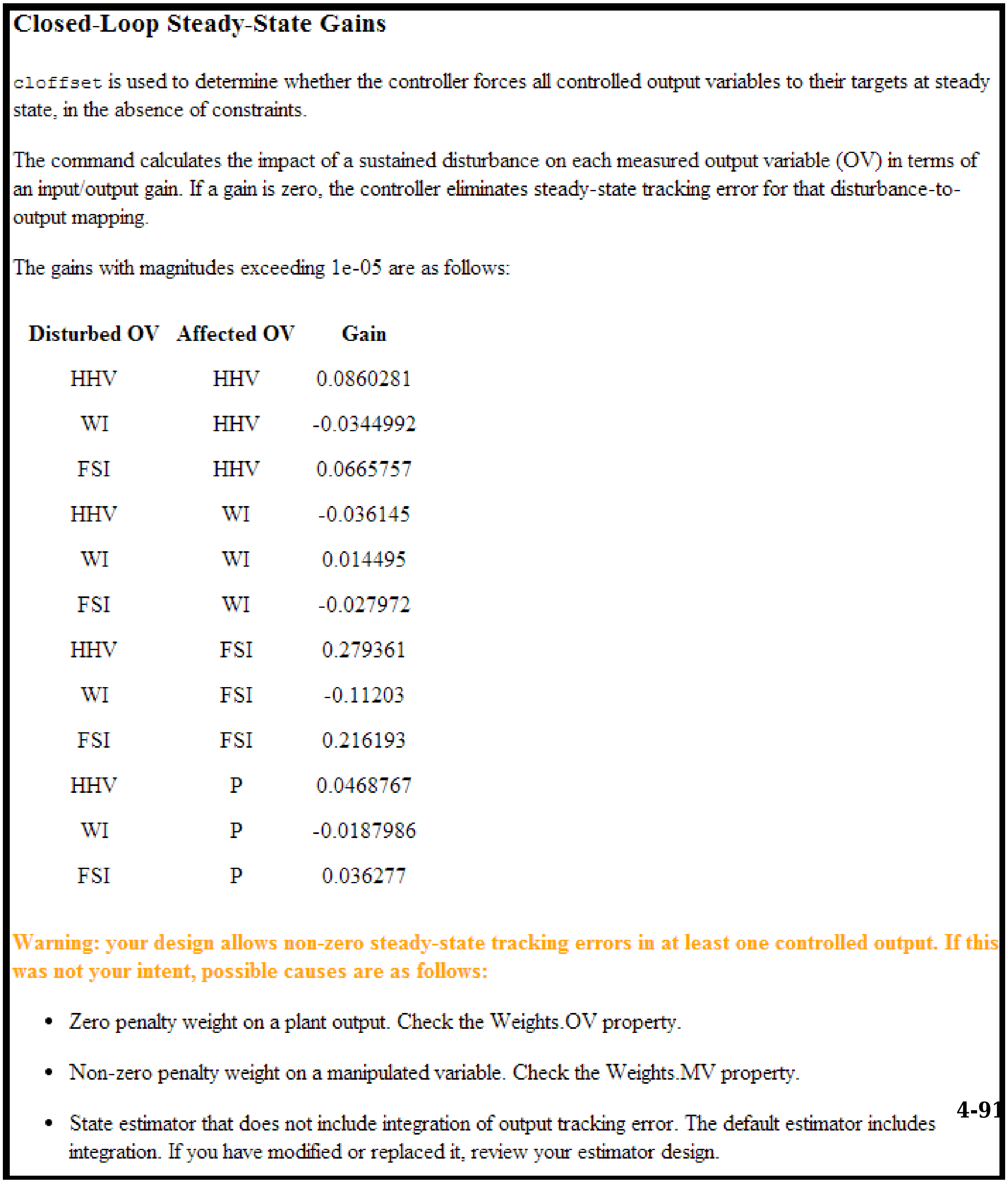


Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Controller Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Warning
Hard MV Constraints	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. Click **Closed-Loop Steady-State Gains** to see a list of the non-zero gains.



Closed-Loop Steady-State Gains

`cloffset` is used to determine whether the controller forces all controlled output variables to their targets at steady state, in the absence of constraints.

The command calculates the impact of a sustained disturbance on each measured output variable (OV) in terms of an input/output gain. If a gain is zero, the controller eliminates steady-state tracking error for that disturbance-to-output mapping.

The gains with magnitudes exceeding 1e-05 are as follows:

Disturbed OV	Affected OV	Gain
HHV	HHV	0.0860281
WI	HHV	-0.0344992
FSI	HHV	0.0665757
HHV	WI	-0.036145
WI	WI	0.014495
FSI	WI	-0.027972
HHV	FSI	0.279361
WI	FSI	-0.11203
FSI	FSI	0.216193
HHV	P	0.0468767
WI	P	-0.0187986
FSI	P	0.036277

Warning: your design allows non-zero steady-state tracking errors in at least one controlled output. If this was not your intent, possible causes are as follows:

- Zero penalty weight on a plant output. Check the Weights.OV property.
- Non-zero penalty weight on a manipulated variable. Check the Weights.MV property.
- State estimator that does not include integration of output tracking error. The default estimator includes integration. If you have modified or replaced it, review your estimator design.

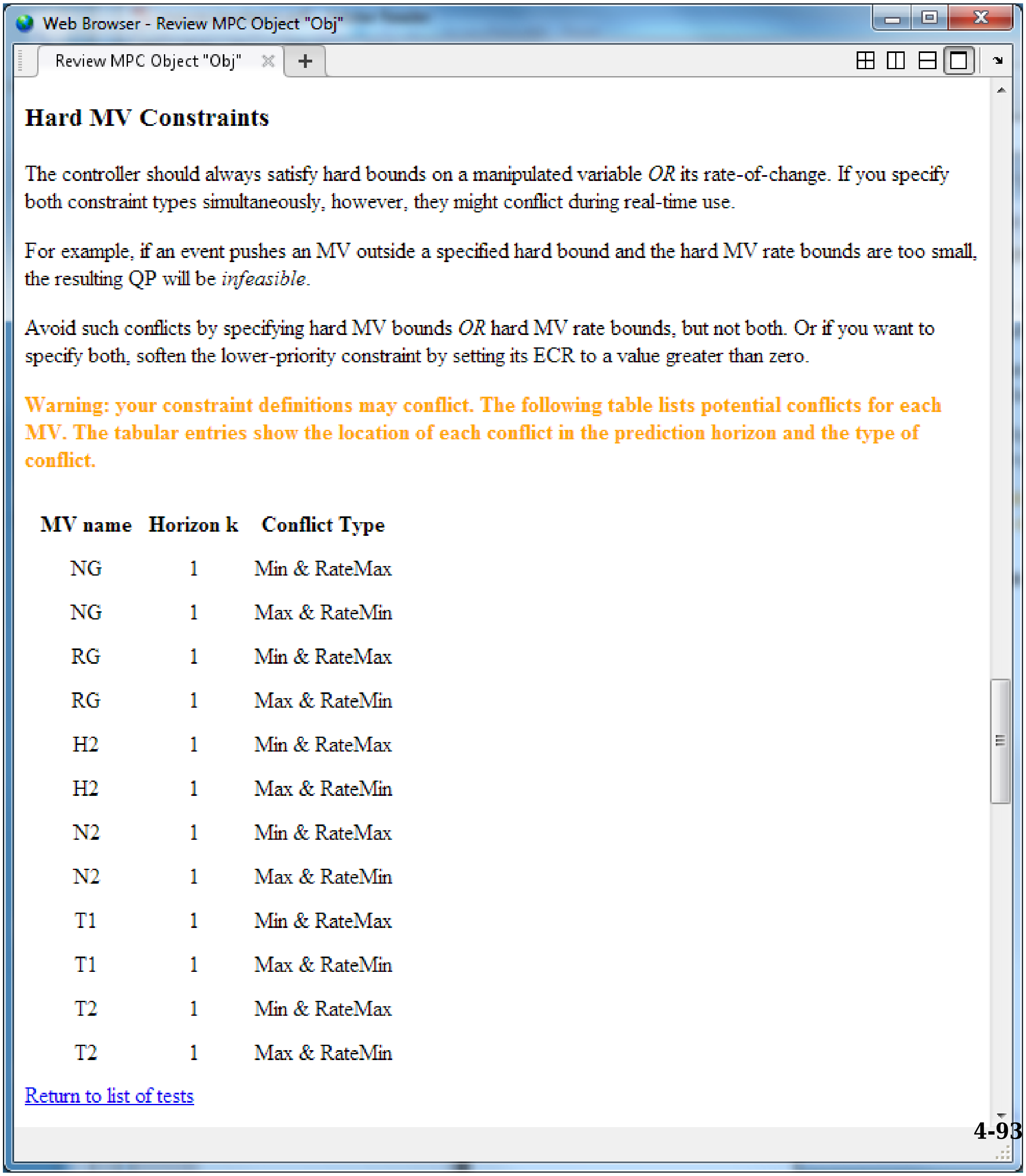
The first entry in the list shows that adding a sustained disturbance of unit magnitude to the HHV output would cause the HHV to deviate 0.0860 units from its steady-state target, assuming no constraints are active. The second entry shows that a unit disturbance in WI would cause a steady-state deviation ("offset") of -0.0345 in HHV, etc.

Since there are six MVs and only four OVs, excess degrees of freedom are available and you might be surprised to see nonzero steady-state offsets. The nonzero MV weights we have specified in order to drive the plant toward the most economical operating condition are causing this.

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, etc.) can vary freely within the specified OV limits.
2 The small offset gain magnitudes indicate that the impact of disturbances would be small.
3 The OV limits are soft constraints. Small, short-term violations are acceptable.

View the second warning by clicking **Hard MV Constraints**, which indicates a potential hard-constraint conflict.



Hard MV Constraints

The controller should always satisfy hard bounds on a manipulated variable *OR* its rate-of-change. If you specify both constraint types simultaneously, however, they might conflict during real-time use.

For example, if an event pushes an MV outside a specified hard bound and the hard MV rate bounds are too small, the resulting QP will be *infeasible*.

Avoid such conflicts by specifying hard MV bounds *OR* hard MV rate bounds, but not both. Or if you want to specify both, soften the lower-priority constraint by setting its ECR to a value greater than zero.

Warning: your constraint definitions may conflict. The following table lists potential conflicts for each MV. The tabular entries show the location of each conflict in the prediction horizon and the type of conflict.

MV name	Horizon k	Conflict Type
NG	1	Min & RateMax
NG	1	Max & RateMin
RG	1	Min & RateMax
RG	1	Max & RateMin
H2	1	Min & RateMax
H2	1	Max & RateMin
N2	1	Min & RateMax
N2	1	Max & RateMin
T1	1	Min & RateMax
T1	1	Max & RateMin
T2	1	Min & RateMax
T2	1	Max & RateMin

Return to list of tests

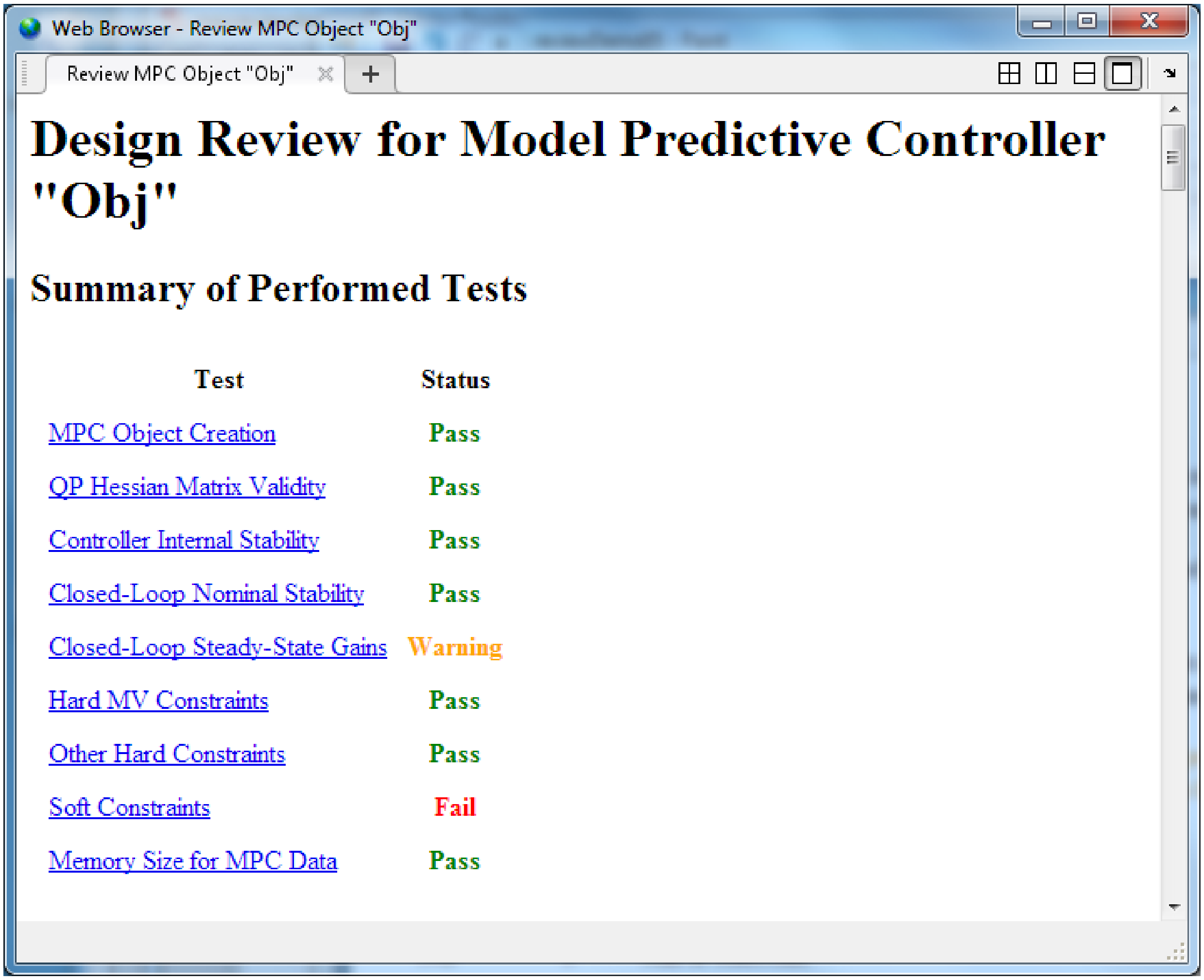
If an external event causes the NG to go far below its specified minimum, the constraint on its rate of increase might make it impossible to return the NG within bounds in one interval. In other words, when you specify both `MV.Min` and `MV.RateMax`, the controller would not be able to find an optimal solution if the most recent MV value is less than (`MV.Min - MV.RateMax`). Similarly, there is a potential conflict when you specify both `MV.Max` and `MV.RateMin`.

An MV constraint conflict would be extremely unlikely in the gas blending application, but it 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, we soften the increment bounds as follows:

```
for i = 1:6
    Obj.MV(i).RateMinECR = 0.1;
    Obj.MV(i).RateMaxECR = 0.1;
end
```

Review the new controller design.

```
review(Obj)
```



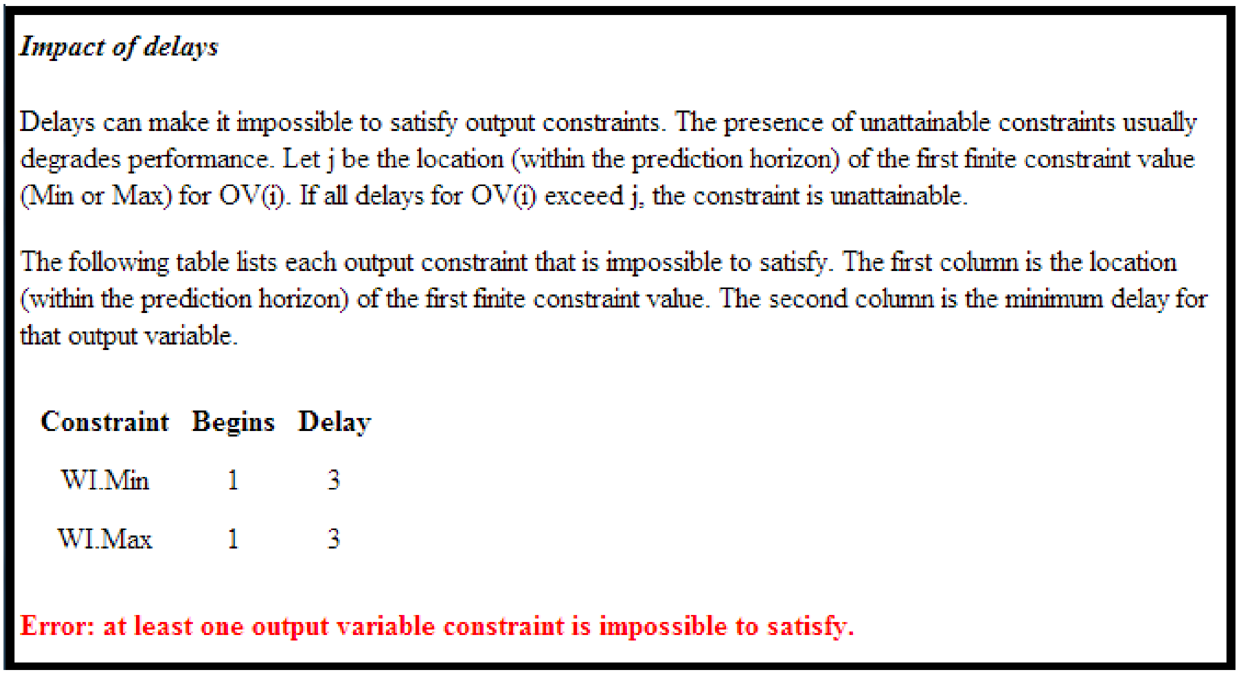
Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

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

The MV constraint conflict warning has disappeared.

Click **Soft Constraints** to view the error message.



Impact of delays

Delays can make it impossible to satisfy output constraints. The presence of unattainable constraints usually degrades performance. Let j be the location (within the prediction horizon) of the first finite constraint value (Min or Max) for OV(i). If all delays for OV(i) exceed j, the constraint is unattainable.

The following table lists each output constraint that is impossible to satisfy. The first column is the location (within the prediction horizon) of the first finite constraint value. The second column is the minimum delay for that output variable.

Constraint	Begins	Delay
WI.Min	1	3
WI.Max	1	3

Error: at least one output variable constraint is impossible to satisfy.

We see that the delay in the WI output makes it impossible to satisfy bounds on that variable until at least three control intervals have elapsed. The WI bounds are soft but it is poor practice to include unattainable constraints in a design. We therefore modify the WI bound specifications such that it is unconstained until the 4th 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 eliminates the error message (see the next step).

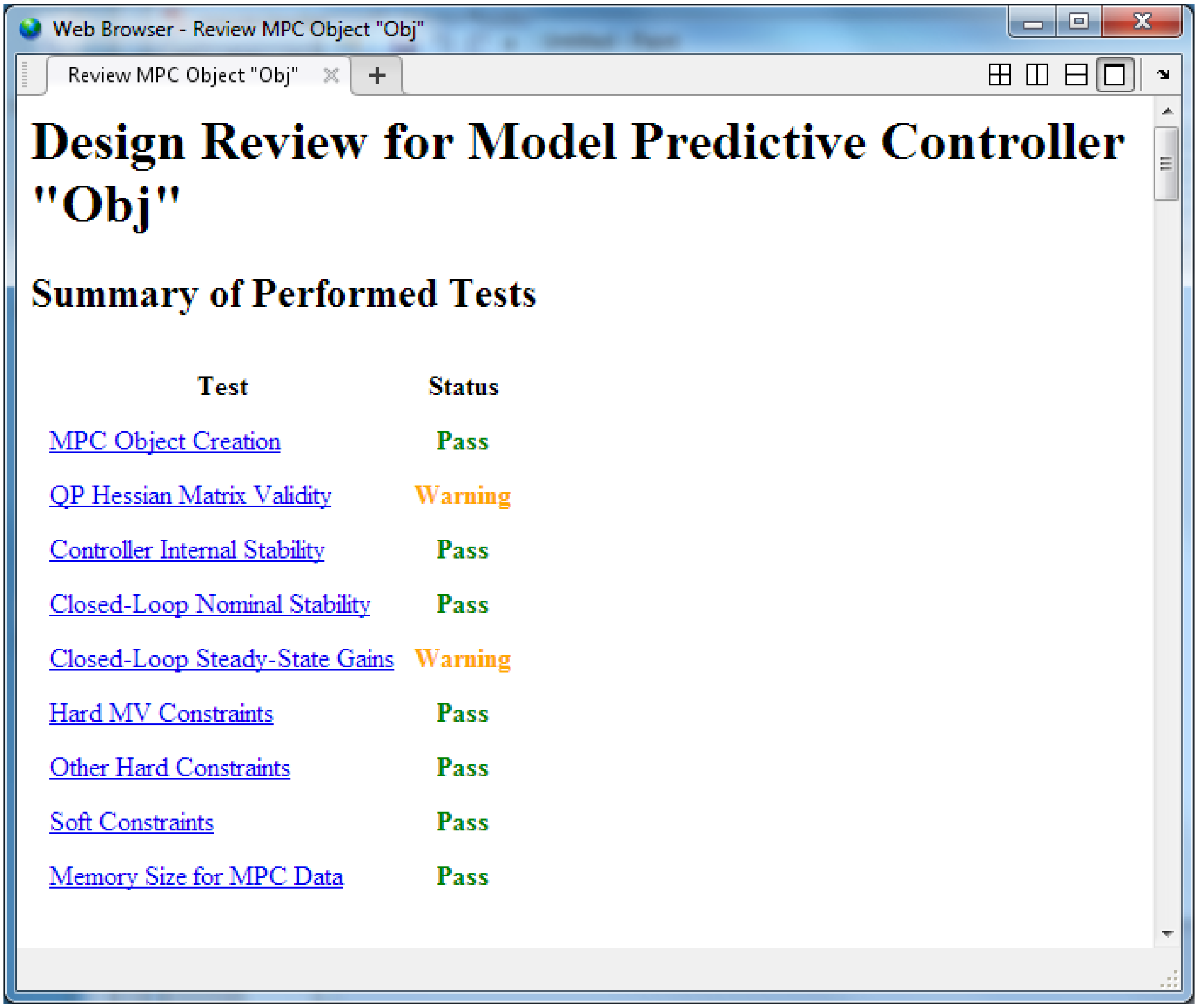
Diagnose Impact of Zero Output Weights

Given that the design requirements allow the OVs to vary freely within their limits, consider zeroing their penalty weights:

```
Obj.Weights.OV = zeros(1,4);
```

Review the impact of this design change.

```
review(Obj)
```



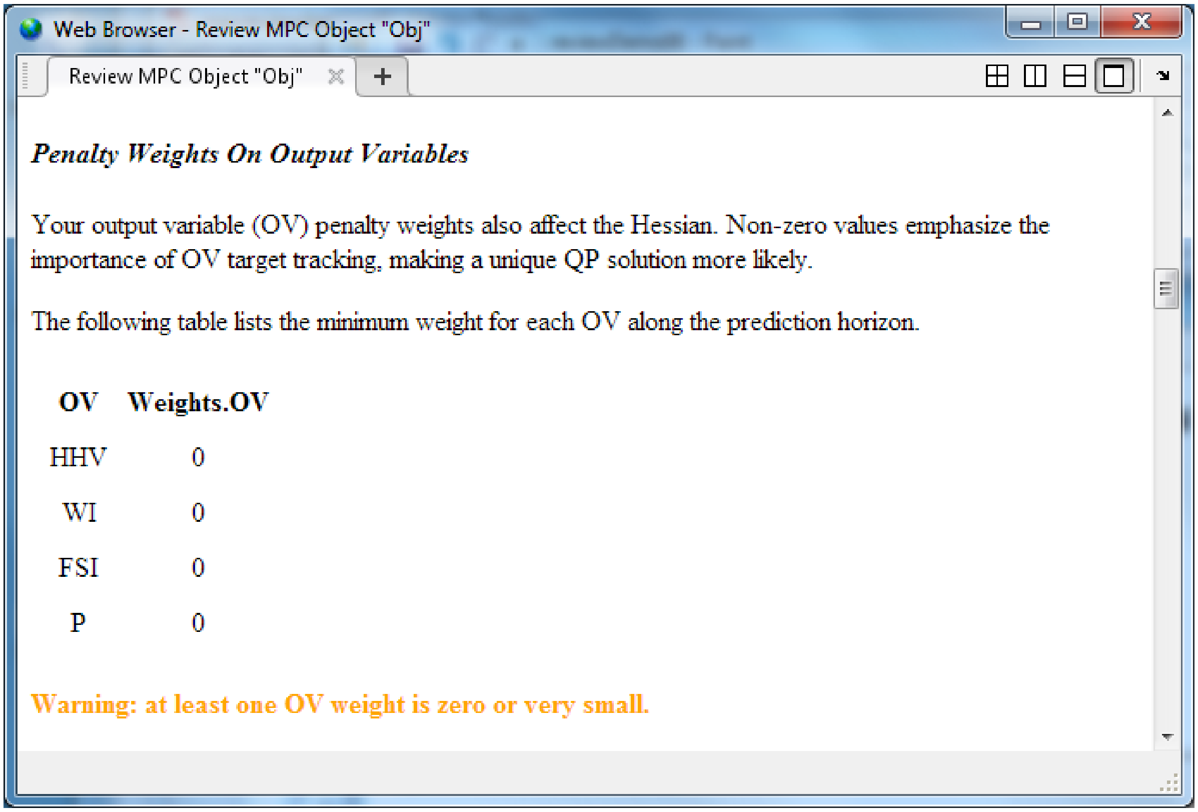
Design Review for Model Predictive Controller "Obj"

Summary of Performed Tests

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

A new warning regarding QP Hessian Matrix Validity has appeared.

Click **QP Hessian Matrix Validity** warning to see the details.



Penalty Weights On Output Variables

Your output variable (OV) penalty weights also affect the Hessian. Non-zero values emphasize the importance of OV target tracking, making a unique QP solution more likely.

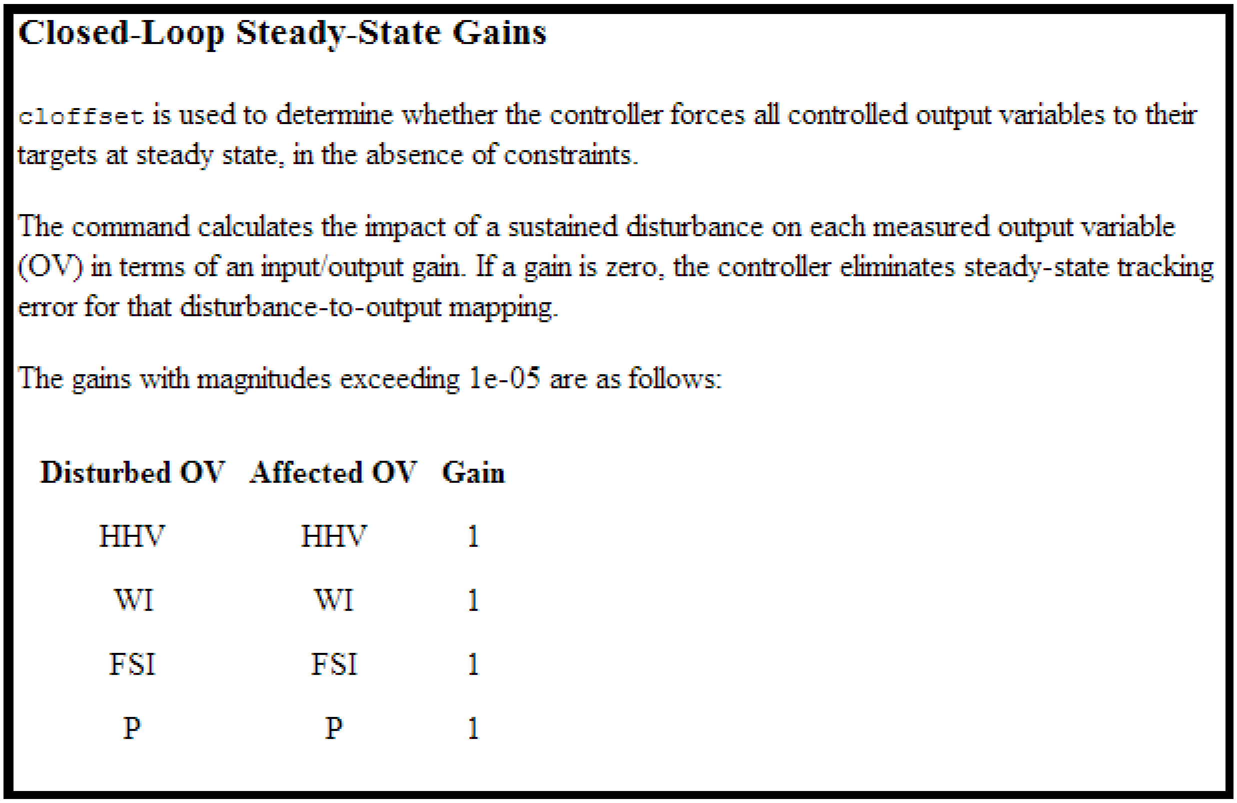
The following table lists the minimum weight for each OV along the prediction horizon.

OV	Weights.OV
HHV	0
WI	0
FSI	0
P	0

Warning: at least one OV weight is zero or very small.

The review has flagged the zero weights on all four output variables. Since the zero weights are consistent with the design requirement and the other Hessian tests indicate that the quadratic programming problem has a unique solution, this warning can be ignored.

Click **Closed-Loop Steady-State Gains** to see the second warning. It shows another consequence of setting the four OV weights to zero. When an OV is not penalized by a weight, any output disturbance added to it will be ignored, passing through with no attenuation.



Closed-Loop Steady-State Gains

`cloffset` is used to determine whether the controller forces all controlled output variables to their targets at steady state, in the absence of constraints.

The command calculates the impact of a sustained disturbance on each measured output variable (OV) in terms of an input/output gain. If a gain is zero, the controller eliminates steady-state tracking error for that disturbance-to-output mapping.

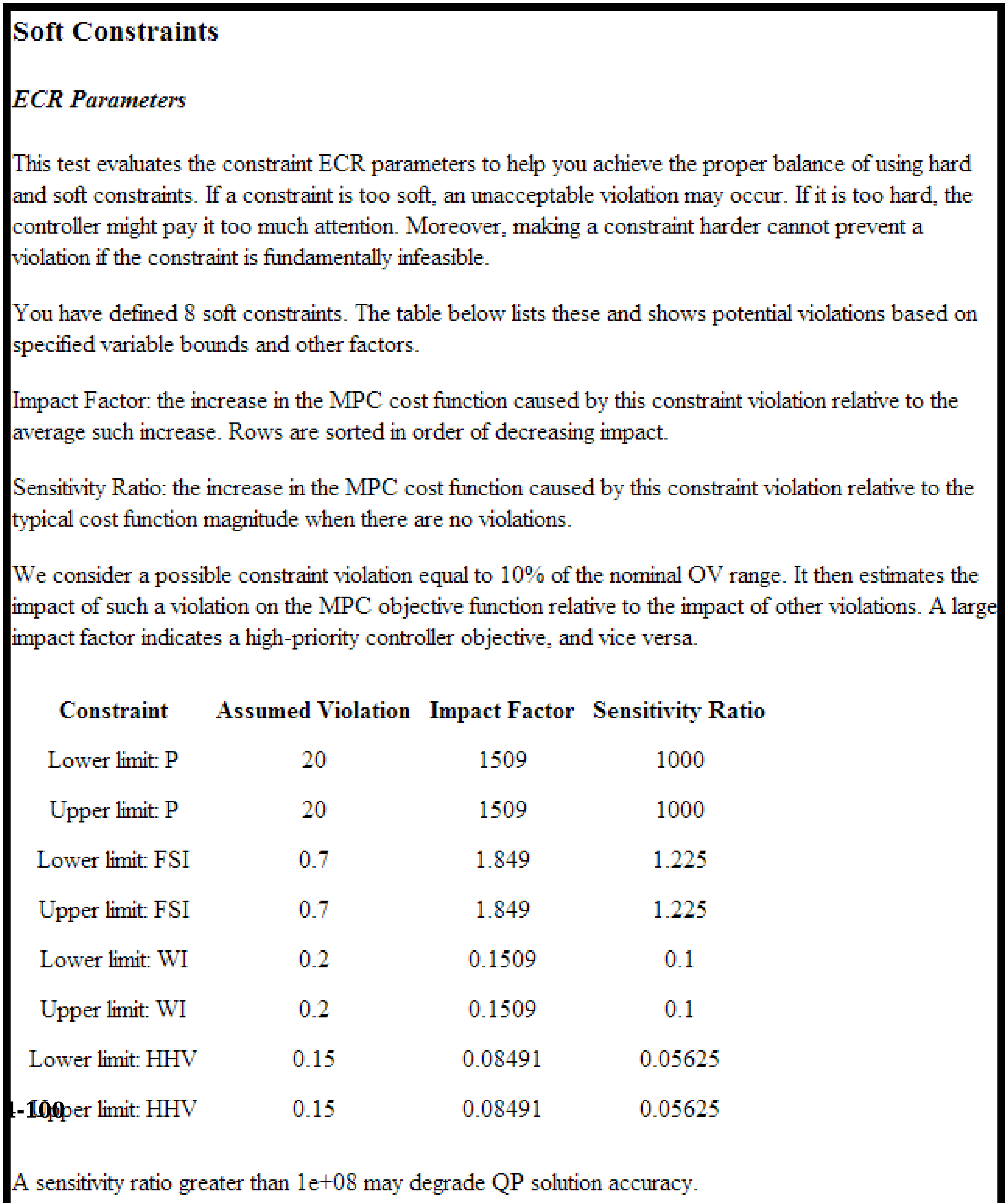
The gains with magnitudes exceeding 1e-05 are as follows:

Disturbed OV	Affected OV	Gain
HHV	HHV	1
WI	WI	1
FSI	FSI	1
P	P	1

Since it is a design requirement, non-zero steady-state offsets are acceptable provided that MPC is able to hold all the OVs within their specified bounds. It is therefore a good idea to examine how easily the soft OV constraints can be violated when disturbances are present.

Review Soft Constraints

Click **Soft Constraints** to see a list of soft constraints - in this case an upper and lower bound on each OV.



Soft Constraints

ECR Parameters

This test evaluates the constraint ECR parameters to help you achieve the proper balance of using hard and soft constraints. If a constraint is too soft, an unacceptable violation may occur. If it is too hard, the controller might pay it too much attention. Moreover, making a constraint harder cannot prevent a violation if the constraint is fundamentally infeasible.

You have defined 8 soft constraints. The table below lists these and shows potential violations based on specified variable bounds and other factors.

Impact Factor: the increase in the MPC cost function caused by this constraint violation relative to the average such increase. Rows are sorted in order of decreasing impact.

Sensitivity Ratio: the increase in the MPC cost function caused by this constraint violation relative to the typical cost function magnitude when there are no violations.

We consider a possible constraint violation equal to 10% of the nominal OV range. It then estimates the impact of such a violation on the MPC objective function relative to the impact of other violations. A large impact factor indicates a high-priority controller objective, and vice versa.

Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: P	20	1509	1000
Upper limit: P	20	1509	1000
Lower limit: FSI	0.7	1.849	1.225
Upper limit: FSI	0.7	1.849	1.225
Lower limit: WI	0.2	0.1509	0.1
Upper limit: WI	0.2	0.1509	0.1
Lower limit: HHV	0.15	0.08491	0.05625
Upper limit: HHV	0.15	0.08491	0.05625

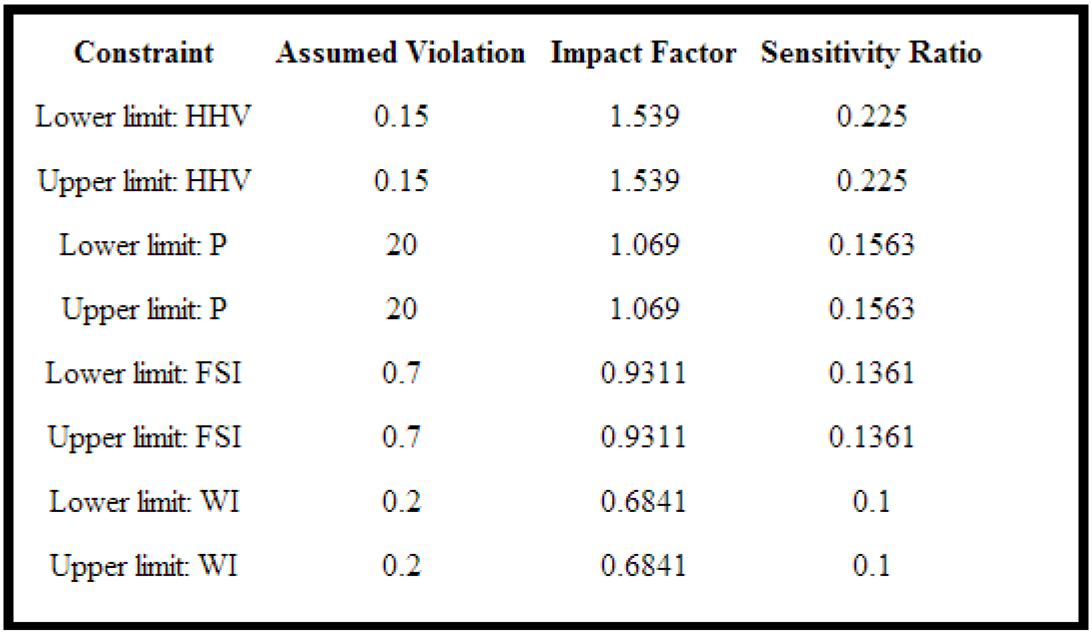
A sensitivity ratio greater than 1e+08 may degrade QP solution accuracy.

The Impact Factor column shows that using the default `MinECR` and `MaxECR` values give the pressure (P) a much higher priority than the other OVs. If we want the priorities to be more comparable, we should increase the pressure constraint ECR values and adjust the others too. For example, consider:

```
Obj.OV(1).MinECR = 0.5;
Obj.OV(1).MaxECR = 0.5;
Obj.OV(3).MinECR = 3;
Obj.OV(3).MaxECR = 3;
Obj.OV(4).MinECR = 80;
Obj.OV(4).MaxECR = 80;
```

Review the impact of this design change.

```
review(Obj)
```



Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: HHV	0.15	1.539	0.225
Upper limit: HHV	0.15	1.539	0.225
Lower limit: P	20	1.069	0.1563
Upper limit: P	20	1.069	0.1563
Lower limit: FSI	0.7	0.9311	0.1361
Upper limit: FSI	0.7	0.9311	0.1361
Lower limit: WI	0.2	0.6841	0.1
Upper limit: WI	0.2	0.6841	0.1

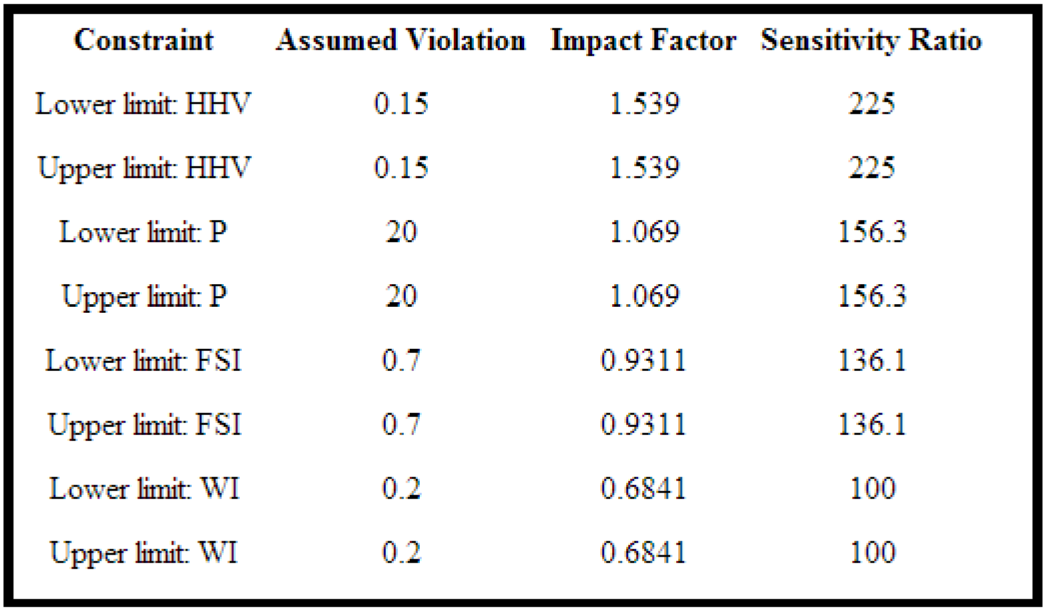
Notice from the Sensitivity Ratio column that all the sensitivity ratios are now less than unity. This means that the soft constraints will receive less attention than other terms in the MPC objective function, such as deviations of the MVs from their target values. Thus, it is likely that an output constraint violation would occur.

In order to give the output constraints higher priority than other MPC objectives, increase the `Weights.ECR` parameter from the default, 1e5, to a higher value to harden all the soft OV constraints.

```
Obj.Weights.ECR = 1e8;
```

Review the impact of this design change.

```
review(Obj)
```

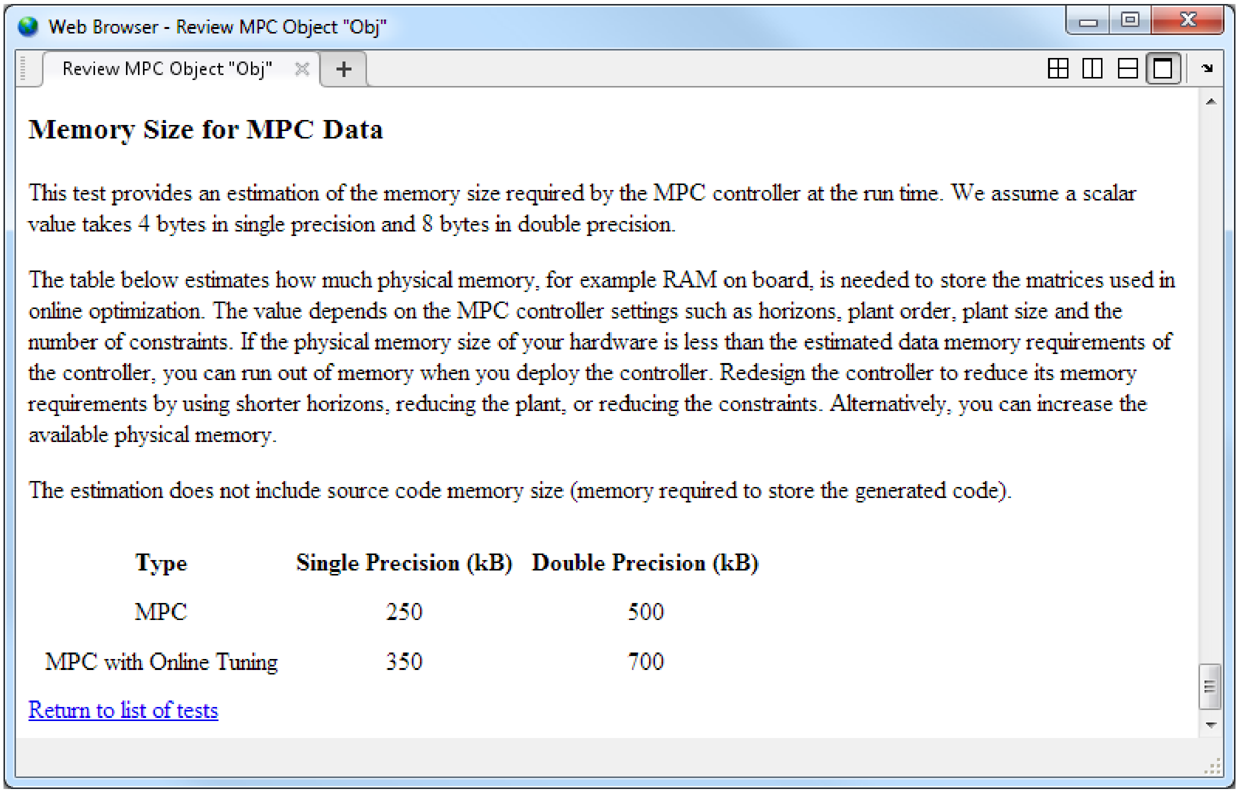


Constraint	Assumed Violation	Impact Factor	Sensitivity Ratio
Lower limit: HHV	0.15	1.539	225
Upper limit: HHV	0.15	1.539	225
Lower limit: P	20	1.069	156.3
Upper limit: P	20	1.069	156.3
Lower limit: FSI	0.7	0.9311	136.1
Upper limit: FSI	0.7	0.9311	136.1
Lower limit: WI	0.2	0.6841	100
Upper limit: WI	0.2	0.6841	100

The controller is now a factor of 100 more sensitive to output constraint violations than to errors in target tracking.

Review Data Memory Size

Click **Memory Size for MPC Data** to see the estimated memory size needed to store the MPC data matrices used on the hardware.



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

Return to list of tests

In this example, if the controller is running using single precision, it requires 250 KB of memory to store its matrices. If the controller memory size exceeds the memory available on the target system, you must redesign the controller to reduce its memory requirements. Alternatively, increase the memory available on the target system.

```
mpcverbosity(MPC_verbosity);
[~, hWebBrowser] = web;
close(hWebBrowser);
```

See Also

`review`

Control of an Inverted Pendulum on a Cart

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

Product Requirement

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

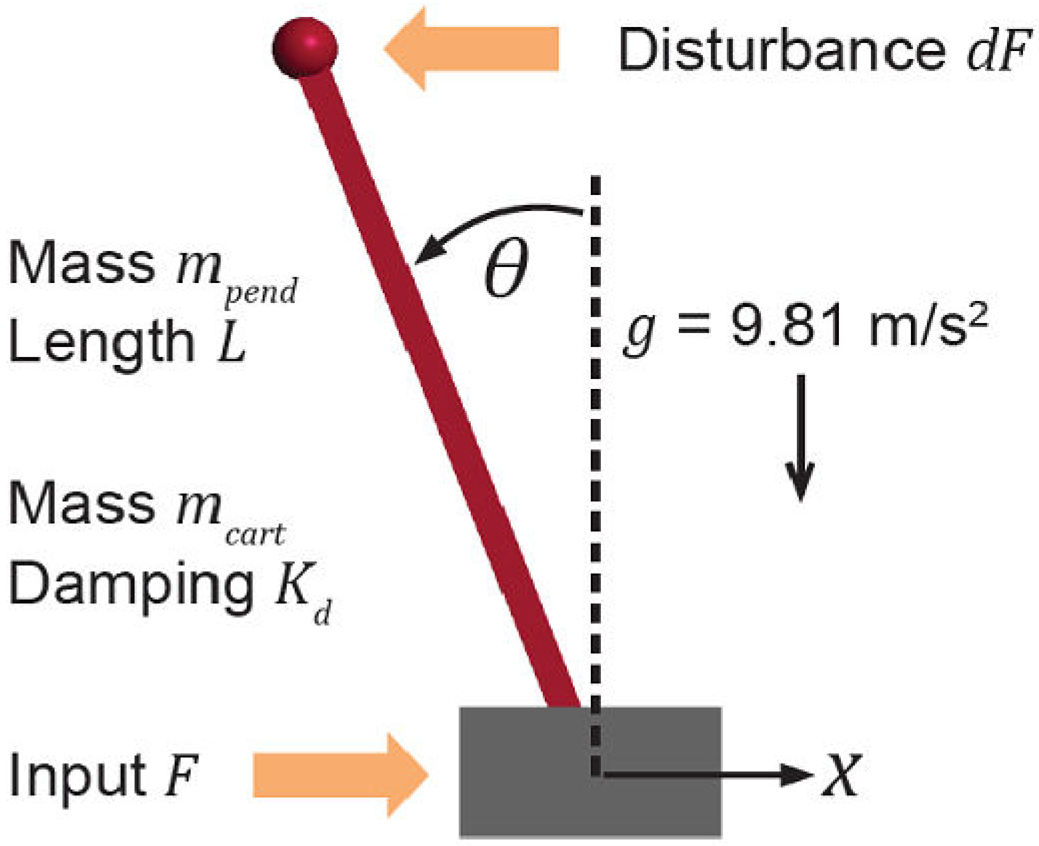
```
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc_featured','main'));
```

Pendulum/Cart Assembly

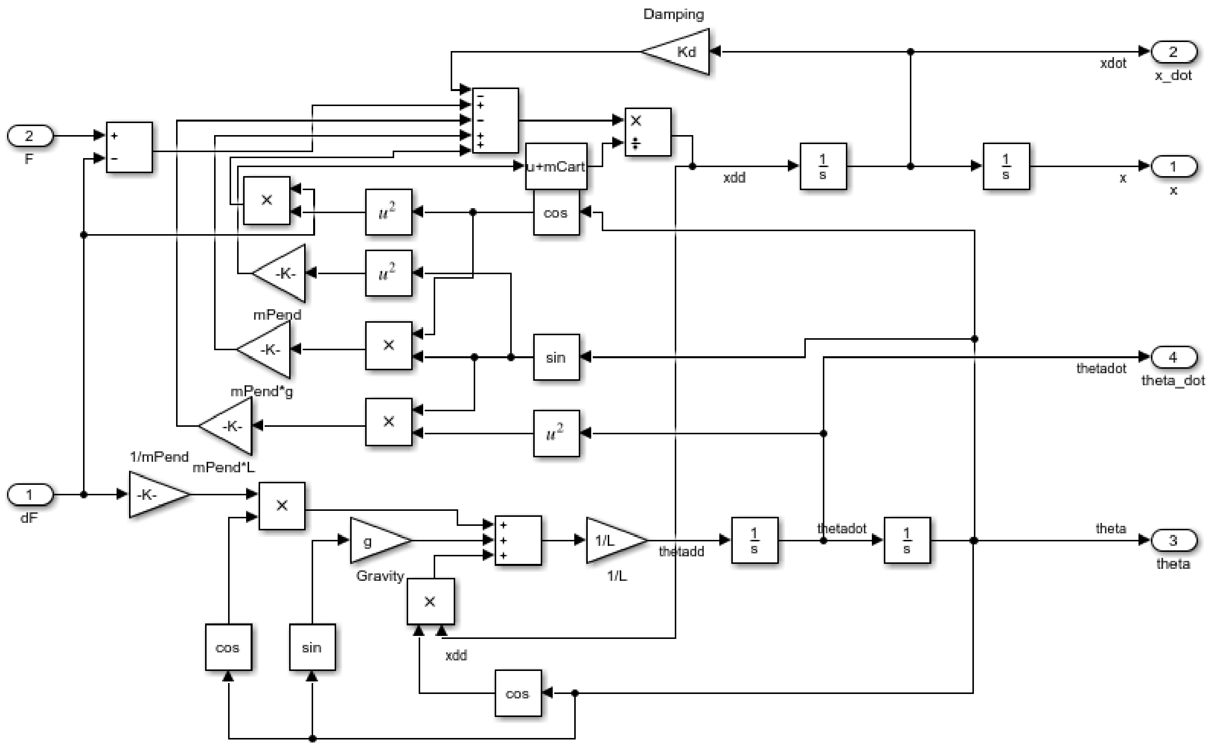
The plant for this example is the following cart/pendulum assembly, where *x* is the cart position and *theta* is the pendulum angle.



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).

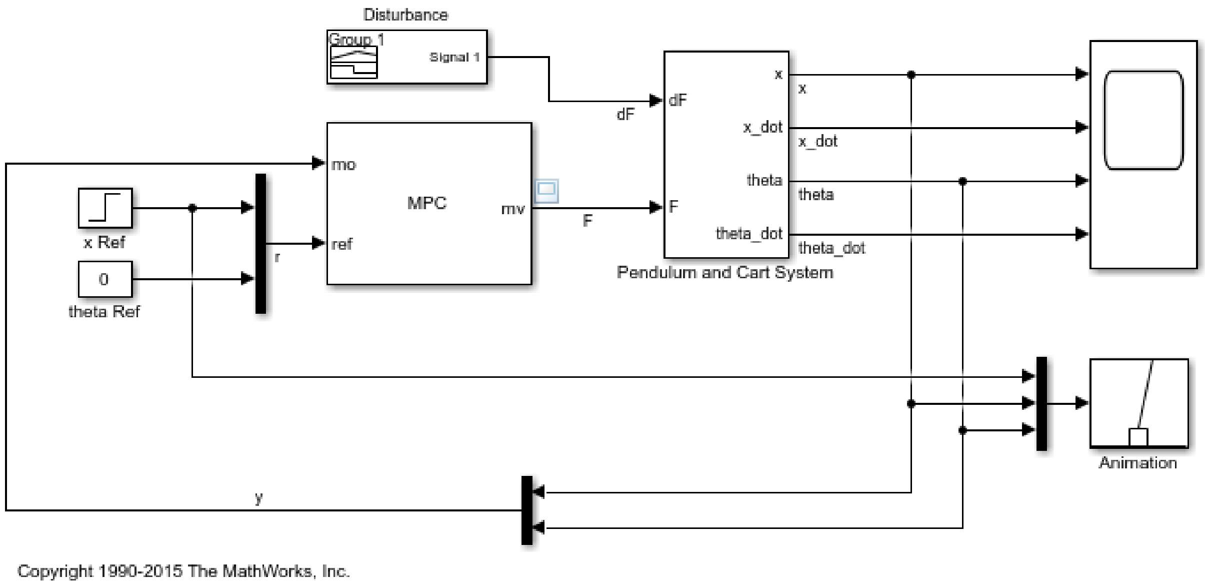
The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

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)
```



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

There is a limitation on how much force can be applied to the cart, which is specified as hard constraints on manipulated variable *F*.

```
mpcobj.MV.Min = -200;
mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by `100`.

```
mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from `0.1` to `1`.

```
mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position *x* and the second weight is associated with angle *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
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Update the output disturbance model.

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

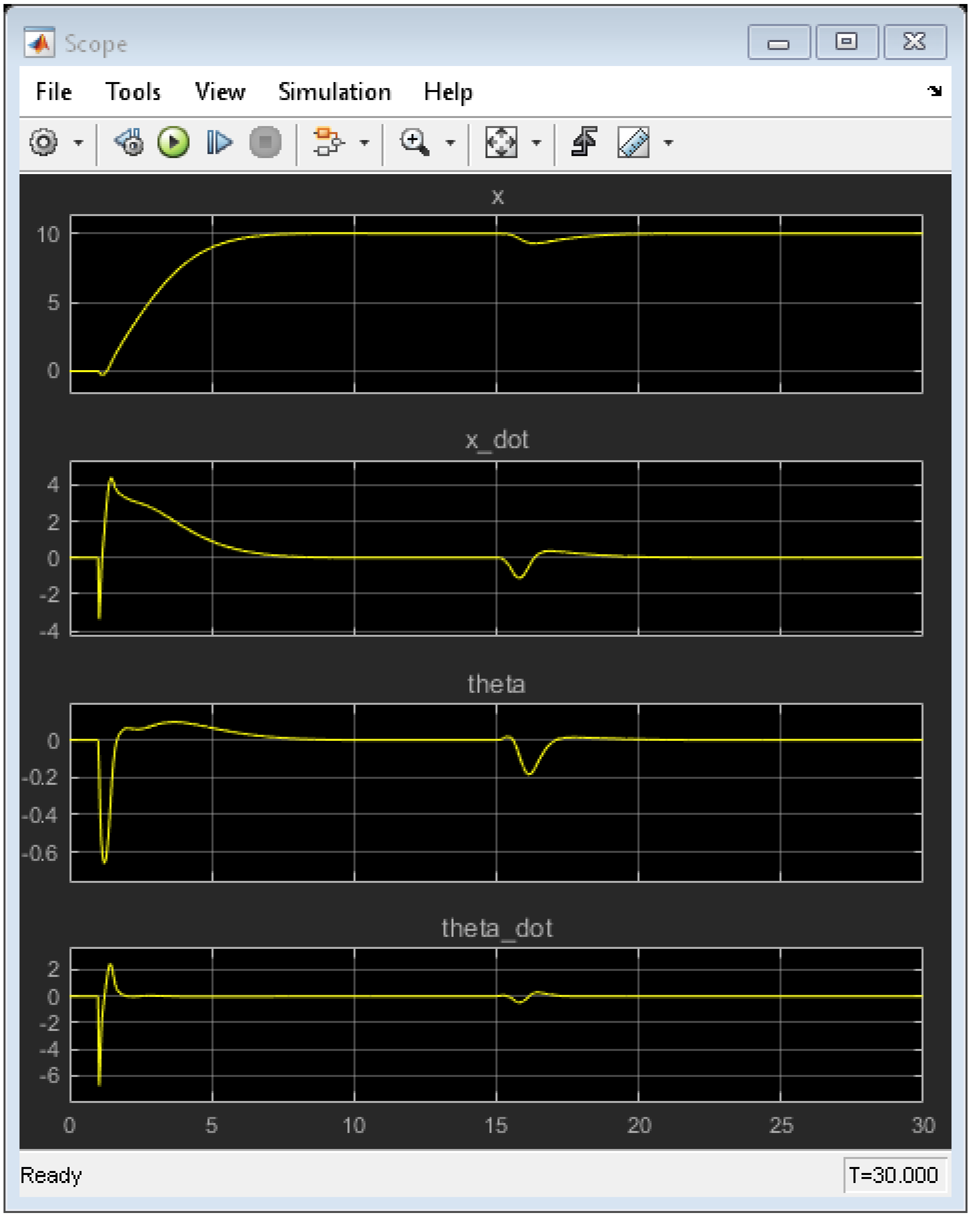
```
-->Converting model to discrete time.
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

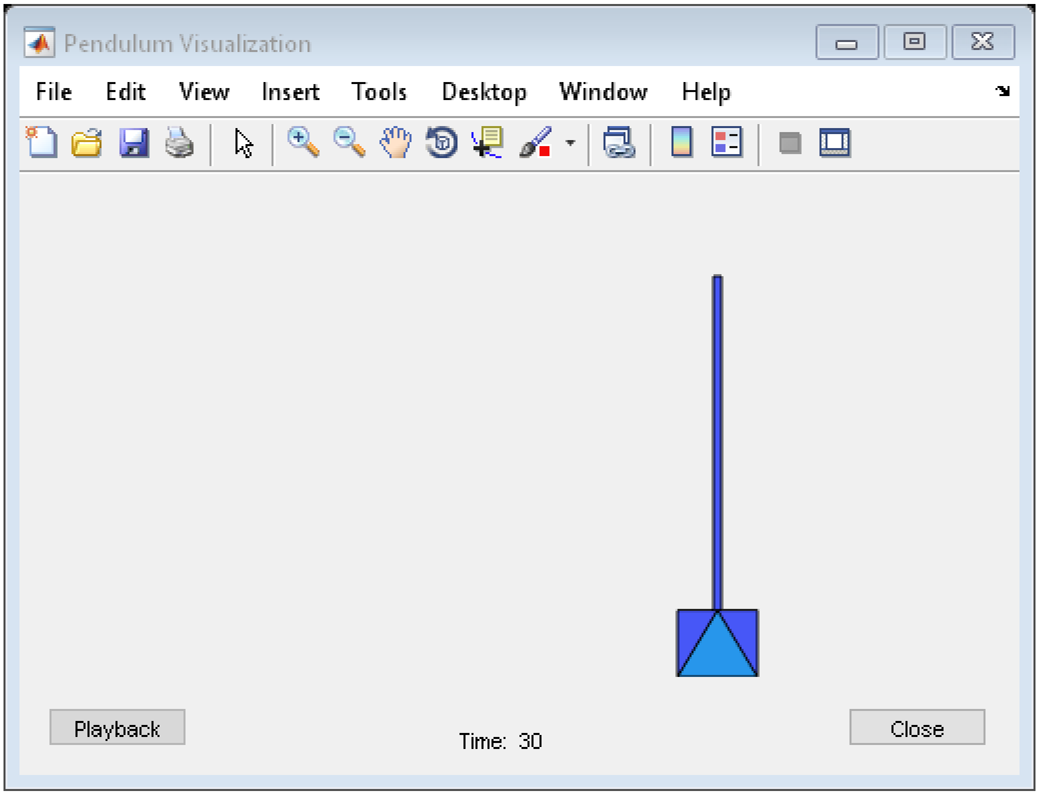
Closed-Loop Simulation

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 eac
```


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
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” shows how to use gain scheduling MPC to achieve the longer distances.

Remove the example file folder from the MATLAB path, and close the Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc_featured','main'));
bdclose(mdlMPC)
```

See Also

More About

- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-42
- “Gain-Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 7-68

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` 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 9-32. For more information on QP Solvers, see “QP Solver” on page 2-37.

In the on-line monitoring example, the `qp.status` output of the MPC Controller block returns a positive integer whenever the controller obtains a valid solution of the current run-time QP problem and sets the `mv` output. The `qp.status` value corresponds to the number of iterations used to solve this QP.

If the QP is infeasible for a given control interval, the controller fails to find a solution. In that case, the `mv` outport stays at its most recent value and the `qp.status` outport returns `-1`. Similarly, if the maximum number of iterations is reached during optimization (rare), the `mv` outport also freezes and the `qp.status` outport returns `0`.

Real-time MPC applications can detect whether the controller is in a "failure" mode (`0` or `-1`) by monitoring the `qp.status` outport. If a failure occurs, a backup control plan should be activated. This is essential if there is any chance that the QP could become infeasible, because the default action (freezing MVs) may lead to unacceptable system behavior, such as instability. Such a backup plan is, necessarily, application-specific.

MPC Application with Online Monitoring

The plant used in this example is a single-input, single-output system with hard limits on both the manipulated variable (MV) and the controlled output (OV). The control objective is to hold the OV at a setpoint of `0`. An unmeasured load disturbance is added to the OV. This disturbance is initially a ramp increase. The controller response eventually saturates the MV at its hard limit. Once saturation occurs, the controller can do nothing more, and the disturbance eventually drives the OV above its specified hard upper limit. When the controller predicts that it is impossible to force the OV below this upper limit, the run-time QP becomes infeasible.

Define the plant as a first-order SISO system with unity gain.

```
Plant = tf(1,[2 1]);
```

Define the unmeasured load disturbance. The signal ramps up from 0 to 2 between 1 and 3 seconds, then ramps back down from 2 to 0 between 3 and 5 seconds.

```
LoadDist = [0 0; 1 0; 3 2; 5 0; 7 0];
```

Design MPC Controller

Create an MPC object using the model of the test plant. The chosen control interval is about one tenth of the dominant plant time constant.

```
Ts = 0.2;
Obj = mpc(Plant, Ts);
```

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

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

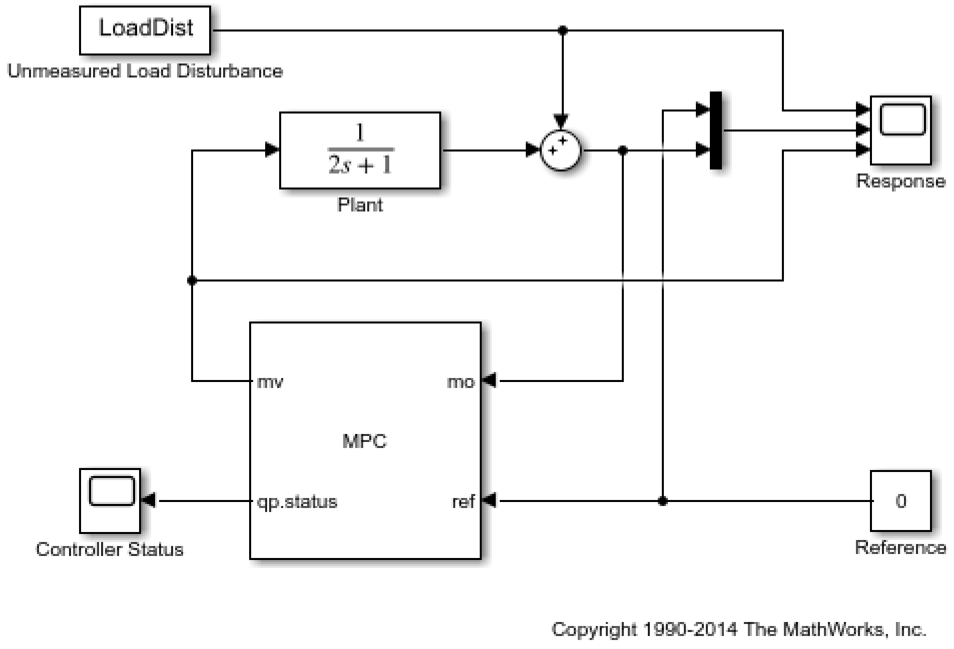
To run this example, Simulink and the Optimization Toolbox are required.

```
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.

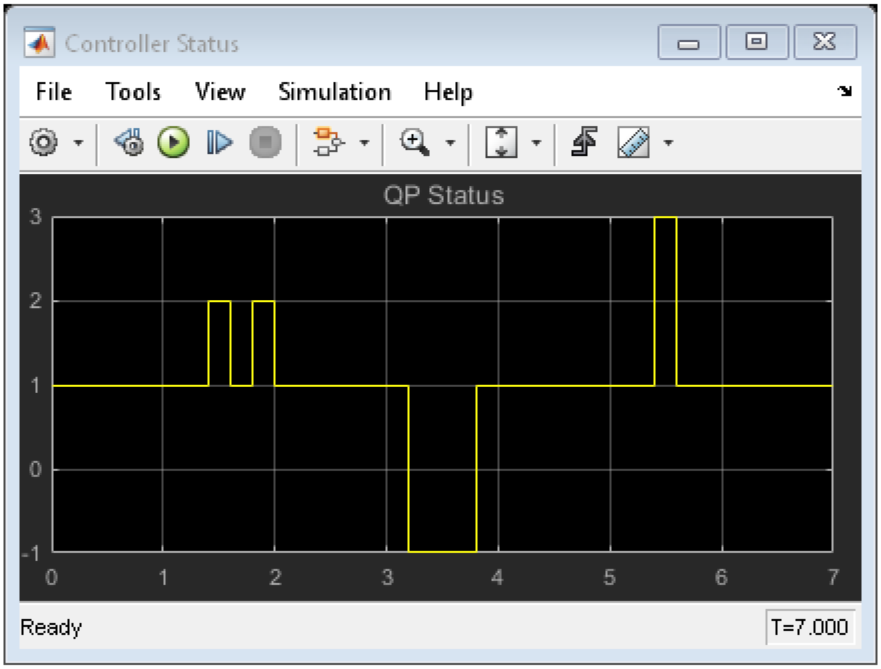
```
mdl = 'mpc_onlinemonitoring';
open_system(mdl)
```

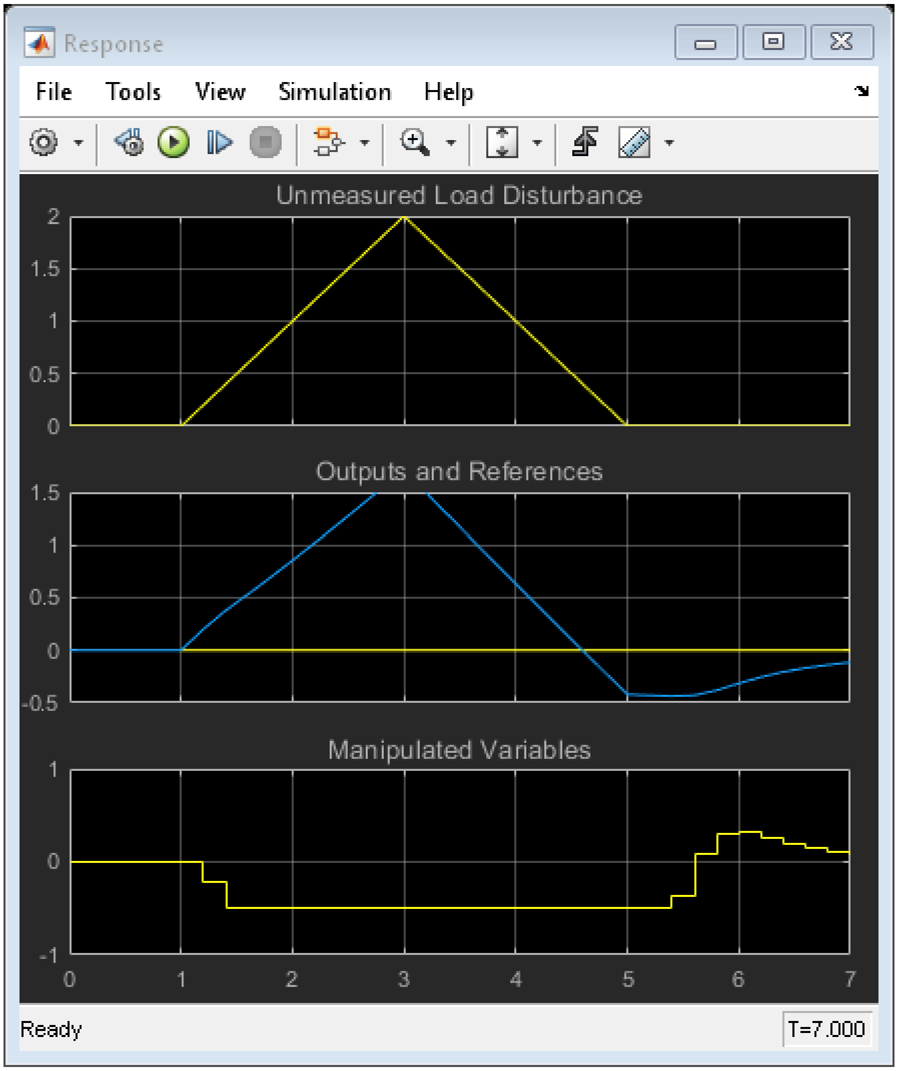


Simulate the closed-loop response using the default Model Predictive Control Toolbox QP solver.

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





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.

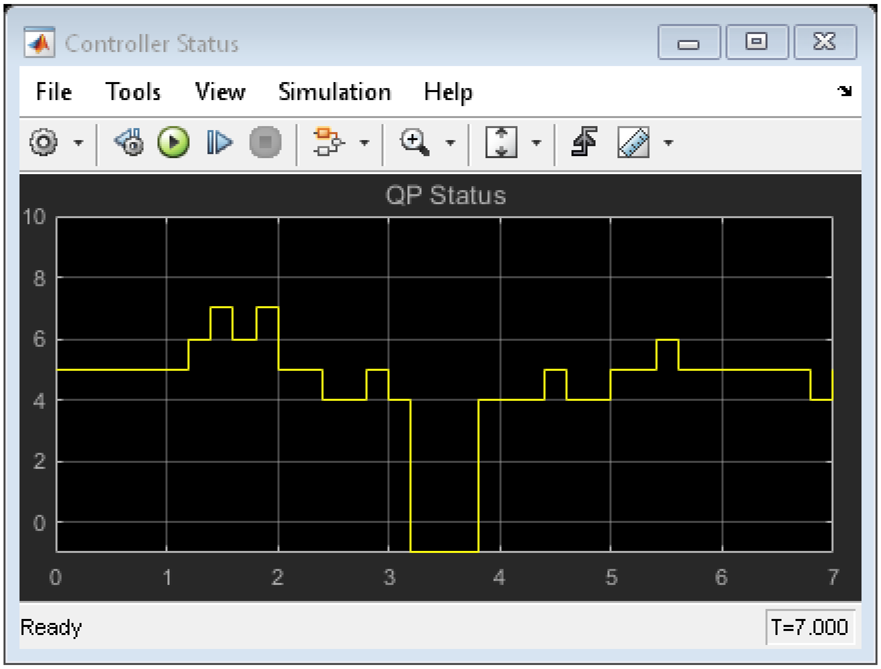
```
function [x, status] = mpcCustomSolver(H, f, A, b, x0)
% mpcCustomSolver allows user to specify a custom quadratic programming
% (QP) solver to solve the QP problem formulated by MPC controller.  When
% the "mpcobj.Optimizer.CustomSolver" property is set true, instead of
% using the built-in QP solver, 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;
```

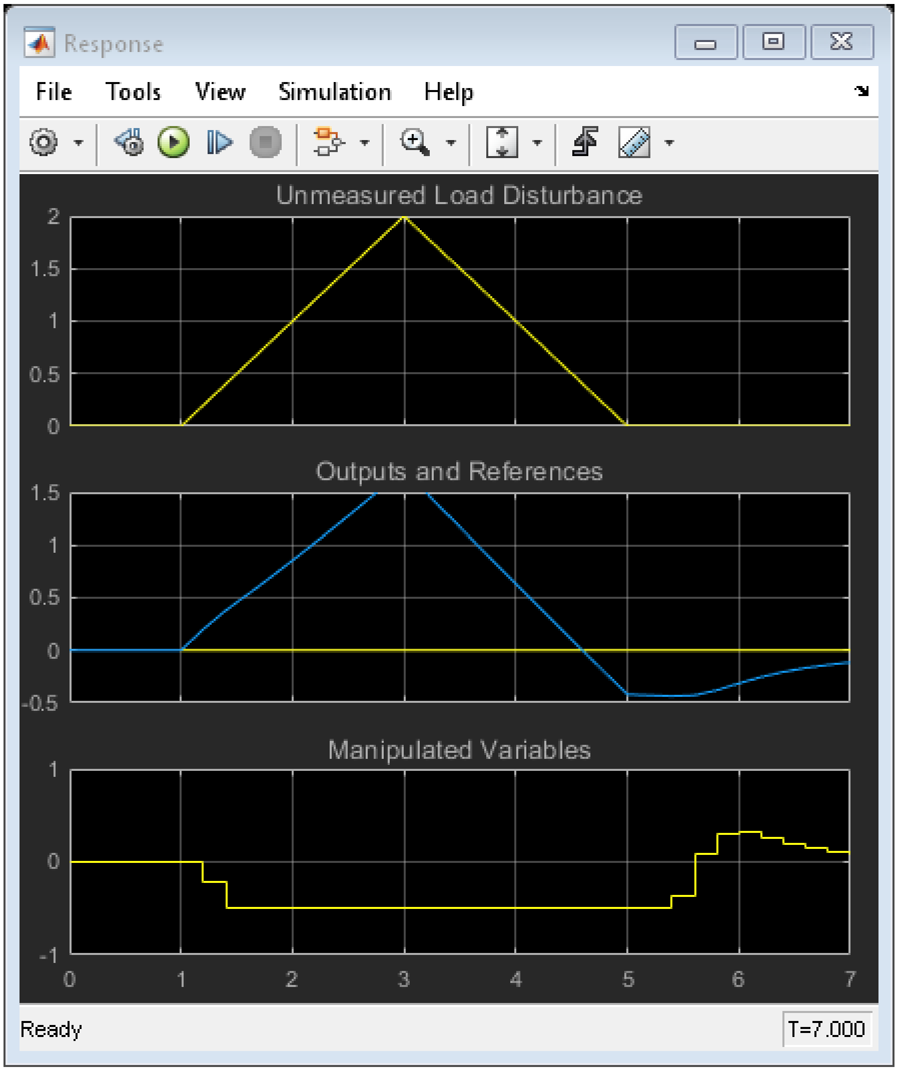
```
% Compute the QP's optimal solution.  Note that the default algorithm used
% by "quadprog" ('interior-point-convex') ignores x0.  "x0" is used here as
% an input argument for illustration only.
H = (H+H')/2; % ensure Hessian is symmetric
[x, ~, Flag, Output] = quadprog(H, f, A_custom, b_custom, [], [], [], [], 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
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on ea
```


Controller Status
File
Tools
View
Simulation
Help
QP Status
10
8
6
4
2
0
0
1
2
3
4
5
6
7
Ready
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 Solver” on page 2-37
- “Simulate and Generate Code for MPC Controller with Custom QP Solver” on page 9-32

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.

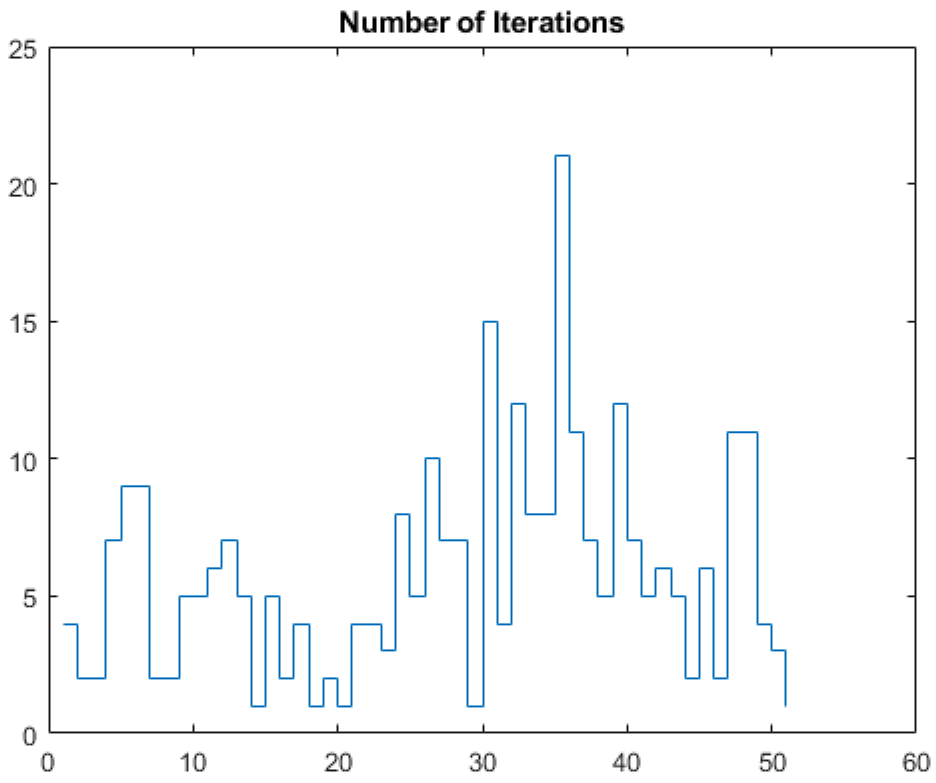
Simulate in MATLAB with Random Output Disturbances

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 to a small number.

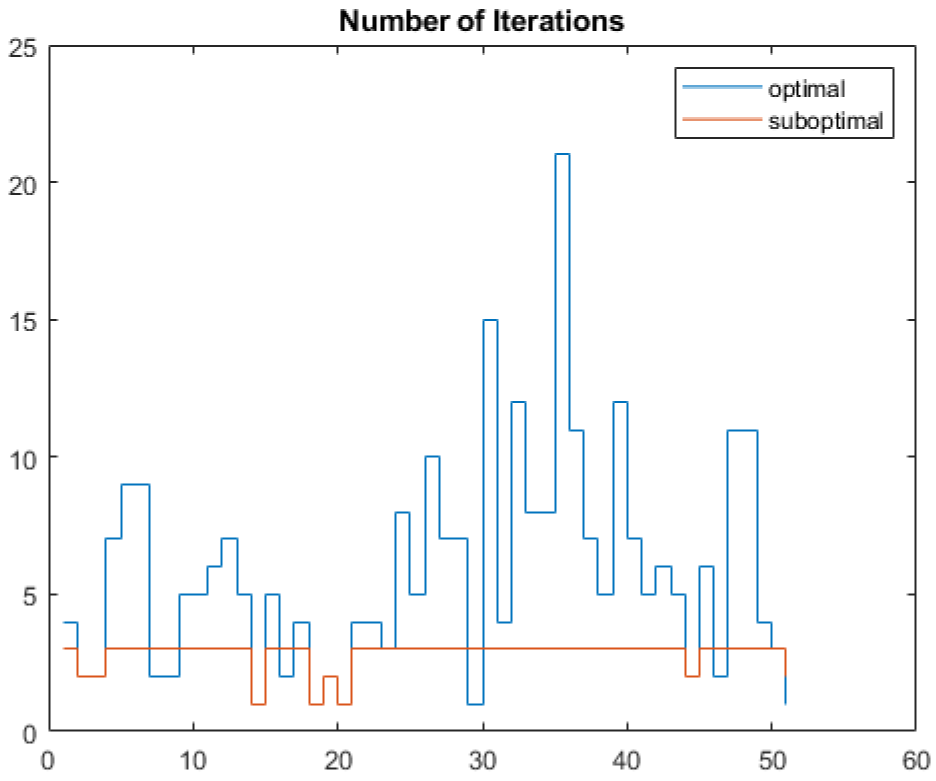
```
mpcobjSub.Optimizer.MaxIter = 3;
```

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`.

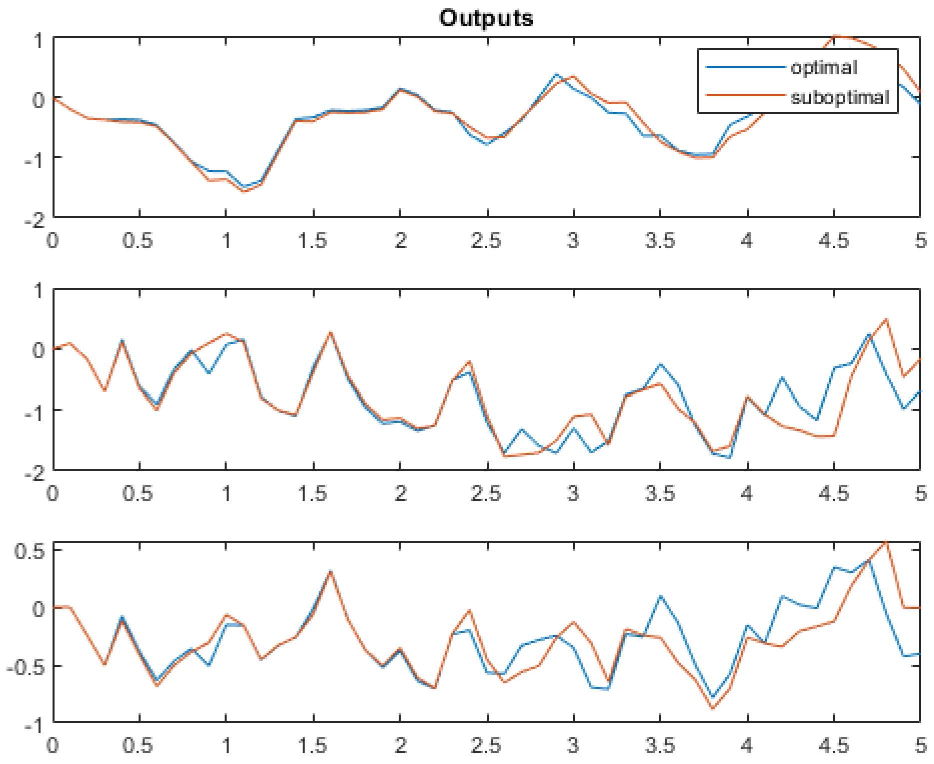
```
statusSub(statusSub == 0) = 3;
stairs(statusSub)
legend('optimal','suboptimal')
```



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')
```

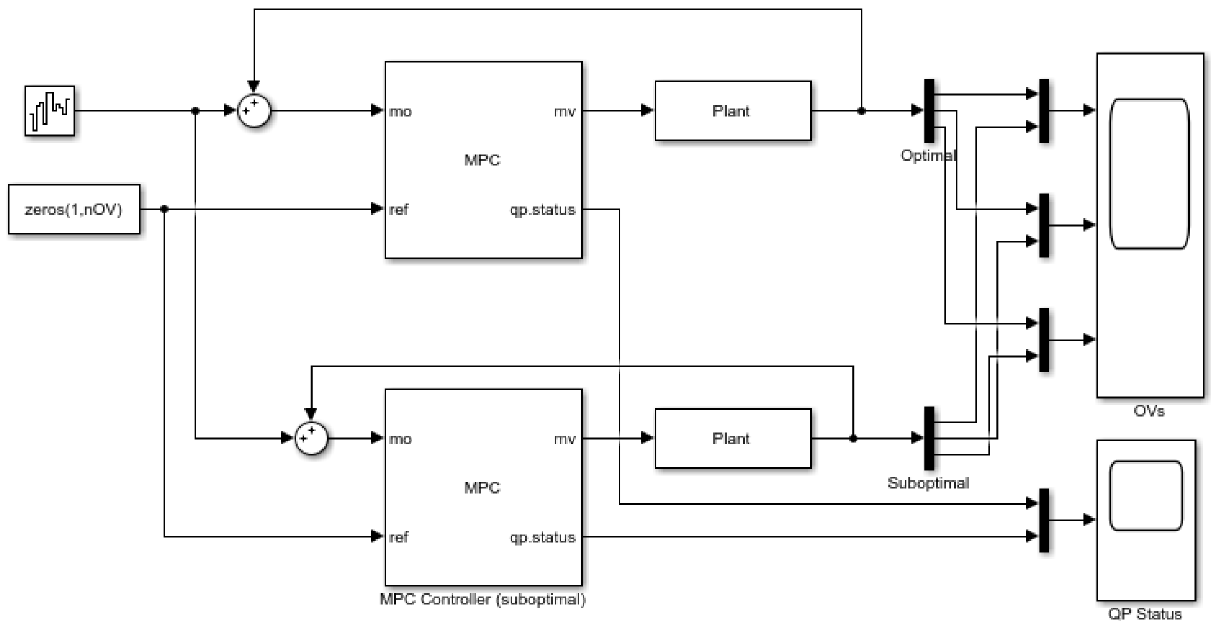


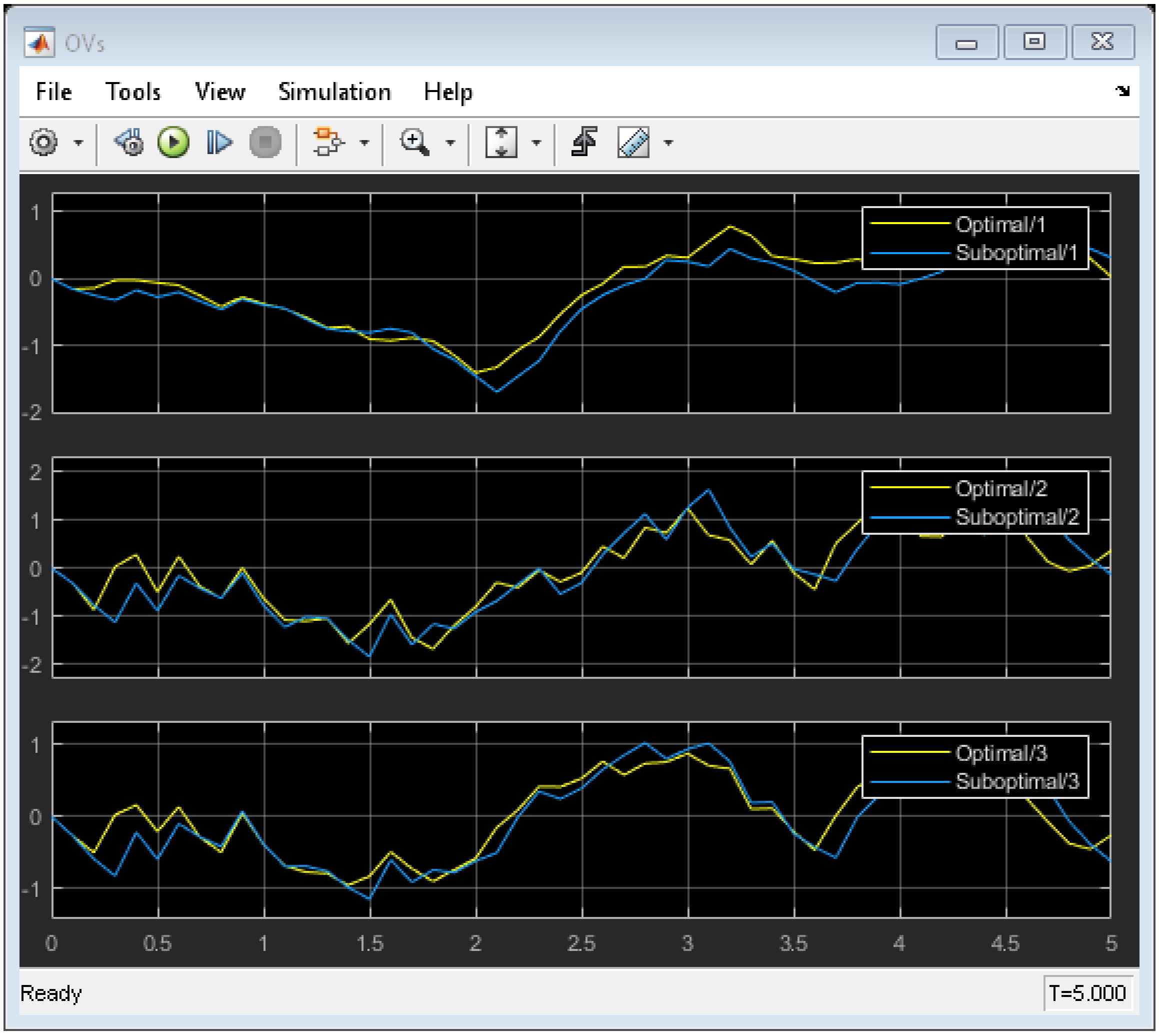
For a real-time application, as long as each solver iteration takes less than 30 milliseconds on the hardware, the worst-case execution time does not exceed the controller sample time (`0.1` seconds). In general, it is safe to assume that the execution time used by each iteration is more or less a constant.

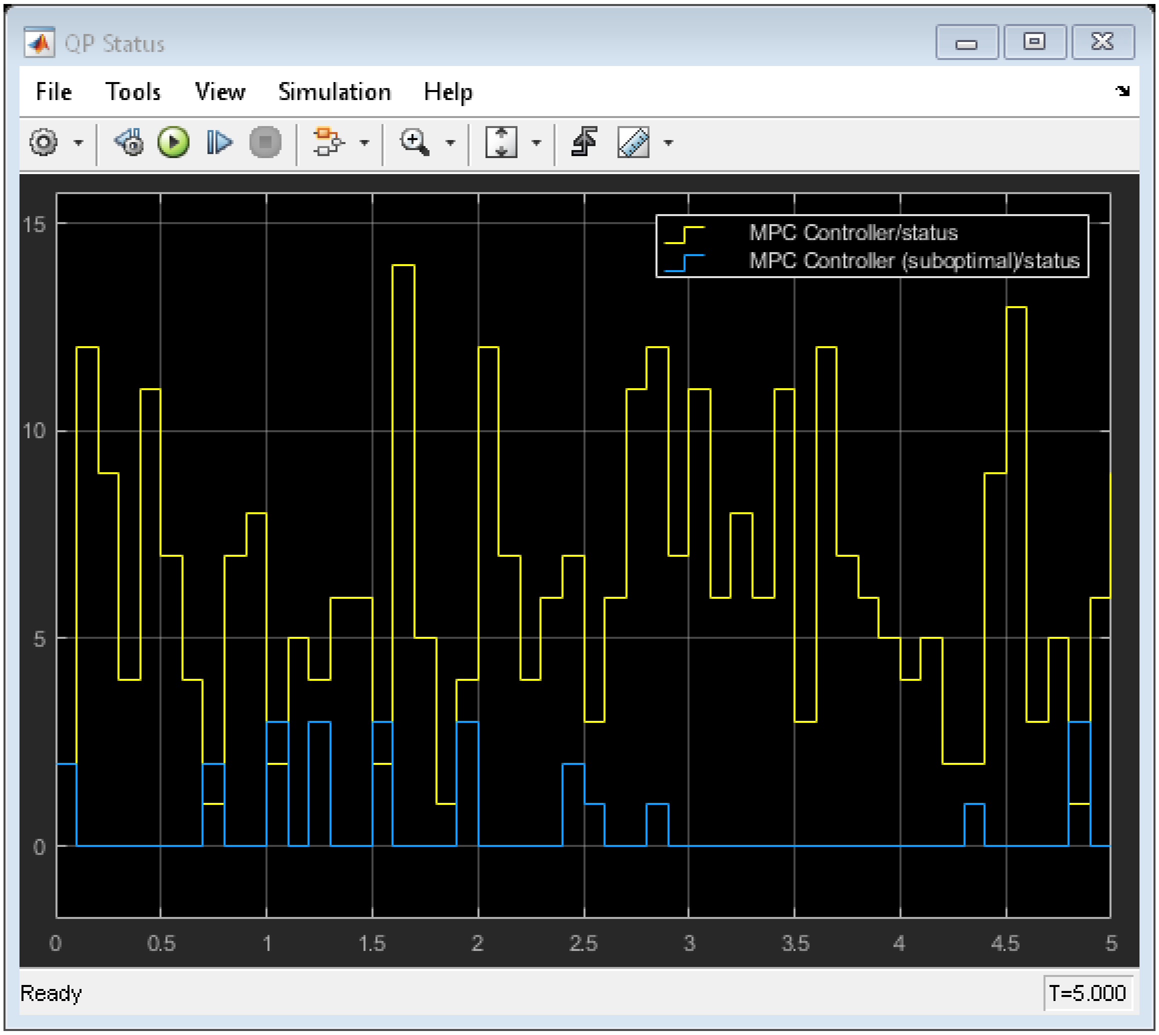
Simulate in Simulink with Random Output Disturbances

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)
```




OVs
File
Tools
View
Simulation
Help
Optimal/1
Suboptimal/1
Optimal/2
Suboptimal/2
Optimal/3
Suboptimal/3
Ready
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
Adaptive MPC Controller | MPC Controller

More About

- “QP Solver” on page 2-37

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”.

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 defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify controller tuning weights.

```
mpcobj.Weights.MV = 0;
mpcobj.Weights.MVrate = 0.5;
mpcobj.Weights.OV = 1;
```

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`.

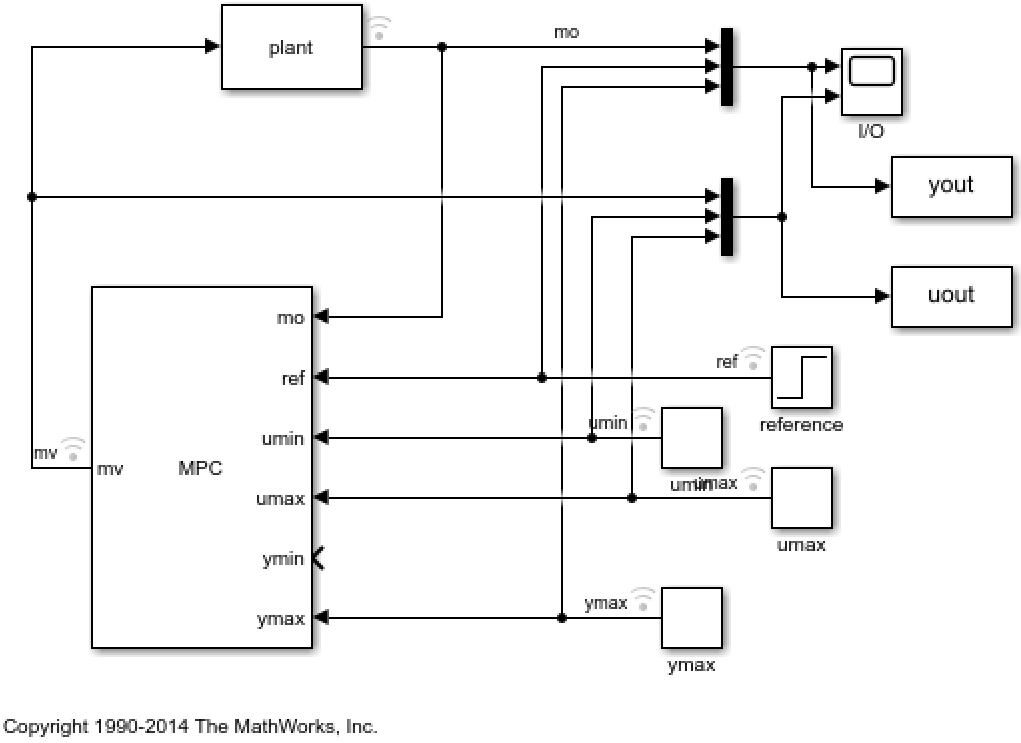
At run time, these constraints are changed using an `mpcmoveopt` object at the command line or corresponding input signals to the MPC Controller block.

```
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)
```



In this model, the input and output constraint input ports of the MPC Controller block are enabled. The `umin`, `umax`, and `ymax` ports are connected to signals which change during the simulation. Since the minimum output bound is unconstrained, the `ymin` input port is disconnected.

Configure the output setpoint, `ref`, and simulation duration, `Tsim`.

```
ref = 1;
Tsim = 20;
```

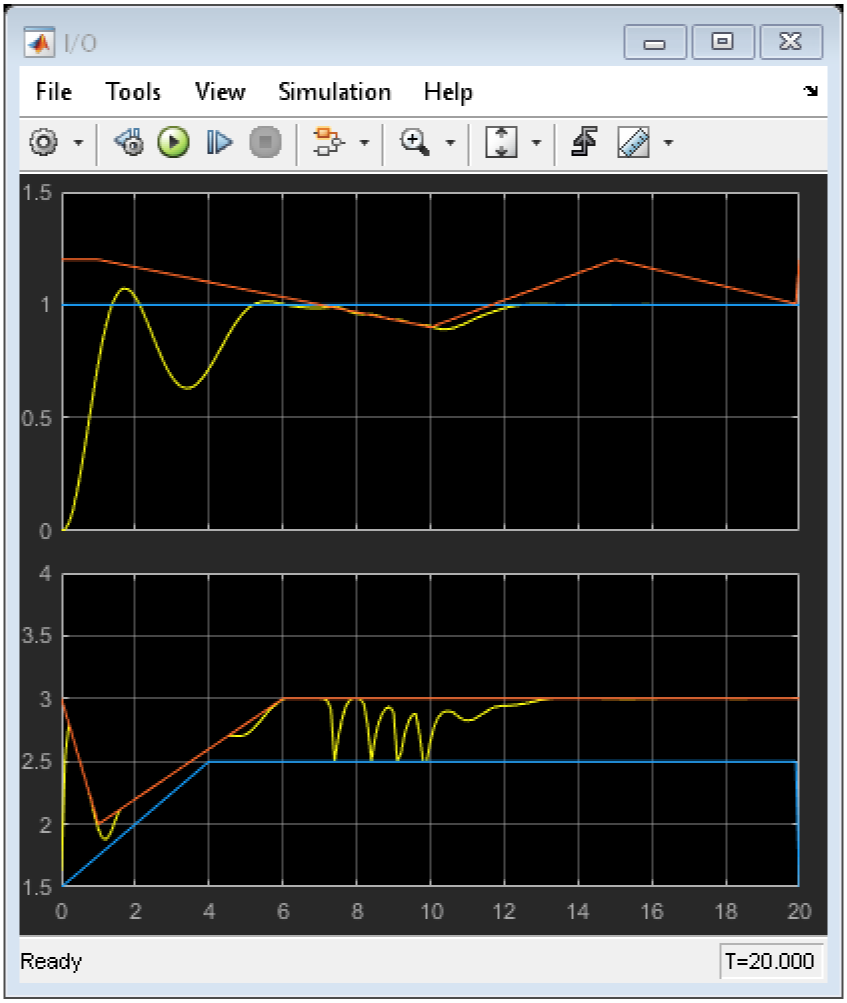
Since the `ymin` port is disconnected, disable input/output not connected warnings.

```
set_param(mdl,'UnconnectedInputMsg','off')
set_param(mdl,'UnconnectedOutputMsg','off')
```

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
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
The MPC block inport "ymin" is unconnected.  Limits specified in the controller object
```



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.

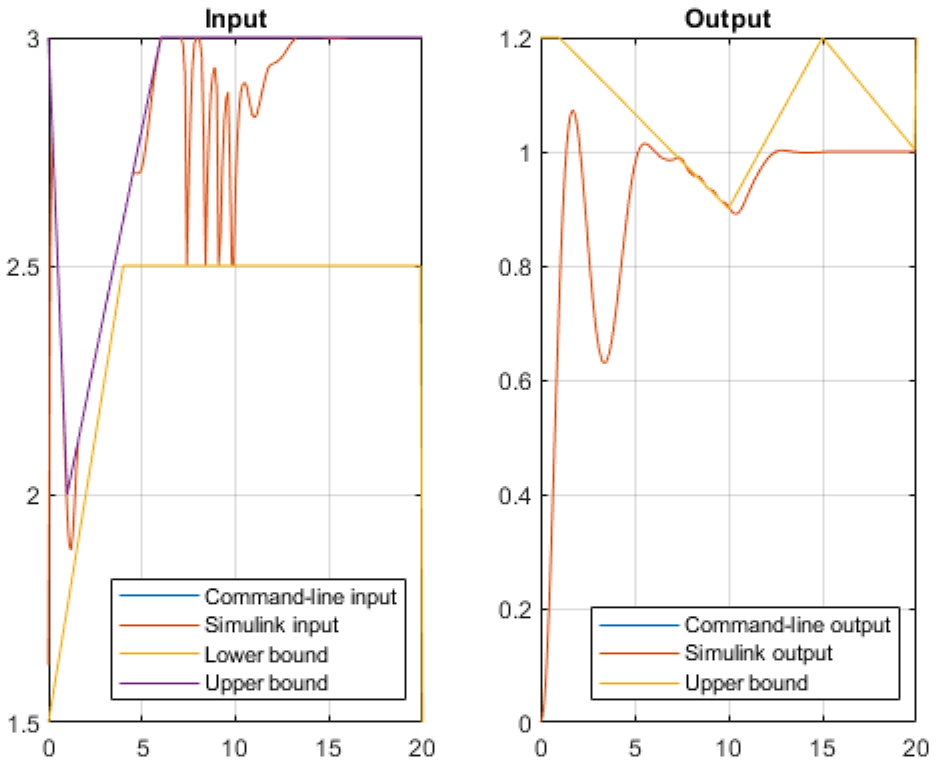
```
for t = 0:round(Tsim/Ts)
    % Store the plant state.
    XX = [XX; x];

    % 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

Adaptive MPC Design

Adaptive MPC

When to Use Adaptive MPC

MPC control predicts future behavior using a linear-time-invariant (LTI) dynamic model. In practice, such predictions are never exact, and a key tuning objective is to make MPC insensitive to prediction errors. In many applications, this approach is sufficient for robust controller performance.

If the plant is strongly nonlinear or its characteristics vary dramatically with time, LTI prediction accuracy might degrade so much that MPC performance becomes unacceptable. Adaptive MPC can address this degradation by adapting the prediction model for changing operating conditions. As implemented in the Model Predictive Control Toolbox software, adaptive MPC uses a fixed model structure, but allows the models parameters to evolve with time. Ideally, whenever the controller requires a prediction (at the beginning of each control interval) it uses a model appropriate for the current conditions.

After you design an MPC controller for the average or most likely operating conditions of your control system, you can implement an adaptive MPC controller based on that design. For information about designing that initial controller, see “Controller Creation”.

At each control interval, the adaptive MPC controller updates the plant model and nominal conditions. Once updated, the model and conditions remain constant over the prediction horizon. If you can predict how the plant and nominal conditions vary in the future, you can use “Time-Varying MPC” on page 5-47 to specify a model that changes over the prediction horizon.

An alternative option for controlling a nonlinear or time-varying plant is to use gain-scheduled MPC control. See “Gain-Scheduled MPC” on page 7-2.)

Plant Model

The plant model used as the basis for adaptive MPC must be an LTI discrete-time, state-space model. See “Basic Models” (Control System Toolbox) or “Linearization Basics” (Simulink Control Design) for information about creating and modifying such systems. The plant model structure is as follows:

$$
\begin{aligned}
x(k+1) &= Ax(k) + B_u u(k) + B_v v(k) + B_d d(k) \\
y(k) &= Cx(k) + D_v v(k) + D_d d(k).
\end{aligned}
$$

Here, the matrices A, B_u, B_v, B_d, C, D_v and D_d are the parameters that can vary with time. The other variables in the expression are:

- k — Time index (current control interval).
- x — n_x plant model states.
- u — n_u manipulated inputs (MVs). These are the one or more inputs that are adjusted by the MPC controller.
- v — n_v measured disturbance inputs.
- d — n_d unmeasured disturbance inputs.
- y — n_y plant outputs, including n_{ym} measured and n_{yu} unmeasured outputs. The total number of outputs, $n_y = n_{ym} + n_{yu}$. Also, $n_{ym} \geq 1$ (there is at least one measured output).

Additional requirements for the plant model in adaptive MPC control are:

- Sample time (`Ts`) is a constant and identical to the MPC control interval.
- Time delay (if any) is absorbed as discrete states (see, e.g., the Control System Toolbox `absorbDelay` command).
- n_x, n_u, n_y, n_d, n_{ym}, and n_{yu} are all constants.
- Adaptive MPC prohibits direct feed-through from any manipulated variable to any plant output. Thus, $D_u = 0$ in the above model.
- The input and output signal configuration remains constant.

For more details about creation of plant models for MPC control, see “Plant Specification”.

Nominal Operating Point

A traditional MPC controller includes a nominal operating point at which the plant model applies, such as the condition at which you linearize a nonlinear model to obtain the LTI approximation. The `Model.Nominal` property of the controller contains this information.

In adaptive MPC, as time evolves you should update the nominal operating point to be consistent with the updated plant model.

You can write the plant model in terms of deviations from the nominal conditions:

$$
\begin{aligned}
x(k+1) &= \bar{x} + A\left(x(k) - \bar{x}\right) + B\left(u_t(k) - \bar{u}_t\right) + \overline{\Delta x} \\
y(k) &= \bar{y} + C\left(x(k) - \bar{x}\right) + D\left(u_t(k) - \bar{u}_t\right).
\end{aligned}
$$

Here, the matrices *A*, *B*, *C*, and *D* are the parameter matrices to be updated. u_t is the combined plant input variable, comprising the *u*, *v*, and *d* variables defined above. The nominal conditions to be updated are:

- $\bar{x}$ — n_x nominal states
- $\overline{\Delta x}$ — n_x nominal state increments
- $\bar{u}_t$ — n_{ut} nominal inputs
- $\bar{y}$ — n_y nominal outputs

State Estimation

By default, MPC uses a static Kalman filter (KF) to update its controller states, which include the n_{xp} plant model states, n_d (≥ 0) disturbance model states, and n_n (≥ 0) measurement noise model states. This KF requires two gain matrices, *L* and *M*. By default, the MPC controller calculates them during initialization. They depend upon the plant, disturbance, and noise model parameters, and assumptions regarding the stochastic noise signals driving the disturbance and noise models. For more details about state estimation in traditional MPC, see “Controller State Estimation” on page 2-44.

Adaptive MPC uses a Kalman filter and adjusts the gains, *L* and *M*, at each control interval to maintain consistency with the updated plant model. The result is a linear-time-varying Kalman filter (LTVKF):

$$
\begin{aligned}
L_k &= \left(A_k P_{k|k-1} C_{m,k}^T + N\right)\left(C_{m,k} P_{k|k-1} C_{m,k}^T + R\right)^{-1} \\
M_k &= P_{k|k-1} C_{m,k}^T \left(C_{m,k} P_{k|k-1} C_{m,k}^T + R\right)^{-1} \\
P_{k+1|k} &= A_k P_{k|k-1} A_k^T - \left(A_k P_{k|k-1} C_{m,k}^T + N\right) L_k^T + Q.
\end{aligned}
$$

Here, *Q*, *R*, and *N* are constant covariance matrices defined as in MPC state estimation. A_k and $C_{m,k}$ are state-space parameter matrices for the entire controller state, defined as for traditional MPC but with the portions affected by the plant model updated to time k. The value $P_{k|k-1}$ is the state estimate error covariance matrix at time k based on information available at time k–1. Finally, L_k and M_k are the updated KF gain matrices. For details on the KF formulation used in traditional MPC, see “Controller State Estimation” on page 2-44. By default, the initial condition, $P_{0|-1}$, is the static KF solution prior to any model updates.

The KF gain and the state error covariance matrix depend upon the model parameters and the assumptions leading to the constant Q, R, and N matrices. If the plant model is constant, the expressions for L_k and M_k converge to the equivalent static KF solution used in traditional MPC.

The equations for the controller state evolution at time k are identical to the KF formulation of traditional MPC described in “Controller State Estimation” on page 2-44, but with the estimator gains and state space matrices updated to time k.

You have the option to update the controller state using a procedure external to the MPC controller, and then supply the updated state to MPC at each control instant, k. In this case, the MPC controller skips all KF and LTVKF calculations.

See Also

More About

- “Model Updating Strategy” on page 5-6
- “Controller State Estimation” on page 2-44
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

Model Updating Strategy

Overview

Typically, to implement “Adaptive MPC” on page 5-2 control, you can use one of the following model-updating strategies:

- **Successive linearization** — Given a mechanistic plant model, for example a set of nonlinear ordinary differential and algebraic equations, derive its LTI approximation at the current operating condition. For example, Simulink Control Design™ software provides linearization tools for this purpose.
- **Using a Linear Parameter Varying (LPV) model** — Control System Toolbox software provides a LPV System Simulink block that allows you to specify an array of LTI models with scheduling parameters. You can perform batch linearization offline to obtain an array of plant models at the desired operating points and then use them in the LPV System block to provide model updating to the Adaptive MPC Controller Simulink block.
- **Online parameter estimation** — Given an empirical model structure and initial estimates of its parameters, use the available real-time plant measurements to estimate the current model parameters. For example, the System Identification Toolbox™ software provides real-time parameter estimation tools.

To implement “Time-Varying MPC” on page 5-47 control, you need to obtain LTI plants for the future prediction horizon steps. In this case, you can use the successive linearization and LPV model approaches as long as each model is a function of time

Other Considerations

There are several factors to keep in mind when designing and implementing an adaptive MPC controller.

- Before attempting adaptive MPC, define and tune an MPC controller for the most typical (nominal) operating condition. Make sure the system can tolerate some prediction error. Test this tolerance via simulations in which the MPC prediction model differs from the plant. See “MPC Design”.
- An adaptive MPC controller requires more real-time computations than traditional MPC. In addition to the state estimation calculation, you must also implement and test a model-updating strategy, which might be computationally intensive.

- You must determine MPC tuning constants that provide robust performance over the expected range of model parameters. See “Tune Weights” on page 1-16.
- Model updating via online parameter estimation is most effective when parameter variations occur gradually.
- When implementing adaptive MPC control, adapt only parameters defining the `Model.Plant` property of the controller. The disturbance and noise models, if any, remain constant.

See Also

Adaptive MPC Controller

More About

- “Adaptive MPC” on page 5-2
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” on page 5-8
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

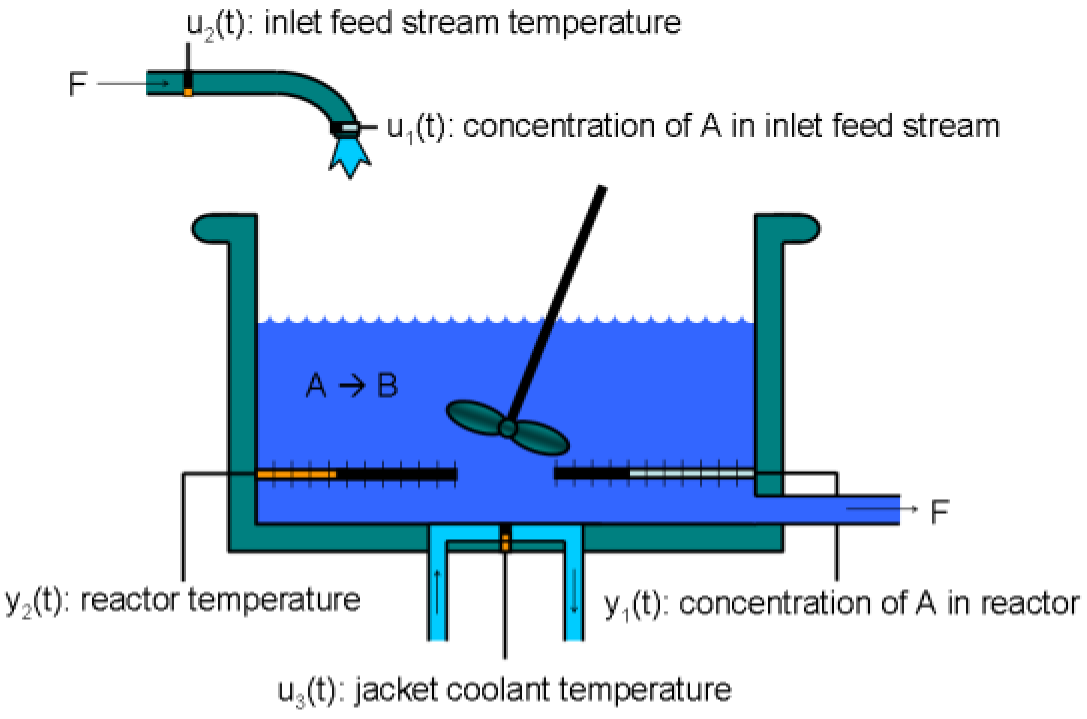
Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization

This example shows how to use an Adaptive MPC controller to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

A first principle nonlinear plant model is available and being linearized at each control interval. The adaptive MPC controller then updates its internal predictive model with the linearized plant model and achieves nonlinear control successfully.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:


u₂(t): inlet feed stream temperature
F
u₁(t): concentration of A in inlet feed stream
A → B
F
y₂(t): reactor temperature
y₁(t): concentration of A in reactor
u₃(t): jacket coolant temperature

This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{array}{ll} u_1 = CA_i & \text{Concentration of A in inlet feed stream}[kgmol/m^3] \\ u_2 = T_i & \text{Inlet feed stream temperature}[K] \\ u_3 = T_c & \text{Jacket coolant temperature}[K] \end{array}$$

and the outputs (y(t)), which are also the states of the model (x(t)), are:

$$\begin{array}{ll} y_1 = x_1 = CA & \text{Concentration of A in reactor tank}[kgmol/m^3] \\ y_2 = x_2 = T & \text{Reactor temperature}[K] \end{array}$$

The control objective is to maintain the concentration of reagent A, CA at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference as well as reject the measured disturbance arising from the inlet feed stream temperature T_i. The inlet feed stream concentration, CA_i, is assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

We also assume that direct measurements of concentrations are unavailable or infrequent, which is the usual case in practice. Instead, we use a "soft sensor" to estimate CA based on temperature measurements and the plant model.

About Adaptive Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical operating range. Next you can choose one of the two approaches to implement MPC control strategy:

(1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based on a desired scheduling strategy. See “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor” for more details. Use this approach when the plant models have different orders or time delays.

(2) Design one MPC controller offline at the initial operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.

- If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:

(1) Use successive linearization as shown in this example. Use this approach when a nonlinear plant model is available and can be linearized at run time.

(2) Use online estimation to identify a linear model when loop is closed. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” for more details. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

To linearize the plant, Simulink® and Simulink Control Design® are required.

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

To implement an adaptive MPC controller, first you need to design a MPC controller at the initial operating point where CAi is 10 kgmol/m^3, Ti and Tc are 298.15 K.

Create operating point specification.

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

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

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

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

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

Operating point specifications were successfully met.
States:
----------
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           311      dx:      8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:          8.57      dx:     -6.87e-12 (0)

Inputs:
----------
(1.) mpc_cstr_plant/CAi
```

```
      u:            10
(2.) mpc_cstr_plant/Ti
      u:           298
(3.) mpc_cstr_plant/Tc
      u:           298

Outputs:
----------
(1.) mpc_cstr_plant/T
      y:           311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:          8.57    [-Inf Inf]
```

Obtain nominal values of x, y and u.

```
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];
```

Obtain linear plant model at the initial condition.

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

Drop the first plant input CAi because it is not used by MPC.

```
sys = sys(:,2:3);
```

Discretize the plant model because Adaptive MPC controller only accepts a discrete-time plant model.

```
Ts = 0.5;
plant = c2d(sys,Ts);
```

Design MPC Controller

You design an MPC at the initial operating condition. When running in the adaptive mode, the plant model is updated at run time.

Specify signal types used in MPC.

```
plant.InputGroup.MeasuredDisturbances = 1;
plant.InputGroup.ManipulatedVariables = 2;
plant.OutputGroup.Measured = 1;
plant.OutputGroup.Unmeasured = 2;
plant.InputName = {'Ti','Tc'};
plant.OutputName = {'T','CA'};
```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant);
```

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

Set nominal values in the controller

```
mpcobj.Model.Nominal = struct('X', x0, 'U', u0(2:3), 'Y', y0, 'DX', [0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude

```
Uscale = [30 50];
Yscale = [50 10];
mpcobj.DV(1).ScaleFactor = Uscale(1);
mpcobj.MV(1).ScaleFactor = Uscale(2);
mpcobj.OV(1).ScaleFactor = Yscale(1);
mpcobj.OV(2).ScaleFactor = Yscale(2);
```

Let reactor temperature T float (i.e. with no setpoint tracking error penalty), because the objective is to control reactor concentration CA and only one manipulated variable (coolant temperature Tc) is available.

```
mpcobj.Weights.OV = [0 1];
```

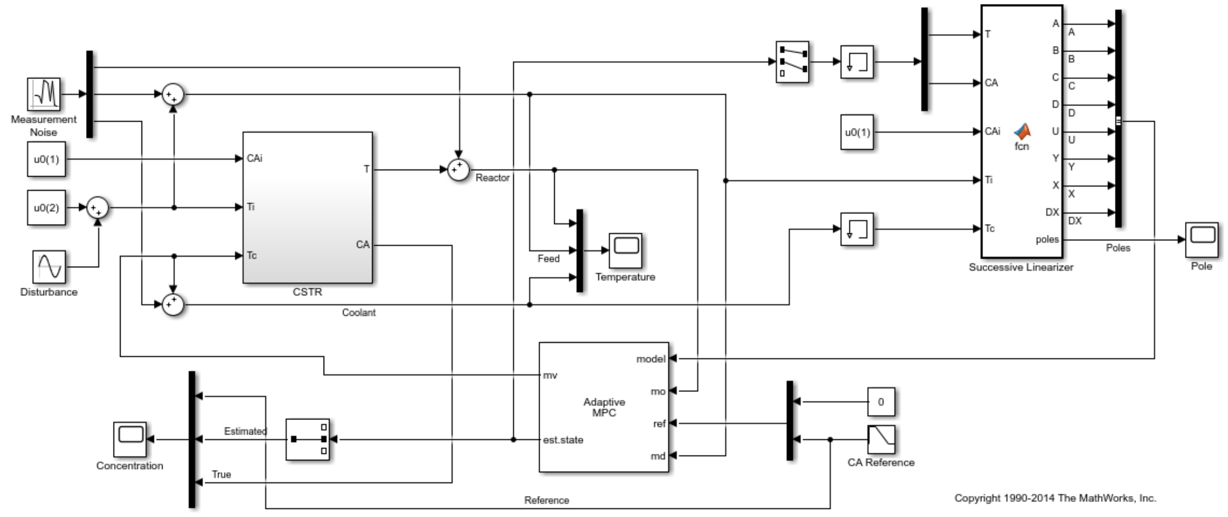
Due to the physical constraint of coolant jacket, Tc rate of change is bounded by degrees per minute.

```
mpcobj.MV.RateMin = -2;
mpcobj.MV.RateMax = 2;
```

Implement Adaptive MPC Control of CSTR Plant in Simulink (R)

Open the Simulink model.

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



The model includes three parts:

1. The "CSTR" block implements the nonlinear plant model.
2. The "Adaptive MPC Controller" block runs the designed MPC controller in the adaptive mode.
3. The "Successive Linearizer" block in a MATLAB Function block that linearizes a first principle nonlinear CSTR plant and provides the linear plant model to the "Adaptive MPC Controller" block at each control interval. Double click the block to see the MATLAB code. You can use the block as a template to develop appropriate linearizer for your own applications.

Note that the new linear plant model must be a discrete time state space system with the same order and sample time as the original plant model has. If the plant has time delay, it must also be same as the original time delay and absorbed into the state space model.

Validate Adaptive MPC Control Performance

Controller performance is validated against both setpoint tracking and disturbance rejection.

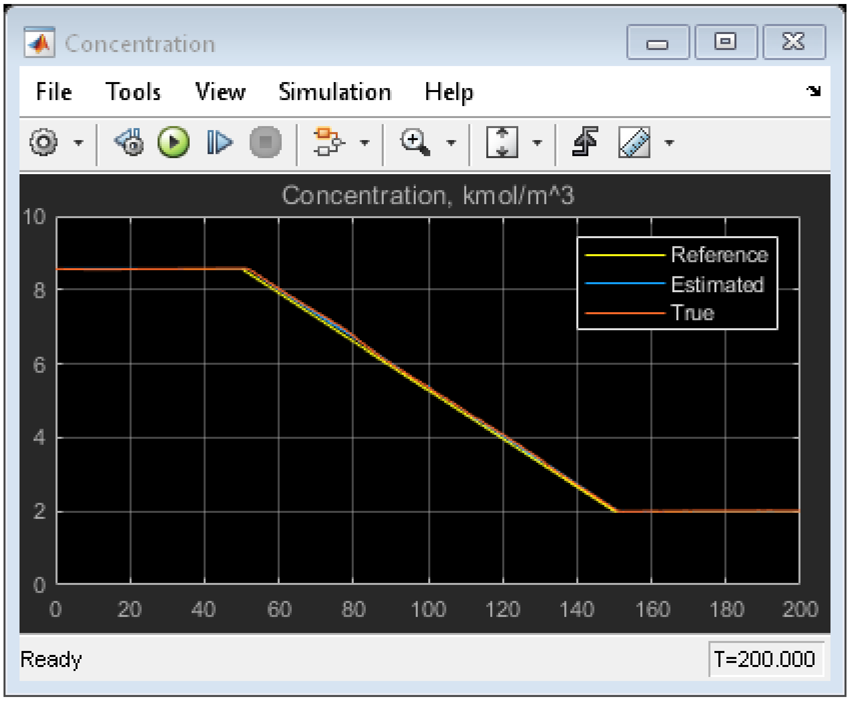
- Tracking: reactor concentration CA setpoint transitions from original 8.57 (low conversion rate) to 2 (high conversion rate) kgmol/m^3. During the transition, the plant first becomes unstable then stable again (see the poles plot).

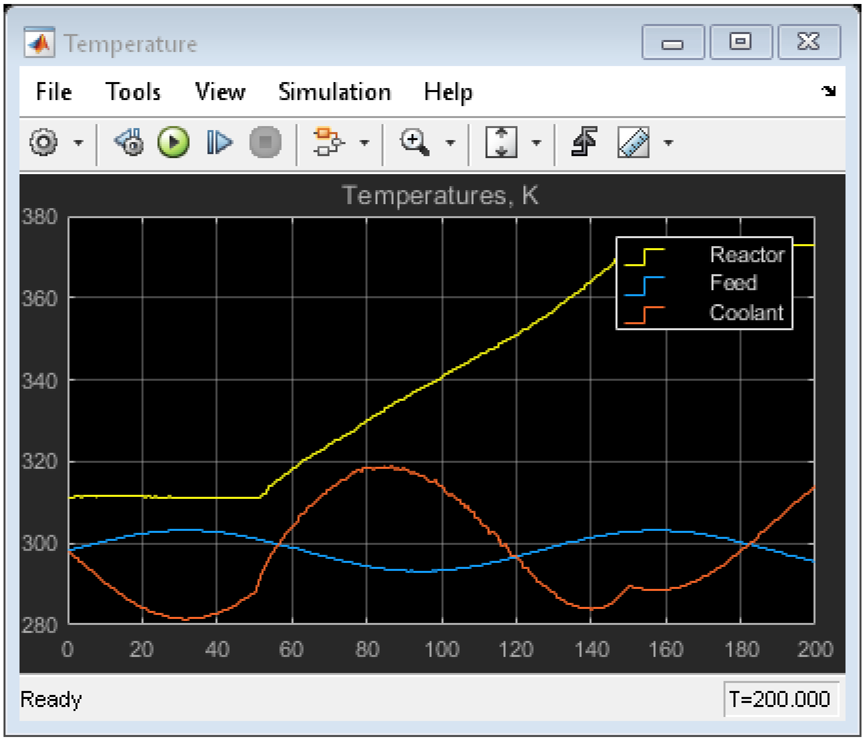
- Regulating: feed temperature Ti has slow fluctuation represented by a sine wave with amplitude of 5 degrees, which is a measured disturbance fed to the MPC controller.

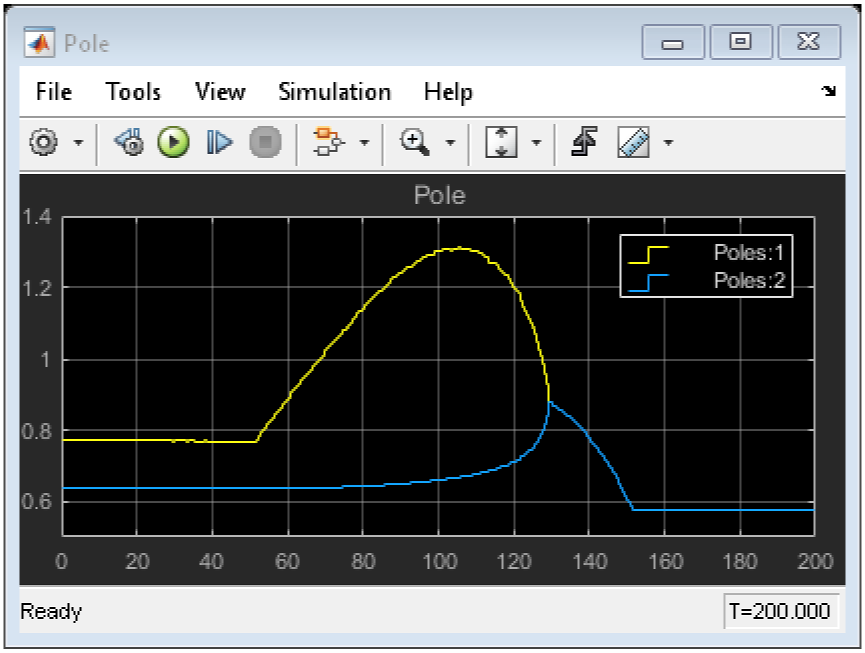
Simulate the closed-loop performance.

```
open_system([mdl '/Concentration'])
open_system([mdl '/Temperature'])
open_system([mdl '/Pole'])
sim(mdl);
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```




Temperature
File
Tools
View
Simulation
Help
Temperatures, K
380
360
340
320
300
280
0
20
40
60
80
100
120
140
160
180
200
Reactor
Feed
Coolant
Ready
T=200.000



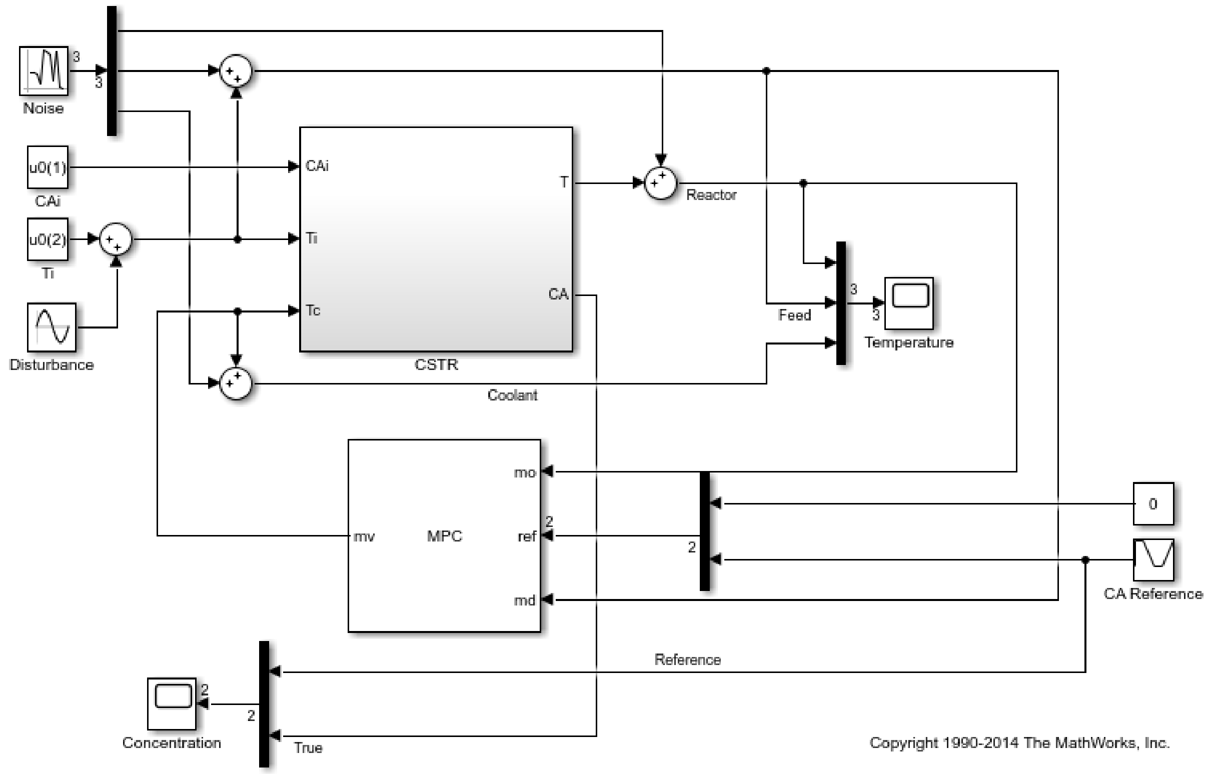
```
bdclose(mdl);
```

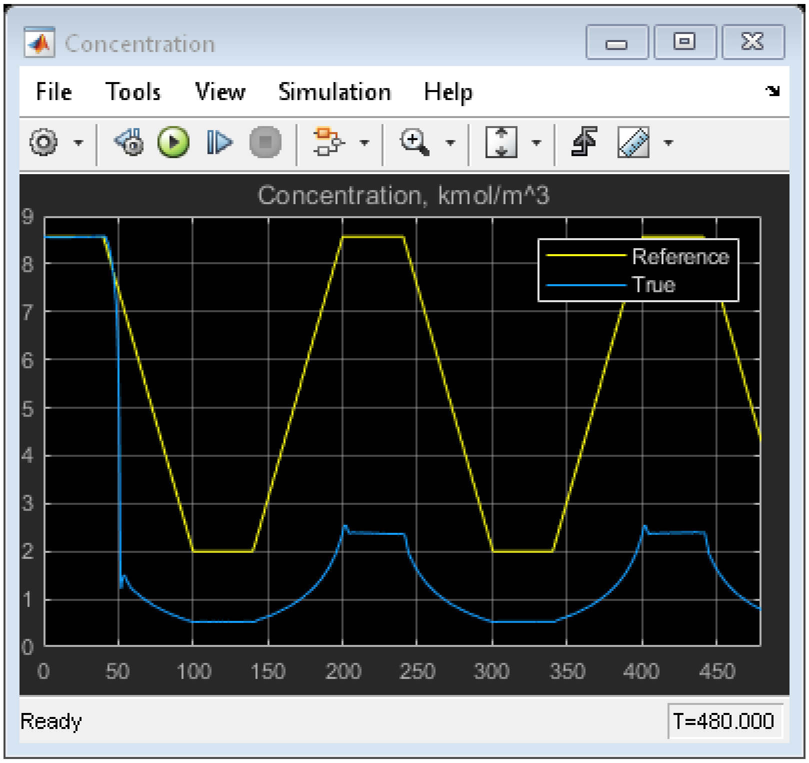
The tracking and regulating performance is very satisfactory. In an application to a real reactor, however, model inaccuracies and unmeasured disturbances could cause poorer tracking than shown here. Additional simulations could be used to study these effects.

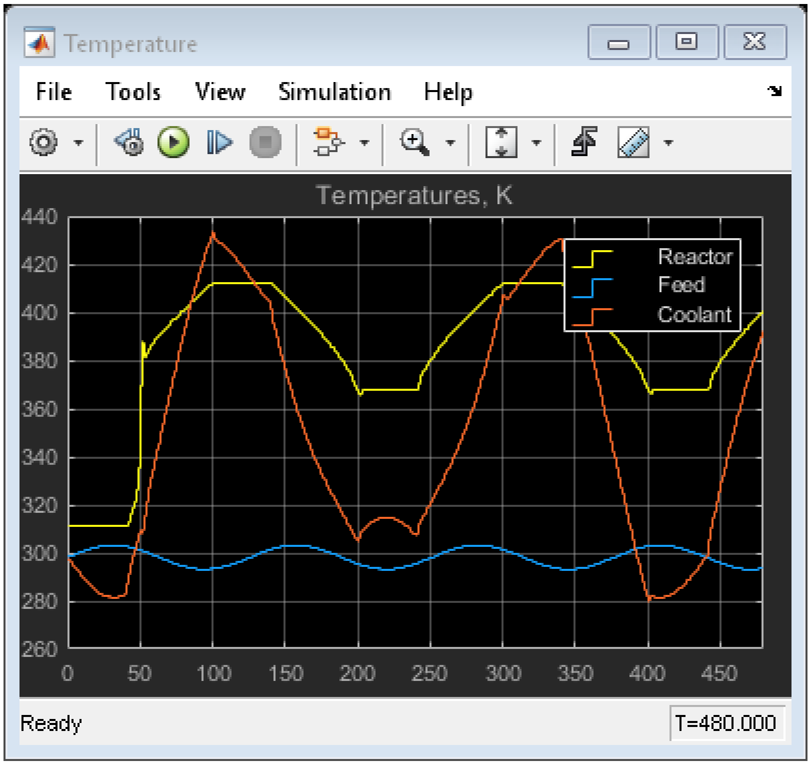
Compare with Non-Adaptive MPC Control

Adaptive MPC provides superior control performance than a non-adaptive MPC. To illustrate this point, the control performance of the same MPC controller running in the non-adaptive mode is shown below. The controller is implemented with a MPC Controller block.

```
mdl1 = 'ampc_cstr_no_linearization';
open_system(mdl1);
open_system([mdl1 '/Concentration'])
open_system([mdl1 '/Temperature'])
sim(mdl1);
```


3
3
Noise
u0(1)
CAi
u0(2)
Ti
Disturbance
CAi
Ti
Tc
T
CA
CSTR
Reactor
Feed
3
3
Temperature
Coolant
mo
mv
MPC
2
ref
md
2
0
CA Reference
Reference
2
2
Concentration
True
Copyright 1990-2014 The MathWorks, Inc.


Concentration
File
Tools
View
Simulation
Help
Concentration, kmol/m^3
Reference
True
0
1
2
3
4
5
6
7
8
9
0
50
100
150
200
250
300
350
400
450
Ready
T=480.000



As expected, the tracking and regulating performance is unacceptable.

```
bdclose(mdl1)
```

See Also

Adaptive MPC Controller

More About

- “Adaptive MPC” on page 5-2
- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” on page 5-21

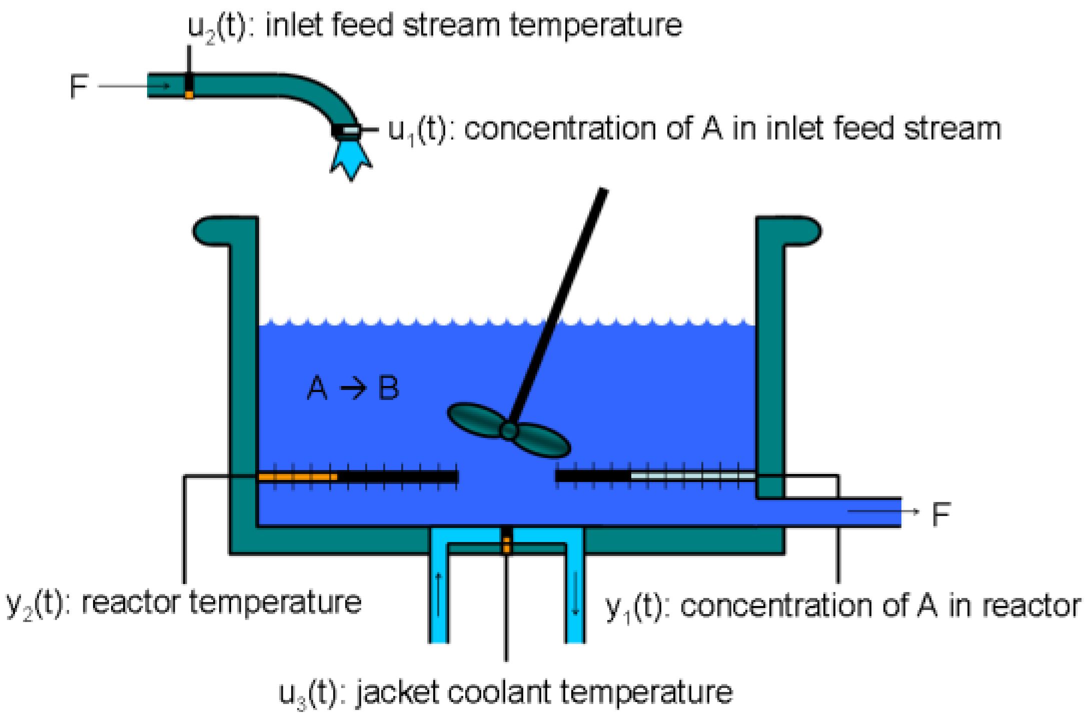
Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation

This example shows how to use an Adaptive MPC controller to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

A discrete time ARX model is being identified online by the Recursive Polynomial Model Estimator block at each control interval. The adaptive MPC controller uses it to update internal plant model and achieves nonlinear control successfully.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:


$u_2(t)$: inlet feed stream temperature
F
$u_1(t)$: concentration of A in inlet feed stream
A → B
F
$y_2(t)$: reactor temperature
$y_1(t)$: concentration of A in reactor
$u_3(t)$: jacket coolant temperature

This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{array}{ll} u_1 = CA_i & \text{Concentration of A in inlet feed stream}[kgmol/m^3] \\ u_2 = T_i & \text{Inlet feed stream temperature}[K] \\ u_3 = T_c & \text{Jacket coolant temperature}[K] \end{array}$$

and the outputs (y(t)), which are also the states of the model (x(t)), are:

$$\begin{array}{ll} y_1 = x_1 = CA & \text{Concentration of A in reactor tank}[kgmol/m^3] \\ y_2 = x_2 = T & \text{Reactor temperature}[K] \end{array}$$

The control objective is to maintain the reactor temperature T at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference as well as reject the measured disturbance arising from the inlet feed stream temperature T_i. The inlet feed stream concentration, CA_i, is assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

About Adaptive Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical operating range. Next you can choose one of the two approaches to implement MPC control strategy:

(1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based on a desired scheduling strategy. See “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor” for more details. Use this approach when the plant models have different orders or time delays.

(2) Design one MPC controller offline at the initial operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.

- If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:

(1) Use successive linearization. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” for more details. Use this approach when a nonlinear plant model is available and can be linearized at run time.

(2) Use online estimation to identify a linear model when loop is closed, as shown in this example. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

To linearize the plant, Simulink® and Simulink Control Design™ are required.

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

To implement an adaptive MPC controller, first you need to design a MPC controller at the initial operating point where CAi is 10 kgmol/m^3, Ti and Tc are 298.15 K.

Create operating point specification.

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

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

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

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

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

Operating point specifications were successfully met.
States:
----------
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           311      dx:      8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:          8.57      dx:     -6.87e-12 (0)

Inputs:
----------
(1.) mpc_cstr_plant/CAi
      u:            10
(2.) mpc_cstr_plant/Ti
      u:           298
(3.) mpc_cstr_plant/Tc
      u:           298
```

```
Outputs:
----------
(1.) mpc_cstr_plant/T
      y:           311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:          8.57    [-Inf Inf]
```

Obtain nominal values of x, y and u.

```
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = [op_report.Outputs(1).y;op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u;op_report.Inputs(2).u;op_report.Inputs(3).u];
```

Obtain linear plant model at the initial condition.

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

Drop the first plant input CAi and second output CA because they are not used by MPC.

```
sys = sys(1,2:3);
```

Discretize the plant model because Adaptive MPC controller only accepts a discrete-time plant model.

```
Ts = 0.5;
plant = c2d(sys,Ts);
```

Design MPC Controller

You design an MPC at the initial operating condition. When running in the adaptive mode, the plant model is updated at run time.

Specify signal types used in MPC.

```
plant.InputGroup.MeasuredDisturbances = 1;
plant.InputGroup.ManipulatedVariables = 2;
plant.OutputGroup.Measured = 1;
plant.InputName = {'Ti','Tc'};
plant.OutputName = {'T'};
```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant);
```

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

Set nominal values in the controller

```
mpcobj.Model.Nominal = struct('X', x0, 'U', u0(2:3), 'Y', y0(1), 'DX', [0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude

```
Uscale = [30 50];
Yscale = 50;
mpcobj.DV.ScaleFactor = Uscale(1);
mpcobj.MV.ScaleFactor = Uscale(2);
mpcobj.OV.ScaleFactor = Yscale;
```

Due to the physical constraint of coolant jacket, Tc rate of change is bounded by 2 degrees per minute.

```
mpcobj.MV.RateMin = -2;
mpcobj.MV.RateMax = 2;
```

Reactor concentration is not directly controlled in this example. If reactor temperature can be successfully controlled, the concentration will achieve desired performance requirement due to the strongly coupling between the two variables.

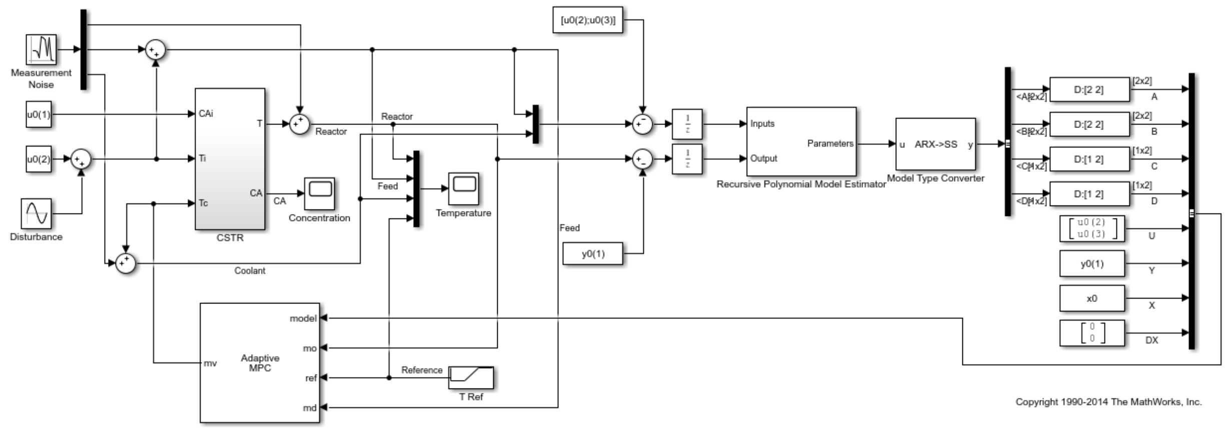
Implement Adaptive MPC Control of CSTR Plant in Simulink (R)

To run this example with online estimation, System Identification Toolbox™ software is required.

```
if ~mpcchecktoolboxinstalled('ident')
    disp('System Identification Toolbox(TM) is required to run this example.')
    return
end
```

Open the Simulink model.

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



The model includes three parts:

1. The "CSTR" block implements the nonlinear plant model.
2. The "Adaptive MPC Controller" block runs the designed MPC controller in the adaptive mode.
3. The "Recursive Polynomial Model Estimator" block estimates a two-input (Ti and Tc) and one-output (T) discrete time ARX model based on the measured temperatures. The estimated model is then converted into state space form by the "Model Type Converter" block and fed to the "Adaptive MPC Controller" block at each control interval.

In this example, the initial plant model is used to initialize the online estimator with parameter covariance matrix set to 1. The online estimation method is "Kalman Filter" with noise covariance matrix set to 0.01. The online estimation result is sensitive to these parameters and you can further adjust them to achieve better estimation result.

Both "Recursive Polynomial Model Estimator" and "Model Type Converter" are provided by System Identification Toolbox. You can use the two blocks as a template to develop appropriate online model estimation for your own applications.

The initial value of A(q) and B(q) variables are populated with the numerator and denominator of the initial plant model.

```
[num, den] = tfdata(plant);
Aq = den{1};
Bq = num;
```

Note that the new linear plant model must be a discrete time state space system with the same order and sample time as the original plant model has. If the plant has time delay, it must also be same as the original time delay and absorbed into the state space model.

Validate Adaptive MPC Control Performance

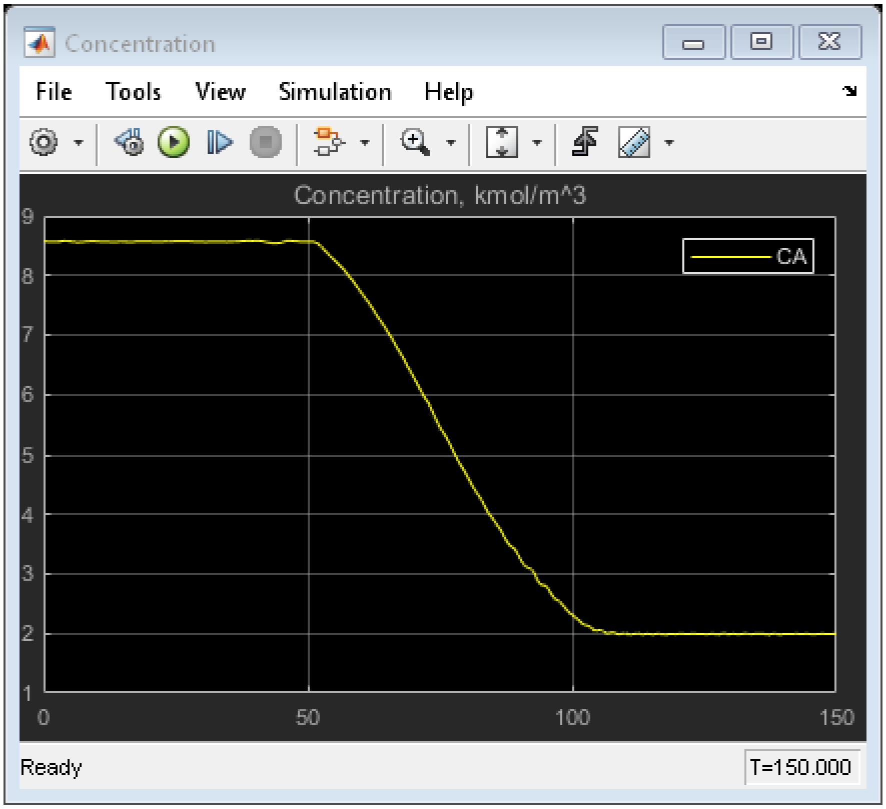
Controller performance is validated against both setpoint tracking and disturbance rejection.

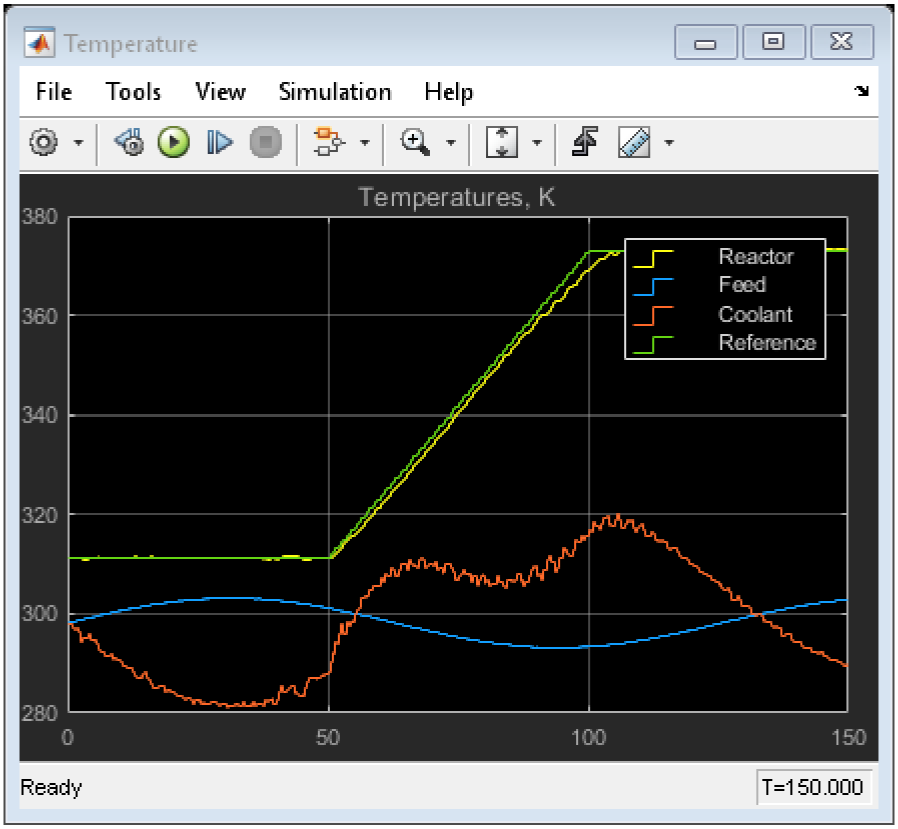
- Tracking: reactor temperature T setpoint transitions from original 311 K (low conversion rate) to 377 K (high conversion rate) kgmol/m^3.
- Regulating: feed temperature Ti has slow fluctuation represented by a sine wave with amplitude of 5 degrees, which is a measured disturbance fed to MPC controller.

Simulate the closed-loop performance.

```
open_system([mdl '/Concentration'])
open_system([mdl '/Temperature'])
sim(mdl);
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```


Concentration
File
Tools
View
Simulation
Help
Concentration, kmol/m^3
CA
9
8
7
6
5
4
3
2
1
0
50
100
150
Ready
T=150.000

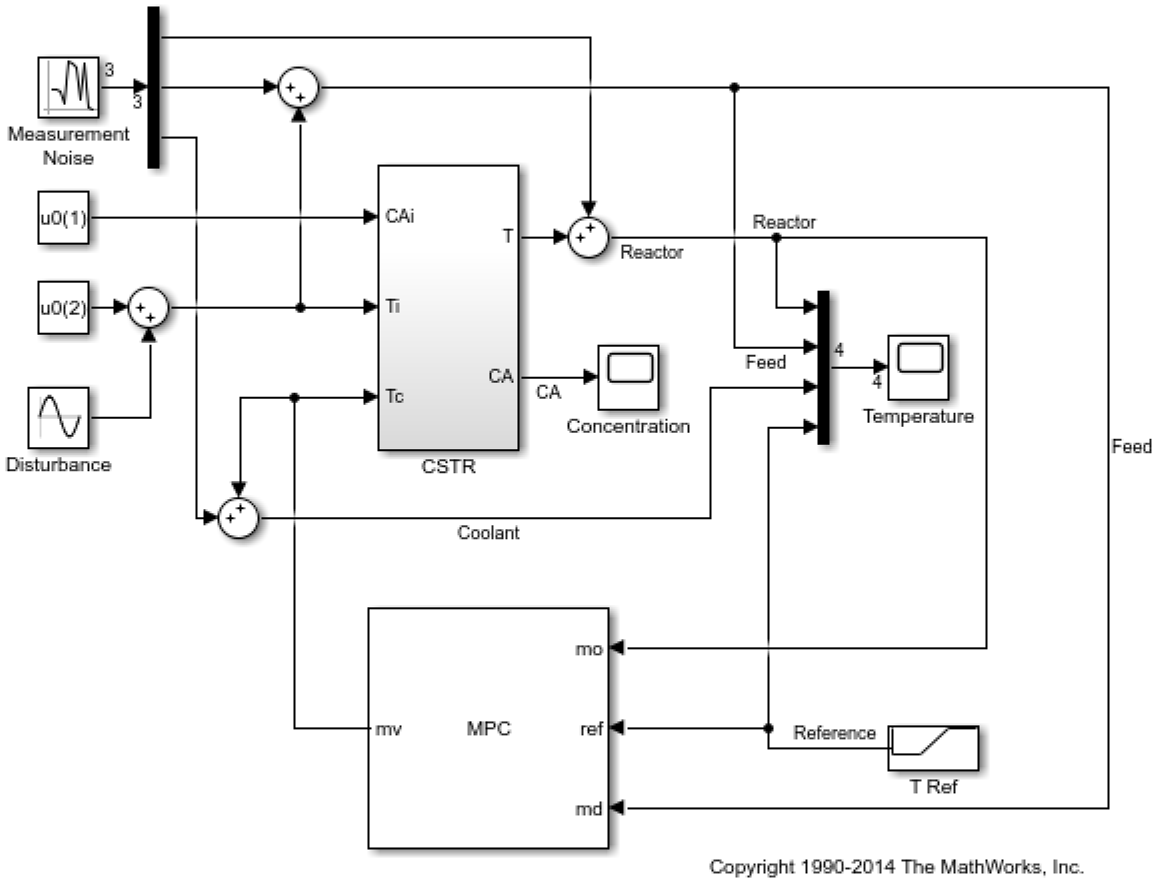


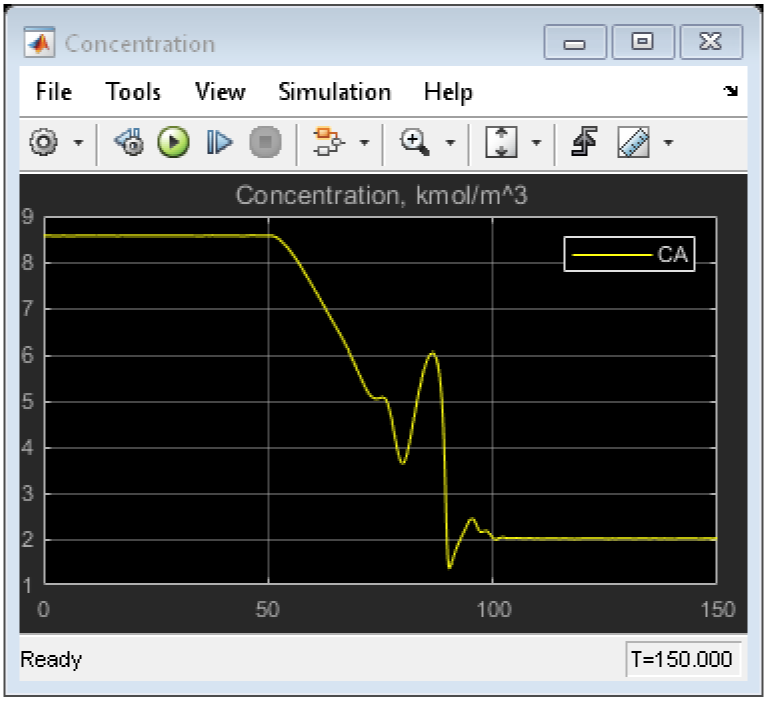
The tracking and regulating performance is very satisfactory.

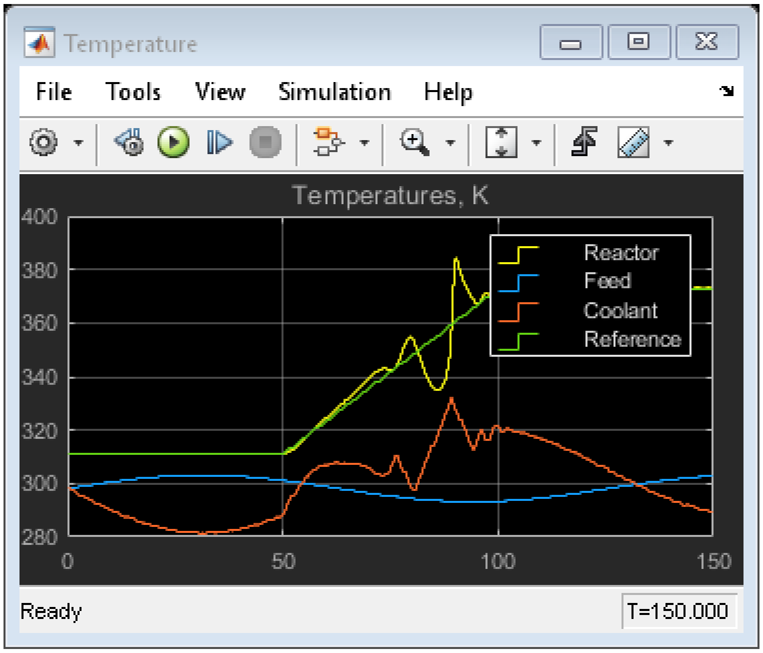
Compare with Non-Adaptive MPC Control

Adaptive MPC provides superior control performance than non-adaptive MPC. To illustrate this point, the control performance of the same MPC controller running in the non-adaptive mode is shown below. The controller is implemented with a MPC Controller block.

```
mdl1 = 'ampc_cstr_no_estimation';
open_system(mdl1);
open_system([mdl1 '/Concentration'])
open_system([mdl1 '/Temperature'])
sim(mdl1);
```


3
3
Measurement Noise
u0(1)
u0(2)
Disturbance
CAi
Ti
Tc
T
CA
CSTR
Reactor
Reactor
CA
Concentration
Feed
4
4
Temperature
Feed
Coolant
mo
mv
MPC
ref
md
Reference
T Ref
Copyright 1990-2014 The MathWorks, Inc.


Concentration
File
Tools
View
Simulation
Help
Concentration, kmol/m^3
CA
9
8
7
6
5
4
3
2
1
0
50
100
150
Ready
T=150.000



As expected, the tracking and regulating performance is unacceptable.

```
bdclose(mdl)
bdclose(mdl1)
```

See Also

Adaptive MPC Controller

More About

- "Adaptive MPC" on page 5-2
- "Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization" on page 5-8

Obstacle Avoidance Using Adaptive Model Predictive Control

This example shows how to make a vehicle (ego car) follow a reference velocity and avoid obstacles in the lane using adaptive MPC. To do so, you update the plant model and linear mixed input/output constraints at run time.

Obstacle Avoidance

A vehicle with obstacle avoidance (or passing assistance) has a sensor, such as lidar, that measures the distance to an obstacle in front of the vehicle and in the same lane. The obstacle can be static, such as a large pot hole, or moving, such as a slow-moving vehicle. The most common maneuver from the driver is to temporarily move to another lane, drive past the obstacle, and move back to the original lane afterward.

As part of the autonomuous driving experience, an obstacle avoidance system can perform the maneuver without human intervention. In this example, you design an obstacle avoidance system that moves the ego car around a static obstacle in the lane using throttle and steering angle. This system uses an adaptive model predictive controller that updates both the predictive model and the mixed input/output constraints at each control interval.

Vehicle Model

The ego car has a rectangular shape with a length of 5 meters and width of 2 meters. The model has four states:

- x - Global X position of the car center
- y - Global Y position of the car center
- θ - Heading angle of the car (`0` when facing east, counterclockwise positive)
- v - Speed of the car (positive)

There are two manipulated variables:

- T - Throttle (positive when accelerating, negative when decelerating)
- δ - Steering angle (0 when aligned with car, counterclockwise positive)

Use a simple nonlinear model to describe the dynamics of the ego car:

$$\begin{aligned}
\dot{x} &= -v\sin(\theta)\cdot\theta + \cos(\theta)\cdot v \\
\dot{y} &= v\cos(\theta)\cdot\theta + \sin(\theta)\cdot v \\
\dot{\theta} &= (\tan(\delta)/C_L)\cdot v + \left(v\left(\tan(\delta)^2+1\right)/C_L\right)\cdot\delta \\
\dot{v} &= 0.5\cdot T
\end{aligned}$$

where C_L is the car length.

Also, assume all the states are measurable. At the nominal operating point, the ego car drives east at a constant speed of 20 meters per second.

```
V = 20;
x0 = [0; 0; 0; V];
u0 = [0; 0];
```

Obtain a linear plant model at the nominal operating point and convert it into a discrete-time model to be used by the model predictive controller.

```
Ts = 0.02;
[Ad,Bd,Cd,Dd,U,Y,X,DX] = obstacleVehicleModelDT(Ts,x0,u0);
dsys = ss(Ad,Bd,Cd,Dd,'Ts',Ts);
dsys.InputName = {'Throttle','Delta'};
dsys.StateName = {'X','Y','Theta','V'};
dsys.OutputName = dsys.StateName;
```

Road and Obstacle Information

In this example, assume that:

- The road is straight and has 3 lanes.
- Each lane is 4 meters wide.
- The ego car drives in the middle of the center lane when not passing.
- Without losing generality, the ego car passes an obstacle only from the left (fast) lane.

```
lanes = 3;
laneWidth = 4;
```

The obstacle in this example is a nonmoving object in the middle of the center lane with the same size as the ego car.

```
obstacle = struct;
obstacle.Length = 5;
obstacle.Width = 2;
```

Place the obstacle `50` meters down the road.

```
obstacle.X = 50;
obstacle.Y = 0;
```

Create a virtual safe zone around the obstacle so that the ego car does not get too close to the obstacle when passing it. The safe zone is centered on the obstacle and has a:

- Length equal to two car lengths.
- Width equal to two lane widths.

```
obstacle.safeDistanceX = obstacle.Length;
obstacle.safeDistanceY = laneWidth;
obstacle = obstacleGenerateObstacleGeometryInfo(obstacle);
```

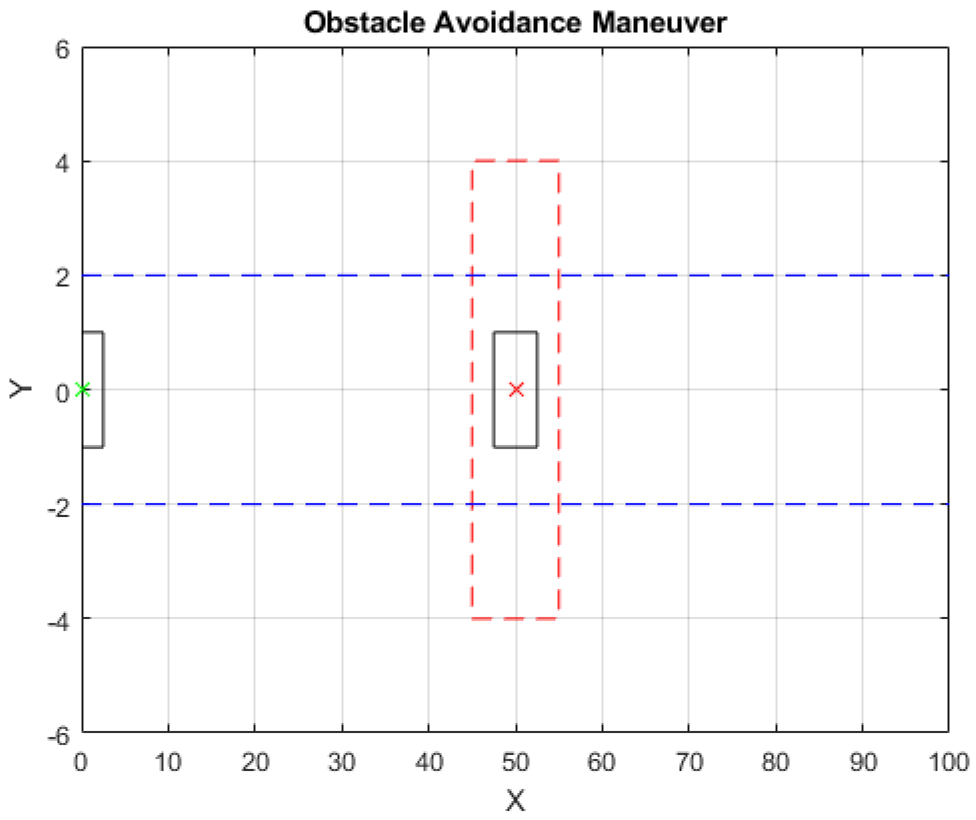
In this example, assume that the lidar device can detect an obstacle `30` meters in front of the vehicle.

```
obstacle.DetectionDistance = 30;
```

Plot the following at the nominal condition:

- Ego car - Green dot with black boundary
- Horizonal lanes - Dashed blue lines
- Obstacle - Red `x` with black boundary
- Safe zone - Dashed red boundary.

```
f = obstaclePlotInitialCondition(x0,obstacle,laneWidth,lanes);
```



MPC Design at the Nominal Operating Point

Design a model predictive controller that can make the ego car maintain a desired velocity and stay in the middle of the center lane.

```
status = mpcverbosity('off');
mpcobj = mpc(dsys);
```

The prediction horizon is 25 steps, which is equivalent to 0.5 seconds.

```
mpcobj.PredictionHorizon = 25;
mpcobj.ControlHorizon = 5;
```

To prevent the ego car from accelerating or decelerating too quickly, add a hard constraint of 0.2 (m^2/sec) on the throttle rate of change.

```
mpcobj.ManipulatedVariables(1).RateMin = -0.2;
mpcobj.ManipulatedVariables(1).RateMax = 0.2;
```

Similarly, add a hard constraint of 6 degrees per sec on the steering angle rate of change.

```
mpcobj.ManipulatedVariables(2).RateMin = -pi/30;
mpcobj.ManipulatedVariables(2).RateMax = pi/30;
```

Scale the throttle and steering angle by their respective operating ranges.

```
mpcobj.ManipulatedVariables(1).ScaleFactor = 2;
mpcobj.ManipulatedVariables(2).ScaleFactor = 0.2;
```

Since there are only two manipulated variables, to achieve zero steady-state offset, you can choose only two outputs for perfect tracking. In this example, choose the Y position and velocity by setting the weights of the other two outputs (X and theta) to zero. Doing so lets the values of these other outputs float.

```
mpcobj.Weights.OutputVariables = [0 1 0 1];
```

Update the controller with the nominal operating condition. For a discrete-time plant:

- `U = u0`
- `X = x0`
- `Y = Cd*x0 + Dd*u0`
- `DX = Ad*X0 + Bd*u0 - x0`

```
mpcobj.Model.Nominal = struct('U',U,'Y',Y,'X',X,'DX',DX);
```

Specify Mixed I/O Constraints for Obstacle Avoidance Maneuver

There are different strategies to make the ego car avoid an obstacle on the road. For example, a real-time path planner can compute a new path after an obstacle is detected and the controller follows this path.

In this example, use a different approach that takes advantage of the ability of MPC to handle constraints explicitly. When an obstacle is detected, it defines an area on the road (in terms of constraints) that the ego car must not enter during the prediction horizon. At the next control interval, the area is redefined based on the new positions of the ego car and obstacle until passing is completed.

To define the area to avoid, use the following mixed input/output constraints:

```
E*u + F*y <= G
```

where u is the manipulated variable vector and y is the output variable vector. You can update the constraint matrices E, F, and G when the controller is running.

The first constraint is an upper bound on y ($y \leq 6$ on this three-lane road).

```
E1 = [0 0];
F1 = [0 1 0 0];
G1 = laneWidth*lanes/2;
```

The second constraint is a lower bound on y ($y \geq -6$ on this three-lane road).

```
E2 = [0 0];
F2 = [0 -1 0 0];
G2 = laneWidth*lanes/2;
```

The third constraint is for obstacle avoidance. Even though no obstacle is detected at the nominal operating condition, you must add a "fake" constraint here because you cannot change the dimensions of the constraint matrices at run time. For the fake constraint, use a constraint with the same form as the second constraint.

```
E3 = [0 0];
F3 = [0 -1 0 0];
G3 = laneWidth*lanes/2;
```

Specify the mixed input/output constraints in the controller using the `setconstraint` function.

```
setconstraint(mpcobj,[E1;E2;E3],[F1;F2;F3],[G1;G2;G3]);
```

Simulate Controller

In this example, you use an adaptive MPC controller because it handles the nonlinear vehicle dynamics more effectively than a traditional MPC controller. A traditional MPC controller uses a constant plant model. However, adaptive MPC allows you to provide a new plant model at each control interval. Because the new model describes the plant dynamics more accurately at the new operating condition, an adaptive MPC controller performs better than a traditional MPC controller.

Also, to enable the controller to avoid the safe zone surrounding the obstacle, you update the third mixed constraint at each control interval. Basically, the ego car must be above the line formed from the ego car to the upper left corner of the safe zone. For more details, open `obstacleComputeCustomConstraint`.

Use a constant reference signal.

```
refSignal = [0 0 0 V];
```

Initialize plant and controller states.

```
x = x0;
u = u0;
egoStates = mpcstate(mpcobj);
```

The simulation time is 4 seconds.

```
T = 0:Ts:4;
```

Log simulation data for plotting.

```
saveSlope = zeros(length(T),1);
saveIntercept = zeros(length(T),1);
ympc = zeros(length(T),size(Cd,1));
umpc = zeros(length(T),size(Bd,2));
```

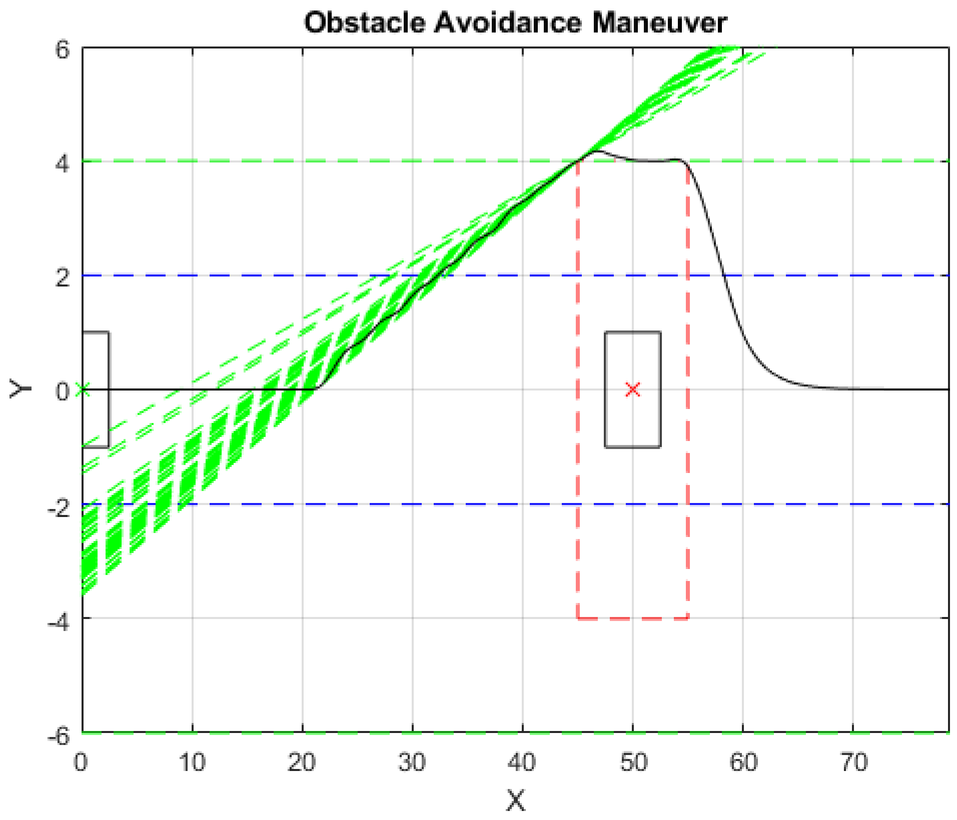
Run the simulation.

```
for k = 1:length(T)
    % Obtain new plant model and output measurements for interval |k|.
    [Ad,Bd,Cd,Dd,U,Y,X,DX] = obstacleVehicleModelDT(Ts,x,u);
    measurements = Cd * x + Dd * u;
    ympc(k,:) = measurements';

    % Determine whether the vehicle sees the obstacle, and update the mixed
    % I/O constraints when obstacle is detected.
    detection = obstacleDetect(x,obstacle,laneWidth);
    [E,F,G,saveSlope(k),saveIntercept(k)] = ...
        obstacleComputeCustomConstraint(x,detection,obstacle,laneWidth,lanes);

    % Prepare new plant model and nominal conditions for adaptive MPC.
    newPlant = ss(Ad,Bd,Cd,Dd,'Ts',Ts);
    newNominal = struct('U',U,'Y',Y,'X',X,'DX',DX);

    % Prepare new mixed I/O constraints.
    options = mpcmoveopt;
    options.CustomConstraint = struct('E',E,'F',F,'G',G);

    % Compute optimal moves using the updated plant, nominal conditions,
    % and constraints.
    [u,Info] = mpcmoveAdaptive(mpcobj,egoStates,newPlant,newNominal,...
        measurements,refSignal,[],options);
    umpc(k,:) = u';

    % Update the plant state for the next iteration |k+1|.
    x = Ad * x + Bd * u;
end

mpcverbosity(status);
```

Analyze Results

Plot the trajectory of the ego car (black line) and the third mixed I/O constraints (dashed green lines) during the obstacle avoidance maneuver.

```
figure(f)
for k = 1:length(saveSlope)
    X = [0;50;100];
    Y = saveSlope(k)*X + saveIntercept(k);
    line(X,Y,'LineStyle','--','Color','g' )
end
plot(ympc(:,1),ympc(:,2),'-k');
axis([0 ympc(end,1) -laneWidth*lanes/2 laneWidth*lanes/2]) % reset axis
```



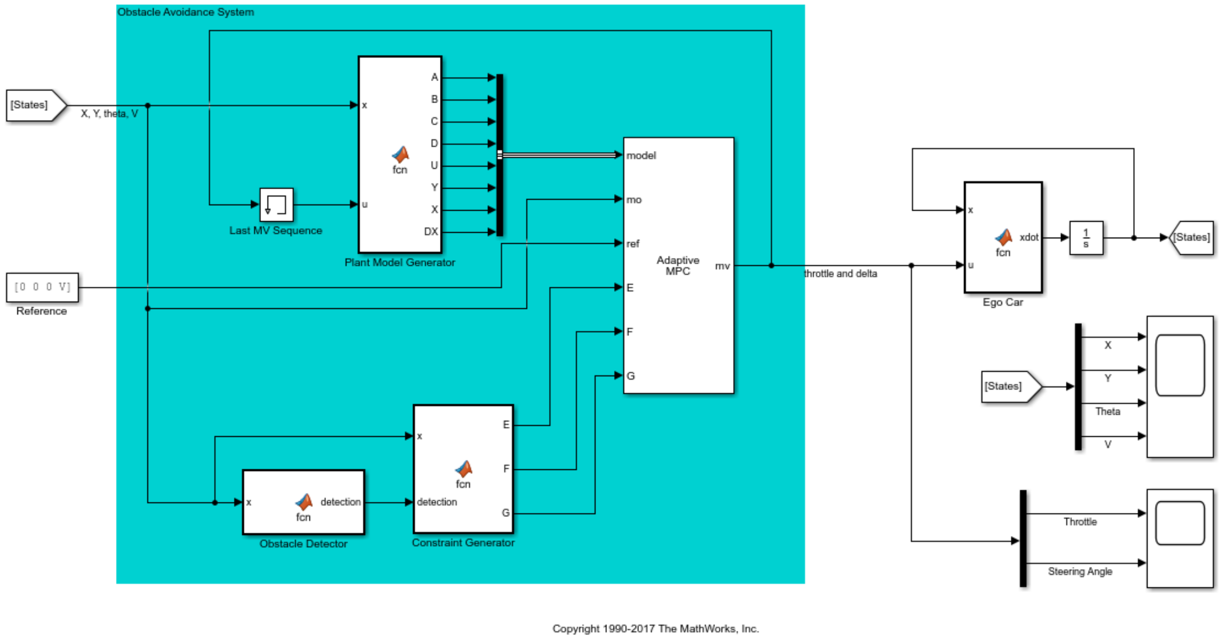
The MPC controller successfully completes the task without human intervention.

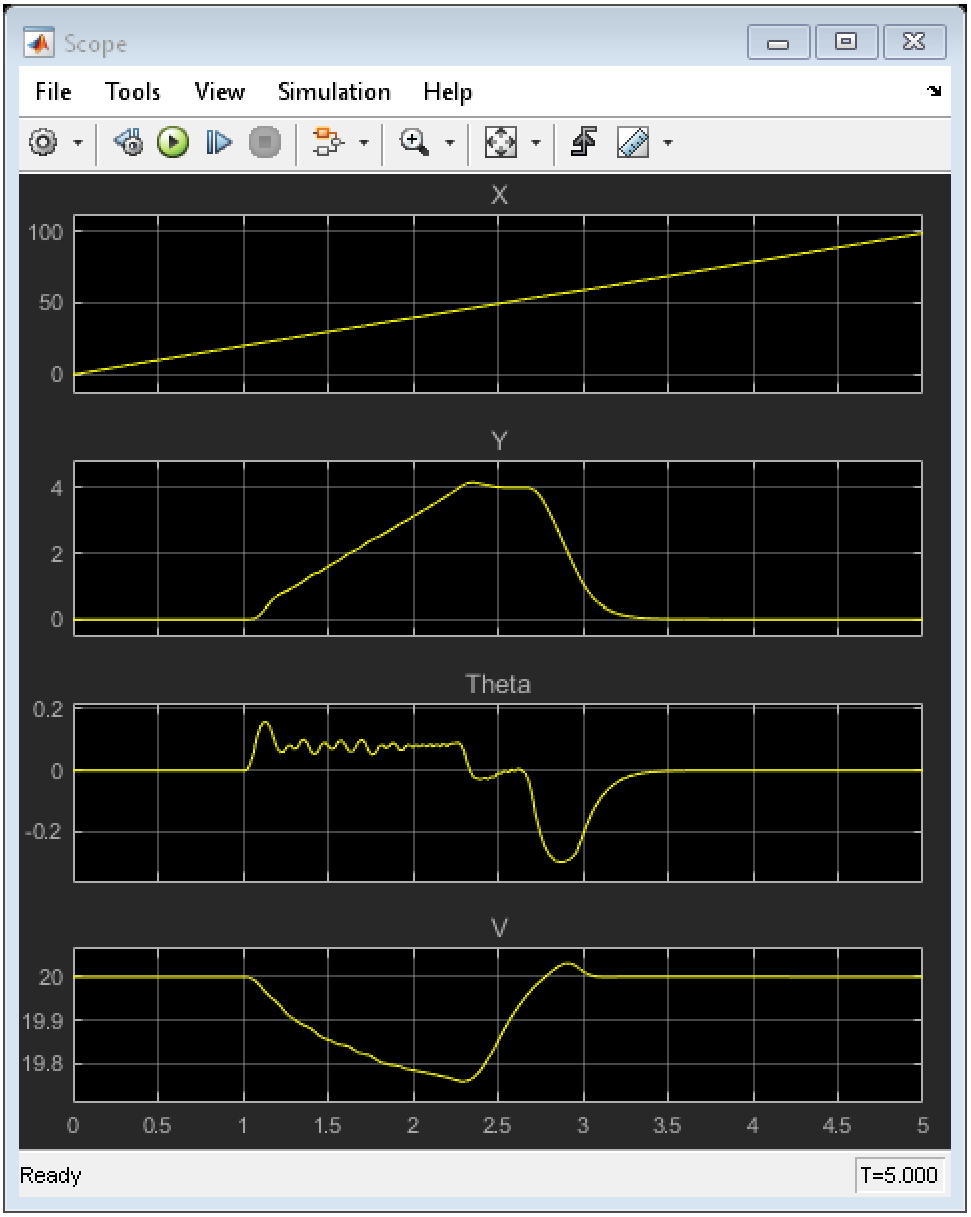
Simulate Controller in Simulink

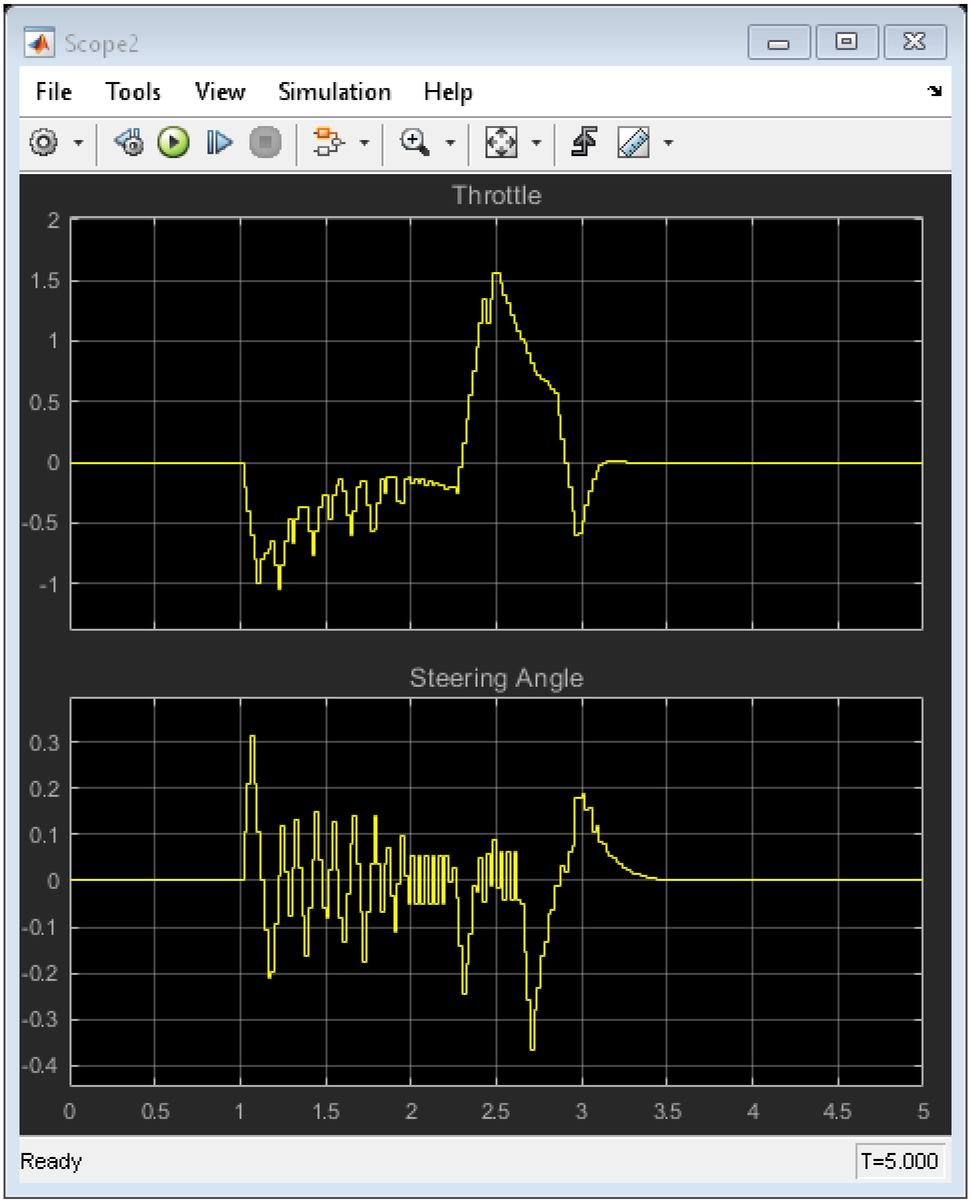
Open the Simulink model. The obstacle avoidance system contains multiple components:

- Plant Model Generator - Produce new plant model and nominal values
- Obstacle Detector - Detect obstacle (lidar sensor not included)
- Constraint Generator - Produce new mixed I/O constraints
- Adaptive MPC - Control obstacle avoidance maneuver

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




Scope
File
Tools
View
Simulation
Help
X
100
50
0
Y
4
2
0
Theta
0.2
0
-0.2
V
20
19.9
19.8
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
T=5.000


Scope2
File
Tools
View
Simulation
Help
Throttle
2
1.5
1
0.5
0
-0.5
-1
Steering Angle
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Ready
T=5.000

The simulation result is identical to the command-line result. To support a rapid prototyping workflow, you can generate C/C++ code for the blocks in the obstacle avoidance system.

```
bdclose(mdl)
```

See Also

Blocks
Adaptive MPC Controller

Functions
`mpcmoveAdaptive` | `mpcmoveopt`

More About

- "Adaptive MPC" on page 5-2
- "Update Constraints at Run Time"
- "Automated Driving Using Model Predictive Control" on page 11-2

Time-Varying MPC

When to Use Time-Varying MPC

To adapt to changing operating conditions, adaptive MPC supports updating the prediction model and its associated nominal conditions at each control interval. However, the updated model and conditions remain constant over the prediction horizon. If you can predict how the plant and nominal conditions vary in the future, you can use time-varying MPC to specify a model that changes over the prediction horizon. Such a linear time-varying (LTV) model is useful when controlling periodic systems or nonlinear systems that are linearized around a time-varying nominal trajectory.

To use time-varying MPC, specify arrays for the `Plant` and `Nominal` input arguments of `mpcmoveAdaptive`. For an example of time-varying MPC, see “Time-Varying MPC Control of a Time-Varying Plant” on page 5-51.

Time-Varying Prediction Models

Consider the LTV prediction model

$$\begin{aligned} x(k+1) &= A(k)x(k) + B_u(k)u(k) + B_v(k)v(k) \\ y(k) &= C(k)x(k) + D_v(k)v(k) \end{aligned}$$

where A, B_u, B_v, C, and D are discrete-time state-space matrices that can vary with time. The other model parameters are:

- k — Current control interval time index
- x — Plant model states
- u — Manipulated variables
- v — Measured disturbance inputs
- y — Measured and unmeasured plant outputs

Since time-varying MPC extends adaptive MPC, the plant model requirements are the same; that is, for each model in the `Plant` array:

- Sample time (`Ts`) is constant and identical to the MPC controller sample time.
- Any time delays are absorbed as discrete states.

- The input and output signal configuration remains constant.
- There is no direct feed-through from the manipulated variables to the plant outputs.

For more information, see “Plant Model” on page 5-2.

The prediction of future trajectories for *p* steps into the future, where *p* is the prediction horizon, is the same as for the adaptive MPC case:

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

However, for an LTV prediction model, the matrices S_x, S_{u1}, S_u, and H_v are:

$$S_x = \begin{bmatrix} C(1)A(0) \\ C(2)A(1)A(0) \\ \vdots \\ C(p)\prod_{i=0}^{p-1} A(i) \end{bmatrix}$$

$$S_{u1} = \begin{bmatrix} C(1)B_u(0) \\ C(2)\left[B_u(1) + A(1)B_u(0)\right] \\ \vdots \\ C(p)\sum_{k=0}^{p-1}\left[\left(\prod_{i=k+1}^{p-1} A(i)\right)B_u(k)\right] \end{bmatrix}$$

$$S_u = \left[\begin{array}{c|cccc} & 0 & 0 & \cdots & 0 \\ S_{u1} & C(2)B_u(1) & 0 & \cdots & 0 \\ & \vdots & & & \\ & C(p)\sum_{k=1}^{p-1}\left[\left(\prod_{i=k+1}^{p-1} A(i)\right)B_u(k)\right] & \cdots & \cdots & C(p)B_u(p-1) \end{array}\right]$$

$$H_v = \begin{bmatrix} C(1)B_v(0) & D_v(1) & 0 & \cdots & 0 \\ C(2)A(1)B_v(0) & C(2)B_v(1) & D_v(2) & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ C(p)\left(\prod_{i=1}^{p-1} A(i)\right)B_v(0) & \cdots & \cdots & C(p)B_v(p-1) & D_v(p) \end{bmatrix}$$

where $\prod_{i=k_1}^{k_2} A(i) \triangleq A(k_2)A(k_2-1)\ldots A(k_1)$ if $k_2 \geq k_1$, or I otherwise.

For more information on the prediction matrices for implicit MPC and adaptive MPC, see "QP Matrices" on page 2-7.

Time-Varying Nominal Conditions

Linear models are often obtained by linearizing nonlinear dynamics around time-varying nominal trajectories. For example, consider the following LTI model, obtained by linearizing a nonlinear system at the time-varying nominal offsets x_{off}, u_{off}, v_{off}, and y_{off}:

$$\begin{aligned} x(k+1) - x_{off}(k+1) &= A(k)\big(x(k) - x_{off}(k)\big) + B_u(k)\big(u(k) - u_{off}(k)\big) \\ &\quad + B_v(k)\big(v(k) - v_{off}(k)\big) + \Delta x_{off}(k) \\ y(k) - y_{off}(k) &= C(k)\big(x(k) - x_{off}(k)\big) + D_v(k)\big(v(k) - v_{off}(k)\big) \end{aligned}$$

If we define

$$\begin{aligned} &\overline{x_{off}} \triangleq x(0), \quad \overline{u_{off}} \triangleq u(0) \\ &\overline{v_{off}} \triangleq v(0), \quad \overline{y_{off}} \triangleq y(0) \end{aligned}$$

as standard nominal values that remain constant over the prediction horizon, we can transform the LTI model into the following LTV model:

$$\begin{aligned} x(k+1) - \overline{x_{off}} &= A(k)\big(x(k) - \overline{x_{off}}\big) + B_u(k)\big(u(k) - \overline{u_{off}}\big) + B_v(k)\big(v(k) - \overline{v_{off}}\big) + \overline{B}_v(k) \\ y(k) - \overline{y_{off}} &= C(k)\big(x(k) - \overline{x_{off}}\big) + D_v(k)\big(v(k) - \overline{v_{off}}\big) + \overline{D}_v(k) \end{aligned}$$

where

$$\begin{aligned} \overline{B}_v(k) &\triangleq \Delta x_{off}(k) + x_{off}(k) - \overline{x_{off}} + A(k)\big(\overline{x_{off}} - x_{off}(k)\big) + B_u(k)\big(\overline{u_{off}} - u_{off}(k)\big) \\ &\quad + B_v(k)\big(\overline{v_{off}} - v_{off}(k)\big) \\ \overline{D}_v(k) &\triangleq y_{off}(k) - \overline{y_{off}} + C(k)\big(\overline{x_{off}} - x_{off}(k)\big) + D_v(k)\big(\overline{v_{off}} - v_{off}(k)\big) \end{aligned}$$

If the original linearized model is already LTV, the same transformation applies.

State Estimation

As with adaptive MPC, time-varying MPC uses a time-varying Kalman filter based on $A(0)$, $B(0)$, $C(0)$, and $D(0)$ from the initial prediction step; that is, the current time at which the state is estimated. For more information, see “State Estimation” on page 5-4.

See Also

`mpcmoveAdaptive`

More About

- “Adaptive MPC” on page 5-2
- “Optimization Problem” on page 2-2
- “Time-Varying MPC Control of a Time-Varying Plant” on page 5-51

Time-Varying MPC Control of a Time-Varying Plant

This example shows how the Model Predictive Control Toolbox™ can use time-varying prediction models to achieve better performance when controlling a time-varying plant.

The following MPC controllers are compared:

1. Linear MPC controller based on a time-invariant average model
2. Linear MPC controller based on a time-invariant model, which is updated at each time step.
3. Linear MPC controller based on a time-varying prediction model.

Time-Varying Linear Plant

In this example, the plant is a single-input-single-output 3rd order time-varying linear system with poles, zeros and gain that vary periodically with time.

$$G = \frac{5s + 5 + 2\cos(2.5t)}{s^3 + 3s^2 + 2s + 6 + \sin(5t)}$$

The plant poles move between being stable and unstable at run time, which leads to a challenging control problem.

Generate an array of plant models at `t = 0`, `0.1`, `0.2`, ..., 10 seconds.

```
Models = tf;
ct = 1;
for t = 0:0.1:10
    Models(:,:,ct) = tf([5 5+2*cos(2.5*t)],[1 3 2 6+sin(5*t)]);
    ct = ct + 1;
end
```

Convert the models to state-space format and discretize them with a sample time of 0.1 second.

```
Ts = 0.1;
Models = ss(c2d(Models,Ts));
```

MPC Controller Design

The control objective is to track a step change in the reference signal. First, design an MPC controller for the average plant model. The controller sample time is 0.1 second.

```
sys = ss(c2d(tf([5 5],[1 3 2 6]),Ts));  % prediction model
p = 3;                                   % prediction horizon
m = 3;                                   % control horizon
mpcobj = mpc(sys,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Set hard constraints on the manipulated variable and specify tuning weights.

```
mpcobj.MV = struct('Min',-2,'Max',2);
mpcobj.Weights = struct('MV',0,'MVRate',0.01,'Output',1);
```

Set the initial plant states to zero.

```
x0 = zeros(size(sys.B));
```

Closed-Loop Simulation with Implicit MPC

Run a closed-loop simulation to examine whether the designed implicit MPC controller can achieve the control objective without updating the plant model used in prediction.

Set the simulation duration to 5 seconds.

```
Tstop = 5;
```

Use the `mpcmove` command in a loop to simulate the closed-loop response.

```
yyMPC = [];
uuMPC = [];
x = x0;
xmpc = mpcstate(mpcobj);
fprintf('Simulating MPC controller based on average LTI model.\n');
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyMPC = [yyMPC,y];
    % Compute and store the MPC optimal move.
    u = mpcmove(mpcobj,xmpc,y,1);
    uuMPC = [uuMPC,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
Simulating MPC controller based on average LTI model.
```

Closed-Loop Simulation with Adaptive MPC

Run a second simulation to examine whether an adaptive MPC controller can achieve the control objective.

Use the `mpcmoveAdaptive` command in a loop to simulate the closed-loop response. Update the plant model for each control interval, and use the updated model to compute the optimal control moves. The `mpcmoveAdaptive` command uses the same prediction model across the prediction horizon.

```
yyAMPC = [];
uuAMPC = [];
x = x0;
xmpc = mpcstate(mpcobj);
nominal = mpcobj.Model.Nominal;
fprintf('Simulating MPC controller based on LTI model, updated at each time step t.\n')
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyAMPC = [yyAMPC, y];
    % Compute and store the MPC optimal move.
    u = mpcmoveAdaptive(mpcobj,xmpc,real_plant,nominal,y,1);
    uuAMPC = [uuAMPC,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end
```

```
Simulating MPC controller based on LTI model, updated at each time step t.
```

Closed-Loop Simulation with Time-Varying MPC

Run a third simulation to examine whether a time-varying MPC controller can achieve the control objective.

The controller updates the prediction model at each control interval and also uses time-varying models across the prediction horizon, which gives MPC controller the best knowledge of plant behavior in the future.

Use the `mpcmoveAdaptive` command in a loop to simulate the closed-loop response. Specify an array of plant models rather than a single model. The controller uses each model in the array at a different prediction horizon step.

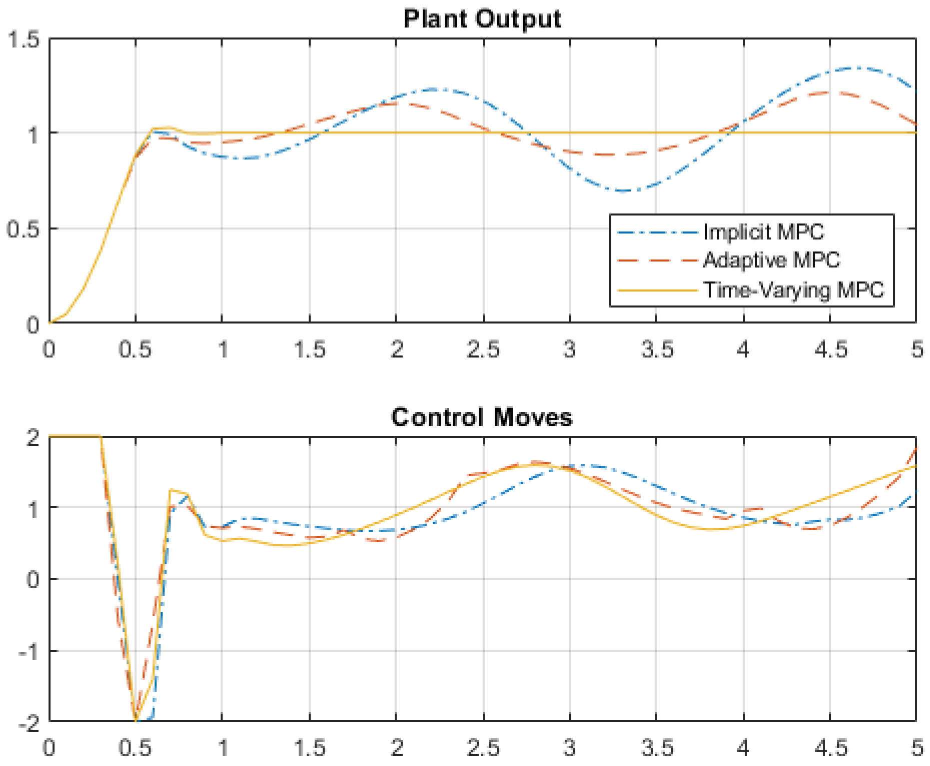
```
yyLTVMPC = [];
uuLTVMPC = [];
x = x0;
xmpc = mpcstate(mpcobj);
Nominals = repmat(nominal,3,1); % Nominal conditions are constant over the prediction h
fprintf('Simulating MPC controller based on time-varying model, updated at each time st
for ct = 1:(Tstop/Ts+1)
    % Get the real plant.
    real_plant = Models(:,:,ct);
    % Update and store the plant output.
    y = real_plant.C*x;
    yyLTVMPC = [yyLTVMPC, y];
    % Compute and store the MPC optimal move.
    u = mpcmoveAdaptive(mpcobj,xmpc,Models(:,:,ct:ct+p),Nominals,y,1);
    uuLTVMPC = [uuLTVMPC,u];
    % Update the plant state.
    x = real_plant.A*x + real_plant.B*u;
end
```

```
Simulating MPC controller based on time-varying model, updated at each time step t.
```

Performance Comparison of MPC Controllers

Compare the closed-loop responses.

```
t = 0:Ts:Tstop;
figure
subplot(2,1,1);
plot(t,yyMPC,'-.',t,yyAMPC,'--',t,yyLTVMPC);
grid
legend('Implicit MPC','Adaptive MPC','Time-Varying MPC','Location','SouthEast')
title('Plant Output');
subplot(2,1,2)
plot(t,uuMPC,'-.',t,uuAMPC,'--',t,uuLTVMPC)
grid
title('Control Moves');
```

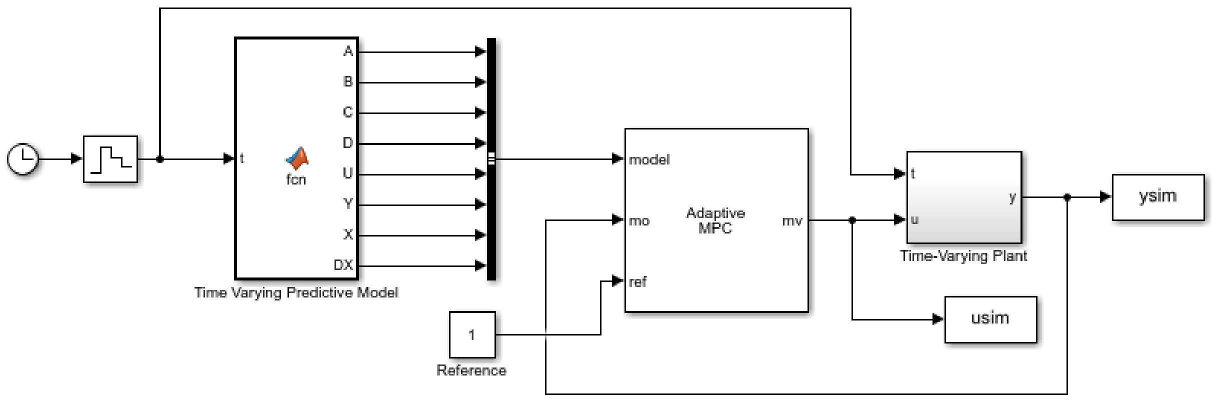


Only the time-varying MPC controller is able to bring the plant output close enough to the desired setpoint.

Closed-Loop Simulation of Time-Varying MPC in Simulink

To simulate time-varying MPC control in Simulink, pass the time-varying plant models to `model` inport of the Adaptive MPC Controller block.

```
xmpc = mpcstate(mpcobj);
mdl = 'mpc_timevarying';
open_system(mdl);
```



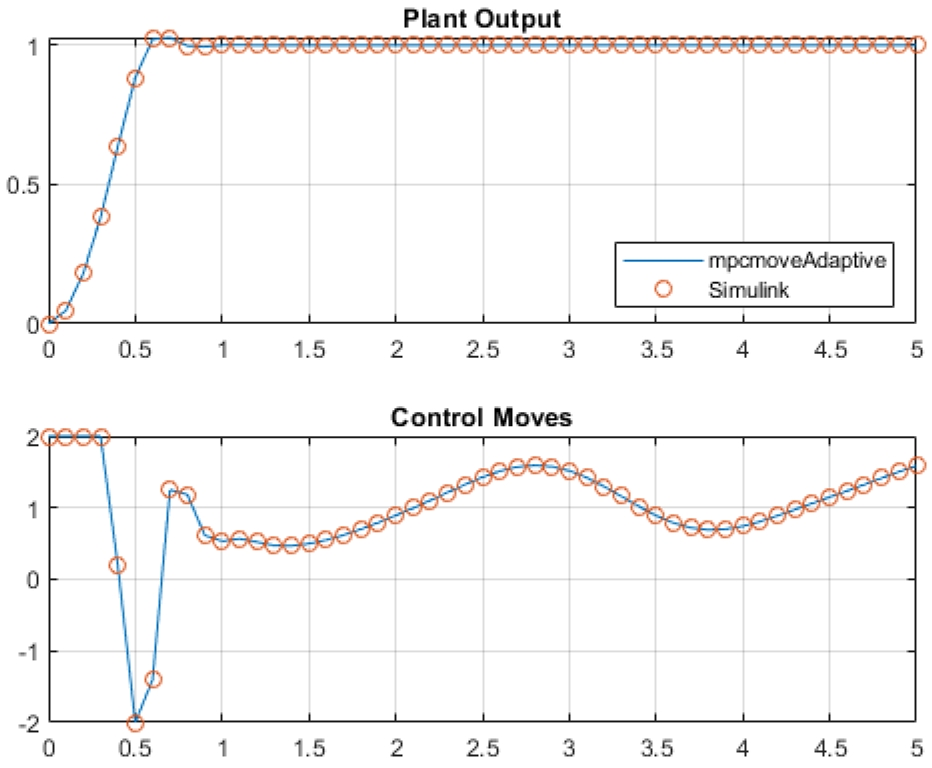
Run the simulation.

```
sim(mdl,Tstop);
fprintf('Simulating MPC controller based on LTV model in Simulink.\n');
```

```
Simulating MPC controller based on LTV model in Simulink.
```

Plot the MATLAB and Simulink time-varying simulation results.

```
figure
subplot(2,1,1)
plot(t,yyLTVMPC,t,ysim,'o');
grid
legend('mpcmoveAdaptive','Simulink','Location','SouthEast')
title('Plant Output');
subplot(2,1,2)
plot(t,uuLTVMPC,t,usim,'o')
grid
title('Control Moves');
```



The closed-loop responses in MATLAB and Simulink are identical.

```
bdclose(mdl);
```

See Also

`mpcmoveAdaptive`

More About

- “Time-Varying MPC” on page 5-47

Explicit MPC Design

Explicit MPC

A traditional model predictive controller solves a quadratic program (QP) at each control interval to determine the optimal manipulated variable (MV) adjustments. These adjustments are the solution of the implicit nonlinear function $u=f(x)$.

The vector x contains the current controller state and other independent variables affecting the QP solution, such as the current output reference values. The Model Predictive Control Toolbox software imposes restrictions that force a unique QP solution.

Finding the optimal MV adjustments can be time consuming, and the required time can vary significantly from one control interval to the next. In applications that require a solution within a certain consistent time, which could be on the order of microseconds, the implicit MPC approach might be unsuitable.

As shown in “Optimization Problem” on page 2-2, if no QP inequality constraints are active for a given x vector, then the optimal MV adjustments become a linear function of x:

$$u = Fx + G.$$

where, F and G are constants. Similarly, if x remains in a region where a fixed subset of inequality constraints is active, the QP solution is also a linear function of x, but with different F and G constants.

Explicit MPC uses offline computations to determine all polyhedral regions where the optimal MV adjustments are a linear function of x, and the corresponding control-law constants. When the controller operates in real time, the explicit MPC controller performs the following steps at each control instant, k:

1. Estimate the controller state using available measurements, as in traditional MPC.
2. Form $x(k)$ using the estimated state and the current values of the other independent variables.
3. Identify the region in which $x(k)$ resides.
4. Looks up the predetermined F and G constants for this region.
5. Evaluate the linear function $u(k) = Fx(k) + G$.

You can establish a tight upper bound for the time required in each step. If the number of regions is not too large, the total computational time can be small. However, as the number of regions increases, the time required in step 3 dominates. Also, the memory required to store all the linear control laws and polyhedral regions becomes excessive. The number of regions characterizing $u = f(x)$ depends primarily on the QP inequality constraints that could be active at the solution. If an explicit MPC controller has many constraints, and thus requires significant computational effort or memory, a traditional (implicit) implementation may be preferable.

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 6-8
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-20
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

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 prior to determining an explicit MPC. Fixed parameters include prediction models (plant, disturbance and measurement noise), scale factors, horizons, penalty weights, manipulated variable targets, and constraint bounds.

  For information about designing a traditional MPC controller, see “Controller Creation”.

  For information about tuning traditional MPC controllers, see “Refinement”.
- Reference and measured disturbance previewing are not supported. At each control interval, the current n_y reference and n_v measured disturbance signals apply for the entire prediction horizon.
- To limit the number of regions needed by explicit MPC, include only essential constraints.

  - When including a constraint on a manipulated variable (MV) use a short control horizon or MV blocking. See “Choose Sample Time and Horizons” on page 1-6.
  - Avoid constraints on plant outputs. If such a constraint is essential, consider imposing it for selected prediction horizon steps rather than the entire prediction horizon.

- Establish upper and lower bounds for each of the $n_x = n_{xc} + n_y + n_v$ independent variables. You might know some of these bounds a priori. However, you must run simulations that record at least the n_{xc} controller states as the system operates over the range of expected conditions. It is very important that you not understimate this range, because the explicit MPC control function is not defined for independent variables outside the range.

  For information about specifying bounds, see `generateExplicitRange`.

  For information about simulating a traditional MPC controller, see “Simulation”.

Explicit MPC Generation

Given the constant MPC design parameters and the n_x upper and lower bounds on the control law’s independent variables, i.e.,

$$x_l \le x(k) \le x_u,$$

the `generateExplicitMPC` command determines n_r regions. Each of these regions is defined by an inequality constraint and the corresponding control law constants:

$$\begin{aligned} H_i x(k) &\le K_i, \quad i = 1, n_r \\ u(k) &= F_i x(k) + G_i, \quad i = 1, n_r. \end{aligned}$$

The Explicit MPC Controller object contains the constants H_i, K_i, F_i, and G_i for each region. The Explicit MPC Controller object also holds the original (implicit) design and independent variable bounds. Provided that $x(k)$ stays within the specified bounds and you retain all n_r regions, the explicit MPC object should provide the same optimal MV adjustments, $u(k)$, as the equivalent implicit MPC object.

For details about explicit MPC, see [1]. For details about how the explicit MPC controller is generated, see [2].

Explicit MPC Simplification

Even a relatively simple explicit MPC controller might need $n_r >> 100$ to characterize the QP solution completely. If the number of regions is large, consider the following:

- Visualize the solution using the `plotSection` command.

- Use the `simplify` command to reduce the number of regions. In some cases, this can be done with no (or negligible) impact on control law optimality. For example, pairs of adjacent regions might employ essentially the same F_i and K_i constants. If so, and if the union of the two regions forms a convex set, they can be merged into a single region.

  Alternatively, you can eliminate relatively small regions or retain selected regions only. If during operation the current $x(k)$ is not contained in any of the retained regions, the explicit MPC will return a suboptimal $u(k)$, as follows:

  $$u(k) = F_j x(k) + G_j.$$

  Here, j is the index of the region whose bounding constraint, $H_j x(k) \le K_j$, 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, $x_l \le x(k) \le x_u$. 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 that needed to test each region, computing the violation term in each case, and then calculating the suboptimal control adjustment.

Simulation

You can perform command-line simulations using the `sim` or `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

Explicit MPC Controller | `generateExplicitMPC` | `mpcmoveExplicit`

More About

- “Explicit MPC” on page 6-2
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-8
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-20
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

Explicit MPC Control of a Single-Input-Single-Output Plant

This example shows how to control a double integrator plant under input saturation in Simulink® using explicit MPC.

See also MPCDOUBLEINT.

Define Plant Model

The linear open-loop dynamic model is a double integrator:

```
plant = tf(1,[1 0 0]);
```

Design MPC Controller

Create the controller object with sampling period, prediction and control horizons:

```
Ts = 0.1;
p = 10;
m = 3;
mpcobj = mpc(plant, Ts, p, m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify actuator saturation limits as MV constraints.

```
mpcobj.MV = struct('Min',-1,'Max',1);
```

Generate Explicit MPC Controller

Explicit MPC executes the equivalent explicit piecewise affine version of the MPC control law defined by the traditional MPC. To generate an Explicit MPC from a traditional MPC, you must specify range for each controller state, reference signal, manipulated variable and measured disturbance so that the multi-parametric quadratic programming problem is solved in the parameter space defined by these ranges.

Obtain a range structure for initialization

Use `generateExplicitRange` command to obtain a range structure where you can specify range for each parameter afterwards.

```
range = generateExplicitRange(mpcobj);
```

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

Specify ranges for controller states

MPC controller states include states from plant model, disturbance model and noise model in that order. Setting the range of a state variable is sometimes difficult when the state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
```

Specify ranges for reference signals

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate Explicit MPC must be at least as large as the practical range.

```
range.Reference.Min = -2;
range.Reference.Max = 2;
```

Specify ranges for manipulated variables

If manipulated variables are constrained, the ranges used to generate Explicit MPC must be at least as large as these limits.

```
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Construct the Explicit MPC controller

Use `generateExplicitMPC` command to obtain the Explicit MPC controller with the parameter ranges previously specified.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range);
display(mpcobjExplicit);
```

```
Regions found / unexplored:       19/       0


Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        19
Number of parameters:      4
Is solution simplified:    No
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'mpcobjExplicit.MPC' for the original implicit MPC design.
Type 'mpcobjExplicit.Range' for the valid range of parameters.
Type 'mpcobjExplicit.OptimizationOptions' for the options used in multi-parametric QP c
Type 'mpcobjExplicit.PiecewiseAffineSolution' for regions and gain in each solution.
```

Use `simplify` command with the "exact" method to join pairs of regions whose corresponding gains are the same and whose union is a convex set. This practice can reduce memory footprint of the Explicit MPC controller without sacrifice any performance.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, 'exact');
display(mpcobjExplicitSimplified);
```

```
Regions to analyze:       15/      15


Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        15
Number of parameters:      4
Is solution simplified:    Yes
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'mpcobjExplicitSimplified.MPC' for the original implicit MPC design.
Type 'mpcobjExplicitSimplified.Range' for the valid range of parameters.
Type 'mpcobjExplicitSimplified.OptimizationOptions' for the options used in multi-paran
Type 'mpcobjExplicitSimplified.PiecewiseAffineSolution' for regions and gain in each sc
```

The number of piecewise affine region has been reduced.

Plot Piecewise Affine Partition

You can review any 2-D section of the piecewise affine partition defined by the Explicit MPC control law.

Obtain a plot parameter structure for initialization

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicitSimplified);
```

Specify parameters for a 2-D plot

In this example, you plot the 1th state variable vs. the 2nd state variable. All the other parameters must be fixed at a value within its range.

```
params.State.Index = [];
params.State.Value = [];
```

Fix other reference signals

```
params.Reference.Index = 1;
params.Reference.Value = 0;
```

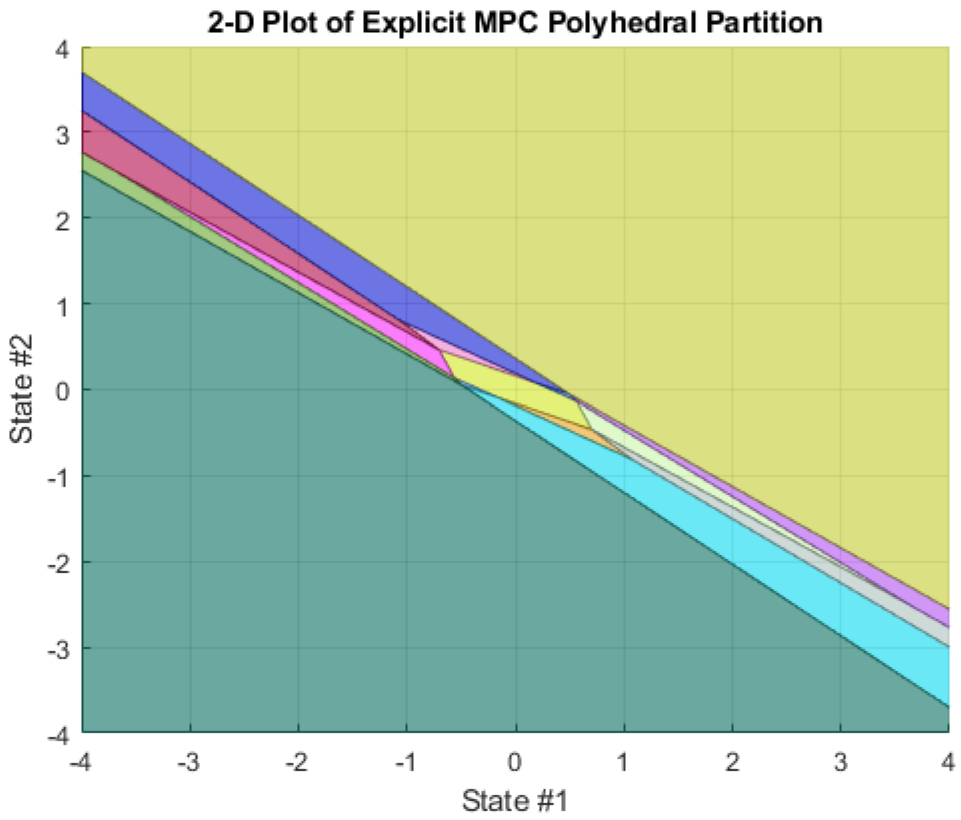
Fix manipulated variables

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Plot the 2-D section

Use `plotSection` command to plot the 2-D section defined previously.

```
plotSection(mpcobjExplicitSimplified, params);
axis([-4 4 -4 4]);
grid
xlabel('State #1');
ylabel('State #2');
```



Simulate Using MPCMOVE Command

Compare closed-loop simulation between tradition MPC (as referred as Implicit MPC) and Explicit MPC using `mpcmove` and `mpcmoveExplicit` commands respectively.

Prepare to store the closed-loop MPC responses.

```
Tf = round(5/Ts);
YY = zeros(Tf,1);
YYExplicit = zeros(Tf,1);
UU = zeros(Tf,1);
UUExplicit = zeros(Tf,1);
```

Prepare the real plant used in simulation

```
sys = c2d(ss(plant),Ts);
xsys = [0;0];
xsysExplicit = xsys;
```

Use MPCSTATE object to specify the initial states for both controllers

```
xmpc = mpcstate(mpcobj);
xmpcExplicit = mpcstate(mpcobjExplicitSimplified);
```

Simulate closed-loop response in each iteration.

```
for t = 0:Tf
    % update plant measurement
    ysys = sys.C*xsys;
    ysysExplicit = sys.C*xsysExplicit;
    % compute traditional MPC action
    u = mpcmove(mpcobj,xmpc,ysys,1);
    % compute Explicit MPC action
    uExplicit = mpcmoveExplicit(mpcobjExplicit,xmpcExplicit,ysysExplicit,1);
    % store signals
    YY(t+1)=ysys;
    YYExplicit(t+1)=ysysExplicit;
    UU(t+1)=u;
    UUExplicit(t+1)=uExplicit;
    % update plant state
    xsys = sys.A*xsys + sys.B*u;
    xsysExplicit = sys.A*xsysExplicit + sys.B*uExplicit;
end
fprintf('\nDifference between traditional and Explicit MPC responses using MPCMOVE comm
```

```
Difference between traditional and Explicit MPC responses using MPCMOVE command is 2.74
```

Simulate Using SIM Command

Compare closed-loop simulation between tradition MPC and Explicit MPC using `sim` commands respectively.

```
Tf = 5/Ts;                        % simulation iterations
[y1,t1,u1] = sim(mpcobj,Tf,1);  % simulation with tradition MPC
[y2,t2,u2] = sim(mpcobjExplicitSimplified,Tf,1);   % simulation with Explicit MPC
```

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

The simulation results are identical.

```
fprintf('\nDifference between traditional and Explicit MPC responses using SIM command
```

```
Difference between traditional and Explicit MPC responses using SIM command is 2.73571e
```

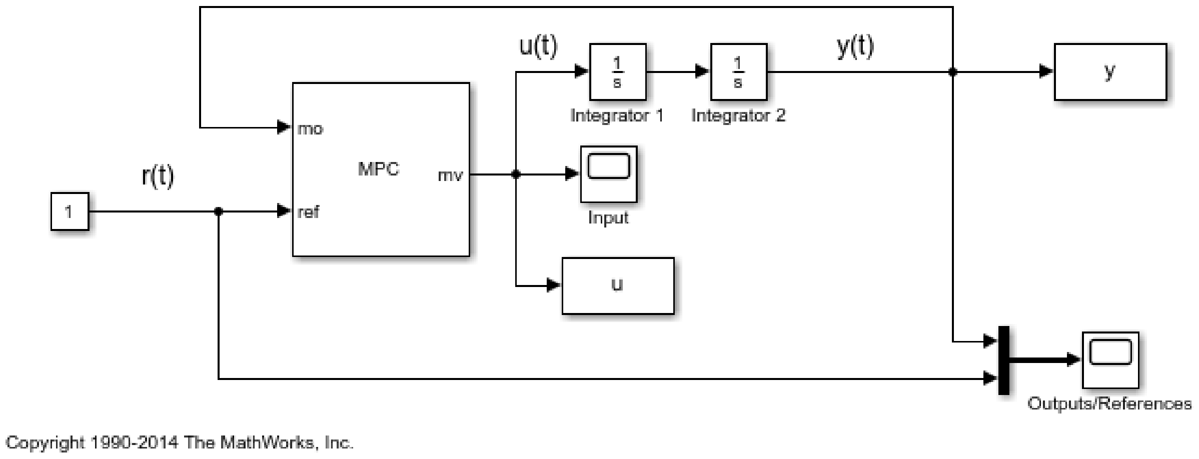
Simulate Using Simulink®

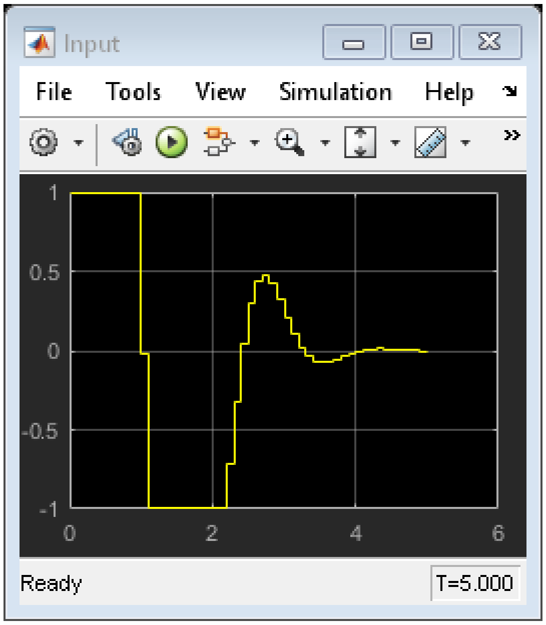
To run this example, Simulink® is required.

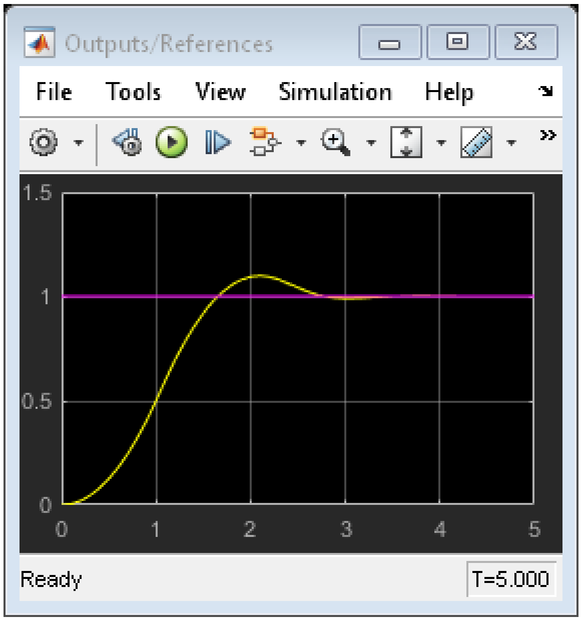
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

Simulate with traditional MPC controller in Simulink. Controller "mpcobj" is specified in the block dialog.

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

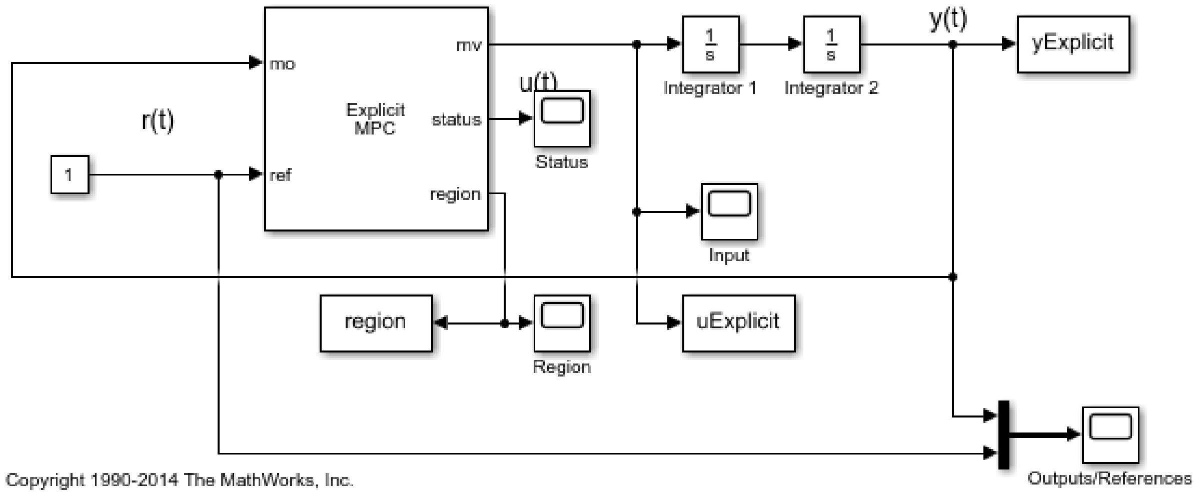


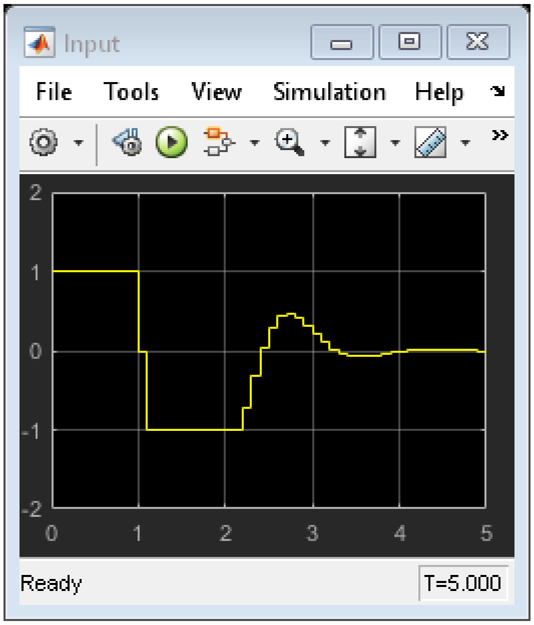

Input
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
0
2
4
6
Ready
T=5.000

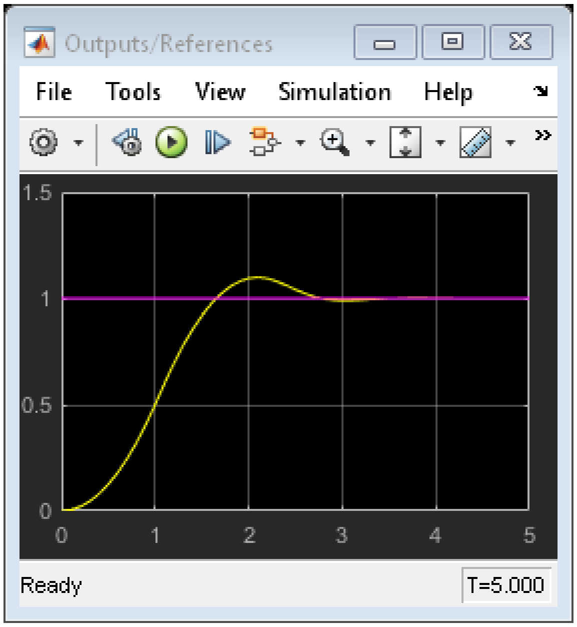

Outputs/References
File
Tools
View
Simulation
Help
1.5
1
0.5
0
0
1
2
3
4
5
Ready
T=5.000

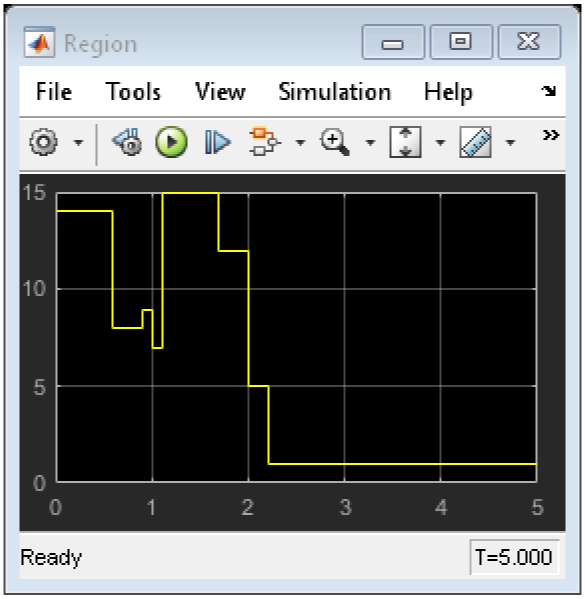
Simulate with Explicit MPC controller in Simulink. Controller "mpcobjExplicitSimplified" is specified in the block dialog.

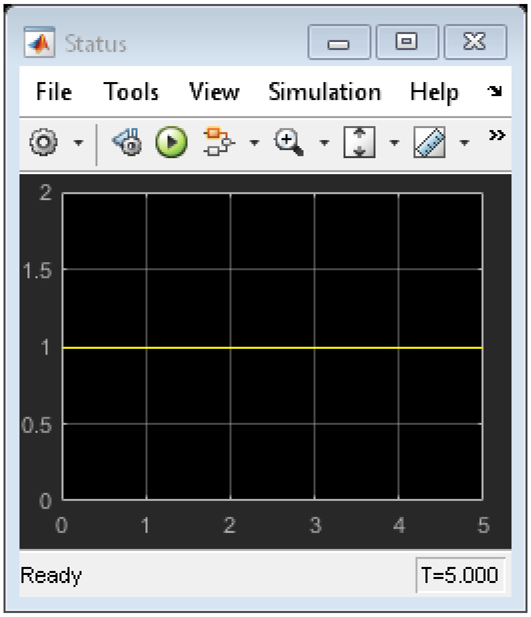
```
mdlExplicit = 'empc_doubleint';
open_system(mdlExplicit);
sim(mdlExplicit);
```




Input
File
Tools
View
Simulation
Help
2
1
0
-1
-2
0
1
2
3
4
5
Ready
T=5.000


Outputs/References
File
Tools
View
Simulation
Help
1.5
1
0.5
0
0
1
2
3
4
5
Ready
T=5.000


Region
File
Tools
View
Simulation
Help
15
10
5
0
0
1
2
3
4
5
Ready
T=5.000


Status
File
Tools
View
Simulation
Help
2
1.5
1
0.5
0
0
1
2
3
4
5
Ready
T=5.000

The closed-loop responses are identical.

```
fprintf('\nDifference between traditional and Explicit MPC responses in Simulink is %g\
```

```
Difference between traditional and Explicit MPC responses in Simulink is 2.71408e-13
```

```
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 6-20
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

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 “Aircraft with Unstable Poles”.

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 undershoots at the first prediction horizon. You also specify scale factors for outputs.

```
OV = struct('Min',{[-0.5;-Inf],[-100;-Inf]},'Max',{[0.5;Inf],[100;Inf]},'ScaleFactor',
```

Weights

The control task is to get zero offset for piecewise-constant references, while avoiding instability due to input saturation. Because both MV and OV variables are already scaled in MPC controller, MPC weights are dimensionless and applied to the scaled MV and OV values. In this example, you penalize the two outputs equally with the same OV weights.

```
Weights = struct('MV',[0 0],'MVRate',[0.1 0.1],'OV',[10 10]);
```

Construct Traditional MPC Controller

Create an MPC controller with the spcified 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 range for each parameter afterwards, use the `generateExplicitRange` command.

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

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 c
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 sacrifice 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-paran
Type 'mpcobjExplicitSimplified.PiecewiseAffineSolution' for regions and gain in each so
```

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.

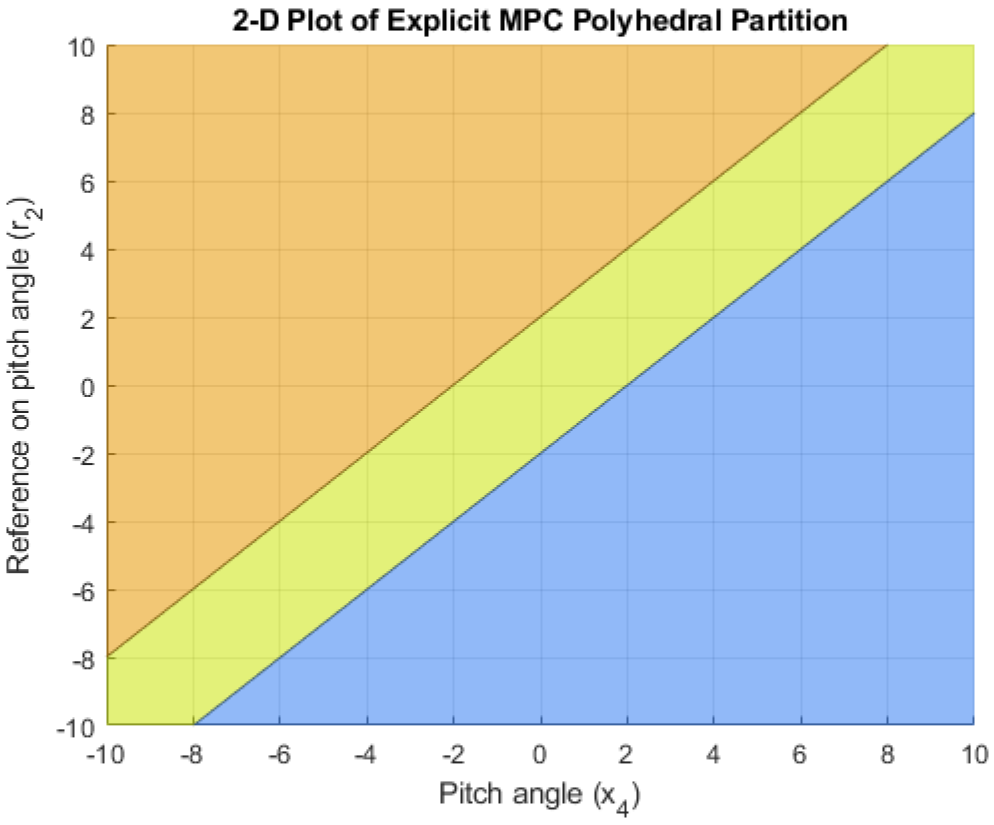
```
params.ManipulatedVariable.Index = [1 2];
params.ManipulatedVariable.Value = [0 0];
```

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)')
```



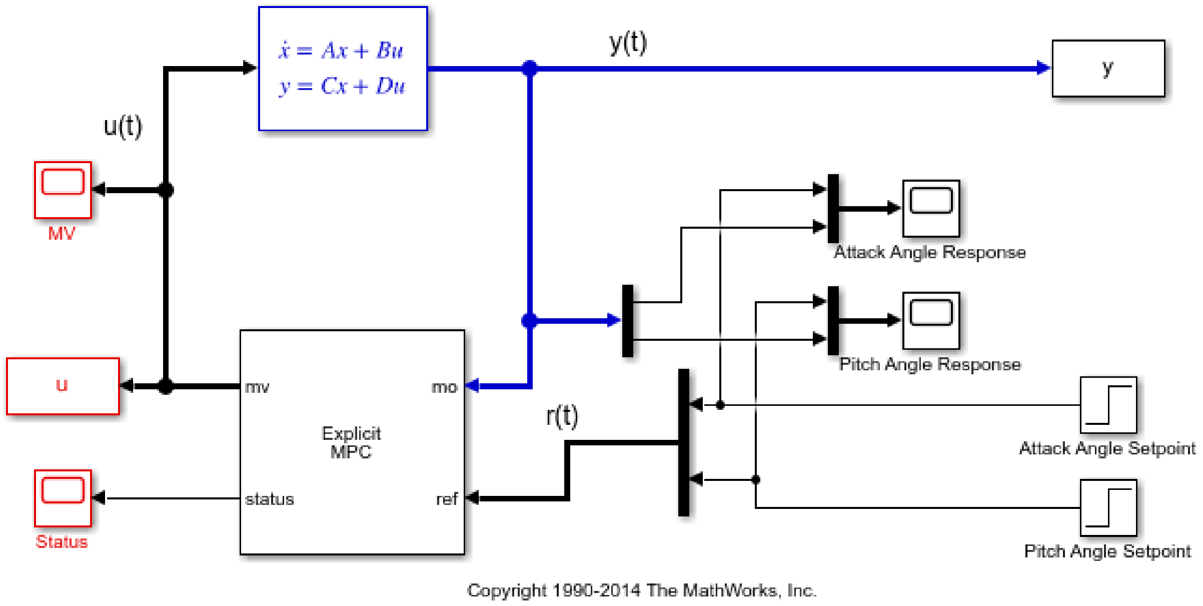
Simulate Using Simulink®

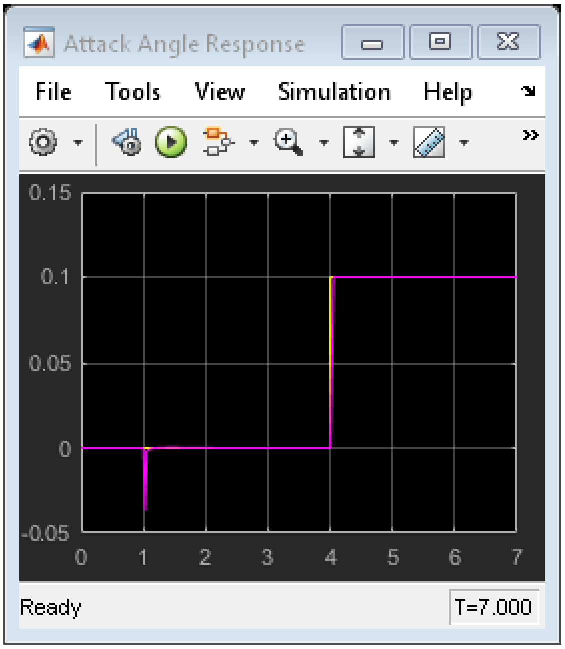
To run this example, Simulink® is required.

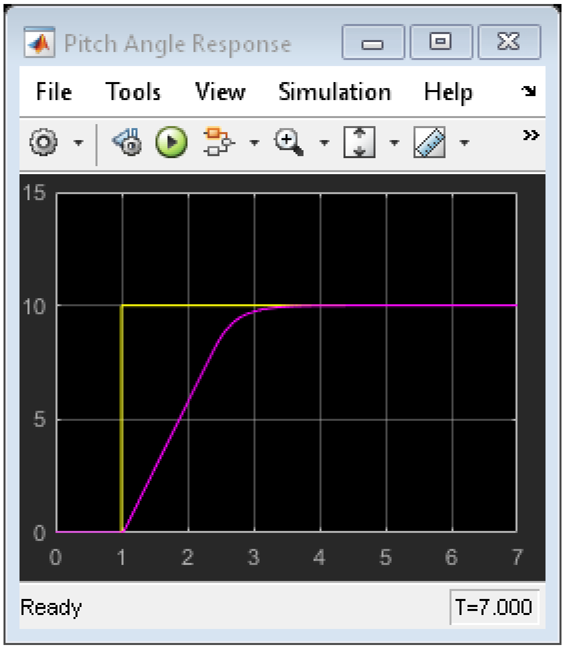
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

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

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




Attack Angle Response
File
Tools
View
Simulation
Help
0.15
0.1
0.05
0
-0.05
0
1
2
3
4
5
6
7
Ready
T=7.000



The closed-loop response is identical to the traditional MPC controller designed in "Aircraft with Unstable Poles".

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 6-8
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output” on page 6-30

Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output

This example shows how to use Explicit MPC to control DC servomechanism under voltage and shaft torque constraints.

Reference

[1] A. Bemporad and E. Mosca, ''Fulfilling hard constraints in uncertain linear systems by reference managing,'' Automatica, vol. 34, no. 4, pp. 451-461, 1998.

See also MPCMOTOR.

Define DC-Servo Motor Model

The linear open-loop dynamic model is defined in "plant". Variable "tau" is the maximum admissible torque to be used as an output constraint.

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

Design MPC Controller

Specify input and output signal types for the MPC controller. The second output, torque, is unmeasurable.

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

MV Constraints

The manipulated variable is constrained between +/- 220 volts. Since the plant inputs and outputs are of different orders of magnitude, you also use scale factors to facilitate MPC tuning. Typical choices of scale factor are the upper/lower limit or the operating range.

```
MV = struct('Min',-220,'Max',220,'ScaleFactor',440);
```

OV Constraints

Torque constraints are only imposed during the first three prediction steps to limit the complexity of the explicit MPC design.

```
OV = struct('Min',{Inf, [-tau;-tau;-tau;-Inf]},'Max',{Inf, [tau;tau;tau;Inf]},'ScaleFac
```

Weights

The control task is to get zero tracking offset for the angular position. Since you only have one manipulated variable, the shaft torque is allowed to float within its constraint by setting its weight to zero.

```
Weights = struct('MV',0,'MVRate',0.1,'OV',[0.1 0]);
```

Construct MPC controller

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

```
Ts = 0.1;           % Sampling time
p = 10;             % Prediction horizon
m = 2;              % Control horizon
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Generate Explicit MPC Controller

Explicit MPC executes the equivalent explicit piecewise affine version of the MPC control law defined by the traditional MPC. To generate an Explicit MPC from a traditional MPC, you must specify the range for each controller state, reference signal, manipulated variable and measured disturbance so that the multi-parametric quadratic programming problem is solved in the parameter sets defined by these ranges.

Obtain a range structure for initialization

Use `generateExplicitRange` command to obtain a range structure where you can specify the range for each parameter afterwards.

```
range = generateExplicitRange(mpcobj);
```

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

Specify ranges for controller states

MPC controller states include states from plant model, disturbance model and noise model in that order. Setting the range of a state variable is sometimes difficult when the state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

```
range.State.Min(:) = -1000;
range.State.Max(:) = 1000;
```

Specify ranges for reference signals

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate Explicit MPC must be at least as large as the practical range. Note that the range for torque reference is fixed at 0 because it has zero weight.

```
range.Reference.Min = [-5;0];
range.Reference.Max = [5;0];
```

Specify ranges for manipulated variables

If manipulated variables are constrained, the ranges used to generate Explicit MPC must be at least as large as these limits.

```
range.ManipulatedVariable.Min = MV.Min - 1;
range.ManipulatedVariable.Max = MV.Max + 1;
```

Construct the Explicit MPC controller

Use `generateExplicitMPC` command to obtain the Explicit MPC controller with the parameter ranges previously specified.

```
mpcobjExplicit = generateExplicitMPC(mpcobj, range);
display(mpcobjExplicit);
```

```
Regions found / unexplored:       75/       0


Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        75
Number of parameters:      6
Is solution simplified:    No
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'mpcobjExplicit.MPC' for the original implicit MPC design.
Type 'mpcobjExplicit.Range' for the valid range of parameters.
Type 'mpcobjExplicit.OptimizationOptions' for the options used in multi-parametric QP c
Type 'mpcobjExplicit.PiecewiseAffineSolution' for regions and gain in each solution.
```

Plot Piecewise Affine Partition

You can review any 2-D section of the piecewise affine partition defined by the Explicit MPC control law.

Obtain a plot parameter structure for initialization

Use `generatePlotParameters` command to obtain a parameter structure where you can specify which 2-D section to plot afterwards.

```
params = generatePlotParameters(mpcobjExplicit);
```

Specify parameters for a 2-D plot

In this example, you plot the 1th state variable vs. the 2nd state variable. All the other parameters must be fixed at a value within its range.

Fix other state variables

```
params.State.Index = [3 4];
params.State.Value = [0 0];
```

Fix reference signals

```
params.Reference.Index = [1 2];
params.Reference.Value = [pi 0];
```

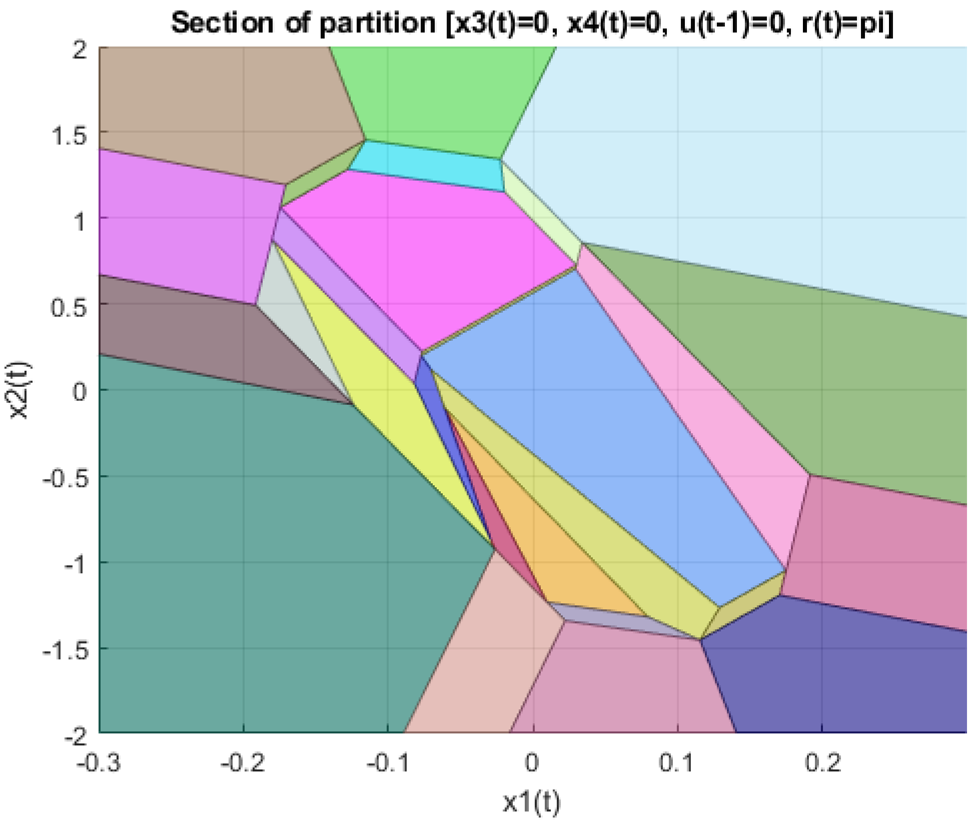
Fix manipulated variables

```
params.ManipulatedVariable.Index = 1;
params.ManipulatedVariable.Value = 0;
```

Plot the 2-D section

Use `plotSection` command to plot the 2-D section defined previously.

```
plotSection(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 Using SIM Command

Compare closed-loop simulation between traditional MPC (as referred as Implicit MPC) and Explicit MPC

```
Tstop = 8;                        % seconds
Tf = round(Tstop/Ts);             % simulation iterations
r = [pi 0];                       % reference signal
[y1,t1,u1] = sim(mpcobj,Tf,r);  % simulation with traditional MPC
[y2,t2,u2] = sim(mpcobjExplicit,Tf,r);   % simulation with Explicit MPC
```

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

The simulation results are identical.

```
fprintf('SIM command: Difference between QP-based and Explicit MPC trajectories = %g\n
```

```
SIM command: Difference between QP-based and Explicit MPC trajectories = 8.68953e-12
```

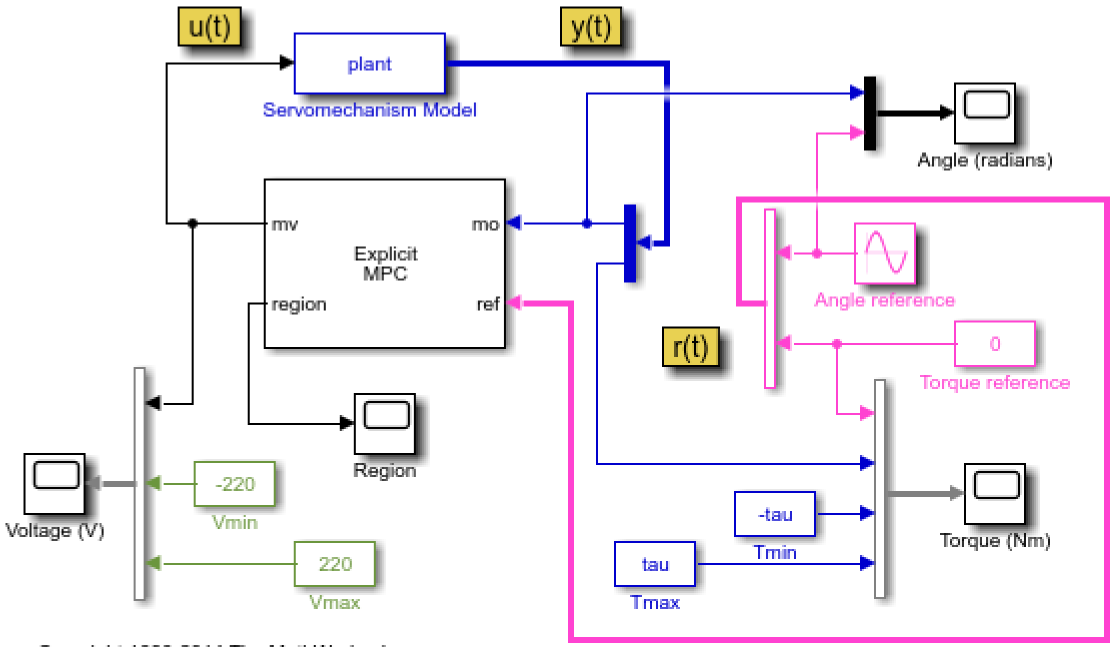
Simulate Using Simulink®

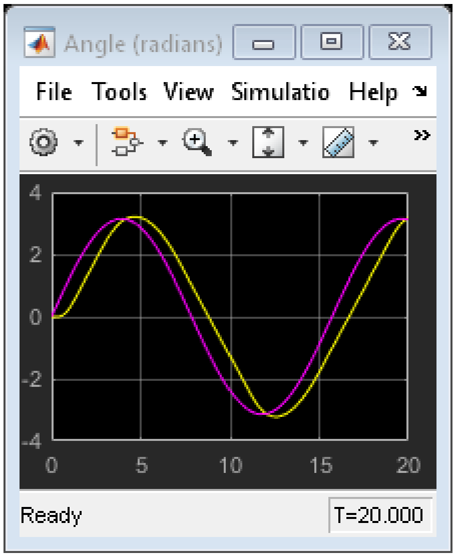
To run this example, Simulink® is required.

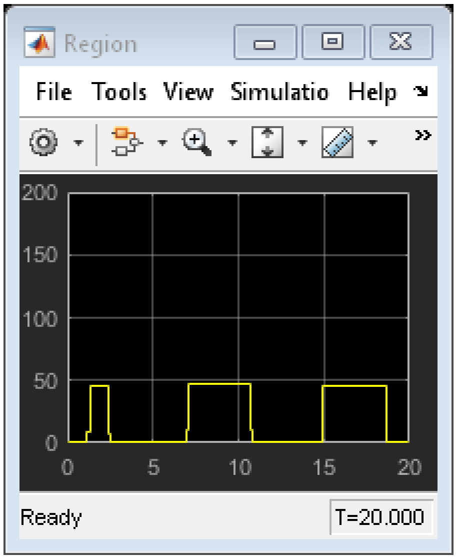
```
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink(R) is required to run this example.')
    return
end
```

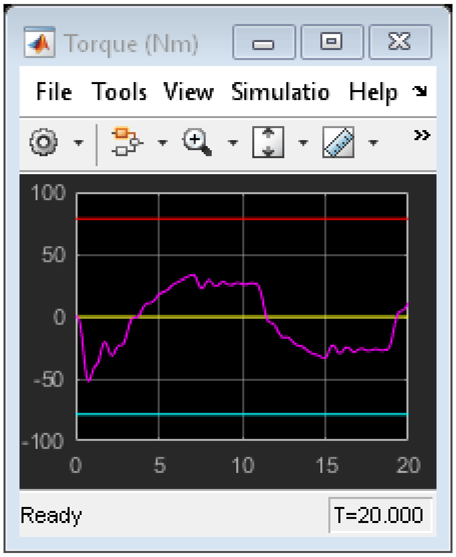
Simulate closed-loop control of the linear plant model in Simulink, using the Explicit MPC Controller block. Controller "mpcobjExplicit" is specified in the block dialog.

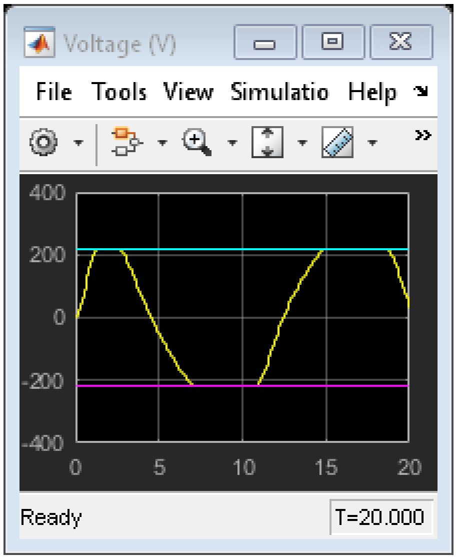
```
mdl = 'empc_motor';
open_system(mdl)
sim(mdl);
```


u(t)
plant
Servomechanism Model
y(t)
Angle (radians)
mv
mo
Explicit MPC
region
ref
Angle reference
r(t)
0
Torque reference
Region
Voltage (V)
-220
Vmin
220
Vmax
-tau
Tmin
tau
Tmax
Torque (Nm)


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


Region
File Tools View Simulatio Help
200
150
100
50
0
0
5
10
15
20
Ready
T=20.000





The closed-loop response is identical to the traditional MPC controller designed in the "mpcmotor" example.

Control Using Sub-optimal Explicit MPC

To reduce the memory footprint, you can use `simplify` command to reduce the number of piecewise affine solution regions. For example, you can remove regions whose Chebychev radius is smaller than .08. However, the price you pay is that the controller performance now becomes sub-optimal.

Use `simplify` command to generate Explicit MPC with sub-optimal solutions.

```
mpcobjExplicitSimplified = simplify(mpcobjExplicit, 'radius', 0.08);
disp(mpcobjExplicitSimplified);
```

```
Regions to analyze:       75/      75 --> 37 regions deleted.

  explicitMPC with properties:

                        MPC: [1x1 mpc]
                      Range: [1x1 struct]
        OptimizationOptions: [1x1 struct]
    PiecewiseAffineSolution: [1x38 struct]
               IsSimplified: 1
```

The number of piecewise affine regions has been reduced.

Compare closed-loop simulation between sub-optimal Explicit MPC and Explicit MPC.

```
[y3,t3,u3] = sim(mpcobjExplicitSimplified, Tf, r);
```

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

The simulation results are not the same.

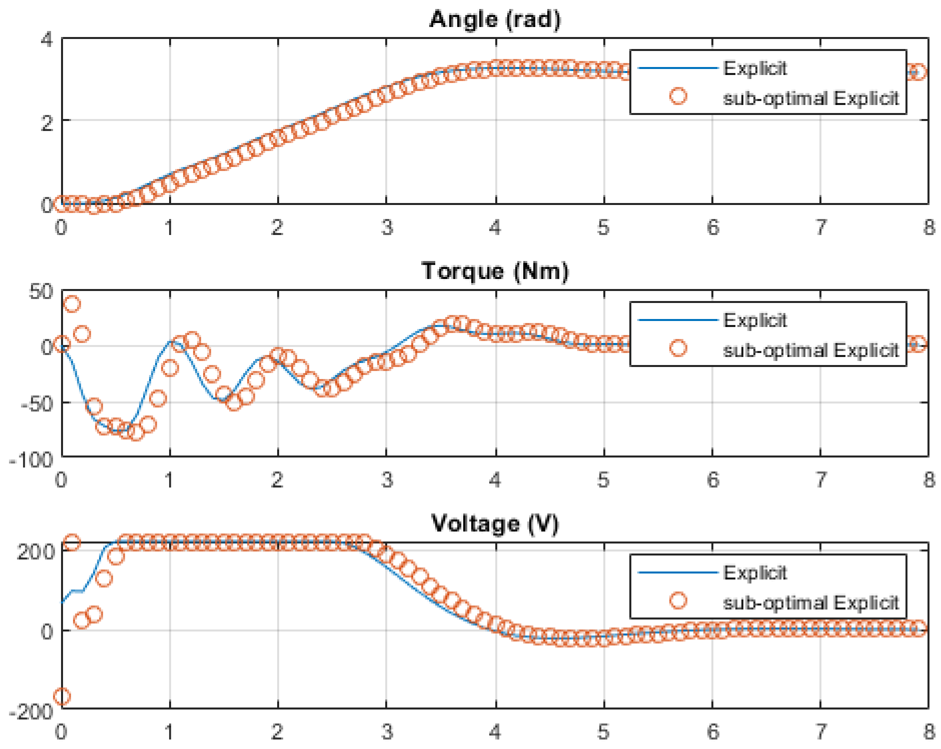
```
fprintf('SIM command: Difference between exact and suboptimal MPC trajectories = %g\n',
```

```
SIM command: Difference between exact and suboptimal MPC trajectories = 439.399
```

Plot results.

```
figure;
subplot(3,1,1)
```

```
plot(t1,y1(:,1),t3,y3(:,1),'o');
grid
title('Angle (rad)')
legend('Explicit','sub-optimal Explicit')
subplot(3,1,2)
plot(t1,y1(:,2),t3,y3(:,2),'o');
grid
title('Torque (Nm)')
legend('Explicit','sub-optimal Explicit')
subplot(3,1,3)
plot(t1,u1,t3,u3,'o');
grid
title('Voltage (V)')
legend('Explicit','sub-optimal Explicit')
```



The simulation result with the sub-optimal Explicit MPC is slightly worse.

```
bdclose(mdl)
```

See Also

More About

- “Explicit MPC” on page 6-2
- “Explicit MPC Control of a Single-Input-Single-Output Plant” on page 6-8
- “Explicit MPC Control of an Aircraft with Unstable Poles” on page 6-20

Explicit MPC Control of an Inverted Pendulum on a Cart

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

Product Requirement

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

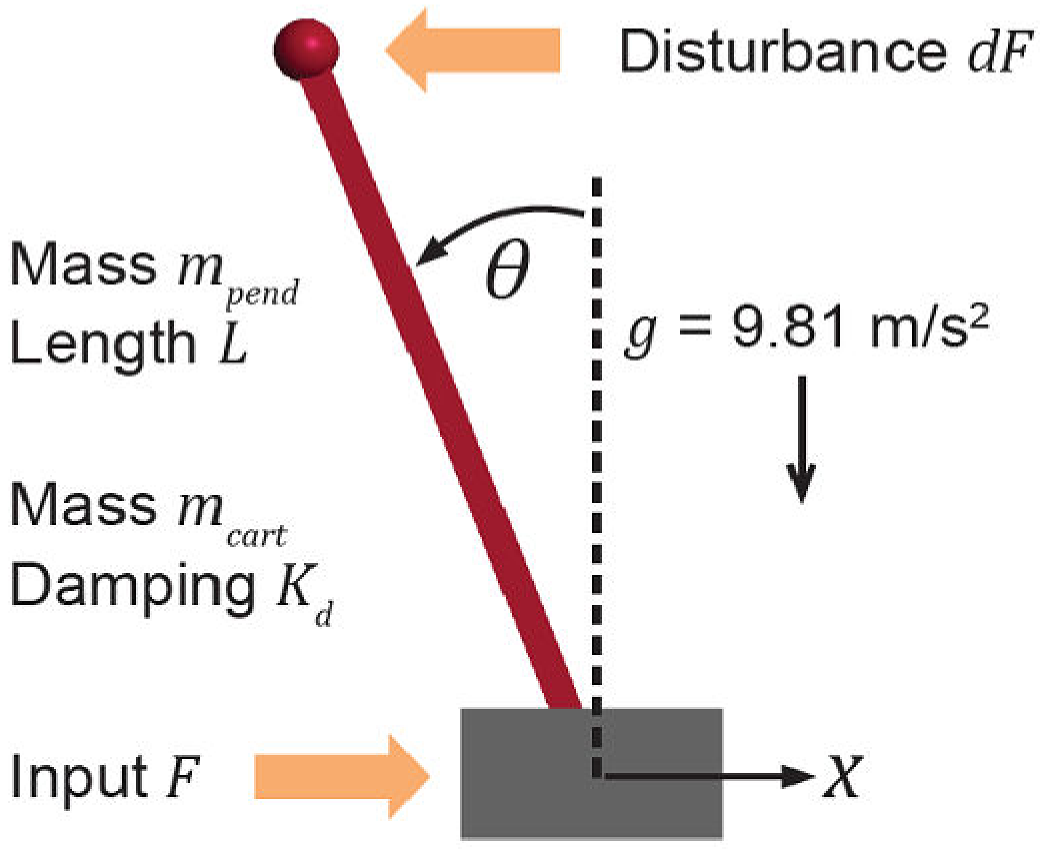
```
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc_featured','main'));
```

Pendulum/Cart Assembly

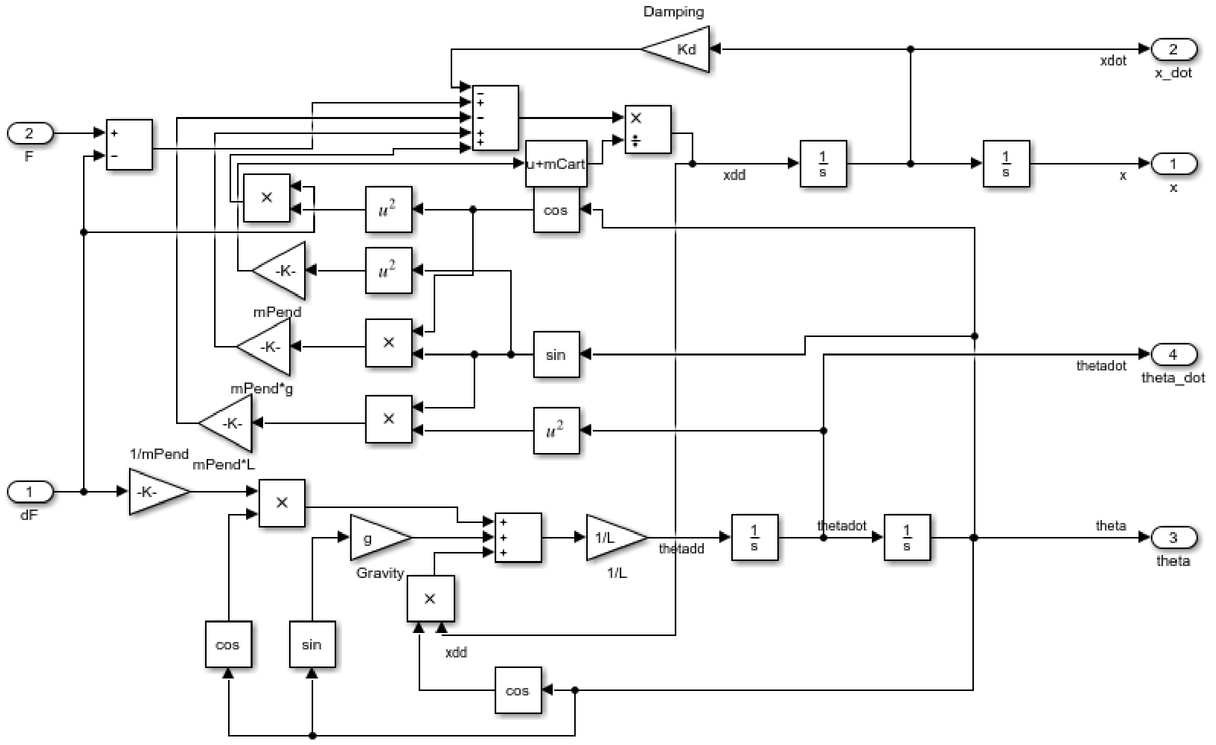
The plant for this example is the following cart/pendulum assembly, where *x* is the cart position and *theta* is the pendulum angle.



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).

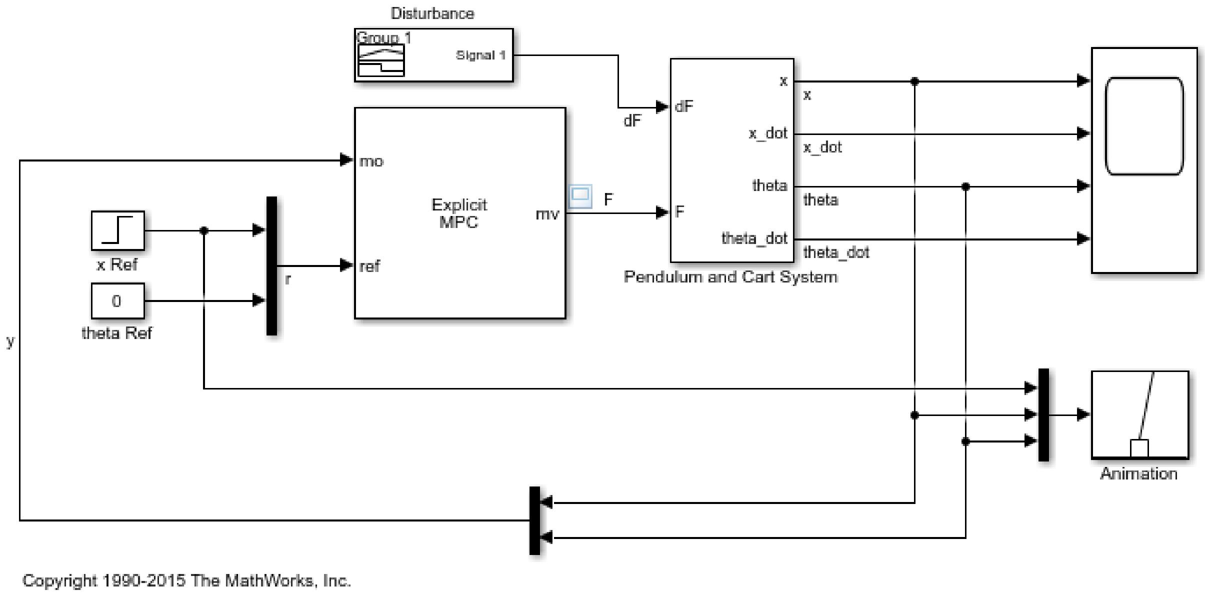
The upright position is an unstable equilibrium for the inverted pendulum, which makes the control task more challenging.

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_pendcartExplicitMPC';
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 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)
```

Traditional (Implicit) 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 defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
   for output(s) y1 and zero weight for output(s) y2
```

There is a limitation on how much force we can apply to the cart, which is specified as hard constraints on manipulated variable *F*.

```
mpcobj.MV.Min = -200;
mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by 100.

```
mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from `0.1` to `1`.

```
mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position *x* and the second weight is associated with angle *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
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Update the output disturbance model.

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

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

Explicit MPC Generation

A simple implicit MPC controller, without the need for constraint or weight changes at run-time, can be converted into an explicit MPC controller with the same control performance. The key benefit of using Explicit MPC is that it avoids real-time optimization, and as a result, is suitable for industrial applications that demand fast sample time. The tradeoff is that explicit MPC has a high memory footprint because optimal solutions for all feasible regions are pre-computed offline and stored for run-time access.

To generate an explicit MPC controller from an implicit MPC controller, define the ranges for parameters such as plant states, references, and manipulated variables. These ranges should cover the operating space for which the plant and controller are designed, to your best knowledge.

```
range = generateExplicitRange(mpcobj);
range.State.Min(:) = -20;   % largest range comes from cart position x
range.State.Max(:) = 20;
range.Reference.Min = -20;  % largest range comes from cart position x
range.Reference.Max = 20;
range.ManipulatedVariable.Min = -200;
range.ManipulatedVariable.Max = 200;
```

```
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Generate an explicit MPC controller for the defined ranges.

```
mpcobjExplicit = generateExplicitMPC(mpcobj,range);
```

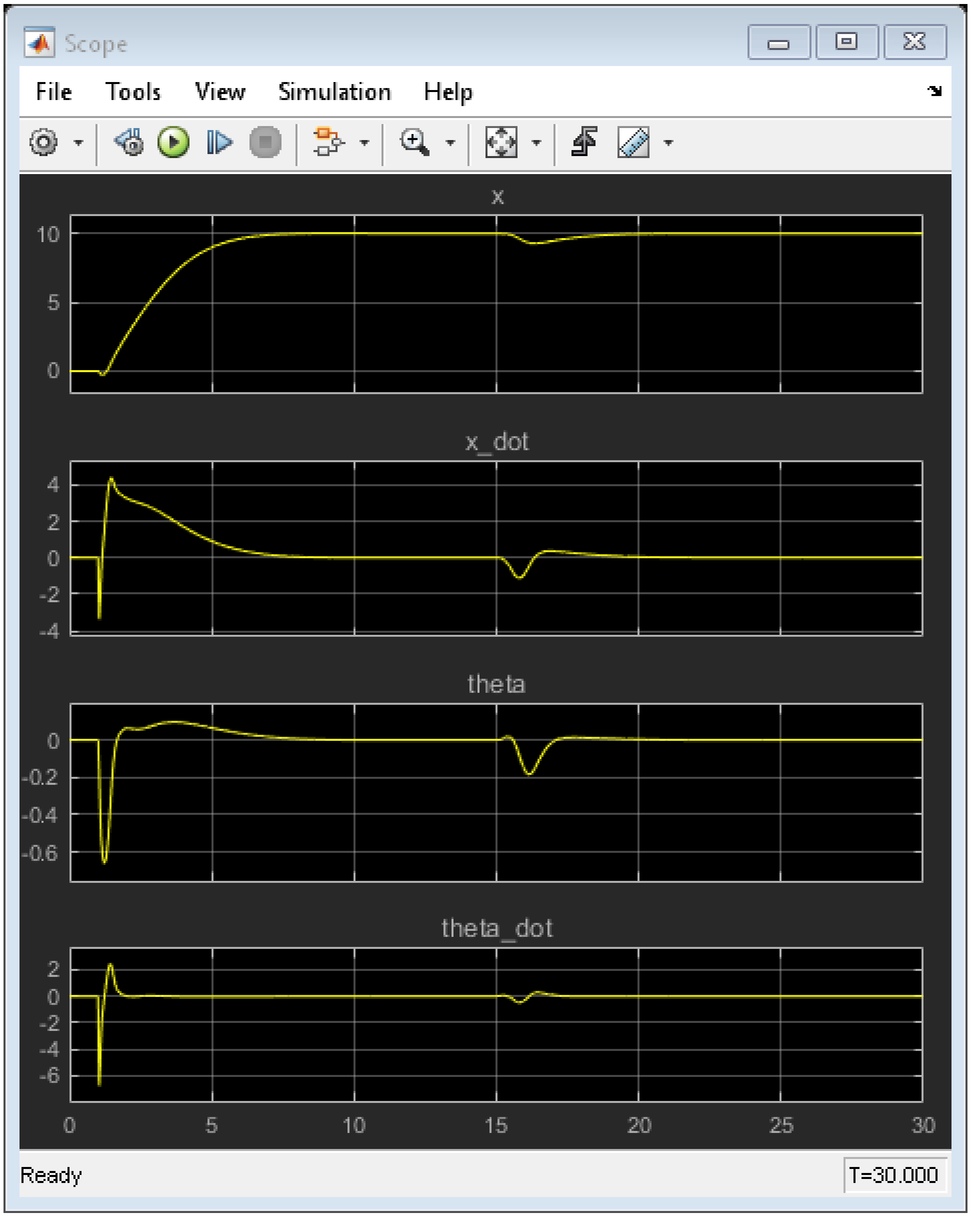
```
Regions found / unexplored:       92/       0
```

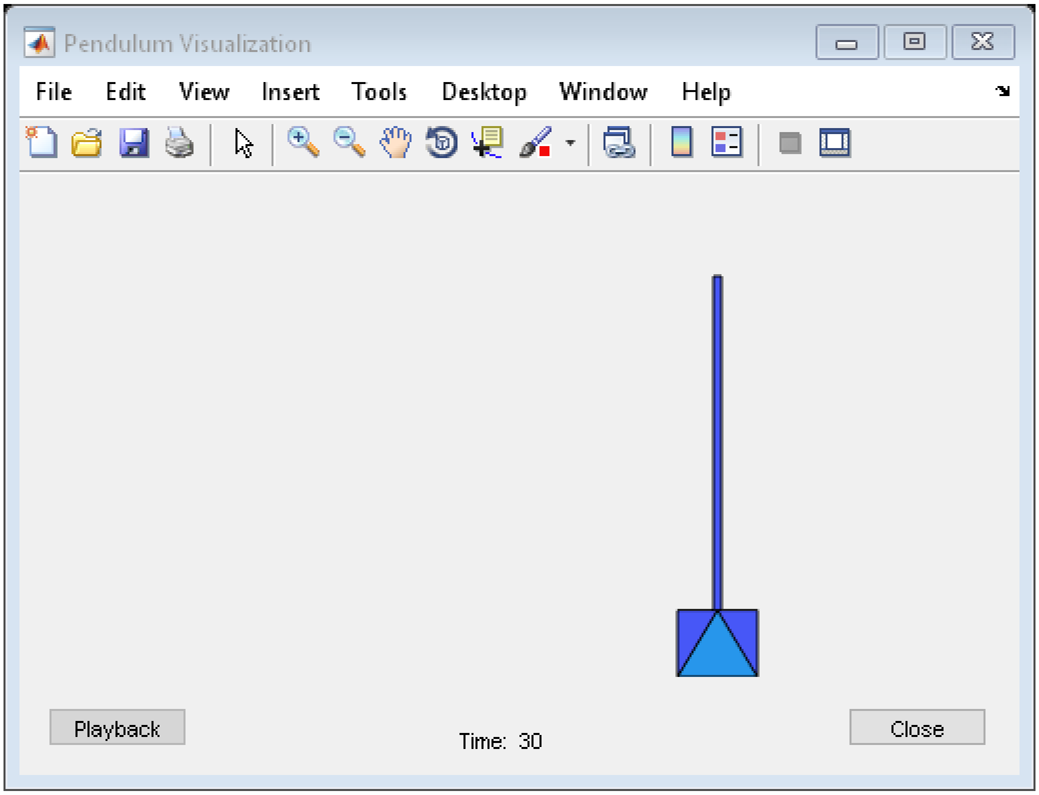
To use the explicit MPC controller in Simulink, specify it in the Explicit MPC Controller block dialog in your Simulink model.

Closed-Loop Simulation

Validate the MPC design with a closed-loop simulation in Simulink.

```
open_system([mdlMPC '/Scope'])
sim(mdlMPC)
```


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
T=30.000



In the nonlinear simulation, all the control objectives are successfully achieved.

Comparing with the results from “Control of an Inverted Pendulum on a Cart”, the implicit and explicit MPC controllers deliver identical performance as expected.

Discussion

It is important to point out that the designed MPC controller has its limitations. For example, if you increase the step setpoint change to `15`, the pendulum fails to recover its upright position during the transition.

To reach the longer distance within the same rise time, the controller applies more force to the cart at the beginning. As a result, the pendulum is displaced from its upright position by a larger angle such as `60` degrees. At such angles, the plant dynamics differ significantly from the LTI predictive model obtained at *theta* = `0`. As a result, errors in the prediction of plant behavior exceed what the built-in MPC robustness can handle, and the controller fails to perform properly.

A simple workaround to avoid the pendulum falling is to restrict pendulum displacement by adding soft output constraints to *theta* and reducing the ECR weight on constraint softening.

```
mpcobj.OV(2).Min = -pi/2;
mpcobj.OV(2).Max = pi/2;
mpcobj.Weights.ECR = 100;
```

However, with these new controller settings, it is no longer possible to reach the longer distance within the required rise time. In other words, controller performance is sacrificed to avoid violation of soft output constraints.

To reach longer distances within the same rise time, the controller needs more accurate models at different angle to improve prediction. Another example “Gain-Scheduled MPC Control of an Inverted Pendulum on a Cart” shows how to use gain scheduling MPC to achieve the longer distances.

Remove the example file folder from the MATLAB path, and close the Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc_featured','main'));
bdclose(mdlMPC)
```

See Also

More About

- “Explicit MPC” on page 6-2
- “Control of an Inverted Pendulum on a Cart” on page 4-104
- “Gain-Scheduled MPC Control of an Inverted Pendulum on a Cart” on page 7-68

Gain Scheduling MPC Design

Gain-Scheduled MPC

Gain-scheduled model predictive control switches between a predefined set of MPC controllers, in a coordinated fashion, to control a nonlinear plant over a wide range of operating conditions. Use this approach if the plant operating characteristics change in a predictable way and the change is such that a single prediction model cannot provide adequate controller performance. This approach is comparable to the use of gain scheduling in conventional feedback control.

To improve efficiency, inactive controllers do not compute optimal control moves. However, to provide bumpless transfer between controllers, the inactive controllers continue to perform state estimation. Bumpless transfer prevents sudden changes in the manipulated variables when the controller switching occurs.

You can design and simulate MPC controllers both in Simulink and at the command line. The Multiple MPC Controllers and Multiple Explicit MPC Controllers blocks enable you to switch between a defined set of MPC Controllers in Simulink. You can perform command-line simulations using the `mpcmoveMultiple` command. However, `mpcmoveMultiple` does not support explicit MPC controllers.

Design Workflow

To implement gain-scheduled MPC, first design a traditional model predictive controller for each operating point, and then design a scheduling signal that switches controllers at run time.

General Design Steps

- Define and tune a nominal MPC controller for the most likely (or average) operating conditions. For more information, see “MPC Design”.
- Use simulations to determine an operating condition at which the nominal controller loses robustness. For more information, see “Simulation”.
- Identify a measurement (or combination of measurements) that indicates when to replace the nominal controller.
- Determine a plant prediction model for the new operating conditions. Its input and output variables must be the same as in the nominal case.
- Define a new MPC controller based on the new prediction model. Use the nominal controller settings as a starting point, and test and retune controller settings if necessary.

- If two controllers are inadequate to provide robustness over the full operational range, consider dividing the range into smaller regions and adding more controllers. Alternatively, you can use an adaptive MPC controller, which has a smaller memory footprint. For more information, see “Adaptive MPC Design”.
- (optional) Consider creating an explicit MPC controller for each traditional MPC controller. Explicit MPC controllers require fewer run-time computations than traditional (implicit) model predictive controllers and are therefore useful for applications that require small sample times. For more information, see “Explicit MPC” on page 6-2.
- In your Simulink model, configure either the Multiple MPC Controllers or Multiple Explicit MPC Controllers block, and specify the switching criterion.
- To verify robustness and bumpless switching, test the controllers over the full operating range using closed-loop simulations.

Tips

- In practice, it is recommended to allow a warm-up period during which the plant operates under manual control while the controller initializes its state estimates. This initialization typically requires 10–20 control intervals. A warm-up is especially important for the Multiple MPC Controllers and Multiple Explicit MPC Controllers blocks. Without an adequate warm-up period, switching between controllers can cause sudden changes in the manipulated variables. Switching on the controllers when the plant is operating far from any of the gain-scheduled operating points can also cause sudden manipulated variable changes.
- If you use custom state estimation, all your gain-scheduled MPC controllers must have the same state dimension. This requirement places implicit restrictions on plant and disturbance models.

See Also

Functions
`mpcmoveMultiple`

Blocks
Multiple Explicit MPC Controllers | Multiple MPC Controllers

More About

- “Schedule Controllers at Multiple Operating Points” on page 7-5

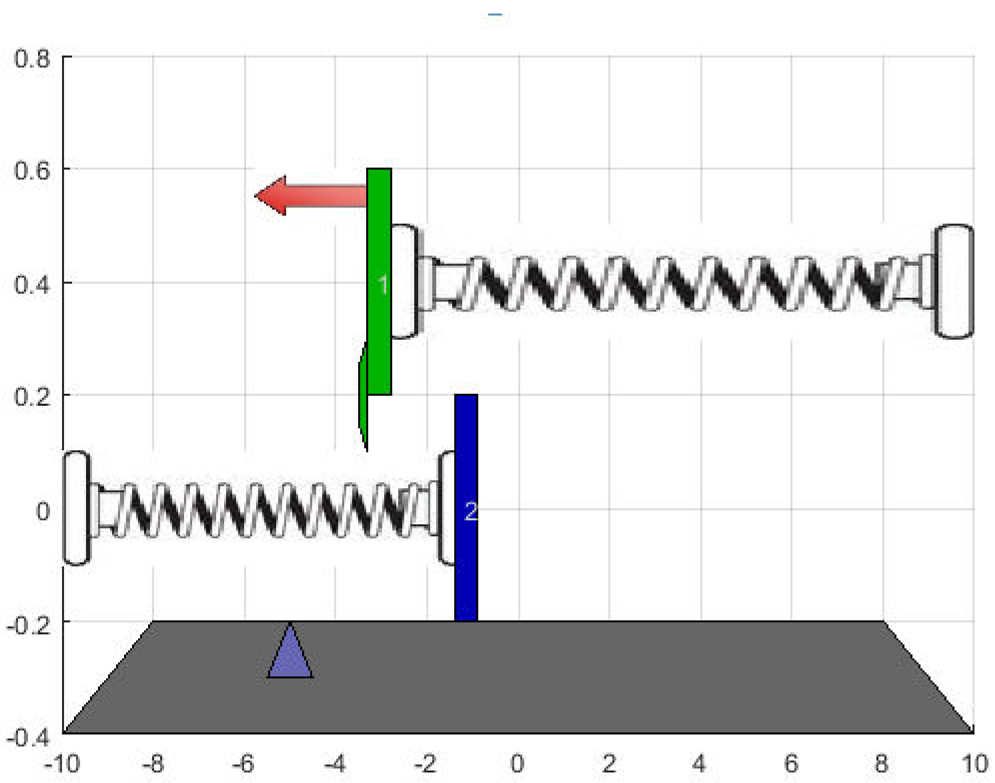
Schedule Controllers at Multiple Operating Points

If your plant is nonlinear, a controller designed to operate in a particular target region may perform poorly in other regions. A common way to compensate is to create multiple controllers, each designed for a particular combination of operating conditions. You can then switch between the controllers in real time as conditions change. For more information, see “Gain-Scheduled MPC” on page 7-2.

The following example shows how to coordinate multiple model predictive controllers for this purpose.

Plant Model

The plant contains two masses, `M1` and `M2`, connected to two springs. A spring with spring constant `k1` pulls mass `M1` to the right, and a spring with spring constant `k2` pulls mass `M2` to the left. The manipulated variable is a force pulling mass `M1` to the left, shown as a red arrow in the following figure.



Both masses move freely until they collide. The collision is inelastic, and the masses stick together until a change in the applied force separates them. Therefore, there are two operating conditions for the system with different dynamics.

The control objective is to make the position of `M1` track a reference signal, shown as a blue triangle in the previous image. Only the position of `M1` and a contact sensor are available for feedback.

Define the model parameters.

```
M1 = 1;       % masses
M2 = 5;
k1 = 1;       % spring constants
k2 = 0.1;
b1 = 0.3;     % friction coefficients
b2 = 0.8;
```

```
yeq1 = 10;    % wall mount positions
yeq2 = -10;
```

Create a state-space model for when the masses are not in contact; that is when mass `M1` is moving freely.

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

Create a state-space model for when the masses are connected.

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

For both models, the:

- States are the position and velocity of `M1`.
- Inputs are the applied force, which is the manipulated variable (MV), and a spring constant calibration signal, which is a measured disturbance (MD).
- Output is the position of `M1`.

Design MPC Controllers

Design one MPC controller for each of the plant models. Both controllers are identical except for their internal prediction models.

Define the same sample time, `Ts`, prediction horizon, `p`, and control horizon, `m`, for both controllers.

```
Ts = 0.2;
p = 20;
m = 1;
```

Create default MPC controllers for each plant model.

```
MPC1 = mpc(sys1,Ts,p,m);
MPC2 = mpc(sys2,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Define constraints for the manipulated variable. Since the applied force cannot change direction, set the lower bound to zero. Also, set a maximum rate of change for the input force. These constraints are the same for both controllers.

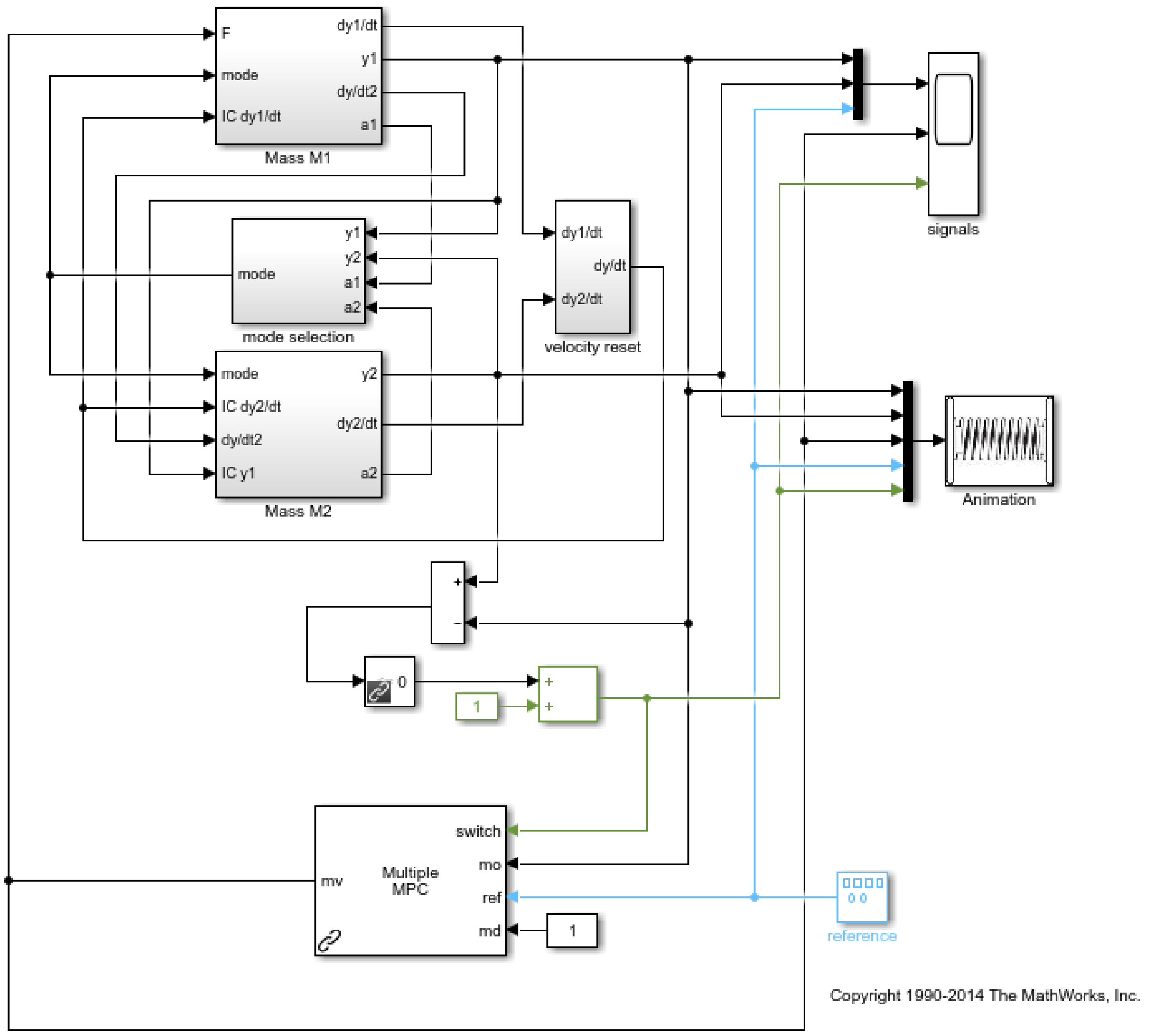
```
MPC1.MV = struct('Min',0,'Max',30,'RateMin',-10,'RateMax',10);
MPC2.MV = MPC1.MV;
```

Simulate Gain-Scheduled Controllers

Simulate the performance of the controllers using the MPC Controller block.

Open the Simulink model.

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



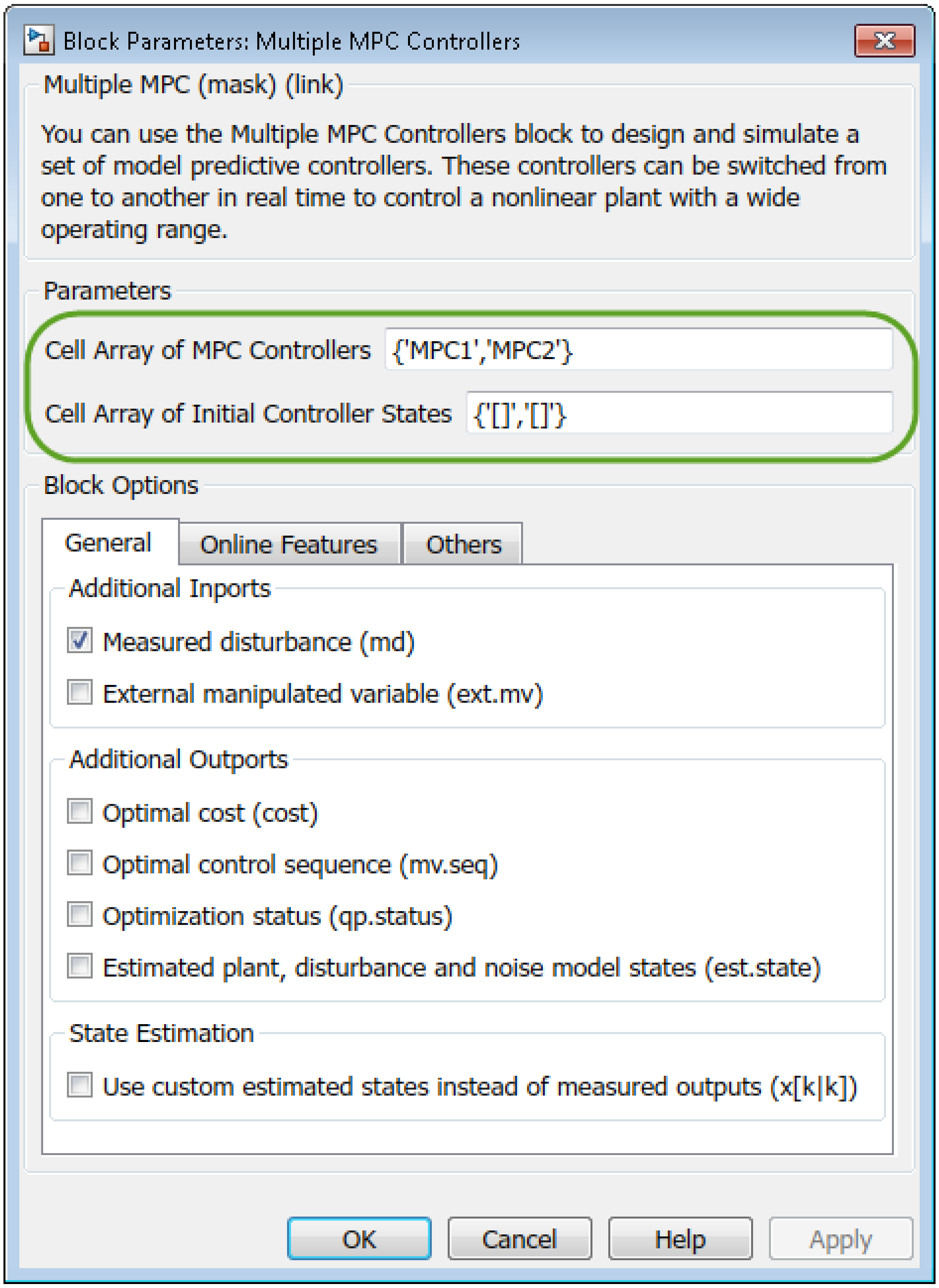
In the model, the Mass M1 subsystem simulates the motion of mass `M1`, both when moving freely and when connected to `M2`. The Mass M2 subsystem simulates the motion of mass M2 when it is moving freely. The mode selection and velocity reset subsystems coordinate the collision and separation of the masses.

The model contains switching logic that detects when the positions of `M1` and `M2` are the same. The resulting switching signal connects to the `switch` inport of the Multiple MPC Controllers block, and controls which MPC controller is active.

Specify the initial position for each mass.

```
y1initial = 0;
y2initial = 10;
```

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 `{'[],'[]'}`.


Block Parameters: Multiple MPC Controllers

Multiple MPC (mask) (link)

You can use the Multiple MPC Controllers block to design and simulate a set of model predictive controllers. These controllers can be switched from one to another in real time to control a nonlinear plant with a wide operating range.

Parameters

Cell Array of MPC Controllers {'MPC1','MPC2'}

Cell Array of Initial Controller States {'[]','[]'}

Block Options

General | Online Features | Others

Additional Inports

- [✓] Measured disturbance (md)
- [ ] External manipulated variable (ext.mv)

Additional Outports

- [ ] Optimal cost (cost)
- [ ] Optimal control sequence (mv.seq)
- [ ] Optimization status (qp.status)
- [ ] Estimated plant, disturbance and noise model states (est.state)

State Estimation

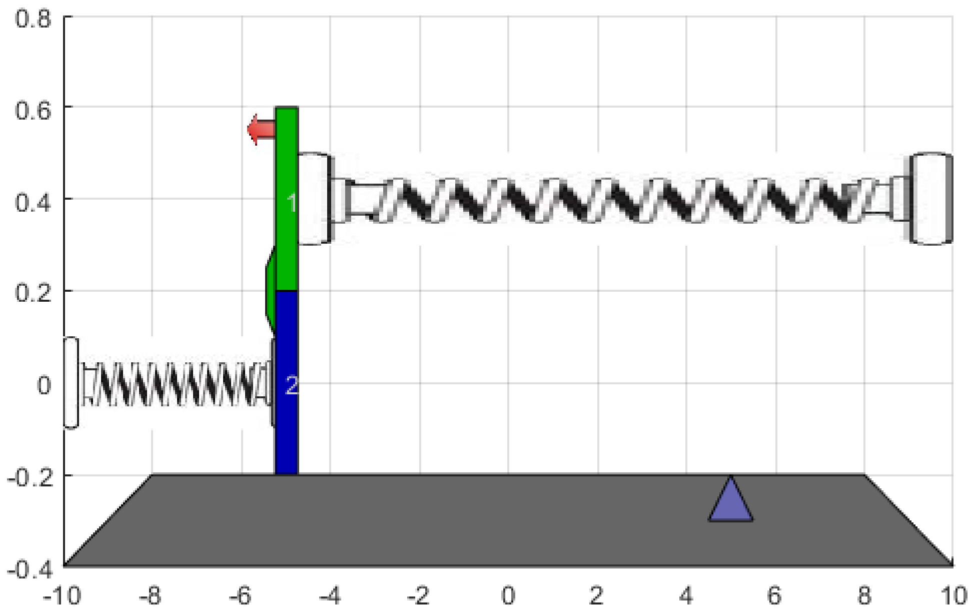
- [ ] Use custom estimated states instead of measured outputs (x[k|k])

Click **OK**.

Run the simulation.

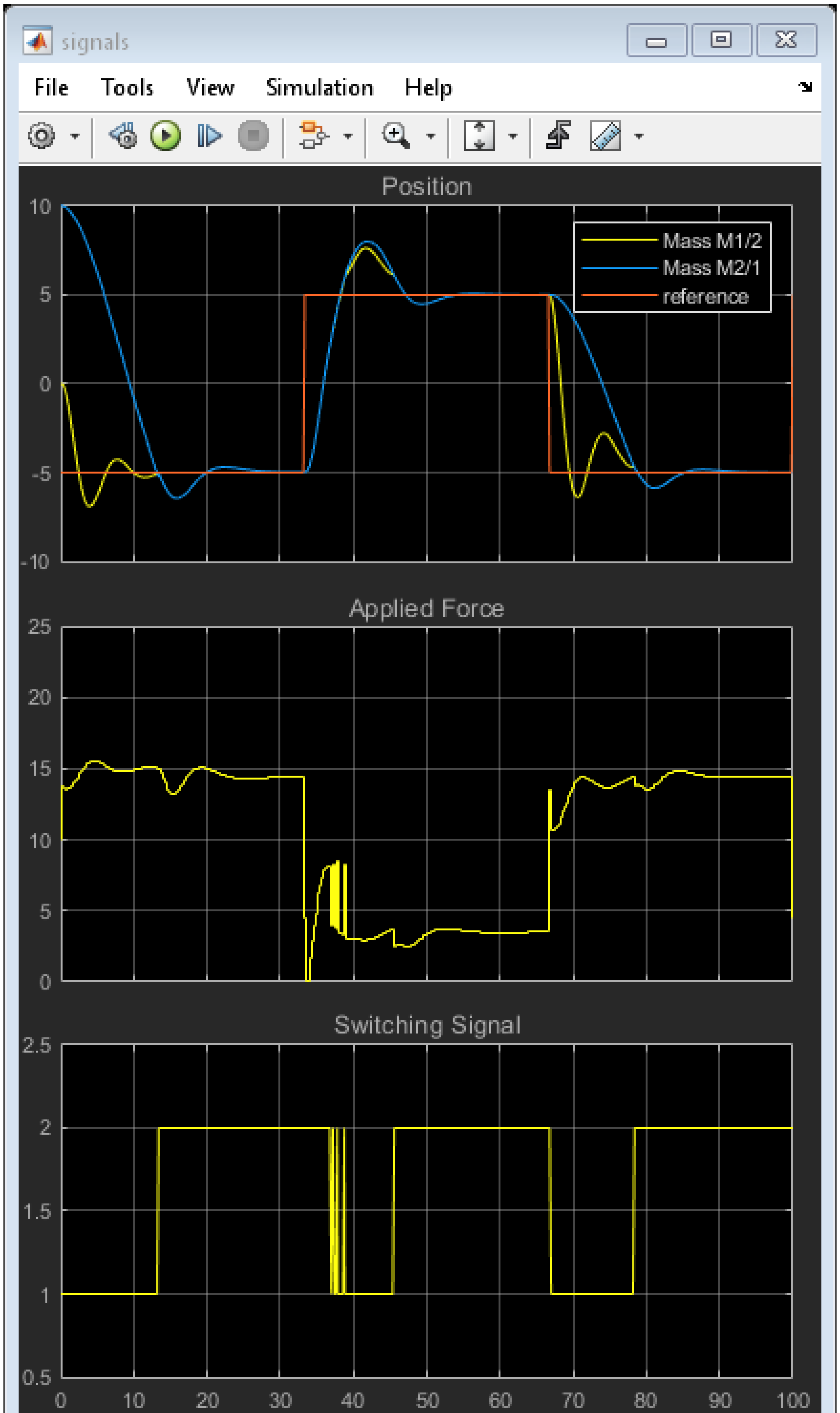
```
sim(mdl)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



To view the simulation results, open the signals scope.

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


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
10
5
0
-5
-10
25
20
15
10
5
0
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100

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

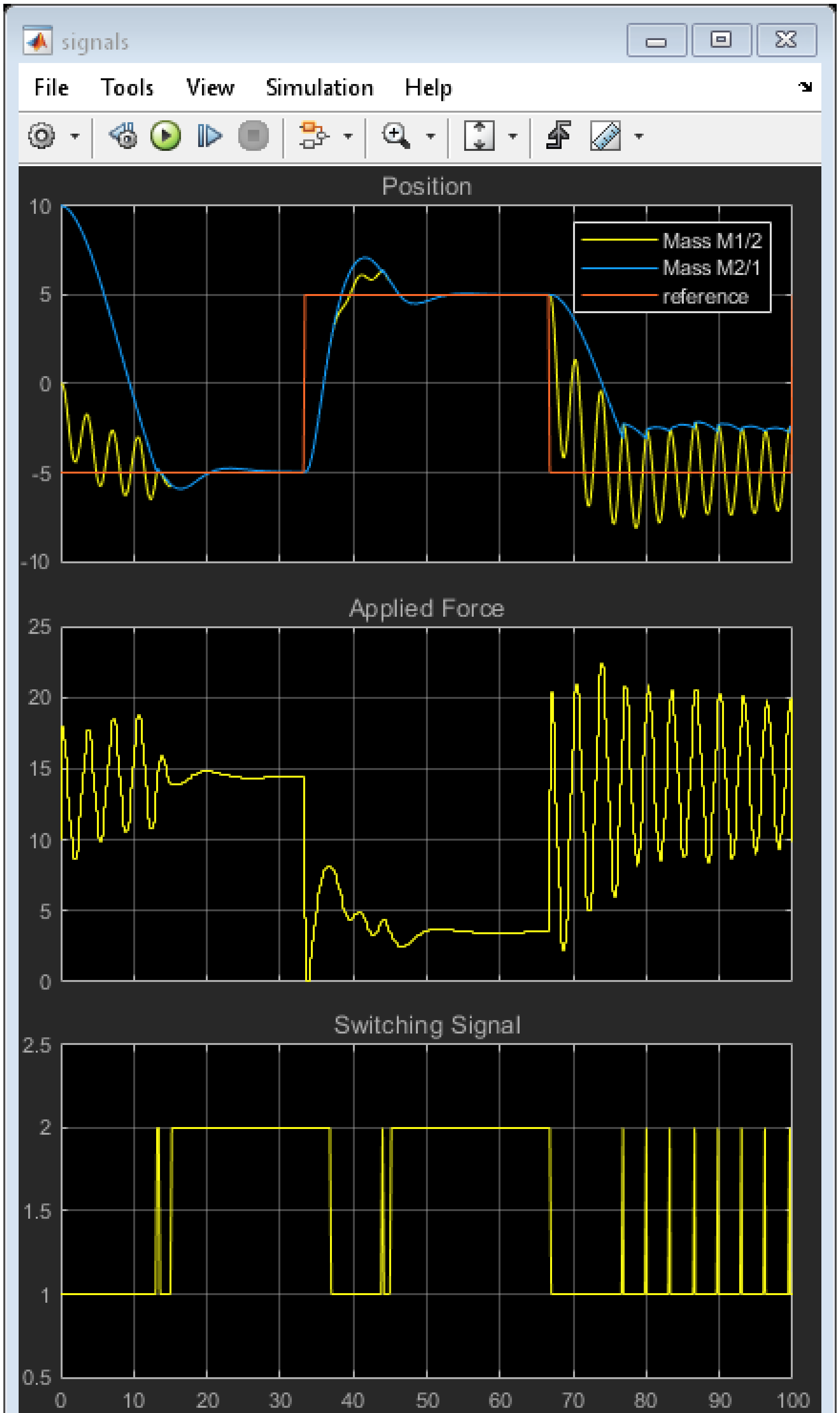
To demonstrate the benefit of using two MPC controllers for this application, simulate the system using just MPC2.

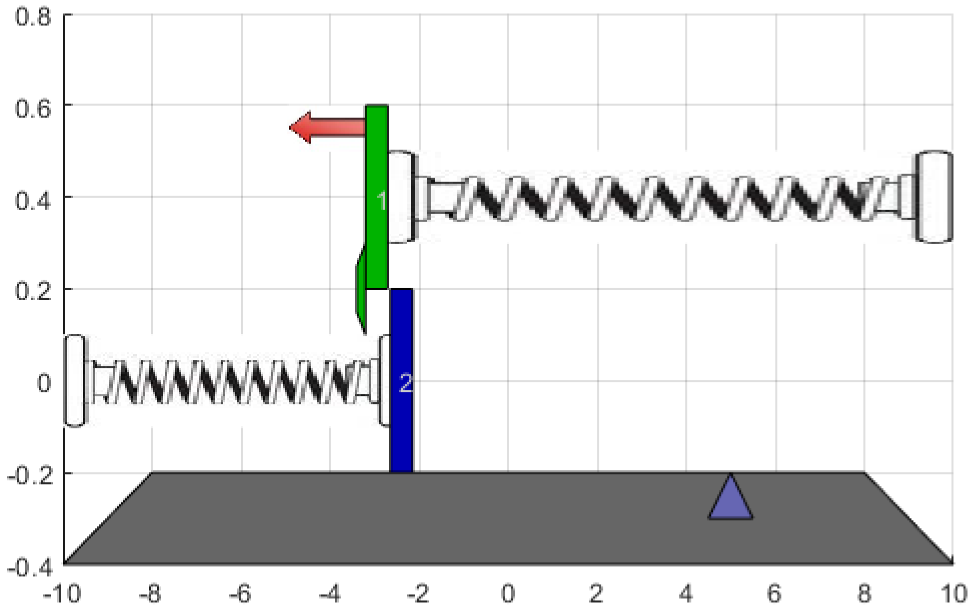
Change MPC1 to match MPC2.

```
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
10
5
0
-5
-10
25
20
15
10
5
0
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100



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`.

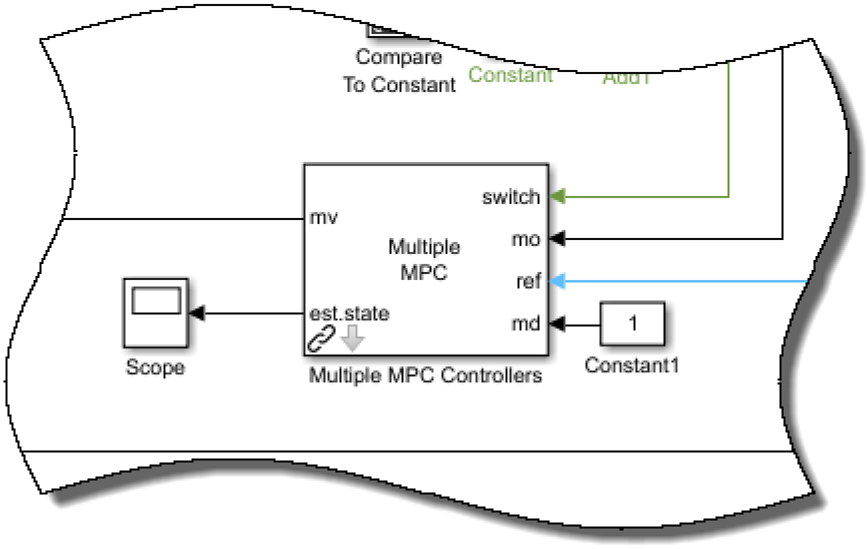
```
range = generateExplicitRange(MPC1);
```

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

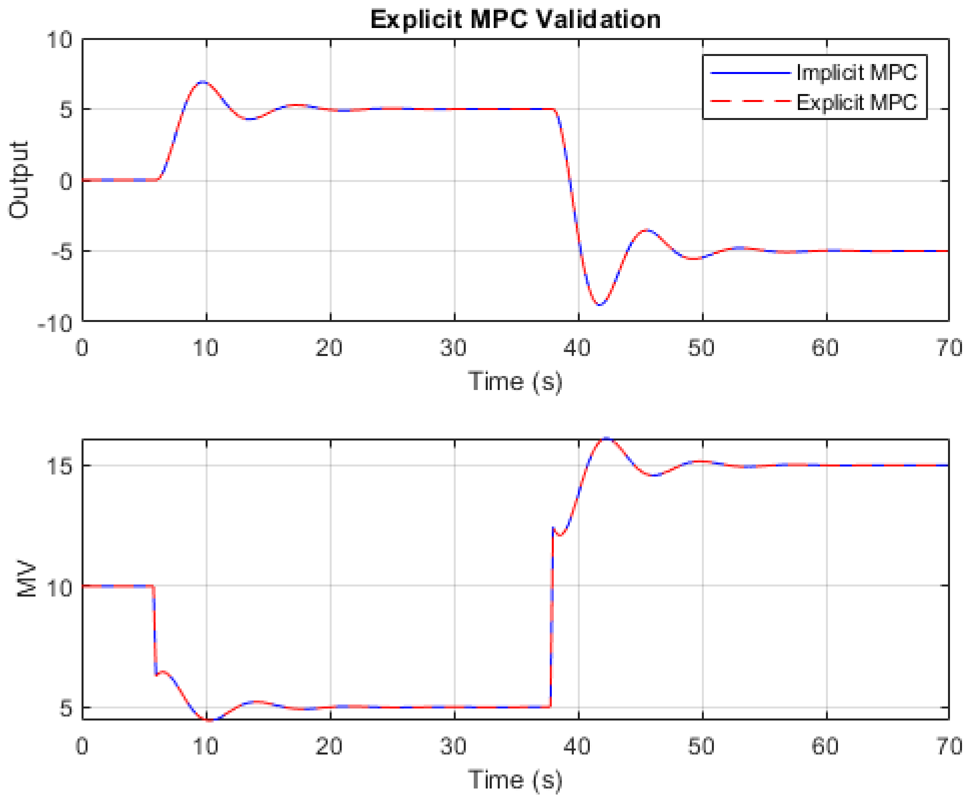
It is good practice to validate the performance of each explicit MPC controller before implementing gain-scheduled explicit MPC. For example, to compare the performance of `MPC1` and `expMPC1`, simulate the closed-loop response of each controller using `sim`.

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

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)')
```



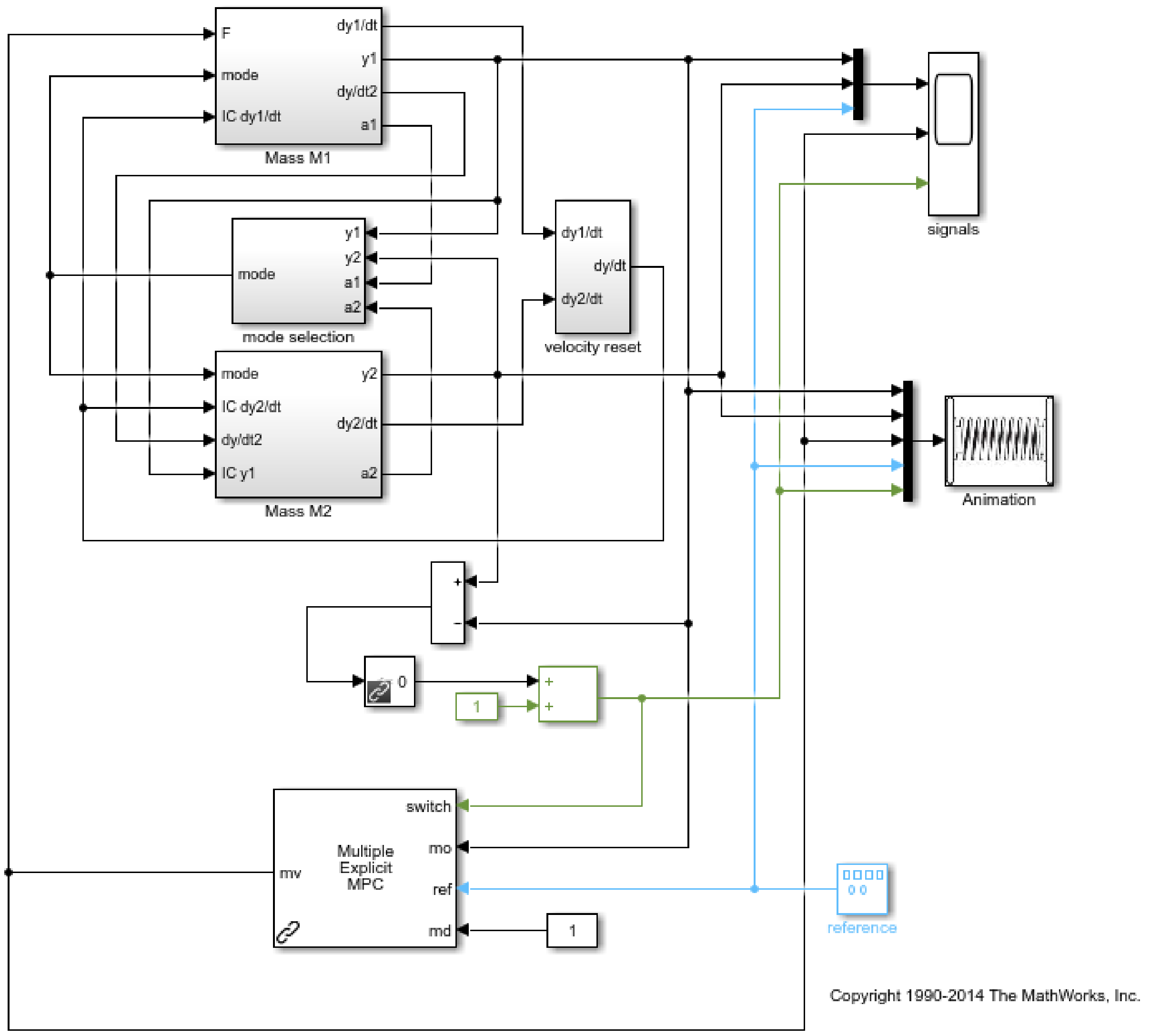
The closed-loop responses and manipulated variable sequences of the implicit and explicit controllers match. Similarly, you can validate the performance of `expMPC2` against that of `MPC2`.

If the responses of the implicit and explicit controllers do not match, adjust the explicit MPC ranges, and create a new explicit MPC controller.

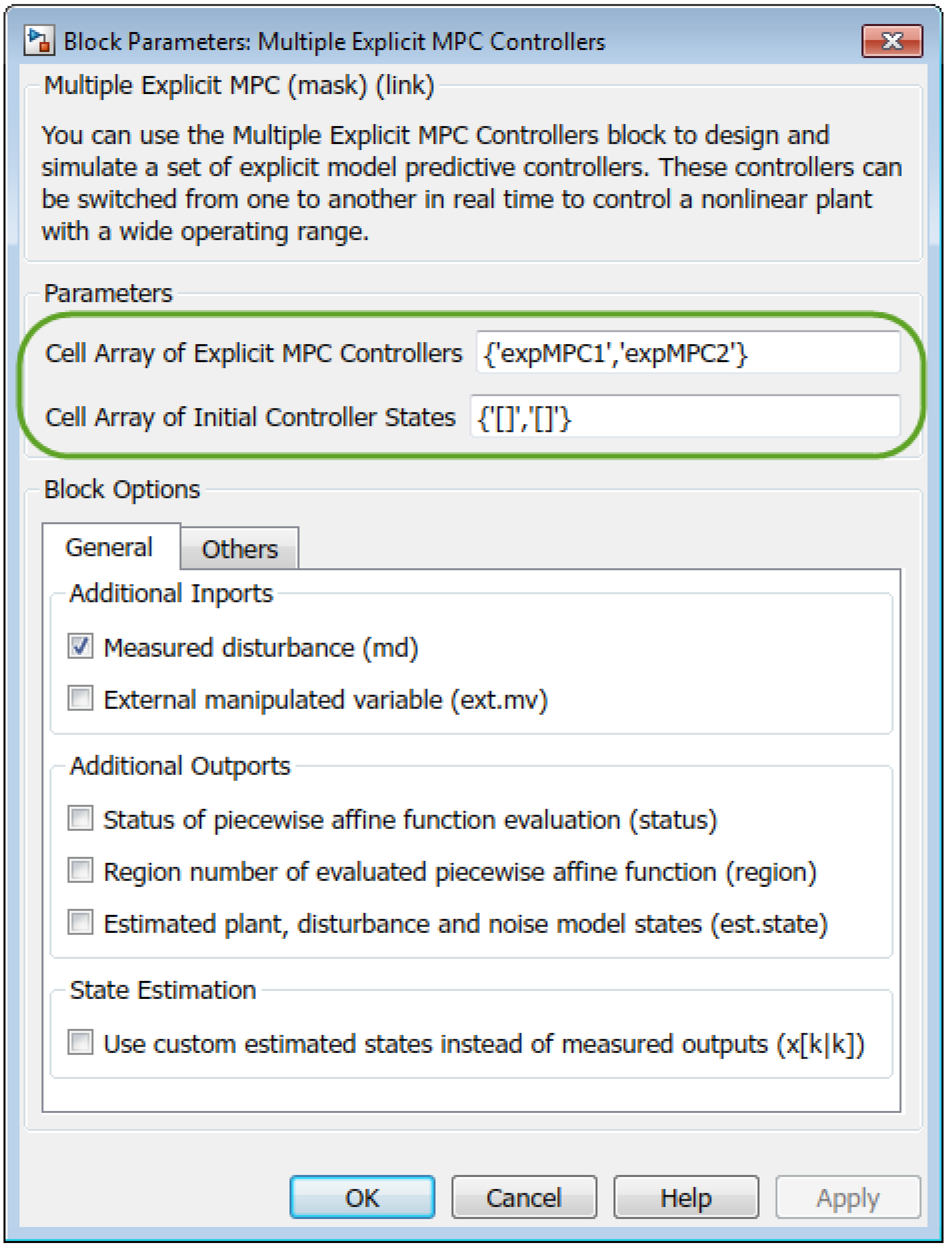
Simulate Gain-Scheduled Explicit MPC

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 `{'[],'[]'}`.



Block Parameters: Multiple Explicit MPC Controllers

Multiple Explicit MPC (mask) (link)

You can use the Multiple Explicit MPC Controllers block to design and simulate a set of explicit model predictive controllers. These controllers can be switched from one to another in real time to control a nonlinear plant with a wide operating range.

Parameters

Cell Array of Explicit MPC Controllers {'expMPC1','expMPC2'}

Cell Array of Initial Controller States {'[]','[]'}

Block Options

General | Others

Additional Inports

✓ Measured disturbance (md)

☐ External manipulated variable (ext.mv)

Additional Outports

☐ Status of piecewise affine function evaluation (status)

☐ Region number of evaluated piecewise affine function (region)

☐ Estimated plant, disturbance and noise model states (est.state)

State Estimation

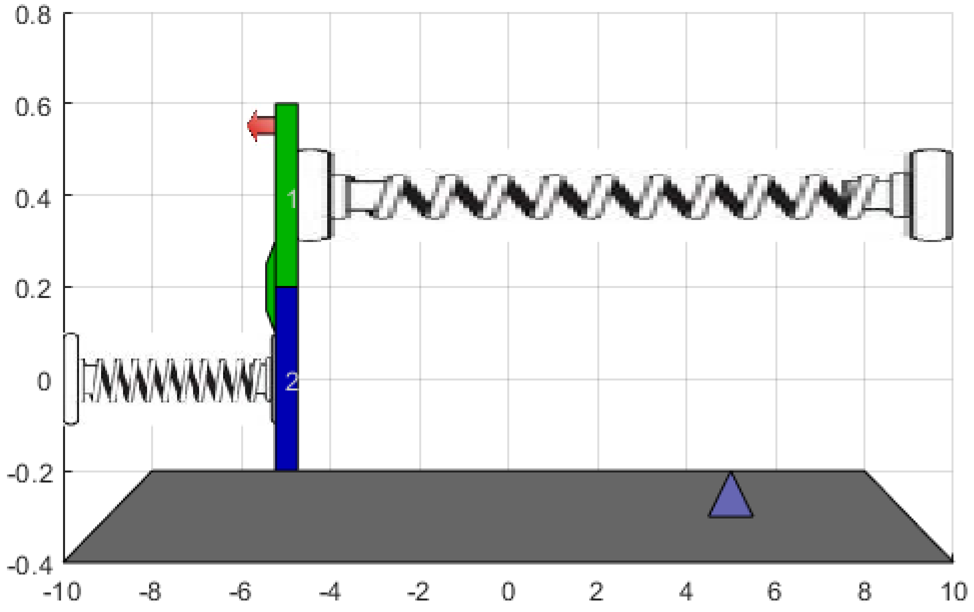
☐ Use custom estimated states instead of measured outputs (x[k|k])

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.

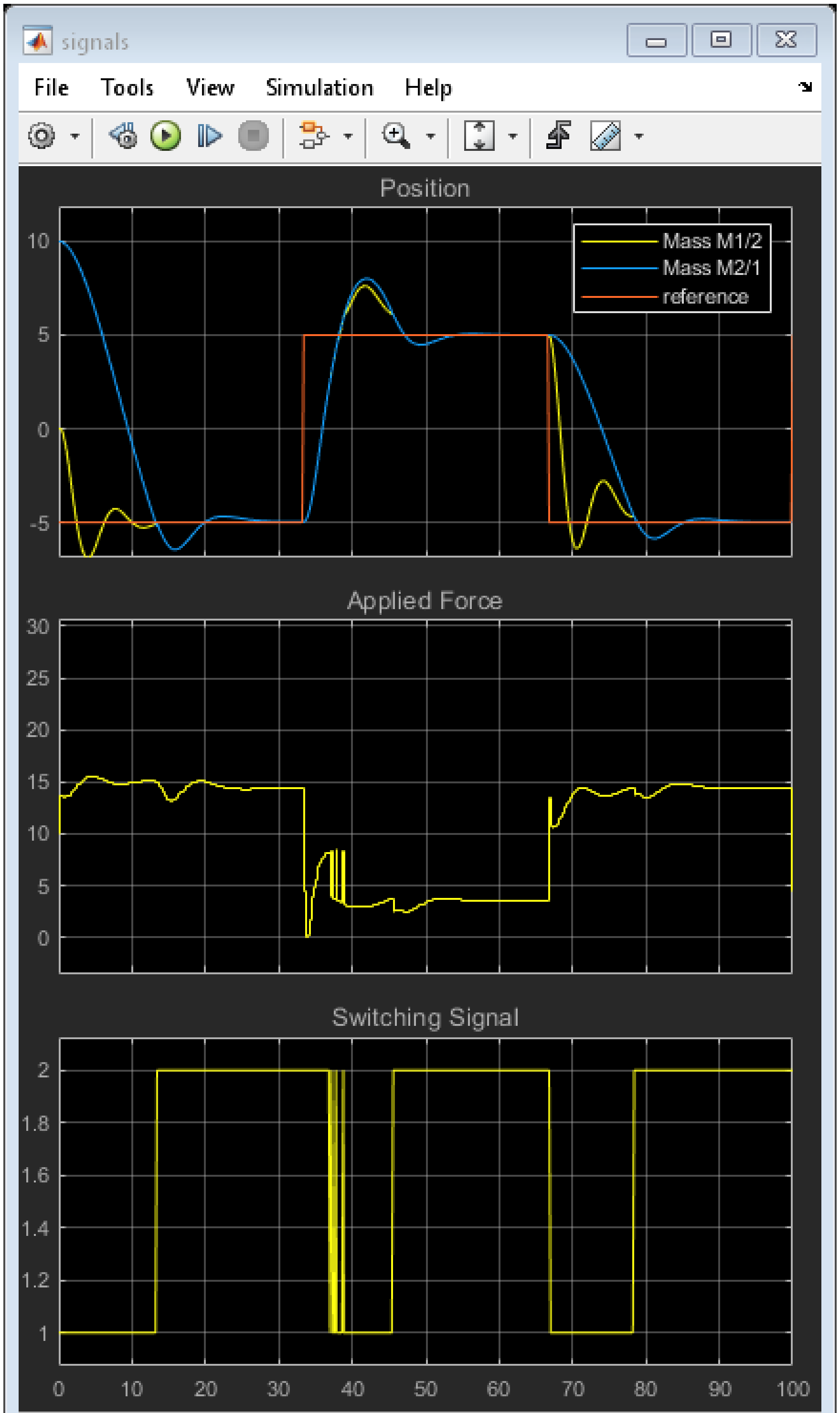
Run the simulation.

```
sim(expModel)
```



To view the simulation results, open the signals scope.

```
open_system([expModel '/signals'])
```


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal

The gain-scheduled explicit MPC controllers provide the same performance as the gain-scheduled implicit MPC controllers.

```
bdclose('all')
```

See Also

Multiple Explicit MPC Controllers | Multiple MPC Controllers

More About

- "Gain-Scheduled MPC" on page 7-2
- "Gain Scheduled Implicit and Explicit MPC Control of Mass-Spring System" on page 7-50

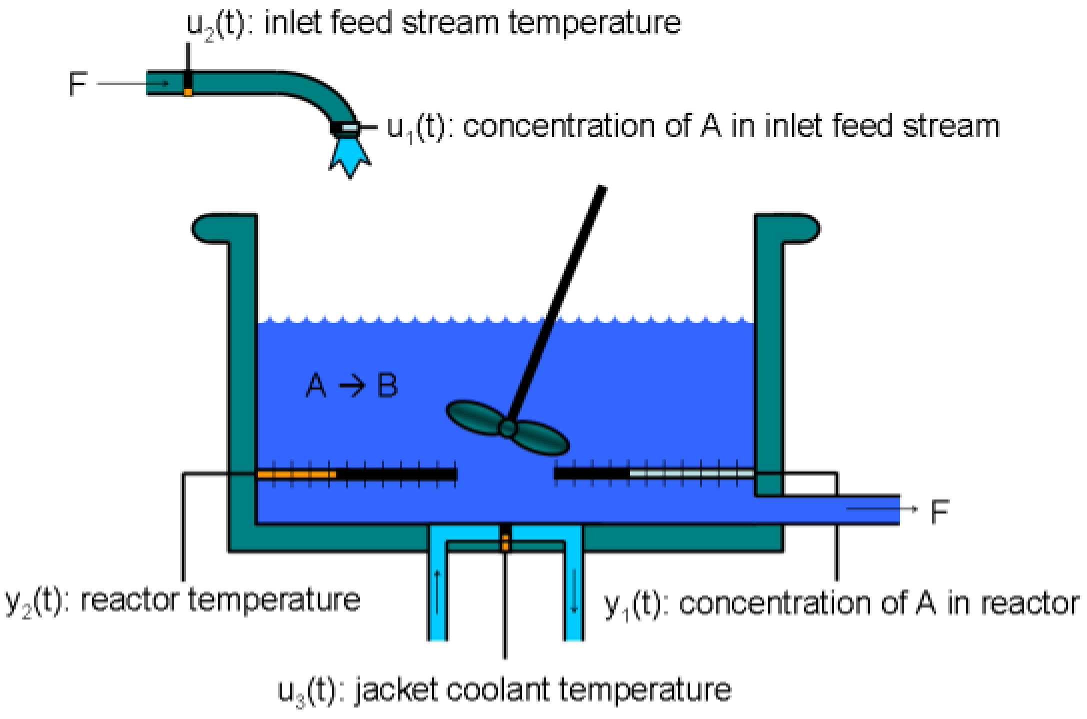
Gain-Scheduled MPC Control of Nonlinear Chemical Reactor

This example shows how to use multiple MPC controllers to control a nonlinear continuous stirred tank reactor (CSTR) as it transitions from low conversion rate to high conversion rate.

Multiple MPC Controllers are designed at different operating conditions and then implemented with the Multiple MPC Controller block in Simulink. At run time, a scheduling signal is used to switch controller from one to another.

About the Continuous Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:


$u_2(t)$: inlet feed stream temperature
F
$u_1(t)$: concentration of A in inlet feed stream
A → B
F
$y_2(t)$: reactor temperature
$y_1(t)$: concentration of A in reactor
$u_3(t)$: jacket coolant temperature

This is a jacketed non-adiabatic tank reactor described extensively in Seborg's book, "Process Dynamics and Control", published by Wiley, 2004. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> B, takes place. The inlet stream of reagent A is fed to the tank at a constant volumetric rate. The product stream exits continuously at the same volumetric rate and liquid density is constant. Thus the volume of reacting liquid is constant.

The inputs of the CSTR model are:

$$\begin{array}{ll} u_1 = CA_i & \text{Concentration of A in inlet feed stream}[kgmol/m^3] \\ u_2 = T_i & \text{Inlet feed stream temperature}[K] \\ u_3 = T_c & \text{Jacket coolant temperature}[K] \end{array}$$

and the outputs (y(t)), which are also the states of the model (x(t)), are:

$$\begin{array}{ll} y_1 = x_1 = CA & \text{Concentration of A in reactor tank}[kgmol/m^3] \\ y_2 = x_2 = T & \text{Reactor temperature}[K] \end{array}$$

The control objective is to maintain the concentration of reagent A, CA at its desired setpoint, which changes over time when reactor transitions from low conversion rate to high conversion rate. The coolant temperature T_c is the manipulated variable used by the MPC controller to track the reference. The inlet feed stream concentration and temperature are assumed to be constant. The Simulink model `mpc_cstr_plant` implements the nonlinear CSTR plant.

About Gain Scheduled Model Predictive Control

It is well known that the CSTR dynamics are strongly nonlinear with respect to reactor temperature variations and can be open-loop unstable during the transition from one operating condition to another. A single MPC controller designed at a particular operating condition cannot give satisfactory control performance over a wide operating range.

To control the nonlinear CSTR plant with linear MPC control technique, you have a few options:

- If a linear plant model cannot be obtained at run time, first you need to obtain several linear plant models offline at different operating conditions that cover the typical operating range. Next you can choose one of the two approaches to implement MPC control strategy:

(1) Design several MPC controllers offline, one for each plant model. At run time, use Multiple MPC Controller block that switches MPC controllers from one to another based

on a desired scheduling strategy, as discussed in this example. Use this approach when the plant models have different orders or time delays.

(2) Design one MPC controller offline at a nominal operating point. At run time, use Adaptive MPC Controller block (updating predictive model at each control interval) together with Linear Parameter Varying (LPV) System block (supplying linear plant model with a scheduling strategy). See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter Varying System” for more details. Use this approach when all the plant models have the same order and time delay.

- If a linear plant model can be obtained at run time, you should use Adaptive MPC Controller block to achieve nonlinear control. There are two typical ways to obtain a linear plant model online:

(1) Use successive linearization. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” for more details. Use this approach when a nonlinear plant model is available and can be linearized at run time.

(2) Use online estimation to identify a linear model when loop is closed. See “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” for more details. Use this approach when linear plant model cannot be obtained from either an LPV system or successive linearization.

Obtain Linear Plant Model at Initial Operating Condition

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

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

First, a linear plant model is obtained at the initial operating condition, CAi is 10 kgmol/m^3, Ti and Tc are 298.15 K. Functions from Simulink Control Design such as `operspec`, `findop`, and `linearize`, are used to generate the linear state-space system from the Simulink model.

Create operating point specification.

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

Feed concentration is known at the initial condition.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known at the initial condition.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Coolant temperature is known at the initial condition.

```
op.Inputs(3).u = 298.15;
op.Inputs(3).Known = true;
```

Compute initial condition.

```
[op_point,op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x; op_report.States(2).x];
y0 = [op_report.Outputs(1).y; op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u; op_report.Inputs(2).u; op_report.Inputs(3).u];
```

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

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

Operating point specifications were successfully met.
States:
----------
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           311      dx:      8.12e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:          8.57      dx:     -6.87e-12 (0)

Inputs:
----------
(1.) mpc_cstr_plant/CAi
      u:            10
```

```
(2.) mpc_cstr_plant/Ti
      u:           298
(3.) mpc_cstr_plant/Tc
      u:           298

Outputs:
----------
(1.) mpc_cstr_plant/T
      y:           311    [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:          8.57    [-Inf Inf]
```

Obtain linear model at the initial condition.

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

Verify that the linear model is open-loop stable at this condition.

```
eig(plant)
```

```
ans =

   -0.5223
   -0.8952
```

Design MPC Controller for Initial Operating Condition

You design an MPC at the initial operating condition.

```
Ts = 0.5;
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```
plant.InputGroup.UnmeasuredDisturbances = [1 2];
plant.InputGroup.ManipulatedVariables = 3;
plant.OutputGroup.Measured = [1 2];
plant.InputName = {'CAi','Ti','Tc'};
plant.OutputName = {'T','CA'};
```

Create MPC controller with default prediction and control horizons

```
mpcobj = mpc(plant,Ts);
```

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

Set nominal values in the controller. Note that nominal values for unmeasured disturbance must be zero.

```
mpcobj.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 0]);
```

Set scale factors because plant input and output signals have different orders of magnitude.

```
Uscale = [10;30;50];
Yscale = [50;10];
mpcobj.DV(1).ScaleFactor = Uscale(1);
mpcobj.DV(2).ScaleFactor = Uscale(2);
mpcobj.MV.ScaleFactor = Uscale(3);
mpcobj.OV(1).ScaleFactor = Yscale(1);
mpcobj.OV(2).ScaleFactor = Yscale(2);
```

The goal will be to track a specified transition in the reactor concentration. The reactor temperature will be measured and used in state estimation but the controller will not attempt to regulate it directly. It will vary as needed to regulate the concentration. Thus, set its MPC weight to zero.

```
mpcobj.Weights.OV = [0 1];
```

Plant inputs 1 and 2 are unmeasured disturbances. By default, the controller assumes integrated white noise with unit magnitude at these inputs when configuring the state estimator. Try increasing the state estimator signal-to-noise by a factor of 10 to improve disturbance rejection performance.

```
Dist = ss(getindist(mpcobj));
Dist.B = eye(2)*10;
setindist(mpcobj,'model',Dist);
```

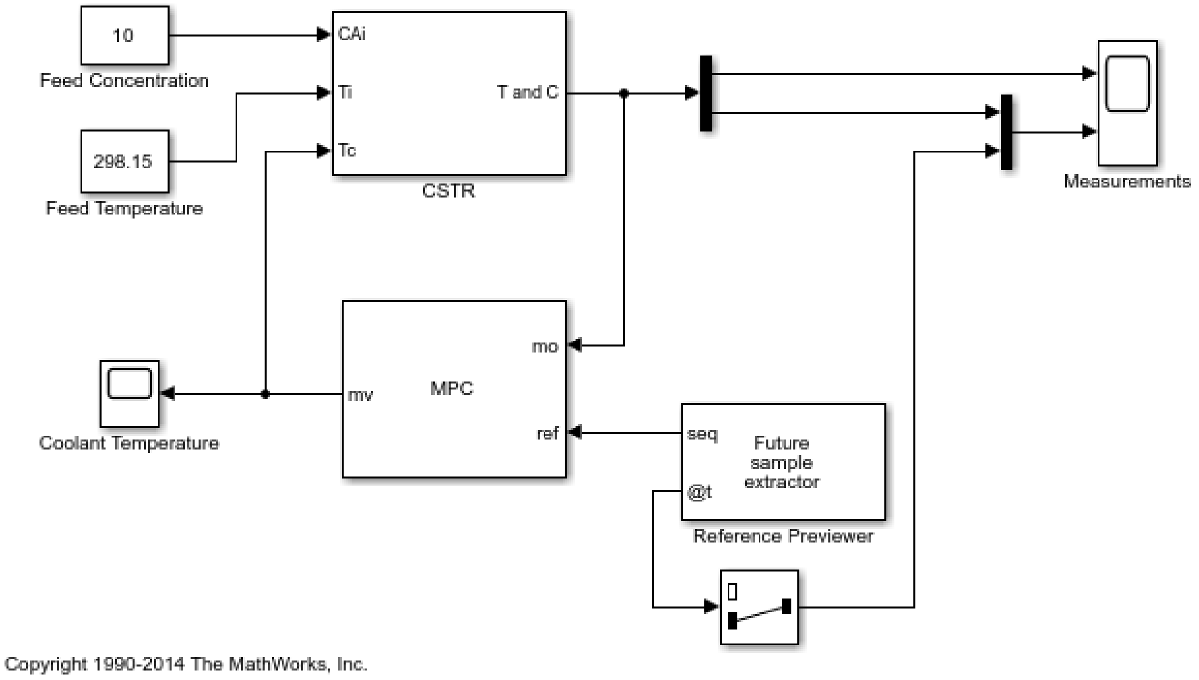
```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #1 is integrated white noise.
   Assuming unmeasured input disturbance #2 is integrated white noise.
   Assuming no disturbance added to measured output channel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

All other MPC parameters are at their default values.

Test the Controller With a Step Disturbance in Feed Concentration

"mpc_cstr_single" contains a Simulink® model with CSTR and MPC Controller blocks in a feedback configuration.

```
mpc_mdl = 'mpc_cstr_single';
open_system(mpc_mdl)
```



Note that the MPC Controller block is configured to look ahead 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]';
```

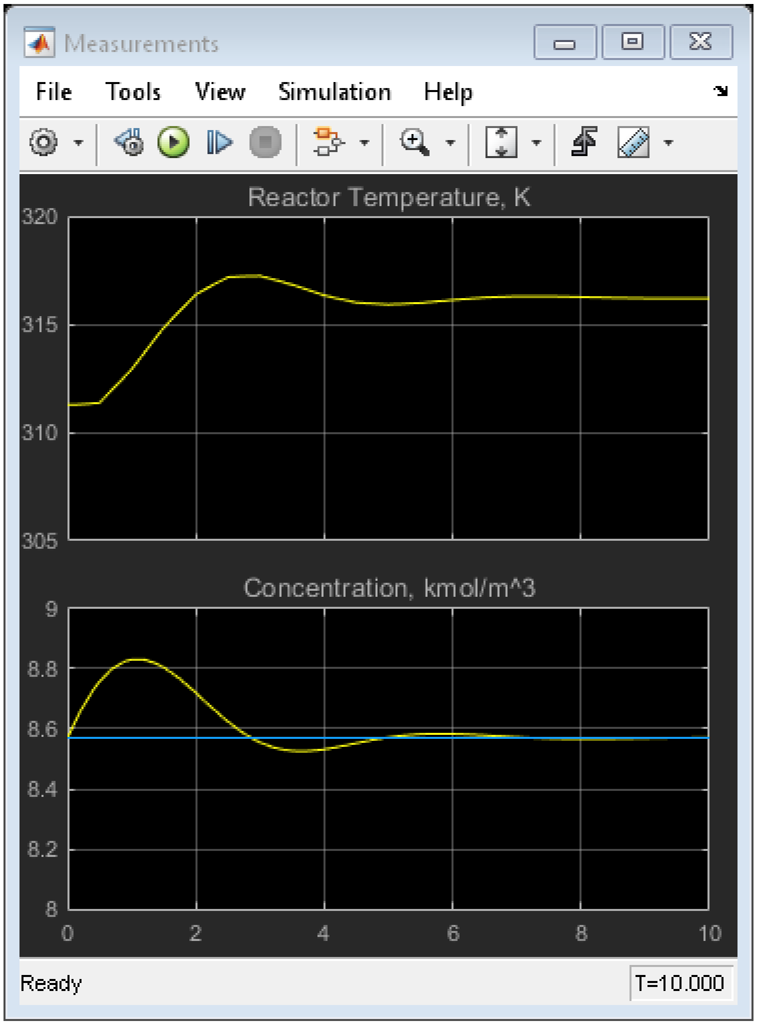
Test the response to a 5% increase in feed concentration.

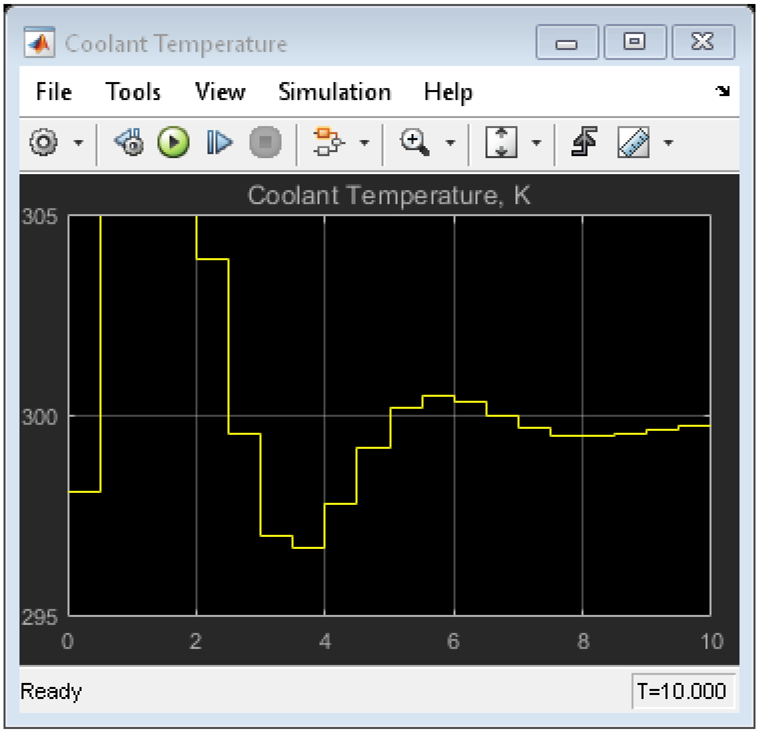
```
set_param([mpc_mdl '/Feed Concentration'],'Value','10.5');
```

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 eac
```


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
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.

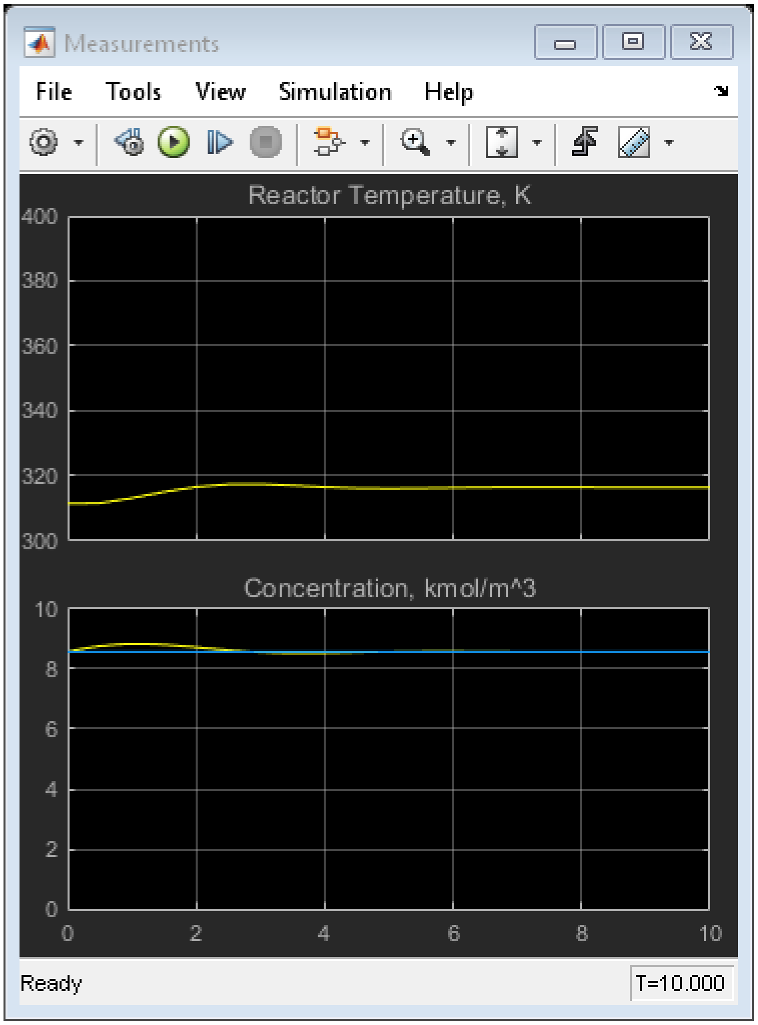
```
CSTR_Setpoints.time = [0 10 11:39]';
CSTR_Setpoints.signals.values = [y0(1)*ones(31,1),[y0(2);y0(2);(y0(2):-0.25:2)';2;2]];
```

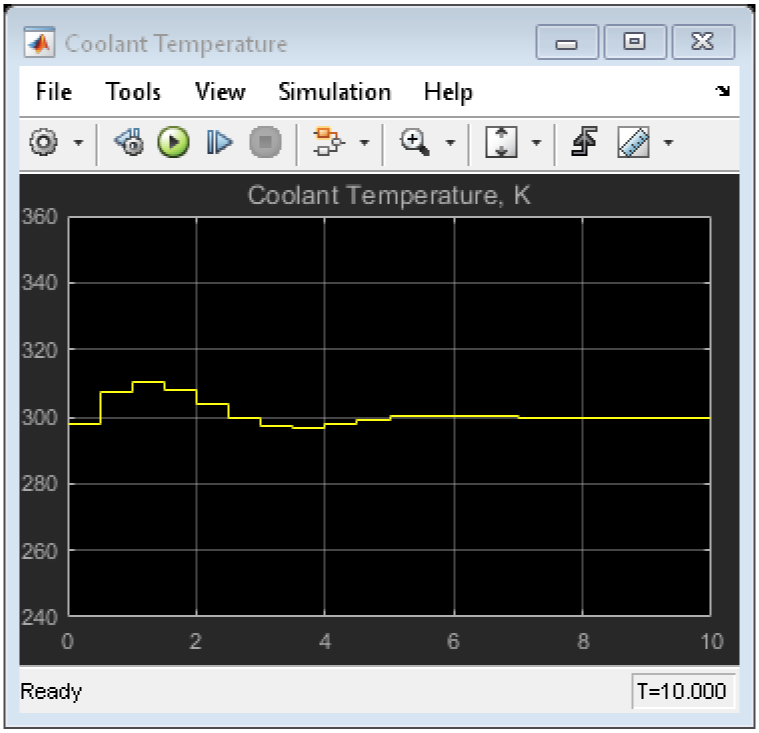
Remove the 5% increase in feed concentration used previously.

```
set_param([mpc_mdl '/Feed Concentration'],'Value','10')
```

Set plot scales and simulate the response.

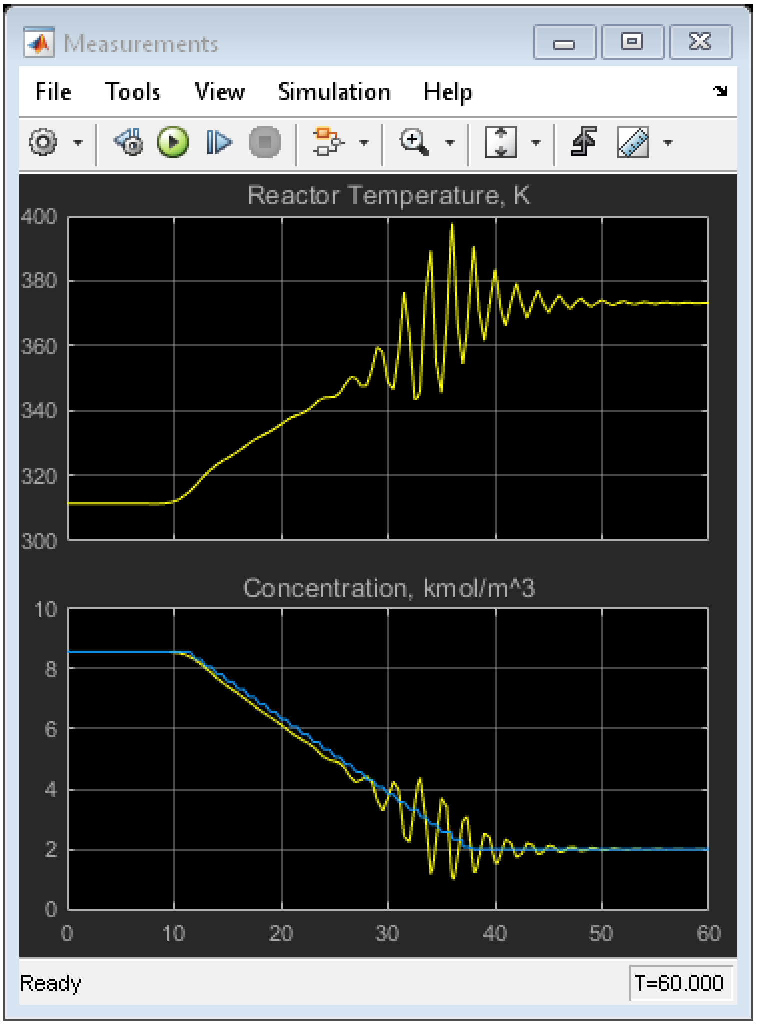
```
set_param([mpc_mdl '/Measurements'],'Ymin','300~0','Ymax','400~10')
set_param([mpc_mdl '/Coolant Temperature'],'Ymin','240','Ymax','360')
```

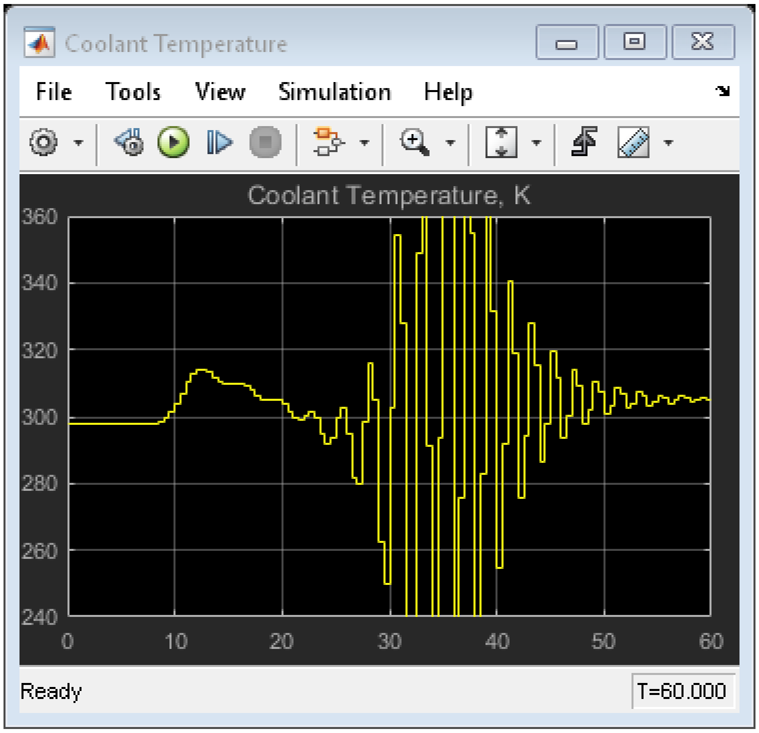

Measurements
File
Tools
View
Simulation
Help
Reactor Temperature, K
400
380
360
340
320
300
Concentration, kmol/m^3
10
8
6
4
2
0
0
2
4
6
8
10
Ready
T=10.000



Simulate 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
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 section, two additional MPC controllers are design at intermediate and final transition stages respectively.

Design MPC Controller for Intermediate Operating Condition

Create operating point specification.

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

Feed concentration is known.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Reactor concentration is known.

```
op.Outputs(2).y = 5.5;
op.Outputs(2).Known = true;
```

Find steady state operating condition.

```
[op_point,op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x; op_report.States(2).x];
y0 = [op_report.Outputs(1).y; op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u; op_report.Inputs(2).u; op_report.Inputs(3).u];
```

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

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

Operating point specifications were successfully met.
States:
----------
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           339      dx:      3.42e-08 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:           5.5      dx:     -2.87e-09 (0)

Inputs:
----------
(1.) mpc_cstr_plant/CAi
      u:            10
(2.) mpc_cstr_plant/Ti
      u:           298
(3.) mpc_cstr_plant/Tc
      u:           298    [-Inf Inf]

Outputs:
----------
(1.) mpc_cstr_plant/T
      y:           339    [-Inf Inf]
(2.) mpc_cstr_plant/CA
```

```
      y:           5.5    (5.5)
```

Obtain linear model at the initial condition.

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

Verify that the linear model is open-loop unstable at this condition.

```
eig(plant_intermediate)
```

```
ans =

    0.4941
   -0.8357
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```
plant_intermediate.InputGroup.UnmeasuredDisturbances = [1 2];
plant_intermediate.InputGroup.ManipulatedVariables = 3;
plant_intermediate.OutputGroup.Measured = [1 2];
plant_intermediate.InputName = {'CAi','Ti','Tc'};
plant_intermediate.OutputName = {'T','CA'};
```

Create MPC controller with default prediction and control horizons.

```
mpcobj_intermediate = mpc(plant_intermediate,Ts);
```

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

Set nominal values, scale factors and weights in the controller.

```
mpcobj_intermediate.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 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 eac
```

Design MPC Controller for Final Operating Condition

Create operating point specification.

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

Feed concentration is known.

```
op.Inputs(1).u = 10;
op.Inputs(1).Known = true;
```

Feed temperature is known.

```
op.Inputs(2).u = 298.15;
op.Inputs(2).Known = true;
```

Reactor concentration is known.

```
op.Outputs(2).y = 2;
op.Outputs(2).Known = true;
```

Find steady-state operating condition.

```
[op_point,op_report] = findop(plant_mdl,op);
% Obtain nominal values of x, y and u.
x0 = [op_report.States(1).x; op_report.States(2).x];
y0 = [op_report.Outputs(1).y; op_report.Outputs(2).y];
u0 = [op_report.Inputs(1).u; op_report.Inputs(2).u; op_report.Inputs(3).u];
```

```
 Operating point search report:
```

```
---------------------------------

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

Operating point specifications were successfully met.
States:
----------
(1.) mpc_cstr_plant/CSTR/Integrator
      x:           373      dx:      5.57e-11 (0)
(2.) mpc_cstr_plant/CSTR/Integrator1
      x:             2      dx:      -4.6e-12 (0)

Inputs:
----------
(1.) mpc_cstr_plant/CAi
      u:            10
(2.) mpc_cstr_plant/Ti
      u:           298
(3.) mpc_cstr_plant/Tc
      u:           305    [-Inf Inf]

Outputs:
----------
(1.) mpc_cstr_plant/T
      y:           373    [-Inf Inf]
(2.) mpc_cstr_plant/CA
      y:             2    (2)
```

Obtain linear model at the initial condition.

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

Verify that the linear model is again open-loop stable at this condition.

```
eig(plant_final)
```

```
ans =

  -1.1077 + 1.0901i
  -1.1077 - 1.0901i
```

Specify signal types used in MPC. Assume both reactor temperature and concentration are measurable.

```
plant_final.InputGroup.UnmeasuredDisturbances = [1 2];
plant_final.InputGroup.ManipulatedVariables = 3;
plant_final.OutputGroup.Measured = [1 2];
plant_final.InputName = {'CAi','Ti','Tc'};
plant_final.OutputName = {'T','CA'};
```

Create MPC controller with default prediction and control horizons.

```
mpcobj_final = mpc(plant_final,Ts);
```

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

Set nominal values, scale factors and weights in the controller.

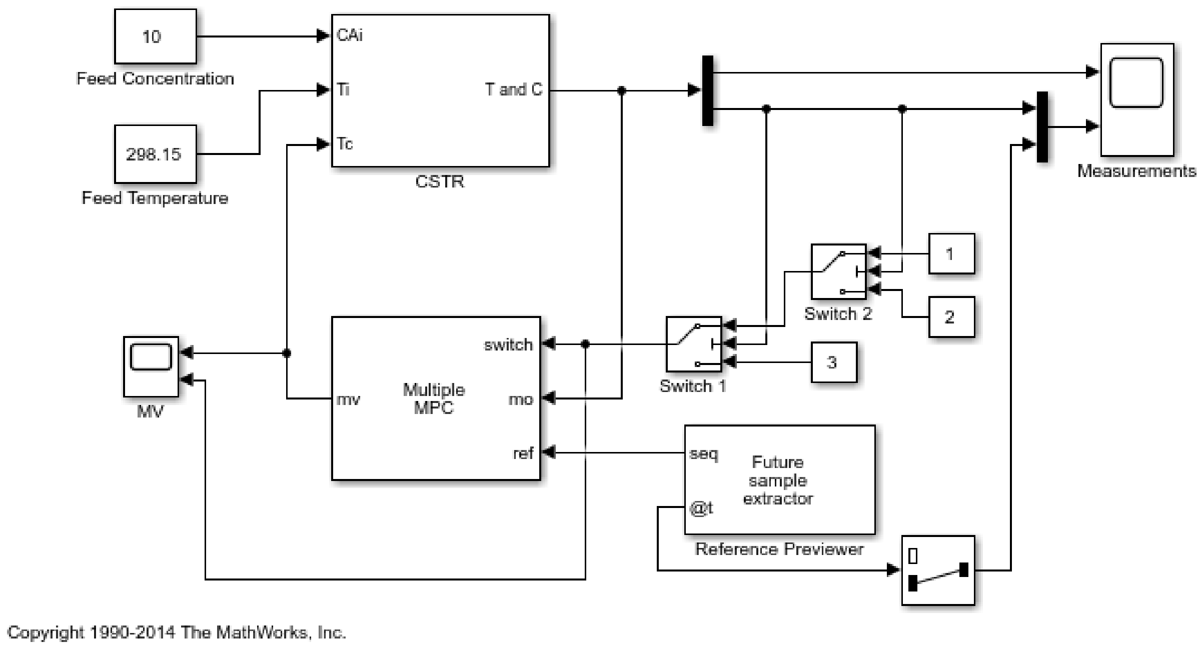
```
mpcobj_final.Model.Nominal = struct('X',x0,'U',[0;0;u0(3)],'Y',y0,'DX',[0 0]);
Uscale = [10;30;50];
Yscale = [50;10];
mpcobj_final.DV(1).ScaleFactor = Uscale(1);
mpcobj_final.DV(2).ScaleFactor = Uscale(2);
mpcobj_final.MV.ScaleFactor = Uscale(3);
mpcobj_final.OV(1).ScaleFactor = Yscale(1);
mpcobj_final.OV(2).ScaleFactor = 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 eac
```

Control the CSTR Plant With the Multiple MPC Controllers Block

The following model uses the Multiple MPC Controllers block to implement three MPC controllers across the operating range.

```
mmpc_mdl = 'mpc_cstr_multiple';
open_system(mmpc_mdl);
```



Note that it has been configured to use the three controllers in a sequence: mpcobj, mpcobj_intermediate and mpcobj_final.

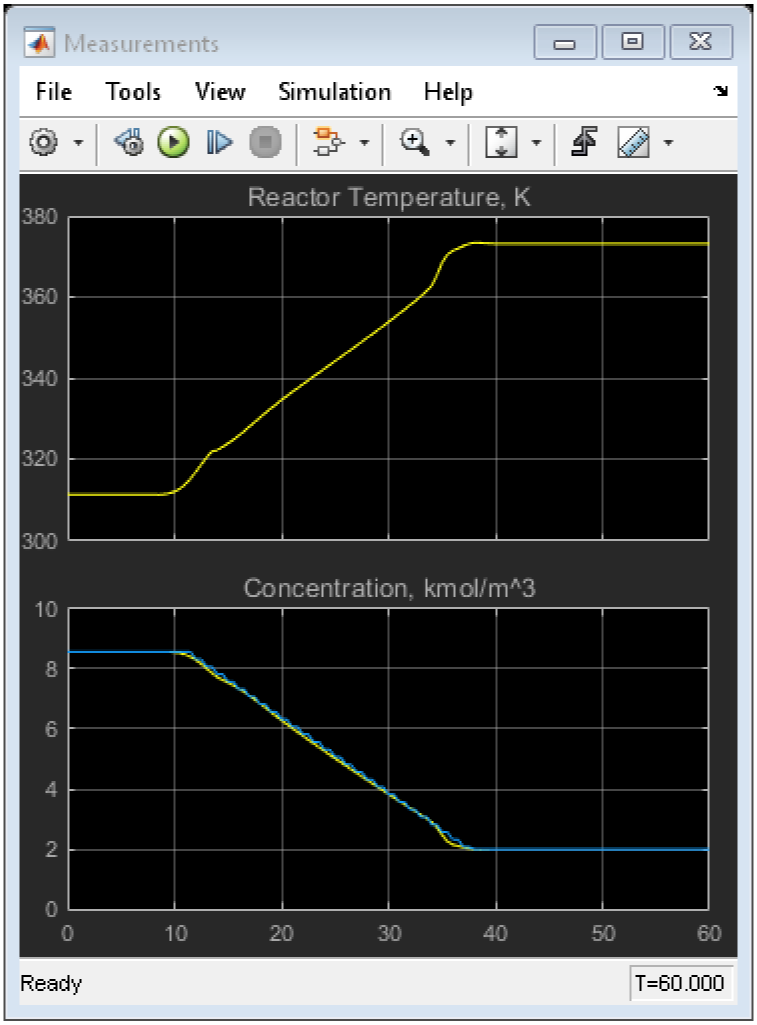
```
open_system([mmpc_mdl '/Multiple MPC Controllers']);
```

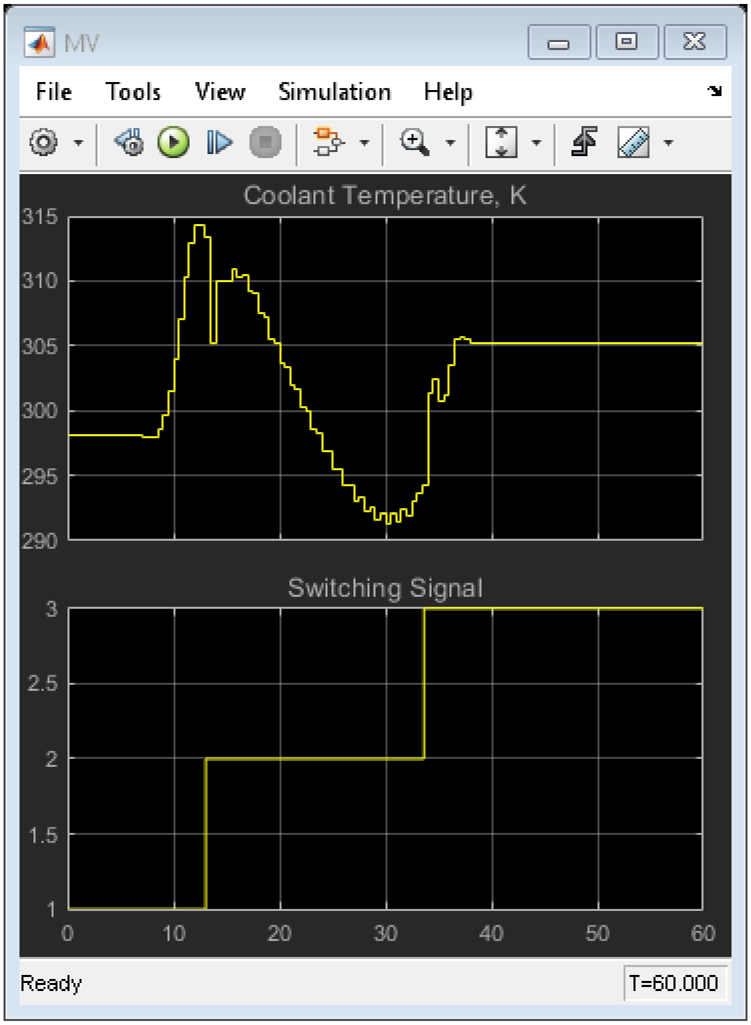
Note also that the two switches specify when to switch from one controller to another. The rules are: 1. If CSTR concentration >= 8, use "mpcobj" 2. If 3 <= CSTR concentration < 8, use "mpcobj_intermediate" 3. If CSTR concentration < 3, use "mpcobj_final"

Simulate with the Multiple MPC Controllers block

```
open_system([mmpc_mdl '/Measurements']);
open_system([mmpc_mdl '/MV']);
sim(mmpc_mdl)
```

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


Measurements
File
Tools
View
Simulation
Help
Reactor Temperature, K
380
360
340
320
300
Concentration, kmol/m^3
10
8
6
4
2
0
0
10
20
30
40
50
60
Ready
T=60.000



The transition is now well controlled. The major improvement is in the transition through the open-loop unstable region. The plot of the switching signal shows when controller transitions occur. The MV character changes at these times because of the change in dynamic characteristics introduced by the new prediction model.

```
bdclose(plant_mdl)
bdclose(mpc_mdl)
bdclose(mmpc_mdl)
```

See Also

More About

- “Gain-Scheduled MPC” on page 7-2
- “Schedule Controllers at Multiple Operating Points” on page 7-5
- “Gain Scheduled Implicit and Explicit MPC Control of Mass-Spring System” on page 7-50

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+M2 move together. We assume that only M1 position and a contact sensor are available for feedback. The latter is used to trigger switching the MPC controllers. Note that position and velocity of mass M2 are not controllable.

```
                         /-----\     k1                ||
                  F <--- | M1  |----/\/\/\-------------[|| wall
    ||                   | |---/                        ||
    ||     k2            \-/      /----\                ||
wall||]--/\/\/\-------------------| M2 |                ||
    ||                            \----/                ||
    ||                                                  ||
----yeq2------------------ y1 ------ y2 ----------------yeq1----> y axis
```

The model is a simplified version of the model proposed in the following reference:

A. Bemporad, S. Di Cairano, I. V. Kolmanovsky, and D. Hrovat, "Hybrid modeling and control of a multibody magnetic actuator for automotive applications," in Proc. 46th IEEE® Conf. on Decision and Control, New Orleans, LA, 2007.

Model Parameters

```
M1 = 1;       % mass
M2 = 5;       % mass
k1 = 1;       % spring constant
k2 = 0.1;     % spring constant
b1 = 0.3;     % friction coefficient
b2 = 0.8;     % friction coefficient
```

```
yeq1 = 10;    % wall mount position
yeq2 = -10;   % wall mount position
```

State Space Models

states: position and velocity of mass M1; manipulated variable: pull force F measured disturbance: a constant value of 1 which provides calibrates spring force to the right value measured output: position of mass M1

State-space model of M1 when masses are not in contact.

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

```
-->Assuming unspecified input signals are manipulated variables.
```

State-space model when the two masses are in contact.

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

```
-->Assuming unspecified input signals are manipulated variables.
```

Design MPC Controllers

Common parameters

```
Ts = 0.2;     % sampling time
p = 20;       % prediction horizon
m = 1;        % control horizon
```

Design 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 defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify constraints on the manipulated variable.

```
MPC1.MV = struct('Min',0,'Max',30,'RateMin',-10,'RateMax',10);
```

Design second MPC controller for the case when mass 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 defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify constraints on the manipulated variable.

```
MPC2.MV = struct('Min',0,'Max',30,'RateMin',-10,'RateMax',10);
```

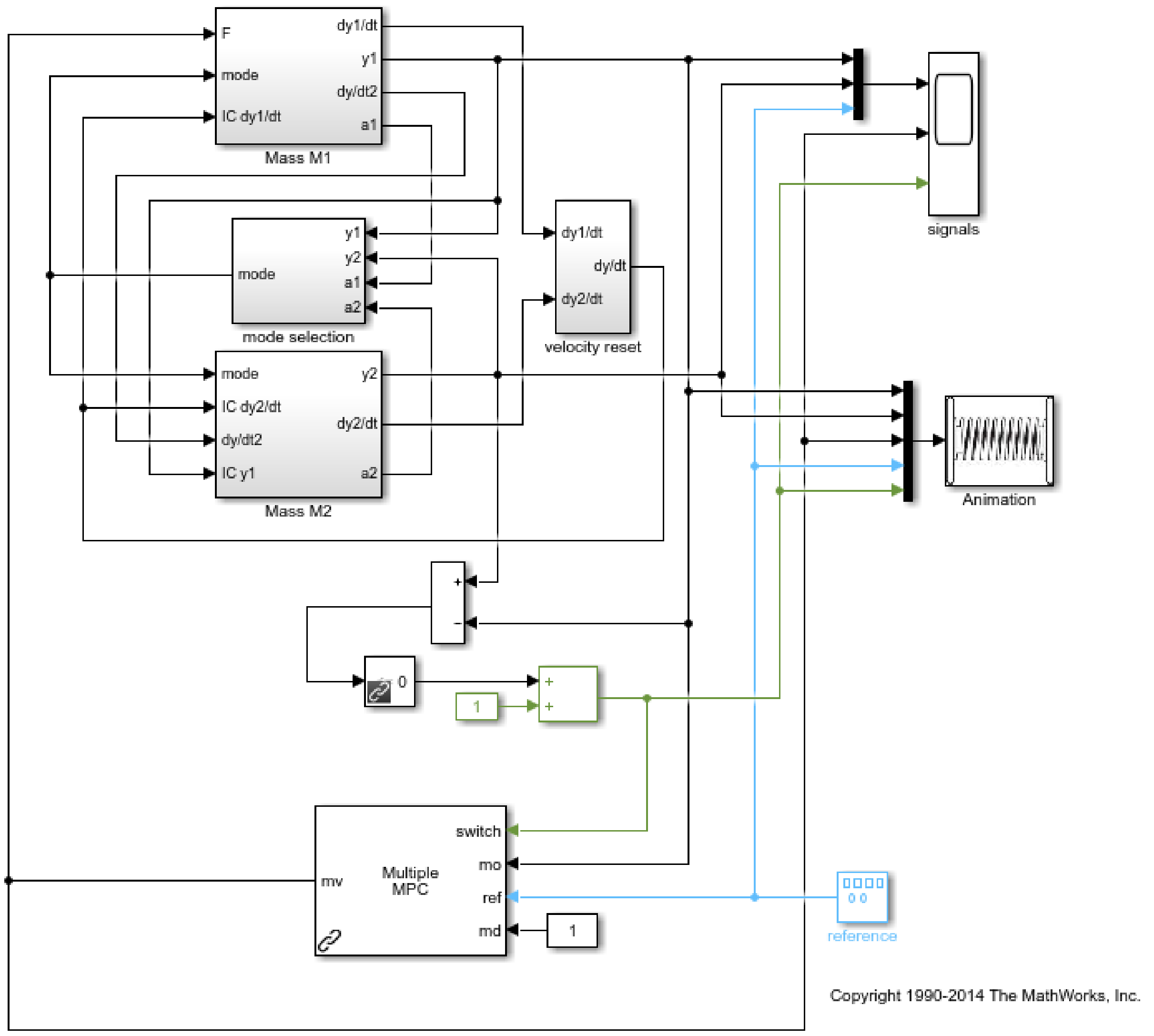
Simulate Gain Scheduled MPC in Simulink®

To run this example, Simulink® is required.

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

Simulate gain scheduled MPC control with Multiple MPC Controllers block.

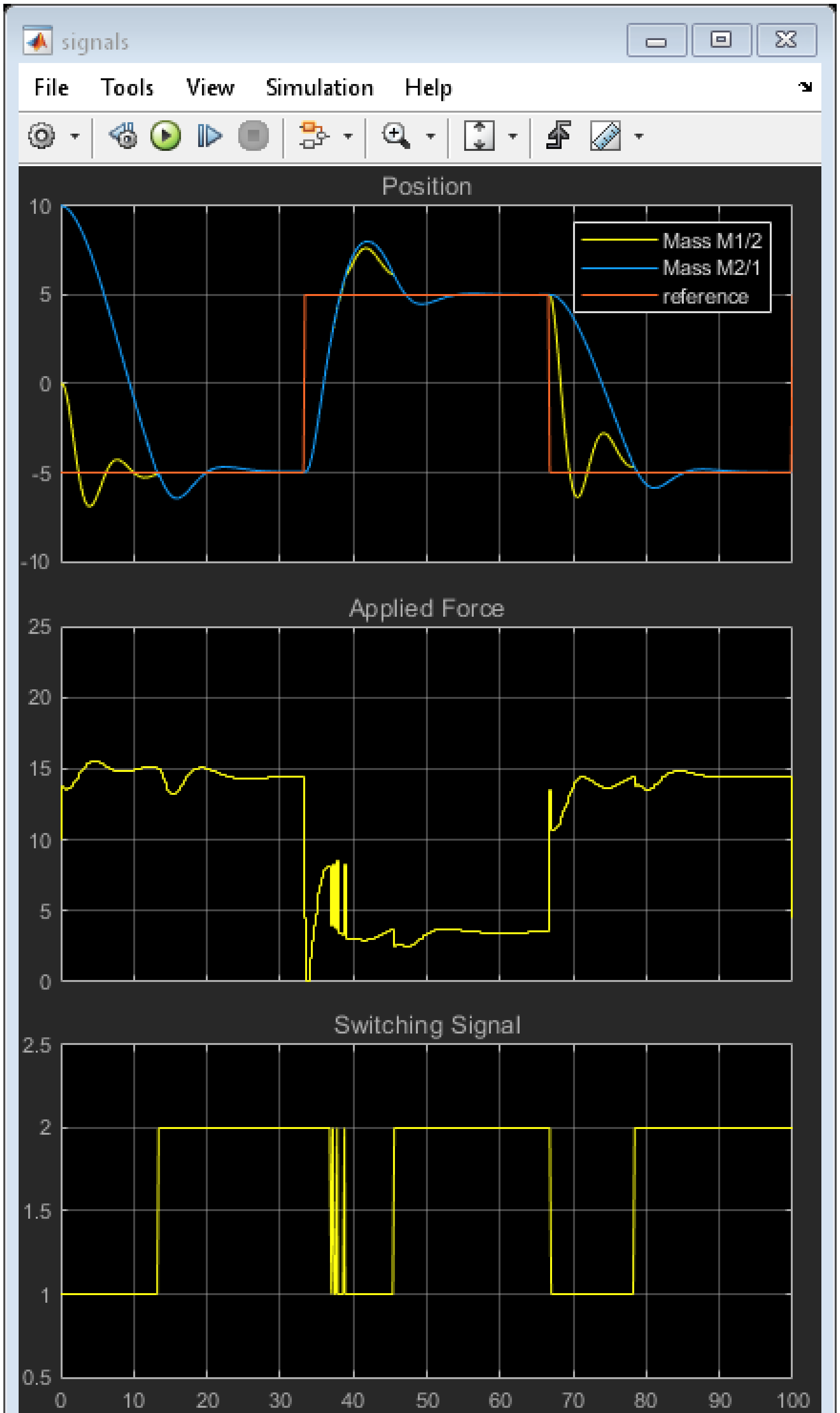
```
y1initial = 0;      % Initial position of M1
y2initial = 10;     % Initial position of M2
mdl = 'mpc_switching';
open_system(mdl);
if exist('animationmpc_switchoff','var') && animationmpc_switchoff
    close_system([mdl '/Animation']);
    clear animationmpc_switchoff
end
```

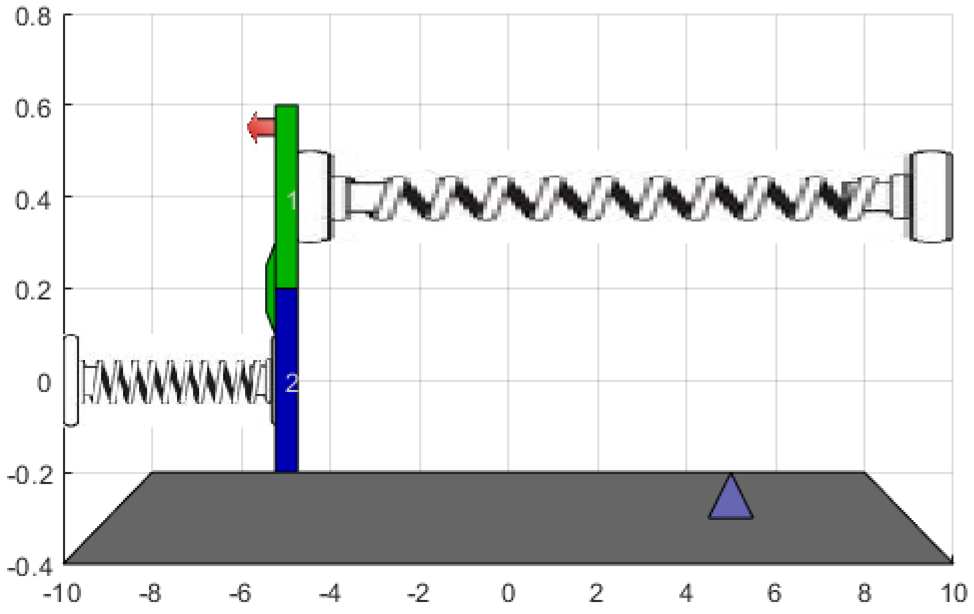


```
disp('Start simulation by switching control between MPC1 and MPC2 ...');
disp('Control performance is satisfactory.');
open_system([mdl '/signals']);
sim(mdl);
```

```
MPC1saved = MPC1;
MPC2saved = MPC2;
```

```
Start simulation by switching control between MPC1 and MPC2 ...
Control performance is satisfactory.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
10
5
0
-5
-10
25
20
15
10
5
0
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100

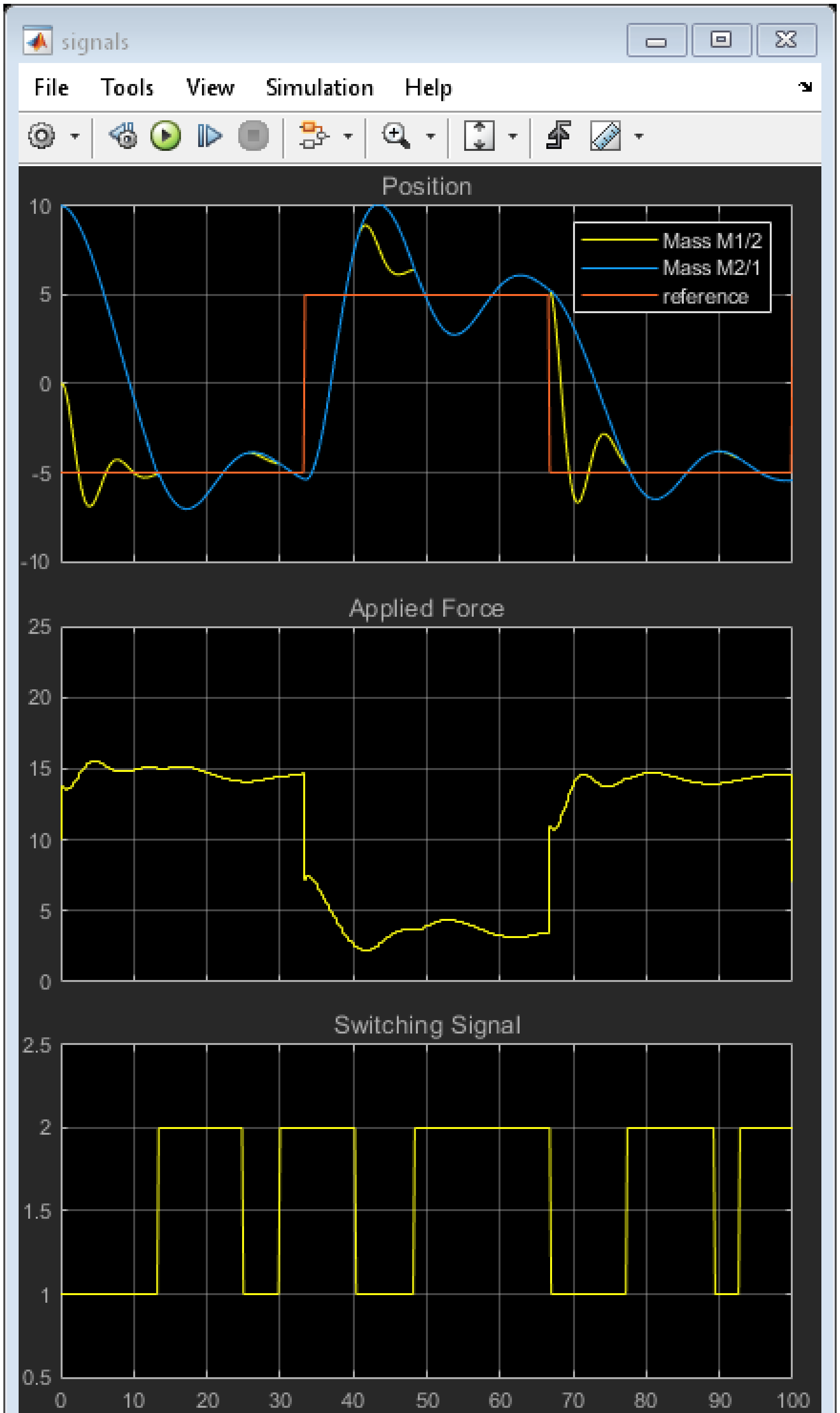


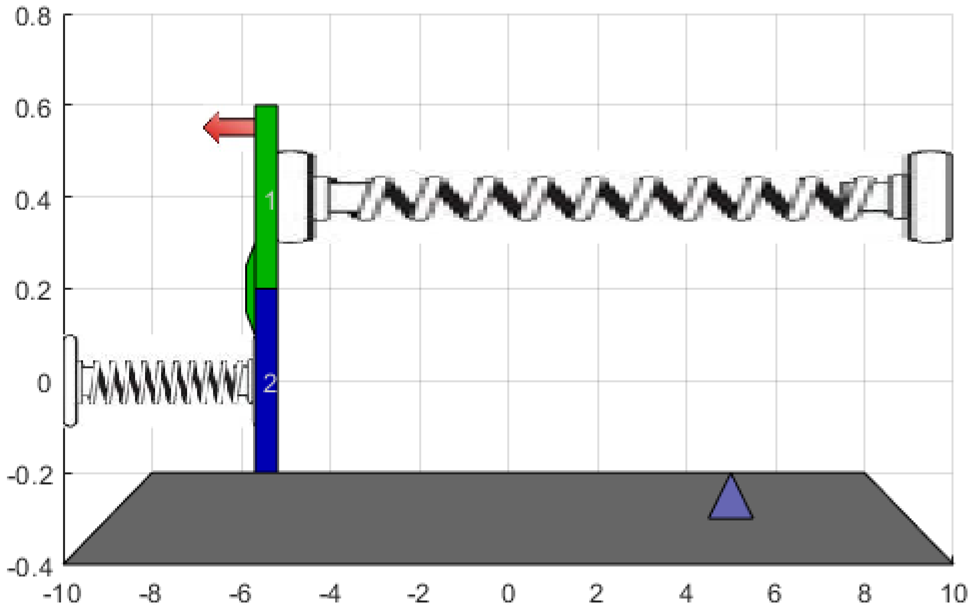
Use of two controllers provides good performance under all conditions.

Repeat Simulation Using MPC1 Only (Assumes Masses Never in Contact)

```
disp('Now repeat simulation by using only MPC1 ...');
disp('When two masses stick together, control performance deteriorates.');
MPC1 = MPC1saved;
MPC2 = MPC1saved;
sim(mdl);
```

```
Now repeat simulation by using only MPC1 ...
When two masses stick together, control performance deteriorates.
```


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
10
5
0
-5
-10
25
20
15
10
5
0
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100

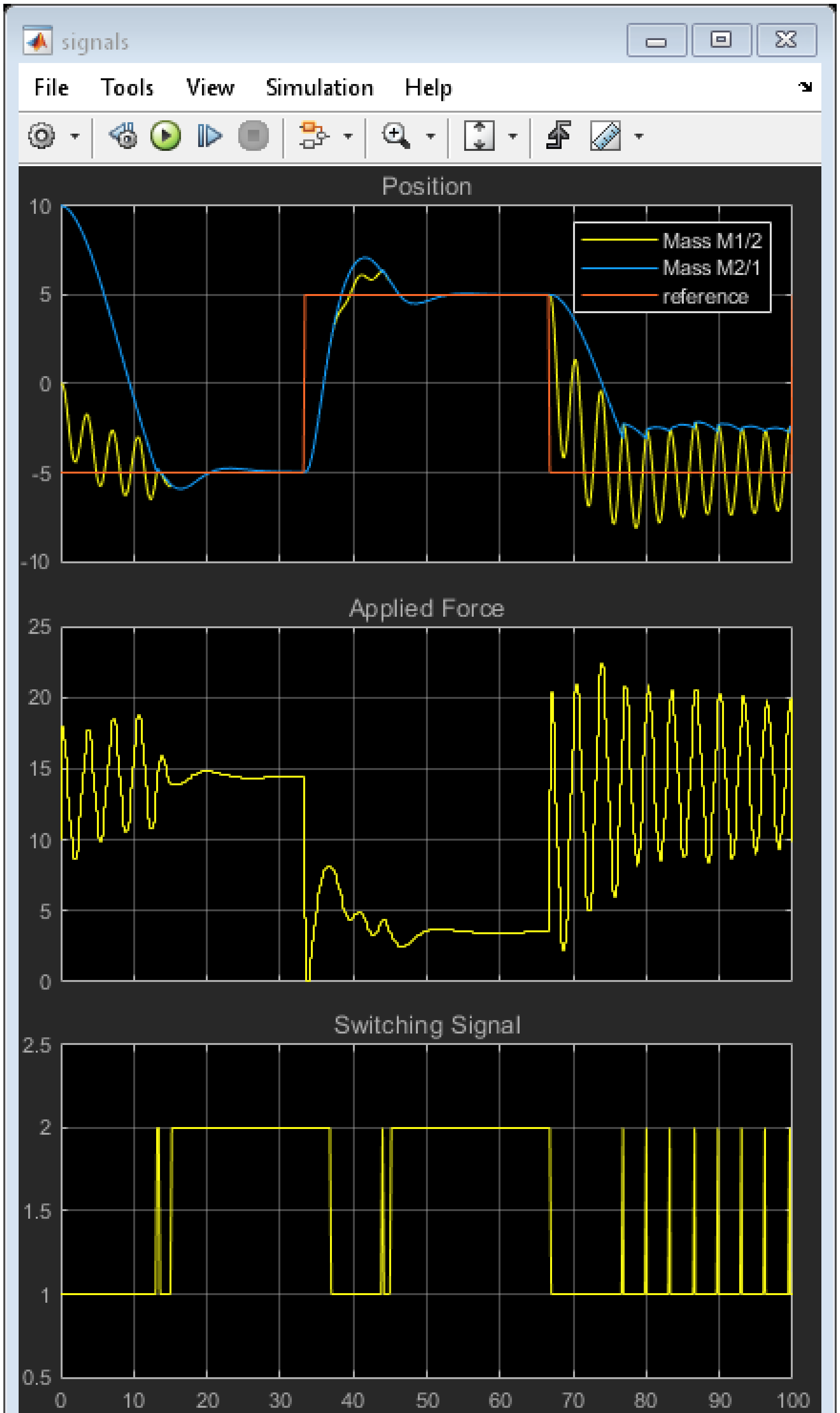


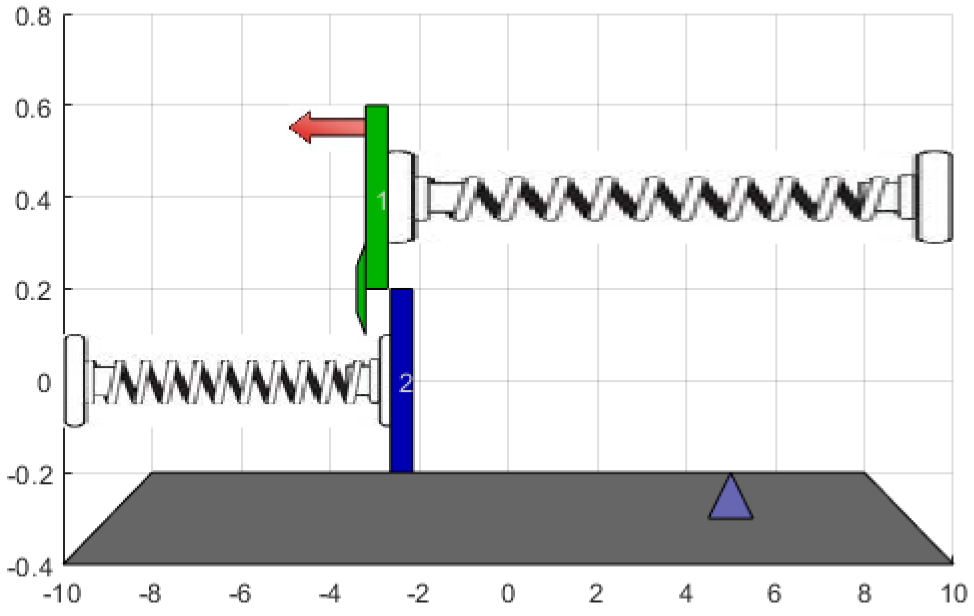
In this case, performance degrades whenever the two masses join.

Repeat Simulation Using MPC2 Only (Assumes Masses Always in Contact)

```
disp('Now repeat simulation by using only MPC2 ...');
disp('When two masses are detached, control performance deteriorates.');
MPC1 = MPC2saved;
MPC2 = MPC2saved;
sim(mdl);
```

```
Now repeat simulation by using only MPC2 ...
When two masses are detached, control performance deteriorates.
```


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal
10
5
0
-5
-10
25
20
15
10
5
0
2.5
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100



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

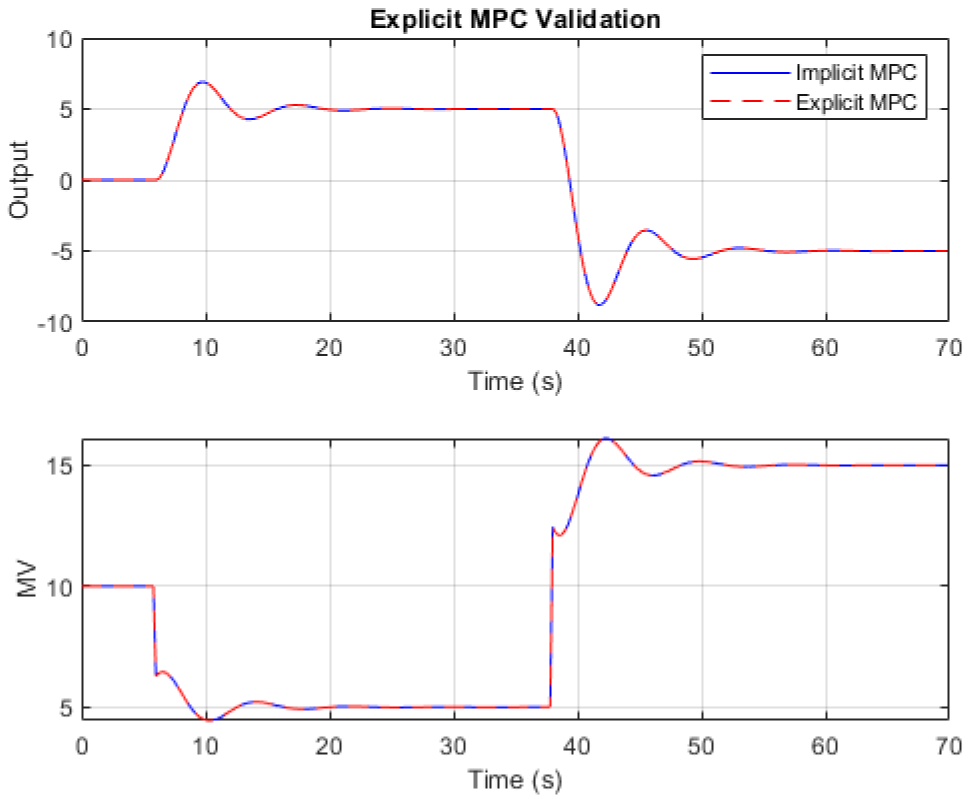
It is good practice to validate the performance of each explicit MPC controller before implementing gain-scheduled explicit MPC. For example, to compare the performance of MPC1 and expMPC1, simulate the closed-loop response of each controller using sim.

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

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)')
```



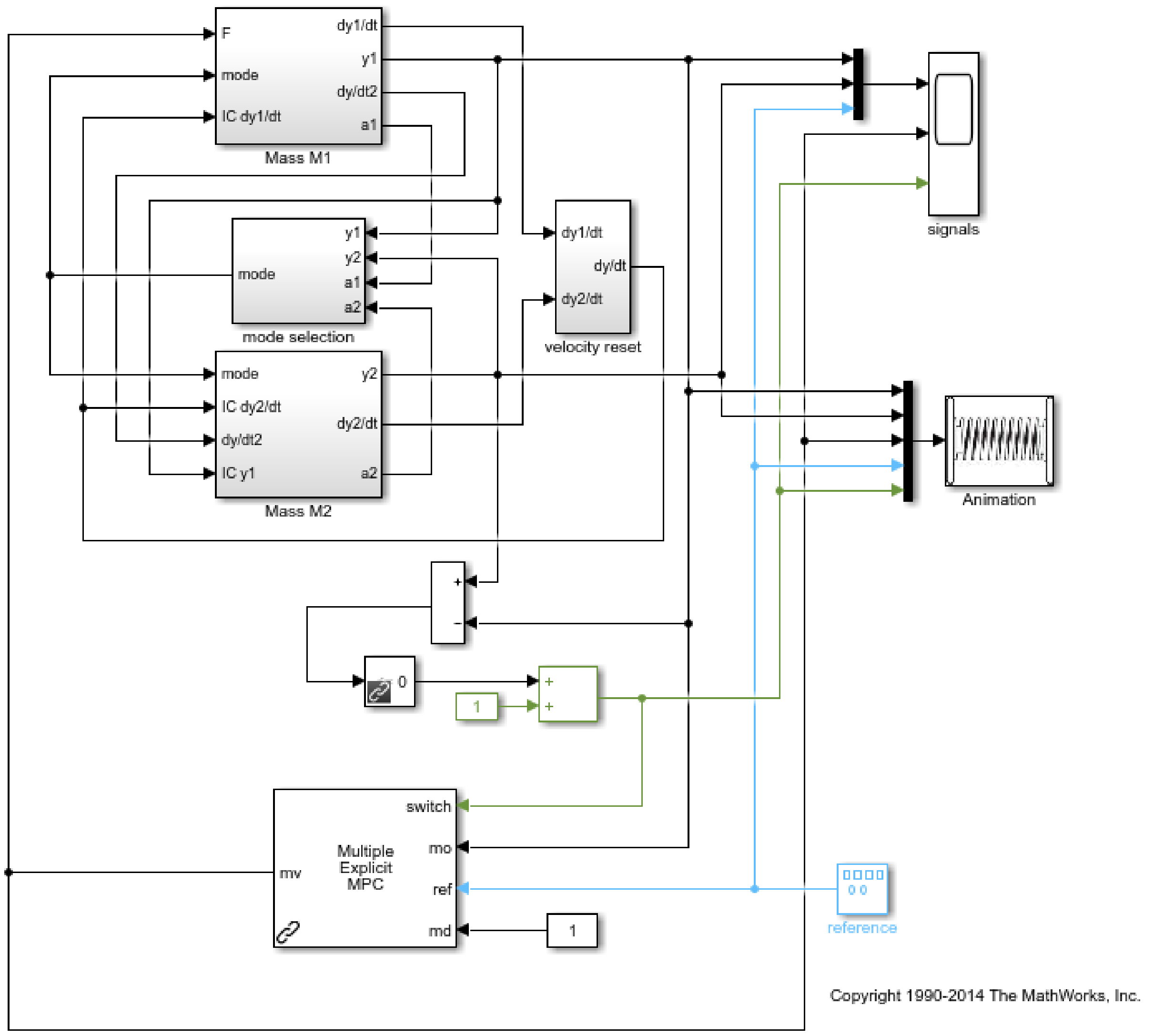
The closed-loop responses and manipulated variable sequences of the implicit and explicit controllers match. Similarly, you can validate the performance of expMPC2 against that of MPC2.

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

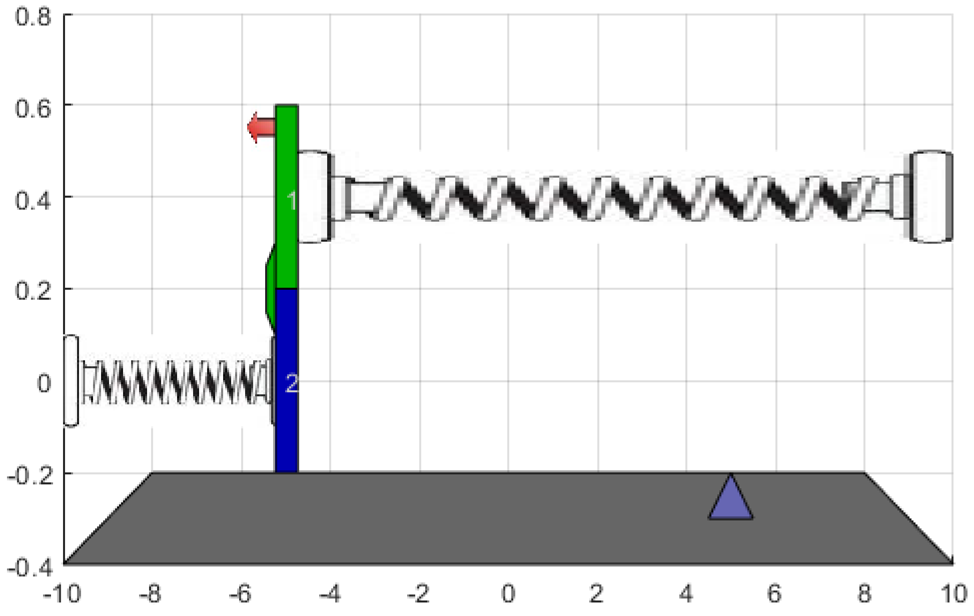
To implement gain-scheduled explicit MPC control, replace the Multiple MPC Controllers block with the Multiple Explicit MPC Controllers block.

```
expModel = 'mpc_switching_explicit';
open_system(expModel);
```



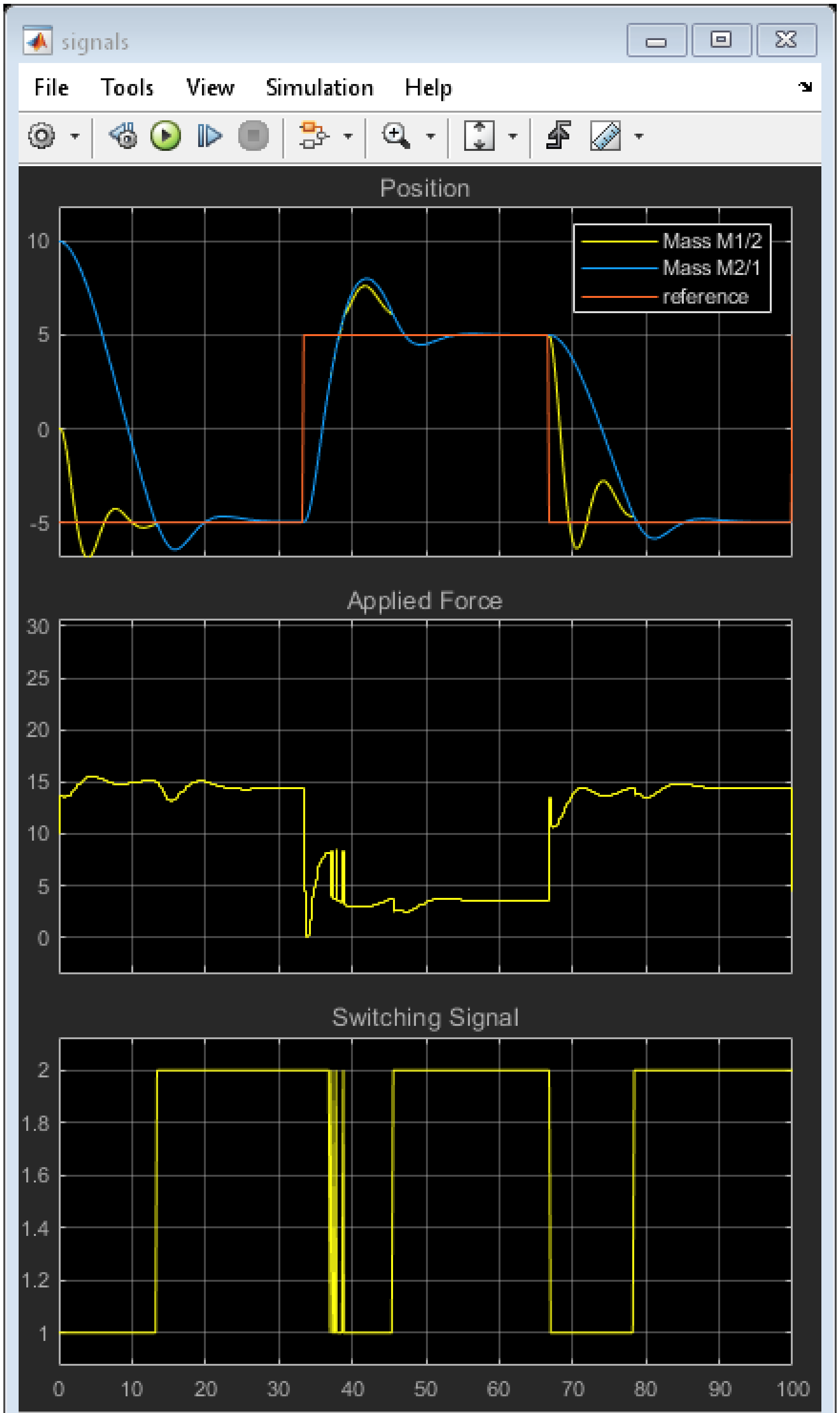
Run the simulation.

```
sim(expModel);
```



To view the simulation results, open the signals scope.

```
open_system([expModel '/signals']);
```


signals
File
Tools
View
Simulation
Help
Position
Mass M1/2
Mass M2/1
reference
Applied Force
Switching Signal

The gain-scheduled explicit MPC controllers provide the same performance as the gain-scheduled implicit MPC controllers.

```
bdclose(expModel);
close(findobj('Tag','mpc_switching_demo'));
```

See Also

More About

- “Gain-Scheduled MPC” on page 7-2
- “Schedule Controllers at Multiple Operating Points” on page 7-5
- “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor” on page 7-26

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.

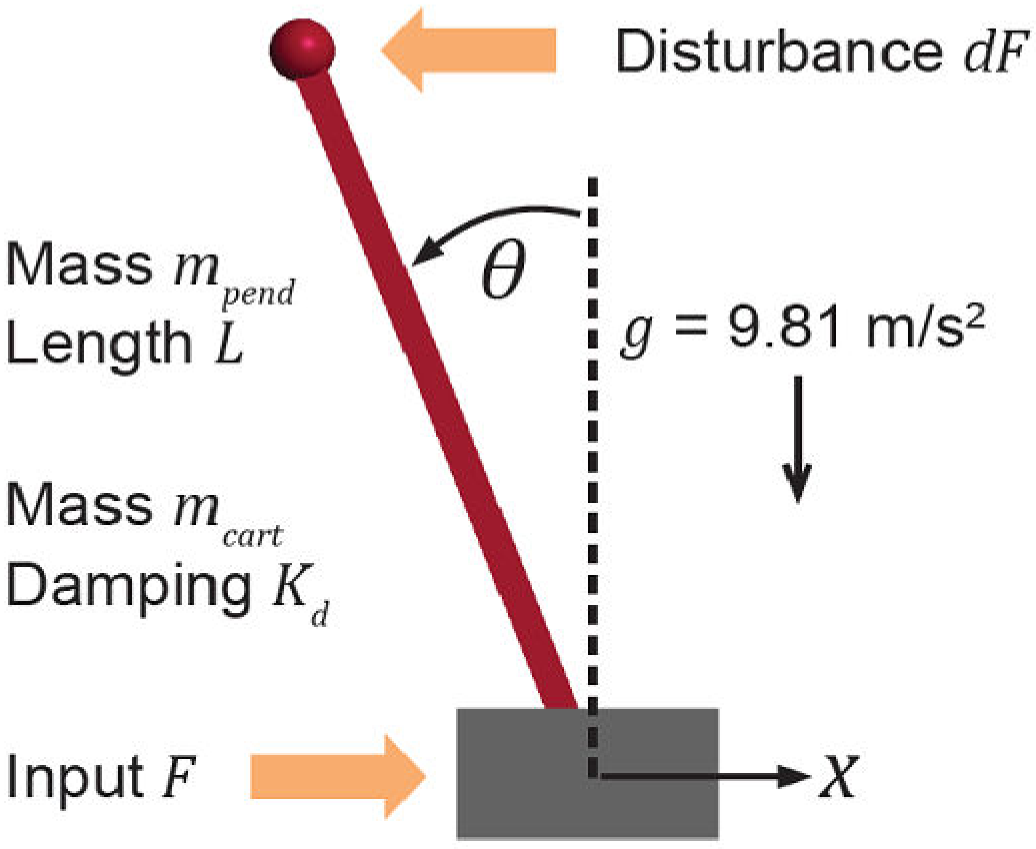
```
if ~mpcchecktoolboxinstalled('slcontrol')
    disp('Simulink Control Design is required to run this example.')
    return
end
```

Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc_featured','main'));
```

Pendulum/Cart Assembly

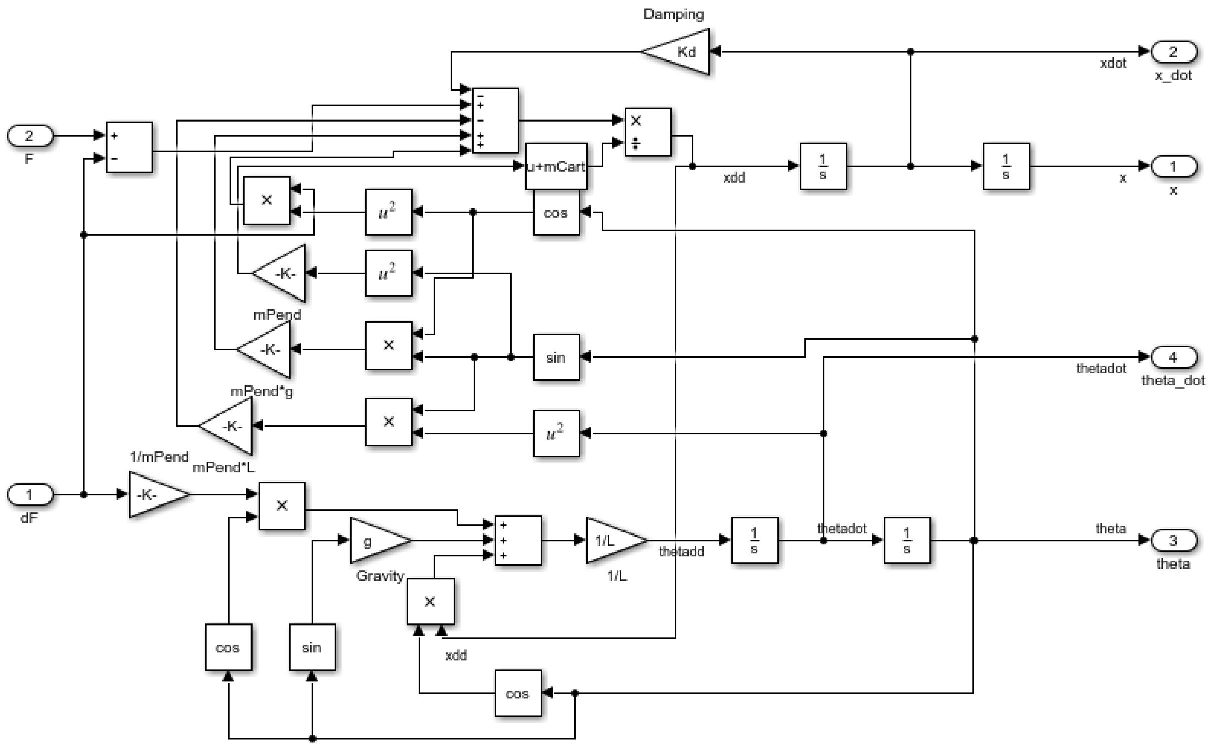
The plant for this example is the following cart/pendulum assembly, where *x* is the cart position and *theta* is the pendulum angle.



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”, 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.

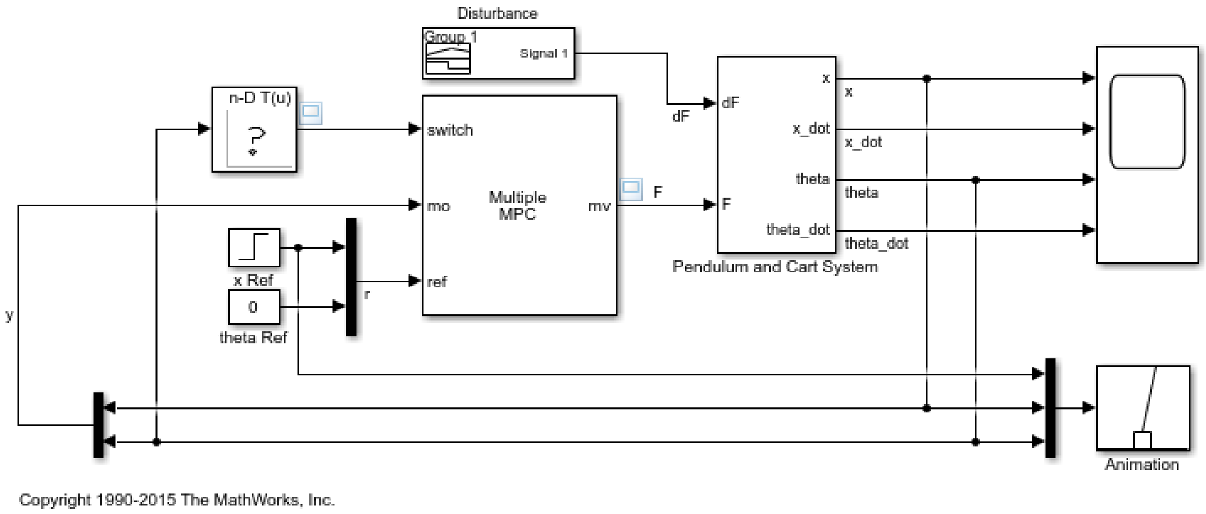
Control Structure

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 point using these specifications.

```
options = findopOptions('DisplayReport',false);
[op,opresult] = findop(mdlPlant,opspec,options);
```

Obtain the linear plant model at the specified operating points.

```
plants = linearize(mdlPlant,op,io);

bdclose(mdlPlant)
```

Multiple MPC Designs

At each operating point, design an MPC controller with the corresponding linear plant model.

```
status = mpcverbosity('off');
for ct=1:length(angles)
```

Get a single plant model.

```
    plant = plants(:,:,ct);
    plant.InputName = {'dF'; 'F'};
    plant.OutputName = {'x'; 'theta'};
```

The plant has two inputs, *dF* and *F*, and two outputs, *x* and *theta*. In this example, *dF* is specified as an unmeasured disturbance used by the MPC controller for prediction. Set the plant signal types.

```
    plant = setmpcsignals(plant,'ud',1,'mv',2);
```

To control an unstable plant, the controller sample time cannot be too large (poor disturbance rejection) or too small (excessive computation load). Similarly, the prediction horizon cannot be too long (the plant unstable mode would dominate) or too short (constraint violations would be unforeseen). Use the following parameters for this example:

```
    Ts = 0.01;
    PredictionHorizon = 50;
    ControlHorizon = 5;
    mpcobj = mpc(plant,Ts,PredictionHorizon,ControlHorizon);
```

Specify nominal input and output values based on the operating point.

```
    mpcobj.Model.Nominal.Y = [0;opresult(ct).States(3).x];
    mpcobj.Model.Nominal.X = [0;0;opresult(ct).States(3).x;0];
    mpcobj.Model.Nominal.DX = [0;opresult(ct).States(2).dx;0;opresult(ct).States(4).dx
```

There is a limitation on how much force we can apply to the cart, which is specified as hard constraints on manipulated variable *F*.

```
    mpcobj.MV.Min = -200;
    mpcobj.MV.Max = 200;
```

It is good practice to scale plant inputs and outputs before designing weights. In this case, since the range of the manipulated variable is greater than the range of the plant outputs by two orders of magnitude, scale the MV input by 100.

```
    mpcobj.MV.ScaleFactor = 100;
```

To improve controller robustness, increase the weight on the MV rate of change from `0.1` to `1`.

```
    mpcobj.Weights.MVRate = 1;
```

To achieve balanced performance, adjust the weights on the plant outputs. The first weight is associated with cart position *x* and the second weight is associated with angle *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);
```

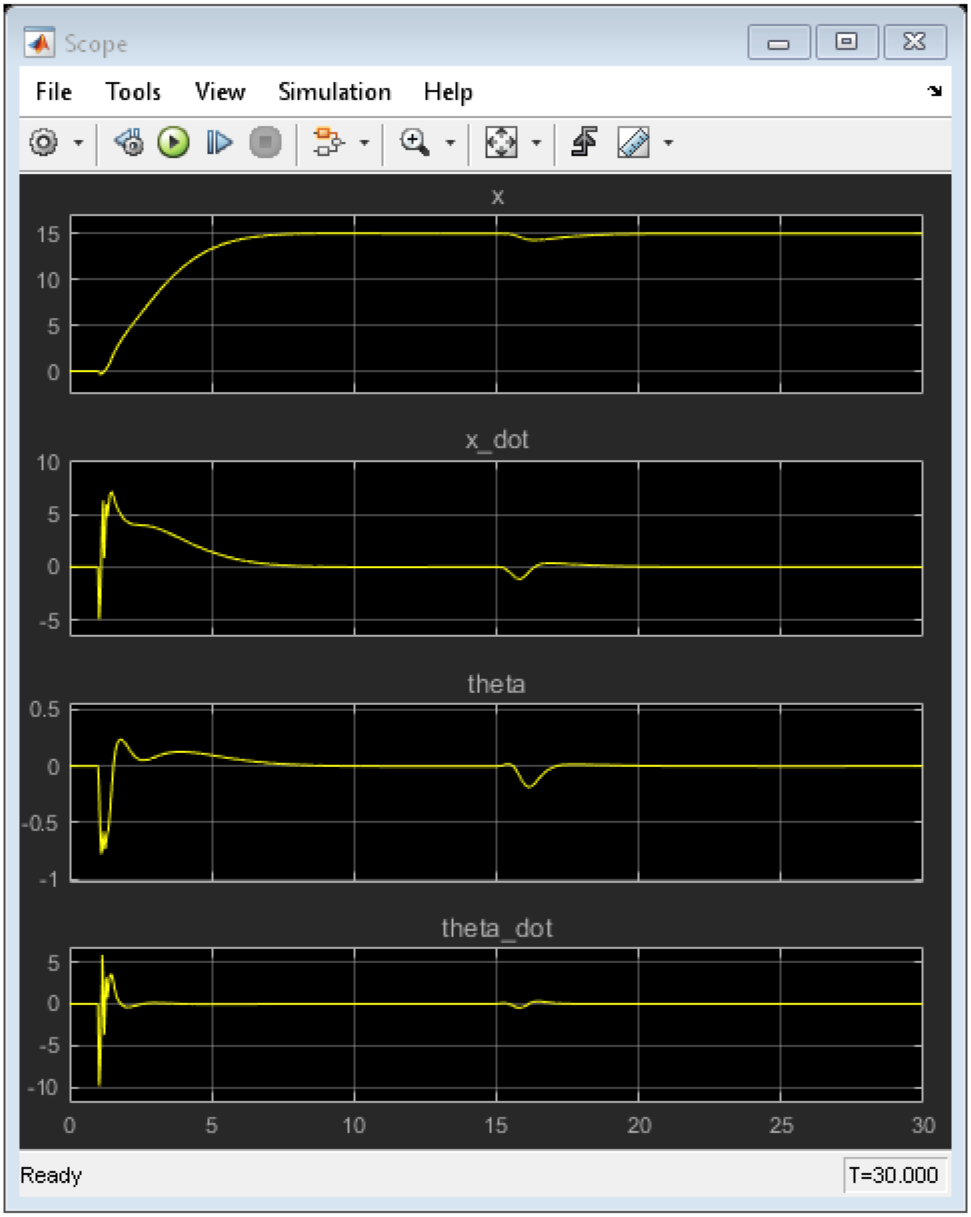
Closed-Loop Simulation

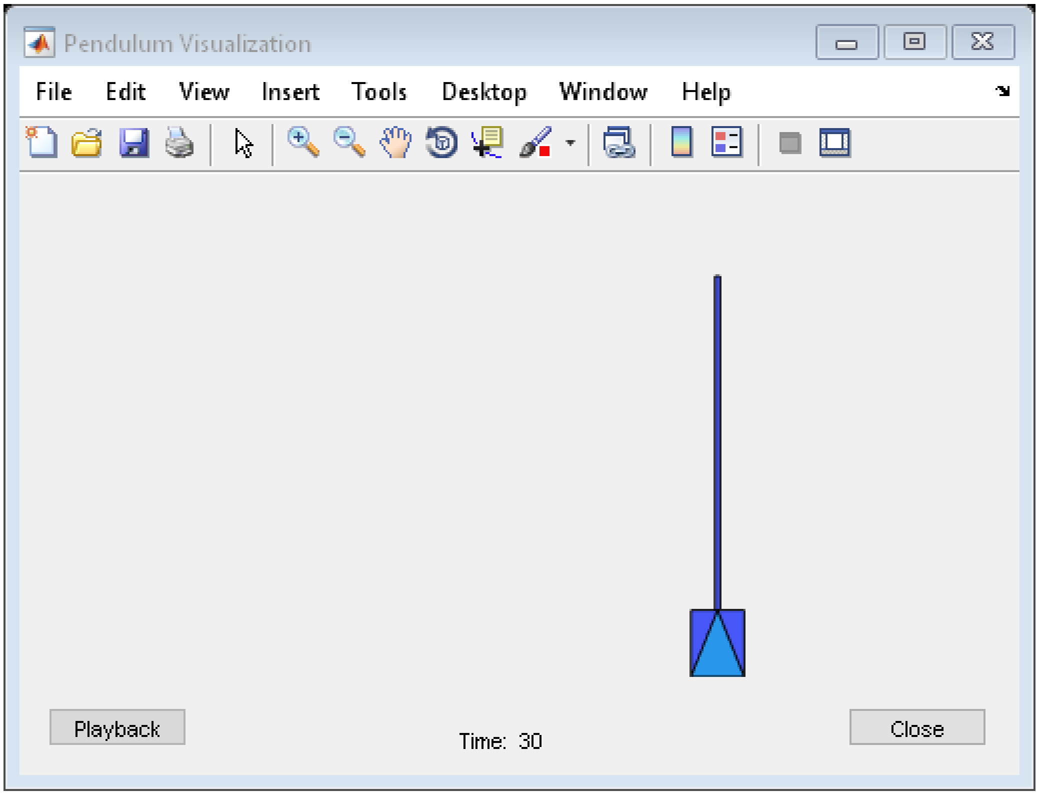
Validate the MPC design with a closed-loop simulation in Simulink.

```
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 eac
```

```
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
-->Converting model to discrete time.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```


Scope
File
Tools
View
Simulation
Help
x
x_dot
theta
theta_dot
Ready
T=30.000



In the nonlinear simulation, all the control objectives are successfully achieved.

Remove example file folder from MATLAB path, and close Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc_featured','main'));
bdclose(mdlMPC)
```

See Also

More About

- “Gain-Scheduled MPC” on page 7-2
- “Control of an Inverted Pendulum on a Cart” on page 4-104

- “Explicit MPC Control of an Inverted Pendulum on a Cart” on page 6-42

Reference for MPC Designer App

This chapter is the reference manual for the Model Predictive Control Toolbox **MPC Designer** app.

Generate MATLAB Code from MPC Designer

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.

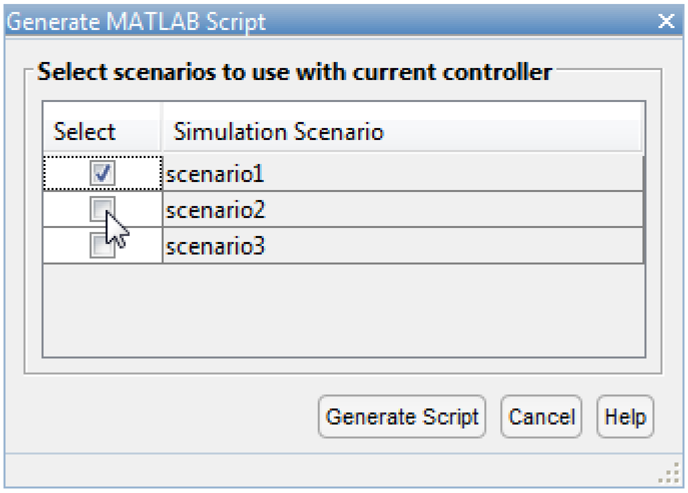
To create a MATLAB script:

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.



Select	Simulation Scenario
✓	scenario1
	scenario2
	scenario3

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 8-4

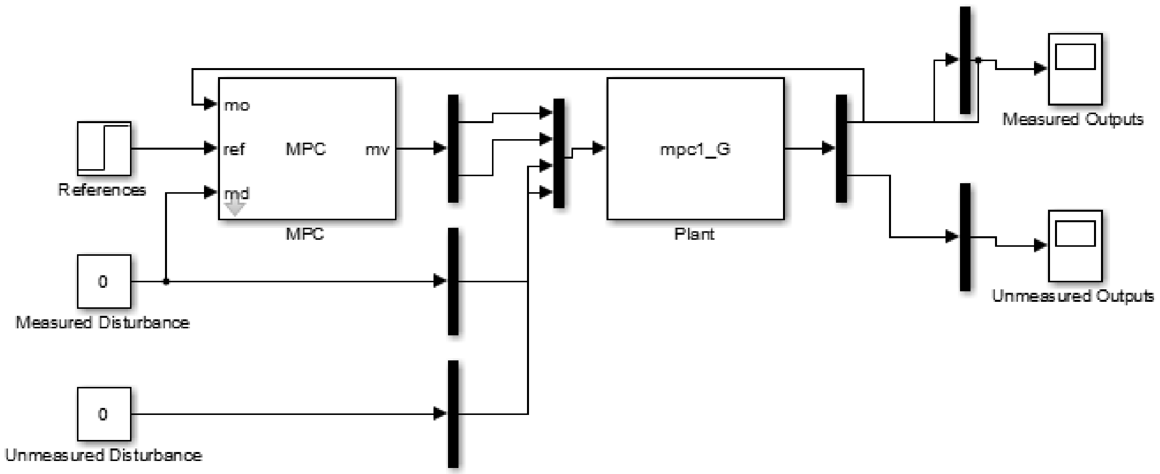
Generate Simulink Model from MPC Designer

This topic shows how to generate a Simulink model that uses the current model predictive controller to control its internal plant model.

To create a Simulink model:

1. In the **MPC Designer** app, interactively design and tune your model predictive controller.
2. On the **Tuning** tab, in the **Analysis** section, click the **Export Controller** arrow ▼.

   Alternatively, 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 9-2.

See Also

MPC Controller | **MPC Designer**

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 9-2
- “Design MPC Controller in Simulink”
- “Generate MATLAB Code from MPC Designer” on page 8-2

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.

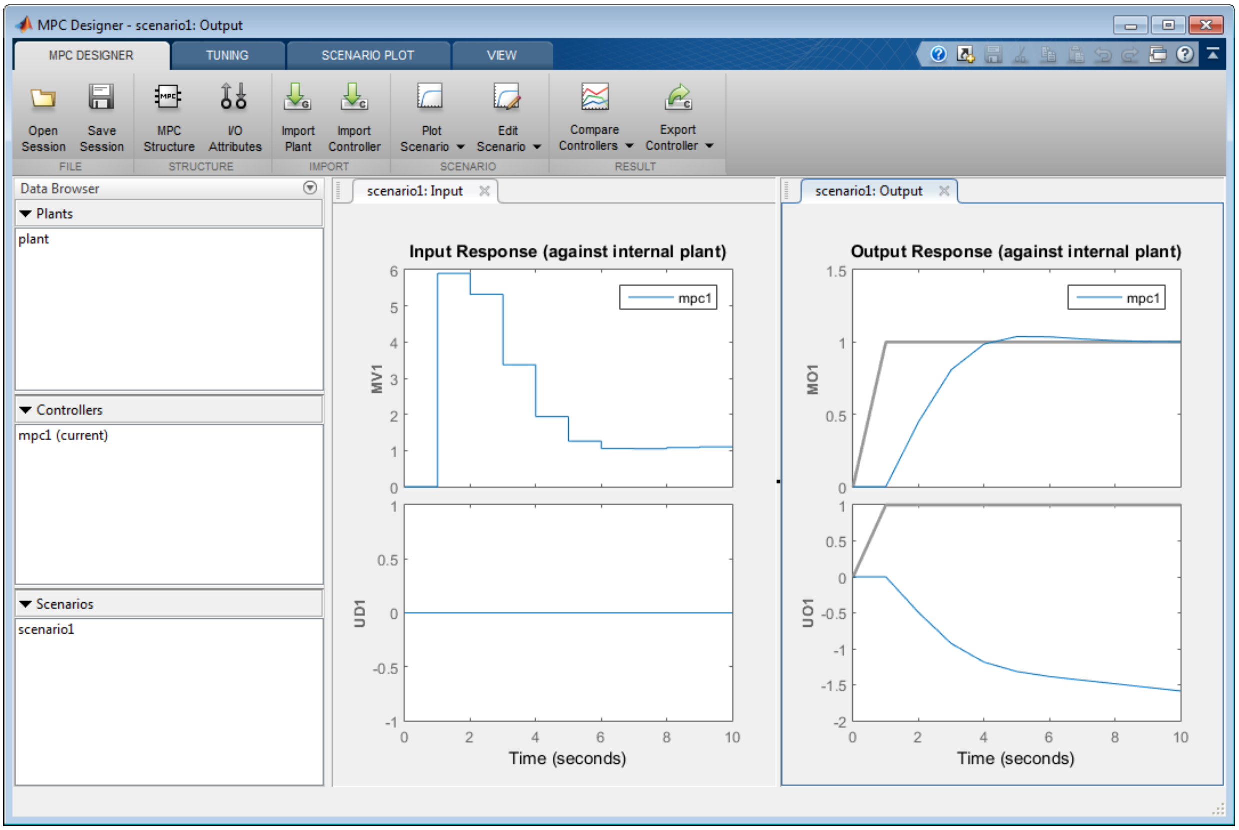
Define Plant Model

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

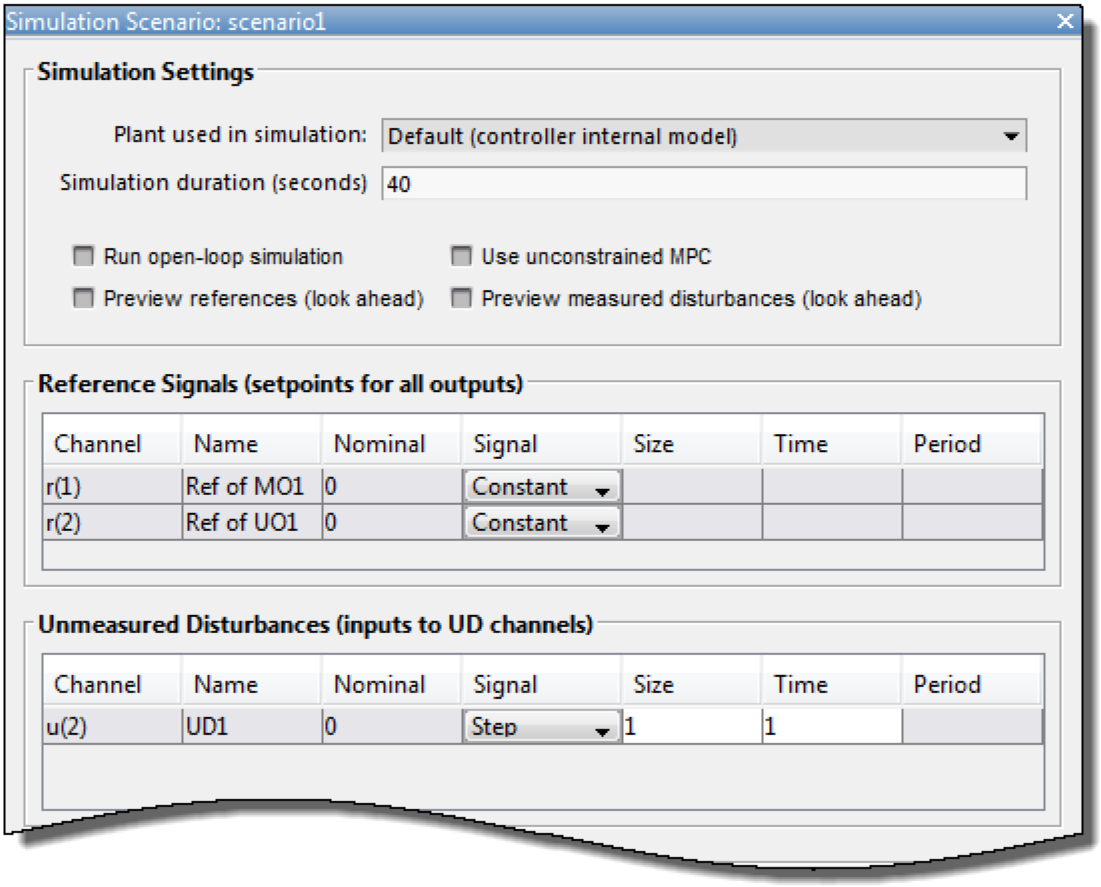
Configure a disturbance rejection simulation scenario.

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.



Simulation Scenario: scenario1

Simulation Settings

Plant used in simulation: Default (controller internal model)

Simulation duration (seconds) 40

☐ 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 MO1	0	Constant			
r(2)	Ref of UO1	0	Constant			

Unmeasured Disturbances (inputs to UD channels)

Channel	Name	Nominal	Signal	Size	Time	Period
u(2)	UD1	0	Step	1	1	

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

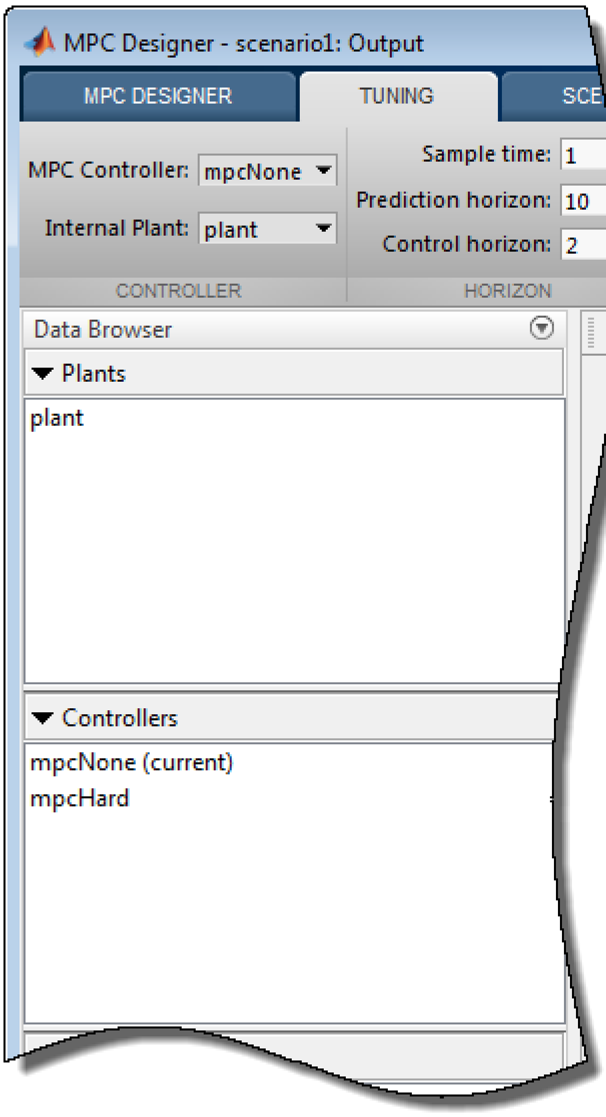
Suppose that the control specifications indicate that such an increase in the unmeasured disturbance is undesirable. To limit the effect of the unmeasured disturbance, create a controller with a hard output constraint.

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.


MPC Designer - scenario1: Output

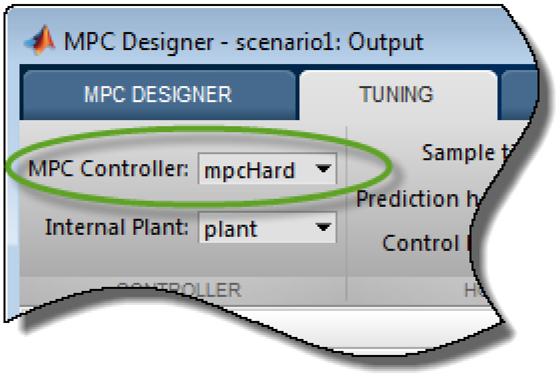
MPC Controller: mpcNone
Internal Plant: plant
Sample time: 1
Prediction horizon: 10
Control horizon: 2

plant

mpcNone (current)
mpcHard

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.

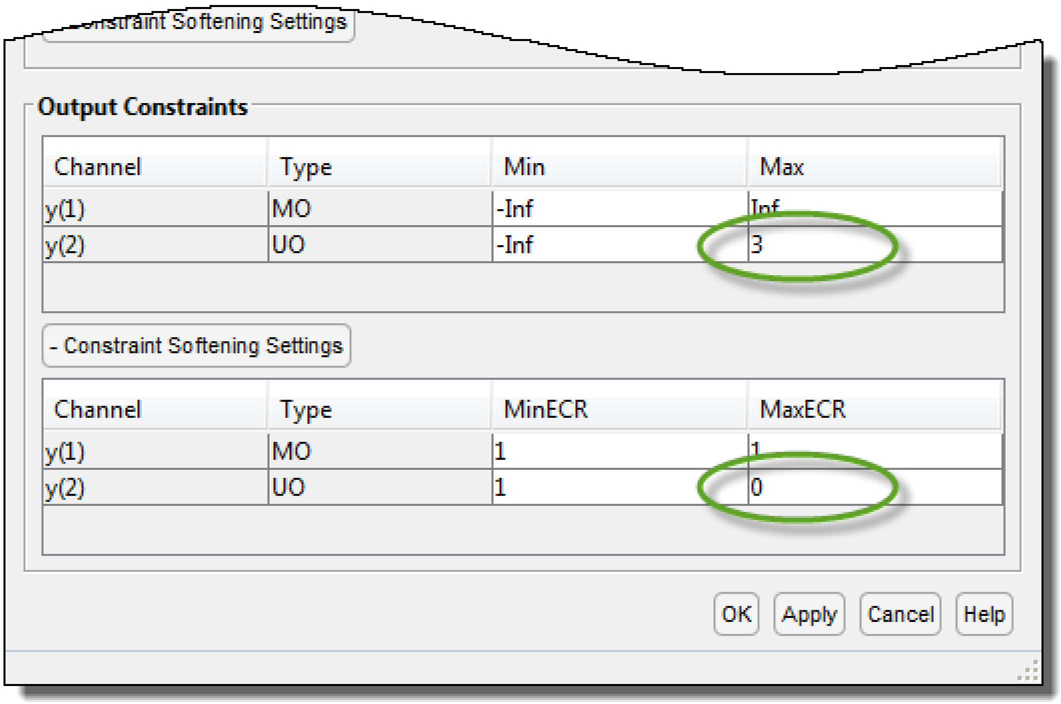


In the **Design** section, click **Constraints**.

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.



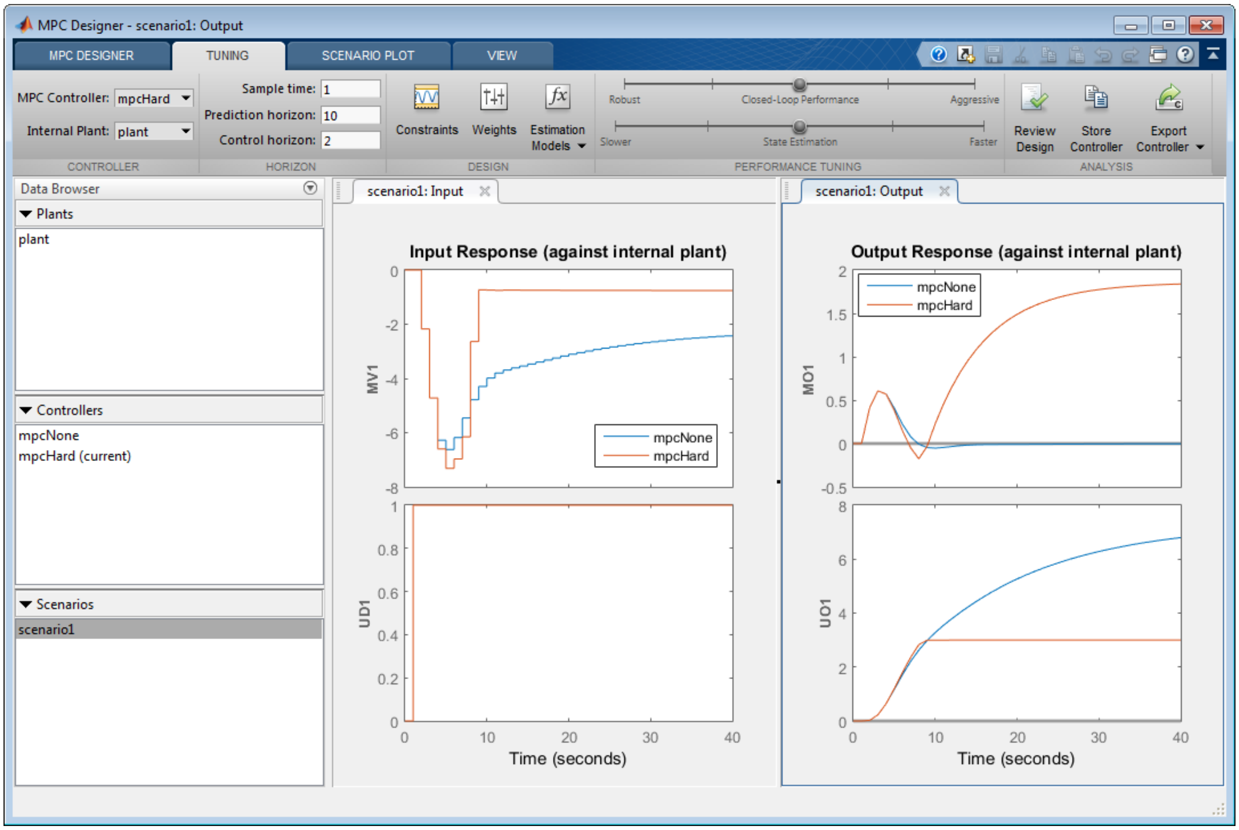
Output Constraints

Channel	Type	Min	Max
y(1)	MO	-Inf	Inf
y(2)	UO	-Inf	3

- Constraint Softening Settings

Channel	Type	MinECR	MaxECR
y(1)	MO	1	1
y(2)	UO	1	0

Click **OK**.



MPC Controller: mpcHard
Internal Plant: plant
Sample time: 1
Prediction horizon: 10
Control horizon: 2

Plants: plant
Controllers: mpcNone, mpcHard (current)
Scenarios: scenario1

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**.

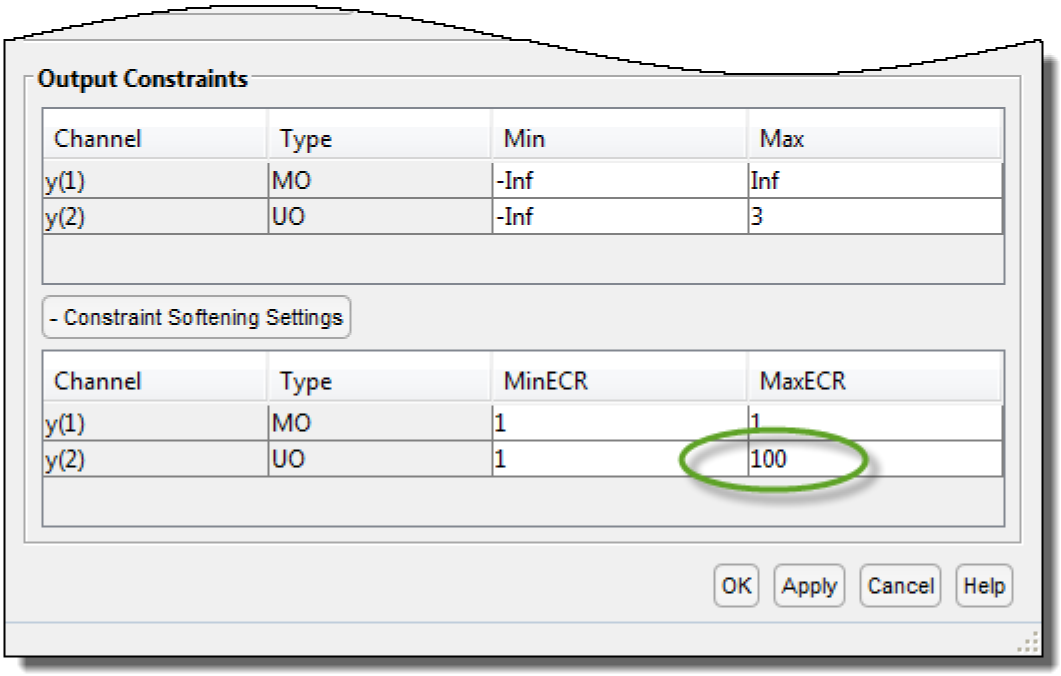
In the **Data Browser**, in the **Controllers** section, rename `mpcHard_Copy` to `mpcSoft`.

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.



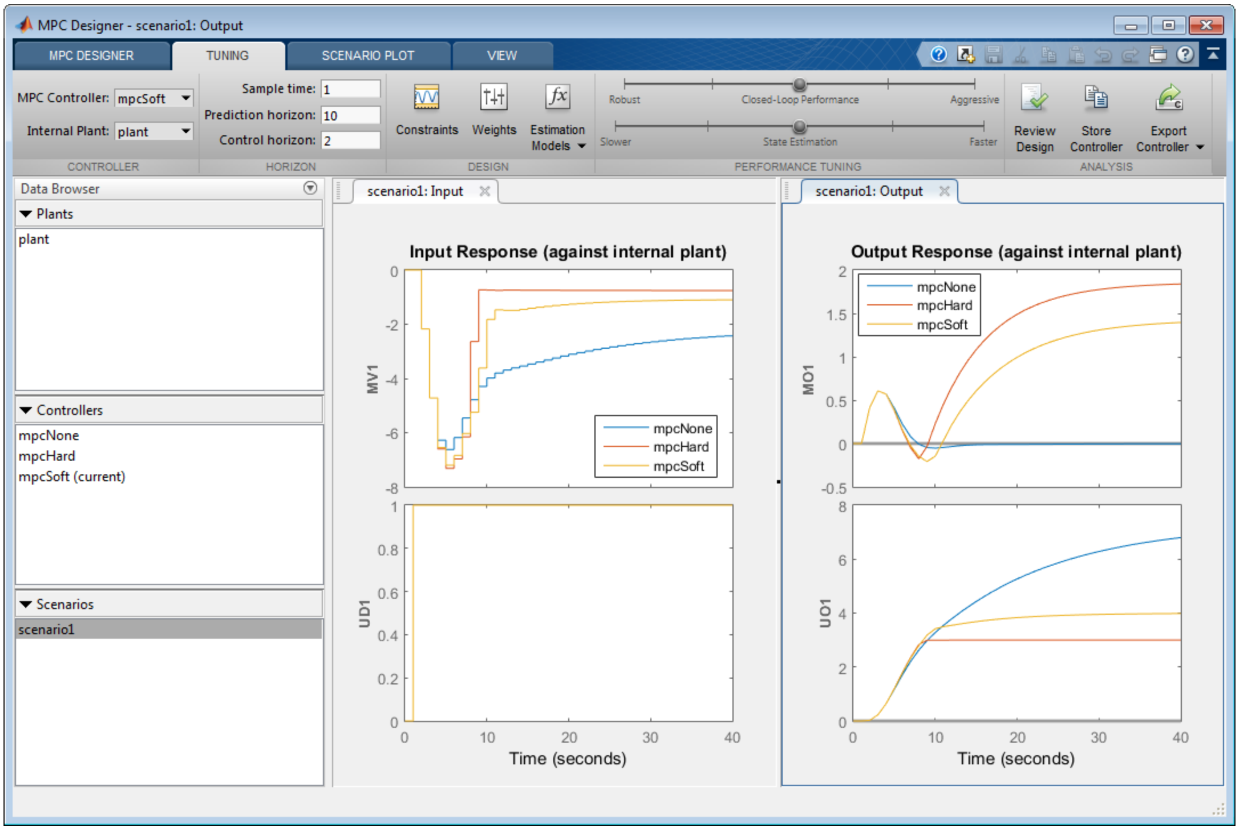
Output Constraints

Channel	Type	Min	Max
y(1)	MO	-Inf	Inf
y(2)	UO	-Inf	3

- Constraint Softening Settings

Channel	Type	MinECR	MaxECR
y(1)	MO	1	1
y(2)	UO	1	100

Click **OK**.

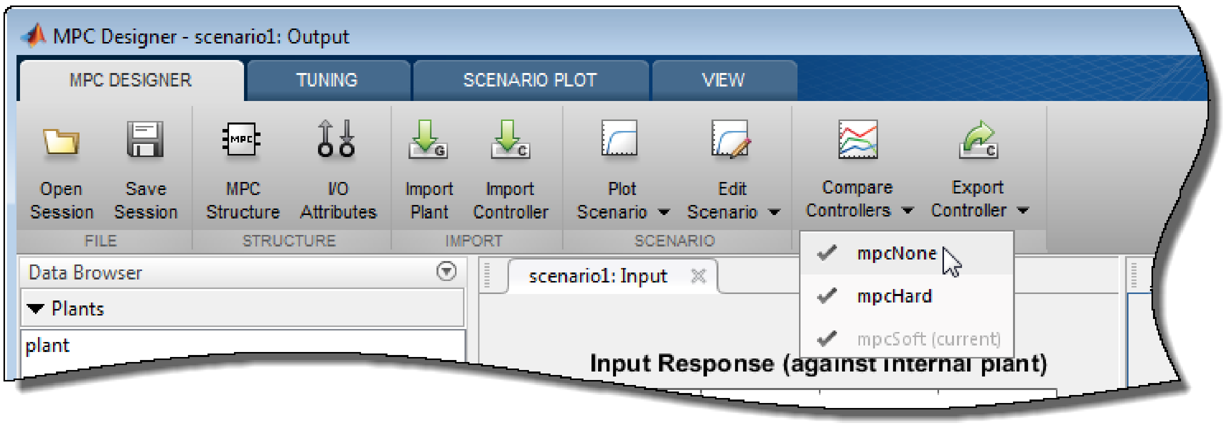


The response plots update to reflect the new `mpcSoft` configuration. In the **Output Response** plot, `mpcSoft` shows a compromise between the previous controller responses.

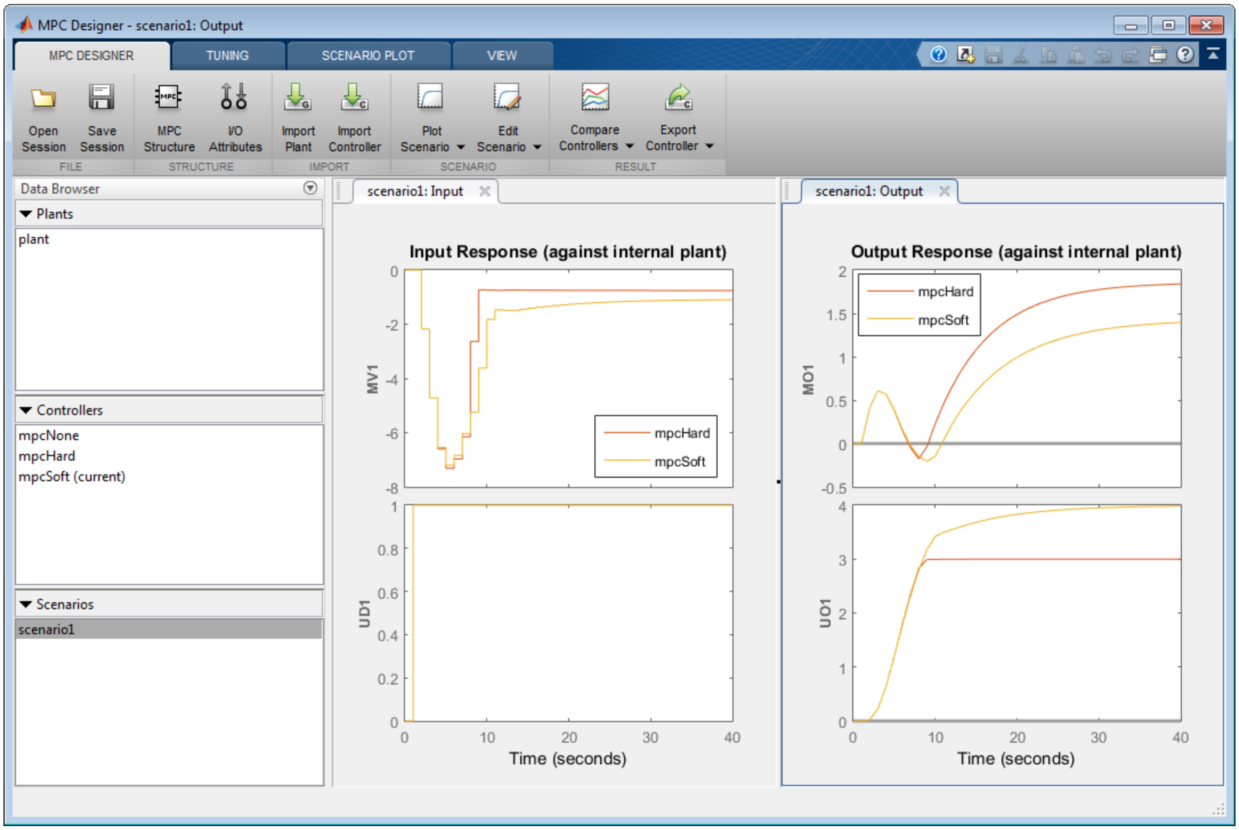
Remove Default Controller Response Plot

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 1-10
- "Design Controller Using MPC Designer"

- “Design MPC Controller in Simulink”

Code Generation

Generate Code and Deploy Controller to Real-Time Targets

Model Predictive Control Toolbox software provides code generation functionality for controllers designed in Simulink and MATLAB.

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:

- MPC Controller
- Multiple MPC Controllers
- Explicit MPC Controller
- Multiple Explicit MPC Controllers
- Adaptive MPC Controller
- Adaptive Cruise Control System
- Lane Keeping Assist System

For more information, see “Simulation and Code Generation Using Simulink Coder” on page 9-8 and “Simulation and Structured Text Generation Using PLC Coder” on page 9-18.

Note The MPC Controller, Explicit MPC Controller, and Adaptive MPC Controller blocks are implemented using the MATLAB Function block. To see the structure, right-click the block, and select **Mask > Look Under Mask**. Then, open the MPC subsystem underneath.

Code Generation in MATLAB

After designing an MPC controller in MATLAB, you can generate C code using MATLAB Coder and deploy it for real-time control.

To generate code for computing optimal MPC control moves:

1. Generate data structures from an MPC or explicit MPC controller using `getCodeGenerationData`.
2. To verify that your controller produces the expected closed-loop results, simulate it using `mpcmoveCodeGeneration` in place of `mpcmove`.
3. Generate code for `mpcmoveCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

For more information, see “Generate Code To Compute Optimal MPC Moves in MATLAB” on page 9-24.

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 go as low as 1–10 ms. 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 2-38.

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 MPC controller constructs a new QP problem, which is defined as:

$$\underset{x}{Min}\left(\frac{1}{2}x^{\text{œ}}Hx+f^{\text{œ}}x\right)$$

subject to the linear inequality constraints

$$Ax \geq b$$

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			

Constant Matrix	Size	Purpose	Implicit MPC	Implicit MPC with Online Weight Tuning	Adaptive MPC or LTV MPC
`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)				
`Ku1`	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
`Mu1`	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:

$$f = \mathtt{Kx}^{\oe} * x_q + \mathtt{Kr}^{\oe} * r_p + \mathtt{Ku1}^{\oe} * m_l + \mathtt{Kv}^{\oe} * v_p + \mathtt{Kut}^{\oe} * u_t$$

$$b = -\left(\mathtt{Mlim} + \mathtt{Mx} * x_q + \mathtt{Mu1} * m_l + \mathtt{Mv} * v_p\right)$$

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 MPC controllers that use a custom QP solver written in either C or Embedded MATLAB®. 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 9-32. For more information on custom QP solvers, see “Custom QP Solver” on page 2-39.

See Also

Functions
`mpcmoveCodeGeneration` | `review`

Blocks
Adaptive MPC Controller | Explicit MPC Controller | MPC Controller | Multiple Explicit MPC Controllers | Multiple MPC Controllers

More About

- “Simulation and Code Generation Using Simulink Coder” on page 9-8
- “Simulation and Structured Text Generation Using PLC Coder” on page 9-18
- “Generate Code To Compute Optimal MPC Moves in MATLAB” on page 9-24

Simulation and Code Generation Using Simulink Coder

This example shows how to simulate and generate real-time code for an MPC Controller block with Simulink Coder. Code can be generated in both single and double precisions.

Required Products

To run this example, Simulink® and Simulink® Coder™ are required.

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

Setup Environment

You must have write-permission to generate the relevant files and the executable. So, before starting simulation and code generation, change the current directory to a temporary directory.

```
cwd = pwd;
tmpdir = tempname;
mkdir(tmpdir);
cd(tmpdir);
```

Define Plant Model and MPC Controller

Define a SISO plant.

```
plant = ss(tf([3 1],[1 0.6 1]));
```

Define the MPC controller for the plant.

```
Ts = 0.1;   %Sampling time
p = 10;     %Prediction horizon
m = 2;      %Control horizon
Weights = struct('MV',0,'MVRate',0.01,'OV',1); % Weights
MV = struct('Min',-Inf,'Max',Inf,'RateMin',-100,'RateMax',100); % Input constraints
OV = struct('Min',-2,'Max',2); % Output constraints
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

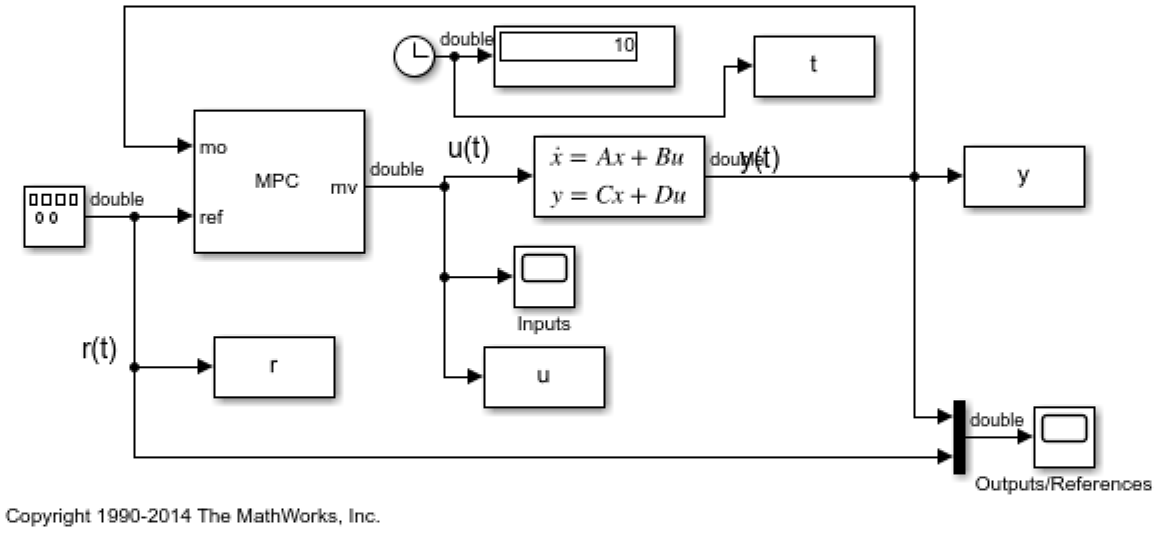
Simulate and Generate Code in Double-Precision

By default, MPC Controller blocks use double-precision in simulation and code generation.

Simulate the model in Simulink.

```
mdl1 = 'mpc_rtwdemo';
open_system(mdl1)
sim(mdl1)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



The controller effort and the plant output are saved into base workspace as variables **u** and **y**, respectively.

Build the model with the `rtwbuild` command.

```
disp('Generating C code... Please wait until it finishes.')
set_param(mdl1,'RTWVerbose','off')
rtwbuild(mdl1);
```

```
Generating C code... Please wait until it finishes.
### Starting build procedure for model: mpc_rtwdemo
### Successful completion of build procedure for model: mpc_rtwdemo
```

On a Windows system, an executable file named "mpc_rtwdemo.exe" appears in the temporary directory after the build process finishes.

Run the executable.

```
if ispc
    disp('Running executable...')
    status = system(mdl1);
else
    disp('The example only runs the executable on Windows system.')
end
```

```
Running executable...

** starting the model **
** created mpc_rtwdemo.mat **
```

After the executable completes successfully (status=0), a data file named "mpc_rtwdemo.mat" appears in the temporary directory.

Compare the responses from the generated code (**rt_u** and **rt_y**) with the responses from the previous simulation in Simulink (**u** and **y**).

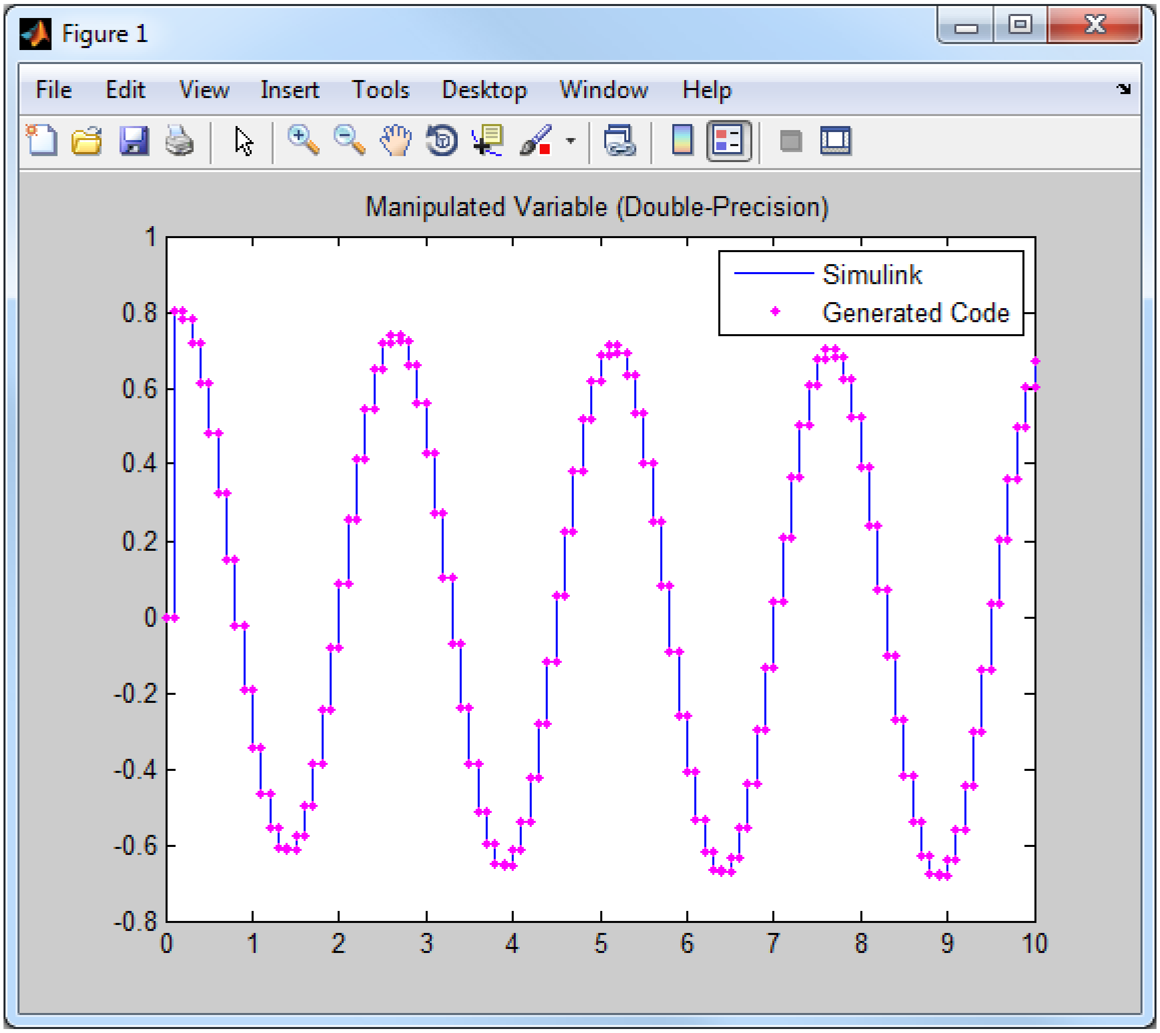
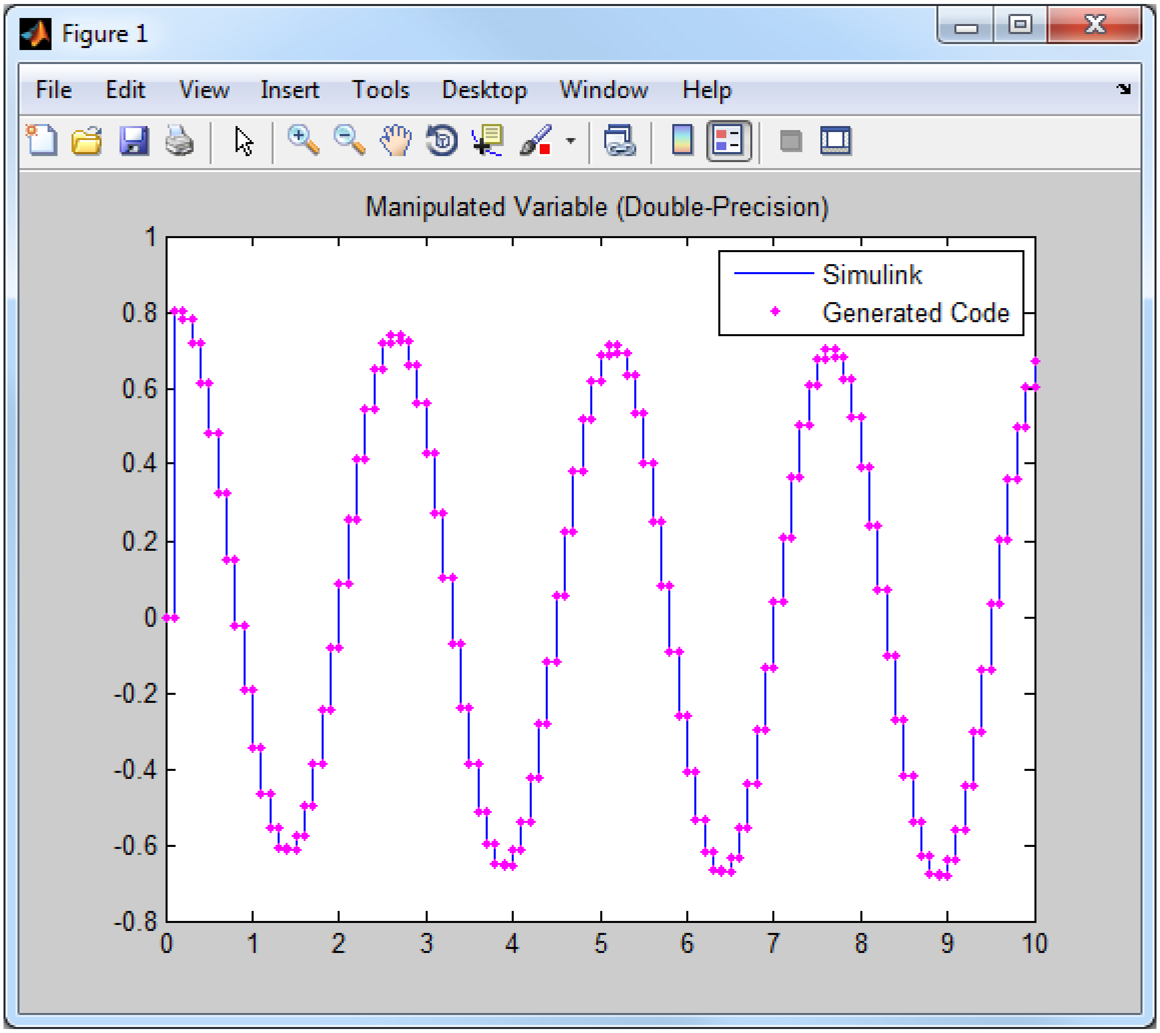


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
Manipulated Variable (Double-Precision)
Simulink
Generated Code
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
1
2
3
4
5
6
7
8
9
10

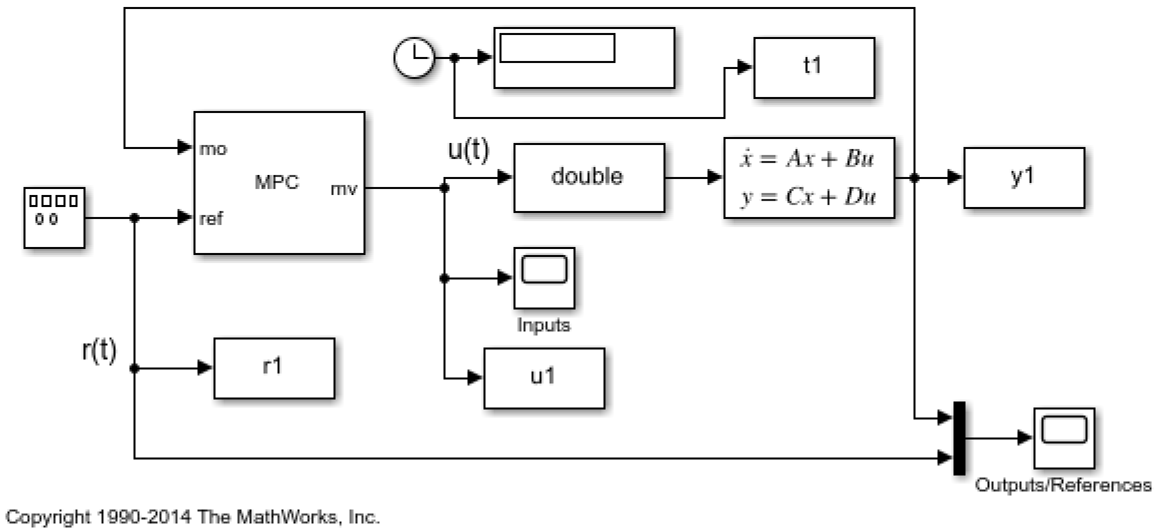


They are numerically equal.

Simulate and Generate Code in Single-Precision

You can also configure the MPC block to use single-precision in simulation and code generation.

```
mdl2 = 'mpc_rtwdemo_single';
open_system(mdl2)
```



To do that, open the MPC block dialog and select "single" as the "output data type" at the bottom of the dialog.

```
open_system([mdl2 '/MPC Controller'])
```

Simulate the model in Simulink.

```
close_system([mdl2 '/MPC Controller'])
sim(mdl2)
```

The controller effort and the plant output are saved into base workspace as variables **u1** and **y1**, respectively.

Build the model with the `rtwbuild` command.

```
disp('Generating C code... Please wait until it finishes.')
set_param(mdl2,'RTWVerbose','off')
rtwbuild(mdl2);
```

```
Generating C code... Please wait until it finishes.
### Starting build procedure for model: mpc_rtwdemo_single
### Successful completion of build procedure for model: mpc_rtwdemo_single
```

On a Windows system, an executable file named "mpc_rtwdemo_single.exe" appears in the temporary directory after the build process finishes.

Run the executable.

```
if ispc
    disp('Running executable...')
    status = system(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**).

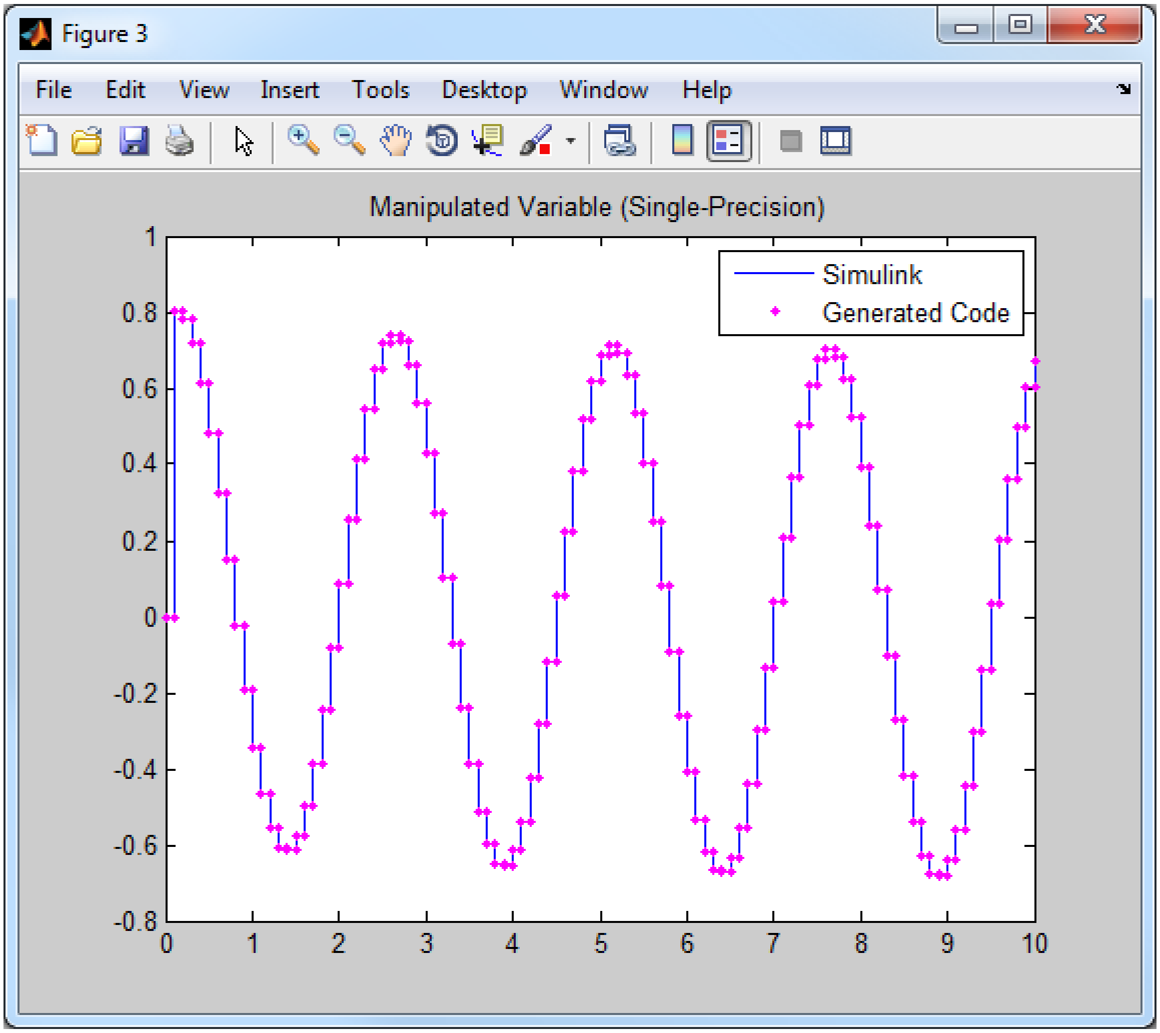
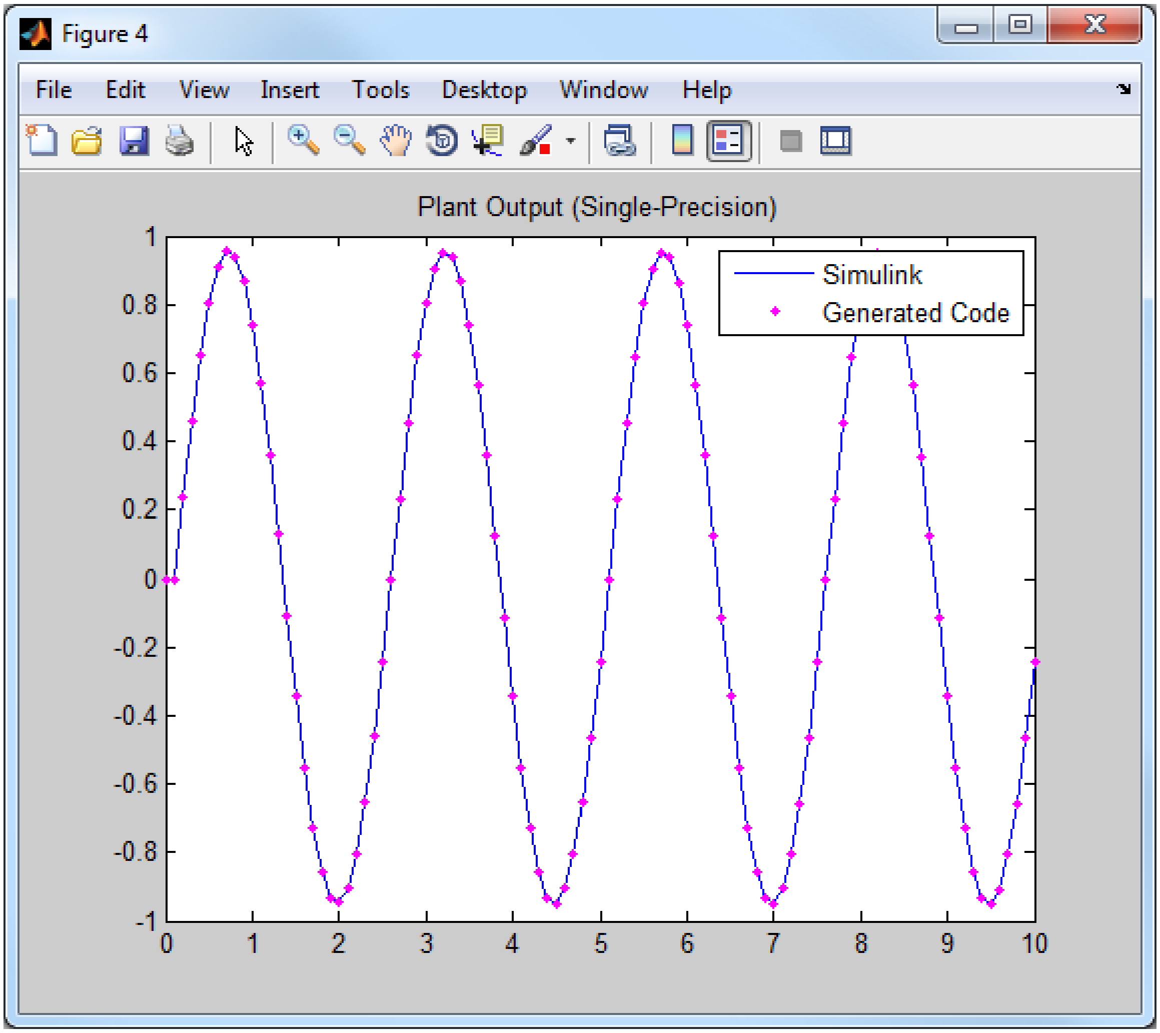


Figure 3
File
Edit
View
Insert
Tools
Desktop
Window
Help
Manipulated Variable (Single-Precision)
Simulink
Generated Code
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
1
2
3
4
5
6
7
8
9
10



They are numerically equal.

Close the Simulink model.

```
bdclose(mdl1)
bdclose(mdl2)
```

```
cd(cwd)
```

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 9-2

Simulation and Structured Text Generation Using PLC Coder

This example shows how to simulate and generate Structured Text for an MPC Controller block using PLC Coder software. The generated code uses single-precision.

Required Products

To run this example, Simulink® and Simulink® PLC Coder™ are required.

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

Setup Environment

You must have write-permission to generate the relevant files and the executable. So, before starting simulation and code generation, change the current directory to a temporary directory.

```
cwd = pwd;
tmpdir = tempname;
mkdir(tmpdir);
cd(tmpdir);
```

Define Plant Model and MPC Controller

Define a SISO plant.

```
plant = ss(tf([3 1],[1 0.6 1]));
```

Define the MPC controller for the plant.

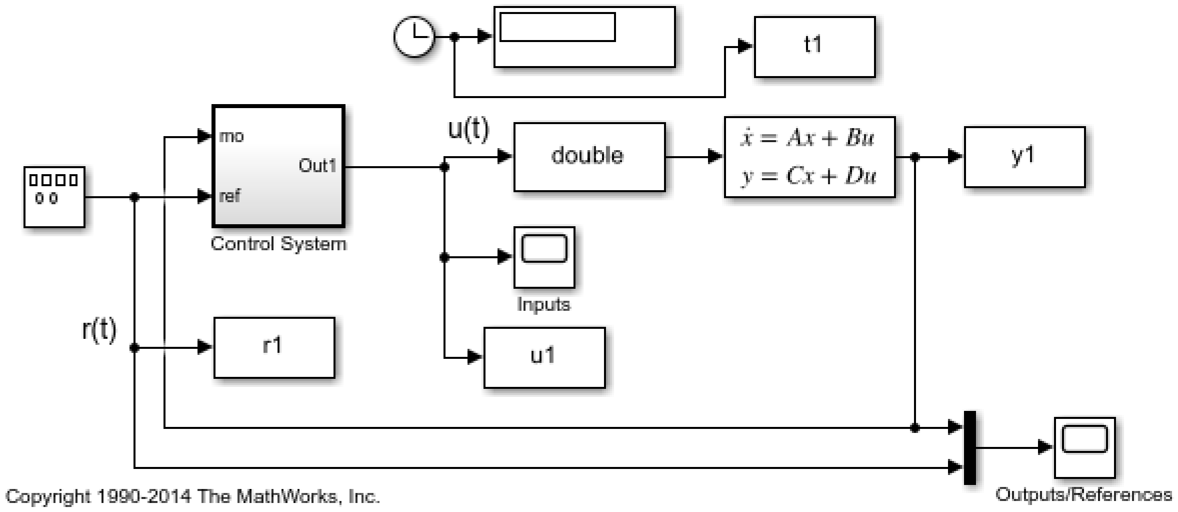
```
Ts = 0.1;   %Sampling time
p = 10;     %Prediction horizon
m = 2;      %Control horizon
Weights = struct('MV',0,'MVRate',0.01,'OV',1); % Weights
MV = struct('Min',-Inf,'Max',Inf,'RateMin',-100,'RateMax',100); % Input constraints
```

```
OV = struct('Min',-2,'Max',2); % Output constraints
mpcobj = mpc(plant,Ts,p,m,Weights,MV,OV);
```

Simulate and Generate Structured Text

Open the Simulink model.

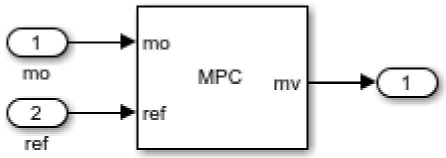
```
mdl = 'mpc_plcdemo';
open_system(mdl)
```



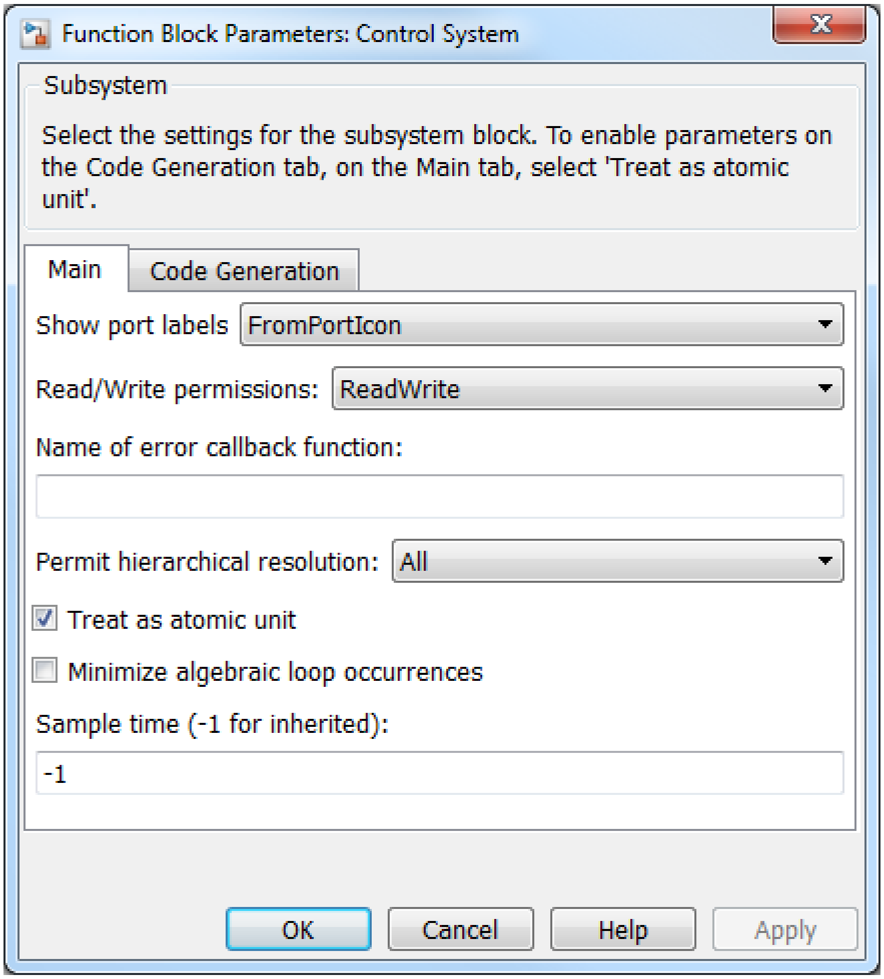
To generate structured text for the MPC Controller block, complete the following two steps:

- Configure the MPC block to use single precision. Select "single" in the "Output data type" combo box in the MPC block dialog.

```
open_system([mdl '/Control System/MPC Controller'])
```



- Put MPC block inside a subsystem block and treat the subsystem block as an atomic unit. Select the "Treat as atomic unit" checkbox in the subsystem block dialog.



Function Block Parameters: Control System

Subsystem

Select the settings for the subsystem block. To enable parameters on the Code Generation tab, on the Main tab, select 'Treat as atomic unit'.

Show port labels: FromPortIcon

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

[✓] Treat as atomic unit

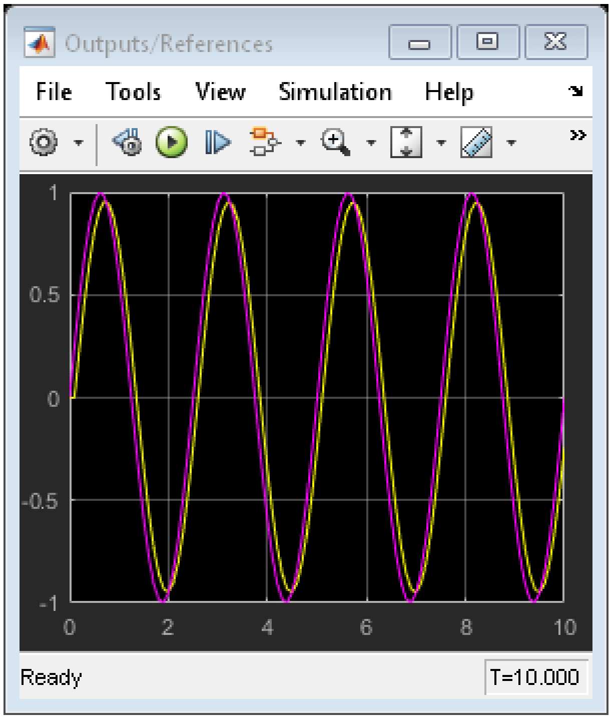
[ ] Minimize algebraic loop occurrences

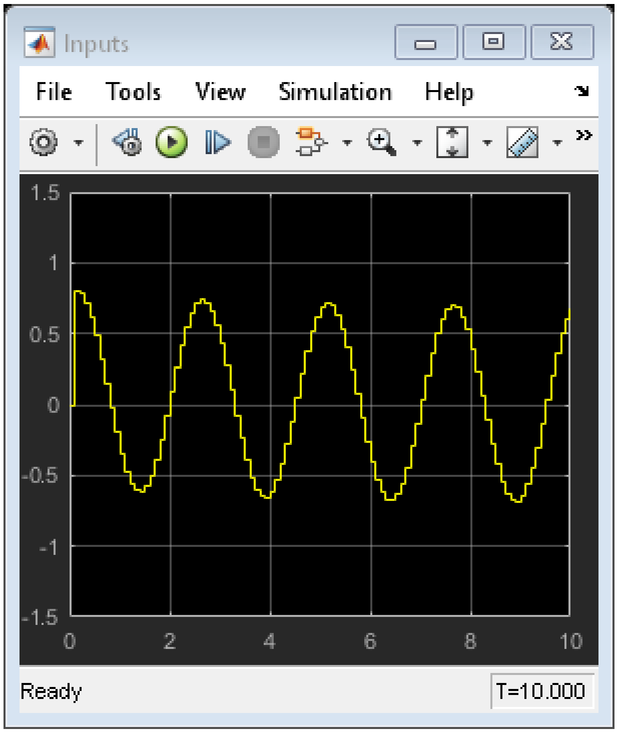
Sample time (-1 for inherited): -1

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



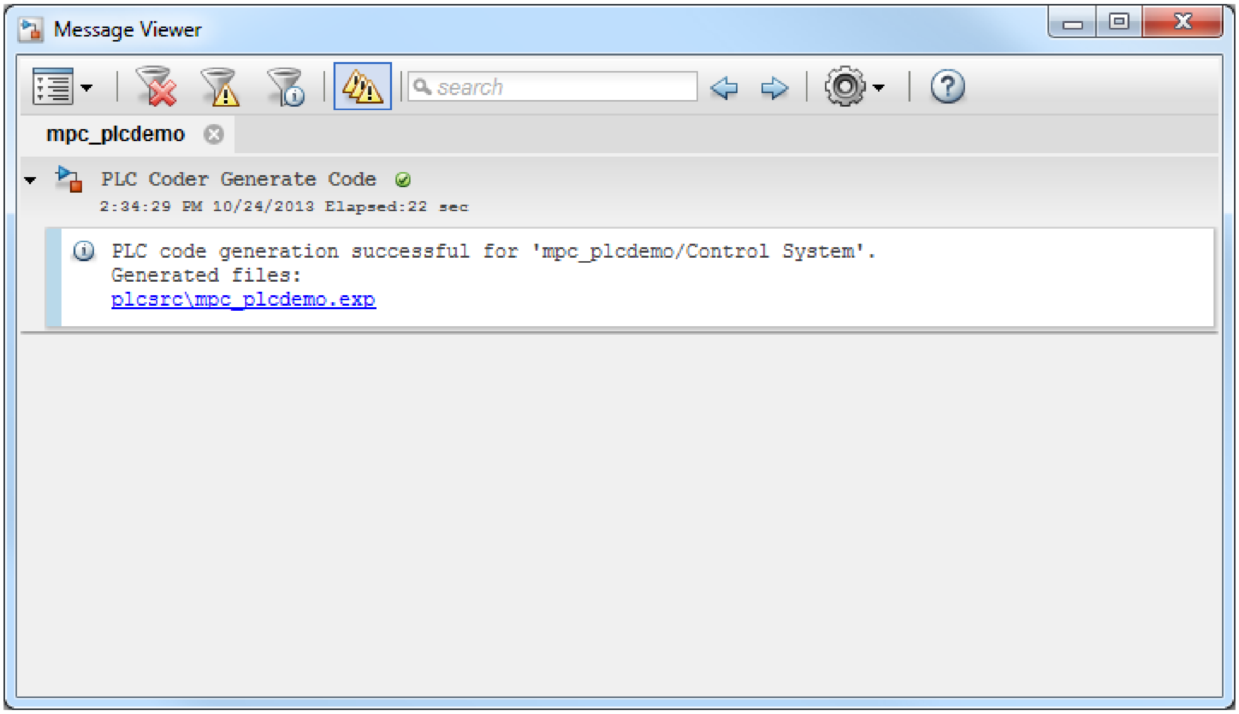


To generate code with the PLC Coder, use the `plcgeneratecode` command.

```
disp('Generating PLC structure text... Please wait until it finishes.')
plcgeneratecode([mdl '/Control System']);
```

```
Generating PLC structure text... Please wait until it finishes.
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.



```
PLC Coder Generate Code
2:34:29 PM 10/24/2013 Elapsed:22 sec
PLC code generation successful for 'mpc_plcdemo/Control System'.
Generated files:
plcsrc\mpc_plcdemo.exp
```

Close the Simulink model.

```
bdclose(mdl)
```

```
cd(cwd)
```

See Also

More About

- “Generate Code and Deploy Controller to Real-Time Targets” on page 9-2

Generate Code To Compute Optimal MPC Moves in MATLAB

This example shows how to use the `mpcmoveCodeGeneration` command to generate C code to compute optimal MPC control moves for real-time applications.

Plant Model

The plant is a single-input, single-output, stable, 2nd order linear plant.

```
plant = tf(5,[1 0.8 3]);
```

Convert the plant to discrete-time, state-space form, and specify a zero initial states vector.

```
Ts = 1;
plant = ss(c2d(plant,Ts));
x0 = zeros(size(plant.B,1),1);
```

Design MPC Controller

Create an MPC controller with default horizons.

```
mpcobj = mpc(plant,Ts);
```

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

Specify controller tuning weights.

```
mpcobj.Weights.MV = 0;
mpcobj.Weights.MVrate = 0.5;
mpcobj.Weights.OV = 1;
```

Specify initial constraints on the manipulated variable and plant output. These constraints will be updated at run time.

```
mpcobj.MV.Min = -1;
mpcobj.MV.Max = 1;
mpcobj.OV.Min = -1;
mpcobj.OV.Max = 1;
```

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.

```
xmpc = mpcstate(mpcobj);
```

```
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

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.
    u = mpcmove(mpcobj,xmpc,y,1,[],options);
    % Update and store plant state.
    x = plant.A*x + plant.B*u;
    uMPCMOVE = [uMPCMOVE u];
end
```

Validate Simulation Results with mpcmoveCodeGeneration Command

To prepare for generating code that computes optimal control moves from MATLAB, it is recommended to reproduce the same control results with the `mpcmoveCodeGeneration` command before using the `codegen` command from the MATLAB Coder product.

```
yCodeGen = [];
uCodeGen = [];
```

Initialize plant states.

```
x = x0;
```

Create data structures to use with `mpcmoveCodeGeneration using getCodeGenerationData`.

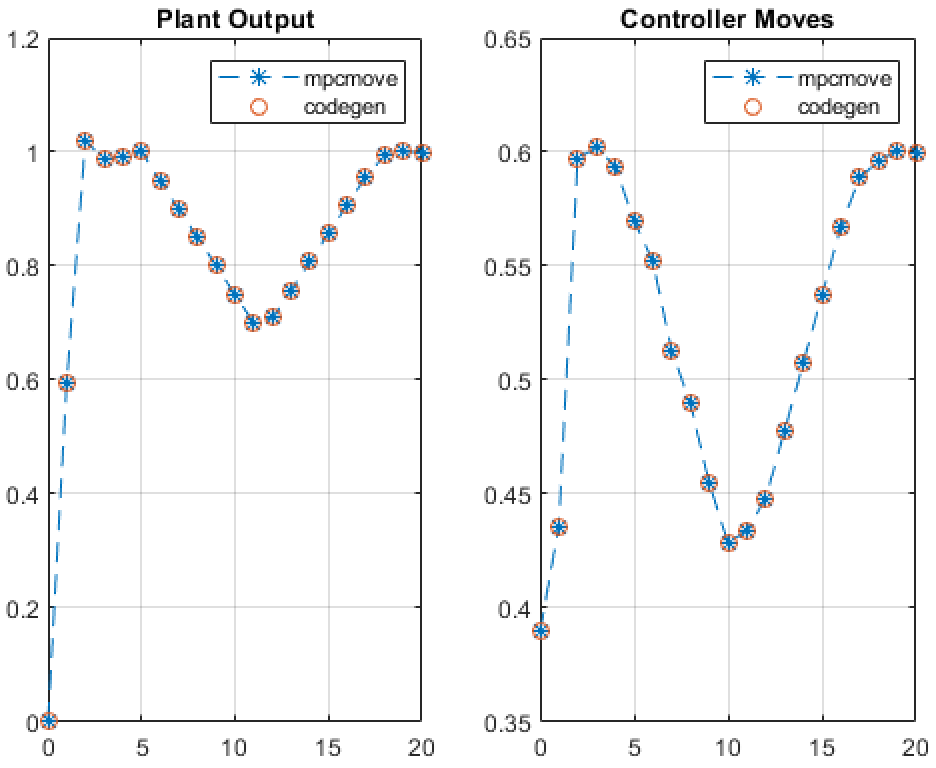
```
[coredata,statedata,onlinedata] = getCodeGenerationData(mpcobj);
```

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 control actions.
    [u,statedata] = mpcmoveCodeGeneration(coredata,statedata,onlinedata);
    % Update and store plant state.
    x = plant.A*x + plant.B*u;
    uCodeGen = [uCodeGen u];
end
```

The simulation results are identical to those using `mpcmove`.

```
t = 0:Ts:Tsim;
figure;
subplot(1,2,1)
plot(t,yMPCMOVE,'--*',t,yCodeGen,'o');
grid
legend('mpcmove','codegen')
title('Plant Output')
subplot(1,2,2)
plot(t,uMPCMOVE,'--*',t,uCodeGen,'o');
grid
legend('mpcmove','codegen')
title('Controller Moves')
```



Genarate MEX Function From `mpcmoveCodeGeneration` Command

To generate C code from the `mpcmoveCodeGeneration` command, use the `codegen` command from the MATLAB Coder product. In this example, generate a MEX function `mpcmoveMEX` to reproduce the simulation results in MATLAB. You can change the code generation target to C/C++ static library, dynamic library, executable, etc. by using a different set of `coder.config` settings.

When generating C code for the `mpcmoveCodeGeneration` command:

- Since no data integrity checks are performed on the input arguments, you must make sure that all the input data has the correct types, dimensions, and values.

- You must define the first input argument, `mpcmove_struct`, as a constant when using `codegen` command.
- The second input argument, `mpcmove_state`, is updated by the command and returned as the second output. In most cases, you do not need to modify its contents and should simply pass it back to the command in the next control interval. The only exception is when custom state estimation is enabled, in which case you must provide the current state estimation with this argument.

```
if ~license ('test', 'MATLAB_Coder')
    disp('MATLAB Coder(TM) is required to run this example.')
    return
end
```

Generate MEX function.

```
fun = 'mpcmoveCodeGeneration';
funOutput = 'mpcmoveMEX';
Cfg = coder.config('mex');
Cfg.DynamicMemoryAllocation = 'off';
codegen('-config',Cfg,fun,'-o',funOutput,'-args',...
    {coder.Constant(coredata),statedata,onlinedata});
```

Initialize data storage.

```
yMEX = [];
uMEX = [];
```

Initialize plant states.

```
x = x0;
```

Use `getCodeGenerationData` to create data structures to use with `mpcmoveCodeGeneration`.

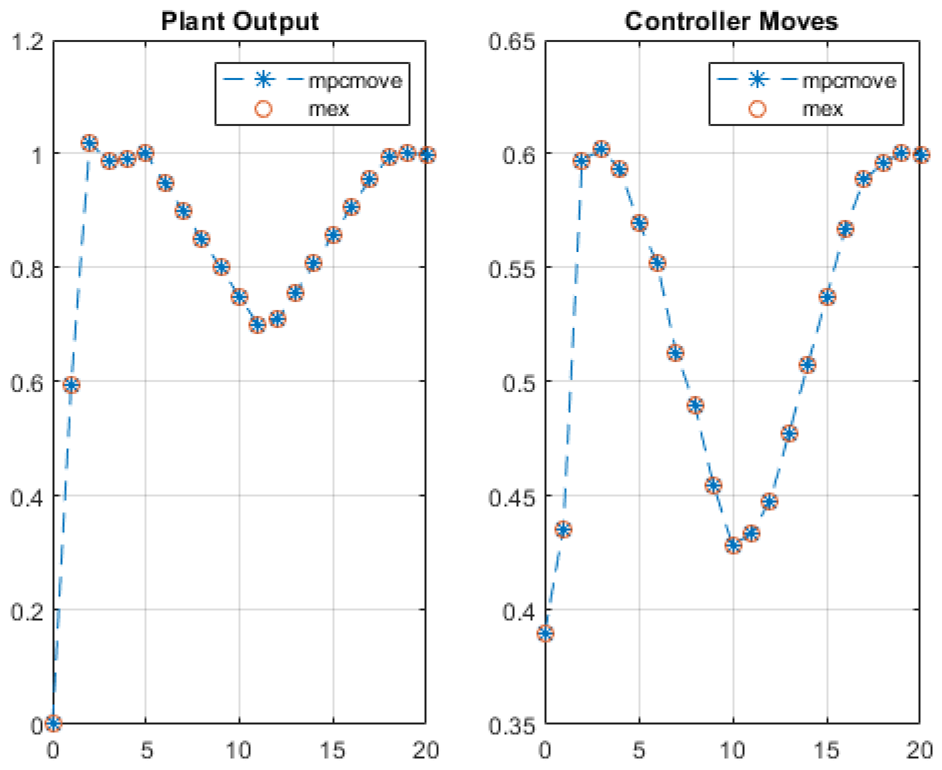
```
[coredata,statedata,onlinedata] = getCodeGenerationData(mpcobj);
```

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 control actions.
    [u,statedata] = mpcmoveMEX(coredata,statedata,onlinedata);
    % Update and store the plant state.
    x = plant.A*x + plant.B*u;
    uMEX = [uMEX u];
end
```

The simulation results are identical to the those 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

`getCodeGenerationData` | `mpcmoveCodeGeneration`

More About

- "Generate Code and Deploy Controller to Real-Time Targets" on page 9-2

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;-tau;-Inf]},...
    'Max',{Inf, [tau;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);
```

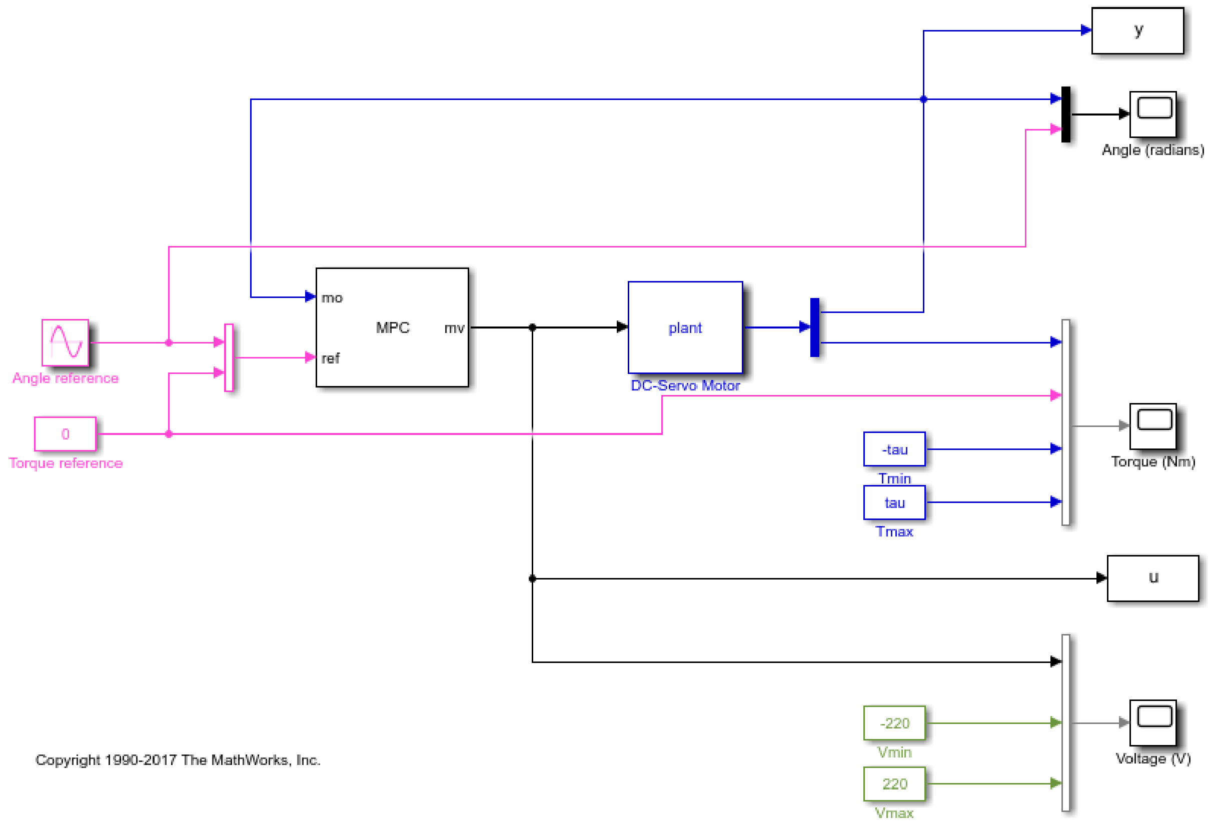
Simulate in Simulink with Built-In QP Solver

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 eac
```

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 eac
```

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 Embedded 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 Embedded MATLAB 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 Code
%   In Simulink, generate code with MPC and Adaptive MPC blocks (require Simuink coder
%
% 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 `rtwbuild` command from Simulink Coder.

```
rtwbuild(mdl)
```

```
### Starting build procedure for model: 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 eac
### Successful completion of build procedure for model: mpc_customQPcodegen
```

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.

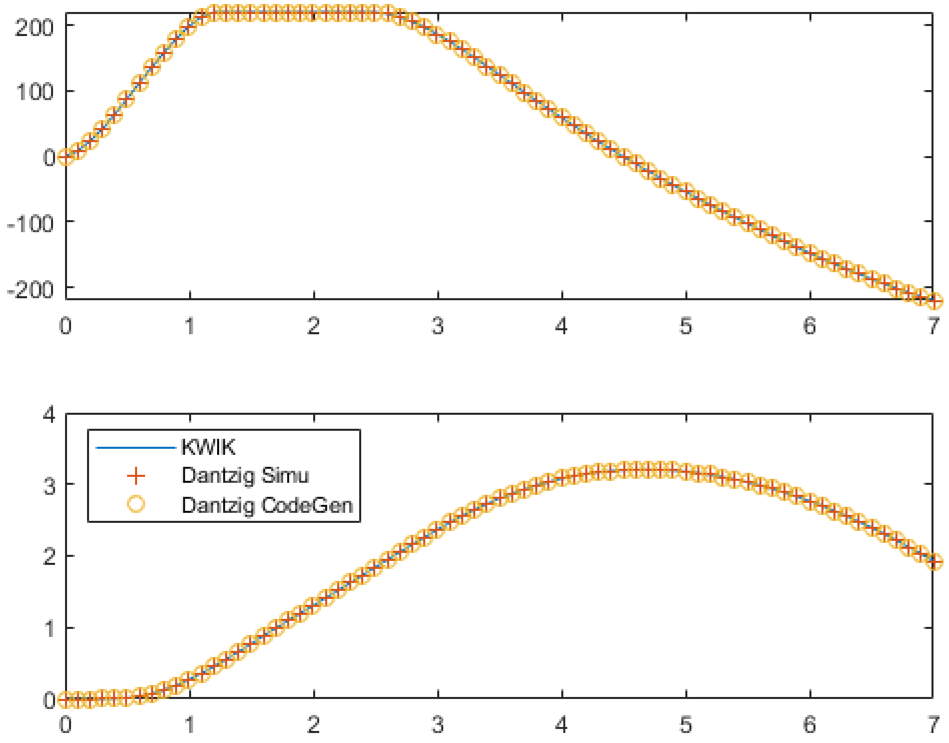
```
if ispc
    status = system(mdl);
    load(mdl)
    uDantzigCodeGen = u;
    yDantzigCodeGen = y;
else
    disp('The example only runs the executable on Windows system.');
end
```

```
** 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 Solver” on page 2-37

Real-Time Control with OPC Toolbox

This example shows how to implement an online model predictive controller application using the OPC client supplied with the OPC Toolbox™.

The example uses the Matrikon™ Simulation OPC server to simulate the behavior of an industrial process on Windows® operating system.

Download the Matrikon™ OPC Simulation Server from "www.matrikon.com"

Download and install the server and set it running either as a service or as an application.

This example needs OPC Toolbox™.

```
if ~mpcchecktoolboxinstalled('opc')
    disp('The example needs OPC Toolbox(TM).')
end
```

```
The example needs OPC Toolbox(TM).
```

Establish a Connection to the OPC Server

Use OPC Toolbox commands to connect to the Matrikon OPC Simulation Server.

```
if mpcchecktoolboxinstalled('opc')
    % Clear any existing opc connections.
    opcreset
    % Flush the callback persistent variables.
    clear mpcopcPlantStep;
    clear mpcopcMPCStep;
    try
        h = opcda('localhost','Matrikon.OPC.Simulation.1');
        connect(h);
    catch ME
        disp('The Matrikon(TM) OPC Simulation Server must be running on the local machi
        return
    end
end
```

Set up the Plant OPC I/O

In practice the plant would be a physical process, and the OPC tags which define its I/O would already have been created on the OPC server. However, since in this case a simulation OPC server is being used, the plant behavior must be simulated. This is achieved by defining tags for the plant manipulated and measured variables and creating a callback (mpcopcPlantStep) to simulate plant response to changes in the manipulated variables. Two OPC groups are required, one to represent the two manipulated variables to be read by the plant simulator and another to write back the two measured plant outputs storing the results of the plant simulation.

```
if mpcchecktoolboxinstalled('opc')
    % Build an opc group for 2 plant inputs and initialize them to zero.
    plant_read = addgroup(h,'plant_read');
    imv1 = additem(plant_read,'Bucket Brigade.Real8', 'double');
    writeasync(imv1,0);
    imv2 = additem(plant_read,'Bucket Brigade.Real4', 'double');
    writeasync(imv2,0);
    % Build an opc group for plant outputs.
    plant_write = addgroup(h,'plant_write');
    opv1 = additem(plant_write,'Bucket Brigade.Time', 'double');
    opv2 = additem(plant_write,'Bucket Brigade.Money', 'double');
    plant_write.WriteAsyncFcn = []; % Suppress command line display.
end
```

Specify the MPC Controller Which Will Control the Simulated Plant

Create plant model.

```
plant_model = ss([-.2 -.1; 0 -.05],eye(2,2),eye(2,2),zeros(2,2));
disc_plant_model = c2d(plant_model,1);
% We assume no model mismatch, a control horizon 6 samples and
% prediction horizon 20 samples.
mpcobj = mpc(disc_plant_model,1,20,6);
mpcobj.weights.ManipulatedVariablesRate = [1 1];
% Build an internal MPC object structure so that the MPC object
% is not rebuilt each callback execution.
state = mpcstate(mpcobj);
y1 = mpcmove(mpcobj,state,[1;1]',[1 1]');
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Build the OPC I/O for the MPC Controller

Build two OPC groups, one to read the two measured plant outputs and the other to write back the two manipulated variables.

```
if mpcchecktoolboxinstalled('opc')
    % Build an opc group for MPC inputs.
    mpc_read = addgroup(h,'mpc_read');
    impcpv1 = additem(mpc_read,'Bucket Brigade.Time', 'double');
    writeasync(impcpv1,0);
    impcpv2 = additem(mpc_read,'Bucket Brigade.Money', 'double');
    writeasync(impcpv2,0);
    impcref1 = additem(mpc_read,'Bucket Brigade.Int2', 'double');
    writeasync(impcref1,1);
    impcref2 = additem(mpc_read,'Bucket Brigade.Int4', 'double');
    writeasync(impcref2,1);
    % Build an opc group for mpc outputs.
    mpc_write = addgroup(h,'mpc_write');
    additem(mpc_write,'Bucket Brigade.Real8', 'double');
    additem(mpc_write,'Bucket Brigade.Real4', 'double');
    % Suppress command line display.
    mpc_write.WriteAsyncFcn = [];
end
```

Build OPC Groups to Trigger Execution of the Plant Simulator & Controller

Build two opc groups based on the same external opc timer to trigger execution of both plant simulation and MPC execution when the contents of the OPC time tag changes.

```
if mpcchecktoolboxinstalled('opc')
    gtime = addgroup(h,'time');
    time_tag = additem(gtime,'Triangle Waves.Real8');
    gtime.UpdateRate = 1;
    gtime.DataChangeFcn = {@mpcopcPlantStep plant_read plant_write disc_plant_model};
    gmpctime = addgroup(h,'mpctime');
    additem(gmpctime,'Triangle Waves.Real8');
    gmpctime.UpdateRate = 1;
    gmpctime.DataChangeFcn = {@mpcopcMPCStep mpc_read mpc_write mpcobj};
end
```

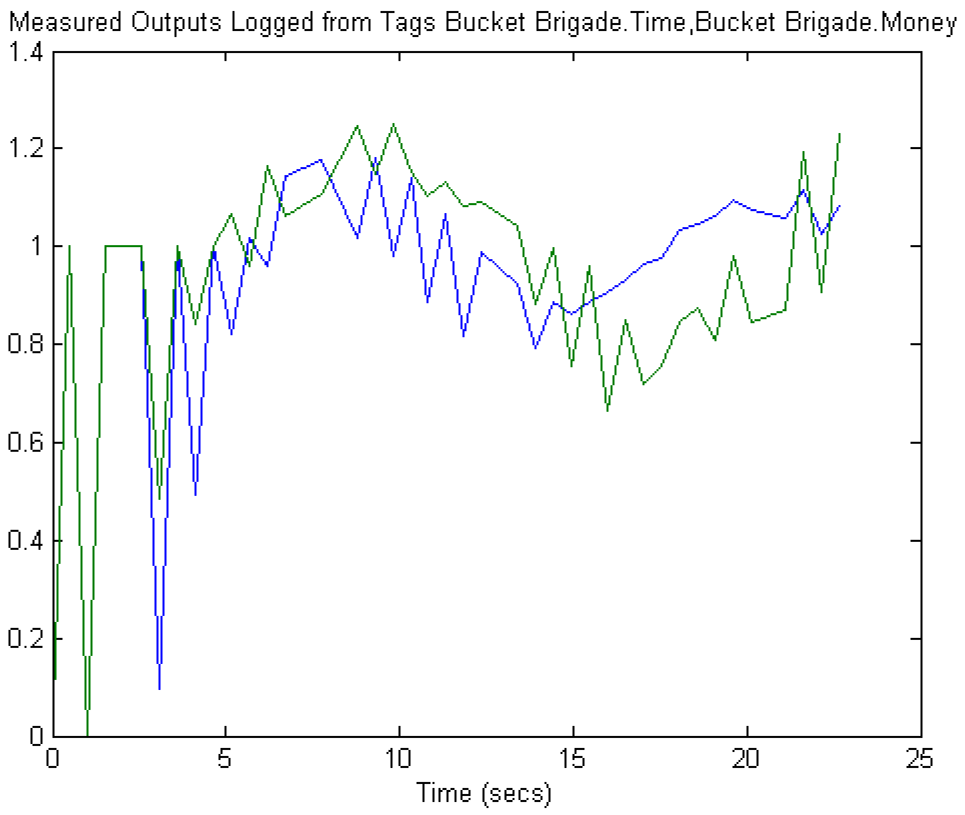
Log Data from the Plant Measured Outputs

Log the plant measured outputs from tags 'Bucket Brigade.Money' and 'Bucket Brigade.Money'.

```
if mpcchecktoolboxinstalled('opc')
    mpc_read.RecordsToAcquire = 40;
    start(mpc_read);
    while mpc_read.RecordsAcquired < mpc_read.RecordsToAcquire
       pause(3)
       fprintf('Logging data: Record %d / %d',mpc_read.RecordsAcquired,mpc_read.Records
    end
    stop(mpc_read);
end
```

Extract and Plot the Logged Data

```
if mpcchecktoolboxinstalled('opc')
    [itemID, value, quality, timeStamp, eventTime] = getdata(mpc_read,'double');
    plot((timeStamp(:,1)-timeStamp(1,1))*24*60*60,value)
    title('Measured Outputs Logged from Tags Bucket Brigade.Time,Bucket Brigade.Money')
    xlabel('Time (secs)');
end
```



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)

Economic MPC

Economic MPC

Economic MPC controllers optimize control actions to minimize a generic cost function under arbitrary nonlinear constraints. 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 have the following features:

- Linear prediction models
- Linear constraints
- Minimization of a quadratic performance criterion (cost function)

A quadratic cost function is adequate for tracking specified output and manipulated variable references. However, some applications can require optimizing for different criteria, such as a combination of linear or nonlinear functions of the system states and inputs. Such applications include using fuel consumption maps in an engine control problem and supporting Wiener models where the output is a nonlinear function of the states. Also, while linear constraints capture the most significant operating constraints of an MPC controller, using a generic cost function can impose additional nonlinear constraints on the system.

When you implement economic MPC, the controller:

- Still uses a linear prediction model.
- Uses your generic cost function instead of the built-in quadratic cost function.
- Applies your nonlinear constraints in addition to any linear constraints you define in the controller object.
- Computes optimal control moves by solving a nonlinear optimization problem using the SQP algorithm in `fmincon`.

Using economic MPC requires Optimization Toolbox™software.

To implement an economic MPC controller, create custom functions that define the generic cost function and arbitrary constraints. For more information, see “Specify Generic Cost Function” on page 10-4 and “Specify Nonlinear Constraints” on page 10-8.

You can simulate economic MPC controllers:

- In Simulink using the MPC Controller, Adaptive MPC Controller, and Multiple MPC Controllers blocks.
- At the command line using `mpcmove`, `mpcmoveAdaptive`, `mpcmoveMultiple`, and `sim`.

Note Economic MPC does not support:

- Code generation.
- Designing controllers using the **MPC Designer** app.

See Also

More About

- “Economic MPC Control of Ethylene Oxide Production” on page 10-13

Specify Generic Cost Function

While traditional implicit MPC controllers optimize control actions to minimize a quadratic cost function, economic MPC controllers support generic cost functions. For example, you can specify your cost function as a combination of linear or nonlinear functions of the system states and inputs.

Using such a custom cost function can impose additional nonlinear constraints on your system. If this is the case for your application, specify these constraints in a separate custom function. For more information, see “Specify Nonlinear Constraints” on page 10-8.

To use a custom generic cost function:

1. Copy the custom cost function template file to your working folder or anywhere on the MATLAB path, and rename it `mpcCustomCostFcn.m`.

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

   As an example, the template file implements a standard quadratic cost function.
2. Specify your generic cost function by modifying the template file.
3. Configure the controller `MPCobj` to use the custom cost function.

   ```
   MPCobj.Optimizer.CustomCostFcn = true;
   ```

Custom Cost Function

As shown in the template file `mpcCustomCostFcn.txt`, your custom cost function must have the following signature:

```
function [f,dfdy,dfdu,dfddu,dfdslack] = ...
    mpcCustomCostFcn(y,yref,u,uref,du,v,slack,varargin)
```

The input arguments for this function are shown in the following table, where:

- k is the current control interval.
- p is the prediction horizon.
- n_y is the number of output variables.

- n_{mv} is the number of manipulated variables.
- n_{md} is the number of measured disturbances.

The MPC controller generates these inputs and passes them to the custom function at each control interval.

Input Argument	**Description**	**Specified As**
`y`	Predicted plant outputs, defined over the prediction horizon from k+1 to k+p	p-by-n_y array
`yref`	Previewed output references, defined over the prediction horizon from k+1 to k+p	p-by-n_y array
`u`	Optimal manipulated variable control sequences, defined over the prediction horizon from k to k+p-1	p-by-n_{mv} array
`uref`	Previewed manipulated variable targets, defined over the prediction horizon from k to k+p-1 While the software does not currently support manipulated variable previewing, this interface is general.	p-by-n_{mv} array
`du`	Manipulated variable rates of change, defined over the prediction horizon from k to k+p-1	p-by-n_{mv} array
`v`	Previewed measured disturbances, defined over the prediction horizon from k to k+p	(p+1)-by-n_{md} array (`[]` if there are no measured disturbances)
`slack`	Global slack variable used by all soft constraints	scalar
`varargin`	Additional input arguments (for future use).	`[]`

The output arguments for this function are shown in the following table.

Output Argument	Description	Returned As
`f`	Generic cost function value	scalar
`dfdy`	Gradients of `f` with respect to the plant outputs	p-by-n_y array (`[]` if unknown)
`dfdu`	Gradients of `f` with respect to the manipulated variables	p-by-n_{mv} array (`[]` if unknown)
`dfddu`	Gradients of `f` with respect to the manipulated variable rates of change	p-by-n_{mv} array (`[]` if unknown)
`dfdslack`	Gradients of `f` with respect to the global slack variable	scalar (`[]` if unknown)

Typically, you optimize the controller 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:

$$f = (y_1 - yref_1)^2 + (u_1 y_2)^2$$

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 output. To compute the total cost function across the prediction horizon, use:

```
f = sum(sum((y(:,1)-yref(:,1)).^2 + (u(:,1).*y(:,2)).^2));
```

While computing and returning gradients is optional, doing so improves the computational efficiency of the controller optimization. To find the gradients, compute the partial derivatives of the cost function at each step of the prediction horizon. For example, for the preceding cost function, the gradients are:

```
dfdy = zeros(p,ny);
dfdy(:,1) = 2*(y(:,1)-yref(:,1));
dfdy(:,2) = 2*y(:,2).*u(:,1).^2;
dfdu = zeros(p,nmv);
dfdu(:,1) = 2*u(:,1).*y(:,2).^2;
dfddu = zeros(p,nmv);
dfdslack = 0;
```

Within your custom function, to obtain the:

- Prediction horizon, `p`, use:

  ```
  p = size(y,1);
  ```
- Number of outputs, `ny`, use:

  ```
  ny = size(y,2);
  ```
- Number of manipulated variables, `nmv`, use:

  ```
  nmv = size(u,2);
  ```
- Number of measured disturbances, `nmd`, use:

  ```
  nmd = size(v,2);
  ```
- Previous manipulated variable values; that is, `u(k-1)`, use:

  ```
  u_prev = u(1,:) - du(1,:);
  ```

See Also

`mpc`

More About

- “Economic MPC” on page 10-2
- “Specify Nonlinear Constraints” on page 10-8
- “Economic MPC Control of Ethylene Oxide Production” on page 10-13

Specify Nonlinear Constraints

Some MPC applications, such as “Economic MPC” on page 10-2, can impose nonlinear constraints on your system. For such applications, you specify these constraints using a custom constraint function.

To use custom nonlinear constraints:

1. Copy the custom constraint function template file to your working folder or anywhere on the MATLAB path, and rename it `mpcCustomConstraintFcn.m`.

   ```
   src = which('mpcCustomConstraintFcn.txt');
   dest = fullfile(pwd,'mpcCustomConstraintFcn.m');
   copyfile(src,dest,'f');
   ```
2. Specify your nonlinear constraints by modifying the template file.
3. Configure the controller `MPCobj` to use the custom constraint function.

   ```
   MPCobj.Optimizer.CustomConstraintFcn = true;
   ```

Custom Constraint Function

As shown in the template file `mpcCustomConstraintFcn.txt`, your custom constraint function must have the following signature:

```
function [C,Ceq,dCdy,dCdu,dCddu,dCdslack,dCeqdy,dCeqdu,dCeqddu,dCeqdslack] = ...
    mpcCustomConstraintFcn(y,yref,u,uref,du,v,slack,varargin)
```

The input arguments for this function are shown in the following table, where:

- k is the current control interval.
- p is the prediction horizon.
- n_y is the number of output variables.
- n_{mv} is the number of manipulated variables.
- n_{md} is the number of measured disturbances.

The MPC controller generates these inputs and passes them to the custom function at each control interval.

Input Argument	Description	Specified As
`y`	Predicted plant outputs, defined over the prediction horizon from k+1 to k+p	p-by-n_y array
`yref`	Previewed output references, defined over the prediction horizon from k+1 to k+p	p-by-n_y array
`u`	Optimal manipulated variable control sequences, defined over the prediction horizon from k to k+p-1	p-by-n_{mv} array
`uref`	Previewed manipulated variable targets, defined over the prediction horizon from k to k+p-1 While the software does not currently support manipulated variable previewing, this interface is general.	p-by-n_{mv} array
`du`	Manipulated variable rates of change, defined over the prediction horizon from k to k+p-1	p-by-n_{mv} array
`v`	Previewed measured disturbances, defined over the prediction horizon from k to k+p	(p+1)-by-n_{md} array (`[]` if there are no measured disturbances)
`slack`	Global slack variable used by all soft constraints	scalar
`varargin`	Additional input arguments (for future use).	`[]`

The output arguments for this function are shown in the following table, where n is the number of inequality constraints and n_{eq} is the number of equality constraints.

Output Argument	Description	Returned As
`C`	Nonlinear inequality constraints, where `C` <= 0	Column vector of length n (`[]` if there are no inequality constraints)

Output Argument	Description	Returned As
`Ceq`	Nonlinear equality constraints, where `Ceq` = 0	Column vector of length n_{eq} (`[]` if there are no equality constraints)
`dCdy`	Gradients of `C` with respect to the plant outputs	p-by-n_y-by-n array (`[]` if unknown or there are no inequality constraints)
`dCdu`	Gradients of `C` with respect to the manipulated variables	p-by-n_{mv}-by-n array (`[]` if unknown or there are no inequality constraints)
`dCddu`	Gradients of `C` with respect to the manipulated variable rates of change	p-by-n_{mv}-by-n array (`[]` if unknown or there are no inequality constraints)
`dCdslack`	Gradients of `C` with respect to the global slack variable	Row vector of length n (`[]` if unknown or there are no inequality constraints)
`dCeqdy`	Gradients of `Ceq` with respect to the plant outputs	p-by-n_y-by-n_{eq} array (`[]` if unknown or there are no equality constraints)
`dCeqdu`	Gradients of `Ceq` with respect to the manipulated variables	p-by-n_{mv}-by-n_{eq} array (`[]` if unknown or there are no equality constraints)
`dCeqddu`	Gradients of `Ceq` with respect to the manipulated variable rates of change	p-by-n_{mv}-by-n_{eq} array (`[]` if unknown or there are no equality constraints)
`dCeqdslack`	Gradients of `Ceq` with respect to the global slack variable	Row vector of length n_{eq} (`[]` if unknown or there are no equality constraints)

Typically, you optimize the controller to satisfy your constraints at each step of the prediction horizon. For example, suppose that you have the following two inequality constraints:

$$2y_1^2 - 3y_2 - 10 \le 0$$
$$u_1^2 - 5 \le 0$$

To define the constraint values across the prediction horizon, use:

```
C = [2*y(:,1).^2 - 3*y(:,2) - 10; u(:,1).^2 - 5];
```

Assuming a prediction horizon of `p = 3`, `C` is a column vector with six constraints (`n = 6`).

While computing and returning gradients is optional, doing so improves the computational efficiency of the controller optimization. To find the gradients, compute the partial derivatives of each constraint at each step of the prediction horizon. For example, `dCdy(i,j,k)` is the partial derivative of `C(k)` with respect to `y(i,j)`.

Assuming `ny = 2` and `nmv = 1`, the gradients of the preceding constraints are:

```
dCdy = zeros(p,ny,n);
dCdy(:,1,1:3) = diag(4*y(:,1));
dCdy(:,2,1:3) = -3*eye(3);

dCdu = zeros(p,nmv,n);
dCdu(:,1,4:6) = diag(2*u(:,1));

dCddu = zeros(p,nmv,n);
dCdslack = 0;
```

Within your custom function, to obtain the:

- Prediction horizon, `p`, use:

  ```
  p = size(y,1);
  ```
- Number of outputs, `ny`, use:

  ```
  ny = size(y,2);
  ```
- Number of manipulated variables, `nmv`, use:

  ```
  nmv = size(u,2);
  ```
- Number of measured disturbances, `nmd`, use:

  ```
  nmd = size(v,2);
  ```

- Previous manipulated variable values; that is, `u(k-1)`, use:

  ```
  u_prev = u(1,:) - du(1,:);
  ```

See Also

`mpc`

More About

- “Economic MPC” on page 10-2
- “Specify Generic Cost Function” on page 10-4
- “Economic MPC Control of Ethylene Oxide Production”

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 with custom cost and constraint functions.

This example requires Simulink® software to simulate nonlinear MPC control of the ethylene oxidation plant in Simulink.

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

The example also uses the `fsolve` and `fmincon` commands from the Optimization Toolbox™.

```
if ~mpcchecktoolboxinstalled('optim')
    disp('Optimization Toolbox must be installed to run this example.')
    return
end
```

Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc_featured','main'));
```

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 reduce C2H4O production. A mixture of air and ethylene is continuously fed into the tank. 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 six 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)
- Total volumetric flow rate exiting the reactor (x_5)
- Integrator state of a PI reactor temperature controller (x_6)

The plant has three inputs:

- Total volumetric feed flow rate (u_1)
- C2H4 concentration in the feed (u_2)
- Reactor temperature setpoint (u_3)

The plant has two outputs:

- C2H4O concentration in the reactor (y_1, equivalent to x_3)
- Total volumetric flow rate exiting the reactor (y_2, equivalent to x_5)

All variables in the model are scaled to be dimensionless and of unity order. The basic plant equations and parameters are obtained from [1].

The reactions are exothermic. Therefore, continuous feedback control of the reactor temperature has been added, as would normally be done for safe and stable operation. To hold the reactor temperature at a specified setpoint, the PI control law manipulates the coolant temperature.

The plant is asymptotically open-loop stable.

Control Objectives

The primary control objective is to maximize the time-averaged ethylene oxide (C2H4O) production rate, which in turn maximizes profit. The instantaneous C2H4O production rate is $y_1 y_2$.

Also, during the production operation, the ethylene feed rate, $u_1 u_2$, must always equal the available ethylene coming from an upstream process. This equality constraint is referred to as *C2H4 availability* in this example.

Therefore, the MPC controller must maximize the C2H4O production rate from the three chemical reactions under the C2H4 availability constraint by manipulating u_1 and u_2.

Nominal Condition

At the nominal condition, the C2H4 availability ($u_1 u_2$) is `0.175`, and the reactor is operating at a steady state with plant inputs `uNom`.

```
uNom = [0.5; 0.35; 1.15];
```

To obtain the nominal operating point, compute the nominal plant states based on the given plant inputs using `fsolve` from the Optimization Toolbox.

```
options = optimoptions('fsolve','Display','none');
x0 = [0.9,0.4,0.1,1.1,0.3,0];   % Initial guess
xNom = fsolve(@(x) oxidationPlantCT(x,uNom),x0,options);
[dxNom,yNom] = oxidationPlantCT(xNom,uNom);
```

Verify that all time derivatives are close to zero at the nominal operating point.

```
norm(dxNom)
```

```
ans =

   8.8674e-09
```

Linear Plant Model

To obtain the nominal LTI plant model used by the MPC controller, linearize the ODEs by numerically perturbing the nominal states and inputs.

```
plant = linearizeOxidationPlant(@oxidationPlantCT,xNom,uNom);
```

Then, add C2H4 availability as an additional input (u_4). This input has no effect on the plant states or outputs. Instead, it is a measured disturbance used to define the equality constraint.

```
plant = ss(plant.A, [plant.B zeros(6,1)], plant.C, [plant.D zeros(2,1)]);
```

The nominal value of u_4 is $u_1 u_2$.

```
uNom = [uNom; uNom(1)*uNom(2)];
```

Therefore, the MPC controller uses four plant inputs. The first two are manipulated variables, and the last two are measured disturbances.

```
plant.InputName = {'Qin','CEin','Tref','Availability'};
plant = setmpcsignals(plant,'MV',[1 2],'MD',[3 4]);
```

Specify names for the six states and two measured outputs.

```
plant.StateName = {'Den','C2H4','C2H4O','Tc','Qout','PIx'};
plant.OutputName = {'C2H4O','Qout'};
```

Traditional MPC Design

To create an economic MPC controller, first design a traditional MPC controller. The relatively large sample time of 5 seconds used here is appropriate when the plant is stable and the primary objective is economic optimization.

```
Ts = 5;      % Sample time
P = 20;      % Prediction horizon
M = 5;       % Control horizon
mpcobj = mpc(plant,Ts,P,M);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defa
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
```

Specify the nominal operating conditions.

```
mpcobj.Model.Nominal.X = xNom;
mpcobj.Model.Nominal.U = uNom;
mpcobj.Model.Nominal.Y = yNom;
```

Plant inputs u_1 and u_2 must stay within saturation limits:

$$0.05 \le u_1 \le 0.7$$
$$0.1 \le u_2 \le 3$$

```
mpcobj.MV(1).Min = 0.05;
mpcobj.MV(1).Max = 0.7;
mpcobj.MV(2).Min = 0.1;
mpcobj.MV(2).Max = 3;
```

The rates of change of u_1 and u_2 are also limited:

$$-0.05 \le u_1(k) - u_1(k-1) \le 0.0.5$$
$$-0.2 \le u_2(k) - u_2(k-1) \le 0.2$$

```
mpcobj.MV(1).RateMin = -0.05;
mpcobj.MV(1).RateMax =  0.05;
mpcobj.MV(2).RateMin = -0.2;
mpcobj.MV(2).RateMax =  0.2;
```

Economic MPC Design with Custom Cost and Constraints

Instead of using the standard quadratic objective function, use a custom cost function that can represent any arbitrary nonlinear cost. The custom cost function must be named `mpcCustomCostFcn.m`. The template `mpcCustomCostFcn.txt` provides a basis for customization. For details, see the comments in `mpcCustomCostFcn.m`.

In this example, the custom cost function is the C2H4O production rate time-averaged over the prediction horizon:

```
f = -sum(y(:,1).*y(:,2))/P;
```

where `P` is the MPC prediction horizon. The negative sign in `f` is used to maximize production, since the controller minimizes `f` during optimization. The custom cost function replaces the standard quadratic cost used in traditional MPC.

To force ethylene consumption $u_1 u_2$ to equal C2H4 availability, use a custom constraint function. C2H4 availability is the second measured disturbance. The controller passes the measured disturbances to the custom constraint function in argument `v`. For this example, the second measured disturbance, `v(:,2)`, corresponds to plant input u_4. Apply this constraint at each step of the prediction horizon, producing `P` nonlinear equality constraints.

```
Ceq = u(:,1).*u(:,2) - v(1:end-1,2);
```

The custom constraint function must be named `mpcCustomConstraintFcn.m`. The template `mpcCustomConstraint.txt` provides a basis for customization. For details, see the comments in `mpcCustomConstraint.m`. The constraints in the custom constraint function are used in addition to linear constraints defined in the MPC controller.

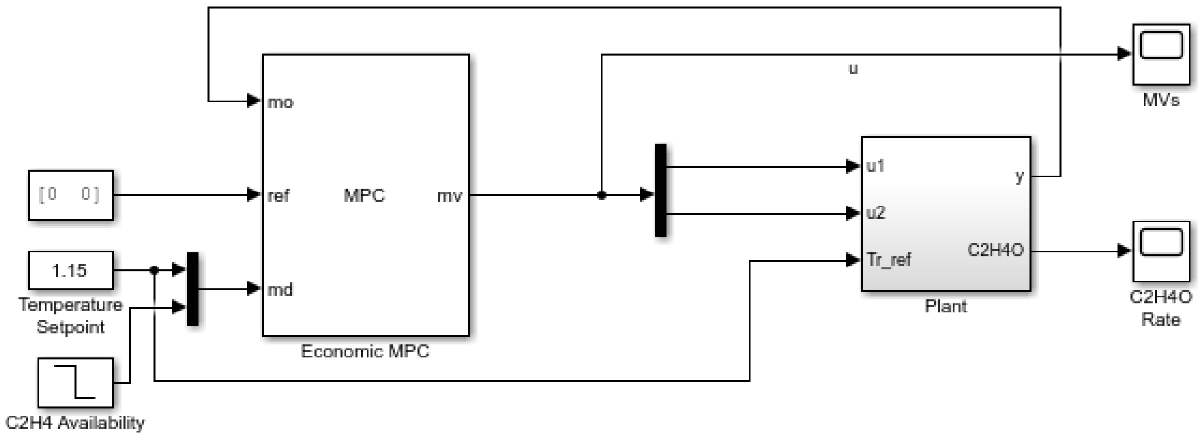
To create an economic MPC controller, enable the traditional MPC controller to use custom cost and constraint functions.

```
mpcobj.Optimizer.CustomCostFcn = true;
mpcobj.Optimizer.CustomConstraintFcn = true;
```

Simulink Model with Economic MPC Controller

Open the Simulink model.

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



C2H4 availability is initially `0.175` and remains constant for the first 500 seconds, which allows the controller to drive the plant to a new steady state with higher C2H4O production. C2H4 availability then steps down to `0.15` at t = 500 and remains constant for another 500 seconds. The economic MPC controller again maximizes C2H4O production at this condition.

The model includes constant (zero) references for the two plant outputs. The MPC Controller block requires these reference signals, but they are ignored in the custom cost function. The reactor temperature setpoint remains constant at its nominal value of `1.15`.

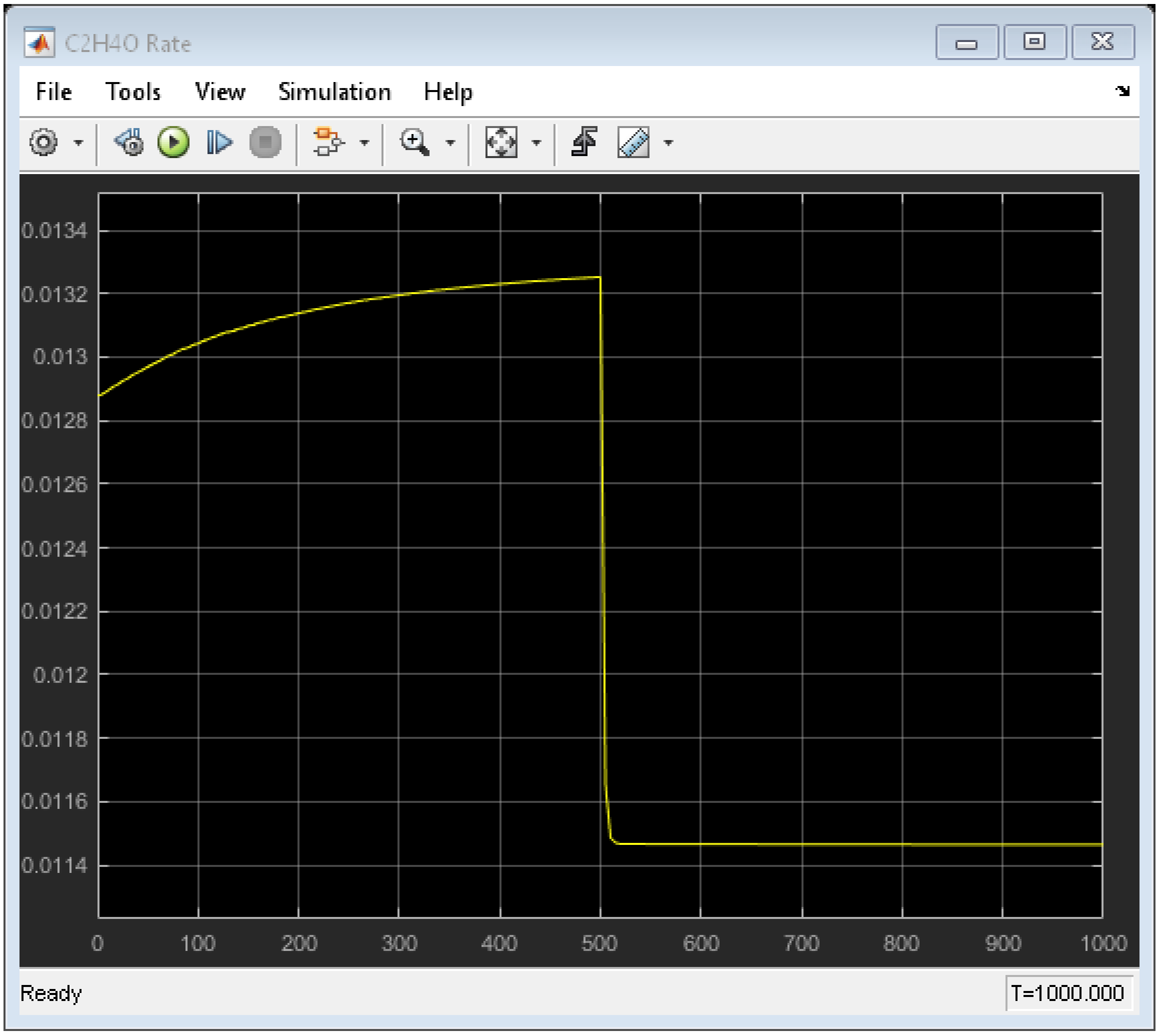
The Plant subsystem calculates the six plant states by integrating the ODEs in `oxidationPlantCT.m`. Two measurements are fed back to the MPC Controller block as measured outputs. The `C2H4O` plant output is the instantaneous C2H4O production rate, which is used for display purposes.

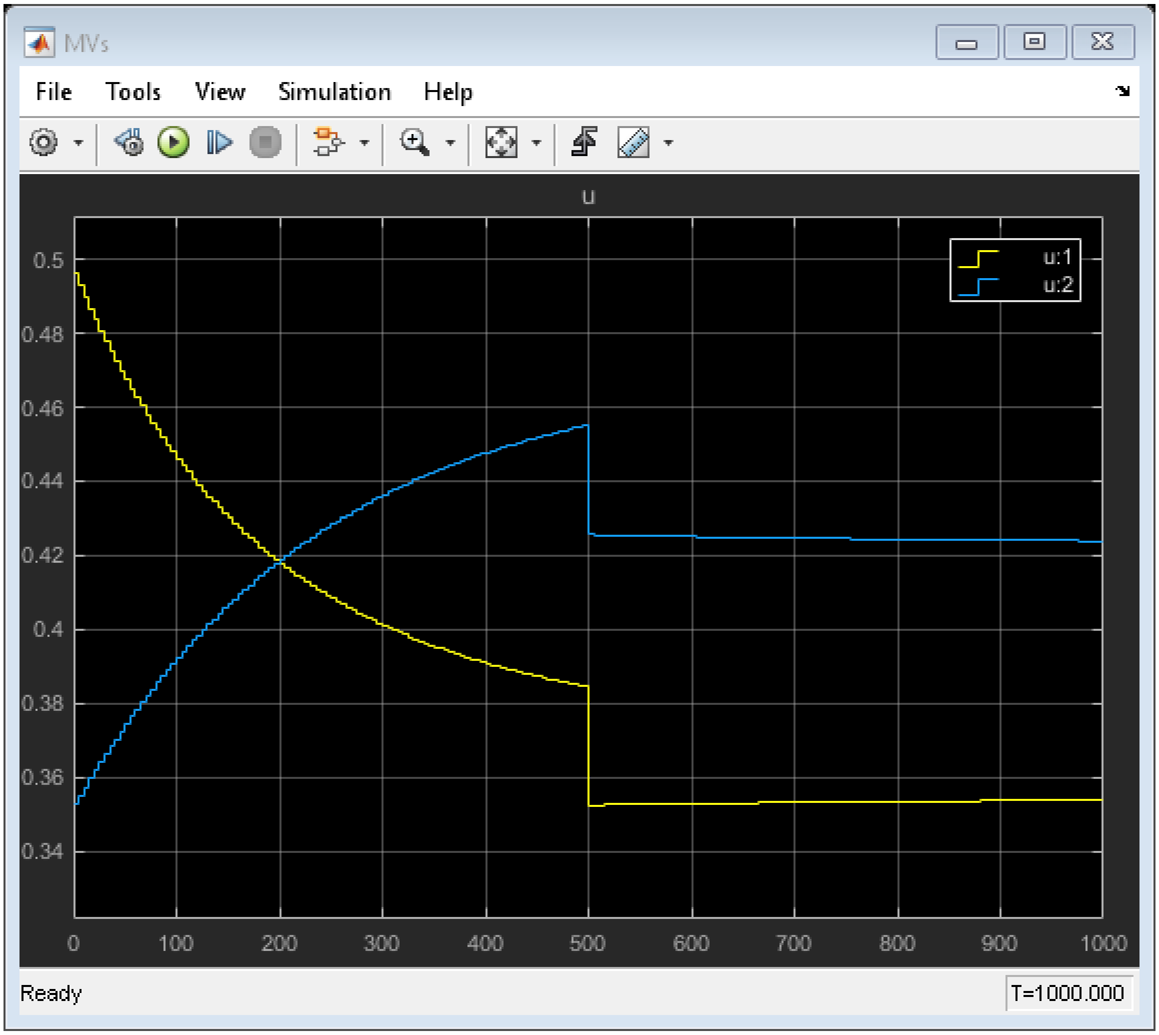
Simulate Model and Analyze Results

Run the simulation.

```
sim(mdl)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```





In this example, you are able to calculate the steady-state u_1 and u_2 that maximize the C2H4O production rate ($y_1 y_2$) for a given C2H4 availability (u_4) and reactor temperature (u_3).

The true optimum steady states are:

- When $u_4 = 0.175$:

$$u_1 = 0.205$$
$$u_2 = 0.854$$
$$y_1 y_2 = 0.0137$$

- When $u_4 = 0.150$:

$$u_1 = 0.245$$
$$u_2 = 0.613$$
$$y_1 y_2 = 0.0116$$

Recall that at the initial condition:

$$u_1 = 0.5$$
$$u_2 = 0.469$$
$$y_1 y_2 = 0.0129$$

Therefore, the C2H4O plant operating at the nominal condition is not optimal, and its profit can be improved.

In the first 500 seconds, the economic MPC controller gradually moves the plant to a new steady state under the same C2H4 availability constraint:

$$u_1 = 0.373$$
$$u_2 = 0.469$$
$$y_1 y_2 = 0.0133$$

Therefore, the economic MPC controller improves C2H4O production rate by

$$(0.0133 - 0.0129)\ /0.0129 = 3.1\%$$

which could be worth millions of dollars per year in large-scale production.

The economic MPC controller does not drive the plant to the true optimum $y_1 y_2 = 0.0137$. This result is due to the controller predicting the influence of u_1 and u_2 on y_1 and y_2 using the LTI model obtained at the nominal condition. The plant behavior described by the ODEs is nonlinear, and therefore, the controller predictions are not perfect.

You can improve controller performance by updating the LTI plant model at run time when conditions change. Doing so requires an adaptive MPC controller with the same custom cost and constraints. Even with adaptive control, it is unlikely that economic MPC would drive a physical plant to operate at the true optimum because MPC optimizes based on an internal model, which is always an imperfect representation of the real plant.

In the second 500 seconds, the C2H4 availability drops from `0.175` to `0.15`. The economic MPC controller moves the plant smoothly to a new steady state:

$$u_1 = 0.355$$
$$u_2 = 0.423$$
$$y_1 y_2 = 0.01146$$

Again, the economic MPC controller brings the C2H4O production rate close to the true optimum `0.0116`.

Remove example file folder from MATLAB path, and close Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc_featured','main'));
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, pp 18-36, 2016.

See Also

More About

- “Economic MPC” on page 10-2

Automated Driving Applications

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. 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 1-10 • “Constraints on Linear Combinations of Inputs and Outputs” on page 2-32
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 1-6 • “MPC Modeling”

MPC Feature	Description	More Information
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"
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 5-2
Generate code	You can automatically generate code for deploying model predictive controllers.	"Generate Code and Deploy Controller to Real-Time Targets" on page 9-2

Simulation in Simulink

To simplify the initial development of automated driving controllers, Model Predictive Control Toolbox software provides Simulink blocks for adaptive cruise control and lane-keeping assistance. 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.

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 and Adaptive MPC Controller blocks. For an example that uses an adaptive model predictive controller, see “Obstacle Avoidance Using Adaptive Model Predictive Control” on page 5-34.

Controller Customization

For the Adaptive Cruise Control System and Lane Keeping Assist 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 **Blocks** 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 System Toolbox

If you have Automated Driving System 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 11-13 and “Lane Keeping Assist with Lane Detection” on page 11-42.

See Also

Blocks
Adaptive Cruise Control System | Adaptive MPC Controller | Lane Keeping Assist System | 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_featured','main'));
```

Adaptive Cruise Control System

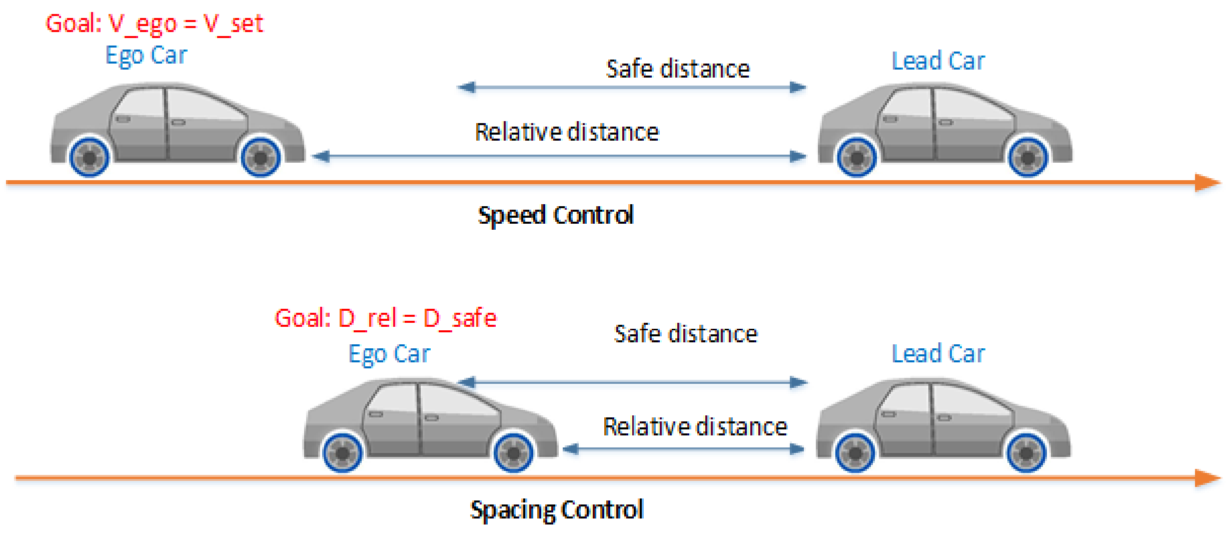
A vehicle (ego car) equipped with adaptive cruise control (ACC) has a sensor, such as radar, that measures the distance to the preceding vehicle in the same lane (lead car), D_{rel}, as well as the relative velocity of the lead car, V_{rel}. The ACC system operates in the following two modes:

- 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:

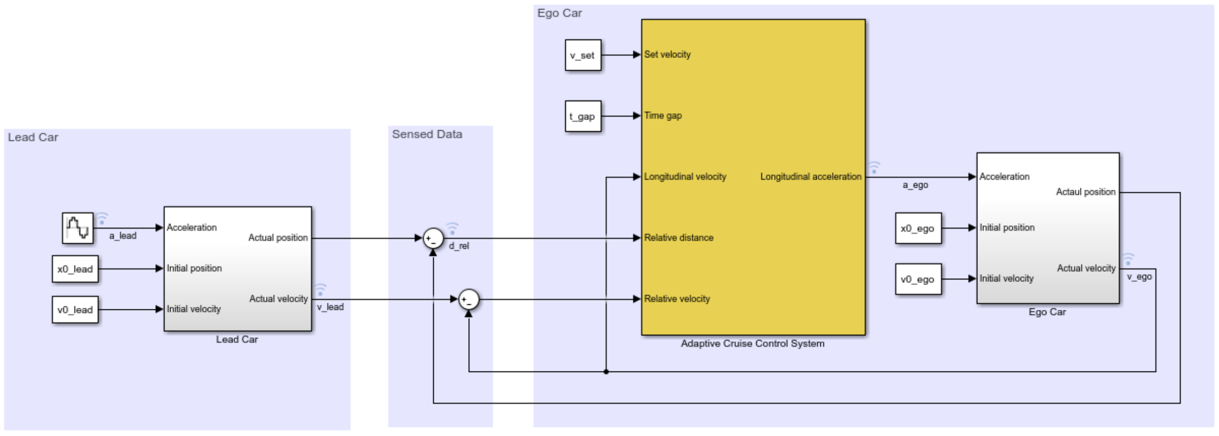
- If $D_{rel} \geq D_{safe}$, then speed control mode is active. The control goal is to track the driver-set velocity, V_{set}.
- If $D_{rel} < D_{safe}$, then spacing control mode is active. The control goal is to maintain the safe distance, D_{safe}.



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)
```



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

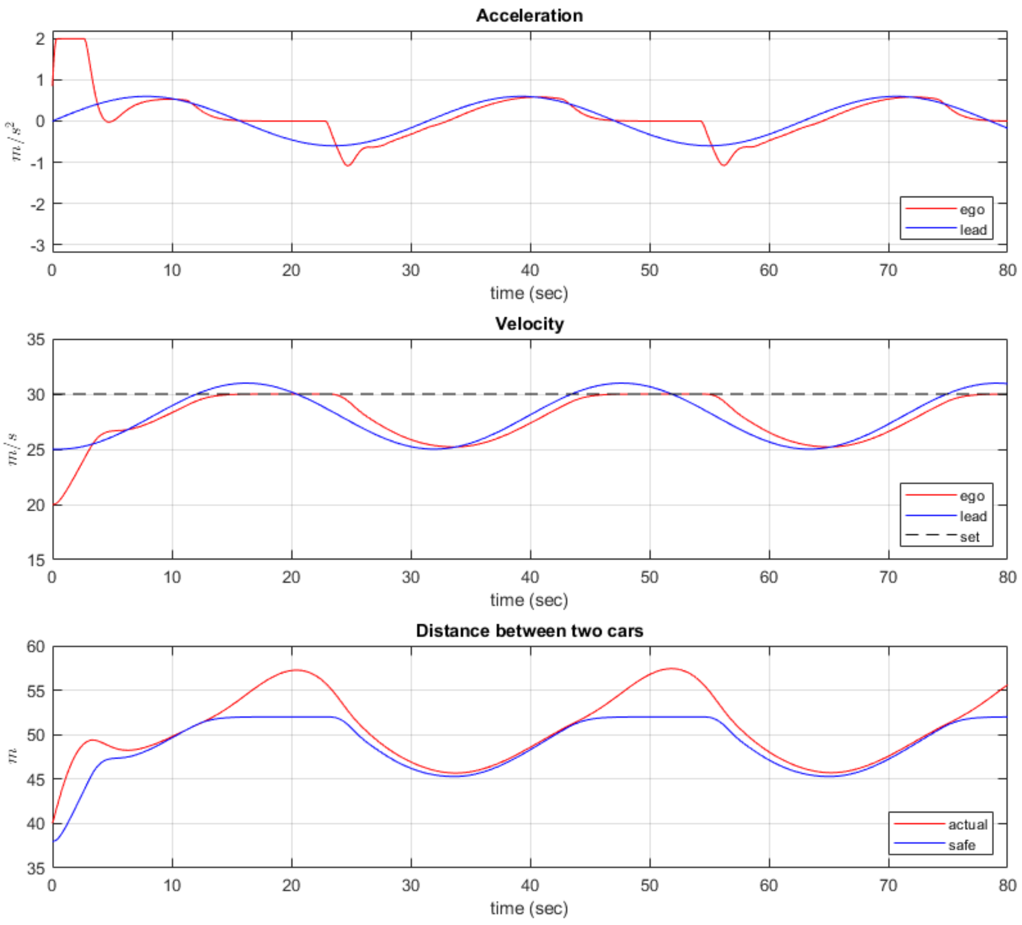
Run the simulation.

```
sim(mdl)
```

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

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_featured','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.

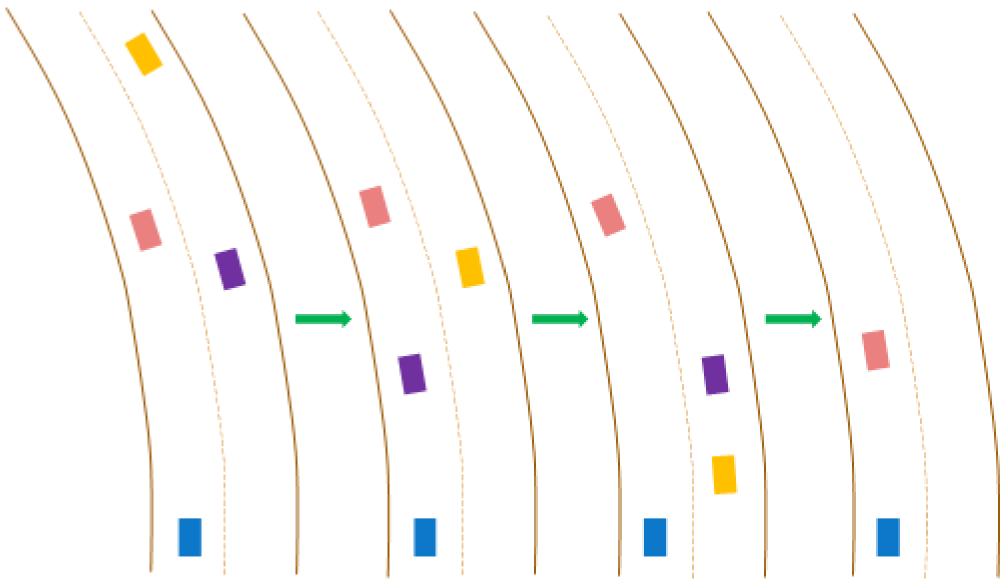
In this example, you will:

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 System 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 car 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 car down if there is another vehicle moving slower in the lane in front of it.

For the ACC to work correctly, the ego car has to determine how the lane in front of it curves, and which car is the 'lead car', that is, in front of the ego car in the lane. A typical scenario from the viewpoint of the ego car is shown in the figure below. The ego car (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 car 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 afterwards. 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” and in “Automotive Adaptive Cruise Control Using FMCW Technology” (Phased Array System Toolbox). Moving from 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 car and that have significant lateral motion relative to the ego car.
2. Estimating the lane ahead of the ego car to find which car in front of the ego car 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 car lane.

This example demonstrates two main additions to existing ACC designs that meet the challenges listed above: 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 provide 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.

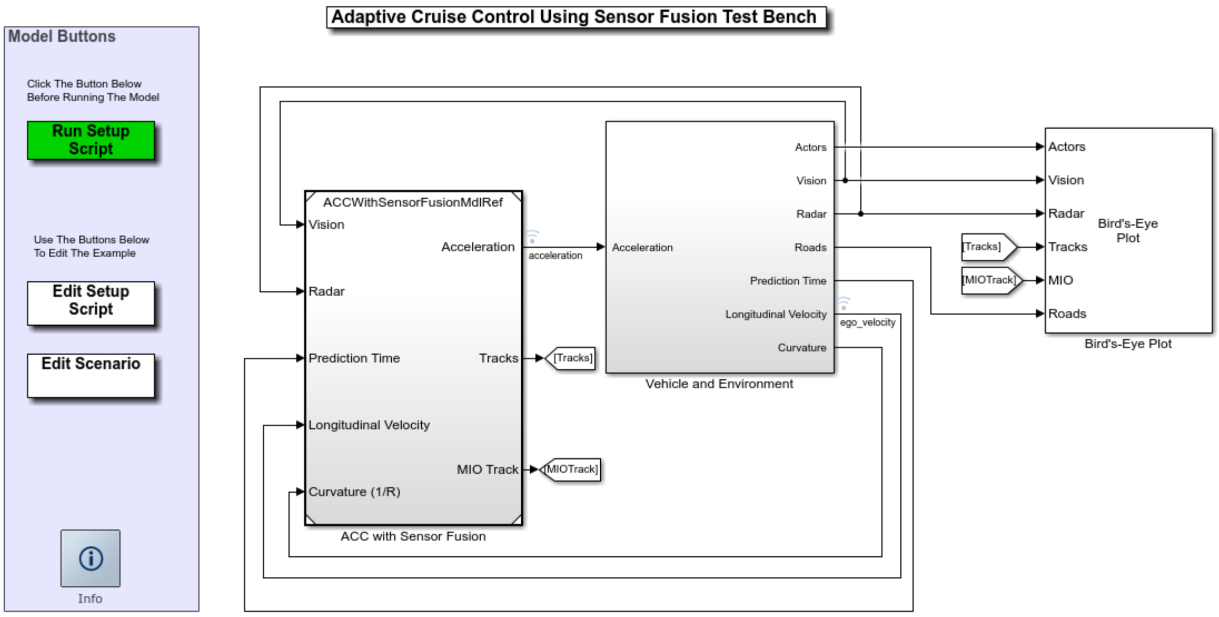
2. A vision sensor can detect lanes, provide an estimate of the lateral position of the lane relative to the ego car, and position the other cars in the scene relative to the ego lane. In this example, we consider an ideal lane detection.

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 car 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 the Test Bench Model and Simulation Results

You use the following command to open the main Simulink model:

```
open_system('ACCTestBenchExample')
```



The model contains three main components:

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 car and models the environment. A simulation of radar and vision sensors provides synthetic data to the control subsystem.
3. A Bird's-Eye Plot display, which plots the results of the simulation and depicts the ego car's surrounding and tracked objects.

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 vehicle 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 prior to model compilation. When the model compiles, additional Simulink buses are automatically generated by their respective blocks.

The following commands run the simulation to 15 seconds to snap 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 to snap
snapnow
sim('ACCTestBenchExample'); %Simulate to end of scenario
```

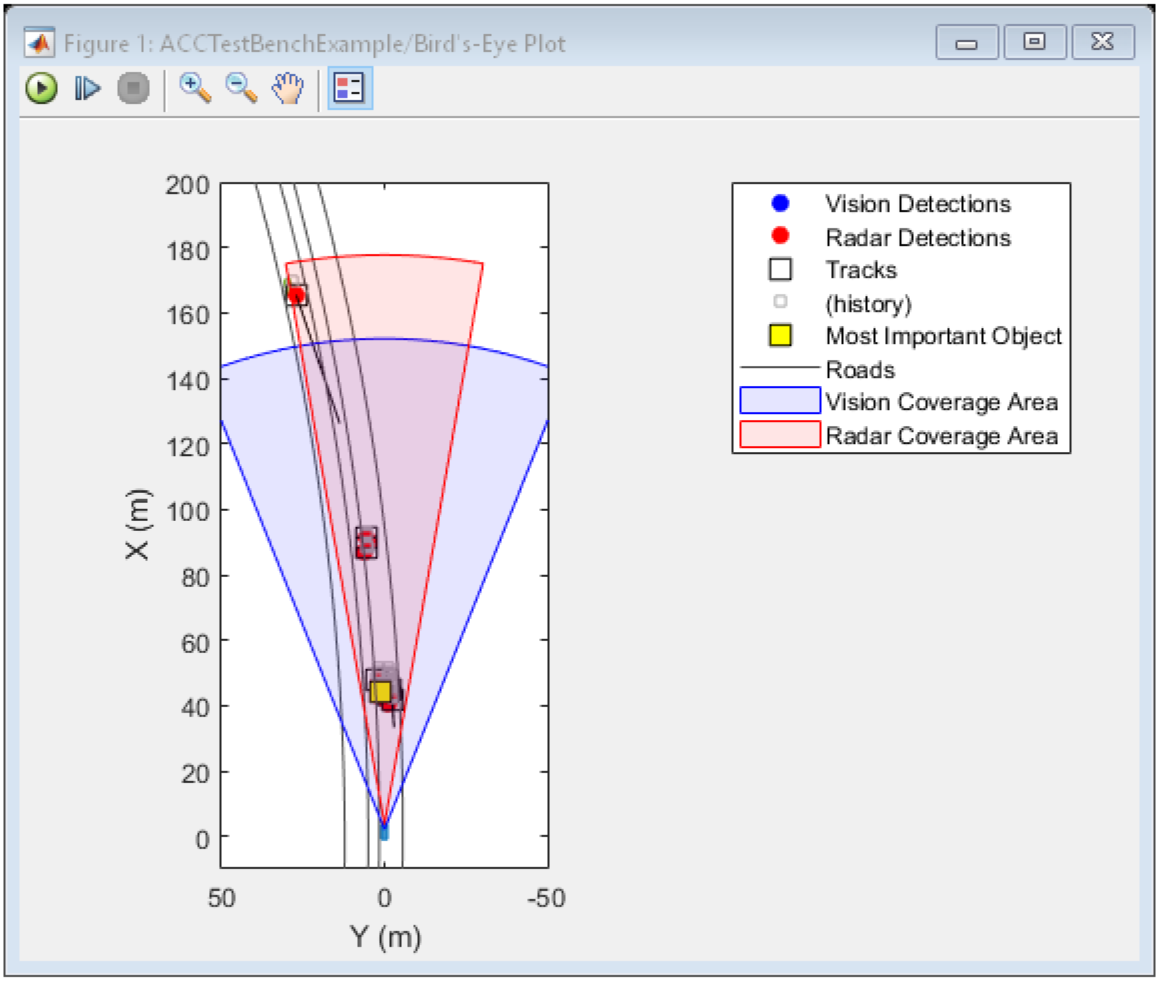
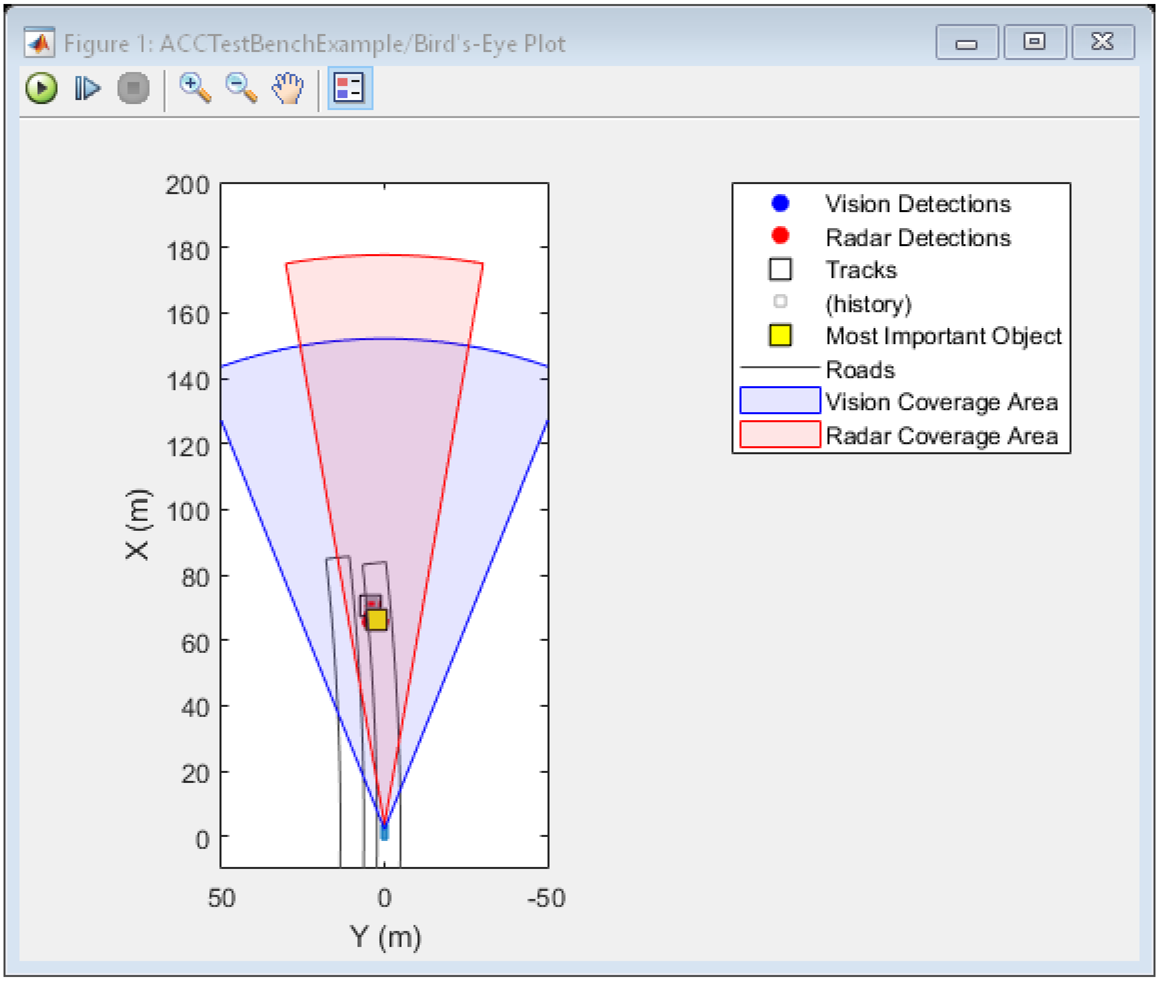


Figure 1: ACCTestBenchExample/Bird's-Eye Plot
200
180
160
140
120
100
80
60
40
20
0
X (m)
50
0
-50
Y (m)
Vision Detections
Radar Detections
Tracks
(history)
Most Important Object
Roads
Vision Coverage Area
Radar Coverage Area



The bird's-eye plot 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 car 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 car. 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 car and much slower than it. Thus, the ACC must slow the ego car down.

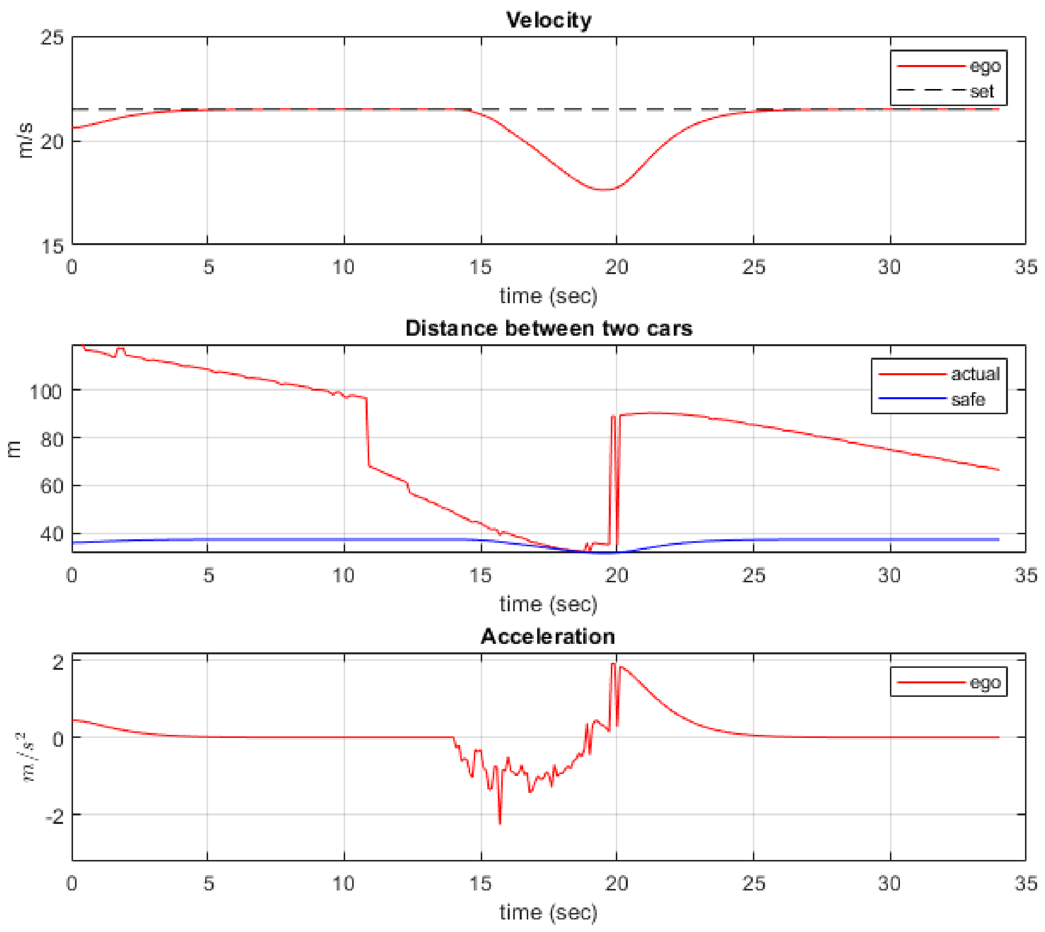
In the following results for the classical ACC system, the:

- Top plot shows the ego car velocity.
- Middle plot shows the relative distance between the ego car and lead car.
- Bottom plot shows the ego car 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 car 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 car (middle plot). The ego car accelerates and reaches the driver-set velocity V_set (top plot).
- Another car becomes the lead car from 11 to 20 seconds when the car cuts into the ego car lane (middle plot). When the distance between the lead car and the ego car is large (11-15 seconds), the ego car still travels at the driver-set velocity. When the

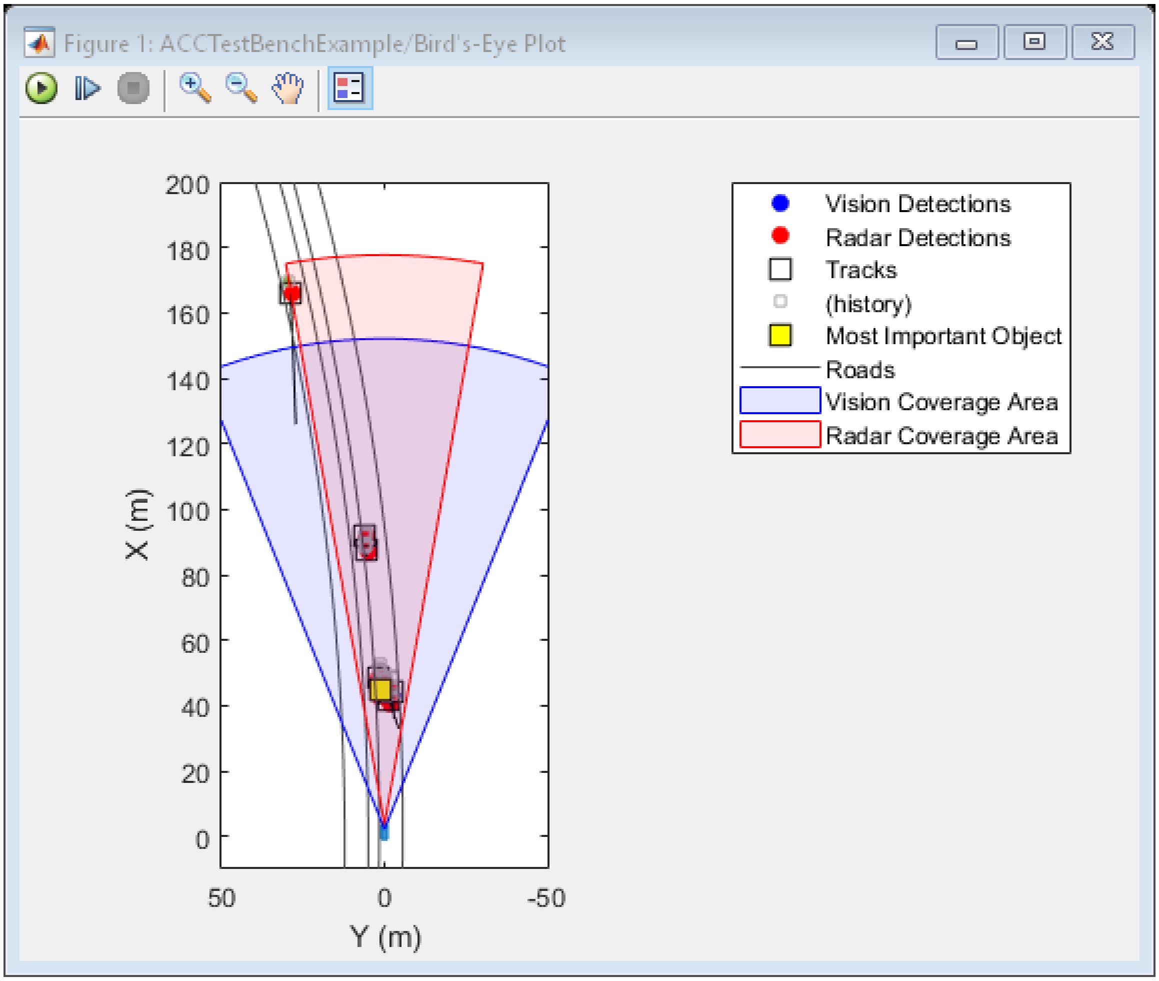
distance becomes small (15-20 seconds), the ego car decelerates to maintain a safe distance from the lead car (top plot).

- 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 car is large, the ego car accelerates until it reaches the driver-set velocity at 27 seconds. After this, the ego car 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.

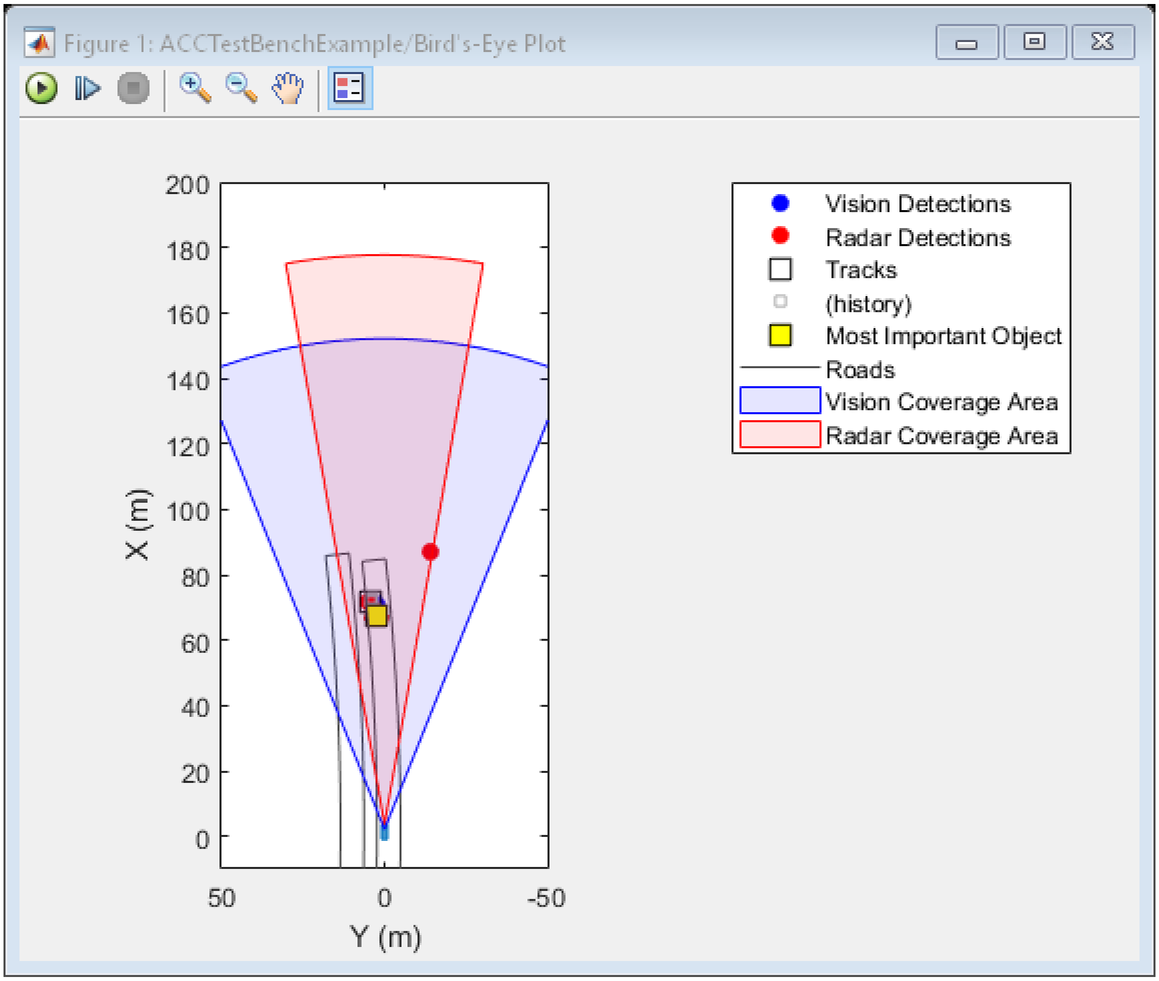
In the MPC-based ACC design, the underlying optimization problem is formulated by tracking the driver-set velocity subject to enforcing a safe distance from the lead car. The MPC controller design is described in the Adaptive Cruise Controller section. To run the model with the MPC design, first activate 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 to snap
    snapnow
    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
   Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



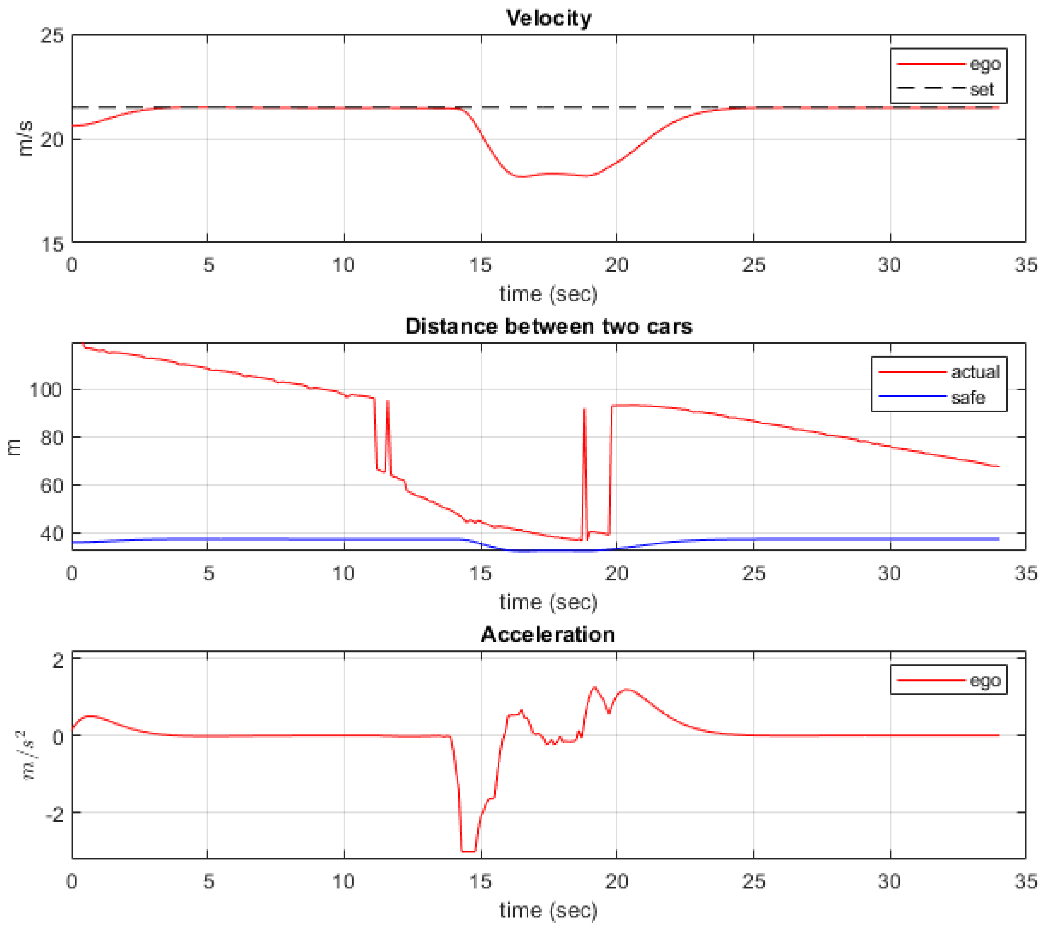
```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #2 is integrated white
   Assuming no disturbance added to measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



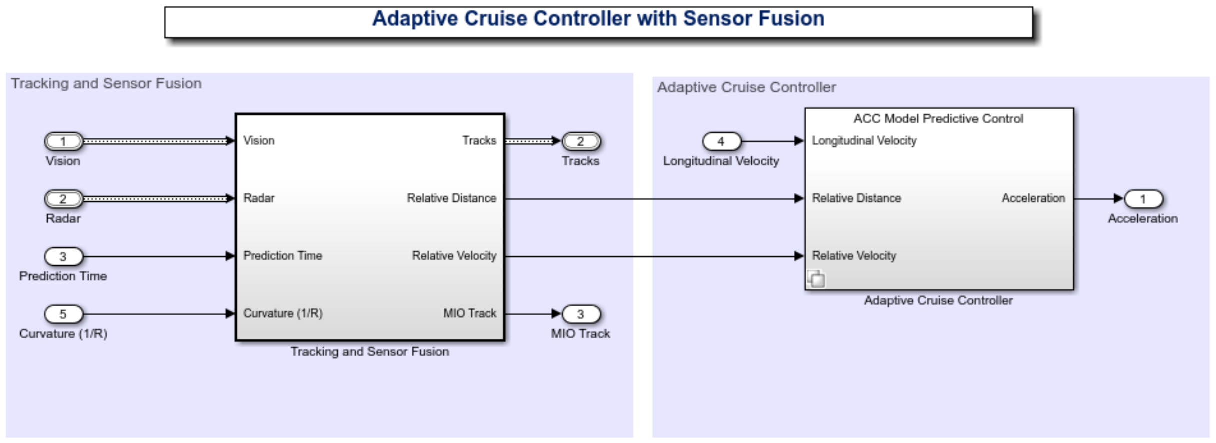
In the simulation results for the MPC-based ACC, similar to the classical ACC design, the objectives of speed and spacing control are achieved. Compared to the classical ACC design, the MPC-based ACC is more aggressive as it uses full throttle or braking for acceleration or deceleration. This behavior 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" slider. As noted above, 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);
```

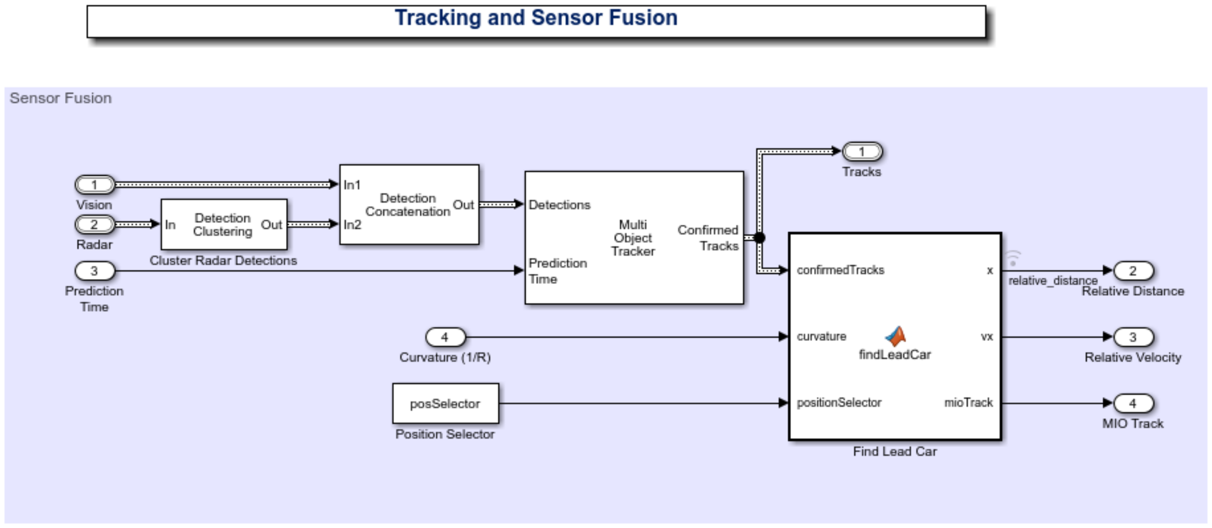


In the following, the functions of each subsystem in the Test Bench Model are described in more details. The ACC with Sensor Fusion subsystem. It contains two main parts: 1) Tracking and sensor fusion and 2) Adaptive cruise controller.



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 car. Also, it provides the ACC with an estimate of the closest car in the lane in front of the ego car.



The main block of the Tracking and Sensor Fusion subsystem is the Multi Object Tracker 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 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 car 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 car. The function provides the position and velocity of the lead car relative to the ego car as well as 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 car) uses sensor fusion to estimate the relative distance and relative velocity to the lead car. The ACC makes the ego car travel at a driver-set velocity while maintaining a safe distance from the lead car. The safe distance between lead car and ego car is defined as

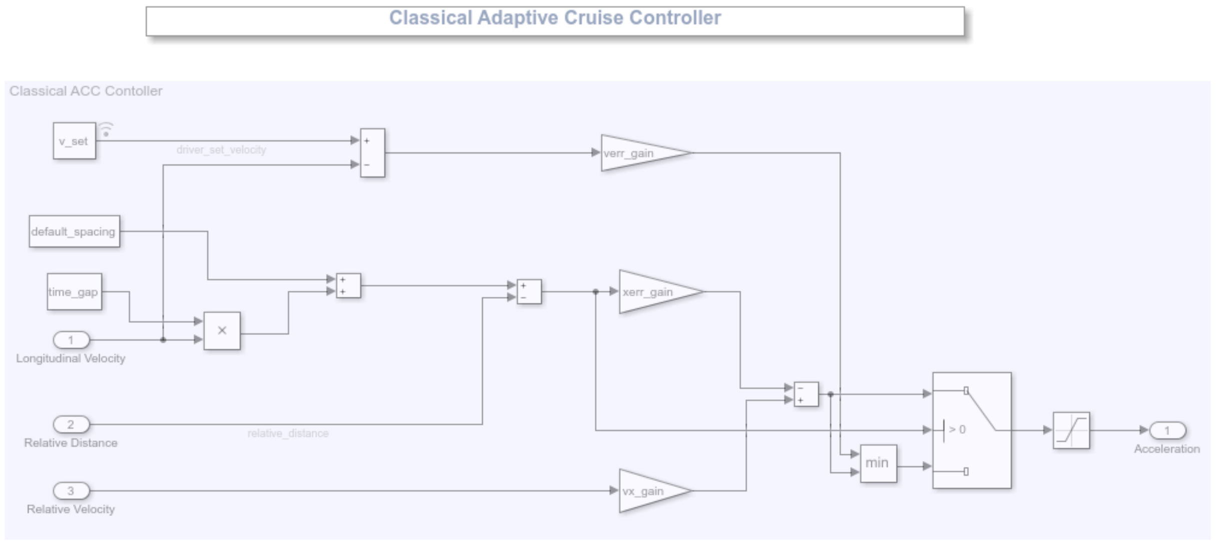
$$D_{safe} = D_{default} + T_{gap} \cdot V_x$$

where the default spacing $D_{default}$, and time gap T_{gap} are design parameters and V_x is the longitudinal velocity of the ego car. The ACC generates the longitudinal acceleration for the ego car based on the following inputs:

- Longitudinal velocity of ego car
- Relative distance between lead car and ego car (from the Tracking and Sensor Fusion system)
- Relative velocity between lead car and ego car (from the Tracking and Sensor Fusion system)

Considering the physical limitations of the ego car, the longitudinal acceleration is constrained to the range [-3,2] m/s^2.

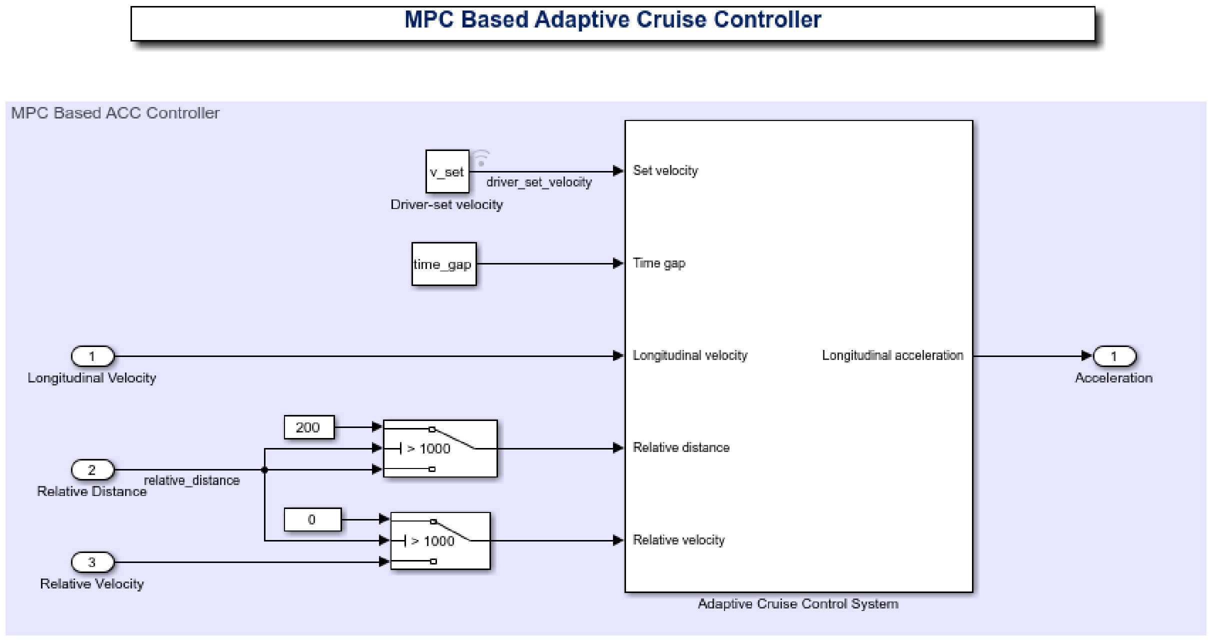
In the classical ACC design, if the relative distance is less than the safe distance, then the primary goal is to slow down and maintain a safe distance. If the relative distance is greater than the safe distance, then the primary goal is to reach driver-set velocity while maintaining a safe distance. These design principles are achieved through the Min and Switch Blocks.



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.

$$\begin{array}{ll} \underset{u}{\text{minimize}} & |V - V_{set}|^2 \\ \text{subject to} & D_{relative} - D_{safe} \geq 0 \\ & -3 \leq u \leq 2 \end{array}$$

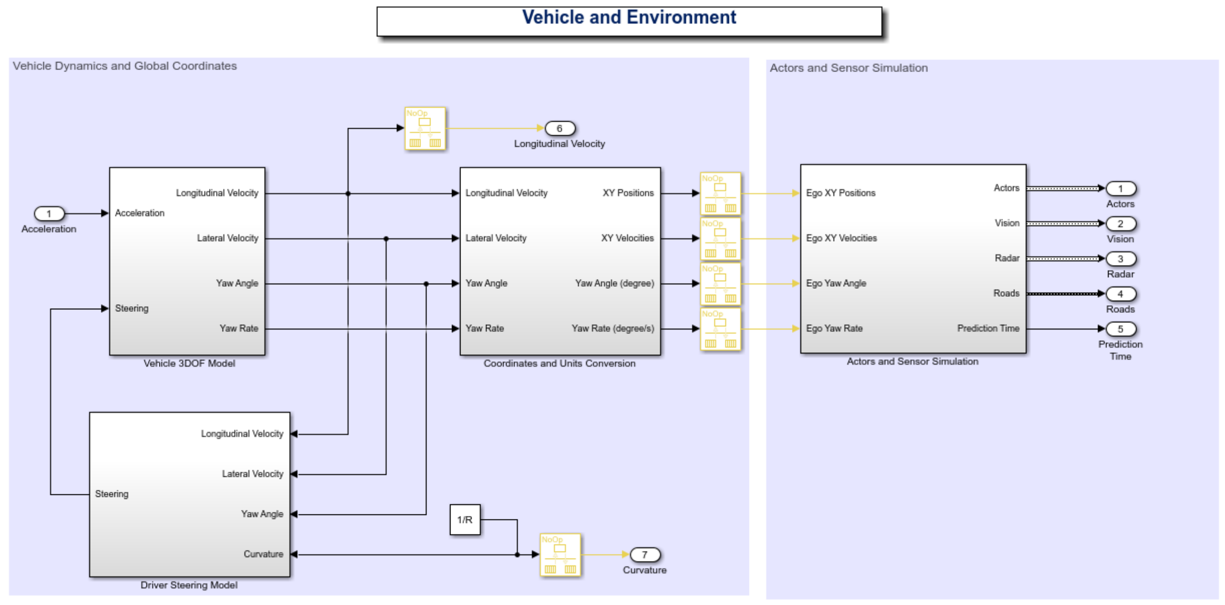
To configure the Adaptive Cruise Control System block, use the parameters defined in the `helperACCSetUp` file. For example, the linear model for ACC design G, and G is obtained from vehicle dynamics. The two Switch blocks implement simple logic to handle large numbers from the sensor (for example, the sensor may return `Inf` when it does not detect an MIO).



For more information on MPC design for ACC, see “Adaptive Cruise Control System Using Model Predictive Control”.

Vehicle and Environment

The Vehicle and Environment subsystem is comprised of two parts: (1) Vehicle Dynamics and Global Coordinates and (2) Actor and Sensor Simulation.



The vehicle dynamical model applies the ''bicycle mode'' of lateral vehicle dynamics and approximates the longitudinal dynamics using a time constant τ. The vehicle dynamics, with input u (longitudinal acceleration) and front steering angle δ, are:

$$\frac{d}{dt}\begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} = \begin{bmatrix} -\frac{2C_f+2C_r}{mV_x} & 0 & -V_x-\frac{2C_f\ell_f-2C_r\ell_r}{mV_x} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -\frac{2C_f\ell_f-2C_r\ell_r}{I_z V_x} & 0 & -\frac{2C_f\ell_f^2+2C_r\ell_r^2}{I_z V_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} V_y \\ \psi \\ \dot{\psi} \\ Vx \\ \dot{V}_x \end{bmatrix} + \begin{bmatrix} \frac{2C_f}{m} \\ 0 \\ \frac{2C_f\ell_f}{I_z} \\ 0 \\ 0 \end{bmatrix}\delta + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{\tau} \end{bmatrix} u + \begin{bmatrix} 0 \\ 0 \\ 0 \\ V_y\dot{\psi} \\ 0 \end{bmatrix}$$

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:

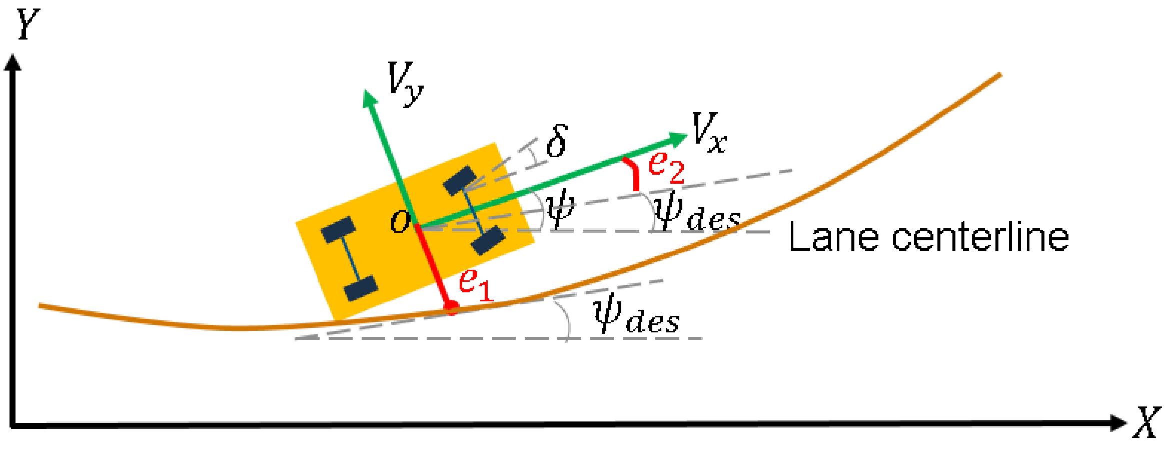
$$\dot{X} = V_x \cos(\psi) - V_y \sin(\psi), \quad \dot{Y} = V_x \sin(\psi) + V_y \cos(\psi)$$

The yaw angle ψ and yaw angle rate $\dot{\psi}$ are also converted into the units of degrees.

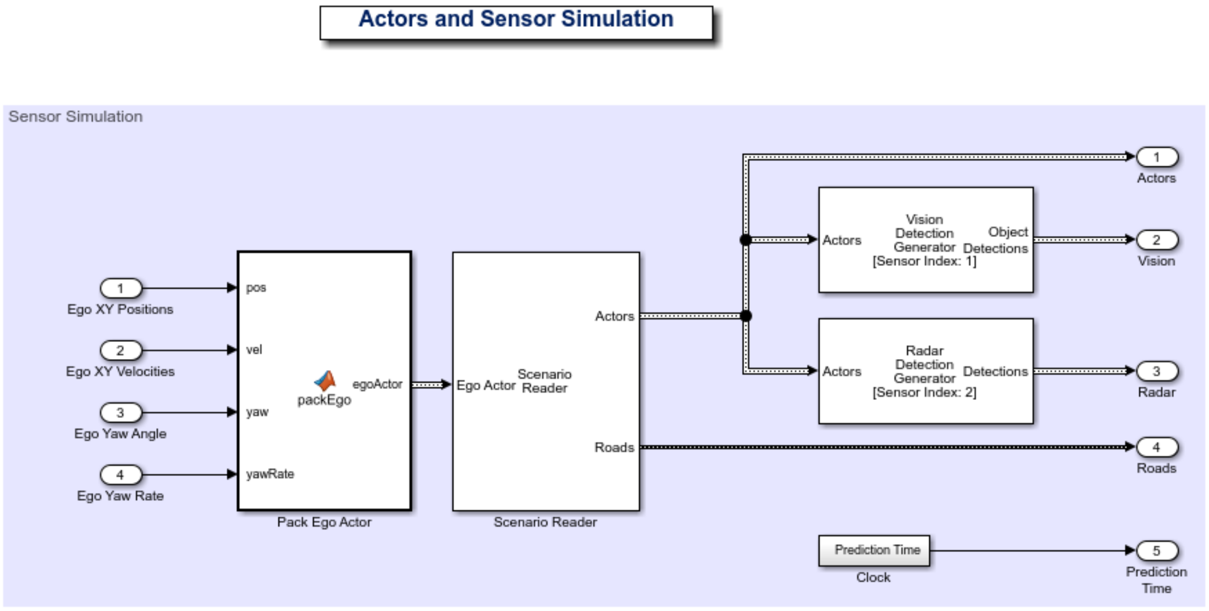
The goal for the driver steering model is to keep the vehicle in its lane and follow the curved road by controlling the front steering angle δ. This goal is achieved by driving the yaw angle error e_2 and lateral displacement error e_1 to zero (see the figure below), where

$$\dot{e}_1 = V_x e_2 + V_y, \quad e_2 = \psi - \psi_{des}$$

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. Prior to running this example, the `drivingScenario` function was used to create a simulation environment with a curved road and multiple actors moving on the road. The scenario was saved to a file. To see how you can define the scenario, see the Scenario Authoring section.



The motion of the ego car is controlled by the control system and is not recorded as part of the recorded scenario. Instead, the ego car 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 block reads the actor pose data and the road boundary data from the file that was used to save the scenario. The block converts both the actor poses and the road boundaries from scenario coordinates to the ego car coordinates. The actor poses are streamed on a bus generated by the block. In this example, you use a Vision Detection Generator block and Radar Detection Generator block. Both sensors are long-range and forward-looking, and provide good coverage of the front of the ego car, as needed for ACC. The sensors use the actor poses in ego coordinates to generate lists of detections of the vehicles in front of the ego car. 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 Authoring

The scenario was authored using the `helperScenarioAuthoring` function. To open the function, click on the **Edit Scenario** in the main model or run the following command to open the function:

```
edit helperScenarioAuthoring
```

The function creates a `drivingScenario`. The driving scenario 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.

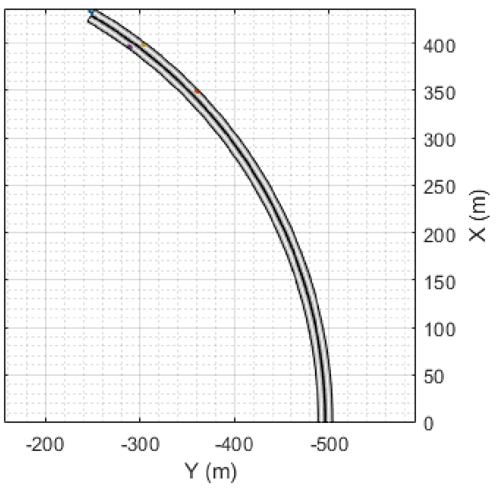
You define all the other vehicles in the scenario, excluding the ego car to be controlled by the model. To define the motion of the vehicles, you define their path by a set of waypoints and speeds. A quick way to define the waypoints is by choosing a subset of the road centers defined earlier, with an offset to the left or right of the road centers to control the lane in which the vehicles travel.

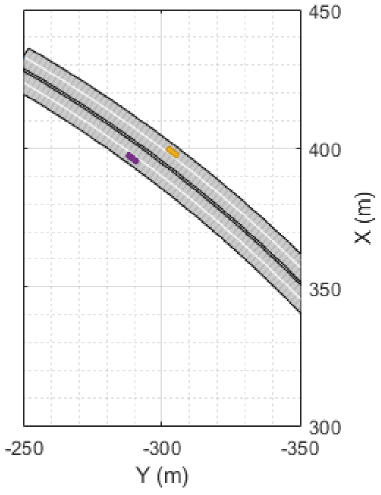
This example shows four vehicles: a fast-moving car in the left lane, a slow-moving car in the right lane, a car approaching on the opposite side of the road, and a car that starts on the right lane, but then moves to the left lane to pass the slow-moving car.

Finally, the driving scenario must be saved to a file. The function uses the `record` method to obtain the actor poses relative to the scenario coordinates at every sample time. The function obtains the road boundaries as a cell array using the `roadBoundaries` function. Due to code generation limitations, it then converts the cell array to a structure to allow it to be loaded by the Scenario Reader block.

This `helperScenarioAuthoring` allows you to:

1. Modify the road radius of curvature, which is the first input to the function. The default is `760` meters.
2. Modify the file name used when saving the driving scenario. The default is `'scenario'` .
3. Visualize the scenario by setting the third input to the helper function to `true` . The default is `false` . You can edit the helper function and create your own scenarios.





Generating Code for the Control Algorithm

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 System Toolbox, componentize it, and automatically generate code for it.

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.

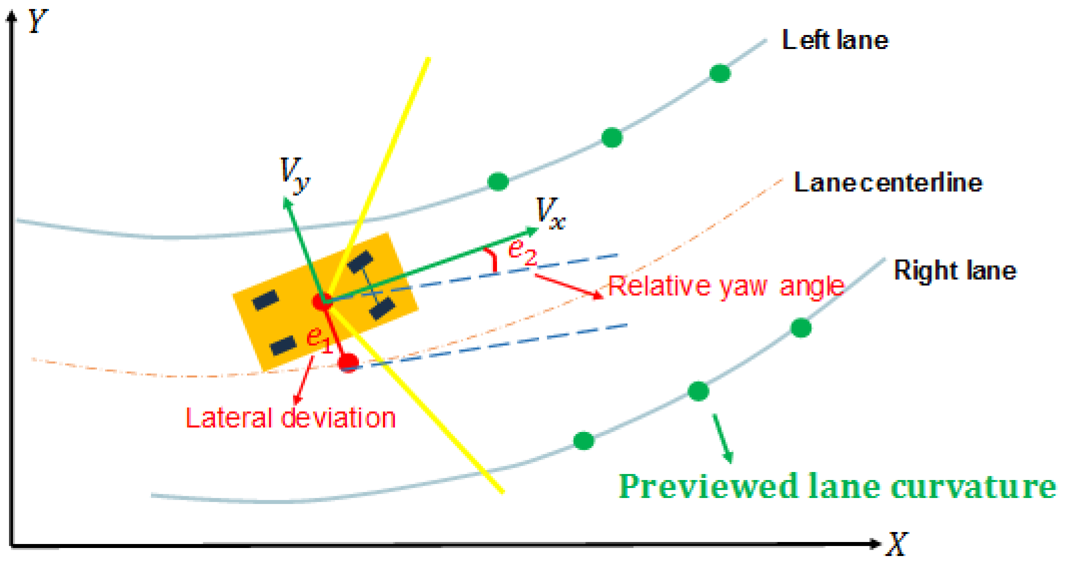
Add example file folder to MATLAB® path.

```
addpath(fullfile(matlabroot,'examples','mpc_featured','main'));
```

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.

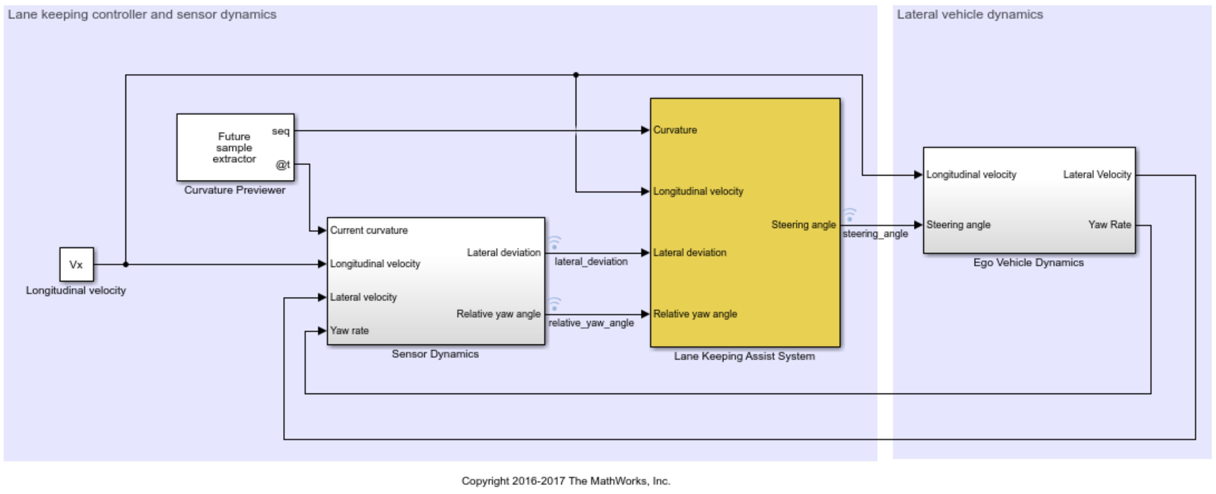
The LKA system keeps the ego car travelling along the centerline of the lanes on the road by adjusting the front steering angle of the ego car. The goal for lane keeping control is to drive both lateral deviation and relative yaw angle close to zero.



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 (mNs^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 δ
- 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 and 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 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, then update the block parameters accordingly.

Simulation Analysis

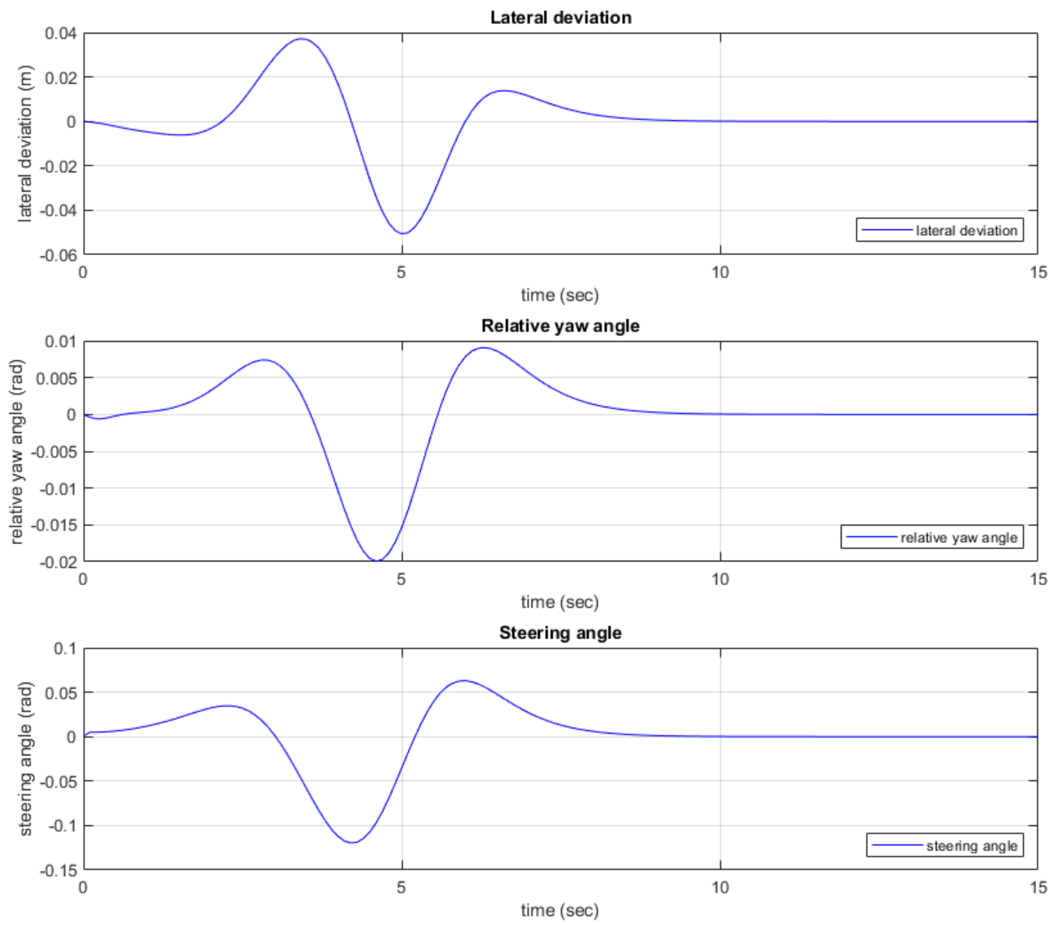
Run the model.

```
sim(mdl)
```

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

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.

Remove example file folder from MATLAB path, and close Simulink model.

```
rmpath(fullfile(matlabroot,'examples','mpc_featured','main'));
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

This example shows how to simulate and generate code for an automotive lane keeping assist (LKA) controller.

In this example, you will:

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 System 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 adjust 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 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.

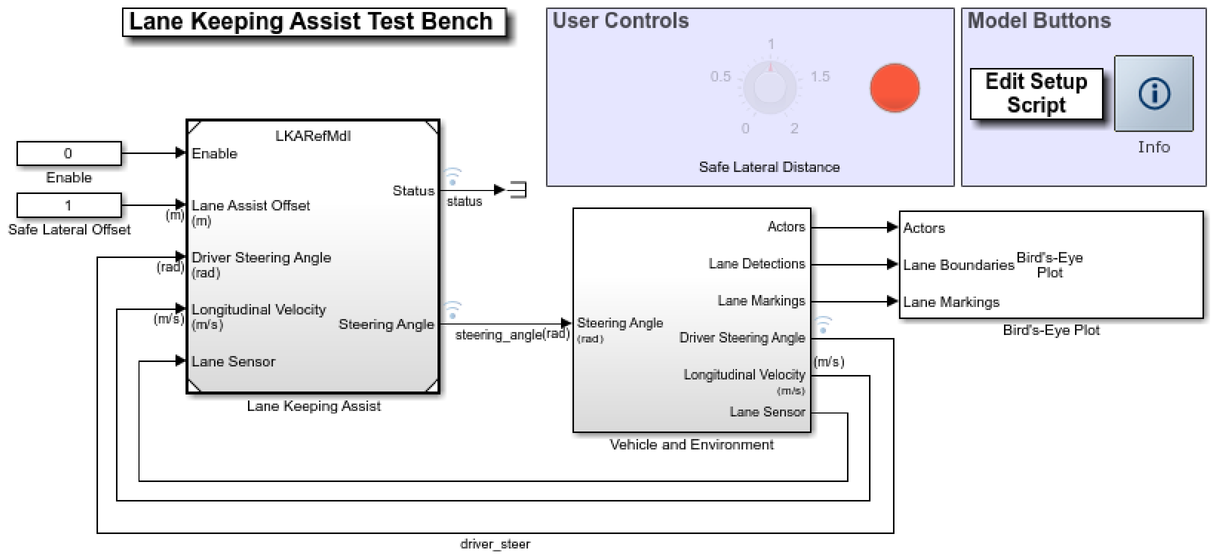
For the LKA to work correctly, the ego car must determine the lane boundaries and how the lane in front of it curves. Idealized LKA designs rely mostly on the previewed curvature, the lateral deviation and relative yaw angle between the centerline of the lane and the ego car. An example of such a system is given in “Lane Keeping Assist System Using Model Predictive Control”. Moving from 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 this, 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 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

You use the following command to open the Simulink test bench model.

```
open_system('LKATestBenchExample')
```

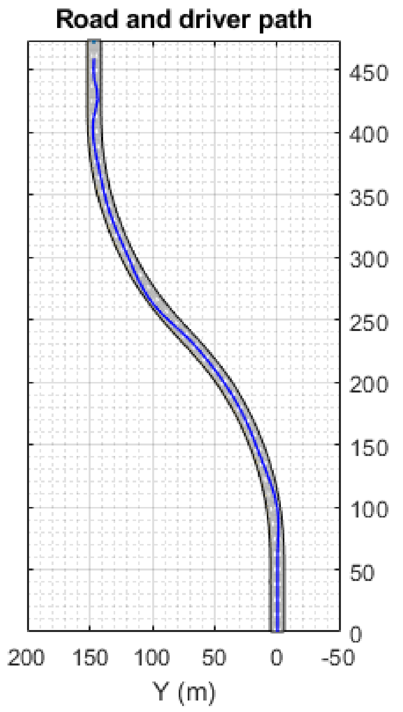


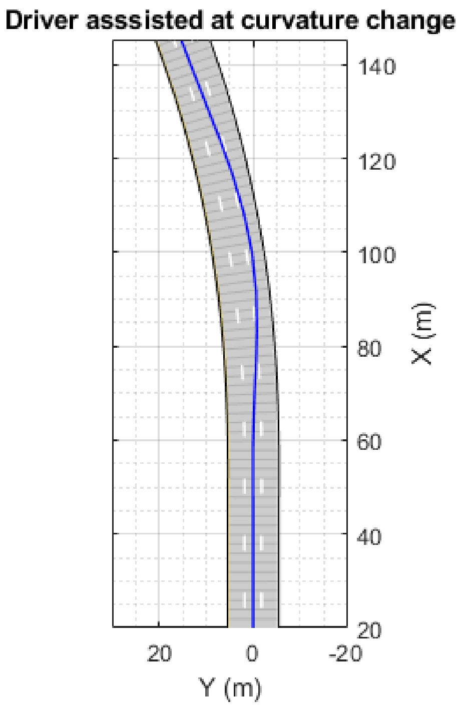
The model contains three main components:

1. Lane Keeping Assist, which controls the front steering angle of the vehicle.
2. A Vehicle and Environment subsystem, which models the motion of the ego car and models the environment.
3. A Bird's-Eye Plot display, which plots the results of the simulation.

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 using

```
plotLKAInputs(scenario, driverPath);
```





Simulate Assisting a Distracted Driver

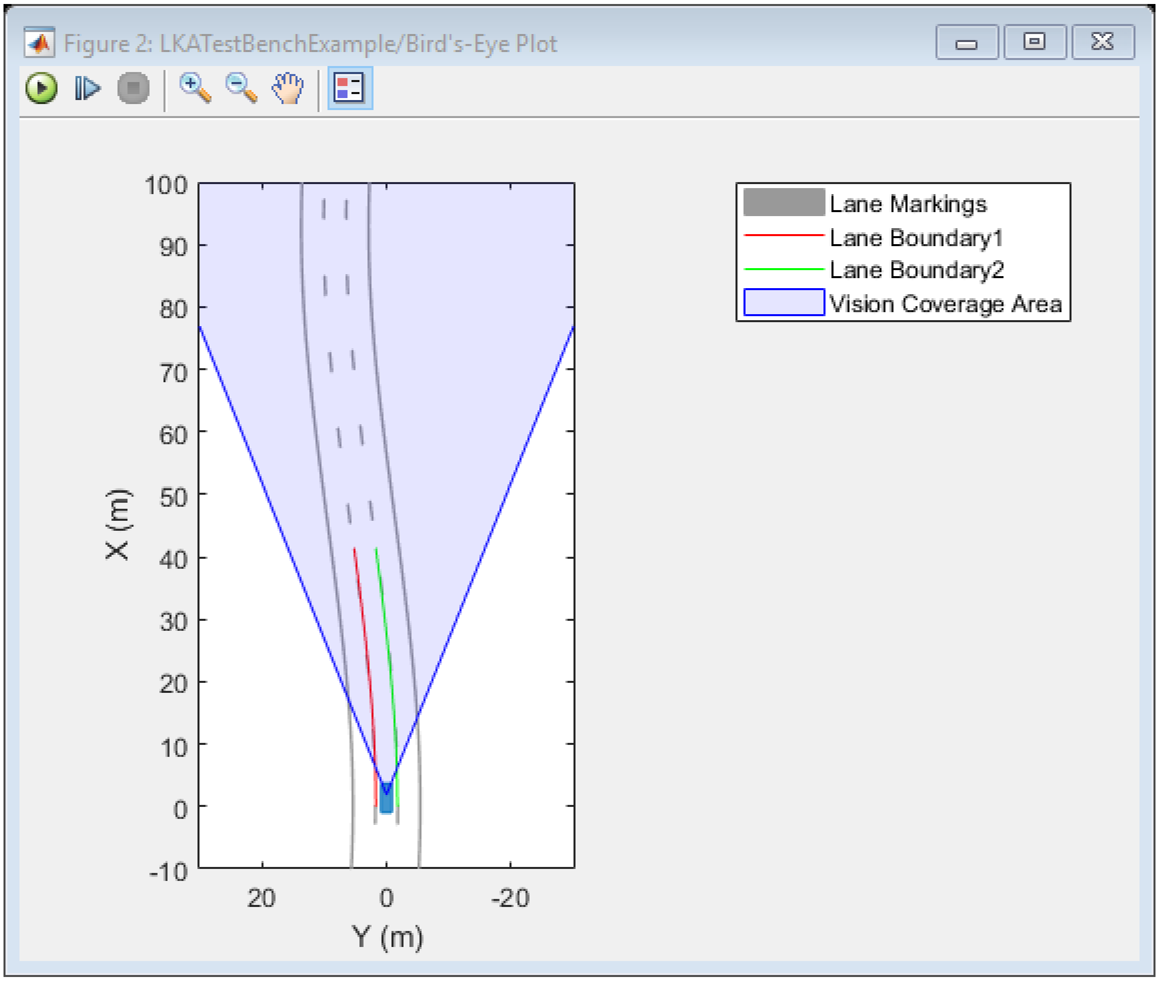
You can explore the behavior of the algorithm by enabling the "Enable Assist" slider and setting the "Safe Lateral Distance" to 1 meter. You can also accomplish this by running the following commands.

```
set_param('LKATestBenchExample/Enable','Value','1')
set_param('LKATestBenchExample/Safe Lateral Offset','Value','1');
```

You can run the simulation to 15 seconds to explore the contents of the Birds Eye Plot.

```
sim('LKATestBenchExample','StopTime','15'); % Simulate 15 seconds to snap
snapnow
```

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

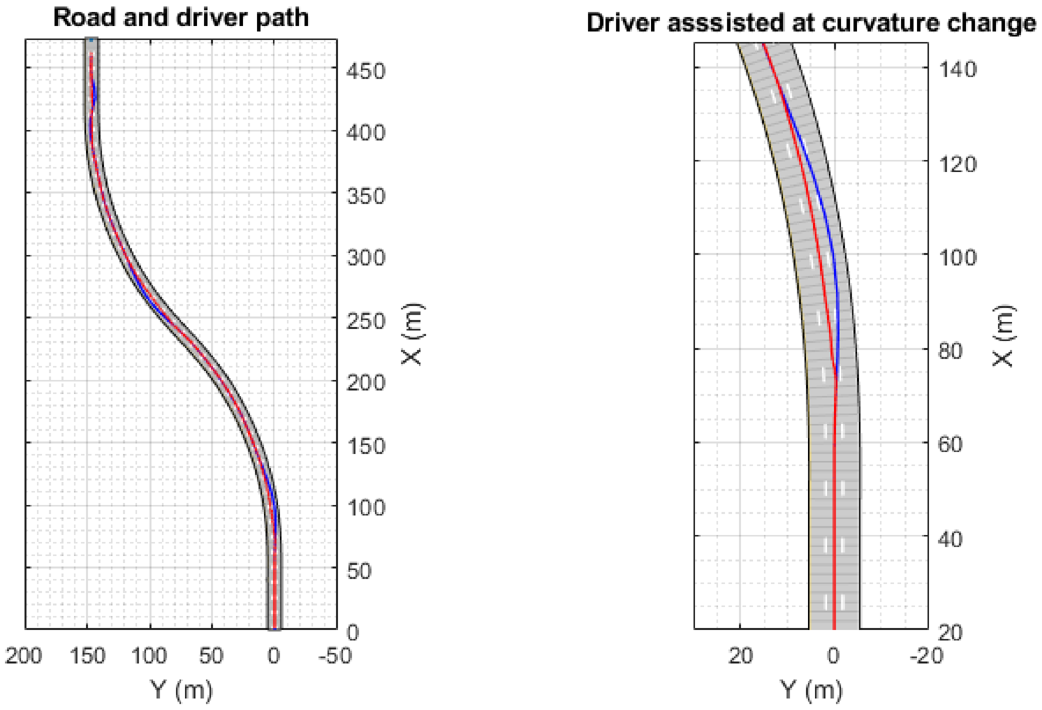


The bird's eye plot shows a symbolic representation of the road in the perspective of the ego car. In this example, the bird's-eye plot renders the coverage area of the synthetic vision detector as a shaded area. The ideal lane markings are additionally shown, as well as the synthetically detected left and right lane boundaries (here shown in red and green, respectively).

You can run the full simulation and explore the results of the using the following commands.

```
sim('LKATestBenchExample');                         % Simulate to end of scenario
close(findall(0,'Tag','LKATestBenchExample/Bird''s-Eye Plot')) % Close the Bird's-Eye P
plotLKAResults(scenario,logsout,driverPath);
```

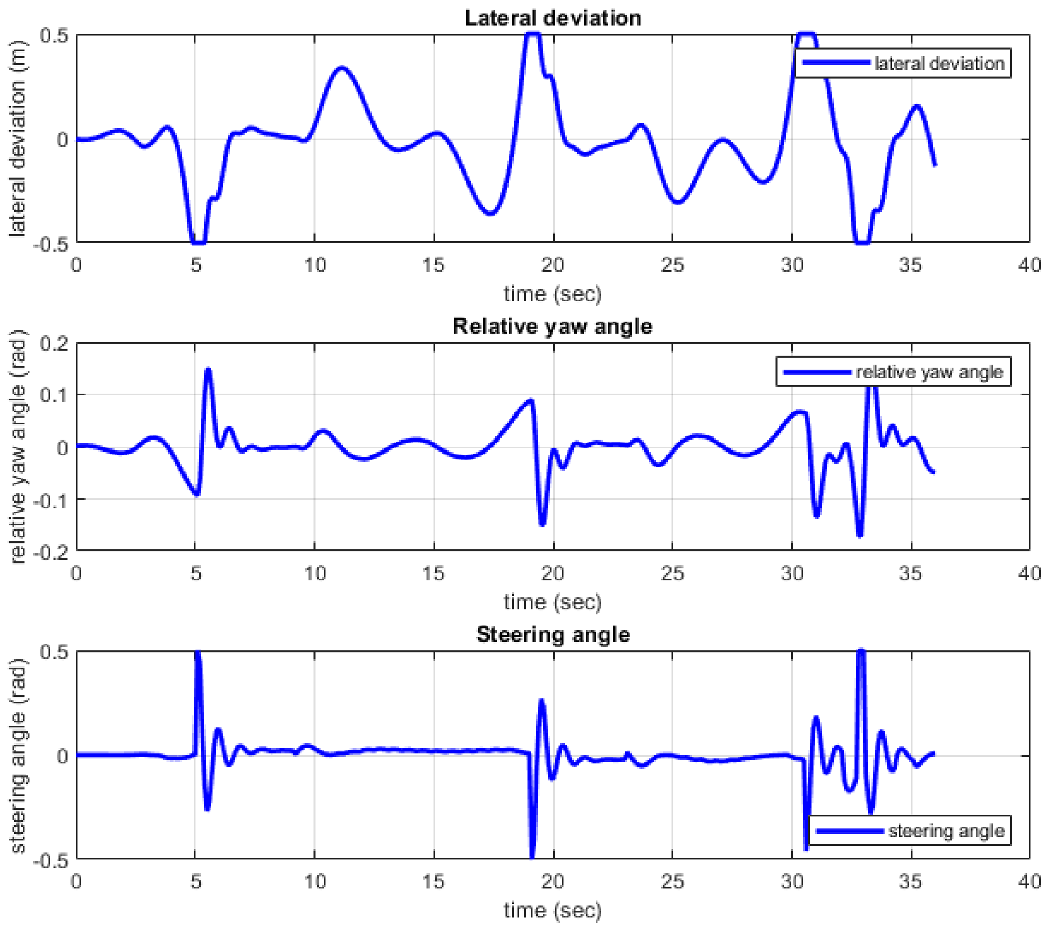
```
   Assuming no disturbance added to measured output channel #1.
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```



The blue curve for the driver path shows that the distracted driver may drive the ego car to another lane when the road curvature changes. The red curve for the driver with Lane Keeping Assist shows that the ego car remains in its lane when the road curvature changes.

Use the following command to depict the controller performance.

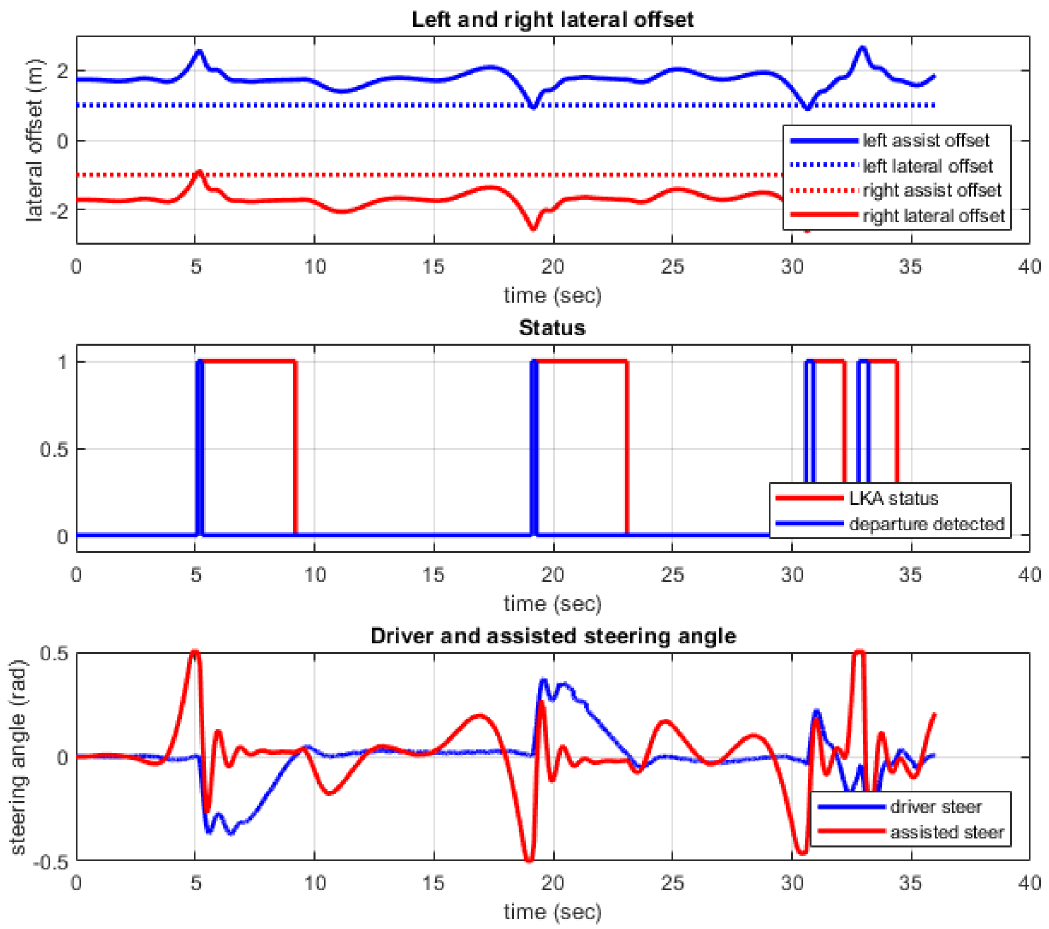
```
plotLKAPerformance(logsout);
```



- Top plot shows the lateral deviation relative to ego car. 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 car. The steering angle with LKA is within [-0.5,0.5] rad.

Use the following command to depict the controller status.

```
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 car correctly.
- Bottom plot shows that 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 (e.g. between 9 s to 17 s).

Simulate Lane Following

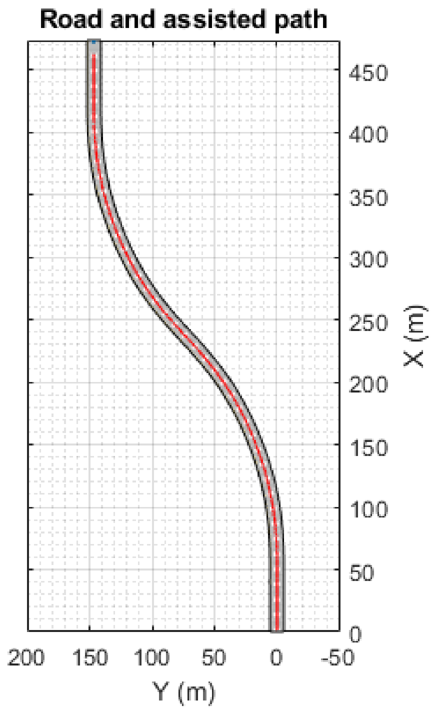
You can modify the value of "Safe Lateral Offset" for LKA to ignore the driver input, putting the controller into a pure lane following mode. By increasing this threshold, the lateral offset is always within the distance set by the lane keeping assist. Thus, the status for lane departure is on and the lane keeping assist takes control all the time.

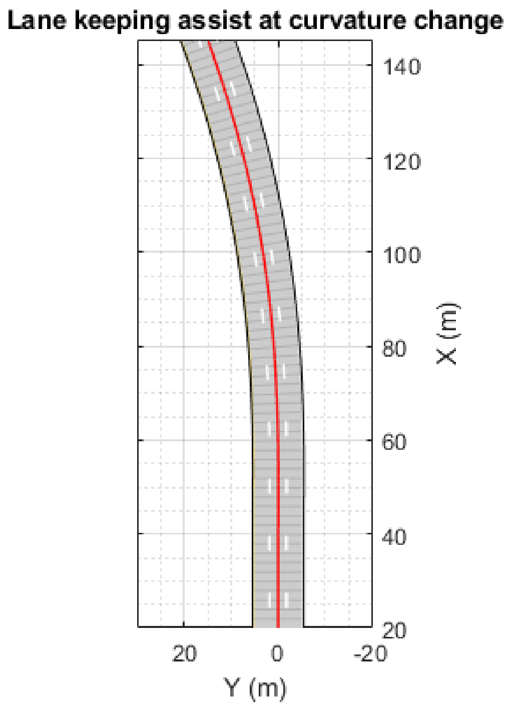
```
set_param('LKATestBenchExample/Safe Lateral Offset','Value','2');
sim('LKATestBenchExample');                % Simulate to end of scenario
close(findall(0,'Tag','LKATestBenchExample/Bird''s-Eye Plot')) % Close the Bird's-Eye F
```

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

You can explore the results of the using the following commands.

```
plotLKAResults(scenario,logsout)
```

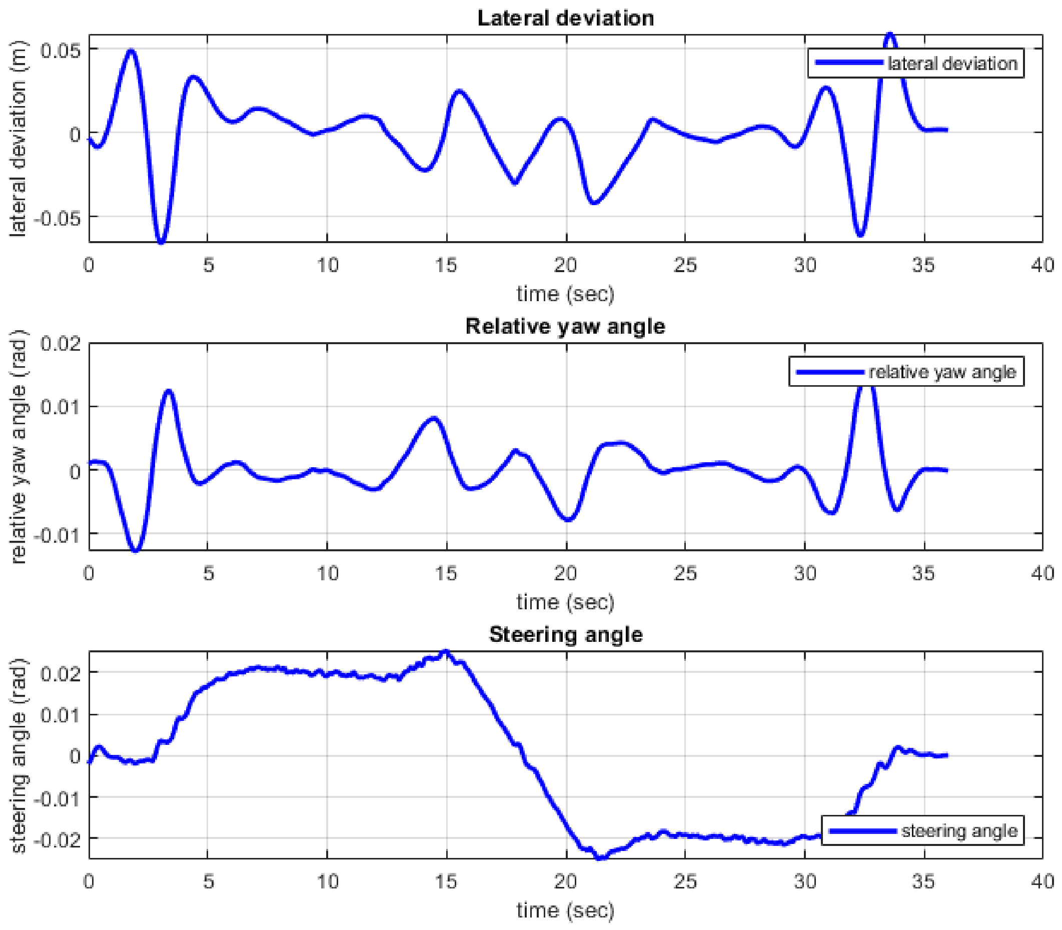




The red curve shows that the Lane Keeping Assist on its own can keep the ego car travelling along the centerline of its lane.

Use the following command to depict the controller performance.

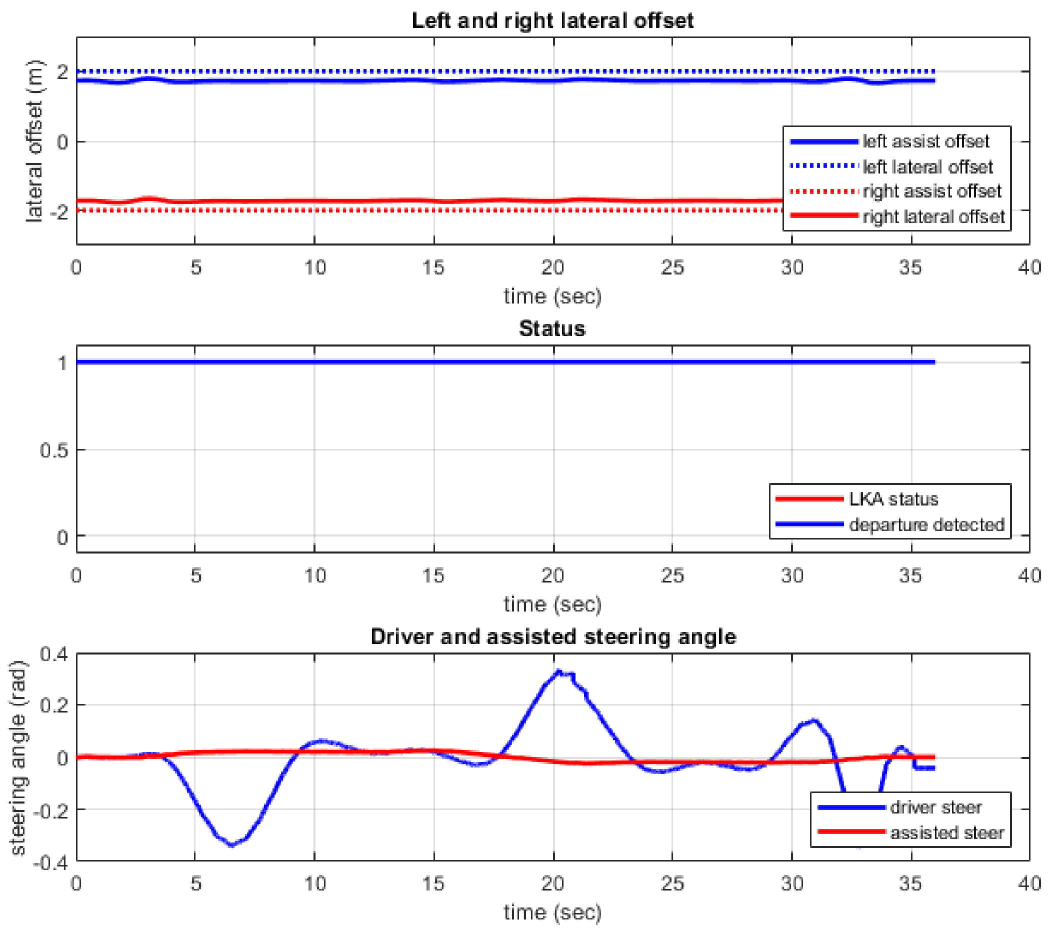
```
plotLKAPerformance(logsout);
```



- Top plot shows the lateral deviation relative to ego car. 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 car. The steering angle with LKA is within [-0.04,0.04] rad.

Use the following command to depict the controller status.

```
plotLKAStatus(logsout);
```



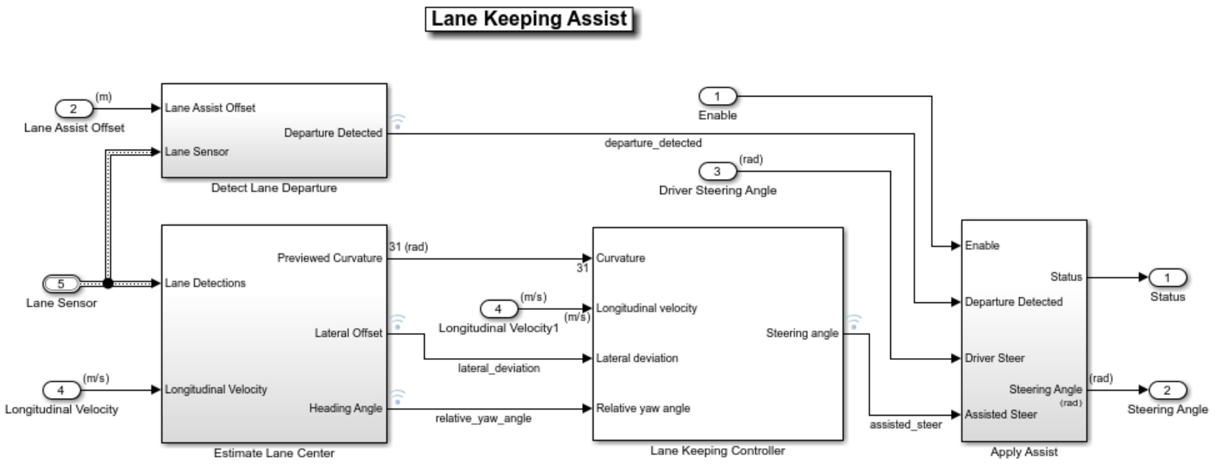
- Top plot shows the left and right lane offset. Since the lateral offset is never within the distance set by the lane keeping assist, the lane departure is not detected.

- Middle plot shows the LKA status is always one, that is, the Lane Keeping Assist takes control all the time.
- Bottom plot shows that 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')
```

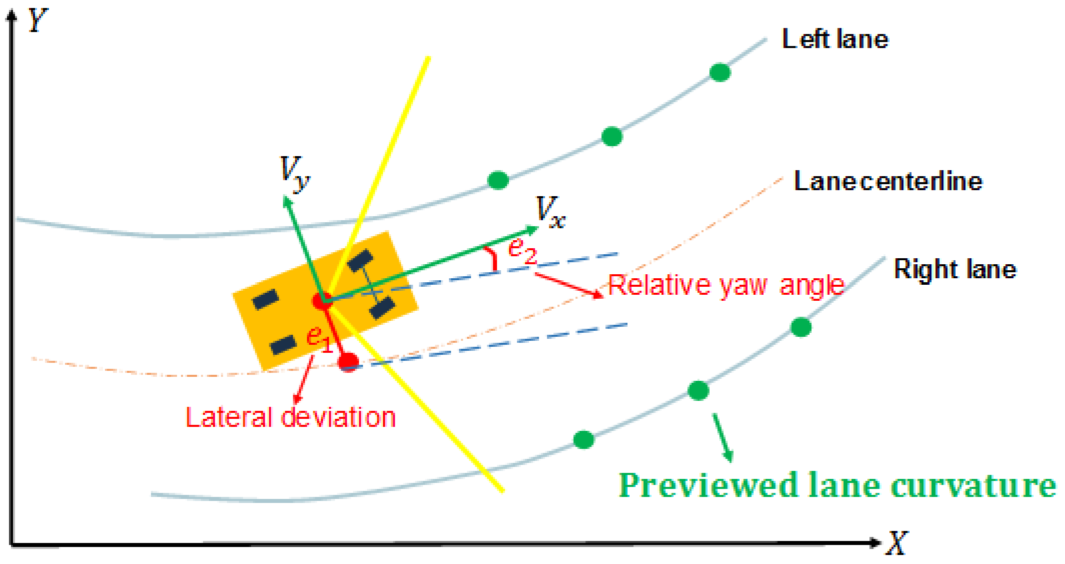


The "Detect Lane Departure" subsystem outputs a signal which is true when the vehicle is too close to a detected lane. We detect a departure when the offset between the vehicle and lane boundary from the Lane Sensor tis less than the Lane Assist Offset input.

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, we 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). We weight each of the resulting centered curves by the strength of the detection and pass the averaged result to the controller. Also, it provides finite values for inputs to the Lane

Keeping Controller subsystem. The previewed curvature provides the centerline of lane curvature ahead of the ego car. In this example, the ego car can look ahead for 3 seconds (i.e. PredictionHorizon*Ts). This enables the controller to use previewed information for calculating steering angle for the ego car 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 δ. This goal is achieved by driving the lateral deviation e_1 and the relative yaw angle e_2 to be small (see the figure below).



The LKA controller calculates a steering angle for the ego car based on the following inputs:

- Previewed curvature (derived from Lane Detections)
- Ego longitudinal velocity
- Lateral deviation (derived from Lane Detections)
- Relative yaw angle (derived from Lane Detections)

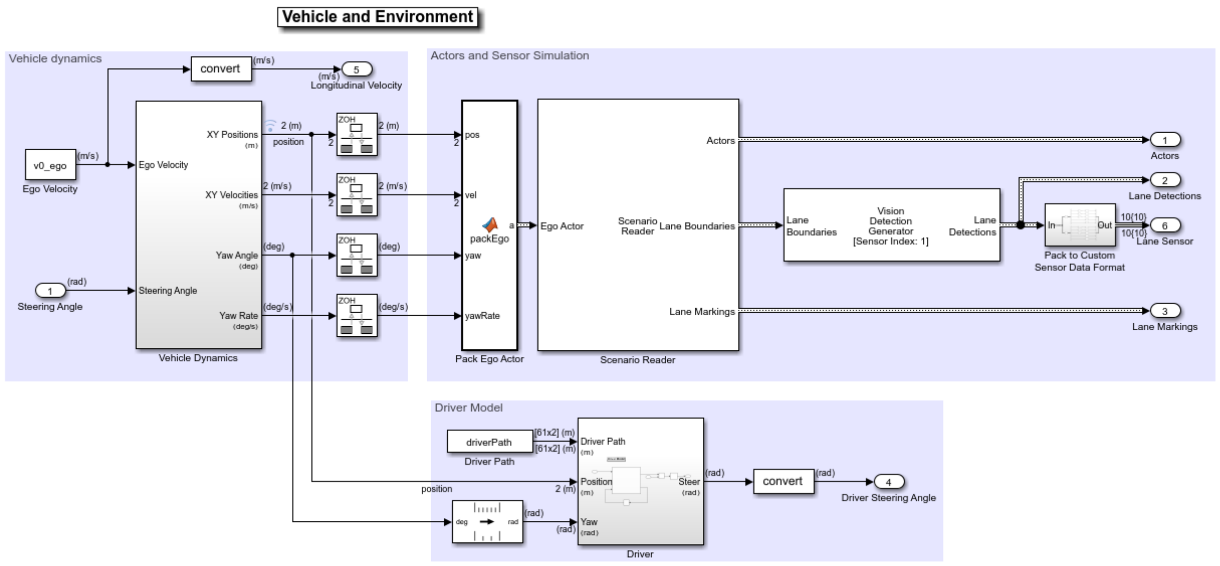
Considering physical limitations of the ego car, the steering angle is constrained to be within [-0.5,0.5] rad. You can change the prediction horizon or move the "Controller Behavior" slider to adjust the performance of the controller.

The "Apply Assist" subsystem decides if the lane keeping controller or the driver takes control of the ego car. The subsystem switches between the driver commanded steer and the assisted steer from the "Lane Keeping Controller". The switch to assisted steer is initiated in 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" block generates the ideal left and right lane boundaries based on the position of the vehicle with respect to the scenario created in `helperLKASetUp`.

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
    rtwbuild('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 enables 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, test it in Simulink using synthetic data generated by the Automated Driving System Toolbox, componentize it, and automatically generate code for it.

See Also

Blocks
Lane Keeping Assist System

More About

- “Automated Driving Using Model Predictive Control” on page 11-2