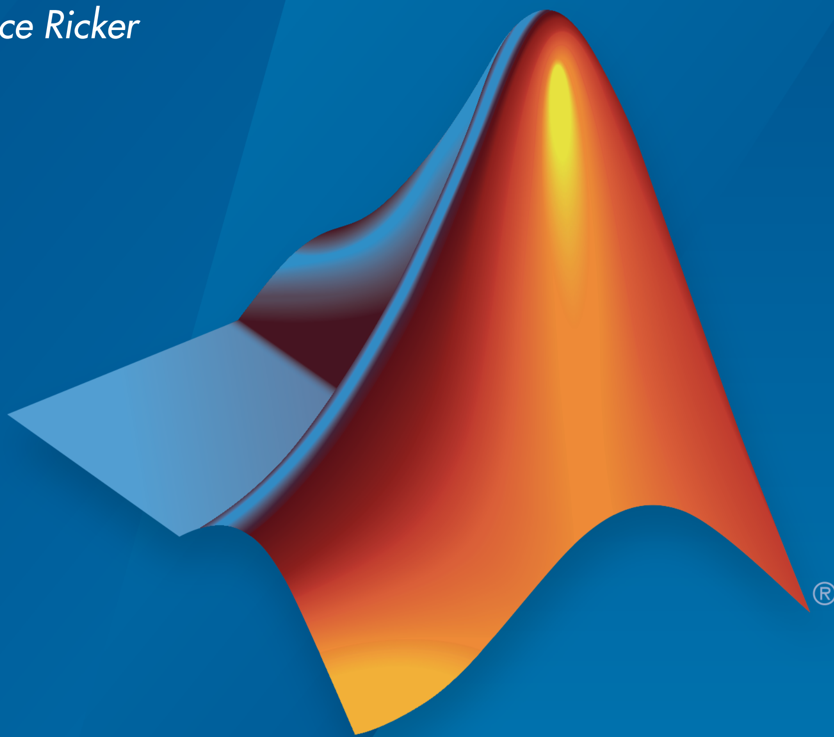
Model Predictive Control Toolbox™

Reference

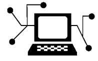
Alberto Bemporad
Manfred Morari
N. Lawrence Ricker



MATLAB®

R2015a

How to Contact MathWorks



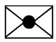
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000



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

Model Predictive Control Toolbox™ Reference

© COPYRIGHT 2005–2015 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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

Contents

1 Functions – Alphabetical List

2 Block Reference

3 Object Reference

MPC Controller Object **3-2**
ManipulatedVariables **3-2**
OutputVariables **3-4**
DisturbanceVariables **3-5**
Weights **3-5**
Model **3-7**
Ts **3-9**
Optimizer **3-9**
PredictionHorizon **3-10**
ControlHorizon **3-10**
History **3-10**
Notes **3-10**
UserData **3-11**
Construction and Initialization **3-11**

MPC Simulation Options Object **3-12**

MPC State Object **3-15**

Explicit MPC Controller Object **3-17**
Properties **3-17**

Functions – Alphabetical List

cloffset

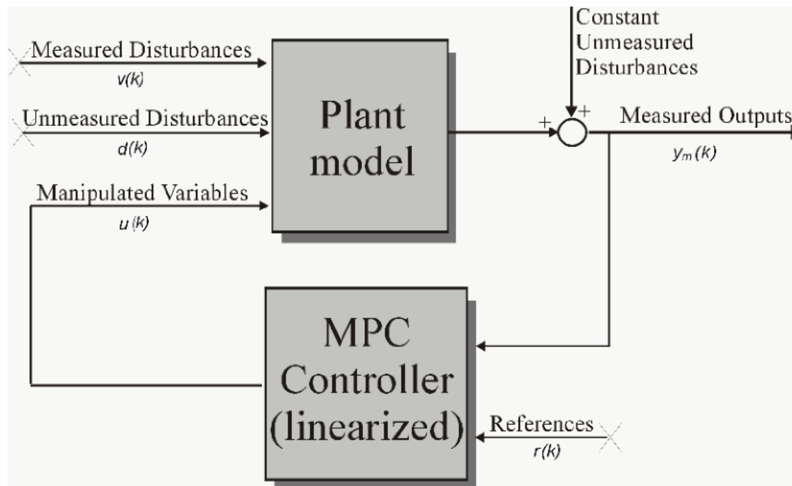
Compute MPC closed-loop DC gain from output disturbances to measured outputs assuming constraints are inactive at steady state

Syntax

```
DCgain=cloffset(MPCobj)
```

Description

The `cloff` function computes the DC gain from output disturbances to measured outputs, assuming constraints are not active, based on the feedback connection between `Model.Plant` and the linearized MPC controller, as depicted below.



Computing the Effect of Output Disturbances

By superposition of effects, the gain is computed by zeroing references, measured disturbances, and unmeasured input disturbances.

`DCgain=cloffset(MPCobj)` returns an n_{ym}-by-n_{ym} DC gain matrix `DCgain`, where n_{ym} is the number of measured plant outputs. `MPCobj` is the MPC object specifying the

controller for which the closed-loop gain is calculated. `DCgain(i,j)` represents the gain from an additive (constant) disturbance on output `j` to measured output `i`. If row `i` contains all zeros, there will be no steady-state offset on output `i`.

See Also

`mpc` | `ss`

Related Examples

- "Compute Steady-State Gain"

compare

Compare two MPC objects

Syntax

```
yesno=compare(MPC1,MPC2)
```

Description

The compare function compares the contents of two MPC objects `MPC1`, `MPC2`. If the design specifications (models, weights, horizons, etc.) are identical, then `yesno` is equal to 1.

Note compare may return `yesno = 1` even if the two objects are not identical. For instance, `MPC1` may have been initialized while `MPC2` may have not, so that they may have different sizes in memory. In any case, if `yesno = 1`, the behavior of the two controllers will be identical.

See Also

`mpc`

d2d

Change MPC controller's sampling time

Syntax

```
MPCobj=d2d(MPCobj,ts)
```

Description

The `d2d` function changes the sampling time of the MPC controller `MPCobj` to `ts`. All models are sampled or resampled as soon as the QP matrices must be computed, e.g., when `sim` or `mpcmove` are used.

See Also

`mpc` | `set`

generateExplicitMPC

Convert implicit MPC controller to explicit MPC controller

Given a traditional Model Predictive Controller design in the implicit form, convert it to the explicit form for real-time applications requiring fast sample time.

Syntax

```
EMPCobj = generateExplicitMPC(MPCobj,range)
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

Description

`EMPCobj = generateExplicitMPC(MPCobj,range)` converts a traditional (implicit) MPC controller to the equivalent explicit MPC controller, using the specified parameter bounds. This calculation usually requires significant computational effort because a multi-parametric quadratic programming problem is solved during the conversion.

`EMPCobj = generateExplicitMPC(MPCobj,range,opt)` converts the MPC controller using additional optimization options.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;
```

```
p = 10;
m = 3;
MPCobj = mpc(plant,Ts,p,m);
```

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

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(MPCobj);
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
   Assuming unmeasured input disturbance #1 is white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use `generateExplicitOptions` to create a default options set, and then modify the `polyreduction` option.

```
opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

```
Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        1
Number of parameters:      4
Is solution simplified:    No
State Estimation:          Default Kalman gain
```

```
----------------------------------------------
Type 'EMPCobj.MPC' for the original implicit MPC design.
Type 'EMPCobj.Range' for the valid range of parameters.
Type 'EMPCobj.OptimizationOptions' for the options used in multi-parametirc QP computat
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

- “Explicit MPC Control of a Single-Input-Single-Output Plant”
- “Explicit MPC Control of an Aircraft with Unstable Poles”
- “Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output”

Input Arguments

`MPCobj` — Traditional MPC controller

MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

`range` — Parameter bounds

structure

Parameter bounds, specified as a structure that you create with the `generateExplicitRange` command. This structure specifies the bounds on the parameters upon which the explicit MPC control law depends, such as state values, measured disturbances, and manipulated variables. See `generateExplicitRange` for detailed descriptions of these parameters.

`opt` — optimization options

structure

Optimization options for the conversion computation, specified as a structure that you create with the `mpcExplicitOptions` command. See `generateExplicitOptions` for detailed descriptions of these options.

Output Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller that is equivalent to the input traditional controller, returned as an explicit MPC controller object. The properties of the explicit MPC controller object are summarized in the following table.

Property	Description
`MPC`	Traditional (implicit) controller object used to generate the explicit MPC controller. You create this MPC controller using is the `mpc` command. It is the first argument to `generateExplicitMPC` when you create the explicit MPC controller. See "MPC Controller Object" on page 3-2 or type `mpcprops` for details regarding the properties of the MPC controller.
`Range`	1-D structure containing the parameter bounds used to generate the explicit MPC controller. These determine the resulting controller's valid operating range. This property is automatically populated by the `range` input argument to `generateExplicitMPC` when you create the explicit MPC controller. See `generateExplicitRange` for details about this structure.
`OptimizationOptions`	1-D structure containing user-modifiable options used to generate the explicit MPC controller. This property is automatically populated by the `opt` argument to `generateExplicitMPC` when you create the explicit MPC controller. See `generateExplicitOptions` for details about this structure.
`PiecewiseAffineSolution`	n_r-dimensional structure, where n_r is the number of piecewise affine (PWA) regions required to represent the control law. The *i*th element contains the details needed to compute the optimal manipulated variables

Property	Description
	when the solution lies within the *i*th region. See "Implementation".
`IsSimplified`	Logical switch indicating whether the explicit control law has been modified using the `simplify` command such that the explicit control law approximates the base (implicit) MPC controller. If the control law has not been modified, the explicit controller should reproduce the base controller's behavior exactly, provided both operate within the bounds described by the `Range` property.

More About

Tips

- Using Explicit MPC, you will most likely achieve best performance in small control problems, which involve small numbers of plant inputs/outputs/states as well as the number of constraints.
- Test the implicit controller thoroughly before attempting a conversion. This helps to determine the range of controller states and other parameters needed to generate the explicit controller.
- Simulate the explicit controller's performance using the `sim` or `mpcmoveExplicit` commands, or the Explicit MPC Controller block in Simulink®.
- `generateExplicitMPC` displays progress messages in the command window. Use `mpcverbosity` to turn off the display.

- "Explicit MPC"
- "Design Workflow for Explicit MPC"

See Also

`generateExplicitOptions` | `generateExplicitRange` | `mpc` | `simplify`

generateExplicitOptions

Optimization options for explicit MPC generation

Syntax

```
opt = generateExplicitOptions(MPCobj)
```

Description

`opt = generateExplicitOptions(MPCobj)` creates a set of options to use when converting a traditional MPC controller, MPCobj, to explicit form using `generateExplicitMPC`. The options set is returned with all options set to default values. Use dot notation to modify the options.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;
p = 10;
m = 3;
MPCobj = mpc(plant,Ts,p,m);
```

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

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(MPCobj);
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
   Assuming unmeasured input disturbance #1 is white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use `generateExplicitOptions` to create a default options set, and then modify the `polyreduction` option.

```
opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

```
Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        1
Number of parameters:      4
Is solution simplified:    No
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'EMPCobj.MPC' for the original implicit MPC design.
Type 'EMPCobj.Range' for the valid range of parameters.
Type 'EMPCobj.OptimizationOptions' for the options used in multi-parametirc QP computat
```

```
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Input Arguments

`MPCobj` — Traditional MPC controller
MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

Output Arguments

`opt` — Options for generating explicit MPC controller
structure

Options for generating explicit MPC controller, returned as a structure. When you create the structure, all the options are set to default values. Use dot notation to modify any options you want to change. The fields and their default values are as follows.

`zerotol` — Zero-detection tolerance
`1e-8` (default) | positive scalar value

Zero-detection tolerance used by the NNLS solver, specified as a positive scalar value.

`removetol` — Redundant-inequality-constraint detection tolerance
`1e-4` (default) | positive scalar value

Redundant-inequality-constraint detection tolerance, specified as a positive scalar value.

`flattol` — Flat region detection tolerance
`1e-5` (default) | positive scalar value

Flat region detection tolerance, specified as a positive scalar value.

`normalizetol` — Constraint normalization tolerance
0.01 (default) | positive scalar value

Constraint normalization tolerance, specified as a positive scalar value.

`maxiterNNLS` — Maximum number of NNLS solver iterations
500 (default) | positive integer

Maximum number of NNLS solver iterations, specified as a positive integer.

`maxiterQP` — Maximum number of QP solver iterations
200 (default) | positive integer

Maximum number of QP solver iterations, specified as a positive integer.

`maxiterBS` — Maximum number of bisection method iterations
100 (default) | positive integer

Maximum number of bisection method iterations used to detect region flatness, specified as a positive integer.

`polyreduction` — Method for removing redundant inequalities
2 (default) | 1

Method used to remove redundant inequalities, specified as either 1 (robust) or 2 (fast).

See Also

`generateExplicitMPC`

generateExplicitRange

Bounds on explicit MPC control law parameters

Syntax

```
Range = generateExplicitRange(MPCobj)
```

Description

`Range = generateExplicitRange(MPCobj)` creates a structure of parameter bounds based upon a traditional (implicit) MPC controller object. The range structure is intended for use as an input argument to `generateExplicitMPC`. Usually, the initial range values returned by `generateExplicitRange` are not suitable for generating an explicit MPC controller. Therefore, use dot notation to set the values of the range structure before calling `generateExplicitMPC`.

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;
p = 10;
m = 3;
MPCobj = mpc(plant,Ts,p,m);
```

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
```

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

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(MPCobj);
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
   Assuming unmeasured input disturbance #1 is white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use `generateExplicitOptions` to create a default options set, and then modify the `polyreduction` option.

```
opt = generateExplicitOptions(MPCobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
EMPCobj = generateExplicitMPC(MPCobj,range,opt)
```

```
Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        1
Number of parameters:      4
Is solution simplified:    No
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'EMPCobj.MPC' for the original implicit MPC design.
Type 'EMPCobj.Range' for the valid range of parameters.
Type 'EMPCobj.OptimizationOptions' for the options used in multi-parametirc QP computat
```

```
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Input Arguments

`MPCobj` — Traditional MPC controller

MPC controller object

Traditional MPC controller, specified as an MPC controller object. Use the `mpc` command to create a traditional MPC controller.

Output Arguments

`Range` — Parameter bounds

structure

Parameter bounds for generating an explicit MPC controller from MPCobj, returned as a structure.

Initially, each parameter's minimum and maximum bounds are identical. All such parameters are considered fixed. When you generate an explicit controller, any fixed parameters must be constant when the controller operates. This is unlikely to happen in general. Thus, you must specify valid bounds for all parameters. Use dot notation to set the values of the range structure as appropriate for your system.

The fields of the range structure are as follows.

`State` — Bounds on controller state values

structure

Bounds on controller state values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_x, where n_x is the number of controller states. `Range.State.Min` and `Range.State.Max` contain the minimum and maximum values, respectively, of all controller states. For example, suppose you are designing a two-state controller. You have determined that the range of the first controller state is `[-1000,1000]`, and that of the second controller state is `[0,2*pi]`. Set these bounds as follows:

```
Range.State.Min(:) = [-1000,0];
```

```
Range.State.Max(:) = [1000,2*pi];
```

MPC controller states include states from plant model, disturbance model, and noise model, in that order. Setting the range of a state variable is sometimes difficult when a state does not correspond to a physical parameter. In that case, multiple runs of open-loop plant simulation with typical reference and disturbance signals are recommended in order to collect data that reflect the ranges of states.

`Reference` — Bounds on controller reference signal values

structure

Bounds on controller reference signal values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_y, where n_y is the number of plant outputs. `Range.Reference.Min` and `Range.Reference.Max` contain the minimum and maximum values, respectively, of all reference signal values. For example, suppose you are designing a controller for a two-output plant. You have determined that the range of the first plant output is `[-1000,1000]`, and that of the second plant output is `[0,2*pi]`. Set these bounds as follows:

```
Range.Reference.Min(:) = [-1000,0];
Range.Reference.Max(:) = [1000,2*pi];
```

Usually you know the practical range of the reference signals being used at the nominal operating point in the plant. The ranges used to generate the explicit MPC controller must be at least as large as the practical range.

`MeasuredDisturbance` — Bounds on measured disturbance values

structure

Bounds on measured disturbance values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_{md}, where n_{md} is the number of measured disturbances. If your system has no measured disturbances, leave the generated values of this field unchanged.

`Range.MeasuredDisturbance.Min` and `Range.MeasuredDisturbance.Max` contain the minimum and maximum values, respectively, of all measured disturbance signals. For example, suppose you are designing a controller for a system with two measured disturbances. You have determined that the range of the first disturbance is `[-1,1]`, and that of the second disturbance is `[0,0.1]`. Set these bounds as follows:

```
Range.Reference.Min(:) = [-1,0];
Range.Reference.Max(:) = [1,0.1];
```

Usually you know the practical range of the measured disturbance signals being used at the nominal operating point in the plant. The ranges used to generate the explicit MPC controller must be at least as large as the practical range.

`ManipulatedVariable` — Bounds on manipulated variable values

structure

Bounds on manipulated variable values, specified as a structure containing fields `Min` and `Max`. Each of `Min` and `Max` is a vector of length n_u, where n_u is the number of manipulated variables. `Range.ManipulatedVariable.Min` and `Range.ManipulatedVariable.Max` contain the minimum and maximum values, respectively, of all manipulated variables. For example, suppose your system has two manipulated variables. The range of the first manipulated variable is `[-1,1]`, and that of the second variable is `[0,0.1]`. Set these bounds as follows:

```
Range.ManipulatedVariable.Min(:) = [-1,0];
Range.ManipulatedVariable.Max(:) = [1,0.1];
```

If manipulated variables are constrained, the ranges used to generate the explicit MPC controller must be at least as large as these limits.

See Also

`generateExplicitMPC` | `generateExplicitOptions` | `mpc`

generatePlotParameters

Parameters for plotSection

Syntax

```
plotParams = generatePlotParameters(EMPCobj)
```

Description

`plotParams = generatePlotParameters(EMPCobj)` creates a structure of parameters for a 2-D sectional plot of the explicit MPC control law of the explicit MPC controller, EMPCobj. You set the fields of this structure and use it to generate the plot using the `plotSection` command.

Examples

Specify Fixed Parameters for 2-D Plot of Explicit Control Law

Suppose `EMPCObj` is an explicit MPC controller involving six controller state variables, two plant outputs (reference signals), and two manipulated variables. Create a plotParams structure that fixes all parameters at their nominal values, except the second manipulated variable and third controller state.

Generate the default plotParams structure for the MPC controller.

```
plotParams = generatePlotParameters(EMPCobj);
```

By default, this structure specifies that all parameters of the controller are fixed at their nominal values.

Allow the third controller state to vary for the purposes of creating a plot. To do so, remove the entry corresponding to that state from `plotParams`.

```
plotParams.State.Index(3) = [];
plotParams.State.Value(3) = [];
```

Similarly, allow the second manipulated variable to vary for the plot.

```
plotParams.ManipulatedVariable.Index(2) = [];
plotParams.ManipulatedVariable.Value(2) = [];
```

You can now use the plot parameters to generate the 2-D section plot for the controller.

```
plotSection(EMPCobj,plotParams)
```

Input Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller for which you want to create a 2-D sectional plot, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

Output Arguments

`plotParams` — Parameters for sectional plot

structure

Parameters for sectional plot of explicit MPC control law, returned as a structure.

As returned by `generatePlotParameters`, the plotParams structure command fixes all the control law's parameters at their nominal values. To obtain the desired plot, eliminate the `Index` and `Value` entries of the two parameters forming the plot axes, and modify fixed values as necessary. Then, use the `plotSection` command to display the 2-D sectional plot of the explicit control law's PWA regions with the remaining free parameters as the x and y axes.

The fields of the plot-parameters structure are as follows.

`State` — Fixed controller states

structure

Fixed controller states, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.State.Index` is a vector that contains the indices of

the controller states to fix for the plot, and `plotParams.State.Value` contains the corresponding constant state values.

Modify the default value of `plotParams.State` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

`Reference` — Fixed reference signal values
structure

Fixed reference signal values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.Reference.Index` is a vector that contains the indices of the reference signals to fix for the plot, and `plotParams.Reference.Value` contains the corresponding constant reference signal values.

Modify the default value of `plotParams.Reference` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

`MeasuredDisturbance` — Fixed measured disturbance values
structure

Fixed measured disturbance values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.MeasuredDisturbance.Index` is a vector that contains the indices of the measured disturbances to fix for the plot, and `plotParams.MeasuredDisturbance.Value` contains the corresponding constant measured disturbance values.

Modify the default value of `plotParams.MeasuredDisturbance` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

`ManipulatedVariable` — Fixed manipulated variable values
structure

Fixed manipulated variable values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.ManipulatedVariable.Index` is a vector that contains the indices of the manipulated variables to fix for the plot, and `plotParams.ManipulatedVariable.Value` contains the corresponding constant manipulated variable values.

Modify the default value of `plotParams.ManipulatedVariable` to generate the desired plot. See “Specify Fixed Parameters for 2-D Plot of Explicit Control Law” on page 1-20.

See Also

`generateExplicitMPC` | `plotSection`

get

MPC property values

Syntax

```
Value = get(MPCobj,'PropertyName')
Struct = get(MPCobj)
get(MPCobj)
```

Description

`Value = get(MPCobj,'PropertyName')` returns the current value of the property `PropertyName` of the MPC controller `MPCobj`. The string `'PropertyName'` can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). You can specify any generic MPC property.

`Struct = get(MPCobj)` converts the MPC controller `MPCobj` into a standard MATLAB® structure with the property names as field names and the property values as field values.

`get(MPCobj)` without a left-side argument displays all properties of `MPCobj` and their values.

More About

Tips

An alternative to the syntax

```
Value = get(MPCobj,'PropertyName')
```

is the structure-like referencing

```
Value = MPCobj.PropertyName
```

For example,

```
MPCobj.Ts
MPCobj.p
```

return the values of the sampling time and prediction horizon of the MPC controller `MPCobj`.

See Also

`mpc` | `set`

getconstraint

Model Predictive Control custom constraint definitions

Syntax

```
[E,F,G,V,S] = getconstraint(mpcobj)
```

Description

`[E,F,G,V,S] = getconstraint(mpcobj)` returns the custom constraints previously defined for the `mpc` object, mpcobj. The constraints are in the general form

$$Eu(k+j)+Fy(k+j)+Sv(k+j)\leq G+\varepsilon V$$

where:

- $j = 0,...,p$.
- p — MPC prediction horizon.
- k — current time index.
- u — column vector of manipulated variables.
- y — column vector of all plant output variables.
- v — column vector of measured disturbance variables.
- ε — scalar slack variable used for constraint softening.
- E, F, G, V and S — constant matrices.

`getconstraint` calculates the last constraint at time *k+p* assuming that `u(`*k+p*`|`*k*`) = u(`*k+p*`-1|`*k*`)`. This is because `u(`*k+p*`|`*k*`)` is not optimized by the model predictive controller.

Input Arguments

`mpcobj`

MPC controller, specified as an `mpc` object.

Output Arguments

E

Constant used in custom constraints as defined in Equation 1-1.

`[]` if mpcobj contains no custom constraints.

E is an n_c-by-n_u matrix, where n_c is the number of custom constraints and n_u is the number of manipulated variables.

F

Constant used in custom constraints as defined in Equation 1-1.

`[]` if mpcobj contains no custom constraints.

F is an n_c-by-n_y matrix, where n_c is the number of custom constraints and n_y is the number of output variables (measured and unmeasured).

G

Constant used in custom constraints as defined in Equation 1-1.

`[]` if mpcobj contains no custom constraints.

G is an n_c-by-1 vector, where n_c is the number of custom constraints.

V

Constant used in custom constraints as defined in Equation 1-1.

`[]` if mpcobj contains no custom constraints.

V is an n_c-by-1 vector, where n_c is the number of custom constraints.

If

- `V(i) = 0` — the ith constraint is hard
- `V(i) > 0` — the ith constraint is soft

  Where `i = 1,...,`n_c.

In general, as `V(i)` decreases, the controller decreases the allowed constraint violation, i.e. the constraint becomes harder.

S

Constant used in custom constraints as defined in Equation 1-1.

`[]` if mpcobj contains no custom constraints or there are no measured disturbances in the custom constraints.

S is an n_c-by-n_{md} matrix, where n_c is the number of custom constraints and n_{md} is the number of measured disturbance inputs.

Examples

Obtain the constraints associated with an MPC controller.

Create an `mpc` object with 2 manipulated variables and 2 measured outputs.

```
p = rss(3,2,3);
p.D = 0;
p = setmpcsignals(p,'mv',[1 2],'md',3);
c = mpc(p,0.1);
```

Assume that you have two soft constraints.

$$u_1 + u_2 \leq 5$$
$$y_2 + v \leq 10$$

Set the constraints for the `mpc` object.

```
E = [1 1; 0 0];
F = [0 0;0 1];
G = [5;10];
V = [1;1];
S = [0;1];
setconstraint(c,E,F,G,V,S);
```

Obtain the constraints for `c`.

```
[E F G V S] = getconstraint(c)
```

```
E =

     1     1
     0     0


F =

     0     0
     0     1


G =

     5
    10


V =

     1
     1


S =

     0
     1
```

See Also

`setconstraint`

getEstimator

Obtain Kalman gains and model for estimator design

Syntax

```
[L,M] = getEstimator(MPCobj)
[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj)
[L,M,model,index] = getEstimator(MPCobj,'sys')
```

Description

`[L,M] = getEstimator(MPCobj)` extracts the Kalman gains used by the state estimator in a model predictive controller. The estimator updates the states of internal plant, disturbance, and noise models at the beginning of each controller interval.

`[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj)` also returns the system matrices used to calculate the estimator gains.

`[L,M,model,index] = getEstimator(MPCobj,'sys')` returns an LTI state-space representation of the system used for state-estimator design and a structure summarizing the I/O signal types of the system.

Examples

Extract Parameters for State Estimation

The plant is a stable, discrete LTI ss model with four states, three inputs and three outputs. The manipulated variables are inputs 1 and 2. Input 3 is an unmeasured disturbance. Outputs 1 and 3 are measured. Output 2 is unmeasured.

Create a model of the plant and specify the signals for MPC.

```
rng(1253);              % For repeatable results
Plant = drss(4,3,3);
```

```
Plant.Ts = 0.25;
Plant = setmpcsignals(Plant,'MV',[1,2],'UD',3,'MO',[1 3],'UO', 2);
Plant.d(:,[1,2]) = 0;
```

The last command forces the plant to satisfy the assumption of no direct feedthrough.

Calculate the default Model Predictive Controller for this plant.

```
MPCobj = mpc(Plant);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming 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 y3 and zero weight for output(s) y2
```

Obtain the parameters to be used in state estimation.

```
[L,M,A,Cm,Bu,Bv,Dvm] = getEstimator(MPCobj);
```

```
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
-->Integrated white noise added on measured output channel #1.
   Assuming unmeasured input disturbance #3 is white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Based on the estimator state equation, the the estimator poles are given by the eigenvalues of `A - L*Cm`. Calculate and display the poles.

```
Poles = eig(A - L*Cm)
```

```
Poles =

   -0.7467
   -0.5019
    0.0769
    0.4850
    0.8825
    0.8291
```

Confirm that the default estimator is asymptotically stable.

```
max(abs(Poles))
```

```
ans =

    0.8825
```

This value is less than 1, so the estimator is asymptotically stable.

Verify that in this case, `L = A*M`.

```
L - A*M
```

```
ans =

   1.0e-16 *

    0.5551    0.4163
         0         0
    0.2776    0.1388
    0.2776    0.2082
    0.1388    0.2776
   -0.1388   -0.2776
```

Input Arguments

`MPCobj` — MPC controller

MPC controller object

MPC controller, specified as an MPC controller object. Use the `mpc` command to create the MPC controller.

Output Arguments

`L` — Kalman gain matrix for time update

matrix

Kalman gain matrix for the time update, returned as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See "State Estimator Equations" on page 1-34.

`M` — Kalman gain matrix for measurement update

matrix

Kalman gain matrix for the measurement update, returned as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See "State Estimator Equations" on page 1-34.

`A,Cm,Bu,Bv,Dvm` — System matrices

matrices

System matrices used to calculate the estimator gains, returned as matrices of various dimensions. For definitions of these system matrices, see "State Estimator Equations" on page 1-34.

`model` — System used for state-estimator design

state-space model

System used for state-estimator design, returned as a state-space (`ss`) model. The input to model is a vector signal comprising the following components, concatenated in the following order:

- Manipulated variables
- Measured disturbance variables
- 1
- Noise inputs to disturbance models
- Noise inputs to measurement noise model

The number of noise inputs depends on the disturbance and measurement noise models within MPCobj. For the category noise inputs to disturbance models, inputs to the input disturbance model (if any) precede those entering the output disturbance model (if any). The constant input, 1, accounts for nonequilibrium nominal values (see "MPC Modeling").

To make the calculation of gains L and M more robust, additive white noise inputs are assumed to affect the manipulated variables and measured disturbances (see "Controller State Estimation"). These white noise inputs are not included in model.

`index` — Locations of variables within model

structure

Locations of variables within the inputs and outputs of model. The structure summarizes these locations with the following fields and values.

Field Name	Value
`ManipulatedVariables`	Indices of manipulated variables within the input vector of model.
`MeasuredDisturbances`	Indices of measured input disturbances within the input vector of model.
`Offset`	Index of the constant input 1 within the input vector of model.
`WhiteNoise`	Indices of unmeasured disturbance inputs within the input vector of model.
`MeasuredOutputs`	Indices of measured outputs within the output vector of model.
`UmeasuredOutputs`	Indices of unmeasured outputs within the output vector of model.

More About

State Estimator Equations

The following equations describe the state estimation. For more details, see “Controller State Estimation”.

Output estimate: $y_m[n|n–1] = C_m\ x[n|n–1] + D_{vm}\ v[n]$.

Measurement update: $x[n|n] = x[n|n–1] + M\ (y_m[n] – y_m[n|n–1])$.

Time update: $x[n+1|n] = A\ x[n|n–1] + B_u\ u[n] + B_v\ v[n] + L\ (y_m[n] – y_m[n|n–1])$.

Estimator state: $x[n+1|n] = (A – L\ C_m)\ x[n|n–1] + B_u\ u[n] + (B_v – L\ D_{vm})\ v[n] + L\ y_m[v]$. The estimator state is based on the current measurement of $y_m[n]$ and $v[n]$ as well as the optimal control action $u[n]$ computed at the current control interval.

The variables in these equations are summarized in the following table.

Symbol	Description
x	Controller state vector, length n_x. It includes (in this sequence): • Plant model state estimates. Dimension obtained by conversion of `MPCobj.Model.Plant` to discrete LTI state-space form (if necessary), followed by use of `absorbDelay` to convert any delays to additional states. • Input disturbance model state estimates (if any). Use the `getindist` command to review the input disturbance model structure. • Output disturbance model state estimates (if any). Use the `getoutdist` command to review the output disturbance model structure. • Output measurement noise states (if any) as specified by `MPCobj.Model.Noise`. The length n_x is the sum of the number of states in the above four categories.
y_m	Vector of measured outputs or an estimate of their true values, length n_{ym}.
u	Vector of manipulated variables, length n_u.
v	Vector of measured input disturbances, length n_v.
$[j \mid k]$	Denotes an estimate of a state or output at time t_j based on data available at time t_k.
$[k]$	Denotes a quantity known at time t_k, i.e., not an estimate.
A	n_x-by-n_x state transition matrix.
B_u	n_x-by-n_u matrix mapping u to x.
B_v	n_x-by-n_x matrix mapping v to x.
C_m	n_{ym}-by-n_x matrix mapping x to y_m.
D_{vm}	n_{ym}-by-n_v matrix mapping v to y_m. Note that $D_{um} = 0$ because there can be no direct feedthrough between any manipulated variable and any measured output.

Symbol	Description
L	n_x-by-n_{ym} Kalman gain matrix for the time update. (See `kalmd` in the Control System Toolbox™ documentation.) Note that *L* = *A***M* minimizes the expected state estimation error for most combinations of plant and disturbance models used in MPC, but this is not true in general.
M	n_x-by-n_{ym} Kalman gain matrix for the measurement update. (See `kalmd` in the Control System Toolbox documentation.)

- "Controller State Estimation"
- "MPC Modeling"

See Also

`getindist` | `getoutdist` | `mpc` | `mpcstate` | `setEstimator`

getindist

Unmeasured input disturbance model

Syntax

```
indist = getindist(MPCobj)
[indist,channels] = getindist(MPCobj)
```

Description

`indist = getindist(MPCobj)` returns the unmeasured input disturbance model, `indist`, used by model predictive controller object, `MPCobj`. This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. The input disturbance model:

- Is a discrete-time, delay-free, state-space (`ss`) object.
- Has unit-variance white noise input signals. By default, the number of inputs depends upon the number of unmeasured input disturbances and the need to maintain controller state observability. For custom input disturbance models, the number of inputs is your choice.
- Has `nd` outputs, where `nd` is the number of unmeasured disturbance inputs defined in `MPCobj.Model.Plant`. Each output channel is sent to the corresponding plant unmeasured disturbance input.

Use this command to review the current default or custom input disturbance model. You can override the controller default behavior using the `setindist` command.

For details on the role of disturbance modeling in model predictive control and about the model used in the algorithm for state estimation, see “Controller State Estimation”.

`[indist,channels] = getindist(MPCobj)` also returns the input channels to which integrated white noise has been added by default. If you specified a custom input disturbance model using `setindist`, `channels` is empty.

More About

- “MPC Modeling”

- "Controller State Estimation"

See Also

`getEstimator` | `getoutdist` | `mpc` | `setEstimator` | `setindist`

Introduced before R2006a

getname

I/O signal names in MPC prediction model

Syntax

```
name=getname(MPCobj,'input',I)
name=getname(MPCobj,'output',I)
```

Description

`name=getname(MPCobj,'input',I)` returns the name of the `I`th input signal in variable `name`. This is equivalent to `name = MPCobj.Model.Plant.InputName{I}`. The name property is equal to the contents of the corresponding `Name` field of `MPCobj.DisturbanceVariables` or `MPCobj.ManipulatedVariables`.

`name=getname(MPCobj,'output',I)` returns the name of the `I`th output signal in variable `name`. This is equivalent to `name=MPCobj.Model.Plant.OutputName{I}`. The name property is equal to the contents of the corresponding `Name` field of `MPCobj.OutputVariables`.

See Also

`setname` | `mpc` | `set`

getoutdist

Unmeasured output disturbance model

Syntax

```
outdist = getoutdist(MPCobj)
[outdist,channels] = getoutdist(MPCobj)
```

Description

`outdist = getoutdist(MPCobj)` returns the output disturbance model, `outdist`, used by the model predictive controller object, `MPCobj`. This model, in combination with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. The output disturbance model:

- Is a discrete-time, delay-free, state-space (`ss`) object.
- Has unit-variance white noise input signals. By default, the number of inputs depends upon the number of measured outputs and the need to maintain controller state observability. For custom output disturbance models, the number of inputs is your choice.
- Has `ny` outputs, where `ny` is the number of plant outputs defined in `MPCobj.Model.Plant`. Each model output is added to the corresponding plant output.

Use this command to review the current default or custom output disturbance model. You can override the controller default behavior using the `setoutdist` command.

For details on the role of disturbance modeling in model predictive control and about the model used in the algorithm for state estimation, see "Controller State Estimation".

`[outdist,channels] = getoutdist(MPCobj)` also returns the output channels to which integrated white noise has been added by default. If you specified a custom output disturbance model using `setoutdist`, `channels` is empty.

More About

- "MPC Modeling"

- "Controller State Estimation"

See Also

`getEstimator` | `getindist` | `mpc` | `setEstimator` | `setoutdist`

Introduced before R2006a

gpc2mpc

Generate MPC controller using generalized predictive controller (GPC) settings

Syntax

```
mpc = gpc2mpc(plant)
gpcOptions = gpc2mpc
mpc = gpc2mpc(plant,gpcOptions)
```

Description

`mpc = gpc2mpc(plant)` generates a single-input single-output MPC controller with default GPC settings and sampling time of the plant, *`plant`*. The GPC is a nonminimal state-space representation described in [1]. *`plant`* is a discrete-time LTI model with sampling time greater than 0.

`gpcOptions = gpc2mpc` creates a structure *`gpcOptions`* containing default values of GPC settings.

`mpc = gpc2mpc(plant,gpcOptions)` generates an MPC controller using the GPC settings in *`gpcOptions`*.

Input Arguments

plant

Discrete-time LTI model with sampling time greater than 0.

Default:

gpcOptions

GPC settings, specified as a structure with the following fields.

`N1`	Starting interval in prediction horizon, specified as a positive integer. **Default:** 1.

`N2`	Last interval in prediction horizon, specified as a positive integer greater than `N1`. **Default:** 10.
`NU`	Control horizon, specified as a positive integer less than the prediction horizon. **Default:** 1.
`Lam`	Penalty weight on changes in manipulated variable, specified as a positive integer greater than or equal to 0. **Default:** 0.
`T`	Numerator of the GPC disturbance model, specified as a row vector of polynomial coefficients whose roots lie within the unit circle. **Default:** [1].
`MVindex`	Index of the manipulated variable for multi-input plants, specified as a positive integer. **Default:** 1.

Default:

Examples

Design an MPC controller using GPC settings:

```
% Specify the plant described in Example 1.8  of
% [1].
G = tf(9.8*[1 -0.5 6.3],conv([1 0.6565],[1 -0.2366 0.1493]));

% Discretize the plant with sample time of 0.6 seconds.
Ts = 0.6;
Gd = c2d(G, Ts);

% Create a GPC settings structure.
GPCoptions = gpc2mpc;

% Specify the GPC settings described in example 4.11 of
% [1].
% Hu
GPCoptions.NU = 2;
% Hp
```

```
GPCoptions.N2 = 5;
% R
GPCoptions.Lam = 0;
GPCoptions.T = [1 -0.8];

% Convert GPC to an MPC controller.
mpc = gpc2mpc(Gd, GPCoptions);

% Simulate for 50 steps with unmeasured disturbance between
% steps 26 and 28, and reference signal of 0.
SimOptions = mpcsimopt(mpc);
SimOptions.UnmeasuredDisturbance = [zeros(25,1); ...
-0.1*ones(3,1); 0];
sim(mpc, 50, 0, SimOptions);
```

More About

Tips

- For plants with multiple inputs, only one input is the manipulated variable, and the remaining inputs are measured disturbances in feedforward compensation. The plant output is the measured output of the MPC controller.
- Use the MPC controller with Model Predictive Control Toolbox™ software for simulation and analysis of the closed-loop performance.

- “Design Controller Using the Design Tool”
- “Design Controller at the Command Line”

References

[1] Maciejowski, J. M. *Predictive Control with Constraints*, Pearson Education Ltd., 2002, pp. 133–142.

See Also

“MPC Controller Object”

mpc

Create MPC controller

Syntax

```
MPCobj = mpc(Plant)
MPCobj = mpc(Plant,Ts)
MPCobj = mpc(Plant,Ts,p,m,W,MV,OV,DV)
MPCobj = mpc(Models,Ts,p,m,W,MV,OV,DV)
```

Description

`MPCobj = mpc(Plant)` creates a Model Predictive Controller object based on a discrete-time prediction model. The prediction model `Plant` can be either an LTI model with a specified sample time, or a System Identification model (see "Identify Plant from Data"). The controller, `MPCobj`, inherits its control interval from `Plant.Ts`, and its time unit from `Plant.TimeUnit`. All other controller properties are default values. After you create the MPC controller, you can set its properties using `MPCobj.PropertyName = PropertyValue`.

`MPCobj = mpc(Plant,Ts)` specifies a control interval of `Ts`. If `Plant` is a discrete-time LTI model with an unspecified sample time (`Plant.Ts` = –1), it inherits sample time `Ts` when used for predictions.

`MPCobj = mpc(Plant,Ts,p,m,W,MV,OV,DV)` specifies additional controller properties such as the prediction horizon (`p`), control horizon (`m`), and input, input increment, and output weights (`W`). You can also set the properties of manipulated variables (`MV`), output variables (`OV`), and input disturbance variables (`DV`). If any of these values are omitted or empty, the default values apply.

`MPCobj = mpc(Models,Ts,p,m,W,MV,OV,DV)` creates a Model Predictive Controller object based on a prediction model set, `Models`. This set includes plant, input disturbance, and measurement noise models along with the nominal conditions at which the models were obtained.

Input Arguments

`Plant`

Plant model to be used in predictions, specified as an LTI model (`tf`, `ss`, or `zpk`) or a System Identification Toolbox™ model. If the `Ts` input argument is unspecified, `Plant` must be a discrete-time LTI object with a specified sample time, or a System Identification Toolbox model.

Unless you specify otherwise, controller design assumes that all plant inputs are manipulated variables and all plant outputs are measured. Use the `setmpcsignals` command or the LTI `InputGroup` and `OutputGroup` properties to designate other signal types.

`Ts`

Controller sample time (control interval), specified as a positive scalar value.

`p`

Prediction horizon, specified as a positive integer. The control interval, `Ts`, determines the duration of each step. The default value is 10 + maximum intervals of delay (if any).

`m`

Control horizon, specified as a scalar integer, 1 ≤ `m` ≤ `p`, or as a vector of blocking factors such that `sum(m)` ≤ `p` (see “Optimization Variables”). The default value is 2.

`W`

Controller tuning weights, specified as a structure. For details about how to specify this structure, see “Weights” on page 1-51.

`MV`

Bounds and other properties of manipulated variables, specified as a 1-by-`nu` structure array, where `nu` is the number of manipulated variables defined in the plant model. For details about how to specify this structure, see “ManipulatedVariables” on page 1-48.

`OV`

Bounds and other properties of the output variables, specified as a 1-by-`ny` structure array, where `ny` is the number of output variables defined in the plant model. For details about how to specify this structure, see “OutputVariables” on page 1-49.

DV

Scale factors and other properties of the disturbance inputs, specified as a 1-by-nd structure array, where nd is the number of disturbance inputs (measured + unmeasured) defined in the plant model. For details about how to specify this structure, see "DisturbanceVariables" on page 1-50.

Models

Plant, input disturbance, and measurement noise models, along with the nominal conditions at which these models were obtained, specified as a structure. For details about how to specify this structure, see "Model" on page 1-53.

Construction and Initialization

To minimize computational overhead, Model Predictive Controller creation occurs in two phases. The first happens at *construction* when you invoke the mpc command, or when you change a controller property. Construction involves simple validity and consistency checks, such as signal dimensions and non-negativity of weights.

The second phase is *initialization*, which occurs when you use the object for the first time in a simulation or analytical procedure. Initialization computes all constant properties required for efficient numerical performance, such as matrices defining the optimal control problem and state estimator gains. Additional, diagnostic checks occur during initialization, such as verification that the controller states are observable.

By default, both phases display informative messages in the command window. You can turn these messages on or off using the mpcverbosity command.

Properties

All of the parameters defining the traditional (implicit) MPC control law are stored in an MPC object, whose properties are listed in the following table.

MPC Controller Object

Property	Description
ManipulatedVariables (or MV or Manipulated or Input)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.

Property	Description
OutputVariables (or OV or Controlled or Output)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
DisturbanceVariables (or DV or Disturbance)	Disturbance scale factors, names, and units
Weights	Weights used in computing the performance (cost) function
Model	Plant, input disturbance, and output noise models, and nominal conditions.
Ts	Controller sample time
Optimizer	Parameters controlling the QP solver
PredictionHorizon	Prediction horizon
ControlHorizon	Number of free control moves or vector of blocking moves
History	Creation time
Notes	Text or comments about the MPC controller object
UserData	Any additional data

ManipulatedVariables

ManipulatedVariables (or MV or Manipulated or Input) is an n_u-dimensional array of structures (n_u = number of manipulated variables), one per manipulated variable. Each structure has the fields described in the following table, where p denotes the prediction horizon. Unless indicated otherwise, numerical values are in engineering units.

Manipulated Variable Structure

Field Name	Content	Default
ScaleFactor	Nonnegative scale factor for this MV	1
Min	1 to p length vector of lower bounds on this MV	-Inf

Field Name	Content	Default
`Max`	1 to p length vector of upper bounds on this MV	`Inf`
`MinECR`	1 to p length vector of nonnegative parameters specifying the `Min` bound softness (0 = hard).	`0` (dimensionless)
`MaxECR`	1 to p length vector of nonnegative parameters specifying the `Max` bound softness (0 = hard).	`0` (dimensionless)
`Target`	1 to p length vector of target values for this MV	`'nominal'`
`RateMin`	1 to p length vector of lower bounds on the interval-to-interval change for this MV	`-Inf`
`RateMax`	1 to p length vector of upper bounds on the interval-to-interval change for this MV	`Inf`
`RateMinECR`	1 to p length vector of nonnegative parameters specifying the `RateMin` bound softness (0 = hard).	`0` (dimensionless)
`RateMaxECR`	1 to p length vector of nonnegative parameters specifying the `RateMax` bound softness (0 = hard).	`0` (dimensionless)
`Name`	Read-only MV signal name (character string)	`InputName` of LTI plant model
`Units`	Read-only MV signal units (character string)	`InputUnit` of LTI plant model

Note Rates refer to the difference $\Delta u(k)=u(k)-u(k-1)$. Constraints and weights based on derivatives du/dt of continuous-time input signals must be properly reformulated for the discrete-time difference $\Delta u(k)$, using the approximation $du/dt \cong \Delta u(k)/T_s$.

OutputVariables

`OutputVariables` (or `OV` or `Controlled` or `Output`) is an n_y-dimensional array of structures (n_y = number of outputs), one per output signal. Each structure has the

fields described in the following table. p denotes the prediction horizon. Unless specified otherwise, values are in engineering units.

Output Variable Structure

Field Name	Content	Default
`ScaleFactor`	Nonnegative scale factor for this OV	1
`Min`	1 to p length vector of lower bounds on this OV	`-Inf`
`Max`	1 to p length vector of upper bounds on this OV	`Inf`
`MinECR`	1 to p length vector of nonnegative parameters specifying the `Min` bound softness (0 = hard).	`1` (dimensionless)
`MaxECR`	1 to p length vector of nonnegative parameters specifying the `Max` bound softness (0 = hard).	`1` (dimensionless)
`Name`	Read-only OV signal name (character string)	`OutputName` of LTI plant model
`Units`	Read-only OV signal units (character string)	`OutputUnit` of LTI plant model

In order to reject constant disturbances due, for instance, to gain nonlinearities, the default measured output disturbance model used in Model Predictive Control Toolbox software is integrated white noise (see "Output Disturbance Model").

DisturbanceVariables

`DisturbanceVariables` (or `DV` or `Disturbance`) is an (n_v+n_d)-dimensional array of structures (n_v = number of measured input disturbances, n_d = number of unmeasured input disturbances). Each structure has the fields described in the following table.

Disturbance Variable Structure

Field Name	Content	Default
`ScaleFactor`	Nonnegative scale factor for this DV	1
`Name`	Read-only DV signal name (character string)	`InputName` of LTI plant model

Field Name	Content	Default
`Units`	Read-only DV signal units (character string)	`InputUnit` of LTI plant model

The order of the disturbance signals within the array `DV` is the following: the first n_v entries relate to measured input disturbances, the last n_d entries relate to unmeasured input disturbances.

Weights

`Weights` is the structure defining the QP weighting matrices. It contains four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see "Standard Cost Function") or the alternative cost function (see "Alternative Cost Function").

Standard Cost Function

The following table lists the content of the four structure fields. In the table, p denotes the prediction horizon, n_u the number of manipulated variables, and n_y the number of output variables.

For the `MV`, `MVRate` and `OV` weights, if you specify fewer than p rows, the last row repeats automatically to form a matrix containing p rows.

Weights for the Standard Cost Function

Field Name (Abbreviations)	Content	Default (dimensionless)
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	(1 to p)-by-n_u dimensional array of nonnegative MV weights	`zeros(1,nu)`
`ManipulatedVariablesRate` (or `MVRate` or `ManipulatedRate` or `InputRate`)	(1 to p)-by-n_u dimensional array of MV-increment weights	`0.1*ones(1,nu)`
`OutputVariables` (or `OV` or `Controlled` or `Output`)	(1 to p)-by-n_y dimensional array of OV weights	`1` (The default for output weights is the following: if $n_u \geq n_y$, all outputs are weighted with unit weight; if $n_u < n_y$, n_u outputs default to 1, with preference given

Field Name (Abbreviations)	Content	Default (dimensionless)
		to measured outputs, and the rest default to 0.)
`ECR`	Scalar weight on the slack variable ε used for constraint softening	`1e5*(max weight)`

Note If all `MVRate` weights are strictly positive, the resulting QP problem is strictly convex. If some `MVRate` weights are zero, the QP Hessian could be positive semidefinite. In order to keep the QP problem strictly convex, when the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12}, the quantity `10*sqrt(eps)` is added to each diagonal term. See "Cost Function".

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To do so, define the fields `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables` as cell arrays, each containing a single positive-semi-definite matrix of the appropriate size. Specifically, `OutputVariables` must be a cell array containing the n_y-by-n_y Q matrix, `ManipulatedVariables` must be a cell array containing the n_u-by-n_u R_u matrix, and `ManipulatedVariablesRate` must be a cell array containing the n_u-by-n_u $R_{\Delta u}$ matrix (see "Alternative Cost Function" and the `mpcweightsdemo` example). You can use diagonal weight matrices for one or more of these fields. If you omit a field, the MPC controller uses the defaults shown in the table above.

For example, you can specify off-diagonal weights, as follows

```
MPCobj.Weights.OutputVariables = {Q};
MPCobj.Weights.ManipulatedVariables = {Ru};
MPCobj.Weights.ManipulatedVariablesRate = {Rdu};
```

where `Q` = Q. `Ru` = R_u, and `Rdu` = $R_{\Delta u}$ are positive semidefinite matrices.

Note You cannot specify non-diagonal weights that vary at each prediction horizon step. The same `Q`, `Ru`, and `Rdu` weights apply at each step.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in “Controller State Estimation”. It is a 1-D structure containing the following fields.

Models Used by MPC

<table>
<tr><th>Field Name</th><th>Content</th><th>Default</th></tr>
<tr><td><code>Plant</code></td><td>LTI model or identified linear model of the plant</td><td>No default</td></tr>
<tr><td><code>Disturbance</code></td><td>LTI model describing expected unmeasured input disturbances</td><td>[ ] (By default, input disturbances are expected to be integrated white noise. To model the signal, an integrator with dimensionless unity gain is added for each unmeasured input disturbance, unless the addition causes the controller to lose state observability. In that case, the disturbance is expected to be white noise, and so, a dimensionless unity gain is added to that channel instead.)</td></tr>
<tr><td><code>Noise</code></td><td>LTI model describing expected noise for output measurements</td><td>[ ] (By default, measurement noise is expected to be white noise with unit variance. To model the signal, a dimensionless unity gain is added for each measured channel.)</td></tr>
<tr><td><code>Nominal</code></td><td>Structure containing the state, input, and output values where <code>Model.Plant</code> is linearized</td><td>The default values of the fields are shown in the following table:
<table>
<tr><th>Field</th><th>Description</th><th>Default</th></tr>
<tr><td><code>X</code></td><td>Plant state at operating point</td><td><code>0</code></td></tr>
<tr><td><code>U</code></td><td>Plant input at operating point, including manipulated variables, measured</td><td><code>0</code></td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Content</th><th>Default</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><th>Field</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>and unmeasured disturbances</td><td></td></tr>
<tr><td>Y</td><td>Plant output at operating point</td><td>0</td></tr>
<tr><td>DX</td><td>For continuous-time models, DX is the state derivative at operating point: DX=f(X,U). For discrete-time models, DX=x(k+1)-x(k)=f(X,U)-X.</td><td>0</td></tr>
</table>
</td></tr>
</table>

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See “MPC Modeling”.

Specify input and output signal types via the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, use the `setmpcsignals` command. Valid signal types are listed in the following tables.

Input Groups in Plant Model

Name (Abbreviations)	Value
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	Indices of manipulated variables in `Model.Plant`
`MeasuredDisturbances` (or `MD` or `Measured`)	Indices of measured disturbances in `Model.Plant`
`UnmeasuredDisturbances` (or `UD` or `Unmeasured`)	Indices of unmeasured disturbances in `Model.Plant`

Output Groups in Plant Model

Name (Abbreviations)	Value
`MeasuredOutputs` (or `MO` or `Measured`)	Indices of measured outputs in `Model.Plant`

Name (Abbreviations)	Value
UnmeasuredOutputs (or UO or Unmeasured)	Indices of unmeasured outputs in Model.Plant

By default, all Model.Plant inputs are manipulated variables, and all outputs are measured.

The structure Nominal contains the values (in engineering units) for states, inputs, outputs, and state derivatives/differences at the operating point where Model.Plant applies. This point is typically a linearization point. The fields are reported in the following table (see also "MPC Modeling").

Nominal Values at Operating Point

Field	Description	Default
X	Plant state at operating point	0
U	Plant input at operating point, including manipulated variables, measured and unmeasured disturbances	0
Y	Plant output at operating point	0
DX	For continuous-time models, DX is the state derivative at operating point: DX=f(X,U). For discrete-time models, DX=$x(k+1)$-$x(k)$=f(X,U)-X.	0

Ts

Sample time of the MPC controller. By default, if Model.Plant is a discrete-time model, Ts = Model.Plant.ts. For continuous-time plant models, specify a controller Ts. The Ts measurement unit is inherited from Model.Plant.TimeUnit.

Optimizer

Parameters for the QP optimization. Optimizer is a structure with the fields reported in the following table.

Optimizer Properties

Field	Description	Default
MaxIter	Maximum number of iterations allowed in the QP solver	'Default'

Field	Description	Default
MinOutputECR	Minimum value allowed for OutputMinECR and OutputMaxECR	0

MaxIter is either a positive scalar integer or the string 'Default'. If set to an integer value, QP solver stops when the iteration number reaches this value. If set to 'Default', the MPC controller automatically determines the maximum iteration number based on the controller specifications.

MinOutputECR is a nonnegative scalar used to specify the minimum allowed ECR for output constraints. By default, MinOutputECR is 0, which means that output constraints are allowed to be hard. If a custom MinOutputECR is specified, when a value smaller than MinOutputECR is provided in the OutputVariables.MinECR or OutputVariables.MaxECR properties of MPC objects, a warning message is issued and the value is raised to MinOutputECR during computation.

PredictionHorizon

PredictionHorizon is the integer number of prediction horizon steps. The control interval, Ts, determines the duration of each step. The default value is 10 + maximum intervals of delay (if any).

ControlHorizon

ControlHorizon is either a number of free control moves, or a vector of blocking moves (see "Optimization Variables"). The default value is 2.

History

History stores the time the MPC controller was created (read only).

Notes

Notes stores text or comments as a cell array of strings.

UserData

Any additional data stored within the MPC controller object.

Examples

Create MPC Controller with Specified Prediction and Control Horizons

Define an MPC controller based on the transfer function model $s+1/(s^2+2s)$, with time $T_s = 0.1$ s. Define bounds on the manipulated variable at $-1 \le u \le 1$, use a 20-interval prediction horizon, and a 3-interval control horizon.

```
Plant = tf([1 1],[1 2 0]);
```

The plant is SISO, so its input must be a manipulated variable and its output must be measured. In general, it is good practice to designate all plant signal types using the `setmpcsignals` command (or the LTI `InputGroup` and `OutputGroup` properties).

In addition, the plant is a continuous-time LTI model, so it is necessary to specify the time of the MPC controller.

```
Ts = 0.1;
MV = struct('Min',-1,'Max',1);
p = 20;
m = 3;
```

Here, `MV` contains only the upper and lower bounds on the manipulated variable. In general, you can specify additional MV properties in general. When you do not specify other properties, their default values apply. Similarly, the fifth input argument (`W`) is empty, so default tuning weights apply.

Create the controller using the values you have specified.

```
MPCobj = mpc(Plant,Ts,p,m,[],MV);
```

- "Design Controller at the Command Line"

More About

- "MPC Modeling"

See Also

`set` | `get` | `setmpcsignals` | `mpcprops` | `mpcverbosity`

Introduced before R2006a

mpcmove

Optimal control action

Syntax

```
u = mpcmove(MPCobj,x,ym,r,v)
[u,Info] = mpcmove(MPCobj,x,ym,r,v)
[u,Info] = mpcmove(MPCobj,x,ym,r,v,Options)
```

Description

`u = mpcmove(MPCobj,x,ym,r,v)` computes the optimal manipulated variable moves, $u(k)$, at the current time. $u(k)$ is calculated given the current estimated extended state, $x(k)$, the measured plant outputs, $y_m(k)$, the output references, $r(k)$, and the measured disturbances, $v(k)$, at the current time k. Call `mpcmove` repeatedly to simulate closed-loop model predictive control.

`[u,Info] = mpcmove(MPCobj,x,ym,r,v)` returns additional information regarding the model predictive controller in the second output argument Info.

`[u,Info] = mpcmove(MPCobj,x,ym,r,v,Options)` overrides default constraints and weights settings in MPCobj with the values specified by Options, an `mpcmoveopt` object. Use Options to provide run-time adjustment in constraints and weights during the closed-loop simulation.

Input Arguments

`MPCobj`

`mpc` object that defines the model predictive controller.

`x`

`mpcstate` object that defines the current controller state.

Before you begin a simulation with `mpcmove`, initialize the controller state using `x = mpcstate(MPCobj)`. Then, modify the default properties of x as appropriate.

If you are using default state estimation, `mpcmove` expects x to represent `x[n|n-1]`. The `mpcmove` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a steady-state Kalman filter.

If you are using custom state estimation, `mpcmove` expects x to represent `x[n|n]`. Therefore, prior to each `mpcmove` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

ym

1-by-n_{ym} vector of current measured output values at time k, where n_{ym} is the number of measured outputs.

If you are using custom state estimation, set `ym = []`.

r

Plant output reference values, specified as a p-by-n_y array, where p is the prediction horizon of MPCobj and n_y is the number of outputs. Row `r(i,:)` defines the reference values at step i of the prediction horizon.

r must contain at least one row. If r contains fewer than p rows, `mpcmove` duplicates the last row to fill the p-by-n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, r must contain the anticipated variations, ideally for p steps.

v

Current and anticipated measured disturbances, specified as a p-by-n_{md} array, where p is the prediction horizon of MPCobj and n_{md} is the number of measured disturbances. Row `v(i,:)` defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use v = `[]`.

v must contain at least one row. If v contains fewer than p rows, `mpcmove` duplicates the last row to fill the p-by-n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

Options

Override values for selected properties of MPCobj, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmove` time instant only. Using Options yields the same result as redefining or modifying MPCobj before each call to `mpcmove`, but involves considerably less overhead. Using Options is equivalent to using an MPC Controller Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

u

1-by-n_u array of optimal manipulated variable moves, where n_u is the number of manipulated variables.

`mpcmove` holds u at its most recent successful solution if the QP solver fails to find a solution for the current time k.

Info

Information regarding the model predictive controller, returned as a structure containing the following fields.

Uopt

Optimal manipulated variable adjustments (moves), returned as a p+1-by-n_u array, where p is the prediction horizon of MPCobj and n_u is the number of manipulated variables.

The first row of `Info.Uopt` is identical to the output argument u, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal

values at time `k+i-1`, for `i = 1,...,p+1`. The `mpcmove` command does not calculate optimal control moves at time `k+p`, and therefore sets `Uopt(p+1,:)` to `NaN`.

Yopt

Predicted output variable sequence, returned as a p+1-by-n_y array, where p is the prediction horizon of MPCobj and n_x is the number of outputs.

The first row of `Info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

Xopt

Predicted state variable sequence, returned as a p+1-by-n_x array, where p is the prediction horizon of MPCobj and n_x is the number of states.

The first row of `Info.Xopt` contains the current states at time `k` as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time `k+i-1`, for `i = 1,...,p+1`.

Topt

Time intervals, returned as a p+1-by-a vector. `Topt(1)` = 0, representing the current time. Subsequent time steps `Topt(i)` are given by `Ts*(i-1)`, where `Ts = MPCobj.Ts`, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

Slack

Slack variable, ε, used in constraint softening, returned as 0 or a positive scalar value.

- ε = 0 — All constraints were satisfied for the entire prediction horizon.
- ε > 0 — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

Iterations — QP solution result

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- `Iterations` > 0 — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- `Iterations` = 0 — QP problem could not be solved in the allowed maximum number of iterations.
- `Iterations` = –1 — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- `Iterations` = –2 — Numerical error occurred when solving the QP problem.

QPCode

QP solution status, returned as one of the following strings:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — QP solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — QP solver failed to converge (`Iterations` = 0)

Cost — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. See "Optimization Problem" for details.

The cost value is only meaningful when `QPCode = 'feasible'`.

Examples

Analyze Closed-Loop Response

Perform closed-loop simulation of a plant with one MV and one measured OV.

Define a plant model and create a model predictive controller with MV constraints.

```
ts = 2;
Plant = ss(0.8,0.5,0.25,0,ts);
MPCobj = mpc(Plant);
```

```
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max = 2;
```

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

Initialize an `mpcstate` object for simulation. Use the default state properties.

```
x = mpcstate(MPCobj);
```

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

Set the reference signal. There is no measured disturbance.

```
r = 1;
```

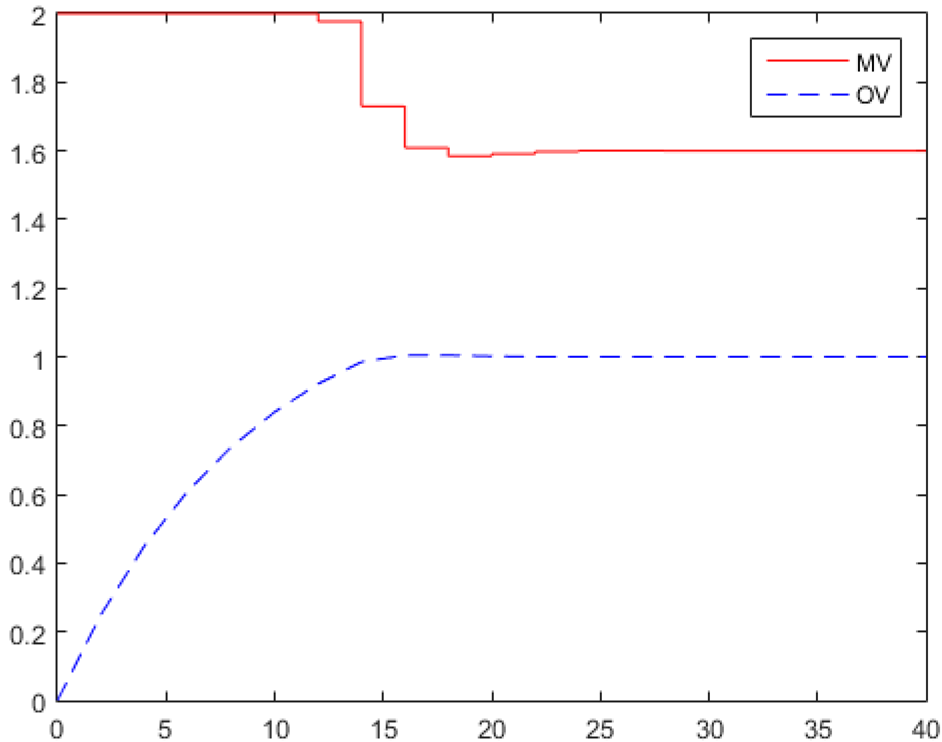
Simulate the closed-loop response by calling `mpcmove` iteratively.

```
t = [0:ts:40];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
  % simulated plant and predictive model are identical
  y(i) = 0.25*x.Plant;
  u(i) = mpcmove(MPCobj,x,y(i),r);
end
```

`y` and `u` store the OV and MV values.

Analyze the result.

```
[ts,us] = stairs(t,u);
plot(ts,us,'r-',t,y,'b--');
legend('MV','OV');
```



Modify the MV upper bound as the simulation proceeds using an `mpcmoveopt` object.

```
MPCopt = mpcmoveopt;
MPCopt.MVMin = -2;
MPCopt.MVMax = 2;
```

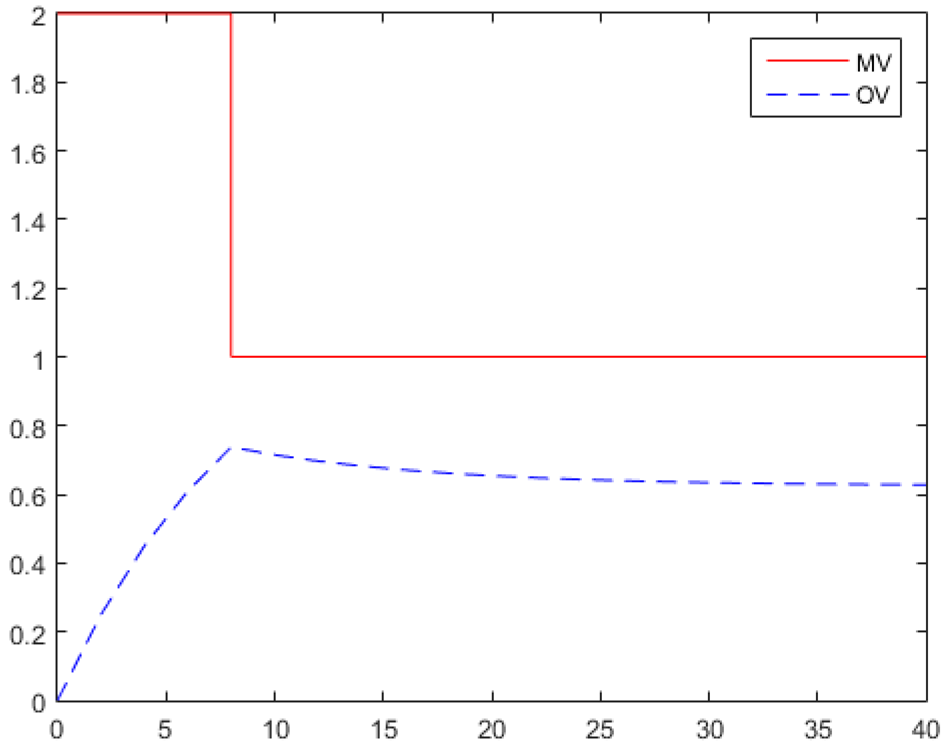
Simulate the closed-loop response and introduce the real-time upper limit change at eight seconds (the fifth iteration step).

```
x = mpcstate(MPCobj);
y = zeros(N,1);
u = zeros(N,1);
for i=1:N
  % simulated plant and predictive model are identical
```

```
  y(i) = 0.25*x.Plant;
  if i == 5
   MPCopt.MVMax = 1;
  end
  u(i) = mpcmove(MPCobj,x,y(i),r,[],MPCopt);
end
```

Analyze the result.

```
[ts,us] = stairs(t,u);
plot(ts,us,'r-',t,y,'b--');
legend('MV','OV');
```



Evaluate Scenario at Specific Time Instant

Define a plant model.

```
ts = 2;
Plant = ss(0.8,0.5,0.25,0,ts);
```

Create a model predictive controller with MV and MVRate constraints. The prediction horizon is ten intervals. The control horizon is blocked.

```
MPCobj = mpc(Plant, ts, 10, [2 3 5]);
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max = 2;
MPCobj.MV(1).RateMin = -1;
```

```
MPCobj.MV(1).RateMax = 1;
```

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

Initialize an mpcstate object for simulation from a particular state.

```
x = mpcstate(MPCobj);
x.Plant = 2.8;
x.LastMove = 0.85;
```

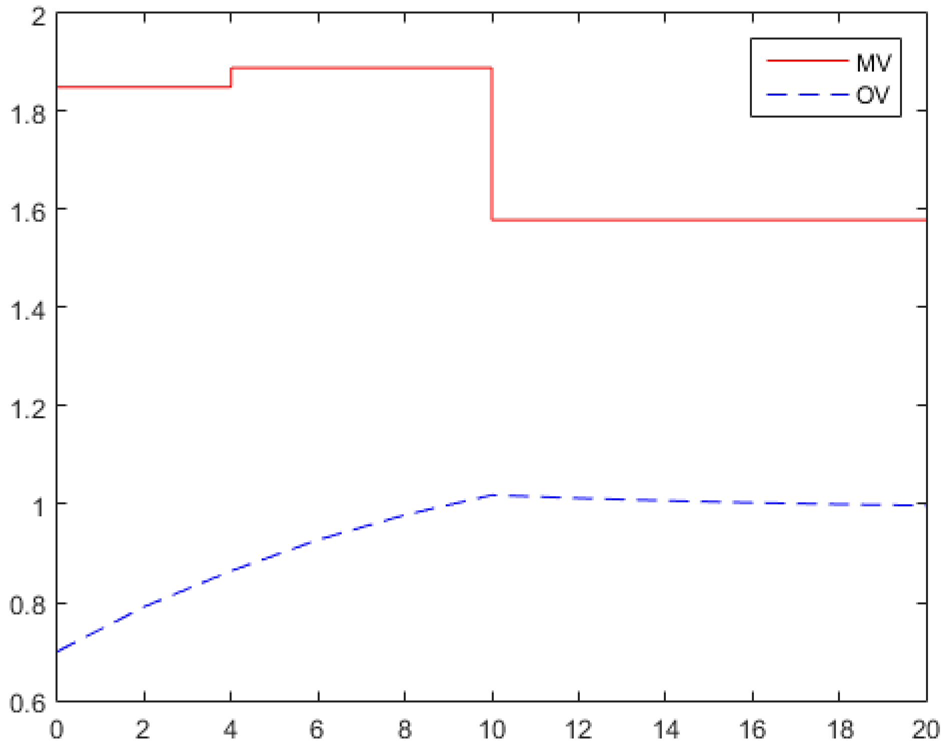
```
-->Integrated white noise added on measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Compute the optimal control at current time.

```
y = 0.25*x.Plant;
r = 1;
[u,Info] = mpcmove(MPCobj,x,y,r);
```

Analyze the predicted optimal sequences.

```
[ts,us] = stairs(Info.Topt,Info.Uopt);
plot(ts,us,'r-',Info.Topt,Info.Yopt,'b--');
legend('MV','OV');
```



`plot` ignores `Info.Uopt(end)` as it is NaN.

Examine the optimal cost.

```
Info.Cost
```

```
ans =

    0.0793
```

- MPC Control with Anticipative Action (Look-Ahead)
- MPC Control with Input Quantization Based on Comparing the Optimal Costs

- Analysis of Control Sequences Optimized by MPC on a Double Integrator System

Alternatives

- Use `sim` for plant mismatch and noise simulation when not using run-time constraints or weight changes.
- Use `mpctool` to graphically and interactively combine model predictive design and simulation.
- Use the MPC Controller block in Simulink and for code generation.

More About

Tips

- `mpcmove` updates x.
- If ym, r or v is specified as `[]`, `mpcmove` uses the appropriate `MPCobj.Model.Nominal` value instead.
- To view the predicted optimal behavior for the entire prediction horizon, plot the appropriate sequences provided in Info.
- To determine the optimization status, check `Info.Iterations` and `Info.QPCode`.

See Also

`mpc` | `mpcmoveopt` | `mpcstate` | `review` | `sim` | `setEstimator` | `getEstimator`

mpcmoveAdaptive

Compute optimal control with prediction model updating

Syntax

```
u = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)
[u,info] = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)
[u,info] = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v,opt)
```

Description

`u = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)` computes the optimal manipulated variable moves at the current time. This result depends on the properties contained in the MPC controller, the controller states, an updated prediction model, and nominal values. The result also depends on the measured output variables, the output references (setpoints), and the measured disturbance inputs. `mpcmoveAdaptive` updates the controller state, x, when using default state estimation. Call `mpcmoveAdaptive` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)` returns additional details about the solution in a structure. To view the predicted optimal trajectory for the entire prediction horizon, plot the sequences provided in info. To determine whether the optimal control calculation completed normally, check `info.Iterations` and `info.QPCode`.

`[u,info] = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v,opt)` alters selected controller settings using options you specify with `mpcmoveopt`. These changes apply for the current time instant only, allowing a command-line simulation using `mpcmoveAdaptive` to mimic the Adaptive MPC Controller block in Simulink in a computationally efficient manner.

Input Arguments

`MPCobj` — MPC controller

MPC controller object

MPC controller, specified as an implicit MPC controller object. Use the `mpc` command to create the MPC controller.

`x` — Current MPC controller state

`mpcstate` object

Current MPC controller state, specified as an `mpcstate` object.

Before you begin a simulation with `mpcmoveAdaptive`, initialize the controller state using `x = mpcstate(MPCobj)`. Then, modify the default properties of x as appropriate.

If you are using default state estimation, `mpcmoveAdaptive` expects x to represent `x[n|n-1]`. The `mpcmoveAdaptive` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a linear time-varying Kalman filter.

If you are using custom state estimation, `mpcmoveAdaptive` expects x to represent `x[n|n]`. Therefore, prior to each `mpcmoveAdaptive` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

`Plant` — Updated prediction model

discrete-time state-space model

Updated prediction model, specified as a delay-free, discrete-time state-space (`ss`) model. This plant is the update to `MPCobj.Model.Plant`. The sampling time `Plant.Ts` must match the controller sampling time `MPCobj.Ts`. The updated prediction model must define the same states as `MPCobj.Model.Plant`. In order to minimize the likelihood of updating errors, `MPCobj.Model.Plant` should be a discrete time, state-space model with sampling period `MPCobj.Ts` and all delays converted to states (e.g., using `absorbDelay`).

`Nominal` — Updated nominal conditions

structure

Updated nominal conditions, specified as a structure having fields `U`, `Y`, `X`, and `DX`. These are the same fields as `MPCobj.Model.Nominal`. See “MPC Controller Object” for a description of the fields.

If Nominal = `[]`, or if a field is missing or empty, `mpcmoveAdaptive` uses the corresponding `MPCobj.Model.Nominal` value.

`ym` — Current measured outputs

vector

Current measured outputs, specified as a 1-by-n_{ym} vector. n_{ym} is the number of measured outputs. If you are using custom state estimation, ym is ignored. If you set ym = `[]`, then `mpcmoveAdaptive` uses the appropriate nominal value.

`r` — Plant output reference values

array

Plant output reference values, specified as a p-by-n_y array, where p is the prediction horizon of MPCobj and n_y is the number of outputs. Row `r(i,:)` defines the reference values at step i of the prediction horizon.

r must contain at least one row. If r contains fewer than p rows, `mpcmoveAdaptive` duplicates the last row to fill the p-by-n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

If you set r = `[]`, then `mpcmoveAdaptive` uses the appropriate nominal value.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, r must contain the anticipated variations, ideally for p steps.

`v` — Current and anticipated measured disturbances

array

Current and anticipated measured disturbances, specified as a p-by-n_{md} array, where p is the prediction horizon of MPCobj and n_{md} is the number of measured disturbances. Row `v(i,:)` defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use v = `[]`.

v must contain at least one row. If v contains fewer than p rows, `mpcmoveAdaptive` duplicates the last row to fill the p-by-n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

If you set v = `[]`, then `mpcmoveAdaptive` uses the appropriate nominal value.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

`opt` — Override values for selected controller properties

`mpcmoveopt` object

Override values for selected properties of MPCobj, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmoveAdaptive` time instant only. Using opt yields the same result as redefining or modifying MPCobj before each call to `mpcmoveAdaptive`, but involves considerably less overhead. Using opt is equivalent to using an Adaptive MPC Controller Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

`u` — Optimal manipulated variable moves

array

Optimal manipulated variable adjustments (moves), returned as a 1-by-n_u array of optimal manipulated variable moves, where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, `mpcmoveAdaptive` holds u at its most recent value, `x.LastMove`.

`info` — Solution details

structure

Solution details, returned as a structure containing the following fields.

`Uopt` — Optimal manipulated variable adjustments (moves)

array

Optimal manipulated variable adjustments (moves), returned as a p+1-by-n_u array, where p is the prediction horizon of MPCobj and n_u is the number of manipulated variables.

The first row of `info.Uopt` is identical to the output argument u, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal values at time `k+i-1`, for `i = 1,...,p+1`. The `mpcmoveAdaptive` command does not calculate optimal control moves at time `k+p`, and therefore sets `Uopt(p+1,:)` to `NaN`.

`Yopt` — Predicted output variable sequence

array

Predicted output variable sequence, returned as a p+1-by-n_y array, where p is the prediction horizon of MPCobj and n_x is the number of outputs.

The first row of `info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

`Xopt` — Predicted state variable sequence
array

Predicted state variable sequence, returned as a p+1-by-n_x array, where p is the prediction horizon of MPCobj and n_x is the number of states.

The first row of `info.Xopt` contains the current states at time `k` as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time `k+i-1`, for `i = 1,...,p+1`.

`Topt` — Time intervals
vector

Time intervals, returned as a p+1-by-a vector. `Topt(1)` = 0, representing the current time. Subsequent time steps `Topt(i)` are given by `Ts*(i-1)`, where `Ts = MPCobj.Ts`, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

`Slack` — Slack variable
0 | positive scalar

Slack variable, ε, used in constraint softening, returned as 0 or a positive scalar value.

- ε = 0 — All constraints were satisfied for the entire prediction horizon.
- ε > 0 — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

`Iterations` — QP solution result
positive integer | 0 | –1 | –2

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- `Iterations` > 0 — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- `Iterations` = 0 — QP problem could not be solved in the allowed maximum number of iterations.
- `Iterations` = –1 — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- `Iterations` = –2 — Numerical error occurred when solving the QP problem.

`QPCode` — QP solution status

`'feasible'` | `'infeasible'` | `'unrealiable'`

QP solution status, returned as one of the following strings:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — QP solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — QP solver failed to converge (`Iterations` = 0)

`Cost` — Objective function cost

nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. See "Optimization Problem" for details.

The cost value is only meaningful when `QPCode = 'feasible'`.

More About

Tips

- If the prediction model is time invariant, use `mpcmove`.
- Use the Adaptive MPC Controller Simulink block for simulations and code generation.

- "Optimization Problem"

See Also

`getEstimator` | `mpc` | `mpcmove` | `mpcmoveopt` | `mpcstate` | `review` | `setEstimator` | `sim`

mpcmoveExplicit

Compute optimal control using explicit MPC

Syntax

```
u = mpcmoveExplicit(EMPCobj,x,ym,r,v)
[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v)
[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v,MVused)
```

Description

`u = mpcmoveExplicit(EMPCobj,x,ym,r,v)` computes the optimal manipulated variable moves at the current time using an explicit model predictive control law. This result depends on the properties contained in the explicit MPC controller and the controller states. The result also depends on the measured output variables, the output references (setpoints), and the measured disturbance inputs. `mpcmoveExplicit` updates the controller state, x, when using default state estimation. Call `mpcmoveExplicit` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v)` returns additional details about the computation in a structure. To determine whether the optimal control calculation completed normally, check the data in info.

`[u,info] = mpcmoveExplicit(EMPCobj,x,ym,r,v,MVused)` specifies the manipulated variable values used in the previous `mpcmoveExplicit` command, allowing a command-line simulation to mimic the Explicit MPC Controller Simulink block with the optional external MV input signal.

Examples

- "Explicit MPC Control of a Single-Input-Single-Output Plant"

Input Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller to simulate, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

x — Current MPC controller state

`mpcstate` object

Current MPC controller state, specified as an `mpcstate` object.

Before you begin a simulation with `mpcmoveExplicit`, initialize the controller state using `x = mpcstate(EMPCobj)`. Then, modify the default properties of x as appropriate.

If you are using default state estimation, `mpcmoveExplicit` expects x to represent `x[n|n-1]`. The `mpcmoveExplicit` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a linear time-varying Kalman filter.

If you are using custom state estimation, `mpcmoveExplicit` expects x to represent `x[n|n]`. Therefore, prior to each `mpcmoveExplicit` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

ym — Current measured outputs

vector

Current measured outputs, specified as a 1-by-n_{ym} vector. n_{ym} is the number of measured outputs. If you are using custom state estimation, ym is ignored. If you set ym = `[]`, then `mpcmoveExplicit` uses the appropriate nominal value.

r — Plant output reference values

vector

Plant output reference values, specified as a vector of n_y values. `mpcmoveExplicit` uses a constant reference for the entire prediction horizon. In contrast to `mpcmove` and `mpcmoveAdaptive`, `mpcmoveExplicit` does not support reference previewing.

If you set r = `[]`, then `mpcmoveExplicit` uses the appropriate nominal value.

v — Current and anticipated measured disturbances

vector

Current and anticipated measured disturbances, specified as a vector of n_{md} values. In contrast to `mpcmove` and `mpcmoveAdaptive`, `mpcmoveExplicit` does not support

disturbance previewing. If your plant model does not include measured disturbances, use v = [].

`MVused` — Manipulated variable values from previous interval

vector

Manipulated variable values applied to the plant during the previous control interval, specified as a vector of n_u values. If this is the first `mpcmoveExplicit` command in a simulation sequence, omit this argument. Otherwise, if the MVs calculated by `mpcmoveExplicit` in the previous interval were overridden, set `MVused` to the correct values in order to improve the controller state estimation accuracy. If you omit `MVused`, `mpcmoveExplicit` assumes `MVused = x.LastMove`.

Output Arguments

u — Optimal manipulated variable moves

array

Optimal manipulated variable adjustments (moves), returned as a 1-by-n_u array of optimal manipulated variable moves, where n_u is the number of manipulated variables.

If the controller fails to find a solution, `mpcmoveExplicit` holds u at its most recent value, `x.LastMove`.

`info` — Explicit MPC solution status

structure

Explicit MPC solution status, returned as a structure having the following fields.

`ExitCode` — Solution status

1 | 0 | –1

Solution status, returned as one of the following values:

- 1 — Successful solution.
- 0 — Failure. One or more controller input parameters is out of range.
- –1 — Undefined. Parameters are in range but an extrapolation must be used.

`Region` — Region to which current controller input parameters belong

positive integer | 0

Region to which current controller input parameters belong, returned as either a positive integer or 0. The integer value is the index of the polyhedron (region) to which the current controller input parameters belong. If the solution failed, `Region` = 0.

More About

Tips

- Use the Explicit MPC Controller Simulink block for simulations and code generation.

- “Explicit MPC”
- “Design Workflow for Explicit MPC”

mpcmoveMultiple

Compute gain-scheduling MPC control action at a single time instant

Syntax

```
u = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v,opt)
```

Description

`u = mpcmoveMultiple(MPCArray,states,index,ym,r,v)` computes the optimal manipulated variable moves at the current time using a model predictive controller selected by index from an array of MPC controllers. This results depends upon the properties contained in the MPC controller and the controller states. The result also depends on the measured plant outputs, the output references (setpoints), and the measured disturbance inputs. `mpcmoveMultiple` updates the controller state when default state estimation is used. Call `mpcmoveMultiple` repeatedly to simulate closed-loop model predictive control.

`[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v)` returns additional details about the computation in a structure. To determine whether the optimal control calculation completed normally, check the data in info.

`[u,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v,opt)` alters selected controller settings using options you specify with `mpcmoveopt`. These changes apply for the current time instant only, allowing a command-line simulation using `mpcmoveMultiple` to mimic the Multiple MPC Controllers block in Simulink in a computationally efficient manner.

Input Arguments

`MPCArray` — MPC controllers

cell array of MPC controller objects

MPC controllers to simulate, specified as a cell array of traditional (implicit) MPC controller objects. Use the `mpc` command to create the MPC controllers.

All the controllers in MPCArray must use either default state estimation or custom state estimation. Mismatch is not permitted.

`states` — Current MPC controller states

cell array of `mpcstate` objects

Current controller states for each MPC controller in MPCArray, specified as a cell array of `mpcstate` objects.

Before you begin a simulation with `mpcmoveMultiple`, initialize each controller state using `x = mpcstate(MPCobj)`. Then, modify the default properties of each state as appropriate.

If you are using default state estimation, `mpcmoveAdaptive` expects `x` to represent `x[n|n-1]` (where `x` is one entry in states, the current state of one MPC controller in MPCArray). The `mpcmoveMultiple` command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update `x` at all. The default state estimator employs a steady-state Kalman filter.

If you are using custom state estimation, `mpcmoveMultiple` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmoveMultiple` command, you must set `x.Plant`, `x.Disturbance`, and `x.Noise` to the best estimates of these states (using the latest measurements) at the current control interval.

`index` — Index of selected controller

positive integer

Index of selected controller in the cell array MPCArray, specified as a positive integer.

`ym` — Current measured outputs

vector

Current measured outputs, specified as a 1-by-n_{ym} vector. n_{ym} is the number of measured outputs. If you are using custom state estimation, ym is ignored. If you set ym = `[]`, then `mpcmoveMultiple` uses the appropriate nominal value.

`r` — Plant output reference values

array

Plant output reference values, specified as a p-by-n_y array, where p is the prediction horizon of the selected controller and n_y is the number of outputs. Row `r(i,:)` defines the reference values at step i of the prediction horizon.

r must contain at least one row. If r contains fewer than p rows, `mpcmoveMultiple` duplicates the last row to fill the p-by-n_y array. If you supply exactly one row, therefore, a constant reference applies for the entire prediction horizon.

If you set r = `[]`, then `mpcmoveMultiple` uses the appropriate nominal value.

To implement reference previewing, which can improve tracking when a reference varies in a predictable manner, r must contain the anticipated variations, ideally for p steps.

v — Current and anticipated measured disturbances

array

Current and anticipated measured disturbances, specified as a p-by-n_{md} array, where p is the prediction horizon of the selected controller and n_{md} is the number of measured disturbances. Row `v(i,:)` defines the expected measured disturbance values at step i of the prediction horizon.

Modeling of measured disturbances provides feedforward control action. If your plant model does not include measured disturbances, use v = `[]`.

v must contain at least one row. If v contains fewer than p rows, `mpcmoveMultiple` duplicates the last row to fill the p-by-n_{md} array. If you supply exactly one row, therefore, a constant measured disturbance applies for the entire prediction horizon.

If you set v = `[]`, then `mpcmoveMultiple` uses the appropriate nominal value.

To implement disturbance previewing, which can improve tracking when a disturbance varies in a predictable manner, v must contain the anticipated variations, ideally for p steps.

`opt` — Override values for selected controller properties

`mpcmoveopt` object

Override values for selected properties of the selected MPC controller, specified as an options object you create with `mpcmoveopt`. These options apply to the current `mpcmoveMultiple` time instant only. Using opt yields the same result as redefining or modifying the selected controller before each call to `mpcmoveMultiple`, but involves

considerably less overhead. Using opt is equivalent to using a Multiple MPC Controllers Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

u — Optimal manipulated variable moves

array

Optimal manipulated variable adjustments (moves), returned as a 1-by-n_u array of optimal manipulated variable moves, where n_u is the number of manipulated variables.

If the controller includes constraints and the QP solver fails to find a solution, `mpcmoveMultiple` holds u at its most recent value, `x.LastMove`.

`info` — Solution details

structure

Solution details, returned as a structure containing the following fields.

`Uopt` — Optimal manipulated variable adjustments (moves)

array

Optimal manipulated variable adjustments (moves), returned as a p+1-by-n_u array, where p is the prediction horizon of the selected controller and n_u is the number of manipulated variables.

The first row of `info.Uopt` is identical to the output argument u, which is the adjustment applied at the current time, `k`. `Uopt(i,:)` contains the predicted optimal values at time `k+i-1`, for `i = 1,...,p+1`. The `mpcmoveMultiple` command does not calculate optimal control moves at time `k+p`, and therefore sets `Uopt(p+1,:)` to `NaN`.

`Yopt` — Predicted output variable sequence

array

Predicted output variable sequence, returned as a p+1-by-n_y array, where p is the prediction horizon of the selected controller and n_x is the number of outputs.

The first row of `info.Yopt` contains the current outputs at time `k` after state estimation. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

`Xopt` — Predicted state variable sequence

array

Predicted state variable sequence, returned as a p+1-by-n_x array, where p is the prediction horizon of the selected controller and n_x is the number of states.

The first row of `info.Xopt` contains the current states at time `k` as determined by state estimation. `Xopt(i,:)` contains the predicted state values at time `k+i-1`, for `i = 1,...,p+1`.

`Topt` — Time intervals

vector

Time intervals, returned as a p+1-by-a vector. `Topt(1)` = 0, representing the current time. Subsequent time steps `Topt(i)` are given by `Ts*(i-1)`, where `Ts = MPCobj.Ts`, the controller sampling time.

Use `Topt` when plotting `Uopt`, `Xopt`, or `Yopt` sequences.

`Slack` — Slack variable

0 | positive scalar

Slack variable, ε, used in constraint softening, returned as 0 or a positive scalar value.

- ε = 0 — All constraints were satisfied for the entire prediction horizon.
- ε > 0 — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by your ECR values for each constraint).

See “Optimization Problem” for more information.

`Iterations` — QP solution result

positive integer | 0 | –1 | –2

QP solution result, returned as a positive integer or one of several values with specific meanings as follows.

- `Iterations` > 0 — Number of iterations needed to solve the quadratic programming (QP) problem that determines the optimal sequences.
- `Iterations` = 0 — QP problem could not be solved in the allowed maximum number of iterations.

- `Iterations` = –1 — QP problem was infeasible. A QP problem is infeasible if no solution can satisfy all the hard constraints.
- `Iterations` = –2 — Numerical error occurred when solving the QP problem.

`QPCode` — QP solution status

`'feasible'` | `'infeasible'` | `'unrealiable'`

QP solution status, returned as one of the following strings:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — QP solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — QP solver failed to converge (`Iterations` = 0)

`Cost` — Objective function cost

nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. See "Optimization Problem" for details.

The cost value is only meaningful when `QPCode = 'feasible'`.

More About

Tips

- Use the Multiple MPC Controllers Simulink block for simulations and code generation.

See Also

`generateExplicitMPC` | `getEstimator` | `mpcmove` | `mpcstate` | `review` | `setEstimator` | `sim`

mpcmoveopt

Options set for `mpcmove` and `mpcmoveAdaptive`

Syntax

```
options = mpcmoveopt
```

Description

`options = mpcmoveopt` creates an empty `mpcmoveopt` object. You can set one or more of its properties using dot notation, and then use the object with `mpcmove` or `mpcmoveAdaptive` to simulate run-time adjustment of selected controller properties, such as tuning weights and bounds.

`mpcmoveopt` property dimensions must be consistent with the number of manipulated variables (`nu`) and output variables (`ny`) defined in the `mpc` or `mpcAdaptive` controller you are simulating.

In general, if you do not specify a value for one of the `mpcmoveopt` properties, it defaults to the corresponding built-in value of the simulated controller.

Output Arguments

`options`

Options for the `mpcmove` or `mpcmoveAdaptive` command with the following fields:

- `OutputWeights` — Output variable tuning weights, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. These replace the controller's `Weight.OutputVariables` property. The weights must be nonnegative, finite real values.
- `MVWeights` — Manipulated variable tuning weights, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. These replace the controller's

`Weight.ManipulatedVariables` property. The weights must be nonnegative, finite real values.

- `MVRateWeights` — Manipulated variable rate tuning weights, specified as a 1-by-`nu` vector, where nu is the number of manipulated variables. These replace the controller's `Weight.ManipulatedVariablesRate` property. The weights must be nonnegative, finite real values.
- `ECRWeight` — Weight on the slack variable used for constraint softening, specified as a finite, real scalar. This value replaces the controller's `Weight.ECR` property.
- `OutputMin` — Lower bounds on the output variables, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. `OutputMin(i)` replaces the controller's `OutputVariables(i).Min` property, for `i = 1,...,ny`.
- `OutputMax` — Upper bounds on the output variables, specified as a 1-by-`ny` vector, where `ny` is the number of output variables. `OutputMax(i)` replaces the controller's `OutputVariables(i).Max` property, for `i = 1,...,ny`.
- `MVMin` — Lower bounds on the manipulated variables, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. `MVMin(i)` replaces the controller's `ManipulatedVariables(i).Min` property, for `i = 1,...,nu`.
- `MVMax` — Upper bounds on the manipulated variables, specified as a 1-by-`nu` vector, where `nu` is the number of manipulated variables. `MVMax(i)` replaces the controller's `ManipulatedVariables(i).Max` property, for `i = 1,...,nu`.
- `MVUsed` — Manipulated variable values used in the plant during the previous control interval, specified as a 1-by-`nu` vector. This property mimics the MPC Controller or Adaptive MPC Controller Simulink blocks' external MV signal. If you do not provide an `MVUsed` value, the controller uses the `LastMove` property of `mpcstate`.
- `OnlyComputeCost` — Logical value that controls whether the optimal sequence is to be calculated and exported.

  - `0` (default) causes the controller to return the predicted optimal policy in addition to the objective function cost value.
  - `1` causes the controller to return the objective function cost only, which saves computational effort.

Examples

Simulation with Varying Controller Property

Vary a manipulated variable upper bound during a simulation.

Define the plant, which includes a 4-second input delay. Convert to a delay-free, discrete, state-space model using a 2-second control interval. Create the corresponding default controller and then specify MV bounds at +/-2.

```
ts = 2;
Plant = absorbDelay(c2d(ss(tf(0.8,[5 1],'InputDelay',4)),ts));
MPCobj = mpc(Plant, ts);
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max = 2;
```

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

Create an empty `mpcmoveopt` object. During simulation, you can set properties of the object to specify controller parameters.

```
options = mpcmoveopt;
```

Pre-allocate storage and initialize the controller state.

```
v = [];
t = [0:ts:20];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
x = mpcstate(MPCobj);
```

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

Use `mpcmove` to simulate the following:

- Reference (setpoint) step change from initial condition $r = 0$ to $r = 1$ (servo response).
- MV upper bound step decrease from 2 to 1, occurring at $t = 10$.

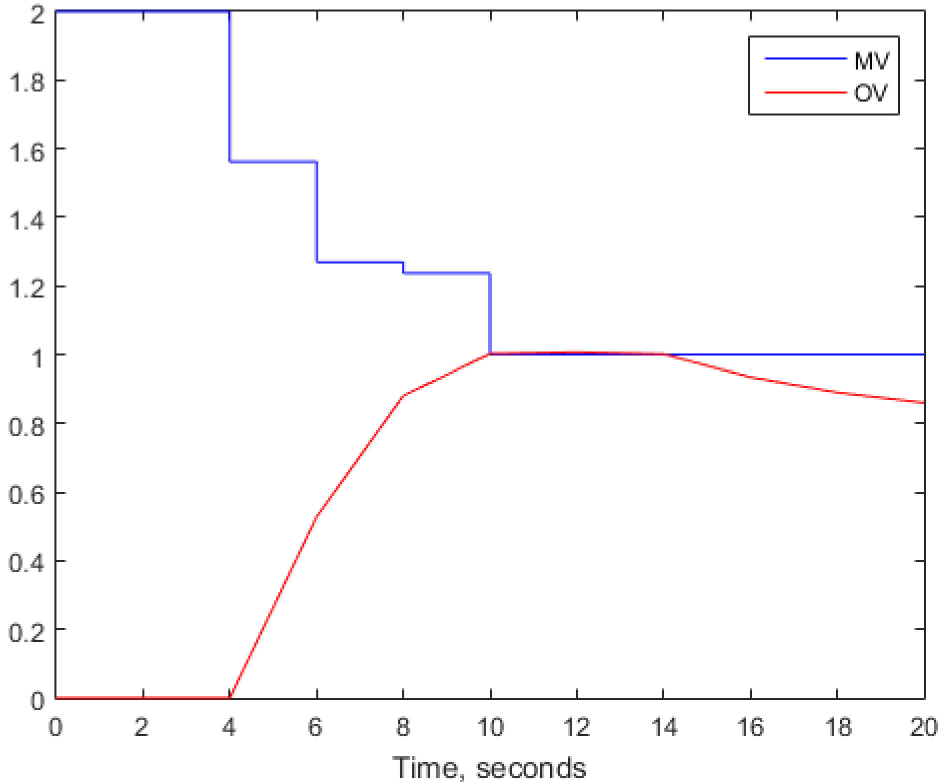
```
r = 1;
for i = 1:N
    y(i) = Plant.c*x.Plant;
    if t(i) >= 10
        options.MVMax = 1;
    end
```

```
    [u(i),Info] = mpcmove(MPCobj,x,y(i),r,v,options);
end
```

As the loop executes, the value of `options.MVMax` is reset to 1 for all iterations that occur after $t = 10$. Prior to that iteration, `options.MVMax` is empty. Therefore, the controller's value for `MVMax` is used, `MPCobj.MV(1).Max = 2`.

Plot the results of the simulation.

```
[ts,us] = stairs(t,u);
plot(ts,us,'b-',t,y,'r-')
legend('MV','OV');
xlabel(sprintf('Time, %s',Plant.TimeUnit))
```



From the plot, you can observe that the original MV upper bound is active until t = 4. After the input delay of 4 seconds, the output variable (OV) moves smoothly to its new target of r = 1. reaching the target at t = 10. The new MV bound imposed at t = 10 becomes avtive immediately. This forces the OV below its target, after the input delay elapses.

Now assume that you want to impose an OV upper bound at a specified location relative to the OV target. Consider the following constraint design command:

```
MPCobj.OV(1).Max = [Inf,Inf,0.4,0.3,0.2];
```

This is a horizon-varying constraint. The known input delay makes it impossible for the controller to satisfy an OV constraint prior to the third prediction-horizon step. Therefore, a finite constraint during the first two steps would be poor practice. For illustrative purposes, the above constraint also decreases from 0.4 at step 3 to 0.2 at step 5 and thereafter.

The following commands produce the same results shown in the previous plot. The OV constraint is never active because it is being varied in concert with the setpoint, r.

```
x = mpcstate(MPCobj);
OPTobj = mpcmoveopt;
for i = 1:N
    y(i) = Plant.c*x.Plant;
    if t(i) >= 10
        OPTobj.MVMax = 1;
    end
    OPTobj.OutputMax = r + 0.4;
    [u(i), Info] = mpcmove(MPCobj, x, y(i), r, v, OPTobj);
end
```

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

The scalar value $r + 0.4$ replaces the first finite value in the `MPCobj.OV(1).Max` vector, and the remaining finite values adjust to maintain the original profile, i.e., the numerical difference between these values is unchanged. r = 1 for the simulation, so the above use of the mpcmoveopt object is equivalent to the command

```
MPCobj.OV(1).Max = [Inf, Inf, 1.4, 1.3, 1.2];
```

The use of the `mpcmoveopt` object involves much less computational overhead, however.

Alternatives

The `mpcmoveopt` object is an optional feature of the `mpcmove` and `mpcmoveAdaptive` commands. The alternative is to redefine the controller and/or state object prior to each command invocation, but this involves considerable overhead.

More About

Tips

- `mpcmoveopt` cannot constrain a variable that was unconstrained in the controller creation step. The controller ignores any such specifications. Similarly, you cannot eliminate a constraint defined during controller creation, but you can change it to a very large (or small) value such that it is unlikely to become active.
- If the controller design includes a vector constraint, the run-time `mpcmoveopt` value replaces the first finite entry, and the remaining values shift to retain the same constraint profile. See "Simulation with Varying Controller Property" on page 1-88.

See Also

`mpc` | `mpcmove` | `setconstraint` | `setterminal`

mpcprops

Provide help on MPC controller's properties

Syntax

```
mpcprops
```

Description

`mpcprops` displays details on the generic properties of MPC controllers. It provides a complete list of all the fields of MPC objects with a brief description of each field and the corresponding default values.

See Also

`set` | `get`

mpcsimopt

MPC simulation options

Syntax

```
options = mpcsimopt(MPCobj)
```

Description

`options = mpcsimopt(MPCobj)` creates an set of options for specifying additional parameters for simulating an `mpc` controller, MPCobj, with `sim`. Initially, options is empty. Use dot notation to change the options as needed for the simulation.

Output Arguments

`options`

Options for simulating an `mpc` controller using `sim`. options has the following properties.

MPC Simulation Options Properties

Property	Description
`PlantInitialState`	Initial state vector of the plant model generating the data.
`ControllerInitialState`	Initial condition of the MPC controller. This must be a valid `mpcstate` object. **Note** Nonzero values of `ControllerInitialState.LastMove` are only meaningful if there are constraints on the increments of the manipulated variables.
`UnmeasuredDisturbance`	Unmeasured disturbance signal entering the plant.

Property	Description
	An array with as many rows as simulation steps, and as many columns as unmeasured disturbances. Default: 0
`InputNoise`	Noise on manipulated variables. An array with as many rows as simulation steps, and as many columns as manipulated variables. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
`OutputNoise`	Noise on measured outputs. An array with as many rows as simulation steps, and as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
`RefLookAhead`	Preview on reference signal (`'on'` or `'off'`). Default: `'off'`
`MDLookAhead`	Preview on measured disturbance signal (`'on'` or `'off'`).
`Constraints`	Use MPC constraints (`'on'` or `'off'`). Default: `'on'`
`Model`	Model used in simulation for generating the data. This property is useful for simulating the MPC controller under model mismatch. The LTI object specified in `Model` can be either a replacement for `Model.Plant`, or a structure with fields `Plant` and `Nominal`. By default, `Model` is equal to `MPCobj.Model` (no model mismatch). If `Model` is specified, then `PlantInitialState` refers to the initial state of `Model.Plant` and is defaulted to `Model.Nominal.x`. If `Model.Nominal` is empty, `Model.Nominal.U` and `Model.Nominal.Y` are inherited from `MPCobj.Model.Nominal`. `Model.Nominal.X/`

Property	Description
	DX is only inherited if both plants are state-space objects with the same state dimension.
StatusBar	Display the wait bar ('on' or 'off'). Default: 'off'
MVSignal	Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action). An array with as many rows as simulation steps, and as many columns as manipulated variables. Default: 0
OpenLoop	Perform open-loop simulation ('on' or 'off'). Default: 'off'

Examples

Simulate MPC Control Under Plant Model Mismatch

Simulate the MPC control of a multi-input multi-output (MIMO) system under predicted / actual plant model mismatch. The system has two manipulated variables, two unmeasured disturbances, and two measured outputs.

Define the plant and create the MPC controller.

```
% Open-loop system parameters
p1 = tf(1,[1 2 1])*[1 1; 0 1];
plant = ss([p1 p1]);

% Define I/O types
plant=setmpcsignals(plant,'MV',[1 2],'UD',[3 4]);

% Define I/O names (optional)
set(plant,'InputName',{'mv1','mv2','umd3','umd4'});

% Model for unmeasured input disturbances
distModel = eye(2,2)*ss(-.5,1,1,0);

% Create MPC object
mpcobj = mpc(plant,1,40,2);
```

```
mpcobj.Model.Disturbance = distModel;
```

Perform the closed-loop MPC simulation with model mismatch and unforeseen unmeasured disturbance inputs. First, define the plant model that generates the data.

```
p2 = tf(1.5,[0.1 1 2 1])*[1 1; 0 1];
psim = ss([p2 p2 tf(1,[1 1])*[0;1]]);
psim=setmpcsignals(psim,'MV',[1 2],'UD',[3 4 5]);
```

Set up the simulation parameters and the options. Create the options set, and then set the relevant options.

```
dist = ones(1,3); % Unmeasured disturbance trajectory
refs = [1 2];     % Output reference trajectory
Tf = 100; % Total number of simulation steps

options = mpcsimopt(mpcobj);
options.UnmeasuredDisturbance = dist;
options.Model = psim;
```

Simulate the system.

```
sim(mpcobj,Tf,refs,options);
```

See Also

`sim`

mpcstate

Define MPC controller state

Syntax

```
xmpc = mpcstate(MPCobj)
xmpc = mpcstate(MPCobj,xp,xd,xn,u,p)
xmpc = mpcstate
```

Description

`xmpc = mpcstate(MPCobj)` creates a controller state object compatible with the controller object, MPCobj, in which all fields are set to their default values that are associated with the controller's nominal operating point.

`xmpc = mpcstate(MPCobj,xp,xd,xn,u,p)` sets the state fields of the controller state object to specified values. The controller may be an implicit or explicit controller object. Use this controller state object to initialize an MPC controller at a specific state other than the default state.

`xmpc = mpcstate` returns an `mpcstate` object in which all fields are empty.

`mpcstate` objects are updated by `mpcmove` through the internal state observer based on the extended prediction model. The overall state is updated from the measured output $y_m(k)$ by a linear state observer (see "State Observer").

Input Arguments

`MPCobj`

MPC controller, specified as either a traditional MPC controller (`mpc`) or explicit MPC controller (`generateExplicitMPC`).

`xp`

Plant model state estimates, specified as a vector with N_{xp} elements, where N_{xp} is the number of states in the plant model.

xd

Disturbance model state estimates, specified as a vector with N_{xd} elements, where N_{xd} is the total number of states in the input and output disturbance models. The disturbance model states are ordered such that input disturbance model states are followed by output disturbance model state estimates.

xn

Measurement noise model state estimates, specified as a vector with N_{xn} elements, where N_{xn} is the number of states in the measurement noise model.

u

Values of the manipulated variables during the previous control interval, specified as a vector with N_u elements, where N_u is the number of manipulated variables.

p

Covariance matrix for the state estimates, specified as an N-by-N matrix, where N is the sum of N_{xp}, N_{xd} and N_{xn}).

Output Arguments

xmpc

MPC state object, containing the following properties.

Property	Description
`Plant`	Vector of state estimates for the controller's plant model. Values are in engineering units and are absolute, i.e., they include state offsets. If the controller's plant model includes delays, the `Plant` field of the MPC state object includes states that model the delays. Therefore `length(Plant)` > order of undelayed controller plant model. Default: controller's `Model.Nominal.X` property.

Property	Description
`Disturbance`	Vector of unmeasured disturbance model state estimates. This comprises the states of the input disturbance model followed by the states of the output disturbances model. Disturbance models may be created by default. Use the `getindist`and `getoutdist`commands to view the two disturbance model structures. Default: zero, or empty if there are no disturbance model states.
`Noise`	Vector of output measurement noise model state estimates. Default: zero, or empty if there are no noise model states.
`LastMove`	Vector of manipulated variables used in the previous control interval, $u(k–1)$. Values are absolute, i.e., they include manipulated variable offsets. Default: nominal values of the manipulated variables.
`Covariance`	n-by-n symmetrical covariance matrix for the controller state estimates, where n is the dimension of the extended controller state, i.e., the sum of the number states contained in the `Plant`, `Disturbance`, and `Noise` fields. Default: If the controller is employing default state estimation the default is the steady-state covariance computed according to the assumptions in “Controller State Estimation”. See also the description of the `P` matrix in the Control System Toolbox `kalmd` command. If the controller is employing custom state estimation, this field is empty (not used).

Examples

Get Controller State Object

Create a Model Predictive Controller for a single-input-single-output (SISO) plant. For this example, the plant includes an input delay of 0.4 time units, and the control interval to 0.2 time units.

```
H = tf(1,[10 1],'InputDelay',0.4);
MPCobj = mpc(H,0.2);
```

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

Create the corresponding controller state object in which all states are at their default values.

```
xMPC = mpcstate(MPCobj)
```

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
-->Converting delays to states.
-->Integrated white noise added on measured output channel #1.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
MPCSTATE object with fields
          Plant: [0 0 0]
    Disturbance: 0
          Noise: [1x0 double]
       LastMove: 0
     Covariance: [4x4 double]
```

The plant model, `H`, is a first-order, continuous-time transfer function. The `Plant` property of the `mpcstate` object contains two additional states to model the two intervals of delay. Also, by default the controller contains a first-order output disturbance model (an integrator) and an empty measured output noise model.

View the default covariance matrix.

```
xMPC.Covariance
```

```
ans =

    0.0624    0.0000    0.0000   -0.0224
    0.0000    1.0000    0.0000   -0.0000
    0.0000    0.0000    1.0000   -0.0000
   -0.0224   -0.0000   -0.0000    0.2301
```

See Also

getoutdist | setindist | setoutdist | getEstimator | setEstimator | ss | mpcmove

mpctool

Start Model Predictive Controller GUI

Syntax

```
mpctool
mpctool(MPCobj)
mpctool(MPCobj,'objname')
mpctool(MPCobj1, MPCobj2, ...)
mpctool(MPCobj1, 'objname1', MPCobj2, 'objname2', ...)
mpctool('TaskName')
```

Description

`mpctool` starts the GUI. For more information about designing and testing model predictive controllers, see “Working with the Design Tool”.

`mpctool(MPCobj)` starts the GUI and loads `MPCobj`, which is an existing controller object.

`mpctool(MPCobj,'objname')` assigns `objname` (specified as a string) to the controller you are loading into the GUI. If you do not specify a name, the GUI uses the name of the variable that stores the controller object.

`mpctool(MPCobj1, MPCobj2, ...)` loads the specified list of controllers.

`mpctool(MPCobj1, 'objname1', MPCobj2, 'objname2', ...)` loads the specified list of controllers and assigns each controller the specified name.

`mpctool('TaskName')` starts the GUI and creates a new Model Predictive Control design task with the name specified by the string `'TaskName'`.

See Also

`mpc`

mpcverbosity

Change toolbox verbosity level

Syntax

```
mpcverbosity on
mpcverbosity off
old_status = mpcverbosity(new_status)
mpcverbosity
```

Description

`mpcverbosity on` enables messages displaying default operations taken by Model Predictive Control Toolbox software during the creation and manipulation of model predictive control objects.

`mpcverbosity off` turns messages off.

`old_status = mpcverbosity(new_status)` sets the verbosity level to the specified value, `new_status`. The function returns the original value of the verbosity level as `old_status`. Specify `new_status` as a string with the value of either `'on'` or `'off'` .

`mpcverbosity` just shows the verbosity status.

By default, messages are turned on.

See also “Construction and Initialization” on page 3-11 .

See Also

`mpc`

plot

Plot responses generated by MPC simulations

Syntax

```
plot(MPCobj,t,y,r,u,v,d)
```

Description

`plot(MPCobj,t,y,r,u,v,d)` plots the results of a simulation based on the MPC object `MPCobj`. `t` is a vector of length `Nt` of time values, `y` is a matrix of output responses of size [`Nt`,`Ny`] where `Ny` is the number of outputs, `r` is a matrix of setpoints and has the same size as `y`, `u` is a matrix of manipulated variable inputs of size [`Nt`,`Nu`] where `Nu` is the number of manipulated variables, `v` is a matrix of measured disturbance inputs of size [`Nt`,`Nv`] where `Nv` is the number of measured disturbance inputs, and `d` is a matrix of unmeasured disturbance inputs of size [`Nt`,`Nd`] where `Nd` is the number of unmeasured disturbances input.

See Also

`sim` | `mpc`

plotSection

Visualize explicit MPC control law as 2-D sectional plot

Syntax

```
plotsection(EMPCobj,plotParams)
```

Description

`plotsection(EMPCobj,plotParams)` displays a 2-D sectional plot of the piecewise affine regions used by an explicit MPC controller. All but two of the control law's free parameters are fixed, as specified by plotParams. The two remaining variables form the plot axes. By default, the `EMPCobj.Range` property sets the bounds for these axes.

Examples

Specify Fixed Parameters for 2-D Plot of Explicit Control Law

Suppose `EMPCObj` is an explicit MPC controller involving six controller state variables, two plant outputs (reference signals), and two manipulated variables. Create a plotParams structure that fixes all parameters at their nominal values, except the second manipulated variable and third controller state.

Generate the default plotParams structure for the MPC controller.

```
plotParams = generatePlotParameters(EMPCobj);
```

By default, this structure specifies that all parameters of the controller are fixed at their nominal values.

Allow the third controller state to vary for the purposes of creating a plot. To do so, remove the entry corresponding to that state from `plotParams`.

```
plotParams.State.Index(3) = [];
plotParams.State.Value(3) = [];
```

Similarly, allow the second manipulated variable to vary for the plot.

```
plotParams.ManipulatedVariable.Index(2) = [];
plotParams.ManipulatedVariable.Value(2) = [];
```

You can now use the plot parameters to generate the 2-D section plot for the controller.

```
plotSection(EMPCobj,plotParams)
```

Input Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller for which you want to create a 2-D sectional plot, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

`plotParams` — Parameters for sectional plot

structure

Parameters for sectional plot of explicit MPC control law, specified as a structure. Use `generatePlotParameters` to create an initial structure in which all the parameters of the controller are fixed at their nominal values. Then, modify this structure as necessary before invoking `plotSection`. See `generatePlotParameters` for more information.

See Also

`generateExplicitMPC` | `generatePlotParameters`

review

Examine MPC controller for design errors and stability problems at run-time

Syntax

```
review(mpcobj)
```

Description

`review(mpcobj)` checks for potential design issues in the Model Predictive Controller mpcobj and generates a report. `review` performs the following diagnostic tests:

- Is the optimal control problem well defined?
- Is the controller internally stable?
- Is the closed loop system stable when no constraints are active and there is no model mismatch?
- Is the controller able to eliminate steady-state tracking error when no constraints are active?
- Is there a likelihood that constraint definitions will result in an ill-conditioned or infeasible optimization problem?
- If the controller were used in a real-time environment, what memory capacity would be needed?

Use `review` iteratively to check your initial MPC design or whenever you make substantial changes to mpcobj. Make the recommended changes to your controller to eliminate potential problems. `review` does not modify mpcobj.

Input Arguments

`mpcobj`

Non-empty Model Predictive Controller (`mpc`) object

Examples

Examine an MPC Controller for Design Errors and Stability Problems

Create a Model Predictive Controller. For this example, use a controller with hard upper and lower bounds on the manipulated variable and its rate-of-change.

```
Plant = tf(1, [10 1]);
ts = 2;
MPCobj = mpc(Plant,ts);

MV = MPCobj.MV;
MV.Min = -2;
MV.Max =  2;
MV.RateMin = -4;
MV.RateMax =  4;
MPCobj.MV = MV;
```

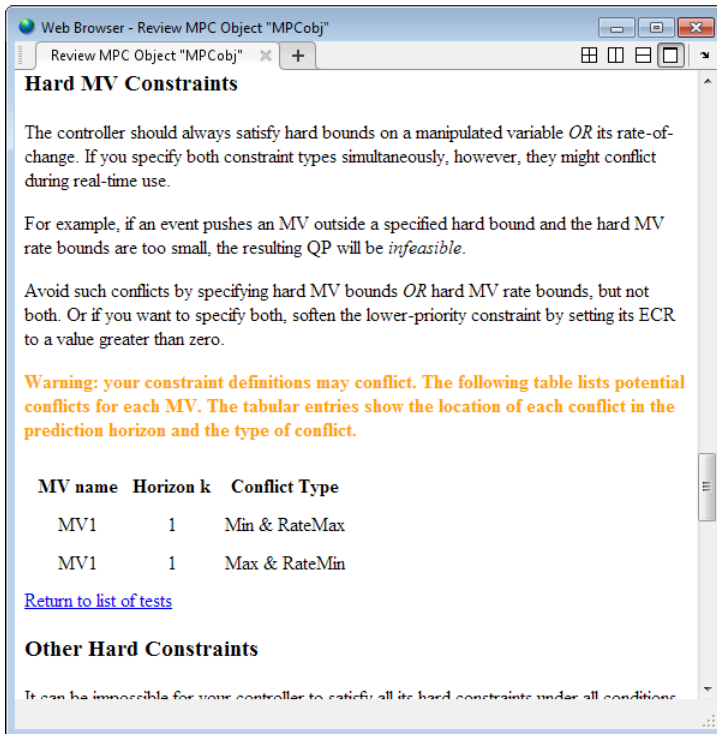
```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property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

Review the design.

```
review(MPCobj)
```

`review` flags the potential constraint conflict that could result if you applied this controller to a real process.

Examine the warning by clicking **Hard MV Constraints**.



Alternatives

`review` automates certain tests that you can perform at the command line.

To test for steady-state tracking errors, use `cloffset`.

To test the internal stability of a controller, check the eigenvalues of the `mpc` object. Use `ss` to convert the `mpc` object to a state-space model and call isstable.

More About

Tips

- If you design your controller using MPC Design Tool, export the controller to the MATLAB Workspace, and analyze it using `review`.

- `review` cannot detect all possible performance factors. So, additionally test your design using techniques such as simulations.

See Also

`cloffset` | `mpc` | `ss`

sensitivity

Compute effect of controller tuning weights on performance

Syntax

```
[J, sens] = sensitivity(MPCobj, PerfFunc, PerfWeights, Tstop, r, v,
simopt, utarget)
[J, sens] = sensitivity(MPCobj,'perf_fun',param1,param2,...)
```

Description

The `sensitivity` function is a controller tuning aid. J specifies a scalar performance metric. `sensitivity` computes J and its partial derivatives with respect to the controller tuning weights. These *sensitivities* suggest tuning weight adjustments that should improve performance, i.e., reduce J.

`[J, sens] = sensitivity(MPCobj, PerfFunc, PerfWeights, Tstop, r, v, simopt, utarget)` calculates the scalar performance metric, `J`, and sensitivities, `sens`, for the controller defined by the MPC controller object `MPCobj`.

`PerfFunc` must be one of the following strings:

`'ISE'` (integral squared error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uij})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right)$$

`'IAE'` (integral absolute error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} \left(\sum_{j=1}^{n_y} | w_j^y e_{yij} | + \sum_{j=1}^{n_u} (| w_j^u e_{uij} | + | w_j^{\Delta u} \Delta u_{ij} |) \right)$$

`'ITSE'` (integral of time-weighted squared error) for which the performance metric is

$$J = \sum_{i=1}^{Tstop} i\Delta t \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uij})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right)$$

$$J = \sum_{i=1}^{Tstop} i\Delta t \left(\sum_{j=1}^{n_y} | w_j^y e_{yij} | + \sum_{j=1}^{n_u} (| w_j^u e_{uij} | + | w_j^{\Delta u} \Delta u_{ij} |) \right)$$

'ITAE' (integral of time-weighted absolute error) for which the performance metric is

In the above expressions n_y is the number of controlled outputs and n_u is the number of manipulated variables. e_{yij} is the difference between output j and its setpoint (or reference) value at time interval i. e_{uij} is the difference between manipulated variable j and its target at time interval i.

The w parameters are nonnegative performance weights defined by the structure PerfWeights, which contains the following fields:

'OutputVariables': 1 by n_y vector containing the w_j^y values

'ManipulatedVariables': 1 by n_u vector containing the w_j^u values

'ManipulatedVariablesRate': 1 by n_u vector containing the $w_j^{\Delta u}$ values

If PerfWeights is unspecified, it defaults to the corresponding weights in MPCobj. In general, however, the performance weights and those used in the controller have different purposes and should be defined accordingly.

Inputs Tstop, r, v, and simopt define the simulation scenario used to evaluate performance. See sim for details.

Tstop is the integer number of controller sampling intervals to be simulated. The final time for the simulations will be $Tstop \times \Delta t$, where Δt is the controller sampling interval specified in MPCobj.

The optional input utarget is a vector of n_u manipulated variable targets. Their defaults are the nominal values of the manipulated variables. Δu_{ij} is the change in manipulated variable j and its target at time interval i.

The structure variable `sens` contains the computed sensitivities (partial derivatives of `J` with respect to the `MPCobj` tuning weights.) Its fields are

`'OutputVariables'`	(1 by n_y) sensitivities with respect to `MPCobj.Weights.OutputVariables`
`'ManipulatedVariables'`	(1 by n_u) sensitivities with respect to `MPCobj.Weights.ManipulatedVariables`
`'ManipulatedVariablesRate'`	(1 by n_u) sensitivities with respect to `MPCobj.Weights.ManipulatedVariablesRate`

See "Weights" on page 3-5 for details on the tuning weights contained in `MPCobj`.

`[J, sens] = sensitivity(MPCobj,'perf_fun',param1,param2,...)` employs a performance function `'perf_fun'` to define `J`. Its function definition must be in the form

```
function J = perf_fun(MPCobj, param1, param2, ...)
```

i.e., it must compute `J` for the given controller and optional parameters `param1`, `param2`, ... and it must be on the MATLAB path.

Note: While performing the sensitivity analysis, the software ignores time-varying, nondiagonal, and ECR slack variable weights.

Examples

Suppose variable `MPCobj` contains a default controller definition for a plant with two controlled outputs, three manipulated variables, and no measured disturbances. Compute its performance and sensitivities as follows:

```
PerfFunc = 'IAE';
PerfWts.OutputVariables = [1 0.5];
PerfWts.ManipulatedVariables = zeros(1,3);
PerfWts.ManipulatedVariablesRate = zeros(1,3);
Tstop = 20;
r = [1 0];
v = [];
simopt = mpcsimopt;
utarget = zeros(1,3);
[J, sens] = sensitivity(MPCobj, PerfFunc, PerfWts, Tstop, ...
```

```
 r, v, simopt, utarget)
```

The simulation scenario in the above example uses a constant $r = 1$ for output 1 and $r = 0$ for output 2. In other words, the scenario is a unit step in the output 1 setpoint.

See Also

`mpc` | `sim`

set

Set or modify MPC object properties

Syntax

```
set(MPCobj,'Property',Value)
set(MPCobj,'Property1',Value1,'Property2',Value2,...)
set(MPCobj,'Property')
set(sys)
```

Description

The `set` function is used to set or modify the properties of an MPC controller (see "MPC Controller Object" on page 3-2 for background on MPC properties). Like its Handle Graphics® counterpart, `set` uses property name/property value pairs to update property values.

`set(MPCobj,'Property',Value)` assigns the value `Value` to the property of the MPC controller `MPCobj` specified by the string `'Property'`. This string can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`).

`set(MPCobj,'Property1',Value1,'Property2',Value2,...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`set(MPCobj,'Property')` displays admissible values for the property specified by `'Property'`. See "MPC Controller Object" on page 3-2 for an overview of legitimate MPC property values.

`set(sys)` displays all assignable properties of `sys` and their admissible values.

See Also

`mpc` | `get` | `mpcprops`

setconstraint

Custom constraints on plant inputs and outputs

Syntax

```
setconstraint(MPCobj,E,F,G)
setconstraint(MPCobj,E,F,G,V)
setconstraint(MPCobj,E,F,G,V,S)
```

Description

`setconstraint(MPCobj,E,F,G)` adds constraints of the following form to an MPC controller MPCobj:

$Eu(k+j\,|\,k) + Fy(k+j\,|\,k) \le G + \varepsilon I$
where:
$j = 0, \ldots, p$
p is the prediction horizon length
y are the measured and unmeasured outputs
u are the manipulated variables
E, F, and G are constants. Each row of E, F, and G represents a linear constraint to be imposed at each prediction horizon step
ε is the slack variable used for constraint softening (as in "Standard Cost Function")

`setconstraint(MPCobj,E,F,G,V)` adds constraints of the following form:

$$Eu(k+j\,|\,k) + Fy(k+j\,|\,k) \le G + \varepsilon V$$

where V is a constant representing the Equal Concern for the Relaxation (ECR).

`setconstraint(MPCobj,E,F,G,V,S)` adds constraints of the following form:

$$Eu(k+j\,|\,k) + Fy(k+j\,|\,k) + Sv(k+j\,|\,k) \le G + \varepsilon V$$

where:
v are the measured disturbances
S is a constant

Input Arguments

`MPCobj`

MPC controller, specified as an MPC Controller object

Default:

`E`

Constant used in custom constraints, specified as a matrix with:

n_u columns, where n_u is the number of manipulated variables
Same number of rows as F, G, V, and S

To remove all the mixed constraints, use `[]` or zero matrix for both the *E* and *F* matrices.

Default:

`F`

Constant used in custom constraints, specified as a matrix with:
n_y columns, where n_y is the number of controlled outputs (measured and unmeasured)
Same number of rows as E, G, V, and S

To remove all the mixed constraints, use `[]` or zero matrix for both the *E* and *F* matrices.

Default:

`G`

Constant used in custom constraints, specified as a column vector with the same number of rows as E, F, V, and S.

Default:

`V`

Constant used in custom constraints, specified as a column vector with the same number of rows as E, F, G, and S.

If V is not specified, the default of 1 is applied to all constraint inequalities and all constraints are soft (default behavior for output bounds as described in “Standard Cost Function”).

To make the ith constraint hard, specify $V(i) = 0$.

To make the ith constraint soft, specify $V(i)(>)0$ in keeping with the constraint violation magnitude you can tolerate. The magnitude violation depends on the numerical scale of the variables involved in the constraint.

In general, as $V(i)$ decreases, the controller decreases the allowed constraint violation, i.e., the constraint becomes harder.

Note: If a constraint is difficult to satisfy, reducing its $V(i)$ value (to make it harder) may be counterproductive, and can lead to erratic control action, instability, or failure of the QP solver that determines the control action.

Default: vector of 1s

S

Constant used in custom constraints, specified as a matrix with:
n_v columns, where n_v is the number of measured disturbances
Same number of rows as E, F, G and V

Default:

Examples

This example shows how to specify the custom constraint $0 \le u_2 - 2u_3 + y_2 \le 15$ on an MPC controller. The controller has three manipulated variables and two controlled outputs.

The constraint imposes upper and lower bounds on $u_2 - 2u_3 + y_2$.

1 Formulate this constraint in the required form:

$$\begin{bmatrix} 0 & -1 & 2 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \le \begin{bmatrix} 0 \\ 15 \end{bmatrix} + \varepsilon I$$

2 Specify the constraints on the plant inputs and outputs:

```
E = [0 -1 2; 0 1 -2];
F = [0 -1; 0 1];
G = [0; 15];
setconstraint(MPCobj,E,F,G);
```

Alternatively, you can use:

```
E = [0 -1 2; 0 1 -2];
F = [0 -1; 0 1];
G = [0; 15];
V = [1; 1];
S = [];
setconstraint(MPCobj,E,F,G,V,S);
```

More About

Tips

- The outputs y are being predicted using a model. If the model is imperfect, there is no guarantee that a constraint can be satisfied.
- Because $u(k + p | k)$ is not optimized by the MPC controller, the last constraint at time $k + p$ assumes that $u(k+p | k) = u(k+p-1 | k)$.

- “Constraints on Linear Combinations of Inputs and Outputs”

See Also

`setterminal`

setEstimator

Modify a model predictive controller's state estimator

Syntax

```
setEstimator(MPCobj,L,M)
setEstimator(MPCobj,'default')
setEstimator(MPCobj,'custom')
```

Description

`setEstimator(MPCobj,L,M)` sets the gain matrices used for estimation of the states of an MPC controller. See "State Estimator Equations" on page 1-34. If L is empty, it defaults to `L = A*M`, where `A` is the state transition matrix defined in "State Estimator Equations" on page 1-34. If M is omitted or empty, it defaults to a zero matrix, and the state estimator becomes a Luenberger observer.

`setEstimator(MPCobj,'default')` restores the gain matrices L and M to their default values. The default values are the optimal static gains calculated by the Control System Toolbox function `kalmd` for the plant, disturbance, and measurement noise models specified in MPCobj.

`setEstimator(MPCobj,'custom')` specifies that controller state estimation will be performed by a user-supplied procedure rather than the equations described in "State Estimator Equations" on page 1-34. This option suppresses calculation of L and M. When the controller is operating in this way, the procedure must supply the state estimate `x[n|n]` to the controller at the beginning of each control interval.

Examples

Design State Estimator by Pole Placement

Design an estimator using pole placement, assuming the linear system *AM=L* is solvable.

Create a plant model.

```
G = tf({1,1,1},{[1 .5 1],[1 1],[.7 .5 1]});
```

To improve the clarity of this example, call `mpcverbosity` to suppress messages related to working with an MPC controller.

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

Create a model predictive controller for the plant. Specify the controller sample time as 0.2 seconds.

```
MPCobj = mpc(G, 0.2);
```

Obtain the default state estimator gain.

```
[~,M,A1,Cm1] = getEstimator(MPCobj);
```

Calculate the default observer poles.

```
e = eig(A1-A1*M*Cm1);
abs(e)
```

```
ans =

    0.9402
    0.9402
    0.8816
    0.8816
    0.7430
    0.9020
```

Specify faster observer poles.

```
new_poles = [.8 .75 .7 .85 .6 .81];
```

Compute a state-gain matrix that places the observer poles at `new_poles`.

```
L = place(A1',Cm1',new_poles)';
```

`place` returns the controller-gain matrix, whereas you want to compute the observer-gain matrix. Using the principle of duality, which relates controllability to observability, you specify the transpose of `A1` and `Cm1` as the inputs to `place`. This function call yields the observer gain transpose.

Obtain the estimator gain from the state-gain matrix.

```
M=A1\L;
```

Specify M as the estimator for MPCobj.

```
setEstimator(MPCobj,L,M);
```

The pair, (A_1, C_{m1}), describing the overall state-space realization of the combination of plant and disturbance models must be observable for the state estimation design to succeed. Observability is checked in Model Predictive Control Toolbox software at two levels: (1) observability of the plant model is checked *at construction* of the MPC object, provided that the model of the plant is given in state-space form; (2) observability of the overall extended model is checked *at initialization* of the MPC object, after all models have been converted to discrete-time, delay-free, state-space form and combined together.

Restore mpcverbosity.

```
mpcverbosity(old_status);
```

Input Arguments

MPCobj — MPC controller

MPC controller object

MPC controller, specified as an MPC controller object. Use the mpc command to create the MPC controller.

L — Kalman gain matrix for time update

A*M (default) | matrix

Kalman gain matrix for the time update, specified as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-34.

If L is empty, it defaults to L = A*M, where A is the state transition matrix defined in “State Estimator Equations” on page 1-34.

M — Kalman gain matrix for measurement update

0 (default) | matrix

Kalman gain matrix for the measurement update, specified as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs. See “State Estimator Equations” on page 1-34.

If M is omitted or empty, it defaults to a zero matrix, and the state estimator becomes a Luenberger observer.

More About

State Estimator Equations

The following equations describe the state estimation. For more details, see "Controller State Estimation".

Output estimate: $y_m[n|n{-}1] = C_m\, x[n|n{-}1] + D_{vm}\, v[n]$.

Measurement update: $x[n|n] = x[n|n{-}1] + M\,(y_m[n] - y_m[n|n{-}1])$.

Time update: $x[n{+}1|n] = A\, x[n|n{-}1] + B_u\, u[n] + B_v\, v[n] + L\,(y_m[n] - y_m[n|n{-}1])$.

Estimator state: $x[n{+}1|n] = (A - L\, C_m)\, x[n|n{-}1] + B_u\, u[n] + (B_v - L\, D_{vm})\, v[n] + L\, y_m[v]$. The estimator state is based on the current measurement of $y_m[n]$ and $v[n]$ as well as the optimal control action $u[n]$ computed at the current control interval.

The variables in these equations are summarized in the following table.

Symbol	Description
x	Controller state vector, length n_x. It includes (in this sequence): • Plant model state estimates. Dimension obtained by conversion of `MPCobj.Model.Plant` to discrete LTI state-space form (if necessary), followed by use of `absorbDelay` to convert any delays to additional states. • Input disturbance model state estimates (if any). Use the `getindist` command to review the input disturbance model structure. • Output disturbance model state estimates (if any). Use the `getoutdist` command to review the output disturbance model structure. • Output measurement noise states (if any) as specified by `MPCobj.Model.Noise`. The length n_x is the sum of the number of states in the above four categories.

Symbol	Description
y_m	Vector of measured outputs or an estimate of their true values, length n_{ym}.
u	Vector of manipulated variables, length n_u.
v	Vector of measured input disturbances, length n_v.
$[j \mid k]$	Denotes an estimate of a state or output at time t_j based on data available at time t_k.
$[k]$	Denotes a quantity known at time t_k, i.e., not an estimate.
A	n_x-by-n_x state transition matrix.
B_u	n_x-by-n_u matrix mapping u to x.
B_v	n_x-by-n_x matrix mapping v to x.
C_m	n_{ym}-by-n_x matrix mapping x to y_m.
D_{vm}	n_{ym}-by-n_v matrix mapping v to y_m. Note that $D_{um} = 0$ because there can be no direct feedthrough between any manipulated variable and any measured output.
L	n_x-by-n_{ym} Kalman gain matrix for the time update. (See `kalmd` in the Control System Toolbox documentation.) Note that $L = A*M$ minimizes the expected state estimation error for most combinations of plant and disturbance models used in MPC, but this is not true in general.
M	n_x-by-n_{ym} Kalman gain matrix for the measurement update. (See `kalmd` in the Control System Toolbox documentation.)

See Also

`getEstimator` | `kalman` | `mpc` | `mpcstate`

setindist

Modify unmeasured input disturbance model

Syntax

```
setindist(MPCobj,'integrators')
setindist(MPCobj,'model',model)
```

Description

`setindist(MPCobj,'integrators')` sets the unmeasured input disturbance model used by model predictive controller object, `MPCobj`, to its default value. This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. The input disturbance model:

- Is a discrete-time, delay-free, state-space (`ss`) object.
- Has unit-variance white noise input signals. By default, the number of inputs depends upon the number of unmeasured input disturbances and the need to maintain controller state observability. For custom input disturbance models, the number of inputs is your choice.
- Has `nd` outputs, where `nd` is the number of unmeasured disturbance inputs defined in `MPCobj.Model.Plant`. Each output channel is sent to the corresponding plant unmeasured disturbance input.

The default input disturbance model has integrators with dimensionless unity gain added to its outputs. An integrator is added for each unmeasured input disturbance, unless doing so would cause a violation of state observability. In this case, a dimensionless unity gain is used instead.

This syntax is equivalent to `MPCobj.Model.Disturbance = []`.

For details on the role of disturbance modeling in model predictive control and about the model used in the algorithm for state estimation, see “Controller State Estimation”.

`setindist(MPCobj,'model',model)` specifies a custom input disturbance model, `model`. This model can be a state-space (`ss`), transfer function (`tf`), or zero-pole-gain

(`zpk`) object with `nd` outputs. If `model` is not specified, the default input disturbance model is used.

When specifying a custom input disturbance model, the number of inputs is your choice. `setindist` does not check custom input disturbance models for violations of state observability. This check is performed later in the MPC design process, when the internal states of `MPCobj.Model` are updated using commands such as `sim` or `mpcmove`.

This syntax is equivalent to `MPCobj.Model.Disturbance = model`.

More About

- "MPC Modeling"
- "Controller State Estimation"

See Also

`getEstimator` | `getindist` | `mpc` | `setEstimator` | `setoutdist`

Introduced before R2006a

setmpcsignals

Set signal types in MPC plant model

Syntax

```
P=setmpcsignals(P,SignalType1,Channels1,SignalType2,Channels2,...)
```

Description

The purpose of `setmpcsignals` is to set I/O channels of the MPC plant model `P`. `P` must be an LTI object. Valid signal types, their abbreviations, and the channel type they refer to are listed below.

Signal Type	Abbreviation	Channel
`Manipulated`	`MV`	Input
`MeasuredDisturbances`	`MD`	Input
`UnmeasuredDisturbances`	`UD`	Input
`MeasuredOutputs`	`MO`	Output
`UnmeasuredOutputs`	`UO`	Output

Unambiguous abbreviations of signal types are also accepted.

`P=setmpcsignals(P)` sets channel assignments to default, namely all inputs are manipulated variables (MVs), all outputs are measured outputs (MOs). More generally, input signals that are not explicitly assigned are assumed to be MVs, while unassigned output signals are considered as MOs.

Examples

We want to define an MPC object based on the LTI discrete-time plant model `sys` with four inputs and three outputs. The first and second input are measured disturbances, the third input is an unmeasured disturbance, the fourth input is a manipulated variable (default), the second output is an unmeasured, all other outputs are measured.

```
sys=setmpcsignals(sys,'MD',[1 2],'UD',[3],'UO',[2]);
mpc1=mpc(sys);
```

Note When using `setmpcsignals` to modify an existing MPC object, be sure that the fields `Weights`, `MV`, `OV`, `DV`, `Model.Noise`, and `Model.Disturbance` are consistent with the new I/O signal types.

See Also

`mpc` | `set`

setname

Set I/O signal names in MPC prediction model

Syntax

```
setname(MPCobj,'input',I,name)
setname(MPCobj,'output',I,name)
```

Description

`setname(MPCobj,'input',I,name)` changes the name of the `I`th input signal to `name`. This is equivalent to `MPCobj.Model.Plant.InputName{I}=name`. Note that `setname` also updates the read-only `Name` fields of `MPCobj.DisturbanceVariables` and `MPCobj.ManipulatedVariables`.

`setname(MPCobj,'output',I,name)` changes the name of the `I`th output signal to `name`. This is equivalent to `MPCobj.Model.Plant.OutputName{I} =name`. Note that `setname` also updates the read-only `Name` field of `MPCobj.OutputVariables`.

Note The `Name` properties of `ManipulatedVariables`, `OutputVariables`, and `DisturbanceVariables` are read-only. You must use `setname` to assign signal names, or equivalently modify the `Model.Plant.InputName` and `Model.Plant.OutputName` properties of the MPC object.

See Also

`getname` | `mpc` | `set`

setoutdist

Modify unmeasured output disturbance model

Syntax

```
setoutdist(MPCobj,'integrators')
setoutdist(MPCobj,'model',model)
```

Description

`setoutdist(MPCobj,'integrators')` sets the output disturbance model used by the model predictive controller object, `MPCobj`, to its default value. This model, in combination with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. The output disturbance model:

- Is a discrete-time, delay-free, state-space (`ss`) object.
- Has unit-variance white noise input signals. By default, the number of inputs depends upon the number of measured outputs and the need to maintain controller state observability. For custom output disturbance models, the number of inputs is your choice.
- Has `ny` outputs, where `ny` is the number of plant outputs defined in `MPCobj.Model.Plant`. Each model output is added to the corresponding plant output.

The default output disturbance model has integrators with dimensionless unity gain added to its outputs according to the following rules:

- No integrators are added for unmeasured plant outputs.
- An integrator is added for each measured output in order of decreasing output weight.
  - For time-varying weights, the sum of the absolute values over time is considered for each output channel.
  - For equal output weights, the order within the output vector is followed.
- For each measured output, an integrator is not added if one of the following is true:

- Adding the integrator would cause a violation of state observability.
- The corresponding value in `MPCobj.Weights.OutputVariables` is zero.

In this case, when an integrator is not added, zero gain is used instead.

For details on the role of disturbance modeling in model predictive control and about the model used in the algorithm for state estimation, see "Controller State Estimation".

`setoutdist(MPCobj,'model',model)` specifies a custom output disturbance model, `model`. This model can be a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) object with `ny` outputs. If `model` is not specified, the default output disturbance model is used.

When specifying a custom output disturbance model, the number of inputs is your choice. `setoutdist` does not check custom output disturbance models for violations of state observability. This check is performed later in the MPC design process, when the internal states of `MPCobj.Model` are updated using commands such as `sim` or `mpcmove`.

More About

- "MPC Modeling"
- "Controller State Estimation"

See Also

`getEstimator` | `getoutdist` | `mpc` | `setEstimator` | `setindist`

Introduced before R2006a

setterminal

Terminal weights and constraints

Syntax

```
setterminal(MPCobj,Y,U)
setterminal(MPCobj,Y,U,Pt)
```

Description

`setterminal(MPCobj,Y,U)` specifies diagonal quadratic penalty weights and constraints at the last step in the prediction horizon. The weights and constraints are on the terminal output $y(t+p)$ and terminal input $u(t+p-1)$, where p is the prediction horizon of the MPC controller MPCobj.

`setterminal(MPCobj,Y,U,Pt)` specifies diagonal quadratic penalty weights and constraints from step *Pt* to the horizon end. By default, *Pt* is the last step in the horizon.

Input Arguments

`MPCobj`

MPC controller, specified as an MPC controller object

Default:

`Y`

Terminal weights and constraints for the output variables, specified as a structure with the following fields:

`Weight`	1-by-n_y vector of nonnegative weights
`Min`	1-by-n_y vector of lower bounds
`Max`	1-by-n_y vector of upper bounds

`MinECR`	1-by-n_y vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
`MaxECR`	1-by-n_y vector of constraint-softening ECR values for the upper bounds

n_y is the number of controlled outputs of the MPC controller.

If the `Weight`, `Min` or `Max` field is empty, the values in MPCobj are used at all prediction horizon steps including the last. For the standard bounds, if any element of the `Min` or `Max` field is infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in "Standard Cost Function"). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation.

By default, `Y.MinECR` = `Y.MaxECR` = 1 (soft output constraints).

Choose the `ECR` magnitudes carefully, accounting for the importance of each constraint and the numerical magnitude of a typical violation.

Default:

`U`

Terminal weights and constraints for the manipulated variables, specified as a structure with the following fields:

`Weight`	1-by-n_u vector of nonnegative weights
`Min`	1-by-n_u vector of lower bounds
`Max`	1-by-n_u vector of upper bounds
`MinECR`	1-by-n_u vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
`MaxECR`	1-by-n_u vector of constraint-softening ECR values for the upper bounds

n_u is the number of manipulated variables of the MPC controller.

If the `Weight`, `Min` or `Max` field is empty, the values in MPCobj are used at all prediction horizon steps including the last. For the standard bounds, if individual elements of

the `Min` or `Max` fields are infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in “Standard Cost Function”). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation.

By default, `U.MinECR` = `U.MaxECR` = 0 (hard manipulated variable constraints)

Choose the ECR magnitudes carefully, accounting for the importance of each constraint and the numerical magnitude of a typical violation.

Default:

`Pt`

Step in the prediction horizon, specified as an integer between 1 and *p*, where *p* is the prediction horizon. The terminal values are applied to Y and U from prediction step `Pt` to the end.

Default: Prediction horizon *p*

Examples

This example shows how to specify constraints and a penalty weight at the last step of the prediction horizon of an MPC controller. The controller has three output variables and two manipulated variables.

1. Specify a prediction horizon of 8.

   ```
   MPCobj.PredictionHorizon = 8;
   ```
2. Define a penalty weight and constraints:

   ```
   Y=struct('Weight',[1,10,0],'Min',[0,-Inf,-1],...
       'Max',[Inf,2,Inf]);
   U=struct('Min',[1,-Inf]);
   ```

   The constraints and weights include:

   - Diagonal penalty weights of 1 and 10 on the first two output variables

- Lower bounds of 0 and –1 on outputs 1 and 3, none on output 2
- Upper bound at 2 on output 2, none on outputs 1 and 3
- Lower bound at 1 on manipulated variable 1
- No other conditions (weights or bounds) on the manipulated variables

3 Specify the constraints and weight at the last step (step 8) of the prediction horizon:

```
setterminal(MPCobj,Y,U);
```

This example shows how to specify constraints and a penalty weight beginning with step 5 and ending at the last step of the prediction horizon of an MPC controller, The controller has three output variables and two manipulated variables.

1 Specify a prediction horizon of 8.

```
MPCobj.PredictionHorizon = 8;
```

2 Define a penalty weight and constraints:

```
Y=struct('Weight',[1,10,0],'Min',[0,-Inf,-1],...
    'Max',[Inf,2,Inf]);
U=struct('Min',[1,-Inf]);
```

The constraints and weights include:

- Diagonal penalty weights of 1 and 10 on the first two output variables
- Lower bounds of 0 and –1 on outputs 1 and 3, none on output 2
- Upper bound at 2 on output 2, none on outputs 1 and 3
- Lower bound at 1 on manipulated variable 1
- No other conditions (weights or bounds) on the manipulated variables

3 Specify the constraints and weight beginning with step 5 and ending at the last step of the prediction horizon:

```
setterminal(MPCobj,Y,U,5);
```

More About

Tips

- Advanced users can impose terminal polyhedral state constraints:

$K_1 \leq Hx \leq K_2$.

First, augment the plant model with additional artificial (unmeasured) outputs, $y = Hx$. Then specify bounds K_1 and K_2 on these y outputs.

- “Terminal Weights and Constraints”

See Also

| `mpc` | `mpcprops` | `setconstraint`

sim

Simulate closed-loop/open-loop response to arbitrary reference and disturbance signals for implicit or explicit MPC

Syntax

```
sim(MPCobj,T,r)
sim(MPCobj,T,r,v)
sim(___,SimOptions)
[y,t,u,xp,xmpc,SimOptions] = sim(___)
```

Description

Use `sim` to simulate the implicit (traditional) or explicit MPC controller in closed loop with a linear time-invariant model, which, by default, is the plant model contained in `MPCobj.Model.Plant`. As an alternative, `sim` can simulate the open-loop behavior of the model of the plant, or the closed-loop behavior in the presence of a model mismatch, when the controller's prediction model differs from the actual plant model.

`sim(MPCobj,T,r)` simulates the closed-loop system formed by the plant model specified in `MPCobj.Model.Plant` and by the MPC controller specified by the MPC controller `MPCobj`, in response to the specified reference signal, `r`. The MPC controller can be either a traditional MPC controller (`mpc`) or explicit MPC controller (`explicitMPC`). The simulation runs for the specified number of simulation steps, `T`. `sim` plots the simulation results.

`sim(MPCobj,T,r,v)` also specifies the measured disturbance signal `v`.

`sim(___,SimOptions)` specifies additional simulation options, which you create with `mpcsimopt`. This syntax allows you to alter the default simulation options, such as initial states, input/output noise and unmeasured disturbances, plant mismatch, etc. You can use SimOptions with any of the previous input combinations.

`[y,t,u,xp,xmpc,SimOptions] = sim(___)` suppresses plotting and instead returns the sequence of plant outputs `y`, the time sequence `t` (equally spaced by `MPCobj.Ts`), the manipulated variables `u` generated by the MPC controller, the sequence `xp` of states

of the model of the plant used for simulation, the sequence `xmpc` of states of the MPC controller (provided by the state observer), and the simulation options, `SimOptions`. You can use this syntax with any of the allowed input argument combinations.

Input Arguments

MPCobj

MPC controller containing the parameters of the Model Predictive Control law to simulate, specified as either an implicit MPC controller (`mpc`) or an explicit MPC controller (`generateExplicitMPC`).

T

Number of simulation steps, specified as a positive integer.

If you omit T, the default value is the row size of whichever of the following arrays has the largest row size:

- The input argument r
- The input argument v
- The `UnmeasuredDisturbance` property of `SimOptions`, if specified
- The `OutputNoise` property of `SimOptions`, if specified

Default: The largest row size of r, v, `UnmeasuredDisturbance`, and `OutputNoise`

r

Reference signal, specified as an array. This array has `ny` columns, where `ny` is the number of plant outputs. r can have anywhere from 1 to T rows. If the number of rows is less than T, the missing rows are set equal to the last row.

Default: `MPCobj.Model.Nominal.Y`

v

Measured disturbance signal, specified as an array. This array has `nv` columns, where `nv` is the number of measured input disturbances. v can have anywhere from 1 to T rows. If the number of rows is less than T, the missing rows are set equal to the last row.

Default: Corresponding entries from `MPCobj.Model.Nominal.U`

`SimOptions`

Simulation options, specified as an options object you create using `mpcsimopt`.

Default: `[]`

Output Arguments

`y`

Sequence of controlled plant outputs, returned as a `T`-by-`Ny` array, where `T` is the number of simulation steps and `Ny` is the number of plant outputs. The values in `y` do not include additive measurement noise, if any).

`t`

Time sequence, returned as a `T`-by-1 array, where `T` is the number of simulation steps. The values in t are equally spaced by `MPCobj.Ts`.

`u`

Sequence of manipulated variables generated by the MPC controller, returned as a `T`-by-`Nu` array, where `T` is the number of simulation steps and `Nu` is the number of manipulated variables.

`xp`

Sequence of plant model states, `T`-by-`Nxp` array, where `T` is the number of simulation steps and `Nxp` is the number of states in the plant model. The plant model is either `MPCobj.Model` or `SimOptions.Model`, if the latter is specified.

`xmpc`

Sequence of MPC controller state estimates, returned as a `T`-by-1 structure array. Each entry in the structure array has the same fields as an `mpcstate` object. The state estimates include plant, disturbance, and noise model states at each time step.

`SimOptions`

Simulation options used, returned as a `mpcsimopt` object.

Examples

Simulate MPC Control of MISO Plant

Simulate the MPC control of a MISO system. The system has one manipulated variable, one measured disturbance, one unmeasured disturbance, and one output.

Create the continuous-time plant model. This plant will be used as the prediction model for the MPC controller.

```
sys = ss(tf({1,1,1},{[1 .5 1],[1 1],[.7 .5 1]}));
```

Discretize the plant model using a sampling time of 0.2 units.

```
Ts = 0.2;
sysd = c2d(sys,Ts);
```

Specify the MPC signal type for the plant input signals.

```
sysd = setmpcsignals(sysd,'MV',1,'MD',2,'UD',3);
```

Create an MPC controller for the `sysd` plant model. Use default values for the weights and horizons.

```
MPCobj = mpc(sysd);
```

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

Constrain the manipulated variable to the `[0 1]` range.

```
MPCobj.MV = struct('Min',0,'Max',1);
```

Specify the simulation stop time.

```
Tstop = 30;
```

Define the reference signal and the measured disturbance signal.

```
num_sim_steps = round(Tstop/Ts);
```

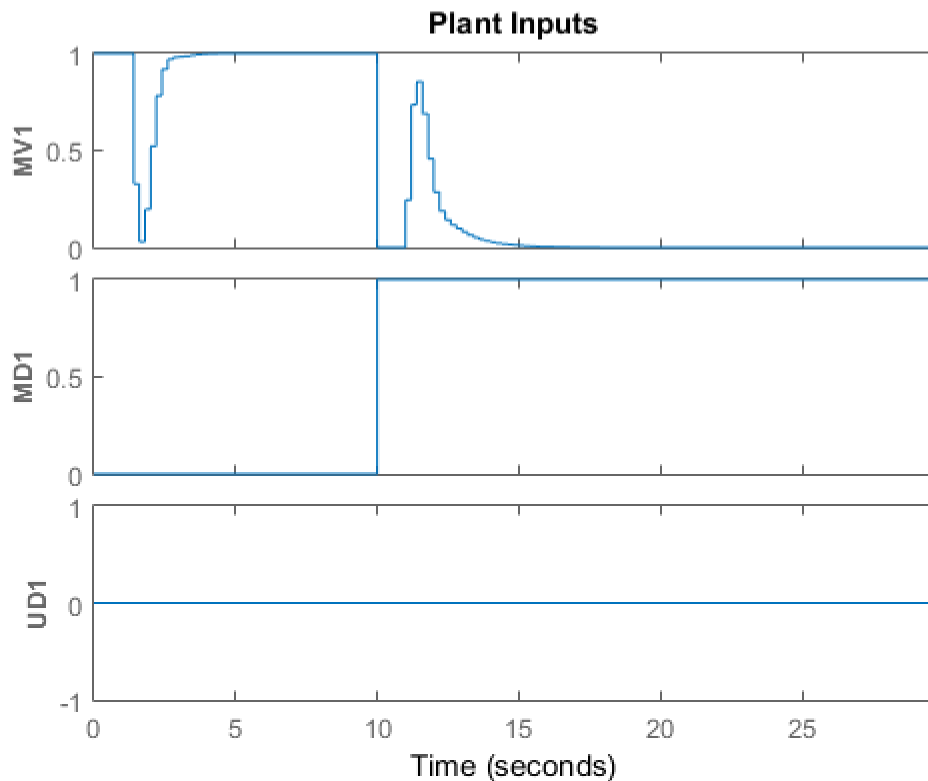
```
r = ones(num_sim_steps,1);
v = [zeros(num_sim_steps/3,1); ones(2*num_sim_steps/3,1)];
```

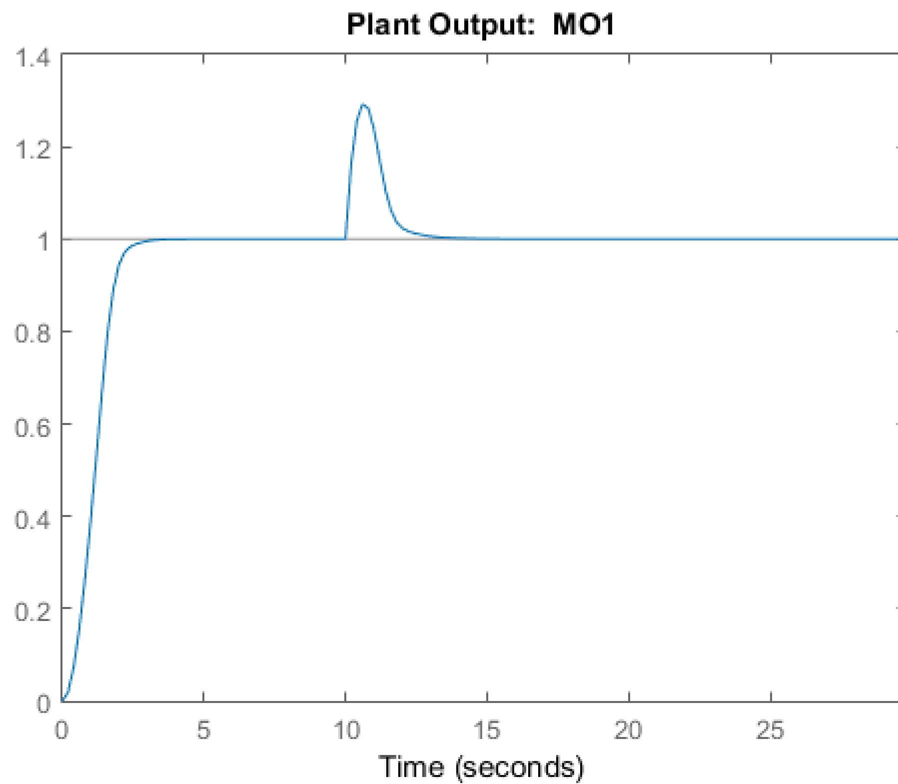
The reference signal, `r`, is a unit step. The measured disturbance signal, `v`, is a unit step, with a 10 unit delay.

Simulate the controller.

```
sim(MPCobj,num_sim_steps,r,v);
```

```
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming unmeasured input disturbance #1 is white noise.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```





See Also

mpcsimopt | mpc | mpcmove

simplify

Reduce explicit MPC controller complexity and memory requirements

Syntax

```
EMPCreduced = simplify(EMPCobj,'exact')
EMPCreduced = simplify(EMPCobj,'exact',uniteeps)
EMPCreduced = simplify(EMPCobj,'radius',r)
EMPCreduced = simplify(EMPCobj,'sequence',index)
simplify(EMPCobj,___)
```

Description

`EMPCreduced = simplify(EMPCobj,'exact')` attempts to reduce the number of piecewise affine (PWA) regions in an explicit MPC controller by merging regions that have identical controller gains and whose union is a convex set. Reducing the number of PWA regions reduces memory requirements of the controller. This command returns a reduced controller, EMPCreduced.

`EMPCreduced = simplify(EMPCobj,'exact',uniteeps)` specifies the tolerance for identifying regions that can be merged.

`EMPCreduced = simplify(EMPCobj,'radius',r)` retains only regions whose Chebyshev radius (the radius of the largest ball contained in the region) is larger than r.

`EMPCreduced = simplify(EMPCobj,'sequence',index)` eliminates all regions except those specified in an index vector.

`simplify(EMPCobj,___)` applies the reduction to the explicit MPC controller EMPCobj, rather than returning a new controller object. You can use this syntax with any of the previous reduction options.

Input Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

Explicit MPC controller to reduce, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

`uniteeps` — Tolerance for joining regions

0.001 (default) | positive scalar

Tolerance for joining PWA regions, specified as a positive scalar.

`r` — Minimum Chebyshev radius

0 (default) | nonnegative scalar

Minimum Chebyshev radius for retaining PWA regions, specified as a nonnegative scalar. When you use the `'radius'` option, `simplify` keeps only the regions whose Chebyshev radius is larger than r. The default value is 0, which causes all regions to be retained.

`index` — Indices of PWA regions to retain

`1:nr` (default) | vector

Indices of PWA regions to retain, specified as a vector. The default value is `[1:nr]`, where `nr` is the number of PWA regions in EMPCobj. Thus, by default, all regions are retained. You can obtain a sequence of regions to retain by performing simulations using EMPCobj and recording the indices of regions actually encountered.

Output Arguments

`EMPCreduced` — Reduced MPC controller

explicit MPC controller object

Reduced MPC controller, returned as an Explicit MPC controller object.

See Also

`generateExplicitMPC`

size

Size and order of MPC Controller

Syntax

```
mpc_obj_size = size(MPCobj)
mpc_obj_size = size(MPCobj,signal_type)
size(MPCobj)
```

Description

`mpc_obj_size = size(MPCobj)` returns a row vector specifying the number of manipulated inputs and measured controlled outputs of an MPC controller. This row vector contains the elements [n_u n_{ym}], where n_u is the number of manipulated inputs and n_{ym} is the number of measured controlled outputs.

`mpc_obj_size = size(MPCobj,signal_type)` returns the number of signals of the specified type that are associated with the MPC controller.

You can specify `signal_type` as one of the following strings:

- `'uo'` — Unmeasured controlled outputs
- `'md'` — Measured disturbances
- `'ud'` — Unmeasured disturbances
- `'mv'` — Manipulated variables
- `'mo'` — Measured controlled outputs

`size(MPCobj)` displays the size information for all the signal types of the MPC controller.

See Also

`mpc` | `set`

ss

Convert unconstrained MPC controller to state-space linear system

Syntax

```
sys = ss(MPCobj)
sys = ss(MPCobj,signals)
sys = ss(MPCobj,signals,ref_preview,md_preview)
[sys,ut] = ss(MPCobj)
```

Description

The `ss` command returns a linear controller in the state-space form. The controller is equivalent to the traditional (implicit) MPC controller `MPCobj` when no constraints are active. You can then use Control System Toolbox software for sensitivity analysis and other diagnostic calculations.

`sys = ss(MPCobj)` returns the linear discrete-time dynamic controller `sys`

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

$u(k) = Cx(k) + Dy_m(k)$

where y_m is the vector of measured outputs of the plant, and u is the vector of manipulated variables. The sampling time of controller `sys` is `MPCobj.Ts`.

Note Vector x includes the states of the observer (plant + disturbance + noise model states) and the previous manipulated variable $u(k\text{-}1)$.

`sys = ss(MPCobj,signals)` returns the linearized MPC controller in its full form and allows you to specify the signals that you want to include as inputs for `sys`.

The full form of the MPC controller has the following structure:

$x(k + 1) = Ax(k) + By_m(k) + B_r r(k) + B_v v(k) + B_{ut} u_{target}(k) + B_{off}$

$$u(k) = Cx(k) + Dy_m(k) + D_r r(k) + D_v v(k) + D_{ut} u_{target}(k) + D_{off}$$

Here, r is the vector of setpoints for both measured and unmeasured plant outputs, v is the vector of measured disturbances, u_{target} is the vector of preferred values for manipulated variables.

Specify `signals` as a single or multicharacter string constructed using any of the following:

- `'r'` — Output references
- `'v'` — Measured disturbances
- `'o'` — Offset terms
- `'t'` — Input targets

For example, to obtain a controller that maps $[y_m; r; v]$ to u, use:

```
sys = ss(MPCobj,'rv');
```

In the general case of nonzero offsets, y_m (as well as r, v, and u_{target}) must be interpreted as the difference between the vector and the corresponding offset. Offsets can be nonzero is `MPCobj.Model.Nominal.Y` or `MPCobj.Model.Nominal.U` are nonzero.

Vectors B_{off}, D_{off} are constant terms. They are nonzero if and only if `MPCobj.Model.Nominal.DX` is nonzero (continuous-time prediction models), or `MPCobj.Model.Nominal.Dx`-`MPCobj.Model.Nominal.X` is nonzero (discrete-time prediction models). In other words, when `Nominal.X` represents an equilibrium state, B_{off}, D_{off} are zero.

Only the following fields of `MPCobj` are used when computing the state-space model: `Model`, `PredictionHorizon`, `ControlHorizon`, `Ts`, `Weights`.

`sys = ss(MPCobj,signals,ref_preview,md_preview)` specifies if the MPC controller has preview actions on the reference and measured disturbance signals. If the flag `ref_preview='on'`, then matrices B_r and D_r multiply the whole reference sequence:

$$x(k + 1) = Ax(k) + By_m(k) + B_r[r(k);r(k + 1);...;r(k + p - 1)] +...$$

$$u(k) = Cx(k) + Dy_m(k) + D_r[r(k);r(k + 1);...;r(k + p - 1)] +...$$

Similarly if the flag `md_preview='on'`, then matrices B_v and D_v multiply the whole measured disturbance sequence:

$$x(k+1) = Ax(k) + \ldots + B_v[v(k);v(k+1);\ldots;v(k+p)] + \ldots$$

$$u(k) = Cx(k) + \ldots + D_v[v(k);v(k+1);\ldots;v(k+p)] + \ldots$$

`[sys,ut] = ss(MPCobj)` additionally returns the input target values for the full form of the controller.

`ut` is returned as a vector of doubles, `[utarget(k); utarget(k+1); ... utarget(k+h)]`.

Here:

- *h* — Maximum length of previewed inputs, that is, `h = max(length(MPCobj.ManipulatedVariables(:).Target))`
- `utarget` — Difference between the input target and corresponding input offsets, that is, `MPCobj.ManipulatedVariables(:).Targets - MPCobj.Model.Nominal.U`

Examples

Convert Unconstrained MPC Controller to State-Space Model

To improve the clarity of the example, suppress messages about working with an MPC controller.

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

Create the plant model.

```
G = rss(5,2,3);
G.D = 0;
G = setmpcsignals(G,'mv',1,'md',2,'ud',3,'mo',1,'uo',2);
```

Configure the MPC controller with nonzero nominal values, weights, and input targets.

```
C = mpc(G,0.1);
C.Model.Nominal.U = [0.7 0.8 0];
C.Model.Nominal.Y = [0.5 0.6];
C.Model.Nominal.DX = rand(5,1);
```

```
C.Weights.MV = 2;
C.Weights.OV = [3 4];
C.MV.Target = [0.1 0.2 0.3];
```

`C` is an unconstrained MPC controller. Specifying `C.Model.Nominal.DX` as nonzero means that the nominal values are not at steady state. `C.MV.Target` specifies three preview steps.

Covert `C` to a state-space model.

```
sys = ss(C);
```

The output, `sys`, is a seventh-order SISO state-space model. The seven states include the five plant model states, one state from the default input disturbance model, and one state from the previous move, `u(k-1)`.

Restore `mpcverbosity`.

```
mpcverbosity(old_status);
```

See Also

`mpc` | `set` | `tf` | `zpk`

tf

Convert unconstrained MPC controller to linear transfer function

Syntax

```
sys=tf(MPCobj)
```

Description

The `tf` function computes the transfer function of the linear controller `ss(MPCobj)` as an LTI system in `tf` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `zpk`

trim

Compute steady-state value of MPC controller state for given inputs and outputs

Syntax

```
x = trim(MPCobj,y,u)
```

Description

The `trim` function finds a steady-state value for the plant state or the best approximation in a least squares sentence such that:

$$x - x_{off} = A(x - x_{off}) + B(u - u_{off})$$
$$y - y_{off} = C(x - x_{off}) + D(u - u_{off})$$

Here, x_{off}, u_{off}, and y_{off} are the nominal values of the extended state x, input u, and output y.

`x` is returned as an `mpcstate` object. Specify `y` and `u` as doubles. `y` specifies the measured and unmeasured output values. `u` specifies the manipulated variable, measured disturbance, and unmeasured disturbance values. The values for unmeasured disturbances must be `0`.

`trim` assumes the disturbance model and measurement noise model to be zero when computing the steady-state value. The software uses the extended state vector to perform the calculation.

See Also

`mpc` | `mpcstate`

zpk

Convert unconstrained MPC controller to zero/pole/gain form

Syntax

```
sys=zpk(MPCobj)
```

Description

The `zpk` function computes the zero-pole-gain form of the linear controller `ss(MPCobj)` as an LTI system in `zpk` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `tf`

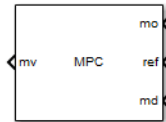
Block Reference

MPC Controller

Compute MPC control law

Library

MPC Simulink Library



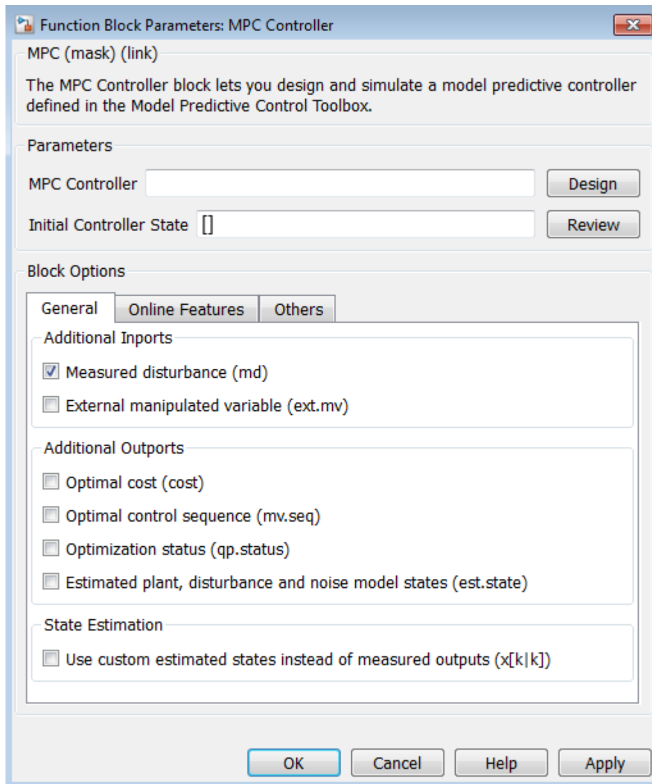
Description

The MPC Controller block receives the current measured output signal (mo), reference signal (ref), and optional measured disturbance signal (md). The block computes the optimal manipulated variables (mv) by solving a quadratic program (QP).

To use the block in simulation and code generation, you must specify an mpc object, which defines a model predictive controller. This controller must have already been designed for the plant that it will control.

Because the MPC Controller block uses MATLAB Function blocks to implement the QP solver, it requires compilation each time you change the MPC object and block. Also, because MATLAB does not allow compiled code to reside in any MATLAB product folder, you must use a non-MATLAB folder to work on your Simulink model when you use MPC blocks.

Dialog Box



The MPC Controller block has the following parameter groupings:

- "Parameters" on page 2-4
- "Required Inports" on page 2-5
- "Required Outports" on page 2-6
- "Additional Inports (General tab)" on page 2-6
- "Additional Outports (General tab)" on page 2-8
- "State Estimation (General tab)" on page 2-10
- "Constraints (Online Features tab)" on page 2-11

- "Weights (Online Features tab)" on page 2-11
- "MV Targets (Online Features tab)" on page 2-14
- "Others tab" on page 2-14

Parameters

MPC controller

You must provide a traditional (implicit) `mpc` object that defines your controller using one of the following methods:

- Enter the name of an `mpc` object in the **MPC Controller** edit box. This object must be present in the base workspace.

  Clicking **Design** opens the MPC design tool where you can modify the controller settings in a graphical environment. For example, you can:

  - Import a new prediction model.
  - Change horizons, constraints, and weights.
  - Evaluate MPC performance with a linear plant.
  - Export the updated controller to the base workspace.

  To see how well the controller works for the nonlinear plant, run a closed-loop Simulink simulation.
- If you do not have an existing `mpc` object in the base workspace, leave the **MPC controller** field empty and, with the MPC Controller block connected to the plant, click **Design**. This action constructs a default `mpc` controller by obtaining a linearized model from the Simulink diagram at the default operating point. Continue your controller design in the MPC design tool.

  To use this design approach, you must have Simulink Control Design™ software.

Initial controller state

Specifies the initial controller state. If this parameter is left blank, the block uses the nominal values that are defined in the `Model.Nominal` property of the `mpc` object. To override the default, create an `mpcstate` object in your workspace that represents the initial state, and enter its name in the field.

Required Inports

Measured output or **State estimate**

If your controller uses default state estimation, this inport is labeled `mo`. Connect the measured plant output variables.

If your controller uses custom state estimation, check **Use custom estimated states instead of measured outputs** in the General tab. Checking that option changes the label on this inport to `x[k|k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables ($ny = n_{ym}$+ number of unmeasured outputs). You have the option to specify future reference values (previewing).

The `ref` signal must be size `N` by n_y, where $N(1 \le N \le p)$ is the number of time steps for which you are specifying reference values and `p` is the prediction horizon. Each element must be a real number. The `ref` dimension must not change from one control instant to the next.

When `N=1`, you cannot preview. To specify future reference values, choose `N` such that $1 < N \le p$ to enable previewing. Doing so usually improves performance via `feedforward` information. The first row specifies the n_y references for the first step in the prediction horizon (at the next control interval `k=1`), and so on for `N` steps. If `N<p`, the last row designates constant reference values to be used for the remaining `p-N` steps.

For example, suppose n_y=2 and `p=6`. At a given control instant, the signal connected to the controller's `ref` inport is

```
[2 5 ←  k=1
 2 6 ←  k=2
 2 7 ←  k=3
 2 8] ←  k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step `(k=1)` are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step `(k=4)` in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

`mpcpreview` shows how to use reference previewing in a specific case. For calculation details on the use of the reference signal, see “Optimization Problem”.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport by solving a quadratic program at each control instant.

Additional Inports (General tab)

Measured disturbance

Add an inport (`md`) to which you can connect a measured disturbance signal.

Your measured disturbance signal `(MD)` must be size `N x` n_{md}, where $n_{md} (\geq 1)$ is the number of measured disturbances defined in your Model Predictive Controller and N $(1 \leq N \leq p+1)$ is the number of time steps for which the MD is known. Each element must be a real, double-precision number. The signal dimensions must not change from one control instant to the next.

If `N = 1`, you cannot preview. At each control instant, the `MD` signal must contain the most recent measurements at the current time `k = 0` (as a row vector, length n_{md}). The controller assumes that the `MDs` remain constant at their current values for the entire prediction horizon.

If you are able to predict future `MD` values, choose `N` such that $1 < N \leq p+1$ to enable previewing. Doing so usually improves performance via `feedforward`. In this case, the

first row must contain the n_{md} current values at k=0, and the remaining rows designate variations over the next N-1 control instants. If N<p+1, the last row designates constant MD values to be used for the remaining p+1-N steps of the prediction horizon.

For example, suppose n_{md} = 2 and p = 6. At a given control instant, the signal connected to the controller's md inport is:

```
[2 5 ←   k=0
 2 6 ←   k=1
 2 7 ←   k=2
 2 8] ←  k=3
```

This signal informs the controller that:

- The current MDs are 2 and 5 at k=0.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the step 3 (k=3) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

mpcpreview shows how to use MD previewing in a specific case.

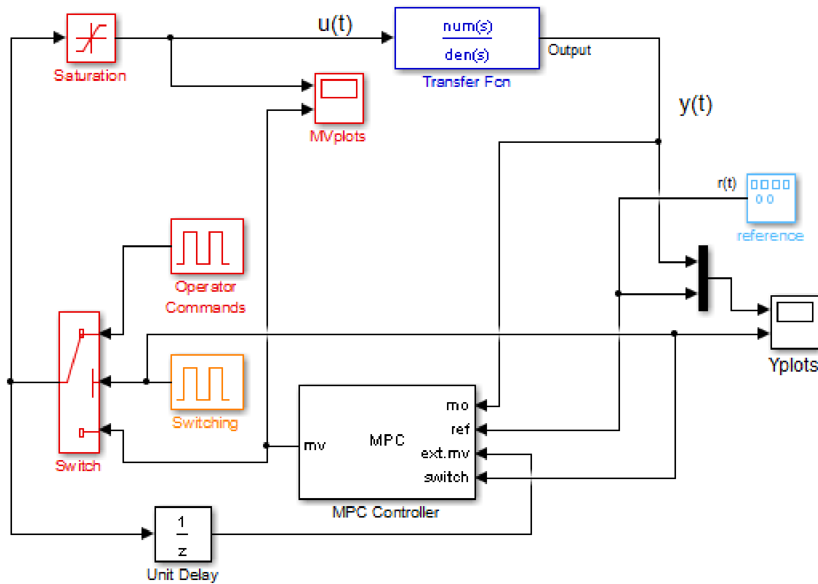
For calculation details, see "MPC Modeling" and "QP Matrices".

External manipulated variable

Add an inport (ext.mv), which you can connect to the actual manipulated variables (MV) used in the plant. The block uses these to update its internal state estimates.

Controller state estimation assumes that the MV is piecewise constant. At time t_k, the ext.mv value must be the effective MV between times t_{k-1} and t_k. For example, if the MV is actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

The following example, from the model mpc_bumpless, includes a switch that can override the controller's output with a signal supplied by the operator. Also, the controller output may saturate. Feeding back the actual MV used in the plant (labeled u(t) in the example) improves the accuracy of controller state estimates.



If the external MV option is inactive or the `ext.mv` inport in unconnected, the controller assumes that its MV output is used in the plant without modification.

Note There is direct feed through from the `ext.mv` inport to the `mv` outport. Thus, use of this option may cause an algebraic loop in the Simulink diagram. In the above examples, the insertion of a unit delay block avoids an algebraic loop.

Additional Outports (General tab)

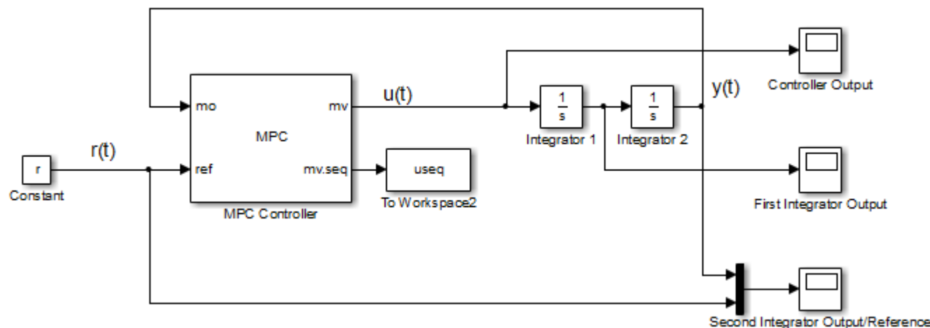
Optimal cost

Add an outport (`cost`) that provides the optimal quadratic programming objective function value at the current time (a nonnegative scalar). If the controller is performing well and no constraints have been violated, the value should be small. If the optimization problem is infeasible, however, the value is meaningless. (See `qp.status`.)

Optimal control sequence

Add an outport (`mv.seq`) that provides the controller's computed optimal `MV` sequence for the entire prediction horizon from `k=0` to `k = p-1`. If n_u is the number of MVs and p is the length of the prediction horizon, this signal is a p by n_u matrix. The first row represents `k=0` and duplicates the block's MV outport.

The following block diagram (from Analysis of Control Sequences Optimized by MPC on a Double Integrator System) illustrates how to use this option. The diagram shows how to collect diagnostic data and send it to the To Workspace2 block, which creates the variable, `useq`, in the workspace. Run the example to see how the optimal sequence evolves with time.



Optimization status

Add an outport (`qp.status`) that allows you to monitor the status of the QP solver.

If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution at this time interval.

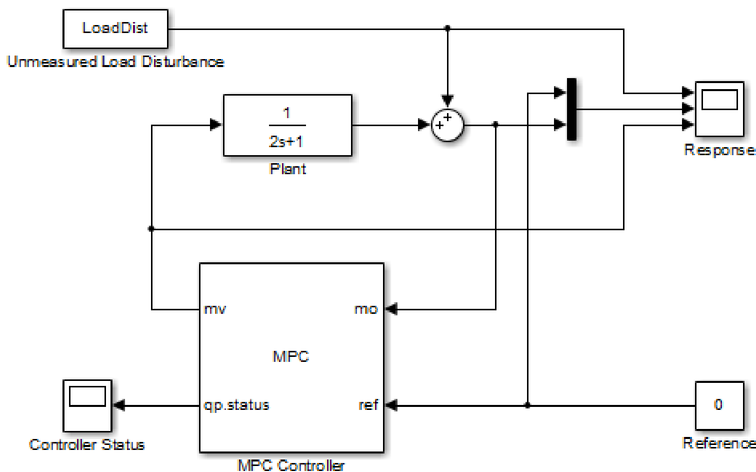
The QP solver may fail to find an optimal solution for the following reasons:

- `qp.status = 0` — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object.
- `qp.status = -1` — The QP solver detects an infeasible QP problem. See Monitoring Optimization Status to Detect Controller Failures for an example where a large, sustained disturbance drives the OV outside its specified bounds.

- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem.

For all the previous three failure modes, the MPC Controller block holds its `mv` output at the most recent successful solution. In a real-time application, you can use status indicator to set an alarm or take other special action.

The next diagram shows how to use the status indicator to monitor the MPC Controller block in real time. See Monitoring Optimization Status to Detect Controller Failures for more details.



Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the controller state estimates, `x[k|k]`, at each control instant. These include the plant, disturbance and noise model states.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

Add the `x[k|k]` inport for custom state estimation as described in "Required Inports" on page 2-5.

Constraints (Online Features tab)

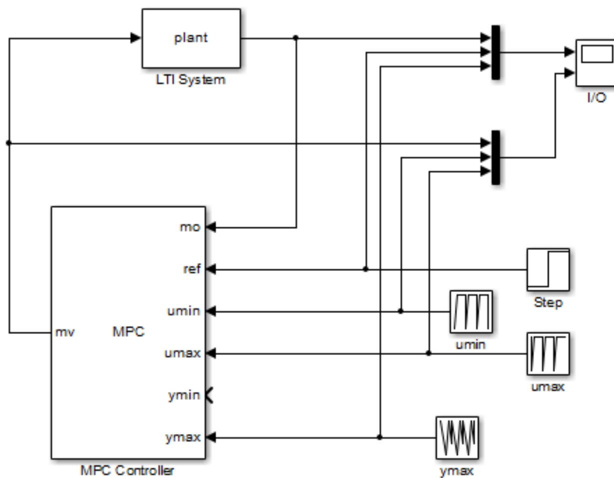
Plant input and output limits

Add inports (umin, umax, ymin, ymax) that you can connect to run-time constraint signals. If this check box is not selected, the block uses the constant constraint values stored within its mpc object. Example connections appear in the model below. (See Varying Input and Output Constraints for an example of using this option.)

An unconnected inport, such as ymin in the example below, is treated as an unbounded signal. The corresponding variable in the mpc object must be unbounded.

For connected inports, such as ymax in the example below, the following rules apply:

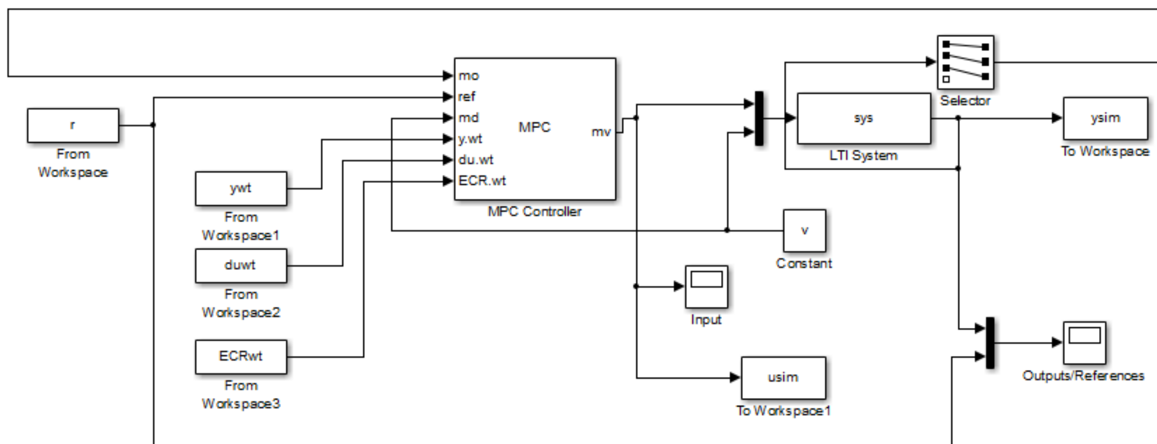
- All connected signals must be finite. Simulink does not support infinite signals.
- If a variable is unconstrained in the controller object, the connected value is ignored.



Weights (Online Features tab)

A controller intended for real-time applications should have “knobs” you can use to tune its performance when it operates with the real plant. This group of optional inports serves that purpose.

The diagram shown below displays three of the MPC Controller block's four tuning knobs. In this simulation context, the knobs are being tuned by pre-stored signals (the `ywt`, `duwt`, and `ECRwt` variables in the `From Workspace` blocks). In practice, you would connect a knob or similar manual adjustment.



Weights on plant outputs

Add an inport (`y.wt`) for a vector signal containing a nonnegative weight for each controlled output variable (OV). This signal overrides the `MPCobj.Weights.OV` property of the `mpc` object, which establishes the relative importance of OV reference tracking.

For example, if the preceding controller defined three OVs, the signal connected to the `y.wt` inport should be a vector with three elements. If the second element is relatively large, the controller would place a relatively high priority on making OV(2) track the `r(2)` reference signal. Setting a `y.wt` signal to zero turns off reference tracking for that OV.

If you do not connect a signal to the `y.wt` inport, the block uses the OV weights specified in your MPC object, and these values remain constant.

Weights on manipulated variables

Add an inport (`u.wt`), whose input is a vector signal defining *nu* nonnegative weights, where *nu* is the number of manipulated variables (MVs). The input overrides the

`MPCobj.Weights.MV` property of the `mpc` object, which establishes the relative importance of MV target tracking.

For example, if your controller defines four MVs and the second `u.wt` element is relatively large, the controller would try to keep MV(2) close to its specified target (relative to other control objectives).

If you do not connect a signal to the `u.wt` inport, the block uses the `Weights.MV` weights property specified in your `mpc` object, and these values remain constant.

Weights on manipulated variable changes

Add an inport (`du.wt`), for a vector signal defining *nu* nonnegative weights, where *nu* is the number of manipulated variables (MVs). The input overrides the `MPCobj.Weights.MVrate` property of the `mpc` object, which establishes the relative importance of MV changes.

For example, if your controller defines four MVs and the second `du.wt` element is relatively large, the controller would use relatively small changes in the second MV. Such *move suppression* makes the controller less aggressive. However, too much suppression makes it sluggish.

If you do not connect a signal to the `du.wt` inport, the block uses the `Weights.MVrate` property specified in your `mpc` object, and these values remain constant.

Weight on overall constraint softening

Add an inport (`ECR.wt`), for a scalar nonnegative signal that overrides the `mpc` controller's `MPCobj.Weights.ECR` property. This inport has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero.

If there are soft constraints, increasing the `ECR.wt` value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

You may not be able to avoid violations of an output variable constraint. Thus, increasing the `ECR.wt` value is often counterproductive. Such an increase causes the controller to pay less attention to its other objectives and does not help reduce constraint violations. You usually need to tune `ECR.wt` to achieve the proper balance in relation to the other control objectives.

MV Targets (Online Features tab)

Targets for manipulated variables

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` inport to which you can connect the target signal (dimension n_u, where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Others tab

Block data type

Specify the block data type as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

  Specify the output data type as `single` if you are implementing the MPC Controller block on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**. For more information, see “Display Port Data Types”.

Inherit sample time

Use the sample time inherited from the parent subsystem as the MPC Controller block’s sample time.

Inheriting the sample time allows you to conditionally execute the MPC Controller block inside the Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note: When you place an MPC controller block inside a Function-Call Subsystem or Triggered Subsystem block, you must execute the subsystem at the controller’s design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see “View Sample Time Information”.

Use external signal to enable or disable optimization

Add an inport (`switch`) whose input specifies whether the controller performs optimization calculations. If the input signal is zero, the controller behaves normally. If the input signal becomes nonzero, the MPC Controller block turns off the controller optimization calculations. This action reduces computational effort when the controller output is not needed, such as when the system is operating manually or another controller has taken over. The controller, however, continues to update its internal state estimate in the usual way. Thus, it is ready to resume optimization calculations whenever the `switch` signal returns to zero. While the controller optimization is turned off, the MPC Controller block passes the current `ext.mv` signal to the controller output. If the `ext.mv` inport is not enabled, the controller output is held at whatever value it had when the optimization was disabled.

See Also

`mpc` | `mpcstate` | Multiple MPC Controllers

Related Examples

- MPC Control with Input Quantization Based on Comparing the Optimal Costs
- Analysis of Control Sequences Optimized by MPC on a Double Integrator System
- “Simulation and Code Generation Using Simulink Coder”
- “Simulation and Structured Text Generation Using PLC Coder”

More About

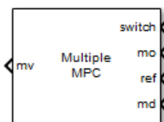
- “MPC Modeling”

Multiple MPC Controllers

Simulate switching between multiple MPC controllers

Library

MPC Simulink Library



Description

As for the MPC Controller block, at each control instant the Multiple MPC Controllers block receives the current measured plant output, reference, and measured plant disturbance (if any). In addition, it receives a switching signal that selects the *active controller* from among a list of two or more candidates. The active controller then solves a quadratic program to determine the optimal plant manipulated variables for the current input signals.

The Multiple MPC Controllers block allows you to achieve better control when operating conditions change. In conventional feedback control, you might compensate for this by gain scheduling. In a similar manner, the Multiple MPC Controllers block allows you to transition between multiple MPC controllers in real-time based on the current conditions. Typically, you design each such controller for a particular region of the operating space. Using the available measurements, you detect the current operating region and choose the appropriate active controller.

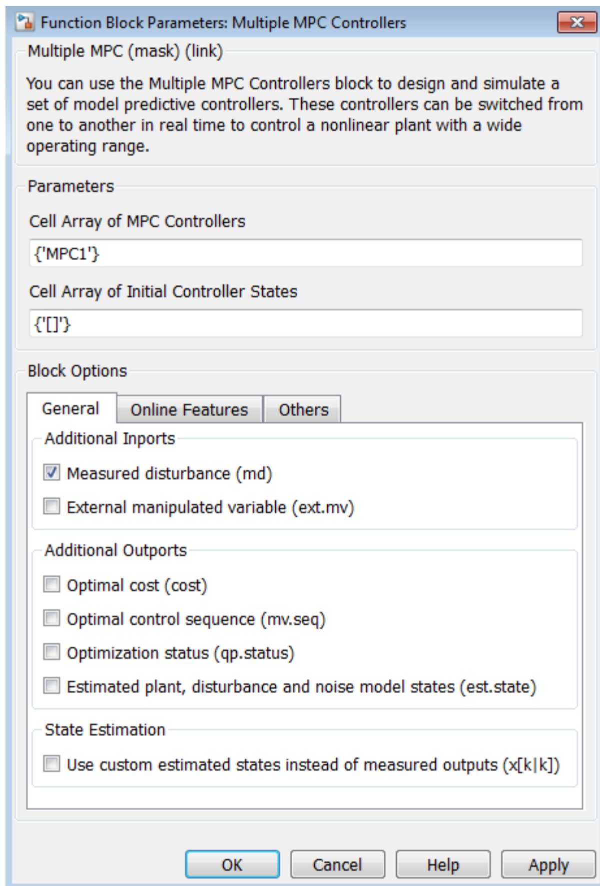
The Adaptive MPC Controller block compensates for operating point variations by modifying its prediction model. The advantages of the Multiple MPC Controllers block over Adaptive MPC Controller block are as follows:

- Simpler configuration – no need to identify prediction model parameters using online data.
- Its candidate controllers form a limited set that you can test thoroughly.

The Multiple MPC Controllers block lacks several optional features found in the MPC Controller block, as follows:

- You cannot disable optimization. One controller must always be active.
- You cannot initiate a controller design from within the block dialog, i.e., there is no **Design** button. You must design all candidate controllers prior to configuring the Multiple MPC Controllers block.
- Similarly, there is no **Review** button. Instead, use the `review` command or the design tool for this purpose.

Dialog Box



The MPC Controller block has the following parameter groupings:

- “Parameters” on page 2-18
- “Required Inports” on page 2-18
- “Required Outports” on page 2-20
- “Additional Inports (General tab)” on page 2-20
- “Additional Outputs (General tab)” on page 2-21
- “State Estimation (General tab)” on page 2-22
- “Constraints (Online Features tab)” on page 2-23
- “Weights (Online Features tab)” on page 2-23
- “MV Targets (Online Features tab)” on page 2-24
- “Others tab” on page 2-24

Parameters

Cell Array of MPC Controllers

Cell array containing an ordered list of at least two `mpc` objects, i.e., the candidate controllers. The first entry in the cell array is the controller to be used when the switch input equals 1, the second entry when the switch input equals 2, and so on. These controller objects must exist in your base workspace.

Cell Array of Initial Controller States

Optional cell array containing the initial state of each `mpc` object in the cell array of MPC controllers. Each entry in the cell array of initial controller states must be an `mpcstate` object. The default is the nominal condition defined in each controller object’s `Model.Nominal` property.

Required Inports

Controller Selection

The `switch` input signal must be a scalar integer between 1 and n_c, where n_c is the number of controllers listed in your block mask. At each control instant, this signal designates the active controller.

Measured output or **State estimate**

If all candidate controllers use default state estimation, this inport is labeled `mo`. Connect the measured plant output variables.

If all candidate controllers use custom state estimation, check **Use custom estimated states instead of measured outputs** in the General tab. Checking

that option changes the label on this inport to `x[k|k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k.

All candidate controllers must use the same state estimation option, default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables (`ny =` n_{ym}`+ number of unmeasured outputs`). You have the option to specify future reference values (previewing).

The `ref` signal must be size `N` by n_y, where $N(1 \le N \le p)$ is the number of time steps for which you are specifying reference values and `p` is the prediction horizon. Each element must be a real number. The `ref` dimension must not change from one control instant to the next.

When `N=1`, you cannot preview. To specify future reference values, choose `N` such that $1 < N \le p$ to enable previewing. Doing so usually improves performance via `feedforward` information. The first row specifies the n_y references for the first step in the prediction horizon (at the next control interval `k=1`), and so on for `N` steps. If `N<p`, the last row designates constant reference values to be used for the remaining `p-N` steps.

For example, suppose n_y`=2` and `p=6`. At a given control instant, the signal connected to the controller's `ref` inport is

```
[2 5 ←  k=1
 2 6 ←  k=2
 2 7 ←  k=3
 2 8] ←  k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step `(k=1)` are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step `(k=4)` in the prediction horizon.

- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

Required Outports

Manipulated Variables

The mv outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its mv outport by solving a quadratic program at each control instant.

Additional Inports (General tab)

Measured disturbance

Add an inport (md) to which you can connect a measured disturbance signal.

Your measured disturbance signal (MD) must be size N x n_{md}, where $n_{md} (\geq 1)$ is the number of measured disturbances defined in the active Model Predictive Controller and N $(1 \leq N \leq p+1)$ is the number of time steps for which the MD is known. Each element must be a real, double-precision number. The signal dimensions must not change from one control instant to the next.

If N = 1, you cannot preview. At each control instant, the MD signal must contain the most recent measurements at the current time k = 0 (as a row vector, length n_{md}). The controller assumes that the MDs remain constant at their current values for the entire prediction horizon.

If you are able to predict future MD values, choose N such that $1 < N \leq p+1$ to enable previewing. Doing so usually improves performance via feedforward. In this case, the first row must contain the n_{md} current values at k=0, and the remaining rows designate variations over the next N-1 control instants. If N<p+1, the last row designates constant MD values to be used for the remaining p+1-N steps of the prediction horizon.

For example, suppose n_{md} = 2 and p = 6. At a given control instant, the signal connected to the controller's md inport is:

```
[2 5 ←    k=0
 2 6 ←    k=1
```

```
 2 7 ←   k=2
 2 8] ←  k=3
```

This signal informs the controller that:

- The current MDs are 2 and 5 at k=0.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the step 3 (k=3) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

mpcpreview shows how to use MD previewing in a specific case.

For calculation details, see "MPC Modeling" and "QP Matrices".

Externally Supplied MV signals

Add an inport (ext.mv) to which you connect the actual manipulated variables (MV) used in the plant. All candidate controllers use this when updating their controller state estimates.

For additional discussion and examples, see the MPC Controller block documentation.

Additional Outputs (General tab)

You may configure a number of optional output signals. At each sampling instant, the active controller determines their values. The following describes each briefly. For more details, see the MPC Controller block documentation.

Optimal cost

Add an outport (cost) that provides the optimal quadratic programming objective function value at the current time (a nonnegative scalar). If the controller is performing well and no constraints have been violated, the value should be small. If the optimization problem is infeasible, however, the value is meaningless. (See qp.status.)

Optimal control sequence

Add an outport (mv.seq) that provides the active controller's computed optimal MV sequence for the entire prediction horizon from k=0 to k = p-1. If n_u is the number of

MVs and p is the length of the prediction horizon, this signal is a p by n_u matrix. The first row represents `k=0` and duplicates the block's MV outport.

Optimization status

Add an outport (`qp.status`) that allows you to monitor the status of the active controller's QP solver.

If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution at this time interval.

The QP solver may fail to find an optimal solution for the following reasons:

- `qp.status = 0` — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object.
- `qp.status = -1` — The QP solver detects an infeasible QP problem. See Monitoring Optimization Status to Detect Controller Failures for an example where a large, sustained disturbance drives the OV outside its specified bounds.
- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem.

For all the previous three failure modes, the Multiple MPC Controllers block holds its `mv` output at the most recent successful solution. In a real-time application, you can use status indicator to set an alarm or take other special action.

Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the active controller's state estimates, `x[k|k]`, at each control instant. These include the plant, disturbance and noise model states.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

Add the `x[k|k]` inport for custom state estimation as described in "Required Inports" on page 2-18. All candidate controllers must use the same state estimation option, default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Constraints (Online Features tab)

At each control instant, the optional features described below apply to the active controller.

Plant input and output limits

Add inports (`umin`, `umax`, `ymin`, `ymax`) that you can connect to run-time constraint signals. If this check box is not selected, the block uses the constant constraint values stored within the active controller.

An unconnected inport is treated as an unbounded signal. The corresponding variable in the `mpc` object must be unbounded.

For connected inports, the following rules apply:

- All connected signals must be finite. Simulink does not support infinite signals.
- If a variable is unconstrained in the controller object, the connected value is ignored.

Weights (Online Features tab)

The optional inputs described below function as controller “tuning knobs.” By default (or when a signal is unconnected), the active controller’s stored tuning weights apply.

When using these online tuning features, you should usually prevent an unexpected change in the active controller. Otherwise, settings intended for a particular candidate controller may instead retune another.

Weights on plant outputs

Add an inport (`y.wt`) for a vector signal containing a nonnegative weight for each controlled output variable (OV). This signal overrides the `MPCobj.Weights.OV` property of the active controller, which establishes the relative importance of OV reference tracking.

If you do not connect a signal to the `y.wt` inport, the block uses the OV weights specified in the active controller, and these values remain constant.

Weights on manipulated variables

Add an inport (`u.wt`), whose input is a vector signal defining *nu* nonnegative weights, where *nu* is the number of manipulated variables (MVs). The input overrides the

`MPCobj.Weights.MV` property of the active controller, which establishes the relative importance of MV target tracking.

If you do not connect a signal to the `u.wt` inport, the block uses the `Weights.MV` weights property specified in the active controller, and these values remain constant.

Weights on manipulated variable changes

Add an inport (`du.wt`), for a vector signal defining *nu* nonnegative weights, where *nu* is the number of manipulated variables (MVs). The input overrides the `MPCobj.Weights.MVrate` property of the active controller, which establishes the relative importance of MV changes.

If you do not connect a signal to the `du.wt` inport, the block uses the `Weights.MVrate` property specified in the active controller, and these values remain constant.

Weight on overall constraint softening

Add an inport (`ECR.wt`), for a scalar nonnegative signal that overrides the active controller's `MPCobj.Weights.ECR` property. This inport has no effect unless the active controller defines soft constraints whose associated ECR values are nonzero.

MV Targets (Online Features tab)

Targets for manipulated variables

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` inport to which you can connect the target signal (dimension n_u, where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Others tab

Block data type

Specify the block data type of the manipulated variables as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

Specify the output data type as `single` if you are implementing the MPC Controller block on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**. For more information, see “Display Port Data Types”.

Inherit sample time

Use the sample time inherited from the parent subsystem as the Multiple MPC Controllers block’s sample time.

Inheriting the sample time allows you to conditionally execute the Multiple MPC Controllers block inside the Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note: When you place an MPC controller inside a Function-Call Subsystem or Triggered Subsystem block, you must execute the subsystem at the controller’s design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see “View Sample Time Information”.

See Also

`mpc` | MPC Controller | `mpcmove` | `mpcstate`

Related Examples

- Scheduling Controllers for a Plant with Multiple Operating Points
- Chemical Reactor with Multiple Operating Points
- “Simulation and Code Generation Using Simulink Coder”
- “Simulation and Structured Text Generation Using PLC Coder”

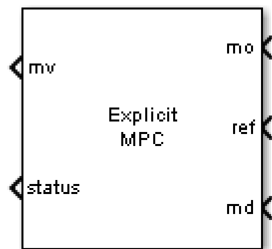
Explicit MPC Controller

Design and simulate explicit model predictive controller

Library

MPC Simulink Library

Description



Like the MPC Controller block, the Explicit MPC Controller block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

The key difference is that the Explicit MPC Controller block uses a table-lookup control law rather than solving a quadratic program during each control interval. The reduced online computational effort is advantageous in applications requiring a short control interval. The primary trade-off is a heavier offline computational effort needed to determine the control law and a larger memory footprint to store it. The combinatorial character of this computation restricts its use to applications with relatively few input, output, and state variables, a short prediction horizon, and few output constraints, if any.

The `Explicit MPC Controller` block also has fewer optional features than the other blocks in the MPC Simulink Library. In particular, it does not support the following:

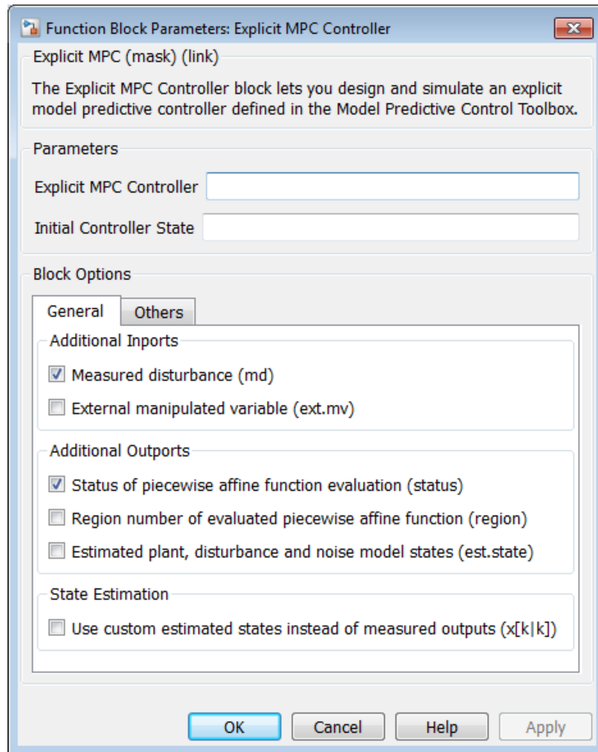
- Online tuning (i.e., penalty weight adjustments)
- Online constraint adjustments
- Manipulated variable target adjustments
- Reference and/or measured disturbance previewing
- Cost function outport
- Optimal control sequence outport

The block does support the following options common to most MPC Library blocks:

- Custom state estimation (default state estimation uses a static Kalman filter)
- Outport for state estimation results
- External manipulated variable feedback signal inport
- Single-precision block data (default is double precision)
- Inherited sample time

It also provides two optional status outports unique to this block.

Dialog Box



The Explicit MPC Controller block has the following parameter groupings:

- “Parameters” on page 2-29
- “Required Inports” on page 2-29
- “Required Outports” on page 2-29
- “Additional Inports (General Tab)” on page 2-30
- “Additional Outports (General tab)” on page 2-30
- “Others tab” on page 2-31

Parameters

Explicit MPC Controller

An Explicit MPC controller object containing the control law to be used. It must exist in the workspace. Use the `generateExplicitMPC` command to create this object.

Initial Controller State

An optional `mpcstate` object specifying the initial controller state. By default the block uses the controller object's `Model.Nominal` property.

Required Inports

Measured output or **State estimate**

If your controller uses default state estimation, this inport is labeled `mo`. Connect the measured plant output variables.

If your controller uses custom state estimation, check **Use custom estimated states instead of measured outputs** in the General tab. Checking that option changes the label on this inport to `x[k|k]`. Connect a signal providing the controller state estimates. (The controller state includes the plant, disturbance, and noise model states.) The estimates supplied at time t_k must be based on the measurements and other data available at time t_k.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables. Reference previewing is not supported. The block assumes each reference value is constant over the prediction horizon.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport at each control instant using the control law contained in the Explicit MPC controller object. If the control law evaluation fails, this signal is unchanged, i.e., held at the previous successful result.

Additional Inports (General Tab)

Measured disturbance

Add an inport (md) to which you can connect a vector signal containing n_{md} elements, where n_{md} is the number of measured disturbances.

Measured disturbance previewing is not supported. The block assumes that each measured disturbance value is constant over the prediction horizon.

External manipulated variable

Add an inport (ext.mv) to which you can connect a vector signal containing the n_u actual manipulated variables (MVs) used in the plant. The block uses this when updating its controller state estimates. Using this inport improves state estimation accuracy when the MVs used in the plant differ from those calculated by the block, e.g., due to signal saturation or an override condition.

Controller state estimation assumes that the MV is piecewise constant. At time t_k, the ext.mv value must be the effective MV between times t_{k-1} and t_k. For example, if the MV is actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

If the external MV option is not selected or its inport in unconnected, the block assumes that its MV output is used in the plant without modification.

For additional discussion and examples, see the corresponding section of the MPC Controller block reference page.

Note There is direct feed through from the ext.mv inport to the mv outport. Thus, use of this option may cause an algebraic loop in the Simulink diagram. You might need to insert a Memory or Unit Delay block to prevent such algebraic loops.

Additional Outports (General tab)

Status of piecewise affine function evaluation

Add an outport (status) that indicates whether the latest explicit MPC control-law evaluation succeeded. The outport provides a scalar signal that has one of the following values:

- 1 — Successful explicit control law evaluation,
- 0 — Failure: one or more of the control law's parameters was out of range.
- –1 — Undefined: control law parameters were within the valid range but an extrapolation was necessary.

Region number of evaluated piecewise affine function

Add an outport (`region`) providing the index of the polyhedral region used in the latest explicit control law evaluation (a scalar). If the control law evaluation fails, the signal at this outport equals zero.

Estimated plant, disturbance, and noise model states

Add an outport (`est.state`) to receive the controller state estimates, `x[k|k]`, at each control instant. These include the plant, disturbance and noise model states.

Others tab

Block data type

Specify the block data type as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

  Specify the output data type as `single` if you are implementing the MPC Controller block on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see "Simulation and Code Generation Using Simulink Coder".

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**. For more information, see "Display Port Data Types".

Inherit sample time

Use the sample time inherited from the parent subsystem as the MPC Controller block's sample time.

Inheriting the sample time allows you to conditionally execute the MPC Controller block inside the Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note: When you place an Explicit MPC Controller block inside a Function-Call Subsystem or Triggered Subsystem block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see "View Sample Time Information".

See Also

`generateExplicitMPC` | `mpc` | MPC Controller | `mpcmoveExplicit` | `mpcstate`

Related Examples

- "Explicit MPC Control of a Single-Input-Single-Output Plant"
- "Explicit MPC Control of an Aircraft with Unstable Poles"
- "Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"

More About

- "Explicit MPC"
- "Design Workflow for Explicit MPC"

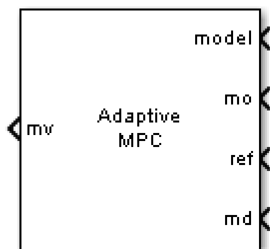
Adaptive MPC Controller

Design and simulate adaptive model predictive controller

Library

MPC Simulink Library

Description



Like the MPC Controller block, the Adaptive MPC Controller block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

In addition, the required model input signal specifies the prediction model to be used when solving the quadratic program (QP) for the block output (`mv`), i.e., the optimal plant manipulated variables. The prediction model can change at each control instant in response to changes in the plant. Two common ways to modify this model are as follows:

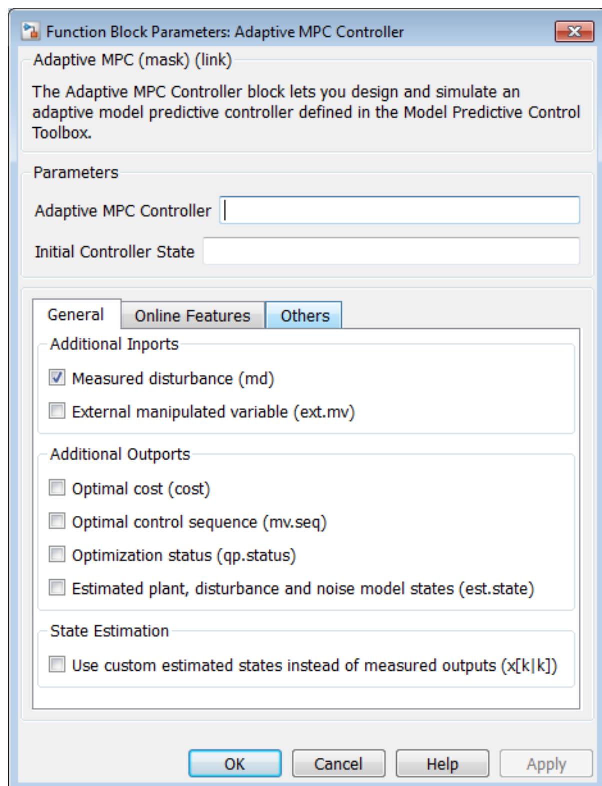
- Given a nonlinear plant model, linearize it at the current operating point.
- Use plant data to estimate parameters in an empirical linear-time-varying model.

By default, the block estimates its prediction model states. Since the prediction model parameters change with time, the static Kalman filter used in the MPC Controller block

is inappropriate. Instead, the Adaptive MPC Controller block employs a linear-time-varying Kalman filter (LTVKF). See "Adaptive MPC" for details.

In all other ways the Adaptive MPC Controller block mimics the simpler MPC Controller block. As the adaptive version involves additional overhead, use the MPC Controller block unless you need to modify the prediction model. Another alternative is the Multiple MPC Controllers block, which allows you to use a finite, predetermined set of prediction models.

Dialog Box



The Adaptive MPC Controller block has the following parameter groupings:

- "Parameters" on page 2-35

- “Required Inports” on page 2-35
- “Required Outports” on page 2-36
- “Additional Inports (General tab)” on page 2-36
- “Additional Inports (General tab)” on page 2-37
- “State Estimation (General tab)” on page 2-37
- “Constraints (Online Features tab)” on page 2-37
- “MV Targets (Online Features tab)” on page 2-37
- “Others tab” on page 2-37

Parameters

Adaptive MPC Controller

A traditional (implicit) `mpc` controller object whose prediction model is to be modified at each control instant. By default, the block assumes all other controller object properties (e.g., tuning weights, constraints) are constant. You can override this assumption using the options in the **Online Features** tab.

The following restrictions apply to the `mpc` controller object:

- It must exist in your base workspace.
- Its prediction model must be an LTI discrete-time, state-space object with no delays. (Use the `absorbDelay` command to convert delays to discrete states.)

Initial Controller State

An optional `mpcstate` object specifying the initial controller state. If you leave this box blank, the block uses the nominal values defined in the controller object’s `Model.Nominal` property. The alternative is to create an `mpcstate` object in your workspace, initialize it to the desired state, and enter its name in the box.

Required Inports

Model

Connect a bus signal to the `model` inport. This signal modifies the controller object’s `Model.Plant` and `Model.Nominal` properties at the beginning of each control interval.

The Adaptive MPC Controller requires `Model.Plant` to be an LTI discrete-time state-space object with no delays. The following command extracts the state-space matrices comprising such a model:

```
[A,B,C,D] = ssdata(MPCobj.Model.Plant)
```

The purpose of the `model` inport is to replace these matrices with new ones having the same dimensions, and representing the same control interval. You must also retain the sequence in which the input, output and state variables appear in `Model.Plant`.

The bus you connect to the `model` inport must contain the following signals, each identified by the specified name:

- `A` — n_x-by-n_x matrix signal, where n_x is the number of plant model states.
- `B` — n_x-by-n_{utot} matrix signal, where n_{utot} is the total number of plant model inputs (i.e., manipulated variables, measured disturbances, and unmeasured disturbances).
- `C` — n_y-by-n_x matrix signal, where n_y is the number of plant model outputs.
- `D` — n_y-by-n_{utot} matrix signal.
- `X` — vector signal, length n_x, replacing the controller's `Model.Nominal.X` property.
- `Y` — vector signal, length n_y, replacing the controller's `Model.Nominal.Y` property.
- `U` — vector signal, length n_{utot}, replacing the controller's `Model.Nominal.U` property.
- `DX` — vector signal, length n_x, replacing the controller's `Model.Nominal.DX` property. It must be appropriate for use with a discrete-time model of the assumed control interval.

One way to form the bus is to use a Simulink Bus Creator block.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport by solving a quadratic program at each control instant.

Additional Inports (General tab)

These optional inports are identical to the corresponding inports in the MPC Controller block. Please see the MPC Controller block reference page for information.

Additional Inports (General tab)

These optional outports are identical to the corresponding outports in the MPC Controller block. Please see the MPC Controller block reference page for information.

State Estimation (General tab)

Use custom estimated states instead of measured outputs

Add the `x[k|k]` inport for custom state estimation.

Constraints (Online Features tab)

These optional inports are identical to the corresponding inports in the MPC Controller block. Please see the MPC Controller block reference page for information.

MV Targets (Online Features tab)

If you want one or more manipulated variable (MV) to track a target value that changes with time, use this option to add an `mv.target` inport to which you can connect the target signal (dimension n_u, where n_u is the number of MVs).

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Others tab

These parameters are identical to the corresponding parameters in the MPC Controller block. Please see the MPC Controller block reference page for information.

See Also

`mpc` | MPC Controller | `mpcmoveAdaptive` | `mpcstate` | Multiple MPC Controllers

Related Examples

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization”

- “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation”

More About

- “Adaptive MPC”

Object Reference

- “MPC Controller Object” on page 3-2
- “MPC Simulation Options Object” on page 3-12
- “MPC State Object” on page 3-15
- “Explicit MPC Controller Object” on page 3-17

MPC Controller Object

All of the parameters defining the traditional (implicit) MPC control law are stored in an MPC object, whose properties are listed in the following table.

MPC Controller Object

Property	Description
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
`OutputVariables` (or `OV` or `Controlled` or `Output`)	Scale factors, input bounds, input-rate bounds, corresponding ECR values, target values, signal names, and units.
`DisturbanceVariables` (or `DV` or `Disturbance`)	Disturbance scale factors, names, and units
`Weights`	Weights used in computing the performance (cost) function
`Model`	Plant, input disturbance, and output noise models, and nominal conditions.
`Ts`	Controller sample time
`Optimizer`	Parameters controlling the QP solver
`PredictionHorizon`	Prediction horizon
`ControlHorizon`	Number of free control moves or vector of blocking moves
`History`	Creation time
`Notes`	Text or comments about the MPC controller object
`UserData`	Any additional data

ManipulatedVariables

`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`) is an n_u-dimensional array of structures (n_u = number of manipulated variables), one per manipulated variable. Each structure has the fields described in the following table, where p denotes the

prediction horizon. Unless indicated otherwise, numerical values are in engineering units.

Manipulated Variable Structure

Field Name	Content	Default
`ScaleFactor`	Nonnegative scale factor for this MV	1
`Min`	1 to p length vector of lower bounds on this MV	`-Inf`
`Max`	1 to p length vector of upper bounds on this MV	`Inf`
`MinECR`	1 to p length vector of nonnegative parameters specifying the `Min` bound softness (0 = hard).	`0` (dimensionless)
`MaxECR`	1 to p length vector of nonnegative parameters specifying the `Max` bound softness (0 = hard).	`0` (dimensionless)
`Target`	1 to p length vector of target values for this MV	`'nominal'`
`RateMin`	1 to p length vector of lower bounds on the interval-to-interval change for this MV	`-Inf`
`RateMax`	1 to p length vector of upper bounds on the interval-to-interval change for this MV	`Inf`
`RateMinECR`	1 to p length vector of nonnegative parameters specifying the `RateMin` bound softness (0 = hard).	`0` (dimensionless)
`RateMaxECR`	1 to p length vector of nonnegative parameters specifying the `RateMax` bound softness (0 = hard).	`0` (dimensionless)
`Name`	Read-only MV signal name (character string)	`InputName` of LTI plant model
`Units`	Read-only MV signal units (character string)	`InputUnit` of LTI plant model

Note Rates refer to the difference $\Delta u(k)=u(k)-u(k-1)$. Constraints and weights based on derivatives du/dt of continuous-time input signals must be properly reformulated for the discrete-time difference $\Delta u(k)$, using the approximation $du/dt \cong \Delta u(k)/T_s$.

OutputVariables

`OutputVariables` (or `OV` or `Controlled` or `Output`) is an n_y-dimensional array of structures (n_y = number of outputs), one per output signal. Each structure has the fields described in the following table. p denotes the prediction horizon. Unless specified otherwise, values are in engineering units.

Output Variable Structure

Field Name	Content	Default
`ScaleFactor`	Nonnegative scale factor for this OV	1
`Min`	1 to p length vector of lower bounds on this OV	`-Inf`
`Max`	1 to p length vector of upper bounds on this OV	`Inf`
`MinECR`	1 to p length vector of nonnegative parameters specifying the `Min` bound softness (0 = hard).	`1` (dimensionless)
`MaxECR`	1 to p length vector of nonnegative parameters specifying the `Max` bound softness (0 = hard).	`1` (dimensionless)
`Name`	Read-only OV signal name (character string)	`OutputName` of LTI plant model
`Units`	Read-only OV signal units (character string)	`OutputUnit` of LTI plant model

In order to reject constant disturbances due, for instance, to gain nonlinearities, the default measured output disturbance model used in Model Predictive Control Toolbox software is integrated white noise (see “Output Disturbance Model”).

DisturbanceVariables

`DisturbanceVariables` (or `DV` or `Disturbance`) is an (n_v+n_d)-dimensional array of structures (n_v = number of measured input disturbances, n_d = number of unmeasured input disturbances). Each structure has the fields described in the following table.

Disturbance Variable Structure

Field Name	Content	Default
`ScaleFactor`	Nonnegative scale factor for this DV	1
`Name`	Read-only DV signal name (character string)	`InputName` of LTI plant model
`Units`	Read-only DV signal units (character string)	`InputUnit` of LTI plant model

The order of the disturbance signals within the array `DV` is the following: the first n_v entries relate to measured input disturbances, the last n_d entries relate to unmeasured input disturbances.

Weights

`Weights` is the structure defining the QP weighting matrices. It contains four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see "Standard Cost Function") or the alternative cost function (see "Alternative Cost Function").

Standard Cost Function

The table below lists the content of the four structure fields. In the table, p denotes the prediction horizon, n_u the number of manipulated variables, and n_y the number of output variables.

For the `MV`, `MVRate` and `OV` weights, if you specify fewer than p rows, the last row repeats automatically to form a matrix containing p rows.

Weights for the Standard Cost Function

Field Name (Abbreviations)	Content	Default (dimensionless)
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	(1 to p)-by-n_u dimensional array of nonnegative MV weights	`zeros(1,nu)`

Field Name (Abbreviations)	Content	Default (dimensionless)
`ManipulatedVariablesRate` (or `MVRate` or `ManipulatedRate` or `InputRate`)	(1 to p)-by-n_u dimensional array of MV-increment weights	`0.1*ones(1,nu)`
`OutputVariables` (or `OV` or `Controlled` or `Output`)	(1 to p)-by-n_y dimensional array of OV weights	`1` (The default for output weights is the following: if $n_u \geq n_y$, all outputs are weighted with unit weight; if $n_u < n_y$, n_u outputs default to 1, with preference given to measured outputs, and the rest default to 0.)
`ECR`	Scalar weight on the slack variable ε used for constraint softening	`1e5*(max weight)`

Note If all `MVRate` weights are strictly positive, the resulting QP problem is strictly convex. If some MVRate weights are zero, the QP Hessian might be positive semidefinite. In order to keep the QP problem strictly convex, when the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12}, the quantity `10*sqrt(eps)` is added to each diagonal term. See "Cost Function".

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To do so, define the fields `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables` as cell arrays, each containing a single positive-semi-definite matrix of the appropriate size. Specifically, `OutputVariables` must be a cell array containing the n_y-by-n_y Q matrix, `ManipulatedVariables` must be a cell array containing the n_u-by-n_u R_u matrix, and `ManipulatedVariablesRate` must be a cell array containing the n_u-by-n_u $R_{\Delta u}$ matrix (see "Alternative Cost Function" and the `mpcweightsdemo` example). You can use diagonal weight matrices for one or more of these fields. If you omit a field, the MPC controller uses the defaults shown in the table above.

For example, you can specify off-diagonal weights, as follows

```
MPCobj.Weights.OutputVariables = {Q};
```

```
MPCobj.Weights.ManipulatedVariables = {Ru};
MPCobj.Weights.ManipulatedVariablesRate = {Rdu};
```

where `Q` = Q. `Ru` = R_u, and `Rdu` = $R_{\Delta u}$ are positive semidefinite matrices.

Note You cannot specify non-diagonal weights that vary at each prediction horizon step. The same `Q`, `Ru`, and `Rdu` weights apply at each step.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in "Controller State Estimation". It is a 1-D structure containing the following fields.

Models Used by MPC

Field Name	Content	Default
`Plant`	LTI model or identified linear model of the plant	No default
`Disturbance`	LTI model describing expected unmeasured input disturbances	[ ] (By default, input disturbances are expected to be integrated white noise. To model the signal, an integrator with dimensionless unity gain is added for each unmeasured input disturbance, unless the addition causes the controller to lose state observability. In that case, the disturbance is expected to be white noise, and so, a dimensionless unity gain is added to that channel instead.)
`Noise`	LTI model describing expected noise for output measurements	[ ] (By default, measurement noise is expected to be white noise with unit variance. To model the signal, a dimensionless unity gain is added for each measured channel.)
`Nominal`	Structure containing the state, input, and output	The default values of the fields are shown in the following table:

<table>
<tr><th>Field Name</th><th>Content</th><th>Default</th></tr>
<tr><td></td><td>values where <code>Model.Plant</code> is linearized</td><td>
<table>
<tr><th>Field</th><th>Description</th><th>Default</th></tr>
<tr><td>X</td><td>Plant state at operating point</td><td>0</td></tr>
<tr><td>U</td><td>Plant input at operating point, including manipulated variables, measured and unmeasured disturbances</td><td>0</td></tr>
<tr><td>Y</td><td>Plant output at operating point</td><td>0</td></tr>
<tr><td>DX</td><td>For continuous-time models, DX is the state derivative at operating point: DX=f(X,U). For discrete-time models, DX=x(k+1)-x(k)=f(X,U)-X.</td><td>0</td></tr>
</table>
</td></tr>
</table>

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See “MPC Modeling”.

Specify input and output signal types via the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, use the `setmpcsignals` command. Valid signal types are listed in the following tables.

Input Groups in Plant Model

Name (Abbreviations)	Value
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	Indices of manipulated variables in `Model.Plant`
`MeasuredDisturbances` (or `MD` or `Measured`)	Indices of measured disturbances in `Model.Plant`
`UnmeasuredDisturbances` (or `UD` or `Unmeasured`)	Indices of unmeasured disturbances in `Model.Plant`

Output Groups in Plant Model

Name (Abbreviations)	Value
`MeasuredOutputs` (or `MO` or `Measured`)	Indices of measured outputs in `Model.Plant`
`UnmeasuredOutputs` (or `UO` or `Unmeasured`)	Indices of unmeasured outputs in `Model.Plant`

By default, all `Model.Plant` inputs are manipulated variables, and all outputs are measured.

The structure `Nominal` contains the values (in engineering units) for states, inputs, outputs, and state derivatives/differences at the operating point where `Model.Plant` applies. This point is typically a linearization point. The fields are reported in the following table (see also “MPC Modeling”).

Nominal Values at Operating Point

Field	Description	Default
`X`	Plant state at operating point	0
`U`	Plant input at operating point, including manipulated variables, measured and unmeasured disturbances	0
`Y`	Plant output at operating point	0
`DX`	For continuous-time models, `DX` is the state derivative at operating point: DX=f(X,U). For discrete-time models, DX=$x(k+1)$-$x(k)$=f(X,U)-X.	0

Ts

Sample time of the MPC controller. By default, if `Model.Plant` is a discrete-time model, `Ts = Model.Plant.ts`. For continuous-time plant models, you must specify a controller `Ts`. Its measurement unit is inherited from `Model.Plant.TimeUnit`.

Optimizer

Parameters for the QP optimization. `Optimizer` is a structure with the fields reported in the following table.

Optimizer Properties

Field	Description	Default
`MaxIter`	Maximum number of iterations allowed in the QP solver	`'Default'`
`MinOutputECR`	Minimum value allowed for `OutputMinECR` and `OutputMaxECR`	`0`

`MaxIter` is either a positive scalar integer or the string `‘Default’`. If set to an integer value, QP solver stops when the iteration number reaches this value. If set to `‘Default’`, the MPC controller automatically determines the maximum iteration number based on the controller specifications.

`MinOutputECR` is a nonnegative scalar used to specify the minimum allowed ECR for output constraints. By default, `MinOutputECR` is 0, which means that output constraints are allowed to be hard. If a custom `MinOutputECR` is specified, when a value smaller than `MinOutputECR` is provided in the `OutputVariables.MinECR` or `OutputVariables.MaxECR` properties of MPC objects, a warning message is issued and the value is raised to `MinOutputECR` during computation.

PredictionHorizon

`PredictionHorizon` is the integer number of prediction horizon steps. The control interval, `Ts`, determines the duration of each step. The default value is 10 + maximum intervals of delay (if any).

ControlHorizon

`ControlHorizon` is either a number of free control moves, or a vector of blocking moves (see “Optimization Variables”). The default value is 2.

History

`History` stores the time the MPC controller was created (read only).

Notes

`Notes` stores text or comments as a cell array of strings.

UserData

Any additional data stored within the MPC controller object.

Construction and Initialization

To minimize computational overhead, Model Predictive Controller creation occurs in two phases. The first happens at *construction* when you invoke the `mpc` command, or when you change a controller property. Construction involves simple validity and consistency checks, such as signal dimensions and non-negativity of weights.

The second phase is *initialization*, which occurs when you use the object for the first time in a simulation or analytical procedure. Initialization computes all constant properties required for efficient numerical performance, such as matrices defining the optimal control problem and state estimator gains. Additional, diagnostic checks occur during initialization, such as verification that the controller states are observable.

By default, both phases display informative messages in the command window. You can turn these messages on or off using the `mpcverbosity` command.

MPC Simulation Options Object

The `mpcsimopt` object type contains various simulation options for simulating an MPC controller with the command `sim`. Its properties are listed in the following table.

MPC Simulation Options Properties

Property	Description
`PlantInitialState`	Initial state vector of the plant model generating the data.
`ControllerInitialState`	Initial condition of the MPC controller. This must be a valid `mpcstate` object. **Note** Nonzero values of `ControllerInitialState.LastMove` are only meaningful if there are constraints on the increments of the manipulated variables.
`UnmeasuredDisturbance`	Unmeasured disturbance signal entering the plant. An array with as many rows as simulation steps, and as many columns as unmeasured disturbances. Default: 0
`InputNoise`	Noise on manipulated variables. An array with as many rows as simulation steps, and as many columns as manipulated variables. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
`OutputNoise`	Noise on measured outputs. An array with as many rows as simulation steps, and as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size. Default: 0
`RefLookAhead`	Preview on reference signal (`'on'` or `'off'`). Default: `'off'`

Property	Description
MDLookAhead	Preview on measured disturbance signal ('on' or 'off').
Constraints	Use MPC constraints ('on' or 'off'). Default: 'on'
Model	Model used in simulation for generating the data. This property is useful for simulating the MPC controller under model mismatch. The LTI object specified in Model can be either a replacement for Model.Plant, or a structure with fields Plant and Nominal. By default, Model is equal to MPCobj.Model (no model mismatch). If Model is specified, then PlantInitialState refers to the initial state of Model.Plant and is defaulted to Model.Nominal.x. If Model.Nominal is empty, Model.Nominal.U and Model.Nominal.Y are inherited from MPCobj.Model.Nominal. Model.Nominal.X/DX is only inherited if both plants are state-space objects with the same state dimension.
StatusBar	Display the wait bar ('on' or 'off'). Default: 'off'
MVSignal	Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action). An array with as many rows as simulation steps, and as many columns as manipulated variables. Default: 0
OpenLoop	Perform open-loop simulation ('on' or 'off'). Default: 'off'

The property Model is useful for simulating an MPC controller with “model mismatch”, i.e., when the controller’s prediction model only approximates the true plant behavior (inevitable in practice).

By default, `Model` is equal to `MPCobj.Model` (no model mismatch). If `Model` is specified, then `PlantInitialState` refers to the initial state of `Model.Plant` and defaults to `Model.Nominal.x`.

If `Model.Nominal` is empty, `Model.Nominal.U` and `Model.Nominal.Y` are inherited from `MPCobj.Model.Nominal`. `Model.Nominal.X/DX` is only inherited if both plants are state-space objects with the same state dimension.

MPC State Object

The mpcstate object type contains the state of an MPC controller. Create the MPC state object using mpcstate. Its properties are as follows.

Property	Description
Plant	Vector of state estimates for the controller's plant model. Values are in engineering units and are absolute, i.e., they include state offsets. If the controller's plant model includes delays, the Plant field of the MPC state object includes states that model the delays. Therefore length(Plant) > order of undelayed controller plant model. Default: controller's Model.Nominal.X property.
Disturbance	Vector of unmeasured disturbance model state estimates. This comprises the states of the input disturbance model followed by the states of the output disturbances model. Disturbance models may be created by default. Use the getindistand getoutdistcommands to view the two disturbance model structures. Default: zero, or empty if there are no disturbance model states.
Noise	Vector of output measurement noise model state estimates. Default: zero, or empty if there are no noise model states.
LastMove	Vector of manipulated variables used in the previous control interval, $u(k–1)$. Values are absolute, i.e., they include manipulated variable offsets. Default: nominal values of the manipulated variables.
Covariance	n-by-n symmetrical covariance matrix for the controller state estimates, where n is the dimension of the extended controller state, i.e., the sum of the number states contained in the Plant, Disturbance, and Noise fields.

Property	Description
	Default: If the controller is employing default state estimation the default is the steady-state covariance computed according to the assumptions in "Controller State Estimation". See also the description of the `P` matrix in the Control System Toolbox `kalmd` command. If the controller is employing custom state estimation, this field is empty (not used).

Explicit MPC Controller Object

An explicit MPC object contains the explicit control law equivalent to the traditional (implicit) MPC controller object from which it derives. Use an explicit MPC controller to implement MPC in applications requiring very rapid computations, i.e., a short control interval. Use the `generateExplicitMPC` command to create the object. Its properties are as follows:

Properties

Property	Description
`MPC`	Traditional (implicit) controller object used to generate the explicit MPC controller. You create this MPC controller using is the `mpc` command. It is the first argument to `generateExplicitMPC` when you create the explicit MPC controller. See "MPC Controller Object" on page 3-2 or type `mpcprops` for details regarding the properties of the MPC controller.
`Range`	1-D structure containing the parameter bounds used to generate the explicit MPC controller. These determine the resulting controller's valid operating range. This property is automatically populated by the `range` input argument to `generateExplicitMPC` when you create the explicit MPC controller. See `generateExplicitRange` for details about this structure.
`OptimizationOptions`	1-D structure containing user-modifiable options used to generate the explicit MPC controller. This property is automatically populated by the `opt` argument to `generateExplicitMPC` when you create the explicit MPC controller. See `generateExplicitOptions` for details about this structure.

Property	Description
`PiecewiseAffineSolution`	n_r-dimensional structure, where n_r is the number of piecewise affine (PWA) regions required to represent the control law. The ith element contains the details needed to compute the optimal manipulated variables when the solution lies within the ith region. See “Implementation”.
`IsSimplified`	Logical switch indicating whether the explicit control law has been modified using the `simplify` command such that the explicit control law approximates the base (implicit) MPC controller. If the control law has not been modified, the explicit controller should reproduce the base controller’s behavior exactly, provided both operate within the bounds described by the `Range` property.